Andrius Grabauskas

Measuring mutual information in Neural Networks

Computer Science Tripos – Part II

Robinson College

Saturday 11th May, 2019

Declaration

I, Andrius Grabauskas of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Proforma

Name:	**Andrius Grabauskas**
College:	**Robinson College**
Project Title:	**Measuring mutual information in Neural Networks**
Examination:	**Computer Science Tripos – Part II, July 2001**
Word Count:	**6208**[1]
Line Count:	**1303**[2]
Project Originator:	Dr. Damon Wischik
Supervisor:	Prof. Alan Mycroft

Original Aims of the Project

Work Completed

Special Difficulties

[1]This word count was computed by `detex *.tex | tr -cd '0-9A-Za-z \n' | wc -w`

[2]This line count was computed by `wc -l **/*.py`

Acknowledgements

Chapter 1

Introduction

Deep Neural Networks (DNNs) are an extremely successful tool, they are widely adopted commercially and closely studied academically, however even given the attention they have gathered there is no comprehensive understanding of how these models generalize data and provide such impressive performance - in fact very little is know about how DNNs learn or about their inner workings. Recently Prof. Tishby produced a paper claiming to understand the basic principle of how DNNs work. He suggested that there are two phases that the network goes trough while being trained - the fitting phase and the compression phase. During the fitting phase the network memorizes the training data and makes predictions based on what it has seen before, during the compression phase the network generalizes, it forgets the unnecessary information from the training data. Tishby suggested that the incredible performance that DNNs are able to achieve is due to this compression phase, and that this process of compression is a result of randomness inherent in Stochastic gradient descent. Tishby showed this by looking at DNNs trough the information domain, most notably he used what is now called the information plane. The information plane summarizes how the information is flowing trough the DNN, for every neuron layer the plane shows mutual information it has with the input data and the label data. In his experiments Tishby has concluded that every layer loses unnecessary information from the input data and tries to keep information of the label. Tishby made some interesting and significant claims about how DNNs work, however he did no provide a formal proof, his conclusion are based only on experimental evidence.

In our work we look at Tishby's claim that DNNs compress data and throw away unnecessary information about the input. We reimplement his experiments as a form of independent verification, showing that the results Tishby got are robust and are stable to parameter changes. We also take a look at a paper produced by Saxe that provides an opposing view to that of Tishby's. Saxe showed that compression that Tishby showed is only a result of Tishby's choice of activation function for the neural network. He showed that compression only happens when Tanh activation function is used and does not happen when ReLu is used.

Lastly, we think that the experiments presented in both papers don't fully align with the ideas that Tishby presented to us, specifically his idea that weights should be treated 'as if' they are random variables. Tishby's and Saxe's experiment deal with this 'as if' random notion quite crudely or try to sidestep it completely. As a result we devised an experiment that tries to capture this idea more explicitly, although it is still relatively crude and more work should be put into it in the future.

1.1 The Information plane

If we look at a neural network trough the lens of information we can think of it as a form of a Markov chain. Where every node takes in a piece of data processes it and passes to the next node. For every such node $t \in T$ we can measure it's mutual information with the input patterns $x \in X$ and labels $y \in Y$.

$$Y - X \rightarrow T_1 \rightarrow ... \rightarrow T_i \rightarrow \hat{Y}$$

In this analogy a node corresponds to a layer in a non-convolutional neural network. When looking at a neural network in this way we can say that it's job is to preserve as much information as possible about the label, or to minimise information loss about the label during the transitions from node to node.

The Information plane summarized this Markov chain view and applies it to the entirety of the training process.

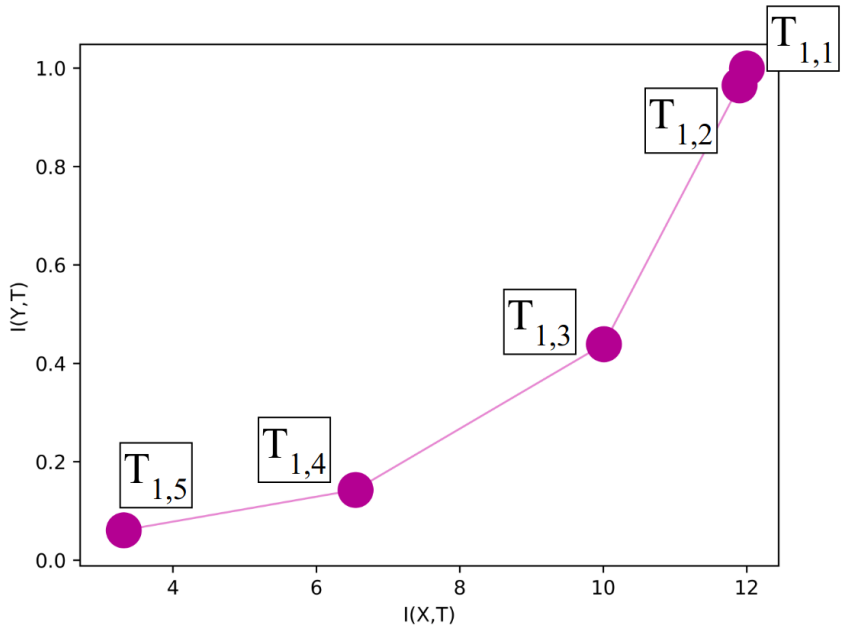


Figure 1.1: information plane for a neural network with 5 layers, which was only trained for one epoch.

Figure 1.1 shows an example of an information plane - in this case the network has 5 layers and has only been trained for one epoch. Every node corresponds to a layer, and the lines between them just help us distinguish the order of the layers. The $x - axis$ shows mutual information between any layer T and the input X, while the $y - axis$ shows mutual information with the label Y.

The figure bellow corresponds to a network that takes in a 12bit number and maps it to a 1bit number hence the mutual information ranges from 0 to 12 for the $x - axis$ and from 0 to 1 for the $y - axis$.

The upper right most node corresponds with the very first layer before any data processing occurs, therefore it has maximum mutual information with the input and the label. As the network was only trained for one epoch the weight are essentially random and we see a steep drop in every subsequent layer, with the last layer of the network having close to 0 mutual

information with X and Y, which is what we expect if we assume that the network guesses at random before any training is done.

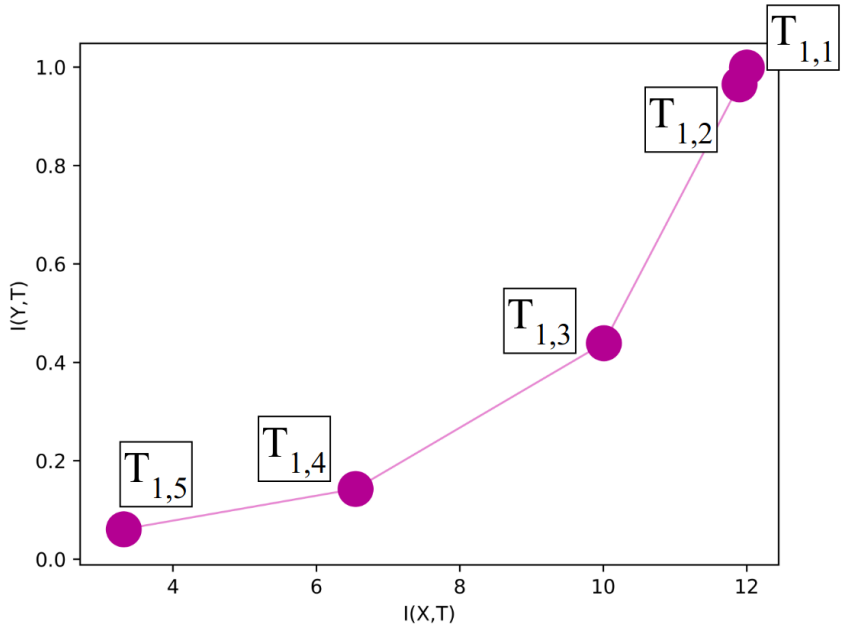


Figure 1.2: Information plane for a neural network with 4 layers, which was trained for approximately 10 000 epochs

Figure 1.2 shows an example of a full information plane. The gradient maps the colour to the epoch. From this figure we can clearly see the fitting and the compression phases Tishby was describing.

