Carnegie Mellon University

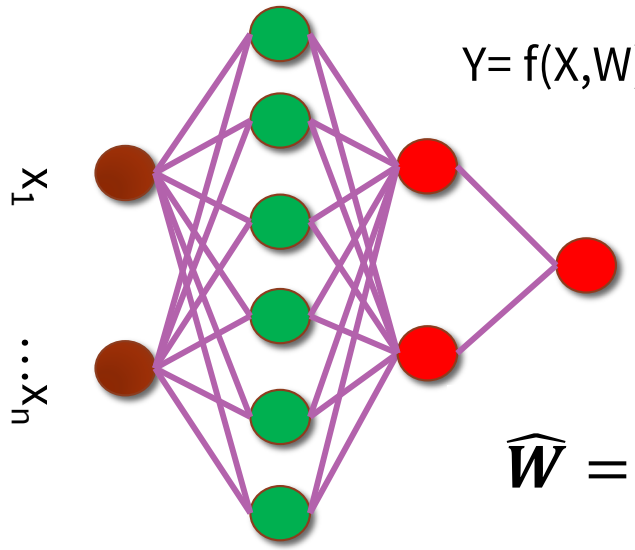
Introduction to Deep Learning for Engineers

Spring 2025, Introduction to Deep Learning for Engineers
Jan 21, 2025, Third Session

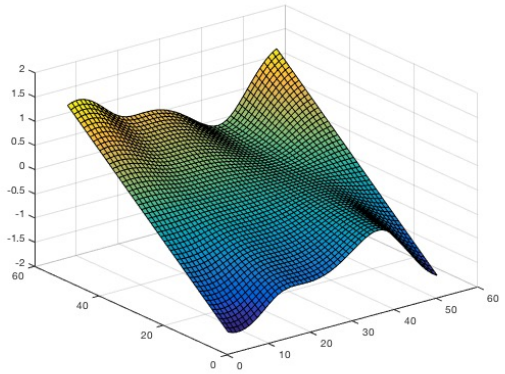
Amir Barati Farimani
Associate Professor of Mechanical Engineering and Bio-Engineering
Carnegie Mellon University

Recap: Automatic Parametrization

More generally, given the function to model, we can derive the parameters of the network to model it, through computation

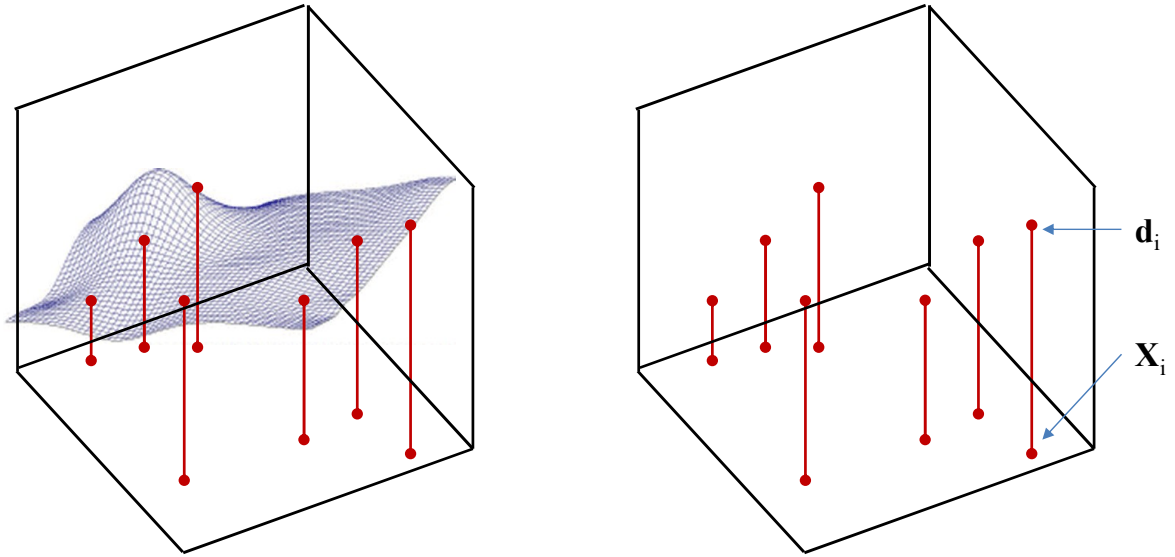


G(X)



$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_{X} div\big(f(X;W), g(X)\big) dX$$

Learning from Samples



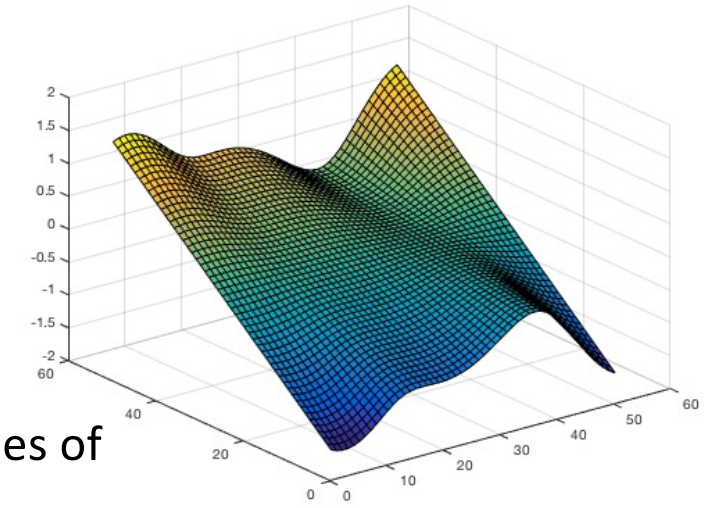
- We must ***learn*** the *entire* function from these few examples
  - The "training" samples

g(X) is unknown

Function must be fully specified – Known everywhere, i.e. for every input

- In practice we will not have such specification

23



Sample

– Basically, get input-output pairs for a number of samples of input

– Good sampling: the samples of will be drawn from

- Very easy to do in most problems: just gather training data – E.g. set of images and their class labels

– E.g. speech recordings and their transcription

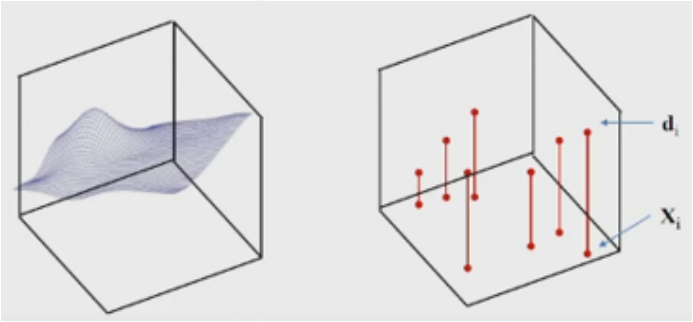
What we have learnt so far

•Previous discussion showed that a single-layer MLP is a universal function approximator

• Can approximate any function to arbitrary precision – But may require infinite neurons in the layer

• More generally, deeper networks will require far fewer neurons for the same approximation error

The Empirical risk



- The empirical estimate of the expected error is the average error over the samples

$$E[div(f(X;W),g(X))] \simeq \frac{1}{T}\sum_{i=1}^{T} div(f(X_i;W),d_i)$$

- This approximation is an unbiased estimate of the expected divergence that we actually want to estimate
  -- Assumption: minimizing the empirical loss will minimize the true loss

