Numeric Optimization

Héctor Corrada Bravo

University of Maryland, College Park, USA

DATA606: 2020-04-19

Historical Overview

Neural networks are a decades old area of study.

Initially, these computational models were created with the goal of mimicking the processing of neuronal networks.

Historical Overview

Inspiration: model neuron as processing unit.

Some of the mathematical functions historically used in neural network models arise from biologically plausible activation functions.

Historical Overview

Somewhat limited success in modeling neuronal processing

Neural network models gained traction as general Machine Learning models.

Historical Overview

Strong results about the ability of these models to approximate arbitrary functions

Became the subject of intense study in ML.

In practice, effective training of these models was both technically and computationally difficult.

Historical Overview

Starting from 2005, technical advances have led to a resurgence of interest in neural networks, specifically in *Deep Neural Networks*.

Deep Learning

Advances in computational processing:

- powerful parallel processing given by Graphical Processing Units

Deep Learning

Advances in computational processing:

- powerful parallel processing given by Graphical Processing Units

Advances in neural network architecture design and network optimization

Deep Learning

Advances in computational processing:

- powerful parallel processing given by Graphical Processing Units

Advances in neural network architecture design and network optimization

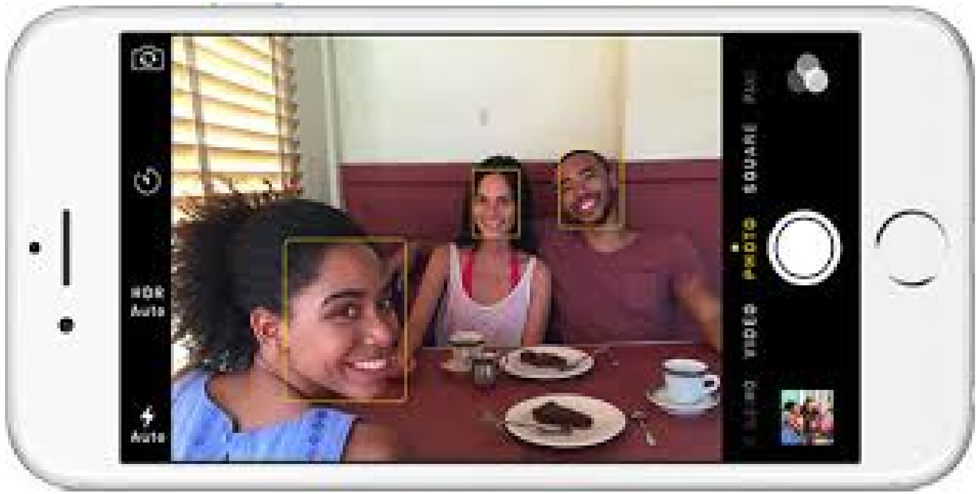
Researchers apply Deep Neural Networks successfully in a number of applications.

Deep Learning

Self driving cars make use of Deep Learning models for sensor processing.

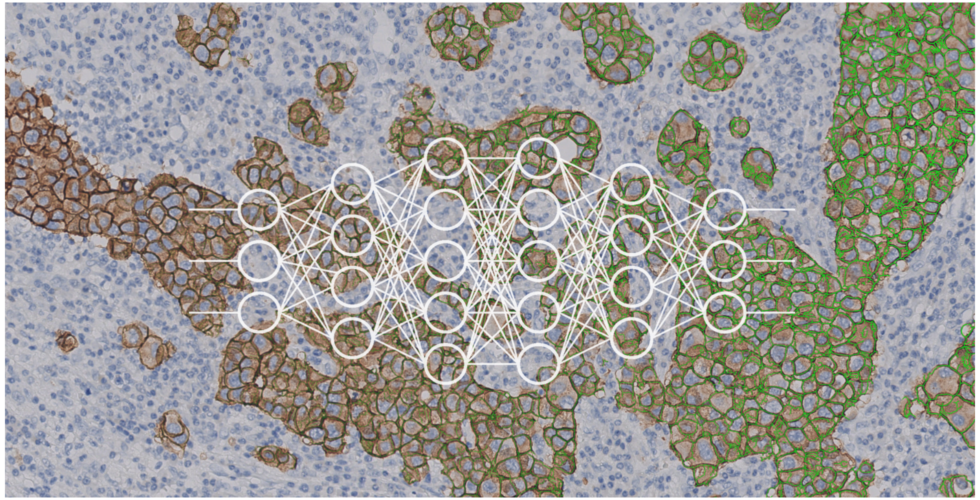
Deep Learning

Image recognition software uses Deep Learning to identify individuals within photos.



Deep Learning

Deep Learning models have been applied to medical imaging to yield expert-level prognosis.



Deep Learning

An automated Go player, making heavy use of Deep Learning, is capable of beating the best human Go players in the world.

Feed-forward Neural Networks

We will present the feed-forward neural network formulation for a general case where we are modeling K outcomes $Y_1, \ldots, Y_k$ as $f_1(X), \ldots, f_K(X)$.

Feed-forward Neural Networks

In multi-class classification, categorical outcome may take multiple values

We consider Y_k as a discriminant function for class k,

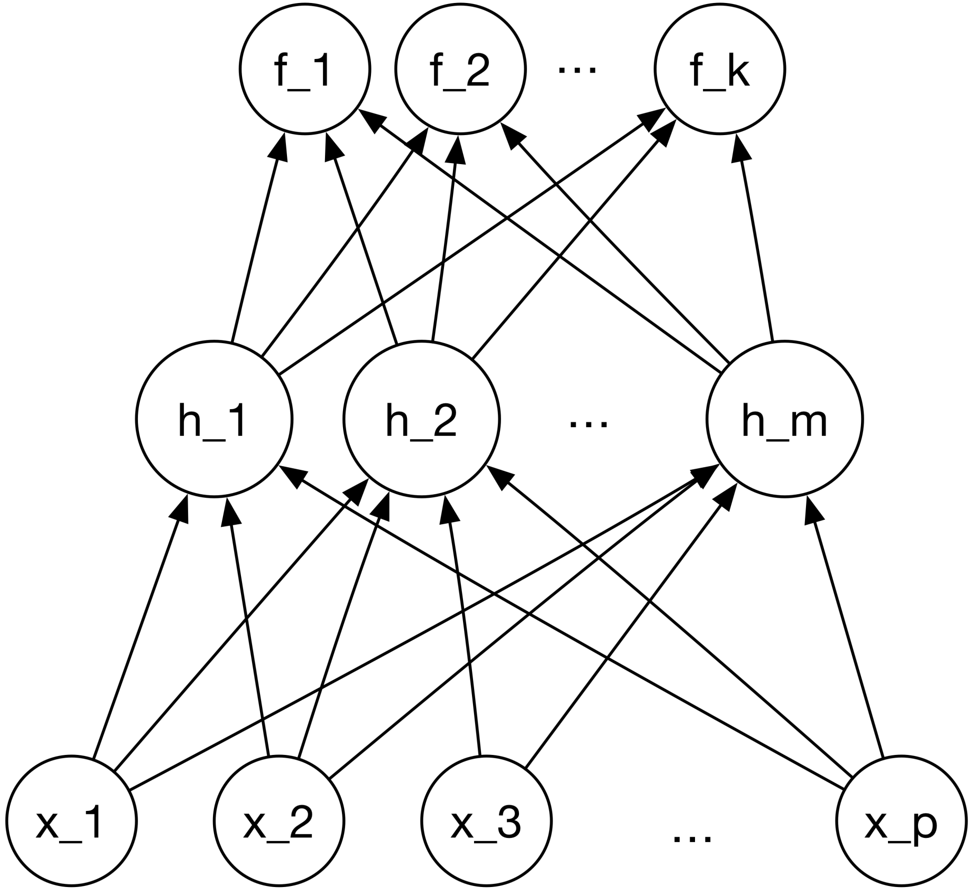
Final classification is made using $\arg\max_k Y_k$. For regression, we can take $K = 1$.

