UDACITY

‹ Return to Classroom

Train a Character Recognition System with MNIST

REVIEW

HISTORY

Meets Specifications

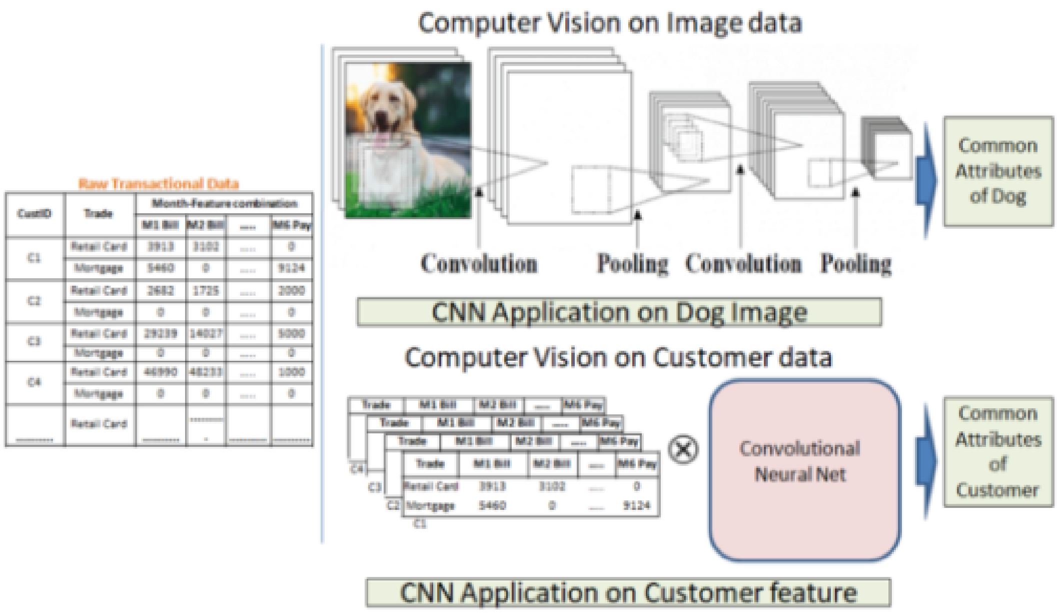
Great submission!! You have knocked it out of the park in this submission with end to end run of model training and saving.

I'll share some takeaways from this project as it will help you to tackle problems like these in the future when datasets will grow beyond MNIST.

1) The generalizability of computer vision and deep learning models

- The concepts you are learning have varied usages and in computer vision, each field has evolved now a lot like object detection, tracking, segmentation and so on.
- Always remember that the processes we do on image matrices remain the same and you will re-use all the concepts of filters and feature extractors in one way or other across different use cases and modelling.

2) Thinking of CNNs as feature extractors followed by a classifier.


Computer Vision on Image data
Raw Transactional Data
Convolution
Pooling
Convolution
Pooling
Common Attributes of Dog
CNN Application on Dog Image
Computer Vision on Customer data
Convolutional Neural Net
Common Attributes of Customer
CNN Application on Customer feature

- You can always use CNNs in general to extract features and then use them for whatever purpose you see fit.

All the best in your AI journey with Udacity :)

Section 1: Data Loading and Exploration

Data is preprocessed and converted to a tensor, either using the .ToTensor() transform from `torchvision.transforms` or simply manually with torch.Tensor.

```
# Define transforms
transforms = transforms.ToTensor()

# Create training set and define training dataloader
train_data = torchvision.datasets.MNIST(root="data", train=True, download=
True, transform=transforms)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Create test set and define test dataloader
test_data = torchvision.datasets.MNIST(root="data", train=False, download=
True, transform=transforms)
test_loader = DataLoader(test_data, batch_size=100)
```

Done well

- Use of normalization can also be done here. A simple 0.5,0.5 normalization also works at times.
- Using Imagenet's standard normalization values are also done in many Computer Vision centric tasks so do take a look at that as well.

https://stackoverflow.com/questions/58151507/why-pytorch-officially-use-mean-0-485-0-456-0-406-and-std-0-229-0-224-0-2

A `DataLoader` object for both train and test sets has been created using the train and test sets loaded from `torchvision`.

```
train_data = torchvision.datasets.MNIST(root="data", train=True, download=
True, transform=transforms)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Create test set and define test dataloader
test_data = torchvision.datasets.MNIST(root="data", train=False, download=
True, transform=transforms)
test_loader = DataLoader(test_data, batch_size=100)
```

- Creating validation split was optional so do look at that directly with dataloaders. There are many ways in which you can specify the split by using different methods for eg. using random_split is a good choice.
- Subsetrandomsampler and using `0.x` multiplied by data length gives you more control when you decide to split by % (80-20 split) instead of hardcoding values. https://stackoverflow.com/questions/50544730/how-do-i-split-a-custom-dataset-into-training-and-test-datasets

Notebook contains code which shows the size and shape of the training and test data.

