4

import torch, torchvision

import torch.nn as nn

from torch.utils.data import DataLoader

# Dataset

t = torchvision.transforms.ToTensor()

train = DataLoader(torchvision.datasets.MNIST('./data', train=True, download=True, transform=t), batch\_size=64, shuffle=True)

test = DataLoader(torchvision.datasets.MNIST('./data', train=False, download=True, transform=t), batch\_size=64)

# Model

class Net(nn.Module):

    def \_\_init\_\_(self, bn=False, do=False):

        super().\_\_init\_\_()

        self.fc1 = nn.Linear(784, 256)

        self.bn = nn.BatchNorm1d(256) if bn else nn.Identity()

        self.do = nn.Dropout(0.5) if do else nn.Identity()

        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):

        x = torch.relu(self.bn(self.fc1(x.view(-1, 784))))

        return self.fc2(self.do(x))

# Train & Evaluate

def run(model):

    opt = torch.optim.Adam(model.parameters(), lr=1e-3)

    loss\_fn = nn.CrossEntropyLoss()

    for \_ in range(5):

        for x, y in train:

            opt.zero\_grad()

            loss\_fn(model(x), y).backward()

            opt.step()

    correct = sum((model(x).argmax(1) == y).sum().item() for x, y in test)

    return 100 \* correct / len(test.dataset)

# Results

print(f"No BN/Dropout: {run(Net()):.2f}%")

print(f"With BatchNorm: {run(Net(bn=True)):.2f}%")

print(f"With Dropout: {run(Net(do=True)):.2f}%")

6

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

import numpy as np

# Load and preprocess MNIST

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

x\_train, x\_test = x\_train[..., None] / 255.0, x\_test[..., None] / 255.0

# Simulate blurry inputs

def blur\_images(images):

    return tf.image.resize(tf.image.resize(images, (14, 14)), (28, 28))

x\_train\_blur = blur\_images(x\_train)

x\_test\_blur = blur\_images(x\_test)

# Simple U-Net model

def simple\_unet(input\_shape):

    inputs = layers.Input(shape=input\_shape)

    x = layers.Conv2D(32, 3, activation='relu', padding='same')(inputs)

    x = layers.MaxPooling2D()(x)

    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)

    x = layers.Conv2DTranspose(32, 3, strides=2, padding='same', activation='relu')(x)

    outputs = layers.Conv2D(1, 1, activation='sigmoid')(x)

    return models.Model(inputs, outputs)

# Compile and train

model = simple\_unet((28, 28, 1))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

model.fit(x\_train\_blur, x\_train, epochs=5, batch\_size=64, validation\_split=0.1)

# Predict on test samples

predicted = model.predict(x\_test\_blur[:5])

# Visualization

for i in range(5):

    plt.figure(figsize=(8, 2))

    for j, (title, img) in enumerate(zip(

        ["Blurry Input", "Ground Truth", "Predicted Mask"],

        [x\_test\_blur[i], x\_test[i], predicted[i]]

    )):

        plt.subplot(1, 3, j + 1)

        plt.title(title)

        plt.imshow(np.squeeze(img), cmap='gray')

        plt.axis('off')

    plt.show()

9

import torch, torch.nn as nn, torch.optim as optim

# Simple RNN model for binary classification

class SimpleRNN(nn.Module):

    def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

        super().\_\_init\_\_()

        self.rnn = nn.RNN(input\_size, hidden\_size, batch\_first=True)

        self.fc = nn.Linear(hidden\_size, output\_size)

    def forward(self, x):

        out, \_ = self.rnn(x)       # (batch, seq\_len, hidden\_size)

        return self.fc(out[:, -1]) # Use last time step's output

# Hyperparameters

input\_size, hidden\_size, output\_size = 8, 32, 1

seq\_len, batch\_size = 10, 16

# Random input and target

X = torch.randn(batch\_size, seq\_len, input\_size)

y = torch.randint(0, 2, (batch\_size, 1)).float()

# Model, loss, optimizer

model = SimpleRNN(input\_size, hidden\_size, output\_size)

criterion = nn.BCEWithLogitsLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop

for epoch in range(10):

    outputs = model(X)

    loss = criterion(outputs, y)

    optimizer.zero\_grad()

    loss.backward()

    optimizer.step()

    print(f"Epoch [{epoch+1}/10] Loss: {loss.item():.4f}")

11

import torch

import torch.nn as nn

class MultiHeadAttention(nn.Module):

    def \_\_init\_\_(self, embed\_size, heads):

        super().\_\_init\_\_()

        assert embed\_size % heads == 0

        self.head\_dim = embed\_size // heads

        self.heads = heads

        self.embed\_size = embed\_size

        self.values = nn.Linear(embed\_size, embed\_size)

        self.keys = nn.Linear(embed\_size, embed\_size)

        self.queries = nn.Linear(embed\_size, embed\_size)

        self.fc\_out = nn.Linear(embed\_size, embed\_size)

    def forward(self, value, key, query, mask=None):

        N, seq\_len = query.shape[0], query.shape[1]

        def transform(x, proj):

            x = proj(x)

            return x.view(N, -1, self.heads, self.head\_dim).transpose(1, 2)

        V, K, Q = map(lambda x, p: transform(x, p), [value, key, query], [self.values, self.keys, self.queries])

        energy = (Q @ K.transpose(-2, -1)) / (self.head\_dim \*\* 0.5)

        if mask is not None:

            energy = energy.masked\_fill(mask == 0, float("-1e20"))

        attention = torch.softmax(energy, dim=-1)

        out = (attention @ V).transpose(1, 2).reshape(N, seq\_len, self.embed\_size)

        return self.fc\_out(out)

# Example usage

attention = MultiHeadAttention(embed\_size=128, heads=8)

x = torch.rand(2, 10, 128)  # (batch, seq\_len, embed\_dim)

output = attention(x, x, x)

print(output.shape)  # → torch.Size([2, 10, 128])