Sapienza University of Rome

Master in Artificial Intelligence and Robotics
Master in Engineering in Computer Science

Machine Learning

A.Y. 2020/2021

Prof. L. Iocchi, F. Patrizi

11. Artificial Neural Networks

L. Iocchi, F. Patrizi

with contributions from Valsamis Ntouskos

Overview

- Feedforward networks
- Architecture design
- Cost functions
- Activation functions
- Gradient computation (back-propagation)
- Learning (stochastic gradient descent)
- Regularization

References
Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning - Chapters 6, 7, 8. `http://www.deeplearningbook.org`

Artificial Neural Networks (ANN)

Alternative names:

- Neural Networks - (NN)
- Feedforward Neural Networks - (FNN)
- Multilayer Perceptrons - (MLP)

Function approximator using a parametric model.

Suitable for tasks described as associating a vector to another vector.

Artificial Neural Networks (ANN)

Goal:

Estimate some function $f : X \rightarrow Y$, with $Y = \{C_1, \ldots, C_k\}$ or $Y = \Re$

Data:

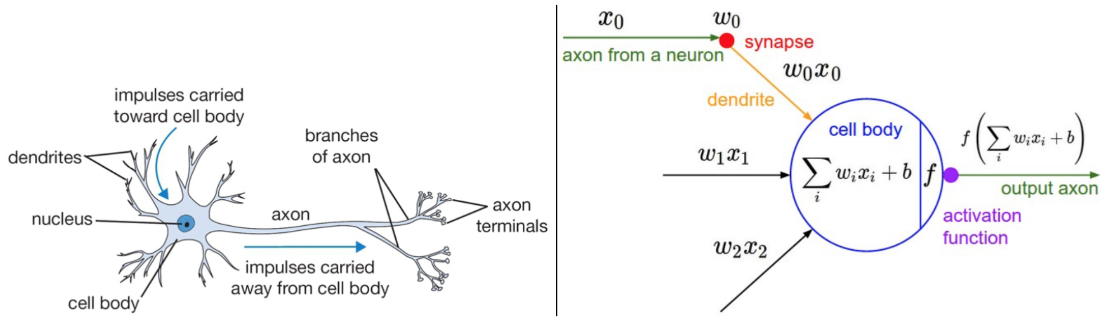
$D = \{(\mathbf{x}_n, t_n)_{n=1}^{N}\}$ such that $t_n \approx f(\mathbf{x}_n)$

Framework:

Define $y = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ and learn parameters $\boldsymbol{\theta}$ so that $\hat{f}$ approximates f.

Feedforward Networks

Draw inspiration from brain structures



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

Image from Isaac Changhau `https://isaacchanghau.github.io`

Hidden layer output can be seen as an array of **unit** (neuron) activations based on the connections with the previous units

Note: Only use some insights, they are not a model of the brain!

Feedforward Networks - Terminology

Feedforward information flows from input to output without any loop

Networks f is a composition of elementary functions in an acyclic graph

Example:

$$f(\mathbf{x};\boldsymbol{\theta}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x};\boldsymbol{\theta}^{(1)});\boldsymbol{\theta}^{(2)});\boldsymbol{\theta}^{(3)})$$

where:

$f^{(m)}$ the m-th layer of the network

and

$\boldsymbol{\theta}^{(m)}$ the corresponding parameters

Feedforward Networks - Terminology

FNNs are **chain structures**

The length of the chain is the **depth** of the network

Final layer also called **output layer**

Deep learning follows from the use of networks with a large number of layers (large depth)

Feedforward Networks

Why FNNs?

Linear models cannot model interaction between input variables

Kernel methods require the choice of suitable kernels

- use generic kernels e.g. RBF, polynomial, etc. (convex problem)
- use hand-crafted kernels - application specific (convex problem)

FNN leaning:
complex combination of many parametric functions (non-convex problem)

XOR Example - Linear model

Learning the XOR function - 2D input and 1D output

Dataset: $D = \{((0,0)^T, 0), ((0,1)^T, 1), ((1,0)^T, 1), ((1,1)^T, 0)\}$

Using linear regression with Mean Squared Error (MSE)

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(t_n - y(\mathbf{x}_n))^2$$

with $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$

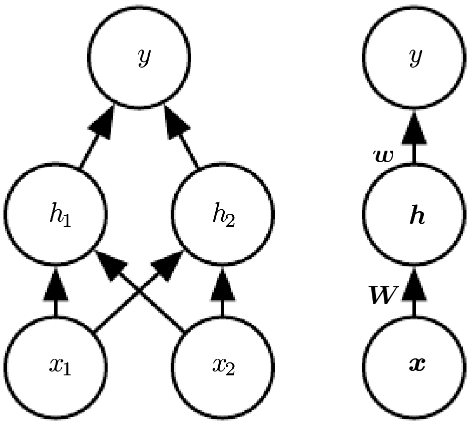
Optimal solution:

$$\mathbf{w} = 0 \text{ and } w_0 = \tfrac{1}{2}\text{, hence } y = 0.5 \text{ everywhere!}$$

Reason: No linear separator can explain the non-linear XOR function

XOR Example - FNN

Specify a two layers network:



XOR Example - FNN

Hidden units:

$$\mathbf{h} = g(\mathbf{W}^T \mathbf{x} + \mathbf{c})$$

with $g(\alpha) = \max(0, \alpha)$

Output:

$$y = \mathbf{w}^T \mathbf{h} + b$$

Full model:

$$y(\mathbf{x}) = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{w}^T \max(0, \mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$

with $\boldsymbol{\theta} = \langle \mathbf{W}, \mathbf{c}, \mathbf{w}, b \rangle$

