STAT448 Big Data - Assignment 3 – nki38 – Noah king - 96851177

Q1:

1. We compute the output of the following 1-layer NN given X=$\begin{bmatrix}1\\2\end{bmatrix}$, as inputs.

2. We apply the following to each node of each layer (excepting the input):

$$w \cdot X + b = output$$

Noting that our model already has the following weights, biases:

$$W_1 = \begin{pmatrix}5 & 10\\1 & 0\\3 & 4\end{pmatrix}, \quad b_1 = \begin{pmatrix}5\\-5\\-1\end{pmatrix}, \quad W_2 = \begin{pmatrix}1\\2\\1\end{pmatrix}, \quad b_2 = -27$$

3. We compute on h1,h2,h3:

$$out_{h1} = W_{1[1]} \cdot X + b_{1[1]} = (5 \times 1) + (10 \times 2) + 5 = 5 + 20 + 5 = 30$$
$$out_{h2} = W_{1[2]} \cdot X + b_{1[2]} = (1 \times 1) + (0 \times 2) - 5 = 1 + 0 - 5 = -4$$
$$out_{h3} = W_{1[3]} \cdot X + b_{1[3]} = (3 \times 1) + (4 \times 2) - 1 = 3 + 8 - 1 = 10$$

4. Then apply relu:

$\begin{bmatrix}h1\\h2\\h3\end{bmatrix} = \begin{bmatrix}30\\-4\\10\end{bmatrix}$ …. Then apply: Relu: g(z) = max(0, z).. i.e. $\begin{bmatrix}h1\\h2\\h3\end{bmatrix} = \begin{bmatrix}30\\0\\10\end{bmatrix}$

Define [h1,h2,h3] as h_output

5. We apply the same logic to the output layer (i.e. output = w2*h_output + b)…

$\begin{bmatrix}w21\\w22\\w23\end{bmatrix} = \begin{bmatrix}1\\2\\1\end{bmatrix}$, b2 = -27 … $f\_output = (1 \times 30) + (2 \times 0) + (1 \times 10) + (-27) = 13$

6. The activation (sigmoid function) is applied at f:

$g(x) = 1(1 + e\text{^}x)$ -> $g(x) = 1(1 + e^{-13})$ -> ≈ 0.999996

7. Fx will output ≈ *0.999996* with input X=$\begin{bmatrix}1\\2\end{bmatrix}$,

Q2:

Preamble:
The fashion MNIST dataset contains 70’000 greyscale images across 10 labels of 28*28 images representing different clothing accessories. In terms of complexity, this is a *step up* from the MNIST numeric dataset due to complexities in the grayscale parts of the image (MNIST symbols are black and white only).

In terms of visualising the networks we create, the .summary() keras method is simply screenshotted.

Such an image set is particularly useful for training classifiers or benchmarking ML systems. We use this dataset to explore the effects of different ML approaches and changes to the architecture / hyperparameters on the loss (and therefore performance) of the model.

All of our models are trained on 10 epochs. Intuitively; 10 epochs is likely insufficient to converge to any minimum; but is sufficient to show the differences between our approaches and the relative performance of the models.

All of the models are being computed in google colab; as such we are limited by the available gpu resource on this platform (10 epochs for sanity too).

We will generate a simple model, followed by a model containing convolutional layers, we will then tune the parameters of this model. We will then consider changes to the architectures, such as introducing dropout and L2 regularization and their effects on the network performance.

Baseline:

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 32)	25,120
dense_1 (Dense)	(None, 10)	330

Total params: 25,450 (99.41 KB)
Trainable params: 25,450 (99.41 KB)
Non-trainable params: 0 (0.00 B)

Where we have two densely connected layers. The model takes in the *flattened* image (28*28 = 784) image into 32 neurons. Those 32 are connected to 10 output neurons and their relative weights are just trained through backpropagation given some loss metric against our test set on each epoch. The hidden layer here simply has a sigmoid activation function and the output layer softmax (i.e. choose the most *probable* class (highest value)).

We will call this model our *baseline*. This is not to indicate any positive or negative aspects of the performance; but simply that this is a basic model.

