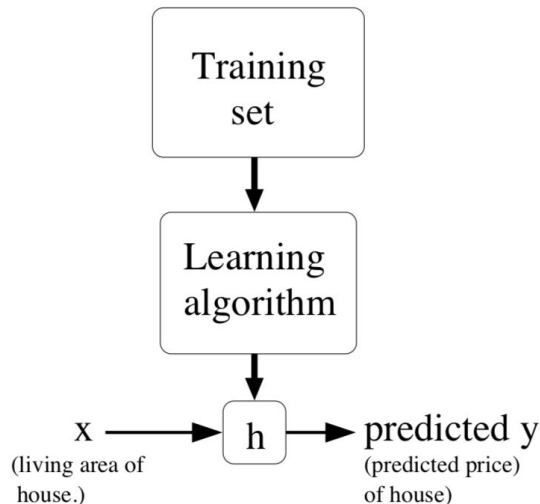


Function Fitting Network

Teddy Vallar

Introduction to ML and Artificial Neural Networks

Linear Regression



A generalized linear model:

$$\hat{y} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

OR a more concisely vectorized (short) form of the model/hypothesis:

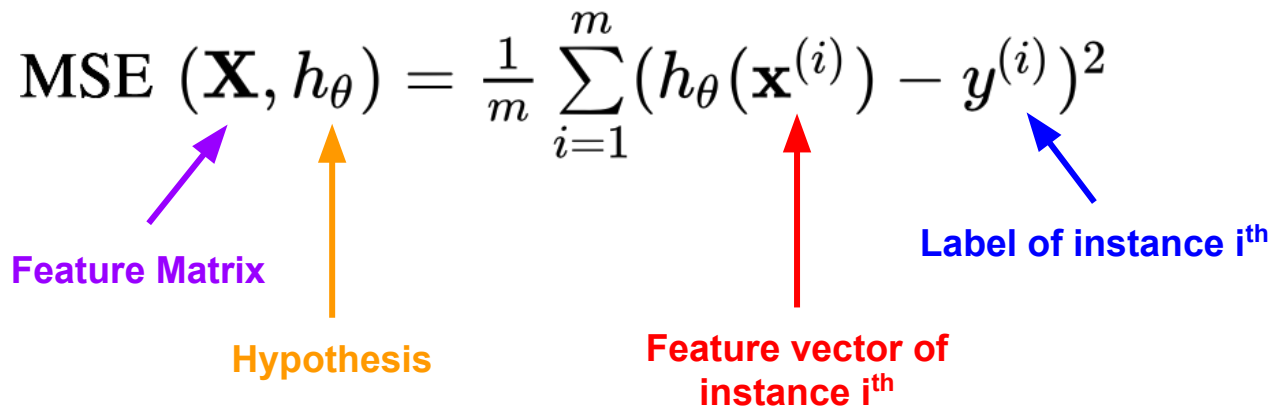
$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^T \mathbf{x}$$

Hypothesis function

1 x (n+1) Parameter Vector

(n+1) x 1 Feature Vector

Loss Function → MSE (Mean Square Error)

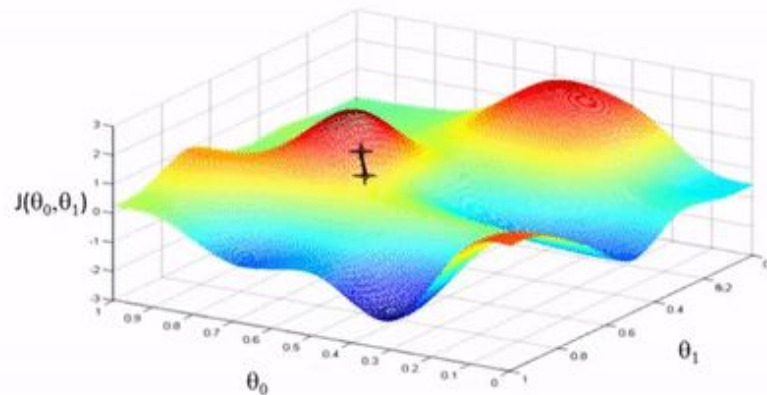
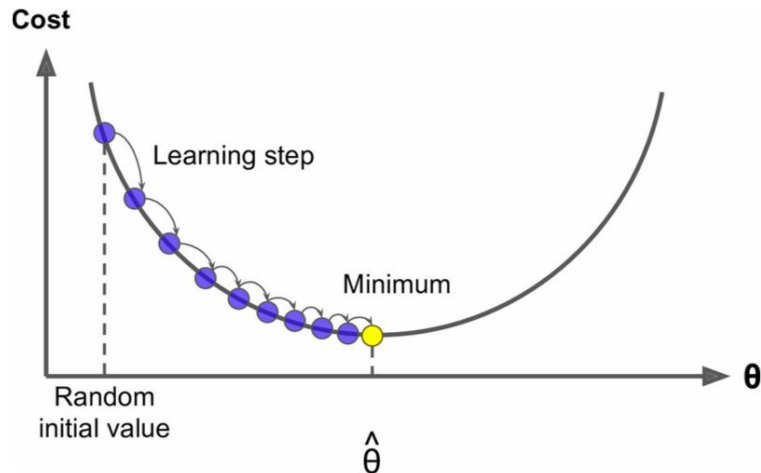
$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$$


The diagram illustrates the components of the Mean Square Error (MSE) formula. It features the equation $\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$. Four colored arrows point to specific parts of the equation: a purple arrow points to \mathbf{X} with the label 'Feature Matrix'; an orange arrow points to h_{θ} with the label 'Hypothesis'; a red arrow points to $\mathbf{x}^{(i)}$ with the label 'Feature vector of instance i^{th} '; and a blue arrow points to $y^{(i)}$ with the label 'Label of instance i^{th} '.

