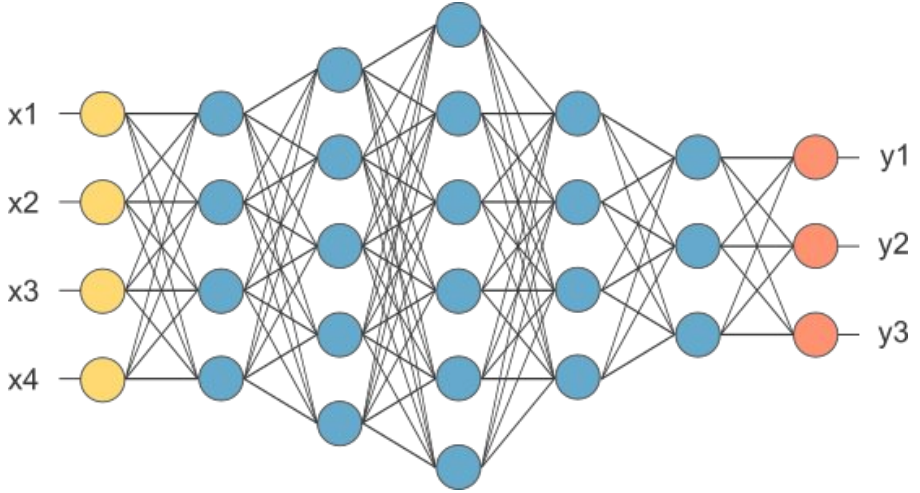
Neural Networks

Introduction to Neural Network

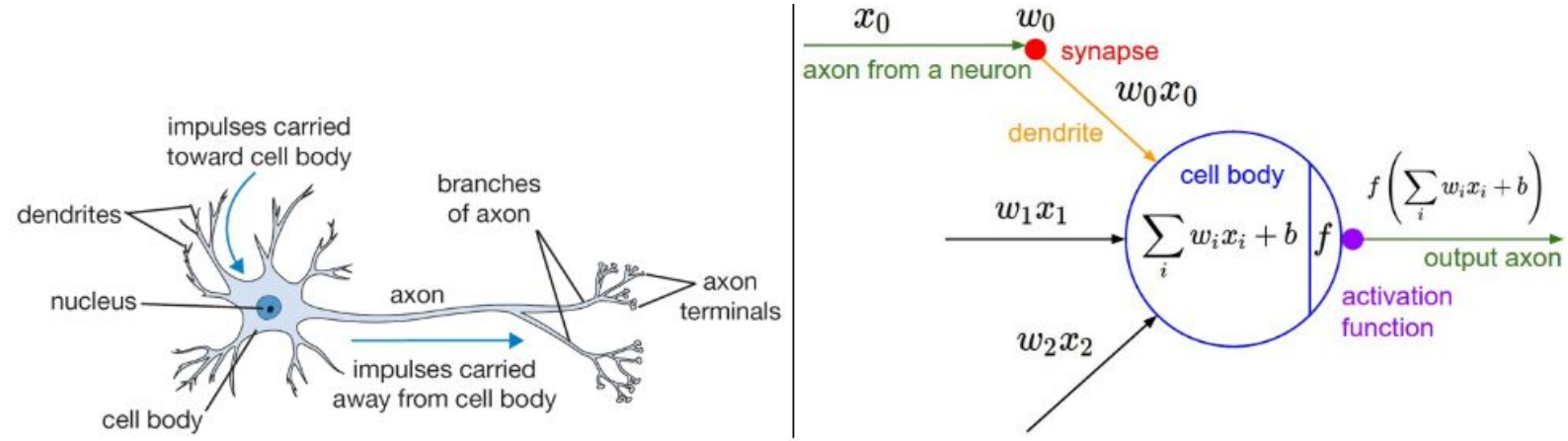
Neural Network

1. From Wiki:
   - NN is based on a collection of connected units of nodes called artificial **neurons** which loosely model the neurons in a biological brain.

2. From another way:
   - NN is running several ‘logistic regression’ at the same time (expanding at width and depth dimensions).



Neuron Computation

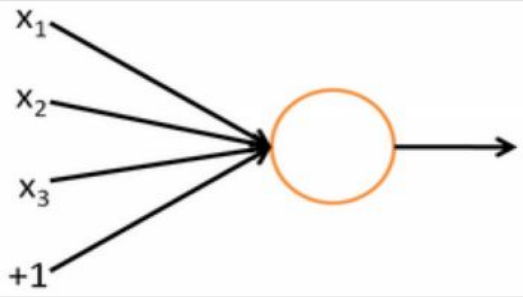


A cartoon drawing of a biological neuron (left) and its mathematical model (right).

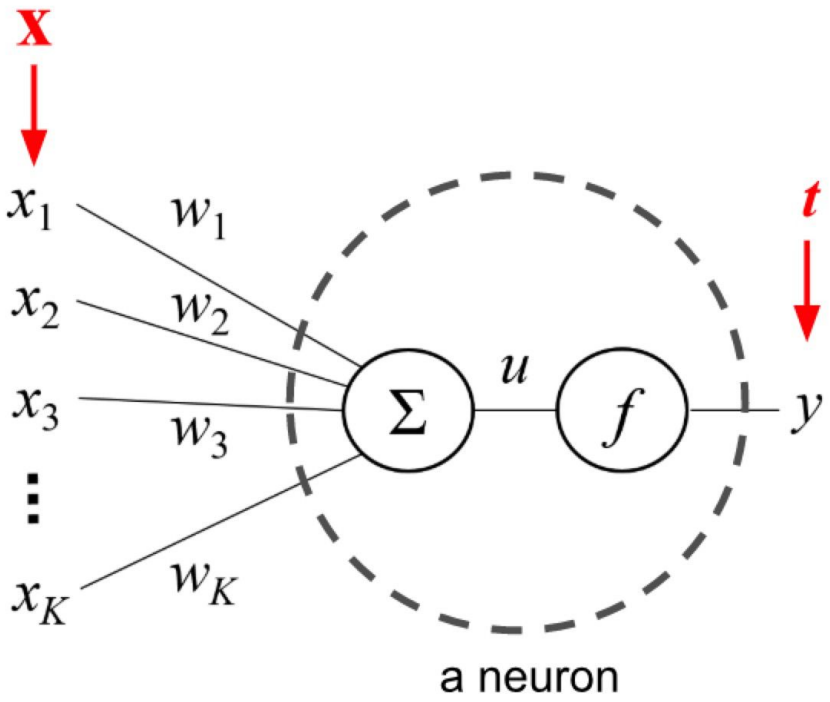
The fact that a neuron is essentially a logistic regression unit:

1 performs a dot product with the input and its weights

2 adds the bias and apply the non-linearity



Neuron Training



$$E = \frac{1}{2}(t-y)^2$$

$$\frac{\partial E}{\partial y} = y - t$$

$$\begin{aligned}\frac{\partial E}{\partial u} &= \frac{\partial E}{\partial y}\cdot\frac{\partial y}{\partial u} \\ &= (y-t)y(1-y)\end{aligned}$$

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial E}{\partial y}\cdot\frac{\partial y}{\partial u}\cdot\frac{\partial u}{\partial w_i} \\ &= (y-t)\cdot y(1-y)\cdot x_i\end{aligned}$$

Layer Computation(Width)

1. Neuron computation: a vector of inputs real-value scalar

2. Layer Computation: feed a vector of inputs into a bunch of neurons at the same time

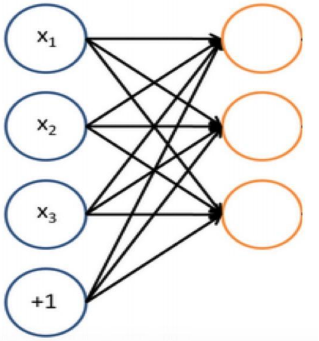
$$a_1 = f(w_1^T x + b_1) = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$

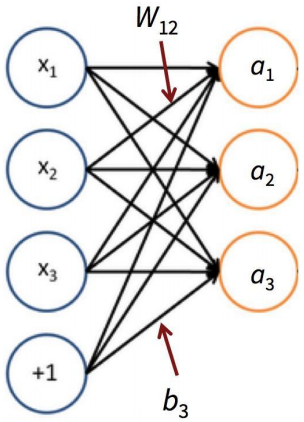
$$a_2 = f(w_2^T x + b_2) = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

