ANNs For Image Classification

Name: Sphesihle Madonsela
Student Number: MDNSPH007

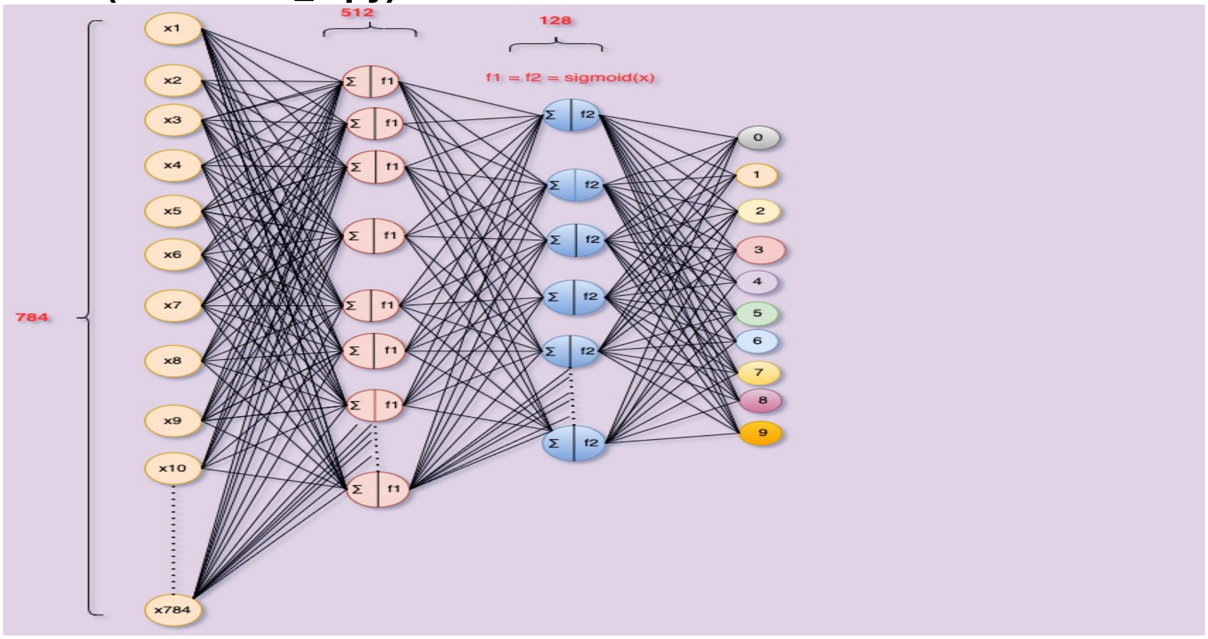
Introduction:

The task of this assignment is to design Artificial Neural Networks That can classify written digits from the MNIST10 dataset. Given a training dataset of 60000 small square 28x28 pixels grayscale images of handwritten digits between 0 and 9 and 10000 test datasets for evaluating how well the neural network has recognise digits. The goal is to design ANNs that can successfully classify a given image of handwritten digits into one of 10 classes, representing integer values from 0 to 9 inclusively, to some degree of accuracy.

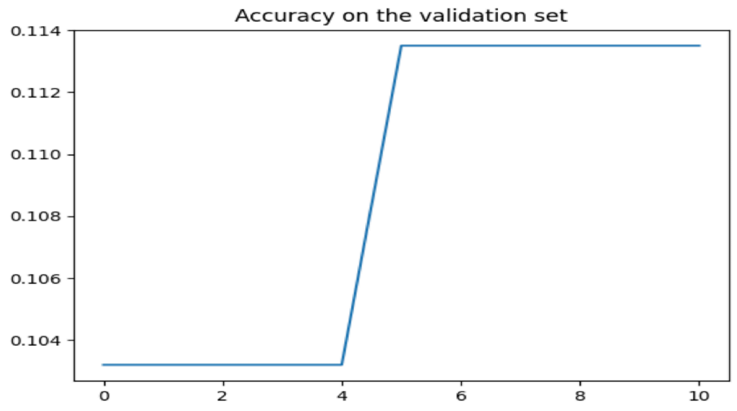
Body:

