Information Processing and the Brain - Coursework

Michael Wagstaff (bj18895@bristol.ac.uk)

December 2021

1 Biological Relevance of back-propagation

1.1 How the Algorithm Works & Performance Evaluation

I implemented backprop on the MNIST dataset, achieving 90% accuracy when running through the test dataset after 500 iterations. The graph shows how the Mean Square Error changes over learning, with test runs on the model occurring every 20 iterations of training, with 200 images in each training batch.

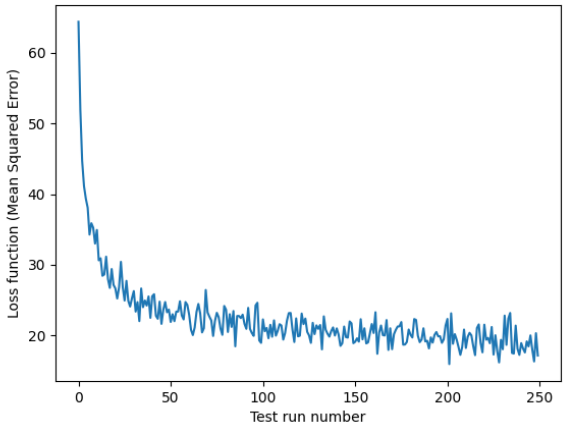

Figure 1: The loss function curve for the original backprop implementation, with test runs occurring every 20 iterations

The equation itself takes a number of inputs. First, it takes the current edge weights that are to be updated, as well as the derivative of the loss function. The derivative of the loss function is the key component to making this version of the algorithm actually learn, as this is what allows gradient descent to occur.

We are using the Mean Square Error for our loss function, which finds the mean of the squared difference between the predicted value for each neuron in the output layer and the value indicated by the label in the test data using a one-hot encoding scheme. This is used as a heuristic for how far the model is from where we need it to be, placing a stronger weight on values that are furthest away from the target. In backprop we find how much each item in the weight matrix contributed to the incorrect result, using the loss derivative as well as the weights matrices and each of the respective layer matrices. Using these, backprop calculates the partial derivatives of the loss with relation to the weight matrices, showing how much the loss would be impacted if the weights were changed. Once backprop has calculated the partial derivatives we multiply these by the learning rate and subtract the result from the weights matrices to perform gradient descent. This learning rate needs to be fairly low in order to moderate corrections to stop serial over-correction occurring, and for gradient descent to actually work. It also needs to be relatively low as our models will not have 100% accuracy and the output layer will not have perfect one-hot encoding, so it is important that we do not change weights too much in the case of small errors in the output layer, as this will stop an optimal solution (given the constraints of the models design), from being found.

```
def loss_function(preds, targets):
    loss = np.sum((preds - targets)**2)
    return 0.5 * loss

def loss_derivative(preds, targets):
    dL_dPred = (preds - targets)
    return dL_dPred

def sigmoid_derivative(a):
    dsigmoid_da = sigmoid(a)*(1-sigmoid(a))
    return dsigmoid_da

def backprop(W1, W2, dL_dPred, U, H, Z):
    dL_dW2 = np.matmul(H.T, dL_dPred)
    dL_dH = np.matmul(dL_dPred, W2.T)
    dL_dZ = np.multiply
    (sigmoid_derivative(Z), dL_dH)
    dL_dW1 = np.matmul(U.T, dL_dZ)

    return dL_dW1, dL_dW2

def train_one_batch(nn, train_imgs, train_lbls,
batch_size, learning_rate):
    inputs, targets = generate_batch
    (train_imgs, train_lbls, batch_size)
    preds, H, Z = nn.forward(inputs)
    def f(x):
        t = np.zeros(10)
        t[x]=1
        return t
    vector_targets = list(map(f, targets))
    loss = loss_function(preds, vector_targets)
    dL_dPred = loss_derivative(preds,
```

```
    vector_targets)
    dL_dW1, dL_dW2 = backprop(nn.W1, nn.W2,
    dL_dPred, U=inputs, H=H, Z=Z)
    nn.W1 -= learning_rate * dL_dW1
    nn.W2 -= learning_rate * dL_dW2
    return loss
```

As learning occurs, our model moves closer to the optimal possible given the constraints of the model design. As this happens, the items in the dL_dW1 and dL_dW2 matrices will decrease in magnitude. This happens because as we approach the minimum loss, the gradient shallows (dL_dPred decreases), and therefore the partial derivative of the loss matrices with respect to the weights will also decrease. As this occurs, the size of the change in the weights matrices $W1$ and $W2$ for each iteration will also decrease. The result of this is that the rate of learning will be highest at the start and will then gradually decrease as we go on.

