Goals for Today

Neural Networks of Increasing Complexity:

- Perceptron
- Single Layer Perceptron Model
- Multi-layer Perceptron Models

Training Neural Networks:

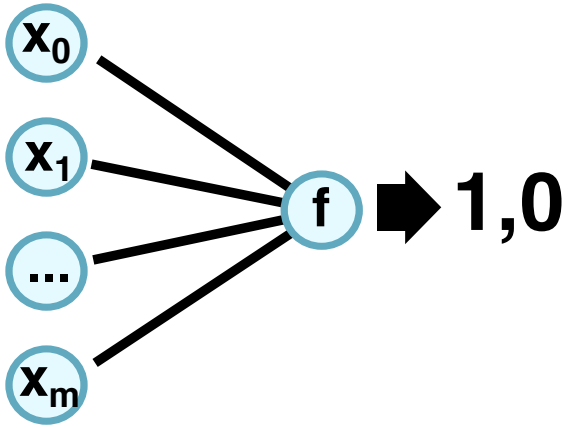
- Delta Rule
- Back-propagation

Design of Neural Networks:

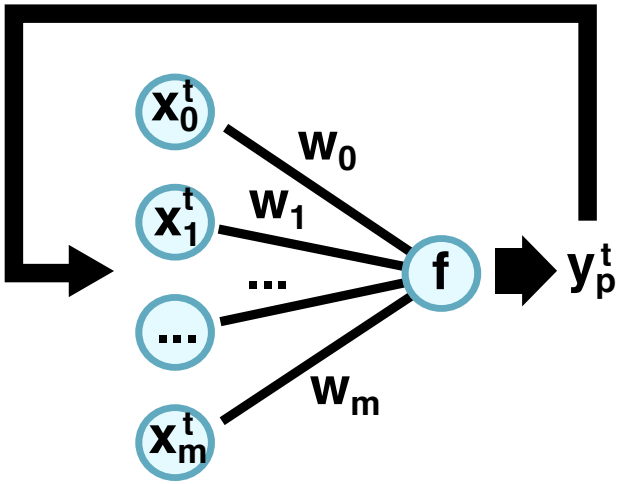
- Activation Functions
- Architectures
- Training Algorithms

History of Deep Learning

Inspired by the discovery of neurons, McCulloch and Pitts (1954) proposed a simple mathematical abstraction that fire (1/0) after some combination of inputs crosses a threshold.

