Coursework: Analysing deep neural networks as brain models

Joel Grimmer

November 2022

Contents

1 Biological Relevance of backprop

1.1 How does the algorithm work?

1.1.1 Theory of backprop

Back-propagation is a method used to train artificial neural networks [3] to solve the credit assignment problem by iteratively updating the network weights to minimise the error between the predicted and true label of the input data in a process known as gradient descent optimisation.

The error between the model's predictions and the ground truth is calculated using a loss function such as Mean Squared Error (MSE) and Mean Absolute Error (MAE):

$$\text{(MSE)} : \mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(v_i - \hat{v}_i)^2 \qquad \text{(MAE)} : \mathcal{L} = \sum_{i=1}^{N}|v_i - \hat{v}_i|$$

where v_i is the target and $\hat{v}_i$ is the model prediction.

The process through which network weights are updated by gradient descent optimisation is represented by the following equation:

$$W_i \leftarrow W_i - \eta\frac{\partial \mathcal{L}}{\partial W_i}$$

where W_i is the weight matrix at the ith layer, η is the learning rate, and L is the loss function. The gradient of the error function with respect to the weights, $\frac{\partial \mathcal{L}}{\partial W_i}$, is calculated using the chain rule. It can be derived that

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{i+1}}\frac{\partial \mathbf{h}_{i+1}}{\partial \mathbf{h}_i}$$
$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_i}\frac{\partial \mathbf{h}_i}{\partial W_i}$$

where $\mathbf{h}_i$ and W_i represent the hidden activity and weights at layer i respectively.

Neural Networks often apply activation functions to each neuron. Common functions include:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}}$$
$$Relu(z) = max(0, z)$$

By exploiting the fact that forward passes of the network can be computed through matrix multiplication of weight matrices a batch of inputs can be used to update the weights wrt. the loss of all batch items.

A network consisting of one hidden layer takes an input batch $\mathbf{U}$, multiplies it by W_1 to produce $\mathbf{Z}$. The activation function f is applied to $\mathbf{Z}$ to produce the hidden activity $\mathbf{H}$. This is multiplied by W_2 to produce a vector of predictions $\hat{\mathbf{v}}$.

$$\mathbf{Z} = \mathbf{U}W_1$$
$$\mathbf{H} = f(\mathbf{Z})$$
$$\hat{\mathbf{v}} = \mathbf{H}W_2$$

The partial derivatives $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_i}$, $\frac{\partial \mathcal{L}}{\partial W_i}$ are derivatives of vectors and so are Jacobian Matricies, making these rules apply [1]:

$$C = AB \Rightarrow \frac{\partial \mathcal{L}}{\partial A} = \frac{\partial \mathcal{L}}{\partial C}B^\top \text{ and } \frac{\partial \mathcal{L}}{\partial B} = A^\top\frac{\partial \mathcal{L}}{\partial C}$$
$$C = g(A) \Rightarrow \frac{\partial \mathcal{L}}{\partial A} = g'(A) \odot \frac{\partial \mathcal{L}}{\partial C}$$

[1] https://mathworld.wolfram.com/Jacobian.html

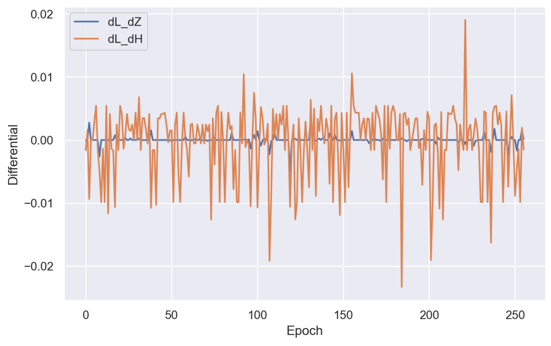
allowing us to derive the following rules for calculating relevant partial derivatives

$$\hat{\mathbf{v}} = \mathbf{H}W_2 \Rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{H}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{v}}} W_2^\top \text{ and } \frac{\partial \mathcal{L}}{\partial W_2} = \mathbf{H}^\top \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{v}}}$$

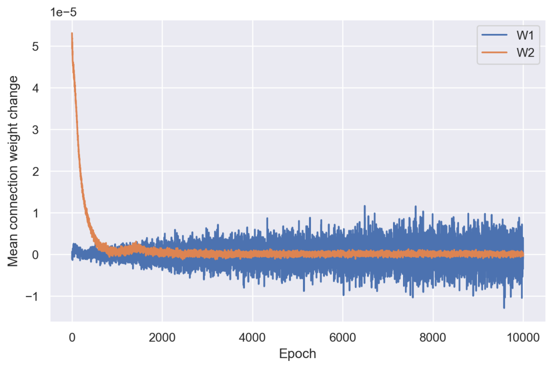
$$\mathbf{H} = f(\mathbf{Z}) \Rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{Z}} = f'(\mathbf{Z}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{H}}$$

$$\mathbf{Z} = \mathbf{U}W_1 \Rightarrow \frac{\partial \mathcal{L}}{\partial W_1} = \mathbf{U}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Z}}$$

Figure 1b shows the convergence of W_2 whilst W_1 keep changing to optimise the hidden layer representation.