3. Math Notation in matrix:

$$z = Wx + b$$

$$a = f(z)$$ where is f is applied element-wise

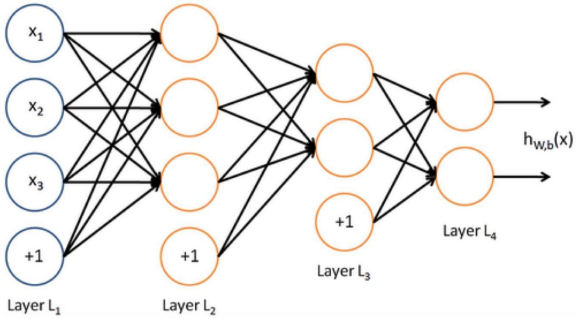




Stacking Layers(Depth)

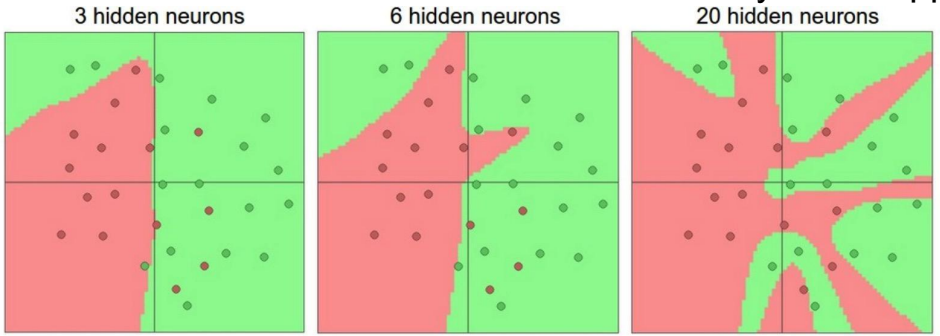
1. We can fed the output of one layer into the next layer（a bunch of neuron computation)

2. The output layer: (not require non-linear activation)
   - For classification: class scores
   - For regression: real-value target

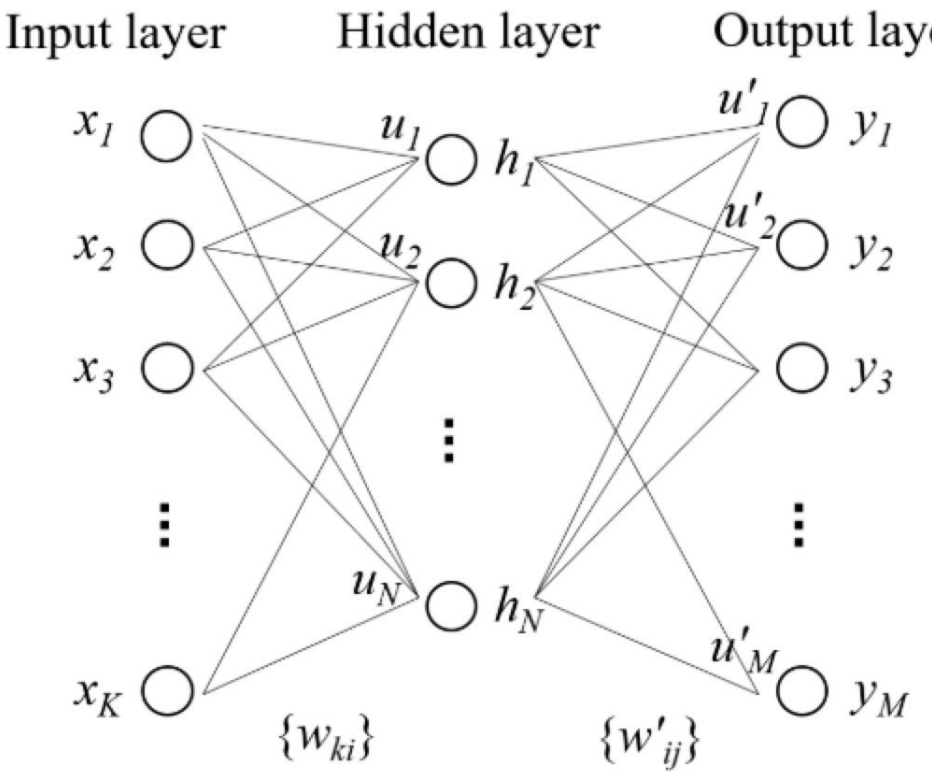


3. Measures to describe the size of neural networks:
   - Number of neurons: 3 + 2 + 2 = 7
   - Number of tunable parameters: (biases and weights)

4. Representation Power: neural networks with at least one hidden layer can approximate any continuous function.



Multilayer Neural Network



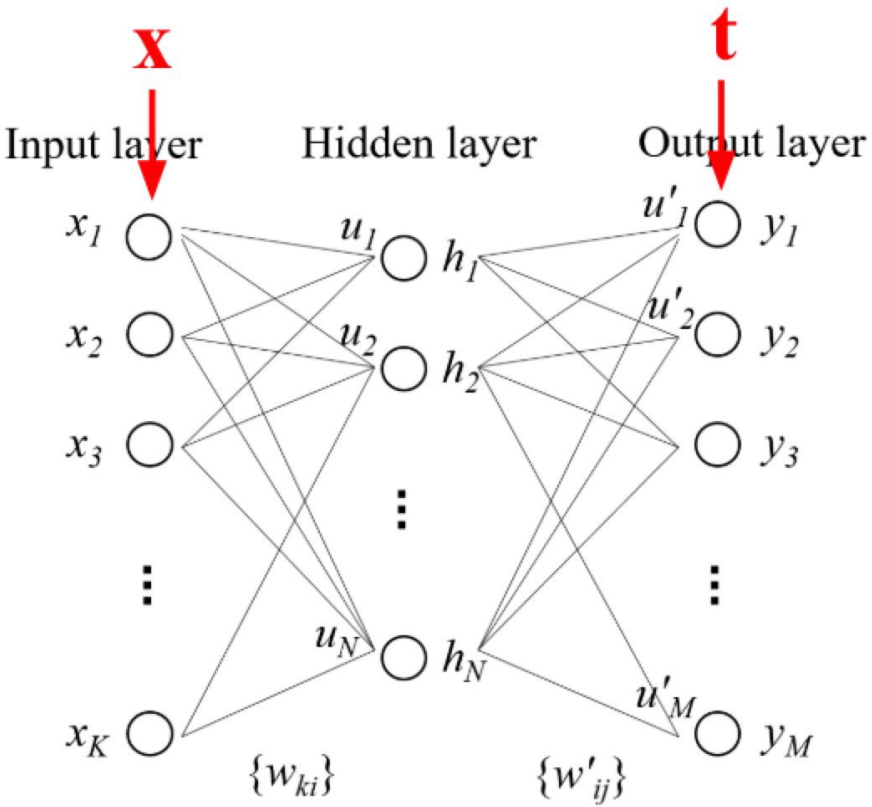
$$u_i = \sum_{k=1}^{K} w_{ki} x_k$$

$$h_i = f(u_i)$$

$$u'_j = \sum_{i=1}^{N} w'_{ij} h_i$$

$$y_j = f(u'_j)$$

Backpropagation



$$E = \frac{1}{2}\sum_{j=1}^{M}(y_j - t_j)^2$$

$$\frac{\partial E}{\partial y_j} = y_j - t_j$$

$$\frac{\partial E}{\partial u'_j} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j}$$

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial w'_{ij}}$$

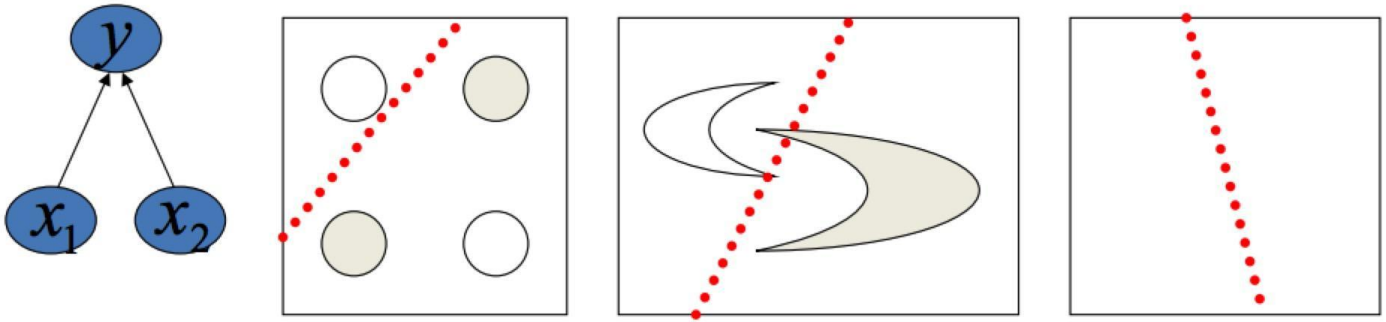
$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{M} \frac{\partial E}{\partial u'_j} \frac{\partial u'_j}{\partial h_i}$$

$$\frac{\partial E}{\partial u_i} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial u_i}$$

