```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import CIFAR100
from torchvision.utils import make_grid
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split, ConcatDataset
import torchvision.transforms as tt
```

```
stats = ((0.5074,0.4867,0.4411),(0.2011,0.1987,0.2025))
train_transform = tt.Compose([
    tt.RandomHorizontalFlip(),
    tt.RandomCrop(32, padding=4, padding_mode="reflect"),
    tt.ToTensor(),
    tt.Normalize(*stats)
])

test_transform = tt.Compose([
    tt.ToTensor(),
    tt.Normalize(*stats)
])
```

```
train_data = CIFAR100(download=True, root="./data", transform=train_transform)
test_data = CIFAR100(root="./data", train=False, transform=test_transform)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz to ./data/cifar-100-python.tar.gz
100%|██████████| 169001437/169001437 [00:04<00:00, 41462449.40it/s]
Extracting ./data/cifar-100-python.tar.gz to ./data
```

```
for image, label in train_data:
    print("Image shape: ",image.shape)
    print("Image tensor: ", image)
    print("Label: ", label)
    break
```

Show hidden output

```
train_classes_items = dict()

for train_item in train_data:
    label = train_data.classes[train_item[1]]
    if label not in train_classes_items:
        train_classes_items[label] = 1
    else:
        train_classes_items[label] += 1

train_classes_items
```

Show hidden output

```
test_classes_items = dict()
for test_item in test_data:
    label = test_data.classes[test_item[1]]
    if label not in test_classes_items:
        test_classes_items[label] = 1
    else:
        test_classes_items[label] += 1

test_classes_items
```

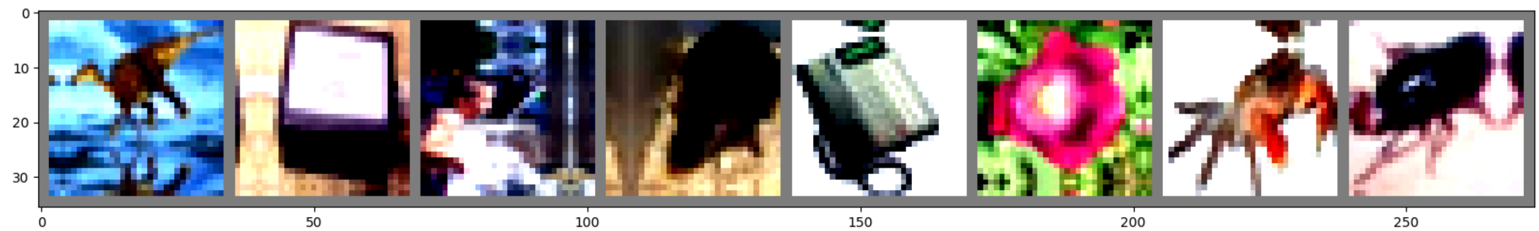
Show hidden output

```
BATCH_SIZE = 128
train_dl = DataLoader(train_data, BATCH_SIZE, num_workers=4, pin_memory=True, shuffle=True)
test_dl = DataLoader(test_data, BATCH_SIZE, num_workers=4, pin_memory=True)
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 worker processes in total. Our s
  warnings.warn(_create_warning_msg(
```

```
# for 8 images
train_8_samples = DataLoader(train_data, 8, num_workers=4, pin_memory=True, shuffle=True)

def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.figure(figsize = (20,20))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

dataiter = iter(train_8_samples)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print(''.join(f'{train_data.classes[labels[j]]:20s}' for j in range(8)))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
dinosaur            television          boy                 mouse               telephone           poppy               crab                cockroach
```

```
def get_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    return torch.device("cpu")

def to_device(data,device):
    if isinstance(data,(list,tuple)):
        return [to_device(x,device) for x in data]
    return data.to(device,non_blocking=True)


class ToDeviceLoader:
    def __init__(self,data,device):
        self.data = data
        self.device = device

    def __iter__(self):
        for batch in self.data:
            yield to_device(batch,self.device)

    def __len__(self):
        return len(self.data)
```

```
device = get_device()
print(device)

train_dl = ToDeviceLoader(train_dl, device)
test_dl = ToDeviceLoader(test_dl, device)
```

```
cuda
```

```
def accuracy(predicted, actual):
    _, predictions = torch.max(predicted, dim=1)
    return torch.tensor(torch.sum(predictions==actual).item()/len(predictions))
```

```
class BaseModel(nn.Module):
    def training_step(self,batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        return loss
```

```
    def validation_step(self,batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        acc = accuracy(out,labels)
        return {"val_loss":loss.detach(),"val_acc":acc}

    def validation_epoch_end(self,outputs):
        batch_losses = [loss["val_loss"] for loss in outputs]
        loss = torch.stack(batch_losses).mean()
        batch_accuracy = [accuracy["val_acc"] for accuracy in outputs]
        acc = torch.stack(batch_accuracy).mean()
        return {"val_loss":loss.item(),"val_acc":acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}".format(
            epoch, result['lrs'][-1], result['train_loss'], result['val_loss'], result['val_acc']))
```

