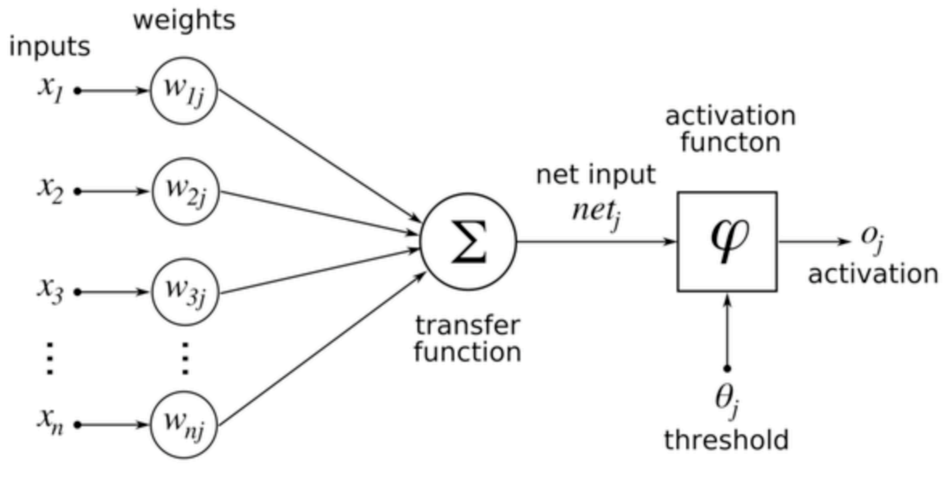
Xaiver

June 16, 2018

1 Xaivier

The motivation for Xavier initialization in Neural Networks is to initialize the weights of the network so that the neuron activation functions are not starting out in saturated or dead regions. In other words, we want to initialize the weights with random values that are not "too small" and not "too large".

Take a single neuron j that takes a set of inputs $X_1, ..., X_n$ and creates an output, shown schematically here:

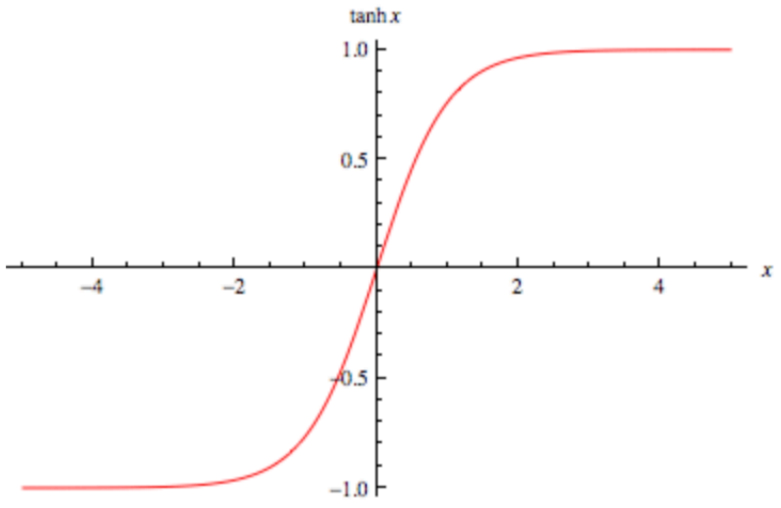


Note that the transfer function is just the sum of the products of the weights with each of the inputs and so we can write the input to neuron j as:

$$net_j = w_{1j}x_1 + w_{1j}x_2 + ... + w_{nj}x_n$$

So the output of the transfer function is a single scalar (net_j, just a number) and the activation function acts on this number to produce another number (this gives us the nonlinearity in the network).

The types of activation functions that the original Glorot and Bengio, known as the Glorot initialization function; was looking at were symmetric type functions such as the hyperbolic tangent function tanh(net_j).



This function looks like the above, where the horizontal axis (input to the hyperbolic tangent activation function) corresponds to net_j. Here you can see that if net_j is very large or very small (negative), then the hyperbolic tangent activation is saturated/stuck at +1.0 or -1.0 respectively. Having saturated neurons limits their dynamic range and so limits their reprentational power.

How do we avoid getting stuck at these saturdated regions? Recall that the activation function $net_j = net_j = w_{1j}x_1 + w_{1j}x_2 + ... + w_{nj}x_n$ depends on the weights w_j and the input X. To avoid net_j being too large or too small, then it makes sense to keep the weights w_j and the input x in some sensible range. **We can restrict X, which comes from our data, by normalizing our data using z-scaling or other methods** (to ensure that the data has zero mean and unit variance).

But what about the weights W_j? Technically, we are free to select the weights to whatever we want and then use a learning rule such as stochastic gradient descent to adjust them to minimize error. However, if we initialize them to be 'too large', (or specifically, in such a way that causes the input neuron net_j to be saturated) then it will take gradient descent more iterations to adjust the weights.

This is where *Xavier Initialization* suggests initializing the weights W_j with a variance so that the variance of net_j, $Var(net_j)$, is unity. By ensuring that $Var(net_j)$ is unity, we can reduce the likelihood of being stuck in saturated regions of the hyperbolic tangent. As this blog post explains, based on the assumptions that our input data x has zero mean and unit variance (reasonable assumption since the data is z-scaled), and also that the input x is statistically independent (in general). then picking IID (independently and identically distributed) random weights with variance inversely proportional to the number of inputs, will ensure that $Var(net_j)$ = 1.0.

