Deep Learning

CS265 - Topics in Artificial Intelligence

January 31, 2018

Overview

Gradient Descent

Error Functions

Node Activation Functions

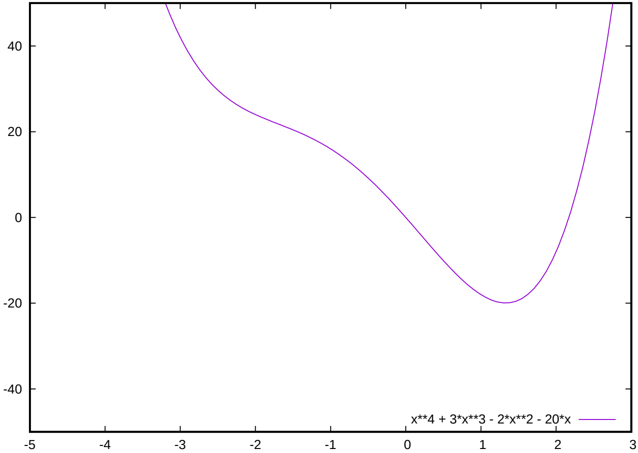
Back-Propagation

Gradient Descent

How do we find the minimum of a function?

Example

$$f(x) = x^4 + 3x^3 - 2x^2 - 20x$$



Derivative

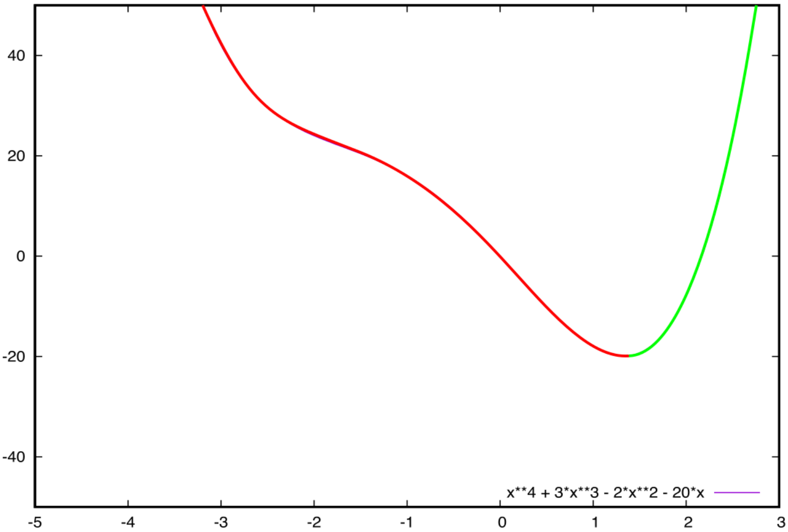
$$f(x) = x^4 + 3x^3 - 2x^2 - 20x$$

$$f'(x) = 4x^3 + 9x^2 - 4x - 20$$

Properties of the derivative:

- Indicates if the function at x if increasing or decreasing
- When $f(x)$ is at a minimum, the derivative is $f'(x) = 0$

Function and Derivative



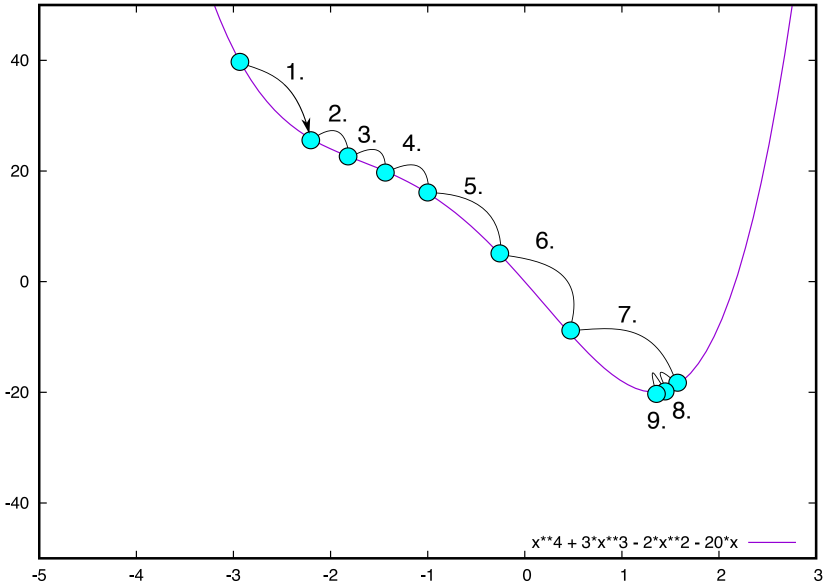
How do we find the minimum

1. Start at a random location
2. If derivative is < 0 (red), then go right
3. If derivative is > 0 (green), then go left
4. If derivative is $= 0$, stop

