```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/Colab_Notebooks/lecture\ 10
```

```
Mounted at /content/drive
/content/drive/MyDrive/Colab_Notebooks/lecture 10
```

Lecture 10.3: Deep Neural Network Optimization and Regularization

1. Stochastic Gradient Descent (SGD)

1.1 Introduction

Stochastic Gradient Descent is an iterative method used for optimizing objective functions, especially suitable for large-scale datasets. SGD differs from the standard gradient descent method. In standard gradient descent, we use all the training data to compute the gradient and update the model parameters, which can be very time-consuming when dealing with large datasets. In contrast, SGD uses only one sample (or a small batch of samples, known as mini-batch SGD) per iteration to compute the gradient and update parameters, greatly accelerating the optimization process.

- Difference between Standard Gradient Descent and Stochastic Gradient Descent
  - **Standard Gradient Descent:** Uses all training data to compute the gradient and update the model parameters.
  - **Stochastic Gradient Descent:** Uses one sample (or a mini-batch of samples) per iteration to compute the gradient and update parameters.
- Advantage
  - It is useful and efficient when dealing with large-scale datasets due to its one-sample-per-iteration approach (or a mini-batch of samples), reducing the computational cost significantly.

```
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch
import torch.nn.functional as F
```

Mathematical Formulation

Stochastic Gradient Descent (SGD) is an optimization technique used to minimize (or maximize) an objective function that is the summation of differentiable functions. The fundamental equation of SGD is as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha^{(t)} \nabla f(\theta^{(t)}).$$

Here,

- $\theta^{(t+1)}$ represents the updated parameter at the next iteration,
- $\theta^{(t)}$ is the current parameter,
- $\alpha^{(t)}$ is the learning rate, a hyperparameter that determines the step size at each iteration while moving towards a minimum of the objective function,
- $\nabla f(\theta^{(t)})$ represents the gradient of the objective function f with respect to the parameter $\theta^{(t)}$.

Difference between SGD and Gradient Descent

- **Gradient Descent (Batch Gradient Descent):**

  $$\theta^{(t+1)} = \theta^{(t)} - \alpha^{(t)} \bigtriangledown_{\theta^{(t)}} \frac{1}{M} \sum_{i=1}^{M} \ell(f(x^{(i)}; \theta^{(t)}), y^{(i)}).$$

  Here, $(x^{(i)}, y^{(i)})$ represents one training sample. i represents the index of a single randomly selected sample. This method calculates the average gradient using all training samples to update the parameters.

- **Stochastic Gradient Descent (SGD):**

  $$\theta^{(t+1)} = \theta^{(t)} - \alpha^{(t)} \nabla_{\theta^{(t)}} \ell(f(x^{(i)}; \theta^{(t)}), y^{(i)}).$$

  This method uses the gradient calculated using one randomly selected training sample to update the parameters.

- **Mini-batch Stochastic Gradient Descent:**

  $$\theta^{(t+1)} = \theta^{(t)} - \alpha^{(t)} \nabla_{\theta^{(t)}} \frac{1}{B} \sum_{i=1}^{B} \ell(f(x^{(i)}; \theta^{(t)}), y^{(i)}).$$

  Here, B represents the batch size. This method uses the average gradient calculated using a mini-batch of training samples to update the parameters.

Summary

In SGD, we update our parameters using the gradient of the objective function with respect to the parameters, based on a single training example at each iteration, unlike gradient descent that uses the whole training dataset to compute the gradient. This leads to faster iterations and is especially useful when dealing with large datasets.

1.2 Implementation

Goal: Implement a Manual SGD optimizer class to optimize the parameters of a given model.

To do so, we will define a `ManualSGD` class, which contains the parameters to be optimized, the method to perform a step of optimization, and the method to zero out the gradients.

