MITx_686x_Project_2_DigitRecognition_P2

July 23, 2019

0.1 Problem 3: Neural Network Basics

Generates a neural network with the following architecture:

- Fully connected neural network.
- Input vector takes in two features.
- One hidden layer with three neurons whose activation function is ReLU.
- One output neuron whose activation function is the identity function.

0.2 1. Introduction

Your friends now want you to try implementing a neural network to classify MNIST digits.

Setup: As with the last project, please use Python's NumPy numerical library for handling arrays and array operations; use matplotlib for producing figures and plots.

1. Note on software: For all the projects, we will use python 3.6 augmented with the NumPy numerical toolbox, the matplotlib plotting toolbox. For THIS project, you will also be using PyTorch for implementing the Neural Nets and scipy to handle sparse matrices.
2. Download mnist.tar.gz and untar it in to a working directory. The archive contains the various data files in the Dataset directory, along with the following python files:

- part2-nn/neural_nets.py in which you'll implement your first neural net from scratch
- part2-mnist/nnet_fc.py where you'll start using PyTorch to classify MNIST digits
- part2-mnist/nnet_cnn.py where you will use convolutional layers to boost performance
- part2-twodigit/mlp.py and part2-twodigit/cnn.py which are for a new, more difficult version of the MNIST dataset

Tip: Throughout the whole online grading system, you can assume the NumPy python library is already imported as np. In some problems you will also have access to python's random library, and other functions you've already implemented. Look out for the "Available Functions" Tip before the codebox, as you did in the last project.

This project will unfold both on MITx and on your local machine. However, we encourage you to first implement the functions locally. For this project, there will not be a test.py script. You are encouraged to think of your own test cases to make sure your code works as you expected before submitting it to the online grader.

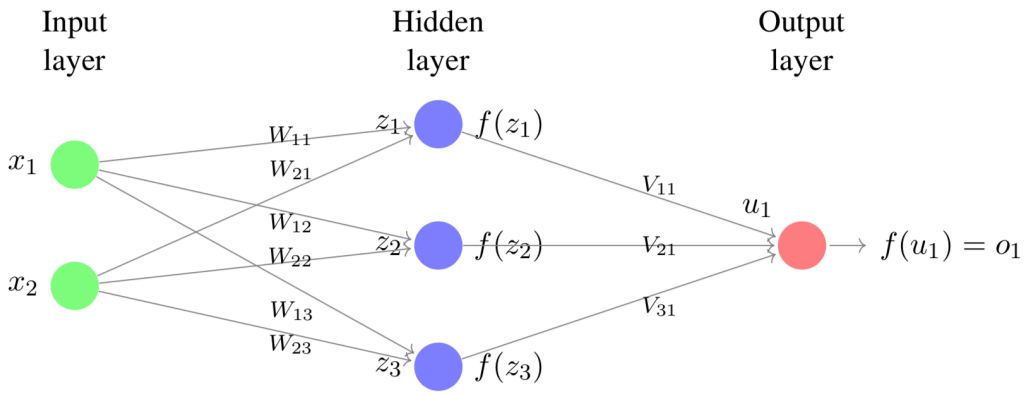
Tip: You may work through the Pytorch tutorial in Introduction to ML Packages (Part 2) (posted in the resource section)

0.3 2. Neural Network Basics

Good programmers can use neural nets. Great programmers can make them. This section will guide you through the implementation of a simple neural net with an architecture as shown in the figure below. You will implement the net from scratch (you will probably never do this again, don't worry) so that you later feel confident about using libraries. We provide some skeleton code in neural_nets.py for you to fill in.

[7]:
```
Image(filename='C:/root/workspace/MITx_6.86x/07_Projects/project3_part2/
↪images_neuralnet.png')
```

[7]:



1). Setup $x = [x_1, x_2]^T$

$z_1 = x_1 W_{11} + x_2 W_{21} + W_{01}$
$z_2 = x_1 W_{12} + x_2 W_{22} + W_{02}$
$z_3 = x_1 W_{13} + x_2 W_{23} + W_{03}$
$f(z_1) = max\{z_1, 0\}$
$f(z_2) = max\{z_2, 0\}$
$f(z_3) = max\{z_3, 0\}$
$u_1 = f(z_1) V_{11} + f(z_2) V_{21} + f(z_3) V_{31} + V_{01}$
$f(u_1) = u_1$
$o_1 = f(u_1)$

2). Error of neuron j in layer l

1. l = L: $\delta_j^L = \frac{\partial C}{\partial a_j^L} f'(z_j^L) = \frac{\partial C}{\partial a_j^L}$
2. l <> L: $\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f'(z_j^l) = \sum_k w_{kj}^{l+1} \delta_k^{l+1}$

3). parameter Derivatives

1. $\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$
2. $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

4). Example: 2 layer, y is the prediction and t is the target

- $z_1 = x_1 W_{11} + x_2 W_{21} + W_{01}$
- $f(z_1) = ReLU(z_1)$
- $u_1 = f(z_1) V_{11} + f(z_2) V_{21} + f(z_3) V_{31} + V_{01}$
- $y = f(u_1)$
- $C = \frac{1}{2}(y - t)^2$

5). SGD Next $w_1 = w_1 - \eta \frac{\partial C}{\partial w_1}$

0.4 3. Activation Functions

The first step is to design the activation function for each neuron. In this problem, we will initialize the network weights to 1, use ReLU for the activation function of the hidden layers, and use an identity function for the output neuron. The hidden layer has a bias but the output layer does not. Complete the helper functions in neural_networks.py, including rectified_linear_unit and rectified_linear_unit_derivative, for you to use in the NeuralNetwork class, and implement them below.

You will be working in the file part2-nn/neural_nets.py in this problem

0.4.1 Rectified Linear Unit

First implement the ReLu activation function, which computes the ReLu of a scalar.

Note: Your function does not need to handle a vectorized input

Available Functions: You have access to the NumPy python library as np

[ ]:
```
import numpy as np
import math
from IPython.display import Image
```

[1]:
```
def rectified_linear_unit(x):
    """ Returns the ReLU of x, or the maximum between 0 and x."""
    # TODO
    return max(0, x)
```

[3]:
```
rectified_linear_unit(-5)
```

[3]:
```
0
```

0.4.2 Taking the Derivative

Now implement its derivative so that we can properly run backpropagation when training the net. Note: we will consider the derivative at zero to have the same value as the derivative at all negative points.

