:: Neural Network

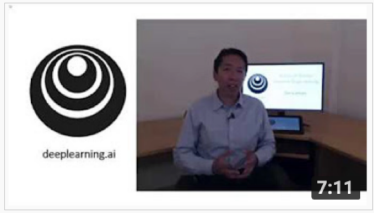
What's derivative & partial derivative?

It's important to understand the basic concept of derivative. It's strongly recommended to watch the following videos if you have already forgotten those concept.

Hold on a second! I have forgotten calculus.

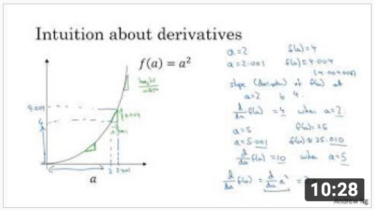
I can't understand the **partial derivative**

$$\frac{\partial E}{\partial w_{ij}}$$



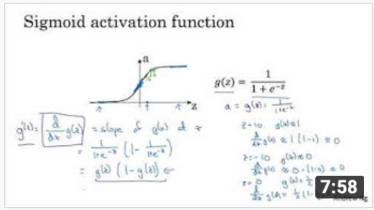
1 Derative

www.youtube.com/watch?v=GzphoJOVEcE



2 More Derivative Examples

www.youtube.com/watch?v=5H7M5Vd3-pk



3 Derivatives of Activation Functions

www.youtube.com/watch?v=-CG_t8e9Bac

4 Partial derivatives, introduction

www.youtube.com/watch?v=AXqhWeUEtQU&t=16s

Picture: Designed by Asierromero / Freepik

:: Neural Network

How do you pick these optimal weights which can minimize the loss function ?

- **Use gradient descent algorithm to find the weights which can bring the error down**

  In the figure below, the loss function is shaped like a bowl. At any point in the training process, the partial derivatives of the loss function with respect to the weights is nothing but the slope of the bowl at that location. One can see that by moving in the direction predicted by the partial derivatives, we can reach the bottom of the bowl and therefore minimize the loss function. This idea of using the partial derivatives of a function to iteratively find its local minimum is called the **gradient descent**[1]

- **The new weights is obtained by moving the direction of the negative gradient along the multidimensional error surface**

Update the weights

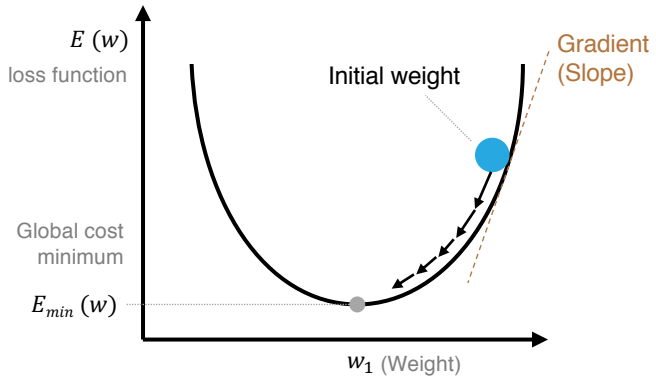
$$w_{ij} = w_{ij} - \eta * \frac{\partial E}{\partial w_{ij}}$$

$$\text{Gradient} = \frac{\text{change in error}}{\text{change in weight}}$$

$$= \frac{\Delta error}{\Delta weight}$$

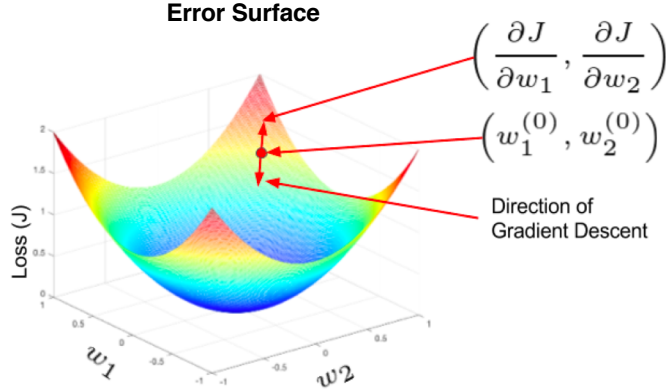
$$= \frac{\partial E}{\partial W}$$

Error Surface



Error surface of a linear neuron with one weight

Error Surface



Error surface of a linear neuron with two input weights

☛ An animation example for gradient descent

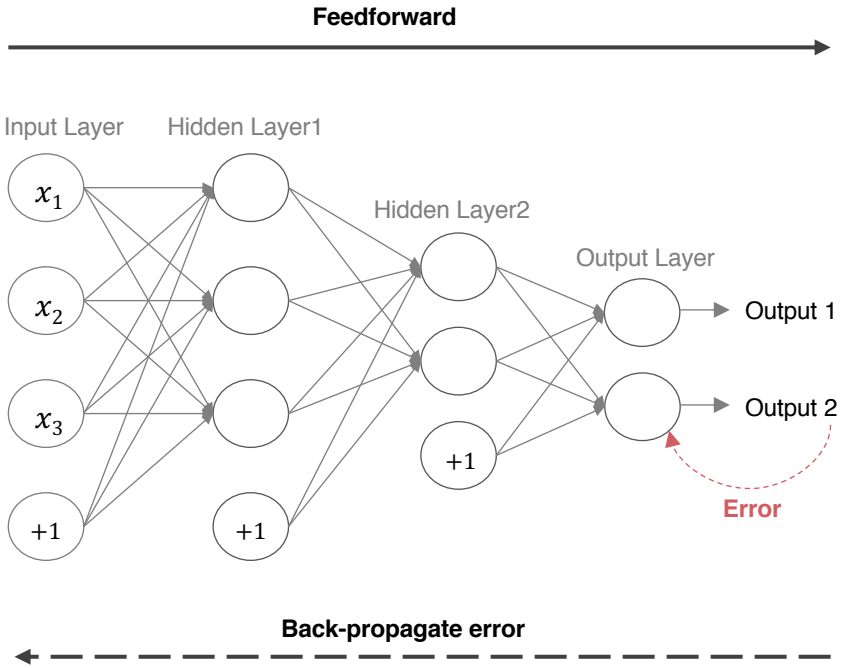
1 Source : https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/

:: Neural Network

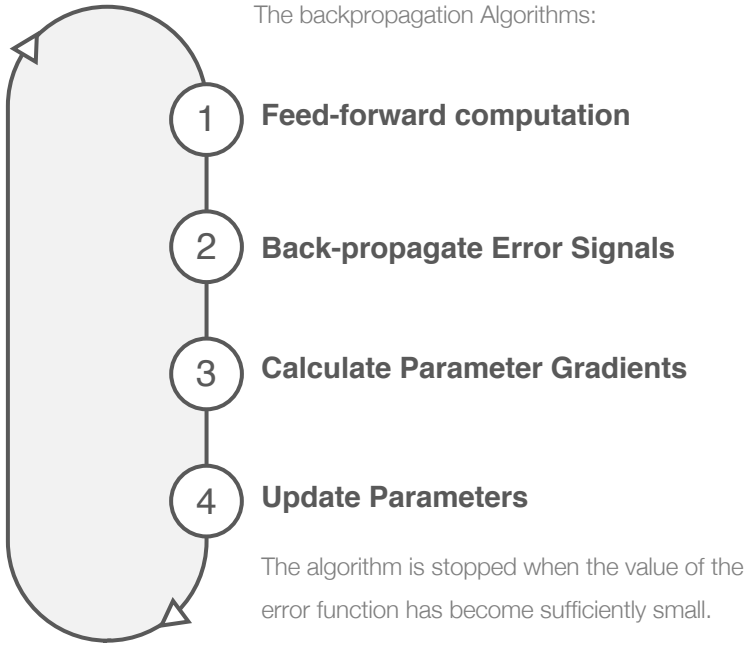
Backpropagation algorithm

- **In Artificial neural networks the weights are updated using a method called Backpropagation.**

  The partial derivatives of the loss function with respect to the weights are used to update the weights. In a sense, the error is backpropagated in the network using derivatives. This is done in an iterative manner and after many iterations, the loss reaches a minimum value, and the derivative of the loss becomes zero.[1]



To tune the weights between the hidden layer and the input layer, we need to know the error at the hidden layer, but we know the error only at the output layer. So we can take the errors at the output layer and proportionally propagate them backwards to the hidden layer[2].



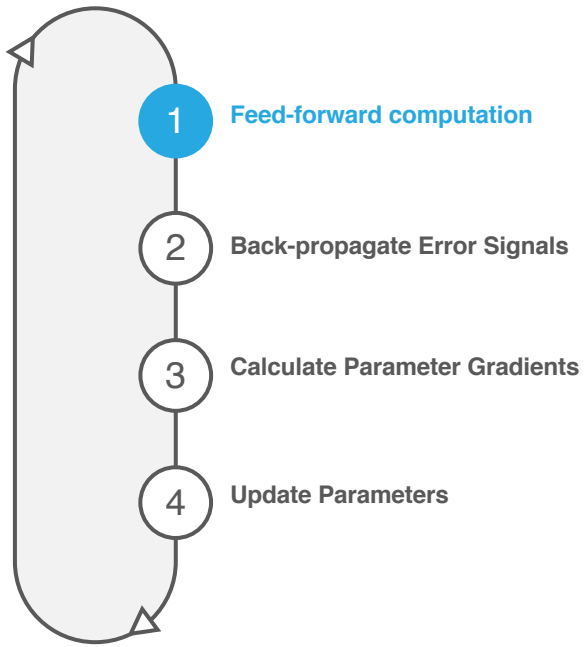
1 Source : https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/
https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-part-2-1218d5dc043

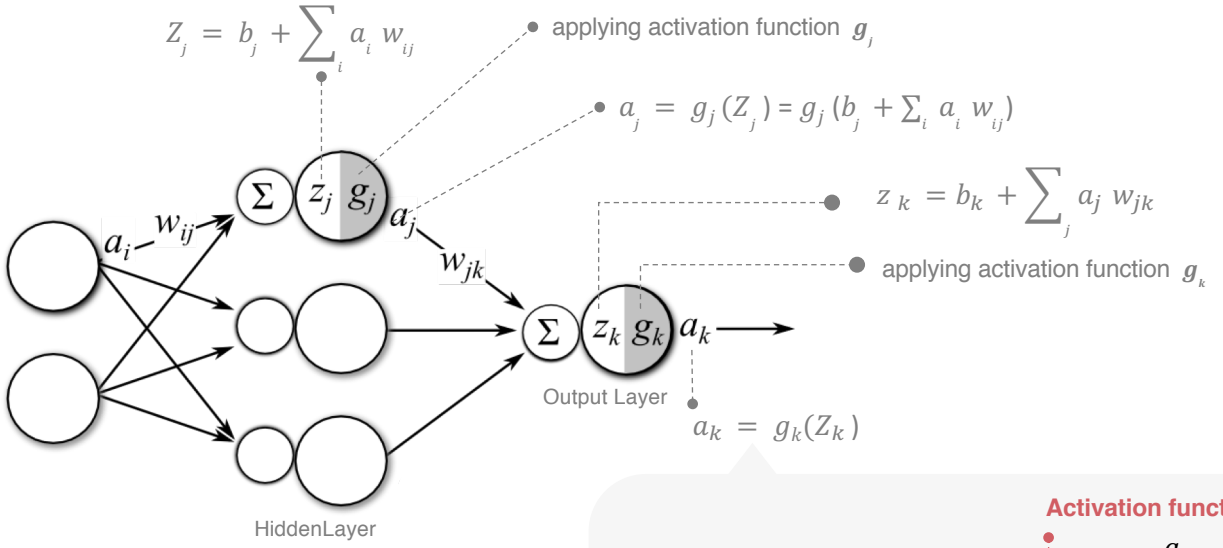
:: Neural Network

Backpropagation algorithm

- **Propagation forward through the network to generate the output value(s)**

  When an input vector is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer. [1].





Activation function for unit in hidden layer

$$\boldsymbol{a}_{out} = \boldsymbol{a}_k = \boldsymbol{g}_k\left(\boldsymbol{b}_k + \sum_j \boldsymbol{g}_j\left(\underbrace{\boldsymbol{b}_j + \sum_i \boldsymbol{a}_i \boldsymbol{w}_{ij}}_{z_j}\right)\boldsymbol{w}_{jk}\right)$$

a_j z_k

Output

Activation function for unit in output layer

w_{ij}: weights connecting node i in layer $(l-1)$ to node j in layer l

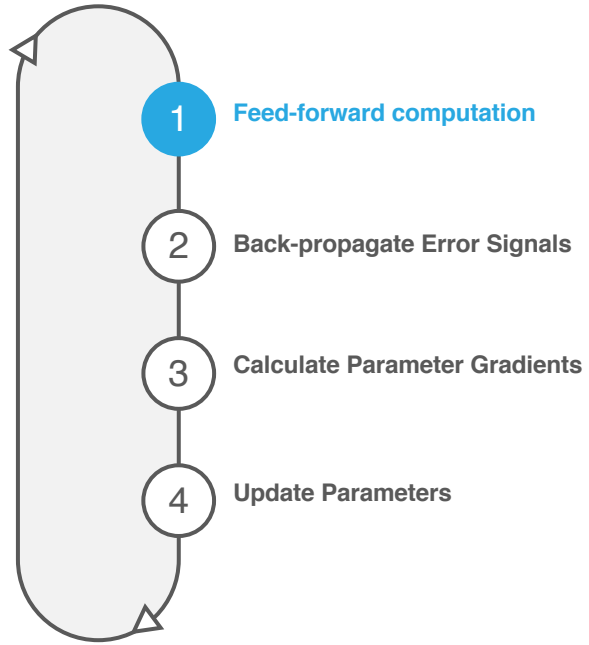
1 source: https://en.wikipedia.org/wiki/Backpropagation
Source of network diagram: https://theclevermachine.wordpress.com/tag/backpropagation/

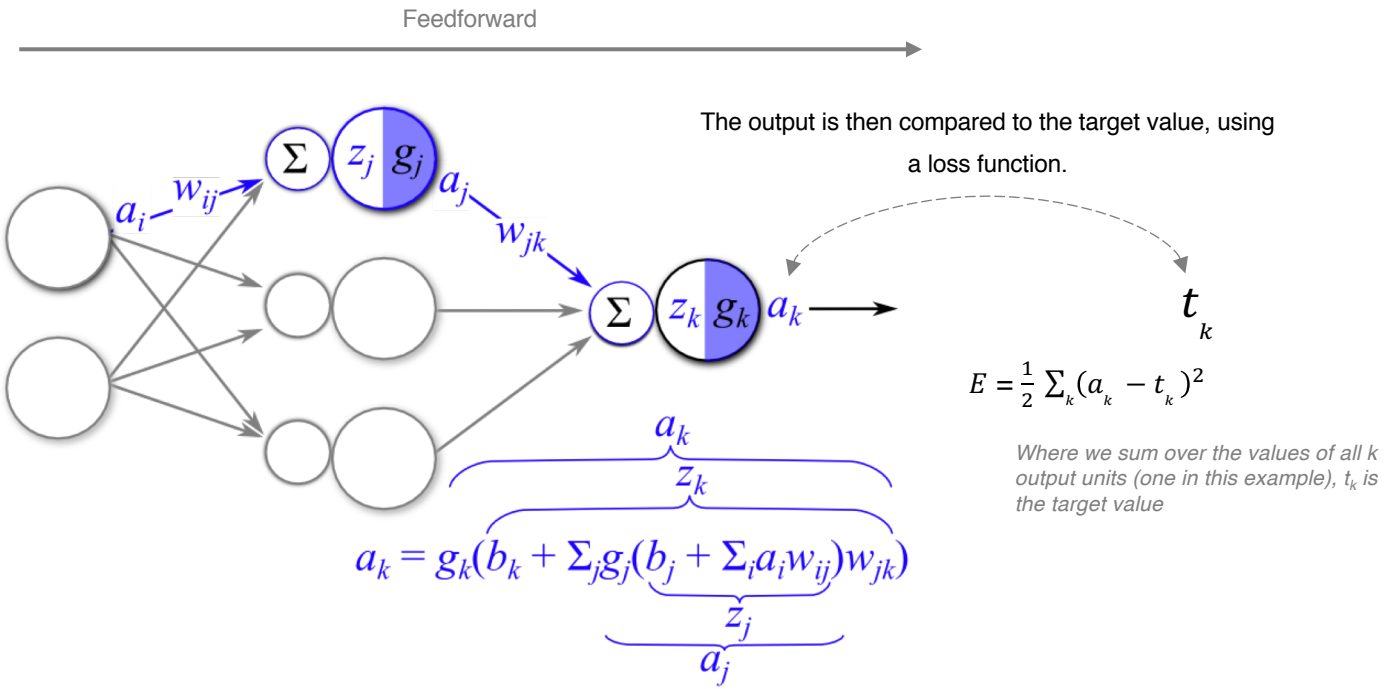
:: Neural Network

Backpropagation algorithm

- **Propagation forward through the network to generate the output value(s)**

  When an input vector is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer. [1].



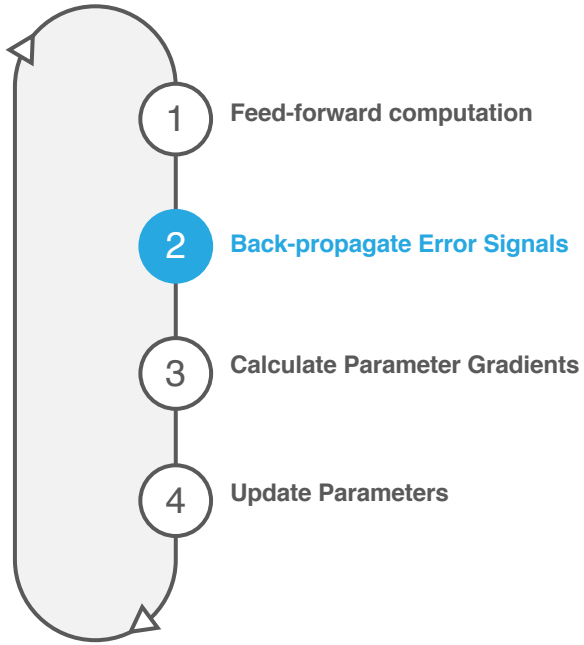


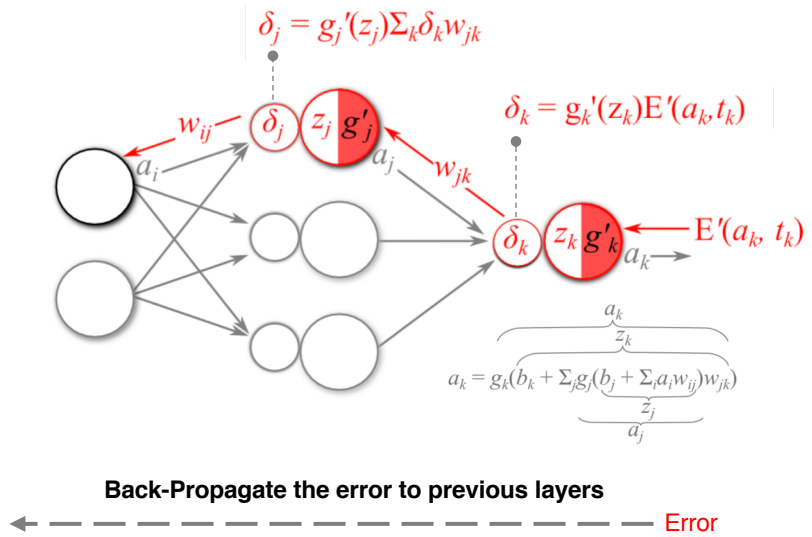
1 source: https://en.wikipedia.org/wiki/Backpropagation
Source of network diagram: https://theclevermachine.wordpress.com/tag/backpropagation/

:: Neural Network

Backpropagation algorithm

- **Calculate output error *E* based on the predictions a_k and the target t_k**
- **Backward propagation of errors**
  - Backpropagation to the output layer
  - Backpropagation to the hidden layer





Back-Propagate the error to previous layers

Error

These error values will be used to calculate the gradient of the loss function.