Feed-forward Neural Networks

A single layer feed-forward neural network is defined as

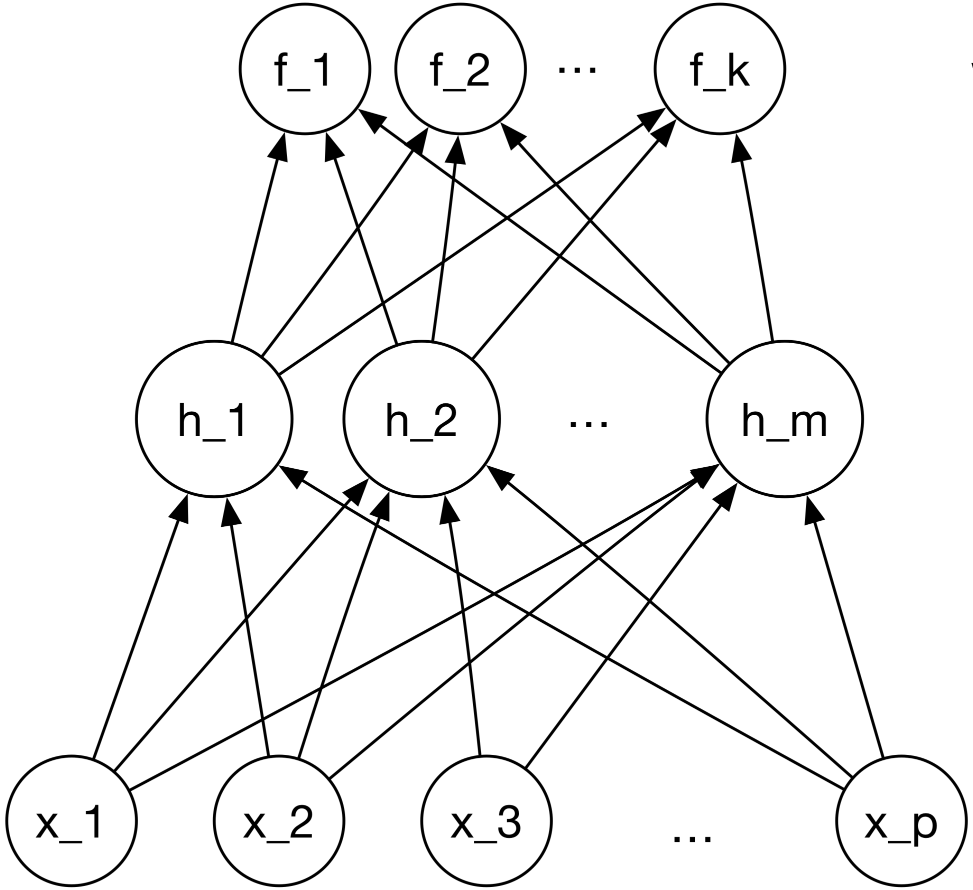
$$h_m = g_h(\mathbf{w}'_{1m} X), \ m = 1, \ldots, M$$
$$f_k = g_{fk}(\mathbf{w}'_{2k} \mathbf{h}), \ k = 1, \ldots, K$$

Feed-forward Neural Networks



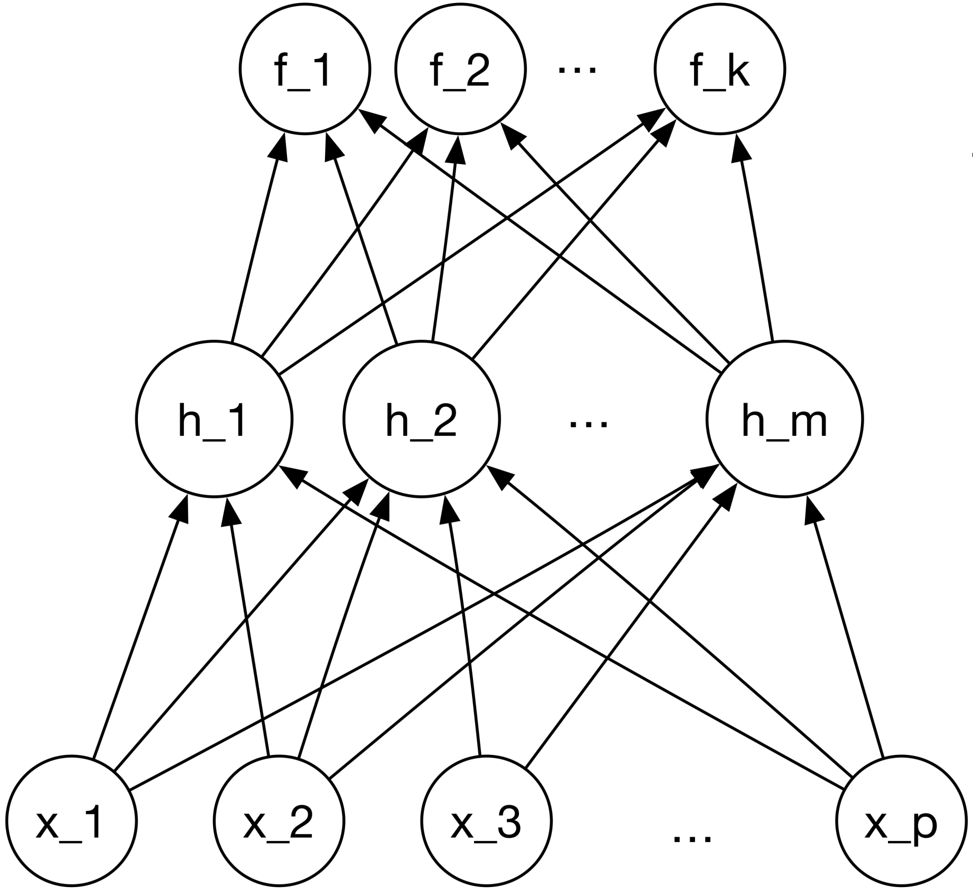
The network is organized into *input*, *hidden* and *output* layers.

Feed-forward Neural Networks



Units h_m represent a *hidden layer*, which we can interpret as a *derived* non-linear representation of the input data as we saw before.

