EE 583 Pattern Recognition HW7

Eda Özkaynar 2375582

Question 1

This model is a Convolutional Neural Network (CNN) specifically designed for processing image data. It consists of several key components that work together to extract features and classify input images. The network begins with two convolutional layers, each followed by a max-pooling operation. The first convolutional layer processes the input image and outputs 8 feature maps, while the second layer increases this to 16 feature maps. These layers use the ReLU activation function, which enhances the network's learning capability by retaining positive values and discarding negative ones.

After the convolutional layers, the data is flattened and passed through three fully connected layers. The first fully connected layer transforms 400 inputs into 256 outputs. The second layer reduces this to 64 outputs, and the final layer produces 10 values, representing the classification results for 10 classes. To prevent overfitting, a dropout layer is applied, which randomly deactivates some neurons during training, helping the model generalize better to unseen data.

The model uses a learning rate of 0.001, controlling how much the model's parameters are updated during each training step. This commonly chosen value ensures stable and effective learning. The optimizer used is Adam, a popular choice in deep learning due to its adaptive learning rate and efficient handling of sparse gradients. This optimizer helps the model converge more quickly and reliably.

For the loss function, the model employs Cross Entropy Loss, which is well-suited for multi-class classification problems. This function measures the difference between the predicted probabilities and the actual class labels, guiding the model to minimize this error during training. Overall, the network is designed to learn effectively from image data while avoiding overfitting and ensuring accurate classification.

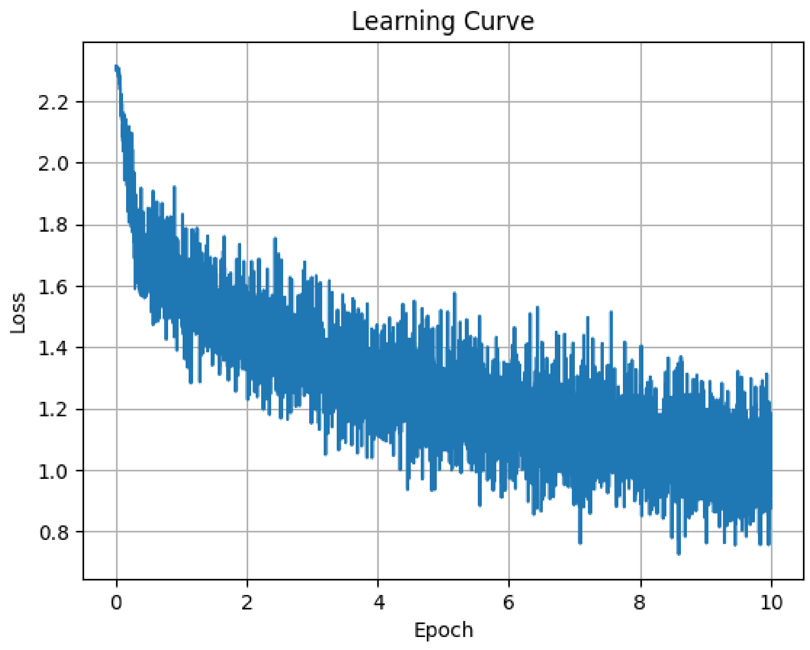


Fig. 1: Learning curve for training data

Metric	**Value**
Loss (Last epoch)	0.899
Validation Accuracy	60.00%

TABLE I: Training and Validation Metrics

Question 2

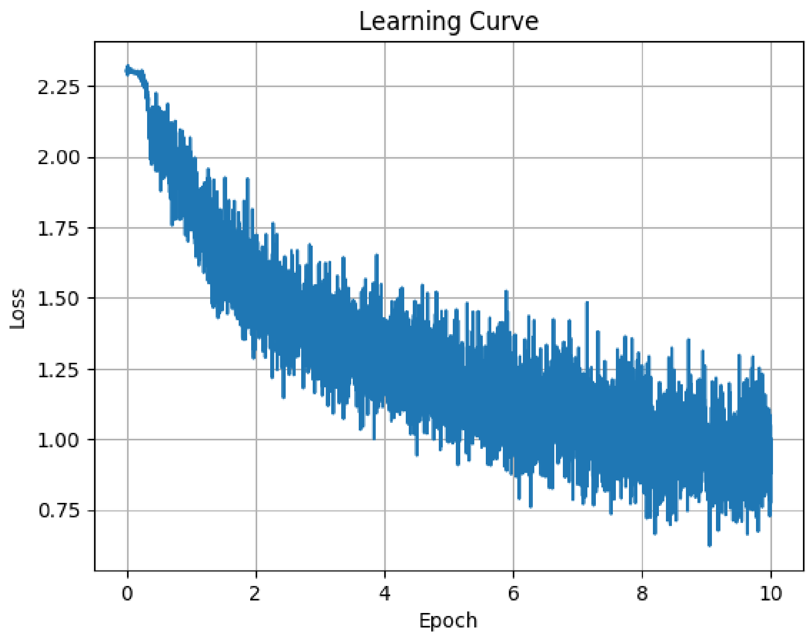


Fig. 2: Learning curve for training data (SGD optimizer)

