Experiment Report

Written by Ruoxuan Li DC027534 in cooperation with Pengze Guo DC12746

Introduction

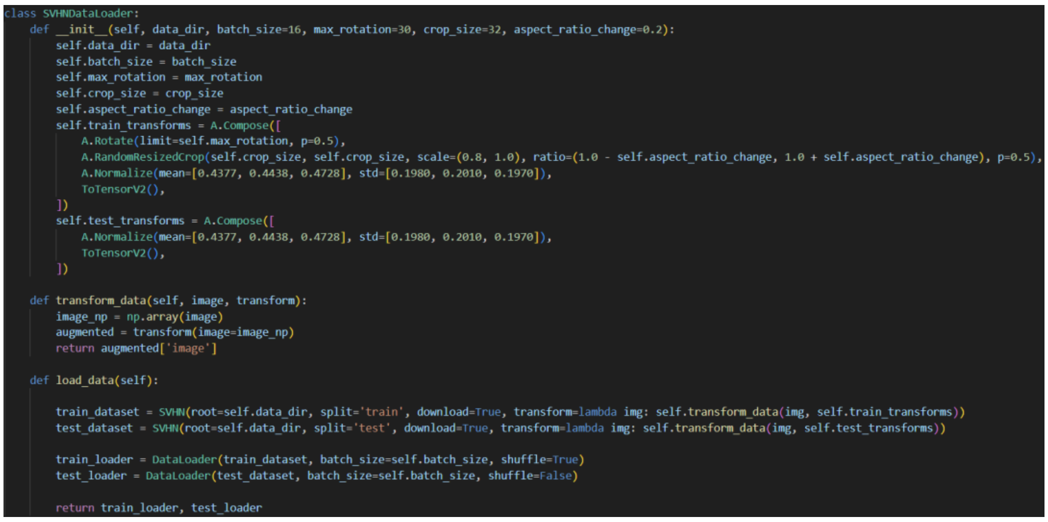
Through this pattern recognition project, I comprehend how to prepare data, implemented models, train, evaluate and analysis. I will introduce this experiment from three perspectives: Methodology, Results, and Conclusion. The code for this experiment is adapted from Lecture Code 5, with a primary focus on observing the impact of learning rate, batch size, number of epochs, and optimizer types on model accuracy. For this model and dataset, the optimal hyperparameter settings are batch size = 16, learning rate = 1E-4, and epochs = 100. The optimizer Adam demonstrates significantly better performance (in terms of accuracy and convergence speed) compared to SGD.

Methodology

I will explain the data processing steps, model architecture, training procedures, and evaluation methods in conjunction with the code.

Data Processing and Augmentation

This is the SVHN data loader. Apart from Normalization, it's easy to notice there are other methods available. However, for this specific implementation, we primarily focus on Normalization. What's more, augmentation parameters are adjustable.



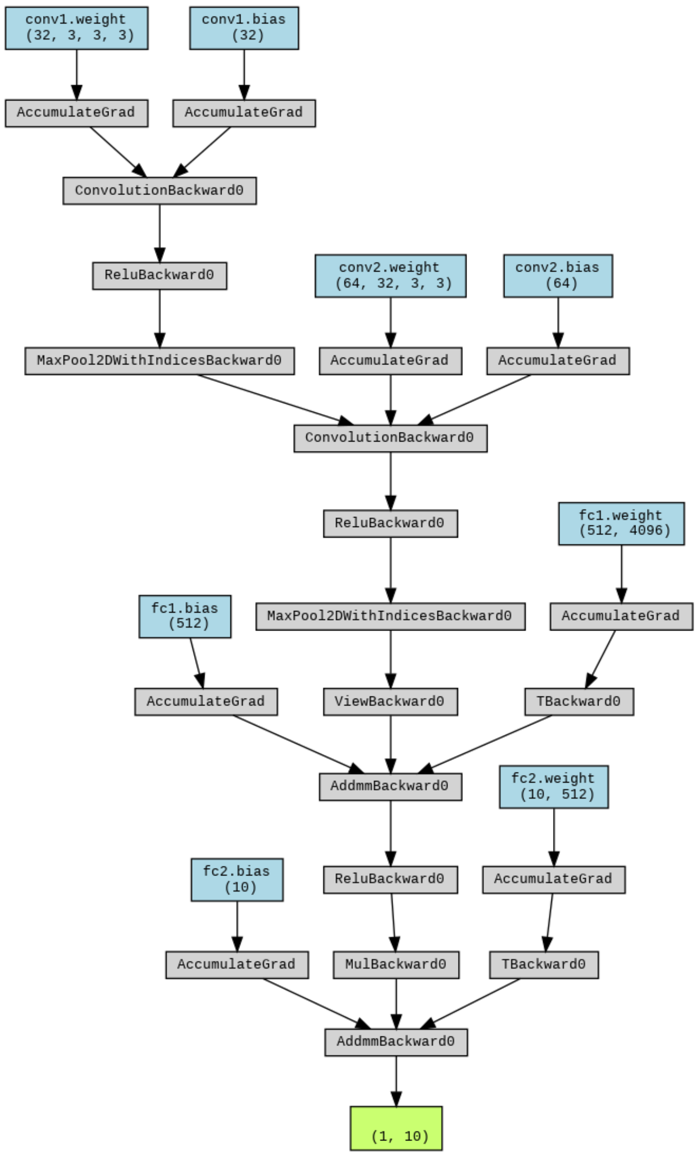
```python
class SVHNDataLoader:
    def __init__(self, data_dir, batch_size=16, max_rotation=30, crop_size=32, aspect_ratio_change=0.2):
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.max_rotation = max_rotation
        self.crop_size = crop_size
        self.aspect_ratio_change = aspect_ratio_change
        self.train_transforms = A.Compose([
            A.Rotate(limit=self.max_rotation, p=0.5),
            A.RandomResizedCrop(self.crop_size, self.crop_size, scale=(0.8, 1.0), ratio=(1.0 - self.aspect_ratio_change, 1.0 + self.aspect_ratio_change), p=0.5),
            A.Normalize(mean=[0.4377, 0.4438, 0.4728], std=[0.1980, 0.2010, 0.1970]),
            ToTensorV2(),
        ])
        self.test_transforms = A.Compose([
            A.Normalize(mean=[0.4377, 0.4438, 0.4728], std=[0.1980, 0.2010, 0.1970]),
            ToTensorV2(),
        ])

    def transform_data(self, image, transform):
        image_np = np.array(image)
        augmented = transform(image=image_np)
        return augmented['image']

    def load_data(self):

        train_dataset = SVHN(root=self.data_dir, split='train', download=True, transform=lambda img: self.transform_data(img, self.train_transforms))
        test_dataset = SVHN(root=self.data_dir, split='test', download=True, transform=lambda img: self.transform_data(img, self.test_transforms))

        train_loader = DataLoader(train_dataset, batch_size=self.batch_size, shuffle=True)
        test_loader = DataLoader(test_dataset, batch_size=self.batch_size, shuffle=False)

        return train_loader, test_loader
```

