Time Series Analysis

ECE 6563, Electrical and Computer Engineering
University of New Brunswick

Recap

In the last section, we continued with another common state-space model; the Hidden Markov Model

- A sequence of latent states, which each generate some observable output
  The observations can be deterministic, discrete, or continuous (based on a pdf)
- State progression is governed by the probabilities in a state transition matrix
  Observations depend only on the current state
- HMMs can be used for time series modeling, or for classification of temporal sequences
  We focused on the classification task
- Three main “problems”
  Evaluation (forward, backward), Decoding (Viterbi), Learning (Baum-Welch)
- The models can be constrained for more efficient learning or to better represent known structure/semantics
  Left-right vs ergodic models, hierarchical HMMs

In this section, we’ll jump forward to look at deep learning models

- Regression & Gradient Descent
- RNN, LSTM, maybe CNN (as time allows)

Deep Learning

As we saw in the HMM tutorial, the extraction of meaningful features is a critical aspects of conventional machine learning approaches.

- “Hand crafting” features requires domain knowledge

The advent of deep learning has changed the way the field approaches these problem

- Deep learning uses nested neural networks to automatically learn what aspects of the data are important to emphasize
- Draws insights directly from raw data
- The field is exploding and changing many other fields

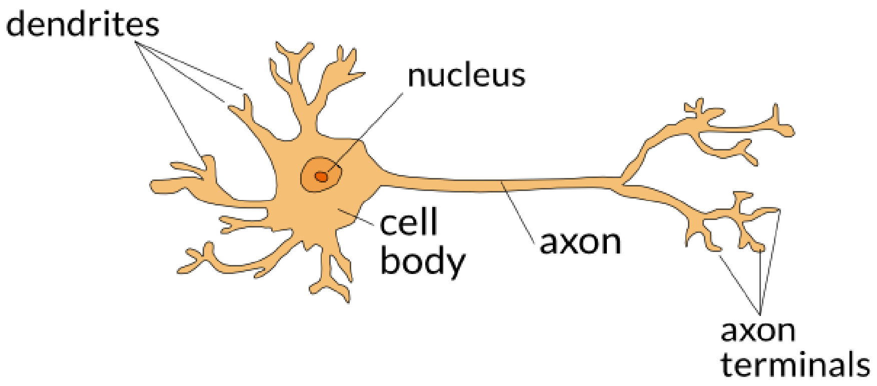
There are many courses available online about deep learning, and far too much material to cover in the remaining time

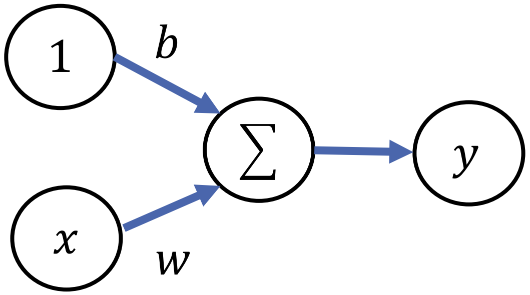
- So, we’ll focus on a high-level understanding of a few temporally motivated algorithms and their implementation.

The Perceptron

The perceptron is a fundamental building block of machine learning

- Equation of a line, or simple form of SVM
- Name given to an artificial neuron, modeled after neurons in the brain





$$y = wx + b$$

- Equation of a line
- Linear regression

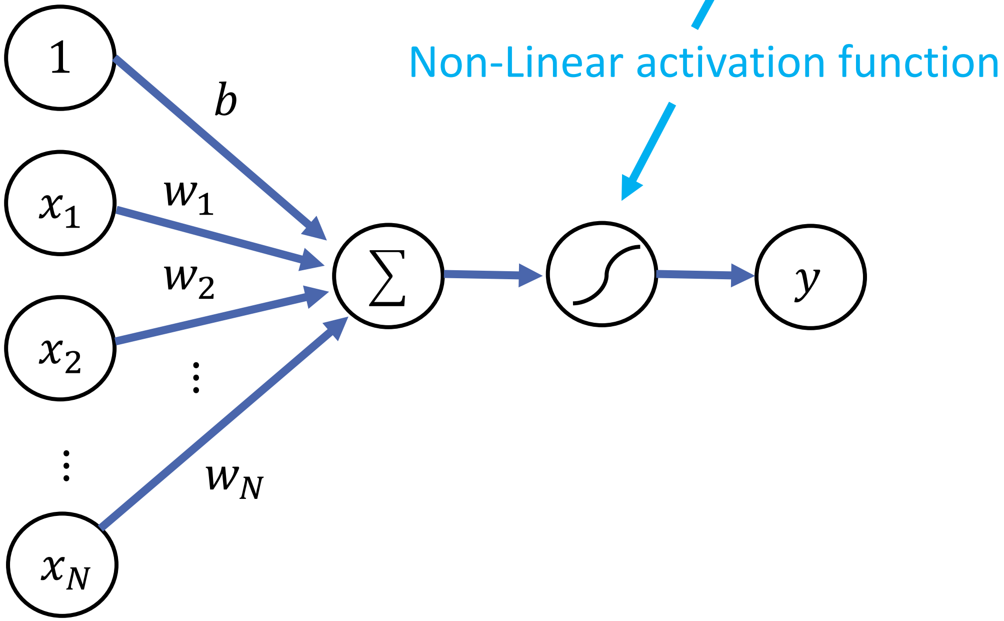
https://www.codeproject.com/Articles/1205732/Build-Simple-AI-NET-Library-Part-Perceptron

The Perceptron

We can easily expand this to account for multiple inputs and/or to describe non-linear mappings/activations as follows:

$$y = wx + b$$

$$y = g\left(b + \sum_{i=1}^{N} x_i w_i\right) = g(b + \boldsymbol{X}^T \boldsymbol{W})$$



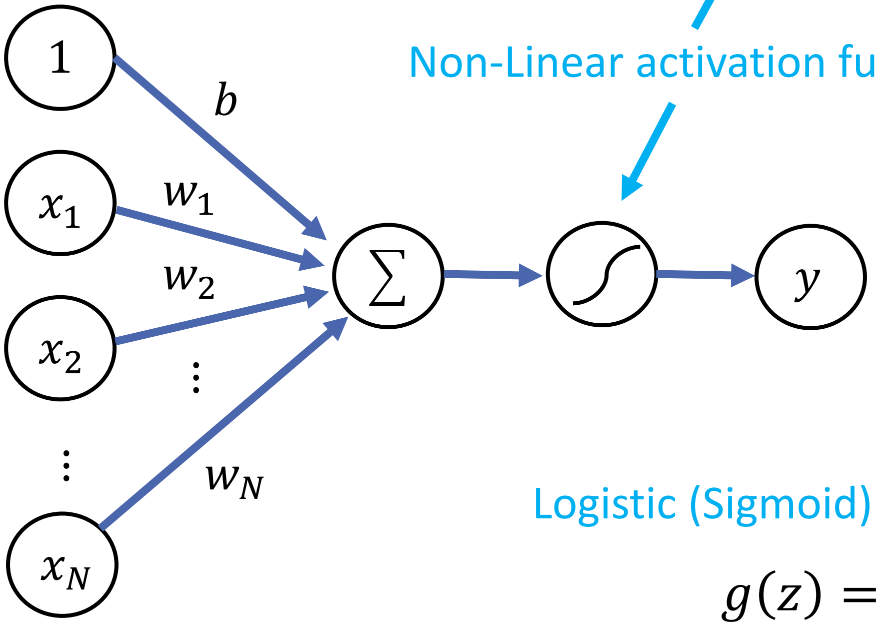
Non-Linear activation function

The Perceptron

We can easily expand this to account for multiple inputs and/or to describe non-linear mappings/activations as follows:

$$y = wx + b$$

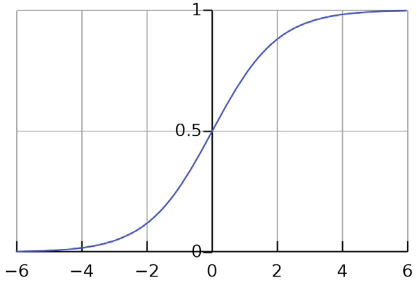
$$y = g\left(b + \sum_{i=1}^{N} x_i w_i\right) = g(b + \boldsymbol{X}^T \boldsymbol{W})$$