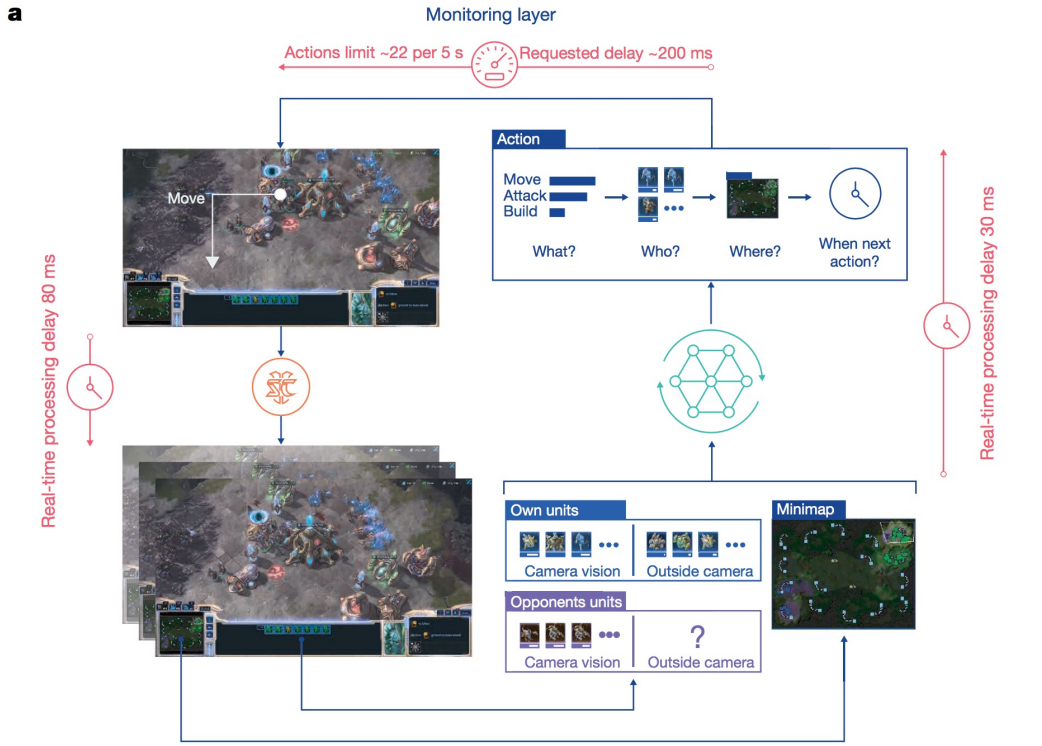


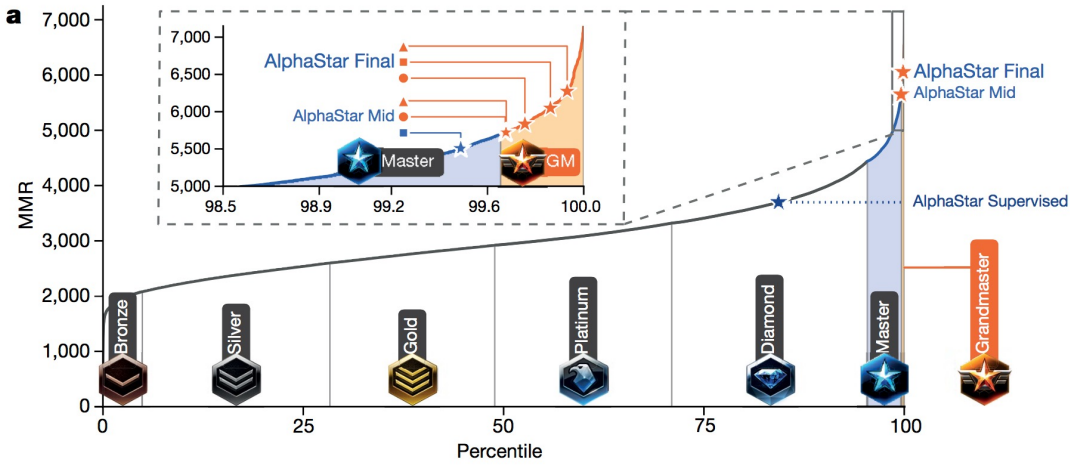
In 1957, Rosenblatt published the first algorithm for training a perceptron (and gives it its name), in 1960 Widrow and Hoff publish adaptive linear neuron (a continuous version of the perceptron).



After this, work on neural networks stalled because no one knew how to effectively train more complex architectures. This changed with the (re)disovery of backpropagation in 1986 by Hinton and Williams. Another AI winter followed. Eventually interest in neural networks was rebirthed by big data and big computation.