- The **input-layer** of the network contains neurons/nodes encoding the pixels input pixels. A picture of hand-written digit consists of 28 by 28 pixels.
- **Training Process and Pre-processing:**
    - The images are converted into a 1D array with 784 i.e. 28x28 items. Thus, there are 784 neurons on the input layer
    - When training our model, we want to pass the data in mini-batches and reshuffle the data every run (epoch) the reduce the model overfitting. And using python's multiprocessing to speed up the data retrieval.
    - A batch size of 512 was used to achieve this
    - An optimization loop was used for training and optimizing the model. An iteration of this loop is known as an epoch.
    - One epoch consists of the training loop – iterating over the training dataset, trying to converge to optimal parameters. And a Validation loop, iterating over the test dataset to check if the performance/accuracy of the model is improving.
    - If the performance is not improving for 5 or more epochs, the optimization loop breaks, and stop the training.

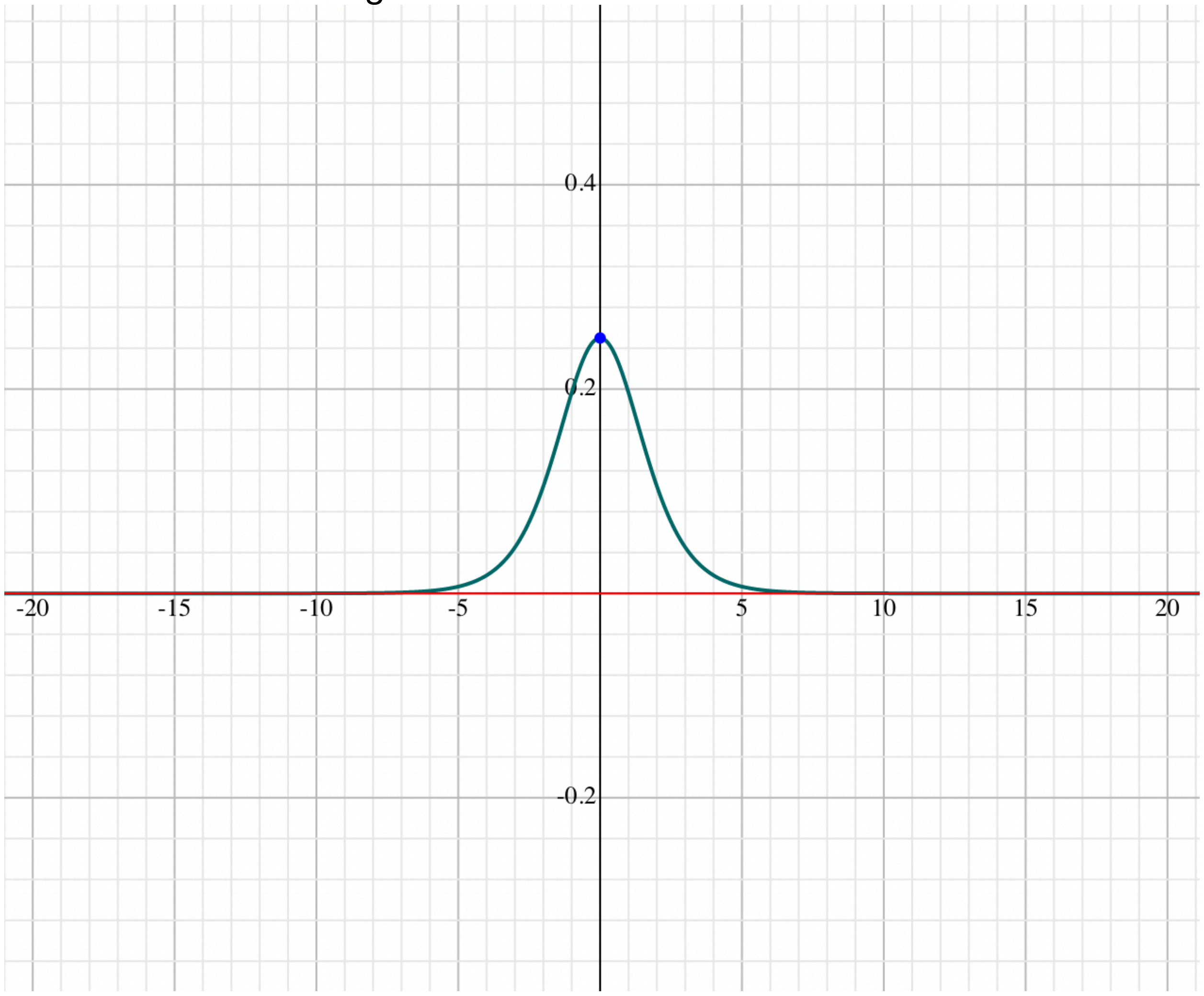
ANN1(Classiffier_1.py)



