# COMP7703 – Demo 2
# Joel Thomas 44793203

**1.**

**a)**

The method used here for selecting (optimising) hyper-parameters is known formally as Random Search Cross-Validation which is essentially a combination of Random Search and $k$-Fold Cross-Validation (Bergstra & Bengio, 2012).

Initially, for each hyper-parameter within a given optimiser, 5 values are generated random uniformly over a closed interval which represents the desired range we wish to test each hyper-parameter over. Then, Random Search chooses random combinations (subsets) for each optimiser. For simplicity, the first value for one hyper-parameter was tested with the first value for the other hyper-parameters, second, and so on (this won't make a difference since the values are generated random uniformly anyway). In total, 5 potential models for each optimiser are tested.

It isn't ideal to train and validate these models over the complete training and testing datasets because we may be overfitting the models which raises uncertainty as to how these models will perform on unseen data. Thus, k-Fold Cross Validation is used with $k = 5$ since there are 5 models for each optimiser. With $k = 5$, the entire training set is split into $\frac{1}{5} \times 100 = 20\%$ validation and $\frac{4}{5} \times 100 = 80\%$ training sets for each split (see diagram below).

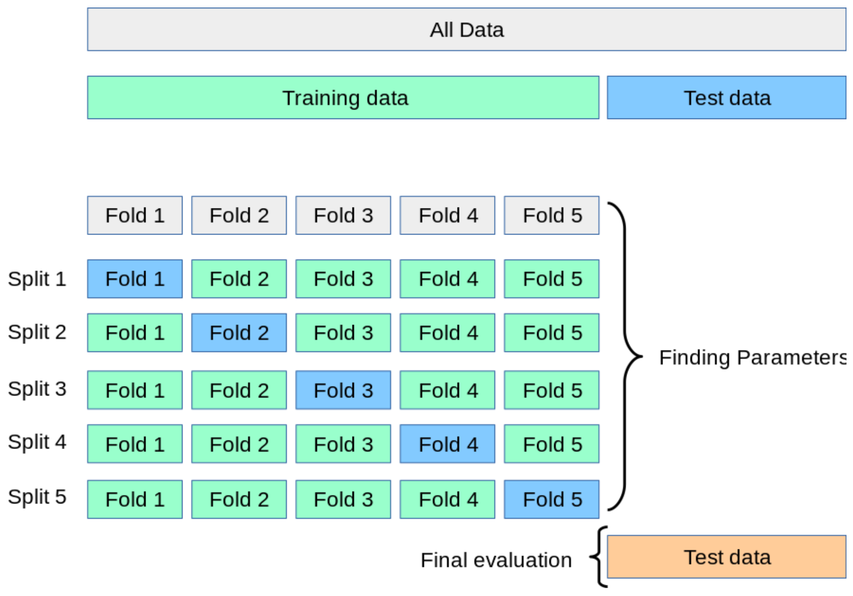

**Figure 1: How $k$-Fold Cross Validation works.**

After evaluating each of the 5 models for each optimiser, the model with the best hyper-parameter settings for a specific optimiser is labelled as the one that achieves the greatest accuracy on its validation set. Finally, these hyper-parameter settings are used when retraining the entire model on the complete training and testing datasets to identify how well the chosen model performs on unseen data (test accuracy).

On a final note, we could have used Grid Search as an alternative to Random Search but it is far too computationally expensive and time inefficient to train every possible combination of hyper-parameters (for RMSProp and Adam, $5^3$ models). Random Search is also known to perform equally well (Bergstra & Bengio, 2012). The results for **Q1.a)** are displayed in tables 1-6 below:

| **Gradient Descent Optimiser** | | | | | |
|---|---|---|---|---|---|
| **Learning rate [0.0001, 0.001)** | 0.0003339 | 0.00018095 | 0.00028503 | 0.00092229 | 0.00010963 |
| **Validation set accuracy** | 84.05% | 82.88% | 84.25% | 80.36% | 79.53% |

**Table 1: Random Search 5-fold Cross-Validation on Gradient Descent optimiser.**

| **Momentum Optimiser** | | | | | |
|---|---|---|---|---|---|
| **Learning rate [0.0001, 0.001)** | 0.0003339 | 0.00018095 | 0.00028503 | 0.00092229 | 0.00010963 |
| **Momentum [0.001, 0.1)** | 0.09998442 | 0.08896199 | 0.02783913 | 0.07007324 | 0.0393303 |
| **Validation set accuracy** | 84.4% | 77.73% | 82.43% | 84.62% | 80.34% |

**Table 2: Random Search 5-fold Cross-Validation on Momentum optimiser.**

| **Adagrad Optimiser** | | | | | |
|---|---|---|---|---|---|
| **Learning rate [0.0001, 0.001)** | 0.0003339 | 0.00018095 | 0.00028503 | 0.00092229 | 0.00010963 |
| **Initial accumulator value [0.01, 0.2)** | 0.02323261 | 0.07627776 | 0.11548787 | 0.12798474 | 0.16298017 |
| **Validation set accuracy** | 62.37% | 49.31% | 59.98% | 79.48% | 30.66% |

**Table 3: Random Search 5-fold Cross-Validation on Adagrad optimiser.**

| **RMSProp Optimiser** | | | | | |
|---|---|---|---|---|---|
| **Learning rate [0.0001, 0.001)** | 0.0003339 | 0.00018095 | 0.00028503 | 0.00092229 | 0.00010963 |
| **Momentum [0.001, 0.1)** | 0.09998442 | 0.08896199 | 0.02783913 | 0.07007324 | 0.0393303 |
| **Decay [0, 0.01)** | 0.00071767 | 0.00412366 | 0.00500406 | 0.00143713 | 0.00887334 |
| **Validation set accuracy** | 82.88% | 79.19% | 83.66% | 86.98% | 75.5% |

