Information Processing and the Brain – Coursework

Erich-Robert Reinholtz

er17660

PART 1

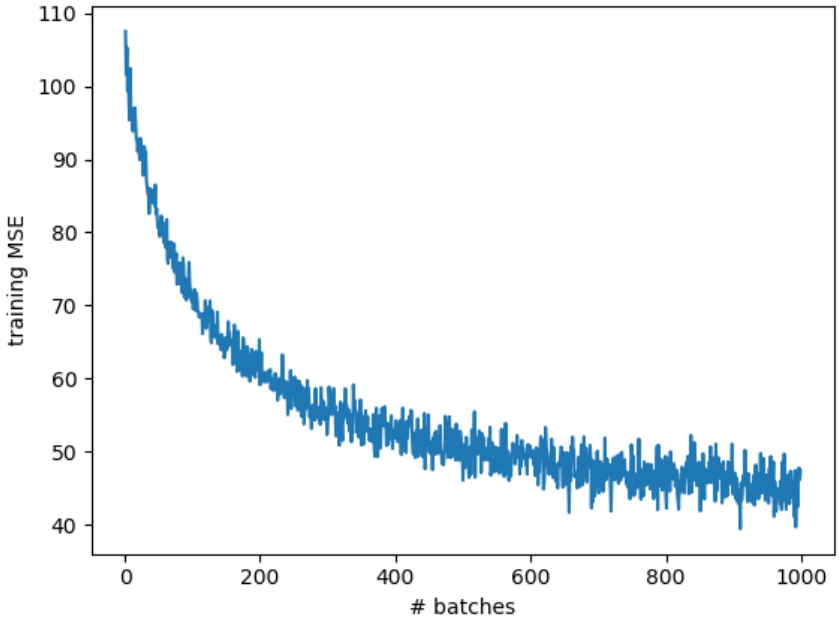
1.

After the parameters are set, the MNIST data is loaded. The inputs are normalized and the labels are converted into one hot encoding.

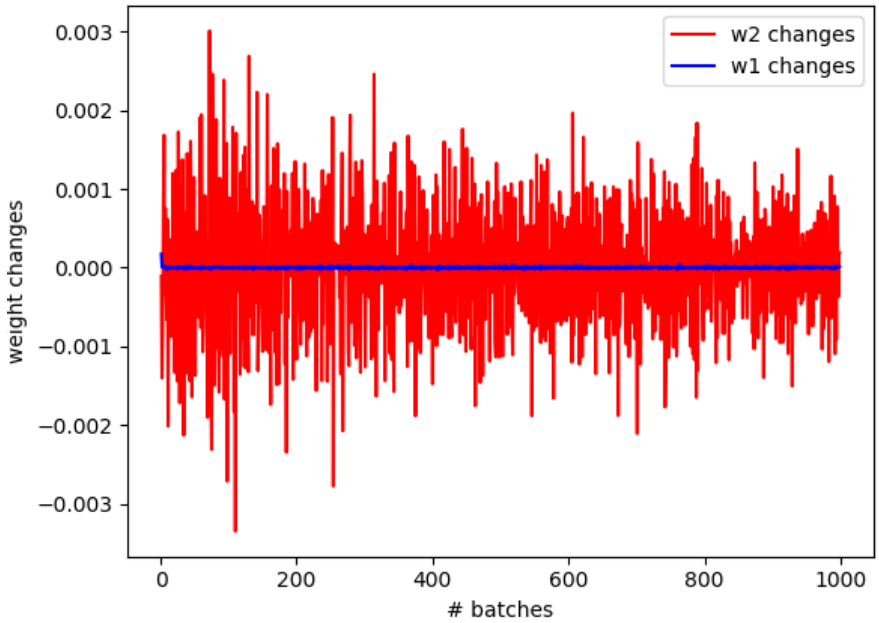
```
65
66  def load_data():
67      mndata = mnist.MNIST('./sets')
68
69      train_images, train_labels = mndata.load_training()
70      test_images, test_labels = mndata.load_testing()
71
72      train_one_hot = []
73      for i in train_labels:
74          one_hot = [0] * 10
75          one_hot[i] = 1
76          train_one_hot.append(one_hot)
77
78      train_images = np.array(train_images)/255
79      train_one_hot = np.array(train_one_hot)
80
81      test_one_hot = []
82      for i in test_labels:
83          one_hot = [0] * 10
84          one_hot[i] = 1
85          test_one_hot.append(one_hot)
86
87      test_images = np.array(test_images)/255
88      test_one_hot = np.array(test_one_hot)
89
90      return train_images, train_one_hot, test_images, test_one_hot
91
```

The neural network is then initiated, similarly to lab 1. The iteration through the number of batches then begins and, at each step, a new batch is generated and trained.

