

# 6140 Machine Learning Assignment 3

Shreeti Shrestha

Due Date: 18 November 2025

**1. Constrained Optimization.** Consider the regression problem on a dataset  $\{(x_i, y_i)\}_{i=1}^N$ , where  $x_i \in \mathbb{R}^d$  denotes the input and  $y_i$  denotes the output/response. Let  $X = [x_1 \dots x_N]^\top \in \mathbb{R}^{N \times d}$  and  $y = [y_1 \dots y_N]^\top \in \mathbb{R}^N$ . Consider the following constrained regression optimization

$$\min_{\theta} \|X\theta - y\|_2^2 \quad \text{s.t.} \quad w^\top \theta = b,$$

where  $w$  and  $b$  are given and indicate the parameters of a hyperplane on which the desired parameter vector,  $\theta$ , lies. Assume that  $X^\top X = I_d$ , where  $I_d$  is a  $d \times d$  identity matrix. Find the closed-form solution of the above regression problem for  $\theta$ .

**Answer:** Let  $f(\theta) = \|X\theta - y\|_2^2$  and  $h(\theta) = w^\top \theta - b$ .

Using the Lagrangian method,

$$L \triangleq f(\theta) + \alpha h(\theta).$$

$$L(\theta, \alpha) = \|X\theta - y\|_2^2 + \alpha(w^\top \theta - b)$$

Since the Lagrangian function, is a function of both  $x$  and  $\alpha$ , we solve to minimize with respect to both these parameters:

$$\min_{(\theta, \alpha)} L(\theta, \alpha) \rightarrow (\theta^*, \alpha^*)$$

$$\begin{aligned} \frac{\delta(L(\theta, \alpha))}{\delta \theta} &= 0 \Rightarrow eq^n1 \\ \text{or, } \frac{\delta(f(\theta))}{\delta \theta} + \alpha \frac{\delta(h(\theta))}{\delta \theta} &= 0 \\ \text{or, } \frac{\delta(\|X\theta - y\|_2^2)}{\delta \theta} + \alpha \frac{\delta(w^\top \theta - b)}{\delta \theta} &= 0 \\ \text{or, } 2X^\top(X\theta - y) + \alpha w^\top &= 0 \\ \text{(from L2 norm where } \frac{\delta(\|AX - b\|_2^2)}{\delta x} &= 2A^\top(Ax - b) \text{ and } \frac{\delta(a^\top x)}{\delta x} = a.) \\ \text{or, } 2X^\top X\theta - 2X^\top y + \alpha w^\top &= 0 \\ \text{or, } 2I_d\theta - 2X^\top y + \alpha w^\top &= 0 \\ \text{or, } 2I_d\theta = 2X^\top y - \alpha w^\top & \\ \text{or, } 2\theta = 2X^\top y - \alpha w^\top & \\ \text{or, } \theta = \frac{2X^\top y - \alpha w^\top}{2} & \\ \text{or, } \theta = \frac{2X^\top y}{2} - \frac{\alpha w^\top}{2} & \\ \text{or, } \theta = X^\top y - \frac{\alpha w^\top}{2} & \Rightarrow eq^n3 \end{aligned} \quad \begin{aligned} \frac{\delta(L(\theta, \alpha))}{\delta \alpha} &= 0 \Rightarrow eq^n2 \\ \text{or, } \frac{\delta(f(\theta))}{\delta \alpha} + \frac{\delta(h(\theta))}{\delta \alpha} &= 0 \\ \text{or, } \frac{\delta(\|X\theta - y\|_2^2)}{\delta \alpha} + \frac{\delta(\alpha(w^\top \theta - b))}{\delta \alpha} &= 0 \\ \text{or, } (w^\top \theta - b) \frac{\delta(\alpha)}{\delta \alpha} &= 0 \\ \text{or, } w^\top \theta - b = 0 & \\ \text{or, } w^\top \theta = b & \Rightarrow eq^n4 \end{aligned}$$

Substituting value of  $\theta$  in  $eq^4$  from  $eq^3$  to enforce constraint, we get:

$$\begin{aligned} w^\top(X^\top y - \frac{\alpha w^\top}{2}) &= b \\ w^\top X^\top y - \frac{w^\top \alpha w^\top}{2} &= b \\ \text{or, } w^\top X^\top y - \frac{\|w\|_2^2 \alpha}{2} &= b \text{ (dot product of a vector } w^\top w \text{ is equivalent to L2 norm)} \\ \text{or, } w^\top X^\top y - b &= \frac{\|w\|_2^2 \alpha}{2} \\ \text{or, } \frac{\|w\|_2^2 \alpha}{2} &= w^\top X^\top y - b \\ \text{or, } \|w\|_2^2 \alpha &= 2(w^\top X^\top y - b) \\ \text{or, } \alpha^* &= \frac{2(w^\top X^\top y - b)}{\|w\|_2^2} \end{aligned}$$

Plugging the value of  $\alpha^*$  in  $eq^3$ , we get :

$$\theta^* = X^\top y - \frac{\frac{2(w^\top X^\top y - b)}{\|w\|_2^2} w^\top}{\frac{\|w\|_2^2}{2}}$$

$\theta^* = X^\top y - \frac{(w^\top X^\top y - b)}{\frac{\|w\|_2^2}{2}} w$  is the closed form solution for  $\theta$  that satisfies the constraint  $w^\top \theta^* = b$

- 2. SVM.** Consider a supervised learning problem in which the training examples are points in 2-dimensional space. The positive examples (samples in class 1) are  $(1, 1)$  and  $(-1, -1)$ . The negative examples (samples in class 0) are  $(1, -1)$  and  $(-1, 1)$ . Are the positive examples linearly separable from the negative examples in the original space? If so, give the coefficients of  $w$ .