In this class:

- The constructor `__init__(self, params, lr)` initializes the optimizer with `params` (the parameters to be optimized) and `lr` (the learning rate).
- The `step` method updates the parameters using their gradient and the learning rate.
- The `zero_grad` method zeros out the gradients of the parameters to prepare for the next optimization step.

```
class ManualSGD:
    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr

    def step(self):
        for param in self.params:
            if param.grad is not None:
                param.data -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

2. Adaptive Moment Estimation （Adam)

2.1 Introduction

For a successful deep-learning project, the optimization algorithm plays a crucial role. SGD has played a major role in many successful deep-learning projects and research experiments.
But SGD has its own limitations as well. The requirement of excessive tuning of the hyperparameters is one of them. Recently, the Adam optimization algorithm has gained a lot of popularity.

Mathematical Formulation

Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called Adam is that it uses estimations of the first and second moments of the gradient to adapt the learning rate (α) for each weight of the neural network. Moment variables are initialized as $s^{(0)} = 0$ and $r^{(0)} = 0$. The formula for calculating gradient estimation is as follows:

$$\hat{g} = \bigtriangledown_{\theta^{(t)}} \ell f((x^{(i)}; \theta^{(t)}), y^{(i)}).$$

Here, $(x^{(i)}, y^{(i)})$ represents the number of training samples. i represents the index of a single randomly selected sample.
That gradient of the cost function of the neural network can be considered as a random variable, since it is usually evaluated on some small random batch of data. To estimate the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$s^{(t+1)} = \rho_1 s^{(t)} + (1 - \rho_1)\hat{g}$$

$$r^{(t+1)} = \rho_2 r^{(t)} + (1 - \rho_2)\hat{g} \odot \hat{g},$$

where s and r are moving averages, ρ_1, ρ_2 are newly introduced hyper-parameters of the algorithm. They have really good default values of 0.9 and 0.999 respectively.
Now we need to correct the estimator so that the expected value is the one we want. This step is usually referred to as bias correction. The final formulas for our estimator will be as follows:

$$\hat{s} = \frac{s^{(t+1)}}{1 - \rho_1^{t+1}}$$

$$\hat{r} = \frac{r^{(t+1)}}{1 - \rho_2^{t+1}}.$$

The only thing left to do is to use those moving averages to scale the learning rate individually for each parameter. To update the weight, we do

the following:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{\hat{s}}{\sqrt{\hat{r}} + \delta},$$

where $\theta^{(t)}$ is the current weight, $\theta^{(t+1)}$ represents the updated weight at the next iteration, α is the global learning rate. Typically we pick $\delta = 10^{-8}$ for a good trade-off between numerical stability and fidelity.

2.2 Implementation

Goal: Implement a Manual Adam optimizer class to optimize the parameters of a given model.
To do so, we will define a `ManualAdam` class, which leverages the power of adaptive learning rates methods to find individual learning rates for each parameter.

In this class:

- The constructor `__init__(self, params, lr, beta1, beta2, epsilon)` initializes the optimizer with `params` (the parameters to be optimized), `lr` (the learning rate), `beta1` and `beta2` (the attenuation factors used to control the first and second moment estimates, which are set by default to 0.9 and 0.999, respectively) and `epsilon` (a small constant used to prevent division by zero errors, with the default value of 10^{-8}).
- The `step` method updates the parameters using their learning rate.
- The `zero_grad` method is used to zero the gradients of the model parameters so that the gradients are recalculated in each iteration.