Metric	**Value**
Loss (Lr=0.01 momentum = 0.9)	0.881
Validation Accuracy	61.00%

TABLE II: Training and Validation Metrics

SGD (Stochastic Gradient Descent) achieves a slightly better validation loss of 0.881 and an accuracy of 61.00

In Figure 1 representing Adam, the training loss starts at around 2.2 and decreases steadily from the very beginning. In Figure 2 representing SGD, the initial loss is slightly higher at 2.25, but it also decreases steadily over time.

Question 3

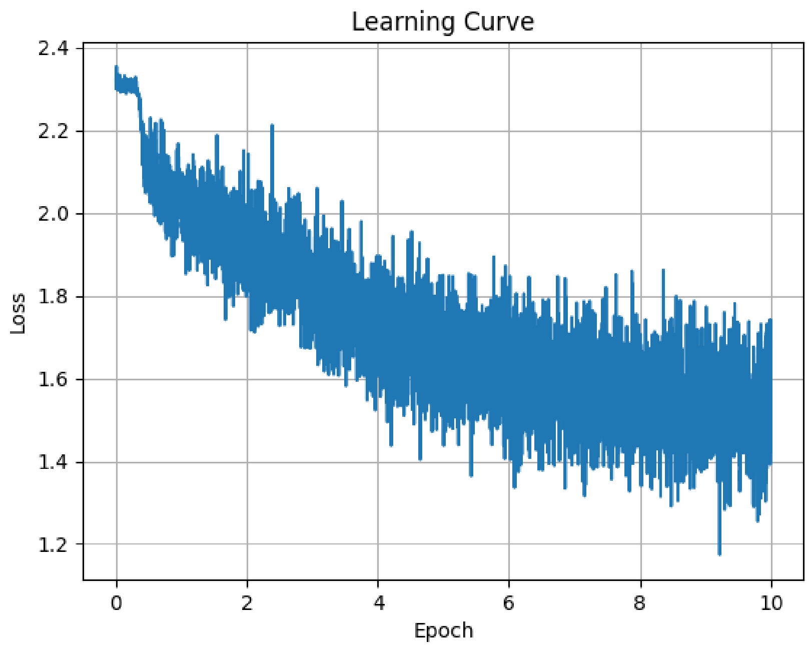


Fig. 3: Learning curve for sigmoid activation functions

Metric	Value
Loss (Last epoch)	1.445
Validation Accuracy	45.00%

TABLE III: Training and Validation Metrics

The ReLU activation function helps the loss decrease steadily and quickly during training. By the 10th epoch, the loss is about 0.9. In comparison, the Sigmoid function has a slower and less smooth convergence rate, with a loss of around 1.4 by the 10th epoch, which is much higher than that of the ReLU network. Overall, the ReLU network learns faster and achieves a lower training loss than the Sigmoid network.

The learning curve for ReLU is smooth and has few ups and downs. This activation function is known for preventing vanishing gradients, which makes training more stable. In contrast, the Sigmoid function's learning curve is often noisy and unstable. Sigmoid activations can face vanishing gradients, especially in deeper networks, which slows down learning. Therefore, ReLU provides a steadier learning curve, while the Sigmoid function shows more fluctuations and instability.

The final training loss for the ReLU network is around 0.9, indicating good optimization. The Sigmoid network, however, ends up with a final training loss of about 1.4, which is significantly higher. This means that the network is not as effective at reducing loss when using Sigmoid activations. Thus, ReLU achieves a much lower final loss than Sigmoid, making it a better choice for this task.

The reasons of the differences could be ReLU avoids the vanishing gradient problem because it does not limit values like Sigmoid does. It also allows for sparsity in the network by producing a zero output for negative inputs, which can improve learning efficiency. On the other hand, Sigmoid squashes all values into the range of [0, 1], which can cause vanishing gradients in deeper networks. Additionally, the gradients for inputs that are far from zero become very small, making learning more difficult. In conclusion, the model could not learn with the sigmoid activation function, resulting in a very poor accuracy of 45%.