- **Calculate the "Error term" δ_k for output node**

  Here the δ_k terms can be interpreted as the network output error after being back-propagated through the output activation function, thus creating an error "signal". [1]

$$\delta_k = g_k{}'(z_k)E'(a_k, tk) = g_k{}'(z_k)(a_k - t_k)$$

- **Calculate the "Error term" δ_j for the hidden layer node**

  Project δ_k back through w_{jk}, then through the activation function for the hidden layer via g'_j to give the error signal δ_j. [2]

$$\delta_j = g_j{}'(z_j)\sum_k \delta_k\, w_{jk}$$

Note: The derivative of a function $f(x)$ will be denoted $f'(x)$. $g_k{}'$ is the derivative of output layer activation function

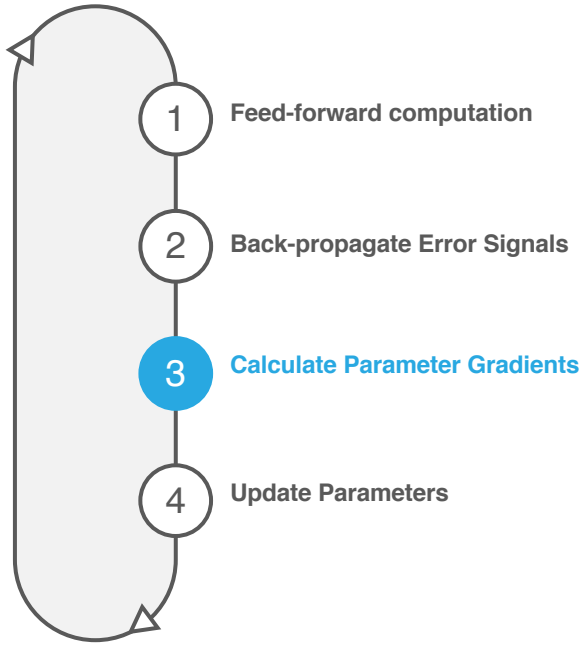
1&2 Source: https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/

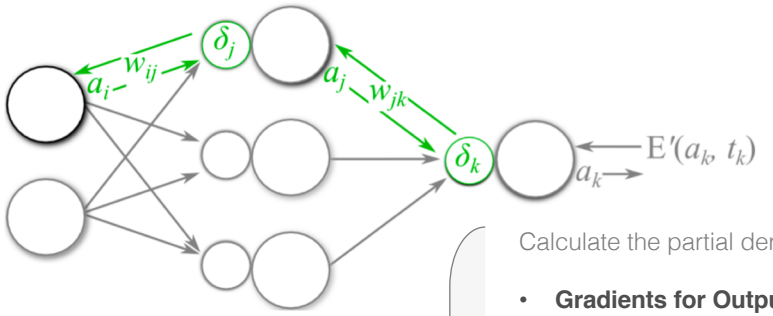
:: Neural Network

Backpropagation algorithm

- **Calculate the gradients $\frac{\partial E}{\partial \theta}$ for the parameters (i.e. Compute the desired partial derivatives)**

  Training a neural network with gradient descent requires the calculation of the gradient of the error function E with respect to the weights and biases (collectively denoted θ) . Step 3 of the backpropagation algorithm is to calculate the gradients of the error function with respect to the model parameters at each layer l using the forward signals a_{l-1}, and the backward error signals δ_l .[1]





$$\frac{\partial E}{\partial w_{l-1,l}} = a_{l-1}\,\delta_l$$

Output on node $l-1$ (previous layer)

Back-propagated error term on node l

Calculate the partial derivative of the error function E with respect to parameters

- **Gradients for Output Layer connection Weights**

$$\frac{\partial E}{\partial w_{jk}} = a_j\,\delta_k \quad \text{where} \quad \delta_k = g_k'(z_k)E'(a_k, tk) = g_k'(z_k)(a_k - tk)$$

- **Gradients for Hidden Layer Weights**

$$\frac{\partial E}{\partial w_{ij}} = a_i\,\delta_j \quad \text{where} \quad \delta_j = g_j'(z_j)\sum_{k_{\epsilon}K}\delta_k\, w_{jk}$$

- **Gradients for Biases** (take the bias b_k as example)

$$\frac{\partial E}{\partial b_k} = \delta_k$$

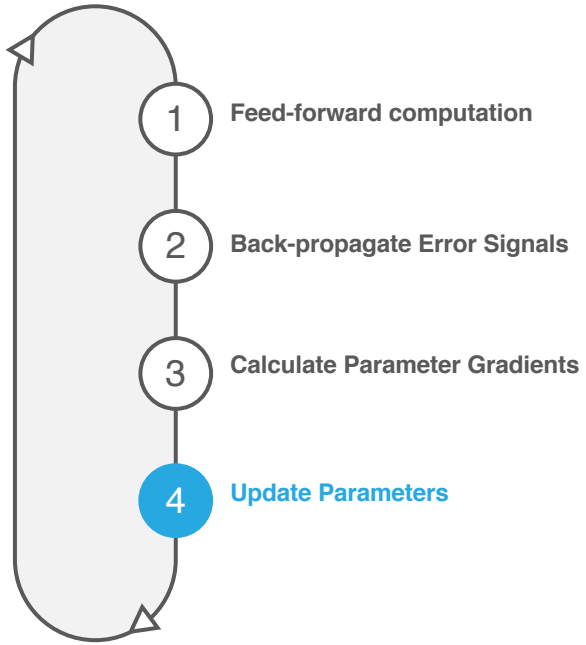
1 source: https://theclevermachine.wordpress.com/tag/backpropagation/

:: Neural Network

Backpropagation algorithm

- **Update parameters (Weights and Bias)**

  Update the model parameters based on the gradients calculated in Step 3. Note that the gradients point in the direction in parameter space that will increase the value of the error function. Thus when updating the model parameters we should choose to go in the opposite direction. How far do we travel in that direction? That is generally determined by a user-defined step size (aka learning rate) parameter, η Thus given the parameter gradients and the step size, the weights and biases for a given layer are updated accordingly.[1]



Update the parameters using the calculated gradients

- **Update the parameter w_{jk}**

$$w_{jk} = w_{jk} - \eta \frac{\partial E}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = a_j \, \delta_k$$

- **Update the parameter w_{ij}**

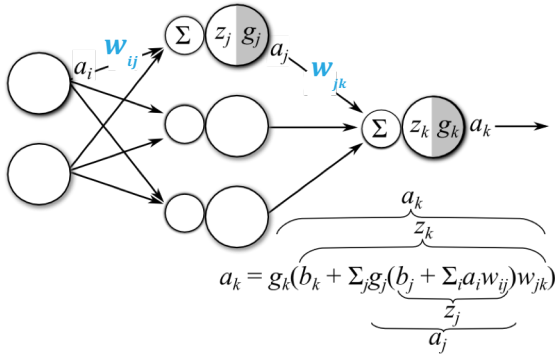
$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = a_i \, \delta_j$$

- **Update the biases** (take b_j as example)

$$b_j = b_j - \eta \frac{\partial E}{\partial b_j}$$

$$\frac{\partial E}{\partial b_j} = \delta_j$$



$$\delta_k = g_k'(z_k) E'(a_k, tk) = g_k'(z_k)(a_k - tk)$$

$$\delta_j = g_j'(z_j) \sum_{k \in K} \delta_k \, w_{jk}$$

★ η *is the learning rate (step size)*

1 source: https://theclevermachine.wordpress.com/tag/backpropagation/

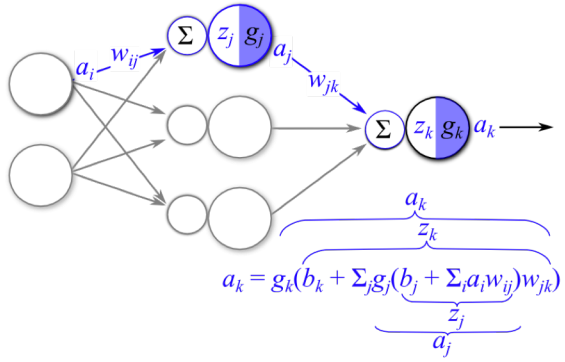
:: Neural Network

Summary of Backpropagation algorithm

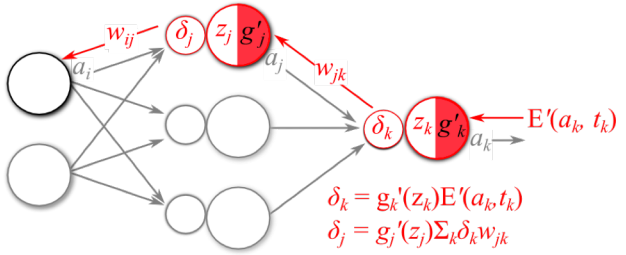
Backpropagation algorithm

1. Initialize the network's weights randomly
2. **Implement forward propagation to get network output**
3. **Compute the error (loss function)**
4. **And then back propagates the error to the previous layers**
5. **These previous layers then adjust their weights and biases accordingly.**
6. Repeat this process (from step 2) to minimize the error until the loss converges

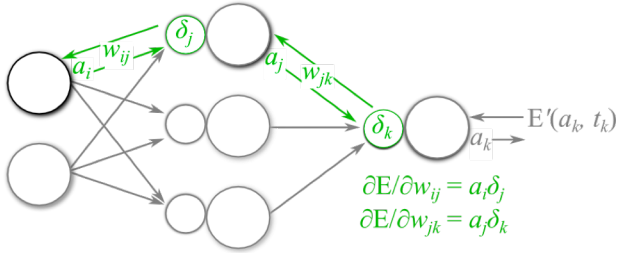
I. Forward-propagate Input Signal



II. Back-propagate Error Signals



III. Calculate Parameter Gradients



IV. Update Parameters

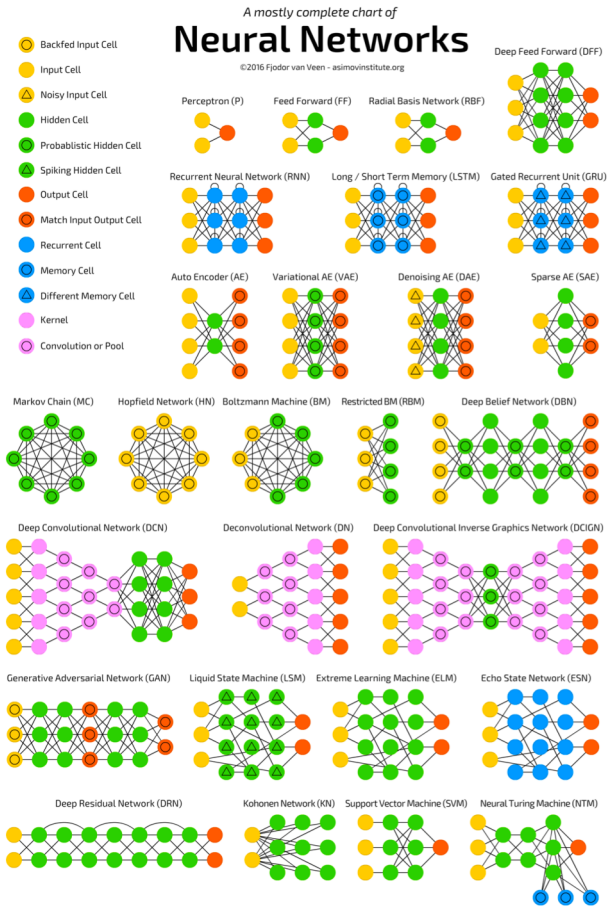
$w_{ij} = w_{ij} - \eta(\partial E/\partial w_{ij})$
$w_{jk} = w_{jk} - \eta(\partial E/\partial w_{jk})$
for learning rate η

Source of diagram: https://theclevermachine.wordpress.com/tag/backpropagation/

:: Neural Network

Popular Neural Network Architectures [1]

- **Perceptron** — Neural Network having two input units and one output units with no hidden layers. These are also known as ‘single layer perceptrons.
- **Radial Basis Function Network** — These networks are similar to the feed forward neural network except radial basis function is used as activation function of these neurons.
- **Multilayer Perceptron** — These networks use more than one hidden layer of neurons, unlike single layer perceptron. These are also known as deep feedforward neural networks.
- **Recurrent Neural Network (RNN)** — Type of neural network in which hidden layer neurons has self-connections. Recurrent neural networks possess memory. At any instance, hidden layer neuron receives activation from the lower layer as well as it previous activation value.
- **Long /Short Term Memory Network (LSTM)** — Type of neural network in which memory cell is incorporated inside hidden layer neurons is called LSTM network.
- **Hopfield Network** — A fully interconnected network of neurons in which each neuron is connected to every other neuron. The network is trained with input pattern by setting a value of neurons to the desired pattern. Then its weights are computed. The weights are not changed. Once trained for one or more patterns, the network will converge to the learned patterns. It is different from other neural networks.
- **Boltzmann Machine Network** — These networks are similar to Hopfield network except some neurons are input, while other are hidden in nature. The weights are initialized randomly and learn through back propagation algorithm.
- **Convolutional Neural Network (CNN)** — A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers[2]

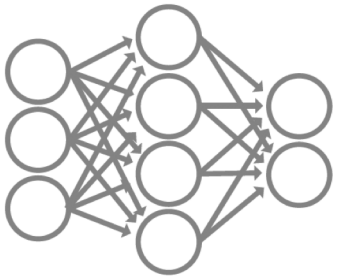


Details can be found on
http://www.asimovinstitute.org/neural-network-zoo/

1 source: https://www.xenonstack.com/blog/overview-of-artificial-neural-networks-and-its-applications
2 source:https://en.wikipedia.org/wiki/Convolutional_neural_network

:: Neural Network

Application of Neural Network



- **Classification**

  A neural network can be trained to classify given pattern or data set into predefined class. It uses feedforward networks.[1]

- **Prediction**

  A neural network can be trained to produce outputs that are expected from given input. Eg: — Stock market prediction.[2]

- **Clustering**

  The Neural network can be used to identify a special feature of the data and classify them into different categories without any prior knowledge of the data. [3]

Application examples

- Character recognition
- Image recognition
- Face recognition
- Stock prediction
- Voice recognition
- Fraud Detection
- Sales Forecast
- Cash Flow Forecasting
- Credit Rating
- …

Source: https://medium.com/@xenonstack/overview-of-artificial-neural-networks-and-its-applications-2525c1addff7

:: Neural Network

Advantage of Neural Network

- **Prediction accuracy is generally high**
  - record-breaking accuracy on a whole range of problems including image and sound recognition, text and time series analysis, etc.
  - Can significantly out-perform other models when the conditions are right (lots of high quality labeled data).

- **Very powerful algorithm which can to learn and model non-linear and complex relationships**
  - Can approximate non-linear function
  - Adapt to unknown situations. Flexible for real world applications. Great for complex/abstract problems like image, voice recognition
  - Can handle large number of features, even thousands of inputs
  - Once trained, the output are given quickly

- **Robust**
  - High tolerance to noisy data

- **Very hot algorithm in recent years**
  - large amount of academic research
  - used extensively in industry
  - lots of libraries / implementations available

Image: Designed by Asierromero / Freepik

:: Neural Network

Disadvantage of Neural Network

- **Black Box model**

  Difficult to interpret what the model has learned.

- **Usually it's hard to train and take long training time**
  - The structure of Deep neural network is complicated
  - Require tuning a number of hyper-parameters such as the number of hidden neurons, layers, and learning rate. Many hyper-parameters have to be determined empirically
  - The training is computationally expensive
  - Very data Hungary. Requires a lot of data to perform well.

- **Prone to overfitting**
  - There are lots of methods for reducing overfitting. Such as applying regularization term, Early Stopping, Dropout, etc

- **Don't perform well on small data sets**
  - There are alternatives that are simpler, faster, easier to train, and provide better performance (svm, decision trees, Bayesian, etc).

Image: Designed by Asierromero / Freepik

:: Neural Network

The common failure cases

There are a number of common ways for backpropagation to go wrong.[1]

- *Backpropagation algorithm can get stuck during gradient descent*
- *The gradients could explode or vanish*
- *ReLU units can be fragile during training and can “die”*
- *The neural network can easily overfit to training data*

Image: Designed by Asier_Relampagoestudio / Freepik

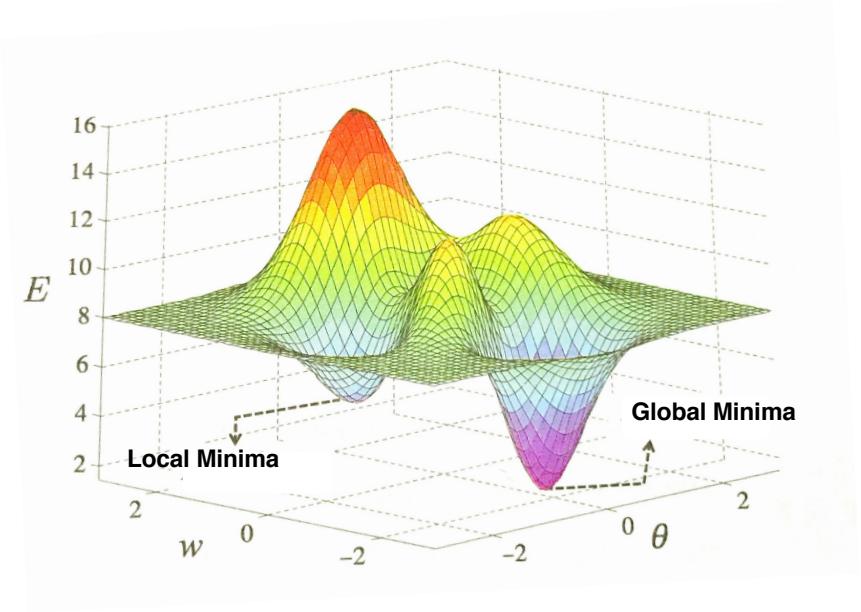
1 source: https://developers.google.com/machine-learning/crash-course/training-neural-networks/best-practices

:: Neural Network

Backpropagation algorithm can get stuck during gradient descent

- **Gradient Descent may have trouble finding the global minimum**

  The error surface can have multiple local minimum. Gradient descent with backpropagation is not guaranteed to find the global minimum of the error function, but only a local minimum; also, it has trouble in crossing plateaus near saddle points in the error function landscape. This issue, is caused by the non-convexity of error functions in neural networks, was long thought to be a major drawback, but Yann LeCun et al. argue that in many practical problems, it is not.[1] "For large-size networks,… struggling to find the global minimum on the training set (as opposed to one of the many good local ones) is not useful in practice and may lead to overfitting [2] "



1 sourcxe: https://en.wikipedia.org/wiki/Backpropagation#cite_note-10

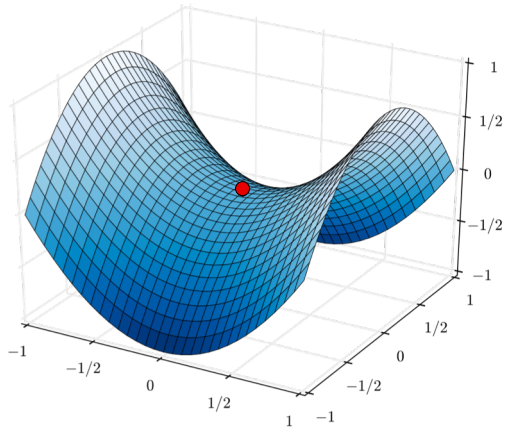
:: Neural Network

Backpropagation algorithm can get stuck during gradient descent – continued

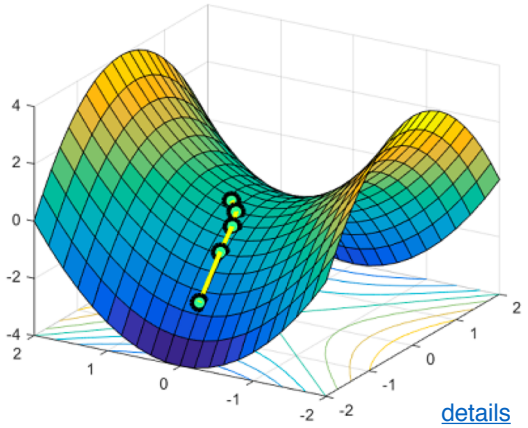
- **The prevalence of saddle points in high dimensional space is main difficulty for minimizing non-convex error functions**

  “ Saddle points, not local minima, provide a fundamental impediment to rapid high dimensional non-convex optimization. … a deeper and more profound difficulty originates from the proliferation of saddle points, not local minima, especially in high dimensional problems of practical interest. Such saddle points are surrounded by high error plateaus that can dramatically slow down learning, and give the illusory impression of the existence of a local minimum.” [2]

Saddle point



How to escape from Saddle point ?



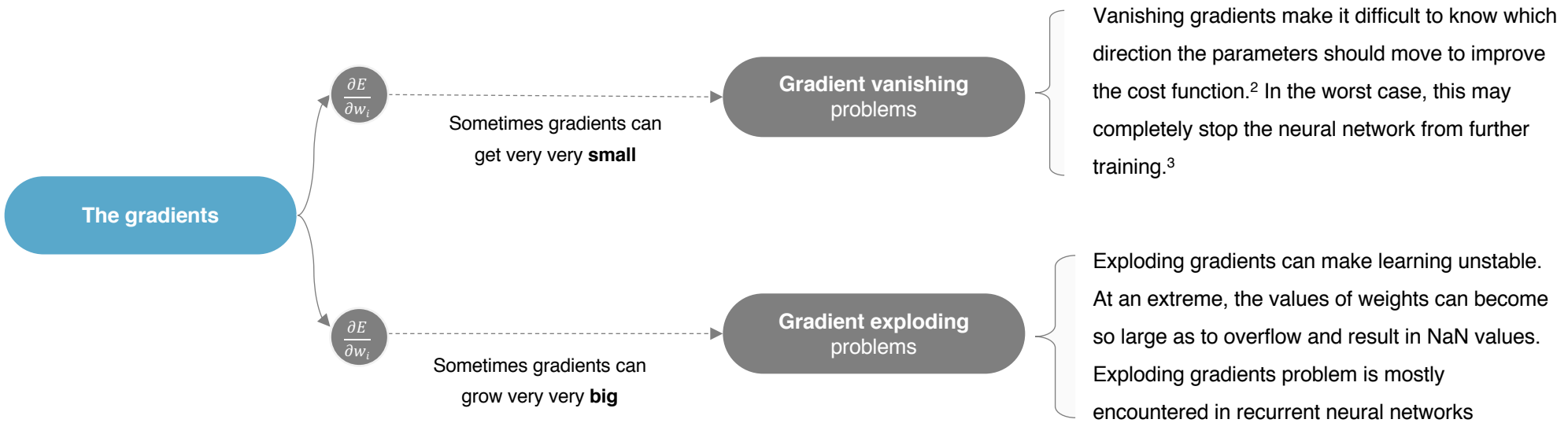
details

2 Source: https://arxiv.org/pdf/1412.0233.pdf
3 Source: https://arxiv.org/pdf/1406.2572.pdf

:: Neural Network

Another problem: the gradients could explode or vanish !!!

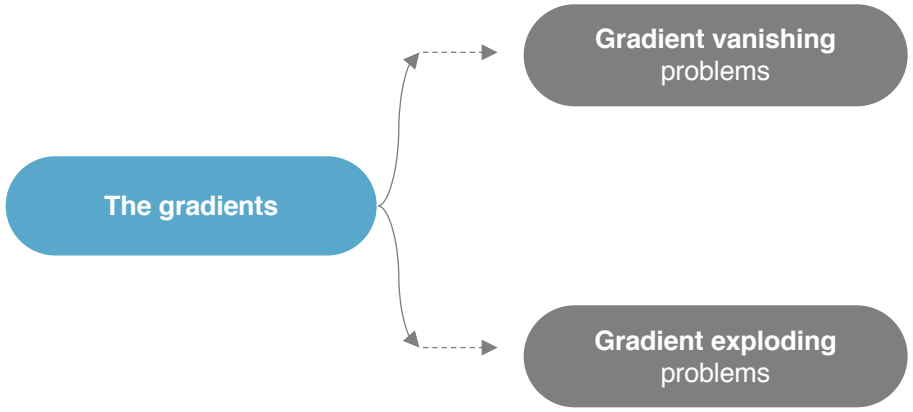
One of the problems of training neural network, especially very deep neural networks, is vanishing/exploding the gradients. In fact, for a long time this problem was a huge barrier to training deep neural networks. [1]



1 Andrew ng, Deep Learning, https://www.youtube.com/watch?v=qhXZsFVxGKo
2 source: Ian Goodfellow, Deep learning
3 source: https://en.wikipedia.org/wiki/Vanishing_gradient_problem

More about Vanishing/Exploding Gradients Problems

The backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step[1]. In deep networks, computing these gradients can involve taking the product of many small/large terms.



The gradients (or derivatives) can sometimes get smaller and smaller

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the vanishing gradients problem [2]

The gradients (or derivatives) can sometimes grow bigger and bigger

When the error is back-propagated through a neural network, it may increase exponentially from layer to layer. In those cases, the gradient with respect to the parameters in lower layers may be exponentially greater than the gradient with respect to parameters in higher layers. This makes the network hard to train if it is sufficiently deep.[3]

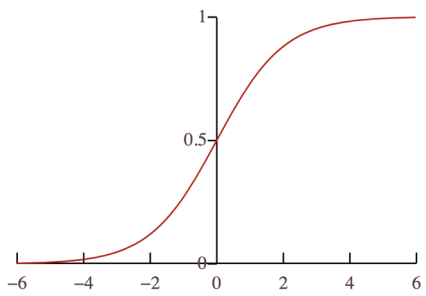
1,2 Aurélien Géron ,Hands-On Machine Learning with Scikit-Learn and TensorFlow
3. The exploding gradient problem demystied - denition, prevalence, impact, origin, tradeos, and solutions

Why Gradients explode or vanish

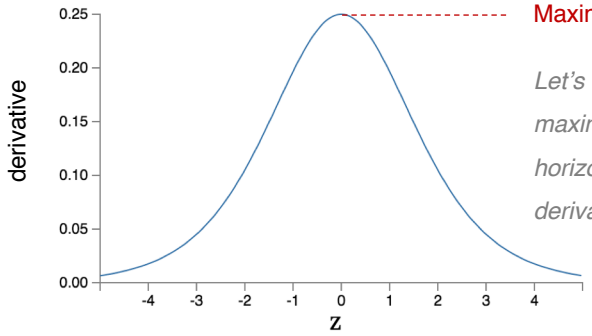
- **Vanishing gradients problems**

  One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function.[1] For example, sigmoid “squeezes” the real number line onto a “small” range of [0, 1]. As a result, there are large regions of the input space which are mapped to an extremely small range. In these regions of the input space, even a large change in the input will produce a small change in the output - **hence the gradient is small**.

