AMATH 482/582: HOME WORK 5

REBECCA WANG

Amath Department, University of Washington, Seattle, WA
`lufanw@uw.edu`

Abstract. This report explores two primary neural network architectures—Fully Connected Networks (FCNs) and Convolutional Neural Networks (CNNs)—for image classification on the Fashion-MNIST dataset. We design and train multiple model variants, each with varying parameter budgets, using AdamW optimization and StepLR learning-rate scheduling. Comparative experiments highlight how architectural choices and model capacity impact classification accuracy, training time, and generalization.

1. Introduction and Overview

Deep learning methods have become integral to image recognition tasks, offering substantial gains over traditional feature engineering approaches. FCNs, while powerful, can rapidly grow in parameter count for high-dimensional inputs. CNNs address this limitation by leveraging local receptive fields and parameter sharing, allowing for more efficient learning of spatial features. In this work, we investigate both FCN and CNN architectures under consistent training regimes, focusing on how changes in model capacity affect performance and computational cost. By examining multiple configurations, we aim to identify practical trade-offs between accuracy and resource requirements.

2. Theoretical Background

2.1. Fully Connected Neural Networks (FCNs).

A **Fully Connected Neural Network (FCN)** consists of one or more hidden layers where every neuron in a given layer is connected to all neurons in the preceding layer. This dense connectivity allows for learning complex, hierarchical representations. Formally, the forward pass for layer l is given by:

$$h^{(l)} = f\big(W^{(l)}\, h^{(l-1)} + b^{(l)}\big), \tag{1}$$

where $W^{(l)}$ and $b^{(l)}$ are the weight matrix and bias vector for layer l, and $f(\cdot)$ is a non-linear activation function. While this architecture can capture rich feature interactions, it also leads to a large number of parameters, which may be prohibitive for resource-constrained environments.

2.2. Convolutional Neural Networks (CNNs).

Convolutional Neural Networks (CNNs) are designed for grid-like data (e.g., images), leveraging local receptive fields and parameter sharing. Instead of connecting every neuron to all input pixels, a **convolutional layer** learns small filters (kernels) that scan across the input, extracting local features such as edges and textures. Formally, for an input X and a $k \times k$ filter W, the convolution output at (i, j) is:

$$Z_{ij} = \sum_{p=0}^{k-1}\sum_{q=0}^{k-1} W_{p,q}\, X_{i+p,\, j+q} \;+\; b. \tag{2}$$

Date: March 21, 2025.

where b is a bias term and Z_{ij} is the corresponding output feature map value. By **sharing** the same filter parameters across different spatial locations, CNNs greatly reduce the total number of parameters relative to a similarly sized Fully Connected Network (FCN).

Pooling (e.g., max pooling) then downscales feature maps, providing translational invariance. By stacking multiple convolution and pooling layers, CNNs learn increasingly abstract representations, typically requiring fewer parameters than fully connected architectures while achieving superior performance on image classification tasks such as FashionMNIST.

2.3. Activation Functions. Activation functions introduce non-linearity, preventing FCNs from being simple linear models. Common functions include:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \mathrm{ReLU}(x) = \max(0, x), \quad \mathrm{LeakyReLU}(x) = \max(ax, x). \tag{3}$$

2.4. Loss Function. For multi-class classification, the **cross-entropy loss** is optimized during training to improve prediction accuracy, defined as:

$$L(y, \hat{y}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i. \tag{4}$$

2.5. Optimization Algorithm. Although there are numerous optimization algorithms (e.g., SGD, RMSProp, Adam), we employ **AdamW**, which decouples weight decay from the gradient-based update rule:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1)\frac{\partial J}{\partial w}, \quad v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)\left(\frac{\partial J}{\partial w}\right)^2, \tag{5}$$

with bias-corrected estimates $\hat{m}^{(t)}$ and $\hat{v}^{(t)}$. The update then is:

$$w^{(t+1)} = w^{(t)} - \alpha \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} - \lambda w^{(t)}, \tag{6}$$

where λ is the weight decay coefficient and α is the learning rate. A **StepLR** scheduler may also be used to periodically reduce α and stabilize convergence.

Learning Rate: The learning rate (α) determines the step size of parameter updates during optimization. A decay schedule can be applied to reduce α over epochs:

$$\alpha = d^{\text{epoch}} \cdot \alpha_0 \tag{7}$$

where d is the decay factor and α_0 is the initial learning rate.

