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

high bias → high error

↓

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

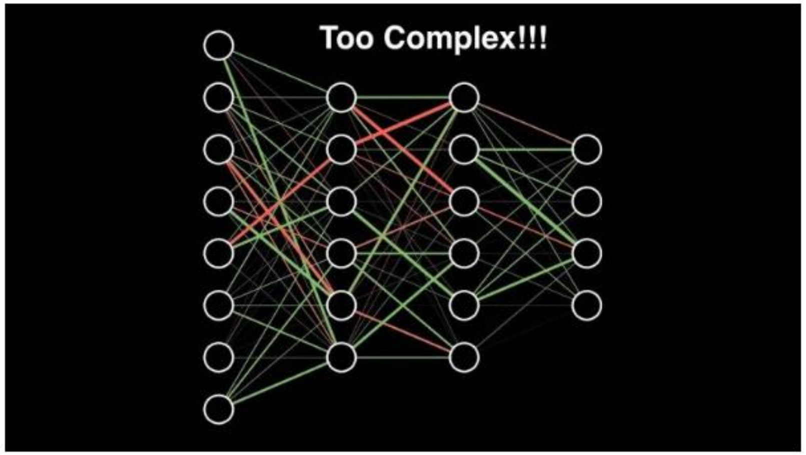
Variance

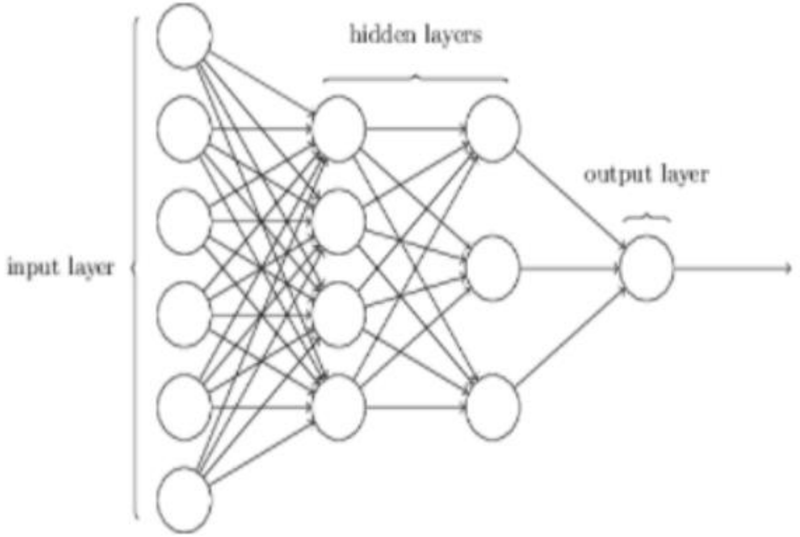
high variance
$\hookrightarrow$ overfitting

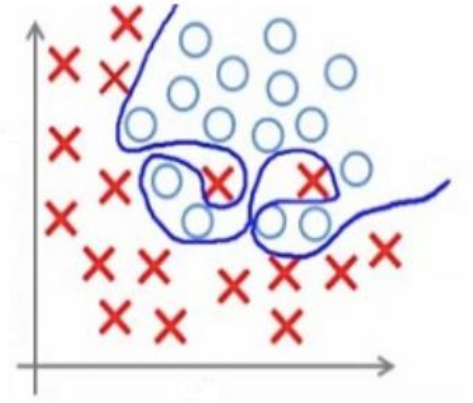
- Difference between error rate of training data and testing data
  - **Low variance -** small variation in prediction of target function with changes in data
  - **High variance -** large variation in prediction of target function with changes in data
- Model learns a lot and perform well with training 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