Note: Your function does not need to handle a vectorized input

Available Functions: You have access to the NumPy python library as np

[10]:
```
def rectified_linear_unit_derivative(x):
    """ Returns the derivative of ReLU."""
    # TODO
    if x > 0:
```

```
        return 1
    else:
        return 0
```

[12]:
```
rectified_linear_unit_derivative(-5)
```

[12]:
```
0
```

0.5 4. Training the Network

Forward propagation is simply the summation of the previous layer's output multiplied by the weight of each wire, while back-propagation works by computing the partial derivatives of the cost function with respect to every weight or bias in the network. In back propagation, the network gets better at minimizing the error and predicting the output of the data being used for training by incrementally updating their weights and biases using stochastic gradient descent.

We are trying to estimate a continuous-valued function, thus we will use squared loss as our cost function and an identity function as the output activation function. f(x) is the activation function that is called on the input to our final layer output node, and $\hat{a}$ is the predicted value, while y is the actual value of the input.

$$C = \frac{1}{2}(y - \hat{a})^2$$

$$f(x) = x$$

When you're done implementing the function train (below and in your local repository), run the script and see if the errors are decreasing. If your errors are all under 0.15 after the last training iteration then you have implemented the neural network training correctly.

You'll notice that the train functin inherits from NeuralNetworkBase in the codebox below; this is done for grading purposes. In your local code, you implement the function directly in your Neural Network class all in one file. The rest of the code in NeuralNetworkBase is the same as in the original NeuralNetwork class you have locally.

In this problem, you will see the network weights are initialized to 1. This is a bad setting in practice, but we do so for simplicity and grading here.

You will be working in the file part2-nn/neural_nets.py in this problem

0.5.1 Implementing Train

Available Functions: You have access to the NumPy python library as np, rectified_linear_unit, output_layer_activation, rectified_linear_unit_derivative, and output_layer_activation_derivative

Note: Functions rectified_linear_unit_derivative, and output_layer_activation_derivative can only handle scalar input. You will need to use np.vectorize to use them

[13]:
```
def output_layer_activation(x):
    """ Linear function, returns input as is. """
    return x
```

[14]:
```
def output_layer_activation_derivative(x):
    """ Returns the derivative of a linear function: 1. """
    return 1
```

[283]:
```
class NeuralNetwork():
    """
        Contains the following functions:
            -train: tunes parameters of the neural network based on error␣
↪obtained from forward propagation.
            -predict: predicts the label of a feature vector based on the␣
↪class's parameters.
            -train_neural_network: trains a neural network over all the data␣
↪points for the specified number of epochs during initialization of the class.
            -test_neural_network: uses the parameters specified at the time in␣
↪order to test that the neural network classifies the points given in␣
↪testing_points within a margin of error.
    """

    def __init__(self):

        # DO NOT CHANGE PARAMETERS
        self.input_to_hidden_weights = np.matrix('1 1; 1 1; 1 1')
        self.hidden_to_output_weights = np.matrix('1 1 1')
        self.biases = np.matrix('0; 0; 0')
        self.learning_rate = .001
        self.epochs_to_train = 10
        self.training_points = [((2,1), 10), ((3,3), 21), ((4,5), 32), ((6, 6),␣
↪42)]
        self.testing_points = [(1,1), (2,2), (3,3), (5,5), (10,10)]

    def train(self, x1, x2, y):

        ### Forward propagation ###
        input_values = np.matrix([[x1],[x2]]) # 2 by 1

        # Calculate the input and activation of the hidden layer
        # hidden_layer_weighted_input = # TODO (3 by 1 matrix)
        # hidden_layer_activation = # TODO (3 by 1 matrix)
        # output =  # TODO

        vf_11 = np.vectorize(rectified_linear_unit)  # ReLU
        vf_21 = np.vectorize(output_layer_activation) # f(x) = x

        ## dot prodution of W and X
        hidden_layer_weighted_input = self.input_to_hidden_weights.
↪dot(input_values) + self.biases
        ## ReLU: rectified_linear_unit
        hidden_layer_activation = vf_11(hidden_layer_weighted_input)

        output = self.hidden_to_output_weights* hidden_layer_activation
        # activated_output = # TODO
```

```
        activated_output = vf_21(output)

        ### Backpropagation ###
        vf_12 = np.vectorize(rectified_linear_unit_derivative) # if >0 then 1␣
↪else 0
        vf_22 = np.vectorize(output_layer_activation_derivative)   # 1

        # Compute gradients
        # output_layer_error = # TODO
        # hidden_layer_error = # TODO (3 by 1 matrix)

        # output layer error = (y_hat -y)*f'(z2)
        output_layer_error = np.squeeze(np.
↪array((activated_output-y)*vf_22(output))) # TODO
        # hidden layer error = f'(z1)*W2*output layer error
        delta_fz = np.squeeze(np.array(vf_12(hidden_layer_weighted_input)))
        hidden_layer_error = np.asmatrix(delta_fz*np.squeeze(np.array(self.
↪hidden_to_output_weights))).T*output_layer_error # TODO (3 by 1 matrix)

        # bias_gradients = # TODO
        # hidden_to_output_weight_gradients = # TODO
        # input_to_hidden_weight_gradients = # TODO
        hidden_to_output_weight_gradients = np.squeeze(np.
↪array(output_layer_error))*hidden_layer_activation # TODO
        bias_gradients = hidden_layer_error # TODO
        input_to_hidden_weight_gradients = hidden_layer_error*input_values.T #␣
↪TODO

        # Use gradients to adjust weights and biases using gradient descent
        # self.biases = # TODO
        self.biases = self.biases - self.learning_rate* bias_gradients
        # self.input_to_hidden_weights = # TODO
        self.hidden_to_output_weights = self.hidden_to_output_weights - self.
↪learning_rate* hidden_to_output_weight_gradients.T # TODO
        # self.hidden_to_output_weights = # TODO
        self.input_to_hidden_weights = self.input_to_hidden_weights - self.
↪learning_rate* input_to_hidden_weight_gradients  # TODO

    def predict(self, x1, x2):

        input_values = np.matrix([[x1],[x2]])

        # Compute output for a single input(should be same as the forward␣
↪propagation in training)
        # hidden_layer_weighted_input = # TODO
        # hidden_layer_activation = # TODO
```

```
        # output = # TODO
        # activated_output = # TODO
        vf_11 = np.vectorize(rectified_linear_unit)  # ReLU
        vf_21 = np.vectorize(output_layer_activation) # f(x) = x

        hidden_layer_weighted_input = self.input_to_hidden_weights.
↪dot(input_values) + self.biases
        hidden_layer_activation = vf_11(hidden_layer_weighted_input)

        output = self.hidden_to_output_weights*hidden_layer_activation
        activated_output = vf_21(output)

        return activated_output.item()

    # Run this to train your neural network once you complete the train method
    def train_neural_network(self):

        for epoch in range(self.epochs_to_train):
            for x,y in self.training_points:
                self.train(x[0], x[1], y)

    # Run this to test your neural network implementation for correctness after␣
↪it is trained
    def test_neural_network(self):

        for point in self.testing_points:
            print("Point,", point, "Prediction,", self.predict(point[0],␣
↪point[1]))
            if abs(self.predict(point[0], point[1]) - 7*point[0]) < 0.1:
                print("Test Passed")
            else:
                print("Point ", point[0], point[1], " failed to be predicted␣
↪correctly.")
                return
```