The calculated loss is then added onto a list, which is eventually plotted, after the iteration is over. Hence, a learning curve such as the following is obtained:



This curve shows that, as training takes place, there is less error accumulated, so the algorithm is successfully learning.
If we are to plot the weight changes, we can clearly see a difference between w1 and w2:



From this graph, we can see that more is learnt in the demeanour from the hidden layer to the output, hence the respective weights face a higher level of change. At the same time, there is no need for much of the image to be inscribed, in the process from input to the hidden layer, so the w1 values do not change as much.

2.

**Information derived from https://www.nature.com/articles/ncomms13276,https://www.sciencedirect.com/science/article/abs/pii/S0959438818300485?dgcid=rss_sd_all as well as lecture slides*

a. Weight transport

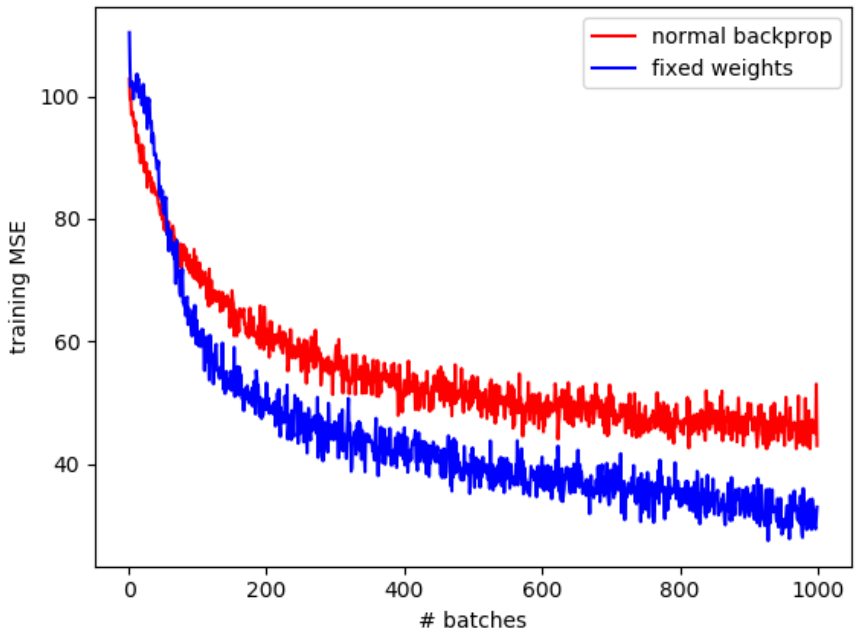
The backpropagation algorithm makes use of each neuron's synaptic weights, which are multiplied with error signals. This mechanism is widely considered to be impossible in a real brain, because, when it comes to backpropagation, precisely symmetric connectivity is considered unachievable.

In order for an appropriate backpropagation sequence to work, each neuron within the hidden layer would have to have specific and accurate knowledge of all the incoming synapses.

Certain papers, such as the one cited above, suggest that this particular assumption, of the requirement of symmetric connectivity, is actually not a necessity for the algorithm.

The alternative proposed consists of patterns of arbitrary connectivity. The desired algorithm is intended to avoid any transport of synaptic weights through a "soft alignment" between the backward and the forward ones. Across multiple layers, this would result in a practical flux of error information, neuron by neuron.

Below, I have simulated the difference between the normal backprop and one in which weight transport can't be carried out, so the weights remain fixed.