Sigmoid function



Derivative of sigmoid function



Maximum value = 0.25

Let’s take a look at the derivative of the sigmoid function. The maximum point of the sigmoid function is 1/4, and the function horizontally asymptotes at 0. In other words, the output of the derivative of the cost function is always between 0 and 1/4.

This becomes much worse when we stack multiple layers of such non-linearities on top of each other. [2] Because backpropagation computes gradients by the chain rule. This has the effect of **multiplying *n* of these small numbers** to compute gradients of the “front” layers in an *n*-layer network, meaning that the gradient (error signal) decreases exponentially with *n* .[3] So there is really nothing left for the front layers. When the gradients vanish toward 0 for the lower layers, these layers train very slowly, or not at all. The ReLU activation function can help prevent vanishing gradients.[4]

1 source: Aurélien Géron ,Hands-On Machine Learning with Scikit-Learn and TensorFlow
2 Source: https://www.quora.com/What-is-the-vanishing-gradient-problem
3source: https://en.wikipedia.org/wiki/Vanishing_gradient_problem#cite_note-17
4 source: https://developers.google.com/machine-learning/crash-course/training-neural-networks/best-practices

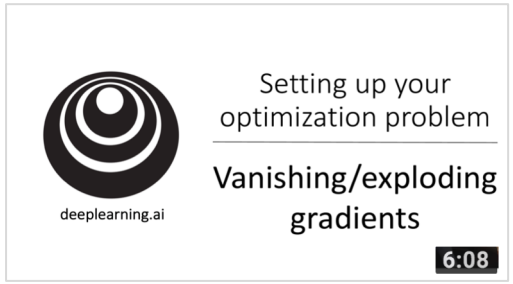
Why Gradients explode or vanish - *continued*

- **Exploding gradients problems**

  In deep networks, computing these gradients can involve taking the product of many small/large terms. The explosion occurs through exponential growth by **repeatedly multiplying gradients** through the network layers that have values larger than 1.0 [2] Therefore, the gradients can grow bigger and bigger, so many layers get insanely large weight updates and the algorithm diverges.[3]

  In other words, If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.

  Batch normalization can help prevent exploding gradients, as can lowering the learning rate.



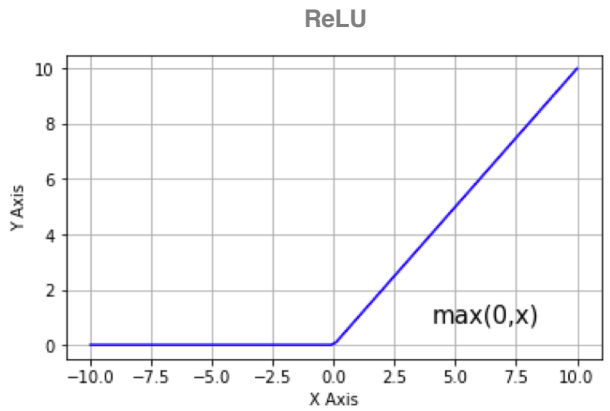
Video: Vanishing/Exploding Gradients by Andrew Ng (6mins)
https://www.youtube.com/watch?v=qhXZsFVxGKo

1 ,3 source: Aurélien Géron ,Hands-On Machine Learning with Scikit-Learn and TensorFlow
2 Source: https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

:: Neural Network

Dead ReLU Units issue

.



$$f(x) = \max(0, x)$$

This means that when the input x < 0 the output is 0 and if x > 0 the output is x.

Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation. With a source of gradients cut off, the input to the ReLU may not ever change enough to bring the weighted sum back above 0.[1]

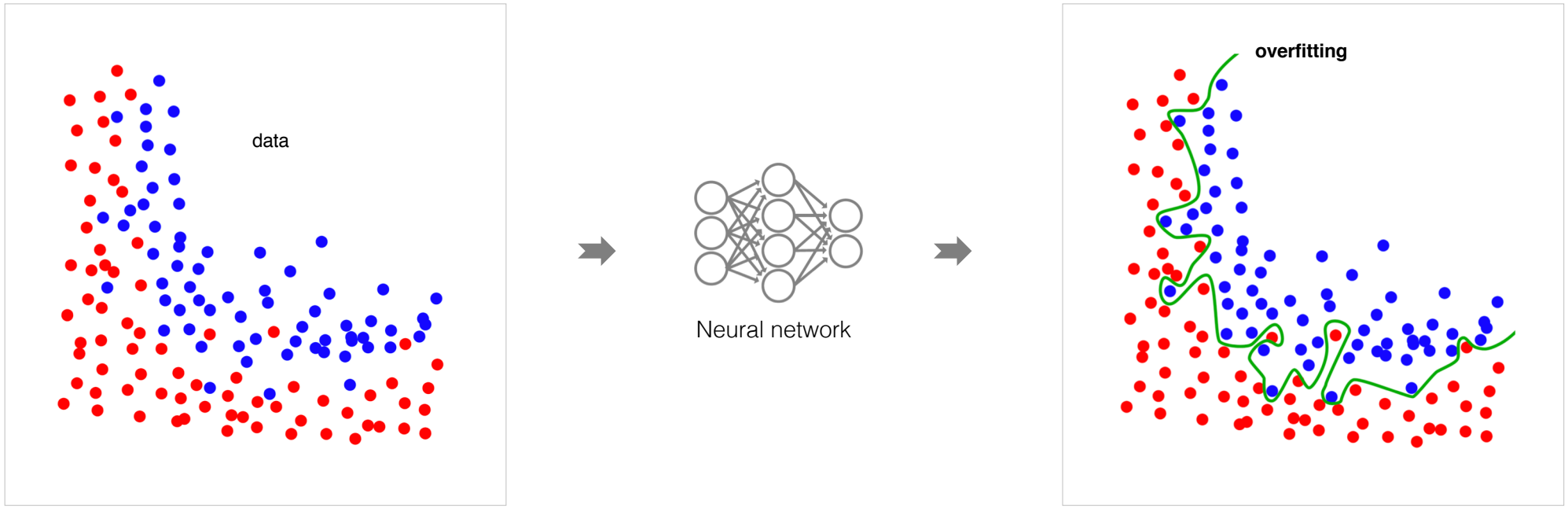
Lowering the learning rate can help keep ReLU units from dying.[2]

1&2 Source: https://developers.google.com/machine-learning/crash-course/training-neural-networks/best-practices

:: Neural Network

Neural networks is prone to overfitting the training set

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set.



Source of image : https://en.wikipedia.org/wiki/Overfitting#/media/File:Overfitting.svg

:: Neural Network

Common approaches to reduce overfitting in neural network

There are a variety of techniques for reducing overfitting issue. For example, Get more Data, Augment data , Use “Bagging”, Use less complicated architecture (e.g. less hidden layers, less nodes per layer), Noise injection, use Regularization, Use Dropout, etc. Typically, a combination of several these methods is used. [1]

Some of popular approaches

- **Early Stopping**

  Start with small weights and stop the learning before it overfits [2]

- **L1 & L2 Regularization**

  Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty) [3]

- **Dropout Regularization**

  randomly dropping neurons from the network during training.

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow
2, 3 Source: Geoffrey Hinton, https://www.youtube.com/watch?v=W0SP8FTmGW0

:: Neural Network

Common approaches to reduce overfitting in neural network

There are a variety of techniques for reducing overfitting issue. For example, Get more Data, Augment data , Use “Bagging”, Use less complicated architecture (e.g. less hidden layers, less nodes per layer), Noise injection, use Regularization, Use Dropout, etc. Typically, a combination of several these methods is used. [1]

Some of popular approaches

- **Early Stopping**

  Start with small weights and stop the learning before it overfits [2]

- **L1 & L2 Regularization**

  Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty) [3]

- **Dropout Regularization**

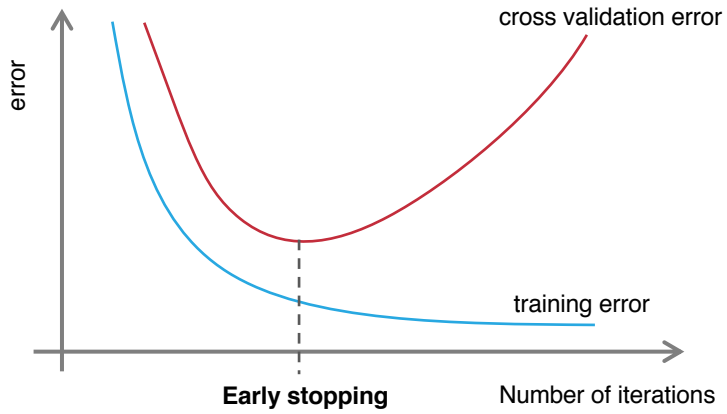
  randomly dropping neurons from the network during training.

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow
2, 3 Source: Geoffrey Hinton, https://www.youtube.com/watch?v=W0SP8FTmGW0

:: Neural Network

Reduce overfitting by Early Stopping

To avoid overfitting the training set, a great solution is early stopping : just interrupt training when its performance on the validation set starts dropping.



- If we have lots of data and a big model, it's very expensive to keep re-training it with different sized penalties on the weights.
- It's much cheaper to **start with very small weights and let them grow until the performance on the validation set starts getting worse**.
  - But it ca be hard to decide when performance is getting worse.
- The capacity of the model is limited because the weights have not had time to grow big

One way to implement Eealy Stopping with TensorFlow is to evaluate the model on a validation set at regular intervals (e.g., every 50 steps), and save a "winner" snapshot if it outperforms previous "winner" snapshots. Count the number of steps since the last "winner" snapshot was saved, and interrupt training when this number reaches some limit (e.g., 2,000 steps). Then restore the last "winner" snapshot.

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow

:: Neural Network

Common approaches to reduce overfitting in neural network

There are a variety of techniques for reducing overfitting issue. For example, Get more Data, Augment data , Use “Bagging”, Use less complicated architecture (e.g. less hidden layers, less nodes per layer), Noise injection, use Regularization, Use Dropout, etc. Typically, a combination of several these methods is used. [1]

Some of popular approaches

- **Early Stopping**

  Start with small weights and stop the learning before it overfits [2]

- **L1 & L2 Regularization**

  Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty) [3]

- **Dropout Regularization**

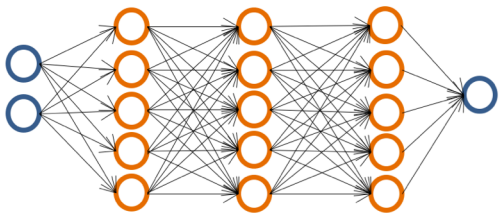
  randomly dropping neurons from the network during training.

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow
2, 3 Source: Geoffrey Hinton, https://www.youtube.com/watch?v=W0SP8FTmGW0

:: Neural Network

Reduce overfitting by applying L2 Regularization (Weight Decay)

You can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights (but typically not its biases) [1]. L2 regularization is perhaps the most common form of regularization



simply add the L2 penalty to your cost function

L2 regularization is the most common type of regularization. It can be implemented by penalizing the **squared** magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. [2]

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \underbrace{\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}_{\text{Regularization term}}$$

A regularization term is added to a loss function. It is typically chosen to impose a penalty on the complexity of predicting function.

Video: Why Regularization Reduces Overfitting by Andrew Ng? (7 mins)
https://www.youtube.com/watch?v=NyG-7nRpsW8

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow
2 Source: http://cs231n.github.io/neural-networks-2/#reg

:: Neural Network

Common approaches to reduce overfitting in neural network

There are a variety of techniques for reducing overfitting issue. For example, Get more Data, Augment data , Use “Bagging”, Use less complicated architecture (e.g. less hidden layers, less nodes per layer), Noise injection, use Regularization, Use Dropout, etc. Typically, a combination of several these methods is used. [1]

Some of popular approaches

- **Early Stopping**

  Start with small weights and stop the learning before it overfits [2]

- **L1 & L2 Regularization**

  Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty) [3]

- **Dropout Regularization**

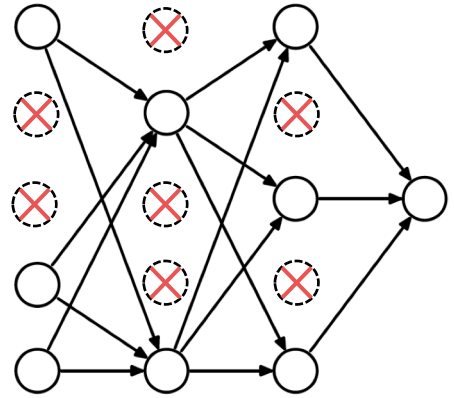
  randomly dropping neurons from the network during training.

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow
2, 3 Source: Geoffrey Hinton, https://www.youtube.com/watch?v=W0SP8FTmGW0

:: Neural Network

Reduce overfitting by applying Dropout Regularization

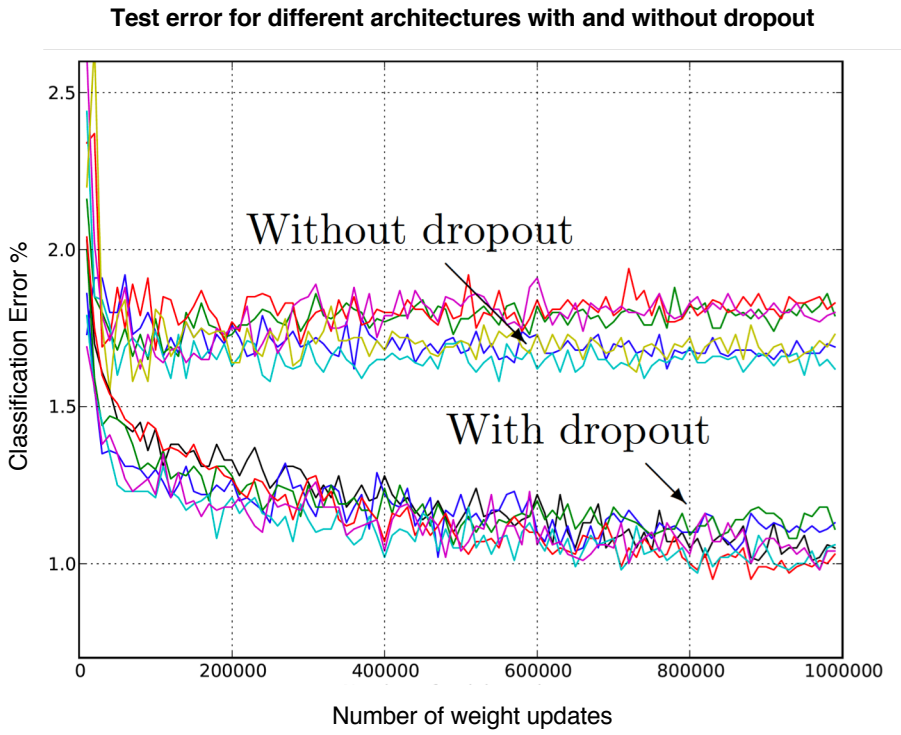
Dropout training simulates the notion of zero weights by eliminating some of the hidden units while training. In dropout training, each time an example is read, some hidden units are removed with probability , and the network is trained and propagates error as if those weights were not there. [1]



Each time we present a training example, we randomly omit each unit with probability [2] (e.g. 0.5)

Applying dropout to a neural network amounts to **sampling a "thinned" network** from it [3]

During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks [4]



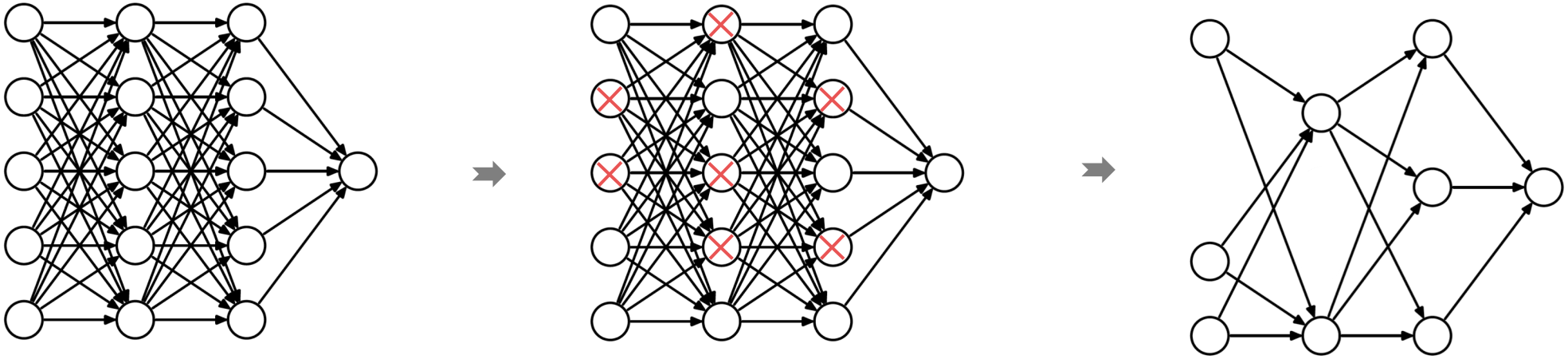
The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

1 Source: Dan Roth, CS 446 Machine Learning Fall 2016
2 Source: Geoffrey Hinton ,https://www.youtube.com/watch?v=kAwF--GJ-ek&t=166s
3 Sourc & image : Source: Nitish Srivastava,et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting
4 Source: http://cs231n.github.io/neural-networks-2/

:: Neural Network

Understand how Dropout works

Standard backpropagation learning builds up brittle co-adaptations that work for the training data but do not generalize to unseen data. Random dropout breaks up these co-adaptations making the presence of any particular hidden unit unreliable.[1]



Standard Neural Net

If a hidden unit know which other units are present, it can co-adapt to them on the training data. But complex co-adaptions are likely to go wrong on new test data.

Apply dropout

At each training stage, individual nodes are either dropped out of the net with probability $1-p$ or kept with probability p

The choice of which units to drop is random.[2]

After applying dropout

A reduced network which is smaller is left.

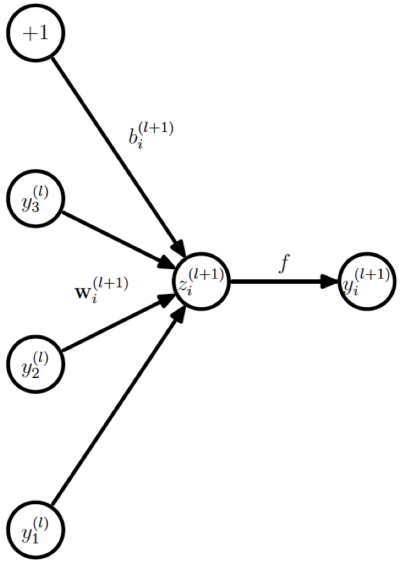
By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections.[3]

1, 2,3 Source: Nitish Srivastava,et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting
Source of image: Srivastava et al ,Dropout: A Simple Way to Prevent Neural Networks from Overtting

:: Neural Network

Understand how Dropout works (continued)

Consider a neural network with L hidden layers. Let $z^{(l)}$ denote the vector of inputs into layer l, $y^{(l)}$ denote the vector of outputs from layer l



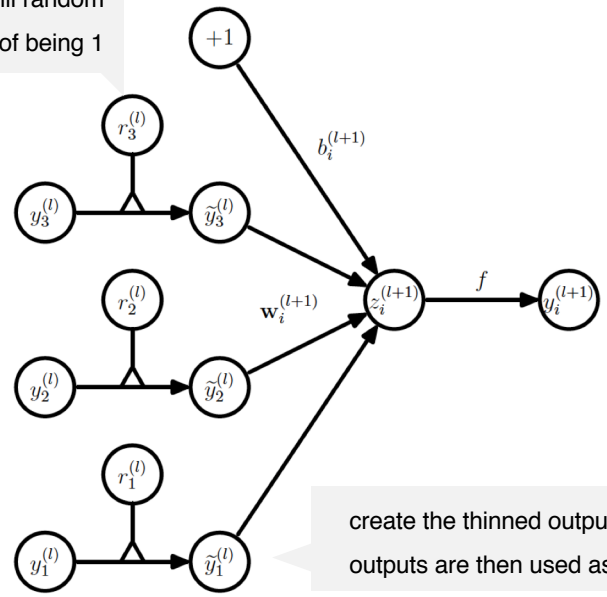
Standard network

The feed-forward operation of a standard neural network can be described as (for $l \in \{0, \ldots, l-1\}$ any hidden unit i)

$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$

where f is any activation function

$r^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability r of being 1



create the thinned outputs $\hat{y}^{(l)}$. The thinned outputs are then used as input to the next layer.