- The Fitting phase is the rapid improvement phase at the start of the training period, where information about the label is rapidly increasing while information about the input remains approximately the same. In this case the training is in the fitting phase for less than 1000 epochs.
- The Compression phase begins when the after the fitting phase ends and the rapid improvement stops. We can see during this phase we start losing information about the input while still gaining information about the label. This phase the was majority of the training time.

Chapter 2

Preparation

- Tishby produced a paper [1] claiming to understand the basic principles of how DNNs work.
- He decided to examine neural networks through the information domain, visualizing the results via Information Plane method section 2.3.
- Tishby made the claim that the incredible performance DNNs are able to achieve is due to the their ability to compress information inherent in the input data. Compressing data means the network is only able to keep relevant input features and it must forget the irrelevant bits of information, leading to the ability to generalize.
- Tishby made interesting claims and provided experimental evidence to support his claims, however he did not provide a formal proof leaving his results up for debate.
- Paper released by Saxe has contested the claims made by Tishby arguing that compression cannot happen in Neural Networks and the results are a consequence of the hyper parameters Tishby used.
- However Saxes problem suffers from the same problem as Tishby's as it does not provide a formal proof only experimental evidence, as such it doesn't settle the rebuttal.
- To fully understand the discussion we need to understand the following topics Entropy and Mutual Information, Neural Networks, and Information Plane, described in section 2.1, section 2.2, and section 2.3.

2.1 Entropy and Mutual Information

Entropy – quantifies information content of a random variable. It is generally measured in bits and can be though of as the expected information content when we sample a random variable once. Let X be a discrete random variable that can take values in $\{x_1, ..., x_n\}$. $H(X)$, the entropy of X, is defined by Equation 2.1.

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{2.1}$$

Consider a random variable Y s.t $P(Y = 1) = P(Y = 0) = 0.5$, Equation 2.1 defines $H(Y)$ to be 1.

Similarly for a random variable Y s.t $P(Y = 0) = 0.5, P(Y = 1) = P(Y = 2) = 0.25$, we have $H(Y) = 1.5$.

Conditional Entropy – quantifies the amount of information needed to describe an outcome of variable Y given that value of another random variable X is already known. Conditional Entropy of Y given X is written as $H(Y|X)$. Let X be defined as before, Let Y be a discrete random variable that can take values in $\{y_i, ..., y_n\}$. The conditional entropy $H(Y|X)$ is defined by Equation 2.2.

$$H(Y|X) \ = - \sum_{x \in X, y \in Y} P(x, y) \log \frac{P(x, y)}{P(x)} \tag{2.2}$$

Let the correlated variables X and Y be defined by Table 2.1.

X \ Y	0	1
0	0.25	0.25
1	0.5	0

Table 2.1: Joint probability distribution for X and Y

Equation 2.1 and Equation 2.2 defines entropy values to be:

$$\begin{aligned} H(Y|X) &= 0.5 \\ H(X|Y) &\approx 0.6887 \\ H(X) &= 1 \\ H(Y) &\approx 0.8112 \end{aligned} \tag{2.3}$$

Mutual Information (MI) – measures how much information two random variables have in common. It quantifies information gained about one variable when observing the other. Equation 2.4 and Equation 2.5 are two examples of how we can compute mutual information, using explicit probability computations or entropies of the random variables respectively, here X and Y are as previously defined.

$$I(X, Y) = \sum_{y \in Y} \sum_{x \in X} P(x, y) \log \left(\frac{P(x, y)}{P(x)\, P(y)} \right) \tag{2.4}$$

$$I(X, Y) = H(X) - H(X|Y) \tag{2.5}$$

For example of mutual information consider the random variables X and Y as before in the conditional entropy section – defined by Table 2.1.

We computed the entropy values in Equation 2.3, we will use them in Equation 2.5 to compute $I(X, Y)$.

$$I(X, Y) = H(X) - H(X|Y) \approx 1 - 0.6887 = 0.3113 \tag{2.6}$$

2.2 Neural Networks

Before we understand neural networks we need to understand The Prediction Problem and the purpose of Machine Learning Frameworks.

2.2.1 The Prediction problem

Suppose we have some dataset (x_i, y_i) for $i = 1, ..., N$. The prediction problem is finding a function f s.t Equation 2.7 is satisfied.

$$f(x_i) = y_i \text{ for } i = 1, ..., N \tag{2.7}$$

Prediction task is a common task that involves having input data $\{x_i, ..., x_N\}$ and finding the label, some desirable feature, $\{y_i, ..., y_N\}$.

The prediction problem could be simple to extract: for example if our input is a natural number $x_i \in \mathbb{N}$, and our label is either $true$ or $false$ depending if x is even or odd – in which case function defined by Equation 2.8 satisfies the problem.

$$g(x) = \begin{cases} True, & \text{if } \exists n \in \mathbb{N}.x = 2n, \\ False, & \text{otherwise.} \end{cases} \tag{2.8}$$

The prediction problem also can be impossible to solve: for example the halting problem, if our x's are programs and y's boolean values corresponding if the program halts or not.

Of course the prediction problem can be hard or impossible to solve as is the case for problems:

input data	label	difficulty
medical symptoms	diagnosis	intractable
picture	object in the picture	
face photograph	identity	
stock market history	future stock prices	
program	does the program halt	proved to be unsolvable
boolean equation	is the equation satisfiable	expensive to compute

Table 2.2: Example of specific prediction problems

Problems listed in Table 2.2 are either intractable, unsolvable or too expensive to compute – hence we cannot produce an algorithm that always give the correct answer and runs in a reasonable time.

2.2.2 Machine Learning Frameworks

If the Prediction problem is too difficult and we are tolerant to errors in our labels we may want to use a supervised machine learning framework[1] to tackle the problem.

Every machine learning framework requires that we have some subset $\hat{X} \subseteq \{x_i, ..., x_N\}$ s.t that $\forall x \in \hat{X}$ we know label the y. A framework uses the data in order to reach some goal – such as minimizing the prediction error. The way any machine learning framework learns from data varies, but generally more data means an increase in prediction performance.

[1] In this thesis we are exclusively talking about supervised machine learning techniques – the word "supervised" will be omitted for brevity

2.2.3 Neural Networks

Neural Networks are an example of a machine learning framework. They learn from data and attempt to solve the prediction problem.

