NTNU

Norwegian University of
Science and Technology

Towards Faster Development of Deep Learning Models Using Meta-Learning

Martin Baklid
Nils Barlaug

Master of Science in Computer Science
Submission date: July 2018
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Deep learning has, in relatively few years, improved significantly the performance of many machine learning applications. Even though its popularity has surged, it's not always easy to apply deep learning to a real-world problem. Developing a good deep learning model is a process that most likely will include several iterations of data collection, training and hyperparameter tuning. One big obstacle in this process is the hunger for data and compute power. Supervised learning often requires a massive amount of annotated examples, and training usually extends over hours or days. This makes the process of developing deep learning models very time and resource consuming. In this thesis we investigate if recent advances in few-shot learning can be used to speed up this process, and look specifically at object detection as an example. Such methods could potentially decrease both the necessary number of examples and the training time.

MAML (Finn et al. 2017) is a promising few-shot learning method based on meta-learning that optimizes the initial parameters of a model to be best possibly suited for fine-tuning. It's model-agnostic by nature and can in principle be applied to most deep learning models. But through extensive exploration we show that it's far from trivial to apply MAML to object detection on natural images. However, we are able to use a simpler method inspired by MAML, Reptile (Nichol et al. 2018b). We show that a model pretrained using Reptile can be fine-tuned considerably faster than a model pretrained normally on object detection, but surprisingly it does not enable using fewer examples. In addition, we show that Reptile is able to speed up the development of deep learning models in practice. This is done by building a proof of concept tool and use this to test some example use cases.

Sammendrag

Dyp læring har, på relativt få år, forbedret ytelsen betydelig for mange applikasjoner av maskinlæring. Selv om populariteten har eksplodert, er det ikke alltid enkelt å anvende dyp læring på reelle problemer. Å utvikle en god modell med dyp læring er en prosess som mest sannsynlig vil involvere flere iterasjoner av datainnsamling, trening og justering av hyperparametere. Et stort hinder i denne prosessen er apetitten på data og beregningskraft. Veiledet læring trenger ofte massive mengder annoterte eksempler, og trening strekker seg ofte over timer eller dager. Dette gjør utviklingsprosessen for modeller lagd med dyp læring veldig tids- og ressurskrevende. I denne oppgaven undersøker vi om nylige framskritt innen få-forsøks-læring kan bli brukt til å gjøre denne prosessen raskere, og ser spesifikt på objekt-detektering som et eksempel. Slike metoder kan potensielt redusere både det nødvendige antallet eksempler og tiden det tar å trene en modell.

MAML (Finn et al. 2017) er en lovende få-forsøks-lærings-metode basert på meta-læring som optimaliserer de initielle parameterne til en modell så de er best mulig egnet til finjustering. Den er modell-agnostisk i sin natur og kan i prinsippet bli anvendt på de fleste modeller innen dyp læring. Men gjennom omfattende utforskning viser vi at det er langt fra trivielt å anvende MAML til objekt-detektering på naturlige bilder. Derimot er vi i stand til å bruke en enklere metode som er inspirert av MAML, Reptile (Nichol et al. 2018b). Vi viser at en modell som er forhåndstrent ved hjelp av Reptile kan bli finjustert vesentlig raskere enn en modell som er forhåndstrent normalt på objekt-detektering, men overraskende nok så muliggjør den ikke å bruke færre eksempler. I tillegg viser vi at Reptile er i stand til å gjøre utviklingen av modeller med dyp læring raskere i praksis. Dette gjøres ved å lage et eksempelverktøy og så bruke dette til å teste noen eksempelbruksmønste.

Preface

Before starting the work on this thesis, we did our specialization project. We did preliminary work and explored the literature on few-shot learning and the possibility of applying it to object detection. The project focused on MAML (Finn et al. 2017), a novel few-shot learning approach based on meta-learning. To explore the characteristics of MAML, we did a comprehensive study of few-shot sinusoid regression.

Since this thesis is a continuation of the specialization project, and we cannot expect the reader to have read our earlier report, we adapt and include some parts of the text from this earlier work into this thesis. Note that this is common practice at our institution, but not commonly mentioned in theses. The large majority of the thesis is new (even though the list below seems long), but some of the more introductory parts are partially reused. We now list which parts are reused and to which degree.

- **Section 1.1 — Motivation**: Our motivation has not changed much, and so this text originates from the motivation in the specialization project report but has been reworked heavily.
- **Section 2.1 — A brief history of deep learning**: This text remains largely the same as for the specialization project report.
- **Section 2.2 — Fundamentals of deep learning**: Figures have been reworked to fit the format of this thesis, but the text remains mostly the same.
- **Section 2.3 — Convolutional Neural Networks**: While originating from the specialization project report, it has been heavily adapted and extended. For the most part, this content is new.
- **Section 2.4 — Few-shot learning and meta-learning**: The specific parts about MAML and Meta-SGD remain largely the same (mostly just adapted to new notation), but everything else is mainly new content.
- **Section 3.1 — Deep learning frameworks**: While our use of the PyTorch deep learning framework has increased in complexity since the specialization project, the considerations and background regarding deep learning frameworks are the same. Most of this section stays the same but has been reworked to reflect major updates in the PyTorch API.
- **Section 5.1 — Functional PyTorch**: The actual implementation of our extension to PyTorch to support our special use case has been extensively reworked, but the motivation behind it and the high-level explanation of it remains to a large degree the same.

We would like to thank our supervisor Magnus Lie Hetland for invalueable support and guidance. In addition, we would like to thank Anne C. Elster for access to hardware resources.

Contents

Chapter 1

Introduction

We will now first present our research motivation. Then we briefly discuss related work, before introducing our research questions and goals. Finally, we list our main contributions and outline the rest of this thesis.

1.1 Motivation

Deep learning has in few years become a major driver of innovative solutions. This is made possible by an increasing amount of available data, increasing computational resources and improved techniques for training deep neural networks. The key to the success of deep learning is in its ability to learn hierarchical features from massive amounts of relatively unprocessed data (LeCun et al. 2015) — for example, images, audio and text. Learned features from deep learning models have proven to be superior to hand engineered features across many tasks, being both more accurate and less resource demanding to develop.

Most breakthroughs come from supervised learning, and its ability to leverage vast amounts of annotated data. But for many tasks, there do not exist large annotated datasets — a significant obstacle for solving new tasks. Training a supervised machine learning model for a new task is usually an iterative process. Several rounds of data collection, annotation and model adjustment might be necessary to achieve the desired performance. A simplified view of this process is shown in Figure 1.1. In this thesis, we will refer to this as the *user–model feedback loop*. In each iteration of the loop, the user[1] adjusts the model and/or provide data, train the model, and then evaluate the model somehow. The evaluation is provided as feedback to the user, and depending on the user's decision a new iteration might be initiated.

[1] Note that *user* in this context need not be a single person. It could be (and often is) a team of individuals.

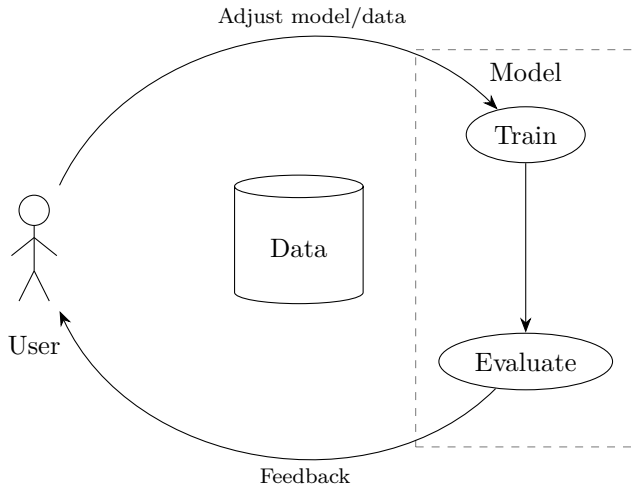


Figure 1.1: The process of developing a supervised machine learning model for a new task shown in a simplified manner. We call this the user–model feedback loop.

Usually, the most resource-demanding part in this process is the user. The user is responsible for model adjustment, data collection and annotation. All these tasks can be very time and resource demanding — and therefore expensive. This is especially true for specialized domains, like medical image analysis. Collecting images might depend on using expensive medical equipment and getting one or more doctors' opinion on every image. Thus budgets can limit the number of images you can collect and annotate. There might even be hard limits on the number of images available if they rely on patients with rare conditions. Getting a model to work on a new task can involve a significant amount of fiddling and tuning of the hyperparameters and model architecture, resulting in potentially many iterations. It's no surprise that if the training of the model takes a long time, the whole process will be very lengthy.

With the above in mind, we argue that speeding up the user–model feedback loop could potentially be significantly resource-saving and even enable new applications. If we ignore ways to improve the user's decisions, there are two ways to speed up the feedback loop:

1. **Decrease the model's need for examples**: If less data is needed, the user can spend less time collecting and annotating data. Annotating thousands of examples can easily take days or weeks for a single person.

2. **Decrease the time used to train and evaluate the model**: If the model can be trained faster, the user can get feedback earlier. Deep learning models are notoriously slow to train, with training times often ranging from hours to

days.

Traditionally, the user–model feedback loop is somewhat decoupled into distinct stages because each step usually takes at least a day, and other work is often done in parallel. This on-and-off workflow can in itself make the feedback loop slower. Ideally, the feedback loop would be so fast that it can be performed in one continuous user session. There exist interactive tools for annotating data and tools for evaluating models, but a single tool usually doesn't do both — at least not in the same session.

Humans have the ability to adapt to new tasks from very few examples. Since most real-world tasks don't come with countless examples of correct behavior (and never will), humans have no choice but to learn from few examples. We call the machine learning problem of learning from few examples *few-shot learning*. In the last couple of years, several methods for doing few-shot learning in a deep learning context has been proposed (see Chapter 2). While not achieving human-level accuracy or accuracy on a level suitable for most fully-automatic real-world applications, they still perform significantly better than random guessing. Low-accuracy methods are not necessarily useless, they might still be used to assist humans in exploring and annotating data in earlier iterations more efficiently — and then achieve higher accuracy in later iterations. These methods might also be able to train considerably faster than other methods. Therefore we argue that it would be interesting to see if recent advances in few-shot learning can be used to speed up the user–model feedback loop. Such methods have the potential to both decrease the model's need for examples and the training time.

Most new few-shot methods are primarily concerned with classification tasks — and focus especially on image classification. We think it would be more interesting to investigate problems that require more manual annotation labor — making gains in efficiency more valuable. One such problem, closely related to image classification but with more intensive annotation needs, is object detection. Having both a large appetite for annotations and large complex models with long training time, we argue object detection is a good candidate for a problem that needs a faster user–model feedback loop.

1.2 Related Work

We note that the most relevant work for this thesis will be covered in greater detail in Chapter 2.

Object detection is a central problem in computer vision, which has been worked on for many years. The goal is to classify, localize and decide the extent of objects in an image. Deep learning based methods have in recent years replaced methods based on handcrafted features as state of the art. We cover object detection extensively in Chapter 2.

The cost of acquiring annotated datasets have been acknowledged for a long time. Different approaches have been made to either reduce the need for annotations or the cost of annotating. Some approaches are quite general, while others are more problem- or model-specific.

Active learning (see Settles 2010, for an extensive review of earlier work) aims to choose the most valuable examples to annotate next in order to improve performance. The idea is that some new annotated examples might not add much beyond the annotated examples you already have, while others might be more informative and make a more significant difference when added to the training set. A common approach is to quantify how uncertain the model is on different unannotated examples and choose the examples the model is least certain about. Active learning has been used on object detection (Bietti 2012; Kao et al. 2018; Roy et al. 2016; Sivaraman & Trivedi 2014; Vijayanarasimhan & Grauman 2014), although more frequently on image classification (e.g. Joshi et al. 2009; Sener & Savarese 2018; Wang et al. 2017). Some methods rely on more traditional computer vision techniques, while other depend on deep learning. While these methods can reduce the number of annotations necessary, they still operate in the scale of thousands of examples — which in practice is still a quite slow user–model feedback loop. Many of the methods also require time-consuming training of the model before new requests for annotations are made.

Semi-supervised learning (see Zhu 2008, for an extensive review of earlier work) makes use of both annotated and unannotated examples. Even though some examples are not annotated, they may still be useful. Semi-supervised learning has not been widely applied to object detection. A more common related approach is *weakly supervised learning*, where the goal is to leverage incomplete or inaccurate annotations. Image-level annotations from humans or image classification models are often used since these are easier to obtain (Bazzani et al. 2016; Bilen et al. 2014; Oquab et al. 2015). Such methods reduce the labor cost because they rely on cheaper annotations, but they still need massive amounts of annotations — which can result in time consuming or expensive iterations in the user–model feedback loop for new or specialized domains. Work is being done to fuse few-shot learning

with weakly supervised learning (Keren et al. 2018), and might be promising in the future.

Few-shot learning is an extreme form of transfer learning where the model has already been trained on data from similar tasks. Recent methods have shown promising performance for regression, image classification and reinforcement learning tasks (see more in Chapter 2). These methods can be extremely valuable if they are able to learn from domains with lots of available annotated data and transfer efficiently to new or specialized domains.

Crowdsourcing has been used effectively to annotate large-scale datasets (Deng et al. 2009; Lin et al. 2014). The idea is to use volunteers or paid workers online to collaborate annotating large amounts of data. It's important to have clear instructions for workers and good routines for quality assurance. Quality is often controlled by letting users verify each other's annotations and by comparing annotations between users and to 'gold standard' annotations. While crowdsourcing might be a cost-effective method, it's often not feasible for datasets which require extensive domain knowledge, are regulated by strict privacy regulations or are considered industry secrets. If annotation demands heavy domain knowledge (e.g. need doctors or engineers), one cannot simply let arbitrary workers annotate. Strict privacy regulations might make it illegal and/or unethical to share data (e.g. medical records) with external persons. And businesses may not be willing to share their data with external personnel because they see their data as an important asset for which they need to control access.

Various steps have been taken to reduce the annotation effort with the help of specialized tools. Papadopoulos et al. (2017), for example, show that bounding boxes can be annotated faster by letting the user click on the right-, left-, bottom- and topmost point of an object. And there has been a considerable amount of work on easing the annotation of videos — often trying to incorporate and make use of motion so not every frame has to be annotated (Ali et al. 2011; Mihalcik & Doermann 2003; Vondrick et al. 2013; Yuen et al. 2009). All of these tools help speed up the annotation process, but they are not able to give any sort of feedback during a session on the impact your new annotations have on the model you want to train.

1.3 Research Questions and Goals

Both object detection and few-shot learning are complex topics. If we want to investigate the use of few-shot learning to speed up the user–model feedback loop for object detection, we do not have time to do extensive testing of all methods. Instead, we think it would be wise to single out a suitable few-shot learning method and try to adapt it to a suitable object detection method — trying to answer if it is possible to do few-shot object detection.

Research question 1. Can few-shot learning methods be used to do object detection on natural images?

Goal 1.1. Choose and adapt a few-shot learning method to object detection.

Goal 1.2. Evaluate performance of the method from Goal 1.1 on standard object detection datasets.

Given that few-shot object detection is possible, does it actually speed up the user–model feedback loop? While one could rely on only theoretical analyses, we argue it's important to consider the technical and practical feasibility. There might be bottlenecks and obstacles in an actual real-world application that are hard to foresee. Therefore we consider a proof of concept implementation to be a suitable way to test if the feedback loop is faster.

Research question 2. Does the method from Goal 1.1 enable a faster user–model feedback loop?

Goal 2.1. Make a proof of concept tool simulating the user–model feedback loop on object detection.

Goal 2.2. Use the proof of concept tool to measure if it's reasonable to expect a faster user–model feedback loop when using the method from Goal 1.1.

1.4 Contributions

We make the following contributions:

- We introduce a new simple taxonomy for meta-learners that enables discussion about meta-learning at a more general level.
- We give valuable insight into how difficult it is to apply MAML (Finn et al. 2017) to object detection on natural images — sharing our failed attempts.
- We show that it is possible to do few-shot object detection with Reptile (Nichol et al. 2018b) — even though it's unclear if the accuracy is any better when compared to standard pretraining.
- We show that Reptile enables us to adapt much quicker to a new object detection task than standard pretraining.
- We demonstrate that Reptile can speed up the user–model feedback loop in realistic use cases by implementing and running a proof of concept tool.

1.5 Outline

Chapter 2 — Background Theory: All necessary background theory are introduced. This includes deep learning in general as well as convolutional neural networks and few-shot learning. For each topic, we first give an overview before we go into greater detail about the most relevant methods for this thesis. Note that our choice of methods are motivated later.

Chapter 3 — Tools and Environment: Describes the tools and environment used in this thesis. Note that our choice of tools are motivated later.

Chapter 4 — Exploration: Lay out the path we followed to achieve Goal 1.1. Choice of methods used in this thesis will be motivated. Since a large part of the work we did was to explore how to get few-shot learning working for object detection, we have decided to present this work in a dedicated chapter. We will also give valuable insight into failed attempts.

Chapter 5 — Implementation: Presents the implementation of different solutions used throughout this thesis. We will only give a high-level overview and consider details to be outside the scope of this text. Choice of tools will be motivated when they are used.

Chapter 6 — Experimental Setup: We explain in detail which experiments we will do, and how we set up and execute them. This will provide all necessary details needed to understand the results in the next chapter.

Chapter 7 — Results: For each experiment, we present the results and then analyze and interpret them.

Chapter 8 — Discussion and Conclusion: With the results and analysis fresh in mind we conclude with respect to our research questions and goals. We also discuss consequences and the potential of the methods we have worked with, and finally look at possible future work.

Chapter 2

Background Theory

In this chapter, we introduce the theoretical material used in this thesis. It should provide the reader with enough information to understand the rest of the document. But not all topics will be covered in detail, and unfamiliar readers will have to consult the referenced sources for more information if they find it necessary.

We will open by explaining why deep learning has become so popular before we cover the fundamentals of deep learning. Extending the fundamentals, we cover Convolutional Neural Networks (CNNs) — a key ingredient for applying deep learning to computer vision tasks. More specifically we look at the building blocks needed to build CNNs and then delve into image classification and object detection. We end the chapter by giving an overview of few-shot learning and meta-learning, including a more detailed coverage of the meta-learning methods most relevant for this thesis.

2.1 A Brief History of Deep Learning

Deep learning has made its march into many domains, often becoming the new state of the art (LeCun et al. 2015). Eye-opening applications and record-breaking performance in standardized benchmarks are responsible for a massive amount of coverage in the press (He et al. 2015; Mnih et al. 2015; Silver et al. 2016). The hallmark of deep learning is its ability to learn the representations from raw data necessary to perform its task. It does so by learning multiple levels of representation, increasing in abstraction from raw data. Earlier machine learning methods had the problem that they were too dependent upon the representation of the data they got. These classical methods need handcrafted feature extractors, which can be extremely challenging or infeasible to engineer. Figuring out which low-level features one needs and how to extract them used to be a big industry, but deep learning is rapidly replacing these methods.

While the term *deep learning* is relatively new, its history (Schmidhuber 2015) has roots going many years back. This is because *deep learning* is simply referring to a neural network with many layers[1] — a rather recent trend.

McCulloch & Pitts (1943) introduced a simple model of neural networks but was not able to train them. Different ways of training the networks were proposed in the years following. Rosenblatt (1958) introduced the perceptron, basically a single neuron, and a way to train it. However, it remained an unsolved challenge how to effectively train multi-layered networks for many years until Rumelhart et al. (1985) introduced backpropagation — a structured way to propagate the influence on the output through neural networks with multiple layers. Despite some successful applications (e.g. LeCun et al. 1990) neural networks never saw the widespread use we see today because they had several obstacles making training hard. The networks demanded large amounts of data and computational resources. Also, it was quite tricky to get the training procedure to convergence in a reasonable time without getting stuck or overfit. G. E. Hinton et al. (2006) marks the start of a revived interest for neural networks. They showed that it was possible to train (some variant) of a deep neural network effectively. In the following years the field made progress for three major reasons:

1. **Improved training techniques and tools**: Making it easier to convergence quickly without getting stuck and generalize well more reliably. More specifically, better ways to deal with vanishing gradients (e.g. ReLU from Nair & G. E. Hinton 2010) and overfitting (e.g dropout from Srivastava et al. 2014).

2. **More data**: Making it possible to leverage neural networks ability to exploit huge amounts of raw data. The general growth of the web and sensor data, together with the construction of large-scale datasets (e.g. ImageNet Deng et al. 2009), have made it possible to obtain a growing amount of training data.

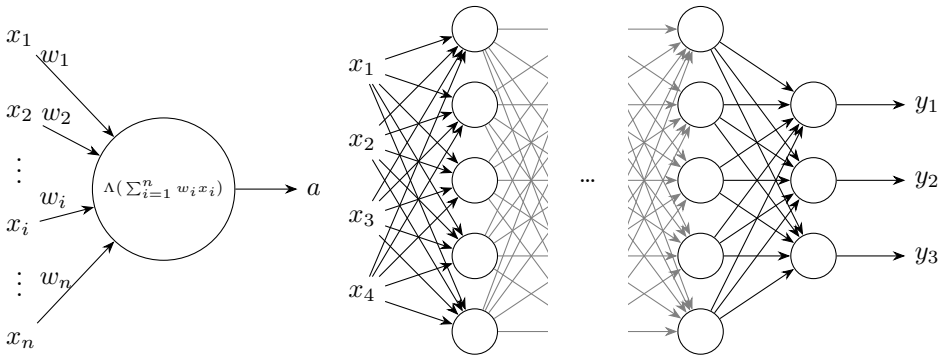
3. **More computational resources**: Making it possible to train big and deep networks. Mass produced GPUs have made it possible and cheap to obtain parallel computing hardware suitable for training neural networks.

All of the above led up Krizhevsky et al. (2012) winning the ImageNet Large Scale Visual Recognition Challenge in 2012, which became the start of mass adaptation of deep learning.

2.2 Fundamentals of Deep Learning

In this section, we will briefly introduce some of the main concepts of deep learning. A complete introduction is outside the scope of this thesis, but we refer the reader to

[1] There exists no strict requirement on the number of layers



(a) (b)

Figure 2.1: (a) Illustration of a neuron. (b) Illustration of a fully connected feedforward neural network.

Goodfellow et al. (2016) for a comprehensive introduction. We will limit ourselves to superficially discussing only standard supervised learning with feedforward networks, as that is most relevant for this thesis.

2.2.1 Goal

Let's first define our goal. We wish to approximate some unknown function[2] f^*. This function could, for example, be used to classify images. So if $\boldsymbol{x}$ is an image, $\boldsymbol{y} = f^*(\boldsymbol{x})$ will be some value representing the class of the image (e.g. dog, cat, etc. if one were classifying animals). Of course, we can't approximate this function without knowing anything about the task at hand. We therefore have some observed data $D = \{(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) | 1 \leq j \leq m\}$ where we know that $f^*(\boldsymbol{x}^{(j)}) \approx \boldsymbol{y}^{(j)}$ for all j. In other words we have some input-output examples for the function we want to approximate[3], and they may contain noise. The way we approximate f^* is by defining a function (also called a model) $f(\boldsymbol{x}; \boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$, and then try to find suitable parameters for this function using our data.

[2]It might seem restrictive to approximate a function when so many real-world examples are best described as a stochastic process without definite outcomes/answers. But this is entirely a question of which function to approximate. Functions can output probabilities, represent probability densities, etc.

[3]We may of course also have implicit domain knowledge about the task at hand guiding us.

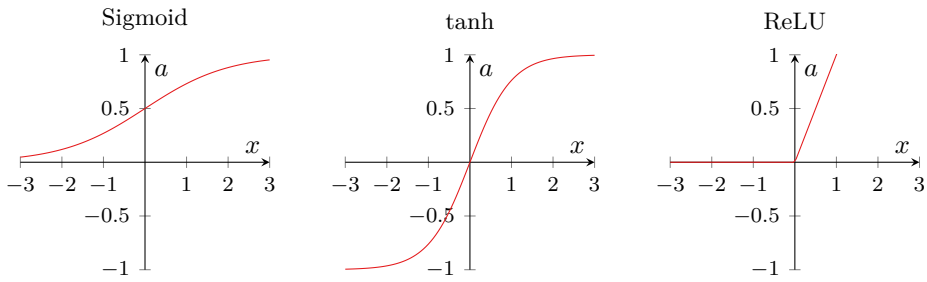


Figure 2.2: Plots showing the sigmoid, tanh and ReLU activation function.

2.2.2 Representation

How should this function $f_{\boldsymbol{\theta}}$ look like[4]? In theory, it could be anything. But in our context, this will be a feedforward neural network. A feedforward network is in essence just a function with many parameters. These networks can be assembled in arbitrarily ways with numerous building blocks and bell and whistles, but we will focus on the basics. The classical motivation used for neural networks is that they (very) loosely model the brain[5]. Our smallest building blocks are neurons — supposed to model brain neurons. A neuron takes multiple inputs, accumulate a weighted sum of the inputs and then performs some nonlinear activation function Λ on the accumulated sum. The resulting output is the activation value a of the neuron. See Figure 2.1a for an illustration. We use these neurons to build our feedforward network by organizing them in sequential layers. The inputs to a neuron are the activation values of all the neurons in the previous layer. Neurons of the first layer use $\boldsymbol{x}$ as inputs and the output of the neurons in the last layer are considered $\hat{\boldsymbol{y}}$. If we now regard $\boldsymbol{\theta}$ as a vector consisting of the weights of all the neurons in the network, we have a feedforward neural network interpreted as a function $f(\boldsymbol{x};\boldsymbol{\theta})$ that maps input $\boldsymbol{x}$ to output $\hat{\boldsymbol{y}} = f(\boldsymbol{x};\boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$. See Figure 2.1b for an illustration.

We just described what is known as a fully connected feedforward network. This is, of course, just a specific variant of feedforward networks. They may vary in numerous ways such as in activation functions and overall network architecture. Examples of typical activation functions are displayed in Figure 2.2. The most important aspect of such networks is that they can be represented as some kind of layered computational graph. We will see later why this is important. These layers can be said to give *depth* to the networks and the calculation they represent. Layers usually involve a big linear transformation where many values interact followed by a nonlinear per-neuron activation. Notice that such layers can often be expressed in vector notation. Let's take a fully connected layer as an example. If $\boldsymbol{W}^l$ is a matrix with the weights going into layer l and $\boldsymbol{a}^{l-1}$ is a vector with the activation values

[4]We use $f_{\boldsymbol{\theta}}$ as a shorthand, with meaning as in $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = f(\boldsymbol{x};\boldsymbol{\theta})$.
[5]Most will probably say they are in reality more "inspired by" than "model of".

from layer $l-1$, then the activation values for layer l become $\boldsymbol{a}^l = \Lambda(\boldsymbol{W}\boldsymbol{a}^{l-1})$ — given that $\boldsymbol{W}$ is defined correctly, and Λ performs elementwise activation.

2.2.3 Optimization

Given that we have selected a $f_{\boldsymbol{\theta}}$ that we think is suitable for the task at hand, we now need to figure out how we should set the parameters $\boldsymbol{\theta}$ to approximate f^*. To do this, we first define some cost[6] function $\mathcal{L}(f_{\boldsymbol{\theta}}, D)$, quantifying how "bad" $f_{\boldsymbol{\theta}}$ approximates f^* over the data D with parameters $\boldsymbol{\theta}$. Then we start with some (random) initial $\boldsymbol{\theta}$ and do gradient descent over $\boldsymbol{\theta}$ with respect to the cost function $\mathcal{L}(f_{\boldsymbol{\theta}}, D)$.

The cost function is typically defined as an average over the per example losses for all examples $(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) \in D$:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{|D|} \sum_j L(f_{\boldsymbol{\theta}}, (\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}))$$

where L is the per example loss. Notice that evaluating $L(f_{\boldsymbol{\theta}}, (\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}))$ usually involves evaluating $\hat{\boldsymbol{y}}^{(j)} = f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(j)})$ and then compare the output $\hat{\boldsymbol{y}}^{(j)}$ of the network to the target $\boldsymbol{y}^{(j)}$, i.e. $L(f_{\boldsymbol{\theta}}, (\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)})) = \widetilde{L}(\boldsymbol{y}^{(j)}, \hat{\boldsymbol{y}}^{(j)})$.

Gradient descent is, in its essence, doing

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(f_{\boldsymbol{\theta}}, D)$$

for many timesteps t, where α is the learning rate. As one can see, we need the gradient of the cost function $\mathcal{L}(f_{\boldsymbol{\theta}}, D)$ with respect to the parameters $\boldsymbol{\theta}$, and not the cost function itself. So for each gradient descent iteration, we would like to evaluate:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}, D) = \frac{1}{|D|} \sum_j \nabla_{\boldsymbol{\theta}} L(f_{\boldsymbol{\theta}}, (\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}))$$

This becomes too computationally heavy when we have a lot of data (as one usually has in deep learning) since we in practice have to evaluate and differentiate $\hat{\boldsymbol{y}}^{(j)} = f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(j)})$ to find $\nabla_{\boldsymbol{\theta}} L(f_{\boldsymbol{\theta}}, (\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}))$. Remember that $f_{\boldsymbol{\theta}}$ is actually a (large multilayered) neural network. So to make gradient descent more tractable, we use only a randomly sampled fix sized subset $B \subset D$ of the examples to approximate $\nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}, D)$:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}, D) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}, B)$$

This is what's called Stochastic Gradient Descent (SGD). B is referred to as a minibatch (or just batch) and $|B|$ is the (mini)batch size. In practice, one does not use standard SGD but rather one of its extensions or derivatives[7].

[6] Cost and loss are used interchangeably

[7] Perhaps most notable SGD with momentum (Rumelhart et al. 1986), RMSProp (Tieleman & G. Hinton 2012) and Adam (Kingma & Ba 2015)

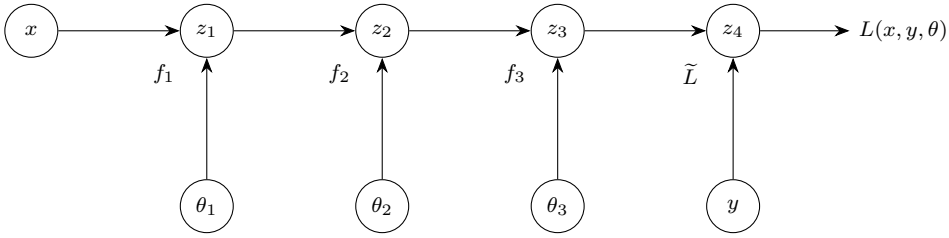


Figure 2.3: Illustration of the computational graph for the loss function in Equation 2.1.

2.2.4 Backpropagation

Even though sampling the examples and only approximating the true $\nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}, D)$ reduces the number of gradients we need to calculate, we still need to calculate some of these expensive gradients. Remember that our network could be represented as a computational graph. Since $\widetilde{L}$ is only a calculation based upon the output of the network and some target value, the graph can be extended to output the loss instead of the network output. We then have a computational graph that takes $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{\theta}$ as input and output the loss. The task of finding loss gradients is now reduced to finding gradients of one (or more) nodes in a computational graph with respect to other nodes. Backpropagation performs this calculation in an efficient manner. It's a procedure that applies the chain rule of differentiation using dynamic programming. We will illustrate this with an example.

Assume that all values are scalars[8] and that we have a loss function that can be expressed as a function composition:

$$L(f_{\boldsymbol{\theta}}, (x, y)) = \widetilde{L}(y, \hat{y}) = \widetilde{L}(y, f_{\boldsymbol{\theta}}(x)) = \widetilde{L}(y, f_3(f_2(f_1(x; \theta_1); \theta_2); \theta_3)) \tag{2.1}$$

And where intermediate values are named:

$$\begin{aligned} z_1 &= f_1(x) \\ z_2 &= f_2(z_1) \\ z_3 &= f_3(z_2) \\ z_4 &= \widetilde{L}(y, z_3) \end{aligned}$$

[8]The example is only for scalars, but the same principle applies to arbitrarily tensors.

