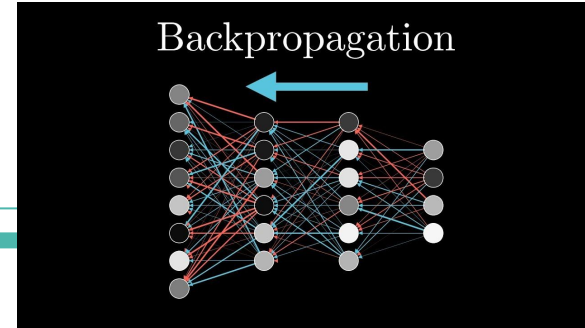
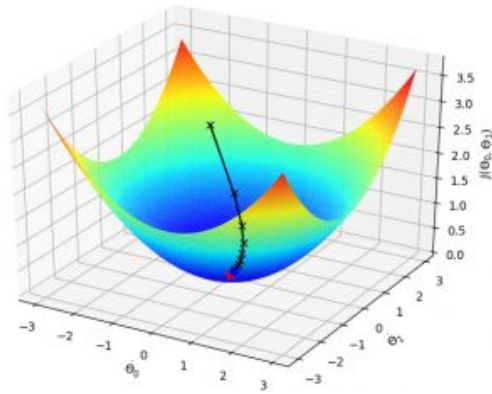

Gradient Descent and Backpropagation

Amogh Tiwari



Gradient Descent

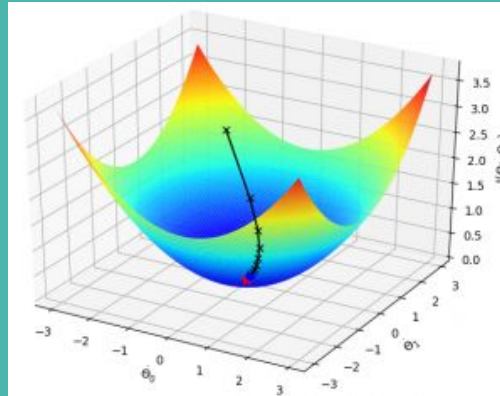


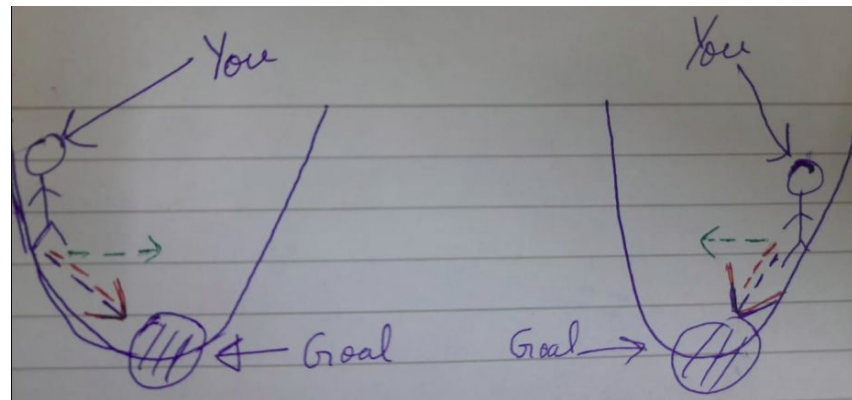
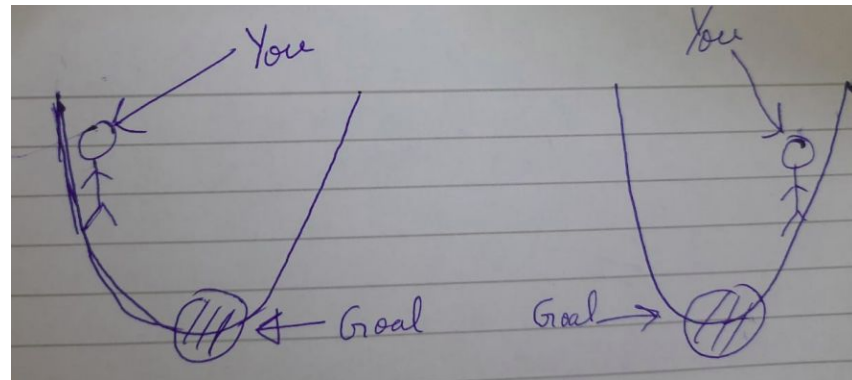
Image Credits: <https://ai-diary-by-znreza.com/neural-network-demystified-part-lli>

Motivation

- Most ML algorithms work by trying to minimize a loss function
- The loss function is defined in a way such that “favourable” outcomes (eg. Machine’s prediction matching the actual value) are rewarded and “unfavourable” outcomes are penalized
- Defining a reliable loss function is challenging! (Eg. Consider SVM)
- But the job doesn’t end once we have defined a loss function
- We also need to minimize it

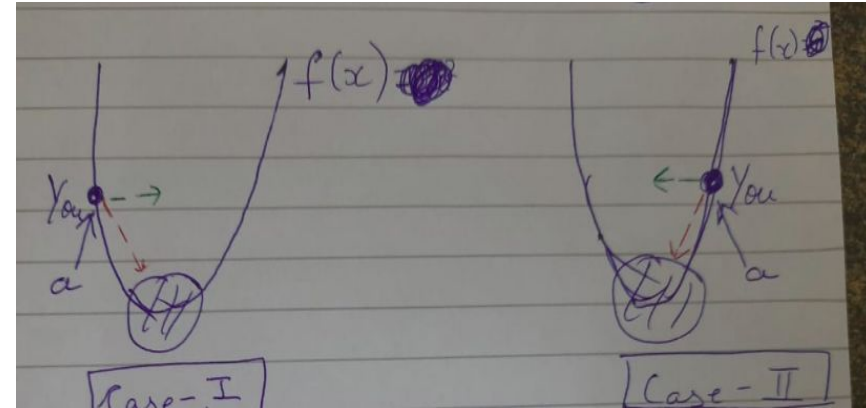
Puzzle

- Suppose you are on top of a hill and want to reach its bottom most point
- Then, you would want to move like this:



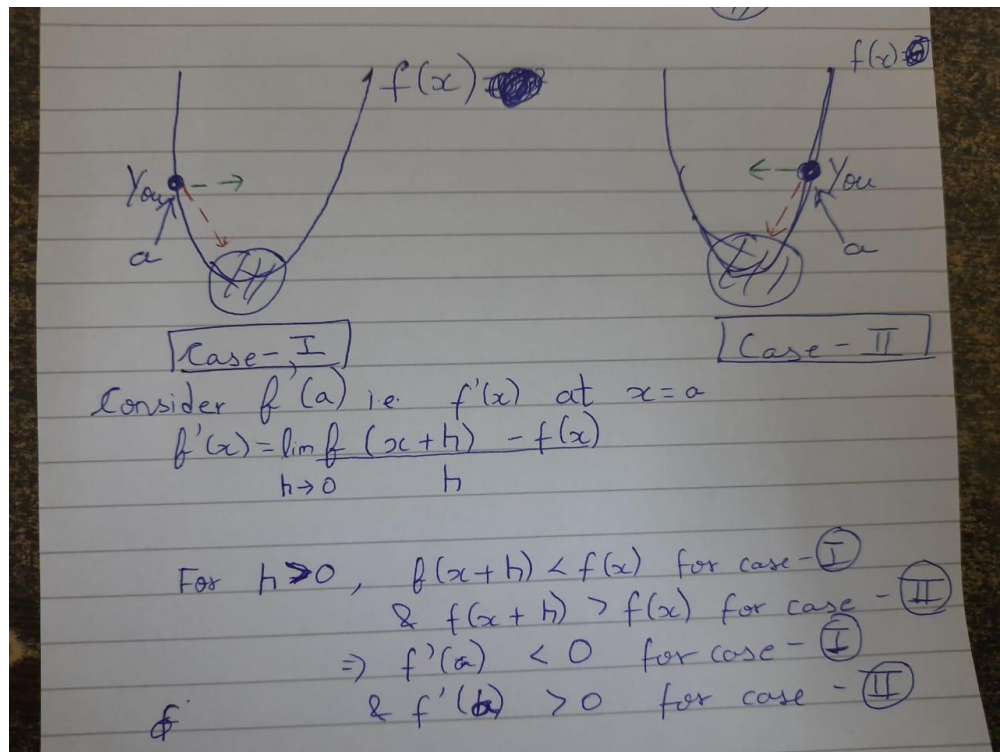
How do we represent this mathematically

- Define the elevation of the hill using a function $f(x)$. So, we have: $y=f(x)$
- The intended path of motion is defined by the direction of x (change in y is a function of changing ' x ')
- In case-1, we want to move in the direction of increasing ' x ', while in case-2, we want to move towards decreasing ' x '



Can we have a single rule to decide the direction ?

- Yes. Using derivatives
- If derivative is negative, move towards direction of increasing 'x', else, move towards direction of decreasing 'x'.
- Further, on reaching the minima, derivative = 0. So you stop (**Important**)
- **Note:** Try the same for $h < 0$ too



Does this rule always work? Why?

- Yes, this rule works always
- This is because: Derivative gives us the “rate of change” of a function i.e. if derivative is +ve, it means that on increasing ‘x’, the function has a +ve change i.e. the functions value is increasing on increasing ‘x’. Therefore, to move towards the direction where function takes minima, we must move in a direction where ‘x’ decreases
- Same argument can be made for negative derivatives (**Pause and think about this!**)

(Additional): Food for Thought

- Why does derivative give the rate of change of a function? (**Hint:** Think about the fundamental definition of derivatives)

(Additional): Food for Thought

- Why does derivative give the rate of change of a function?
- **Answer:** They are defined that way!
- $\lim_{h \rightarrow 0} [f(x+h) - f(x) / h]$ is essentially telling us how the function is “changing” values when we modify ‘x’ by a small amount. Further, the magnitude of it tells us the “rate” of change

Fun Fact: Gradient Descent algorithm was first introduced in 1847 by Cauchy

Examples

- Cubic Function
- Quartic Function
- 2D function: $Z = f(x,y)$
[Provide visualization]