Problem Statement

- Given a training set of input-output pairs

(X_1, d_1), (X_2, d_2), …, (X_r, d_r),

- Minimum the following function

$$Loss(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$, w.r.t W

- This is problem of function minimization
  --An instance of optimization

Problem Setup: Things to define

- Given a training set of input-output pairs

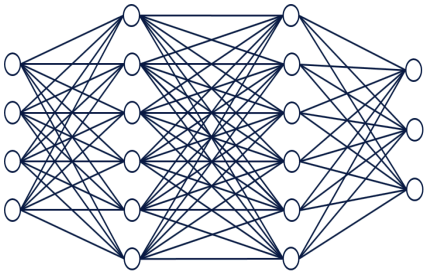
$(X_1, d_1), (X_2, d_2), \ldots, (X_r, d_r),$

what are input-output pairs?

$$Loss(W) = \frac{1}{T} \sum_i div(f(X_i; W), d_i)$$

What is the divergence div()?
What is f() and what are its parameters W?

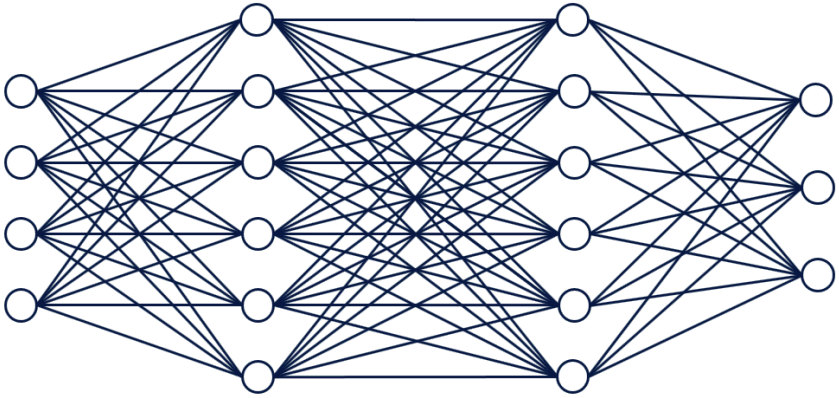
What is f()? Typical network



- Multi-layer perceptron
- A *directed* network with a set of inputs and outputs
  -- No loops
- Generic terminology
  -- We will refer to the inputs as the *input units*
    - No neurons here – the “input units” are just the units
  -- We refer to the outputs as the output units
  -- Intermediate units are “hidden” units

Typical network

- We assume a “layered” network for simplicity
  -- We will refer to the inputs as the *input layer*
    - No neurons here – the “layer” simply refers to inputs
  -- We refer to the outputs as the output layer
  -- Intermediate layer are “hidden” layer



The individual neurons

- Individual neurons operate on a set of inputs and produce a single output assume a "layered" network for simplicity

-- Standard setup: A differentiable activation function applied to an affine combination of the input

$$y = f\left(\sum_i w_i x_i + b\right)$$

We will assume this unless otherwise specified

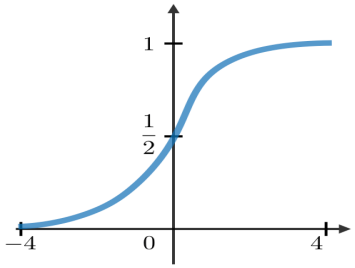
Parameters are weights w_i and bias b

-- More generally: aby differentiable function

$$y = f(x_1, x_2, \ldots, x_n; w)$$

Activation and their derivatives

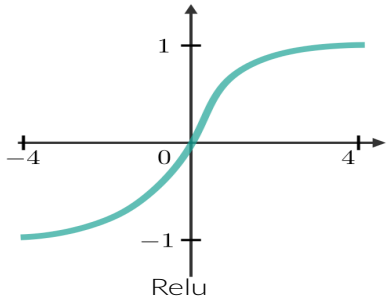
Sigmoid



$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$\acute{f}(z) = f(z)\big(1 - f(z)\big)$$

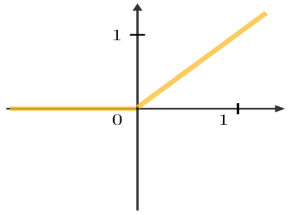
Tanh



$$f(z) = \tanh(z)$$

$$\acute{f}(z) = \big(1 - f^2(x)\big)$$

Relu



$$f(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$

$$\acute{f}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

Some popular activation functions and their derivatives

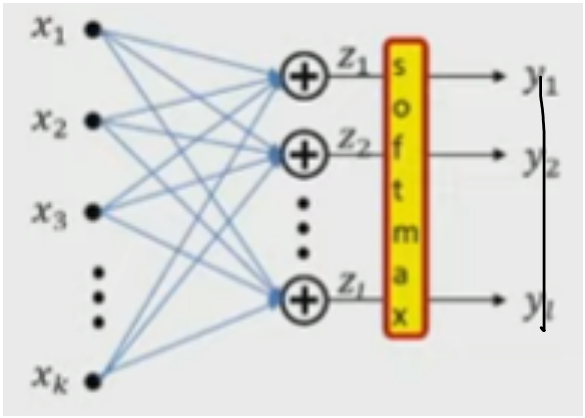
Vector activation example; Softmax

- Example: Softmax vector activation

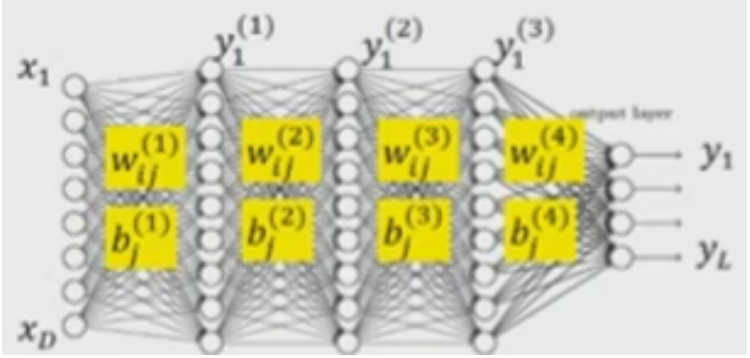
$$z_i = \sum_j w_{ji} x_j + b_i$$

Parameters are weights w_{ji} and bias b_i

$$y = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$



Notation



- The input layer is the layer
- We will represent the output of the i-th perceptron of the k^{th} layer as $y_i^{(k)}$.

