NTNU
Norwegian University of
Science and Technology

Active One-shot Learning with Memory-Augmented Neural Networks

Andreas Henriksen Kvistad

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Massimiliano Ruocco, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

In this thesis I have explored different strategies and uses of memory-networks in Active Learning and few-shot learning, by using Deep Reinforcement Learning to *learn* an AL agent. Following a period of experimenting and development, I finally ended up with three different Memory-Augmented models. These models have different implementations of memory usage, and are instances of either Long Short-Term Memory networks or Neural Turing Machines. The models are a combination of implementations from state-of-the-art articles addressing the use of Memory-Augmented networks for One-shot Learning and Active Learning. All models are evaluated on the accuracy percentages of label predictions - especially one-shot predictions - versus percentage of label requests on two different datasets. The OMNIGLOT dataset is used for image classification, and the India News Headlines dataset is used for text classification. The results show that all models are capable of learning how to classify both images and text from few examples, while requesting a low amount of labels. They also show that the models struggle more to classify texts, which most likely is caused by sub-optimal feature representations. By varying rewards given during training, the models can be tailored to fit different problems, where varying the penalty for incorrect predictions directly affects the prediction accuracy and the percentage label requests.

All source code for my project and experiments are available at: `https://github.com/andrehk93/Master`

Sammendrag

I denne oppgaven har jeg utforsket strategier of bruk av minne-nettverk innen Aktiv Læring og few-shotlæring, ved å benytte Reinforcement Learningfor å trene opp en AL-agent. Etter en periode med utvikling og testing, endte jeg til slutt opp med tre forskjellige minne-augmenterte modeller. Disse modellene har forskjellige implamentasjoner av minnebruk, and er instanser av enten såkalte Long Short Term Memorynettverk, eller Neural Turing Machines". Modellene er en kombinasjon av implementasjoner fra state-of-the-art artikler som addresserer bruken av minne-nettverk innen "one-shotlæring og AL. Alle modellene er evaluert basert på treffsikkerhetsprosent basert på hvor ofte modellen klarer å predikere riktig kategori på input - og da særlig input den bare har sett en gang tidligere. Dette blir veid opp mot prosentandelen label requests", altså hvor ofte modellen forespør kategori på input gitt istedetfor å forsøke å predikere denne kategorien. OMNIGLOT datasettet for bildeklassifisering er brukt, samt India News Headlines datasettet for tekstklassifisering. Resultatene viser as modellene klarer å klassifisere både bilder og tekst kun fra få eksempler, samtidig som de forespør kun et lavt antall kategorier. Modellene klarer derimot ikke å oppnå like gode resultater på tekst som med bilder, noe som antakeligvis skyldes suboptimale data-representasjoner av tekst. Ved å variere rewardssom blir gitt i løpet av trening av modellene, kan de bli utviklet for å passe til forskjellige formål, siden endring av straffen for en feil prediksjon korrelerer med treffsikkerheten på disse prediksjonene.

All kildekode for prosjektet og dets eksperimenter kan bli funnet på: `https://github.com/andrehk93/Master`

Innhold

Tabeller

Figurer

List of Abbreviations

AI	Artificial Intelligence
AL	Active Learning
ANN	Artificial Neural Network
CMS	Class Margin Sampling
CNN	Convolutional Neural Network
EGL	Estimated Gradient Length
IC	Image Classification
INH	India News Headlines
LRUA	Least Recently-Used Access
LSTM	Long Short-Term Memory
ML	Machine Learning
NTM	Neural Turing Machine
RL	Reinforcement Learning
RNN	Recurrent Neural Network
TC	Text Classification

Kapittel 1

Introduction

Being able to learn rapidly (i.e. from few examples) in machine learning is a complex task, and can be done in many ways, from utilizing computer memory for sheer store-and-produce functionality, to advanced feature representation matching. Such few-shot prediction models can be used in life-long systems (i.e. continuously evolving systems), which makes them suitable for real-life applications with their ability to learn new representations of previously unseen objects. These models are often trained in a *meta-learning* setting, where each classification is enhanced with some relevant meta-information that is learnt during training and could potentially offer context, experience and other helpful data.

1.1 Background

The applications of artificial intelligence is ever increasing, and can be found in areas from image classification of medical images in hospitals [13], to question-answering systems in many companies customer services. Such real-life models should be dynamic and capable of handling both input they've seen many times before, and input they've *never* seen before. With both training and use of AI, there's always and associated cost, whether it's the need of a human expert to validate its decisions - a doctor being asked to look at an X-ray that the model couldn't classify - and especially the human effort in manually annotating large datasets for training. At some point, the cost of employing artificial intelligence can surpass its own performance, and thus wont be needed.

This problem has in conjunction with few-shot learning been addressed by Active Learning (AL) and meta-learning. Instead of standard passivelearning approaches - drawing training samples from an already labelled training set - a method for selecting samples which affect the parameters of a network the most, have been proposed[23, 1, 22]. Another recent approach to AL is described in [2], where meta-learning is employed to *learn* how to learn effectively, and from few examples (few-shot learning) using explicit memory structures capable of learning. In [14], a similar approach capable of one-shot learning combines reinforcement learning (RL) with meta-learning to *learn* an AL-agent, which decides whether to classify an image, or request its label in an episodic setup. This approach will be the inspiration of most experiments in this thesis.

Active learning is usually divided into two categories:

1. **Heuristic Estimates** - By using a heuristic function, this approach tries to estimate how

different training samples impacts the parameters of a model, and select the ones that results in the biggest change. Different functions can be used, depending on the problem, and many have shown improvements over the standard passive learning, which uses randomly selected samples.

2. **Meta-learning** - *Learning to learn*, where instead of using a heuristic function, this approach tries to implicitly learn a strategy for selecting training samples. Some meta-learning models utilize some heuristic function [10], whilst some models tries to learn from a completely blank slate [14, 2]. In regards to AL, meta-learning approaches are often embodied as neural networks themselves, and thus require training and tuning, which in turn can be costly.

In this thesis, the latter category will be experimented with in conjunction with RL to *learn* an active learner agent for few-shot classifications.

1.2 Objectives of the work

In this project, the goal is to implement and experiment with different memory networks for AL, trained with RL for One-shot predictions. All models presented in this thesis are similar to [14], but use different memory structures. As all the models built are of a generic nature, they can be used on different datasets without having to make major changes to the core architecture. The models should all be able to achieve state-of-the-art few-shot predictions for different datasets, and additionally request a low number of labels. By varying the rewards given during training, the balance between prediction accuracy and label request could be changed accordingly, which will be a subject of experimentation. The models will also be augmented with a specialized version of Margin Sampling I call "Class Margin Sampling", which select classes that are more difficult to classify during training, and thus should potentially increase the performance of all models.

For the sake of comparability, the same dataset as in [14] - OMNIGLOT - is used for all models. The OMNIGLOT [16] dataset is a difficult dataset for image classification, with 1623 classes, with a small number of examples per class. This dataset fits the task of few-shot learning with its many classes and few examples. The models are further trained on a textual dataset - the India News Headlines (INH) dataset, which consists of 2.7 million different *headlines* (and 1423 categories) of news articles from an Indian newspaper. By using two different types of datasets, the AL strategy learnt by the models can be evaluated in terms of genericness, and how they adapt to slightly different task setups.

Since the models are trained using RL, visualizing the models' distinctive behaviours is an important tool to help understand how they work independently. Finding the core differences between the models is more in focus than finding the best performing model. Also important is the evaluation of the cross-dataset generality - the modularity - of the models, by using both text- and image-datasets.

1.3 Report overview

In chapter 2 I will review the necessary background information needed to understand the models in this project, and the approaches used. This includes different neural networks, active learning and meta learning, as well as reinforcement learning and Neural Turing Machines.

In chapter 3 the state-of-the-art is presented for active learning implementations, as well as few-shot prediction models and meta-learning approaches.

In chapter 4 the datasets used will be examined, and all experimental models and training challenges will be presented together with both training and data-sampling procedures.

In chapter 5 the experimental setup and the results from experiments will be presented and evaluated. The models presented will be compared with each other on low-shot predictions, as well as rate of label requests and prediction accuracy.

In the last chapter I will summarize the project, and discuss the results which were presented in the project and review possibilities for future work.

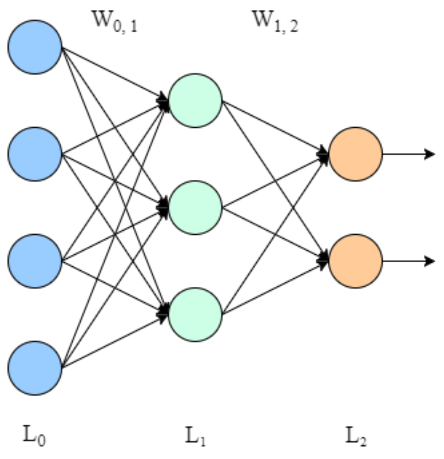
Kapittel 2

Background Theory

2.1 Neural Networks

Neural networks have been around for a while, but due to the lack of computational power and other modern commodities, they haven't been used much before recently. A neural network consists of nodes, structured in connected layers, and are in some way meant to simulate the human brain, were these nodes act similar to neurons. Between each layer of a network there's connections between the nodes, called weights, which usually have an associated value - or weight - that determines the strength of the connection between two nodes in different layers. There's usually *no* connection between nodes in the same layer.

The standard behaviour of neural networks begin with propagating input received at the input layer L_0, through all layers $L_0 - L_N$, and collect the output from the output layer L_N. In a standard feed forward network, it is common to have dense connections, meaning all nodes in two concurrent layers L_i, L_j are fully connected by weights W_{ij}.



Figur 2.1: Illustration of a fully connected Neural Network with 1 hidden layer

Since these layers are fully connected, the activation values can easily be calculated via matrix multiplication, where activations of layer L_k are $X_k = W^T X_{k-1}$. If we have an activation function at layer L_k, this becomes $X_k = F(W^T X_{k-1})$. Some commonly used activation functions are ReLU and sigmoid, as seen in equations 2.1 and 2.2, which produce non-linear outputs, which is usually what we want from these activation functions.

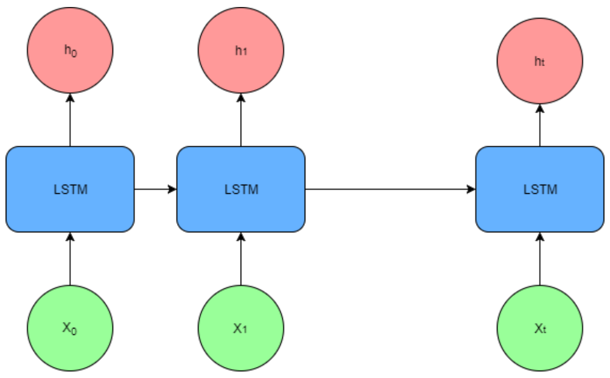
$$f(x) = max(x, 0) \tag{2.1}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

The activations from the last layer can be seen as the output of the network, and are compared with the desired output to estimate the error in the network. Functions for minimizing this error, called loss functions or cost functions, are often specialized to different network types, and help the networks learn the correct mapping from input to output. This is done by propagating the error backwards, through all the weights of the network, and updating these weights with the respective error they produced.

2.2 Recurrent Neural Networks

Recurrent networks are a different implementation of a neural network, and have been successful in different tasks, especially on problems regarding sequence learning. These networks typically have a connection between the output and the next input, called a recurrent connection. This enables the network to keep track of earlier states, often referred to as a form of memory. Having a recurrent connection helps the network capture latent relationships between different input sequences, especially when they are concurrently or closely presented to the network during training.



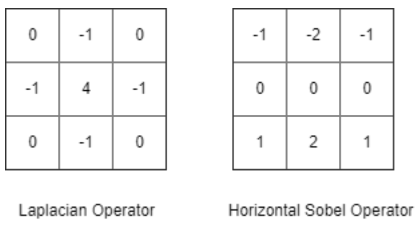
Figur 2.2: Overview of how the hidden state propagates in a LSTM network

Despite their ability to learn from sequences, RNN's have suffered from not being able to learn long term dependencies, especially over long sequences, called the *vanishing gradient problem.* To handle this problem, architectures such as the LSTM (Long Short-Term Memory) network have been introduced. These networks store earlier information observed in cells (often called hidden units), which can both read, erase and write data. These cells decide what information they keep at each timestep, what information that will be allowed further propagation, and what information to block, using different gated units[3]. The LSTM has a recurrent connection from the output of the hidden state h_t to h_{t+1}, which means that h_{t+1} is dependent on h_t as well as the input at that time step, X_{t+1}, as seen in figure 2.2.

2.3 Convolutional Neural Neworks

Convolutional Neural Networks, abbreviated CNNs, have three-dimensional layers, instead of the usual two dimensions we see in standard neural networks and RNNs. The input to a CNN is applied a *convolution*, using a kernel, with a specified size, stride and weights. The output size of the convolution is highly dependent on the parameters of the kernel. The convolution produces local connections between nodes in concurrent layers, instead of fully connected nodes. Having local connections can be advantageous in for example detecting edges in images or word dependencies in sentences.

The convolution is done for the whole input once per kernel (often referred to as a filter) and stride determines the interval between each convolution. The weights in the kernel are multiplied with the weights of the input in their respective locations, and the output of one convolution is a single number at the location of the center of the kernel. This procedure is then repeated for the whole input, and for example with $stride = 1$ does a convolution for *every* input value, and $stride = 2$ does it for every second input value. By using pooling layers the input dimensions can easily be downscaled, still keeping latent information, but reducing the network size and complexity. Max pooling for instance, will quite similar to a convolution use a kernel with weights, but the result of this will be the pixel with the highest value of all pixels covered by the kernel.



0	-1	0
-1	4	-1
0	-1	0

Laplacian Operator

-1	-2	-1
0	0	0
1	2	1

Horizontal Sobel Operator

Figur 2.3: Two different kernels often used for convolution

CNNs have seen extensive use in image classification and object detection due to their ability to capture local dependencies regardless of relative position in the original image. They have also seen some use in natural language processing, often used as input to an RNN.

2.4 Text Embedding

When dealing with NLP tasks, a neural network needs some way of representing text as input. Usually, text embedding is used to produce *dense* vector representations of words - *word embeddings* - or even characters, *character-level embedding* (As I use word embeddings, I will not focus on character-level embedding). By compiling a dictionary of words, each word have an unique index that can be mapped to an embedding, representing the given word. When the input is a sentence, it's usually tokenized, producing a set of words. Usual methods of tokenizing involves removing special characters, stemming, and capital letters, so that the dictionary doesn't contain several versions of *technically* the same word.

The tokens are then sent to dictionary, which returns a one-hot vector encoding of *each* word, comprising by their unique index in said dictionary, and these vectors are in turn used as input to an *embedding layer*. The embedding layer produces dense word representations of each one-hot encoded word, and can be trained in a similar manner as neural networks. One can also use pre-trained word embeddings, albeit that these will not be as specialized for the given task than learnt word embeddings. This could be achieved by allowing updates in the pre-trained embedding layer during training.

The goal of word embeddings is to give each word context and meaning, both individually and combined closely together. To capture such information, word embeddings are usually trained in one of two manners:

- CBOW - which aims at predicting a single word, given its context
- Continuous Skip-Gram - which aims at predicting the context of a single word.

This is known as Word2Vec, and is used for training large-scale word embeddings, which can be used as pre-trained embeddings. The resulting *word vectors* are then used to represent words as input to an ANN.

Correctly trained word vectors will usually have similar dense representations for similar words, or words that are often used in the same context. By visualizing these vectors (by dimensionality reduction or other tools), we can see that similar words are clustered together. Even more interesting is that these embeddings can be capable of capturing relationships between words, and since we are working with vectors, simple vector addition and subtraction could actually produce other word vectors of similar relationship. For example if we look at the relationship between countries and capitals, we would get relationships *across* these relationships:

$$\vec{Country1} - \vec{Capital1} = \vec{Country2} - \vec{Capital2} \tag{2.3}$$

$$\vec{Country1} - \vec{Capital1} + \vec{Capital2} = \vec{Country2} \tag{2.4}$$

2.5 Reinforcement Learning

A natural way of learning from actions, is by receiving positive and negative reinforcements following the action performed. This is an important part of learning for both humans and animals, and is also adapted to ML. The goal in RL is usually to find the *optimal* policy $\pi^*(s_t)$, which is the policy that receives the maximum expected reward for *any* initial state. This is equivalent to being omniscient, which is a fairly unreasonable assumption - at least in the real world - where we usually deal with *partially-observable* environments, meaning that we only have *some* knowledge about the state we're in. Thus we use *estimators* - albeit Q-networks or simple lookup tables - to accumulate rewards based on actions taken in a state, in the search for an estimate for the optimal policy, $Q^*(s_t, a_t)$. A usual RL-setup consists of:

- **Actions:** A finite set of actions possible to perform in any given state.
- **State:** A representation of the world at the given time, that can either be partially- or fully-observable.
- **Rewards:** A finite set of rewards given as a consequence of an action, given a state.
- **Agent:** The performing part, which receives a state, and chooses an action to perform.
- **Environment:** The reacting part which simulates the world, and creates the states based on the actions performed by the agent.

The learning component is usually contained to inside the agent, and is trained on simulated *episodes*, explained in alg. 2.1. By learning over many different episodes, the agent should be able to learn action-values of (s_t, a_t) for any state s_t. As mentioned above, in Deep Learning a function estimator - more precisely an ANN - is used to represent these action-values as $Q(s_t, a_t)$. After observing rewards received after carrying out different actions in different states, these action-values are consistently updated, and should ultimately result in a policy that is an *estimate* of the optimal policy. This type of training is called "Q-Learning", where "Q-values"are representing the action-state values.

Algorithm: RL Value-Function

```
def episode(Env, Agent):
    1. $s_1 = Env.getInitialState$;
    2. for t in range(1, done):
        • $a_t = Agent(s_t)$
        • $r_t, s_{t+1} = Env(a_t, s_t)$
        • $Agent \longleftarrow Reinforce(s_t, a_t, r_t, s_{t+1})$
```

Tabell 2.1: RL value-function algorithm

Most Q-learning models use an exploration-module in order to avoid converging at local maximum. The exploration can be simple functions, like ϵ-greedy exploration, that just choose random actions with a certain probability ϵ in every time-step of an episode, or more complex mathematical functions that usually decrease the probability of exploration over time.

Most modern Q-learning agents are trained using experience-replaymemory, which stores all state transitions (s_t, a_t, r_t, s_{t+1}) which it can access for later use. When preferred, transitions can be collected from memory and then trained on, which helps reduce the temporal connection between episodes in a training batch. This is used instead of training on the episodes most recently finished, and are often used as a technique to also avoid local maxima, to advocate exploration, and for *recycling* of previous transitions (which may or may not be of value, but nonetheless increase the value of a single transition).

2.6 Active Learning

Training deep networks can be a time consuming and costly process, and though an increasing amount of data is available to train on, the cost associated with manually annotating these datasets is high. Supervised learning (passive learning) of neural networks is the most common way to train artificial intelligence, where training examples and their corresponding labels are randomly selected during training. Active learning methods for training artificial intelligence are related to how humans define active learning, as we want to learn more efficiently in both cases, and explores methods other than standard random sampling in training. Using a heuristic function for selecting samples, based on an estimate of how much a sample will change the parameters of the network, will enable the network to learn faster, and hopefully require less data to train on. These heuristics are often trying to quantify the uncertainty in the network given a set of training samples, and train on those it's least certain about.

An active learning setup usually involves an AL agent and an oracle, where the AL agent retrieves unlabelled data, for which it queries the oracle for the label. This is typically done in one of two fashions:

1. **Stream-based AL**, where the agent receives one unlabelled example at a time, and either requests a label for it, or ignores/classifies it.
2. **Pool-based AL**, where the agent are given a large pool of unlabelled examples, from which it tries to choose the most *informative* samples.

Stream-based AL is similar to online learning, and can be used in setups where live-classification has to be made. The oracle can be a human expert, and if the model is uncertain about the label, it could ask the human expert for the label. These models are relevant for datasets where the data is of a sequential order, and when the structure of the sequences are important.

Pool-based AL is often used to accelerate training, and increase performance of classifiers. When given a pool of unlabelled examples, the AL agent needs a way to *rank* each sample without actually requesting the label. There are many ways of ranking samples in such a manner, and they usually measure some sort of uncertainty in the current model, where we usually want to train on the samples we are *least* certain about. Samples of which class we are reasonably certain about doesn't provide that much new information to the model, and will (at least in the early stages) not be trained on. Some pool-based sampling methods are for example:

- **Margin Sampling:** $x_m = argmin_x P_\theta(y_1|x) - P_\theta(y_2|x)$, where y_1 and y_2 are the most probable labels for the sample x, and we train on the sample with the *smallest* margin.

