CSC 411 Lecture 10: Neural Networks I

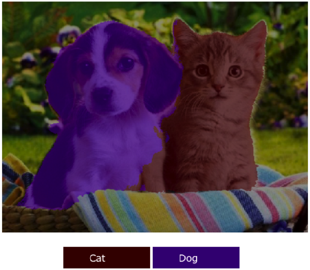
Ethan Fetaya, James Lucas and Emad Andrews

University of Toronto

Today

- Multi-layer Perceptron
- Forward propagation
- Backward propagation

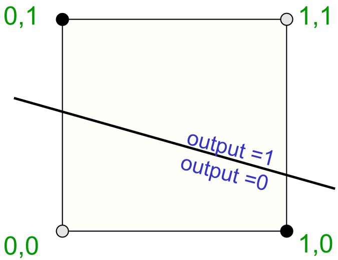
Motivating Examples



Are You Excited about Deep Learning?

Limitations of Linear Classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features x_i
- Many decisions involve non-linear functions of the input
- Canonical example: do 2 input elements have the same value?



- The positive and negative cases cannot be separated by a plane
- What can we do?

How to Construct Nonlinear Classifiers?

- We would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Use a large number of simpler functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs
  - Or we can make these functions depend on additional parameters $\rightarrow$ need an efficient method of training extra parameters

Inspiration: The Brain

- Many machine learning methods inspired by biology, e.g., the (human) brain
- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^{4}$ other neurons

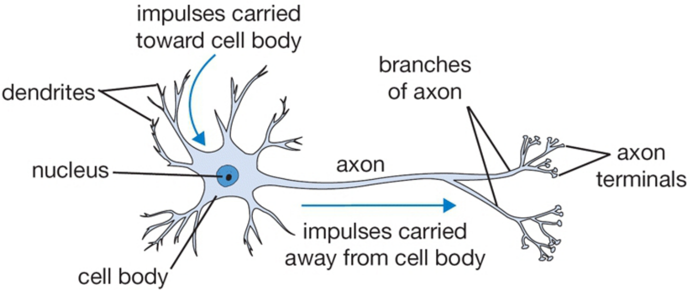


Figure: The basic computational unit of the brain: Neuron

[Pic credit: http://cs231n.github.io/neural-networks-1/]

Mathematical Model of a Neuron

- Neural networks define functions of the inputs (hidden features), computed by neurons
- Artificial neurons are called units

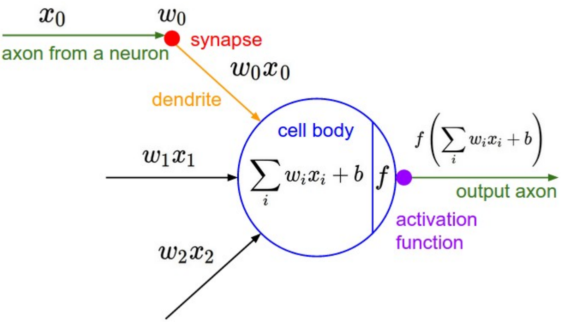


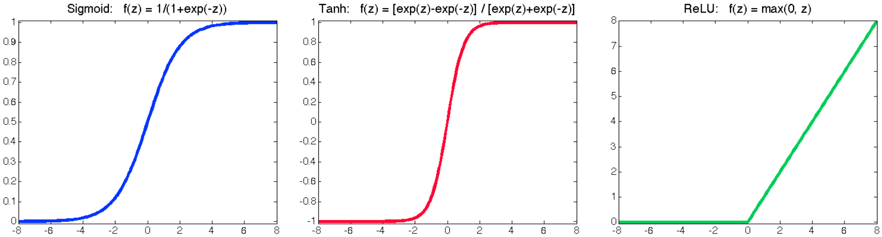
Figure: A mathematical model of the neuron in a neural network

[Pic credit: http://cs231n.github.io/neural-networks-1/]

Activation Functions

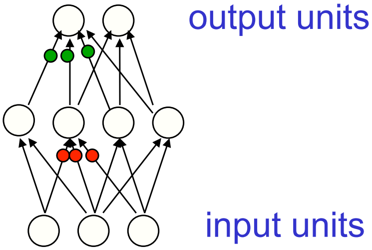
Most commonly used activation functions:

- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$



Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:



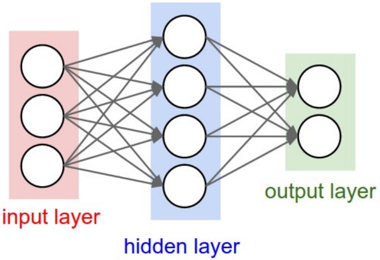


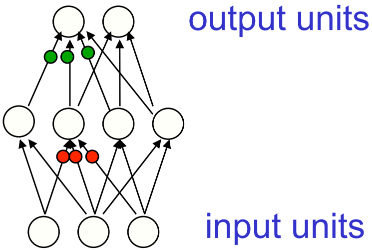
Figure: Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Each unit computes its value based on linear combination of values of units that point into it, and an activation function

[http://cs231n.github.io/neural-networks-1/]

Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:



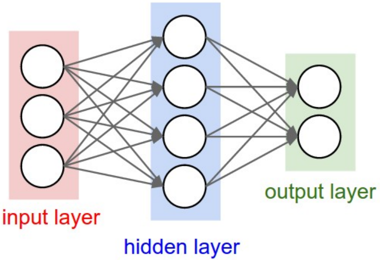


Figure: Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Naming conventions; a 2-layer neural network:
  - One layer of hidden units
  - One output layer
    (we do not count the inputs as a layer)

[http://cs231n.github.io/neural-networks-1/]

Neural Network Architecture (Multi-Layer Perceptron)

- Going deeper: a 3-layer neural network with two layers of hidden units

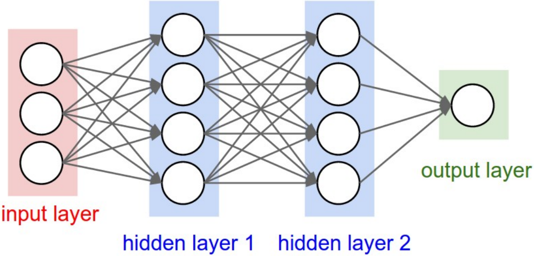


Figure: A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

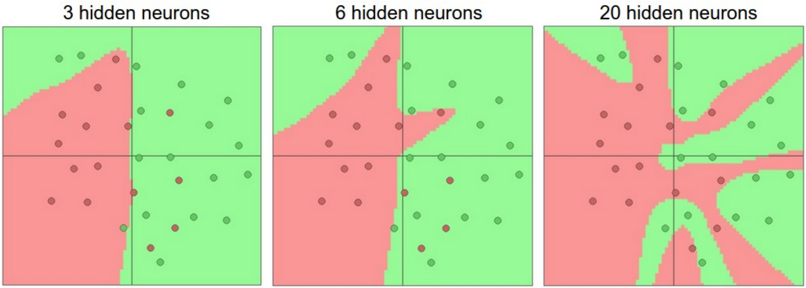
- Naming conventions; a N-layer neural network:
  - $N - 1$ layers of hidden units
  - One output layer

[http://cs231n.github.io/neural-networks-1/]

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper



- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper (still kind of an open theory question)? One hidden layer might need exponential number of neurons, deep can be more compact.

[http://cs231n.github.io/neural-networks-1/]

Demo

- Great tool to visualize networks http://playground.tensorflow.org/
- Highly recommend playing with it!

Neural Networks

- Two main phases:
  - Forward pass: Making predictions
  - Backward pass: Computing gradients

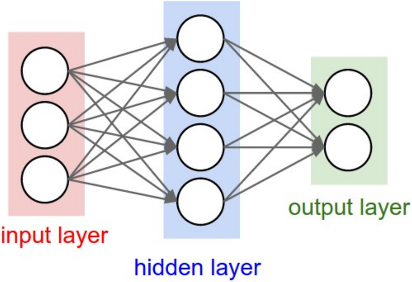
Forward Pass: What does the Network Compute?

- Output of the network can be written as:

$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})$$

(j indexing hidden units, k indexing the output units, D number of inputs)



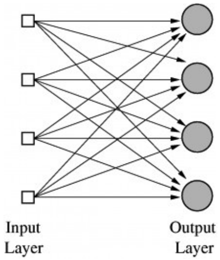
- Activation functions f, g: sigmoid/logistic, tanh, or rectified linear (ReLU)

$$\sigma(z) = \frac{1}{1 + \exp(-z)}, \quad \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}, \quad \mathrm{ReLU}(z) = \max(0, z)$$

- What if we don't use any activation function?

