Ronan Dale

Experimenting with the Hyperparameters of a Recurrent Neural Network

About RNNs

A key advantage to using an RNN is their ability to work with variable length sequences, both as inputs and outputs so their relationship can be anything from ‘One-to-One’ to ‘Many-to-Many’. In Natural Language Processing, there are two common use cases for an RNN. Sentiment Analysis uses a ‘Many-to-One’ RNN as multiple words are passed as input, whereas the output is a single classification. In Translators, a ‘Many-to-Many’ RNN is used as both the input and output are a collection of words.

What does the Maths look like?

- Matrices
- Vectors
- Trigonometric Functions

Consider a ‘Many-to-Many’ RNN; X_0, X_1 ... X_n are the inputs and Y_0, Y_1 ... Y_n are the outputs. X and Y are *vectors*. In the hidden layer, there is a *vector* ‘h’.

h_n is calculated using h_{n-1} and X_n.

Y_n is calculated using h_t.

A typical RNN uses the same 3 weights for each step. All of these are *matrices*.

W_{xh} is used for x_n to h_n links (Input Layer to Hidden Layer)

W_{hh} is used for h_{n-1} to h_n links (Inside Hidden Layer)

W_{hy} is ised for h_n to y_n links (Hidden Layer to Output Layer)

There are also two bias neurons.

b_h is used when calculating h_n

b_y is used when calculating Y_n

Formulae:

$h_n = \tanh((W_{xh}*x_n)+(W_{hh}*h_{n-1})+ b_h)$

$y_n = (W_{hy}*h_n) + b_y$

Optimising an RNN: Phase 1

‘Learning Rate’ refers to by how much the network adjusts a parameter

Aim of Phase 1 is to modify the *Learning Rate* and observe its effect on the accuracy of the model at given intervals. Optimiser is defaulted to Adam for this phase.

Vary the Learning Rate from 0.0001, 0.001, 0.01 and 0.1.

Hypothesis:

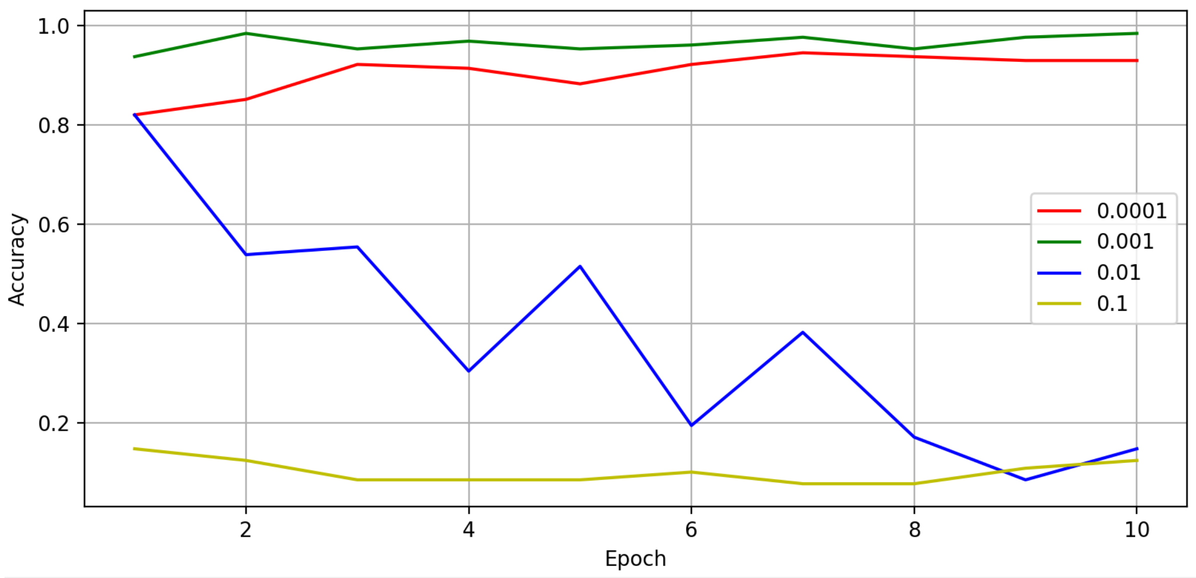
0.0001: Converges in later epochs.

0.001: Converges in later epochs

0.01: Diverges quickly.

0.1: No improvement or detriment to performance.

Results:



Phase 2 – Adversarial Examples

https://ml.berkeley.edu/blog/posts/adversarial-examples/

Now that the model is producing results, it's important to see if there is any commonality between the images it incorrectly categorised. Are there pairs of numbers which are often mistaken for each other? If so, can we isolate enough of these incorrect answers to produce an adversarial training dataset which may further improve the accuracy achieved on the testing dataset?

Targeted Adversarial Examples add carefully constructed noise over an image which would normally fall correctly into Class A with a high confidence, but due to the noise is now incorrectly categorised as Class B with a very high confidence.

Non-Targeted Adversarial Examples are ones without any additional noise which simply trick the Neural Network into incorrectly categorising an image. To do this, we generate an image that has been designed to make the network give a certain output.

