Module 2

Training Neural Networks

Training process of neural networks involves following steps such as

1. Initialize parameters
2. Choose an *optimization algorithm*
3. Repetition of the following steps:
    1. Forward propagate an input
    2. Compute the cost function
    3. Compute the gradients of the cost with respect to parameters using backpropagation
    4. Update each parameter using the gradients, according to the optimization algorithm

The importance of effective initialization of parameters or Weight Initialization

- **Weight initialization** is an important consideration in the design of a neural network model.
- The nodes in neural networks have parameters which are referred to as weights, used to calculate a weighted sum of the inputs.
- Neural network models are fit using an optimization algorithm called s gradient descent that incrementally changes the network weights to minimize a loss function, hopefully resulting in a set of weights for the nodes that is capable of making useful predictions.

Because this initialization step is critical to the model's ultimate performance, and it requires the right method. **Any constant initialization scheme will perform very poorly. ie ,.** *Initializing all the weights with a constant value leads the neurons to learn the same features during training.*

Eg: Consider a *neural network* with two hidden units, and assume we initialize all the biases to 0 and the weights with some constant α.

If we forward propagate an input $(x1,x2)$ in this network, the output of both hidden units will be same. Thus, both hidden units will have identical influence on the cost, which will lead to identical gradients.

Thus, both neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things.

Problems associated with initializing weights with values too small or too large

- *Despite breaking the symmetry, initializing the weights with values too small leads to slow learning and initializing the weights with too large leads to divergence.*

The problem of exploding or vanishing gradients

Vanishing Gradients

As the back propagation algorithm advances backward (ie, in phase II- the backpropagation of errors) from the output layer towards the input layer, the gradient get smaller and smaller and approaches to zero which eventually leaves the weights of initial or lower layers nearly unchanged.

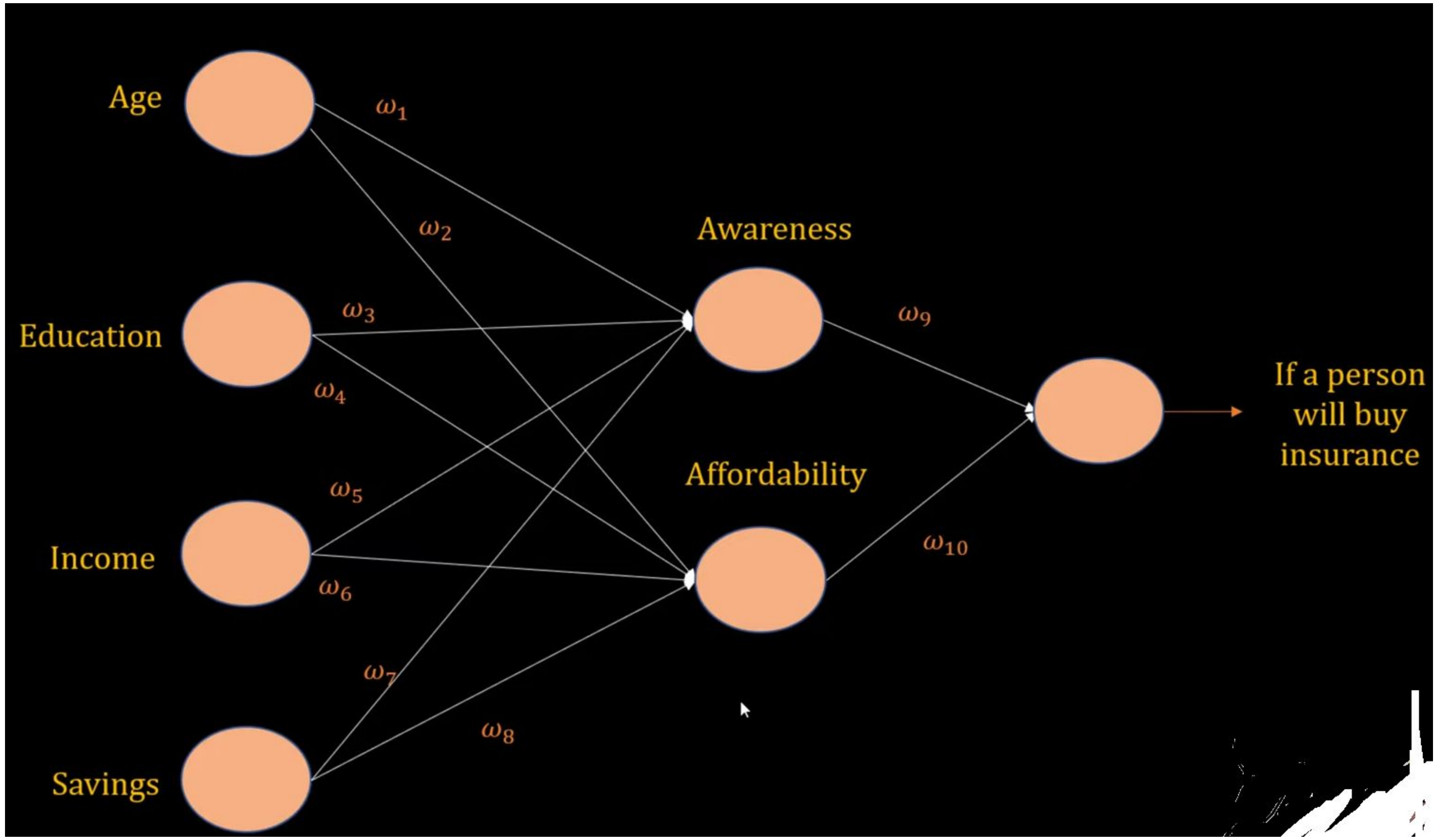
As a result, the gradient descent never converges to the optimum. This is known as vanishing gradient problem.

