Lab 4 Report:

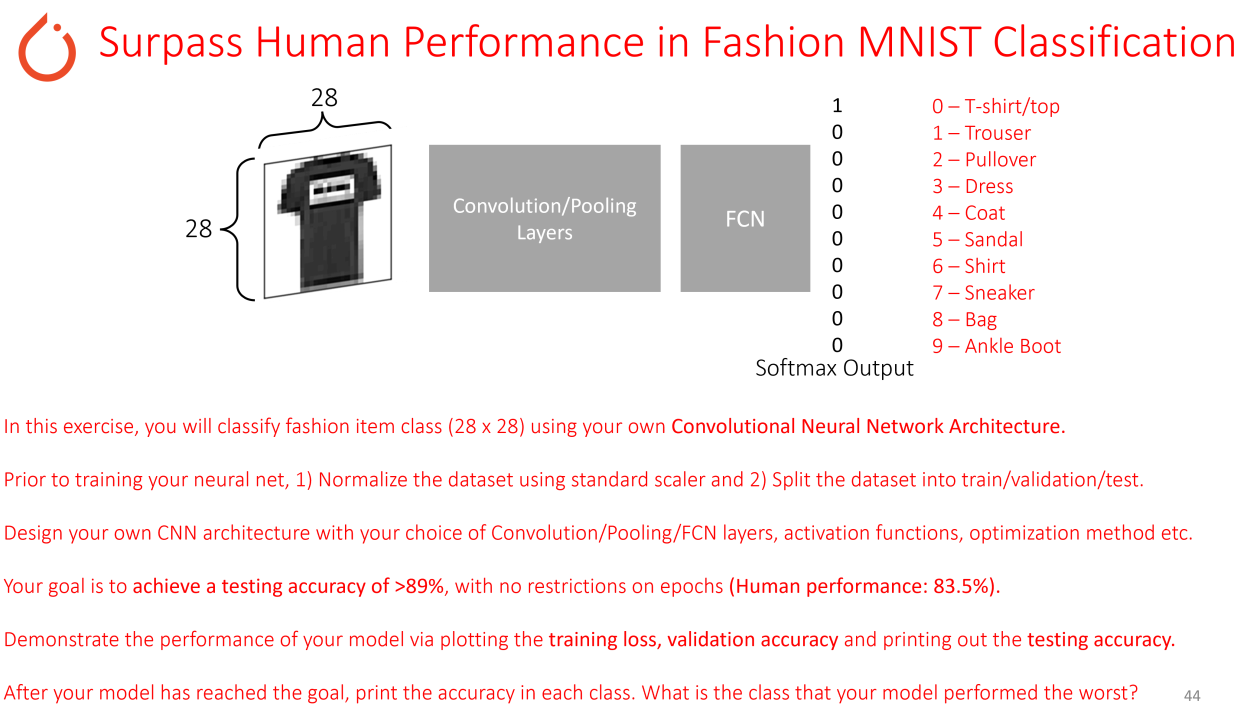
Surpass Human Performance in Fashion MNIST Classificaion

Name: Travis Hand

```
%matplotlib inline
import matplotlib.pyplot as plt
import torch
import numpy as np
```

```
from IPython.display import Image # For displaying images in colab jupyter cell
```

```
Image('lab4_exercise.png', width = 1000)
```



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?

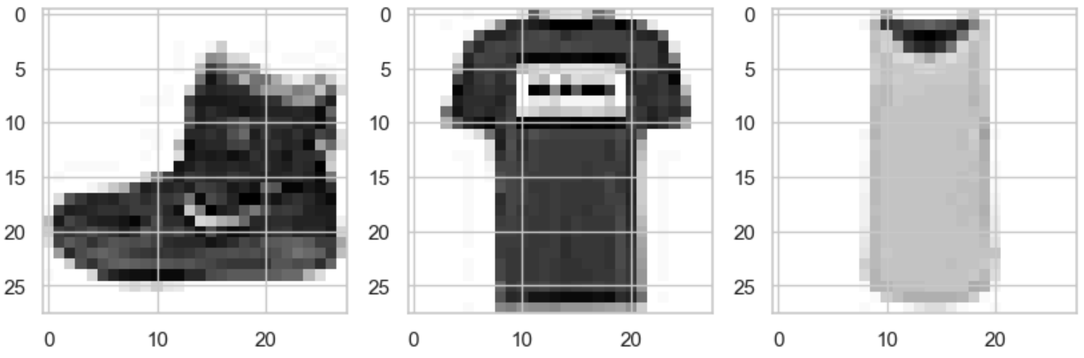
Prepare Data

```
# Load Fashion-MNIST Dataset in Numpy
import os
# 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,)
```

```
# 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')
```

```
<matplotlib.image.AxesImage at 0x165b60920>
```



```
# 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 = train_features.reshape(10000, 784)
test_features = test_features.reshape(1000, 784)
```