Note: non-linear model in $\boldsymbol{\theta}$

XOR Example - FNN

Model:

$$y(\mathbf{x}) = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{w}^T \max(0, \mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$

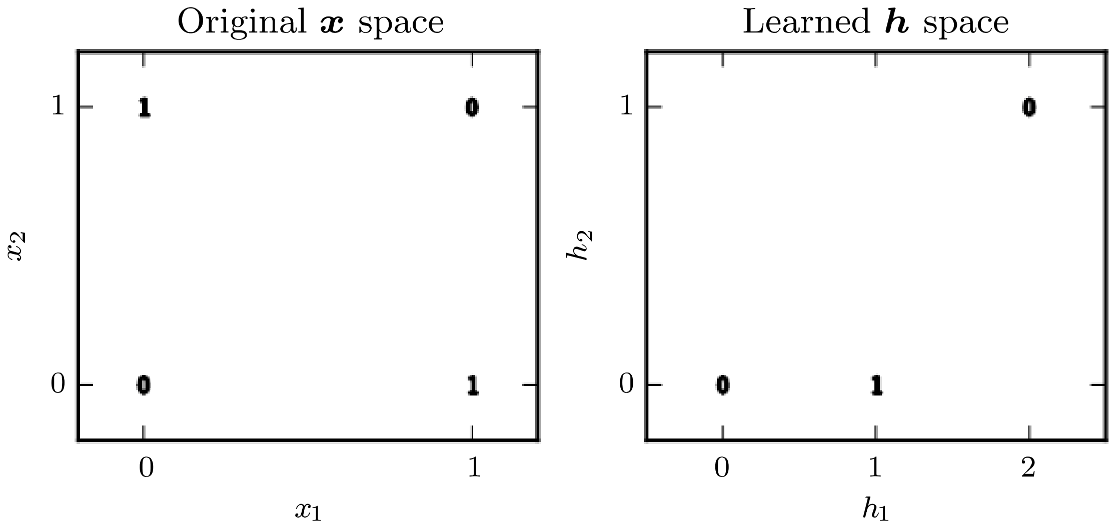
Mean Squared Error (MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(\mathbf{x}_n))^2$$

Solution:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

XOR Example - FNN



Architecture design

Overall structure of the network

How many hidden layers? **Depth**

How many units in each layer? **Width**

Which kind of units? **Activation functions**

Which kind of cost function? **Loss function**

Architecture design

How many hidden layers? **Depth**

Universal approximation theorem: a FFN with a linear output layer and at least one hidden layer with any “squashing” activation function (e.g., sigmoid) can approximate any Borel measurable function with any desired amount of error, provided that enough hidden units are used.

It works also for other activation functions (e.g., ReLU)

Architecture design

How many units in each layer? **Width**

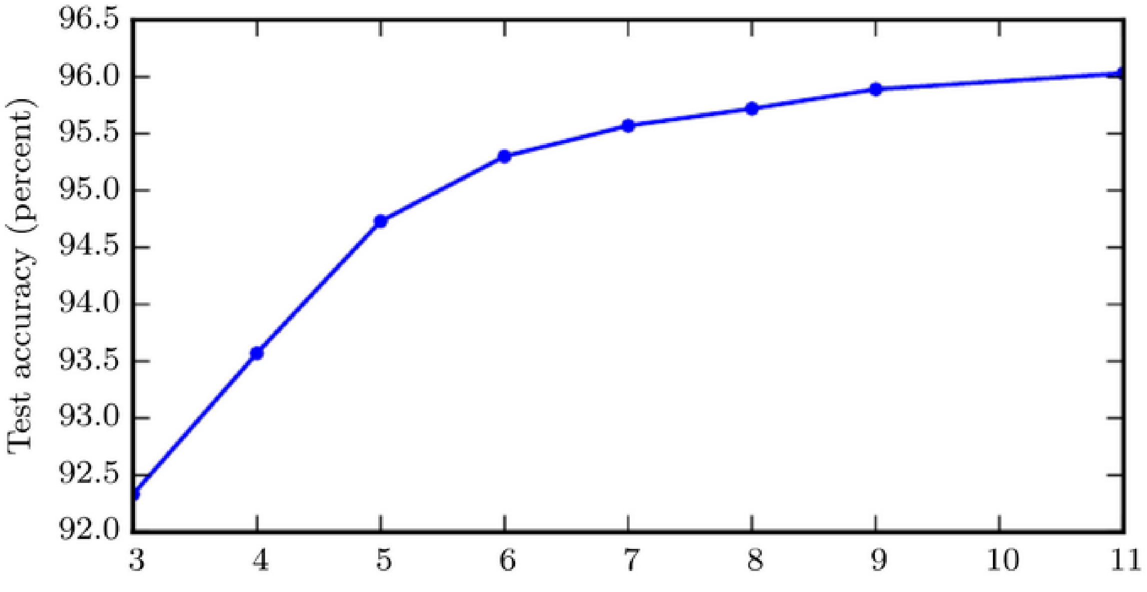
Universal approximation theorem does not say how many units.

In general it is exponential in the size of the input.

In theory, a short and very wide network can approximate any function.