(a) Intermediate differential values during training



(b) Average weight change during training

Figure 1: Plot of different network components over learning.

1.1.2 Implementation details

Our simple feed-forward neural network features a single hidden layer and is trained on the fashion MNIST dataset[2]. The dataset consists of 10,000 28x28 greyscale images with an associated label from 10 different classes. Labels for these classes are given as one-hot encodings[3] making them easy to compare to the one-hot output of the network. We do not employ regularisation to the network weights, so the objective function is equal to the error function. MSE loss is chosen as the loss function for training the network.

The network features a single hidden layer, and experiments have been completed with hidden layer size 15, 30, 60, and 120 neurons. A learning rate of $\eta = 5e-6$ is used and weights are initialized under a gaussian distribution with $\mu = 0, \sigma = 0.01$.

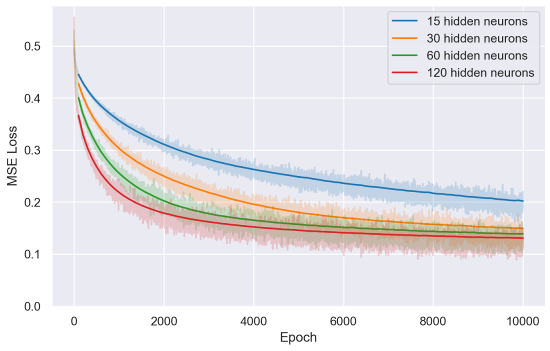
`backprop` returns the derivatives $\frac{\partial \mathcal{L}}{\partial W_1}$ and $\frac{\partial \mathcal{L}}{\partial W_2}$ using the above rules and the function `train_one_batch` updates the weights by the derivatives at a given learning rate.

```python
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_prime(Z), dL_dH)
    dL_dW1 = np.matmul(U.T, dL_dZ)

    return dL_dW1, dL_dW2

def train_one_batch(nn, inputs, targets, batch_size, lr):
    inputs, targets = generate_batch(inputs, targets, batch_size)
    preds, H, Z = nn.forward(inputs)
    dL_dPred = nn.loss_deriv(preds, targets)

    loss = loss_mse(preds, targets)

    dL_dPred = loss_deriv(preds, targets)
    dL_dW1, dL_dW2 = backprop(nn.W1, nn.W2, dL_dPred, U=inputs, H=H, Z=Z)

    nn.W1 -= lr * dL_dW1
    nn.W2 -= lr * dL_dW2

    return loss, preds
```

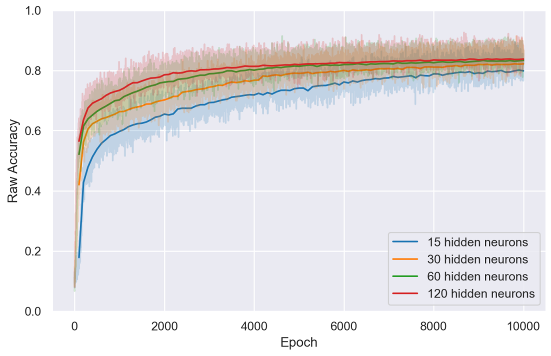
[2]github.com/zalandoresearch/fashion-mnist
[3]en.wikipedia.org/wiki/One-hot

Plots of the value of the loss function and the classification accuracy during training demonstrate the iterative nature of back-propagation. While training it is often useful to test the network's ability to generalise to unseen data. For our chosen plots we test the network against the test set every 100 epochs and plot this on the same graph.

Figure 2a demonstrates that given an equal learning rate, a network with more neurons within its hidden layer converges more quickly with regards to both training and test cost.



