Hands On #5

Chapter 10 – Introduction to Artificial Neural Networks with Pytorch

File name convention: For group 42 and members Richard Stallman and Linus Torvalds it would be:
"05_neural_nets_with_pytorch_Stallman_Torvalds.pdf"

Submission via blackboard (UA).

Feel free to answer free text questions in text cells using markdown and possibly $\LaTeX$ if you want to.

You don't have to understand every line of code here and it is not intended for you to try to understand every line of code.
Big blocks of code are usually meant to just be clicked through.

Setup

In [17]:
```
import sys
assert sys.version_info >= (3, 5)

import sklearn
assert sklearn.__version__ >= "0.20"

import torch
assert torch.__version__ >= "2.0"

import numpy as np
import os

np.random.seed(42)

%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
```

```
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

Perceptrons

Perceptrons are a form of linear classifier. The characteristic expression is $\Sigma_i w_i \cdot x_i + b$. Where x are your inputs and w are your model weights and b is a learnable bias term. The classification part comes in by setting **any positive result of the above expression as the 1 or True label** and any **negative result as the 0 or False label**. We then use stochastic gradient descent to optimize the weights and bias.

In [18]:
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(int)
```

Task 1:

Fit the iris dataset into a Perceptron layer and predict the class of a sample with petal length of 2 and a petal width of 0.5.

Use: `max_iter=1000`, `tol =1e-3` and `random_state= 42`.

Note: we set `max_iter` and `tol` explicitly to avoid warnings about the fact that their default value will change in future versions of Scikit-Learn.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [19]:
```
per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)
```

Out[19]:

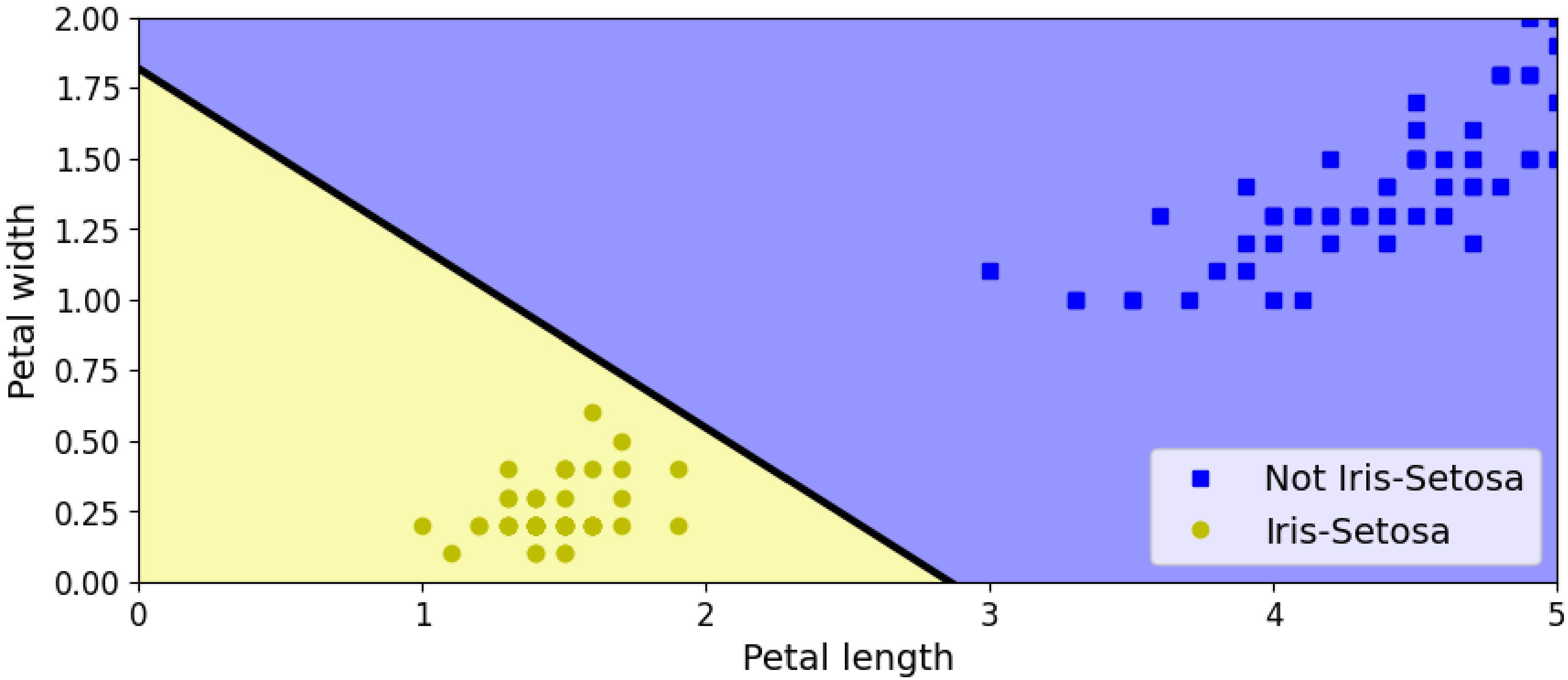
Perceptron

Perceptron(random_state=42)

In [20]:
```
y_pred = per_clf.predict([[2, 0.5]])
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

In [21]:
```
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linewidth=3)
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)
plt.show()
```



Task 2

Elaborate on the difference between a perceptron and logistic regression.

Hint: Consider the nature of the boundary in the above plot.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Task 2 answer:

Logistic regression function is y = 1/(1 + e ^ (ax + b)) Its y value is from (0,1), and it works well on non-linear splitable data set. Perceptron is a linear calssifer. it's function is y = ax + b. It works well on linear splitable data set. It may not generalize well if the dataset is slightly non-linearly separable.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above

Activation functions

Task 3

Describe the role of activation functions within a neural network. If you build a neural network with no activation function, which model that we've seen in this class would your network resemble?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Task 3 answer:

Activation functions play a crucial role in neural networks by introducing non-linearity into the model. If I build a neural network with no activation function,ti will be a multi- layer perceptron.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above

Building an Image Classifier

First let's import Pytorch. **Pytorch is a machine learning platform developed by Meta**. It is now a free, open-source platform.

In [22]:
```
torch.__version__
```

Out[22]:
```
'2.5.1+cu124'
```

Let's start by loading the fashion MNIST dataset. Keras has a number of functions to load popular datasets in `keras.datasets`. **The dataset is already split for you between a training set and a test set**, but it can be useful to split the training set further to have a validation set.

Keras is another machine learning toolkit that acts as the python interface for tensorflow. We generally won't use it in this course other than for accessing some data collections.

In [23]:
```
import keras
```

In [24]:
```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

The training set contains 60,000 grayscale images, each 28x28 pixels:

In [25]:
```
X_train_full.shape
```

Out[25]:
```
(60000, 28, 28)
```

Each pixel intensity is represented as a byte (0 to 255):