**Answer:**

These examples are not linearly separable in the original 2D space. There is no line that cuts the plane into 2 half-spaces, each of which contain only positives or only negatives.

- For the example above, consider the feature transformation  $\phi(x) = [1, x_1, x_2, x_1x_2]$ , where  $x_1$  and  $x_2$  are, respectively, the first and second coordinates of a generic example  $x$ . Can we find a hyperplane  $w^\top \phi(x)$  in this feature space that can separate the data from positive and negative class. If so, give the coefficients of  $w$  (You should be able to do this by inspection, without significant computation).

**Answer:**

For  $\phi(x) = [1, x_1, x_2, x_1x_2]$ , the four possible values based on the example points could be:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \text{ for } (1, 1), \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \text{ for } (-1, -1), \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \text{ for } (-1, 1), \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \text{ for } (1, -1)$$

For samples in class 1,  $x_1 \times x_2$  is positive. For samples in class 0,  $x_1 \times x_2$  is negative. So, hyperplane  $x_1 \times x_2 = 0$  could separate the classes perfectly.

Using the feature map  $\phi(x) = [1, x_1, x_2, x_1x_2]$ , and weight vector  $w = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ , the hyperplane  $w^\top \phi(x) = x_1x_2$  separates the classes where the decision boundary lies at  $x_1x_2 = 0$ .

- Compute  $K(x, z) = \phi(x)^\top \phi(z)$ , which is called the kernel function corresponding to the feature map  $\phi(\cdot)$  in the last part.

**Answer:**

$K(x, z) = \phi(x)^\top \phi(z)$  (we can compute directly on the original vectors/features)

$$\begin{aligned} &= \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \end{bmatrix} \times [1, z_1, z_2, z_1z_2] \\ &= 1 + x_1z_1 + x_2z_2 + x_1x_2z_1z_2 \end{aligned}$$

**3. Feedforward Neural Network.** Consider a three-layer neural network to learn a function  $f : X \rightarrow Y$ , where  $X = [X_1, X_2]$  consists of two features. The weights  $w_1, \dots, w_6$  can be arbitrary. There are two possible choices for the function implemented by each unit in this network:

- **S:** sigmoid function,  $S(z) = \frac{1}{1+\exp(-z)}$ ,
- **L:** linear function,  $L(z) = cz$ ,

where in both cases  $z = \sum_i w_i X_i$ . Assign proper activation functions (S or L) to each unit in the following graph so that we can generate functions of the form

$$f(X_1, X_2) = \frac{1}{1 + \exp(\beta_1 X_1 + \beta_2 X_2)}$$

at the output of the neural network  $Y$ . Derive  $\beta_1$  and  $\beta_2$  as a function of  $w_1, \dots, w_6$  and  $c$ .

**Answer:**

For the top hidden node,  $z_1 = w_1 X_1 + w_3 X_2$ .

For the bottom hidden node,  $z_2 = w_2 X_1 + w_4 X_2$

To achieve the above target function, we assign the following activations:

Linear function, L to hidden nodes  $h_1, h_2$

With linear activation functions:

$$h_1 = L(z_1) = L(w_1 X_1 + w_2 X_2) = c(w_1 X_1 + w_2 X_2)$$

$$h_2 = L(z_2) = L(w_3 X_1 + w_4 X_2) = c(w_3 X_1 + w_4 X_2)$$

For the output node pre-activation,

$$\begin{aligned} z_3 &= w_5 h_1 + w_6 h_2 \\ &= w_5 \cdot c(w_1 X_1 + w_2 X_2) + w_6 \cdot c(w_3 X_1 + w_4 X_2) \\ &= c[w_5(w_1 X_1 + w_2 X_2) + w_6(w_3 X_1 + w_4 X_2)] \\ &= c[(w_5 w_1 + w_6 w_3)X_1 + (w_5 w_2 + w_6 w_4)X_2] \end{aligned}$$

Then, we apply Sigmoid function to the output node such that  $Y = S(z_3)$ :

$$Y = S(z_3) = \frac{1}{1+\exp(-z_3)} = \frac{1}{1+\exp(-c[(w_5 w_1 + w_6 w_3)X_1 + (w_5 w_2 + w_6 w_4)X_2])} \Rightarrow \text{eq}^n 1$$

$$\text{We have, output}(X_1, X_2): Y = \frac{1}{1+\exp(\beta_1 X_1 + \beta_2 X_2)} \Rightarrow \text{eq}^n 2$$

Comparing the exponents from  $\text{eq}^n 1$  and  $\text{eq}^n 2$ , we get:

$$-c[(w_5 w_1 + w_6 w_3)X_1 + (w_5 w_2 + w_6 w_4)X_2] = \beta_1 X_1 + \beta_2 X_2$$

$$\text{or, } -c(w_5 w_1 + w_6 w_3)X_1 - c(w_5 w_2 + w_6 w_4)X_2 = \beta_1 X_1 + \beta_2 X_2$$

Therefore, from above, we can derive:

$$\begin{aligned} \beta_1 &= -c(w_1 w_5 + w_3 w_6) \\ \beta_2 &= -c(w_2 w_5 + w_4 w_6) \end{aligned}$$

where all hidden layers use linear activation functions, output uses the sigmoid activation, and parameters  $\beta_1$  and  $\beta_2$  are linear combinations of products of weights, scaled by  $-c$

**4. Support Vector Machine (SVM) Classification.** In this assignment, you will write Python code to implement Support Vector Machine (SVM) Classification. You can download the data via this link. The files `X_train.csv` and `X_test.csv` contain the features (each row represents a point in 2D space), and `y_train.csv` and `y_test.csv` contain the labels (one column).

