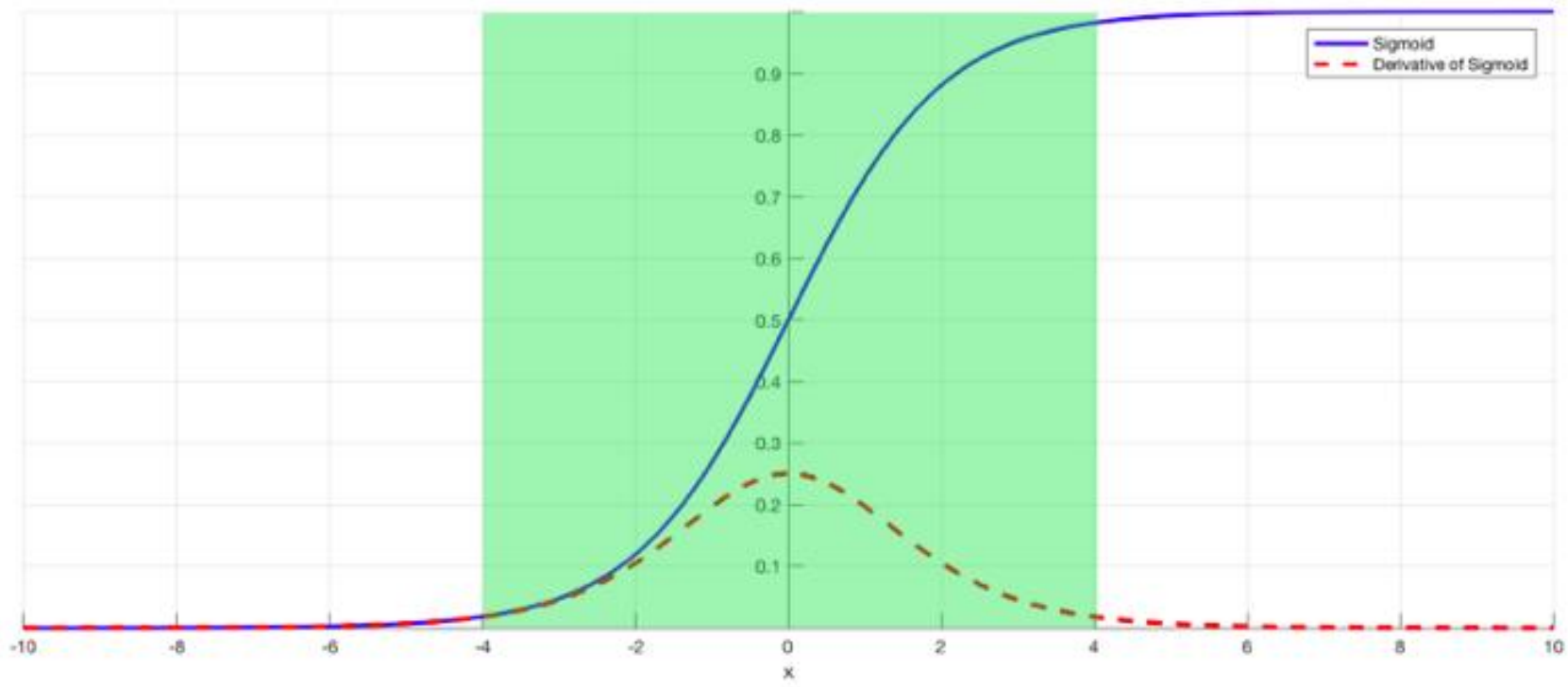
Challenges of Activation Functions

Vanishing Gradient

- In networks using certain activation functions, adding more layers may lead to:
  - Gradients of loss function approaching zero
- Why?:
  - Certain activation functions (eg. sigmoid) squeeze a large input space into a small input space between 0 and 1
  - A large change in input of sigmoid function will cause a small change in output - derivative becomes small
  - For shallow network, not a big problem
  - With more layers - cause more activation function with them, can cause gradient to be too small

Vanishing Gradient



Vanishing Gradient

- Gradients found using backpropagation
  - Finds derivatives of network by moving layer by layer from final layer to initial one
  - By chain rule, derivatives of each layer multiplied down the network to compute derivatives of initial layers
- When *n* hidden layers use an activation like sigmoid
  - *n* small derivatives are multiplied together
  - Gradient decreases exponentially as they propagate down to initial layers
  - Networks are unable to backpropagate gradient information to input layers of model
  - Difficult to train network

Vanishing Gradient

- Small gradient means:
  - Weights and biases of initial layers will not be updated effectively with each training session
  - Since initial layers are crucial to recognize core elements of input data, it can lead to overall inaccuracy of whole network
- Simplest solution
  - Use other activation functions, such as ReLU – does not cause a small derivative
  - Other solutions – residual networks, batch normalization

Exploding Gradient

- In some cases, gradients keep on getting larger and larger as backpropagation algorithm progresses
  - Causes very large weight updates: causes gradient descent to diverge; known as ***exploding gradients*** problem
- Why?
  - Suppose initial weights assigned to network generate some large loss
  - Gradients can accumulate during an update and result in very large gradients
  - Eventually results in large updates to network weights and leads to an unstable network.
  - Parameters can sometimes become so large that they overflow and result in NaN values

Solutions: Exploding Gradient

- Re-design Network Model
  - Redesign network to have fewer layers
  - Can use smaller batch size while training
- Use Long Short-Term Memory Networks
- Use Gradient Clipping
  - Values of error gradient checked against a threshold value
  - Clipped/set to threshold value if error gradient exceeds threshold
- Weight Regularization
  - Check size of network weights and apply a penalty to loss function for large weight values

How to know?