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

scaler = StandardScaler()
# Scale the dataset according to standard scaling
train_features = scaler.fit_transform(train_features)
test_features = scaler.fit_transform(test_features)
```

```
from sklearn.model_selection import train_test_split
# Take the first 1000 (or randomly select 1000) training features and targets as validation set
# Take the remaining 9000 training features and targets as training set
# Use train_test_split to efficiently split the training targets and tests into validation with 9000:1000 ratio and randomly shu
train_features, validation_features, train_targets, validation_targets =  train_test_split(train_features, train_targets, test_s
```

```
# Reshape train/validation/test sets to conform to PyTorch's (N, Channels, Height, Width) standard for CNNs
train_features = train_features.reshape(9000, 1, 28, 28)
validation_features = validation_features.reshape(1000, 1, 28, 28)
test_features = test_features.reshape(1000, 1, 28, 28)
```

Define Model

```
# Define your CNN architecture here

class CNNModel(torch.nn.Module):

    def __init__(self):

        super(CNNModel, self).__init__()

        # Two convolutional layers with 32 and 64 filters respectively
        # Each convolutional layer is followed by a ReLU activation function
        self.conv_layer1 = torch.nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1) # add stride as well? might
        self.conv_layer2 = torch.nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1) # mayb increase padding to

        # Pooling layer to streamline model
        self.pool = torch.nn.MaxPool2d(2, 2)
        # Fully connected layers for understanding feature relationships
        self.fc_layer1 = torch.nn.Linear(64 * 7 * 7, 128)
        self.fc_layer2 = torch.nn.Linear(128, 10)


    def forward(self, x):

        # Note: If you are using CrossEntropyLoss() do NOT apply softmax to the final ouput
        # since it's incorporated within the loss function
        x = self.pool(torch.nn.functional.relu(self.conv_layer1(x)))   # [N, 32, 14, 14]
        x = self.pool(torch.nn.functional.relu(self.conv_layer2(x)))   # [N, 64, 7, 7]
        x = x.view(x.size(0), -1)             # Flatten: [N, 64*7*7]
        x = torch.nn.functional.relu(self.fc_layer1(x))
        x = self.fc_layer2(x)
        return x
```

Select Hyperparameters

```
# Fix the random seed so that model performance is reproducible
torch.manual_seed(69)

# Initialize your CNN model

model = CNNModel()

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

learning_rate = 0.001
epochs = 33
batchsize = 50

# Define loss function and optimizer

loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate, weight_decay=1e-4)

model
```

```
CNNModel(
  (conv_layer1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv_layer2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc_layer1): Linear(in_features=3136, out_features=128, bias=True)
  (fc_layer2): Linear(in_features=128, out_features=10, bias=True)
)
```

Identify Tracked Values

```
# 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 = []
validation_accuracy_list = []
```

Train Model