- **Entropy Sampling:** $x_{ent} = argmax_x - (\sum_{i=0}^{n} P_\theta(y_i|x) log P_\theta(y_i|x))$, where n = number of possible labels, and we train on the sample with the *highest* entropy.

2.7 Meta Learning and Few-Shot Learning

Meta-learning in computer science is often used to learn heuristics and hyper parameters for which a lot of tuning to specific problems are necessary. By learning a generalized approach for solving similar problems, meta-learning can be used as a performance enhancer, and help reduce costs associated with training and tuning of machine learning networks. The meta-learning approach is composed of two levels, a learning subsystem which handles the learning within each task, and a generalized learner which learns *across* these tasks.

There have for example been combinations of Meta Learning and AL, where instead of using heuristics for ranking samples (i.e. Margin Sampling), the model tries to *learn* how to rank samples implicitly. In this way, a complex AL-algorithm can be learned for separate different problems and datasets via Meta Learning. Meta-learning models is also frequently used in few-shot learning problems, as their ability to learn high-level information *across* tasks, as well as learning a task, suits most few-shot learning settings.

Few-shot learning (or k-shot learning) is another adaptation from how humans learn, and is based on how we're able to learn a representation only by a few examples (sometimes even from no examples). For example if we are presented with an image of a bird we haven't seen before, and told which bird it is, we could be able to remember which bird it is the next time we see it. Few-shot learning tries to emulate this process, usually by adding memory to a network, or by meta-learning. A memory-augmented network can be trained using back-propagation if the memory is implemented as a differentiable component. The LSTM network is an example of a memory-augmented network, but there are also other implementations which utilizes more explicit memory structures, like the NTM (Neural Turing Machine).

The main difference between few-shot learning and standard classification learning, is usually distinguished in the learning environment and setup. As few-shot learning is more compatible with stream-based learning and online learning (as opposed to pool-based learning and offline learning), it is often trained in a similar setup which emulates these situations. This also means that training on a large number of classes can be difficult, as it would require increasingly more powerful memory structures to handle the task. Thus, few-shot learning often use a small amount of classes, and assigns *pseudo-labels* to every class, so that the memory network refrains from learning image-class bindings. Instead, it learns *how to* store examples with labels in memory, which it then uses to reason about later examples.

2.8 Neural Turing Machine

The Neural Turing Machine (NTM), introduced by [4], is a fully differentiable memory network. This means that the network is trainable by gradient descent from end-to-end. The architecture of the NTM are divided into three components:

- Controller: Input- and output-layer of the NTM, can either be a feed-forward network or an LSTM
- Head: Read- and write-heads of the NTM implemented as a feed-forward network. Write-heads decides what to write to memory and where to write it, while read-heads returns a vector-representation of the memory contents.
- Memory: Represented by a tensor with N slots, each of size M. Can be addressed by keys K given by a head (both read and write).

The standard flow in the NTM starts with some external input given to the controller, which produce a representation of said input, and sends it on to *every* head in the model. Each read head perform a read-operation on the memory, which is a convex combination of each memory-slot $\mathbf{M}_t(i)$ in the memory $\mathbf{M}_t$, seen in equation 2.5.

$$r_t \longleftarrow \sum_i w_t(i)\mathbf{M}_t(i) \tag{2.5}$$

The weight-vector $\mathbf{w}_t$ is produced by the read-head in combination with the input-representation given by the controller, and is normalized so that $\sum_i^N w_t(i) = 1$. This combines two different types of focus, content-based and location-based focus. Content-based focus uses a cosine-similarity measure to compare the key K and its strength β, produced by the controller, with all slots in the memory to create a content-vector $\mathbf{w}_t^c(i)$, seen in equation 2.6.

$$w_t^c(i) \longleftarrow \frac{exp(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(i)])}{\sum_j exp(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(j)])} \tag{2.6}$$

The location-based focus reads from specific memory locations, instead of values. This procedure is more complex, and consists of three different methods. First an interpolation gate consisting of a scalar parameter called $g_t \in (0, 1)$ is used to determine whether to apply content-based addressing, or keep the *previous* weightings (thereby ignoring $\mathbf{w}_t^c$), seen in equation 2.7. Since g_t is a scalar parameter, the interpolation can be learnt during training.

$$\mathbf{w}_t^g \longleftarrow g_t\mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1} \tag{2.7}$$

Secondly the NTM applies a shift to the gated-vector. This allows the model to focus other rows nearby the current focus, and is done similarly to a convolution. This could be done in conjunction with the content-based focus, for example finding a nearby memory-slot *similar* to the content-vector, thus combining both location- and content-based focus. The shift is done by modulo of N, meaning a positive shift from the last memory-slot gives us the *first* memory-slot, seen in equation 2.8.

$$w_t^s(i) \longleftarrow \sum_{j=0}^{N-1} w_t^g(j)s_t(i - j) \tag{2.8}$$

Finally, the shifted weight-vector is *sharpened* in order to reduce blurring of weights which can occur as a consequence of the convolutional shift in the previous step. Thus another learnable scalar parameter is needed $\gamma \geq 1$, which is used to sharpen the shifted weight-vector and thus create the final weight-vector $w_t(i)$, seen in equation 2.9.

$$w_t(i) \longleftarrow \frac{w_t^s(i)^\gamma}{\sum_j w_t^s(j)^\gamma} \tag{2.9}$$

Using the weight-vector $\mathbf{w}_t$, the memory $\mathbf{M}_t$ can be updated. The update includes using an *erase*-gate and an *add*-gate. The erase-gate tells us which memory cells we should erase or keep in each memory-slot, and the add-gate tells us which entries in the weight-vector to store in the memory. The write-step can be seen in equation 2.10.

$$\mathbf{M}_t(i) \longleftarrow \mathbf{M}_{t-1}(i)[\mathbf{1} - w_t(i)\mathbf{e}_t] + w_t(i)\mathbf{a}_t \tag{2.10}$$

2.8.1 Least Recently Used Access

Another way of addressing memory in the NTM can be done by an approach called Least Recently Used Access (LRUA), which is solely a content-based addressing scheme, introduced in [2]. LRUA addressing focuses on locations in memory that are either *most recently* used, or *least recently* used, and writes to these. The only distinction between these two choices is that by default, the least used slot is zeroed out *before* an eventual write, and the most recently used slot is simply updated with new information after a write. Similar to the interpolation gate used in equation 2.7, an interpolation gate is used to determine where to write (i.e. least used location or last used location).

The two options are both weight-vectors, where the last used weight vector is simply the read weights $\mathbf{w}_{t-1}^r$ of the previous time-step, and the least used weights $\mathbf{w}_{t-1}^{lu}$ are a binary matrix pointing to the location of the least used memory-slots. The read-weights are produced as in equation 2.6, but in order to create the least used weights, we require additional vectors called *usage*-weights $\mathbf{w}_t^u$ and *write*-weights $\mathbf{w}_t^w$. Usage weights is an addition of the previous usage weights, decayed by a parameter δ, the current read-weights and the current write-weights as in equation 2.11.

$$\mathbf{w}_t^u \longleftarrow \delta \mathbf{w}_{t-1}^u + \mathbf{w}_t^r + \mathbf{w}_t^w \tag{2.11}$$

These usage weights are then utilized to create the binary least-used weights $\mathbf{w}_t^{lu}$, where an entry is set to 0 only if the corresponding entry in the usage-weights is one of the n^{th} *smallest*, or else it's set to 1. Finally, the interpolation can be applied, as seen in equation 2.12.

$$\mathbf{w}_t^w \longleftarrow g_t \mathbf{w}_{t-1}^r + (1 - g_t)\mathbf{w}_{t-1}^{lu} \tag{2.12}$$

The write-vector is then used to write to memory, in conjunction with the controller key $\mathbf{k}_t$. As noted above, all least used locations are being zeroed out *before* writing to them. Thus the memory is updated as in equation 2.13

$$\mathbf{M}_t(i) \longleftarrow \mathbf{M}_{t-1}(i) + w_t^w(i)\mathbf{k}_t, \forall i \tag{2.13}$$

Kapittel 3

State of the Art

3.1 Active Learning

I start by looking into the heuristics based method for actively selecting samples to train on from a pool of samples, proposed by Zhang et al. [23]. The authors define different heuristics for estimating the impact that different training samples can have on the parameters of a model, in the attempt to learn a representation using less data, and thus have a smaller cost and potentially a better performance.

They first introduce a heuristic function called EGL (Expected Gradient Length), which estimates how much a training sample, with the provided label, changes the current parameters of the model. The process of requesting a label can be viewed as a purchase, where we want the cost to be small and the information gain to be large. They proceed to propose their own version of the EGL specifically made for networks with an embedding layer, which they call EGL-word. It's important to note that the EGL-word function does *not* assume that it knows the label beforehand, which is important for pool-based AL. Instead, the expected gradient length is calculated by measuring the expected embedding gradient of each word in the pool of sentences x_i.

$$\max_{j \in x_i} \sum_k P(y_i = k | x_i; \theta) \| \bigtriangledown J_{E^{(j)}}(\langle x_i, y_i = k \rangle; \theta) \| \tag{3.1}$$

The EGL-word estimate (3.1) is calculated by only the embedding gradients $\bigtriangledown J_{E^{(j)}}$ of the words in the given sentence x_i, thus disregarding all other gradients in the embedding layer, resulting in a faster computation. Depending on the task at hand, they adapt the EGL-word with different features, such as estimating entropy based on how uncertain the model is about the sample for *document* classification, where EGL-word is used for *sentence* classification. [1] et al. use a similar approach, where they employ a deep CNN for both training on informative samples, as well as for pseudo-labelling unlabelled training samples. They use specifically three selection criteria for these samples; least confidence, margin sampling and entropy, where the former is modified to find the high confidence samples for pseudo-labelling. The model is trained on the samples of which associated class it's most uncertain about. This further decreases the cost associated with requesting labels for samples, as the pseudo-labelling allow for automatically annotating unlabelled high confidence samples, without any manual interaction needed.

In [22] et al. they use a Bayesian CNN for image classification on the MNIST dataset, and utilize an acquisition strategy of training samples similar to the EGL. A Bayesian CNN is similar to a standard CNN, but places a prior probability distribution over its parameters, allowing it to reason about its uncertainty regarding its own parameters. They further evaluate different acquisition strategies for pool based sampling, and compare these using both the Bayesian CNN and a standard CNN. The Bayesian CNN outperforms the standard CNN with three different strategies, Max Entropy which maximise the predictive entropy, Var ratios which maximise the variation ratio, and BALD which maximise the information gain in the model parameters (similar to EGL). They explain that because the Bayesian CNN captures the uncertainty over its own parameters, and tries to minimize this uncertainty, it can better estimate the certainty measure in the acquisition strategy.

3.2 Meta Learning Models and Few-shot Learning

3.2.1 Deep Meta Learning

The authors of [21] propose a method for not only learning the hyper-parameters of a classifier - the *learner* - but also learning the *update rule* for training said classifier. They implement a LSTM which serves as the *meta-learner*, which they use to learn a few-shot classification task on a set of different small datasets, $D = (D_{train}, D_{test}) \in \vartheta_{meta-test}$. The learner is then receiving proposed updates, which is a series of learned gradients and losses from the LSTM on D_{train}, which is then supposed to increase the performance of the learner on D_{test}. Following the performance on D_{test}, the loss L_{test} from the learner is again used to update the parameters of the meta-learner, creating a synergistic relationship between the learner and the meta-learner. Additionally, the meta-learner can learn the optimal initial weights of the learner, which corresponds to the initial value of the cell state c_0 in the LSTM. This optimization model outperforms recent state-of-the-art results on 5-shot classification on the ImageNet-dataset, as well as performs equally as good on 1-shot classification.

Meta-learning has been successful in many different few-shot learning settings, and the authors of [9] argues that the potential of Deep Meta-Learning (DEML) is yet to be reached. They proceed with the notion that it's the quality of the data representation that is the shortcoming for meta-learning. and propose a method for aquiring better data representation during learning. By concurrent training of a "concept generator which in this case is a state-of-the-art CNN for image classification - a meta-learner, and a "concept discriminator", they greatly improve on standard meta-learning. The idea is to capture high-level information about the dataset, while simultaneously capturing the meta-level concepts *across* a large number of related few-shot learning tasks. The joint training of the concept generator and the meta-learner pipeline produce a synergistic relationship, which both modules benefits from.

The concept generator and concept discriminator are also trained jointly, which means there are two different pipelines in the model. The concept discriminator is receiving representations from the concept generator, but these representations are from large external (though similar) datasets, and not the few-shot meta-learning task. In this way, the concept generator should learn high-capacity representations, which in turn should help elevate the performance of the meta-learner, by feeding it better representations of the data than for example raw pixels. Since the concept generator can be constantly improved on with more data, the authors argue that their model could be implemented as a life-long learning system. This further supplements their claim of creating a more generic model for meta-learning.

A more specialized version is proposed in [18], where they use a CNN as an embedding function for extracting features from images of the OMNIGLOT- and ImageNet-dataset. These features are then used as input to a bi-directional LSTM network, with read-attention over a support-set S. Their "Matching Netsarchitecture lets it learn multiple classifiers, each specialized to any given support-set S, which is a set of *k* examples of image-label pairs. Then they use a form of meta-learning to train the chosen classifier for one shot learning purposes. As long as the support-set S doesn't grow to big, their model outperforms several state-of-the-art classifiers on the OMNIGLOT dataset.

3.2.2 Memory Augmented Models for Few-shot Learning

The authors in [2] et al. address the problem of few-shot learning in neural networks, and implement their model in a meta-learning environment. They explore how augmenting neural networks with external memory can enhance the rapid learning of input representations, by implementing an NTM (Neural Turing Machine) proposed by [4] et al., which they also enhance with their own module. An NTM use a feed-forward network or a LSTM as a controller, which interacts with an external memory module that allows for both reads and writes of explicit memories. They are implemented as a differentiable network, and are thus suitable for learning.

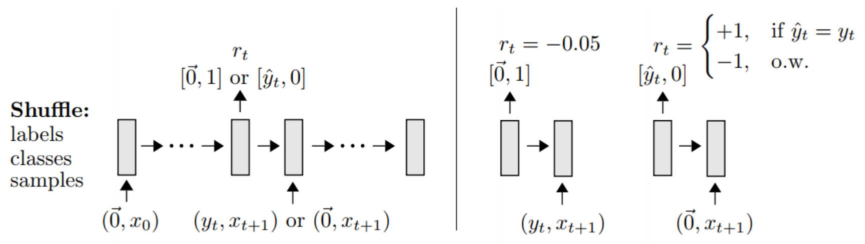
The authors implement an architecture they call MANN (Memory Augmented Neural Network), which they train on episodes of a constant number of random classes, and a total length of either 50 or 100 instances. In each time step, the network is presented with an image from the Omniglot dataset, applied a random rotation and downsized to $20x20$, concatenated with the label of the *previous* image. In this way, the network avoids learning specific sample-target dependencies, and thus have to learn to keep images in memory until the label is provided in the next time step, and then store the image-label pair. As a result of this, the network rapidly learns the representation of these images in every episode, and can output the stored label if it receives a sufficiently similar image.

The authors augments the standard NTM with a LRUA (Least Recently Used Access) module, which either writes data to the *least* used memory location, or the *most recently* used location. They show that enhancing the NTM with a LRUA module result in a significant increase in performance on the second instances of a class prediction and further instances. As expected, the model does almost random guesses on the first instances of the first classes, but then proceeds to very accurately predict the correct class of the second instance of any class, as their representation are stored in memory. This implementation outperforms the basic LSTM network as well as human level one-shot learning for the Omniglot dataset.

Finally, to allow for training on more classes, the authors change the one-hot vector class encoding, to a larger string-based encoding scheme. By uniformly sampling *five* characters from an array of five different characters, they encode every character with their relative position in the array into five 5-bit one-hot vectors. These vectors are then concatenated to produce the final class representation of length 25. This is also done to reduce the probability of a image class receiving the same relative class in different episodes during training, further decreasing the likelihood on overfitting on sample-target bindings. The models were now allowed to train on 15 different classes, where the LSTM performed very poorly, and the LRUA performed almost equally to the one-hot vector encoding of five classes.

3.2.3 Deep Reinforcement Models for Active Learning

A different approach to meta learning is proposed by [14] et al. where they use reinforcement learning to learn an active learning policy for stream based classification. They use a LSTM to act as a function approximator for a Q-network, and the output of the LSTM is connected to a fully connected linear layer, which produces the actual Q-values. Every episode, the model is presented with three different classes of images, and in total 30 images, sampled randomly from these three classes. The setup is similar to the one in [2], except from that the true label is given to the model only if it *requests* it. When the system receives an image, it can either choose to label the image, or request the true label. Both choices impose rewards or penalties on the system, where requests typically give a low penalty, $r_{req} = -0.05$, a correct prediction is rewarded, $r_{cor} = 1.0$, and an incorrect prediction is penalized, $r_{inc} = -1.0$.


Shuffle:
labels
classes
samples
r_t
$[\vec{0}, 1]$ or $[\hat{y}_t, 0]$
$(\vec{0}, x_0)$
(y_t, x_{t+1}) or $(\vec{0}, x_{t+1})$
$r_t = -0.05$
$[\vec{0}, 1]$
(y_t, x_{t+1})
$r_t = \begin{cases} +1, & \text{if } \hat{y}_t = y_t \\ -1, & \text{o.w.} \end{cases}$
$[\hat{y}_t, 0]$
$(\vec{0}, x_{t+1})$

Figur 3.1: Experimental setup in [14], with the associated rewards for each action, and resulting state following the action

By updating the internal hidden state of the LSTM throughout a whole episode, the agent learns to remember image representations and their associated labels it has seen earlier in the episode, and thus with some certainty can output the correct class when a similar image is presented. Penalizing the agent for label requests, forces it to decide based on the cost of possibly incorrectly predicting the label of the image presented, with the cost of requesting a label. Notably, the model doesn't learn the classifications of the images, but rather *how* to learn to classify without having to request the label for every image, or learn their actual class. Their final model reports an accuracy close to [2], both for the first instance of a class in an episode, as well as for the second instance, but require substantially less label requests (especially for the second instance). They also found that training the model with a greater penalty for incorrect predictions, increases the number of requests, but also increases the accuracy of the model, which makes the model flexible to problems where the cost associated with incorrect predictions are high.

In [6] et al. a more generalized model is proposed, where deep reinforcement learning are used to learn an AL policy that generalize over different *datasets*, instead of within a single dataset. To be able to accomodate different datasets in the same model, they use generic embedding layers that maps dataset-dependent features to embeddings. The model includes a meta network, as well as a policy network, and are trained on episodes of mostly unlabelled examples from different datasets (each mini-batch samples a random dataset to train on). Thus, the model needs to learn which samples that improves the performance of the base learner the most, similar to the approach in [23, 1], but also how to generalize over a distribution of datasets. As opposed to the policy learnt in

[14], this model will always receive the label of every sample it uses. Instead it focus on learning a generalized active learning policy across different datasets, and includes a more generic network architecture.

A different approach to AL is proposed in [7], where the AL agent is put between a user and and a black-box QA network. The agents task is to actively reformulate the questions from the user, and query the QA network with the goal of maximizing the networks confidence in the answer, which it returns to the user. The reformulation module is a Seq2Seq network, and is initially trained on multilingual tasks (translating between two languages). They emphasize that there only exists scarce English-English corpora, which is why they use a different approach. Instead, they train on translating from English to Spanish, English to French, but also Spanish to English and French to English. Both the encoder and the decoder should now have learnt all three languages, and should be able to translate between English to English, French to Spanish and Spanish to French. The module that selects the best answer is a CNN, which predicts the best F1 score for each answer. Policy gradient is used to train the RL agent, which is both the Seq2Seq and CNN. Their final model outperforms earlier approaches, and also produce results similar to human performance on the SearchQA dataset, which is extracted from the game show "Jeopardy!".

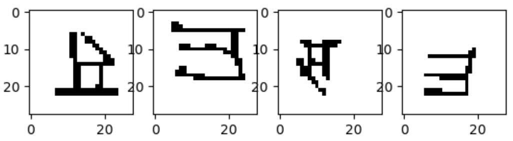
Kapittel 4

Developments and Experiments

4.1 Datasets

4.1.1 OMNIGLOT

The OMNIGLOT dataset [16] is an image classification dataset consisting of 1623 classes of different characters from 50 different alphabets. Each class consist of 20 hand drawn characters, all drawn by different people. The OMNIGLOT dataset is a difficult dataset, with few examples per class, and many challenging classes. It is thus an excellent dataset for one-shot learning models.