Vanishing Gradient	Exploding Gradient
Parameters of higher layers change significantly whereas parameters of lower layers would not change much	Exponential growth in model parameters
Model weights may become 0 during training	Model weights may become NaN during training
Model learns very slowly and perhaps training stagnates at a very early stage just after a few iterations	Model experiences avalanche learning

Hyperparameters

Hyperparameters

- Number of layers
- Number of neurons
- Batch size
- Epochs
- Learning rate
- Activation functions
- Optimizer

Batch, Epoch, Iteration

- A training dataset can be divided into one or more batches
- **Batch size** - hyperparameter that defines number of samples to work through before updating internal model parameters
  - At end of batch, predictions are compared to expected output variables and error is calculated
  - From this error, the update algorithm is used to improve model, e.g. move down along error gradient
- Batch size - number of samples processed before model is updated

  $1 \leq$ Batch size $\leq$ number of samples in training dataset

Batch, Epoch, Iteration

- **Epoch** - hyperparameter that defines number times learning algorithm will work through entire training dataset
  - Comprises of one or more batches
  - One epoch: each sample in training dataset has had a chance to update internal model parameters
  - A *for*-loop over number of epochs - each loop proceeds over complete training dataset
  - Within this *for*-loop: another nested *for*-loop that iterates over each batch of samples, of specified “batch size”
  - Number of **iterations** is equivalent to number of batches needed to complete one epoch

Batch, Epoch, Iteration

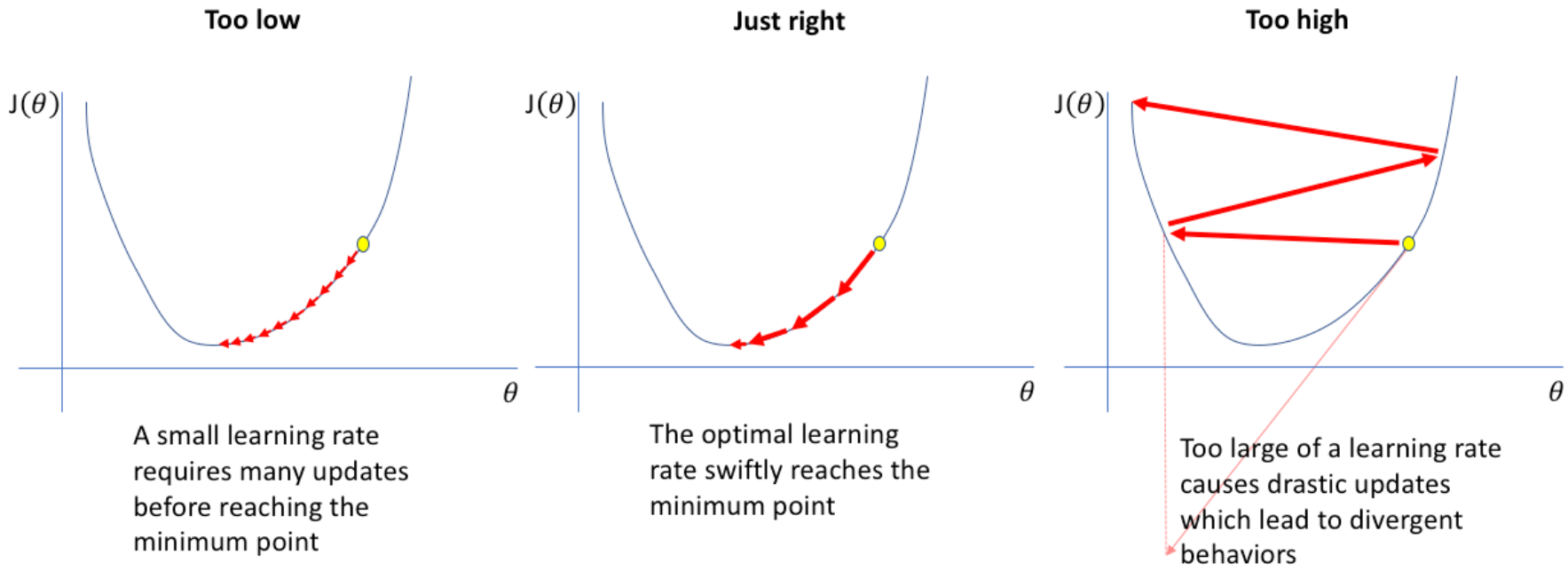
- Assume dataset with 200 samples
  - Ex. Batch size = 5, Epochs = 1,000
  - Dataset will be divided into 40 batches, each with five samples
  - Model weights will be updated after each batch of five samples
  - One epoch will involve 40 batches or 40 updates to model (40 iterations)
  - 1,000 epochs: model will be passed through whole dataset 1,000 times
    - Total of 40,000 batches during entire training process

