0.1 Compression In Neural Networks

Mutual Information Let us introduce two properties of mutual information

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

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

$$I(A, B) = I(u(A), v(B)) \tag{1}$$

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

$$I(a, b) \geq I(a, c) \tag{2}$$

In the neural network case this implies

$$H(X) \geq I(X, T_{e,1}) \geq I(X, T_{e,2}) \geq ... \geq I(X, T_{e,N}) \text{ for all epochs } e, \text{ and} \tag{3}$$

$$I(X, Y) \geq I(T_{e,1}, Y) \geq I(T_{e,2}, Y) \geq ... \geq I(T_{e,N}, Y) \text{ for all epochs } e \tag{4}$$

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

0.1.1 Viability of Compression

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

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

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

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

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

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

this implies

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

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

$$\begin{aligned} &\text{we know from } \textbf{??} \\ &H(T_{e,t}) - H(T_{e,t}|X) = H(X) - H(X|T_{e,t}) \\ &\implies H(T_{e,t}) = H(X) - H(X|T_{e,t}) \\ &\implies H(T_{e,t}) + H(X|T_{e,t}) = H(X) \\ &\implies H(T_{e,t}) \leq H(X) \end{aligned} \tag{5}$$

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

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

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

Recall the definition

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

this implies

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

If equality is achieved in Equation 5 and F^t_θ is invertible that would imply that we have equalities in the Data Processing equations – Equation 3 and Equation 4, this would imply that every layer contains the same information content as the input and no compression can happen.

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

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

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

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

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

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

Let us define $\theta'(e)$ to be a probability distribution defined by Figure 1 (6)

Let us define $\theta(e)$ to be an instance of $\theta'(e)$ for any epoch e (7)

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

Figure 1: Probability distribution of $\theta'(e)$, where e is an epoch

The notation defined in Equation 6 and in Equation 7 implies that $F_{\theta}^{t}(x)$ is a value (as before), however $F_{\theta'}^{t}(x)$ is a probability distribution. This is due to the fact that $F(x)$ represent a layer activation for a neural network.

- If the networks weights are deterministic i.e θ, then the whole neural network is deterministic which implies that F_{θ}^{t} is a deterministic function and $F_{\theta}^{t}(x)$ is a value.
- If however the weights form a probability distribution i.e θ', then the whole neural network is probabilistic, as such $F_{\theta'}^{t}(x)$ does not return an exact value, but a distribution of possible values. Probability distribution $F_{\theta'}^{t}(x)$ is defined by Figure 2

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

Figure 2: Probability distribution of $F_{\theta'}^{t}(x)$, where θ' – is probability distribution of the weights, t – is the layer number, x – is the input to the neural network

If we assume the neural network weights θ' to be a probability distribution, then $F_{\theta'}^{t}(x)$ is not generally invertible. In order for $F_{\theta'}^{t}$ to still be invertible the Equation 8 has to be satisfied.

$$\forall i, j \in \{1, ..., N\}. P(F_{\theta'}^{t}(x_i) = t) > 0 \wedge P(F_{\theta'}^{t}(x_j) = t) > 0 \Leftrightarrow i = j \tag{8}$$

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

The degree to what the probability distributions $F_{\theta'}^{t}(x_i), i = 1, ..., N$ overlap signify the degree of compression.

0.2 Mutual Information Estimation

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

0.2.1 Mutual Information Definition

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

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

Using the entropy of the distributions to infer the value

$$I(X, Y) = H(X) - H(X|Y) \tag{9}$$

$$I(X, Y) = H(X) + H(Y) - H(X, Y) \tag{10}$$

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

$$I(X;Y) = \sum_{y \in \mathcal{Y}} \sum_{x \in \mathcal{X}} p(x,y) \log \left(\frac{p(x,y)}{p(x)\, p(y)} \right) \tag{11}$$

$$I(X;Y) = \int_{\mathcal{Y}} \int_{\mathcal{X}} p(x,y) \log \left(\frac{p(x,y)}{p(x)\, p(y)} \right) dx\, dy \tag{12}$$

0.2.2 Theoretically undefined

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

The field of estimating mutual information is relatively new as a result the tools that are available are quite primitive. There has been some papers published that use more advanced techniques to tackle the problem, we will talk about in subsection 0.3.4

0.3 Calculating Mutual Information

0.3.1 Experimental setup

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

Figure 3: The general algorithm for calculating mutual information inside a neural network.

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

0.3.2 Discrete method

The method used by Tishby in his paper. The method estimates mutual information between the X or Y and the hidden layer T by assuming the observed empirical distribution of input samples is the true distribution. We use this assumption to compute entropies of T and varying subsets as outlined in Equation 13

$$I(X,Y) = H(X) - H(X|Y) = H(X) - \sum_{y \in Y} H(X|Y=y)p(Y=y) \tag{13}$$

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

Figure 4 trough to Figure 6 outline the full algorithm.

```
Algorithm: Mutual Information
Input:
X = x_1, x_2, ..., x_n
Y = y_1, y_2, ..., y_n
Output: I(X : Y)
X = Bin close values of X together
Y = Bin close values of Y together
H(X) = Calculate entropy of X
H(X|Y) = Calculate conditional entropy of X given Y
I = H(X) − H(X|Y)
```

Figure 4: Pseudo code for computing mutual information refer to Figure 5 and Figure 6 for entropy computation.