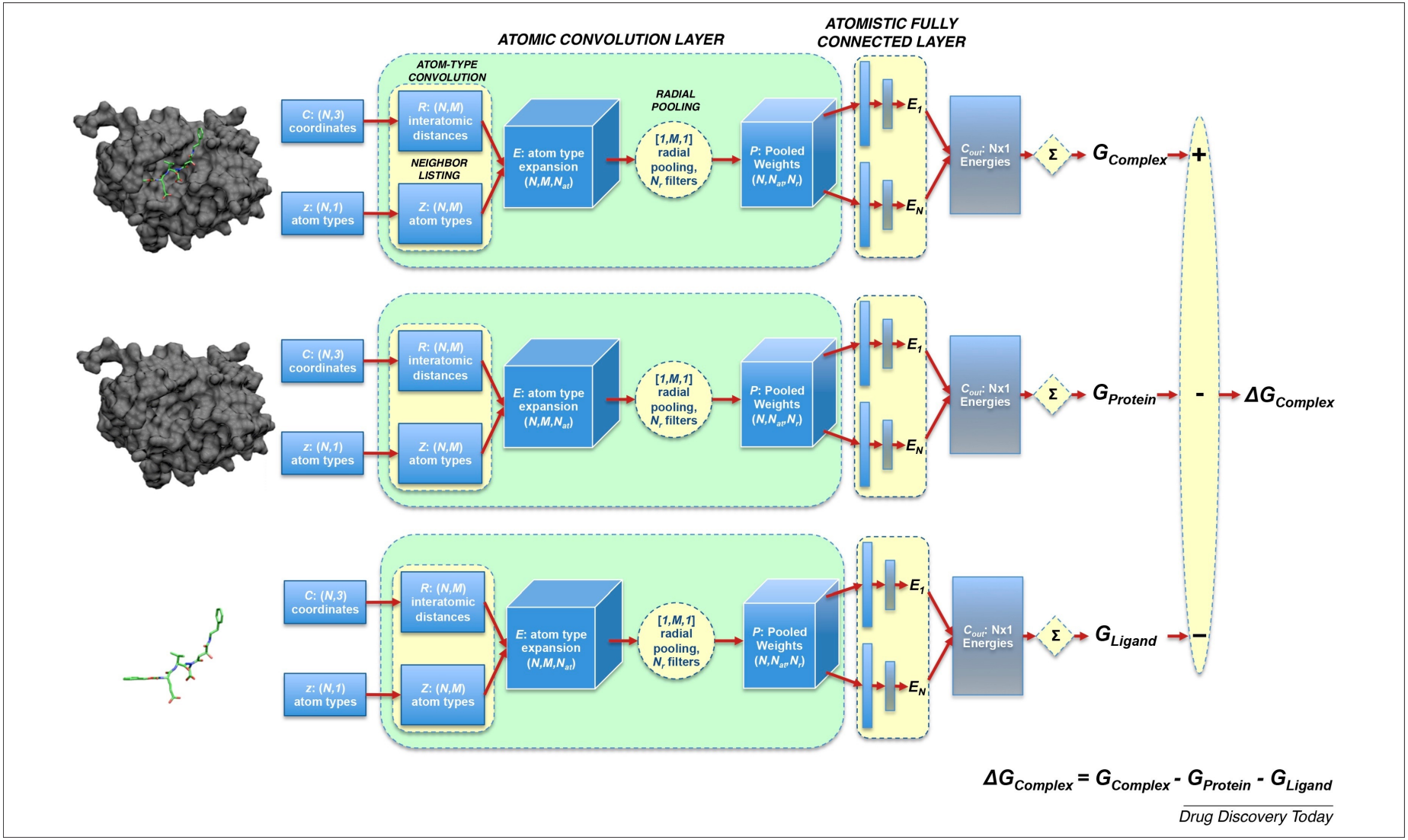
Modern Deep Learning (Starcraft 2)





Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning. *Nature* **2019**, *575* (7782), 350–354. https://doi.org/10.1038/s41586-019-1724-z.

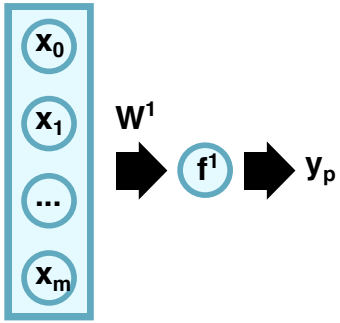
Modern Deep Learning (Drug Docking)



Gomes, J.; Ramsundar, B.; Feinberg, E. N.; Pande, V. S. Atomic Convolutional Networks for Predicting Protein-Ligand Binding Affinity. arXiv:1703.10603 [physics, stat] 2017.