Non-Linear activation function

Logistic (Sigmoid) Function

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



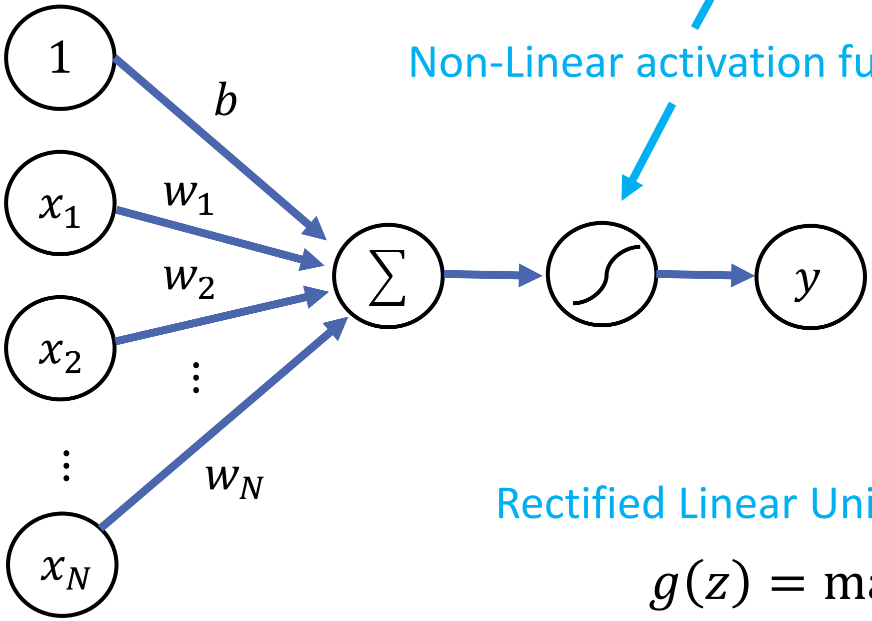
Bounds the output between 0 and 1!   Good for probabilities

The Perceptron

We can easily expand this to account for multiple inputs and/or to describe non-linear mappings/activations as follows:

$$y = wx + b$$

$$y = g\left(b + \sum_{i=1}^{N} x_i w_i\right) = g(b + \boldsymbol{X}^T \boldsymbol{W})$$

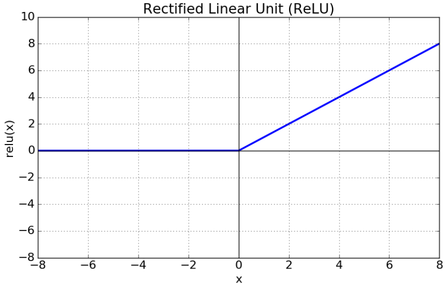


Non-Linear activation function

$$g(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

Rectified Linear Unit (ReLU)

$$g(z) = \max(0, z)$$

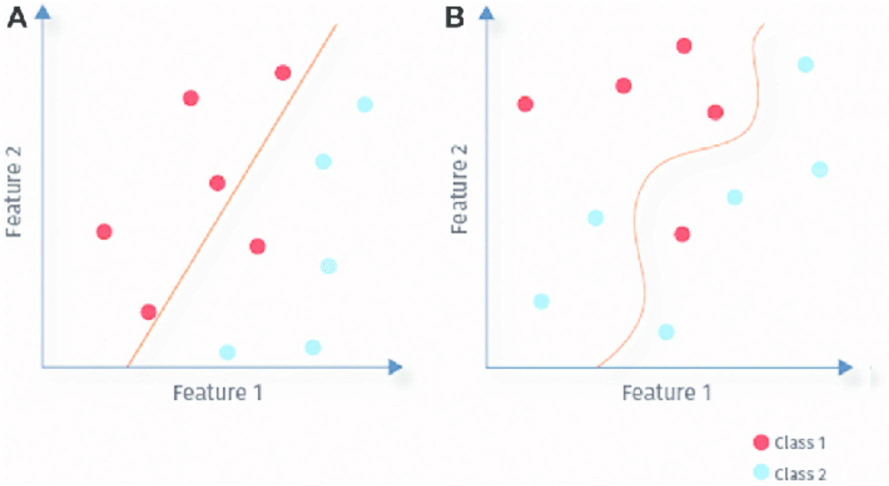


Piecewise linear function; popular in deep networks

Nonlinear Activation Functions

These nonlinear activation functions enable us to determine nonlinear mappings and boundaries

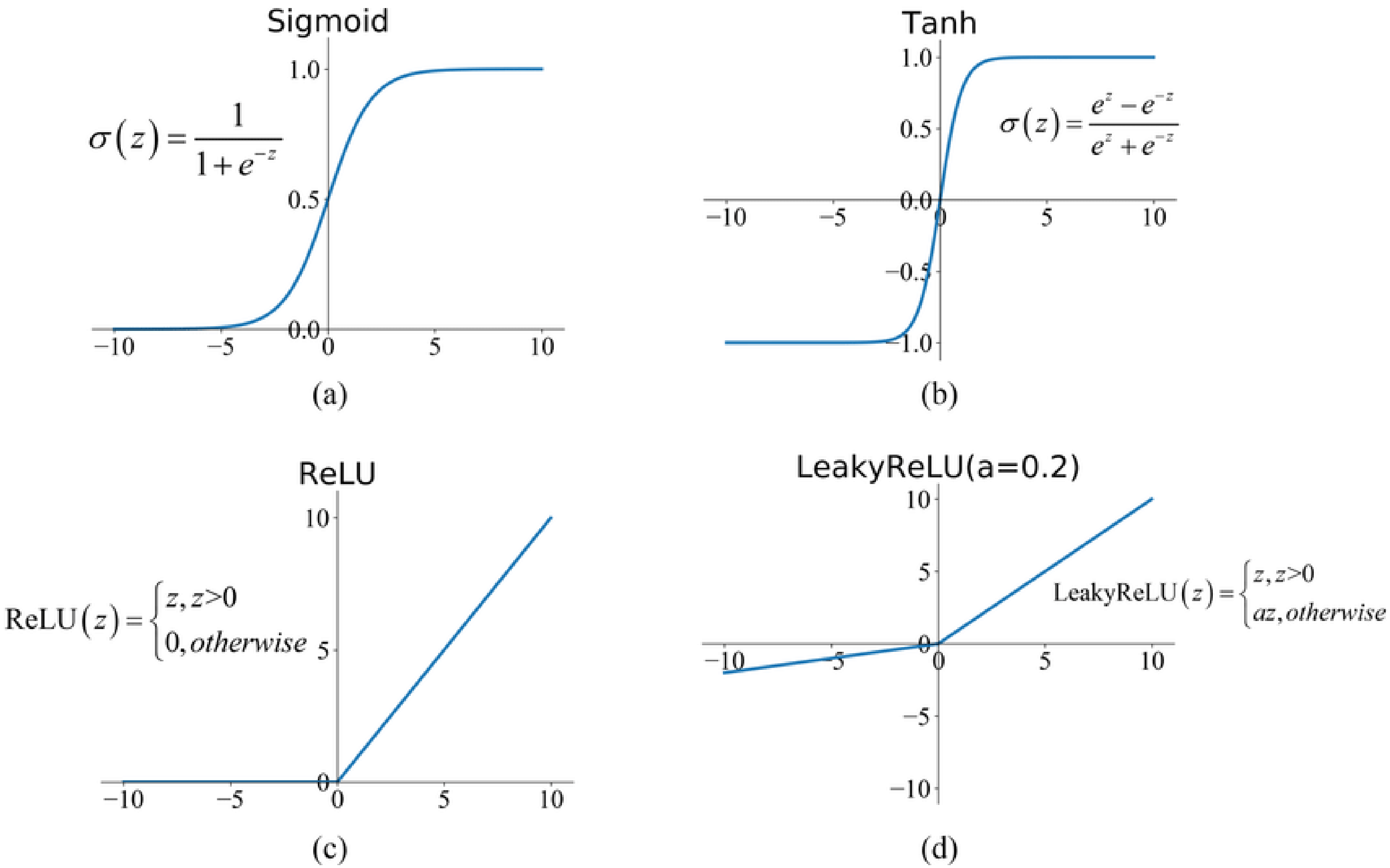
- Even deep networks, if composed of many linear activation functions, would only yield a linear decision