A Neural Networks structure consists of layers of fully connected nodes as in **??**. Every node takes a value in the real number space $\mathbb{R}$. Values for the Input nodes are provided, whereas values for the nodes in non-input layers are generated based on nodes from the preceding layer. Let $n_{l,i}$ be the value that the $i'th$ node in layer l takes. Let $w_{l,j,i}$ be a parameter that influences how relevant a node in layer l index j is to the node in layer $l+1$ index i.

$$n_{l+1,i} = f(\sum_{j=0}^{\text{layer } l \text{ size}} w_{l,j,i} * n_{l,j}) \tag{2.9}$$

f in **??** is called the activation function and is generally taken to be:

- ReLu: $f(x) = max(0, x)$
- Sigmoid: $f(x) = \frac{1}{1+e^x}$
- Tanh: $f(x) = tanh(x)$

For our purposes we will extract away the individual nodes in the network and

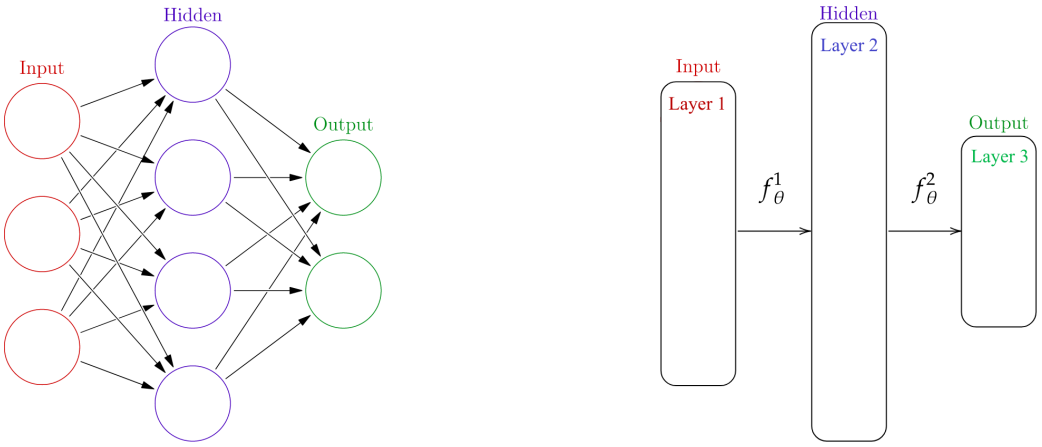
---

Neural Networks are a machine learning framework, they have been used effectively in predicting and classifying data.

Suppose we have a dataset:

$$(x_i, y_i) \text{ for } i = 1, ..., N$$
$$\text{where, } x_i \in \mathbb{R}^d, y_i \in \mathbb{R}^{d'}$$

where x_i is the input to the Neural Network and y_i is the label or the value we want the neural network to predict. For example suppose we are trying to classify if an image contains a cat or no, then x_i would be an encoding of an image and the label y_i could be 1 if the image contains a cat and 0 if it does not.



(a) Structure of a typical neural network. (b) Structure of our abstracted neural network.

Figure 2.1: Source: Wikimedia Commons

The network is structured in layers where every layer holds an intermediate representation of the final prediction output – let us call this intermediate representation an **activation** of that specific layer. For our purposes we will define a neural network to be a sequence of functions $f_\theta^1, f_\theta^2, ..., f_\theta^L$ that are parameterized by weights θ s.t

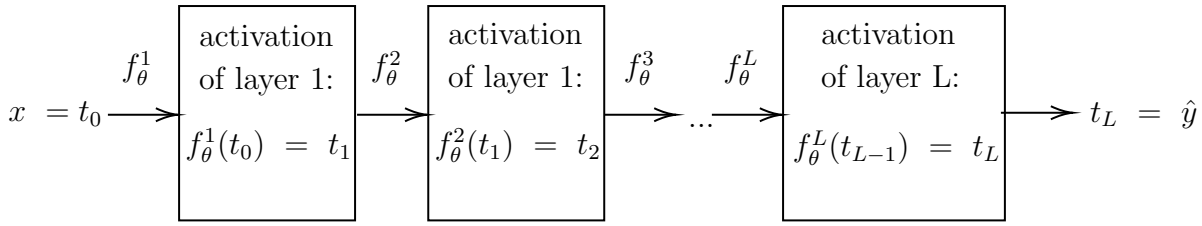


Figure 2.2: Visualization of a neural networks structure. x here is any input to the network from the set $\{x_i, ..., x_N\}$. $\hat{y}$ is the prediction of the network for the input x which may or may not be to the correct label. The values $t_1, ..., t_L$ here are **activations** of layers $1, ..., L$.

We have one function for each layer transition of the neural network this allows us to extract the activation of a specific layer as in Figure 2.3 – we will need to get access to intermediate layers in our experiments.

$$t_l = f_\theta^l(f_\theta^{l-1}(...(f_\theta^1(x))))$$

Figure 2.3: **activation** of layer l for input x

Stochastic Gradient Descent Each of the transition function are parameterized by the weights function θ. The θ function is directly controlled by Stochastic Gradient Descent (SGD) algorithm. SGD is an algorithm for training a neural network it gradually updates the weights θ according to some goal such as minimising the prediction error. SGD is an iterative process, it has a notion of epochs where one iteration of the algorithm advances epochs by one, this implies that θ function depends on which epoch we are currently at thus it must take the epoch number as an argument – $\theta(e)$.

2.3 The Information plane

2.3.1 Setup

The Information plane is a way of visualizing the Neural Network‘s training process through the information domain. By looking at mutual information between the input X the intermediate neural network layer activations T and the label Y we can see how the information flows through the network.

Mutual Information is only applicable to Probability distributions, however we only have the dataset (x_i, y_i) for $i = 1...N$, if we assume that every input x_i equally likely we can provide a routine that defines our probability distribution.

For convenience let us define F_θ^t to be the activation of layer t given input x, i.e

$$F_\theta^t(x) = f_\theta^t(f_\theta^{t-1}(...(f_\theta^1(x)))) \tag{2.10}$$

Consider Figure 2.4, the routine defines the random variables $X, T_{e,t}, Y$. Here $T_{e,t}$ is the distribution of layer t for the epoch e, X and Y is the original data with assumption that it is uniformly distributed.

Using the probability distributions we now have values:

- $I(T_{e,t}, X)$ – Mutual Information between the input distribution and the layer activations
- $I(T_{e,t}, Y)$ – Mutual Information between the label distribution and the layer activations.

This allows us to generate the Information plane.

```
1 def rxty(e, t):
2   pick i ∼ Uniform {1...N}
3   return (x_i, F^t_θ(e)(x_i), y_i)
```

Figure 2.4: Definition of correlated random variables $X, T_{e,t}$ and, Y

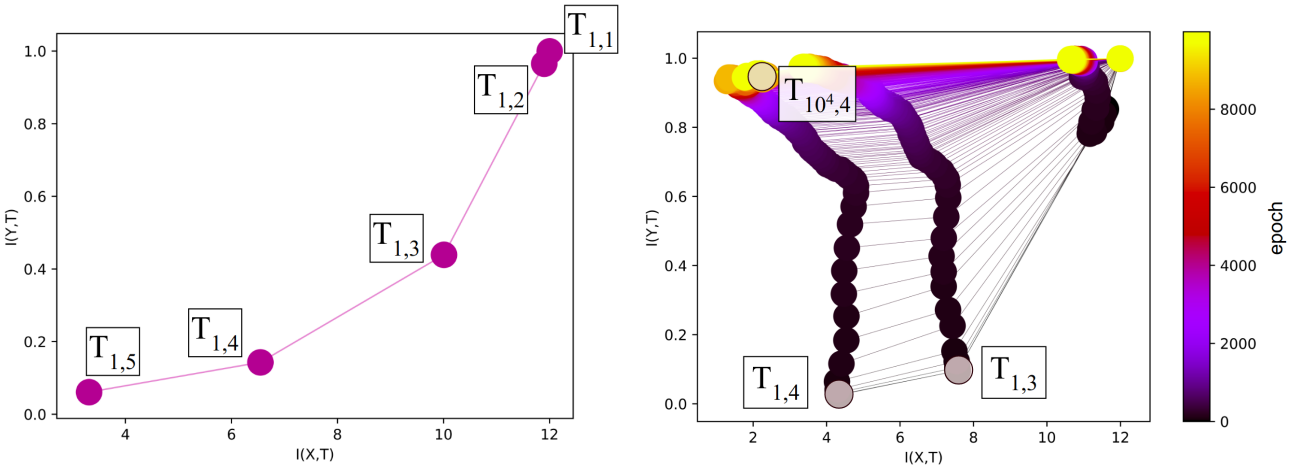
2.3.2 Visualization

The Information plane visualizes the whole training process, in order to generate the information plane we need $I(X, T_{e,t})$ and $I(Y, T_{e,t})$ for every epoch e and every layer t.