- **Hidden layer-1** consists of 512 nodes
- **Hidden layer-2** consists of 128 nodes
- The hidden layers transform the decision boundary to a non-linear space when a non-linear activation like sigmoid or tanh is chosen.
- **Activation function:** This network uses sigmoid in both hidden layers.
- The diagram of the sigmoid activation function, applied on the ∑(w1x1 +…x784w787+ b) where b is the bias term. The sigmoid squashes this sum to a value between 0 and 1.]
- There are down-sides in using sigmoid comes in the backpropagation, the derivative of the sigmoid function is a continuous function that lies between 0 and 0.25. When x-> ∞ or x-> -∞ the derivative becomes approximately equal to zero.
- When doing the backpropagation, it multiplies the partial derivatives of the weights. To get ∂E/∂wij
- Wij(new) = Wij(old) - n * ∂E/∂wij, when ∂E/∂wij ≈ 0 then Wij(new) ≈ Wij(old)
- Then the network converges much quicker, as the accuracy graph shows below.



The derivative of the sigmoid function



- The **Stochastic Gradient Optimiser (SGD)** was used here as an optimiser. SGD updates the weights incrementally. This ensures that we don't get stuck in the local minima. It is faster than the Traditional Gradient Decent.

- **Output-layer:** Evaluating the labels against neurons this is an array of 10 elements where one of the elements is set to a value ≈ 1 the rest to 0 e.g., for value 9 then the array will be [0,0,0,0,0,0,0,0,0,1].
- The network is fully connected, i.e., all the neurons in the previous layer are connected to all the neurons in the following layer.
- **Cross-Entropy cost function was used:**

$$Loss = E = -\frac{1}{n}\sum\left[t \cdot ln(o) + (1-t) \cdot ln(1-o)\right]$$

- Where t is the desired output and o is the actual output from the from a neuron and n is the size of the training dataset.
- This error function solves the problem of learning slowing down. When computing the partial derivative of the loss function E, i.e ∂E/∂wj

$$\frac{\partial E}{\partial wj} = -\frac{1}{n}\sum\frac{f'(ui) \cdot xj}{f(ui)(1-f(ui))} \cdot \left(f(ui) - t\right)$$

- f(ui) is the activation function, for the sigmoid f'(ui) = f(ui)(1-f(ui))
- Substituting the derivative and simplifying:

$$\frac{\partial E}{\partial wj} = -\frac{1}{n}\sum xj \cdot \left(f(ui) - t\right)$$

ANN2(Classiffier_2.py)

- This ANN uses the same topology as the one in ANN1, but now using a different optimizer.
- It uses Adaptative Moment Estimation (Adam), computes adaptative learning rates for each parameter. On the equation for updating the weights
- $$wj(new) = wj(old) + \Delta wj$$
- Where Δwj = -n∂E/∂wj
- Now the Adam optimizer computes an Exponential Weighted averages of for past gradients.

