Andrius Grabauskas

Measuring mutual information in Neural Networks

Computer Science Tripos – Part II

Robinson College

Thursday 16th 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

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

The aim of the project was to reproduce Tishby's results, and see if they robust. Explore ideas that Tishby presented in his paper, such as:

- Compression in Neural Networks – if a Neural Network can lose information, if so how it happens what assumptions are made.
- Compression Phase in Neural Networks – Tishby claims that the reason why Neural Networks generalize is because they learn how to compress the input representation. It would be interesting to explore if his claims hold any water.

Work Completed

Reproduced Tishby's code and experiments. Extended the code to reproduce experiments conducted by Saxe. Extended the code to explore A Novel Mutual Information Estimation Idea. Produced code to plot the Information Plane as well as visualize the process in a video format.

Special Difficulties

- Getting Mutual Information Estimation to work was tricky and took a big chunk of time. Had to abandon some methods as they were too complex and required too much time.
- Wasn't able to conduct all the experiments I wanted – Such as testing different Datasets. The code supports such experiments, but I wasn't able to get them done in time.

[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`

Contents

List of Figures

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.

Chapter 2

Preparation

Neural Networks (NN) are an extremely successful tool, they are widely adopted commercially and closely studied academically, however even given the attention they have there is no comprehensive understanding of how these models generalize data and provide such impressive performance – in fact very little is known about how NN learn or about their inner workings. Recently Prof. Tishby [8] produced a paper claiming to understand the basic principles of how NN work. He decided to examine NNs through the information domain. Tishby made the claim that the incredible performance of NNs is due to their ability to compress information. Compressing data means the network is only able to keep relevant input features and it must discard the irrelevant bits of information, leading to 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. A paper release by Saxe [4] has contested the claim made by Tishby arguing that compression cannot happen in Neural Networks and the results are a consequence of the hyper parameters Tishby used. However, Saxe's paper suffers from the same problem as Tishby's as it does not provide a formal proof only experimental evidence – as such ti does not 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

2.1.1 Entropy

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 $H(Y) = 1.5$.

2.1.2 Conditional Entropy

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 $\{y_i, ..., y_m\}$. Equations 2.2 and 2.3 are two examples how we can compute conditional entropies – using explicit probabilities or entropies of or entropies of random variables conditioned on an event.

$$H(Y|X) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P(x_i, y_j) \log \frac{P(x_i, y_j)}{P(x_i)} \tag{2.2}$$

$$H(Y|X) = \sum_{i=1}^{n} P(x_i) H(Y|X = x_i), \tag{2.3}$$

$$\text{where } H(Y|X = x) = \sum_{i=1}^{m} P(y_i|X = x) \log P(y_i|X = x) \tag{2.4}$$

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:

$$\begin{aligned} H(Y|X) &= 0.5 \\ H(X|Y) &\approx 0.6887 \\ H(X) &= 1 \\ H(Y) &\approx 0.8112 \end{aligned} \tag{2.5}$$

2.1.3 Mutual Information

Mutual Information (MI) – measures how much information two random variables have in common. It quantifies information gained about one variable when observing the other. Equations 2.6 and 2.7 are two examples of how we can compute MI – using explicit probability computations or entropies of the random variables respectively, here X and Y are as previously defined.

$$I(X, Y) = \sum_{i=1}^{n}\sum_{j=1}^{n} P(x_i, y_j) \log \left(\frac{P(x_i, y_j)}{P(x_i)\, P(y_j)} \right) \tag{2.6}$$

$$I(X, Y) = H(X) - H(X|Y) \tag{2.7}$$

For example of MI 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 2.5, we will use them in Equation 2.7 to compute $I(X, Y)$.

$$I(X, Y) = H(X) - H(X|Y) \approx 1 - 0.6887 = 0.3113 \tag{2.8}$$

Properties of Mutual Information

There are some important properties of MI that we need to take note of. Let X and Y be any probability distributions then:

Commutative

$$I(X,Y) = I(Y,X) \tag{2.9}$$

Information Loss MI of two random variables cannot exceed entropy of either of them.

$$\begin{aligned} H(X) &\geq I(X,Y) \\ H(Y) &\geq I(X,Y) \end{aligned} \tag{2.10}$$

Data Processing Inequality Let u be some function; then,

$$I(X,Y) \geq I(X,u(Y)) \tag{2.11}$$

Invertible Transformation Let u be some invertible function; then,

$$I(X,Y) = I(X,u(Y)) \tag{2.12}$$

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 D defined as (x_i, y_i) for $i = 1, ..., N$. The prediction problem is finding a function f s.t Equation 2.13 is satisfied.

$$f(x_i) = y_i \text{ for } i = 1, ..., N \tag{2.13}$$

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

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 2.14 satisfies the problem.

$$g(x) = \begin{cases} True, & \text{if } \exists n \in \mathbb{N}.x = 2n, \\ False, & \text{otherwise.} \end{cases} \tag{2.14}$$

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.

2.2.3 Neural Networks

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

The structure of a NN consists of layers of nodes, where every consecutive layer is fully connected as in Figure 2.1a. When we try to predict a label of a specific input every node gets assigned 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 the $j'th$ node in layer l is to the $i'th$ node in layer $l+1$.

$$n_{l+1,i} = g(\sum_{j=0}^{\text{layer } l \text{ size}} w_{l,j,i} * n_{l,j}) \tag{2.15}$$

g in Equation 2.15 is called the activation function and is generally taken to be:

- Leaky ReLu:

$$g(x) = \begin{cases} x, & if x > 0 \\ \alpha x, & \text{otherwise.} \end{cases}$$

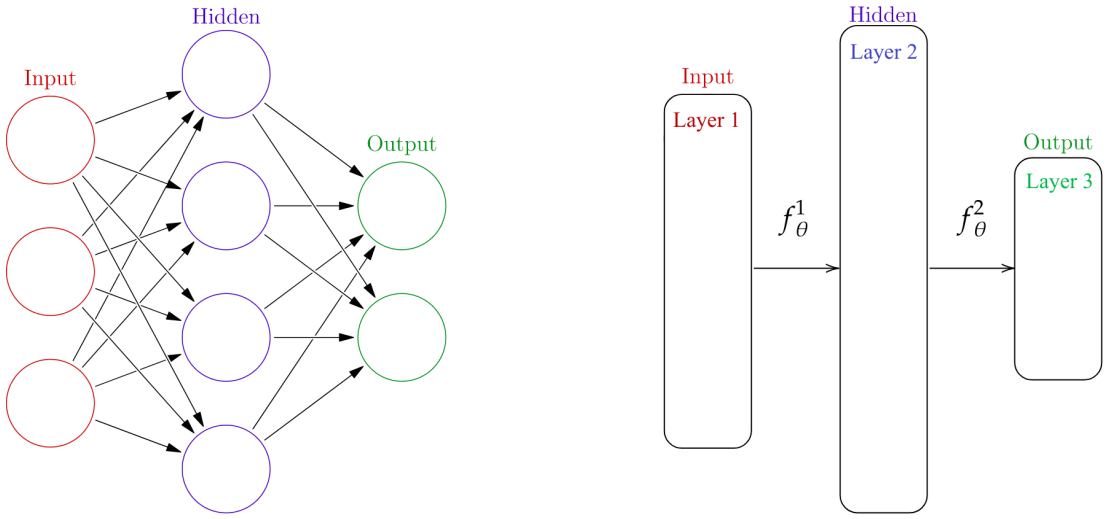
  α here is some small value such as $\alpha = 0.01$

- Sigmoid: $g(x) = \frac{1}{1+e^x}$
- Tanh: $g(x) = tanh(x)$

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

Stochastic Gradient Descent (SGD) The NNs training algorithm is called Stochastic Gradient Descent. SGD periodically adjusts the parameters w according to some goal such as minimising the prediction error.