Consider for now that we only trained our neural network for one epoch, the Figure 2.5a shows an example of this.

In this case the network consists of 5 layers, in the figure every node corresponds to a distinct layer. The lines between the nodes help us distinguish epochs from each other and gives helps us to see the order of the layers, the upper-right-most node corresponds to the first layer in the neural network, the lower-left-most node corresponds to the fifth and last layer of the network.

Consider now Figure 2.5b, it shows an information plane for a full training phase. The color signifies what epoch the data belongs to and lets us see how the network progressed over time. We can see that at the start $I(Y, T_{1,4}) \approx 0$ meaning the network has not preserved any information about the label distribution Y, but by the end of the training we see $I(Y, T_{10^4,4}) \approx 1$ which means we have preserved almost all the information about the label.



(a) information plane for a neural network with 5 layers, which was only trained for one epoch.

(b) Information plane for a neural network with 4 layers, which was trained for approximately 10 000 epochs.

Figure 2.5: The Neural Networks in both figures have been trained on the same dataset as used by Tishby[1], hence input entropy, $H(X)$, is 12 and label entropy, $H(Y)$, is 1. Some Nodes are labeled $T_{e,l}$, where e is the epoch number and l is the layer the node belongs to. Consider $T_{1,3}$ from Figure (b) – The node corresponds to the information content of layer 3 for the 1'st epoch, the x coordinate of the node is the value $I(X, T_{1,3}) \approx 8$, the y coordinate is the value $I(Y, T_{1,3}) \approx 0.1$

2.3.3 Interpretation of the Information Plane

Let us once again consider Figure 2.5b, we can see two phases in the figure Tishby has named them The Fitting Phase and The Compression Phase.

The Fitting Phase In Figure 2.5b the neural network is in the fitting phase from the start of the training up until epoch ∼1500. The duration of the fitting phase varies heavily on the training parameters and is most influenced by the size of our input dataset. The fitting phase is characterized by:

- A rapid increase in $I(Y, T_{e,t})$, the information about the label, as we advance through the epochs e, the increase is especially visible in the later layers, in our case layers 3 and 4.
- Either an increase or no change in $I(X, T_{e,t})$, the information about the input, as we advance through the epochs e, in our case we see very little change in $I(X, T_{e,t})$.

During the fitting phase a neural network tries to memorize the data and make predictions based on the observations, this means that the network may learn useless features that only superficially correlate with the correct label.

The Compression Phase In Figure 2.5b the neural network enters the compression phase when the fitting phase ends around epoch ∼1500 and lasts until we finish the training process. The compression phase is characterized by:

- A slowdown of how fast $I(Y, T_{e,t})$ is increasing with respect to epochs e.

- A slow decrease of $I(X, T_{e,t})$ with respect to epoch e.

During the compression phase a neural network compresses representation of the input discarding more features that did not help with predicting correct labels. Discarding irrelevant features helps the neural network generalize and produce better predictions for new data.

Before developing a plan for how we are going to realize the project in code we needed to fully understand the ideas presented in the paper:

- We needed to identify the main ideas of the paper and understand why some parts of the paper are not agreed upon in the scientific community. Understand why his ideas are contentious and whether reproducing his experiments could bring more validity to his claims. This involved reading papers published by Tishby and academics who shown an opposing view to him.

- A main tool that the paper relies on is MIE (Mutual Information Estimation). Reading about MIE we quickly understood that MIE is a contentious part of the project as a result we had to do a decent amount of research regarding the subject. MIE is difficult because we are trying to estimate information between two continuous distributions using only a discrete sample set. This area has not seen much academic attention so the tools we ended up using could be greatly improved in the future.

Once we had a reasonable understanding of the ideas in the paper and which areas needed more attention we diverted our attention to figuring out the details of how the experiments were conducted figure out what hyper parameters Tishby decided are important and what assumptions he made whilst devising the experiments.

In addition we needed to find out what resources are available to us online, what programming frameworks we are going to use for the projects implementation, and to think about possible extensions to the project once the success criteria has been achieved.

- Online Resources: The two main papers by Tishby and by Saxe have made their code public online via Github, we made

  Online Resources: The two main papers we were looking at has made their code available to the public via Github, the papers are Tishby‘s paper and the main opposing paper by Saxe.

- Programming frameworks: The original experiment implementation by Tishby has used the Tensorflow framework. We have decided to use the Keras framework as it produces code that is more concise and is easier to read/maintain. Furthermore rewriting the experiments in a different framework means that we cannot rely on the details of Tishby's and potentially avoid any mistakes that may exist in the original implementation.

- Thinking about how we could extend the project helped us understand the scope of the project and what areas were most important and/or interesting to us.

  We came up with a couple of extensions before having written any code but the most interesting one only materialized after a good deal amount of work into the project (that is the AS-IF-Random experiment described below)

- Different Datasets : the most straight forward extension to the project just using different dataset to the one Tishby used. This is essentially just varying one of the parameters in the Neural Network. (Implemented)
- Quantized Neural Network : the idea behind this was to only allow single neurons to acquire values in a given range say 1...256. This would make the distribution within a DNN later discrete and hence it would make calculating mutual information straightforward. (Not Implemented)
- As-If-Random : one problem with Tishby's work is that he calculates mutual information for a single epoch at a time which by definition is zero (in his paper he tries to justify the result will explore this later) this extension tries to explore the weights of a neural network as random variables by calculating mutual information for multiple epochs at a time.

Chapter 3

Implementation

3.1 Compression In Neural Networks

Mutual Information Let us introduce two properties of mutual information

Let us revisit Mutual Information and introduce two properties that will be useful.

Invertible Transformation Let u and v be invertible functions; then,

$$I(A, B) = I(u(A), v(B)) \tag{3.1}$$

Data Processing Inequality Let $A \to B \to C$ be a Markov chain; then,

$$I(a, b) \geq I(a, c) \tag{3.2}$$

In the neural network case this implies

$$H(X) \geq I(X, T_{e,1}) \geq I(X, T_{e,2}) \geq ... \geq I(X, T_{e,N}) \text{ for all epochs } e, \text{ and} \tag{3.3}$$

$$I(X, Y) \geq I(T_{e,1}, Y) \geq I(T_{e,2}, Y) \geq ... \geq I(T_{e,N}, Y) \text{ for all epochs } e \tag{3.4}$$

i.e trough out the layers we can only lose or maintain information about the input distribution X and label distribution Y, equality is achieved iff transformation functions f_θ^t are invertible.