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial u_i} \cdot \frac{\partial u_i}{\partial w_{ki}}$$

Decision Level

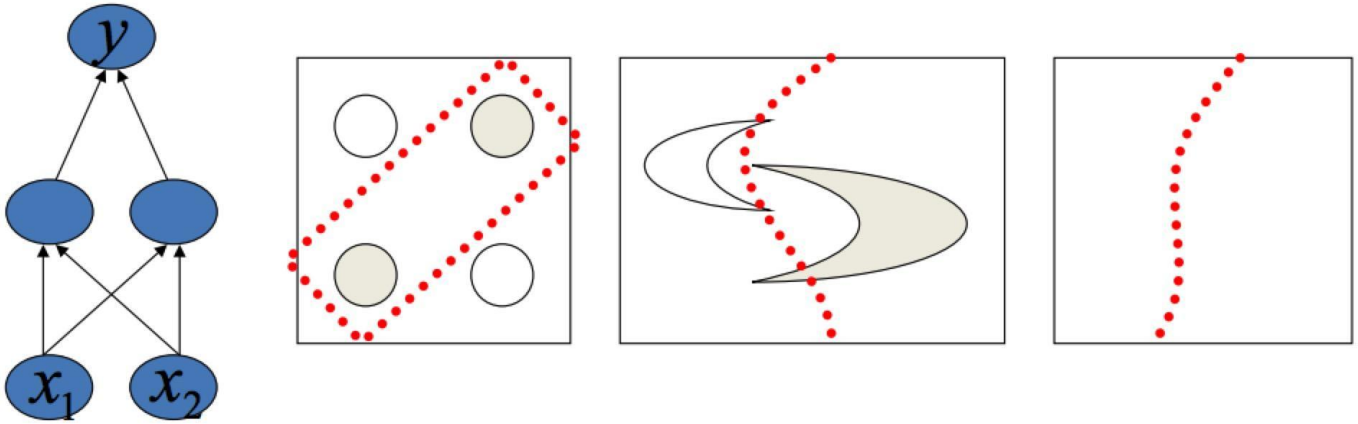
- 0 hidden layers: linear classifier
  - Hyperplanes



Example from to Eric Postma via Jason Eisner

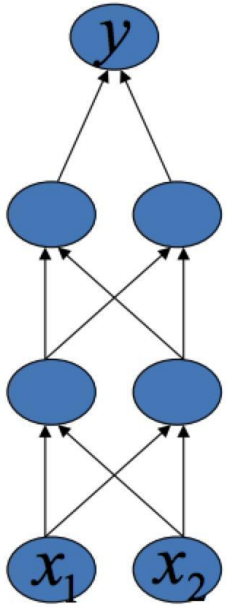
Decision Level

- 1 hidden layer
  - Boundary of convex region (open or closed)

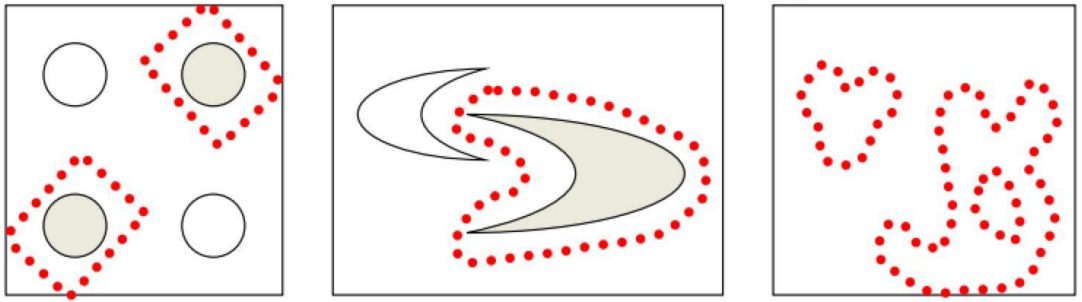


Example from to Eric Postma via Jason Eisner

Decision Level



- 2 hidden layers
  - Combinations of convex regions

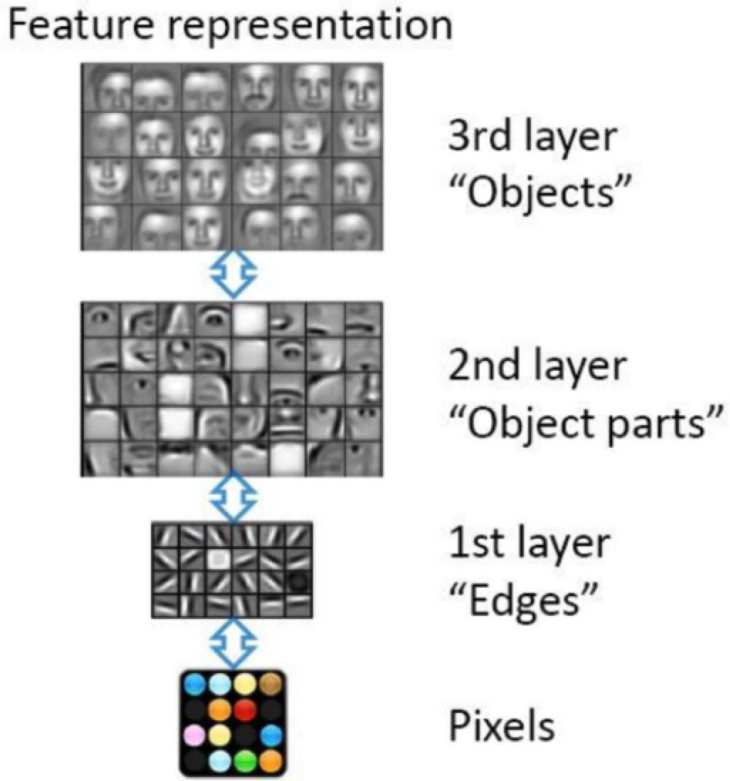


Example from to Eric Postma via Jason Eisner

Feature Level

Face Recognition:

- Deep Network can build up increasingly higher levels of abstraction
- Lines, parts, regions



Example from Honglak Lee (NIPS 2010)

Feed-forward Computation

1. Take f as the non-linear activation
2. Linear Transformation: $h = W_1 x$
3. 2-layer Neural Network: $h = W_2 f(W_1 x)$
4. 3-layer Neural Network: $h = W_3 f(W_2 f(W_1 x))$

- Neural Network is a model that recursively applies the matrix multiplication and non-linear activation function.

Why Multi-layers

Deep Residual Learning Network (achieved the first place in 2015 ILSVRC image classification) has 152 layers.

1. Even one single layer can approximate any function, the number of required nodes may be infinite.

2. It can be more efficient for multi-layer neural network compared to one-hidden-layer network

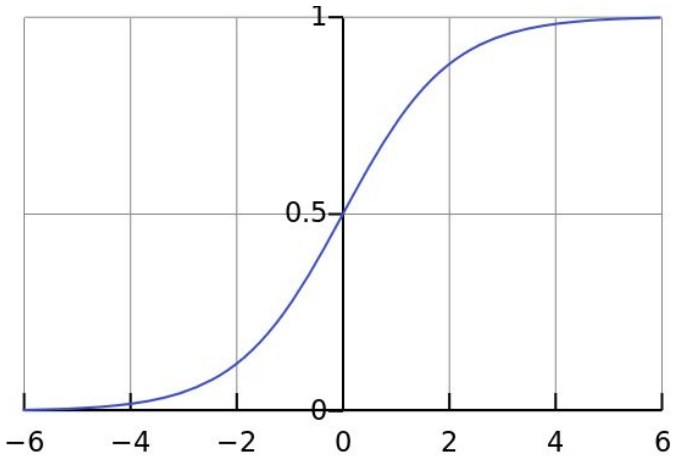
3. *Since a single sufficiently large hidden layer is adequate for approximation of most functions, why would anyone ever use more? One reason hangs on the words "sufficiently large". Although a single hidden layer is optimal for some functions, there are others for which a single-hidden-layer-solution is very inefficient compared to solutions with more layers.*