Refined algorithm

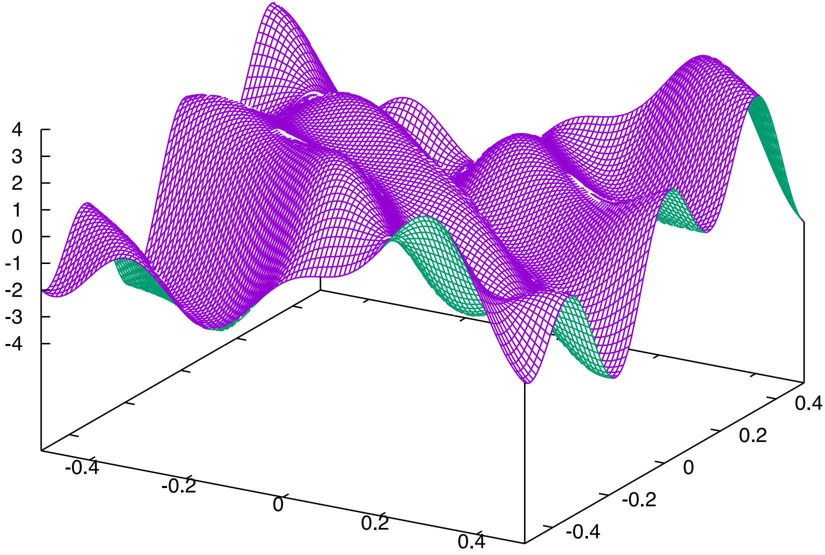
1. Start at location x_0
2. Set
   2.1 iteration counter $i = 0$
   2.2 exit condition $exit = False$
   2.3 Error threshold θ
   2.4 Learning rate η
   2.5 Maximum number of iterations $Max_{Iterations}$
3. while $exit$ is $False$
   3.1 Compute $f'(x_i)$
   3.2 Set $x_{i+1} = x_0 - \eta \cdot f'(x_i)$
   3.3 If $\|f'(x_i)\| < \theta$, set $exit = True$
   3.4 If $i > Max_{Iterations}$, set $exit = True$

Gradient descent steps



Gradient Descent in 2 dimensions

Now, we add one dimension $\mathbf{x} = (x_1, x_2)$



Gradient Descent

Extend the same algorithm to 2 dimensions

- Start at $\mathbf{x_0} = (x_0, x_1)_0$
- Compute the direction in which the function decreases
- Update $\mathbf{x_1} = \mathbf{x_0} + \eta \cdot f'(\mathbf{x_0})$

Gradient

What is $f'(\mathbf{x_0})$ for a multi-dimensional function?

Partial derivatives

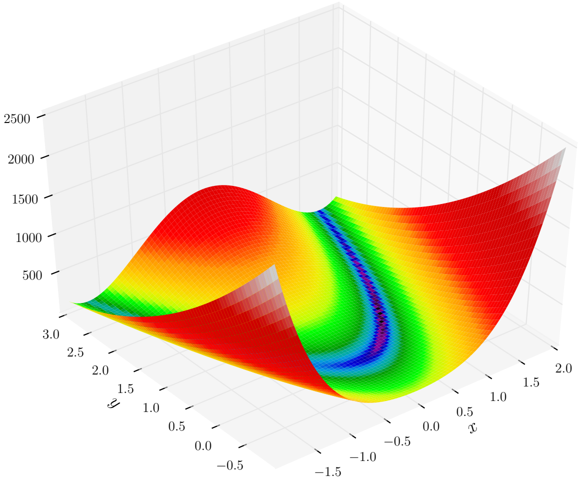
$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{pmatrix}$$

Example: Rosenbrock Function

Function is given as

$$R_{a,b}(x, y) = (a - x)^2 + b\left(y - x^2\right)^2$$

usually $a = 1,\ b = 100$



Partial derivatives

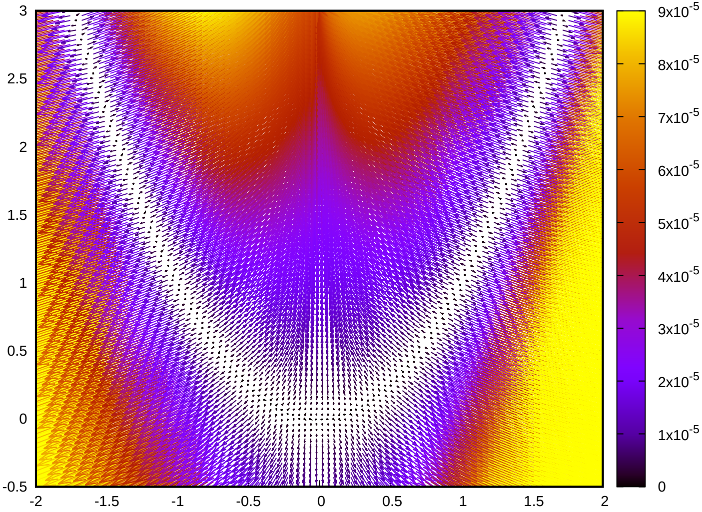
$$R_{a,b}(x,y) = (a-x)^2 + b\left(y - x^2\right)^2$$