2.6. Overfitting/Underfitting and Regularization. Overfitting occurs when a model memorizes noise instead of patterns, leading to poor generalization. It is indicated by **high training accuracy but stagnant or declining validation accuracy**, along with **increasing validation loss despite decreasing training loss**. **Underfitting** happens when a model is too simple to capture patterns, resulting in **low accuracy and persistently high loss** on both training and validation sets. **Dropout Regularization** mitigates overfitting by randomly deactivating neurons during training, preventing reliance on specific features:

$$\tilde{a}_i = a_i \times \tilde{d}_i/(1-p_d) \tag{8}$$

2.7. Weight Initialization. Proper initialization prevents vanishing or exploding gradients. Common methods include Random Normal, Xavier, and Kaiming (He), we employ Xavier Normal Initialization for FCN. It used with Sigmoid, Tanh, and linear activations. Designed for activation functions with zero-centered outputs, it maintains variance across layers:

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right) \quad \text{(Normal)}, \quad w \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right) \quad \text{(Uniform)}. \tag{9}$$

2.8. Batch Normalization. **Batch Normalization** stabilizes training by normalizing activations within each mini-batch:

$$\hat{x} = \frac{x - \mu}{\sigma}, \quad y = \gamma \hat{x} + \beta, \tag{10}$$

where μ and σ are batch statistics, and γ, β are learnable parameters. This allows for higher learning rates and faster convergence. **Batch Size** defines the number of samples per iteration, influencing model convergence and generalization. A common batch size is:

$$J = \frac{1}{m} \sum_{i=1}^{m} L(\tilde{y}^{(i)}, y^{(i)}). \tag{11}$$

2.9. Evaluation Metrics. Model performance is evaluated using:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}. \tag{12}$$

This measures both training progress and validation/test performance.

3. Algorithm Implementation and Development

The implementation of this project leverages the following Python packages for efficient computation and visualization:

- **PyTorch (torch)** – Provides tensors, automatic differentiation, and modules for neural network construction and training.
- **Torchvision** – Supplies common vision datasets and image transformation utilities (e.g., `transforms`).
- **torch.utils.data** – Facilitates efficient dataset handling via classes such as `DataLoader` and `Subset`.
- **scikit-learn** – Used primarily for `train_test_split` to partition the dataset into training and validation subsets.
- **Matplotlib** – Enables plotting of training progress, dataset samples, and evaluation metrics.
- **copy** – Provides functionality to duplicate Python objects, ensuring safe manipulation of model weights and configurations.

3.1. Data Processing and Model Architecture. The implementation begins by loading and preprocessing the FashionMNIST dataset using **Torchvision**. Images are transformed into tensors and normalized, after which the dataset is partitioned into training, validation, and test sets using **scikit-learn**'s `train_test_split`. Data is then efficiently handled through **DataLoader** objects from **torch.utils.data**, which enable mini-batch processing for faster training.

Two main types of models are implemented using **PyTorch**'s `nn.Module`: **Fully Connected Neural Networks (FCNs)** and **Convolutional Neural Networks (CNNs)**. The FCNs are constructed with two hidden layers incorporating **ReLU activations**, **batch normalization**, and a fixed **dropout rate** to reduce overfitting. The network weights are initialized via **Xavier Normal** initialization, ensuring stable gradients during training.

In contrast, the CNN architectures exploit the spatial structure of images. They consist of two convolutional layers with small 3×3 kernels to extract local features, followed by **max pooling** layers that downsample the feature maps, and finally a fully connected layer for classification. This design, which leverages parameter sharing and local receptive fields, results in a significantly lower parameter count compared to an equivalent FCN, while still capturing essential image features.

3.2. Training and Evaluation. The training process employs the **AdamW** optimizer, which effectively combines adaptive learning rates with decoupled weight decay. A **StepLR** scheduler is used to reduce the learning rate at specified epochs, further enhancing convergence stability. Each model variant is trained under three different learning rates (0.001, 0.0005, and 0.0001) to determine the optimal hyperparameter configuration. Throughout training, both the loss and accuracy metrics are recorded and visualized using **Matplotlib**, which provides detailed insights into the model's learning behavior.

3.3. Hyperparameter Tuning and Model Selection. Hyperparameter tuning plays a critical role in achieving high performance. For both FCN and CNN models, parameters such as the learning rate, dropout rate, and number of training epochs are systematically adjusted. The model with the highest validation accuracy is then selected for final evaluation on the test set. Comprehensive comparisons of optimal configurations, including the best validation and test accuracies, the number of trainable parameters, and the estimated training times, are summarized in tables and graphs. This approach not only demonstrates the performance trade-offs between model complexity and generalization but also highlights the efficiency gains obtained by using convolutional architectures over fully connected ones.