```
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

test_features = torch.tensor(test_features, dtype=torch.float32)
test_targets = torch.tensor(test_targets, dtype=torch.long)
train_features = torch.tensor(train_features, dtype=torch.float32)
train_targets = torch.tensor(train_targets, dtype=torch.long)
validation_features = torch.tensor(validation_features, dtype=torch.float32)
validation_targets = torch.tensor(validation_targets, dtype=torch.long)
```

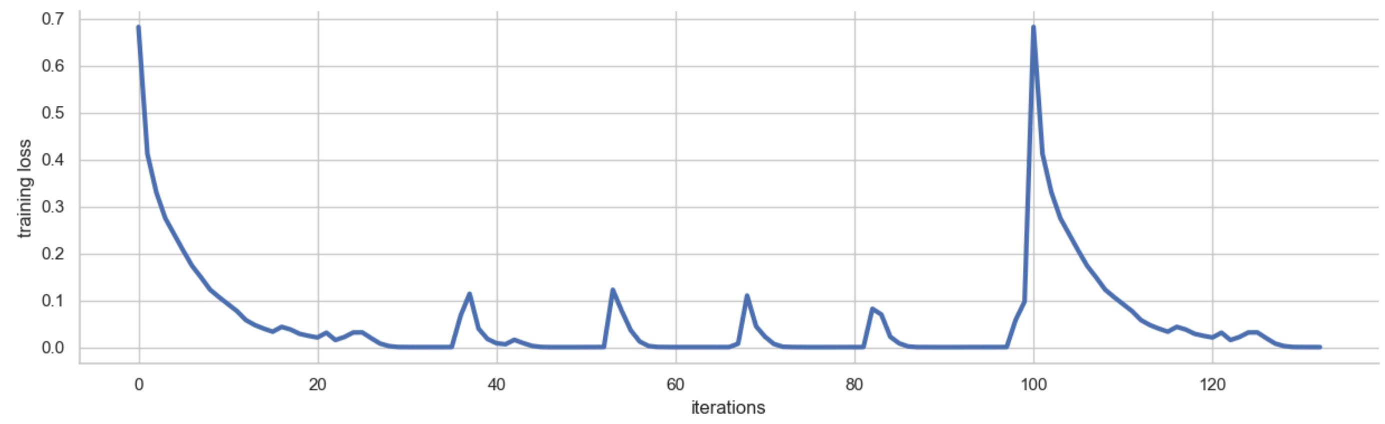
```
# Create TensorDatasets (feature, target)
train = torch.utils.data.TensorDataset(train_features, train_targets)
validation = torch.utils.data.TensorDataset(validation_features, validation_targets)
test = torch.utils.data.TensorDataset(test_features, test_targets)

# Create data loaders, using batches for more efficient training
train_loader = torch.utils.data.DataLoader(train, batch_size=batchsize, shuffle=True)
val_loader = torch.utils.data.DataLoader(validation, batch_size=batchsize)
test_loader = torch.utils.data.DataLoader(test, batch_size=batchsize)

# select hardware based on availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
if torch.cuda.is_available():
    model.cuda()


# Training Loop -----------------------------------------------------------------------------------------

for epoch in tqdm.trange(epochs):

    model.train()
    train_loss = 0
    correct = 0
    total = 0

    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)

        loss = loss_func(outputs, targets)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * inputs.size(0)
        predicted = torch.argmax(outputs, 1)
        correct += (predicted == targets).sum().item()
        total += targets.size(0)

    avg_train_loss = train_loss / total
    accuracy = correct / total
    train_loss_list.append(avg_train_loss)

    # Compute Validation Accuracy ---------------------------------------------------------------------

    model.eval()
    val_loss = 0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for val_inputs, val_targets in val_loader:
            val_inputs, val_targets = val_inputs.to(device), val_targets.to(device)

            val_outputs = model(val_inputs)
            loss = loss_func(val_outputs, val_targets)

            val_loss += loss.item() * val_inputs.size(0)
            val_predicted = torch.argmax(val_outputs, dim=1)
            val_correct += (val_predicted == val_targets).sum().item()
            val_total += val_targets.size(0)

    avg_val_loss = val_loss / val_total
    val_accuracy = val_correct / val_total
    validation_accuracy_list.append(val_accuracy)