  -- Input to network: $y_i^{(0)} = x_i$

  -- Output of network: $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i-th unit of the k-1th layer and the jth unit of the k-th layer as $w_{ij}^{(k)}$

  -- The bias to the jth unit of the k-th layer is $b_j^{(k)}$

Problem Setup: Things to define

- Given a training set of input-output pairs

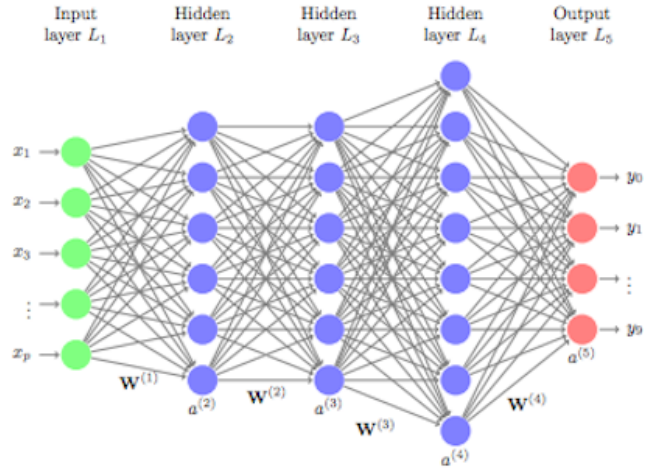
$(X_1, d_1), (X_2, d_2), \ldots, (X_r, d_r),$

- what are input-output pairs?

$$Loss(W) = \frac{1}{T} \sum_i div(f(X_i; W), d_i)$$

Vector notation

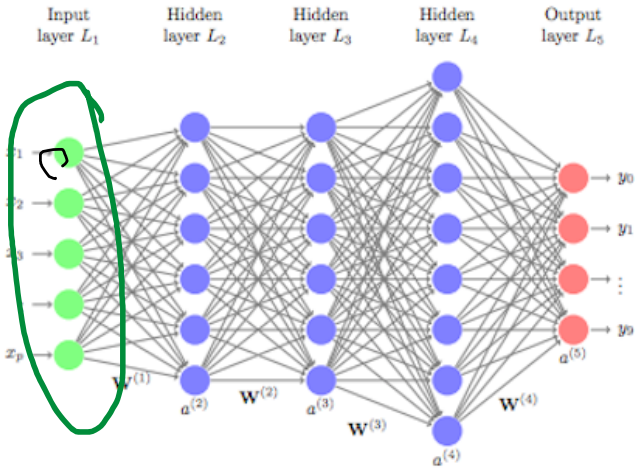
- Given a training set of input-output pairs (X_1, d_1), (X_2, d_2), …, (X_r, d_r)
- $X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]$ is the nth vector of actual outputs of the network
- We will sometimes drop the first subscript when referring to a specific instance



Carnegie Mellon University

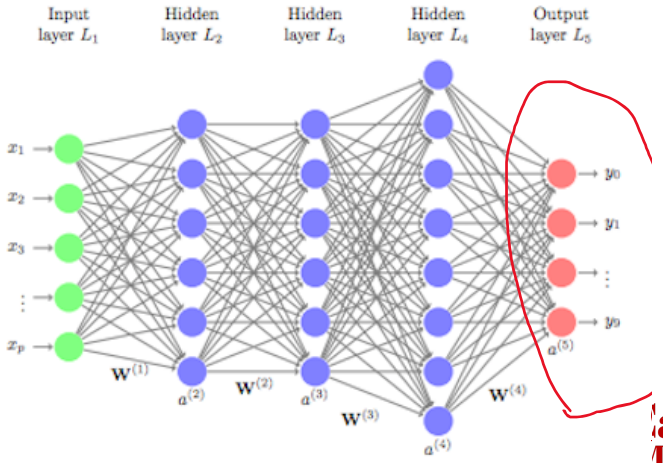
Representing the Input

- Vectors of numbers
  -- (or may even be just a scalar, if input layer is of size 1)
  -- e.g. vector of pixel values
  -- e.g. vector of speech features
  -- e.g. real-valued vector representing text
    - We will see hoe this happens later in the course
  -- Other real valued vectors



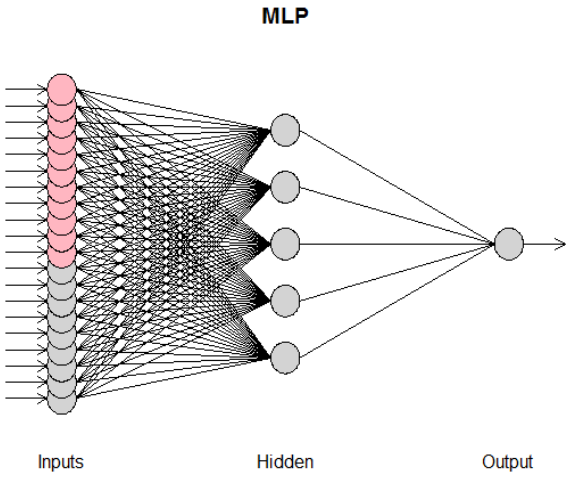
Representing the Output

- If the desired output is real-valued , no special tricks are necessary
  -- Scalar Output : single output neuron
    - d = scalar (real value)
  -- Vector Output : as many output neurons as the dimension of the desire output
    - $d = [d_1, d_2, d_l]$ (vector of real values)



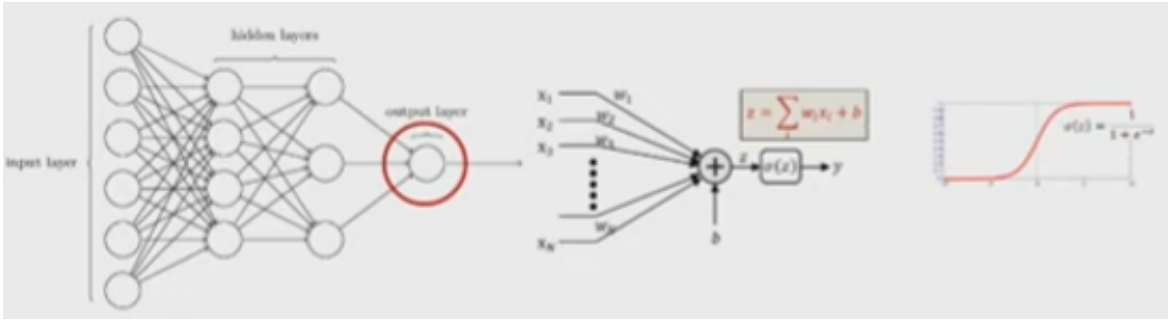
Representing the Output

- If the desired output is binary (is this a cat or not),use a simple 1/0 representation of the desired output
  -- 1 = Yes, it's a cat
  -- 0 = No, it's not a cat



Representing the Output

- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
- Output activation: Typically, a sigmoid
  -- Viewed as the probability P(Y = 1/X) of class value 1
  - Indicating the fact that for actual data, in general a feature value X may occur for both classes, but with different probabilities
  - Is differentiable



Multi-class output: One-hot representations

- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector:

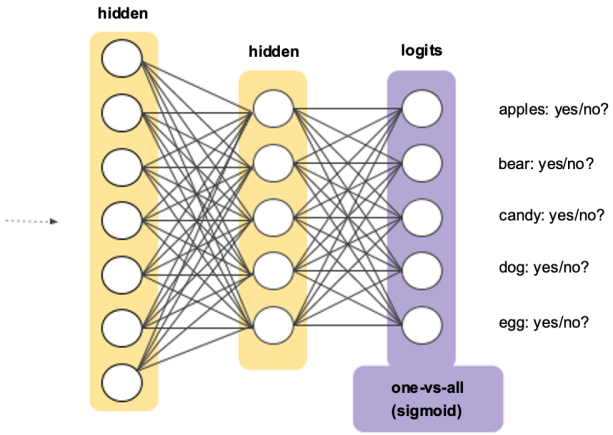
$$[cat\ \ dog\ \ camel\ hat\ \ flower]^T$$