Special Case

- What is a single layer (no hiddens) network with a sigmoid act. function?

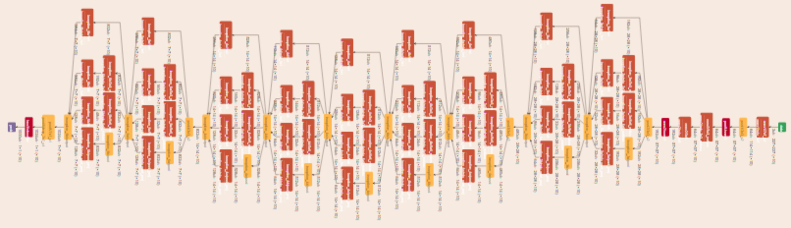


- Network:

$$
\begin{aligned}
o_k(\mathbf{x}) &= \frac{1}{1+\exp(-z_k)} \\
z_k &= w_{k0} + \sum_{j=1}^{J} x_j w_{kj}
\end{aligned}
$$

- Logistic regression!

- Feedforward network - Connections are a directed acyclic graphs (DAG)
- Layout can be more complicated than just k hidden layers.



How do we train?

- We've seen how to compute predictions.
- How do we train the network to make sensible predictions?

Training Neural Networks

- How do we find weights?

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \mathrm{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

  where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network
  - can use any (smooth) loss function we want.
- Problem: With hidden units the objective is no longer convex!
- No guarantees gradient methods won't end up in a (bad) local minima/ saddle point.
- Some theory/experimental evidence that most local minimas are good, i.e. almost as good as the global minima.
- SGD with some (critical) tweaks works well. It is not really well understood.

Training Neural Networks: Back-propagation

- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

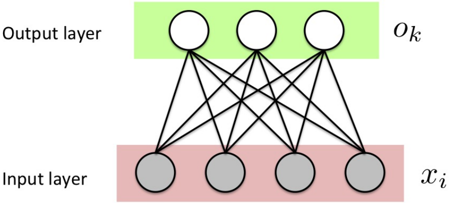
Key Idea behind Backpropagation

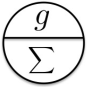
- We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit
- This is just the chain rule!

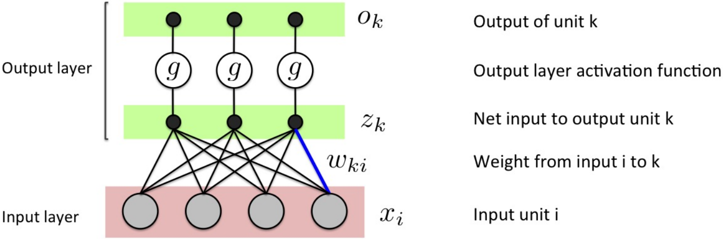
Useful Derivatives

name	**function**	**derivative**
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

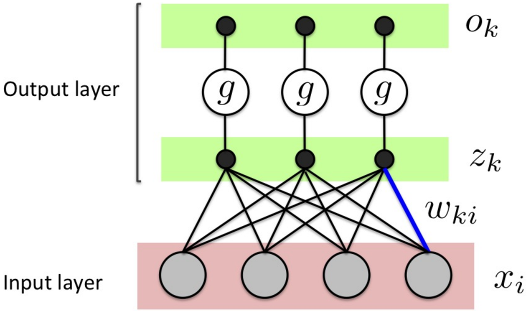
- Let's take a single layer network and draw it a bit differently







Computing Gradients: Single Layer Network

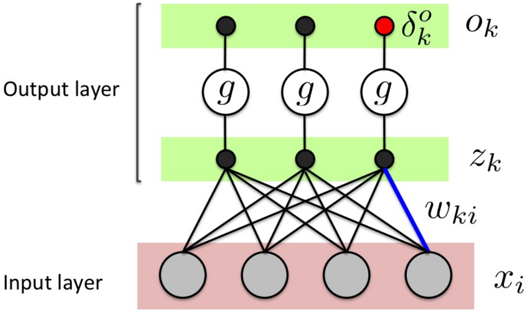


- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

- Error gradient is computable for any smooth activation function $g()$, and any smooth error function