Learning Rate

- Learning rate (step size)
  - Configurable hyperparameter
  - Controls how much to change model in response to estimated error when model weights are updated
  - Small positive value, often in range between 0.0 and 1.0
- Large learning rate
  - May cause model to learn a sub-optimal set of weights too fast or an unstable training process
- Small learning rate
  - Requires more training epochs
  - May result in a long training process

Learning Rate

- How to set the learning rate, η?
  - Small η converges slowly; gets stuck in local minima
  - Large η overshoots, becomes unstable and diverges
  - Stable η converges smoothly and avoids local minima



https://www.jeremyjordan.me/nn-learning-rate/

Batch Gradient Descent

- Batch Size = Size of Training Set
- Batch gradient descent computes gradient of cost function w.r.t. to parameters (*w,b*) for entire training dataset
  - Computationally intensive
  - Summation over all data points in dataset in each iteration
  - Can be very slow
  - Difficult to deal with datasets that do not fit in memory
  - Converges to global minimum for convex error surfaces and to a local minimum for non-convex surfaces

Stochastic Gradient Descent (SGD)

- Batch Size = 1
- Performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$
- Algorithm:
  1. Initialize weights randomly
  2. Loop until convergence:
     1. Pick single data point ***i***
     2. Compute gradient $\partial \boldsymbol{J}(\boldsymbol{W})/\partial \boldsymbol{W}$
     3. Update weights $\boldsymbol{W} := \boldsymbol{W} - \boldsymbol{\eta} * \partial \boldsymbol{J}(\boldsymbol{W})/\partial \boldsymbol{W}$
  3. Return weights

Mini Batch Gradient Descent

- 1 < Batch Size < Size of Training Set
  - Popular batch sizes include 32, 64, and 128 samples
- Performs an update for every mini-batch of *n* training examples
- Algorithm:
  1. Initialize weights randomly
  2. Loop until convergence:
     1. Pick batch of ***n*** data points
     2. Compute gradient $\partial J(W)/\partial W = (1/n) * \sum \partial J_k(W)/\partial W$ $\quad k = 1 \ldots n$
     3. Update weights $W := W - \eta * \partial J(W)/\partial W$
  3. Return weights

Regularization

Bias

- To make predictions, model analyzes data and finds patterns in it
- Using these patterns, we can make generalizations about certain instances in data
- Model, after training, learns these patterns and applies them to test set to predict them
- **Bias** is the difference between actual and predicted values
- It is the simple assumptions that model makes about data to be able to predict new data

Bias

- Difference between expected (or average) prediction of our model and correct value
  - High Bias - when error rate has a high value
  - Low Bias - when error rate has a low value
- When bias is high, assumptions made by model are too basic
- This instance, where model cannot find patterns in training set and hence fails for both seen and unseen data, is called **Underfitting**
  - Model unable to capture relationship between input and output variables accurately
  - Generates a high error rate on both training set and unseen data

Underfitting

- Occurs when model is too simple
  - Can be a result of a less training time, more input features, or less regularization
- Some ways to reduce high bias:
  - Increase input features as model is underfitted
  - Increase training time
  - Decrease regularization term
  - Use more complex models, such as including some polynomial features

Variance

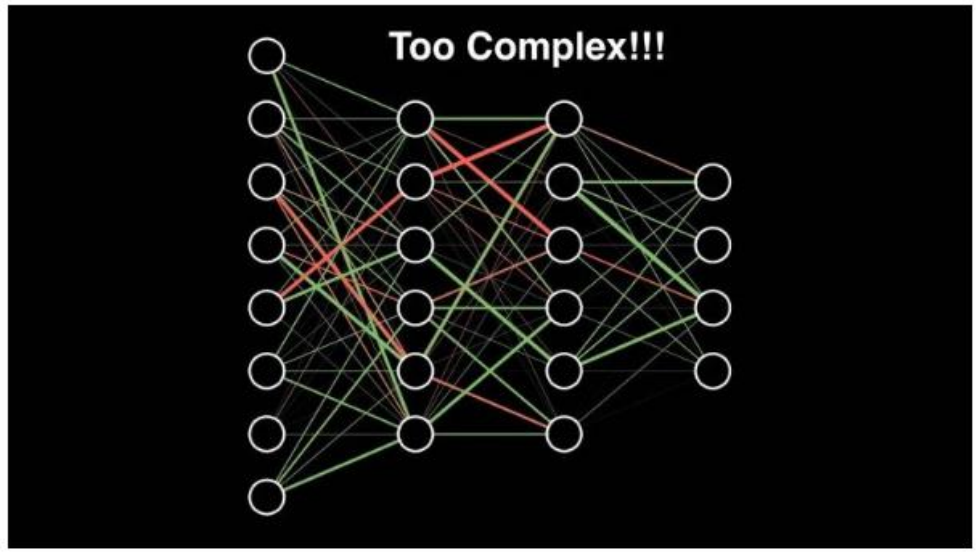
- During training, allow model to ‘see’ data a certain number of times to find patterns in it
- If it does not work on data long enough - bias occurs
- If model allowed to view data too many times, it will learn well for only that data
  - Will capture most patterns, but will also learn from unnecessary data present, or from noise
  - Will cause model to consider trivial features as important
  - Will tune itself to data, and predict it very well
  - **When given new data, it cannot predict on it as it is too specific to training data**
- Can define variance as model’s sensitivity to fluctuations in data