- For inputs of each of the five classes the desired output is:

  cat: $[1\ 0\ 0\ 0\ 0]^T$
  dog: $[0\ 1\ 0\ 0\ 0]^T$
  camel: $[0\ 0\ 1\ 0\ 0]^T$
  hat: $[0\ 0\ 0\ 1\ 0]^T$
  flower: $[0\ 0\ 0\ 0\ 1]^T$

- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class
- This is a one hot vector

Multi-class network



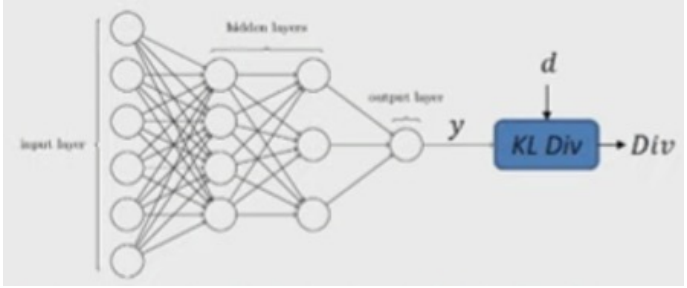
- For a multi-class classifier with N classes, the one-hot representation will have N binary outputs
  -- An N-dimensional binary vector
- The neural network's output too must ideally be binary (N-1 zeros and a single 1 in the right place)
- More realistically, it will be a probability vector
  -- N probability values that sum to 1

For binary classifier

- For binary classifier with scalar output, $Y \in (0,1)$, d is 0/1, the cross entropy between the probability distribution $[Y, 1-Y]$ and the ideal output probability $[d, 1-d]$ is popular

$$Div(Y, d) = -\mathrm{d}\log \mathrm{Y} - (1 - \mathrm{d})\log(1 - \mathrm{Y})$$

-- Minimum when d = Y



- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} \frac{-1}{Y} \; if \; d = 1 \\ \frac{1}{1-Y} \; if \; d = 0 \end{cases}$$

Note: when y=d the derivative is not 0
Even though div()=0 (minimum) when y=d

Cross entropy vs L2

$$Div(Y, d) = -\mathrm{d}\log \mathrm{Y} - (1 - \mathrm{d})\log(1 - \mathrm{Y})$$

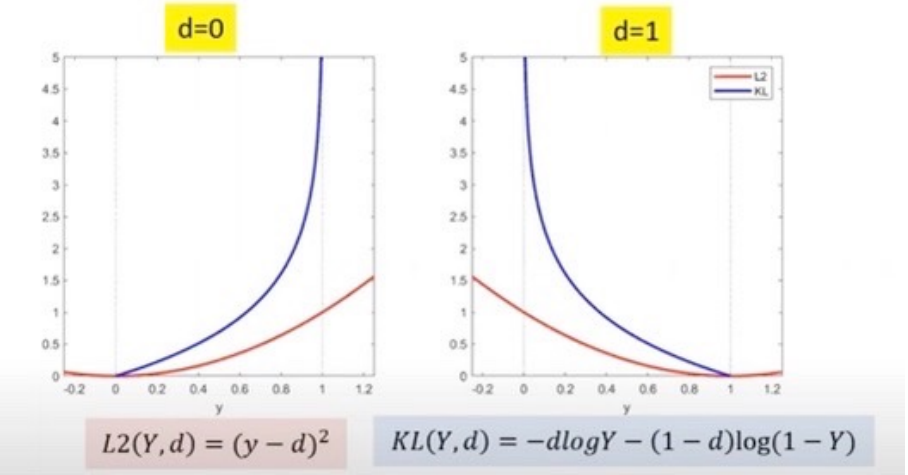
-- Minimum when d = Y

- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} \frac{-1}{Y} \; if \; d = 1 \\ \frac{1}{1-Y} \; if \; d = 0 \end{cases}$$

$$Div(Y, d) = \frac{1}{2}\|Y - d\|^2 = \frac{1}{2}\sum_i (y_i - d_i)^2$$

$$\frac{dDiv(Y, d)}{dy_i} = (y_i - d_i)$$



Note: when y=d the derivative is not 0
Even though div()=0 (minimum) when y=d

For multi-class classification

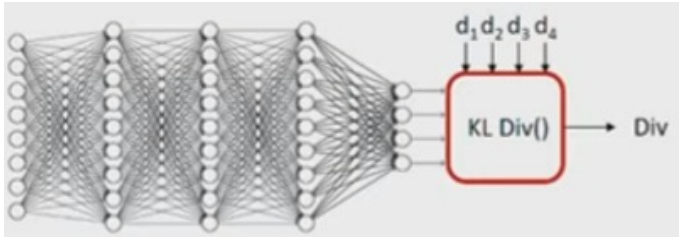
- Desired output d is a one hot vector [0 0 0… 1 …0 0 0] with the 1 in the c-th position {for class c}
- Actual output will be probability distribution $[y_1, y_2, \ldots]$
- The cross-entropy between the desired one-hot output and actual output:

$$Div(Y,d) = -\sum_i d_i \log y_i = -\log y_c$$

- Derivative

$$\frac{dDiv(Y,d)}{dY_i} = \begin{cases} \frac{-1}{y_c} & for\ the\ \mathrm{c}-th\ component \\ 0 & for\ remaining\ component \end{cases}$$

$$\nabla_Y Div(Y,d) = \left[0\ 0\ \ldots\ \frac{-1}{y_c}\ \ldots 0\ 0\right]$$

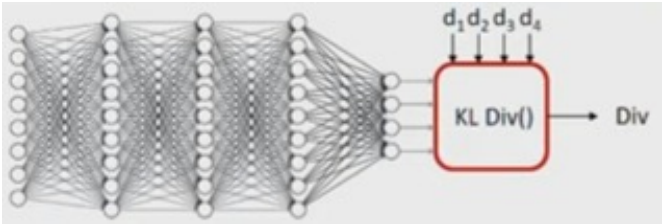


If y_c <1, the slop is negative w.r.t. y_c
Indicates increasing y_c will reduce
Note: when y=d the derivative is not 0
Even though div()=0 (minimum) when y=d

For multi-class classification

- It is sometimes useful to set the target output to [ϵ ϵ … (1-(K-1) ϵ) … ϵ ϵ ϵ] with the value 1-(K-1) ϵ in the c-th position (for class c) and ϵ elsewhere for some small ϵ
  -- "Label smoothing" aids gradient descent

The cross-entropy remains: $Div(Y, d) = -\sum_i d_i \log y_i$



- Derivative

$$\frac{dDiv(Y, d)}{dY_i} = \begin{cases} -\frac{1-(K-1)\epsilon}{y_c} & for\ the\ c-th\ component \\ -\frac{\epsilon}{y_i} & for\ remaining\ component \end{cases}$$