[284]:
```
x = NeuralNetwork()
```

[285]:
```
x.train_neural_network()
```

[286]:
```
# UNCOMMENT THE LINE BELOW TO TEST YOUR NEURAL NETWORK
x.test_neural_network()
```

```
Point, (1, 1) Prediction, 7.038453196038151
Test Passed
Point, (2, 2) Prediction, 14.042814826755537
Test Passed
Point, (3, 3) Prediction, 21.047176457472922
Test Passed
```

```
Point, (5, 5) Prediction, 35.055899718907696
Test Passed
Point, (10, 10) Prediction, 70.07770787249461
Test Passed
```

0.6 5. Predicting the Test Data

Now fill in the code for the function predict, which will use your trained neural network in order to label new data.

You will be working in the file part2-nn/neural_nets.py in this problem

0.6.1 Implementing Predict

Available Functions: You have access to the NumPy python library as np, rectified_linear_unit and output_layer_activation

Note: Functions rectified_linear_unit_derivative, and output_layer_activation_derivative can only handle scalar input. You will need to use np.vectorize to use them

[ ]:
```
def predict(self, x1, x2):

        input_values = np.matrix([[x1],[x2]])

        vf_11 = np.vectorize(rectified_linear_unit)  # ReLU
        vf_21 = np.vectorize(output_layer_activation) # f(x) = x

        # Compute output for a single input(should be same as the forward␣
↪propagation in training)
        hidden_layer_weighted_input = self.input_to_hidden_weights.
↪dot(input_values) + self.biases
        hidden_layer_activation = vf_11(hidden_layer_weighted_input)
        output = self.hidden_to_output_weights*hidden_layer_activation
        activated_output = vf_21(output)

        return activated_output.item()
```

0.7 6. Conceptual Questions

0.7.1 Thinking about Network Size

What is the danger to having too many hidden units in your network? * It will take up more memory correct * It may overfit the training data correct * It will take longer to train correct

0.7.2 Training and Testing Accuracy Over Time

0.8 7. Classification for MNIST using deep neural networks

In this section, we are going to use deep neural networks to perform the same classification task as in previous sections. We will use PyTorch, a python deep learning framework. Using a framework

like PyTorch means you don't have to implement all of the details (like in the earlier problem) and can spend more time thinking through your high level architecture.

Setup Overview To setup PyTorch, navigate to their website in your browser, select your preferences and begin downloading. Your selection for OS and Package Manager will depend on your local setup. For example, if you are on a Mac and use pip as your Python package manager, select "OSX" and "Pip". We recommend you select Python version 3 for use with PyTorch. Finally, you are not required to train large models for this course, so you can safely select "None" for CUDA. If you have access to a NVIDIA GPU enabled device with the CUDA library installed, and want to try training your neural models on GPUs, feel free to install PyTorch with CUDA selected but you will have to troubleshoot on your own.

Test your installation Once you have successfully installed PyTorch using the instructions on their website, you should test your installation to ensure it is running properly before trying to complete the project. For basic functionality, you can start a python REPL environment with the python command in your terminal. Then try importing PyTorch with import torch.

0.9 8. Fully-Connected Neural Networks

First, we will employ the most basic form of a deep neural network, in which the neurons in adjacent layers are fully connected to one another.

You will be working in the filespart2-mnist/nnet_fc.pyin this problem

0.9.1 Training and Testing Accuracy Over Time

We have provided a toy example nnet_fc.py in which we have implemented for you a simple neural network. This network has one hidden layer of 10 neurons with a rectified linear unit (ReLU) nonlinearity, as well as an output layer of 10 neurons (one for each digit class). Finally, a softmax function normalizes the activations of the output neurons so that they specify a probability distribution. Reference the PyTorch Documentation and read through it in order to gain a better understanding of the code. Then, try running the code on your computer with the command python3 nnet_fc.py. This will train the network with 10 epochs, where an epoch is a complete pass through the training dataset. Total training time of your network should take no more than a couple of minutes. At the end of training, your model should have an accuracy of more than %85 on test data.

Report the test accuracy below.

Test Accuracy = 0.9204727564102564

0.9.2 Improving Accuracy

We would like to try to improve the performance of the model by performing a mini grid search over hyper parameters (note that a full grid search should include more values and combinations). To this end, we will use our baseline model (batch size 32, hidden size 10, learning rate 0.1, momentum 0 and the ReLU activation function) and modify one parameter each time while keeping all others to the baseline. We will use the validation accuracy of the model after training for 10 epochs. For the LeakyReLU activation function, use the default parameters from pyTorch (negative_slope=0.01).

Note: If you run the model multiple times from the same script, make sure to initialize the numpy and pytorch random seeds to 12321 before each run.