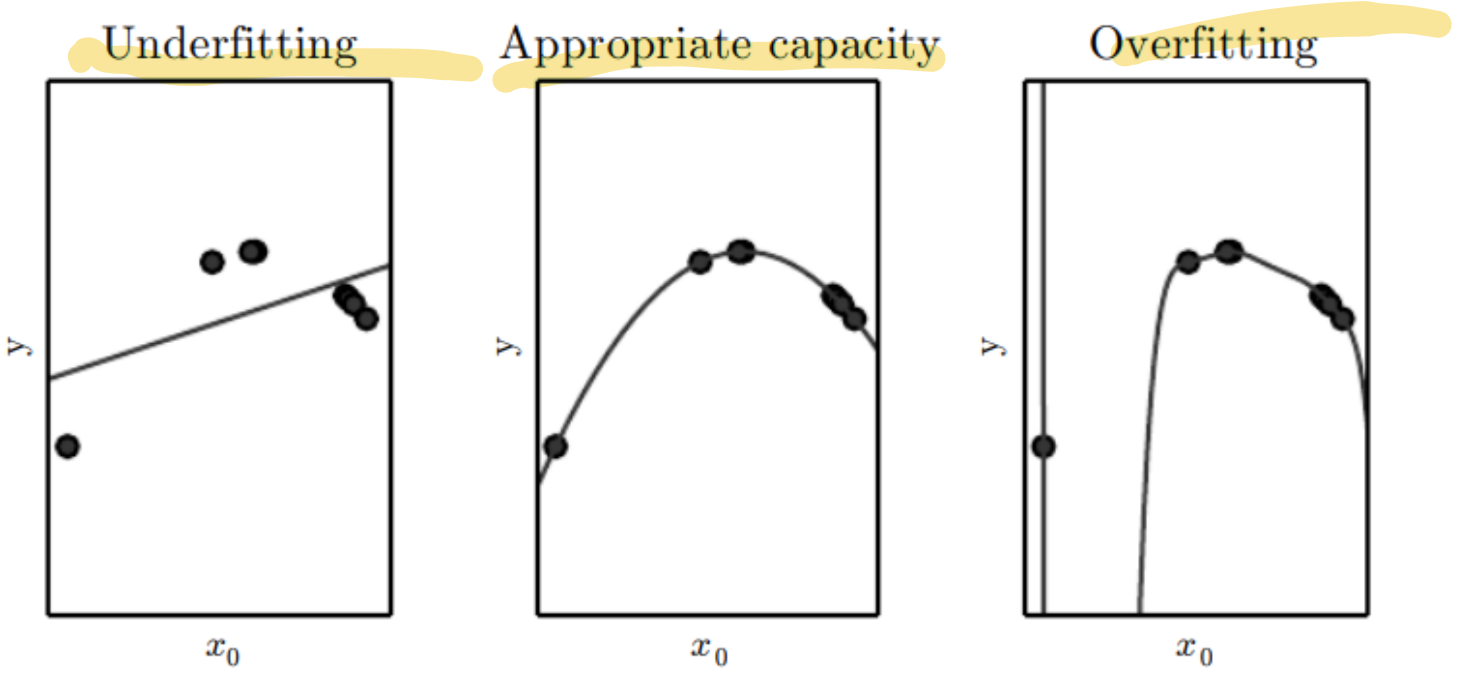


Over-fitting

Variance

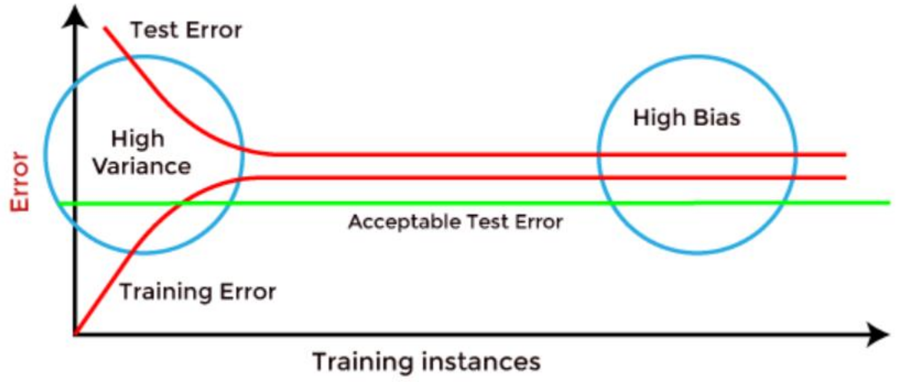
- Ways to Reduce High Variance:
  - Reduce input features or number of parameters
  - Do not use a much complex model
  - Increase training data
  - Increase Regularization term