Cost Function: $\sum(x_n - y_n)^2$ takes the sum of (actual output – expected output)^2. A high output shows a low accuracy and the inverse of that is true. The average of all outputs over the full training process is indictative of the quality of the network. Finding the local minimum of a cost function involves beginning at a random x co-ordinate on a graph and working out if the gradient is positive or negative. A positive gradient means that the local minimum is to the left of the current x position and a negative gradient means that the local minimum is to the right. As you move closer to the local minimum, your movement reduces so that you don't overshoot and end up passing it. This process is then scaled up for each neuron in the network.

Consider the magnitude of each value in the column vector to be an instruction. If the value of w_3 corresponds to the adjustment '-0.13', then w_3 needs to decrease slightly. However if the value of w_4 correspondes to '+1.34', then w_4 must increase greatly. This relative importance shows which neurons need to be adjusted to have the greatest overall positive effect on the result.

1. Consider the image you want to make to be a 784-dimensional vector (28*28)

2. Define a cost function:

RNN is not ideal for MNIST as it is a classification task. Implementing LSTM instead of SimpleRNN in Keras yields an improved accuracy of +5% by simply changing the type of network and no other code.

LSTMs can learn long-term dependencies.

Why do Simple RNNs underperform on the MNIST dataset compared to CNNs?

- Shallow vs Deep: Simple RNN is a shallow model and tends to only have 3 weights to adjust. This makes it less suited for a task in which better performance is achieved by 'learning' common patterns that exist in all images of a certain digit. Deep Models like CNN have considerably more weights to adjust because of their multilayer hidden layers.
- RNNs are suited to work with variable-length data. If the input was clusters of handwritten digits eg; '7834', an argument could be made to use an RNN but CNNs are

Key Takeaways about the LSTM RNN models.

- LSTM models are used for large vocabulary speech recognition and general speech-to-text tasks. It can also be used for machine translation and multilingual language processing. By combining an LSTM network with a CNN, automatic image captioning is possible (the AI recognises the image and assigns it text-based metadata based on what it shows).

Ronan Dale

Which Batch Size yields the highest Accuracy?

Batch Size = 32

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.906	0.875	0.938	0.969	1.000	0.943
0.001	0.969	1.000	1.000	0.969	0.969	0.945
0.01	0.125	0.125	0.094	0.094	0.094	0.098
0.1	0.125	0.094	0.188	0.156	0.188	0.101

Batch Size = 64

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.812	0.922	0.906	0.922	0.891	0.940
0.001	0.922	0.922	0.984	0.938	0.938	0.969
0.01	0.516	0.312	0.375	0.375	0.125	0.098
0.1	0.141	0.125	0.219	0.094	0.172	0.101

Batch Size = 128

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.797	0.859	0.891	0.906	0.906	0.924
0.001	0.930	0.969	0.961	0.961	0.953	0.965
0.01	0.727	0.656	0.039	0.422	0.477	0.438
0.1	0.141	0.125	0.102	0.133	0.109	0.113

Batch Size = 256

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.001	0.754	0.867	0.902	0.891	0.910	0.914
0.001	0.902	0.957	0.949	0.965	0.961	0.952
0.01	0.844	0.875	0.898	0.809	0.371	0.408
0.1	0.078	0.109	0.094	0.102	0.148	0.103

Key Takeaways about altering the batch size.

Even though the (Batch Size = 32, Learning Rate = 0.001) has the highest average accuracy in training overall, with such small sample sizes, it's highly likely that the samples it scored 1.000 on contained no adversarial examples at all. Therefore a higher batch size includes more adversarial examples and is a more representative test.

By doubling the batch size from 128 to 256, the test accuracies stabilise, indicating once again that bigger data pool = more representative accuracy. However even at lower batch sizes, the lower learning rates outperform the higher ones considerably.

Ronan Dale

Now consider the Number of Neurons:

Num. Neurons = 32

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.273	0.422	0.512	0.523	0.547	0.543
0.001	0.641	0.738	0.762	0.797	0.785	0.809
0.01	0.727	0.875	0.887	0.875	0.906	0.893
0.1	0.379	0.121	0.086	0.070	0.105	0.091

Num. Neurons = 64

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.477	0.586	0.723	0.730	0.793	0.801
0.001	0.777	0.848	0.910	0.926	0.957	0.915
0.01	0.898	0.891	0.930	0.883	0.855	0.830
0.1	0.121	0.121	0.113	0.133	0.109	0.096