We will **derive** this into a loss function and try to **minimize** it

Gradient Descent (GD)

- Generic optimization algorithm to find optimal solution to wide range of problems.
- Tweak parameters **iteratively** in order to **minimize** a loss function.
- Determined by a **learning rate** hyperparameter



Batch Gradient Descent (BGD)

- Calculate how much the loss function will change if we change the parameter just a bit (ie. partial derivatives)
- Same as: “what is the slope of the mountain if I take a step to the east?” then as the same question for other directions
- Use all (or part) of training data → batch

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), (j = 1 \dots n)$$

Learning Rate

Cost Function

BGD Formulation

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), (j = 1 \dots n)$$

Expand loss function:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} \left(\frac{1}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)})^2 \right)$$

Take partial derivative:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Learning Rate too small

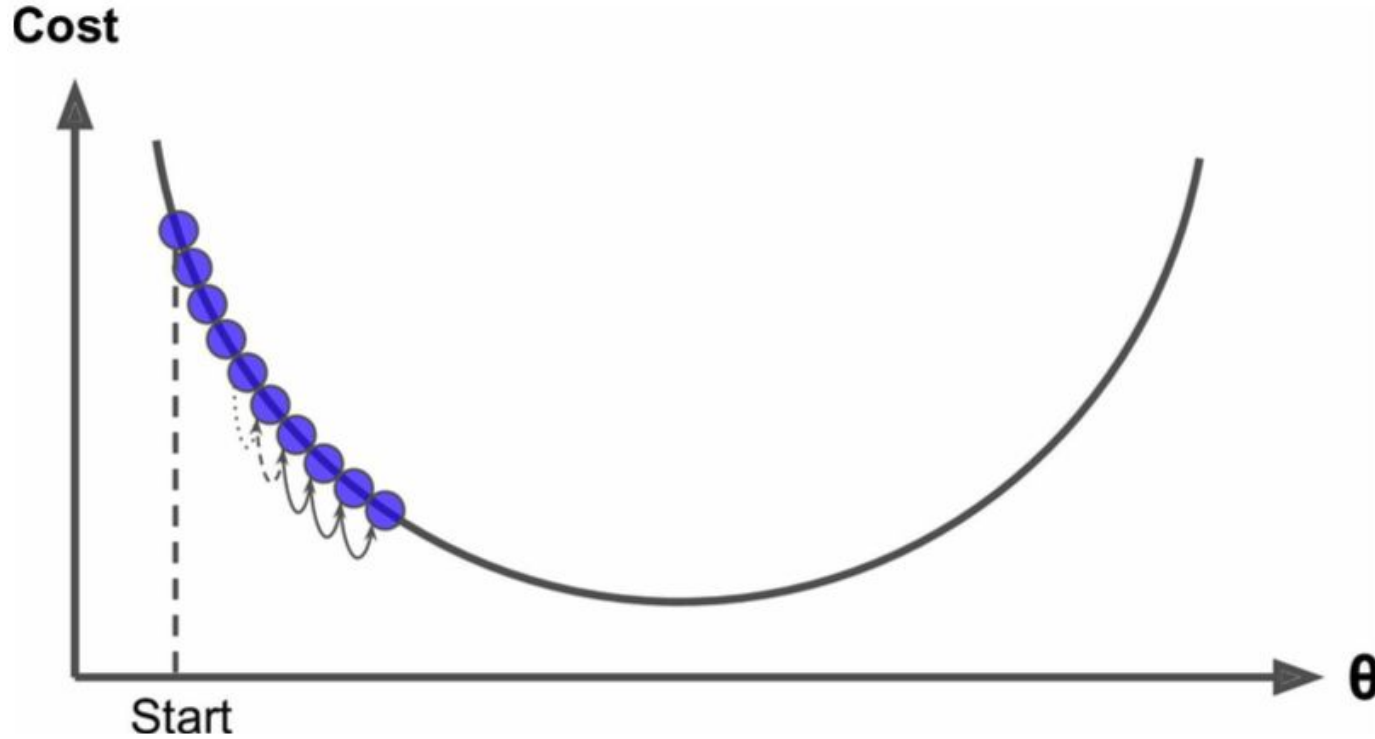


Figure 4-4. Learning rate too small

Learning rate too big

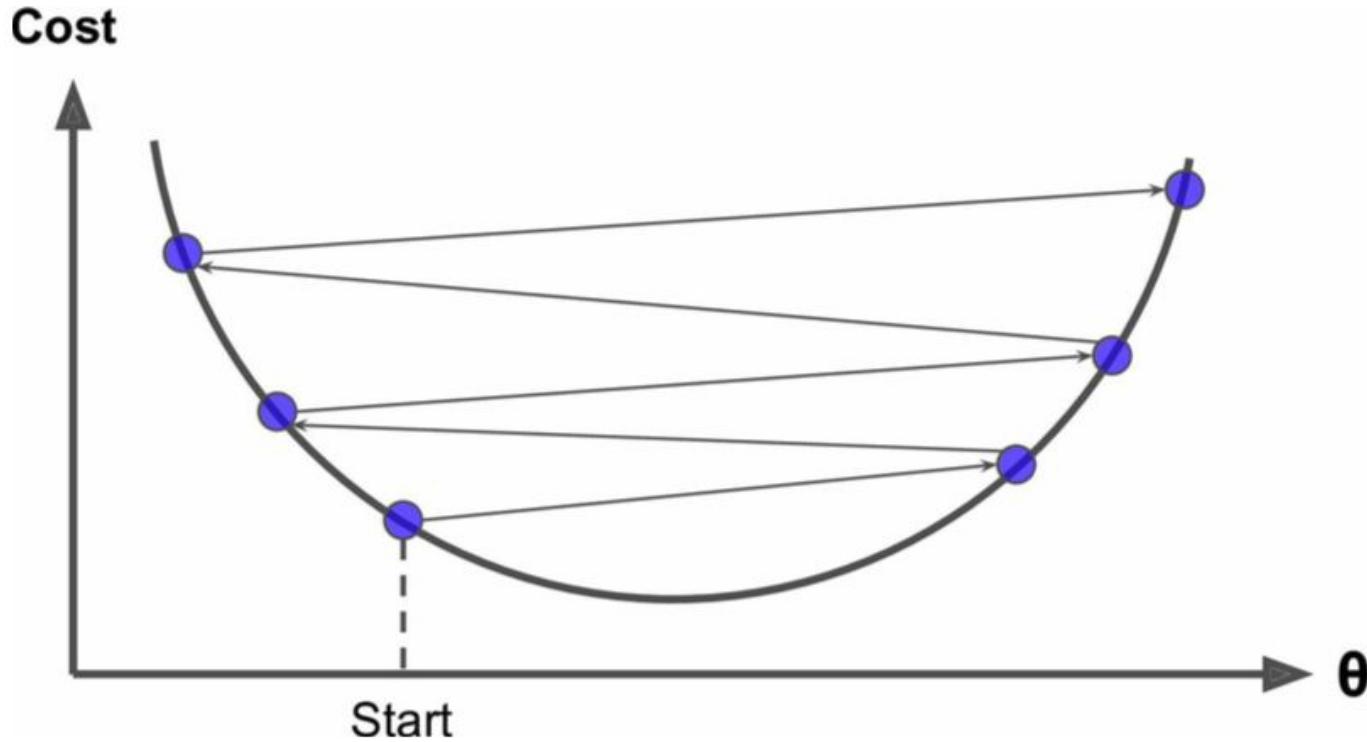
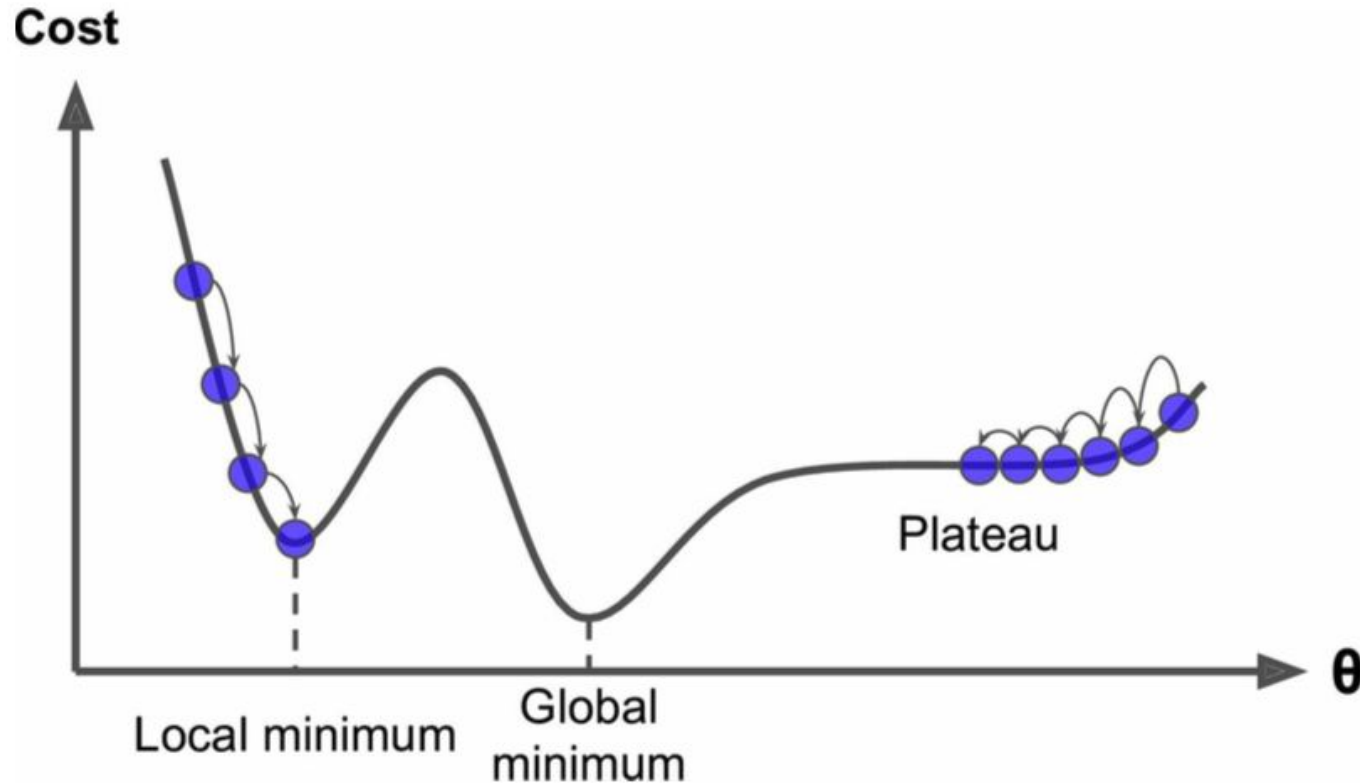
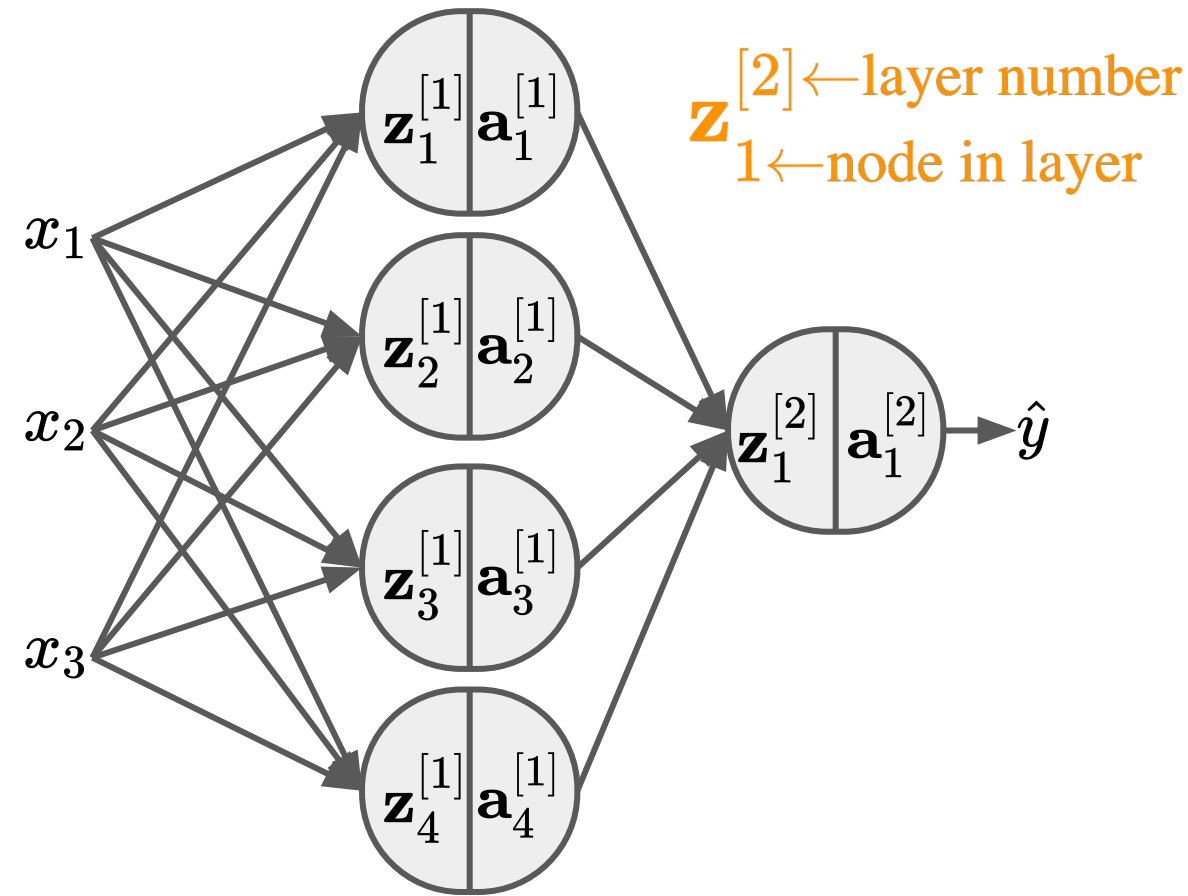


