Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for matematiske fag

Marius Mario Cervera Landsverk

Inductive Bias And The Information Bottleneck Method

Masteroppgave i Fysikk og Matematikk
Veileder: Signe Riemer-Sørensen
Medveileder: Knut-Andreas Lie

Juni 2021

NTNU
Norwegian University of
Science and Technology

Marius Mario Cervera Landsverk

Inductive Bias And The Information Bottleneck Method

Masteroppgave i Fysikk og Matematikk
Veileder: Signe Riemer-Sørensen
Medveileder: Knut-Andreas Lie
Juni 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for matematiske fag

Abstract

Inductive bias refers to architectural choices made when designing a deep learning model in order to facilitate the model learning on a particular kind of data. In particular, one makes assumptions about the structure of the data and designs a suitable model accordingly. Examples of architectures are convolutional neural networks for image data, graph-convolutional neural networks for graph data, and recurrent neural networks for sequential data.

The information bottleneck method arose to quantify the optimal trade-off between compression and accuracy when summarizing a random variable X. As applied to neural networks, one considers each successive representation $Z^i, Z^{i+1}, \ldots$ as functions of the input X, and one can thus compute the mutual information $I(X, Z^i)$ between the representation and the input, or $I(Y, Z^i)$ for the mutual information between the representation and the output. The main idea is that as the input is processed deeper in the network, the representations will lose information about the input X and gain information about the output Y. This means that the network is able to generalize away unnecessary variance in X, and only retain the parts relevant for predicting Y.

Using the information bottleneck, we seek to elucidate the training procedure and learning capabilities of different neural architectures. Previous works have mainly applied the method on synthetic datasets and architectures not commonly found in practical applications. In this work, we will be comparing the performance and information bottleneck for three different architectures of neural networks to their fully-connected counterparts. First, we will compare a graph neural network and a fully-connected network trained on the Cora citation dataset [1]. Then we compare a recurrent neural network and a fully-connected network on a dataset consisting of names from different languages [2], with the task of classifying the correct language for each name. Finally, we compare a convolutional neural network with a fully-connected network on the MNIST dataset [3].

Sammendrag

Induktiv bias referer til forskjellige arkitekturvalg som gjøres når man designer modeller for dyp læring. Spesielt så handler det om hvilke antakelser som gjøres om inngangsdataen, noe som i sin tur påvirker arkitekturvalget. Eksempler på forskjellige nevrale arkitekturer er konvolusjonale nevrale nett for bildedata, grafkonvolusjonale nevrale nett for grafdata og rekurrente nevrale nett for sekvensiell data.

Informasjonsflaskehalsmetoden søker å kvantifisere en optimal balanse mellom kompresjon og presisjon for å beskrive en tilfeldig variabel X. For nevrale nettverk betrakter man påfølgende representasjoner $Z^i, Z^{i+1}, \ldots$ som funksjoner av inngangsdataen X, og dermed kan man beregne den gjensidige informasjonen $I(X, Z^i)$, eller beregne $I(Y, Z^i)$ for den gjensidige informasjonen mellom representasjonen Z^i og målvariabelen Y. Hovedideen er at jo dypere i nettverket man kommer, så vil representasjonene Z^i få mindre informasjon om inngangsdataen X, og mer med målvariabelen Y. Dette kan tolkes som at nettverket er i stand til å fjerne unødvendig informasjon i inngangsvariabelen X, og er i stand til å generalisere ved å kun beholde informasjon som er relevant for å predikere Y.

Ved å bruke informasjonsflaskehalsmetoden ønsker vi å belyse treningsprosedyren og læringsevnen til forskjellige nevrale arkitekturer. Tidligere arbeid har i hovedsak betraktet syntetiske datasett og nevrale strukturer som ikke brukes i praktiske anvendelser. I dette arbeidet så kommer vi til å benytte informasjonsflaskehalsmetoden for å sammenligne tre forskjellige nevrale arkitekturer med deres fulltilkoblede alternativer, sammen med sammenligninger av deres ytelsesevner. Vi begynner med å sammenligne et grafkonvolusjonalt nevralt nett med et fulltilkoblet nettverk trent på Cora [1] datasettet. Deretter sammenligner vi et rekurrent nevralt nettverk med et fulltilkoblet nettverk på et datasett som inneholder navn fra forskjellige språk, der oppgaven er å klassifisere navn til riktige språk [2]. Til slutt sammenligner vi et konvolusjonsnettverk med et fulltikoblet nettverk på MNIST datasettet [3].

Introduction

Modern deep learning algorithms have brought enormous progress to a wide variety of fields [4]. Examples of this include completing sentences [5], generating images [6, 7], or more traditional classification and regression tasks [8]. Their success is based on being adaptable to a variety of problem formulations and data formats by adapting their architecture. An abundance of data and processing power has also been a key enabler for deep learning technologies [9]. A key feature of deep learning is that of representation learning. In order to reason across different types of data, it is required to bring different data into a common representation to compare and reason about them. An example of this is designing a neural network for text data and another for image data. To be able to reason about them, one can compress both data types into a common vector representation.

Another benefit of representation learning is that of dimensionality reduction. In the case of predicting the correct digit on the MNIST dataset [3] of handwritten numbers, the input image has $28 \cdot 28 = 784$ features (pixels). However, there are only ten classes. Deep neural networks are trained to produce a new representation of the images that are subsequently linearly separated into ten classes.

This thesis is concerned with how to analyze the representations produced by different deep learning networks. Different architectures yield different performance in terms of predictive accuracy, which can be attributed to them learning different representations of the same data. Identical representations would yield identical performance. One property that all neural networks share in common is that they sequentially produce a new representation Z^{i+1} from their previous representation Z^i, with the initial representation being the input data $X = Z^0$. The information bottleneck method [10, 11] proposes to elucidate neural networks' training progress and their learning and generalization capabilities by recording the shared information between sequential representations, Z^i.

In this thesis, we will only consider classification problems. The networks are trained on data (X, Y) of features X and are tasked with predicting the correct class contained in the label Y. Classification problems arise naturally in several real-life applications. In an industrial plant there are several sensors, and a machine learning system may be used to detect irregularities in the plant from the sensor measurements. Self-driving vehicles need to classify and interpret what kind of objects they see, which may range from a person to a traffic light. Even though the problems of detecting irregularities in an industrial plant and that of detecting pedestrians may seem very different, they are primarily so due to the data input, X, being different. One can use a very similar model in both cases and a very similar setup too. The difference is in what features X [12] and labels Y [12] to choose for the particular problem. Whether one *should* use a similar model and setup in both cases is an entirely different question, and will be the main emphasis of my thesis.

In computer vision problems, the input data X are images. The task Y which the model needs to solve can be numerous, for instance classifying a part of or the whole image, or segmenting away different parts of the image. In the industrial plant example, the sensor measurements in X are usually not as related to each other as the pixels in an image. An image also has the notion of locality, i.e., that pixels that are close to each other are related to each other. Both problems can be treated similarly if one flattens the image into a long vector of numbers. This would obviously destroy a lot of information inherent in the image, most notably locality. Intuitively and practically, methods that preserve the locality information, like convolutions, are more efficient on image data than the flattening. However, there are no standard way of quantifying this increased efficiency.

Neural networks can be used to solve various tasks on many different datasets. Nevertheless, interpreting neural networks proves challenging, since each task preferably requires its own specific neural architecture and training progress. In an attempt to analyze different network architectures in terms of representation learning and performance, we will be using the information bottleneck method [10].

The information bottleneck method from Tishby et.al [10] relies upon tracking the representations generated by neural networks during training and subsequently plotting them in an information plane. The axes of the information plane are the information the representation has about the input and output, plotted on the horizontal and vertical scales, respectively. By analyzing these plots of different models we seek to elucidate why and how different models' representations differ. In doing this, we differ from the more common approach of gauging model fit from the validation or test accuracy [12]. While these are good metrics, it may be possible to also gain insight into the model itself by analyzing the information plane. In addition, accuracy does not measure to which degree the model is exploiting the available information in the data. Maybe it would be possible to train a model with a different architecture to the same precision using less data?

The information measure is the mutual information $I(A,B)$ between two random variables A and B. The mutual information $I(A,B)$ is a symmetric quantity that quantifies how much of the variation in A can be explained by B and vice-versa. Mutual information is defined in terms of probability distributions of the variables A and B, however, these distributions are unknown in practical applications, and hence approximation methods are necessary. The method we will be using is from Kolchinsky et al. [13]. Other mutual information estimation methods were attempted [14, 15], but they yielded significantly subpar performance compared to the method of Kolchinsky [13].

There were three main research questions when starting this thesis:

1. The first is if one can use the information bottleneck method in order to measure the *inductive bias* of different neural network architectures. Inductive bias is a term used to signify that a model is particularly well-suited to work on a specific kind of data.

2. The second question is if the information bottleneck method can be used in order to gauge model fit, for example, to visualize and compare the information planes of several models trained on the same data with different sampling rates.

3. The third research question was if the information bottleneck method could be used in conjunction with hyperparameter optimization in order to find an optimal model.

In order to address the research questions, background information on neural networks and different neural architectures are presented in Chapter 1. Their training procedure is explained in Chapter 2. Chapter 3 introduces the information bottleneck method together with methods for estimating mutual estimation. This was all put together in terms of a framework, which will be explained in Chapter 4.

It was found that the mutual information estimation method, while it performed well on the test examples in the literature, struggled when presented with new datasets and neural architectures. As such, answering the second and third research questions proved difficult. In this thesis, we will, as far as it is possible, answer the first research question in Section 5.1, together with analyzing the failure modes of the estimation method from [13] in Section 5.2. The mutual information method from [15] (MINE) was numerically unstable, and the Rényi estimation method [14] had prohibitive computational costs associated, even for smaller problems. The MINE method worked on the example dataset in the paper by Tishby et.al [10], but diverged on more complicated data and architectural patterns. The MINE method will be described in Section 3.2, and for information on the Rényi method we refer to [14]. Possible future avenues of approach will be discussed in Chapter 6.

Contents

Chapter 1

Neural Networks

While the term “neural” in neural networks comes from comparisons between the structure of a neural network and the human brain, the comparison is not quite accurate. As stated in [8], “While the kinds of neural networks used for machine learning have sometimes been used to understand brain function, they are generally not designed to be realistic models of biological function.” As such, we prefer to explain different kinds of neural networks in terms of mathematical operations and structures rather than being explained through analogies to the human brain.

In this thesis, we will consider four different types of neural network architectures. They are fully-connected neural networks, convolutional neural networks, recurrent neural networks, and graph-convolutional neural networks. Except for the fully-connected networks, the other three networks are specialized at handling different kinds of data X. Recurrent neural networks are suited for sequential data, like text or time series. Convolutional networks are designed for image-like data, and graph-convolutional neural networks are suited for graph structured data. The same kind of data can be interpreted in several ways; for instance, one can interpret an image as a sequence of pixels or as a graph where neighboring pixels are connected (this is usually not done). The three models are specialized due to there being additional processing steps in the model, in addition to learning the parameters of the model. When a neural architecture is well-suited for a particular kind of data, we say that the neural architecture has an *inductive bias* towards that kind of data. In this chapter we will present all four of these neural network models, together with how they work and their strengths and weaknesses.

1.1 Fully-Connected Neural Networks

Neural networks were first proposed as having a single layer, or matrix, connecting the input X to the predicted output $\hat{Y}$. Some theoretical measures on their performance were also established early. The universal approximation theorem was first derived by Kolmogorov [16] in 1957. The theorem states that any multivariate continuous function can be approximated by a single-layer neural network with arbitrary accuracy. However, it was also established that a single-layer neural network is unable to learn the XOR function (exclusive or function) [17]. Single-layer neural networks are rather uninteresting for modern applications due to neural networks relying on scale and having available training data. The scale comes from stacking multiple layers together in order to form a ”deep” neural network. A single-layer neural network is not much different from linear regression [18].

In order to train deep neural networks, it is common practice to use the backpropagation algorithm to find the gradients by which to update the network. The backpropagation algorithm is explained in Chapter 2, and according to [18] the first application of the backpropagation algorithm to neural networks was in 1981 by Werbos [19]. Recent research has also tried to train neural networks without using backpropagation [20], with many prominent researchers seeking methods that resemble the human brain more than backpropagation does [21].

Modern advancements within computing technology have also allowed neural networks to be larger in size, which has enabled them to tackle greater problems with more accuracy [18]. Several improvements over the vanilla "backprop" algorithm have also been developed. For more information, we refer to [18].

A fully-connected neural network (FCN) operates on data X with dimensions $N \times D$, where N is the number of examples (samples) in the dataset, and D is the number of features used to describe each example. The fully-connected network has L weight matrices $\{W^l\}_{l=1}^{L}$ with weight matrix W^l having dimension $H^l \times H^{l-1}$. H^{l-1} is the number of features in the previous layer and H^l the number of features in the current layer.

A FCN revolves around applying linear transformations to the data followed by nonlinear functions, also called activation functions. The most common activation function used is the rectified linear unit (ReLU), defined as

$$\mathrm{ReLU}(x) = \sigma(x) = \max(0, x). \tag{1.1}$$

A plot of the ReLU function can be found in Figure 1.2. If x is a matrix, $\sigma(\cdot)$ operates elementwise, and element i, j in the output is $z_{i,j} = \sigma(0, x_{i,j})$.

An example illustration of a neural network is shown in Figure 1.1. There are three input features in green, followed by four sets of hidden layers in blue, each with a hidden size of four. There are two output classes in red. The arrows between different layers are the trainable connections in the network. The arrows are in practice elements of a matrix. The first matrix W^1 will have dimension 4×3 since the input has three features and the first hidden layer has four features. Subsequent weight matrices will have dimension 4×4 until the last prediction layer W^5 has a weight matrix with dimensions 2×4. It is these connections, or matrices, that are learned in deep learning. Different neural architectures will have different structures for the weight matrices W^l. For a FCN, the matrices W^l are dense, where each element $W^l_{i,j}$ is adjusted to relate the output feature at index i and the input feature at index j.

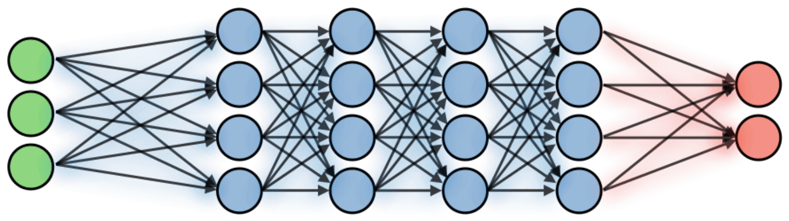

Figure 1.1: A neural network with four hidden states, colored in blue. There are three input features (green) and two classes in red. The arrows indicates connections between diferent layers. In practice, the arrows are the elements of a matrix. Image credit: [22]

The core part of what makes neural networks and deep learning special is that one applies several of the linear transformations $\{W^l\}_{l=1}^{L}$ after one another, with a nonlinear function in between. The layer-wise propagation rule for a FCN is

$$Z^{l+1} = \sigma(Z^l W^l), \tag{1.2}$$

where $Z^0 = X$ and $Z^L = \hat{Y}$, with $\hat{Y}$ being the prediction of Y, and $1 \le l \le L$. The output $\hat{Y}$ has dimension $N \times C$ for a classification task with C classes or $N \times 1$ for a one-dimensional regression task. When training the networks, the elements in the weight matrix W^l are optimized in order to reduce the training loss. The ”learning” part in deep learning comes from finding the gradients ∇W^l for all layers l, which is done through the backpropagation algorithm. More on neural network training in Chapter 2.

The advantage of a FCN is that they are very general since they work on data represented by a feature vector of length D. The downside is that if the number of features D becomes very large, it forces there to be many parameters in the model. For example, image data usually have a very large number of features (pixels) D.

1.1.1 Activation Functions in Deep Learning

The ReLU function defined in (1.1) is the most commonly used activation function for modern deep learning architectures [8]. The ReLU function is very close to a linear function, and as such, it preserves several properties of linear models which make them easy to optimize through gradient descent [8]. Other notable activation functions are the sigmoid and tanh activation functions, defined as follows

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{1.3}$$

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}. \tag{1.4}$$

A plot of the tanh, ReLU, and sigmoid activation functions are shown in Figure 1.2. The output of tanh is always in the interval $(-1, 1)$ whilst the output of the ReLU function is in the interval $[0, \infty)$. The output of the sigmoid is always in the interval $(0, 1)$. Note that the tanh and sigmoid function saturates when its input is far away from zero, leading to a derivative that is close to zero. When the derivative is close to zero, it is harder for the network to learn, which is part of the motivation for using the ReLU nonlinearity instead.

The prediction layer $\hat{Y}$ usually also has an activation function. For a classification problem one usually uses the softmax function [18]. Let p be a vector of length C, where the i'th element of p is p_i. The softmax function is defined as follows

$$\text{softmax}(p)_i = \frac{e^{p_i}}{\sum_{j=1}^{C} e^{p_j}}. \tag{1.5}$$

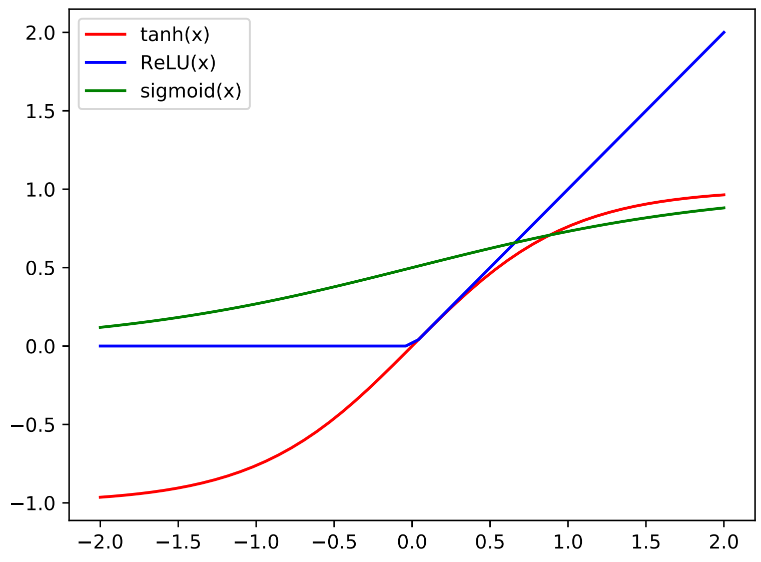

Figure 1.2: The tanh, ReLU and sigmoid activation functions plotted on the interval $[-2,2]$. The output of tanh is always in the interval $(-1,1)$ whilst the output of the ReLU function is in the interval $[0,\infty)$. The output of the sigmoid is in the interval $(0,1)$.

The softmax function is a way to normalize an unnormalized score vector p, in this case, into a vector with a probability of each class C. If the problem only contains two classes, one may also use the sigmoid (1.4) activation function, where a classification of ”1” designates one class and ”0” the other class. For regular regression problems, no activation function on the output is usually needed, since the number Y need not be bounded.

1.2 Convolutional Neural Networks

First appearing as the Neocognitron in 1980 [23] and later applied to handwritten digits in 1998 [24], Convolutional Neural Networks (CNNs) have seen a resurgence from 2013 when a CNN [25] trained on the ImageNet [26] dataset was able to substantially increase the precision compared to traditional computer-vision (CV) methods. Figure 1.3 from [27] shows the prediction accuracy of different machine learning models from 2011 to May 2021. The top-1 accuracy on the vertical axis is the percentage of times the most probable class predicted by the model was the correct class. Another metric is the top-5 accuracy, which is the percentage of times the correct class is within the top five most probable classes predicted by the model. The first CNN model in the figure from 2013 is the AlexNet structure [25]. All the subsequent state-of-the-art models have been neural networks. The traditional method from 2011 uses Scale-Invariant Feature Transform (SIFT) [28] and the Fisher Vector (FV), which is a special case of the Fisher kernel [29]. The increase in computing power and model size has played a role in the increase of accuracy from 63.3% for AlexNet with 60 million parameters to 90.2% for the Meta Pseudo Labels [30] model with 480 million parameters [27]. The number of parameters is certainly not the only factor, as for instance the ResNeXt-101 model [31] from 2018 has 829 million parameters whilst obtaining an accuracy of 85.4% [27]. The architectural nuances in the more advanced convolutional network models in Figure 1.3 are not relevant for this thesis, and we refer to [27] for a comprehensive list of different models used for the ImageNet [26] classification task. However, we note that all but one [32] of the state-of-the-art models are based on CNNs and exploit their inductive bias towards image data. The paper [32] is based on the transformer architecture [33], but has since been surpassed by CNN architectures [30, 34]. The authors of [32] note that one of the reasons their model had the state-of-the-art performance was due to them being able to pre-train on an enormous dataset of labeled images before fine-tuning their model on the actual ImageNet dataset [32]. The dataset they pre-trained on is the JFT-300M dataset owned by Google, containing 300 million images with associated (noisy) labels [35].

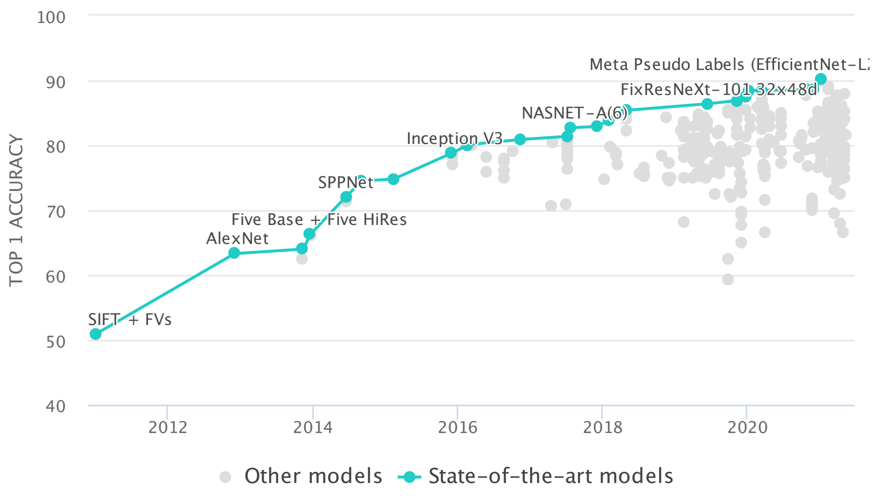

Figure 1.3: The Top-1 accuracy of different computer vision models from 2011 to May 2021. Top-1 accuracy is the percentage of times the most probable class predicted by the model is the correct class. Image credit: [27]

The basic operation of CNNs is the convolution operation. Equation (1.8) shows an example of a convolution operation. The input Z^l is usually the input image. There are some different equations for the two-dimensional convolution operation. They differ in whether the input dimensions are retained or if they are reduced. If they are retained, the filter W^l will also multiply elements at negative indices, i.e. outside the input matrix. The "imaginary" elements outside of the input matrix will not contribute to the final output. However, the common implementation found in our machine learning framework [36] does not operate this way. Instead, the convolution operator is going to reduce the input dimensions depending on the dimension of the filter.