We can use the SVHN data loader in main.py with the following code.

```
data_loader = SVHNDataLoader(data_dir='./data', batch_size=16)
train_loader, test_loader = data_loader.load_data()
```

Neural Network Setup

This SimpleCNN model is a modified version of the architecture provided in Lecture Code 5. Notably, a dropout layer was added to prevent overfitting. Additionally, we implemented an early stopping mechanism in subsequent steps to further reduce overfitting. This is a graph of neural network.



Next, I will provide a detailed introduction:

Convolutional Layers:

conv1: A 2D convolutional layer with 3 input channels (for RGB images) and 32 output channels, a kernel size of 3, stride of 1, and padding of 1.

conv2: Another 2D convolutional layer that takes 32 input channels and outputs 64 channels, with the same kernel size, stride, and padding.

Pooling Layer:

pool: A max-pooling layer with a 2x2 kernel and a stride of 2, which down samples the input by a factor of 2.

Fully Connected Layers:

fc1: A fully connected (dense) layer with input size 64 * 8 * 8 (from flattening the output of the convolutional layers) and output size 512.

fc2: Another fully connected layer that maps from 512 features to 10 output classes (assuming 10 classes in the dataset, such as for SVHN or CIFAR-10).

Dropout Layer:

dropout: A dropout layer with a dropout probability of 0.5, applied to prevent overfitting.

Forward Pass

In the forward method, the input x goes through the following stages:

Convolution + ReLU + Pooling:

Applies conv1, followed by ReLU activation and max-pooling.

Applies conv2, followed by ReLU activation and max-pooling.

Flattening:

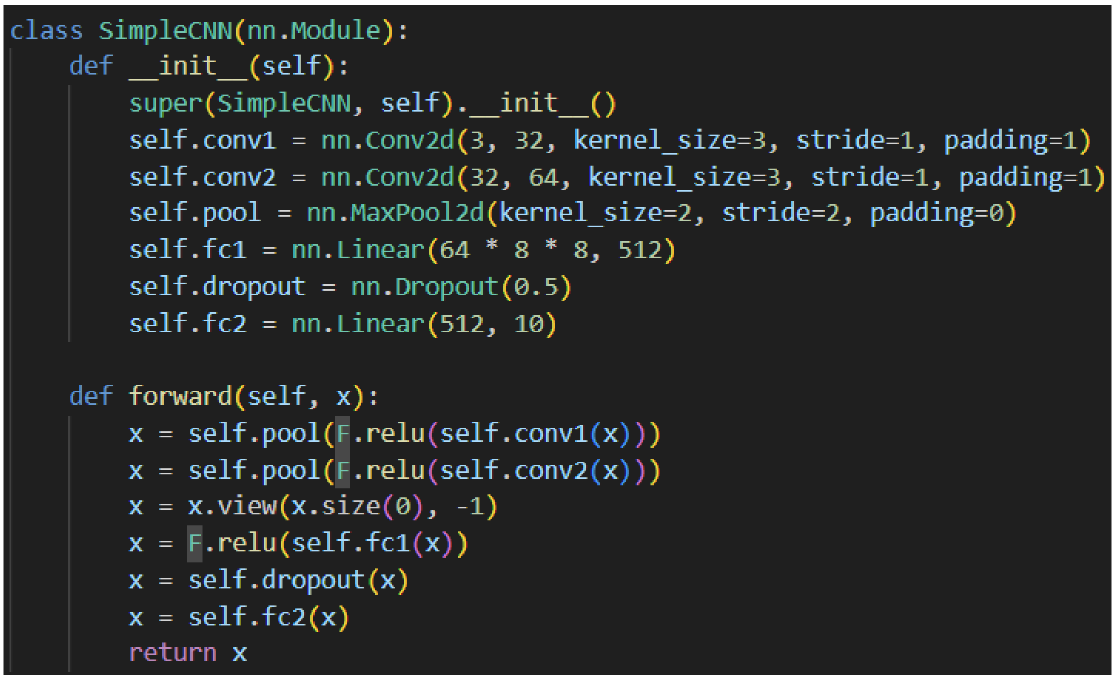
The output of the convolutional layers is reshaped (flattened) to a 1D vector for each batch item.

