0.1 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^l_θ to be

$$f^l_\theta(x) = 0$$

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^l_θ to be

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

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.

0.1.1 Viability of compression

Viability of compression inside a NN. Asking if a NN compresses data is equivalent to asking if Equation 1 holds.

$$I(X, T_{e,l}) < H(X) \tag{1}$$

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

$$I(X, T_{e,l}) = H(X) - H(X|T_{e,l}) \tag{2}$$

Form Equation 1 and Equation 2 we have,

$$I(X, T_{e,l}) < H(X) \Leftrightarrow H(X|T_{e,l}) > 0 \tag{3}$$

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

$$H(X|T_{e,l}) = 0 \Leftrightarrow P(X = x|T_{e,t} = t) = \begin{cases} 1, & \text{if } t = F^l_{\theta(e)}(x). \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Equation 4 holds iff $F^l_\theta(x)$ is an invertible function. Recall the definition of $F^l_\theta(x)$

$$F^l_\theta(x) = f^l_\theta(f^{l-1}_\theta(...(f^1_\theta(x)))) \tag{5}$$

Equation 5 implies

$$F^l_\theta \text{ is invertible } \Leftrightarrow \forall i \in \{1...l\}.f^i_\theta \text{ is invertible}$$

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

Putting it all together we get the bi implications

$$\begin{aligned} I(X, T_{e,l}) < H(X) &\Leftrightarrow \neg(\forall i \in \{1...l\}.f_\theta^i \text{ is invertible}) \\ &\Leftrightarrow \exists i \in \{1...l\}.f_\theta^i \text{ is not invertible} \\ &\Leftrightarrow \exists i \in \{1...l\} \exists u, v \in \{x_1, ..., x_N\}.f_\theta^t(u) = f_\theta^t(v) \wedge u \neq v \end{aligned} \tag{6}$$

and

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

0.1.2 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 **??**.

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

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

$$f_\theta^l \text{ is invertible} \Leftrightarrow m_\theta^l \text{ is invertible} \tag{7}$$

A Note on Invertible Matrices

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

$$M^{-1} \text{ is defined } \Leftrightarrow det(M) \neq 0 \tag{8}$$

Let M be a random matrix, then;

$$P(det(M) = 0) = 0 \tag{9}$$

Let M be random matrix, then from Equation 8 and Equation 9 we have

$$P(M^{-1} \text{ is defined}) = 1 \tag{10}$$

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 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 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.

$$\hat{\theta}(1) \sim \mathcal{N}_k(\mu, \Sigma) \tag{11}$$

Let the sequence be defined as;

$$P(\hat{\theta}(e+1) = t | \hat{\theta}(e) = \hat{t}) = P(\phi = t), \tag{12}$$

$$\text{where } \phi \text{ is } \hat{\theta}(e) \text{ with one SGD iteration applied, we can assume } \phi \sim \mathcal{N}_k \tag{13}$$

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 0.1.1 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

$$\begin{aligned} &(\forall t.\ P(T_{e,l} = t) > 0 \implies P(X = x | T_{e,l} = t) = 1) \\ \implies &(\forall t.\ P(T_{e,l} = t) > 0 \implies P(T_{e,l} = t | X \neq x) = 0) \end{aligned} \tag{14}$$

Notice from Equation 11 and Equation 13 that

$$\forall e.\ \hat{\theta}(e) \sim \mathcal{N}_k, \text{ where } e \text{ is an epoch} \tag{15}$$

Equations 14 and 15 are unsatisfiable together. Proof: Consider x and $\hat{x}$ s.t $\hat{x} \neq x$. Let ϕ be s.t. $f^1_\phi(x) = t$

$$\begin{aligned} & f^1_\phi(x) = t, \\ \implies & P(T_{e,1} = t) > 0, \\ \implies & P(T_{e,1} = t | X \neq x) = 0, \text{ by Equation 14} \\ \implies & P(T_{e,1} = t | X = \hat{x}) = 0 \end{aligned} \tag{16}$$

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

$$P(T_{e,1} = t | X = \hat{x}) = P(\hat{\theta}(e) = \hat{\phi}) > 0, \text{ by Equation 15} \tag{17}$$

hence, we get a contradiction

$$P(T_{e,1} = t | X = \hat{x}) = 0 \wedge P(T_{e,1} = t | X = \hat{x}) > 0 \tag{18}$$

and Equation 14 is unsatisfiable, this implies

$$H(X | T_{e,l}) > 0 \tag{19}$$

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.

0.1.3 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.