In [26]:
```
X_train_full.dtype
```

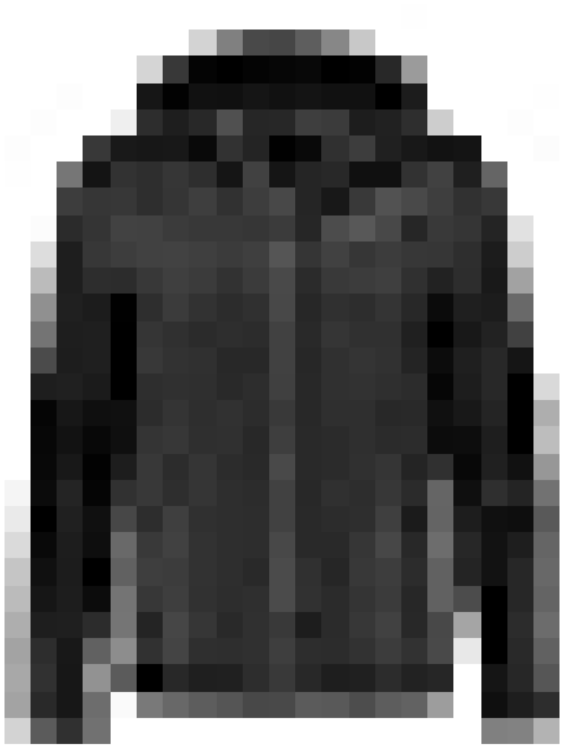
Out[26]:
```
dtype('uint8')
```

Let's **split the full training set into a validation set and a (smaller) training set**. We also **scale the pixel intensities down to the 0-1 range** and convert them to floats, by dividing by 255. This is essentially min-max scaling or normalization for pixels with a maximum value of 255 and a minimum of 0.

In [27]:
```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

You can plot an image using Matplotlib's `imshow()` function, with a `'binary'` color map:

In [28]:
```
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```



The labels are the class IDs (represented as uint8), from 0 to 9:

In [29]:
```
y_train
```

Out[29]:
```
array([4, 0, 7, ..., 3, 0, 5], dtype=uint8)
```

Here are the corresponding class names:

In [30]:
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

So the first image in the training set is a coat:

In [31]:
```
class_names[y_train[0]]
```

Out[31]:
```
'Coat'
```

The validation set contains 5,000 images, and the test set contains 10,000 images:

In [32]:
```
X_valid.shape
```

Out[32]: (5000, 28, 28)

In [33]:
```
X_test.shape
```

Out[33]: (10000, 28, 28)

Let's take a look at a sample of the images in the dataset:

In [34]:
```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

Data Loaders: Pytorch is built on a data type called a tensor. Numpy arrays are not themselves tensors. So we'll use a *dataloader* to **convert our numpy arrays to tensors** and pass those to our pytorch model.

In [35]:
```
from torch.utils.data import Dataset, DataLoader
```

In [36]:
```
class MnistDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X.copy()).float()
        self.y = torch.from_numpy(y.copy()).long()
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

In [37]:
```
train_data = MnistDataset(X_train, y_train)
valid_data = MnistDataset(X_valid, y_valid)
test_data = MnistDataset(X_test, y_test)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
valid_loader = DataLoader(valid_data, batch_size=64, shuffle=False)
```

Here's an introductory tutorial to using Datasets and Dataloaders in Pytorch that you may find helpful.

Defining a neural network using Pytorch:

In the cells below we build an deep neural network model using the Pytorch Sequential class. The input is an image of shape 28 by 28 whose dimensions are reshaped to a flat 1d array of length $28^2 = 784$ by the `nn.Flatten` layer. The network consists of `Linear` layers (fully-connected multi-output perceptrons) sandwiched between non-linear activations `ReLU` and `Softmax`. The `Softmax` output activation function is used for multi-label classification. The layers are wrapped in a `nn.Sequential` function that executes each operation in the given order when the `model` object is called.

In [38]:
```
np.random.seed(42)
torch.manual_seed(42)
```

Out[38]: <torch._C.Generator at 0x7b8b1c4eaeb0>

In [39]:
```
from torch import nn

model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28*28, 300),
    nn.ReLU(),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.Linear(100, 10),
    nn.Softmax(dim=1)
)
```

This is essentially a multi-layer perceptron model with activation functions to each "neuron". Additionally, **we no longer end the model with a decision function** that outputs either a 0 or 1. Here our final layer's **weights will be passed through a softmax function which will output a collection of 10 values which add up to 1**. These are the probabilities of each class being the correct class.

In [40]:
```
print(model)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=300, bias=True)
  (2): ReLU()
  (3): Linear(in_features=300, out_features=100, bias=True)
  (4): ReLU()
  (5): Linear(in_features=100, out_features=10, bias=True)
  (6): Softmax(dim=1)
)
```

We can look at the layers directly:

In [41]:
```
hidden1 = model[1] # 1st hidden layer is at index 1
```

In [42]:
```
weights, biases = hidden1.weight, hidden1.bias
```

In [43]:
```
weights
```

Out[43]: Parameter containing:

```
tensor([[ 0.0273,  0.0296, -0.0084,  ..., -0.0142,  0.0093,  0.0135],
        [-0.0188, -0.0354,  0.0187,  ..., -0.0106, -0.0001,  0.0115],
        [-0.0008,  0.0017,  0.0045,  ..., -0.0127, -0.0188,  0.0059],
        ...,
        [-0.0283,  0.0160, -0.0331,  ..., -0.0214, -0.0285,  0.0025],
        [ 0.0149, -0.0222,  0.0158,  ...,  0.0002,  0.0009,  0.0010],
        [ 0.0060, -0.0062, -0.0113,  ...,  0.0222,  0.0135, -0.0049]],
       requires_grad=True)
```