Underfitting, Overfitting



Bias vs. Variance

- High Bias can be identified if model has high training error and test error is almost similar to training error
- High variance can be identified if model has low training error and high test error

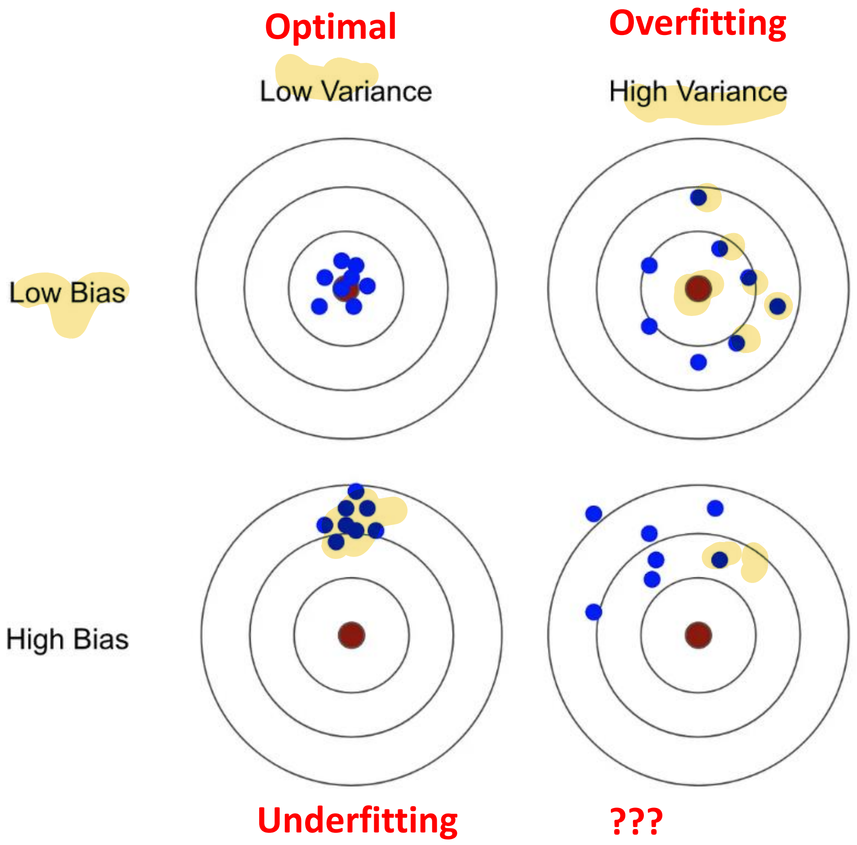


Bias-Variance Trade-off

- 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

Bull's Eye Diagram - Bias-Variance Trade-off



Bias-Variance Trade-off

1. Low-Bias, Low-Variance:

- An ideal machine learning model. However, not possible practically

2. Low-Bias, High-Variance:

- Model learns with a large number of parameters and hence leads to **overfitting**

3. High-Bias, Low-Variance:

- Model does not learn well with training dataset or uses few numbers of parameters; leads to **underfitting** problems

4. High-Bias, High-Variance:

- Predictions are inconsistent and also inaccurate on average

Bias-Variance Trade-off

- Assume:
  - **Y** - variable we are trying to predict
  - **X** - covariates
  - Relationship: $\mathbf{Y} = f(\mathbf{X}) + \epsilon$
  - Error term ϵ is normally distributed with a mean of zero $\sim N(0, \sigma_\epsilon)$
  - $E[\epsilon] = 0$, $\text{var}[\epsilon] = E[\epsilon^2] = \sigma_\epsilon^2$
- May estimate model $f'(\mathbf{X})$ using any modeling technique
- Expected squared prediction error at a point x is:

$$Err(x) = E\left[(Y - f'(x))^2\right]$$

Bias-Variance Trade-off

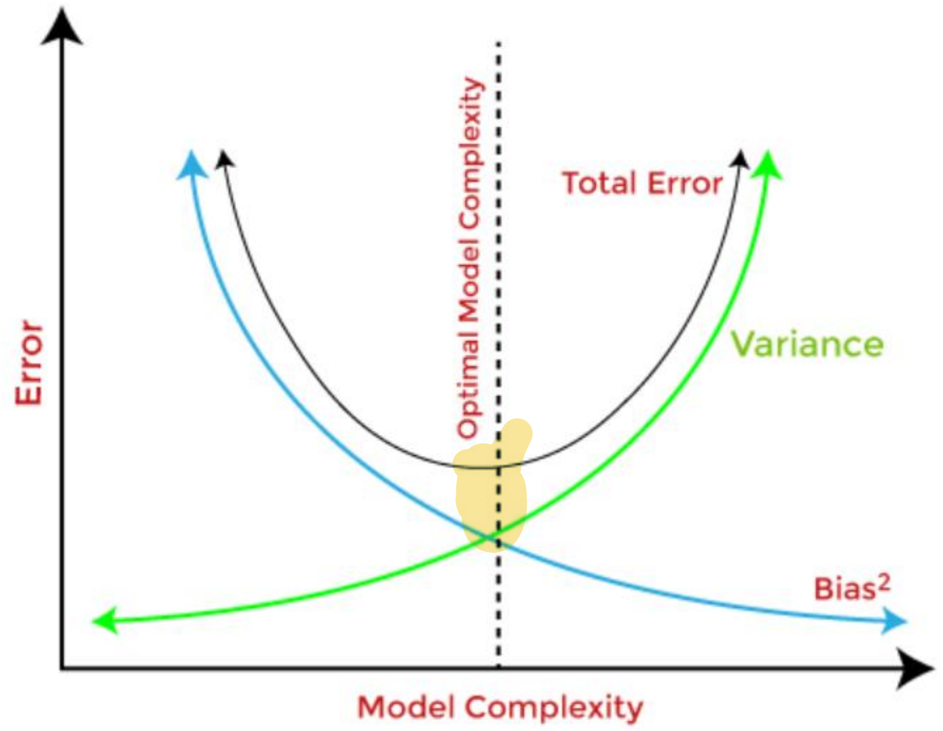
- $Err(x) = E\,[(Y - f'(x))^2]$
  $= E[(f(x) + \epsilon - f'(x))^2]$
  $= E[(f(x) - f'(x))^2] + E[\epsilon^2] + 2E[(f(x) - f'(x))\,\epsilon]$
  $= E[(f(x) - f'(x))^2] + \sigma_\epsilon^2 + 2E[(f(x) - f'(x))]E[\epsilon]$
  $= E[(f(x) - f'(x))^2] + \sigma_\epsilon^2$ E[ϵ] = 0 (as it is noise)
  $= E[((f(x) - E[f'(x)]) - (f'(x) - E[f'(x)]))^2] + \sigma_\epsilon^2$ Adding/subtracting E[f'(x)]
  $= E[(E[f'(x)] - f(x))^2] + E[(f'(x) - E[f'(x)])^2] - 2E[(f(x) - E[f'(x)])(f'(x) - E[f'(x)])] + \sigma_\epsilon^2$
- Error may be decomposed into bias and variance components
  $Err(x) = (E[f'(x)] - f(x))^2 + E[(f'(x) - E[f'(x)])^2]) + \sigma_\epsilon^2$
  $Err(x) = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$

https://en.wikipedia.org/wiki/Mean_squared_error

Bias-Variance Trade-off

- Irreducible error is noise term in the true relationship
    - Cannot fundamentally be reduced by any model
- Given true model and infinite data to calibrate it, should be able to reduce both bias and variance terms to 0
    - However, in a world with imperfect models and finite data, a tradeoff between minimizing bias and variance

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**

$\downarrow$ reduce test error

Bias-Variance Trade-off

- Central challenge that model must perform well on new, previously unseen inputs, not just those on which model was trained
- Ability to perform well on previously unobserved inputs is called **generalization**
- When training a model, can compute error measure on training set called *training error*
  - Reduce training error - an optimization problem
- Want *generalization error*, also called *test error,* to be low as well
  - Defined as expected value of error on a new input

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

$\downarrow$ Sparsy $\qquad \searrow$

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:

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

$$L'(w) = \frac{\lambda}{2}||w||^2_2 + L(w)$$

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

$\nabla_w L(w)$

L2 regularization

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

$$\nabla_W L'(W) = \lambda * W + \nabla_W L(W)$$

$$W_{new} = W_{old} - \eta * (\lambda * W_{old} + \nabla_W L(W)) \qquad C$$

$$= (1 - \eta * \lambda) * W_{old} + \eta * \nabla_W L(W)$$

//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

L2 regularization ✓

- L2 regularizer adds penalty as model complexity increases
- Forces weights to be small but does not make them zero
- Can learn complex data patterns
- Non-sparse solution ✓
- Not robust to outliers

L1 regularization

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

$$\Omega(W) = ||W||_1^{2} = \sum \sum |w_{ij}| \qquad \text{for all } i, j$$

- New loss function:

$$L'(W) = \lambda/2\, ||W||_1 + L(W)$$

- Derivative of new loss function

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

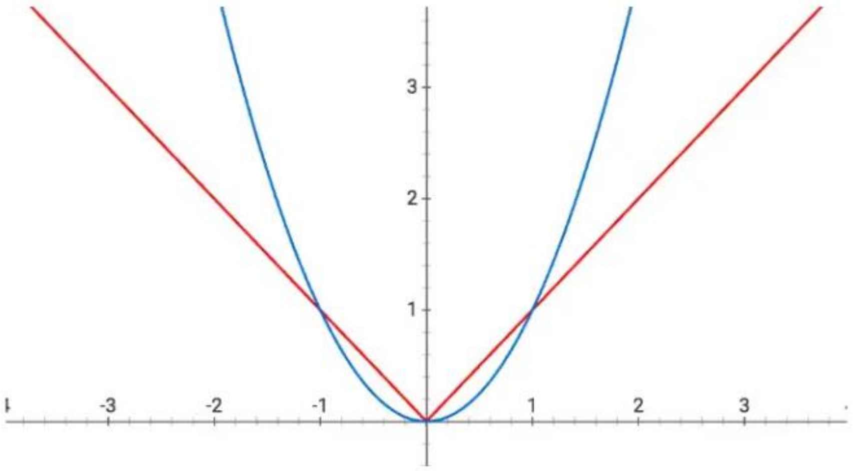
- $W_{new} = W_{old} - \eta * (\lambda * \underline{sign}(W_{old}) + \nabla_W L(W))$

L1 regularization

- L1 regularizer looks for parameter vectors that minimize norm of parameter vector (length of vector)
- Shrinks coefficient of less important features to zero
- Works well for feature selection where there are large number of features
- Robust to outliers (ex. noise)

L1, L2 regularization

- In case of L2 regularization, weight parameters decrease, but not necessarily become zero, since curve becomes flat near zero
- During L1 regularization, weights are always forced towards zero



L1 function (red), L2 function (blue).

L1, L2 regularization

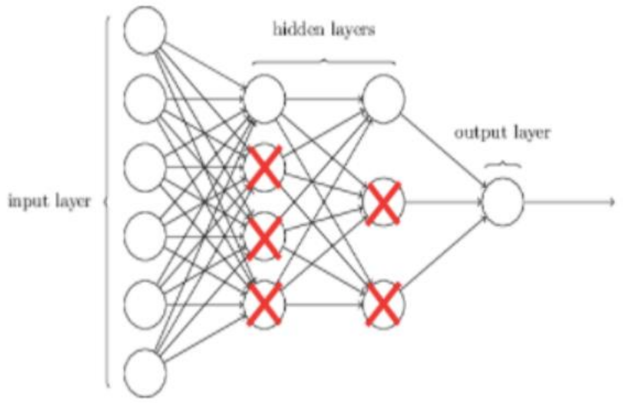
- L2: can think of solving an equation, where sum of squared weight values is equal or less than a value ***s***
  - **s** is the constant that exists for each possible value of regularization term λ
  - For just two weight values **W1** and **W2:** $W1^{2} + W^{22} \leq s$
- L1: can be thought of as an equation where sum of modules of weight values is less than or equal to a value ***s***
  - $|W1| + |W2| \leq s$