The provided function or some other method (e.g. plt.imshow) is used to print one or more images from the dataset

Done well.
Do check the visualization utilities in PyTorch as these are handy functions and transformations to display/save images in a particular way.
https://pytorch.org/vision/stable/auto_examples/plot_visualization_utils.html

```
    batch = next(dataiter)
    labels = batch[1][0:5]
    images = batch[0][0:5]
    for i in range(5):
        print(int(labels[i].detach()))

        image = images[i].numpy()
        plt.imshow(image.T.squeeze().T)
        plt.show()
```

```
[4]: print("Number of MNIST train data examples: {}".format(len(train_data)))
     print("Number of MNIST test data examples: {}".format(len(test_data)))
```

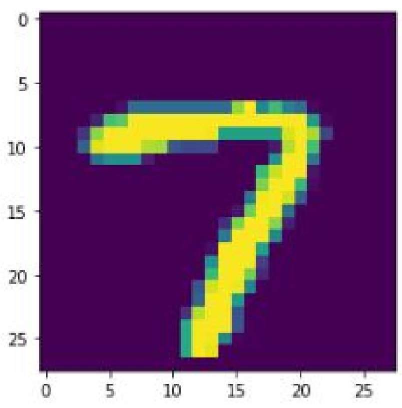
```
Number of MNIST train data examples: 60000
Number of MNIST test data examples: 10000
```

```
[5]: dataiter = iter(train_loader)
     images, labels = next(dataiter)

     print("number of images: {}".format(images.shape))
     print("number of labels: {}".format(labels.shape))
```

```
number of images: torch.Size([100, 1, 28, 28])
number of labels: torch.Size([100])
```

```
[6]: show5(train_loader)
```

```
7
```



The submission contains a justification of necessary preprocessing steps (e.g. flattening, converting to tensor, normalization) in a comment or (preferably) a markdown block.

```
**To convert an image data set into a tensor.**
```

Do note that transforms is not just for converting the data into a format that can be perceived by model but also to augment, enhance and change features of input data at times as well.

Some more references for you to refer to.

- https://www.datanami.com/2019/11/08/why-you-need-data-transformation-in-machine-learning/
- https://stats.stackexchange.com/questions/211436/why-normalize-images-by-subtracting-datasets-image-mean-instead-of-the-current

Section 2: Model Design and Training

A `Model` or `Sequential` class is created with at least two hidden layers and implements a `forward` method that outputs a prediction probability for each of the 10 classes using softmax.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.activation = F.relu
        self.layer1 = nn.Linear(28 * 28, 128)
        self.layer2 = nn.Linear(128, 64)
        self.layer3 = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = self.activation(self.layer1(x))
        x = self.activation(self.layer2(x))
        x = self.layer3(x)
        return x

net = Net()
```

- A nice simple ANN model. There have been lots of experiments around architecture testing as MNIST is used a default dataset for testing things, especially ML/DL models.
- A great benchmark if you want to check out what researchers have built to achieve state-of-the-art accuracy on it\
  https://paperswithcode.com/sota/image-classification-on-mnist

A loss function that works for classification tasks is specified.

```
 Choose a loss function
criterion = nn.CrossEntropyLoss()
```

Good use of cross entropy loss.
This cheatsheet is always handy to refer when you want to decide which loss function to use in different use cases and architectures.

https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

Any optimizer from `torch.optim` is used to minimize the loss function.