Which of the following modifications achieved the highest validation accuracy? * baseline (no modifications) * batch size 64 correct * learning rate 0.01 * momentum 0.9 * LeakyReLU activation

What is the validation accuracy of the best performing model?

Validation Accuracy = 0.940020 correct

Does the model variation that achieved the highest validation accuracy achieved also the highest test accuracy? * Yes correct * No

0.9.3 Improving Accuracy - Hidden 128

Modifying the model's architecture is also worth considering. Increase the hidden representation size from 10 to 128 and repeat the grid search over the hyper parameters. This time, what modification achieved the highest validation accuracy? * baseline (no modifications) * batch size 64 * learning rate 0.01 * momentum 0.9 * LeakyReLU activation correct

0.10 9. Convolutional Neural Networks

Next, we are going to apply convolutional neural networks to the same task. These networks have demonstrated great performance on many deep learning tasks, especially in computer vision.

You will be working in the files part2-mnist/nnet_cnn.py and part2-mnist/train_utils.py in this problem

0.10.1 Convolutional Neural Networks

We provide skeleton code part2-mnist/nnet_cnn.py which includes examples of some (not all) of the new layers you will need in this part. Using the PyTorch Documentation, complete the code to implement a convolutional neural network with following layers in order: * A convolutional layer with 32 filters of size 3Œ3 * A ReLU nonlinearity * A max pooling layer with size 2Œ2 * A convolutional layer with 64 filters of size 3Œ3 * A ReLU nonlinearity * A max pooling layer with size 2Œ2 * A flatten layer * A fully connected layer with 128 neurons * A dropout layer with drop probability 0.5 * A fully-connected layer with 10 neurons Note: We are not using a softmax layer because it is already present in the loss: PyTorch's nn.CrossEntropyLoss combines nn.LogSoftMax with nn.NLLLoss.

Without GPU acceleration, you will likely find that this network takes quite a long time to train. For that reason, we don't expect you to actually train this network until convergence. Implementing the layers and verifying that you get approximately 93% training accuracy and 98% validation accuracy after one training epoch (this should take less than 10 minutes) is enough for this project. If you are curious, you can let the model train longer; if implemented correctly, your model should achieve >99% test accuracy after 10 epochs of training. If you have access to a CUDA compatible GPU, you could even try configuring PyTorch to use your GPU.

After you successfully implement the above architecture, copy+paste your model code into the codebox below for grading.

Available Functions: You have access to the torch.nn module as nn and to the Flatten layer as Flatten; No need to import anything.

[ ]:
```
model = nn.Sequential(
    nn.Conv2d(1, 32, (3, 3)), # 28*28*1/ 26*26*32
    nn.ReLU(),
    nn.MaxPool2d((2, 2)),  # 13*13*32
```

```
    nn.Conv2d(32, 64, (3, 3)), # 11*11*64
    nn.ReLU(),
    nn.MaxPool2d((2, 2)), # 5*5*64

    # A flatten layer
    Flatten(),
    # A fully connected layer with 128 neurons
    nn.Linear(5*5*64, 128),
    # A dropout layer with drop probability 0.5
    nn.Dropout(p=0.5),
    # A fully-connected layer with 10 neurons - OUTPUT layer
    nn.Linear(128, 10),
)
```

0.11 10. Overlapping, multi-digit MNIST

In this problem, we are going to go beyond the basic MNIST. We will train a few neural networks to solve the problem of hand-written digit recognition using a multi-digit version of MNIST.

You will be working in the files part2-twodigit/mlp.py, part2-twodigit/conv.py, and part2-twodigit/train_utils.py in this problem

In your project folder, look at the part2-twodigit subfolder. There you can find the files mlp.py and conv.py. Your main task here is to complete the code inside the method main in these files.

Do the following steps: * Look at main method in each file. Identify the training and test data and labels. How many images are inside the train and test data? What is the size of each image? * Look at the definition of the MLP class in mlp.py. Try to make sense of what those lines are trying to achieve. What is y_train[0] and y_train[1]? * Look at train_utils.py, particularly the run_epoch function.

Now given the intuition you have built with the above steps, complete the following tasks.

0.11.1 Fully connected network

Complete the code main in mlp.py to build a fully-connected model with a single hidden layer with 64 units. For this, you need to make use of Linear layers in PyTorch; we provide you with an implementation of Flatten, which maps a higher dimensional tensor into an N x d one, where N is the number of samples in your batch and d is the length of the flattend dimension (if your tensor is N x h x w, the flattened dimension is d=(hw)). Hint: Note that your model must have two outputs (corresponding to the first and second digits) to be compatible with the data.

Available Functions: You have access to the torch.nn module as nn, to the torch.nn.functional as F and to the Flatten layer as Flatten; No need to import anything.

[ ]:
```
class MLP(nn.Module):

    def __init__(self, input_dimension):
        super(MLP, self).__init__()
        self.flatten = Flatten()
        # TODO initialize model layers here
        self.linear1 = nn.Linear(input_dimension, 64)
        self.linear2 = nn.Linear(64, 20)
```

```
    def forward(self, x):
        xf = self.flatten(x)

        # TODO use model layers to predict the two digits
        xf = F.relu(self.linear1(xf))
        xf = self.linear2(xf)
        out_first_digit = F.log_softmax(xf[:,:10], dim=1)
        out_second_digit = F.log_softmax(xf [:,10:], dim=1)

        return out_first_digit, out_second_digit
```

0.11.2 Convolutional model

Complete the code main in conv.py to build a convolutional model. For this, you need to make use of Conv2D layers and MaxPool2d layers (and perhaps Dropout) in PyTorch. Make sure that the last layer of the neural network is a fully connected (Linear) layer.

Available Functions: You have access to the torch.nn module as nn, to the torch.nn.functional as F and to the Flatten layer as Flatten; No need to import anything.