1.2 How the Algorithm Relates to the Brain

Backprop is considered one possibility for how the human brain works. This intuitively makes sense, however it also presents a number of issues. These have varying degrees of severity in either the impact on the performance of the algorithm or on the feasibility of backprop.

1.2.1 Weight Transport Problem

Perhaps the most significant of these is the Weight Transport Problem. To calculate the partial derivatives within backprop, we have to know what the weights were at the time of the computation. This is potentially a large issue for the biological feasibility of backprop as no one part of the brain can have knowledge of the weights in other parts of the brain.

To do this without an external observer would require an equivalent network in both the backward and forward direction, with complex update rules to keep them in sync. We can be fairly conclusive that the structures required for this do not exist biologically. To see how this would impact backprop, I ran the model using a randomly generated fixed matrix of backward weights to be used for updates, with the loss derivative being used as a multiplier for this, as in normal backprop.

The result is a little noisy, however it shows clearly a strong similarity for the mean square error between the conventional backprop and the version with fixed random backward weights, which is perhaps a surprising result as it suggests we do not need to know how much each edge contributes to final errors to effectively train the network. This result is supported by established research [4], and largely nullifies the argument that the weight transport problem is a reason that backprop is not biologically feasible.

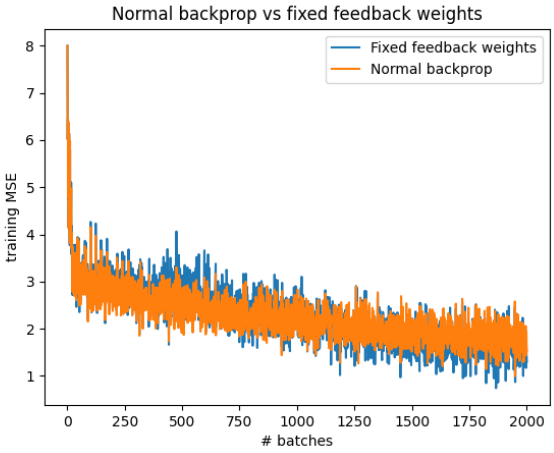

Figure 2: The mean square error for conventional backprop and backprop with fixed weights.

One intuitive way to think of this is that there are many neural networks that would produce very similar results in terms of the predictive accuracy of the output layer. During training we aim to decrease loss, so we will descend into one of these solutions. With fixed feedback weights therefore, instead of descending into any solution (with the solution being chosen in part by the random initialisation of forward matrix weights and the randomly chosen initial training data), we are instead much more likely to end up towards the theoretical minimum loss a different near-optimal solution, with the solution being chosen in part depending on the random initialisation of the fixed weights that are to be used.

1.2.2 Derivative of the Activation Function

The activation function in this case is the sigmoid function. As brain structures to calculate accurate derivatives are biologically implausible this problem is still the subject of contemporary research. Intuition however suggests that knowing the direction of the gradient and some rough idea of its magnitude will be enough to get an accurate result in time, especially as we can lower the learning rate to compensate for a higher value from any pseudo-derivative function, to ensure that over-correction will not occur. Taking this approach with the learning rate may however increase the amount of time needed to train the network.

I believe it may also be worth noting that inhibitory neurons can be used to moderate what would otherwise be frequent spike trains, which could provide a solution to this issue that is part way between having a proper derivative and the naïve approach of just knowing the sign and adjusting the learning rate accordingly. This moderation would mean that we could still get good results without having to decrease the learning rate too much, meaning that the speed of learning may not be

too adversely affected.

