California State University, Long Beach
Department of Computer Engineering and Computer Science
CECS 553 Sec 02 11792 (Machine Vision) – Fall 2022
Assignment 07 – Thursday, 11/03/2022

Digit Classification with Softmax

Objectives

- Download the Training and Validation MNIST Digit Images
- Create a Softmax Classifier using PyTorch
- Create a Criterion, Optimizer, and Data Loaders
- Create a Data Loader and set the Batch Size
- Train a Model
- Analyze Results and Model

Table of Contents

In this lab, you will use a single-layer Softmax Classifier to classify handwritten digits from the MNIST database.

Estimated Time Needed: **25 min**

Preparation

We'll need the following libraries

In [ ]:

```
# Import the libraries we need for this lab

# Using the following line code to install the torchvision library
# !conda install -y torchvision

# PyTorch Library
import torch
# PyTorch Neural Network
import torch.nn as nn
# Allows us to transform data
import torchvision.transforms as transforms
# Allows us to get the digit dataset
import torchvision.datasets as dsets
# Creating graphs
import matplotlib.pylab as plt
# Allows us to use arrays to manipulate and store data
import numpy as np
```

Use the following function to plot out the parameters of the Softmax function:

In [ ]:

```
# The function to plot parameters

def PlotParameters(model):
    W = model.state_dict()['linear.weight'].data
    w_min = W.min().item()
    w_max = W.max().item()
    fig, axes = plt.subplots(2, 5)
    fig.subplots_adjust(hspace=0.01, wspace=0.1)
    for i, ax in enumerate(axes.flat):
        if i < 10:

            # Set the label for the sub-plot.
            ax.set_xlabel("class: {0}".format(i))

            # Plot the image.
            ax.imshow(W[i, :].view(28, 28), vmin=w_min, vmax=w_max, cmap='se

            ax.set_xticks([])
            ax.set_yticks([])

        # Ensure the plot is shown correctly with multiple plots
        # in a single Notebook cell.
    plt.show()
```

Use the following function to visualize the data:

In [ ]:
```
# Plot the data

def show_data(data_sample):
    plt.imshow(data_sample[0].numpy().reshape(28, 28), cmap='gray')
    plt.title('y = ' + str(data_sample[1].item()))
```

Make Some Data

Load the *training* dataset by setting the parameters `train` to `True` and convert it to a tensor by placing a transform object in the argument `transform`.

In [ ]:
```
# Create and print the training dataset

train_dataset = dsets.MNIST(root='./data', train=True, download=True, transf
print("Print the training dataset:\n ", train_dataset)
```

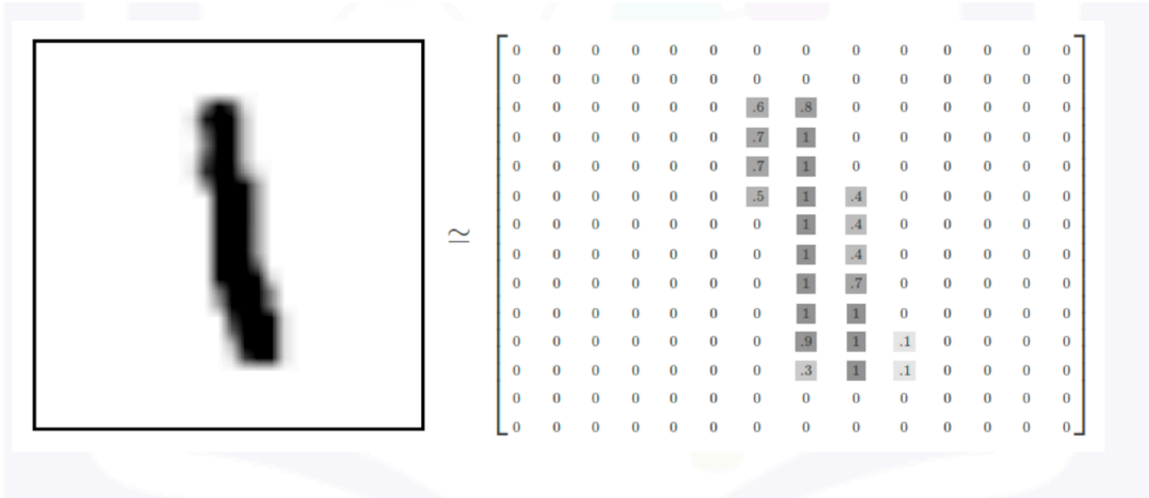
Load the *testing* dataset and convert it to a tensor by placing a transform object in the argument `transform`.