Question 4

Output Size Calculation

Given:

- Input size: 32×32
- Kernel size for convolution: 6
- Kernel size for max-pooling: 2
- Stride for convolution: 1

- Stride for max-pooling: 2
- Padding: 0

After Conv1 (Kernel size = 6):

$$\begin{aligned}\text{Output size of Conv1} &= \left\lfloor \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1\\ &= \left\lfloor \frac{32 - 6 + 2 \times 0}{1} \right\rfloor + 1\\ &= \left\lfloor \frac{26}{1} \right\rfloor + 1 = 27\end{aligned}$$

Output dimensions: $27 \times 27 \times 8$

After MaxPool1 (Kernel size = 2):

$$\begin{aligned}\text{Output size of MaxPool1} &= \left\lfloor \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} \right\rfloor + 1\\ &= \left\lfloor \frac{27 - 2}{2} \right\rfloor + 1\\ &= \left\lfloor \frac{25}{2} \right\rfloor + 1 = 13\end{aligned}$$

Output dimensions: $13 \times 13 \times 8$

After Conv2 (Kernel size = 6):

$$\begin{aligned}\text{Output size of Conv2} &= \left\lfloor \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1\\ &= \left\lfloor \frac{13 - 6 + 2 \times 0}{1} \right\rfloor + 1\\ &= \left\lfloor \frac{7}{1} \right\rfloor + 1 = 8\end{aligned}$$

Output dimensions: $8 \times 8 \times 16$

After MaxPool2 (Kernel size = 2):

$$\begin{aligned}\text{Output size of MaxPool2} &= \left\lfloor \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} \right\rfloor + 1\\ &= \left\lfloor \frac{8 - 2}{2} \right\rfloor + 1\\ &= \left\lfloor \frac{6}{2} \right\rfloor + 1 = 4\end{aligned}$$

Output dimensions: $4 \times 4 \times 16$

Final Output:

Final Output Dimensions: $4 \times 4 \times 16$

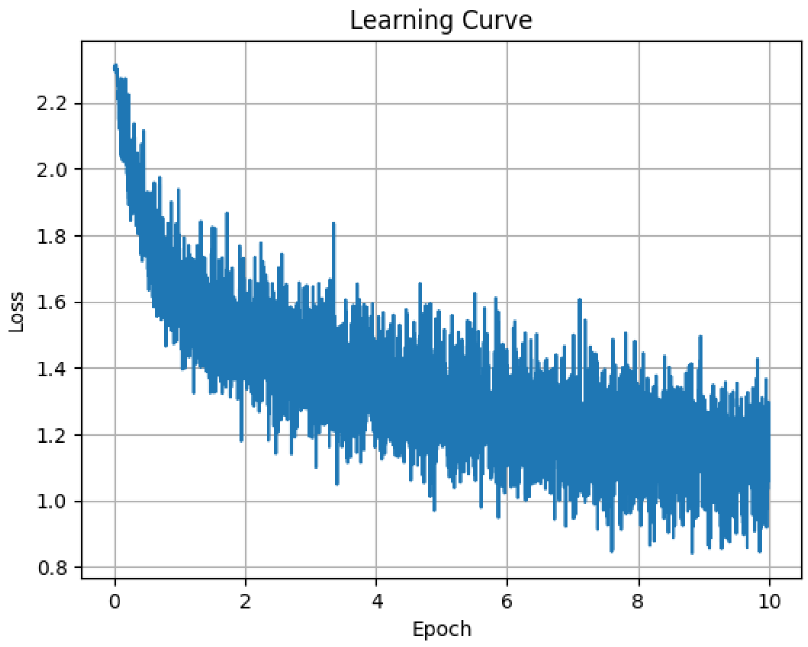


Fig. 4: Learning curve for kernel size = 6)

Metric	Value
Loss (Last epoch)	1.133
Validation Accuracy	57.00%

TABLE IV: Training and Validation Metrics for kernel size = 6

A larger kernel size reduces the number of neurons in the subsequent layer. For example, using a 6x6 kernel combines input from a larger region and results in fewer output neurons. This reduction can lead to the loss of finer details in the image. Smaller kernels capture finer details from the image, resulting in more neurons in the next layer and better feature extraction.

