Optimization (Gradient Descent & Backprop)

Deep Learning: Bryan Pardo, Northwestern University, Fall 2020

Thanks to Max Morrison for a number of slides

Supervised Machine Learning in one slide

1. Pick data **X**, labels **Y**, model **M(**$\boldsymbol{\theta}$**)** and loss function $\boldsymbol{L}(\mathbf{X}, \mathbf{Y}; \boldsymbol{\theta})$

2. Initialize model parameters $\boldsymbol{\theta}$, somehow

3. Measure model performance with the loss function $\boldsymbol{L}(\boldsymbol{X}, \boldsymbol{Y}; \boldsymbol{\theta})$

HOW?

4. Modify parameters θ somehow, hoping to improve $\boldsymbol{L}(\boldsymbol{X}, \boldsymbol{Y}; \boldsymbol{\theta})$

5. Repeat 3 and 4 until you stop improving or run out of time

A common approach to picking the next parameters

HOW?

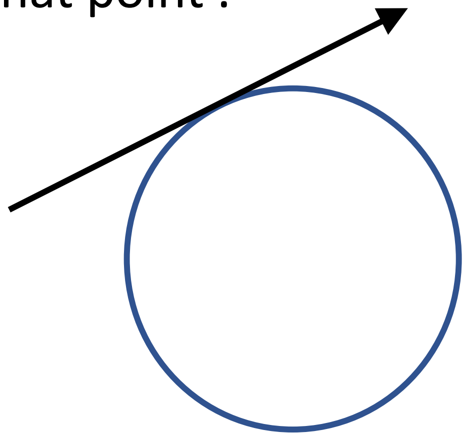
1. Measure how the the loss changes when we change the parameters θ slightly

2. Pick the next set of parameters to be close to the current set, but in the direction that most changes the loss function for the better

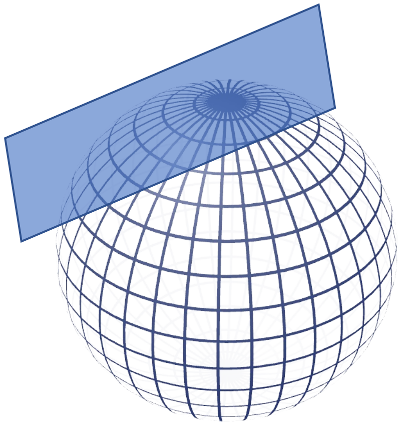
3. Repeat

Slope vs gradient

- Slope of $f(\theta)$ is a scalar describing a line perpendicular to the tangent of the function at that point .

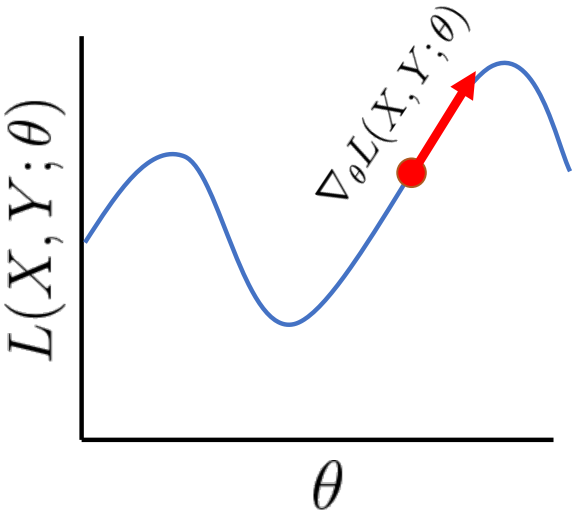


- Gradient $\nabla f(\boldsymbol{\theta})$ is a vector describing a hyperplane perpendicular to the tangent at $\boldsymbol{\theta}$



What does the gradient tell us?