In [44]:

```
weights.shape
```

Out[44]:

```
torch.Size([300, 784])
```

In [45]:

```
biases
```

```
Out[45]: Parameter containing:
tensor([ 0.0206,  0.0060,  0.0232, -0.0270, -0.0098, -0.0182, -0.0304, -0.0039,
          0.0041,  0.0309, -0.0117,  0.0326,  0.0127, -0.0348, -0.0126, -0.0341,
         -0.0169, -0.0226, -0.0347,  0.0201,  0.0293, -0.0016, -0.0167,  0.0077,
          0.0332, -0.0142, -0.0279, -0.0221,  0.0331, -0.0055, -0.0130,  0.0019,
         -0.0034, -0.0280,  0.0010,  0.0137,  0.0272,  0.0116, -0.0218,  0.0247,
          0.0040, -0.0131, -0.0045, -0.0003, -0.0321, -0.0075,  0.0031, -0.0330,
          0.0140, -0.0321, -0.0045, -0.0035,  0.0332,  0.0188, -0.0277, -0.0120,
         -0.0268, -0.0152, -0.0327, -0.0103,  0.0282,  0.0170,  0.0344, -0.0027,
         -0.0096, -0.0202, -0.0189, -0.0084, -0.0194, -0.0327, -0.0287, -0.0225,
         -0.0017, -0.0080, -0.0102, -0.0216,  0.0171,  0.0219, -0.0348,  0.0229,
         -0.0177,  0.0216, -0.0281,  0.0350,  0.0019, -0.0181, -0.0004, -0.0327,
          0.0191,  0.0283, -0.0205, -0.0212,  0.0040,  0.0131, -0.0218,  0.0054,
          0.0006,  0.0229,  0.0301, -0.0169,  0.0102,  0.0157, -0.0213,  0.0213,
         -0.0085,  0.0229,  0.0178,  0.0188,  0.0182, -0.0264, -0.0269,  0.0040,
         -0.0072,  0.0134, -0.0148, -0.0343,  0.0021, -0.0109,  0.0104,  0.0252,
          0.0197, -0.0216, -0.0314, -0.0205, -0.0249, -0.0145, -0.0177, -0.0287,
         -0.0255,  0.0345, -0.0086, -0.0318,  0.0079, -0.0163,  0.0244, -0.0346,
          0.0136, -0.0108, -0.0209,  0.0069,  0.0082, -0.0290, -0.0010,  0.0033,
          0.0268, -0.0198,  0.0222,  0.0044, -0.0089, -0.0072,  0.0161,  0.0176,
          0.0052,  0.0185,  0.0143,  0.0303, -0.0152, -0.0214, -0.0170,  0.0035,
          0.0126,  0.0108,  0.0145,  0.0027,  0.0223, -0.0070,  0.0163,  0.0120,
         -0.0244, -0.0142,  0.0148, -0.0338,  0.0328,  0.0084, -0.0183, -0.0049,
          0.0203,  0.0166,  0.0236, -0.0264, -0.0247,  0.0014,  0.0115, -0.0032,
         -0.0337, -0.0156, -0.0340, -0.0174, -0.0018,  0.0325, -0.0249, -0.0346,
         -0.0167,  0.0035,  0.0243,  0.0037, -0.0216,  0.0006, -0.0157,  0.0345,
          0.0039, -0.0312,  0.0323, -0.0182, -0.0256, -0.0253,  0.0293, -0.0324,
          0.0216, -0.0241, -0.0197,  0.0255, -0.0262, -0.0302, -0.0184,  0.0301,
         -0.0024,  0.0309,  0.0228, -0.0151, -0.0154, -0.0127,  0.0166,  0.0179,
          0.0032, -0.0045, -0.0057, -0.0158, -0.0121, -0.0134, -0.0023,  0.0037,
          0.0185,  0.0124,  0.0041, -0.0323, -0.0156, -0.0015,  0.0220,  0.0246,
          0.0106, -0.0315, -0.0049, -0.0019, -0.0284,  0.0174, -0.0328,  0.0021,
         -0.0252,  0.0307, -0.0017,  0.0279,  0.0300,  0.0309, -0.0162,  0.0122,
          0.0180, -0.0009, -0.0233, -0.0241,  0.0114, -0.0210,  0.0139, -0.0034,
          0.0135,  0.0196,  0.0108,  0.0251, -0.0069, -0.0152, -0.0047,  0.0335,
          0.0318, -0.0221, -0.0136, -0.0106,  0.0055, -0.0294,  0.0353,  0.0098,
          0.0196, -0.0197,  0.0241, -0.0116, -0.0244, -0.0294,  0.0096, -0.0086,
         -0.0121, -0.0340,  0.0203,  0.0332, -0.0226, -0.0016,  0.0235, -0.0143,
         -0.0162, -0.0013, -0.0140, -0.0216], requires_grad=True)
```

```
In [46]: biases.shape
```

```
Out[46]: torch.Size([300])
```

Training loop in Pytroch

In pytorch we need to **create loops** that do things like **training** our model. Below is an example training loop that we'll use in this code. Each step is commented and it's **very important to understand this code** in order to use pytorch and other similar machine learning libraries.

In [47]:
```
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None):
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
        'val_loss': [],
        'val_metric': []
    }  # Initialize a dictionary to store epoch-wise results

    for epoch in range(num_epochs):
        model.train()  # Set the model to training mode
        epoch_loss = 0.0  # Initialize the epoch loss and metric values
        epoch_metric = 0.0

        # Training loop
        for X, y in train_loader: # Iterate through train data loader
            optimizer.zero_grad()  # Clear existing gradients
            outputs = model(X)  # Make predictions

            #REPLACE THE FOLLOWING LINE FOR EXERCISE 8
            #loss = criterion(outputs.squeeze(-1), y)  # Compute the loss
            loss = criterion(outputs, y)  # Compute the loss

            loss.backward()  # Compute gradients
            optimizer.step()  # Update model parameters

            epoch_loss += loss.item() # Add up loss across each batch in the epoch
            if metric is not None: # Check whether you've passed a loss metric
                epoch_metric += metric(outputs, y) # Compute metric value
            else:
                epoch_metric += 0 # If no metric, append 0

        # Average training loss and metric
        epoch_loss /= len(train_loader)
        epoch_metric /= len(train_loader)

        # Validation loop
        model.eval()  # Set the model to evaluation mode
        with torch.no_grad():  # Disable gradient calculation
```

```
            val_loss = 0.0
            val_metric = 0.0
            for X_val, y_val in val_loader:
                outputs_val = model(X_val)  # Make predictions

                #REPLACE THE FOLLOWING LINE FOR EXERCISE 8
                #val_loss += criterion(outputs_val.squeeze(-1), y_val).item()  # Compute the loss
                val_loss += criterion(outputs_val, y_val).item()  # Compute loss

                if metric is not None:
                    val_metric += metric(outputs_val, y_val)
                else:
                    val_metric += 0

            val_loss /= len(val_loader)
            val_metric /= len(val_loader)

        # Append epoch results to history
        history['epoch'].append(epoch)
        history['train_loss'].append(epoch_loss)
        history['train_metric'].append(epoch_metric)
        history['val_loss'].append(val_loss)
        history['val_metric'].append(val_metric)

        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.4f}, '
              f'Train Metric: {epoch_metric:.4f}, Val Loss: {val_loss:.4f}, '
              f'Val Metric: {val_metric:.4f}')

    return history, model
```

Model parameters: Here, we choose the **"sgd" optimizer**. The SGD optimizer is a derivative of the SGD algorithm but, instead of updating coefficients for a linear regression as we saw in previous exercises, we compute the **gradient of our loss function** (sparse categorical crossentropy) **with respect to our model weights and layer biases** and use that to update our weights and biases. Note that our gradient will now be the sum of more complicated partial derivatives. Note that **our gradient will now be the sum of more complicated partial derivatives**. $\partial L/\partial w_i = (\partial L/\partial f(w_i))(\partial f(w_i)/\partial w_i)$ where $f(w_i) = ReLU(w_i x_i + b)$. Additionally we may have a partial derivative for our bias term $\partial L/\partial b$. The total gradient for stochastic gradient descent will then be $\nabla L_{layer} = \Sigma_i \partial L/\partial w_i + \partial L/\partial b$.

In [48]:
```
criterion = torch.nn.CrossEntropyLoss() # This is a loss function/criterion
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # This updates model weights

def accuracy_metric(pred, target): # This is an accuracy metric
    if pred.shape[1] > 1: # "Accuracy" calcuation is different depending on # of classes
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    else:
        accuracy = torch.sum(pred > 0.5 == target).item() / len(pred)
    return accuracy
```

Now let's train the model for 30 epochs.

We save the most crucial parameters (`['loss', 'accuracy', 'val_loss', 'val_accuracy']`) in a dictionary named "history".