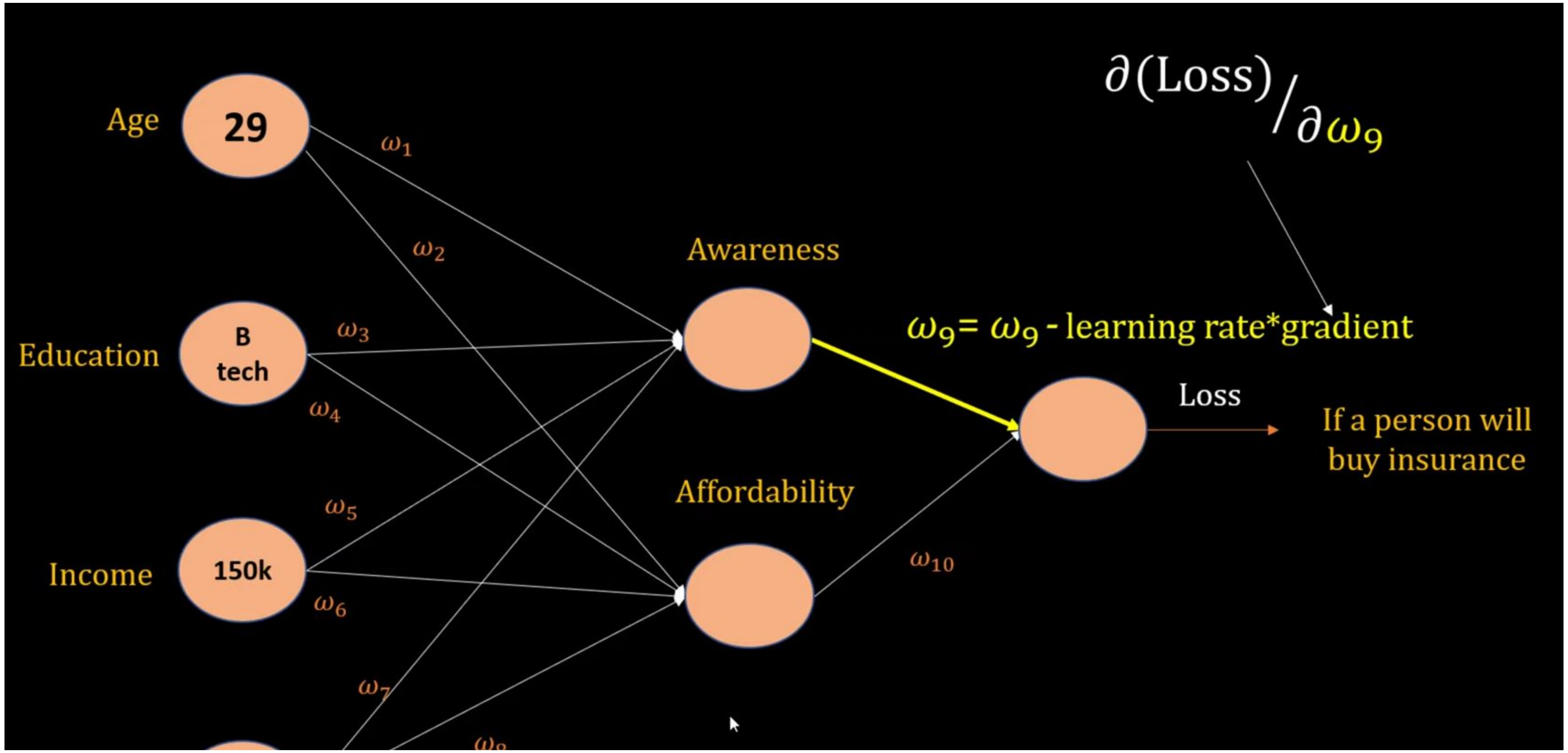
Exploding gradients

In some cases, as the back propagation algorithm advances backward, the gradient keep on getting larger. This in turn causes very large weight updations and causes gradient descent to diverge.This is known as exploding gradient problem.

Note:

- ***A too-large initialization leads to exploding gradients.***
- ***A too-small initialization leads to vanishing gradients***


Age
Education
Income
Savings
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6
ω_7
ω_8
Awareness
Affordability
ω_9
ω_{10}
If a person will buy insurance


$\partial(\text{Loss}) / \partial \omega_9$
Age
29
Education
B tech
Income
150k
Awareness
Affordability
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6
ω_7
ω_{10}
$\omega_9 = \omega_9$ - learning rate*gradient
Loss
If a person will buy insurance

$$\partial(\text{Loss})/\partial\omega_1 = \partial(\text{Loss})/\partial Awareness * \partial(Awareness)/\partial\omega_1$$

$$gradient = d1 * d2$$

$$gradient = 0.03 * 0.05$$

$$gradient = 0.0015$$

$\omega_{1\text{ new}} = \omega_{1\text{ old}}$ - learning rate* gradient

$$gradient = 0.0015$$

As number of hidden layers grow, gradient becomes very small and weights will hardly change . This will hamper the learning process.

W
d
T
r
i
N
c
t
r
n

$$\partial(\text{Loss})/\partial\omega_1 = \partial(\text{Loss})/\partial Awareness * \partial(Awareness)/\partial\omega_1$$

$$gradient = d1 * d2$$

$$gradient = 100 * 500$$

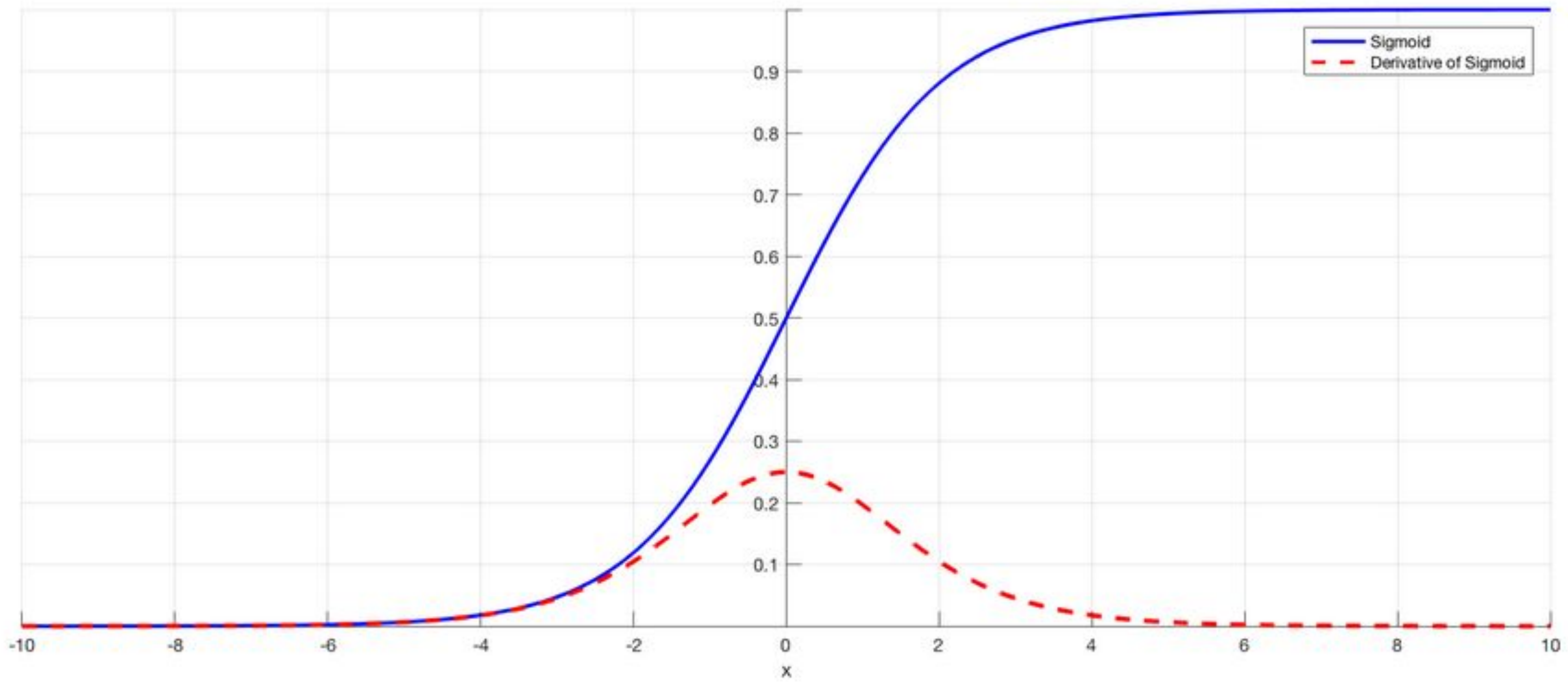
$$gradient = 50000$$

Why do the gradients vanish or explode

- Certain activation functions, like the sigmoid function, reduces a large input space into a small input space between 0 and 1. As more layers using certain activation functions are added to neural networks, the derivative becomes small and the gradients of the loss function approaches zero, making the network hard to train.