Problem Setup

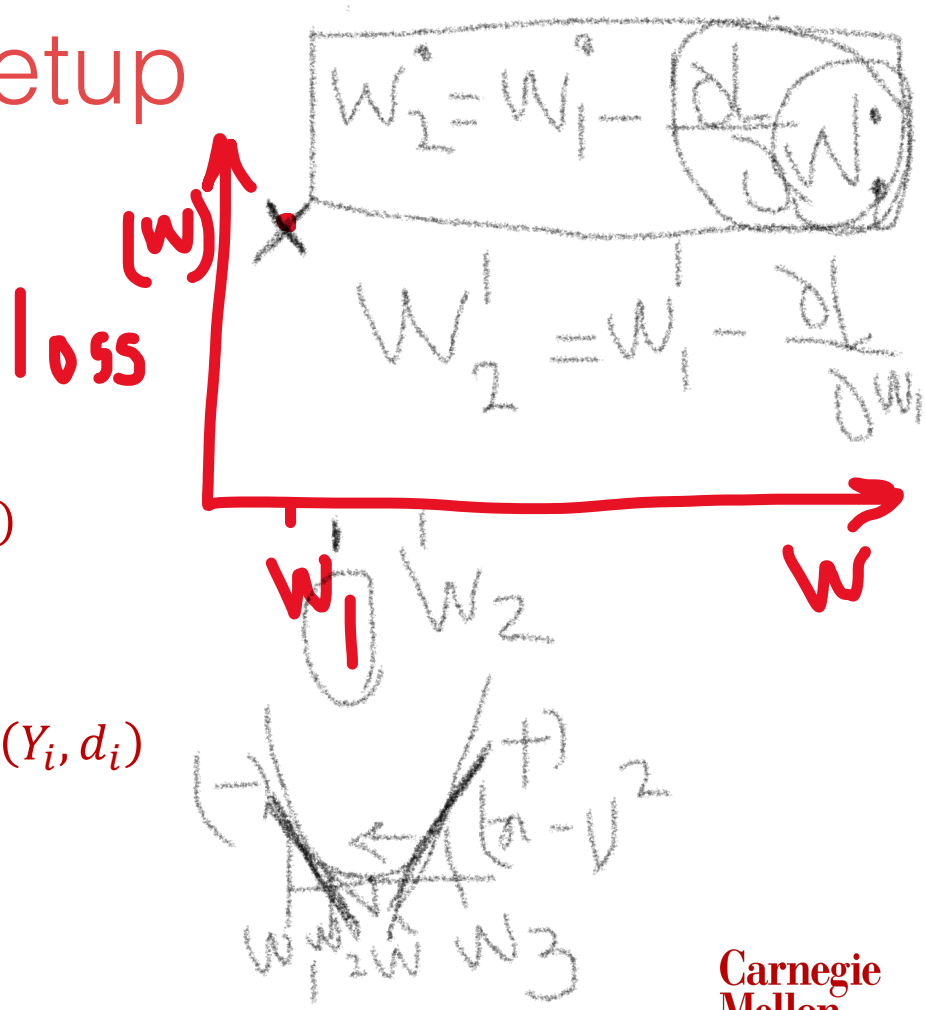
- Given a training set of input-output pairs

$(X_1, d_1), (X_2, d_2), \ldots, (X_r, d_r),$

- The error on the i-th instance is $div(Y_i, d_i)$
- The loss

$$Loss = \frac{1}{T}\sum_i div(Y_i, d_i)$$

- Minimize Loss w.r.t. $\{w_{ij}^{(k)}, b_j^{(k)}\}$



Recap: Gradient Descent Algorithm

- Initialize:
  -- x^0
  -- k = 0

- Do
  -- $x^{k+1} = x^k - \eta^k \nabla f(x^k)^r$
  -- k = k+1

- While $\left|f\left(x^k\right) - f(x^{k-1})\right| > \varepsilon$

Recap: Gradient Descent Algorithm

- In order to minimize any function f(x) w.r.t. x
- Initialize:

  -- x^0

  -- k = 0

- Do

  -- for every component i

  - $x_i^{k+1} = x_i^k - \eta^k \frac{df}{dx_i}$ Explicitly stating it by component

  -- k = k+1

- While $\left|f\left(x^k\right) - f(x^{k-1})\right| > \varepsilon$

Carnegie Mellon University

Training Neural Nets through Gradient Descent

- Total training Loss:

$$Loss = \frac{1}{T}\sum_{t} Div(Y_t, d_t)$$

- Gradient descent algorithm:
- Initialize all weights and biases $\{w_{ij}^{(k)}\}$

  -- Using the extended notation: the bias is also a weight
- Do:

  -- For every layer k for all I,j, update:

$$w_{ij}^{(k)} = w_{ij}^{(k)} - \eta \frac{dLoss}{dw_{ij}^{(k)}}$$

- Until Loss has converged

Assuming the bias is also represented as a weight

Training Neural Nets through Gradient Descent

- Total training Loss:

$$Loss = \frac{1}{T}\sum_{t} Div(Y_t, d_t)$$

- Gradient descent algorithm:
- Initialize all weights $\{w_{ij}^{(k)}\}$
- Do:
  -- For every layer k for all i,j, update:

$$w_{ij}^{(k)} = w_{ij}^{(k)} - \eta \frac{dLoss}{dw_{ij}^{(k)}}$$

- Until Err has converged

The derivative

Total training Loss:

$$\boldsymbol{Loss} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

- Computing the derivative

Total derivative:

$$\frac{\boldsymbol{dLoss}}{\boldsymbol{dw_{ij}^{(k)}}} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \frac{\boldsymbol{dDiv}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{\boldsymbol{dw_{ij}^{(k)}}}$$

The derivative

Total training Loss:

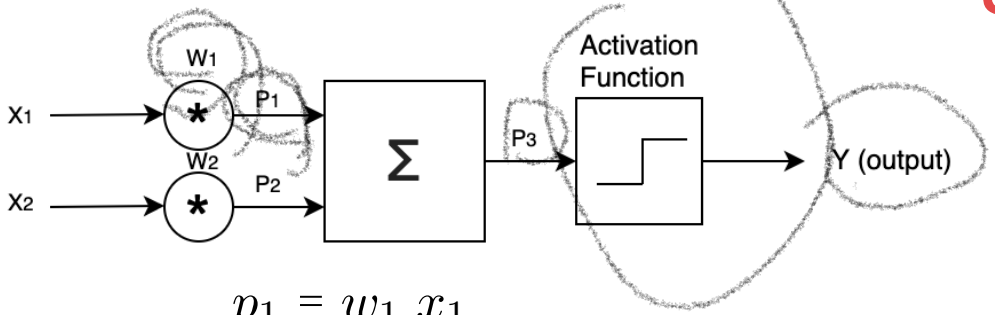
$$\boldsymbol{Loss} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

Total derivative:

$$\frac{\boldsymbol{dLoss}}{\boldsymbol{dw_{ij}^{(k)}}} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \frac{\boldsymbol{dDiv}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{\boldsymbol{dw_{ij}^{(k)}}}$$

- So, we must first figure out how to compute the derivative of divergences of individual training inputs

Back Propagation

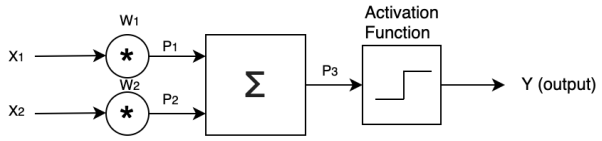


$$p_1 = w_1 \, x_1$$
$$p_2 = w_2 \, x_2$$
$$p_3 = p_1 + p_2 = w_1 \, x_1 + w_2 \, x_2$$