[ ]:
```
class CNN(nn.Module):

    def __init__(self, input_dimension):
        super(CNN, self).__init__()
        # TODO initialize model layers here
        self.flatten = Flatten()
        self.conv1 = nn.Conv2d(1, 32, (5, 5))
        self.maxpool2d_1 = nn.MaxPool2d((2, 2))
        self.conv2_drop = nn.Dropout(p=0.6)
        self.linear1 = nn.Linear(19*12*32, 20)

    def forward(self, x):

        # TODO use model layers to predict the two digits
        x = F.relu(self.conv1(x))
        x = self.maxpool2d_1(x)
        x = F.dropout(x)
        x = self.flatten(x)
        x = self.linear1(x)
        out_first_digit = F.log_softmax(x[:,:10], dim=1)
        out_second_digit = F.log_softmax(x [:,10:], dim=1)

        return out_first_digit, out_second_digit
```

0.11.3 Hyperparameter tuning

Next, change the parameters and settings of the models above. Train your model with various parameter settings. Some things you can try is to modify the learning algorithm from SGD to

more sophisticated ones (such as ADAM) or you can modify the network architecture (number of layers or unit per layers, activation function, etc.). Something to ponder: What parameters or settings were more conductive to get a better model with greater generalization capability and lower error? Did extra training for a model always help or did the training accuracy plateau or even get worse after some point?

Finally, we will grade you on finding at least one architecture that achieves over 98% accuracy on both the validation and test set.

Please enter your test accuracy .

0.11.4 Conclusion and What's Next

As you have seen in this project, neural networks can pretty successfully solve the MNIST task. In fact, since 2012, following the impressive performance of AlexNet on the ImageNet dataset, deep neural networks have been the standard in computer vision. As datasets went growing in size and complexity and as computing power became cheaper and more efficient, the trend has been to build deeper and bigger neural nets.

The last part of the project has given you a hint as to why neural networks can be very versatile: by merely changing the output layer, you were able to train the network to predict overlapping MNIST digits. The same building blocks can be reused to build more complex architecture and solve more difficult problems. Using a deep learning framework like Pytorch makes this process even more accessible.

If you have access to a GPU, you can try implementing an object classification system with Resnet, which we have not covered in this course, and maybe expanding it to an object detection or an image segmentation system.

If you do not have access to a GPU, you can try renting resources from an online provider, such as Paperspace (<1$/hour) or Google Colab (free).

0.12 Appendix

0.12.1 nnet_fc.py

- Baseline Model
- batch size = 32
- hidden size = 10
- learning rate = 0.1
- momentum = 0
- Activation funciton: ReLU

[ ]:
```
import _pickle as cPickle, gzip
import numpy as np
from tqdm import tqdm  # progress
import torch
import torch.autograd as autograd
import torch.nn.functional as F
import torch.nn as nn
import sys
sys.path.append("..")
import utils
from utils import *
```

```
from train_utils import batchify_data, run_epoch, train_model
```

[ ]:
```
def main():
    # Load the dataset
    num_classes = 10
    X_train, y_train, X_test, y_test = get_MNIST_data()

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)  #smart
    X_dev = X_train[dev_split_index:]
    y_dev = y_train[dev_split_index:]
    X_train = X_train[:dev_split_index]
    y_train = y_train[:dev_split_index]

    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [y_train[i] for i in permutation]

    # Split dataset into batches
    batch_size = 32
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    #################################
    ## Model specification TODO
    model = nn.Sequential(
              nn.Linear(784, 10),
              nn.ReLU(),  # baseline
              nn.Linear(10, 10),
            )
    lr=0.1      # baseline
    momentum=0  # baseline
    #################################

    train_model(train_batches, dev_batches, model, lr=lr, momentum=momentum)

    ## Evaluate the model on test data
    loss, accuracy = run_epoch(test_batches, model.eval(), None)

    print ("Loss on test set:"  + str(loss) + " Accuracy on test set: " +␣
 ↪str(accuracy))


if __name__ == '__main__':
```

```
    # Specify seed for deterministic behavior, then shuffle. Do not change seed␣
↪for official submissions to edx
    np.random.seed(12321)  # for reproducibility
    torch.manual_seed(12321)  # for reproducibility
    main()
```

[ ]:
```
def main(p_batch_size, p_learning_rate, p_momentum):
    print(p_batch_size, p_learning_rate, p_momentum)
    # Load the dataset
    num_classes = 10
    X_train, y_train, X_test, y_test = get_MNIST_data()

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)  #smart
    X_dev = X_train[dev_split_index:]
    y_dev = y_train[dev_split_index:]
    X_train = X_train[:dev_split_index]
    y_train = y_train[:dev_split_index]

    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [y_train[i] for i in permutation]

    # Split dataset into batches
    batch_size = p_batch_size
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    #################################
    ## Model specification TODO
    model = nn.Sequential(
              nn.Linear(784, 128),
              nn.ReLU(),  # baseline
              nn.Linear(128, 10),
            )
    lr=p_learning_rate      # baseline
    momentum=p_momentum  # baseline
    #################################

    train_model(train_batches, dev_batches, model, lr=lr, momentum=momentum)

    ## Evaluate the model on test data
    loss, accuracy = run_epoch(test_batches, model.eval(), None)
```

```
    print ("Loss on test set:"  + str(loss) + " Accuracy on test set: " +␣
 ↪str(accuracy))


def m_LeakyReLU():
    # Load the dataset
    num_classes = 10
    X_train, y_train, X_test, y_test = get_MNIST_data()

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)  #smart
    X_dev = X_train[dev_split_index:]
    y_dev = y_train[dev_split_index:]
    X_train = X_train[:dev_split_index]
    y_train = y_train[:dev_split_index]

    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [y_train[i] for i in permutation]

    # Split dataset into batches
    batch_size = 32
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    #################################
    ## Model specification TODO
    model = nn.Sequential(
              nn.Linear(784, 10),
              nn.LeakyReLU(),  # baseline
              nn.Linear(10, 10),
            )
    lr=0.1      # baseline
    momentum=0  # baseline
    #################################

    train_model(train_batches, dev_batches, model, lr=lr, momentum=momentum)

    ## Evaluate the model on test data
    loss, accuracy = run_epoch(test_batches, model.eval(), None)

    print ("m_LeakyReLU Loss on test set:"  + str(loss) + " Accuracy on test␣
 ↪set: " + str(accuracy))
```

```
if __name__ == '__main__':
    # Specify seed for deterministic behavior, then shuffle. Do not change seed␣
↪for official submissions to edx
    np.random.seed(12321)  # for reproducibility
    torch.manual_seed(12321)  # for reproducibility

    p_batch_size, p_learning_rate, p_momentum = 32, 0.1, 0
    main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.267226892666674793 Accuracy on test set: 0.
↪9204727564102564
    # Loss on test set:0.07372987373039508 Accuracy on test set: 0.
↪9767628205128205

    #p_batch_size, p_learning_rate, p_momentum = 64, 0.1, 0
    #main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.2423846272703929 Accuracy on test set: 0.
↪9314903846153846
    # Loss on test set:0.08878818156723028 Accuracy on test set: 0.
↪9749599358974359

    #p_batch_size, p_learning_rate, p_momentum = 32, 0.01, 0
    #main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.27886550653821385 Accuracy on test set: 0.
    # Loss on test set:0.20072906261357742 Accuracy on test set: 0.
↪9425080128205128

    #p_batch_size, p_learning_rate, p_momentum = 32, 0.1, 0.9
    #main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.5125645012952961 Accuracy on test set: 0.8828125
    # Loss on test set:0.1940298633506665 Accuracy on test set: 0.
↪9671474358974359

    m_LeakyReLU()
    # Loss on test set:0.2689260929488601 Accuracy on test set: 0.
↪9207732371794872
    # m_LeakyReLU Loss on test set:0.07587968118679829 Accuracy on test set: 0.
↪9766626602564102
```