Dropout network

With dropout, the feed-forward operation becomes

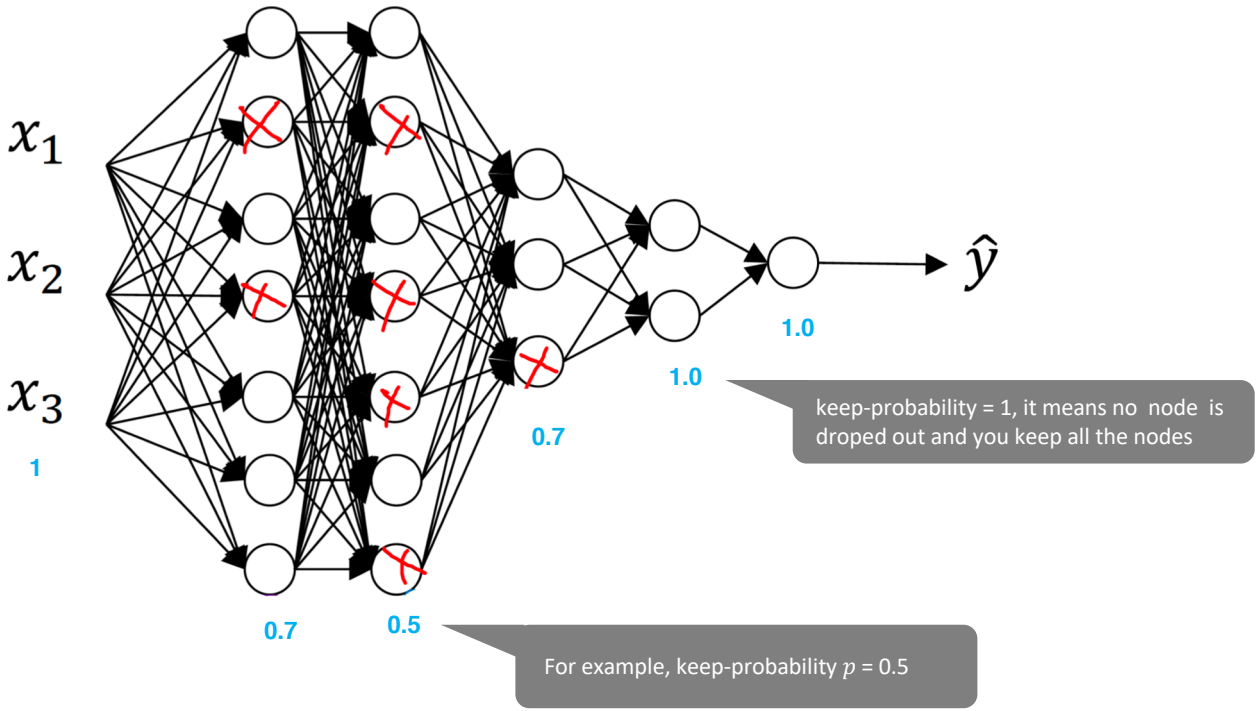
$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$

Source: Nitish Srivastava,et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting

:: Neural Network

Different layers could have different probabilities for keeping units/nodes in network

Dropout has a tunable hyperparameter p (the probability of retaining a unit in the network). These could be different "keep_probability" p for different layers. If you're more worried about some layers overfitting than others, you can set a lower keep_prob for some layers than others.[1]

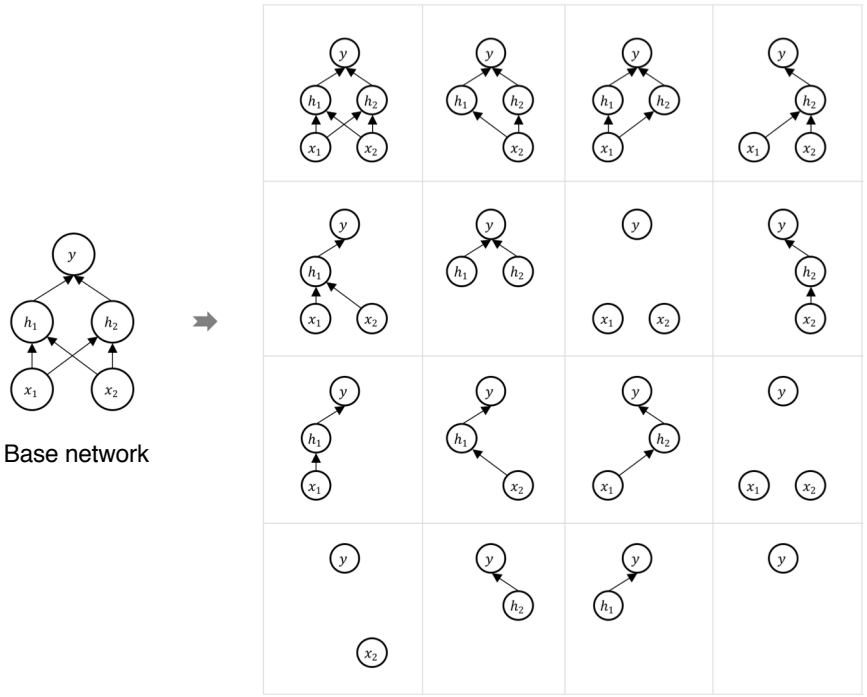


1 & image Source: Andrew Ng, https://www.youtube.com/watch?v=ARq74QuavAo

:: Neural Network

Dropout can be viewed as a form of model averaging

The central idea of dropout is to take a large model that overfits easily and repeatedly sample and train smaller sub-models from it. It prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently.[1]



Base network

Ensemble of subnetworks

There are 16 possible combinations, of which only 9 are viable. Non-viable are (6, 7, 12, 13, 14, 15, 16).

- **Model combination nearly always improves the performance of machine learning methods**
  Another way to view the dropout procedure is as a very efficient way of performing model averaging with neural networks. The standard way to do this is to train many separate networks and then to apply each of these networks to the test data, but this is computationally expensive during both training and testing.

  Random dropout makes it possible to train a huge number of different networks in a reasonable time. There is almost certainly a different network for each presentation of each training case but all of these networks share the same weights for the hidden units that are present.[2]

- **The sharing of the weights means that every model is very strongly regularized**
  Dropout can be seen as an extreme form of bagging in which each model is trained on a single case and each parameter of the model is very strongly regularized by sharing it with the corresponding parameter in all the other models. [3] This is a much better regularizer thanL2 or L1 penalties that pull the weights towards zero.

- **Dropout also has drawback**
  One of the drawbacks of dropout is that it increases training time. A dropout network typically takes 2-3 times longer to train than a standard neural network of the same architecture [4]

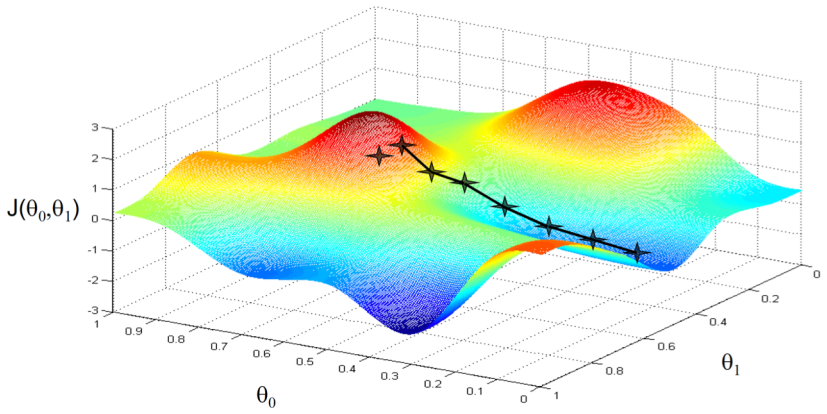
1,2,3,4 Source: Nitish Srivastava,et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting
source of image: http://www.cedar.buffalo.edu/~srihari/CSE676/7.12%20Dropout.pdf

Gradient Descent

:: Gradient Descent

Recap: Gradient descent is an iterative algorithm to find a minimum of a cost function

Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.



The goal of gradient descent is to start on a random point on this error surface , and find out the best parameters (θ_0 , θ_1) which yields the minimum value of the cost function. In other words, the best parameters (θ_0 , θ_1) corresponds to the the lowest point on the error surface.

Gradient Descent algorithm

1. Initialize the parameters with some random values.
2. Keep changing these parameters iteratively in such a way it minimizes the cost function $J(\theta)$

repeat until convergence {

// Simultaneously update θ_0, θ_1

$$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

}

Fig. Sourcce: Adnrew Ng

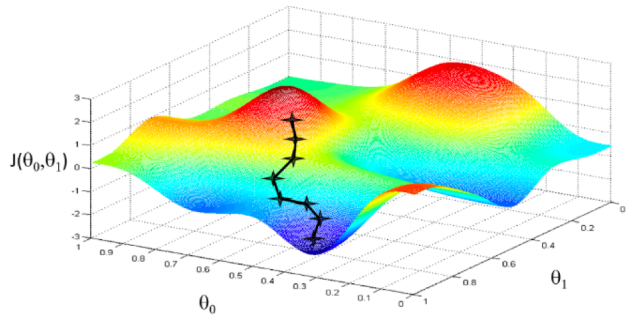
:: Gradient Descent

Gradient Descent Variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.[1]

Gradient Descent

- **Batch gradient descent**
  use all m examples in each iteration
- **Stochastic gradient descent**
  use 1 example in each iteration
- **Mini-batch gradient descent**
  use b examples in each iteration.
  (b =mini-batch size, e.g. b =10)



Gradient Descent

1 Source: http://ruder.io/optimizing-gradient-descent/

:: Gradient Descent

Gradient Descent Variants : Batch Gradient Descent

The traditional Batch Gradient Descent computes the gradient of the vw.r.t. to the parameters θ for the **entire training dataset**.[1]

use all m examples in each iteration

Batch Gradient Descent uses the whole batch of training data at every step.

$$\theta := \theta - \eta \cdot \nabla_\theta J(\theta)$$

Gradient of the cost function

Batch Gradient Descent is NOT suitable for huge datasets

As we need to calculate the gradients for **the whole dataset** to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent also doesn't allow us to update our model *online*, i.e. with new examples on-the-fly.[2]

How big or small of an update to do is determined by the learning rate η . Batch gradient descent is guaranteed

- to converge to the global minimum for convex error surfaces
- to a local minimum for non-convex surfaces.

1,2 Source: http://ruder.io/optimizing-gradient-descent/

:: Gradient Descent

Gradient Descent Variants : Stochastic Gradient Descent

Instead of going through all examples, Stochastic Gradient Descent (SGD) performs the parameters update on **each example** $(x^{(i)}; y^{(i)})$. [1]

use 1 example in each iteration
It uses only one training example in every iteration to compute the gradient of cost function.

$$\theta := \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Gradient of the cost function

Stochastic Gradient Descent is much faster and suitable for huge datasets

Stochastic Gradient Descent just picks a random instance in the training set at every step and computes the gradient is based only on that single instance.[2] It is therefore usually much faster and can also be used to learn online.

The randomness is good to escape from local optima

Now due to these frequent updates ,parameters updates have high variance and causes the Loss function to fluctuate to different intensities. This is actually a good thing because it helps us discover new and possibly better local minima , whereas Standard Gradient Descent will only converge to the minimum of the basin.[3]

Stochastic Gradient Descent can never settle at the minimum

But the problem with SGD is that due to the frequent updates and fluctuations it ultimately complicates the convergence to the exact minimum and will keep overshooting due to the frequent fluctuations [4]. It means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate.[5]

1,Source: https://medium.com/@ImadPhd/gradient-descent-algorithm-and-its-variants-10f652806a3 /
2,5 Source: : Hands-On Machine Learning with Scikit-Learn & TensorFlow
3,4 Source: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

:: Gradient Descent

Gradient Descent Variants : Mini-batch Gradient Descent

Mini-batch gradient descent performs an update for **every mini-batch** of n training examples:

use b examples in each iteration. (b =mini-batch size)
Common mini-batch sizes range between 50 and 256, but can vary for different applications.

$$\theta := \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:\, i+n)}; y^{(i:\, i+n)})$$

Gradient of the cost function

Mini-batch gradient descent finally takes the best of both Batch GD and Stochastic GD

At each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini- Gradient Descent computes the gradients on small random sets of instances called *mini-batches*. Mini-batch gradient descent is a trade-off between stochastic gradient descent and batch gradient descent.[1]

- It reduces the variance of the parameter updates, which can lead to more stable convergence (than Stochastic GD)[2]
- It can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.[3]

- Mini-batch gradient descent is faster than Batch Gradient Descent because it goes through less examples.
- If you choose reasonable value of mini-batch size and if you use a good vectorized implementation, sometimes it can be faster than Stochastic gradient descent.[4]

1 Source: Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow
2, 3 Source: http://ruder.io/optimizing-gradient-descent/
4 Source: https://www.coursera.org/learn/machine-learning/lecture/9zJUs/mini-batch-gradient-descent

:: Gradient Descent

Comparison between Batch gradient descent and **Stochastic gradient descendent**

Let's take a closer look at the difference

Batch gradient descent

Cost function: $J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

Update the parameters:

Repeat {

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

(for every $j = 0, \ldots, n$)

}

// If the dataset size m=10,000,000, the cost for computing the derivatives will be huge. Batch gradient descent can be very slow.

Stochastic gradient descent

Cost function: $J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$

Where $cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

1. Randomly shuffle (reorder) training examples

2. Update the parameters:

Repeat {

for $i := 1, \ldots, m$ {

$$\theta_j := \theta_j - \alpha\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

(for every $j = 0, \ldots, n$)

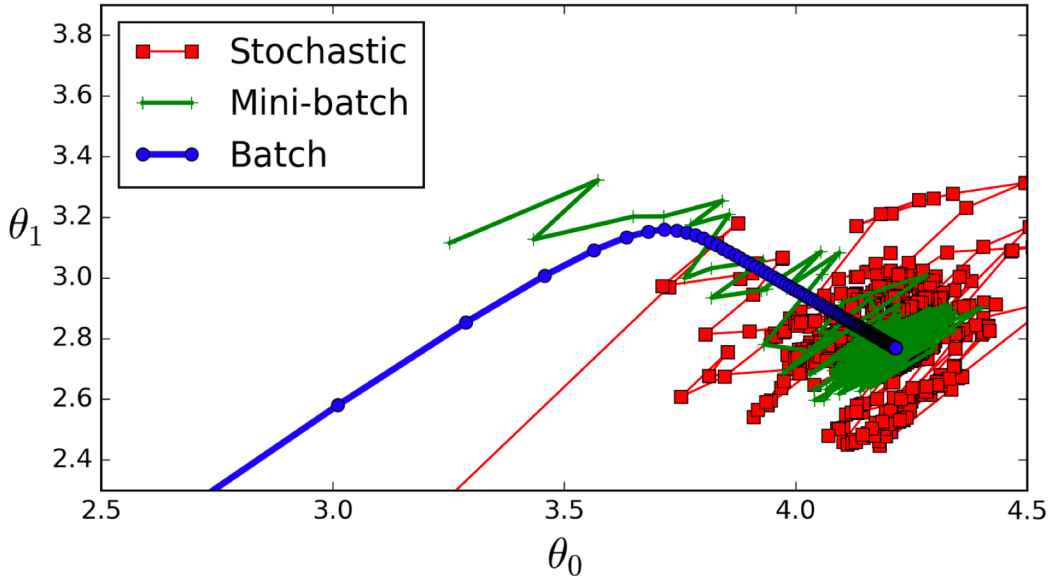
}

}

:: Gradient Descent

Here's a picture comparing the 3 Gradient Descent variants getting to the local minima

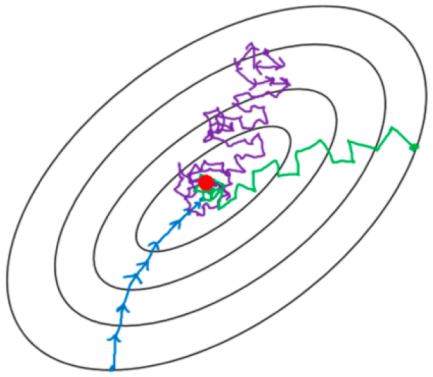
The picture shows the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around. However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.[1]



1 & image : Hands-On Machine Learning with Scikit-Learn & TensorFlow

:: Gradient Descent

Wrap-up: Gradient Descent Variants



Batch gradient descent

use all m examples in each iteration

Stochastic gradient descent (SGD)

use 1 example in each iteration

Mini-batch gradient descent

use b examples in each iteration.

(b =mini-batch size, e.g. b =10)

Stochastic gradient descent (SGD)

It's direction towards the minimum is very noisy compared to mini-batch

Due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. So once the algorithm stops, the final parameter values are good, but not optimal.

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily[2].

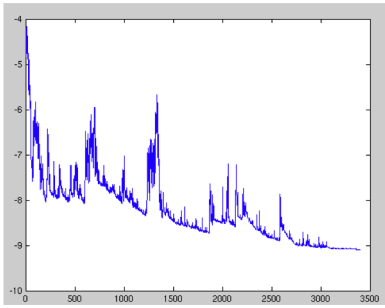


Image2

Source2 & Image2:http://ruder.io/optimizing-gradient-descent/

:: Gradient Descent

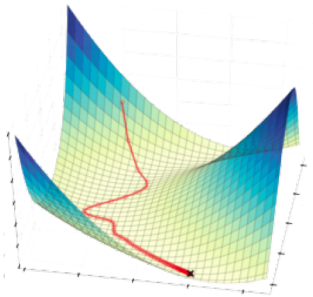
Challenges faced while using Gradient Descent and its variants

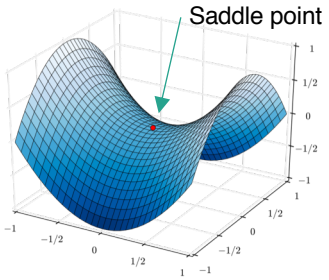
- **Choosing a proper learning rate can be difficult**

  A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.[1]

  Learning rate schedules try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics [2]

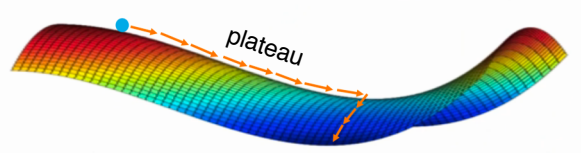
  Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.[3]





- **Gradient Descent might get trapped in local minima or saddle points**

  Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Actually, the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.[4]



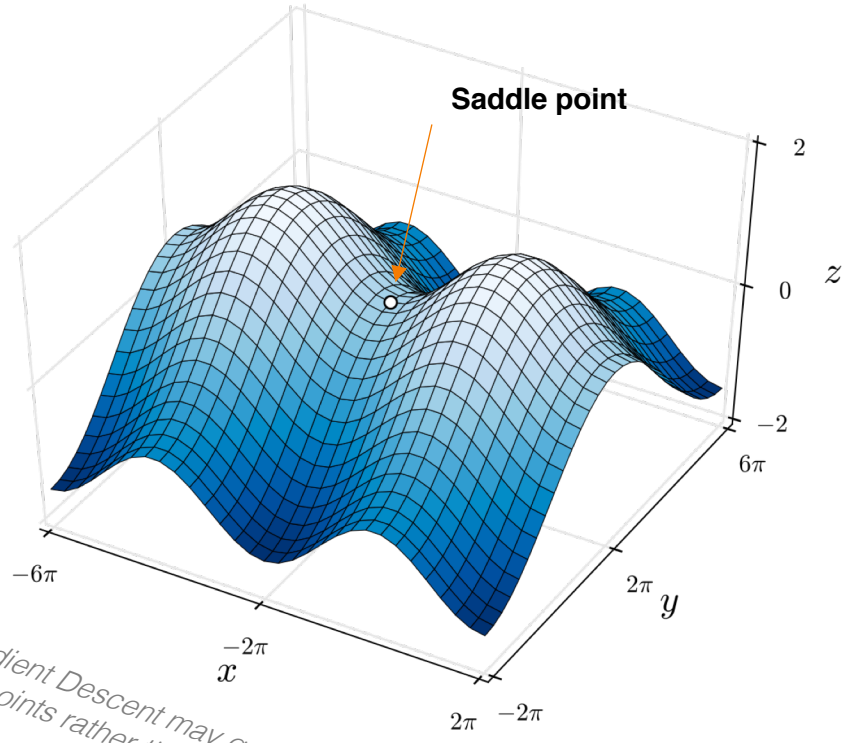
Plateau can make learning very slow

1,2,3 ,4 Source: http://ruder.io/optimizing-gradient-descent/
Image1 Source: https://en.wikipedia.org/wiki/Saddle_surface
Image2 Source: Andrew Ng
Image 3: http://dsdeepdive.blogspot.com/2016/03/optimizations-of-gradient-descent.html

:: Gradient Descent

Saddle Point

- A point where one dimension slopes up while another slopes down, usually surrounded by a plateau of about equal error
- Regardless of the direction GD goes, it is difficult to escape because the surrounds gradients are usually around zero



Gradient Descent may get stuck in saddle points rather than local minima

:: Gradient Descent

Optimizing the Gradient descent

Now we will outline some algorithms that are widely used by the deep learning community to deal with the aforementioned challenges.

- **Momentum**
- **Nesterov accelerated gradient (NAG)**

momentum based

- **Adagrad**
- **Adadelta**
- **RMSprop**
- **Adam**

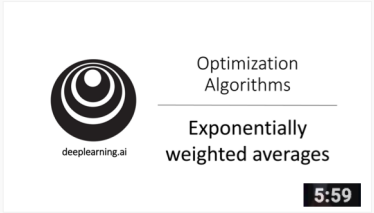
adaptive learning rate methods

:: Gradient Descent

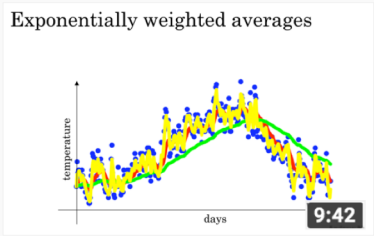
Prerequisite knowledge: Exponentially Weighted Moving Averages

In order to understand the optimization algorithms like Adam, RMSProp, you should understand what Exponentially Weighted Averages means. It's recommended to watch 3 videos by Andrew Ng.

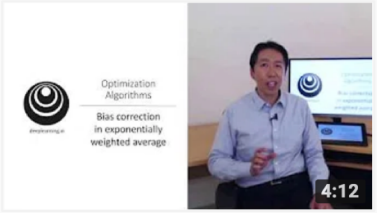
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$



Exponentially Weighted Averages (6 mins)
https://www.youtube.com/watch?v=lAq96T8FkTw



Understanding Exponentially Weighted Averages (9 mins)
https://www.youtube.com/watch?v=NxTFlzBjS-4



Bias Correction in Exponentially Weighted Average (4 mins)
https://www.youtube.com/watch?v=lWzo8CajF5s

:: Gradient Descent

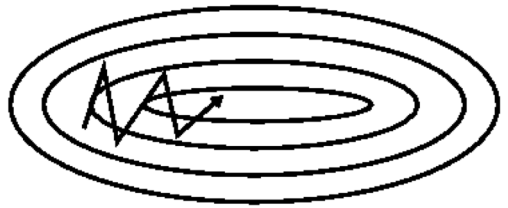
Momentum

Gradient Descent with Momentum is a method that helps accelerate Stochastic Gradient Descent (SGD) in the relevant direction and dampens oscillations [1].

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way [2]

SGD without momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another , which are common around local optima. In these scenarios, SGD **oscillates** across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.[3]



SGD with momentum

Momentum helps SGD dampen oscillations and speeding up the iterations, leading to faster convergence.

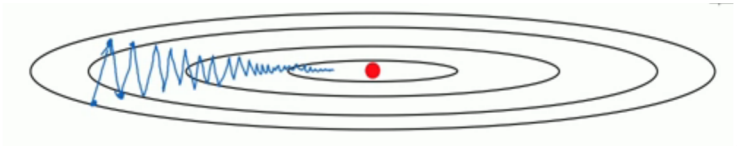
Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, Momentum optimization will roll down the bottom of the valley faster and faster until it reaches the bottom (the optimum).[4]

1,2,3 , Source: http://ruder.io/optimizing-gradient-descent/
4 Source: : Hands-On Machine Learning with Scikit-Learn & TensorFlow

:: Gradient Descent

Momentum leads to faster convergence and fewer oscillations

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.[1]



On horizontal direction, you want to the learning be slower. Because you don't want those oscillations

On horizontal direction, you want faster learning because you want to quickly move toward that red dot (the minimum)

- **Gradient Descent:**

  $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$

  It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly[2]

- **Momentum optimization**

  Momentum optimization cares a great deal about what previous gradients were: at each iteration, it adds the local gradient to the momentum vector $\boldsymbol{v}$.[3]

  With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient. [4]

1 , Source: http://ruder.io/optimizing-gradient-descent/
2,3 , Source: http://ruder.io/optimizing-gradient-descent/
4 Source: http://cs231n.github.io/neural-networks-3/

:: Gradient Descent

Momentum algorithm

Formally, the momentum algorithm introduces a variable v that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient.[1]

Momentum algorithm

On iteration t, compute :

1. $v_t = \beta v_{t-1} + \eta \nabla_\theta J(\theta)$

Learning rate

Momentum Parameter :

It must be set between 0 (high friction) and 1 (no friction).