When the kernel size is large relative to the input, it may result in the loss of important local details. For example, if both the input and the kernel are NxN, the next layer will produce only one output neuron, resulting in significant information loss.

While a larger kernel size reduces memory usage and speeds up computations,since fewer neurons need to be processed, it sacrifices important details, which can lead to underfitting.

On the other hand, smaller kernel sizes can lead to overfitting, as the model may capture too many small patterns or noise in the data. The model trained with a kernel size of 6 achieves moderate performance: validation accuracy is 57%, and validation loss is 1.133. A kernel size of 6 balances computational efficiency and feature extraction but may compromise finer details in the input images.

QUESTION 5

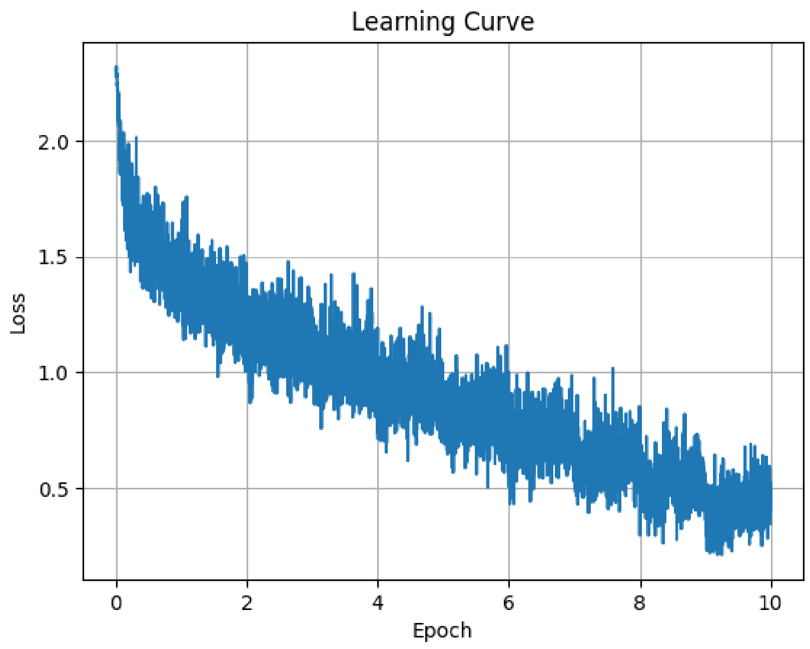


Fig. 5: Learning Curve without max-pooling

Metric	Value
Loss (Last epoch)	0.347
Validation Accuracy	62.00%

TABLE V: Training and Validation Metrics without max-pooling

Max-pooling layers are widely used in neural networks for several important reasons. They effectively summarize local regions of the input, which reduces the size of feature maps, resulting in fewer parameters and lower computational costs. By retaining only the most prominent features and discarding less relevant details, max-pooling allows the model to focus on high-level patterns. Furthermore, by decreasing the number of parameters, max-pooling plays a crucial role in preventing overfitting, especially when working with smaller datasets.

Without max-pooling, the spatial dimensions of feature maps remain larger, allowing the network to retain more detailed information from the input images. This may explain the higher validation accuracy (62Removing max-pooling also increases the number of neurons and parameters in subsequent layers. This can lead to higher memory usage and longer training times. Moreover, without max-pooling, the network may risk overfitting due to the increased number of parameters and finer details.