Models:
We consider the following broad architectural changes:

Conv2d: 1
Convolutional Neural Networks are a perfect fit for image classifiers such as our intended model. CNN layers work by passing a filter over an input image and *highlighting* different image features (i.e. the filter would lead more prevalent image features to be most visible in the convolved image). As such, we consider a first model which includes two Conv2d layers (i.e. a convolutional layer in keras).

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 28, 28, 1)	0
conv2d_4 (Conv2D)	(None, 26, 26, 32)	320
conv2d_5 (Conv2D)	(None, 24, 24, 64)	18,496
flatten_2 (Flatten)	(None, 36864)	0
dense_8 (Dense)	(None, 64)	2,359,360
dense_9 (Dense)	(None, 10)	650

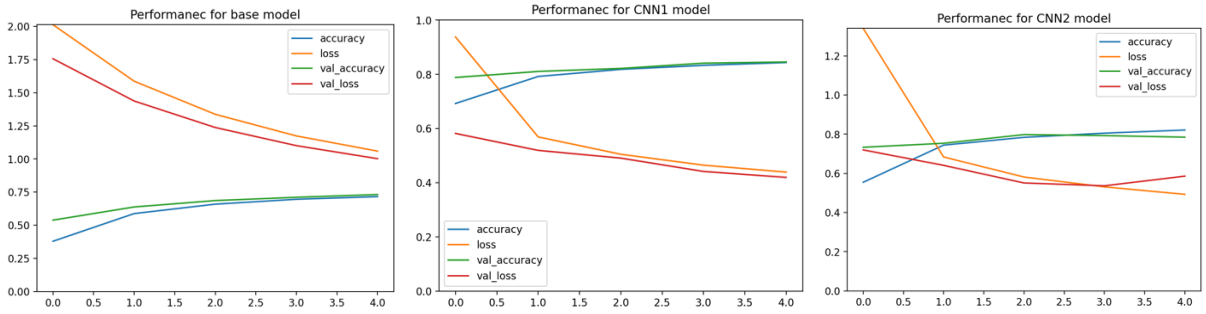
Where the two *Conv2D* layers introduce 32 and 64 channels respectively; before being flattened and the final two dense layers effectively acting in the same way as the **base model.** Note; across the board we have used *Relu* for the activation layer (i.e a function which returns zero until zero, then return y=x for values >0).

Conv2d: 2:
We can decrease the verbosity of our input images in Convolutions using strides: Just like with a typical CNN, the filter *detects* (or rather, highlights) image features given the product of the filter process; here we reduce the size of the image by *increasing* the number of steps that the filter moves…. The rest of the architecture is the same here

We generate this model:

Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(None, 28, 28, 1)	0
conv2d_6 (Conv2D)	(None, 26, 26, 32)	320
conv2d_7 (Conv2D)	(None, 24, 24, 64)	18,496
conv2d_8 (Conv2D)	(None, 11, 11, 128)	73,856
conv2d_9 (Conv2D)	(None, 5, 5, 256)	295,168
flatten_3 (Flatten)	(None, 6400)	0
dense_10 (Dense)	(None, 64)	409,664
dense_11 (Dense)	(None, 10)	650

Running both models, the base model on the following hyperparameters: [optimizer:'sgd', loss:'categorical_crossentrophy', lr:0.01, epochs:5] we see the following:



Model	Best tr acc	Best val acc	Best tr loss	Best val loss
Basic Model	0.7160	0.7313	1.0592	1.0020
CNN Model 1	0.8434	0.8455	0.4389	0.4194
CNN Model 2	0.8215	0.7980	0.4929	0.5364
Difference (CNN1 - CNN2)	-0.0219	-0.0475	+0.0540	+0.1169

We see that there isn't a great improvement in terms of MSE (loss) on our training set and the validation set clearly actually sees a decrease moving from the CNN1 model to CNN2. (noting 5 epochs only)
Our architecture is quite convoluted in our second convolutional model. We note that for all of the *5* hidden layers; we employ relu as an activation function. Relu is defined as:

$$f(x) = max\ (0, x)$$

A particular issue in deep neural networks is *vanishing gradient*. Wherein the outputs of a layer, and therefore the updates to the gradient diminish with each layer.

Another issue with neural networks generally is they risk *overfitting* to their training data. Extra steps like dropout layers or regularization layers can help us to overcome this.

We may be able to improve the performance of CNN2 by making changes in this space. In fact our goal for the remainder of this paper becomes, at least achieving *parity* between conv2 and conv1.