SGD is an iterative process – it introduces a notion of epochs where one iteration of the algorithm advances the epochs by one. Notion of epochs implies that the parameters w depends on which epoch we are currently at – thus it must take the epoch number e as an argument – $w(e)$.



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

Figure 2.1: Source: Wikimedia Commons

2.2.4 Abstracting the Neural Network

For our purposes we will abstract away the individual nodes in the NN and only consider the layer structure – as in Figure 2.1b.

In our abstraction the NN 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. We will formally define a neural network with L layers to be a sequence of L functions $f^1_{\theta(e)}, f^2_{\theta(e)}, ..., f^L_{\theta(e)}$ that are parameterized by θ s.t. Equations 2.16 hold. In our abstract the parameters θ correspond to the parameters w, therefore θ depends on the current epoch – $\theta(e)$. Let us also define an **activation** of layer l to be output of the function f^l_θ.

$$\begin{aligned} \text{let } t_0 &= x, \\ t_0 \rightarrow f_\theta^1(t_0) &= t_1, \\ t_1 \rightarrow f_\theta^2(t_1) &= t_2, \\ &\dots \\ t_{L-1} \rightarrow f_\theta^L(t_{L-1}) &= t_L, \\ \text{let } \hat{y} &= t_L \end{aligned} \tag{2.16}$$

values $t_1, t_2, ..., t_L$ here are **activations** of layers $1, 2, ..., L$,
x is any input to the NN from the set $\{x_i, ..., x_N\}$,
$\hat{y}$ is the prediction of the NN for the input,x which may or not be correct label,
the arrow ' $\rightarrow$ ' signifies a transition from one NN layer to another.

We now have one function for each layer transition of the NN this allows us to extract the activation of a specific layer as in Equation 2.17.

$$t_l = f_\theta^l(f_\theta^{l-1}(...(f_\theta^1(x)))) \tag{2.17}$$
activtion of later l for input x

Let us define F_θ^t to be the activation of layer l given input x i.e

$$F_\theta^l(x) = f_\theta^l(f_\theta^{l-1}(...(f_\theta^1(x)))) \tag{2.18}$$

Figure 2.2 summarizes our NN definition.

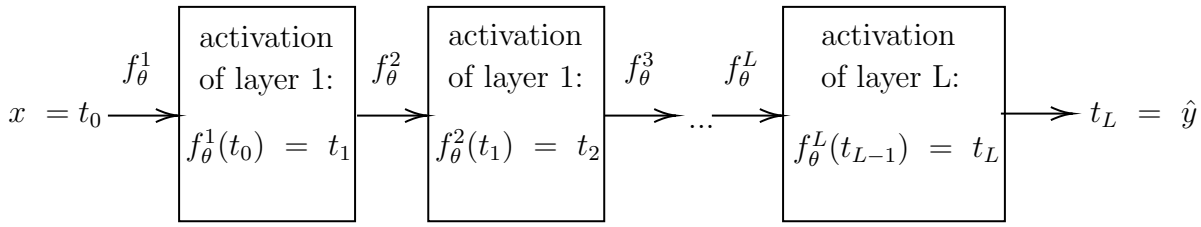


Figure 2.2: Visualization of a neural networks structure. x here is any input to the network from the set $\{x_i, ..., x_N\}$. $\hat{y}$ is the prediction of the network for the input x which may or may not be to the correct label. The values $t_1, ..., t_L$ here are **activations** of layers $1, ...,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

2.3 The Information plane

The information plane is a way of visualizing the Neural Network's training process through the information domain. Meaning we are looking what information the NN is retaining throughout the layers, and how the information retained changes throughout the training process.

We are interested see what information about the input and the label is retained – as having this information would help us understand how the neural network learns. Suppose we have this data available then we can examine this data with respect to:

- Epochs – lets us ask questions such as:
  - Is the network getting rid of noise in the network.
  - Does the network ever retain all the information about the label.
  - If the network does retain all the information about the label does training it after the point lead to performance increase.
- Layer – lets us ask questions such as:
  - Is there a difference between how much information the layers are discarding.
  - How does the information dynamics change if we add more layers.

Suppose we are examining a specific NN. At the start of the training process we expect the network to perform poorly – meaning it does not retain information about the label. Through the training process we expect:

- The information about the label to rise until it saturates. If we train past this saturation point we don't expect the information about the label to change much.
- The information about the input to rise until we have saturated the information about the label. At this point we either expect:
  - The information about the input to keep rising if we believe the network starts to learn noise about the data.
  - The information about the input to start reducing until the networks does not retain any noise about the input if we believe the network generalizes the data.

Ideally a neural network would retain all the information relevant to the label and none of the noise inherent in the input.

2.3.1 Setup

We want to compute what information about the label and the input the network is retaining – that is we want to compute the mutual information:

- Between the input distribution and all of the network layers,
- Between the label distribution and all of the network layers.

In order to compute the needed MI values we need to define the probability distributions:

- X - probability distribution of the input,
- Y - probability distribution of the label,
- $T_{e,l}$ - probability distribution of the layer l for the epoch e.

The routine in Figure 2.3 defines the correlated random variables: $X, T_{e,l}, Y$. The probability distributions define the needed MI values: $I(X, T_{e,l})$ and $I(Y, T_{e,l})$.

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

Figure 2.3: Definition of correlated random variables $X, T_{e,l}$ and, Y. The Probability distributions are generated from the dataset $D = \{(x_i, y_i)|i = 1, .., N\}$. $F^l_{\theta(e)}$ is defined by Equation 2.18

Input Probability Distribution The probability distribution X is generated from the input dataset D. Any input dataset D has an inherent probability distribution but most of the time we don't know it exactly. Figure 2.3 makes the assumption that D is uniformly distributed and thus takes every input value to be equally likely. If we had a specific dataset we could adjust our assumption and assign a different probability distribution to the input dataset.

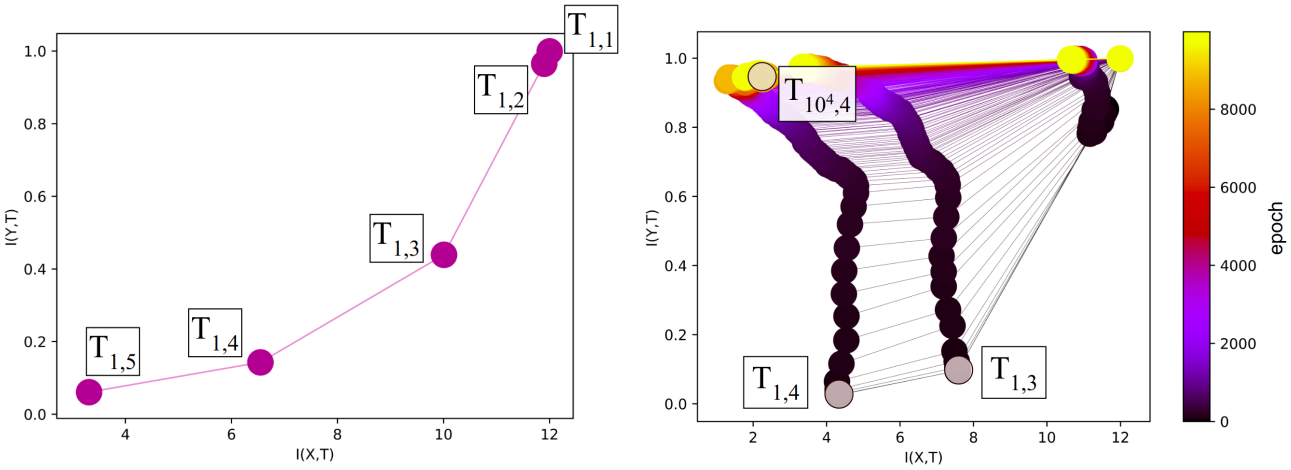
2.3.2 Visualization

The Information Plane visualizes how the information retained in the NN changes throughout the training period. In order to do this we need values $I(X, T_{e,l})$ and $I(Y, T_{e,l})$, for every epoch e and every layer l – we show how we obtain them in subsection 2.3.1.