1.2.3 Two Phase Learning

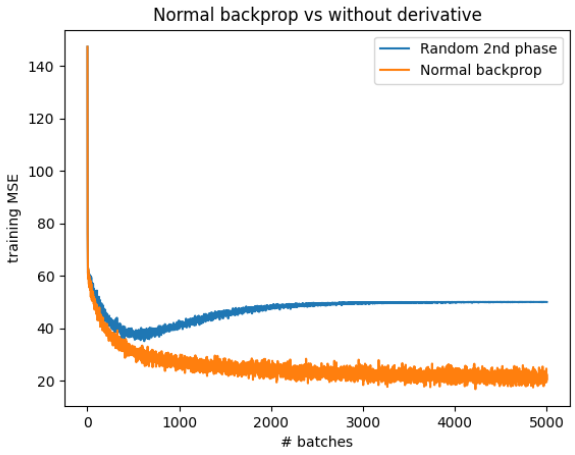

Figure 3: The mean squared error for conventional backprop and for backprop occurring randomly 20% of the time.

The final suggested issue is in regards to two phase learning. To test this we switch between using backprop and sending a random error signal 80% of the time stochastically, to model the more random behaviour of the brain, in which correct backprop may not, or more accurately would not, occur all the time. We found that the mean squared error for the system that uses a random 2nd phase 80% of the time decreases sharply in a manner roughly similar to that using normal backprop, before moving upwards as it asymptotically approaches a steady state.

Interestingly to note, the Mean Square Error when stochastically using a random second phase is better than one may initially expect, with a steady state MSE of about 55 as opposed to about 20 for regular backprop implementation. This suggests that the sum of absolute errors generated by the network trained by an often random second phase is ~66% higher. Given how well backprop performs in relation to biologically inspired supervised learning algorithms, this is still a large success.

1.2.4 Conclusion

Overall I believe that the Weight Transport Problem is does not in any way suggest that a modified version of backprop is not feasible, with little to no degradation to performance when using random weights. From a theoretical point of view, two-phase learning also does not seem to be a problem. If a bandwidth limit does exist, then using random weights 80% of the time only causes the sum of absolute errors to be 66% higher. This is a perfectly acceptable margin for biological brains to be worse than optimised algorithms, and still gives good performance. On the other hand the issue of the lack of a derivative function is a much more interesting limitation, and it is left to be seen the impact this will have of the biological plausibility of backprop.

1.3 Advantages and Disadvantages of Supervised Learning

Supervised learning has an advantage over both unsupervised learning and reinforcement learning in terms of performance, due to the fact that we know what the correct output is for a set of data. As such, we can give receive quality feedback quickly after a forward pass of the neural network. This is in contrast to unsupervised learning where we update the network without a signal - meaning we do not create system for finding correct answers, we create a system that can extract useful data from the input. In reinforcement learning, the reward and failure signals are often sparse, meaning it is difficult to know which weights caused the failure, and therefore which weights we should update and by how much. In short, supervised learning is the best way to learn quickly if there is a large amount of training data. It therefore follows that in real-world teaching, schemes that implement this approach can be used in situations where humans need to learn something novel quickly. Examples include techniques for learning languages, an example being through immersion with labelled data, including but not limited to authentic media with subtitles [3]. To discuss where this takes place in the real world though, it may also be worth slightly broadening the requirements for what classes as supervised learning.

With humans possessing multiple sensory systems, it is easy to imagine that one system can act as a sort-of-labeller for other systems. Speech can be labelled either in real time or in knowledge gained from the auditory system in toddlers learning to speak - after all they've had years of unsupervised learning to be able to separate patterns. This continues for a number of other pairs of systems, where one is able to generate data for another, including but not limited to, sight and touch, visual knowledge and perceived temperature, taste and smell etc.

The most widely used algorithm for supervised learning is backprop, which has unsolved problems relating to how it could be implemented biologically, with major ones such as the derivative of activation functions being discussed above, and others such as the requirement for a separate error network being well documented. Although strong progress has been made in recent years, we are still a long way from a solution for how backprop could work in the brain, and alternative biologically inspired algorithms are lacking in performance [4]. On the other hand, it has been documented for many years that unsupervised learning techniques exist in and are used in

key parts of the brain including the visual cortex [5].