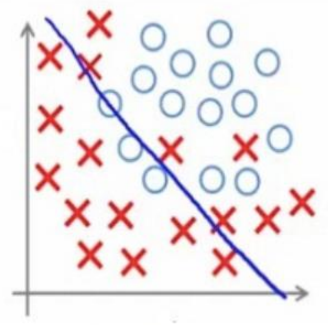
L1, L2 regularization

- Performing L2 regularization encourages weight values towards zero (but not exactly zero)
- Performing L1 regularization encourages weight values to be zero
- Intuitively, smaller weights reduce impact of hidden neurons
    - Those hidden neurons become neglectable and overall complexity of neural network gets reduced
    - Less complex models typically avoid modeling noise in data, and therefore, there is no overfitting

L1, L2 regularization

- When choosing the regularization term λ
  - Goal is to strike right balance between low complexity of model and accuracy
- If λ is too high, model will be simple, but risk of *underfitting* data
  - Some of weight matrices nearly equal to zero
- If λ is too low, model will be more complex, risk of *overfitting* data

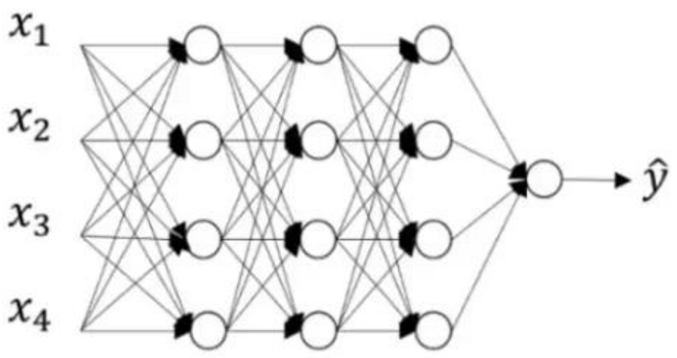


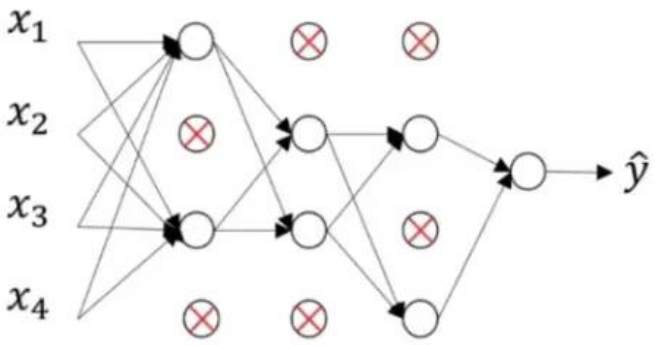


Under-fitting

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

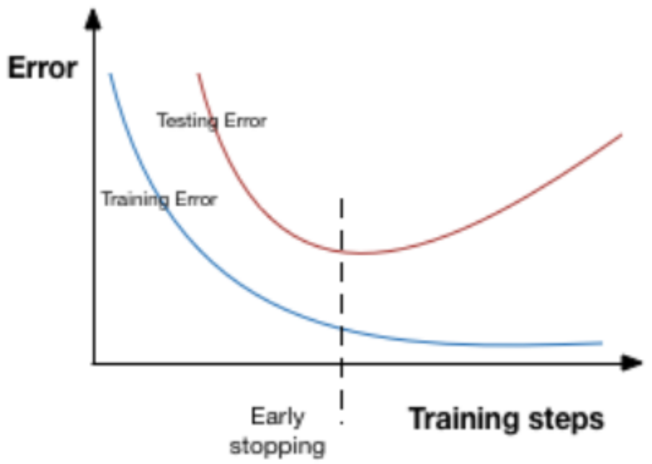
Early Stopping

validation

- 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

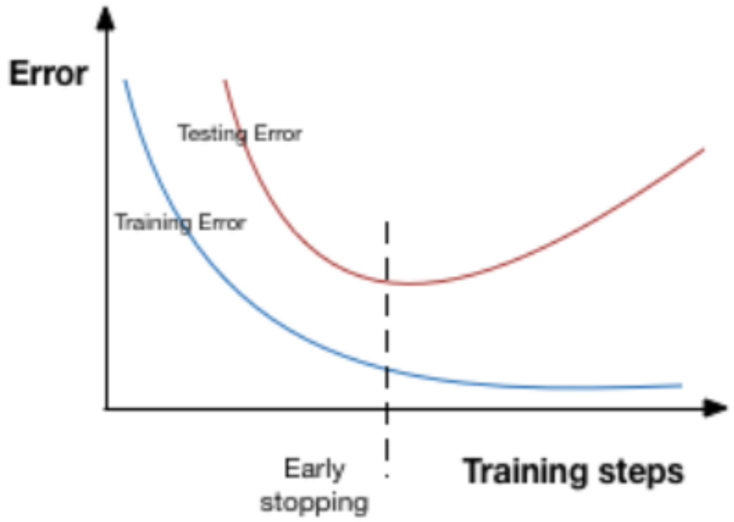
Early Stopping

- 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*



Age Salary

0 - 90 $\quad$ 1 $\quad \frac{x-\mu}{\sigma}$

Normalization

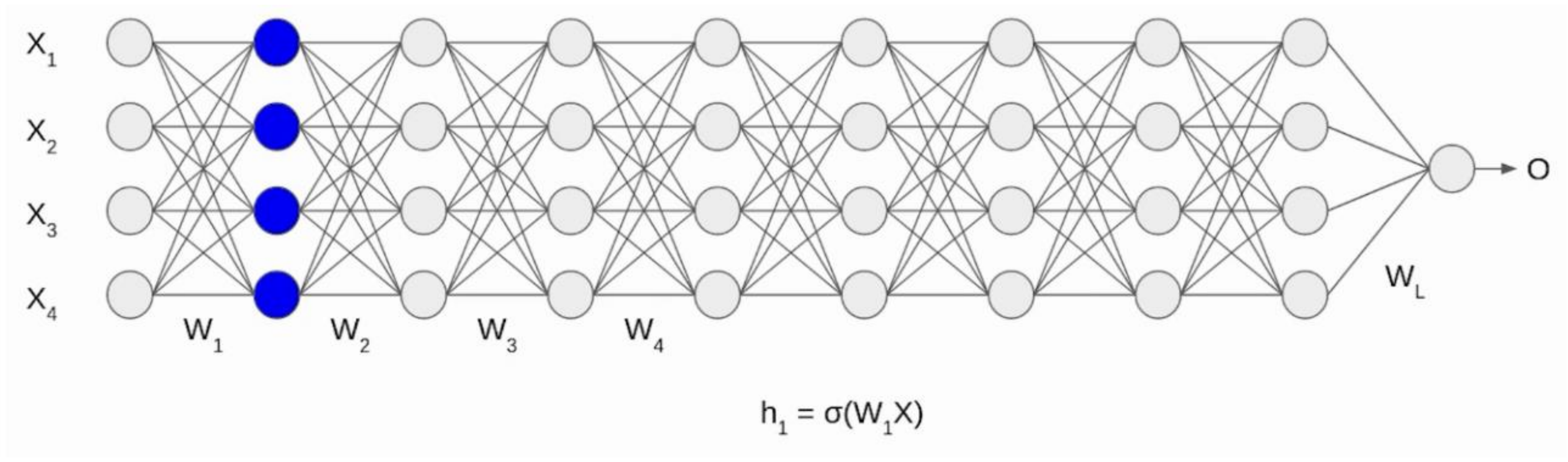
- Training deep neural networks is challenging as they can be sensitive to initial random weights and configuration of learning algorithm
- One possible reason:
  - Distribution of inputs to layers deep in network may change after each mini-batch when weights are updated
  - Can cause learning algorithm 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