In [ ]:
```
# Create and print the validation dataset

validation_dataset = dsets.MNIST(root='./data', download=True, transform=tra
print("Print the validation dataset:\n ", validation_dataset)
```

We can access the data by indexing the train_dataset and test_dataset

In [ ]:
```
# Print the first image and label

print("First Image and Label", show_data(train_dataset[0]))
```

Each element in the rectangular tensor corresponds to a number which represents a pixel intensity, as demonstrated by the following image:



In this image, the values are inverted i.e black represents white.

Print out the label of the fourth element:

In [ ]:
```
# Print the label

print("The label: ", train_dataset[3][1])
```

The result shows the number in the image is 1

Plot the fourth sample:

In [ ]:
```
# Plot the image

print("The image: ", show_data(train_dataset[3]))
```

You see that it is a 1. Now, plot the third sample:

In [ ]:
```
# Plot the image

show_data(train_dataset[2])
```

Build a Softmax Classifer

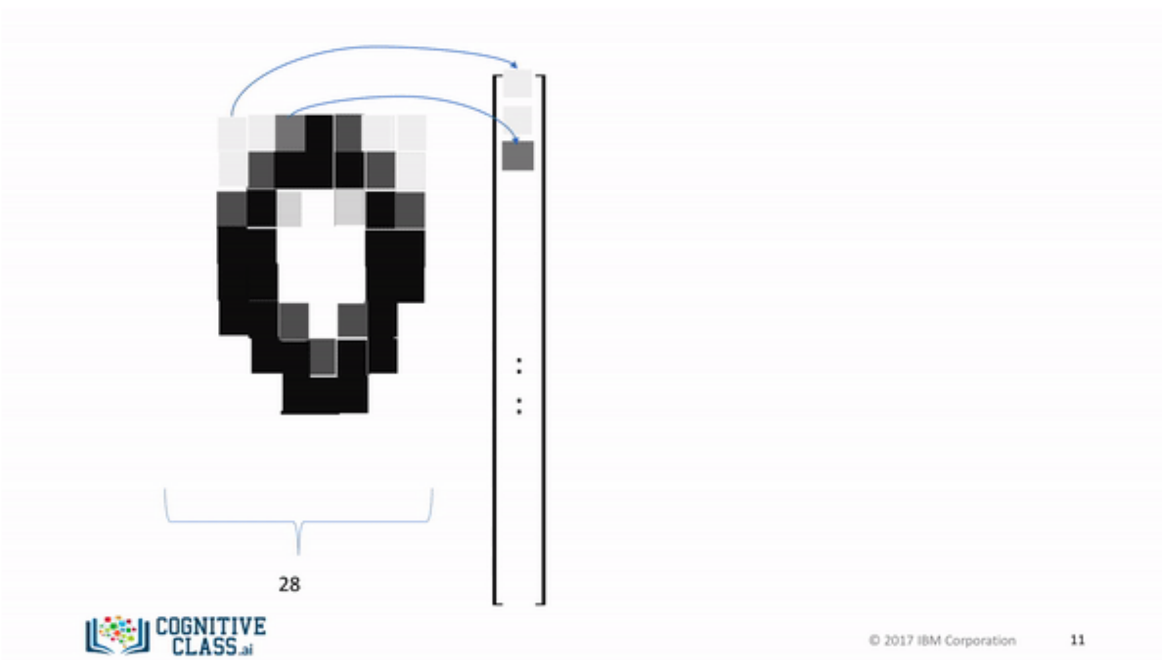
Build a Softmax classifier class:

