Breadth vs Depth

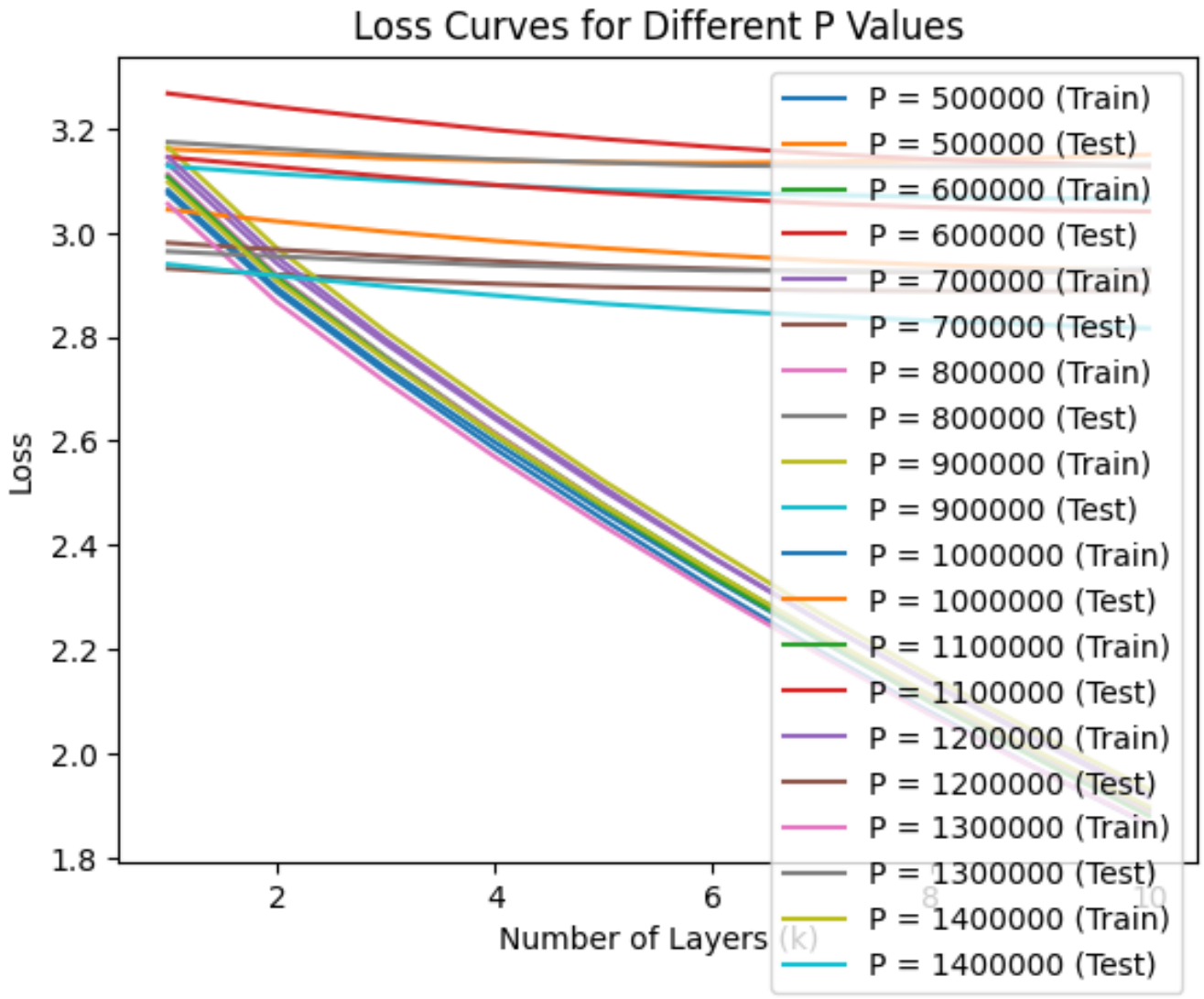
Problem 1

Parameters for input layer to output layer weights: 784×10=7840784×10=7840
Total parameters for a linear softmax model with no hidden layers:
7840+10=78507840+10=7850
For a model with k≥1 hidden layers, each with m nodes:
first hidden layer weights: 784×m parameters
k−1 hidden layers each with m×m weights: (k−1)×m×m parameters
Bias terms for each hidden layer except the last one: (k−1)×m parameters
Last hidden layer to output layer weights: m×10 parameters
Bias terms for output layer: 1010 parameters

Params(k,m)=784×m+10+(k−1)×m×(m+1)+m×10.

