Lab 4 Report:

Surpass Human Performance in Fashion MNIST Classificaion

Name: Forest Tschirhart

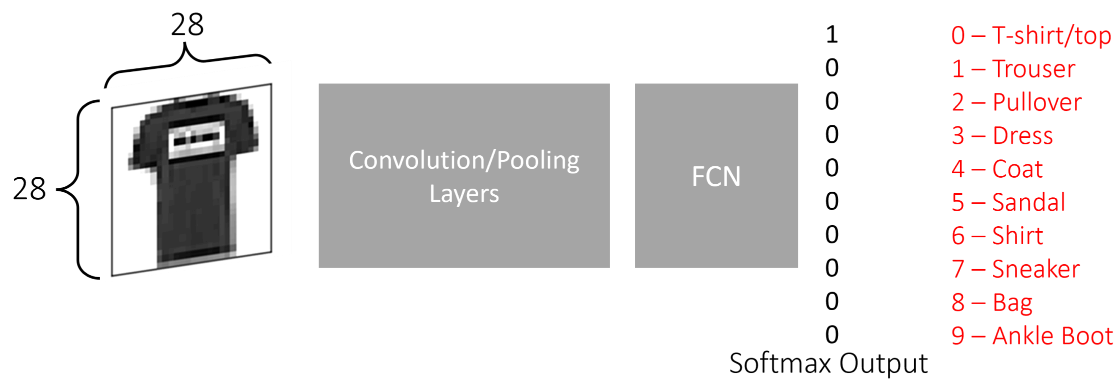
In [187…
```
%matplotlib inline
import matplotlib.pyplot as plt
import torch
import numpy as np
```

In [188…
```
from IPython.display import Image # For displaying images in colab jupyter cell
```

In [189…
```
Image('lab4_exercise.png', width = 1000)
```

Out[189…

Surpass Human Performance in Fashion MNIST Classification



In this exercise, you will classify fashion item class (28 x 28) using your own **Convolutional Neural Network Architecture.**

Prior to training your neural net, 1) Normalize the dataset using standard scaler and 2) Split the dataset into train/validation/test.

Design your own CNN architecture with your choice of Convolution/Pooling/FCN layers, activation functions, optimization method etc.

Your goal is to **achieve a testing accuracy of >89%**, with no restrictions on epochs **(Human performance: 83.5%).**

Demonstrate the performance of your model via plotting the **training loss, validation accuracy** and printing out the **testing accuracy.**

After your model has reached the goal, print the accuracy in each class. What is the class that your model performed the worst? 44

Prepare Data

In [190…

```
# Load Fashion-MNIST Dataset in Numpy

# 10000 training features/targets where each feature is a greyscale image with shape (28, 28)
train_features = np.load('fashion_mnist_train_features.npy')
train_targets = np.load('fashion_mnist_train_targets.npy')

# 1000 testing features/targets
test_features = np.load('fashion_mnist_test_features.npy')
test_targets = np.load('fashion_mnist_test_targets.npy')

# Let's see the shapes of training/testing datasets
```

```
print("Training Features Shape: ", train_features.shape)
print("Training Targets Shape: ", train_targets.shape)
print("Testing Features Shape: ", test_features.shape)
print("Testing Targets Shape: ", test_targets.shape)
```

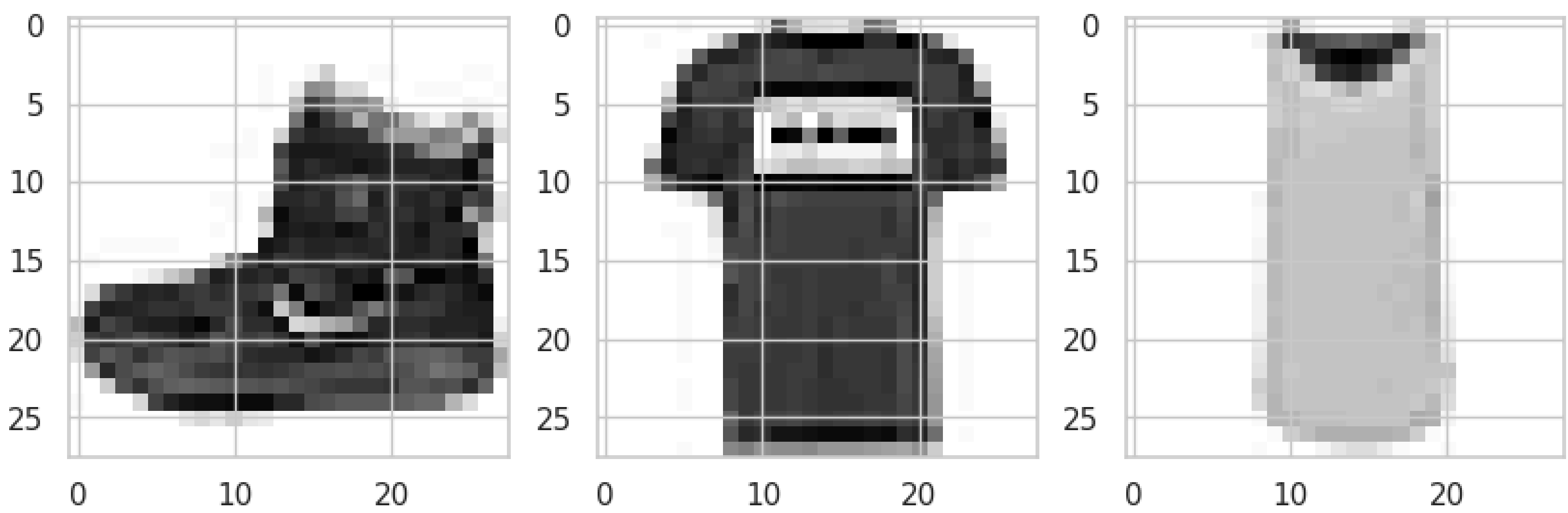
```
Training Features Shape:  (10000, 28, 28)
Training Targets Shape:  (10000,)
Testing Features Shape:  (1000, 28, 28)
Testing Targets Shape:  (1000,)
```