What is "Deep" Learning?

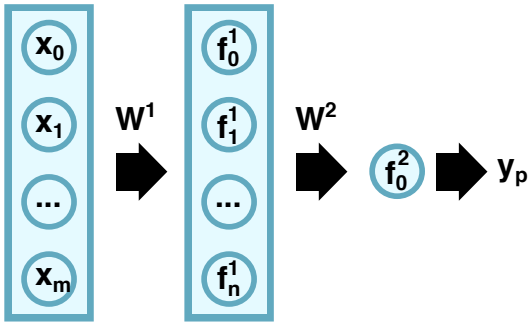
Perceptron



No hidden layers

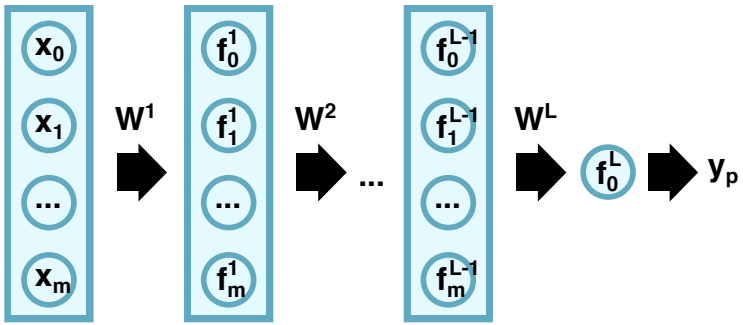
Note: I have replaced the "weights" by a matrix; we'll return to this momentarily.

Multi-Layer Perceptron (3 layer)



1 "hidden layer"
(not directly observable in the outputs)
still not "deep"

Multi-Layer Perceptron (>3 layers)

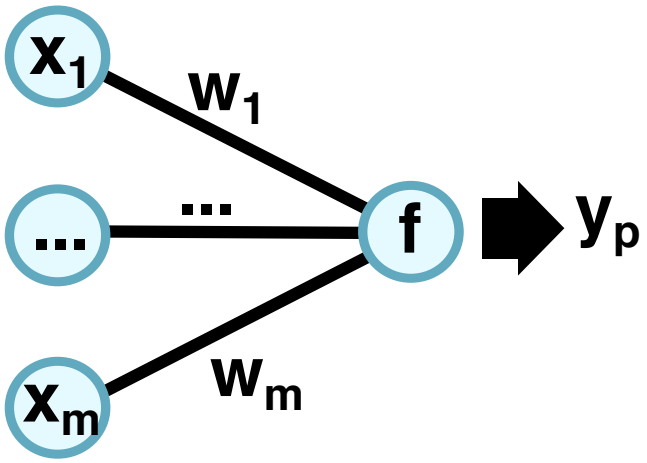


> 1 "hidden layers"
(constitutes "deep")

Note: MLP is a misnomer but it has stuck (i.e., the f's are rarely Heaviside f's)

Models like this are inherently difficult to train. Multilayer networks are known as the "universal function approximation". The corollary is that they are prone to overfitting and require careful training.

Perceptron Basics



X_i are features from our data

w_i are scalar weights for each feature

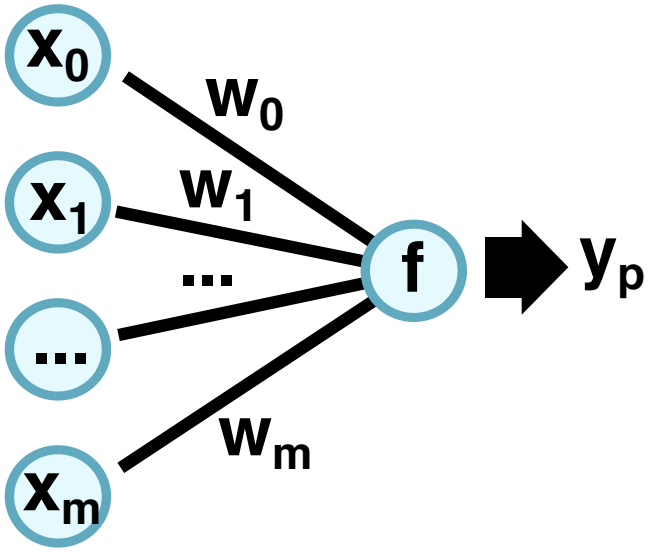
$$f(z) = \begin{cases} 1 & \text{if } z > \theta \\ 0 & \text{else} \end{cases}$$