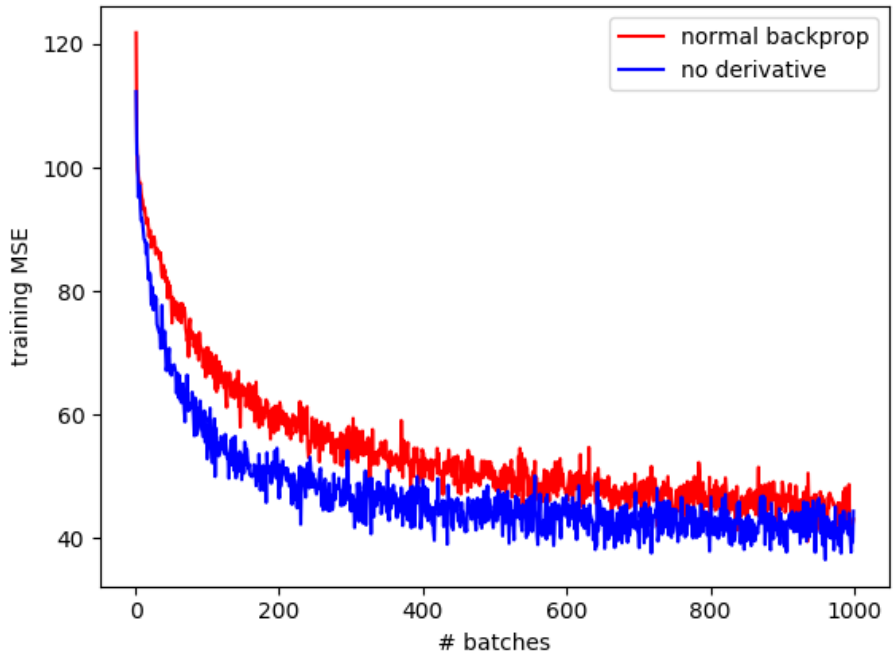


b. The need for derivative of the activation function

The second issue raised is the reliance of the backpropagation algorithm onto the calculation of the derivative of the activation function.

The main dissociation from reality is that actual physical neurons would not be able to calculate their own differentiated value. The outcome variable that can influence the functionality is the effect onto the resulted gradient direction.

Similarly to the first implication mentioned before, that of the weight transport, the solution for this problem also hangs onto the effects of this issue not being critical towards the overall performance.



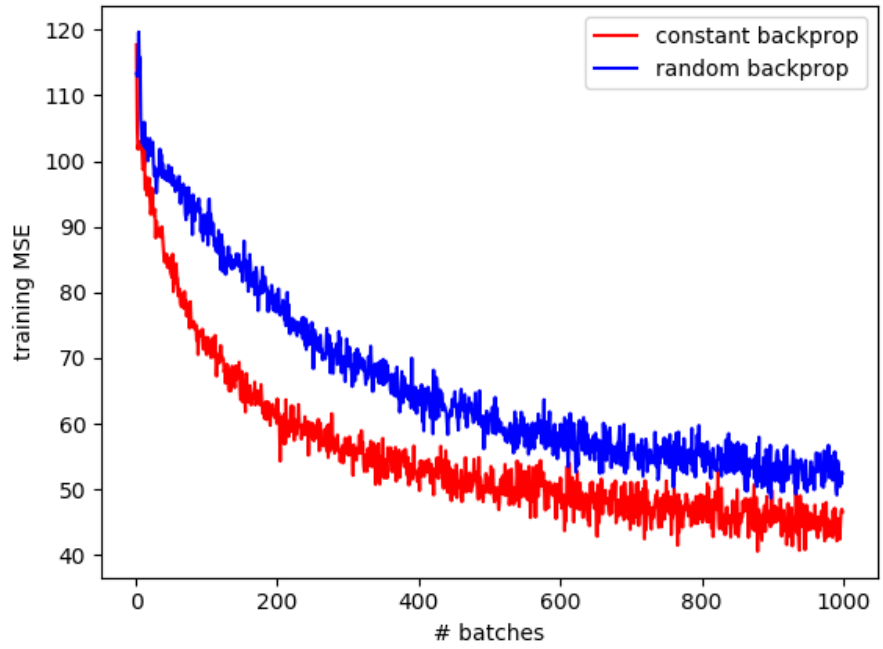
c. Two phase learning

The third issue relates to the two phases of backprop: the feedforward pass of the activity, respectively the backward pass of errors.

The main issue with this is that, in real brains, there is little sign of a clear segregation between the two phases. However, there is evidence of certain aspects, though inconclusive, which could hint towards this.