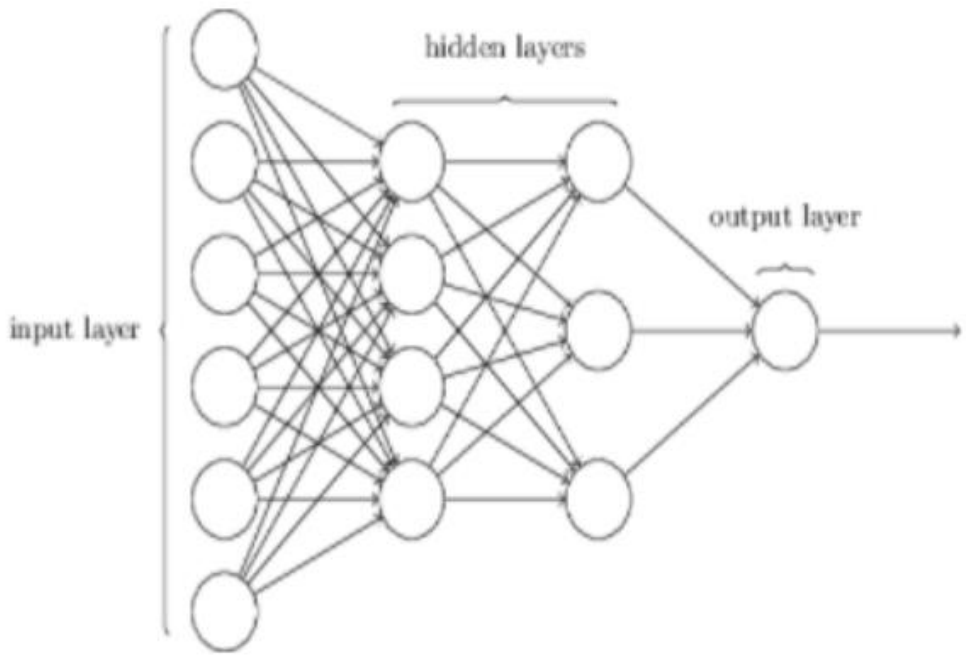
Variance

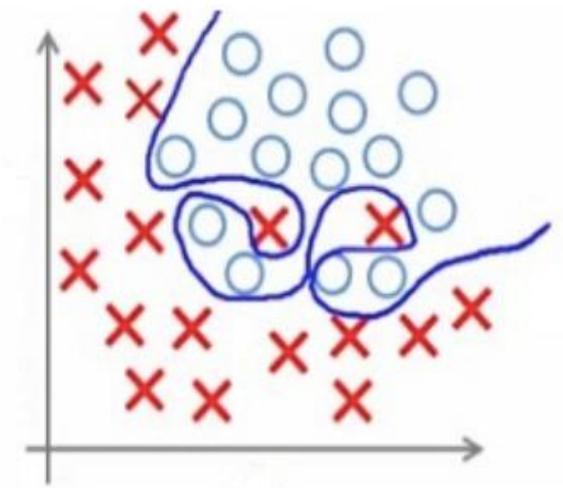
- Difference between error rate of training data and testing data
  - **Low variance -** small variation in prediction of target function with changes in data
  - **High variance -** large variation in prediction of target function with changes in data
- Model learns a lot and perform well with training dataset
  - Does not generalize well with unseen dataset
  - Gives good results with training dataset but shows high error rates on test dataset

Overfitting

- Phenomenon where network models training data very well but fails when it sees new data from same problem domain
- Caused by noise in training data that network picks up during training and learns it as an underlying concept of data
- Learned noise unique to each training set
  - As soon as model sees new data from same problem domain, but that does not contain this noise, performance of network gets worse
- ***"Why does neural network picks up that noise in the first place?"***
  - Complexity of network is too high