$$\frac{\partial}{\partial x} R_{a,b}(x,y) = -2a\,(1-x) - 4bx\left(y - x^2\right)$$

$$\frac{\partial}{\partial y} R_{a,b}(x,y) = 2b\left(y - x^2\right)$$

.

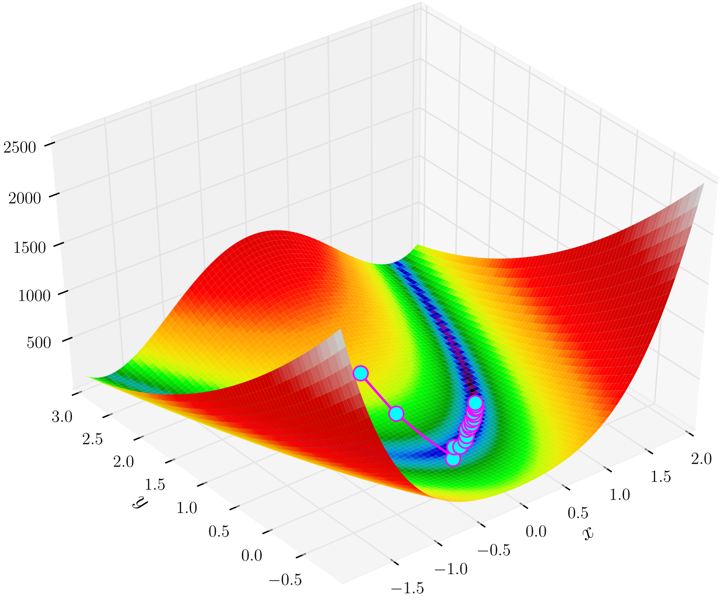
Gradient shows direction of largest increase



Gradient Descent Algorithm in n Dimensions

1. Set $\mathbf{x_0}$, θ, η, $i = 0$, $Max_{Iterations}$, $exit = False$
2. While *exit* is *False*
   2.1 set $\mathbf{x_{i+1}} = \mathbf{x_i} - \eta \cdot \nabla f(\mathbf{x_i})$
   2.2 If $\|\nabla f(\mathbf{x_i})\| < \theta$, set $exit = True$
   2.3 If $i > Max_{Iterations}$, set $exit = True$

Gradient shows direction of largest increase



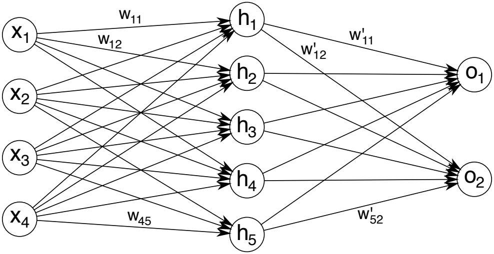
Error Functions

inNow we have a tool at our hands find local minima of a function (if the function can be differentiated).
How can we use this to train a neural network?

Error Functions

Consider a neural network with weights

$$NN_{W,W'}(x)$$



$$x \to Wx \to h(Wx) \to W'h(Wx) \to o\left(W'h(Wx)\right)$$

Error Functions

The neural network can be seen as a function with w, w' as parameters

$$NN(w_{11}, w_{12}, w_{13}, \ldots, w_{45}, w'_{11}, w'_{12}, \ldots, w'_{52}; x)$$

Now, all the weights are parameters to a multidimensional function.

