Assignment3

May 20, 2025

1 Computer Vision 2025 Assignment 3: Deep Learning for Perception Tasks

This assignment contains 2 questions. The first question probes understanding of deep learning for classification. The second question requires you to write a short description of a Computer Vision method. You wil need to submit two separate PDF files, one for each question.

All results presented in this report represent the average of three independent trials conducted using the same data and parameters.

1.1 Question 1: A Simple Classifier *(20 marks, 60%)*

For this exercise, we provide demo code showing how to train a network on a small dataset called Fashion-MNIST. Please run through the code *"tutorial-style"* to get a sense of what it is doing. Then use the code alongside lecture notes and other resources to understand how to use pytorch libraries to implement, train and use a neural network. For the Fashion-MNIST dataset the labels from 0-9 correspond to various clothing classes so you might find it convenient to **create a python list as follows:**

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

You will need to answer various questions about the system, how it operates, the results of experiments with it and make modifications to it yourself. You can change the training scheme and the network structure. Organise your own text and code cell to show the answer of each question below. **Detailed requirements:**

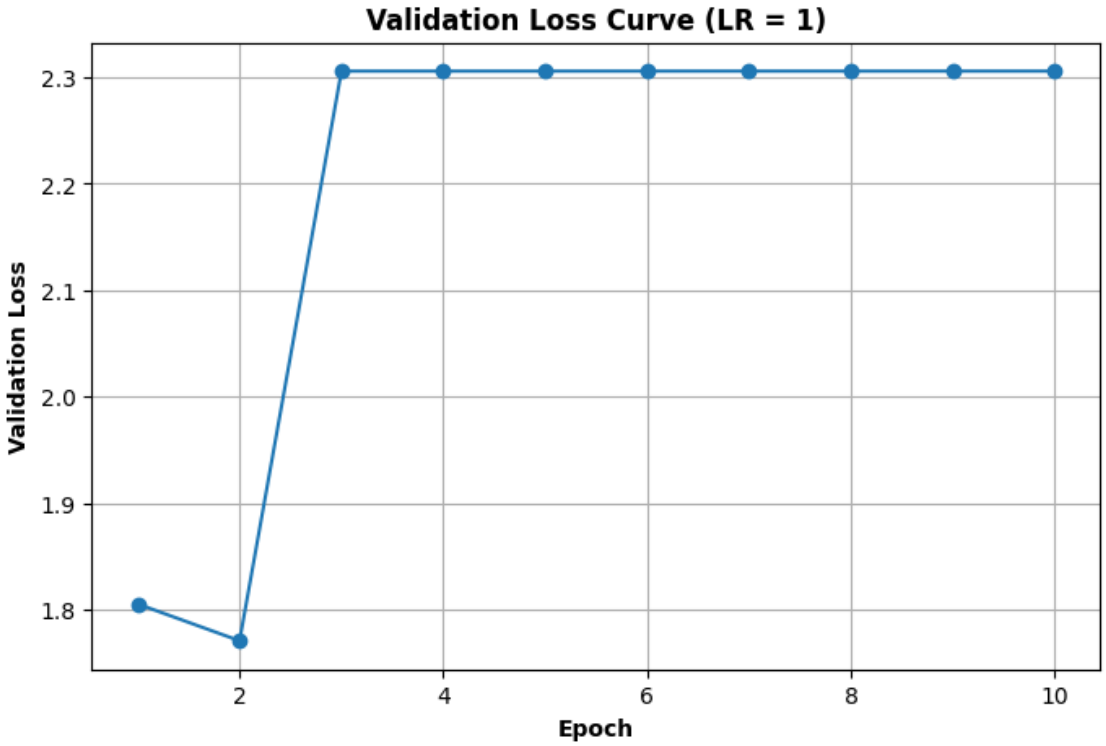
1.1.1 Q1.1 *(1 Point)*

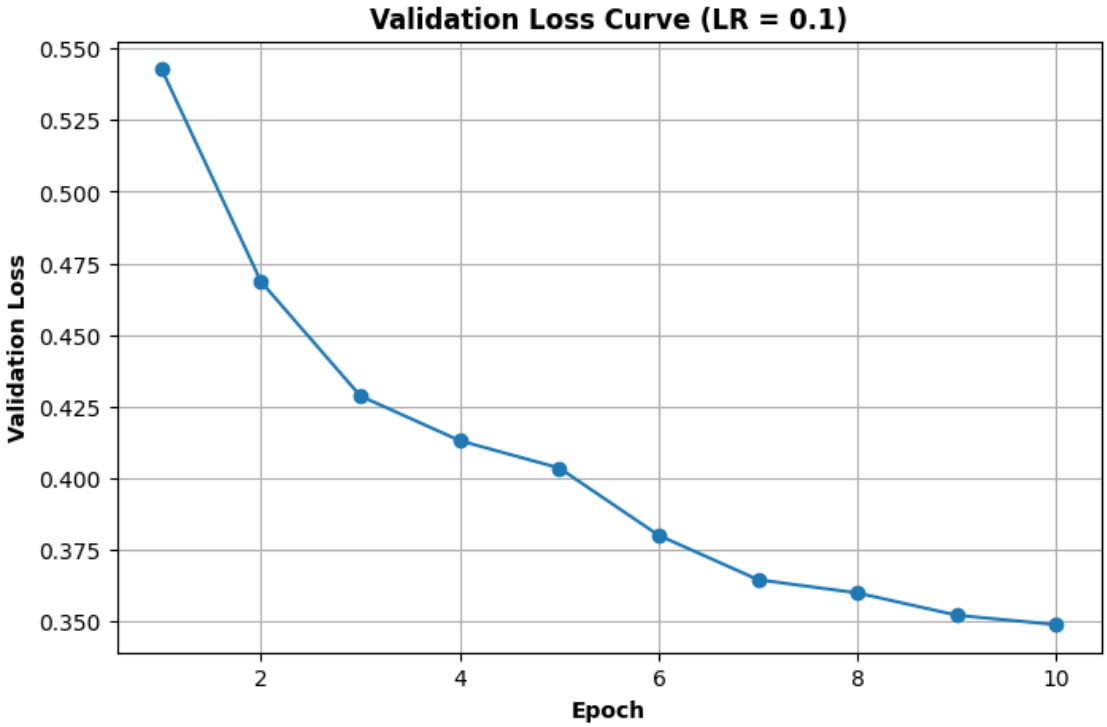
Extract 3 images of different types of clothing from the training dataset, print out the size/shape of the training images, and display the three with their corresponding labels.