- So in this situation, the backpropagation phase of the back propagation algorithm, virtually has no gradient to propagate backward in the network and whatever little residual gradient exists keeps on diluting as the algorithm progresses from output layer to input layer. So this leaves nothing for the lowest layers.

- In some cases the initial weights assigned to the network generate large loss. The gradients can accumulate during an update and result in very large gradients which eventually results in large update to the network weights and leads to an unstable network.


Sigmoid
Derivative of Sigmoid
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
-10
-8
-6
-4
-2
0
2
4
6
8
10
x

Solution : Exploding and vanishing Gradients

- There are many approaches to address exploding and vanishing gradients:

1. **Reducing the amount of Layers**

- This solution could be used in both scenarios (exploding and vanishing gradient).
- However, by reducing the number of layers in our network, we give up some of our model’s complexity, since having more layers makes the networks more capable of representing complex mappings.

2. Gradient Clipping (Exploding Gradients)

- Checking for and limiting the size of the gradients while our model trains.
- Define an interval within which we expected the gradients to fall.
  - If the gradients exceed the permissible maximum, automatically set them to the maximum upper bound of our interval.
  - Similarly, if they fall below the permissible minimum, automatically set them to the lower bound.

3. Weight Initialization

- In a neural network, we initialize the weights randomly. Certain techniques such as He initialization and Xavier initialization ensure that the weights are close to 1.

How to find appropriate initialization values

1. **Xavier initialization**.

- It is the current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function.
- There are two versions of this weight initialization method, which are referred to as “Xavier initialization” and “normalized Xavier initialization.”
- This technique tries to make the variance of the outputs of a layer to be equal to the variance of the inputs.

• The xavier initialization method calculates weight as a random number with a uniform probability distribution (U) between the range

  -(1/sqrt(n)) and 1/sqrt(n)

  where *n* is the number of inputs to the node.

• weight = U [-(1/sqrt(n)), 1/sqrt(n)]

Normalized Xavier Weight Initialization

- The normalized xavier initialization method calculates weight as a random number with a uniform probability distribution (U) between the range -(sqrt(6)/sqrt(n + m)) and sqrt(6)/sqrt(n + m)
- where *n* is the number of inputs to the node (e.g. number of nodes in the previous layer) and *m* is the number of outputs from the layer (e.g. number of nodes in the current layer).
- weight = U [-(sqrt(6)/sqrt(n + m)), sqrt(6)/sqrt(n + m)]

- The X*avier* weight initialization was found to have problems when used to initialize networks that use the rectified linear (ReLU) activation function.
- As such, a modified version of the approach was developed specifically for nodes and layers that use ReLU activation.
- The current standard approach for initialization of the weights of neural network layers and nodes that use the rectified linear (ReLU) activation function is called "H*e*" initialization.
- It is named for Kaiming He, currently a research scientist at Facebook.

2. He Weight Initialization

- The He initialization method calculates weight as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of sqrt(2/n), where *n* is the number of inputs to the node.
- weight = G (0.0, sqrt(2/n))

It can be implemented directly in Python.

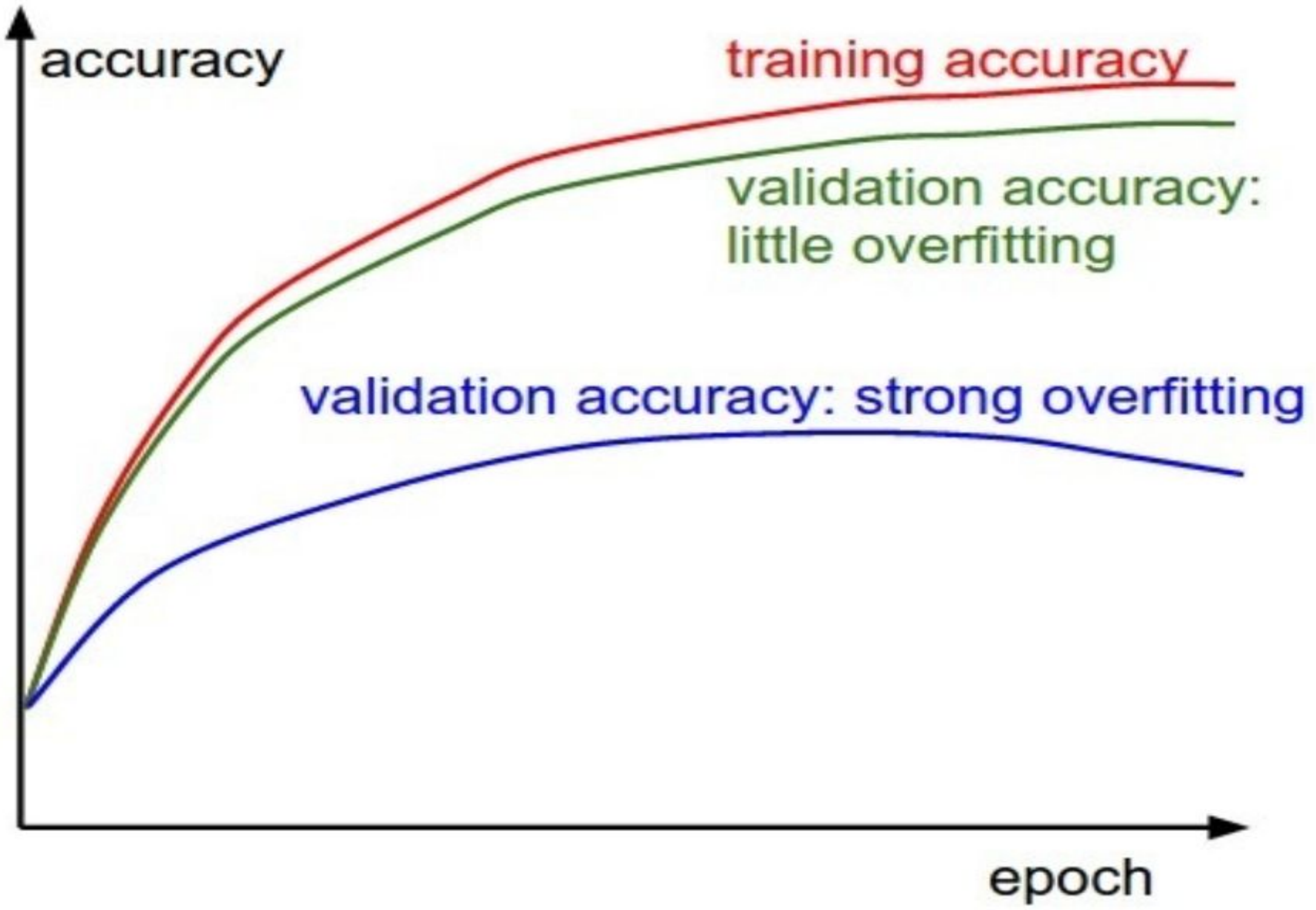
- These modern weight initialization techniques are divided based on the type of activation function used in the nodes that are being initialized, such as “*Sigmoid and Tanh*” and “*ReLU*.”