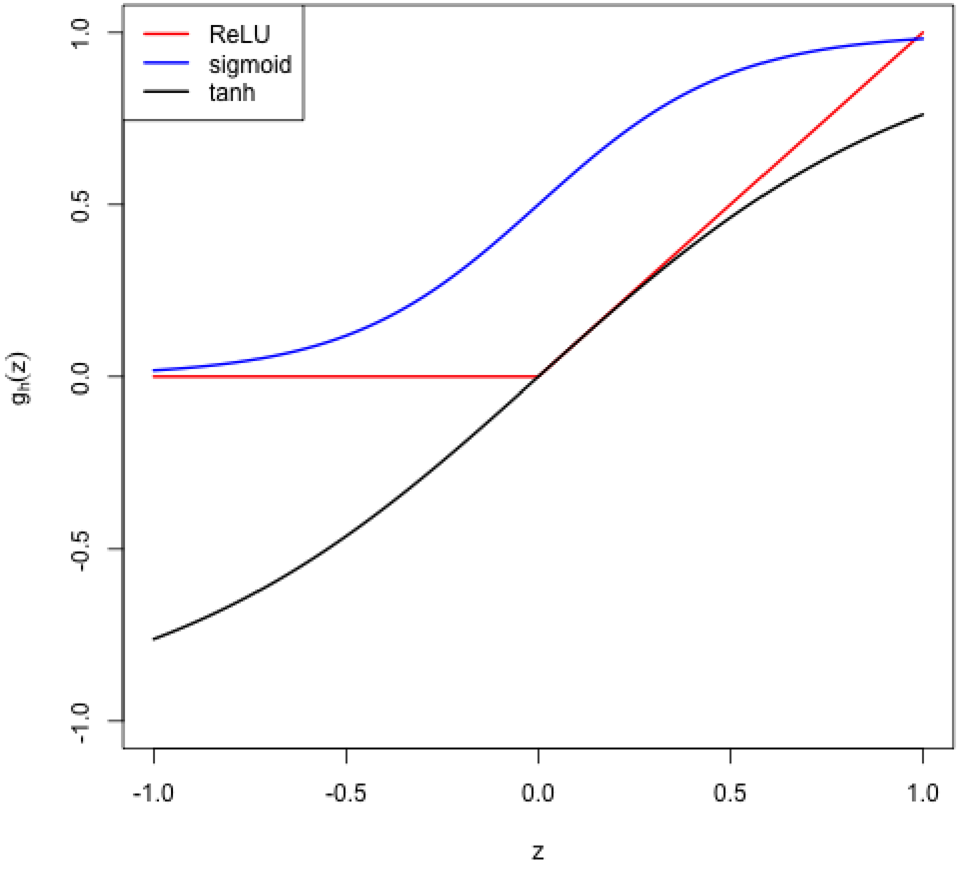
Feed-forward Neural Networks



Function g_h is an *activation* function used to introduce non-linearity to the representation.

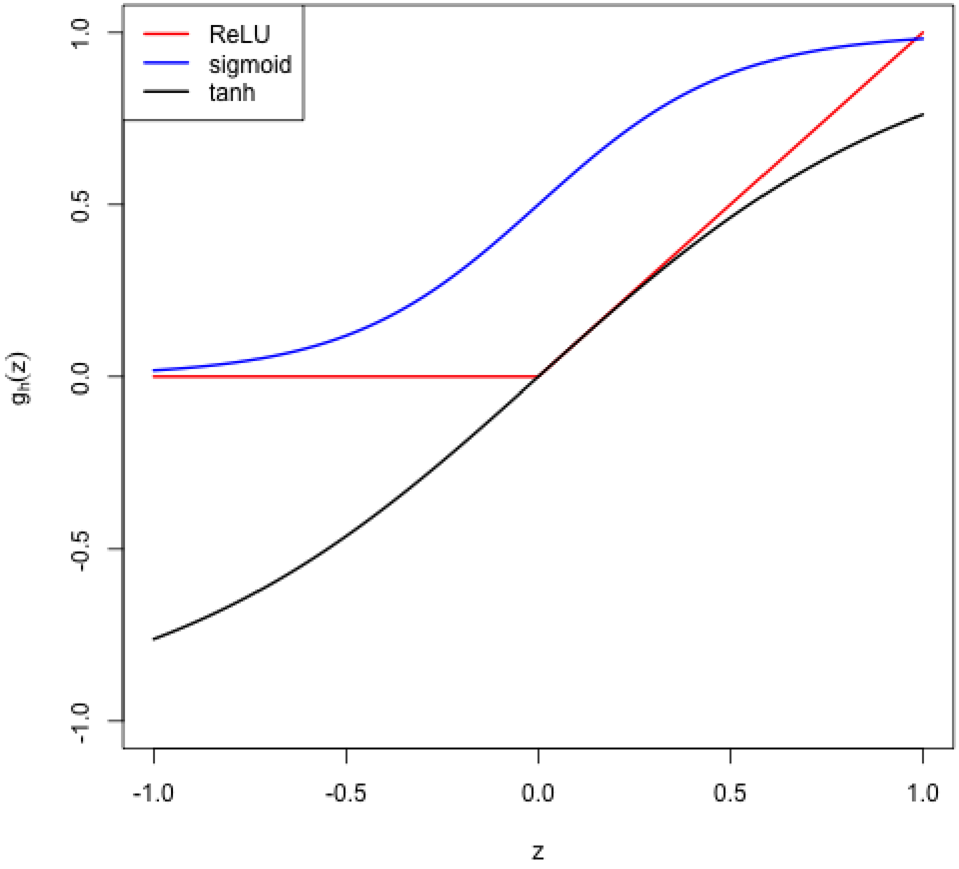
Feed-forward Neural Networks

Historically, the sigmoid activation function was commonly used $g_h(v) = \frac{1}{1+e^{-v}}$ or the hyperbolic tangent.