In [ ]:
```python
# Define softmax classifier class
# Inherits nn.Module which is the base class for all neural networks
class SoftMax(nn.Module):

    # Constructor
    def __init__(self, input_size, output_size):
        super(SoftMax, self).__init__()
        # Creates a layer of given input size and output size
        self.linear = nn.Linear(input_size, output_size)

    # Prediction
    def forward(self, x):
        # Runs the x value through the single layers defined above
        z = self.linear(x)
        return z
```

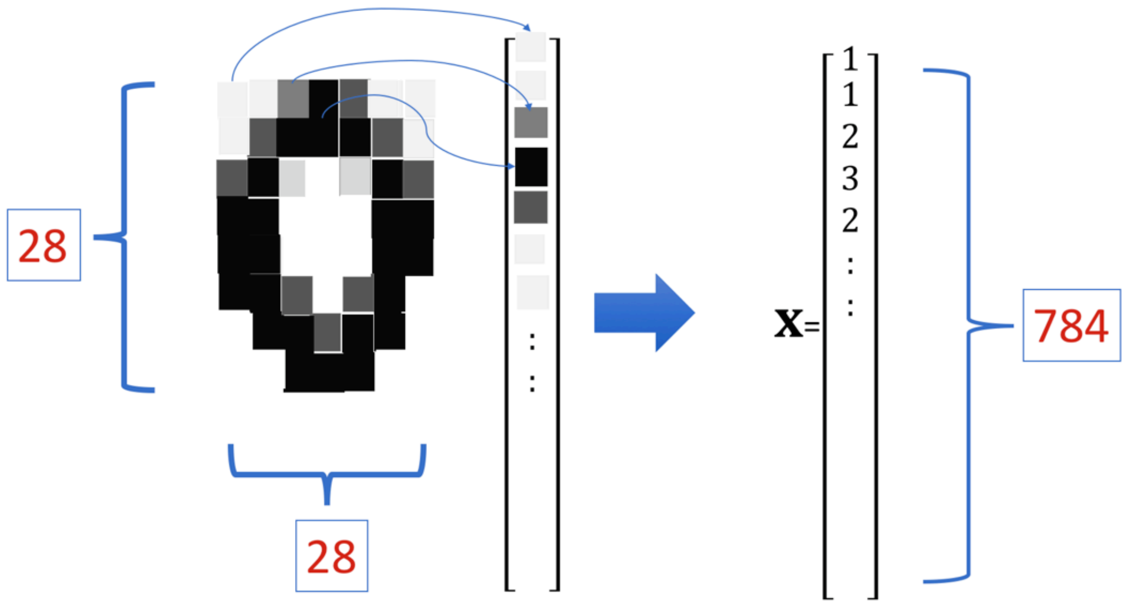
The Softmax function requires vector inputs. Note that the vector shape is 28x28.

In [ ]:
```python
# Print the shape of the training dataset

train_dataset[0][0].shape
```

Flatten the tensor as shown in this image:



The size of the tensor is now 784.



Set the input size and output size:

```
In [ ]: # Set input size and output size

input_dim = 28 * 28
output_dim = 10
```

