

# Machine Learning for Textual and Unstructured Data

## Pre-Sessional Lecture II: Neural Networks

Stephen Hansen  
University College London

# Introduction

Neural network models have driven many of the important breakthroughs in machine learning and AI over the last ten years.

They were already well-known and popular in the 1990s.

Several interrelated drivers of their recent growth:

1. Availability of large datasets for training
2. Hardware advances: GPU/TPU, cloud infrastructure
3. Software advances: high-level frameworks for model estimation that hide many details from the user
4. Algorithmic advances: efficient gradient computation

In this lecture, we describe the structure of the simplest class of neural networks: feed-forward neural networks.

# Basic Architecture of a Neural Network

# A neuron

As usual, suppose that each data observation  $i$  has  $J$  associated covariates which we stack in  $\mathbf{x}_i$ .

As in linear regression, we can linearly combine covariates and add a constant:

$$z_i = \theta_0 + \sum_{j=1}^J \theta_j x_{i,j}.$$

In neural network jargon, the  $\{\theta_j\}_{j=1}^J$  terms are sometimes called *weights* and  $\theta_0$  is called a *bias*.

We can transform the linear combination via an *activation function*:

$$y = \phi(z_i) = f(\mathbf{x}_i \mid \boldsymbol{\theta})$$

# Activation Functions

Traditionally:

1. A sigmoid function (logistic regression):

$$\phi(z_i) = \frac{1}{1 + e^{-z_i}}$$

2. A particular limiting case: step function.
3. Hyperbolic tangent.
4. Identity function (linear regression):  $\phi(z_i) = z_i$ .

Other activation functions have gained popularity recently:

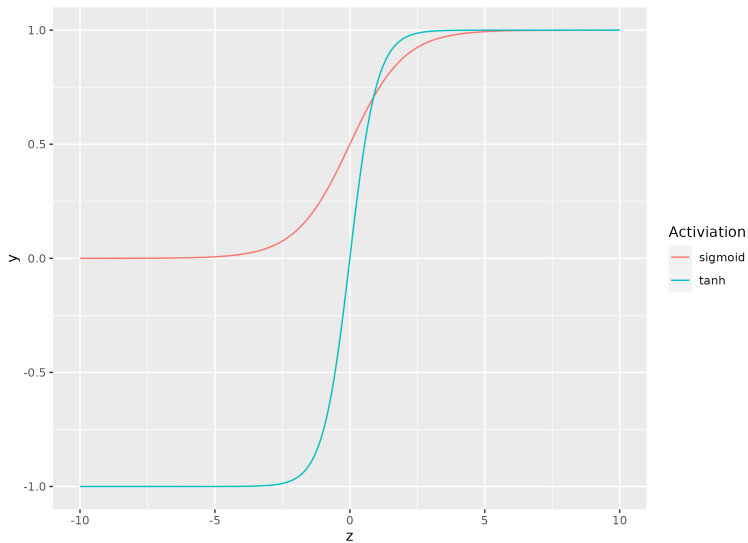
1. Rectified linear unit (ReLU):

$$\phi(z_i) = \max(0, z_i)$$

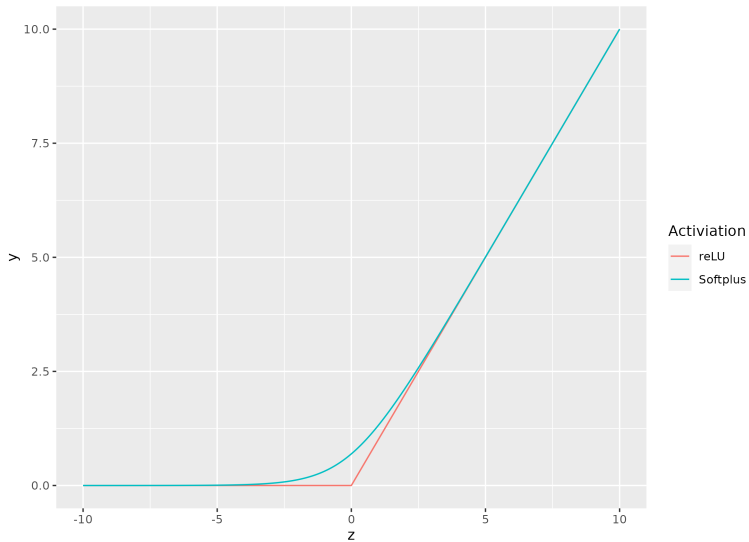
2. Softplus:

$$\phi(z_i) = \log(1 + e^{z_i})$$

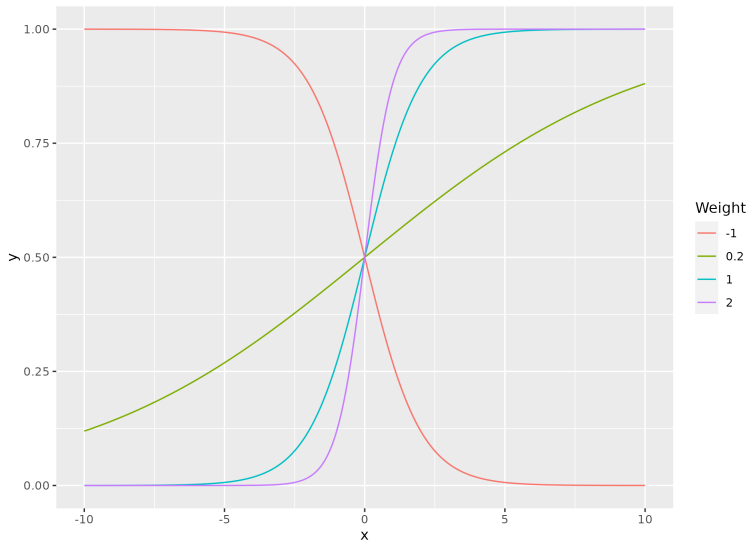
# Activation Function Plots I



# Activation Function Plots II

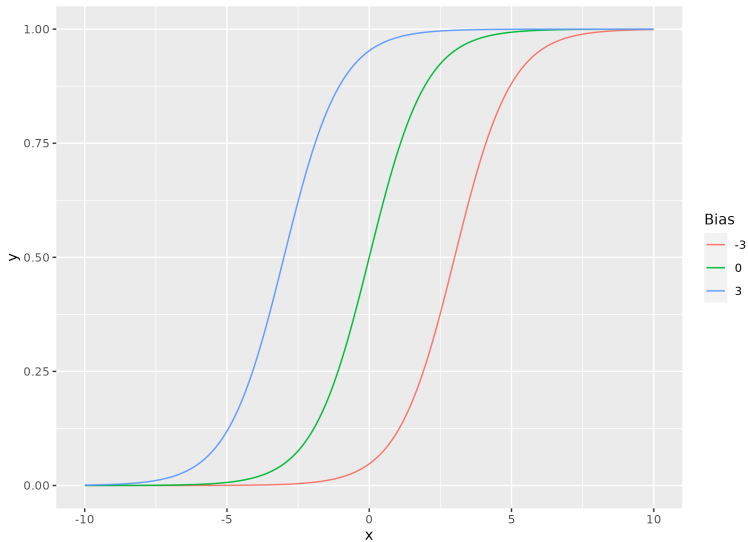


# Effect of Varying Weight (Sigmoid)

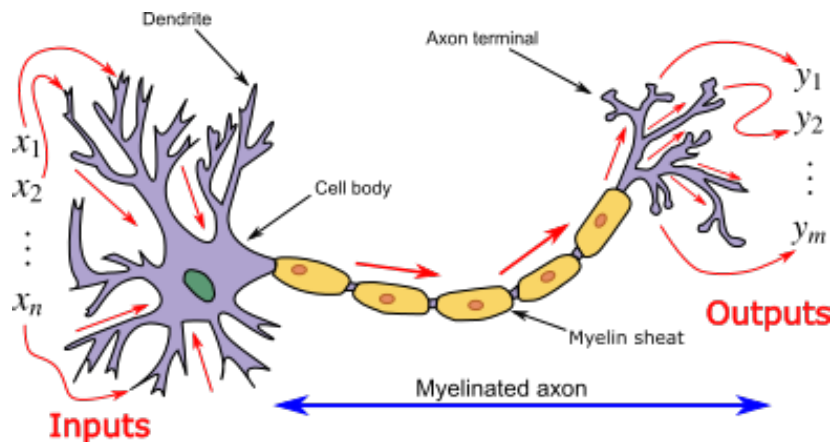




# Effect of Varying Bias (Sigmoid)



# The Biological Analogue



# Combining Neurons into a Neural Network

Now suppose we have  $K$  separate activation functions operating over the same input  $\mathbf{x}_i$ .

We first obtain  $K$  separate linear combinations of covariates

$$z_{i,k} = \theta_{0,k} + \sum_{j=1}^J \theta_{j,k} x_{i,j} \quad \forall k = 1, \dots, K$$

Next we linearly combine each separate neuron to create a final output

$$f(\mathbf{x}_i | \boldsymbol{\theta}) = \theta_0 + \sum_k \theta_k \phi(z_{i,k}).$$

This is known as a single layer network.

Consists of  $K$  non-linear transformations of linear combinations of input variables.

# Classic Result I

## Result

*Universal approximation theorem: [Hornik et al., 1989] A neural network with at least one hidden layer can approximate any Borel measurable function mapping finite-dimensional spaces to any desired degree of accuracy.*

