APS360—Intro to Machine Learning

Jonah Chen

October 4, 2021

Contents

1 Data Splits

Attempt to split the data in a way that is representative to the real purpose of your model.

- Training Data: Optimizing model parameters.
- Validation Data: Optimizing hyperparameters.
- Test Data: Final "real world" testing.

2 Artificial Neural Networks

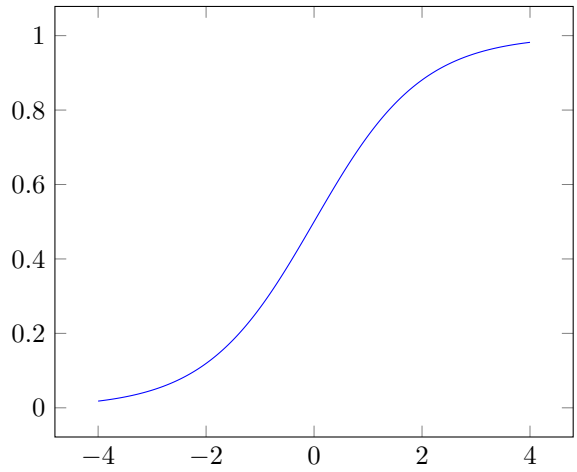
Conventially, an artificial neuron consists of

- $\mathbf{x}$ is an input vector
- $\mathbf{w}$ is a weight vector
- b is a bias scalar
- f is the activation function
- y is the output (scalar)

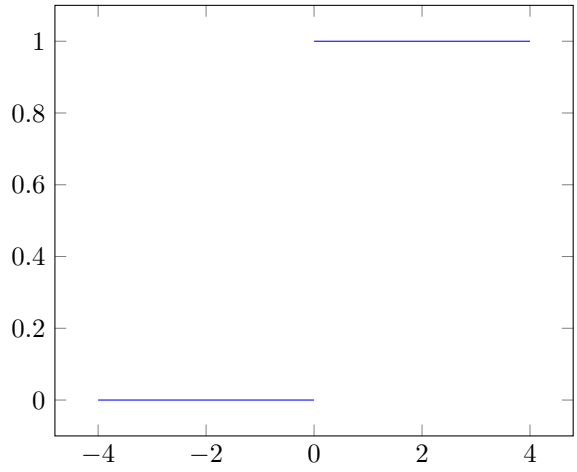
2.1 Activation Functions

- A common activation function is the *sigmoid* function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$



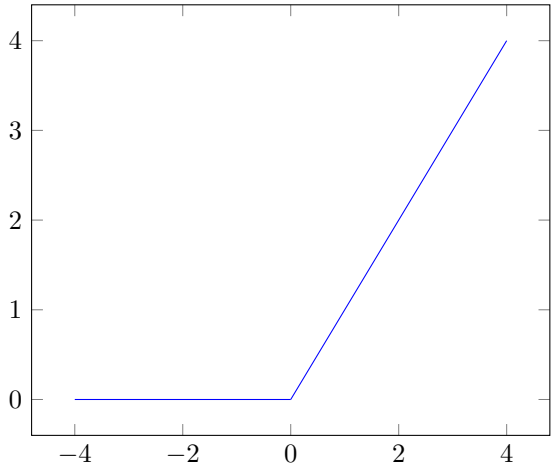
- The first artificial neurons uses a step function. This models more accurately the biological neuron but staganates progress since the function is not differentiable.



- Another activation function is the *tanh* function, which has similar behavior to the sigmoid but ranges from $(-1, 1)$.
- The sigmoid and tanh activation fucntions have vanishing gradients as $x \to \pm\infty$. To solve this, we use the *Rectified Linear Unit*(ReLU) activation function

$$f(x) = \max(0, x) \tag{2.2}$$

with very easy derivative $f'(x) = x > 0?1 : 0$



2.2 Loss Functions

In *maximum likelihood estimations*, the mean square error comes from a gaussian distribution whereas cross entropy correlates from a beronulli distribution.

- The mean square error, useful for **regression problems**.

$$C = \frac{1}{N}|\mathbf{y} - \mathbf{t}|^2 \tag{2.3}$$

- The cross entropy loss, useful for **classification problems**.

$$C = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\log(y_{n,k}) \tag{2.4}$$

2.3 Training

- Given the ground truth, We can use a loss function to evaluate the model.
- Firstly we start with the forward pass:

```
import math

#data
x = [[1,...]]
t = [0,0,0,1]
w = [1,-1,1]
def simple_ann(x, w, t):
    y = []
    for n in range(len(x)):
        v = 0
        for p in range(len(x[0])):
            v = v + x[n][p]*w[p]
        y.append(1 / (1+math.exp(-v)))
    return y
```

- We can represent a neural network as layers of multiple neurons. Where each neuron's weight vector is a **row** of the weight matrix $\mathbf{W}$ and the input is a column vector $\mathbf{x}$.

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.5}$$

- We will take the gradient of the loss function with respect to the weights. The gradient gives the direction of greatest ascent. To minimize the loss, we will change adjust the weights to the opposite direction of the gradient.

$$w_{ji}^{t+1} = w_{ji}^{t} - \gamma\frac{\partial E}{\partial w_{ji}} \tag{2.6}$$

  where γ is the learning rate.

- Learning rate is a hyperparameter that you can tune. Low learning rate causes the model to be stuck at a local minimum or saddle point. High learning rate will cause the gradient to explode.
- Since neural networks are very high-dimensional, it is very uncommon to have a convex local minimum. Saddle points are far more common.

```
learning = 10
def simple_ann(x,w,t,iterations,learning):
    E = [] #error
    for ii in range(iterations):
        err,y = [],[]
        for n in range(len(x)):
            v = 0
            for p in range(len(x[0])):
                v = v + x[n][p]*w[p]
            y.append(1 / (1 + math.exp(-v)))
```

```
            err.append((y[n]-t[n])**2)

            #gradient descent to compute new weights
            for p in range(len(w)):
                d = x[n][p]*(y[n]-t[n])*(1-y[n])*(y[n])
                w[p] = w[p] - learning*d
        E.append(2*sum(err)/len(x))
    return (y,w,E)
```

- A single neuron (or layer) can solve all problems that are linearly seperable. A neural network with one hidden layer can fit any functions (*Universal Approximation Theorem*).
- Credit Assignment Problem: Naively computing gradients require exponential computation. This can be solved by dynamic programming, where complexity grows **linearly** with depth, and **quadratically** with the width.

3 Convolutional Neural Networks

- If we are given a 200×200 image, with a two hidden layers of 500 and 200 neurons. Then, a fully connected model will be have a billion parameters.
- Fully connected layers has inductive bias, as it cannot process spacial data.
- Convolutional filters are used in signal processing. To convolve a kernel K with image I
- There are hand-designed kernals used for classical computer vision. For example, the kernal

$$K = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{3.1}$$

  produces a blur for the image

$$K = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{3.2}$$

  is a vertical edge detector. K^T is a horizontal edge detector.
- The idea of CNNs are not to hand-craft these kernals, but to learn them from data. This has several benefits against fully connected networks:
  - Locally connected layers: look for local features in the small regions of the image
  - Weight sharing: detects the same local features across the entire image
- For square images, $o = (i - k + 2p)/s + 1$. Where o is the output size, i is the input size, p is the padding, and s is the stride.
- In a fully connected neural network, we reduced the number of units before the final layer. This is to consolidate information, and only keep the most important information for classification. In CNN, we use strided convolutions and pooling to accomplish the same task.
- Pooling is deterministic.
- In general, if you go deeper, you increase the number of kernals, but you decrease the height and width of the features map.