To better understand how the Information Plane present the data let us consider **Figure 2.4a** – it shows the Information Plane for a NN for the first epoch before any training has been done. This implies that parameters $\theta(e)$ have not been affected by SGD and are thus random. If the network is random we would expect it not be able to retain much information. In the image we see that the network loses almost all information about the label $I(Y, T_{1,5}) \approx 0$, but retains some information about the input $I(X, T_{1,5}) \approx 3$ – we can see that this random NN only retains noise.

Let us now bring our attention to **Figure 2.4b** – it shows a NN that was trained for $\approx 10^4$. The NN was initialized with random parameters $\theta(1)$, then trained via the SGD algorithm which altered the parameters θ. The NN was trained so we expect to see a rise of mutual information with the label $I(Y, T_{e,l})$, throughout the training period – and we do see this in the Figure. However, we also see some other interesting features:

- Notice that the increase in $I(Y, T_{e,l})$ is very rapid at the start of the training period, but slows down considerably after ≈ 1000 epochs. The rapid increase might be the NN learning features that are highly correlated with the label, hence easy to learn.
- Notice how when increase in $I(Y, T_{e,l})$ slows down we start to see a decrease in the information about the input – $I(X, T_{e,l})$. This might be the point when the networks starts to remove the noise from the input.
- Notice how in both figures the NNs retain all the information about the label – This suggest that in order to achieve high performance retaining all the information is not enough. If all we needed to do is retain all the label information there would be no reason to add more layers to the neural network.



(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 2.4: The nodes in the figures correspond to information content of layers in the NN. The lines between the nodes help us distinguish individual epochs if more than one is plotted on the screen and lets us see the order of layers within a single epoch.
Notice that 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 2.4b – 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 $I(Y, T_{1,3}) \approx 0.1$
The Neural Networks in both figures have been trained on the same dataset as used by Tishby[8], The datasets input entropy, $H(X)$, is 12 and label entropy, $H(Y)$, is 1. Notice the x and y axis don't exceed the entropies $H(X)$ and $H(Y)$ respectively – this is due to the information loss property of MI Equation 2.10.

2.3.3 Interpretation of the Information Plane

Let us once again consider Figure 2.4b – we can see two general phases in the neural network. Tishby has named them *The Fitting Phase* and *The Compression Phase*.

These phases are observations made by Tishby and seem to be reproducible in by experiments. Even though Tishby gave no clear definition of how to identify the phases, they seem to follow the properties outlined bellow.

The Fitting Phase In Figure 2.4b 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 $I(Y, T_{e,l})$, 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 $I(X, T_{e,l})$, the information about the input, as we advance through the epochs e, in our case we see very little change in $I(X, T_{e,l})$.

Tishby's understanding of the Fitting phase is that the network tries to memorize the data and makes predictions based on the observations, this means that the network may learn features that only superficially correlate with the correct label.

The Compression Phase In Figure 2.4b 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 $I(Y, T_{e,l})$ is increasing with respect to epoch e.
- A slow decrease of $I(X, T_{e,l})$ with respect to epoch e.

Tishby's understanding of the Compression phase is that the network learns how to compress the representation of the input. This means the network has to discard features that do not help with predicting the correct labels. Discarding irrelevant features helps the neural network to generalize and produce better prediction for new data.

2.4 Testability

Software related to NN tends to be very compact as it heavily relies on external libraries that are extensively tested. In our case the most effective way to test the software is to provide tools that help examine the data. I have provided such tools in form of data visualization – such as a tool to generate a video of the training process, which lets a human detect anomalies.

2.5 Software Engineering

The project was build using the Spiral model. I needed to build a working version as fast as possible in order to verify the validity of Tishby's experiments. Once the Minimal Viable Product was build I added features as needed. Features such as different experiments, performance optimizations, visualization tools.

2.6 Starting Point

In order to understand the theory required I had learn the ideas presented in papers by Tishby[8][7] and Saxe[4]. I had to have a strong understanding of Neural Networks and Information Theory.

The starting for implementation was knowledge of *Python 3*. For this project I had to learn the Machine Learning Frameworks: *Keras* and *Tensorflow*.

Chapter 3

Implementation

3.1 The Compression Rebuttal

This section talks about if a NN can compress[1] data – i.e is it possible to lose information from layer to layer. There is an argument to be made for both sides. We can easily construct a NN that loses information. If we take the layer transition function f_θ^l to be

$$f_\theta^l(x) = 0$$

This function would lose information as we are not able to recover the input given the output, hence the NN must have discarded some information. Therefore compression is definitely possible. However, it is equally possible to create a function that preserves all the information about the input. Every layer in a NN is just a number $\mathbb{R}^{d_l}$,where d_l is the dimension of layer l. We know there is a one to one mapping between $\mathbb{R}$ and $\mathbb{R}^d$ for any dimension d. Therefore we can construct a neural network that loses no information if we take the layer transition function f_θ^l to be

$$f_\theta^l(x) = \text{map\_}\mathbb{R}\text{\_to\_}\mathbb{R}^{d_l}(\text{map\_}\mathbb{R}^{d_{l-1}}\text{\_to\_}\mathbb{R}(x))$$

That is mapping the value of previous layer to $\mathbb{R}$ and mapping it back to $\mathbb{R}^{d_l}$. It is important to understand if NN can compress data if we are to study how NN manipulate information.

3.1.1 Viability of compression

Viability of compression inside a NN. Asking if a NN compresses data is equivalent to asking if Equation 3.1 holds.

$$I(X, T_{e,l}) < H(X) \tag{3.1}$$

Lets examine the value $I(X, T_{e,l})$ more closely. From Equation 2.7 we know,

$$I(X, T_{e,l}) = H(X) - H(X|T_{e,l}) \tag{3.2}$$

Form Equation 3.1 and Equation 3.2 we have,

$$I(X, T_{e,l}) < H(X) \Leftrightarrow H(X|T_{e,l}) > 0 \tag{3.3}$$

Consider the value $H(X|T_{e,l})$. $H(X|T_{e,l})$ is 0 iff the value of $T_{e,l}$ uniquely identifies the value of X i.e

$$H(X|T_{e,l}) = 0 \Leftrightarrow P(X = x|T_{e,t} = t) = \begin{cases} 1, & \text{if } t = F_{\theta(e)}^l(x). \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

Equation 3.4 holds iff $F_\theta^l(x)$ is an invertible function. Recall the definition of $F_\theta^l(x)$

$$F_\theta^l(x) = f_\theta^l(f_\theta^{l-1}(...(f_\theta^1(x)))) \tag{3.5}$$

[1] N.B This section does not talk about the compression phase, just about compression in the network for a single epoch

Equation 3.5 implies

$$F_\theta^l \text{ is invertible } \Leftrightarrow \forall i \in \{1...l\}.f_\theta^i \text{ is invertible}$$

Putting it all together we get the bi implications

$$\begin{aligned} I(X, T_{e,l}) < H(X) &\Leftrightarrow \neg(\forall i \in \{1...l\}.f_\theta^i \text{ is invertible}) \\ &\Leftrightarrow \exists i \in \{1...l\}.f_\theta^i \text{ is not invertible} \\ &\Leftrightarrow \exists i \in \{1...l\} \exists u, v \in \{x_1, ..., x_N\}.f_\theta^t(u) = f_\theta^t(v) \wedge u \neq v \end{aligned} \tag{3.6}$$

and

This implies that compression can only happen if at least on transition function f_θ^l is not invertible.

3.1.2 Determinism of the Transition Function

Decomposing the Transition Function Lets break away from our NN abstraction and consider the transition function f_θ^l and how it is defined in concrete implementations of NNs – recall the Equation 2.15.

