BCE Loss with Logits

Why? Numerical instability. Exponentiating things leads to very large numbers (larger than a computer can represent).

In [ ]:

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:

```
# Load in the data
from sklearn.datasets import load_breast_cancer
```

In [ ]:

```
# load the data
data = load_breast_cancer()
```

In [ ]:

```
# check the type of 'data'
type(data)
```

Out[ ]:

```
sklearn.utils.Bunch
```

In [ ]:

```
# 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 [ ]:

```
# 'data' (the attribute) means the input data
data.data.shape
# it has 569 samples, 30 features
```

Out[ ]:

```
(569, 30)
```

In [ ]:

```
# '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 [ ]:

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

Out[ ]:

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

In [ ]:

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

Out[ ]:

```
(569,)
```

In [ ]:

```python
# 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 [ ]:

```python
# 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 [ ]:

```python
# 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.Linear(D, 1)
```

In [ ]:
```
# Loss and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters())
```

In [ ]:
```
# 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))
```

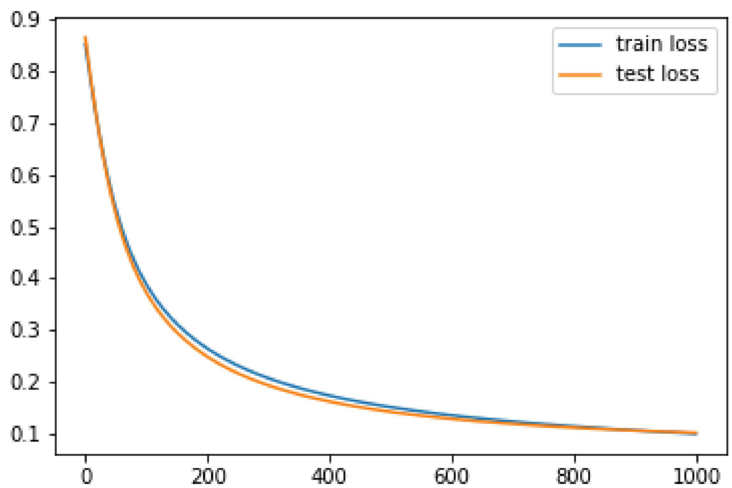
In [ ]:
```
# Train the model
n_epochs = 1000

# Stuff to store
train_losses = np.zeros(n_epochs)
test_losses = np.zeros(n_epochs)
train_acc = np.zeros(n_epochs)
test_acc = 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: 0.5387, Test Loss: 0.5280
Epoch 100/1000, Train Loss: 0.3902, Test Loss: 0.3742
Epoch 150/1000, Train Loss: 0.3129, Test Loss: 0.2965
Epoch 200/1000, Train Loss: 0.2648, Test Loss: 0.2490
Epoch 250/1000, Train Loss: 0.2316, Test Loss: 0.2168
Epoch 300/1000, Train Loss: 0.2072, Test Loss: 0.1934
```

```
Epoch 350/1000, Train Loss: 0.1884, Test Loss: 0.1758
Epoch 400/1000, Train Loss: 0.1735, Test Loss: 0.1620
Epoch 450/1000, Train Loss: 0.1614, Test Loss: 0.1511
Epoch 500/1000, Train Loss: 0.1513, Test Loss: 0.1421
Epoch 550/1000, Train Loss: 0.1427, Test Loss: 0.1347
Epoch 600/1000, Train Loss: 0.1354, Test Loss: 0.1285
Epoch 650/1000, Train Loss: 0.1290, Test Loss: 0.1233
Epoch 700/1000, Train Loss: 0.1234, Test Loss: 0.1188
Epoch 750/1000, Train Loss: 0.1184, Test Loss: 0.1149
Epoch 800/1000, Train Loss: 0.1139, Test Loss: 0.1115
Epoch 850/1000, Train Loss: 0.1099, Test Loss: 0.1085
Epoch 900/1000, Train Loss: 0.1063, Test Loss: 0.1059
Epoch 950/1000, Train Loss: 0.1029, Test Loss: 0.1036
Epoch 1000/1000, Train Loss: 0.0999, Test Loss: 0.1016
```

In [ ]:

```python
# 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 [ ]:

```python
# Get accuracy
with torch.no_grad():
  p_train = model(X_train)
  p_train = (p_train.numpy() > 0)
  train_acc = np.mean(y_train.numpy() == p_train)

  p_test = model(X_test)
  p_test = (p_test.numpy() > 0)
  test_acc = np.mean(y_test.numpy() == p_test)
print(f"Train acc: {train_acc:.4f}, Test acc: {test_acc:.4f}")
```

```
Train acc: 0.9843, Test acc: 0.9840
```