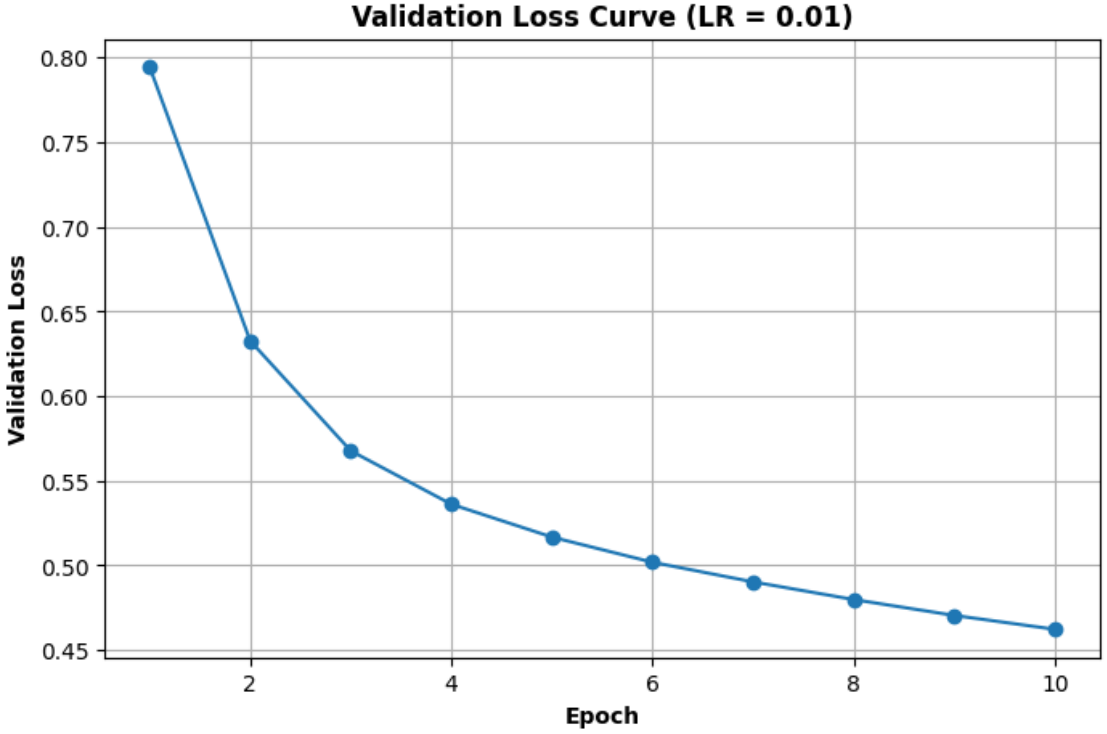
1.1.2 Q1.2 *(2 Points)*

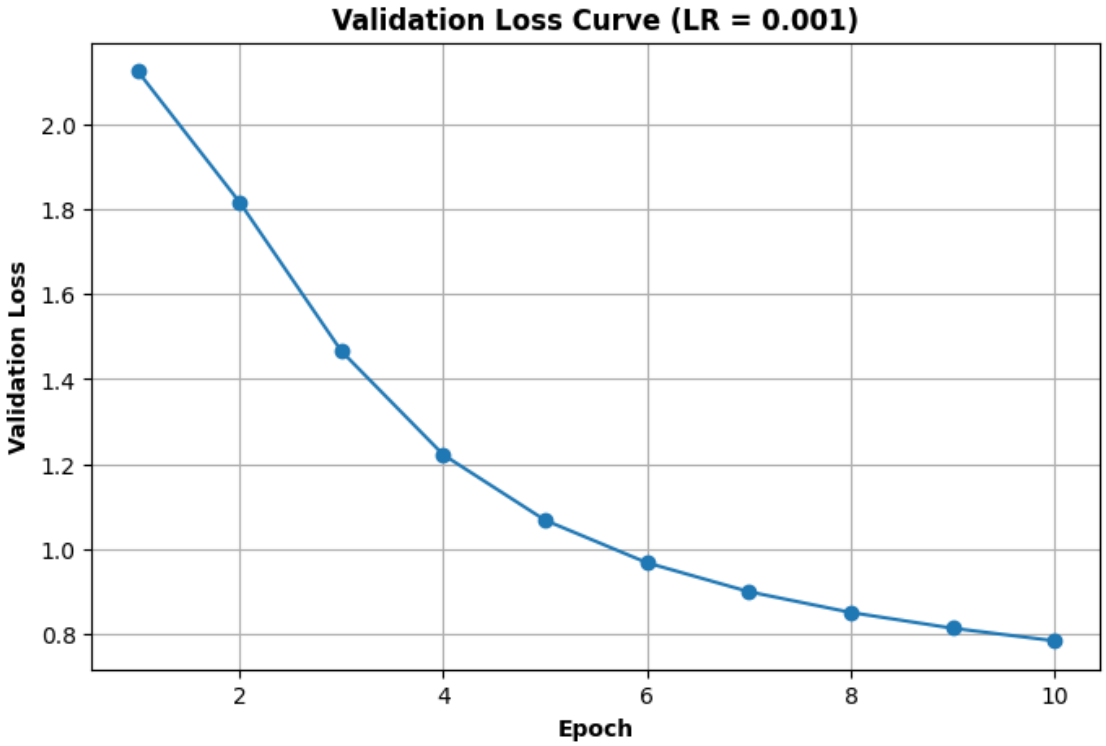
Run the training code for 10 epochs, for different values of the learning rate. **Fill in the table below and plot the loss curves for each experiment:**

LR	Accuracy
1	10.00%
0.1	87.40%
0.01	83.40%
0.001	70.80%




Validation Loss Curve (LR = 0.1)
Validation Loss
0.550
0.525
0.500
0.475
0.450
0.425
0.400
0.375
0.350
2
4
6
8
10
Epoch

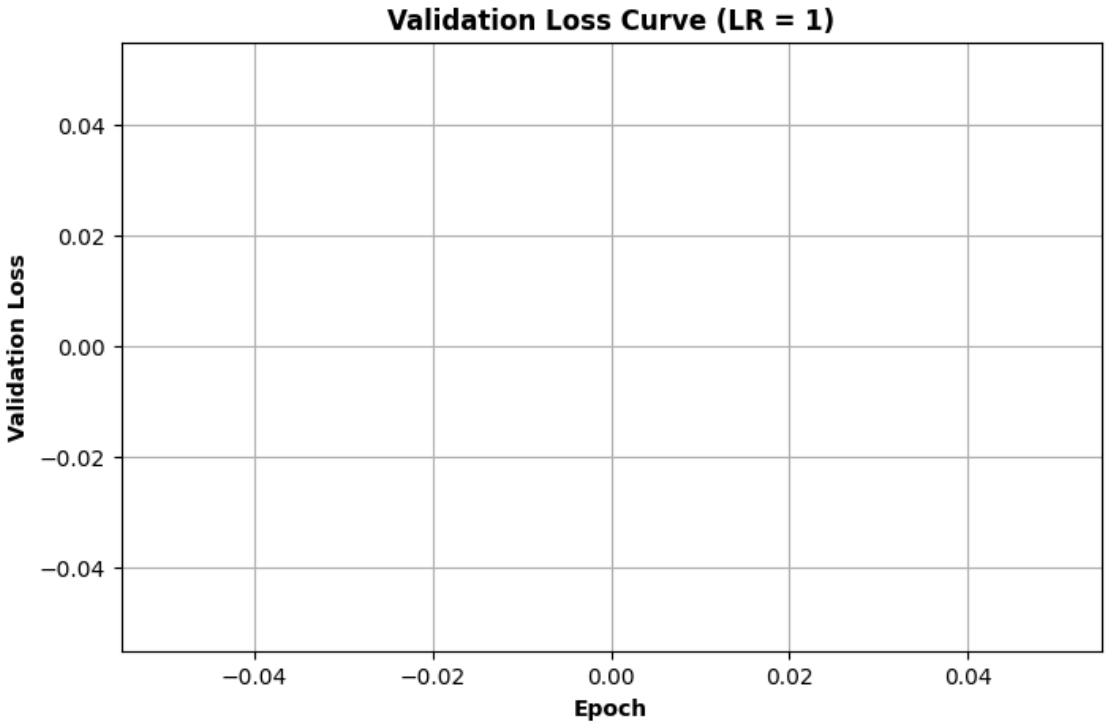

Validation Loss Curve (LR = 0.01)
Validation Loss
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
2
4
6
8
10
Epoch