# Classic Result II

## Result

*Breaking the curse of dimensionality: [Barron, 1993] A one-layer NN achieves integrated square errors of order  $\mathcal{O}(1/K)$ , where  $K$  is the number of nodes. In comparison, for series approximations, the integrated square error is of order  $\mathcal{O}(1/(B^{2/J}))$  where  $J$  is the dimension of the function to be approximated and  $B$  is the number of basis functions.*

# Adding Layers

Additional layers can easily be added as follows:

$$z_{i,k}^{(1)} = \theta_{0,k}^{(1)} + \sum_{j=1}^J \theta_{j,k}^{(1)} x_{i,j} \quad (\text{Input Layer})$$

$$z_{i,l}^{(2)} = \theta_{0,l}^{(2)} + \sum_{k=1}^{K_1} \theta_{l,k}^{(2)} \phi \left( z_{i,k}^{(1)} \right) \quad (\text{Hidden Layer 1})$$

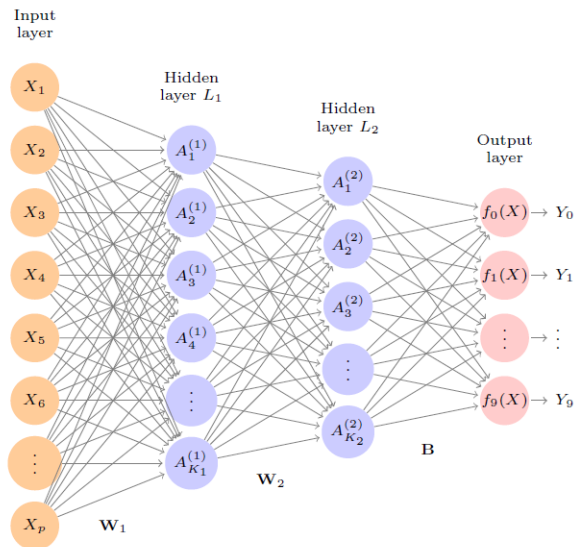
$$z_{i,l}^{(3)} = \theta_{0,l}^{(3)} + \sum_{k=1}^{K_2} \theta_{l,k}^{(3)} \phi \left( z_{i,k}^{(2)} \right) \quad (\text{Hidden Layer 2})$$

...

$$f(\mathbf{x}_i | \boldsymbol{\theta}) = \theta_{0,m}^{(L)} + \sum_{k=1}^{K_{L-1}} \theta_{m,k}^{(L)} \phi \left( z_{i,k}^{(L-1)} \right) \quad (\text{Output Layer})$$

Overall structure is called a *feed-forward neural network*.