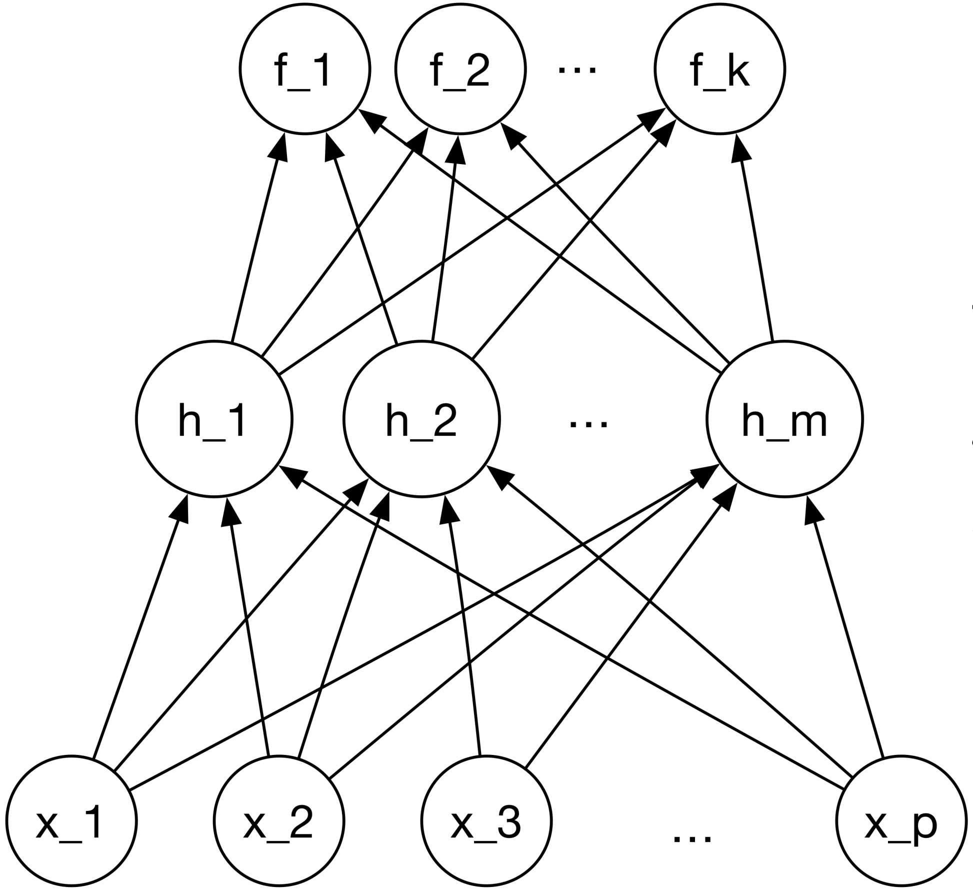


Feed-forward Neural Networks

Nowadays, a rectified linear unit (ReLU) $g_h(v) = \max\{0, v\}$ is used more frequently in practice. (there are many extensions)



Feed-forward Neural Networks

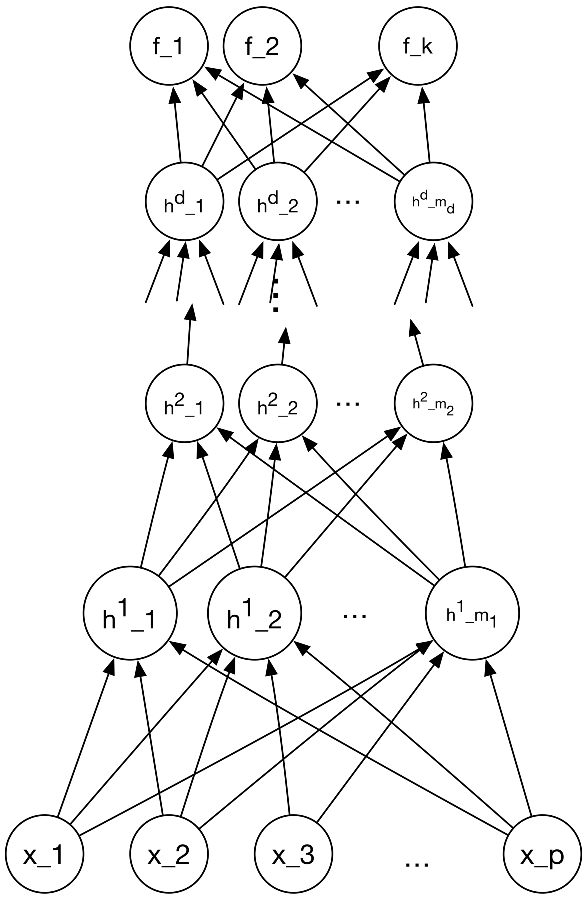


Function g_f used in the output layer depends on the outcome modeled.

For classification a *soft-max* function can be used $g_{fk}(t_k) = \frac{e^{t_k}}{\sum_{l=1}^{K} e^{t_k}}$ where $t_k = \mathbf{w}'_{2k}\mathbf{h}$.

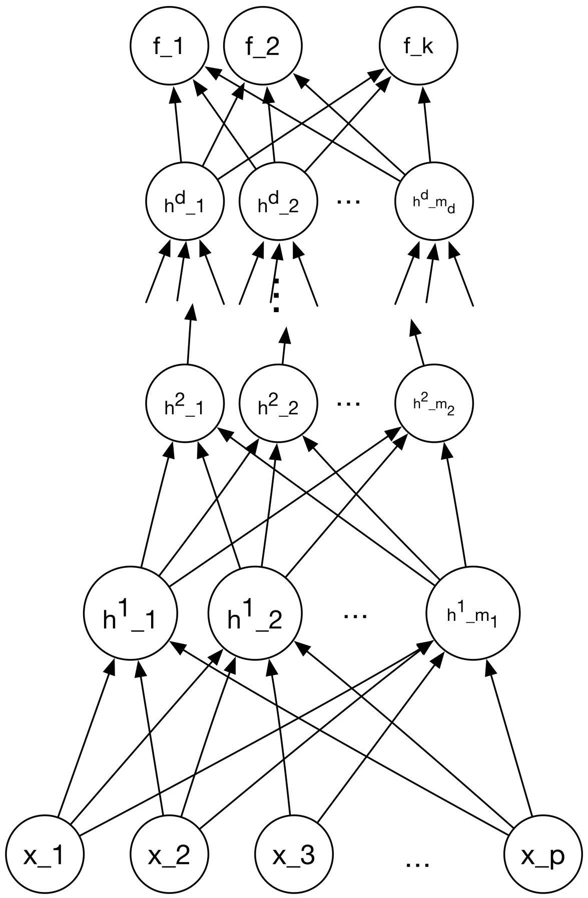
For regression, we may take g_{fk} to be the identify function.

Deep Feed-Forward Neural Networks



The general form of feed-forward network can be extended by adding additional *hidden layers*.

Deep Feed-Forward Neural Networks

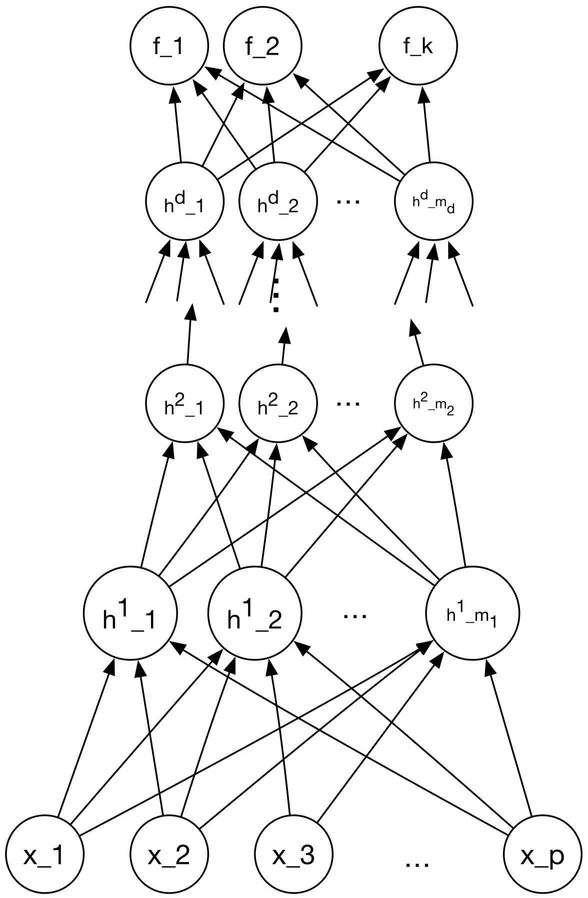


Empirically, it is found that by using more, thinner, layers, better expected prediction error is obtained.