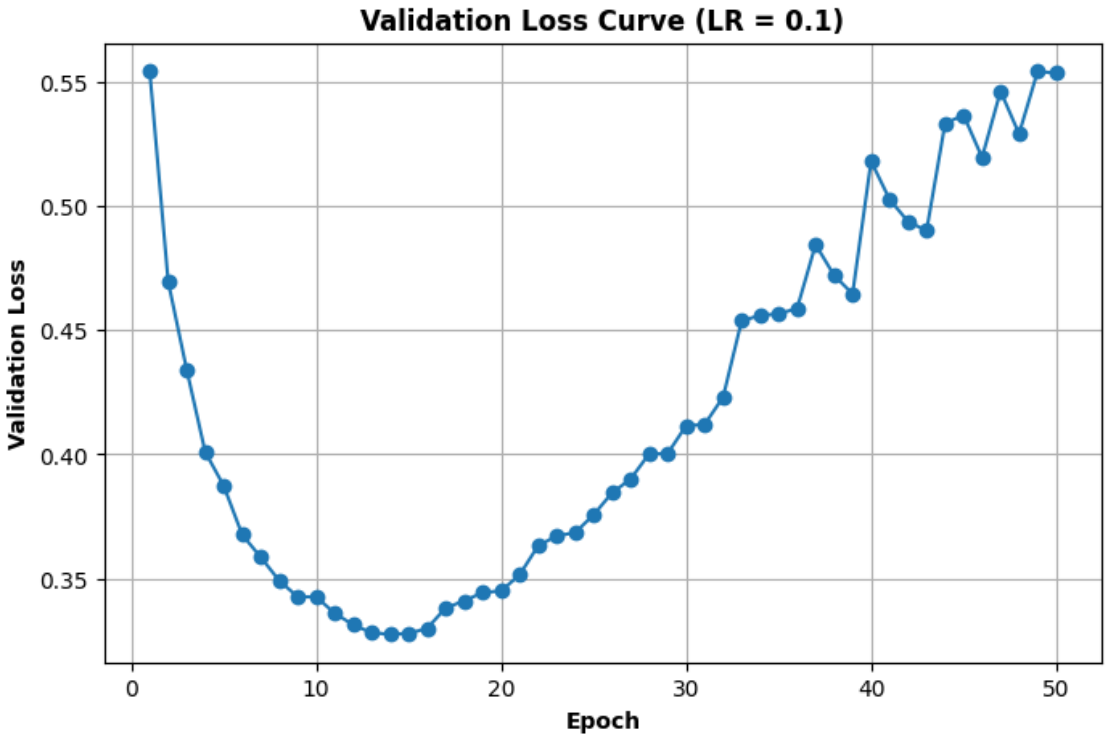


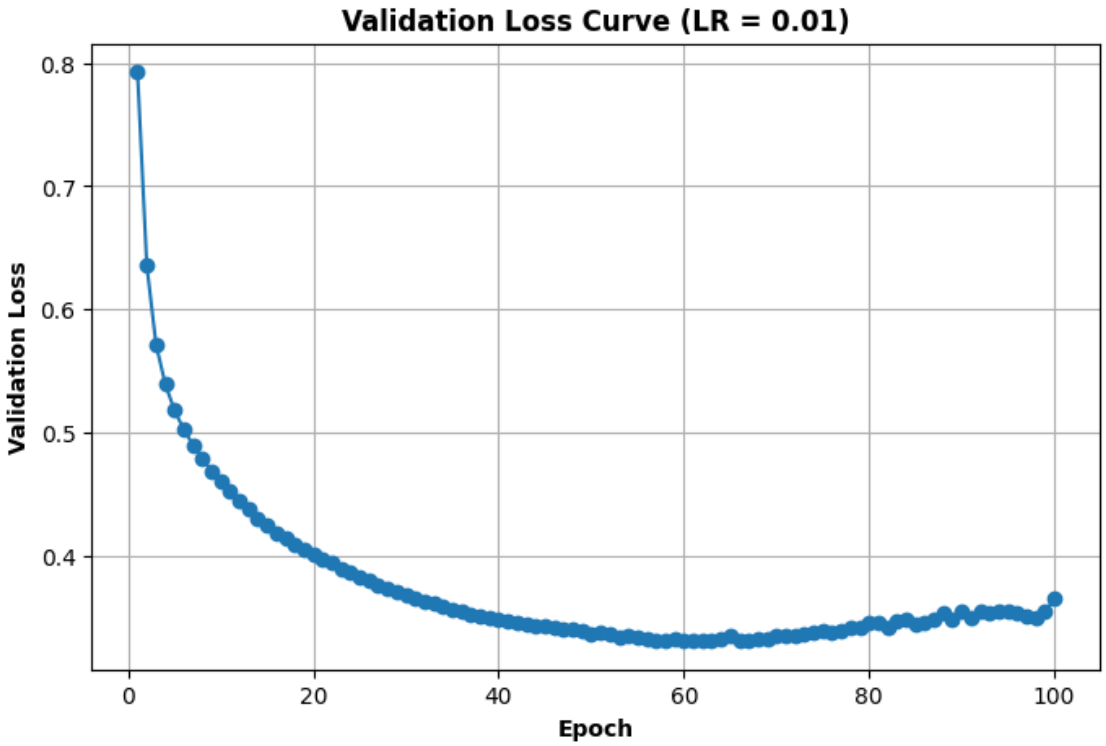
1.1.3 Q1.3 *(3 Points)*

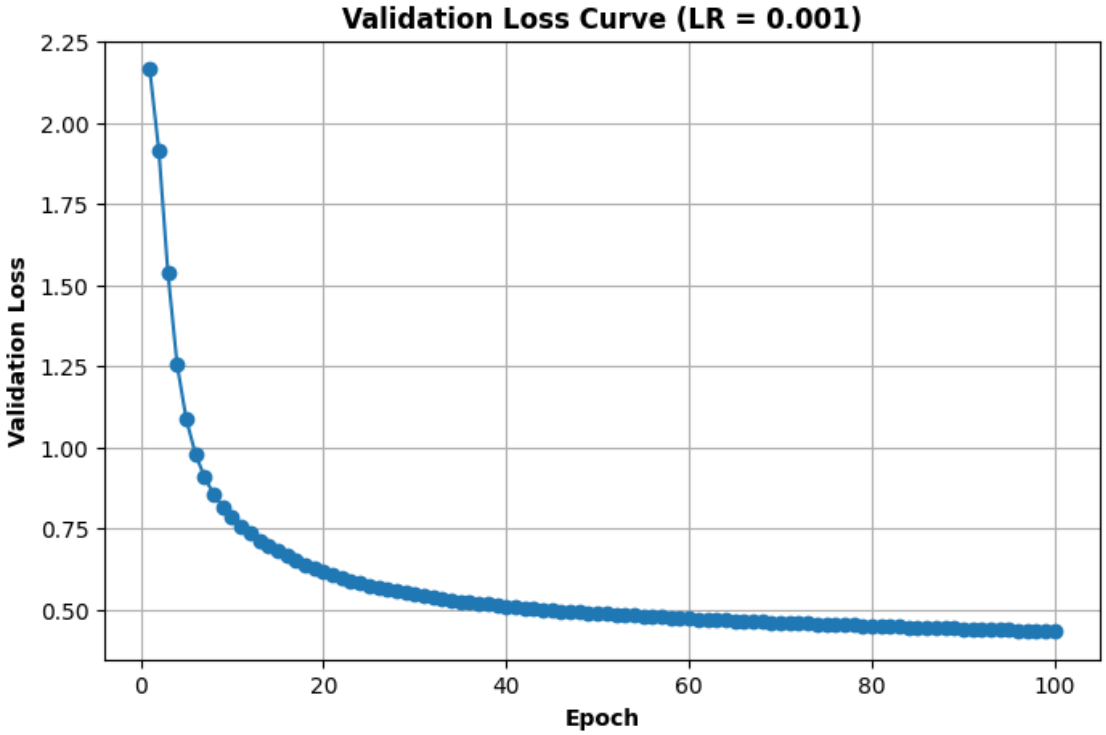
Report the number of epochs when the network converges *(or number of epochs for the best accuracy, if it fails to converge)*. Fill in the table below and plot the loss curve for each experiment. **Please run the code for more than 10 epochs *(e.g. 50 or 100)* and report when you observe convergence:**

LR	Accuracy	Epoch
1	NaN	NaN
0.1	88.20%	14
0.01	87.80%	60
0.001	84.60%	>100


Validation Loss Curve (LR = 1)
Validation Loss
0.04
0.02
0.00
−0.02
−0.04
−0.04
−0.02
0.00
0.02
0.04
Epoch


Validation Loss Curve (LR = 0.1)
Validation Loss
0.55
0.50
0.45
0.40
0.35
0
10
20
30
40
50
Epoch


Validation Loss Curve (LR = 0.01)
Validation Loss
0.8
0.7
0.6
0.5
0.4
0
20
40
60
80
100
Epoch


Validation Loss Curve (LR = 0.001)
Validation Loss
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0
20
40
60
80
100
Epoch

1.1.4 Q1.4 *(2 Points)*

Compare the results in table 1 and table 2, what is your observation and your understanding of learning rate?

Upon examining the results from Tables 1 and 2, a few key observations can be made about the effect of the learning rate on training and convergence. A learning rate of 1 is excessively high, leading to unstable and unreliable training behavior. In the 10-epoch evaluation, the model consistently achieved 10% accuracy *(likely equivalent to random guessing)* indicating that no meaningful learning occurred. Furthermore, in the convergence test, training either diverged *(resulting in NaN values as seen above)* or quickly plateaued at 10% accuracy within approximately 12 epochs. This phenomenon occurs due to exploding gradients, where large weight updates cause the model to overshoot and fail to converge. Overall, this suggests that the learning rate was too large for the model to converge effectively, leading it to fail to maintain stability during training. Unfortunately, the resulting data is not particularly informative for assessing model performance.

Additionally, the learning rate of 0.1 produces the highest accuracy *(87.40% on average)* within the first 10 epochs, as shown in Table 1. This value strikes a balance between large enough updates to quickly reduce loss, but not so large that it overshoots optimal points. The model converges to the best accuracy *(~88.20%)* in approximately 14 epochs, as confirmed in Table 2. This learning rate is ideal for quick yet stable convergence, which is why the accuracy is higher compared to smaller learning rates.

With a learning rate of 0.01, the model requires more time to converge, as it makes smaller, more stable updates. While this rate leads to a lower accuracy *(83.40% on average)* in 10 epochs, it performs much closer to the 0.1 learning rate with extended training *(~60 epochs gives 87.8% on average)*. However, the model still takes longer to converge compared to 0.1, suggesting that small learning rates can lead to slow progress but a more refined convergence over time. This is evident in the smooth gradient of the loss curve. Finally, a learning rate of 0.001 is very small and results in slow convergence. The model's accuracy remains low in the first 10 epochs *(70.80% on average)* and takes over 100 epochs to completely converge, achieving an accuracy of around 84.60% after 100 epochs. While this rate offers stability, it also illustrates that extremely slow updates may not allow the model to reach its potential within a reasonable amount of time.

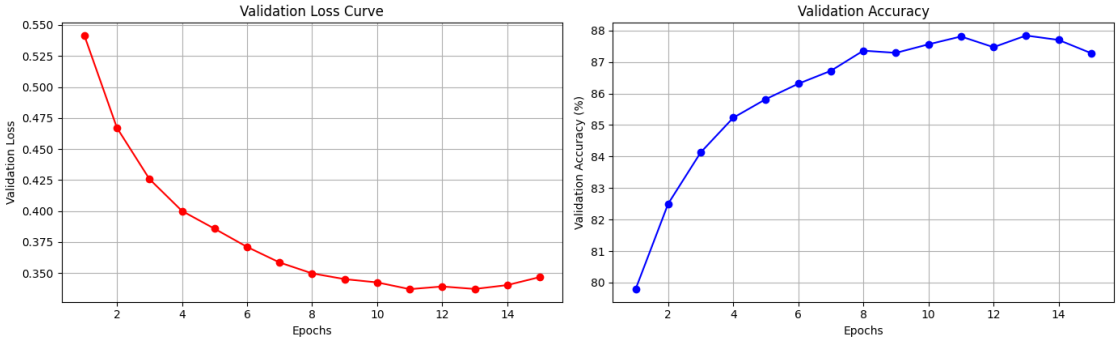
Overall, comparing the results in Table 1 and Table 2, it is clear that the learning rate has a significant impact on both training stability and convergence speed. A learning rate of 1 is too high, causing unstable updates and divergence due to exploding gradients, often resulting in NaN losses. In contrast, a very low learning rate like 0.001 ensures stability but requires far more epochs to converge, leading to lower accuracy within a limited training window. The optimal performance in this experiment was achieved with a learning rate of 0.1, which balanced convergence speed and training stability, reaching a maximum accuracy of approximately 88% in just 14 epochs. This suggests that 0.1 is well-tuned for the current model and dataset. However, the observed accuracy plateau also indicates a possible limitation of the model's capacity. Future improvements could involve modifying the network architecture, such as adjusting the number of hidden layers or neurons, to better capture complex data patterns and potentially exceed the current performance ceiling. Ultimately, this experiment highlights the importance of selecting an appropriate learning rate, as it directly influences not just how fast a model learns but whether it learns at all.