Error Functions

construct function E

$$E(w_{11}, w_{12}, w_{13}, \ldots, w_{45}, w'_{11}, w'_{12}, \ldots, w'_{52}; x, y)$$

Requirements:

- Weights as parameters
- Input value x
- Target output y
- Function E is differentiable
- The function value is minimal when an element is correctly classified

Error Functions

- E is called error function, or loss function
- Distance from the network output to the target output

$$E(w_{11}, \ldots, w'_{52}; x, y) = d\left(y,\ NN(w_{11}, \ldots, w'_{52}, x)\right)$$

- If we find the w that minimize E, the network output is closest to the target output
- Gradient Descent to find the values of w that minimize E

Some popular error functions

Quadratic loss function

- Output layer activations

$$o_j(x) = NN(w_{11}, \ldots, x)_j$$

- $o_j(x)$ is the value of node j in the output layer of the neural network
- Quadratic loss function is now

$$E_{QL}(w_{11}, \ldots; x, y) = \frac{1}{2} \sum_j (o_j(x) - y_j)^2$$

- The square is positive
- Zero when the network output equals the target
- The further from the target, the larger the error
- Output can be anything

Some popular error functions

Cross entropy loss function

- Output layer activations $o_j(x)$ are normalized using softmax

$$s\left(o_j(x)\right) = \frac{\exp\left(o_j(x)\right)}{\sum_k \exp\left(o_k(x)\right)}$$

- Exponentiate each output and divide it by the sum of all output
  - strictly positive
  - bounded by 1
  - The sum of all node outputs is 1
- Thus, the output can be treated as a probability
- Cross entropy is a distance measure between probability distributions

Cross entropy

- The network output are now probabilities for classes.
- $s(o_j(x))$ indicates the probability that element x belongs to class C_j
- The true class is C_i, encoded as one-hot

$$y = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$$

- Cross entropy loss is now

$$L_{CE} = -\log \prod_j s(o_j(x))^{y_j} = -\sum_j \log(s(o_j(x))) \cdot y_j$$

- Mathematically well founded

Cross Entropy

- The cross-entropy loss is basically never used directly
- Instead, consider softmax output layer and cross-entropy loss together
- Consider the derivative of the cross entropy loss wrt. to the input of the softmax layer

$$\begin{aligned}\frac{\partial}{\partial o_j(x)} L_{CE} &= -\sum_j \frac{\partial}{\partial o_j(x)} \log\left(s\left(o_j(x)\right)\right) \cdot y_j = -\sum_j y_j \frac{\partial}{\partial o_j(x)} \cdots \\ &= \ldots \text{ (lines of boring math)} \\ &= s\left(o_j(x)\right) - y_j\end{aligned}$$

- Using softmax and cross entropy loss results makes the error gradient very simple:
  - The difference between the softmax output and the target

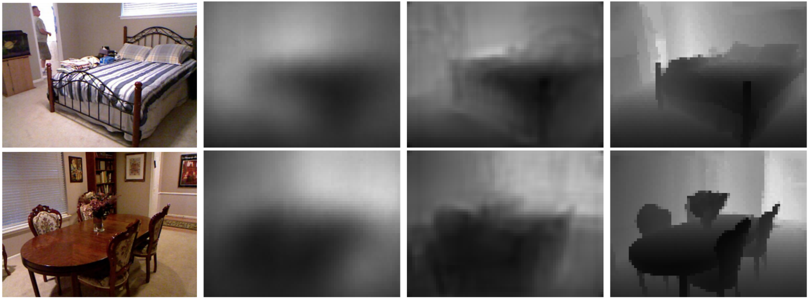
Error Functions

Usage hints:

- When target values should be unbounded $\rightarrow$ Quadratic Loss
- When target values represent a class probabilities (classification) $\rightarrow$ Cross Entropy Loss
- Other loss functions exist, but are less popular

Example Quadratic Loss

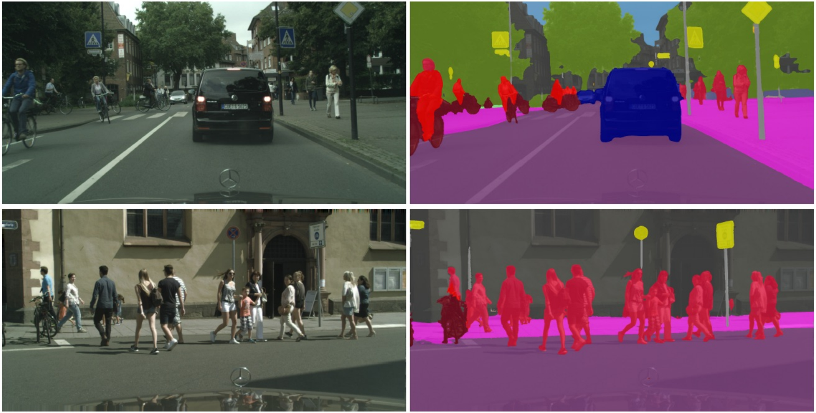
Estimate depth of each pixel in an image