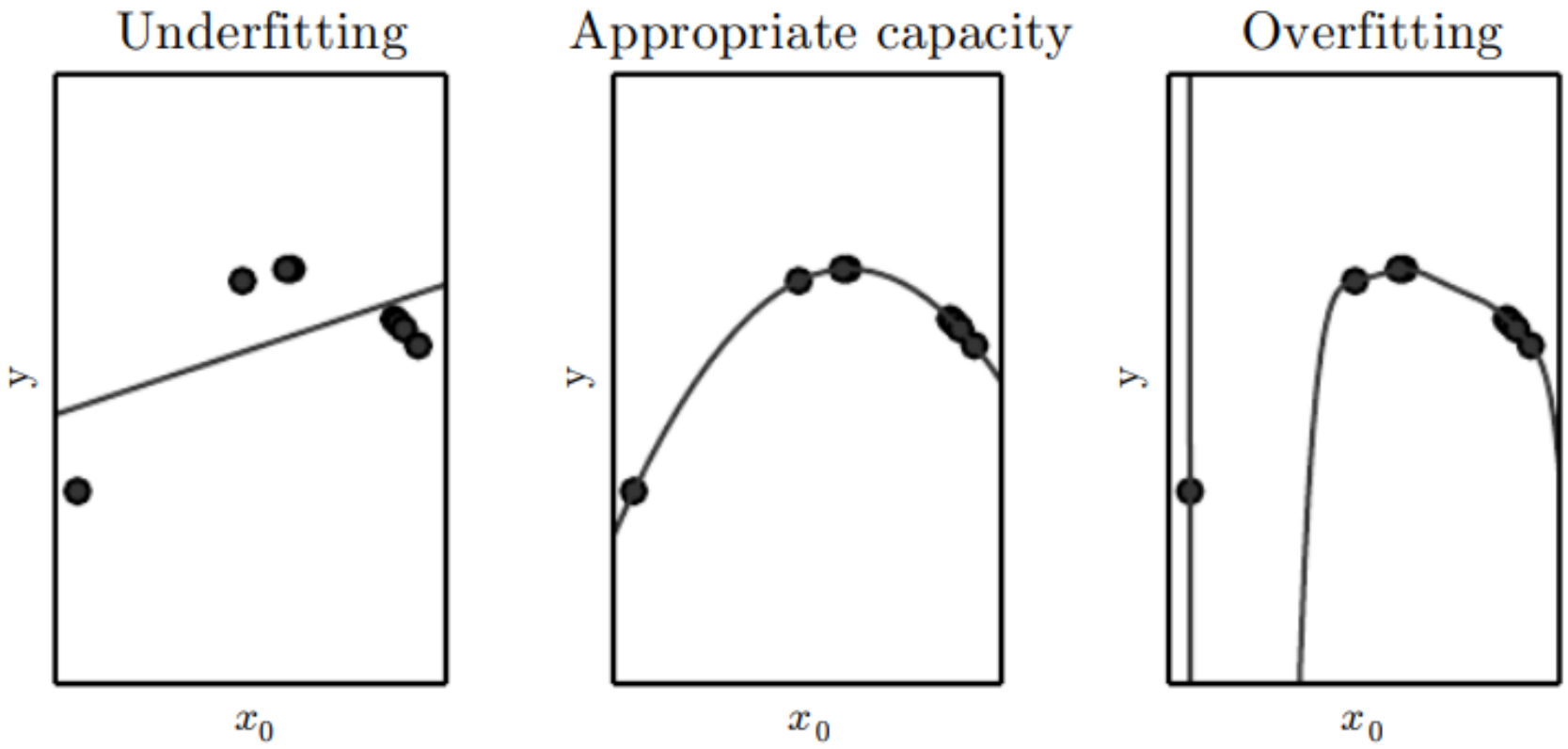


Over-fitting

Variance

- Ways to Reduce High Variance:
  - Reduce input features or number of parameters
  - Do not use a much complex model
  - Increase training data
  - Increase Regularization term