Fully Connected Layers:

Passes through fc1 with ReLU activation.

Applies dropout to the activations from fc1 to help prevent overfitting.

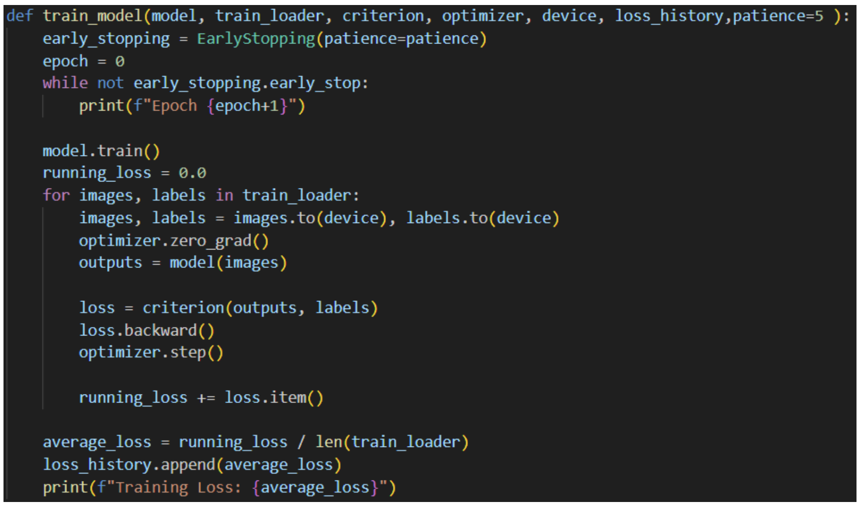
Finally, passes through fc2 to output the final class scores.



```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(64 * 8 * 8, 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Coding Training and Evaluation Functions

In train.py, develop functions to handle the training process of the neural network and assess its performance on the test dataset.

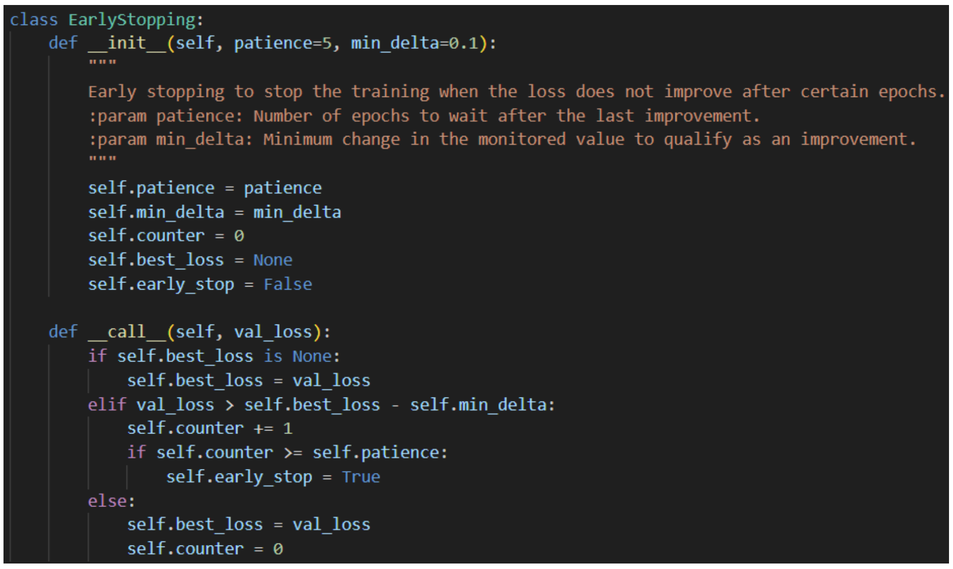


```python
def train_model(model, train_loader, criterion, optimizer, device, loss_history,patience=5 ):
    early_stopping = EarlyStopping(patience=patience)
    epoch = 0
    while not early_stopping.early_stop:
        print(f"Epoch {epoch+1}")