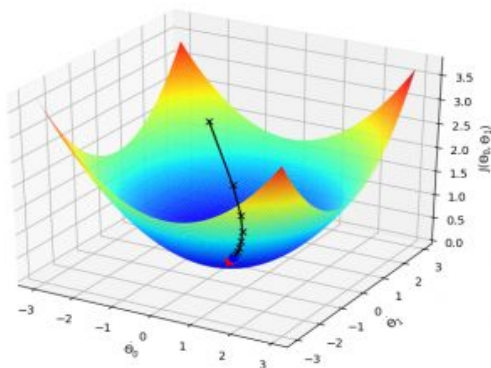
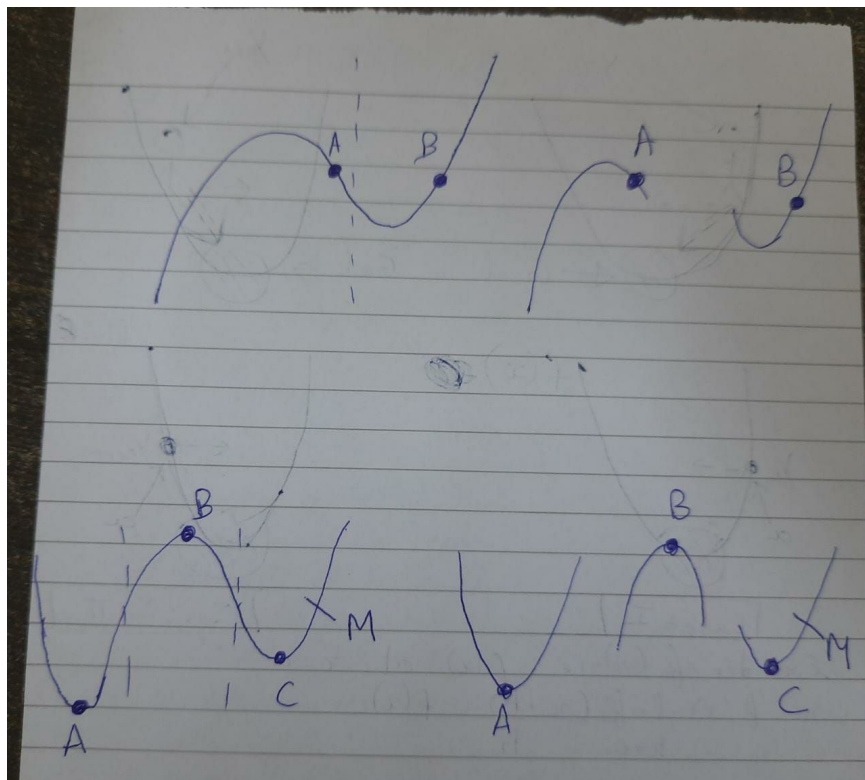


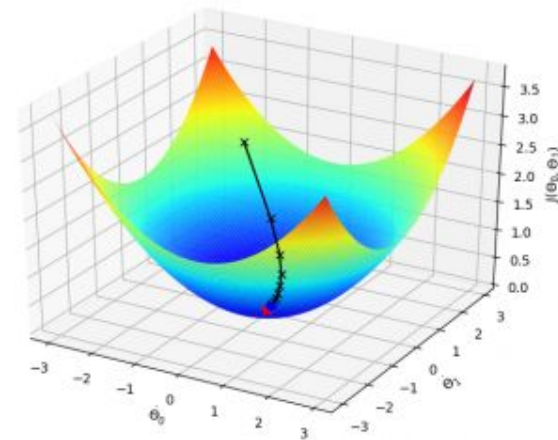
Image Credits:

<https://ai-diary-by-znreza.com/neural-network-demystified-part-lli>



Generalization to Gradient

- What do we do if we have a 2D function: $Z = f(x,y)$?
- Consider the components of our intended direction of motion along 'x' and 'y'
- Keep 'y' constant and find the derivative along 'x'
- Keep 'x' constant and find the derivative along 'y'
- Move in the direction given by the above vector
- This is same as taking the gradient

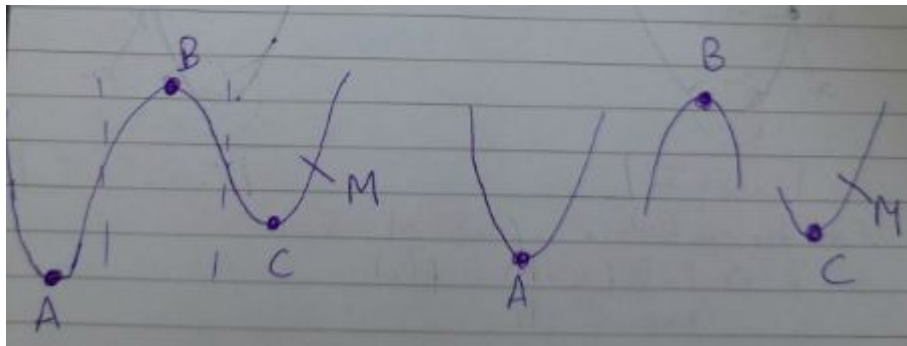


Target dirⁿ of motⁿ along x: $-\partial f / \partial x$
y: $-\partial f / \partial y$

"Overall" target dirⁿ of motion = $\begin{bmatrix} -\partial f / \partial x \\ -\partial f / \partial y \end{bmatrix}$
 $= -\begin{bmatrix} \partial f / \partial x \\ \partial f / \partial y \end{bmatrix}$
 $= -\nabla f$

Limitations

- Local minima
 - Suppose you start from M
 - Our method will guide you till C
 - But you will get “stuck” at C as gradient becomes 0 there
 - Solutions:
 - Use “momentum”
 - Use “noise” while walking (happens “automatically” as part of Stochastic GD)
- Non-differentiability!
 - Choose a different function
 - Use local differentiability or any such property
 - Define a differentiable approximation for the function of interest (easier said than done!)
 - Give up !!



