AWS Logo

Application of Deep Learning to Text and Image Data

Module 1, Lab 4: Introducing CNNs

In the previous labs, you used neural networks to predict the target field of a given dataset. You used a feed-forward neural network for a multiclass classification task using images as inputs.

Now you will use a convolutional neural network (CNN) that is specialized to extract useful information from images. You will train and evaluate this network on a dataset of handwritten digits, and you will try to predict a number that is represented in an image.

You will learn how to do the following:

- Build a CNN.
- Train a CNN.
- Test the performance of a CNN.

You will be presented with two kinds of exercises throughout the notebook: activities and challenges.

Activity	**Challenge**
No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.	Challenges are where you can practice your coding skills.

Index

MNIST dataset

The MNIST dataset is a large collection of handwritten digits. Each example contains a pixel map showing how a person wrote a digit. The images have been size-normalized and centered with fixed dimensions. The labels correspond to the digit in the image, ranging from 0 to 9. This is a multiclass classification task with 10 output classes.

MNIST Examples

First, download the MNIST dataset.

In [1]:
```
%%capture
# Install libraries
!pip install -U -q -r requirements.txt
```

In [2]:
```
# Import the library dependencies
import boto3
import os
import pandas as pd

%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import nn
import torchvision
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.optim import SGD
```

In [3]:
```
# Load the train data (it's included in the torchvision library)
train_data = torchvision.datasets.MNIST(
    root="data", train=True, transform=transforms.ToTensor(), download=True
)

# Load the test data (it's included in the torchvision library)
test_data = torchvision.datasets.MNIST(
    root="data", train=False, transform=transforms.ToTensor(), download=True
)

# Print the dimensions of the datasets
print(
    "Training data shape: {}. \nTest data shape: {}".format(
        list(train_data.data.shape), list(test_data.data.shape)
    )
)
```

```
Training data shape: [60000, 28, 28].
Test data shape: [10000, 28, 28]
```

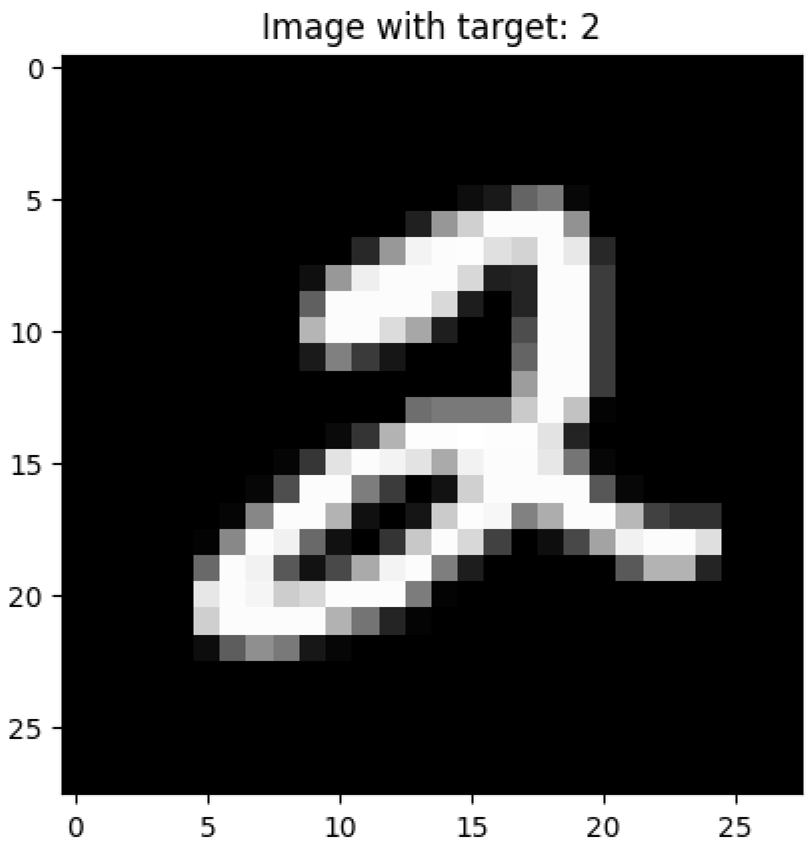
Try it yourself!

Activity

To observe a sample image from the MNIST dataset, run the following cell. The image is labeled with the digit that is present in the sample image.

In [4]:
```
# Show an example image
plt.imshow(train_data.data[5], cmap="gray")
plt.title("Image with target: %i" % train_data.targets[5])
```

Out[4]: `Text(0.5, 1.0, 'Image with target: 2')`



Creating a CNN

Convolutional neural networks (CNNs) are popular with image data. The network automatically extracts useful features from images, such as edges, contours, and objects.

This lab introduces CNNs, but the details of CNNs will be discussed in a later module.

CNNs require minimal preprocessing compared to older algorithms, such as feed-forward neural networks, that are used for computer vision. Although feed-forward neural networks can still be used with image data, CNNs can capture the spatial and temporal properties in an image with a significant reduction in the number of parameters. In this notebook, you will use a simple CNN to extract information from image data.

You will use PyTorch's Conv2D layer with the following interface to process the images:

`nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, ...)`

Parameter definitions:

- **in_channels (int):** Number of channels in the input image
- **out_channels (int):** Number of channels that are produced by the convolution
- **kernel_size (int or tuple):** Size of the convolving kernel
- **stride (int or tuple, optional):** Stride of the convolution (default is 1)

The output dimension of the Conv2D layer can be calculated using the following formula:

`((W - K + 2P)/S + 1)`

Where:

- W = Input size
- K = Kernel size
- S = Stride
- P = Padding (not used in the notebook)

Example:

For an `image of size = (28x28)`, `kernel size = 3`, `stride = 1`, and `padding = 0`, the output dimension is `(28 - 3 + 0)/1 + 1 = 26`.