Computing Gradients: Single Layer Network

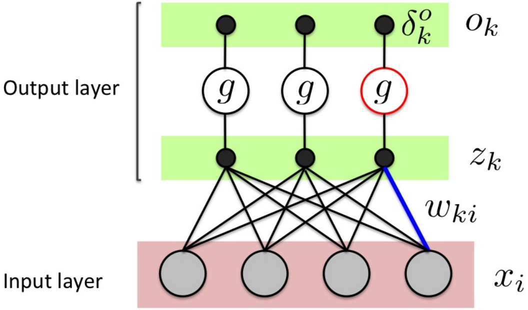


$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \underbrace{\frac{\partial E}{\partial o_k}}_{\delta_k^o} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network

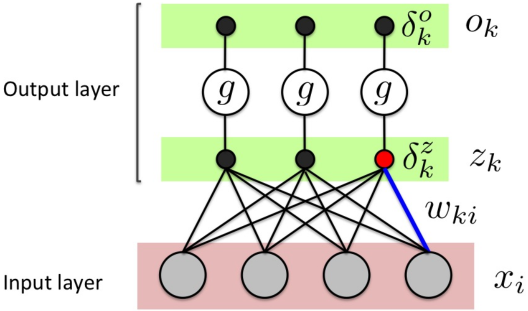


$$\frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^o \frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network

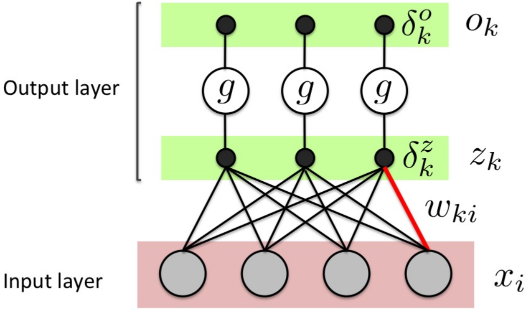


$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}} = \underbrace{\delta_k^o \cdot \frac{\partial o_k}{\partial z_k}}_{\delta_k^z} \frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network



$$\frac{\partial z_k}{\partial w_{ki}}$$

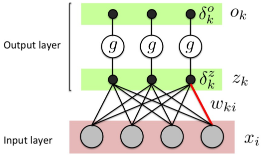
- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \cdot x_i$$

Gradient Descent for Single Layer Network

- Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)} := \delta_k^o$$



Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1 + \exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1 - o_k^{(n)}) \end{aligned}$$

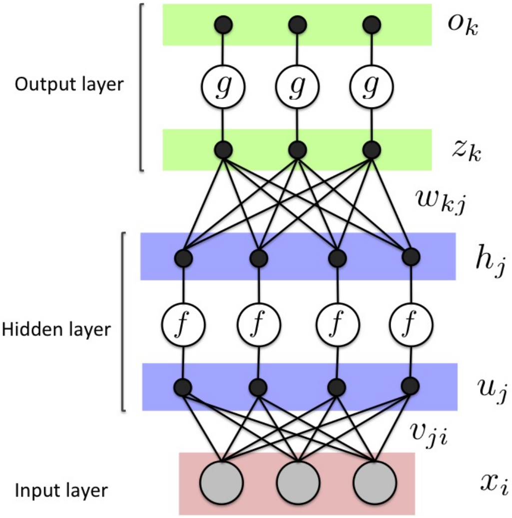
- The error gradient is then:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{ki}} = \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

- The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

Multi-layer Neural Network



Output of unit k

Output layer activation function

Net input to output unit k

Weight from hidden unit j to output k

Output of hidden unit j

Hidden layer activation function

Net input to unit j

Weight from input i to j

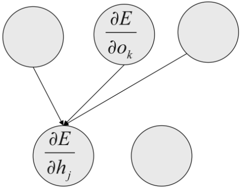
Input unit i

Back-propagation: Sketch on One Training Case

- Convert discrepancy between each output and its target value into an error derivative

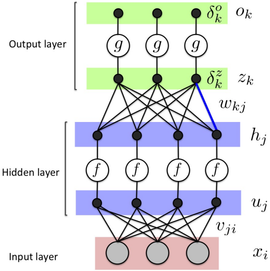
$$E = \frac{1}{2}\sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at k to each unit j according to its influence on k (depends on w_{kj})]



- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

Gradient Descent for Multi-layer Network



- The output weight gradients for a multi-layer network are the same as for a single layer network

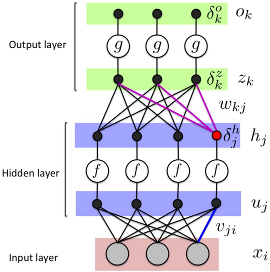
$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

- Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} =$$

Gradient Descent for Multi-layer Network



- The output weight gradients for a multi-layer network are the same as for a single layer network

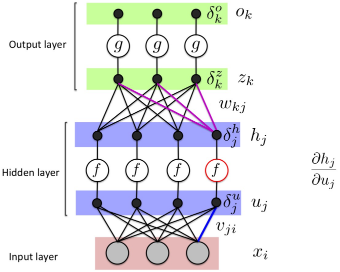
$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

- Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_{k} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_{k} \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

Gradient Descent for Multi-layer Network



- The output weight gradients for a multi-layer network are the same as for a single layer network

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

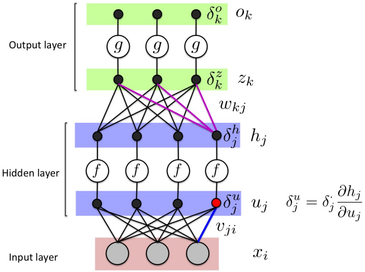
where δ_k is the error w.r.t. the net input for unit k

- Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} =$$

Gradient Descent for Multi-layer Network



- The output weight gradients for a multi-layer network are the same as for a single layer network

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

- Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{u,(n)} x_i^{(n)}$$

Backprob in deep networks

- The exact same ideas (and math) can be used when we have multiple hidden layer - compute $\frac{\partial E}{\partial h_j^L}$ and use it to compute $\frac{\partial E}{\partial w_{ij}^L}$ and $\frac{\partial E}{\partial h_j^{L-1}}$
- Two phases:
  - Forward: Compute output layer by layer (in order)
  - Backwards: Compute gradients layer by layer (reverse order)
- Modern software packages (theano, tensorflow, pytorch) do this automatically.
  - You define the computation graph, it takes care of the rest.

Training neural networks

Why was training neural nets considered hard?

- With one or more hidden layers the optimization is no longer convex.
  - No Guarantees, optimization can end up in a bad local minima/ saddle point.
- Vanishing gradient problem.
- Long compute time.
  - Training on imagenet can take 3 weeks on GPU ($\sim \times 30$ speedup!)

We will talk about a few simple tweaks that made it easy!

Activation functions

- Sigmoid and tanh can saturate.
  - $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$ what happens when z is very large/small?
- Even without saturation gradients can vanish in deep networks
- ReLU have 0 or 1 gradients, as long as not all path to the error are zero the gradient doesn't vanish.
  - Neurons can still "die".
- Other alternatives: maxout, leaky ReLU, ELU (ReLU is by far the most common).
- On output layer usually no activations or sigmoid/softmax (depends on what do we want to represent)

Initialization

How do we initialize the weights?

- What if we initialize all to a constant c?
  - All neurons will stay the same!
  - Need to break symmetry - random initialization
- Standard approach - $W_{ij} \sim \mathcal{N}(0, \sigma^2)$
  - If we pick σ^2 too small - output will converge to zero after a few layers.
  - If we pick σ^2 too large - output will diverge.
- Xavier initialization - $\sigma^2 = 2/(n_{in} + n_{out})$
  - n_{in} and n_{out} are the number of units in the previous layer and the next layer
- He initialization - $\sigma^2 = 2/n_{in}$
  - Builds on the math of Xavier initialization but takes ReLU into account.
  - Recommended method for ReLUs (i.e. almost always)

Momentum

"Vanilla" SGD isn't good enough to train - bad at ill-conditioned problems.

- Solution - add momentum

$$v_{t+1} = \beta v_t + \nabla L(w_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

  - Builds up when we continue at the same direction.
  - decreases when we change signs
- Normality pick $\beta = 0.9$
- More recent algorithms like ADAM still use momentum (just add a few more tricks).

Nice visualization - `http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html`