Figure 4-5. Learning rate too large

Local minimum vs. global minimum



Representation of a MLP (ANN)



$$\mathbf{z}_1^{[1]} = \mathbf{w}_1^{[1]T} \mathbf{x} + b_1^{[1]}$$

$$\mathbf{a}_1^{[1]} = \sigma(\mathbf{z}_1^{[1]})$$

$$\mathbf{z}_2^{[1]} = \mathbf{w}_2^{[1]T} \mathbf{x} + b_2^{[1]}$$

$$\mathbf{a}_2^{[1]} = \sigma(\mathbf{z}_2^{[1]})$$

$$\mathbf{z}_3^{[1]} = \mathbf{w}_3^{[1]T} \mathbf{x} + b_3^{[1]}$$

$$\mathbf{a}_3^{[1]} = \sigma(\mathbf{z}_3^{[1]})$$

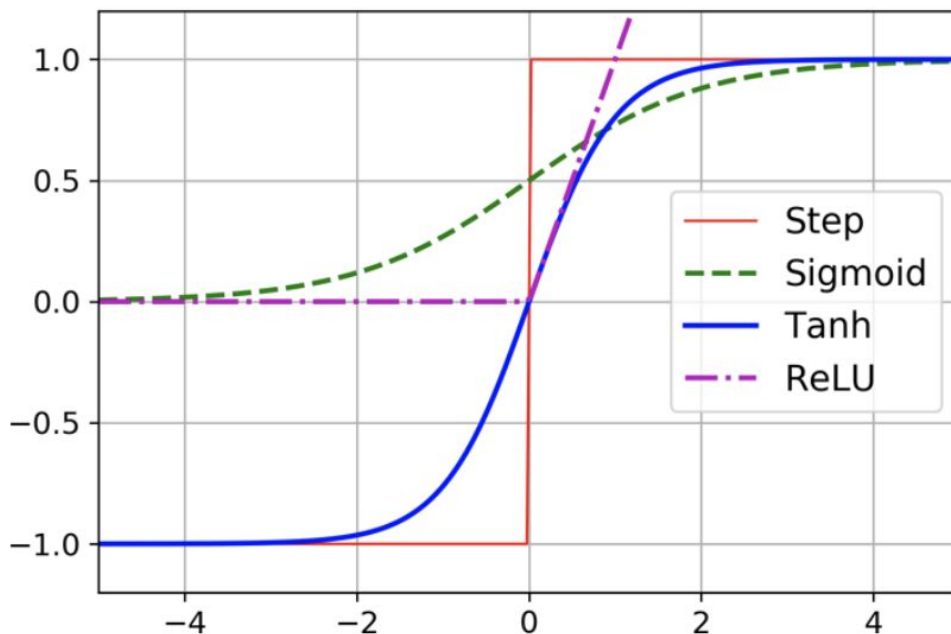
$$\mathbf{z}_4^{[1]} = \mathbf{w}_4^{[1]T} \mathbf{x} + b_4^{[1]}$$