With `out_channels = 1`, the output dimension is `(26, 26)`.

With `out_channels = 3`, the output dimension is `(26, 26, 3)`.

In [5]:
```
# Define hyperparameters
batch_size = 100  # Size of input data for one iteration
num_classes = 10  # Number of output classes, discrete range [0,9]
num_epochs = (
    10  # Number of times that the entire dataset is passed through the mode
)

# Size of step
lr = 1e-3

# Use GPU if available; otherwise, use CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Use PyTorch DataLoaders to load the data in batches
train_loader = torch.utils.data.DataLoader(
    dataset=train_data, batch_size=batch_size, shuffle=True, drop_last=True
```

```
)

# Repeat for test dataset
test_loader = torch.utils.data.DataLoader(
    dataset=test_data, batch_size=batch_size, shuffle=False
)
```

Try it yourself!

Challenge

Create a neural network with a 2D convolutional layer and the following attributes:

- Conv2D layer with `in_channel=1`, `out_channel=32`, and `kernel_size=3`
- Flatten the layer to squash the data into a one-dimensional tensor
- Linear layer with 128 units
- One output layer
- Softmax activation function for the output layer

In [9]:
```
input_size = 26 * 26 * 32  # Flattened dimension for the linear layer

############### CODE HERE ###############
import torch.nn.functional as F
# Define the CNN model class
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # Convolution layer: 1 input channel, 32 output channels, kernel siz
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3
        # Calculate output size after convolution to use in linear layer
        self.output_size_after_conv = (28 - 3) // 1 + 1  # For 28x28 input i
        # First fully connected layer: flattened size to 128 nodes
        self.fc1 = nn.Linear(32 * self.output_size_after_conv ** 2, 128)
        # Output layer: 128 input features to 10 output classes (digits 0-9)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        # Apply ReLU activation function after convolution
        x = F.relu(self.conv1(x))
        # Flatten the output of conv layers to feed into the linear layer
        x = x.view(-1, 32 * self.output_size_after_conv ** 2)
        # Apply ReLU activation function after first fully connected layer
        x = F.relu(self.fc1(x))
        # No activation function before softmax; PyTorch combines LogSoftmax
        x = self.fc2(x)
        # Return log probability; use LogSoftmax for numerical stability
        return F.log_softmax(x, dim=1)

# Create an instance of the CNN
net = CNN()
net = net.to(device)
############## END OF CODE ##############
```

```
def xavier_init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform_(m.weight)

# Initialize weights/parameters for the network
net.apply(xavier_init_weights)
```

Out[9]:
```
CNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=21632, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

In [10]:
```
# Define the loss function and the optimizer

# Choose cross-entropy loss for this classification problem
loss = nn.CrossEntropyLoss()

# Choose the Adam optimizer. You can also experiment with other optimizers.
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
```

Training the network

Now you are ready to train the CNN.

In [11]:
```
import time

# Network training and validation

# Start the outer epoch loop (epoch = full pass through the dataset)
for epoch in range(num_epochs):
    start = time.time()

    training_loss = 0.0

    # Training loop (with autograd and trainer steps)
    # This loop trains the neural network
    # Weights are updated here
    net.train()  # Activate training mode (dropouts and so on)
    for images, target in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()
        images = images.to(device)
        target = target.to(device)
        # Forward + backward + optimize
        output = net(images)
        L = loss(output, target)
        L.backward()
        optimizer.step()
        # Add batch loss
        training_loss += L.item()
```

```
    # Take the average losses
    training_loss = training_loss / len(train_loader)

    end = time.time()
    print("Epoch %s. Train_loss %f Seconds %f" % (epoch, training_loss, end
```

```
Epoch 0. Train_loss 0.255611 Seconds 25.233345
Epoch 1. Train_loss 0.074579 Seconds 6.989597
Epoch 2. Train_loss 0.044465 Seconds 6.990955
Epoch 3. Train_loss 0.030154 Seconds 6.997209
Epoch 4. Train_loss 0.020868 Seconds 7.002403
Epoch 5. Train_loss 0.014300 Seconds 6.998096
Epoch 6. Train_loss 0.009187 Seconds 7.004603
Epoch 7. Train_loss 0.006953 Seconds 6.989209
Epoch 8. Train_loss 0.006399 Seconds 6.980443
Epoch 9. Train_loss 0.005275 Seconds 6.980605
```

Testing the network

Finally, evaluate the performance of the trained network on the test set.

In [12]:
```
from sklearn.metrics import classification_report

net.eval()  # Activate eval mode (don't use dropouts and such)

# Get test predictions
predictions, labels = [], []
for images, target in test_loader:
    images = images.to(device)
    target = target.to(device)

    predictions.extend(net(images).argmax(axis=1).tolist())
    labels.extend(target.tolist())

# Print performance on the test data
print(classification_report(labels, predictions, zero_division=1))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.97      0.99      0.98      1032
           3       0.98      0.98      0.98      1010
           4       0.99      0.98      0.98       982
           5       0.98      0.98      0.98       892
           6       0.99      0.97      0.98       958
           7       0.99      0.98      0.98      1028
           8       0.98      0.97      0.98       974
           9       0.97      0.98      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

Conclusion

In this notebook, you practiced using a CNN.

Next Lab: Processing text

In the next lab you will learn how to do more advanced text processing.