ResNet Implementation

```
def conv_shortcut(in_channel, out_channel, stride):
    layers = [nn.Conv2d(in_channel, out_channel, kernel_size=(1,1), stride=(stride, stride)),
             nn.BatchNorm2d(out_channel)]
    return nn.Sequential(*layers)

def block(in_channel, out_channel, k_size,stride, conv=False):
    layers = None

    first_layers = [nn.Conv2d(in_channel,out_channel[0], kernel_size=(1,1),stride=(1,1)),
                    nn.BatchNorm2d(out_channel[0]),
                    nn.ReLU(inplace=True)]
    if conv:
        first_layers[0].stride=(stride,stride)

    second_layers = [nn.Conv2d(out_channel[0], out_channel[1], kernel_size=(k_size, k_size), stride=(1,1), padding=1),
                    nn.BatchNorm2d(out_channel[1])]

    layers = first_layers + second_layers
    return nn.Sequential(*layers)


class ResNet(BaseModel):

    def __init__(self, in_channels, num_classes):
        super().__init__()

        self.stg1 = nn.Sequential(
                                   nn.Conv2d(in_channels=in_channels, out_channels=64, kernel_size=(3),
                                             stride=(1), padding=1),
                                   nn.BatchNorm2d(64),
                                   nn.ReLU(inplace=True),
                                   nn.MaxPool2d(kernel_size=3, stride=2))
        ##stage 2
        self.convShortcut2 = conv_shortcut(64,256,1)

        self.conv2 = block(64,[64,256],3,1,conv=True)
        self.ident2 = block(256,[64,256],3,1)


        ##stage 3
        self.convShortcut3 = conv_shortcut(256,512,2)

        self.conv3 = block(256,[128,512],3,2,conv=True)
        self.ident3 = block(512,[128,512],3,2)


        ##stage 4
        self.convShortcut4 = conv_shortcut(512,1024,2)

        self.conv4 = block(512,[256,1024],3,2,conv=True)
        self.ident4 = block(1024,[256,1024],3,2)
        ##Classify
        self.classifier = nn.Sequential(
```

```
                                        nn.AvgPool2d(kernel_size=(4)),
                                        nn.Flatten(),
                                        nn.Linear(1024, num_classes))

    def forward(self,inputs):
        out = self.stg1(inputs)

        #stage 2
        out = F.relu(self.conv2(out) + self.convShortcut2(out))
        out = F.relu(self.ident2(out) + out)
        out = F.relu(self.ident2(out) + out)
        out = F.relu(self.ident2(out) + out)

        #stage3
        out = F.relu(self.conv3(out) + (self.convShortcut3(out)))
        out = F.relu(self.ident3(out) + out)
        out = F.relu(self.ident3(out) + out)
        out = F.relu(self.ident3(out) + out)
        out = F.relu(self.ident3(out) + out)
          #stage4
        out = F.relu(self.conv4(out) + (self.convShortcut4(out)))
        out = F.relu(self.ident4(out) + out)
        out = F.relu(self.ident4(out) + out)
        out = F.relu(self.ident4(out) + out)
        out = F.relu(self.ident4(out) + out)
        out = F.relu(self.ident4(out) + out)
        out = F.relu(self.ident4(out) + out)

        #Classify
        out = self.classifier(out)#100x1024

        return out
```

```
model = ResNet(3,100)
model = to_device(model, device)
```

```
@torch.no_grad()
def evaluate(model,test_dl):
    model.eval()
    outputs = [model.validation_step(batch) for batch in test_dl]
    return model.validation_epoch_end(outputs)
```

```
def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']

def fit (epochs, train_dl, test_dl, model, optimizer, max_lr, weight_decay, scheduler, grad_clip=None):
    torch.cuda.empty_cache()

    history = []

    optimizer = optimizer(model.parameters(), max_lr, weight_decay = weight_decay)

    scheduler = scheduler(optimizer, max_lr, epochs=epochs, steps_per_epoch=len(train_dl))

    for epoch in range(epochs):
        model.train()

        train_loss = []

        lrs = []
        for batch in train_dl:
            loss = model.training_step(batch)

            train_loss.append(loss)

            loss.backward()

            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
```

```
        optimizer.zero_grad()

        scheduler.step()
        lrs.append(get_lr(optimizer))
    result = evaluate(model, test_dl)
    result["train_loss"] = torch.stack(train_loss).mean().item()
    result["lrs"] = lrs

    model.epoch_end(epoch,result)
    history.append(result)

  return history
```