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

Where g is an invertible function such as Leaky ReLu, Sigmoid, Tanh. This means we can consider the function f_θ^l to be a composition of two different functions:

- A matrix multiplication – which is responsible for the weighted sum of previous layers activations $\sum_{j=0}^{\text{layer } l \text{ size}} w_{l,j,i} * n_{l,j}$. Let us call this matrix m_θ^l.
- An application of the invertible function g to every element of the vector produced by the matrix multiplication.

This function decomposition implies that

$$f_\theta^l \text{ is invertible} \Leftrightarrow m_\theta^l \text{ is invertible} \tag{3.7}$$

A Note on Invertible Matrices

Let M be a matrix and M^{-1} be the inverse, then;

$$M^{-1} \text{ is defined } \Leftrightarrow det(M) \neq 0 \tag{3.8}$$

Let M be a random matrix, then;

$$P(det(M) = 0) = 0 \tag{3.9}$$

Let M be random matrix, then from Equation 3.8 and Equation 3.9 we have

$$P(M^{-1} \text{ is defined}) = 1 \tag{3.10}$$

i.e every random matrix has an inverse.

Randomness in SGD Consider the matrix m_θ^l. The Matrix is defined by the parameters θ which are controlled by Stochastic Gradient Descent (SGD). At the start of the training period the parameters θ are initialized to random values. Every iteration of SGD we update the parameters – this update can also be considered random due to the Stochastic nature of the algorithm. From this we can treat the parameters θ and consequently the matrix m_θ^l as random throughout the training process.

Having m_θ^l be an instance of a random matrix would mean that Equation 3.10 holds – which would imply that every transition function is invertible, which in turn means that there cannot be any compression in a NN.

This would mean having discussion about Information in NN is moot. However, we can still have a meaningful discussion about neural networks if consider the parameters θ to be a random distribution rather than an instance of a concrete value. This is not agreed upon fully within the scientific community – Tishby assumes this is inherent in SGD, while Saxe has contested the claim.

Let us refer to the parameters as probability distribution with notation $\hat{\theta}$. $\hat{\theta}$ is a sequence of probability distributions that depend s.t $\hat{\theta}(e+1)$ depends on $\hat{\theta}(e)$, where e is the epoch.

Let us formally define $\hat{\theta}$. Since at the start of the training period SGD initializes parameters at random – let the start of the sequence $\hat{\theta}(1)$ be defined by Equation 3.11, where: $\mathcal{N}_k$ – is the Multivariate Normal distribution with dimension k, μ – is a vector in k'th dimension defining the mean of the distribution, Σ – is a $k * k$ matrix defining the variance of the distribution.

$$\hat{\theta}(1) \sim \mathcal{N}_k(\mu, \Sigma) \tag{3.11}$$

Let the sequence be defined as;

$$P(\hat{\theta}(e+1) = t | \hat{\theta}(e) = \hat{t}) = P(\phi = t), \tag{3.12}$$

$$\text{where } \phi \text{ is } \hat{\theta}(e) \text{ with one SGD iteration applied, we can assume } \phi \sim \mathcal{N}_k \tag{3.13}$$

We have shown that parameters of a NN can be though as being a probability distribution.

Impact of $\hat{\theta}$ – parameters as probability distributions With the idea that parameters $\hat{\theta}$ are a probability distribution let us again consider the question of compression in neural networks. From subsection 3.1.1 we know that no compression in NN is equivalent to

$$I(X, T_{e,l}) = H(X)$$

which is equivalent

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

$H(X|T_{e,l}) = 0$ means by observing $T_{e,l}$ we can deduce the exact value of X. i.e

$$\begin{aligned} &(\forall t.\ P(T_{e,l} = t) > 0 \implies P(X = x | T_{e,l} = t) = 1) \\ \implies &(\forall t.\ P(T_{e,l} = t) > 0 \implies P(T_{e,l} = t | X \neq x) = 0) \end{aligned} \tag{3.14}$$

Notice from Equation 3.11 and Equation 3.13 that

$$\forall e.\ \hat{\theta}(e) \sim \mathcal{N}_k, \text{ where } e \text{ is an epoch} \tag{3.15}$$

Equations 3.14 and 3.15 are unsatisfiable together. Proof: Consider x and $\hat{x}$ s.t $\hat{x} \neq x$. Let ϕ be s.t. $f^1_\phi(x) = t$

$$\begin{aligned} & f^1_\phi(x) = t, \\ \implies & P(T_{e,1} = t) > 0, \\ \implies & P(T_{e,1} = t | X \neq x) = 0, \text{ by Equation 3.14} \\ \implies & P(T_{e,1} = t | X = \hat{x}) = 0 \end{aligned} \tag{3.16}$$

however, we can construct $\hat{\phi}$ s.t $f_{\hat{\phi}^1}(\hat{x}) = t$

$$P(T_{e,1} = t | X = \hat{x}) = P(\hat{\theta}(e) = \hat{\phi}) > 0, \text{ by Equation 3.15} \tag{3.17}$$

hence, we get a contradiction

$$P(T_{e,1} = t | X = \hat{x}) = 0 \wedge P(T_{e,1} = t | X = \hat{x}) > 0 \tag{3.18}$$

and Equation 3.14 is unsatisfiable, this implies

$$H(X | T_{e,l}) > 0 \tag{3.19}$$

which finally implies

$$I(X, T_{e,l}) < H(X)$$

and proves that if we consider parameters to be random distributions the compression happens inside NN.

3.1.3 Recap

There is contention if NN are actually capable of compressing information. I have shown that if we assume the parameters of a NN to be concrete values compression is not possible and discussion about Information within them is moot. However, if we make the assumptions that parameters are actually random variables then compression does happen.

3.2 Measuring Mutual Information within NN

3.2.1 General Algorithm

We need a way to measure information content in NN throughout the training period – Figure 3.1 shows an algorithm that achieves this. It does this by explicitly running the SGD algorithm iteration by iteration and estimating information for every epoch and layer.

```
def informationNN(Data, Hyper, MIE):
# Measures how much information is retained in the NN. Aggregates the data
# over epochs and layers.
# Input:
# Data  - Training data D = {(x_i, y_i)|i = 1, .., N},
# Hyper - Hyper-parameters - refer to subsection 3.2.2.
# MIE   - Mutual Information Estimator - refer to subsection 3.2.3.
# Output:
# I_x(e, l) - Information content about the input in layer l and epoch e,
# I_y(e, l) - Information content about the label  in layer l and epoch e.
I_x, I_y = {}, {}
N = len(Data) # Number of data points
NN = Instanciate_Neural_Network(Hyper)
L = NN.layer_count()
for e in range(0,N):
  NN.run_SGD_once(Data)
  for l in range(0,L)
    data = []
    for x ∈ Data.X:
      l̂ = NN.layer(l).predict(x)
      data.append(l̂)
    I_x(e, l) = MIE.estimateX(data, Data.X) # Data.X = {x_1, ..., x_N}
    I_y(e, l) = MIE.estimateY(data, Data.Y) # Data.Y = {y_1, ..., y_N}
return I_x(e, l), I_y(e, l)
```

Figure 3.1: The general algorithm for calculating mutual information inside a neural network.

3.2.2 Hyperparameters

Figure 3.1 mentions Hyperparameters – these are parameters that do not change over the course of the training period and are usually set by Person conducting the experiment.

Hyperparameters can have a large impact on our results. It is important to vary our hyperparameters – by doing so we make sure that the results we see are robust and not just an epiphenomenon of our specific hyperparameters.

Examples of hyperparameters:

- Data Set Used – for example if we use Tishby's[1] or the MNIST[3] dataset.
- Size of the training set – how much of the dataset should we use for the training of the neural network.
- Activation Function of the Network – activation function between layers, examples given previously: Leaky ReLu, Sigmoid, Tanh.
- Batch size for SGD – At every iteration SGD only consider a subset of the dataset the size of this subset is determined by the Batch Size.