(a) Train, validation MSE loss



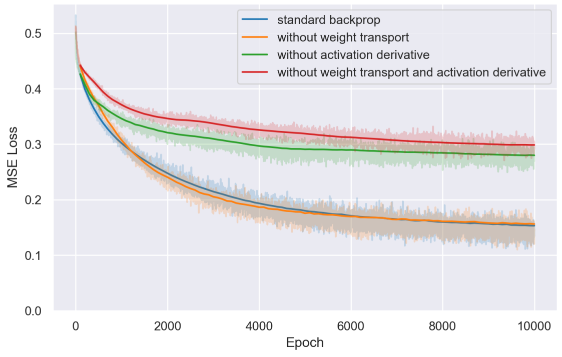
(b) Train, validation accuracy

Figure 2: Learning curves for networks of hidden layer size 15, 20, 60 and 120. Train values are lighter while test values are opaque.

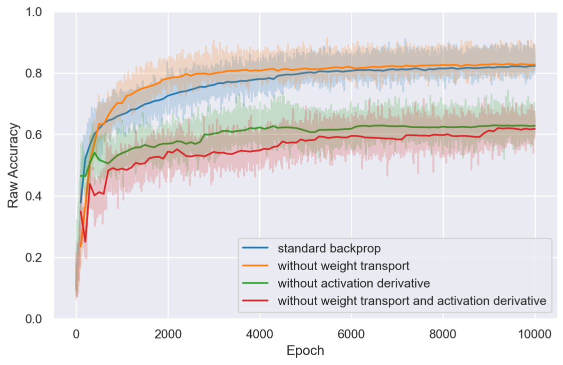
1.2 How does the algorithm relate to the brain?

There is evidence that backprop-trained models can account for some neural responses found in different regions of the brain, especially with regards to how the brain assigns credit to particular neurons. [5]

We will explore using simulations potential solutions three of the key issues which prevent backprop from being able to act as a biologically viable learning.



(a) MSE loss against epoch



(b) Accuracy against epoch

Figure 3: Comparison of training curves for four simulations of back-propagation, those being standard backprop, backprop without weight transport, backprop without use of an activation derivative, and backprop with neither weight transport nor use of an activation derivative

1.2.1 Weight Transport

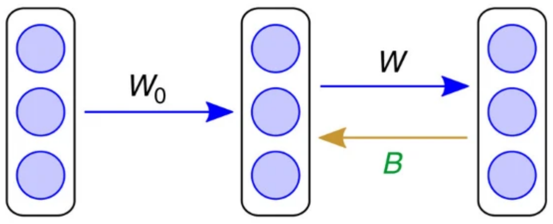
The weight transport problem specifies that the chain rule requires symmetric feedback and feedforward weights. There is no evidence for this in the brain.

We generate fixed feedback weights in place of network weights

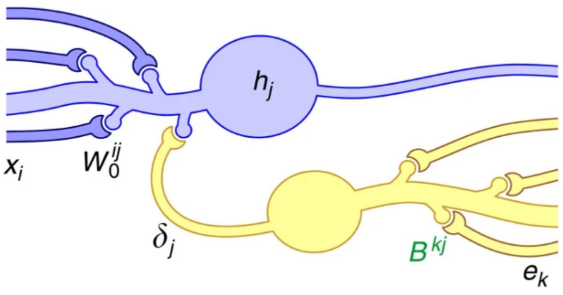
```
W1_fixed_feedback = np.random.randn(input_size, hidden_size)
W2_fixed_feedback = np.random.randn(hidden_size, output_size)

# intermediate code omitted

def train_one_batch(lr):
    # earlier steps in function omitted
    W1_feedback = model.W1
    W2_feedback = model.W2

    if not weight_transport:
        W1_feedback = W1_fixed_feedback
        W2_feedback = W2_fixed_feedback

    dL_dW1, dL_dW2 = backprop(
        W1_feedback,
        W2_feedback,
        dL_dPred,
        U=inputs,
        H=H,
        Z=Z,
        activate=activation_derivative,
    )
    # next steps in function omitted
```

Figure 3 demonstrates that backprop is not impeded. Loss and accuracy converge at least as fast when using fixed weights. This is inline with previous research, that finds enforcing symmetric weights is unnecessary for credit assignment. [4]

This phenomenon is known as feedback-alignment, [6] where each neuron in the hidden layer gets a random projection of the error vector, which allows weights to be adjusted in such a way as to reduce error.



(a) Feedback alignment replaces W^T with random fixed feedback weights B



(b) Potential synaptic circuitry underlying feedback alignment

Figure 4: Demonstration of learning with feedback alignment in a Neural Network and in the brain, from Lilicrap et al. [6]

figure 4 demonstrates the biological basis for the idea that the brain may use fixed feedback weights. The Martinotti cell (yellow) provides fixed feedback weights to the pyramidal cell (blue) in a dendritic microcircuit [8].