Parameters
Before we get into changing the activation functions or introducing L2 layers or indeed dropout layers consider the LR.

As expected; high learning rates lead to poor accuracy, high loss but use less compute. Lower learning rates might not necessarily improve the performance of the model but cost great performance. Interestingly, our base model converges quickly for a learning rate of 1 (which is a huge learning rate already).

We will continue to use 0.01 as the lr for the continuation of this document.

learning_rates = [10.0, 1.0, 0.5, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001] (note the diff is quite small for small lr; linear display chosen of log)

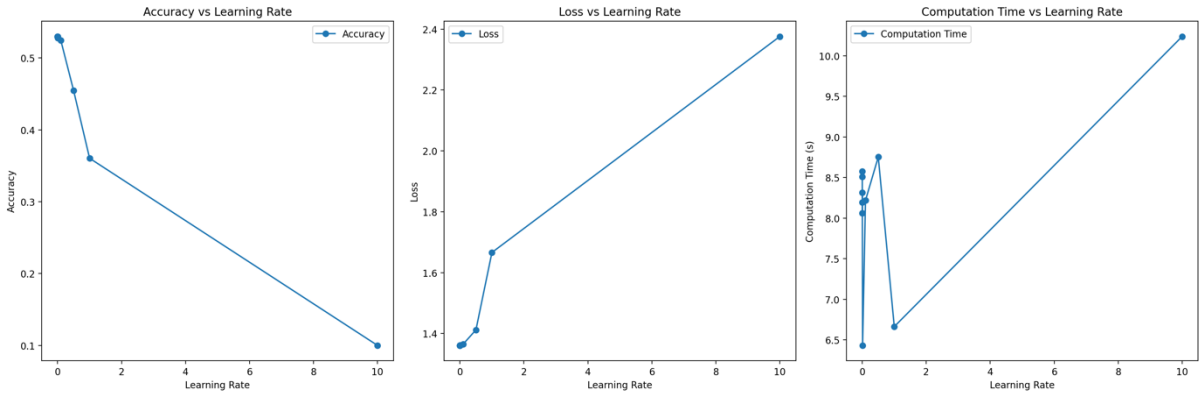


Figure 1 Compute Time, Loss, Accuracy on Basic Model against L.R.

We will just use the 0.001 LR that is the default for keras for hereon out; we would expect similar behaviour to the above on our CNN and CNN 2 models.

Batch size:
We will not alter batch sizes for our tests; however greater batch sizes may help to capture cross batch dependencies in the weights of our matrices; small batches can introduce more noise, they might not converge as quickly (for obvious reasons). However, if we were stuck in a local minimum small batches might help mitigate this with more frequent updates. We won't touch this, but it's trivial to see how we would with our models.

Comparing Optimizers
We could also compare optimizers for our model. Adam and SGD are the most prudent options. In the interst in time / compute resource, we won't pursue this, but it is straightforward to see how we would vary these and run the models.

We will consider three approaches to improving the performance of our model.

Leaky Relu

We touched briefly on how relu could contribute to our vanishing gradient problem. We choose *leak relu* instead, which is as

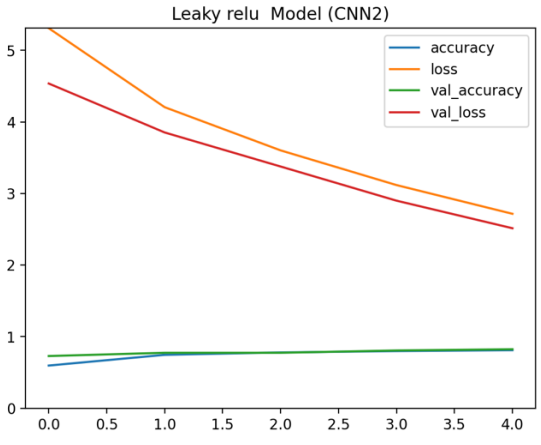
$$f(x) = \{xif\ x > 0, \alpha x\ if\ x \leq 0$$

Where alpha is some hyperparameter (determining the slope of the <0 inputs)

Our Conv2d Model with Leaky Relu replacing all Relu and the same softmax activation function performs as follows; noting that keras defines this activation function as it's own layer; yet it is interpretably the same layer.