Figur 4.1: Processed example images from the Omniglot dataset

The dataset is already split into a training set and an evaluation set, but I manually split the dataset into 1200 training classes and 423 evaluation classes, as in [14] for the DRQN AL model. All images are of the same size, with dimensions 105x105, and are all grayscale images.

4.1.2 India News Headlines

The India News Headlines (INH) dataset is downloaded from kaggle [12], and consists of a compilation of 2.7 million news headlines which has been published by the Indian newspaper Times of India". The articles have been collected over a period of 16 years. The dataset consist of three parts: ID, Category and Headline Text, as seen below.

In total there are 1226 different categories in the dataset, but it is important to note that not all these categories contains enough headlines to satisfy the criteria for our experiments (which usually

ID	**Category**	**Headline**
20010101	sports.wwe	”win over cena satisfying but defeating undertaker bigger roman reigns”
20010102	bollywood	Raju Chacha”
20010102	unknown	Status quo will not be disturbed at Ayodhya says Vajpayee”

Tabell 4.1: Snippet of India News Headline dataset

demands at least 10 instances of each class). As a consequence, the dataset is heavily unbalanced, with some categories only containing *one* single headline, and some categories which consists of over 100 headlines. It is possible to create both training- and test-datasets which consist of categories with exactly 20 headlines for example, but this would greatly reduce the number of categories introduced to the model, which could lead to overfitting. The final dataset I use is constructed using only classes of 10 samples or more, and thus consists of substantially less classes. Additionally, the dataset still consist of a large number of examples with an unknown category, which is not included in neither the training-set nor the test-set.

4.1.3 Reuters

The Reuters dataset is a collection of old news articles from the Reuters news in 1987, sorted on topic into 116 categories. As an AL few-shot dataset, this is a low number of classes, but most articles consist of several paragraphs, which distinguishes this dataset from the INH, which only consists of headlines. This is also a slightly unbalanced dataset, some classes with many examples and some with few, but this shouldn’t be a big problem, as the objective at hand doesn’t involve learning example-class bindings. Some example texts can be seen in table 4.2.

ID	**Category**	**Headline**	**Tokens**
0002892	cocoa	”JACOBS SUCHARD SEES 100,000 TONNE COCOA SURPLUS ZURICH, March 12 - Jacobs Suchard AG expects a world cocoa surplus of around 100,000 tonnes in 1987 compared with a 104,000 tonne surplus in 1986, Jens Sroka, head of commodity buying, told a news conference. The company expects prices to remain at around current levels despite the likelihood of agreement on buffer stock rules at the forthcoming London cocoa talks, and believes market intervention by the buffer stock manager would stabilise prices. Sroka said world coffee prices are expected to remain weak if any international coffee talks fail to produce agreement. Sroka said stagnating consumption and slight overproduction will continue to weigh on coffee prices and he forecast a continued build-up in stocks. ”	’jacob’, ’suchard’, ’see’, ’100’, ’000’, ’tonn’, ’cocoa’, ’surplu’, ’zurich’, ’march’, ’12’, ’jacob’, ’suchard’, ’ag’, ’expect’, ’world’, ’cocoa’, ’surplu’, ’around’, ’100’, ’000’, ’tonn’, ’1987’, ’compar’, ’104’, ’000’, ’tonn’, ’surplu’, ’1986’, ’jen’, ’sroka’, ’head’, ’commod’, ’buy’, ’told’, ’news’, ’confer’, ’compani’, ’expect’, ’price’, ’remain’, ’around’, ’current’, ’level’, ’despit’, ’likelihood’, ’agreement’, ’buffer’, ’stock’, ’rule’, ’forthcom’, ’london’, ’cocoa’, ’talk’, ’believ’, ’market’, ’intervent’, ’buffer’, ’stock’, ’manag’, ’would’, ’stabilis’, ’price’, ’sroka’, ’said’, ’world’, ’coffe’, ’price’, ’expect’, ’remain’, ’weak’, ’intern’, ’coffe’, ’talk’, ’fail’, ’produc’, ’agreement’, ’sroka’, ’said’, ’stagnat’, ’consumpt’, ’slight’, ’overproduct’, ’continu’, ’weigh’, ’coffe’, ’price’, ’forecast’, ’continu’, ’build’, ’stock’
0002857	cotton	”INDIA 1986/87 COTTON EXPORT QUOTA UP 190,000 BALES NEW DELHI, March 12 - India’s raw cotton export quota has been raised by 190,000 170-kg bales to 600,000 bales in 1986/87 ending August, still well below the 1985/86 quota of 1.35 mln bales, Minister of State for Textiles R.N. Mirdha said. State and private agencies contracted to export 1.34 mln bales in 1985/86, he told journalists. But only 433,000 bales were shipped that year, with the rest to be delivered in 1986/87. About 758,000 bales from 1985/86 contracts were shipped up to February 2 in 1986/87. The government will export 600,000 bales of long and extra- long staple cotton in the three years from 1986/87, he said.”	’india’, ’1986’, ’87’, ’cotton’, ’export’, ’quota’, ’190’, ’000’, ’bale’, ’new’, ’delhi’, ’march’, ’12’, ’india’, ’raw’, ’cotton’, ’export’, ’quota’, ’rais’, ’190’, ’000’, ’170’, ’kg’, ’bale’, ’600’, ’000’, ’bale’, ’1986’, ’87’, ’end’, ’august’, ’still’, ’well’, ’1985’, ’86’, ’quota’, ’1’, ’35’, ’mln’, ’bale’, ’minist’, ’state’, ’textil’, ’r’, ’n’, ’mirdha’, ’said’, ’state’, ’privat’, ’agenc’, ’contract’, ’export’, ’1’, ’34’, ’mln’, ’bale’, ’1985’, ’86’, ’told’, ’journalist’, ’433’, ’000’, ’bale’, ’ship’, ’year’, ’rest’, ’deliv’, ’1986’, ’87’, ’758’, ’000’, ’bale’, ’1985’, ’86’, ’contract’, ’ship’, ’februari’, ’2’, ’1986’, ’87’, ’govern’, ’export’, ’600’, ’000’, ’bale’, ’long’, ’extra’, ’long’, ’stapl’, ’cotton’, ’three’, ’year’, ’1986’, ’87’, ’said’

Tabell 4.2: Snippet of Reuters News dataset

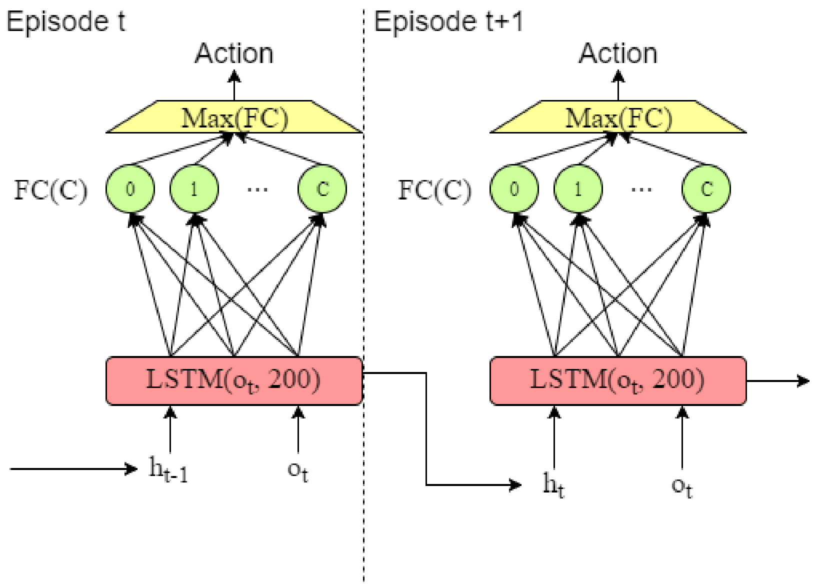
4.2 Experimental Models

In this section I will elaborate on which models I used for my experiments, and how the models were trained and tested. Everything is written in Python, with the PyTorch[19] library for building deep neural networks. PyTorch allows for easy manipulation of many different layers and forward propagation functions, as well as provides support for training on GPUs using the CUDA library. PyTorch is similar to other deep learning libraries as Tensorflow[11] and theano[8], but were chosen as the most suitable platform for the experiments.

4.2.1 LSTM Baseline Model

The baseline model is similar to the one implemented in [14], and consist of a LSTM-network connected to a fully connected linear output layer.

The model trains on short episodes (usually ≤ 50 timesteps), in which it either predicts a class for an image received, or request the true label for it. This means the output space of the model is $C + 1$, where C =number of classes. Before every episode, the hidden state is zeroed. Additionally, the images for the given episode are randomly drawn from the training set, and are given a random slot in a one-hot vector which indicates which class it's associated with *for the given episode*. In other words, the activations applied to the model will be episode specific, and should not force it to learn image-class binding dependencies. There is also no guarantee that there's an equal number of images from every class in an episode, as they are randomly drawn.



Figur 4.2: Experimental setup, with the resulting state following each action

$$Action = \begin{cases} Request & \text{if } Max(FC) = C \\ Predict(Max(FC)) & \text{if } Max(FC) < C \end{cases}$$

Figur 4.3: Actions available for the RL models

The model is a LSTM with 200 hidden units, and a single hidden layer, as seen in figure 4.2. The hidden layer is connected to a fully connected linear layer, which outputs the Q-values. Both the input size and output size depend on the chosen number of classes per episode, with the notation C. The network has an input size of $20 \times 20 + C = 400 + C$, and the output size of the fully connected layer is $C + 1$, where the last node *always* represents the request labelaction. For optimization of the network, I use Adam with default parameters, which tries to minimize the Bellman error in the Q-network, given by the equation below.

$$L(\theta) = \sum_t [Q_\theta(o_t, a_t) - (r_t + \gamma \max_{a_{t+1}} Q_\theta(o_{t+1}, a_{t+1})]^2 \tag{4.1}$$

Here θ represents the model parameters, r_t the reward at time t, a_t the action taken at time t, and o_t the observation at time t. The observation is a concatenation of the previous label, if the model requested it, or a zero-vector of equal size, if the model predicted a class, and the image at the current time step. Equation 4.1 is trying to update the current Q-values of the network, so that the network only perform actions that maximizes the expected discounted future rewards $Q_\theta(o_{t+1}, a_{t+1})$ and thus estimates an optimal policy. The episodic structure predisposes the model to online learning, which together with the capability of one-shot learning can be very cost effective.

$$Reward_t = \begin{cases} Request & r_t = -0.05 \\ Predict & \text{if } \widehat{y} = y_t, r_t = 1.0 \\ Predict & \text{if } \widehat{y} \neq y_t, r_t = -1.0 \end{cases}$$

Figur 4.4: Rewards associated with each action in the RL models

Training a LSTM on sequences of data, requires a modified method for back-propagation called BPTT (Back-Propagation Through Time). Since the hidden layers of the LSTM is reset (zeroed out) at the beginning of every episode, this needs to be accounted for in the optimization, as every action taken is based on the current observation, together with all previous observation in the same episode. There are many different approaches to BPTT, and *all* models use a non-truncated version, which means that every time-step depends on information gained in previous time-steps. This adheres to the task at hand, as the model needs to reason about what it has seen earlier in an episode, given the partial-observability. One fall-back of doing non-truncated back-propagation through time is that the gradients might either vanish or explode, but with short episodes ($T \leq 100$) this shouldn't be a problem.

By using increased penalties for incorrect predictions, the model can be tuned more precisely to problems where the cost of doing an incorrect prediction is high. Naturally, this will most likely result in an increase in label request, thus imposing a different cost to the use of the model. When the punishment for incorrectly predicting a class increases, I expect the training time to increase, or at least use slightly more time in the explorationphase. Using an increased batch-size will be the preferred countermeasure to ensure convergence, and is hence increased during training with higher penalty for incorrect predictions. This improves on the existing generic nature of the model with the capability of more precisely adapting the model to different situations, thus increasing its flexibility.

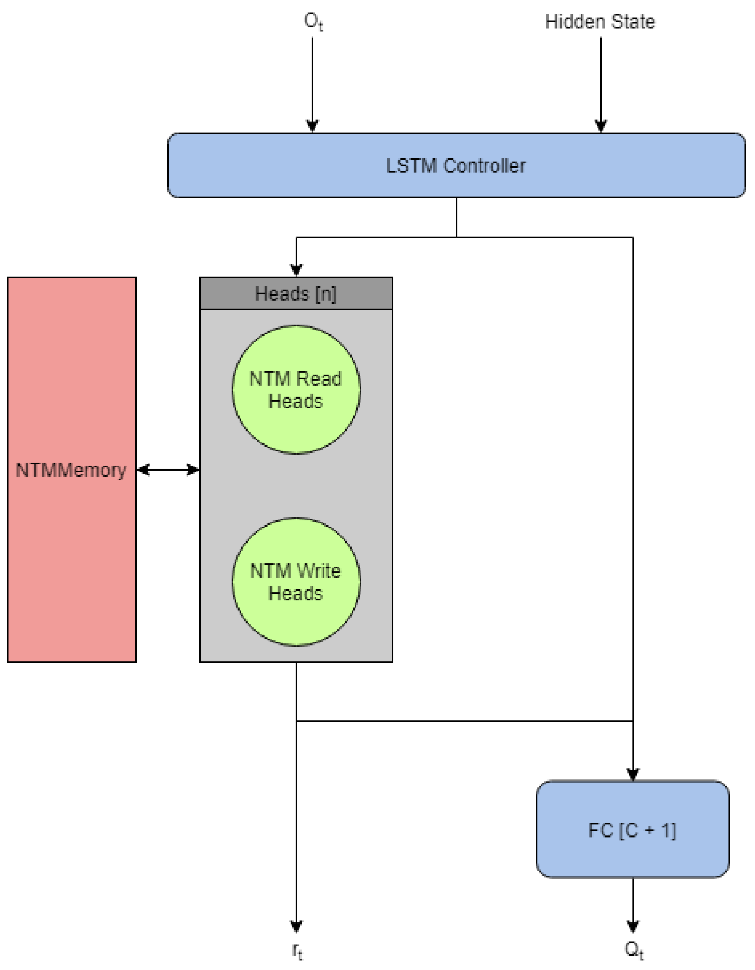
4.2.2 Reinforced NTM

As the LSTM is relying solely on it's internal state for representing the previous states, adding an external more explicit memory-structure could be helpful in increasing the accuracy of the system, similar to what's done in [2]. The first approach implemented is using a basic Neural Turing Machine described in 2.8 as the Q-network, with a LSTM as the memory-controller. As shown in [2], the NTM can surpass the basic LSTM in a similar task setup, especially increasing accuracy on one-shot predictions. The base code for the NTM was collected from [20], which were slightly changed in order to fit the task setup.

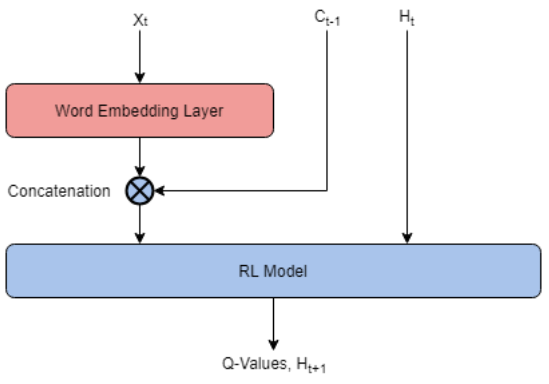
Given that the NTM is a fully differentiable memory-structure, the implementation is similar to the LSTM model, and doesn't require any different task setup than in 4.2.1. For every episode, when the state is given to the NTM, the LSTM controller produces an output which in turn is presented to all the read- and write-heads in the model. The read-heads returns a memory r_t - which is collected as in 4.2 - which together with the output from the controller, serves as input to the final fully connected layer, which produces the Q-values. Note that this is done in a similar fashion as in [2], i.e. without emitting the key-strength vector β, as displayed in the equation below.

$$r_t \longleftarrow \sum_i w_t^r(i) M_t(i) \tag{4.2}$$

The write-heads does as the name suggests write to memory, using the same appraoch as in section 2.8. This operation is equal to the one in [4], using a combination of erase-gates and add-gates to determine what to write from the new memories, and what to erase from the old memories. It is important to note that the write-heads are *not* used when estimating the future discounted rewards - only the read-heads. This is because the model only *simulates* the next state and which Q-values it possibly would produce, and therefore shouldn't write anything to memory in this procedure.



Figur 4.5: NTM architecture for active learning task



Figur 4.6: Embedding extension on all models for text classifying

4.2.3 Reinforced LRUA

The authors of [2] propose a different strategy of writing to memory using an NTM, which they call LRUA (Least Recently Used Access). With LRUA, reading from memory is identical as in the NTM approach, but the write function is different. Instead of only using the read-weights to determine where to write to memory, several additional weight-vectors are introduced. As shown in 2.8.1, we employ a binary weight vector, w_{t-1}^{lu}, which is used in a convex combination with the read weights w_{t-1}^{r} to create the *write weights* w_t^w. The read weights are created as in 2.6, but without emitting the key strength β, similar as in [2]. The write weights are used to write to memory in conjunction with the memory key-strength k_t. Parameters for the LRUA model was collected from [17].

As this impose only minor changes to the NTM, implementation is trivial. It is important to note that these memory structures can become very large matrices, depending on memory size, memory length and batch size. Thus, utilizing efficient methods to create the different weight vectors will be imperative to ensure good performance. Also, the additional weights used in writing-procedures may cause an increased use of memory, which can be overbearing on systems with a small amount of memory, or older GPU's. The architecture will be identical to the one in 4.5.

The LRUA is simply a more specialized version of the NTM, and its features fits our task setup, and thus should ideally produce better results. It's a pure content-based memory writer, and have two main choices when writing:

1. Write to the *least* recently used memory location
2. Write to the *most* recently used memory location

The main difference between the two choices is that the former approach is zeroing out the memory location before writing, successfully replacing the memory, and the latter is *updating* the most recently used memory location with possibly more relevant information. In this way, important information is kept (i.e. information that has been used recently), as well as the memory is constantly updated with new information. Thus for our task setup, the inclusion of new classes will most likely be written to the currently least used slot, whilst samples of already existing slots will either update the most recently used slot (if the previous sample was of the same class), or be written to a least used slot.

It's important to note that there are almost no singular"writes to memory, i.e. these are matrix operations which concerns all memory slots, but some more than other (determined by the write-functionality). Usually, the degree"of which a memory should be written over is determined by the scalar interpolation gate in the NTM (and LRUA), which is a learnable parameter. This means that usually all memory slots receives an update during writing, but these are mostly rather small.

4.3 Complications during Implementation of NTM

The difference between the NTM models implemented in this thesis, and the LSTM used, is the addition of the explicit memory. During experimenting and testing, I found it particularly difficult to deduce whether the implementations was correct, as both the NTM and the LRUA models produced expected results. This is especially for the addressing scheme - the procedure which determines what to read, and what/where to store. Experience shows that the implementation of the addressing scheme could result in only minor changes in the results, which in turn leads to false assumptions.

As the NTM were collected from [20], this mostly concerns the implementation of the LRUA.

It turns out that many different implementations - which in some way are incorrect, or at least not intended - actually produce results that seems reasonable. Given that the writing of memory in the LRUA model is distinctive from the standard NTM, it's also difficult to compare weight matrices, memory slots and results in any way. This proved a challenging barrier, which also was severely time-consuming, as most models used more than a day to train. Carefully designing and implementing these models are greatly recommended.

4.4 Text Datasets

When training on text datasets, it's desirable to do as few changes to the models architecture as possible to maintain its generic nature. Thus I only add an embedding layer to the front of the models, which handles the text inputs. These are of typically word-indexes collected from a dictionary, which gets their respective word-vectors from the embedding layer. This setup allows for both pre-trained word-vectors, or co-trained word-vectors and is used on *all* models for text classification.

4.5 Episode Construction by Class Margin Sampling

To further enhance training performance - possibly reducing training time (which can be of substantial size) - I introduce Class Margin Sampling (CMS). As opposed to standard margin sampling, CMS estimates the margin between $T > 1$ images of the *same class*, for a given number of classes (usually $C_{cms} = C \times 2$). Given that this is a one-shot problem, standard margin sampling would not capture any valuable information. This is because the first time the model receives an observation in every episode, it shouldn't have much bias to which class to assign the observation, which anyhow wouldn't provide much information about the sampled class. By this particular design, all first-instance Q-values provide little but no information about the model. Thus, instead of calculating the smallest margin in a pool of samples, I change the method to better fit our task setup, using a pool of *classes*.