We can then represent $L(f_{\boldsymbol{\theta}}, (x, y))$ as the computational graph shown in Figure 2.3. The question is now how we find the gradients $\frac{\partial L}{\partial \theta_1}$, $\frac{\partial L}{\partial \theta_2}$ and $\frac{\partial L}{\partial \theta_3}$. If we naively apply the chain rule, we get (2.2):

$$\begin{aligned}
\frac{\partial L}{\partial \theta_3} &= \frac{\partial z_4}{\partial \theta_3} = \frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial \theta_3} \\
\frac{\partial L}{\partial \theta_2} &= \frac{\partial z_4}{\partial \theta_2} = \frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial \theta_2} \\
\frac{\partial L}{\partial \theta_1} &= \frac{\partial z_4}{\partial \theta_1} = \frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial \theta_1}
\end{aligned} \tag{2.2}$$

$$\begin{aligned}
\frac{\partial z_4}{\partial z_4} &= 1 \\
\frac{\partial z_4}{\partial z_3} &= \frac{\partial z_4}{\partial z_4}\frac{\partial z_4}{\partial z_3} \\
\frac{\partial z_4}{\partial \theta_3} &= \frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial \theta_3} \\
\frac{\partial z_4}{\partial z_2} &= \frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2} \\
\frac{\partial z_4}{\partial \theta_2} &= \frac{\partial z_4}{\partial z_2}\frac{\partial z_2}{\partial \theta_2} \\
\frac{\partial z_4}{\partial z_1} &= \frac{\partial z_4}{\partial z_2}\frac{\partial z_2}{\partial z_1} \\
\frac{\partial z_4}{\partial \theta_1} &= \frac{\partial z_4}{\partial z_1}\frac{\partial z_1}{\partial \theta_1}
\end{aligned} \tag{2.3}$$

Here we see how the calculation lends itself easily to dynamic programming. Sub-calculations will be repeated more and more as the computational graph gets deeper. If we start at the output, we can work our way backward through the graph, building the gradients with the chain rule along the way — as seen in Equation 2.3 and illustrated using the computational graph in Figure 2.4. This is, in essence, the backpropagation algorithm.

The insight from this example is that we can find the gradient of the output of a computational graph with respect to internal nodes in the graph by doing only local calculations. To find the gradient of the graph output with respect to a node, we only need to know the gradient of the graph output with respect to its predecessor and the gradient of its predecessors with respect to the node itself — as illustrated in Figure 2.5. To find the gradient of the loss with respect to all parameters, one simply traverse the computational graph in reverse topological order applying the chain rule. The base case is that the gradient of the graph output with respect to itself is 1. In our example, the computational graph was a tree, but backpropagation will work for arbitrarily computational graphs (they will always be directed and acyclic). Also, the algorithm is not limited to working with scalars but extend easily to tensors of one or more dimensions.

The way backpropagation just was shown only involved symbolic values. When it's performed in practice, these will (of course) all be numerical values. The gradients

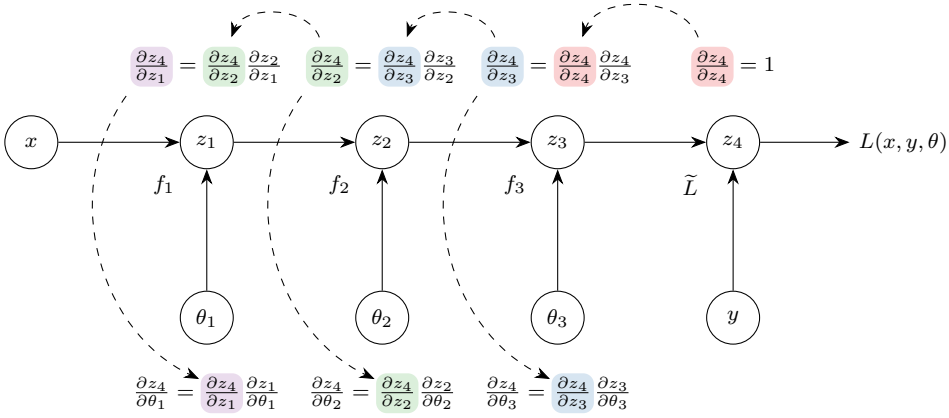


Figure 2.4: Illustration showing backpropagation performed on the computational graph in Figure 2.3 for the loss in Equation 2.1.

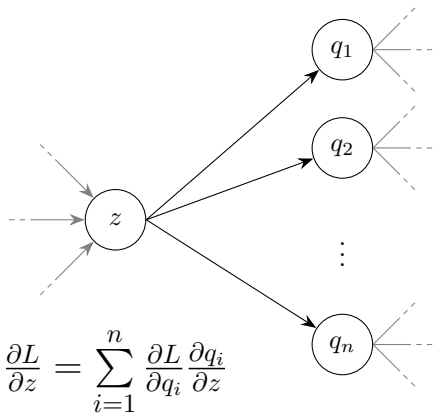


Figure 2.5: Illustration of the local computation necessary to perform dynamic programming in backpropagation.

calculated will depend on the input-output pair $(\boldsymbol{x}, \boldsymbol{y})$ and the parameters $\boldsymbol{\theta}$. All the local gradients will be expressions involving node values. Since we need the value of the different nodes in the graph, we actually have to evaluate the whole computational graph before doing backpropagation. This is called the forward pass. Backpropagation is then appropriately named the backward pass.

2.2.5 Generalization

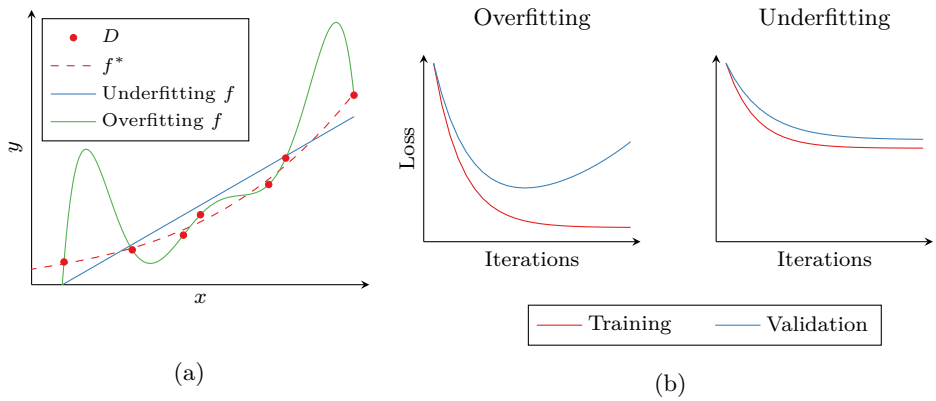
Let's assume that we are able to use SGD with backpropagation to optimize the parameters in large neural networks[9]. Then we are able to efficiently choose parameters $\boldsymbol{\theta}$ such that $f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(j)}) \approx \boldsymbol{y}^{(j)} \approx f^*(\boldsymbol{x}^{(j)})$ for all observed data points $(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) \in D$. However, just because $f_{\boldsymbol{\theta}}$ approximates f^* well on observed data doesn't necessarily mean it will on unobserved data from the true data distribution. Being able to approximate unobserved values from the true data distribution with the help of only a finite sample is what's known as generalizing. There is an underlying issue when we try to generalize. We posed the problem of approximating f^* as the problem of optimizing some parameterized function $f_{\boldsymbol{\theta}}$ to fit some (potentially noisy) samples of input-output examples. This will not in general lead to good generalization. Several things can go wrong, but there are mainly two of interest to us (both are illustrated in Figure 2.6a):

- **Underfitting**: When it should be possible to perform better on observed data and unobserved data at the same time. This could be a result of the model not being expressive enough or because of problems with the training procedure.
- **Overfitting**: When it should be possible to trade off worse performance on observed data with better performance on unobserved data. This can occur for many reasons. The underlying problem is, as mentioned, that we have posed generalization as optimization. In essence, we are fitting noise. Either sampling noise — so that the sampling gives a wrong impression on the overall data trend. Or noise in measurement — so that we mistake noise for actual data trends.

In deep learning, the problem of overfitting is by far the most dominant.

How do we know if we are underfitting or overfitting? This is in general hard to know, but one can take action to try to hypothesize what is going on. One useful metric to know is how good the model performs on the true unobserved data distribution. Unfortunately, we do not have/know this distribution. So instead we can get a (possibly biased) sample of the unobserved data by splitting our data in

[9] Optimizing large neural networks with SGD can be very tricky, and many problems may arise. We do not discuss details of how to get things working in practice, even though this a big part of the deep learning field. We recommend looking up other sources — e.g. Goodfellow et al. (2016).



(a) (b)

Figure 2.6: (a) Plot illustrating overfitting and underfitting. (b) Illustration of typical learning curves for models that overfit and underfit.

training and validation data[10]. We only use the training data when doing SGD, and approximate the performance on unobserved data with the validation data. It's especially informative to plot the loss on training and validation data during training — showing the progress as the number of SGD iterations increase. Figure 2.6b show one typical example of overfitting and one typical for underfitting. Overfitting is often characterized by the training loss distancing itself from the validation loss. The validation loss might even go up. Underfitting is often discovered by noting that the training (and validation) loss is unacceptably high.

One thing is detecting if the model is overfitting or underfitting, but how does one avoid/fix it? Underfitting can usually be fixed by increasing the model size, making it more expressive. In reality, the struggle seems to be that of overfitting. One might, of course, get more data to make it less likely to overfit, but this is not always possible/feasible. Moreover one could try to make the model smaller, but this might hurt expressiveness in a way that makes it underfit. There exist many ways to combat and balance overfitting, and it remains an active field of research. The techniques that are made to help generalization is called regularization techniques. We will briefly mention some of the most prominent techniques. The reader is encouraged to look up other sources or refer to Goodfellow et al. (2016).

- **Parameter norm penalty**: Penalize high magnitude parameters in the loss function. Popular variants include L1 and L2 regularization — the latter also known as *weight decay*.

[10]In reality, one almost always split the data in a training, validation and testing part. The model is trained on the training data, generalization is monitored with validation data and the final test of how the model does on unobserved data is done with the test data. Since one often do decisions about hyperparameters based on validation performance this data can't be regarded as unobserved, and we need the test data to get a reliable sample of the performance on unobserved data.

- **Early stopping**: Monitor performance on the validation set, stop training when the performance hasn't improved in a while and choose the parameters for when the validation loss was at its lowest.
- **Ensemble methods**: Train (partially) independent models and combine their results (average, majority vote, etc.). Independence can be achieved by using different training data, different random initialization or different network architectures.
- **Dropout** (Srivastava et al. 2014): Randomly set neurons to zero during training. Can be seen as an extreme form of ensemble method — training a different network each iteration. It forces the network to rely on multiple information sources, making it less likely to fit noise.
- **Data augmentation**: Synthesize more data by augmenting the already provided training data in ways that do not change the desired output.
- **Transfer learning**: Train on a different task with other data first, and then reuse the trained parameters to fine-tune the model for the task at hand.
- **Parameter sharing**: Let the same parameter be used several places in the network. The most famous example is CNNs — introduced in Section 2.3.

2.3 Convolutional Neural Networks

Computer vision is one of the areas where deep learning has been applied most successfully. A success that can be largely attributed to Convolutional Neural Networks (CNNs). These are deep learning networks that perform one or more convolutions. The convolution operation itself is not novel in the domain of computer vision. It has a long history in image processing and computer vision and has been used to perform tasks such as smoothing, sharpening and edge detection. But while classical techniques usually use relatively few handcrafted convolution kernels, CNNs use many learned kernels in a hierarchy of layers. The earlier layers detect low-level features which the later layers combine into more and more complex features. It is this hierarchical processing which gives CNNs the ability to learn high-level tasks, such as classifying images using raw images as its input (LeCun et al. 2015). CNNs have taken some inspiration from biology and the hierarchical processing is often compared to how the visual cortex in the brain works.

While computer vision is concerned with a wide variety of tasks, our focus is object detection. To understand how object detection is done we first have to take a quick look at image classification. So in this section, we will first explain the building blocks of CNNs. Then we take a look at how image classification is done, before we dive into object detection.

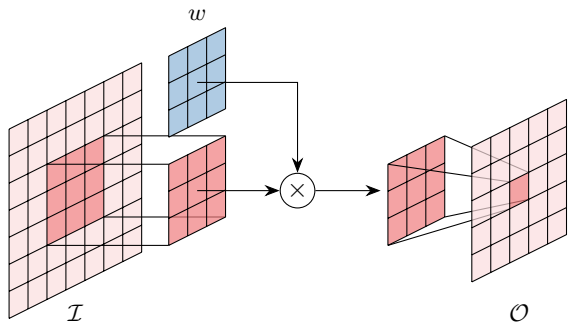


Figure 2.7: Illustration of a 3x3 convolution on a 7x7 input — showing the computation of $\mathcal{O}(2,2)$. $\mathcal{I}$ is the input image, w is the kernel and $\mathcal{O}$ is the result.

2.3.1 Building Blocks

The main building block of a CNN is the convolution operation[11]. It's not limited to two-dimensional domains or images specifically, but we will explain it in the context of images. The operation slides a kernel over the image performing a dot product between the kernel and the image at each spatial location:

$$\mathcal{O}(i,j) = (\mathcal{I} * w)(i,j) = \sum_n \sum_m \mathcal{I}(i+n, j+m) w(n,m)$$

where $\mathcal{I}$ is the input image and w is the kernel matrix. This is also illustrated in Figure 2.7. For classical computer vision tasks, the kernels are handcrafted to detect different features in the image, e.g. edges, and the output $\mathcal{O}$ from the convolution operation is a map that tells where in the input image a particular feature exists. In CNNs the kernels are treated as parameters — i.e. $w \subseteq \boldsymbol{\theta}$[12]. A convolution operation is usually followed by the addition of bias b and an elementwise activation function Λ. This is similar to fully connected layers, having both weight and bias as parameters and then sent through an activation function. Let $+^b(z) = z + b$ be the function which adds the bias. We call $\Lambda \circ +^b \circ \mathcal{O}$ a *convolutional layer* and the output of a convolutional layer a *feature map*[13]. A CNN is then a neural network that uses one or more convolutional layers, and may be structured in arbitrary ways and incorporate other types of layers. In deep learning, of course, one usually stack many convolutional layers. The input to the first layer is the image itself, while the other layers receive the feature map from the layer before.

The convolution operation incorporates strong prior knowledge about images: local-

[11] Strictly speaking, usually we actually use what's called cross-correlation, but this is not of any practical importance.

[12] $w \subseteq \boldsymbol{\theta}$ is not well defined since w is a matrix and $\boldsymbol{\theta}$ a vector, but we define it to mean that w is a subrange of $\boldsymbol{\theta}$.

[13] The use of these two terms vary in the literature and do not have an exact agreed upon definition.

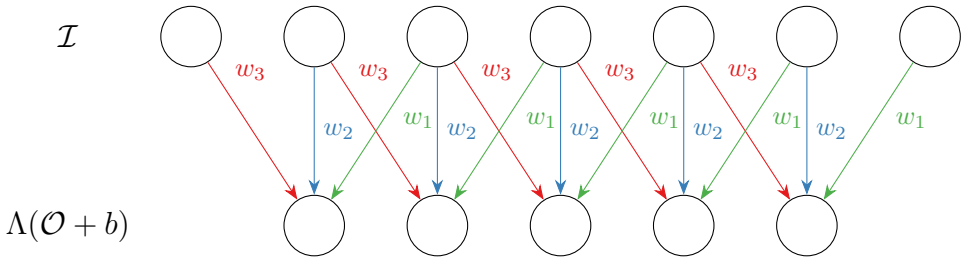


Figure 2.8: Illustration of a one-dimensional convolutional layer with a kernel of size 3.

ity and translation invariance. To easier appreciate this, let's look at one-dimensional convolutional layers and compare this to fully connected layers. Having only one dimension, the convolution operation simply becomes

$$\mathcal{O}(i) = (I * w)(i) = \sum_n \mathcal{I}(i+n)w(n)$$

Figure 2.8 illustrates this operation in the form of neurons — as discussed in Section 2.2. We can contrast this with fully connected layers by looking at the illustration in Figure 2.1b. The most noticeable difference between convolutional layers and fully connected layers is that the connections are sparse, i.e. a node in one layer is connected to some, but not all, nodes in the next layer. Not only are the neurons sparsely connected, but they are also locally connected. We are in a way incorporating prior knowledge of the spatial structure of images into the network architecture. Features that lie spatially close in an image are more likely to influence each other. Sparse connections lead to much fewer parameters in the network compared to fully connected layers, giving a strong regularizing effect since it's harder to overfit with fewer parameters.

Also notice that the weights are reused to compute the values of each of the nodes in $\mathcal{O}$ — this is the equivalent to sliding a kernel over multiple locations in the input. Sharing weights also have a regularizing effect since there are fewer parameters and the kernel is forced to learn weights that work well across all spatial positions in the input. This weight sharing incorporates our prior knowledge of translation invariance in images into the network architecture. A cat is a cat no matter where in the image it appears.

The observant reader may have noticed that the output of a convolution operation is smaller than its input. As long as the kernel is bigger than 1x1, we will necessarily get this effect because values in the output need a certain neighborhood of values in the input. This is not always a desirable effect. So to avoid it we usually pad the input to mimic a larger spatial extent. This is illustrated in Figure 2.9. If we pad correctly, we can make sure the output has the same size as the input. There exist

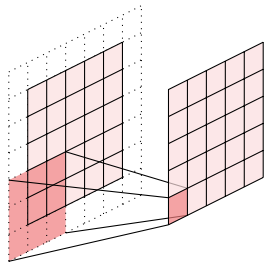
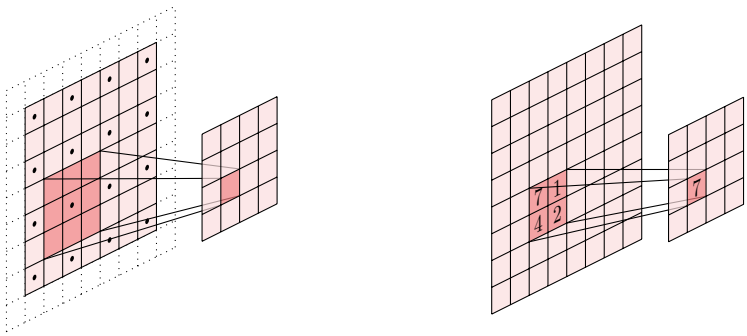


Figure 2.9: Illustration of padding in convolution. This show how padding can mitigate the problem of different sized input and output.



(a) (b)

Figure 2.10: (a) Illustration of a strided convolution with stride 2 on a 7x7 feature map with padding, resulting in a 4x4 feature map. The dots show the kernel center for every spatial location where the kernel is used. Notice that without padding the resulting feature map would be 3x3. (b) Illustration of 2x2 max pooling with stride 2 performed on a 8x8 feature map, resulting in a 4x4 feature map. This is an efficient way to scale down feature maps — not requiring any weights.

many schemes for what values to pad with, but the most common is to just use zeros (which is usually the mean value).

Sometimes, it may actually be of interest to scale down feature maps. Either to reduce the computational burden of the following layers or to achieve a more compact representation. This is usually done by either strided pooling or strided convolutions — the latter becoming more and more popular. Strided convolutions are like regular convolutions, but the kernel is applied to a lower resolution grid over the input:

$$\mathcal{O}_s(i,j) = (\mathcal{I} * w)(si, sj) = \sum_n \sum_m \mathcal{I}(i+n, j+m) w(n,m)$$

where s is the *stride*. A convolution with stride 2 is illustrated in Figure 2.10a. Pooling is a reduction function without parameters that is applied spatially across the

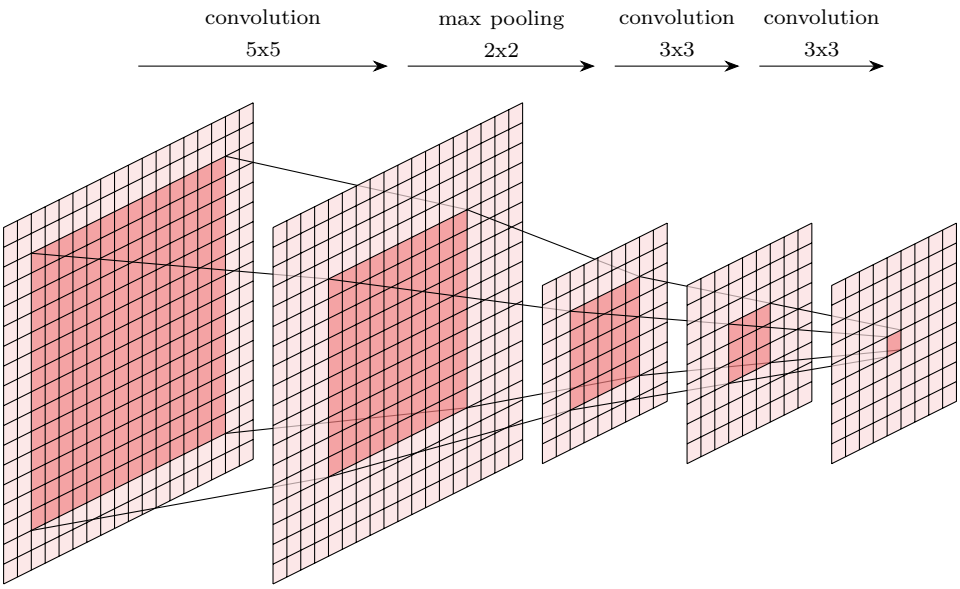


Figure 2.11: Illustration of the receptive field for the output of four layers. First a 5x5 convolutional layer, then a 2x2 max pooling layer with stride 2 and ending with two 3x3 convolutional layers. The input is a 22x22 image, and the output is a 9x9 feature map. All convolutional layers use padding. Notice the significant effect strided max pooling has on the receptive field.

whole feature map. The most common types of pooling are max and average pooling. 2x2 max pooling with stride 2 is illustrated in Figure 2.10b. The advantage with strided convolution over strided pooling is that the network may learn specialized downsampling schemes, while the advantage of pooling is its efficiency and the lack of parameters.

Other times it may be of interest to scale up feature maps, as we will see when we look at RetinaNet (Lin et al. 2017b). This is normally done by either *transposed convolutions* or *interpolation* techniques. Interpolation techniques are equivalent to traditional interpolation techniques used to upsample images. And we will not look closer at transposed convolutions since they are not relevant for us. The main distinction between these two techniques are as with strided convolutions and pooling — the first one is parameterized, while the latter one is not.

To understand the content of an image it won't suffice with small patches — you need to see larger parts of the image. So while a single convolutional layer might not be able to give valuable insight to an image, stacking multiple layers will enable the feature maps to base their individual values on a larger and larger part of the image. We call the area of the original input image affecting a value in a feature map the

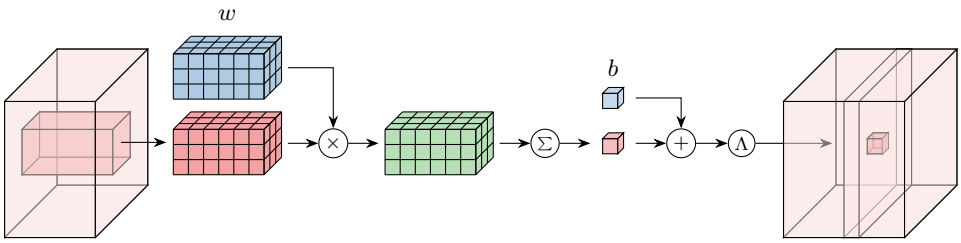


Figure 2.12: Illustration of a complete convolutional layer with multiple feature maps, bias and activation function. Note that there are separate kernel weights and biases for each output feature map. The kernel has to be a volume in order to slide over the input feature volume.

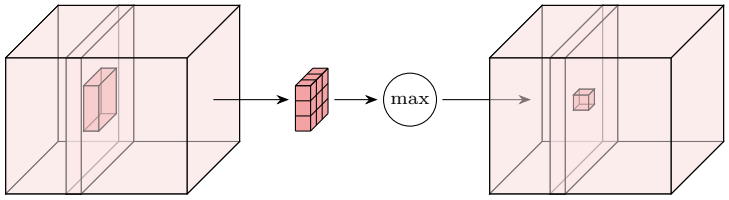


Figure 2.13: Illustration of max pooling applied to a feature volume. One simply applies max pooling on each of the feature maps in the input feature volume.

receptive field of that feature map. This is illustrated in Figure 2.11. It's important to take this into consideration when designing a CNN architecture. This will be clear when we look at object detection in Section 2.3.3. It might be unreasonably hard to detect a large horse when you can only see small patches of it without spatial relationship.

So far we have provided a simplified view of convolutions and pooling. In reality an image is not represented simply by a two-dimensional matrix, but rather as several matrices, because it contains several channels. It's common that digital photographies have three channels — a red, green and blue color channel. In addition we won't normally produce only one feature map per layer. We produce several feature maps, and thus have several kernels. The number of feature maps/kernels is referred to as the number of *filters* or *channels*. The input image, kernels and layer output can all be thought of as three-dimensional tensors or volumes. We sometimes call multiple feature maps used as input and output of convolutional layers for *feature volumes*. Extending convolution and pooling to suit this is trivial, and is illustrated in Figure 2.12 and 2.13.

These are the most central building blocks of CNNs, which have been successfully applied to many computer vision tasks. Common for convolution and pooling (with their different variants) is that they lend themselves well for GPU computation.

They are highly data parallel — the same operation should be executed on all a lot of data. So although these operations are computationally expensive, they are still feasible because they can be efficiently executed on relatively cheap consumer grade graphics cards. We will now look at how one can apply these building blocks to image classification, before we dive into object detection.

2.3.2 Image Classification

In image classification, the goal is to classify images as one of the multiple predetermined classes. Performance is measured in accuracy, i.e. number of correctly classified images divided by the total number of images. Sometimes an image is considered correctly classified if the correct class are among top k predictions.

The ubiquitous ILSVRC[14] (Deng et al. 2009) image classification dataset provides a very large number of training examples and act as a standard benchmark. Making the dataset suitable for both testing new methods and comparing them to others. It contains 14 million natural images annotated as one of 1000 object classes. The performance on this standard benchmark has increased dramatically in the last few years with the help of CNN architectures. There are of course many other datasets for different purposes and domains, ranging from the standard toy dataset MNIST (LeCun n.d.) to difficult large-scale datasets.

So how can we do image classification with a CNN? We can produce rich feature maps with convolutions and pooling, but how do we output predictions? The standard solution is to feed the feature volume into one or more fully connected layers and then softmax. The last fully connected layer outputs a vector with length corresponding to the number of classes, and the softmax operation normalizes this vector. Softmax is defined as

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

and can be considered a continuous argmax — it suppresses all but the highest value and provides a continuous one-hot vector picking out the highest value. Since $\sum_i \text{softmax}(\boldsymbol{x})_i = 1$ one can interpret the output vector as a probability distribution over classes, telling how certain the model is. So cross entropy is often used as loss function.

Let's take a look at some popular architectures, more specifically VGG16 (Simonyan & Zisserman 2014) and ResNet34 (He et al. 2016). They will be relevant when we discuss object detection. Both have a similar structure, as can be seen in Figure 2.14. Stacking convolutional layers with occasional strided convolutions or pooling to scale down the feature maps, and then ending with one or more fully connected layers and softmax. The number of feature maps is increased as their dimensions

[14] Also just called ImageNet.

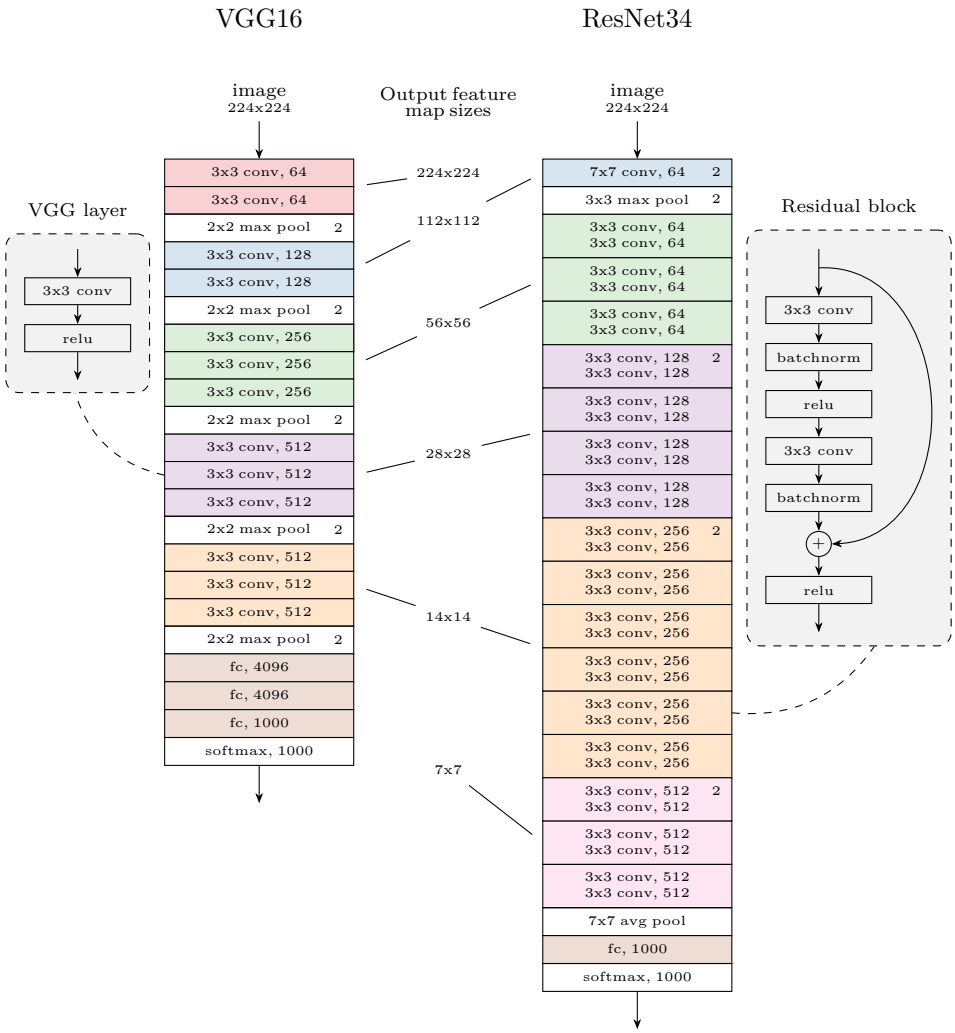


Figure 2.14: The architecture of VGG16 and ResNet34. For each convolution (conv) the kernel size and the number of kernels are indicated. Stride is specified in the margin. Fully connected layers (fc) are denoted by the output vector size. In the middle we can see the size of the feature maps produced by the different layers. VGG is shown as a stack of standard convolutional layers while ResNet is shown as residual blocks. Notice the characteristic skip-connection in the residual block — this is the main selling point of ResNet. The figure is inspired by He et al. (2016).

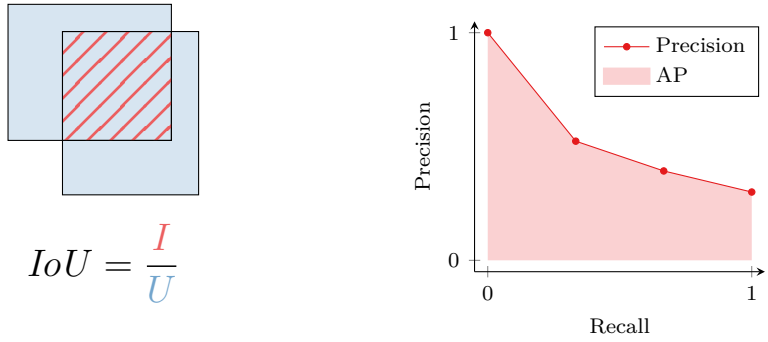
Figure 2.15: Illustration of object detection. One wants to localize and classify all objects of some set of predetermined classes.

are reduced. Notice that all images are resized to a fixed size (in this case 224x224) before processed by the network.

VGG stacks only standard 3x3 convolutional layers, consisting of a convolution followed by the ReLU activation function. Scaling down is done by strided max pooling. A considerable amount of weights and computational burden in VGG are found in the last fully connected layers. The second fully connected layer alone has $4096 \times 4096 = 16\,777\,216$ weights. Other variants than VGG16 with different depth are presented by the authors[15], but these only differ in the number of convolutional layers.