For a 3×3 filter, it will reduce the input dimension by two for both rows and columns. For a 5×5 filter both input dimensions will be reduced by four. This is due to the filter stopping at the edges of the input matrix, and not multiplying elements outside of it. Examples are shown in equations (1.6) and (1.7), where the input matrices with dimensions 3×3 are reduced to 1×1 scalars after applying a convolution filter with dimension 3×3. Note the subscripts in equations (1.6) and (1.7), for instance $Z^l_{1:3,1:3}$ denotes the upper left 3×3 submatrix of Z^l.

When the filter has been applied across an entire row, it moves one row down and does the same process for the rest of the input matrix in equation (1.8).

$$\overset{Z^l_{1:3,1:3}}{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}} \circledast \overset{W^l}{\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}} = 0 \tag{1.6}$$

$$\overset{Z^l_{1:3,2:4}}{\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}} \circledast \overset{W^l}{\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}} = 3 \tag{1.7}$$

$$\overset{Z^l}{\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}} \circledast \overset{W^l}{\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}} = \overset{Z^{l+1}}{\begin{bmatrix} 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \end{bmatrix}} \tag{1.8}$$

In the above example, the input matrix has only one channel. Usually images have three channels, for red, blue and green. In equation (1.8), the input is $1 \times 9 \times 9$, where the leading one is the number of channels. The filter shape is $1 \times 3 \times 3 \times 1$, where the leading number indicates the number of channels in the input and the trailing one is the number of channels in the output. Any number of output filters can be used, and the choice of output channels is very similar to the choice of hidden layer size, H^l, for fully-connected networks.

Z^{l+1} in equation (1.8) is an example of a feature map from one channel. By using more channels one will obtain more feature maps.

When working with the ImageNet dataset, one may resize the images into the dimensions $3 \times 256 \times 256$, with three color channels and 256 pixels in height and width. If we assign the first convolutional layer to have 16 channels, and a 5×5 filter size, the dimension of the filter W^1 is then $3 \times 5 \times 5 \times 16$, since there are three input channels, filter size of five and 16 output channels.

The benefits of convolutional neural networks are that they are more parameter-efficient when working with image data compared to their fully-connected counterparts. In equation (1.8), the input Z^l has $6 \cdot 6 = 36$ features, and the output Z^{l+1} has $4 \cdot 4 = 16$ features. In order to perform such a transformation with a FCN, one would require a matrix with dimension 36×16. In comparison, the filter weight W^l only contains 9 parameters.

The reason CNNs are able to perform well on image data are due to the principles of locality and parameter sharing. Let us imagine we have a filter W^*. W^* is able to detect a paw. Now, it does not matter where in the input image X the paw is located, since the convolution operation is applied on the whole image with the same filter. The locality part stems from the filter W^* being able to detect local attributes in the image, irrespective of where in the image that attribute is. Parameter sharing is due to the fact that the filters are applied across the whole input image. When translating the input of a convolution operation, the output will be translated equivalently. The technical term for this is that the convolution operation is equivariant to translations [8, 37]. Equivariant means that if the input to the function is transformed a certain way, the output will be transformed in a similar way. For CNNs, this means that a translated input image will have a translated output image after applying a convolution function.

Fully-connected neural networks, however, are not to robust to translations in the input image. If a FCN is trained on dog pictures, and the paws are always in the bottom part of the image, the FCN will perform poorly if the paws start appearing in the top part of the image. This will not happen to the W^* filter, since it applies the same convolution operation over the whole input, and so any translations in the input will result in an equivalent translation in the output.

In summary, CNNs use the powerful mechanism of convolution in order to use fewer parameters while still being able to generate meaningful intermediate feature representations Z^l. However, CNNs are somewhat limited in terms of the data they can process, due to them necessitating the input being in a structured format, like the grid structure of an image.

1.2.1 Example Datasets: The ImageNet And MNIST Datasets

The ImageNet dataset [26] has been one of the principal benchmarks for computer vision in the last decade [27]. It contains images of several different object categories, like birds, vehicles, apparel and household items. Figure 1.4 shows a visualization of several images from the ImageNet dataset. There are images of car tires, butterflies, owls, typewriters, speedometers, baseball, and much more. There are a total of 1000 classes spread among 14,197,122 images [38]. Human classifiers obtain an error rate at about 5.1% [38], whilst the current best deep learning model has an error rate of 9.55%[27, 7]. Figure 1.6 shows a visualization of a set of $11 \times 11 \times 3$ filters of the AlexNet model [25]. We see that certain filters are trained to recognize certain shapes, for instance horizontal or vertical edges, or particular color patterns.

Some images from the MNIST dataset are shown in Figure 1.5. The MNIST dataset contains 60,000 labeled images of handwritten digits from zero to nine. The famous LeNet [24] architecture was used on the MNIST dataset. The images are greyscale with a height and width of 28×28. It is a rather simple dataset, and most neural network models are able to perform well on it.

Figure 1.4: A collection of images from the ImageNet dataset. Image credit: [39]

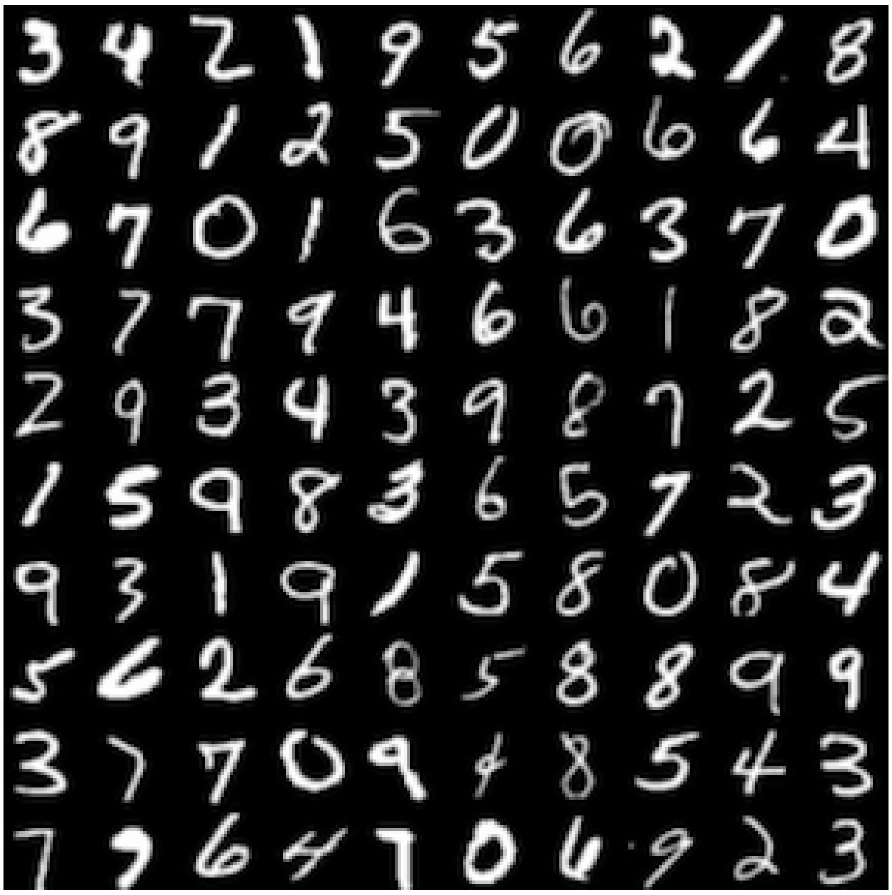

Figure 1.5: Some of the pictures from the MNIST dataset. Image credit: [40]

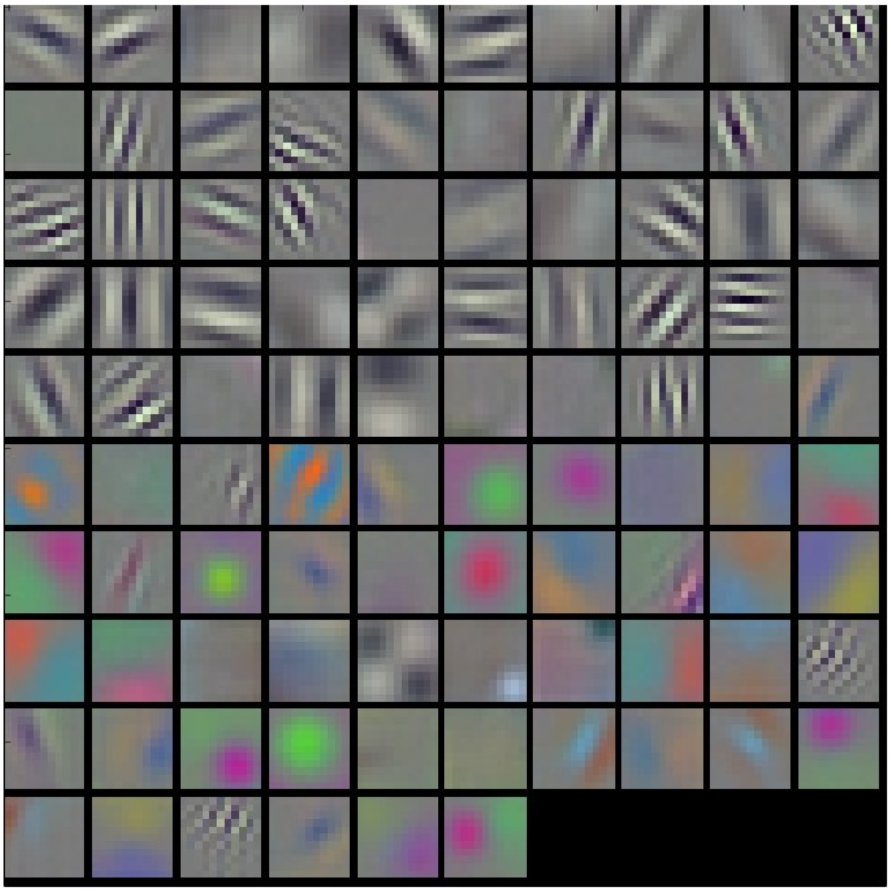

Figure 1.6: Visualization of filters with dimension $11 \times 11 \times 3$, which are part of the AlexNet [25] model. Image credit: [41]

1.3 Recurrent Neural Networks

The origins of recurrent neural networks (RNNs) can be traced back to the end of the 1980s, with papers like [42] using the newly-developed tools of backpropagation in order to apply fully-connected networks on sequences. The vanilla RNN operates in a very similar way as its fully-connected counterpart, and takes in new information for each new element that it processes in the sequence.

Figure 1.7 shows how a recurrent neural network processes an input sequence with length T_x in order to produce the output $\hat{y}$. Following the Figure 1.7, the propagation rule is based around how $a^{<t+1>}$ is generated from $a^{<t>}$, and how to finally produce $\hat{y}$. Using the ReLU activation function $\sigma(\cdot)$ from subsection 1.1, we may write the update rule as follows

$$a^{<t+1>} = \sigma(W_{aa}a^{<t>} + W_{ax}x^{<t>}). \tag{1.9}$$

The weight matrix W_{aa} controls what infromation from the previous timestep goes into the next one, and the weight matrix W_{ax} controls what information from the input goes into the current representation. This is a single-layer neural network with hidden size H, and the dimension of the matrices are $W_{aa}: H \times H$ and $W_{ax}: H \times D$. Note that we omit the bias term as this is added by the Pytorch framework [36] automatically, and it reduces the readability of the definitions and equations.

By using the softmax function from section 1.1.1, the prediction rule for $\hat{y}$ is

$$\hat{y} = \text{softmax}(W_{ya} a^{<T_x>}), \tag{1.10}$$

where the dimension of W_{ya} is $C \times H$.

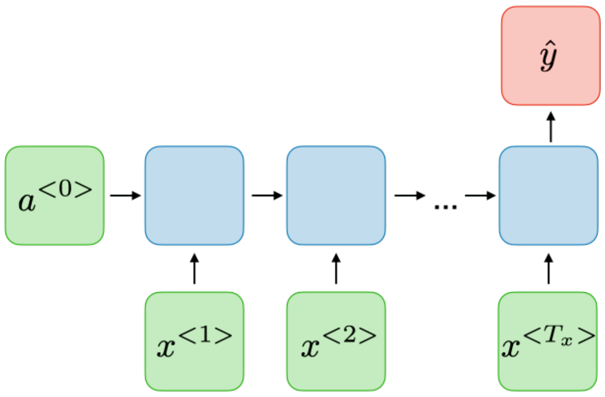

Figure 1.7: The many-to-one recurrent neural network structure. In this formulation, the only output is at the last timestep, after all of the input sequence $x^{<1>}$ through $x^{<T_x>}$ has been processed. Image credit: [22]

Recurrent neural networks are flexible, and can be used for a variety of tasks. Figures 1.8 and 1.9 show that RNNs can be designed in different ways depending on what kind of input data x and output y is desired. The setup in Figure 1.8 can be used for named entity recognition, with the inputs at time t, $x^{<t>}$, being feature vectors for words. The output at time t, $\hat{y}^{<t>}$ is a category for each word $x^{<t>}$. The setup in Figure 1.9 can be used for translation, where the input x is processed, and one successively generates output $\hat{y}^{<t>}$ until finished. Usually a stopping symbol is also one of the possible outputs in $\hat{y}^{<t>}$, in order to know when to stop the output sentence. A multi-layer RNN works by taking the output from the previous RNN as input to the subsequent RNN. As a concrete example, say we have RNN A and RNN B. RNN B takes the output of RNN A as input. Hence, after RNN A has produced hidden states $\{a^{<At>}\}_{t=1}^{T_x}$, the features $\{a^{<At>}\}_{t=1}^{T_x}$ are used as input to RNN B, producing $\{a^{<Bt>}\}_{t=1}^{T_x}$. RNN B can then predict output tokens $\hat{y}^{<t>}$ depending on the task at hand. For more information regarding RNNs we refer to the explanations on the website [22].

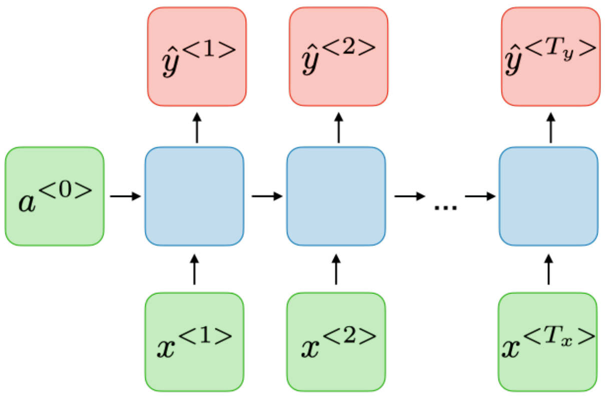

Figure 1.8: A many-to-many recurrent neural network, with predictions on every input timestep t. Such a setup can be used for named entity recognition, i.e. recognizing names in a sentence. Image credit: [22]

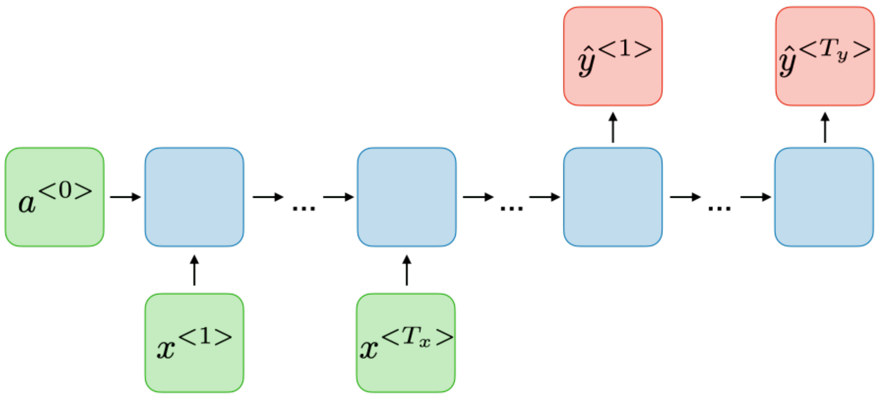

Figure 1.9: A many-to-many recurrent neural network, with predictions after all of the input sequence x has been processed. Such a setup can be used for translating between languages. After having processed the input sequence x in one language, one can have the output y be symbols for a different language. Note that in order to know when to stop the translation, it is common to predict a stopword in the output y. Image credit: [22]

1.3.1 Example Dataset: Name Classification Dataset

As a dataset example, we let the dataset X contain 1080 names from 18 different languages, with 60 names from each language. The dataset is taken from [2]. Some example name and country combinations are listed below

Arabic :	Khoury, Gerges, Asghar, Maalouf
Russian :	Katzarev, Dovgalev, Shahin, Zhdanko
Italian :	Abate, Marchesi, Lazzari, Vitolo
Scottish :	Russell, Muir, Hunter, Hamilton.

The dimension of the data X is $N \times S \times D = 1080 \times 12 \times 57$, where N is the number of examples in the data, S is the length of the longest name, and D is the length of the feature vector for each letter. Hence we have $T_x = S$. There are 57 different letters, and we have truncated the dataset in order for the longest name to be 12 characters long. If one wants to use a fully-connected network on this dataset instead, one could concatenate the feature vectors for each letter at every postion. The resulting matrix would then have dimension $N \times (S \cdot D) = 1080 \times 684$.

1.4 Graph-Convolutional Neural Networks

One of the fastest-growing new fields in machine learning is that of graph-convolutional neural networks (GNNs). They are suited to work on graph data, which could be a social network, where the nodes are user profiles and the edges denotes which people are friends. It could also be a molecule, where nodes are atoms and the edges are chemical bindings. Figure 1.10 shows an example graph with six nodes and seven edges connecting them. In order for graph neural networks to account for the edges in their forward pass, they require information on the graph in form of an edge matrix E. The dimension of the edge matrix E is $2 \times N_e$, where N_e is the number of edges in the graph. For the graph in Figure 1.10, the corresponding edge matrix E is shown in equation (1.11).

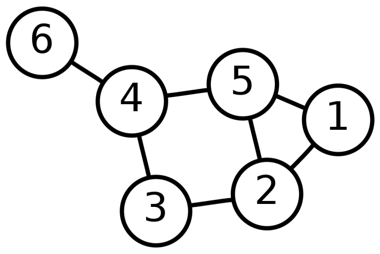

Figure 1.10: A graph with 6 nodes and 7 edges. For our graph dataset, each node will be described by a feature vector of length D. Image credit: [43].

$$\begin{pmatrix} 1 & 1 & 2 & 2 & 3 & 4 & 4 \\ 2 & 5 & 5 & 3 & 4 & 5 & 6 \end{pmatrix} \tag{1.11}$$

Graph neural networks thus are designed to operate on a graph, and take both the feature matrix X and the edge matrix E as input. This is an example of a model (GNN) desiged to work on non-euclidean data. Euclidean data is data which can be represented in $\mathbb{R}^n$. Examples of this is an image, or simply a feature vector. A graph has edges however, and there is no way one can represent both the feature vectors of each node and the edges in the graph in $\mathbb{R}^n$.

Recent interest in creating neural networks for non-euclidean data has sparked a new view of interpreting neural networks. The Geometric Deep Learning [37] community has created a taxonomy for understanding different neural architectures, for instance CNNs, RNNs, GNNs and also the transformer [33] architectures in terms of symmetries and invariants [37, 44]. The authors [44] state “We believe that the current state of affairs in the field of deep (representation) learning is reminiscient of the situation of geometry in the nineteenth century ... Geometric Deep Learning is an umbrella term we introduced in [44] referring to recent attempts to come up with a geometric unification of ML similar to Klein’s Erlangen Programme. It serves two purposes: first, to provide a common mathematical framework to derive the most successful neural network architectures, and second, give a constructive procedure to build future architectures in a principled way.” Note that the geometric deep learning programme of [37] is not entirely related to graph neural networks, or vice versa. However, the recent work of [37] can trace its history from earlier designs of graph neural networks from the same authors [45]. To summarize, graph neural networks have paved the way for a new way of understanding neural networks, where the focus has shifted into creating models which by design satisfy symmetry properties of the data [46, 37]. This has also been done in physics to create neural networks invariant to different choices of gauges [47].

In this thesis we will use one of the simplest GNN architectures, namely the GCN architecture from [48]. The learnable parameters for graph neural networks are also weight matrices W^l, but the weight matrices will take on a different role than the fully-connected matrices in (1.2). A graph neural network operates on a graph, so in addition to specifying X, edge information is also required. Here we follow the notation from the Pytorch

Geometric framework [49] where the edges between examples are undirected and denote the similarity between the examples. The definition of similarity depends on the dataset. For our example dataset in Section 1.4.1, we have a dataset consisting of scientific articles, and similarity between article A and B means that paper A cited paper B, paper B cited paper A or both papers cited each other.

The propagation rule for the GCN architecture is as follows

$$Z^{l+1} = \sigma\big(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}Z^{l}W^{l}\big), \tag{1.12}$$

where σ is a nonlinear function as in (1.2), $\tilde{A} = A + I_N$ is the adjacency matrix of the undirected graph with added self-connections, and $\tilde{D}$ is the diagonal degree matrix of $\tilde{A}$, defined as $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. For the example edge matrix E in (1.13) we will get the matrix $\tilde{A}$ in (1.14).

$$\begin{pmatrix} 1 & 3 \\ 2 & 1 \end{pmatrix} \tag{1.13}$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \tag{1.14}$$

The ones along the diagonal in equation (1.14) come from the identity matrix I_3, and the off-diagonal elements represent the edges from the edge matrix (1.13).

For the GCN architecture, each node's updated representation in Z^{l+1} is an aggregation of that node's representation in Z^{l} and of its neighbors. When considering a single example n from the data, its updated representation takes the form

$$Z_n^{l+1} = \sigma\left(\left(\sum_{j\in\mathcal{N}(n)} \frac{1}{\sqrt{\hat{d}_j\hat{d}_n}} Z_j^{l}\right)W^{l}\right). \tag{1.15}$$

The term $\mathcal{N}(n)$ is the set of all neighbors of node n in the graph including itself, Z_j^l is the previous representation of the j'th neighbor of n, and $\hat{d}_i = \sum_{j\in\mathcal{N}(i)} 1$. So $\hat{d}_i$ counts the number of neighbors of node i, including itself. We recognize (1.15) as a fully-connected forward propagation (1.2), but having replaced the previous representation Z_n^l by the term in the inner parenthesis in (1.15), which is a weighted mean of its own representation and that of its neighbors. For data that can be expressed as graphs, graph neural networks often outperform fully-connected networks [48]. This is due to this graph-specific summation term in their forward propagation rule. Note that the dimensions of the Z^i and W^i's are the same as in the fully-connected case but they are now also affected by the graph structure contained in $\tilde{A}$ and $\tilde{D}$.