1. Train an SVM classifier with a linear kernel on the training data. You can use the `sklearn.svm.SVC` module in `sklearn`.
  - (a) Compute and report the training and testing accuracy.

**Answer:**

## 1. Training SVM Classifier with a Linear Kernel

```
# Train SVM with linear kernel
model = SVC(kernel="linear", random_state=0)
model.fit(X_train, y_train)

# Predictions and accuracies
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

train_acc = accuracy_score(y_train, y_train_pred)
test_acc = accuracy_score(y_test, y_test_pred)

print(f"Training accuracy: {train_acc:.4f}")
print(f"Testing accuracy: {test_acc:.4f}")

[3]
...
Training accuracy: 0.8500
Testing accuracy: 0.7500
```

Figure 1: Training and testing accuracy for SVM classifier with linear kernel.

- (b) Plot the data and the decision boundary for linear SVM. You may refer to the following code snippets for visualizing the decision boundary.

**Answer:**

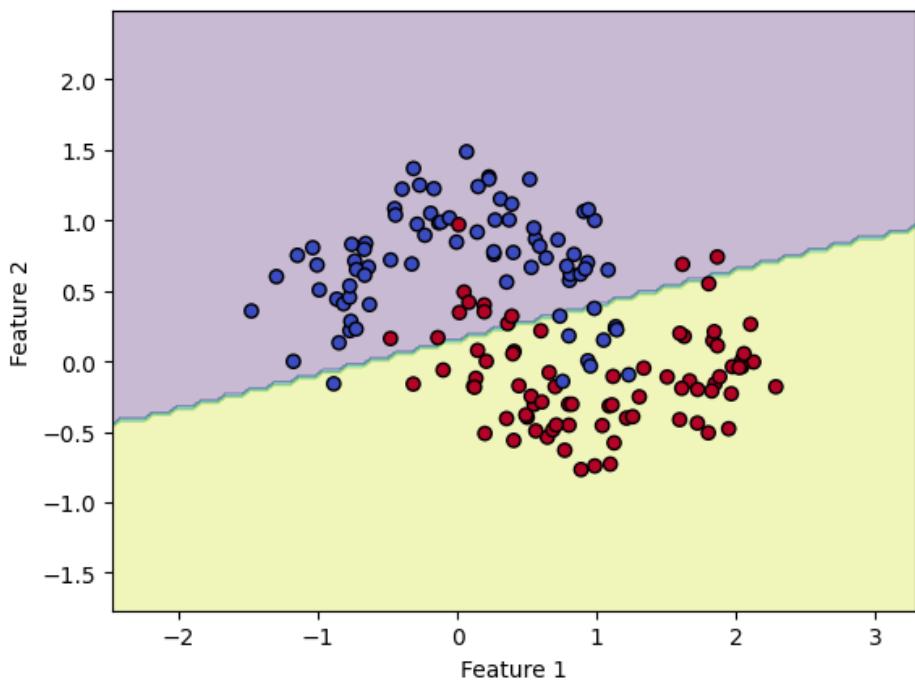


Figure 2: Training data decision boundary for linear SVM.

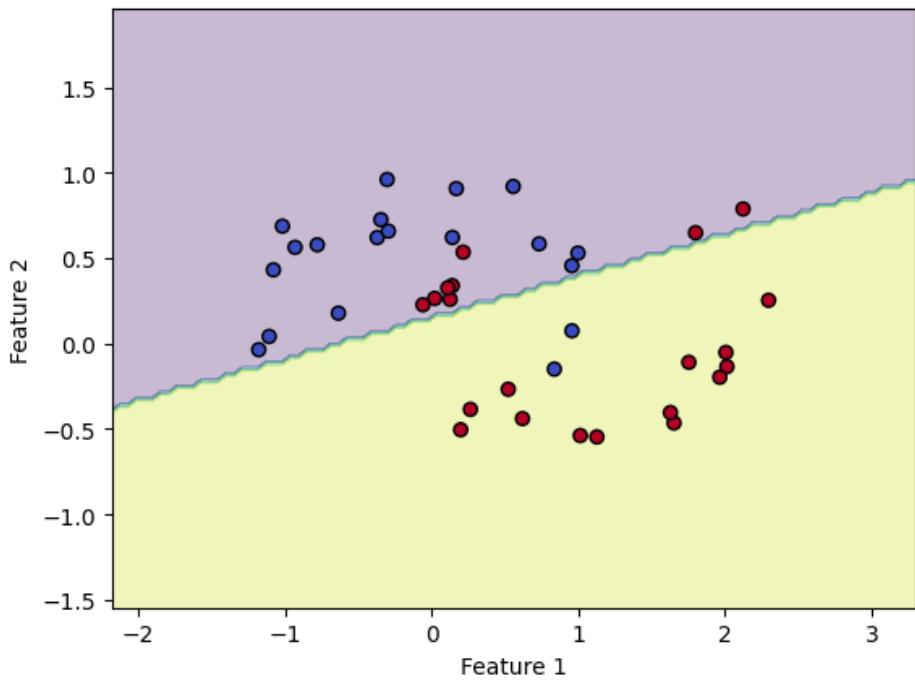


Figure 3: Testing data decision boundary for linear SVM.

