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_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.

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

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

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

$$F_\theta^l(x) = f_\theta^l(f_\theta^{l-1}(...(f_\theta^1(x)))) \tag{5}$$

Equation 5 implies

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

---

[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

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

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

$$f_\theta^l \text{ is invertible} \Leftrightarrow m_\theta^l \text{ is invertible} \quad (7)$$

A Note on Invertible Matrices

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

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

Let M be a random matrix, then;

$$P(det(M) = 0) = 0 \quad (9)$$

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

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

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

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

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. $f_\phi^1(x) = t$

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

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.

```
1   def informationNN(Data, Hyper, MIE):
2   # Measures how much information is retained in the NN. Aggregates the data
3   # over epochs and layers.
4   # Input:
5   # Data  - Training data D = {(x_i, y_i)|i = 1, .., N},
6   # Hyper - Hyper-parameters - refer to subsection 0.2.2.
7   # MIE   - Mutual Information Estimator - refer to subsection 0.2.3.
8   # Output:
9   # I_x(e, l) - Information content about the input in layer l and epoch e,
10  # I_y(e, l) - Information content about the label  in layer l and epoch e.
11  I_x, I_y = {}, {}
12  N = len(Data) # Number of data points
13  NN = Instanciate_Neural_Network(Hyper)
14  L = NN.layer_count()
15  for e in range(0,N):
16    NN.run_SGD_once(Data)
17    for l in range(0,L)
18      data = []
19      for x ∈ Data.X:
20        l̂ = NN.layer(l).predict(x)
21        data.append(l̂)
22      I_x(e, l) = MIE.estimateX(data, Data.X) # Data.X = {x_1, ..., x_N}
23      I_y(e, l) = MIE.estimateY(data, Data.Y) # Data.Y = {y_1, ..., y_N}
24  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 MIE subsection 0.2.4 – defines *bins*: which assigns activations of individual nodes to one of these *bins*.

  - Kernel Density Estimator MIE subsection 0.2.5 – defines *Noise variance*: which tunes the added Gaussian Noise

  - Batching MIE subsection 0.2.6 – 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.

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(θ̂, l):
2       Let φ be one update step of SGD on θ̂
3       pick i ∼ Uniform {1...N}
4       return (x_i, F_φ^l(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 $D = \{(x_i, y_i) | i = 1, ..., N\}$. $F_{\theta(e)}^l$ 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.

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: 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.

0.2.4   MIE - Binning

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:

$$v = (v_1, ..., v_k)s + \hat{e}_v \text{ and} \tag{20}$$

$$u = (u_1, ..., u_k)s + \hat{e}_u, \tag{21}$$

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

$$v =_{binning} u \Leftrightarrow \forall i \in \{1...k\}. \, v_i = u_i \tag{22}$$

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

$$-1 \leq tanh(x) \leq 1 \tag{23}$$

$$0 \leq sigmoid(x) \leq 1 \tag{24}$$

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 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 5 for the implementation of binning. See Figure 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 **??**

$$I(T_{e,l}, X) = H(T_{e,l}) \tag{25}$$

See Figure 6 routine binning.estimateX

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

$$I(T_{e,l}, Y) = H(T_{e,l}) - H(T_{e,l}|Y) \tag{26}$$

See Figure 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 4.

$$P(T = t) = \frac{\texttt{count}(\texttt{T} =_{binning} t)}{\texttt{len(T)}}, \text{ where} \tag{27}$$

$\texttt{count}(\texttt{T} =_{binning} t)$ 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 **??**

$$H(T_{e,l}) = - \sum_{t \in Unique(\texttt{T})} P(T_{e,l} = t) \log P(T_{e,l} = t) \tag{28}$$

See Figure 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

$$H(T|Y) = - \sum_{y \in Unique(\texttt{Y})}^{N} P(Y = y) H(T|Y = y_i). \tag{29}$$

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

$$t \in \texttt{T}_y \Leftrightarrow (t, y) \in \texttt{D} \tag{30}$$

we know how to do this from the previous paragraph.
    See Figure 6 routine binning.conditionalEntropy for implementation details

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

Figure 4: Definition of correlated random variables $X, T_{e,l}$ and, Y. Used by the Binning MIE.