$$\mathbf{a}_4^{[1]} = \sigma(\mathbf{z}_4^{[1]})$$

Activation function $a = \sigma(z)$

There are a number of activation functions available:

Activation functions



Step: $a = 1$ if $z > 0$,
 0 if $z < 0$

Sigmoid: $a = \frac{1}{1+e^{-z}}$

Tanh: $a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

ReLU: $a = \max(0, z)$

Loss function

aka. cost function, objective function, error function

It is the quantity that will be minimized during training and represents a measure of success for the task at hand.

Examples: Square Loss, Log Loss, Hinge Loss, Gini, Entropy

Training / Learning the weights' value

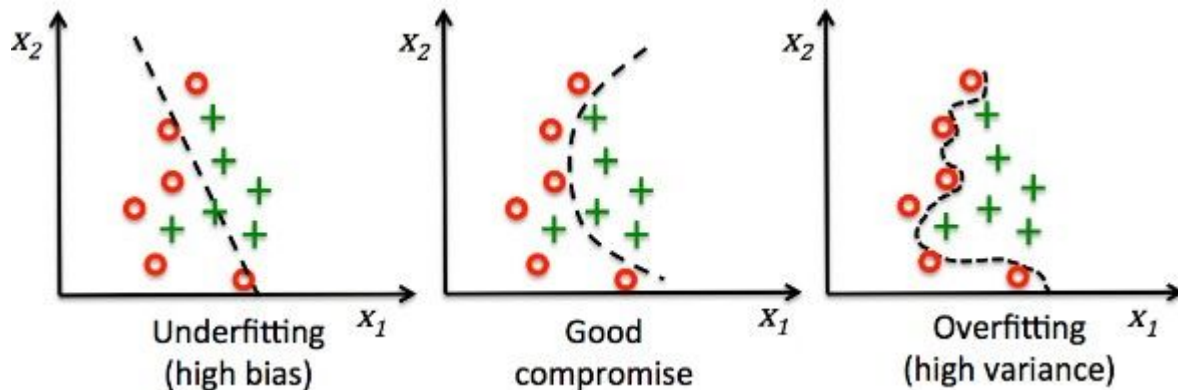
Using **backpropagation** (groundbreaking article by Rumelhart et al, 1986):

- **Forward Pass:** Feeds a training instance to the network and computes the output of every neuron
- **Backward Pass:** Measures the network error, and computes the error contributions came from each neuron in previous hidden layer
- **Gradient Descent:** Runs Gradient Descent on all connection weights using the measured error

Overfitting

Overfitting is when the network has learned the training data too well, meaning it is not generalizable

Can typically happen when there are too many training iterations, the training data is not generalized, there are too few training examples, or the model is too complex



The Goal of this Network

Fitting Equations

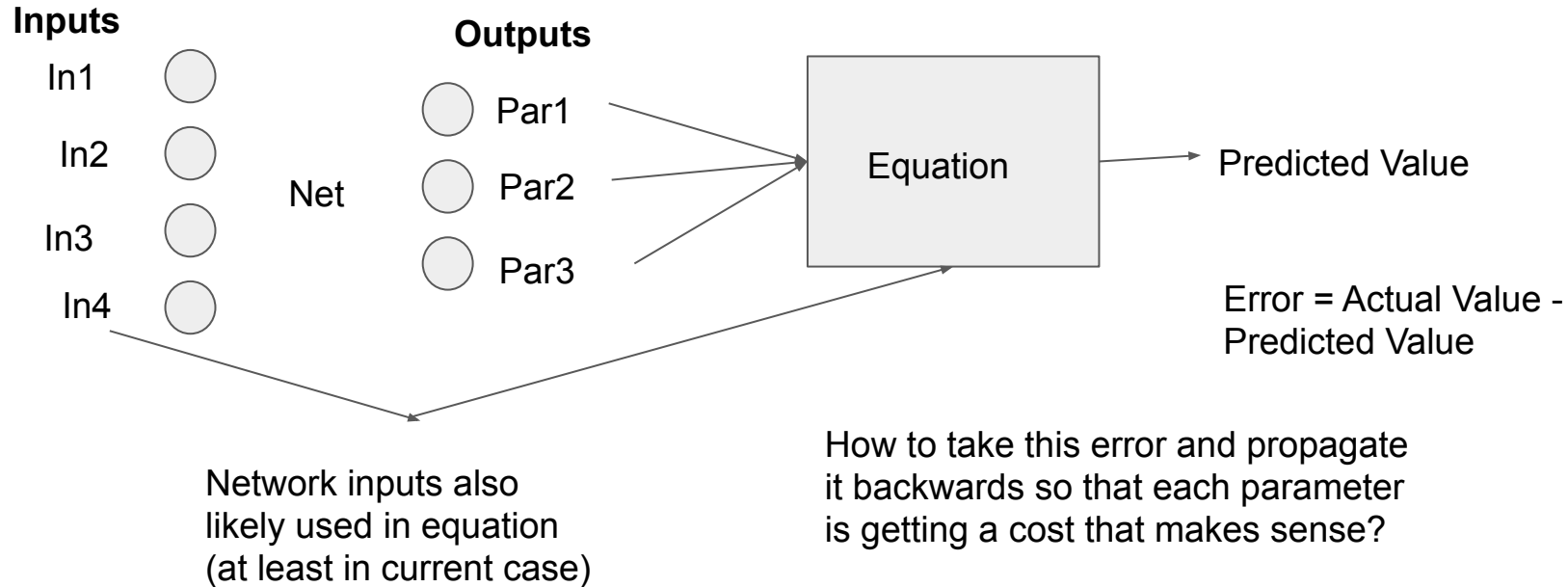
Typically, when using neural networks, the network trains using the expected output values which are known in the file.