1.2.2 Derivative of Activation Function

In biology it is unclear how neurons would be able to calculate their own derivative.

The below snippet demonstrates how to simulate backprop without the use of a activation function derivative, by setting $\frac{\partial \mathcal{L}}{\partial Z} = \frac{\partial \mathcal{L}}{\partial H}$ rather than setting $\frac{\partial \mathcal{L}}{\partial \mathbf{Z}} = f'(\mathbf{Z}) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{H}}$.

```
def backprop(W1, W2, dL_dPred, U, H, Z, activate=True):
        dL_dW2 = np.matmul(H.T, dL_dPred)

        dL_dH = np.matmul(dL_dPred, W2.T)
        if activate:
            dL_dZ = np.multiply(self.sigmoid_prime(Z), dL_dH)
        else:
            dL_dZ = dL_dH
        dL_dW1 = np.matmul(U.T, dL_dZ)

        return dL_dW1, dL_dW2
```

Figure 3 demonstrates that removing the activation derivative from backprop results in loss and train curves converging to inferior values than those produced by default backprop however learning still takes place, showing that this process may be applicable in biology.

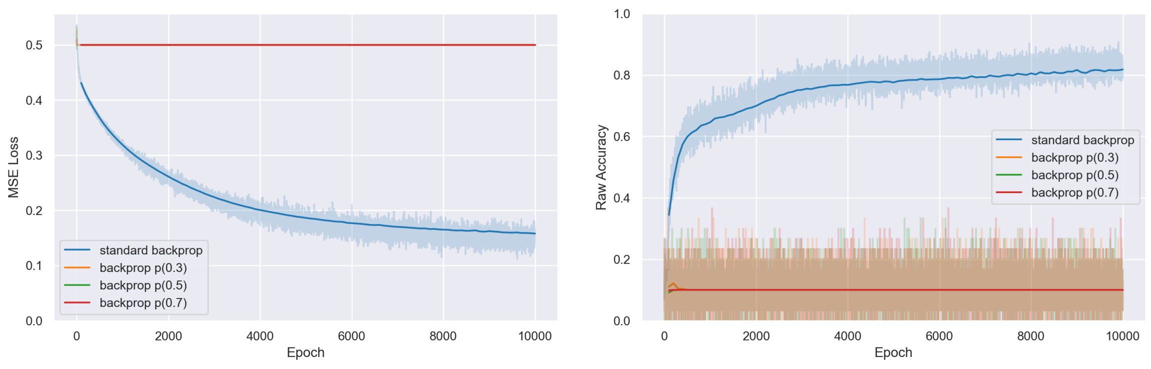
1.2.3 Two Phase Learning

Back propagation relies on the use of a forward and backward pass through the network whereas in biology there is no clear separation between these two phases.

A potential solution given by Sacramental et al. [8], produces a random backwards pass `prob_not_backprop`% of times the backprop algorithm is run, simulating the highly stochastic nature of the brain.

```python
def backprop(W1, W2, dL_dPred, U, H, Z, activate=True, prob_not_backprop=0):
    dL_dW2 = np.matmul(H.T, dL_dPred)

    if np.random.uniform() > prob_not_backprop:
        dL_dH = np.matmul(dL_dPred, W2.T)
        dL_dZ = np.multiply(self.sigmoid_prime(Z), dL_dH)
        dL_dW1 = np.matmul(U.T, dL_dZ)
    else:
        dL_dH = 1
        dL_dZ = dL_dH
        dL_dW1 = U.T

    return dL_dW1, dL_dW2
```

During out simulations learning is prevented from taking place entirely, as shown in figure 5.



(a) MSE Loss against Epoch (b) Accuracy against Epoch

Figure 5: Demonstration of learning with stochastic randomness during the backward pass

The layered structure of sensory processing in the brain is akin to the layers of a neural network, and recent research has found that dendritic microcircuits can approximate backprop. This presents us with another solution to the issue of two phase learning.

