Stochastic/Approximate Gradient Descent

Approximate Gradient Descent
(Stochastic Backpropagation)

Normally, in gradient descent, we would need to compute how the error over all input samples (true gradient) changes with respect to a small change in a given weight.

But the common form of the gradient descent algorithm takes one input pattern, **compute the error of the network on that pattern only**, and updates the weights using only that information.

- Notice that the new weight may not be good/better for all patterns, but we expect that if we take a small step, we will average and approximate the true gradient.

Stochastic Approximation to Steepest Descent

Instead of updating every weight until all examples have been observed, **we update on every example**:

$$\nabla w_i \cong \eta\ (t-o)\ x_i$$

Remarks:

- Speeds up learning significantly when data sets are large
  - Use a smaller learning step!
- When there are multiple local minima, stochastic gradient descent may avoid the problem of getting stuck on a local minimum.

Stopped here

Gradient Descent Backpropagation Algorithm

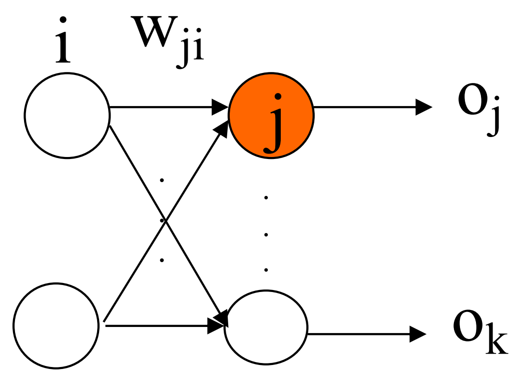
Derivation for

General Activation Functions Networks

Stochastic Backpropagation

To calculate the partial derivative of E (here: loss on a single input) w.r.t a given weight w_{ji} of a node j , we have to consider whether this is the weight of an output or hidden node:

If w_{ji} is an **output** node weight, the situation is simple.



We always use the Chain Rule

Assuming multi-label MSE loss (simplest to show):

$$E = \sum_{j=1}^{K} \left(t_j - o_j\right)^2$$

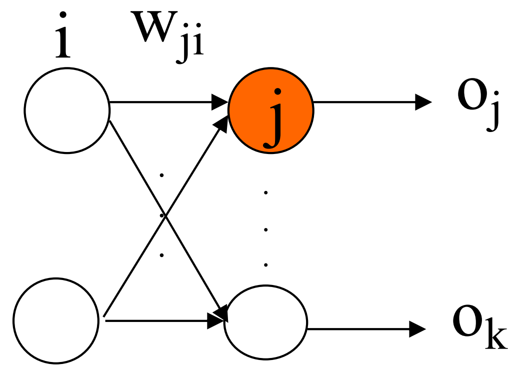
$$o_j = f(net_j)$$

$$net_j = \sum_i o_i w_{ji}$$

Stochastic Backpropagation

To calculate the partial derivative of E w.r.t a given weight w_{ji} of a node j , we have to consider whether this is the weight of an output or hidden node:

If w_{ji} is an **output** node weight, the situation is simple. We use the Chain Rule:



$$\frac{dE}{dw_{ji}} = \frac{dE}{do_j} \times \frac{do_j}{dnet_j} \times \frac{dnet_j}{dw_{ji}}$$

$$E = \sum_{j=1}^{K} (t_j - o_j)^2$$

$$\frac{dE}{dw_{ji}} = -(t_j - o_j) \times f'(net_j) \times o_i$$

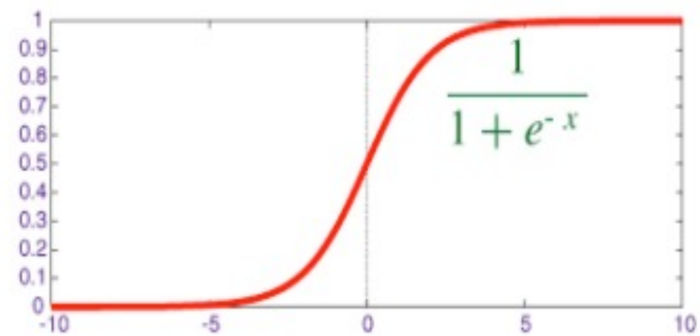
Note that output of node i (o_i) is the input to node j.