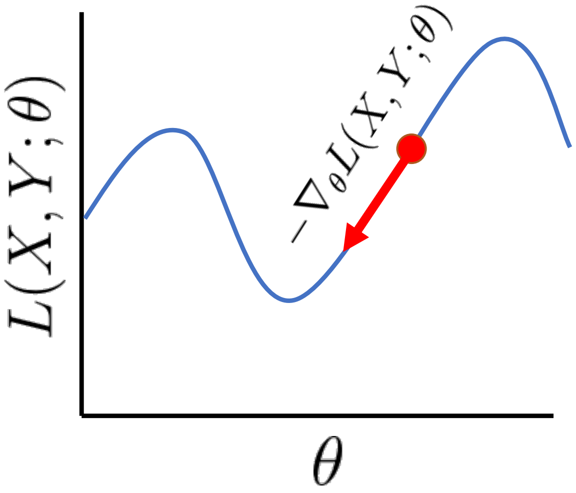
- If the loss function and hypothesis function encoded by the model are differentiable* (i.e., the gradient exists)
- We can evaluate the gradient for some fixed value of our model parameters θ and get the *direction* in which the loss *increases* fastest



*or subdifferentiable

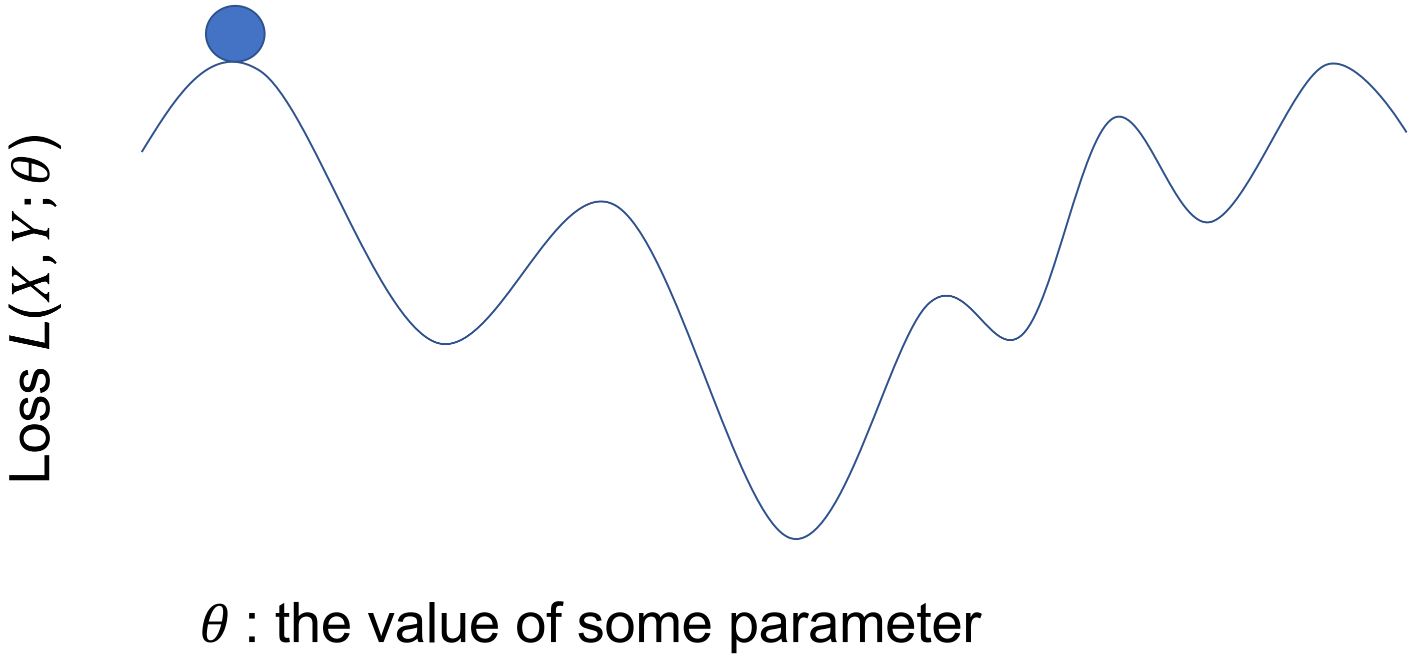
What does the gradient tell us?