For the network we are building, the network outputs are the inputs for an equation and we are given the correct output of that equation, meaning we can not get loss for each output neuron and backprop like other networks do. The only “cost” we can access is the difference between expected equation output and predicted.

The goal is to build a network that can learn to predict the parameters for any equation, not just the physics Cross-Section equation I used during my work.

Visual Description

Training the network is unconventional



Initial Attempts (before TensorFlow)

When using simpler equation:

- Use the analytical derivative, cost for each parameter is:
 - $\text{Cost_Par1} = \text{Partial Derivative With Respect to Par1 (given all other parameter values as constant)} \times \text{Error}$

However, analytical derivative is nearly impossible when the equation gets too complex

- Numerical Derivative: treat partial derivative for Par1 as equation with Par1 value input minus equation with Par1 - 0.1 (or less) value input
 - Very slow if equation is computationally intensive, may not work depending on equation if numerical derivative step is too large or too small

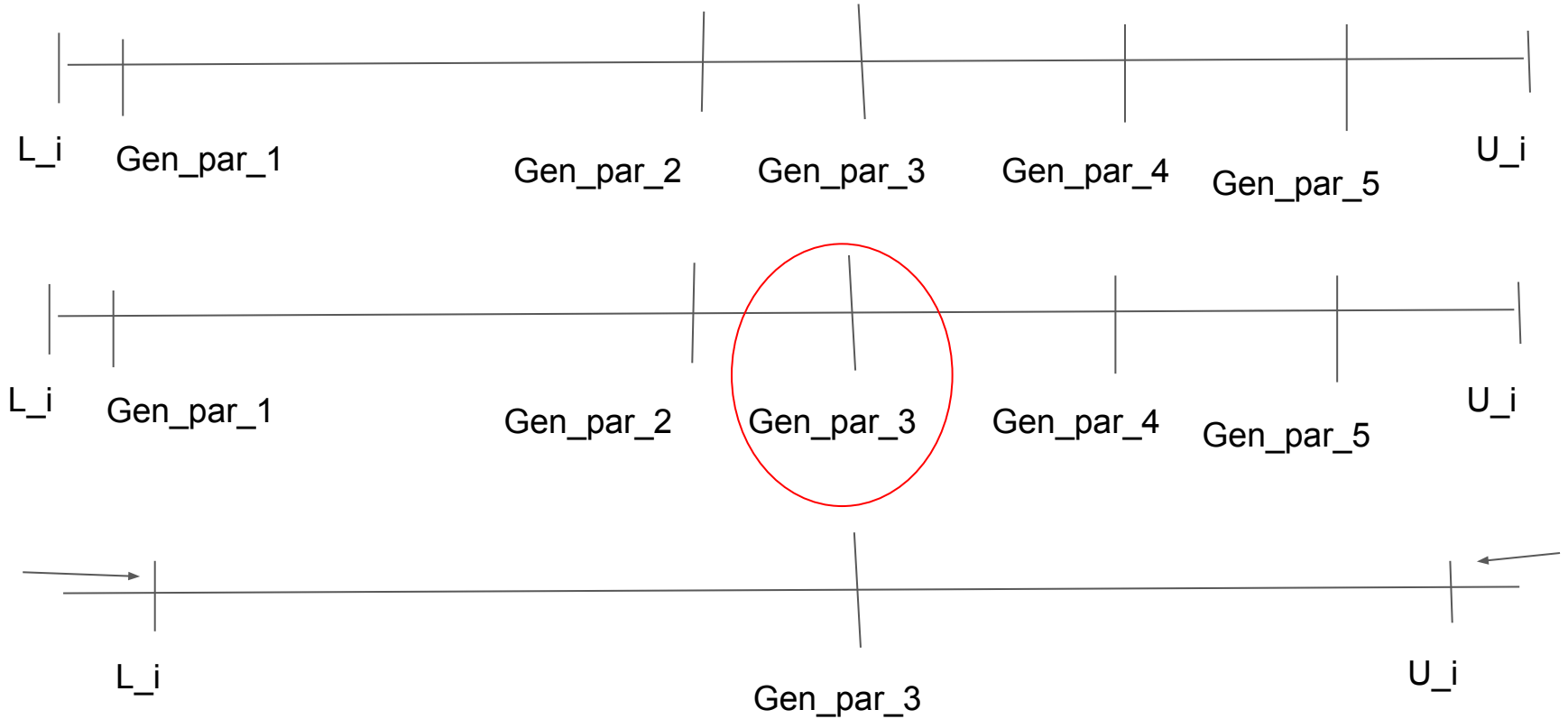
Initial Attempts (before TensorFlow)

Moving Window (as described in report):

Eventually a shrinking window sampling method was settled on. In this method each parameter (1, 2, 3) has a low value, l_i , and an upper value, u_i where i is the parameter number. Each mini-batch, 5 values are selected through uniform distribution between l_i and u_i for each i . These 3 sets of 5 values are then used to create 125 permutations of possible parameters. Each of the 125 sets of parameters are then used to approximate each of the points in the mini-batch. The parameters that best fit the points in the mini-batch are then the ones that are back-propagated. Then the parameter ranges close around the parameter values chosen through an operation like $u_i = 0.99 * u_i + 0.01 * p$ where p is the chosen parameter value.