2. Change the linear kernel to a RBF kernel. Train an SVM classifier with a RBF kernel on the training data.

- (a) Compute and report the training and testing accuracy.

**Answer:**

## 2. Training SVM Classifier with a RBF Kernel

```
# Train SVM with RBF kernel
model_rbf = SVC(kernel='rbf', random_state=0)
model_rbf.fit(X_train, y_train.values.ravel())

# Predictions and accuracies
y_train_pred_rbf = model_rbf.predict(X_train)
y_test_pred_rbf = model_rbf.predict(X_test)

train_acc_rbf = accuracy_score(y_train, y_train_pred_rbf)
test_acc_rbf = accuracy_score(y_test, y_test_pred_rbf)

print(f"RBF SVM - Training accuracy: {train_acc_rbf:.4f}")
print(f"RBF SVM - Testing accuracy: {test_acc_rbf:.4f}")

[6]
...
RBF SVM - Training accuracy: 0.9688
RBF SVM - Testing accuracy: 0.9250
```

Figure 4: Training and testing accuracy for RBF SVM.

- (b) Plot the data and the decision boundary for RBF kernel SVM.

**Answer:**

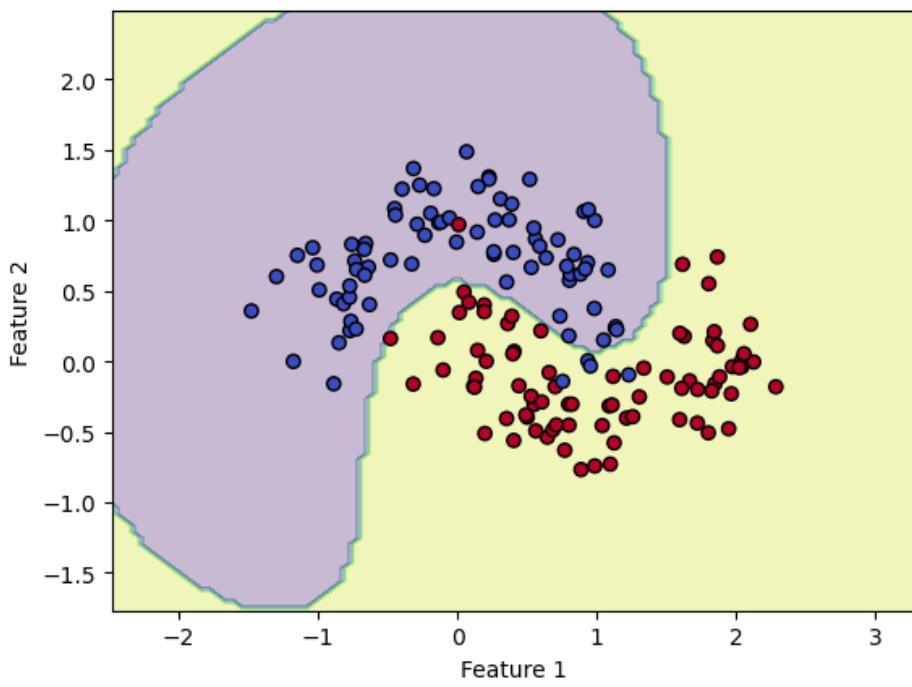


Figure 5: Training data decision boundary for RBF SVM.

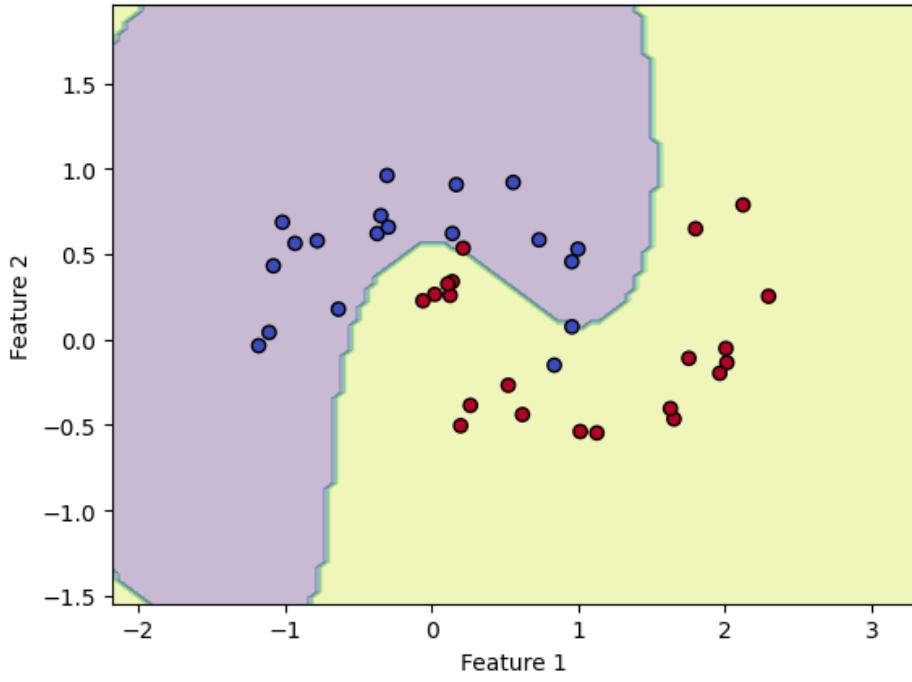


Figure 6: Testing data decision boundary for RBF SVM.