[ ]:
```
def main(p_batch_size, p_learning_rate, p_momentum):
    print(p_batch_size, p_learning_rate, p_momentum)
    # Load the dataset
    num_classes = 10
    X_train, y_train, X_test, y_test = get_MNIST_data()

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)  #smart
    X_dev = X_train[dev_split_index:]
```

```
    y_dev = y_train[dev_split_index:]
    X_train = X_train[:dev_split_index]
    y_train = y_train[:dev_split_index]

    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [y_train[i] for i in permutation]

    # Split dataset into batches
    batch_size = p_batch_size
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    #################################
    ## Model specification TODO
    model = nn.Sequential(
              nn.Linear(784, 128),
              nn.ReLU(),  # baseline
              nn.Linear(128, 10),
            )
    lr=p_learning_rate      # baseline
    momentum=p_momentum  # baseline
    #################################

    train_model(train_batches, dev_batches, model, lr=lr, momentum=momentum)

    ## Evaluate the model on test data
    loss, accuracy = run_epoch(test_batches, model.eval(), None)

    print ("Loss on test set:"  + str(loss) + " Accuracy on test set: " +␣
 ↪str(accuracy))


def m_LeakyReLU():
    # Load the dataset
    num_classes = 10
    X_train, y_train, X_test, y_test = get_MNIST_data()

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)  #smart
    X_dev = X_train[dev_split_index:]
    y_dev = y_train[dev_split_index:]
    X_train = X_train[:dev_split_index]
    y_train = y_train[:dev_split_index]
```

```
    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [y_train[i] for i in permutation]

    # Split dataset into batches
    batch_size = 32
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    #################################
    ## Model specification TODO
    model = nn.Sequential(
              nn.Linear(784, 128), # change from 10 to 128 hidden size
              nn.LeakyReLU(),  # baseline
              nn.Linear(128, 10),
            )
    lr=0.1      # baseline
    momentum=0  # baseline
    #################################

    train_model(train_batches, dev_batches, model, lr=lr, momentum=momentum)

    ## Evaluate the model on test data
    loss, accuracy = run_epoch(test_batches, model.eval(), None)

    print ("m_LeakyReLU Loss on test set:"  + str(loss) + " Accuracy on test␣
↪set: " + str(accuracy))


if __name__ == '__main__':
    # Specify seed for deterministic behavior, then shuffle. Do not change seed␣
↪for official submissions to edx
    np.random.seed(12321)  # for reproducibility
    torch.manual_seed(12321)  # for reproducibility

    p_batch_size, p_learning_rate, p_momentum = 32, 0.1, 0
    main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.267226892666674793 Accuracy on test set: 0.
↪9204727564102564
    # Loss on test set:0.073729873373039508 Accuracy on test set: 0.
↪9767628205128205

    #p_batch_size, p_learning_rate, p_momentum = 64, 0.1, 0
    #main(p_batch_size, p_learning_rate, p_momentum)
```

```
    # Loss on test set:0.2423846272703929 Accuracy on test set: 0.
↪9314903846153846
    # Loss on test set:0.08878818156723028 Accuracy on test set: 0.
↪9749599358974359

    #p_batch_size, p_learning_rate, p_momentum = 32, 0.01, 0
    #main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.27886550653821385 Accuracy on test set: 0.
    # Loss on test set:0.20072906261357742 Accuracy on test set: 0.
↪9425080128205128

    #p_batch_size, p_learning_rate, p_momentum = 32, 0.1, 0.9
    #main(p_batch_size, p_learning_rate, p_momentum)
    # Loss on test set:0.5125645012952961 Accuracy on test set: 0.8828125
    # Loss on test set:0.1940298633506665 Accuracy on test set: 0.
↪9671474358974359

    m_LeakyReLU()
    # Loss on test set:0.2689260929488601 Accuracy on test set: 0.
↪9207732371794872
    # m_LeakyReLU Loss on test set:0.07587968118679829 Accuracy on test set: 0.
↪9766626602564102
```