Fully-connected networks can be applied on more varied problems than graph neural networks since graph neural networks require that the edges between the nodes are specified. However, if one has a dataset that contains edges between nodes, for instance data about a social network or a molecule, a graph network will generally be a more powerful machine-learning method than a fully-connected network. Both the fully-connected and the graph neural network require the node features represented in the data X with dimension $N \times D$. However, the graph neural network also requires the edge information E with dimension $2 \times N_E$. In doing so, the learnable weights W^l in the graph neural network will learn about each node and its neighbors. This is more powerful than learning about each node by its own features, which fully-connected networks do. Since the graph neural network is explicitly designed to operate on graph data, we say that the graph neural network model contains an inductive bias to operate on graph data. Our goal then is to investigate whether the inductive bias of the graph neural network impacts the test-time accuracy and whether the information bottleneck method explained in Section 3 can provide intuition as to which architecture is able to generalize the most from the features X.

1.4.1 Example Dataset: The Cora Citation Dataset

The Cora dataset is a graph-like dataset meaning that in addition to a feature matrix X with dimensions $N \times D$ where N is the number of examples and D is the number of features, with the edge matrix E containing the edges of the graph. The Cora dataset contains 2708 nodes and 5429 edges. The edges in the Cora data describe citations. They are undirected so an edge between nodes A and B means either paper A citing paper B, paper B citing paper A or both citing each other. Moreover, each node in the Cora dataset is described by a feature vector of length $D = 1433$ where each element of that vector is the count of a predetermined word (bag-of-words feature vector). Each of the 2708 articles has one of seven labels indicating which scientific field their article comes from. As an example, let us choose to count the occurrence of the words ("like", "movie", "to", "watch", "game"). It is usual to just count the word stems, which means that ”likes” and ”liked” will both be counted under the word stem ”like”. Two example texts are provided in (1.16) and (1.17).

$$\text{Robert likes to watch movies. Marie likes movies too.}$$
$$(2,2,1,1,0) \tag{1.16}$$

$$\text{Mary also likes to watch football games}$$
$$(1,0,1,1,1) \tag{1.17}$$

Chapter 2

Backpropagation and Neural Network Training

2.1 Backpropagation

Backpropagation is the manner in which all neural networks learn (adjust the weights). Backpropagation relies on the sequential structure of neural networks in order to compute gradients to update the parameters of the network. The gradients are computed in a sequential manner, by using the chain rule from calculus. Due to our method relying on analyzing the training procedure for a neural network, it is worth understanding in-depth how a neural network actually trains.

2.1.1 Single Hidden Layer Example

We start with an illustrating example, a single-layer neural network. Its forward propagation rule is as follows

$$\hat{Y} = W^2\sigma(W^1 X), \tag{2.1}$$

where X is the input, and $\sigma(x)$ is an activation function, for instance the ReLU function $= \max(0, x)$. We consider a regression objective, where the loss function is

$$L(Y, \hat{Y}) = \frac{1}{2}\|Y - \hat{Y}\|_2^2 \tag{2.2}$$

Our desire is to find the gradient of the loss function $L(Y, \hat{Y})$ with respect to the model parameters W^1 and W^2.

When dealing with neural networks, it is convenient to introduce the Einstein summation method when interpreting matrix multiplications. The Einstein summation convention states that repeated indices on the same side of an equation is implicitly summed over. For instance, element (i, j) of the matrix multiplication $A = BC$ is represented as

$$A_{i,j} = B_{i,k}C_{k,j}$$

which is equivalent to

$$A_{i,j} = \sum_k B_{i,k}C_{k,j}.$$

The gradients we are looking for are

$$\frac{\partial L}{\partial W^1}$$

and

$$\frac{\partial L}{\partial W^2}.$$

The chain rule of calculus can be applied when we have a chain of functions $P = f(g(Q))$. We can denote $Z = g(Q)$. The chain rule of calculus then states

$$\frac{\mathrm{d}P}{\mathrm{d}Q} = \frac{\mathrm{d}f}{\mathrm{d}Q} = \frac{\mathrm{d}f}{\mathrm{d}Z}\frac{\mathrm{d}Z}{\mathrm{d}Q}.$$

Hence, the chain rule of calculus has reduced the problem of finding $\frac{\mathrm{d}f}{\mathrm{d}Q}$ into finding the two derivatives $\frac{\mathrm{d}f}{\mathrm{d}Z}$ and $\frac{\mathrm{d}Z}{\mathrm{d}Q}$.

We are going to be working with indices when finding gradients, and an object that commonly appears is the Kronecker delta function, defined as

$$\delta_{i,j} = \begin{cases} 1, \text{ if } i = j \\ 0, \text{ otherwise} \end{cases} \tag{2.3}$$

Note that for a matrix A, we have the relation $\frac{\partial A_{k,q}}{\partial A_{i,j}} = \delta_{i,k} \cdot \delta_{q,j}$.

Referring to our single-layer neural network (2.1), we denote $Z = \sigma(W^1 X)$. The gradients for W^1 and W^2 then take the form

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial W^2} \tag{2.4}$$

and

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial W^1} = \frac{\partial L}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial Z} \frac{\partial Z}{\partial W^1}. \tag{2.5}$$

We begin by finding $\frac{\partial L}{\partial \hat{Y}}$.

$$\begin{aligned} \frac{\partial L}{\partial \hat{Y}_{ij}} &= \frac{1}{2} \frac{\partial}{\partial \hat{Y}_{ij}} \sum_{k,q} \left(\hat{Y}_{kq} - Y_{kq} \right)^2 \\ &= \sum_{kq} \left(\hat{Y}_{kq} - Y_{kq} \right) \odot \frac{\partial \hat{Y}_{kq}}{\partial \hat{Y}_{ij}} \\ &= \sum_{kq} \left(\hat{Y}_{kq} - Y_{kq} \right) \odot \delta_{ik} \delta_{jq} \\ &= \hat{Y}_{ij} - Y_{ij} = \frac{\partial L}{\partial \hat{Y}_{ij}} \end{aligned}$$

In order to obtain the gradients for W^1 and W^2, we need the gradients $\frac{\partial L}{\partial \hat{Y}}$ and $\frac{\partial \hat{Y}}{\partial W^2}$. We note that $\hat{Y} = W^2 Z$. Since element i, j of $\hat{Y}$ is $\hat{Y}_{ij} = W^2_{is} Z_{sj}$, we obtain

$$\begin{aligned} \frac{\partial \hat{Y}_{ij}}{\partial W^2_{pr}} &= \frac{\partial}{\partial W^2_{pr}} W^2_{is} Z_{sj} \\ &= \delta_{ip} \delta_{sr} Z_{sj} = \delta_{ip} Z_{rj}. \end{aligned}$$

By the chain rule of calculus, the gradient $\frac{\partial L}{\partial W^2}$ is thus

$$\begin{aligned} \frac{\partial L}{\partial W^2_{pr}} &= \frac{\partial L}{\partial \hat{Y}_{ij}} \frac{\partial \hat{Y}_{ij}}{\partial W^2_{pr}} = \left(\hat{Y}_{ij} - Y_{ij} \right) \delta_{ip} Z_{rj} \\ &= \left(\hat{Y}_{pj} - Y_{pj} \right) Z_{rj} = \left(\hat{Y}_{pj} - Y_{pj} \right) Z^T_{jr} \end{aligned}$$

which in matrix form is

$$\frac{\partial L}{\partial W^2} = \left(\hat{Y} - Y\right) Z^T. \tag{2.6}$$

As for finding $\frac{\partial L}{\partial W^1}$, we decompose it into $\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W^1}$.
As for the gradient of the ReLU function, $\sigma(x) = \max(0, x)$, it is

$$\frac{\partial \sigma(x)}{\partial x}_{ij} = \begin{cases} 1, \text{ if } x_{ij} > 0 \\ 0, \text{ if } x_{ij} \leq 0. \end{cases}$$

The gradient $\frac{\partial L}{\partial Z}$ is found as follows

$$\begin{aligned} \frac{\partial L}{\partial Z_{ij}} &= \frac{\partial L}{\partial \hat{Y}_{kp}}\frac{\partial \hat{Y}_{kp}}{\partial Z_{ij}} = \left(\hat{Y}_{kp} - Y_{kp}\right)\frac{\partial\left(W^2_{kr}Z_{rp}\right)}{\partial Z_{ij}} \\ &= \left(\hat{Y}_{kp} - Y_{kp}\right) W^2_{kr}\delta_{ir}\delta_{pj} = \left(\hat{Y}_{kj} - Y_{kj}\right) W^2_{ki} \\ &= (W^2)^T_{ik}\left(\hat{Y}_{kj} - Y_{kj}\right) \\ \frac{\partial L}{\partial Z} &= (W^2)^T\left(\hat{Y} - Y\right) \end{aligned}$$

Moreover, $\frac{\partial Z}{\partial W^1}$ is found as follows

$$\begin{aligned} \frac{\partial Z_{ij}}{\partial W_{pq}} &= \sigma'\left(W^1 X\right)_{ij}\frac{\partial\left(W^1_{ir}X_{rj}\right)}{\partial W^1_{pq}} \\ &= \sigma'\left(W^1 X\right)_{ij} X_{kj}\delta_{ip}\delta_{qr}. \end{aligned}$$

To find $\frac{\partial L}{\partial W^1}$, we compute

$$\begin{aligned} \frac{\partial L}{\partial W^1_{pq}} &= \frac{\partial L}{\partial Z_{ij}}\frac{\partial Z_{ij}}{\partial W^1_{pq}} \\ &= \left[(W^2)^T_{ik}\left(\hat{Y}_{kj} - Y_{kj}\right) \odot \sigma'(W^1 X)_{ij}\right] X_{kj}\delta_{ip}\delta_{qr} \\ &= \left[(W^2)^T_{pk}\left(\hat{Y}_{kj} - Y_{kj}\right) \odot \sigma'(W^1 X)_{pj}\right] X_{qj} \\ \frac{\partial L}{\partial W^1} &= \left[\left((W^2)^T(\hat{Y} - Y)\right) \odot \sigma'(W^1 X)\right] X^T \end{aligned}$$

We have now found the gradients $\nabla W^1, \nabla W^2$ for the parameters W^1 and W^2 in our model by using the backpropagation algorithm. We identify the key part of the backpropagation algorithm as decomposing the gradient computation $\frac{\partial L}{\partial W^1}$ into $\frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W^1}$ where $Z = \sigma(W^1 X)$. By using the same procedure as above, one can find the gradient for any layer W^l in a deep fully-connected network by using the chain rule $\frac{\partial L}{\partial W^l} = \frac{\partial L}{\partial Z^l}\frac{\partial Z^l}{W^l}$.

2.1.2 Backpropagation in PyTorch

The main motivation for using a framework like PyTorch [36] or Tensorflow [50] is that it is sufficient to specify the forward pass of the model which defines how to go from the input X to the prediction $\hat{Y}$. Having done this, the frameworks are able to compute all the necessary gradients $\{\nabla W^l\}_{l=1}^L$ for all the layers. The way this is achieved is through the backward pass which uses backpropagation. Backpropagation is very similar to reverse-mode autodifferentiation [51].

When working in PyTorch, all of the data is specified in terms of tensors. Tensors are matrices, but can have multiple dimensions, not just two. The tensor class of PyTorch has attributes such as requires_grad, which specifies if a gradient for that tensor is required. If one has some constants in the model, these would have requires_grad = False. Figure 2.1 shows an example of a computational graph in PyTorch. The inputs are x and y, where x has requires_grad = True. This forces the output $z = x \cdot y$ to have requires_grad = True, since in order to compute the gradient for x we also need to have the gradient for z since the gradients moves in the opposite direction of the backward pass. If z had requires_grad = False we would have no information about how to compmute the gradient for x. The is_leaf attribute is simply to indicate that one does not have to continue the backward pass after reaching that node.

When specifying a function in PyTorch, it is required to specify both the forward and backward pass. PyTorch already has specifications for a wide variety of functions, for instance the tanh, sigmoid and ReLU functions, together with multiplication, addition and many more functions documented at [52]. When we then apply those functions in our neural networks, we use the forward pass of that function, and the PyTorch engine can then access the backward pass for that function in order to compute gradients.

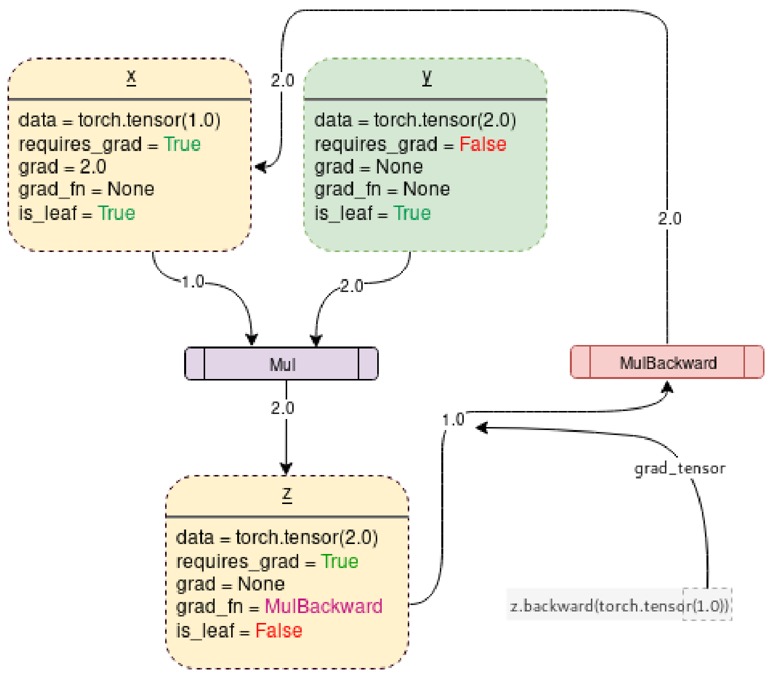

Figure 2.1: An example of a computational graph created by PyTorch. X and y are the inputs, where x hasrequires_grad = True. The "Mul" operator also has a defined MulBackward function, which computes the gradients for x and y. Image credit: [53].

In Figure 2.1, backpropagation starts by calling the backward method of tensor z. It tells the PyTorch engine, PyTorch Autograd, to start computing the gradients of all the tensors involved in producing z which have requires_grad = True. The MulBackward method specifies how to compute the gradients $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. Since y has requires_grad = False, however, the MulBackward operation will not compute $\frac{\partial z}{\partial y}$. Internally, MulBackward has stored that $\frac{\partial z}{\partial x} = y = 2.0$. Then the gradient for x is $\nabla x = 1.0 \cdot \frac{\partial z}{\partial x} = 1.0 \cdot 2.0 = 2.0$. The number 1.0 comes from initializing the backward method of z with 1.0, and can be used to scale the gradients if necessary.

The schematic in Figure 2.1 can be used to understand any neural network model in PyTorch, albeit with different inputs x, y and functions. We note that if one defines a function, $g(\cdot)$, where g is defined through other PyTorch functions, g will also be considered a PyTorch function since PyTorch Autograd can recursively decompose g into applications of functions which have the PyTorch forward and backward pass defined.

We can draw several analogies from the backpropagation example in Section 2.1.1 and Figure 2.1. By replacing x with Z^l, y with W^l and z with Z^{l+1}, one can use the gradient computation algorithm from Section 2.1.1 in order to recursively compute the gradient for any hidden layer. Code listing 2.1 shows an example neural network with a single layer in PyTorch, without any hidden layers. By calling the backward method of the tensor z, PyTorch would perform the exact same operation as is done when finding $\frac{\partial Z}{\partial W^1}$ in Section 2.1.1. Moreover, this backward function call can be understood with analogies to Figure 2.1, where the weight W in the nn.Linear object takes the spot of x, and x in Figure 2.1 takes the spot of y, since y does not require a gradient.

The backward operations required in order to compute $\frac{\partial z}{\partial W}$ will be the backward method defined by the nn.Linear object and of the torch.relu function.

```python
import torch
import torch.nn as nn

class SingleLayerFCN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = nn.Linear(in_features=input_dim, out_features=output_dim)
    def forward(self, x):
        z = torch.relu(self.linear(x))
        return z
```

Listing 2.1: Single-Layer Fully-Connected Network in PyTorch

2.2 Neural Network Training

2.2.1 Specifying the Training Procedure In Tensorflow and Keras

When doing deep learning, one commonly uses an already implemented deep learning framework in order to be more efficient and write more readable code (e.g., Tensorflow, PyTorch or Flux). In these frameworks it is required that one specifies the forward pass that defines how to generate $\hat{Y}$ from X, which loss function to use, and which optimization algorithm to use. The forward pass is specified by selecting the number of layers L in the model and the dimensions of the matrices $W_l, 1 \leq l \leq L$. Larger matrices W_l result in more parameters and a model able to learn more complex relationships, however issues arise due to overfitting, explained in Section 2.2.5. An example program for training a neural network with TensorFlow [50] parsed through Keras [54] on 28×28 black-and-white images of numbers 0 through 9 is shown in the code code listing 2.2, taken from their website (tensorflow.org/overview). The keras interface has many high-level abstractions, which makes it suited for creating concise neural network training setups. In TensorFlow, a fully-connected layer is called a dense layer.

```
1  import tensorflow as tf
2  mnist = tf.keras.datasets.mnist
3
4  (x_train, y_train),(x_test, y_test) = mnist.load_data()
5  x_train, x_test = x_train / 255.0, x_test / 255.0
6
7  model = tf.keras.models.Sequential([
8    tf.keras.layers.Flatten(input_shape=(28, 28)),
9    tf.keras.layers.Dense(128, activation='relu'),
10   tf.keras.layers.Dropout(0.2),
11   tf.keras.layers.Dense(10, activation='softmax')
12 ])
13
14 model.compile(optimizer='adam',
15               loss='sparse_categorical_crossentropy',
16               metrics=['accuracy'])
17
18 model.fit(x_train, y_train, epochs=5)
19 model.evaluate(x_test, y_test)
```

Listing 2.2: Training a Model with Tensorflow

Lines 1 through 5 load the data and normalize it. The data X has dimensions $N \times 28 \times 28$ with integer values in the range $[0, 255]$, which is the pixel intensity at that location. Lines 7-12 define the forward pass of the model. An input X_i of dimension 28×28 is first flattened into an array of dimension 784. Then follows matrix multiplication with the weight W_1 of dimension 784×128, followed by a ReLU activation. The representation of X_i after this ReLU layer is Z_1^i. The dropout layer randomly sets a proportion, in this case $0.2 = 20\%$, of the activations in Z_1^i to zero for a single pass. The dropout layer consequently generates the output $d(Z_1^i)$. The output $d(Z_1^i)$ is not considered as a representation of the data, since the dropout layer is only used during training to help the network generalize. More on dropout in Section 2.2.5. The final dense layer has a layer weight W_2 of dimension 128×10 with a softmax activation and generates the prediction $\hat{Y}$. The softmax function is defined in Equation (1.5).

The optimizer keyword when compiling the model (line 14) specifies which optimization algorithm to use. The optimizer specifies how the weights W_1 and W_2 in the network are updated when the model is trained during the model fitting stage. A common choice is the *adam* optimizer [55] with adaptive learning rates. For more on the *adam* optimizer we refer to Section 2.2.4.

2.2.2 Measuring Model Fit Quality With Log Likelihood

Let (X, Y) be the given data, with $y_i \in \{0, 1, \dots K\}$ and $x_i \in \Omega$ where $\Omega \subset \mathbb{R}^n$, with $X = (x_1, x_2, \dots x_N)$. This is a classification problem with data in $\mathbb{R}^n$ having K different classes. One considers each datapoint (x_i, y_i) as a random variable, and further assumes that all N random variables are independent and identically distributed (iid). The distribution of the data is defined by the unknown distribution p, where $p(x_i) = y_i$. One can think of y_i as either being a number indicating the class, as in $y_i \in \{0, 1, \dots K\}$, or y_i being a vector indicating the location for the correct class, for instance $y_i = (0, 1, 0, \dots, 0)$ if the correct class is 2. The likelihood and log-likelihood of the sample under the distribution p is then

$$L(p) = \prod_{i=1}^{N} \prod_{k=1}^{K} p_{i,k}^{y_{i,k}}, \tag{2.7}$$

$$\frac{1}{N} \log L(p) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{i,k} \log p_{i,k}. \tag{2.8}$$

Here, we have scaled the log likelihood by $\frac{1}{N}$ in order to normalize the magnitude of the loss. The domain of p is Ω, and p outputs a probability density for each sample n, $p_n = (p_{n,1}, p_{n,2}, \dots p_{n,K})$ with $p_{n,1} + \cdots + p_{n,K} = 1$. If the correct class for sample n is class 2, then $y_n = (0, 1, 0, \dots 0)$. Knowing the labeled pairs (x_n, y_n) allows us to train a probability density model p_θ with parameters θ such that $p_\theta(x_n) \approx y_n$.

The neural network model p_θ contains L layers with weights W_l, $1 \le l \le L$, where each gradient ∇W_l is found by backpropagation and optimized with gradient descent. The whole training process relies upon choosing a set of L weights $\{W_l\}_{l=1}^{L}$ which yields an output $p_\theta(X)$ close to Y by minimizing the loss function (2.7).

2.2.3 Batching

The loss function (2.7) is a mean over all the examples in the dataset. However, it is often the case that the whole dataset, including all the weights W_l, the gradients ∇W_l and the activations Z_l are too big to be stored in memory. One can then randomly subsample the dataset X to create a batch with fewer elements, say B. This results in $n_B = \lfloor N/B \rfloor$ batches that must be processed in order to iterate through the dataset once. One such iteration is called an epoch. This process is repeated for as many epochs as necessary, and the data is usually shuffled to obtain different batches on each epoch. Hence for each epoch, the network will have been updated n_B times.

Another reason for batching the data is that the gradients ∇W_l become noisier since there are fewer datapoints in a batch B than in the whole dataset N. Introducing this noise makes the network less prone to overfit on the training data (X, Y) [8]. More on overfitting in Section 2.2.5.

2.2.4 Optimizing The Objective

After having obtained the gradients ∇W_l from backpropagation, one can update the weights W_l. The simplest way is the following update rule, called Stochastic Gradient Descent (SGD)

$$W_l^{k+1} = W_l^k - \alpha \nabla W_l^k. \tag{2.9}$$

The iterator k will range through $n_B \cdot E$ numbers if E is the number of epochs and n_B is the number of batches in an epoch. The learning rate α gives the step size of the optimization, and a common choice is setting $\alpha = 0.001$ [56]. However each problem is different, and in more complicated models only a certain range of values for α will work. Choosing α too small would yield no training progress, whilst choosing α too large will produce numerical instabilities and inability to converge to a local (or global) minima.