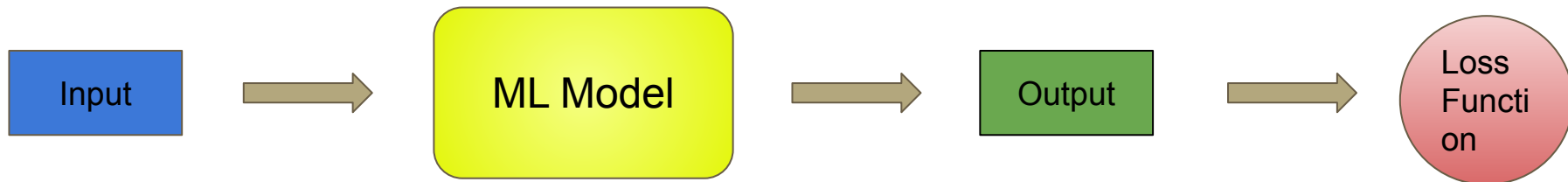
Code!

- Summary **till here**:
 - Most ML algos involve minimizing a given loss function
 - To find the value at which a given function attains a minima, we should move towards the direction opposite to the gradient of the function
 - Limitations:
 - Local minimas
 - Non-differentiability
- How do we code this up?
 - Start with a random value of the independent variable(s)
 - Find the gradients
 - Update the variables using: $x_{new} = x_{old} + (- grad\{f\})$

Lessons from the Code

- About convergence threshold
- While updating the input values, we also need to keep a parameter called learning rate - to control our “step size” (i.e. the magnitude of change in value of the variable)
- The value of learning rate is a **hyperparameter** which requires some tuning
- **Note:** *Parameter vs Hyperparameter: Parameters are the configuration model, which are internal to the model. Hyperparameters are the explicitly specified parameters that control the training process*
- Practical is different from theory!
- Any other practical problems ?

Typical ML Pipeline



- Slightly incomplete (but we'll come to that)
- The ML Model can be seen as learning a function to map the input to the output (don't confuse this with the loss function).
- You can see the real world too as a mapping b/w input & output! (and our model is trying to learn that)
- You have an input $x = [x_1, x_2, \dots, x_n]$ and your function has some definition: Eg. $w_1x_1 + w_2x_2 + w_3x_3 \dots w_nx_n$ (Note: Usually the function would be much complex - with higher degrees and non-linearities)

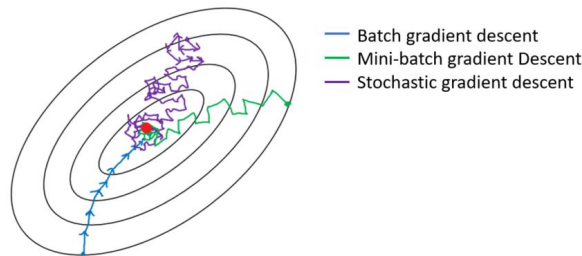
Typical ML Pipeline

- You have an input $\mathbf{x} = [x_1, x_2, \dots, x_m]$ and your function has some definition: Eg. $w_1x_1 + w_2x_2 + w_3x_3 \dots w_mx_m$ (Note: Usually the function would be much complex - with higher degrees and non-linearities)
- The above can also be represented compactly as a vector multiplication: $\mathbf{w}^T \mathbf{x}$
- This provides ease of representation
- This is for a single training example. For the whole data, this would become: $\mathbf{W} @ \mathbf{X}$
- Where the size of the \mathbf{W} matrix would be: (? x No. of samples). Which means the matrices (\mathbf{W} and \mathbf{X}) and the corresponding multiplications can be huge to hold in memory (**Another practical problem!**)

Mini-Batch and Stochastic Gradient Descent

- Since the matrix can be huge, processing all the data together is not possible
- **Mini - Batch Gradient Descent:**
 - Split the dataset into small subsets (batches) and compute gradients for them (and update the gradients right away).
 - And keep doing this till you have iterated over the whole dataset
 - One iteration over the whole dataset would be 1 epoch
 - Number of Iterations: $(\text{Num samples} / \text{batch size})$; Num epochs = Hyper-parameter
 - **Note:** English meaning of “epoch”: A particular period of time in history or someone’s life
- **Stochastic Gradient Descent:** Mini - Batch with just 1 sample per iteration

Mini-Batch and SGD



- Since mini-batch and SGD do not provide us an exact idea the gradient of the data (only some representation), the gradient direction found using them is not fully accurate. This leads to some “noise” in our direction of motion, and also a slower convergence (Imagine a drunk man walking)
- However, this noise sometimes comes to our benefit! As this noise can help us escape local extremas
- But slow convergence is an issue. The higher the batch size, faster the convergence. But also more chances of getting stuck at a local extrema
- Thus mostly, we go for a bargain between the two (based on our machine’s capabilities)
- **Gradient Accumulation:** “Simulate” larger Batch Size (sum gradients over some iterations)

Pseudo Code

Regular:

```
while( iter_count < max_iters or convergence_not_reached)
```

$$x_{new} = x_{old} - lr * (grad\{f\})$$

Mini-Batch:

```
while( epoch_count < max_epochs or convergence_not_reached)
```

```
  for batch_num in (num_samples / batch_size)
```

$$x_{new} = x_{old} - lr * (grad\{f\})$$

Stochastic: Put batch-size = 1

Pseudo Code

Gradient Accumulation:

```
while( epoch_count < max_epochs or convergence_not_reached)
```

```
    grad = 0
```

```
    for batch_num in (num_samples / batch_size)
```

```
        grad+ = grad_for_this_iter
```

```
     $x_{new} = x_{old} - lr * (grad_{f})$ 
```

Gradient Accumulation is used when we want to use a particular batch size (for faster convergence) but can't use it due to system constraints.