1.1.5 Q1.5 *(5 Points)*

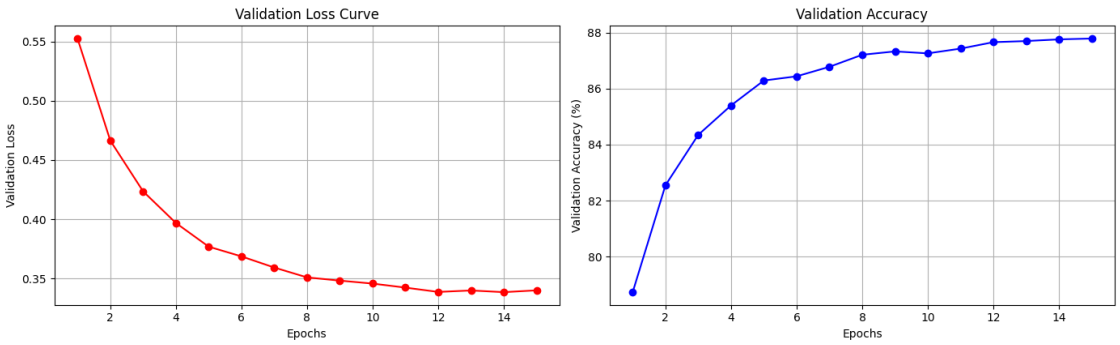
Build a wider network by modifying the code that constructs the network so that the hidden layer(s) contain more perceptrons, and record the accuracy along with the number of trainable parameters in your model. Now modify the original network to be deeper instead of wider *(**i.e.** by adding more hidden layers)*. Record your accuracy and network size findings. Plot the loss curve for each experiment. Also plot the test accuracy and loss for both the wider and deeper architectures and discuss what you observe. **Write down your conclusions about changing the network structure.**

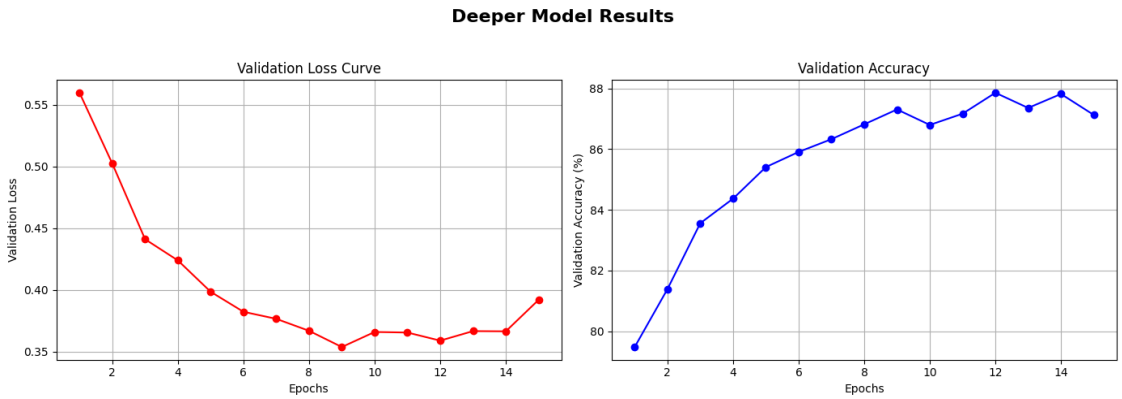
Structures	Accuracy	Parameters
Base	86.50%	669,706
Deeper	87.40%	830,090
Wider	88.20%	1,863,690

Base Model Results



Wider Model Results





The base model employed in this experiment uses a standard fully connected architecture with two hidden layers of 512 neurons each, achieving a strong test accuracy of 86% on average. The learning rate was set to 0.1, and the model was trained over 15 epochs. These hyperparameters were chosen based on their ability to provide fast and stable convergence on the relatively simple FashionMNIST dataset, as seen in the discussion above. These parameters also achieve reasonable results while keeping the model training within the hardware constraints of this project. With 669,706 trainable parameters, the base model strikes a healthy balance between model complexity and performance. The validation curves for both loss and accuracy demonstrated reasonably smooth convergence, indicating effective learning.

To explore the impact of model width, the number of neurons in each hidden layer was doubled to 1024, significantly increasing the trainable parameters to 1,863,690. The wider model achieved a higher test accuracy of 88.20%, compared to 86.50% for the base model, indicating a modest but meaningful improvement. Additionally, the wider model displayed smoother training dynamics. This is evident in the loss decreasing more steadily, and the accuracy increasing with less fluctuation compared to the base model. This suggests that the wider network facilitated better gradient flow and optimisation stability. While the performance gain was not dramatic, it shows that the wider network was able to extract more information from the data, providing a small generalisation benefit on the FashionMNIST dataset.

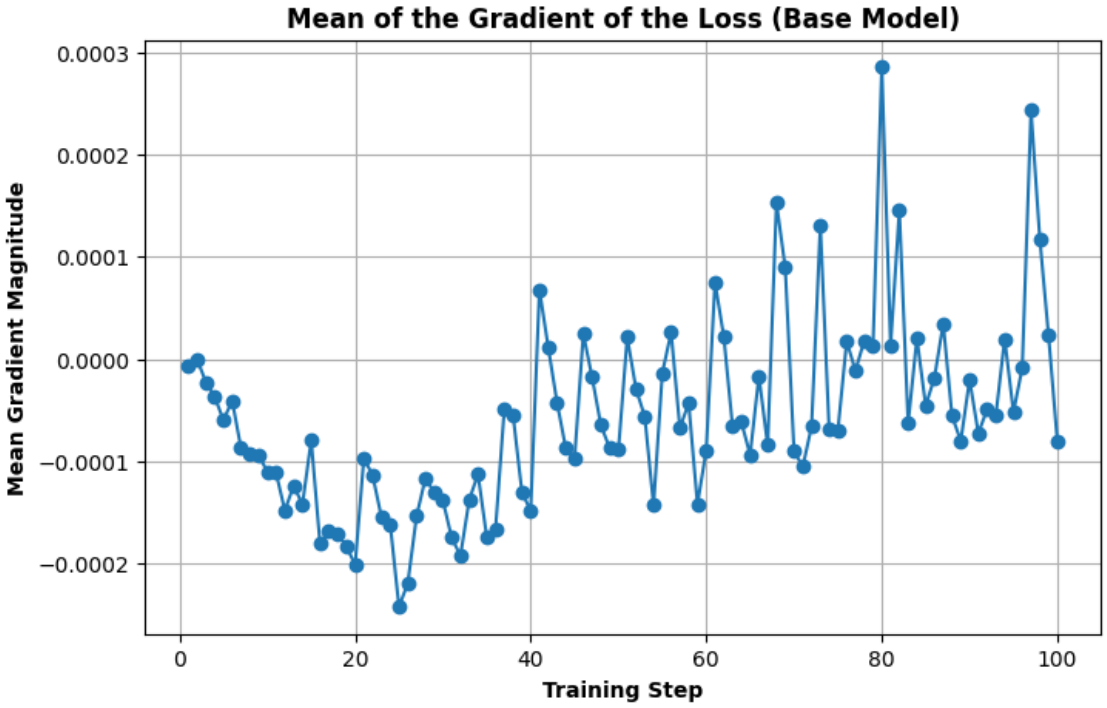
In contrast, the deeper model introduced two additional hidden layers, resulting in a total of five fully connected layers with 512 neurons each. This increased the parameter count to 830,090, which is more than the base model but still less than the wider one. Surprisingly, this configuration led to a slightly lower test accuracy than the wider model, achieving only 87.40% accuracy on average. The loss curve showed sharper fluctuations, and the accuracy curve was more erratic, indicating less stable convergence during validation. Deeper networks can suffer from issues such as vanishing gradients and optimisation difficulties, particularly when not paired with architectural enhancements like batch normalisation or residual connections. In this case, the added depth may have made it harder for the model to learn effectively, resulting in higher loss and only marginally better performance over the base model.