- We want to *decrease* our loss, so let's go the other way instead



Gradient Descent: Promises & Caveats

- Much faster than guessing new parameters randomly
- Finds the global optimum only if the objective function is convex



Gradient Descent Pseudocode

```
Initialize $\theta^{(0)}$
Repeat until stopping condition met:
    $\theta^{(t+1)} = \theta^{(t)} - \eta\nabla L(X, Y; \theta^{(t)})$
Return $\theta^{(t_{max})}$
```

$\theta^{(t)}$ are the parameters of the model at time step t

X, Y are the input data vectors and the output values.

$\nabla L(X, Y; \theta^{(t)})$ is the gradient of the loss function with respect to model parameters $\theta^{(t)}$

η controls the step size

$\theta^{(t_{max})}$ is the set of parameters that did best on the loss function.

Design choices

```
Initialize $\theta^{(0)}$
Repeat until stopping condition met:
    $\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(X, Y, \theta^{(t)})$
Return $\theta^{(t_{max})}$
```

- Initialization of θ
- Convergence criterion (i.e. when to stop)
- How much data to use (batch size)
- Step size for updating model parameters
- Choosing a loss function

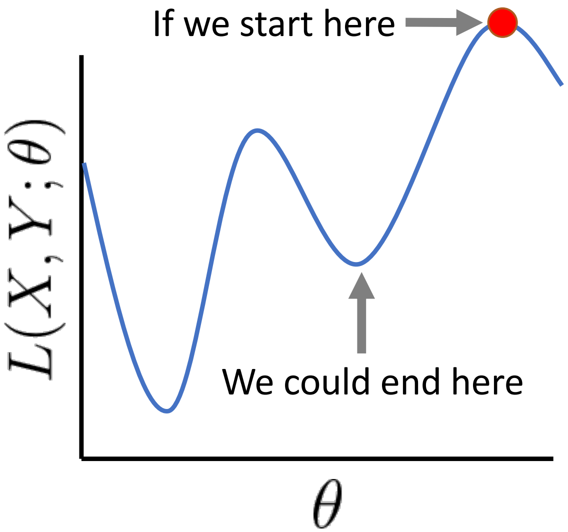
Parameter Initialization

Common initializations:

- $\theta^{(0)} = 0$
- $\theta^{(0)} =$ random values

What happens if our initialization is bad?

- Convergence to a *local* minimum
- No way to determine if you've converged to the global minimum



Convergence criterion: when to stop

- Stop when the gradient is close (within ε) to 0
  (i.e., we reached a minimum)