**5. Neural Network (NN) Classification.** In this assignment, you will write Python code to implement a Feedforward Neural Network (FNN) using PyTorch. We will use the same data as the previous question.

1. Implement a Feedforward Neural Network (FNN) in PyTorch.

- (a) The input size is 2, and the output size is 1.
- (b) Use ReLU activation for the hidden layers.
- (c) Use Sigmoid activation for the output layer.
- (d) Use Binary Cross-Entropy Loss (BCELoss) as the training objective.
- (e) Use the Adam optimizer.

**Answer:**

```
# Train and evaluate MLPs with 1, 2, and 3 hidden layers (16 neurons each)
# Uses existing tensors/objects: X_train_t, y_train_t, X_test_t, y_test_t, device, criterion
# Loss and optimizer
criterion = nn.BCELoss()           # Binary Cross-Entropy Loss

def make_fnn(n_hidden_layers=1, hidden_size=16, input_size=2, output_size=1):
    layers = []
    in_features = input_size
    for _ in range(n_hidden_layers):
        layers.append(nn.Linear(in_features, hidden_size))
        layers.append(nn.ReLU())
        in_features = hidden_size
    layers.append(nn.Linear(in_features, output_size))
    layers.append(nn.Sigmoid())
    return nn.Sequential(*layers)

def train_and_evaluate(model, X_train, y_train, X_test, y_test, epochs=200, lr=0.01):
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        out = model(X_train)
        loss = criterion(out, y_train)
        loss.backward()
        optimizer.step()
    model.eval()
    with torch.no_grad():
        out_train = model(X_train)
        pred_train = (out_train >= 0.5).float()
        train_acc = (pred_train.eq(y_train).float().mean().item())
        out_test = model(X_test)
        pred_test = (out_test >= 0.5).float()
        test_acc = (pred_test.eq(y_test).float().mean().item())
    return train_acc, test_acc
```

Figure 7: Implementing an FNN with the requirements.

2. Train and evaluate the model with different architectures:

- (a) Train a network with 1 hidden layer, and the number of neurons for the hidden layer is 16.
- (b) Train a network with 2 hidden layers, and the number of neurons for each hidden layer is 16.
- (c) Train a network with 3 hidden layers, and the number of neurons for each hidden layer is 16.
- (d) For each model, compute and report the training and testing accuracy.

**Answer:**

```
▶ v
results = {}
configs = {
    '1_hidden_16': 1,
    '2_hidden_16': 2,
    '3_hidden_16': 3
}

for name, n_layers in configs.items():
    torch.manual_seed(0)    # reset initialization for fair comparison
    np.random.seed(0)
    model_i = make_fnn(n_hidden_Layers=n_layers, hidden_size=16)
    train_acc, test_acc = train_and_evaluate(model_i, X_train_t, y_train_t, X_test_t, y_test_t, epochs=200, lr=0.01)
    results[name] = (train_acc, test_acc)
    print(f"{name}: train_acc={train_acc:.4f}, test_acc={test_acc:.4f}")
[12]
...
1_hidden_16: train_acc=0.9250, test_acc=0.9000
2_hidden_16: train_acc=0.9688, test_acc=0.9500
3_hidden_16: train_acc=0.9688, test_acc=0.9750
```

Figure 8: Accuracy scores from training network with 1, 2 and 3 hidden layers (increasing test data accuracy scores with increasing hidden layers: 0.90 vs 0.95 vs 0.97).

**6. CNN-based Image Classification on CIFAR-10.** In this assignment, you will use the CIFAR-10 dataset and implement a Convolutional Neural Network (CNN) in PyTorch for image classification. CIFAR-10 consists of 60,000 RGB images of size  $32 \times 32$  pixels, evenly divided into 10 distinct classes including airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset has 50,000 images for training and 10,000 for testing. You can download the dataset directly using `torchvision.datasets.CIFAR10`. Example codes for loading the CIFAR-10 dataset are as follows:

1. Implement the following CNN architecture:

- `Conv2D(in_channels=3, out_channels=32, kernel_size=3, padding=1) → ReLU → MaxPool2D(kernel_size=2, stride=2)`
- `Conv2D(32 → 64, kernel_size=3, padding=1) → ReLU → MaxPool2D(kernel_size=2, stride=2)`
- `Conv2D(64 → 128, kernel_size=3, padding=1) → ReLU → MaxPool2D(kernel_size=2, stride=2)`
- `Flatten → Linear( $128 \times 4 \times 4 = 2048 \rightarrow 256$ ) → ReLU → Linear(256 → 10)`

Use `CrossEntropyLoss` and the Adam optimizer. Set batch size as 64, learning rate as 0.001, and train the model for 20 epochs. Plot the training loss over epochs. Plot the training and testing accuracy over epochs.