    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    average_loss = running_loss / len(train_loader)
    loss_history.append(average_loss)
    print(f"Training Loss: {average_loss}")
```

In this section, I implemented the following steps:

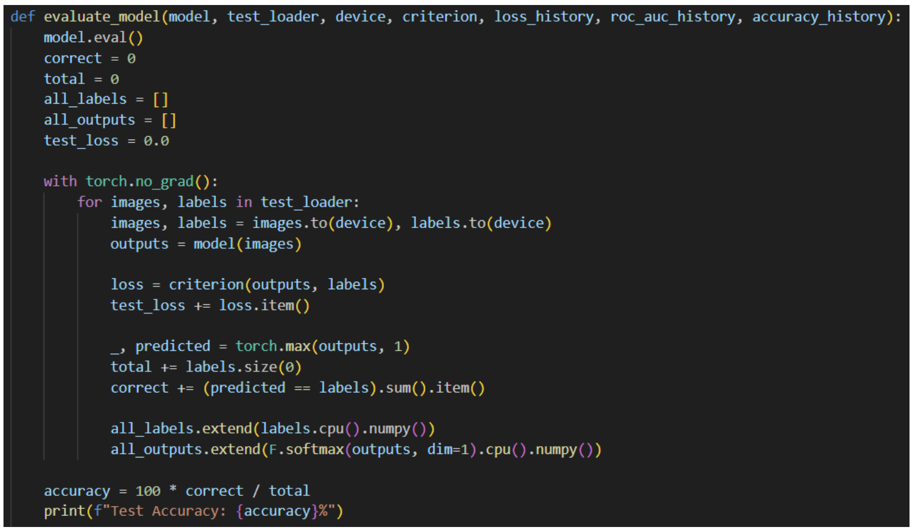
1. Processed input batches (an alternative is to handle this in the data loader).
2. Executed forward and backward passes.
3. Updated models use optimizers (SGD, Adam).
4. Tracked and displayed training loss.

Furthermore, I added early stopping; the implementation details are outlined below.

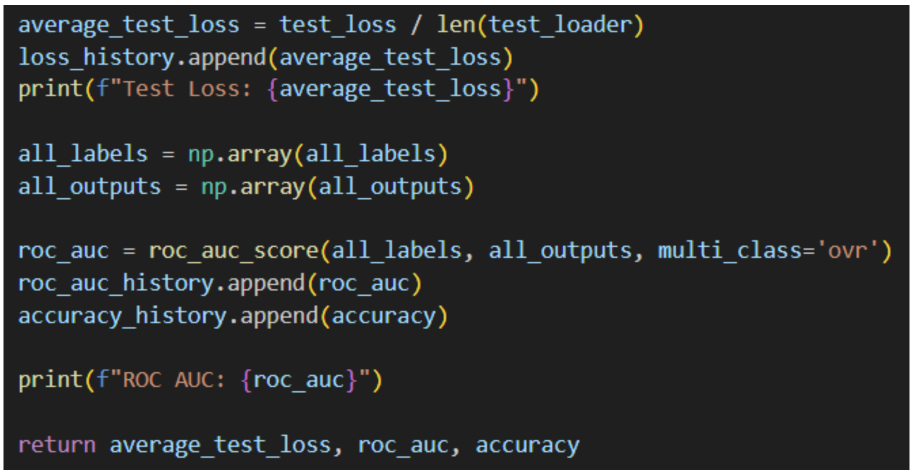


```python
class EarlyStopping:
    def __init__(self, patience=5, min_delta=0.1):
        """
        Early stopping to stop the training when the loss does not improve after certain epochs.
        :param patience: Number of epochs to wait after the last improvement.
        :param min_delta: Minimum change in the monitored value to qualify as an improvement.
        """
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0
```

After that I implement an evaluation function that:



```python
def evaluate_model(model, test_loader, device, criterion, loss_history, roc_auc_history, accuracy_history):
    model.eval()
    correct = 0
    total = 0
    all_labels = []
    all_outputs = []
    test_loss = 0.0

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)

            loss = criterion(outputs, labels)
            test_loss += loss.item()

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_labels.extend(labels.cpu().numpy())
            all_outputs.extend(F.softmax(outputs, dim=1).cpu().numpy())

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy}%")
```



```python
average_test_loss = test_loss / len(test_loader)
loss_history.append(average_test_loss)
print(f"Test Loss: {average_test_loss}")

all_labels = np.array(all_labels)
all_outputs = np.array(all_outputs)

roc_auc = roc_auc_score(all_labels, all_outputs, multi_class='ovr')
roc_auc_history.append(roc_auc)
accuracy_history.append(accuracy)

print(f"ROC AUC: {roc_auc}")