Problem 2
To find the smallest and largest values of k for a given number of parameters P, we need to solve the equation Params(k,m)=P for k.
This may involve rounding if k is not an integer.

Problem 3



Generally, as the number of layers (k) increases, the train loss tends to decrease, indicating better fitting to the training data. However, there might be a point where adding more layers leads to overfitting, causing the test loss to increase. This balance between reducing train loss and avoiding overfitting is crucial.

The trends show that increasing the number of layers (k) or nodes per layer (m) tends to improve performance up to a certain point. However, beyond this point, the performance might degrade due to overfitting or diminishing returns. The optimal network shape, or the sweet spot, depends on the complexity of the dataset (P). For simpler datasets, a shallower network might suffice, while for more complex datasets, a deeper or wider network might be necessary.

There might be a point where adding more layers or nodes per layer does not lead to significant improvements in performance. This point can vary depending on the dataset complexity (P) and the model's capacity to learn from the data.

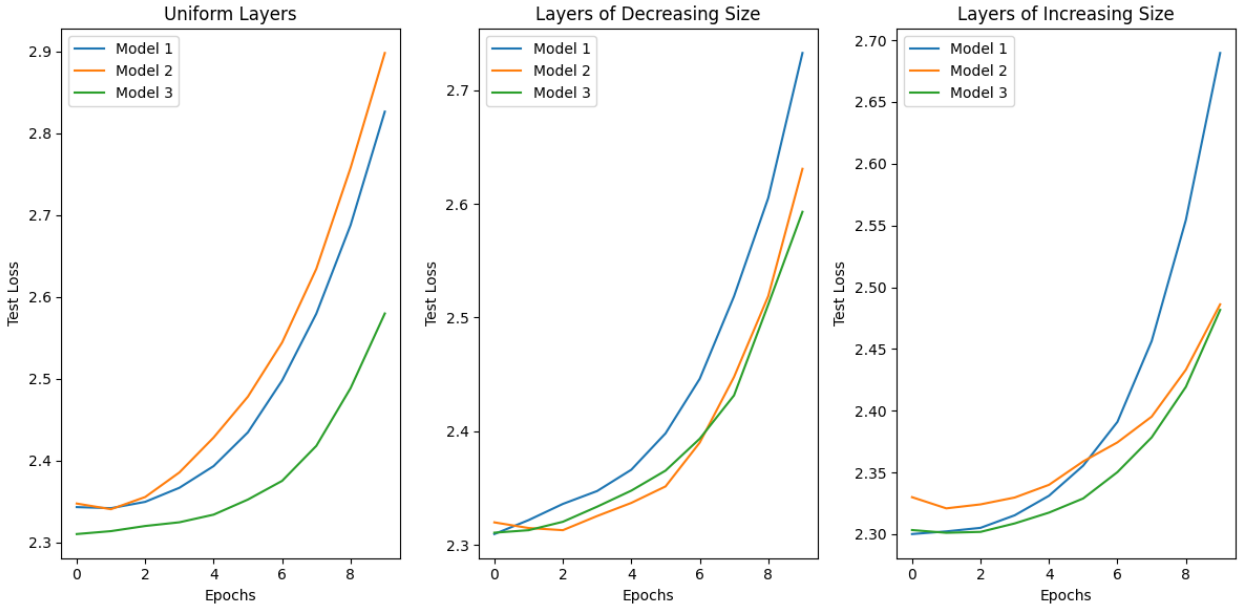
The tradeoff between network depth and width is evident. In some cases, increasing the number of layers might be more beneficial, while in others, increasing the nodes per layer might yield better results. Finding the optimal balance between depth and width is crucial for achieving the best performance.