In [191…

```
# Visualizing the first three training features (samples)

plt.figure(figsize = (10, 10))

plt.subplot(1,3,1)
plt.imshow(train_features[0], cmap = 'Greys')

plt.subplot(1,3,2)
plt.imshow(train_features[1], cmap = 'Greys')

plt.subplot(1,3,3)
plt.imshow(train_features[2], cmap = 'Greys')
```

Out[191…

```
<matplotlib.image.AxesImage at 0x7d429050e090>
```



In [192…

```
# Reshape features via flattening the images
# This refers to reshape each sample from a 2d array to a 1d array.
# hint: np.reshape() function could be useful here

train_features = np.reshape(train_features, (train_features.shape[0], train_features.shape[1] * train_featu
test_features = np.reshape(test_features, (test_features.shape[0], test_features.shape[1] * test_features.s
print(train_features.shape)
print(test_features.shape)
```

```
(10000, 784)
(1000, 784)
```

In [193…

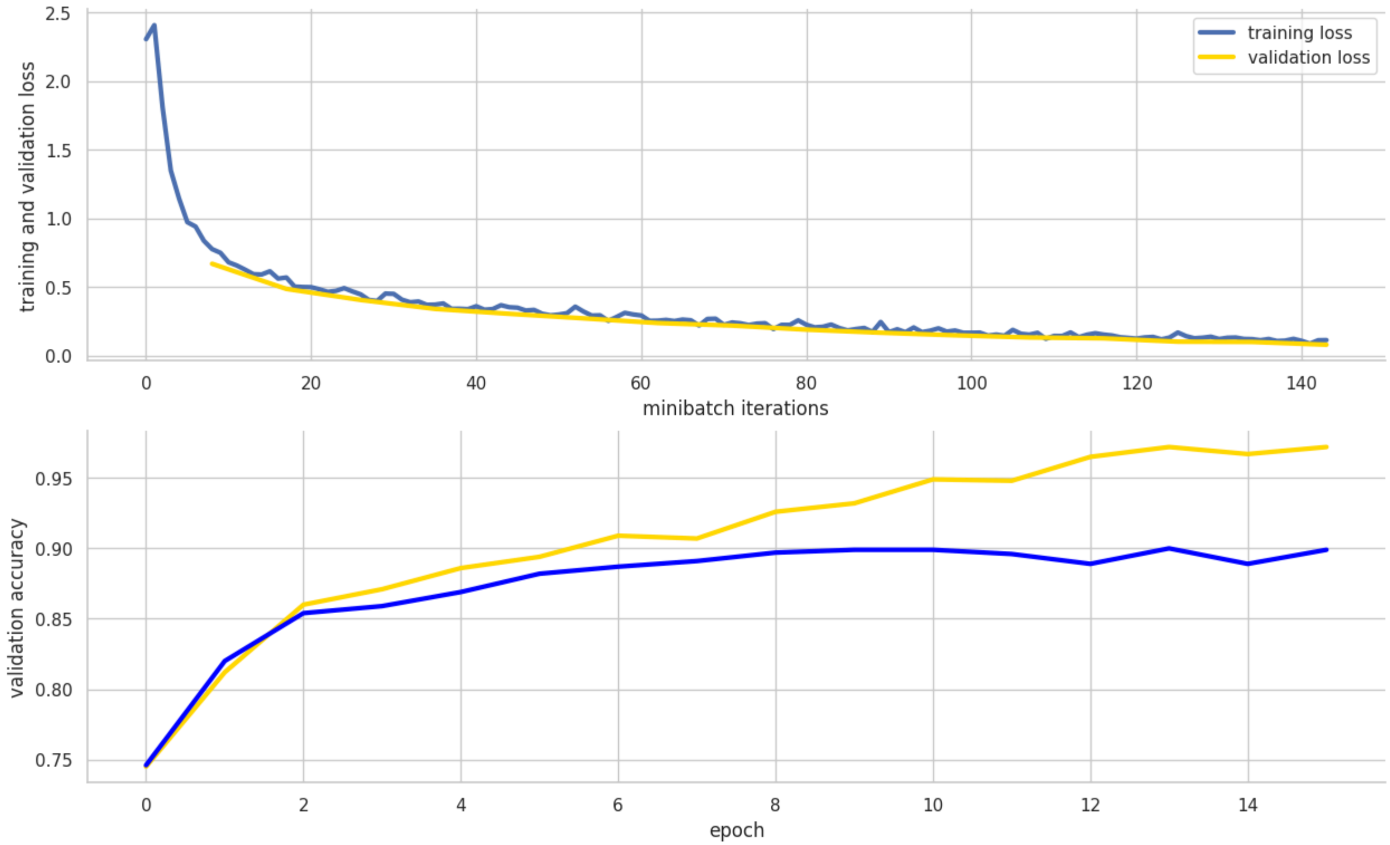
```
# Define your scaling function
from sklearn.preprocessing import StandardScaler

# Scale the dataset according to standard scaling
scaler = StandardScaler()

# EXPERIMENTING WITH SCALER FITTING TO SEE IF IT ELIMINATES THE TESTING VALIDATION ACCURACY DISCREPANCY --
# all_features = np.concatenate((train_features, test_features), axis = 0)
# print(all_features.shape)
# scaler.fit(all_features)
# train_features = scaler.transform(train_features)
# test_features = scaler.transform(test_features)


train_features = scaler.fit_transform(train_features)
test_features = scaler.transform(test_features)
```