Its typical value is about **0.9**

Its physical meaning is more consistent with the coefficient of friction

2. $\theta_t = \theta_{t-1} - v_t$

- The velocity v accumulates the gradient elements $\nabla_\theta J(\theta)$
- The larger β is relative to η (the learning rate), the more previous gradients affect the current direction.[2]

1,2 , Source: Ian Goodfellow et al. Deep Learning

:: Gradient Descent

RMSprop

To be added later

:: Gradient Descent

Adam

Adam (Adaptive Moment Estimation) is another method that computes adaptive learning rates for each parameter. Adam tries to combine the best of both world of momentum and adaptive learning rate

The Adam algorithm

1. Compute gradient g_t at current time t
2. Update biased 1st moment estimate
3. Update biased 2st raw moment estimate
4. Update bias-corrected 1st moment estimate
5. Update bias-corrected 2st raw moment estimate
6. Update parameters

Adam performs well in practice

Adam is currently recommended as the default algorithm to use

:: Gradient Descent

Adam – Update biased moment estimate

The algorithm updates exponential moving averages of the gradient (m_t) and the squared gradient(v_t) where the hyper-parameters β_1, β_2 control the exponential decay rates of these moving averages.[1]

The Adam algorithm

1. Compute gradient g_t at current time t
2. Update biased 1st moment estimate
3. Update biased 2st raw moment estimate
4. Update bias-corrected 1st moment estimate
5. Update bias-corrected 2st raw moment estimate
6. Update parameters

Update biased first moment estimate

Adam keeps track of an exponentially decaying average of past gradients m_t like RMSprop

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

// g_t represents the gradient $\nabla_\theta J(\theta)$

// β_1 : Exponential decay rates for the moment estimates. Good default setting: $\beta_1 = 0.9$

Update biased second raw moment estimate

Adam stores an exponentially decaying average of past squared gradients v_t just like RMSProp

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

// g_t^2 indicates the elementwise $g_t \odot g_t$

// β_2 : Exponential decay rates for the moment estimate. Good default setting: $\beta_2 = 0.999$

// The moving averages themselves are estimates of the 1st moment (the mean) and the 2nd raw moment (the uncentered variance) of the gradient[2]

1, 2 Source: ADAM: A METHOD FORSTOCHASTICOPTIMIZATION

:: Gradient Descent

Adam – Bias Correction

The full Adam update includes a bias correction mechanism. About Bias Correction, a good video is recommended: https://www.youtube.com/watch?v=lWzo8CajF5s

The Adam algorithm

1. Compute gradient g_t at current time t
2. Update biased 1st moment estimate
3. Update biased 2st raw moment estimate
4. Update bias-corrected 1st moment estimate
5. Update bias-corrected 2st raw moment estimate
6. Update parameters

As m and v are initialized at 0,

- $m_0 \leftarrow 0$ (Initialize 1st moment vector)
- $v_0 \leftarrow 0$ (Initialize 2st moment vector)

they will be biased toward 0 at the beginning of training .

so these are two steps will help boost m and v at the beginning of training. They counteract these biases by computing bias-corrected first and second moment estimates:

- **Compute bias-corrected first moment estimate**

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^{\,t}}$$

- **Compute bias-corrected second raw moment estimate**

$$\widehat{v_t} = \frac{v_t}{1-\beta_2^{\,t}}$$

Use them to update parameters

1 Source: http://cs231n.github.io/neural-networks-3/

:: Gradient Descent

Adam – Update the parameters

The Adam algorithm

1. Compute gradient g_t at current time t
2. Update biased 1st moment estimate
3. Update biased 2st raw moment estimate
4. Update bias-corrected 1st moment estimate
5. Update bias-corrected 2st raw moment estimate
6. Update parameters

Update parameters

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

where

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

default settings:

$\eta = 0.001$

$\epsilon = 10^{-8}$

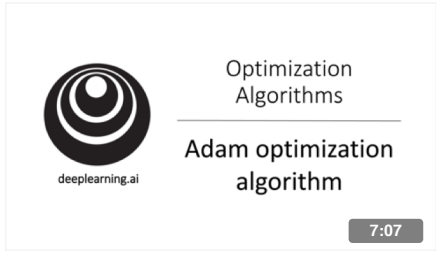
ϵ is a very small number to prevent any division by zero in the implementation

:: Gradient Descent

Adam – wrap up

The Adam algorithm

1. Compute gradient g_t at current time t
2. Update biased 1st moment estimate
3. Update biased 2st raw moment estimate
4. Update bias-corrected 1st moment estimate
5. Update bias-corrected 2st raw moment estimate
6. Update parameters



Youtube Video: Adam Optimization Algorithm
https://www.youtube.com/watch?v=g25U4HSZKmQ

Further Reading:

ADAM: a method for Stochastic Optimization
https://arxiv.org/pdf/1412.6980.pdf

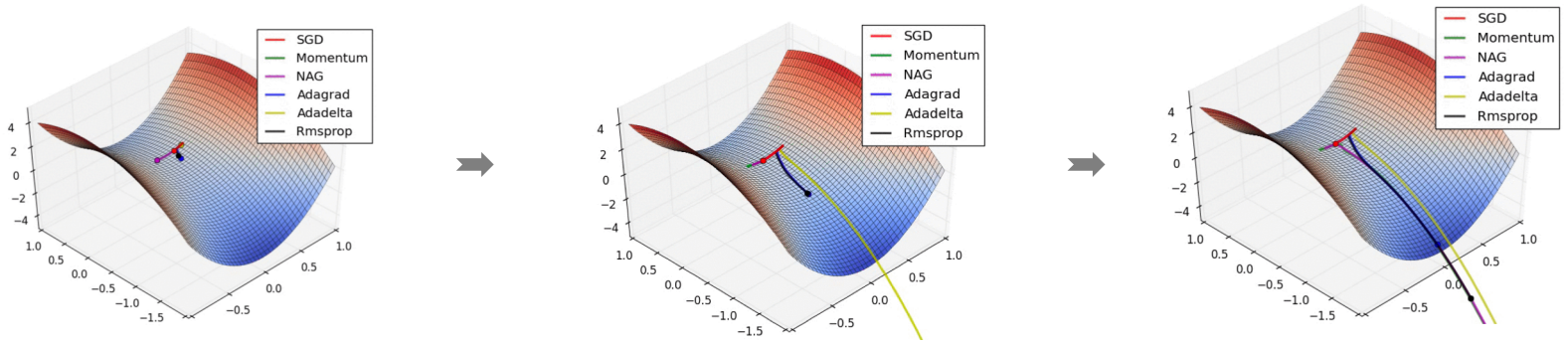
:: Gradient Descent

Visualization of gradient descent optimization algorithms

The following animation shows the behavior of the algorithms at a saddle point, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope, which pose a difficulty for SGD as we mentioned before.

Cool Animation
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

SGD optimization on saddle point：



> Notice here that SGD, Momentum, and NAG find it difficulty to break symmetry, although the two latter eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope.[1]
>
> As we can see, the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios.[2]

1,2 Source: http://ruder.io/optimizing-gradient-descent/
Image credit: Alec Radford

:: Gradient Descent

Which optimizer should we use?

- If your input data is sparse then methods such as **SGD**,**NAG** and **momentum** are inferior and perform poorly. For sparse data sets one should use one of the adaptive learning-rate methods like RMSprop, Adam, etc. An additional benefit is that we won't need to adjust the learning rate but likely achieve the best results with the default value. [1]
- If you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.[2]

Momentum

Nesterov accelerated gradient (NAG)

Adagrad

Adadelta

RMSprop

Adam

Adam might be the best overall choice
Widely used in deep learning community

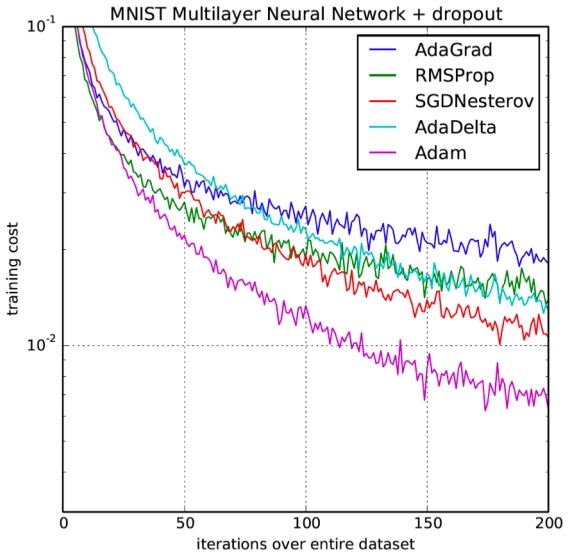


Image 1 Source: ADAM: A METHOD FORSTOCHASTICOPTIMIZATION
Image 2 Source: Designed by Freepik
12 Source: http://ruder.io/optimizing-gradient-descent/
Source: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

:: Gradient Descent

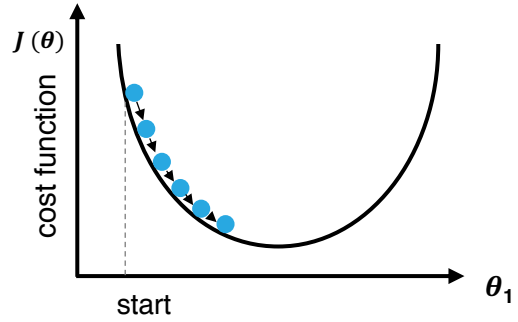
Finding a good Learning Rate can be tricky

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyper-parameter.[1]

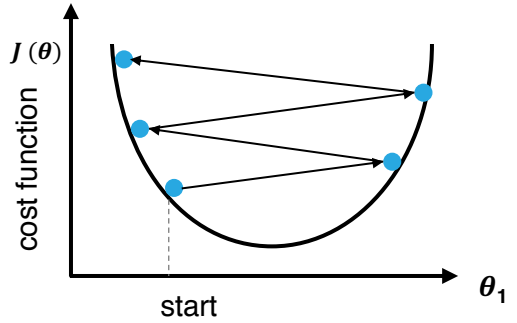
$$\theta := \theta - \alpha \cdot \frac{\partial}{\partial \theta} J(\theta)$$

Learning rate α
(Step size)

- **If α is too small: slow convergence**
  gradient descent would take long time to converge and can be very slow



- **If α is too large: $J(\theta)$ may note decrease on every iteration; may not converge**
  gradient descent can overshoot the minimum. You might jump across the valley and end up on the other side, possibly even higher up than you were before[2]. So the algorithm may fail to converge, or even diverge.



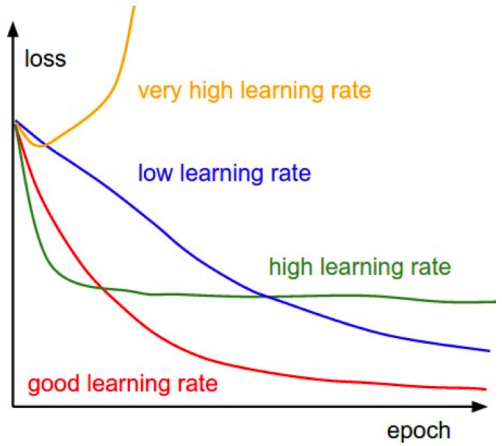
1,2 Source: Aurélien Géron , Hands-On Machine Learning with Scikit-Learn & TensorFlow

:: Gradient Descent

Learning Rate Scheduling (Learning rate annealing)

Slowly reducing the learning rate over time might speed up your learning gradient descent optimization

The cost function changes over time with different learning rates



- With low learning rates the improvements will be linear.
- With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape [1]

You can try to reduce the learning rate over time

Usually using constant learning rate to train the model . However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate.[2]

Since AdaGrad, RMSProp, and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule. For other optimization algorithms, using exponential decay or performance scheduling can considerably speed up convergence.[3]

1 Sourcxe: http://cs231n.github.io/neural-networks-3/#vis
2,3 Source: Aurélien Géron , Hands-On Machine Learning with Scikit-Learn & TensorFlow

:: Gradient Descent

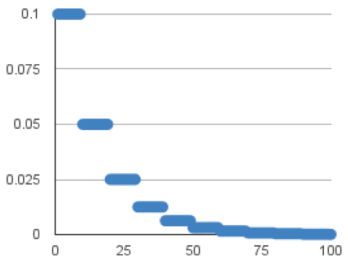
Learning Rate Scheduling (Learning rate annealing)

- **Common strategies to decay the learning rate during training**

  Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay:[1]

Step decay

Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.[2]

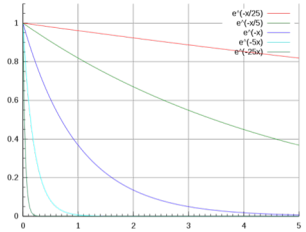


Exponential decay

It has the mathematical form

$$\alpha = \alpha_0 e^{-kt}$$

where α_0 (initial leaning rate) , k (decay rate) are hyper-parameters and t is the iteration number (but you can also use units of epochs).[3]



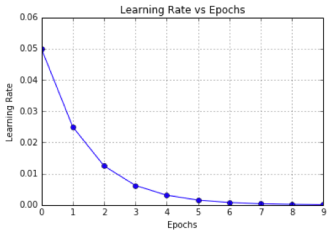
This plot shows decay for decay rates

1/t decay

It has the mathematical form

$$\alpha = \alpha_0/(1 + kt)$$

where α_0 (initial leaning rate) , k (decay rate) are hyper-parameters and t is the iteration number [4]



Video: Learning Rate Decay by Andrew Ng (6 mins)
https://www.youtube.com/watch?v=QzulmoOg2JE

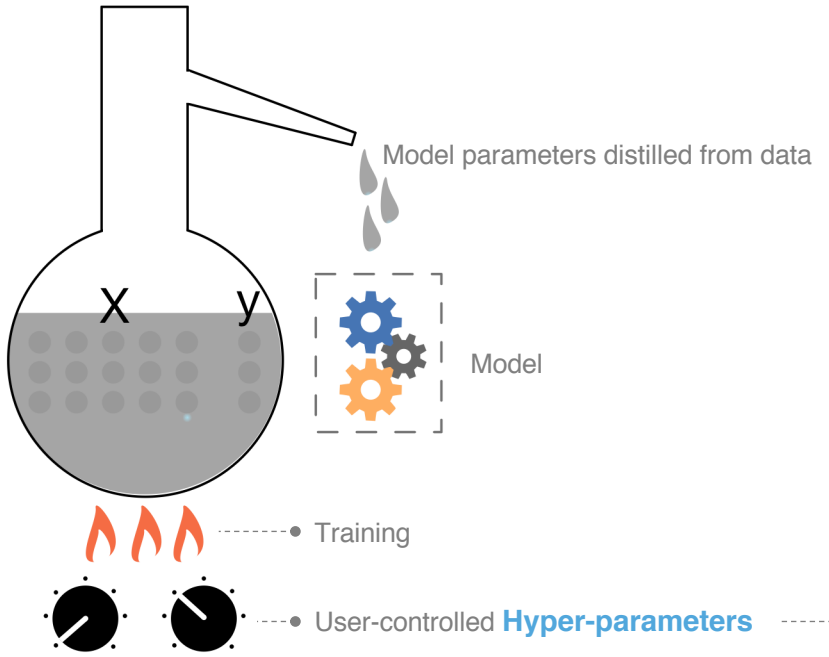
1234 Sourcxe: http://cs231n.github.io/neural-networks-3/#vis

Neural Network – part 2

:: Neural Network

Training neural network involves setting of lots of different hyper-parameters

Hyperparameters control how a machine learning algorithm fits the model to the data. The hyper-parameters are specified by the developer/data scientist while parameters are computed from the data via the algorithms.



Some of these hyperparameters are more important than others: [2]

- α - **Learning rate** : Most important hyperparameter
- β - **Momentum term**
- **Mini-batch size**
- **Number of hidden units for different layers**

  } probably second in importance

- **Number of layers**
- **Learning rate decay**

  } probably 3rd most important hyperparameters

- β_1 β_2 ε (Hyperparameters for Adam Optimization) : not really important
  usually using the default value is enough

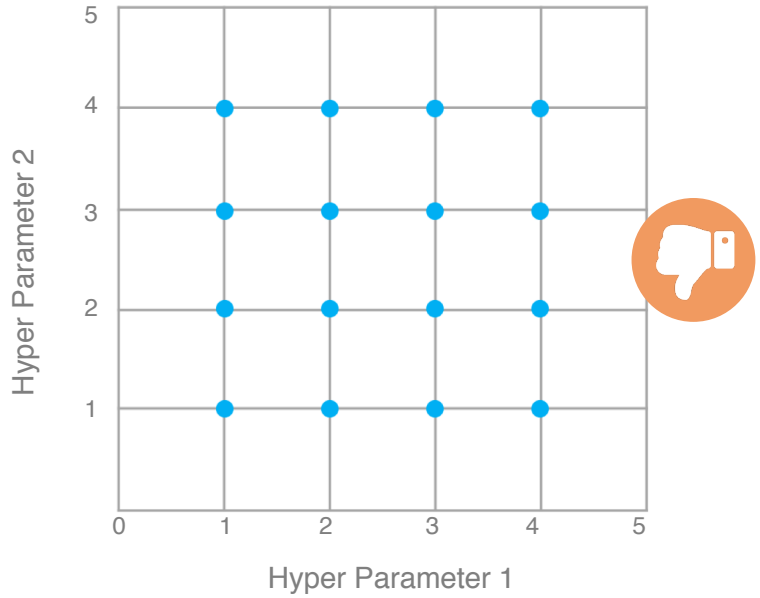
1 & Source of figure: A First Taste of Machine Learning, https://mlbook.explained.ai/intro.html#sec:2.2.3
2 source: Andrew Ng, tuning Process : https://www.youtube.com/watch?v=AXDByU3D1hA

:: Neural Network

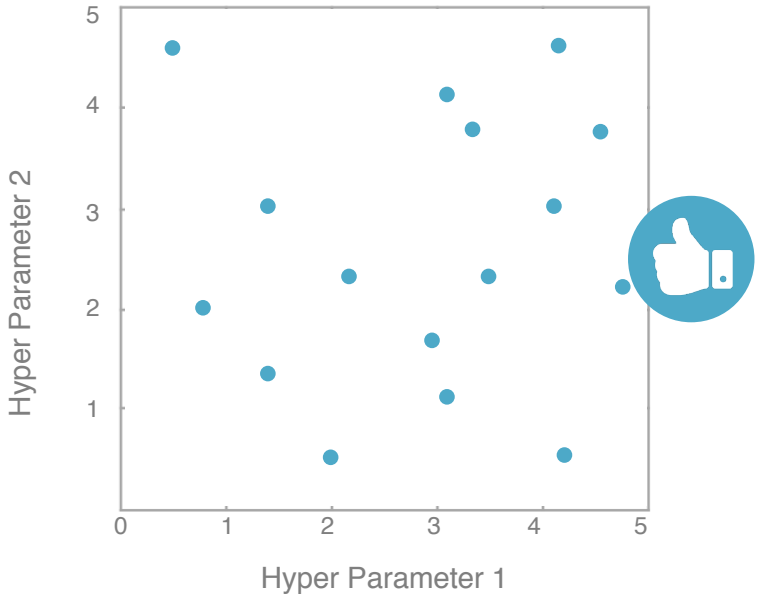
How to find a good setting for the hyper-parameters ?

One of painful things about training deep nets is about the sheer number of hyper-parameters you have to deal with [1]. Using grid search to find good setting is not recommended. **Don't use grid search, try random values !**

Grid search



Random search



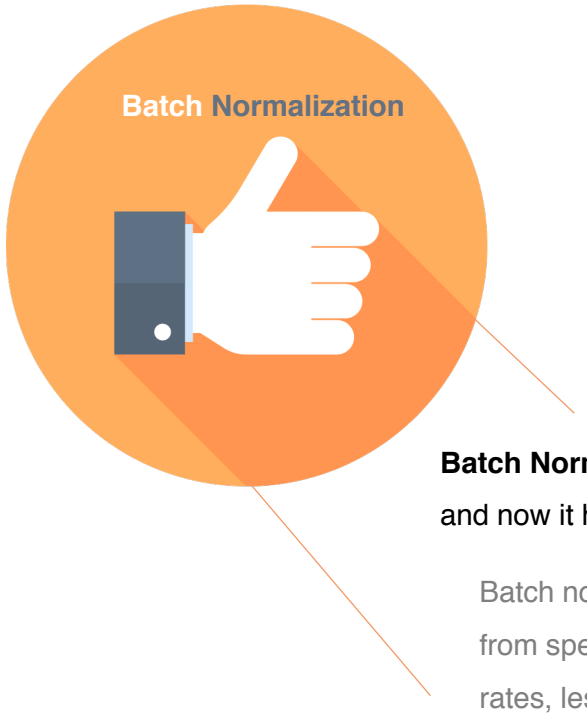
YouTube Video : Tuning Process (7mins)
https://www.youtube.com/watch?v=AXDByU3D1hA

1source: Andrew Ng, tuning Process : https://www.youtube.com/watch?v=AXDByU3D1hA

:: Neural Network

Batch normalization is a milestone technique in optimizing deep neural networks

Batch normalization is a technique for improving the performance and stability of neural networks. Batch normalization makes your hyperparameter search problem much easier, makes your neural network much more robust to the choice of the hyperparameters. And will also enable you to much more easily train even very deep networks[1]



Batch Normalization was introduced by 2 Google researchers in 2015. And it's so successful and now it has recently become a part of the standard toolkit for training deep networks.

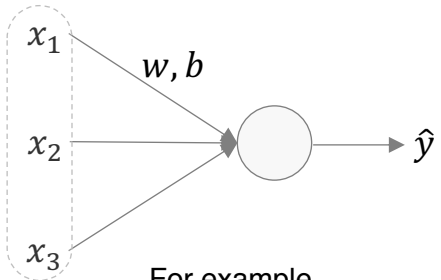
> Batch normalization is very effective in accelerating in training of the deep network. Apart from speed improvements, the technique reportedly enables the use of higher learning rates, less careful parameter initialization, and saturating nonlinearities. [2]

source of pictogram: www.flaticon.com
1 source: Andrew Ng, https://www.coursera.org/lecture/deep-neural-network/normalizing-activations-in-a-network-4ptp2
2. source: https://kth.diva-portal.org/smash/get/diva2:955562/FULLTEXT01.pdf

:: Neural Network

Do you still remember feature scaling ?

When using Gradient Descent, you should ensure that all features have a similar scale, or else it will take much longer to converge [1]. In general, gradient descent converges much faster with feature scaling than without it. [2]



For example,

- x_1 is house's size . The range is (0 - 2000 feet^2).
- x_2 is the number of bedrooms. Its range is (1-5).
- x_3 is the age of house. Its range is from 1 to 33 years.

Because they have different scale, it would take longer time to converge.