    print(f"Epoch {epoch+1}/{epochs} - "
          f"Train Loss: {avg_train_loss:.4f}, Accuracy: {accuracy:.4f} - "
          f"Val Loss: {avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.4f}")
```

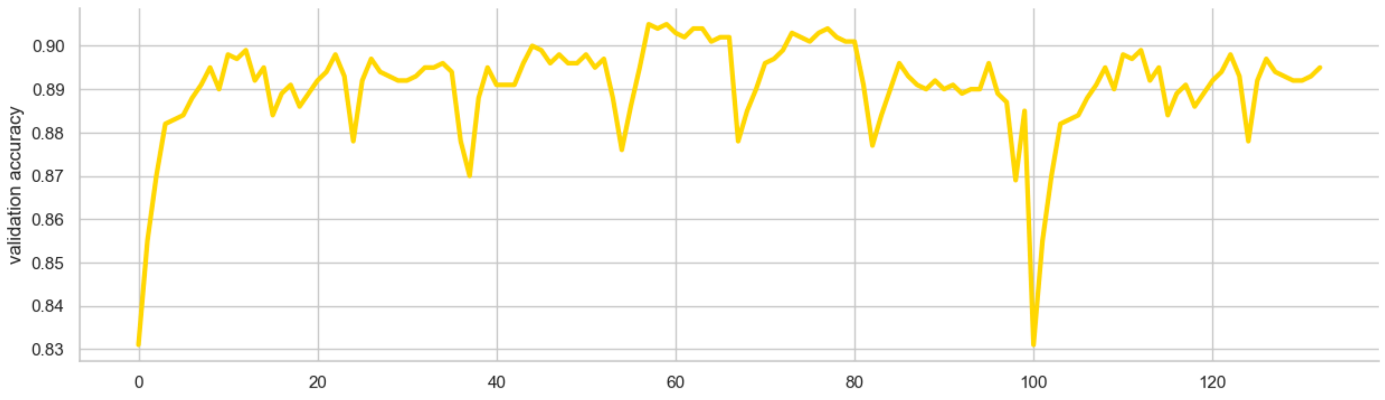
```
/var/folders/nn/yzl4s0_97y791sb31vclsv7w0000gn/T/ipykernel_77070/2139800984.py:6: UserWarning: To copy construct from a tens
  test_features = torch.tensor(test_features, dtype=torch.float32)
/var/folders/nn/yzl4s0_97y791sb31vclsv7w0000gn/T/ipykernel_77070/2139800984.py:7: UserWarning: To copy construct from a tens
  test_targets = torch.tensor(test_targets, dtype=torch.long)
/var/folders/nn/yzl4s0_97y791sb31vclsv7w0000gn/T/ipykernel_77070/2139800984.py:8: UserWarning: To copy construct from a tens
```

