Training Deep Neural Networks

File name convention: For group 42 and memebers Richard Stallman and Linus Torvalds it would be
"06_Goup42_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 [3]:

```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

import torch
assert torch.__version__ >= "2.0"
from torch import nn
from torch.utils.data import DataLoader, Dataset

import keras

%load_ext tensorboard

# Common imports
import numpy as np
import os
```

```python
# to make this notebook's output stable across runs
torch.manual_seed(42)
np.random.seed(42)

# To plot pretty figures
%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)
```

Vanishing/Exploding Gradients Problem

Just like with SGD for linear regression, the fundamental procedure in simple neural networks is to **update model weights and biases** by taking some form of **partial derivative of a loss function** with respect to our weights and biases and then **stepping our weights** in the direction of the negative partial derivative.

By **chain rule**, that means that we'll also need to have some kind of **partial derivative of our activation function** with respect to our weights and biases. If the **slope of an activation function** has a tendency to **explode** or **vanish**, then our gradient might also explode or vanish which means we end up taking **steps in our weights** that are **too large** or **too small**.

TASK 1: Sigmoid, Relu, Leaky Relu

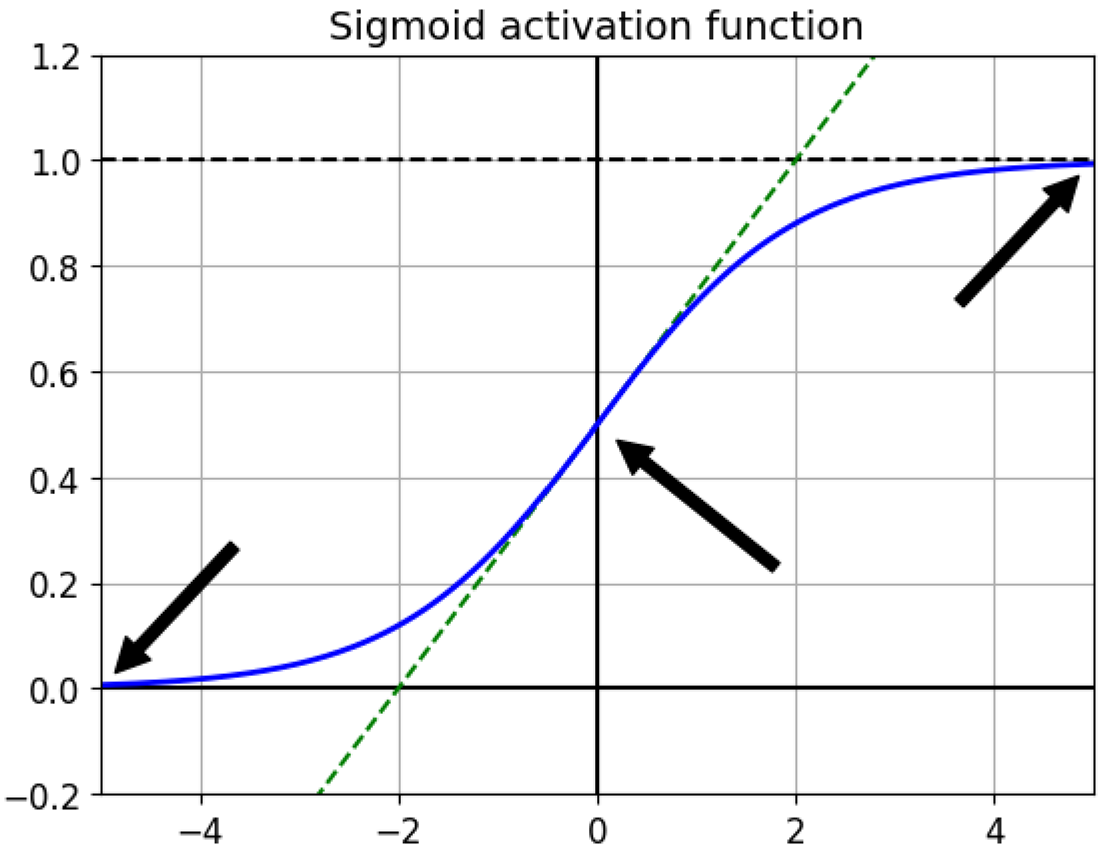
In [4]:
```python
def logit(z):
    return 1 / (1 + np.exp(-z))
```

In [5]:
```python
z = np.linspace(-5, 5, 200)

plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [1, 1], 'k--')
plt.plot([0, 0], [-0.2, 1.2], 'k-')
plt.plot([-5, 5], [-3/4, 7/4], 'g--')
plt.plot(z, logit(z), "b-", linewidth=2)
props = dict(facecolor='black', shrink=0.1)
plt.annotate('', xytext=(3.5, 0.7), xy=(5, 1), arrowprops=props, fontsize=14, ha="center")
```

```python
plt.annotate('', xytext=(-3.5, 0.3), xy=(-5, 0), arrowprops=props, fontsize=14, ha="center")
plt.annotate('', xytext=(2, 0.2), xy=(0, 0.5), arrowprops=props, fontsize=14, ha="center")
plt.grid(True)
plt.title("Sigmoid activation function", fontsize=14)
plt.axis([-5, 5, -0.2, 1.2])

plt.show()
```



Task 1 a) Describe the sigmoid activation function in the three indicated regions in the above plot.

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

Task 1 a) answer:

In the left region, the slope is very small. Since the gradient is nearly zero, backpropagation through this region causes vanishing gradients. In the middle region, The slope is the highest in this region, meaning small changes in x result in significant changes in the sigmoid activation function. The right region is similar to the left region.

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

Leaky ReLU

Task 1 b) Write the leaky relu#Leaky_ReLU) function as `def leaky_relu():`.
It should take `z` as argument and also another optional argument `alpha` with default value 0.01.
The leaky relu function is defined to be `alpha*z` for z<0 and `z` for z>0.

(alterntively you can think about using `np.maximum` to make the distinction, assuming alpha>0)

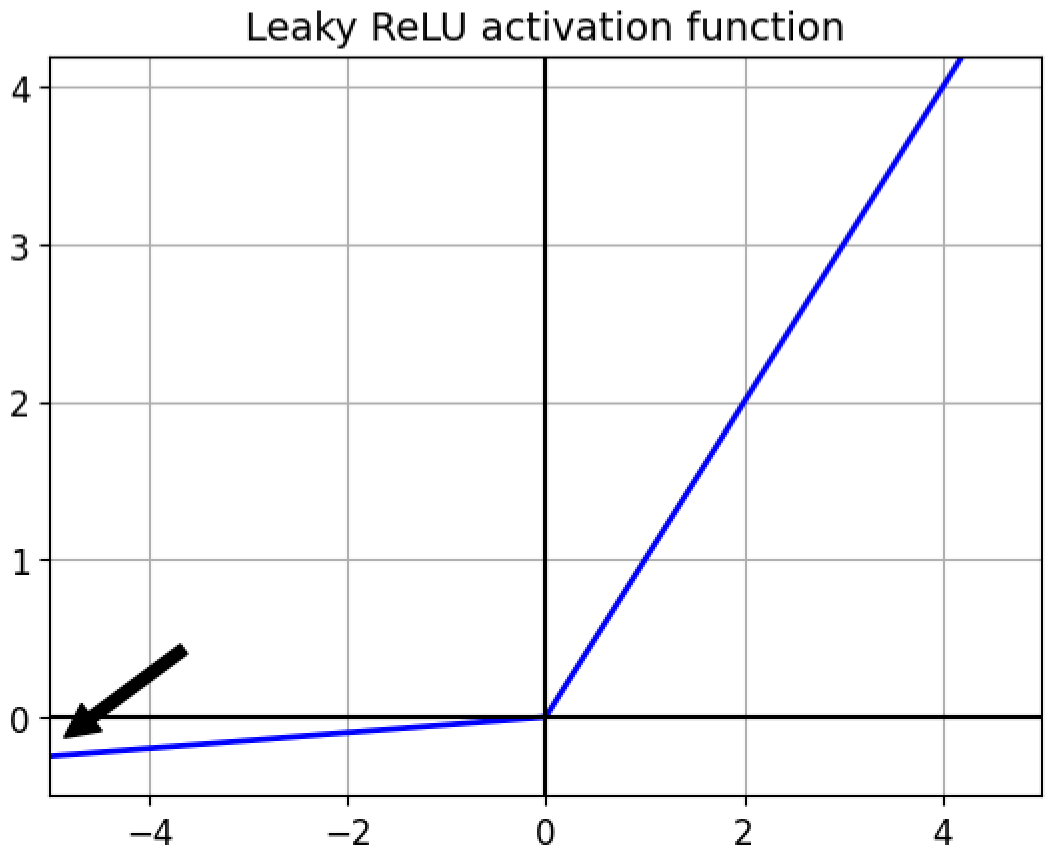
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [6]:
```
def leaky_relu(z, alpha=0.01):
    return np.maximum(alpha * z, z)
```

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

In [7]:
```
plt.plot(z, leaky_relu(z, 0.05), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([0, 0], [-0.5, 4.2], 'k-')
plt.grid(True)
props = dict(facecolor='black', shrink=0.1)
plt.annotate('', xytext=(-3.5, 0.5), xy=(-5, -0.2), arrowprops=props, fontsize=14, ha="center")
plt.title("Leaky ReLU activation function", fontsize=14)
plt.axis([-5, 5, -0.5, 4.2])

plt.show()
```



Task 1c) Describe the difference between relu and leaky relu? Also explain why one might want to use leaky relu.

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

Task 1c) answer:

ReLU completely sets all negative values to zero. However, Leaky ReLU allows small negative values through by using a small slope alpha. In ReLU, neurons with negative inputs stop learning because their gradient is zero. Leaky ReLU provides a small gradient even for negative inputs, ensuring continuous learning. That's why one might want to use leaky relu.

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

Let's train a neural network on Fashion MNIST using the Leaky ReLU:

In [8]:
```
# load fashion MNIST + train_test split
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
X_train_full = X_train_full / 255.0
```

```python
X_test = X_test / 255.0
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

```
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 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

In [9]:

```python
class ClassificationDataset(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 [10]:

```python
train_data = ClassificationDataset(X_train, y_train)
valid_data = ClassificationDataset(X_valid, y_valid)
test_data = ClassificationDataset(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)
```

In [11]:

```python
torch.manual_seed(42)
np.random.seed(42)

model = torch.nn.Sequential(
    nn.Flatten(),
    nn.Linear(28*28, 300),
    nn.LeakyReLU(),
    nn.Linear(300, 100),
    nn.LeakyReLU(),
    nn.Linear(100, 10),
)
```

In [12]:

```python
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric):
    history = {
        'epoch': [],
        'train_loss': [],
```

```python
        'train_metric': [],
        'val_loss': [],
        'val_metric': []
    }  # Initialize a dictionary to store epoch-wise results

    with torch.no_grad():
        proper_dtype = torch.int64
        X,y = next(iter(train_loader))
        try:
            loss = criterion(model(X), y.to(proper_dtype))
        except:
            try:
                proper_dtype = torch.float32
                loss = criterion(model(X), y.to(proper_dtype))
            except:
                print("No valid data-type could be found")

    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:
            y = y.to(proper_dtype)
            optimizer.zero_grad()  # Clear existing gradients
            outputs = model(X)  # Make predictions
            loss = criterion(outputs, y)  # Compute the loss
            loss.backward()  # Compute gradients
            optimizer.step()  # Update model parameters

            epoch_loss += loss.item()
            epoch_metric += metric(outputs, y)

        # 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:
                y_val = y_val.to(proper_dtype)
                outputs_val = model(X_val)  # Make predictions
```

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

            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
```

In [13]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

def accuracy_metric(pred, target):
    if len(pred.shape) == 1:
        accuracy = torch.sum(torch.eq(pred > 0.5, target)).item() / len(pred)
    else:
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    return accuracy
```

In [14]:
```
history, model = train_and_validate(train_loader, valid_loader, model,
                                    optimizer=optimizer, criterion=criterion,
                                    num_epochs=10, metric=accuracy_metric)
```

```
Epoch [1/10], Train Loss: 2.2718, Train Metric: 0.2384, Val Loss: 2.2315, Val Metric: 0.3792
Epoch [2/10], Train Loss: 2.1785, Train Metric: 0.4119, Val Loss: 2.1116, Val Metric: 0.4159
Epoch [3/10], Train Loss: 2.0091, Train Metric: 0.4537, Val Loss: 1.8796, Val Metric: 0.5184
Epoch [4/10], Train Loss: 1.7300, Train Metric: 0.5712, Val Loss: 1.5703, Val Metric: 0.6133
Epoch [5/10], Train Loss: 1.4556, Train Metric: 0.6115, Val Loss: 1.3360, Val Metric: 0.6248
Epoch [6/10], Train Loss: 1.2630, Train Metric: 0.6273, Val Loss: 1.1769, Val Metric: 0.6414
Epoch [7/10], Train Loss: 1.1285, Train Metric: 0.6417, Val Loss: 1.0645, Val Metric: 0.6549
Epoch [8/10], Train Loss: 1.0308, Train Metric: 0.6549, Val Loss: 0.9806, Val Metric: 0.6711
Epoch [9/10], Train Loss: 0.9576, Train Metric: 0.6683, Val Loss: 0.9182, Val Metric: 0.6808
Epoch [10/10], Train Loss: 0.9013, Train Metric: 0.6775, Val Loss: 0.8705, Val Metric: 0.6913
```

TASK 2: ELU

Task 2 a) Describe the ELU activation function and compare to LeakyRelu.
The definition is described in Chapter 11 or alternatively at the above link.

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

Task 2 a) answer:

The ELU function is: $f(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha(e^z - 1), & \text{if } z < 0 \end{cases}$

Compared to LeakyRelu, ELU has an exponential curve for negative values, making it smoother. ELU can push mean activations closer to zero.

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

Task 2 b) Similar to `leaky_relu` above, write the function `def elu():` as a function of z with optional argument `alpha=1` (meaning that the default value is 1).

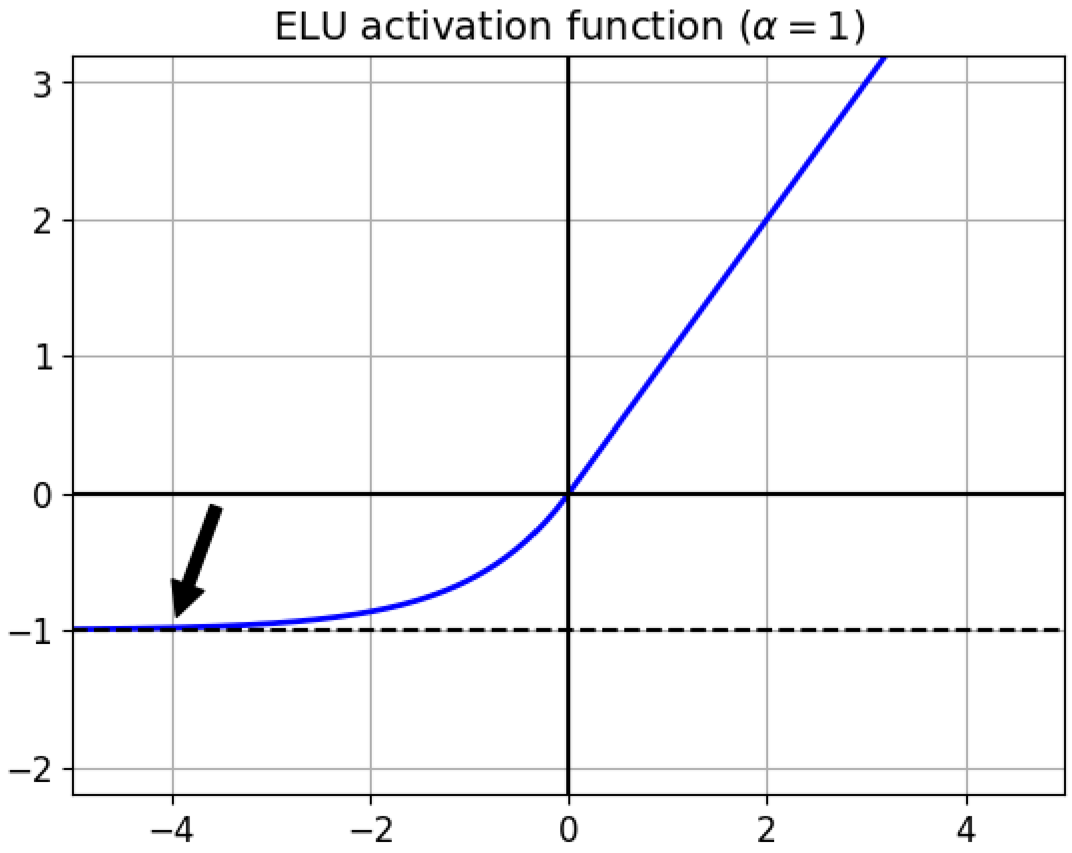
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [15]:
```
def elu(z, alpha=1):
    return np.where(z < 0, alpha * (np.exp(z) - 1), z)
```

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

In [16]:
```
plt.plot(z, elu(z), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [-1, -1], 'k--')
plt.plot([0, 0], [-2.2, 3.2], 'k-')
plt.grid(True)
plt.title(r"ELU activation function ($\alpha=1$)", fontsize=14)
plt.annotate('', xytext=(-3.5, 0), xy=(-4, -1), arrowprops=props, fontsize=14, ha="center")
plt.axis([-5, 5, -2.2, 3.2])

plt.show()
```



In [17]:
```
torch.manual_seed(42)
np.random.seed(42)
```

To use the elu activation function in TensorFlow you need to specify the activation function when building each layer (Check on the Pytorch Website for some examples):
`nn.ELU()`

Task 2 c) Using the same layer dimensions from the previous model (LeakyRelu), train with ELU activation instead.

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

In [18]:
```
model = torch.nn.Sequential(
    nn.Flatten(),
    nn.Linear(28*28, 300),
    nn.ELU(),
```

```
    nn.Linear(300, 100),
    nn.ELU(),
    nn.Linear(100, 10),
)
```

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

In [19]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

def accuracy_metric(pred, target):
    if len(pred.shape) == 1:
        accuracy = torch.sum(torch.eq(pred > 0.5, target)).item() / len(pred)
    else:
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    return accuracy
```

In [20]:
```
history, model = train_and_validate(train_loader, valid_loader, model,
                                    optimizer=optimizer, criterion=criterion,
                                    num_epochs=1, metric=accuracy_metric)
```

```
Epoch [1/1], Train Loss: 2.1936, Train Metric: 0.3600, Val Loss: 2.0593, Val Metric: 0.5154
```

Task 3: Batch Normalization

Task 3 a) Build a NN with two hidden layers with 300 and 100 nodes. Use RELU as activation function. Add Batch Normalization layers before each dense layer (check the definition in Chapter 11)

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

In [21]:
```
model = nn.Sequential(
    nn.Flatten(),
    nn.BatchNorm1d(28*28),
    nn.Linear(28*28, 300),
    nn.ReLU(),
    nn.BatchNorm1d(300),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
```

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

In [22]:
```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

def accuracy_metric(pred, target):
    if len(pred.shape) == 1:
        accuracy = torch.sum(torch.eq(pred > 0.5, target)).item() / len(pred)
    else:
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    return accuracy
```

In [23]:
```
history, model = train_and_validate(train_loader, valid_loader, model,
                                    optimizer=optimizer, criterion=criterion,
                                    num_epochs=25, metric=accuracy_metric)
```

```
Epoch [1/25], Train Loss: 1.5208, Train Metric: 0.6060, Val Loss: 1.0514, Val Metric: 0.7261
Epoch [2/25], Train Loss: 0.9074, Train Metric: 0.7344, Val Loss: 0.7548, Val Metric: 0.7617
Epoch [3/25], Train Loss: 0.7119, Train Metric: 0.7676, Val Loss: 0.6288, Val Metric: 0.7892
Epoch [4/25], Train Loss: 0.6188, Train Metric: 0.7887, Val Loss: 0.5679, Val Metric: 0.8131
Epoch [5/25], Train Loss: 0.5634, Train Metric: 0.8059, Val Loss: 0.5133, Val Metric: 0.8291
Epoch [6/25], Train Loss: 0.5245, Train Metric: 0.8185, Val Loss: 0.4847, Val Metric: 0.8388
Epoch [7/25], Train Loss: 0.4970, Train Metric: 0.8262, Val Loss: 0.4574, Val Metric: 0.8463
Epoch [8/25], Train Loss: 0.4762, Train Metric: 0.8319, Val Loss: 0.4411, Val Metric: 0.8503
Epoch [9/25], Train Loss: 0.4597, Train Metric: 0.8375, Val Loss: 0.4337, Val Metric: 0.8552
Epoch [10/25], Train Loss: 0.4458, Train Metric: 0.8431, Val Loss: 0.4172, Val Metric: 0.8590
Epoch [11/25], Train Loss: 0.4335, Train Metric: 0.8473, Val Loss: 0.4128, Val Metric: 0.8608
Epoch [12/25], Train Loss: 0.4237, Train Metric: 0.8504, Val Loss: 0.4066, Val Metric: 0.8610
Epoch [13/25], Train Loss: 0.4139, Train Metric: 0.8533, Val Loss: 0.3935, Val Metric: 0.8621
Epoch [14/25], Train Loss: 0.4064, Train Metric: 0.8566, Val Loss: 0.3886, Val Metric: 0.8627
Epoch [15/25], Train Loss: 0.4014, Train Metric: 0.8573, Val Loss: 0.3905, Val Metric: 0.8657
Epoch [16/25], Train Loss: 0.3934, Train Metric: 0.8613, Val Loss: 0.3886, Val Metric: 0.8673
Epoch [17/25], Train Loss: 0.3842, Train Metric: 0.8643, Val Loss: 0.3787, Val Metric: 0.8691
Epoch [18/25], Train Loss: 0.3819, Train Metric: 0.8645, Val Loss: 0.3887, Val Metric: 0.8697
Epoch [19/25], Train Loss: 0.3761, Train Metric: 0.8679, Val Loss: 0.3654, Val Metric: 0.8712
Epoch [20/25], Train Loss: 0.3696, Train Metric: 0.8688, Val Loss: 0.3635, Val Metric: 0.8744
Epoch [21/25], Train Loss: 0.3660, Train Metric: 0.8707, Val Loss: 0.3706, Val Metric: 0.8740
Epoch [22/25], Train Loss: 0.3599, Train Metric: 0.8726, Val Loss: 0.3653, Val Metric: 0.8772
Epoch [23/25], Train Loss: 0.3566, Train Metric: 0.8737, Val Loss: 0.3597, Val Metric: 0.8748
Epoch [24/25], Train Loss: 0.3521, Train Metric: 0.8752, Val Loss: 0.3533, Val Metric: 0.8768
Epoch [25/25], Train Loss: 0.3468, Train Metric: 0.8761, Val Loss: 0.3544, Val Metric: 0.8770
```

In [24]:
```
print(model)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): BatchNorm1d(784, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): Linear(in_features=784, out_features=300, bias=True)
  (3): ReLU()
  (4): BatchNorm1d(300, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): Linear(in_features=300, out_features=100, bias=True)
  (6): ReLU()
  (7): Linear(in_features=100, out_features=10, bias=True)
)
```

Task 3 b) Explain what batch normalization does and discuss the results of above training.

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

Task 3b) answer:

Batch normalization normalizes activations in a neural network during training. Each input is standardized to have zero mean and unit variance. Batch Normalization acts like a regularizer, reducing the need for other regularization techniques. As shown in the results, with BatchNorm, the network achieved better loss and metric values in fewer epochs. The model with BatchNorm was able to generalize better, reaching a val metric of 0.8770.

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

Task 4: Reusing a Pytorch model

Let's split the fashion MNIST training set in two:

- `X_train_A`: all images of all items except for sandals and shirts (classes 5 and 6).
- `X_train_B`: a much smaller training set of just the first 200 images of sandals or shirts.

The validation set and the test set are also split this way, but without restricting the number of images.

We will train a model on set A (classification task with 8 classes), and try to reuse it to tackle set B (binary classification). We hope to transfer a little bit of knowledge from task A to task B, since classes in set A (sneakers, ankle boots, coats, t-shirts, etc.) are somewhat similar to classes in set B (sandals and shirts). However, **since we are using `Dense` layers, only patterns that occur at the same location can be reused** (in contrast, **convolutional**

layers will transfer much better, since learned patterns can be detected anywhere on the image, as we will see in the CNN chapter).

In [25]:
```python
def split_dataset(X, y):
    y_5_or_6 = (y == 5) | (y == 6) # sandals or shirts
    y_A = y[~y_5_or_6]
    y_A[y_A > 6] -= 2 # class indices 7, 8, 9 should be moved to 5, 6, 7
    y_B = (y[y_5_or_6] == 6).astype(np.float32) # binary classification task: is it a shirt (class 6)?
    return ((X[~y_5_or_6], y_A),
            (X[y_5_or_6], y_B))

(X_train_A, y_train_A), (X_train_B, y_train_B) = split_dataset(X_train, y_train)
(X_valid_A, y_valid_A), (X_valid_B, y_valid_B) = split_dataset(X_valid, y_valid)
(X_test_A, y_test_A), (X_test_B, y_test_B) = split_dataset(X_test, y_test)
X_train_B = X_train_B[:200]
y_train_B = y_train_B[:200]
```

In [26]:
```python
class ClassificationDataset(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 [27]:
```python
train_data = ClassificationDataset(X_train_A, y_train_A)
valid_data = ClassificationDataset(X_valid_A, y_valid_A)
test_data = ClassificationDataset(X_test_A, y_test_A)

train_loader_A = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader_A = DataLoader(test_data, batch_size=64, shuffle=False)
valid_loader_A = DataLoader(valid_data, batch_size=64, shuffle=False)
```

In [28]:
```python
train_data = ClassificationDataset(X_train_B, y_train_B)
valid_data = ClassificationDataset(X_valid_B, y_valid_B)
test_data = ClassificationDataset(X_test_B, y_test_B)

train_loader_B = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader_B = DataLoader(test_data, batch_size=64, shuffle=False)
valid_loader_B = DataLoader(valid_data, batch_size=64, shuffle=False)
```

In [29]:
```
torch.manual_seed(42)
np.random.seed(42)
```

In [30]:
```
model_A = nn.Sequential()
model_A.append(nn.Flatten())
n_last = 28*28
for n_hidden in (300, 100, 50, 50, 50):
    model_A.append(nn.Linear(n_last, n_hidden))
    model_A.append(nn.SELU())
    n_last = n_hidden
model_A.append(nn.Linear(n_last, 8))
```

Out[30]:
```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=300, bias=True)
  (2): SELU()
  (3): Linear(in_features=300, out_features=100, bias=True)
  (4): SELU()
  (5): Linear(in_features=100, out_features=50, bias=True)
  (6): SELU()
  (7): Linear(in_features=50, out_features=50, bias=True)
  (8): SELU()
  (9): Linear(in_features=50, out_features=50, bias=True)
  (10): SELU()
  (11): Linear(in_features=50, out_features=8, bias=True)
)
```

In [31]:
```
criterion_A = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_A.parameters(), lr=0.001)

def accuracy_metric(pred, target):
    if len(pred.shape) == 1:
        accuracy = torch.sum(torch.eq(pred > 0.5, target)).item() / len(pred)
    else:
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    return accuracy
```

In [32]:
```
history_A, model_A = train_and_validate(train_loader_A, valid_loader_A, model_A,
                                        optimizer=optimizer, criterion=criterion_A,
                                        num_epochs=30, metric=accuracy_metric)
```

```
Epoch [1/30], Train Loss: 1.9313, Train Metric: 0.4385, Val Loss: 1.7502, Val Metric: 0.5501
Epoch [2/30], Train Loss: 1.5114, Train Metric: 0.6223, Val Loss: 1.2759, Val Metric: 0.6607
Epoch [3/30], Train Loss: 1.1340, Train Metric: 0.6585, Val Loss: 1.0003, Val Metric: 0.6815
Epoch [4/30], Train Loss: 0.9253, Train Metric: 0.6988, Val Loss: 0.8443, Val Metric: 0.7169
Epoch [5/30], Train Loss: 0.7979, Train Metric: 0.7232, Val Loss: 0.7438, Val Metric: 0.7372
Epoch [6/30], Train Loss: 0.7080, Train Metric: 0.7589, Val Loss: 0.6662, Val Metric: 0.7966
Epoch [7/30], Train Loss: 0.6326, Train Metric: 0.8105, Val Loss: 0.5969, Val Metric: 0.8352
Epoch [8/30], Train Loss: 0.5643, Train Metric: 0.8379, Val Loss: 0.5326, Val Metric: 0.8477
Epoch [9/30], Train Loss: 0.5059, Train Metric: 0.8471, Val Loss: 0.4812, Val Metric: 0.8529
Epoch [10/30], Train Loss: 0.4611, Train Metric: 0.8544, Val Loss: 0.4426, Val Metric: 0.8606
Epoch [11/30], Train Loss: 0.4292, Train Metric: 0.8600, Val Loss: 0.4149, Val Metric: 0.8635
Epoch [12/30], Train Loss: 0.4064, Train Metric: 0.8645, Val Loss: 0.3952, Val Metric: 0.8677
Epoch [13/30], Train Loss: 0.3897, Train Metric: 0.8683, Val Loss: 0.3815, Val Metric: 0.8681
Epoch [14/30], Train Loss: 0.3763, Train Metric: 0.8711, Val Loss: 0.3689, Val Metric: 0.8741
Epoch [15/30], Train Loss: 0.3658, Train Metric: 0.8733, Val Loss: 0.3585, Val Metric: 0.8781
Epoch [16/30], Train Loss: 0.3566, Train Metric: 0.8764, Val Loss: 0.3512, Val Metric: 0.8791
Epoch [17/30], Train Loss: 0.3491, Train Metric: 0.8783, Val Loss: 0.3446, Val Metric: 0.8818
Epoch [18/30], Train Loss: 0.3426, Train Metric: 0.8796, Val Loss: 0.3410, Val Metric: 0.8822
Epoch [19/30], Train Loss: 0.3367, Train Metric: 0.8817, Val Loss: 0.3322, Val Metric: 0.8859
Epoch [20/30], Train Loss: 0.3320, Train Metric: 0.8834, Val Loss: 0.3269, Val Metric: 0.8870
Epoch [21/30], Train Loss: 0.3267, Train Metric: 0.8848, Val Loss: 0.3224, Val Metric: 0.8902
Epoch [22/30], Train Loss: 0.3223, Train Metric: 0.8863, Val Loss: 0.3217, Val Metric: 0.8913
Epoch [23/30], Train Loss: 0.3183, Train Metric: 0.8878, Val Loss: 0.3162, Val Metric: 0.8926
Epoch [24/30], Train Loss: 0.3141, Train Metric: 0.8897, Val Loss: 0.3113, Val Metric: 0.8906
Epoch [25/30], Train Loss: 0.3106, Train Metric: 0.8910, Val Loss: 0.3092, Val Metric: 0.8953
Epoch [26/30], Train Loss: 0.3069, Train Metric: 0.8929, Val Loss: 0.3032, Val Metric: 0.8976
Epoch [27/30], Train Loss: 0.3038, Train Metric: 0.8935, Val Loss: 0.3022, Val Metric: 0.8956
Epoch [28/30], Train Loss: 0.3007, Train Metric: 0.8956, Val Loss: 0.2978, Val Metric: 0.9004
Epoch [29/30], Train Loss: 0.2974, Train Metric: 0.8966, Val Loss: 0.2947, Val Metric: 0.9029
Epoch [30/30], Train Loss: 0.2955, Train Metric: 0.8977, Val Loss: 0.2928, Val Metric: 0.9026
```

In [33]:

```
model_B = nn.Sequential()
model_B.append(nn.Flatten())
n_last = 28*28
for n_hidden in (300, 100, 50, 50, 50):
    model_B.append(nn.Linear(n_last, n_hidden))
    model_B.append(nn.SELU())
    n_last = n_hidden
model_B.append(nn.Linear(n_last, 1))
model_B.append(nn.Sigmoid())
model_B.append(nn.Flatten(start_dim=0))
```

Out[33]:

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=300, bias=True)
  (2): SELU()
  (3): Linear(in_features=300, out_features=100, bias=True)
  (4): SELU()
  (5): Linear(in_features=100, out_features=50, bias=True)
  (6): SELU()
  (7): Linear(in_features=50, out_features=50, bias=True)
  (8): SELU()
  (9): Linear(in_features=50, out_features=50, bias=True)
  (10): SELU()
  (11): Linear(in_features=50, out_features=1, bias=True)
  (12): Sigmoid()
  (13): Flatten(start_dim=0, end_dim=-1)
)
```

In [34]:

```
criterion_B = nn.BCELoss()
optimizer = torch.optim.SGD(model_B.parameters(), lr=0.001)

def accuracy_metric(pred, target):
    if len(pred.shape) == 1:
        accuracy = torch.sum(torch.eq(pred > 0.5, target)).item() / len(pred)
    else:
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    return accuracy
```

In [35]:

```
history_B, model_B = train_and_validate(train_loader_B, valid_loader_B, model_B,
                                        optimizer=optimizer, criterion=criterion_B,
                                        num_epochs=30, metric=accuracy_metric)
```

```
Epoch [1/30], Train Loss: 0.6977, Train Metric: 0.4102, Val Loss: 0.6892, Val Metric: 0.5014
Epoch [2/30], Train Loss: 0.6879, Train Metric: 0.5195, Val Loss: 0.6883, Val Metric: 0.5014
Epoch [3/30], Train Loss: 0.6886, Train Metric: 0.4922, Val Loss: 0.6875, Val Metric: 0.5014
Epoch [4/30], Train Loss: 0.6932, Train Metric: 0.4375, Val Loss: 0.6867, Val Metric: 0.5014
Epoch [5/30], Train Loss: 0.6831, Train Metric: 0.5469, Val Loss: 0.6858, Val Metric: 0.5014
Epoch [6/30], Train Loss: 0.6886, Train Metric: 0.4648, Val Loss: 0.6850, Val Metric: 0.5014
Epoch [7/30], Train Loss: 0.6871, Train Metric: 0.4688, Val Loss: 0.6842, Val Metric: 0.5014
Epoch [8/30], Train Loss: 0.6841, Train Metric: 0.4961, Val Loss: 0.6834, Val Metric: 0.5014
Epoch [9/30], Train Loss: 0.6848, Train Metric: 0.4688, Val Loss: 0.6826, Val Metric: 0.5014
Epoch [10/30], Train Loss: 0.6831, Train Metric: 0.4961, Val Loss: 0.6818, Val Metric: 0.5014
Epoch [11/30], Train Loss: 0.6829, Train Metric: 0.4688, Val Loss: 0.6810, Val Metric: 0.5014
Epoch [12/30], Train Loss: 0.6831, Train Metric: 0.4688, Val Loss: 0.6802, Val Metric: 0.5014
Epoch [13/30], Train Loss: 0.6787, Train Metric: 0.5234, Val Loss: 0.6794, Val Metric: 0.5014
Epoch [14/30], Train Loss: 0.6884, Train Metric: 0.4141, Val Loss: 0.6787, Val Metric: 0.5014
Epoch [15/30], Train Loss: 0.6731, Train Metric: 0.5508, Val Loss: 0.6779, Val Metric: 0.5014
Epoch [16/30], Train Loss: 0.6849, Train Metric: 0.4414, Val Loss: 0.6772, Val Metric: 0.5014
Epoch [17/30], Train Loss: 0.6827, Train Metric: 0.4414, Val Loss: 0.6765, Val Metric: 0.5024
Epoch [18/30], Train Loss: 0.6746, Train Metric: 0.4961, Val Loss: 0.6757, Val Metric: 0.5034
Epoch [19/30], Train Loss: 0.6750, Train Metric: 0.4961, Val Loss: 0.6749, Val Metric: 0.5034
Epoch [20/30], Train Loss: 0.6766, Train Metric: 0.4961, Val Loss: 0.6741, Val Metric: 0.5034
Epoch [21/30], Train Loss: 0.6740, Train Metric: 0.4961, Val Loss: 0.6733, Val Metric: 0.5044
Epoch [22/30], Train Loss: 0.6764, Train Metric: 0.4688, Val Loss: 0.6726, Val Metric: 0.5053
Epoch [23/30], Train Loss: 0.6722, Train Metric: 0.4961, Val Loss: 0.6718, Val Metric: 0.5063
Epoch [24/30], Train Loss: 0.6717, Train Metric: 0.5234, Val Loss: 0.6711, Val Metric: 0.5073
Epoch [25/30], Train Loss: 0.6762, Train Metric: 0.4414, Val Loss: 0.6704, Val Metric: 0.5122
Epoch [26/30], Train Loss: 0.6683, Train Metric: 0.5000, Val Loss: 0.6696, Val Metric: 0.5141
Epoch [27/30], Train Loss: 0.6599, Train Metric: 0.5820, Val Loss: 0.6687, Val Metric: 0.5131
Epoch [28/30], Train Loss: 0.6738, Train Metric: 0.4414, Val Loss: 0.6679, Val Metric: 0.5151
Epoch [29/30], Train Loss: 0.6651, Train Metric: 0.5273, Val Loss: 0.6671, Val Metric: 0.5151
Epoch [30/30], Train Loss: 0.6641, Train Metric: 0.5273, Val Loss: 0.6663, Val Metric: 0.5151
```

In [36]:

```
model_B_on_A = nn.Sequential()
for module in list(model_A.modules())[1:]:
    model_B_on_A.append(module)
model_B_on_A.append(nn.Linear(8, 1))
model_B_on_A.append(nn.Sigmoid())
model_B_on_A.append(nn.Flatten(start_dim=0))
```

Out[36]:
```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=300, bias=True)
  (2): SELU()
  (3): Linear(in_features=300, out_features=100, bias=True)
  (4): SELU()
  (5): Linear(in_features=100, out_features=50, bias=True)
  (6): SELU()
  (7): Linear(in_features=50, out_features=50, bias=True)
  (8): SELU()
  (9): Linear(in_features=50, out_features=50, bias=True)
  (10): SELU()
  (11): Linear(in_features=50, out_features=8, bias=True)
  (12): Linear(in_features=8, out_features=1, bias=True)
  (13): Sigmoid()
  (14): Flatten(start_dim=0, end_dim=-1)
)
```

Note that `model_B_on_A` and `model_A` actually share layers now, so **when we train one, it will update both models**. If we want **to avoid that**, we need to **build `model_B_on_A` on top of a clone of `model_A`**:

In [37]:
```python
import copy
```

In [38]:
```python
model_A_clone = copy.deepcopy(model_A)
model_A_clone.load_state_dict(model_A.state_dict())
```

Out[38]:
```
<All keys matched successfully>
```

In [39]:
```python
for param in model_B_on_A.parameters():
    param.requires_grad = False
```

In [40]:
```python
optimizer = torch.optim.SGD(model_B_on_A.parameters(), lr=0.001)
```

In [41]:
```python
for param in model_B_on_A.parameters():
    param.requires_grad = True

history_B_on_A, model_B_on_A = train_and_validate(train_loader_B, valid_loader_B, model_B_on_A,
                                                  optimizer=optimizer, criterion=criterion_B,
                                                  num_epochs=30, metric=accuracy_metric)
```

```
Epoch [1/30], Train Loss: 1.4013, Train Metric: 0.0938, Val Loss: 1.1492, Val Metric: 0.1835
Epoch [2/30], Train Loss: 1.1006, Train Metric: 0.1719, Val Loss: 0.9850, Val Metric: 0.2553
Epoch [3/30], Train Loss: 0.9084, Train Metric: 0.3594, Val Loss: 0.8502, Val Metric: 0.3811
Epoch [4/30], Train Loss: 0.7989, Train Metric: 0.4492, Val Loss: 0.7369, Val Metric: 0.5260
Epoch [5/30], Train Loss: 0.7037, Train Metric: 0.6016, Val Loss: 0.6398, Val Metric: 0.6552
Epoch [6/30], Train Loss: 0.5925, Train Metric: 0.6875, Val Loss: 0.5652, Val Metric: 0.7322
Epoch [7/30], Train Loss: 0.5261, Train Metric: 0.8164, Val Loss: 0.5045, Val Metric: 0.7991
Epoch [8/30], Train Loss: 0.4771, Train Metric: 0.8320, Val Loss: 0.4536, Val Metric: 0.8542
Epoch [9/30], Train Loss: 0.3928, Train Metric: 0.8984, Val Loss: 0.4137, Val Metric: 0.8873
Epoch [10/30], Train Loss: 0.3638, Train Metric: 0.9453, Val Loss: 0.3772, Val Metric: 0.9234
Epoch [11/30], Train Loss: 0.3231, Train Metric: 0.9727, Val Loss: 0.3463, Val Metric: 0.9410
Epoch [12/30], Train Loss: 0.3041, Train Metric: 0.9844, Val Loss: 0.3216, Val Metric: 0.9537
Epoch [13/30], Train Loss: 0.2857, Train Metric: 0.9883, Val Loss: 0.2987, Val Metric: 0.9595
Epoch [14/30], Train Loss: 0.2589, Train Metric: 0.9922, Val Loss: 0.2790, Val Metric: 0.9663
Epoch [15/30], Train Loss: 0.2516, Train Metric: 0.9961, Val Loss: 0.2614, Val Metric: 0.9732
Epoch [16/30], Train Loss: 0.2191, Train Metric: 0.9961, Val Loss: 0.2459, Val Metric: 0.9761
Epoch [17/30], Train Loss: 0.2246, Train Metric: 0.9961, Val Loss: 0.2314, Val Metric: 0.9795
Epoch [18/30], Train Loss: 0.1872, Train Metric: 0.9961, Val Loss: 0.2191, Val Metric: 0.9834
Epoch [19/30], Train Loss: 0.1955, Train Metric: 0.9961, Val Loss: 0.2072, Val Metric: 0.9844
Epoch [20/30], Train Loss: 0.1830, Train Metric: 0.9961, Val Loss: 0.1966, Val Metric: 0.9854
Epoch [21/30], Train Loss: 0.1634, Train Metric: 0.9961, Val Loss: 0.1877, Val Metric: 0.9854
Epoch [22/30], Train Loss: 0.1546, Train Metric: 1.0000, Val Loss: 0.1796, Val Metric: 0.9863
Epoch [23/30], Train Loss: 0.1408, Train Metric: 1.0000, Val Loss: 0.1720, Val Metric: 0.9863
Epoch [24/30], Train Loss: 0.1298, Train Metric: 1.0000, Val Loss: 0.1656, Val Metric: 0.9863
Epoch [25/30], Train Loss: 0.1340, Train Metric: 1.0000, Val Loss: 0.1591, Val Metric: 0.9873
Epoch [26/30], Train Loss: 0.1217, Train Metric: 1.0000, Val Loss: 0.1534, Val Metric: 0.9873
Epoch [27/30], Train Loss: 0.1235, Train Metric: 1.0000, Val Loss: 0.1478, Val Metric: 0.9873
Epoch [28/30], Train Loss: 0.1169, Train Metric: 1.0000, Val Loss: 0.1427, Val Metric: 0.9873
Epoch [29/30], Train Loss: 0.1072, Train Metric: 1.0000, Val Loss: 0.1383, Val Metric: 0.9883
Epoch [30/30], Train Loss: 0.1060, Train Metric: 1.0000, Val Loss: 0.1340, Val Metric: 0.9883
```

Task 4: a) Evaluate the loss and accuracy of the two models `model_B` and `model_B_on_A` on the sandals/shirts dataset.

In [42]:
```python
def test_model(model, data_loader, criterion, metric=None):
    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():
        proper_dtype = torch.int64
        X,y = next(iter(train_loader))
        try:
            loss = criterion(model(X), y.to(proper_dtype))
        except:
```

```python
        try:
            proper_dtype = torch.float32
            loss = criterion(model(X), y.to(proper_dtype))
        except:
            print("No valid data-type could be found")


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

            # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
            if metric is not None:
                total_metric += metric(outputs, y)
            else:
                total_metric += 0.0

    # 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
```

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

In [43]:
```python
avg_loss1, avg_metric1 = test_model(model_B, test_loader_B, criterion_B, metric = accuracy_metric)
avg_loss2, avg_metric2 = test_model(model_B_on_A, test_loader_B, criterion_B, metric = accuracy_metric)
```

```
No valid data-type could be found
Test Loss: 0.6670, Test Metric: 0.5137
No valid data-type could be found
Test Loss: 0.1254, Test Metric: 0.9907
```

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

b) In your own words, explain above "transfer learning". Did it help?

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

Task

Transfer Learning is a deep learning technique where a pre-trained model is used as a starting point for a new task. Instead of training a neural network from scratch, we take a model that has already been trained on a large dataset and fine-tune it for a different but related problem. We can see from the above results that model_B_on_A has a lower test loss and the accuracy of model_B_on_A is much higher than model_B. This suggests that transfer learning significantly boosted performance, likely due to leveraging knowledge from the larger dataset A.

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

Task 5: Learning Rate Scheduling

Just like when we learned about SGD for linear regression, **decreasing the learning rate over time can improve convergence**. One way to do this is to write a schedule to decay the learning rate as a function of epoch number.

Let's add a an exponential decay of the learning rate:

We will use the following learning rate schedule (exponential):
$lr = lr_0 \cdot 0.1^{epoch/20}$

In [44]:
```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Note: If you want to use learning rate decay, it is probably better to use a Pytorch built-in function like ExponentialLR and not program it yourself.

In order to make our training loop more flexible, we'll add **default values** of `None` for the **metric** and **scheduler**. Depending on the task, we might not always need or want to use a metric or scheduler but this will be our new train and validation loop. We've also added **learning rate tracking** to our model history output.

In [45]:
```
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None, scheduler=None):
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
        'val_loss': [],
        'val_metric': [],
        'learning_rate': []
    }  # Initialize a dictionary to store epoch-wise results

    with torch.no_grad():
        proper_dtype = torch.int64
        X,y = next(iter(train_loader))
        try:
            loss = criterion(model(X), y.to(proper_dtype))
        except:
            try:
                proper_dtype = torch.float32
                loss = criterion(model(X), y.to(proper_dtype))
            except:
                print("No valid data-type could be found")

    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:
            y = y.to(proper_dtype)
            optimizer.zero_grad()  # Clear existing gradients
            outputs = model(X)  # Make predictions
            loss = criterion(outputs, y)  # Compute the loss
            loss.backward()  # Compute gradients
            optimizer.step()  # Update model parameters

            epoch_loss += loss.item()

            # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
            if metric is not None:
                epoch_metric += metric(outputs, y)
            else:
```

```
            epoch_metric += 0.0

    # 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:
            y_val = y_val.to(proper_dtype)
            outputs_val = model(X_val)  # Make predictions
            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 += 0.0

        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)
    history['learning_rate'].append(optimizer.param_groups[0]['lr'])

    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}')

    # THESE LINES ARE NEW AND ACCOUNT FOR SCHEDULERS
    if scheduler is not None:
        scheduler.step()

return history, model
```

Task 5:

- Build a NN with: two hidden layers with 300 and 100 nodes, add Batch Normalization layers before the linear layers,

- Train the model with `nn.CrossEntropyLoss()` as `criterion`, `scheduler` and `optimizer` provided above and `accuracy` as the `metric`,
- Fit the model to `train_loader` for 25 epochs. Use `valid_loader` for the validation data.

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

In [46]:
```
model = nn.Sequential(
    nn.Flatten(),
    nn.BatchNorm1d(28*28),
    nn.Linear(28*28, 300),
    nn.ReLU(),
    nn.BatchNorm1d(300),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.NAdam(model.parameters())
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, exponential_decay_fn)
```

In [47]:
```
history, model = train_and_validate(train_loader, valid_loader, model, optimizer = optimizer, criterion=criterion, num_epochs =
```

```
Epoch [1/25], Train Loss: 1.0664, Train Metric: 0.7152, Val Loss: 0.6314, Val Metric: 0.8054
Epoch [2/25], Train Loss: 0.5708, Train Metric: 0.8152, Val Loss: 0.4754, Val Metric: 0.8432
Epoch [3/25], Train Loss: 0.4741, Train Metric: 0.8377, Val Loss: 0.4178, Val Metric: 0.8616
Epoch [4/25], Train Loss: 0.4316, Train Metric: 0.8492, Val Loss: 0.3911, Val Metric: 0.8661
Epoch [5/25], Train Loss: 0.4058, Train Metric: 0.8572, Val Loss: 0.3789, Val Metric: 0.8651
Epoch [6/25], Train Loss: 0.3902, Train Metric: 0.8621, Val Loss: 0.3651, Val Metric: 0.8732
Epoch [7/25], Train Loss: 0.3780, Train Metric: 0.8651, Val Loss: 0.3612, Val Metric: 0.8712
Epoch [8/25], Train Loss: 0.3692, Train Metric: 0.8696, Val Loss: 0.3562, Val Metric: 0.8736
Epoch [9/25], Train Loss: 0.3609, Train Metric: 0.8715, Val Loss: 0.3507, Val Metric: 0.8782
Epoch [10/25], Train Loss: 0.3547, Train Metric: 0.8738, Val Loss: 0.3452, Val Metric: 0.8762
Epoch [11/25], Train Loss: 0.3488, Train Metric: 0.8747, Val Loss: 0.3433, Val Metric: 0.8768
Epoch [12/25], Train Loss: 0.3446, Train Metric: 0.8762, Val Loss: 0.3389, Val Metric: 0.8786
Epoch [13/25], Train Loss: 0.3411, Train Metric: 0.8785, Val Loss: 0.3365, Val Metric: 0.8784
Epoch [14/25], Train Loss: 0.3381, Train Metric: 0.8791, Val Loss: 0.3355, Val Metric: 0.8813
Epoch [15/25], Train Loss: 0.3352, Train Metric: 0.8799, Val Loss: 0.3385, Val Metric: 0.8815
Epoch [16/25], Train Loss: 0.3318, Train Metric: 0.8817, Val Loss: 0.3320, Val Metric: 0.8819
Epoch [17/25], Train Loss: 0.3308, Train Metric: 0.8816, Val Loss: 0.3308, Val Metric: 0.8817
Epoch [18/25], Train Loss: 0.3291, Train Metric: 0.8812, Val Loss: 0.3322, Val Metric: 0.8823
Epoch [19/25], Train Loss: 0.3269, Train Metric: 0.8836, Val Loss: 0.3255, Val Metric: 0.8841
Epoch [20/25], Train Loss: 0.3256, Train Metric: 0.8835, Val Loss: 0.3315, Val Metric: 0.8825
Epoch [21/25], Train Loss: 0.3247, Train Metric: 0.8840, Val Loss: 0.3310, Val Metric: 0.8811
Epoch [22/25], Train Loss: 0.3220, Train Metric: 0.8859, Val Loss: 0.3285, Val Metric: 0.8819
Epoch [23/25], Train Loss: 0.3215, Train Metric: 0.8839, Val Loss: 0.3273, Val Metric: 0.8823
Epoch [24/25], Train Loss: 0.3218, Train Metric: 0.8852, Val Loss: 0.3282, Val Metric: 0.8855
Epoch [25/25], Train Loss: 0.3198, Train Metric: 0.8854, Val Loss: 0.3306, Val Metric: 0.8829
```

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

Let's check out our model performance with a test loop.

In [48]:
```
def test_model(model, data_loader, criterion, metric=None):
    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():
        proper_dtype = torch.int64
        X,y = next(iter(train_loader))
        try:
            loss = criterion(model(X), y.to(proper_dtype))
        except:
            try:
                proper_dtype = torch.float32
                loss = criterion(model(X), y.to(proper_dtype))
            except:
```

```python
                print("No valid data-type could be found")


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

            # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
            if metric is not None:
                total_metric += metric(outputs, y)
            else:
                total_metric += 0.0

    # 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
```

In [49]:

```python
# note that the model is overfitting a lot. Might want to use dropout
# also a CNN will perform much better as we will see next Hands-On
print("train loss:", test_model(model, train_loader, nn.CrossEntropyLoss())[0])
print("test loss:", test_model(model, test_loader, nn.CrossEntropyLoss())[0])
```

```
Test Loss: 0.3111, Test Metric: 0.0000
train loss: 0.3110733171881631
Test Loss: 0.3700, Test Metric: 0.0000
test loss: 0.3699826099880182
```

In [50]:

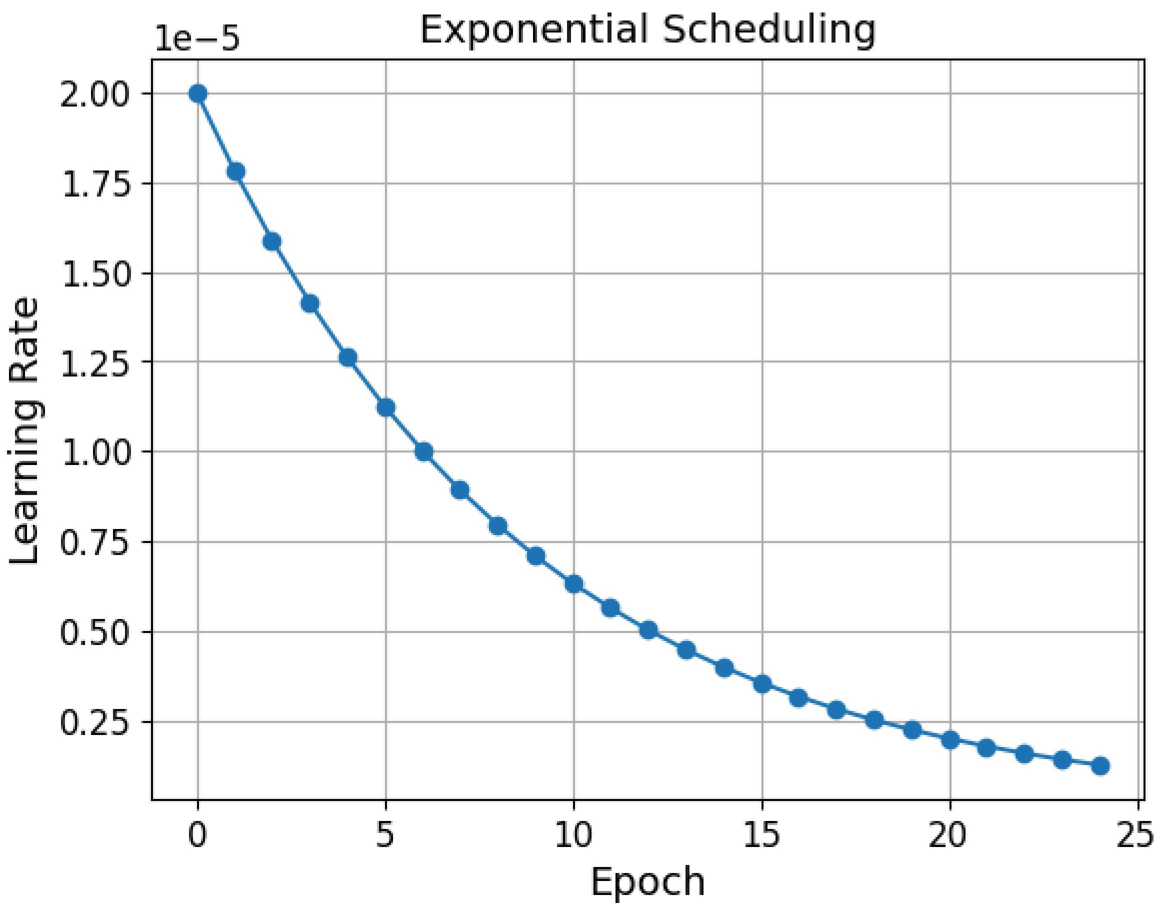
```python
# the learning rate is saved in the history under the key 'learning_rate'
print(history.keys())
```

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

In [52]:

```python
plt.plot(history['epoch'], history["learning_rate"], "o-")
plt.xlabel("Epoch")
```

```
plt.ylabel("Learning Rate")
plt.title("Exponential Scheduling", fontsize=14)
plt.grid(True)
plt.show()
```



Task 6: Performance Scheduling

Because **Loss vs. weight spaces are generally not globally convex-up**, it's likely that your model will get stuck in a **local minimum loss**. When that happens, it can mean that your **learning rate is** so **low** that it's unable to push your model weights outside of the local minimum region. We can try to **increase the learning rate** in that case **to escape a local minimum**. This is like pushing a ball further up the side of a valley in the hopes that it eventually rolls over a cliff and down into a deeper valley when you get to the top.

For performance scheduling, use the `ReduceLROnPlateau` scheduler. Example: if you step the following learning rate scheduler it will step the learning rate by 0.5 whenever the best validation loss does not improve for two consecutive epochs:

```
scheduler2 = torch.optim.lr_scheduler.ReduceLROnPlateau(factor=0.5, patience=2)
```

You will need to update the `scheduler.step()` portion of the train_and_validate loop for `ReduceLROnPlateau`. Refer to the example at the bottom of this page. This is due to the fact that this optimizer has a patience argument that will check a certain quantity to decide whether it's time to step.

In [53]:
```
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None, scheduler=None):
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
        'val_loss': [],
        'val_metric': [],
        'learning_rate': []
    }  # Initialize a dictionary to store epoch-wise results

    with torch.no_grad():
        proper_dtype = torch.int64
        X,y = next(iter(train_loader))
        try:
            loss = criterion(model(X), y.to(proper_dtype))
        except:
            try:
                proper_dtype = torch.float32
                loss = criterion(model(X), y.to(proper_dtype))
            except:
                print("No valid data-type could be found")

    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:
            y = y.to(proper_dtype)
            optimizer.zero_grad()  # Clear existing gradients
```

```
        outputs = model(X)  # Make predictions
        loss = criterion(outputs, y)  # Compute the loss
        loss.backward()  # Compute gradients
        optimizer.step()  # Update model parameters

        epoch_loss += loss.item()

        # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
        if metric is not None:
            epoch_metric += metric(outputs, y)
        else:
            epoch_metric += 0.0

    # 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:
            y_val = y_val.to(proper_dtype)
            outputs_val = model(X_val)  # Make predictions
            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 += 0.0

        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)
    history['learning_rate'].append(optimizer.param_groups[0]['lr'])

    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}')
```

```
        if scheduler is not None:
            # MODIFY THIS LINE TO WORK PROPERLY FOR REDUCELRONPLATEAU
            scheduler.step(val_loss) # This will crash if you don't fix it
    return history, model
```

Task 6:

a) Re-use (copy-paste) the NN from Task 5: two hidden layers with 300 and 100 nodes and Batch Normalization layers before the linear layers. But, now use Adam optimizer with a initial lr=0.01 and ReduceLROnPlateau scheduler.

b) Compare the results with the previous one (Task 5),

c) Comment on the learning rate as a function of epochs using the plot given below.

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

In [67]:
```
model = nn.Sequential(
    nn.Flatten(),
    nn.BatchNorm1d(28*28),
    nn.Linear(28*28, 300),
    nn.ReLU(),
    nn.BatchNorm1d(300),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.NAdam(model.parameters(),lr=0.01)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,factor=0.5, patience=2)
```

In [68]:
```
history, model = train_and_validate(train_loader, valid_loader, model, optimizer = optimizer, criterion=criterion, num_epochs =
```

```
Epoch [1/25], Train Loss: 0.4850, Train Metric: 0.8225, Val Loss: 0.3949, Val Metric: 0.8606
Epoch [2/25], Train Loss: 0.3758, Train Metric: 0.8619, Val Loss: 0.3809, Val Metric: 0.8582
Epoch [3/25], Train Loss: 0.3462, Train Metric: 0.8728, Val Loss: 1.0111, Val Metric: 0.8732
Epoch [4/25], Train Loss: 0.3340, Train Metric: 0.8770, Val Loss: 11.5039, Val Metric: 0.8758
Epoch [5/25], Train Loss: 0.3180, Train Metric: 0.8821, Val Loss: 21.1793, Val Metric: 0.8631
Epoch [6/25], Train Loss: 0.2693, Train Metric: 0.8980, Val Loss: 35.6356, Val Metric: 0.8914
Epoch [7/25], Train Loss: 0.2532, Train Metric: 0.9045, Val Loss: 9.2138, Val Metric: 0.8855
Epoch [8/25], Train Loss: 0.2471, Train Metric: 0.9067, Val Loss: 8.9410, Val Metric: 0.8888
Epoch [9/25], Train Loss: 0.2149, Train Metric: 0.9173, Val Loss: 19.3152, Val Metric: 0.8966
Epoch [10/25], Train Loss: 0.2029, Train Metric: 0.9226, Val Loss: 6.4690, Val Metric: 0.8944
Epoch [11/25], Train Loss: 0.1960, Train Metric: 0.9251, Val Loss: 0.4145, Val Metric: 0.8972
Epoch [12/25], Train Loss: 0.1746, Train Metric: 0.9334, Val Loss: 0.5814, Val Metric: 0.8997
Epoch [13/25], Train Loss: 0.1665, Train Metric: 0.9359, Val Loss: 6.5054, Val Metric: 0.9033
Epoch [14/25], Train Loss: 0.1605, Train Metric: 0.9387, Val Loss: 0.8533, Val Metric: 0.9021
Epoch [15/25], Train Loss: 0.1496, Train Metric: 0.9428, Val Loss: 2.3133, Val Metric: 0.9015
Epoch [16/25], Train Loss: 0.1446, Train Metric: 0.9446, Val Loss: 0.3931, Val Metric: 0.9029
Epoch [17/25], Train Loss: 0.1406, Train Metric: 0.9468, Val Loss: 3.5964, Val Metric: 0.9031
Epoch [18/25], Train Loss: 0.1340, Train Metric: 0.9482, Val Loss: 3.6673, Val Metric: 0.9033
Epoch [19/25], Train Loss: 0.1316, Train Metric: 0.9493, Val Loss: 3.7741, Val Metric: 0.9041
Epoch [20/25], Train Loss: 0.1293, Train Metric: 0.9496, Val Loss: 2.2670, Val Metric: 0.9047
Epoch [21/25], Train Loss: 0.1267, Train Metric: 0.9512, Val Loss: 2.5834, Val Metric: 0.9039
Epoch [22/25], Train Loss: 0.1249, Train Metric: 0.9528, Val Loss: 0.5349, Val Metric: 0.9037
Epoch [23/25], Train Loss: 0.1244, Train Metric: 0.9526, Val Loss: 4.5653, Val Metric: 0.9017
Epoch [24/25], Train Loss: 0.1204, Train Metric: 0.9535, Val Loss: 1.9649, Val Metric: 0.9025
Epoch [25/25], Train Loss: 0.1220, Train Metric: 0.9528, Val Loss: 1.6165, Val Metric: 0.9064
```

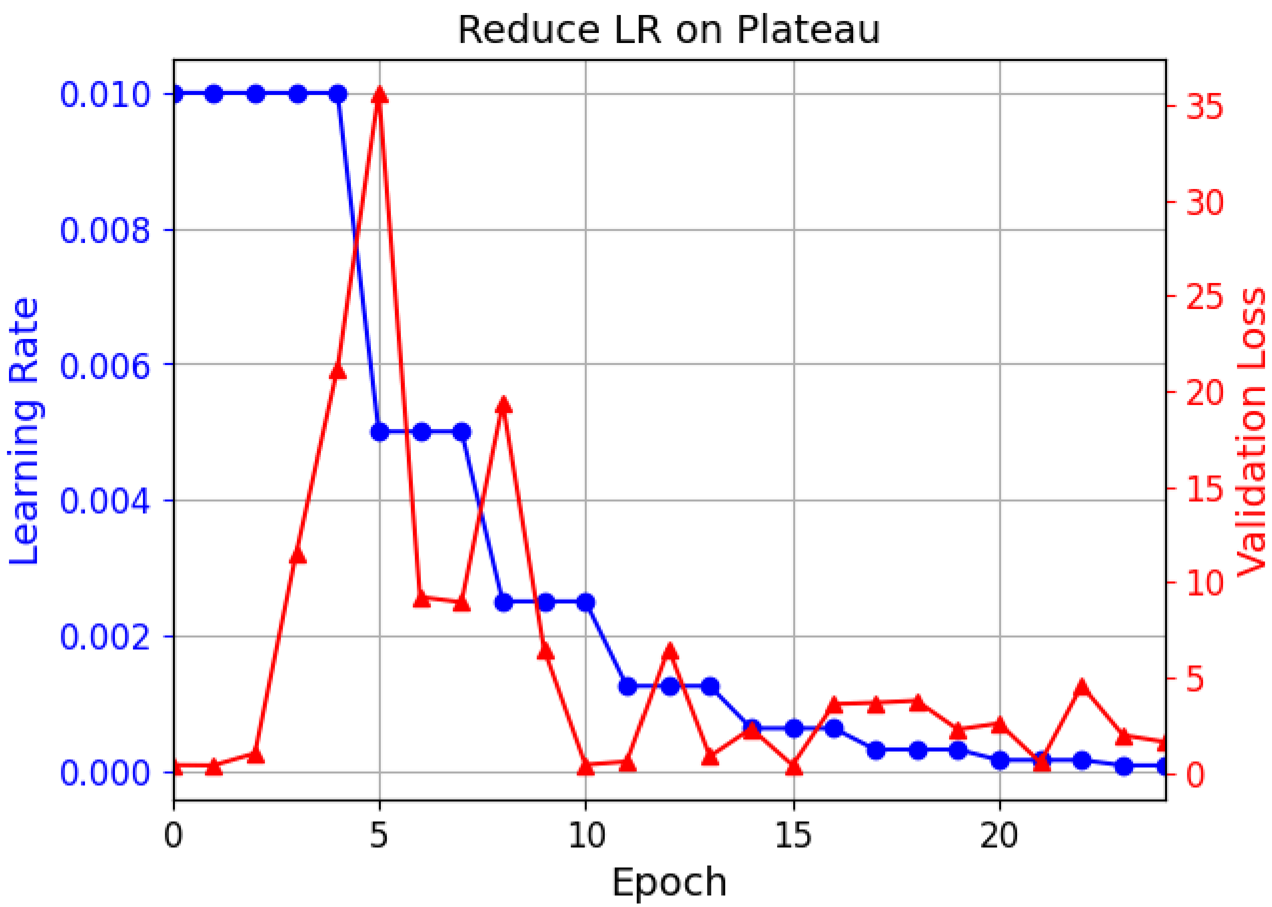
In [69]:
```python
print("train loss:", test_model(model, train_loader, nn.CrossEntropyLoss())[0])
print("test loss:", test_model(model, test_loader, nn.CrossEntropyLoss())[0])
```

```
Test Loss: 1.0150, Test Metric: 0.0000
train loss: 1.0150460173874054
Test Loss: 0.9885, Test Metric: 0.0000
test loss: 0.9885464462030465
```

In [70]:
```python
n_epochs = 25
plt.plot(history['epoch'], history["learning_rate"], "bo-")
plt.xlabel("Epoch")
plt.ylabel("Learning Rate", color='b')
plt.tick_params('y', colors='b')
plt.gca().set_xlim(0, n_epochs - 1)
plt.grid(True)

ax2 = plt.gca().twinx()
ax2.plot(history['epoch'], history["val_loss"], "r^-")
ax2.set_ylabel('Validation Loss', color='r')
ax2.tick_params('y', colors='r')
```

```
plt.title("Reduce LR on Plateau", fontsize=14)
plt.show()
```



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

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

Task 6 c) answer:

Initial LR stays constant for a few epochs before reducing. First drop (Epoch 6) occurs as validation loss plateaus. Further reductions (Epochs 9, 11, 14) indicate continued stagnation in loss improvement. Final LR is very low (0.0001), suggesting fine-tuning with minor updates

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

Task 7 Avoiding Overfitting Through Regularization

A **dropout layer** essentially **takes all inputs** passed to it and randomly **sets some inputs to 0** with a certain rate. One way that models can **overfit** to training data is by essentially *"memorizing"* or encoding into its function some **properties that are specific to one data set**.

We can help mitigate this by ensuring our data is as **non-homogeneous within each sample** (train, val, test) and as **homogeneous across our samples as possible**. This isn't always enough as we often end up showing our model the same training data multiple times and **it may learn patterns about the training data that don't exist in other data**.

By adding dropout, we **decrease the likelihood that it learns** these sorts of **inter-sample patterns by adding random variations** to either the data or the way it handles the same data each time.

Our models above all overfit (why?). Let's now tackle this problem using dropout.

Task 7:
a) Copy the code for the model of Task 6, add a dropout (20% rate) before each hidden layer (https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html)
b) Compare the results.

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

In [73]:
```
model = nn.Sequential(
    nn.Flatten(),
    nn.BatchNorm1d(28*28),
    nn.Linear(28*28, 300),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.BatchNorm1d(300),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(100, 10)
)
```

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.NAdam(model.parameters(),lr=0.01)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,factor=0.5, patience=2)
```

In [74]:
```
history, model = train_and_validate(train_loader, valid_loader, model, optimizer = optimizer, criterion=criterion, num_epochs =
```

```
Epoch [1/25], Train Loss: 0.5331, Train Metric: 0.8088, Val Loss: 0.3835, Val Metric: 0.8669
Epoch [2/25], Train Loss: 0.4387, Train Metric: 0.8430, Val Loss: 0.4103, Val Metric: 0.8515
Epoch [3/25], Train Loss: 0.4178, Train Metric: 0.8492, Val Loss: 0.5961, Val Metric: 0.8706
Epoch [4/25], Train Loss: 0.4057, Train Metric: 0.8543, Val Loss: 4.9144, Val Metric: 0.8752
Epoch [5/25], Train Loss: 0.3552, Train Metric: 0.8711, Val Loss: 2.6200, Val Metric: 0.8888
Epoch [6/25], Train Loss: 0.3390, Train Metric: 0.8751, Val Loss: 1.7430, Val Metric: 0.8827
Epoch [7/25], Train Loss: 0.3285, Train Metric: 0.8785, Val Loss: 1.7010, Val Metric: 0.8886
Epoch [8/25], Train Loss: 0.2975, Train Metric: 0.8890, Val Loss: 0.8783, Val Metric: 0.8920
Epoch [9/25], Train Loss: 0.2839, Train Metric: 0.8935, Val Loss: 1.9031, Val Metric: 0.8966
Epoch [10/25], Train Loss: 0.2816, Train Metric: 0.8939, Val Loss: 3.1526, Val Metric: 0.8973
Epoch [11/25], Train Loss: 0.2611, Train Metric: 0.9015, Val Loss: 1.5318, Val Metric: 0.9005
Epoch [12/25], Train Loss: 0.2560, Train Metric: 0.9029, Val Loss: 0.4791, Val Metric: 0.9015
Epoch [13/25], Train Loss: 0.2532, Train Metric: 0.9034, Val Loss: 0.7447, Val Metric: 0.9015
Epoch [14/25], Train Loss: 0.2400, Train Metric: 0.9081, Val Loss: 2.5850, Val Metric: 0.9005
Epoch [15/25], Train Loss: 0.2369, Train Metric: 0.9090, Val Loss: 4.0188, Val Metric: 0.9023
Epoch [16/25], Train Loss: 0.2340, Train Metric: 0.9108, Val Loss: 1.0806, Val Metric: 0.9009
Epoch [17/25], Train Loss: 0.2279, Train Metric: 0.9126, Val Loss: 2.9399, Val Metric: 0.9057
Epoch [18/25], Train Loss: 0.2287, Train Metric: 0.9134, Val Loss: 1.9076, Val Metric: 0.9021
Epoch [19/25], Train Loss: 0.2249, Train Metric: 0.9126, Val Loss: 2.4155, Val Metric: 0.9037
Epoch [20/25], Train Loss: 0.2228, Train Metric: 0.9139, Val Loss: 0.6456, Val Metric: 0.9068
Epoch [21/25], Train Loss: 0.2229, Train Metric: 0.9139, Val Loss: 2.8921, Val Metric: 0.9053
Epoch [22/25], Train Loss: 0.2184, Train Metric: 0.9154, Val Loss: 0.9122, Val Metric: 0.9033
Epoch [23/25], Train Loss: 0.2173, Train Metric: 0.9171, Val Loss: 0.2771, Val Metric: 0.9045
Epoch [24/25], Train Loss: 0.2187, Train Metric: 0.9153, Val Loss: 0.3924, Val Metric: 0.9037
Epoch [25/25], Train Loss: 0.2191, Train Metric: 0.9151, Val Loss: 1.4089, Val Metric: 0.9041
```

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

In [ ]:
```
print("train loss:", test_model(model, train_loader, nn.CrossEntropyLoss())[0])
print("test loss:", test_model(model, test_loader, nn.CrossEntropyLoss())[0])
```

In [ ]:
```
plt.plot(history['epoch'], history["learning_rate"], "bo-")
plt.xlabel("Epoch")
plt.ylabel("Learning Rate", color='b')
plt.tick_params('y', colors='b')
plt.gca().set_xlim(0, n_epochs - 1)
plt.grid(True)

ax2 = plt.gca().twinx()
```

```
ax2.plot(history['epoch'], history["val_loss"], "r^-")
ax2.set_ylabel('Validation Loss', color='r')
ax2.tick_params('y', colors='r')

plt.title("Reduce LR on Plateau", fontsize=14)
plt.show()
```

Optional Exercise (Bonus points): Using Callbacks during Training

Task 8: Add two of the following features to your model training loop:
a) Keep track of the model with the best validation loss and return the best model at the end of the traininig loop instead of the last model.
b) Stop model training after 5 epochs of loss not impoving
c) Add Tensorboard logging for one or more quantities from your model training history

You can use the code snippets below.

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

In [75]:
```
from torch.utils.tensorboard import SummaryWriter
```

In [76]:
```
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None, scheduler=None):
    writer = SummaryWriter()
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
        'val_loss': [],
        'val_metric': [],
        'learning_rate': []
    }  # Initialize a dictionary to store epoch-wise results

    best_val_loss = float('inf')
    best_model_state = None
    patience_counter = 0
    patience = 5
    with torch.no_grad():
        proper_dtype = torch.int64
        X,y = next(iter(train_loader))
        try:
```

```
            loss = criterion(model(X), y.to(proper_dtype))
        except:
            try:
                proper_dtype = torch.float32
                loss = criterion(model(X), y.to(proper_dtype))
            except:
                print("No valid data-type could be found")

    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:
            y = y.to(proper_dtype)
            optimizer.zero_grad()  # Clear existing gradients
            outputs = model(X)  # Make predictions
            loss = criterion(outputs, y)  # Compute the loss
            loss.backward()  # Compute gradients
            optimizer.step()  # Update model parameters

            epoch_loss += loss.item()

            # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
            if metric is not None:
                epoch_metric += metric(outputs, y)
            else:
                epoch_metric += 0.0

        # 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:
                y_val = y_val.to(proper_dtype)
                outputs_val = model(X_val)  # Make predictions
                val_loss += criterion(outputs_val, y_val).item()  # Compute loss
                if metric is not None:
                    val_metric += metric(outputs_val, y_val)
                else:
```

```python
                    val_metric += 0.0

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

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict()
            patience_counter = 0
        else:
            patience_counter += 1

        # Early stopping check
        if patience_counter >= patience:
            print(f"Early stopping at epoch {epoch+1} as validation loss did not improve for {patience} epochs.")
            break

        # 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)
        history['learning_rate'].append(optimizer.param_groups[0]['lr'])
        # 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)
        history['learning_rate'].append(optimizer.param_groups[0]['lr'])

        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}')

        writer.add_scalar("Loss/Train", epoch_loss, epoch)
        writer.add_scalar("Loss/Validation", val_loss, epoch)
        writer.add_scalar("Metric/Train", epoch_metric, epoch)
        writer.add_scalar("Metric/Validation", val_metric, epoch)
        writer.add_scalar("Learning Rate", optimizer.param_groups[0]["lr"], epoch)


        if scheduler is not None:
            # MODIFY THIS LINE TO WORK PROPERLY FOR REDUCELRONPLATEAU
            scheduler.step(val_loss)# This will crash if you don't fix it
```

```python
    writer.flush()
    writer.close()

    # Return the best model
    if best_model_state is not None:
        model.load_state_dict(best_model_state)

    return history, model
```

In [77]:
```python
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.5, patience=2)

def accuracy_metric(pred, target):
    if len(pred.shape) == 1:
        accuracy = torch.sum(torch.eq(pred > 0.5, target)).item() / len(pred)
    else:
        pred = pred.argmax(dim=1)
        accuracy = torch.sum(pred == target).item() / len(pred)
    return accuracy
```

In [78]:
```python
history, model = train_and_validate(train_loader, valid_loader, model,
                                    optimizer=optimizer, criterion=criterion,
                                    num_epochs=50, metric=accuracy_metric)
```

```
Epoch [1/50], Train Loss: 0.3759, Train Metric: 0.8658, Val Loss: 0.3819, Val Metric: 0.8780
Epoch [2/50], Train Loss: 0.3729, Train Metric: 0.8668, Val Loss: 0.3171, Val Metric: 0.8875
Epoch [3/50], Train Loss: 0.3702, Train Metric: 0.8693, Val Loss: 7.1846, Val Metric: 0.8784
Epoch [4/50], Train Loss: 0.3640, Train Metric: 0.8693, Val Loss: 0.5561, Val Metric: 0.8681
Epoch [5/50], Train Loss: 0.3759, Train Metric: 0.8662, Val Loss: 27.9485, Val Metric: 0.8679
Epoch [6/50], Train Loss: 0.3863, Train Metric: 0.8627, Val Loss: 6.1669, Val Metric: 0.8851
Early stopping at epoch 7 as validation loss did not improve for 5 epochs.
```

In [79]:
```python
%load_ext tensorboard
%tensorboard --logdir=./runs --port=6006
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

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