```
class ManualAdam:
    def __init__(self, params, lr=1e-3, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.params = list(params)
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = [torch.zeros_like(p) for p in self.params]
        self.v = [torch.zeros_like(p) for p in self.params]
        self.t = 0

    def step(self):
        self.t += 1
        for i, param in enumerate(self.params):
            if param.grad is not None:
                self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * param.grad
                self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * param.grad ** 2

                m_hat = self.m[i] / (1 - self.beta1 ** self.t)
                v_hat = self.v[i] / (1 - self.beta2 ** self.t)

                param.data -= self.lr * m_hat / (torch.sqrt(v_hat) + self.epsilon)

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

3. Comparisons with PyTorch SGD and Adam Optimizer

3.1 Model Architecture and Dataset

Model Architecture

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
def evaluate_model(model, testloader):
    correct = 0
    total = 0
    model.eval()
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

Dataset

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = DataLoader(testset, batch_size=4, shuffle=False)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:03<00:00, 49099393.91it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

3.2 Comparison with PyTorch SGD

Result of Manual SGD

```
model_manual_sgd = CNN()
criterion = nn.CrossEntropyLoss()
optimizer_manual = ManualSGD(model_manual_sgd.parameters(), lr=0.001)

for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_manual.zero_grad()
        outputs = model_manual_sgd(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_manual.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[Manual SGD, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with Manual SGD')
model_manual_sgd.eval()
accuracy_manual_sgd = evaluate_model(model_manual_sgd, testloader)
print('Accuracy of the network on the 10000 test images using Manual SGD: %d %%' % accuracy_manual_sgd)
```

```
[Manual SGD, 1,  2000] loss: 2.304
[Manual SGD, 1,  4000] loss: 2.302
[Manual SGD, 1,  6000] loss: 2.300
[Manual SGD, 1,  8000] loss: 2.296
[Manual SGD, 1, 10000] loss: 2.289
[Manual SGD, 1, 12000] loss: 2.266
[Manual SGD, 2,  2000] loss: 2.185
[Manual SGD, 2,  4000] loss: 2.106
[Manual SGD, 2,  6000] loss: 2.026
[Manual SGD, 2,  8000] loss: 1.946
[Manual SGD, 2, 10000] loss: 1.867
[Manual SGD, 2, 12000] loss: 1.799
Finished Training with Manual SGD
Accuracy of the network on the 10000 test images using Manual SGD: 35 %
```

Result of PyTorch SGD

```
model_torch_sgd = CNN()
criterion = nn.CrossEntropyLoss()
optimizer_torch = optim.SGD(model_torch_sgd.parameters(), lr=0.001)

for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_torch.zero_grad()
```

```
        outputs = model_torch_sgd(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_torch.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[Torch SGD, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with PyTorch SGD')
model_torch_sgd.eval()
accuracy_torch_sgd = evaluate_model(model_torch_sgd, testloader)
print('Accuracy of the network on the 10000 test images using PyTorch SGD: %d %%' % accuracy_torch_sgd)
```

```
[Torch SGD, 1,  2000] loss: 2.305
[Torch SGD, 1,  4000] loss: 2.302
[Torch SGD, 1,  6000] loss: 2.301
[Torch SGD, 1,  8000] loss: 2.300
[Torch SGD, 1, 10000] loss: 2.297
[Torch SGD, 1, 12000] loss: 2.292
[Torch SGD, 2,  2000] loss: 2.275
[Torch SGD, 2,  4000] loss: 2.225
[Torch SGD, 2,  6000] loss: 2.135
[Torch SGD, 2,  8000] loss: 2.049
[Torch SGD, 2, 10000] loss: 1.958
[Torch SGD, 2, 12000] loss: 1.926
Finished Training with PyTorch SGD
Accuracy of the network on the 10000 test images using PyTorch SGD: 31 %
```