I start by randomly drawing a given number of classes C_{cms} from the training set, which will act as the pool of classes. From each class I draw T samples, which are then processed (as in 5.1.1), and then fed to the model. This is done one class at a time, and when I am done with a class, the memory and hidden state are reset. Each class is also assigned a random label in a one-hot vector before they are fed to the model, to simulate the behaviour in standard episodes. The margin is then calculated based on the *minimum* absolute[1] Q-values generated by the T samples assigned to the random class. This procedure serves to reduce the likelihood of *two* previously occurring problems during training:

1. If an image class is assigned the same random label multiple times, it could start to create inter-episode sample-class bindings.
2. The image classes that are most recognizable or distinguish themselves most from others (given the model's current parameters) doesn't provide optimal information gain during training

The first problem is addressed in [2], where they argue that the NTM and LRUA overfits on the one-hot vector class-encodings, and proceed to use a more robust encoding scheme, greatly reducing this phenomenon. The task structure isn't compatible with a similar scheme, and thus I employ CMS instead. The second problem is usually the reason for using margin sampling, or AL in general, but CMS considers a pool of *classes*, and not samples. This way, CMS will select the image classes that provide the most valuable information, given the models current parameters. Increasing the number of classes drawn C_{cms} could potentially enhance performance further, but will also result in slower data collection, and thus finding an equilibrium will be beneficial.

[1]The absolute value of the Q-values are calculated *after* the maximum values are selected.

```
Algorithm: Class Margin Sampling

def sample($T$, $C_{cms}$, $C$, $Dataloader$, $model$):
    1. $classes = np.random.choice(Dataloader.labels, C_{cms}, replace = False)$;
    2. $margins = []$;
    3. for $c$ in $classes$:
        • $samples_c = np.random.choice(Dataloader.samples[c], T, replace = False)$;
        • $randomLabel = random.randint(0, C - 1)$
        • $marginState = torch.cat((oneHotEncode(randomLabel, C), samples_c[s])$ if $(s > 0)$ else $(\vec{0}, samples_c[s])$ for s in range(len($samples_c$)))
        • $hiddenState = model.resetHidden()$
        • $Q_{cms} = model(marginState, hiddenState)$
        • $margins.append(sum(abs(max(Q_{cms}))))$
    4. return $indexOf(max(margins, C))$;
```

Tabell 4.3: Class Selection by Class Margin Sampling Algorithm

The algorithm above describe how the C classes are selected from C_{cms} margin classes, over T instances of the same sample. The C classes with the lowest margin are then given to the dataloader, which constructs an episode with these classes. Given that this procedure will select the most difficult"images to classify in some sense, this might not result in a speed-up in training, but hopefully increase the general performance of the models.

Since the models are trained by RL, any added bias in the task setup - e.g. conditioning the data sampling procedure - can be viewed as "unnatural"and potentially inhibit the exploration done by the model. I argue that since CMS consider the value of *all* output nodes, the inherit exploration in the model is still maintained.

Kapittel 5

Results and Analysis

5.1 Experimental Setup

All models are trained on personal computer of student, as the required hardware is a decent CPU. Using a more powerful CPU would most likely result in a substantial speed-up, but any test on nVIDIA's TITAN X GPU proved *slower* than on a standard CPU.

- Personal Laptop of the author. Intel Core i7 Quad-core CPU, NVIDIA GTX 970m 3GB GPU and 16 GB system RAM. All models utilizing the CUDA library were run on the GPU, rest on the CPU

Since the RL task setup is nondeterministic (the next time step is dependent on the actions of the previous time step in an episode), I am unable to fully utilize the parallel computations available for LSTM networks in PyTorch. Additionally, LSTM networks doesn't get the same increase in performance on a GPU with small batch sizes and few hidden layers [5] as for example CNN's. As said, the models had in fact reduced performance on the GPU, and were thus trained using the CPU.

The task setup is identical to 5.3.1 for image datasets, and slightly different for text datasets. Similar for both is the episodical stream-based setup, where every episode consist of a *series* of $C * 10$ examples from the datasets. These $C * 10$ examples can only be from C different classes from the dataset, and these classes are *randomly* drawn before every episode, and the $C * 10$ examples are again *randomly* drawn from these C classes. This means that the same class of examples can be drawn into concurrent episodes. This can be an issue, but the classes aren't labelled with their true label, only a pseudo-label randomly assigned when constructing the episode. The pseudo-labels are simply one-hot vectors of size equal to the number of classes drawn.

Action	**Reward**
Request true label	-0.05
Predict **false** label	-1.0
Predict **true** label	1.0

Tabell 5.1: Model possible actions w/rewards

At the initial time-step in every episode, the model receives an example from the dataset, concatenated with a zero-vector of size equal to the number of classes C. The output space of the model can be divided into two choices:

1. Classify the example as one of C possible classes
2. Request the label of the example.

Additionally, the model employs an epsilon-greedy exploration strategy, with $\epsilon = 0.05$ (meaning there's always a 5% probability of choosing a random action). If the model chooses to explore, there are three possible actions, each with 1/3 probability, as described in table 5.2.

Probability	**Action**
1/3	Request true label
1/3	Predict false label
1/3	Predict true label

Tabell 5.2: Epsilon Greedy Exploration strategy

It is important to note that proper task setup is imperative to force the models to learn the specific task at hand. The point of the setup is to expose the model to examples it may or may not have seen before, and thus prompt the model to reason about its own internal representation of the state (i.e. question its own memory). The model should be able to learn how to best handle a complete episode, by storing the examples it has seen, and which label it was associated with, in memory, and later access the memory to reason about any new example that is being presented.

The image- and text-datasets require different pre-processing. All models that use the text dataset also applies an embedding layer, preceding the LSTM input layer, as seen in figure 4.6. The embedding layer and LSTM hidden layer can be of different sizes, and the concatenation of the class one-hot vector is done *after* the text has been embedded, and added to each word embedding in a sentence. This is not a state-of-the-art text embedding procedure, but that is not the intent of the model. The purpose of the model architecture is for it to be generic, and applicable to many different datasets *without* having to change the model drastically.

5.1.1 Image Classification

Every image in the OMNIGLOT dataset is resized to a dimension of 20×20 (original size is 105x105) and every pixel normalized in the interval [0.0, 1.0], and are applied a random rotation of either 0, 90, 180 or 270 degrees (to all images in a class for an entire episode). The script for loading the omniglot images is based on the implementation in [15]. The dataset was then split into disjoint training and validation sets, as seen in the table below.

Dataset	**Nof. classes**
Training	1200
Validation	423

Tabell 5.3: OMNIGLOT Dataset partition sizes

Furthermore, some hyper-parameters are vital to achieving satisfactory performance, and are listed below in table 5.4.

Parameter	**Value**
Batch size	32
Episode size	30
Classes	3
LSTM size	200

Tabell 5.4: Standard parameters for all models, batch size increased to 50 when $r_{inc} = -2.0$

Parameter	**Value**
Nof. Read-heads	4
Nof. Write-heads	1
Memory Length	128
Memory block size	40

Tabell 5.5: Standard parameters for NTM

Parameter	**Value**
Nof. Read-heads	4
Nof. Write-heads	Nof. Read-heads
Memory Length	128
Memory block size	40

Tabell 5.6: Standard parameters for LRUA

5.1.2 Text Classification

With textual datasets, some additional preprocessing are necessary in order to make it work properly for the different models. The dataset will also be split into two disjoint datasets (in terms of categories), but we cannot simply feed words into the model and expect it to know what to do. Instead, we use vector representations of words, which is produced in an embedding layer that precedes the input layer of the different models. The embedding layer then creates word embeddings of a desired size, based on the dictionary in use.

Having a dictionary that can keep a lot of words will be important for the ”India News Headlinedataset, as the headlines are short, and otherwise would produce really sparse representations to feed to the models. Then again, using a large dictionary size might decrease the models ability to make generalizations, especially if the training dataset is small. Hence, for the Reuters-dataset, a slightly smaller sized dictionary could be advantageous. The words kept in the dictionary is the N most frequent words in the dataset, and words that during training isn't represented in the dictionary receives an ”Out of Vocabularytoken (OOV), with a unique index to better distinguish it from other words tokens. The datasets with associated dictionary sizes is described further in table 5.7.

Dataset	**Dictionary Size (N)**	**Training**	**Validation**	**Embedding Size**
Reuters	10 000	92	24	128
INH	20 000	367	92	128

Tabell 5.7: Text Dataset partition sizes (number of classes/categories)

Finally, I must define a sentence size, which is constant for *all* examples collected either in training or validation. This means that if a sample's text exceeds the predetermined sentence size, the rest of the text is simply ignored. The reason for this is a consequence of using LSTM-networks, as they are dependent on homogeneous input matrices *if* they are to be trained in a parallel manner (i.e. batch-training with sequenced input), which is imperative to getting an affordable training time.

While training on text corpora, avoiding using too large dictionaries, word embeddings and sentences will assist in keeping training time on a tolerable level. This can easily be visualized by adding up the resulting matrix dimensions, and also by observing the number of parameters in the model that needs tuning. In table 5.8 I show how the matrix sizes can explode"with careless parameter choices. These dimensions are many times larger than the 400 pixels I train on in image classification, and could be a potential bottleneck for text classification, if the models needs large parameters.

Embedding Size	**Sentence Length**	**Nof. Sentences**	**Matrix size**
64	16	8	8192
128	16	8	16384
200	16	8	25600

Tabell 5.8: Different matrix dimension sizes for text when training on REUTERS-dataset

Given the sequential nature of the LSTM, dealing with sentences can be beneficial when serving each word at a time while incrementing the hidden state for each word. This can capture latent relationships between words, which makes the LSTM networks a powerful tool in text classification. This approach is used when training on text datasets, where each words first receives an embedding, which is then concatenated with the *class-vector* - either the zero-vector or one-hot class encoding of the previous class - and then sent to the LSTM in a sequence. Since PyTorch offers support for sequenced input on LSTM, this is an affordable procedure in terms of performance during training, keeping training times low.

Concatenating *each* word-vector in a sentence with the class-vector could potentially be harmful, where the previous class - if its class was requested - could be overfitted on (inside the episode). This procedure is some sort of a necessary evil, as the task structure requires a class-vector concatenated to each input to the LSTM. Since the zero-vector is reserved for the result of a prediction in the previous time-step, it cant be used as a nothingoperator. Other means could be developed and tested, but due to the lack of training time - and the desire to keep the generic structure of the task - this was deemed unnecessary and thus neglected.

5.2 Training Procedure

All models are trained using the same algorithm, described in 5.9. The underlying interface is PyTorch, which is a ML library for Python. PyTorch offers easy methods for back-propagation, which is the core functionality for learning in ANNs. By manually writing the forward-propagation function for a network using Tensors for representing data, PyTorch stores every operation applied to the relevant Tensors, and automatically create a computation graph, which is then used for back-propagation by simply calling *.backward()* on the model.

One problem regarding training, is that the episodes aren't deterministic, and by using non-truncated BPTT I need to accumulate the error in the network until the batch of episodes are finished. Luckily, this isn't a problem in PyTorch, but could potentially limit the length of episodes, as BPTT shouldn't be used over too many time-steps. If the episode lengths are set too large, using experience-replay memory or other techniques for doing back-propagation would be highly recommended. This would most certainly result in lower performance, as the environment state is only partially observable, and the models are heavily reliant on their memory to represent the state of the environment. By using experience-replay memory techniques, it could be difficult to train the memory correctly, as the hidden state needs to be zeroed before back-propagating, thus erasing the memory of potential previous time-steps in an episode.

Algorithm: Active One-shot Learning

```
def train($Env$, $Model$, $DataLoader$):
  1. $E$, $L = DataLoader.getBatch()$;
  2. $L = 0$; $H_0 = \vec{0}$; $S_0 = \vec{0}$;
  3. for t in range(1, $E$):
       • $X_t, Y_t = E[t], L[t]$;
       • $O_t = concat(S_{t-1}, X_t)$;
       • $Q_t, H_t = Model(O_t, H_{t-1})$;
       • $A_t = Max(Q_t)$ if ($random.random() < \epsilon$) else choose $\epsilon$-greedy action;
       • $R_t, S_t = Env(A_t, O_t, Y_t)$;
       • $O_{t+1} = concat(S_t, E_{t+1})$;
       • $Q_{t+1} = Model(O_{t+1}, H_t)$;
       • $Q_{t+1} = (\gamma Q_{t+1}) + R_t$ if ($t < episodeLength$) else $Q_{t+1} = R_t$;
       • $L$ += $BellmanError(Q_t, Q_{t+1})$;
  4. $L.backward()$;
```

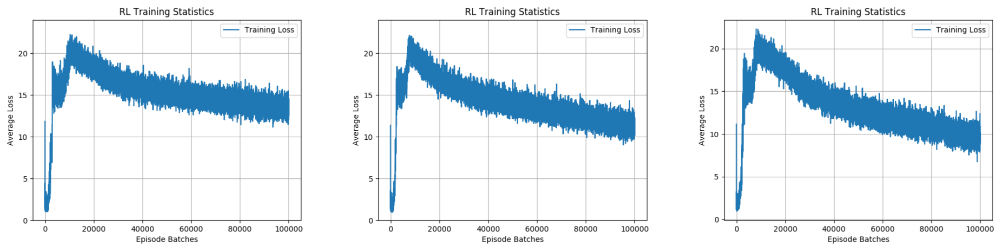
Tabell 5.9: Training procedure for all models, where capital letters indicate the use of batch-training

5.3 Quantitative Results

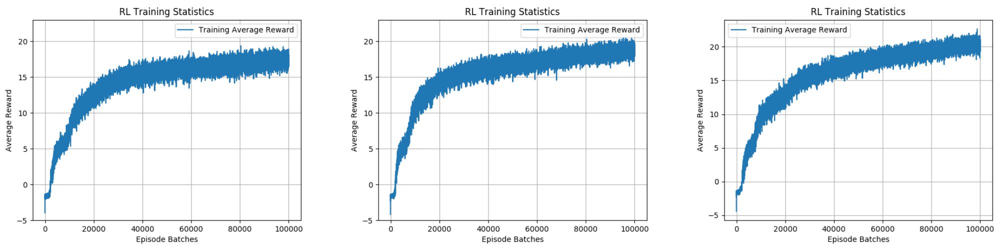
In this section I will present and analyze the performance of the different models. The learnt AL strategies for all models will be evaluated based on the accuracies of classifications, quality of few-shot prediction, and frequencies of label requests. Unless otherwise mentioned, the rewards given are the standard rewards provided in equation 4.4. The models will further be tested on different isolated scenarios in an attempt at visualizing the complexity of the information gained, and how the different models handle different episode structures. The standard models will be first be evaluated on both IC and TC. Then results from models augmented with CMS will be presented, and lastly some experiments with rewards in combination with CMS will be done.

5.3.1 Active Learning for Image Classification

For IC, the models were trained on 100 000 episode batches from the training set, and then evaluated on 10 000 episode batches, on both datasets (training and test). The training process is time-consuming, and have a tendency of more oscillating results than supervised learning approaches. The metric for choosing the best model during training is *highest average reward* gained during an episode batch.



Figur 5.1: Average loss per episode batch during training, for LSTM, NTM and LRUA respectively.



Figur 5.2: Average rewards per episode batch during training, for LSTM, NTM and LRUA respectively.

Training neural networks for Q-learning can be a difficult task, as convergence is slow, and the usual metric for proof"of convergence doesn't behave similarly to as in supervised learning (i.e. loss decreasing towards 0). The reason for this is found in the RL loss function 4.1, the Bellman Error . As the equation states, the loss in the model is the difference between the Q-value of the current state s_t combined with the reward r_t received after performing an action a_t, and the discounted maximum of the next state's Q-value, Q_{t+1}. The goal is to minimize this error, as in many other standard loss functions, but the Q-learning function is more intricate and could potentially produce strange looking results - at first glance - as shown in figure 5.1.

The initial loss almost instantly decrease towards 0, and oscillate around that value for some time. If you're new to reinforcement learning, it could seem that the model has converged, and that it didn't work as expected since it's not getting any good results. Patience is key, and as we can see, the loss suddenly increase drastically, and all models create similar plateaus in the average loss. It's assumed that the increase in loss is a consequence of the simultaneous increase in rewards imposed on the system, given that rewards are the value that determines whether an action should be viewed as bad or good. At $\sim$ 10 000 episode batches, the loss stop increasing and start to slowly shrink instead, whilst the reward still increase. This visualizes that the rewards have been propagated throughout the model, where the model now has sufficiently explored the solution space, and starts to produce realistic assumptions about the future rewards, given the current state. In other words, it has explored and observed rewards for most of the possible states, and can now perform qualified actions where it has to weigh the cost of possibly predicting a wrong class versus the small cost of requesting a label. As this is a meta-learner, it is difficult to pinpoint whether the model has learnt to classify and remember images *inside* episodes, or learnt the more high-level structure of the episodes (e.g. knowing that after receiving several images of *one* class, it is more likely that an image from a different class is presented).

After training, as previously stated, the models are tested on 10 000 episode batches, and on *both* the training- and validation-set. This is to observe any overfitting or unusual behaviour. From table 5.10 and 5.11 we can see that the LRUA are outperforming the NTM and LSTM models on the training set, but all models perform similarly on the validation set. With a drop of 7% in accuracy, it is obvious that the LRUA-model is overfitting in some way on the training set. This is expected behaviour, as the authors of [14] claims their experiments with NTM and LRUA on the same task resulted in overfitting models. The NTM and LSTM produce similar results on both datasets, and both models receive $2-3\%$ lower accuracy on the test set. All models request more labels when provided the test set, which consist of exclusively unseen images, but the difference in label request between the two dataset-partitions aren't of notable size.

The similar results for the LSTM and NTM could be a result of the similar architectures - where the LSTM acts as a controller in the NTM - and the simple addressing scheme in the NTM. Since the NTM are only using a similarity measure for storing and receiving memories, and said similarity measure is accumulated from the LSTM controller, there's a strong correlation between the memories written and read and the representation given by the LSTM controller. In a way the NTM simply adds a storage for the different LSTM states created incrementally every time-step. By this analogy, it would be expected that the NTM would perform better on episodes of greater length - and more classes - given it's augmented memory abilities.