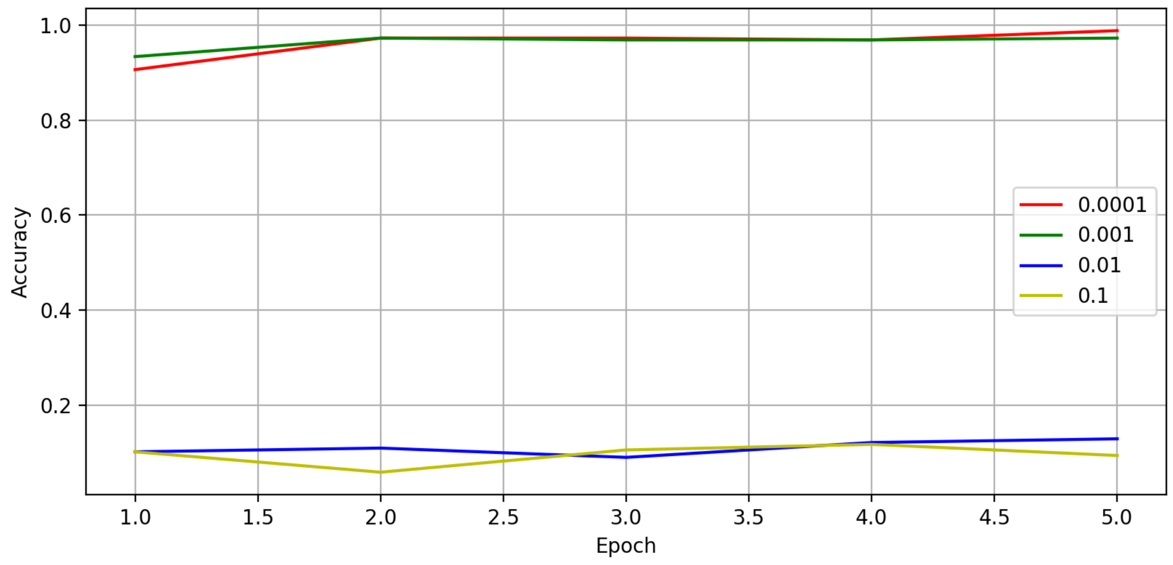
Num. Neurons = 128 is 'Batch Size = 256' from above. Around this stage, the training time starts to increase greatly although I prioritised finding the combination with maximum accuracy in this experiment. I'm deducing that the larger number of neurons means that the network remembers more patterns but this also increases the size of the network so it takes longer for the data to be passed from Input Layer to Output Layer.

Num. Neurons = 256

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.883	0.863	0.938	0.953	0.926	0.950
0.001	0.922	0.953	0.969	0.965	0.973	0.970
0.01	0.617	0.629	0.656	0.621	0.645	0.606
0.1	0.098	0.117	0.129	0.113	0.129	0.096

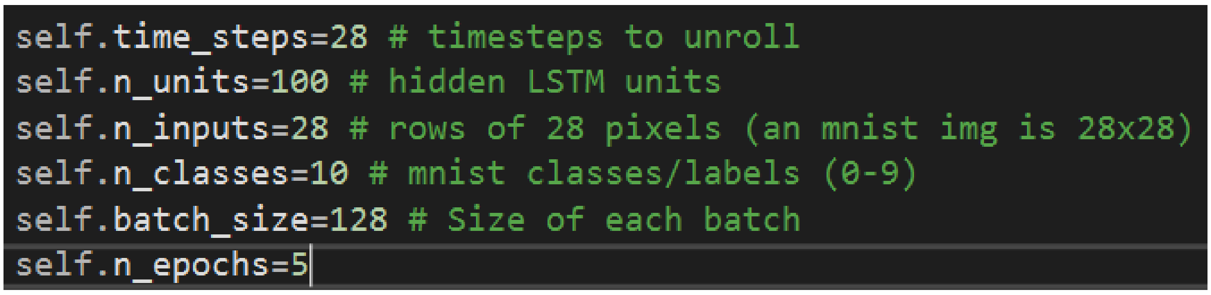
Num. Neurons = 512

	Train 1	Train 2	Train 3	Train 4	Train 5	Test
0.0001	0.906	0.973	0.973	0.969	0.988	0.964
0.001	0.934	0.973	0.969	0.969	0.973	0.971
0.01	0.102	0.109	0.090	0.121	0.129	0.089
0.1	0.102	0.059	0.105	0.117	0.094	0.103



The final table shows that although the number of neurons was doubled, the increase in accuracy (+0.001%) simply isn't worth doubling the amount of time taken to train the network. Therefore, increasing the number of neurons causes the accuracy to converge towards a maximum and it's best to find the size with the highest accuracy and lowest training time. The graph shows that the learning rate is almost a binary choice: high or low. If high is selected, accuracy is consistently poor however, if low is selected, then accuracy is consistently high.

Measuring LSTM Performance by varying the optimiser



```
self.time_steps=28 # timesteps to unroll
self.n_units=100 # hidden LSTM units
self.n_inputs=28 # rows of 28 pixels (an mnist img is 28x28)
self.n_classes=10 # mnist classes/labels (0-9)
self.batch_size=128 # Size of each batch
self.n_epochs=5
```

Optimiser is rmsprop.

LSTM Units	Train 1	Train 2	Train 3	Train 4	Train 5	Test
100	0.7789	0.9338	0.9595	0.9699	0.9761	0.9745
200	0.8060	0.9477	0.9679	0.9775	0.9826	0.9783

Optimiser is adam.

LSTM Units	Train 1	Train 2	Train 3	Train 4	Train 5	Test
100	0.7887	0.9390	0.9593	0.9680	0.9747	0.9723
200	0.8241	0.9518	0.9666	0.9755	0.9821	0.9787