- Stop after some fixed number of iterations

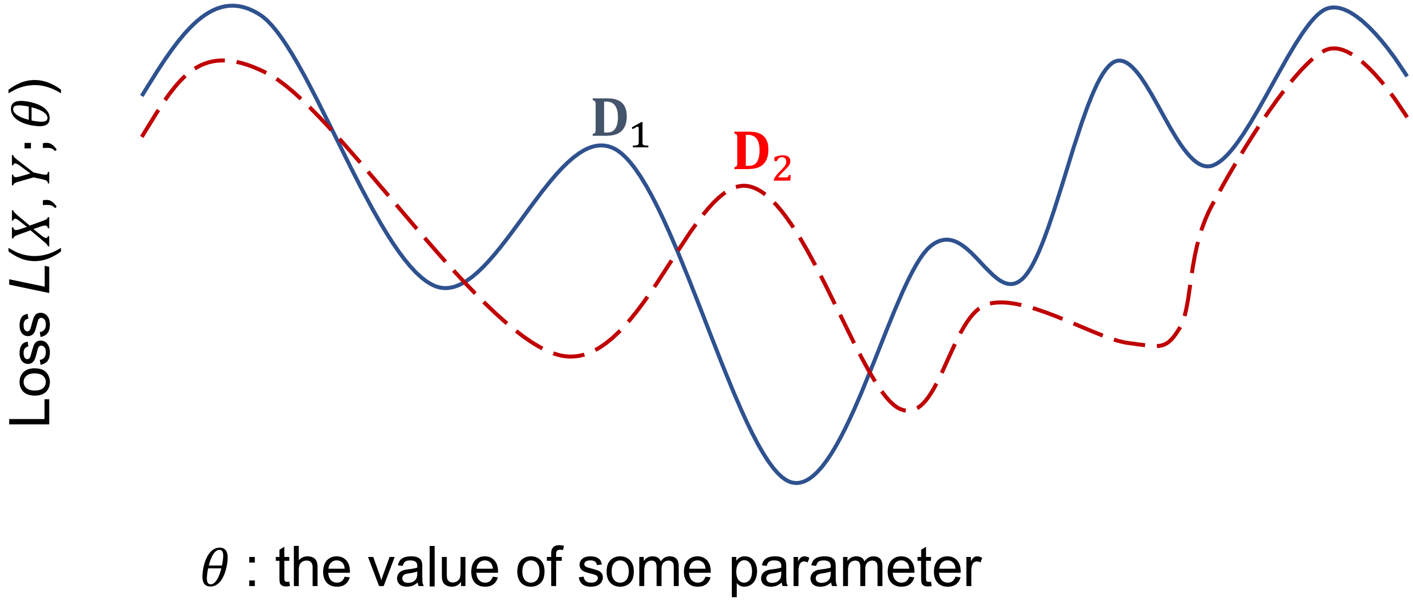
- Stop when the loss on a *validation set* stops decreasing
  (This helps prevent overfitting)

Batch Size: How much data?

- Call D the set of X,Y pairs we measure loss on
- In **batch gradient descent**, the loss is a function of both the parameters θ and the set of all training data **D.**
  (What if if |**D**| > memory?)
- In **stochastic gradient descent**, loss is a function of the parameters and a different single random training sample at each iteration.
- In **mini-batch gradient descent**, random subsets of the data (e.g. 100 examples) are used at each step in the iteration.

Different data, different loss

- Call D the set of X,Y pairs we measure loss on.
- If D changes, then the landscape of the loss function changes
- You typically won't know how it has changed.

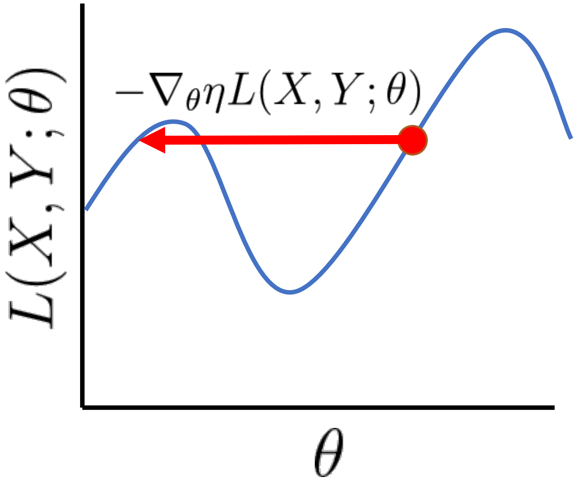


How much data to use in each step?

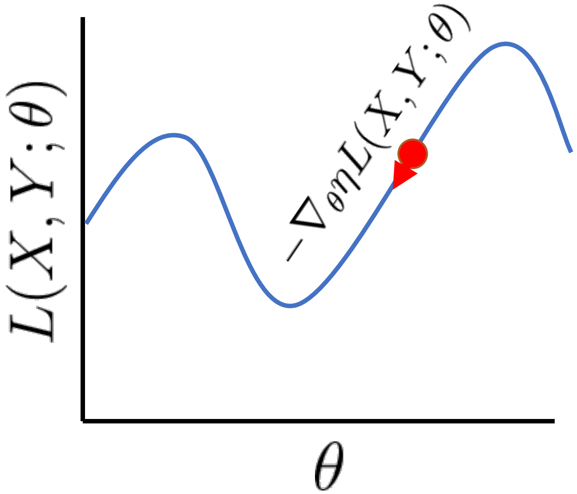
- **All of it (*batch gradient descent*)**
  - The *most accurate* representation of your training loss
  - It can be slow
  - Not possible if data does not fit in RAM
- Just one data point (*stochastic gradient descent*)
  - A *noisy, inaccurate* representation of your training loss
  - *very* fast
  - Random shuffling is important
- More than one data point, but less than all (*mini-batch gradient descent)*
  - Most common approach today
  - Balances *speed* and *accuracy*
  - Random shuffling is important
  - Usually want batch size to be as large as possible for your machine