However, each layer introduces more non-linearity into the network.

Making optimization markedly more difficult.

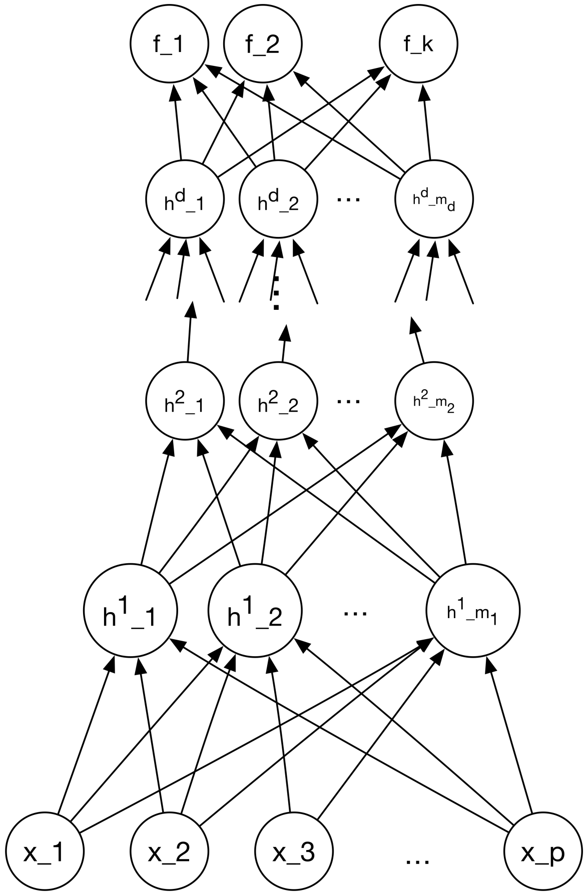
Deep Feed-Forward Neural Networks



We may interpret hidden layers as progressive derived representations of the input data.

Since we train based on a loss-function, these derived representations should make modeling the outcome of interest progressively easier.

Deep Feed-Forward Neural Networks



In many applications, these derived representations are used for model interpretation.

Stochastic Gradient Descent

Many data analysis methods are thought of as **optimization problems**

We can design gradient-descent based optimization algorithms that process data efficiently.

We will use linear regression as a case study of how this insight would work.

Case Study

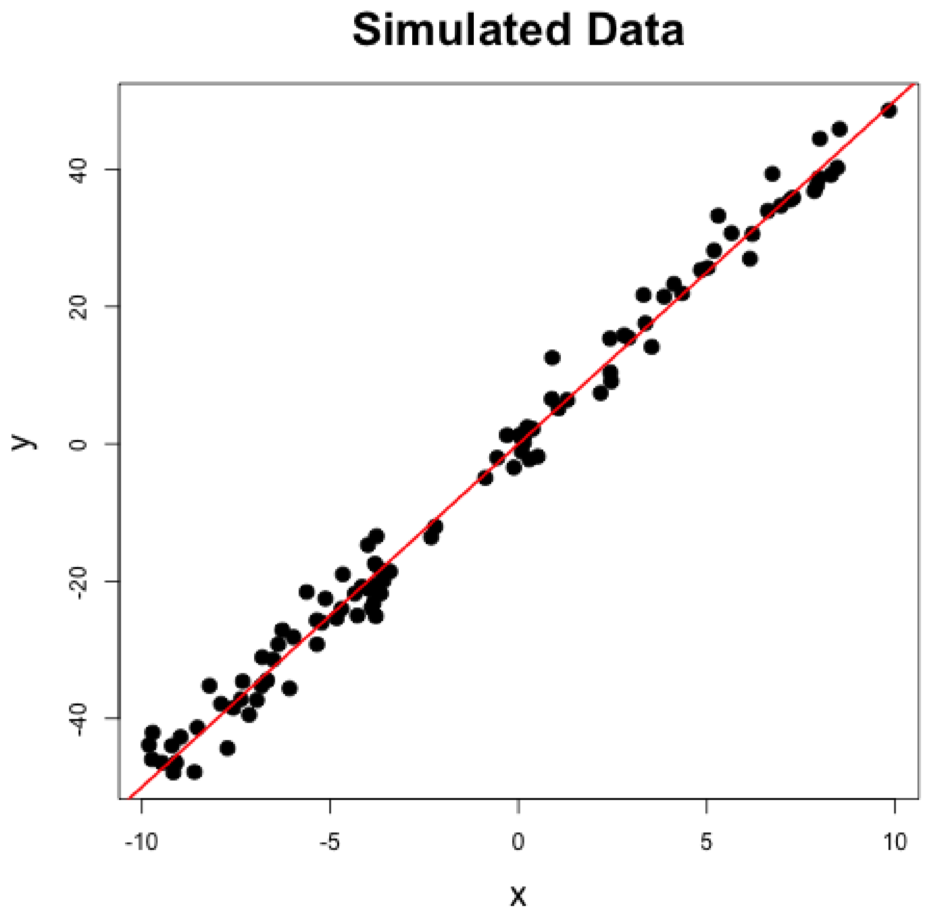
Let's use linear regression with one predictor, no intercept as a case study.

Given: Training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, with continuous response y_i and single predictor x_i for the i-th observation.