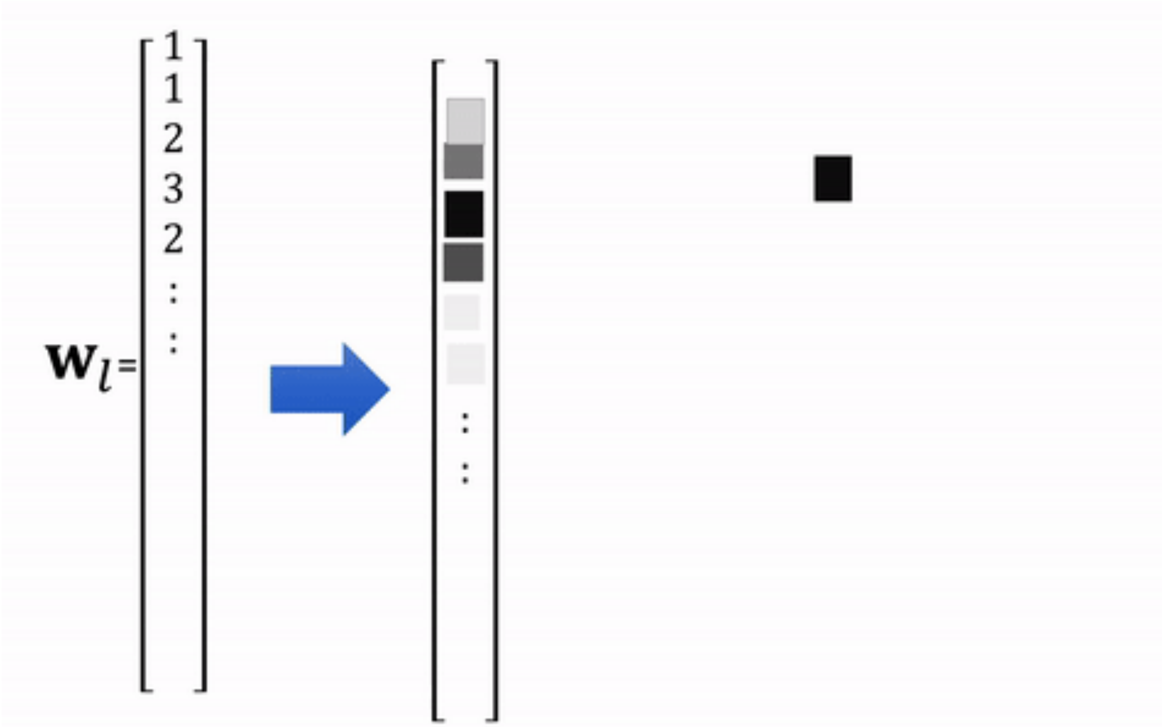
Define the Softmax Classifier, Criterion Function, Optimizer, and Train the Model

```
In [ ]: # Create the model
# Input dim is 28*28 which is the image converted to a tensor
# Output dim is 10 because there are 10 possible digits the image can be
model = SoftMax(input_dim, output_dim)
print("Print the model:\n ", model)
```

View the size of the model parameters:

```
In [ ]: # Print the parameters

print('W: ',list(model.parameters())[0].size())
print('b: ',list(model.parameters())[1].size())
```

You can convert the model parameters for each class to a rectangular grid:



Plot the model parameters for each class as a square image:

In [ ]:
```
# Plot the model parameters for each class
# Since the model has not been trained yet the parameters look random

PlotParameters(model)
```

We can make a prediction

In [ ]:
```
# First we get the X value of the first image
X = train_dataset[0][0]
# We can see the shape is 1 by 28 by 28, we need it to be flattened to 1 by
print(X.shape)
X = X.view(-1, 28*28)
print(X.shape)
# Now we can make a prediction, each class has a value, and the higher it is
model(X)
```

Define the learning rate, optimizer, criterion, data loader:

In [ ]:
```
# Define the learning rate, optimizer, criterion, and data loader

learning_rate = 0.1
# The optimizer will updates the model parameters using the learning rate
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
# The criterion will measure the loss between the prediction and actual labe
# This is where the SoftMax occurs, it is built into the Criterion Cross Ent
criterion = nn.CrossEntropyLoss()
# Created a training data loader so we can set the batch size
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size
# Created a validation data loader so we can set the batch size
validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset,
```

How Cross Entropy Loss uses SoftMax

We have X which is the X values of the first image and `actual` which is the the digit class the image belongs to. The output `model_output` is the value the model assigns to each class for that image.