$y = wx + b$ $\qquad\qquad$ $y = ?$

https://www.researchgate.net/figure/Linear-versus-non-linear-classification-Illustration-of-the-separation-of-data-samples_fig1_311249567

Nonlinear Activation Functions



(a)

(b)

(c)

(d)

https://www.researchgate.net/figure/Commonly-used-activation-functions-a-Sigmoid-b-Tanh-c-ReLU-and-d-LReLU_fig3_335845675

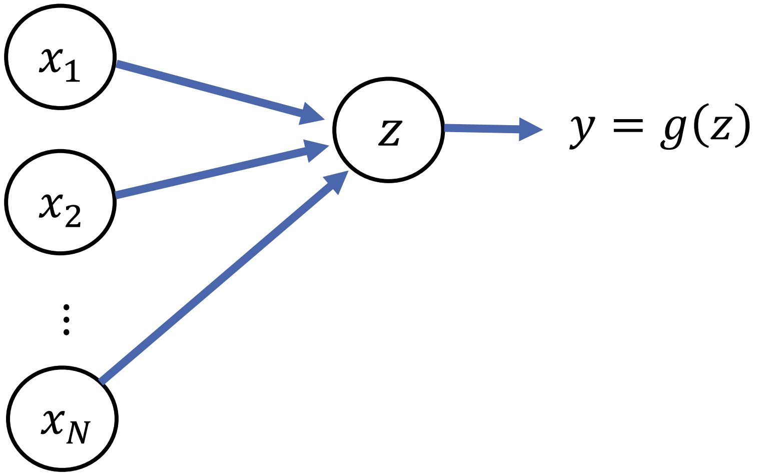
Perceptrons

Now that we have a basic building block, we can start adding to it.

- For simplicity, let's replace

$$y = g\left(b + \sum_{i=1}^{N} x_i w_i\right) \qquad y = g(z) \qquad z = b + \sum_{i=1}^{N} x_i w_i$$

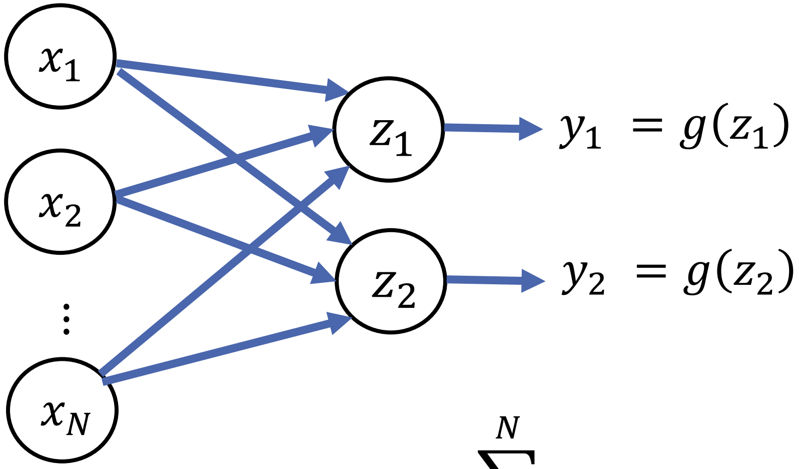
and clean up the diagram as



Dense Layers

We can then modify this to allow for multiple outputs by adding different mixtures of the inputs

- We can stack additional perceptrons together, with each connected to the inputs via their own set of weights
- When these are connected to each input, they are referred to as dense layers
- There can be as many perceptrons added as we want



$$y_1 = g(z_1)$$

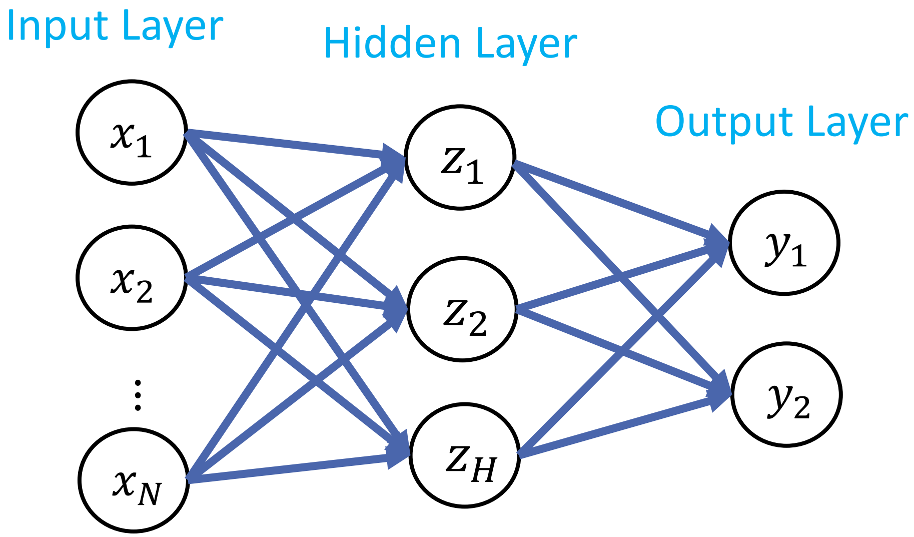
$$y_2 = g(z_2)$$

$$z_j = b_j + \sum_{i=1}^{N} x_i w_{i,j}$$

Neural Network

A neural network is a small extension of the multi-output perceptron

- We add another stage, making the middle layer "hidden"
- The weights for each stage are different