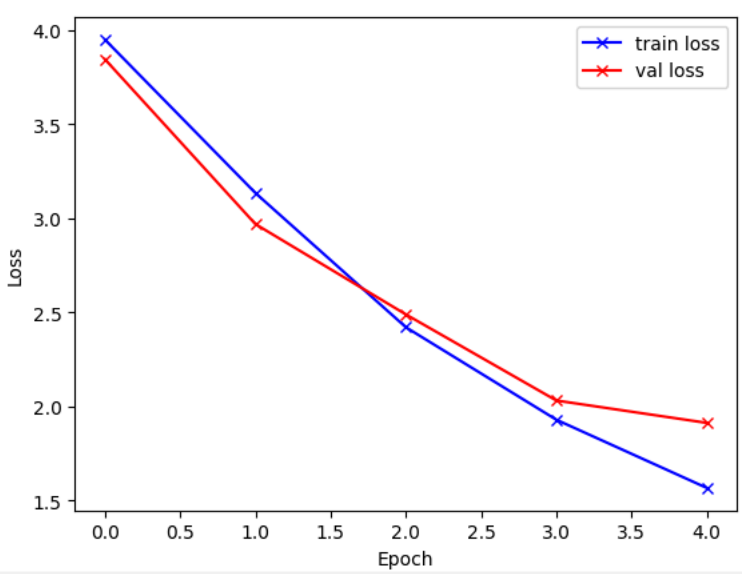
```
epochs = 5
optimizer = torch.optim.Adam
max_lr = 1e-3
grad_clip = 0.1
weight_decay = 1e-5
scheduler = torch.optim.lr_scheduler.OneCycleLR
```

```
history = fit(epochs=epochs, train_dl=train_dl, test_dl=test_dl, model=model,
              optimizer=optimizer, max_lr=max_lr, grad_clip=grad_clip,
              weight_decay=weight_decay, scheduler=torch.optim.lr_scheduler.OneCycleLR)
```