In [49]:
```
# Here we train the model and then return the training history and trained model
history, model = train_and_validate(train_loader, valid_loader, model,
                                    optimizer=optimizer, criterion=criterion,
                                    num_epochs=30, metric=accuracy_metric)
```

```
Epoch [1/30], Train Loss: 2.2993, Train Metric: 0.2244, Val Loss: 2.2947, Val Metric: 0.3475
Epoch [2/30], Train Loss: 2.2869, Train Metric: 0.3563, Val Loss: 2.2752, Val Metric: 0.3530
Epoch [3/30], Train Loss: 2.2372, Train Metric: 0.3208, Val Loss: 2.1675, Val Metric: 0.3412
Epoch [4/30], Train Loss: 2.0716, Train Metric: 0.4775, Val Loss: 1.9794, Val Metric: 0.5487
Epoch [5/30], Train Loss: 1.9373, Train Metric: 0.5767, Val Loss: 1.8899, Val Metric: 0.6286
Epoch [6/30], Train Loss: 1.8712, Train Metric: 0.6273, Val Loss: 1.8493, Val Metric: 0.6325
Epoch [7/30], Train Loss: 1.8431, Train Metric: 0.6333, Val Loss: 1.8294, Val Metric: 0.6422
Epoch [8/30], Train Loss: 1.8257, Train Metric: 0.6539, Val Loss: 1.8139, Val Metric: 0.6746
Epoch [9/30], Train Loss: 1.8090, Train Metric: 0.6837, Val Loss: 1.7965, Val Metric: 0.6998
Epoch [10/30], Train Loss: 1.7925, Train Metric: 0.7006, Val Loss: 1.7818, Val Metric: 0.7091
Epoch [11/30], Train Loss: 1.7795, Train Metric: 0.7091, Val Loss: 1.7698, Val Metric: 0.7150
Epoch [12/30], Train Loss: 1.7697, Train Metric: 0.7142, Val Loss: 1.7625, Val Metric: 0.7174
Epoch [13/30], Train Loss: 1.7623, Train Metric: 0.7173, Val Loss: 1.7556, Val Metric: 0.7203
Epoch [14/30], Train Loss: 1.7569, Train Metric: 0.7199, Val Loss: 1.7527, Val Metric: 0.7213
Epoch [15/30], Train Loss: 1.7523, Train Metric: 0.7226, Val Loss: 1.7469, Val Metric: 0.7267
Epoch [16/30], Train Loss: 1.7488, Train Metric: 0.7247, Val Loss: 1.7433, Val Metric: 0.7275
Epoch [17/30], Train Loss: 1.7460, Train Metric: 0.7262, Val Loss: 1.7412, Val Metric: 0.7288
Epoch [18/30], Train Loss: 1.7435, Train Metric: 0.7281, Val Loss: 1.7388, Val Metric: 0.7308
Epoch [19/30], Train Loss: 1.7414, Train Metric: 0.7288, Val Loss: 1.7368, Val Metric: 0.7322
Epoch [20/30], Train Loss: 1.7395, Train Metric: 0.7303, Val Loss: 1.7354, Val Metric: 0.7332
Epoch [21/30], Train Loss: 1.7378, Train Metric: 0.7317, Val Loss: 1.7337, Val Metric: 0.7344
Epoch [22/30], Train Loss: 1.7365, Train Metric: 0.7322, Val Loss: 1.7325, Val Metric: 0.7344
Epoch [23/30], Train Loss: 1.7350, Train Metric: 0.7333, Val Loss: 1.7309, Val Metric: 0.7367
Epoch [24/30], Train Loss: 1.7335, Train Metric: 0.7348, Val Loss: 1.7299, Val Metric: 0.7362
Epoch [25/30], Train Loss: 1.7327, Train Metric: 0.7350, Val Loss: 1.7290, Val Metric: 0.7379
Epoch [26/30], Train Loss: 1.7316, Train Metric: 0.7358, Val Loss: 1.7277, Val Metric: 0.7375
Epoch [27/30], Train Loss: 1.7307, Train Metric: 0.7367, Val Loss: 1.7270, Val Metric: 0.7395
Epoch [28/30], Train Loss: 1.7299, Train Metric: 0.7368, Val Loss: 1.7260, Val Metric: 0.7399
Epoch [29/30], Train Loss: 1.7290, Train Metric: 0.7379, Val Loss: 1.7255, Val Metric: 0.7395
Epoch [30/30], Train Loss: 1.7280, Train Metric: 0.7385, Val Loss: 1.7256, Val Metric: 0.7397
```

In [50]:

```python
print(history.keys())
```

```
dict_keys(['epoch', 'train_loss', 'train_metric', 'val_loss', 'val_metric'])
```

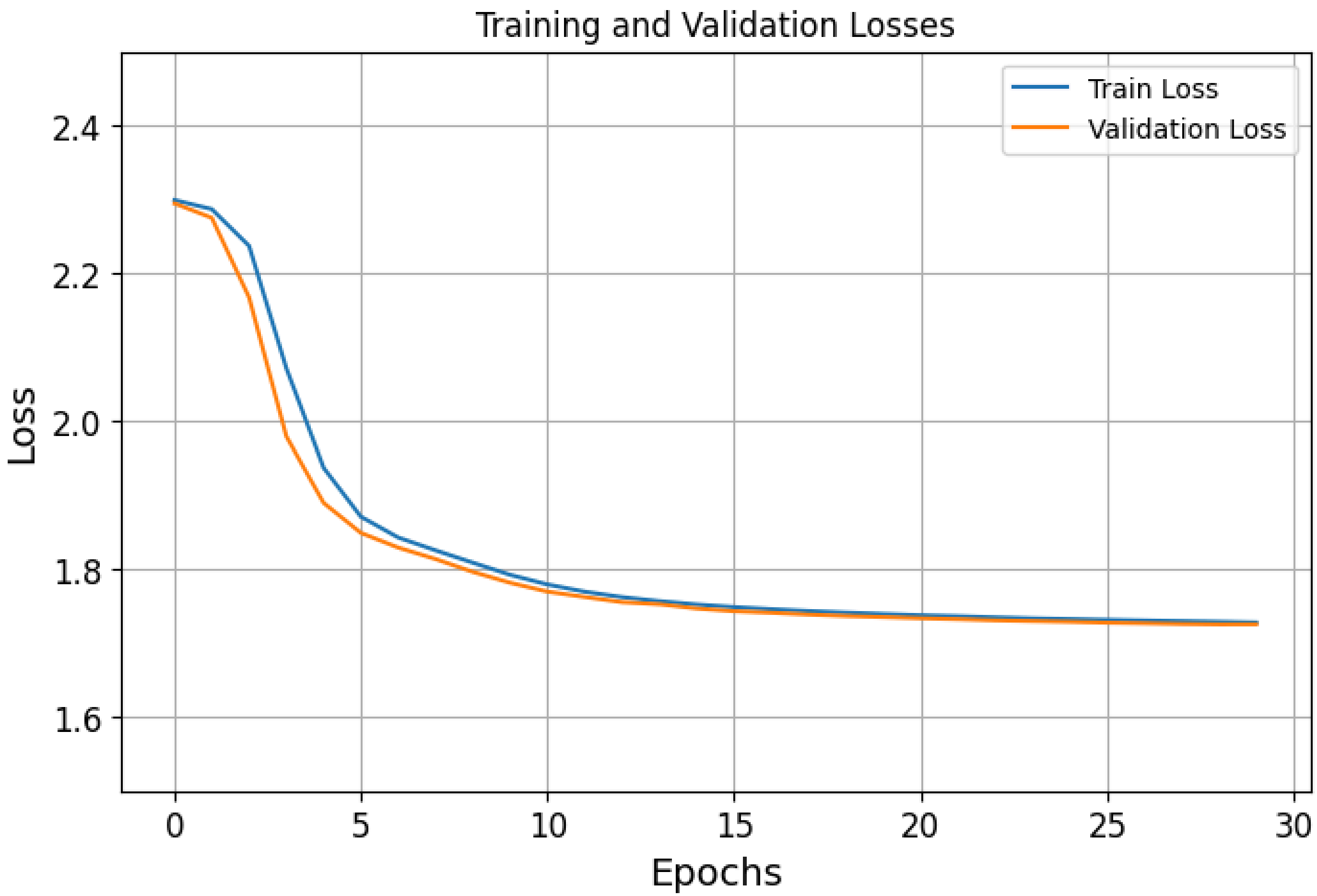
In [51]:

```python
print(history['epoch'])
print(history['train_loss'])
print(history['train_metric'])
print(history['val_loss'])
print(history['val_metric'])
```