θ is the "threshold" for activation, - θ is known as the "bias"

$$z = \sum_{i=1}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$

"z" is considered a "derived feature" (i.e., a learned linear combination of input features

Incorporation of bias as a node in the l-1 layer



$$x_0 = 1 \quad w_0 = \theta$$

special node and weight

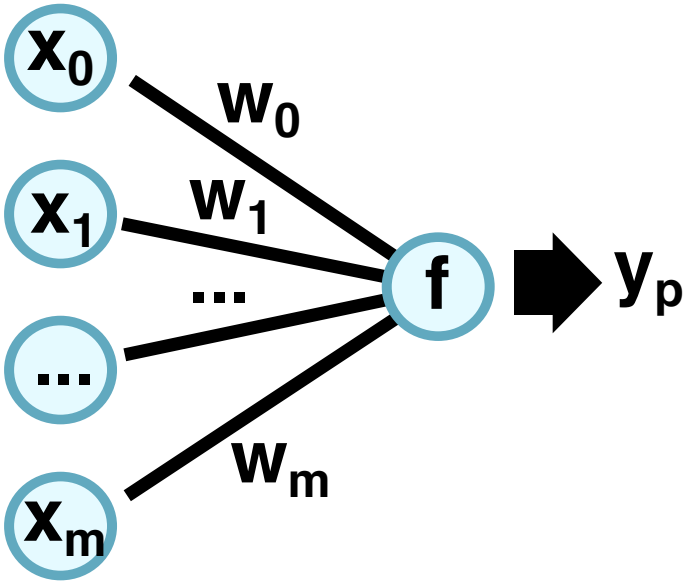
$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{else} \end{cases}$$

activation is now centered about 0

$$z = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$

Index runs from 0

Perceptron Training



Let's say that we have t samples ($\mathbf{x}$,y_i) and we are training for a mean squared loss on y:

$$L\left[\mathbf{y},\mathbf{y}_p\right]=\frac{1}{2}\sum_t\left(y_0^t-y_p^t\right)^2$$

½ is just for convenience, not necessary

$$y_p^t=f\left(\mathbf{x}^{\mathrm{T}}\mathbf{w}\right)$$

(1x1) *(1xm+1) (m+1x1)*

To train by gradient descent we need to evaluate dL/dw_i (Delta rule):

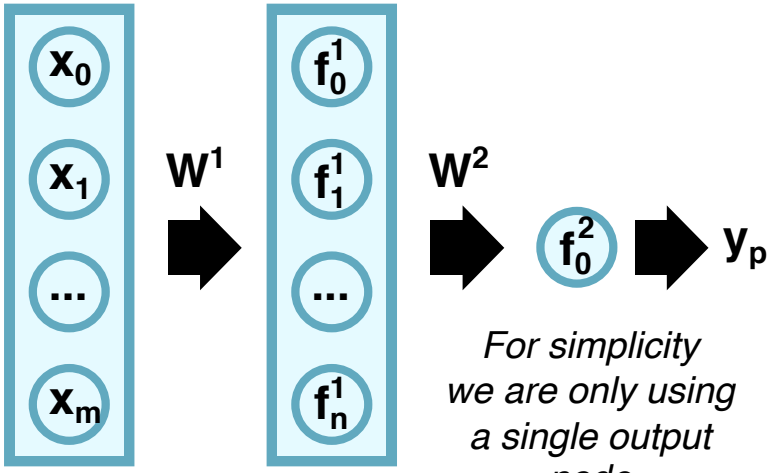
$$\begin{aligned}\frac{dL}{dw_i}&=\frac{1}{2}\frac{d}{dw_i}\sum_t\left[y_0^t-f(z_t)\right]^2\\&=\sum_t\left[y_0^t-f(z_t)\right]\frac{d}{dw_i}\left[y_0^t-f(\sum_i w_ix_i)\right]\\&=-\sum_t\left[y_0^t-f(z_t)\right]f'(z_t)x_i\end{aligned}$$