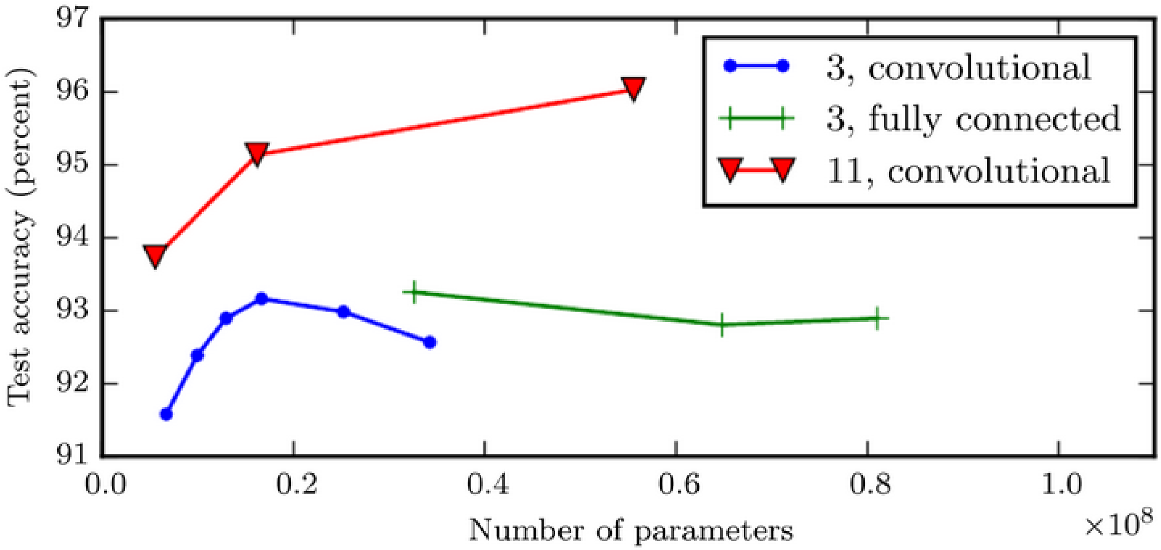
In practice, a deep and narrow network is easier to train and provides better results in generalization.

Architecture design



Depth

Architecture design



Architecture design

Which kind of units? **Activation functions**

Which kind of cost function? **Loss function**

Gradient-based learning remarks

- Unit saturation can hinder learning
- When units saturate gradient becomes very small
- Suitable cost functions and unit nonlinearities help to avoid saturation

Cost function

Model implicitly defines a conditional distribution $p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta})$

Cost function: Maximum likelihood principle (**cross-entropy**)

$$J(\boldsymbol{\theta}) = E_{\mathbf{x},\mathbf{t}\sim\mathcal{D}}\left[-\ln(p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}))\right]$$

Example:
Assuming additive Gaussian noise we have

$$p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}) = \mathcal{N}(\mathbf{t}|f(\mathbf{x};\boldsymbol{\theta}),\beta^{-1}I)$$

and hence

$$J(\boldsymbol{\theta}) = E_{\mathbf{x},\mathbf{t}\sim\mathcal{D}}\left[\frac{1}{2}||\mathbf{t}-f(\mathbf{x};\boldsymbol{\theta})||^2\right]$$

Maximum likelihood estimation with Gaussian noise corresponds to mean squared error minimization.

Output units activation functions

Let $\mathbf{h} = f(\mathbf{x};\boldsymbol{\theta}^{(n-1)})$ the output of the hidden layers,

which output model $y = f^{(n)}(\mathbf{h};\boldsymbol{\theta}^{(n)})$?

which cost function $J(\boldsymbol{\theta})$?

Choice of network output units and cost function are related.

- Regression
- Binary classification
- Multi-classes classification

Output units activation functions

Regression

Linear units: Identity activation function

$$y = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

Use a Gaussian distribution noise model

$$p(t|\mathbf{x}) = \mathcal{N}(t|y, \beta^{-1})$$

Cost function: maximum likelihood (cross-entropy) that is equivalent to minimizing **mean squared error**.

Note: linear units do not saturate

Output units activation functions

Binary classification

Sigmoid units: Sigmoid activation function $y = \sigma(\mathbf{w}^T \mathbf{h} + b)$

The likelihood corresponds to a Bernoulli distribution

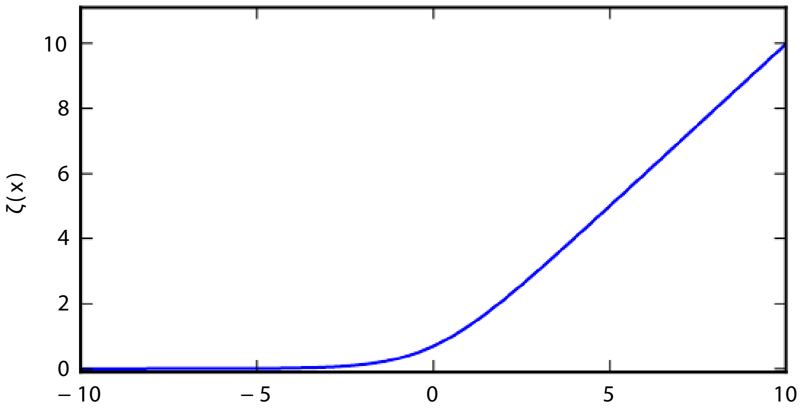
$$J(\boldsymbol{\theta}) = E_{\mathbf{x},t \sim \mathcal{D}} \left[- \ln p(t|\mathbf{x}) \right]$$

$$\begin{aligned} - \ln p(t|\mathbf{x}) &= - \ln \sigma(\alpha)^t (1 - \sigma(\alpha))^{1-t} \\ &= - \ln \sigma((2t-1)\alpha) \\ &= \mathtt{softplus}((1-2t)\alpha) \end{aligned}$$

with $\alpha = \mathbf{w}^T \mathbf{h} + b$.

Note: Unit saturates only when it gives the correct answer.