From the book 'neural smithing'

Non-linear Activation Function

1. Sigmoid Function:

$$f(z) = \frac{1}{1 + e^{-z}}$$

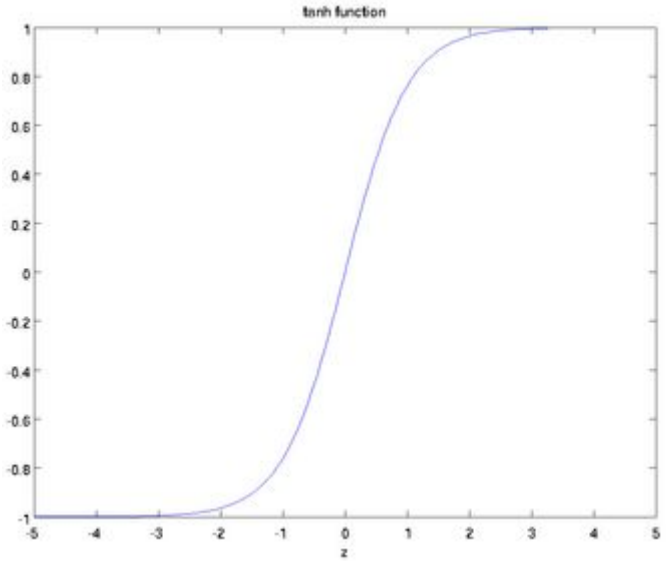


- Keep activation bound in range (0,1)
- Good for classification problem (clear distinctions)
- Smooth non-linear function
- Vanishing gradients (near –horizontal part of the curve)

Non-linear Activation Function

1. Tanh Function:

$$f(z) = \frac{2}{1 + e^{-2z}} - 1$$



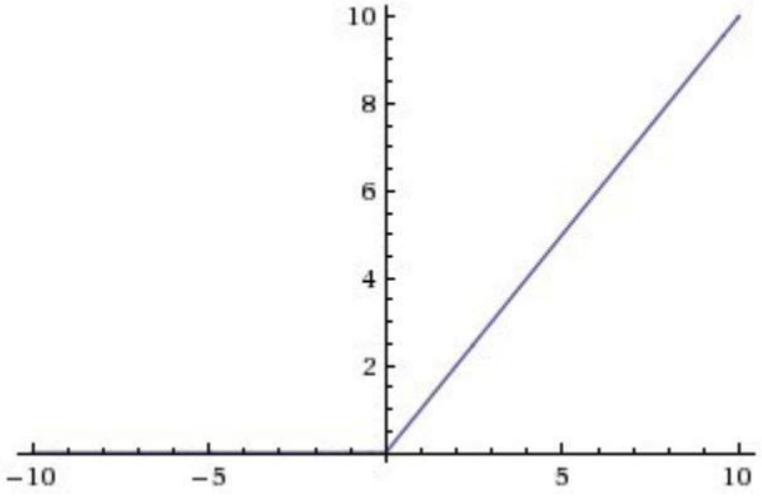
- Keep activation bound in range (-1,1)
- More steeper curve
- Smooth non-linear function
- Vanishing gradients (near –horizontal part of the curve)

Non-linear Activation Function

1. ReLu Function:

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

- Nonlinear Activation, but may blow up (gradient is one)
- Sparsity/ Efficient
- Less computational expensive that is good for deep structure
- Dying ReLu problem. (Several neurons will not respond)

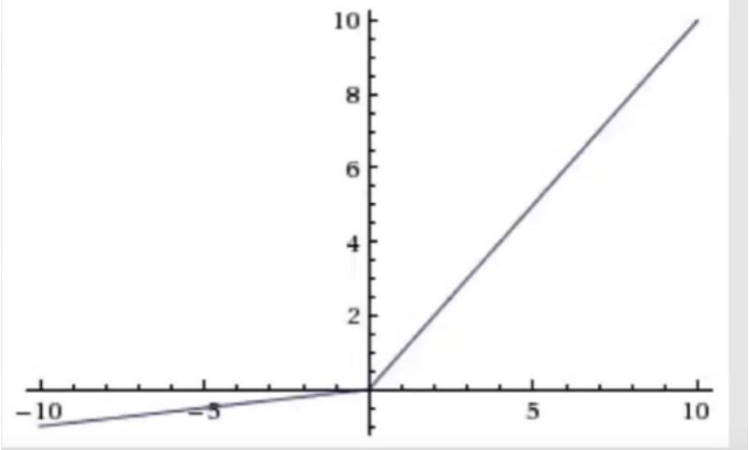


Non-linear Activation Function

1. Leaky ReLu Function:

$$f(z) = \mathbb{I}(x < 0)(ax) + \mathbb{I}(x > 0)(x)$$

- Proposed to solve the dying ReLu problem
- Still non-linear



Why Non-linear Activation

1. The non-linearities activation function increase the capacity of model
2. Without non-linearities, deep neural networks is meaningless.
    - Extra layers is just one linear transform:
3. How to select activation functions?

you can select an activation function which will approximate the function faster leading to faster training process.

one personal advice is to start with ReLU, which is a general approximator at most of the time.

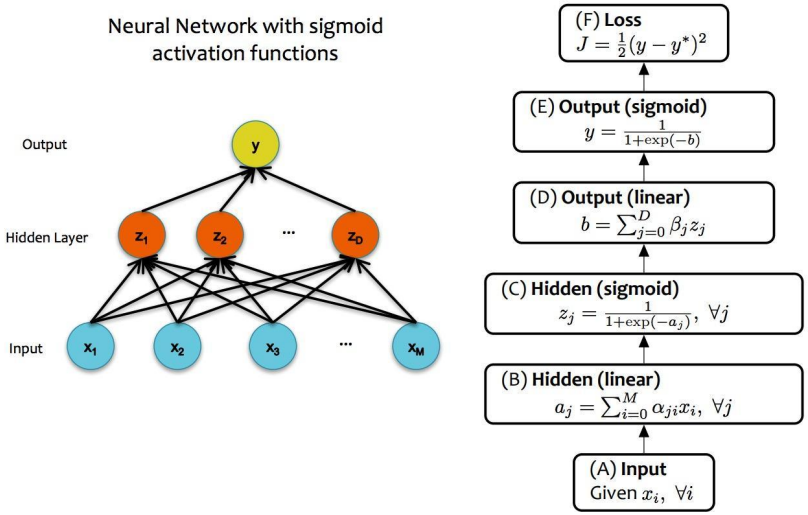
Loss Function

1. Regression: Mean-squared Error

$$J = \frac{1}{2}(y - y^*)^2$$

2. Classification: Cross-entropy Error (y* is the ground-truth label)

$$J = y^* \log y + (1 - y^*)\log(1 - y)$$



From Matt Gormley

Optimizer for Neural Network

Gradient Descent Algorithms

1. Core idea: minimize an objective function J(θ) by updating parameters θ in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters

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

1. Variants: how much of data that u will use to compute gradients
    - Stochastic gradient descent
    - Batch gradient descent
    - Mini-batch gradient descent

Stochastic Gradient Descent

- Given training data (X1, d1), (X2, d2), …, (XT, dT)
- Initialize all weights W1, W2, …, WK
- Do
    - For all t = 1: T
        - For every layer k:
            - Compute the gradient of the layer parameters
            - Update model parameters

$$W_k = W_k - \eta \nabla_{W_k} J(W_k)$$

Until loss value have converged

Batch SGD

Batch SGD

1 only update model parameters after all training data have been evaluated.