Training set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	82.66	7.99
NTM	83.45	8.28
LRUA	**88.05**	**7.32**

Tabell 5.10: OMNIGLOT: Training set accuracy and request percentage per episode. Accuracies are only calculated from predictions

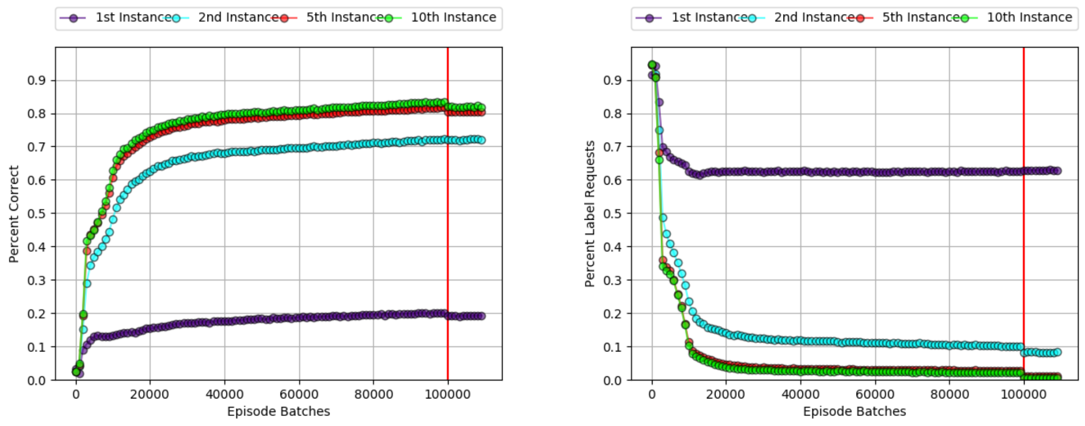
Test set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	80.18	8.09
NTM	80.44	8.45
LRUA	**80.95**	**7.51**

Tabell 5.11: OMNIGLOT: Test set accuracy and request percentage per episode. Accuracies are only calculated from predictions

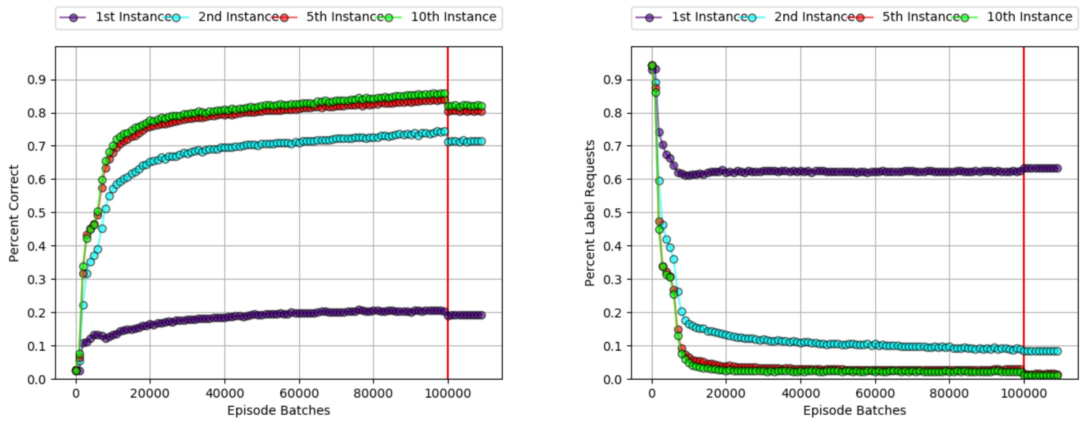
Figure 5.5 shows that the LRUA-models performance on late class predictions - after 1 instance of the same class in an episode - is noticeably reduced on the test set, as well as the percentage label requests are constant. Since the images provided in the test set are of previously unseen classes, it would be generally expected of the models to request slightly more labels, but this is not the case regarding the LRUA model. In fact, this behaviour is barely present in one-shot predictions and other late-class predictions. The AL strategy the models learn should *ideally* be independent of datasets, at least between two disjoint partitions of the *same* dataset, and thus it makes sense that the percentage label request are similar on both the training- and test-set. Any difference could be a consequence of some images being easier to classify, the inherent random structure of episodes, or simply some overfitting on the training images (i.e. sample-class bindings as discussed in 4.5. The LRUA model has a tendency of requesting less labels than both the LSTM and NTM, which could be explained with the same reason for its increased zero-shot prediction accuracy, namely that the LRUA is learning *more* meta-information about the episodic structure than the other models, and thus should normally:

- Request *more* first class-instances in episodes
- Request *less* late class-instances in episodes

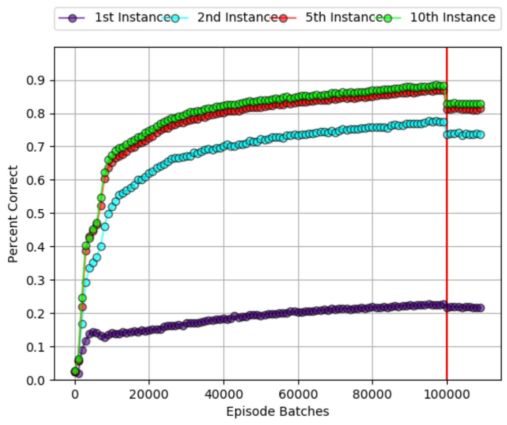
Table 5.13 shows that indeed the LRUA request less labels for late class-instances, but it actually doesn't request more labels for first-class instances (it instead request the least amount of labels on the first class instances), despite *increased* accuracy. This suggests that the LRUA model is learning a better AL strategy for zero-shot classification of images than the other models.

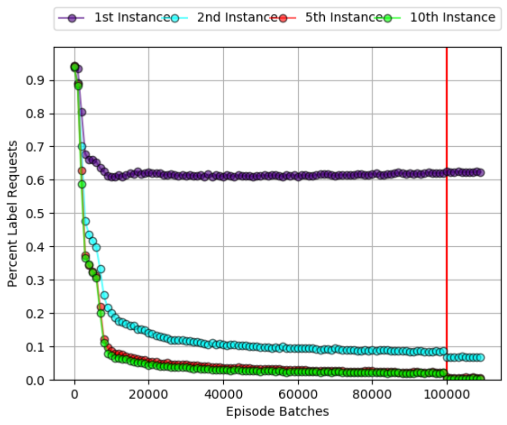


Figur 5.3: Average accuracy and request percentage for k-shot predictions in LSTM. The red line indicates when I stop training and switch to the test set for validation



Figur 5.4: Average accuracy and request percentage for k-shot predictions in NTM. The red line indicates when I stop training and switch to the test set for validation





Figur 5.5: Average accuracy and request percentage for k-shot predictions in LRUA. The red line indicates when I stop training and switch to the test set for validation

Since this is a meta-learning task, it is difficult to deduce why sometimes the explicit memory structures have a bigger discrepancy in performance between the test- and training-set than the LSTM. As seen in previous work, both the NTM and LRUA have outperformed the LSTM model on a similar task, especially when more classes are introduced [2]. Despite this, I can observe that all models have learnt decent meta-level information, given the zero-shot accuracy. Without any *meta-information*, the zero-shot accuracy for all models would be equal to random guessing, with an accuracy of $\sim 1/C$. All models have a zero-shot classification accuracy of $\geq 50\%$ which should only be possible after learning some high-level episode-information. The class-instance prediction accuracies for the first, second, fifth and tenth instance of a class in an episode for each model is listed below, where the best results are highlighted for every instance.

Training set								
	Instance (% Correct)				**Instance (% Requested)**			
Model	1st	2nd	5th	10th	1st	2nd	5th	10th
LSTM	52.3	80.6	84.1	85.2	62.9	7.7	1.2	0.7
NTM	52.8	80.4	84.9	86.3	63.3	7.7	1.4	1.1
LRUA	**59.3**	**85.4**	**89.3**	**90.5**	**62.3**	**6.1**	**0.6**	**0.3**

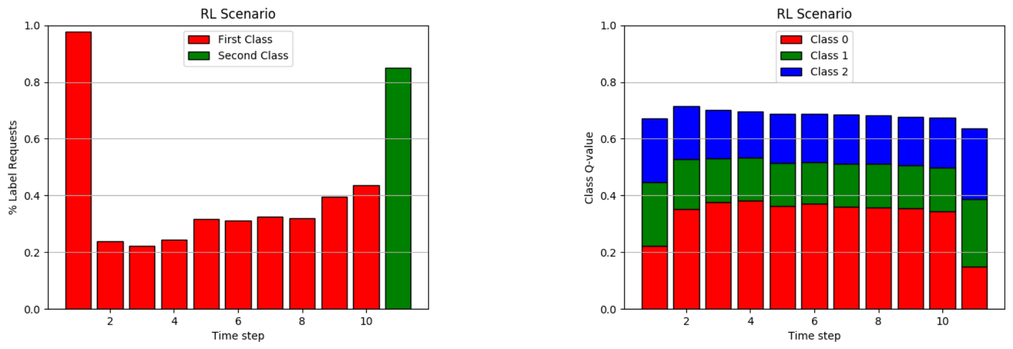
Tabell 5.12: OMNIGLOT: Class instance accuracies on the training set, and percentage label requests made. Accuracies are calculated based on predictions made thus excluding label-requests.

Test set								
	Instance (% Correct)				**Instance (% Requested)**			
Model	1st	2nd	5th	10th	1st	2nd	5th	10th
LSTM	51.6	78.6	81.4	82.4	62.8	8.3	1.2	0.7
NTM	52.5	77.9	81.8	83.0	63.3	8.4	1.6	1.1
LRUA	**58.0**	**79.2**	**81.8**	**83.2**	**62.3**	**6.9**	**0.7**	**0.4**

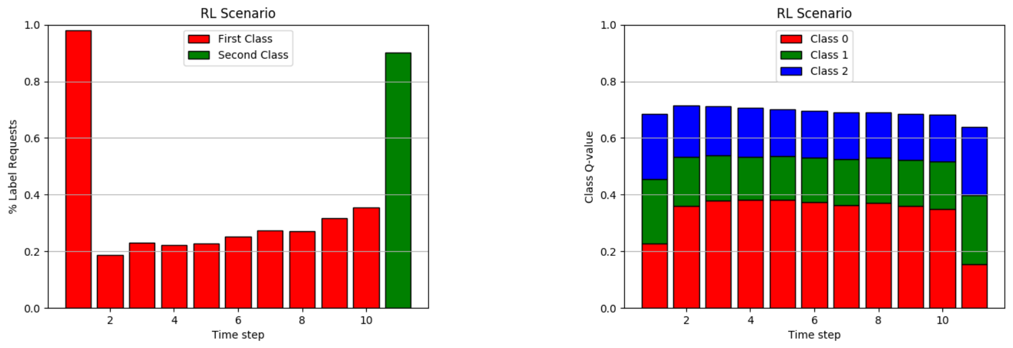
Tabell 5.13: OMNIGLOT: Class instance accuracies on the test set, and percentage label requests made. Accuracies are calculated based on predictions made thus excluding label-requests.

Analyzing Behaviour with Scenarios

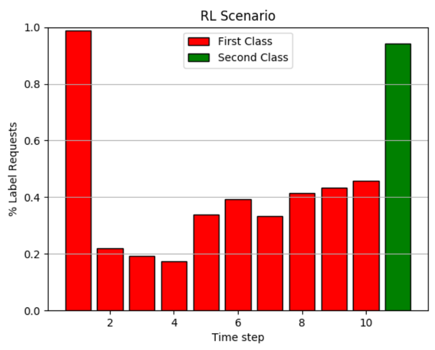
As mentioned above, by only observing class instance accuracy and request frequency, it is difficult to deduce whether the models have learnt any structural information of an episode (i.e. meta-information). Analyzing the behaviour of ANN's is usually an intricate problem, given their BBlack-Box"architecture. By carefully designing the input of ANN's, it is possible to evaluate their behaviour given their output, but its difficult to be certain as we actually dont know exactly how the output is generated. To better understand these models' behaviour, several scenarios (inspired by [14]) are used. First the models are tested on a scenario as in [14], where the model is presented with 10 samples of the same class, and then *one* sample of another class. The percentage of label request per time-step for the different models can be seen below, in the leftmost figures. The rightmost figures displays the evolution of class Q-values per time-step during the scenario.

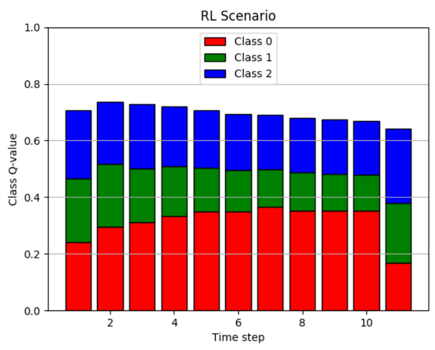


Figur 5.6: Average requests made, and class Q-values for LSTM, where first 10 samples presented is of the same class, and than the next sample is of a different class



Figur 5.7: Average requests made, and class Q-values for NTM, where first 10 samples presented is of the same class, and than the next sample is of a different class





Figur 5.8: Average requests made, and class Q-values for LRUA, where first 10 samples presented is of the same class, and than the next sample is of a different class

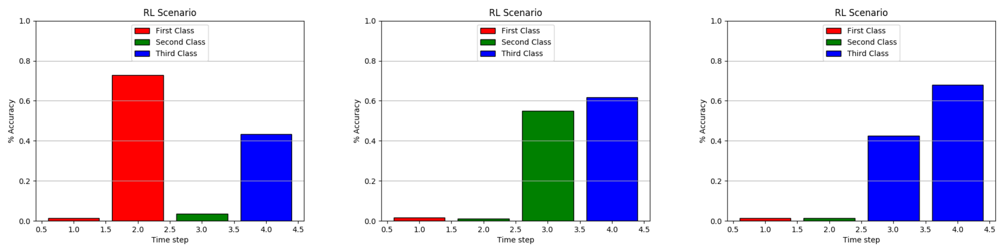
The scenarios are used mainly to show two things:

1. After the model has seen *one* sample from a class, it should be able to recognize it the next time it sees a sample of the same class (i.e. one-shot learning).
2. After the model has seen samples from the same class multiple times concurrently, it should start to expect new classes, resulting in an increasing rate of label requests.

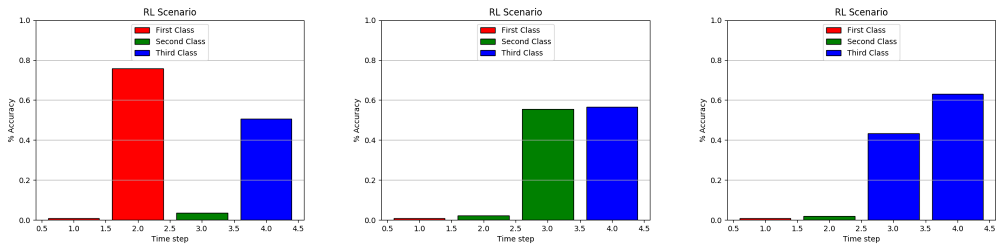
These two behaviours are present for all models, but there is a subtle difference between the LSTM and the NTM models, and the LRUA model. The LSTM and NTM models have a steady increase in label requests per time-step after two time-steps, while the LRUA model have a similar trend, but without the near-continuously increasing label request percentage. By evaluating the class Q-values in the rightmost graphs, we can see that the LRUA model exhibits a bias towards "class 2", even though this class hasn't been seen in the scenario-episode. For example at time-step 10, the Q-values for class 2 is almost twice that of class 1, but at time-step 1 they are almost equal. A reason for this could potentially be that the LRUA model has started to create class-sample bindings *across* episodes, meaning that the same image class - or even classes with similar images - has been delegated the same pseudo-class often enough for this to happen.

The same figure shows that the LRUA reacts differently to the second instance of a class than the two other models, where the Q-values for the corresponding class isn't increased as much. The class Q-value for the samples of the first class doesn't necessarily get reduced over time, despite the increase in label requests, but rather the Q-values of the *other* classes are reduced. This suggests that the models doesn't start to doubt their own prediction performance, but rather the episode they're given.

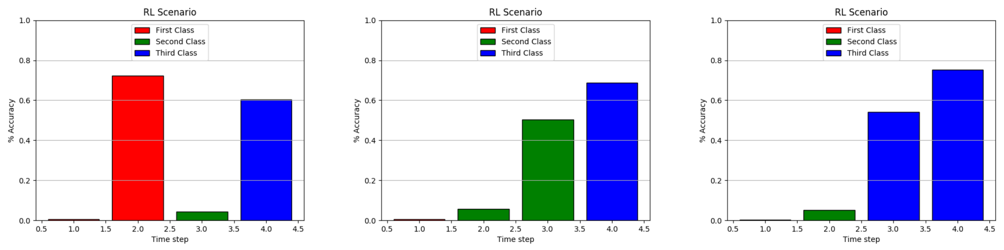
The learnt AL-strategy can further be evaluated for their capability of zero-shot classification, using a different scenario structure. An attempt at displaying how the different models are handling zero-shot classification can be seen below. All three classes are used, where *two* samples from either the first, second, or third class, and *one* sample from both of the other two classes are seen, where the two samples from the same class is seen concurrently.



Figur 5.9: Percentage correctly predicted class (including incorrect prediction for label requests) for the LSTM. From left to right, two instances of the first class, two instances of the second class, two instances of the third class.



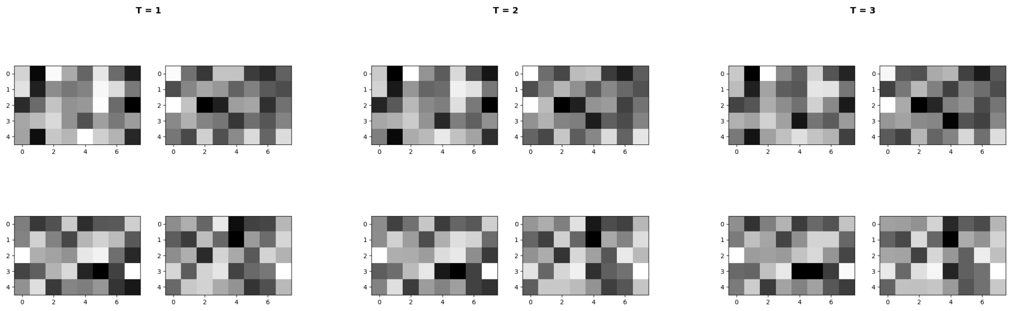
Figur 5.10: Percentage correctly predicted class (including incorrect prediction for label requests) for the NTM. From left to right, two instances of the first class, two instances of the second class, two instances of the third class.



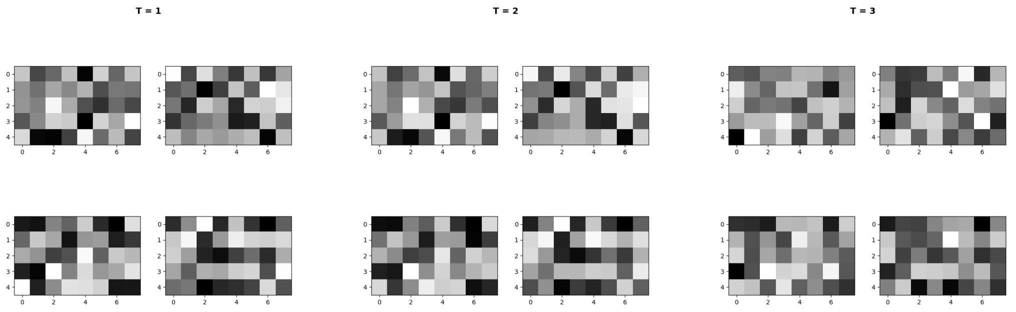
Figur 5.11: Percentage correctly predicted class (including incorrect prediction for label requests) for the LRUA model. From left to right, two instances of the first class, two instances of the second class, two instances of the third class.

Figure 5.11 indicates that the LRUA is better at distinguishing the images from each other, as the LRUA have $\sim 20\%$ better zero-shot accuracy on the *last class introduced*. This is especially apparent when two samples of the first class has been shown, seen in the leftmost graph. The NTM behaves similar to the LSTM, but has the highest one-shot accuracy for when the first and second class has two samples. The request percentage for the zero-shot classification on the first *two* classes are very high, which is why the accuracy are almost negligible. The magnitude of the zero-shot prediction accuracy on the *last* unseen class is much higher, since it's expected from the model to deduce which class it is, and thus will request almost no labels. The LRUA seems to be able to both learn to keep samples from the previously seen classes in memory, as well as aquire sufficient meta-information to expect the last unseen class, better then both the LSTM and NTM.

The reason that the LRUA are consistently doing better on zero-shot predictions, is most likely found in its method for writing to memory - as all results show an increase in these predictions - and I can in good confidence assume that this is not happening by chance. The LRUA have always two choices when writing to memory - which I reiterate - either write to the *most* recently used memory slot, or the *least* recently used memory slot. The most likely procedure for these scenarios, is that the first classes shown is always written to the least recently used memory slot (which is zeroed out), and thus it can maintain a memory of each class without additional interference from other classes.



Figur 5.12: Snapshot from 4 different memory slots in the NTM model, over 3 time-steps.



Figur 5.13: Snapshot from 4 different memory slots in the LRUA model, over 3 time-steps.

The two different procedures from writing to memory are shown above in figure 5.12 and 5.13. While the NTM usually relies on a steady update of *all* memory slots, the latter figure shows how the LRUA allows for large changes of different memory slots, by erasing their respective data before writing the new memories. The memory slot top left receives almost no update the first time step, but looks completely different at $t = 3$. The change in the memory shows that this slot was one of the n least used memory slots, and was therefore erased before a new memory write. This ability of sudden change is implemented in both memory structures by the erase-gates, which allows for some erasing of cells in a memory slot, but is more emphasized with the LRUA's consistent memory wipes derived from the smallest elements of the usage weights $\mathbf{w}_u$.

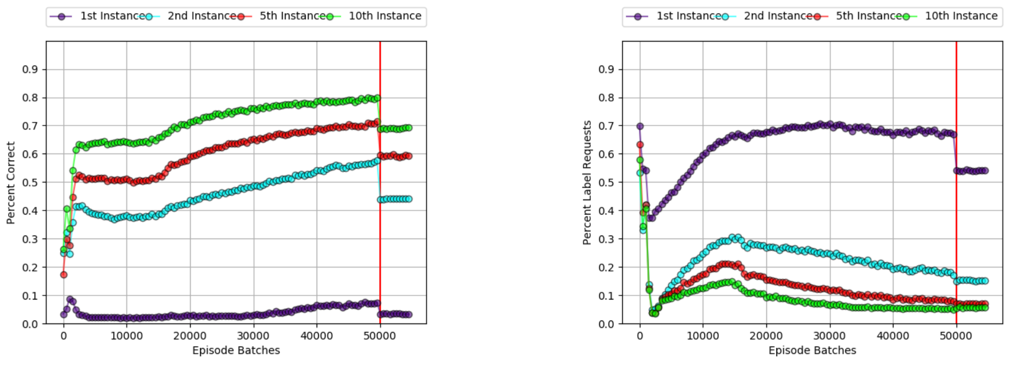
Architectural Predispositions

From the results and performed experiments on IC, general trends and similar behaviours can be observed across all models. Architecturally speaking, all models are similar, only separated by either the application of explicit memory, or *how* they write to the explicit memory. The foundation is created by the LSTM network, either as the model itself or as a memory-controller. A core feature of the LSTM network is the hidden layer of nodes that represents the network state, and is used in correlation with new input to produce the output. Initially, this state is *zeroed*, meaning that it has no previous input to take into account whilst producing the output. The first input after the hidden layer is zeroed will have no interference from any previous state (hidden output), and as shown above, will produce precise changes in Q-value for the class it represents. If the next input is of the same class, the prediction accuracy will be ~ 1. When a different class is presented, the hidden state needs to represent a state where *both* classes are included in the previous state. This seems to be a more difficult task, as one-shot prediction accuracy are substantially decreased when two or more classes are introduced, and shows that different classes causes interference in the representation of the hidden state, which in turn makes for harder predictions.