0.12.2 nnet_cnn.py

[ ]:
```
import numpy as np
import torch.nn as nn
import _pickle as c_pickle, gzip
from tqdm import tqdm
import torch
import torch.autograd as autograd
import torch.nn.functional as F
import sys
sys.path.append("..")
import utils
from utils import *
from train_utils import batchify_data, run_epoch, train_model, Flatten

def main():
    # Load the dataset
    num_classes = 10
    X_train, y_train, X_test, y_test = get_MNIST_data()

    # We need to rehape the data back into a 1x28x28 image
    X_train = np.reshape(X_train, (X_train.shape[0], 1, 28, 28))
    X_test = np.reshape(X_test, (X_test.shape[0], 1, 28, 28))
```

```
# Split into train and dev
dev_split_index = int(9 * len(X_train) / 10)
X_dev = X_train[dev_split_index:]
y_dev = y_train[dev_split_index:]
X_train = X_train[:dev_split_index]
y_train = y_train[:dev_split_index]

permutation = np.array([i for i in range(len(X_train))])
np.random.shuffle(permutation)
X_train = [X_train[i] for i in permutation]
y_train = [y_train[i] for i in permutation]

# Split dataset into batches
batch_size = 32
train_batches = batchify_data(X_train, y_train, batch_size)
dev_batches = batchify_data(X_dev, y_dev, batch_size)
test_batches = batchify_data(X_test, y_test, batch_size)

#################################
## Model specification TODO
model = nn.Sequential(
          # 28*28*1
          nn.Conv2d(1, 32, (3, 3)),
          nn.ReLU(),
           # 26*26*32
          nn.MaxPool2d((2, 2)),

          # 13*13*32 , 11*11*64
          nn.Conv2d(32, 64, (3, 3)),
          nn.ReLU(),
          # 5*5*64
          nn.MaxPool2d((2, 2)),

          # A flatten layer
          Flatten(),
          # A fully connected layer with 128 neurons
          nn.Linear(1600, 128),
          # A dropout layer with drop probability 0.5
          nn.Dropout(p=0.5),
          # A fully-connected layer with 10 neurons - OUTPUT layer
          nn.Linear(128, 10),
        )
#################################

train_model(train_batches, dev_batches, model, nesterov=True)
```

```
    ## Evaluate the model on test data
    loss, accuracy = run_epoch(test_batches, model.eval(), None)

    print ("Loss on test set:"  + str(loss) + " Accuracy on test set: " +␣
 ↪str(accuracy))

if __name__ == '__main__':
    # Specify seed for deterministic behavior, then shuffle. Do not change seed␣
 ↪for official submissions to edx
    np.random.seed(12321)  # for reproducibility
    torch.manual_seed(12321)
    main()
```

0.12.3 mlp.py

[ ]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

from train_utils import batchify_data, run_epoch, train_model, Flatten
import utils_multiMNIST as U

path_to_data_dir = '../Datasets/'
use_mini_dataset = True

batch_size = 64
nb_classes = 10
nb_epoch = 30
num_classes = 10
img_rows, img_cols = 42, 28 # input image dimensions

class MLP(nn.Module):

    def __init__(self, input_dimension):
        super(MLP, self).__init__()
        self.flatten = Flatten()
        # TODO initialize model layers here
        self.linear1 = nn.Linear(input_dimension, 64)
        self.linear2 = nn.Linear(64, 20)

    def forward(self, x):
        xf = self.flatten(x)

        # TODO use model layers to predict the two digits
        xf = self.linear1(xf)
```

```
        xf = F.relu(xf)
        xf = self.linear2(xf)
        out_first_digit = F.log_softmax(xf[:,:10], dim=1)
        out_second_digit = F.log_softmax(xf [:,10:], dim=1)
        return out_first_digit, out_second_digit

def main():
    X_train, y_train, X_test, y_test = U.get_data(path_to_data_dir,␣
 ↪use_mini_dataset)

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)
    X_dev = X_train[dev_split_index:]
    y_dev = [y_train[0][dev_split_index:], y_train[1][dev_split_index:]]
    X_train = X_train[:dev_split_index]
    y_train = [y_train[0][:dev_split_index], y_train[1][:dev_split_index]]

    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [[y_train[0][i] for i in permutation], [y_train[1][i] for i in␣
 ↪permutation]]

    # Split dataset into batches
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    # Load model
    input_dimension = img_rows * img_cols
    model = MLP(input_dimension) # TODO add proper layers to MLP class above

    # Train
    train_model(train_batches, dev_batches, model)

    ## Evaluate the model on test data
    loss, acc = run_epoch(test_batches, model.eval(), None)
    print('Test loss1: {:.6f}  accuracy1: {:.6f}  loss2: {:.6f}   accuracy2: {:.
 ↪6f}'.format(loss[0], acc[0], loss[1], acc[1]))

if __name__ == '__main__':
    # Specify seed for deterministic behavior, then shuffle. Do not change seed␣
 ↪for official submissions to edx
    np.random.seed(12321)  # for reproducibility
    torch.manual_seed(12321)  # for reproducibility
    main()
```

0.12.4 cnn.py

[ ]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from train_utils import batchify_data, run_epoch, train_model, Flatten
import utils_multiMNIST as U
path_to_data_dir = '../Datasets/'
use_mini_dataset = True

batch_size = 64
nb_classes = 10
nb_epoch = 30
num_classes = 10
img_rows, img_cols = 42, 28 # input image dimensions



class CNN(nn.Module):

    def __init__(self, input_dimension):
        super(CNN, self).__init__()
        # TODO initialize model layers here
        self.flatten = Flatten()
        self.conv1 = nn.Conv2d(1, 32, (5, 5))
        self.maxpool2d_1 = nn.MaxPool2d((2, 2))
        self.conv2 = nn.Conv2d(32, 64, (3, 3))
        self.maxpool2d_2 = nn.MaxPool2d((2, 2))
        self.conv2_drop = nn.Dropout(p=0.6)
        self.linear1 = nn.Linear(19*12*32, 20)

    def forward(self, x):

        # TODO use model layers to predict the two digits
        x = F.relu(self.conv1(x))
        x = self.maxpool2d_1(x)
        x = F.dropout(x)
        x = self.flatten(x)
        x = self.linear1(x)
        out_first_digit = F.log_softmax(x[:,:10], dim=1)
        out_second_digit = F.log_softmax(x [:,10:], dim=1)

        return out_first_digit, out_second_digit

def main():
```

```
    X_train, y_train, X_test, y_test = U.get_data(path_to_data_dir,␣
↪use_mini_dataset)

    # Split into train and dev
    dev_split_index = int(9 * len(X_train) / 10)
    X_dev = X_train[dev_split_index:]
    y_dev = [y_train[0][dev_split_index:], y_train[1][dev_split_index:]]
    X_train = X_train[:dev_split_index]
    y_train = [y_train[0][:dev_split_index], y_train[1][:dev_split_index]]

    permutation = np.array([i for i in range(len(X_train))])
    np.random.shuffle(permutation)
    X_train = [X_train[i] for i in permutation]
    y_train = [[y_train[0][i] for i in permutation], [y_train[1][i] for i in␣
↪permutation]]

    # Split dataset into batches
    train_batches = batchify_data(X_train, y_train, batch_size)
    dev_batches = batchify_data(X_dev, y_dev, batch_size)
    test_batches = batchify_data(X_test, y_test, batch_size)

    # Load model
    input_dimension = img_rows * img_cols
    model = CNN(input_dimension) # TODO add proper layers to CNN class above

    # Train
    train_model(train_batches, dev_batches, model)

    ## Evaluate the model on test data
    loss, acc = run_epoch(test_batches, model.eval(), None)
    print('Test loss1: {:.6f}  accuracy1: {:.6f}  loss2: {:.6f}   accuracy2: {:.
↪6f}'.format(loss[0], acc[0], loss[1], acc[1]))

if __name__ == '__main__':
    # Specify seed for deterministic behavior, then shuffle. Do not change seed␣
↪for official submissions to edx
    np.random.seed(12321)  # for reproducibility
    torch.manual_seed(12321)  # for reproducibility
    main()
```

