a02-notebook

May 8, 2022

[ ]:
```
# Student: Timur Carstensen
# Student ID: 1722194
```

[ ]:
```
#
# IE 678 Deep Learning, University of Mannheim
# Author: Rainer Gemulla
```

[ ]:
```
import torch
import torch.nn as nn
import torchvision
import matplotlib
import matplotlib.pyplot as plt
import os
import numpy as np
from sklearn.metrics import confusion_matrix

from IPython import get_ipython
from helper import *
from util import nextplot

%matplotlib inline
```

[ ]:
```
# Use GPU if CUDA is available
DATA_PATH = "data/"
MODEL_PATH = "data/"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
```

[ ]:
```
# To prevent from retraining models, you can save them to disk and load them
# later on
def save(model, filename):
    torch.save(model.state_dict(), os.path.join(MODEL_PATH, filename))


def load(model, filename):
    model.load_state_dict(torch.load(os.path.join(MODEL_PATH, filename)))
    return model
```

1 1 Convolutional Neural Networks

1.1 Load the data

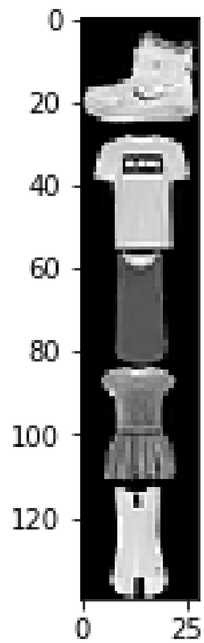
[ ]:
```
X, y, Xtest, ytest = load_dataset("fashionmnist")
class_dict = {
    0: "t-shirt/top",
    1: "trouser",
    2: "pullover",
    3: "dress",
    4: "coat",
    5: "sandal",
    6: "shirt",
    7: "sneaker",
    8: "bag",
    9: "ankle boot",
}
print(f"{len(X)} training examples")
print(f"{len(Xtest)} test examples")
```

```
60000 training examples
10000 test examples
```

[ ]:
```
def show_image(x):
    "Show one (or multiple) 28x28 MNIST images as a gray-scale image."
    plt.imshow(x.reshape(-1, 28).cpu(), cmap="gray", interpolation="none")


# Plot first 5 training examples. Each example consists of a 1x28x28 tensor␣
 ↪with values
# in [0,1] and a label
nextplot()
show_image(X[:5])
for i in range(5):
    print(f"Label of example i={i}: {class_dict[y[i].item()]}")
```

```
Label of example i=0: ankle boot
Label of example i=1: t-shirt/top
Label of example i=2: t-shirt/top
Label of example i=3: dress
Label of example i=4: t-shirt/top
```



1.2 1a+b Implement a CNN model

[ ]:
```
# Here is a PyTorch version of logistic regression.
class LogisticRegression(nn.Module):
    def __init__(self, num_features):
        super().__init__()
        self.linear = nn.Linear(num_features, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.linear(x.float())
        out = self.sigmoid(out)
        return out
```

[ ]:
```
# Implement a simple CNN
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # Create the required layers/function and store them as instance␣
↪variables
        # YOUR CODE HERE

        self.layer1 = nn.Sequential(
            nn.Conv2d(kernel_size=3, in_channels=1, out_channels=32, stride=1,␣
↪padding=1, padding_mode="zeros"),
```

```
            nn.Sigmoid(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc1 = nn.Linear(in_features=6272, out_features=10)
        self.log_softmax = nn.LogSoftmax()


    def forward(self, x):
        """
        Perform the forward pass.

        Parameters
        ----------
        x: tensor of shape (batch_size, 1, 28, 28)

        Returns
        -------
        model output as a tensor of shape (batch_size, 10)
        """
        out = None
        # Use the layers/functions created above to compute model output
        # YOUR CODE HERE
        out = self.layer1(x)
        out = torch.flatten(out, 1)
        out = self.fc1(out)
        out = self.log_softmax(out)

        return out
```

[ ]:
```
# here is a description of what you created (this uses the member variables)
print(SimpleCNN())

# SimpleCNN(
#   (layer1): Sequential(
#     (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
#     (1): Sigmoid()
#     (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,␣
 ↪ceil_mode=False)
#   )
#   (fc1): Linear(in_features=6272, out_features=10, bias=True)
#   (log_softmax): LogSoftmax()
# )
```

```
SimpleCNN(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Sigmoid()
```

```
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (fc1): Linear(in_features=6272, out_features=10, bias=True)
  (log_softmax): LogSoftmax(dim=None)
)
```

[ ]:
```
# One way to see the parameters of your model is to look at its "state". Check␣
 ↪that the
# shapes of the parameters that you see here match your computations of task␣
 ↪1a).
model = SimpleCNN().to(DEVICE)
model.state_dict()
```

[ ]:
```
OrderedDict([('layer1.0.weight',
              tensor([[[[-2.4413e-01,  4.2617e-02,  3.1756e-01],
                        [-7.0482e-02,  1.7290e-02, -1.0355e-01],
                        [ 3.1558e-02, -1.2980e-01,  3.2857e-01]]],


                      [[[ 6.5351e-03,  9.9397e-02, -2.2767e-01],
                        [ 2.9914e-01,  1.1743e-01,  4.4040e-02],
                        [-2.8349e-01, -6.3990e-02,  2.0376e-01]]],


                      [[[-1.0722e-01,  2.3458e-02,  5.1296e-02],
                        [-6.3506e-03,  1.2136e-01, -1.3855e-01],
                        [ 2.7447e-01,  4.4186e-02, -3.2704e-02]]],


                      [[[-1.3704e-01, -2.0820e-01,  3.0723e-01],
                        [-2.5863e-01,  2.2444e-01, -1.2951e-01],
                        [ 1.4762e-01, -5.5286e-02,  1.5073e-01]]],


                      [[[-3.0081e-01, -1.6919e-01,  9.5023e-02],
                        [ 6.2922e-03, -2.3756e-01, -2.3372e-01],
                        [-2.3056e-01,  1.1161e-02, -5.2626e-02]]],


                      [[[-2.3484e-01,  1.2478e-02,  3.3158e-01],
                        [ 1.5155e-01,  2.0131e-01, -3.0494e-01],
                        [-1.5692e-01, -2.5777e-01, -7.3264e-02]]],


                      [[[-1.1912e-01,  1.9668e-01, -2.6741e-01],
                        [-2.2863e-02, -1.7096e-01,  6.7788e-02],
```

```
[ 6.1866e-02, -1.2586e-01, -1.2740e-01]]],


[[[-2.7602e-02, -1.3564e-01, -2.1127e-01],
  [ 2.1656e-01,  1.4129e-01, -2.4218e-01],
  [-1.9584e-01, -2.3433e-01, -1.7348e-01]]],


[[[-2.3539e-01, -1.3138e-01, -3.2553e-01],
  [-1.1223e-01, -2.0720e-01,  2.9110e-01],
  [ 3.9437e-02,  8.4036e-03, -4.0171e-02]]],


[[[-8.4452e-02,  2.9662e-01,  2.5323e-01],
  [ 2.7753e-01,  2.4395e-01, -1.2840e-01],
  [ 2.9421e-01, -1.8414e-01, -3.1143e-01]]],


[[[-2.2773e-01, -2.5044e-01, -1.3125e-01],
  [-7.8448e-02,  2.9281e-01,  1.4514e-01],
  [ 7.3863e-02, -2.6794e-01,  1.6106e-03]]],


[[[-2.8990e-01,  1.1630e-01,  5.3863e-02],
  [-3.3235e-02, -9.5065e-03, -3.1731e-01],
  [-8.7020e-02,  1.1119e-01, -1.8182e-02]]],


[[[-3.2994e-01, -1.9481e-01,  3.1867e-01],
  [-3.4428e-02, -1.4467e-01,  6.3393e-04],
  [ 5.3119e-02,  2.4644e-01, -2.4222e-01]]],


[[[-2.7880e-01, -1.1047e-01,  1.6090e-01],
  [-2.5582e-01, -9.9603e-02, -3.1559e-01],
  [-6.4307e-02,  2.6226e-01, -3.1198e-01]]],


[[[ 6.3477e-02, -2.0518e-01,  1.4477e-01],
  [-6.0590e-02,  2.6558e-01,  2.7426e-01],
  [-2.9511e-01,  3.3769e-02, -3.1349e-01]]],


[[[ 2.0599e-02,  2.1773e-01,  2.1587e-01],
  [ 3.1572e-01, -2.2507e-01,  6.9792e-02],
  [ 1.0981e-01, -1.7750e-01,  2.5679e-01]]],
```

```
[[[-2.2161e-01,  2.0771e-01,  1.7525e-01],
  [-3.5824e-02, -1.1653e-01, -2.8805e-01],
  [-9.0304e-02, -2.3229e-01,  1.6389e-01]]],


[[[ 2.8958e-01, -1.5212e-01, -2.3699e-01],
  [ 2.6953e-01,  1.2830e-01, -3.9619e-02],
  [ 3.1547e-01,  2.9752e-01, -1.5085e-01]]],


[[[ 1.0693e-01,  7.6142e-02, -1.3993e-01],
  [ 1.6542e-01, -1.8272e-01,  3.0806e-01],
  [ 1.4720e-01, -2.1153e-01, -2.9563e-03]]],


[[[-1.7642e-01,  9.0950e-02,  1.2180e-01],
  [ 2.8606e-01, -3.1954e-01, -2.7676e-01],
  [ 1.3660e-01,  1.0776e-01, -1.5727e-03]]],


[[[ 1.6720e-01, -2.6336e-04, -2.5182e-01],
  [ 1.7061e-01,  3.0210e-01,  8.9676e-02],
  [ 1.7778e-01,  3.0081e-01, -2.5432e-01]]],


[[[-1.4334e-01,  1.7685e-01,  5.0942e-02],
  [-7.5547e-03, -1.5844e-01, -2.2489e-01],
  [ 1.6404e-01,  1.5218e-01, -2.7796e-01]]],


[[[ 3.0824e-01, -2.1534e-02,  1.5150e-01],
  [-3.0472e-01, -1.1949e-02, -2.8825e-01],
  [-2.6855e-01, -1.9724e-01, -1.4083e-01]]],


[[[ 1.3439e-01, -3.7127e-02,  1.3735e-01],
  [ 5.0808e-02, -2.3116e-01, -3.1062e-01],
  [-9.1478e-02, -1.1906e-01,  2.5787e-01]]],


[[[-1.2887e-01,  1.8439e-01, -3.1071e-01],
  [-3.3853e-03,  8.4458e-02,  9.3417e-03],
  [-1.5631e-01, -2.5000e-01, -6.6180e-03]]],


[[[ 2.3939e-01,  2.9764e-02,  1.7813e-01],
```

```
                      [-8.3289e-02,  2.5261e-01,  2.3511e-01],
                      [ 1.1759e-01,  2.8765e-01,  1.3256e-01]]],


                    [[[ 2.3241e-02,  7.4002e-02,  9.9316e-02],
                      [ 1.6059e-01,  2.3090e-02,  1.1511e-01],
                      [-2.6682e-01,  2.3910e-01, -2.2539e-01]]],


                    [[[ 1.0598e-01,  2.5343e-01, -1.8034e-01],
                      [-2.1721e-01,  2.0421e-01, -3.9024e-02],
                      [ 1.2270e-01, -3.7205e-04,  1.2685e-02]]],


                    [[[ 7.0892e-02,  2.6390e-01, -2.6142e-01],
                      [ 1.5967e-01, -1.8723e-02, -3.2449e-01],
                      [ 1.6712e-01, -1.9518e-01, -5.5869e-02]]],


                    [[[ 1.2532e-03, -1.7499e-01, -1.0941e-01],
                      [ 2.9381e-01, -1.0250e-01,  2.7956e-01],
                      [-5.1640e-02,  2.9207e-02,  6.1875e-02]]],


                    [[[ 2.4452e-01,  3.0876e-01, -4.8092e-03],
                      [-1.9571e-01, -2.7417e-01, -1.7358e-01],
                      [ 9.1399e-02,  1.1284e-01, -7.6799e-03]]],


                    [[[ 2.2495e-01,  3.0138e-01, -7.7089e-02],
                      [ 3.4538e-03,  1.6920e-01, -8.8056e-02],
                      [ 2.3367e-01,  4.6525e-02,  2.5716e-01]]]])),
             ('layer1.0.bias',
              tensor([ 0.1437, -0.0331,  0.0052, -0.2324,  0.2480, -0.0151,
-0.0358,  0.2478,
                      -0.1060, -0.1052,  0.3299, -0.2489,  0.1990, -0.2528,
0.0044,  0.2342,
                       0.0279, -0.1444, -0.0194, -0.0461, -0.2956,  0.1159,
-0.0909, -0.0087,
                      -0.0306, -0.1698, -0.2644,  0.0666, -0.1278,  0.1081,
-0.0649,  0.3312])),
             ('fc1.weight',
              tensor([[ 0.0030, -0.0038, -0.0019,  …, -0.0062,  0.0048,
-0.0011],
                      [ 0.0067,  0.0064, -0.0086,  …, -0.0016,  0.0011,
0.0124],
                      [-0.0015,  0.0104, -0.0052,  …, -0.0099,  0.0007,
```

```
-0.0018],
                        …,
                        [-0.0110,  0.0022, -0.0017,  …, -0.0086, -0.0051,
-0.0027],
                        [ 0.0094, -0.0006,  0.0122,  …,  0.0076,  0.0049,
0.0052],
                        [-0.0088, -0.0055, -0.0002,  …, -0.0109,  0.0005,
-0.0030]])),
             ('fc1.bias',
              tensor([ 0.0035, -0.0126, -0.0096,  0.0048,  0.0121,  0.0049,
-0.0088,  0.0044,
                        -0.0051,  0.0046]))])
```

[ ]:
```
# Run the forward pass on the first 5 examples.
with torch.no_grad():  # tell torch to not compute backward graph
    print(model(X[:5]))
```

```
tensor([[-2.0608, -2.0418, -2.5162, -2.9079, -2.1383, -2.2506, -1.8348, -2.6992,
         -2.2903, -2.8794],
        [-2.2168, -2.3191, -2.3916, -2.8283, -2.5097, -2.3808, -1.7170, -2.3541,
         -2.0668, -2.7011],
        [-2.1600, -2.2941, -2.4255, -2.9933, -2.4440, -2.3244, -1.7916, -2.4609,
         -2.0707, -2.4955],
        [-2.0819, -2.3726, -2.3657, -2.8566, -2.3902, -2.4340, -1.7798, -2.3896,
         -2.2686, -2.4349],
        [-2.2163, -2.3349, -2.3170, -3.0505, -2.3059, -2.3469, -1.7984, -2.3935,
         -2.0606, -2.6784]])
```

```
/var/folders/hq/pyn2nf8x3f94nbk8r749084r0000gn/T/ipykernel_72444/260300387.py:35
: UserWarning: Implicit dimension choice for log_softmax has been deprecated.
Change the call to include dim=X as an argument.
  out = self.log_softmax(out)
```

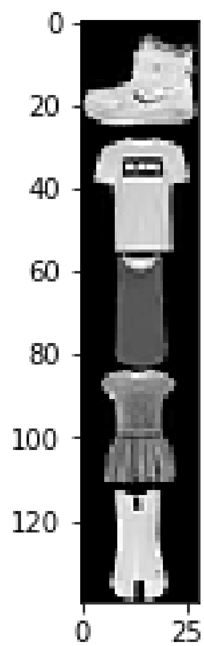
[ ]:
```
# Plot training (or test) examples, their correct labels, and the most likely␣
↪model
# predictions. Do not be worried if your (untrained) model always seems to␣
↪predict the
# same class.
@torch.no_grad()
def mnist_predict(model, start, end=None, use_test_data=False):
    if end is None:
        end = start + 1
    if use_test_data:
        images = Xtest[start:end].to(DEVICE)
        labels = ytest[start:end].to(DEVICE)
    else:
        images = X[start:end].to(DEVICE)
```

```python
        labels = y[start:end].to(DEVICE)
    nextplot()
    show_image(images)
    print("     Labels:", [class_dict[label.item()] for label in labels])
    out = model(images)
    _, yhat = torch.max(out, 1)
    print("Predictions:", [class_dict[pred.item()] for pred in yhat])


# first 5 examples from training + predictions
mnist_predict(model, 0, 5)
```

```
     Labels: ['ankle boot', 't-shirt/top', 't-shirt/top', 'dress',
't-shirt/top']
Predictions: ['shirt', 'shirt', 'shirt', 'shirt', 'shirt']
```

```
/var/folders/hq/pyn2nf8x3f94nbk8r749084r0000gn/T/ipykernel_72444/260300387.py:35
: UserWarning: Implicit dimension choice for log_softmax has been deprecated.
Change the call to include dim=X as an argument.
  out = self.log_softmax(out)
```



1.3 1c Evaluate model performance

[ ]:
```python
# test model
@torch.no_grad()
def mnist_test(model, batch_size=100, reshape_1d=False):
    """
    Function to test your CNN on test data

    Parameters
    ----------

    model: trained CNN from task 1a
    batch_size: size of batch for dataloader
    reshape_1d: Reshape images to a 1d vectors (allows use of models other than␣
↪CNNs
                such as fully-connected FNNs)

    Returns
    -------
    accuracy of input model
    """
    correct = 0  # number of correct predictions
    total = 0  # total number of examples
    model.eval()  # set layers like dropout and batch norm to eval mode

    # Create test data loader
    if reshape_1d:
        dataset = torch.utils.data.TensorDataset(Xtest.reshape(len(Xtest), -1),␣
↪ytest)
    else:
        dataset = torch.utils.data.TensorDataset(Xtest, ytest)
    test_loader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size, shuffle=False
    )
    # confusion_matrix = np.zeros(shape=(10,10))
    pred = list()
    y = list()
    # Loop over data
    for batch in test_loader:
        # YOUR CODE HERE
        # Update correct and total using the examples in the batch. To␣
↪understand what a
        # DataLoader does, have a look at the contents of "batch" before you␣
↪start.
        images, labels = batch
        # calculate outputs by running images through the network
        outputs = model(images)
```

```
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        pred.append(predicted)
        y.append(labels)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        # confusion matrix over classes

    M = confusion_matrix(torch.cat(y), torch.cat(pred))
    accuracy = (correct / total) * 100
    print(f"Accuracy on {total} test images: {accuracy:.2f} %")
    return accuracy, M
```

[ ]:
```
# Test your code. Output should be: 6.97%
class DummyModel(nn.Module):
    def forward(self, x):
        return x.reshape(len(x), -1)[:, 200:210] * torch.arange(10).to(DEVICE)


mnist_test(DummyModel().to(DEVICE))[0]
```

```
Accuracy on 10000 test images: 6.97 %
```

[ ]: 6.97

1.4 1d Train a model

[ ]:
```
# train model
def mnist_train(
    model, num_epochs=5, learning_rate=0.001, batch_size=100, reshape_1d=False
):
    """
    Function to train the provided CNN network.

    Parameters
    ----------
    model: the model to train
    num_epochs: number of epochs to train
    learning_rate: learning rate to use
    batch_size: size of batch for data loader
    reshape_1d: Reshape images to a 1d vectors (allows use of models other than␣
↪CNNs
                such as fully-connected FNNs)
    """
    # YOUR CODE HERE

    if reshape_1d:
```

```
        dataset = torch.utils.data.TensorDataset(X.reshape(len(X), -1), y)
    else:
        dataset = torch.utils.data.TensorDataset(X, y)
    test_loader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size, shuffle=False
    )

    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    criterion = nn.NLLLoss()
    i = 0

    for epoch in range(num_epochs):  # loop over the dataset multiple times

        running_loss = 0.0

        for batch in test_loader:
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = batch

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            i += 1
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.
↪3f}')
                running_loss = 0.0


    return model
```

1.5 1e Train and evaluate the simple CNN model

[ ]:
```
# load model
load(model, "simple_cnn.pt")
```

[ ]:
```
SimpleCNN(
  (layer1): Sequential(
```

```
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Sigmoid()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (fc1): Linear(in_features=6272, out_features=10, bias=True)
  (log_softmax): LogSoftmax(dim=None)
)
```

[ ]:
```
# Train a model.
model = mnist_train(SimpleCNN().to(DEVICE))
```

```
/var/folders/hq/pyn2nf8x3f94nbk8r749084r0000gn/T/ipykernel_72444/260300387.py:35
: UserWarning: Implicit dimension choice for log_softmax has been deprecated.
Change the call to include dim=X as an argument.
  out = self.log_softmax(out)
```

```
[4,  2000] loss: 0.027
```

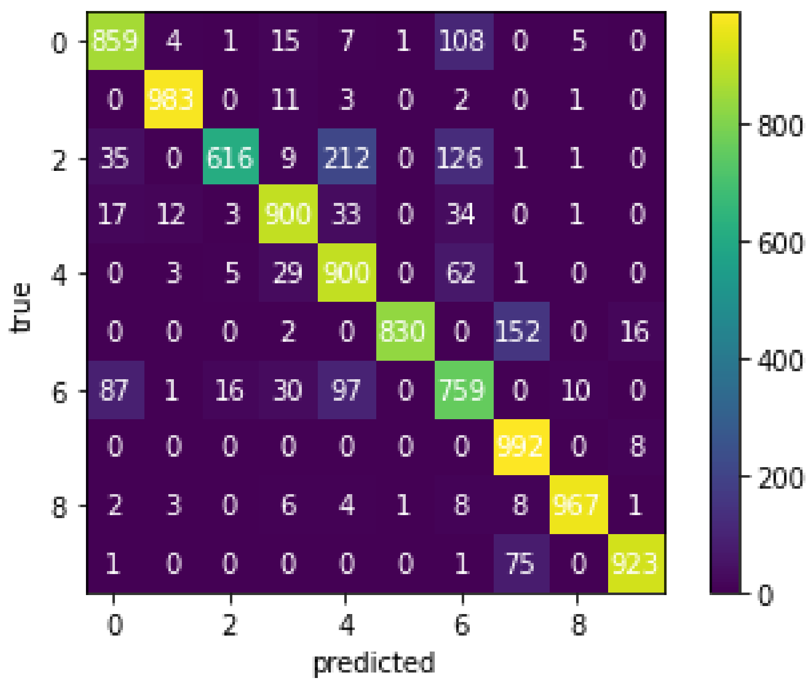
[ ]:
```
# persisting the model
save(model, "simple_cnn.pt")
```

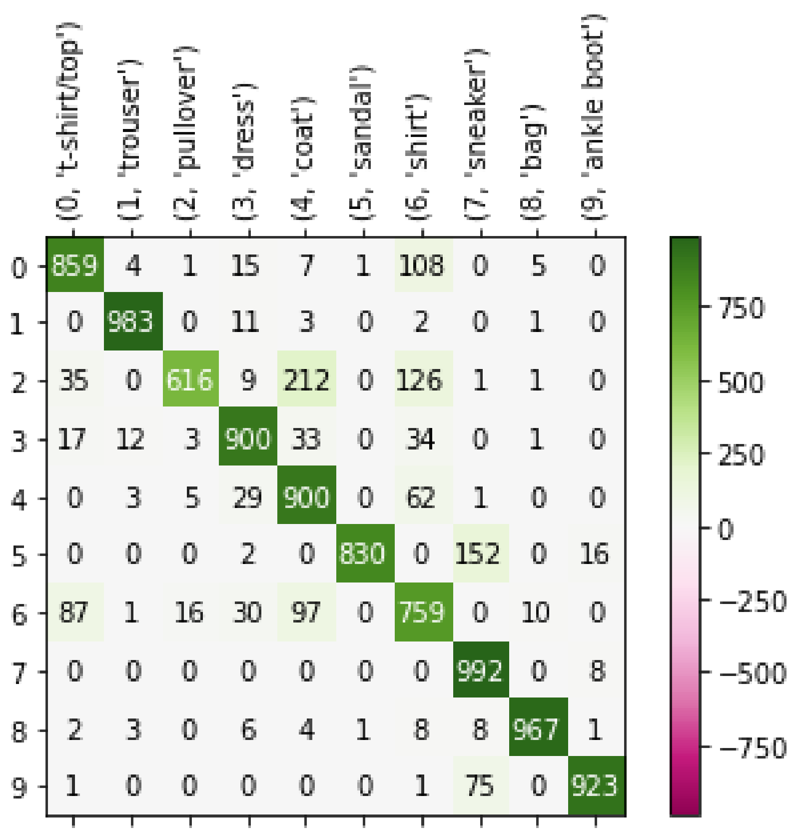
[ ]:
```
# Test a model. The simple CNN should perform much better after training.
# model = load(SimpleCNN(), "simple_cnn.pt").to(DEVICE)
_, M = mnist_test(model)
plt.imshow(M, origin="upper")
for ij, v in np.ndenumerate(M):
    i, j = ij
    plt.text(j, i, str(v), color="white", ha="center", va="center")
plt.xlabel("predicted")
plt.ylabel("true")
plt.colorbar()
plt.savefig("tex/assets/conf_mat.png", dpi=1000)
acc = M.diagonal()/M.sum(axis=1)
from util import plot_matrix

out = {v:acc[i] for i, (k, v) in enumerate(class_dict.items())}
plot_matrix(M, colnames=class_dict.items())
```

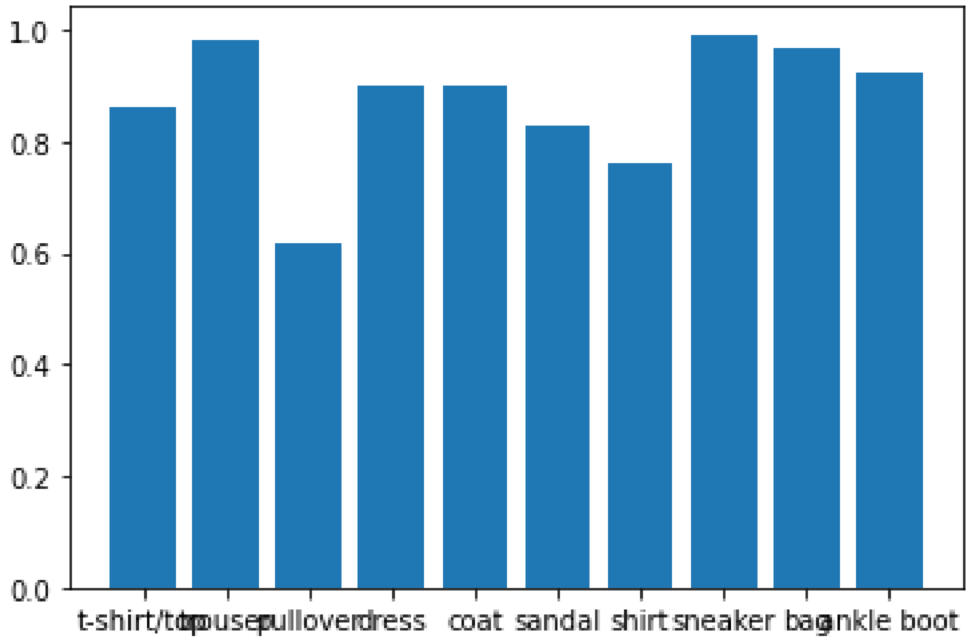
```
/var/folders/hq/pyn2nf8x3f94nbk8r749084r0000gn/T/ipykernel_72444/260300387.py:35
: UserWarning: Implicit dimension choice for log_softmax has been deprecated.
Change the call to include dim=X as an argument.
  out = self.log_softmax(out)
```

```
Accuracy on 10000 test images: 87.29 %
```


true
predicted
0
2
4
6
8
859 4 1 15 7 1 108 0 5 0
0 983 0 11 3 0 2 0 1 0
35 0 616 9 212 0 126 1 1 0
17 12 3 900 33 0 34 0 1 0
0 3 5 29 900 0 62 1 0 0
0 0 0 2 0 830 0 152 0 16
87 1 16 30 97 0 759 0 10 0
0 0 0 0 0 0 0 992 0 8
2 3 0 6 4 1 8 8 967 1
1 0 0 0 0 0 1 75 0 923
800
600
400
200
0


(0, 't-shirt/top')
(1, 'trouser')
(2, 'pullover')
(3, 'dress')
(4, 'coat')
(5, 'sandal')
(6, 'shirt')
(7, 'sneaker')
(8, 'bag')
(9, 'ankle boot')
0
1
2
3
4
5
6
7
8
9
859 4 1 15 7 1 108 0 5 0
0 983 0 11 3 0 2 0 1 0
35 0 616 9 212 0 126 1 1 0
17 12 3 900 33 0 34 0 1 0
0 3 5 29 900 0 62 1 0 0
0 0 0 2 0 830 0 152 0 16
87 1 16 30 97 0 759 0 10 0
0 0 0 0 0 0 0 992 0 8
2 3 0 6 4 1 8 8 967 1
1 0 0 0 0 0 1 75 0 923
750
500
250
0
−250
−500
−750

[ ]:
```
nextplot()
plt.bar(range(len(out)), list(out.values()), align='center')
plt.xticks(range(len(out)), list(out.keys()))
plt.show()
plt.savefig("tex/assets/accuracies_per_class.png", dpi=200)
```




```
&lt;Figure size 432x288 with 0 Axes&gt;
```

1.6 1f Sandbox

[ ]:
```
# try adding more layers, regularization, etc.
class MyCnn(nn.Module):
    def __init__(self):
        super(MyCnn, self).__init__()
        # YOUR CODE HERE

    def forward(self, x):
        out = None
        # YOUR CODE HERE

        return out
```

[ ]:
```
my_cnn = MyCnn().to(DEVICE)
```

[ ]:
```
# train
mnist_train(my_cnn)
# save(my_cnn, "my_cnn.pt")
```

[ ]:
```
# test
# my_cnn = load(my_cnn, "my_cnn.pt")
mnist_test(my_cnn)
```

2 2 Recurrent Neural Networks and Pretraining

2.1 Load and preprocess the data

[ ]:
```
# Load review and label data
with open(os.path.join(DATA_PATH, "reviews_small.txt")) as f:
    reviews_lines = f.readlines()

with open(os.path.join(DATA_PATH, "labels_small.txt")) as f:
    label_lines = f.readlines()

print(reviews_lines[0])
print(label_lines[0])
```

```
bromwell high is a cartoon comedy . it ran at the same time as some other
programs about school life  such as  teachers  . my   years in the teaching
profession lead me to believe that bromwell high  s satire is much closer to
reality than is  teachers  . the scramble to survive financially  the insightful
students who can see right through their pathetic teachers  pomp  the pettiness
of the whole situation  all remind me of the schools i knew and their students .
when i saw the episode in which a student repeatedly tried to burn down the
school  i immediately recalled . . . . . . . . . at . . . . . . . . . . . high . a
classic line inspector i  m here to sack one of your teachers . student welcome
to bromwell high . i expect that many adults of my age think that bromwell high
is far fetched . what a pity that it isn  t

positive
```

[ ]:
```
# Remove punctuations from the reviews and split words
raw_reviews, words, labels = reviews_preprocess(reviews_lines, label_lines)
print(raw_reviews[0])
print("First ten words: ", words[:10])
print("First label: ", labels[0])
```

```
bromwell high is a cartoon comedy it ran at the same time as some other programs
about school life such as teachers my years in the teaching profession lead me
to believe that bromwell high s satire is much closer to reality than is
teachers the scramble to survive financially the insightful students who can see
```

```
right through their pathetic teachers pomp the pettiness of the whole situation
all remind me of the schools i knew and their students when i saw the episode in
which a student repeatedly tried to burn down the school i immediately recalled
at high a classic line inspector i m here to sack one of your teachers student
welcome to bromwell high i expect that many adults of my age think that bromwell
high is far fetched what a pity that it isn t
First ten words:  ['bromwell', 'high', 'is', 'a', 'cartoon', 'comedy', 'it',
'ran', 'at', 'the']
First label:  1
```

[ ]:
```python
# Determine an integer id for each unique word
word_ids = reviews_create_word_ids(words)
print(word_ids.get("the"))
print(word_ids.get("movie"))
```

```
1
16
```

[ ]:
```python
# Encode each word in the review by its unique identifier
encoded_reviews = reviews_encode(word_ids, raw_reviews)
print(raw_reviews[0])
print(encoded_reviews[0])
```

```
bromwell high is a cartoon comedy it ran at the same time as some other programs
about school life such as teachers my years in the teaching profession lead me
to believe that bromwell high s satire is much closer to reality than is
teachers the scramble to survive financially the insightful students who can see
right through their pathetic teachers pomp the pettiness of the whole situation
all remind me of the schools i knew and their students when i saw the episode in
which a student repeatedly tried to burn down the school i immediately recalled
at high a classic line inspector i m here to sack one of your teachers student
welcome to bromwell high i expect that many adults of my age think that bromwell
high is far fetched what a pity that it isn t
[10455, 307, 6, 3, 1177, 202, 8, 2217, 33, 1, 168, 56, 15, 49, 85, 8269, 43,
422, 122, 140, 15, 3151, 59, 144, 9, 1, 5230, 5946, 452, 72, 5, 260, 12, 10455,
307, 13, 2017, 6, 73, 2765, 5, 689, 76, 6, 3151, 1, 19565, 5, 1706, 6897, 1,
5947, 1707, 36, 52, 68, 211, 143, 63, 1390, 3151, 14816, 1, 19566, 4, 1, 221,
755, 31, 2710, 72, 4, 1, 5948, 10, 729, 2, 63, 1707, 54, 10, 208, 1, 321, 9, 64,
3, 1578, 3922, 737, 5, 2843, 187, 1, 422, 10, 1246, 9217, 33, 307, 3, 380, 322,
5949, 10, 135, 136, 5, 9218, 30, 4, 134, 3151, 1578, 2480, 5, 10455, 307, 10,
528, 12, 113, 1839, 4, 59, 676, 103, 12, 10455, 307, 6, 227, 4097, 48, 3, 2169,
12, 8, 231, 21]
```

[ ]:
```python
# Padding/truncating all reviews to the same length. Although this isn't␣
↪strictly
# necessary, it facilitates batch processing: all inputs of a batch need to␣
↪have the
```

```
# same length.
sequence_length = 200
padded_reviews = reviews_pad(encoded_reviews, sequence_length)
print(padded_reviews[0])
```

```
[    0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
 10455   307     6     3  1177   202     8  2217    33     1   168    56
    15    49    85  8269    43   422   122   140    15  3151    59   144
     9     1  5230  5946   452    72     5   260    12 10455   307    13
  2017     6    73  2765     5   689    76     6  3151     1 19565     5
  1706  6897     1  5947  1707    36    52    68   211   143    63  1390
  3151 14816     1 19566     4     1   221   755    31  2710    72     4
     1  5948    10   729     2    63  1707    54    10   208     1   321
     9    64     3  1578  3922   737     5  2843   187     1   422    10
  1246  9217    33   307     3   380   322  5949    10   135   136     5
  9218    30     4   134  3151  1578  2480     5 10455   307    10   528
    12   113  1839     4    59   676   103    12 10455   307     6   227
  4097    48     3  2169    12     8   231    21]
```

[ ]:
```
# Split dataset into 80% training, 10% test, and 10% validation Dataset
train_x, train_y, valid_x, valid_y, test_x, test_y = reviews_split(
    padded_reviews, labels
)
print(len(train_y), len(valid_y), len(test_y))
```

```
3200 400 400
```

[ ]:
```
# Create data loaders for training
train_loader, valid_loader, test_loader = reviews_create_dataloaders(
    train_x, train_y, valid_x, valid_y, test_x, test_y
)
```

[ ]:
```
# Here is an example how to use the train and test functions. Note that logistic
# regression is a bogus model when used like this (since its assigns weights to
# positions, but not word ids). So results will be bad.
model = LogisticRegression(sequence_length).to(DEVICE)
reviews_train(model, train_loader, valid_loader, epochs=3, device=DEVICE)
reviews_test(model, test_loader, device=DEVICE)
```

```
Starting epoch 1
Epoch:  1/ 3    Batch:  5    Batch loss: 55.862564    Val loss: 52.007381 Val
acc: 0.470000
Epoch:  1/ 3    Batch: 10    Batch loss: 44.230236    Val loss: 50.456723 Val
acc: 0.490000
```

```
Epoch:  1/ 3    Batch: 15   Batch loss: 50.000000    Val loss: 50.462179 Val
acc: 0.490000
Epoch:  1/ 3    Batch: 20   Batch loss: 44.000000    Val loss: 50.634626 Val
acc: 0.490000
Epoch:  1/ 3    Batch: 25   Batch loss: 32.900143    Val loss: 52.853641 Val
acc: 0.470000
Epoch:  1/ 3    Batch: 30   Batch loss: 52.671864    Val loss: 51.840658 Val
acc: 0.480000
Epoch:  1/ 3    Batch: 35   Batch loss: 62.000000    Val loss: 52.332016 Val
acc: 0.470000
Epoch:  1/ 3    Batch: 40   Batch loss: 53.558922    Val loss: 51.445327 Val
acc: 0.480000
Epoch:  1/ 3    Batch: 45   Batch loss: 48.000038    Val loss: 51.347229 Val
acc: 0.480000
Epoch:  1/ 3    Batch: 50   Batch loss: 58.108727    Val loss: 51.581363 Val
acc: 0.480000
Epoch:  1/ 3    Batch: 55   Batch loss: 54.000000    Val loss: 51.581363 Val
acc: 0.480000
Epoch:  1/ 3    Batch: 60   Batch loss: 46.955475    Val loss: 51.581358 Val
acc: 0.480000
96
Finished epoch 1. Average batch loss: 49.876608073711395. Average validation
loss: 51.51032213370005
Starting epoch 2
Epoch:  2/ 3    Batch: 65   Batch loss: 54.000000    Val loss: 50.764032 Val
acc: 0.490000
Epoch:  2/ 3    Batch: 70   Batch loss: 46.027721    Val loss: 51.250000 Val
acc: 0.487500
Epoch:  2/ 3    Batch: 75   Batch loss: 52.763466    Val loss: 50.798164 Val
acc: 0.487500
Epoch:  2/ 3    Batch: 80   Batch loss: 46.976650    Val loss: 51.250000 Val
acc: 0.487500
Epoch:  2/ 3    Batch: 85   Batch loss: 52.000000    Val loss: 51.250000 Val
acc: 0.487500
Epoch:  2/ 3    Batch: 90   Batch loss: 44.125820    Val loss: 51.478319 Val
acc: 0.477500
Epoch:  2/ 3    Batch: 95   Batch loss: 42.000000    Val loss: 51.478445 Val
acc: 0.477500
Epoch:  2/ 3    Batch: 100   Batch loss: 56.000000    Val loss: 51.692595 Val
acc: 0.480000
Epoch:  2/ 3    Batch: 105   Batch loss: 38.283012    Val loss: 52.004820 Val
acc: 0.477500
Epoch:  2/ 3    Batch: 110   Batch loss: 42.000000    Val loss: 51.599065 Val
acc: 0.482500
Epoch:  2/ 3    Batch: 115   Batch loss: 45.428814    Val loss: 51.441724 Val
acc: 0.482500
Epoch:  2/ 3    Batch: 120   Batch loss: 52.000000    Val loss: 51.477280 Val
acc: 0.482500
```

```
Epoch:  2/ 3    Batch: 125   Batch loss: 40.000000    Val loss: 51.572995 Val
acc: 0.482500
104
Finished epoch 2. Average batch loss: 49.45479238033295. Average validation
loss: 51.38903372104351
Starting epoch 3
Epoch:  3/ 3    Batch: 130   Batch loss: 50.000000    Val loss: 51.572995 Val
acc: 0.482500
Epoch:  3/ 3    Batch: 135   Batch loss: 57.750259    Val loss: 50.593139 Val
acc: 0.487500
Epoch:  3/ 3    Batch: 140   Batch loss: 50.000000    Val loss: 51.007401 Val
acc: 0.487500
Epoch:  3/ 3    Batch: 145   Batch loss: 60.000000    Val loss: 50.301833 Val
acc: 0.492500
Epoch:  3/ 3    Batch: 150   Batch loss: 51.195168    Val loss: 50.301455 Val
acc: 0.492500
Epoch:  3/ 3    Batch: 155   Batch loss: 52.000000    Val loss: 50.271038 Val
acc: 0.492500
Epoch:  3/ 3    Batch: 160   Batch loss: 54.000942    Val loss: 50.500000 Val
acc: 0.495000
Epoch:  3/ 3    Batch: 165   Batch loss: 51.285694    Val loss: 50.500000 Val
acc: 0.495000
Epoch:  3/ 3    Batch: 170   Batch loss: 60.000000    Val loss: 50.500000 Val
acc: 0.495000
Epoch:  3/ 3    Batch: 175   Batch loss: 39.232018    Val loss: 50.030633 Val
acc: 0.497500
Epoch:  3/ 3    Batch: 180   Batch loss: 58.000000    Val loss: 50.030633 Val
acc: 0.497500
Epoch:  3/ 3    Batch: 185   Batch loss: 58.000000    Val loss: 50.431529 Val
acc: 0.495000
Epoch:  3/ 3    Batch: 190   Batch loss: 60.000000    Val loss: 50.116792 Val
acc: 0.495000
104
Finished epoch 3. Average batch loss: 49.30076479911804. Average validation
loss: 50.473649795238785
Test loss: 49.750
Test accuracy: 0.502
```

2.2 2a Define your model

[ ]:
```python
# Create an LSTM for sentiment analysis
class SimpleLSTM(nn.Module):
    """
    The RNN model that will be used to perform sentiment analysis.
    """

    def __init__(
```

```
        self,
        vocab_size,
        embedding_dim,
        hidden_dim,
        num_layers=1,
        lstm_dropout_prob=0.5,
        dropout_prob=0.3,
    ):
        """
        Initialize the model by setting up the layers

        Parameters
        ----------
        vocab_size: number of unique words in the reviews
        embeddings_dim: size of the embeddings
        hidden_dim: dimension of the LSTM output
        num_layers: number of LSTM layers
        lstm_dropout_prob: dropout applied between the LSTM layers
        dropout_prob: dropout applied before the fully connected layer
        """
        super().__init__()

        self.num_layers = num_layers
        self.hidden_dim = hidden_dim

        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim
            )

        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=lstm_dropout_prob
            )

        self.dropout = nn.Dropout(
            p=dropout_prob,
            inplace=False
            )

        self.fc = nn.Linear(
            in_features=hidden_dim,
            out_features=1
            )
```

```python
        self.sigmoid = nn.Sigmoid()


        # YOUR CODE HERE

    def forward(self, x):
        """
        Perform a forward pass of our model on some input and hidden state.

        Parameters
        ----------
        x: batch as a (batch_size, sequence_length) tensor

        Returns
        -------
        Probability of positive class.
        """
        # init hidden layer, which is needed for the LSTM
        batch_size = len(x)
        hidden = self.init_hidden(batch_size)

        # YOUR CODE HERE
        embedding = self.embedding(x)
        out, (hidden, cell_state) = self.lstm(embedding, hidden)
        out = self.dropout(out[:, -1, :])
        out = self.fc(out)
        out = self.sigmoid(out)
        return out


    def init_hidden(self, batch_size):
        """
        Initialize hidden state.

        Returns
        -------
        Empty hidden LSTM state.
        """

        # Create two new tensors with sizes num_layers x batch_size x
↪hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters())  # only used to determine device

        hidden = (
            weight.new(self.num_layers, batch_size, self.hidden_dim).zero_(),
```

```
            weight.new(self.num_layers, batch_size, self.hidden_dim).zero_(),
        )

        return hidden
```

[ ]:
```
# Test model setup
lstm_model = SimpleLSTM(1, 10, 32, 2, 0, 0).to(DEVICE)
print(lstm_model)

# SimpleLSTM(
#   (embedding): Embedding(1, 10)
#   (lstm): LSTM(10, 32, num_layers=2, batch_first=True, dropout=0.0)
#   (dropout): Dropout(p=0.0, inplace=False)
#   (fc): Linear(in_features=32, out_features=1, bias=True)
#   (sigmoid): Sigmoid()
# )
```

```
SimpleLSTM(
  (embedding): Embedding(1, 10)
  (lstm): LSTM(10, 32, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0, inplace=False)
  (fc): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

[ ]:
```
# Test forward function.
# dummy data
dummy_data = torch.zeros(train_loader.batch_size, sequence_length).long().
 ↪to(DEVICE)
# fix model parameters
for key in lstm_model.state_dict():
    lstm_model.state_dict()[key][:] = 0.1
print(lstm_model(dummy_data).reshape(10, -1))
# Output after reshape should be the following tensor
#tensor([[0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
#         [0.9643, 0.9643, 0.9643, 0.9643, 0.9643]], device='cuda or cpu',
#        grad_fn=<ViewBackward>)
```

```
tensor([[0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
```

```
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643],
 [0.9643, 0.9643, 0.9643, 0.9643, 0.9643]],
grad_fn=<ReshapeAliasBackward0>)
```

2.3 2b Train and evaluate the model

[ ]:
```
# Instantiate the model w/ hyperparams
vocab_size = len(word_ids) + 1  # +1 for the 0 padding
embedding_dim = 100
hidden_dim = 64
num_layers = 1

# this may raise a warning when num_layers=1 (which is fine)
# make sure to reinizialize the model if you want to train multiple times
lstm_model = SimpleLSTM(vocab_size, embedding_dim, hidden_dim, num_layers).
 ↪to(DEVICE)
```

[ ]:
```
# Fit and evaluate a model (without pretrained embeddings)
n_epochs = 5
# YOUR CODE HERE
reviews_train(lstm_model, train_loader, valid_loader, epochs=5, device=DEVICE)
```

```
Starting epoch 1
Epoch:  1/ 5    Batch:  5   Batch loss: 0.679525    Val loss: 0.703965 Val acc:
0.495000
Epoch:  1/ 5    Batch: 10   Batch loss: 0.698851    Val loss: 0.695553 Val acc:
0.517500
Epoch:  1/ 5    Batch: 15   Batch loss: 0.705797    Val loss: 0.695675 Val acc:
0.512500
Epoch:  1/ 5    Batch: 20   Batch loss: 0.708614    Val loss: 0.694467 Val acc:
0.550000
Epoch:  1/ 5    Batch: 25   Batch loss: 0.667098    Val loss: 0.693591 Val acc:
0.547500
Epoch:  1/ 5    Batch: 30   Batch loss: 0.688021    Val loss: 0.694176 Val acc:
0.530000
Epoch:  1/ 5    Batch: 35   Batch loss: 0.679528    Val loss: 0.693740 Val acc:
0.540000
Epoch:  1/ 5    Batch: 40   Batch loss: 0.673149    Val loss: 0.693195 Val acc:
0.552500
Epoch:  1/ 5    Batch: 45   Batch loss: 0.683008    Val loss: 0.691528 Val acc:
0.547500
```

```
Epoch:  1/ 5    Batch: 50   Batch loss: 0.701330    Val loss: 0.689682 Val acc:
0.550000
Epoch:  1/ 5    Batch: 55   Batch loss: 0.668185    Val loss: 0.687032 Val acc:
0.547500
Epoch:  1/ 5    Batch: 60   Batch loss: 0.677919    Val loss: 0.685461 Val acc:
0.565000
96
Finished epoch 1. Average batch loss: 0.6865371307358146. Average validation
loss: 0.693172030771772
Starting epoch 2
Epoch:  2/ 5    Batch: 65   Batch loss: 0.672716    Val loss: 0.684752 Val acc:
0.555000
Epoch:  2/ 5    Batch: 70   Batch loss: 0.634400    Val loss: 0.682325 Val acc:
0.562500
Epoch:  2/ 5    Batch: 75   Batch loss: 0.627135    Val loss: 0.677084 Val acc:
0.597500
Epoch:  2/ 5    Batch: 80   Batch loss: 0.644966    Val loss: 0.679041 Val acc:
0.555000
Epoch:  2/ 5    Batch: 85   Batch loss: 0.568219    Val loss: 0.812599 Val acc:
0.535000
Epoch:  2/ 5    Batch: 90   Batch loss: 0.628705    Val loss: 0.682299 Val acc:
0.557500
Epoch:  2/ 5    Batch: 95   Batch loss: 0.622430    Val loss: 0.671059 Val acc:
0.600000
Epoch:  2/ 5    Batch: 100   Batch loss: 0.538470    Val loss: 0.666089 Val acc:
0.600000
Epoch:  2/ 5    Batch: 105   Batch loss: 0.653860    Val loss: 0.669573 Val acc:
0.605000
Epoch:  2/ 5    Batch: 110   Batch loss: 0.577353    Val loss: 0.665001 Val acc:
0.612500
Epoch:  2/ 5    Batch: 115   Batch loss: 0.587399    Val loss: 0.657142 Val acc:
0.615000
Epoch:  2/ 5    Batch: 120   Batch loss: 0.686283    Val loss: 0.650623 Val acc:
0.625000
Epoch:  2/ 5    Batch: 125   Batch loss: 0.596217    Val loss: 0.715745 Val acc:
0.575000
104
Finished epoch 2. Average batch loss: 0.6149493930861354. Average validation
loss: 0.6856409878684924
Starting epoch 3
Epoch:  3/ 5    Batch: 130   Batch loss: 0.498110    Val loss: 0.642683 Val acc:
0.632500
Epoch:  3/ 5    Batch: 135   Batch loss: 0.568852    Val loss: 0.667370 Val acc:
0.610000
Epoch:  3/ 5    Batch: 140   Batch loss: 0.532824    Val loss: 0.660348 Val acc:
0.622500
Epoch:  3/ 5    Batch: 145   Batch loss: 0.475732    Val loss: 0.659632 Val acc:
0.637500
```

```
Epoch:  3/ 5    Batch: 150   Batch loss: 0.547082    Val loss: 0.698349 Val acc:
0.590000
Epoch:  3/ 5    Batch: 155   Batch loss: 0.509110    Val loss: 0.642167 Val acc:
0.652500
Epoch:  3/ 5    Batch: 160   Batch loss: 0.507828    Val loss: 0.656309 Val acc:
0.645000
Epoch:  3/ 5    Batch: 165   Batch loss: 0.442918    Val loss: 0.642163 Val acc:
0.647500
Epoch:  3/ 5    Batch: 170   Batch loss: 0.548849    Val loss: 0.642707 Val acc:
0.650000
Epoch:  3/ 5    Batch: 175   Batch loss: 0.440895    Val loss: 0.635169 Val acc:
0.635000
Epoch:  3/ 5    Batch: 180   Batch loss: 0.491875    Val loss: 0.649510 Val acc:
0.650000
Epoch:  3/ 5    Batch: 185   Batch loss: 0.967413    Val loss: 0.647366 Val acc:
0.637500
Epoch:  3/ 5    Batch: 190   Batch loss: 0.559213    Val loss: 0.735722 Val acc:
0.542500
104
Finished epoch 3. Average batch loss: 0.522889809217304. Average validation
loss: 0.6599610298871994
Starting epoch 4
Epoch:  4/ 5    Batch: 195   Batch loss: 0.521300    Val loss: 0.649883 Val acc:
0.645000
Epoch:  4/ 5    Batch: 200   Batch loss: 0.353082    Val loss: 0.648531 Val acc:
0.652500
Epoch:  4/ 5    Batch: 205   Batch loss: 0.409361    Val loss: 0.635625 Val acc:
0.670000
Epoch:  4/ 5    Batch: 210   Batch loss: 0.418146    Val loss: 0.628939 Val acc:
0.662500
Epoch:  4/ 5    Batch: 215   Batch loss: 0.571482    Val loss: 0.676091 Val acc:
0.610000
Epoch:  4/ 5    Batch: 220   Batch loss: 0.525052    Val loss: 0.682291 Val acc:
0.632500
Epoch:  4/ 5    Batch: 225   Batch loss: 0.456031    Val loss: 0.654979 Val acc:
0.665000
Epoch:  4/ 5    Batch: 230   Batch loss: 0.340121    Val loss: 0.654314 Val acc:
0.652500
Epoch:  4/ 5    Batch: 235   Batch loss: 0.343296    Val loss: 0.661900 Val acc:
0.655000
Epoch:  4/ 5    Batch: 240   Batch loss: 0.322669    Val loss: 0.652139 Val acc:
0.665000
Epoch:  4/ 5    Batch: 245   Batch loss: 0.371951    Val loss: 0.722962 Val acc:
0.642500
Epoch:  4/ 5    Batch: 250   Batch loss: 0.364596    Val loss: 0.744807 Val acc:
0.665000
Epoch:  4/ 5    Batch: 255   Batch loss: 0.424592    Val loss: 0.642369 Val acc:
0.662500
```

```
104
Finished epoch 4. Average batch loss: 0.42475875513628125. Average validation
loss: 0.6657560762877648
Starting epoch 5
Epoch:  5/ 5    Batch: 260   Batch loss: 0.520416    Val loss: 0.673290 Val acc:
0.657500
Epoch:  5/ 5    Batch: 265   Batch loss: 0.435737    Val loss: 0.651521 Val acc:
0.667500
Epoch:  5/ 5    Batch: 270   Batch loss: 0.303476    Val loss: 0.634039 Val acc:
0.672500
Epoch:  5/ 5    Batch: 275   Batch loss: 0.378886    Val loss: 0.698585 Val acc:
0.647500
Epoch:  5/ 5    Batch: 280   Batch loss: 0.423104    Val loss: 0.630883 Val acc:
0.687500
Epoch:  5/ 5    Batch: 285   Batch loss: 0.296592    Val loss: 0.645418 Val acc:
0.687500
Epoch:  5/ 5    Batch: 290   Batch loss: 0.341979    Val loss: 0.657843 Val acc:
0.682500
Epoch:  5/ 5    Batch: 295   Batch loss: 0.260322    Val loss: 0.674640 Val acc:
0.675000
Epoch:  5/ 5    Batch: 300   Batch loss: 0.216686    Val loss: 0.748435 Val acc:
0.657500
Epoch:  5/ 5    Batch: 305   Batch loss: 0.300631    Val loss: 0.652457 Val acc:
0.672500
Epoch:  5/ 5    Batch: 310   Batch loss: 0.452203    Val loss: 0.664092 Val acc:
0.647500
Epoch:  5/ 5    Batch: 315   Batch loss: 0.334228    Val loss: 0.669618 Val acc:
0.675000
Epoch:  5/ 5    Batch: 320   Batch loss: 0.356790    Val loss: 0.666637 Val acc:
0.690000
104
Finished epoch 5. Average batch loss: 0.3577388192061335. Average validation
loss: 0.6667274890037683
```

[ ]:
```
reviews_test(lstm_model, test_loader, device=DEVICE)
```

```
Test loss: 0.918
Test accuracy: 0.682
```

2.4 2c Load pretrained word embeddings

[ ]:
```
@torch.no_grad()
def reviews_load_embeddings(
    embedding_layer, word_ids, pretrained_embeddings_file="data/word-embeddings.
 ↪txt"
):
    """Load pretrained embeddings into an embedding layer.
```

```
    Updates the weights of the embedding layer with with the embeddings given␣
↪in the
    provided word embeddings file.

    Parameters
    ----------
    embedding_layer: torch.nn.Embedding used in the model
    word_ids: dictionary mapping each word to its unique identifier
    pretrained_embeddings_file: path to the file containing pretrained␣
↪embeddings

    """
    print("Initializing embedding layer with pretrained word embeddings...")
    embeddings_index = dict()
    words_initialized = 0
    with open(pretrained_embeddings_file, encoding="utf8") as f:
        for line in f:
            values = line.split()
            word = values[0]
            encoded_word = word_ids.get(word)
            if encoded_word is not None:
                words_initialized += 1
                embedding_layer.weight[encoded_word, :] = torch.from_numpy(
                    np.asarray(values[1:], dtype="float32")
                )
    print(
        "Initialized {}/{} word embeddings".format(
            words_initialized, embedding_layer.num_embeddings
        )
    )
```

[ ]:
```
# Try it
test_embeddings = nn.Embedding(len(word_ids) + 1, 100).to(DEVICE)
reviews_load_embeddings(test_embeddings, word_ids)
print(test_embeddings(torch.LongTensor([word_ids.get("movie")]).to(DEVICE)))
del test_embeddings
```

```
Initializing embedding layer with pretrained word embeddings…
Initialized 29841/32363 word embeddings
tensor([[ 0.3825,  0.1482,  0.6060, -0.5153,  0.4399,  0.0611, -0.6272, -0.0254,
          0.1643, -0.2210,  0.1442, -0.3721, -0.2168, -0.0890,  0.0979,  0.6561,
          0.6446,  0.4770,  0.8385,  1.6486,  0.8892, -0.1181, -0.0125, -0.5208,
          0.7785,  0.4872, -0.0150, -0.1413, -0.3475, -0.2959,  0.1028,  0.5719,
         -0.0456,  0.0264,  0.5382,  0.3226,  0.4079, -0.0436, -0.1460, -0.4835,
          0.3204,  0.5509, -0.7626,  0.4327,  0.6175, -0.3650, -0.6060, -0.7962,
          0.3929, -0.2367, -0.3472, -0.6120,  0.5475,  0.9481,  0.2094, -2.7771,
```

```
        -0.6022,  0.8495,  1.2549,  0.0179, -0.0419,  2.1147, -0.0266, -0.2810,
         0.6812, -0.1417,  0.9925,  0.4988, -0.6754,  0.6417,  0.4230, -0.2791,
         0.0634,  0.6891, -0.3618,  0.0537, -0.1681,  0.1942, -0.4707, -0.1480,
        -0.5899, -0.2797,  0.1679,  0.1057, -1.7601,  0.0088, -0.8333, -0.5836,
        -0.3708, -0.5659,  0.2070,  0.0713,  0.0556, -0.2976, -0.0727, -0.2560,
         0.4269,  0.0589,  0.0911,  0.4728]], grad_fn=<EmbeddingBackward0>)
```

[ ]:
```python
embbed_dict = {}
number_lines = 0
with open("data/word-embeddings.txt") as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:],'float32')
        emmbed_dict[word]=vector
        number_lines += 1
print(f"Number of words in Glove Embedding: {number_lines}")
```

```
Number of words in Glove Embedding: 29841
```

[ ]:
```python
len(emmbed_dict['in'])
# each word is associated with a vector of length 100
```

[ ]: 100

length of word ids is 32363 and the number of words in the embedding file is 29841; every word from the embedding file is used in the review dataset

2.5 2d Train and evaluate with pretraining

[ ]:
```python
# Fit and evaluate a model with pretrained embeddings without fine-tuning
class EmbeddingLSTM(nn.Module):
    """
    The RNN model that will be used to perform sentiment analysis.
    """

    def __init__(
        self,
        vocab_size,
        embedding_dim,
        hidden_dim,
        num_layers=1,
        lstm_dropout_prob=0.5,
        dropout_prob=0.3,
        finetuning=False
    ):
        """
        Initialize the model by setting up the layers
```

```
        Parameters
        ----------
        vocab_size: number of unique words in the reviews
        embeddings_dim: size of the embeddings
        hidden_dim: dimension of the LSTM output
        num_layers: number of LSTM layers
        lstm_dropout_prob: dropout applied between the LSTM layers
        dropout_prob: dropout applied before the fully connected layer
        """
        super().__init__()
        self.finetuning = finetuning
        self.num_layers = num_layers
        self.hidden_dim = hidden_dim

        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim
            )

        reviews_load_embeddings(embedding_layer=self.embedding,␣
↪word_ids=word_ids)

        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=lstm_dropout_prob
            )

        self.dropout = nn.Dropout(
            p=dropout_prob,
            inplace=False
            )

        self.fc = nn.Linear(
            in_features=hidden_dim,
            out_features=1
            )

        self.sigmoid = nn.Sigmoid()


        # YOUR CODE HERE

    def forward(self, x):
```

```
        """
        Perform a forward pass of our model on some input and hidden state.

        Parameters
        ----------
        x: batch as a (batch_size, sequence_length) tensor

        Returns
        -------
        Probability of positive class.
        """
        # init hidden layer, which is needed for the LSTM
        batch_size = len(x)
        hidden = self.init_hidden(batch_size)

        # YOUR CODE HERE

        self.embedding.weight.requires_grad = self.finetuning
        embedding = self.embedding(x)
        out, (hidden, cell_state) = self.lstm(embedding, hidden)
        out = self.dropout(out[:, -1, :])
        out = self.fc(out)
        out = self.sigmoid(out)
        return out

    def init_hidden(self, batch_size):
        """
        Initialize hidden state.

        Returns
        -------
        Empty hidden LSTM state.
        """

        # Create two new tensors with sizes num_layers x batch_size x␣
↪hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters())  # only used to determine device

        hidden = (
            weight.new(self.num_layers, batch_size, self.hidden_dim).zero_(),
            weight.new(self.num_layers, batch_size, self.hidden_dim).zero_(),
        )

        return hidden
```

```
[ ]: # Fit and evaluate a model with pretrained embeddings without fine-tuning.
# YOUR CODE HERE
vocab_size = len(word_ids) + 1  # +1 for the 0 padding
embedding_dim = 100
hidden_dim = 64
num_layers = 1

embedding_model = EmbeddingLSTM(vocab_size, embedding_dim, hidden_dim,
 ↪num_layers, finetuning=False).to(DEVICE)
reviews_train(embedding_model, train_loader, valid_loader, epochs=5,
 ↪device=DEVICE)
```

```
Initializing embedding layer with pretrained word embeddings…
Initialized 29841/32363 word embeddings
Starting epoch 1
Epoch:  1/ 5    Batch:  5   Batch loss: 0.686438    Val loss: 0.706561 Val acc:
0.515000
Epoch:  1/ 5    Batch: 10   Batch loss: 0.695074    Val loss: 0.686327 Val acc:
0.527500
Epoch:  1/ 5    Batch: 15   Batch loss: 0.666638    Val loss: 0.684639 Val acc:
0.532500
Epoch:  1/ 5    Batch: 20   Batch loss: 0.679411    Val loss: 0.683167 Val acc:
0.547500
Epoch:  1/ 5    Batch: 25   Batch loss: 0.628898    Val loss: 0.688541 Val acc:
0.545000
Epoch:  1/ 5    Batch: 30   Batch loss: 0.701615    Val loss: 0.675063 Val acc:
0.570000
Epoch:  1/ 5    Batch: 35   Batch loss: 0.747103    Val loss: 0.689529 Val acc:
0.550000
Epoch:  1/ 5    Batch: 40   Batch loss: 0.657241    Val loss: 0.673139 Val acc:
0.570000
Epoch:  1/ 5    Batch: 45   Batch loss: 0.644778    Val loss: 0.669597 Val acc:
0.582500
Epoch:  1/ 5    Batch: 50   Batch loss: 0.652246    Val loss: 0.678360 Val acc:
0.597500
Epoch:  1/ 5    Batch: 55   Batch loss: 0.661263    Val loss: 0.719608 Val acc:
0.540000
Epoch:  1/ 5    Batch: 60   Batch loss: 0.702641    Val loss: 0.664131 Val acc:
0.572500
96
Finished epoch 1. Average batch loss: 0.6720854407176375. Average validation
loss: 0.6848886503527561
Starting epoch 2
Epoch:  2/ 5    Batch: 65   Batch loss: 0.678968    Val loss: 0.684704 Val acc:
0.590000
Epoch:  2/ 5    Batch: 70   Batch loss: 0.598958    Val loss: 0.741897 Val acc:
0.520000
```

```
Epoch:  2/ 5    Batch: 75   Batch loss: 0.743488    Val loss: 0.681598 Val acc:
0.607500
Epoch:  2/ 5    Batch: 80   Batch loss: 0.607069    Val loss: 0.657020 Val acc:
0.590000
Epoch:  2/ 5    Batch: 85   Batch loss: 0.657863    Val loss: 0.651997 Val acc:
0.602500
Epoch:  2/ 5    Batch: 90   Batch loss: 0.731028    Val loss: 0.658597 Val acc:
0.587500
Epoch:  2/ 5    Batch: 95   Batch loss: 0.577248    Val loss: 0.663950 Val acc:
0.612500
Epoch:  2/ 5    Batch: 100   Batch loss: 0.662151    Val loss: 0.715565 Val acc:
0.520000
Epoch:  2/ 5    Batch: 105   Batch loss: 0.610220    Val loss: 0.663269 Val acc:
0.650000
Epoch:  2/ 5    Batch: 110   Batch loss: 0.628149    Val loss: 0.650179 Val acc:
0.635000
Epoch:  2/ 5    Batch: 115   Batch loss: 0.596552    Val loss: 0.646293 Val acc:
0.610000
Epoch:  2/ 5    Batch: 120   Batch loss: 0.539033    Val loss: 0.621515 Val acc:
0.665000
Epoch:  2/ 5    Batch: 125   Batch loss: 0.708193    Val loss: 0.645570 Val acc:
0.637500
104
Finished epoch 2. Average batch loss: 0.6357951182872057. Average validation
loss: 0.667857927771715
Starting epoch 3
Epoch:  3/ 5    Batch: 130   Batch loss: 0.581725    Val loss: 0.655778 Val acc:
0.625000
Epoch:  3/ 5    Batch: 135   Batch loss: 0.534587    Val loss: 0.664557 Val acc:
0.612500
Epoch:  3/ 5    Batch: 140   Batch loss: 0.614694    Val loss: 0.673386 Val acc:
0.595000
Epoch:  3/ 5    Batch: 145   Batch loss: 0.570041    Val loss: 0.653104 Val acc:
0.625000
Epoch:  3/ 5    Batch: 150   Batch loss: 0.675055    Val loss: 0.649794 Val acc:
0.622500
Epoch:  3/ 5    Batch: 155   Batch loss: 0.597248    Val loss: 0.634350 Val acc:
0.642500
Epoch:  3/ 5    Batch: 160   Batch loss: 0.583497    Val loss: 0.630251 Val acc:
0.657500
Epoch:  3/ 5    Batch: 165   Batch loss: 0.585778    Val loss: 0.632745 Val acc:
0.657500
Epoch:  3/ 5    Batch: 170   Batch loss: 0.667148    Val loss: 0.627554 Val acc:
0.657500
Epoch:  3/ 5    Batch: 175   Batch loss: 0.560197    Val loss: 0.626309 Val acc:
0.662500
Epoch:  3/ 5    Batch: 180   Batch loss: 0.580412    Val loss: 0.639708 Val acc:
0.642500
```

```
Epoch:  3/ 5    Batch: 185   Batch loss: 0.577425    Val loss: 0.625761 Val acc:
0.665000
Epoch:  3/ 5    Batch: 190   Batch loss: 0.779278    Val loss: 0.649485 Val acc:
0.625000
104
Finished epoch 3. Average batch loss: 0.5963751999661326. Average validation
loss: 0.6432909939724666
Starting epoch 4
Epoch:  4/ 5    Batch: 195   Batch loss: 0.526449    Val loss: 0.640628 Val acc:
0.637500
Epoch:  4/ 5    Batch: 200   Batch loss: 0.608472    Val loss: 0.644570 Val acc:
0.632500
Epoch:  4/ 5    Batch: 205   Batch loss: 0.664701    Val loss: 0.649488 Val acc:
0.622500
Epoch:  4/ 5    Batch: 210   Batch loss: 0.527539    Val loss: 0.618145 Val acc:
0.680000
Epoch:  4/ 5    Batch: 215   Batch loss: 0.519513    Val loss: 0.619536 Val acc:
0.675000
Epoch:  4/ 5    Batch: 220   Batch loss: 0.588045    Val loss: 0.626369 Val acc:
0.657500
Epoch:  4/ 5    Batch: 225   Batch loss: 0.554402    Val loss: 0.618975 Val acc:
0.672500
Epoch:  4/ 5    Batch: 230   Batch loss: 0.639828    Val loss: 0.627784 Val acc:
0.657500
Epoch:  4/ 5    Batch: 235   Batch loss: 0.578442    Val loss: 0.611042 Val acc:
0.680000
Epoch:  4/ 5    Batch: 240   Batch loss: 0.552774    Val loss: 0.628441 Val acc:
0.660000
Epoch:  4/ 5    Batch: 245   Batch loss: 0.494681    Val loss: 0.634671 Val acc:
0.632500
Epoch:  4/ 5    Batch: 250   Batch loss: 0.656134    Val loss: 0.699023 Val acc:
0.590000
Epoch:  4/ 5    Batch: 255   Batch loss: 0.574947    Val loss: 0.610381 Val acc:
0.675000
104
Finished epoch 4. Average batch loss: 0.5753812100738287. Average validation
loss: 0.633004194841935
Starting epoch 5
Epoch:  5/ 5    Batch: 260   Batch loss: 0.566240    Val loss: 0.615236 Val acc:
0.687500
Epoch:  5/ 5    Batch: 265   Batch loss: 0.482931    Val loss: 0.609328 Val acc:
0.690000
Epoch:  5/ 5    Batch: 270   Batch loss: 0.456848    Val loss: 0.610621 Val acc:
0.667500
Epoch:  5/ 5    Batch: 275   Batch loss: 0.505860    Val loss: 0.605004 Val acc:
0.682500
Epoch:  5/ 5    Batch: 280   Batch loss: 0.534595    Val loss: 0.607091 Val acc:
0.677500
```

```
Epoch:  5/ 5    Batch: 285   Batch loss: 0.502120    Val loss: 0.613821 Val acc:
0.682500
Epoch:  5/ 5    Batch: 290   Batch loss: 0.599801    Val loss: 0.617692 Val acc:
0.660000
Epoch:  5/ 5    Batch: 295   Batch loss: 0.557508    Val loss: 0.614699 Val acc:
0.650000
Epoch:  5/ 5    Batch: 300   Batch loss: 0.545653    Val loss: 0.613553 Val acc:
0.677500
Epoch:  5/ 5    Batch: 305   Batch loss: 0.559772    Val loss: 0.665818 Val acc:
0.655000
Epoch:  5/ 5    Batch: 310   Batch loss: 0.538530    Val loss: 0.603941 Val acc:
0.672500
Epoch:  5/ 5    Batch: 315   Batch loss: 0.640959    Val loss: 0.601522 Val acc:
0.677500
Epoch:  5/ 5    Batch: 320   Batch loss: 0.585376    Val loss: 0.604607 Val acc:
0.670000
104
Finished epoch 5. Average batch loss: 0.5449454039335251. Average validation
loss: 0.6140717204946738
```

[ ]:
```
reviews_test(embedding_model, test_loader, device=DEVICE)
```

```
Test loss: 0.548
Test accuracy: 0.725
```

[ ]:
```
# Fit and evaluate a model with pretrained embeddings with fine-tuning.
# YOUR CODE HERE
vocab_size = len(word_ids) + 1  # +1 for the 0 padding
embedding_dim = 100
hidden_dim = 64
num_layers = 1

embedding_model = EmbeddingLSTM(vocab_size, embedding_dim, hidden_dim,␣
 ↪num_layers, finetuning=True).to(DEVICE)
reviews_train(embedding_model, train_loader, valid_loader, epochs=5,␣
 ↪device=DEVICE)
```

```
Initializing embedding layer with pretrained word embeddings…
Initialized 29841/32363 word embeddings
Starting epoch 1
Epoch:  1/ 5    Batch:  5   Batch loss: 0.675380    Val loss: 0.703210 Val acc:
0.502500
Epoch:  1/ 5    Batch: 10   Batch loss: 0.673556    Val loss: 0.688418 Val acc:
0.552500
Epoch:  1/ 5    Batch: 15   Batch loss: 0.688058    Val loss: 0.686013 Val acc:
0.530000
Epoch:  1/ 5    Batch: 20   Batch loss: 0.692807    Val loss: 0.685314 Val acc:
0.555000
```

```
Epoch:  1/ 5    Batch: 25   Batch loss: 0.675730    Val loss: 0.689419 Val acc:
0.520000
Epoch:  1/ 5    Batch: 30   Batch loss: 0.663634    Val loss: 0.682638 Val acc:
0.555000
Epoch:  1/ 5    Batch: 35   Batch loss: 0.671265    Val loss: 0.676966 Val acc:
0.600000
Epoch:  1/ 5    Batch: 40   Batch loss: 0.675061    Val loss: 0.672660 Val acc:
0.602500
Epoch:  1/ 5    Batch: 45   Batch loss: 0.633352    Val loss: 0.673495 Val acc:
0.580000
Epoch:  1/ 5    Batch: 50   Batch loss: 0.666285    Val loss: 0.668230 Val acc:
0.622500
Epoch:  1/ 5    Batch: 55   Batch loss: 0.777652    Val loss: 0.661762 Val acc:
0.597500
Epoch:  1/ 5    Batch: 60   Batch loss: 0.606284    Val loss: 0.665211 Val acc:
0.580000
96
Finished epoch 1. Average batch loss: 0.6720362938940525. Average validation
loss: 0.6794445620228847
Starting epoch 2
Epoch:  2/ 5    Batch: 65   Batch loss: 0.541709    Val loss: 0.669934 Val acc:
0.585000
Epoch:  2/ 5    Batch: 70   Batch loss: 0.615435    Val loss: 0.656927 Val acc:
0.595000
Epoch:  2/ 5    Batch: 75   Batch loss: 0.510195    Val loss: 0.638432 Val acc:
0.647500
Epoch:  2/ 5    Batch: 80   Batch loss: 0.696451    Val loss: 0.651840 Val acc:
0.597500
Epoch:  2/ 5    Batch: 85   Batch loss: 0.509652    Val loss: 0.681286 Val acc:
0.592500
Epoch:  2/ 5    Batch: 90   Batch loss: 0.581267    Val loss: 0.771913 Val acc:
0.552500
Epoch:  2/ 5    Batch: 95   Batch loss: 0.661655    Val loss: 0.630278 Val acc:
0.650000
Epoch:  2/ 5    Batch: 100   Batch loss: 0.585351    Val loss: 0.619743 Val acc:
0.667500
Epoch:  2/ 5    Batch: 105   Batch loss: 0.496805    Val loss: 0.747108 Val acc:
0.590000
Epoch:  2/ 5    Batch: 110   Batch loss: 0.524939    Val loss: 0.636721 Val acc:
0.655000
Epoch:  2/ 5    Batch: 115   Batch loss: 0.621672    Val loss: 0.610787 Val acc:
0.680000
Epoch:  2/ 5    Batch: 120   Batch loss: 0.434247    Val loss: 0.586571 Val acc:
0.695000
Epoch:  2/ 5    Batch: 125   Batch loss: 0.607140    Val loss: 0.586273 Val acc:
0.687500
104
Finished epoch 2. Average batch loss: 0.5507332952693105. Average validation
```

```
loss: 0.6529088149277064
Starting epoch 3
Epoch:  3/ 5    Batch: 130   Batch loss: 0.453490    Val loss: 0.679404 Val acc:
0.617500
Epoch:  3/ 5    Batch: 135   Batch loss: 0.583140    Val loss: 0.575940 Val acc:
0.717500
Epoch:  3/ 5    Batch: 140   Batch loss: 0.358122    Val loss: 0.831501 Val acc:
0.580000
Epoch:  3/ 5    Batch: 145   Batch loss: 0.411806    Val loss: 0.582739 Val acc:
0.690000
Epoch:  3/ 5    Batch: 150   Batch loss: 0.384330    Val loss: 0.655523 Val acc:
0.705000
Epoch:  3/ 5    Batch: 155   Batch loss: 0.344274    Val loss: 0.620297 Val acc:
0.695000
Epoch:  3/ 5    Batch: 160   Batch loss: 0.358667    Val loss: 0.822217 Val acc:
0.625000
Epoch:  3/ 5    Batch: 165   Batch loss: 0.310987    Val loss: 0.616756 Val acc:
0.727500
Epoch:  3/ 5    Batch: 170   Batch loss: 0.349381    Val loss: 0.570314 Val acc:
0.717500
Epoch:  3/ 5    Batch: 175   Batch loss: 0.404174    Val loss: 0.572764 Val acc:
0.732500
Epoch:  3/ 5    Batch: 180   Batch loss: 0.485828    Val loss: 0.653823 Val acc:
0.662500
Epoch:  3/ 5    Batch: 185   Batch loss: 0.266039    Val loss: 0.603032 Val acc:
0.732500
Epoch:  3/ 5    Batch: 190   Batch loss: 0.398966    Val loss: 0.558743 Val acc:
0.730000
104
Finished epoch 3. Average batch loss: 0.38209939328953624. Average validation
loss: 0.6417732508136675
Starting epoch 4
Epoch:  4/ 5    Batch: 195   Batch loss: 0.220744    Val loss: 0.575362 Val acc:
0.745000
Epoch:  4/ 5    Batch: 200   Batch loss: 0.275859    Val loss: 0.575091 Val acc:
0.737500
Epoch:  4/ 5    Batch: 205   Batch loss: 0.330835    Val loss: 0.572881 Val acc:
0.737500
Epoch:  4/ 5    Batch: 210   Batch loss: 0.106531    Val loss: 0.623701 Val acc:
0.737500
Epoch:  4/ 5    Batch: 215   Batch loss: 0.256790    Val loss: 0.640666 Val acc:
0.672500
Epoch:  4/ 5    Batch: 220   Batch loss: 0.637759    Val loss: 0.679671 Val acc:
0.735000
Epoch:  4/ 5    Batch: 225   Batch loss: 0.208216    Val loss: 0.641050 Val acc:
0.747500
Epoch:  4/ 5    Batch: 230   Batch loss: 0.265210    Val loss: 0.604293 Val acc:
0.735000
```

```
Epoch:  4/ 5    Batch: 235   Batch loss: 0.372622    Val loss: 0.641315 Val acc:
0.737500
Epoch:  4/ 5    Batch: 240   Batch loss: 0.205042    Val loss: 0.677802 Val acc:
0.670000
Epoch:  4/ 5    Batch: 245   Batch loss: 0.379012    Val loss: 0.648437 Val acc:
0.752500
Epoch:  4/ 5    Batch: 250   Batch loss: 0.270986    Val loss: 0.580526 Val acc:
0.742500
Epoch:  4/ 5    Batch: 255   Batch loss: 0.232753    Val loss: 0.571961 Val acc:
0.747500
104
Finished epoch 4. Average batch loss: 0.27838339447043836. Average validation
loss: 0.6179042008633797
Starting epoch 5
Epoch:  5/ 5    Batch: 260   Batch loss: 0.209052    Val loss: 0.580086 Val acc:
0.752500
Epoch:  5/ 5    Batch: 265   Batch loss: 0.161226    Val loss: 0.627696 Val acc:
0.730000
Epoch:  5/ 5    Batch: 270   Batch loss: 0.161622    Val loss: 0.694488 Val acc:
0.727500
Epoch:  5/ 5    Batch: 275   Batch loss: 0.195214    Val loss: 0.645984 Val acc:
0.742500
Epoch:  5/ 5    Batch: 280   Batch loss: 0.227829    Val loss: 0.684968 Val acc:
0.742500
Epoch:  5/ 5    Batch: 285   Batch loss: 0.155799    Val loss: 0.556669 Val acc:
0.765000
Epoch:  5/ 5    Batch: 290   Batch loss: 0.174857    Val loss: 0.586511 Val acc:
0.732500
Epoch:  5/ 5    Batch: 295   Batch loss: 0.158507    Val loss: 0.726699 Val acc:
0.702500
Epoch:  5/ 5    Batch: 300   Batch loss: 0.325574    Val loss: 0.714649 Val acc:
0.695000
Epoch:  5/ 5    Batch: 305   Batch loss: 0.169596    Val loss: 0.574627 Val acc:
0.755000
Epoch:  5/ 5    Batch: 310   Batch loss: 0.354525    Val loss: 0.598333 Val acc:
0.727500
Epoch:  5/ 5    Batch: 315   Batch loss: 0.179471    Val loss: 0.694602 Val acc:
0.692500
Epoch:  5/ 5    Batch: 320   Batch loss: 0.142213    Val loss: 0.628788 Val acc:
0.732500
104
Finished epoch 5. Average batch loss: 0.19638578174635768. Average validation
loss: 0.6395460776984692
```

[ ]:
```
reviews_test(embedding_model, test_loader, device=DEVICE)
```

```
Test loss: 0.558
Test accuracy: 0.770
```

2.6 2e Sandbox

[ ]:
```
# Explore different architectures and hyperparameters.
```