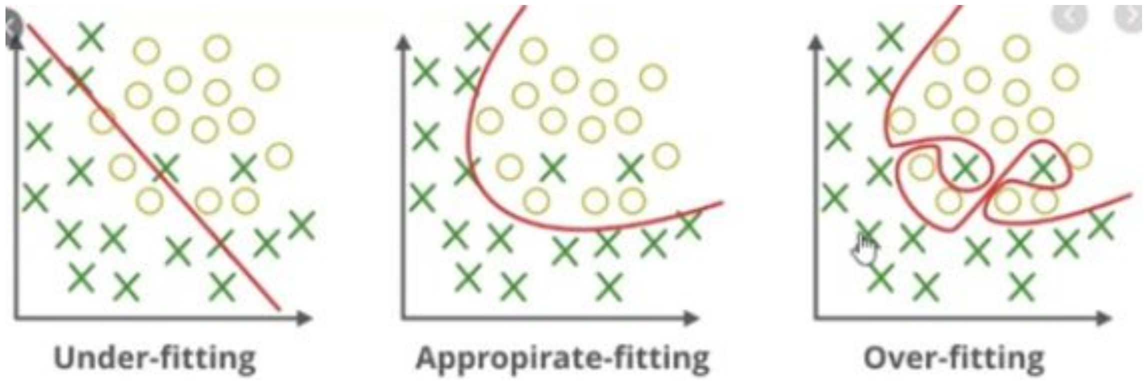
OVERFITTING

- Deep neural networks have lots of layers between the inputs layer and output layer
- Number of hidden layers increases the complexity of the network.
- When the training dataset has very few examples, there is likely to be **overfitting, which is when the network is able to accurately predict the samples of the training data but has poor performance and cannot generalize well on the validation and the test data**.

- When the model is ***overfitting, the accuracy of the training data would be very high and that of the validation very low***, because the model has learned all the patterns in the training data but cannot generalize and make good predictions on data it has not seen before.
- In other words, Overfitting occurs when the model is good and able to classify or predict on data that was included in the training set, but is not as good at classifying data that it wasn't trained on.
- **Overfitting can be reduced if the accuracies of training and validation is close and high.**
- **If there is no overfitting the model can make better predictions on data it has not seen before.**

OVERFITTING




Under-fitting
Appropirate-fitting
Over-fitting

How to reduce **OVERFITTING?**

There are several things we can do to reduce overfitting whiles training our neural network models

1) **Reduce the complexity of the model (that is reduce the number of hidden layers).**
2) **Adding more data to the training set.**
3) **Augment the data.**
4) **Apply regularization, that is applying a penalty factor to the loss function (L1 or L2 regularization).**
5) **Dropout.**

How to reduce Overfitting

1. Reduce the complexity of the model

- Reduce complexity of the model by making simple changes, like

- removing some layers from the model, or

- reducing the number of neurons in the layers. This may help our model generalize better to data it hasn't seen before.

2. **Adding more data to the training set**

3. Data augmentation

- Another technique to reduce overfitting is to use data augmentation. This is the process of creating additional augmented data by reasonably modifying the data in the training set.
- For image data, for example, we can do these modifications by:
- Cropping
- Rotating
- Flipping
- Zooming

- The data augmentation allows us to add more data to the training set that is similar to the data that we already have, but is just reasonably modified to some degree so that it's not the exact same.

4. Apply Regularization

- ***Regularization*** is a technique that helps reduce overfitting or reduce variance in a Neural network by penalizing for complexity. The idea is that certain complexities in our model may make our model unlikely to generalize well, even though the model fits the training data.

- To implement regularization is to simply add a term to our loss function that penalizes for large weights.

- $Y= x_1w_1 +x_2w_2+ \dots +x_nw_n$
- $Y = 15+x_1*20 +x_2*1.5+ \dots +x_n*30$

- $Y = x_1*2 +x_2*0.2+ \dots +x_n*5$

L1 regularization

- L1 regularization, also known as L1 norm or Lasso (in regression problems), combats overfitting by shrinking the parameters towards 0. This makes some features obsolete.
- It’s a form of feature selection, because it shrinks the less important feature’s coefficient to zero.
- Mathematically, L1 regularization is expressed by extending loss function as below:

$$LossFunction = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y} - Y)^2 + \lambda \sum_{i=1}^{N} |w_i|$$

- L1 regularization, we are penalizing the absolute value of the weights.

L2 regularization

- L2 regularization, or the L2 norm, or Ridge (in regression problems), combats overfitting by forcing weights to be small, but not making them exactly 0.
- The regularization term that we add to the loss function when performing L2 regularization is the sum of squares of all of the feature weights:

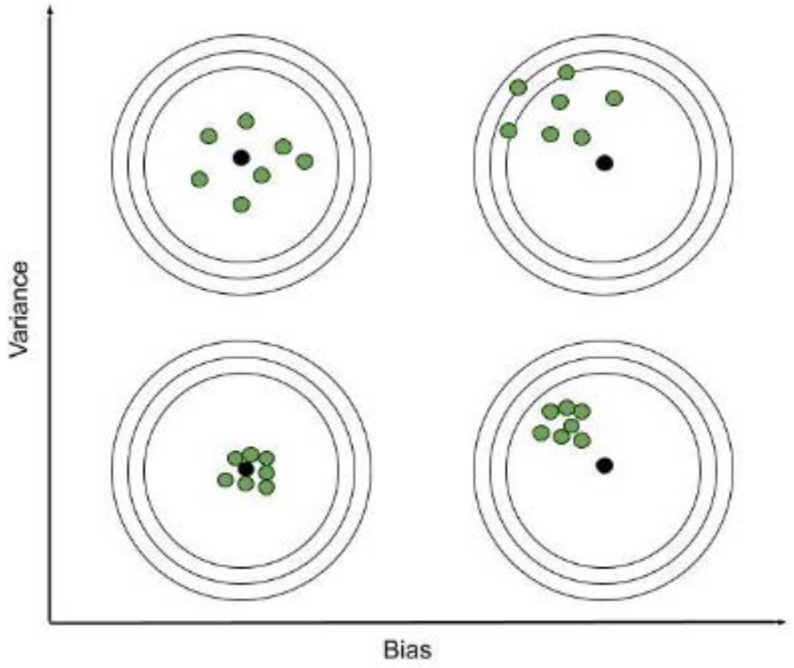
$$LossFunction = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y} - Y)^2 + \lambda \sum_{i=}^{N} ||w_i^2||$$

The term λ is called the regularization parameter, and this is another hyperparameter that we'll have to choose and then test and tune in order to choose the correct number for our specific model.