0.12.5 utils_multiMNIST.py

[1]:
```
import gzip, _pickle, numpy as np

num_classes = 10
img_rows, img_cols = 42, 28
```

```
def get_data(path_to_data_dir, use_mini_dataset):
    if use_mini_dataset:
        exten = '_mini'
    else:
        exten = ''

    f = gzip.open(path_to_data_dir + 'train_multi_digit' + exten + '.pkl.gz',␣
↪'rb')
    X_train = _pickle.load(f, encoding='latin1')
    f.close()
    X_train =  np.reshape(X_train, (len(X_train), 1, img_rows, img_cols))

    f = gzip.open(path_to_data_dir + 'test_multi_digit' + exten +'.pkl.gz',␣
↪'rb')
    X_test = _pickle.load(f, encoding='latin1')
    f.close()
    X_test =  np.reshape(X_test, (len(X_test),1, img_rows, img_cols))

    f = gzip.open(path_to_data_dir + 'train_labels' + exten +'.txt.gz', 'rb')
    y_train = np.loadtxt(f)
    f.close()

    f = gzip.open(path_to_data_dir +'test_labels' + exten + '.txt.gz', 'rb')
    y_test = np.loadtxt(f)
    f.close()

    return X_train, y_train, X_test, y_test
```

0.12.6 train_utils.py

[ ]:
```
"""Training utilities."""

from tqdm import tqdm
import numpy as np
import torch
import torch.nn.functional as F
import torch.nn as nn


class Flatten(nn.Module):
    """A custom layer that views an input as 1D."""

    def forward(self, input):
        return input.view(input.size(0), -1)
```

```
def batchify_data(x_data, y_data, batch_size):
    """Takes a set of data points and labels and groups them into batches."""
    # Only take batch_size chunks (i.e. drop the remainder)
    N = int(len(x_data) / batch_size) * batch_size
    batches = []
    for i in range(0, N, batch_size):
        batches.append({
            'x': torch.tensor(x_data[i:i + batch_size],
                              dtype=torch.float32),
            'y': torch.tensor([y_data[0][i:i + batch_size],
                               y_data[1][i:i + batch_size]],
                               dtype=torch.int64)
        })
    return batches


def compute_accuracy(predictions, y):
    """Computes the accuracy of predictions against the gold labels, y."""
    return np.mean(np.equal(predictions.numpy(), y.numpy()))


def train_model(train_data, dev_data, model, lr=0.01, momentum=0.9,␣
↪nesterov=False, n_epochs=30):
    """Train a model for N epochs given data and hyper-params."""
    # We optimize with SGD
    optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum,␣
↪nesterov=nesterov)

    for epoch in range(1, n_epochs + 1):
        print("-------------\nEpoch {}:\n".format(epoch))

        # Run **training***
        loss, acc = run_epoch(train_data, model.train(), optimizer)
        print('Train | loss1: {:.6f}  accuracy1: {:.6f} | loss2: {:.6f} ␣
↪accuracy2: {:.6f}'.format(loss[0], acc[0], loss[1], acc[1]))

        # Run **validation**
        val_loss, val_acc = run_epoch(dev_data, model.eval(), optimizer)
        print('Valid | loss1: {:.6f}  accuracy1: {:.6f} | loss2: {:.6f} ␣
↪accuracy2: {:.6f}'.format(val_loss[0], val_acc[0], val_loss[1], val_acc[1]))

        # Save model
        torch.save(model, 'mnist_model_fully_connected.pt')


def run_epoch(data, model, optimizer):
```

```
    """Train model for one pass of train data, and return loss, acccuracy"""
    # Gather losses
    losses_first_label = []
    losses_second_label = []
    batch_accuracies_first = []
    batch_accuracies_second = []

    # If model is in train mode, use optimizer.
    is_training = model.training

    # Iterate through batches
    for batch in tqdm(data):
        # Grab x and y
        x, y = batch['x'], batch['y']

        # Get output predictions for both the upper and lower numbers
        out1, out2 = model(x)

        # Predict and store accuracy
        predictions_first_label = torch.argmax(out1, dim=1)
        predictions_second_label = torch.argmax(out2, dim=1)
        batch_accuracies_first.append(compute_accuracy(predictions_first_label,␣
↪y[0]))
        batch_accuracies_second.
↪append(compute_accuracy(predictions_second_label, y[1]))

        # Compute both losses
        loss1 = F.cross_entropy(out1, y[0])
        loss2 = F.cross_entropy(out2, y[1])
        losses_first_label.append(loss1.data.item())
        losses_second_label.append(loss2.data.item())

        # If training, do an update.
        if is_training:
            optimizer.zero_grad()
            joint_loss = 0.5 * (loss1 + loss2)
            joint_loss.backward()
            optimizer.step()

    # Calculate epoch level scores
    avg_loss = np.mean(losses_first_label), np.mean(losses_second_label)
    avg_accuracy = np.mean(batch_accuracies_first), np.
↪mean(batch_accuracies_second)
    return avg_loss, avg_accuracy
```