-

$$V_{dw_t} = \beta V_{dw_{t-1}} + (1-\beta)\frac{\partial L}{\partial w_{t-1}}$$

- Then computes the Exponential Weighted averages for the past squared gradients.

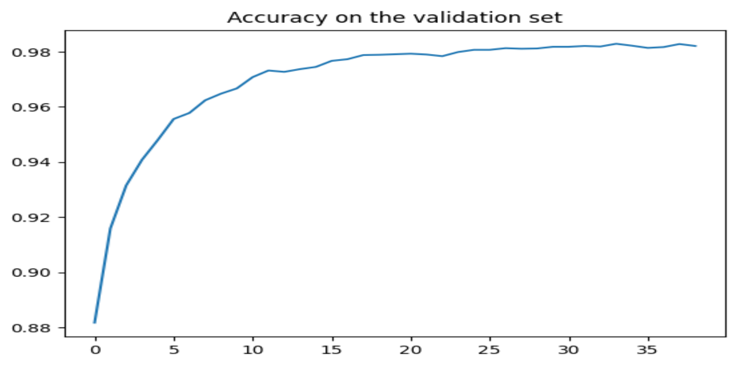
- $$S_{dw_t} = \gamma S_{dw_{t-1}} + (1-\gamma)\left(\frac{\partial L}{\partial w_{t-1}}\right)^2$$

- L is here is the same as the E for Loss/Cost function

- Now determining Δwj

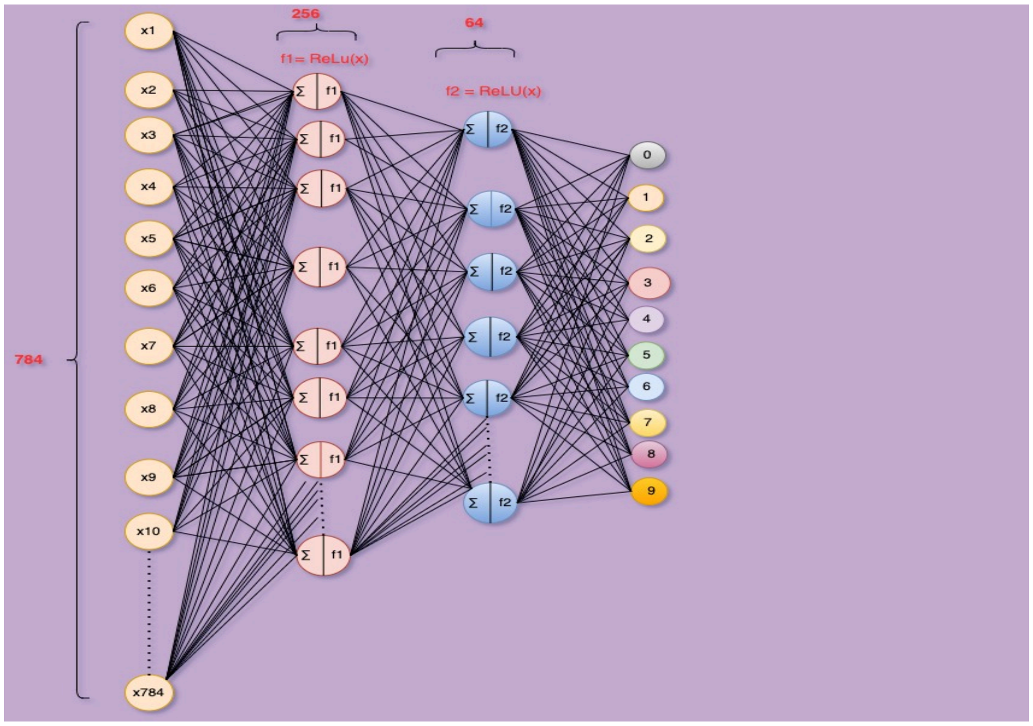
- $$\Delta wj = -\frac{n}{\sqrt{Sd(w(old))} - \xi} \cdot Vd(w(old))$$