```
# Choose an optimizer
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

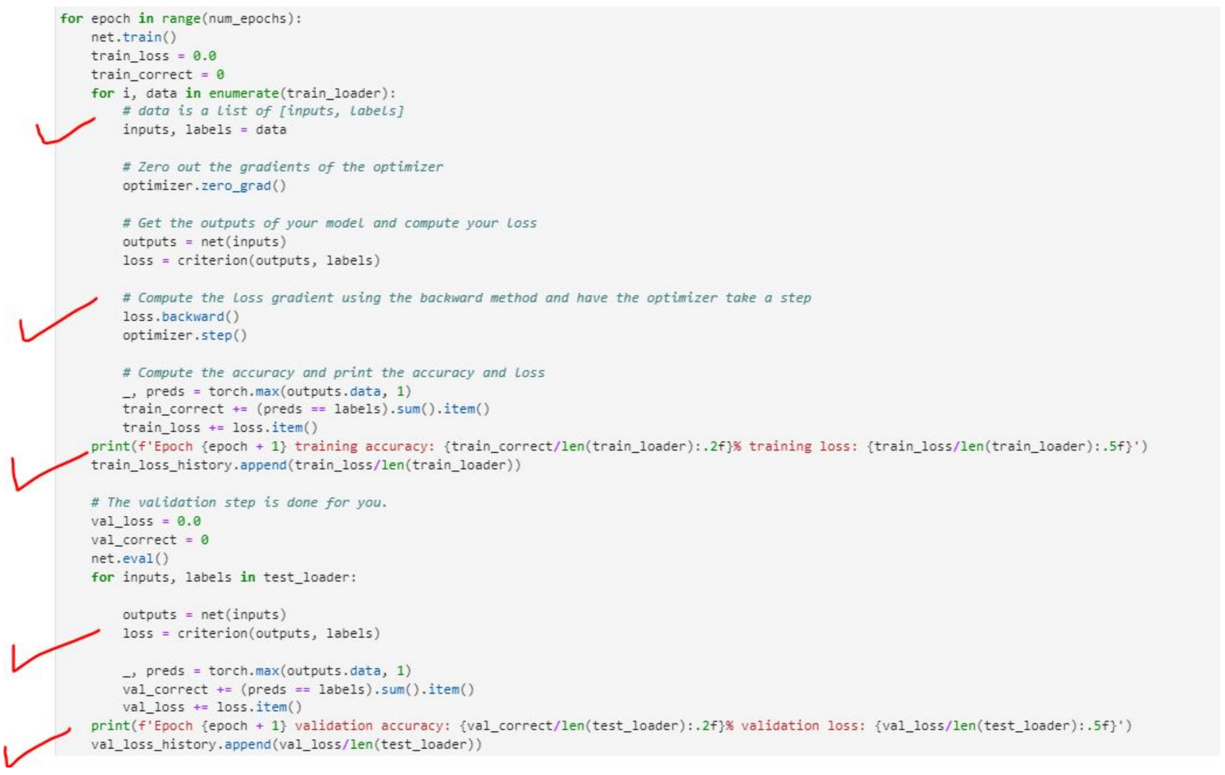
Glad that you passed the learning rate value as custom function parameter.

There are also advanced adaptive learning rate algorithms now too which increase/decrease the lr in steps based on the epochs and loss values. Do check them out as learning rate is quite an interesting and important hyperparameter in deep learning models.

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f4339

Section 3: Model Testing and Evaluation

The test `DataLoader` is used to get predictions from the neural network and compare the predictions to the true labels.