Step Size: how far should we go?

- The gradient we calculated was based on a fixed value of θ
- As we move away from this point, the gradient changes



If the step size is too large, we may overshoot the minimum



If the step size is too small, we need to take more steps (more computation)

Add Momentum

```
Initialize $\theta^{(0)}, V^{(0)}$
Repeat until stopping condition met:
        $V^{(t+1)} = mV^{(t)} - \eta\nabla L(X, Y, \theta^{(t)})$
        $\theta^{(t+1)} = \theta^{(t)} + V^{(t+1)}$
Return $\theta^{(t_{max})}$
```

There are many variants on gradient descent

- Lots of kinds of momentum/step size selection algorithms (e.g. ADAM)

- Lots of 2nd order algorithms (e.g. BGFS)

- This is an entire field of study.

- Check out classes taught in IEMS on this.

Loss functions

A good objective (loss) function $L(\mathrm{X}, \mathrm{Y}; \theta)$

data labels parameters

Required

$L(\mathrm{X}, \mathrm{Y}; \theta) \geq 0$

$L(\mathrm{X}, \mathrm{Y}; \theta)$ decreases as performance improves

Required for gradient descent

$L(\mathrm{X}, \mathrm{Y}; \theta)$ is differentiable*, with respect to θ

helpful For gradient descent

The gradient of L is bounded … $\mathbf{0} < |\nabla L| \ll \infty$

*or subdifferentiable

Notational conventions

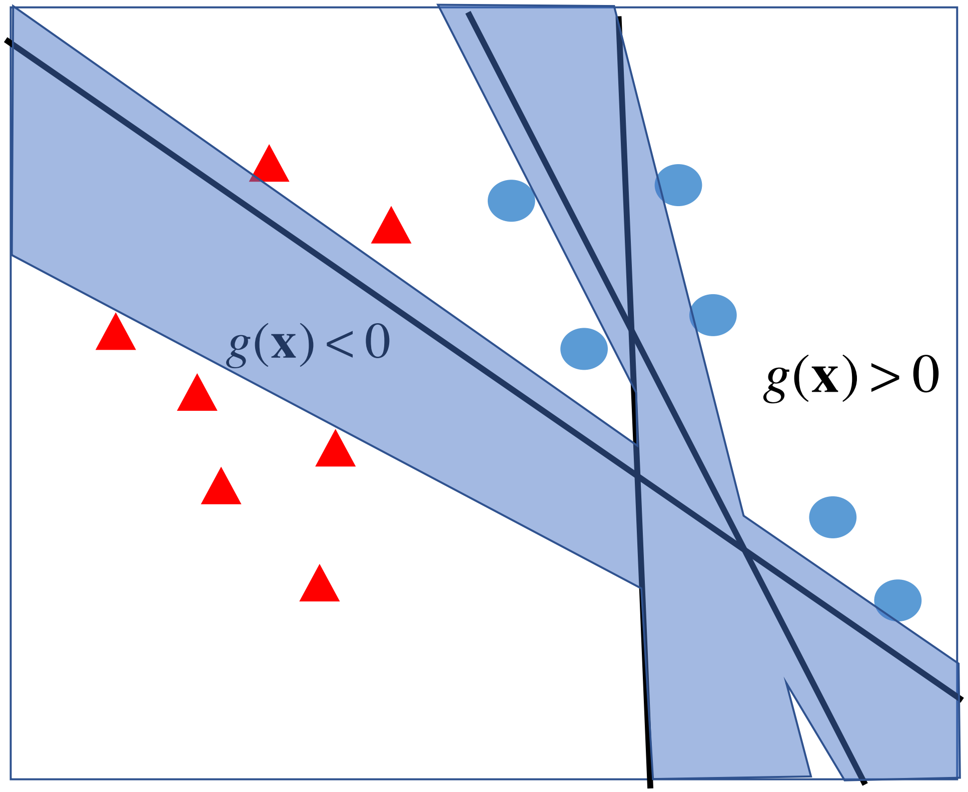
D is the total number of dimensions
d is the current dimension