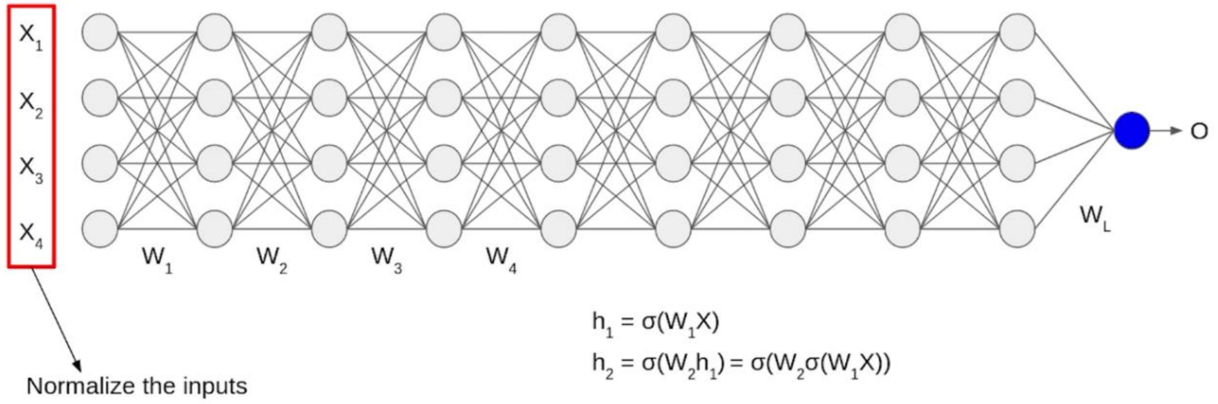
Batch Normalization

- 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



$$h_1 = \sigma(W_1X)$$
$$h_2 = \sigma(W_2h_1) = \sigma(W_2\sigma(W_1X))$$

$$O = \sigma(W_Lh_{L-1})$$

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

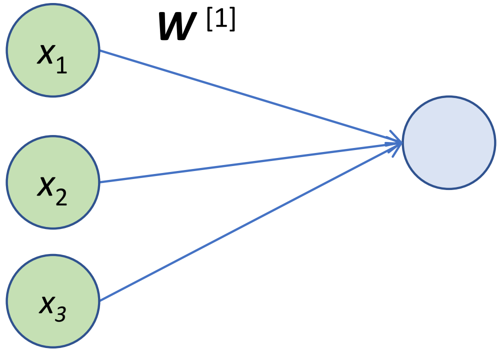
$$h_{i(norm)} = (h_i - \mu)/(\sigma + \varepsilon)$$

- 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

Batch Normalization



Normalizing input features can speed up learning

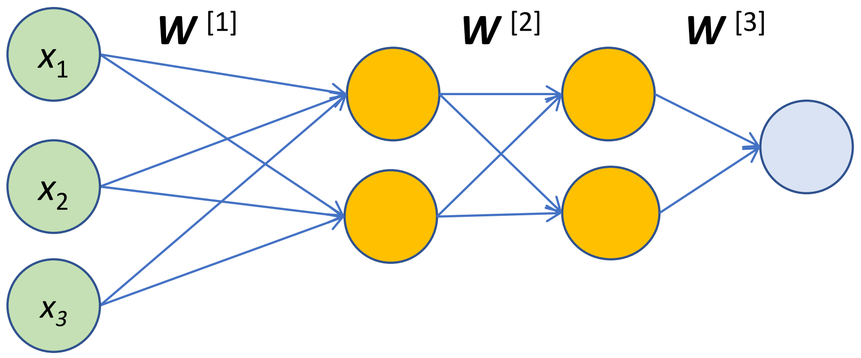
$\mu = (1/m) * \sum X^{(\mathrm{i})}$

$X = X - \mu$

$\sigma^2 = (1/m) * \sum [X^{(i)}]^2$

$X = X/\sigma$

Batch Normalization



Can normalize input features to all layers?)
Normalize before or after activation?
Usually, normalization is done after applying activation functions

$$X \dashrightarrow g(z^{[1]}) \xrightarrow[\text{BN}]{\gamma^{[1]},\beta^{[1]}} a_N^{[1]} \dashrightarrow g(z^{[2]}) \xrightarrow[\text{BN}]{\gamma^{[2]},\beta^{[2]}} a_N^{[2]} \dashrightarrow \ldots$$

Batch Normalization

For any layer *l*, given intermediate values of $z^{(1)}....z^{(n)}$ in the layer:

$\mu = (1/n) * \sum z^{(i)}$

$z^{(i)} = z^{(i)} - \mu$ for each $z^{(i)}$

$\sigma^2 = (1/n) * \sum [z^{(i)}]^2$

$z_{norm}^{(i)} = z^{(i)}/(\sigma + \varepsilon)$ //Every normalized $z^{(i)}$ has zero mean and unit variance

$z^{(i)} = \gamma z_{norm}^{(i)} + \beta$

~