- Number of training Epochs – How long we train the neural network, can be considered a hyperparameter. However, in our case it is less important as we measure information for individual epochs.
- Network shape – Network shape refers to number of layers and dimension of every layer.
- We can also consider the MIE and their hyperparameters to be hyper parameter as it affects our results. As for example of MIEs hyper parameters:
  - Binning MIE subsection 3.2.4 – defines *bins*: which assigns activations of individual nodes to one of these *bins*.
  - Kernel Density Estimator MIE subsection 3.2.5 – defines *Noise variance*: which tunes the added Gaussian Noise
  - Batching MIE subsection 3.2.6 – defines *Batches*: which defines how many epochs will be aggregated together to perform Mutual Information Estimation. It also defines *bins* as in the Binning MIE.

3.2.3 Mutual Information Estimators

Recall subsection 2.3.1 – which talks about how we can measure information within a NN. We note that in order to measure MI we need random variables, however we only have the empirical dataset $D = \{(x_i, y_i)|i = 1, .., N\}$. We would like to use the random variables defined by the routine in Figure 2.3. However, in subsection 3.1.2 we prove that parameters θ have to be treated as random variables – a property which is missing from the current definition. Figure 3.2 updates the definition of our random variable to capture the missing property.

Figure 3.2 defines the correlated random variables: $X, T_{e,l}, Y$ – which specify the MI values: $I(X, T_{e,l})$ and $I(Y, T_{e,l})$. However, computing the MI values exactly is a computationally infeasible task – as we cannot capture the full randomness produced by `line 2`. As such we need a way to estimate the MI values. Fundamentally we need to make assumptions about the input distribution – which would allow us to efficiently compute the values. We will consider 3 different Mutual Information Estimators **(MIE)**, all of which simulate randomness in a different way: Binning, KDE, and AIR.

```
1 def rxty(θ̂, l):
2   Let φ be one update step of SGD on θ̂
3   pick i ∼ Uniform {1...N}
4   return (x_i, F^l_φ(x_i), y_i)
```

Figure 3.2: Updated definition of correlated random variables $X, T_{e,l}$ and, Y. The Probability distributions are generated from the dataset $D = \{(x_i, y_i)|i = 1, ..., N\}$. $F^l_{\theta(e)}$ is defined by Equation 2.18

A Note on The Input Dataset

We have previously defined the data set $D = \{(x_i, y_i)|i = 1, .., N\}$ – it is the sequence of pairs where x is the input and y is the label. This is the dataset available to us when we are training

the NN. However, often D is a subset of D_{global}, where D_{global} is all valid possible (x, y) pairs, which in most interesting cases is infinite. Take for example the MNIST dataset. The MNIST dataset is a collection of labeled handwritten digits – where every input x is an image and every label y is digit in the image. In this case D is a set containing about 70,000 (image, label) pairs and D_{global} is the set of all images containing a handwritten image and the labels associated with the image.

The task of a NN is to predict the labels given any $x \in D_{global}$, regardless if $x \in D$ holds or not. Hence, we are fundamentally interested in how much information is retained in the NN about the inputs from the dataset D_{global}. However, we can only make an attempt to measure information retained about inputs from the dataset D. Therefore, even if we were able to exactly compute MI values defined by Figure 3.2, they would still be only estimates.

Auxiliary functions

```
def getProbabilitiesOfData(A):
  # A = [a_1,...,a_N]
  # Unique(A) - returns A with duplicate elements removed
  for α in Unique(A):
    count = 0
    for a in A:
      if α == a:
        count = count + 1
    P_α = count / N
  return P
```

Figure 3.3: The routines defines a random variable $\hat{A}$ for any dataset A s.t. the equation $\forall a \in A.\ P(\hat{A} = a) = \frac{\text{Number of occurences of } a \text{ in } A}{\text{size of } A}$ holds.

3.2.4 MIE - Binning (Used by Tishby)

Simulating Randomness by Binning

Binning Suppose we have values $v, u \in \mathbb{R}^k$ the binning subdivides the space $\mathbb{R}^k$ into smaller sub-spaces and equates the two vales u and v if they both fall into the same smaller sub-space. This introduces randomness since if two input values yield similar layer activations that fall into the same sub-space we cannot distinguish them. Hence, given only the layer information we cannot uniquely identify the input.

Subspaces Let the hyperparameter s define the size of a subspace. Then the subspaces themselves are:

- for $k = 1$: intervals, of size s
- for $k = 2$: squares, of side length s
- for $k = 3$: cubes, of side length s

- for $k > 3$: natural extensions of prior.

Then we can rewrite v and u as:

$$v = (v_1, ..., v_k)s + \hat{e_v} \text{ and} \tag{3.20}$$
$$u = (u_1, ..., u_k)s + \hat{e_u}, \tag{3.21}$$

where $v_1, ..., v_k, u_1, ..., u_k \in \mathbb{N}$ and $\hat{e_u}, \hat{e_v} \in \mathbb{R}^k$ s.t $\forall i \in \{1...k\}$. $\hat{e}(i) < 1$. Thus, we can define the $=_{binning}$ equality operator to be

$$v =_{binning} u \Leftrightarrow \forall i \in \{1...k\}.\ v_i = u_i \tag{3.22}$$

Implementation When implementing binning instead of defining the hyperparameter s, the space subdivision is defined by three values: `low, high, bins`. Where `low,high`$\in \mathbb{R}$ and `bins`$\in \mathbb{N}$. The space in the interval `[low, high]` is subdivided into `bins` equal intervals, the algorithm then assumes that it will never see values smaller than `low` or larger than `high`. We are able to limit the space we subdivide because most of the activation functions have limited value ranges they can acquire. For example

$$-1 \leq tanh(x) \leq 1 \tag{3.23}$$
$$0 \leq sigmoid(x) \leq 1 \tag{3.24}$$

However, we need to make adjustments if we are using activation function with range that can take any value in $\mathbb{R}$, i.e Leaky ReLu. In order to fix this issue we have to redefine values `low` and `high` every SGD iteration. Consider again the general algorithm in Figure 3.1 – would have to insert code from Figure **??** in the loop defined `line 19`

```
1 low = min(low, NN.minNodeActivation.predict(x))
2 high = max(high, NN.maxNodeActivation.predict(x))
```

See Figure 3.5 for the implementation of binning. See Figure 3.4 for the definition of correlated random variables.

Estimating Mutual Information

MI with Input X Recall every X is unique so $H(X|T) = 0$, then by equation Equation 2.7

$$I(T_{e,l}, X) = H(T_{e,l}) \tag{3.25}$$

See Figure 3.6 routine `binning.estimateX`

MI with label Y We have not made any assumption about the label distribution, so by Equation 2.7

$$I(T_{e,l}, Y) = H(T_{e,l}) - H(T_{e,l}|Y) \tag{3.26}$$

See Figure 3.6 routine `binning.estimateY`

Entropy of Data The general algorithm provides us with a list of observations `T`. We need to calculate entropy $H(T)$ that is in, where T is the probability distribution associated with `T` and is inline with assumptions made by Figure 3.4.

$$P(T = t) = \frac{\texttt{count}(\texttt{T} =_{binning} t)}{\texttt{len(T)}}, \text{ where} \tag{3.27}$$

$\texttt{count}(\texttt{T} =_{binning} t)$ is number of elements in `T`, that are $=_{binning}$ equivalent to t, and `len(T)` is the number of observations

Using this definition we can calculate the entropy explicitly, by Equation 2.1

$$H(T_{e,l}) = -\sum_{t \in Unique(\texttt{T})} P(T_{e,l} = t) \log P(T_{e,l} = t) \tag{3.28}$$

See Figure 3.6 routine `binning.Entropy` for implementation details

Conditional Entropy of Data We have a sequence of paired observations `(t_i, y_i)` for `i=1,...,N`, let this dataset be `D`. Let T and Y be random variables associated with the datasets in order to calculate, then we can calculate the entropy as follows

$$H(T|Y) = -\sum_{y \in Unique(\texttt{Y})}^{N} P(Y = y) H(T|Y = y_i). \tag{3.29}$$

Here calculating $H(T|Y = y_i)$ is equivalent to calculating entropy of the random variable associated with the dataset $\texttt{T}_y$, where $\texttt{T}_y$ is defined by

$$t \in \texttt{T}_y \Leftrightarrow (t, y) \in \texttt{D} \tag{3.30}$$

we know how to do this from the previous paragraph.