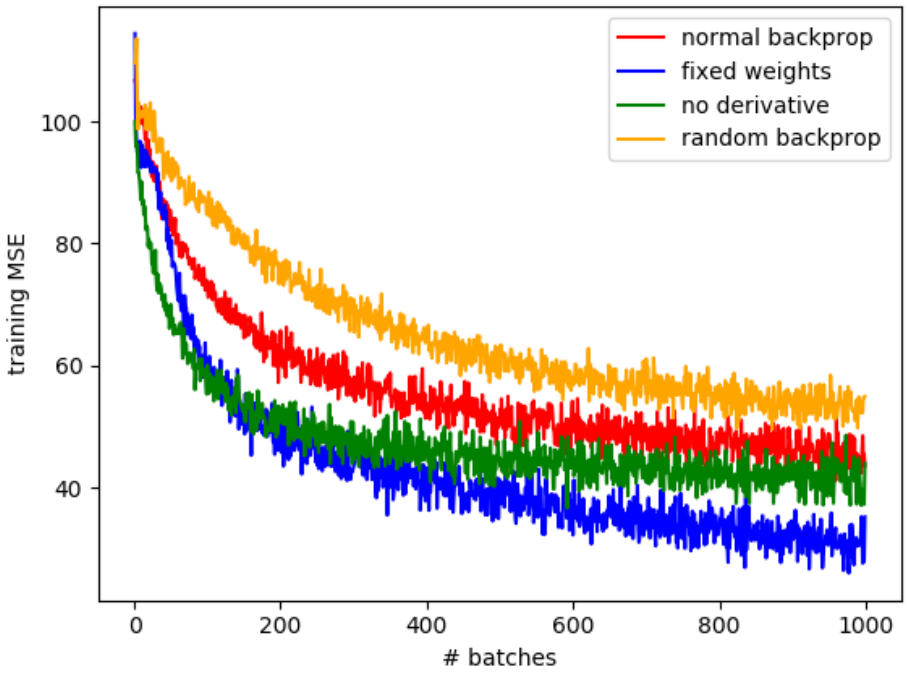
For example, it has been proven that certain parts of the brain feature “clock-like phasic activity”. However, this is does not exhibit an exact replica of the wanted behaviour.

If each neuron is associated to a respective sole voltage, credit information will be joined with other feedback signals.



Conclusion

In conclusion, if we were to overlay all of the graphs into one, we can see that, despite some differences, the outcoming shapes tend to follow the same patters:



3. Advantages:

In comparison to unsupervised learning, supervised learning provides the element of viability and practicality. One of the reasons is that previous occurrences allow for enhancements of the overall performance. Another reason is the ability to computationally solve many realistic and actual issues, applicable in real life.

The key difference between supervised learning and the one alternative paradigm, reinforcement learning, is the presence of ground truth. Hence, the learning process can be way more efficient, especially for non-complex tasks.

4. Disadvantages:

One of the disadvantages of supervised learning methods is the necessity of providing not only input, but also respective labelled output. These can prove to be costly to supply, as having the labels signifies a need for human intervention.

Another disadvantage is the fact that, even after providing the aforementioned tuples of data, the algorithm becomes constrained by them. On the other hand, unsupervised leaning, for example, would have a higher degree of flexibility and adaptability and could spot patterns that had been unanticipated. Respectively, reinforcement learning, after the occurrence and correction of an error, would have a much higher chance of avoiding it in the future, similarly to human brains' workflow.

Many possible improvements have been proposed, *one example of which is "adapting the problem to the learner": for instance, finding the means to generate a function that is easier for the algorithm to learn by relabelling the targets in the training sets.

**Source: https://www.worldscientific.com/doi/abs/10.1142/S0129065709001793*

PART 2

1.

The first step I took in order to solve this task was binarising the neuron activity. I chose **0.5** as a threshold, used a list of dictionaries for the evidence of dependencies on labels and one extra dictionary for the independent frequency of sequences.

```
58
59      for i in range(len(h)):
60          activity = h[i]
61          aseq = ''
62
63          # binarising neuron activity
64          for a in activity:
65              if a > 0.5:
66                  aseq += '1'
67              else:
68                  aseq += '0'
69
70          # dictionary for each label
71          if aseq in dicts[ds[i]]:
72              dicts[ds[i]][aseq] += 1
73          else:
74              dicts[ds[i]][aseq] = 1
75
76          # frequency dictionary regardless of input
77          if aseq in freqDict:
78              freqDict[aseq] += 1
79          else:
80              freqDict[aseq] = 1
81
```

Afterwards, I turned the information into probabilities. I obtained *H(seq)* as well as *H(seq|label)* using the formula ***sum(-p*log(p))***.