The inputs to the 2nd stage are the outputs from the 1st

$$z_j = b_j + \sum_{i=1}^{N} x_i w_{i,j}$$

N Inputs

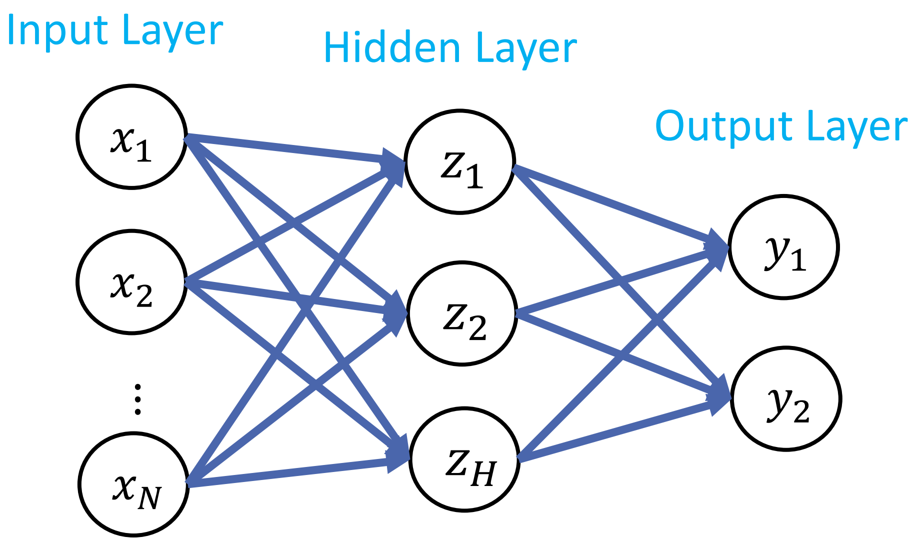
$$y_k = g\left\{b_k \sum_{i=1}^{H} g(z_j) w_{i,k}\right\}$$

H Nodes

Neural Network

A neural network is a small extension of the multi-output perceptron

- We add another stage, making the middle layer "hidden"
- The weights for each stage are different



Input Layer

Hidden Layer

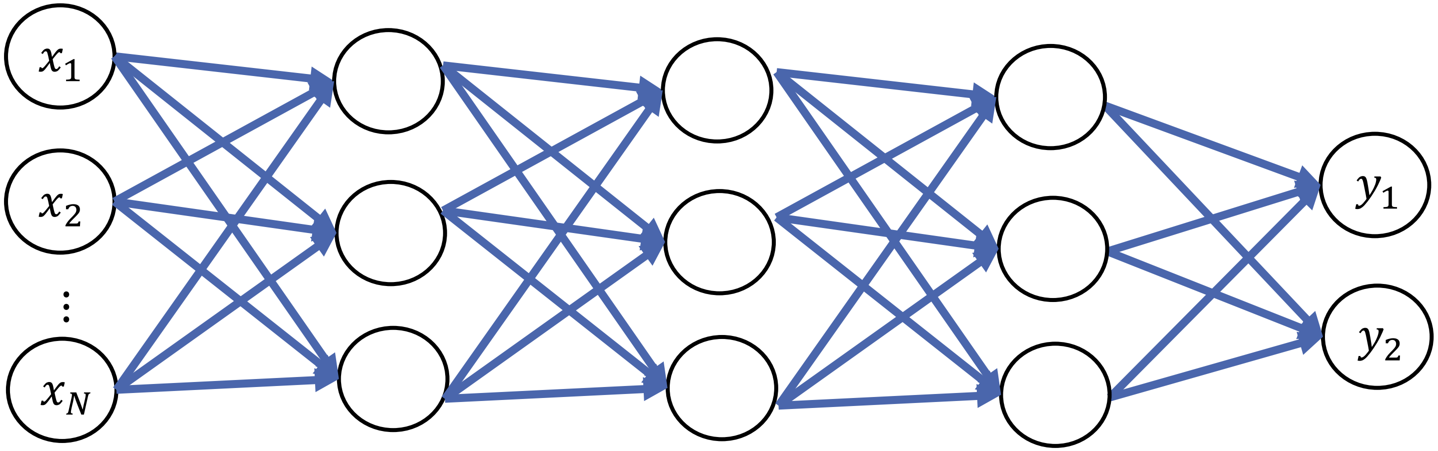
Output Layer

We can start to define complex structures/models by changing how these layers are composed and stacked.

Neural Network

A neural network is a small extension of the multi-output perceptron

- We add another stage, making the middle layer “hidden”
- The weights for each stage are different



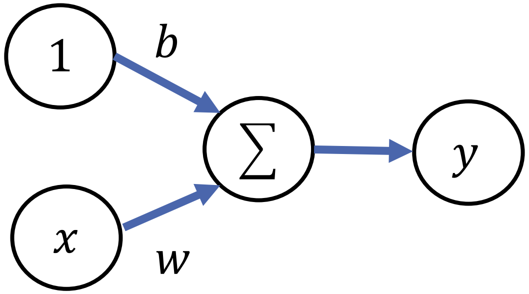
We can start to define complex structures/models by changing how these layers are composed and stacked.

- Different numbers of layers, nodes, activation functions, connections
- By combining many layers, we build very deep networks, capable of describing very detailed aspects of the inputs

Neural Network

The key for these models to work is the correct determination of the weighting terms b and w that we defined earlier.

- In the most basic linear perceptron model, we saw that



$$y = wx + b$$

- But how do we determine the values of b and w?

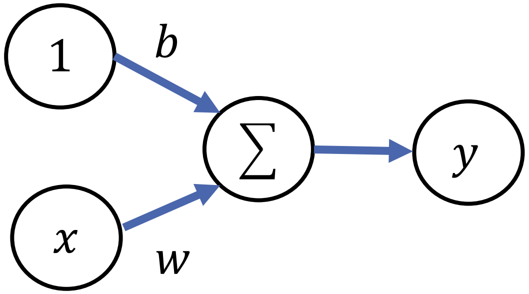
In neural networks, we iteratively a) measure how well we did and b) update our parameters

- Similar to the expectation maximization approach when fitting Gaussians for HMM
- This requires us to somehow define how well we did

Loss Function

In neural networks, we iteratively a) measure how well we did and b) update our parameters

- Similar to the expectation maximization approach when fitting Gaussians for HMM
- This requires us to somehow define how well we did



$$y = wx + b$$

- For a set of supervised training data, we have pairs of inputs, $\boldsymbol{X}_i = \{x_1, x_2, \dots\}$ and outputs, $\boldsymbol{Y}_i = \{y_1, y_2, \dots\}$
- Based on a current set of weights, $\boldsymbol{W}$, we can measure how well we do at approximating these true outputs based on the inputs
- We conduct this measurement using a Loss Function

$$\mathcal{L}\{f(\boldsymbol{X}_i, \boldsymbol{W}), \boldsymbol{Y}_i\}$$

Note that the b term (often called w_0 is mixed into $\boldsymbol{W}$ for sake of clarity

Loss Function

We can then combine the "loss" on each of these predictions, $\mathcal{L}\{f(\boldsymbol{X}_i, \boldsymbol{W}), \boldsymbol{Y}_i\}$ by taking the average across all training pairs