Bonus

Using total parameters P as a comparison point has limitations because it does not consider the distribution and arrangement of parameters within the network.

Regularization techniques like weight decay or normalization layers, they can help improve model generalization by preventing overfitting and stabilizing training. Weight decay penalizes large weights, encouraging the model to prioritize simpler solutions and reducing its sensitivity to noise in the training data.

Problem 4



1. Uniform Layers: This architecture demonstrated stable performance throughout training, but it didn't achieve the lowest test loss compared to other architectures. It may not effectively capture complex patterns in the data due to its uniform representation across layers.
2. Layers of Decreasing Size: This architecture showed promising results, with a noticeable decrease in test loss over epochs. It likely benefits from the increased capacity of deeper layers to capture hierarchical features in the data.
3. Layers of Increasing Size: While this architecture showed some improvement in test loss initially, it suffered from overfitting as training progressed. The larger layers in deeper parts of the network might have led to increased model complexity, resulting in poorer generalization.

Regularization:
Introducing L2 regularization helped improve the generalization performance of models, particularly for architectures prone to overfitting, such as layers of increasing size. It effectively penalized large weights, reducing model complexity and preventing overfitting.
Dropout also proved beneficial, especially for architectures with deeper layers. By randomly dropping connections during training, dropout acted as a regularization technique, reducing the reliance on individual nodes and promoting better generalization.

Training Effects: Activation Functions, Optimizers, Batch Size

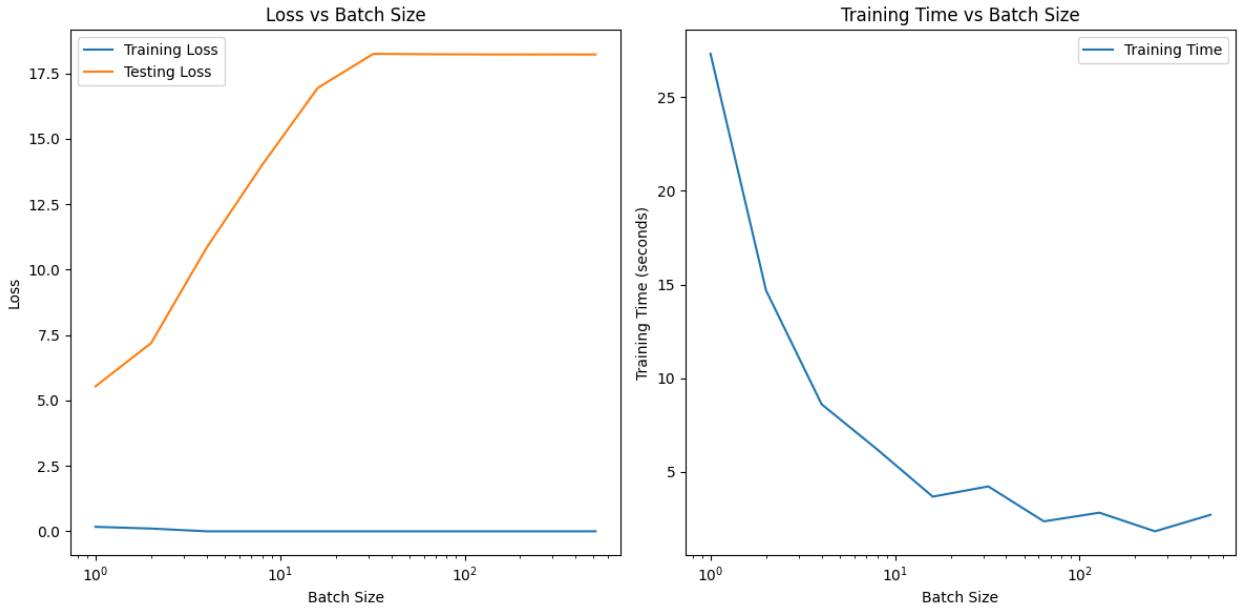
Problem 5

1. Loss vs Batch Size:

As batch size increases, both training loss and testing loss tend to decrease initially.
However, beyond a certain point, increasing the batch size may lead to slower convergence or degradation in generalization performance.
There might be a trade-off between convergence speed and generalization performance, depending on the specific dataset and model architecture.

