Notes on MIT Introduction to Deep Learning

Tianzong Cheng

December 15, 2023

Preface

Thanks to Professor Ban for introducing this lecture to me.

Tianzong Cheng

December 15, 2023

Contents

1 Lecture 1

1.1 Perceptron

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} x_i w_i) \tag{1}$$

x_i is the input, w_i is the weight of each input, w_0 is the bias term, and g is a non-linear activation funcion.

Or we can rewrite the equation in linear algebra language.

$$\hat{y} = g(w_0 + X^T W) \tag{2}$$

where: $X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

An example of the activation function: sigmoid function. It can be intepreted as a continous version of the threshold function.

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

In TensorFlow, we can access this function by:

```
tf.math.sigmoid(z)
```

1.2 Building Neural Networks with Perceptrons

Perceptron: dot product, bias, non-linearity.

Because all inputs are densly connected to all outputs, these layers are called **Dense** layers.

The first layer, which is a hidden layer can be expressed as follows:

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \tag{4}$$

Then, the second layer, which calculates the final outputs, can be expressed as follows:

$$\hat{y}_i = g(w_{0,i}^{(2)} + \sum_{j=1}^{m} g(z_j) w_{j,i}^{(2)}) \tag{5}$$

We can express the whole model using two lines of codes with the help of TensorFlow:

```
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

By stacking these layers on top of each other, we create a sequential model.

1.3 Loss Functions and Gradient Descent

The **empirical loss** measures the total loss over our entire dataset. Empirical loss is also known as objective function, cost function and emprical risk.

$$J(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; \boldsymbol{W}), y^{(i)}) \tag{6}$$

In this equation, J is a function of the weights of the neuron network. We want to find the network weights that **achieve the lowest loss**.

First, we randomly pick an initial (w_0, w_1). Then, by taking a small step in the opposite direction of gradient at this point, we can get closer to where the loss is the lowest. Repeat this step until convergence.

```
import tensorflow as tf
weights = tf.Variable([tf.random.normal()])
```

```
while True:
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)
    weights = weights - lr * gradient
```

1.4 Backpropagation

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1} \tag{7}$$

Repeat this for **every weight in the network** using gradients from later layers. We can see backpropagation is simply an instantiation of the chain rule.

1.5 Bathced Gradient Descent

Compute the gradient of a batch of (usually around a hundred) data points, rather than a single data point or all the data points. This is a fast and accurate way of estimating the gradient.

1.6 Regularization

The problem of overfitting describes that the model is too complex and does not generalize well.

Regularization is a technique that constrains our optimization problem to discourage complex models. It helps improving generalization of our model on unseen data.

The idea of **dropout** is randomly selecting some (typically half) activations to 0. The idea of **early stopping** is to stop training before we have a chance to overfit.