2 stable error gradient

3 need a large memory

4 may lead to a less optimal solution

Mini-batch SGD

Mini-batch SGD: split the dataset into small batches and take the average of the gradient over the batch and update the weights

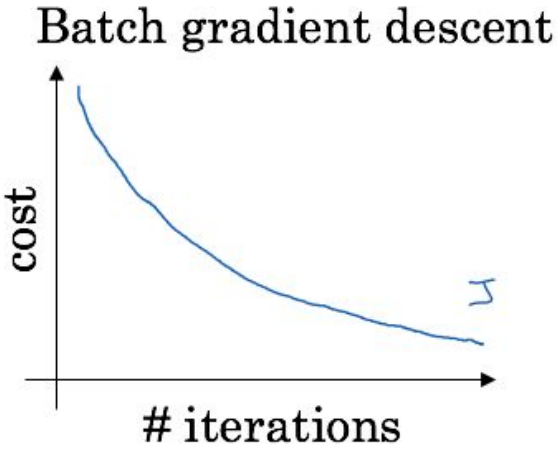
1 more efficient than SGD

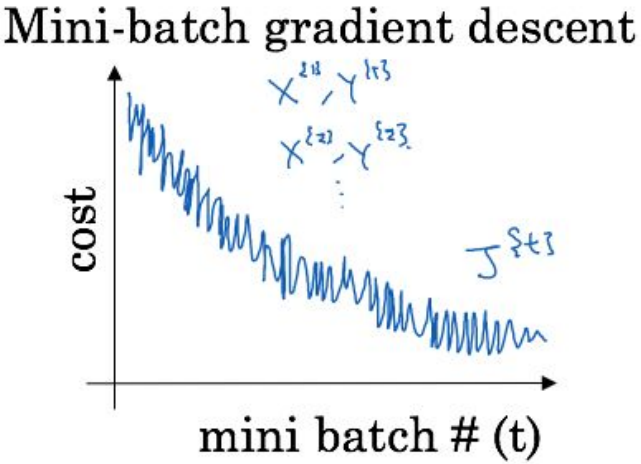
2 requires additional hyperparameter i.e. mini-batch size

3 hints on batch size:

* a power of two that fits the memory requirements of GPU or CPU.
* small -> a learning process that converges quickly at the cost of noise in the training
* large -> a learning process that converges slowly with accurate estimate the error gradient

Mini-batch SGD





How about SGD?

Challenges

1 how to select a proper learning rate

2 how to design a proper and data-dependent learning rate schedules

3 same learning rate works for all parameter updates

4 get trapped in various suboptimal local minima.

Momentum

Core idea: the current gradient computation will keep the direction as the previous gradient computation

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - v_t$$

1 help accelerate SGD

2 dampens oscillations

3 faster convergence

Adagrad

Core idea: different learning rates for different parameters. Smaller updates for parameters associated with frequently occurring features and larger updates for parameters associated with infrequent features.

$$\theta_{t+1;i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} v_{t,i}$$

where i is the parameter index. $G_{t,ii}$ is the sum of the squares of the gradients $v_{t,i}$ w.r.t. θ_i up to time step t

1 no need to manually tune the learning rate (start with 0.01)

2 good for sparse data

3 the learning rate shrinks all the time

RMSprop

Core idea: exponentially decaying average of squared gradients instead of sum.

$$E[v^2]_t = 0.9E[v^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1;i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[v^2]_t + \varepsilon}} v_{t,i}$$

1 resolve Adagrad's radically diminishing learning rates.

2 a good default value is 0.9

Adadelta

Core idea: 1 exponentially decaying average of squared gradients instead of sum.
2 use squared parameters updated to replace learning rate

$$E[\Delta\theta^2]_t = 0.9E[\Delta\theta^2]_{t-1} + 0.1\Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g^2]_t} v_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

1 no need to set default learning rate

2 similar to RMSprop

3 faster convergence

Adaptive Moment Estimation (Adam)

Core idea: 1 exponentially decaying average of squared gradients
2 exponentially decaying average of gradients

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1-\beta_1) g_t^2$$

Bias-corrected:

$$\widetilde{m_t} = \frac{m_t}{1 - \beta_1^t} \qquad \widetilde{v_t} = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widetilde{v_t} + \varepsilon}} \widetilde{m_t}$$

1 adaptive learning rates for different parameters

2 with momentum

3 practical suggestion: $\beta_1 = 0.9 \quad \beta_2 = 0.999 \quad \varepsilon = 10^{-8}$

How to select the optimizer

1 It is hard to find a general answer

2 sparse data -> try adaptive learning-rate methods

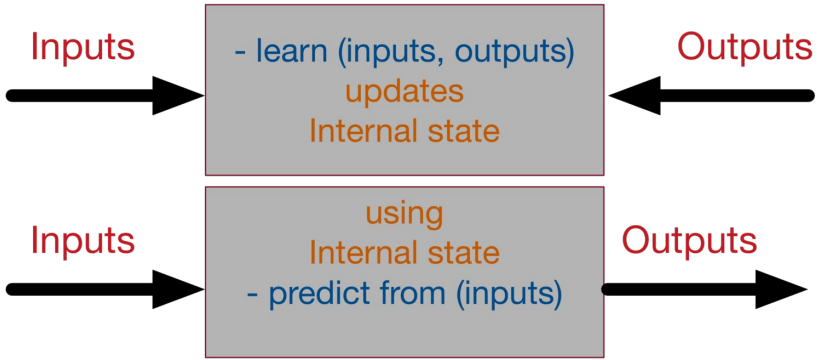
3 If u want to train a deep or complex neural networks with fast converge, do not just use SGD.

Back-Propagation

Neural Network WorkFlow

1. Given training data: $\{x_i, y_i\}^N$
2. Set decision function: $\hat{y} = f_\theta(x_i)$
3. Compute loss function: $\ell(\hat{y}, \mathrm{y_i})$
4. Define optimization goal: $\theta^* = argmin_\theta \sum_{i=1}^{N} \ell(f_\theta(x_i), \mathrm{y_i})$
5. Train with SGD: $\theta^{t+1} = \theta^t - \eta_t \nabla \ell(f_\theta(x_i), \mathrm{y_i})$

How to get the gradient for parameters: *Backpropagation*



Pic from Assaad

Numerical Differentiation

The method of finite differences: get two points (x, f(x) and (x+h, f(x+h)). Compute the slope

1. Pro: Great for testing implementations of backpropagation

2. Con: Slow of high dimensional inputs and outputs

3. Required: f(x) can be called on any input x

Given $f : \mathbb{R}^A \rightarrow \mathbb{R}^B, f(\mathbf{x})$

Compute $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

Symbolic Differentiation

The methods learned in calculus class

1. Pro: Derivatives are easily interpretable

2. Con: Required manual derivation and may leads to exponential computation time

3. Required: mathematical formulation that defines f(x) and some derivatives skills

Given $f : \mathbb{R}^A \rightarrow \mathbb{R}^B, f(\mathbf{x})$

Compute $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

Automatic Differentiation - Reverse Mode

Based on backpropagation

Given $f : \mathbb{R}^A \to \mathbb{R}^B, f(\mathbf{x})$

Compute $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

1. Pro: Computes partial derivatives of one input f(x) with respect to all inputs in reasonable time

2. Con: Slow of high dimensional inputs and outputs

3. Required: Algorithm for computing f(x)

```
from math import pi

def f(x):
  return tf.square(tf.sin(x))

assert f(pi/2).numpy() == 1.0


# grad_f will return a list of derivatives of f
# with respect to its arguments. Since f() has a single argument
# grad_f will return a list with a single element.
grad_f = tfe.gradients_function(f)
assert tf.abs(grad_f(pi/2)[0]).numpy() < 1e-7
```

Backpropagation

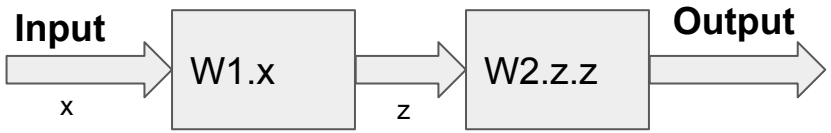
Chain Rule:
Deal with composition of functions:

If f(u): differential function of u and u=g(x) is a differential function of x, then
y=f(g(x)) is a differential function of x

Neural network can be regarded as the composition of Layers Computation.