QUESTION 6

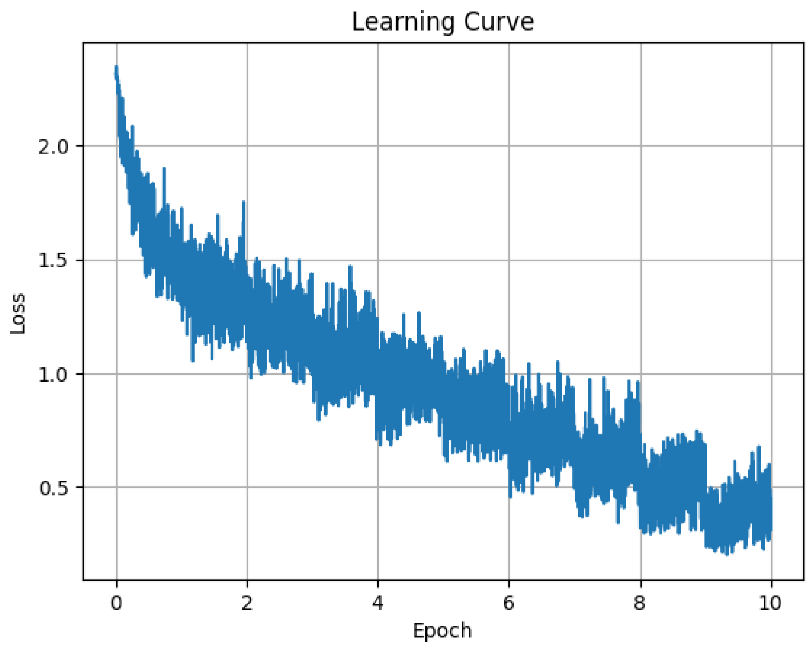


Fig. 6: Learning curve with additional convolution layer

Metric	**Value**
Loss (Last epoch)	0.339
Validation Accuracy	59.00%

TABLE VI: Training and Validation Metrics with additional convolution layer

Adding an extra layer to the model provides another opportunity to capture complex patterns and hierarchical features from the data. This improvement is reflected in a lower training loss of 0.339 compared to configurations with fewer layers, indicating that the model fits the training data effectively.

However, increasing the number of layers also raises the number of parameters in the model, leading to greater complexity. This added complexity could heighten the risk of overfitting.

The validation accuracy of 59% suggests that although the model generalizes fairly well, the increase in complexity may not significantly enhance performance on unseen data, and overfitting could be observed.

Furthermore, more layers result in higher computational requirements and increased memory usage during training.

Question 7

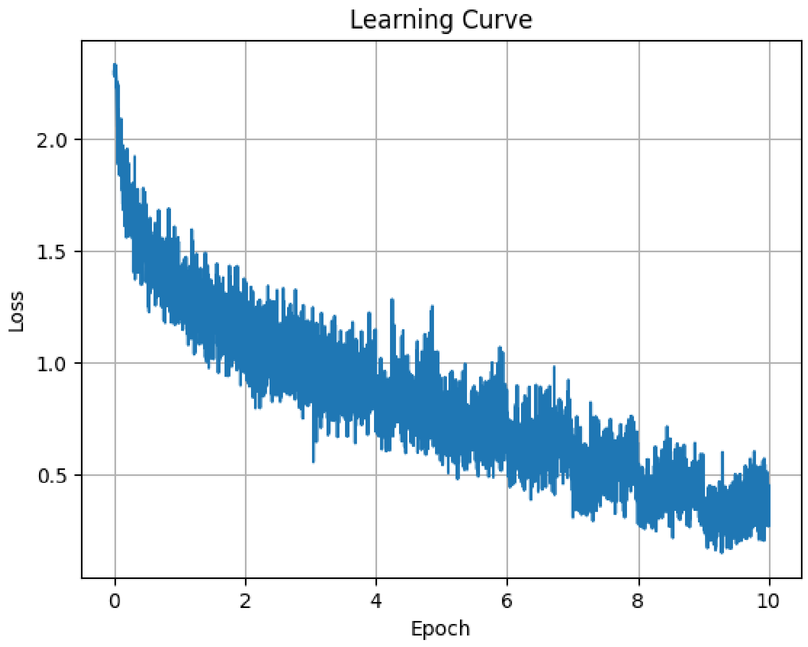


Fig. 7: Learning Curve

Metric	Value
Loss (Last epoch)	0.368
Validation Accuracy	68.00%

TABLE VII: Training and Validation Metrics

This model's architecture increases the output size of the first convolutional layer to 32, the second convolutional layer to 64, and adjusts the fully connected layers accordingly, with the first FCL increased to 1024 and the second FCL to 256. The validation accuracy reaches 68%, which is a significant improvement compared to previous configurations. This indicates that the increased capacity of the model helps it generalize better to unseen data.

By increasing the output size of the first convolutional layer to 32 and the second to 64, the model can extract more features at each stage. This allows the network to learn richer and more detailed representations of the input data.

Increasing the first FCL to 1024 and the second to 256 provides the model with greater capacity to process the extracted features. This enables it to learn more complex patterns and relationships in the data, which likely contributes to the higher validation accuracy.

The increased output sizes result in a larger number of parameters, which could lead to overfitting on smaller datasets. However, in this case, the validation accuracy improvement suggests that the model is not overfitting significantly.