0.2 Measuring Mutual Information within NN

0.2.1 General Algorithm

We need a way to measure information content in NN throughout the training period – Figure 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 0.2.2.
# MIE   - Mutual Information Estimator - refer to subsection 0.2.3.
# 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 $x \in$ Data.X:
      $\hat{l}$ = NN.layer($l$).predict($x$)
      data.append($\hat{l}$)
    $I_x(e,l)$ = 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 1: The general algorithm for calculating mutual information inside a neural network.

0.2.2 Hyperparameters

Figure 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[?] or the MNIST[?] 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 subsection 0.2.4 – defines *bins*: which assigns activations of individual nodes to one of these *bins*.
  - Kernel Density Estimator subsection 0.2.5 – defines *Noise variance*: which tunes the added Gaussian Noise
  - As If Random subsection 0.2.7 – 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.

0.2.3 Mutual Information Estimators

Recall **??** – 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 **??**. However, in subsection 0.1.2 we prove that parameters θ have to be treated as random variables – a property which is missing from the current definition. Figure 2 updates the definition of our random variable to capture the missing property.

Figure 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($\hat{\theta}$, $l$):
2   Let $\phi$ be one update step of SGD on $\hat{\theta}$
3   pick i $\sim$ Uniform {1...N}
4   return $(x_i, F^l_\phi(x_i), y_i)$
```

Figure 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 **??**

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 2, they would still be only estimates.

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

Figure 3

0.2.4 MIE - Binning

The Assumptions

- assumes X is uniform
- simulates randomness by binning. What is binning?? takes in hyperparameter *bins*
- low and high are usually decided by the activation function Sigmoid 0 to 1, Tanh -1 to 1, as for Leaky ReLu we first need to aggregate we take it to be the minimum and maximum values for the observed layer activations
- Consider value $\hat{t}$ in Figure 4, it can only obtain values in the range $1, ..., bins$
- using binning to simulate randomness implies T can only take values in the range 1,...,bins – that is rather than being a continuous variable T is discrete.
- Which implies we can use the discrete entropy functions **??** and **??** to compute $H(T)$ $H(T|X)$. Which in turn lets use the MI function **??**.
- ¡++¿

```
def rxty($e$, $l$):
  pick i $\sim$ Uniform {1...N}
  $t$ = $F^{l}_{\theta(e)}(x_i)$
  $\hat{t}$ = batchVector($t$) # refer to Figure 5
  return $(x_i, \hat{t}, y_i)$
```

Figure 4

```
def batchVector($t$):
  # applies batchValue to every value of vector $t$
  # Input: t - a value in $\mathbb{R}^d$
  $\hat{t}$ = [batchValue(a) for a in $t$]
  return $\hat{t}$

def batchValue($t$):
  # If we separate the range $[low, high]$ into $bins$ number of intervals
  # batch returns the interval index that $t$ would fall into.
  # Input: $t$ - a value in $\mathbb{R}$, s.t. $low \leq t \leq high$
  # 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 $\lfloor (t - low)/s \rfloor$
```

Figure 5

The Algorithm

```
def binning.estimateX(T, X):
  # Estimates mutual information between the observed states T and input X
  # $\forall i \in \{0,..,N-1\}$. T[i] is the observed state for X[i].
  # Since every x∈X is equally likely and unique H(T|X) = 0.
  # hence, I(T,X) = H(T)
  $\hat{T}$ = binning.batchData(T)
  return binning.entropy($\hat{T}$)

def binning.estimateY(T, Y):
  # Estimates mutual information between the observed states T and Y
  $\hat{T}$ = binning.batchData(T)
  H($\hat{T}$) = binning.entropy($\hat{T}$)
  H($\hat{T}$|Y) = binning.conditionalEntropy($\hat{T}$, Y)
  return H($\hat{T}$) - H($\hat{T}$|Y)

def binning.entropy(A):
  # A = [a_1,...,a_N]
  # $Unique(A)$ - returns A with duplicate elements removed
  $P$ = getProbabilitiesOfData(A) # refer to $Figure\ 3$
  H(A) = - $\sum_{\alpha \in Unique(A)} P_\alpha log_2(P_\alpha)$
  return H(A)

def binning.conditionalEntropy(A, B):
  # Returns H(A|B)
  # A = [a_1,...,a_N]
  # B = [b_1,...,b_N]
  $P$ = getProbabilitiesOfData(B) # refer to $Figure\ 3$
  H(A|B) = 0
  for $\beta$ in $Unique(B)$
    $\hat{A}$ = []
    for a, b in zip(A, B) # zip(A, B) = [(a_1, b_1),...,(a_N, b_N)]
      if b == $\beta$
        $\hat{A}$.append(a)
    H(A|B) += $P_\beta$*binning.entropy($\hat{A}$)
  return H(A|B)