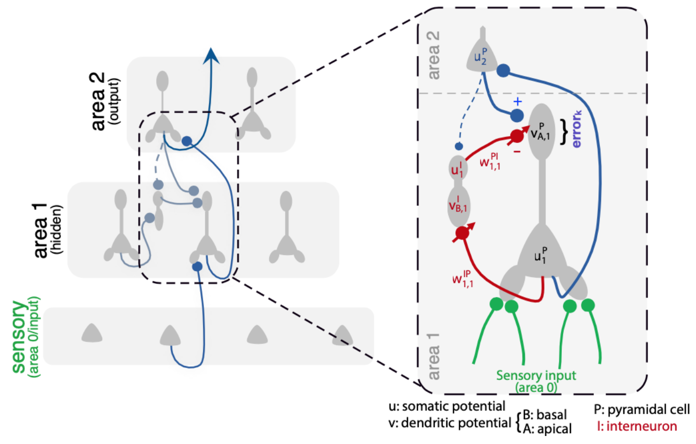


Figure 6: Sensory layers in the brain made comprised of layers of dendritic microcircuits, from Sacramento et al. [8]

There is no need for two seperate phases during backprop when using this structure as the difference between lateral inhibition u_k^l and topdown activity u_{k+1}^P generates an error signal which is then fed backwards using seperate circatory to trigger changes in bottom weights $w_{k,k-1}^{PP}$, which when recursively trained to lead to zero error. This has been found by Sacramento et al. [8] to correctly classify MNIST handwritten digit images.

1.3 Advantages of supervised learning

Supervised learning generally requires fewer data points to achieve good performance than unsupervised learning. Unsupervised learning algorithms must discover the underlying structure of the data without the aid of input-label mappings in order to cluster or find a lowest cost data encoding.

TDL and Q-learning sample action-state pairs and for a large state and action space many action-state pairs may be required to create a mapping of the action-space state and find an optimal policy. Value Iteration requires repeated iterations of actions in certain spaces to find an environment, and as the number of possible actions and states increases the number of iterations required to find the mapping can increase exponentially. This impacts are compounded by Markov Decision Spaces which may be stochastic and/or history dependent. Supervised learning provides ground truth labels in the data-set and is stateless so the error between prediction and ground truth is far quicker to find than the difference between policy value and optimal policy value.

Whilst unsupervised and reinforcement learning are considered to take place more often in the brain there is evidence that supervised learning takes place in the cerebellum. The forward model of the cerebellum states that given a motor command the cerebellum predicts its output in the environment using sensory information, with the difference between the sensory consequences of the action and the prediction action used as an error signal.

1.4 Disadvantages of supervised learning

Supervised learning requires a large amount of labeled data to train a model that performs well. This can be challenging and time-consuming to obtain, especially for tasks that require specialized knowledge or expertise to correctly label data. Unsupervised learning does not require labelled data, and reinforcement learning samples an environment directly for action-reward pairs rather than needing a supervisor to label data-points.

Supervised learning can lack flexibility and adaptability when generalising to unseen data or unseen labels compared to unsupervised or reinforcement learning. Unsupervised algorithms such as k-means clustering can perform accurately when given data that may need to have a new cluster, whereas KNN can only predict existing labels for new data-points.

There is far less evidence of supervised learning taking place in the brain than the other two paradigms. Supervised learning networks often require far more training examples than a human would witness within their lifetime, and brains often do not have access to ground truth labels for correctly identifying sensory information.

Deep Q-learning uses neural networks, a form of supervised learning, as a function approximator to improve reinforcement learning performance, allowing supervised learning to be used in stateful contexts and be applied to unseen situations by learning a set of parameters to feed to the Q function.

Semi-supervised learning can improve on supervised learning by training the model on a combination of labeled and unlabeled data, which can help it learn more generalizable patterns and reduce the need for large amounts of labeled data.

2 Information theory analysis

2.1 Analysis of hidden and output layer activity entropy

For a layer consisting of N neurons, Y and Z each correspond to a binary sequence of length N. Figure 7 demonstrates this process.

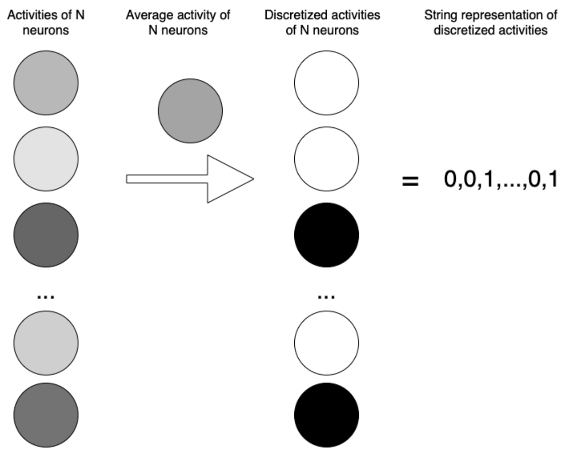

Activities of N neurons
Average activity of N neurons
Discretized activities of N neurons
String representation of discretized activities
...
...
= 0,0,1,...,0,1

Figure 7: Visualisation of generation of network activity bit-string representation