**Table 4: Random Search 5-fold Cross-Validation on RMSProp optimiser.**

| **Adam Optimiser** | | | | | |
|---|---|---|---|---|---|
| **Learning rate [0.0001, 0.001)** | 0.0003339 | 0.00018095 | 0.00028503 | 0.00092229 | 0.00010963 |
| $\boldsymbol{\beta_1}$ **[0.85, 0.95)** | 0.9387497 | 0.88373554 | 0.91936885 | 0.89783665 | 0.92459742 |
| $\boldsymbol{\beta_2}$ **[0.9, 1)** | 0.91632579 | 0.98701609 | 0.93724213 | 0.90311939 | 0.99082757 |
| **Validation set accuracy** | 87.88% | 82.01% | 85.67% | 89.1% | 78.32% |

**Table 5: Random Search 5-fold Cross-Validation on Adam optimiser.**

| Comparing test set accuracies for each Optimiser using best hyper-parameter settings | | | | | |
|---|---|---|---|---|---|
| **Optimiser** | **Gradient Descent** | **Momentum** | **Adagrad** | **RMSProp** | **Adam** |
| **Test set accuracy** | 84.5% | 84.27% | 79.84% | 87.74% | 89.78% |

**Table 6: Evaluation of best hyper-parameter settings for each optimiser on unseen data (testing set).**

For the Gradient Descent optimiser, a learning rate too large could lead to the model being unable to converge even to a local optimum because with every step, it could keep bouncing back and forth on the gradient slope. Conversely, a learning rate too small is inefficient because several more steps would need to be performed to reach a local/global minimum – more steps is equivalent to more calculations so computationally and time inefficient.

The Momentum optimiser can be thought of as a ball rolling down a hill – momentum is gained the further the ball is down the hill. In regions with gentle slopes, GD may take a long time whereas the Momentum optimiser is expected to converge faster. The problem here is because a ball crosses the "bottom" of the hill several times before settling, the same could also occur with the Momentum optimiser's convergence.

The Adagrad optimiser is traditionally known to work very well with sparse data where there are a lot of 0s embedded within the data (perfect for MNIST!). It does this by decreasing the initial set learning rate (lower) for parameters (weights) with frequently occurring features and vice-versa (Ruder, 2016). The main weakness is that eventually, the learning rate may become so small (dividing $\eta$ by accumulated squared gradients, see formula) that the model does not learn anymore.

The RMSProp optimiser is built on top of the Adagrad with its main purpose being to counter the vanishing gradients issue above (Ruder, 2016). Its main weakness is that it only keeps track of past squared gradients and not past gradients (notion of momentum).

The Adam optimiser is basically the Momentum and RMSProp optimisers combined together, so in this way, it has a notion of momentum as well as an adaptive learning rate similar to RMSProp (intuition – heavy ball with friction) (Ruder, 2016). Since every optimiser has built on standard Gradient Descent thus far, Adam optimiser can be thought of as the "gold standard" of optimisers since it attempts to cover the weaknesses of each optimiser before it – hence why it is so popular.

**b)**

| **Comparing test set accuracies for each Activation Function** | | | |
|---|---|---|---|
| **Activation Function** | **ReLU** | **Tanh** | **Sigmoid** |
| **Test set accuracy** | 86.11% | 88.29% | 89.21% |

**Table 7: Evaluation of each activation function on complete training and testing sets.**

The sigmoid activation function is given by $f(x) = \frac{1}{1+e^{-x}}$. It is bounded from 0 to 1, non-linear and differentiable everywhere. An advantage is that for values lying in a small interval around 0, it has an aggressive activation (Kızrak, 2019). Another advantage is that because it is bounded so it can take in arbitrarily large values. A disadvantage is the issue of vanishing gradients – if the input → $\pm\infty$ or the input $= 0$, the derivative at these parts of the function is 0 so there is minimal (if any) learning (Kızrak, 2019).

The hyperbolic tangent activation function is given by $f(x) = \tanh(x) = \frac{1-e^{-2x}}{1+e^{2x}}$. It is bounded from -1 to 1, also non-linear and differentiable everywhere. It is very similar to the sigmoid function, the only difference being a steeper gradient at the inflection point (Kızrak, 2019). Hence, it shares the same advantages and disadvantages as the sigmoid function.

The rectified linear unit activation function is given by $f(x) = \begin{cases} 0, x \leq 0 \\ x, x > 0 \end{cases} = \max(0, x)$. It has a lower bound at 0 but does not have an upper bound, is non-linear and differentiable almost everywhere except at $x = 0$ (left-derivative of 0, right-derivative of 1 → derivative undefined). Because the sigmoid and hyperbolic tangent functions have almost every input yielding an output $> 0$, for a large neural network, almost all neurons will be activated in some way, so the activation is intense (Kızrak, 2019). Using the reLU function is desirable because it enables the entire network to run much faster during both training (several derivatives during backprop will simply be 0) as well as during prediction (faster matrix multiplications), so the reLU is computationally efficient. The downside here is that whenever the reLU outputs 0 (i.e. $x \leq 0$), no learning is done on that neuron (Kızrak, 2019).

**References:**

- Bergstra, J. and Bengio, Y., 2012. *Random Search for Hyper-Parameter Optimisation*. Journal of Machine Learning Research, 13.
- Ruder, S., 2016. *An overview of gradient descent optimization algorithms*. [online] Available at: <https://ruder.io/optimizing-gradient-descent/> [Accessed 23 April 2021].
- Kızrak, A., 2019. *Comparison of Activation Functions for Deep Neural Networks*. [online] Medium. Available at: <https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a> [Accessed 23 April 2021].