3.1.1 Viability of Compression

Let us consider the value $I(X, T_{e,t})$, in order to generate the probability distribution X we have assumed that it is uniform, i.e

$$P(X = x_i) = 1/N \text{ for all } i = 1...N \implies H(X) = log_2(N)$$

we also know the probability distribution of $T_{e,t}$ given X, i.e

$$P(T_{e,t} = t|X = x) = \begin{cases} 1, & \text{if } t = F_{\theta(e)}^t(x), \\ 0, & \text{otherwise.} \end{cases}$$

This implies if we have observed the value of X there is no uncertainty of the value of $T_{e,t}$, hence

$$H(T_{e,t}|X) = 0$$

this implies

$$I(X, T_{e,t}) = H(T_{e,t}) - H(T_{e,t}|X) = H(T_{e,t})$$

Consider now $H(T_{e,t})$,

$$\begin{aligned}
&\text{we know from Equation 2.5} \\
&H(T_{e,t}) - H(T_{e,t}|X) = H(X) - H(X|T_{e,t}) \\
&\implies H(T_{e,t}) = H(X) - H(X|T_{e,t}) \\
&\implies H(T_{e,t}) + H(X|T_{e,t}) = H(X) \\
&\implies H(T_{e,t}) \leq H(X) \qquad (3.5)
\end{aligned}$$

where equality is achieved iff $H(X|T_{e,t}) = 0$, i.e

$$P(X = x|T_{e,t} = t) = \begin{cases} 1, & \text{if } t = F^t_{\theta(e)}(x). \\ 0, & \text{otherwise.} \end{cases}$$

this is the case when if $F^t_{\theta(e)}$ is invertible and every x_i generates a unique layer activation t.

Recall the definition

$$F^t_\theta(x) = f^t_\theta(f^{t-1}_\theta(...(f^1_\theta(x))))$$

this implies

$$F^t_\theta \text{ is invertible } \Leftrightarrow f^i_\theta \text{ is invertible for } i = 1...t$$

If equality is achieved in Equation 3.5 and F^t_θ is invertible that would imply that we have equalities in the Data Processing equations – Equation 3.3 and Equation 3.4, this would imply that every layer contains the same information content as the input and no compression can happen.

Let us consider a real world neural network. We have previously defined neural networks to be a sequence of parameterized functions $f^1_\theta, f^2_\theta, ..., f^N_\theta$. In actual neural networks f^t_θ is a matrix that when applied to an activation vector of layer t produces the activation vector of layer $t + 1$.

The matrix is parameterized by weights θ which are controlled by the SGD algorithm, at the start of the training process weights θ are assigned random values, meaning we have completely random matrices, during every iteration process of SGD algorithm the weight are chosen at random and tweaked.

A matrix M is invertible iff its determinant is not zero $det(M) \neq 0$, A matrix determinant can take values in the range $(-\infty, \infty)$, thus if we consider a random matrix M the probability that it takes any specific value is 0,

$$P(det(M) = x) = 0, \text{ for } x \in \mathbb{R}$$

specifically $P(det(M) \neq 0) = 1$, hence $det(M) \neq 0$ and every random matrix M is invertible.

Given that every random matrix is invertible and that SGD generates random matrices we might conclude that no compression is possible, however this issue is resolvable if we consider weights θ to be a random distribution rather than an instance of a concrete value.

Let us define $\theta'(e)$ to be a probability distribution defined by Figure 3.1 (3.6)

Let us define $\theta(e)$ to be an instance of $\theta'(e)$ for any epoch e (3.7)

```
def random_θ'(epoch):
  if epoch == 1:
    pick θ ∼ multinomialGaussian(dimension = d)
    return θ
  old_θ = random_θ(epoch - 1)
  Let θ = One update step of SGD applied to old_θ
  return θ
```

Figure 3.1: Probability distribution of $\theta'(e)$, where e is an epoch

The notation defined in Equation 3.6 and in Equation 3.7 implies that $F^t_\theta(x)$ is a value (as before), however $F^t_{\theta'}(x)$ is a probability distribution. This is due to the fact that $F(x)$ represent a layer activation for a neural network.

- If the networks weights are deterministic i.e θ, then the whole neural network is deterministic which implies that F^t_θ is a deterministic function and $F^t_\theta(x)$ is a value.
- If however the weights form a probability distribution i.e θ', then the whole neural network is probabilistic, as such $F^t_{\theta'}(x)$ does not return an exact value, but a distribution of possible values. Probability distribution $F^t_{\theta'}(x)$ is defined by Figure 3.2

```
def random_F^t_θ'(θ',t,x):
  pick θ ∼ θ'
  return F^t_θ(x)
```

Figure 3.2: Probability distribution of $F^t_{\theta'}(x)$, where θ' – is probability distribution of the weights, t – is the layer number, x – is the input to the neural network

If we assume the neural network weights θ' to be a probability distribution, then $F^t_{\theta'}(x)$ is not generally invertible. In order for $F^t_{\theta'}$ to still be invertible the Equation 3.8 has to be satisfied.

$$\forall i, j \in \{1, ..., N\}.P(F^t_{\theta'}(x_i) = t) > 0 \wedge P(F^t_{\theta'}(x_j) = t) > 0 \Leftrightarrow i = j \tag{3.8}$$

If the equation is satisfied it would mean that for every x in the input data set the generated probability distribution is disjoint.

The degree to what the probability distributions $F^t_{\theta'}(x_i), i = 1, ..., N$ overlap signify the degree of compression.

3.2 Mutual Information Estimation

As stated before we are trying to validate the fact that compression is happening within the hidden layers in a neural network as such having robust tools to measure mutual information is of utmost importance.

3.2.1 Mutual Information Definition

Mutual information is a measure of dependence between two variables X and Y, it quantifies the number of bits obtained about one when observing the other.

Suppose we have two random variables X and Y if we want to measure mutual information between the two of them we usually refer to one of the following equations:

Using the entropy of the distributions to infer the value

$$I(X,Y) = H(X) - H(X|Y) \tag{3.9}$$

$$I(X,Y) = H(X) + H(Y) - H(X,Y) \tag{3.10}$$

Or calculating the mutual information explicitly from probabilities, for discrete and continuous distributions respectively

$$I(X;Y) = \sum_{y \in \mathcal{Y}} \sum_{x \in \mathcal{X}} p(x,y) \log \left(\frac{p(x,y)}{p(x)\, p(y)} \right) \tag{3.11}$$

$$I(X;Y) = \int_{\mathcal{Y}} \int_{\mathcal{X}} p(x,y) \log \left(\frac{p(x,y)}{p(x)\, p(y)} \right) dx\, dy \tag{3.12}$$

3.2.2 Theoretically undefined

Calculating mutual information is mathematically defined for both continuous and discrete distributions by using one of the functions above. However since we are looking at neural networks we do not have the full distribution – only an empirical sample of the unknown joint distribution $P(X,Y)$. As such we cannot use conventional methods to calculate the mutual information and instead have to use tools to estimate it.

The field of estimating mutual information is relatively new as a result the tools that are available are quite primitive. There has been some papers published that use more advanced techniques to tackle the problem, we will talk about in subsection 3.3.4

3.3 Calculating Mutual Information

3.3.1 Experimental setup

```
Algorithm: Generate Information Plane Data
Input:
T = number of layers
X = training data
Y = label data
N = number of epochs
Output:
I_x(epoch, layer), # mutual information with X
I_y(epoch, layer)  # mutual information with Y
Algorithm:
NN = setup_neural_network(T, X, Y)
for e in 0..N:
  NN.run_SGD_once()
  for t in 0..T:
    data = []
    for x ∈ X:
      t̂ = NN.predict(x).layer_activation(t)
      data.append(t̂)
    I_x(e, t) = calculate_mutual_information(data, X)
    I_y(e, t) = calculate_mutual_information(data, Y)
```

Figure 3.3: The general algorithm for calculating mutual information inside a neural network.