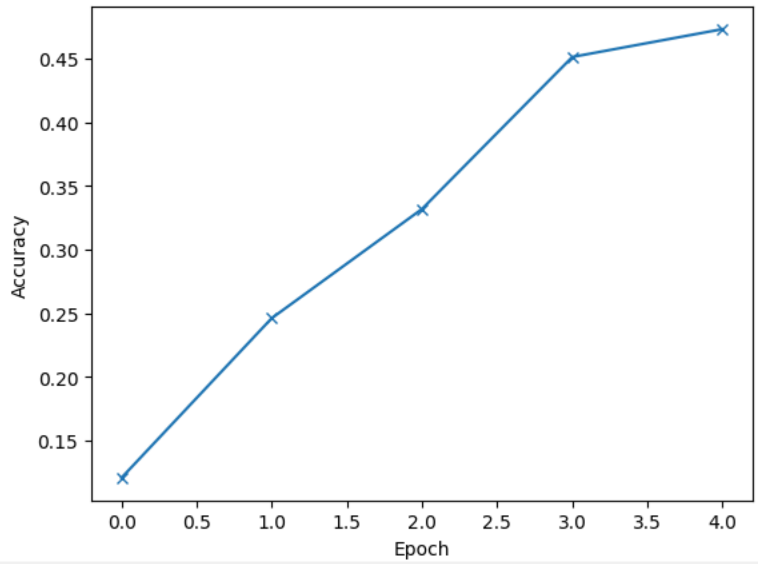
```
Epoch [0], last_lr: 0.00076, train_loss: 3.9503, val_loss: 3.8460, val_acc: 0.1208
Epoch [1], last_lr: 0.00095, train_loss: 3.1348, val_loss: 2.9695, val_acc: 0.2462
Epoch [2], last_lr: 0.00061, train_loss: 2.4209, val_loss: 2.4891, val_acc: 0.3320
Epoch [3], last_lr: 0.00019, train_loss: 1.9292, val_loss: 2.0315, val_acc: 0.4513
Epoch [4], last_lr: 0.00000, train_loss: 1.5666, val_loss: 1.9125, val_acc: 0.4732
```

```
def plot_acc(history):
    plt.plot([x["val_acc"] for x in history],"-x")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")

def plot_loss(history):
    plt.plot([x.get("train_loss") for x in history], "-bx")
    plt.plot([x["val_loss"] for x in history],"-rx")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(["train loss","val loss"])

def plot_lrs(history):
    plt.plot(np.concatenate([x.get("lrs",[]) for x in history]))
    plt.xlabel("Batch number")
    plt.ylabel("Learning rate")
```

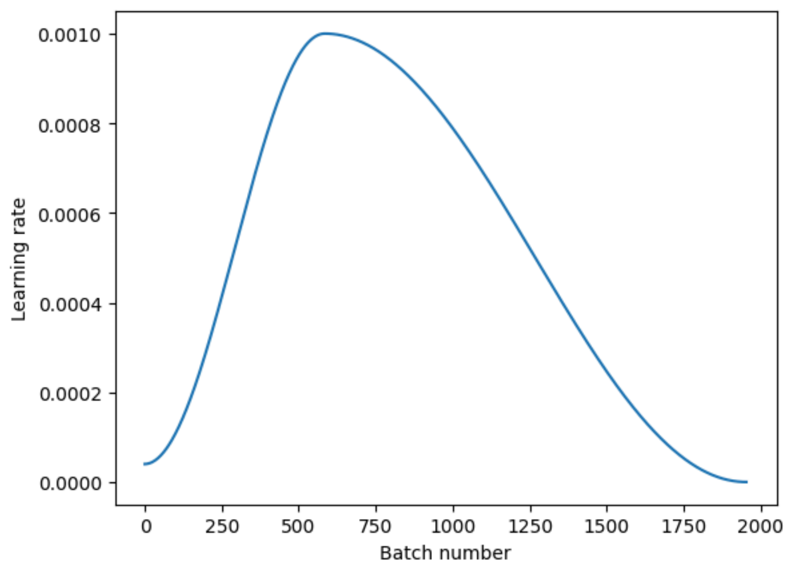
```
plot_loss(history)
```



```
plot_acc(history)
```



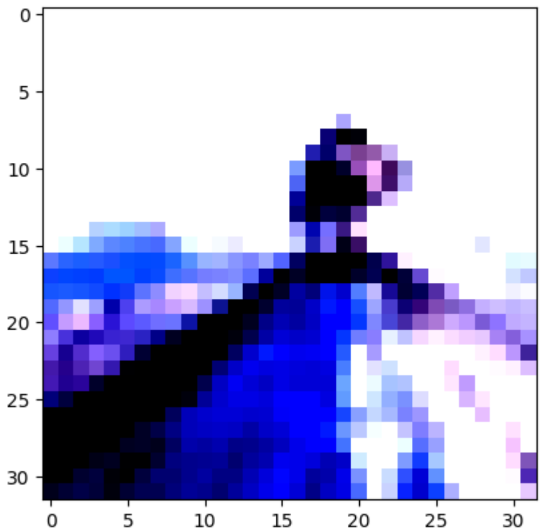

```
plot_lrs(history)
```




```
def predict_image(img, model):
    xb = to_device(img.unsqueeze(0), device)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return test_data.classes[preds[0].item()]
```

```
img, label = test_data[0]
plt.imshow(img.permute(1, 2, 0))
print('Label:', test_data.classes[label], ', Predicted:', predict_image(img, model))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Label: mountain , Predicted: dolphin
```