Input Image | Coarse and refined output | target output

Example Cross Entropy Loss

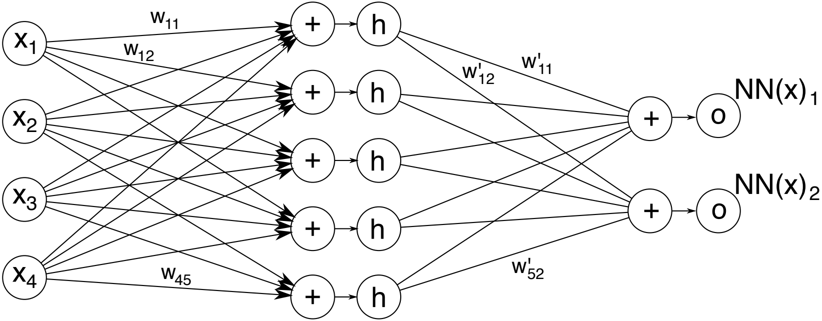
Estimate class of each pixel in an image



Node Activation Functions

Let's look at a neural network in greater detail.

Network Overview



Non-Linear activation functions

Activation functions

- non-linear to make the network more powerful
- Need to be differentiable
- symmetric activation functions can be useful for some applications
- The derivative must be easy (runtime)

Binary Step Function

Activation:

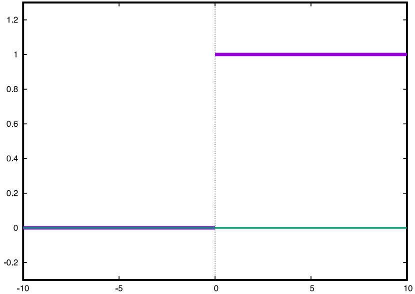
$$bsf = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Derivative:

$$bsf'(x) = 0$$

Min/Max:

$$[0 : 1]$$



Binary Step Function

Activation:

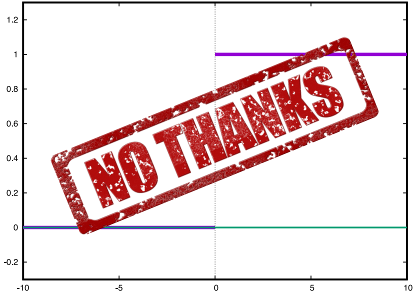
$$bsf = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Derivative:

$$bsf'(x) = 0$$

Min/Max:

$$[0:1]$$



Logistic Sigmoid

Activation:

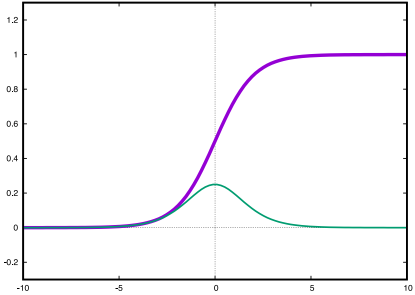
$$sig(x) = \frac{1}{1+e^{-x}}$$

Derivative:

$$sig'(x) = \frac{1}{1+e^{x}} - \left(\frac{1}{1+e^{x}}\right)^2$$
$$= sig(x) \cdot (1 - sig(x))$$

Min/Max:

$$[0:1]$$



Tanh Function

Activation:

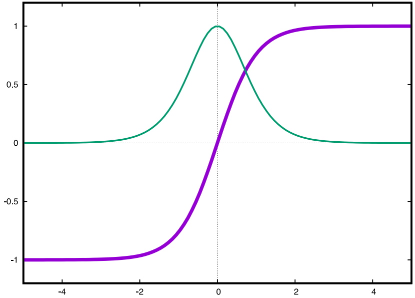
$$
\begin{aligned}
tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
&= 2 \cdot sig(2x) - 1
\end{aligned}
$$

Derivative:

$$tanh'(x) = 1 - tanh(x)^2$$

Min/Max:

$$[-1 : 1]$$



Rectified Linear Unit

Activation:

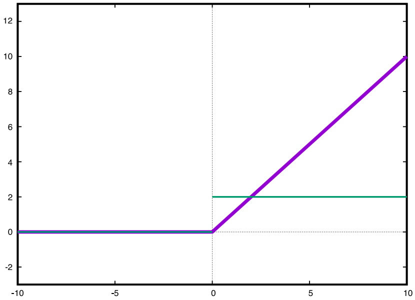
$$\text{ReLU}(x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Derivative:

$$\text{ReLU'}(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Min/Max:

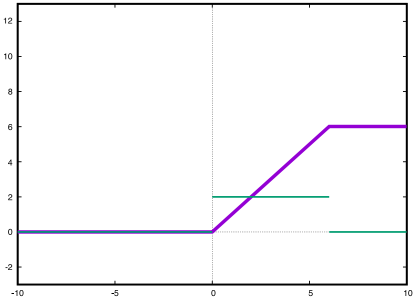
$$[0 : \infty]$$



Rectified Linear Unit 6

Activation:

$$\mathrm{ReLU}(x) = \begin{cases} x & 0 < x < 6 \\ 0 & \text{otherwise} \end{cases}$$