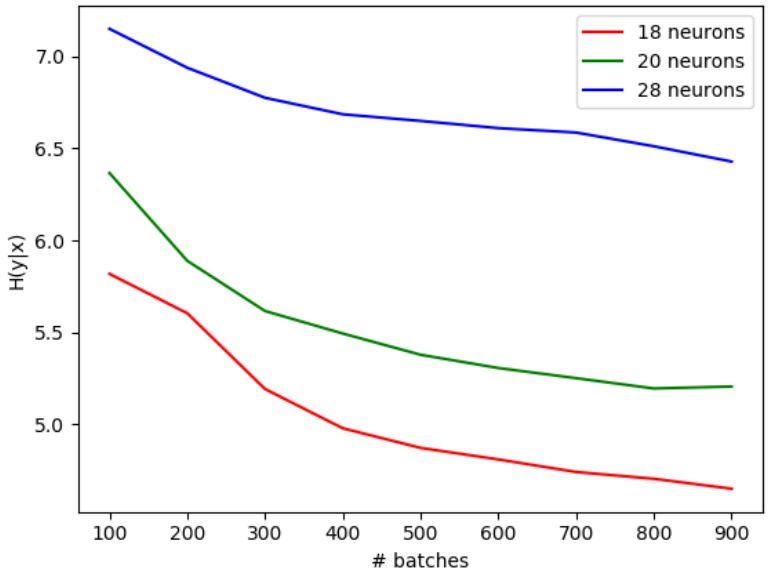
```
174         # H(y|x)
175         ps = []
176
177         for d in range(10):
178             hgivenl = 0
179             values = dicts[d].values()
180             n = 0
181
182             for v in values:
183                 n += v
184
185             for v in values:
186                 hgivenl -= float(v)/n * np.log(float(v)/n)
187
188             ps.append(hgivenl)
189
190         hyx.append(np.average(ps))
191
192         # H(y)
193         hseq = 0
194         values = freqDict.values()
195         n = 0
196
197         for v in values:
198             n += v
199
200         for v in values:
201             hseq -= float(v)/n * np.log(float(v)/n)
202
203         hy.append(hseq)
```

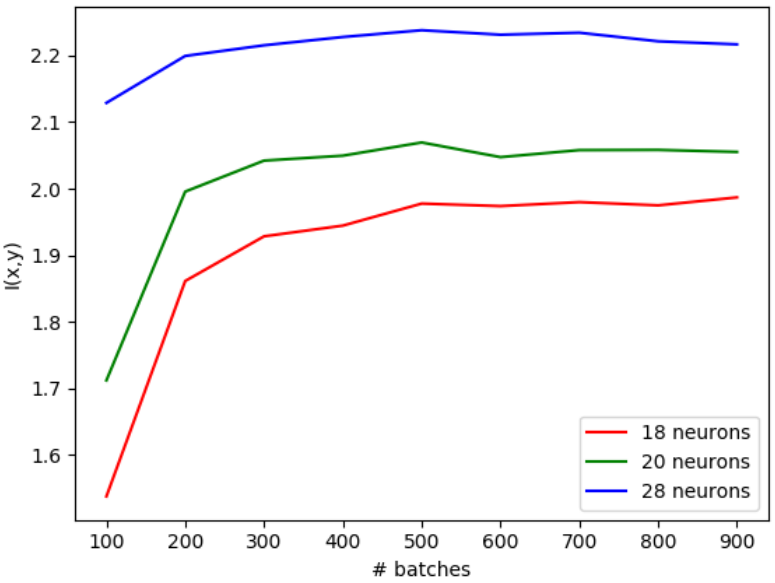
Obtaining the mutual information was then trivial, only requiring the subtraction formula and applying the data already acquired:

```
# I(labels, sequences) = H(seq) - H(seq|labels)
ixy.append(hseq - np.average(ps))

# emptying dictionaries
dicts = [dict() for i in range(10)]
freqDict = dict()
```

The process, including the emptying of dictionaries, occurs once every 100 batches. Generally, when plotting the *H(Y|X)* and *I(X, Y)*, the following shapes are obtained:

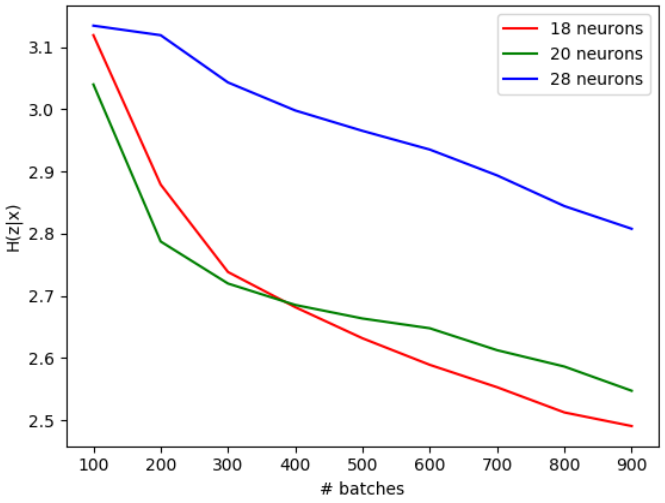


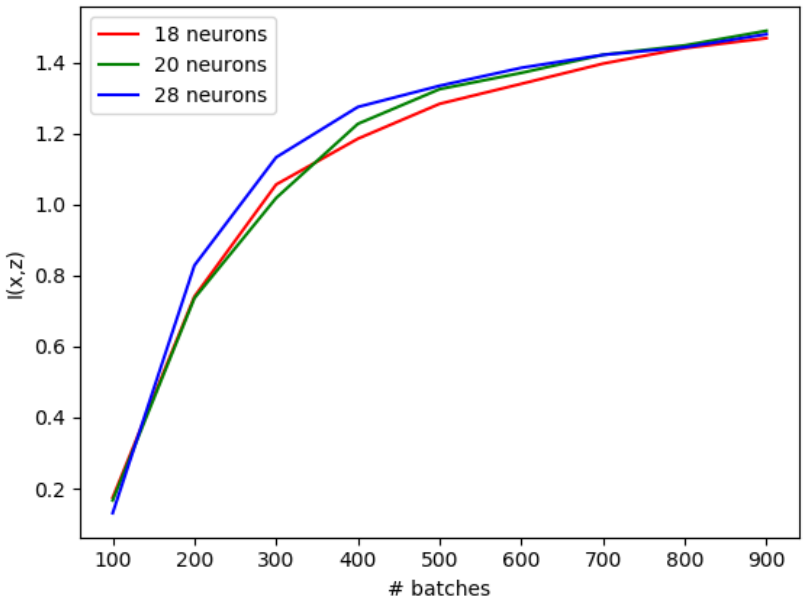


We can see that, on a general level, as training happens, the hidden neuron activity depends less and less on the input data, which shows that the algorithm is learning. Hence, the mutual information between the two variables features an increase.
The number of neurons in the hidden layer also has an effect on these patterns:

A varying number of neurons can lead to differences in the conditional entropy values, though the patterns tend to follow a similar path. When a higher number of neurons are active, the mutual information value appears to reach the conversion point quicker.

We can run this process similarly with regards to the changes in *H(Z|X)* and *I(X,Z)*. In my case, *Z* represents the most confident prediction.





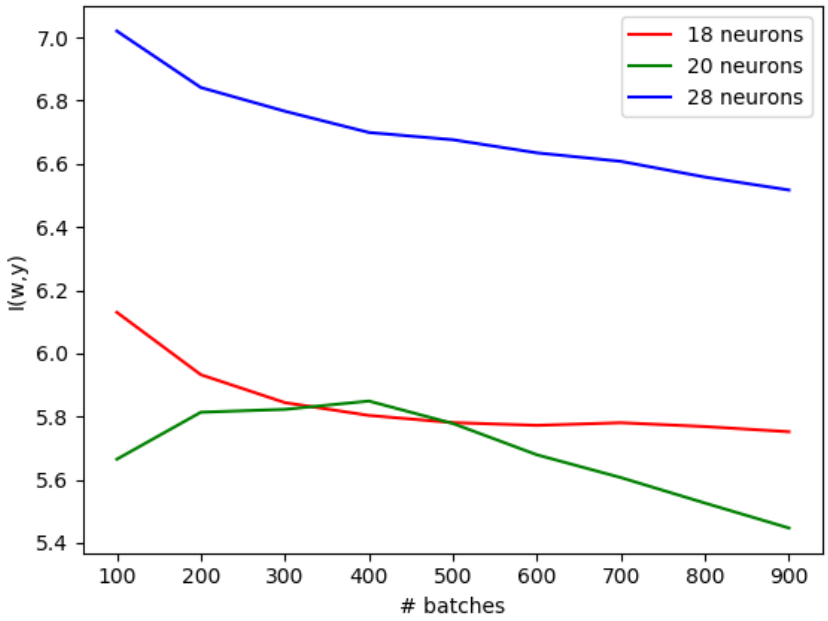
The conclusion drawn mimics the ones mentioned previously: The algorithm slowly grows independence from the labels, the more training and learning it undergoes.