ResNet's main building block is the residual block. He et al. (2016) hypothesize that adding skip connections over layers will make it easier to optimize. This way it's easier for the net to achieve identity mapping and gradients may reach deep layers more undisturbed. Scaling down is done mainly by strided convolutions instead of max pooling, only one fully connected layer is used, and batch normalization (Ioffe & Szegedy 2015) is heavily utilized. There are other variants of ResNet that differ in depth[16], but they all share the same underlying structure — differing only in the number of residual blocks and slightly how the residual blocks are structured inside.

This is of course just a brief overview and there are lot more details that could have been outlined (e.g. ResNet's convolutions doesn't use bias because they are followed by batch normalization). And there are of course more variants of VGG and ResNet, as well as other architectures. But this will be sufficient to understand the rest of this thesis.



(a) (b)

Figure 2.16: (a) Illustration of Intersection over Union (IoU). (b) Illustration of Average Precision (AP) using precision-recall curve.

2.3.3 Object Detection

Object detection is the task of localizing and classifying objects in an image — as illustrated in Figure 2.15. To correctly detect an object one has to predict the correct class and a bounding box indicating the outline of the object. There can be arbitrarily many (or none) objects in the image, and there may exist several objects of the same class.

The first problem faced in object detection is how to measure performance. How well does a model perform object detection? Two issues arise:

1. When is a bounding box correct? How close should it be to ground truth? We should obviously allow a certain degree of slack.
2. How do we trade off precision and recall? We might not detect every object and we might false positively detect objects. It's desirable to have a measure that gives partial credit, instead of simply deeming the result either completely correct or incorrect.

Both issues can of course be solved in numerous ways, but the community seems to have converged on a solution.

The first issue is solved with the use of Intersection over Union (IoU). This is a measure for comparing bounding boxes, and is simply the area of the intersection divided by the area of the union of the two boxes — as illustrated in Figure 2.16a. An IoU of 1 is a complete overlap between the boxes while an IoU of 0 means there is no overlap. To judge a predicted bounding box as correct or incorrect one simply calculates the IoU between the predicted bounding box and the ground

[15] All: VGG11, VGG13, VGG16, VGG19
[16] All: ResNet18, ResNet34, ResNet50, ResNet101, ResNet152

truth bounding box, and then judge based on some IoU threshold (e.g. IoU above 0.5 is correct).

The second issue is solved using Average Precision (AP). As we just saw we use IoU to judge bounding boxes as correct or not. All that remains to judge a single object prediction correct or not is to check if the predicted class is correct. We then use AP to combine all the individual predictions to one measure. AP is found by ranking the individual object predictions by confidence[17] and averaging the precision for all prefixes of this ranking. This could also be interpreted as the area under the precision-recall curve — as illustrated in Figure 2.16b. AP is just a measure for one class, and the performance of multiple classes are reported as the mean Average Precision (mAP).

We just explained how one could measure performance in object detection. The specific thresholds and details for how AP is calculated differ slightly from dataset to dataset. We would like to note that it's not completely straightforward or even desirable to translate this performance measure directly to a loss function. A thresholded success metric gives no useful information to gradient descent — continuous errors are needed. There's also the problem of matching predicted bounding boxes against ground truth. If a predicted box doesn't overlap any ground truth boxes, how "incorrect" is that box? And which ground truth box should a predicted box be compared to if it overlaps several ground truth boxes? Because of this complexity and different methods having different demands, one will see that various loss functions are used.

Another problem in object detection is that of annotated data. Annotating every object in an image is resource demanding. Object detection datasets are therefore significantly smaller than image classification datasets. This is the reason that most object detection methods use transfer learning. They first pretrain a CNN on image classification (typically on ImageNet) and then reuse parameters to fine-tune on object detection.

The two most prominent object detection datasets are PASCAL VOC[18] and MS COCO (Lin et al. 2014)[19]. VOC contains 20 object classes, and there are two non-overlapping versions: VOC2007 and VOC2012. Both are split into `train`, `val` and `test`. VOC2007 has almost 5 000 images in `trainval` and in `test`, while VOC2012 has around 11 500 images in `trainval`. VOC2012 `test` is not made public, so it's normal to train on VOC2007 `trainval` and VOC2012 `trainval` — and then test on VOC2007 `test`. COCO is considerably larger with its 80 object classes and over 100 000 images. The splits have changed since its initial release in 2014. In the 2017 version `train`/`val`/`test` split is approximately 115 000/5 000/40 000. The number of object classes and instances per picture is considerably higher for COCO than

[17] Most models output some sort of confidence score in addition to a bounding box and a class. Usually a model output a confidence score distribution over classes and the class with highest confidence score is chosen.

[18] `http://host.robots.ox.ac.uk/pascal/VOC/`

[19] With ImageNet detection dataset at third. There are other datasets, but we see these less frequently used. Refer to Lin et al. (2014) for a comparison of other object detection datasets.

VOC. VOC has 1.4 classes and 2.3 instances on average in an image, while COCO has 3.5 classes and 7.7 instances. Most images (i.e. over 50%) in VOC contain only one object. COCO on the other hand usually has multiple objects.

For PASCAL VOC the performance is measured as mAP with an IoU overlap of 0.5. The way mAP is computed differs slightly between the 2007 and 2012 version. For the 2007 version the individual APs are approximated by sampling the precision-recall curve with 11 evenly spaced recall values, while for the 2012 version the exact APs are calculated. COCO uses several more elaborate performance measures:

- **AP**: Average precision averaged over IoU thresholds $\{0.50, 0.55, 0.60, \ldots, 0.95\}$.
- $\mathbf{AP}^{50}$: Average precision at IoU 0.5.
- $\mathbf{AP}^{75}$: Average precision at IoU 0.75.
- $\mathbf{AP}^{S}$: Average precision for small objects (smaller than 32×32 pixels).
- $\mathbf{AP}^{M}$: Average precision for medium objects (between 32×32 and 96×96).
- $\mathbf{AP}^{L}$: Average precision for large objects (larger than 96×96).
- $\mathbf{AR}^{1}$: Average recall when top 1 detection per image is used.
- $\mathbf{AR}^{10}$: Average recall when top 10 detections per image are used.
- $\mathbf{AR}^{100}$: Average recall when top 100 detections per image are used.
- $\mathbf{AR}^{S}$: Average recall for small objects (smaller than 32×32).
- $\mathbf{AR}^{M}$: Average recall for medium objects (between 32×32 and 96×96).
- $\mathbf{AR}^{L}$: Average recall for large objects (larger than 96×96).

Even though it's just called AP, and not mAP, these scores are usually averaged across all 80 classes. It's most normal to report AP as explained above as the main performance metric on COCO, but we have not been able to find out why this measure was created and used instead of the PASCAL VOC measure (AP^{50}). Redmon & Farhadi (2018) has expressed a sceptical opinion about its lack of justification, and pointed out it's not obvious which metric is best.

We will now summarize some of the most popular object detection methods. These methods can be divided into two groups: two-stage and one-stage detection[20]. Two-stage detection has an initial stage where (relatively) few promising object candidates are identified, and the second stage then evaluates these candidates. While one-stage detection just does some kind of dense regular sampling of candidates across the image and evaluates them all. First we will briefly cover R-CNN based methods — which are two-stage detection methods. Then we also briefly cover YOLO, a popular one-stage detector, before we go more into detail on the one-stage detectors SSD and RetinaNet since these are most relevant for this thesis. There are of course

[20] Also called sparse and dense detection. Or proposal-based and end-to-end approach.

other methods, but this is not an exhaustive review, and we think these are the most representative. We will also refrain from doing an in-depth comparison of the different methods as this will be too lengthy.

R-CNN Based Methods

R-CNN (Girshick et al. 2014) is a method combining traditional computer vision techniques with CNNs. It consists of three distinct steps:

1. **Region proposal**: Use a traditional computer vision technique[21] to propose regions that might contain objects. This is considerably more efficient than brute force sliding window proposals.
2. **Computing features**: Extract the region proposals from the image and feed them through a CNN to get a feature vector.
3. **Classify regions**: Use a linear classifier[22] to classify the feature vector for each region.

Step 2 and 3 need to be trained. The CNN is pretrained on normal image classification and then fine-tuned on actual region proposals. The linear classifiers are trained on feature vectors from the CNN. To get better bounding boxes they also extend the method with a bounding box regression — i.e. they predict bounding box corrections/offsets based on the feature vector.

The R-CNN method greatly increased the state-of-the-art performance for object detection. But a drawback of this method is that it's very slow (i.e. several seconds for a single image even on a powerful GPU). Girshick et al. (2014) use ≈ 2000 region proposals — all of which must be propagated through the CNN. Fast R-CNN (Girshick 2015) solves this problem by extracting the region proposals directly from a feature volume produced by the CNN. So instead of propagating each region proposal through a CNN, the whole image is propagated through a CNN to get a big feature volume and then all regions are extracted from this feature volume. This extraction step is done with what the author names RoI pooling (Region of Interest pooling). Girshick (2015) ends up with a stack of new feature maps which is used to classify and produce bounding box corrections by some fully connected layers. In effect the neural network has taken over the role as a linear classifier. The architecture basically has two steps now: 1) Region proposal and 2) Classifying regions.

The slowest part of Fast R-CNN is the region proposal. Faster R-CNN (Ren et al. 2015) replaces the traditional region proposal technique with a CNN, resulting in the whole architecture being one big CNN. With this approach the CNN does region proposals, feature extraction and classification. Faster R-CNN have state-of-the-art accuracy and is able to process several images per second.

[21] They use selective search.
[22] They use SVMs.

YOLO

YOLO (You Only Look Once) (Redmon et al. 2016) is a method that treats object detection as a regression task, and does not rely on region proposals. It uses a CNN to produce a feature volume and then each feature volume cell is responsible for detecting objects that have its center in that cell. A cell predicts a class and a fixed number of bounding boxes. Each bounding box prediction consists of position, width, height and a confidence score telling how confident the network is that the box contains an object. Object predictions are chosen by non-maximum suppression.

The strength of YOLO lies in its simple architecture. This also makes it very fast — so fast it can do real-time object detection (i.e. above 30fps). The drawback was initially the accuracy, but this has improved in YOLOv2 (Redmon & Farhadi 2017) and YOLOv3 (Redmon & Farhadi 2018) where the authors extend and improve YOLO in different ways.

SSD

SSD (Single Shot multibox Detector) (Liu et al. 2016) is another one-stage end-to-end approach that does not rely on region proposals. Instead of region proposals SSD bases its predictions on a fixed number of anchor boxes of different scales and aspect ratios. For each anchor box it predicts a class and offset corrections of the position, width and height. As with YOLO actual object predictions is chosen by non-maximum suppression. SSD is composed of a base network pretrained on ImageNet and then extended with convolutional layers that decrease in scale. Detection heads are attached to feature volumes of multiple scales to get predictions. The SSD architecture, more specifically SSD300, is shown in Figure 2.17. Now that we have summarized the method we will look closer at some details.

The base network is derived from VGG16 pretrained on ImageNet. All layers up to the last convolutional layers are kept unmodified, but the last layers are changed to make an all convolutional network. The exact details of these modifications are outside the scope of this introduction, so we will just summarize the superficial outcome. The last max pooling layer is changed to 3x3 without stride, the two first fully connected layers are converted to convolutional layers, and the last fully connected layer and softmax are removed. In a simplified view the fully connected layers and softmax are removed and two convolutional layers have instead been appended.

After the base network eight more convolutional layers are added. They alternate between 1x1 convolutions reducing the number of feature maps and 3x3 convolutions scaling down the feature maps. These layers make feature volumes in different scales, which are used to produce detections. We feed different sized feature volumes from the network to detection heads. In SSD these detection heads consist of a location and a class confidence part. The location part is a 3x3 convolution, and the class confidence part is a 3x3 convolution followed by softmax.

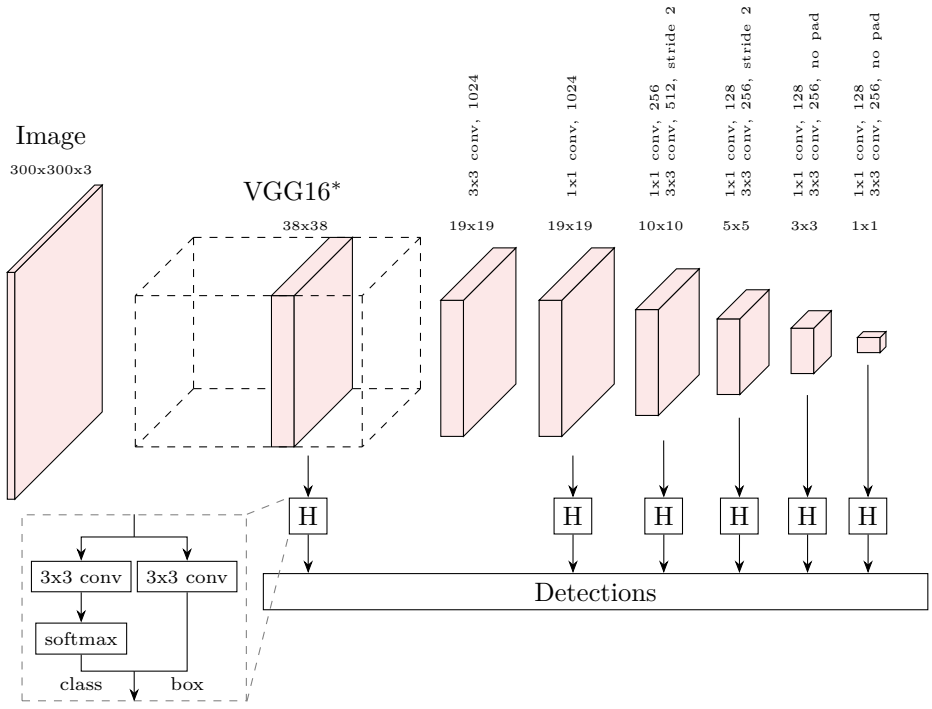


Figure 2.17: The architecture of SSD300. The image is first processed by a pretrained VGG16, before going trough several additional convolutional layers decreasing in scale. Detections are produced by detection heads attached to feature volumes of different scales. The figure is inspired by Liu et al. (2016). *Not the entire VGG16 — see the main text.

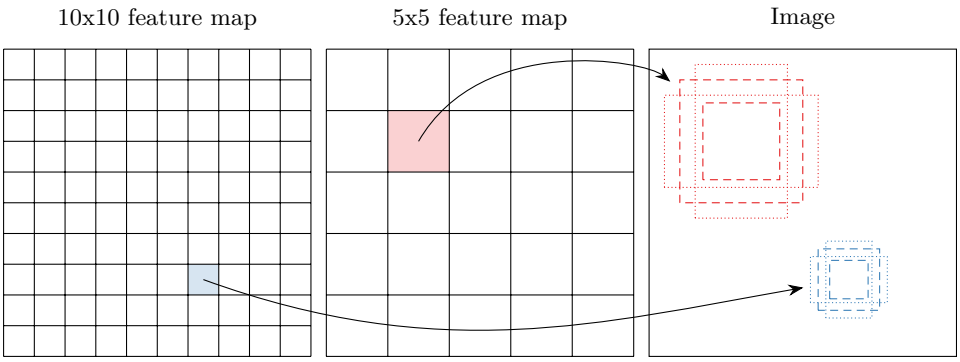


Figure 2.18: Illustration of anchor boxes in SSD. Each cell in a feature volume is associated with a number of predetermined anchor boxes of different scales and aspect ratios. The anchor boxes are centered at the cells receptive field in the image. And cells in lower resolution feature volumes have larger anchor boxes than those in higher resolution feature volumes. The figure is inspired by Liu et al. (2016).

The final feature volume from the detection heads is predictions. Each cell[23] in a feature volume is associated with some anchor boxes of different scales and aspect ratios. These anchor boxes are centered at the cells receptive field in the image. Cells from lower resolution feature volumes have larger anchor boxes, and cells from higher resolution feature volumes have smaller anchor boxes. See Figure 2.18 for an illustration. Liu et al. (2016) use a fixed scheme and some manual tinkering to choose aspect ratios and scales for the anchor boxes. For each box two things are predicted: 1) class confidence and 2) offset corrections of box position and dimension. The idea is that for each anchor box we predict which object class it belongs to and how to make it fit better to the object. Notice that we just predict the box offsets for each anchor box, and not per class. This is why SSD is said to do class-agnostic localization. Since not every anchor box might contain an object, background is added as the 0th class — making it possible for the model to signal the absence of objects. So if we have A anchor boxes for a cell and $|C|$ object classes, we need $A(|C| + 4)$ feature maps ($+4$ because we correct x, y, width and height).

To produce detections for inference we first remove detections classified as background. Then we do non-maximum suppression with IoU 0.45: If two detections overlap with more than an IoU of 0.45 and have the same predicted object class, we remove the detection with lowest class confidence. And if more than 200 detections remain, we only keep the 200 with the highest confidence.

To get the loss at training time we need a more involved setup. Since we generally

[23] A feature volume cell is in effect a vector consisting of values from all feature maps at the same spatial location. In other words, a feature volume is a 2D grid of feature vectors across the image, and one such vector is a feature volume cell.

have multiple detections and ground truth boxes we need to figure out which detections should be compared to which ground truth box. We do this by comparing the anchor boxes and ground truth boxes. First we match each ground truth box to the detection with the anchor box that has the highest IoU overlap. Then for the remaining detections we match if its anchor box has an IoU overlap with a ground truth box above 0.5. This way each detection is either matched with a single ground truth box or none at all, while several detections may be matched to the same ground truth box. The loss is most easily conveyed with formulas, so we will repeat the loss formulas from Liu et al. (2016)[24].

We let $x_{ij}^p = \{0, 1\}$ indicate if detection i is matched with ground truth box j of object class p. The loss is a weighted sum of confidence and location loss:

$$\mathcal{L}(x, c, l, g) = \frac{1}{N}(\mathcal{L}_{conf}(x, c) + \alpha\mathcal{L}_{loc}(x, l, g))$$

where c is the class confidence[25], l the predicted anchor box relative offset, and g the ground truth bounding box. In practice α is set to 1. Let *Pos* be the detections that match with a ground truth box and *Neg* the detections that do not:

$$Pos = \left\{ i : \sum_p \sum_j x_{ij}^p > 0 \right\}$$
$$Neg = \left\{ i : \sum_p \sum_j x_{ij}^p = 0 \right\}$$

Then the location loss is defined as

$$\mathcal{L}_{loc}(x, l, g) = \sum_p \sum_j \sum_{i \in Pos} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^p \widetilde{L1}(l_i^m - \hat{g}_{ij}^m)$$

where $\widetilde{L1}$ is the smooth *L1* loss and $\hat{g}_{ij}$ is the relative offset between the ground truth box j and the anchor box i

$$\hat{g}_{ij}^{cx} = \frac{g_j^{cx} - d_i^{cx}}{d_i^w \sigma_1^2} \qquad \hat{g}_{ij}^{w} = \frac{\log\left(\frac{g_j^w}{d_i^w}\right)}{\sigma_2^2}$$
$$\hat{g}_{ij}^{cy} = \frac{g_j^{cy} - d_i^{cy}}{d_i^h \sigma_1^2} \qquad \hat{g}_{ij}^{h} = \frac{\log\left(\frac{g_j^h}{d_i^h}\right)}{\sigma_2^2}$$

Notice that the offset target $\hat{g}_{ij}$ is normalized in relation to the anchor box and that the dimensions are log scaled. When we want to do inference and relate detections from the network to the image we need to use the anchor boxes and perform these

[24] We present them in a slightly different way suitable for this thesis.

[25] Note that we use c for confidence, while Liu et al. (2016) use it for the class score before softmax.

operations in reverse. σ_1^2 and σ_2^2 are not included in the formulas by Liu et al. (2016) in their publication, but they are used as expressed above in their code. They are set to 0.1 and 0.2 respectively. The motivation behind these is not known, since the authors don't mention them. Lastly the confidence loss is the cross entropy loss:

$$\mathcal{L}_{conf}(x, l, g) = -\sum_p \sum_j \Big(\sum_{i \in Pos} x_{ij}^p \log(c_i^p) + \sum_{i \in Neg^*} \log(c_i^0) \Big)$$

To avoid a large imbalance between positive and negative examples (there are far more negative then positive) we do *hard negative mining*. So we sort the negative detections by confidence and keep only the $3|Pos|$ with the lowest confidence for background. The negative detections that are kept are denoted Neg^*.

Liu et al. (2016) use SGD with momentum and weight decay and schedule the learning rate two be divided by 10 two times They make heavy use of data augmentation, which they found to be essential. Their setup changes slightly depending on which model they train (SSD300 or SSD512) and which dataset they train on. All images are resized to fit the predefined size (300x300 or 512x512), but models for other image sizes can be trained by adjusting anchor boxes. SSD300 and SSD512 achieve 74.3% mAP and 76.9% mAP respectively on VOC2007 `test`, and SSD300 does so with real-time speed.

There are a lot more details that we have not covered. Like how the authors handle that the values in the feature volumes from the pretrained layer in VGG16 are scaled very high. Or how they choose the anchor boxes. But we have covered what is most essential for this thesis and direct the reader to Liu et al. (2016) for more details. In summary — SSD make detections by extracting feature volumes of different scales and regress on them. It's a fast and quite accurate end-to-end approach that can be scaled to trade off speed and accuracy.

RetinaNet

Lin et al. (2017b) combined several ideas into RetinaNet — a state-of-the-art one-stage object detector. Compared to SSD (Liu et al. 2016), the two most important changes are the use of *focal loss* as confidence loss and extracting feature volumes from a *Feature Pyramid Network* (Lin et al. 2017a).

The motivation behind focal loss is that dense object detectors will often have an extreme imbalance between positive and negative (foreground and background) detections, since most locations in an image do not contain objects. Even though most of these locations are easily classified as background by a detector, the share amount of negative detections can still overwhelm the loss — giving the optimization routine little to work with. A standard way to deal with this is hard negative mining (as explained with SSD above) — in effect ignoring some negative detections to balance out against positive. With focal loss no negative detections are ignored. Instead the confidence loss is changed to put less weight on easy detections. This is

simply done by introducing the factor $(1 - c_i^p)^\gamma$ into the cross entropy loss, so that the new confidence loss becomes

$$\mathcal{L}_{conf}(x, l, g) = FL(x, l, g) = -\sum_p \sum_j \Big(\sum_{i \in Pos} x_{ij}^p (1 - c_i^p)^\gamma \log(c_i^p) + \sum_{i \in Neg} \log(c_i^0) \Big)$$

And the full loss still remains

$$\mathcal{L}(x, c, l, g) = \frac{1}{N}(\mathcal{L}_{conf}(x, c) + \alpha \mathcal{L}_{loc}(x, l, g))$$

Notice that we now use all the negative detections *Neg* instead of the detections resulting from hard negative mining Neg^*. The focal loss introduces a new hyperparameter γ that must be set — and it might affect the α hyperparameter. Luckily the authors have found these parameters to have large stable ranges, and they provide strong defaults ($\gamma = 2$ and $\alpha = 0.25$). The focal loss is the main contribution from Lin et al. (2017b).

RetinaNet uses the same concept of anchor boxes as SSD. They have a simplified and more structured scheme for creating anchors — with 9 anchors per feature volume cell across scales. The matching between anchors and ground truth boxes are also simplified. Anchors with IoU overlap above 0.5 are matched with ground truth boxes, and anchors with overlap below 0.4 IoU are matched to background.

The whole architecture of RetinaNet is shown in Figure 2.19. RetinaNet uses a Feature Pyramid Network (FPN) (Lin et al. 2017a) to produce the feature volumes used for detections. FPN starts with ResNet pretrained on ImageNet[26]. Let the last feature volume from each scale of ResNet be called C_1 to C_5. A 1x1 convolution on C_5 is used to produce the feature volume P_5. Then P_5 is upsampled and added with a 1x1 convoluted C_4 before a 3x3 convolution to produce P_4. The same as P_4 is repeated to produce P_3, as shown in Figure 2.19. The 1x1 convolutions are there to get the correct number of feature maps, so they can be added. In addition, smaller feature volumes, P_6 and P_7, are produced with strided convolutions after C_5 to make detection of large objects easier. P_3 through P_7 are used to produce detections with the detection head. The intuition behind using an FPN is that higher resolution feature volumes early in the base network might benefit from higher level features found in lower resolution feature volumes further down in the network. By starting from the last feature volume and scaling up we ensure that all high-level features are used in all scales. But by scaling up we might lose fine-grained details that might be important for localization — thus we make sure to add back the original feature volumes.

The detection head differs in two ways from SSD, it's bigger and weights are shared among all scales instead of unique weights for each scale. The head is now a five-layer convolutional network, and the same head is used at all scales. Notice that the head still has two parts, one for classification and one for bounding box regression, each with its own weights.

[26]The authors use ResNet50 and ResNet101 for their experiments, but all ResNet variants are compatible.

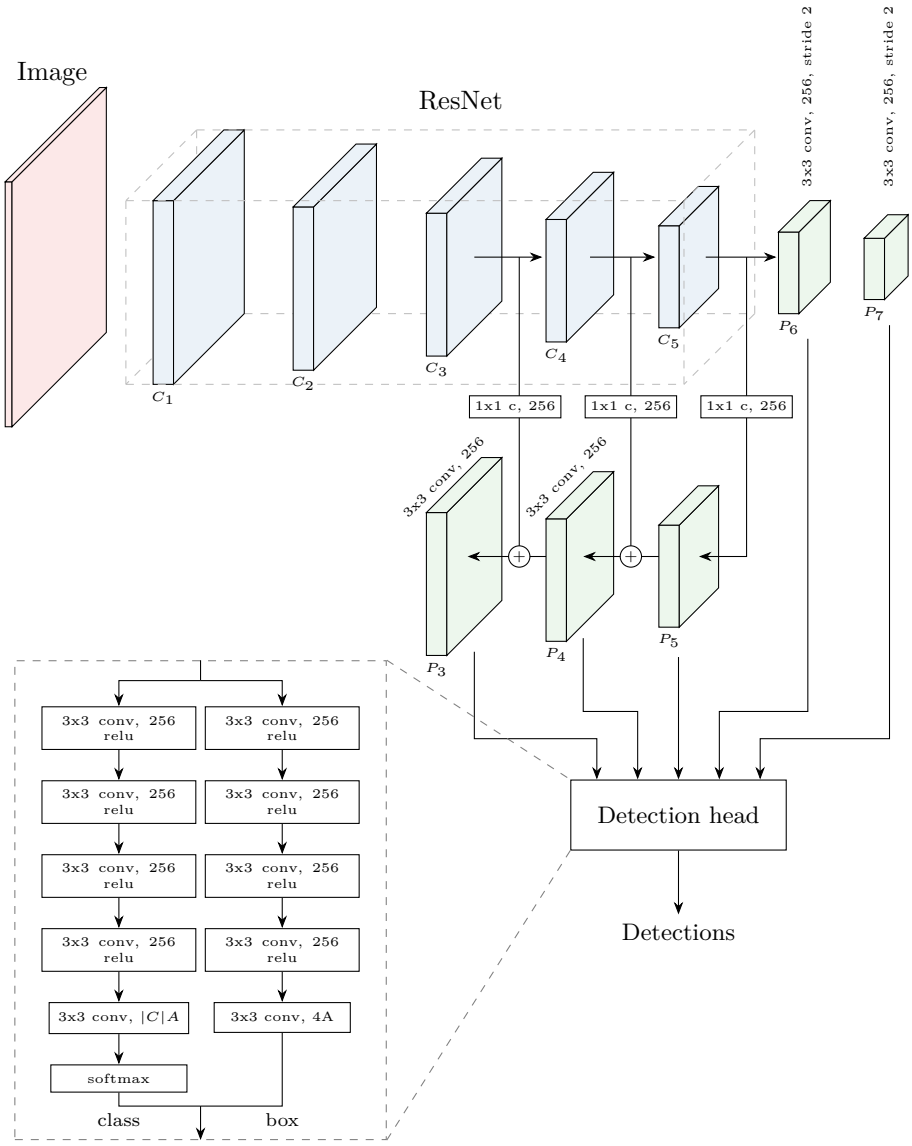


Figure 2.19: The RetinaNet architecture. The image is first passed through ResNet (could be any variant), and then feature volumes of different scales are produced. P_6 and P_7 are just realized by using strided convolutions. But the higher resolution feature volumes P_3, P_4, P_5 are achieved by bilinear upsampling and skip connections from upstream layers. The skip connections make sure no fine-grained details are lost, while still getting the higher level features from C_5. Note that the skip connections include a 1x1 convolution to match up the number of filters. P_3 through P_7 are used to make detections, and they all share the same detection head — i.e sharing weights. $|C|$ denotes the number of classes, and A the number of anchor boxes for a feature volume cell. The figure is inspired by Lin et al. (2017b).

The authors train RetinaNet using synchronized SGD with momentum 0.9 and weight decay 0.0001 on 8 GPUs. They use batch size 16 and train for 90 000 iterations. An initial learning rate of 0.01 is used and divided by 10 after 60 000 and 80 000 iterations. Instead of scaling each image to fit a quadratic form, they keep the aspect ratio and scale the shortest side of the image to some predefined size[27]. RetinaNet achieves competitive accuracy, while still being fairly fast.

2.4 Few-Shot Learning and Meta-Learning

In this section, we will first introduce the few-shot learning setting and contrast it to similar settings. Noticeable work and popular benchmarks will be summarized to contextualize the material. Relevant for this thesis is the approach of meta-learning to do few-shot learning. So we will then introduce meta-learning and see how meta-learning fit into the few-shot learning setting. To more easily motivate the meta-learning methods we use and to see how they differ from others we will introduce a simplified meta-learner taxonomy. After the introduction of meta-learning we will go into greater detail about the methods that are relevant for this thesis — i.e. MAML and its descendants.

2.4.1 Few-Shot Learning

As mentioned earlier, the popularity of deep learning can be partly attributed to impressive results on benchmark datasets, such as ImageNet (Deng et al. 2009). These results show that in some cases deep learning can be used to solve very complex tasks given sufficient amounts of data. Acquiring the amounts of data necessary for training a deep neural network from scratch is often expensive or impossible. To make deep learning applicable to a wide range of tasks we need to be able to learn new tasks efficiently — i.e. learn using as little data as possible.

Transferring knowledge learned from one task to another related task can help make learning more efficient — this is called *transfer learning* (see for example Pan & Yang 2010). Transfer learning is useful if the amount of data for the task at hand is sparse, but there exist large amounts of data for a different but related task. Transferring the knowledge gained from a related task might make learning the task at hand much more efficient.

Transfer learning has seen widespread success in computer vision. Typically, one would train an image classification model on some standardized dataset (e.g. ImageNet), which contains vast amounts of data. And then fine-tune for some other image classification task or a related computer vision task (e.g. object detection), for which there are much less data available. In these days one can often skip

[27] They report accuracy/speed trade-off for 400,500,600,700 and 800.

the first (often time consuming) step completely, as pretrained models are already publicly available for download.

Few-shot learning[28], which is the focus of this thesis, is an extreme form of transfer learning. In a few-shot learning setting, there are only a few examples (e.g. five examples) for the task at hand. The goal of a few-shot learner is then to learn the task at hand using the few examples available and knowledge transferred from a related task. While few-shot learning is not restricted to supervised learning (e.g. Finn et al. (2017) uses it for reinforcement learning), we will mostly cover it in that context.