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

Figure 5: Algorithm for computing entropy – Discrete method

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

Figure 6: Algorithm for computing conditional entropy

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

$$H(T|X) = 0 \tag{14}$$

$$I(T, X) = H(T) - H(T|X) = H(T) \tag{15}$$

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

0.3.3 Kernel Density Estimation

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

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

The algorithm closely follows figures :Figure 4, Figure 5, and Figure 6, however there is a change the way entropy is calculated so instead of Figure 5 we have Figure 7 below.

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

Figure 7: Algorithm for computing entropy – Kernel Density Estimation method. The same algorithm as used by Saxe.

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

0.3.4 Advanced methods

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

Mutual Information by Gaussian approximation

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

Geometrical Adaptive Entropy Estimation

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

0.4 Implementation Optimizations

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

0.4.1 Maximising Resource Utilization

The obvious way to maximise the resource usage is to parllelize the workload. If we refer to Figure 3, we can see two main ways we can do this.

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

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

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

0.4.2 Minimising the Workload

Even when using all the systems resources the calculations take a very long time to complete as such we need to find a way to speed it up. If we consider an information plane graph for ex. **??**, 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 8 and Figure 9. However there are two main ways how we can parallelize the algorithm.

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

```
Algorithm: DeltaSkipExact
Input:
prev = mutual information result of the previous epoch
curr = mutual information result of the current epoch
δ = user specified maximum "distance" between epochs
multiplier = how much to multiply skip by
Output:
Algorithm:
dist = Distance(prev, curr)
if dist > δ:
  Backtrack(prev, curr) # Figure 9
else:
  skip = skip*multiplier
```

Figure 8: 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 9: Backtrack Algorithm

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

The distance is used for the purposes of speeding up the computation and won't meaningfully impact the results. I've chosen to define the distance as the maximum shift between the two epochs refer to Equation 16 how to compute it or to Figure 10 for a graphical representation.

$$D = max_{t \in T}[max(I_e(X,t) - I_{\hat{e}}(X,t), I_e(Y,t) - I_{\hat{e}}(Y,t))] \tag{16}$$

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

If we consider the Figure 10 we can see that the axis are different in scale this is due to input X and label Y having different entropy values $H(X) = 12$ and $H(Y) = 1$. As such we might wish to adjust Equation 16, and scale mutual information values by the entropy as in Equation 17.

$$D = max_{s \in \{X,Y\}} max_{t \in T}[(I_e(s,t) - I_{\hat{e}}(s,t))/H(s)] \tag{17}$$

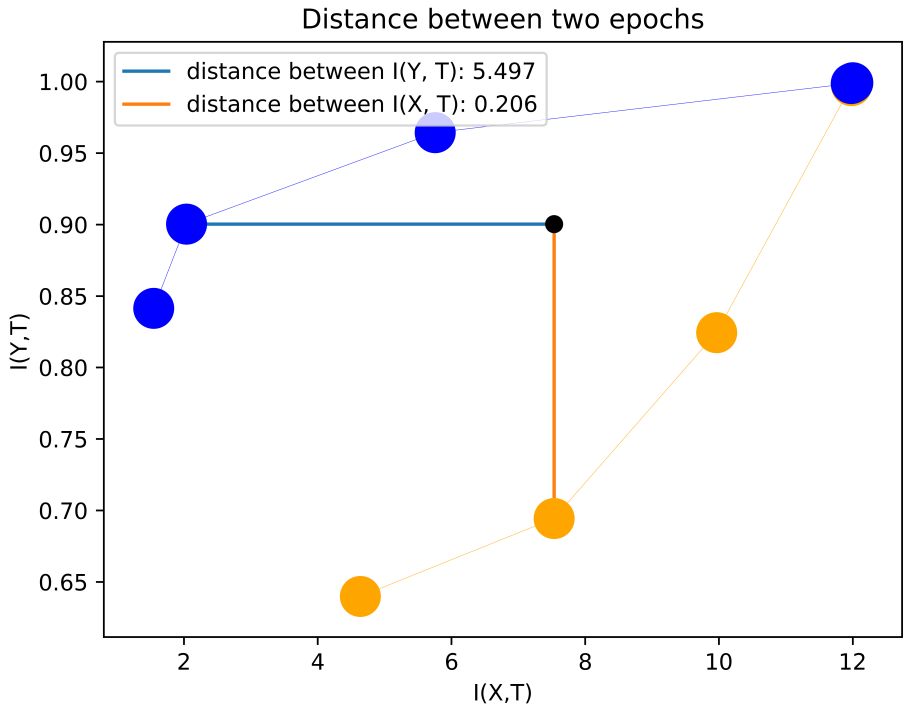

Figure 10: Example how distance between two epochs is measured.

Delta Skip – Approximate

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

The Algorithm – refer to Figure 11, as previously, in the Exact method it uses a distance metric and measures every $n'th$ epoch where n is adaptive and depends on how close the

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

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

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

Figure 11: Delta Skip Approximate

0.5 As-If-Random Experiment

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

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

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

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

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

0.6 Repository Structure