Underfitting, Overfitting

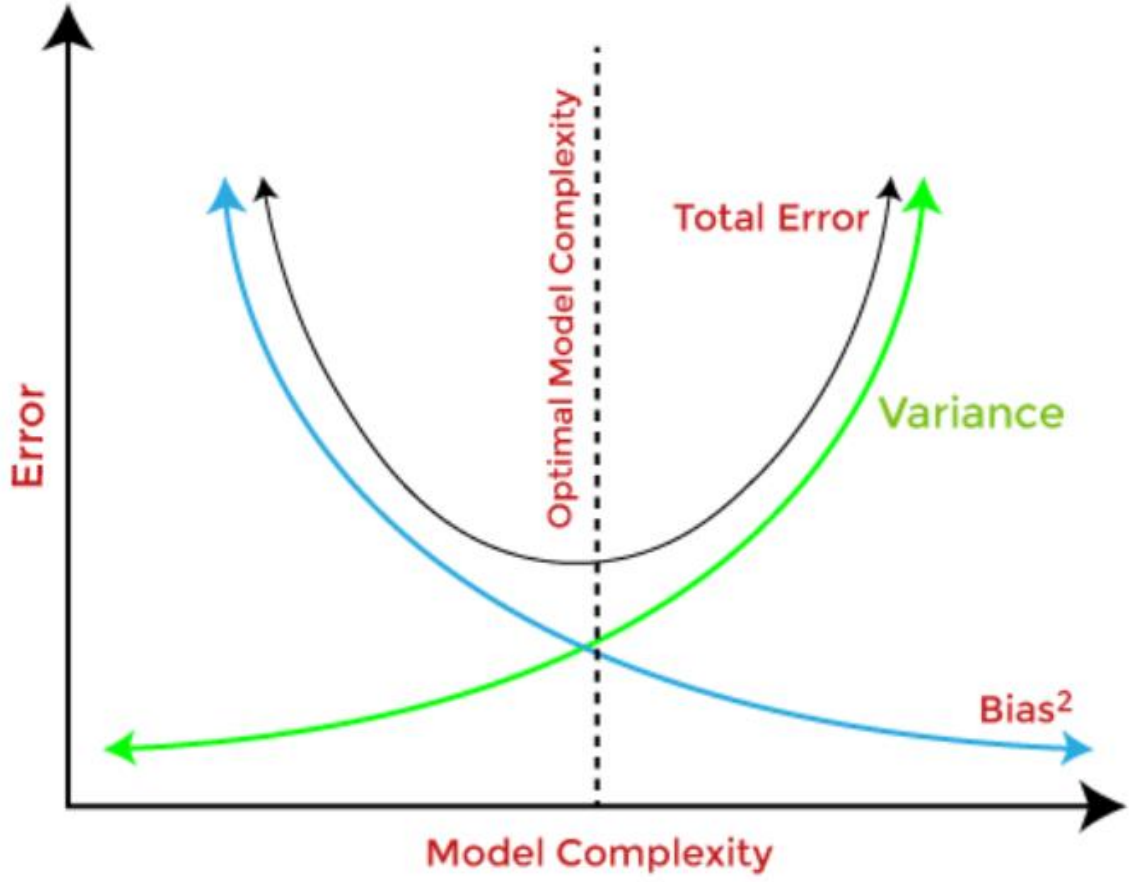


Bias-Variance Trade-off

- For any model, have to find perfect balance between bias and variance
  - Capture essential patterns while ignoring noise present in it
  - Called **Bias-Variance Tradeoff**
  - Helps optimize error in model and keeps it as low as possible

- An optimized model is sensitive to patterns in data, but able to generalize to new data
  - Both bias and variance should be low to prevent overfitting and underfitting

Bias-Variance Trade-off



Bias-Variance Trade-off

- Can tackle trade-off in multiple ways:
- **Increasing complexity of model**
  - Decreases overall bias while increasing variance to an acceptable level
  - Aligns model with training dataset without incurring significant variance errors
- **Increasing training data set**
  - Preferred method when dealing with overfitting models
  - Allows users to increase complexity without variance errors
  - A large data set offers more data points for algorithm to generalize data easily

Bias-Variance Trade-off

- How to make an algorithm that will perform well not just on training data, but also on new inputs
- Many strategies explicitly designed to reduce test error, possibly at expense of increased training error
- These strategies collectively known as **regularization**

Regularization

- Regularization is modification to a learning algorithm that is intended to reduce its generalization error but not its training error
- Refers to a set of different techniques that lower complexity of a neural network model during training, prevent overfitting
  - Penalizes weight matrices of nodes
- Three very popular and efficient regularization techniques:
  - *L1*, *L2*, and dropout

L1 regularization

- Also known as **Lasso regression**
- Use another regularization term **Ω, s**um of absolute values of weight parameters in a weight matrix:

$$\Omega(W) = ||W||_1 = \sum \sum |w_{ij}| \qquad \text{for all } i, j$$

- New loss function:

$$L'(W) = \lambda\ ||W||_1 + L(W)$$

- Derivative of new loss function

$$\frac{\partial L'(W)}{\partial W} = \lambda * sign(W_{old}) + \frac{\partial L(W)}{\partial W}$$

$$W_{new} = W_{old} - \eta * (\lambda * sign(W_{old}) + \frac{\partial L(W)}{\partial W})$$

L2 regularization

- L2 regularization is most common type of all regularization techniques
  - Commonly known as **weight decay** or **Ridge Regression**
- Loss function of network extended by a **regularization term**, **Ω**

$$\Omega(W) = ||W||^2_2 = \sum \sum w^2_{ij} \qquad \text{for all } i, j$$

- **Ω** defined as Euclidean Norm (or L2 norm) of weight matrices
  - Sum over all squared weight values of a weight matrix
- New expression for loss function:

$$L'(W) = (\lambda/2)||W||^2_2 + L(W)$$

λ is **regularization rate** and is an additional hyperparameter ($0 < \lambda < \infty$)

L2 regularization

- In next step, compute gradient of new loss function and put gradient into update rule for weights:

$$\nabla_W L'(W) = \lambda * W + \frac{\partial L(W)}{\partial W}$$

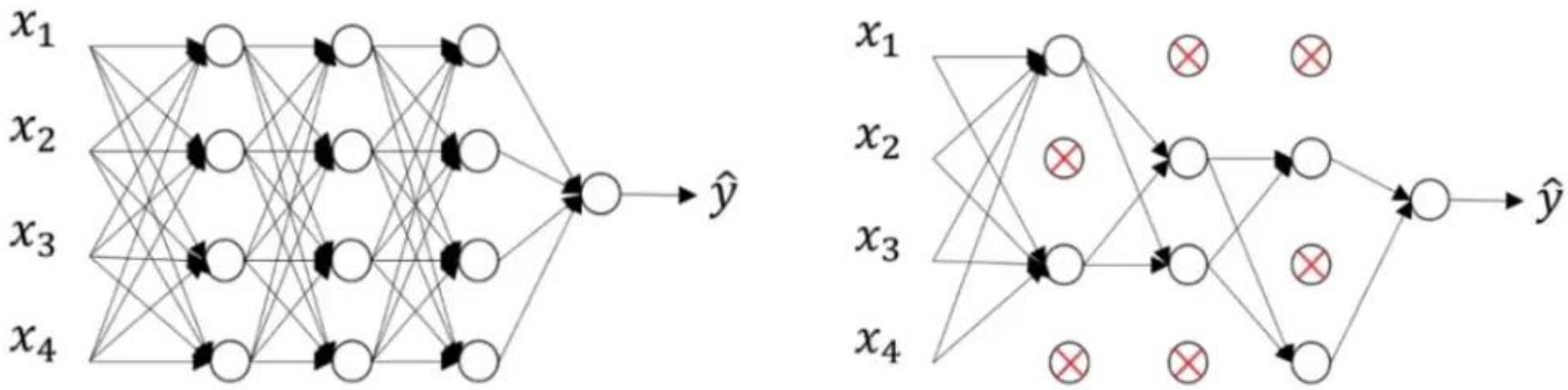
$$W_{new} = W_{old} - \eta * (\lambda * W_{old} + \frac{\partial L(W)}{\partial W})$$

$$= \mathbf{(1 - \eta * \lambda) * W_{old}} + \eta * \frac{\partial L(W)}{\partial W} \qquad \textbf{//weight decay}$$