```
1   def binVector(t):
2       # applies binValue to every value of vector t
3       # Input: t - a value in R^d
4       t̂ = [binValue(a) for a in t]
5       return t̂
6
7   def binValue(t):
8       # If we separate the range [low, high] into bins number of intervals
9       # binValue returns the interval index that t would fall into.
10      # Input: t - a value in R, s.t. low ≤ t ≤ high
11      # Globally defined values:
12      # bins  - number of intervals to separate the range [low, high] into.
13      # low   - lowest  expected value of t
14      # high  - highest expected value of t
15      s = (low − high)/bins
16      return ⌊(t − low)/s⌋
```

Figure 5: Routines binVector and binValue are used by Figure 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.binData(T)
7       return binning.entropy(T̂)
8
9   def binning.estimateY(T, Y):
10      # Estimates mutual information between the observed states T and Y
11      T̂ = binning.binData(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
20      H(A) = - ∑_{α∈Unique(A)} P_α log₂(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
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.binData(T):
38      T̂ = [binVector(t) for t in T] # refer to Figure 5
39      return T̂
```

Figure 6: Implementation of binning MIE. Functions estimateX and estimateY are called by Algorithm from Figure 1.

0.2.5 MIE - Kernel Density Estimation

Simulating Randomness

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

$$T_{e,l} = f^l_{\theta(e)}(X) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}_{d_l}(0, \Sigma) \tag{31}$$

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 $I(V, U)$

$$I(V, U) = H(U) - H(U|V)$$

$$= H(U) - \sum_{i=1}^{M} P(v_i)H(U|V = v_i), \text{ ??} \tag{32}$$

Equation 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

$$H(U) = - \int_{\tilde{U}} p_U(u) \log p_U(u)\, du \tag{33}$$

Estimating Mutual Information

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

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

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

$$= H(T_{e,l}) - \frac{1}{N} \sum_{i=1}^{N} H(T_{e,l}|X = x_i) \tag{34}$$

and

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

$$= H(T_{e,l}) - \sum_{i=1}^{C} P(Y = y_i)H(T_{e,l}|Y = y_i) \tag{35}$$

we cannot reduce Equation 35 further as the labels in the dataset might be imbalanced.

As in Equations 34 and 35 the implementation computes the values as a sum of differential entropies. See Figure 7 routines KDE.estimateX and KDE.estimateY for the implementation.

Deriving Conditional Entropy with input X   Consider $H(T_{e,l}|X)$. Equation 31 implies

$$P(T_{e,l} > f^l_{\theta(e)}(x) + t|X = x) = P(\mathcal{N}_{d_l}(0, \Sigma) > t) \tag{36}$$

$$P(T_{e,l} > t|X = x) = P(\mathcal{N}_{d_l}(f^l_{\theta(e)}(x), \Sigma) > t) \tag{37}$$

Equation 37 implies that the random variables are identical. Hence, their entropies must be equal

$$H(T_{e,l}|X = x) = H(\mathcal{N}_{d_l}(f^l_{\theta(e)}(x), \Sigma)) \tag{38}$$

$$= H(\mathcal{N}_{d_l}(0, \Sigma)), \text{ as entropy is invariant to the mean.} \tag{39}$$

$$\tag{40}$$

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

$$H(T_{e,l}|X) = \sum_{i=1}^{N} P(X = x_i)H(T|X = x_i) \tag{41}$$

$$= \sum_{i=1}^{N} \frac{1}{N} H(\mathcal{N}_{d_l}(0, \Sigma)) \tag{42}$$

$$= H(\mathcal{N}_{d_l}(0, \Sigma)) \tag{43}$$

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

$$H(\mathcal{N}_{d_l}(0, \Sigma)) = \frac{d_l}{2log(2)}(log(2\pi) + log_e(det(\Sigma)) + 1) \tag{44}$$

In the implementation we have defined Σ to be the identity matrix I multiplied by some noise value $\sigma \in \mathbb{R}$. This implies $det(\Sigma) = \sigma$ and hence Equation 44 can be simplified to

$$H(\mathcal{N}_{d_l}(0, \Sigma)) = \frac{d_l}{2log(2)}(log(2\pi\sigma) + 1) \tag{45}$$

Hence, from Equations 40 and 45 – Equation 46 holds

$$H(T_{e,l}|X) = \frac{d_l}{2log(2)}(log(2\pi\sigma) + 1) \tag{46}$$

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 0.2.4 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[**?**] Section 4, and Kolchinsky and Tracey[**?**] Eq. 10. Or for the implementation see[**?**]

The Algorithm

```
1   def KDE.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       # Globally defined : σ − is the assumed noise of the observations T
5       d = T[0].dimension # an observation t of T is a value in ℝ^d
6       # KDE assumes H(T|X) is a Gaussian distribution
7       # we can compute entropy of a Gaussian analytically as follows
8       H(T|X) = (d/2)(log_e(2πσ) + 1)/log_e(2)
9       H(T) = KDE.entropy(T)
10      return H(T) − H(T|X)
11
12  def KDE.estimateY(T, Y):
13      # Estimates mutual information between the observed states T and Y
14      H(T) = KDE.entropy(T)
15      H(T|Y) = KDE.conditionalEntropy(T, Y)
16      return H(T) − H(T|Y)
17
18  def KDE.entropy(A):
19      # See Kolchinsky and Tracey[?] Section 4, and Kolchinsky
20      # and Tracey[?] Eq. 10
21      # Code Implementation[?]
22
23  def KDE.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
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*KDE.entropy(Â)
35      return H(A|B)
```

Figure 7: Implementation of KDE MIE. Functions estimateX and estimateY are called by Algorithm from Figure 1.