Note: Avoid taking too large batches to minimize probability of getting stuck at local extremas

Pseudo Code

Gradient Accumulation:

```
while( epoch_count < max_epochs or convergence_not_reached)
```

```
    grad = 0
```

```
    for batch_num in (num_samples / batch_size)
```

```
        grad+ = grad_for_this_iter
```

```
     $x_{new} = x_{old} - lr * (grad_{f})$ 
```

Gradient Accumulation is used when we want to use a particular batch size (for faster convergence) but can't use it due to system constraints.

Note: Avoid taking too large batches to minimize probability of getting stuck at local extremas

Back Propagation

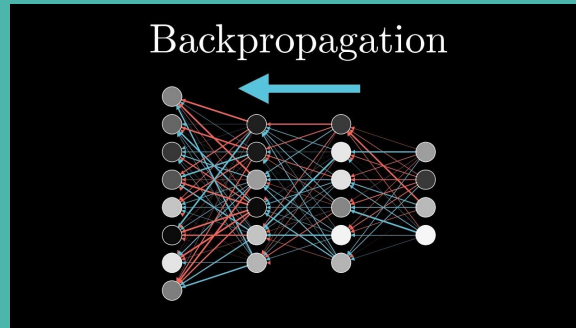
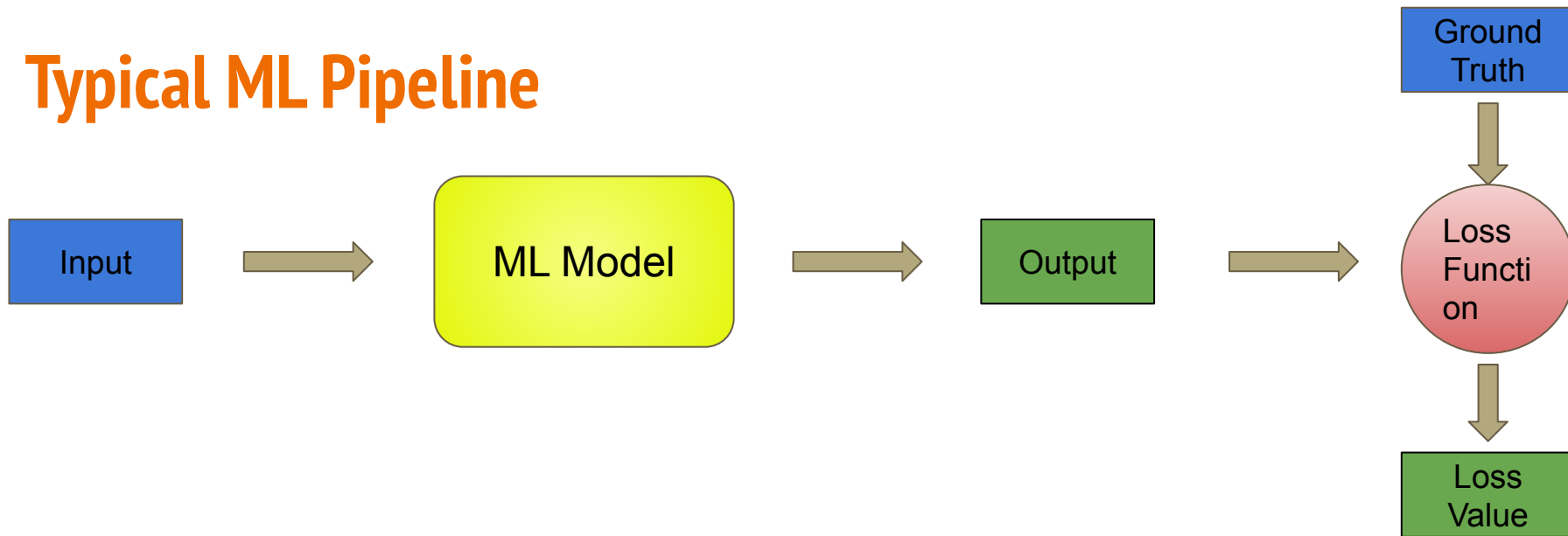


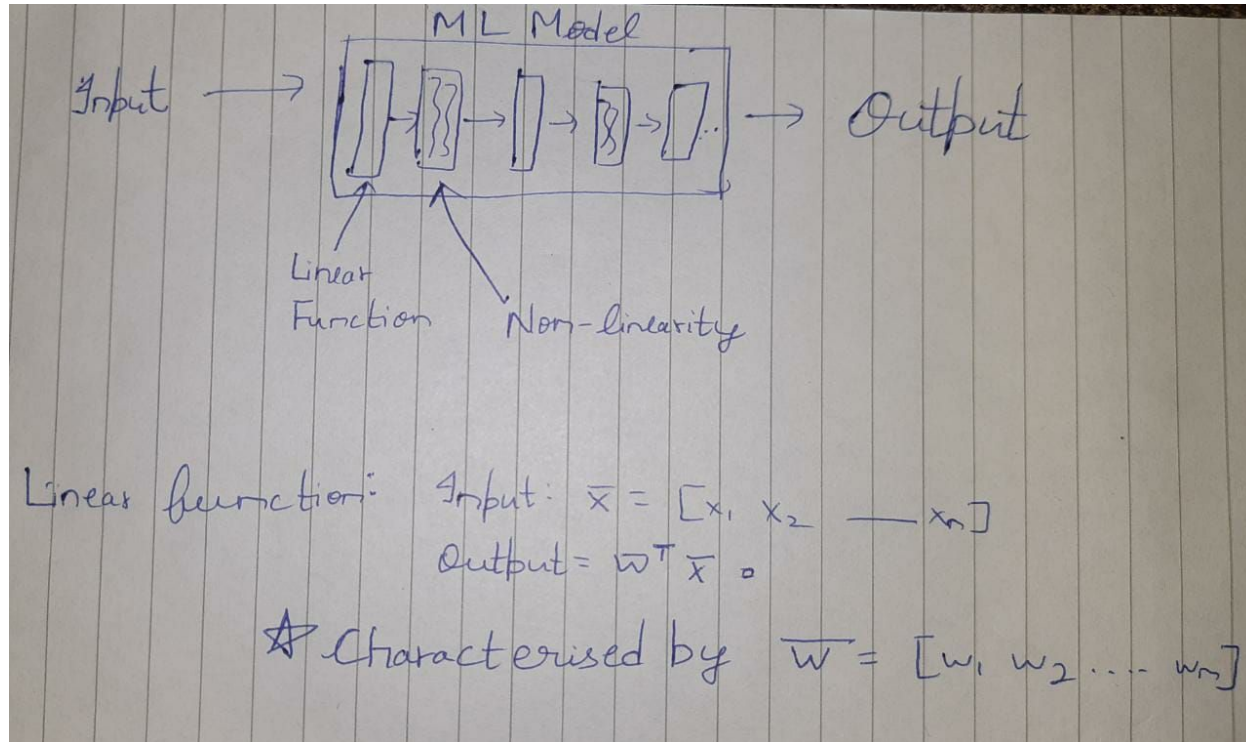
Image Credits: <https://www.youtube.com/watch?v=llg3gGewQ5U>