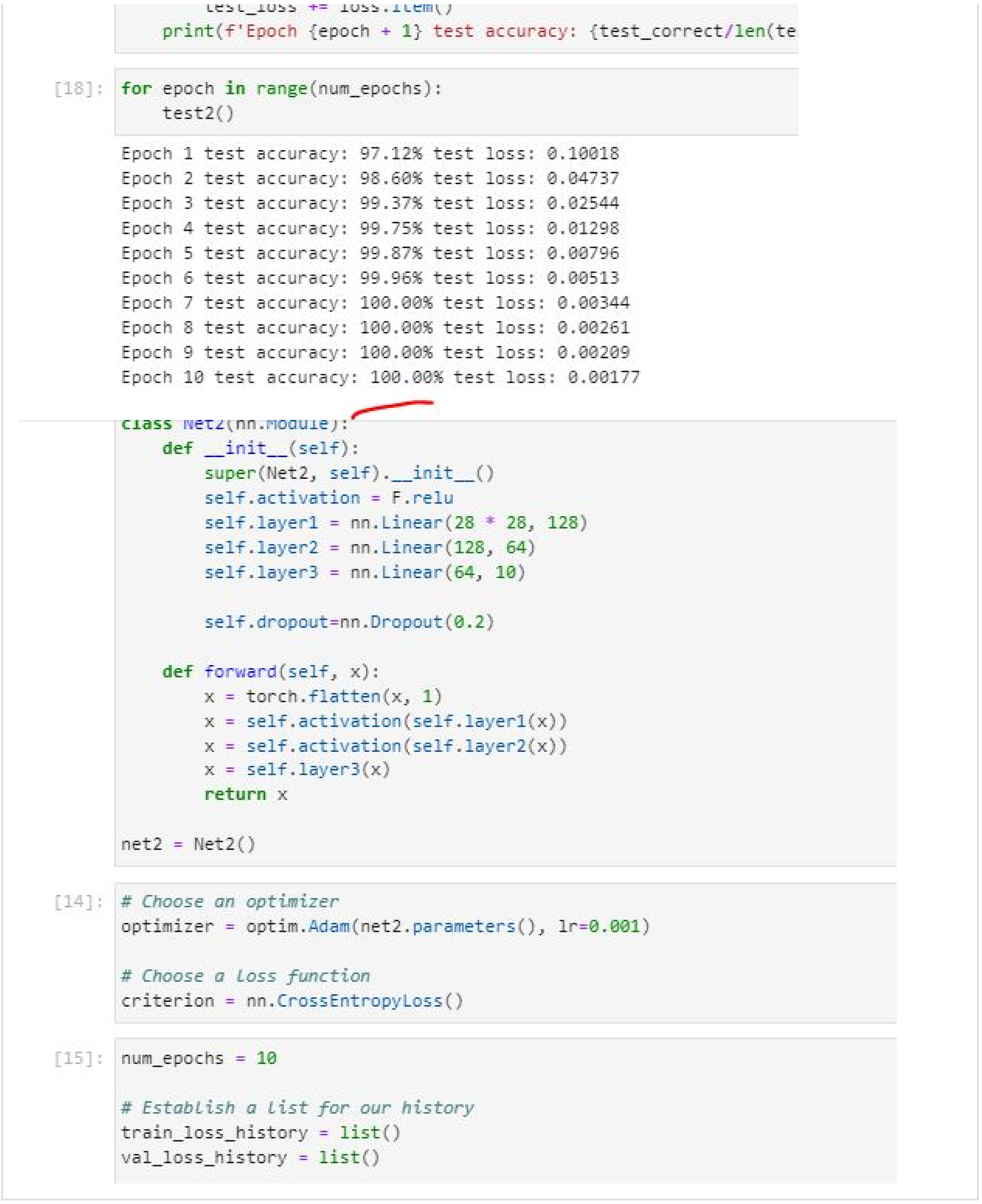


```python
for epoch in range(num_epochs):
    net.train()
    train_loss = 0.0
    train_correct = 0
    for i, data in enumerate(train_loader):
        # data is a list of [inputs, labels]
        inputs, labels = data

        # Zero out the gradients of the optimizer
        optimizer.zero_grad()

        # Get the outputs of your model and compute your loss
        outputs = net(inputs)
        loss = criterion(outputs, labels)

        # Compute the loss gradient using the backward method and have the optimizer take a step
        loss.backward()
        optimizer.step()

        # Compute the accuracy and print the accuracy and loss
        _, preds = torch.max(outputs.data, 1)
        train_correct += (preds == labels).sum().item()
        train_loss += loss.item()
    print(f'Epoch {epoch + 1} training accuracy: {train_correct/len(train_loader):.2f}% training loss: {train_loss/len(train_loader):.5f}')
    train_loss_history.append(train_loss/len(train_loader))

    # The validation step is done for you.
    val_loss = 0.0
    val_correct = 0
    net.eval()
    for inputs, labels in test_loader:

        outputs = net(inputs)
        loss = criterion(outputs, labels)

        _, preds = torch.max(outputs.data, 1)
        val_correct += (preds == labels).sum().item()
        val_loss += loss.item()
    print(f'Epoch {epoch + 1} validation accuracy: {val_correct/len(test_loader):.2f}% validation loss: {val_loss/len(test_loader):.5f}')
    val_loss_history.append(val_loss/len(test_loader))
```

You have used test_loader to display test accuracy which is correct.

Hyperparameters are modified to attempt to improve accuracy and the model achieves at least 90% classification accuracy.