I observed there was a consistently poor testing accuracy compared to validation accuracy. I wondered if it was due to fitting the scaler to only the testing data, resulting in a slight difference in how well the scaling fit the testing data. I fit the scaler to the combined testing and training data and it did not eliminate this issue. The image below shows both validation and testing accuracy tracked over the training process, exhibiting this laggin behavior. The blue accuracy curve is testing, the orange is validation.

In [194…

```
Image('validation_testing_discrepancy.png', width = 700)
```

Out[194…



In [195…

```
# Take the first 1000 (or randomly select 1000) training features and targets as validation set

validation_features = train_features[:1000] # YOUR CODE HERE
validation_targets = train_targets[:1000] # YOUR CODE HERE

# Take the remaining 9000 training features and targets as training set

train_features = train_features[1000:] # YOUR CODE HERE
train_targets = train_targets[1000:] # YOUR CODE HERE


# SLICING DIFFERENT PARTS FOR THE VALIDATION SET -----------------------------------------------------
# train_features = train_features[:-1000] # YOUR CODE HERE
# train_targets = train_targets[:-1000] # YOUR CODE HERE
```

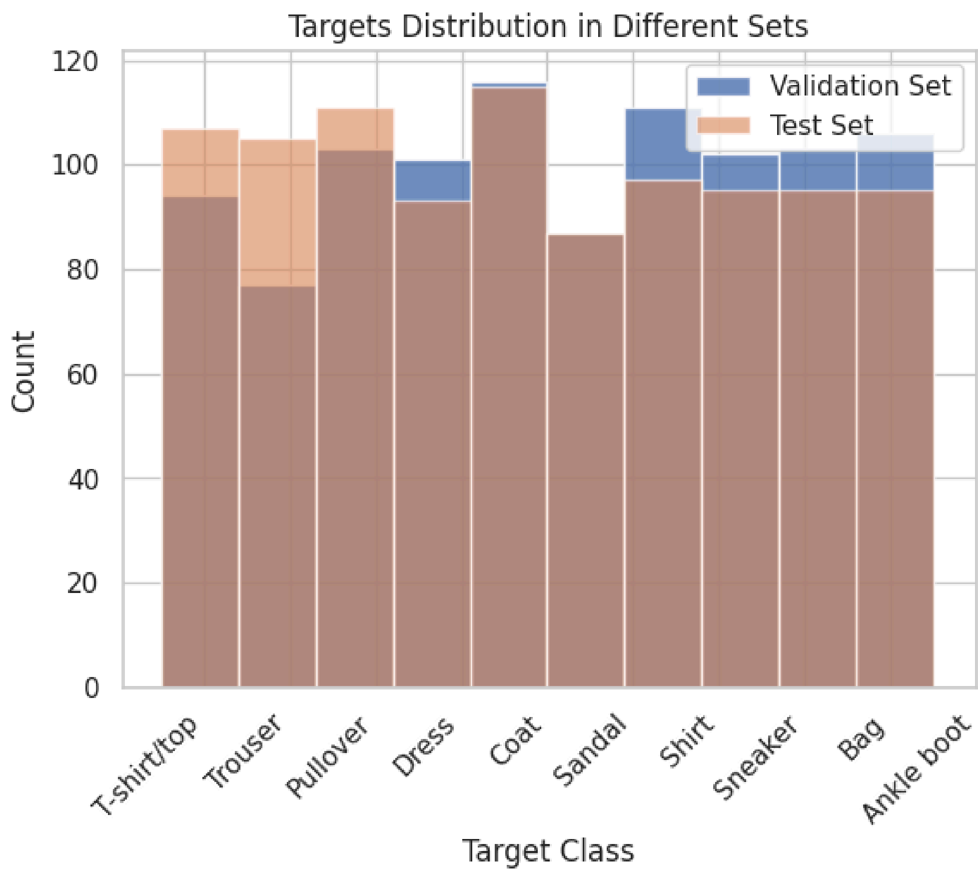
```
# # Take the remaining 9000 training features and targets as training set

# validation_features = train_features[-1000:] # YOUR CODE HERE
# validation_targets = train_targets[-1000:] # YOUR CODE HERE
```