Typical ML Pipeline

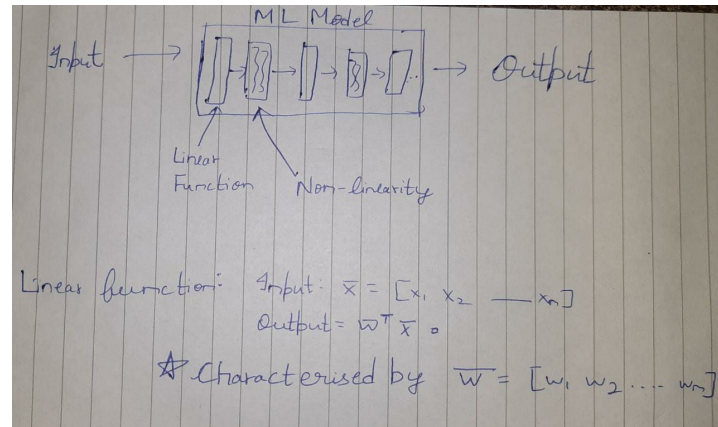


- (Still slightly incomplete)
- To minimize the loss value, the input(s) of the loss function must change
- Which means the output of our ML Model must change (because GT can't be changed!)
- Which means the definition of our ML model must change (because its input can't be!)
- So, to change our model, let's look inside it

Inside ML Model

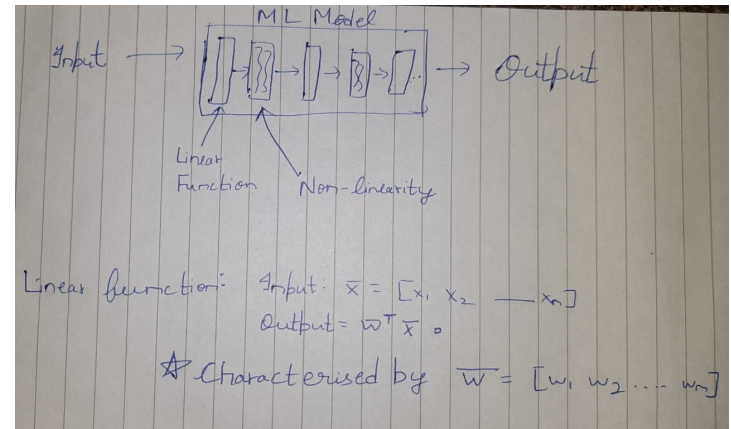


Inside ML Model



- The full model is a composition of multiple linear functions, along with non-linear layers (Eg. ReLu) in between
- The linear functions can be represented as matrices
 - See this: <https://www.youtube.com/watch?v=kYB8lZa5AuE> (full series [here](#))
- The nonlinear layers allow the model to learn non-linear functions
- Without nonlinear layers: (1) We can't work on nonlinear data; (2) Composition of multiple functions is as good as a single function (**Think:** Multiplication of a series of matrices can be represented as a single matrix)
- This kind of a setup allows us to learn non-linear functions, without having to explicitly define the “degree” of non-linearity and also use an “easier” matrix based notation

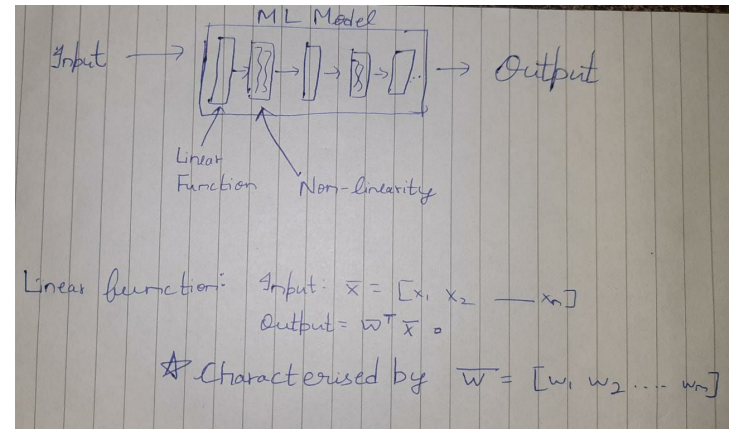
Inside ML Model



- The functions are characterized by their coefficients (commonly referred to as “weights” of the model in ML)
- If we modify w , we can modify the output of the individual linear function(s)
- Which can in turn modify the value of the output of the ML model
- But, how do we decide what values to increase and what to decrease?