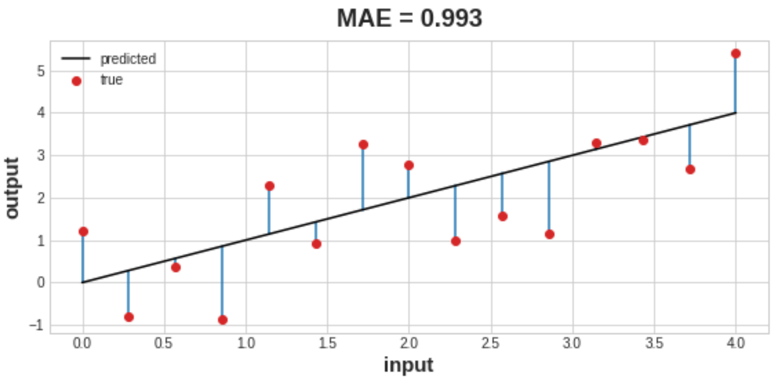
$$J(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\{f(\boldsymbol{X}_i, \boldsymbol{W}), \boldsymbol{Y}_i\}$$

where N is the number of training pairs, and $J(\boldsymbol{W})$ is the loss associated with a given set of weights

- Known as empirical loss/risk
- We want to update our weights so that we minimize $J(\boldsymbol{W})$

We haven't yet defined what the $\mathcal{L}\{\ \}$ function does yet

- We can select different functions for this
- e.g. For continuous outputs (regression), we often use the L1 Loss - Mean Absolute Error or L2 Loss – Mean Squared Error
- e.g. For binary classification outputs, we use Softmax, Binary Cross Entropy



https://www.kaggle.com/ryanholbrook/stochastic-gradient-descent

Learning from Loss

Now that we have a function that can give us a number of how well we perform given a particular set of $\boldsymbol{W}$, we need to somehow optimize the those $\boldsymbol{W}$ to reduce the loss.

$$\boldsymbol{W} = argmin_w \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\{f(\boldsymbol{X}_i, \boldsymbol{W}), \boldsymbol{Y}_i\} = argmin_w J(\boldsymbol{W})$$

But, remember that $\boldsymbol{W}$ describes ALL of the weights we have in our network

- This could include the weights for many neurons across many layers

https://www.kaggle.com/ryanholbrook/stochastic-gradient-descent

Learning from Loss

Now that we have a function that can give us a number for how well we perform given a particular set of $\boldsymbol{W}$, we need to somehow optimize those $\boldsymbol{W}$ to reduce that loss.

$$\boldsymbol{W} = argmin_w \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\{f(\boldsymbol{X}_i, \boldsymbol{W}), \boldsymbol{Y}_i\} = argmin_w J(\boldsymbol{W})$$

The process of doing this is referred to as gradient descent

- Iterative approach
- Begin with some set of weights (to be determined)
- Using known training data (supervised learning), use the network to predict the associated outputs
- Measure the loss between the predictions and “true” values
- Adjust the weights such that the loss becomes smaller

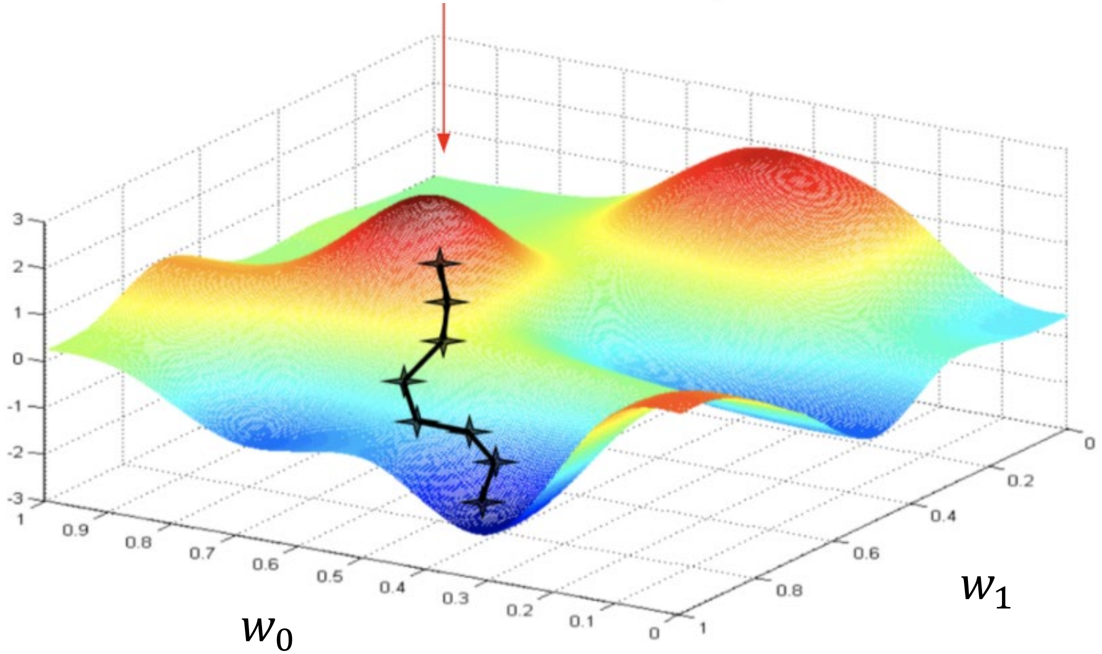
This last step is important and non-trivial, as the choice of direction and rate of change is critical for speed and performance.

Gradient Descent

Th process of updating weights is critical for speed and performance.

- We can find the loss , $J(\boldsymbol{W})$ associated with every combination of (w_0, w_1)
- Follow the path “down the hill” by adjusting our weights based on the gradient, $\frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$

Some random starting point (w_0, w_1)



$J(w_0, w_1)$

w_0

w_1

https://miro.medium.com/max/1400/1*IfVAzJJWIrCw0saOZAxpzA.png

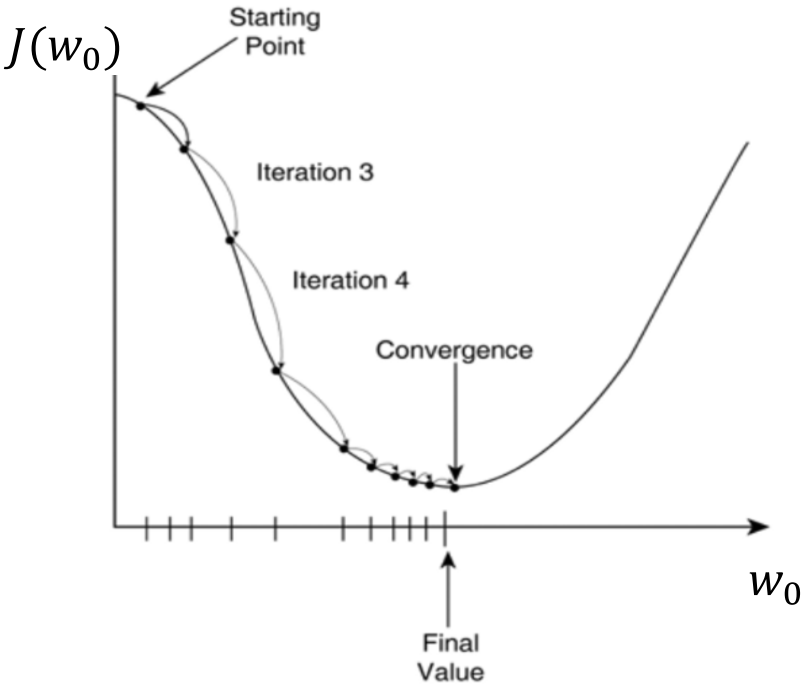
Gradient Descent

The process of updating weights is critical for speed and performance.