The layer weight W_l has dimension $h_{l-1} \times h_l$, where h_l denotes the representation at step l, with $h_0 = X$. It could be that different elements in the weight matrix W_l requires different learning rates in order to achieve convergence. In Figure 2.2 there are two scalar parameters $w1, w2$ to optimize. Also, Figure 2.2 is color coded such that larger values of the cost function is darker. One sees that in the direction $(1, 1)$ from the red center plot, the cost function changes rapidly. However, in the direction $(1, -1)$, it is almost constant. In order to optimize such objectives easier, the optimization method *adam* [55] proposes to track a running mean of the gradient ∇W_l for each layer, and a running mean for the squared gradient, $(\nabla W_l)^2$. Let m_l^k be the running mean for the gradient ∇W_l, and v_l^k be the running mean for the squared gradient $(\nabla W_l)^2$, at step k. The update rule for *adam* is then as follows

$$\begin{aligned} W_l^{k+1} &= W_l^k - \alpha \cdot \frac{\hat{m}_l^k}{\sqrt{\hat{v}_l^k} + \epsilon} \\ \hat{m}_l^k &= \frac{m_l^k}{1-\beta_1^k} \\ \hat{v}_l^k &= \frac{v_l^k}{1-\beta_2^k} \\ m_l^k &= (1-\beta_1)\cdot \nabla W_l^k + \beta_1 m_l^{k-1} \\ v_l^k &= (1-\beta_2)\cdot (\nabla W_l^k)^2 + \beta_2 v_l^{k-1}, \end{aligned} \tag{2.10}$$

where $\epsilon > 0$ is used to avoid division by zero. The parameters β_1, β_2 for calculating the running mean are usually chosen as $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The running mean m_l^k for the gradient ∇W_l serves to alleviate the noise introduced by batching (Section 2.2.3), and by dividing with $\sqrt{\hat{v}_l^k}$ one effectively "normalizes" the optimization landscape in each direction, yielding a more efficient optimization process [55]. For visualizations of different optimization methods and how to implement them we refer to the course website of CS231n [57].

Even though the update rule in (2.9) and (2.10) are different, the gradient at step k, ∇W_l^k, always follows from the backpropagation algorithm. The methods differ in how they use the gradient information in order to optimize the parameters.

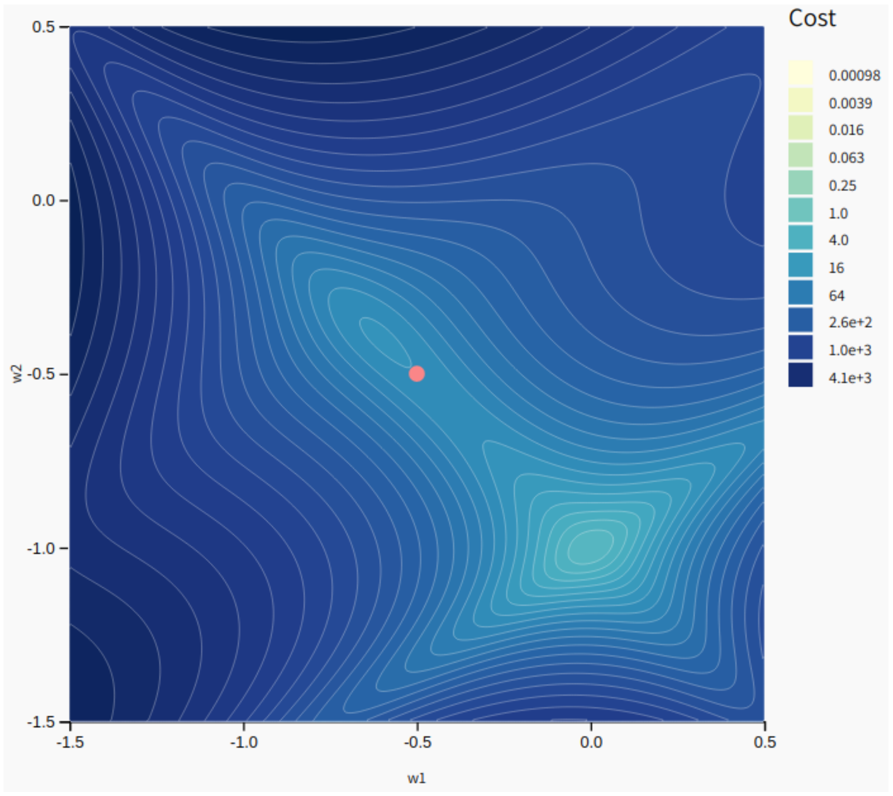

Figure 2.2: A heat plot of a two dimensional optimization problem with scalar parameters $w1$ and $w2$. The darker regions indicate a higher loss. Image credit: [58]

2.2.5 Overfitting and Regularization

Figure 2.3 shows the training and validation accuracies over several epochs. Since we are measuring the accuracy of our model we are considering a classification problem. The accuracy, either train or validation, is the percentage of correctly classified data points. When there is a small gap between the training and validation accuracy curves, there is little overfitting, which is good. If the gap between the training and validation accuracy curves is large, it is beneficial to introduce a regularization of the training.

Regularization serves to help the network generalize by enforcing penalties when the network learns on the training data. In Section 2.2 we have used dropout, which is a form of regularization. When using dropout, a random subset of the activations in a layer l, Z_l, are set to zero. The next layer thus cannot use all the information in Z_l since the dropout has set elements to zero. When elements in Z_l are set to zero they do not affect the next representation Z_{l+1}.

Dropout solves the problem of co-adaptation. With co-adaptation, the next

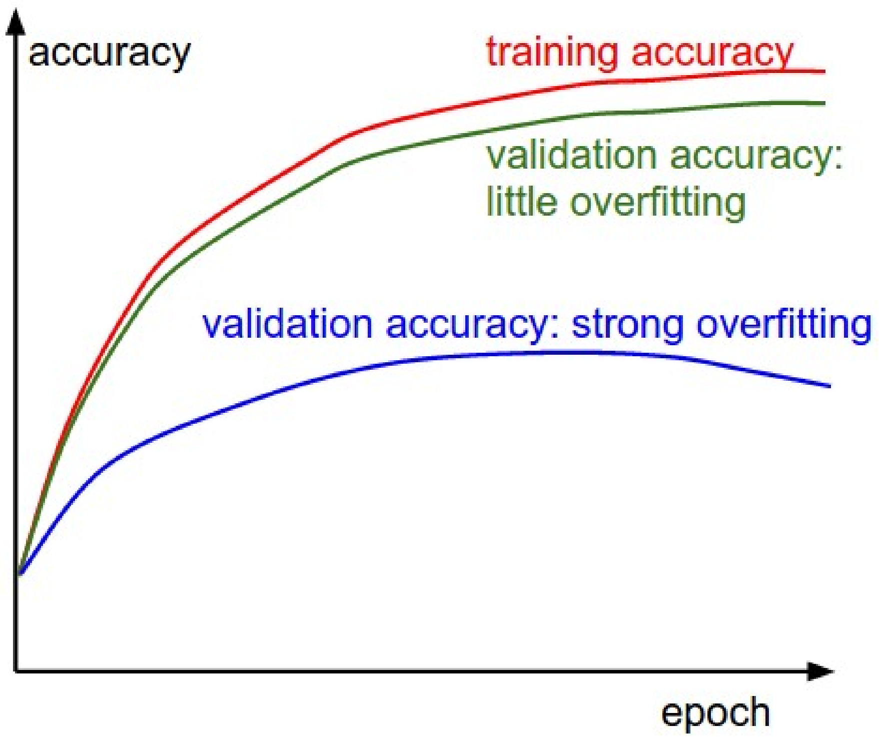

Figure 2.3: A plot of the validation and training accuracies across epochs. It is straightforward to identify if the neural network is overfitting or underfitting based on the distance between the training accuracy curve and the validation accuracy curve. Image credit: [59].

representation Z_{l+1} is highly reliant on the previous representation Z_l. Co-adaptation appears in neural network since the input is sequentially generated. A bad representation Z_l could yield and even worse Z_{l+1}. With dropout however, the layer weight W_{l+1} needs to be able to adapt to many different Z_l's during training, which makes the representation Z_{l+1} more robust to different Z_l.

Another form of regularization is adding a penalty for the magnitude of the layer weights $\{W_l\}_{l=1}^{L}$ to the loss function. In deep learning, the L2-norm is most commonly used as a penalty. An example of an L2 regularized loss function and resulting gradient is shown in equation (2.11) and (2.12), respectively. The neural network with parameters θ is denoted as f_θ where $\hat{Y} = f_\theta(X)$ and Y are the correct labels from the data. The parameter λ determines the weight put on the penalty term $\|W_l\|_2^2$. When using gradient descent to minimize the objective (2.11), we are also going to minimize each of the W_l's L2 norms, which forces the weights $\{W_l\}_{l=1}^{L}$ closer to zero. When the layer weights W_l are large in magnitude, they are able to change the previous representation Z_{l-1} drastically. However, the large magnitude in the elements of W_l means it becomes more sensitive to changes in Z_{l-1} and

$Z_0 = X$. Since one seeks a neural network that is robust for different Z_{l-1} and X, the layer weights W_l can be forced closer to zero by applying a penalty such as the L2-norm penalty.

$$L_{reg}((X,Y)|\theta) = L(\hat{Y},Y) + \frac{\lambda}{2}\sum_{i=1}^{L} \|W_i\|_2^2 \tag{2.11}$$

$$\nabla W_i = \frac{\partial L}{\partial W_i} + \lambda W_i \tag{2.12}$$

Since deep neural networks have many parameters and are difficult to optimize, the approach of adding a penalty (2.11) to the objective is used instead of constraining the domains of the elements in the matrices $\{W_i\}_{i=1}^{L}$. The penalty will alter the obtained gradients with an additive term. Modern neural networks can have millions of parameters, so setting up a constrained optimization problem solver for so many parameters is computationally infeasible. If a neural network f_θ has a million parameters it would require storing $(10^6)^2$ values when optimizing the constraints [51]. However, some progress has been made for optimizing large networks while enforcing hard constraints [60, 61].

Altering the gradients with additive terms is also commonly done in physics-informed neural networks (PINN) [62]. PINN is an example of adding expert knowledge in the system through creating custom loss functions. The network is guided towards predictions that fulfill certain criteria equations added to the loss function, and these act as a kind of regularization [62].

Figure 2.4 shows level curves of an objective function, in red, together with the constraint sets for the L1 (Left) and L2 (Right) norms shaded in cyan. The optimal solution to the unconstrained problem is $\hat{\beta} = (\hat{\beta}_1, \hat{\beta}_2)$. The L1 penalty is defined, for a matrix A, as $\|A\|_1^1 = \sum_{i,j} |A_{i,j}|$ where $|a|$ denotes the absolute value of the scalar a. Figure 2.4 indicates that different constraints will lead to different optimal solutions that satisfy those constraints, and similarly different penalties will lead to different optimal solutions under those penalties.

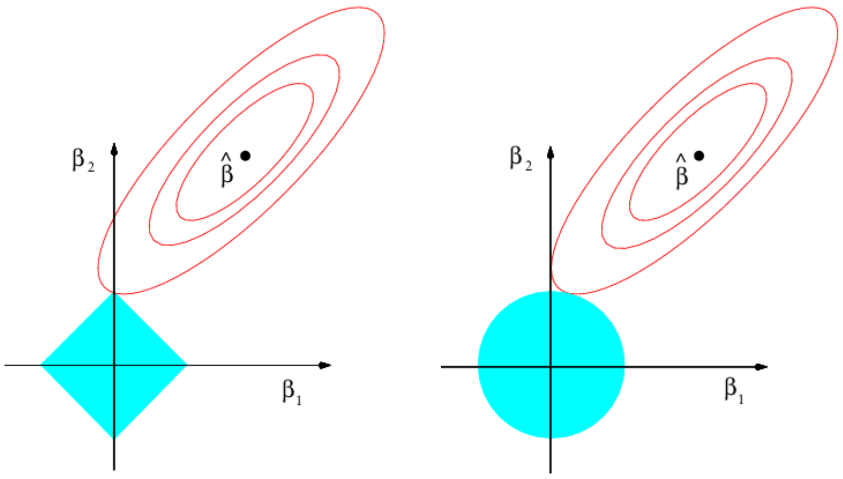

Figure 2.4: The constraint sets for the L1 penalty (Left) and L2 penalty (Right) are colored in cyan, whilst level curves of the objective function is colored in red. The optimal solution to the unconstrained problem is $\hat{\beta} = (\hat{\beta}_1, \hat{\beta}_2)$. The constraint sets affect what the optimal solution of the constrained problem will be. The areas in cyan are examples of hard constraints on the parameter values (β_1, β_2) whereas a regularization term in the loss function will pull $\hat{\beta}$ towards the edge of the cyan regions. Image credit: [12].

Chapter 3

Information Bottleneck And Mutual Information

3.1 The Information Bottleneck

The method relies on tracking the representations from every layer during training, and finally to visualize a two-dimensional information plane. The axes of the information plane is the mutual information (3.1) between the input X and a representation Z^l on the first axis, and the mutual information between label Y and Z^l on the second axis. Note that we will be using superscripts for layer numbers. Figure 3.1 (taken from [63]) illustrates the information-plane for a five layer neural network. The early layers contain the most information about the input data and consequently also much information about the output Y. The arrows from nodes A to E on the left in Figure 3.1 indicate the trajectory that layer five follows during the training process. Each green line connects different layers in the same epoch. Hence we see a tendency for the final layers to increase their information about Y during training whilst keeping little information about X, meaning they are able to generalize away unnecessary features in X. Another part of the information bottleneck method is that of tracking gradients during training. The method will be explained in section 3.1.2, despite gradient tracking not being the focus of this thesis.

3.1.1 The Information Plane

The information plane in Figure 3.1 visualizes the representations learned from each layer during the course of progress. The example in this figure is a neural network with five hidden layers, labeled L1 through L5. L5 is the final layer before the prediction layer $\hat{Y}$. There is compression in the network, in the sense that L5 contains less information about the input data, but more information about the output data. This is interpreted as the representation learned by L5 is a generalization of the representations learned by earlier layers, retaining information about the task $\hat{Y}$ but discarding unnecessary variance and information from the input X.

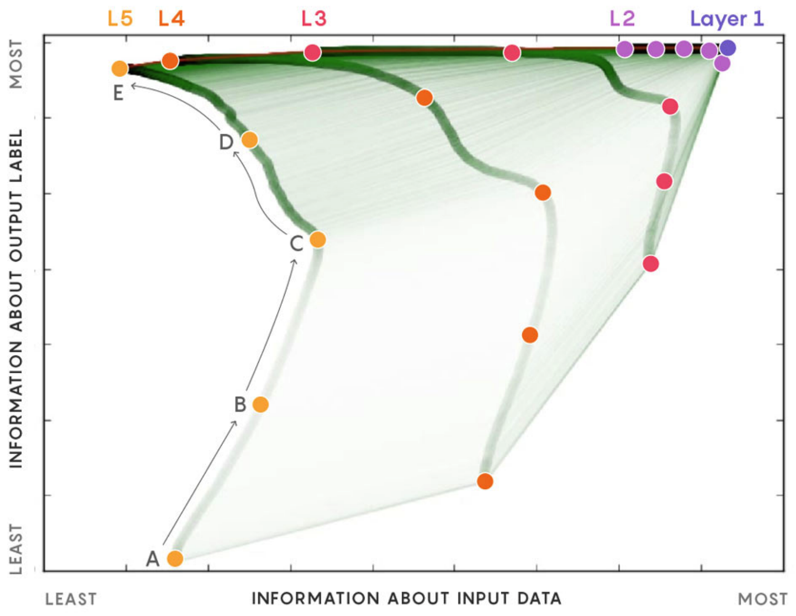

Figure 3.1: Visualization of the information-plane for a five layer neural network. Layer 1 is the first layer, L2 the second layer, and L5 the last layer before the prediction $\hat{Y}$. The green line connecting the colored points indicate that those points belong to the same epoch. Image credit: [63]

To quantify the amount of information, one uses the mutual information between two random variables X and Y with densities $p(x)$ and $p(y)$. The mutual information is defined as

$$\begin{aligned}
I(X;Y) &= \sum_{x\in X, y\in Y} p(x,y)\log\left(\frac{p(x,y)}{p(x)p(y)}\right) = \sum_{x\in X, y\in Y} p(x,y)\log\left(\frac{p(x|y)}{p(x)}\right) \\
&= -\sum_{x\in X}\log p(x)\sum_{y\in Y} p(x|y)p(y) + \sum_{x\in X, y\in Y} p(x|y)p(y)\log\big(p(x|y)\big) \\
&= -\sum_{x\in X} p(x)\log p(x) + \sum_{y\in Y} p(y)\left(\sum_{x\in X} p(x|y)\log p(x|y)\right) \\
&= H(X) - \sum_{y\in Y} p(y)H(X|Y=y) \\
&= H(X) - H(X|Y), \qquad (3.1)
\end{aligned}$$

where $H(X)$ and $H(X|Y)$ are the entropy and conditional entropy defined as

$$H(X) = -\sum_{x\in X} p(x)\log(p(x)), \tag{3.2}$$

$$H(X|Y=y) = -\left(\sum_{x\in X} p(x|y)\log p(x|y)\right). \tag{3.3}$$

The entropy is a measure of the uncertainty of a random variable. Note that if the value of X is completely determined by the value $Y = y$, the conditional entropy $H(X|Y = y)$ is zero, since there is no uncertainty left in X.

Figure 3.2 shows a fully-connected neural network for a two-class prediction task (e.g. labelling pictures with cats or dogs). Here, Y and X are the data we are given, with Y being either 1 or 0 depending on which class is associated with the features X. The arrows in the figure represent matrix multiplications (equation 1.2). For example, $\mathbf{h}_1$ is produced from X following a matrix multiplication and an element-wise nonlinear function, for instance, the function $f(x) = \max(0, x)$. In Figure 3.2, the representation Z^l is denoted as $\mathbf{h}_l$. There are no arrows between Y and X, since one considers the data X being generated from Y by an unknown mechanism, i.e., the data-generating process.[1] The role of the neural network, then, is to reverse this process and go from the features X to a prediction $\hat{Y}$ as close as possible to Y.

Consider the Markov chain $X \to Y \to Z$ satisfying the distribution $p(x, y, z) = p(x)p(y|x)p(z|y)$. The Data Processing Inequality (DPI) [65] states that the following relation holds on the mutual information between the variables:

$$I(X;Y) \geq I(X;Z), \tag{3.4}$$

since no function of Y can contain more information about X than Y itself. By applying the DPI consecutively on the layers of the neural network, one obtains the relations

$$I(X;Y) \geq I(Z_1;Y) \geq \cdots \geq I(Z^l;Y) \geq I(\hat{Y};Y) \tag{3.5}$$

and

$$H(X) \geq I(X;Z_1) \geq \cdots \geq I(X;Z^l) \geq I(X;\hat{Y}). \tag{3.6}$$

[1]This is best exemplified by the data X being images of real-world objects like cats, cars, dogs, ships, plants or any other object. The dataset X is considered as a random sample from an underlying distribution or set. This distribution could be the "Set of all natural images", natural indicating that they are photos of an actual cat, not a photo of an image of a cat on a computer screen. The label Y is which category the photo belongs to. Hence, one can consider each image x as a random variable, and when the layers of a neural network process an image x, they will generate representations z^i of that image. These representations will again be random variables, since they are functions of the random variable x.

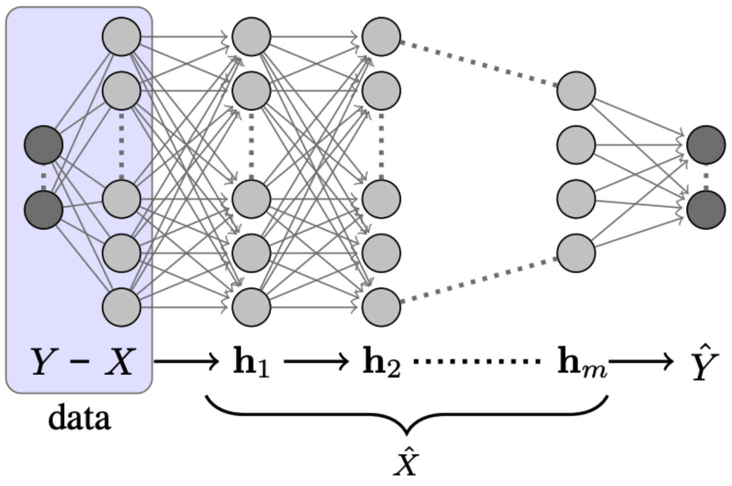

Figure 3.2: Visualization of the hidden layers $\mathbf{h}_i$, the data X and Y and the final representation $\hat{Y}$ of a fully-connected deep neural network. In the figure the hidden representations are referred to as $\mathbf{h}_i$ instead of Z^i. Image credit: [64].

Hence when interpreting information plane plots such as in Figure 3.1, one expects the later layers to be down and to the left compared to previous layers. As training progresses, however, the later layers would learn more about the output Y.

The Information Bottleneck method thus relies on computing the mutual information between each layer Z^l and the data X and Y during the course of training.

3.1.2 Gradient evolution

Another element of the information bottleneck method is to track the gradients of the layers Z^l during training. We briefly describe the idea here, even though it is not the focus of the thesis. It is however complementary to the information plane, and both methods have been implemented in the framework described in Chapter 4. Note that each layer has a weight matrix W^l, and it is the gradient of this layer weight ∇W^l that we store.

For this subsection we are going to denote layer numbers with subscripts, so the representation from layer number l is denoted as Z_l.

After having chosen a batch size B, one is left with $n_B = \lfloor N/B \rfloor$ number of batches in an epoch. Training over E epochs yields a total of $K = n_B \cdot E$ number of optimization steps. Hence for a neural network with L layers,

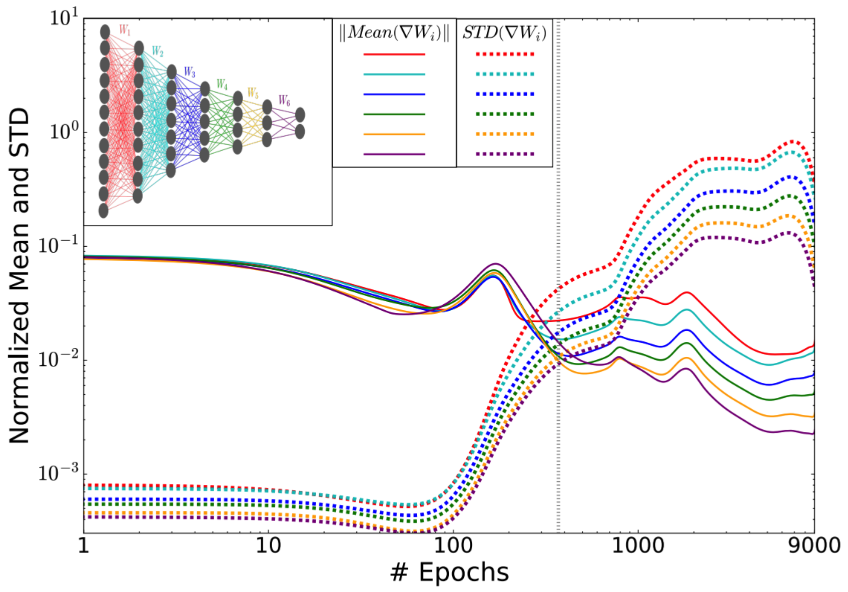

Figure 3.3: Visualization of the gradient SNR for each of the five layers in the network from [10]. There are five layers but six weight matrices, since the binary output $\hat{Y}$ is not considered a layer.

there is in total $L \cdot K$ different gradient matrices to store, one for each update in every layer. Let us consider the visualization for a single layer l and its gradient ∇W_l. Denote $\nabla W_l^{b,e}$ as batch number b in epoch e, where $1 \leq b \leq n_B$ and $1 \leq e \leq E$. The mean and standard deviation for each epoch is then defined as

$$\text{Mean}(\nabla W_l^e) = \frac{1}{n_B} \sum_{b=1}^{n_B} \nabla W_l^{b,e} \tag{3.7}$$

$$\text{STD}(\nabla W_l^e)^2 = \frac{1}{n_B} \sum_{b=1}^{n_B} \left(\nabla W_l^{b,e} - \text{Mean}(\nabla W_l^e)\right)^2. \tag{3.8}$$

The results from equations (3.7) and (3.8) are matrices. In order to obtain a single scalar number, one takes the L2 norm of both matrices and divide by the L2 norm of the layer weight at the end of each epoch:

$$\frac{\|\text{Mean}(\nabla W_l^e)\|_2}{\|W_l^{n_B,e}\|_2} \quad \text{and} \quad \frac{\|\text{STD}(\nabla W_l^e)\|_2}{\|W_l^{n_B,e}\|_2}. \tag{3.9}$$

By doing this procedure for every layer, one can visualize the gradient signal-to-noise ratio (SNR) across every epoch for the whole network. Figure 3.3 visualizes the gradient SNR-plane during the course of training from the example data and network from [10].