There are several similar settings to few-shot learning. Since the terminology isn't used consistently, one will find different variants and nuances of these in the literature. We provide a summary of the different similar settings to few-shot learning as we see them and describe briefly how they differ:

- **Zero-shot learning**: Learning to perform a new task for which there are no provided examples. There are instead provided some other helpful information outside the task domain. An example would be to recognize cats in pictures only based on a textual description of cats.
- **Semi-supervised few-shot learning**: Learning a new task with few annotated examples, but many unannotated examples. This differs from standard few-shot learning — where only labeled examples are provided. Situations like these are normal. One may have large amounts of data available for the task at hand, but annotating them may be expensive. It would be beneficial to only annotate some examples, but still learn from all the unannotated examples. Notice that this is just a special case of semi-supervised learning. This has been confusingly called 'Few-example learning' by Dong et al. (2018).
- **Few-shot domain adaptation**: Learning an already known task but in another domain with few examples — in contrast to few-shot learning, where the task is not known beforehand and the domain might be similar. The desired behavior is the same, but the input distribution is different. One might have vast amounts of examples in one domain, but not in another. An example would be to go from recognizing cats in natural images to recognizing cats in hand-drawn sketches with only a few hand-drawn example sketches available.
- **Domain generalization**: Making a model that will perform well on a known task, but in another unknown domain without examples. This can be seen as domain adaptation without examples. The model is supposed to perform well on a new domain without any examples or training.
- **Weakly supervised few-shot learning**: Learning a new task based on

[28] Also called *k-shot learning* or *low-shot learning* in the literature. *One-shot learning* refer to the situation of having only one example, but are also sometimes confusingly mixed with the case of several examples.

few incomplete or inaccurate examples. For example learning to detect cats (classifying and localizing) when examples only specify how many cats are present in an image and contain no localization information.

There are mainly three popular benchmarks for supervised few-shot learning used today:

- **Omniglot** (introduced by Lake et al. 2015): Often described as the 'transposed MNIST'. Omniglot contains 1623 characters from 50 different alphabets written by 20 people. So instead of few classes and many examples per class such as MNIST, it contains many classes with few examples per class. This has become the standard toy dataset for few-shot image classification, and as with MNIST almost perfect accuracy is achieved with state-of-the-art methods. The standard metrics are accuracy on 1-shot and 5-shot for both 5-way and 20-way classification.

- **MiniImageNet** (introduced by Vinyals et al. 2016): A subset of ImageNet consisting of 100 classes and 600 examples per class. Accuracy is tested on both 1-shot and 5-shot for 5-way classification.

- **Sinusoid regression** (introduced by Finn et al. 2017): Mostly used as an exploratory benchmark. When varying the amplitude and phase of sinusoids, the challenge is to perform regression with only a few data points. This was used extensively in our specialization project.

The earliest work on few-shot learning we could find was Fei-Fei et al. (2003, 2006). These are Bayesian approaches for few-shot learning on computer vision tasks, and are from before the rise of deep learning. Lake et al. (2015) presented a new few-shot learning dataset (Omniglot) and another Bayesian approach to few-shot learning — still with no competition from deep learning based methods. But several deep learning based methods have since been introduced and replaced this method as state of the art.

We will now briefly summarize some of the prominent and noticeable methods for few-shot learning introduced since Lake et al. (2015). And afterwards we will discuss the different categories of approaches more generally.

This will not be an exhaustive list (there are of course many more) — as that would be outside the scope of this thesis.

- **Siamese Neural Networks** (Koch et al. 2015): One-shot learning for image classification framed as a matching problem. The output of two twin CNNs with tied parameters are combined using L_1 distance. Presented with two images the network tries to decide if they are of the same class or not. The CNN learns to map an image to an embedding space where images of the same class are close while images of other classes are further apart.

- **Prototypical Networks** (Snell et al. 2017): The network learns an em-

bedding space where new instances can be classified by the nearest[29] class prototype in that space. The class prototype is simply the mean of the embeddings to the provided class examples.

- **MANN — Memory-Augmented Neural Networks** (Santoro et al. 2016): Inspired by Neural Turing Machines (Graves et al. 2014), a fully differentiable network with external memory learns to read examples in sequence and then classify a new example at the end. The network store useful information about the examples in the external memory that can be used to classify an unlabeled example.
- **Matching Networks** (Vinyals et al. 2016): A matching network takes a support set (labeled examples) and a single unlabeled example, and then predicts the label the unlabeled example. The support set is provided as a memory and is accessed with an attention mechanism. That way the network can "look" at examples while classifying the new example. Noticeable, the authors also introduce the MiniImageNet benchmark for few-shot image classification.
- **Optimization as a model for few-shot learning** (Ravi & Larochelle 2017): Learns both initial parameters of a network and the optimizer used for adaptation. The optimizer is an LSTM based RNN.
- **MAML — Model-Agnostic Meta-Learning** (Finn et al. 2017): Optimizes only the initial parameters of a network to be able to adapt to a new task with only one (or a few) gradient descent step. This method is model-agnostic of nature because it only requires SGD for training. It has been shown to work on regression, classification and reinforcement learning.

Some methods are tailored towards specific problems, while others are more general in nature — there are especially many tailored for classification problems. A popular strategy aims to learn a metric space where new instances can be easily compared (Koch et al. 2015; Snell et al. 2017; Vinyals et al. 2016). Such methods are usually non-parametric, i.e. they need the few provided training examples at inference time. One shortcoming of these methods is that they are not easily applied to other problems than classification. The most dominating trend for approaching few-shot learning today is *meta-learning*.

2.4.2 Meta-Learning

Meta-learning[30] is a form of transfer learning where the learner transfers knowledge about how to learn. A meta-learner will try to find common knowledge across tasks, which it then can use to learn new unseen tasks faster and/or more data efficiently. It

[29] By Squared Euclidian distance

[30] There are many possible definitions and variations of meta-learning (see Lemke et al. 2015; Vilalta & Drissi 2002, for a survey). The one that is used in the literature we work with, and therefore also in this thesis, is *learning to learn* as defined by Thrun & Pratt (1998).

uses the data from different tasks to *learn to learn* — as opposed to learning the task itself — and can transfer this knowledge to learn similar tasks. If a meta-learning approach is efficient enough, one can do few-shot learning. Meta-learning can be thought of as learning an inductive bias (Mitchell 1997), and in effect incorporating prior knowledge into the learning procedure.

Let's introduce the abstract problem of doing meta-learning and its notation[31]. A task $\mathcal{T}$ is a distribution over input-target pairs $(\boldsymbol{x}, \boldsymbol{y})$. It's possible to sample data points $\mathcal{D} = \{(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) \sim \mathcal{T}\}$ from a task. We write this as $\mathcal{D} \overset{K}{\sim} \mathcal{T}$ on shorthand, where K is the number of points, or just $\mathcal{D}_{\mathcal{T}}$ if K is not important. Normally, we would train specifically on a particular task $\mathcal{T}$, but for meta-learning we wish to *learn how to learn* a future task $\mathcal{T} \sim p(\mathcal{T})$ from a distribution of tasks $p(\mathcal{T})$. The goal is to have a meta-learner F_Φ parameterized by Φ that given some data points $\mathcal{D} \sim \mathcal{T}$ sampled from a previously unseen task $\mathcal{T} \sim p$ will provide a model $f_{\boldsymbol{\phi}}$ parameterized by $\boldsymbol{\phi}$ that performs well on unseen data $(\boldsymbol{x}^*, \boldsymbol{y}^*) \sim \mathcal{T}$ from that task. Or more straightforward, given some previously unseen task $\mathcal{T} \sim p$ we want

$$F(\mathcal{D} \overset{K}{\sim} \mathcal{T}; \Phi) = f_{\boldsymbol{\phi}}$$

to be such that

$$f(\boldsymbol{x}^*; \boldsymbol{\phi}) = \hat{\boldsymbol{y}}^* \approx \boldsymbol{y}^*$$

for unseen data from the task $(\boldsymbol{x}^*, \boldsymbol{y}^*) \sim \mathcal{T}$. Summarized as

$$F(\mathcal{D}_{\mathcal{T} \sim p}; \Phi)(\boldsymbol{x}^*) = \hat{\boldsymbol{y}}^* \approx \boldsymbol{y}^*$$

In other words, the meta-learner F should be able to learn a new task $\mathcal{T} \sim p$ with K examples. We call this step, going from $\mathcal{D}_{\mathcal{T}}$ to a model $f_{\boldsymbol{\phi}}$ suitable for the task $\mathcal{T}$ using F, *adaptation*. One could of course learn this new task without a meta-learner, but the idea is that a meta-learner has somehow learned to learn faster or more data efficiently. To achieve this we train the meta-learner on tasks sampled from a train distribution of tasks $p_{\text{train}}(\mathcal{T})$ — which is hopefully close to $p(\mathcal{T})$ — to find appropriate meta parameters Φ. And to help us we have a loss function $\mathcal{L}(f_{\boldsymbol{\phi}}, \mathcal{D})$, giving the loss for the model $f_{\boldsymbol{\phi}}$ over the data points $\mathcal{D}$.

There has been proposed several ways to approach meta-learning. Based on the work of Finn et al. (2017) and Finn & Levine (2018), we introduce (in our opinion) a more clear and better-suited way to categorize meta-learners at a high-level for this thesis. This simplified taxonomy, with prominent meta-learners included, are shown in Figure 2.20. We separate the meta-learners into two groups: 1) *Parameter-inducing* and 2) *Non-parameter-inducing*. Parameter-inducing meta-learners use the provided data points $\mathcal{D}_{\mathcal{T}}$ to find suitable parameters $\boldsymbol{\phi} = \boldsymbol{\theta}'_{\mathcal{T}}$ for a model $f_{\boldsymbol{\phi}}$:

$$F(\mathcal{D}_{\mathcal{T}}; \Phi)(\boldsymbol{x}^*) = f(\boldsymbol{x}^*; \boldsymbol{\phi}) = f(\boldsymbol{x}^*; g(\mathcal{D}_{\mathcal{T}}; \Phi)) = f(\boldsymbol{x}^*; \boldsymbol{\theta}'_{\mathcal{T}})$$

[31] We use some of the notation from Finn et al. (2017), Finn & Levine (2018), Nichol & Schulman (2018a), and Nichol et al. (2018b), but it have been adapted and expanded where necessary.

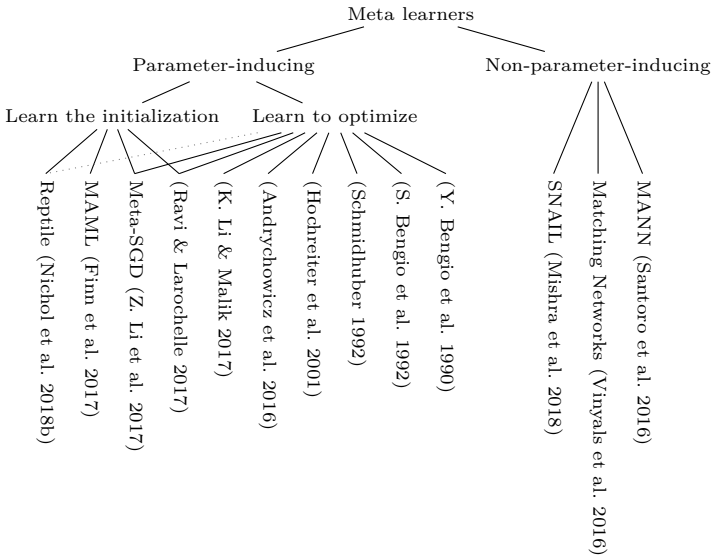


Figure 2.20: Simplified taxonomy of meta-learners. Some prominent meta-learners are included as leaf nodes.

where g is a function parameterized by Φ, taking the potentially variably sized $\mathcal{D}_\mathcal{T}$ as input and giving the constant sized $\boldsymbol{\theta}'_\mathcal{T}$ as output. While non-parameter-inducing meta-learners instead use the provided data points $\mathcal{D}_\mathcal{T}$ as parameters directly to the model:

$$F(\mathcal{D}_\mathcal{T}; \Phi)(\boldsymbol{x}^*) = f(\boldsymbol{x}^*; \boldsymbol{\phi}) = f(\boldsymbol{x}^*; \mathcal{D}_\mathcal{T}, \Phi)$$

and in this case the model parameters are just $\boldsymbol{\phi} = (\mathcal{D}_\mathcal{T}, \Phi)$. One can think of parameter-inducing meta-learners as eagerly training a parametric model when adapting, and non-parameter-inducing meta-learners as lazily training a non-parametric model (in terminology from Mitchell 1997). So notice that parameter-inducing methods will make use of the loss function $\mathcal{L}$ at adaptation time, while non-parameter-inducing methods will not.

Non-parameter-inducing meta-learners (e.g. Mishra et al. 2018; Santoro et al. 2016; Vinyals et al. 2016) have different techniques for letting the model process $\mathcal{D}_\mathcal{T}$. A common theme is to use some kind of mechanism over $\mathcal{D}_\mathcal{T}$ enabling the model to decide which parts to use. Mishra et al. (2018) and Vinyals et al. (2016) use an attention mechanism over input embeddings, while Santoro et al. (2016) input $\mathcal{D}_\mathcal{T}$ iteratively into an RNN which can write to and read from an external memory.

Parameter-inducing meta-learners condense $\mathcal{D}_\mathcal{T}$ into parameters $\boldsymbol{\theta}'_\mathcal{T}$ for some (usually fixed) model f. In practice an initial model $f_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$ is adapted to a

task $\mathcal{T}$ by using the parameter update function g to find new parameters[32]

$$g(\mathcal{D}_{\mathcal{T}}; \Phi) = g(\mathcal{D}_{\mathcal{T}}; [\boldsymbol{\theta}, \Phi - \boldsymbol{\theta}]) = \boldsymbol{\theta}'_{\mathcal{T}}$$

to get the model $f_{\boldsymbol{\theta}'_{\mathcal{T}}}$. Notice that the initial model parameters $\boldsymbol{\theta}$ is considered a part of the meta parameters Φ. There are basically two ways for meta-learners to achieve this. They can either learn the parameter update function g or they can learn the initial model parameters $\boldsymbol{\theta}$. Or they could of course do both. We refer to this as *learn to optimize* and *learn the initialization*, respectively.

As mentioned by Finn et al. (2017), learning to optimize is a popular approach. Earlier methods used quite small models compared to today and the authors were more interested in the low-level biological interpretations (S. Bengio et al. 1992; Y. Bengio et al. 1990; Schmidhuber 1992). Hochreiter et al. (2001) successfully used LSTMs to meta-learn easy regression tasks. But more recent methods apply the approach to more complex problems (Andrychowicz et al. 2016; K. Li & Malik 2017). Ravi & Larochelle (2017) learn both the initialization and how to optimize, but this is not as common as just learning to optimize (from some fixed or random initialization).

As far as we know, learning the initialization is a rather new approach. Finn et al. (2017) introduced *MAML*, where the idea is to optimize the initial model parameters $\boldsymbol{\theta}$ so that they perform well after one or a few SGD steps. In other words, the function g is predefined to be SGD, and MAML tries to find suitable initial parameters $\boldsymbol{\theta}$. Learning just the initial parameters $\boldsymbol{\theta}$ might seem like a very restrictive way to do meta-learning, but Finn & Levine (2018) show that MAML is a *universal learning procedure*. Simplified, this means that MAML can approximate any learning algorithm, and in theory is as expressive as meta-learners that learn to optimize. Z. Li et al. (2017) proposed a method called *Meta-SGD*, extending MAML by also learning parameter-wise learning rates to be used by SGD — and in effect letting MAML learn to optimize to some degree. But this extension has not seen widespread adaptation, and in our specialization project we show why this extension is probably not desirable. Inspired by MAML, Nichol et al. (2018b) introduced *Reptile*[33], which have a less intricate way of learning $\boldsymbol{\theta}$. These methods learning the initialization will be covered in more detail in the next section, where we take a closer look at MAML and its descendants Meta-SGD and Reptile.

2.4.3 MAML and its Descendants

We will now describe the MAML method (Finn et al. 2017), followed by Meta-SGD (Z. Li et al. 2017) and Reptile (Nichol et al. 2018b). None of the methods are limited to supervised learning but will be presented in a simplified manner and focus solely on supervised learning — leaving out unnecessary abstractions and information.

[32] $\Phi - \boldsymbol{\theta}$ should be interpreted as the vector Φ where the subrange containing $\boldsymbol{\theta}$ is removed.
[33] Wordplay on MAM(MA)L

MAML

Algorithm 1 MAML for Few-Shot Supervised Learning

Require: $p_{\text{train}}(\mathcal{T})$: train distribution over tasks
Require: α, β: inner step size and meta-step size
Require: K: examples to train on K-shot learning
1: Randomly initialize $\boldsymbol{\theta}$
2: **while** not done **do**
3: $\quad\mathcal{B} \leftarrow \{\mathcal{T}_i \sim p_{\text{train}}(\mathcal{T})\}$ ▷ Sample meta batch of tasks
4: $\quad$**for all** $\mathcal{T}_i \in \mathcal{B}$ **do**
5: $\quad\quad\mathcal{D}_i \overset{K}{\sim} \mathcal{T}_i$ ▷ Sample K data points from $\mathcal{T}_i$
6: $\quad\quad\boldsymbol{\theta}_i' \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{D}_i)$ ▷ Gradient descept step
7: $\quad\quad\mathcal{D}_i' \overset{K}{\sim} \mathcal{T}_i$ ▷ Sample K new data points from $\mathcal{T}_i$
8: $\quad$**end for**
9: $\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \beta\nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_i \in \mathcal{B}} \mathcal{L}(f_{\boldsymbol{\theta}_i'}, \mathcal{D}_i')$ ▷ Meta gradient descent step
10: **end while**

When using MAML we wish to adjust the initial parameters $\boldsymbol{\theta}$ in such a way that the performance of the model can be as good as possible on a new task with only one or a few gradient steps. Since extending from one ($n = 1$) to more gradient steps ($n > 1$) is straightforward, we will without any loss of generality assume only a single gradient step for the sake of simplicity. Summarized, we sample batches of tasks $\mathcal{B} = \{\mathcal{T}_i \sim p_{\text{train}}(\mathcal{T})\}$, perform a gradient step for each task $\mathcal{T}_i \in \mathcal{B}$ on sampled data points $\mathcal{D}_i \sim \mathcal{T}_i$ to get adapted parameters $\boldsymbol{\theta}_i'$, and then adjust the initial parameters $\boldsymbol{\theta}$ using a gradient step with respect to the loss $\mathcal{L}(f_{\boldsymbol{\theta}_i'}, \mathcal{D}_i')$ on other sampled data points $\mathcal{D}_i' \sim \mathcal{T}_i$ with the adapted parameters $\boldsymbol{\theta}_i'$. The procedure is shown in Algorithm 1 — taken from Finn et al. (2017) and then adapted to the context of this thesis. Note that the stop criterion is considered an implementation detail and may vary. In this thesis, we will do as Finn et al. (2017) and just run for a fixed number of N iterations.

Explained as an optimization procedure, we perform SGD with the objective

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_i \sim p_{\text{train}}(\mathcal{T})}\left[\mathcal{L}(f_{\boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{D}_i)}, \mathcal{D}_i')\right]$$

In practice Finn et al. (2017) use Adam (Kingma & Ba 2015) for this meta-optimization.

Adapting a model trained with MAML to a new task $\mathcal{T}'$ using a few adaptation data points $\mathcal{D}_{\mathcal{T}'}$ is straightforward. Simply adapt by performing n_a (i.e. one or more) gradient steps

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{D}_{\mathcal{T}'})$$

much like on line 6. Only that the parameters $\boldsymbol{\theta}$ are mutated to be the adapted parameters $\boldsymbol{\theta}'$ instead of using $\boldsymbol{\theta}'$ for meta optimization. Notice that the number of adaptation data points $K = |\mathcal{D}_{\mathcal{T}'}|$ is not necessarily the same as when training. A model trained on 10-shot might be used for 5-shot adaption.

One of MAML's biggest strengths is that it's model-agnostic. It's not tied to a specific architecture or type of problem. As long as the model is trained using gradient descent MAML can be applied. Another big advantage is that when the meta-training have found suitable initial parameters $\boldsymbol{\theta}$, adaptation can be achieved by standard SGD techniques. No debt, in the form of method complexity, is inherited from the meta-training step to the adaptation step. When you have found $\boldsymbol{\theta}$ with MAML you are not dependent on the MAML setup anymore. This makes it easier to integrate into already existing pipelines.

As mentioned earlier, even though it might seem very restrictive that MAML only learns the initialization and not to optimize, Finn & Levine (2018) showed that MAML is actually capable of approximating any learning algorithm. The authors call it a *universal learning procedure*. The definition of a universal learning procedure, and the proof that MAML is one, is somewhat technical and involves some assumptions. We will not go into great detail about this since that would be outside the scope of this thesis. But put simply, a meta-learner F is a universal learning procedure if $F(\mathcal{D}_{\mathcal{T}}; \Phi)(\boldsymbol{x}^*)$ is a universal function approximator[34] of $(\mathcal{D}_{\mathcal{T}}, \boldsymbol{x}^*)$.

On the downside, MAML is very resource demanding. The need to backpropagate through a normal gradient step is effectively making the network twice as deep — thereby increasing both the compute and memory needed. And while it is theoretically straightforward to optimize the initial parameters with respect to the loss after more than one gradient step, the already high compute and memory demand scale linearly with the number of steps. In addition, since it is generally harder to optimize deeper networks and that MAML are in effect making the network deeper, this may make it hard for MAML to optimize large models. Some of these problems were addressed by Finn et al. (2017). They tried using a first-order approximation of MAML (named *FOMAML* by Nichol et al. 2018b) which ignore the second order derivatives. FOMAML showed no significant drop in performance compared to MAML. But while it sounds desirable, FOMAML was not given much attention and details of how this was done were left out by the authors. Only very recently Nichol et al. (2018b) pointed out that FOMAML is actually much simpler to implement than what was initially recognized.

Meta-SGD

Z. Li et al. (2017) propose to optimize the step size α in addition to the initial parameters $\boldsymbol{\theta}$. Instead of being a constant scalar, $\boldsymbol{\alpha}$ is a learnable vector with per parameter learning rates. This procedure is called Meta-SGD and is shown in Algorithm 2. Notice the only difference from MAML is the learnable learning rate. In our specialization project we found empirical data indicating that although Meta-SGD models might be better after just one adaptation step $n_a = 1$, the

[34] It's a well known result that (at least some types of) neural networks are expressive enough to, with some assumptations, approximate any function (see for example Hornik et al. 1989). This has also been proved for many of the more modern variants of neural networks.

Algorithm 2 Meta-SGD for Few-Shot Supervised Learning

Require: $p_{\text{train}}(\mathcal{T})$: train distribution over tasks
Require: β: meta-step size
Require: K: examples to train on K-shot learning
Randomly initialize $\boldsymbol{\theta}$ and $\boldsymbol{\alpha}$
while not done **do**
  $\mathcal{B} \leftarrow \{\mathcal{T}_i \sim p_{\text{train}}(\mathcal{T})\}$ ▷ Sample meta batch of tasks
  for all $\mathcal{T}_i \in \mathcal{B}$ **do**
    $\mathcal{D}_i \overset{K}{\sim} \mathcal{T}_i$ ▷ Sample K data points from $\mathcal{T}_i$
    $\boldsymbol{\theta}_i' \leftarrow \boldsymbol{\theta} - \boldsymbol{\alpha} \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{D}_i)$ ▷ Gradient descept step
    $\mathcal{D}_i' \overset{K}{\sim} \mathcal{T}_i$ ▷ Sample K new data points from $\mathcal{T}_i$
  end for
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{T}_i \in \mathcal{B}} \mathcal{L}(f_{\boldsymbol{\theta}_i'}, \mathcal{D}_i')$ ▷ Meta gradient descent step for $\boldsymbol{\theta}$
  $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \beta \nabla_{\boldsymbol{\alpha}} \sum_{\mathcal{T}_i \in \mathcal{B}} \mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{D}_i')$ ▷ Meta gradient descent step for $\boldsymbol{\alpha}$
end while

learned learning rates make the adaptation unstable when $n_a > 1$. In practice, a MAML trained model often becomes better than a Meta-SGD trained model when you allow it to make multiple adaptation steps $n_a > 1$. We showed that the instability could be attributed to overfitted learning rates — we saw both learning rates of extremely high magnitude and learning rates with wrong sign. The problem could be made smaller by optimizing for performance after more than one step, but (as discussed with MAML above) this is not necessarily possible or without its problems. It might be possible to solve the issues with regularization techniques, such as weight decay or parameter sharing, but we hypothesize that most of the claimed advantage Meta-SGD has over MAML is just overfitted learning rates.

Reptile

Algorithm 3 Reptile for Few-Shot Supervised Learning

Require: $p_{\text{train}}(\mathcal{T})$: train distribution over tasks
Require: α, β: inner step size and meta-step size
Require: K: examples to train on K-shot learning
Require: n: number of inner optimization steps
Require: b: size of the inner batches — constrained to $b \leq K$
Randomly initialize $\boldsymbol{\theta}$
while not done **do**
  $\mathcal{B} \leftarrow \{\mathcal{T}_i \sim p_{\text{train}}(\mathcal{T})\}$ ▷ Sample meta batch of tasks
  for all $\mathcal{T}_i \in \mathcal{B}$ **do**
    $\mathcal{D}_i \overset{K}{\sim} \mathcal{T}_i$ ▷ Sample K data points from $\mathcal{T}_i$
    $\boldsymbol{\theta}_i' \leftarrow O_f^{\mathcal{L}}(\boldsymbol{\theta}, \mathcal{D}_i; n, b, \alpha,)$ ▷ Inner optimization
  end for
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \beta \frac{1}{|\mathcal{B}|} \sum_{\mathcal{T}_i \in \mathcal{B}} (\boldsymbol{\theta} - \boldsymbol{\theta}_i')$ ▷ Meta parameter update step
end while

Nichol & Schulman (2018a) first introduced Reptile as a simpler and more scal-

able initialization-learner than MAML. Reptile does not rely on backpropagating through gradient descent steps, and are thereby less resource demanding. Instead of optimizing $\boldsymbol{\theta}$ with respect to the loss on some test data points $\mathcal{D}'_{\mathcal{T}}$ when using the updated parameters $\boldsymbol{\theta}'_{\mathcal{T}}$, Reptile just moves $\boldsymbol{\theta}$ closer to $\boldsymbol{\theta}'_{\mathcal{T}}$ to minimize the difference between them. In other words just moving the initial parameters closer to the updated parameters. As in MAML the meta parameter update is averaged over multiple tasks in a task batch $\mathcal{B}$. The whole procedure is shown in Algorithm 3. $O_f^{\mathcal{L}}(\boldsymbol{\theta}, \mathcal{D}_i; n, b, \alpha)$ is an optimization procedure (e.g. SGD, Adam, or similar) optimizing $\boldsymbol{\theta}$ with respect to the loss $\mathcal{L}$ of f by doing n steps with learning rate α and sampling minibatches of size b from $\mathcal{D}_i$. Notice that no test data points $\mathcal{D}'_i$ are used and the meta update of the parameters $\boldsymbol{\theta}$ is simply an average over the difference from initial parameters $\boldsymbol{\theta}$ to adapted parameters $\boldsymbol{\theta}'_i$ weighted by the meta-learning rate β. Explained as an optimization procedure, Reptile performs optimization with the objective

$$\min_{\boldsymbol{\theta}} \ \mathbb{E}_{\mathcal{T}_i \sim p_{\text{train}}(\mathcal{T})} \left[\frac{1}{2} \left\| \boldsymbol{\theta} - O_f^{\mathcal{L}}(\boldsymbol{\theta}, \mathcal{D}_i; n, b, \alpha) \right\|^2 \right]$$

The meta parameter update step could in theory use any optimizer, but the authors only use SGD — mainly with linearly decreasing β from 1 to 0.

The authors use Adam as inner optimizer O, but to avoid harmful behavior when continously switching tasks momentum is turned off[35] — i.e. setting Adam's $\beta_1 = 0$. They keep the optimizer state and reuse it at adaptation time. So one could argue that the Reptile method *learns to optimize* to some degree since they transfer the optimizer state. But we have found empirically that it makes no difference if we reset the optimizer state before adaptation, so it's not necessary at all to transfer the optimizer state. Therefore we will categorize Reptile as a method that *learns the initialization*, and does not *learn to optimize*.

Some time later Nichol et al. (2018b) updated their article considerably. The claim that Reptile is simpler was removed, most likely because the authors discovered that FOMAML can be implemented much easier than what was known before. They provide both empirical as well as theoretical[36] results indicating that the difference between Reptile, MAML and FOMAML is not that large — except that MAML is harder to implement and less scalable than the first-order methods. Both the authors and earlier work (Andrychowicz et al. 2016; Ravi & Larochelle 2017) indicate that first-order derivatives are most important, and second-order derivatives can be ignored. Another important update from the authors is that of hyperparameter sensitivity. Reptile has an even larger number of hyperparameters than MAML, but contrary to the impression given in the first version of the article,

[35] Although not noted by Nichol & Schulman (2018a), this is actually just RMSprop (Tieleman & G. Hinton 2012).

[36] By doing leading order Taylor expansion of Reptile, MAML and FOMAML they show how these compare with respect to the use of first-order derivatives. They show that all three has one term that minimizes the expected loss and another term that maximizes the inner product between gradients from minibatches in the same task $\mathcal{T}$. Where the latter term can be interpreted as generalization within $\mathcal{T}$.

the authors point out that most hyperparameters are actually very insensitive and robust. This was discovered in our work in parallel before the update from the authors.

Chapter 3

Tools and Environment

In this chapter, we will describe the tools used in this thesis. We will begin by introducing deep learning frameworks in general and summarize popular frameworks. Since we will be using PyTorch, this is covered in greater detail. We then cover hardware requirements, before we introduce tools used to realize our proof of concept tool.

3.1 Deep Learning Frameworks

One of many advances in deep learning the past few years has been in tooling. Specialized frameworks have emerged, and there are currently several major options to choose from. The frameworks enable rapid construction of different deep learning architectures, while still maintaining high performance during training and inference.

Specialized deep learning frameworks exist and thrive because they provide four features:

- **Computational graphs**: A deep learning architecture can from a computational viewpoint be understood most easily as a tensor-oriented computational graph. Computational graphs make it easy to break up architectures into manageable parts, and also serve as a powerful abstract fundament to enable automatic differentiation. Deep learning frameworks enable simple construction and execution of such graphs.
- **Automatic differentiation**: Most deep learning models rely on gradient descent for optimization. To do gradient descent we (of course) need gradients. Deep learning frameworks offer seamless automatic differentiation on computational graphs using backpropagation.
- **High-performance hardware integration**: The computational costs of training neural networks can be large. Because they can easily be parallelized,

we want to run them on high-performance hardware such as GPUs. Deep learning frameworks make it trivial to move computation from CPU to GPU.

- **Deep learning specific utilities**: Many building blocks in deep learning are used over and over again. Optimization algorithms (e.g. SGD, RMSprop, Adam), different kinds of layers and activation functions, data loading and so on are part of nearly every deep learning application. In this context, deep learning frameworks come with batteries included. They offer an abundance of utilities and include implementations of the most used building blocks.

3.1.1 Framework Variation

Even though most frameworks have the four features mentioned above, they can differ in implementation and extra features. We will now outline some of the ways deep learning frameworks can differ.

Interface

Perhaps the most noticeable way frameworks differ from a users point of view is in the way one interacts with them. The majority of frameworks provide a programming interface, but some are (almost) entirely configuration-based. Since the frameworks perform resource demanding tasks, they are mostly implemented in low-level languages (e.g. C and/or C++). However, when they do most of the heavy lifting, there's no reason to make the user work within the same low-level language. Therefore the user is often provided with bindings in one or more high-level languages. As of this writing, the most dominant language is Python, but other languages are used (e.g. Lua, Julia, R, C#, Java).

The philosophy and motivation behind the interface differ between the frameworks. Some aim at being more high-level and fast to prototype, others at giving more fine-grained control. Some aim at being more industry/production-friendly, others at being more academia/research-friendly.

Dynamic vs. Static Graphs

Computational graphs are a powerful abstraction that all major frameworks supply. However, the way this abstraction is handled in practice internally and how it is presented to the user can differ. Most prominent is the distinction between static and dynamic graphs. Static graphs are defined explicitly and declaratively before execution. One first defines how the graph should be. Then feed the graph data and execute it — usually a large number of times, but with different data each time. Dynamic graphs are defined implicitly while the graph executes. All tensors are nodes in the computational graph, and all operations on these tensors produce new

nodes at graph runtime. Dynamic graphs are therefore typically created by writing regular imperative programs operating on tensors.