After the fitting alteration did not fix the validation testing discrepancy I tried sourcing the validation data from different parts of the training data set. This worked, and I assume this is due to a significant difference in the distribution of each class within the validation and testing sets. I confirmed there was a slight difference by plotting a histogram of the different classes within the two sets. Below shows a histogram of this discrepancy. I theorize that the randomly low amount of tshirts/tops nd trousers in the validation set allowed it to boost performance. Maybe it frequenly confuses these classes with something else.

In [196…

```
Image('distro_discrep.png', width = 500)
```

Out[196…



In [197…

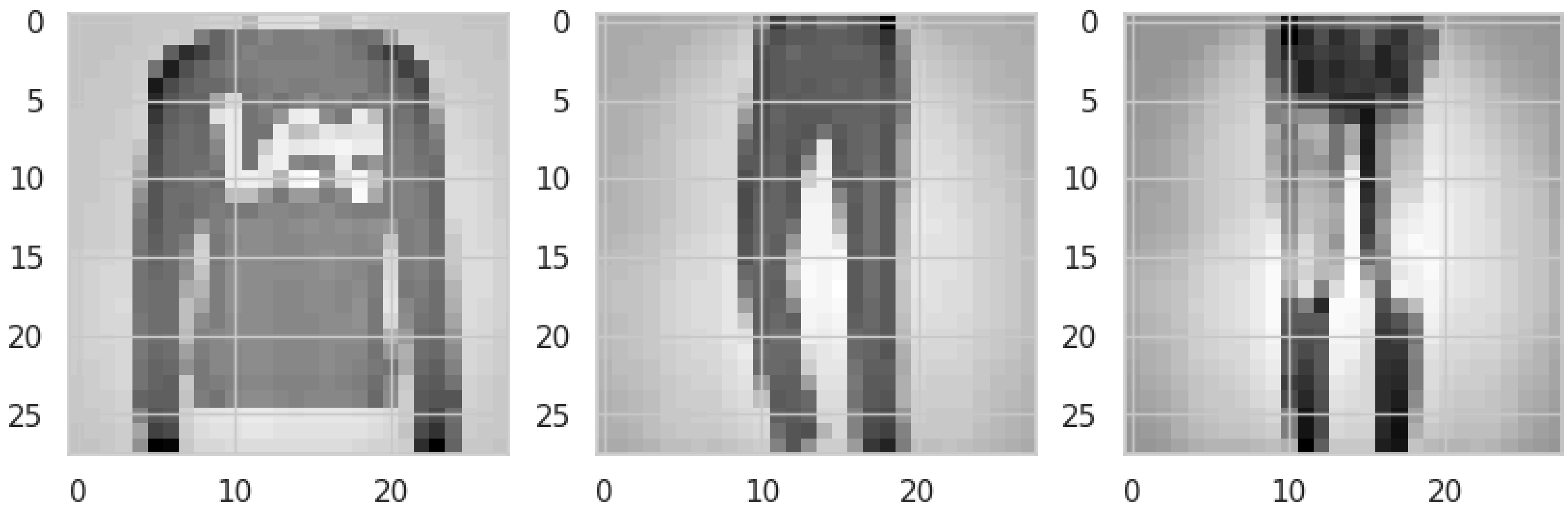
```
# Reshape train/validation/test sets to conform to PyTorch's (N, Channels, Height, Width) standard for CNN.

train_features = np.reshape(train_features, (train_features.shape[0], 28, 28))
validation_features = np.reshape(validation_features, (validation_features.shape[0], 28, 28))
test_features = np.reshape(test_features, (test_features.shape[0], 28, 28))
```

In [198…

```
# Visualizing the scaling effect
plt.figure(figsize = (10, 10))

plt.subplot(1,3,1)
plt.imshow(test_features[1], cmap = 'Greys')

plt.subplot(1,3,2)
plt.imshow(test_features[2], cmap = 'Greys')

plt.subplot(1,3,3)
plt.imshow(test_features[3], cmap = 'Greys')
```