**Answer:**



```

6.1 CNN architecture
Generate + Code

# CNN training on CIFAR-10
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 256), nn.ReLU(),
            nn.Linear(256, num_classes) # logits for CrossEntropyLoss
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

```

Figure 9: CNN architecture implementation

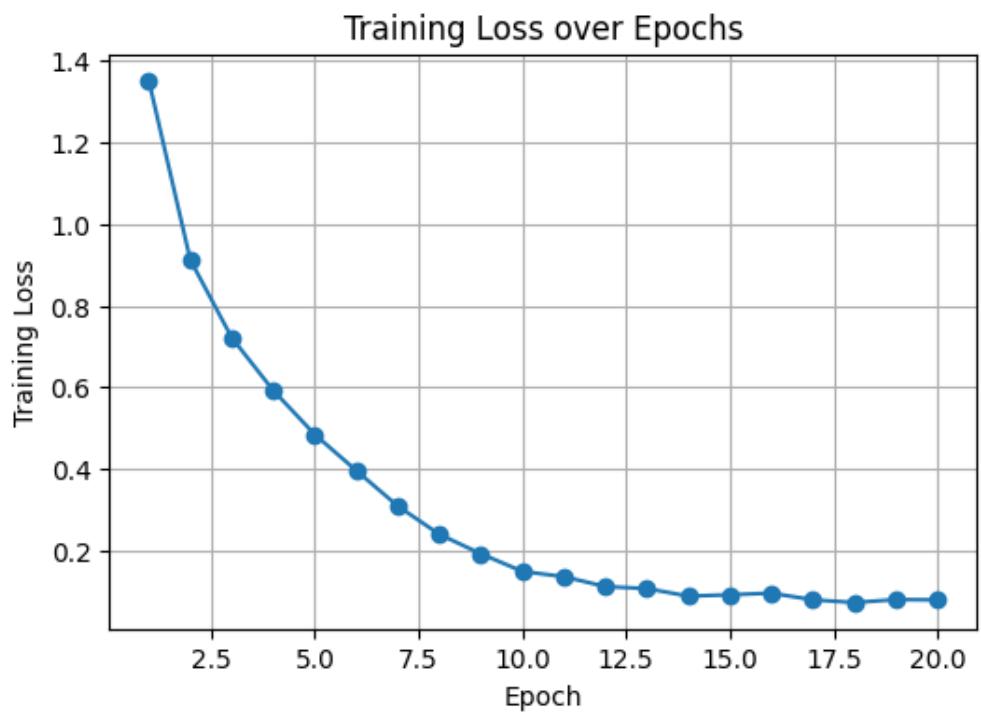


Figure 10: Training loss over 20 epochs

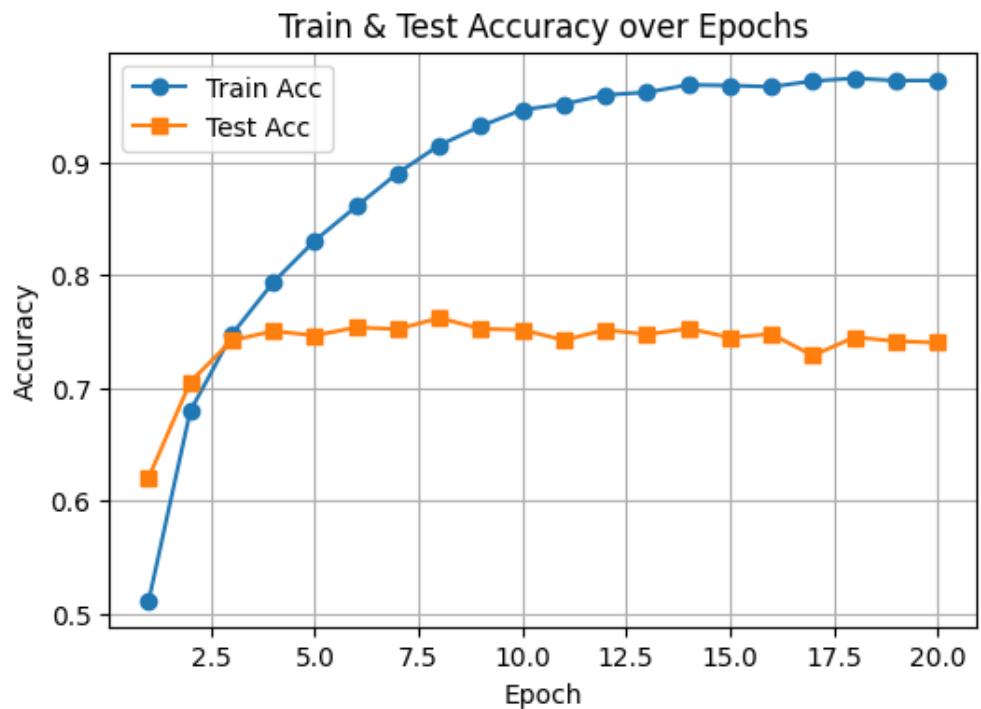


Figure 11: Training and testing accuracy over 20 epochs

2. Train and evaluate the model with different hyperparameters:

- (a) For batch size = 64, change the learning rate: [0.01, 0.001, 0.0001]. Plot training loss, training accuracy, and testing accuracy over epochs. Discuss how learning rate affects performance and convergence.