$\mathbf{w}$ is the *D* dimensional model weight vector (i.e. the model parameters θ)
w_d is the model weight for dimension *d*

$\mathbf{x}$ is one *D* dimensional input example
x_d is the value for $\mathbf{x}$ at dimension *d*
X is a set of examples
$\mathbf{x}_{\boldsymbol{i}}$ is the *i*th example in *X* (note the boldface and use of *i* instead of *d*).

y is one scalar label, drawn from {+1, -1}
Y is a set of labels
y_i is the *i*th example in *Y.*

Example: 0 1 loss



Our linear model

$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$

Our hypothesis function

$$h(\mathbf{x}) = \begin{cases} 1 & if\ 0 < \mathbf{w}^T\mathbf{x} \\ -1 & else \end{cases}$$

Our label estimate

$$\hat{y} = h(\mathbf{x})$$

Sum of squared errors loss

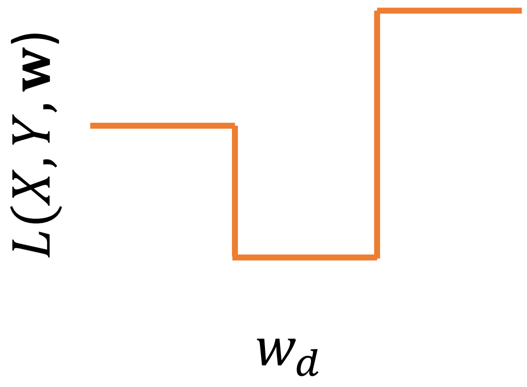
$$L(X, Y, \mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

SSE is same everywhere in the blue

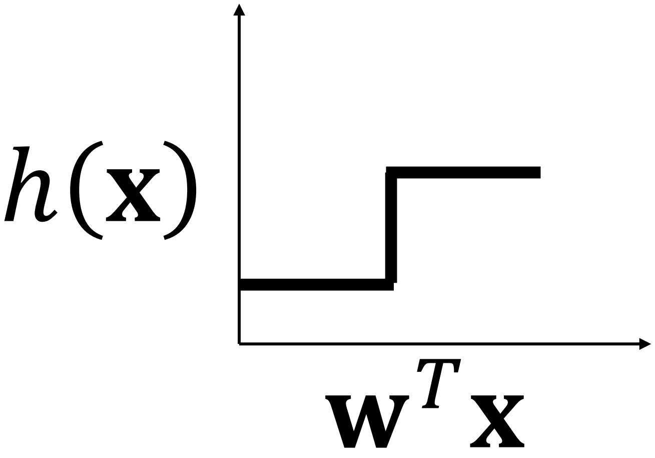
Gradient 0 in the blue region!