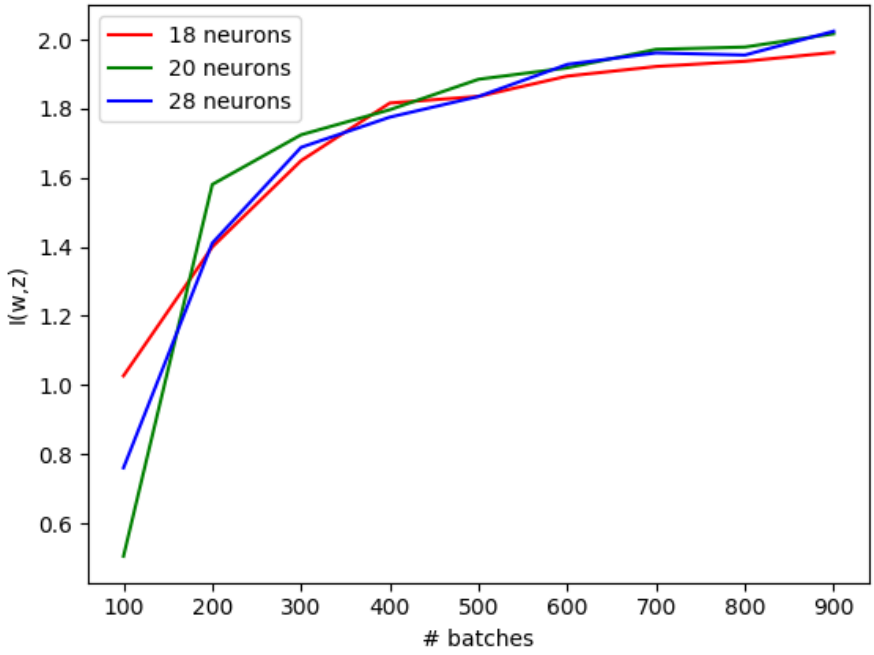
2.

For this final task, I used a similar code to 2.1, comparably obtaining *I(w,a)* (where *a* is *y* or *z*) from the formula: *I(w, a) = H(a) – H(a|w)*. Likewise, I created a dictionary for the sequence frequency depending on w, and another dictionary for the independent frequency.
The quadrants were obtained from each image and then filtered to obtain the one with the most amount of white. Since white represents the largest colour value, all that had to be done to identify the quadrant was extracting the one with the highest summation of values:

```
107
108     qs = []
109
110     for i in inputs:
111
112         resShapeInput = (28, 28)
113         resShapeQuad = (196,1)
114
115         i = np.reshape(i, resShapeInput)
116
117         q1 = np.reshape(i[:14,:14], resShapeQuad)
118         q2 = np.reshape(i[:14,14:], resShapeQuad)
119         q3 = np.reshape(i[14:,:14], resShapeQuad)
120         q4 = np.reshape(i[14:,14:], resShapeQuad)
121
122         q1_sum = np.sum(q1)
123         q2_sum = np.sum(q2)
124         q3_sum = np.sum(q3)
125         q4_sum = np.sum(q4)
126
127         q_max = np.argmax([q1_sum, q2_sum, q3_sum, q4_sum])
128
129         qs.append(q_max)
130
```

Same as before, the threshold is *0.5* and *Z* represents the most confident prediction. The trends showcase slight decreases in I(w,y), as well as high variations depending on the number of hidden layer neurons. I(w,z) features an increase, which keeps consistency when the hidden size varies.


18 neurons
20 neurons
28 neurons
I(w,y)
# batches


18 neurons
20 neurons
28 neurons
I(w,z)
# batches

Sources:

https://www.nature.com/articles/ncomms13276,https://www.sciencedirect.com/science/article/abs/pii/S0959438818300485?dgcid=rss_sd_all

https://www.worldscientific.com/doi/abs/10.1142/S0129065709001793

IPB lecture materials