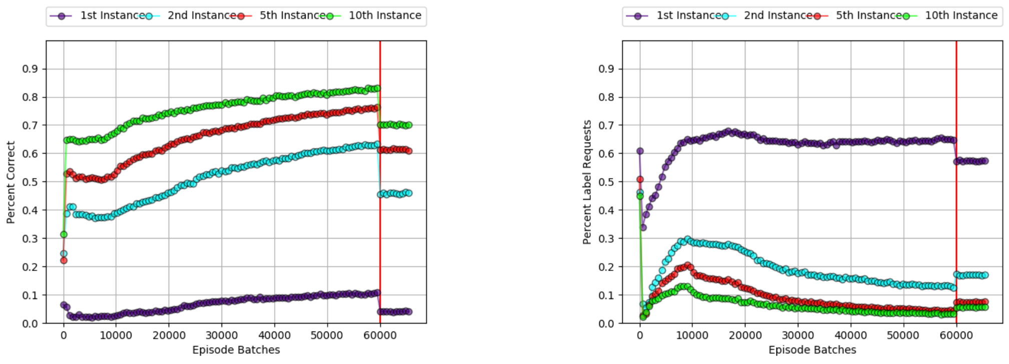
The memory networks, which employs the LSTM network as a memory-controller and thus doesn't send its output directly to the fully connected layer, but is used to *address* the memory instead, does slightly better when *all* classes are presented than the basic LSTM model. Both the zero-shot accuracy of the last class and the following one-shot accuracy is increased, as shown in figure 5.11. They additionally use a LSTM network of almost half the size, which potentially could be increased for a possible increase in performance. The memory networks could thus be better at distinguishing classes from each other, and could be a preferred choice when more classes are involved, despite their extended training time.

5.3.2 Active Learning for Text Classification

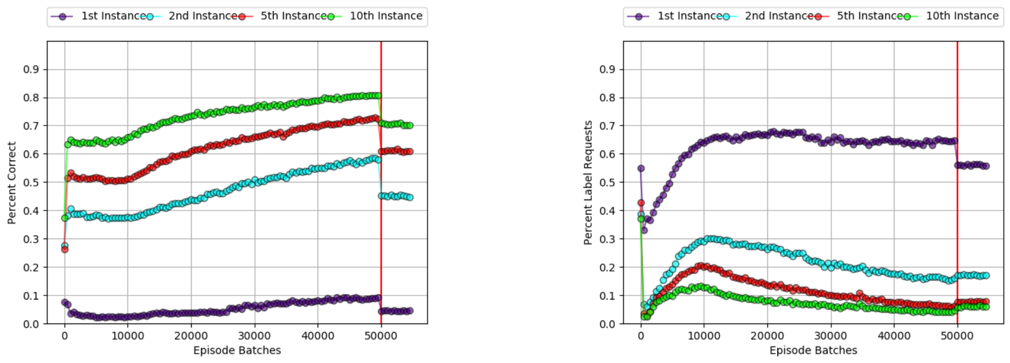
For text classification, the models are evaluated on the same criteria as with IC, but apply a few different scenarios to further visualize how their behaviour are different from the IC models. Since these models converged faster than usual, they are only trained for 60 000 episode batches, and tested for 6000 episode batches.



Figur 5.14: INH: Average accuracy and request percentage for k-shot predictions for LSTM. The red line indicates when I stop training and switch to the test set for validation



Figur 5.15: INH: Average accuracy and request percentage for k-shot predictions for NTM model. The red line indicates when I stop training and switch to the test set for validation



Figur 5.16: INH: Average accuracy and request percentage for k-shot predictions for LRUA model. The red line indicates when I stop training and switch to the test set for validation

As expected, all models struggle more with text classification when I switch to the test set for validation, as shown in tables 5.14 and 5.14. When employing a non pre-trained text embedding layer, the word vectors are co-trained with the model, which in some way should specialize the word embeddings for this particular problem, at least the specific dataset and dictionary. Two *disjoint* dataset partitions are also used, with completely different classes (i.e. headline categories), which also means that there could be completely different words in these two partitions. For example some headlines consist only of names of actors and places, which most likely will be rarely seen elsewhere in the dataset.

Deciding the size of the dictionary in use is a complicated choice, where a too large size potentially would include words rarely seen, which in turn might only exist in one of the two partitions, and where a OOV-token would be more preferable. If the size is too small, word vectors only consisting of the OOV-tokens could be created, which offers little to none information about the category of the headline. Thus using pre-trained word embeddings could be a way of achieving more generalized word vectors for the dataset, but co-trained word embeddings are used in this thesis to challenge the different models.

The spikes in accuracy shown in figures 5.14 and 5.16 seem to happen sooner for the NTM models than the LSTM. After the spike, all models seems to reach an equilibrium, where the accuracy is kept relatively constant, whilst the percentage label requests are *increasing*. Given that the accuracy also counts label requests as incorrect predictions, the *prediction accuracy* is increasing during this equilibrium, and continues to increase when the model converges towards a value for percentage label requests. It seems that the models grow over-confident on the task to begin with, and thus needs to "correct"its own strategy when under-performing, which then is the reason for the increasing percentage label request.

Training set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	82.22	10.63
NTM	**83.56**	**9.68**
LRUA	81.65	10.39

Tabell 5.14: INH: Training set accuracy and request percentage per episode. Accuracies are only calculated from predictions

Test set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	69.24	**10.63**
NTM	70.92	11.47
LRUA	**71.25**	11.66

Tabell 5.15: INH: Test set accuracy and request percentage per episode. Accuracies are only calculated from predictions

Tables 5.16 and 5.17 show that all class instance accuracies decrease when switching to the test set, and *especially* the zero-shot accuracies. Interestingly enough, while the zero-shot accuracy is greatly decreased, the corresponding label request percentages is decreased as well. I will try to explain why this happens later. For the remaining class instance accuracies, the request percentages are increased as expected, which shows that the TC models struggle more with adapting to the test dataset than the IC models. This could be because of the word embeddings being co-trained with the RL task, and words unique to - or more occurring in - the test set will bring uncertainty to the predictions, i.e. that words in the test set are generally under-represented in the training set.

	Instance (% Correct)				**Instance (% Requested)**			
Model	1st	2nd	5th	10th	1st	2nd	5th	10th
LSTM	24.1	66.4	74.5	82.4	**61.3**	14.7	5.9	4.2
NTM	**29.2**	**72.7**	**79.5**	**85.4**	66.2	**13.3**	**4.7**	**3.7**
LRUA	26.6	68.4	77.0	83.8	63.5	15.9	6.3	4.2

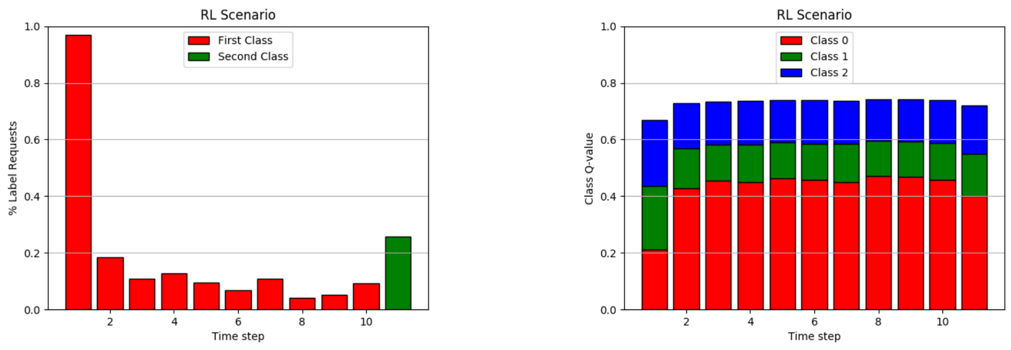
Tabell 5.16: INH: Class instance accuracies on the training set, and percentage requests made. Accuracies are calculated based on predictions made, excluding label-requests.

	Instance (% Correct)				**Instance (% Requested)**			
Model	1st	2nd	5th	10th	1st	2nd	5th	10th
LSTM	7.43	52.0	63.7	73.1	**54.1**	**15.2**	**7.0**	5.7
NTM	9.5	**55.2**	**66.2**	74.3	57.3	16.9	7.5	**5.6**
LRUA	**10.2**	54.4	66.1	**75.0**	56.0	17.1	7.7	6.1

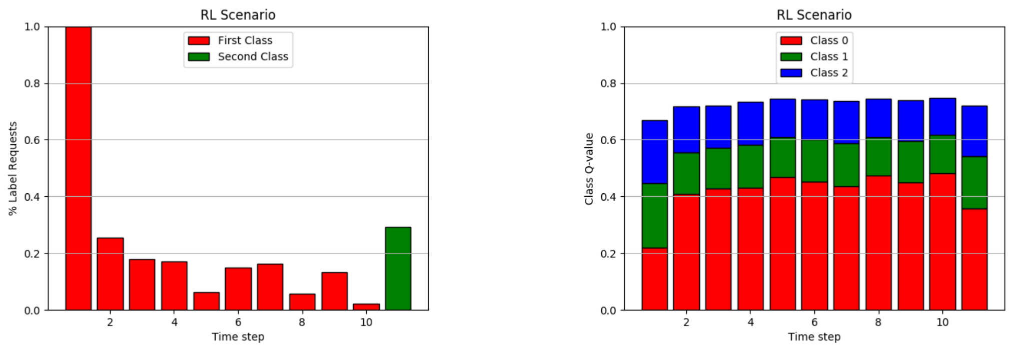
Tabell 5.17: INH: Class instance accuracies on the test set, and percentage requests made. Accuracies are calculated based on predictions made, excluding label-requests.

Analyzing Behaviours

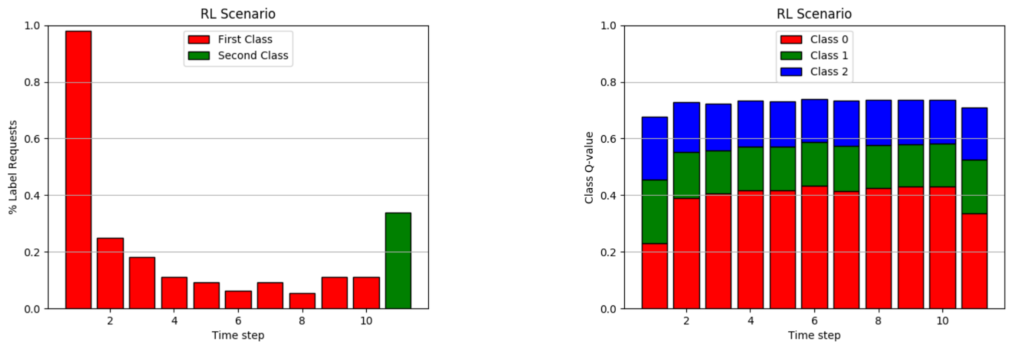
The same scenarios applied to the models during image classification is here used to evaluate the AL strategy learnt for the models on text classification.



Figur 5.17: Average requests made, and class-specific Q-values for LSTM, where first 10 samples presented is of the same class, and than the next sample is of a different class



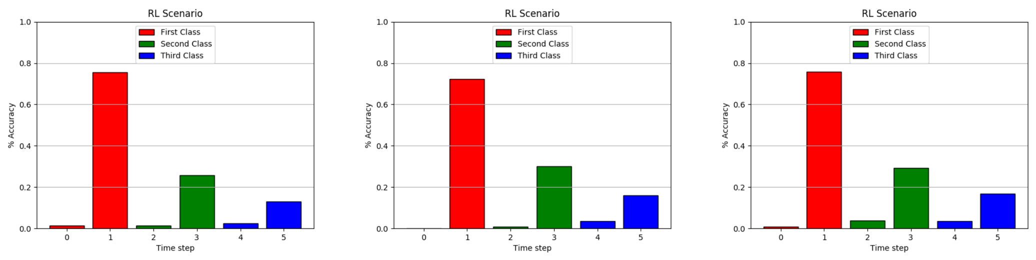
Figur 5.18: Average requests made, and class-specific Q-values for NTM, where first 10 samples presented is of the same class, and than the next sample is of a different class



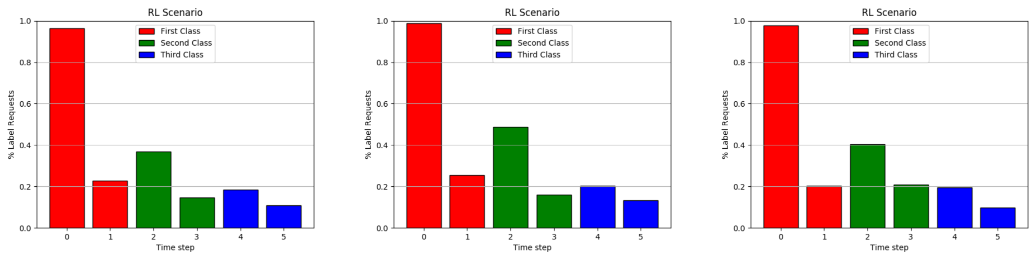
Figur 5.19: Average requests made, and class-specific Q-values for LRUA, where first 10 samples presented is of the same class, and than the next sample is of a different class

Comparing the figures above with figures 5.6, 5.7 and 5.8, its obvious that the models are struggling more to learn a capable AL strategy to classify these sparse headlines. After the first sample of the first class, the label requests decrease as expected. This is similar to the behaviour in IC, but where the label requests in IC gradually starts increasing with the incoming same-class samples, there's almost a continuously decreasing percentage label requests in text classification. If we evaluate the rightmost graphs, we can see that immediately after receiving the first class sample, the class Q-value of that class almost doubles for all models. The following samples doesn't improve these Q-values notably, as the model seems to have reached a maximum Q-value for the given class already. When provided the first sample of the *second* class, the Q-value is reduced, but the value for the *first* class is still heavily dominating the output, resulting in only a small increase in percentage label requests. Since these results are very different from the IC results, it's most likely that the change in behaviour is defined by the textual representation of data, since this is the only change to the models.

The LRUA model seems to be able to distinguish better between the two classes, as it request a slightly higher percentage of labels. It also doesn't increase the Q-values for the first class as much as the LSTM, which makes it less likely to overfit on the first class provided in an episode. Once again comparing with the scenarios done for IC, it's apparent that the models are capable of distinguishing images from each other *easier* than these sparse headlines, which is expected. Behaviour that both models for IC *and* text classification exert, is that the first sample of an episode produce the most change in class Q-values. This makes sense, as the hidden layer of the LSTM (either the model or the controller) is *zero* at this point, and any input to this layer will have no interference from other samples. It's easier to remember one class of images or texts, than two. Since the memories in the NTM and LRUA are wiped, the same goes here. The result of the interference introduced by different classes could reduce the one-shot accuracy, or at least the *confidence* in these predictions, in turn resulting in more label requests for these instances. This effect can be seen below in a scenario where pairs of samples from the same class are presented concurrently.

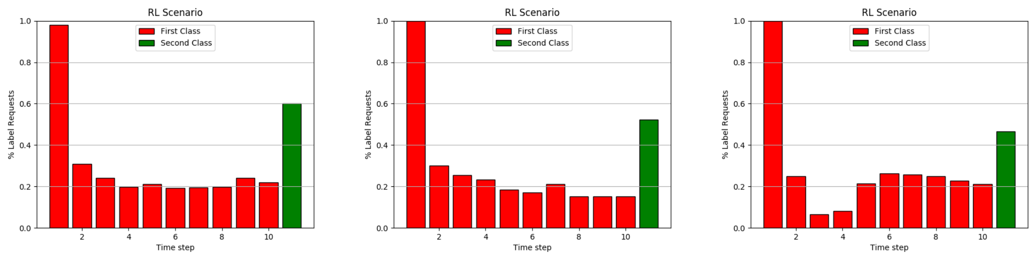


Figur 5.20: Average accuracy (with label requests determined as an incorrect prediction) for LSTM, NTM and LRUA respectively on the **test**-set, where the models receives two samples of each class in pairs.



Figur 5.21: Average percentage label requests for LSTM, NTM and LRUA respectively on the **test**-set, where the models receives two samples of each class in pairs.

From these scenarios, we can see that the models struggle when new classes are introduced, which could be a result of the interference in the hidden layer of the LSTM, and also in the explicit memory in the NTM and LRUA model. Figure 5.20 shows that Q-values for the first class seen in the episode continue to dominate the output even after all classes have been seen. As more classes are introduced in an episode, the models struggles to classify the last-arriving classes. If the first scenario's redone using the training set, the results immediately looks more similar to those of IC, as seen below.



Figur 5.22: Average requests for LSTM, NTM and LRUA respectively, made where the first 10 samples presented is of the same class, and than the next sample is of a different class.

Impact of Input Representation

The most likely reason for the discrepancies between performance on the training- and test-set is the use of word-embeddings, and especially for short texts - not unlike the INH dataset. While treating images as input, "unknown"images - images of classes we haven't seen before - produce *one* matrix of size 20×20 to give to the model. By simple reasoning, images of the same class will produce similar looking matrices, and very different images will produce very different matrices, but still only one matrix. With text, two similar headlines could be similar in two different ways:

- The *words* in the headlines are the same, or
- The *word-vectors* in the headlines are similar, meaning similar words are used

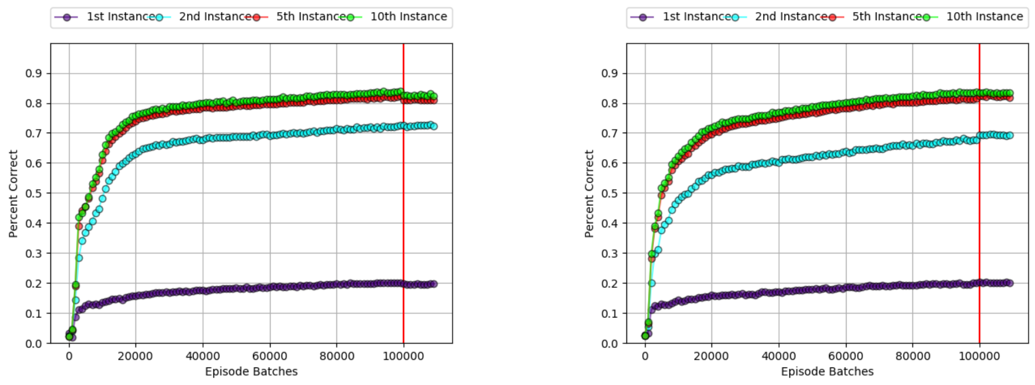
This also means that two headlines could be different in the same ways. In fact, two headlines could consist of completely different words, all with completely different word-vectors, potentially resulting in 12 completely different 128-bit long vectors! Thus, insufficiently trained word-vectors for words in the test set would almost exclusively produce noise to a potential prediction, and with too much noise, predictions will be difficult.

As figure 5.22 shows, it seems that this simple version of training on textual datasets isn't sufficient enough. Thus experimenting with CMS for textual datasets were not prioritized during development of this thesis.

