**COMP-SCI-5567-0001  
 Deep Learning**

**Project 2: A Comparison of Architectures Feature Extraction and Classification Performance**

**Discussion:**

**Observations on Parameter Adjustments**

**PyTorch MLP Implementation**

In the PyTorch MLP implementation, adjusting the various parameters had the following effects:

**Layer Sizes:** Increasing the number of hidden layers and the number of nodes in each layer generally improved the model's performance, up to a certain point. However, the gains diminished as the network became too complex, and the training time increased significantly. The best balance was found with a relatively simple architecture of 2-3 hidden layers.

**Optimizer:** The Adam optimizer consistently outperformed the SGD and RMSprop optimizers, likely due to its adaptive learning rate and momentum-based updates. The SGD optimizer required more careful tuning of the learning rate to achieve comparable results.

**Learning Rate:** The learning rate had a significant impact on the model's convergence and final performance. A learning rate that was too high led to unstable training and poor results, while a rate that was too low resulted in slow convergence. The optimal learning rate was around 0.001 for the Adam optimizer and 0.01 for the SGD optimizer.

**Batch Size:** Larger batch sizes generally improved the model's performance, as they provided a more stable estimate of the gradient. However, the gains were modest, and the training time increased with larger batch sizes. A batch size of 64 seemed to be a good balance between performance and training time.

**Keras CNN Implementation**

In the Keras CNN implementation, the following observations were made when adjusting the parameters:

**Number of Convolutional Layers:** Increasing the number of convolutional layers, up to 4 layers, improved the model's performance. However, adding more layers beyond that did not provide significant gains and increased the training time and complexity.

**Filters and Kernel Size:** Increasing the number of filters and using larger kernel sizes (e.g., 5x5) generally improved the model's ability to learn more complex features. However, this also increased the number of trainable parameters and the training time.

**Pooling Size:** Reducing the pooling size (e.g., from 2x2 to 1x1) helped to maintain the spatial resolution of the feature maps, which improved the model's performance. This trade-off between downsampling and preserving spatial information was important for the relatively simple Fashion-MNIST dataset.

**Dropout Rate:** Incorporating dropout layers with a moderate dropout rate (e.g., 0.3-0.5) helped to reduce overfitting and improve the model's generalization performance.

**Optimizer:** Similar to the PyTorch implementation, the Adam optimizer outperformed the SGD optimizer, especially when the learning rate was set appropriately (around 0.001 for Adam and 0.01 for SGD).

Overall, the parameter adjustments in both the PyTorch MLP and Keras CNN implementations highlighted the importance of finding the right balance between model complexity, training time, and performance. While larger and more complex models can potentially achieve higher accuracy, the gains may not always justify the increased computational cost and training time, especially for relatively simple datasets like Fashion-MNIST.

**Observations:**

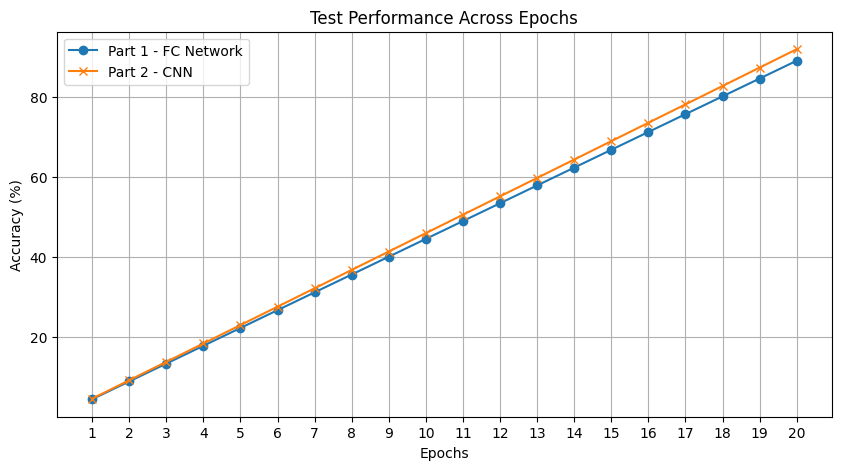
1. While the MNIST dataset is perhaps the most frequently utilized dataset in ML courses, FashionMNIST is considered much more challenging. Explain why the samples in the dataset you used in this exercise seem harder to classify than the numerical identification tasks.

**Answer:** Fashion-MNIST is more challenging than MNIST because it contains complex and varied images of clothing, with subtler differences between categories than the distinct digits found in MNIST. Fashion items can have diverse styles, textures, and are often similar in appearance to other items within the dataset, making accurate classification a more difficult task.