- We can find the loss , $J(\boldsymbol{W})$ associated with every combination of (w_0, w_1)
- Follow the path "down the hill" by adjusting our weights based on the gradient, $\frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$

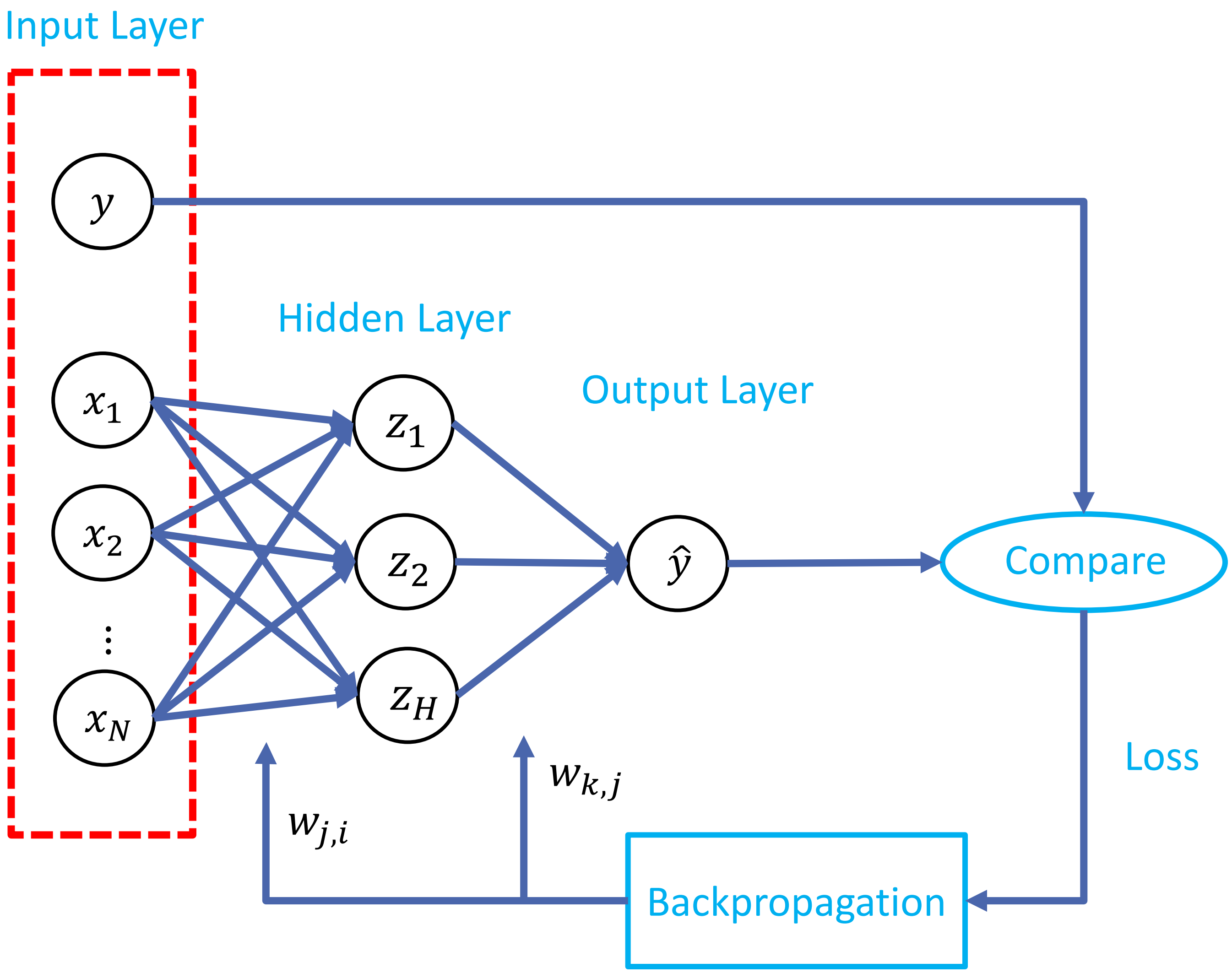
1) Initialize weights randomly
2) Compute loss gradient, $\frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$
3) Update weights, $\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$
4) Repeat until convergence

- α is the learning rate, which is used to adjust how aggressive the adaptation is.



https://www.kdnuggets.com/2020/05/5-concepts-gradient-descent-cost-function.html

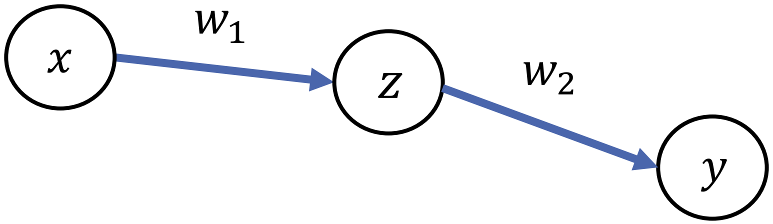
Backpropagation



Backpropagation

The adjustment of the weights in each iteration of the learning requires us to know how to change the w's so that we travel an appropriate amount in the right direction along the loss curve

- We haven't really talked about how to compute the gradient, $\frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$, though
- The question is, how does a change in w_i affect the loss function $J(\boldsymbol{W})$.
- This is done using the chain rule to backpropagate our way through the network



$$\frac{dJ(\boldsymbol{W})}{dw_2} = \frac{dJ(\boldsymbol{W})}{dy} * \frac{dy}{dw_2}$$

$$\frac{dJ(\boldsymbol{W})}{dw_1} = \frac{dJ(\boldsymbol{W})}{dy} * \frac{dy}{dw_1} = \frac{dJ(\boldsymbol{W})}{dy} * \left\{\frac{dy}{dz_1} * \frac{dz_1}{dw_1}\right\}$$

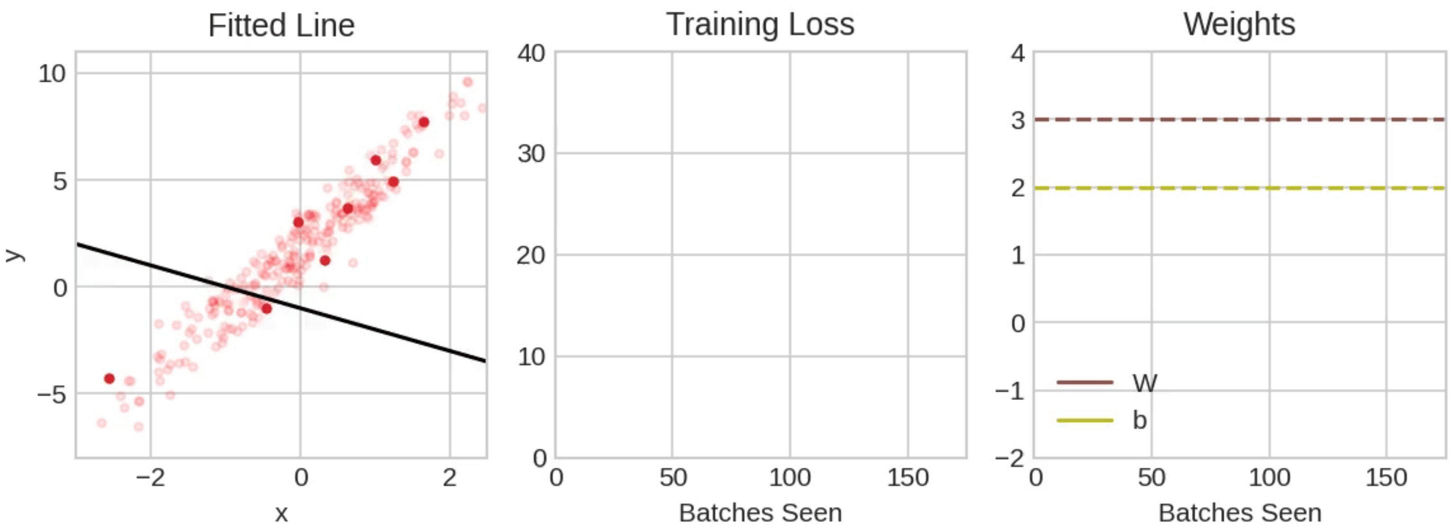
Shows us the effect of making a small change to our weights on the loss function