By saying the best parameters are backpropagated, I mean those parameter values are used in the sense $\text{error} = \text{Estimated_Par} - \text{Chosen_Par}$. The initial parameters outputted from the equation are also used if the output of the equation using them is “better” (closer to actual value) than any of the generated parameter sets (Also very slow but a lot more accurate than numerical in most cases)

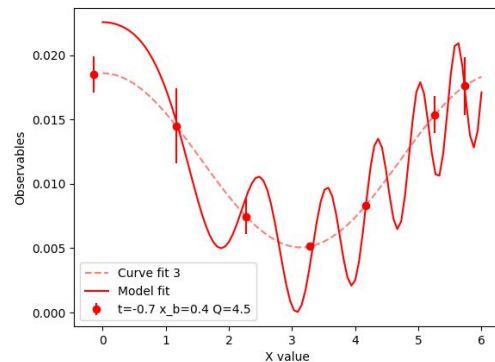
Moving Window Visual



Cost/Loss Function

While the network uses tradition MSE cost function, there were some additions and changes based on problems in training.

- Normalizing for Curve Length:
 - As you can see in the example to the right, while the curve fit through a few of the points well, the curve was overfitting in a sense and did not match what was expected
 - To account for this, the cost was adjusted in such a way that the actual length of the curve with respect to the linear distance between the points was added as a factor
 - This would not work with all functions, however, as it was dependent on the results I was getting at the time



TensorFlow Version

While the moving window worked best for a Numpy Network, there was a function built into Tensorflow that could do it better.

When the function was turned into a Tensor based equation, the TF function “gradients” was used.

Gradients takes in the loss value and trainable variables then returns the gradients of loss function with respect to the trainable variables

This gradient is then used to train the network

Results

Moving Window Results

Unfortunately, there were issues with the Tensorflow equation and outputs so I could not easily graph the results

Here are some results from the numpy version that describe the current (even in TF) problem well

**Line fits, points fit,
parameters are wrong**

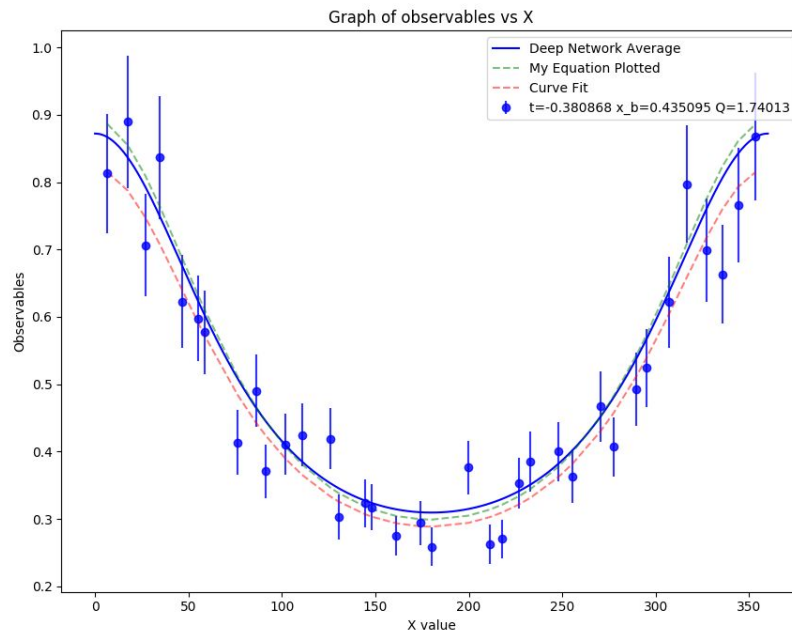


Table 1: Network 1 Results			
Type	ReH	ReE	ReHT
Correct	0.677287	1.0538	0.908049
Curve Fit	$0.4964 + / - 0.5279$	$1.2793 + / - 0.78$	$0.12243 + / - 3.918$
Deep Network	$1.1296 + / - 0.014$	$1.1006 + / - 0.031$	$0.8609 + / - 0.0939$

Description of Tensor Network, Data, and running the code

Data

Data is pulled from the csv as shown below.

Training Variables used: k, Q2, x_b, t.