Bias and Variance

- Bias refers to the gap or difference the predicted value from the model and target value expected from the model

- Variance - variance is **the variability in the model prediction.**


Variance
Bias

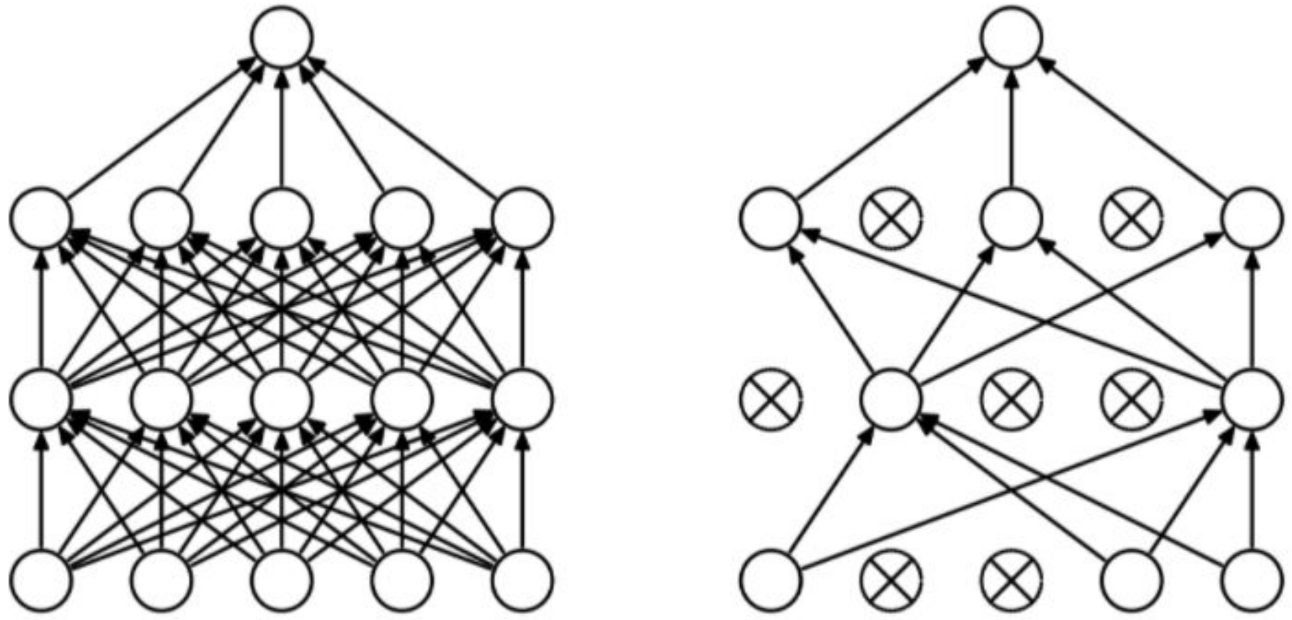
Dropout

- As we train the model iteratively, all the weights are learned together and some of the neurons adapt better and make better predictions than others.
- As the network is trained over several epochs, **the stronger neurons learn more**, while the **weaker ones are ignored**.
- A portion of the neurons are trained after several iterations and the **weaker ones avoid taking part in the training**.
- One method to handle this issue is dropout.

- Dropout is a technique that drops neurons from the neural network or ‘ignores’ them during training.

Or

- Different neurons are removed from the network on a temporary basis

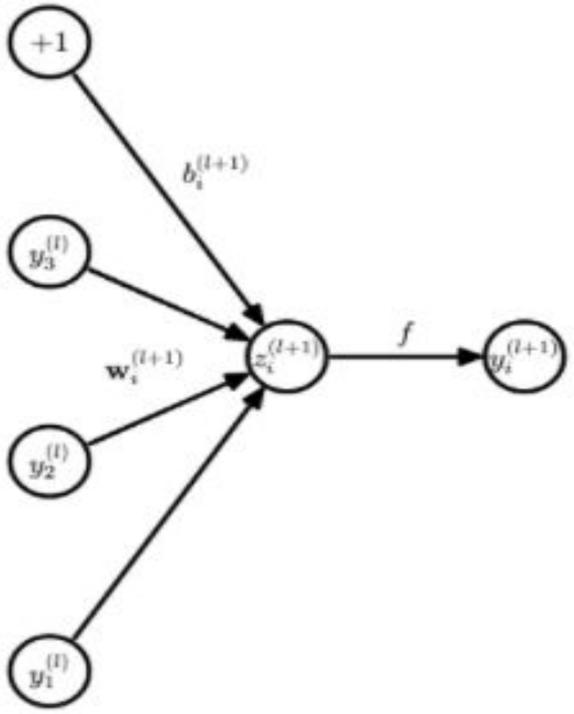


(a) Standard Neural Net (b) After applying dropout.

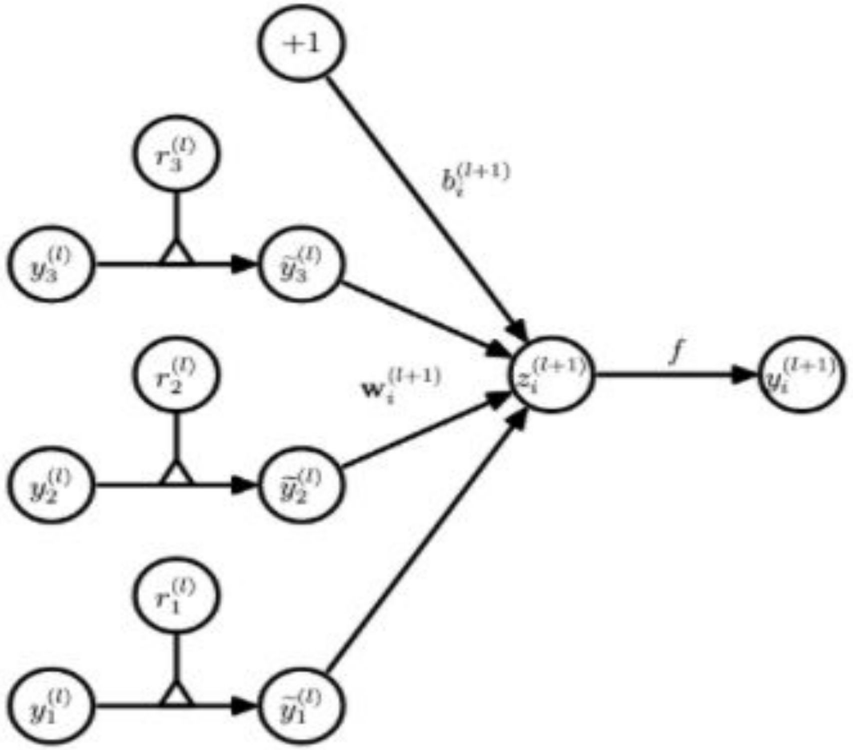
- During training, dropout modifies the idea of learning all the weights in the network to learning just a fraction of the weights in the network.
- During the standard training phase, all neurons are involved .In dropout, only a few select neurons are involved with the rest 'turned off' ie. **we switch off some neurons during training**.
- So after every iteration, different sets of neurons are activated, to prevent some neurons from dominating the process.
- This helps us reduce the menace of overfitting and allows for the rise of deeper and bigger network architectures that can make good predictions on data the network has not seen before.

Dropout.

- This can be done as below.