3.3 Comparison with PyTorch Adam

Result of Manual Adam

```
model_manual_adam = CNN()
criterion = nn.CrossEntropyLoss()
optimizer_manual_adam = ManualAdam(model_manual_adam.parameters(), lr=0.001)


for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_manual_adam.zero_grad()
        outputs = model_manual_adam(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_manual_adam.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[Manual Adam, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with Manual Adam')
model_manual_adam.eval()
accuracy_manual_adam = evaluate_model(model_manual_adam, testloader)
print('Accuracy of the network on the 10000 test images using Manual Adam: %d %%' % accuracy_manual_adam)
```

```
[Manual Adam, 1,  2000] loss: 1.855
[Manual Adam, 1,  4000] loss: 1.596
[Manual Adam, 1,  6000] loss: 1.535
[Manual Adam, 1,  8000] loss: 1.475
[Manual Adam, 1, 10000] loss: 1.439
[Manual Adam, 1, 12000] loss: 1.415
[Manual Adam, 2,  2000] loss: 1.342
[Manual Adam, 2,  4000] loss: 1.332
[Manual Adam, 2,  6000] loss: 1.311
[Manual Adam, 2,  8000] loss: 1.303
[Manual Adam, 2, 10000] loss: 1.283
[Manual Adam, 2, 12000] loss: 1.278
Finished Training with Manual Adam
Accuracy of the network on the 10000 test images using Manual Adam: 53 %
```

Result of PyTorch Adam

```
model_torch_adam = CNN()
criterion = nn.CrossEntropyLoss()
optimizer_torch = optim.Adam(model_torch_adam.parameters(),lr=0.001)

for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
```

```
        inputs, labels = data
        optimizer_torch.zero_grad()
        outputs = model_torch_adam(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_torch.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[Torch Adam, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with PyTorch Adam')
model_torch_adam.eval()
accuracy_torch_adam = evaluate_model(model_torch_adam, testloader)
print('Accuracy of the network on the 10000 test images using PyTorch Adam: %d %%' % accuracy_torch_adam)
```

```
[Torch Adam, 1,  2000] loss: 1.859
[Torch Adam, 1,  4000] loss: 1.602
[Torch Adam, 1,  6000] loss: 1.518
[Torch Adam, 1,  8000] loss: 1.472
[Torch Adam, 1, 10000] loss: 1.399
[Torch Adam, 1, 12000] loss: 1.368
[Torch Adam, 2,  2000] loss: 1.288
[Torch Adam, 2,  4000] loss: 1.281
[Torch Adam, 2,  6000] loss: 1.252
[Torch Adam, 2,  8000] loss: 1.253
[Torch Adam, 2, 10000] loss: 1.252
[Torch Adam, 2, 12000] loss: 1.226
Finished Training with PyTorch Adam
Accuracy of the network on the 10000 test images using PyTorch Adam: 56 %
```

4. Batch Normalization, Weight Initialization and Learning Rate Schedule

4.1 Batch Normalization

Batch normalization, often abbreviated as "batchnorm", is introduced as a method to facilitate the synchronization of updates across multiple layers within a model. It offers an elegant approach to reparameterize nearly any deep neural network, thereby substantially alleviating the challenge of coordinating updates across numerous layers.

Model Architecture and Model Training

```
class CNNWithBN(nn.Module):
    def __init__(self):
        super(CNNWithBN, self).__init__()

        # Convolution Layer 1
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.bn1 = nn.BatchNorm2d(6)
        self.pool = nn.MaxPool2d(2, 2)

        # Convolution Layer 2
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.bn2 = nn.BatchNorm2d(16)

        # Fully Connected Layer 1
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.bn3 = nn.BatchNorm1d(120)

        # Fully Connected Layer 2
        self.fc2 = nn.Linear(120, 84)
        self.bn4 = nn.BatchNorm1d(84)

        # Output Layer
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.bn3(self.fc1(x)))
        x = F.relu(self.bn4(self.fc2(x)))
        x = self.fc3(x)
        return x

model_torch_adam = CNNWithBN()
criterion = nn.CrossEntropyLoss()
optimizer_torch = optim.Adam(model_torch_adam.parameters(),lr=0.001)

for epoch in range(2):
    running_loss = 0.0
```

```
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_torch.zero_grad()
        outputs = model_torch_adam(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_torch.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[Torch Adam, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with BN')
model_torch_adam.eval()
accuracy_torch_adam = evaluate_model(model_torch_adam, testloader)
print('Accuracy of the network on the 10000 test images using BN: %d %%' % accuracy_torch_adam)
```

```
[Torch Adam, 1,  2000] loss: 2.016
[Torch Adam, 1,  4000] loss: 1.849
[Torch Adam, 1,  6000] loss: 1.800
[Torch Adam, 1,  8000] loss: 1.755
[Torch Adam, 1, 10000] loss: 1.732
[Torch Adam, 1, 12000] loss: 1.695
[Torch Adam, 2,  2000] loss: 1.630
[Torch Adam, 2,  4000] loss: 1.629
[Torch Adam, 2,  6000] loss: 1.639
[Torch Adam, 2,  8000] loss: 1.603
[Torch Adam, 2, 10000] loss: 1.598
[Torch Adam, 2, 12000] loss: 1.593
Finished Training with BN
Accuracy of the network on the 10000 test images using BN: 53 %
```