Out[198…

```
<matplotlib.image.AxesImage at 0x7d428f909040>
```

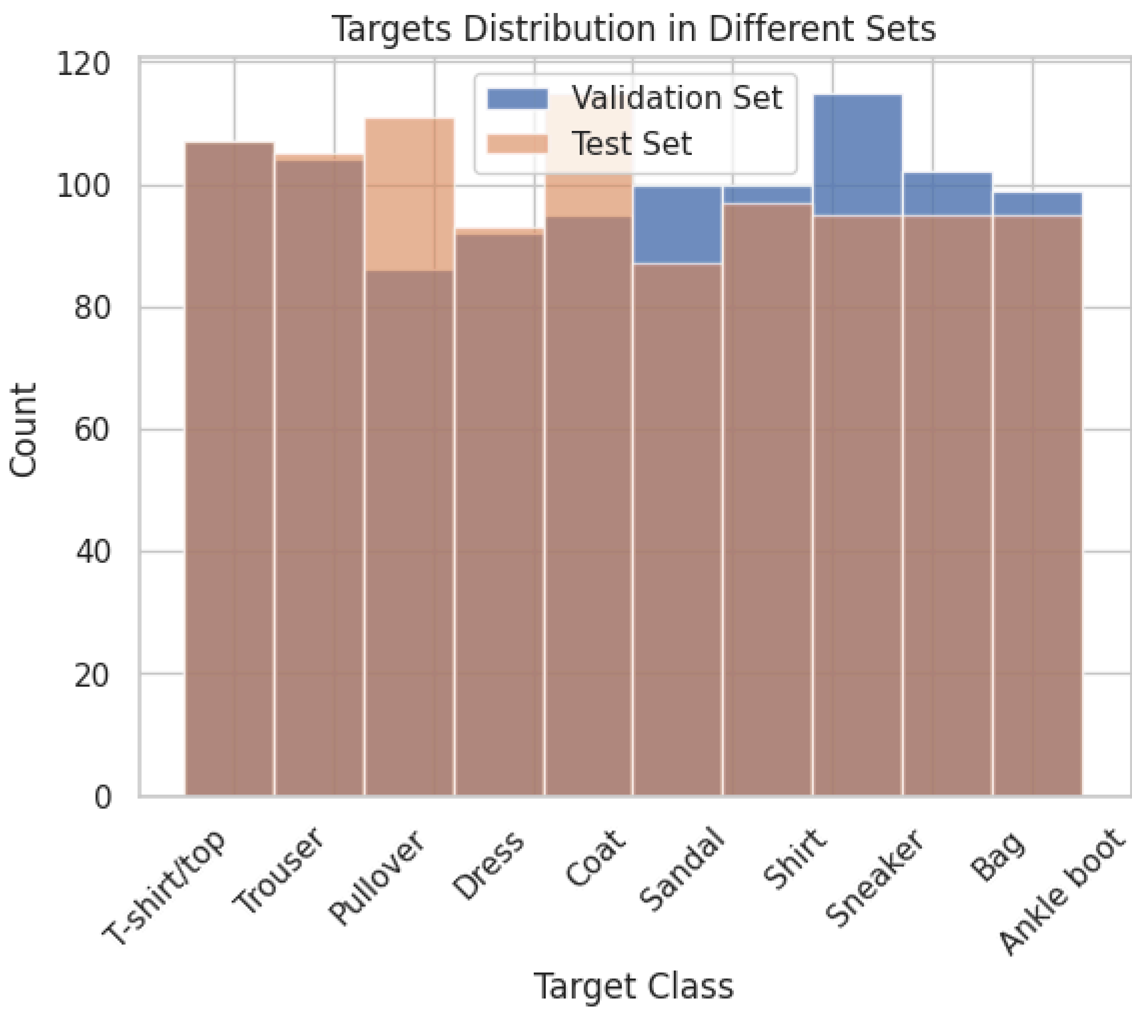


In [199…

```
# Make sure validation and test sets are moslty uniform distributions. Was dealing with consistently worse
# testing accuracy than validation accuracy which wasn't making sense to me.
# This particular slice choice seems to have fixed it, and both distributions are now more uniform.
```

```
classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Anl
plt.hist(validation_targets, bins = 10, alpha = 0.8, label = 'Validation Set')
plt.hist(test_targets, bins = 10, alpha = 0.6, label = 'Test Set')
plt.xlabel('Target Class')
plt.ylabel('Count')
plt.title('Targets Distribution in Different Sets')
plt.xticks(ticks=np.arange(len(classes)) + 0.5, labels=classes, rotation=45, ha='right')
plt.legend()
```