Both dynamic and static graphs have advantages and disadvantages. Static graphs incur lower runtime overhead and are easier to optimize, but in practice, this is not synonymous with significantly better performance. Because static graphs are explicit and do not require running the graph it makes them easier to package, ship and distribute, making them attractive to production systems. The flexibility of dynamic graphs is their main advantage over static graphs. Blending the graph definition with regular computer code and flow is more flexible because one is less dependent on the abstractions the framework provides for graph construction. This also makes dynamic graphs more effortless to debug. Debugging the graph is just like debugging regular computer code, while in static graphs one have to debug through the graph execution engine. The flexibility and ease of debugging make them attractive for exploring novel ideas/architectures.

Implementation of Automatic Differentiation

Basically all deep learning frameworks support automatic differentiation with backpropagation, but they vary in implementation. This might not make any difference to the user depending on the use case. The difference might be most noticeable when doing work that is tightly coupled to optimization and backpropagation (which we do in this thesis). The main distinction between implementations is in whether they backpropagate directly through the graph and just produce numerical results or extend the graph to include nodes that do the calculations of backpropagation. In the terms used by Goodfellow et al. (2016) the first approach is **symbol-to-number differentiation** and the latter **symbol-to-symbol differentiation**. Symbol-to-number differentiation calculates the gradients in a distinct step decoupled from the execution of the graph. In contrast, symbol-to-symbol differentiation extends the graph and makes the gradient calculations a non-distinguishable part of the execution of the graph. This is illustrated in Figure 3.1. Symbol-to-symbol differentiation has one major advantage: It lends itself much more to intimate use of the gradients. The gradients are just nodes in the graph — as everything else. This makes it more intuitive to implement things such as gradient descent over gradient descent. On the other side, extending the graph may incur overhead if it's not needed.

Deployment

When one has successfully trained a network, one might want to deploy it to a production environment. Some frameworks are supported by ecosystems that make it straightforward deploy and integrate models in production systems (e.g. cloud-based solutions). Moreover, since mobile platforms are becoming increasingly important some frameworks also support deploying and running on mobile devices.

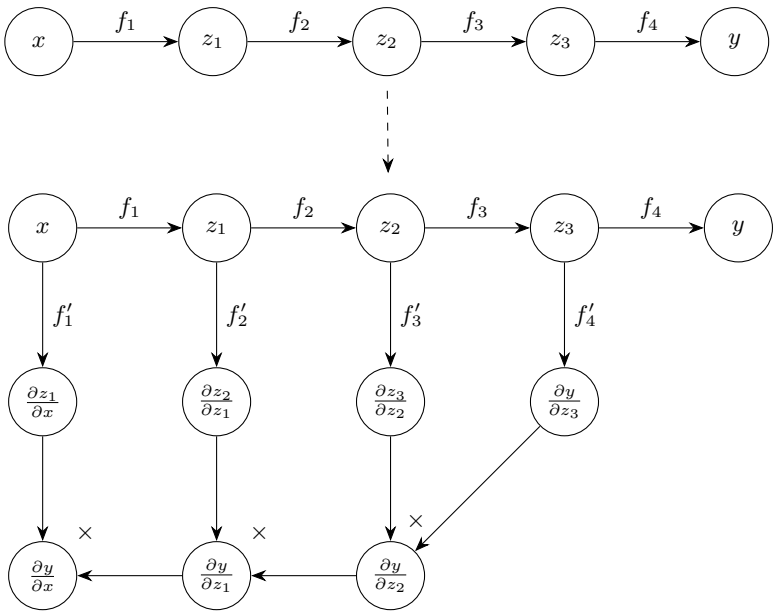


Figure 3.1: Illustration of symbol-to-symbol differentiation. Differentiation is performed by extending the graph with nodes necessary to calculate the gradients using the backpropagation method. In this example, we have a computational graph for the function $y = f(x)$, where f can be composed as $y = f(x) = f_4(f_3(f_2(f_1(x))))$. We backpropagate from y (the output of the computational graph) to find the gradients of y with respect to x (the input in this case). Note that this can be done for arbitrary (directed acyclic) computational graphs, as long as backpropagation can be applied, not just simple graphs such as this one.

Distributed Computing

Training neural networks can be so resource intensive that one machine or GPU isn't enough. Some frameworks support spreading workloads across multiple CPUs/GPUs on one machine and/or across multiple machines. What they support and the level of support varies.

Availability of Pretrained Models

For many applications, it can be of great use to have access to pretrained models. The selection of pretrained models for a framework is largely dependent on the community. Models can be converted to other frameworks, but this is often non-trivial. So it's desirable that someone has already implemented and trained the network one needs.

3.1.2 Popular Frameworks

A full review of different deep learning frameworks is out of scope for this thesis. We will restrict this text to briefly mentioning some of the most popular frameworks.

- **Theano** (**Theano**): Primarily developed at Université de Montréal and released in 2007, it was one of the first major frameworks. It was not made specifically for deep learning, but that is what it's mainly used for. Active development stopped in 2017.
- **TensorFlow** (Martín Abadi et al. 2015): Developed at Google and released in 2015, and has gained massively in popularity since then. It's largely inspired by Theano.
- **Caffe** (Jia et al. 2014): Is geared towards and has been especially popular for image processing. Developed at Berkley AI Research, initially started by a Ph.D. student.
- **Caffe2**[1]: Inspired by Caffe and aimed at production use. Backed by Facebook, released in 2017 and long-term merging with PyTorch ongoing.
- **Keras** (Chollet et al. 2015): A high-level interface on top of TensorFlow, Theano and CNTK developed by a Google engineer and released in 2015.
- **MXNet** (T. Chen et al. 2015): Backed by many industry partners and supports a large number of programming languages and platforms.
- **CNTK**[2]: Developed by Microsoft and first released in 2016.

[1] `https://caffe2.ai/`
[2] `https://github.com/Microsoft/CNTK`

```
import torch

x = torch.tensor([5], requires_grad=True)
z1 = 2*x
z2 = z1**3
y = z2 / 50
```

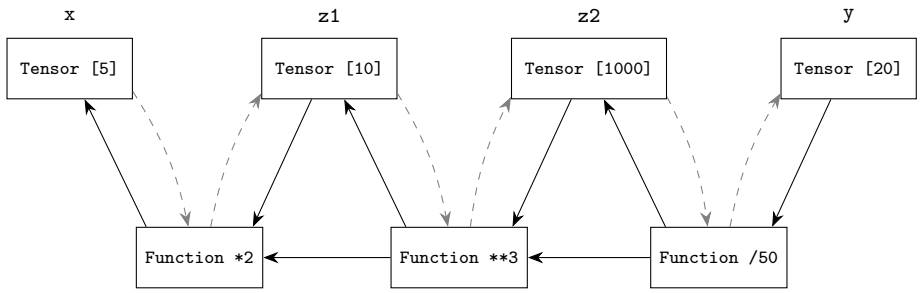


Figure 3.2: Illustration of how PyTorch does automatic differentiation. In this example, we look at the `Tensor` and `Function` objects created after executing the code above — which calculate $\frac{(2x)^3}{50}$ for $x = 5$. The dashed arrows show how the variables were computed, and the solid lines show references. The `Function` objects have essentially made a graph for the reverse mode, and the `Tensor` objects have references into this graph. This example is of course very simple, but the principle remains the same for non-scalars and arbitrary computational DAGs.

- **Torch** (Collobert et al. 2011): A scientific computing framework aimed at machine learning that was first released in 2002.
- **PyTorch**[3]: Backed by Facebook, released in 2017 and built on top of the same backend as Torch, but with very tight and intimate integration with the Python ecosystem. Aimed at research use, but long-term merging with Caffe2 is ongoing.

3.1.3 PyTorch

We will now give a high-level introduction to some of the central parts of PyTorch, which will serve as a background for later sections.

The fundamental data structure in PyTorch is **`torch.Tensor`**, which (of course) represent a tensor. In crude terms, a `Tensor` is equivalent to a NumPy (Walt et al. 2011) array, but can be used on both CPU and GPU. They provide (mostly) the same interface and operations as NumPy arrays, and can also be converted seamlessly to and from NumPy arrays.

[3] `https://github.com/pytorch`

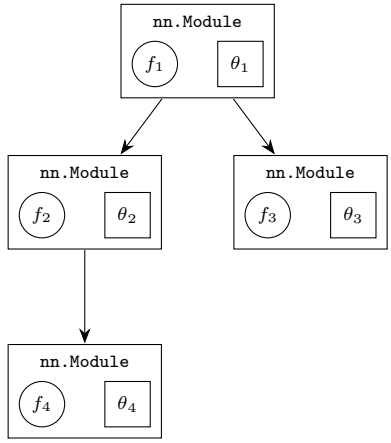


Figure 3.3: Illustration of hierarchically organized `nn.Module`. Each `nn.Module` keep track of its own parameters θ necessary for their forward pass computation f, as well as any submodules used.

The **`torch.autograd`** package makes it possible to do automatic differentiation. This is achieved with the use of some constructs built into `Tensor` — which can be enabled by setting `requires_grad = True` on a `Tensor`. A `Tensor` has a `.grad_fn` property, which is a **`torch.autograd.Function`** object that keeps track of how the `Tensor` was computed. A `Function` object is essentially a node in a computational graph. It keeps track of the computational parents to its associated `Tensor` and necessary intermediate values to make backpropagation possible. When doing an operation on one or more `Tensor`(s) it will create a new `Tensor` for the result with a corresponding `Function`. So `autograd` builds the (reversed) computational graph and cache necessary intermediate values while one perform the forward pass of the graph. See Figure 3.2 for an illustration. One can then easily do backpropagation while traversing this graph. Moreover, here comes a really powerful feature of `autograd`, it can perform this backpropagation with calculations on `Tensor`. That way the gradients are calculated just like every other `Tensor`, making them nodes in the computational graph that can be backpropagated through. In effect, doing symbol-to-symbol differentiation. This makes it trivial to find the gradients of the gradients.

So while one can do arbitrary calculations on tensors and do automatic differentiation over those calculations, these are still relatively low-level building blocks when working with large neural networks. In practice, one wants to manage calculations in larger reusable building blocks and at the same time manage parameters. PyTorch makes the high-level abstraction **`torch.nn.Module`** available for this purpose. `Module` is meant to be subclassed, and when one does so one has to define two things: 1) The constructor — where one initializes parameters and other `Module` instances. 2) A `forward` method — where one performs the forward pass of the

module (optionally with the help of other submodules initialized in the constructor). This way one can organize calculations in hierarchies, as illustrated in Figure 3.3. A `Module` instance will initialize all necessary parameters for itself and its whole hierarchy under it, giving one interface to a large computational graph with its associated parameters. In addition PyTorch provides **`torch.optim.Optimizer`** (and already implemented subclasses), which can keep track of parameters and do optimization steps. Making it possible to write code like this:

```
net = Net() # A Module subclass
optimizer = SGD(net.parameters(), lr=0.01) # Keep track of parameters
for input, target in dataset:
    optimizer.zero_grad() # Zero out accumulated gradients
    output = net(input) # Forward pass of net module
    loss = loss_fn(output, target)
    loss.backward() # Automatic differentiation with respect to the loss
    # Gradients are now accumulated for all parameters
    optimizer.step() # SGD update step
```

3.1.4 TensorBoard

Training a neural network is often very time consuming. Days of training are not unusual — even on powerful GPUs. Hyperparameter search or sensitivity analysis might require running many experiments. One quickly ends up having multiple, simultaneous and long-running experiments — sometimes across machines. Therefore it's necessary to be able to monitor such experiments. We achieve this by logging loss and other relevant data to file while running to a format that is readable by TensorBoard[4] — a tool from TensorFlow (Martín Abadi et al. 2015). TensorBoard launches as a server that monitors arbitrarily many log files and makes them available to the user in a dashboard.

Visualization enables us to detect issues during the training process and compare different training runs.

3.2 Hardware

Training deep neural networks is computationally heavy. With many layers and parameters, it takes time to do both the forward and backward passes. Fortunately, these operations can be expressed as tensor operations that are easily parallelizable — which can leverage the power of modern mass-produced GPUs. Almost all deep learning frameworks use NVIDIAs cuDNN[5] to support GPU computation, this is also true for PyTorch. So for deep learning a cuDNN enabled GPU is required.

The speedup gained from using GPUs imposes requirements on RAM-to-GPU transfer bandwidth because the mini-batches needs to be transferred from the

[4] `https://github.com/tensorflow/tensorboard`
[5] https://developer.nvidia.com/cuDNN

RAM to the GPU for processing. If it takes more time to transfer a mini-batch than process it, the transfer becomes a bottleneck. To calculate gradients with backpropagation the activation values from the forward pass is needed for every example in the mini-batch. These activation values are stored in the GPU's RAM to enable efficient processing. State-of-the-art neural networks for image processing typically need several gigabytes for processing the required batch sizes for stable training.

We have shared access to workstations with Nvidia GeForce GTX 1080Ti (11GB RAM). These are available to us over a 100Mbit local network.

3.3 Proof of Concept Tool

3.3.1 React

React is a JavaScript library used to create interactive user interfaces for the web. Over the past few years React has become one of the most popular user interface frameworks for the web.

In React, user interfaces are designed by declaring views for the application state. To make it easy to declare views, React provides a JavaScript language extension called JSX. The render method in Figure 3.4 is an example of how JSX can be used to declare a view for `state.count`.

As the reader may notice JSX resembles HTML, in fact the `<div>` tag will be rendered as an HTML div element in the browser. In React the `Counter` view is called a React component.

React user interfaces can easily be made interactive through event handlers attached to the user interface elements. In Figure 3.4 an event handler `handleClick` is attached to the button. When the user clicks the increment button `handleClick` will be invoked. The event handler `handleClick` increments `state.count` by calling the `React.Component` method `setState`. When the state changes, React calls the `render` method which returns the updated view. React then inserts the updated view into the browser in an efficient manner. This user interaction process is illustrated in Figure 3.5.

The separation of application state logic and view declaration is part of Reacts appeal. Another strong point for React is the ability to compose components. Figure 3.6 illustrates how components can be composed. Composing allows the developer to create complex user interfaces by arranging simple components into a hierarchy.

```
class Counter extends React.Component {
    state = {
        count: 1
    }

    handleClick = () => {
        this.setState(prevState => ({
            count: prevState.count + 1
        }));
    }

    render() {
        return (
            <div>
                Count: {this.state.count}
                <br />
                <button onClick={this.handleClick}>
                    Increment
                </button>
            </div>
        );
    }
}
```

Figure 3.4: A simple counter view declared in JSX syntax.

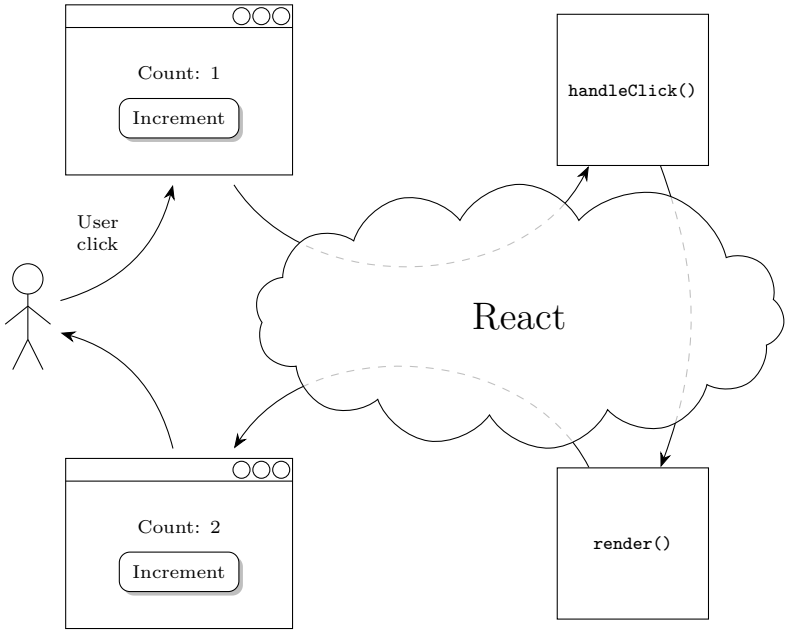
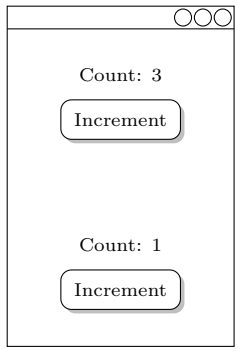


Figure 3.5: Illustration of a user interaction for the `Counter` component in Figure 3.4.




```
class TwoCounters extends React.Component {
    render() {
        return (
            <div>
                <Counter />
                <Counter />
            </div>
    );
    }
}
```

Figure 3.6: Illustration of a composition of two `Counter` components.

```
from flask import Flask
app = Flask(__name__)

@app.route('/path/to/endpoint', methods=['GET'])
def endpoint():
    return "String returned to client"
```

Figure 3.7: Flask endpoint

3.3.2 Flask

Flask[6] is a microframework for making web services in Python. Creating an endpoint in Flask is simple through the use of Python's function decorator syntax, shown in Figure 3.7. Part of Flask's philosophy is to make it simple to get started without limiting the ability to make complex applications later on.

3.3.3 SQLite

SQLite[7] is a simple and self-contained SQL database engine. It comes built-in in the Python standard library, and requires no setup. Compared to other large-scale

[6] `http://flask.pocoo.org`
[7] `https://www.sqlite.org`

and complex RDBMS (relational database management system), SQLite lacks some advanced features and is not suited for a massive number of users. But what it lacks in scalability and features, it makes up for in simplicity. No dependencies, no setup and databases are just stored in a single normal file, but SQLite still exhibits respectable performance. If one wants an SQL database but do not require a large-scale full-blown RDBMS, SQLite will be a natural choice.

Chapter 4

Exploration

In this chapter, we will give the reader insight into our process working our way from sinusoid regression to object detection. Figure 4.1 shows a timeline of key milestones and publications of related work. We will describe the challenges we face and our process to overcome these challenges.

In this thesis, we will focus on MAML and related methods. We chose to focus on these methods because they have many attractive properties for the application we are concerned with — i.e. the user–model feedback loop. When we started working on this thesis, we planned to apply MAML on object detection. We experienced difficulties when scaling up MAML to full-blown object detection, so when Nichol & Schulman (2018a) published a simplified version of MAML, called Reptile, we saw no reason not to try this method instead. The difficulties experienced with MAML will be elaborated on later in this chapter.

W will now motivate why we chose to explore MAML. Although not explicitly stated, the arguments for MAML holds for its descendants (Reptile and Meta-SGD) as well.

As stated, MAML has some attractive properties for our purposes. The foremost property of MAML is that it is model agnostic — i.e. it has few restrictions on the model. MAML's only restriction on the model f is for it to be learnable by gradient descent. A wide range of deep learning models for a wide range of problems are learnable by gradient descent. Therefore MAML can be considered a very general method. Many of the methods in Section 2.4 are tailored to one specific model. This tailoring restricts which problems the method can be applied to, making these methods less general. We argue that a general method is more valuable to explore since it can be part of many useful applications of the user–model feedback loop. We also argue that MAML, because of its generality, is a natural choice for exploring the uncharted territory of few-shot object detection.

In order to apply MAML to a new problem one has to find a suitable model f and

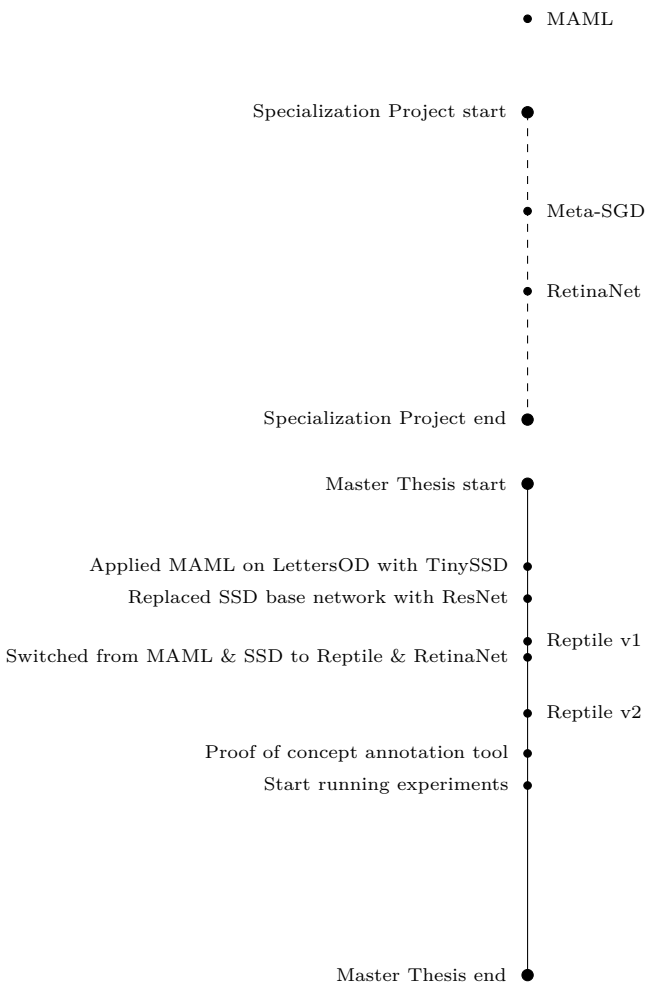


Figure 4.1: Timeline of key milestones and publications of important related work. The key milestones are displayed on the left and publications of related work on the right.

define $p_{train}(\mathcal{T})$. Finding a suitable model f is easy. Because of the model agnostic nature of MAML, one can simply choose an off-the-shelf deep learning model suitable for a task $\mathcal{T}$. Defining $p_{train}(\mathcal{T})$ is the only challenging part. However, we argue that it is a smaller challenge than adapting a method specialized for image classification to object detection.

The downside to the model agnostic nature of MAML is an involved and computationally heavy training procedure. As we can see from Algorithm 1 in Section 2.4, there are two levels of optimization. The outer optimization involves calculation of second-order derivatives by backpropagating through a deep computational graph. This backpropagation step is expensive, and since the computational graph is deep, achieving a good gradient flow can be challenging. Some of the other methods in Section 2.4 have, at least for classification, a more straightforward learning procedure. MAML's use of second-order derivatives may explain why we had issues when training on PASCAL VOC images. These issues are discussed later in this chapter.

Non-parameter-inducing and parameter-inducing methods tradeoff training and evaluation time differently. Non-parameter-inducing methods use the examples $\mathcal{D}_\mathcal{T}$ as parameters directly to the model f. This causes the evaluation time of the user–model feedback loop to increase with the number of examples $|\mathcal{D}_\mathcal{T}|$. For parameter-inducing methods like MAML on the other hand, the model f is not dependent on the examples $\mathcal{D}_\mathcal{T}$. So no matter how many examples $|\mathcal{D}_\mathcal{T}|$, MAML will use the same amount of time on the evaluation step. MAML avoids the dependency by condensing $\mathcal{D}_\mathcal{T}$ into fixed-size parameters $\boldsymbol{\theta}'_\mathcal{T}$ using gradient descent. This might cause the training step of the user–model feedback loop to take longer time as the number of examples increases. For non-parameter-inducing methods the training step in the user–model feedback loop will usually be instantaneous, because the examples are used as parameters directly to the model. There is, therefore, a tradeoff between inference time and training time between parameter-inducing and non-parameter inducing methods. Which category of methods that yield the fastest iterations of the user–model feedback loop will likely vary depending on the number of examples and the application.

By the taxonomy in Figure 2.20 there are two categories of parameter-inducing methods: *learning the initialization* and *learning to optimize*. MAML is a method that learns the initialization. The initialization it learns can be used just like any other standard random initialization of neural network parameters. Therefore MAML can be used and combined with the same methods and techniques usually used to train neural networks. This capability is a significant advantage it has over methods that learn to optimize.

To summarize, there are two main reasons we chose MAML:

1. MAML is a general method easily applicable to a wide range of problems. There might be use cases of the user–model feedback loop that benefits from another meta-learning method, but we think there is more value in a method

applicable to a wide range of use cases and problems.

2. MAML can be easily combined with standard tools and techniques. We believe being able to use MAML with current and future tools and techniques is very important when solving real world problems.

With MAML chosen, we needed to find a suitable object detection model. There are many object detection architectures to choose from. In Section 2.3.3, we identify two types of object detection architectures one-stage and two-stage detection. The two-stage detection models have traditionally had higher accuracy than one-stage methods, but their architecture and training process are usually more involved. To apply MAML we need to modify the training process, so an object detection model with a simple training procedure is preferable. Also, the inference time of one-stage detectors is usually lower, which is beneficial when the goal is to speed up the user–model feedback loop. We chose to use the one-stage detector SSD, described in Section 2.3.3, with MAML. In addition to being a one-stage detector, SSD uses class-agnostic localization — as opposed to R-CNN based methods, which use class-specific localization. In a few-shot learning context, class-agnostic localization is beneficial because all examples of all classes contribute to improving the localization — making the localization training more data efficient.

As far as we know, there hasn't been done a great deal of work on few-shot learning for object detection. A challenge we faced as a result of this was the lack of existing benchmarks tailored to few-shot object detection. Therefore we have adapted the general object detection datasets PASCAL VOC and COCO to our needs. This process is described in detail in Chapter 6.

In our earlier work, we employed MAML on few-shot sinusoid regression, which is a more straightforward task than object detection on natural images. To make the transition from sinusoid regression to object detection on natural images easier, we designed a simple object detection dataset with black letters on white background — we call this dataset LettersOD. The implementation details for this dataset can be found in Chapter 5.

Using a simple dataset like LettersOD allowed us to use a relatively small neural network, which is fast to train and do inference with. It's also easier to get small neural networks to converge. We know from our earlier work that MAML is sensitive to the correct hyperparameters — so having a network that converges easier is beneficial. Therefore we implemented a smaller object detection network we call TinySSD — a scaled-down version of the full-scale SSD network.

As far as we know, MAML has not been used with a larger neural network than the one used by (Finn et al. 2017). Also, we have not found anyone that has used MAML for object detection. Since both using a large neural network and doing object detection with MAML is a novelty, we followed a four-step approach to avoid taking on too much at once.

1. **Standard object detection with TinySSD on LettersOD**: Testing stan-

dard object detection first allows us to get rid of any bugs in our implementation of Functional PyTorch. Also, it's useful to test this first to verify that TinySSD has sufficient capacity to achieve a good score on LettersOD.

2. **Few-shot object detection with MAML and TinySSD on LettersOD**: To test if it's possible to do few-shot object detection at all with MAML.

3. **Few-shot object detection with MAML and SSD300 on LettersOD**: If few-shot object detection works with TinySSD and MAML, the next step is to scale up the neural network. We therefore test SSD and MAML on LettersOD to make sure the network size alone doesn't break MAML.

4. **Few-shot object detection with MAML and SSD300 on PASCAL VOC**: This will test if it's possible to use MAML for object detection on natural images.

For standard object detection, we employed a standard experimental setup where we first train a model on the training set and then test on the test set. After some initial bug fixes, we were able to get near perfect score for standard object detection on LettersOD with TinySSD. This result made us confident that TinySSD had sufficient capacity to do object detection on LettersOD. The next step was to try few-shot object detection with TinySSD and MAML on LettersOD.

To test few-shot object detection on LettersOD we did 5-way 5-shot learning, as Vinyals et al. (2016) do for MiniImageNet. Details of how we defined few-shot learning for the object detection problem is provided in Chapter 6. When using TinySSD and MAML we also achieved a near perfect score. This score proved that, at least for a simple object detection dataset, few-shot learning is possible.

Switching to a full-scale SSD network also went without significant complications, yielding a similar score as TinySSD on LettersOD. With both object detection and larger network working, we moved on to PASCAL VOC images.

The leap from LettersOD to PASCAL VOC images proved difficult. We even had a hard time getting the training process stable. At closer inspection, we saw that the gradients and activations exploded during training. To resolve this issue, we tried both to facilitate more stable learning and to reduce the problem complexity. We tried many things to get MAML to work on PASCAL VOC images:

- **1-way classification**: The first simplification we made was to do 1-way classification instead of 5-way. In a 1-way classification setting there is only one class to detect, all other classes are considered background. This is considerably easier than distinguishing five different classes. Unfortunately, this simplification alone was not enough to make the training stable.

- **Adjusting hyperparameters**: In deep learning, tuning hyperparameters is important, and sometimes the difference between success and failure. The most critical parameter in deep learning is often learning rate. We tried tuning both learning rates and all the other hyperparameters. No combination

of hyperparameters we tried was able to stabilize the training. Due to computational complexity, some of the hyperparameters of MAML is restricted. For example, the compute and memory scales linearly with the number of inner steps n. So the amount of GPU memory sets an upper limit on the number of inner steps.

- **Different learning rates and Meta-SGD**: It is a well-known fact that early layers of CNNs detect low-level features in the image, like edges and basic shapes. These low-level features are useful across domains. The base network we use, VGG16, is pretrained on ImageNet, so it has already learned to detect these features. Moreover, all the classes in PASCAL VOC exist in ImageNet, so the network has also learned some high-level features relevant for these classes. We therefore expect it to be possible to get a decent score on PASCAL VOC with minimal adjustments to the base network's weights. We experiment with different learning rates for the different parts of the network and also try freezing the VGG network and train only the detection heads with MAML. Despite considerable effort we weren't able to find a configuration that made the training stable. Perhaps the learner could tune the learning rates itself? In our specialization project, we explored an extension for MAML called Meta-SGD where the learning rates for each weight is learned in addition to the weights themselves. Having a learning rate for each weight gives the model more flexibility and eliminates α as a hyperparameter, but doubles the number of parameters, makes it more unstable and prone to overfitting. So instead we tried two different approaches inspired by Meta-SGD: per layer learning rates and per filter learning rates. Ideally, the learner would be able to set an appropriate learning rate for the different parts of the network, avoiding too drastic changes in the base network. None of these attempts got rid of the instability, at best it was only postponed.

- **3 step learning**: At this point we thought the instability in training might stem from learning the detection head and meta learn at the same time. When we start training a network from random weights, the loss and therefore the gradients are high. To calculate the second order derivative for MAML's meta update step one has to use backpropagation. Because the graph to backpropagate through is deep the gradient flow is more likely to be poor — i.e. gradients are more likely to explode or vanish. Untrained heads and second-order gradient calculations combined might explain the instability we are experiencing. We try to mitigate this by applying a three-step process. First, we train standard object detection. Then we transfer learn with MAML where we only adjust the classification head of the network. Finally, we allow MAML to finetune the whole network. With this approach, we were able to stabilize the training, but we were not able to achieve meaningful results.

- **Batch normalization and ResNet**: As mentioned above the meta update step is prone to poor gradient flow due to the deep computational graph. Using batch normalization is a common trick in deep learning to aid the

gradient flow. Also, different architectures try to improve the gradient flow with clever tricks. One such network architecture is ResNet, which uses skip connections and batch normalization to allow gradients to flow more easily through the network. Therefore we think it is worth a shot switching to ResNet as the base net to see if that can improve our results. Unfortunately, the switch to ResNet made no significant improvements.

- **Training and testing on the same classes**: To make the task even easier we both train and test on the training set, which contains 15 classes. Training and testing on the training set are just for debugging purposes. With this setup the model is not doing true few-shot learning since the model sees the same classes for both training and testing. We train the model with the 3 step process described above and verify that the score after the first step (standard training) is satisfactory. The model should now be able to distinguish all classes. So in the transfer step we just add an additional layer on top of the detection heads. This was done so the model easily can easily pick out the class it wants to detect. This setup should make it easy to learn the task $\mathcal{T}$, but to our surprise we were not able to achieve meaningful performance.

About halfway through the semester Nichol & Schulman (2018a) released an improvement of MAML called Reptile. Reptile is similar to MAML but use a different method to calculate the meta-step. More details on the differences between MAML and Reptile can be found in Section 2.4. The most significant difference between the two is that Reptile eliminates the second order derivatives, which saves computation and stabilizes the training process. We also switched to RetinaNet (Lin et al. 2017b), an updated version of SSD. The architecture of RetinaNet is explained in detail in Section 2.4. RetinaNet has many advantages over SSD, we summarize some here:

- **Flexibility with respect to base networks**: Easier to use different base networks. This allows us to experiment with different base networks to see what works best. As a compromise between training time and accuracy, we used ResNet34 for most of our exploration.
- **Flexibility with respect to image size**: RetinaNet has a simpler scheme for anchor boxes, making it easier to test images of different scales. This flexibility allows us to trade off accuracy and speed.
- **Better performance**: Through different improvements like Feature Pyramid Network and Focal Loss, RetinaNet has improved performance compared to SSD.