Our random variables approximations of neural network activity can be constructed as a communication channel

$$X : \text{label} \to \text{image} \to Y : \text{hidden layer} \to Z : \text{output}$$

where each random variable in the channel represents some encoding of the previous representation. We express this as a Markov Chain, where:

$$p(x, z|y) = p(x|y)p(z|y)$$

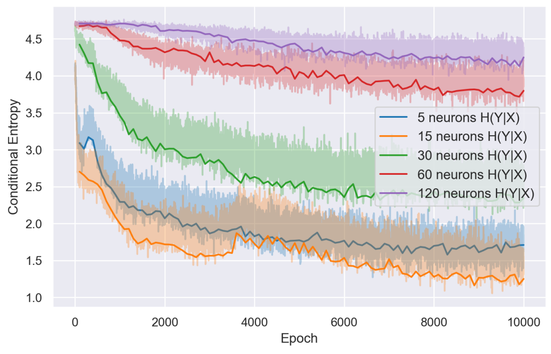
Z knows no more information about X than Y, which is formally stated in the data processing inequality.

$$I(X, Y) \geq I(X, Z)$$

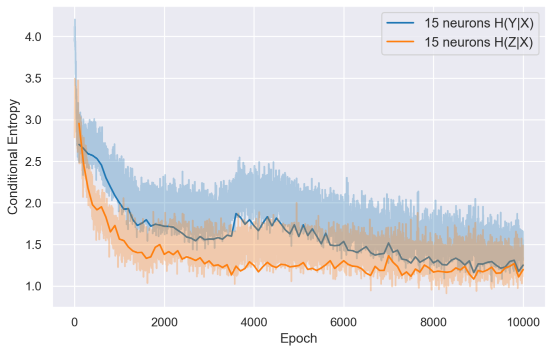
When X represents the class of the input and Y represents the discretized hidden layer activity of the network, $H(Y|X)$ represents the uncertainty in the discretized hidden layer when the input label X is known. Figure 8a demonstrates how $H(Y|X)$ decreases during training for all hidden layer sizes.

Increasing the number of hidden neurons exponentially increases the total permutations of binary sequences, lowering chance that a given binary sequence can be predicted to appear given an image label. Figure 8a shows that a larger hidden layer results in higher conditional entropy over training due to the increased uncertainty of Y.

Conditional entropy is also a product of how closely the hidden layer encoding can represent the label encoding. The network with 15 hidden neurons converges to the lowest conditional entropy of all networks by balancing the ability to encode the label accurately with maintaining a low number of hidden neurons. Figure 8b shows that the test values of $H(Y|X)$ and $H(Z|X)$ for the network with 15 neurons converge to near equality during training, demonstrating that the uncertainty in Y is nearly as low as Z for a given label X. This convergence represents the DNN undergoing compression, where the network finds the maximal compression of the input. [1]

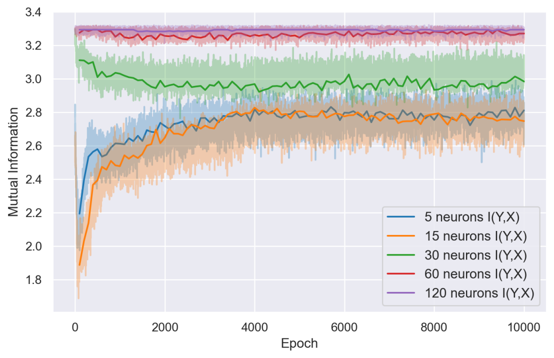


(a) $H(Y|X)$ during training for different sizes of hidden layer

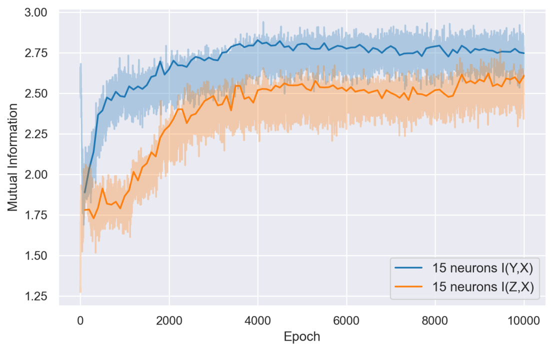


(b) Comparison between $H(Y|X)$ and $H(Z|X)$ during training

Figure 8: Changes in conditional entropy during training



(a) $I(Y, X)$ during training for different sizes of hidden layer



(b) Comparison between $I(Y, X)$ and $I(Z, X)$ during training

Figure 9: Changes in mutual information during training

The mutual information $I(X, Y)$ between two random variables X and Y measures the average reduction in uncertainty about the distribution of x that results from learning the value of y or vice versa.[7]

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

As the interdependence between the two random variables increases so does $I(Y, X)$.