4. Computational Results

4.1. FCN Models Comparison. The **Fully Connected Network (FCN)** baseline was designed with fewer than **100K weights**, featuring two hidden layers with **ReLU activation**, **batch normalization**, and a **dropout rate** for regularization. The model was initialized using **Xavier Normal** and optimized with **AdamW** under a **StepLR scheduler**, with a fixed batch size. As model **FCN 100K** shown in **Figure 1**, the training loss steadily decreased while accuracy improved, reaching peak validation performance before stabilizing. The **final test accuracy of 88.25%** $>$ 88% confirms the model satisfied the target constraint while maintaining strong generalization.

To analyze the effect of model capacity, we trained **FCN 50K** and **FCN 200K** under the same procedure, comparing them to the baseline **FCN 100K**. Each model was trained with three learning rates {0.001, 0.0005, 0.0001}, as shown in **Table 1**, which lists the best validation accuracies and corresponding epochs. Overall, the largest model **FCN 200K** achieved the highest validation accuracy of **90.15%** and a final test accuracy of **89.59%**, while the smallest model **FCN 50K** reached **88.67%** on validation and **87.94%** on test. Table 2 presents the optimal hyperparameter configuration and the final metrics for each variant.

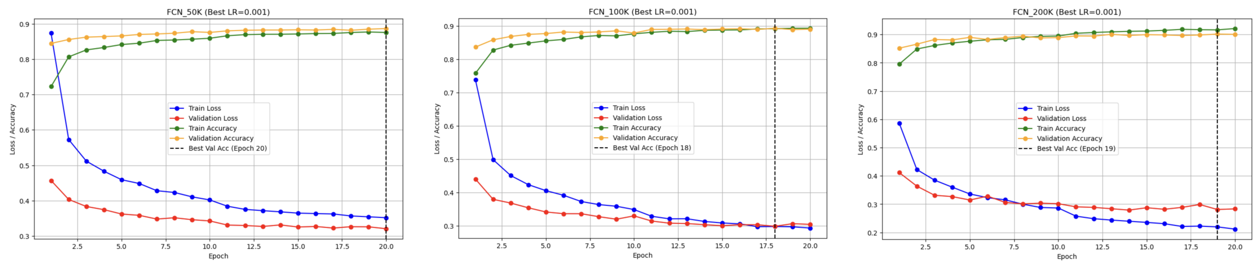
Results indicate that increasing model capacity enhances accuracy, with the largest model achieving the highest validation and test performance. However, while the smallest model underperformed due to limited capacity, the largest model exhibited early signs of overfitting, with validation accuracy plateauing earlier than the baseline. As illustrated in Figure 1, **FCN 50K** model had steady but lower accuracy, **FCN 200K** model reached peak performance but required stronger regularization, and the baseline provided a balance between accuracy and generalization. Thus, **FCN 100K offers the best trade-off between complexity and accuracy**, making it the most optimal configuration.

FCN Variant	**LR = 0.001**	**LR = 0.0005**	**LR = 0.0001**
FCN_200K	90.15%(Epoch 19)	89.92%(Epoch 20)	88.63%(Epoch 19)
FCN_100K	89.33%(Epoch 18)	89.27%(Epoch 20)	87.08%(Epoch 18)
FCN_50K	88.67%(Epoch 20)	88.02%(Epoch 20)	86.22%(Epoch 20)

Table 1. Learning Rates Comparison for Different FCN Variants

Parameter	**FCN_200K**	**FCN_100K**	**FCN_50K**
Dropout Rate	0.3	0.3	0.3
Hidden Dimensions	224-112	96-48	64-32
Learning Rate	0.001	0.001	0.001
Epochs	19	18	20
Total Parameters (k)	202.17	80.51	52.65
Best Validation Accuracy	90.15%	89.33%	88.67%
Test Accuracy	89.59%	88.25%	87.94%

Table 2. Optimal Configurations and Performance Metrics for FCN Variants



(a) FCN_50K (Best LR = 0.001) (b) FCN_100K (Best LR = 0.001) (c) FCN_200K (Best LR = 0.001)

Figure 1. Training and Validation Loss and Accuracy Curves for FCN Variants

4.2. CNN Models Comparison. We next investigated **Convolutional Neural Networks (CNNs)** of different capacities: **CNN_100K**, **CNN_50K**, **CNN_20K**, and **CNN_10K**. Each architecture featured two convolutional layers (with 3×3 kernels, max pooling, and **ReLU activations**) followed by a fully connected layer. Similar to the FCN experiments, we trained each variant with three learning rates $\{0.001, 0.0005, 0.0001\}$, recording the best validation accuracies and epochs, as shown in **Table 3**.