Output units activation functions



The softplus function

Output units activation functions

Multi-class classification

Softmax units: Softmax activation function

$$y_i = \texttt{softmax}(\alpha^{(i)}) = \frac{\exp(\alpha^{(i)})}{\sum_j \exp(\alpha_j)}$$

Likelihood corresponds to a Multinomial distribution

$$J_i(\boldsymbol{\theta}) = E_{\mathbf{x},\mathbf{t}\sim\mathcal{D}}\left[-\ln \texttt{softmax}(\alpha^{(i)})\right]$$

with $\alpha^{(i)} = \mathbf{w}_i^T \mathbf{h} + b_i$.

Note: Unit saturates only when there are minimal errors.

Output units activation functions

Summary

Regression: **linear** output unit, **mean squared error** loss function

Binary classification: **sigmoid** output unit, **binary cross-entropy**

Multi-class classification: **softmax** output unit, **categorical cross-entropy**

Note: on-going research on different loss functions.

Hidden units activation functions

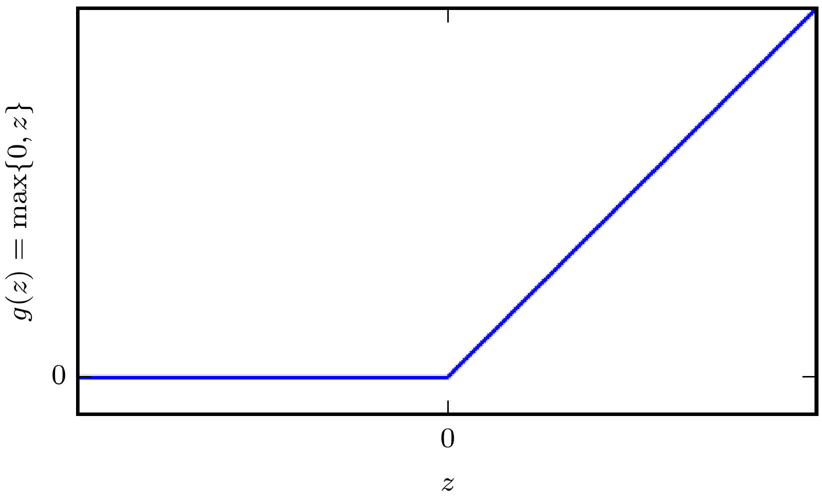
Many choices, some intuitions, no theoretical principles.
Predicting which activation function will work best is usually impossible.

Rectified Linear Units (ReLU):

$$g(\alpha) = \max(0, \alpha).$$

- Easy to optimize - similar to linear units
- Not differentiable at 0 - does not cause problems in practice

Hidden unit activation functions



Hidden unit activation functions

Sigmod and hyperbolic tangent:

$$g(\alpha) = \sigma(\alpha)$$

and

$$g(\alpha) = \tanh(\alpha)$$

Closely related as $\tanh(\alpha) = 2\sigma(2\alpha) - 1$.

Remarks:

- No logarithm at the output, the units saturate easily.
- Gradient based learning is very slow.
- Hyperbolic tangent gives larger gradients with respect to the sigmoid.
- Useful in other contexts (e.g., recurrent networks, autoencoders).

Activation functions overview

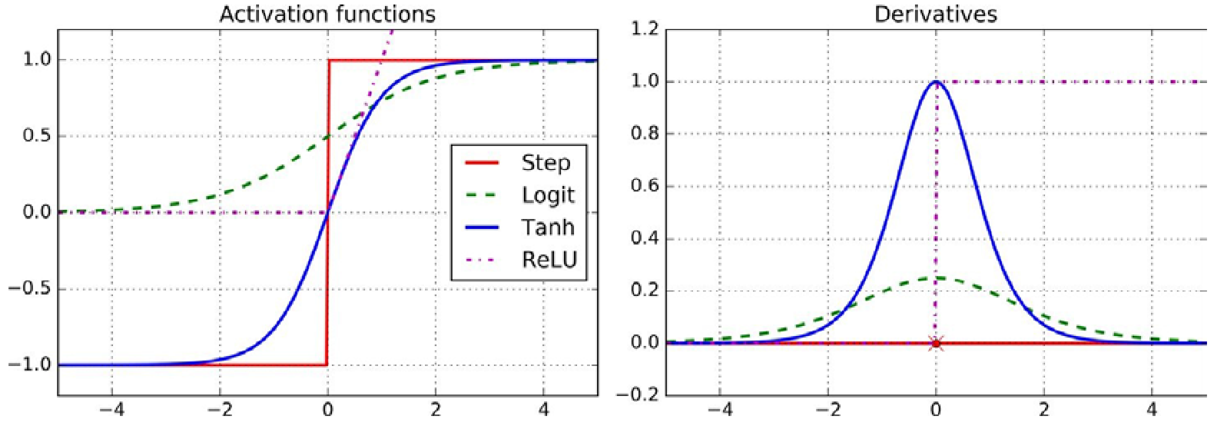


Image from Geron A. "Hands-On Machine Learning with Scikit-Learn and TensorFlow", O'Reilly 2017

Gradient Computation

Information flows forward through the network when computing network output y from input $\mathbf{x}$

To train the network we need to compute the gradients with respect to the network parameters $\boldsymbol{\theta}$

The **back-propagation** or **backprop** algorithm is used to propagate gradient computation from the cost through the whole network

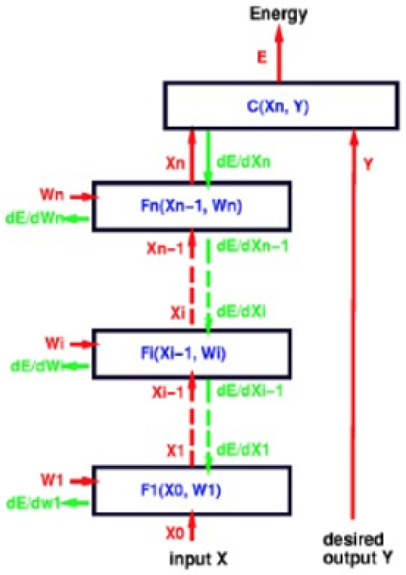


Image by Y. LeCun

Gradient Computation

Goal: Compute the gradient of the cost function w.r.t. the parameters

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Analytic computation of the gradient is straightforward

- simple application of the chain rule
- numerical evaluation can be expensive

Back-propagation is *simple* and *efficient*.