- X input is an empirical sample of all possible inputs.
- Y label data. Every $x \in X$ has a corresponding label $y \in Y$.
- $T_{e,i}$ data for a specific epoch e and specific layer i in the neural network. The dataset is generated by feeding every $x \in X$ trough the neural network and recording the activations.

3.3.2 Discrete method

The method used by Tishby in his paper. The method estimates mutual information between the X or Y and the hidden layer T by assuming the observed empirical distribution of input samples is the true distribution. We use this assumption to compute entropies of T and varying subsets as outlined in Equation 3.13

$$I(X,Y) = H(X) - H(X|Y) = H(X) - \sum_{y \in Y} H(X|Y=y)p(Y=y) \tag{3.13}$$

However just calculating mutual information for a discrete distribution is not enough as this yields mutual information equal to 0, in order to sidestep this issue we need to simulate randomness Tishby achieves this by grouping multiple values together, which he calls binning. A more detailed explanation why this is done is given in subsection 4.0.1

Figure 3.4 trough to Figure 3.6 outline the full algorithm.

```
1  Algorithm: Mutual Information
2  Input:
3  X = x_1, x_2, ..., x_n
4  Y = y_1, y_2, ..., y_n
5  Output: I(X : Y)
6  X = Bin close values of X together
7  Y = Bin close values of Y together
8  H(X) = Calculate entropy of X
9  H(X|Y) = Calculate conditional entropy of X given Y
10 I = H(X) − H(X|Y)
```

Figure 3.4: Pseudo code for computing mutual information refer to Figure 3.5 and Figure 3.6 for entropy computation.

```
1  Algorithm: Entropy - Discrete Method
2  Input: X = x_1, x_2, ..., x_n
3  Output: H(X)
4  for x̂ ∈ Unique(X):
5    count = 0
6    for x ∈ X:
7      if x̂ = x:
8        count = count + 1
9    P_x = count / len(X)
10 H(X) = −∑_{x∈Unique(X)} P_x log(P_x)
```

Figure 3.5: Algorithm for computing entropy – Discrete method

```
Algorithm: Conditional Entropy
Input:
X = x_1, x_2, ..., x_n
Y = y_1, y_2, ..., y_n
Output: H(X|Y)
H(X|Y) = H(X)
for ŷ ∈ Unique(Y):
  X̂ = []
  for x, y ∈ zip(X, Y):
    if ŷ = y:
      X̂.append(x)
    H(X|Y) = H(X|Y) − H(X̂)
```

Figure 3.6: Algorithm for computing conditional entropy

When computing mutual information between a hidden layer T and the input set X we can abuse the fact that every element $x \in X$ is unique and uniquely identifies an element $t \in T$ hence

$$H(T|X) = 0 \tag{3.14}$$

$$I(T, X) = H(T) - H(T|X) = H(T) \tag{3.15}$$

This does not affect the result but increases performance of our algorithm.

3.3.3 Kernel Density Estimation

The KDE method used in Saxe's paper but originally devised by Kolchinsky & Tracey (2017); Kolchinsky et al. (2017). As well as the Discrete method KDE assumes that the observed empirical distribution in the hidden layer T is the true distribution.

However, instead of binning values together KDE assumes the distribution is a mixture of Gaussians. Using this fact we can get an upper bound when calculating entropy for a collection of data points.

The algorithm closely follows figures :Figure 3.4, Figure 3.5, and Figure 3.6, however there is a change the way entropy is calculated so instead of Figure 3.5 we have Figure 3.7 below.

```
Algorithm: Entropy - KDE
Input:
X = x_1, x_2, ..., x_n
var = noise variance 0.05 by default
Output: H(X)
dists = compute distance matrix of X
dists = dists / 2*var # divide every distance by a value
# an x is an observation of a hidden layer so, hence it's a vector
# which has an associated dimension
dim = x_1.dimension
normconst = (dim / 2)*log(2*π*var)
lprobs = []
for row in dists:
  lprobs.append(log(sum(exp(-row))) - log(n) - normconst
H(X) = mean(lprobs) + (dim / 2)
```

Figure 3.7: Algorithm for computing entropy – Kernel Density Estimation method. The same algorithm as used by Saxe.

As before when computing mutual information between a hidden layer T and the input set X we can use the fact that every element $x \in X$ is unique and hence uniquely identifies an element in $t \in T$.

3.3.4 Advanced methods

Since mutual information estimation is a contentious part of the project we wanted to experiment with more advanced techniques however we were not able to adopt the methods for this project, the methods that we have tried are outlined below.

Mutual Information by Gaussian approximation

"Estimating Mutual Information by Local Gaussian Approximation" (Shuyang Gao). A promising way to estimate mutual information. However the implementation proved to be to difficult and time consuming so we abandoned it. I contacted out to the author but he was not able to provide any concrete code.

Geometrical Adaptive Entropy Estimation

"Breaking the Bandwidth Barrier: Geometric Adaptive Entropy Estimation" (Weihao Gao). We were pointed to this paper by S. Gao the author of the previous method, as previously this looked like a promising method to measure entropy and mutual information. Furthermore, the code was available online unfortunately we were not able to adapt the code for multidimensional values as a result we were getting wrong and inconsistent results. We decided making this method work would require too much time and is out of scope of this project.

3.4 Implementation Optimizations

Producing data for the information plane requires a substantial amount of computational power and memory, in order for the computation to complete in a reasonable amount of time we have to utilize all the available resources and minimize the amount of work we are doing.

3.4.1 Maximising Resource Utilization

The obvious way to maximise the resource usage is to parllelize the workload. If we refer to Figure 3.3, we can see two main ways we can do this.

The first way is to parallelize one of the two outer loops and run mutual information calculations in parallel, this is easy to do, however an issue occurs if we need a lot of memory since every instance of mutual information calculation manipulates the datasets creating copies, this is an issue for bigger datasets such as MNIST.

The second way is to parallelize the mutual information calculation itself, this is nice since we use minimal amounts of memory, however might be hard or impossible to implement as it depends on the method.

The second way is to parallelize the mutual information calculation itself, this method has a bonus that uses minimal amounts of memory improving performance for bigger datasets such as MNIST. However implementing this option might be hard as it heavily depends on the maths of the method, or impossible if the method has an inherently linear part to it. In our case KDE parallel performance is very good, however Tishby's Discrete method has some bottlenecks that I wasn't able to remove.

3.4.2 Minimising the Workload

Even when using all the systems resources the calculations take a very long time to complete as such we need to find a way to speed it up. If we consider an information plane graph for ex. Figure 1.1, we see that from epoch to epoch there is very little change that is occurring, skipping some epochs might be a good way to reduce workload while keeping the overall result unchanged. We have implemented a couple of ways to skip the epochs

Simple Skip

A very simple simple way to skip epochs calculates mutual information for every $n'th$ epoch.

It's quite effective and is fast to implement and easy to parallelize, however it yields subpar results. At the beginning of the training period during the fitting phase the step sizes are too big and yields gaps in data as there are big changes between consecutive epochs. Toward the end of the training period during the compression phase the step size becomes too small and we are wasting computation as the changes between epochs is negligible. The next two methods address this problem, but have their own drawbacks.