4.2 Weight Initialization

Kaiming Normal Initialization

Kaiming Normal, also known as He Normal initialization, is a method for initializing the weights of neural network layers. Kaiming Normal initialization is primarily used for initializing the weights of layers in deep neural networks, especially in convolutional neural networks (CNNs) and deep feedforward neural networks.

```
def init_weights_kaiming(m):
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        nn.init.kaiming_normal_(m.weight)
        m.bias.data.fill_(0.01)


def init_weights_xavier(m):
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)


model_kaiming = CNNWithBN()
model_kaiming.apply(init_weights_kaiming)
criterion = nn.CrossEntropyLoss()
optimizer_kaming = optim.Adam(model_kaiming.parameters(), lr=0.001)



for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_kaming.zero_grad()
        outputs = model_kaiming(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_kaming.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[kaiming_normal, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with kaiming_normal')
model_kaiming.eval()
accuracy_torch_adam = evaluate_model(model_kaiming, testloader)
print('Accuracy of the network on the 10000 test images using kaiming_normal: %d %%' % accuracy_torch_adam)
```

```
[kaiming_normal, 1,  2000] loss: 2.097
[kaiming_normal, 1,  4000] loss: 1.898
```

```
[kaiming_normal, 1,  6000] loss: 1.824
[kaiming_normal, 1,  8000] loss: 1.761
[kaiming_normal, 1, 10000] loss: 1.756
[kaiming_normal, 1, 12000] loss: 1.699
[kaiming_normal, 2,  2000] loss: 1.685
[kaiming_normal, 2,  4000] loss: 1.656
[kaiming_normal, 2,  6000] loss: 1.626
[kaiming_normal, 2,  8000] loss: 1.611
[kaiming_normal, 2, 10000] loss: 1.595
[kaiming_normal, 2, 12000] loss: 1.587
Finished Training with kaiming_normal
Accuracy of the network on the 10000 test images using kaiming_normal: 52 %
```

Xavier Uniform Initialization

Xavier Uniform Initialization, also known as Glorot Initialization, is a technique used to initialize the weights of neural networks, particularly deep neural networks. It is named after Xavier Glorot, one of its creators.

```
model_xavier = CNNWithBN()
model_xavier.apply(init_weights_xavier)
criterion = nn.CrossEntropyLoss()
optimizer_xavier = optim.Adam(model_xavier.parameters(), lr=0.001)



for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_xavier.zero_grad()
        outputs = model_xavier(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_xavier.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[xavier_uniform, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with xavier_uniform')
model_xavier.eval()
accuracy_torch_adam = evaluate_model(model_xavier, testloader)
print('Accuracy of the network on the 10000 test images using xavier_uniform: %d %%' % accuracy_torch_adam)
```

```
[xavier_uniform, 1,  2000] loss: 2.131
[xavier_uniform, 1,  4000] loss: 1.908
[xavier_uniform, 1,  6000] loss: 1.839
[xavier_uniform, 1,  8000] loss: 1.813
[xavier_uniform, 1, 10000] loss: 1.748
[xavier_uniform, 1, 12000] loss: 1.714
[xavier_uniform, 2,  2000] loss: 1.666
[xavier_uniform, 2,  4000] loss: 1.658
[xavier_uniform, 2,  6000] loss: 1.630
[xavier_uniform, 2,  8000] loss: 1.625
[xavier_uniform, 2, 10000] loss: 1.594
[xavier_uniform, 2, 12000] loss: 1.596
Finished Training with xavier_uniform
Accuracy of the network on the 10000 test images using xavier_uniform: 52 %
```