0.2.6   MIE - Batching

Simulating Randomness

The hope is 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. Figure 8 defines random variables that captures these assumptions.

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

Figure 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 it is incompatible with the general algorithm defined in Figure 1. However, to no

- X is uniform

- want a better way estimate randomness

- In the later phases of training the NNs weights changes very little.

- The hypothesis is that aggregating multiple layers together can give a better estimation for MI as for a specific $P(T|X = x) = 1/batch$

- only trusted in the later epochs as in the early period of training there are allot of changes

The Algorithm

```
1  def Batching.estimateX(T, X):
2      # Estimates mutual information between the observed states T and input X
3      return H(T) - H(T|X)
4
5  def Batching.estimateY(T, Y):
6      # Estimates mutual information between the observed states T and Y
7      return H(T) - H(T|Y)
8
9  def Batching.entropy(A):
10
11  def Batching.conditionalEntropy(A, B)
12      # Returns H(A|B)
13      return H(A|B)
```

Figure 9: Implementation of Batching MIE. Functions estimateX and estimateY are called by Algorithm from Figure 1.

0.2.7 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 10 and Figure 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 11
12  else:
13    skip = skip*multiplier
```

Figure 10: Delta Skip Exact. The skip value is assumed to be global it specifies how many epochs to skip until we measure again

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

Figure 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 47 how to compute it or to Figure 12 for a graphical representation.

$$D = \max_{t \in T}[\max(I_e(X,t) - I_{\hat{e}}(X,t), I_e(Y,t) - I_{\hat{e}}(Y,t))] \tag{47}$$

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

If we consider the Figure 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 47, and scale mutual information values by the entropy as in Equation 48.

$$D = \max_{s \in \{X,Y\}} \max_{t \in T}[(I_e(s,t) - I_{\hat{e}}(s,t))/H(s)] \tag{48}$$
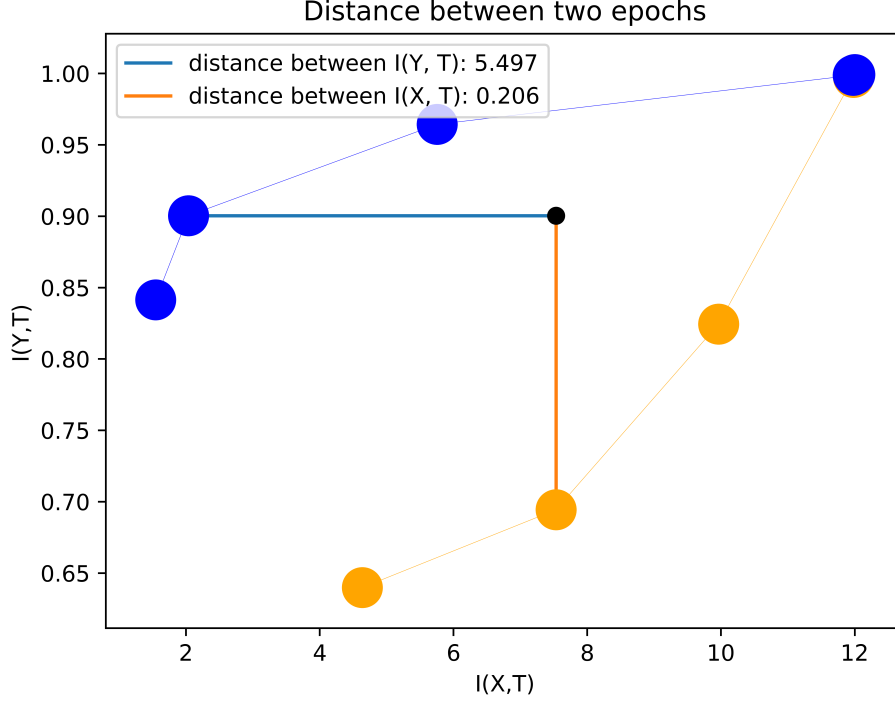
Figure 12: Example how distance between two epochs is measured.

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 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.

```
1   Algorithm: Delta Skip - Approximate
2   Input:
3   prev = mutual information result of the previous epoch
4   curr = mutual information result of the current epoch
5   δ = user specified estimated "distance" between epochs
6   Output:
7   Algorithm:
8   dist = Distance(prev, curr)
9   if dist > δ:
10    skip = skip
11  else:
12    skip = skip*2
```

Figure 13: Delta Skip Approximate

0.4   Repository Structure