RESNET

```
import keras
from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np
```

```
from keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
import os
from os import listdir
from PIL import Image as PImage
img_width, img_height = 224, 224
```

```
model_pretrained = ResNet50(weights='imagenet',
                      include_top=True,
                      input_shape=(img_height, img_width, 3))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels.h5
102967424/102967424 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

```
# Insert correct path of your image below
img_path = '/content/panda.jpg'
img = image.load_img(img_path, target_size=(img_width, img_height))
img_data = image.img_to_array(img)
img_data = np.expand_dims(img_data, axis=0)
img_data = preprocess_input(img_data)
```

```
#predict the result
cnn_feature = model_pretrained.predict(img_data,verbose=0)
# decode the results into a list of tuples (class, description, probability)
label = decode_predictions(cnn_feature)
label = label[0][0]


plt.imshow(img)

stringprint ="%.1f" % round(label[2]*100,1)
plt.title(label[1] + " " + str(stringprint) + "%", fontsize=20)
plt.axis('off')
plt.show()
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
35363/35363 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

giant_panda 66.8%

```
import numpy as np
import os
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions

# Define the path to your images
folder_path = '/content/drive/MyDrive/food_dataset/Kaggle_train_images'

# List all image files in the directory
image_files = [f for f in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_path, f))]

# Example dimensions for the model
img_width, img_height = 224, 224

for img_name in image_files:
    img_path = os.path.join(folder_path, img_name)
    img = image.load_img(img_path, target_size=(img_width, img_height))
    img_data = np.expand_dims(image.img_to_array(img), axis=0)
    img_data = preprocess_input(img_data)

    # Predict the result
    cnn_feature = model_pretrained.predict(img_data, verbose=0)
    label = decode_predictions(cnn_feature, top=1)[0][0]

    # Display the result
    plt.imshow(img)
    plt.title(f"{label[1]} {label[2]*100:.1f}%", fontsize=20)
    plt.axis('off')
    plt.show()

    print(f"Predicted: {label[1]} with probability {label[2]*100:.1f}%")
```

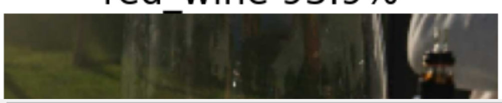
plate 18.0%

Predicted: plate with probability 18.0%

mashed_potato 18.6%

Predicted: mashed_potato with probability 18.6%

red_wine 93.9%



```
import keras
import tensorflow as tf

print("Keras version:", keras.__version__)
print("TensorFlow version:", tf.__version__)
```

```
Keras version: 3.4.1
TensorFlow version: 2.17.0
```

Imports:

Libraries like torch, torchvision, matplotlib, and sklearn are imported for deep learning, image processing, and plotting.

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Pre-trained model Resnet

```
import torch
from torchvision import models

model = models.resnet50(pretrained=True)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be r
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are d
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 156MB/s]
```

```
net = models.resnet50(pretrained=False)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are d
  warnings.warn(msg)
```

Data Transformations:

Data augmentation techniques like random horizontal flipping and random cropping are applied to the training images, while normalization is done for both training and testing images.

```
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

Data Loading:

CIFAR-10 dataset is downloaded and loaded into training and testing data loaders.

```
train = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)

trainloader = torch.utils.data.DataLoader(train, batch_size=128, shuffle=True, num_workers=2)

test = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)

testloader = torch.utils.data.DataLoader(test, batch_size=128,shuffle=False, num_workers=2)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:04<00:00, 41869849.45it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.1, momentum=0.9, weight_decay=0.0001)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, factor = 0.1, patience=5)
```

Training Loop:

The model is trained over a specified number of epochs. The loss for each batch is calculated, and the model weights are updated. The average loss for the epoch is calculated and printed.