The dependence between the distribution of binary sequences produced by Y and the input X increases during training. As such we would expect $I(Y, X)$ to increase during learning. Figure 9b demonstrates that the data processing inequality $I(X, Y) \geq I(X, Z)$ is maintained throughout training. Furthermore, both $I(Y, X)$ and $I(Z, X)$ increase at the start of training for a network with 15 hidden neurons, showing that the amount of information in Y increases as the network learns. Both $I(Y, X)$ and $I(Z, X)$ plateau after 4,000 epochs, as opposed to a constant increase which would imply memorization, a trait linked to over-fitting and poor generalization [1].

Figure 9a shows this is true for a network consisting of 5 or 15 neurons, however for a network with 30 hidden neurons the mutual information decreases and for a network with 60 or 120 hidden neurons the mutual information remains flat. The amount of bits stored

2.2 Analysis of image property entropy

We explore how the mutual of the brightest quadrant changes through learning.

```
def brightest_corner(image: np.array) -> int:
    i = image.reshape((28, 28))
    corners = np.array(
        [
            i[0:14, 0:14],
            i[0:14, 14:28],
            i[14:28, 0:14],
            i[14:28, 14:28],
        ]
    )
    brightest = np.argmax(corners.sum(axis=1).sum(axis=1))
    return brightest
```

Brighter pixel values correspond to higher pixel values, and so we split the image into four quadrants and then find which section has the greatest sum of pixel values. We then return either $0, 1, 2, 3$ depending on which quadrant is brightest.

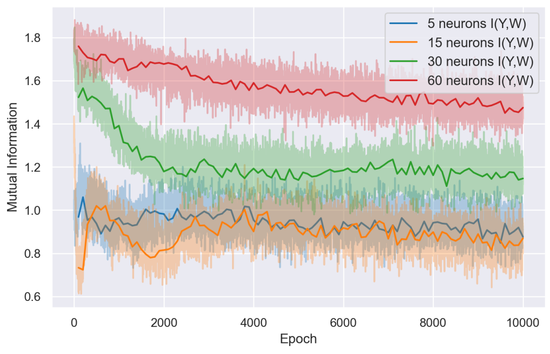
10b demonstrates how the mutual information between the brightest quadrant and the output and hidden layer activity change over training.

$I(Y, W)$ increases from 0 to 500 epochs, then decreases to 2000 epochs, then increases to 4000 epochs and then decreases to 10,000 epochs. In contrast $I(Z, W)$ remains relatively consistent throughout training.

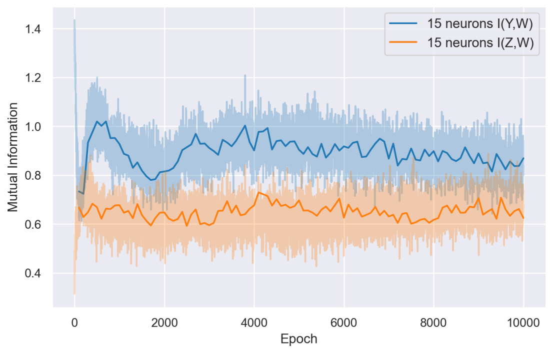
Information between Y and W increases when the network reduces the loss in a phase known as fitting and decreases when the network maximally compresses information from the input image in a phase known as compression [1]. Compression can take place due to the use of sigmoid as an activation function which is saturating.

$I(Z,W)$ remains consistent because compression reduces the information on input data relative to learning the output error [9]. This means the network loss is not increased so meaning Z remains constant.

An explanation for the sharp increase and then decrease in $I(Y,W)$ may be that Y is encoding basic image properties at the start of trainig, which can include features that approximate to the brightest quadrant. As the network continues training these basic features are discarded for more complex features, leading to a decrease in $I(Y,W)$.



(a) $I(Y,W)$ during training for different sizes of hidden layer



(b) Comparison between $I(Y,W)$ and $I(Z,W)$ during training

Figure 10: Changes in mutual information of image property during training

2.3 Analysis of autoencoder entropy

An autoencoder is a neural network that takes in an input and and imposes a bottleneck in the network which forces a representation of the original input within a compressed number of bits.

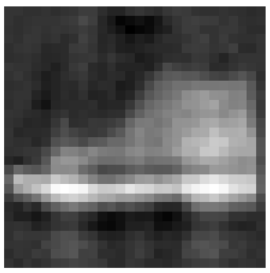
The encoded sequence is then reconstructed through further layer(s) back into an output layer of equal size to the input layer.