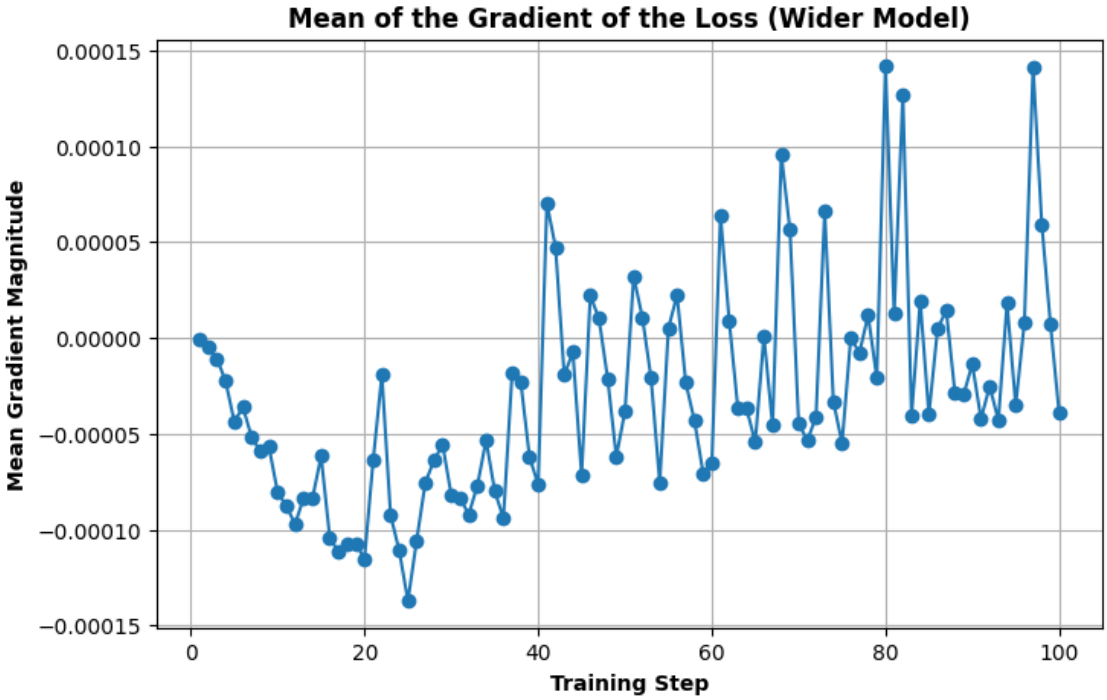
Overall, the experiments above demonstrate that widening the network can improve the training behaviour and enhance generalisation, while deepening the network makes training more unstable, but can result in performance improvments. These results suggest that for datasets like FashionMNIST, where the classification task is relatively straightforward, increasing model complexity

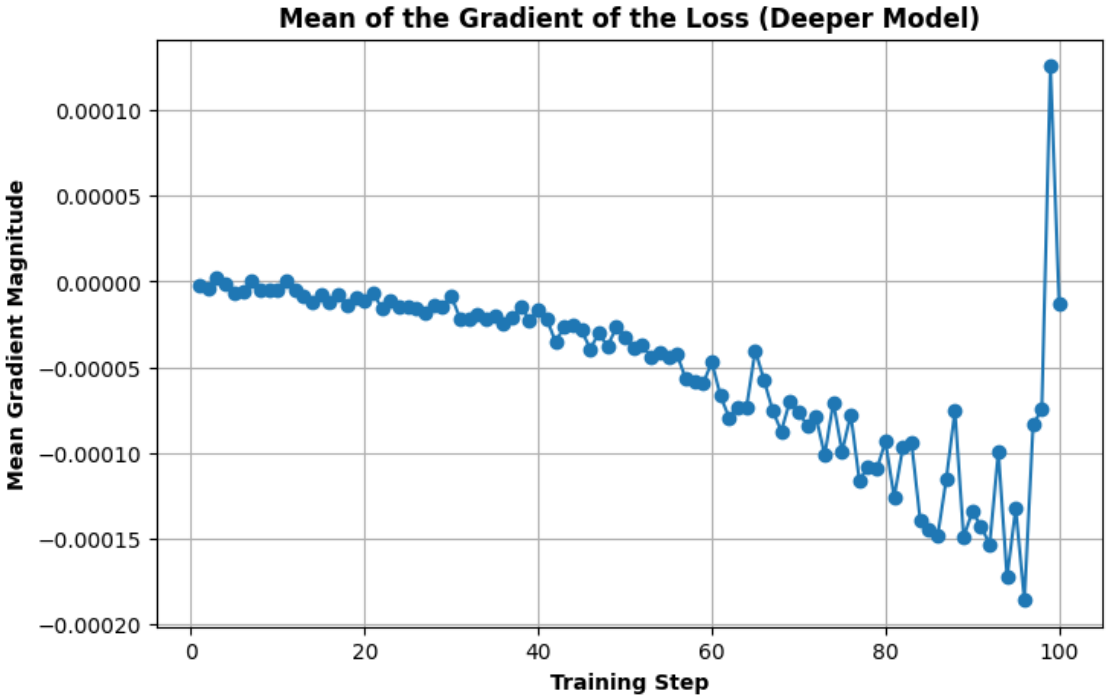
beyond a certain point provides diminishing returns.

1.1.6 Q1.6 *(2 Points)*

Calculate the mean of the gradients of the loss to all trainable parameters. Plot the gradients curve for the first 100 training steps. What are your observations? Note that this gradients will be saved with the training weight automatically after you call loss.backwards(). **Hint:** The mean of the gradients decrease.







Gradients are central to how neural networks learn, indicating how much the loss would change if a parameter were adjusted. As training progresses, gradients typically shrink toward zero, especially

near a loss minimum, where further updates have less impact, a sign of convergence. The gradient mean can be positive or negative, depending on the average direction of parameter updates. In this context, a negative mean suggests that, on average, the model is decreasing the value of its weights, while a positive mean implies that, on average, the value of the model weights is increasing. Oscillation around zero often indicates the model is near a minimum, updating weights in both directions due to high learning rates or stochasticity in gradient estimates. If gradients become too small, the model may suffer from vanishing gradients, halting learning. Conversely, excessively large gradients can lead to unstable training, known as exploding gradients.

After visualising the mean gradient of the loss with respect to all trainable parameters over the first 100 steps, key patterns emerge across different MLP architectures. In the base model *(fewest parameters)*, the mean gradient steadily decreases to about -0.0002, a typical sign of early learning. This is followed by increasing oscillations that draw the mean back toward zero, eventually spiking as high as +0.003. These spikes likely reflect overshooting caused by a high learning rate (0.1), as the model bounces back and forth across a minimum. These spikes likely reflect the model overshooting the optimal weights due to a high learning rate (0.1), causing it to bounce repeatedly across a loss minimum. The increasing size of the spikes suggests that training is becoming unstable, possibly because the model lacks the capacity to effectively capture the data or because the large updates are too coarse to allow fine adjustments.

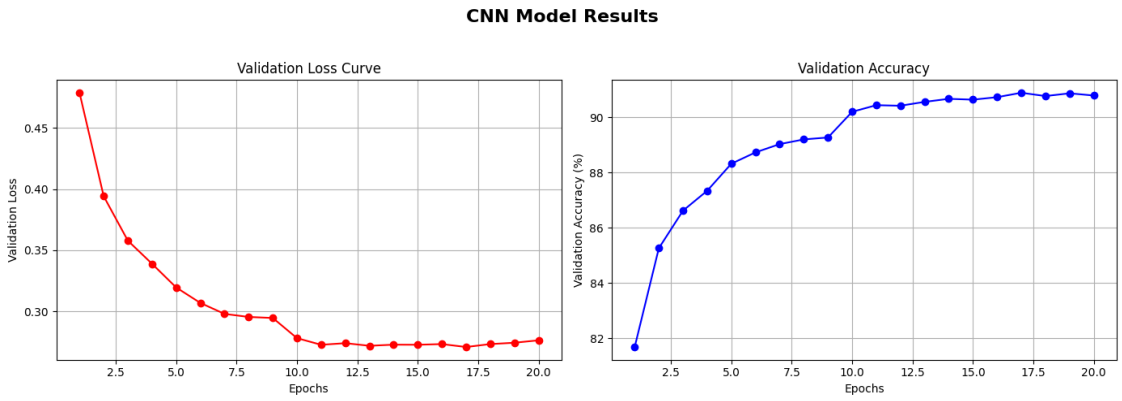
The wider model *(more neurons per layer)* shows a similar trend but with significantly smaller fluctuations, between -0.00015 and +0.00015. Like the base model, it begins with a smooth descent, followed by oscillations around zero. However, the increased number of parameters likely provides a smoothing effect, averaging out noisy gradients and resulting in more stable training. This reflects how model width can dampen abrupt changes, contributing to steadier convergence. The deeper model *(additional hidden layers)* begins with a significantly smooth descent into negative gradients, but then produces a sharp, isolated spike up to +0.0001. Unlike the wider model, this deeper architecture is more sensitive to accumulated errors across layers. The sudden spike is likely an overcorrection, a single large update caused by the learning rate being too high, especially since the step before was near zero. This underscores the importance of learning rate tuning in deeper models, where instability can amplify more easily. In summary, architecture significantly affects gradient behaviour. The base model shows high volatility, the wider model gains stability from its breadth, and the deeper model introduces stability coupled with a greater sensitivity to update dynamics. Observing these gradients is a powerful way to diagnose training behaviour and guide model and hyperparameter choices.

1.1.7 Q1.7 *(5 Points)*

Modify the network structure and training/test to use a small convolutional neural network instead of an MLP. Discuss your findings with regard to convergence, accuracy and number of parameters, relative to MLPs. **Hint:** Look at the structure of the CNN in the Workshop 3 examples.