$$o_j = f(net_j)$$

$$net_j = \sum_i o_i w_{ji}$$

Transfer Function Derivatives

Sigmoid:



$$f'(n) = \frac{d}{dn}\left(\frac{1}{1+e^{-n}}\right) = \frac{e^{-n}}{(1+e^{-n})^2}$$

$$= \left(1 - \frac{1}{1+e^{-n}}\right)\left(\frac{1}{1+e^{-n}}\right) = \boxed{(1-a)(a)}$$

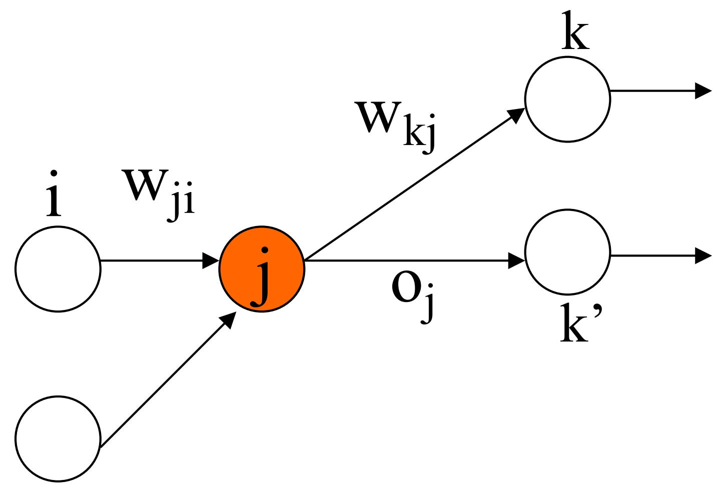
Linear: $f'(n) = \frac{d}{dn}(n) = 1$

- Computing the derivative is **very easy after the forward pass**
- Sigmoid nodes **saturate** **when output is large in magnitude.**
- Earlier layers learn much slower – **vanishing gradient**

Backpropagation – Hidden nodes

The situation is more complex with a hidden node, because we don't know what the output of a hidden node should be how it affects loss.

If w_{ji} is a **hidden** node weight:



$$\frac{dE}{dw_{ji}} = \frac{dE}{do_j} \times \frac{do_j}{dnet_j} \times \frac{dnet_j}{dw_{ji}}$$

$$= \frac{dE}{do_j} \times f'(net_j) \times o_i$$

$$net_j = \sum_i o_i w_{ji}$$

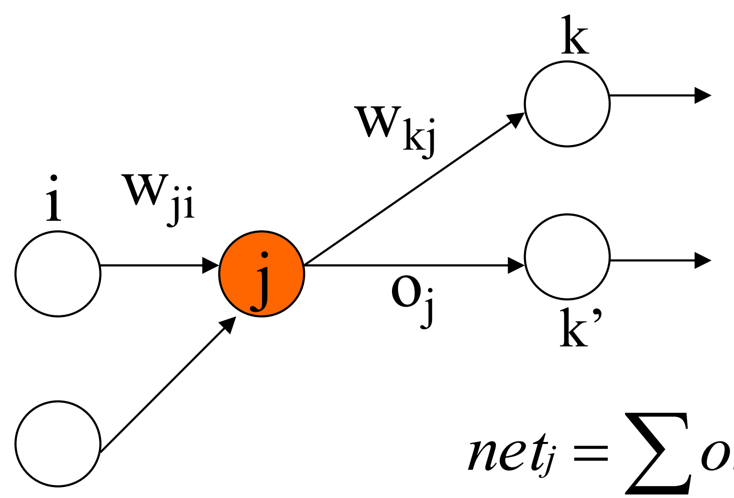
$$o_j = f(net_j)$$

Backpropagation – Hidden nodes

If w_{ji} is a **hidden** node weight:

$$\frac{dE}{dw_{ji}} = \frac{dE}{do_j} \times \frac{do_j}{dnet_j} \times \frac{dnet_j}{dw_{ji}}$$

$$= \frac{dE}{do_j} \times f'(net_j) \times o_i$$



$$net_j = \sum_i o_i w_{ji}$$

$$o_j = f(net_j)$$

Note that as j is a hidden node, **we do not know its target.**
Hence, dE/do_j can only be calculated through j's **contribution to the derivative of E w.r.t net_k at the output nodes:**