Then, Backpropagation is repeated application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates

Toy Examples

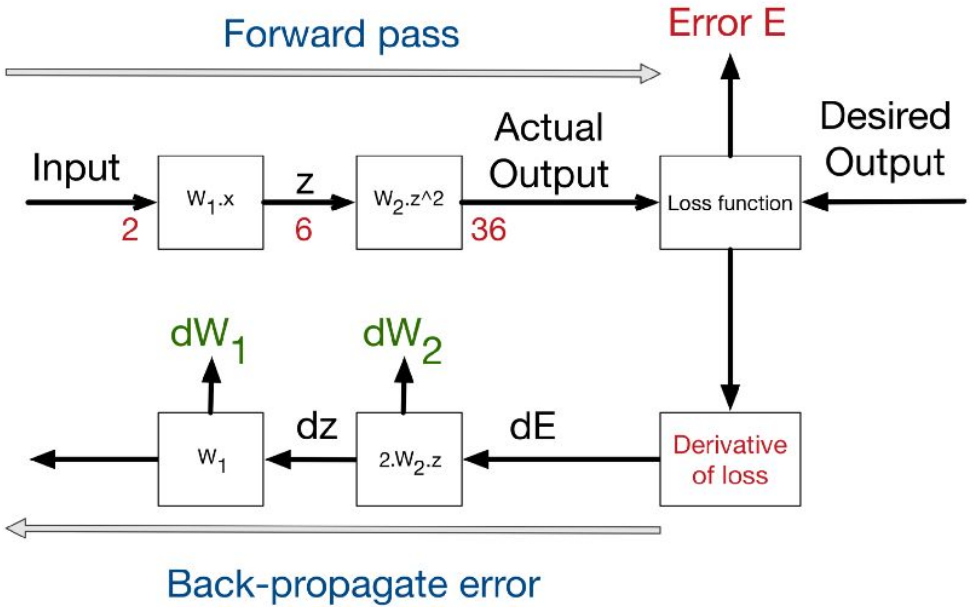


The above function can be regarded as a two-layers neural network:

Layer One: z = W1.x
Layer Two: W2 * z * z

Toy Examples

Input -> Forward Calls -> Loss Function -> Derivative -> Backpropagation of Errors -> Deltas on the weight of this stage



WorkFlow for Backpropagation

Forward Computation:

1 Write an algorithms for evaluation the function y=f(x). The algorithms defines a directed acyclic graph, where each variable is a node (i.e. the computation graph)

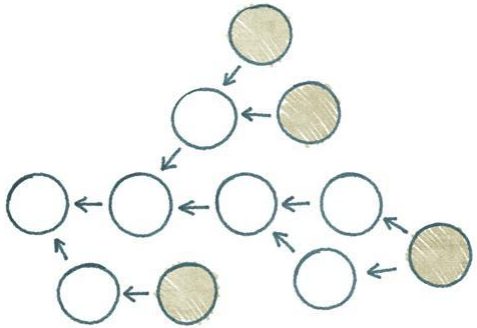
2 Visit each node in topological order

For variable u_i with inputs $v_1,\ldots,v_N$

a. compute $u_i=g_i(v_1,\ldots, v_n)$

b. store the result at the node

3 Keep a stack of function calls and their parameters during the forward pass



WorkFlow for Backpropagation

Backward Computation:

1 Initialize all partial derivative dy/du_j to 0 and $dy/dy=1$

2 Visit each node in reverse topological order

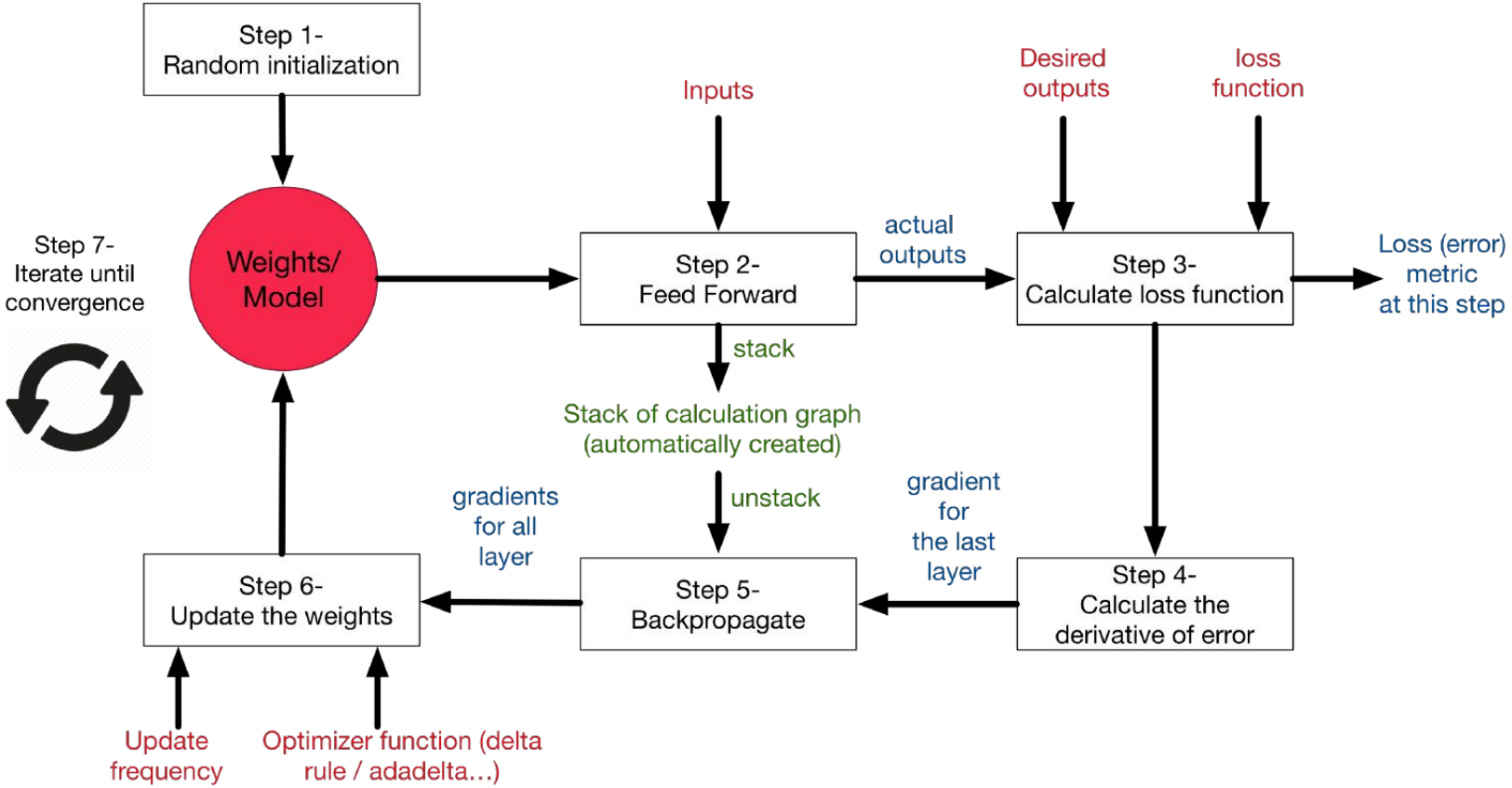
For variable $u_i=g_i(v_1,\ldots, v_n)$

a. we already know dy/du_i

b. Increment dy/dv_i by $(dy/du_i)(du_i/dv_j)$

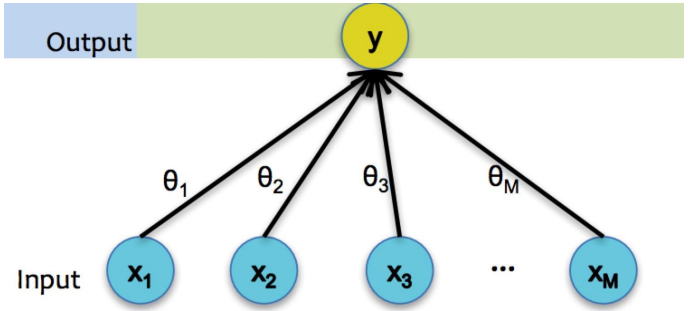
3 De-stacking through the function calls

Auto-Diff Scheme



Neural Network Frameworks

Logistic Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

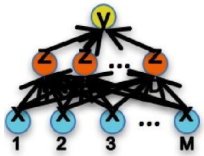
$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \frac{da}{d\theta_j} = x_j$$

$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \frac{da}{dx_j} = \theta_j$$

Two Layers Neural Network

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$$

Backpropagation

1. Backpropagation provides an efficient way to compute gradients

2. Is a special case of reverse-mode automatic differentiation

3. Deep learning frameworks such as tensorflow, pytorch all adopt this kind of automatic differentiation.

Neural Network Initialization

Initialization

1. Optimization for neural network in nature is a iterative method, which requires initialization.

2. Some general rules for initialization of model parameters:

    1. Sometimes all 0 may be a bad idea (saddle point with tanh function),
    2. Can not initialize all weights to the same value
    3. Randomness should be incorporated
    4. Bias terms can be safely set to be zero (only depend on the linear activation of that layer)

Normal Initialization

1. Initialize weights randomly, following standard normal distribution .

2. Two potential issues:
    1. Vanishing gradients
    2. Exploding gradients: oscillating around the minima or number overflow