$$\mathbf{w}_{i+1}=\mathbf{w}_i-\eta\nabla\mathbf{L}$$

For stochastic gradient descent choose t randomly

Important to choose f() with simple derivative w.r.t. weights

Multilayer Perceptron Model (1 hidden layer)



For simplicity we are only using a single output node

Assuming fully connected layers:

$$\mathbf{W}^1 = \begin{bmatrix} 0 & w_{01} & \dots & w_{0m} \\ 0 & w_{11} & \dots & w_{1m} \\ \dots & & & \\ 0 & w_{n1} & \dots & w_{nm} \end{bmatrix}$$

Zeros in first column are from the special treatment of the bias node.

n+1 x m+1 matrix where wij is the weight connecting node i in layer 1 with node j in layer 0.

Bias node in layer i is unconnected to previous bias.

Calculating predictions (y_p):

+1 from bias nodes omitted for clarity

$$\underset{(nxm)}{\mathbf{W}^1}\underset{(mx1)}{\mathbf{x}} = \underset{(nx1)}{\mathbf{z}^1}$$

Where $\mathbf{z}^1$ is the input vector for layer 1

$$\underset{(1xn)}{\mathbf{W}^2}\underset{(nx1)}{f^1(\mathbf{z}^1)} = \underset{(1x1)}{\mathbf{z}^2}$$

Where $\mathbf{z}^2$ is the input vector for layer 2

$$\underset{(1x1)}{f^2(\mathbf{z}^2)} = \underset{(1x1)}{y_p}$$

Thus, $y_p = f^2(\mathbf{W}^2 f^1(\mathbf{W}^1\mathbf{x}))$

and in the general case: $\mathbf{y}_p = f^L(\mathbf{W}^L \ldots f^2(\mathbf{W}^2 f^1(\mathbf{W}^1\mathbf{x}))\ldots)$

The Backpropagation Concept

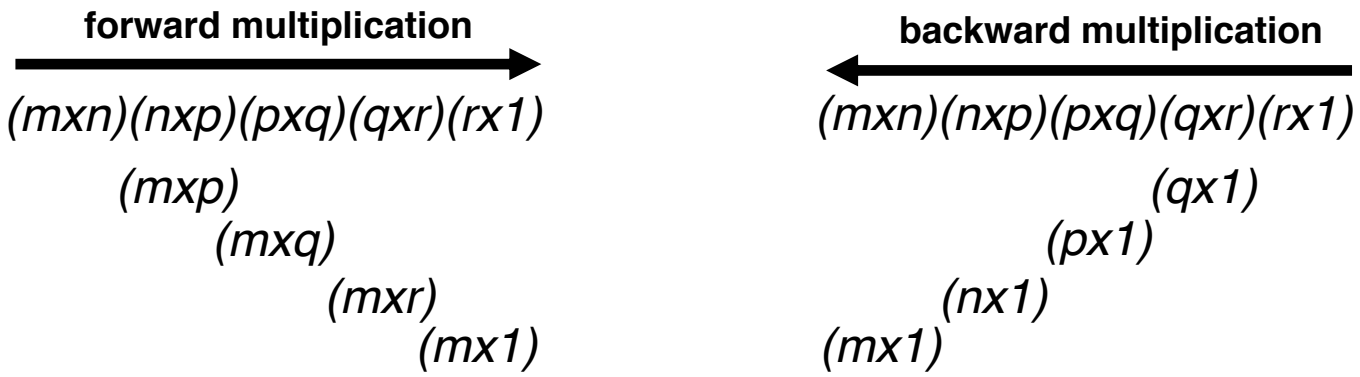
Backpropagation is a special application of **automatic differentiation**. In practice it is just the chain rule for matrices and realizing that order is important. Suppose we want to take the derivative of this function with respect to x:

$$F = f(g(h(u(v(x)))))$$

Applying the chain rule:

$$\frac{\partial F}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$

Suppose these quantities are each matrices such that:



Calculating the derivatives in reverse ends up being much more efficient with neural networks.

Multilayer Perceptron (Backpropagation)

For our case with a single hidden layer and one output node:

$$\mathbf{x} \xrightarrow{\mathbf{W}^1} \mathbf{a}^1 \xrightarrow{f^1} \mathbf{z}^1 \xrightarrow{\mathbf{W}^2} \mathbf{a}^2 \xrightarrow{f^2} y_p$$

Where $\mathbf{a}^1$/$\mathbf{a}^2$ are the vectors of input values to the activation functions and $\mathbf{z}^1$/y_p are the outputs after applying the activation functions (needed in the partial derivatives that follow).

To train our model we need: $\frac{\partial L}{\partial \mathbf{W}^1}$ $\frac{\partial L}{\partial \mathbf{W}^2}$

For notational simplicity these will be evaluated for a single training sample (not showing index)

Starting with the $\mathbf{W}^2$ term (the "back" part of back propagation):

$$\frac{\partial L}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial}{\partial \mathbf{W}^2}\left[\left(\mathbf{W}^2\right)^{\mathrm{T}}\mathbf{z}^1\right] = \left(y_p - y_0\right) f^2\left(\mathbf{a}^2\right)' \mathbf{z}^1$$

Assuming squared loss; same as perceptron result

Now, the $\mathbf{W}^1$ term:

$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{a}^1}\frac{\partial \mathbf{a}^1}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{a}^1}\frac{\partial}{\partial \mathbf{W}^1}\left[\left(\mathbf{W}^1\right)^{\mathrm{T}}\mathbf{x}\right] = \frac{\partial L}{\partial \mathbf{a}^1}\mathbf{x}$$

Multilayer Perceptron (Backpropagation)

For our case with a single hidden layer and one output node:

$$\mathbf{x} \xrightarrow{\mathbf{W}^1} \mathbf{a}^1 \xrightarrow{f^1} \mathbf{z}^1 \xrightarrow{\mathbf{W}^2} \mathbf{a}^2 \xrightarrow{f^2} y_p$$

Where $\mathbf{a}^1$/$\mathbf{a}^2$ are the vectors of input values to the activation functions and $\mathbf{z}^1$/y_p are the outputs after applying the activation functions (needed in the partial derivatives that follow).

To train our model we need: $\frac{\partial L}{\partial \mathbf{W}^1}$ $\frac{\partial L}{\partial \mathbf{W}^2}$

For notational simplicity these will be evaluated for a single training sample (not showing index)

Starting with the $\mathbf{W}^2$ term (the "back" part of back propagation):

$$\frac{\partial L}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial}{\partial \mathbf{W}^2}\left[\left(\mathbf{W}^2\right)^{\mathrm{T}}\mathbf{z}^1\right] = \left(y_p - y_0\right) f^2\left(\mathbf{a}^2\right)' \mathbf{z}^1$$

Assuming squared loss; same as perceptron result

Now, the $\mathbf{W}^1$ term:

$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{a}^1}\frac{\partial \mathbf{a}^1}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{a}^1}\frac{\partial}{\partial \mathbf{W}^1}\left[\left(\mathbf{W}^1\right)^{\mathrm{T}}\mathbf{x}\right] = \frac{\partial L}{\partial \mathbf{a}^1}\mathbf{x}$$

