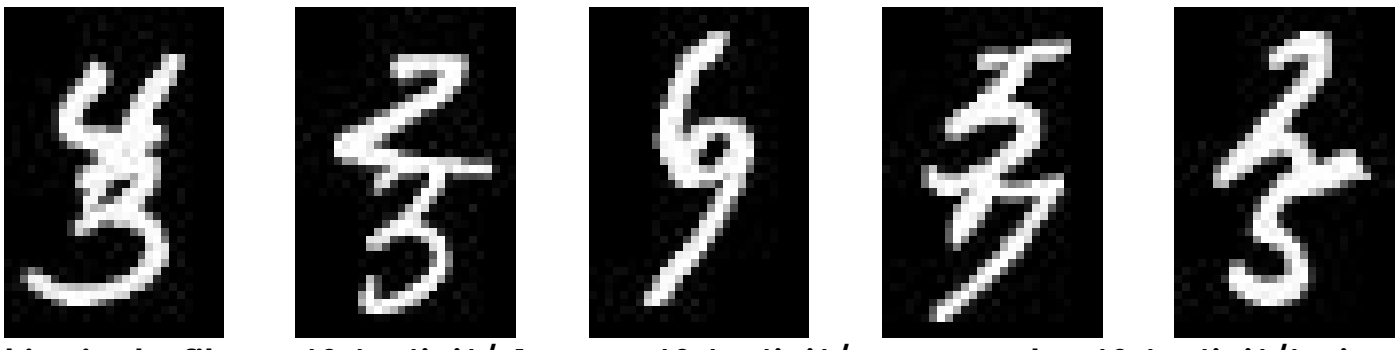
10. Overlapping, multi-digit MNIST

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.

Fully connected network

5.0/5.0 points (graded)

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 $d = (h \cdot w)$). 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()
        # initialize model layers here
        self.fc1 = nn.Linear(input_dimension, 64)
        self.fc2_1 = nn.Linear(64, 10)
        self.fc2_2 = nn.Linear(64, 10)
        self.logsoftmax = nn.LogSoftmax()


    def forward(self, x):
```

```
14         xf = self.flatten(x)
15
```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
class MLP(nn.Module):

    def __init__(self, input_dimension):
        super(MLP, self).__init__()
        self.flatten = Flatten()
        self.linear1 = nn.Linear(input_dimension, 64)
        self.linear2 = nn.Linear(64, 64)
        self.linear_first_digit = nn.Linear(64, 10)
        self.linear_second_digit = nn.Linear(64, 10)

    def forward(self, x):
        xf = self.flatten(x)
        out1 = F.relu(self.linear1(xf))
        out2 = F.relu(self.linear2(out1))
        out_first_digit = self.linear_first_digit(out2)
        out_second_digit = self.linear_second_digit(out2)
        return out_first_digit, out_second_digit
```

Test results

See full output

CORRECT

See full output

Submit You have used 4 of 25 attempts

Answers are displayed within the problem

Convolutional model

5.0/5.0 points (graded)
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__()
        # initialize model layers here
        self.conv1 = nn.Conv2d(input_dimension, 128, 5)
        self.relu = nn.ReLU()
        self.mp = nn.MaxPool2d((2, 2))
        self.fc1 = nn.Linear(64, 32)
        self.dropout = nn.Dropout(p = 0.5)
        self.fc2_1 = nn.Linear(32, 10)
        self.fc2_2 = nn.Linear(32, 10)
        self.logsoftmax = nn.LogSoftmax()


    def forward(self, x):
```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
class CNN(nn.Module):

    def __init__(self, input_dimension):
        super(CNN, self).__init__()
        self.linear1 = nn.Linear(input_dimension, 64)
        self.linear2 = nn.Linear(64, 64)
        self.linear_first_digit = nn.Linear(64, 10)
        self.linear_second_digit = nn.Linear(64, 10)

        self.encoder = nn.Sequential(
              nn.Conv2d(1, 8, (3, 3)),
              nn.ReLU(),
              nn.MaxPool2d((2, 2)),
              nn.Conv2d(8, 16, (3, 3)),
              nn.ReLU(),
              nn.MaxPool2d((2, 2)),
              Flatten(),
              nn.Linear(720, 128),
              nn.Dropout(0.5),
        )

        self.first_digit_classifier = nn.Linear(128,10)
        self.second_digit_classifier = nn.Linear(128,10)

    def forward(self, x):
        out = self.encoder(x)
        out_first_digit = self.first_digit_classifier(out)
        out_second_digit = self.second_digit_classifier(out)
        return out_first_digit, out_second_digit
```

Test results

See full output

CORRECT

See full output

Submit You have used 2 of 25 attempts

Answers are displayed within the problem

Hyperparameter tuning

1/1 point (graded)

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** .

0.981603 ✔ **Answer:** 0.99

Submit You have used 3 of 5 attempts

Answers are displayed within the problem

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).

Discussion

Show Discussion

Topic: Unit 3 Neural networks (2.5 weeks):Project 3: Digit recognition (Part 2) / 10. Overlapping, multi-digit MNIST