3. Best Practices: initialization should work with activation functions

Some Hints

1. For ReLU:

$$\sqrt{\frac{2}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(2/size\_l\text{-}1)$$

2. For Tanh: Xavier initialization

$$\sqrt{\frac{1}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(1/size\_l\text{-}1)$$

3. Another Good Practice:

$$\sqrt{\frac{2}{size^{[l-1]} + size^{[l]}}}$$

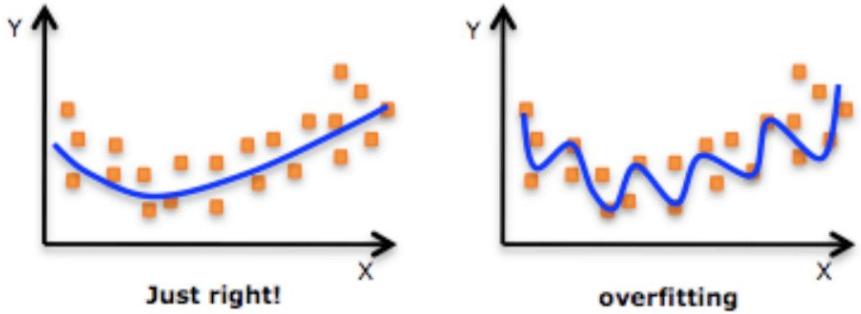
$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(2 / (size\_l\text{-}1 + size\_l))$$

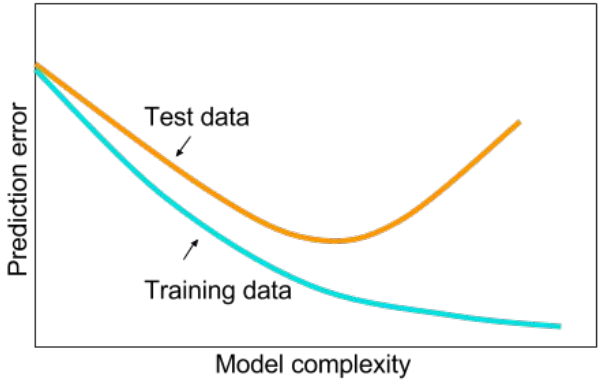
The gradients will not be vanished or exploded. It can avoid slow convergence and ensure that we do not keep oscillating off the minima.

Overfitting and Preventing Tips

Overfitting

1. Overfitting refers to a model that models the training data too well, which negatively impact the models ability to generalize





Neural Network with a deep structure easily get overfitted

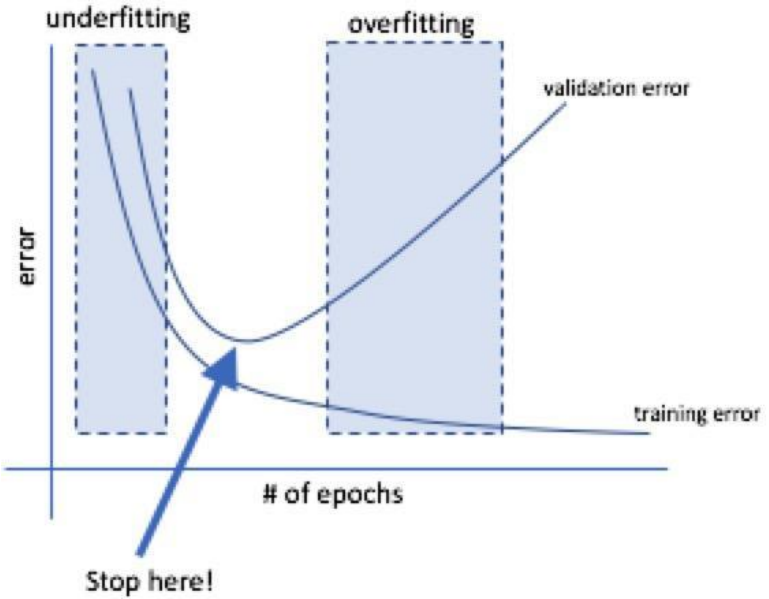
Overfitting for NN

Neural Network with a deep structure easily get overfitted.

1. Early Stopping
2. Parameters Regularization
3. Dropout
4. Train with more data or remove features

Early Stopping

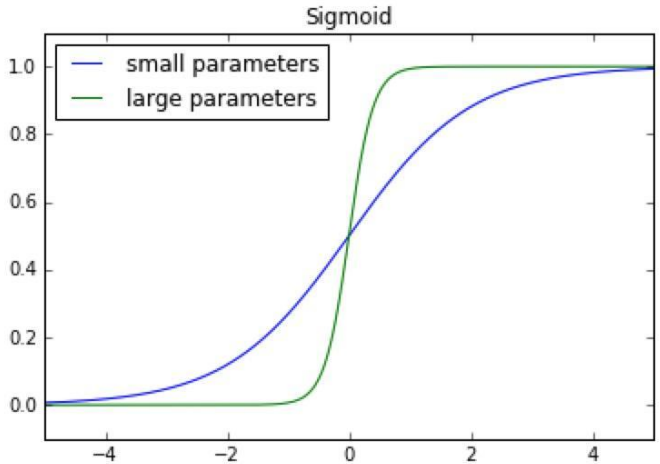
1. Watch the validation curve
2. Stop updating the weights once validation error starts increasing



Regularization

1. In NN, inputs are linearly combined with parameters. Therefore, large parameters can amplify small changes in the input.

2. Large parameters may arbitrarily increases the confidence in our predictions.



3. In cost function, add regularization term to penalize parameters

Control the degree to which we select to penalize large parameters

4. The core idea is to prevent parameters from becoming excessively large during training $J = \frac{1}{N}\sum_{i=1}^{N} \ell(f_\theta(x_i), y_i) + \lambda g(\theta)$

L1 Norm

1. L1 Regularization:

$$\lambda \sum_{i=1}^{k} |\theta_i|$$

This expression is commonly used for feature selection as it tends to produce sparse parameter vectors where only the important features take on non-zero values

L2 Norm

1. L2 Regularization:

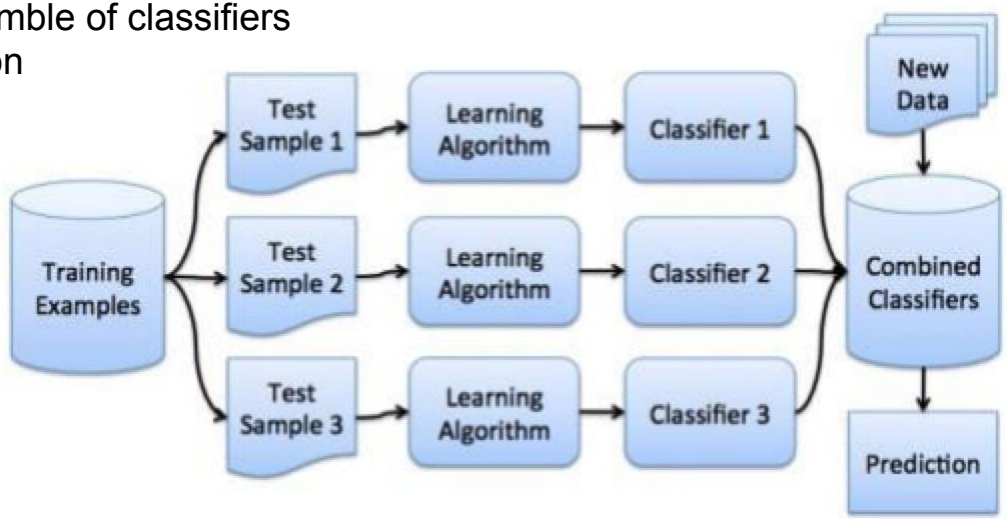
$$\lambda \sum_{i=1}^{k} \theta_i^2$$

This expression does not tend to push less important weights to zero and typically produces better results when training a model.