Do: Estimate parameter w in model $y = wx$ to solve

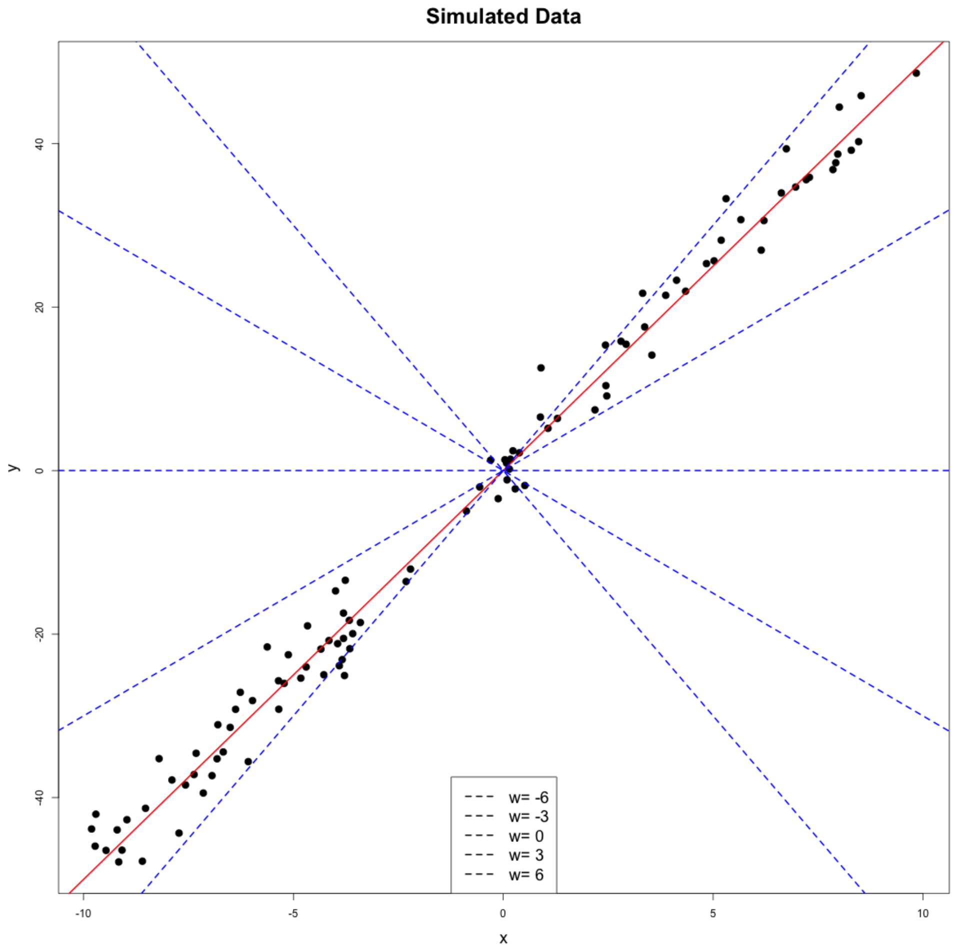
$$\min_w L(w) = \frac{1}{2} \sum_{i=1}^{n} (y_i - wx_i)^2$$

Case Study



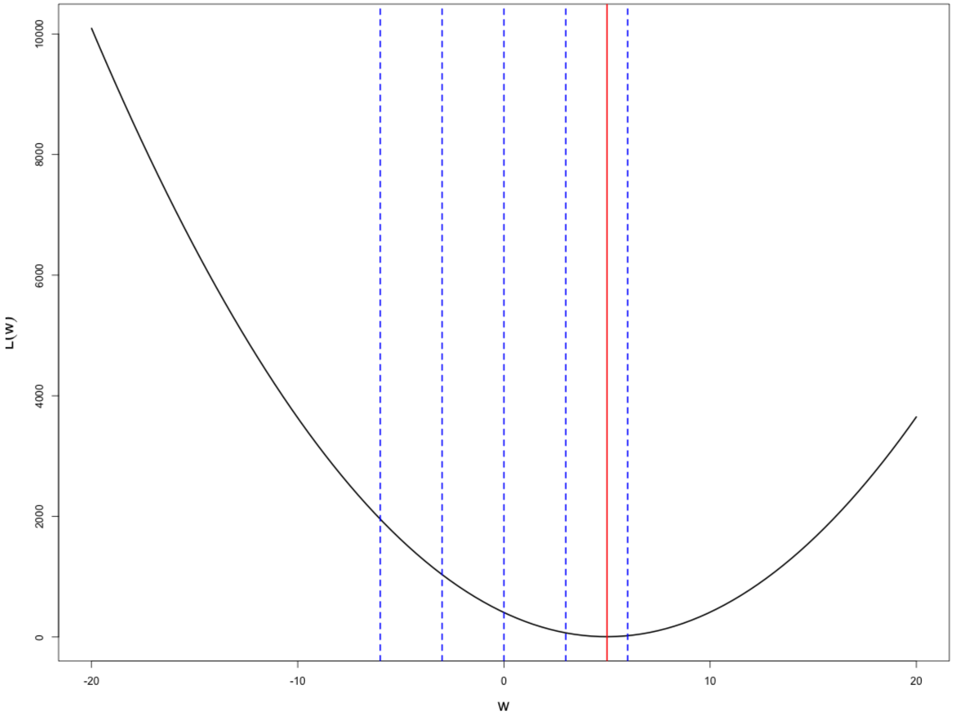
Suppose we want to fit this model to the following (simulated) data:

Case Study



Our goal is then to find the value of (w) that minimizes mean squared error. This corresponds to finding one of these many possible lines.

Case Study



Each of which has a specific error for this dataset:

Case Study

1) Loss is minimized when the derivative of the loss function is 0

2) and, the derivative of the loss (with respect to w) at a given estimate w suggests new values of w with smaller loss!

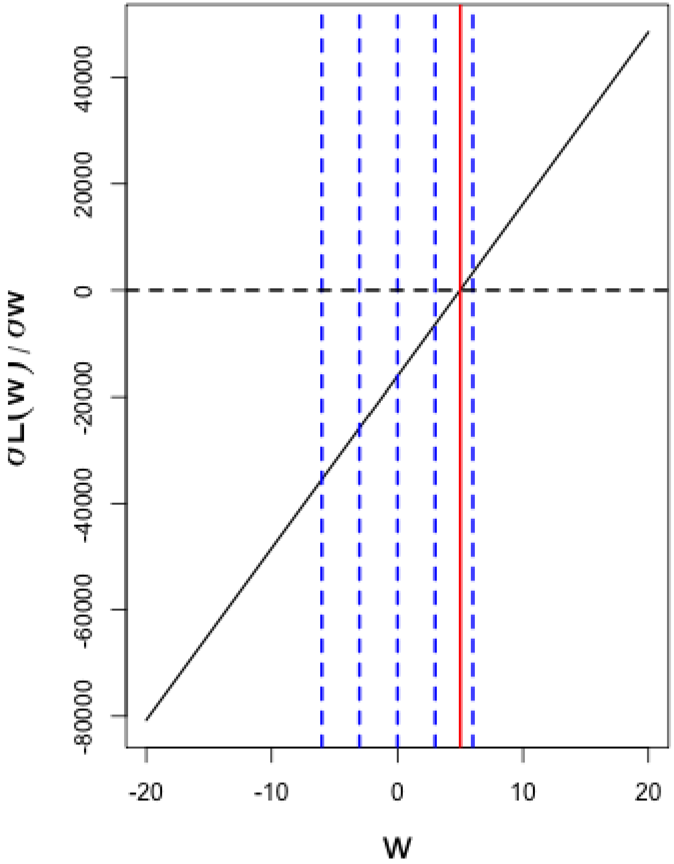
Case Study

Let's take a look at the derivative:

$$\frac{\partial}{\partial w} L(w) =$$

$$\frac{\partial}{\partial w} \frac{1}{2} \sum_{i=1}^{n} (y_i - w x_i)^2$$

$$= \sum_{i=1}^{n} (y_i - w x_i)(-x_i)$$



Gradient Descent

This is what motivates the Gradient Descent algorithm

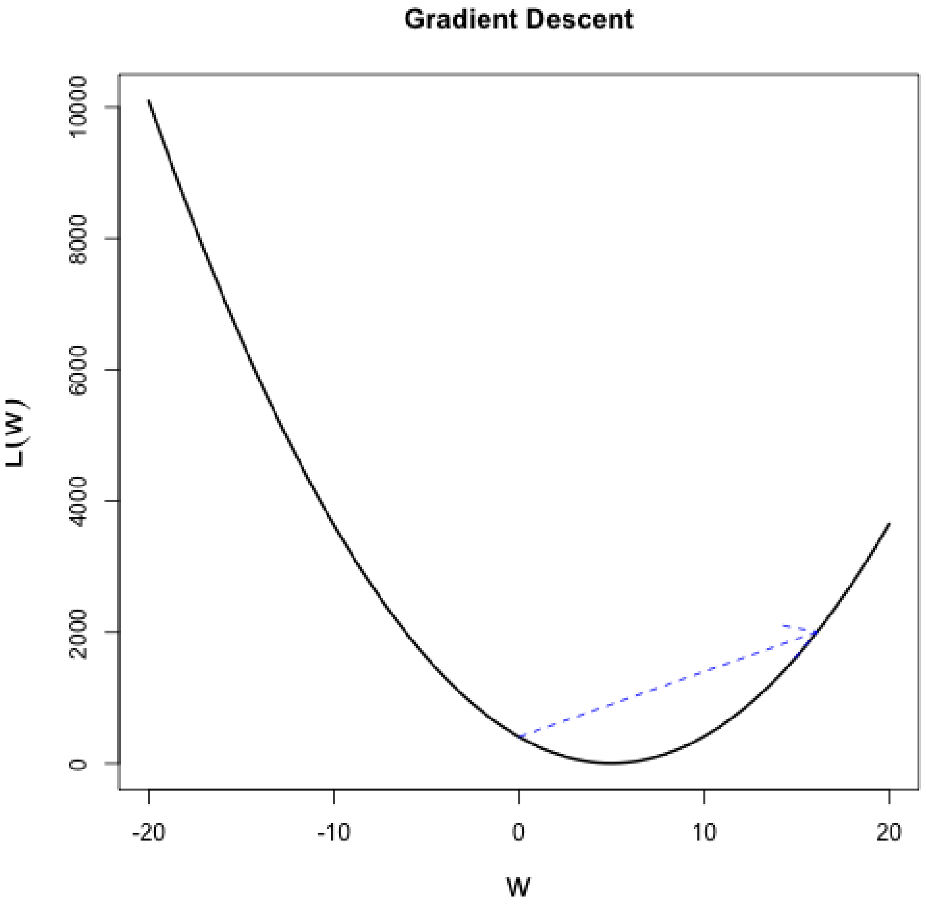
1. Initialize $w = \text{normal}(0, 1)$
2. Repeat until convergence
   - Set $w = w + \eta \sum_{i=1}^{n} (y_i - f(x_i; w)) x_i$

With $f(x_i; w) = w x_i$

Gradient Descent

The basic idea is to move the current estimate of w in the direction that minimizes loss the *fastest*.

Gradient Descent



Let's run GD and track what it does:

Gradient Descent

"Batch" gradient descent: take a step (update w) by calculating derivative with respect to *all* n observations in our dataset.

$$w = w + \eta \sum_{i=1}^{n} (y_i - f(x_i; w)) x_i$$

where $f(x_i) = w x_i$.

Gradient Descent

For multiple predictors (e.g., adding an intercept), this generalizes to the *gradient*

$$\mathbf{w} = \mathbf{w} + \eta \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i; \mathbf{w}))\mathbf{x}_i$$

where $f(\mathbf{x}_i; \mathbf{w}) = w_0 + w_1 x_{i1} + \cdots + w_p x_{ip}$

Gradient Descent

Gradiest descent falls within a family of optimization methods called *first-order methods* (first-order means they use derivatives only). These methods have properties amenable to use with very large datasets:

1. Inexpensive updates
2. "Stochastic" version can converge with few sweeps of the data
3. "Stochastic" version easily extended to streams
4. Easily parallelizable

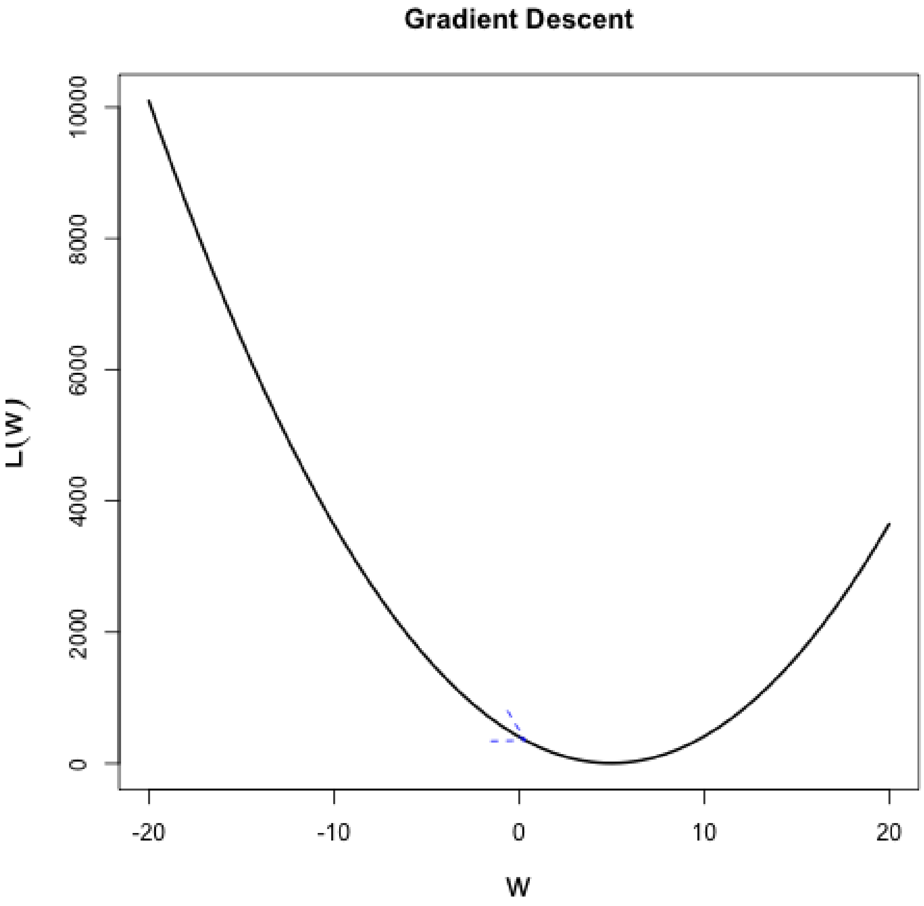
Drawback: Can take many steps before converging

Stochastic Gradient Descent

Key Idea: Update parameters using update equation *one observation at a time*:

1. Initialize $\mathbf{w} = \text{normal}(0, \sqrt{p}), i = 1$
2. Repeat until convergence
   - For $i = 1$ to n
   - Set $\mathbf{w} = \mathbf{w} + \eta(y_i - f(\mathbf{x}_i; \mathbf{w}))\mathbf{x}_i$

Stochastic Gradient Descent



Let's run this and see what it does:

Stochastic Gradient Descent

Why does SGD make sense?