Another disadvantage of supervised learning is the requirement for an error signal to be generated. Outside of the possibility for some parts of the body to create either the input or the label for another, either in real time or through previous data and learning, the circumstances for supervised learning in animals are sparse. For many everyday scenarios the data simply does not exist, or is not provided in a timely enough manner for supervised learning to take place. As such reinforcement learning is often used for when a correct answer is needed, and unsupervised learning is used for when pattern recognition is needed.

2 Information Theory Analysis

2.1 Part A

For X as a random variable representing the input, $H(X) = log(10)$ as mentioned in the brief. With Y representing the hidden layer activity, $H(Y|X) = 0$, as Y is fully derivable from X given U, the matrix that converts the input data into the hidden layer values [2]. This does not change with learning, as given the matrix U for an iteration, which we know for all iterations, we can derive Y from X.

$I(X,Y) = H(Y) - H(Y|X)$

With $H(Y|X) = 0$:

$I(X,Y) = H(Y)$

$$H(Y) = -\sum_{x_i \in X} p_x(x_i) \cdot log_2(p_x(x_i))$$

To fill this equation, I decided to bin the hidden layer values [1] using a single threshold value, the mean value for elements in the H matrix across the entire test space. I set each value in the H matrix to either 0 or 1, in order to limit the number of potential H matrices, and hence stop the entropy from being infinite. Following this we count how many times each possible discretised H matrix is reached. We use this to generate an estimated probability for each possible discretised activation of the hidden layer.

```
if(find_avg_H):
    avg = np.sum(H)/(H.size)
    discretised_occurrences_count = {}
    for i in range(H.shape[0]):
        for j in range(H.shape[1]):
            H[i][j] = 1 if H[i][j] > avg else 0
        if not tuple(H[i].tolist()) in
        discretised_occurrences_count:
            discretised_occurrences_count
            [tuple(H[i].tolist())] = 1
        else:
            discretised_occurrences_count
            [tuple(H[i].tolist())] += 1
    for _, item in
    discretised_occurrences_count.items():
        current_entropy_sum +=
        - (item/H.shape[0]) *
        math.log2(item/H.shape[0])
    print("Entropy: " +
    str(current_entropy_sum))
```

I then plotted the entropy of Y over learning, with 500 test runs, showing an initial sharp increase, followed by a continuous decrease over time

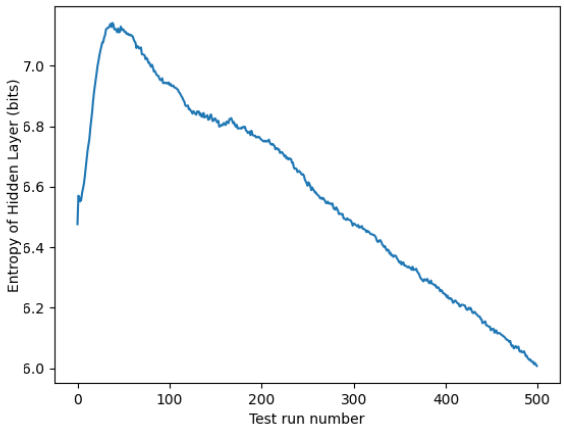

Figure 4: The entropy $H(Y)$ (and therefore $I(X,Y)$) with hidden layer size 10 over learning with 500 test runs.

Following this, I wished to see how the entropy changed with the size of the hidden layer. I chose to iterate in multiples of 5 between 5 and 40, however this presents a problem. $2^{40} = 1.1 \cdot 10^{12}$, much higher than our 10,000 strong test dataset. To partly compensate for this, I chose to instead use the entire MNIST dataset for calculating entropy, although it is worth noting that results for higher sizes should perhaps still be taken with a pinch of salt.