Delta Skip – Exact

The Exact Delta Skip method introduces a distance metric which measures mutual information distance between two epochs. Using the distance metric the method tries to skip as many epochs

as possible while still guaranteeing that the distance is at most delta δ, and backtracks when necessary.

The Algorithm starts by measuring every consecutive epoch ($skip = 1$) as at the start of the training epochs are far apart – distance between them is more than δ. When the distance become smaller there is no need to measure every epoch so we exponentially increase *skip* until the difference between consecutively measured epochs is larger than δ. At this point we run into the issue of backtracking since we made a guarantee that every measured epoch will be at most δ apart.

We can consider the backtracing problem as an array of unknown – let's call this array a section. Every unknown in a section corresponds to the saved state of the neural network, revealing the unknown is equivalent to computing mutual information for the epoch. In this context we can rephrase the problem of backtracking as finding a subset of the section such that the difference between every consecutive number is less than or equal δ.

Consider an example with $\delta = 2$. At the start all the values are unknown

?	?	?	?	?	?	?	?	?

We measure values at the start and the end of the section to get the range.

2	?	?	?	?	?	?	?	**14**

We continue to split the section and measure the middle element until the difference between consecutive elements is less than or equal to δ or there are no more elements left.

2	?	?	?	**10**	?	?	?	14

2	?	**6**	?	10	?	**12**	?	14

2	**5**	6	**8**	10	?	12	?	14

The algorithm stops at this stage as:

- The distances between 10, 12 and 14 are 2 and $\delta <= 2$ holds.
- There is no cell between 2 and 5, even though the distance is $3 > \delta$.

Every unknown has to save the state of the neural network, that means that every unknown contains $|X| * |nodes|$ float numbers. To put it into perspective if we use the MNIST dataset which has 240,000 samples into a small network of 128 nodes every unknown would contain 0.25GB of data. As such this method is unsuited for large datasets in that case Delta Skip Approximate is more suited for the job. A way to remedy the problem is instead of saving all the activations, just save the neural network weights and compute the activations just before calculating Mutual Information for the epoch

Parallelizing The entire single threaded algorithm is described in Figure 3.8 and Figure 3.9. However there are two main ways how we can parallelize the algorithm.

- Parallelize Backtracking – every time we need to backtrack we can launch two threads to do the computation, it's quite easy to implement however we need to be careful as we don't want to change the global *skip* value.
- Compute Multiple sections – consider a section as before. Once we compute the latest epoch of the array we can launch backtracking and the next section computation in parallel as separate threads. We need to wait until the latest epoch is computed before computing the next section as we might need to update the *skip* value.

```
1  Algorithm: DeltaSkipExact
2  Input:
3  prev = mutual information result of the previous epoch
4  curr = mutual information result of the current epoch
5  δ = user specified maximum "distance" between epochs
6  multiplier = how much to multiply skip by
7  Output:
8  Algorithm:
9  dist = Distance(prev, curr)
10 if dist > δ:
11   Backtrack(prev, curr) # Figure 3.9
12 else:
13   skip = skip*multiplier
```

Figure 3.8: Delta Skip Exact. The skip value is assumed to be global it specifies how many epochs to skip until we measure again

```
1  Algorithm: Backtrack
2  Input:
3  prev = mutual information result of the previous epoch
4  curr = mutual information result of the current epoch
5  δ = user specified maximum "distance" between epochs
6  Output:
7  if prev.epoch + 1 < curr.epoch:
8    mid_epoch = average(prev.epoch, curr.epoch)
9    mid = Calculate Mutual Information of mid_epoch
10   DeltaSkipExact(prev, mid, δ, 1)
11   DeltaSkipExact(mid, curr, δ, 1)
```

Figure 3.9: Backtrack Algorithm

Distance Metric Every epoch that we measure yields us with a vector of mutual information values, that is for every layer T we receive two values $I(X,T)$ and $I(Y,T)$. Given the information vectors for two epochs we need to find a reasonable way to measure distance between them.

The distance is used for the purposes of speeding up the computation and won't meaningfully impact the results. I've chosen to define the distance as the maximum shift between the two epochs refer to Equation 3.16 how to compute it or to Figure 3.10 for a graphical representation.

$$D = max_{t \in T}[max(I_e(X,t) - I_{\hat{e}}(X,t), I_e(Y,t) - I_{\hat{e}}(Y,t))] \tag{3.16}$$

The equation reduces the vector to a single value that is easy to compare and to track.

If we consider the Figure 3.10 we can see that the axis are different in scale this is due to input X and label Y having different entropy values $H(X) = 12$ and $H(Y) = 1$. As such we might wish to adjust Equation 3.16, and scale mutual information values by the entropy as in Equation 3.17.

$$D = max_{s \in \{X,Y\}} max_{t \in T}[(I_e(s,t) - I_{\hat{e}}(s,t))/H(s)] \tag{3.17}$$

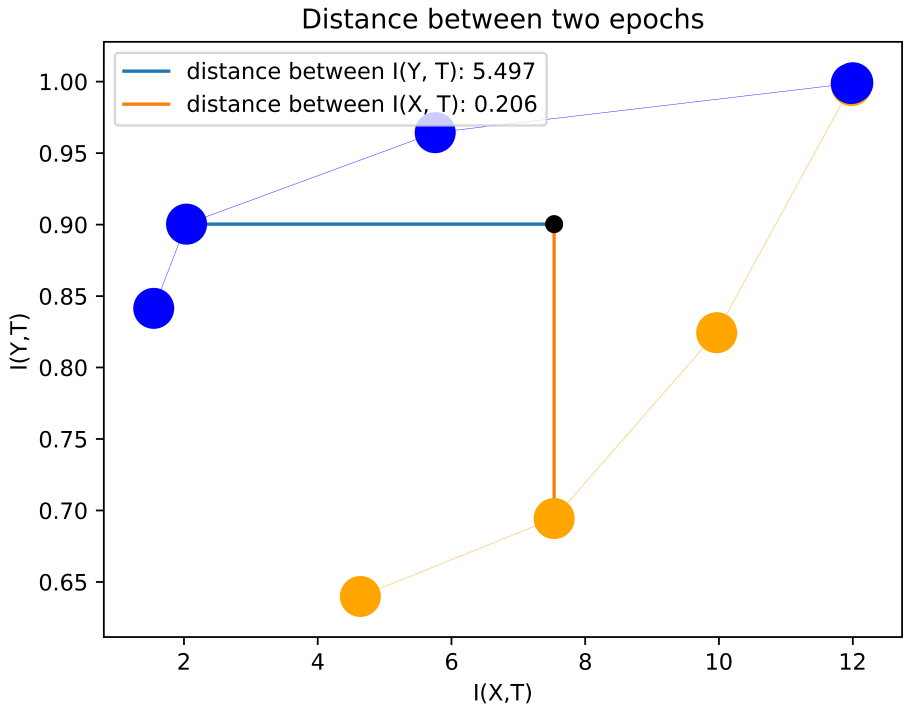


Figure 3.10: Example how distance between two epochs is measured.

Delta Skip – Approximate

The Exact method suffers the same problem as when we try to instantiate too many mutual information calculation instances, namely it runs out of memory if our dataset is too big. In order to solve this we use the approximate method which follows closely the Exact method.

The Algorithm – refer to Figure 3.11, as previously, in the Exact method it uses a distance metric and measures every $n'th$ epoch where n is adaptive and depends on how close the

epochs are. The critical difference between Exact and Approximate method happens when the distance between epochs is more than δ, the Exact method attempts to backtrack and fill in the gap whereas the Approximate doesn't backtrack and just continues to the next epoch, this is justified because the approximate method assumes that distance between epochs only shrinks and never increases.

