Discovering activation functions

Activation functions add non-linearity to the network

1. Sigmoid for binary classification
2. Softmax for multi-class classification

A network can learn more complex relationships with non-linearity

```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[6]])
sigmoid = nn.Sigmoid()
output = sigmoid(input_tensor)
print(output) # now bounded in [0,1]

model = nn.Sequential(
    nn.Linear(6,4) # linear layer 1
    nn.Linear(4,1) # linear layer 2
    nn.Sigmoid() # activation layer 1
)
```

Softmax takes three-dimensional as input and outputs the same shape
Outputs a probability distribution in [0,1]

```
input_tensor = torch.tensor([[4.3, 6.1, 2.3]])

probabilities = nn.Softmax(dim=-1) # dim=-1, softmax applied to input's last
dim
output_tneosr = probabilities(input_tensor)
print(output_tensor) # bounded
```

This acts similarly to a scaler because values are bounded between 0 and 1
This is different than a StandardScaler, because the probabilities output to 1

Running a forward pass

Running a forward pass through a network generates a prediction/output

Calculations are performed at each layer and outputs are produced based on weights and biases

A threshold of 0.5 is commonly used for classification

Using loss functions to assess model predictions

See how good our predictions are compared to the actual values

The loss function tells us how good our model is at making predictions during training

loss = F(y, y_hat)

y is a single integer (class label), and y_hat is a tensor (prediction before softmax)

```
import torch.nn.functional as F
print(F.one_hot(torch.tensor(0), num_classes=3) # tensor([1,0,0])
print(F.one_hot(torch.tensor(1), num_classes=3) # tensor([0,1,0])
print(F.one_hot(torch.tensor(2), num_classes=3) # tensor([0,0,1])
```

```
from torch.nn import CrossEntropyLoss
scores = torch.tensor([-5.2, 4.6, 0.8])
one_hot_target = torch.tensor([1,0,0])

criterion = CrossEntropyLoss()

print(criterion(scores.double(), one_hot_target.double()))
# scores - model predictions before softmax, outputs a single float (loss)
# one_hot_target - one hot encoded ground truth label
# tensor(9.82222, d.type=torch.float64)
```

Using derivatives to update model parameters

We can use derivatives to minimize loss

Steep slopes indicate high derivatives and large steps

We want to find the loss function's global minimum, where the derivative is zero

Derivatives = gradients

In backpropagation, we begin with loss gradients at L2 and work backwards

```
# forward pass
model = nn.Sequential(
    nn.Linear(16,8)
    nn.Linear(8,4)
    nn.Linear(4,2)
)
prediction = model(sample)

# calculate the loss and gradeints
criterion = CrossEntropyLoss()
```

```python
loss = criterion(prediction, target)
loss.backward()

# Access each layer's gradients
model[0].weight.grad
model[0].bias.grad
model[1].weight.grad
model[1].bias.grad
model[2].weight.grad
model[2].bias.grad

lr = 0.001 # learning rate is typically small

# update weights and biases

weight = model[0].weight
weight_grad = model[0].weight.grad
weight = weight - lr*weight_grad

bias = model[0].bias
bias_grad = model[0].bias.grad
bias = bias - lr*bias_grad

# we can find gradient descent with SGD
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=0.001)
optimizer.step() # updates params
```