Each set of unique k, Q, x_b, t values has 36 phi_x measurements. When training for 1 set of ReH, ReE, and ReHTilde values, 36 will be used

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Index	k	QQ	x_b	t	phi_x	L	F	errF	F1	F2	ReH	ReE	ReHTilde
2	0	2.75	1.72395	0.355194	-0.20537	4.6426	1	10.4022	1.14418	0.631324	0.95597	0.631324	0.95597	0.852383
3	1	2.75	1.72395	0.355194	-0.20537	12.2379	1	10.158	1.11654	0.631324	0.95597	0.631324	0.95597	0.852383
4	2	2.75	1.72395	0.355194	-0.20537	24.5827	1	9.22548	1.01274	0.631324	0.95597	0.631324	0.95597	0.852383
5	3	2.75	1.72395	0.355194	-0.20537	37.8822	1	7.74295	0.852762	0.631324	0.95597	0.631324	0.95597	0.852383
6	4	2.75	1.72395	0.355194	-0.20537	44.7016	1	6.51672	0.71587	0.631324	0.95597	0.631324	0.95597	0.852383
7	5	2.75	1.72395	0.355194	-0.20537	52.1727	1	7.56232	0.832748	0.631324	0.95597	0.631324	0.95597	0.852383
8	6	2.75	1.72395	0.355194	-0.20537	69.6681	1	6.12349	0.6707	0.631324	0.95597	0.631324	0.95597	0.852383
9	7	2.75	1.72395	0.355194	-0.20537	71.3469	1	5.33995	0.586726	0.631324	0.95597	0.631324	0.95597	0.852383
10	8	2.75	1.72395	0.355194	-0.20537	84.2394	1	5.02968	0.553433	0.631324	0.95597	0.631324	0.95597	0.852383
11	9	2.75	1.72395	0.355194	-0.20537	101.143	1	4.3169	0.474285	0.631324	0.95597	0.631324	0.95597	0.852383

Data - Generation

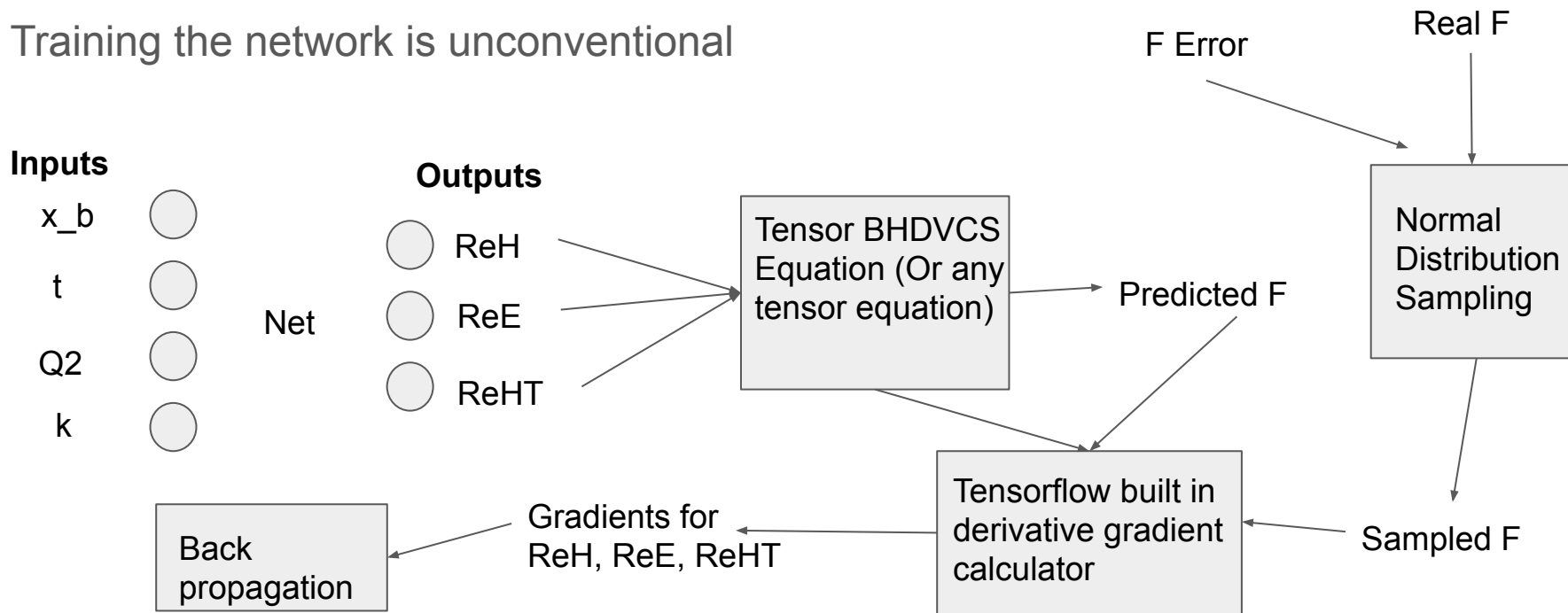
Most values are untouched when pulled from csv.

F value(s) used during training are generated based on the F and errF values for each data point. Instead of using the real F value during backpropagation, an estimated F value generated by sampling is used

The sampling is done through a normal distribution using the F value as the average and errF to set the standard deviation.

Training

Training the network is unconventional



Running Network

Run “python run_network.py” in terminal after full installation

This will prompt for three inputs:

- **Name of Run:** Name to store trained network under
- **Number of Epochs:** Number of times to train through full training dataset
- **Line Number:** The set of 36 points in data file to use for training, starts with 0 and -1 is reserved for training on all data

This will run a basic training loop from loading the data to training the network and printing the averaged results of the network

No graph analysis at the moment because of issues with graphing tensors that I could not solve, however all of the necessary structures to create graphs should be available in code

Things to Work on or Try

- Normalizing inputs:
 - Since neural networks function on linear regression, the problems it is suitable for are largely linear in nature
 - Normalize inputs so that they have a linear relationship with expected outputs
- More experimentation with other types of cost functions
 - May have to be equation specific, otherwise run into issues like seen with moving window results and tensor network

Questions?