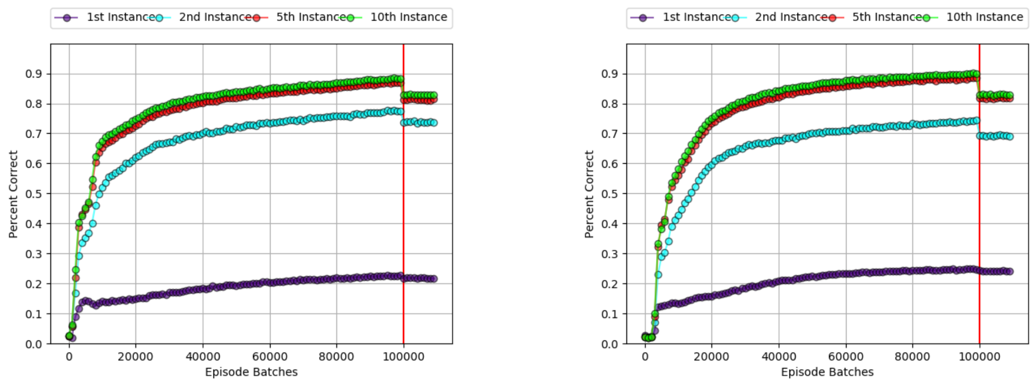
5.4 Class Margin Sampling

I further experiment with augmenting each model with CMS, either with $C_{cms} = C * 2$ or $C_{cms} = C * 3$, hereby written $C_{cms} = 2$ and $C_{cms} = 3$, and a margin time $T = 4$. Unless otherwise mentioned, $C_{cms} = 2$ is used to train the model. The different applications of CMS are evaluated on the OMNIGLOT dataset for IC.

5.4.1 CMS for Image Classification

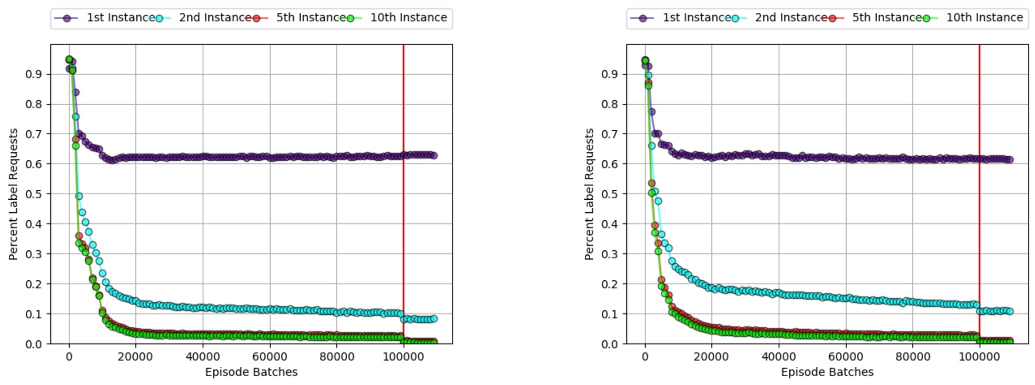


Figur 5.23: K-shot accuracies for LSTM, without margin sampling, and with margin sampling $C_{cms} = 3 * C$, respectively. The red line indicates when training is stopped, and I switch to the test set for validation.

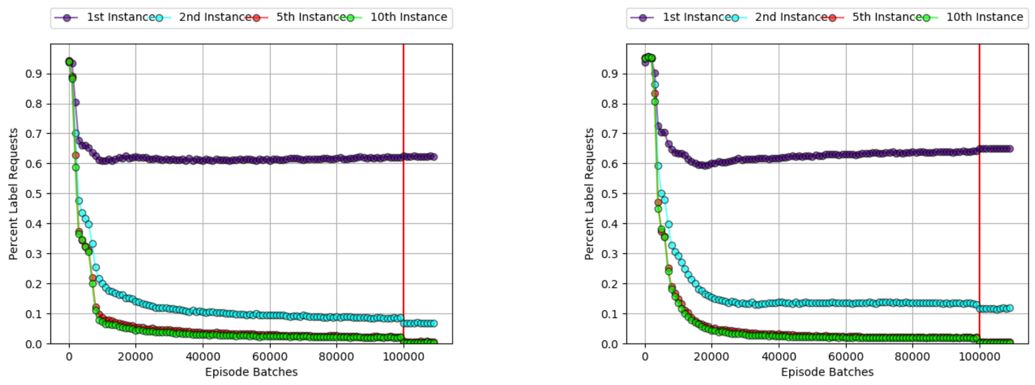


Figur 5.24: K-shot accuracies for the LRUA model, without margin sampling, and with margin sampling $C_{cms} = 3 * C$, respectively. The red line indicates when training is stopped, and I switch to the test set for validation.

The rightmost graphs in figure 5.23 show that the models using CMS have a similar sudden increase in accuracy as the one without CMS, but this spike smooths out *before* the standard model. This behaviour is usual for AL-methods, as they usually perform better in the early stages of training. In a standard classification setup, a difficult"classification would result in a greater parameter update, as the gradient would be larger as a result. Since this is a RL-problem, this process will be different. Instead of resulting in a larger gradient, the model will receive a more difficultepisode, and thus learn to balance the label requests and classification more carefully, possibly resulting in a slower convergence. By examining figure 5.25 displaying average request percentage during training, we can see the model exerting this behaviour.



Figur 5.25: K-shot request percentages for LSTM, without margin sampling, and with margin sampling $C_{cms} = 3 * C$, respectively. The red line indicates when I stop training and switch to the test set for validation.

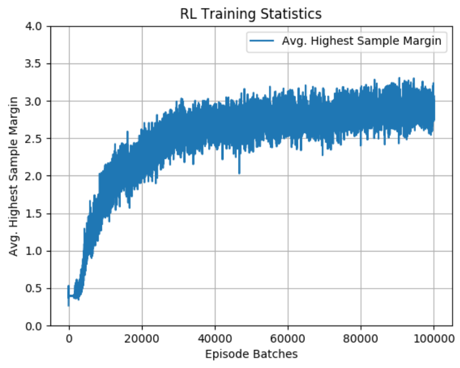


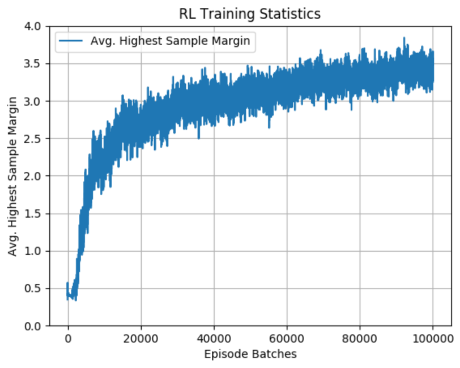
Figur 5.26: K-shot request percentages for LRUA, without margin sampling, and with margin sampling $C_{cms} = 3 * C$, respectively. The red line indicates when I stop training and switch to the test set for validation.

Instead of producing large updates to the model as in standard margin sampling, CMS will most likely produce a smoother learning curve that eventually should result in better performance. From figure 5.27 we can see that the average highest sample margin per episode batch is increasing similarly to the prediction accuracy in figure 5.23. In the first epochs before any rewards has propagated through the model, the average sample margin is - as expected - close to zero. Thus, CMS should have almost no effect before the model starts to improve, but as the figure shows, the models employing CMS actually start improving *sooner* than the ones without.

The reason for this early improvement might be that CMS identifies and inhibits any class-bias created by the initial state of the parameters. By default, each output Q-value should be activated with a $1/(C+1)$ chance in the first epochs of training, but with a random uniform seeding of model parameters, different input will most likely generate some initial bias towards an output value. The core functionality of CMS will allow the model to choose the input classes with the *least* amount of bias, and thus could speed up the initial learning in the model. It's important to note that CMS use the maximum value of the output nodes, which could be the request"output node, or the class prediction nodes, in the fully connected layer. The effect of this is more clear in figure 5.31, where the penalty for incorrect predictions are increased and the model thus uses more time in the explorationphase.

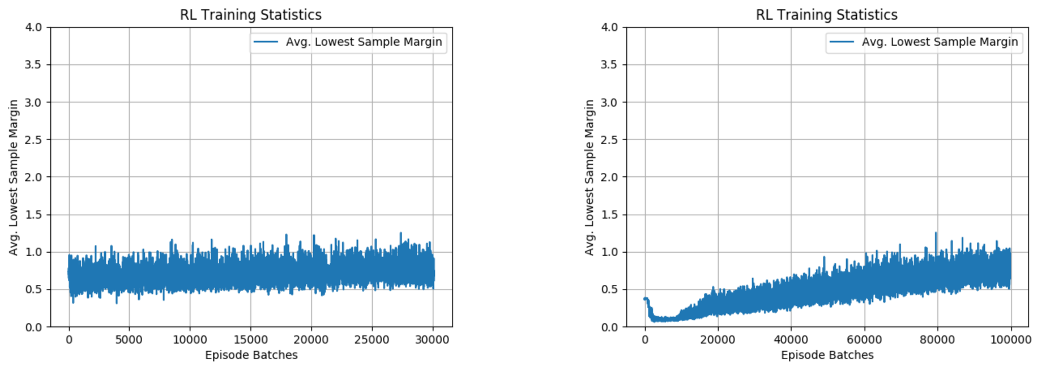
During training, the highest and lowest average margin per episode batch is logged, and the results can be seen in figures 5.27 and 5.28 respectively. By increasing C_{cms}, the corresponding highest average margin increase accordingly. The highest average margins are the margins from the highest scoring classes in a batch (i.e. some of the classes I *don't* train on). By increasing C_{cms} the probability of adding an easy"class to an episode decreases, but as a consequence of applying the CMS algorithm to sample selection, the training time increases. Table 5.19 shows that the difference in prediction accuracy is small, and actually that $C_{cms} = 2$ does better than $C_{cms} = 3$.[1]





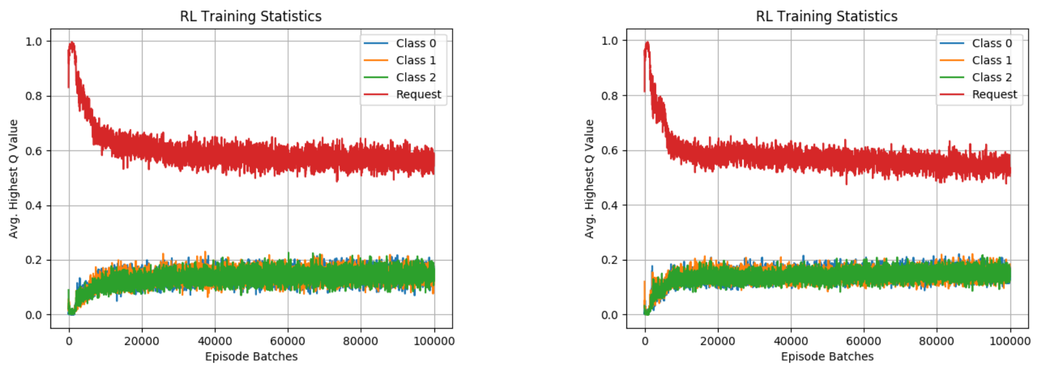
Figur 5.27: Batch average *highest* sample margin per episode batch for LSTM, with $C_{cms} = 2$ to the left, $C_{cms} = 3$ to the right.

[1]This will be discussed further in chapter 6.



Figur 5.28: Batch average *lowest* sample margin per episode batch for LSTM, with $C_{cms} = 2$ to the left, $C_{cms} = 3$ to the right.

The graphs in figure 5.27 show that increasing the size of the classes in the pool of classes in CMS also increases the highest average sample margin as expected. In other words, by evaluating more classes we will usually sample a class with higher margin. More interesting is the results in 5.28, where $C_{cms} = 3$ seems to do better in the beginning. After ~ 80000 episode batches, both graphs oscillate around the same values. This indicates that using a larger pool size in CMS in the beginning and decreasing this size over time could result in better performance, and reduce the training time of CMS with many classes.



Figur 5.29: These graphs show the output nodes that had the highest Q-value *and* were selected to train on during CMS. E.g with $T = 4$, the 4 different highest Q-values used in calculating the margin will be plotted for that episode. This is averaged over every episode batch for the LSTM with $C_{cms} = 2$ to the left, $C_{cms} = 3$ to the right.

The graphs displaying the different highest outputs chosen during CMS show that the difference between $C_{cms} = 2$ and $C_{cms} = 3$ is small in terms of how the highest output nodes for chosen image classes develop over time. There's a slight decrease in the highest and lowest point of the oscillations with $C_{cms} = 3$, which makes sense as more classes are evaluated, and a more consistent selection scheme is thus expected.

Model	**Instance (% Correct)** 1st	2nd	5th	10th	**Instance (% Requested)** 1st	2nd	5th	10th
LSTM	51.6	78.6	81.4	82.4	62.8	8.30	1.2	0.7
NTM	52.5	77.9	81.8	83.0	63.3	8.4	1.6	1.1
LRUA	58.0	**79.2**	81.8	83.2	62.3	**6.9**	0.7	0.4
LSTM CMS	53.3	78.8	82.6	83.2	63.3	9.9	1.2	0.5
NTM CMS	50.8	77.7	83.0	84.2	62.2	9.3	1.7	1.1
LRUA CMS	63.7	79.4	82.6	83.7	63.9	8.9	0.8	0.5
LSTM $C_{cms} = 3$	52.7	77.9	**83.1**	**83.7**	**61.6**	11.0	1.1	0.5
LRUA $C_{cms} = 3$	**69.1**	78.4	82.2	83.1	64.9	11.7	**0.6**	**0.3**

Tabell 5.18: OMNIGLOT: Class instance accuracies on the test set. Accuracies are only calculated from predictions made.

As we would expect, augmenting the models with CMS increase the percentages of label requests done by the model. Additionally, by increasing C_{cms}, the label requests generally increase even more. From the table below, we can also see that CMS increase prediction performance of all models. This strengthens the belief that CMS forces the model to make more difficult predictions, which in turn forces the model to *learn* how to better reason about its own uncertainty, and thus learn a better AL-strategy.

As mentioned before, the LRUA model seems to learn more meta-information than the other models, as they perform significantly better on zero-shot predictions. With CMS, this performance is even better, with a prediction accuracy of 69.1% on first class-instances. This is an improvement of 11.1% from the standard LRUA model, and further substantiates our claim that using CMS could potentially decrease the amount of class-bias in the models (which is a product of repeated training on images receiving the same pseudo-label multiple times). Table 5.19 show the performance when increasing C_{cms} for all models[2].

	Test set	
Model	**Accuracy (%)**	**Requests (%)**
LSTM	80.18	8.09
NTM	80.44	8.45
LRUA	80.95	**7.51**
LSTM CMS	81.7	8.27
NTM CMS	81.35	8.47
LRUA CMS	81.81	7.95
LSTM $C_{cms} = 3$	81.35	8.19
LRUA $C_{cms} = 3$	**82.2**	8.25

Tabell 5.19: OMNIGLOT: Test set accuracy and request percentage per episode. Accuracies are only calculated from predictions.

By increasing C_{cms}, some of the models perform better, but not all. The LSTM *reduce* its prediction accuracy with $\sim 0.3\%$ whilst the LRUA model *increase* its performance by $\sim 0.4\%$. The training time is increased as well, and unless trained on a powerful CPU, CMS with $C_{cms} = 2$ performs

[2]the files for the NTM $C_{cms} = 3$ model got corrupted during training, and hence are not represented here.

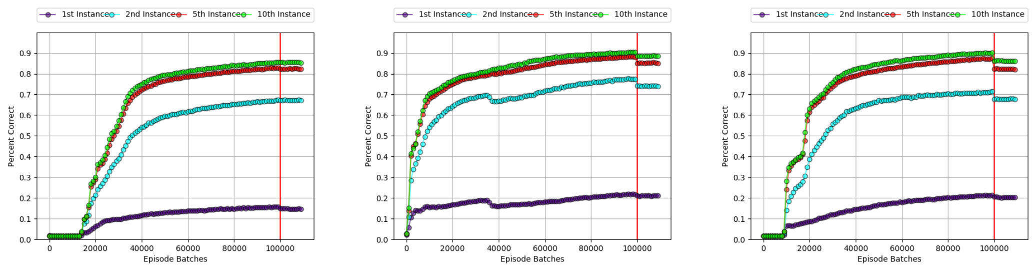
almost no different than with $C_{cms} = 3$ and could be preferable. Despite this, if the penalty for incorrect predictions is increased, it could be beneficial to employ the latter. This is because the stability of CMS is assumed to increase with the number of classes in the pool of classes, as it reduces the probability of generating an easier episode. If by chance all classes collected are regarded as easy"classes to classify, the model might momentarily try to predict more classes. Thus, the loss will receive a spike and hence will do a gradient descent in the opposite direction which might be an over-correction.

As the collection of data is different than for IC, the CMS algorithm needs some changes to be operational on the text datasets. As the experiments took longer time than expected, and this wasn't a priority, I simply didn't have time to implement a functioning version for TC.

5.5 Conditioning Behaviour by Varying Rewards

As the behaviour of a model is determined by the choice of rewards, changing these would in turn cause a change in behaviour. To identify how the different models react to different rewards, all models receive an additional training task. Here the penalty for doing an incorrect prediction (the negativereward) is increased to $r_{inc} = -2.0$. These experiments are done on both the TC and IC models. Finally, the combination of CMS and increased penalty for incorrect predictions were experimented with on the LSTM model for IC.

Image Classification



Figur 5.30: K-shot accuracies for LSTM, NTM and LRUA model respectively with $r_{inc} = -2.0$. The red line indicates when training is stopped and I switch to the test set for validation.

With increased penalty for incorrect prediction, both the LSTM and LRUA model request almost twice the amount of labels, but increases their overall accuracy with $\sim 6.0\%$. There's still a discrepancy between the LRUA model's performance on the training- and test-set, but the *increase* in prediction accuracy is actually greater on the test-set between the standard LRUA model and the one with $r_{inc} = -2.0$ for all class instances, and the total accuracy is increased with $\sim 4.0\%$ and $\sim 5.6\%$ for the training- and test-set respectively. This suggests that the LRUA model is able to generalize better when the penalty for an incorrect prediction is increased, meaning that when the LRUA model learns to be more careful with its predictions, it also learns a more generalized AL strategy. Another potential explanation is that increasing prediction accuracy could be more difficult when the prediction accuracy is already sufficiently high. The LSTM also increase the test-set prediction accuracy more than the training-set prediction accuracy with increased rewards, which suggests that training with higher penalty for incorrect predictions could help avoid overfitting for all models.

	Instance (% Correct)				**Instance (% Request)**			
Model	1st	2nd	5th	10th				
LSTM	51.6	78.6	81.4	82.4	62.8	8.3	1.2	0.7
NTM	52.5	77.9	81.8	83.0	63.3	8.4	1.6	1.1
LRUA	58.0	79.2	81.8	83.2	**62.3**	**6.9**	**0.7**	**0.4**
LSTM ($r_{inc} = -2.0$)	56.9	84.4	87.8	88.7	74.0	20.4	6.2	3.6
NTM ($r_{inc} = -2.0$)	**69.5**	85.6	89.9	**91.3**	69.6	13.4	5.3	3.1
LRUA ($r_{inc} = -2.0$)	64.8	82.2	87.5	89.0	68.4	17.5	6.0	3.3
LSTM $C_{cms} = 3$ ($r_{inc} = -2.0$)	61.0	**85.9**	**90.1**	90.6	73.7	20.5	4.6	2.5

Tabell 5.20: OMNIGLOT: Class instance accuracies on the test set with $r_{inc} = -2.0$. Accuracies are only calculated from predictions made.

Increasing the penalty for incorrect predictions seems to have a fortunate effect, as the models reach a similar prediction accuracy of that of [14] when they use a penalty of $r_{inc} = -10.0$, but with almost $\sim 72\%$ less label request than their model. Additionally, the LSTM model augmented with CMS request *less* labels for the test set than the training set, which is opposite of the results in 5.19, while still maintaining a high prediction accuracy.

Different from previous results, the NTM seems to handle zero-shot predictions substantially better with increased penalty for incorrect predictions. This class-instance accuracy is increased with $\sim 17\%$, with only an increase of $\sim 6\%$ label requests. The LRUA is also performing better on zero-shot predictions, and both NTM instances are superior to the LSTM on these predictions, especially with $r_{inc} = -2.0$. It seems that the explicit memory modules learns a more high-level strategy for classifying than the LSTM. By increasing the penalty for incorrect predictions, the expected Q-value of predicting the class of the first samples of the first $C - 1$ classes are lowered, which most likely results in an increased rate of label request for these particular instances. When the C^{th} class arrives, a sufficiently trained model should in both settings have a similar expected Q-value of a prediction, as all other classes have been seen. Thus the models with $r_{inc} = -2.0$ should learn to better use their high-level information about the episode structure, and adapt to a more careful approach to predicting (where now a 50/50 chance prediction is penalized more).

Training set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	82.66	7.99
NTM	83.45	8.28
LRUA	88.05	**7.32**
LSTM ($r_{inc} = -2.0$)	88.33	13.47
NTM ($r_{inc} = -2.0$)	**92.87**	10.68
LRUA ($r_{inc} = -2.0$)	91.98	11.35
LSTM $C_{cms} = 3$ ($r_{inc} = -2.0$)	91.02	12.47

Tabell 5.21: OMNIGLOT: Training set accuracy and request percentage per episode with $r_{inc} = -2.0$. Accuracies are only calculated from predictions.