Remarks:

- back-propagation is only used to compute the gradients
- back-propagation is **not** a training algorithm
- back-propagation is **not** specific to FNNs

Chain rule

Let: $y = g(x)$ and $z = f(g(x)) = f(y)$
Applying the chain rule we have:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

For vector functions, $g : \mathbb{R}^m \mapsto \mathbb{R}^n$ and $f : \mathbb{R}^n \mapsto \mathbb{R}$ we have:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i},$$

equivalently in vector notation:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z,$$

with $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ the $n \times m$ Jacobian matrix of g.

Back-propagation algorithm

Forward step

```
Require: Network depth $l$
Require: $W^{(i)}, i \in \{1, \dots, l\}$ weight matrices
Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$ bias parameters
Require: $\mathbf{x}$ input value
Require: $\mathbf{t}$ target value
  $h^{(0)} = \mathbf{x}$
  for $k = 1, \dots, l$ do
    $\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
    $\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
  end for
  $\mathbf{y} = \mathbf{h}^{(l)}$
  $J = L(\mathbf{t}, \mathbf{y})$
```

Back-propagation algorithm

Backward step

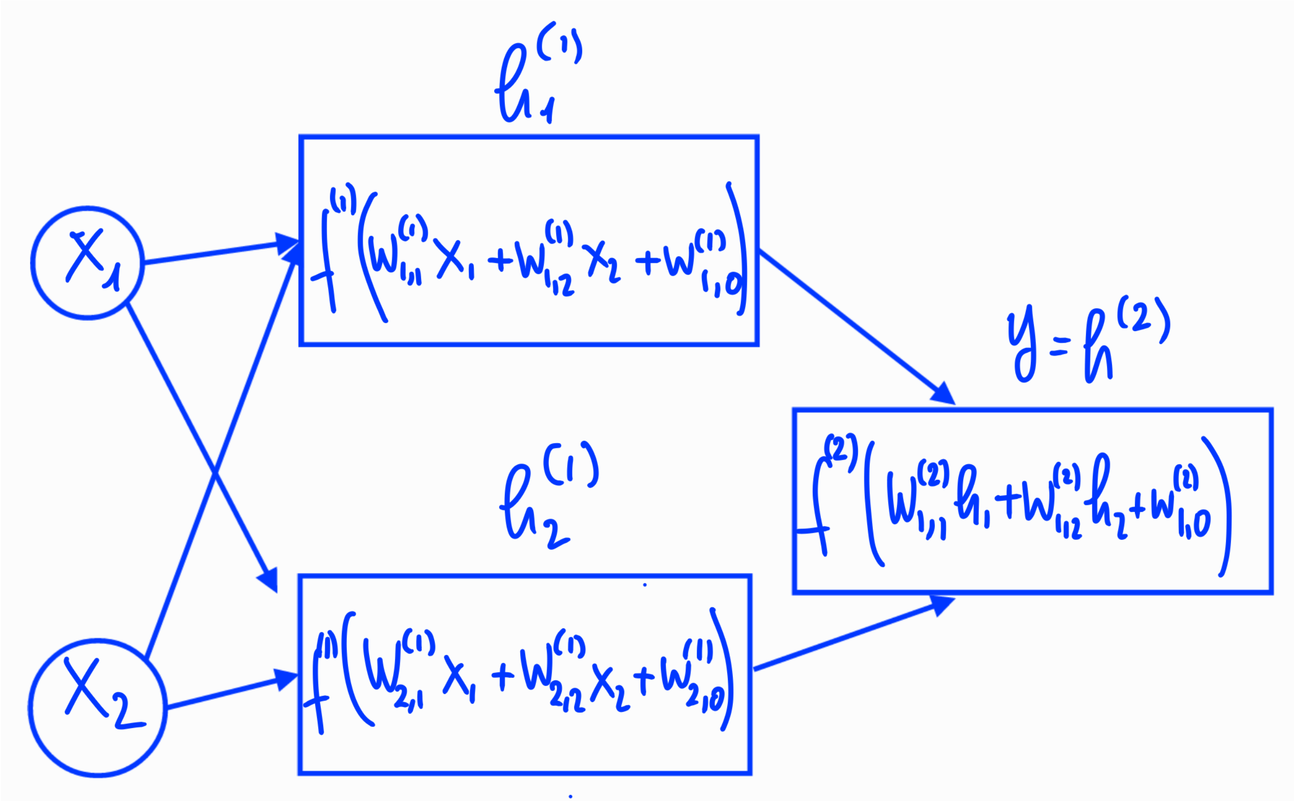
```
  $\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$
  for $k = l, l-1, \dots, 1$ do
    Propagate gradients to the pre-nonlinearity activations:
    $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}
    $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$
    $\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$
    Propagate gradients to the next lower-level hidden layer:
    $\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$
  end for
```

Back-propagation algorithm

Remarks:

- The previous version of backprop is specific for fully connected MLPs
- More general versions for acyclic graphs exist
- Dynamic programming is used to avoid doing the same computations multiple times
- Gradients can be computed either in symbolic or numerical form

Example of BackProp


$h_1^{(1)}$
$f^{(1)}(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + w_{1,0}^{(1)})$
x_1
$y = h^{(2)}$
$f^{(2)}(w_{1,1}^{(2)} h_1 + w_{1,2}^{(2)} h_2 + w_{1,0}^{(2)})$
$h_2^{(1)}$
x_2
$f^{(1)}(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{2,0}^{(1)})$