(a) Standard network



(b) Dropout network

- Training of dropout network is similar to the standard network.
- Here **a new neuron r, is introduced**, **which keeps the neuron active or turns it off, by assigning a 1(neuron participates in the training) or 0 (neuron does not participate or is turned off)**, then the training process continues.
- This way, **overfitting is reduced and our model can now make excellent and accurate predictions on real-world data(data not seen by the mode**

Why Overfitting is a problem?

- The model is unable to generalize well.
- It has learned the features of the training set extremely well, but if we give the model any data that slightly deviates from the exact data used during training, it's unable to generalize and accurately predict the output.

UnderFitting

- When the model neither learns from the training dataset nor generalizes well on the test dataset, it is termed as underfitting.

  i.e., A data model is unable to capture the relationship between the input and output variables accurately, generating a high error rate on both the training set and unseen data.
- It occurs when a model is too simple, which can be a result of a model needing more training time, more input features, or less regularization.

To reduce underfitting:

- Increase model complexity.
- Increase the number of features.
- Remove noise from the data.
- Increase the number of epochs or increase the duration of training to get better results.

How to avoid underfitting

1. Decrease regularization

- Methods such as L1 regularization, Lasso regularization, dropout, etc., help to reduce the noise and outliers within a model.
- If the data features become too uniform, the model is unable to identify the dominant trend, leading to underfitting.
- By decreasing the amount of regularization, more complexity and variation is introduced into the model, allowing for successful training of the model.

2. **Increase the duration of training**

- As mentioned earlier, stopping training too soon can also result in underfit model. Therefore, by extending the duration of training, it can be avoided.

3. **Feature selection**

- If there are not enough predictive features present, then more features or features with greater importance, should be introduced.

For example, in a neural network, you might add more hidden neurons This process will inject more complexity into the model, yielding better training results.

Batch Normalization.

- Normalization is a data pre-processing tool used to bring the numerical data to a common scale without distorting its shape.
- Generally, when we input the data to a machine or deep learning algorithm the values should be changed to a balanced scale. Normalization is done to ensure that model can generalize appropriately.

Normalization and standardization of Data

- Normalization helps to scale down features between 0 and 1.
- Generally, when we input the data to a machine or deep learning algorithm, we tend to change the values to a balanced scale.

  Eg: if we have a dataset with feature vector X(X1, X2, ….Xn) that have very different ranges of values (e.g., [0,1] as well as [1,1000], we should normal ize this data to speed up the learning of the network.

- Input data can be Nromalized using the equation

$$x_i = \frac{x_i - \mu}{\sigma}$$

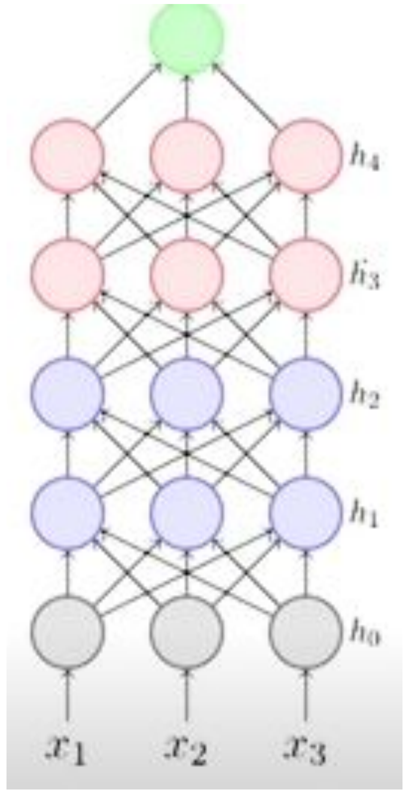
Where μ is the mean of the feature and σ the standard deviation of that feature.

- This way, we can make the training of a deep learning model both **more stable and faster**.

- A typical neural network is trained using a collection of input data called batch. Similarly, the normalizing process in batch normalization takes place in batches, not as a single input.
- Neural networks learn to make a prediction for a given problem using the backpropagation algorithm.
- During the training period, the weights and biases in the neural network are adjusted to make a better prediction next time. The weights and biases are adjusted, the values of the output of the network become closer to the actual values (labels).

The Problem of internal Covariance Shifting

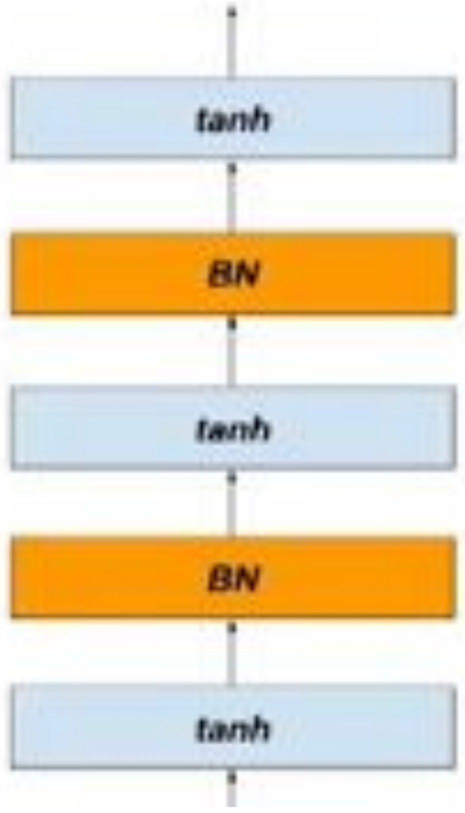
- The key issue that batch normalization tackles is internal covariate shift.
- At every epoch of training, **weights are updated and different data is being processed**, which means that the inputs to a neuron is slightly different every time. As these changes get passed on to the next neuron, it creates a situation where the input distribution of every neuron is different at every epoch.
- Due to the adjustment of weights and biases in the current layer, the subsequent layer is forced to learn on new input data. This behavior is called "internal covariance shift". As the network trains and the weights move away from their random initial values. This phenomenon is known as **covariate shift**. This phenomenon usually increases the training time of a neural network.
- This problem can be addressed with **batch normalization.**



Batch Normalization

- Batch normalization is a process to make neural networks faster and more stable through adding extra layers in a deep neural network.
- The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.
- Batch normalization is applied to individual layers

  Batch-Normalization in Practice
  - Normalise output from Activation function
  - Multiply (scale) normalised output by arbitrary parameter
  - Add (shift) an arbitrary parameter to the resulting product
- Since the scaling and shifting parameters are learnable parameters, the neural network can learn during training and decide for itself whether or to what extent to batch normalize.



In batch normalization, we normalize the neuron values over a mini-batch of **m** instances, rather than over all instances of the training dataset.

consider a mini-batch B of size m where m refers to the number of training samples in the mini-batch

D can be considered as the number of neurons in a given hidden layer.

- Formally, denoting by x∈B an input to batch normalization (BN) that is from a minibatch B, batch normalization transforms x according to the following expression:

$$BN(x) = \gamma \left(\frac{x - \mu_B}{\sigma_B} \right) + \beta$$

μ_B is the sample mean and σ_B is the sample standard deviation of the minibatch B.