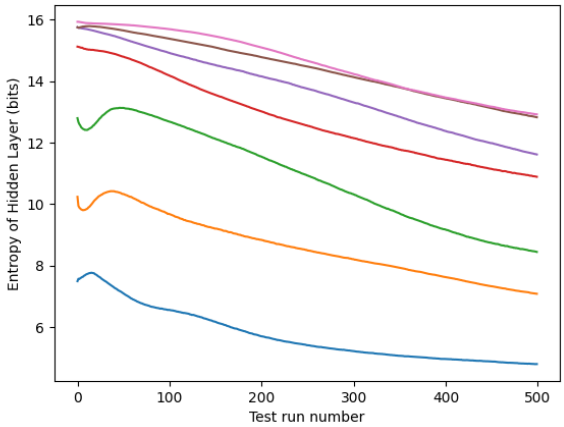

Figure 5: The entropy $H(Y)$ (and therefore $I(X,Y)$) with hidden layer sizes ranging from 10 to 40 over learning with 500 test runs.

In this graph the entropy for hidden layer size 10 is at the bottom, with each successive line representing a

hidden layer size 5 neurons bigger. These results roughly reflect what I would expect, with smaller hidden layer sizes having lower entropy levels. It is interesting that the entropy for the larger layer sizes tends to bunch up at the top, however it is hard to say if this is due to not enough data to generate an accurate distribution for the activations of the discretised hidden layer.

To calculate the mutual information $I(X, Z)$, similar logic is used to earlier. As U is known as well as $W1$, H is therefore derivable for each iteration. Given H and $W2$, we also know that Z is derivable. As U, $W1$, and $W2$ are known for each iteration, we can determine Z, meaning that $H(Z|X) = 0$ As such $I(X, Z) = H(Z)$

This generated a similar result to what is seen above, with entropy initially rising before falling continuously.

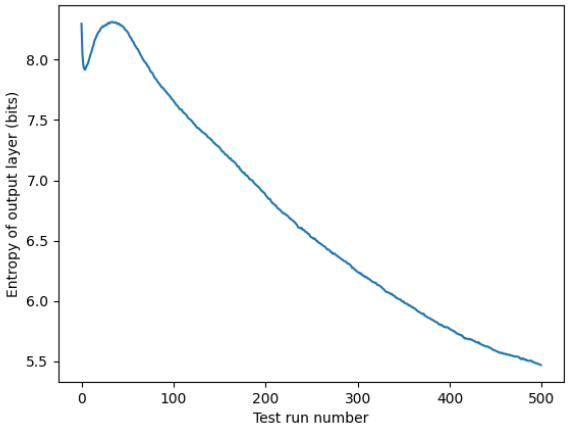

Figure 6: The entropy of $H(Z)$ (and therefore $I(X, Z)$) with hidden layer size 10 over learning with 500 test runs.

This asymptotically approaches a number near the theoretical perfect value of 3.32 ($log_2(10)$) from above, which is the entropy of the labels, with the assumption that an equal number of labels exist for each number. Given that we randomly distribute the weights on initialisation of the neural network, we may expect large number of 1s than would be anticipated in a one-hot encoding scheme after applying the threshold function on the output layer. The highest entropy in the output layer would occur if we expect an average of 5 items in the output matrix to be 1s chosen randomly. This would imply a 0.5 chance of each neuron being a 1, giving an entropy of 10 bits. My starting entropy is slightly lower than this theoretical maximum at around 8.2 bits.

2.2 Part B

Following this, I looked at the mutual information between W, the random variable representing the quadrant with the largest amount of white, and Y or Z, the hidden and output layers respectively.

$I(W, Y) = H(Y) - H(Y|W)$

$H(Y)$ has been calculated in part A.

$H(Y|W) = -\sum p_{Y|W}(y_i|w_i) \cdot log_2(p_{Y|W}(y_i|w_i))$

To compute this, we will use our discretised H matrix, and will count the number of times each discretised possibility is hit for a given w_i. Counting the number of times each w_i occurs allows us to calculate $p_{Y|W}(y_i|w_i)$ for a given w_i. We then compute $H(Y|W = w_i)$ by plugging the probabilities in. We then calculate the weighted sum using $p(w_i)$ for the weights.

$$H(Y|W) = \sum_{w_i \in W} p(w_i) \cdot H(Y|W = w_i)$$

The next task is to compute $I(W, Y)$, which is done simply by subtracting the conditional entropy just calculated from the entropy of Y, as calculated before.