Question 8

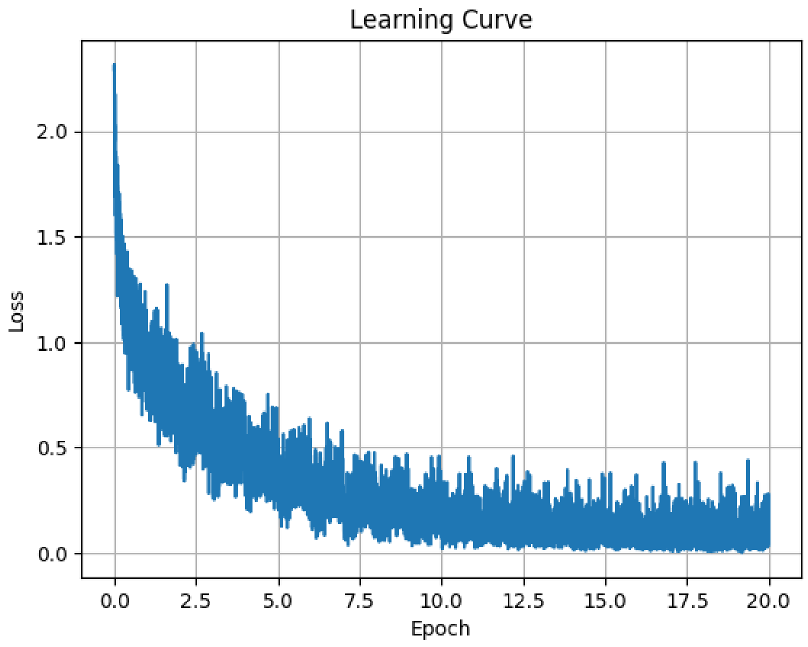


Fig. 8: Learning Curve

Metric	Value
Loss (Last epoch)	0.136
Validation Accuracy	80.00%

TABLE VIII: Training and Validation Metrics

The network begins with four convolutional layers, each progressively increasing the number of feature maps. The first layer takes a three-channel input, an RGB image, and outputs 32 feature maps. This output is passed through a ReLU activation function, followed by batch normalization, which stabilizes and accelerates training by normalizing activations. Subsequent convolutional layers expand the feature space to 64, 128, and finally 256 feature maps. Each convolutional layer applies a kernel size of 3x3 with a stride of 1, allowing the model to capture fine-grained spatial details. I have tried a model with kernel size of 4x4 but it performs poorly, then I decided to implement a model with kernel size of 3x3. Similarly, increasing number of convolution layers, increase the accuracy.

After the second, third, and fourth convolutional layers, max-pooling is applied to reduce the spatial dimensions by half, which helps in summarizing the features while reducing computational complexity. For example, if the input image size starts at 32x32 pixels, max-pooling gradually reduces this size to smaller dimensions, such as 15x15, 7x7, and so on. The feature maps from the final convolutional layer are flattened into a one-dimensional vector of size 1024, which is then passed through a sequence of fully connected layers. In first layer max-pooling is not applied since it reduced the accuracy.

The fully connected layers serve to combine the extracted features into high-level representations. The first fully connected layer takes 1024 inputs and outputs 2048 neurons, followed by the second fully connected layer, which reduces this to 256 neurons. Both of these layers use ReLU activations to introduce non-linearity, and dropout with a rate of 0.4 is applied after each fully connected layer to prevent overfitting by randomly deactivating some neurons during training. The final fully connected layer outputs 10 neurons, corresponding to the number of classes in the classification task.

Batch normalization is applied after every convolutional layer to normalize the intermediate activations, making the network less sensitive to initialization and enabling faster convergence. This also improves generalization to unseen data. Dropout further helps in regularizing the model and reducing overfitting.

Available codes: https://github.com/eozkaynar/EE583-Pattern-Recognition/tree/main/HW7

```
import torch
import numpy as np
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from torchsummary import summary
```

```
#DEFINE YOUR DEVICE
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

print(device) #if cpu, go Runtime-> Change runtime type-> Hardware accelerator GPU -> Save -> Redo previous steps
```

```
cuda:0
```

```
#DOWNLOAD CIFAR-10 DATASET
train_data = datasets.CIFAR10('./data', train = True, download = True, transform = transforms.ToTensor())

test_data = datasets.CIFAR10('./data', train = False, transform = transforms.ToTensor())
```

```
Files already downloaded and verified
```

```
#DEFINE DATA GENERATOR
batch_size = 100
train_generator = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle = True)

test_generator = torch.utils.data.DataLoader(test_data, batch_size = batch_size, shuffle = False)
```