In order to obtain the partial derivative of cost function L with respect to weight w_1, we use chain rule to propagate the partial gradient of loss function with respect to outputs from the activation function and summation function until we arrive at w_1. Such expression can be denoted as:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial p_3} \cdot \frac{\partial p_3}{\partial p_1} \cdot \frac{\partial p_1}{\partial w_1} \tag{7}$$

Back Propagation



In order to obtain the partial derivative of cost function L with respect to weight w_1, we use chain rule to propagate the partial gradient of loss function with respect to outputs from the activation function and summation function until we arrive at w_1. Such expression can be denoted as:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial p_3} \cdot \frac{\partial p_3}{\partial p_1} \cdot \frac{\partial p_1}{\partial w_1} \tag{7}$$

If we use least squares for loss function and sigmoid function for activation function, then the terms in the chain rule expression can be derived to be:

$$\frac{\partial L}{\partial y} = (y - T) \quad \frac{\partial y}{\partial p_3} = \hat{g}(p_3) = g(p_3)(1 - g(p_3)) \tag{8}$$

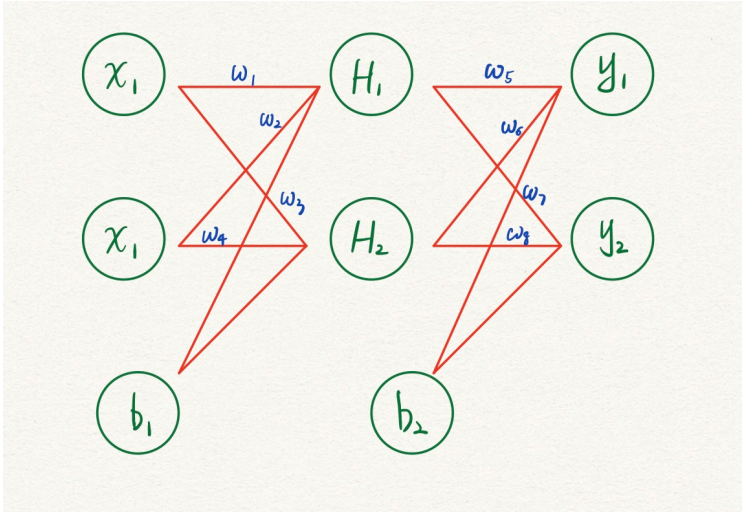
$$\frac{\partial p_3}{\partial p_1} = 1 \qquad \frac{\partial p_1}{\partial w_1} = x_1 \tag{9}$$

With the above building blocks, the partial derivatives of loss function with respect to the two weights w_1 and w_2 are:

$$\frac{\partial L}{\partial w_1} = (y - T)g(p_3)(1 - g(p_3)x_1 \quad \frac{\partial L}{\partial w_2} = (y - T)g(p_3)(1 - g(p_3)x_2 \tag{10}$$

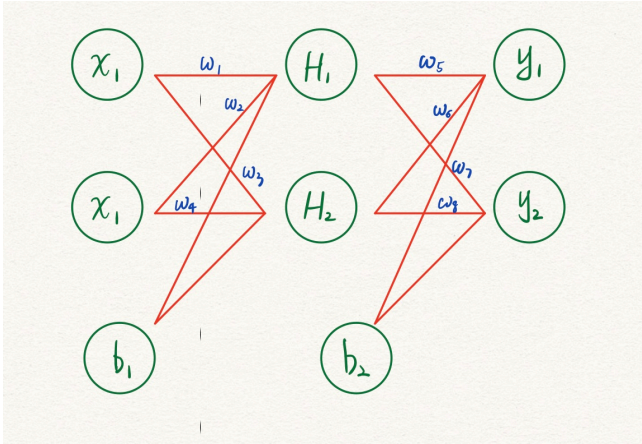
Back Propagation (Example)

Consider a neural network that has a one input layer, one hidden layer and one output layer with 2 neurons in each layer, and has 2 bias neurons connected to hidden layer and output layer. The schematic is shown in fig.7. The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



Back Propagation (Example)

The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



$$H_1 = w_1 x_1 + w_2 x_2 + b_1 = 0.3775$$

$$H_{1out} = \frac{1}{1 + \exp -H_1} = 0.59326$$

H_{2out} can be calculated the same way and $H_{2out} = 0.59688$. Now calculate y_1 as follows:

$$y_1 = H_{1out} \times w_5 + H_{2out} \times w_6 + b_2 = 1.059$$

$$y_{1out} = \frac{1}{1 + \exp -y_1} = 0.75136 \qquad y_{2out} = 0.77293$$

$$E_{total} = \sum \frac{1}{2}(T - y_{out})^2$$

To update the weights, we should back propagate the errors to the weights. The update rule is equation(6).

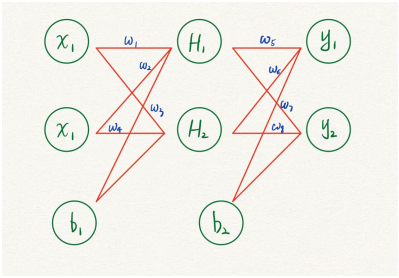
For w_5:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial y_{1out}} \cdot \frac{\partial y_{1out}}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_5}$$

Carnegie Mellon University

Back Propagation (Example)

The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



$$\frac{\partial E_{total}}{\partial w_5} = 2 \times \frac{1}{2} \times (T_1 - y_{1out}) \times (-1) \times y_{1out}(1 - y_{1out}) \times H_{1out} = 0.082187 \quad (17)$$

Then the weight w_5 can be updated as:

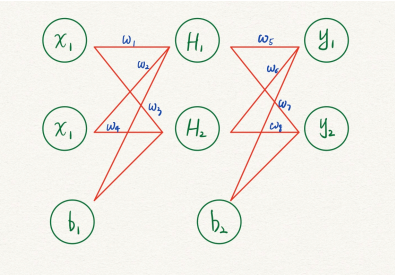
$$w_{5new} := w_{5old} - \alpha \frac{\partial E_{total}}{\partial w_5} = 0.35891 \quad (18)$$

w_6, w_7, w_8 can be updated the same way and equal to 0.40866, 0.511301, 0.06137 respectively.