Common Activation Functions ($R^n \rightarrow R^1$)

Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$

Small grad issues

Rectified Linear Unit (Relu): $f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$

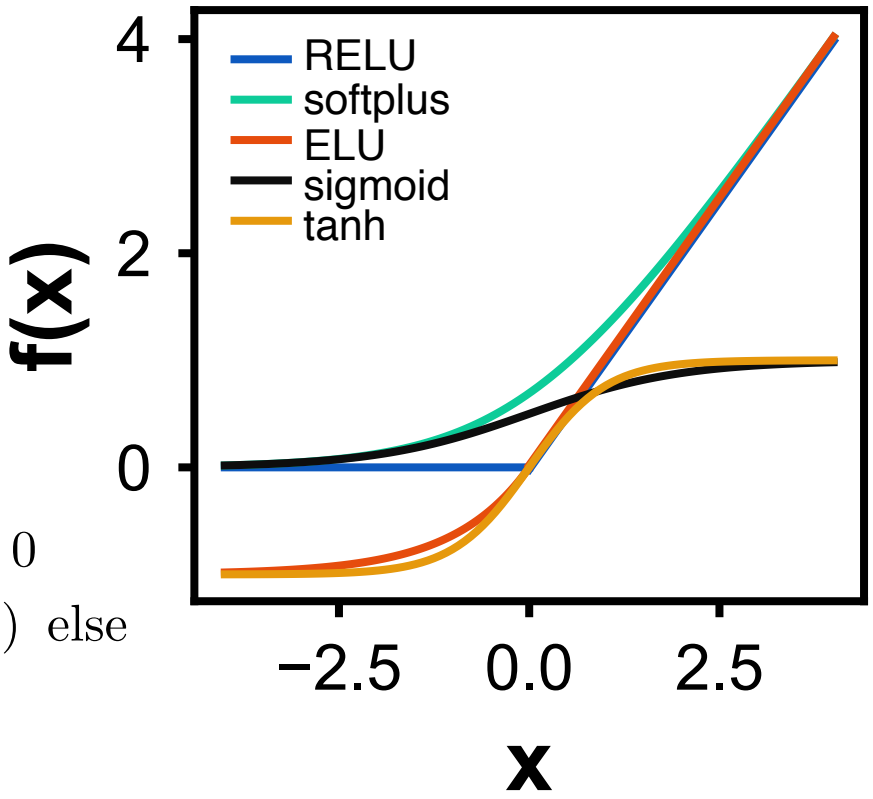
Dying Relu phenomenon

Softplus (smoothRelu): $f(x) = ln(1+e^x)$

Exponential Linear Unit (ELU) $f(x) = \begin{cases} x & \text{if } x > 0 \\ a(e^x - 1) & \text{else} \end{cases}$

Tanh: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Small grad issues



Layer Operations ($R^n \rightarrow R^n$):

Max:

$$f(x_i) = \begin{cases} 1 & \text{if } x_i = \max(\mathbf{x}) \\ 0 & \text{else} \end{cases}$$

Softmax:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Additional Training Details

If you initialize training with weights near 0, you are operating within the approximately linear regime of most activation functions. Thus, you can think of the NN as starting with a linear approximation and learning non-linear transformations of the input features as needed.

As networks become deeper, the gradients associated with individual weights vanish (the "**vanishing gradient**" problem). A lot of modern research is dedicated to solving this problem (e.g., by searching for better weight initialization schemes.)

The optimization surfaces of NN's are **highly non-convex**. Stochastic gradient descent (and its variants) and multiple training are necessary for training predictive models.

NN's are **prone to overfitting/memorization**. Careful evaluation of learning curves, node dropout, and regularization are necessary for training predictive models.

The parameters in a NN are not interpretable. You can train many independent models and compare/average their predictions, but you can't average their parameters.