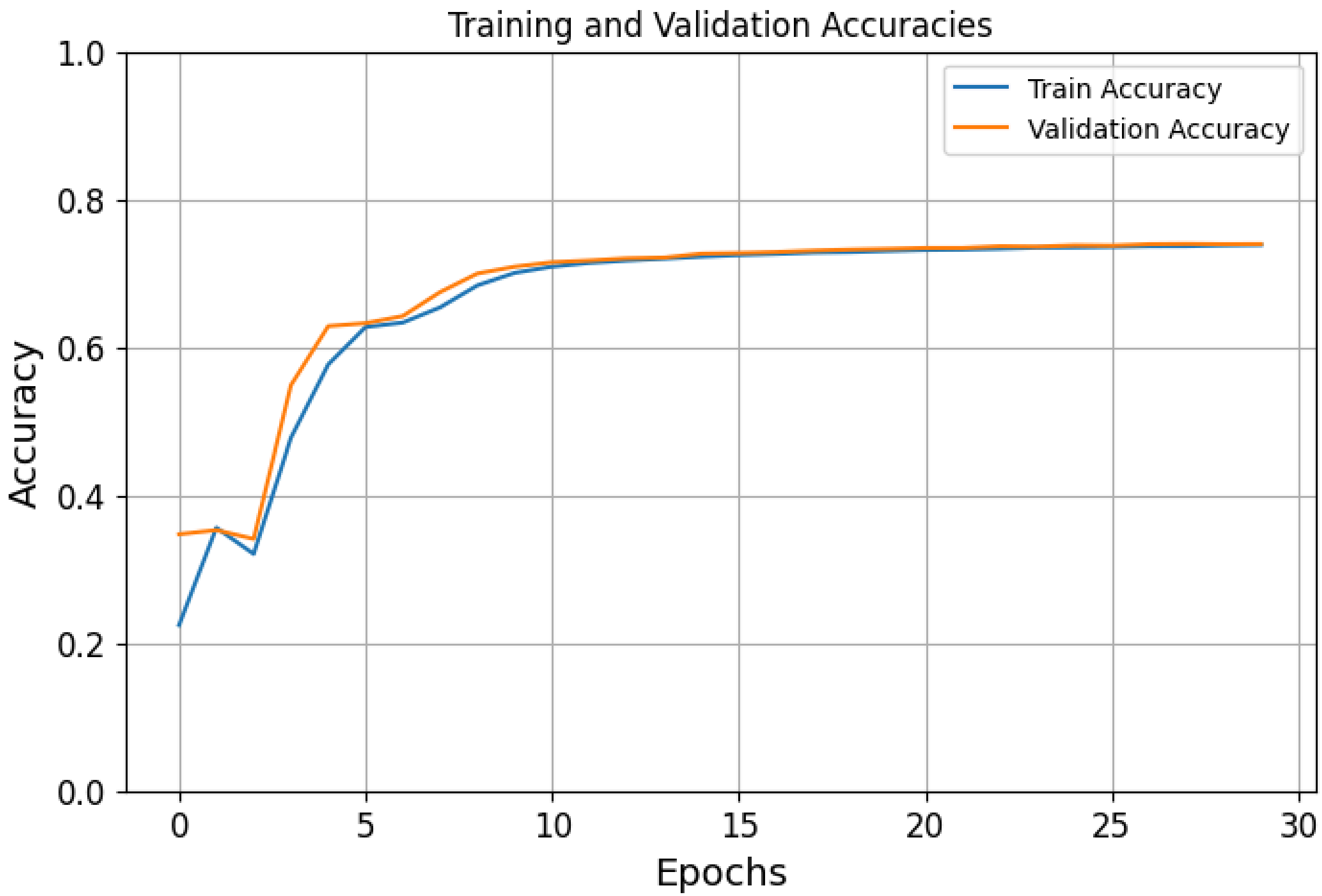
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[2.299255026772965, 2.2869284840517268, 2.2372200128644013, 2.0715981857721197, 1.9372768113779468, 1.8711816965147505, 1.843055
0411690114, 1.8256678121034489, 1.80900295640147, 1.7925438666066458, 1.779526698589325, 1.769691792199778, 1.7623358999573908,
1.7568707409293152, 1.7523392612157866, 1.748844196768694, 1.745989260424015, 1.7434659591940946, 1.7413751172464946, 1.73947687
98074056, 1.7377510845661164, 1.736477952918341, 1.7349662062733673, 1.7335456074670303, 1.7326628113901892, 1.7315952174885327,
1.7306757070297418, 1.7298972662105117, 1.7290010549301325, 1.7280245229255322]
[0.22443071705426357, 0.3563408430232558, 0.3207848837209302, 0.47748304263565894, 0.5767139050387596, 0.6273013565891472, 0.633
3333333333333, 0.6539244186046511, 0.6837148740310077, 0.7006177325581395, 0.7091388081395349, 0.7141593992248062, 0.71729651162
7907, 0.719858284883721, 0.7225654069767442, 0.7247274709302326, 0.7262294089147286, 0.7281371124031008, 0.7288396317829458, 0.7
302931201550388, 0.7317163275193798, 0.7321765988372093, 0.7332788275193798, 0.7347989341085271, 0.7350351259689923, 0.735755813
9534883, 0.73671875, 0.7368459302325582, 0.7379118217054264, 0.7384871608527133]
[2.2947069602676584, 2.2751974606815772, 2.167499590523635, 1.9794300462626204, 1.8898716636850863, 1.8492559119115901, 1.829385
380201702, 1.8139158457140379, 1.7965033688122714, 1.7817917669875711, 1.769795757305773, 1.76249751260009, 1.7555593206912656,
1.7527418242225163, 1.746918178811858, 1.74328711968434, 1.741240303727645, 1.7387831512885759, 1.736819474002983, 1.73539429978
4793, 1.7337251841267454, 1.732498976248729, 1.7308916532540624, 1.7299228453937965, 1.7289734203604203, 1.7277342337596266, 1.7
26968994623498, 1.726049174236346, 1.725511375861832, 1.7255770375456991]
[0.34750791139240506, 0.3530458860759494, 0.34117879746835444, 0.5486550632911392, 0.6285601265822784, 0.6325158227848101, 0.642
2072784810127, 0.6746439873417721, 0.6997626582278481, 0.7090585443037974, 0.7149920886075949, 0.7173655063291139, 0.72033227848
10127, 0.7213212025316456, 0.7266613924050633, 0.7274525316455697, 0.7288370253164557, 0.7308148734177216, 0.7321993670886076,
0.7331882911392406, 0.734375, 0.734375, 0.736748417721519, 0.7361550632911392, 0.7379351265822784, 0.7375395569620253, 0.7395174
050632911, 0.7399129746835443, 0.7395174050632911, 0.7397151898734177]
```

Let's visualize the learning curves for this model training.

In [52]:
```python
import pandas as pd

losses = pd.DataFrame({
    'Train Loss': history['train_loss'],
    'Validation Loss': history['val_loss']
})