Üretilen kod lisansa tabi olabilir | Shefin-CSE16/Bangla-Handwritten-Character-Recognition

```
#DEFINE NEURAL NETWORK MODEL
class CNN(torch.nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = torch.nn.Conv2d(3, 32, kernel_size = 3, stride = 1)
    self.conv2 = torch.nn.Conv2d(32, 64, kernel_size = 3, stride = 1)
    self.conv3 = torch.nn.Conv2d(64, 128, kernel_size = 3, stride = 1)
    self.conv4 = torch.nn.Conv2d(128, 256, kernel_size = 3, stride = 1)
    self.mpool = torch.nn.MaxPool2d(2)

    self.fc1 = torch.nn.Linear(1024, 2048)  # Increased output size to 1024
    self.fc2 = torch.nn.Linear(2048, 256)  # Increased output size to 256
    self.fc3 = torch.nn.Linear(256, 10)

    self.relu = torch.nn.ReLU()
    self.sigmoid = torch.nn.Sigmoid()
    self.drop = torch.nn.Dropout(0.4)

    self.bn1 =  torch.nn.BatchNorm2d(32)
    self.bn2 =  torch.nn.BatchNorm2d(64)
    self.bn3 =  torch.nn.BatchNorm2d(128)
    self.bn4 =  torch.nn.BatchNorm2d(256)
  def forward(self, x):
    hidden = self.bn1(self.relu(self.conv1(x)))
    hidden = self.mpool(self.bn2(self.relu(self.conv2(hidden))))
    hidden = self.mpool(self.bn3(self.relu(self.conv3(hidden))))
    hidden = self.mpool(self.bn4(self.relu(self.conv4(hidden))))
    hidden = hidden.view(-1,1024)
    hidden = self.relu(self.fc1(hidden))
    hidden = self.drop(hidden)
    hidden = self.relu(self.fc2(hidden))
    hidden = self.drop(hidden)
    output = self.fc3(hidden)
    return output
```

```
#CREATE MODEL
model = CNN()
model.to(device)
summary(model,(3,32,32))
```

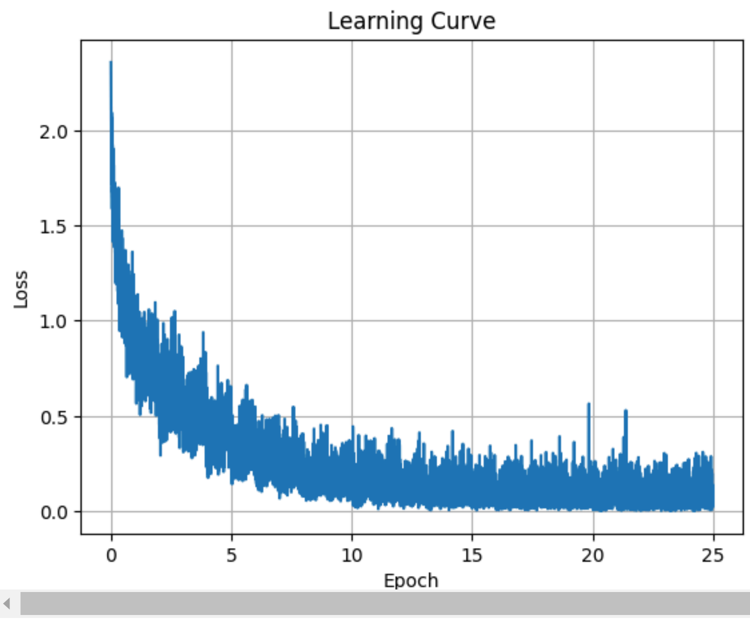
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             896
              ReLU-2           [-1, 32, 30, 30]               0
       BatchNorm2d-3           [-1, 32, 30, 30]              64
            Conv2d-4           [-1, 64, 28, 28]          18,496
              ReLU-5           [-1, 64, 28, 28]               0
       BatchNorm2d-6           [-1, 64, 28, 28]             128
         MaxPool2d-7           [-1, 64, 14, 14]               0
            Conv2d-8          [-1, 128, 12, 12]          73,856
```