Out[199… <matplotlib.legend.Legend at 0x7d42907ba630>



Define Model

In [200…

```
# Define your CNN architecture here
# Model designed to be flexible to different architectures for testing purposes

class CNNModel(torch.nn.Module):

    def __init__(self, arch, params, output_dimension):

        # temporary function calculates proper initailization dimensions for different architectures
        # based on the kernel size, stride, and padding
        def out_layer_dim(input, k, s, p):
            return (input+2*p-k)//s + 1

        # Apply above func over and over to calculate final flattened dimension before feeding into
        # the fully connected layer
        # Achritecture keys use C for conv layer and P for pooling layer
        # Example: CPCP = Conv -> Pool -> Conv -> Pool
        if arch == 'CPCP':
            dimadome = out_layer_dim(28, params['k1'], params['s1'], params['p1'])
            dimadome = out_layer_dim(dimadome, params['pool1_k'], params['pool1_s'], params['pool1_p'])
            dimadome = out_layer_dim(dimadome, params['k2'], params['s2'], params['p2'])
            dimadome = out_layer_dim(dimadome, params['pool2_k'], params['pool2_s'], params['pool2_p'])
        elif arch == 'CCP':
            dimadome = out_layer_dim(28, params['k1'], params['s1'], params['p1'])
            dimadome = out_layer_dim(dimadome, params['k2'], params['s2'], params['p2'])
            dimadome = out_layer_dim(dimadome, params['pool1_k'], params['pool1_s'], params['pool1_p'])
        else:
            raise ValueError("Invalid architecture. Choose from 'CPCP', 'CCP'.")

        super(CNNModel, self).__init__()
        self.arch = arch
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = params['out1_ch'],
                                     kernel_size = params['k1'], stride = params['s1'],
                                     padding = params['p1'])
        self.conv2 = torch.nn.Conv2d(in_channels = params['out1_ch'], out_channels = params['out2_ch'],
                                     kernel_size = params['k2'], stride = params['s2'],
                                     padding = params['p2'])
        self.pool1 = torch.nn.MaxPool2d(kernel_size = params['pool1_k'], stride = params['pool1_s'],
                                        padding = params['pool1_p'])
```

```
        self.pool2 = torch.nn.MaxPool2d(kernel_size = params['pool2_k'], stride = params['pool2_s'],
                                        padding = params['pool2_p'])
        self.fc1 = torch.nn.Linear(params['out2_ch'] * dimadome**2, 128)
        self.fc2 = torch.nn.Linear(128, output_dimension)
        self.activation = torch.nn.ReLU()
        self.dropout = torch.nn.Dropout(p = 0.3)

    # def data flow for different architectures
    def forward(self, x):
        if self.arch == 'CPCP':
            out = self.activation(self.conv1(x))
            out = self.pool1(out)
            out = self.activation(self.conv2(out))
            out = self.pool2(out)
            out = out.view(out.size(0), -1)
            out = self.activation(self.fc1(out))
            out = self.dropout(out)
            out = self.fc2(out)
        elif self.arch == 'CCP':
            out = self.activation(self.conv1(x))
            out = self.activation(self.conv2(out))
            out = self.pool2(out)
            out = out.view(out.size(0), -1)
            out = self.activation(self.fc1(out))
            out = self.dropout(out)
            out = self.fc2(out)
        return out
```

Select Hyperparameters

In [201…

```
# Fix the random seed so that model performance is reproducible
torch.manual_seed(55)

# Initializ CNN model paramters
p_nasty = {
    'out1_ch': 32,'k1': 3, 's1': 1, 'p1': 1,
    'out2_ch': 64,'k2': 3, 's2': 1, 'p2': 1,
    'pool1_k': 3, 'pool1_s': 2, 'pool1_p': 0,
    'pool2_k': 3, 'pool2_s': 2, 'pool2_p': 0,
```

```
}

# Initialize CNN model
# CCP stands for Convolutional-Convolutional-Pooling and uses pool_2 params for P
# CPCP stands for Convolutional-Pooling-Convolutional-Pooling
model = CNNModel(arch='CCP', params=p_nasty, output_dimension = 10)


# Define learning rate, epoch and batchsize for mini-batch gradient

learning_rate = 0.002
epochs = 13
batchsize = 1000

# Define loss function and optimizer

loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

model
```

Out[201…

```
CNNModel(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (pool2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=10816, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
  (activation): ReLU()
  (dropout): Dropout(p=0.3, inplace=False)
)
```

Identify Tracked Values

In [202…

```
# Placeholders for training loss and validation accuracy during training
# Training loss should be tracked for each iteration (1 iteration -> single forward pass to the network)
# Validation accuracy should be evaluated every 'Epoch' (1 epoch -> full training dataset)
# If using batch gradient, 1 iteration = 1 epoch

train_loss_list = []
```

```
train_loss_axis = [] # used for plotting
val_axis = [] # used for plotting
val_loss_list = [] # not used in this lab ususally but I like it for diagnosing overfitting
validation_accuracy_list = []

# test_accuracy_list = [] #sanity check not used in final results
```