Derivative:

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & 0 < x < 6 \\ 0 & \text{otherwise} \end{cases}$$

Min/Max:

$$[0 : 6]$$

Leaky ReLU Function

Activation:

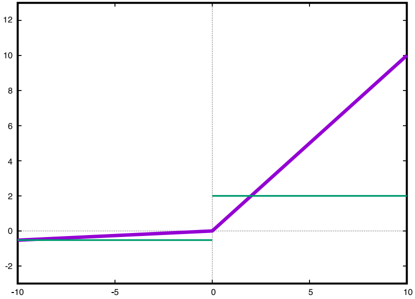
$$\mathrm{LReLU}_\alpha(x) = \begin{cases} x & 0 < x \\ -\alpha x & \text{otherwise} \end{cases}$$

Derivative:

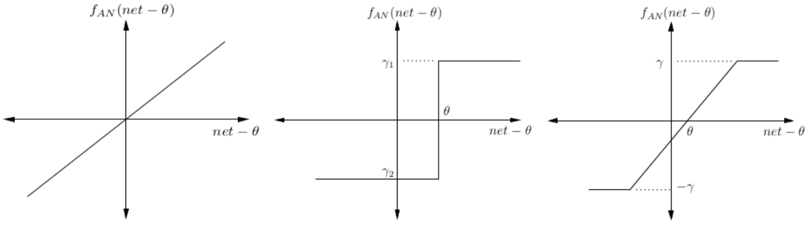
$$\mathrm{LReLU'}_\alpha(x) = \begin{cases} 1 & 0 < x \\ -\alpha & \text{otherwise} \end{cases}$$

Min/Max:

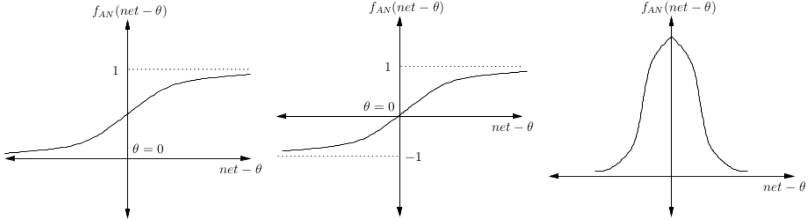
$$[-\infty : \infty]$$



Overview Activation Functions



(a) Linear function (b) Step function (c) Ramp function



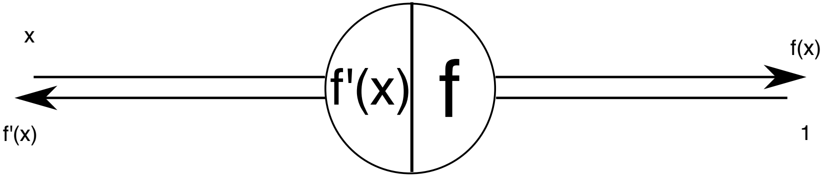
(d) Sigmoid function (e) Hyperbolic tangent function (f) Gaussian function

Back-Propagation

How to compute $\frac{\partial}{\partial w} Error$ for each weight w in the network?

A helpful notation for function application and derivatives

Applying a function: left to right



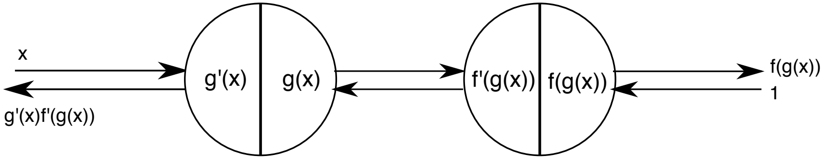
Derivative:

- Right to left
- Start with value 1
- Multiply all nodes along the way

A helpful notation for function application and derivatives

Chaining 2 functions

$$(f\,(g(x)))' = f'\,(g(x)) \cdot g'(x)$$



Derivative:

- Start with value 1
- Multiply all nodes along the way

Basic rules of derivatives

Addition $$\frac{\partial}{\partial x}(x+y) = 1$$

Multiplication $$\frac{\partial}{\partial x} xy = x$$
$$\frac{\partial}{\partial x} xy = x$$