```
                ReLU-9          [-1, 128, 12, 12]               0
         BatchNorm2d-10          [-1, 128, 12, 12]             256
           MaxPool2d-11            [-1, 128, 6, 6]               0
              Conv2d-12            [-1, 256, 4, 4]         295,168
                ReLU-13            [-1, 256, 4, 4]               0
         BatchNorm2d-14            [-1, 256, 4, 4]             512
           MaxPool2d-15            [-1, 256, 2, 2]               0
              Linear-16                 [-1, 2048]       2,099,200
                ReLU-17                 [-1, 2048]               0
             Dropout-18                 [-1, 2048]               0
              Linear-19                  [-1, 256]         524,544
                ReLU-20                  [-1, 256]               0
             Dropout-21                  [-1, 256]               0
              Linear-22                   [-1, 10]           2,570
================================================================
Total params: 3,015,690
Trainable params: 3,015,690
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.51
Params size (MB): 11.50
Estimated Total Size (MB): 14.03
----------------------------------------------------------------
```

```
#DEFINE LOSS FUNCTION AND OPTIMIZER
learning_rate = 0.001

loss_fun = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)


#TRAIN THE MODEL
model.train()
epoch = 25

num_of_batch=np.int32(len(train_generator.dataset)/batch_size)

loss_values = np.zeros(epoch*num_of_batch)
for i in range(epoch):
  for batch_idx, (x_train, y_train) in enumerate(train_generator):
    x_train, y_train = x_train.to(device), y_train.to(device)
    optimizer.zero_grad()
    y_pred = model(x_train)
    loss = loss_fun(y_pred, y_train)
    loss_values[num_of_batch*i+batch_idx] = loss.item()
    loss.backward()
    optimizer.step()
    if (batch_idx+1) % batch_size == 0:
        print('Epoch: {}/{} [Batch: {}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            i+1, epoch, (batch_idx+1) * len(x_train), len(train_generator.dataset),
            100. * (batch_idx+1) / len(train_generator), loss.item()))
```

```
Epoch: 21/25 [Batch: 20000/50000 (40%)] Loss: 0.039289
Epoch: 21/25 [Batch: 30000/50000 (60%)] Loss: 0.025615
Epoch: 21/25 [Batch: 40000/50000 (80%)] Loss: 0.048198
Epoch: 21/25 [Batch: 50000/50000 (100%)]        Loss: 0.113723
Epoch: 22/25 [Batch: 10000/50000 (20%)] Loss: 0.002894
Epoch: 22/25 [Batch: 20000/50000 (40%)] Loss: 0.069682
Epoch: 22/25 [Batch: 30000/50000 (60%)] Loss: 0.020700
Epoch: 22/25 [Batch: 40000/50000 (80%)] Loss: 0.049101
Epoch: 22/25 [Batch: 50000/50000 (100%)]        Loss: 0.059412
Epoch: 23/25 [Batch: 10000/50000 (20%)] Loss: 0.141085
Epoch: 23/25 [Batch: 20000/50000 (40%)] Loss: 0.027072
Epoch: 23/25 [Batch: 30000/50000 (60%)] Loss: 0.117825
Epoch: 23/25 [Batch: 40000/50000 (80%)] Loss: 0.061632
Epoch: 23/25 [Batch: 50000/50000 (100%)]        Loss: 0.060520
Epoch: 24/25 [Batch: 10000/50000 (20%)] Loss: 0.072651
Epoch: 24/25 [Batch: 20000/50000 (40%)] Loss: 0.048862
Epoch: 24/25 [Batch: 30000/50000 (60%)] Loss: 0.098408
Epoch: 24/25 [Batch: 40000/50000 (80%)] Loss: 0.243279
Epoch: 24/25 [Batch: 50000/50000 (100%)]        Loss: 0.083427
Epoch: 25/25 [Batch: 10000/50000 (20%)] Loss: 0.039074
Epoch: 25/25 [Batch: 20000/50000 (40%)] Loss: 0.005645
Epoch: 25/25 [Batch: 30000/50000 (60%)] Loss: 0.136557
Epoch: 25/25 [Batch: 40000/50000 (80%)] Loss: 0.101074
Epoch: 25/25 [Batch: 50000/50000 (100%)]        Loss: 0.135330
```

```
#PLOT THE LEARNING CURVE
iterations = np.linspace(0,epoch,num_of_batch*epoch)
plt.plot(iterations, loss_values)
plt.title('Learning Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid('on')
```



```
#TEST THE MODEL
model.eval()
correct=0
total=0

for x_val, y_val in test_generator:
  x_val = x_val.to(device)
  y_val = y_val.to(device)

  output = model(x_val)
  y_pred = output.argmax(dim=1)

  for i in range(y_pred.shape[0]):
    if y_val[i]==y_pred[i]:
      correct += 1
    total +=1

print('Validation accuracy: %.2f%%' %((100*correct)//(total)))
```

```
Validation accuracy: 80.00%
```