In Figure 3.3, the standard deviation of the gradients increases whilst the mean decreases as training progresses. In [10] this is interpreted as the network generalizing away unnecessary information. The early steps of the optimization process decrease the loss function and require a large mean to change the layer weights. When the network converges, the training signal becomes more noisy, which is interpreted as the network generalizing away unnecessary components of X (compressing the information). This claim is however disputed, most notably by [66], which state that the compression phase characteristic is due to the usage of tanh activation function, and not a general feature of neural networks.

The information bottleneck method is used to investigate how the training process of a neural network evolves, and can be used to evaluate the convergence of deep networks. Small changes in network architecture can lead to significant changes in convergence, and using the information bottleneck visualizations could provide useful into investigating the convergence of a neural network [67].

3.1.3 Optimal encoder and decoder

A key part of the information bottleneck method is understanding a neural network as an encoder-decoder pair. This will be briefly explained in this subsection.

An encoder is a function which encodes an input, in this case X, into a new representation Z. An encoder can be understood in an information-theoretic way as compressing data X into a representation Z which has fewer bits. Then a decoder can be used to generate X or Y from Z. For example, the data X could be a movie. One hour of movie streaming at a resolution of 720×480 uses about 1GB, according to Netflix [68]. If we assume 30 images per second, this amounts to $(720 \times 480 \times 3 \times 30 \times 3600)$ bytes $\approx$ 112GB. Hence, Netflix is able to encode 112GB of movie information into a representation Z of 1GB. The representation Z is thus transmitted instead of the data X. Optimality of Z in this case requires that Z has the shortest bit-length for which the decoder is able to reproduce X.

In the classification setting, the decoder is trained to generate Y from Z. Figure 3.4 visualizes a neural network as multiple encoder-decoder pairs, where the representation Z^l is denoted by T^l.

In the information bottleneck view, an optimal representation Z^l is

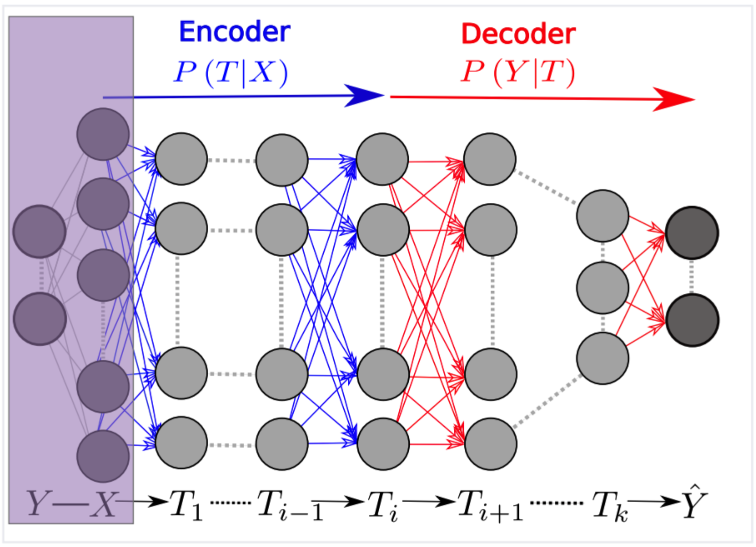

Figure 3.4: Visualization of a neural network as an encoder-decoder pair. The representation Z^l is denoted as T_i in the figure. Image credit: [64].

considered a solution to the following optimization problem

$$Z^l = \text{argmax}_{Z \in \mathcal{Z}} I(Y, Z) \quad \text{s.t.} \quad I(X, Z) \leq R, \tag{3.10}$$

where R is a compression threshold and $\mathcal{Z}$ is the domain of the representation Z^l. The domain $\mathcal{Z}$ is defined by the hidden layer sizes and the activation function used. If the ReLU activation function is used with a hidden layer size of 2, the domain is $\mathcal{Z} = \mathbb{R}^2_+$, the set of non-negative real numbers in the plane. Successfully training a neural network should thus result in a final representation Z^L which has a minimal $I(X, Z^L)$ but a maximal $I(Y, Z^L)$, in essence being a compressed representation of X which is able to provide sufficient information to classify Y correctly.

Early methods for computing the mutual information did not consider a layer-wise optimization process as suggested by (3.10), but used (3.10) as a theoretical tool to understand the representations Z_i learned by a neural network. However, recent work [69, 70] consider a layer-wise error term based on the information bottleneck optimality (3.10). For estimating mutual information, the probability density approach of Kolchinsky et. al [71] described in Section 3.2 relies heavily upon the encoder-decoder interpretation of neural networks in order to derive their estimation method.

Equation (3.10) does not characterize what a representation Z^l *is*, but rather what it should be. So the IB method ranks a representation Z^l based on how well it is able to fulfill equation (3.10). However, the representation Z^l found by optimizing a neural network through gradient descent does not actually fulfill equation (3.10).

3.2 Mutual Information Estimation

The original article [11] uses a binning procedure to estimate $p(z)$ for an arbitrary activation z. This is possible since they use tanh as activation function which has its output in the interval $(-1,1)$. By selecting a certain number of bins on the interval $(-1,1)$ one can count the number of elements in each bin, normalize, and then one obtains an approximation for $p(z)$.

Due to the output of the ReLU function being in the interval $[0,\infty)$, approaches for computing the mutual information based on binning the output of a ReLU function perform poorly.

In an effort to alleviate these issues, the Nonlinear Information Bottleneck (NIB) authors [71] rewrite the mutual information term to only include the hidden layer representations Z, and are able to provide both upper and lower bounds for both $I(X,Z)$ and $I(Z,Y)$[71, 66].

3.2.1 Estimator Based on Pairwise-Distances

The main method we use is taken from [13] and [71]. The method assumes that each data points representation is the mean of a normal distribution. In turn, this creates a mixture distribution due to there being multiple data points in the data set. The estimation of mutual information is thus reduced to estimating the mutual information for this particular kind of mixture distribution.

The authors of [13] and [71] mainly consider the case in which there is only one bottleneck variable Z, but the same argument may be applied on any hidden representation $Z^1, Z^2, \dots Z^L$. The outset of their method is an estimation rule of the form

$$\hat{I}(X,Z) = -\sum_i c_i \ln \sum_j c_j e^{-D(p_i \| p_j)}, \tag{3.11}$$

where c_i is the weight of each mixture component, and $D(p_i \| p_j)$ is a *premetric*, meaning it is nonnegative and $D(p_i \| p_j) = 0$ if $p_i = p_j$. D is not assumed to be symmetric, nor to obey the triangle inequality. Note that even if $p_i \neq p_j$, it may still be the case that $D(p_i \| p_j) = 0$. The functions $D(p, q)$ are used to measure the distance between probability distributions p and q. An example of such a function is the *Kullback-Leibler* (KL) - *divergence* [65], defined as

$$\mathrm{KL}(p \| q) = \int p(x) \ln \frac{p(x)}{q(x)} \mathrm{d}x. \tag{3.12}$$

Let us compute the KL divergence between two normal distributions p and q, where $p \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $q \sim \mathcal{N}(\mu_2, \sigma_2^2)$

$$\begin{aligned}
KL(p \| q) &= \int p(x) \ln \frac{p(x)}{q(x)} \mathrm{d}x \\
&= \int p(x) \left(-\frac{(x-\mu_1)^2}{2\sigma_1^2} - \frac{1}{2} \ln(2\pi\sigma_1^2) \right) \mathrm{d}x \\
&- \int p(x) \left(-\frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \ln(2\pi\sigma_2^2) \right) \\
&= \ln \frac{\sigma_2}{\sigma_1} + \int p(x) \left(\frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2} \right) \\
&= \ln \frac{\sigma_2}{\sigma_1} - \frac{1}{2} + \frac{1}{2\sigma_2^2} \int p(x)(x^2 - \mu_2 x + \mu_2^2) \mathrm{d}x \\
&= \ln \frac{\sigma_2}{\sigma_1} - \frac{1}{2} + \frac{1}{2\sigma_2^2} \left(\sigma_1^2 + \mu_1^2 - 2\mu_1\mu_2 + \mu_2^2 \right) \\
&= \ln \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}.
\end{aligned}$$

For the case when $\sigma_1 = \sigma_2 = \sigma$, this reduces to $KL(p \| q) = \frac{(\mu_1 - \mu_2)^2}{2\sigma^2}$. Using the KL-divergence as a distance function then yields the upper bound

$$I(X, Z) \leq \hat{I}(X, Z) = -\frac{1}{N} \sum_i \log \frac{1}{N} \sum_j \exp\left(-\frac{\| Z_i - Z_j \|_2^2}{2\sigma^2} \right), \tag{3.13}$$

where the subscripts on Z_i and Z_j refer to row number i and j of the hidden representation Z. Z has the dimension $N \times H$, where H is the hidden layer dimension and N is the number of examples in the dataset. The mutual information term $I(Y, Z)$ can be computed as [66]

$$\begin{aligned}
I(Y,Z) &= H(Z) - H(Z|Y) && (3.14)\\
&\leq -\frac{1}{N}\sum_i \log\frac{1}{N}\sum_j \exp\left(-\frac{\|Z_i - Z_j\|_2^2}{2\sigma^2}\right) && (3.15)\\
&-\sum_{c=1}^{C} p_c\left[-\frac{1}{N_c}\sum_{\{i|Y_i=c\}}\log\frac{1}{N_c}\sum_{\{j|Y_j=c\}}\exp\left(-\frac{\|Z_i - Z_j\|_2^2}{2\sigma^2}\right)\right]. && (3.16)
\end{aligned}$$

Where N_c is the number of samples with the label $Y = c$, $\sum_{c=1}^{C} N_c = N$, and $p_c = \frac{N_c}{N}$.

Another choice of distance function is the *Bhattacharyya distance* (BD) defined as [13]

$$\mathrm{BD}(p\|q) = -\ln\int\sqrt{p(x)q(x)}\,\mathrm{d}x. \tag{3.17}$$

For two normal distributions $p \sim \mathcal{N}(\mu_1,\sigma)$ and $q \sim \mathcal{N}(\mu_2,\sigma)$, we compute

$$\begin{aligned}
\mathrm{BD}(p\|q) &= -\ln\int_x \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{4\sigma^2}\left((x-\mu_1)^2+(x-\mu_2)^2\right)\right)\mathrm{d}x\\
&= -\ln\int_x \frac{1}{\sigma\sqrt{2\pi}}\left(\exp\left(-\frac{(\mu_1-\mu_2)^2}{8\sigma^2}\right)\right)\cdot\exp\left(-\frac{(x-\mu^*)^2}{2\sigma^2}\right)\mathrm{d}x\\
&= -\ln\exp-\frac{(\mu_1-\mu_2)^2}{8\sigma^2} = \frac{(\mu_1-\mu_2)^2}{8\sigma^2}
\end{aligned}$$

where we define $\mu^* = \frac{\mu_1+\mu_2}{2}$, and have used the relations

$$(x-\mu_1)^2+(x-\mu_2)^2 = 2\cdot(x-\mu^*)^2+\frac{1}{2}(\mu_1-\mu_2)^2$$

and

$$\int_x \exp\left(-\frac{(x-\mu^*)^2}{2\sigma^2}\right)\mathrm{d}x = \sigma\sqrt{2\pi}.$$

The Bhattacharyya distance leads to lower bounds [13, 71], which are as follows

$$I(X,Z) \geq -\frac{1}{N}\sum_i \log \frac{1}{N}\sum_j \exp\left(-\frac{\|Z_i - Z_j\|_2^2}{8\sigma^2}\right) \tag{3.18}$$

$$I(Y,Z) \geq -\frac{1}{N}\sum_i \log \frac{1}{N}\sum_j \exp\left(-\frac{\|Z_i - Z_j\|_2^2}{8\sigma^2}\right) \tag{3.19}$$

$$-\sum_{c=1}^{C} p_c \left[-\frac{1}{N_c}\sum_{\{i|Y_i=c\}} \log \frac{1}{N_c}\sum_{\{j|Y_j=c\}} \exp\left(-\frac{\|Z_i - Z_j\|_2^2}{8\sigma^2}\right)\right]. \tag{3.20}$$

In this thesis we will mainly be using the Bhattacharyya lower bound for estimating mutual information.

3.2.2 Dual Formulation Approach

Equations (3.13) and (3.14) are derived by making assumptions on the form of distribution of the hidden variables Z. The approach of MINE [15] is different. Their approach does not yield a closed-form solution for their estimates of the mutual information terms $I(X,Z)$ and $I(Z,Y)$. Rather their estimates come from an optimization process. By using principles of duality [51] and several information-theoretic inequalities they obtain a lower bound for the mutual information I. For more details we refer to the MINE paper and related works [15, 72, 73, 51]. We note that the bounds found by MINE are lower bounds. Denote a neural network with parameters $\theta \in \Theta$ as F_θ, with $F_\theta : \mathcal{D} \times \mathcal{Z} \rightarrow \mathbb{R}$, with $\mathcal{D}$ and $\mathcal{Z}$ the domains of D and Z, respectively. The space $\mathcal{D}$ can either be the space for the input X, $\mathcal{X}$ or the space of the label Y, $\mathcal{Y}$. The space $\mathcal{Z}$ denotes the space for the representations Z. MINE then states that the mutual information $I(D,Z)$ can be approximated by the following optimization problem,

$$I_\Theta(D,Z) = \sup_{\theta \in \Theta} \mathbb{E}_{\mathbb{P}_{DZ}}[F_\theta(D,Z)] - \log(\mathbb{E}_{\mathbb{P}_D \otimes \mathbb{P}_Z}[e^{F_\theta(D,Z)}]), \tag{3.21}$$

with

$$I(D,Z) \geq I_\theta(D,Z),$$

where $\theta \in \Theta$ is the parameter space for the neural network F_θ. The expectation term $\mathbb{E}_{\mathbb{P}_{DZ}}$ denotes that samples (d,z) are sampled from the joint distribution $\mathbb{P}_{DZ}$, whilst $\mathbb{E}_{\mathbb{P}_D \otimes \mathbb{P}_Z}$ denotes samples (d,z), where d and z come from the marginal distributions $\mathbb{P}_D$ and $\mathbb{P}_Z$ respectively.

One note on sampling from $\mathbb{P}_X \otimes \mathbb{P}_{Z_i}$, where we have chosen $D = X$ and layer i for the representation Z_i. The data X has N datapoints, $(X_1, X_2, \dots X_N)$. The graph neural network and fully-connected neural network trained on the original problem will produce representations $(Z_{G,2}, Z_{G,2}, \dots Z_{G,N})$ for the graph network and $(Z_{FC,1}, Z_{FC,2} \dots Z_{FC,N})$ for the fully-connected network. When sampling from the joint distribution $\mathbb{P}_{XZ_i}$ with $Z_i = Z_{G,i}$ or $Z_i = Z_{FC,i}$, one samples batches of length B and obtains a batch $\big((X_{1b}, X_{2b}, \dots X_{Bb}), (Z_{1b}, Z_{2b}, \dots Z_{Bb})\big)$ where we have used indices ib for element number i in the batch since the i'th element of the batch is not necessarily the i'th element in the data X. When sampling from the joint distribution, the representation Z_{1b} is the representation of the input data X_{1b}. However, when sampling from the product of the marginal distributions one shuffles the batch of representations $\{Z_{ib}\}_{i=1}^{B}$, obtaining a batch $\big((X_{1b}, X_{2b}, \dots X_{Bb}),\ \alpha(Z_{1b}, Z_{2b}, \dots Z_{Bb})\big)$ where $\alpha(a, b, c, \dots)$ is a function yielding a random permutation of its arguments. This permutation operation causes misaligned representations in the X batch and the Z_i batch, which together yields samples from the distribution $\mathbb{P}_X \otimes \mathbb{P}_{Z_i}$.

The MINE approach thus relies on a completely separate neural network F_θ and training process in order to estimate the mutual information of the problem-specific neural network. The MINE neural network F_θ is usually chosen to be fully connected. Let Z_G and Z_F be representations obtained by the graph and fully-connected network trained on the original problem, for instance a classification problem. Then the same neural architecture F_θ can be used to approximate the mutual information terms $I(X, Z_G)$, $I(Z_G, Y)$, $I(X, Z_F)$, $I(Z_F, Y)$. However, each term $I(X, Z_G)$ through $I(Z_F, Y)$ need their own training process, requiring a total of 4 neural networks with the architecture of F_θ to be trained. The training process for training a network F_θ on data (D, Z) is straightforward. Given the data (D, Z) one computes the output $F_\theta(D, Z)$ in equation (3.21), and this objective is optimized through gradient ascent. The output of the network is the mutual information, $F_\theta(D, Z) = I_\theta(D, Z)$, so we are directly optimizing the mutual information lower bound. The pseudo code in Algorithm 1 shows an overview for estimating the mutual information with MINE. The criterion "not converged" can be chosen in several ways, but the easiest is to choose a fixed (large) number of epochs E.

Algorithm 1: MINE procedure

```
Result: MINE estimate for $I_\theta(D, Z)$
choose a fully-connected architecture for $F_\theta$
while not converged do
    sample batch (d, z) from data (D, Z)
    objective = $F_\theta(d,z) - \log(\exp F_\theta(d,\alpha(z)))$
    optimize parameters $\theta$ through gradient ascent on the objective
end
return $F_\theta(D, Z)$
```

When training the classification network, each layer i will generate a representation Z_i^e of the input data X at epoch e. We let E be the number of epochs, N the number of samples in the dataset and B the batch size. Even though the network will be updated $n_B = \lfloor N/B \rfloor$ times during an epoch E, the representation for an epoch e, Z_i^e is just the concatenated representations from batches in that epoch, $z_i^{b,e}$ where the dimension of z_i is $B \times H_i$ where H_i is the hidden size of layer i. The dimension of Z_i^e is $N \times H_i$. Hence, when training is complete, we have a set Z_i^e for $1 \leq i \leq L$ and $1 \leq e \leq E$, where L is the number of layers in our network. We then use Algorithm 1 to estimate $I_\theta(X, Z_i^e)$ and $I_\theta(Y, Z_i^e)$ for every epoch e and each layer i.

Chapter 4

The Developed Framework

4.1 The Developed Framework

For this project, a framework for extracting gradients and activations of a generic neural network has been developed in PyTorch [36] and can be found on GitHub (github.com/mariusmcl/MasterProject). The framework mainly revolves around PyTorch Module objects. A PyTorch Module object contains parameters and the forward pass for a certain block of the neural network. An example module object of a fully-connected neural network is shown in the code listing 4.1. Lines 7 through 13 define the submodules we want to use in our fully-connected neural network. We have ReLU activation functions, and fully-connected linear layers. The forward method starting at line 14 specifies how to compute a prediction $\hat{y}$ given the input x. In order to store the activations in the forward pass it is required that the activation functions used in the forward pass are specified as attributes in the object initialization. The nn.Module superclass contains methods for accessing these object attributes, for example the submodules ”linear1” or ”relu1” in listing 4.1.

Once our neural network is defined, we attach PyTorch hooks on the linear and ReLU submodules from listing 4.1. There are two kinds of PyTorch hooks: forward and backward hooks. The forward hooks stores the forward pass of the model, which are the intermediate representations Z^l produced by a neural network. The backward hooks store the gradients ∇W^l of the parameters W^l in the network. More details on forward and backward hooks may be found at [74]. The main challenge when working with PyTorch hooks is to identify which part of the PyTorch hook stores the necessary

```python
import torch.nn as nn
import torch

class LinearNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(LinearNetwork, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.relu1 = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.relu2 = nn.ReLU()
        self.linear3 = nn.Linear(hidden_size, hidden_size)
        self.relu3 = nn.ReLU()
        self.linear4 = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        x = self.relu1(self.linear1(x))
        x = self.relu2(self.linear2(x))
        x = self.relu3(self.linear3(x))
        x = self.linear4(x)
        return x
```

Listing 4.1: A Fully-Connected Network In PyTorch As A Module Object

information, since this is organized differently for different types of neural network layers. Code listing 4.2 shows an example training procedure when using a ”tracker” to store the forward pass of the network. The ”register_new_epoch” method tells the tracker which modules to store the activations for in this epoch. Then, the ”save” function saves the desired activations for later use. When training is completed, the tracker object has stored all the activations $\left(\{Z_e^1\}_{e=1}^E, \{Z_e^2\}_{e=1}^E, \dots \{Z_e^L\}_{e=1}^E\right)$ for every layer l, $1 \leq l \leq L$ and every epoch e during training. L is the number of layers and E is the number of training epochs.

The novelty of the developed framework is that it is agnostic to what kind of model architecture is specified, as long as every submodule to be tracked is defined explicitly in the initialization method of the PyTorch module. Instead of a nn.Linear layer, one could have a nn.Conv layer, or a nn.RNN layer. The existing codebases from [10, 66] are well suited for their specific neural architectures and setup, but they lack the flexibility of being easily applied on different neural architectures. We will be using our framework to track the representations of a graph neural network and a recurrent neural network, which, as far as the author knows, has not been done before. Hence

```python
for epoch in range(num_epochs):
    tracker.register_new_epoch(list(tracker.forward_hooks.keys()))
    # forward_hooks.keys() contains 'relu1', 'linear1', 'relu2', ...
    for x, y in train_loader:
        optimizer.zero_grad()
        out = linear_classifier(x)
        loss = crossentropy_loss(out, y)
        loss.backward()
        optimizer.step()
        tracker.save()
```

Listing 4.2: Example Training Loop Using The Data Tracker

we build on the gradient and representation tracking capabilities from [10, 66] and extend it by making it more suited for different neural architectures.