return average_test_loss, roc_auc, accuracy
```

In this section, I implemented the following steps:

1. Set the model to evaluation mode.
2. Calculated accuracy on the test dataset.
3. Computed Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC) metrics.
4. Included calculations for both macro and micro ROC AUC values.

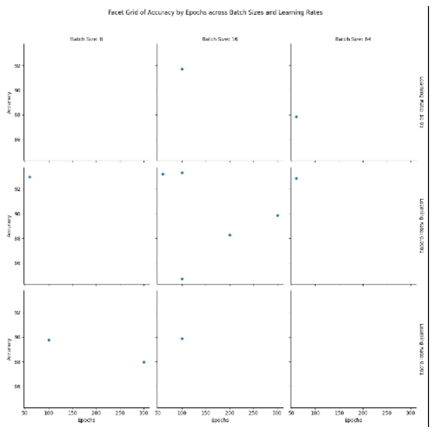
Results

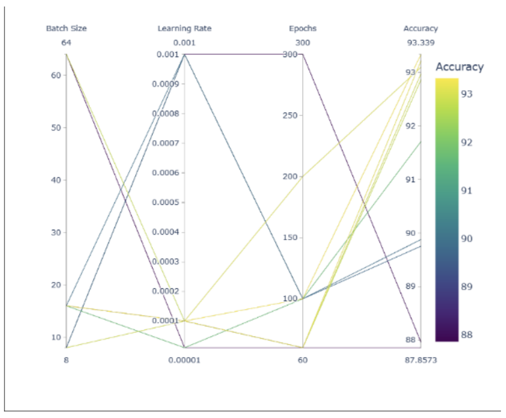
In this section, I will analyze how various hyperparameters and network designs impact the model's behavior and performance, providing quantitative results and visualizations. Following this, I will examine the results in detail, discuss the effects of different hyperparameters and network structures, and reflect on the model's overall performance.

quantitative results along with visualizations

○ Hyperparameter Tuning:

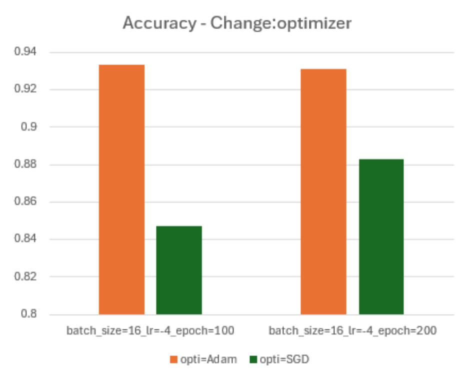
The independent variables in my study are learning rate, batch size, number of epochs, and optimizer types. For each of these, I designed two sets of experiments (i.e., when the independent variable under investigation is batch size, I used two different combinations of learning rates and epochs to obtain more rigorous results). I observed a very clear difference between optimizers: SGD failed to converge even after 500 epochs, while Adam converged within a few dozen epochs. Therefore, all experiments in the studies on learning rate, batch size, and number of epochs were conducted using the Adam optimizer. The overall distribution of the data is as follows.

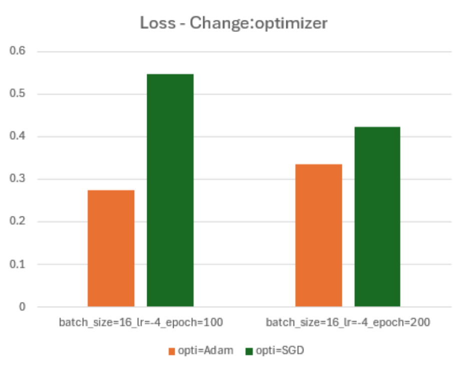


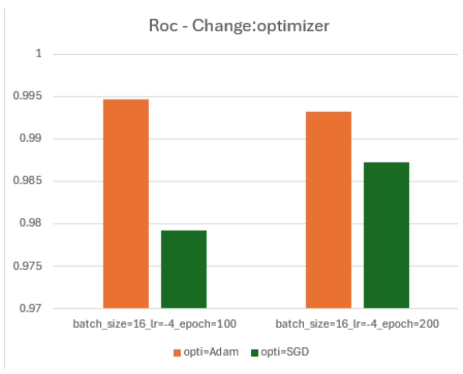


Control variable method

change: optimizer								
	batch_size = 16	lr = -4	epoch = 100			batch_size = 16	lr = -4	epoch = 200
Adam	93.33896	0.2751	0.99468		Adam	93.09695759	0.335903001	0.993240571
SGD	84.7111247	0.5481241	0.97919		SGD	88.2721266	0.42390118	0.987283526





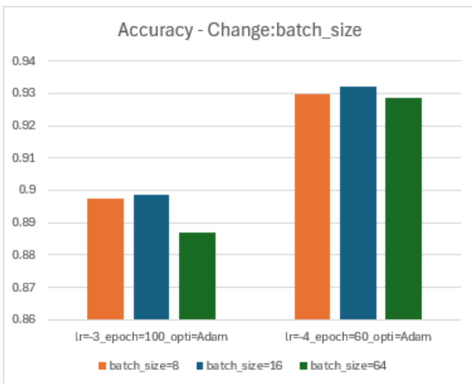


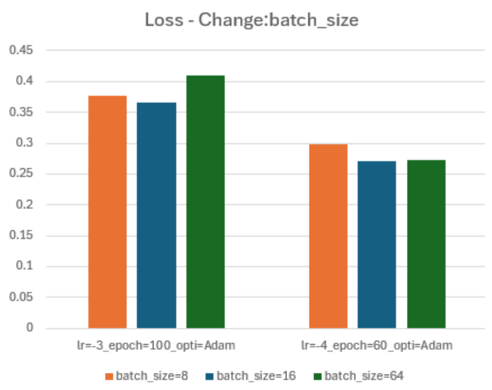
Optimizer Comparison:

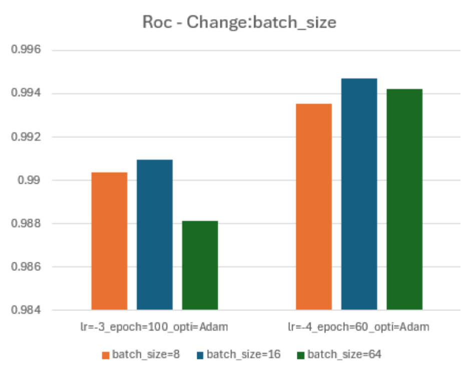
Findings: The Adam optimizer significantly outperformed SGD in terms of convergence speed and final accuracy. For example, with batch_size = 16, lr = 0.0001, and 100 epochs, Adam achieved 93.34% accuracy, whereas SGD only managed 84.71% accuracy, even after extensive training.

Explanation: Adam's adaptive learning rate mechanism allows it to handle noisy and sparse gradients more effectively, making it better suited for deep learning tasks, particularly when the dataset or gradients are complex. SGD often requires more fine-tuning of hyperparameters and more epochs to reach comparable performance.

change: batch__size									
	lr = – 3	epoch = 100	optimizer = Adam			lr = – 4	epoch = 60	optimizer = Adam	