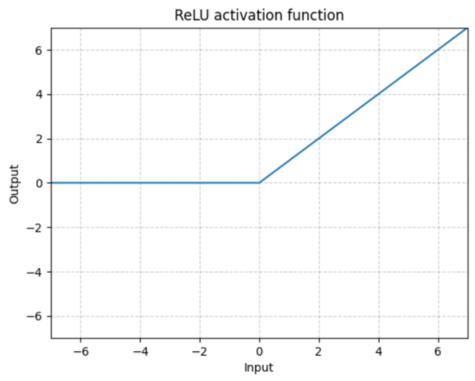
- Which overall changes how the waits are updated
- Also, Adam is suitable for problems with very noisy gradients.
- The graph showing the results after changing the optimizer from SGD.
- Which gives us a big jump from 11.4% to about 98% accuracy
-



-

ANN1(Classiffier_3.py)

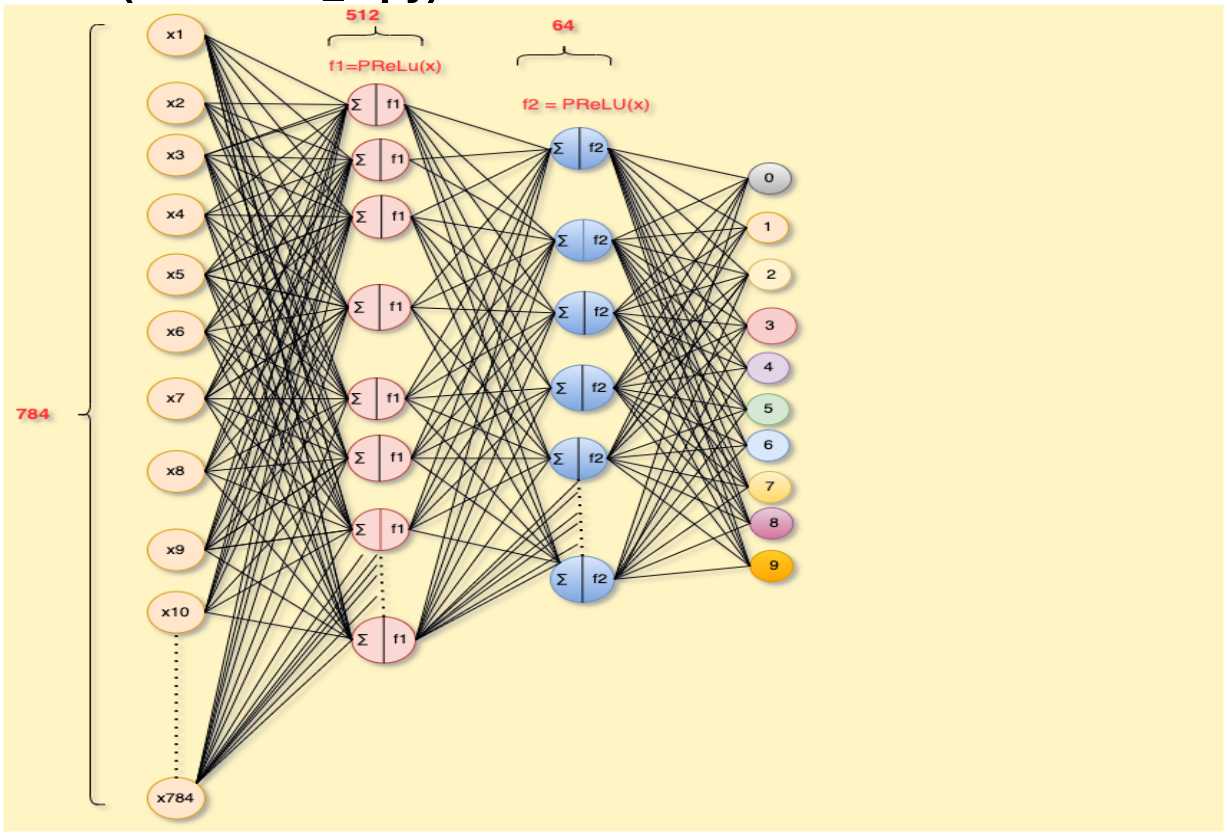


- Hidden layer 1 has 256 nodes and Hidden layer 2 has 64 nodes.
- Hidden layer 1 uses a ReLU(x) activation function.
- $\mathrm{ReLU}(x) = \max(0,x)$, this just says that if the weighted sum is less than or equal to 0 then this function is going to output else keeps the weighted as is.
- 

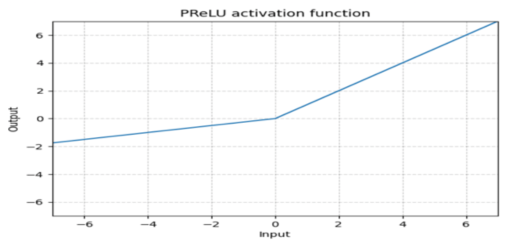
- When the input of the is positive there’s no gradient satuaration
- The calculating speed is much faster, the ReLu function has only a linear relationship, whether it’s much faster than sigmoid and tanh.
- The second hidden layer also has ReLU

- The derivative of ReLu is 0 for negative values and 1 for positive input. The forward pass is not an issue, but the back propagation process ReLu encounters negative weights. The gradient will be zero.
- Uses Cross Entropy loss function, explained in ANN1

ANN4(Classifier_4.py)



- Hidden layer 1 has 512 nodes and hidden layer 2 has 64 nodes.
- Uses Cross-Entropy loss function
- Adam optimizer
- In both hidden layers the PReLU was used.
- $\text{PReLU}(x)=\max(0,x)+a*\min(0,x)$
-



-

- PReLu(x) = x if x≥0 else ax
- Where 0<a<1， a is a learnable parameter.
- PReLU is an improved version of the ReLu in the negative region, PReLU has a small gradient which can avoid the problem of ReLu death basically the derivative of ReLu being equal to 0.