This method does not suffer from memory issues but cannot be parallelized as in order to compute next epoch we need to know the distance between current epoch and the previous one. Delta Approximate method performs best when paired with highly parallel Mutual Information Estimator.

```
Algorithm: Delta Skip - Approximate
Input:
prev = mutual information result of the previous epoch
curr = mutual information result of the current epoch
δ = user specified estimated "distance" between epochs
Output:
Algorithm:
dist = Distance(prev, curr)
if dist > δ:
  skip = skip
else:
  skip = skip*2
```

Figure 3.11: Delta Skip Approximate

3.5 As-If-Random Experiment

Tishby's paper relies heavily on the notion that weights of a neural network behave 'as if' they are random, however his and Saxe's experiments don't capture this idea as much as they could – refer to subsection 4.0.2.

An experiment that better captures the ideas of random is instead of

- Tishby and Saxe didn't capture randomness incredibly well
- We have a bettter way
- How we've done it, only measure compression at the end of the training period when the network is stable and only "brownian randomness happens"(refer paper what randomness).
- the problem is every x in X is unique, if we sample multiple epochs instead of only one we have more compresssion
- we rely on the property of the network that there is very little change at the end of the training period

- Achieves compression even with ReLu yay!!!!!!!!!

We believe that Tishby's experiments Saxe's experiments could be improved, with introducing a better notion of randomness. blah blah blah blah not finished.

3.6 Repository Structure

Chapter 4

Evaluation

4.0.1 Deterministic networks

There's a very real argument to be made against compression in neural networks. Consider a generic neural network we can think of it as a function that is a series of matrix transformation, where a matrix corresponds to weights of a specific layer. However these matrices are all random (at least at the start of training) and hence probability of them being invertible is 100%.

Knowing that every single matrix is invertible allows us to conclude that that neural network as a whole is an invertible function, which means no information is lost and compression is impossible.

4.0.2 Why Randomness is hard to capture

Chapter 5

Conclusion

Bibliography

[1] Naftali Tishby. Opening the black box of deep neural networks via information, 2017.

Appendix A

Project Proposal

Computer Science Tripos: Part II Project Proposal

Measuring mutual information within Neural networks

Andrius Grabauskas, ag939
Robinson College
Saturday 20th October, 2018

Project Originator:	Andrius Grabauskas	
Project Supervisor:	Dr. Damon Wischik	
Director of Studies:	Prof. Alan Mycroft	
Overseers:	Dr. Robert Mullins	Prof. Pietro Lio'

Introduction and Description of the Work

The goal of this project is to confirm or deny the results produced by Shwartz-ziv & Tishby in their paper "Opening the black box of Deep Neural Networks via Information"[1]

The paper tackles our understating of Deep Neural Networks (DNN's). As of yet there is no comprehensive theoretical understanding of how DNN's learn from data. The authors proposed to measure how information travels within the DNN's layers.

They found that training of neural networks can be split into to two distinct phases: memorization followed by the compression phase.

- memorization - each layer increases information about the input and the label
- compression - this is the generalization stage where each layer tries to forget details about the input while still increasing mutual information with the label thus improving performance of the DNN. This phase takes the wast majority of the training time.

They found that each layer in neural network tries to throw out unnecessary data from the input while preserving information about the output/label. As the network is trained each layer preserves more information about the label

[1] https://arxiv.org/abs/1703.00810

The results they found were interesting but also contentious as they have not yet provided a formal proof, just experimental data as a result there are many peers that are cautious and sceptical of the theory even a paper[2] was produced that tries to suggest that the theory is wrong, however this was dismissed by Tishby & Shwartz-Ziv[3]

Starting Point

I have watched a talk that Prof. Tishby gave on this topic at Yandex, no other preparation was done.

Resources Required

The training DNN's and measuring mutual information will be computationally expensive so I will be using Azure cloud GPU service to acquire the required compute for this project. The GPU credits will be provided by Damon Wischik

For backups I intend to store my work on GitHub and my own personal machine. In case my laptop breaks I will get another one or use the MCS machines.

Substance and Structure of the Project

The aim of this project to reproduce the results provided by Prof. Tishby and his colleagues. The intention of my work is to help settle the debate surrounding the topic either strengthening the arguments in favour of the theory in case my results are inline with the aforementioned results or encourage discussion in case my results contradict the theory.

My work will require me to have a comprehensive understanding of Information theory, Information bottleneck and neural networks.

One of the more contentious parts of my project will be measuring mutual information between the input a layer in the DNN and the label. It will be computationally expensive to measure it in DNN since we will need to retrain the network in order to get a distribution rather than a single value. I will use Gaussian approximation to measure it (relevant paper[4])

Will need to use Python to train the neural networks and GNUplot or alternative to plot the results.

Success Criteria

Reimplement the code that was used to generate the papers results. Confirm or deny the results produced in"Opening the black box of Deep Neural Networks via Information" paper on the same dataset as the paper. In order to do that I will need to: Train a neural network on the same dataset

[2]https://openreview.net/pdf?id=ry_WPG-A-
[3]https://openreview.net/forum?id=ry_WPG-A-&noteId=S1lBxcE1z
[4]https://arxiv.org/abs/1508.00536

that was used in the paper and measure mutual information between the layers. Analyse the results produced and address any discrepancies that may have occurred.

Extensions

Provided I achieve the success criteria there are two main ways to extend it.

- Use different datasets to test the theory. Using different datasets would confirm that the results are not data specific. Current datasets we are considering: MNIST[5] and NOT-MNIST[6].
- Explore different ways of measuring mutual information. One interesting way would be to explore a discrete neural network where every node would only be able assigned discrete values say 1...256. This would make the distribution within a DNN layer discrete and hence it would make calculating mutual information straightforward. However quantizing the neural network could possibly hurt the performance of the network.

[5]http://yann.lecun.com/exdb/mnist/
[6]https://www.kaggle.com/quanbk/notmnist

Schedule

- **20th Oct – 2nd Nov**

  I expect to spend the first two weeks reading up on Information theory (primarily from Mackay's book[7]) and the information bottleneck method in order to understand the nuances of the paper.

- **3rd Nov – 30th Nov**

  The following weeks I intend to spend reading up on DNN's doing some introductory courses, I will train the neural network on the same data as the paper but at this point will not yet try to measure the mutual information between the layers.

  At this point I will also start examining the code[8] provided and start to implement parts of it which don't deal with information measurement.

- **1st Dec – 28th Dec**

  Will start reading up on mutual Information measurement with local Gaussian approximation.

  Implementing mutual information measurement in code.

  At this point I expect the computation to be too demanding for my machine and will need to use provided compute.

- **29th Dec – 1st Feb**

  Having a working system to test data sets I will try to reproduce results from the paper on the same dataset. This will achieve my success criteria.

  At this point my success criteria should be completed I will spend some time writing the skeleton of the thesis. Look for any discrepancies between my results and the ones provided in the paper.

- **2nd Feb – 2nd Feb**

  Assuming everything goes as planned I will start looking into implementing one of the extensions. Which are :

  - Testing the theory on different datasets.
  - Implementing a quantized neural network implementation.

  or both, if time is in my favour.

- **3rd Feb – 2nd Mar**

  Will use the remaining time to write up the dissertation.

[7] Informaition Theory, Inference, and Learning Algorithms by David J. C. MacKay
[8] https://github.com/ravidziv/IDNNs