$$\frac{dE}{do_j} = \sum_k w_{kj} \times \frac{dE}{dnet_k}$$

dE/dy for Other Loss Functions

Binary Cross Entropy

$$E = - [\ t \log y + (1-t) \log(1-y)\ ]$$

$$dE/dy = - [\ t/y - (1-t).1/(1-y)\ ]$$
$$= -t/y + (1-t)/(1-y)$$

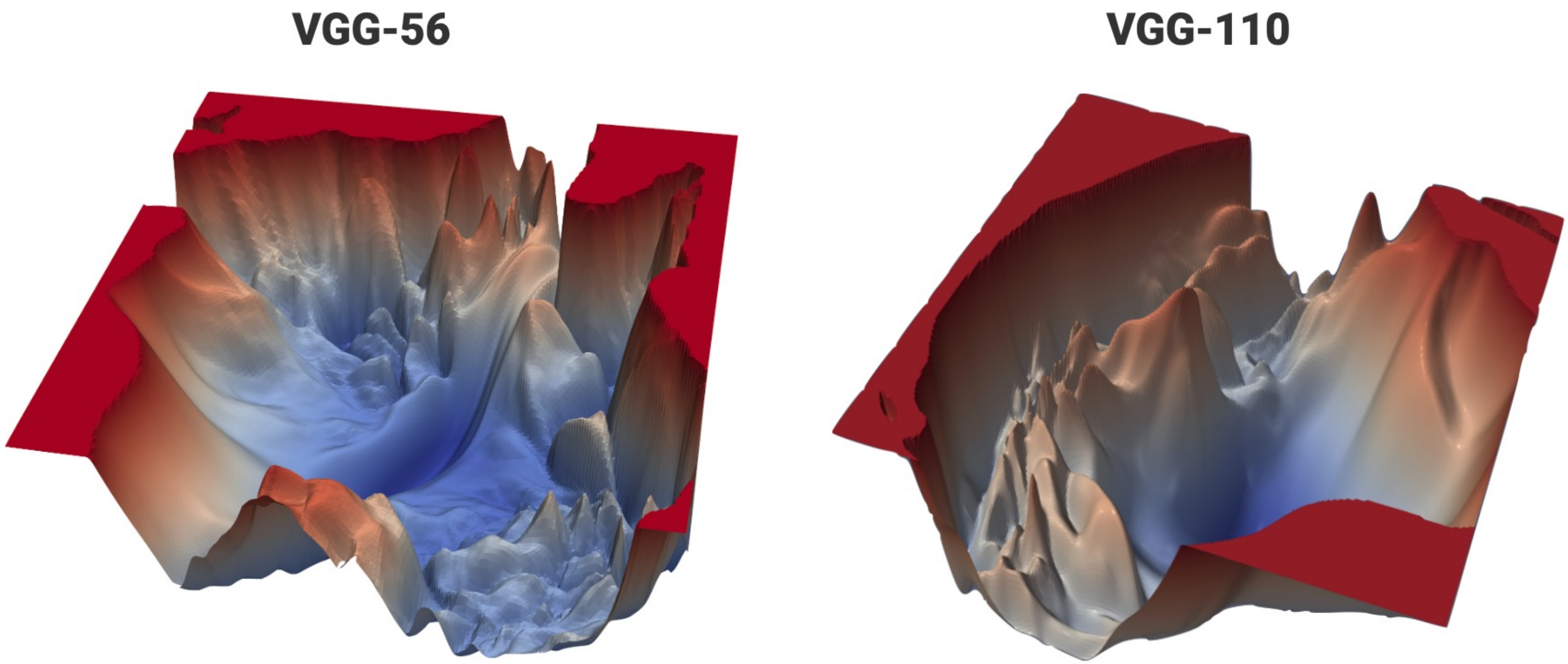
If t=1, derivative is simply -1/y.

- Since the gradient is negative, increasing y reduces the loss, which is intuitive because we want y to approach 1.

If t=0, derivative is simply 1/(1-y).

- Since the gradient is positive, decreasing y reduces the loss, which is intuitive because we want y to approach 0.

Error Landscape



Summary

- Gradient descent is the first and standard learning algorithm used in NNs
  - Finds a local minima of the error function
  - Stochastic gradient descent (SGD) can escape local minima
  - Error gradients are always computed using chain rule and propagated backwards – layer by layer

- Be able to compute gradient descent manually