See Figure 3.6 routine `binning.conditionalEntropy` for implementation details

```
1 def rxty(e, l):
2   pick i ~ Uniform {1...N}
3   t = F^l_θ(e)(x_i)
4   t̂ = binVector(t) # refer to Figure 3.5
5   return (x_i, t̂, y_i)
```

Figure 3.4: Definition of correlated random variables $X, T_{e,l}$ and, Y. Used by the Binning MIE.

```
def binVector(t):
  # applies binValue to every value of vector t
  # Input: t - a value in ℝ^d
  t̂ = [binValue(a) for a in t]
  return t̂

def binValue(t):
  # If we separate the range [low, high] into bins number of intervals
  # binValue returns the interval index that t would fall into.
  # Input: t - a value in ℝ, s.t. low ≤ t ≤ high
  # Globally defined values:
  # bins  - number of intervals to separate the range [low, high] into.
  # low   - lowest  expected value of t
  # high  - highest expected value of t
  s = (low − high)/bins
  return ⌊(t − low)/s⌋
```

Figure 3.5: Routines binVector and binValue are used by Figure 3.4 in order to simulate randomness.

The Algorithm

```
1  def binning.estimateX(T, X):
2    # Estimates mutual information between the observed states T and input X
3    # ∀i ∈ {0,..,N−1}. T[i] is the observed state for X[i].
4    # Since every x∈X is equally likely and unique H(T|X) = 0.
5    # hence, I(T,X) = H(T)
6    T̂ = binning.addNoise(T)
7    return binning.entropy(T)
8
9  def binning.estimateY(T, Y):
10   # Estimates mutual information between the observed states T and Y
11   T̂ = binning.addNoise(T)
12   H(T̂) = binning.entropy(T̂)
13   H(T̂|Y) = binning.conditionalEntropy(T̂, Y)
14   return H(T̂) - H(T̂|Y)
15
16 def binning.entropy(A):
17   # A = [a_1,...,a_N]
18   # Unique(A) - returns A with duplicate elements removed
19   P = getProbabilitiesOfData(A) # refer to Figure 3.3
20   H(A) = - ∑_{α∈Unique(A)} P_α log_2(P_α)
21   return H(A)
22
23 def binning.conditionalEntropy(A, B):
24   # Returns H(A|B)
25   # A = [a_1,...,a_N]
26   # B = [b_1,...,b_N]
27   P = getProbabilitiesOfData(B) # refer to Figure 3.3
28   H(A|B) = 0
29   for β in Unique(B)
30     Â = []
31     for a, b in zip(A, B) # zip(A, B) = [(a_1, b_1),...,(a_N, b_N)]
32       if b == β
33         Â.append(a)
34       H(A|B) += P_β*binning.entropy(Â)
35   return H(A|B)
36
37 def binning.addNoise(T):
38   T̂ = [binVector(t) for t in T] # refer to Figure 3.5
39   return T̂
```

Figure 3.6: Implementation of binning MIE. Functions `estimateX` and `estimateY` are called by Algorithm from Figure 3.1.

3.2.5 MIE - Kernel Density Estimation (Used by Saxe)

Simulating Randomness

In order to simulate random parameters $\hat{\theta}$ KDE assumes the inner layers $T_{e,l}$ to have Gaussian noise

$$T_{e,l} = f^l_{\theta(e)}(X) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}_{d_l}(0, \Sigma) \tag{3.31}$$

Σ here is the covariance matrix.

Discrete vs. Continuous Distributions

The added Gaussian noise implies that our random variable $T_{e,l}$ is a continuous. However, the method still defines X and Y are still defined to be discrete random variables. This means the MI values that we are interested $I(X, T_{e,l})$ and $I(Y, T_{e,l})$ – are between a continuous and a random variable.

Let V be a discrete random variable that takes values in $\{v_1, .., v_M\}$ and let U be a continuous distribution. Consider $I(V, U)$

$$\begin{aligned} I(V,U) &= H(U) - H(U|V) \\ &= H(U) - \sum_{i=1}^{M} P(v_i) H(U|V = v_i), \text{ Equation 2.3} \end{aligned} \tag{3.32}$$

Equation 3.32 shows that MI between a continuous an discrete random variable can be treated as a sum of differential entropies.

Differential Entropy – Entropy for a continuous random variable. Let U be a continuous random variable with probability density function p_U and domain $\tilde{U}$; then differential entropy is defined as

$$H(U) = -\int_{\tilde{U}} p_U(u) \log p_U(u)\, du \tag{3.33}$$

Estimating Mutual Information

We need to estimate MI values $I(X, T_{e,l})$ and $I(Y, T_{e,l})$

$$\begin{aligned} I(X, T_{e,l}) &= H(T_{e,l}) - H(T_{e,l}|X) \\ &= H(T_{e,l}) - \sum_{i=1}^{N} P(X = x_i) H(T_{e,l}|X = x_i) \\ &= H(T_{e,l}) - \frac{1}{N} \sum_{i=1}^{N} H(T_{e,l}|X = x_i) \end{aligned} \tag{3.34}$$

and

$$\begin{aligned} I(Y, T_{e,l}) &= H(T_{e,l}) - H(T_{e,l}|Y) \\ &= H(T_{e,l}) - \sum_{i=1}^{C} P(Y = y_i) H(T_{e,l}|Y = y_i) \end{aligned} \tag{3.35}$$

we cannot reduce Equation 3.35 further as the labels in the dataset might be imbalanced.

As in Equations 3.34 and 3.35 the implementation computes the values as a sum of differential entropies. See Figure 3.7 routines `KDE.estimateX` and `KDE.estimateY` for the implementation.

Deriving Conditional Entropy with input X Consider $H(T_{e,l}|X)$. Equation 3.31 implies

$$P(T_{e,l} > f^l_{\theta(e)}(x) + t|X = x) = P(\mathcal{N}_{d_l}(0, \Sigma) > t) \tag{3.36}$$

$$P(T_{e,l} > t|X = x) = P(\mathcal{N}_{d_l}(f^l_{\theta(e)}(x), \Sigma) > t) \tag{3.37}$$

Equation 3.37 implies that the random variables are identical. Hence, their entropies must be equal

$$H(T_{e,l}|X = x) = H(\mathcal{N}_{d_l}(f^l_{\theta(e)}(x), \Sigma)) \tag{3.38}$$

$$= H(\mathcal{N}_{d_l}(0, \Sigma)), \text{ as entropy is invariant to the mean.} \tag{3.39}$$

$$\tag{3.40}$$

Let us apply Equation 2.3 to $H(T_{e,l}|X)$

$$H(T_{e,l}|X) = \sum_{i=1}^{N} P(X = x_i)H(T|X = x_i) \tag{3.41}$$

$$= \sum_{i=1}^{N} \frac{1}{N} H(\mathcal{N}_{d_l}(0, \Sigma)) \tag{3.42}$$

$$= H(\mathcal{N}_{d_l}(0, \Sigma)) \tag{3.43}$$

Let *log* refer to the natural logarithm. We can analytically derive the differential entropy of a Gaussian to be

$$H(\mathcal{N}_{d_l}(0, \Sigma)) = \frac{d_l}{2log(2)}(log(2\pi) + log_e(det(\Sigma)) + 1) \tag{3.44}$$

In the implementation we have defined Σ to be the identity matrix I multiplied by some noise value $\sigma^2 \in \mathbb{R}$. This implies $det(\Sigma) = \sigma$ and hence Equation 3.44 can be simplified to

$$H(\mathcal{N}_{d_l}(0, \Sigma)) = \frac{d_l}{2log(2)}(log(2\pi\sigma) + 1) \tag{3.45}$$

Hence, from Equations 3.40 and 3.45 – Equation 3.46 holds

$$H(T_{e,l}|X) = \frac{d_l}{2log(2)}(log(2\pi\sigma) + 1) \tag{3.46}$$

Conditional Entropy with input Y In order to compute the conditional entropy $H(T_{e,l}|Y)$ we can use the same logic as in subsection 3.2.4 paragraph **Conditional Entropy of Data** and reduce our computation to a sum of differential entropies.