4.3 Learning Rate Schedule

Learning rate scheduling is a deep learning technique that dynamically modifies the learning rate throughout neural network training. It aims to enhance training efficiency and prevent instability during the training process.

StepLR

The **StepLR** scheduler is a learning rate scheduler in PyTorch that drops the learning rate by a factor every few epochs.

```
from torch.optim import lr_scheduler
model_exp_lr = CNNWithBN()
optimizer_exp = optim.Adam(model_exp_lr.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
scheduler = lr_scheduler.StepLR(optimizer_exp, step_size=30, gamma=0.95)

for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_exp.zero_grad()
        outputs = model_exp_lr(inputs)
        loss = criterion(outputs, labels)
```

```
        loss.backward()
        optimizer_exp.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[StepLR, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with StepLR schedule')
model_exp_lr.eval()
accuracy_torch_adam = evaluate_model(model_exp_lr, testloader)
print('Accuracy of the network on the 10000 test images using StepLR: %d %%' % accuracy_torch_adam)
```

```
[StepLR, 1,  2000] loss: 1.999
[StepLR, 1,  4000] loss: 1.866
[StepLR, 1,  6000] loss: 1.809
[StepLR, 1,  8000] loss: 1.752
[StepLR, 1, 10000] loss: 1.729
[StepLR, 1, 12000] loss: 1.700
[StepLR, 2,  2000] loss: 1.662
[StepLR, 2,  4000] loss: 1.650
[StepLR, 2,  6000] loss: 1.608
[StepLR, 2,  8000] loss: 1.625
[StepLR, 2, 10000] loss: 1.600
[StepLR, 2, 12000] loss: 1.578
Finished Training with StepLR schedule
Accuracy of the network on the 10000 test images using StepLR: 55 %
```

ExponentialLR

The **ExponentialLR** scheduler is a learning rate scheduler in PyTorch that reduces the learning rate exponentially.

```
from torch.optim import lr_scheduler
model_exp_lr = CNNWithBN()
optimizer_exp = optim.Adam(model_exp_lr.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
scheduler_exp = lr_scheduler.ExponentialLR(optimizer_exp, gamma=0.95)

for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer_exp.zero_grad()
        outputs = model_exp_lr(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_exp.step()

        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[ExponentialLR, %d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training with ExponentialLR schedule')
model_exp_lr.eval()
accuracy_torch_adam = evaluate_model(model_exp_lr, testloader)
print('Accuracy of the network on the 10000 test images using ExponentialLR: %d %%' % accuracy_torch_adam)
```

```
[ExponentialLR, 1,  2000] loss: 1.994
[ExponentialLR, 1,  4000] loss: 1.832
[ExponentialLR, 1,  6000] loss: 1.813
[ExponentialLR, 1,  8000] loss: 1.760
[ExponentialLR, 1, 10000] loss: 1.710
[ExponentialLR, 1, 12000] loss: 1.709
[ExponentialLR, 2,  2000] loss: 1.652
[ExponentialLR, 2,  4000] loss: 1.634
[ExponentialLR, 2,  6000] loss: 1.629
[ExponentialLR, 2,  8000] loss: 1.625
[ExponentialLR, 2, 10000] loss: 1.606
[ExponentialLR, 2, 12000] loss: 1.588
Finished Training with ExponentialLR schedule
Accuracy of the network on the 10000 test images using ExponentialLR: 52 %
```