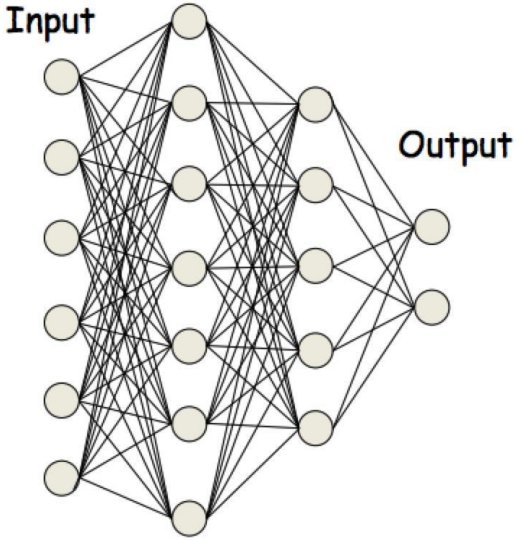
Dropout

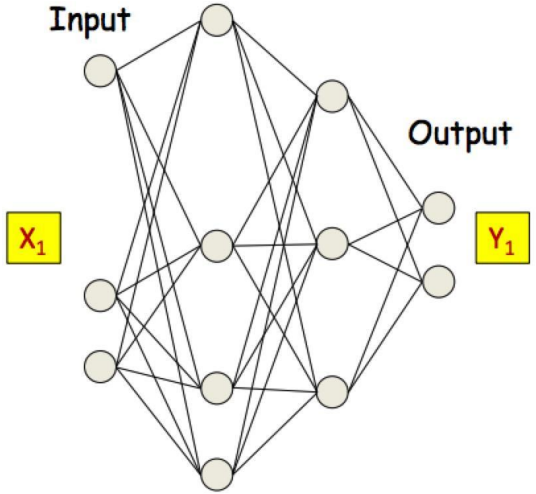
1. Very simple way to prevent overfitting

2. The ideas comes from Bagging:
   - Sample training data and train several different classifiers
   - Classify test instance with entire ensemble of classifiers
   - Vote across classifiers for final decision



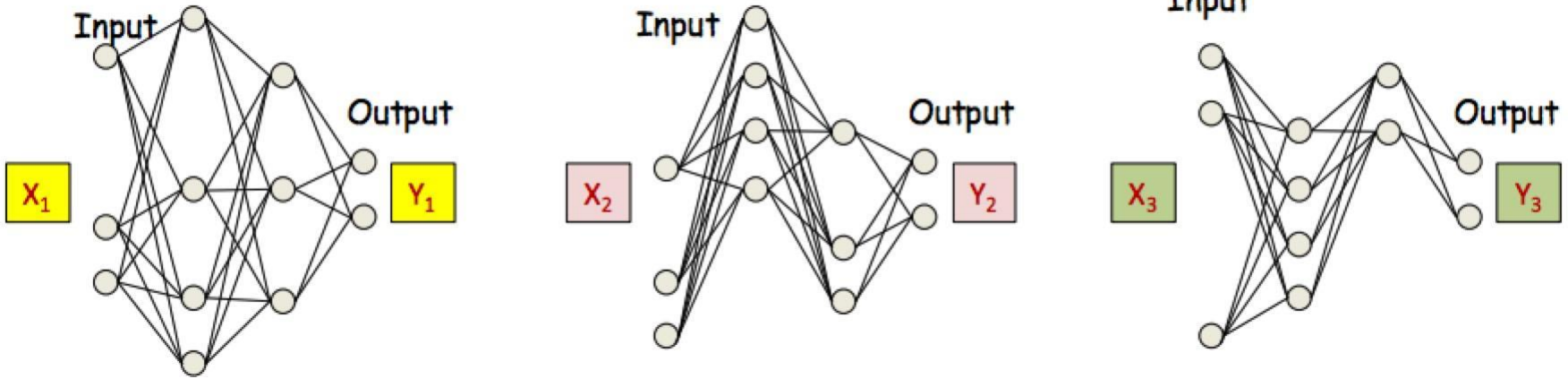
Dropout





- During training: for each input, at each iteration, “turn off” each neuron with a probability 1-α

Dropout



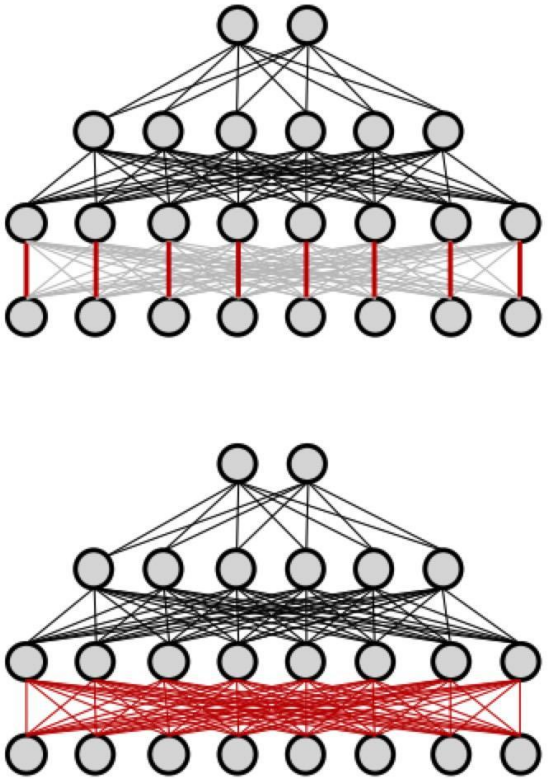
- Due to randomness, the drop nodes changes for each input
- BP is effectively performed only over the remaining network
- For closed nodes, the gradients are zero

Dropout

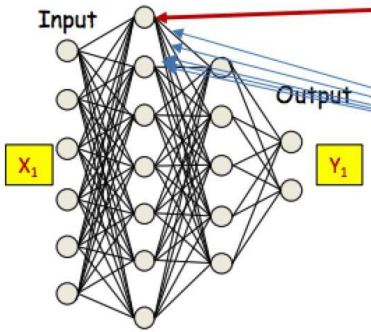
1. Dropout forces the neuros to learn “rich” and redundant pattern

2. Without dropout, a layer may just clone its input to output.

3. Dropout forces the neuros to learn denser pattern



Dropout During Test



apply α here (to the output of the neuron) OR..

Push the α to all outgoing weights

$$y_i^{(k)} = \alpha\sigma\left(z_i^{(k)}\right)$$

$$\begin{aligned} z_i^{(k)} &= \sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)} \\ &= \sum_j w_{ji}^{(k)} \alpha\sigma\left(z_j^{(k-1)}\right) + b_i^{(k)} \\ &= \sum_j \left(\alpha w_{ji}^{(k)}\right) \sigma\left(z_j^{(k-1)}\right) + b_i^{(k)} \end{aligned}$$

$$W_{test} = W_{train}\alpha,$$

Kind of average voting

Dropout Effects

Experimental Studies on MINIST dataset:

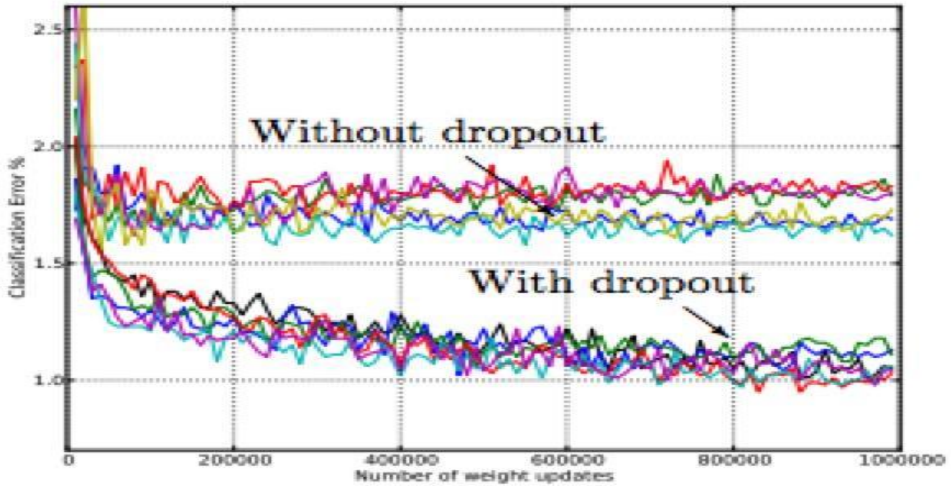


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

General Tricks

Train with more data

Remove irrelevant features

If training data is complete and match the distribution of testing data, overfitting is no more a disaster.

Summary

1. Neural network is the recursive application of linear transformation and non-linear activation function.
2. Non-linear activation function is important
3. Gradient descent algorithms and various variants for optimization
4. Backpropagation for gradient computing in Neural Network
5. Auto-diff techniques are widely used in deep learning frameworks
6. Several regularization techniques to overcome overfitting