batch_size	accuracy	loss	roc		batch_size	accuracy	loss	roc	
8	89.7549	0.375905128	0.9904		8	92.9778	0.29736	0.993565	
16	89.877842	0.36621945	0.99099		16	93.21988	0.27175	0.9947	
64	88.7062077	0.409011044	0.98814		64	92.86647	0.2735594	0.994241	





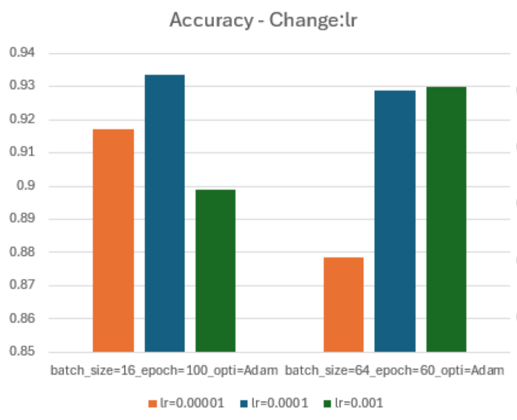


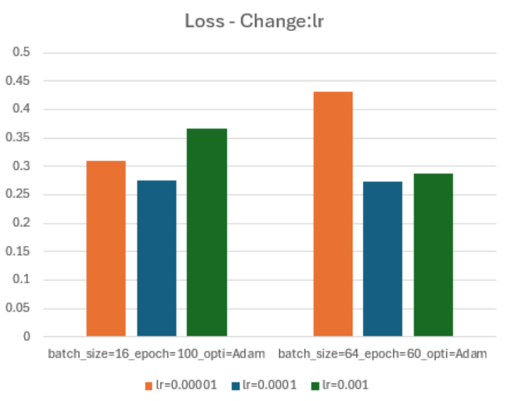
Batch Size Effects:

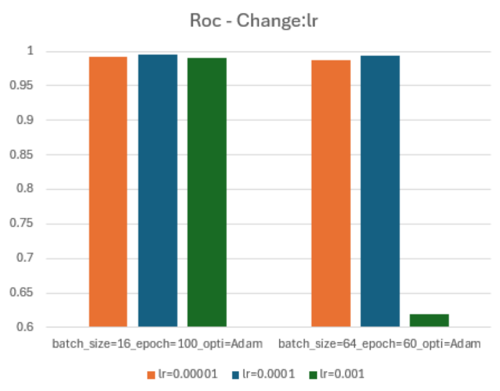
Findings: Smaller batch sizes (8 and 16) generally resulted in higher accuracy and better ROC scores compared to larger batch sizes (64). For instance, with lr = 0.0001 and 60 epochs, a batch size of 16 achieved 93.22% accuracy, while a batch size of 64 achieved slightly lower accuracy of 92.87%.

Explanation: Smaller batch sizes can lead to more frequent updates to model weights, often resulting in smoother convergence and better generalization. However, they may also introduce noise in the gradient estimates, which sometimes helps escape local minima. Larger batch sizes, while more stable, may require more epochs to converge.

c h a n g e :　l r									
	batch_size = 16	epoch = 100	optimizer = Adam			batch_size = 64	epoch = 60	optimizer = Adam	
lr	accuracy	loss	roc		lr	accuracy	loss	roc	
0.00001	91.710202	0.3105115	0.9926		0.00001	87.85725	0.431059	0.98684	
0.0001	93.33896	0.2751	0.99468		0.0001	92.86647	0.2735594	0.994241	
0.001	89.877842	0.36621945	0.99099		0.001	92.9778733	0.288312866	0.618626	





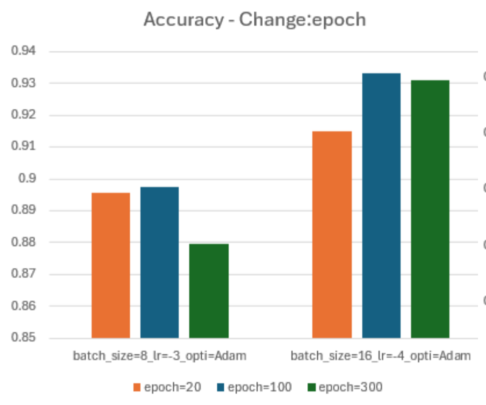


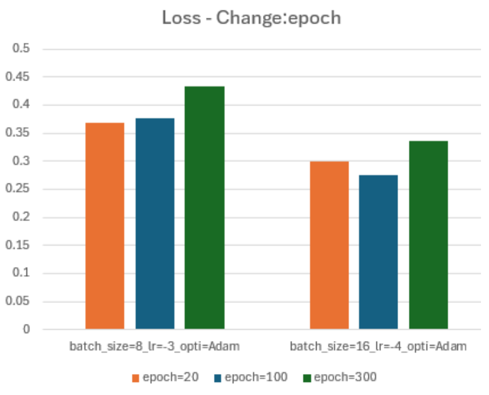
Learning Rate Impact:

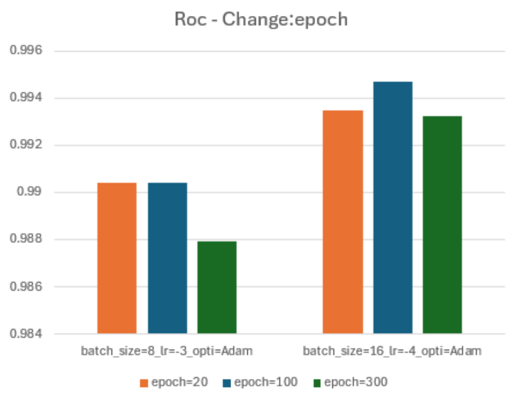
Findings: A learning rate of 0.0001 consistently provided the best balance between convergence speed and model performance, achieving the highest accuracy and lowest loss. Extremely small learning rates (0.00001) led to slow convergence, while larger learning rates (0.001) sometimes resulted in higher loss and worse ROC scores due to unstable updates.

Explanation: A learning rate that is too high can cause the model to diverge or oscillate around the optimal point, while a learning rate that is too low slows down convergence, requiring more epochs and time. 0.0001 provided a sweet spot for convergence in this case.

c h a n g e : e p o c h									
	batch_size = 8	lr = -3	optimizer = Adam			batch_size = 16	lr = -4	optimizer = Adam	
epoch	accuracy	loss	roc		epoch	accuracy	loss	roc	
20	89.5513214	0.369617	0.99043		20	91.483558	0.30037732	0.99347202	
100	89.7549	0.375905128	0.9904		100	93.33896	0.2751	0.99467973	
300	87.957129	0.433008	0.98796		200	93.09695759	0.335903001	0.993240571	







Epoch Count Influence:

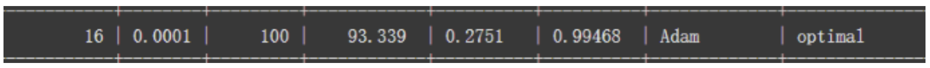