γ and β are parameters that need to be learned jointly with the other model parameters.

$$\mu_{B=\frac{1}{m}\sum_{x \in B} x} \qquad \sigma_B = \sqrt{\frac{1}{m}\sum_{x \in B}(x - \mu_B)^2 + \epsilon}$$

Note that we add a small constant ϵ>0 to the variance estimate to ensure that we never attempt division by zero

Training and Validation Curves

- Learning curves plot the training and validation loss of a sample of training examples by incrementally adding new training examples. Learning curves help us in identifying whether adding additional training examples would improve the validation score (score on unseen data). If a model is overfit, then adding additional training examples might improve the model performance on unseen data. Similarly, if a model is underfit, then adding training examples doesn’t help.

- **Overfitting(Variance)**
- A model is said to be overfit if it is over trained on the data such that, it even learns the noise from it. An overfit model learns each and every example so perfectly that it misclassifies an unseen example or new example. For a model that's overfit, we have a perfect/close to perfect training set score while a poor test(validation)score.
- **Reasons behind overfitting:**
- Using a complex model for a simple problem which picks up the noise from the data.
- Small datasets, as the training set may not be a right representation of the universe.

- **Underfitting (Bias):**

A model is said to be underfit if it is unable to learn the patterns in the data properly. An underfit model doesn’t fully learn each and every example in the dataset. In such cases, we see a low score on both the training set and test/validation set.

- **Reasons behind underfitting:**
- Using a simple model for a complex problem which doesn’t learn all the patterns in the data.
- The underlying data has no inherent pattern. Example, trying to predict a student’s marks with his father’s weight.

- **Train Learning Curve**: A learning curve is a plot that shows time or experience on the x-axis and learning or improvement on the y-axis.
- It is a learning curve calculated from the training dataset that gives an idea of how well the model is learning.

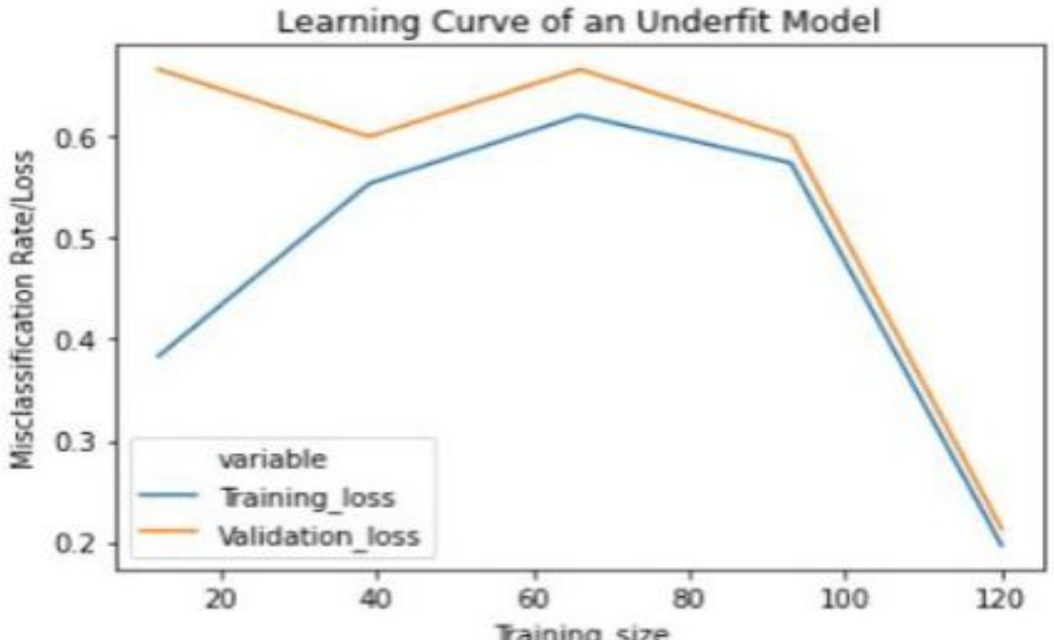
- **Validation Learning Curve**:It is a learning curve calculated from a hold-out validation dataset that gives an idea of how well the model is generalizing.

Diagnosing Model Behavior using Training and Validation Curves

- The shape and growth of a learning curve can be used to diagnose the behavior of a machine learning model and in turn suggest the type of configuration changes that may be made to improve learning and/or performance
- There are three common caes that are observed in learning curves; they are:
  - Underfit.
  - Overfit.
  - Good Fit.

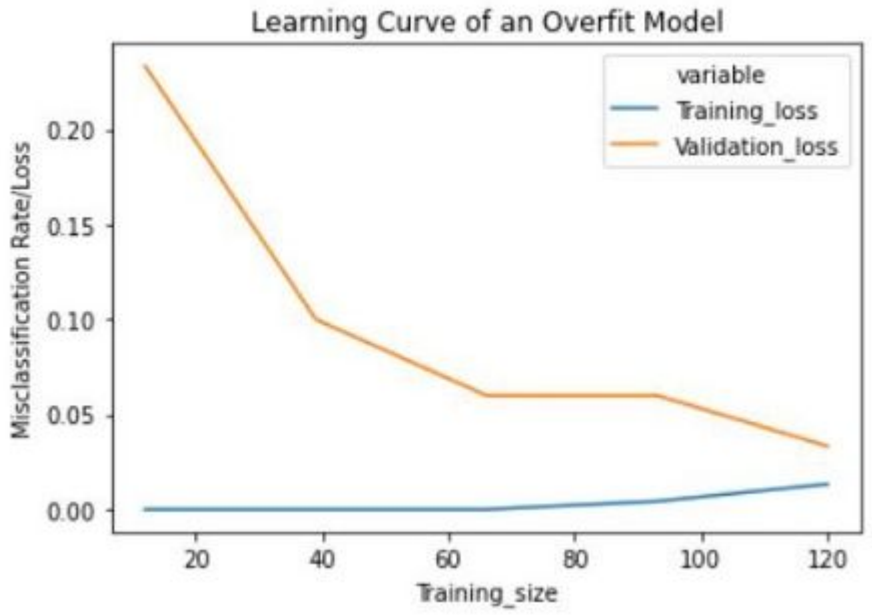
Underfit Learning Curves

- The standard deviation of cross validation accuracies is low for Underfit model compared to overfit and good fit model. However, underfitting can be detected from the learning curve.
- **Typical features of the learning curve of an underfit model**
- Increasing training loss upon adding training examples.
- Training loss and validation loss are close to each other at the end.