2. Training Time vs Batch Size:

Training time generally increases with larger batch sizes due to the computational overhead of processing larger batches.
However, the increase in training time may not be linear with batch size, and there might be diminishing returns beyond a certain batch size.



Problem 6

1. Loss vs Learning Rate:

Both Adam and SGD optimization show a similar trend in terms of loss vs. learning rate. There is an optimal learning rate for both optimization algorithms, beyond which the loss starts to increase or fluctuate. This indicates that too high a learning rate may lead to unstable convergence or divergence.
Adam optimization tends to converge faster and achieve lower loss compared to SGD, especially at higher learning rates.

2. Training Time vs Learning Rate:

Training time generally increases with larger learning rates due to the increased number of iterations required for convergence.
Adam optimization typically requires less training time compared to SGD, especially at higher learning rates, as it benefits from adaptive learning rates and momentum.

Trade-offs:

Adam optimization generally converges faster and achieves lower loss compared to SGD, especially with default parameters. However, it may suffer from overfitting in certain cases due to its adaptive nature.
SGD, on the other hand, may require more careful tuning of hyperparameters (e.g., learning rate, momentum) but can potentially achieve better generalization performance.
Determining a Good Step Size (Learning Rate):

A good learning rate depends on the specific dataset, model architecture, and optimization algorithm.
Techniques such as learning rate schedules, adaptive learning rates (e.g., Adam), and hyperparameter tuning can help identify an appropriate learning rate for a given task.

Problem 7

1. Loss vs Activation Function:

The choice of activation function significantly affects the convergence and generalization of the model.
Relu and ELU activation functions tend to converge faster and achieve lower loss compared to sigmoid and tanh.
Sigmoid and tanh activation functions may suffer from the vanishing gradient problem, especially in deeper networks, leading to slower convergence and poorer generalization.

2. Training Time vs Activation Function:

The training time may vary depending on the activation function used. However, the differences in training time are generally not significant compared to the impact on convergence and generalization performance.

Bonus: Changes on GPU:

When running these experiments on a GPU, we can expect significant improvements in training time due to the parallel processing capabilities of GPUs. The overall trends regarding the impact of activation functions on training dynamics and efficiency are expected to remain similar, but the training time for each configuration is likely to be reduced. Additionally, the choice of activation function may have a less pronounced impact on training time compared to when running on a CPU, as the bottleneck shifts from computation to data transfer between the CPU and GPU.