Example of BackProp

$\alpha_i^{(1)} = w_{i,0}^{(1)} + w_{i,1}^{(1)} x_1 + w_{i,2}^{(1)} x_2 \quad i = 1,2$

$h_i^{(1)} = f^{(1)}(\alpha_i^{(1)}) = \texttt{ReLU}(\alpha_i^{(1)}) \quad i = 1,2$

$\alpha^{(2)} = w_{1,0}^{(2)} + w_{1,1}^{(1)} h_1^{(1)} + w_{1,2}^{(1)} h_2^{(1)}$

$h^{(2)} = f^{(2)}(\alpha^{(2)}) = \alpha^{(2)}$

$y = h^{(2)}$

Loss function MSE

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

$\boldsymbol{\theta} = \langle w_{1,0}^{(1)}, w_{1,1}^{(1)}, w_{1,2}^{(1)}, w_{2,0}^{(1)}, w_{2,1}^{(1)}, w_{2,2}^{(1)}, w_{1,0}^{(2)}, w_{1,1}^{(2)}, w_{1,2}^{(2)} \rangle$

Example of BackProp

Forward step

Given $x_1, x_2, w_{i,j}^{(k)}, t$

compute $\alpha_1^{(1)}, \alpha_2^{(1)}, \alpha^{(2)}, h_1^{(1)}, h_2^{(1)}, h^{(2)}, y, J = L(t,y)$

Backward step

Given $x_1, x_2, w_{i,j}^{(k)}, t, \alpha_1^{(1)}, \alpha_2^{(1)}, \alpha^{(2)}, h_1^{(1)}, h_2^{(1)}, h^{(2)}, y, J = L(t,y)$

compute $\frac{\partial J}{\partial w_{i,j}^{(k)}}$

Example of BackProp

Gradient computation

$$\frac{\partial J(\boldsymbol{\theta})}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(t-y)^2 = y - t$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial w^{(2)}_{i,j}} = \frac{\partial J(\boldsymbol{\theta})}{\partial y}\frac{\partial y}{\partial w^{(2)}_{i,j}} \quad \text{with} \quad \frac{\partial y}{\partial w^{(2)}_{1,0}} = 1, \quad \frac{\partial y}{\partial w^{(2)}_{1,1}} = h^{(1)}_1 \quad \frac{\partial y}{\partial w^{(2)}_{1,2}} = h^{(1)}_2$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial h^{(1)}_i} = \frac{\partial J(\boldsymbol{\theta})}{\partial y}\frac{\partial y}{\partial h^{(1)}_i} \quad \text{with} \quad \frac{\partial y}{\partial h^{(1)}_1} = w^{(2)}_{1,1} \quad \frac{\partial y}{\partial h^{(1)}_2} = w^{(2)}_{1,2}$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \alpha^{(1)}_i} = \frac{\partial J(\boldsymbol{\theta})}{\partial h^{(1)}_i}\frac{\partial h^{(1)}_i}{\partial \alpha^{(1)}_i} = \frac{\partial J(\boldsymbol{\theta})}{\partial h_i}\,\texttt{step}(\alpha^{(1)}_i)$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial w^{(1)}_{i,j}} = \frac{\partial J(\boldsymbol{\theta})}{\partial \alpha^{(1)}_i}\frac{\partial \alpha^{(1)}_i}{\partial w^{(1)}_{i,j}} \quad \text{with} \quad \frac{\partial \alpha^{(1)}_i}{\partial w^{(1)}_{i,0}} = 1, \quad \frac{\partial \alpha^{(1)}_i}{\partial w^{(1)}_{i,1}} = x_1 \quad \frac{\partial y}{\partial w^{(1)}_{i,2}} = x_2$$

Example of BackProp (compact notation)

Gradient computation (in vector notation)

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mathbf{W}^{(1)} = \begin{bmatrix} w^{(1)}_{1,1} w^{(1)}_{1,2} \\ w^{(1)}_{2,1} w^{(1)}_{2,2} \end{bmatrix}, \ \mathbf{b}^{(1)} = \begin{bmatrix} w^{(1)}_{1,0} \\ w^{(1)}_{2,0} \end{bmatrix}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} w^{(2)}_{1,1} w^{(2)}_{1,2} \end{bmatrix}, \ \mathbf{b}^{(2)} = \begin{bmatrix} w^{(2)}_{1,0} \end{bmatrix}$$

$$f^{(1)}(z) = \texttt{ReLU}(z), \ f^{(2)}(z) = z$$

Example of BackProp (compact notation)

$$\mathbf{h}^{(0)} = \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\boldsymbol{\alpha}^{(1)} = \begin{bmatrix} \alpha_1^{(1)} \\ \alpha_2^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}$$

$$\mathbf{h}^{(1)} = \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \end{bmatrix} = f^{(1)}(\boldsymbol{\alpha}^{(1)}) = \begin{bmatrix} f^{(1)}(\alpha_1^{(1)}) \\ f^{(1)}(\alpha_2^{(2)}) \end{bmatrix} = \begin{bmatrix} \mathtt{ReLu}(\alpha_1^{(1)}) \\ \mathtt{ReLu}(\alpha_2^{(2)}) \end{bmatrix}$$

$$\boldsymbol{\alpha}^{(2)} = \left[\alpha^{(2)}\right] = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{h}^{(2)} = \left[h^{(2)}\right] = f^{(2)}(\boldsymbol{\alpha}^{(2)}) = \left[f^{(2)}(\alpha^{(2)})\right] = \left[\alpha^{(2)}\right] = \boldsymbol{\alpha}^{(2)}$$

$$\mathbf{y} = \mathbf{h}^{(2)} = \boldsymbol{\alpha}^{(2)}$$

Example of BackProp (compact notation)