Test set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	80.18	8.09
NTM	80.44	8.45
LRUA	80.95	**7.51**
LSTM ($r_{inc} = -2.0$)	86.76	14.01
NTM ($r_{inc} = -2.0$)	**89.09**	12.11
LRUA ($r_{inc} = -2.0$)	86.55	12.80
LSTM $C_{cms} = 3$ ($r_{inc} = -2.0$)	88.85	12.92

Tabell 5.22: OMNIGLOT: Test set accuracy and request percentage per episode with $r_{inc} = -2.0$. Accuracies are only calculated from predictions.

The tables above show that the model with the best prediction accuracy for IC is the NTM with $r_{inc} = -2.0$. Still the LRUA model perform much better on the training-set than the test-set, whereas the LSTM has the smallest discrepancy in performance between the two dataset partitions. Still, while the models increase their prediction performances by $\sim 7-11\%$, the corresponding increases in label request percentages are between $\sim 43-74\%$. Hence there's a greater increase in label increase, which means that these models are more applicable where the cost of doing an incorrect prediction is much higher than the cost of requesting a label.

Text Classification

Even with increased penalty for incorrect predictions, the models struggle with the INH dataset. The LSTM increases it's performance with $\sim 5\%$, while increasing the percentage label request with $\sim 15\%$. All models using increased penalty request substantially more labels on the test set than the training set, while not increasing prediction accuracy noteworthy. This suggests that other ways of improving the models for text classification should be applied in order to increase the prediction accuracy.

Training set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	82.22	10.63
NTM	83.56	**9.68**
LRUA	81.65	10.39
LSTM ($r_{inc} = -2.0$)	88.66	21.27
NTM ($r_{inc} = -2.0$)	88.54	14.35
LRUA ($r_{inc} = -2.0$)	**90.08**	12.51

Tabell 5.23: INH: Training set accuracy and request percentage per episode with $r_{inc} = -2.0$. Accuracies are only calculated from predictions.

Test set		
Model	**Accuracy (%)**	**Requests (%)**
LSTM	69.24	**10.63**
NTM	70.92	11.47
LRUA	71.25	11.66
LSTM ($r_{inc} = -2.0$)	**74.94**	25.59
NTM ($r_{inc} = -2.0$)	73.68	21.23
LRUA ($r_{inc} = -2.0$)	73.51	20.66

Tabell 5.24: INH: Test set accuracy and request percentage per episode with $r_{inc} = -2.0$. Accuracies are only calculated from predictions.

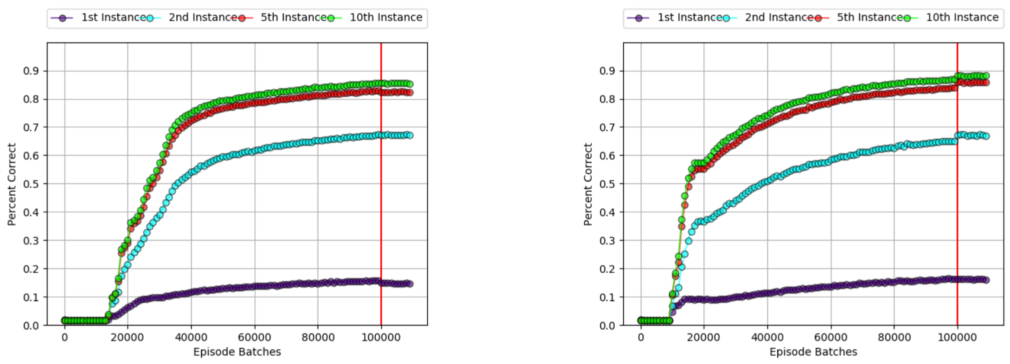
It is difficult to identify any of the same patterns from the results of the TC models as the IC models. All models perform similarly on both dataset partitions, both with and without increased penalty for incorrect predictions. The LRUA and NTM models does slightly better on zero-shot predictions as in IC, but these accuracies are much *worse* than random guessing of the label, and thus doesn't seem that reliable. More iterations of the same models should be done to suggest whether this happens by chance or not.

Model	**Instance (% Correct)** 1st	2nd	5th	10th	**Instance (% Requests)**			
LSTM	7.43	52.0	63.7	73.1	**54.1**	**15.2**	**7.0**	**5.7**
NTM	9.52	55.2	66.2	74.3	57.3	16.9	7.5	5.7
LRUA	10.2	54.4	66.1	75.0	56.0	17.1	7.7	6.1
LSTM ($r_{inc} = -2.0$)	5.7	56.0	69.2	**79.1**	66.2	32.0	23.9	21.6
NTM ($r_{inc} = -2.0$)	10.3	58.7	68.9	77.1	72.4	66.0	19.2	15.6
LRUA ($r_{inc} = -2.0$)	**12.1**	**59.4**	**69.3**	76.2	67.1	28.5	17.7	14.5

Tabell 5.25: INH: Class instance accuracies on the test set with $r_{inc} = -2.0$. Accuracies are only calculated from predictions made.

CMS and Rewards

By using CMS, we force the models to take greater risk during training, as we always choose the C classes we're *least* certain about. Additionally, the requests made are almost exclusively first instance label-requests, and as shown in [14], the accuracy of the model can be increased by increasing the penalty for incorrect predictions. These observation suggest that training with an increased penalty for incorrect predictions could potentially synergize well with the use of CMS. As these models require more training time than the standard models, this experiment was only done for the LSTM model.



Figur 5.31: K-shot accuracies for LSTM model with $r_{inc} = -2.0$, without margin sampling to the left, with margin sampling to the right. The red line indicates when we stop training and switch to the test set for validation.

The graphs above shows that the LSTM model with CMS are performing better on the test-set than the model without. As mentioned earlier, CMS seems to cause a sooner improvement during training, which is especially apparent with increased penalty for incorrect predictions, as this improvement starts nearly half the episode-batches. The spike in accuracy is also more steep using CMS, but as for the other models using CMS, seems to smooth out at a lower accuracy than the model without CMS. Referring to tables 5.21 and 5.22 we can see that the use of CMS increases the performance of the model by $\sim 2-3\%$ and a *decrease* in label request percentage of $\sim 1\%$. This shows that even though CMS is supposed to generate more difficult episodes during training, the model doesn't necessarily learn to request more labels, and actually requests *less* labels.

As discussed earlier, using CMS will most likely create more difficult episodes for the models to handle. The resulting change in parameters of the models will be determined on how they perform on these episodes, for example if they suddenly starts predicting a lot of incorrect labels, or even almost only correct labels, the reward will be large in *absolute* value. If this was not expected (by Q-value estimates), the corresponding loss would most likely be of a greater magnitude. From this observation its clear that radical change in behaviour would create correspondingly large changes in model parameters, subsequently creating further changes in behaviour. Thus, by increasing the penalty for incorrect predictions, the models will react *more* to an episode batch of a lot of incorrect predictions, than a similar episode batch of correct predictions. Since these two are mutually exclusive (one cannot be correct and wrong at the same time), the compromise would be to request more labels. When the models finally starts to achieve better prediction accuracies, the label requests will naturally decrease, and thus the probability of an incorrect predictions will increase.

Kapittel 6

Discussion and Future Work

6.1 Project Summary

I started this project by researching different strategies for AL, and experimenting with different basic implementations of RNNs, more precisely long short-term memory networks, and NTMs. I further reviewed what's considered the state-of-the-art in many topics, like AL, One-shot learning, Meta-Learning and similar topics in Deep Learning.

During the project, I've experimented with LSTM networks, and different variations of NTMs, for image and text classification, active learning and one-shot learning. The internal memory of a LSTM makes it suitable for tasks like sequence learning and one-shot learning, while the augmentation of an external memory in the NTM offers an end-to-end differentiable memory-network with greater memory capacity than the basic LSTM. The OMNIGLOT- and INH-dataset were used to evaluate the models' capabilities of learning a viable AL strategy for one-shot learning with a minimal percentage label requests. The meta-learning task setup presented is not particularly problem specific and may be adopted to similar tasks, which was one of the objectives of the thesis.

Three main models were created where the baseline model is a LSTM network as in [14], and the two other models were instances of NTMs - only different in the way they write to memory. All models were tested on both text and image datasets, and were also later in the project expanded on with an additional augment for creating more difficult episodes, called CMS. In an attempt at displaying the adaptive nature of RL training, some models were trained with different rewards, penalizing *more* for incorrect predictions. The results of these experiments showed how the task setup can be tweaked in order to satisfy different criteria - albeit better prediction accuracy, or lower label request percentages. The different models were consistently compared against each other in an attempt at identifying unique behaviour, shortcomings and benefits.

Issues with implementations of the NTM models and their respective long training times proved a challenge with a certain time limit. Small errors in the architecture - which usually in AI causes distinct results - didn't behave much different from the expected behaviour of the models, and we're thus hard to identify. With these complex memory structures, visualization of how the memory structures develop over time was helpful in finally implementing the correct models.

The LRUA model did usually perform best on the *training set*, whilst performing similarly to the other models on the test set. The LSTM and NTM performed similarly in many experiments, and are also the ones most similar in nature, but only slightly more than the LRUA. By increasing the penalty for incorrect predictions, the NTM model performed closely to the one in [14] on IC, but with $\sim 73\%$ less label requests. The models did generally struggle more with TC, and many of the same experiments as with IC didn't provide the same results. Using CMS, most models performed better, and the combination of CMS and increased penalty for incorrect predictions improved the performance further.

6.2 Discussion

As most models were trained only once, some of the results has to be further validated before any clear conclusion about which models that generally perform best can be drawn. Nonetheless, there's some behaviour that remain constant for models in different settings. For example, the LRUA is usually performing very well on the training set while under-performing on the test set. Since the training setup ideally should be indifferent to which dataset it uses to train the models, I would expect all models to perform more similarly on both dataset partitions, especially for IC. Despite this, the LRUA shows that it has the potential of performing really well, which is also shown in [2].

The NTM - which was the model that got the highest prediction accuracy on the test set - did almost perform as consistently as the LSTM, but with a slightly higher discrepancy between the two dataset partitions. In terms of zero-shot prediction accuracy, the LRUA augmented with CMS and with $C_{cms} = 3$ perform second best with $\sim 0.4\%$ less prediction accuracy, but with $\sim 5\%$ less label requests than the best model. The best model, which is the NTM with $r_{inc} = -2.0$, reaches a zero-shot prediction accuracy of 69.5%. It seems that the NTM and LRUA models are consistently doing better on zero-shot predictions than the LSTM models, suggesting that they either learn an AL-strategy with more meta-information than that of the LSTM, or that they are better at distinguishing the classes when few samples are seen due to their memory capabilities.

Despite not being the best performing model, the LSTM $C_{cms} = 3$ ($r_{inc} = -2.0$) has the highest one-shot prediction accuracy, which is $\sim 0.3\%$ higher than that of the NTM ($r_{inc} = -2.0$). Thus training the NTM models with CMS and ($r_{inc} = -2.0$) would most likely increase their performance further, especially for late-shot predictions, which is increased for the LSTM ($r_{inc} = -2.0$) augmented with CMS. As this model is performing better *and* requesting less labels, it shows that the potential of the model is not reached with only increasing the penalty for incorrect predictions.

During training of the TC models, it was early evident that the models struggled more learning a decent AL strategy for the INH datasets. As this dataset is a difficult one, and TC being generally more complicated than IC, I concluded that these models needed a better way of representing text, instead of augmenting the data sampling, or other approaches. As this wasn't the scope of the thesis, only the basic models with $r_{inc} = -1.0$ and $r_{inc} = -2.0$ were trained and evaluated.

When writing to memory, the LRUA will always zero out a given number of slots in the memory (i.e. the least-used locations), possibly resulting in less interference from memories since these in no way will affect the new memories written. This means that the LRUA emphasizes the value of keeping recent information, and adds the possibility of removing old information. This is also a possibility of the standard NTM, using the erase-gates during memory writes, but this method is not equal to the memory wiping in the LRUA models. This could potentially predispose the LRUA to better handle long episodes, as they are more capable of rapid change. While inspecting the memory matrices of the NTM, some seems to saturate at the end of episodes, which suggest that either the memory slot size should be increased with the length of the episode, or that the erase-functionality isn't correctly trained. As this is a complex memory structure, more experiments are necessary to determine this.

Even though the LSTM performs similarly to the NTM and LRUA with a much smaller training time, the results indicate that the performance potential is greater for the explicit memory structures, with mostly higher training set performances. By for example training on more classes in an episode, I would expect the NTM and LRUA to perform much better than the LSTM, as shown in

[2]. As discussed in [9], taking the strain of the model to e.g. only handle the feature extraction or the high-level information part of the meta-learning setup, the models will perform much better. For this particular task setup, allowing the models to store sample representation in an explicit memory could have a similar effect, allowing the LSTM to rely on the external memory to remember, instead of solely using its internal state to represent the environment. But as seen in [14], the LSTM is capable of doing this for at least 3 classes and over 30 timesteps. Thus increasing these parameters could potentially distinguish the models further. I can't see any particular reason for the NTM and LRUA models to perform better than the LSTM on this task, but more future work are needed to validate this postulate.

6.3 Future Work

As these models usually train for several days, optimization of the training procedure could be important to reduce the training time. Also, more systematically averaging results over several models will most likely produce more comparable results, and maybe determine which model that is most suited for this task.

The experiments in this thesis is deliberately performed in a generic manner, and thus any further work with specialized models expanding on my experiments will be a natural progression for the future work. As seen in [9], augmenting the model with better feature representations could potentially alleviate the combined pressure on the agent, by enhancing the meta-learning with more informative input. By for example using powerful existing implementations of CNNs to extract features from images, which is sent to the agent could result in better performance. This could also possibly allow the models to accommodate cross-dataset training and use, as the input-layer of the CNN is usually more forgiving than the LSTM (especially for TC). By using a CNN, images of greater size could be used without increasing the training time drastically, which also more easily predispose the models to cross-dataset evaluation and training.

The text embedding module implemented, is as previously stated not a state-of-the-art text embedding in any way. It's merely used as a method for the model to handle textual data representations. Thus augmenting the text embedding with more powerful methods for text representation, or using pre-trained word vectors will be an interesting area to expand on in order to increase the performance on text datasets. The inclusion of pre-trained word vectors would also predispose the model to different datasets, increasing its generality. Most of the results show that the models are overfitting on the words in the training set, possibly due to an uneven balance of words in the two dataset partitions. By using a globaldictionary - independent of the datasets - this issue could be greatly reduced. This will also allow the model to be evaluated using *different* datasets, e.g. the Reuters dataset. The IC models could equally be evaluated on the MNIST dataset, or other similar image datasets, to determine how well they perform on a similar dataset.

Some results - especially on the NTM models - are not averaged over several tries, due to lack of training time. In the future it would be interesting to experiment with adaptive reward functions, with an increasing penalty for either requesting labels or incorrect predictions scaling with the number of training epochs. It's usually expected *more* from models that perform well, than those that don't, and thus penalizing accordingly could be an interesting experiment. Another interesting practice that wasn't tried, is to reverse the CMS procedure to always choose the easiest episodes. As the training of models using RL is complex, it's difficult to predict exactly how the different models would react to this.

As mentioned in [2], learning weights of a classifier with large one-hot vectors increases in difficulty with scale, meaning that expanding the experiments with more classes could potentially become difficult and produce poor results. Thus increasing the number of classes of experiments could be done by a similar approach as in the article, where classes are created by a string of characters uniformly sampled from an array of characters, and represented as concatenated one-hot vectors to the model. Figuring out a meaningful way of representing the choice of requesting a label in combination with this approach could predispose the model to incorporate a large number of classes. I believe that this will let the NTM and LRUA models surpass the LSTM in performance.

Bibliografi

[1] Dongyu Zhang ... Keze Wang. “Cost-Effective Active Learning for Deep Image Classification”. Versjon 1. I: *arXiv.org* (2017). URL: `https://arxiv.org/pdf/1701.03551.pdf`.

[2] Sergey Bartunov ... Adam Santoro. “One-shot Learning with Memory-Augmented Neural Networks”. Versjon 1. I: *arXiv.org* (2016). URL: `https://arxiv.org/pdf/1605.06065.pdf`.

[3] *A Begunner’s Guide to Recurrent Networks and LSTMs.* [Accessed 05.12.2017]. URL: `https://deeplearning4j.org/lstm.html`.

[4] Ivo Danihelka Alex Graves Greg Wayne. “Neural Turing Machine”. Versjon 2. I: *arXiv.org* (2014). URL: `https://arxiv.org/pdf/1410.5401.pdf`.

[5] Jeremy Appleyard. *Optimizing Recurrent Neural Networks in cuDNN 5.* [Accessed 11.12.2017]. URL: `https://devblogs.nvidia.com/parallelforall/optimizing-recurrent-neural-networks-cudnn-5/`.

[6] Anonymous authors. “Meta-Learning Transferable Active Learning Policies by Deep Reinforcement Learning”. I: *ICLR (Under review)* (2017). URL: `https://openreview.net/pdf?id=HJ4IhxZAb`.

[7] ... Christian Buck Jannis Bulian. “Ask the Right Questions: Active Question Reformulation with Reinforcement Learning”. Versjon 2. I: *arXiv.org* (2017). URL: `https://arxiv.org/pdf/1705.07830.pdf`.

[8] Theano Development. *Theano.* [Accessed 13.12.2017]. URL: `http://deeplearning.net/software/theano/#`.

[9] Zhenguo Li Fengwei Zhou Bin Wu. “Deep Meta Learning: Learning to Learn in the Concept Space”. Versjon 1. I: *arXiv* (feb. 2018). [Accessed 01.05.2018]. URL: `https://arxiv.org/pdf/1802.03596.pdf`.

[10] Thierry Artieres Gabriella Contardo Ludovic Denoyer. “A Meta-Learning Approach to One-Step Active-Learning”. Versjon 2. I: *arXiv.org* (2017). URL: `https://arxiv.org/pdf/1706.08334.pdf`.

[11] Google. *TensorFlow.* [Accessed 13.12.2017]. URL: `https://www.tensorflow.org/`.

[12] Rohit Kulkarni. *News Headlines of India 2001-2017.* [Accessed 01.05.2018]. 2017. URL: `https://www.kaggle.com/therohk/india-headlines-news-dataset`.

[13] ... Mark Gorriz Axel Carlier. *Active Deep Learning for Medical Imaging.* [Accessed 13.12.2017]. URL: `https://www.slideshare.net/xavigiro/active-deep-learning-for-medical-imaging`.

[14] Chelsea Finn Mark Woodward. “Active One-shot Learning”. Versjon 1. I: *arXiv.org* (2017). URL: `https://arxiv.org/pdf/1702.06559.pdf`.

[15] *Omniglot dataloader.* [Accessed 09.12.2017]. URL: `https://github.com/ludc/vision/blob/1a82a107e30145605bd8109e4afe547e69ca9973/torchvision/datasets/omniglot.py`.

[16] *Omniglot Dataset for One-shot Learning.* [Accessed 11.12.2017]. URL: `https://github.com/brendenlake/omniglot`.

[17] *One Shot Learning using Memory-Augmented Neural Networks in Tensorflow.* [Accessed 29.05.2018]. URL: `https://github.com/hmishra2250/NTM-One-Shot-TF`.

[18] ... Oriol Vinyals Charles Blundell. "Matching Networks for One Shot Learning". Versjon 2. I: *arXiv* (2017). [Accessed 03.05.2018]. URL: `https://arxiv.org/pdf/1606.04080.pdf`.

[19] PyTorch. *PyTorch.* [Accessed 13.12.2017]. URL: `http://pytorch.org/`.

[20] *PyTorch Neural Turing Machine (NTM).* [Accessed 29.05.2018]. URL: `https://github.com/loudinthecloud/pytorch-ntm`.

[21] Hugo Larochelle Sachin Ravi. "Optimization as a Model for Few-Shot Learning". I: *ICLR 2017* (2017). [Accessed 11.05.2018]. URL: `https://openreview.net/pdf?id=rJY0-Kcll`.

[22] Zoubin Ghahramani Yarin Gal Riashat Islam. "Deep Bayesian Active Learning with Image Data". Versjon 1. I: *arXiv.org* (2017). URL: `https://arxiv.org/pdf/1703.02910.pdf`.

[23] Byron C. Wallace Ye Zhang Matthew Lease. "Active Discriminative Text Representation Learning". Versjon 4. I: *arXiv.org* (des. 2016). URL: `https://arxiv.org/pdf/1701.03551.pdf`.