Findings: Increasing the number of epochs generally improved the model's accuracy until a certain point, after which performance gains diminished. For example, with batch_size = 16 and lr = 0.0001, accuracy improved from 91.48% at 20 epochs to 93.34% at 100 epochs but only slightly increased or even stagnated at 200 epochs.

Explanation: More training epochs allow the model to learn better but also increase the risk of overfitting. Early stopping mechanisms can be helpful to identify the point where additional training no longer benefits performance.

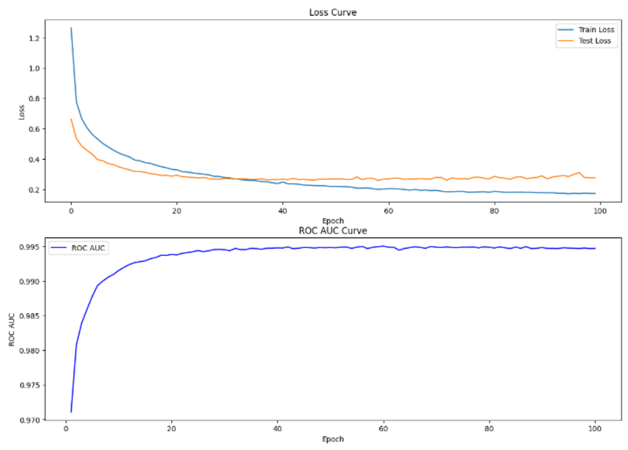
○ Evaluation Metrics & Visualizations

Since I've conducted numerous experiments, I won't present all the results. Instead, I'll showcase the best model.

Optimal model



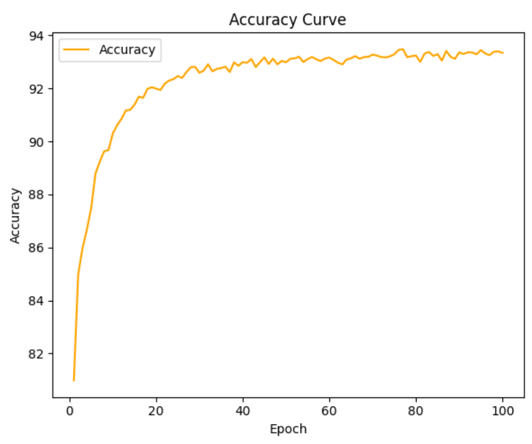
```
   16 | 0.0001 |    100 |   93.339 | 0.2751 | 0.99468 | Adam        | optimal
```



Loss 0.2751 & ROC 0.994

Good Convergence: Both training and test losses are low, and the high ROC score indicates strong classification performance.

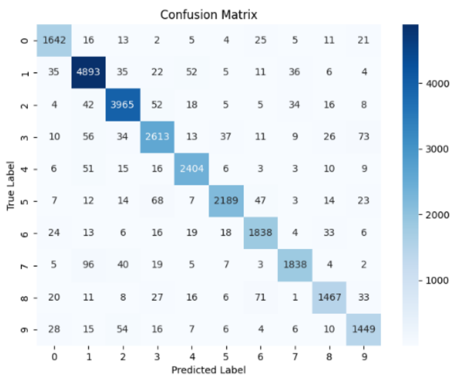
Slight Overfitting: gap between the training and test loss curves. Early Stopping & Regularization via dropout



Accuracy 93.3%

Rapid Initial Increase: The accuracy increases sharply during the initial epochs. This indicates that the model is quickly learning the patterns in the data.

Plateaus may means overfitting



	0	1	2	3	4	5	6	7	8	9
0	1642	16	13	2	5	4	25	5	11	21
1	35	4893	35	22	52	5	11	36	6	4
2	4	42	3965	52	18	5	5	34	16	8
3	10	56	34	2613	13	37	11	9	26	73
4	6	51	15	16	2404	6	3	3	10	9
5	7	12	14	68	7	2189	47	3	14	23
6	24	13	6	16	19	18	1838	4	33	6
7	5	96	40	19	5	7	3	1838	4	2
8	20	11	8	27	16	6	71	1	1467	33
9	28	15	54	16	7	6	4	6	10	1449

Confusion Matrix

High Overall Accuracy: The matrix shows that most predictions are along the diagonal, indicating high accuracy across most classes.

Potential for Improvement: the classification of digits "8" and "9"

Improvement: Model Fine-Tuning & Data Augmentation

In fact, this may not necessarily be the optimal model as it shows slight overfitting. However, I added dropout to counteract the overfitting, and the early stopping mechanism was not triggered, suggesting it is a relatively well-fitted result. I will also try pruning, if it is indeed overfitting, pruning should yield better results.

Discussion

I've already discussed the details of hyperparameters above; in this section, I will focus on the challenges and their impact.

Challenge1: Slow Convergence with SGD:

Problem: The model took significantly longer to converge when using the SGD optimizer, and performance remained suboptimal even after many epochs.

Solution: To address this, I switched to the Adam optimizer, which converged faster and achieved better performance. Additionally, implementing early stopping ensured efficient training by halting when no further improvement was observed.