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(2)}} J = \nabla_y J = \nabla_y \tfrac{1}{2}(t-y)^2 = \frac{\partial(\frac{1}{2}(t-y)^2)}{\partial y} = y - t$$

$$\mathbf{g} \leftarrow \nabla_{\alpha^{(2)}} J = \mathbf{g} \odot f^{(2)\prime}(\alpha^{(2)}) = \mathbf{g} \odot \frac{\partial \alpha^{(2)} - t}{\partial \alpha^{(2)}} = \mathbf{g} \odot 1 = \mathbf{g}$$

$$\nabla_{\mathbf{b}^{(2)}} J \leftarrow \frac{\partial J}{\partial w_{1,0}^{(2)}} = \mathbf{g}$$

$$\nabla_{\mathbf{W}^{(2)}} J \leftarrow \begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^{(2)}} & \frac{\partial J}{\partial w_{1,2}^{(2)}} \end{bmatrix} = \mathbf{g} \cdot (\mathbf{h}^{(1)})^T$$

Example of BackProp (compact notation)

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(1)}} J = \begin{bmatrix} \frac{\partial J}{\partial h_1^{(1)}} \\ \frac{\partial J}{\partial h_2^{(1)}} \end{bmatrix} = (\mathbf{W}^{(2)})^T \cdot \mathbf{g}$$

$$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(1)}} J = \mathbf{g} \odot f^{(1)\prime}(\boldsymbol{\alpha}^{(1)}) = \mathbf{g} \odot \begin{bmatrix} \frac{\partial \mathtt{ReLU}(\alpha_1^{(1)})}{\partial \alpha_1^{(1)}} \\ \frac{\partial \mathtt{ReLU}(\alpha_2^{(1)})}{\partial \alpha_2^{(1)}} \end{bmatrix} = \mathbf{g} \odot \begin{bmatrix} \mathtt{step}(\alpha_1^{(1)}) \\ \mathtt{step}(\alpha_2^{(1)}) \end{bmatrix}$$

$$\nabla_{\mathbf{b}^{(1)}} J \leftarrow \begin{bmatrix} \frac{\partial J}{\partial w_{1,0}^{(1)}} \\ \frac{\partial J}{\partial w_{2,0}^{(2)}} \end{bmatrix} = \mathbf{g}$$

$$\nabla_{\mathbf{W}^{(1)}} J \leftarrow \begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^{(1)}} & \frac{\partial J}{\partial w_{1,2}^{(1)}} \\ \frac{\partial J}{\partial w_{2,1}^{(1)}} & \frac{\partial J}{\partial w_{2,2}^{(1)}} \end{bmatrix} = \mathbf{g} \cdot (\mathbf{h}^{(1)})^T$$

Learning algorithms

- Stochastic Gradient Descent (SGD)
- SGD with momentum
- Algorithms with adaptive learning rates

Stochastic Gradient Descent

```
Require: Learning rate $\eta \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  while stopping criterion not met do
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of $m$ examples from the dataset $D$
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta \mathbf{g}$
    $k \leftarrow k + 1$
  end while
```

Observe: $\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{t})$ obtained with backprop

Stochastic Gradient Descent

η usually changes according to some rule through the iterations

Until iteration τ ($k \leq \tau$):

$$\eta^{(k)} = \left(1 - \frac{k}{\tau}\right) \eta^{(k)} + \frac{k}{\tau} \eta^{(\tau)}$$

After iteration τ ($k > \tau$):

$$\eta^{(k)} = \eta^{(\tau)}$$

SGD with momentum

Momentum can accelerate learning
Motivation: Stochastic gradient can largely vary through the iterations

```
Require: Learning rate $\eta \geq 0$
Require: Momentum $\mu \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  $\mathbf{v}^{(1)} \leftarrow 0$
  while stopping criterion not met do
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of $m$ examples from the dataset $D$
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Compute velocity: $\mathbf{v}^{(k+1)} \leftarrow \mu \mathbf{v}^{(k)} - \eta \mathbf{g}$, with $\mu \in [0, 1)$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} + \mathbf{v}^{(k+1)}$
    $k \leftarrow k + 1$
  end while
```

SGD with momentum

Momentum μ might also increase according to some rule through the iterations.

SGD with Nesterov momentum

Nesterov momentum

Momentum is applied before computing the gradient

$$\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}^{(k)} + \mu \mathbf{v}^{(k)}$$

$$\mathbf{g} = \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \mathbf{t}^{(i)})$$

Sometimes it improves convergence rate.

Algorithms with adaptive learning rates

Based on analysis of the gradient of the loss function it is possible to determine, at any step of the algorithm, whether the learning rate should be increased or decreased.

Some examples:

- AdaGrad
- RMSProp
- Adam

(see Deep Learning Book, Section 8.5 for details)

Which optimization algorithm should I choose?

Empirical approach.

Regularization

As with other ML approaches, regularization is an important feature to reduce overfitting (generalization error).

For FNN, we have several options (can be applied together):

- Parameter norm penalties
- Dataset augmentation
- Early stopping
- Parameter sharing
- Dropout

Parameter norm penalties

Add a regularization term E_{reg} to the cost function

$$E_{\text{reg}}(\boldsymbol{\theta}) = \sum_j |\theta_j|^q.$$

Resulting cost function:

$$\bar{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda E_{\text{reg}}(\boldsymbol{\theta}).$$

Dataset augmentation

Generate additional data and include them in the dataset.