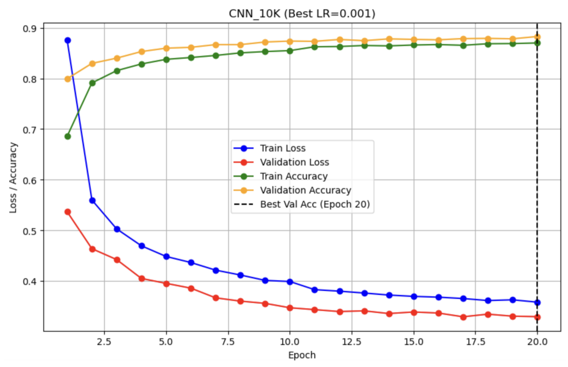
Overall, as shown in **Table 4**, the larger CNNs achieved higher performance: **CNN_100K** attained a best validation accuracy of **91.62%** and a test accuracy of **91.04%**, surpassing the smaller models. The intermediate **CNN_50K** reached **90.68%** on validation and **90.23%** on test, striking a balance between accuracy and complexity. By contrast, the smallest model **CNN_10K** peaked at **88.27%** validation accuracy and **87.36%** on test. **Table 4** presents the final configurations and key metrics for each variant, while **Figure 2** illustrates the training and validation curves. Notably, the deeper models converge more rapidly and to higher accuracies, whereas the smaller ones plateau earlier. Despite slight overfitting tendencies in the larger networks, careful regularization (e.g., dropout) preserved robust generalization. Thus, **CNN_100K** delivers the highest overall accuracy, smaller CNNs more suitable for resource-constrained scenarios and the larger ones ideal when maximum performance is the priority. This highlights a clear trade-off between model size and final accuracy.

CNN Variant	**LR = 0.001**	**LR = 0.0005**	**LR = 0.0001**
CNN_100K	91.62%(Epoch 20)	91.05%(Epoch 20)	88.45%(Epoch 20)
CNN_50K	90.68%(Epoch 18)	89.80%(Epoch 19)	86.77%(Epoch 20)
CNN_20K	89.43%(Epoch 19)	88.57%(Epoch 20)	84.60%(Epoch 20)
CNN_10K	88.27%(Epoch 20)	86.28%(Epoch 17)	81.70%(Epoch 19)

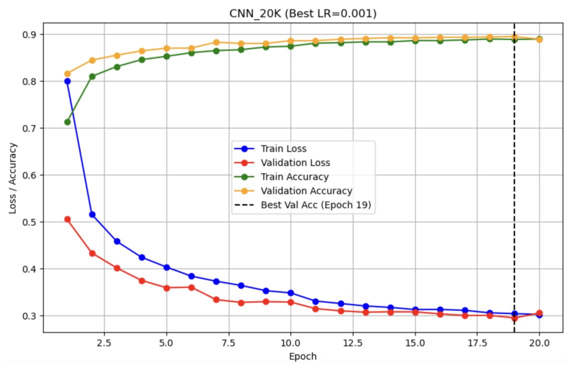
Table 3. Learning Rates Comparison for Different CNN Variants

Parameter	CNN_100K	CNN_50K	CNN_20K	CNN_10K
Convolution Channels	[16, 32]	[8, 16]	[6, 12]	[4, 8]
Learning Rate	0.001	0.001	0.001	0.001
Epochs	20	18	19	20
FC Units	120	120	60	45
Total Parameters (k)	102.13	50.58	19.39	9.84
Best Validation Accuracy (%)	91.62	90.68	89.43	88.27
Test Accuracy (%)	91.04	90.23	88.62	87.36

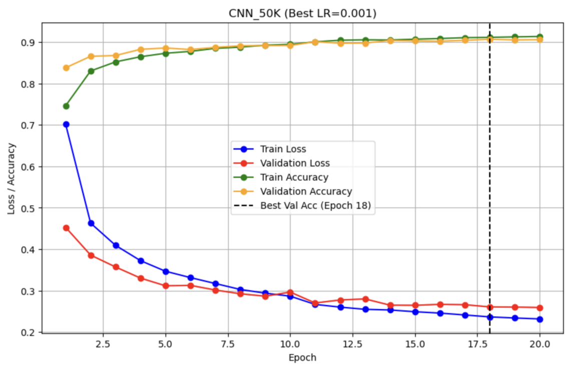
Table 4. Optimal Configurations and Performance Metrics for the CNN Variants