def binning.batchData(T):
  $\hat{T}$ = [batchVector(t) for t in T] # refer to $Figure\ 5$
  return $\hat{T}$
```

Figure 6

- By Tishby
- Binning Method
- Reimplemented
- Assumes input is uniformly distributed
- By treating grouping values together we are simulating overlapping values generated by different θ values. This is a very crude method to simulate randomness.
- two values that are very close together but are separated by a boundary are treated as if they have nothing in common.

0.2.5 MIE - Kernel Density Estimation

The Assumptions

Gaussian

- as well as the binning method – KDE as assumes that X is uniform.
- In this section let log refer to the natural logarithm
- KDE assumes T to have Gaussian noise.

  $$T_{e,l} = f^{l}_{\theta(e)}(X) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}_{d_l}(0, \Sigma) \tag{20}$$

  Σ here is the covariance matrix. In the implementation Σ is taken to be the identity matrix I multiplied by some σ. σ - is just a number in $\mathbb{R}$
- Consider $H(T|X)$ Under this assumption we get

  $$P(T_{e,l} = f^{l}_{\theta(e)}(x) + t|X = x) = P(\mathcal{N}_{d_l}(0, \Sigma) = t) \tag{21}$$

- Equation 21 implies that

  $$H(T_{e,l}|X) = H(\mathcal{N}_{d_l}(0, \Sigma)) \tag{22}$$

- The entropy of a Gaussian is analytically computable

  $$H(\mathcal{N}_{d_l}(0, \Sigma)) = \frac{d_l}{2log(2)}(log(2\pi) + log_e(det(\Sigma)) + 1) \tag{23}$$

- Let us assume that $\Sigma = \sigma I$, where $\sigma \in \mathbb{R}$ is value representing noise and I is the identity matrix. This implies $det(\Sigma) = \sigma$ and hence Equation 23 can be simplified to

  $$H(\mathcal{N}_{d_l}(0, \Sigma)) = \frac{d_l}{2log(2)}(log(2\pi\sigma) + 1) \tag{24}$$

- Hence, from Equations 22 and 24 – Equation 25 holds

  $$H(T_{e,l}|X) = \frac{d_l}{2log(2)}(log(2\pi\sigma) + 1) \tag{25}$$

0.2.6 Discrete vs Continuous Distributions

- KDE defines the random variable $T_{e,l}$ as a continuous distribution, while X and Y are still defined to be discrete.

- This means that we need to compute MI between between a continuous and a discrete distribution: $I(X, T_{e,l})$ and $I(Y, T_{e,l})$.

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

$$\begin{aligned} I(V, C) &= H(C) - H(C|V) && (26) \\ &= H(C) - \sum_{i=1}^{M} P(v_i) H(C|V = v_i) && (27) \end{aligned}$$

The Algorithm

```
def KDE.estimateX(T, X):
  # Estimates mutual information between the observed states T and input X
  # $\forall i \in \{0,..,N-1\}$. T[i] is the observed state for X[i].
  # Globally defined : $\sigma$ - is the assumed noise of the observations T
  $d$ = T[0].dimension # an observation t of T is a value in $\mathbb{R}^d$
  # KDE assumes H(T|X) is a Gaussian distribution
  # we can compute entropy of a Gaussian analytically as follows
  H(T|X) = $\frac{d}{2}(log_e(2\pi\sigma)+1)/log_e(2)$
  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[?] Section 4, and Kolchinsky
  # and Tracey[?] Eq. 10
  # Code Implementation[?]

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
  H(A|B) = 0
  for $\beta$ in $Unique(B)$
    $\hat{A}$ = []
    for a, b in zip(A, B) # zip(A, B) = [(a_1, b_1),...,(a_N, b_N)]
      if b == $\beta$
        $\hat{A}$.append(a)
      H(A|B) += $P_\beta$*KDE.entropy($\hat{A}$)
  return H(A|B)
```

Figure 7

0.2.7 MIE - As If Random

The Assumptions

The Algorithm

—————————————-

0.2.8 AIR

- I, novel
- Tishby's method is not ideal
- consider the perfect method teh new rxty
- hopefully a better way to measure mutual information, more computationally expensive. Only applicable for later epochs.
- AIR
- 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.

—————————————

0.2.9 Binning 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 28

$$I(X,Y) = H(X) - H(X|Y) = H(X) - \sum_{y \in Y} H(X|Y=y)p(Y=y) \tag{28}$$

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 **??**

Figure 8 trough to Figure 10 outline the full algorithm.