So you can use Z-score Normalization to speed up the learning

Use normalization to make sure multiple features are on a similar scale.

$$z = \frac{x - \mu}{\sigma}$$

where

- x is an **original value**,
- μ is the **mean** of that feature vector
- Sigma σ is its **standard deviation** from the mean

Video: Why normalize inputs ? by Andrew Ng (5 mins)
https://www.youtube.com/watch?v=FDCfw-YqWTE

1 Source: Hands-On Machine Learning with Scikit-Learn & TensorFlow 2 source: 李弘毅 https://www.youtube.com/watch?v=BZh1ltr5Rkg

:: Neural Network

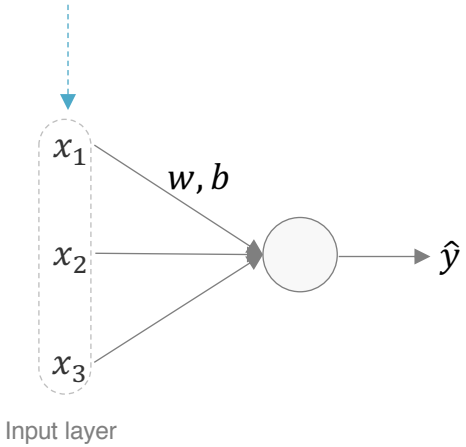
Use Batch Normalization (also known as Batch Norm) in the network

What Batch Norm does is it applies that normalization process not just to the input layer, but also to the hidden unit values Z in the hidden layers in the neural networks.

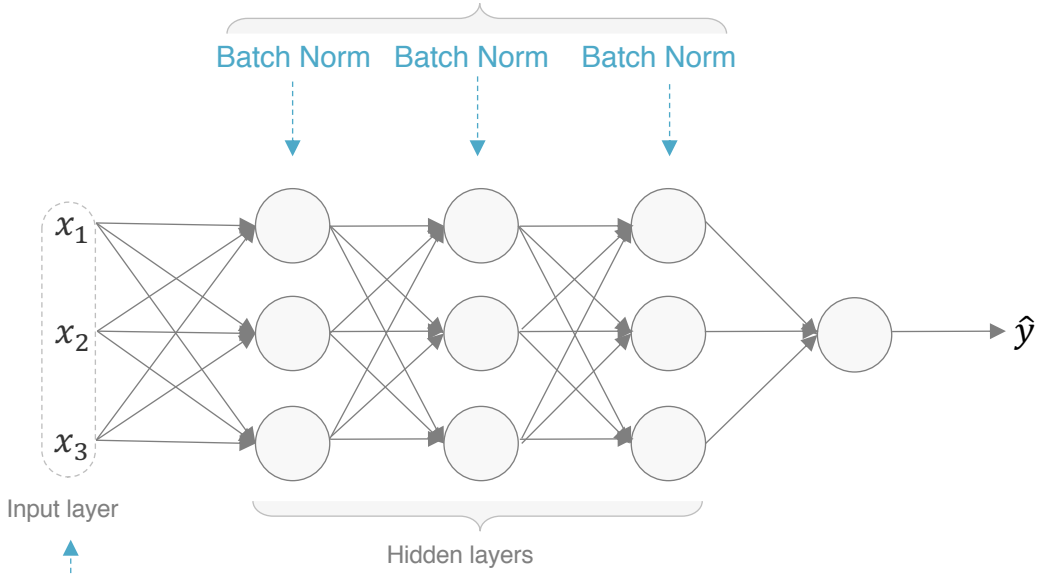
The idea is to normalize the inputs of each layer to provide any layer in a Neural Network with inputs that are zero mean/unit variance.

Feature scaling:

Normalize the inputs to speed up learning



Apply **Batch Norm** in hidden layers



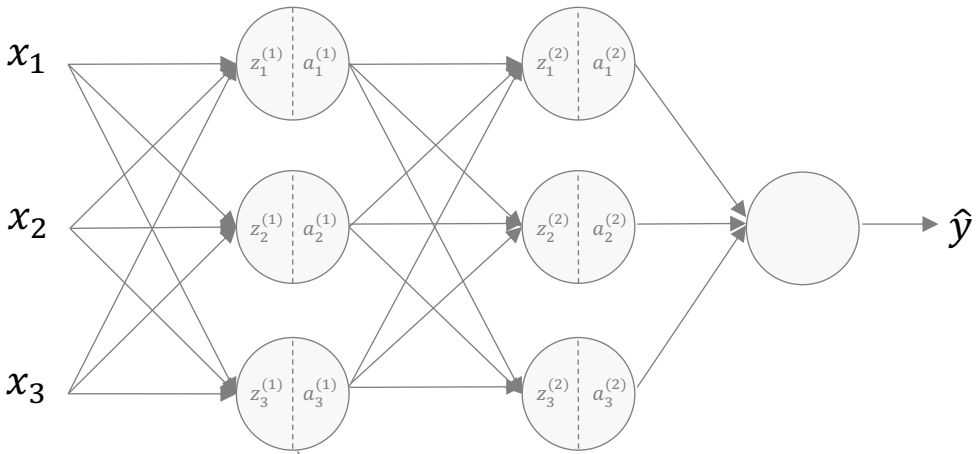
Feature scaling:

Normalize the inputs to speed up learning

:: Neural Network

Let's see how Batch Normalization works

let's take a look at the regular neural network first.



Neural network **without** applying Batch Norm

Each hidden unit computes two things

1. Compute z
2. And then fit z into **activation function** to compute a .
   for example,

$$a_3^{(1)} = g\left(z_3^{(1)}\right)$$

activation function

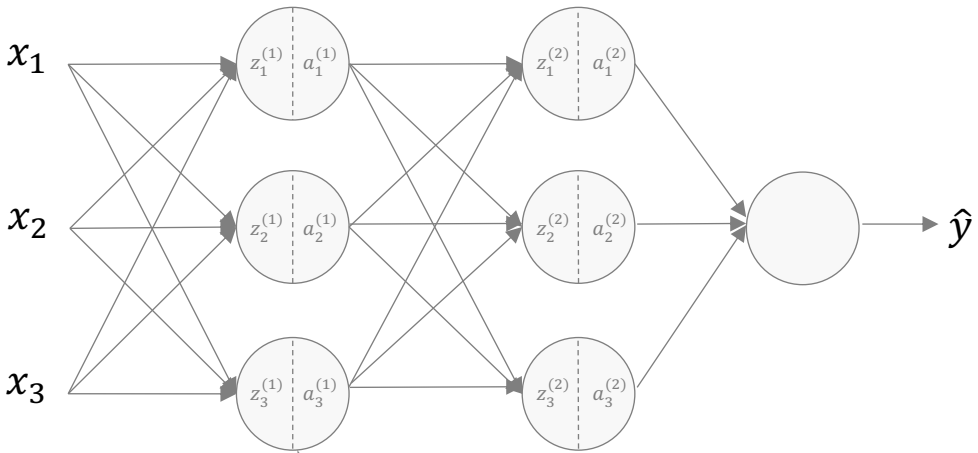
The left figure is about the neural network which is not applied Batch Norm. The output of each hidden in hidden layer, namely a, will be the input to hidden units in next layer.

What will happen if applying Batch Norm in hidden layers ?

:: Neural Network

Let's see how Batch Normalization works *- continued*

After using batch norm transformation, you would now use $\tilde{z}$ instead of z for the later computation in your neural network .



Neural network **without** applying Batch Norm

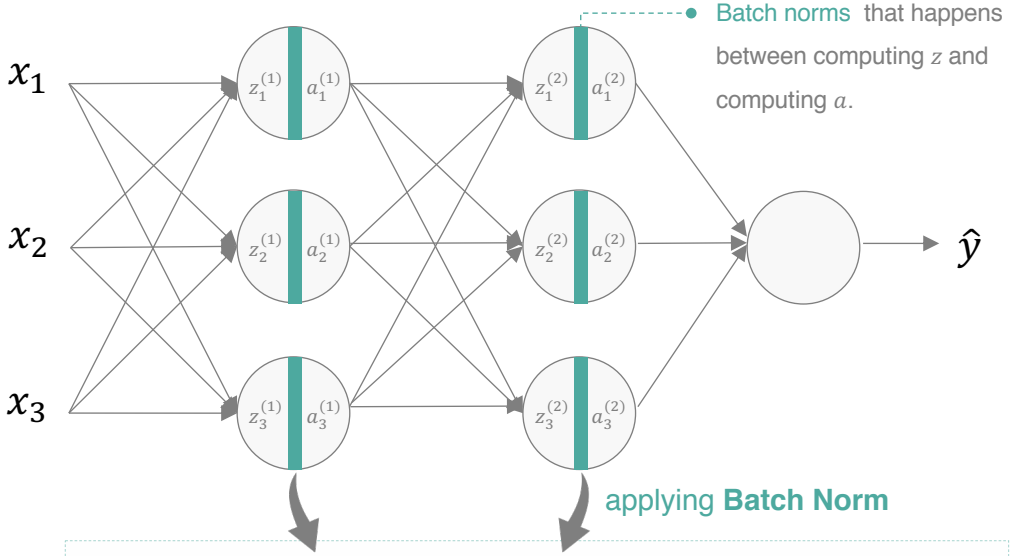
Each hidden unit computes two things

1. Compute z
2. And then fit z into **activation function** to compute a .

for example,

$$a_3^{(1)} = g\left(z_3^{(1)}\right)$$

activation function



applying **Batch Norm**

z — After the Batch Norm Transformation → $\tilde{z}$

Apply **batch norm** to unnormalized value z and get new normalized value $\tilde{z}$, and then fit $\tilde{z}$ instead of z into activation function to get a .

New normalized value $\tilde{z}$

For example, $a^{(1)} = g^{(1)}\left(\tilde{z}^{(1)}\right)$

$a^{(2)} = g^{(2)}\left(\tilde{z}^{(2)}\right)$

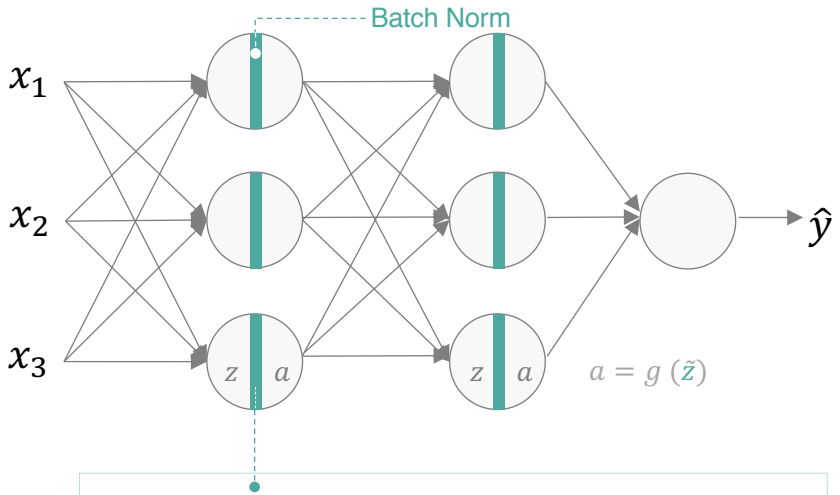
g is activation function

Inspired by video: Andrew Ng, https://www.youtube.com/watch?v=em6dfRxYkYU

:: Neural Network

Let's see how Batch Normalization works *- continued*

After using batch norm transformation, you would now use $\tilde{z}$ instead of z for the later computation in your neural network .



Apply **batch norm** to transform z to new value $\tilde{z}$ just before the activation function of each layer, then fit it into activation function g to compute the result a.

$$a = g(\tilde{z})$$

How to compute $\tilde{z}$?

Input: Consider a mini-batch $\mathcal{B}$ of size m . let's say you have some intermediate values $z^{(1)} \ldots z^{(m)}$ from the hidden layers;

1. $\mu = \frac{1}{m}\sum_{i=1}^{m} z^{(i)}$ // compute mini-batch mean $\mu_{\mathcal{B}}$
2. $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(z^{(i)} - \mu)$ // compute mini-batch variance σ^2
3. $z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ // **Normlaize:** $z_{norm}^{(i)}$ is the zero-centered and normalized input. A tiny number ε is added to avoid division by zero
4. $\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$ // **Scale and shift:** $\tilde{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.[1] Go to next slide for the details.

1 source: Hands-On Machine Learning with Scikit-Learn and TensorFlow

:: Neural Network

The parameter γ , β are used to control the mean and variance

The effect of gamma and beta is that it allows you to set the mean of $\tilde{z}$ to be whatever you want it to be. Later you would use $\tilde{z}$ instead of z for the later computation in your neural network[1]. Note: the batch normalization is performed on each training mini-batch .

1 $$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{(i)}$$

2 $$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (z^{(i)} - \mu)$$

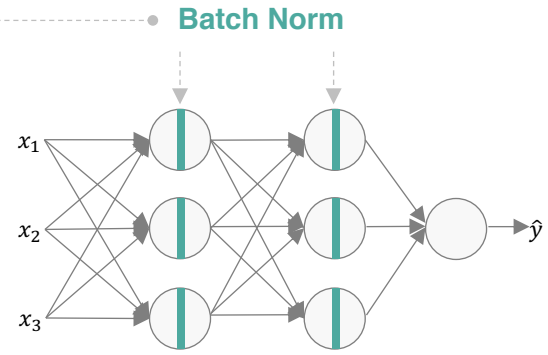
3 $$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Take these value z and normalize them to have **mean zero** and **unit variance.** [2]

4 $$\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$$

Scaling our normalized values z_{norm} by γ and shifting by β

However, we don't want the hidden units to always have mean zero and variance one. By choosing an appropriate setting of gamma γ and beta β, you can make sure that your $\tilde{z}$ have the range of values that you want. So that mean and variance could be zero and one . or it could be some other value and it's controlled by these parameters gamma and beta.[3]



Here β and γ are learnable parameters of your model, just like other parameter w, b. Note: β here is not same with the hyperparameter β in momentum..

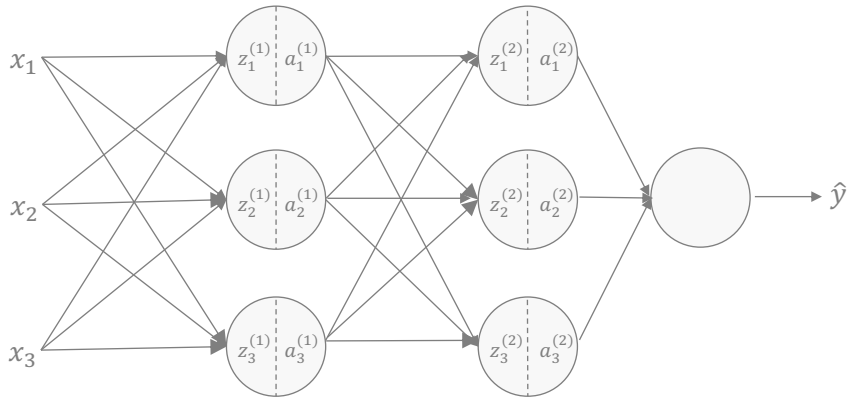
1 ,2,3 source: Andrew Ng, Normalizing Activations in a Networ, https://www.youtube.com/watch?v=tNIpEZLv_eg

:: Neural Network

Why does Batch Norm work ?

One reason is that normalizing input features and the value in the hidden units can speed up the learning. A second reason why batch norm works, is that it reduces **internal covariate shift** problem. It accomplishes this via a normalization step that fixes the means and variances of layer inputs.[1]

Internal covariate shift



We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training.[5]

Small changes in the output of the preceding layer might shift the distribution of the inputs of the next layer.

Batch Norm reduces the amount that the distribution of the hidden unit values shifts around

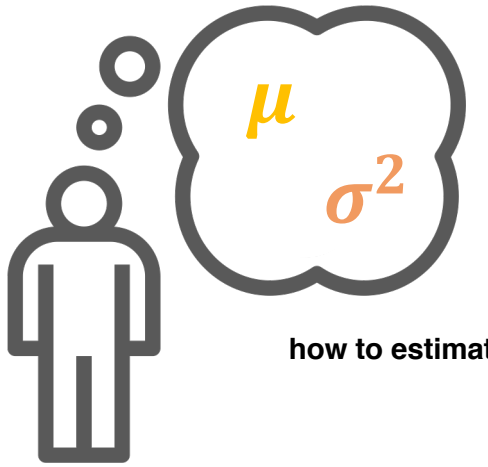
Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as internal covariate shift.[2]

The inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters amplify as the network becomes deeper [3]. The change in the distributions of layers' inputs presents a problem because the layers need to continuously adapt to the new distribution. When the input distribution to a learning system changes, it is said to experience covariate shift [4].

1,2,3,4 ,5 Source: Sergey Ioff , Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Batch Norm at testing time

Batch norm processes your data one mini batch at a time, but at the test time you might need to process the examples one at a time [1]. That means there is no mini-batch to compute the empirical mean and standard deviation at test time.



how to estimate μ and σ^2 ?

we don't have **batch** at testing stage

- **Ideal solution:**
  compute μ and σ using the whole training dataset. [2]

- **Practical solution:**
  computing the moving average of μ and σ of the batches during training [3]

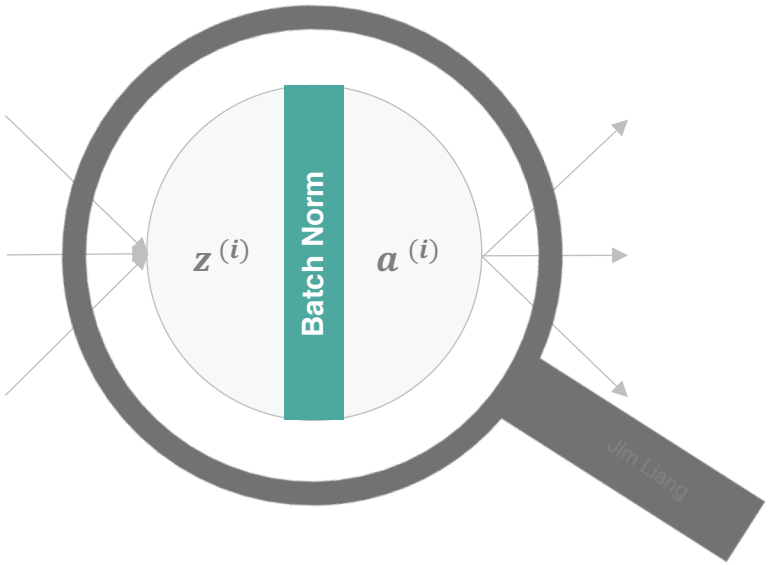
Video: Batch Norm At Test Time (5mins)
https://www.youtube.com/watch?v=5qefnAek8OA

1 source: Andrew Ng, https://www.youtube.com/watch?v=5qefnAek8OA
2、3 source: 李弘毅, https://www.youtube.com/watch?v=BZh1ltr5Rkg

:: Neural Network

The benefit of Batch Norm

By normalizing activations, batch normalization helps stabilize the distributions of internal activations as the model trains. Batch normalization also makes it possible to use significantly higher learning rates, and reduces the sensitivity to initialization. These effects help accelerate the training, sometimes dramatically so. [1]



- **Batch Norm reduces training times of deep network**
  - Because of less Covariate Shift, we can **use much higher learning rates** without the risk of divergence. High learning rate significantly speeding up the learning process. In traditional deep networks, too-high learning rate may result in the gradients that explode or vanish, as well as getting stuck in poor local minima [2].
  - **Less exploding/vanishing gradients.** The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the logistic activation function [3]
- **The networks were also much less sensitive to the weight initialization**
- **Batch Norm has a slight regularization effect** [4]

1 source: Sergey Ioffe, Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-NormalizedModels
2 source: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
3 source: Hands-On Machine Learning with Scikit-Learn and TensorFlow
4 source: Andrew Ng, Why does Batch Norm work?

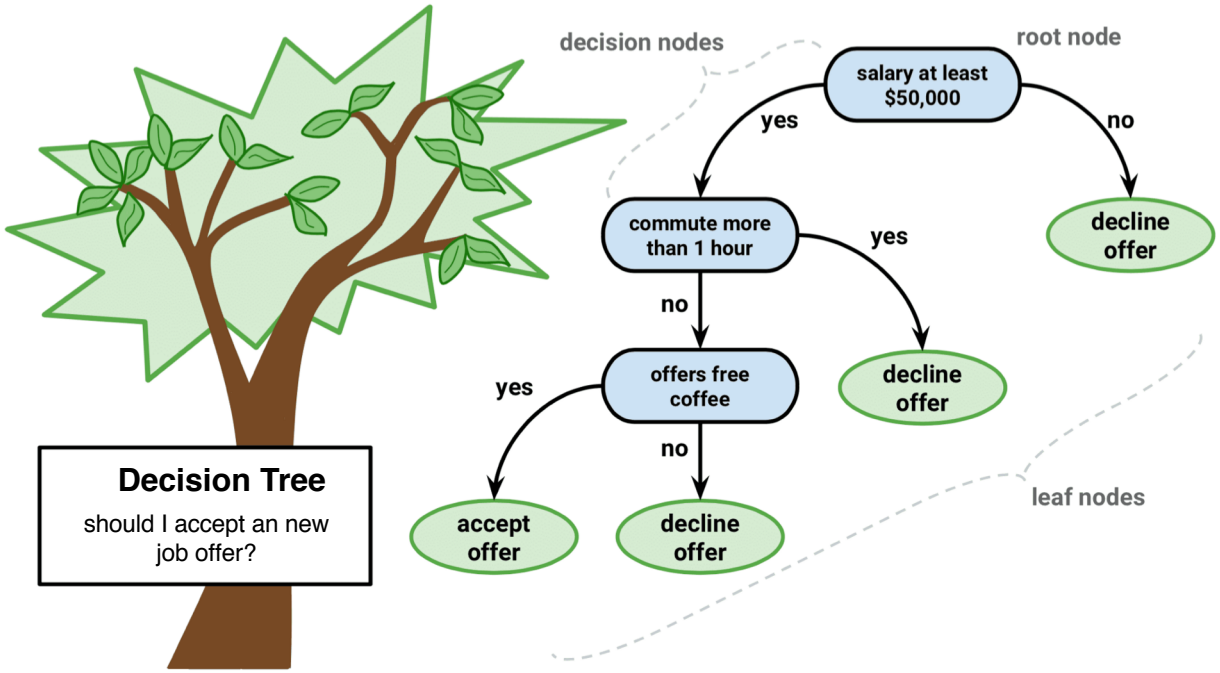
Decision Tree

:: Decision Tree

How it Works

Decision tree builds **classification** or **regression** models in the form of a tree structure. A decision tree lets you predict the value of a target variable by following the decisions in the tree from the root (beginning) down to a leaf node. [1]

A tree consists of branching conditions where the value of a predictor is compared to a trained weight. The number of branches and the values of weights are determined in the training process. Decision trees are prone to overfitting, additional modification, or pruning, may be used to simplify the model. [2]