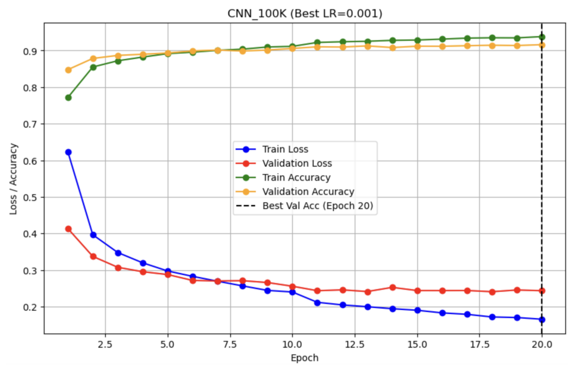
(a) CNN_10K (Best LR = 0.001)



(b) CNN_20K (Best LR = 0.001)



(c) CNN_50K (Best LR = 0.001)



(d) CNN_100K (Best LR = 0.001)

Figure 2. Training and Validation Loss and Accuracy Curves for CNN Variants

4.3. FCN and CNN Models Comparison. Finally, we compare the **FCN** and **CNN** variants side by side in terms of validation/test accuracies, number of parameters, and estimated training times. Table 5 consolidates the best-performing configuration for each model. Overall, we observe that **CNN_100K** achieves the highest test accuracy (91.04%), surpassing even the largest fully-connected model **FCN_200K** (89.59%). The CNN variants generally benefit from localized feature extraction in convolution layers, leading to higher accuracy with fewer parameters than an equally large FCN. However, **CNN_100K** also incurs the longest training time (about 305.7s), reflecting the computational overhead of deeper convolutional architectures.

On the other hand, smaller models such as **FCN_50K** and **CNN_10K** train faster (150.4s and 178.8s, respectively) but exhibit lower accuracy (87–88%), highlighting the trade-off between capacity and performance. Among the moderate-sized configurations, **CNN_50K** stands out with

90.23% test accuracy in just 50.58k parameters, providing an excellent balance of speed, model size, and generalization. Hence, from an efficiency standpoint, smaller networks require less computation time and memory, though at the cost of reduced accuracy, while larger models (especially CNN_100K) achieve superior results but demand more training time and computational resources.

From Table 5, we conclude that **CNN_100K** is the most accurate model overall, while **FCN_200K** is the largest fully-connected variant. The **CNN** architectures leverage fewer parameters to achieve competitive or superior accuracies relative to the FCNs, though they typically require more complex operations (convolutions). Ultimately, the choice depends on one's resource constraints and accuracy goals: larger models deliver higher accuracy but at increased computational cost, whereas smaller networks train faster and consume fewer resources at the expense of lower accuracy.

Model	**Weights (k)**	**Best Val Acc (%)**	**Test Acc (%)**	**Train Time (s)**
FCN_50K	52.65	88.67	87.94	150.4
FCN_100K	80.51	89.33	88.25	180.2
FCN_200K	202.17	90.15	89.59	220.6
CNN_10K	9.84	88.27	87.36	178.8
CNN_20K	19.39	89.43	88.62	222.2
CNN_50K	50.58	90.68	90.23	184.6
CNN_100K	102.13	91.62	91.04	305.7

Table 5. Comparison of FCN and CNN Variants

5. Summary and Conclusions

In this report, we compared various **FCN** and **CNN** configurations for FashionMNIST classification, emphasizing how architecture and parameter budgets influence performance. Each model was trained using an identical procedure but with distinct hyperparameter settings, including different hidden dimensions or convolutional channels. Our findings underscore the effectiveness of CNNs in exploiting spatial information, with the best **CNN** variant (`CNN_100K`) reaching a **91.04%** test accuracy—surpassing the best **FCN** variant (`FCN_200K`), which achieved **89.59%**. Furthermore, our experiments demonstrate that a higher weight count consistently yields higher accuracy, although it comes with increased computational overhead. Nevertheless, smaller FCNs and CNNs remain attractive when memory and training speed are prioritized. Overall, the choice of network depends on resource constraints and accuracy goals, with high weights CNNs typically offering superior efficiency and representational power.

Acknowledgements

This author thanks Professor Frank for providing useful information to tackle the problem through the hints in the assignment, as well as the HW4-Helper.ipynb code. The author is also grateful to TA Rohin Gilman for providing theoretical background to assist in constructing my code. Finally, my peers Liam Lin and Lanmin Lin were also helpful when I ran into issues related to coding.

References

[1] J.N. Kutz, *Methods for Integrating Dynamics of Complex Systems and Big Data*, Oxford, 2013.
[2] N. Frank, *Lecture Notes*, March 2025.