Any guesses?

Inside ML Model



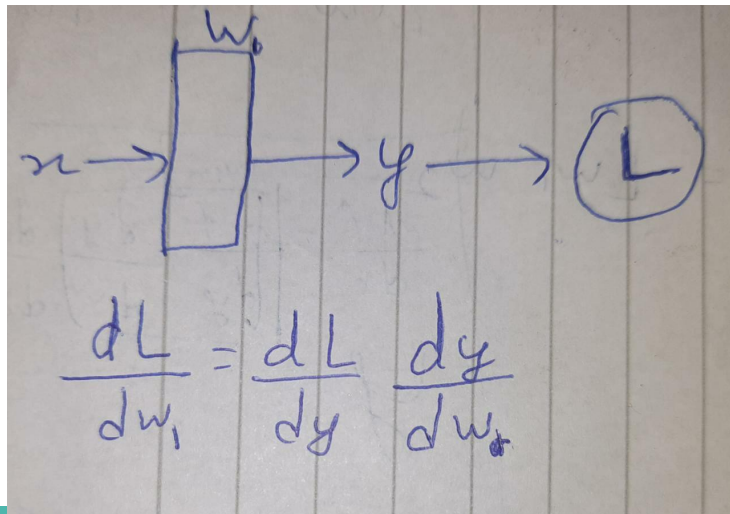
- The functions are characterized by their coefficients (commonly referred to as “weights” of the model in ML)
 - If we modify w , we can modify the output of the individual linear function(s)
 - Which can in turn modify the value of the output of the ML model
 - But, how do we decide what values to increase and what to decrease?
- Any guesses?**
- We will use gradient descent!

Using Gradient Descent to Train ML Models

- To find in which direction we should change w , we'll need to compute the derivative of our loss function (let's call it L) wrt w
- But L is defined wrt our output variable (y)
- **So what do we do ?**

Using Gradient Descent to Train ML Models

- To find in which direction we should change w (or in other words, to find how “change in w effects the loss function”), we’ll need to we’ll need to compute the derivative of our loss function (let’s call it L) wrt w
- But L is is defined wrt our output variable (y)
- **So what do we do ?**
- Chain Rule to the rescue!

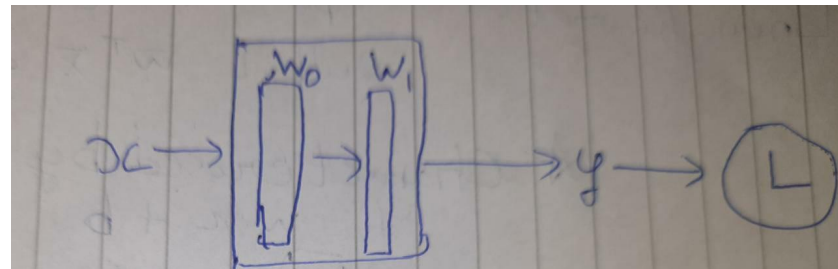


The diagram shows a flow from input x to a box representing a model, which outputs y . The box is labeled with w_0 above it. An arrow points from y to a circle containing the letter L , representing the loss function. Below the diagram, the chain rule is written as:

$$\frac{dL}{dw_i} = \frac{dL}{dy} \frac{dy}{dw_i}$$

For multiple layers

- Notice that the derivative calculated at w1 layer can be reused at w2

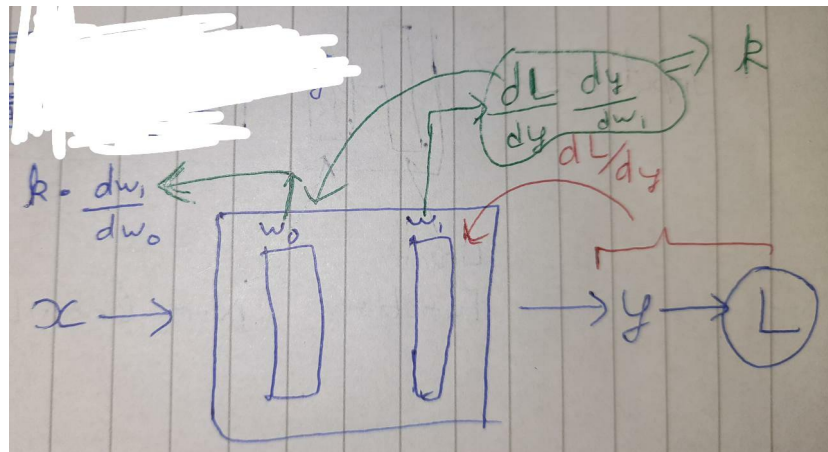
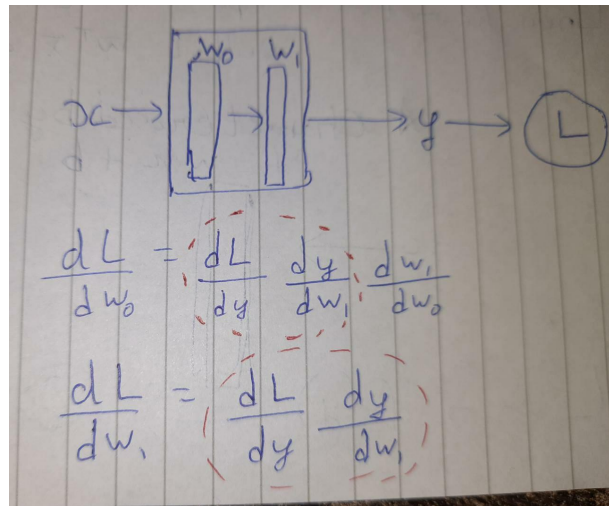