The graphs below have the number of epochs on our Y axis. (unfortunately the notebook was cleared and rerunning to generate these is time consuming)

Layer (type)	Output Shape	Param #
reshape_8 (Reshape)	(None, 28, 28, 1)	0
conv2d_20 (Conv2D)	(None, 26, 26, 32)	320
leaky_re_lu (LeakyReLU)	(None, 26, 26, 32)	0
conv2d_21 (Conv2D)	(None, 24, 24, 64)	18,496
leaky_re_lu_1 (LeakyReLU)	(None, 24, 24, 64)	0
conv2d_22 (Conv2D)	(None, 11, 11, 128)	73,856
leaky_re_lu_2 (LeakyReLU)	(None, 11, 11, 128)	0
conv2d_23 (Conv2D)	(None, 5, 5, 256)	295,168
leaky_re_lu_3 (LeakyReLU)	(None, 5, 5, 256)	0
flatten_7 (Flatten)	(None, 6400)	0
dense_22 (Dense)	(None, 64)	409,664
leaky_re_lu_4 (LeakyReLU)	(None, 64)	0
dense_23 (Dense)	(None, 10)	650



This alteration (leaky relu) to CNN2 achieved average loss of 2.54 and 81.89% accuracy on test data in 1507s

We see the validation and test set performance on the graph on the right (and herein) given epochs. Test set vs validation also graphed.

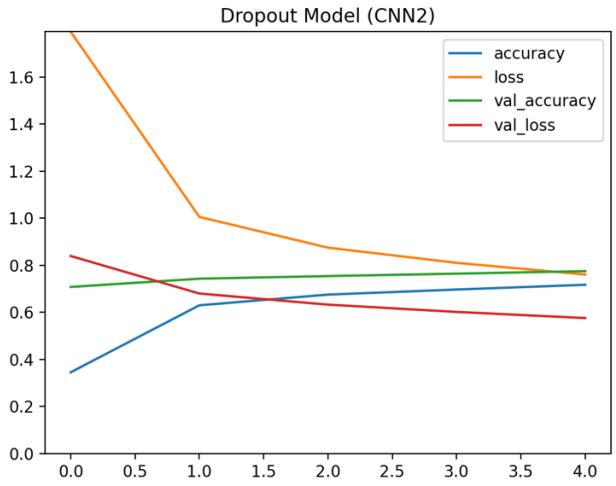
Dropout

A final step today is adding a drop out and regularization layers; as this is an architectural decision; we would typically construct the architecture algorithmically. For the purpose of this document; we are 'chucking a few' drop out layers in to the conv2d layer.

Why dropout? By randomly assigning values to some nodes at some point of the compute to zero; we ensure that there is no over reliance on a particular node in our NN; this helps to mitigate overfitting and therefore improves importance on unseen data (ideally).We use a fixed dropout rate too, but this again is a tunable hyperparameter

The architecture for Conv2 with the introduced dropout layer is as follows.

Layer (type)	Output Shape	Param #
reshape_13 (Reshape)	(None, 28, 28, 1)	0
conv2d_40 (Conv2D)	(None, 26, 26, 32)	320
dropout_15 (Dropout)	(None, 26, 26, 32)	0
conv2d_41 (Conv2D)	(None, 24, 24, 64)	18,496
dropout_16 (Dropout)	(None, 24, 24, 64)	0
conv2d_42 (Conv2D)	(None, 11, 11, 128)	73,856
dropout_17 (Dropout)	(None, 11, 11, 128)	0
conv2d_43 (Conv2D)	(None, 5, 5, 256)	295,168
dropout_18 (Dropout)	(None, 5, 5, 256)	0
flatten_12 (Flatten)	(None, 6400)	0
dense_32 (Dense)	(None, 64)	409,664
dropout_19 (Dropout)	(None, 64)	0
dense_33 (Dense)	(None, 10)	650