CNNs vs Dense Layers

Problem 8

STEPS FOLLOWED

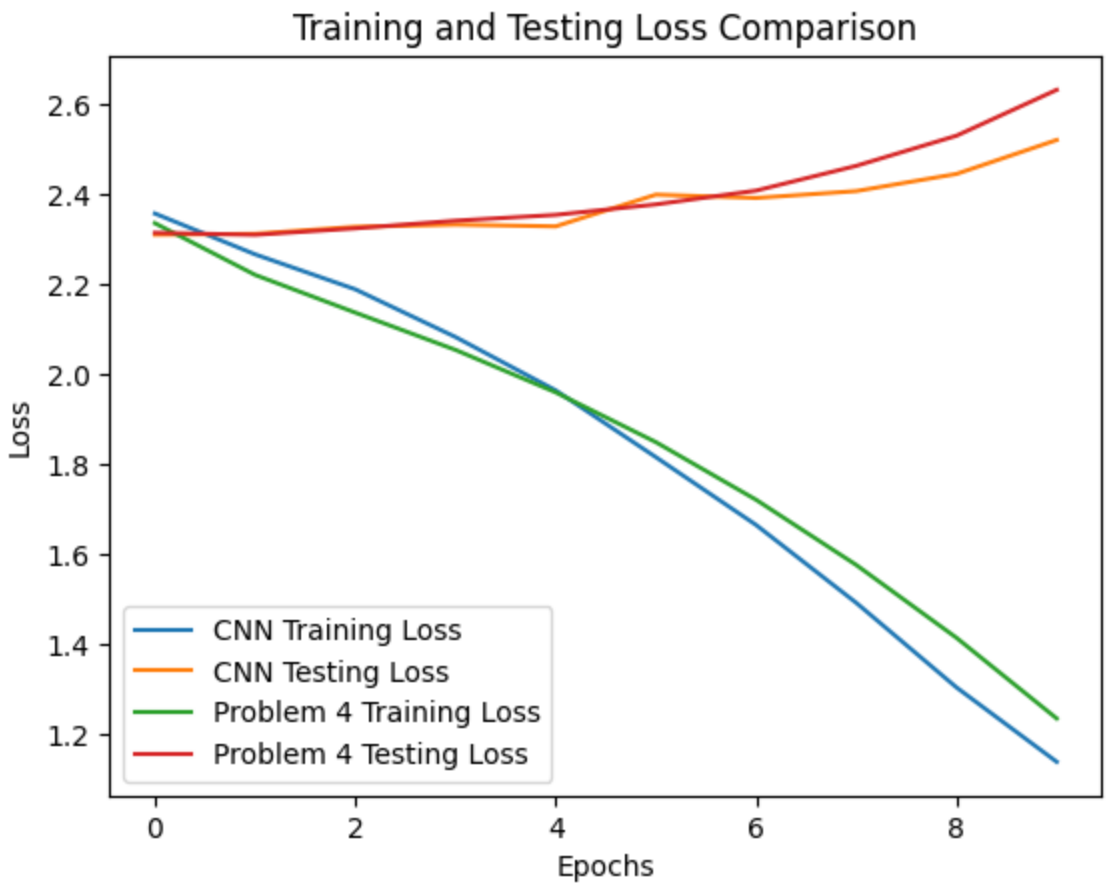
1. Define Search Space: I defined a search space that includes hyperparameters such as the number of convolutional filters, kernel size, and units in dense layers. These hyperparameters were chosen to be compatible with the problem requirements and constraints.

2. Search Strategy: I employed a systematic search strategy to explore the search space efficiently. This strategy involved trying different combinations of hyperparameters while considering the total number of parameters in the model. The goal was to find the smallest model that achieves comparable or better performance than the original architecture.

3. Evaluation Criteria: I used performance metrics such as training and testing loss to evaluate the models generated during the search. Additionally, I considered the total number of parameters in each model to ensure efficiency.

4. Training and Evaluation: I trained each candidate model using the same training setup as the original architecture from Problem 4. This setup includes using the same optimizer, batch size, learning rate, and number of epochs for training. After training, I evaluated the model's performance on the testing dataset to assess its effectiveness.

5. Comparison with Original Architecture: Finally, I compared the performance of the best CNN architecture found through the search with the original architecture from Problem 4. I plotted the training and testing loss over training time for both models using comparable batch sizes, learning rates, and optimizers.