The reconstruction error measures the difference between the original image and the output using a loss function such as MSE.

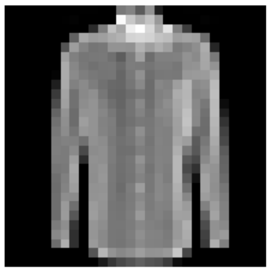
The autoencoder learns weights in its encoding (before bottleneck representation) and decoding (after bottleneck) that give the most efficient encoding of the image that retains only the most relevant information through minimising reconstruction error[2].

(a) Input image

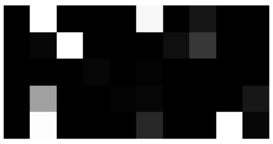
(b) 50 neuron activation representation



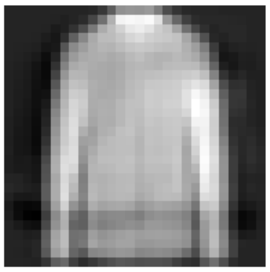
(c) Reconstructed image



(d) Input image



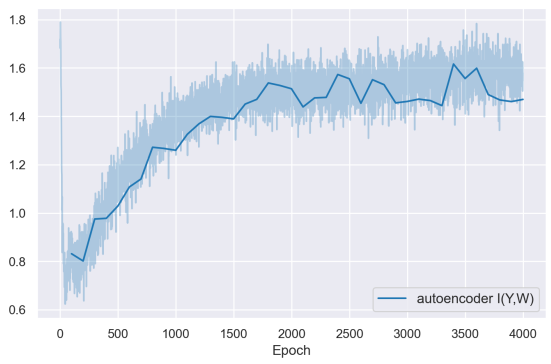
(e) 50 neuron activation representation



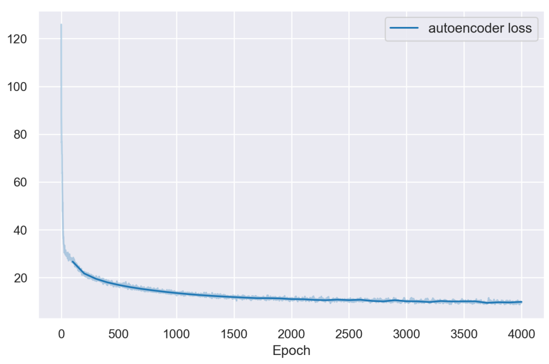
(f) Reconstructed image

Figure 11: Input images, bottleneck encodings, reconstructed images

The autoencoder has a 50 neuron bottleneck. Figure 11 demonstrates the autocoder in action. This shows that the power of autoencoders is a result of their ability to remove noisy and extraneous data.



(a) $I(Y, W)$ during training for the autoencoder



(b) MSE loss for the autoencoder

The mutual information between the brightest corner W and the hidden layer Y increases throughout training, shown in figure 12a. The increase in $H(Y, W)$ is a result of the autoencoder encoding information which approximates to the brightest corner throughout training. The autoencoder does not need to encode advanced features and instead aims to remove noise which is why this property may remain relevant throughout training.

References

[1] Ivan Chelombiev, Conor Houghton, and Cian O'Donnell. *Adaptive Estimators Show Information Compression in Deep Neural Networks*. 2019. eprint: arXiv:1902.09037.

[2] Ian Fischer. *The Conditional Entropy Bottleneck*. 2020. eprint: arXiv:2002.05379.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[4] J.F. Kolen and J.B. Pollack. "Backpropagation without weight transport". In: *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*. Vol. 3. 1994, 1375–1380 vol.3. DOI: 10.1109/ICNN.1994.374486.

[5] Timothy P. Lillicrap et al. "Backpropagation and the brain". In: *Nature Reviews Neuroscience* 21.6 (2020), pp. 335–346. DOI: 10.1038/s41583-020-0277-3. URL: https://doi.org/10.1038/s41583-020-0277-3.

[6] Timothy P. Lillicrap et al. "Random synaptic feedback weights support error backpropagation for deep learning". In: *Nature Communications* 7.1 (2016), p. 13276. DOI: 10.1038/ncomms13276. URL: https://doi.org/10.1038/ncomms13276.

[7] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[8] João Sacramento et al. "Dendritic cortical microcircuits approximate the backpropagation algorithm". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/1dc3a89d0d440ba31729b0ba74b93a33-Paper.pdf.

[9] Ravid Shwartz-Ziv and Naftali Tishby. *Opening the Black Box of Deep Neural Networks via Information*. 2017. eprint: arXiv:1703.00810.