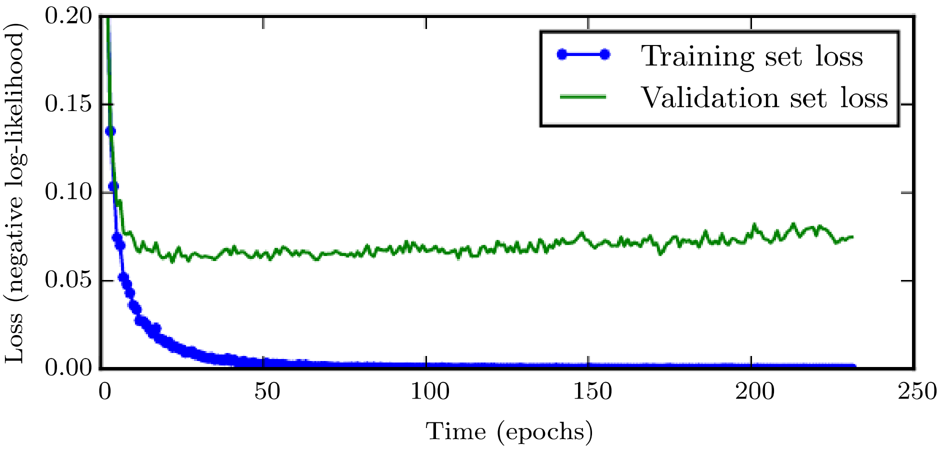
- Data transformations (e.g., image rotation, scaling, varying illumination conditions, ...)
- Adding noise

In Exercise 7, noisy XOR converges faster than XOR.

Early stopping

Early stopping:

Stop iterations early to avoid overfitting to the training set of data



When to stop? Use cross-validation to determine best values.

Parameter sharing

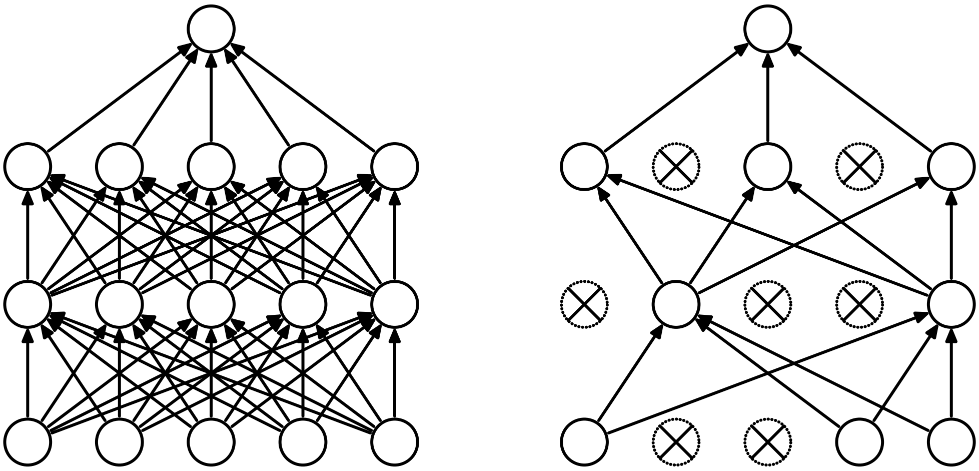
Parameter sharing: constraint on having subsets of model parameters to be equal.

Advantages also in memory storage (only the unique subset of parameters need to be stored).

In Convolutional Neural Networks (CNNs) parameter sharing allows for invariance to translation.

Dropout

Dropout: Randomly remove network units with some probability α



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

Image from Srivastava *et al.*. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Dropout

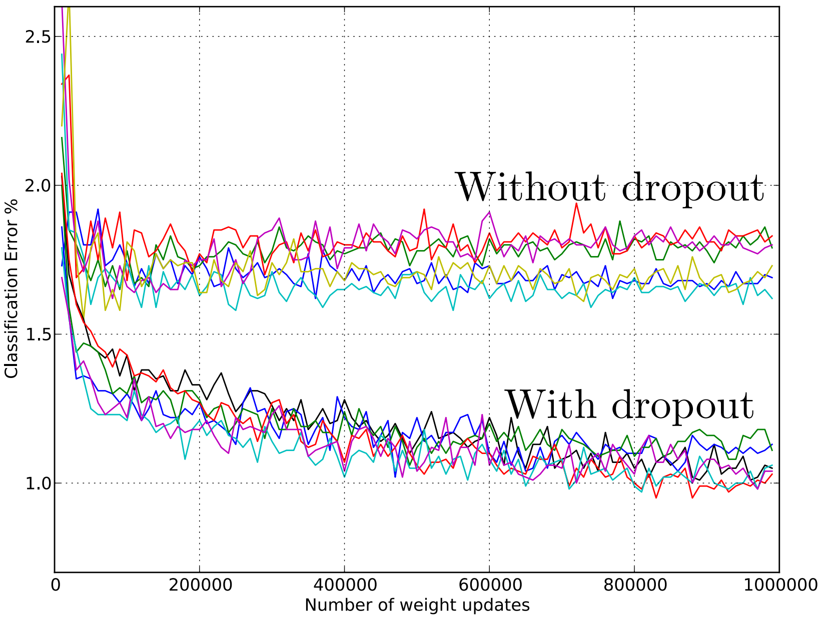


Image from Srivastava *et al.*. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Summary

Feedforward neural networks (FNNs)

- parametric models with many combination of simple functions
- can effectively approximate any function (no need to guess kernel models)
- must be carefully designed (empirically)
- efficient ways to optimize the loss function
- deep architectures perform better
- optimization performance can be improved with momentum and adaptive learning rate
- generalization error can be reduced with regularization