For many problems we are minimizing a cost function of the type

$$\arg\min_f \frac{1}{n} \sum_i L(y_i, f_i) + \lambda R(f)$$

Which in general has gradient

$$\frac{1}{n} \sum_i \nabla_f L(y_i, f_i) + \lambda \nabla_f R(f)$$

Stochastic Gradient Descent

$$\frac{1}{n}\sum_i \nabla_f L(y_i, f_i) + \lambda \nabla_f R(f)$$

The first term looks like an empirical estimate (average) of the gradient at f_i

SGD then uses updates provided by a different *estimate* of the gradient based on a single point.

- Cheaper
- Potentially unstable

Stochastic Gradient Descent

In practice

- Mini-batches: use ~100 or so examples at a time to estimate gradients
- Shuffle data order every pass (epoch)

Stochastic Gradient Descent

This still presents challenges:

- Choosing proper learning rate
- How to update learning rate as iterations increase
- Per-parameter learning rates
- Avoiding local minima

Stochastic Gradient Descent

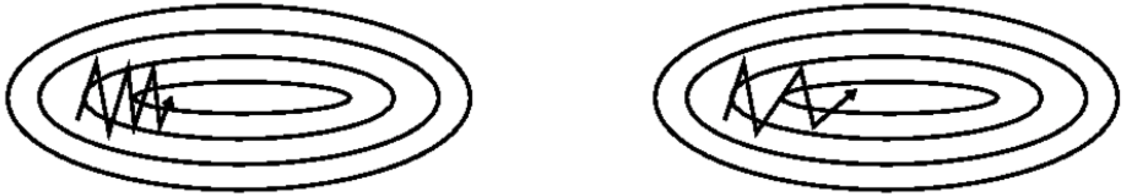
This still presents challenges:

- Choosing proper learning rate
- How to update learning rate as iterations increase
- Per-parameter learning rates
- Avoiding local minima

We will see modern derivatives of SGD that can address some of these challenges

Momentum

Avoid short-step oscillation in SGD by incorporating previous step information



(a) SGD without momentum (b) SGD with momentum

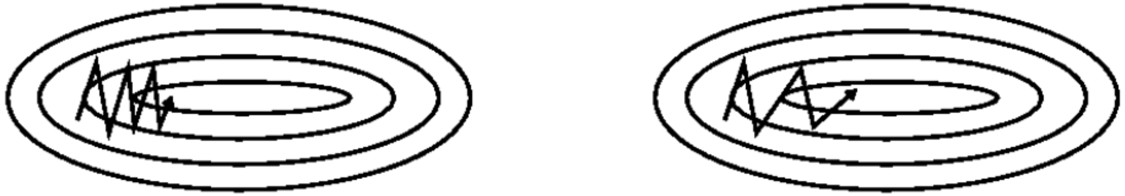
Figure 2: Source: Genevieve B. Orr

SGD:

$$w = w - \eta \nabla_w L(y, w)$$

Momentum

Avoid short-step oscillation in SGD by incorporating previous step information



(a) SGD without momentum

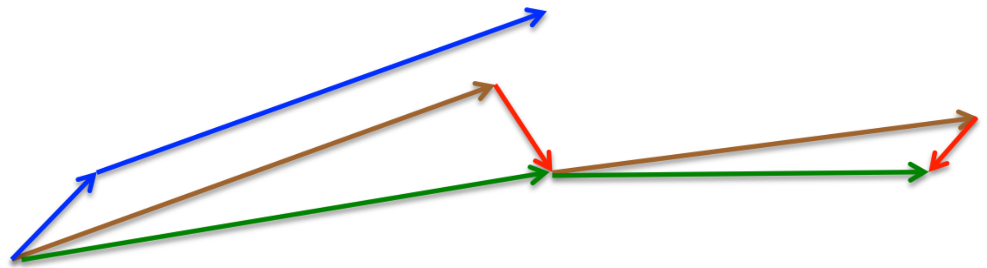
(b) SGD with momentum

Figure 2: Source: Genevieve B. Orr

SGD w/ momentum:

$$v_t = \gamma v_{t-1} + \eta \nabla_w L(y, w)$$
$$w = w - v_t$$

Accelerated Momentum



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_w L(y, w - \gamma v_{t-1})$$

$$w = w - v_t$$

Adaptive Moment Estimation (Adam)

Computes adaptive learning rates for each parameter in model

Updates based on exponentially decaying average of past squared gradients (for adaptation)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_w L(y, w))^2$$

Adaptive Moment Estimation (Adam)

Computes adaptive learning rates for each parameter in model

Updates based on exponentially decaying average of past squared gradients (for adaptation)

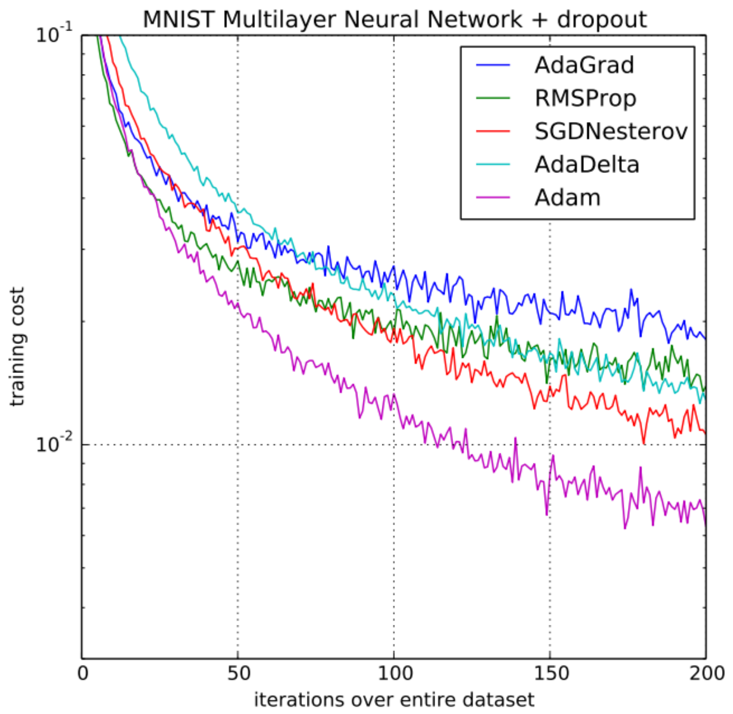
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_w L(y, w))^2$$

And past gradients (for momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_w L(y, w)$$

Adaptive Moment Estimation (Adam)

$$w = w - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$



Summary

- Improved stablilty by exploiting 'estimation' interpretation of gradient descent
- Stabilize by aggregating estimates of gradients
- Scaling by variance of estimates
- Intepretation of algoritms in terms of estimators can greatly improve performance