Deriving Entropy In how to compute upper and lower bounds for the differential entropies $H(T_{e,l})$ see Kolchinsky and Tracey[5] Section 4, and Kolchinsky and Tracey[6] Eq. 10. Or for the implementation see[2]

The Algorithm

```
def KDE.estimateX(T, X):
  # Estimates mutual information between the observed states T and input X
  # ∀i ∈ {0,..,N−1}. T[i] is the observed state for X[i].
  # Globally defined : σ - is the assumed noise of the observations T
  d = T[0].dimension # an observation t of T is a value in ℝ^d
  # KDE assumes H(T|X) is a Gaussian distribution
  # we can compute entropy of a Gaussian analytically as follows
  H(T|X) = (d/2)(log_e(2πσ) + 1)/log_e(2)
  H(T) = KDE.entropy(T)
  return H(T) - H(T|X)

def KDE.estimateY(T, Y):
  # Estimates mutual information between the observed states T and Y
  H(T) = KDE.entropy(T)
  H(T|Y) = KDE.conditionalEntropy(T, Y)
  return H(T) - H(T|Y)

def KDE.entropy(A):
  # See Kolchinsky and Tracey[5] Section 4, and Kolchinsky
  # and Tracey[6] Eq. 10
  # Code Implementation[2]

def KDE.conditionalEntropy(A, B)
  # Returns H(A|B)
  # A = [a_1,...,a_N]
  # B = [b_1,...,b_N]
  P = getProbabilitiesOfData(B) # refer to Figure 3.3
  H(A|B) = 0
  for β in Unique(B)
    Â = []
    for a, b in zip(A, B) # zip(A, B) = [(a_1, b_1),...,(a_N, b_N)]
      if b == β
        Â.append(a)
      H(A|B) += P_β*KDE.entropy(Â)
  return H(A|B)
```

Figure 3.7: Implementation of KDE MIE. Functions `estimateX` and `estimateY` are called by Algorithm from Figure 3.1.

3.2.6 MIE - Batching

Batching method proposes a way how to improve performance of already existing MIE. Recall that in order to achieve compression we are required to add noise to the layer activations t. Fundamentally Batching method proposes that by aggregating information about multiple epochs we can reduce the amount of noise we need to add in order to achieve compression.

Simulating Randomness

Let us assume that at the later stages of the training period changes to the probability distribution of parameters $\hat{\theta}$ are very small. That is we can consider parameter instances $\theta(i), \theta(j)$ to be instances of the same probability distribution if i and j are close enough. Let us use the assumption to define the Batching method to be formally defined by Figure 3.8, i.e we assume that the parameters θ in the epoch range $[e, e+b]$ are from the same underlying probability distribution.

However, sampling b consecutive epochs does not introduce enough randomness to introduce compression. Recall that our activations t are in the $\mathbb{R}^k$ space and generated by random matrices. That means that the Batch method picks a random matrix out of b and applies it to the input x. The Probability that any two matrices produce the same output $\hat{t}$ is zero. Hence, we need to apply another layer of noise to this method. For example to introduce extra noise we can apply the binning method or add Gaussian noise as in KDE. However, the noise applied can be reduced – in the binning method we could increase the number of sub-spaces or when applying the Gaussian noise we could use lower values noise.

```
1 def rxty(e, l, b):
2   pick i ~ Uniform {1...N}
3   pick j ~ Uniform {e...(e + b)}
4   return (x_i, F^l_{θ(j)}(x_i), y_i)
```

Figure 3.8: Definition of correlated random variables $X, T_{e,l}$ and, Y. Used by the Batching MIE.

Implementation

As this method requires to have access to information about multiple epochs at the same time. Hence, it is incompatible with the general algorithm defined in Figure 3.1. However, to not deviate from the format assume that routines `estimateX` and `estimateY` get the output for all the epochs aggregated to one i.e.

$$\texttt{T} = [t_1, ..., t_{N*b}] \tag{3.47}$$

$$\texttt{X} = [x_1, ..., x_{N*b}] \tag{3.48}$$

$$\texttt{Y} = [y_1, ..., y_{N*b}] \tag{3.49}$$

Since Batching method relies on adding extra noise, we cannot have unified algorithm that calculates entropy as it depends on the type of noise that gets added. Hence, we have to specify

different entropy calculations for different noise – in the implementation in Figure 3.9 we do this by having a global MIE parameter.

Notice that in our implementation methods `estimateX` and `estimateY` are identical. This is due to the fact that our dataset `X` no longer contains only unique elements and now has duplicates.

- In the Binning method we cannot assume that $H(T|X)$ is 0.
- In the KDE method we cannot assume that $H(T|X)$ is the entropy of a Gaussian.

The Algorithm

```
# MIE - here is a globally defined value that can be any mutual
# estimator for example: KDE or Binning.
def Batching.estimateX(T, X):
  T̂ = MIE.addNoise(T)
  return Batching.estimateMI(T̂, X)

def Batching.estimateY(T, Y):
  T̂ = MIE.addNoise(T)
  return Batching.estimateMI(T̂, X)

def Batching.estimateMI(T, XY):
    # T - must be the activation dataset, as there might be assumptions
    # made by the MIE
    # XY - either the input set X or the label set Y
    H(T) = MIE.entropy(T)
    H(T|XY) = MIE.conditionalEntropy(T, XY)
    return H(T) - H(T|XY)

# Need to define a function for KDE that does nothing, since the noise is
# added analytically.
def KDE.addNoise(T):
  return T
```

Figure 3.9: Implementation of Batching MIE. With the assumption that the added noise is

3.2.7 Advanced methods

I also took a look at more advanced Mutual Information Estimation methods. However, I was not able to adopt these methods for my project. The methods we have looked into are outlined below.

Estimating Mutual Information by Local Gaussian Approximation Shuyang Gao 2016[9]. I was directed to this by my supervisor. The Method looked promising at first as the

paper suggested it had better performance than other Entropy Estimators and had Pseudocode in the paper. However, the implementation of the method proved to be to difficult to implement;

- The method had complicated Hyper-Parameters – that were not well explained in the paper,
- The pseudo code had confusing types that were difficult to understand,
- The method relied on computing Hessian matrices, and satisfying Wolfe inequalities. The paper did not explain either of these topics and I was not able to find much information on them,
- The method required to implement Gradient descent, which added more complexity to the algorithm.

Due to these hurdles I was not able to implement the pseudocode. I contacted the author of the paper, however he was not able to provide working code for us to adapt. The author however, pointed us to another paper discussed below.

Breaking the Bandwidth Barrier: Geometrical Adaptive Entropy Estimation Weihao Gao 2016[10] 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 our problem as a result we were getting wrong and inconsistent results such as negative mutual information. We decided making this method work would require too much time and is out of scope of this project.

3.3 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.

3.3.1 Maximising Resource Utilization

The obvious way to maximise the resource usage is to parllelize the workload. If we refer to **??**, 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 Binning method has some bottlenecks that I wasn't able to remove.