Another essential part of the framework is the mutual information estimation method. We use the pairwise-distance method from Kolchinsky et. al [13, 71], since this method was the most consistent of all the methods we tried. The Rényi [14] estimation method was attempted, but it had prohibitive computational costs associated with it. The MINE method had numerical difficulties estimating the mutual information when the data X had many features or was sparse. MINE also has other issues, due to the requirement of a neural network and an additional training loop in the estimation process. This adds extra hyperparameters which results in another unnecessary layer of complexity in the estimation process.

The only hyperparameter in the Kolchinsky method is the noise variance σ^2 of the assumed normal distribution. Different choices of σ^2 produce qualitatively different results, which will be shown in Section 5.1 and 5.2.

4.2 Validation of Framework

We begin by validating our framework on the same neural architecture and dataset used in [10]. The neural architecture is a fully-connected network with hidden layer sizes of 10, 7, 5, 4, and 3. There are 12 input features which are to be classified into one of two categories. Figure 4.1 shows the information-plane visualization of a neural network training procedure, by using the pairwise-distance estimators from Kolchinsky et al [71, 13]. Figure 4.1 is obtained using the BD distance (left) and KL-divergence (right).

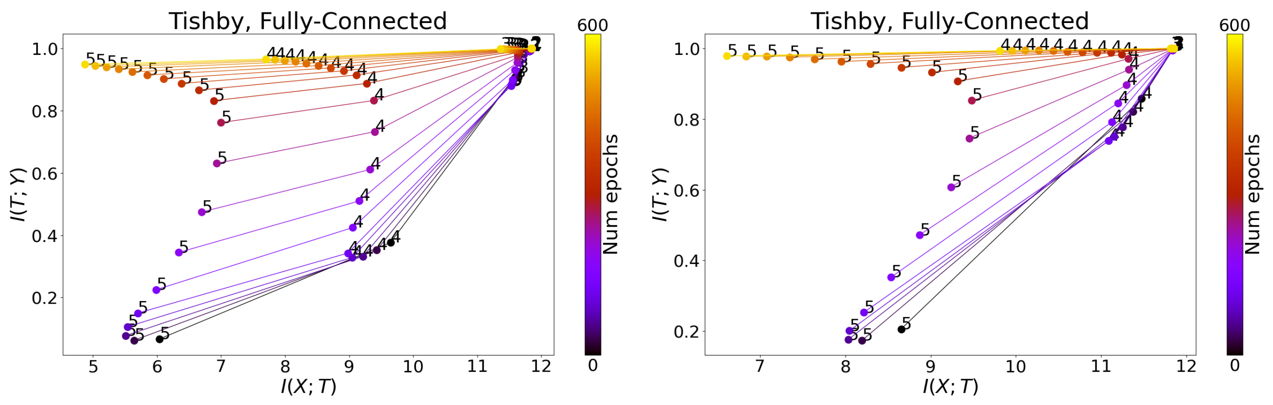

Figure 4.1: The information-plane visualisation of the neural network layers. The leftmost figure is attained using the BD distance, which yields a lower bound on the mutual information. The rightmost figure uses the KL divergence, which is an upper bound. The neural architecture is a fully-connected network with hidden layer sizes of 10-7-5-4-3. The annotated numbers denote which layer the point corresponds to, with number 1 denoting the layer with a hidden size of 10. During training, the last layers are able to attain a high performance for predicting the label Y while having a low mutual information with X, meaning that the final representations are able to generalize well.

The BD distance is a lower bound, whilst the KL divergence yields an upper bound. The gap between the lower and upper bound seems to decrease during training. Each color corresponds to a particular epoch. The annotated numbers denote the layer, with layer 1 having 10 activations, layer 2 having 7 activations, and similarly for the remaining layers. The x-axis is the mutual information between layer Z^l and the data X, and the y-axis is the mutual information between layer Z^l and Y, where Z^l is a layer activation and the layer number l is annotated in the figure.

In Figure 4.1 we see that the mutual information that the final layers contain about Y increases during training, while managing to maintain a low mutual information with the features X. This is a sign that the network is able to generalize away unnecessary information in X, whilst retaining important information about Y. The original information plane from [10] is shown in Figure 4.2. They have three plots, where they have used 5% (left), 45% (middle) and 85% (right) of the training data. There are some differences for the estimated value of $I(X,Z)$ for the later layers compared to our results, but this can be attributed to different learning rates and frameworks. We also note that the results from [67] shown in Figure 4.3 are neither able to precisely replicate the original results from [10]. Figure 4.3 shows the information plane on the original data from Tishby et.al [10] (left) and the information plane for a fully-connected network with hidden layer sizes of $1024-20-20-20$ (right) trained on the MNIST [3] dataset. Our results using the same fully-connected architecture on the MNIST dataset is shown in Figure 4.4.

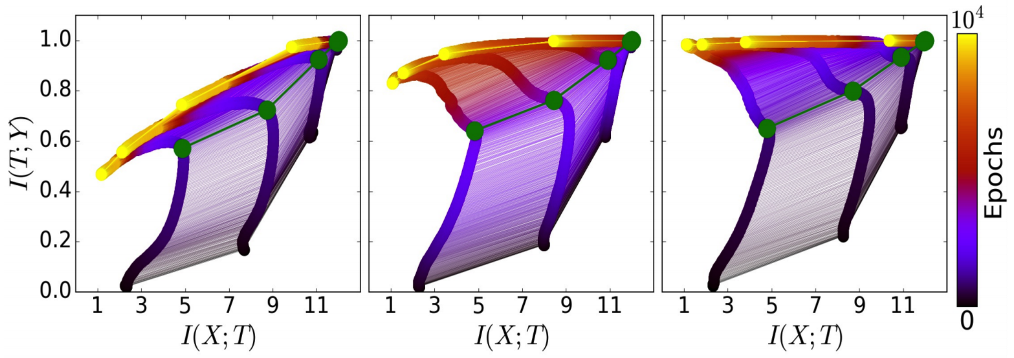

Figure 4.2: The information-plane visualisation of the neural network layers from the original paper [10]. The neural architecture is a fully-connected network with hidden layer sizes of 10-7-5-4-3. The original authors have plotted the information plane using different proportions of avaiable data. On the left - 5% of the data, middle -45% and the rightmost was trained on 85% of the data.

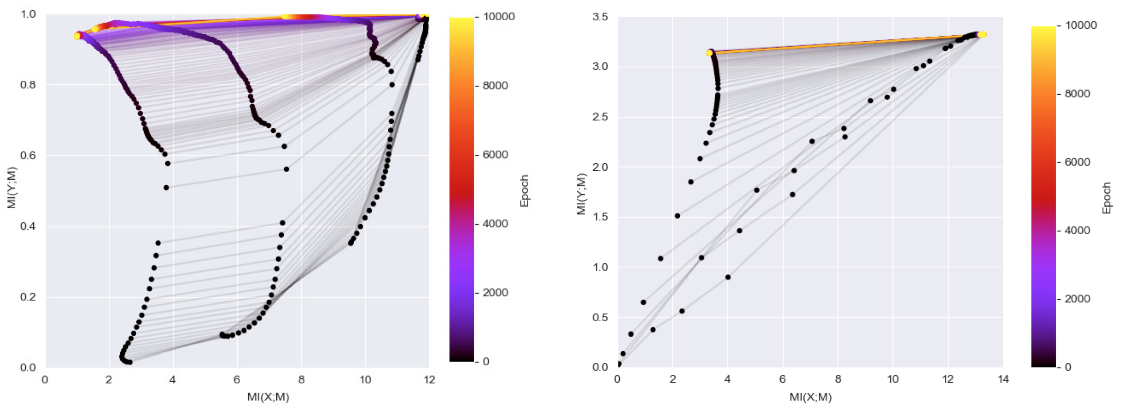

Figure 4.3: The information-plane visualisation from [67] with fully-connected networks on the Tishby dataset (left) and the MNIST dataset (right). The hidden layer sizes are $10-7-5-4-3$ and $1024-20-20-20$, respectively.

Figure 4.4 shows our framework applied on the MNIST dataset [3], using the BD distance. The MNIST dataset contains 60000 images of pictures with the numbers 0 through 9 in them. The images are 28×28 greyscale pictures. Hence this is a classification dataset, with 10 classes. We use the same fully-connected neural network architecture as [67]. Our results differ from theirs in that we obtain less compression in the final layers, as shown in Figure 3 in [67]. It is unknown what training and test accuracy they attain with this network, and they have trained for a total of 9000 epochs. In Figure 4.4 we have trained for 600 epochs, obtaining a validation accuracy of 0.85 and training accuracy of 0.97. The difference is probably related to [67]

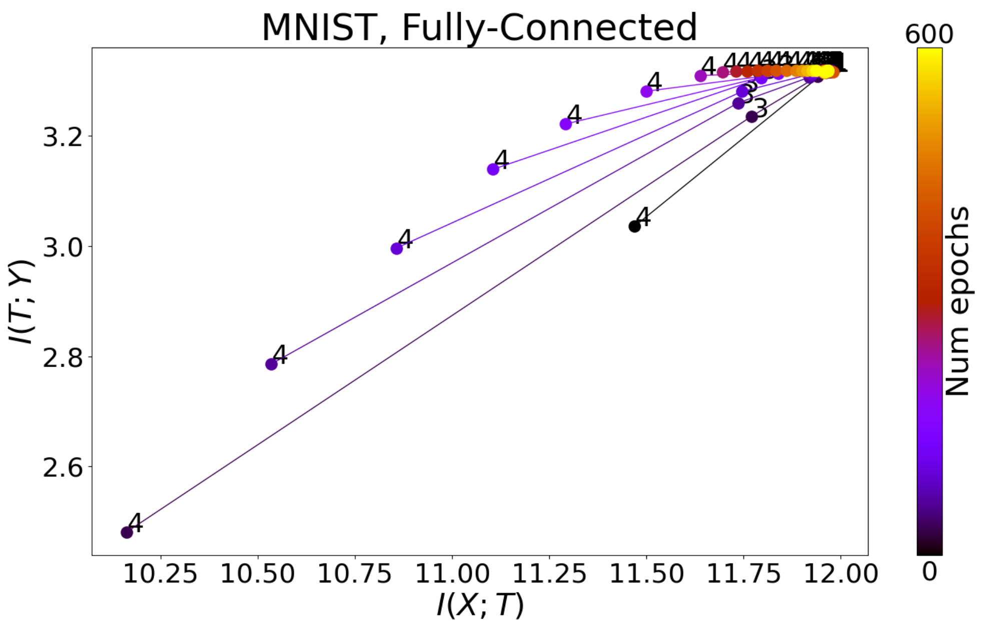

Figure 4.4: The information-plane visualisation of the neural network layers for the FCN trained on the MNIST dataset. The hidden layer sizes are 1024 – 20 – 20 – 20, as in [67]. The annotated numbers denote which layer the point corresponds to, where number four being the final layer with a hidden size of 20.

using a lower learning rate and training for more epochs but the details are not specified in the paper. In Figure 4.4 we have used a learning rate of 10^{-5}, and learning rates below that were found to severely hinder training. On the contrary, using a default 10^{-3} [56] learning rate yields equivalent results in fewer training epoch, which makes it hard to justify a learning rate below 10^{-5}.

Another part of the Information Bottleneck Method is to track the gradient signal-to-noise (SNR) ratios during training. For this, we track the gradients of each batch, and compute the mean and standard deviation across the batches contained in the same epoch. We then normalize by the $L2$-norm of the layer weight at the end of that epoch. Figure 4.5 shows our framework tracking the means and standard deviations of the gradients during the course of training. The most notable way in which our results differ from the results in the original paper [10] shown in Figure 3.3 is that we obtain standard deviations and means of similar magnitude in our framework, whereas they are offset from each other in Figure 3.3. Moreover, the result in Figure 3.3 is smoother since they have taken a mean over 50 runs.

The differences could be attributed to various causes, for instance the learning rate, optimizer and initialization of the neural networks. The

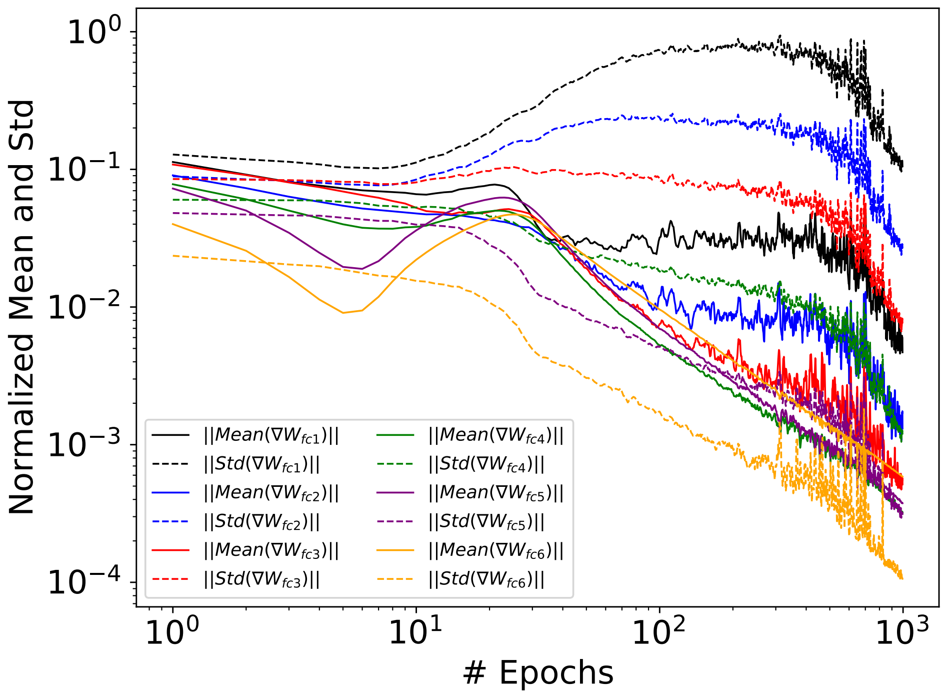

Figure 4.5: Normalised norms of the mean and standard deviations of the gradients during the course of training. The data has 12 input features and the task is classification into 2 categories with a fully connected neural network with layer sizes 10, 7, 5, 4, and 3. To be compared with the plot in Figure 3.3. The difference is attributed to different learning rate, optimizer and initialization of the neural networks.

normalization method could also differ. However, the most important point for being able to use the method is the clear transition point which is independent of scaling. In Figure 4.5 it is seen as the ’bump’ in the means, which occurs after about 40 epochs in both plots.

In Figures 4.6 and 4.7, we show the results from training a graph neural network and a fully-connected network on the Cora dataset. The citations of each paper are included as edges in the graph. The fully-connected and graph neural network differ only in that the graph-neural network uses the edge information contained in the graph. Both the fully-connected and graph neural network have the same number of neurons in every layer. In Figure 4.6, both networks are trained on 140 examples and validated on 1000 examples, testing their generalization capability. In Figure 4.7, the networks are trained on 1000 examples and validated on 140 examples.

The graph neural network significantly outperforms the fully-connected network in terms of accuracy in both training regimes. Even the graph neural model trained with fewer data points outperforms the

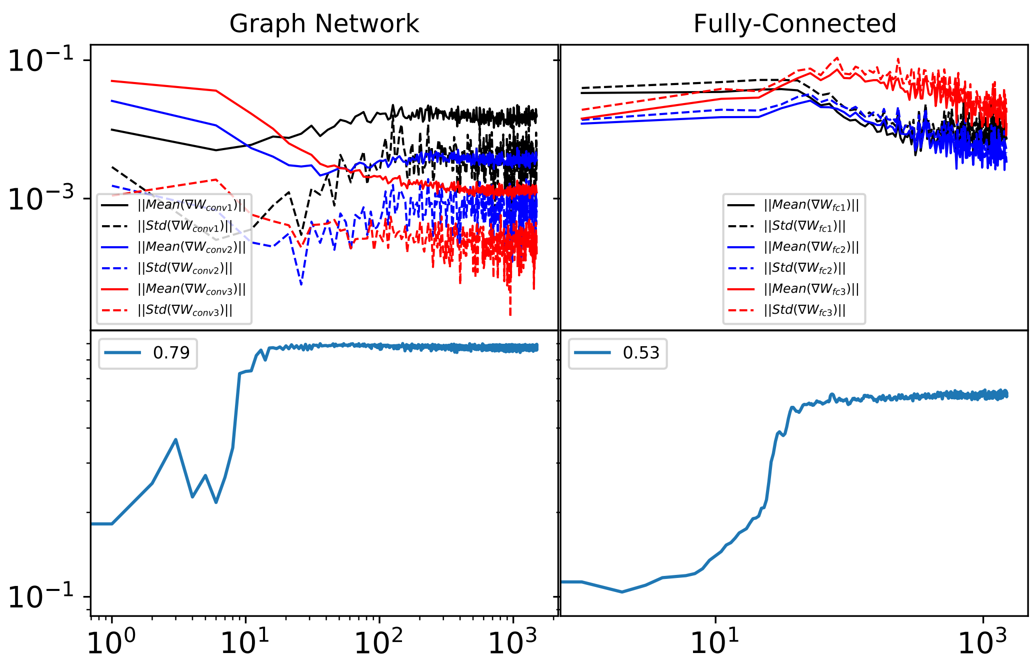

Figure 4.6: The normalised gradient mean and standard deviation signal-to-noise ratios (upper panels) and the validation accuracies during training on 140 examples and validated on 1000 examples from the Cora data set. The labels in the lower panels indicate the final accuracy. Given the small training sample, the final accuracy of the fully connected network is rather poor. The left column shows the results for the graph network and the right for the fully connected. We observe a transition in the mean and standard deviations at approximately the same location as the validation accuracy stabilizes.

fully-connected model trained with more data points. The gradient SNR-plots, which are the top rows of Figures 4.6 and 4.7, indicate a clear transition when the validation accuracy stabilizes. The validation accuracy is a proxy measure for test accuracy. The 53% accuracy of the fully-connected network is decent since there are seven different classes, so a random classifier would yield about 16% accuracy. From this it seems possible that the gradient SNR-plot can be used as a measure of test-time performance metric when a test-time performance metric is unavailable or difficult to define. For instance, in reinforcement learning [75], the notion of a training, validation and test set is not widely used. Instead the goal is to obtain as high performance as possible on a single problem, for instance playing Pong [76] or the board game of Go [77]. Hence one could use the gradient SNR plots to investigate whether a neural network attains such a gradient transition phase during training. Overfitting of neural networks in reinforcement learning has also been studied in [78].

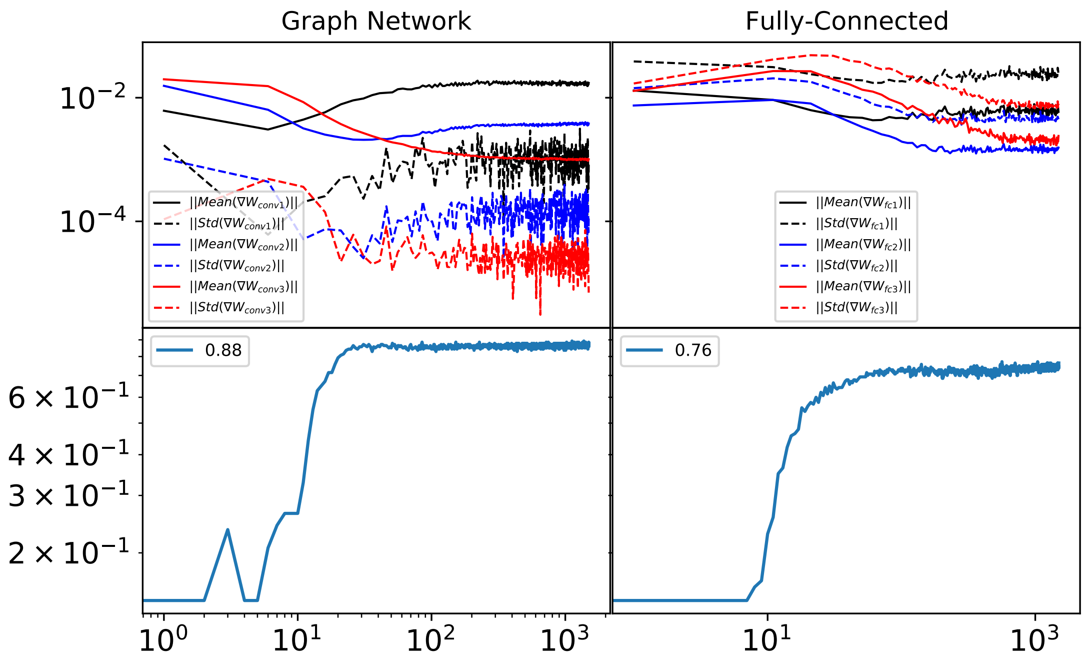

Figure 4.7: The normalised gradient mean and standard deviation signal-to-noise ratios (upper panels) and the validation accuracies during training on 1000 examples and validated on 140 examples from the Cora data set. The labels in the lower panels indicate the final accuracy. The large training sample allows for the models to generalise over a large amount of data, but due to the smaller validation sample, the generalisation abilities are not really stress tested. The left column shows the results for the graph network and the right for the fully connected. We observe a transition in the mean and standard deviations at approximately the same location as the validation accuracy stabilizes.

Chapter 5

Results and Discussion

5.1 Information Plane And Inductive Bias

In this section we are going to use the information plane from Section 3.1.1 in order to better understand why some models are better at modeling certain kinds of data than others. Our first comparison is going to be between a fully-connected and graph neural network on the Cora citation dataset from Section 1.4.1.

5.1.1 Fully-connected and Graph Neural Networks The Cora Citation Dataset

In this section we are going to compare a graph-convolutional network with the GCN architecture [48] with a fully-connected network. Both networks have three hidden layers, each with a hidden size of 200. We used a learning rate of 0.001, and used the whole dataset on each update. The reason for this is that it is difficult to split a graph dataset into batches, since the dataset contains edges between nodes in the dataset. Figure 5.1 shows the training accuracy in blue and validation accuracy in red for both the FCN (left) and GCN (right) models. Unsurprisingly, the GCN is able to achieve a better validation accuracy than the FCN model. We also see that the FCN model is overfitting the data more severely than its GCN counterpart.