Using Reptile with RetinaNet for 1-way classification on PASCAL VOC we had no problems with training and were able to achieve meaningful results right away. Therefore we decided to use Reptile for rest of the thesis.

With Reptile working on PASCAL VOC we conclude that we have reached Goal 1.1. In Chapter 6 we will devise an experimental setup for the next goal (1.2).

Chapter 5

Implementation

5.1 Functional PyTorch

We use PyTorch as our framework of choice for our work in this thesis. We do not do an in-depth comparison of every framework for our use case since we can't defend the time usage. Instead, we choose PyTorch for two simple reasons: 1) It's flexible and transparent. 2) Its interface is already familiar.

PyTorch uses dynamic graphs and tight integration with Python, which allows for great flexibility in use. One writes control flow in plain Python, and the computational graph is defined dynamically while running. The process is also transparent because the graph execution follows the normal program flow, in contrast to a distinct graph execution environment that is ran after the graph has been defined. Debugging can be made step by step directly in Python code as one would ordinarily do. At the same time, one has access to both common high-level abstractions as well as fine-grained building blocks needed for implementing novel ideas.

The developers of PyTorch have gone a long way in making the use close to that of the SciPy (Jones et al. 2001–) ecosystem. We are familiar with this ecosystem, saving a considerable amount of time when learning to use PyTorch compared to other frameworks.

5.1.1 Extending PyTorch

PyTorch provides all the fundamental building blocks necessary to train a deep neural network (i.e. automatic differentiation, seamless GPU support, etc.). But some of its more high-level constructs, meant to enable structured and easy development, are not well-suited for our special use case with MAML and its descendants. We will now explain why some of PyTorch's high-level abstractions do not fit our needs

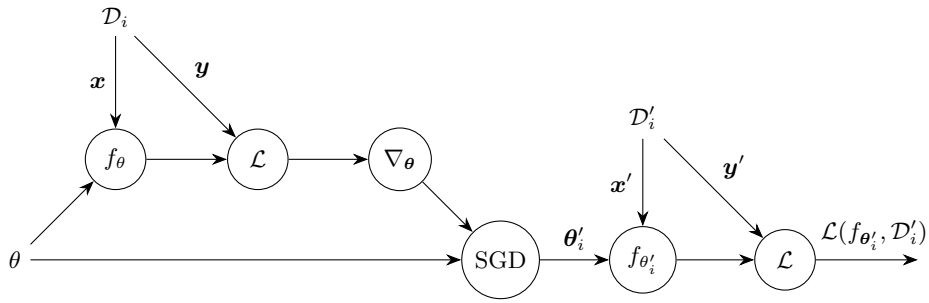


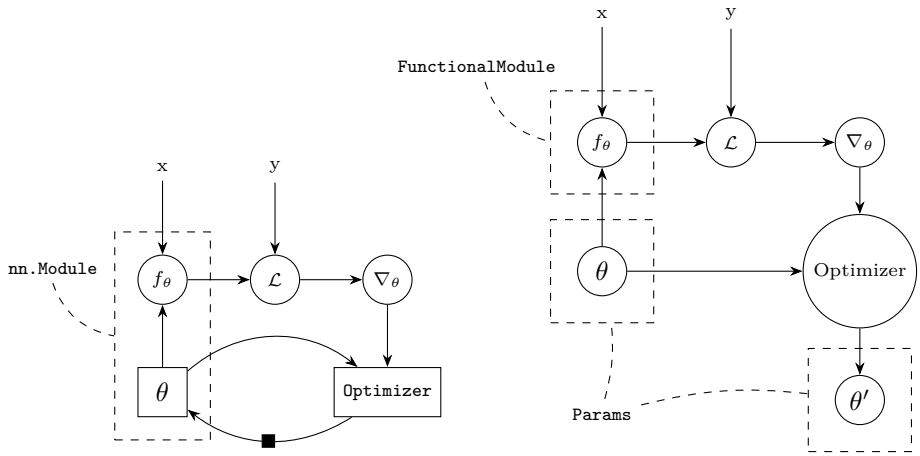
Figure 5.1: A simplified computational graph of supervised learning with MAML for a single task $\mathcal{T}_i$. SGD is the Stochastic Gradient Descent function, i.e. $\theta'_i = \theta - \alpha \nabla_\theta \mathcal{L}(f_{\boldsymbol{\theta}'_i}, \mathcal{D}_i)$ in this graph. The objective is to minimize the meta-loss $\sum_{\mathcal{T}_i \in \mathcal{B}} \mathcal{L}(f_{\boldsymbol{\theta}'_i}, \mathcal{D}'_i)$ with respect to $\boldsymbol{\theta}$.

and how we adapt/extend PyTorch to do so. When we are done we will be able to use PyTorch (with our extensions) to implement MAML and its descendants in an ergonomic and straightforward manner.

Let's first see what we need when using MAML. Figure 5.1 shows a simplified version of the computational graph we need to optimize over when using MAML on supervised learning. The most important thing to notice in this graph is that we have two instances of the parameters (actually more because we have a θ'_i for each example i in a minibatch). Usually, when we do gradient descent we will just update the parameters in-place, but in this case the gradient descent step is just a part of a computational graph that one will do gradient descent over. It's problematic do to in-place operations if one wants to backpropagation because it breaks up the computational graph. And since we want to adjust the initial parameters, based on the performance of updated parameters, we must keep them around.

Figure 5.2a shows how the usual high-level constructs in PyTorch are used for training. The problem is that both `nn.Module` and `optim.Optimizer` are tightly coupled with some set of parameters. `nn.Module` associates a given computational graph with some specific instance of the parameters. This is a good way to do it when doing normal gradient descent. But when we do gradient descent over gradient descent in MAML we want to be able to use several instances of the same parameters. `optim.Optimizer` has the same problem — it's associated with some particular instance of the parameters. But in addition it's also problematic because it does in-place updates of the parameters. We want to be able to keep both the initial parameters and the updated parameters.

We introduce the package **`functional_pytorch`** to solve these problems. The main contribution is `functional_pytorch.FunctionalModule` and `functional_pytorch.Params`. `FunctionalModule` provides the same features as `nn.Module`, but in a functional and stateless manner. Meaning that it does not

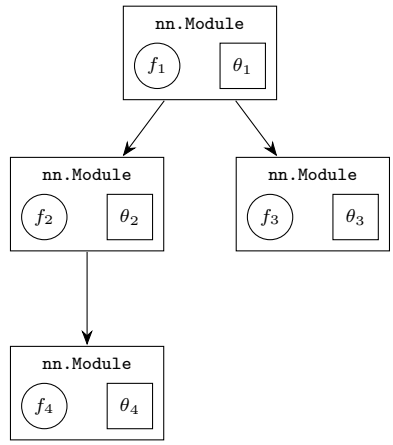


(a) Standard PyTorch training flow (b) Functional training flow

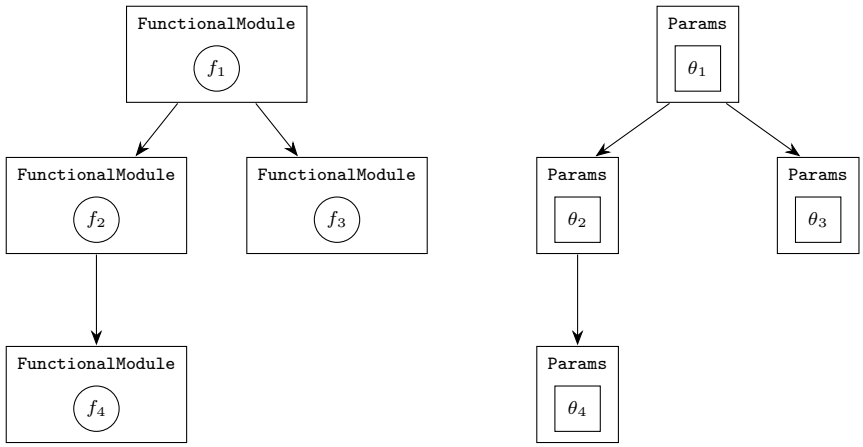
Figure 5.2: Illustration of the transition from stateful to stateless modules, making it possible to implement MAML in PyTorch. Rectangles represent elements with state, while circles represent elements without state — i.e. pure functions. A box on an edge indicates state change at the end of an iteration.

bind itself to a specific instance of the parameters, but rather take a parameter instance as an argument for the forward pass method. `Params` is the structure that encapsulates a parameter instance. Figure 5.3 shows how the parameters are now managed in their own structure (`Params`) separate from `FunctionalModule`. This makes it straightforward to manage several instances of the same parameters. `FunctionalModule` initializes `Params` instances that correspond to that module. The two constructs make up a recursive pair in the sense that a `FunctionalModule` can pass subparams to the corresponding submodule when doing a forward pass. Since parameters are treated as a flat dictionary elsewhere in PyTorch, `Params` has been implemented to behave as such unless used otherwise. Only `FunctionalModule` makes use of the underlying tree structure.

Figure 5.2 shows the change in behavior between the normal stateful training flow and the new stateless flow that `functional_pytorch` enables. Notice that the optimizer also needs to be stateless. This can easily be achieved by not using the stateful optimizers from `torch.optim`. And then just implement the optimizer as a standard function that takes in parameters and gradients, and outputs the updated parameters.



(a) `nn.Module` hierarchy



(b) `FunctionalModule` and `Params` hierarchy

Figure 5.3: An illustration of hierarchically organized `FunctionalModule` and `Params` from the `functional_pytorch` package is shown in (b). Contrast this to (a), where `nn.Module` is used.

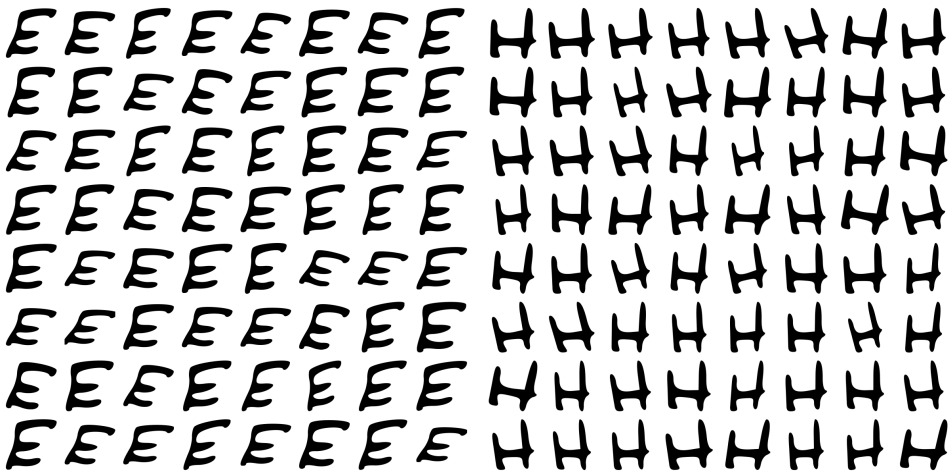


Figure 5.4: Examples of random affine transformations on a specific instance of E and H. The transformations should introduce additional variation in the dataset, but without making the letters harder to identify. Note that for example rotating too much might make it hard to distinguish certain letters.

5.2 LettersOD

The goal of LettersOD is to provide a very simple object detection dataset suitable for development and early testing. We would like it to have similar characteristics, with regard to the number of classes and instances, as PASCAL VOC and MS COCO. MNIST (LeCun n.d.) has too few classes, so instead we base our dataset on EMNIST (Cohen et al. 2017) — an extended version of MNIST with letters. We use only letters, not digits, and thus have 26 classes. Since lowercase letters can be hard to distinguish from certain uppercase letters we choose the simple solution of just including uppercase letters.

To mimic object detection datasets for natural images we would like to insert letters of different scales randomly on a white background. The letters in EMNIST are only 28x28 pixels, and they do not lend themselves well for considerable upscaling. Therefore, all letters are first preprocessed. We use the vectorization tool `Potrace`[1] to convert all letters to SVG. Then all letters are normalized, the SVG stripped to a minimum and then inserted into an SQLite database. This way we can sample and fetch letters by different criterions fast.

Having all letters ready in a database, we can produce object detection images. Letters are simply sampled from the database, combined in a single SVG and then rasterized to a desired resolution. To introduce some variation and avoid overfitting we apply random affine transformations to each letter — see Figure 5.4

[1] `http://potrace.sourceforge.net`

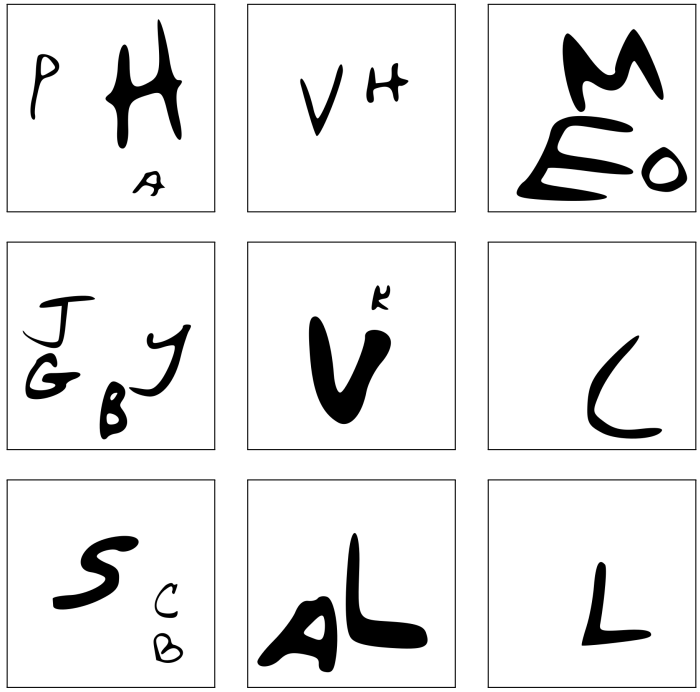


Figure 5.5: Examples from LettersOD. The number of letters and the scale of each letter vary. In addition, we apply random affine transformations on each letter. Note that letters are placed non-overlapping to avoid ambiguity.

for examples of these transformations. The scale is varied for each letter. Some may take up almost the entire image, while others might only take up a very small patch. Objects in natural images may partially occlude each other while it still is possible to identify them. The same is not true for all black letters on a white background. We therefore place the letters non-overlapping. Correctly sampling, augmenting and placing objects on a canvas without overlap is actually an interesting non-trivial problem, but we deem that outside the scope of this thesis, and will not go into detail about how this was done. Figure 5.5 shows some examples of LettersOD.

5.3 Proof of Concept Tool

Our proof of concept tool consists of three main part: *clients*, *workers* and the *server*. The client is implemented as a web app where the user can trigger adaptation of new models and browse their detections. The workers are responsible for training and evaluating the models. The clients and workers communicate with the server over their designated APIs, the *client API*, and the *worker API*. An overview of

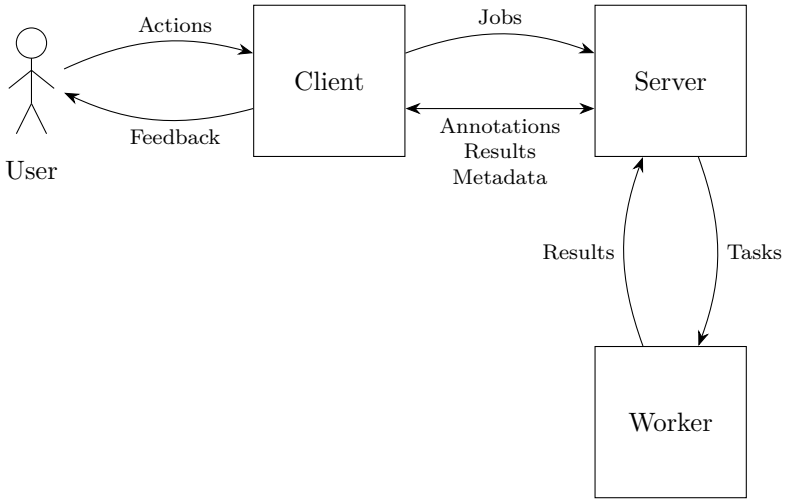


Figure 5.6: Overview of our proof of concept tool. Note that there could be multiple clients and workers.

the system is presented in Figure 5.6.

Two central concepts in the system are jobs and tasks. A job consists of one or more tasks. There are two job types:

- **Adapt job**: Given a model and a set of annotated examples this job produces a new model which is adapted to the set of annotated examples.
- **Detect job**: This job takes a model and a set of images and produces detections for those images.

Typically, the client submits such jobs to the server. The server then divides jobs into tasks and distributes the tasks to the workers for processing. When the workers complete a task, they report the result back to the server.

We will now explain each part in more detail.

5.3.1 Client

Multiple clients can be connected simultaneously. The client is implemented as a simple web app using React (see Section 3.3.1 for more on React). We use React for two main reasons: 1) It has excellent support and a variety of libraries — with for instance premade UI (user interface) components. 2) We already have some experience with it from before.

React has become one of the most popular frameworks for building web applications[2].

[2] `https://insights.stackoverflow.com/survey/2018/`

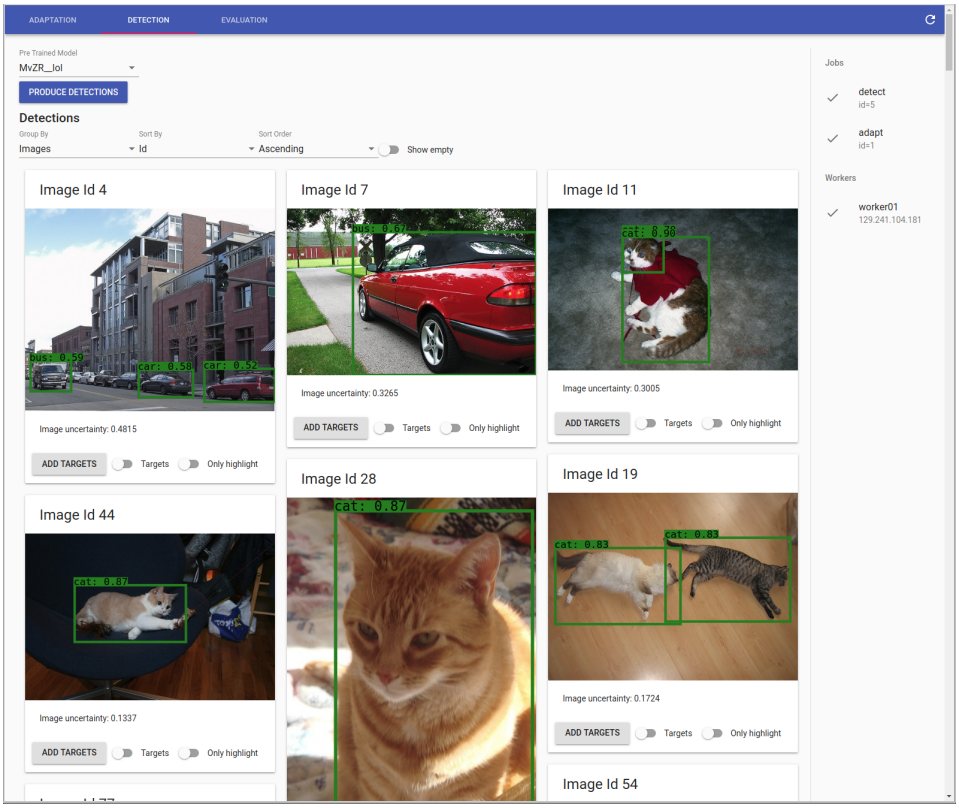


Figure 5.7: Screenshot of the web application's Detection tab.

The popularity has led to an active community with a vast ecosystem — providing many libraries and extensive documentation. The excellent ecosystem makes it less likely that we will get stuck. And we used a community created library in our project to speed up the design and implementation process of the user interface. The library implements Googles Material design as React UI components, and is simply called MaterialUI[3].

Since we have limited time, the fact that we were somewhat familiar with React from before made it a natural choice. This familiarity allowed us to focus on developing features rather than learning a new library.

A screenshot of the web application is shown in Figure 5.7, and more screenshots can be found in Appendix B. We kept the design of the web application simple with three main tabs and a sidebar.

- **Adaption tab**: On this tab, the user can look through the set of annotated examples, adjust some adaptation hyperparameters and trigger adaptation of

[3] `https://material-ui.com/`

the annotated examples. Since this is a proof of concept tool, we have not implemented controls for all the hyperparameters.

- **Detection tab**: On this tab, the user can select a model and trigger a detection job. Then as the detection job finishes, the user can look through a list of images with detections and compare them to ground truth annotations. The list can be grouped by images or detections and sorted by confidence, uncertainty or image id. If the user sees an image where the model has produced an inaccurate detection, the image can be added to the set of annotated examples by clicking "Add targets". The "Add targets" button is a stand in for an annotation tool in our proof of concept tool. For a real-world application, one would have to make a user interface for manually annotating the images to make it possible to use such a system with datasets that don't have ground truth annotations.
- **Evaluation tab**: On this tab, the user can see the performance metrics of the models. The metrics are calculated based on the detections produced in the Detection tab.
- **Status sidebar**: On the right side of the web application the latest jobs are listed with job status, type and id. Also, the registered workers are listed, with the status, name and IP-address of the workers. The web application polls the backend regularly to get the status of the jobs and workers. This feature is useful to get an idea of the state of a job the user has submitted.

5.3.2 Worker

The tool allows for one or more workers to be connected. These workers are powerful workstations with GPUs. GPUs are needed to make the computation efficient since the tasks involve training and evaluating large CNNs.

When a worker starts up it reports to the server which models and datasets it has access to and starts polling the server for tasks. The workers can be ordered to process one of the following tasks by the server.

- Adapt a model to a set of annotated examples.
- Produce detections for a set of images with a given model.
- Download a model from another worker.

When a worker has completed a task, it reports the result back to the server.

5.3.3 Server

The server orchestrates clients and workers, and is realized as a Flask server. It is responsible for receiving jobs from clients, divide it into tasks and distribute the

tasks to workers. The server is not connection-oriented with respect to clients and workers, and clients and workers may come and go as they want during a session.

The tasks are distributed to the workers in a first come, first served fashion — i.e. when a worker polls the server, the worker is assigned the next unfinished task. Note that tasks are pulled rather than pushed — i.e. the workers ask the server for another task, rather than the server telling the worker to start another task. This design choice makes it simple to implement the worker communication using a RestAPI. The downside to this the strategy is that the workers need to continually poll the server for new jobs, which causes unnecessary load on the server and network. However, with a limited number of workers, the load will never become an issue in our proof of concept tool. To manage the application state the server uses an SQLite database. The database is used for managing the jobs, tasks, and workers as well as keeping control with the annotated examples and detections for a model.

Information about the annotated examples and detections for the different models are made available to the clients through another RestAPI. This API also offers endpoints for adding examples and trigger adaptation and detection jobs. To make sure the client has up to date state for the jobs and the workers we let the client poll the server for status at regular intervals. As for the workers, this causes some unnecessary load on the server and network. However, this will not affect the user–model feedback loop duration, therefore we argue that polling is a reasonable solution for our proof of concept tool.

Chapter 6

Experimental Setup

In this chapter, we will motivate and describe our experiments. We will do this in two parts, each part related to its own research question and goal:

1. **Reptile evaluation — Goal 1.2**: To find out how well meta-learning with Reptile performs on few-shot object detection we set up experiments measuring standard metrics for object detection, but in the few-shot setting. There exists no standard way to benchmark few-shot object detection, so an important part of this chapter will be to introduce and motivate the benchmarks we run and the baselines we use for comparison. In addition to measuring performance, we would also like to run some quantitative experiments highlighting the advantages and limitations of Reptile.

2. **Proof of concept tool evaluation — Goal 2.2**: Running a system for training object detection machine learning models would require considerable infrastructure. While the quantitative experiments above will show how capable a Reptile model is to perform few-shot object detection in terms of artificial and isolated benchmark metrics, there are still many other factors that must be taken into consideration when considering the feasibility of a faster user–model feedback loop. There may be many unforeseen practical challenges or obstacles when implementing a real world system that are hard to analyze up front with only traditional machine learning benchmark setups. So to verify the feasibility of actually realizing such a system in practice and testing its speed we set up a less rigorous experiment with our proof of concept tool.

6.1 Reptile Evaluation

Since we are not aware of or found any previous work on either few-shot object detection[1] or meta-learning for object detection we have to establish a framework for evaluating meta-learners. So in this section, we will first explain how we make the problem of object detection fit into the framework of meta-learning and few-shot learning. More specifically, we will establish how a task $\mathcal{T}$ is to be interpreted in the context of object detection and how we sample data points $\mathcal{D}_\mathcal{T}$ from it when doing few-shot learning. The standard benchmarks for object detection are not suited for evaluating few-shot learning. Therefore we will introduce and motivate a few benchmarks with corresponding datasets. These are based upon PASCAL VOC and MS COCO, but adapted to the few-shot setting. And finally, we specify the different experiments we run — both justifying and detailing our choices. Since there are no already established benchmarks or significant earlier work, we also importantly introduce and motivate some baselines for comparison.

6.1.1 Meta-Learning and Few-Shot Learning for Object Detection

We define a task $\mathcal{T}$ in object detection to be such that the input $\boldsymbol{x}$ is an image containing zero or more objects. The desired output $\boldsymbol{y}$ is a set of object detections, where each detection consists of a bounding box and class label chosen among M classes. While $\boldsymbol{x}$ may be an arbitrary image, $\boldsymbol{y}$ must be detections for the same M classes for all $(x, y) \sim \mathcal{T}$. Note that an image might contain objects that are not among the M classes to be recognized. So in other words, an object detection task is to detect all objects of M specific classes. We say a task $\mathcal{T}$ is an M-way object detection task.

It's also necessary to establish exactly what few-shot learning is in the object detection context — which data points are provided when doing K-shot learning? This is not trivial, so let's first define how we handle few-shot learning for object detection, and then discuss it afterwards. Given a task $\mathcal{T}$, let $\mathcal{D}_\mathcal{T} \overset{K}{\sim} \mathcal{T}$ be such that $|\mathcal{D}_\mathcal{T}| = MK$ and each object class is present in at least K images. So for each of the M object classes we sample K images containing that class. We are guaranteed that each class is present in K images, but they might also be arbitrarily present in the remaining $(M-1)K$ images. We say we do K-shot object detection if we adapt to the task $\mathcal{T}$ with the data points $\mathcal{D}_\mathcal{T} \overset{K}{\sim} \mathcal{T}$.

Note that the term K-shot becomes somewhat more convoluted when used this way

[1] We actually found some very recently unpublished work (H. Chen et al. 2018) from March this year doing few-shot object detection, but we discovered this too late to effect our experiments. The authors use a method specifically designed for object detection. There are also other work (e.g. Dong et al. 2018) closely related to few-shot object detection, but no so close that it's natural to have any place in or influence over our experiments.

compared to other meta-learning settings like regression, where the interpretation is straightforward. We argue this unavoidable in practice because of the way object detection is structured. There are three main factors that must be addressed when defining what K-shot should mean:

1. **Multiple classes**: How do you handle classes with respect to K? It is unreasonable to expect a model to learn a class if it has seen no instances of it. So 1-shot learning would be ill-defined if it entailed that you are only provided with a single example for 1 out of M classes. It's therefore natural to assume that each class will have at least one example. In meta-learning for classification this is solved by defining K-shot as the setting where K examples of each class are provided, giving a total of MK examples. This way K-shot is always well-defined, also when $K < M$, because you will always be guaranteed examples of all classes. We adopt the same scheme for object detection.

2. **Multiple objects per image**: Are examples counted at image level or object instance level? In object detection an image can contain an arbitrary number of objects from the same class. So the information contained in a single image may vary significantly. If examples are counted at image level, the number of object instances in $\mathcal{D}_\mathcal{T}$ will vary. Even though $K = 5$, the meta-learner may see way more than K objects instances of each class. But then again, if examples are counted at object instance level, the number of images may vary greatly. Either way $\mathcal{D}_\mathcal{T}$ will vary in ways we can't control. For image level counting, the size of the desired output will vary, while for object instance level counting the size of the input will vary.

   You could of course try to combine them both and fix both the number of object instances and images, but we argue counting examples at object instance level should be avoided in practice. It would make the sampling of examples both too restrictive and too complex. Restrictive because it would severely limit which combination of images that could be sampled together — we risk distancing us from what is realistic in real world settings and also simply excluding too much training data. And complex because it would require solving an intricate problem[2] of which images can be combined to achieve the right number of object instances.

3. **Multiple classes per image**: Closely related to both of the factors discussed above — how do we handle that multiple classes are present in the same image? In normal image classification only one class is considered correct for an image, but in object detection an image can contain multiple classes that must be recognized. If we simply sample K images per class (containing that class), we might end up with classes that are present in more than K images. This would depend on how often different classes co-exist in an image. It would be possible to make a scheme where you could control how many

[2] This problem is in fact NP-hard in general, following from the *subset-sum problem* (Cormen et al. 2009).

images a class would be present in. But we argue, similarly as for the previous factor, that taking into consideration which classes co-exist in the image is best avoided in practice because it would be both restrictive and complex.

In summary, we have defined how K-shot learning is to be understood for object detection and discussed why we choose to do it this way, Mostly because we mean the alternatives are too restrictive and too complex to defend against a simpler approach.

6.1.2 Benchmarks

Few-shot object detection cannot be tested the same way as normal object detection. For normal object detection one would first train a model on some training data consisting of all classes, and then test on some separate test data also consisting of all classes. But in the case of few-shot learning we would like to first train on data only consisting of some classes, and then test by adapting to some other classes afterwards. And in the case for meta-learning, in effect testing if the meta-learner has *learned to learn*.

More formally we would first like to have a dataset split into `train` and `test`[3]. Let C denote the set of all classes. Then the `test` split should consist of the classes $C_{\text{test}} \subset C$, and `train` should consist of the remaining classes $C_{\text{train}} = C - C_{\text{test}}$. A few-shot learner is given full access to all of `train` during training. During testing a task $\mathcal{T}$ using some (or all) of the `test` split classes is sampled. Some adaptation data $\mathcal{D}_{\mathcal{T}} \overset{K}{\sim} \mathcal{T}$ and some separate test data $\mathcal{D}'_{\mathcal{T}} \sim \mathcal{T}$ are sampled from `test`. The few-shot learner can use $\mathcal{D}_{\mathcal{T}}$ to adapt and is then tested on $\mathcal{D}'_{\mathcal{T}}$. Several tasks are sampled and tested this way.

Even though not strictly necessary, we would also like to divide the `test` split into `test-adapt` and `test-test`. And then always sample $\mathcal{D}_{\mathcal{T}}$ from `test-adapt` and let $\mathcal{D}'_{\mathcal{T}}$ just be all data points in `test-test`. This is in contrast to just sampling both adaptation and testing data points from the same pool without replacement. We have three reasons to do it this way:

1. It makes it simple to sample separate adaptation and test data points without the risk of mixing data points between adaptation and testing.
2. It makes it easier to compare results across models adapted to the same task, but with different adaptation data points, because all are tested on the same data points.
3. It combines the traditional way of testing supervised learning with meta-learning. For many traditional supervised machine learning methods we want a standard train/test split. But if we do few-shot learning we would also like to first separately train on some other classes. This way of effectively splitting

[3] In practice one would also often need a validation split. This can be achieved by dividing `train` further. We do not go into detail about this.