In other words, for the schematic example that we have shown, we compute the variance as:

$$Var(net_j) = w_{1j}x_1 + w_{1j}x_2 + ... + w_{nj}x_n$$

which simplifies, based on our input data's unit variance, zero mean, and independence assumption, to:

$$Var(net_j) = nVar(W_{ji})$$

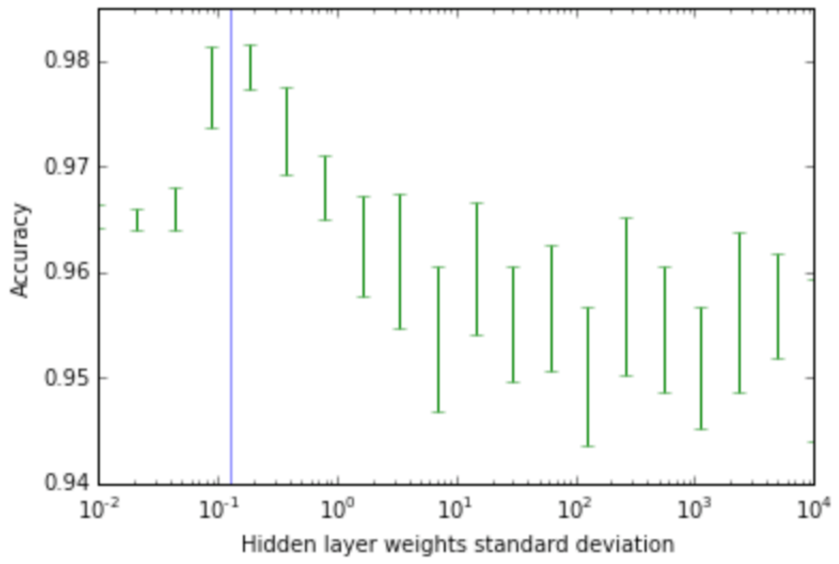
valid for all $i \in 1, \ldots, n$. Therefore, we initialize the weights from an IDD normal so that:

$$w_{ji}N(0, \frac{1}{n})$$

This is the Xaiver initialization function.

Here we consider a fully connected multilayer perceptron with only 1 hidden unit and a softmax output layer. For illustration purposes, we will initialize the input layer weights to random values and keep them fixed and only update the output layer weights (to simplify code, we'll just use logistic regression from Scikit-learn on the hidden layer).

The dataset that we use is the digits dataset provided by Scikit-learn. These are 1,797 examples of 8x8 images (so we have $n = 64$ inputs to each neuron) showing the different digits between 0 and 9. Our neural network will have a hidden layer with 500 neurons. We consider several different variance initializations for the hidden layer weights and for each variance initialization we run the simulation 10 times. Below shows how the accuracy on a test set 33% of the data based on different standard deviations (square root of variance) used for initializing the weights:



The vertical blue line corresponds to picking an Xavier initialization, so that the variance of the hidden weights are $\frac{1}{64}$. Each error bar is the result of running the simulation 10 times using the same variance for this hidden weight initialization. This figure suggests that certain initializations are better than others. Keep in mind that in this experiment we are keeping the initialization of the hidden layer weights fixed at random but only vary their variance. Although it's certainly possible that to update the hidden weights with gradient descent that we can also gain high accuracies, the Xavier initialization starts us off in a better region and we in effect have to do less work (that is, less gradient descent iterations).

```
import numpy as np
from sklearn import datasets
```

```
    from sklearn.cross_validation import train_test_split
    from sklearn.linear_model import LogisticRegression
    from sklearn.preprocessing import StandardScaler
    import matplotlib.pyplot as plt

# load digits dataset
digits = datasets.load_digits()
X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

# z-sclae data
sc = StandardScaler()
X_train_s = sc.fit_transform(X_train)
X_test_s = sc.transform(X_test)

## generate a range of different initializations with different variances

sds = np.logspace(-2, 4, 20)

mean_scores = []
sd_scores = []
num_inputs = X_train.shape[1]
num_units = 500
clf =  LogisticRegression(C = 10)
for sd in sds:
    scores = []
    for i in range(10):
        # generate random inner weight matrix scaled by sd
        weights = sd*np.random.randn(num_inputs, num_units)

    # hidden layer computation
        Z_train = np.tanh(np.dot(X_train_s, weights))
        Z_test = np.tanh(np.dot(X_test_s, weights))

        clf.fit(Z_train, y_train)
        scores.append(clf.score(Z_test, y_test))
    mean_scores.append(np.mean(scores))
    sd_scores.append(np.sqrt(np.var(scores)))

plt.semilogx(sds.T, np.array(mean_scores), linestyle='None')
plt.errorbar(sds.T, np.array(mean_scores), np.array(sd_scores), fmt="none", color='b')
plt.ylabel('Accuracy')
plt.xlabel('Hidden layer weights standard deviation')
plt.axvline(x=np.sqrt(1)/np.sqrt(64), alpha = .5)
plt.show()
```