For more explanation of Q1.7, you could refer to the following simple instructions: https://colab.research.google.com/drive/1XAsyNegGSvMf3_B6MrsXht7-fHqtJ7OW?usp=sharing.

```
Trainable parameters: 20490
```



After training and analysing Multi-Layer Perceptron (MLP) models, a simple Convolutional Neural Network (CNN) was implemented and trained using a comparable learning rate *(0.1)*. The CNN architecture includes two convolutional layers followed by a fully connected output layer. Crucially, ReLU activation functions were applied after each convolutional layer to introduce non-linearity, which enables the model to learn complex, hierarchical feature representations from the input images. This non-linearity is a significant advantage of CNNs over traditional MLPs, as it allows the network to better capture spatial hierarchies in image data. Additionally, the CNN employs max pooling layers to progressively reduce spatial dimensions, which helps reduce the number of parameters and computational cost compared to a fully connected architecture. The results of this show several key differences that can be observed regarding convergence, accuracy, and parameter count.

The CNN model converged in approximately 13 epochs, whereas the best-performing MLP reached convergence in around 15 epochs. This difference in convergence speed can be attributed to the structural differences between the two architectures. Convolutional Neural Networks incorporate inductive biases such as spatial locality and translation invariance, which enable them to learn hierarchical representations of features from image data more efficiently. While convolution and pooling operations introduce additional computational steps, these mechanisms allow the CNN to extract meaningful patterns and reduce spatial dimensions progressively, facilitating faster convergence. The reduced number of training epochs reflects the model's ability to rapidly learn low to high-level features across its layers.

As well as taking less time to converge, the CNN achieved a higher accuracy of approximately 90.5% on average, compared to the best MLP *(~88%)*. This is expected because CNNs are inherently better at processing spatial data, like images. The use of convolutional layers allows the network to learn local patterns *(edges and textures)* and build up to more abstract features in deeper layers. Additionally, the incorporation of ReLU activation layers introduces non-linearity, enabling the model to learn more complex functions and improving its capacity to generalise to the data. In contrast, MLPs treat all input pixels equally and don't capture spatial relationships, which limits their performance on image classification tasks. This is why CNNs outperform MLPs even with fewer training parameters.

An interesting and important observation is that the CNN had only 20,490 trainable parameters, compared to 669,706 for the smallest MLP, a reduction of about 31.5x. This large difference is due to weight sharing in convolutional layers. In MLPs, every neuron in one layer is connected to

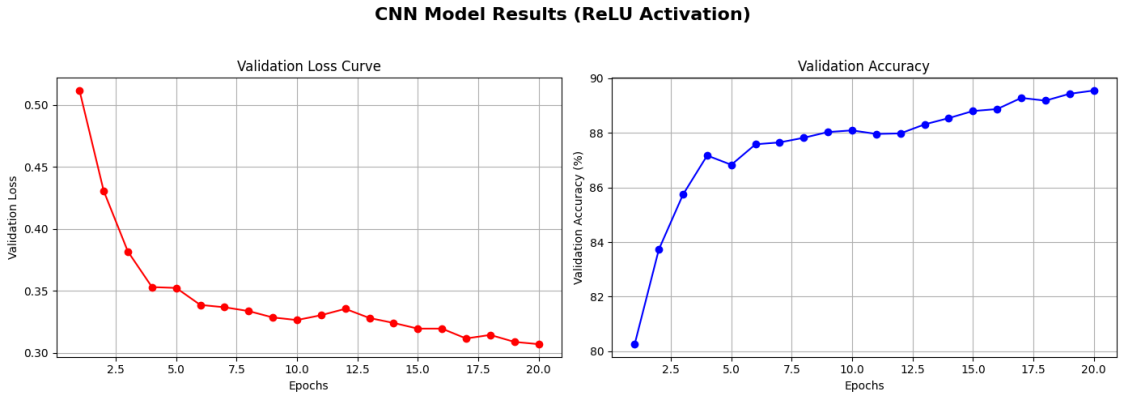
every neuron in the next, resulting in a huge number of parameters. In CNNs, each filter is applied across the entire input image, dramatically reducing the number of weights while still allowing the network to extract relevant features. Pooling layers further reduce spatial dimensions, leading to a much more compact and accurate model.

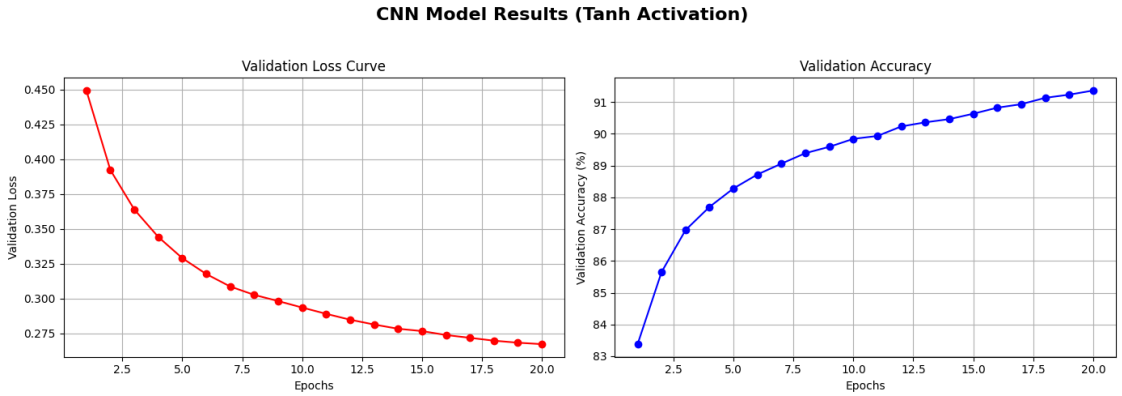
---

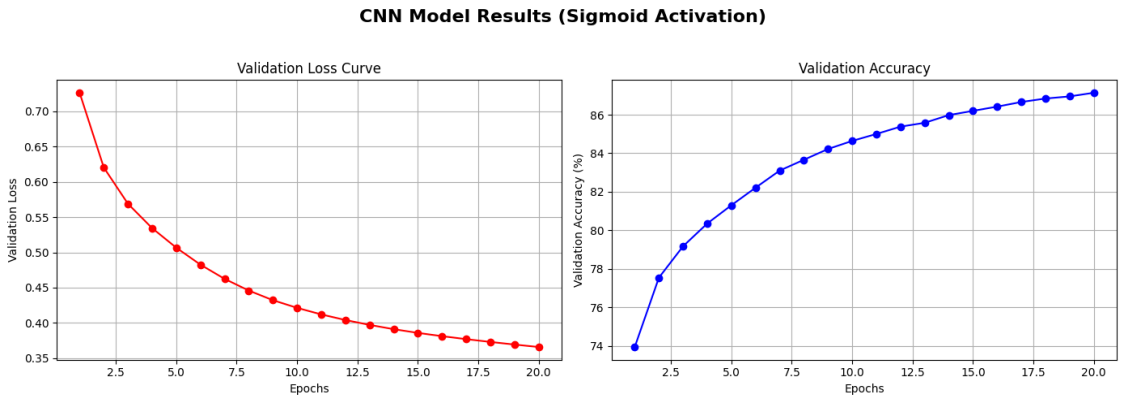
2 Question 2: Optional Bonus Question *(5 Marks, 20% Bonus Marks)*

2.0.1 Q2.1 *(2 Points)*

Experiment with different activation functions (ReLU, Tanh, Sigmoid) and analyse their impact on training performance.







Similar to the convolutional neural network (CNN) used in part 1.7, the model implemented above consists of two convolutional layers, each followed by an activation layer and a max-pooling operation, before passing through a fully connected linear layer. This structure was chosen for its balance between model complexity and interpretability, especially given the nature of the dataset. The first convolutional layer captures low-level features such as edges, while the second layer builds on these to detect more abstract patterns. By applying a padding of one in the convolutional layers, the spatial dimensions of the input are preserved, which helps retain information near the borders. Bias terms were included in each layer to increase the flexibility of the learned transformations, and the max-pooling operations serve to reduce the spatial resolution, encourage translational invariance, and decrease computational load.

The inclusion of activation functions is critical, as they introduce non-linearity, allowing the model to learn complex, real-world patterns that a purely linear system could not capture. ReLU (Rectified Linear Unit), Tanh, and Sigmoid functions were each tested to observe their impact on training behaviour and model performance. The training was conducted using a consistent number of epochs and learning rate across all three activation setups *(e.g. 20 epochs with a learning rate of 0.1)*, to ensure a fair comparison.

With ReLU, the model achieved approximately 90% accuracy on average when validated on the test dataset, a clear improvement over the baseline models used in question 1. Additionally, the validation loss decreased steadily and corrected itself effectively within the first few epochs. This behaviour is typical of ReLU due to its ability to maintain strong gradients and avoid saturation, enabling rapid and stable learning in the early stages.