- L2 regularization penalizes weights ($z = wx + b$)
  - A gentle slope indicates lower impact of variable on z
  - $g(z)$ will be comparatively linear
  - This reduces non-linearity

Dropout

- Dropout means that during training, with some probability **P,** a neuron of neural network gets turned off during training
  - Cannot rely on one feature; spread out the weights
  - 0 < P < 1
- Using dropout, with let's say a probability of **P=0.5** that a random neuron gets turned off during training:



Dropout

- Each iteration has a different set of nodes
  - Results in a different set of outputs
  - **Can also be thought of as an ensemble technique**
  - Ensemble models usually perform better as they capture more randomness
- May be implemented on any or all hidden layers in network as well as on input layer; not used on output layer
- Probability of choosing how many nodes should be dropped is hyperparameter of dropout function
  - Typical values: $P = 0.5$ in a hidden layer and P is close to 0, such as 0.2, for input layer
- Weights of network will be larger than normal because of dropout
  - Before finalizing network, weights are scaled by chosen dropout rate
- Not used after training when making a prediction with test data

Early Stopping

- A major challenge in training neural networks - how long to train?
- Too little training - model will underfit train and test sets
- Too much training - model will overfit training dataset and have poor performance on test set
- Compromise: train on training dataset but to stop training when performance on a validation dataset starts to degrade
- During training - model evaluated on a holdout validation dataset after each epoch
- If performance of model on validation dataset starts to degrade, training process is stopped

Early Stopping

- Early stopping is a kind of cross-validation strategy where we keep one part of training set as validation set
  - When performance on validation set starts getting worse, stop training on model. This is known as **early stopping**

Early Stopping

- **Patience -** number of epochs with no further improvement after which training will be stopped
- After dotted line, each epoch will result in a higher value of validation error
  - 5 epochs after dotted line (since our patience is equal to 5), model will stop because no further improvement is seen
  - *May be possible that after 5 epochs (this is the value defined for **patience** in general), model starts improving again and validation error starts decreasing. Need to take extra care while tuning this hyperparameter*

Normalization

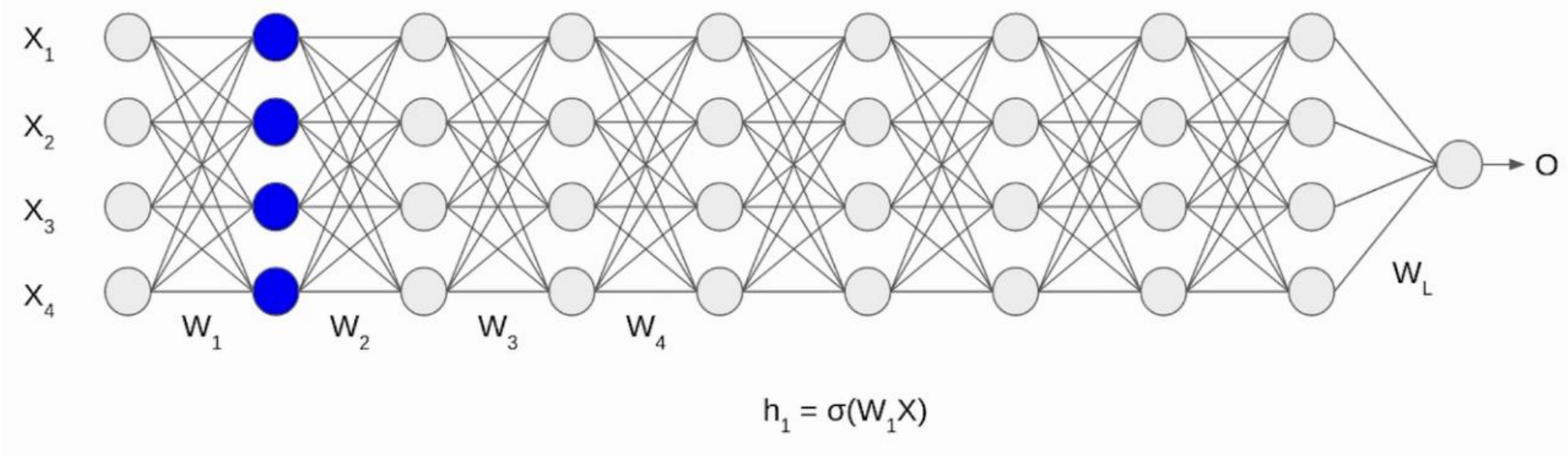
- Training deep neural networks is challenging as they can be sensitive to initial random weights and configuration of learning algorithm
- One possible reason:
  - Distribution of inputs to layers deep in network may change after each mini-batch when weights are updated
  - Can cause learning algorithm to forever chase a moving target
  - Change in distribution of inputs to layers in network referred to as **internal covariate shift**