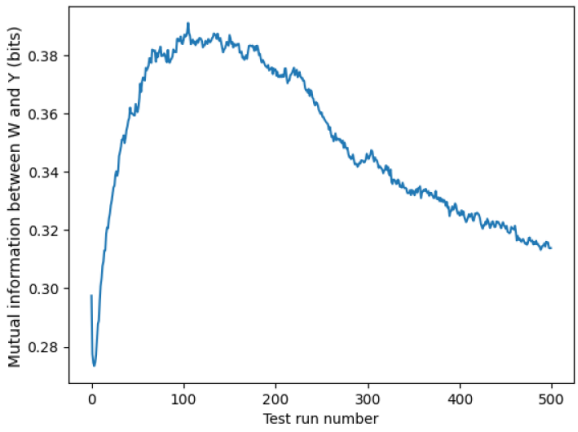

Figure 7: The mutual information $I(W, Y)$ with hidden layer size 10 over learning with 500 test runs.

Following this, I used the same technique outlines above to calculate the mutual information between W and Z.

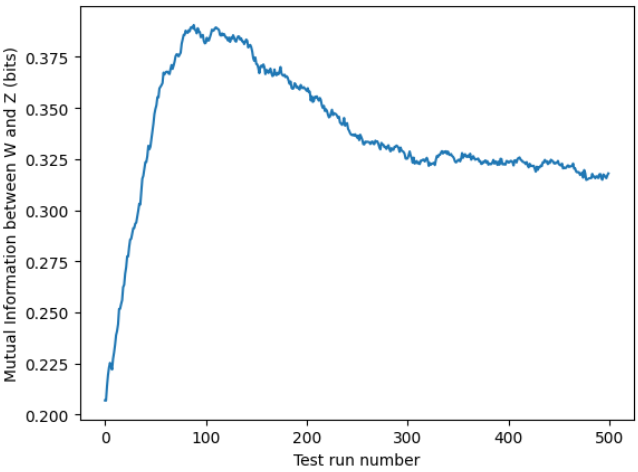

Figure 8: The mutual information $I(W, Z)$ with output layer size 10 over learning with 500 test runs.

This graph is lower than one might initially expect for the mutual information between the quadrant with the most white and the output layer, however I believe this makes sense. Although some numbers are more likely to have the most white in some quadrants (e.g. 9 is most likely to have the most white in one of the top two quadrants), for others we simply do not know. An example of this is the number 1, which depending on which side of the centre line it predominantly falls may have the most white in any of the quadrants, with rough;y equal probabilities.

To extend this, I wanted to look at the mutual information between the level of white in the whole image and the output layer. This is informed by the intuition that some numbers are made of less 'line' than others, for example 1 vs 8 in most common handwriting. I did this by binning the brightness of the image into one of four groups, before calculating the average value within the output layer for each quartile. The mutual information is then computed in a similar manner to before. Code for this process is found below.

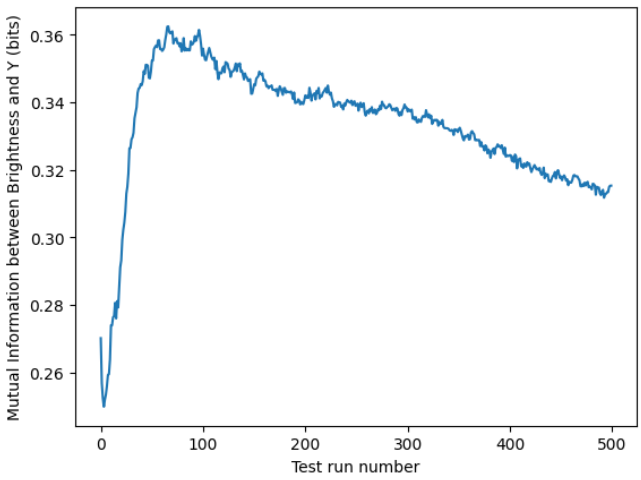

Figure 9: The mutual information $I(B, Z)$ with hidden layer size 10 over learning with 500 test runs.