Decision trees are a non-parametric supervised learning method used for

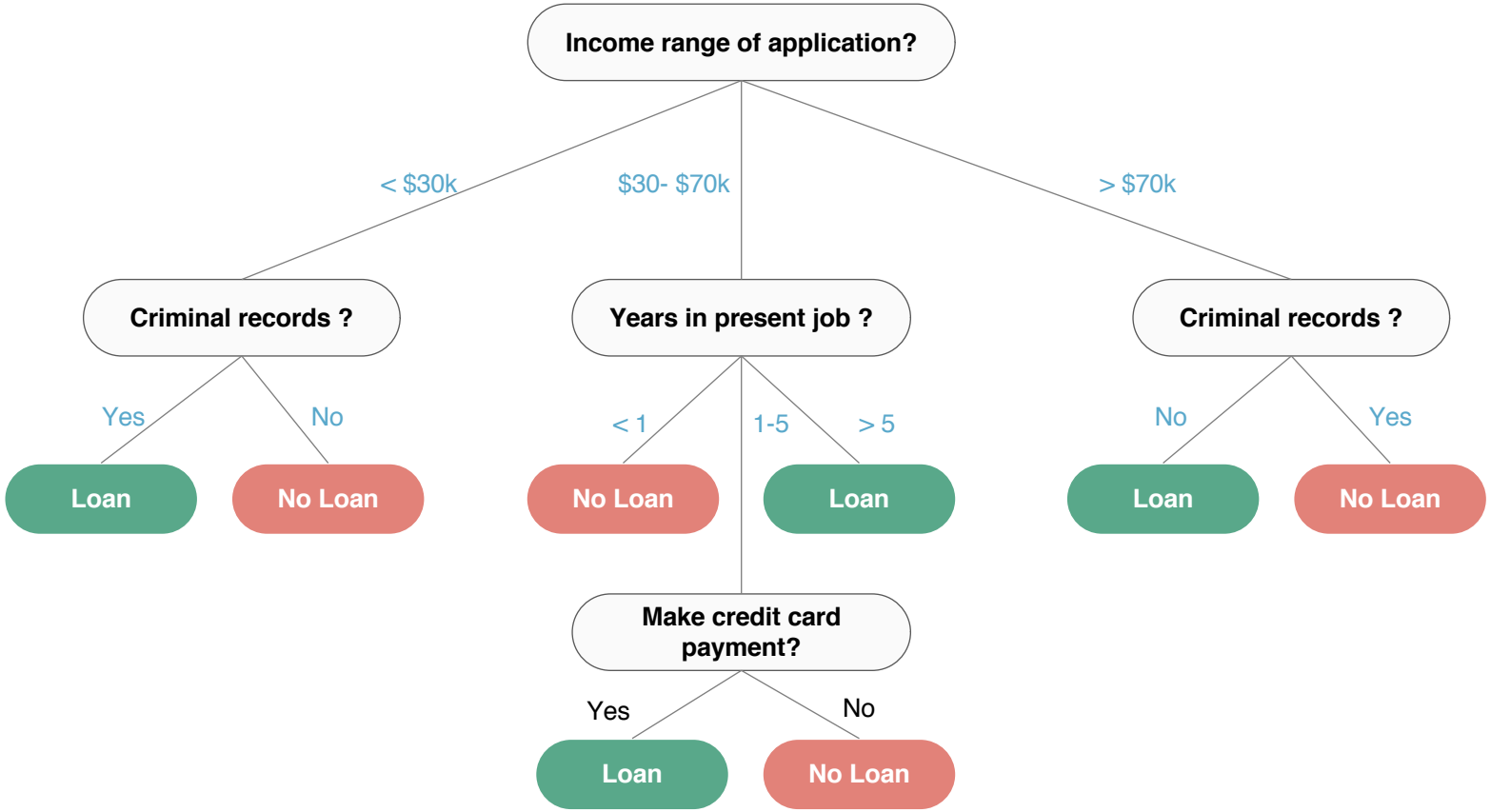
- **Classification**
- **Regression**

1 & 2 Source : Mathworks, Applying Supervised Learning
image: https://cdn-images-1.medium.com/max/2000/1*S10T4ah3_JqdQ-eY6Hau0Q.png

:: Decision Tree

Example 1 : using decision tree to decide whether or not to offer someone a loan

Decision trees can be applied to both classification and regression problems. We might have a decision tree to help a financial institution decide whether a person should be offered a loan

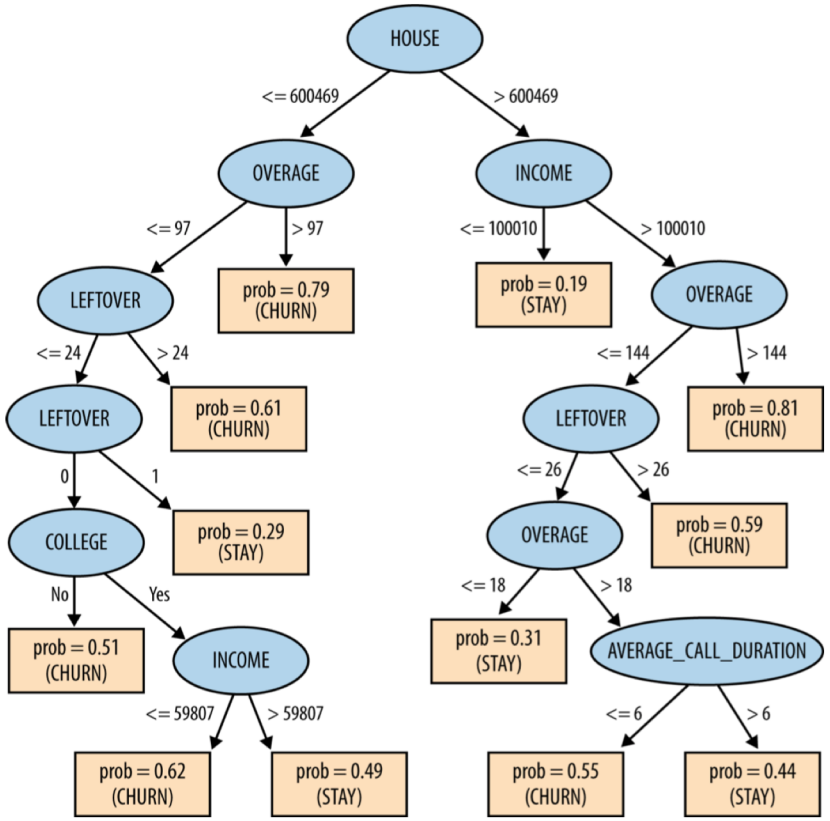


Source of original diagram:https://towardsdatascience.com/what-is-a-decision-tree-22975f00f3e1

:: Decision Tree

Example 2: Classification tree learned from the cellular phone churn data

A Decision Tree can also estimate the probability that an instance belongs to a particular class k. [1]



Probabilities at the leaves are the estimated probabilities of churning for the corresponding segment;

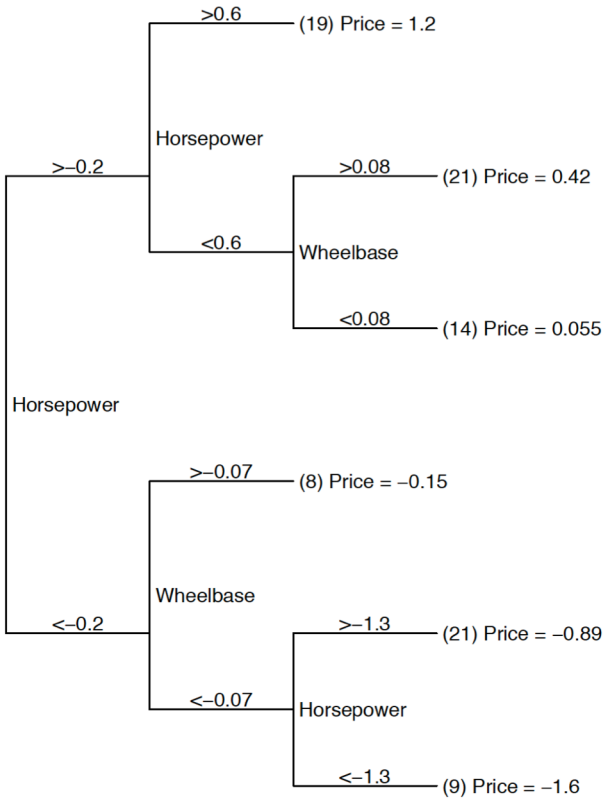
In parentheses are shown the classifications resulting from applying a decision threshold of 0.5 to the probabilities (i.e., are the individuals in the segment more likely to CHURN or to STAY?). [2]

Variable	Explanation
COLLEGE	Is the customer college educated?
INCOME	Annual income
OVERAGE	Average overcharges per month
LEFTOVER	Average number of leftover minutes per month
HOUSE	Estimated value of dwelling (from census tract)
HANDSET_PRICE	Cost of phone
LONG_CALLS_PER_MONTH	Average number of long calls (15 mins or over) per month
AVERAGE_CALL_DURATION	Average duration of a call
REPORTED_SATISFACTION	Reported level of satisfaction
REPORTED_USAGE_LEVEL	Self-reported usage level
LEAVE *(Target variable)*	Did the customer stay or leave (churn)?

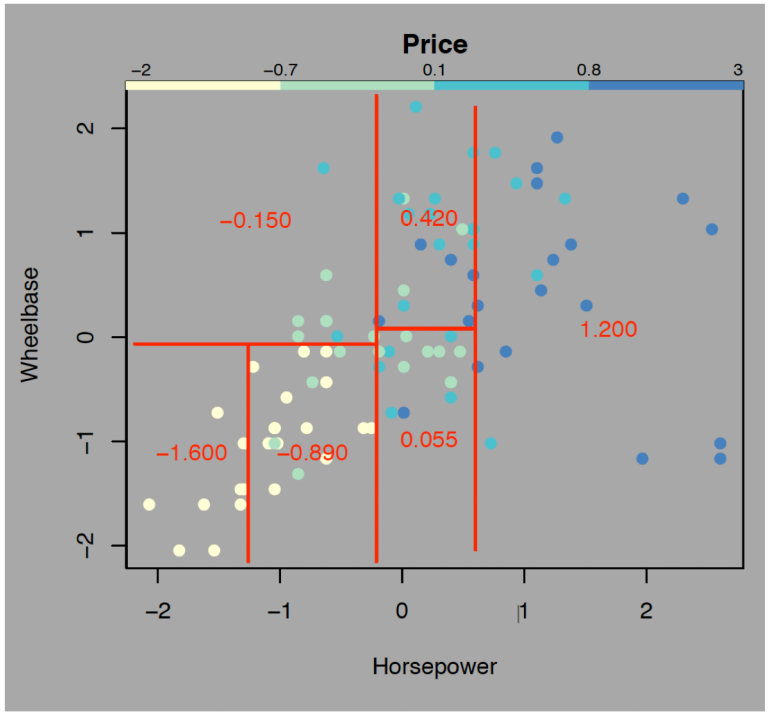
1 Hands-On Machine Learning with Scikit-Learn and TensorFlow
2 & diagram : Data Science for Business

:: Decision Tree

Example 3 : using decision tree to predict the price of 1993-model cars



Note that the order in which variables are examined depends on the answers to previous questions. The numbers in parentheses at the leaves indicate how many cases (data points) belong to each leaf[1]



The partition of the data implied by the regression tree from left figure. . Notice that all the dividing lines are parallel to the axes, because each internal node checks whether a single variable is above or below a given value[2]

1,2 & iamges source: http://www.stat.cmu.edu/~cshalizi/350-2006/lecture-10.pdf

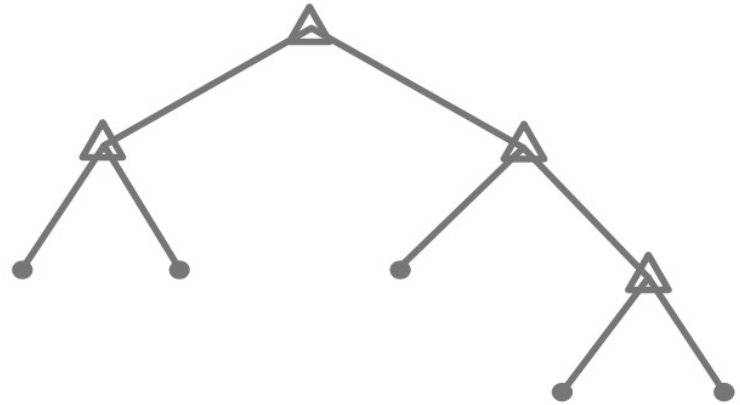
:: Decision Tree

Tree construction algorithms

There are various algorithms used in decision tree

- **ID3 :** It's for classification. It was developed in 1986 by Ross Quinlan.
- **C4.5 :** It's for classification. It is the successor to ID3
- **C5.0:** C5.0 is Quinlan's latest version release under a proprietary license. It uses less memory and builds smaller rulesets than C4.5 while being more accurate.
- **CART** (Classification and Regression Trees): It can used for both classification and regression problem.

Scikit-learn , a famous machine learning library for Python, uses an optimized version of the CART algorithm.

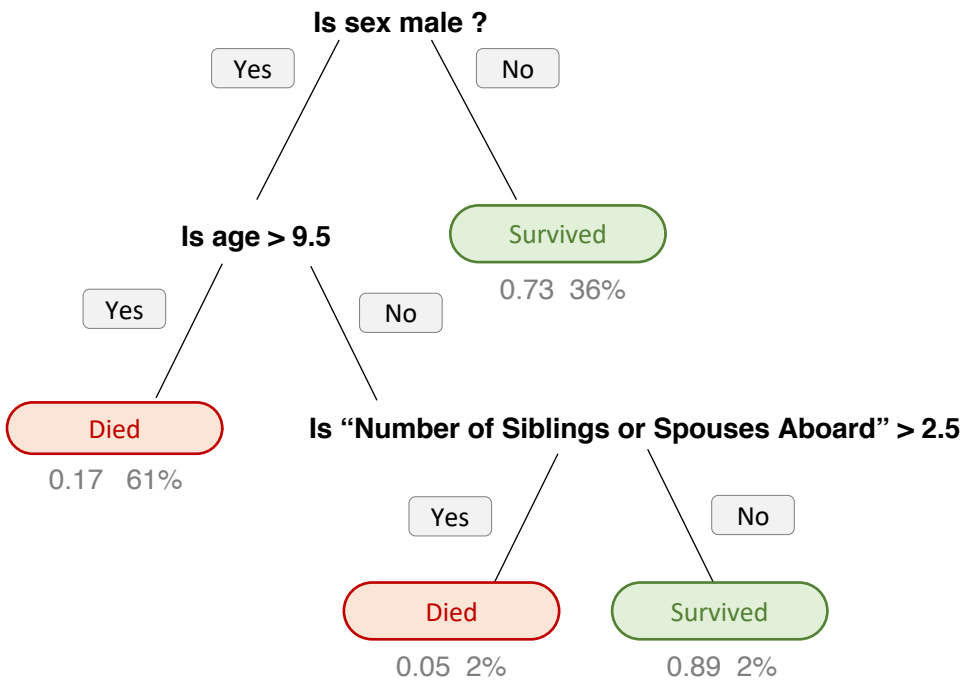


A Decision Tree model contains rules to predict the target variable.

:: Decision Tree

Decision Tree generates the output as a tree-like structure

A decision tree is built top-down from a root node. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with **decision nodes** and **leaf nodes**. A decision node (e.g., Outlook) has two or more branches (e.g., Sunny, Overcast and Rainy). Leaf node (e.g., Play) represents a classification or decision. The topmost decision node in a tree which corresponds to the best predictor called root node. [1]



A tree showing survival of passengers on the Titanic. The figures under the leaves show the probability of survival and the percentage of observations in the leaf.

A decision tree is drawn upside down with its root node at the top.

In the image left, the bold text in black represents a **condition**/internal node, based on which the tree splits into branches/ edges.

The end of the branch that doesn’t split anymore is the decision/leaf, in this case, whether the passenger died or survived, represented as red and green text respectively.[2]

1 http://www.saedsayad.com/decision_tree.htm
2 Source: https://medium.com/towards-data-science/decision-trees-in-machine-learning-641b9c4e8052

:: Decision Tree

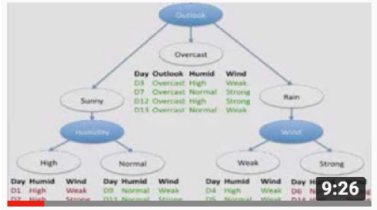
A must-watch video which explains the high level concept of Decision tree

Example: Predict if John will play tennis

Training example: 9 yes / 5 No

Day	Outlook	Humidity	Wind	Play
D1	Sunny	High	Weak	No
D2	Sunny	High	Strong	No
D3	Overcast	High	Weak	Yes
D4	Rain	High	Weak	Yes
D5	Rain	Normal	Weak	Yes
D6	Rain	Normal	Strong	No
D7	Overcast	Normal	Strong	Yes
D8	Sunny	High	Weak	No
D9	Sunny	Normal	Weak	Yes
D10	Rain	Normal	Weak	Yes
D11	Sunny	Normal	Strong	Yes
D12	Overcast	High	Strong	Yes
D13	Overcast	Normal	Weak	Yes
D14	Rain	High	Strong	No

Decision Tree 1: how it works



https://www.youtube.com/watch?v=eKD5gxPPeY0&t=48s

:: Decision Tree

Constructing decision tree

A decision tree is built top-down from a root node.

- Strategy: top- down Recursive divide-and-conquer fashion[1]

    1. **First: select attribute for root node**
       Create branch for each possible attribute value

    2. **Then: split instances into subsets**
       One for each branch extending from the node

    3. **Finally: repeat recursively for each branch, using only instances that reach the branch**

- Stop if all instances have the same class[2]

⇒ Pure subsets

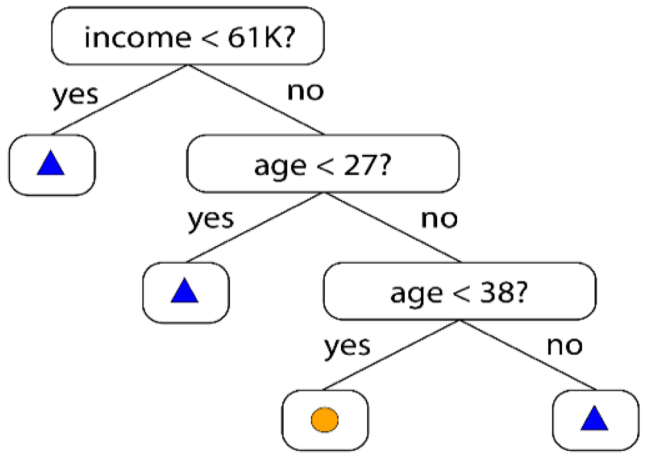
Recursively partitioning the data into subsets that contain instances with similar values (homogenous)

Recursively partitions the training set until each partition consists entirely or dominantly of examples from one class. Each non-leaf node of the tree contains a split point that is a test on one or more attributes and determines how the data is partitioned. The tree is built by recursively partitioning the data. Partitioning continues until each partition is either '**pure**' (all members belong to the same class) or sufficiently small (a parameter set by the user).[3]

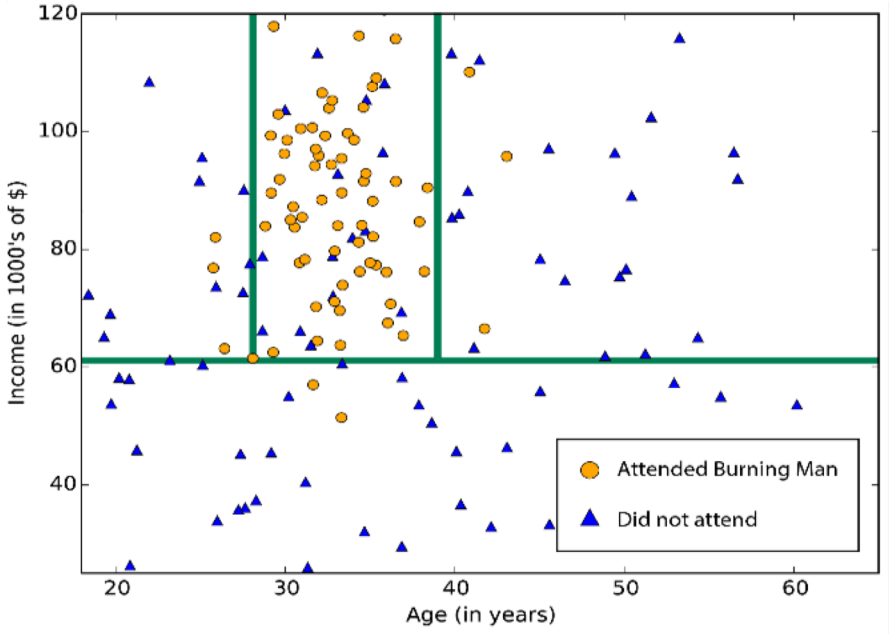
1,2source: Marina Santini Decision Trees 3 Source: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.5.0/com.ibm.im.model.doc/c_decision_tree_calssification.html

:: Decision Tree

Want subsets to be as <u>homogeneous</u> as possible



A decision tree subdivides a feature space into regions of roughly uniform values



Growing a tree involves deciding on **which features to choose** and **what conditions to use** for splitting, along with knowing **when to stop**.

In this procedure all the features are considered and different split points are tried and tested using a cost function. The split with the best cost (or lowest cost) is selected[1].

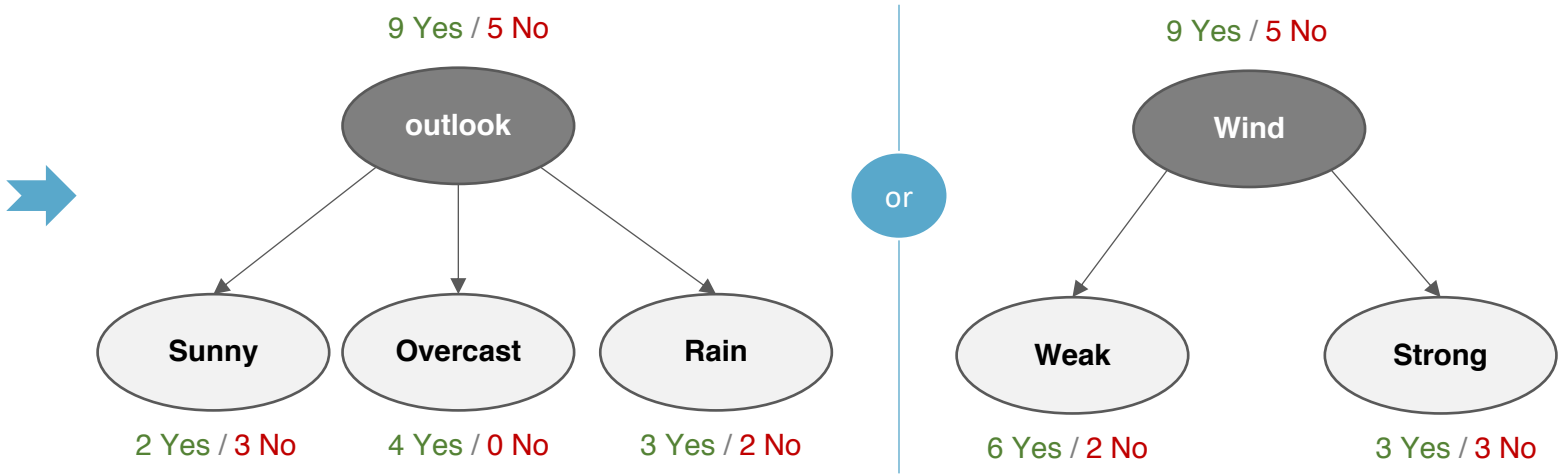
1 Source: https://medium.com/towards-data-science/decision-trees-in-machine-learning-641b9c4e8052
2 Source: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.5.0/com.ibm.im.model.doc/c_decision_tree_calssification.html
Source of diagram: https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice

:: Decision Tree

The key is to decide which attribute to split on

You need to pick the best attribute for splitting the training examples.