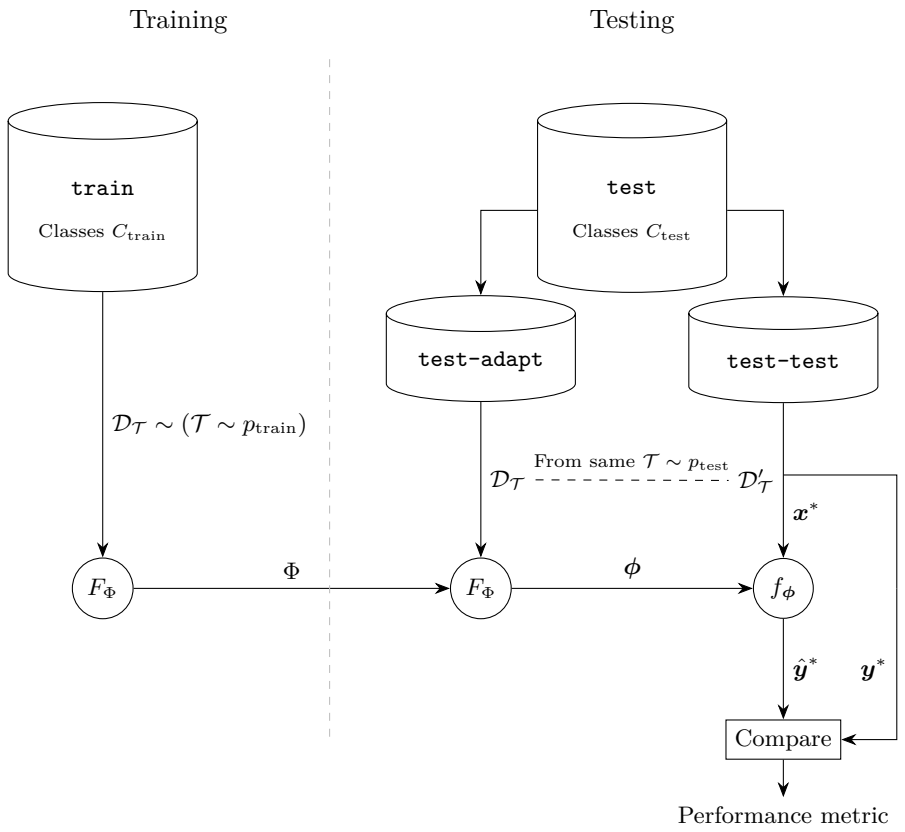


Figure 6.1: Illustration of how training and testing is done for a meta-learner — showing the flow of data.

all the data into three splits (`train`, `test-adapt`, `test-test`) achieves both. It makes it possible to do combinations where one meta-learns first and then use for example active learning at adaptation time.

Given a few-shot dataset as just explained, let's summarize and see how one would test a meta-learner F_Φ in practice. Before testing starts, the meta-learner F_Φ has trained on data from `train` by sampling data points from tasks in p_{train}. The only data transferred from training to testing is the meta-learner parameters Φ. Testing starts by sampling a task $\mathcal{T}$ — which might use any of the test classes C_{test}. Then some adaptation data points $\mathcal{D}_{\mathcal{T}} \overset{K}{\sim} \mathcal{T}$ are sampled from `test-adapt`, the meta-learner provide an adapted model $F(\mathcal{D}_{\mathcal{T}}; \Phi) = f_{\boldsymbol{\phi}}$. Finally, all data points $\mathcal{D}'_{\mathcal{T}}$ in `test-test` are used to test the model $f_{\boldsymbol{\phi}}$ the same way one would do it in a normal non-meta-learning setting. Of course, one should usually test multiple tasks or a task multiple times to take into account the natural variations in task and datapoint sampling. The whole procedure, the flow of data and how the dataset is

used is illustrated in Figure 6.1.

There already exist popular datasets and benchmarks for object detection (e.g. PASCAL VOC and MS COCO). But they are not suited for few-shot performance evaluation out of the box. The main issue is that they are not split by class. We propose three benchmarks that adapt PASCAL VOC and MS COCO to few-shot learning:

1. **PASCAL VOC in 4 folds (FS-VOC)**: Combines VOC2007 `trainval`, VOC2007 `test` and VOC2012 `trainval`. For each fold, 5 unique classes are used for `test`, while the remaining 15 classes are used for `train`. `test` is randomly split 50-50 into `test-adapt` and `test-test`.
2. **MS COCO in 4 folds (FS-COCO)**: Use all images in MS COCO 2017 `train`. For each fold 20 unique classes are used for `test`, while the remaining 60 classes are used for `train`. `test` is randomly split 50-50 into `test-adapt` and `test-test`.
3. **MS COCO to PASCAL VOC transfer (FS-COCO2VOC)**: The 60 classes in COCO that remain when you remove the 20 that exist in VOC are used for `train`. And all images in VOC are used as `test`. `test` is randomly split 50-50 into `test-adapt` and `test-test`.

We will now go into detail about these three and discuss choices we make.

The idea behind the three variants is that FS-VOC and FS-COCO will serve as a test on few-shot object detection when transferring from similar images, but with other object classes. And FS-COCO2VOC will test resilience to a larger change in domain distribution. Since neither VOC or COCO contains a vast amount of classes, we fold FS-VOC and FS-COCO to be able to both train and test on all classes.

One potential issue with testing on VOC and COCO images is that the classes overlap with ImageNet — many of the classes in VOC and COCO are also classes in ImageNet. Since we use a base image classification network pretrained on ImageNet our model will strictly speaking actually have seen some of the target classes it's tested on. One solution would be to pretrain our own base network on ImageNet with the classes from VOC and COCO removed, but this is simply too time and resource consuming. Training a large image classification network on ImageNet require considerable setup, infrastructure and compute resources. We don't even have access to hardware with enough storage capacity to store all the millions of images in ImageNet. Another solution would be to find a dataset without class overlap with ImageNet, but we were not able to find other datasets with enough data and without overlap. While initially this seems like a big issue, it might actually be acceptable in practice. With ImageNet containing 1000 classes it's quite likely that a new task at hand will contain either classes that exist in ImageNet or classes that are closely related to classes in ImageNet. At least if detecting common real world objects — image classification networks pretrained on ImageNet will in general be

able to detect a very wide range of features. For tasks that have little resemblance to classes in ImageNet (e.g. medical image analysis) one would most likely not use an ImageNet pretrained model anyway. The whole point is that the base network is pretrained on similar images. So even though it's not optimal we still argue it's not an unreasonable way to test few-shot object detection. And we will have to defer further investigation of the impact of base network dataset overlap to future work.

The observant reader may wonder exactly how we split the datasets by class. Images in object detection may contain multiple classes, so an image does not immediately fall into either `train` or `test`. Let $h_j^c \in \{0, 1\}$ denote whether image j contain class c or not, and let C_{train} and C_{test} denote the classes in `train` and `test` respectively. We consider every image in the original dataset (VOC or COCO) — across all original splits. We then include an image j in `test` if

$$\sum_{c \in C_{\text{test}}} h_j^c > 0$$

and `train` otherwise. In other words, all images containing at least one of the test classes C_{test} will be included in `test`, and all other in `train`. Only classes in C_{train} and C_{test} are provided as targets in `train` and `test` respectively. This way we make sure a model trained on `train` have never seen any objects from the test classes, although images in `test` may contain objects from the train classes as part of the background (they are not a target, but are still there). Ideally one might want no leakage of classes between the splits, but if we were to remove all images mixing classes between the two splits we would exclude a lot of images. And we argue our solution is reasonable with respect to real world scenarios. Models will be tested on detecting object classes that are clearly unseen, but they may at the same time take advantage of prior knowledge of objects in the background.

We have decided to have four folds for FS-VOC and FS-COCO. This is a good trade-off between getting enough data to train on, test on and compute time (more folds $\implies$ more models). Which classes should be in `train` and `test` for the different folds? We need to divide the classes into four distinct sets $C_{\text{test}}^1, C_{\text{test}}^2, C_{\text{test}}^3, C_{\text{test}}^4$. And let the train classes for fold i be defined by $C_{\text{train}}^i = C - C_{\text{test}}^i$. If we were to divide the classes randomly, we risk very unbalanced `train`/`test` splits. Some object classes occur much more frequently together than other (e.g. bicycle and person). So choosing some particular classes for `test` might result in some classes in `train` having very few images. This is especially the case for COCO, which generally have more classes per image. We solve this problem by casting the dividing of classes as an optimization problem.

C is to be divided into disjoint sets C_{test}^i for $i \in \{1, 2, 3, 4\}$ such that $\bigcup C_{\text{test}}^i = C$ and $|C_{\text{test}}^i| = \frac{|C|}{4}$ for all i. The objective is

$$\min \sum_i \sum_c \Big(\frac{\sum_j h_j^c \hat{h}_j^i}{\sum_j h_j^c} \Big)^2$$

where $\hat{h}_j^i$ denotes whether image j contains any of the test classes C_{test}^i (in other words, if image j is in `test` for fold i):

$$\hat{h}_j^i = H\Big(\sum_{c \in C_{\text{test}}^i} h_j^c\Big)$$

Here is H the Heaviside function. This objective can be interpreted as the fraction of the images containing a class that are being placed in `test`, squared and summed across classes and splits. The main motivation is that we want to avoid `train` having some classes which occur in very few images, making the dataset too unbalanced. One might be tempted to optimize for the size of `train` or some other simple strategy, but this would lead to more rarer classes being sacrificed for more common classes. The number of images a class occurs in vary hugely. Especially, person is present in an order of magnitude more than others. To avoid some training classes being present in too few images in `train` our objective does two things:

1. The loss of images from `train` to `test` per class is normalized. This way each class count equally — common classes can not dominate the rarer.

2. The fraction of lost images from `train` to `test` per class is squared. Effectively prioritizing getting more images for a training class with relatively few images than one that has relatively many.

We tried formulating an integer program and using a strong commercial solver, but it was too large for the solver (though there might exists alternative formulations that would make it feasible). Instead we used randomized local search using simulated annealing (Kirkpatrick et al. 1983), since this is a simple and popular general purpose heuristic. The initial solution is just a random division of classes, and each iteration we try to swap one class from a fold with some class from another fold. The temperature was decayed exponentially. We ran for at least 10000 iterations and terminated when there had been no improvement for 5000 iterations[4]. The final folds will now be presented for FS-VOC — while FS-COCO can be found in Appendix A.

The final folds are presented in Table 6.1. We see that the test classes have clustered somewhat into groups that commonly occur in the same type of scenery. So to make them easier to remember we have given them human-readable names, as shown in the table. In Figure 6.2 we see the fraction of images containing a class that are used in the same split as the class. Since person is present in so many images, it will always make a big impact when chosen as test class. Figure 6.3 shows how many images there are in `test` compared to `train`. Again, we see that the fold with person as a test class suffers the most. It's unfortunate, but still considerably better than what it would be like if we made four random folds.

[4] Note that we implemented this in C++ with some some tricks to bring down the running time to reasonable levels. The objective value is very entangled and it's hard to find shortcuts or ways to calculate it incrementally — especially for COCO.

Urban w/o person	Urban w/ person	Rural	Indoor
C^1_{test}	C^2_{test}	C^3_{test}	C^4_{test}
aeroplane	bicycle	bird	chair
bus	bottle	boat	diningtable
car	horse	cow	pottedplant
cat	motorbike	dog	sofa
train	person	sheep	tvmonitor

Table 6.1: The four different folds of FS-VOC. The classes in `test` are listed for each fold.

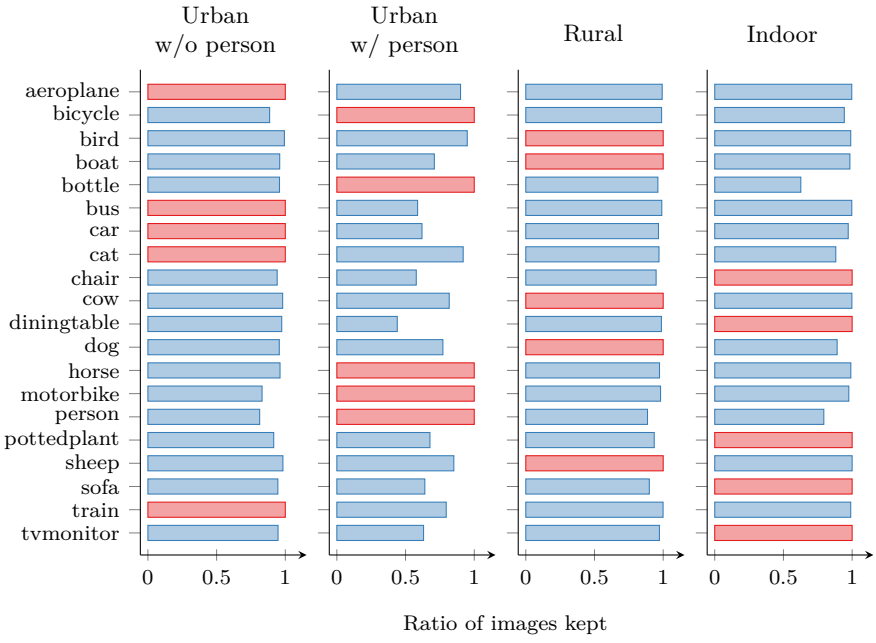


Figure 6.2: Plot showing the fraction of images containing a class that are kept in `test` (red) and `train` (blue) for all four folds of FS-VOC. Remember, because of the way we divide the images, `test` always gets all images containing test classes.

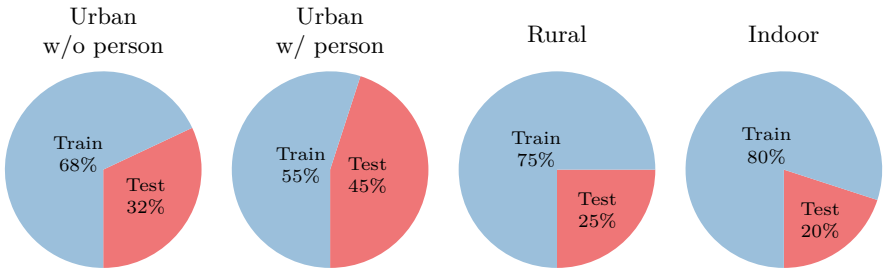


Figure 6.3: Plot showing ratio between the number of images in `train` and `test` for each folds in FS-VOC.

For FS-COCO2VOC the train and test classes are predetermined by COCO and VOC. Removing all VOC classes from COCO the way described above will remove a lot of images. So in order to address this and also see to which degree the method is able to generalize to new unseen data we choose to let FS-COCO2VOC be made in three ways:

1. **FS-COCO2VOC, strictly no VOC classes**: All images with any of the VOC classes are removed from `train`. This is equivalent to the strategy used for FS-VOC and FS-COCO.
2. **FS-COCO2VOC, strictly no VOC classes except person**: The same as above, but allow images with person — so there might be persons in the background of the training images. This drastically increases the number of images in `train`.
3. **FS-COCO2VOC, VOC classes allowed in background**: Allow all images in `train` as long as some of the training classes are present, regardless of whether an image contains a test class or not.

To produce performance metrics we use the procedure described above, and illustrated in Figure 6.1. For FS-VOC and FS-COCO2VOC we report PASCAL VOC 2012 metrics, while for FS-COCO we report COCO metrics. For FS-VOC and FS-COCO we average metrics across folds. The number of tasks and samples per task can be varied depending on the compute resources available and the desired uncertainty. As we have limited compute resources we will run some of the experiments with relatively few repetitions.

6.1.3 Baselines

Since we have no earlier work to compare to and we introduce new benchmarks, we need to have one or more baselines for comparison. We will operate with two baselines: 1) Standard transfer learning and 2) Transfer learning from joint training.

Standard transfer learning simply trains normal object detection on all classes in

`train` at the same time, and then does transfer learning to $\mathcal{D}_{\mathcal{T}}$ to learn the task $\mathcal{T}$. Adaptation is basically the same as for a model trained with Reptile. The main difference is that the initial parameters $\boldsymbol{\theta}$ are obtained by standard training instead of Reptile training. Since the number of classes changes from `train` to `test` we must change the class confidence heads and the optimizer state. We argue that standard transfer learning represent the most simple, straightforward and natural way to do few-shot learning if one doesn't use few-shot learning specific methods.

Joint training can be achieved by simply setting $n = 1$. Then we optimize for the expected loss across tasks — and in effect doing joint training (Nichol et al. 2018b). According to Nichol et al. (2018b) this should be a strong baseline.

6.1.4 Setup

We will now present the quantitative experiments we will run with Reptile. Note that we have limited time and compute resources, so we have to scale the experiments accordingly. First we will list all meta-learners and baseline models trained. And then we will list all evaluations

Models

The authors of RetinaNet use ResNet101 as base network and resize images so that the shortest edge has some large constant size and keep the aspect ratio, but we use ResNet34 for all models and resize all images to 500x500. Because, while Lin et al. (2017b) can run synchronized SGD on eight GPUs we need to run everything on one GPU. We adapt the same scheme for anchor boxes, but scale them with respect to our image size. This setup has been tested on both VOC and COCO, and achieve a score of 81.3 and 30.1. Even though we have scaled down considerably, we get decent accuracy.

We train 11 Reptile models using the hyperparameters in Table 6.2a, one for each fold on FS-VOC and FS-COCO and one for each FS-COCO2VOC variant. And for each Reptile model we also do joint pretraining and standard pretraining, using the hyperparameters in Table 6.2b and 6.3 respectively. While Nichol & Schulman (2018a) first reported what seemed to be extremely fine-tuned hyperparameters, Nichol et al. (2018b) reported later (after we started using Reptile) that Reptile is actually very robust to the choice of hyperparameters. This is consistent with our experience, and we so didn't spend much time finding suitable hyperparameters. As Nichol et al. (2018b), we choose the K used during training independent from the number of examples the models will be adapted to when tested.

We make extensive use of data augmentation. Including horizontal flipping, zoom in, zoom out, cropping and photometric distortion. All augmentations are aggressively toned down for COCO images to avoid small object becoming unrecognizable. In addition to these standard augmentations on the image level we also apply

Hyperparameter	Notation	Value
Inner steps	n	10
Inner batch size	b	16
Inner step size*	α	0.0001
Training shots	K	10
Meta-step size**	β	1.0
Meta-batch size	$\lvert\mathcal{B}\rvert$	2
Iterations	N	5 000

(a) Reptile

Hyperparameter	Notation	Value
Inner batch size	b	16
Inner step size*	α	0.0001
Training shots	K	10
Meta-step size**	β	1.0
Meta-batch size	$\lvert\mathcal{B}\rvert$	2
Iterations	N	50 000

(b) Joint pretraining

*Adam with no momentum ($\beta_1 = 0$) is used as inner optimizer — as Nichol et al. (2018b).
**β is linearly annealed to 0 as Nichol et al. (2018b).

Table 6.2: Reptile and joint pretraining hyperparameters used for all experiments.

Hyperparameter	Notation	Value
Batch size		16
Step size*	α	0.0001
Weigh decay		0.0001
Iterations	N	100 000

*Adam is used and α is decreased by a factor of 10 after $N = 60\,000$ and $N = 80\,000$.

Table 6.3: Standard pretraining hyperparameters used for all experiments.

augmentation on the task level. We randomly rotate tasks 90, 180 or 270 degrees. If task $\mathcal{T}$ is rotated 90 degrees then all images from that task are rotated 90 degrees. The idea is to increase the variation in tasks — learning to detect upside down horses are different than normal horses — and thereby increasing the model's ability to learn new tasks.

While it would be ideal to train more than one model for each dataset variant, we limit ourselves to the above setup because it would be impossible to train more models given the computational resources we have access to. We have set the hyperparameters so that each model takes approximately 25 hours to pretrain on the hardware available for us. With a total of 33 models, it's no surprise this is very time consuming.

Experiments

- **FS-VOC benchmark**: We measure the performance of Reptile on FS-VOC. Following the standard of testing 5-way classification on MiniImageNet (Vinyals et al. 2016), we test 5-way object detection. In a user–model feedback

loop you would ideally start with relatively few annotated examples, and then iteratively add more as needed. We argue that both $K = 5$ and $K = 20$ would be a reasonable number of examples to start with in an ideal setting, depending on the task. 5 and 20 examples are relatively few examples and can most likely be acquired quite quick[5], but is still enough to capture some of the variation of the target object classes. We therefore test both 5-shot and 20-shot as two reasonable start levels for a fast user–model feedback loop.

For each model we evaluate after 10, 100 and 1000 steps, and repeat the evaluation 20 times for each fold to account for variations in $\mathcal{D}_{\mathcal{T}}$ sampled from `test-adapt`. All repeats use all data points in `test-test` to produce a mAP score.

- **Many examples**: We measure the performance of Reptile when given many examples from FS-VOC. Reptile might speed up early iterations in the user–model feedback loop. But what happens when you later acquire more examples and you don't have *few* examples anymore? Can you still use the same meta-learner, or will the performance degrade compared to standard methods? We let $\mathcal{D}_{\mathcal{T}}$ be all images in `test-adapt` and evaluate after 10, 100 and 1000 steps as before, but also after 10000 because of the large number of examples. Since this is time consuming we only do one repetition per fold.

- **Very few examples**: We measure the performance of Reptile when given very few examples. Is Reptile valuable even if you provide a minimum number of examples? We test $K = 1$ through $K = 4$ and repeat the evaluation three times per fold.

- **FS-COCO benchmark**: We measure the performance of Reptile on FS-COCO. Doing object detection on COCO images is considered significantly more challenging than VOC. This will test Reptile on a harder benchmark than FS-VOC. We evaluate the same way as we do for FS-VOC.

- **FS-COCO2VOC benchmark**: We measure the performance of Reptile when trained on COCO images, and then evaluated on VOC images. Images from COCO and VOC are noticeably different. How will Reptile handle the larger shift in domain distribution? For each model we evaluate after 10, 100 and 1000 steps, and repeat the evaluation 5 times per fold.

Since we train on relatively few examples some task adaptations might be unstable when doing a fixed large number of steps. To give all models the best possible chance and to avoid outliers we provide a scheme for early stopping of unstable models when adapting. When we adapt for a fixed number of steps (i.e. 10, 100, 1000 or 10000) we maintain a running mean of the loss for the last 30 steps. If the running mean ever reach a value twice as high as the lowest running mean observed

[5] With $K = 5$ and $K = 20$ for 5-way classification we will need 25 and 100 examples accordingly. If we assume annotation times from Papadopoulos et al. (2017) and three object instances per image it will take approximately 10 and 40 minutes respectively.

so far, we stop and keep the model as it were when the running mean was at its lowest. Otherwise we just keep the model as is after the fixed number of steps.

6.2 Proof of Concept Tool Evaluation

The main goal of this evaluation is to discover how much speedup (if any) Reptile could potentially bring to a user–model feedback loop for object detection. There are basically three time-consuming steps in this loop: 1) User actions — i.e. data collection, annotation, choosing hyperparameters 2) Training a model 3) Evaluating a model. How fast the feedback loop is in practice will of course depend heavily on the hardware and the use case, and we can't do an extensive analysis. Instead, we would like to set up a simple experiment finding out the time it takes to do one iteration in the feedback loop using our proof of concept tool for some reasonable example use cases. This will shed some light on the potential speedup of using Reptile.

We assume use cases where all data are already available, but not necessarily annotated[6]. The four use cases we investigate are:

Use case 1. **Change hyperparameters**: The user changes the hyperparameters, and wants to retrain and evaluate with the previously annotated examples. We assume for simplicity that the time it takes to choose some new hyperparameters are negligible.

Use case 2. **Annotate 1 new image per class**: The user will annotate 1 new image per class, in total 5 images, and then retrain and evaluate.

Use case 3. **Annotate 5 new images per class**: The user will annotate 5 new images per class, in total 25 images, and then retrain and evaluate.

Use case 4. **Annotate 20 new images per class**: The user will annotate 20 new images per class, in total 100 images, and then retrain and evaluate.

All use cases are performed on FS-VOC. Images from `test-adapt` are annotated and adapted models are evaluated on `test-test`. And as for the experiments in the last section we do 5-way object detection.

Since our tool doesn't do proper annotation, and timing annotation speed would require testing with real users, we use the annotation times from Papadopoulos et al. (2017) and assume three object instances per image on average. This means we assume it takes 21 seconds per image to annotate.

Training time will depend heavily on how many examples you train on, so we run each use case in two versions: 1) No (or very few examples) already annotated.

[6] It's actually quite common that large amounts of unannotated data are available — just not annotations.

	No examples		Many examples	
	Reptile	Normal	Reptile	Normal
Use case 1.	10	1 000	2 000	10 000
Use case 2.	10	1 000	2 000	10 000
Use case 3.	100	1 000	2 000	10 000
Use case 4.	100	1 000	2 000	10 000

Table 6.4: The number of adaptation steps used for the different use cases depending on the method used and the number of annotated examples provided.

2) Most examples are already annotated. For use case 1 we assume 5[7] already annotated images for version 1 and all images in `test-adapt` for version 2, while for the rest we assume 0 and all images in `test-adapt` for the two versions respectively. To keep the experiments simple we only include standard pretraining as baseline. We consider it most natural to compare to normal training since this represents doing nothing few-shot learning related at all.

We run both the server and client locally on a normal workstation. And all use cases are performed with both one and two worker machines — where each worker has a Nvidia 1080Ti graphics card. The local workstation and the workers are on the same local network with 100Mbit bandwidth (so it takes approximately 30 seconds to transfer an adapted model from one worker to another). All use cases are executed once per fold and averaged.

How many steps should we use to adapt? We empirically find that both Reptile and standard pretraining are able to achieve almost the same level of accuracy, but Reptile needs fewer adaptation steps[8]. Therefore we let each model adapt until the accuracy flattens out. In Table 6.4 we present how many steps we do for each use case depending on the number of annotated examples. These number of steps have been found empirically.

All use cases are performed (except the annotation) by us in our web client. Since we rely on many approximate assumptions and manual execution, the measured times will not be accurate down to seconds. Therefore we will only report measured time in minutes.

[7] Since we are only changing hyperparameters and not adding any annotations we have to assume some annotated examples.

[8] As will be shown in Chapter 7, this claim is also backed up by the results of the experiments from the section above.

Chapter 7

Results

7.1 Reptile Evaluation

In this section, we will present and analyze the results of the quantitative experiments we have done.

FS-VOC Benchmark

Figure 7.1 presents the averaged mAP score across all folds. Since the results between folds vary significantly, we also provide the average mAP score for individual folds in Figure 7.2 for completeness. The plots show that Reptile adapts faster than the two baseline models. Perhaps surprisingly, Reptile does not seem to perform considerably better than the baselines after 1000 steps. One might wonder if the faster adaptation can only be attributed to Reptile inheriting the optimizer state from the meta-learning stage, but we tested (results not shown) resetting the optimizer state before adaptation, which did not produce significantly different results. Joint pretraining adapts faster than standard pretraining, but is still considerably slower than Reptile, showing that the inner steps in Reptile is important.

As one can see from Figure 7.2, there is a significant difference in mAP score between the different folds, 0.3 mAP difference between max performance on the *Urban w/o person* fold and the *Indoor* fold. We know from the literature that some VOC classes are more challenging than others. The *Urban w/o person* fold for example contains several classes that are considered easy (e.g. plane and bus). This difference in difficulty highlights the value of testing all classes when experimenting with few-shot learning on datasets with a limited number of classes.

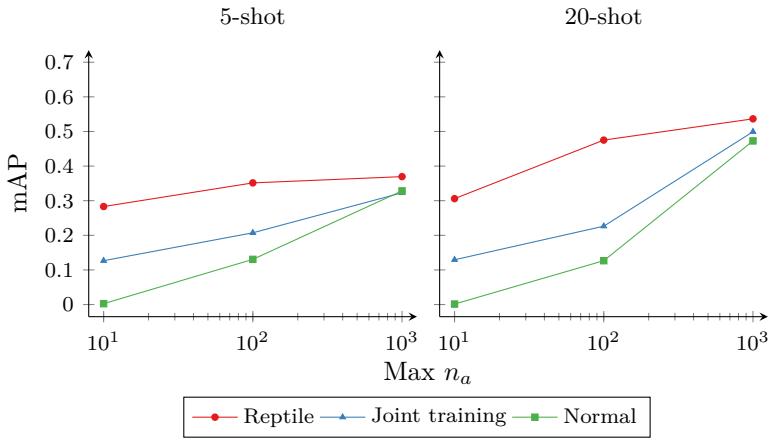


Figure 7.1: Plots showing mAP score for 5-way object detection on FS-VOC. The mAP score is averaged across all four folds for 10, 100 and 1000 adaptation steps. Each evaluation is repeated 20 times.

Many Examples

Figure 7.3 shows how Reptile performs when given many examples. We see that Reptile still adapts faster than the two baselines, and never drop below the baselines in accuracy. This is interesting because it shows that Reptile holds up even when we are not doing few-shot learning. There's no reason to stop using Reptile when the number of examples increase — it still adapts faster. Note that all models continue to improve all the way up to 10000 steps — this is not surprising since we have so many examples that it takes many minibatches to see them all.

Very Few Examples

The performance of Reptile for $K = 1$ through $K = 4$ is shown in Figure 7.4. Although the accuracy is not especially high, we see that Reptile still adapts considerably faster than the two baselines. As for the experiments above, the baselines are able to close the accuracy gap to Reptile given enough steps. Not surprisingly, the accuracy increases across the board along with the number of examples.

FS-COCO & FS-COCO2VOC Benchmark

Unfortunately, we were not able to get any reasonable results when training on COCO images — achieving scores of basically zero. The models behaved normally

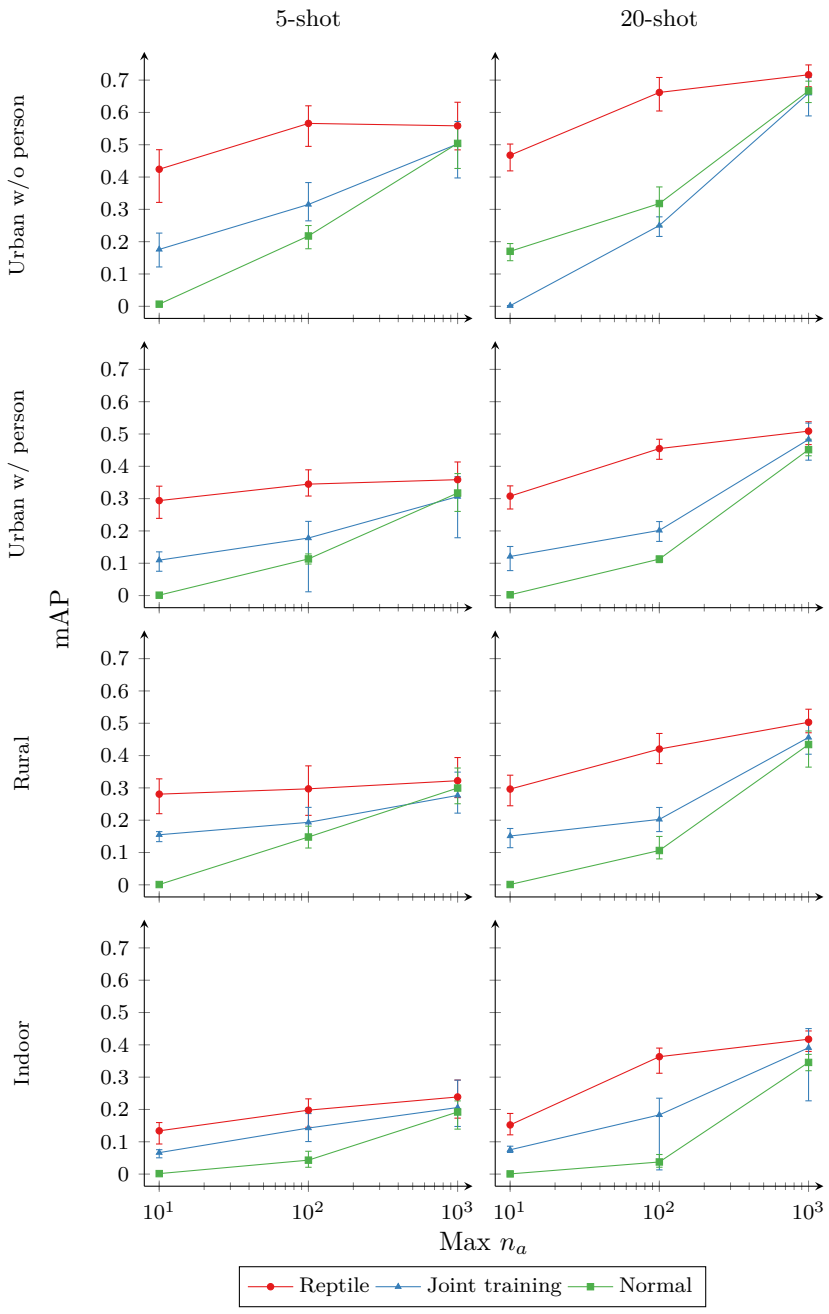


Figure 7.2: Plots showing mAP score for 5-way object detection on each of the FS-VOC folds. Each evaluation is repeated 20 times and the whiskers indicate the maximum and minimum score.