In [ ]:
```
model_output = model(X)
actual = torch.tensor([train_dataset[0][1]])

show_data(train_dataset[0])
print("Output: ", model_output)
print("Actual:", actual)
```

The criterion will take these values and return a loss

In [ ]:
```
criterion(model_output, actual)
```

Cross Entropy Loss takes probabilities and we can see that `model_output` are not probabilities, this is where softmax comes in

In [ ]:
```
softmax = nn.Softmax(dim=1)
probability = softmax(model_output)
print(probability)
```

Now that we have probabilities, we can just calculate the negative log of the probability of the class that this image belongs to. The image belongs to the target class so we calculate the negative log of the probability at the target index.

In [ ]:
```
-1*torch.log(probability[0][actual])
```

As you can see the result above matches the result of the criterion, this is how Cross Entropy Loss uses Softmax.

Train

Train the model and determine validation accuracy **(should take a few minutes)**:

In [ ]:

```
# Number of times we train our model useing the training data
n_epochs = 10
# Lists to keep track of loss and accuracy
loss_list = []
accuracy_list = []
# Size of the validation data
N_test = len(validation_dataset)

# Function to train the model based on number of epochs
def train_model(n_epochs):
    # Loops n_epochs times
    for epoch in range(n_epochs):
        # For each batch in the train loader
        for x, y in train_loader:
            # Resets the calculated gradient value, this must be done each t
            optimizer.zero_grad()
            # Makes a prediction based on the image tensor
            z = model(x.view(-1, 28 * 28))
            # Calculates loss between the model output and actual class
            loss = criterion(z, y)
            # Calculates the gradient value with respect to each weight and
            loss.backward()
            # Updates the weight and bias according to calculated gradient v
            optimizer.step()

        # Each epoch we check how the model performs with data it has not se
        correct = 0
        # For each batch in the validation loader
        for x_test, y_test in validation_loader:
            # Makes prediction based on image tensor
            z = model(x_test.view(-1, 28 * 28))
            # Finds the class with the higest output
            _, yhat = torch.max(z.data, 1)
            # Checks if the prediction matches the actual class and incremen
            correct += (yhat == y_test).sum().item()
        # Calculates the accuracy by dividing correct by size of validation
        accuracy = correct / N_test
        # Keeps track loss
        loss_list.append(loss.data)
        # Keeps track of the accuracy
        accuracy_list.append(accuracy)

# Function call
train_model(n_epochs)
```

Analyze Results

Plot the loss and accuracy on the validation data:

```
In [ ]: # Plot the loss and accuracy

fig, ax1 = plt.subplots()
color = 'tab:red'
ax1.plot(loss_list,color=color)
ax1.set_xlabel('epoch',color=color)
ax1.set_ylabel('total loss',color=color)
ax1.tick_params(axis='y', color=color)

ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('accuracy', color=color)
ax2.plot( accuracy_list, color=color)
ax2.tick_params(axis='y', color=color)
fig.tight_layout()
```

View the results of the parameters for each class after the training. You can see that they look like the corresponding numbers.

```
In [ ]: # Plot the parameters

PlotParameters(model)
```

We Plot the first five misclassified samples and the probability of that class.

```
In [ ]: # Plot the misclassified samples
Softmax_fn=nn.Softmax(dim=-1)
count = 0
for x, y in validation_dataset:
    z = model(x.reshape(-1, 28 * 28))
    _, yhat = torch.max(z, 1)
    if yhat != y:
        show_data((x, y))
        plt.show()
        print("yhat:", yhat)
        print("probability of class ", torch.max(Softmax_fn(z)).item())
        count += 1
    if count >= 5:
        break
```

We plot the first five correctly classified samples and the probability of that class. We see the probability is much larger.

In [ ]:
```
# Plot the classified samples
Softmax_fn=nn.Softmax(dim=-1)
count = 0
for x, y in validation_dataset:
    z = model(x.reshape(-1, 28 * 28))
    _, yhat = torch.max(z, 1)
    if yhat == y:
        show_data((x, y))
        plt.show()
        print("yhat:", yhat)
        print("probability of class ", torch.max(Softmax_fn(z)).item())
        count += 1
    if count >= 5:
        break
```