**Answer:**

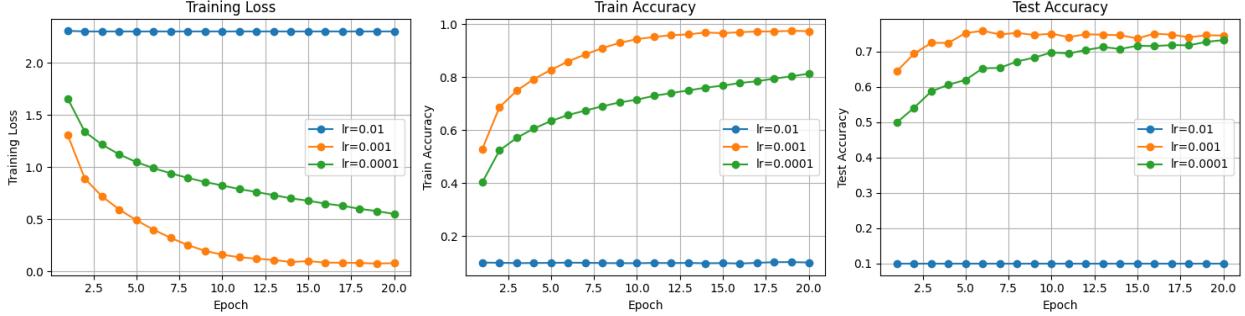


Figure 12: Training with varying learning rates.

**Discussion:** When learning rate is too large (0.01-blue), the gradient updates are too aggressive, it oscillates and diverges and the model doesn't learn anything.

When learning rate is too small (0.0001-green), it converges very slowly over a lot of epochs/iterations which makes training very slow.

Near an optimal learning rate (0.001-orange), the model learns efficiently, as shown by higher accuracy scores and converges well within 20 epochs.

- (b) For learning rate = 0.001, change the batch size: [32, 64, 128]. Plot training loss, training accuracy, and testing accuracy over epochs. Discuss how batch size affects performance and convergence.

**Answer:**

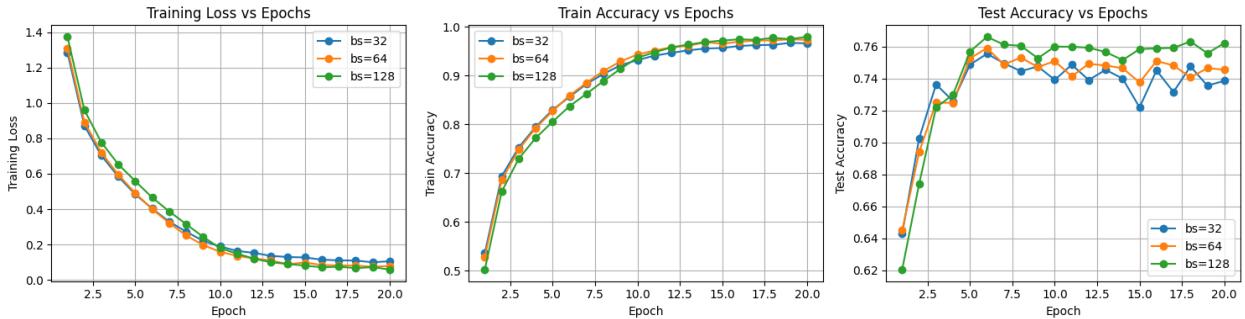


Figure 13: Training with varying batch sizes (32-blue, 64-orange, 128-green).

**Discussion:** Based on the plots, all batch sizes have similar performance and convergence speed. The higher batch size (128-green) sees higher accuracy in test scores. In general, larger batch sizes produce more smoother and stable learning curves but may converge to a sharp minima. On the other hand, lower batch sizes (32-blue) can have more noisy updates and high variance as shown in test accuracy.

- (c) For batch size = 64, and learning rate = 0.001, change the optimizer to RMSProp. Plot training loss, training accuracy, and testing accuracy over epochs. Describe your observations in the difference between RMSProp and Adam optimizers.

**Answer:**

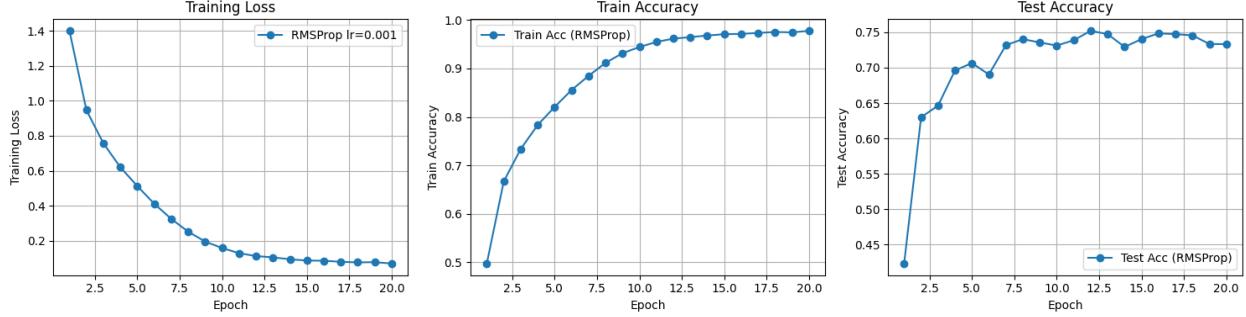


Figure 14: Training with RMSProp Optimizer (comparing with lr=0.001-orange plot from Figure 12).

**Discussion:** While trends are similar (both have low training loss, and similar final performance/accuracy scores), Adam optimizer seems to offer more stable convergence and better generalization, and perform slightly better for overall training and testing accuracies.

3. For batch size = 64, and learning rate = 0.001, modify the architecture and observe the impact:

- (a) Add a **Dropout** layer (e.g.,  $p = 0.5$ ) after the first **Linear** layer. Plot training loss, training accuracy, and testing accuracy over epochs. Discuss your observation.