# Graphical Representation



# Feed-Forward Networks

Multiple layers allow for complex downstream structures to develop by combining simpler upstream structures.

Typical to have  $K_l < K_{l-1}$ , e.g. fewer units as we move along network.

Output layer can have multiple dimensions or represent class probabilities by specifying

$$z_{i,m}^{(L)} = \theta_{0,m}^{(L)} + \sum_{k=1}^{K_{L-1}} \theta_{m,k}^{(L)} \phi(z_{i,k}^{(L-1)}) \rightarrow f_m(\mathbf{x}_i | \boldsymbol{\theta}) = \frac{\exp(z_{i,m}^{(L)})}{\sum_{m=1}^M \exp(z_{i,m}^{(L)})}$$

where  $M$  is the number of classes to predict.



# Estimation

# Objective Function

The first step in estimating a neural network is to define an objective (loss) function to target of the form  $L(\theta) = \frac{1}{N} \sum_{i=1}^N L'(y_i, f(\mathbf{x}_i | \theta))$ .

Typical choices:

1. Continuous  $y_i$ :  $-(y_i - f(\mathbf{x}_i | \theta))^2$
2. Categorical  $y_i$ :  $-\sum_{m=1}^M y_i f_m(\mathbf{x}_i | \theta)$

Idea of gradient descent algorithm:

1. Randomly initialize  $\theta = \theta^0$ .
2. For all  $i = 1, \dots, l$ : update  $\theta^i = \theta^{i-1} - r \nabla L(\theta^{i-1})$ .

$r$  is the *learning rate* and is set to small number.

# Stochastic Gradient Descent

For large datasets, computing the gradient over all data points is expensive.

Stochastic gradient descent forms an unbiased estimate of the gradient by randomly sampling a data point.

Mini-batch stochastic gradient descent instead uses a random draw of  $N_B$  data points where  $N_B$  is the *batch size*.

In practice, data is often randomly shuffled once and then cycled through  $N_B$ -sized-chunk-by- $N_B$ -sized-chunk.

An *epoch* is a single pass through all of the training data.

# Gradient Computation

Analytically deriving the gradient is in most cases straightforward and involves repeated application of the chain rule.

Efficiently evaluating the gradient on a computer requires some care and is typically done via *backpropagation* which avoids repeated evaluation of the same expression.

The combination of efficient gradient computation with modern hardware (GPU/TPU) allows for the estimation of large models.

# Regularization

As with other machine learning algorithms, overfitting can be controlled via regularization.

The first approach is to add a ridge (or LASSO) penalty to the loss function which adds minimal computational overhead.

The second is *dropout* whereby a random fraction of nodes is left out of the network when it is being trained.

There is also a large number of parameters to tune in a network which can contribute substantially to performance.

## Conclusion

# Conclusion

Neural networks, in particular those with many layers, have generated important recent advances in machine learning.

They reach their full potential when trained on enormous datasets and powerful modern hardware.

Not clear that these conditions hold in many applied problems relevant for economics and social science.

Neural networks also require more fine-tuning than more “off-the-shelf” regression models.

One domain in which they are clearly valuable is modeling unstructured data, which we cover next.

# References I

Barron, A. (1993).

Universal approximation bounds for superpositions of a sigmoidal function.

IEEE Transactions on Information Theory, 39(3):930–945.

Hornik, K., Stinchcombe, M., and White, H. (1989).

Multilayer feedforward networks are universal approximators.

Neural Networks, 2(5):359–366.