# Plotting
losses.plot(figsize=(8, 5))
plt.grid(True)
plt.title("Training and Validation Losses")
plt.gca().set_ylim(1.5, 2.5)  # Adjust the y-axis limits if needed
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



In [53]:
```
accuracies = pd.DataFrame({
    'Train Accuracy': history['train_metric'],
    'Validation Accuracy': history['val_metric']
})

# Plotting
accuracies.plot(figsize=(8, 5))
plt.grid(True)
plt.title("Training and Validation Accuracies")
plt.gca().set_ylim(0, 1)  # Adjust the y-axis limits if needed
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.show()
```



Testing loop

We also need to create a testing loop if we want to check the performance of our model after we train it. Notice a few key things:

1. We put the model into evaluation mode instead of training mode
2. We run the model without computing any gradients
3. There is no more optimizer needed

We also did this with the validation portion of `train_and_validate` loop.

In [54]:
```
def test_model(model, data_loader, criterion, metric):
    model.eval()  # Set the model to evaluation mode
```

```
    total_loss = 0.0  # Initialize the total loss and metric values
    total_metric = 0.0

    with torch.no_grad():  # Disable gradient tracking
        for batch in data_loader:
            X, y = batch

            # Pass the data to the model and make predictions
            outputs = model(X)

            # Compute the loss
            loss = criterion(outputs, y)

            # Add the loss and metric for the batch to the total values
            total_loss += loss.item()
            total_metric += metric(outputs, y)

    # Average loss and metric for the entire dataset
    avg_loss = total_loss / len(data_loader)
    avg_metric = total_metric / len(data_loader)

    print(f'Test Loss: {avg_loss:.4f}, Test Metric: {avg_metric:.4f}')

    return avg_loss, avg_metric
```

Task 4

Validate your model using `test_model` using the test dataloader `test_loader`.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [55]:
```
# evaluate model on  test set
test_loss, test_metric = test_model(model, test_loader, criterion, accuracy_metric)
```

```
Test Loss: 1.7316, Test Metric: 0.7344
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

Task 5

Select the first three samples from the test dataset not the dataloader and predict their corresponding classes using:

`prediction = model(torch.from_numpy(X_new).float())`. Then print the names/ categories of the elements in question (Eg. "Pants", "trouser")

You'll need to use `prediction.detach().numpy()` to get the outputs in a nice numpy format. **Pay attention to the above steps to avoid data type errors.**

Also, use `np.argmax(prediction, axis=-1)` on the one-hot-encoded predicitons to get the classes numbers.

In [56]:
```
X_new = X_test[:3]
```

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [57]:
```
prediction = model(torch.from_numpy(X_new).float())
prediction = prediction.detach().numpy()
print(prediction)
prediction = prediction.argmax(axis=-1)
prediction
np.array(class_names)[prediction]
```

```
[[4.6030599e-11 1.2689732e-11 1.2130785e-09 1.8984639e-07 3.5680960e-09
  3.4590818e-02 3.9018415e-09 2.2880553e-01 3.0471154e-05 7.3657304e-01]
 [2.5656430e-11 1.4879530e-19 9.9999976e-01 6.4507177e-10 1.8701478e-13
  4.5569996e-11 8.2898781e-16 4.9057829e-32 2.9427986e-07 3.6776139e-24]
 [3.5285372e-11 9.9999988e-01 1.0061653e-13 9.1475606e-08 9.0985450e-15
  3.1242169e-21 4.1934390e-15 1.7403466e-12 1.1485387e-19 3.9438671e-15]]
```

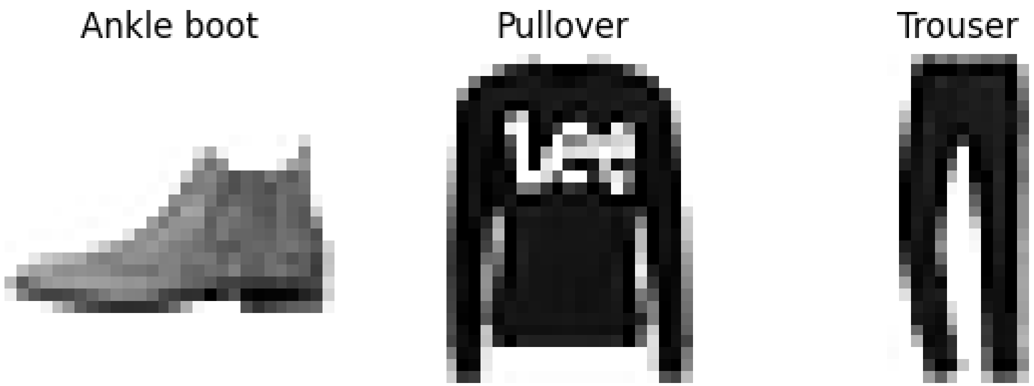
Out[57]:
```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

In [58]:
```
y_test[0:3]
```

Out[58]:
```
array([9, 2, 1], dtype=uint8)
```

In [59]:
```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```



Regression MLP

We can also build multi-layer perceptrons for regression. The difference here is that we remove features like softmax that are specific to multi-class classification and instead end with an appropriately sized linear layer and use mean-squared-error as our loss function instead of cross-entropy loss.

Let's load, split and scale the California housing dataset.

In [60]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full, y_train_full, random_state=42)
```

Task 6

Scale your training, validation, and test feature matrices using scikit-learn's StandardScaler. Best practice is to fit your scaler to either the full dataset or a large subset e.g. X_train and scale all samples using the same means and standard deviations. It is critical that you **use the same mean and standard deviation for all data**. This standard scaler is implementing standardization also sometimes referred to as standard normalization.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [61]:
```
scaler = StandardScaler()
scaler = scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

Task 7

Create a dataloader for the California housing dataset.

Be sure to use appropriate data types for the inputs (X) and outputs (y).

Read the documentation for this dataset. If you use the wrong data type in your __getitem__ function then the code will not work properly.

You must include the functions `__init__`, `__len__`, and `__getitem__`. Then, create your data sets and data loaders with `batch_size=64`.

Task 7 Question:

What are the appropriate data types for the inputs X and outputs y according to the California housing dataset documentation?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

Answer:

In [62]:
```
class CaliHousingDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X.copy()).float()
        self.y = torch.from_numpy(y.copy()).float()
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

In [63]:
```
train_data = CaliHousingDataset(X_train, y_train)
test_data = CaliHousingDataset(X_test, y_test)
valid_data = CaliHousingDataset(X_valid, y_valid)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data,batch_size=64, shuffle=False)
valid_loader = DataLoader(valid_data, batch_size=64, shuffle=False)
```