This result matches somewhat closely with the result for $I(W, Y)$, which is not too surprising given the similarity in what they show. I would also expect a similar graph to occur for the mutual information of a number of other properties of the input layer and the hidden layer, as it is hard to imagine that a non-vacuous property would share no mutual information with partially computed data.

This result (Fig. 10) is perhaps a little surprising to me and shows that there is a small but very limited amount of mutual information between the brightness of the input image and the output layer. This also roughly similar to the graph between showing the mutual information between the brightest quadrant and the output layer. The graoh being similar makes sense, however I would have expected the final mutual information to be slightly higher

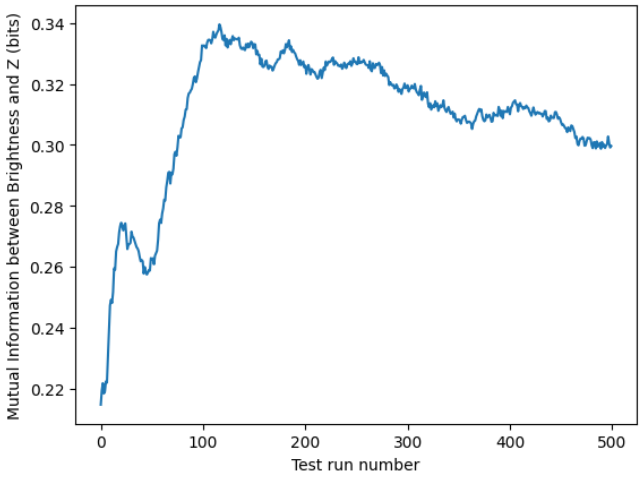

Figure 10: The mutual information $I(B, Z)$ with output layer size 10 over learning with 500 test runs.

```
discretised_occurrences_count = [{},{},{},{}]
avg = [0,0,0,0]
Hs = []
brightness_sums = []
brightness_sums_array = []
for i in range(H.shape[0]):
    brightness_sums.append
    ((str(np.sum(inputs[i])), inputs[i]))
brightness_sums.sort(key=lambda x:x[0])
for i in range(len(brightness_sums)):
    brightness_sums_array.append
    (brightness_sums[i][1])
for i in range(0, 4):
    preds, H1, Z1 = nn.forward
    (brightness_sums_array[17500*i:17500*(i+1)])
    avg[i] = np.sum(H1) / (17500 *10)
    Hs.append(H1)
for i in range(4):
    for j in range(Hs[i].shape[0]):
        for k in range(Hs[i].shape[1]):
            Hs[i][j][k] = 1 if
            Hs[i][j][k] > avg[i] else 0
        if not tuple(Hs[i][j].tolist())
        in discretised_occurrences_count[i]:
            discretised_occurrences_count[i]
            [tuple(Hs[i][j].tolist())] = 1
        else:
            discretised_occurrences_count[i]
            [tuple(Hs[i][j].tolist())] += 1
for i in range(0,4):
    for _, item in
    discretised_occurrences_count[i].items():
        current_entropy_sum[i] +=
        -(item/17500) * math.log2(item/17500)
    final_entropy += 0.25*current_entropy_sum[i]
final_entropy = test(nn, test_images,
test_labels, True, False, True)
- final_entropy
print("Mutual Information: "+str(final_entropy))
```

References

[1] R. Schwartz-Ziv & N. Tishby, "Opening the black box of Deep Neural Networks via Information", 2017

[2] I. Chelombiev, C. J. Houghton & C. O'Donnell, "Adaptive Estimators Show Information Compression in Deep neural Networks", 2019

[3] H. Mitterer & J. M. McQueen, "Foreign Subtitles Help but Native-Language Subtitles Harm Foreign Speech Perception", 2009

[4] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman & G. Hinton, "Backpropagation and the brain", Nature Reviews Neuroscience Vol. 21, pp. 335-346, 2020

[5] H. B Barlow, "Unsupervised Learning", Neural Computation Vol. 1, No. 3, pp. 295-311, 1989