- Of course, this gets more complicated when we have more weights

Stochastic Gradient Descent

Each time the network sees all of the data is called an epoch.

- Networks are typically trained using multiple epochs.



Example of a linear network being trained.

- The pale red dots depict the full training set, solid red dots are the mini-batches.
- After each batch, the weights (b and w) are adjusted
- The line eventually converges to its best fit, the loss gets approaches a minimum, and the weights converge to their final values

https://imgur.com/rFI1tlk

Stochastic Gradient Descent

The process of conventional gradient descent can be problematic and computationally expensive

- You need to evaluate all training data to compute $J(\boldsymbol{W})$ in each iteration
- Sometimes called batch gradient descent

An alternative approach is stochastic gradient descent

- Assumes that the individual samples stochastically describe the overall dataset
- In each iteration, a single example is used to update $J(\boldsymbol{W})$ and update the weights, rather than all samples
- Can be a noisier descent, but is general faster and avoids local minima

Combining both approaches yields mini-batch gradient descent

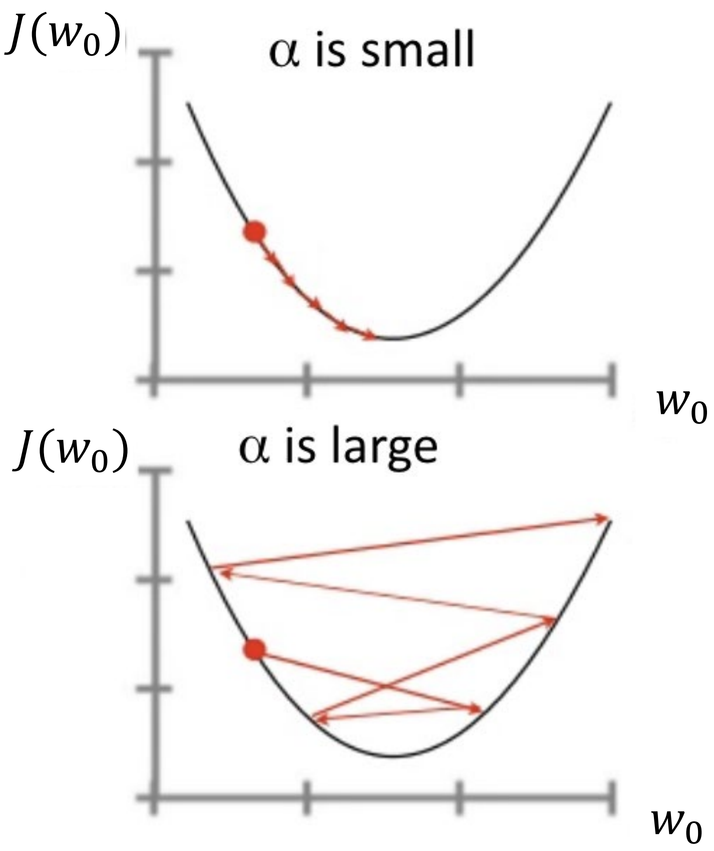
- The training data are split into smaller subsets (batches)
- These batches are then iteratively used to update $J(\boldsymbol{W})$ and update the weights, rather than all samples or a single sample
- Most common approach to gradient descent in deep learning

Learning Rate

The process of updating weights is critical for speed and performance.

- We can find the loss , $J(\boldsymbol{W})$ associated with every combination of (w_0, w_1)
- Follow the path "down the hill" by adjusting our weights based on the gradient, $\frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$

1) Initialize weights randomly
2) Compute loss gradient, $\frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$
3) Update weights, $\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{dJ(\boldsymbol{W})}{d\boldsymbol{W}}$
4) Repeat until convergence



- α is the learning rate, which is used to adjust how aggressive the adaptation is.
- The larger the learning rate, the faster the descent
- Too small, and you waste time
- Too large, and you waste time

https://www.kdnuggets.com/2020/05/5-concepts-gradient-descent-cost-function.html

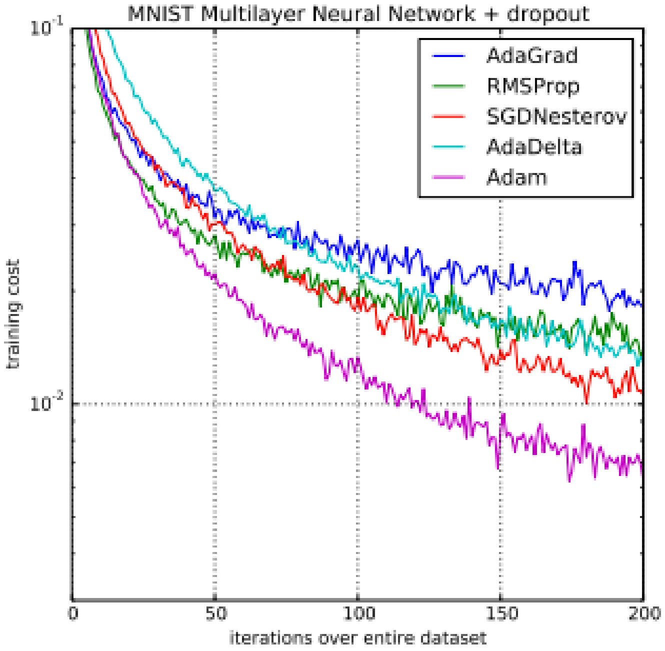
Adam Optimizer

The selection of mini-batch size and learning rate is not deterministic

- Requires trial and error
- There's been a lot of work on optimizing the learning process

Adam Optimization is a popular algorithm that uses an adaptive learning rate

- Adam builds on previous approaches like AdaGrad and RMSProp, and can handle sparse gradients on noisy datasets
- Maintains a per-parameter learning rate (instead of a global α learning rate) that can be adapted based on the gradients of the weights (adaptive moment estimation)
- Easier to configure and requires less customization (the default is often 'good enough')



https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Adam Optimizer

There are still parameters to choose, but the defaults tend to work well enough because of the adaptive behavior

- **alpha:** Learning rate (or step size), which governs how aggressively the weights are updated (e.g. 0.001)
- **beta1:** Exponential decay rate for the first moment estimates (e.g. 0.9)
- **beta2:** Exponential decay rate for the second-moment estimates (e.g. 0.999)
- **epsilon:** Numerical trick to prevent division by zero (e.g. $10e^{-8}$)