As for weights between the input layer and the hidden layer, take w_1 for instance: There is two path from output layer to node H_1, we need to take both the effects into consideration. That is to say, when calculate the derivative

Back Propagation (Example)

The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



of loss function of w_1, it has two terms:
from y_1:

$$\frac{\partial E_{total}}{\partial w_1}_{w_5} = \frac{\partial E_{total}}{\partial y_{1out}} \cdot \frac{\partial y_{1out}}{\partial y_1} \cdot \frac{\partial y_1}{\partial H_{1out}} \cdot \frac{\partial H_{1out}}{\partial H_1} \cdot \frac{\partial H_1}{\partial w_1} \tag{19}$$

from y_2:

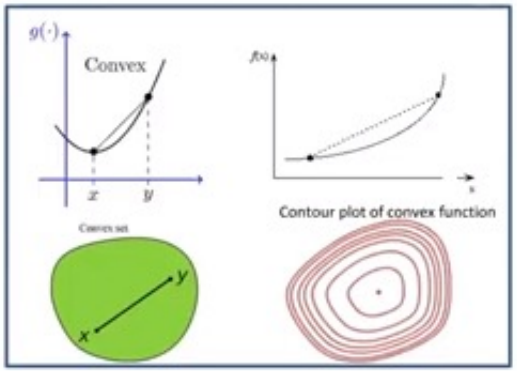
$$\frac{\partial E_{total}}{\partial w_1}_{w_7} = \frac{\partial E_{total}}{\partial y_{2out}} \cdot \frac{\partial y_{2out}}{\partial y_2} \cdot \frac{\partial y_2}{\partial H_{1out}} \cdot \frac{\partial H_{1out}}{\partial H_1} \cdot \frac{\partial H_1}{\partial w_1} \tag{20}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial w_1}_{w_5} + \frac{\partial E_{total}}{\partial w_1}_{w_7} = 0.000438 \tag{21}$$

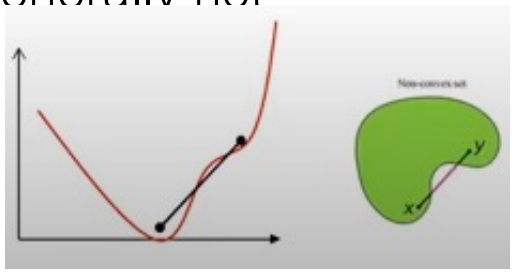
And then w_1 is updated substitute eq(21) to eq(6):$w_1 = 0.14978$. Through similar calculation,$w_2 = 0.19956$, $w_3 = 0.29975$, $w_4 = 0.29950$. Through the example above, we now have basic concept of how forward and back propagation works in a neural network.

Convex Loss Function

- A surface is “convex” if it is continuously curving upward
  -- We can connect any two points on or above the surface without intersecting it
  -- Many mathematical definitions that are equivalent



- **Caveat**: Neural network loss surface is generally not convex
  -- Streetlight effect

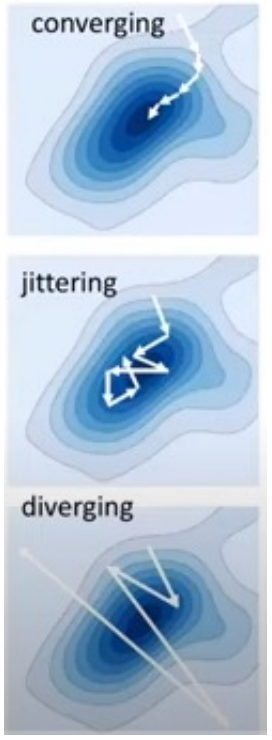


Convergence of gradient descent

- An iterative algorithm is said to converge to a solution if the value updates arrives at a fixed point
  -- Where the gradient is 0 and further updates do not change the estimate

- The algorithm may not actually converge
  -- It may jitter around the local minimum
  -- It may even diverge

- Conditions for convergence?



Batch, Stochastic, Minibatch

Batch gradient descent
Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters w for the entire training dataset.

Stochastic gradient descent
Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example x(i) and label y(i)

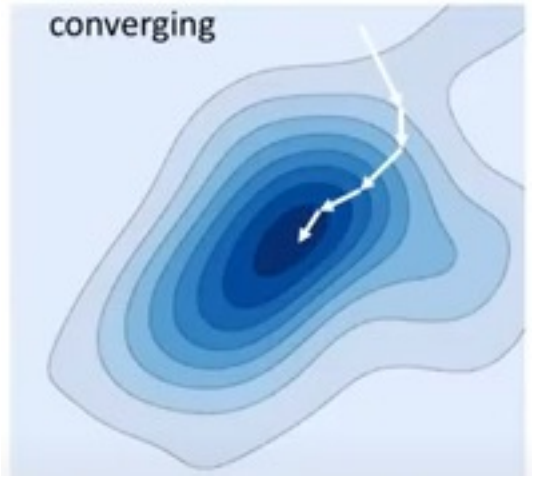
Mini-batch gradient descent
Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples.

Convergence and convergence rate

- Convergence rate: How fast the iterations arrive at the solution
- Generally quantified as

$$R = \frac{\left|f\left(x^{(k+1)}\right) - f(x^*)\right|}{\left|f(x^{(k)}) - f(x^*)\right|}$$

-- x^{k+1} is the k-th iteration
-- x^* is the optimal value of x



- If R is a constant (or upper bounded), the convergence is linear
  -- In reality, its arriving at the solution exponentially fast

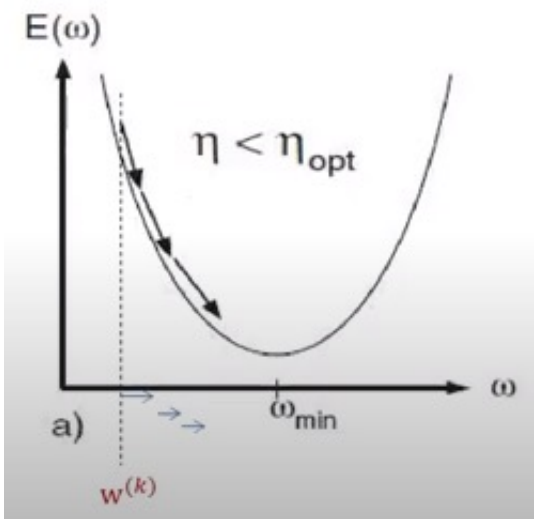
$$\left|f\left(x^{(k)}\right) - f(x^*)\right| \leq R^k \left|f\left(x^{(k)}\right) - f(x^*)\right|$$