Day	Outlook	Humidity	Wind	Play
D1	Sunny	High	Weak	No
D2	Sunny	High	Strong	No
D3	Overcast	High	Weak	Yes
D4	Rain	High	Weak	Yes
D5	Rain	Normal	Weak	Yes
D6	Rain	Normal	Strong	No
D7	Overcast	Normal	Strong	Yes
D8	Sunny	High	Weak	No
D9	Sunny	Normal	Weak	Yes
D10	Rain	Normal	Weak	Yes
D11	Sunny	Normal	Strong	Yes
D12	Overcast	High	Strong	Yes
D13	Overcast	Normal	Weak	Yes
D14	Rain	High	Strong	No



- **To compare the different ways to split data in a node**
- **Want to measure the "purity" of the split [2].**
  - Pure set (4yes/ 0 No) ⇒ completely certain(100%)
  - Impure (3 yes/ 3 No) ⇒ completely uncertain(50%)

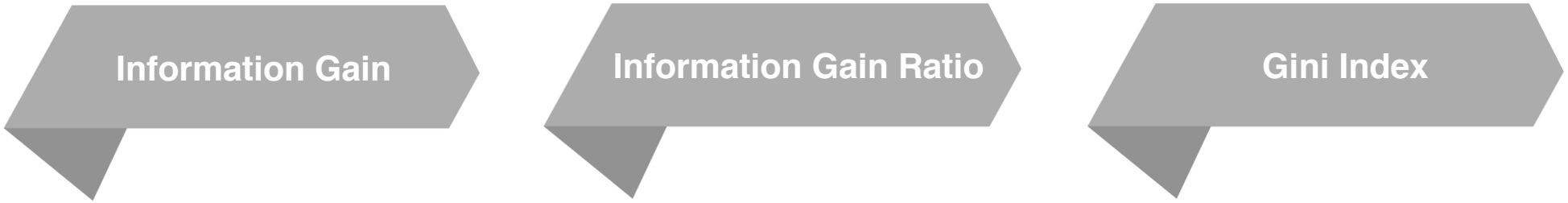
- **Splits on all attributes are tested**
  - Constructing a decision tree is all about finding attribute that returns the highest information gain or lowest Gini Index (i.e., the most homogeneous branches)[3].

1 Source: https://medium.com/towards-data-science/decision-trees-in-machine-learning-641b9c4e8052
2 source: https://www.youtube.com/watch?v=AmCV4g7_-QM&t=11s
3 Source: http://www.saedsayad.com/decision_tree.htm

:: Decision Tree

Measures that can be used to capture the purity of split

Information Gain, Gain Ratio and Gini Index are the most common methods of *attribute selection*



:: Decision Tree

We need to understand Entropy first before we move on

ID3 uses *Entropy* and *Information Gain* to construct a decision tree. It's necessary to understand what Entropy means.

What's Entropy ?

3 Source: http://www.saedsayad.com/decision_tree.htm

:: Decision Tree

Generally entropy is a measure of disorder or uncertainty

Entropy is a concept used in Physics, mathematics, computer science (information theory) and other fields of science.[1] The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. [2]

Entropy in Physics:

Low

Medium

High

Entropy, so far, had been a concept in physics. If the particles inside a system have many possible positions to move around, then the system has **high entropy**, and if they have to stay rigid, then the system has **low entropy.** [2]

For example, water in its three states, solid, liquid, and gas, has different entropies. The molecules in ice have to stay in a lattice, as it is a rigid system, so ice has low entropy. The molecules in water have more positions to move around, so water in liquid state has medium entropy. The molecules inside water vapor can pretty much go anywhere they want, so water vapor has high entropy.[3]

1 source: https://bricaud.github.io/personal-blog/entropy-in-decision-trees/
2 Source: Hands-On Machine Learning with Scikit-Learn and TensorFlow
3,4 & image source: https://medium.com/udacity/shannon-entropy-information-gain-and-picking-balls-from-buckets-5810d35d54b4

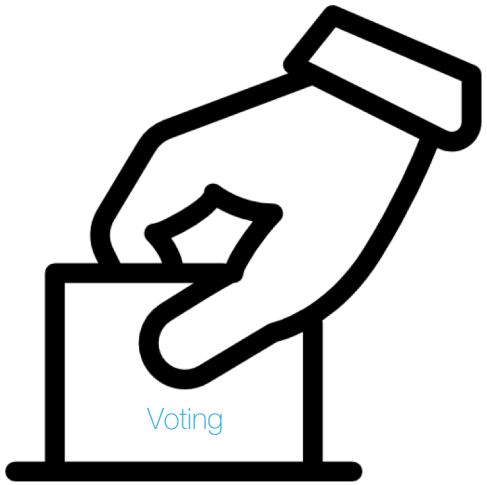
:: Decision Tree

Entropy in Information theory

Information entropy is defined as the average amount of information produced by a stochastic source of data… Generally, information entropy is the average amount of information conveyed by an event

- **when a low-probability event occurs** (i.e. when the data source has a lower-probability value,)
  the event carries **more "information" ("surprisal")** than when the source data has a higher-probability value. Rarer events provide more information when observed.[1]

- **when a high-probability event occurs ,**
  the event with high-probability value is less informative than less common event. Entropy is zero when one outcome is certain to occur.



Entropy is a measure of *unpredictability* of the state, or equivalently, of its *average information content*. To get an intuitive understanding of these terms, consider the example of a political poll. [2]

- Usually, such polls happen because the outcome of the poll is not already known. In other words, the outcome of the poll is relatively *unpredictable*, and actually performing the poll and learning the results gives some new *information*; these are just different ways of saying that the *a priori* entropy of the poll results is large. [3]

- Now, consider the case that the same poll is performed a second time shortly after the first poll. Since the result of the first poll is already known, the outcome of the second poll can be predicted well and the results should not contain much new information; in this case the *a priori* entropy of the second poll result is small relative to that of the first.[4]

1 source: https://en.wikipedia.org/wiki/Entropy_(information_theory)

:: Decision Tree

More uncertainty, more entropy!

entropy is a measure of

disorder or **uncertainty**

High Entropy

- High Disorder
- Messy
- High Randomness;
- High uncertainty
- Low certainty

Increase entropy

Low Entropy

- Well- ordered
- Neat
- Low Randomness
- Low uncertainty
- High certainty

:: Decision Tree

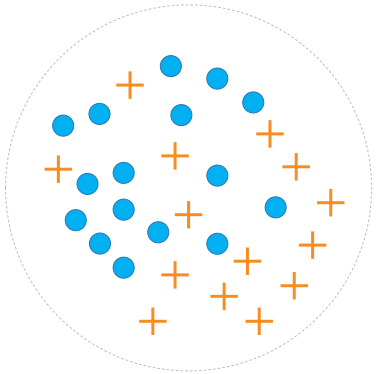
Mathematical definition of entropy

The measure of information entropy associated with each possible data value is the negative logarithm of the *probability mass function* for the value.[1] If we have a set with n different values in it, we can calculate the entropy as follows:

$$Entropy = -\sum_{i=1}^{n} p_i \log_2 p_i$$

- Where p_i is the **probability** of getting the i^{th} value when randomly selecting one from the set.
  In other words, Where there are n classes, and p_i is the probability an object from the i^{th} class appearing.

Example



Entire population (30 instances)

16/30 are blue circles:

$$\log_2\left(\frac{16}{30}\right) = -0.9\ ;$$

14/30 are orange crosses:

$$\log_2\left(\frac{14}{30}\right) = -1.1\ ;$$

$$\boldsymbol{Entropy} = -\sum_{i=1}^{n} p_i \log_2 p_i = -\left(\frac{16}{30}\right)(-0.9) - \left(\frac{14}{30}\right)(-1.1) = 0.99$$

Video: Shannon Entropy and Information Gain (21mins)
https://www.youtube.com/watch?time_continue=1247&v=9r7FIXEAGvs

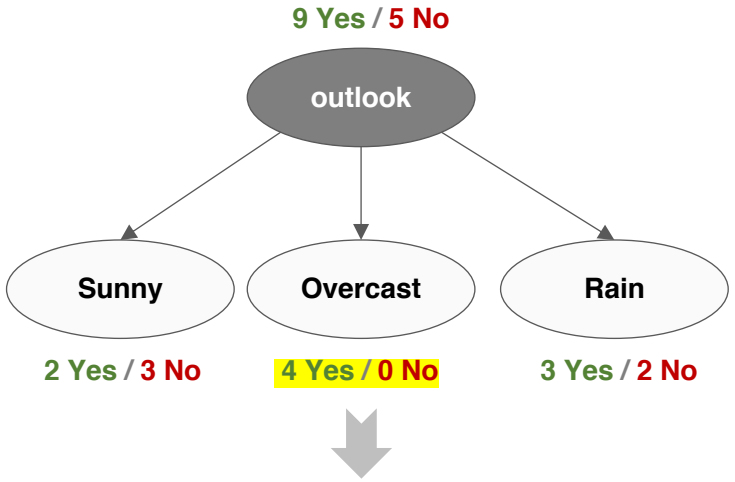
1 source: https://en.wikipedia.org/wiki/Entropy_(information_theory)
Source of example: Linda Shapiro, Introduction to Artificial Intelligence

:: Decision Tree

Another example of calculating entropy

Use entropy to measure the “purity” of the split

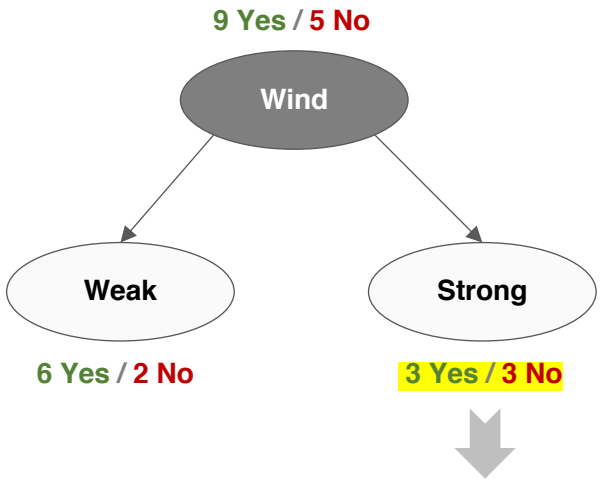
$$Entropy = -\sum_{i=1}^{n} p_i \log_2 p_i$$



Pure set (4 yes / 0 No) ⇒ completely certain(100%)

The entropy is 0 If the sample is completely homogeneous

$$Entropy = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4} = 0$$



Impure set (3 yes / 3 No) ⇒ completely uncertain(50%)

The entropy is 1 If the sample is an equally divided

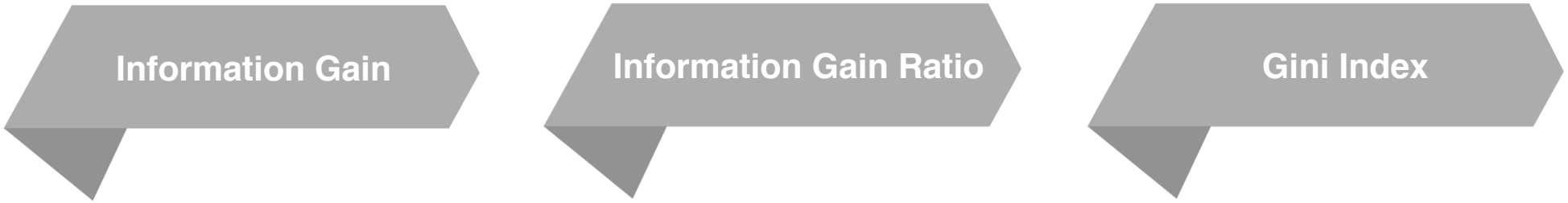
$$Entropy = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1$$

Video : Decision tree entropy (5mins)
https://www.youtube.com/watch?v=tJmhT3oLXCU

:: Decision Tree

Measures that can be used to capture the purity of split

Information Gain, Gain Ratio and Gini Index are the most common methods of *attribute selection*



:: Decision Tree

Measures that can be used to capture the purity of split – Information Gain

Information gain increases with the average purity of the subsets. Strategy: choose attribute that gives greatest information gain.

Information Gain

i.e. entropy reduction

A reduction of entropy is often called an information gain. ID3 algorithm uses entropy to calculate the homogeneity of a sample.

$$Information\ Gain = Entropy_{before} - Entropy_{after}$$

Constructing a decision tree is all about **finding attribute that returns the highest information gain** (i.e., the most homogeneous branches)

- A decision tree is built top-down from a root node and involves partitioning the data into subsets that contain instances with similar values (homogenous).
- The information gain is based on the decrease in entropy after a dataset is split on an attribute. [3]

3 Source: http://www.saedsayad.com/decision_tree.htm

:: Decision Tree

Measures that can be used to capture the purity of split – Information Gain (continued)

The goal is to decrease in entropy (uncertainty) after splitting. And also to want many items in pure sets

Information Gain

i.e. entropy reduction

Example 1

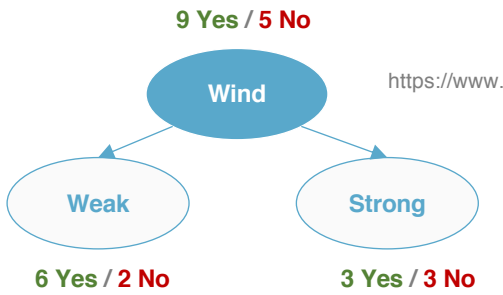
Entropy before splitting
(parent)

$$Information\ Gain(S, A) = H(S) - \sum_{v \in Value\ (A)} \frac{|s_v|}{|s|} H(s_v)$$

Entropy after splitting
(Weighted) Average Entropy of Children

Where
v : *possible value of* A
S: *set of example* $\{x\}$
s_v: *subset where* $x_A = v$

Video: Information Gain (9mins)
https://www.youtube.com/watch?v=2k-ZlWwxoCM



$$H(S_{weak}) = -\frac{6}{8}\log_2\frac{6}{8} - \frac{2}{8}\log_2\frac{2}{8} = 0.81$$

$$H(S_{strong}) = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1$$

$$Gain(S, Wind) = H(S) - \frac{8}{14}H(S_{weak}) - \frac{6}{14}H(S_{strong})$$
$$= 0.94 - \frac{8}{14} * 0.81 - \frac{6}{14} * 1 = 0.049$$

:: Decision Tree

Measures that can be used to capture the purity of split – Information Gain (continued)

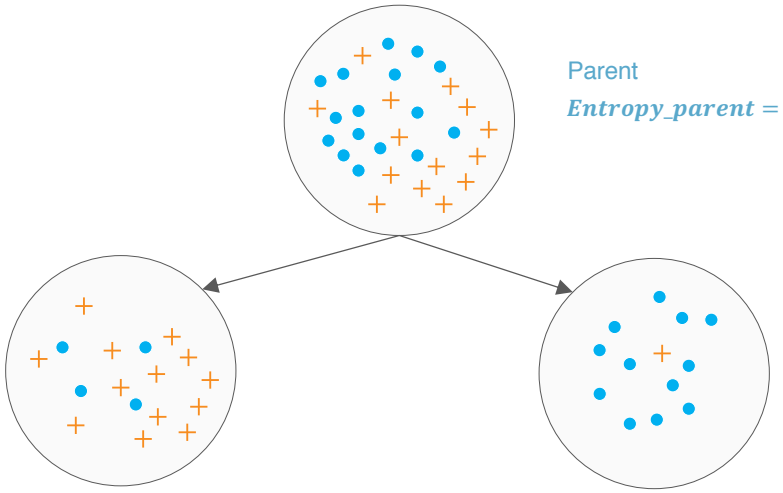
The goal is to decrease in entropy (uncertainty) after splitting. And also to want many items in pure sets

Information Gain

i.e. entropy reduction

Example 2

Entire population (30 instances)



Parent

$$Entropy\_parent = -\frac{14}{30}\log_2\frac{14}{30} - \frac{16}{30}\log_2\frac{16}{30} = 0.996$$

17 instances
(4/17 blue circle, 13/17 orange crosses)

$$Entropy\_child = -\frac{13}{17}\log_2\frac{13}{17} - \frac{4}{17}\log_2\frac{4}{17} = 0.787$$

13 instances
(12/13 blue circle, 1/13 orange crosses)

$$Entropy\_child = -\frac{1}{13}\log_2\frac{1}{13} - \frac{12}{13}\log_2\frac{12}{13} = 0.391$$

$$Entropy\_children = \frac{17}{30} * 0.787 + \frac{13}{30} * 0.391 = 0.615$$

For this split

$$Information\ gain = Entropy_{parent} - Entropy\_chlldren = 0.996 - 0.615 = 0.38$$

Source of example: https://homes.cs.washington.edu/~shapiro/EE596/notes/InfoGain.pdf

:: Decision Tree

Measures that can be used to capture the purity of split – Information Gain Ratio

Information gain approach has a problem that it bias towards attributes that have a large number of values over attributes that have a smaller number of values. These 'Super Attributes' will easily be selected as the root, resulted in a broad tree that classifies perfectly but performs poorly on unseen instances. We can penalize attributes with large numbers of values by using an alternative method for attribute selection, referred to as Gain Ratio.[1]

Information Gain Ratio

- Information Gain ratio overcomes the bias of Information gain.
- C4.5 algorithm uses Information Gain ratio for selecting the best attribute for splitting

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo\ (A)}$$

splitting the training data set D into v partitions, corresponding to v outcomes on attribute A

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2(\frac{|D_j|}{D})$$

- The attribute with the **maximum gain ratio** is selected as the splitting attribute
- Short introduction of Gain Ratio

Video: Information gain ratio (3 mins)
https://www.youtube.com/watch?v=rb1jdBPKzDk

Detailed explanation of Gain Ratio is going to be added into this document later

1 Source: http://www.saedsayad.com/decision_tree_super.htm

:: Decision Tree

Measures that can be used to capture the purity of split – Gini Index

The impurity measure used in building decision tree in CART algorithm is Gini Index. The attribute that maximizes the reduction in impurity is chosen as the splitting attribute

- The impurity measure used in building decision tree in CART algorithm is Gini Index.
- Equation for **Gini impurity**

$$G_i = 1 - \sum_{k=1}^{n} (p_{i,k})^2$$

$p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node

- A node's Gini attribute measures its impurity: a node is "pure" (gini=0) if all training instances it applies to belong to the same class. [1] In other words, Gini Index would be zero if perfectly classified.

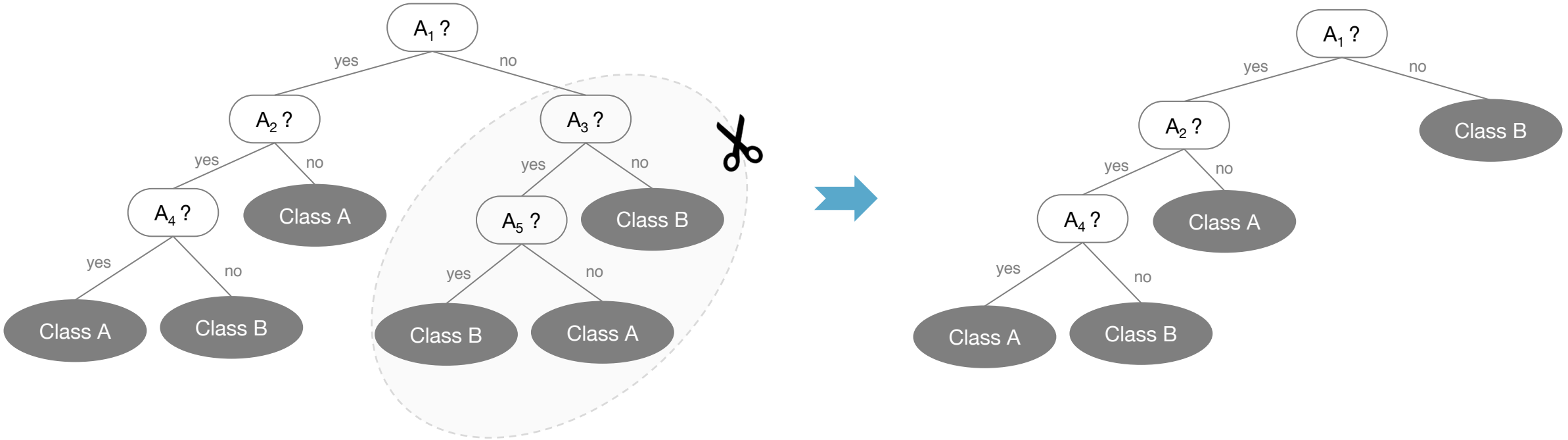
Detailed explanation of Gini Index is going to be added into this document later

1 source: Hands-On Machine Learning with Scikit-Learn and TensorFlow

:: Decision Tree

Address overfitting in Decision Trees by Pruning

Pruning means to change the model by deleting the child nodes of a branch node. Pruning usually results in reducing size of tree, avoids unnecessary complexity, and to avoid overfitting.



The main problem with tree induction is that it will keep growing the tree to fit the training data until it creates pure leaf nodes. This will likely result in large, overly complex trees that overfit the data 1

The pruned node is regarded as a leaf node. Leaf nodes cannot be pruned.

1 Source: Data Science for Business
Source of diagram : http://www.inf.unibz.it/dis/teaching/DWDM/slides2011/lesson5-Classification-2.pdf

:: Decision Tree

Approaches for Pruning

pre-pruning

One strategy to prevent overfitting is to prevent the tree from becoming really detailed and complex by stopping its growth early.

- Halt tree construction early—do not split a node if this would result in the goodness measure falling below a threshold[1]
- Upon halting the node becomes a leaf Upon halting, the node becomes a leaf[2]
- The leaf may hold the most frequent class among the subset tuples[3]

Problem: difficult to choose an appropriate threshold[4]

post-pruning

Another strategy is to build a complete tree with pure leaves but then to prune back the tree into a simpler form

- Remove non-significant branches from a “fully grown” tree[5]
- After trimming, replace subtree by a leaf node
- The leaf is labeled with the most frequent class [6]

Practically, post-pruning strategy is more successful because it is not easy to precisely estimate when to stop growing the tree[7].

1,2,3,4,5,6,Source: http://www.inf.unibz.it/dis/teaching/DWDM/slides2011/lesson5-Classification-2.pdf
7 Source: http://www.saedsayad.com/decision_tree_overfitting.htm

:: Decision Tree

Advantages of Decision Tree

- **Simple to understand and to interpret**

  Decision tree is a “white box” model. Decision Trees are able to produce ‘understandable’ rules. The trees can be also visualized. In contrast, in a black box model (e.g. in neural network, random forest), it is usually hard to explain in simple terms why the predictions were made.

- **To build decision tree requires little data preparation**

  Decision trees do not need feature scaling. Decision trees can deal with a reasonable amount of missing values; Decision trees are also not sensitive to outliers. In contrast, other algorithms often require data normalization, dummy variables need to be created and blank values to be removed[1].

- **Decision trees are able to handle both continuous and categorical variables.**

  Other techniques are usually specialized in analyzing datasets that have only one type of variable[2]. Able to handle multi-output problems[3].

- **Implicitly perform feature selection**

  Decision trees such as CART, for instance, have a built-in mechanism to perform variable selection. Decision trees provide a clear indication of which fields are most important for prediction or classification

1 source: http://scikit-learn.org/stable/modules/tree.html

Image: Designed by Asierromero / Freepik