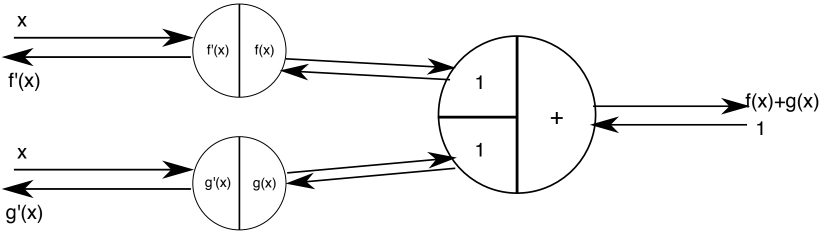
Composition $$\frac{\partial}{\partial x} f\left(g(x)\right) = \frac{\partial}{\partial g(x)} f\left(g(x)\right) \frac{\partial}{\partial x} g(x)$$

$$\left(f\left(g(x)\right)\right)' = f'\left(g(x)\right) g'(x)$$

A helpful notation for function application and derivatives

Addition

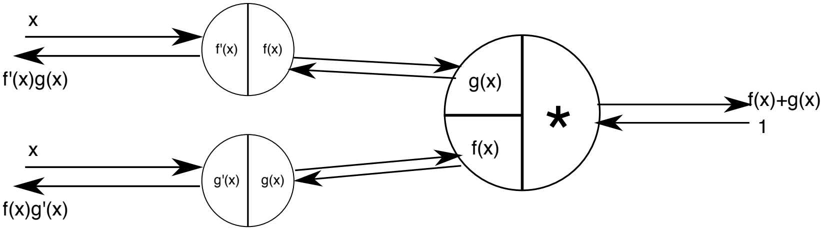
$$\frac{\partial}{\partial a}(a+b) = 1 \quad \frac{\partial}{\partial b}(a+b) = 1$$



A helpful notation for function application and derivatives

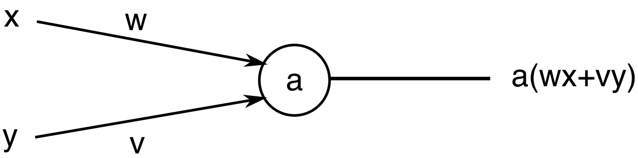
Multiplication

$$\frac{\partial}{\partial a}ab = a \quad \frac{\partial}{\partial b}ab = a$$



A helpful notation for function application and derivatives

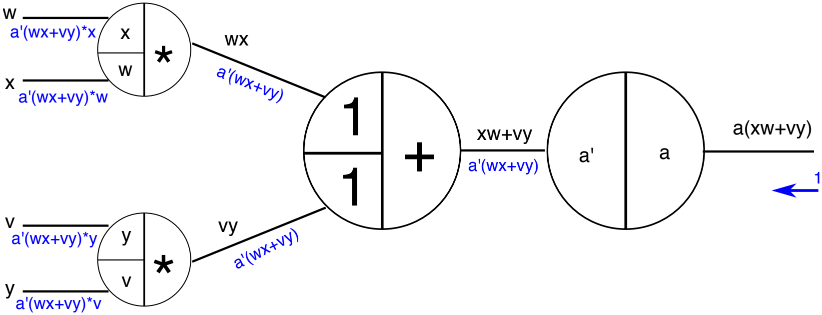
A simple example:
A network with 2 inputs x, y two weights v, w and an activation function a to compute $a(xw + vy)$



A helpful notation for function application and derivatives

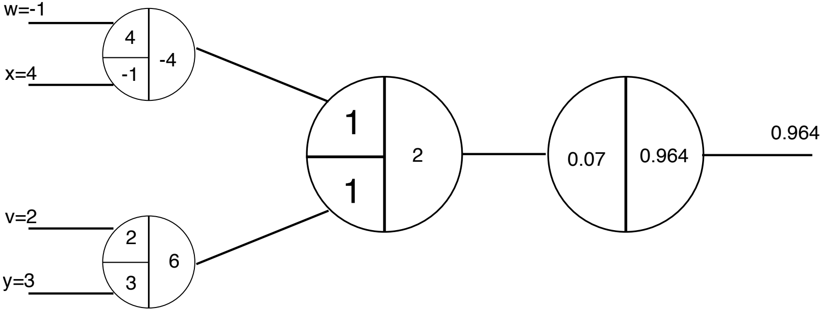
A simple example:

A network with 2 inputs x, y two weights v, w and an activation function a to compute $a(xw + vy)$



A helpful notation for function application and derivatives

A simple example:
A network with 2 inputs x, y two weights v, w and an activation function a to compute $a(xw + vy)$
Let $x = 1$, $y = 3$, $w = -1$, $v = 2$, $a = \tanh$, target output $= 0$