In [64]:
```
np.random.seed(42)
torch.manual_seed(42)
```

Out[64]:
```
<torch._C.Generator at 0x7b8b1c4eaeb0>
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

Task 8

This is a Neural Network with one hidden layer with 30 neurons. The output layer has one neuron, which is the regression value. **Create a train_and_validate loop for this model. Build the model, and train it using the SGD optimizer with a learning rate of 1e-2 for 30 epochs.**

This task is similar to task 5 except that we now do **regression, NOT classification**. There will be some changes like no longer passing an accuracy metric since that's not a regression metric. We'll also remove the softmax which is used for multi-class classification. Finally, we'll need to use an appropriate loss function.

Read through the California housing dataset documentation to determine the correct number of features and classes for the model.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [65]:
```
# # Note: No flatten layer or softmax layer and last dimension should be 1
model = nn.Sequential(
  #Linear layer that connects input features to 30 neurons
  nn.Linear(X_train.shape[1], 30),
  #ReLU activation function
  nn.ReLU(),
  #Linear layer that connects 30 hidden dims to classes
  nn.Linear(30, 1)
```

```
)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.002)
```

In [67]:
```
# # Note: No metric needed and remember to change the loss function
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None):
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
        'val_loss': [],
        'val_metric': []
    }  # Initialize a dictionary to store epoch-wise results

    for epoch in range(num_epochs):
        model.train()  # Set the model to training mode
        epoch_loss = 0.0  # Initialize the epoch loss and metric values
        epoch_metric = 0.0

        # Training loop
        for X, y in train_loader: # Iterate through train data loader
            optimizer.zero_grad()  # Clear existing gradients
            outputs = model(X)  # Make predictions

            #REPLACE THE FOLLOWING LINE FOR EXERCISE 8
            loss = criterion(outputs.squeeze(-1), y)  # Compute the loss
            #loss = criterion(outputs, y)  # Compute the loss

            loss.backward()  # Compute gradients
            optimizer.step()  # Update model parameters

            epoch_loss += loss.item() # Add up loss across each batch in the epoch

        # Average training loss and metric
        epoch_loss /= len(train_loader)

        # Validation loop
        model.eval()  # Set the model to evaluation mode
        with torch.no_grad():  # Disable gradient calculation
            val_loss = 0.0
            val_metric = 0.0
            for X_val, y_val in val_loader:
                outputs_val = model(X_val)  # Make predictions

                #REPLACE THE FOLLOWING LINE FOR EXERCISE 8
                val_loss += criterion(outputs_val.squeeze(-1), y_val).item()  # Compute the loss
```

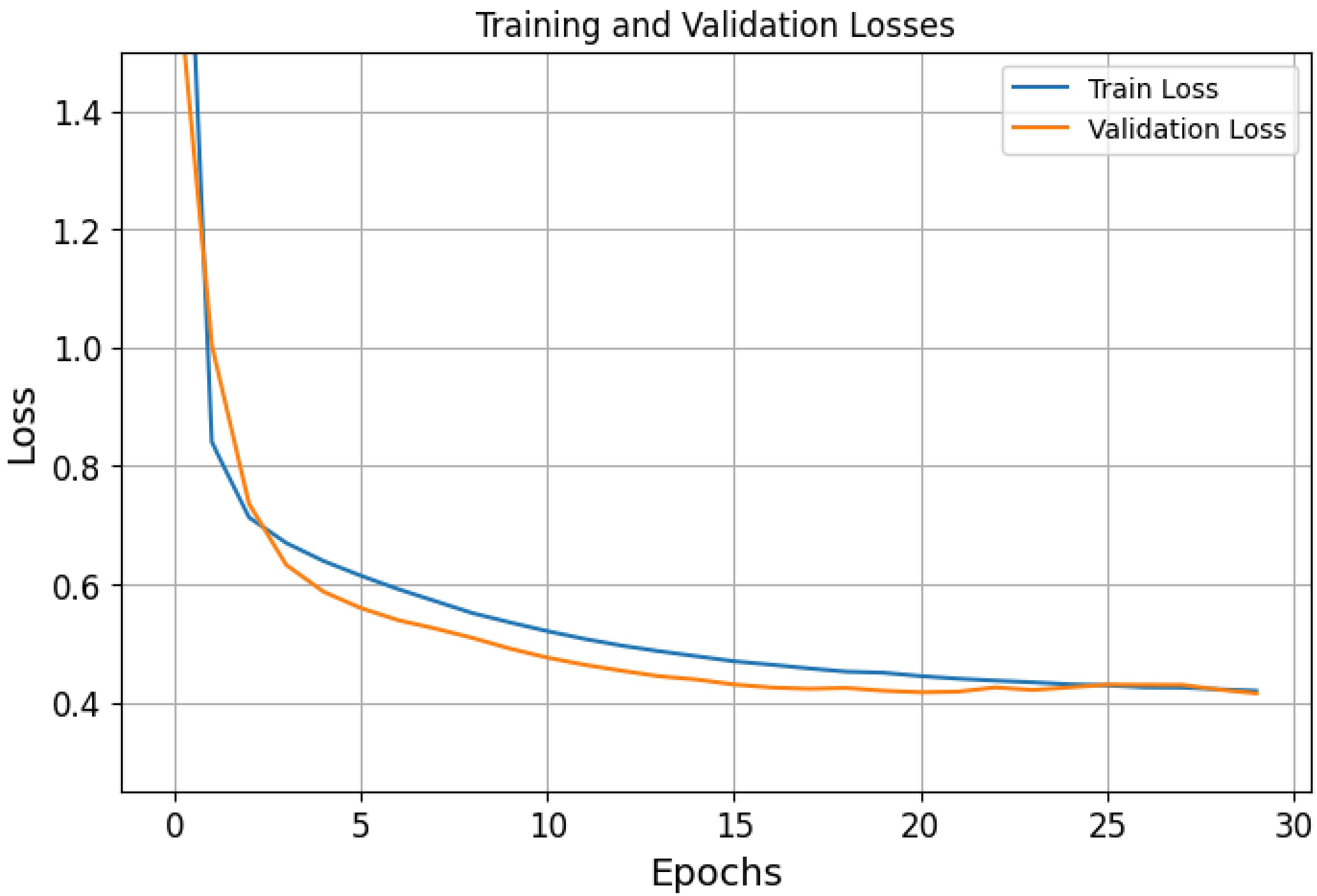
```
                #val_loss += criterion(outputs_val, y_val).item()  # Compute loss


            val_loss /= len(val_loader)

        # Append epoch results to history
        history['epoch'].append(epoch)
        history['train_loss'].append(epoch_loss)
        #history['train_metric'].append(epoch_metric)
        history['val_loss'].append(val_loss)
        #history['val_metric'].append(val_metric)

        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.4f}, '
              f'Val Loss: {val_loss:.4f}, ')

    return history, model


history, model = train_and_validate(train_loader, valid_loader, model,
                                    optimizer=optimizer, criterion=criterion,
                                    num_epochs=30, metric=accuracy_metric)
```