Train Model

In [203…

```
import tqdm # Use "for epoch in tqdm.trange(epochs):" to see the progress bar

# Convert the training, validation, testing dataset (NumPy arrays) into torch tensors
# Split your training features/targets into mini-batches if using mini-batch gradient

train_features = torch.from_numpy(train_features).float()
train_targets = torch.from_numpy(train_targets).long()
validation_features = torch.from_numpy(validation_features).float()
validation_targets = torch.from_numpy(validation_targets).long()
test_features = torch.from_numpy(test_features).float()
test_targets = torch.from_numpy(test_targets).long()

# Combine features and targets into a dataset
train_dataset = torch.utils.data.TensorDataset(train_features, train_targets)
# Create DataLoader with shuffling
batch_train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batchsize, shuffle=True)
i=0
#Training Loop -------------------------------------------------------------------------------------
for epoch in tqdm.trange(epochs, desc='Training Epochs'):
    for batch_features, batch_targets in batch_train_loader: # batch loader takes care of iterating
        temp_features = batch_features.unsqueeze(1)
        optimizer.zero_grad() # Reset gradients
        outputs = model(temp_features)
        loss = loss_func(outputs, batch_targets)
        loss.backward()
        optimizer.step()
        train_loss_list.append(loss.item()) # append the loss for each mb iteration
        train_loss_axis.append(i) # append and increment the axis for plotting
        i +=1

    # Compute Validation Accuracy & Loss-------------------------------------------------------------------
```

```
    with torch.no_grad():
        model.eval() # Set the model to evaluation mode
        temp_features = validation_features.unsqueeze(1)
        outputs = model(temp_features)
        val_loss = loss_func(outputs, validation_targets)
        val_loss_list.append(val_loss.item())
        _, predicted = torch.max(outputs.data, 1)
        correct = (predicted == validation_targets).sum().item() # simple way to calculate accuracy
        accuracy = correct / validation_targets.size(0)
        validation_accuracy_list.append(accuracy)
        i-=1
        val_axis.append(i) # decrement append increment axis for plotting correctly
        i+=1

         # Compute Test Accuracy --Sanity check not used in final results
        # temp_features = test_features.unsqueeze(1)
        # outputs = model(temp_features)
        # _, predicted = torch.max(outputs.data, 1)
        # correct = (predicted == test_targets).sum().item()
        # accuracy = correct / test_targets.size(0)
        # test_accuracy_list.append(accuracy)

        model.train()
```

```
Training Epochs:   0%|          | 0/13 [00:00<?, ?it/s]
Training Epochs: 100%|██████████| 13/13 [02:12<00:00, 10.16s/it]
```

Visualize & Evaluate Model

In [204…

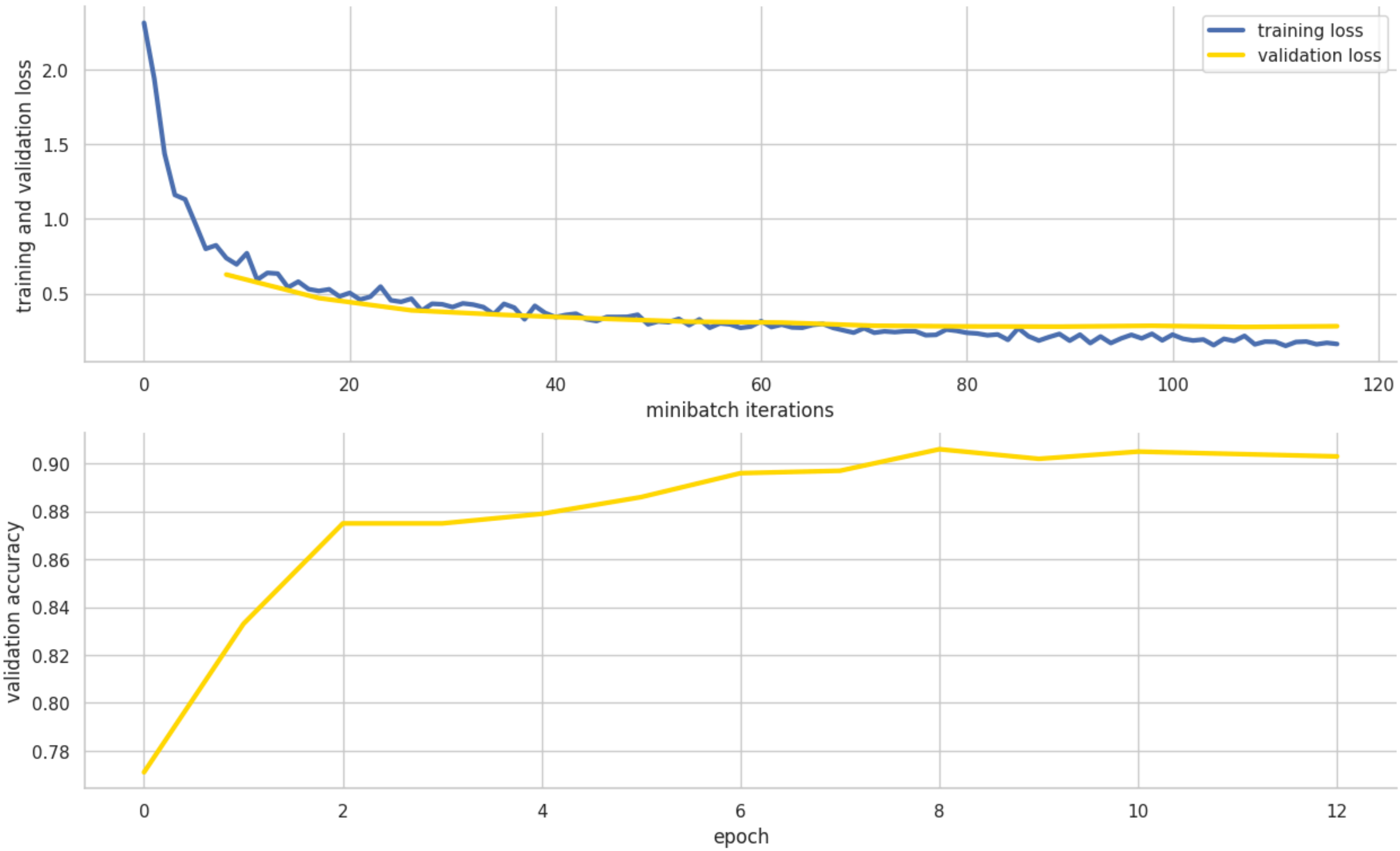
```
# Seaborn for prettier plot

import seaborn as sns

sns.set(style = 'whitegrid', font_scale = 1)
```