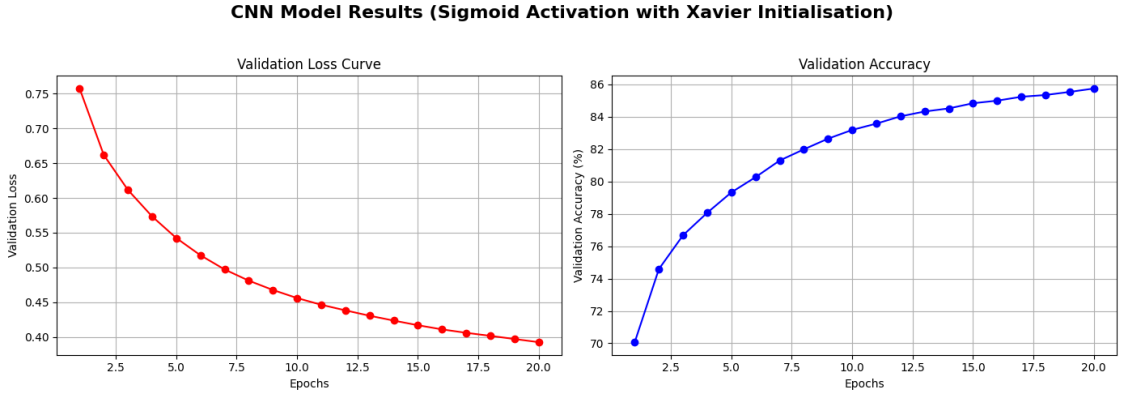
Interestingly, the model using the Tanh activation function outperformed ReLU, achieving a test accuracy of approximately 91.5% on average. The loss curve was very smooth, indicating stable convergence throughout training. While Tanh generally performs best with zero-centred inputs, this result suggests that even without normalising the Fashion-MNIST dataset around zero *[-1,1]*, Tanh was able to effectively transform the positively skewed input values. This may be due to its non-linearity and ability to output both positive and negative values, which still supports a more balanced gradient flow compared to the Sigmoid activation function. Although ReLU is typically more robust to input scale, in this case, Tanh's smoother gradient across its range appears to have offered a slight advantage in convergence and final accuracy.

In contrast, the Sigmoid activation function led to slower learning and lower overall performance,

with the model only reaching around 86.5% accuracy on average. The gradient was not as steep during the early epochs, and the model converged more slowly. This is a known limitation of the Sigmoid function, which tends to suffer from vanishing gradients as the output saturates for large input values. While preprocessing operations like Xavier initialisation may help mitigate this by keeping the signal variance stable across layers, the fundamental limitations of Sigmoid in deep networks mean it typically underperforms compared to ReLU or Tanh in this kind of setting.

2.0.2 Q2.2 *(1 Point)*

In particular, focus your analysis on the Sigmoid activation function and discuss your finding of training with and without Xavier initialisation. You may use the provided code for Xavier initialisation for this.

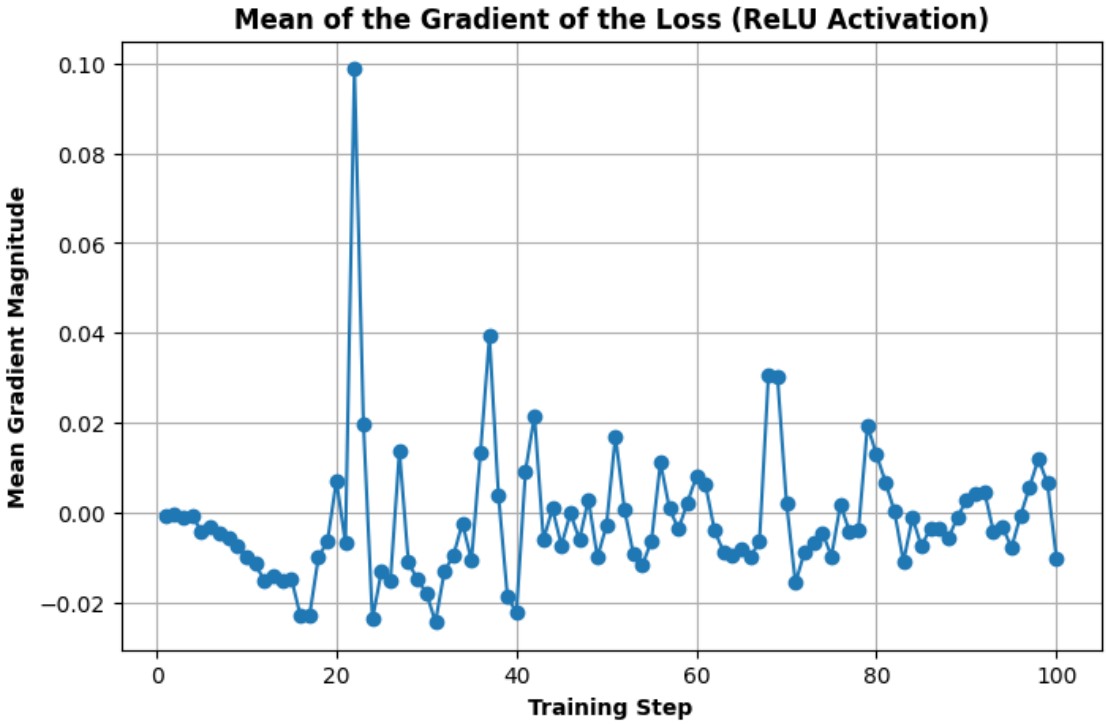


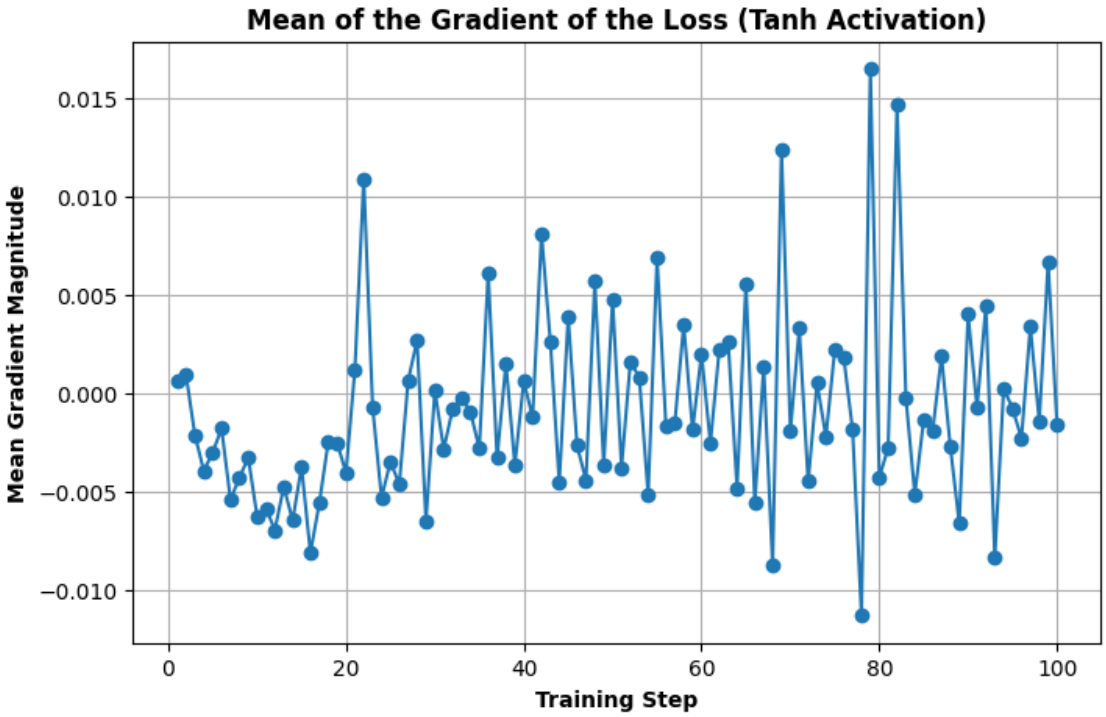
When using the Sigmoid activation function without any special weight initialisation, the model exhibited slow convergence and achieved a lower final accuracy *(approximately 86.5%)* compared to models using ReLU or Tanh. This is expected behaviour, as Sigmoid activations are known to suffer from vanishing gradients, especially when deeper in the network or when weights are poorly scaled. To try and improve the results for the Sigmoid Function, Xavier initialisation was applied in an attempt to mitigate this issue.