Examples of default Adam parameter values

- TensorFlow: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08
- Keras: lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08
- Caffe: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08
- Torch: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8

https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

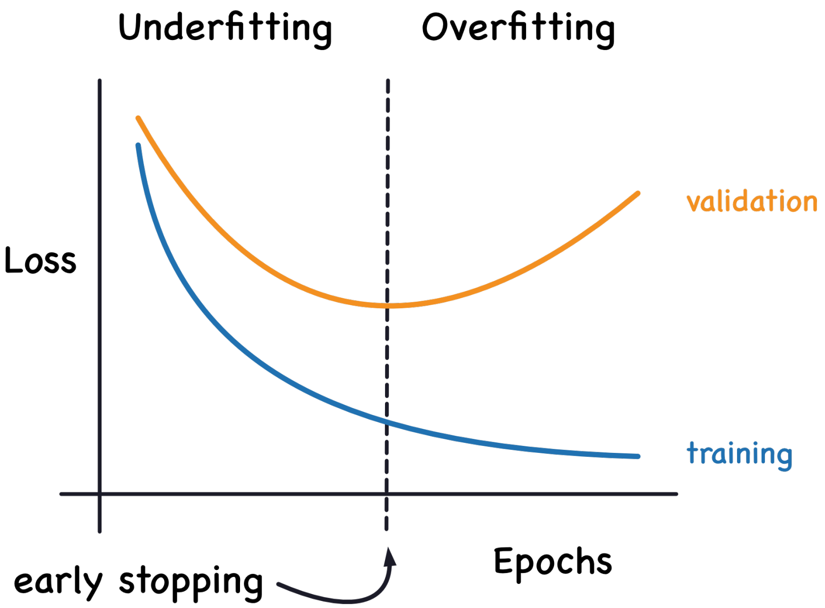
Overfitting

Just like other classifiers, increased capacity can lead to over-tuning/over-fitting for the training set, and poor generalization to new data

- Because neural networks are able explain complex details, this is historically a very real challenge (they can "memorize" the data, learning the noise)
- The more neurons, interconnections, and layers, the more capacity of the network

To mitigate this, validation sets are used

- Hold out some of the training data
- Test how well the learned parameters generalize to the new data
- Once the loss function on the validation set stops decreasing, training is stopped (a.k.a early stopping)



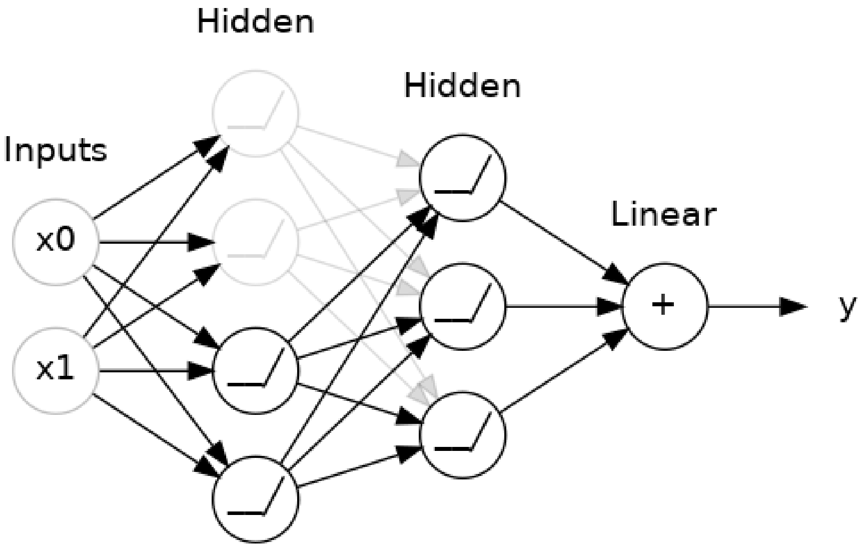
https://www.kaggle.com/ryanholbrook/overfitting-and-underfitting

Dropout

Another technique that is commonly used to reduce overfitting is dropout.

- To avoid learning very specific combinations of parameters that explain only a specific (possibly noisy) result, we randomly drop out some of a layer’s inputs each iteration of training
- This forces the network to find higher-level, more generalizable, patterns that aren’t dependent on fringe cases

- Some people think of this as a form of majority vote or ensemble learning, which tends to smooth out errors
- Here, both hidden layers have been implemented as dropout layers
- Done by setting the activations to 0



https://www.kaggle.com/ryanholbrook/dropout-and-batch-normalization

Batch Normalization

Because the weights of a network are multiplied by the inputs, it's important and helpful for interpretation, to normalize the inputs to a common scale (e.g. normalize by mean and standard deviation)

- Batch normalization layers take this a step further by also re-normalizing/scaling the data within the network

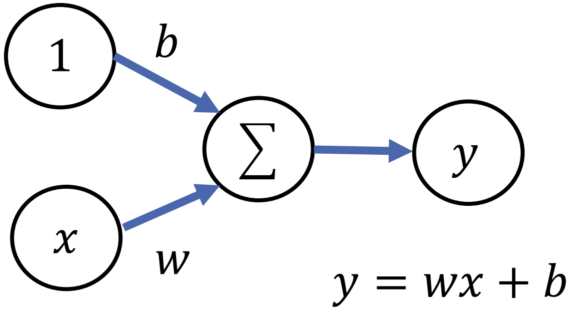
There are, of course, many other specific tweaks and considerations when it comes to practical implementation of neural networks, but these concepts form the basis of feed-forward neural networks

- We'll use these concepts, and knowledge of the modularity of the perceptron in neural networks, to look at deep learning

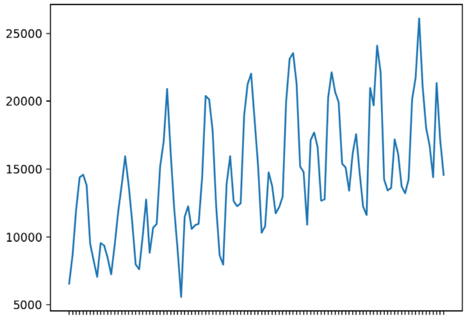
Neural Networks in Time Series

Although there is no temporal framework built directly into the conventional feedforward network, we can still use them for time series prediction

- Recall that we used regression as a naïve method of prediction based on trend, and that a single perceptron, even without non-linear activation, is just



$$y = wx + b$$



We could simply train the model by giving it past examples of the data, and asking it to learn to predict a future point

- Think about and AR(1) model…
- But this approach makes some naïve assumptions about the data that don't tend to translate all that well

Summary

In this section, we briefly jumped into the foundations of neural networks and how to train them

- Built on a single perceptron modeled after neurons in the brain
  The output(s) are a weighted combination of the inputs
- Nonlinearities made possibly by adding non-linear activation functions
  Sigmoid, TANH, ReLU, etc…
- Backpropagation through the network allows weights to be tweaked so that they reduce a loss function (error in the predicted outputs)
- Adam is a well-known optimized used to speed up and control the gradient descent process as the weghts are learned
- Overfitting can be a problem with complex networks
  Early stopping (validation), drop out, batch normalization are all tools to help control the learning process
- Can be used for time series, but not really designed for it

In the next sections, we'll build on these to look at deep learning networks designed to handle spatial or temporal data.

Q&A