The diagram shows a neural network with two layers, w_0 and w_1 , leading to output y and loss L . The loss L is circled. The derivative $\frac{dL}{dw_0}$ is calculated using the chain rule, with the intermediate derivative $\frac{dy}{dw_1}$ being reused for the calculation of $\frac{dL}{dw_1}$.

$$\frac{dL}{dw_0} = \frac{dL}{dy} \frac{dy}{dw_1} \frac{dw_1}{dw_0}$$
$$\frac{dL}{dw_1} = \frac{dL}{dy} \frac{dy}{dw_1}$$

For multiple layers

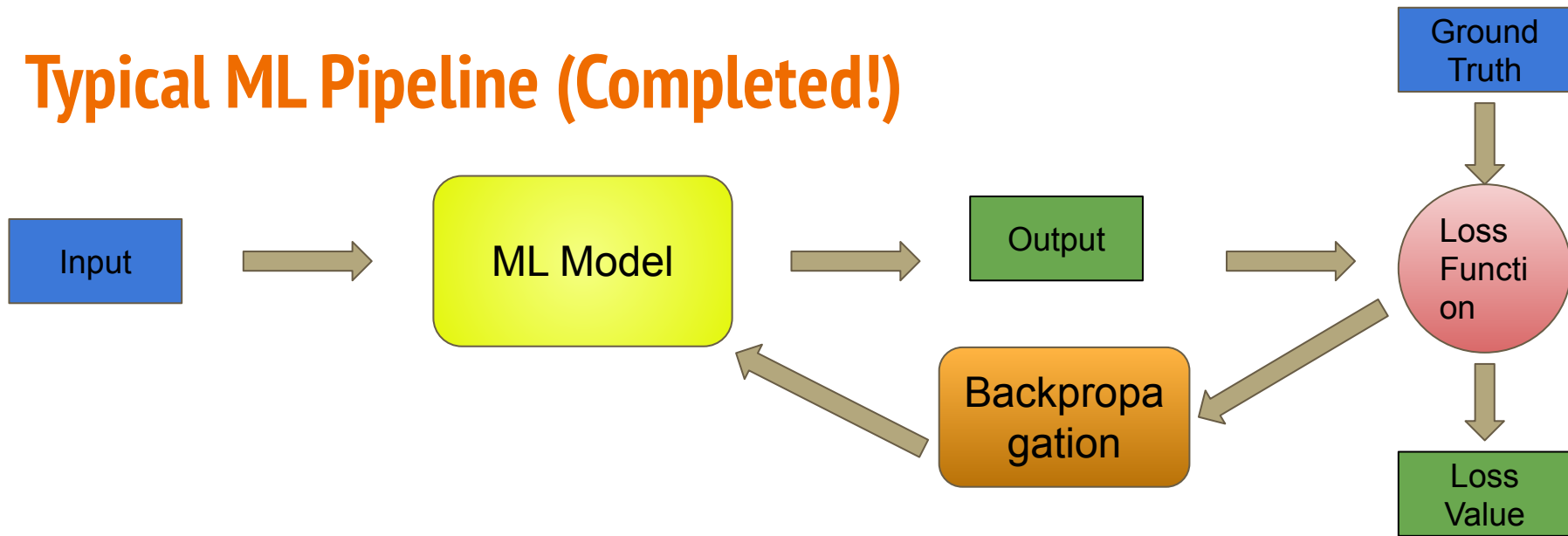
- Notice that the derivative calculated at w_1 layer can be reused at w_0
- So we can send the derivatives “back” from w_1 layer to w_0 . This makes computation easier
- This is called as **backpropagation**



Backpropagation

- What we saw is a very simplified version. In practice, you'll be dealing with gradients and partial derivatives and a lot of layers.
- For more details, check out:
 - Intuition: <https://www.youtube.com/watch?v=llg3gGewQ5U>
 - Maths: <https://www.youtube.com/watch?v=tleHLnjs5U8>
- Notice that we won't be able to reuse gradients had we gone from left to right (i.e. **forward pass / propagation**)
- **Note:** The “flow” of inputs inside the model and to the output is known as forward pass.
- **Feedforward Neural Networks:** Artificial neural network wherein connections between the nodes do not form a cycle.

Typical ML Pipeline (Completed!)



- Finally complete!

Coding Time!

- Computing so many gradients for complex functions can be pretty tricky
- PyTorch to the rescue!
- For most common loss functions, PyTorch has their function definitions and their derivatives pre-defined. And handles gradient computation using a function called **autograd** (It's way more powerful than what it may look like!). For more details:
 - https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
 - <https://towardsdatascience.com/automatic-differentiation-explained-b4ba8e60c2ad>
- Detailed maths behind gradient descent
 - <https://towardsdatascience.com/step-by-step-the-math-behind-neural-networks-490dc1f3cfd9>
 - <https://towardsdatascience.com/step-by-step-the-math-behind-neural-networks-ac15e178bbd>
 - <https://towardsdatascience.com/step-by-step-the-math-behind-neural-networks-d002440227fb>
 - <https://towardsdatascience.com/calculating-gradient-descent-manually-6d9bee09aa0b>

Thank You