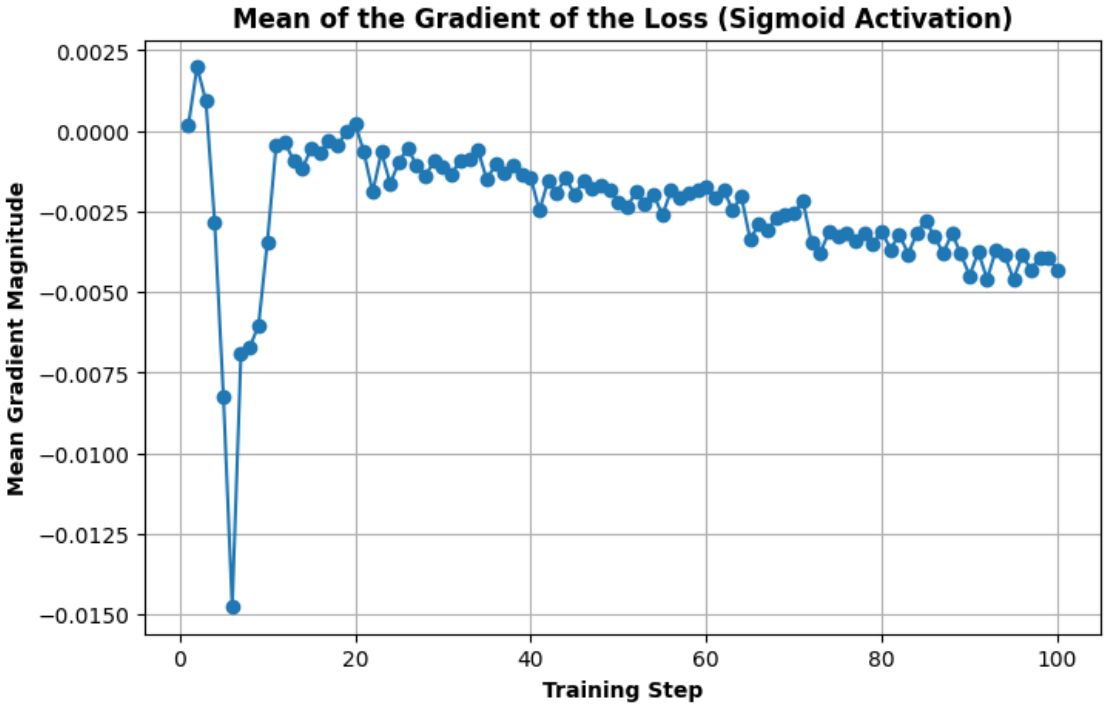
Theoretically, Xavier initialisation helps maintain stable gradients throughout training by scaling the initial weights based on the number of input and output connections, a method that works particularly well with symmetric activation functions like Sigmoid. However, in practice, the model using Sigmoid activation layers with Xavier initialisation actually performed about the same, achieving an accuracy of approximately 86% on average. One likely explanation is that the Fashion-MNIST dataset is relatively simple, and the network is shallow, so the benefits of careful weight scaling are less pronounced. In such cases, small variations due to weight initialisation may have a minor impact, and training instability due to suboptimal initialisation is unlikely to manifest significantly. Another possibility is that Sigmoid's intrinsic limitations *(such as saturation at extremes)* still hinder its learning dynamics, even with better-initialised weights.

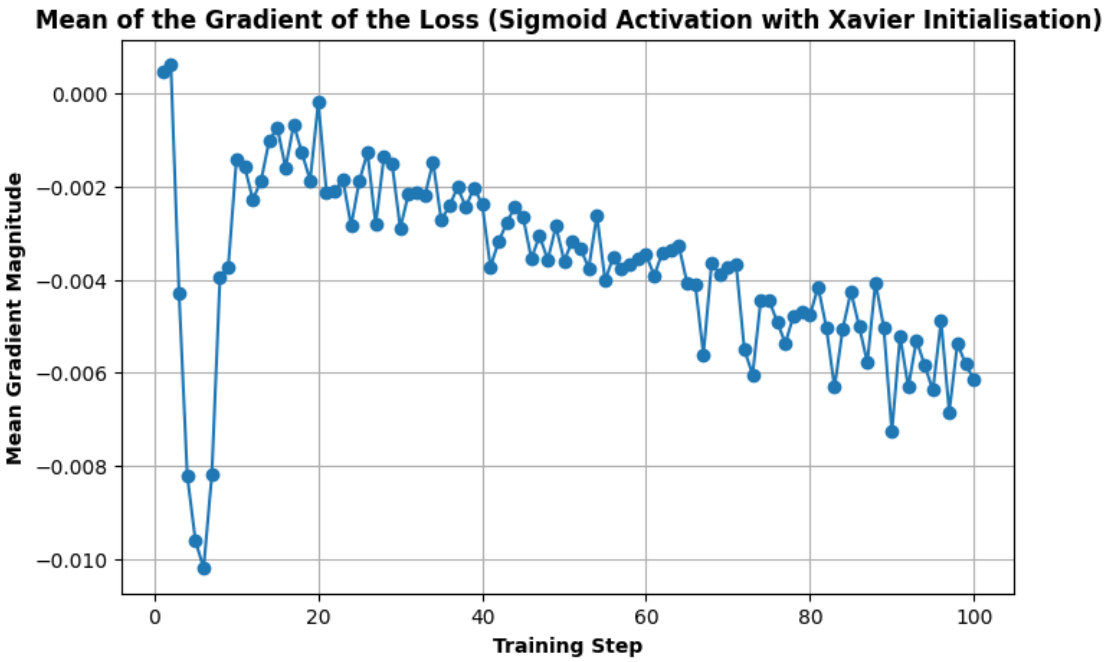
2.0.3 Q2.3 *(1 Point)*

Additionally, plot both the gradient and loss curves for your experiments. For gradient analysis, you may select one representative layer to monitor throughout training and briefly explain your choice.




Mean of the Gradient of the Loss (Tanh Activation)
Mean Gradient Magnitude
0.015
0.010
0.005
0.000
−0.005
−0.010
0
20
40
60
80
100
Training Step


Mean of the Gradient of the Loss (Sigmoid Activation)
Mean Gradient Magnitude
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.0125
−0.0150
0
20
40
60
80
100
Training Step



For this analysis, the first convolutional layer `conv1` was selected as the representative layer for monitoring gradients. This layer was chosen because it is directly impacted by the activation function and is close to the input. As a result, it provides a clear view of how well gradients are able to propagate backwards through the network. If vanishing gradients occur due to poor activation or initialisation, they are most likely to show up clearly in the earliest layers.

In the case of the **ReLU** activation function, the gradient mean oscillated between approximately -0.02 and 0.02, with rare but significant spikes reaching as high as 0.1 during the early training steps. These sharp increases likely correspond to batches with high error or increased neuron activity, which trigger strong updates. ReLU is known for sparse activations, many neurons may output zero and stop contributing to learning *(so-called "dying ReLU" problem)*, but when activated, they can transmit strong gradients. This explains the sharp, intermittent spikes, especially in early training, where error signals are large. Overall, this behaviour suggests fast learning but some instability.

In contrast, the **Tanh** activation produced much smoother gradients, fluctuating within -0.005 to 0.005 with occasional peaks up to 0.015. This stability is likely due to Tanh's continuous, symmetric, and zero-centred nature, which encourages balanced weight updates and mitigates the risk of dead neurons. The result is a more consistent flow of gradients during backpropagation, albeit potentially at the cost of slower learning compared to ReLU, especially in deeper networks where saturation near ± 1 may still cause gradients to shrink. This is evident in section 2.1 above, where Tanh doesn't quite reach convergence in 20 epochs while training.

The **Sigmoid** function exhibited a notably different profile. Early training saw a steep drop in gradient mean from ~0 to -0.015, followed by a quick recovery toward zero and a steady, negative trend from around -0.0005 to -0.005. This behaviour reflects the known limitations of sigmoid activations. The initial drop likely corresponds to the model pushing activations into the saturated

regions of the sigmoid function, where gradients approach zero. However, the rapid recovery suggests that the model quickly adjusted its weights to bring activations back into the non-saturated, more responsive region of the sigmoid curve *(around 0)*. Once this stabilisation occurred, gradients settled into a shallow, negative trend, reflecting a slow and cautious learning dynamic. The persistently small magnitude of the gradient implies limited learning ability, the model is updating weights, but very conservatively. This is characteristic of sigmoid functions in deeper networks or those without carefully chosen initialisation strategies. In this case, although the model partially avoided long-term saturation, it still struggled to maintain a gradient strength sufficient for effective learning.

Introducing **Xavier initialisation** to the Sigmoid-activated model helped reduce the severity of the initial gradient drop *(limiting it to around -0.01)* and introduced more pronounced fluctuations afterwards. While the gradient remained entirely negative, these post-spike oscillations suggest that Xavier improved gradient flow, allowing the model to respond more dynamically during training. This contrasts with the flatter, more subdued gradient curve seen without Xavier, which indicates more severe vanishing gradients. However, the increased activity did not fully translate into better learning; the updates remained unstable and conservative, likely due to the inherent limitations of the sigmoid activation. As a result, the model still struggled to train effectively and ultimately performed slightly worse than its non-Xavier counterpart.

Overall, this analysis underscores how activation function choice and weight initialisation jointly influence gradient dynamics. ReLU enables strong updates but can be volatile, Tanh promotes stability, and Sigmoid is highly sensitive to vanishing gradients, especially without careful initialisation such as Xavier.

2.0.4 Q2.4 *(1 Point)*

Discuss how gradients and loss behave across the network for different activation functions and initialisation methods if you see any difference.

Across all experiments, the behaviour of gradients and loss during training varied notably with the choice of activation function and weight initialisation. ReLU produced strong, fluctuating gradients that enabled fast convergence and high accuracy, although its *"spiky"* pattern reflects its sparse activation nature. Tanh led to smoother, more stable gradients and slightly better performance, likely due to its symmetric and zero-centred output that supports more consistent learning dynamics. In contrast, Sigmoid exhibited vanishing gradients, especially later in training, leading to slower convergence and reduced accuracy. While Xavier initialisation partially improved gradient flow in the Sigmoid model by reducing the severity of early saturation and enabling more activity later in training, it also introduced instability and did not fully resolve the function's tendency toward vanishing gradients. These observations highlight the critical interplay between activation functions and initialisation strategies in maintaining effective gradient flow, ensuring fast convergence, and maximising model performance.