- There is nice gain which is nearly making your test accuracy reach 100% !
- Do look at changing the optimizer and also try with CNN layers instead of simple NN and compare and contrast results for yourself.



```
    test_loss += loss.item()
    print(f'Epoch {epoch + 1} test accuracy: {test_correct/len(te
```

[18]:
```
for epoch in range(num_epochs):
    test2()
```

```
Epoch 1 test accuracy: 97.12% test loss: 0.10018
Epoch 2 test accuracy: 98.60% test loss: 0.04737
Epoch 3 test accuracy: 99.37% test loss: 0.02544
Epoch 4 test accuracy: 99.75% test loss: 0.01298
Epoch 5 test accuracy: 99.87% test loss: 0.00796
Epoch 6 test accuracy: 99.96% test loss: 0.00513
Epoch 7 test accuracy: 100.00% test loss: 0.00344
Epoch 8 test accuracy: 100.00% test loss: 0.00261
Epoch 9 test accuracy: 100.00% test loss: 0.00209
Epoch 10 test accuracy: 100.00% test loss: 0.00177
```

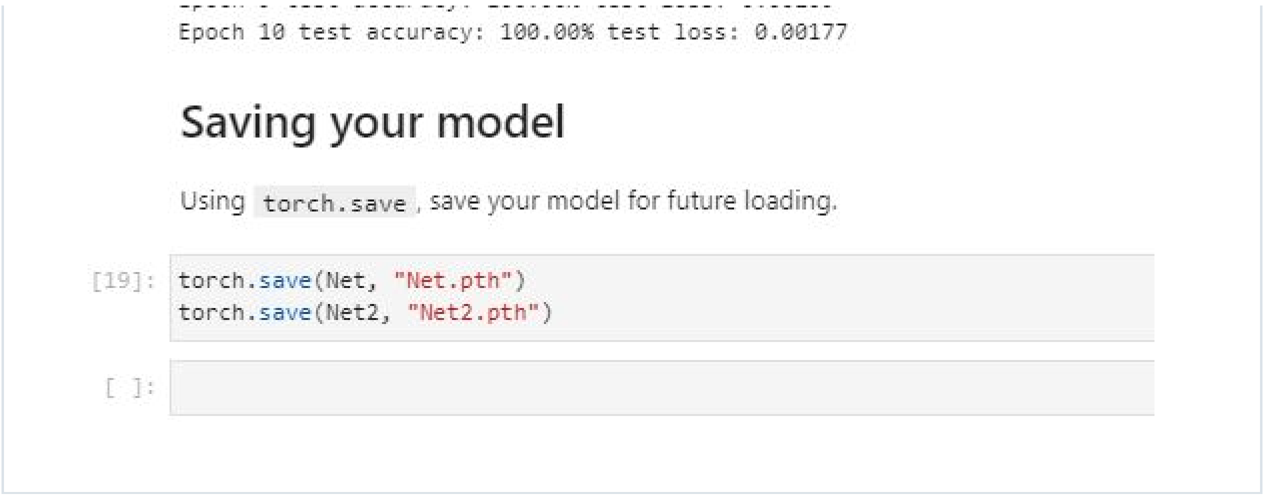
```
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.activation = F.relu
        self.layer1 = nn.Linear(28 * 28, 128)
        self.layer2 = nn.Linear(128, 64)
        self.layer3 = nn.Linear(64, 10)

        self.dropout=nn.Dropout(0.2)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = self.activation(self.layer1(x))
        x = self.activation(self.layer2(x))
        x = self.layer3(x)
        return x

net2 = Net2()
```

[14]:
```
# Choose an optimizer
optimizer = optim.Adam(net2.parameters(), lr=0.001)

# Choose a loss function
criterion = nn.CrossEntropyLoss()
```

[15]:
```
num_epochs = 10

# Establish a list for our history
train_loss_history = list()
val_loss_history = list()
```

The `torch.save()` function is used to save the weights of the trained model.

- Good use of torch.save()
- load and load_save_dict are also important methods with some parameters when you want to load these pretrained models for re-training or inference so do check them out too :) https://pytorch.org/tutorials/beginner/saving_loading_models.html



```
Epoch 10 test accuracy: 100.00% test loss: 0.00177
```

Saving your model

Using `torch.save`, save your model for future loading.

```
[19]: torch.save(Net, "Net.pth")
      torch.save(Net2, "Net2.pth")
```

```
[ ]:
```

DOWNLOAD PROJECT

RETURN TO PATH