Challenge2: Finding the Optimal Learning Rate:

Problem: Choosing a suitable learning rate was critical. Too high a value led to unstable training, while too low a value resulted in excessively slow convergence.

Solution: I experimented with a range of learning rates and found 0.0001 to be the most effective. I also considered using a learning rate scheduler to dynamically adjust the rate during training.

Challenge3: Overfitting with Extended Training:

Problem: Training for too many epochs caused the model to overfit, especially when using larger batch sizes or higher learning rates.

Solution: I used techniques such as dropout and early stopping to mitigate overfitting and ensure the model generalized well to unseen data.

Reflections on Model Performance: Overall, the Adam optimizer combined with a learning rate of 0.0001 and a moderate batch size of 16 provided the best performance balance. The experiments highlighted the importance of tuning hyperparameters and monitoring training behavior carefully to achieve optimal results. In future iterations, more sophisticated approaches like hyperparameter optimization techniques (e.g., grid search or Bayesian optimization) could be explored to further fine-tune the model.

Conclusion

My key findings from this study reveal that smaller batch sizes (8 and 16) generally led to better accuracy and ROC scores due to more frequent weight updates, while larger batch sizes required fewer epochs but risked poorer generalization. A learning rate of 0.0001 offered the best balance between convergence speed and stability, as higher rates caused instability and lower rates slowed down training. Increasing the number of epochs improved performance up to a point, after which overfitting became a concern, highlighting the importance of finding an optimal range for each configuration. The Adam optimizer significantly outperformed SGD in both convergence speed and accuracy, thanks to its adaptive learning rate mechanism that handled noisy gradients effectively. For future work, implementing automated hyperparameter optimization techniques like grid search or Bayesian optimization could yield further improvements by refining batch size, learning rate, and epoch settings more systematically. Additionally, incorporating a learning rate scheduler might enhance convergence without risking instability, while exploring other optimizers, such as RMSprop, or regularization techniques, like weight decay, could help in managing overfitting. Cross-validation would also provide a more robust measure of generalization, ensuring consistent performance across different data splits. These enhancements could refine the model further, making it more resilient and effective across diverse scenarios.