Batch Normalization

- **Normalization -** a data pre-processing tool used to bring numerical data to a common scale without distorting its shape
  - Partly to ensure that model can generalize appropriately
- **Batch normalization -** technique for training very deep neural networks that normalizes inputs to a layer for every mini-batch
  - Distribution of each mini batch can be different after each layer; training can be hard
  - Performs standardizing and normalizing operations on input of a layer coming from a previous layer
  - Normalizing process takes place in batches, not as a single input
  - Settles learning process and drastically decreasing number of training epochs required to train deep neural networks

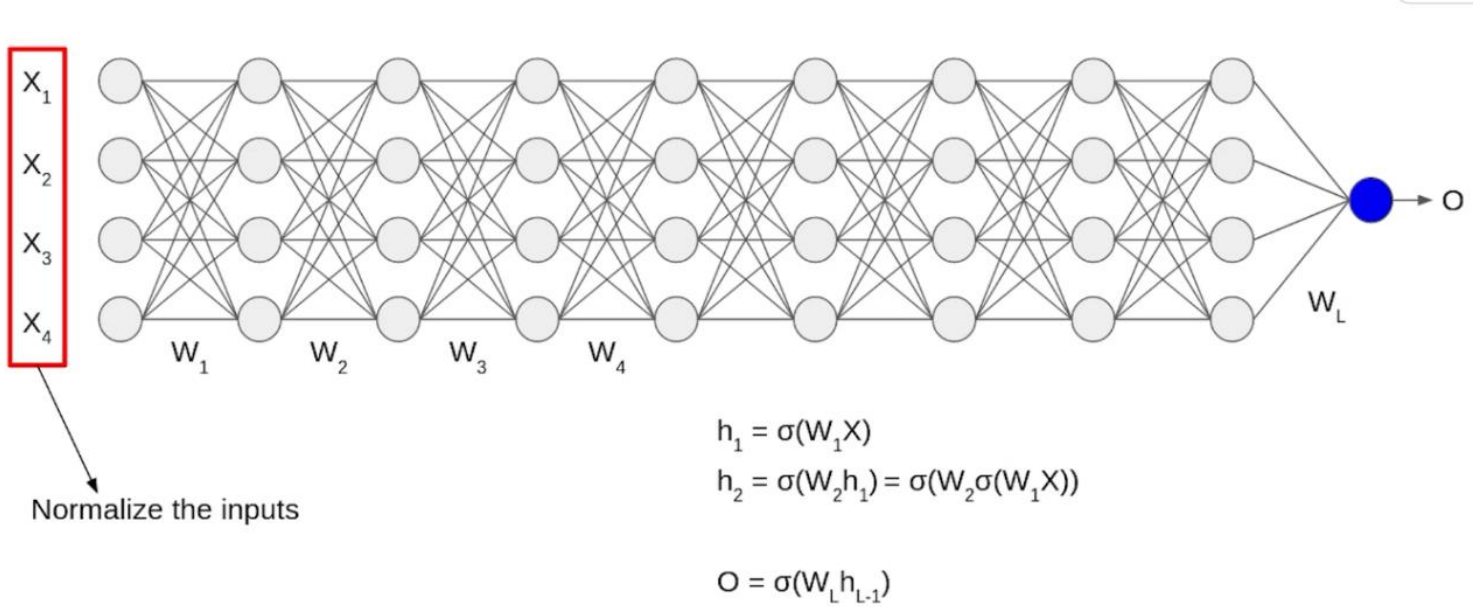
Batch Normalization

- Inputs X1, X2, X3, X4 are in normalized form as they are coming from pre-processing stage
- When input passes through first layer, it transforms (assuming bias =0, here)



Batch Normalization

- Transformation takes place for second layer and go till last layer *L*
- Although, input **X** was normalized, output no longer on same scale
- As data goes through multiple layers and *L* activation functions are applied, it leads to an internal co-variate shift in data



Batch Normalization

- Two-step process:
  1. Normalize input
  2. Rescaling and offsetting done
- **Normalization of Input:**
  - Transform data to have a zero mean and standard deviation one
  - Assume, batch input from layer *h*
  - Mean of this hidden activation

$$\mu = (1/m)*\sum h_i$$

m is number of neurons at layer *h*

Batch Normalization

- Calculate standard deviation of hidden activations

$$\sigma = [(1/m)*\sum (h_i - \mu)^2]^{1/2}$$

- Normalize hidden activations - subtract mean from each input and divide by sum of standard deviation and smoothing term (ε)
  - Smoothing term (ε) avoids division by a zero value

$$h_{i(norm)} = (h_i - \mu)/(\sigma + \varepsilon)$$

- Final step: re-scaling and offsetting of input – let network decide the normalization weights
- Parameters used: for re-scaling (γ) and shifting (β)

$$h_i = \gamma * h_{i(norm)} + \beta$$

  - Learnable parameters - during training network ensures optimal values of γ and β used
  - Will enable accurate normalization of each batch