**Answer:**

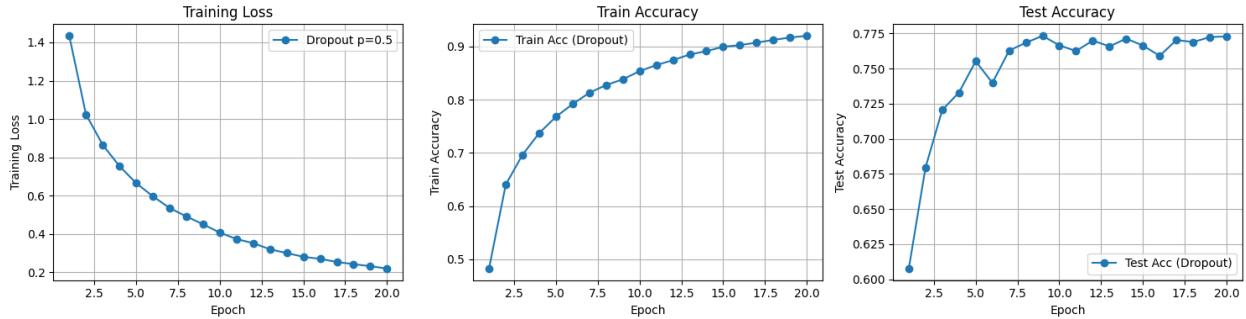


Figure 15: Adding a Dropout Layer after the first Linear Layer

**Discussion:** With about half neurons dropped in training, the training loss is now higher ( $\geq 0.2$ ) compared to previous training cases, where it was lower. The training accuracy is also relatively lower since half the neurons are disabled (0.9-0.95 vs 0.95-1.0 before). Test accuracy is however, slightly better and almost comparable (0.77 vs 0.75 before), since dropout reduces overfitting and allows better generalization to unseen data.

- (b) Further add a **BatchNorm2d** layer before each **MaxPool2D** layer. The **num\_features** argument of **BatchNorm2d** should match the **out\_channels** of the preceding **Conv2D**

layer. Plot training loss, training accuracy, and testing accuracy over epochs. Discuss your observation.

**Answer:**

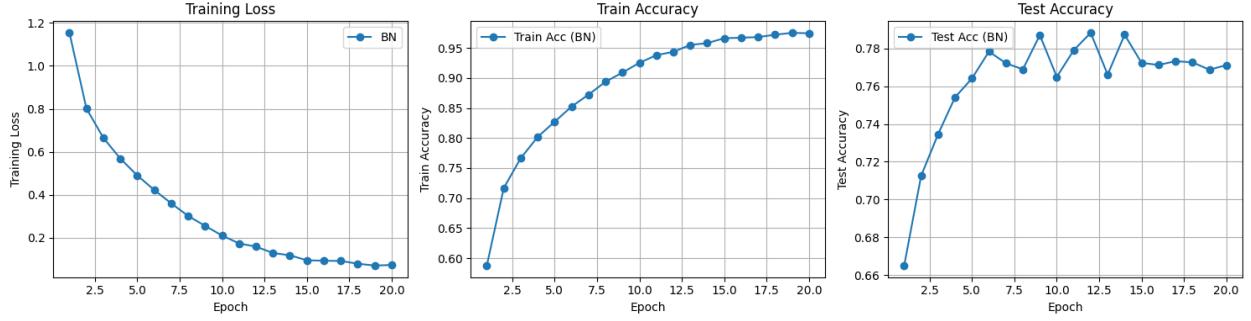


Figure 16: Adding a BatchNorm2d Layer before each MaxPool2D Layer (without Dropout)

**Discussion:** Adding a BatchNorm2d Layer makes training loss lower, and accuracies higher (0.95+ vs 0.9-0.95 before for training, 0.77-0.79 vs 0.75-0.77 before for testing). The convergence is much faster with training reaching higher accuracies and lower loss in relatively less epochs, which can be helpful when we want to speed up training.

**Answer:**

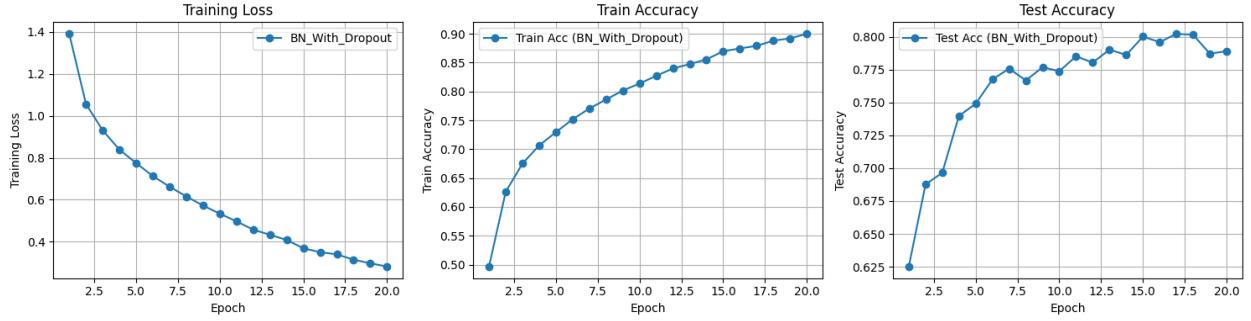


Figure 17: Adding a BatchNorm2d Layer before each MaxPool2D Layer (with Dropout)

**Discussion:** With dropout, we see lower training accuracy as expected but a dropout combined with BatchNorm2D performs by far, well with testing data (0.8 accuracy as opposed to without dropout) and improves generalization to unseen data.