```
Epoch [1/30], Train Loss: 2.3169, Val Loss: 1.6865,
Epoch [2/30], Train Loss: 0.8400, Val Loss: 1.0039,
Epoch [3/30], Train Loss: 0.7127, Val Loss: 0.7356,
Epoch [4/30], Train Loss: 0.6695, Val Loss: 0.6322,
Epoch [5/30], Train Loss: 0.6389, Val Loss: 0.5870,
Epoch [6/30], Train Loss: 0.6141, Val Loss: 0.5593,
Epoch [7/30], Train Loss: 0.5914, Val Loss: 0.5389,
Epoch [8/30], Train Loss: 0.5710, Val Loss: 0.5246,
Epoch [9/30], Train Loss: 0.5507, Val Loss: 0.5087,
Epoch [10/30], Train Loss: 0.5351, Val Loss: 0.4908,
Epoch [11/30], Train Loss: 0.5204, Val Loss: 0.4756,
Epoch [12/30], Train Loss: 0.5071, Val Loss: 0.4633,
Epoch [13/30], Train Loss: 0.4958, Val Loss: 0.4533,
Epoch [14/30], Train Loss: 0.4863, Val Loss: 0.4440,
Epoch [15/30], Train Loss: 0.4778, Val Loss: 0.4383,
Epoch [16/30], Train Loss: 0.4693, Val Loss: 0.4300,
Epoch [17/30], Train Loss: 0.4635, Val Loss: 0.4249,
Epoch [18/30], Train Loss: 0.4573, Val Loss: 0.4228,
Epoch [19/30], Train Loss: 0.4518, Val Loss: 0.4242,
Epoch [20/30], Train Loss: 0.4500, Val Loss: 0.4195,
Epoch [21/30], Train Loss: 0.4443, Val Loss: 0.4169,
Epoch [22/30], Train Loss: 0.4399, Val Loss: 0.4179,
Epoch [23/30], Train Loss: 0.4366, Val Loss: 0.4251,
Epoch [24/30], Train Loss: 0.4338, Val Loss: 0.4208,
Epoch [25/30], Train Loss: 0.4303, Val Loss: 0.4252,
Epoch [26/30], Train Loss: 0.4287, Val Loss: 0.4299,
Epoch [27/30], Train Loss: 0.4253, Val Loss: 0.4296,
Epoch [28/30], Train Loss: 0.4249, Val Loss: 0.4295,
Epoch [29/30], Train Loss: 0.4219, Val Loss: 0.4213,
Epoch [30/30], Train Loss: 0.4194, Val Loss: 0.4151,
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

In [68]:

```
losses = pd.DataFrame({
    'Train Loss': history['train_loss'],
    'Validation Loss': history['val_loss']
})

# Plotting
losses.plot(figsize=(8, 5))
plt.grid(True)
plt.title("Training and Validation Losses")
plt.gca().set_ylim(0.25, 1.5)  # Adjust the y-axis limits if needed
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



Task 9.1

Create a `test_model` function and evalute your model's performance by using it on the test set. Also predict one element of the test set of your choice (`X_test[42]` for example) and compare to the real value. You can look at the validation part of the `test_model` function from before as a reference for how to create a test loop.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [78]:
```
def test_model(model, data_loader, criterion):
    model.eval()  # Set the model to evaluation mode

    total_loss = 0.0  # Initialize the total loss and metric values
```

```
    with torch.no_grad():  # Disable gradient tracking
        for batch in data_loader:
            X, y = batch

            # Pass the data to the model and make predictions
            outputs = model(X).squeeze()

            # Compute the loss
            loss = criterion(outputs, y.squeeze())

            # Add the loss and metric for the batch to the total values
            total_loss += loss.item()


    # Average loss and metric for the entire dataset
    avg_loss = total_loss / len(data_loader)
    print(f'Test Loss: {avg_loss:.4f}')

    return avg_loss
```

In [79]:
```
test_prediction = test_model(model, test_loader, criterion)
```

```
Test Loss: 0.4099
```

In [82]:
```
prediction = model(torch.from_numpy(X_test[42]).float())
prediction = prediction.detach().numpy()
print(prediction)
print(y_test[42])
```

```
[0.7801598]
0.713
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above

Saving the model weights for future use

Refer to the example on saving files in the course Resources folder on github

Task 9.2

Create a new blank model using the same architecture you used from part 8.
Load the model weights from your previously trained model from part 8.

Finally, compare the predictions of the two models on the same datapoint.

In [83]:
```
model(torch.from_numpy(X_test[42:43].copy()).float())
```

Out[83]:
```
tensor([[0.7802]], grad_fn=<AddmmBackward0>)
```

In [84]:
```
torch.save(model.state_dict(), "my_pytorch_model")
```

In [85]:
```
model_reloaded = nn.Sequential(
  #Linear layer that connects input features to 30 neurons
  nn.Linear(X_train.shape[1], 30),
  #ReLU activation function
  nn.ReLU(),
  #Linear layer that connects 30 hidden dims to classes
  nn.Linear(30, 1)
)
#Insert the same Sequential model architecture from part 8
```

In [86]:
```
model_reloaded.load_state_dict(torch.load("my_pytorch_model"))
```

```
<ipython-input-86-676a16534123>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default valu
e), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitr
ary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In
a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be execut
ed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlis
ted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case
where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental
feature.
  model_reloaded.load_state_dict(torch.load("my_pytorch_model"))
```

Out[86]:
```
<All keys matched successfully>
```

In [87]:
```
model_reloaded(torch.from_numpy(X_test[42:43].copy()).float())
```

Out[87]:
```
tensor([[0.7802]], grad_fn=<AddmmBackward0>)
```

Are these the same prediction?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Task 9.2 answer:same

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above

Model Layer Naming

For particularly large models or new architectures, it can be helpful to name model layers or blocks so that you can identify the source of bugs, etc.

In [88]:
```
from collections import OrderedDict
```

In [89]:
```
model_with_layer_naming = torch.nn.Sequential(OrderedDict([
          ('simple_linear_layer_1', nn.Linear(10,100)),
          ('a_neat_activation_function', nn.ReLU()),
        ]))
```

In [90]:
```
print(model_with_layer_naming)
```

```
Sequential(
  (simple_linear_layer_1): Linear(in_features=10, out_features=100, bias=True)
  (a_neat_activation_function): ReLU()
)
```

In [91]:
```
print(model_with_layer_naming.simple_linear_layer_1)
```

```
Linear(in_features=10, out_features=100, bias=True)
```

Task 10 (Bonus: 3 points)

For `X_train[0]` perform the forward pass yourself using matrix multiplications. Remember to include the biases.
Check with the prediction of the model that you get exactly the same!

Hints:

- use `np.dot(x,y)` for matrix multiplication
- for the first layer it would look like this:
  - matrix mult: `X_new` dot `l1`
  - add bias `b1`
  - apply `relu(...)`

In [92]:
```
print(model)
```

```
Sequential(
  (0): Linear(in_features=8, out_features=30, bias=True)
  (1): ReLU()
  (2): Linear(in_features=30, out_features=1, bias=True)
)
```

In [93]:
```python
l1 = np.array(model[0].weight.data.numpy()).T
b1 = np.array(model[0].bias.data.numpy())
l2 = np.array(model[2].weight.data.numpy()).T
b2 = np.array(model[2].bias.data.numpy())
```

In [94]:
```python
X_new = X_train[0]
```

In [95]:
```python
X_new.shape
```

Out[95]:
```
(8,)
```

In [96]:
```python
l1.shape
```

Out[96]:
```
(8, 30)
```

In [97]:
```python
b1.shape
```

Out[97]:
```
(30,)
```

In [98]:
```python
# This is the entirety of the ReLU function. How cool is that!
def relu(z):
    return np.maximum(0, z)
```

In [100…
```python
model(torch.from_numpy(X_new).float().unsqueeze(0))   # reproduce this!
```

Out[100…
```
tensor([[2.7561]], grad_fn=<AddmmBackward0>)
```

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [101…
```python
x = np.dot(X_new, l1) + b1
x = relu(x)
x = np.dot(x, l2) + b2
print(x)
```

```
[2.75605407]
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above