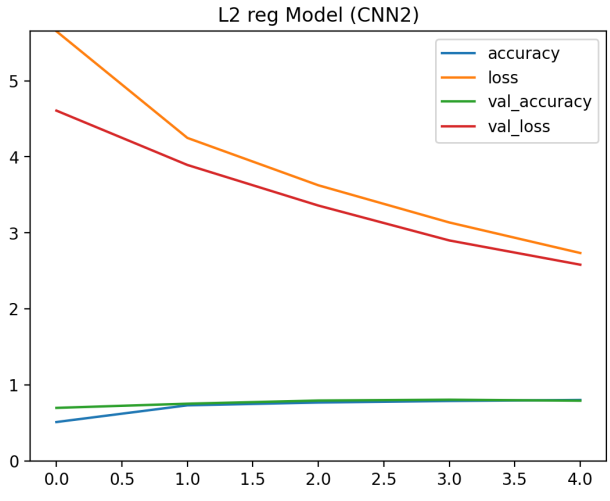


Dropout: Achieving 0.6033 loss 76.41% accuracy on the test set (26 minute compute time on google colab single gpu). (1592s)

Regularization

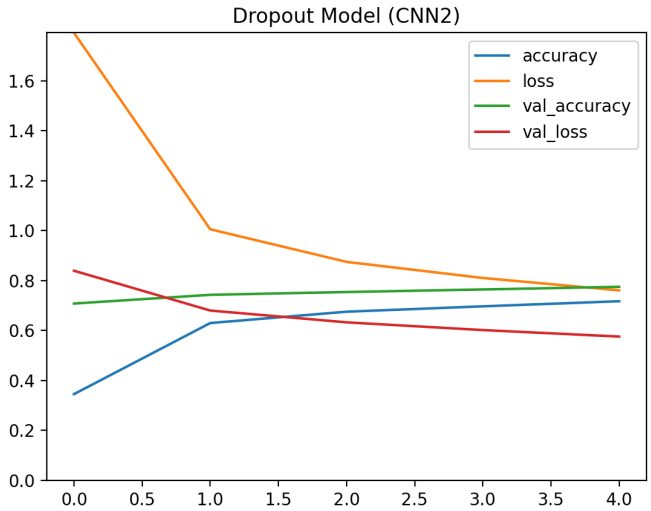
We take the same approach for Conv2 with L2 regularization.

Layer (type)	Output Shape	Param #
reshape_9 (Reshape)	(None, 28, 28, 1)	0
conv2d_24 (Conv2D)	(None, 26, 26, 32)	320
conv2d_25 (Conv2D)	(None, 24, 24, 64)	18,496
conv2d_26 (Conv2D)	(None, 11, 11, 128)	73,856
conv2d_27 (Conv2D)	(None, 5, 5, 256)	295,168
flatten_8 (Flatten)	(None, 6400)	0
dense_24 (Dense)	(None, 64)	409,664
dense_25 (Dense)	(None, 10)	650



Regularization: Achieving 2.61 loss on the test set 77.99 % accuracy on the test set aross 23.86m (1432s)

Our conv2d with 2 layers model looked like; with loss: 0.6032 and accuracy (on the same sets as above of 0.764%)



Comparative results:

Model Changes	Test Loss	Test Accuracy (%)	Compute Time (s)
Leaky ReLU	2.54	81.89	1507
Dropout	0.6033	76.41	1592
Regularization	2.61	77.99	1432
Conv2D with 2 layers (base)	0.6032	76.40	1350

Conclusion / summary of results

Designing, implementing and optimizing a neural network is a broad task; the pool of plausible architectural decisions; hyperparameters and also loss functions is vast and hardware constrained Above, we focus on

1. Implementing a CNN layer with strides and convolutions

2. Improving that model by tuning hyper parameters (to success!)
3, improving the model by adding dropout and regularization layers. (to no great success)

We note that our approach is exploratory and not indicative of a practical real world approach to improving the performance (i.e. accuracy / loss) of a NN model.

We touched on some unimplemented refinements, other refinements that we could have included include early stopping, augmenting the input set with transformations to inflate the input size and of course, an algorithmic search of the hyperparameter space.

In terms of our changes Leaky ReLU significantly improved accuracy, but didn't reduce the loss as effectively as Dropout. Regularization provided balanced performance but didn't outperform Dropout in terms of loss. Each model alteration increased compute time, with Dropout requiring the most. These are, naturally a small subset of plausibilities; but highlight the positive effect that a good activation function has on this particular domain (with a deep – *er*) neural network in mitigating vanishing gradient

The Google Colab Notebook used for this is available [here]

(running out of space at end)

//