Convergence for quadratic surfaces

$$Minimize\ E = \frac{1}{2}aw^2 + bw + c$$

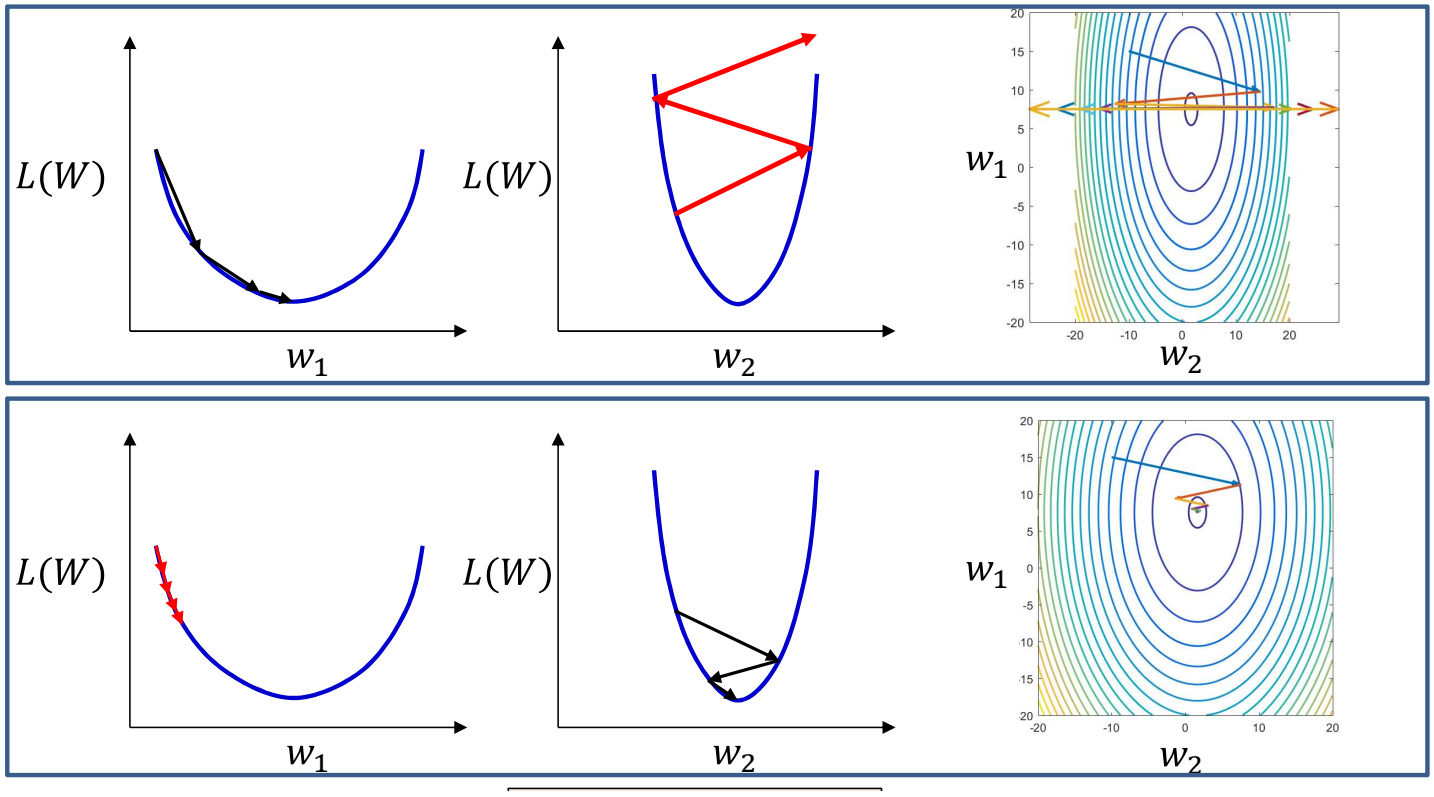
$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

Gradient descent with fixed step size η to estimate scalar parameter w



- Gradient descent to find the optimum of a quadratic, starting from $w^{(k)}$
- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

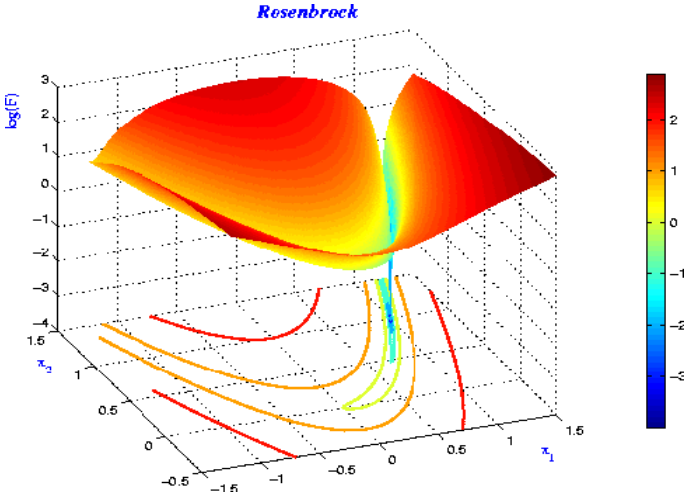
Eccentricity with other dimensions

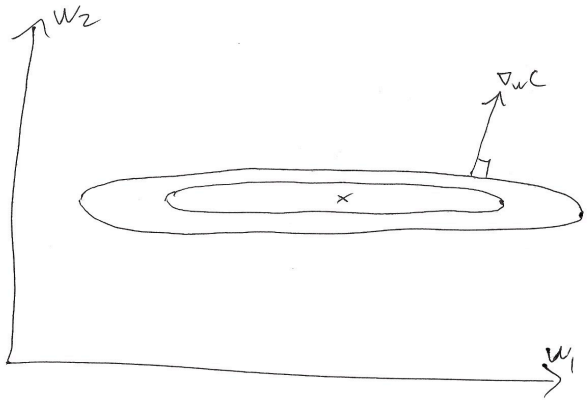


Carnegie Mellon University

Eccentricity with other dimensions

Long, narrow ravines:



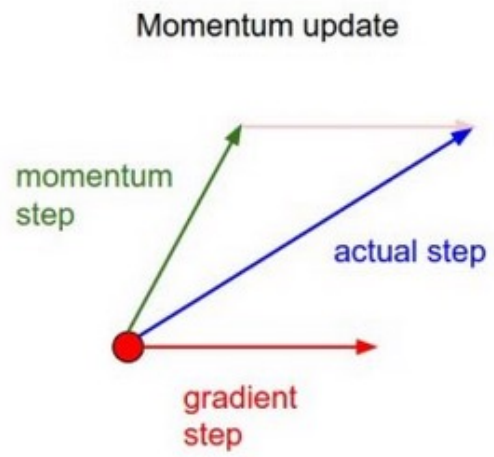


Lots of sloshing around the walls, only a small derivative along the slope of the ravine's floor.

Momentum Method

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another

This scenario is common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along



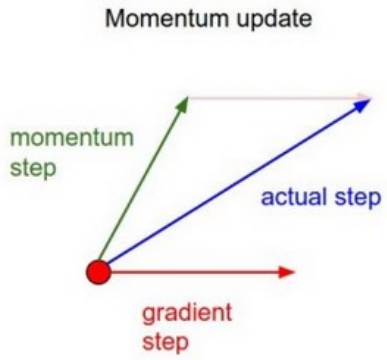
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

Momentum Method

In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.

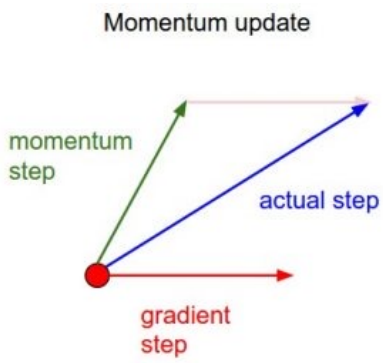
In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.
If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of

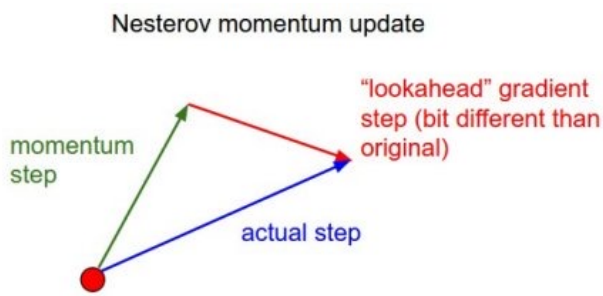


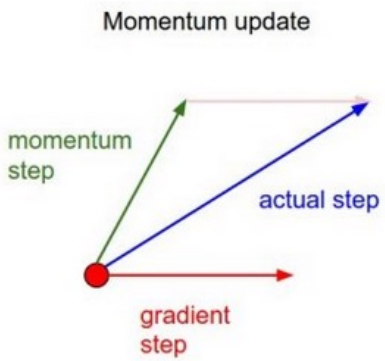
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

Nestrov Momentum







$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Second order methods