3.3.2 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 2.4b, 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 3.10 and Figure 3.11. 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 3.11
12 else:
13   skip = skip*multiplier
```

Figure 3.10: Delta Skip Exact. The skip value is assumed to be global it specifies how many epochs to skip until we measure again

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

Figure 3.11: 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 3.50 how to compute it or to Figure 3.12 for a graphical representation.

$$D = \max_{t \in T}[\max(I_e(X,t) - I_{\hat{e}}(X,t), I_e(Y,t) - I_{\hat{e}}(Y,t))] \tag{3.50}$$

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

If we consider the Figure 3.12 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 3.50, and scale mutual information values by the entropy as in Equation 3.51.

$$D = \max_{s \in \{X,Y\}} \max_{t \in T}[(I_e(s,t) - I_{\hat{e}}(s,t))/H(s)] \tag{3.51}$$

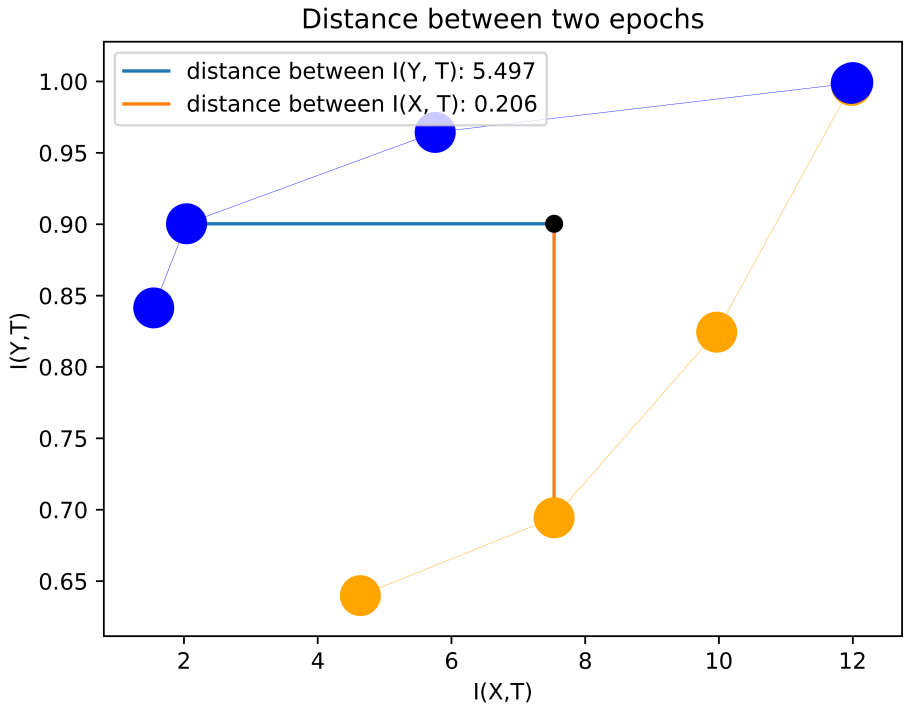


Figure 3.12: Example how distance between two epochs is measured.

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 3.13, 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 3.13: Delta Skip Approximate

3.4 Repository Structure

The code is hosted on github at **https://github.com/etherandrius/information-networks/releases/tag/2**.

The code has 2 main entry points

- `main_data.py` - is used to run experiments using Binning MIE or KDE MIE.

  run : `python3 main_data.py --help`, for help

- `main_as_if_random.py` - is used to run the Batching MIE with added Noise from the Binning MIE.

  run : `python3 main_as_if_random.py --help`, for help

And two auxiliary entry points

- `main_plot.py` - is to produce the visualization of the Information Plane.

  run : `python3 main_plot.py --help`, for help

- `main_movie.py` - is used to make a video of the Information Plane to help analyse the data.

  run : `python3 main_movie.py --help`, for help

Chapter 4

Evaluation

4.1 Success Criteria

- Successfully reproduced Tishby's results as presented in his paper.
- Showed That Tishby's results are robust to changes in some hyperparameters

4.2 Extensions

- Extended the code to be able to reproduce results by Saxe.
- Reproduced some results Published by Saxe
- Extended the code to accommodate for different Datasets
- However, did not have the time to test if the results of Both Tishby and Saxe are robust to changes in datasets.
- Extended the code to be able to compare multiple MIE in real time on the same instance of NN.
- Implemented adaptive ways to skip MI computations in order to save on compute, Multi Threaded ones and Single Threaded ones, depending on how much memory is available.
- Implemented movie plotting which allows us to see training over time, and better analyse the data.

Tishby's Experiment Tishby[**?**] talks about a subset of the hyper parameters mentioned above.

Tishby varies the following

- Even the implementation of the mathematical ideas can be considered a hyper parameter – as it is unlikely that implementations follow the mathematical model exactly. This is the reason why independent verification is important and partly the reason of this thesis.
- Notes on comparing MIEs need to run on the same NN as they vary, need to run in real time as saving the whole network
- ¡++¿
- ¡++¿

- How to choose hyper parameters for binning, kde and batch. For the batching MIE, I have not investigated how we could choose a potential value of b such that the range parameters θ for epochs in the range$[e, ..., (e + b)]$ can be considered to be from the same probability distribution. And I am not aware of any method to choose the number of intervals in the Binning MIE or the noise size in KDE MIE.

4.3 Ending remarks

- Tishby and Saxe makes big claims about phases that a neural network is in NN. Such as "Is the compression phase an inherent part of NN and SGD algorithm" with out first agreeing on if compression can actually happen inside NNs.
- Furthermore, they are using very simple MIE.
- However they are using very simple MIE. In my opinion we need better tools before we can judge
- Success Criteria
- we have successfully reproduced the results showed by Tishby and Saxe. However there are reasons to not trust either of them as they have flaws with them.
  - Tishby – used only a toy dataset
  - Saxe – Changed allot of parameters at once made the claim that no compression phase happens
- We need better tools for MIE
  - cannot judge subtleties if something has a compression phase our MIE are not trustesd
  - we have seen KDE and Discrete show inconsistent results, when n'th layers hass less information about the input than the n+1'th layer.
- compression phase
  - judging if the networks have a compression phase is moot point as of yet as tools for measuring information are not good enough. Case and point Saxe argues that there cannot be compression in NN but every every experiments show that compression exists.
  - Saxe states that there is no compression in NN, however their experiments disagree.
  - we don't have the tools to say if the compression phase is actually happening
  - Tishby says compression phase happens but he is using a toy dataset, which was shown to not have a compression phase by Saxe.
- performance

Chapter 5

Conclusion

- Success
- A number unforeseen extensions completed
- Learned more about compression in NN.
- Devised a method that should better estimate how NN retain MI.

5.1 Looking Back

5.2 Further Work

Bibliography

[1] Dataset used by tishby. https://github.com/ravidziv/IDNNs/tree/master/data. Accessed : 14/05/2019, Refer to Tishby and Schwarz-Ziv, 2017, section 3.1.

[2] Implementation of kde entropy function. https://github.com/artemyk/ibsgd/blob/iclr2018/kde.py. Accessed : 14/05/2019,.

[3] Mnist dataset. http://yann.lecun.com/exdb/mnist/. Accessed : 14/05/2019.

[4] Joel Dapello Madhu Advani Artemy Kolchinsky Brendan D. Tracey David D. Cox Andrew M.S̃axe, Yamini Bansal. On the information bottleneck theory of deep learning, 2018.

[5] Brendan D.T̃racey Artemy Kolchinsky.

[6] David H.W̃olpert Artemy Kolchinsky, Brendan D.T̃racey.

[7] Noga Zaslavsky Naftali Tishby. Deep learning and the information bottleneck principle, 2015.

[8] Naftali Tishby Ravid Schwarz-Ziv. Opening the black box of deep neural networks via information, 2017.

[9] Aram Galstyan Shuyang Gao, Greg Ver Steeg.

[10] Pramod Viswanath Weihao Gao, Sewoong Oh.

Appendix A

Project Proposal

Computer Science Tripos: Part II Project Proposal

Measuring mutual information within Neural networks

2359A
REDACTED College
Wednesday 15th May, 2019

Project Originator:	2359A	
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 – 15th Mar**

  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.

- **16th Mar – 12th Apr**

  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