```
Algorithm: Mutual Information
```

Figure 8: Pseudo code for computing mutual information refer to Figure 9 and Figure 10 for entropy computation.

```
Algorithm: Entropy - Binning Method
```

Figure 9: Algorithm for computing entropy – Binning method

```
Algorithm: Conditional Entropy
```

Figure 10: 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

$$H(T|X) = 0 \tag{29}$$

$$I(T, X) = H(T) - H(T|X) = H(T) \tag{30}$$

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

0.2.10 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 Binning 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 8, Figure 9, and Figure 10, however there is a change the way entropy is calculated so instead of Figure 9 we have Figure 11 below.

```
Algorithm: Entropy - KDE
```

Figure 11: 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 $t \in T$.

0.2.11 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.

0.3 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.

0.3.1 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.

0.3.2 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. **??**, 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 12 and Figure 13. 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.

```
Algorithm: DeltaSkipExact
Input:
prev = mutual information result of the previous epoch
curr = mutual information result of the current epoch
$\delta$ = user specified maximum "distance" between epochs
multiplier = how much to multiply skip by
Output:
Algorithm:
dist = Distance(prev, curr)
if dist > $\delta$:
  Backtrack(prev, curr) # Figure 13
else:
  skip = skip*multiplier
```

Figure 12: 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
$\delta$ = 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, $\delta$, 1)
  DeltaSkipExact(mid, curr, $\delta$, 1)
```

Figure 13: 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 31 how to compute it or to Figure 14 for a graphical representation.

$$D = \max_{t \in T}[\max(I_e(X,t) - I_{\hat{e}}(X,t), I_e(Y,t) - I_{\hat{e}}(Y,t))] \tag{31}$$

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

If we consider the Figure 14 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 31, and scale mutual information values by the entropy as in Equation 32.

$$D = \max_{s \in \{X,Y\}} \max_{t \in T} [(I_e(s,t) - I_{\hat{e}}(s,t))/H(s)] \tag{32}$$

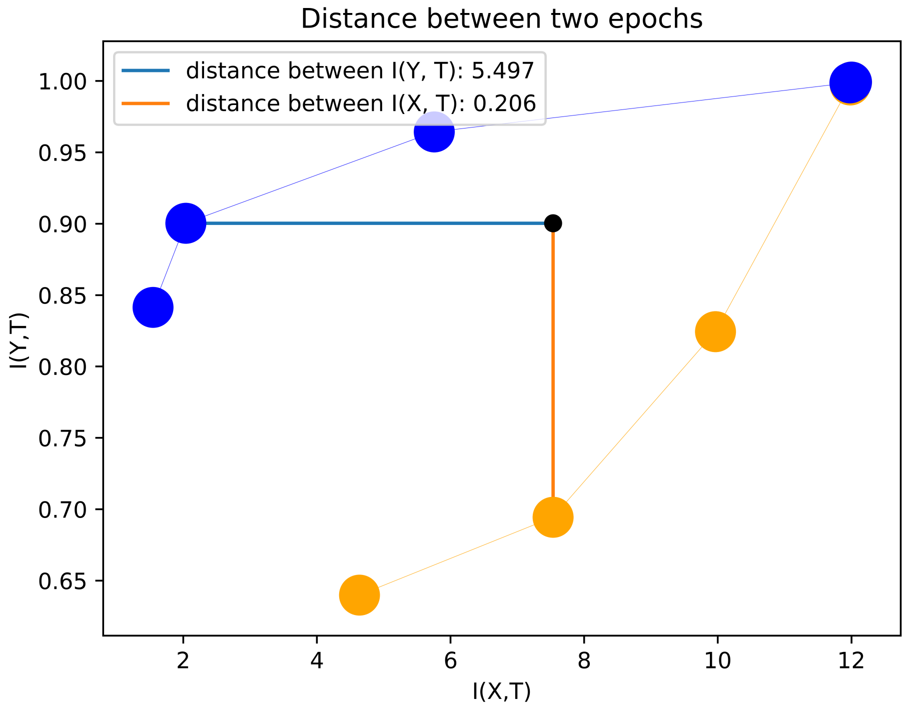

Figure 14: Example how distance between two epochs is measured.

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 15, 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 15: Delta Skip Approximate

0.4 Repository Structure