Overfit Learning Curves

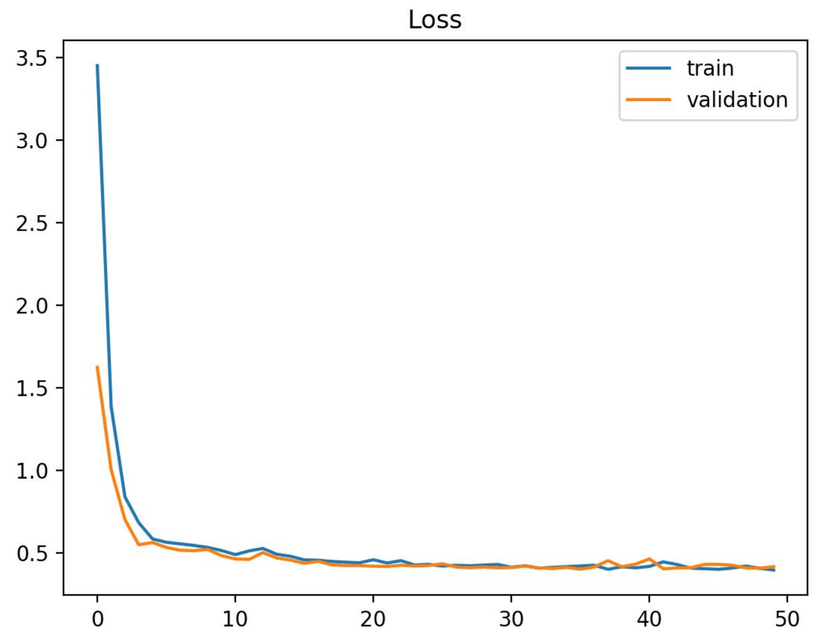
- The standard deviation of validation accuracies is high for overfitting model when compared to under fit and good fit model. Training accuracy is higher than validation accuracy, typical to an overfit model and overfitting can be detected from the learning curve.



- **Interpreting the training loss**
- Learning curve of an overfit model has a very low training loss at the beginning which gradually increases very slightly upon adding training examples and doesn’t flatten.
- **Interpreting the validation loss**
- Learning curve of an overfit model has a high validation loss at the beginning which gradually decreases upon adding training examples and doesn’t flatten, indicating addition of more training examples can improve the model performance on unseen data.
- We can also see that the training and validation losses are far away from each other, which may come close to each other upon adding additional training data.

Goodfit Learning Curves

- A plot of learning curves shows a good fit if:
- The plot of training loss decreases to a point of stability.
- The plot of validation loss decreases to a point of stability and has a small gap with the training loss.



- **Interpreting the training loss**
- Learning curve of a good fit model has a moderately high training loss at the beginning which gradually decreases upon adding training examples and flattens gradually, indicating addition of more training examples doesn’t improve the model performance on training data.
- **Interpreting the validation loss**
- Learning curve of a good fit model has a high validation loss at the beginning which gradually decreases upon adding training examples and flattens gradually, indicating addition of more training examples doesn’t improve the model performance on unseen data.
- Upon adding a reasonable number of training examples, both the training and validation loss moves close to each other.

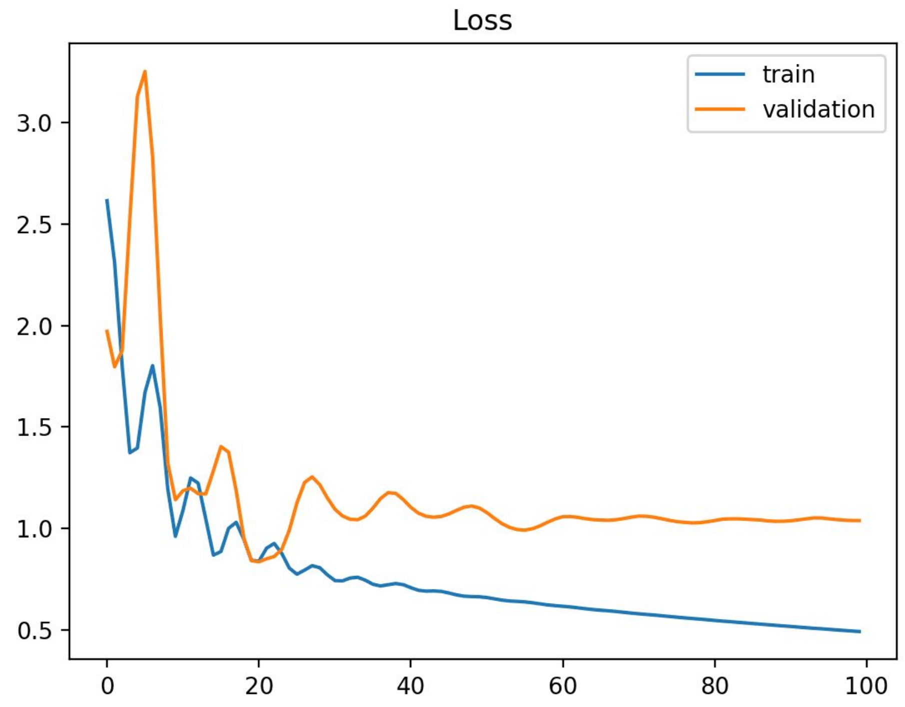
Diagnosing Unrepresentative Datasets

Learning curves can also be used to diagnose properties of a dataset and whether it is relatively representative.

- There are two common cases that could be observed; they are:
- Training dataset is relatively unrepresentative.
- Validation dataset is relatively unrepresentative.

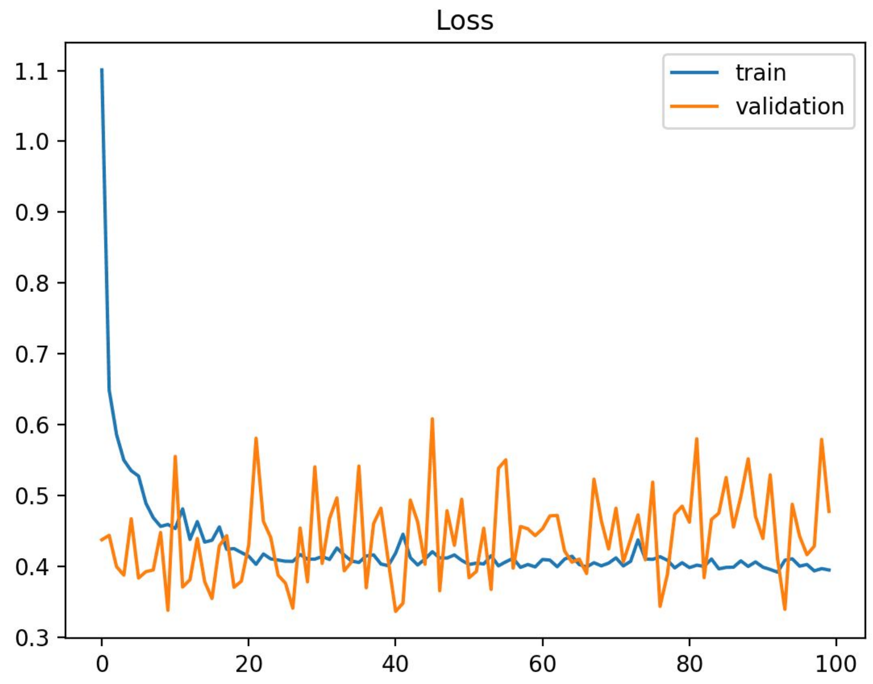
Diagnosing Unrepresentative Train Datasets

- An unrepresentative training dataset means that the training dataset does not provide sufficient information to learn the problem, relative to the validation dataset used to evaluate it.
- This may occur if the training dataset has too few examples as compared to the validation dataset.



Unrepresentative Validation Dataset

- An unrepresentative validation dataset means that the validation dataset does not provide sufficient information to evaluate the ability of the model to generalize.
- This may occur if the validation dataset has too few examples as compared to the training dataset.


Loss
train
validation