In [205…

```
# Visualize training loss

print("Val Acc: ", validation_accuracy_list)
print("Validation Loss: ", val_axis)
```

```
plt.figure(figsize = (15, 9))

plt.subplot(2, 1, 1)
plt.plot(train_loss_axis, train_loss_list, linewidth = 3, label = 'training loss')
# use x axes computed in training loop
plt.plot(val_axis, val_loss_list, linewidth = 3, color = 'gold', label = 'validation loss')
plt.ylabel("training and validation loss")
plt.xlabel("minibatch iterations")
plt.legend()
sns.despine()

plt.subplot(2, 1, 2)
plt.plot(validation_accuracy_list, linewidth = 3, color = 'gold')
# plt.plot(test_accuracy_list, linewidth = 3, color = 'blue') # sanity check not used in final results
plt.ylabel("validation accuracy")
plt.xlabel("epoch")
sns.despine()
```

```
Val Acc:  [0.771, 0.833, 0.875, 0.875, 0.879, 0.886, 0.896, 0.897, 0.906, 0.902, 0.905, 0.904, 0.903]
Validation Loss:  [8, 17, 26, 35, 44, 53, 62, 71, 80, 89, 98, 107, 116]
```



In [206…

```
# Compute the testing accuracy

with torch.no_grad():
    model.eval()
    temp_features = test_features.unsqueeze(1)
    outputs = model(temp_features)
    _, predicted = torch.max(outputs.data, 1)
    correct = (predicted == test_targets).sum().item()
    accuracy = correct / test_targets.size(0)
```

```
    print("Testing Accuracy: ", accuracy)
    model.train()
```

```
Testing Accuracy:  0.892
```

In [207…

```
# (OPTIONAL) Print the testing accuracy for each fashion class. Your code should produce something that lo
# Clever usage of np.where() could be useful here

predicted_np = predicted.numpy()
test_targets_np = test_targets.numpy()

truth_array_full = test_targets_np==predicted_np # boolean array of prediction truth with correlated indic


accuracies = []
for i in range(10): # iterate over the 10 classes
    class_indices = np.where(test_targets_np==i)[0] # find all indices of the ith class in test set
    class_truth_array = truth_array_full[class_indices] # take all true/false values corresponding to the 
    class_acc = class_truth_array.sum()/class_truth_array.shape[0] # all true / total
    print(f"Accuracy of {classes[i]}: {class_acc*100:.2f} %")
    accuracies.append(class_acc) # save accuracy value
# What's the fashion item that your model had the hardest time classifying?
print(f"Worst class: {classes[np.argmin(accuracies)]}") # find worst class index and print classes_list co
print(f"Model doesn't know the difference between a {classes[np.argmin(accuracies)]} and a {classes[np.arg
```

```
Accuracy of T-shirt/top: 76.64 %
Accuracy of Trouser: 97.14 %
Accuracy of Pullover: 87.39 %
Accuracy of Dress: 87.10 %
Accuracy of Coat: 80.00 %
Accuracy of Sandal: 96.55 %
Accuracy of Shirt: 80.41 %
Accuracy of Sneaker: 97.89 %
Accuracy of Bag: 97.89 %
Accuracy of Ankle boot: 94.74 %
Worst class: T-shirt/top
Model doesn't know the difference between a T-shirt/top and a Dress lmao idiot
```

In [ ]: