In [ ]:
```python
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```python
# Load in the data
from sklearn.datasets import load_breast_cancer
```

In [ ]:
```python
# load the data
data = load_breast_cancer()
```

In [ ]:
```python
# check the type of 'data'
type(data)
```

Out[ ]:
```
sklearn.utils.Bunch
```

In [ ]:
```python
# note: it is a Bunch object
# this basically acts like a dictionary where you can treat the keys like attributes
data.keys()
```

Out[ ]:
```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

In [ ]:
```python
# 'data' (the attribute) means the input data
data.data.shape
# it has 569 samples, 30 features
```

Out[ ]:
```
(569, 30)
```

In [ ]:
```python
# 'targets'
data.target
# note how the targets are just 0s and 1s
# normally, when you have K targets, they are labeled 0..K-1
```

Out[ ]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
       1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1,
       1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
       0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1,
       1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0,
       0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0,
       1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1,
       1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0,
       0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0,
       0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
       1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1,
```

```
       1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1,
       1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
       1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
       1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
       1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1])
```

In [ ]:

```
# their meaning is not lost
data.target_names
```

Out[ ]:

```
array(['malignant', 'benign'], dtype='<U9')
```

In [ ]:

```
# there are also 569 corresponding targets
data.target.shape
```

Out[ ]:

```
(569,)
```

In [ ]:

```
# you can also determine the meaning of each feature
data.feature_names
```

Out[ ]:

```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error',
       'fractal dimension error', 'worst radius', 'worst texture',
       'worst perimeter', 'worst area', 'worst smoothness',
       'worst compactness', 'worst concavity', 'worst concave points',
       'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

In [ ]:

```
# normally we would put all of our imports at the top
# but this lets us tell a story
from sklearn.model_selection import train_test_split


# split the data into train and test sets
# this lets us simulate how our model will perform in the future
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0
N, D = X_train.shape
```

In [ ]:

```
# Scale the data
# you'll learn why scaling is needed in a later course
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [ ]:

```
# Now all the fun PyTorch stuff
# Build the model
model = nn.Sequential(
```

```python
    nn.Linear(D, 1),
    nn.Sigmoid()
)
```

In [ ]:

```python
# Loss and optimizer
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters())
```

In [ ]:

```python
# Convert data into torch tensors
X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32).reshape(-1, 1))
y_test = torch.from_numpy(y_test.astype(np.float32).reshape(-1, 1))
```

In [ ]:

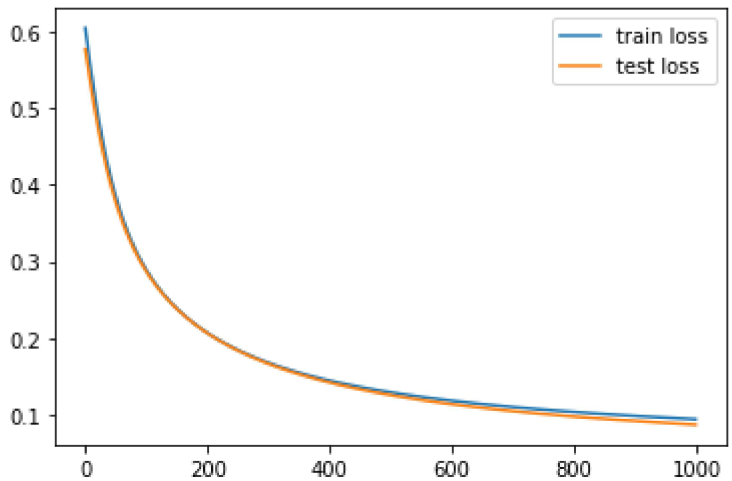
```python
# Train the model
n_epochs = 1000

# Stuff to store
train_losses = np.zeros(n_epochs)
test_losses = np.zeros(n_epochs)

for it in range(n_epochs):
  # zero the parameter gradients
  optimizer.zero_grad()

  # Forward pass
  outputs = model(X_train)
  loss = criterion(outputs, y_train)

  # Backward and optimize
  loss.backward()
  optimizer.step()

  # Get test loss
  outputs_test = model(X_test)
  loss_test = criterion(outputs_test, y_test)

  # Save losses
  train_losses[it] = loss.item()
  test_losses[it] = loss_test.item()

  if (it + 1) % 50 == 0:
    print(f'Epoch {it+1}/{n_epochs}, Train Loss: {loss.item():.4f}, Test Loss: {loss_te
```

```
Epoch 50/1000, Train Loss: 0.3874, Test Loss: 0.3792
Epoch 100/1000, Train Loss: 0.2903, Test Loss: 0.2879
Epoch 150/1000, Train Loss: 0.2395, Test Loss: 0.2387
Epoch 200/1000, Train Loss: 0.2074, Test Loss: 0.2068
Epoch 250/1000, Train Loss: 0.1849, Test Loss: 0.1840
Epoch 300/1000, Train Loss: 0.1682, Test Loss: 0.1668
Epoch 350/1000, Train Loss: 0.1553, Test Loss: 0.1533
Epoch 400/1000, Train Loss: 0.1450, Test Loss: 0.1425
Epoch 450/1000, Train Loss: 0.1366, Test Loss: 0.1335
Epoch 500/1000, Train Loss: 0.1296, Test Loss: 0.1260
Epoch 550/1000, Train Loss: 0.1237, Test Loss: 0.1197
Epoch 600/1000, Train Loss: 0.1187, Test Loss: 0.1142
Epoch 650/1000, Train Loss: 0.1143, Test Loss: 0.1094
```

```
Epoch 700/1000, Train Loss: 0.1104, Test Loss: 0.1051
Epoch 750/1000, Train Loss: 0.1070, Test Loss: 0.1014
Epoch 800/1000, Train Loss: 0.1040, Test Loss: 0.0980
Epoch 850/1000, Train Loss: 0.1013, Test Loss: 0.0950
Epoch 900/1000, Train Loss: 0.0988, Test Loss: 0.0922
Epoch 950/1000, Train Loss: 0.0966, Test Loss: 0.0897
Epoch 1000/1000, Train Loss: 0.0945, Test Loss: 0.0874
```

In [ ]:

```
# Plot the train loss and test loss per iteration
plt.plot(train_losses, label='train loss')
plt.plot(test_losses, label='test loss')
plt.legend()
plt.show()
```



In [ ]:

```
# Get accuracy
with torch.no_grad():
  p_train = model(X_train)
  p_train = np.round(p_train.numpy())
  train_acc = np.mean(y_train.numpy() == p_train)

  p_test = model(X_test)
  p_test = np.round(p_test.numpy())
  test_acc = np.mean(y_test.numpy() == p_test)
print(f"Train acc: {train_acc:.4f}, Test acc: {test_acc:.4f}")
```

```
Train acc: 0.9790, Test acc: 0.9840
```

In [ ]:

```
# Exercise: Plot the accuracy per iteration too
```

Save and Load Model

In [ ]:

```
# Look at the state dict
model.state_dict()
```

Out[ ]:

```
OrderedDict([('0.weight',
              tensor([[-0.2566, -0.4819, -0.2348, -0.5012, -0.2742, -0.2846, -0.4044, -
0.2511,
                       -0.1886,  0.3691, -0.4129, -0.1090, -0.2744, -0.4740,  0.1219,
0.2563,
                        0.1953, -0.1353,  0.1774,  0.2940, -0.3595, -0.4200, -0.3499, -
```

```
0.4715,
                        -0.5630, -0.2870, -0.4693, -0.3190, -0.3968, -0.2124]])),
             ('0.bias', tensor([0.7237]))])
```

In [ ]:
```python
# Save the model
torch.save(model.state_dict(), 'mymodel.pt')
```

In [ ]:
```python
!ls
```

```
mymodel.pt  sample_data
```

In [ ]:
```python
# Load the model
# Note: this makes more sense and is more compact when
# your model is a big class, as we will be seeing later.
model2 = nn.Sequential(
    nn.Linear(D, 1),
    nn.Sigmoid()
)
model2.load_state_dict(torch.load('mymodel.pt'))
```

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

In [ ]:
```python
# Evaluate the new model
# Results should be the same!
with torch.no_grad():
  p_train = model2(X_train)
  p_train = np.round(p_train.numpy())
  train_acc = np.mean(y_train.numpy() == p_train)

  p_test = model2(X_test)
  p_test = np.round(p_test.numpy())
  test_acc = np.mean(y_test.numpy() == p_test)
print(f"Train acc: {train_acc:.4f}, Test acc: {test_acc:.4f}")
```

```
Train acc: 0.9816, Test acc: 0.9628
```

In [ ]:
```python
# Download the model
from google.colab import files
files.download('mymodel.pt')
```