We have plotted the information plane for both models using two noise levels, both $\sigma^2 = 0.1$ and $\sigma^2 = 0.001$. Figure 5.2 shows the information planes for the FCN (left) and GCN (right) models when choosing $\sigma^2 = 0.001$. Figure 5.2 is one of the more interesting figures of the thesis. We noticed that a flip occurred in the information plane plot for the GCN model, at about

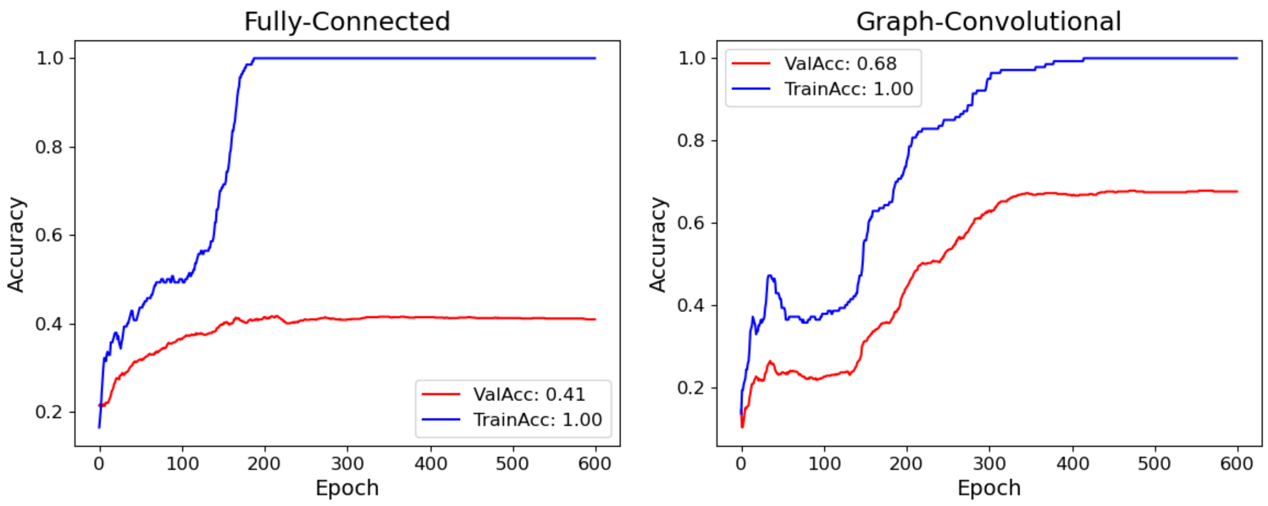

Figure 5.1: The training and validation accuracies for both the fully-connected and graph-convolutional models on the cora dataset. Both the FCN and the GCN had three hidden layers with 200 neurons in each.

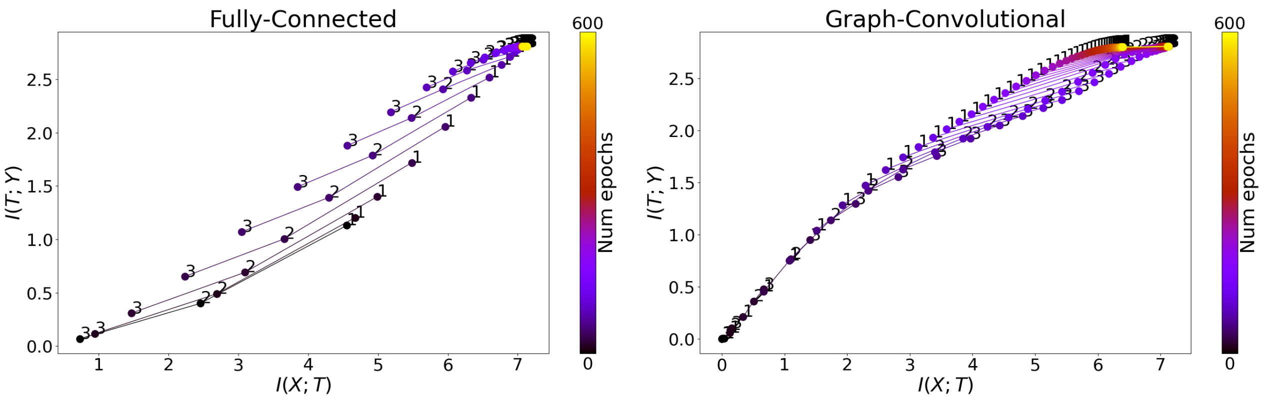

Figure 5.2: The information plane of the FCN and GCN models on the Cora dataset, with noise parameter $\sigma^2 = 0.001$.

the same epoch as the severe overfitting kick in.

The FCN information plane with $\sigma^2 = 0.001$ is behaving as expected and fulfilling the Data-Processing Inequality (DPI, Eqn. 3.4).

Our hypothesis was that one could identify certain behaviors of the model by analyzing the information plane for different neural network architectures. For instance, we would expect that since the GCN model is better at generalizing to new data, its later layers would compress away more information in X than the earlier layers. Unfortunately, this is not visible either when estimating with $\sigma^2 = 0.001$ or with $\sigma^2 = 0.1$ as shown in Figure 5.3.

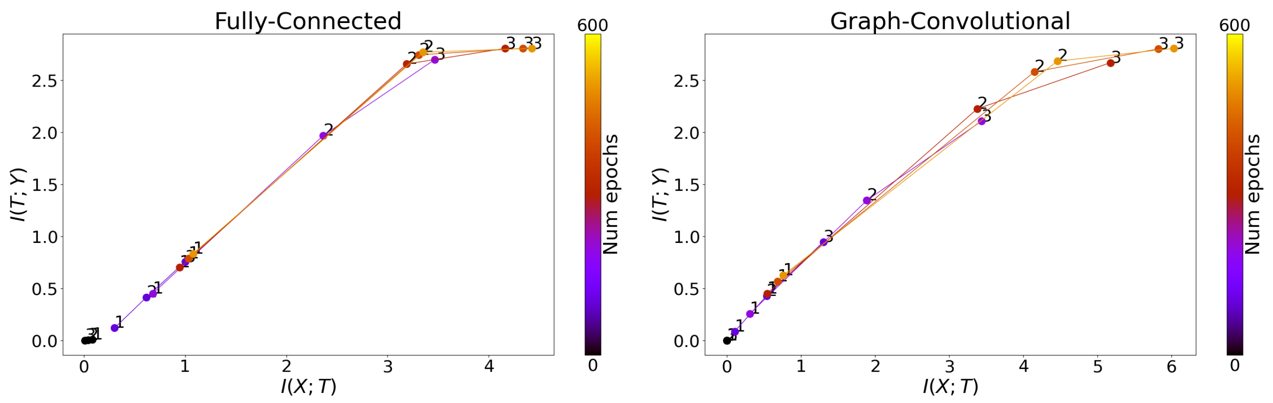

Figure 5.3: The information plane of the FCN and GCN models on the Cora dataset, with a noise parameter of $\sigma^2 = 0.1$.

The more unsettling fact is that the DPI (Eqn. 3.4) is violated when using $\sigma^2 = 0.1$ with the FCN model in Figure 5.3. This should not happen, since the features Z^{l+1} are a deterministic function of Z^l in the FCN model. The GCN model would not be expected to fulfill the DPI, since Z^{l+1} is computed by first averaging over all the neighboring representations Z^l. It is not clear why the DPI is violated, but the issue will be further discussed in Section 5.2.

5.1.2 Recurrent Neural Networks and Fully-Connected Networks on Text Data

Due to RNNs being designed for sequential data, we are going to compare a multi-layer RNN and a fully-connected network trained on the example name-country dataset from section 1.3.1. The RNN is a stacked, multi-layer RNN. As such, we are going to use the last hidden representations from each layer of the RNN. Given the input of length S, there are going to be S hidden representations for the first layer, $Z_1^{<1>}, Z_1^{<2>}, \dots Z_1^{<S>}$, where subscripts now indicate layer number. Hence we are going to track $Z_1^{<S>}, Z_2^{<S>}$ and $Z_3^{<S>}$. Figure 5.4 shows the training and validation accuracy of the FCN (left) and RNN (right) models on our text dataset from section 1.3.1. We note that the RNN is able to achieve a higher validation accuracy, which is a sign of the RNN being able to generalize better to unseen data as compared to the FCN model.

The information plane plots are shown in Figure 5.5 and 5.6 where we have chosen noise levels $\sigma^2 = 0.001$ and $\sigma^2 = 0.1$ respectively. We note that the representations from the RNN model need not satisfy the data-processing inequality, since the representation $Z_2^{<S>}$ is not solely a function of $Z_1^{<S>}$ but

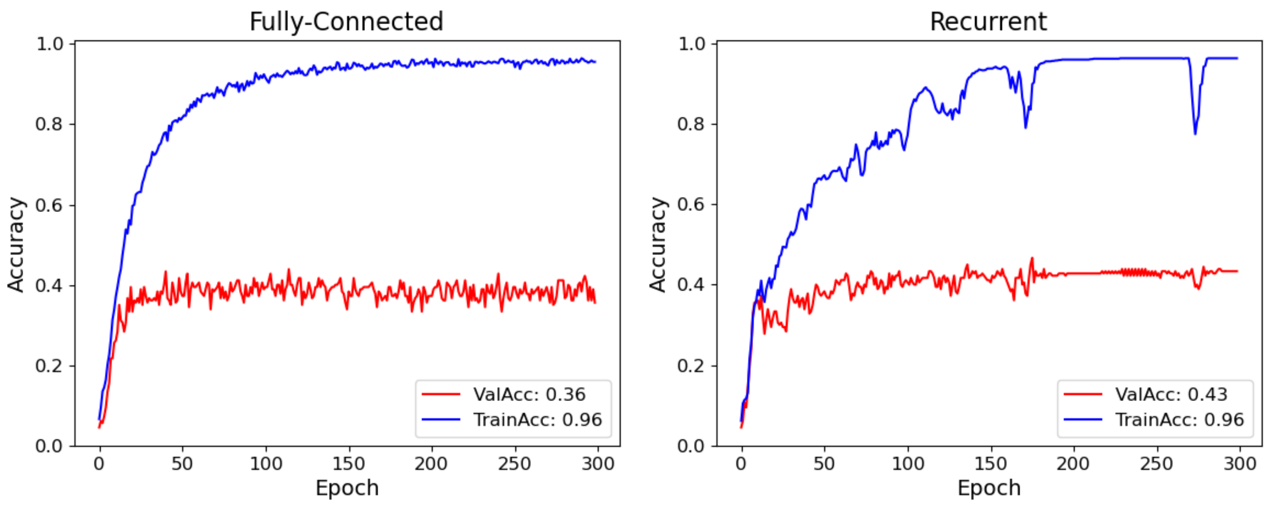

Figure 5.4: The training and validation accuracies for both the fully-connected (left) and recurrent (right) models on the text dataset. Both the FCN and the RNN had three hidden layers with 200 neurons in each.

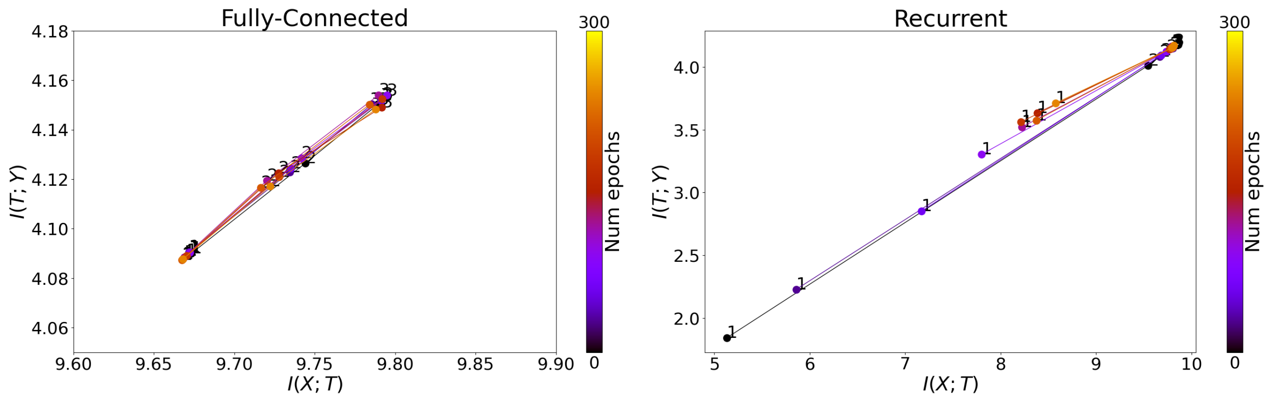

Figure 5.5: The information plane of the FCN and RNN models, with noise parameter $\sigma^2 = 0.001$.

also a function of $Z_2^{<S-1>}$, and similarly for $Z_3^{<S>}$. When using $\sigma^2 = 0.1$, we obtain the most interpretable plot for the RNN model, with the final layer containing the most information about both X and Y. It seems as though the RNN model does not ”compress” as one would expect, but rather that the increase in number of layers is able to capture more of the variation in the data than it otherwise would, and can be used to create better predictions. For the $\sigma^2 = 0.001$ case, we see a similar qualitative pattern, where the later layers have more information about both X and Y.

The plots for the FCN model in Figure 5.5 (left) and Figure 5.6 are not as expected. The most glaring fault is that they violate the data-processing inequality, by the later layers having more information about both X and Y compared to the initial layers. Several different values for σ^2 was attempted,

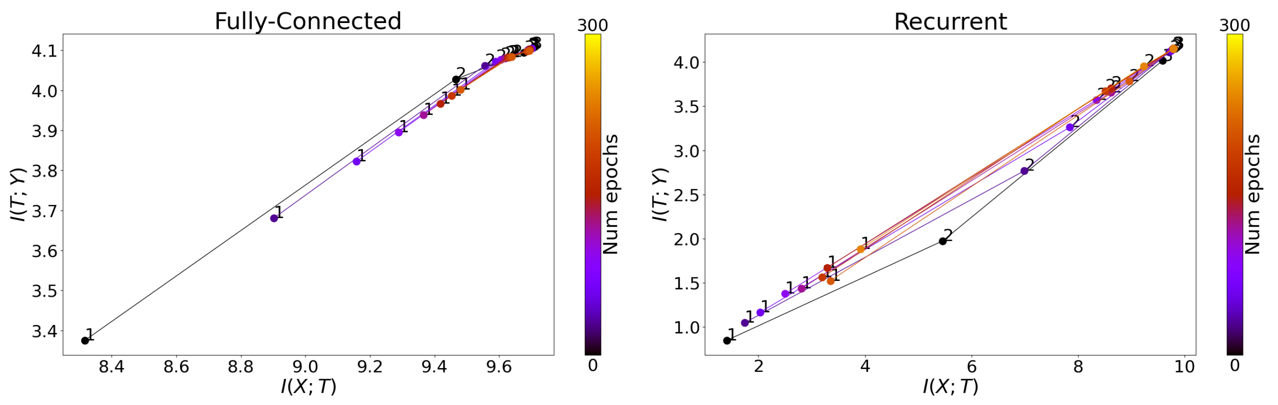

Figure 5.6: The information plane of the FCN and RNN models, with a noise parameter of $\sigma^2 = 0.1$.

but did not yield different qualitative behavior. The larger noise level $\sigma^2 = 0.1$ seems to help spread out the estimates of mutual information more than when using $\sigma^2 = 0.001$. A larger value of σ^2 allows for more representations i in layer l, Z_i^l, to affect the mutual information equation, as it can be interpreted as a bandwidth term in the exponential $\exp\left(\frac{\|Z_i^l - Z_j^l\|_2^2}{8\sigma^2}\right)$ in equation (3.18). Allowing more representations to contribute introduces more variance in the estimation process but it does not alleviate the flip.

5.1.3 Convolutional Neural Networks and Fully-Connected Networks on MNIST

Figure 5.7 shows the training and validation accuracy for a FCN model with hidden layers of size $1024 - 20 - 20 - 20$ (left) and the LeNet CNN architecture [24] on the right. The LeNet architecture consists first of two convolutional layers, with the the inital layer having 10 channels and the subsequent layer having 20 channels, both using a ReLU activation function. Max-pooling [8] is performed after each convolutional operation. This is followed by a linear layer with hidden size of 50 and then the prediction layer with 10 activations. Both the LeNet and FCN models are trained on a subset of 4000 images from the MNIST [3] dataset. A subset is used due to memory requirements. The FCN was trained with a learning rate of $1e-5$, whilst LeNet was trained with a learning rate of $1e-3$ for fewer epochs.

Figure 5.8 shows the information plane for the FCN and LeNet models, using noise parameter $\sigma^2 = 0.1$. The information plane for the FCN model is very well behaved, and is similar to information planes from previous work [67]. The paper [67] also visualizes an information-plane plot for the LeNet model.

However, it seems as if they only visualize the information plane for the two linear layers, shown in (Figures 4c) and 4d) in [67]). Hence Figure 5.8 may be the first plot actually visualizing the mutual information of the convolutional representations. The authors of [67] also visualize the information plane for a DenseNet [79] model, however it is not stated which activations they track.

The main issue with the mutual information estimates for the first convolutional layer, labeled with a ”1” in Figure 5.8, is that they violate the upper bound of the mutual information $I(Y,Z) \leq H(Y)$. This is a problem since we use the Bhattacharyya distance, which yields an estimate for the lower bound. In our case, we have $H(Y) = 3.32$, which is lower than a uniform distribution over ten classes which would have $H^*(Y) = 3.318$, where the entropy is computed by equation (3.2). The difference is likely due to choosing an unsuitable σ^2. However we note that choosing $\sigma^2 = 0.001$ instead of 0.1 yielded a noisy information plane. In section 5.2 we are going to employ a heuristic in order to determine different values of σ^2 for each layer in the network.

We surmise that the convolutional layers require different choices of σ^2 than linear layers since their parameters and transformation are inherently different than for linear layers. This would create a difference in the distribution of the pairwise distances, which the estimation method depends upon, as presented in Section 3.2.1. Both the RNN and GCN use regular matrix-multiplications with dense matrices in order to create their representation, but the CNN is particular since it applies a convolution operation.

5.1.4 Overfitting And The Information Plane

We would also like to note that in the cases where overfitting was present, it was more likely for the information planes of the fully-connected networks to have an abnormal behavior. This is apparent for the Cora dataset, where the information plane for the FCN model flips by changing noise parameter σ^2. The information plane of the FCN model on the text dataset was also flipped, in the sense that the later layers were up and to the right of the earlier layers, violating the DPI. On the Tishby example data and for MNIST, there were not any signs of the information plane flipping, and in both these cases the FCN model did not overfit the training data in any appreciable way. The author is uncertain as to how this can be used in practice, since very few modern challenges in deep learning are solved with pure fully-connected networks,

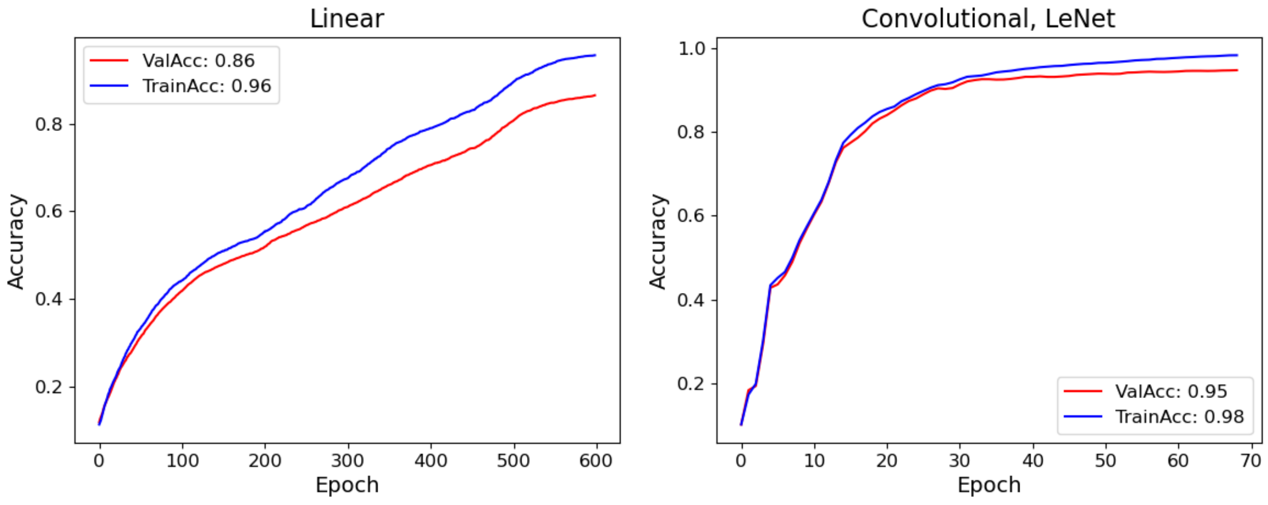

Figure 5.7: The training and validation accuracies of a FCN model with hidden sizes of $1024-20-20-20$ (left) and the LeNet [24] CNN architecture (right).

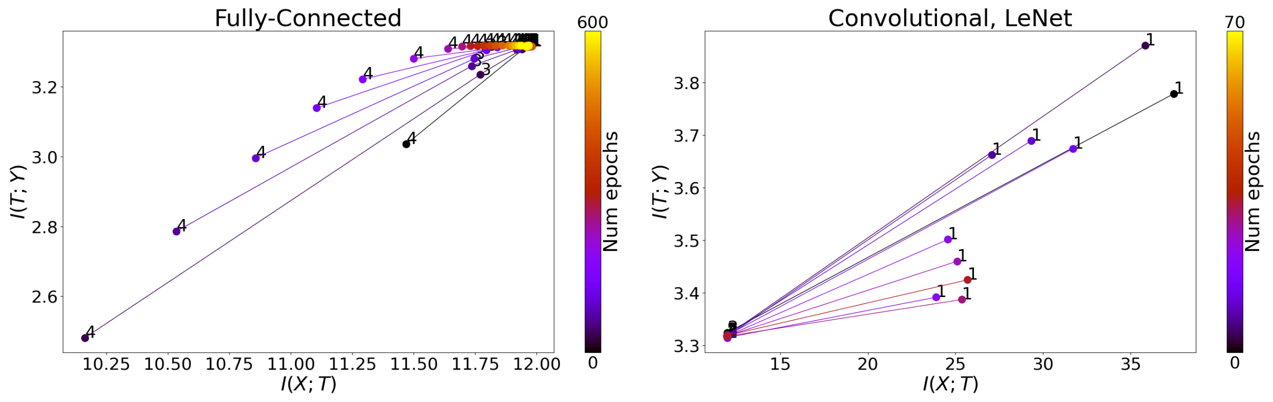

Figure 5.8: The information plane of the FCN and CNN models, both estimated using $\sigma^2 = 0.1$.

however some recent work has tried replacing CNNs with FCNs for image classification [80].

It does not seem that overfitting impacts the representations getting grouped into a single point, as is apparent for the FCN trained on MNIST and the FCN trained on the Cora dataset. If the representations adhered to the DPI before converging to a cluster, they would also satisfy the DPI when they were tightly grouped in the cluster, up to numerical estimation uncertainties, on the order of 10^{-3} to 10^{-5} in both cases.

5.2 Analysis of Mutual Information Estimation

In this section we are going to investigate the representations of the FCN trained on the Cora dataset and the representations of the CNN trained on the MNIST dataset. The reason for this is that the information plane for the FCN trained on the Cora dataset completely changed its qualitative behavior when changing σ^2, which is unexpected. The CNN trained on the MNIST dataset also had an abnormal behavior in the information plane when compared to any of the other models. When changing the noise level σ^2 from 0.001 to 0.1 for the FCN, it resulted in an information plane plot which violated the DPI. The estimate $I(Y, Z)$ for the CNN model also violates the upper bound on the mutual information, which is the entropy $H(Y)$.

We are going to use the distribution over pairwise distances in order to investigate if it can be used as a heuristic when determining σ^2 for both the CNN trained on MNIST and the FCN trained on the Cora dataset.