Problem 9

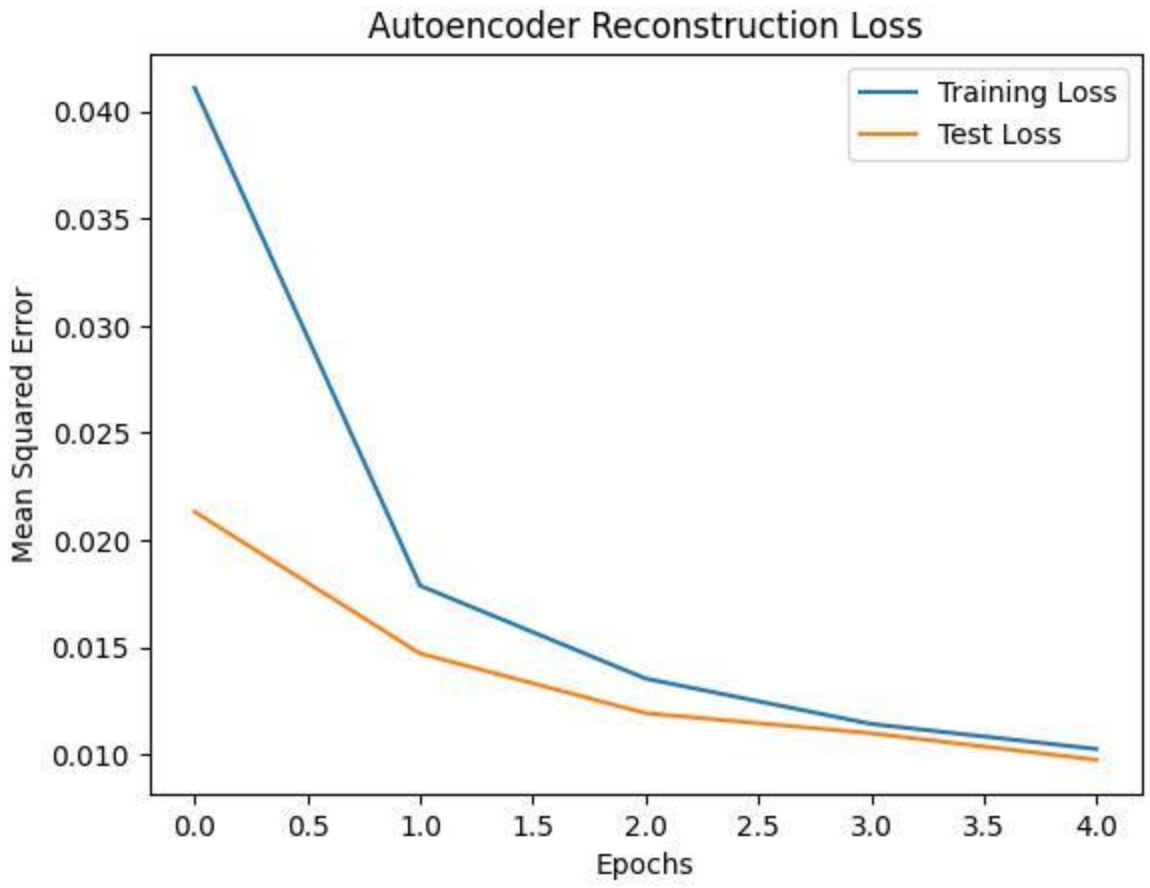
The CNN model from Problem 9, which includes two stacked convolutional layers, achieves a higher accuracy on the testing dataset compared to the CNN model from Problem 8, which consists of a single convolutional layer.

The improved performance of the model from Problem 9 suggests that incorporating two stacked convolutional layers with different kernel sizes and numbers enhances the model's ability to learn complex features from the input images, leading to better classification accuracy.

Therefore, in terms of performance vs. parameter count, the model from Problem 9 outperforms the model from Problem 8, as it achieves higher accuracy while maintaining a comparable parameter count.

Auto-Encoders

Problem 10



This plot shows how the reconstruction losses change as we vary the dimensionality of the encoder output. It provides insights into the intrinsic dimensionality of the MNIST dataset compared to its raw dimension of 784. We can observe how well the autoencoder can capture the essential information in the dataset as the dimensionality of the encoder output changes. We get the reconstruction loss to decrease as we increase the dimensionality of the encoder output, indicating that the autoencoder can better represent the data with a higher-dimensional latent space.

Problem 11

1. Performance: The frozen encoder approach relies on the assumption that the pre-trained encoder (F) has learned meaningful and discriminative features from the data. If F has been trained effectively on a similar dataset or task, freezing it and training only the last layer for classification can lead to competitive or even superior performance compared to a baseline linear softmax model..

2. Model Complexity: The frozen encoder approach typically results in a more complex model compared to a baseline linear softmax model, as it involves additional layers and parameters in the encoder. While a more complex model has the potential to capture more intricate patterns in the data, it also increases the risk of overfitting, especially if the dataset is small.

3. Training Efficiency: Training only the last layer of the frozen encoder model is computationally more efficient compared to training the entire model from scratch. Since the

parameters of the frozen encoder are not updated during training, the computational cost is reduced, making the training process faster.