Applied Deep Learning with Tensorflow and Pytorch, Chapter 4

MLP and Activation Functions

Multilayer Perceptron (MLP) adds one or multiple fully-connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.

- **Activation functions** decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias.
- Activation functions are **differentiable** operators to transform input signals to outputs, while most of them **add non-linearity**.

ReLU Function: $\text{ReLU}(x) = \max(x, 0)$**.**

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)

y.backward(torch.ones_like(x), retain_graph=True)
```

```
x = tf.Variable(tf.range(-8.0, 8.0, 0.1), dtype=tf.float32)
y = tf.nn.relu(x)

with tf.GradientTape() as t:
    y = tf.nn.relu(x)
```

Sigmoid Function: $\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$.

```
y = torch.sigmoid(x)
```

```
y = tf.nn.sigmoid(x)
```

Tanh Function: $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$.

```
y = torch.tanh(x)
```

```
y = tf.nn.tanh(x)
```

TensorFlow Implementation of MLP from Scratch

1: Import libraries

```
import tensorflow as tf
```

2: Initializing the Model Parameters

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = tf.Variable(
    tf.random.normal(shape=(num_inputs, num_hiddens), mean=0, stddev=0.01))
b1 = tf.Variable(tf.zeros(num_hiddens))
W2 = tf.Variable(
    tf.random.normal(shape=(num_hiddens, num_outputs), mean=0, stddev=0.01))
b2 = tf.Variable(tf.random.normal([num_outputs], stddev=.01))

params = [W1, b1, W2, b2]
```

3: Activation Function

```
def relu(X):
    return tf.math.maximum(X, 0)
```

4: Define Model

```
def net(X):
    X = tf.reshape(X, (-1, num_inputs))
    H = relu(tf.matmul(X, W1) + b1)
    return tf.matmul(H, W2) + b2
```

5: Loss Function

```
def loss(y_hat, y):
    return tf.losses.sparse_categorical_crossentropy(y, y_hat,
                                                     from_logits=True)
```

6: Training

```
num_epochs, lr = 10, 0.1
def updater(batch_size):
    return sgd([W, b], lr, batch_size)

def train(net, train_iter, test_iter, loss, num_epochs, updater):
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

PyTorch Implementation of Weight Decay from Scratch

1: Import library and Initialize the Parameters

```
import torch
from torch import nn
import torch.nn.functional as F

def init_params():
    w = torch.normal(0, 1, size=(num_inputs, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return [w, b]
```

2: Define L_2 Norm Penalty

```
def l2_penalty(w):
    return torch.sum(w.pow(2)) / 2
```

Note that by adding regularization term to the loss function, with the effect of shrinking the parameter estimates, making the model simpler and less likely to overfit.

3: Define Training Loop

```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: nn.Linear(X, w, b), F.mse_loss
    num_epochs, lr = 100, 0.003

    for epoch in range(num_epochs):
        for X, y in train_iter:
            l = loss(net(X), y) + lambd * l2_penalty(w)
            l.sum().backward()
            torch.optim.SGD([w, b], lr, batch_size)
    print('L2 norm of w:', torch.norm(w).item())
```

- A validation set can be used for model selection, provided that it is not used too liberally.
- Underfitting means that a model is not able to reduce the training error.
- When training error is much lower than validation error, there is overfitting.
- Regularization is a common method for dealing with overfitting. It adds a penalty term to the loss function on the training set to reduce the complexity of the learned model.
- Dropout is another technique to avoid overfitting. Dropout replaces an activation h with a random variable with expected value h.
- Dropout is only used during training.

Dropout Implementation

```
import torch
from torch import nn

def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1:
        return torch.zeros_like(X)
    if dropout == 0:
        return X
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```