```
  train_features = torch.tensor(train_features, dtype=torch.float32)
/var/folders/nn/yzl4s0_97y791sb31vclsv7w0000gn/T/ipykernel_77070/2139800984.py:9: UserWarning: To copy construct from a tens
  train_targets = torch.tensor(train_targets, dtype=torch.long)
/var/folders/nn/yzl4s0_97y791sb31vclsv7w0000gn/T/ipykernel_77070/2139800984.py:10: UserWarning: To copy construct from a ten
  validation_features = torch.tensor(validation_features, dtype=torch.float32)
/var/folders/nn/yzl4s0_97y791sb31vclsv7w0000gn/T/ipykernel_77070/2139800984.py:11: UserWarning: To copy construct from a ten
  validation_targets = torch.tensor(validation_targets, dtype=torch.long)
  3%|▎         | 1/33 [00:03<01:53,  3.55s/it]Epoch 1/33 - Train Loss: 0.6822, Accuracy: 0.7553 - Val Loss: 0.4578, Val Accu
  6%|▌         | 2/33 [00:06<01:45,  3.42s/it]Epoch 2/33 - Train Loss: 0.4120, Accuracy: 0.8542 - Val Loss: 0.4199, Val Accu
  9%|▉         | 3/33 [00:10<01:39,  3.32s/it]Epoch 3/33 - Train Loss: 0.3298, Accuracy: 0.8807 - Val Loss: 0.3747, Val Accu
 12%|█▏        | 4/33 [00:13<01:34,  3.28s/it]Epoch 4/33 - Train Loss: 0.2749, Accuracy: 0.8996 - Val Loss: 0.3422, Val Accu
 15%|█▌        | 5/33 [00:16<01:30,  3.23s/it]Epoch 5/33 - Train Loss: 0.2406, Accuracy: 0.9108 - Val Loss: 0.3203, Val Accu
 18%|█▊        | 6/33 [00:19<01:26,  3.21s/it]Epoch 6/33 - Train Loss: 0.2060, Accuracy: 0.9249 - Val Loss: 0.3434, Val Accu
 21%|██        | 7/33 [00:22<01:22,  3.19s/it]Epoch 7/33 - Train Loss: 0.1733, Accuracy: 0.9370 - Val Loss: 0.3334, Val Accu
 24%|██▍       | 8/33 [00:25<01:19,  3.19s/it]Epoch 8/33 - Train Loss: 0.1489, Accuracy: 0.9451 - Val Loss: 0.3335, Val Accu
 27%|██▋       | 9/33 [00:29<01:16,  3.18s/it]Epoch 9/33 - Train Loss: 0.1229, Accuracy: 0.9533 - Val Loss: 0.3451, Val Accu
 30%|███       | 10/33 [00:32<01:13,  3.19s/it]Epoch 10/33 - Train Loss: 0.1070, Accuracy: 0.9589 - Val Loss: 0.3700, Val Ac
 33%|███▎      | 11/33 [00:35<01:10,  3.19s/it]Epoch 11/33 - Train Loss: 0.0921, Accuracy: 0.9664 - Val Loss: 0.3602, Val Ac
 36%|███▋      | 12/33 [00:38<01:07,  3.20s/it]Epoch 12/33 - Train Loss: 0.0773, Accuracy: 0.9720 - Val Loss: 0.3700, Val Ac
 39%|███▉      | 13/33 [00:41<01:03,  3.17s/it]Epoch 13/33 - Train Loss: 0.0581, Accuracy: 0.9798 - Val Loss: 0.4045, Val Ac
 42%|████▏     | 14/33 [00:44<00:59,  3.16s/it]Epoch 14/33 - Train Loss: 0.0475, Accuracy: 0.9840 - Val Loss: 0.4255, Val Ac
 45%|████▌     | 15/33 [00:48<00:56,  3.15s/it]Epoch 15/33 - Train Loss: 0.0399, Accuracy: 0.9884 - Val Loss: 0.4013, Val Ac
 48%|████▊     | 16/33 [00:51<00:53,  3.15s/it]Epoch 16/33 - Train Loss: 0.0334, Accuracy: 0.9891 - Val Loss: 0.5454, Val Ac
 52%|█████▏    | 17/33 [00:54<00:50,  3.18s/it]Epoch 17/33 - Train Loss: 0.0437, Accuracy: 0.9841 - Val Loss: 0.4787, Val Ac
 55%|█████▍    | 18/33 [00:57<00:47,  3.17s/it]Epoch 18/33 - Train Loss: 0.0380, Accuracy: 0.9878 - Val Loss: 0.4929, Val Ac
 58%|█████▊    | 19/33 [01:00<00:44,  3.17s/it]Epoch 19/33 - Train Loss: 0.0289, Accuracy: 0.9928 - Val Loss: 0.5302, Val Ac
 61%|██████    | 20/33 [01:03<00:41,  3.16s/it]Epoch 20/33 - Train Loss: 0.0245, Accuracy: 0.9932 - Val Loss: 0.5344, Val Ac
 64%|██████▎   | 21/33 [01:07<00:37,  3.16s/it]Epoch 21/33 - Train Loss: 0.0210, Accuracy: 0.9940 - Val Loss: 0.5587, Val Ac
 67%|██████▋   | 22/33 [01:10<00:34,  3.16s/it]Epoch 22/33 - Train Loss: 0.0313, Accuracy: 0.9892 - Val Loss: 0.5123, Val Ac
 70%|██████▉   | 23/33 [01:13<00:31,  3.15s/it]Epoch 23/33 - Train Loss: 0.0157, Accuracy: 0.9958 - Val Loss: 0.5500, Val Ac
 73%|███████▎  | 24/33 [01:16<00:28,  3.15s/it]Epoch 24/33 - Train Loss: 0.0221, Accuracy: 0.9938 - Val Loss: 0.6120, Val Ac
 76%|███████▌  | 25/33 [01:19<00:25,  3.15s/it]Epoch 25/33 - Train Loss: 0.0318, Accuracy: 0.9917 - Val Loss: 0.5160, Val Ac
 79%|███████▉  | 26/33 [01:22<00:22,  3.15s/it]Epoch 26/33 - Train Loss: 0.0320, Accuracy: 0.9890 - Val Loss: 0.5403, Val Ac
 82%|████████▏ | 27/33 [01:25<00:18,  3.16s/it]Epoch 27/33 - Train Loss: 0.0196, Accuracy: 0.9933 - Val Loss: 0.5422, Val Ac
 85%|████████▍ | 28/33 [01:29<00:16,  3.20s/it]Epoch 28/33 - Train Loss: 0.0081, Accuracy: 0.9977 - Val Loss: 0.5974, Val Ac
 88%|████████▊ | 29/33 [01:32<00:13,  3.34s/it]Epoch 29/33 - Train Loss: 0.0027, Accuracy: 0.9998 - Val Loss: 0.6259, Val Ac
 91%|█████████ | 30/33 [01:36<00:09,  3.29s/it]Epoch 30/33 - Train Loss: 0.0008, Accuracy: 1.0000 - Val Loss: 0.6379, Val Ac
 94%|█████████▍| 31/33 [01:40<00:06,  3.47s/it]Epoch 31/33 - Train Loss: 0.0006, Accuracy: 1.0000 - Val Loss: 0.6422, Val Ac
 97%|█████████▋| 32/33 [01:43<00:03,  3.39s/it]Epoch 32/33 - Train Loss: 0.0005, Accuracy: 1.0000 - Val Loss: 0.6444, Val Ac
100%|██████████| 33/33 [01:46<00:00,  3.22s/it]Epoch 33/33 - Train Loss: 0.0005, Accuracy: 1.0000 - Val Loss: 0.6458, Val Ac
```

Visualize & Evaluate Model