- PReLU is a linear operation in the negative region, although the slope is small it does not tend to zero which is an advantage when computing the derivative for back-propagation and updating the weights.
- This ANN obtains the highest accuracy 98+% in fewer epochs because it uses an activation function that ensures that the ∂E/∂wj is not zero.

Overfitting or over train:

- Using early stopping to determine the number of training epochs, at the end of each epoch the classification accuracy of the test data was computed. When the accuracy stops improving for 5 epochs stop.
- Early stopping automatically solves the overfitting issue.

Conclusion:

- There’s a challenge in choosing hyperparameters, special for the hidden layers. It’s a ‘black-box’ we can’t really explain whether increasing the number of nodes increases the performance of our network, and if does we can’t explain it.
- To achieve an even higher accuracy than 98% we can use CNNs

References

Neural Network From Scratch with Numpy and MNIST
https://mlfromscratch.com/neural-network-tutorial/#/

Neural Networks and Deep Learning, by Michael Nielsen
https://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf

Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747

Preparing your data for training with DataLoaders:

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

Optimization Loop:

https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

Adam Optimizer:

https://towardsdatascience.com/deep-learning-optimizers-436171c9e23f

https://ruder.io/optimizing-gradient-descent/index.html#adam

ReLU Diagram:
https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

Activation Functions, by Krish Naik
https://youtu.be/qVLQ9Cqm-ec

PReLU Diagram:
https://pytorch.org/docs/stable/generated/torch.nn.PReLU.html#torch.nn.PReLU