Figure 5.9 shows the distribution over distances for the FCN trained on the Cora dataset. Each hidden layer has a hidden representation of length 200. The x-axis is the computed distance, whilst the y-axis is the count of distances in each bin. As a heuristic, we use the 90th percentile value and divide it by 1000 to get our noise parameter. The percentile value is picked by eye from the plot. Doing this, we get noise parameter values of $0.001, 0.003$ and 0.01 for layers one through three, respectively. Figure 5.10 shows the result of using these noise levels when generating the information plane. We note that by using the distance-dependent noise heuristic on this FCN the first layer starts in the top-right corner of the plot, which resembles the behavior of the information plane visualization on the data from Tishby [10] in Figure 4.1.

We do not observe a compression phase in this network, however this may be attributed to the hidden layer structure of $200-200-200$ that does not force the network to compress into shorter-length features. Using the distance distribution as a heuristic for choosing the noise level however yields a much more interpretable information plane visualization. The initial layer contains the most information about both the input X and output Y, and the final layer, during training, learns the information that the first layer contains.

Figure 5.11 and 5.12 shows the distance distributions for the LeNet model and the fully-connected model on the MNIST dataset, respectively. Looking at the first two layers of each model, we note that the convolutional layer

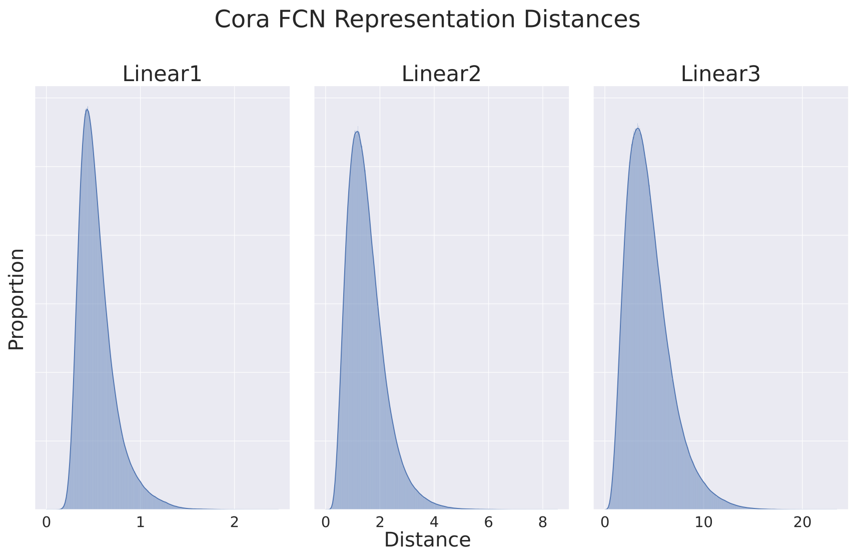

Figure 5.9: The distribution over pairwise distances for each layer of the FCN trained on the Cora dataset. The representations are from the final epoch of training. Each hidden layer has a hidden representation length of 200. The x axis is the pairwise distance, and the y-axis is the proportion of distances within a specified interval.

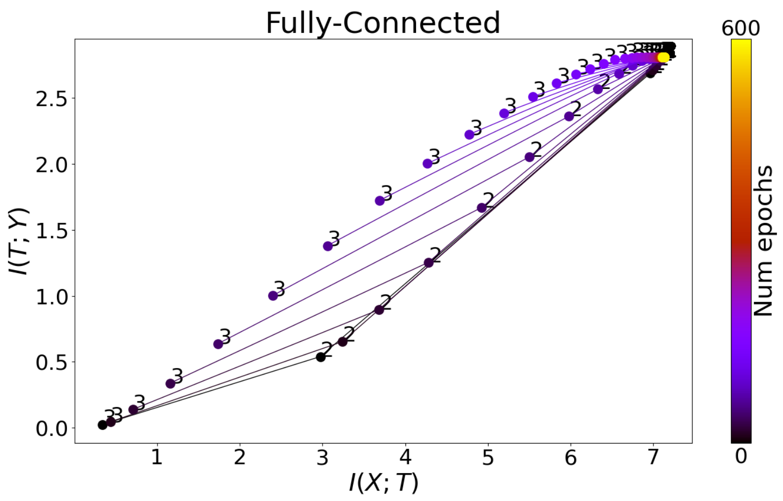

Figure 5.10: The information plane for the fully-connected network trained on the Cora dataset, using layer-wise noise terms σ^2 where layers one through three have noise levels $\sigma^2 = 0.001$, 0.003, and 0.01 respectively.

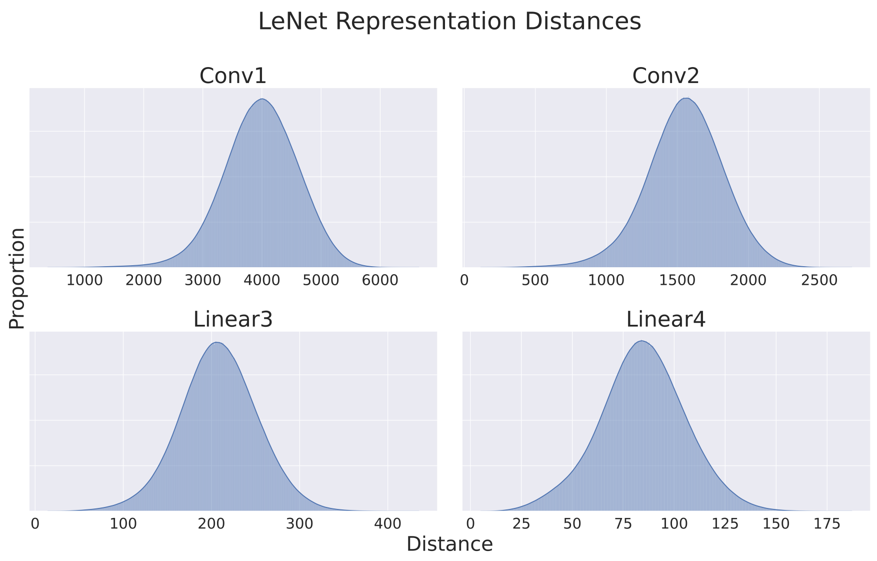

Figure 5.11: The distribution over pairwise distances for the LeNet architecture trained on the MNIST dataset. The representations are from the final epoch of training. The x-axis is the pairwise distance, and the y-axis is the proportion of distances within a specified interval.

tends to have a much larger distance between different samples in the dataset. The linear layers tend to group different samples closer than the convolutional layers. We used the same heuristic as for the Cora model, where we took the value at the 90th percentile and divided it by 1000 to obtain σ. From Figure 5.11, this results in noise levels of $5, 2, 0.3$, and 0.125 for layers one through four, respectively. Figure 5.13 shows the information plane with this noise configuration. The major improvement here is that the upper bound satisfies the technical bound $I(Y,Z) \leq H(Y)$ where $H(Y)$ is the entropy of the labels, in our case $H(Y) = 3.32$, measured in bits. The third and fourth representations are tightly clustered, and we also note that the third layer does not satisfy the DPI, in particular regards to Y. However, Figure 5.13 is a considerable improvement over the previous visualization in Figure 5.8 where the constant noise level $\sigma^2 = 0.1$ was used.

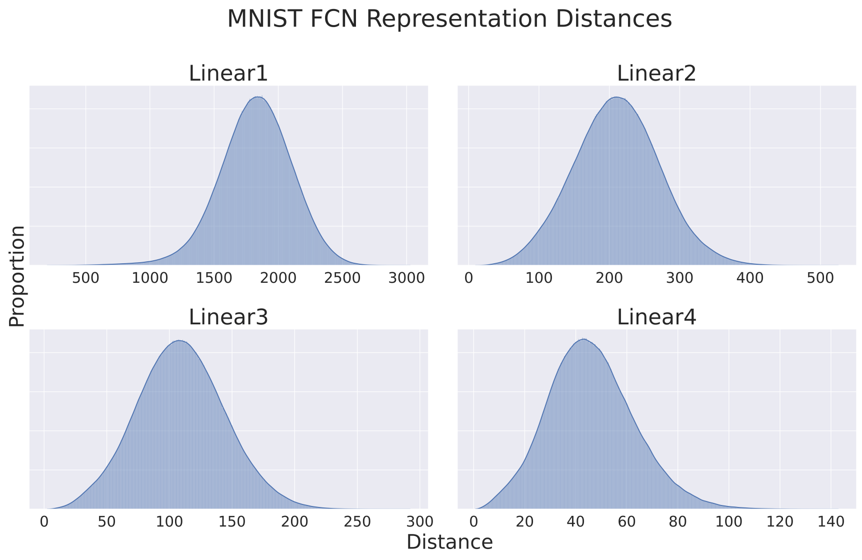

Figure 5.12: The distribution over pairwise distances for a FCN architecture with hidden layer sizes $1024-20-20-20$ trained on the MNIST dataset. The representations are from the final epoch of training. The x-axis is the pairwise distance, and the y-axis is the proportion of distances within a specified interval.

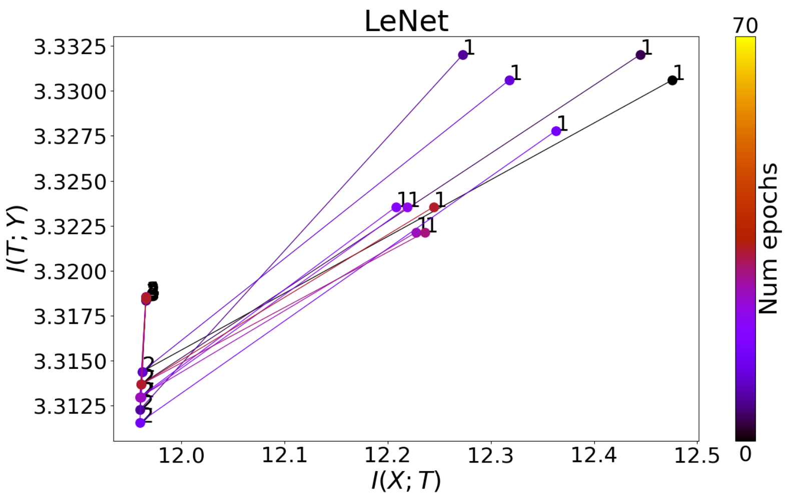

Figure 5.13: The information plane for the LeNet model trained on the MNIST dataset. For layers one through four we have used noise levels $\sigma^2 = 5$, 2, 0.3, and 0.125 respectively.

Chapter 6

Conclusion and Further Work

In this thesis, we have progressed from previous work [10, 64, 67, 66] and created a more flexible framework for mutual information estimation that is easily adapted to different neural network architectures. This has been achieved through the ability to track any activations as long as the modules are stored as submodules in a PyTorch model, which enables the tracking of representations from both FCN, GCN, RNN and CNN models. Tracking the hidden layer representations allowed for the first information plane plots of GCN and RNN models as far as the author is aware of. As for the CNN model, it seems as if the authors of [67] only tracked the activations from the linear layers of LeNet, and not the convolutional layers. It is uncertain as to which activations from DenseNet [79] they tracked.

The main motivation for this thesis was the possibility of using the information plane to determine if certain neural network structures were more suited for a particular kind of data than others. If so one could use the information plane as a reliable alternative in order to gauge model fit, or to evaluate if the model has generalized well. Unfortunately, there are two main drawbacks for using the information-plane in order to compare different neural structures. Firstly, the pairwise-distance estimation method used by Kolchinsky et al. [71, 13] is sensitive to the choice of σ^2, and also sometimes violates the DPI. Secondly, not all neural model structures necessarily satisfy the DPI, making comparisons difficult. A possible solution would be to find an estimation method that works more reliably without being required to tune any parameters. We tried applying the MINE [15] estimation method, but it had numerical issues on for instance the Cora dataset. The Rényi [14] estimation method was too time consuming in order to be practically feasible for extensive testing.

In conclusion, it seems as if methods of estimating mutual information are not robust enough in order to tackle the vast variety of neural architectures and the variety of data used for different deep learning applications. Further work could be to analyze the failure modes of the estimation methods, and propose a different estimation scheme. More analyzes can also be done in order to understand if underfitting or overfitting produces different behaviors in the information plane, and in particular if they are related to the violation of the DPI.

References

[1] A. Barragao. Cora citation dataset. https://relational.fit.cvut.cz/dataset/CORA(Retrieved: 18.5.21).

[2] Nlp from scratch: classifying names with a character-level rnn. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html(Retrieved: 12.5.21).

[3] Y. LeCun and C. Cortes. MNIST handwritten digit database, 2010. URL: http://yann.lecun.com/exdb/mnist/.

[4] J. Dean. The deep learning revolution and its implications for computer architecture and chip design. *CoRR*, abs/1911.05289, 2019. arXiv: 1911.05289. URL: http://arxiv.org/abs/1911.05289.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020. arXiv: 2005.14165 [cs.CL].

[6] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models, 2020. arXiv: 2006.11239 [cs.LG].

[7] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, and A. Ku. Image transformer. *CoRR*, abs/1802.05751, 2018. arXiv: 1802.05751. URL: http://arxiv.org/abs/1802.05751.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] R. S. Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

[10] W. B. Naftali Tishby Fernando C. Pereira. The information bottleneck method, 2000. URL: https://arxiv.org/abs/physics/0004057.

[11] R. Shwartz-Ziv and N. Tishby. Opening the black box of deep neural networks via information. *CoRR*, abs/1703.00810, 2017. arXiv: 1703.00810. URL: http://arxiv.org/abs/1703.00810.

[12] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. URL: https://faculty.marshall.usc.edu/gareth-james/ISL/.

[13] A. Kolchinsky and B. D. Tracey. Estimating mixture entropy with pairwise distances. *CoRR*, abs/1706.02419, 2017. arXiv: 1706.02419. URL: http://arxiv.org/abs/1706.02419.

[14] X. Yu, S. Yu, and J. C. Prıncipe. Deep deterministic information bottleneck with matrix-based entropy functional. *CoRR*, abs/2102.00533, 2021. arXiv: 2102.00533. URL: https://arxiv.org/abs/2102.00533.

[15] I. Belghazi, S. Rajeswar, A. Baratin, R. D. Hjelm, and A. C. Courville. MINE: mutual information neural estimation. *CoRR*, abs/1801.04062, 2018. arXiv: 1801.04062. URL: http://arxiv.org/abs/1801.04062.

[16] A. K. Kolmogorov. On the representation of continuous functions of several variables by superposition of continuous functions of one variable and addition. *Doklady Akademii Nauk SSSR*, 114:369–373, 1957.

[17] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[18] J. Schmidhuber. Deep learning in neural networks: an overview. *CoRR*, abs/1404.7828, 2014. arXiv: 1404.7828. URL: http://arxiv.org/abs/1404.7828.

[19] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pages 762–770, 1981.

[20] J. Launay, I. Poli, F. Boniface, and F. Krzakala. Direct feedback alignment scales to modern deep learning tasks and architectures, 2020. arXiv: 2006.12878 [stat.ML].

[21] Artificial neural nets finally yield clues to how brains learn. https://www.quantamagazine.org/artificial-neural-nets-finally-yield-clues-to-how-brains-learn-20210218/ (Retrieved:25.5.21.

[22] Cs 230 - deep learning: recurrent neural networks cheatsheet. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks(Retrieved: 25.5.21).

[23] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86 of number 11, pages 2278–2324, 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: a large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[27] Papers with code: image classification on imagenet. https://paperswithcode.com/sota/image-classification-on-imagenet (Retrieved: 24.5.21).

[28] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[29] T. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11 – NIPS Conference, Denver, Colorado, USA, November 30 - December 5, 1998)*, pages 487–493. The MIT Press, 1999. ISBN: 0-262-11245-0.

[30] H. Pham, Q. Xie, Z. Dai, and Q. V. Le. Meta pseudo labels. *CoRR*, abs/2003.10580, 2020. arXiv: 2003.10580. URL: https://arxiv.org/abs/2003.10580.

[31] D. Mahajan, R. B. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten. Exploring the limits of weakly supervised pretraining. *CoRR*, abs/1805.00932, 2018. arXiv: 1805 . 00932. URL: http://arxiv.org/abs/1805.00932.

[32] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. arXiv: 2010 . 11929. URL: https://arxiv.org/abs/2010.11929.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[34] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur. Sharpness-aware minimization for efficiently improving generalization. *CoRR*, abs/2010.01412, 2020. arXiv: 2010.01412. URL: https://arxiv.org/abs/2010.01412.

[35] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. *CoRR*, abs/1707.02968, 2017. arXiv: 1707 . 02968. URL: http://arxiv.org/abs/1707.02968.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: an imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[37] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković. Geometric deep learning: grids, groups, graphs, geodesics, and gauges, 2021. arXiv: 2104.13478 [cs.LG].

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F.-F. Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014. arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[39] T-sne visualization of cnn codes. https://cs.stanford.edu/people/karpathy/cnnembed/(Retrieved: 1.6.21).

[40] Mnist dataset. https://deepai.org/dataset/mnist(Retrieved: 12.6.21).

[41] Cs231n: visualizing what convnets learn. https://cs231n.github.io/understanding-cnn/(Retrieved: 1.6.21).

[42] R. J. Williams. Complexity of exact gradient computation algorithms for recurrent neural networks. 1989. Technical Report NU-CCS-89-27, Boston: Northeastern University.

[43] Wikimedia:graphimage. https://commons.wikimedia.org/wiki/File:6n-graf.png(Retrieved: 12.6.21).

[44] Geometric foundations of deep learning. https://towardsdatascience.com/geometric-foundations-of-deep-learning-94cdd45b451d (Retrieved: 25.5.21).

[45] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *CoRR*, abs/1611.08097, 2016. arXiv: 1611.08097. URL: http://arxiv.org/abs/1611.08097.

[46] V. G. Satorras, E. Hoogeboom, and M. Welling. E(n) equivariant graph neural networks. *CoRR*, abs/2102.09844, 2021. arXiv: 2102.09844. URL: https://arxiv.org/abs/2102.09844.

[47] D. Luo, Z. Chen, K. Hu, Z. Zhao, V. M. Hur, and B. K. Clark. Gauge invariant autoregressive neural networks for quantum lattice models, 2021. arXiv: 2101.07243 [cond-mat.str-el].

[48] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907.

[49] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[50] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/. Software available from tensorflow.org.

[51] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[52] Torch.nn.functional. https://pytorch.org/docs/stable/nn.functional.html(Retrieved: 1.6.21).

[53] Towardsdatascience: pytorchautograd. https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95(Retrieved: 12.6.21).

[54] F. Chollet et al. Keras. 2015. URL: https://github.com/fchollet/keras.

[55] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization, 2017. arXiv: 1412.6980 [cs.LG].

[56] Adam optimizer in pytorch. https://pytorch.org/docs/stable/optim.html#torch.optim.Adam(Retrieved: 18.5.21).

[57] Cs231n: convolutional neural networks for visual recognition. https://cs231n.github.io/(Retrieved: 18.5.21).

[58] Deep learning optimization. https://www.deeplearning.ai/ai-notes/optimization/(Retrieved: 18.5.21).

[59] Cs231n: neural networks. https://cs231n.github.io/neural-networks-3/(Retrieved: 12.6.21).

[60] P. Márquez-Neila, M. Salzmann, and P. Fua. Imposing hard constraints on deep networks: promises and limitations. *CoRR*, abs/1706.02025, 2017. arXiv: 1706.02025. URL: http://arxiv.org/abs/1706.02025.

[61] D. Pathak, P. Krähenbühl, and T. Darrell. Constrained convolutional neural networks for weakly supervised segmentation. *CoRR*, abs/1506.03648, 2015. arXiv: 1506.03648. URL: http://arxiv.org/abs/1506.03648.

[62] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics informed deep learning (part I): data-driven solutions of nonlinear partial differential equations. *CoRR*, abs/1711.10561, 2017. arXiv: 1711.10561. URL: http://arxiv.org/abs/1711.10561.

[63] N. Wolchover. New theory cracks open the black box of deep learning. URL: https://www.quantamagazine.org/new-theory-cracks-open-the-black-box-of-deep-learning-20170921. (accessed: 26.11.2020).

[64] N. Tishby and N. Zaslavsky. Deep learning and the information bottleneck principle. *CoRR*, abs/1503.02406, 2015. arXiv: 1503.02406. URL: http://arxiv.org/abs/1503.02406.

[65] T. M. Cover and J. A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, USA, 2006. ISBN: 0471241954.

[66] Andrew Saxe, Yamini Bansal, Joel Dapello, Madhu Advani, Artemy Kolchinsky, Brendan Tracey, and David Cox. On the information bottleneck theory of deep learning, 2017. URL: https://openreview.net/pdf?id=ry_WPG-A-.

[67] H. Fang, V. Wang, and M. Yamaguchi. Dissecting deep learning networks—visualizing mutual information, Jan. 2018. URL: https://hdl.handle.net/2134/36577.

[68] Netflix. Netflix movie size. https://help.netflix.com/en/node/87.

[69] K. W.-D. Ma, J. P. Lewis, and W. B. Kleijn. The HSIC bottleneck: deep learning without back-propagation. *CoRR*, abs/1908.01580, 2019. arXiv: 1908.01580. URL: http://arxiv.org/abs/1908.01580.

[70] R. Pogodin and P. E. Latham. Kernelized information bottleneck leads to biologically plausible 3-factor hebbian learning in deep networks, 2020. arXiv: 2006.07123 [cs.LG].

[71] A. Kolchinsky, B. D. Tracey, and D. H. Wolpert. Nonlinear information bottleneck. *CoRR*, abs/1705.02436, 2017. arXiv: 1705.02436. URL: http://arxiv.org/abs/1705.02436.

[72] A. Ruderman, M. D. Reid, D. Garcıa-Garcıa, and J. Petterson. Tighter variational representations of f-divergences via restriction to probability measures. *CoRR*, abs/1206.4664, 2012. arXiv: 1206.4664. URL: http://arxiv.org/abs/1206.4664.

[73] M. Donsker and S. Varadhan. Asymptotic evaluation of certain markov process expectations for large time. *Communications on Pure and Applied mathematics*, 1983. 36(2):183?212.

[74] Pytorch: forward and backward function hooks. https://pytorch.org/tutorials/beginner/former_torchies/nnft_tutorial.html#forward-and-backward-function-hooks(Retrieved: 18.5.21).

[75] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN: 0262039249.

[76] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik,

I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[77] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[78] C. Zhang, O. Vinyals, R. Munos, and S. Bengio. A study on overfitting in deep reinforcement learning. *CoRR*, abs/1804.06893, 2018. arXiv: 1804.06893. URL: http://arxiv.org/abs/1804.06893.

[79] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. arXiv: 1608.06993. URL: http://arxiv.org/abs/1608.06993.

[80] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy. Mlp-mixer: an all-mlp architecture for vision. *CoRR*, abs/2105.01601, 2021. arXiv: 2105.01601. URL: https://arxiv.org/abs/2105.01601.

NTNU

Norwegian University of
Science and Technology