```
# Seaborn for prettier plot

import seaborn as sns

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


# Visualize training loss

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

plt.subplot(2, 1, 1)
plt.plot(train_loss_list, linewidth = 3)
plt.ylabel("training loss")
plt.xlabel("iterations")
sns.despine()

plt.subplot(2, 1, 2)
plt.plot(validation_accuracy_list, linewidth = 3, color = 'gold')
plt.ylabel("validation accuracy")
sns.despine()
```





```
# Compute the testing accuracy

# Set model to evaluation mode
model.eval()

# Track test accuracy
test_correct = 0
test_total = 0
test_loss = 0.0

with torch.no_grad():
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        outputs = model(inputs)
        loss = loss_func(outputs, targets)

        test_loss += loss.item() * inputs.size(0)  # sum loss over batch
        predicted = torch.argmax(outputs, 1)
        test_correct += (predicted == targets).sum().item()
        test_total += targets.size(0)

# Final metrics
avg_test_loss = test_loss / test_total
test_accuracy = test_correct / test_total

print(f" Test Loss: {avg_test_loss:.4f}")
print(f" Test Accuracy: {test_accuracy:.4f}")
```

```
 Test Loss: 0.7045
 Test Accuracy: 0.8960
```

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

# "Accuracy of T-shirt/top: 93.5 %"
# "Accuracy of Trouser: 89.3 %"
# etc...

# What's the fashion item that your model had the hardest time classifying?

import numpy as np

# Switch to evaluation mode and move model to correct device
model.eval()
all_preds = []
all_targets = []

with torch.no_grad():
    for inputs, targets in test_loader:
        inputs = inputs.to(device)
        outputs = model(inputs)
        preds = torch.argmax(outputs, dim=1)

        all_preds.append(preds.cpu())
        all_targets.append(targets)

# Concatenate predictions and targets into single arrays
all_preds = torch.cat(all_preds).numpy()
all_targets = torch.cat(all_targets).numpy()
clothing = {
    0 : 'T-shirt/top',
    1 : 'Trouser',
    2 : 'Pullover',
    3 : 'Dress',
    4 : 'Coat',
    5 : 'Sandal',
    6 : 'Shirt',
    7 : 'Sneaker',
    8 : 'Bag',
    9 : 'Ankle Boot'
}
least_acc = [1, -1]

# Compute per-class accuracy using np.where
for cls in range(10):
    idx = np.where(all_targets == cls)[0]
    total = len(idx)
    correct = np.sum(all_preds[idx] == all_targets[idx])
    acc = correct / total if total > 0 else 0
    least_acc = least_acc if acc > least_acc[0] else [acc, cls]
    print(f"Class {clothing[cls]}:\t\t Accuracy = {acc:.2%}")

print(f'\nModel had hardest time classifying {clothing[least_acc[1]]}s')
```

```
Class T-shirt/top:               Accuracy = 89.72%
Class Trouser:           Accuracy = 98.10%
Class Pullover:          Accuracy = 85.59%
Class Dress:             Accuracy = 89.25%
Class Coat:              Accuracy = 77.39%
Class Sandal:            Accuracy = 97.70%
Class Shirt:             Accuracy = 73.20%
Class Sneaker:           Accuracy = 97.89%
Class Bag:               Accuracy = 96.84%
Class Ankle Boot:                Accuracy = 93.68%

Model had hardest time classifying Shirts
```