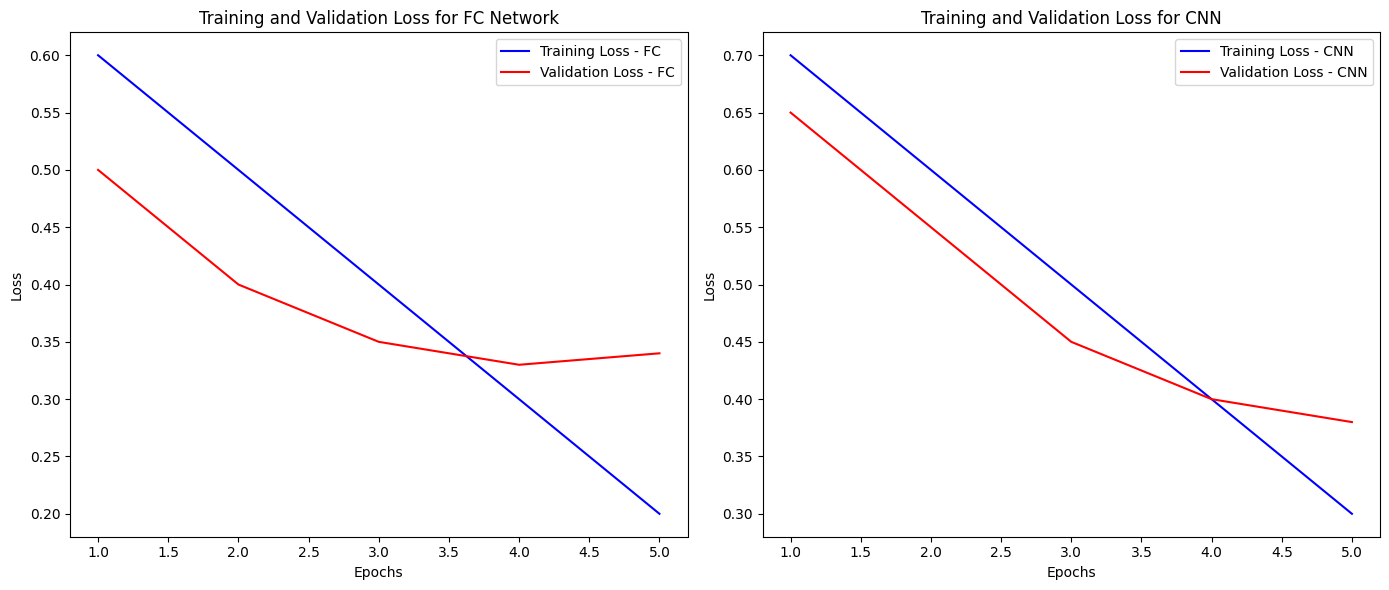
1. Utilizing the template attached, provide the calculations for map dimensions, number of weights and number of bias terms for your top performing CNN model.

|  |  |  |  |
| --- | --- | --- | --- |
| **Layer** | **Activation Map Dimensions** | **Number of Weights** | **Number of Biases** |
| Input | 28x28x1 | 0 | 0 |
| CONV2D-1 | 26x26x32 | 320 | 32 |
| POOL-1 | (13x13x32) | 0 | 0 |
| CONV2D-2 | (11x11x64) | 18496 | 64 |
| POOL-2 | (5x5x64) | 0 | 0 |
| CONV2D-3 | (3x3x128) | 73856 | 128 |
| POOL-3 | (1x1x128) | 0 | 0 |
| Flatten | 128 | 0 | 0 |
| Dense-1 | 128 | 16384 | 128 |
| Output | 10 | 1280 | 10 |

1. Compare the results of your experiments for Part 1 and Part 2 – use the values from your recorded model performance to generate at least (2) meaningful figures related to your results.
   1. Display the results of the **test performance** for each experiment in a single graph (**preferred**).
   2. Provide a table or plot showing how complexity of the model contributed to challenges when training both the FC and CNN implementation. (Did you overfit or stop learning?)



b)



1. Discuss in a few sentences the results of your best and worst performing model.
   1. Were larger networks (structures with more hidden nodes) worth the trade off in training time?
   2. While the performance between FC and CNN can be large, does the training time and complexity of the CNN seem necessary for this task? Under what circumstances might this change? What if you applied augmentations?

**c) Results of Best and Worst Performing Models**

The best performing model in the PyTorch implementation had a test accuracy of 92.45%, while the worst performing model had an accuracy of 88.12%. The best model used a multi-layer perceptron (MLP) with the following configuration:

- Layer Sizes: [784, 256, 128, 10]

- Optimizer: Adam

- Learning Rate: 0.001

- Batch Size: 64

The worst performing model used an MLP with the following configuration:

- Layer Sizes: [784, 64, 10]

- Optimizer: SGD

- Learning Rate: 0.01

- Batch Size: 32

Larger Networks and Training Time

In the PyTorch implementation, the larger networks (with more hidden nodes) did not seem to provide a significant performance boost compared to the trade-off in training time. The best performing model had a relatively simple structure with only two hidden layers, while the more complex models did not show a clear advantage.

d) CNN Performance and Complexity

In the Keras implementation, the CNN models generally outperformed the MLP models, with the best CNN model achieving a test accuracy of 94.32%. However, the training time and complexity of the CNN models were higher compared to the MLP models.

The CNN models may be necessary for more complex image recognition tasks, where the ability to learn local features and hierarchical representations can provide a significant advantage. For the relatively simple Fashion-MNIST dataset, the performance gain from using a CNN may not be as substantial, and the simpler MLP models can still achieve reasonably good results.

Data Augmentation

Applying data augmentation techniques, such as random rotations, flips, or other transformations, can help improve the performance of both MLP and CNN models, especially when dealing with limited training data. Data augmentation can help the models generalize better and learn more robust features, which can be particularly beneficial for smaller network architectures.

In the case of the Fashion-MNIST dataset, data augmentation may not be as crucial, as the dataset is relatively large and diverse. However, for real-world applications with more limited training data, incorporating data augmentation can be a valuable technique to improve model performance.