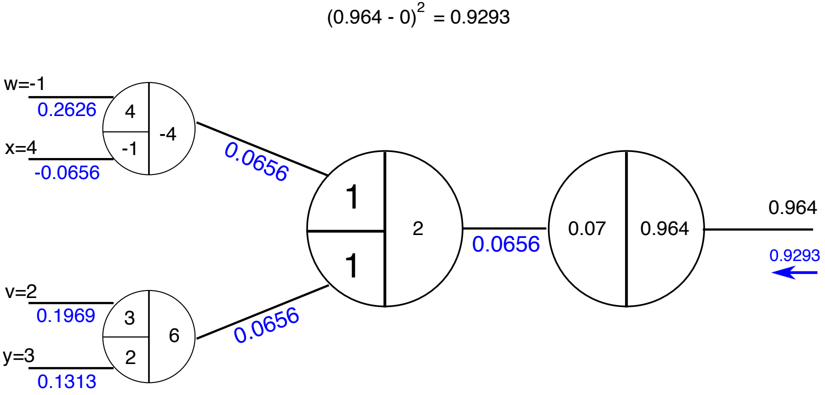


A helpful notation for function application and derivatives

A simple example:

A network with 2 inputs x, y two weights v, w and an activation function a to compute $a(xw + vy)$

Let $x = 1$, $y = 3$, $w = -1$, $v = 2$, $a = \tanh$, target output $= 0$



$$\frac{\partial}{\partial w} Error(x, y) = 0.262 \quad \frac{\partial}{\partial v} Error(x, y) = 0.1969$$

Back-propagation

A simple example:
A network with 2 inputs x, y two weights v, w and an activation function a to compute $a(xw + vy)$
Let $x = 1$, $y = 3$, $w = -1$, $v = 2$, $a = \tanh$, target output $= 0$

$$\frac{\partial}{\partial w} Error(x, y) = 0.262 \quad \frac{\partial}{\partial v} Error(x, y) = 0.1969$$

Following the gradient descent approach, and a learning rate of $\eta = 0.1$, the new weights are

$$\begin{aligned} w' &\leftarrow w - \eta \frac{\partial}{\partial w} Error(x, y) && = -1 - 0.1 \cdot 0.262 = -1.0262 \\ v' &\leftarrow v - \eta \frac{\partial}{\partial v} Error(x, y) && = 2 - 0.1 \cdot 0.197 = 1.98 \end{aligned}$$

Back-propagation recap

The basic back-propagation steps are

1. Do the forward pass given a sample (x, y), i.e. evaluating
   1.1 the functions at each node
   1.2 the derivatives of the function at each node
2. Compute the error at the output layer $E(x)$
3. Feed the error value into the right side of the network
4. Compute the error gradient at each weight
   - The output activation of the node at the lower layer times the gradient at the input of the higher layer
5. Update the weights

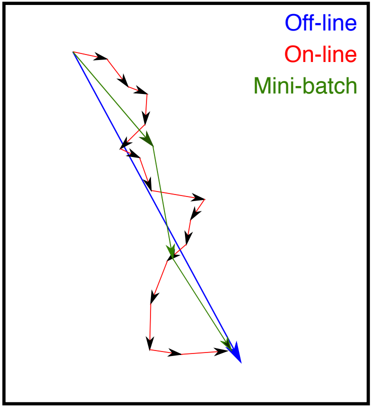
$$w_{i+1} = w_i - \eta \frac{\partial}{\partial w} E(x)$$

Back-propagation strategies

Off-line Sum the gradients for each sample of the training set. Do one weight update after seeing the entire training set

On-line Perform the weight update after each forward-backward pass

Mini-batch Do the forward-backward pass for a small subset (usually up to a few hundred) before updating the weights.



Back-propagation strategies

Off-line

- (+) Robust against noise in the training set
- (-) Slow convergence

On-line

- (+) Faster convergence
- (-) Very sensitive to noise (outliers)
- (-) Not well suited to find the global minimum

Mini-batch

A good compromise between off-line and on-line

- (+) Faster than off-line learning
- (+) Better residual error than on-line learning
- (+) Suitable for parallelization (batch size can be tuned to architecture)

Training Strategies

Momentum

- Similarly to a ball rolling down a hill, a momentum parameter can be introduced to stabilize the gradient descent
- recall weight update

$$W_{i+1} = W_i - \eta \nabla E(x)$$

- Now, with Momentum

$$V_{i+1} = \gamma V_i + \eta \nabla E(x)$$
$$W_{i+1} = W_i - V_i$$

- Momentum term γusually close to 1 (e.g. 0.9)
- Reduces Oscillations
- Faster, stable convergence

Training Strategies

Learning Rate

- The learning rate is not easy to optimize
  - high values, the step size can be too large $\rightarrow$ difficult to find the minimum
  - low values, the algorithm takes too long
- Often we want a high learning rate in the beginning and a lower learning rate afterwards
- Adagrad, AdaDelta, and ADAM are strategies to dynamically adapt the learning rate
- AdaDelta and ADAM do not even need a default learning rate

The End

Questions?