```
# Ensure the model is on GPU
net = net.to('cuda')

EPOCHS = 50
for epoch in range(EPOCHS):
    losses = []
    running_loss = 0
    for i, inp in enumerate(trainloader):
        inputs, labels = inp
```

```
        # Move inputs and labels to GPU
        inputs, labels = inputs.to('cuda'), labels.to('cuda')

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        losses.append(loss.item())

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Print running loss for every 100 minibatches
        if i % 100 == 0 and i > 0:
            print(f'Loss [{epoch+1}, {i}](epoch, minibatch): ', running_loss / 100)
            running_loss = 0.0

    # Average loss for the epoch
    avg_loss = sum(losses) / len(losses)

    # Step the learning rate scheduler
    scheduler.step(avg_loss)

print('Training Done')
```

```
Loss [1, 100](epoch, minibatch):  1.3350788819789887
Loss [1, 200](epoch, minibatch):  1.3090847527980805
Loss [1, 300](epoch, minibatch):  1.289872134923935
Loss [2, 100](epoch, minibatch):  1.2822782766819
Loss [2, 200](epoch, minibatch):  1.2793977773189544
Loss [2, 300](epoch, minibatch):  1.2770849692821502
Loss [3, 100](epoch, minibatch):  1.2873174273967742
Loss [3, 200](epoch, minibatch):  1.248957382440567
Loss [3, 300](epoch, minibatch):  1.2455534541606903
Loss [4, 100](epoch, minibatch):  1.2149528098106384
Loss [4, 200](epoch, minibatch):  1.2156300485134124
Loss [4, 300](epoch, minibatch):  1.1947327744960785
Loss [5, 100](epoch, minibatch):  1.2155398970842362
Loss [5, 200](epoch, minibatch):  1.1809903544187546
Loss [5, 300](epoch, minibatch):  1.1732692480087281
Loss [6, 100](epoch, minibatch):  1.174017487168312
Loss [6, 200](epoch, minibatch):  1.1913472670316696
Loss [6, 300](epoch, minibatch):  1.1354758673906327
Loss [7, 100](epoch, minibatch):  1.150130045413971
Loss [7, 200](epoch, minibatch):  1.1594330888986588
Loss [7, 300](epoch, minibatch):  1.1175240421295165
Loss [8, 100](epoch, minibatch):  1.0878787249326707
Loss [8, 200](epoch, minibatch):  1.0819730752706527
Loss [8, 300](epoch, minibatch):  1.0645360666513444
Loss [9, 100](epoch, minibatch):  1.045373850464821
Loss [9, 200](epoch, minibatch):  1.0340056937932969
Loss [9, 300](epoch, minibatch):  1.050294045805931
Loss [10, 100](epoch, minibatch):  1.012904617190361
Loss [10, 200](epoch, minibatch):  1.0367707347869872
Loss [10, 300](epoch, minibatch):  1.0187771952152251
Loss [11, 100](epoch, minibatch):  0.9831012964248658
Loss [11, 200](epoch, minibatch):  0.9612731003761291
Loss [11, 300](epoch, minibatch):  0.9592066687345505
Loss [12, 100](epoch, minibatch):  1.1149044227600098
Loss [12, 200](epoch, minibatch):  1.0306628793478012
Loss [12, 300](epoch, minibatch):  0.9787292093038559
Loss [13, 100](epoch, minibatch):  0.9724466270208358
Loss [13, 200](epoch, minibatch):  0.9406642258167267
Loss [13, 300](epoch, minibatch):  0.950731395483017
Loss [14, 100](epoch, minibatch):  0.9513772386312485
Loss [14, 200](epoch, minibatch):  0.9270699924230575
Loss [14, 300](epoch, minibatch):  0.9224846857786179
Loss [15, 100](epoch, minibatch):  0.8728020107746124
Loss [15, 200](epoch, minibatch):  0.8984036475419999
Loss [15, 300](epoch, minibatch):  0.8613464796543121
Loss [16, 100](epoch, minibatch):  0.9428083449602127
Loss [16, 200](epoch, minibatch):  0.8766898357868195
Loss [16, 300](epoch, minibatch):  0.8727684211730957
Loss [17, 100](epoch, minibatch):  0.8499965167045593
```