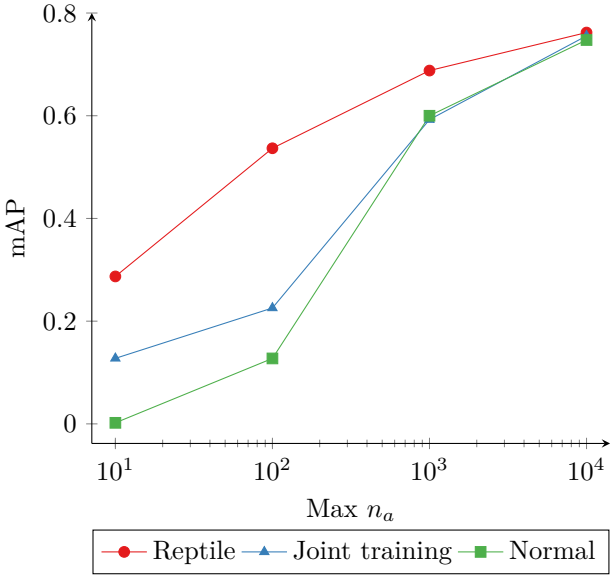


Figure 7.3: Plots showing mAP score for 5-way object detection on FS-VOC. The mAP score is averaged across all four folds for 10, 100, 1000, 10000 adaptation steps. Each evaluation is only performed once.

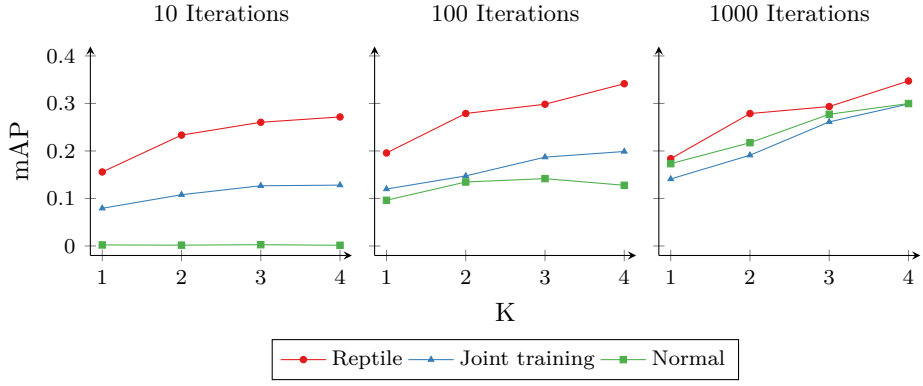


Figure 7.4: Plots showing mAP score for 5-way object detection on FS-VOC. The mAP score is averaged across all four folds for $K = 1$ through $K = 4$. Each evaluation is repeated three times.

during training, and we were not able to find out why adaptation failed. We have three possible explanations:

1. **Not pretrained long enough**: The model might need more iterations when pretraining on COCO images compared to VOC images. We observed that the loss was still decreasing.
2. **Wrong hyperparameters**: Other hyperparameters might be necessary when training on COCO images compared to VOC images.
3. **Too complex**: It could just simply be that object detection on COCO images is too hard given our approach.

Training these meta-learners and running evaluations is a time-consuming[1] process, which requires extensive amounts of compute. We simply do not have the time or the computational resources to investigate this further.

7.2 Proof of Concept Tool Evaluation

We present all the measured results in Table 7.1 rounded to the nearest minute. The first thing we notice is that Reptile is considerably faster in all use cases. As expected, the relative distance to the baseline shrink as the user need to do more work — since the feedback loop then is dominated by the time it takes the user to annotate. Adding an extra worker only has a noticeable effect in use case 1 and 2 when we use Reptile on very few examples. This is not surprising, given that the evaluation only takes a few minutes on FS-VOC. It would probably have a much greater impact on a larger dataset.

In addition to seeing how long one iteration in the user–model feedback loop takes, it's also interesting to see how long the user needs to wait from when the training was started to when the evaluation results are ready. This is the perceived waiting time if the user performs no further actions before the evaluation is known. We present these waiting times in Table 7.2. Here we see clearly that the waiting time is considerably smaller when using Reptile, but it's still far away from being considered real-time.

[1] With the (shared) resources available to us a complete iteration of training a meta-learner and evaluating it takes several days.

		No examples		No examples	
		1 worker	2 workers	1 worker	2 workers
Use case 1.	Reptile	3 min	2 min	25 min	24 min
	Normal	14 min	13 min	114 min	113 min
Use case 2.	Reptile	5 min	3 min	27 min	26 min
	Normal	16 min	14 min	116 min	114 min
Use case 3.	Reptile	13 min	11 min	34 min	33 min
	Normal	23 min	21 min	123 min	121 min
Use case 4.	Reptile	39 min	38 min	60 min	59 min
	Normal	49 min	48 min	149 min	148 min

Table 7.1: The time it takes to do one iteration in the user–model feedback loop for each of the four use cases presented in Section 6.2.

		No examples		No examples	
		1 worker	2 workers	1 worker	2 workers
Use case 1. & 2.	Reptile	3 min	2 min	25 min	24 min
	Normal	14 min	13 min	114 min	113 min
Use case 3. & 4.	Reptile	4 min	3 min	25 min	24 min
	Normal	14 min	13 min	114 min	113 min

Table 7.2: The time it takes from the user requests a model to be trained and evaluated to the evaluation is done for each of the four use cases presented in Section 6.2. In other words the time for one iteration, but without user actions.

Chapter 8

Discussion and Conclusion

We will now go through our research questions and goals while we look back and discuss the results. For each research question we will first assess if we have achieved its goals, before we conclude the question itself.

To answer the Research question 1, we identified two goals. The first goal aimed at finding a suitable few-shot learning method and adapt it to object detection, and then the second goal was to evaluate this method.

Goal 1.1. Choose and adapt a few-shot learning method to object detection.

This goal was mostly addressed in Chapter 4. We first settled on MAML (Finn et al. 2017), but after a substantial effort invested in getting it to work with full-scale object detection on natural images we switched to the similar method Reptile (Nichol et al. 2018b). With Reptile we were able to train object detection on PASCAL VOC images, and was ready to evaluate. We conclude that we reached Goal 1.1.

Note that we didn't show it's impossible to get MAML to work, but at least we have shown that it's non-trivial. We went out of our way to give MAML a fair shot at succeeding, and even then was not able to produce meaningful results. We conclude that it's still an open question if it's possible to apply MAML on large and complex neural networks. One of the things that make MAML hard to train is its use of second-order derivatives. Finn et al. (2017) and Nichol et al. (2018b) suggest that it's not necessary to use second-order derivatives and that a first-order approximation will suffice. Unfortunately, Nichol et al. (2018b) are the first to detail how this can be done and how simple it is (Finn et al. (2017) provide no details), but this was released too late for us to investigate further. Given how similar first-order MAML and Reptile[1] are and the analyses by Nichol et al. (2018b) it's not unreasonable to think it's likely that first-order MAML would give similar results as Reptile.

[1] Remember that Reptile is a first-order method — it does not rely on second-order derivatives.

Goal 1.2. Evaluate performance of the method from Goal 1.1 on standard object detection datasets.

We saw that Reptile is able to adapt considerably faster than standard pretraining no matter how many examples were provided. But our experiments show that Reptile does not achieve much better accuracy if the standard pretrained model is given enough steps. Reptile might have a slight edge over standard pretraining with $K = 5$ examples, but it's hard to reached any final conclusions since the difference is small, the noise is considerable and we haven't tuned hyperparameters for standard pretrained models. Even though we were able to train Reptile on PASCAL VOC images, we were not able to successfully train on MS COCO images. Due to limited time and compute we did not have the chance to investigate further why this is the case. It remains an open question if it's possible to use Reptile on COCO images. We conclude that we have reached Goal 1.2, but our evaluation opens up for more further work.

Nichol et al. (2018b) emphasize that Reptile can adapt fast to new tasks, which our results support. And Finn et al. (2017) emphasize the same for MAML. They both test their methods on MiniImageNet, and are able to significantly outperform the standard pretraining baseline in accuracy. The impression is that these methods not only adapts fast, they are also able to get higher accuracy than the baseline. We were not able to firmly show a considerable difference in accuracy between Reptile and standard pretraining for object detection. The MiniImageNet baseline originates from Ravi & Larochelle (2017), and they used SGD with some fixed number of steps to adapt. Ravi & Larochelle (2017) did not specify the number of steps, but upon inspection of their published code they seem to have used 15 steps. Since they do not use data augmentation, this might be the maximum number of steps they are able to do before overfitting. Our strong standard pretraining baseline might be attributed to better regularization. This is something to investigate in future work.

Research question 1. Can few-shot learning methods be used to do object detection on natural images?

We have explored using MAML and Reptile for object detection. We were not able to use MAML for object detection on natural images, but had more success with Reptile. With Reptile we were able to adapt significantly faster than standard pretraining on new tasks involving PASCAL VOC images, but we were not able to show that Reptile is able to learn from fewer examples. And it remains an open question if on can successfully train Reptile on MS COCO images (i.e. more complex images). We conclude that it's possible to use a few-shot learning method to do object detection, but it's uncertain how much better it works compared to standard pretraining.

For Research question 2 we also had two goals: to make a proof of concept tool and to measure the speedup achieved by using Reptile compared to the standard pretraining method.

Goal 2.1. Make a proof of concept tool simulating the user–model feedback loop on object detection.

The tool we created has all the components needed to simulate the user–model feedback loop. A client with a simple user interface, workers to do training and evaluation of the model and a server to manage the process. We therefore conclude that we have achieved Goal 2.1.

Goal 2.2. Use the proof of concept tool to measure if it's reasonable to expect a faster user–model feedback loop when using the method from Goal 1.1.

We executed some plausible use cases, and showed that Reptile is able to speed up the user–model feedback loop. The improvement is most noticeable when user actions are small and one is training on few examples. As user actions grow in size and training involves more examples these steps dominate the time consumption. We conclude that it is in fact reasonable to expect Reptile to speed up the user–model feedback loop, but how much depends heavily on the use case.

We note that the actual perceived feedback latency from the users perspective (i.e. ignoring user actions) is always considerably lower when using Reptile. This is because only training speed is improved. Ideally, Reptile would be able to generalize better with few examples than the baseline so one could annotate fewer examples and get the same accuracy — and thereby save annotation time. But as mentioned above, we were not able to conclude that Reptile makes better use of few examples with respect to accuracy compared to the baseline.

Research question 2. Does the method from Goal 1.1 enable a faster user–model feedback loop?

While Reptile is not able to reduce the number of annotations relative to standard pretraining, it is able to reduce the training time considerably. The workload on the user is the same, but the user receives feedback from training and evaluating a model faster. We therefore conclude that Reptile enables a faster user–model feedback loop, but speedup depends on the use case.

Implications and Future Work

Standard training procedures often last several days, which is both slow and potentially costly. With deep learning models being applied in an increasing number of applications, it lies great value in speeding up the user–model feedback loop. We have shown that recent advances in few-shot learning might have the potential to do exactly that.

Since we're not able to train Reptile on MS COCO images, it's still unclear if it's possible to do object detection on more complex images with Reptile. Further investigation is needed to uncover whether we need longer training time, different hyperparameters or if it's just not possible with our method of choice.

Reptile is able to adapt with fewer steps, but we could not conclude that it makes better use of few examples compared to standard pretraining. Upon a closer look we see that existing work on few-shot image classification seems to adapt their baselines with relatively few steps. Whether standard pretraining is able to perform on the same level as Reptile or not, given more steps and appropriate regularization, is a question for future work.

Bibliography

Ali, Karim, David Hasler & Frangois Fleuret (2011). "FlowBoost—Appearance learning from sparsely annotated video". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Andrychowicz, Marcin et al. (2016). "Learning to learn by gradient descent by gradient descent". In: *Advances in Neural Information Processing Systems (NIPS)*.

Bazzani, Loris et al. (2016). "Self-taught object localization with deep networks". In: *IEEE Winter Conference on Applications of Computer Vision (WACV)*.

Bengio, Samy et al. (1992). "On the optimization of a synaptic learning rule". In: *Optimality in Artificial and Biological Neural Networks*.

Bengio, Yoshua, Samy Bengio & Jocelyn Cloutier (1990). *Learning a synaptic learning rule*. Université de Montréal, Département d'informatique et de recherche opérationnelle.

Bietti, Alberto (2012). *Active learning for object detection on satellite images*. Tech. rep. California Institute of Technology.

Bilen, Hakan, Marco Pedersoli & Tinne Tuytelaars (2014). "Weakly supervised object detection with posterior regularization". In: *British Machine Vision Conference (BMVC)*.

Chen, Hao et al. (2018). "LSTD: A Low-Shot Transfer Detector for Object Detection". In: *arXiv preprint arXiv:1803.01529*.

Chen, Tianqi et al. (2015). "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274*.

Chollet, François et al. (2015). *Keras*. `https://github.com/fchollet/keras`.

Cohen, Gregory et al. (2017). "EMNIST: an extension of MNIST to handwritten letters". In: *arXiv preprint arXiv:1702.05373*.

Collobert, R., K. Kavukcuoglu & C. Farabet (2011). "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, Advances in Neural Information Processing Systems (NIPS) Workshop*.

Cormen, Thomas H. et al. (2009). *Introduction to algorithms*. MIT press.

Deng, Jia et al. (2009). "ImageNet: A large-scale hierarchical image database". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Dong, Xuanyi et al. (2018). "Few-Example Object Detection with Model Communication". In: *arXiv preprint arXiv:1706.08249v6*.

Fei-Fei, Li et al. (2003). "A Bayesian approach to unsupervised one-shot learning of object categories". In: *IEEE International Conference on Computer Vision (ICCV)*.

Fei-Fei, Li, Rob Fergus & Pietro Perona (2006). "One-shot learning of object categories". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 28.4.

Finn, Chelsea, Pieter Abbeel & Sergey Levine (2017). "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *International Conference on Machine Learning (ICML)*.

Finn, Chelsea & Sergey Levine (2018). "Meta-Learning and Universality: Deep Representations and Gradient Descent can Approximate any Learning Algorithm". In: *International Conference on Learning Representations (ICLR)*.

Girshick, Ross (2015). "Fast R-CNN". In: *IEEE International Conference on Computer Vision (ICCV)*.

Girshick, Ross et al. (2014). "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Goodfellow, Ian et al. (2016). *Deep learning*. Vol. 1. MIT press Cambridge.

Graves, Alex, Greg Wayne & Ivo Danihelka (2014). "Neural turing machines". In: *arXiv preprint arXiv:1410.5401*.

He, Kaiming et al. (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *IEEE International Conference on Computer Vision (ICCV)*.

– (2016). "Deep residual learning for image recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Hinton, Geoffrey E, Simon Osindero & Yee-Whye Teh (2006). "A fast learning algorithm for deep belief nets". In: *Neural Computation* 18.7.

Hochreiter, Sepp, A Steven Younger & Peter R Conwell (2001). "Learning to learn using gradient descent". In: *International Conference on Artificial Neural Networks (ICANN)*. Springer.

Hornik, Kurt, Maxwell Stinchcombe & Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5.

Ioffe, Sergey & Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning (ICML)*.

Jia, Yangqing et al. (2014). "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093*.

Jones, Eric, Travis Oliphant, Pearu Peterson, et al. (2001–). *SciPy: Open source scientific tools for Python*. URL: `http://www.scipy.org/`.

Joshi, Ajay J, Fatih Porikli & Nikolaos Papanikolopoulos (2009). "Multi-class active learning for image classification". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Kao, Chieh-Chi et al. (2018). "Localization-Aware Active Learning for Object Detection". In: *arXiv preprint arXiv:1801.05124*.

Keren, Gil et al. (2018). "Weakly Supervised One-Shot Detection with Attention Siamese Networks". In: *arXiv preprint arXiv:1801.03329*.

Kingma, Diederik P & Jimmy Ba (2015). "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations (ICLR)*.

Kirkpatrick, Scott, C Daniel Gelatt & Mario P Vecchi (1983). "Optimization by simulated annealing". In: *Science* 220.4598.

Koch, Gregory, Richard Zemel & Ruslan Salakhutdinov (2015). "Siamese neural networks for one-shot image recognition". In: *International Conference on Machine Learning (ICML), Deep Learning Workshop*. Vol. 2.

Krizhevsky, Alex, Ilya Sutskever & Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems (NIPS)*.

Lake, Brenden M, Ruslan Salakhutdinov & Joshua B Tenenbaum (2015). "Human-level concept learning through probabilistic program induction". In: *Science* 350.6266.

LeCun, Yann (n.d.). *The MNIST database of handwritten digits*. `http://yann.lecun.com/exdb/mnist/`.

LeCun, Yann, Yoshua Bengio & Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553.

LeCun, Yann et al. (1990). "Handwritten digit recognition with a back-propagation network". In: *Advances in Neural Information Processing Systems (NIPS)*.

Lemke, Christiane, Marcin Budka & Bogdan Gabrys (2015). "Metalearning: a survey of trends and technologies". In: *Artificial Intelligence Review* 44.1.

Li, Ke & Jitendra Malik (2017). "Learning to optimize". In: *International Conference on Learning Representations (ICLR)*.

Li, Zhenguo et al. (2017). "Meta-SGD: Learning to Learn Quickly for Few Shot Learning". In: *arXiv preprint arXiv:1707.09835*.

Lin, Tsung-Yi et al. (2014). "Microsoft COCO: Common objects in context". In: *European Conference on Computer Vision (ECCV)*. Springer.

Lin, Tsung-Yi et al. (2017a). "Feature Pyramid Networks for Object Detection". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Lin, Tsung-Yi et al. (2017b). "Focal Loss for Dense Object Detection". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Liu, Wei et al. (2016). "SSD: Single shot multibox detector". In: *European Conference on Computer Vision (ECCV)*. Springer.

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. URL: `https://www.tensorflow.org/`.

McCulloch, Warren S & Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4.

Mihalcik, David & David Doermann (2003). *The design and implementation of ViPER*. Tech. rep. University of Maryland.

Mishra, Nikhil et al. (2018). "A simple neural attentive meta-learner". In: *International Conference on Learning Representations (ICLR)*.

Mitchell, Thomas M. (1997). *Machine Learning*. McGraw-Hill, Inc.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540.

Nair, Vinod & Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *International Conference on Machine Learning (ICML)*.

Nichol, Alex & John Schulman (2018a). "Reptile: a Scalable Metalearning Algorithm". In: *arXiv preprint arXiv:1803.02999v1*.

Nichol, Alex, Joshua Achiam & John Schulman (2018b). "On First-Order Meta-Learning Algorithms". In: *arXiv preprint arXiv:1803.02999v2*.

Oquab, Maxime et al. (2015). "Is object localization for free?-weakly-supervised learning with convolutional neural networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Pan, Sinno Jialin & Qiang Yang (2010). "A survey on transfer learning". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 22.10.

Papadopoulos, Dim P et al. (2017). "Extreme clicking for efficient object annotation". In: *IEEE International Conference on Computer Vision (ICCV)*.

Ravi, Sachin & Hugo Larochelle (2017). "Optimization as a model for few-shot learning". In: *International Conference on Learning Representations (ICLR)*.

Redmon, Joseph & Ali Farhadi (2017). "YOLO9000: Better, Faster, Stronger". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

– (2018). "YOLOv3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767*.

Redmon, Joseph et al. (2016). "You Only Look Once: Unified, real-time object detection". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ren, Shaoqing et al. (2015). "Faster R-CNN: Towards real-time object detection with region proposal networks". In: *Advances in Neural Information Processing Systems (NIPS)*.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6.

Roy, Soumya, Vinay P Namboodiri & Arijit Biswas (2016). "Active learning with version spaces for object detection". In: *arXiv preprint arXiv:1611.07285*.

Rumelhart, David E, Geoffrey E Hinton & Ronald J Williams (1985). *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science.

– (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088.

Santoro, Adam et al. (2016). "Meta-learning with memory-augmented neural networks". In: *International Conference on Machine Learning (ICML)*.

Schmidhuber, Jürgen (1992). "Learning to control fast-weight memories: An alternative to dynamic recurrent networks". In: *Neural Computation*.

– (2015). "Deep learning in neural networks: An overview". In: *Neural Networks* 61.

Sener, Ozan & Silvio Savarese (2018). "Active Learning for Convolutional Neural Networks: A Core-Set Approach". In: *International Conference on Learning Representations (ICLR)*.

Settles, Burr (2010). *Active learning literature survey.* Tech. rep. University of Wisconsin – Madison.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587.

Simonyan, Karen & Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556.*

Sivaraman, Sayanan & Mohan M Trivedi (2014). "Active learning for on-road vehicle detection: A comparative study". In: *Machine Vision and Applications* 25.3.

Snell, Jake, Kevin Swersky & Richard Zemel (2017). "Prototypical networks for few-shot learning". In: *Advances in Neural Information Processing Systems (NIPS).*

Srivastava, Nitish et al. (2014). "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1.

Thrun, Sebastian & Lorien Pratt (1998). "Learning to learn: Introduction and overview". In: *Learning to learn.* Springer.

Tieleman, Tijmen & Geoffrey Hinton (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2.

Vijayanarasimhan, Sudheendra & Kristen Grauman (2014). "Large-scale live active learning: Training object detectors with crawled data and crowds". In: *International Journal of Computer Vision* 108.1-2.

Vilalta, Ricardo & Youssef Drissi (2002). "A perspective view and survey of meta-learning". In: *Artificial Intelligence Review* 18.2.

Vinyals, Oriol et al. (2016). "Matching networks for one shot learning". In: *Advances in Neural Information Processing Systems (NIPS).*

Vondrick, Carl, Donald Patterson & Deva Ramanan (2013). "Efficiently scaling up crowdsourced video annotation". In: *International Journal of Computer Vision* 101.1.

Walt, S. van der, S. C. Colbert & G. Varoquaux (Mar. 2011). "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2.

Wang, Keze et al. (2017). "Cost-effective active learning for deep image classification". In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.12.

Yuen, Jenny et al. (2009). "LabelMe video: Building a video database with human annotations". In: *IEEE International Conference on Computer Vision (ICCV).*

Zhu, Xiaojin (2008). "Semi-Supervised Learning Literature Survey". In: *Computer Sciences Technical Report.*

Appendix A

FS-COCO Folds

Table A.1 shows which classes belong to `test` for each fold (i.e. C^i_{test} for each fold i). We have given each fold a name to make it easier to remember and distinguish. Note that the name only reflects a general trend, and do not reflect all images (e.g. *Indoor* contains many images of bears outside). In Figure A.2 we show the fraction of images containing a class that are kept in `train` for all four folds of FS-COCO. It's generally harder to find splits in FS-COCO, but the gain of optimizing the splits is much larger than in FS-VOC. We see that the split containing person in `test` is again the most difficult. Figure A.1 show the ratio between `train` and `test`. As for FS-VOC, there's a strong unbalance for the split containing person in `test`.

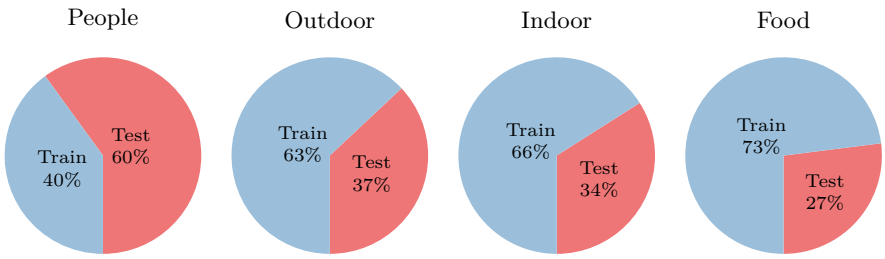


Figure A.1: Plot showing ratio between the number of images in `train` and `test` for each folds in FS-COCO.

People	Outdoor	Indoor	Food
C^1_{test}	C^2_{test}	C^3_{test}	C^4_{test}
person	car	cat	bootle
bicycle	airplane	dog	wine glass
motorcycle	bus	bear	cup
boat	train	chair	fork
bench	truck	couch	knife
backpack	traffic light	potted plant	spoon
umbrella	fire hydrant	bed	bowl
handbag	stop sign	tv	sandwich
tie	parking meter	laptop	broccoli
suitcase	bird	mouse	carrot
frisbee	horse	remote	hot dog
skis	sheep	keyboard	pizza
snowboard	cow	cell phone	cake
sports ball	elephant	book	dining table
kite	zebra	clock	toilet
baseball bat	giraffe	vase	microwave
baseball glove	banana	scissors	oven
skateboard	apple	teddy bear	toaster
surfboard	orange	hair drier	sink
tennis racket	donut	toothbrush	refrigerator

Table A.1: The four different folds of FS-COCO. The classes in `test` are listed for each fold.

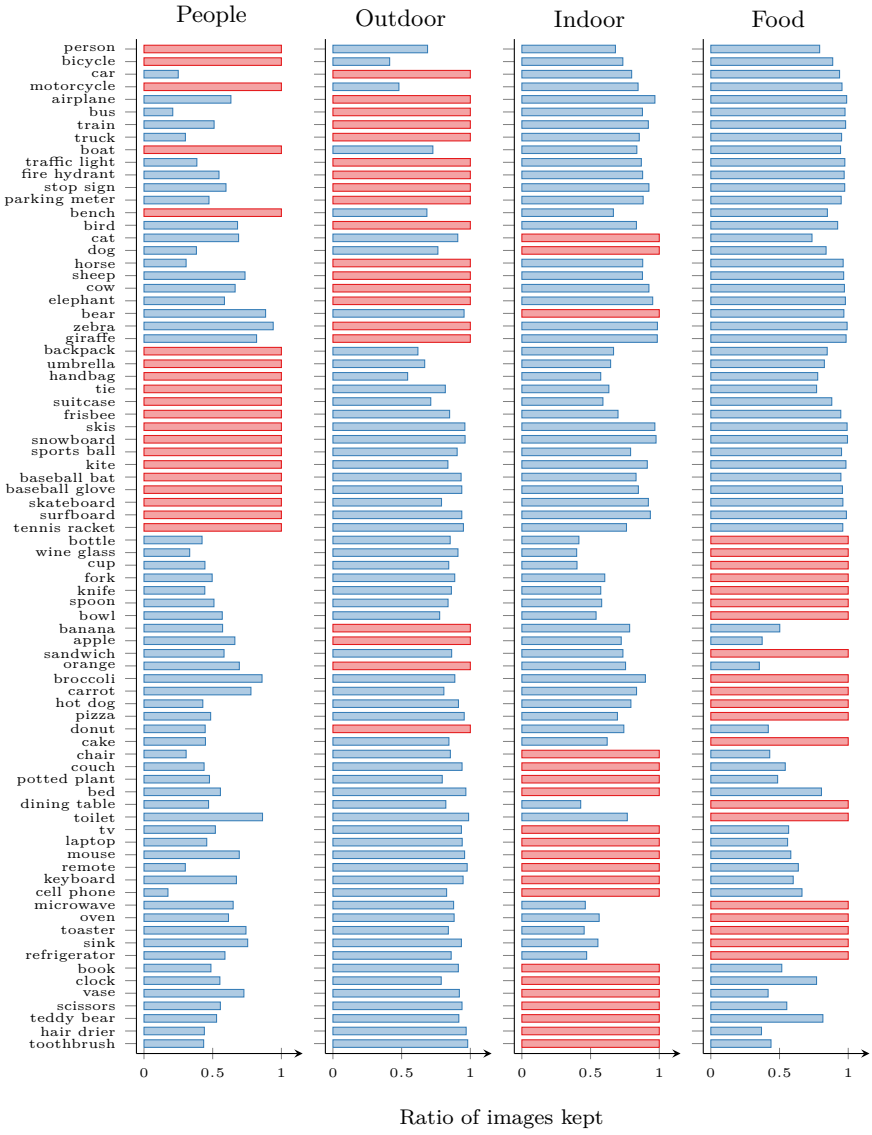


Figure A.2: Plot showing the fraction of images containing a class that are kept in `test` (red) and `train` (blue) for all four folds of FS-COCO. Remember, because of the way we divide the images, `test` always gets all images containing test classes.

Appendix B

Proof of Concept Tool Screenshots

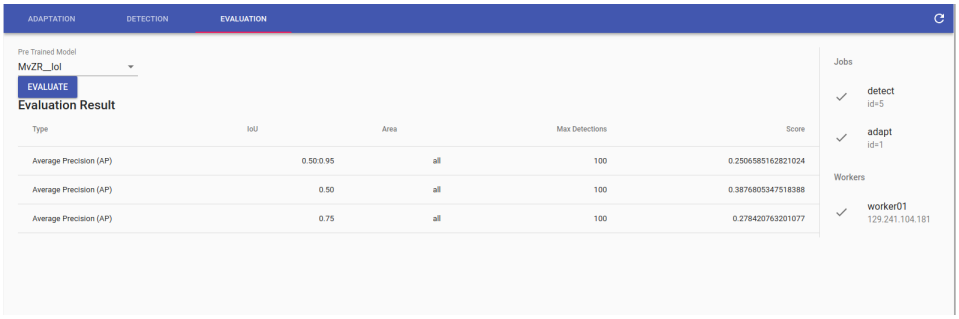


Figure B.1: Screenshot of the web application's Evaluation tab.

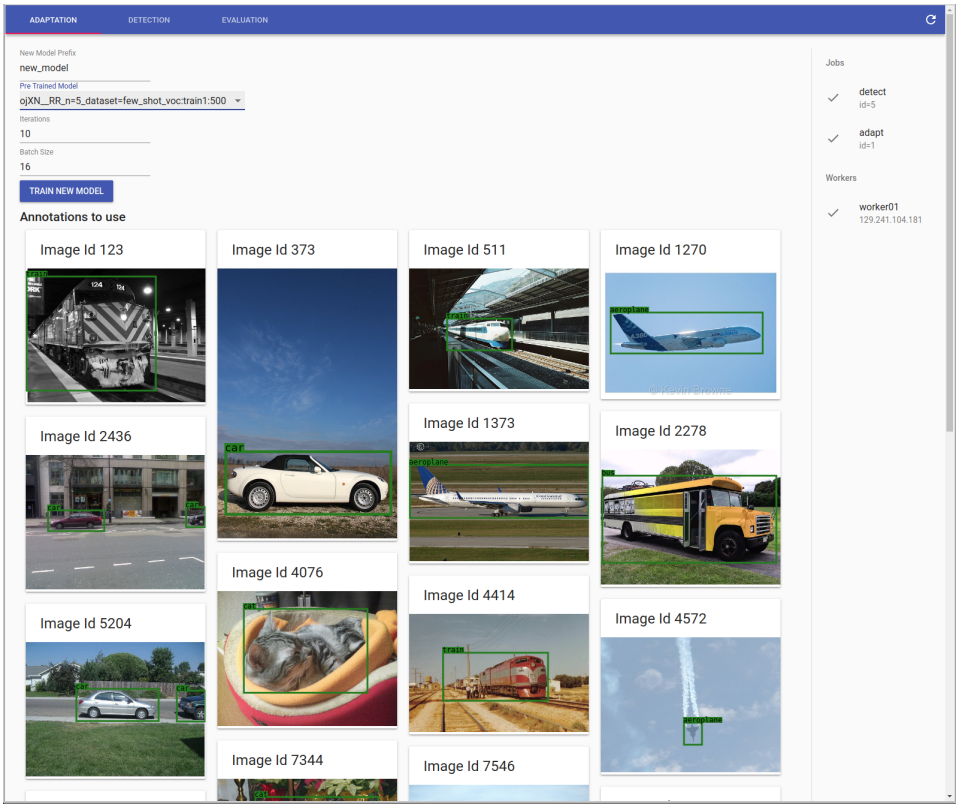


Figure B.2: Screenshot of the web application's Adaption tab.