The 0 1 Loss function

- $\text{Loss} = 1 \text{ if } y \neq h(x)$, else it’s 0
- A count of mislabeled items
- Results in a step function
- Not useful for for gradient descent

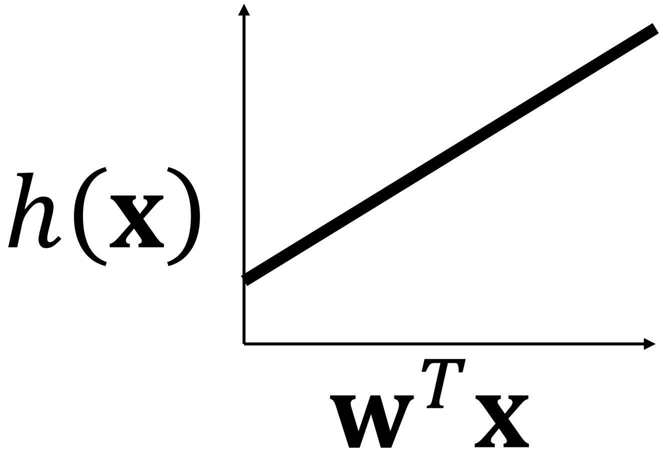


Perceptron Problem: The step function



$$h(x) = \begin{cases} \quad 1 \quad if\ 0 < \mathbf{w}^T\mathbf{x} \\ -1 \quad\ \ else \end{cases}$$

Solution: Remove the step function



$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

Squared loss: we now have a gradient

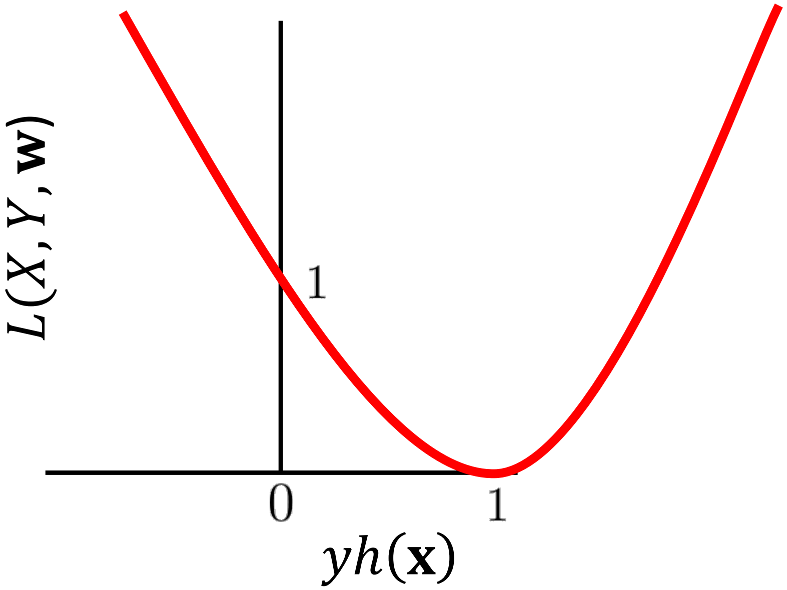
- Our hypothesis function is now $h(\mathbf{x})$ where $\mathbf{w}$ are the model parameters.

- We write our loss function as..

$$L(X, Y, \mathbf{w}) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^{\mathbf{2}}$$

- If we use a linear model, then..

$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$$



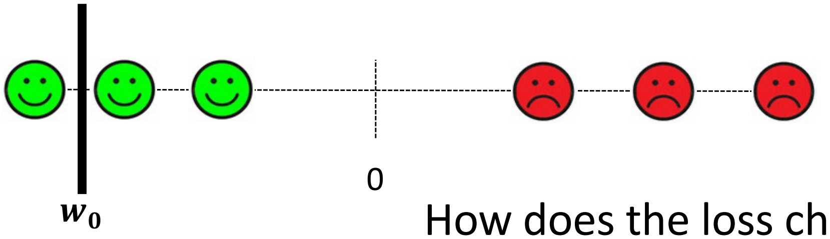
A simple example: where do you draw the line?

Happy faces have label *y* = +1 and sad faces have label y = -1.

We have a linear model with 2 parameters: $\hat{y} = \mathbf{w}^T\mathbf{x} = w_0 x_0 + w_1 x_1$

Our loss function will be sum-of-squared-errors:

$$L(X, Y, \mathbf{w}) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^{\mathbf{2}}$$



How does the loss change as we move the line defined by $\boldsymbol{w_0}$?
Can we use that to decide where to move it?
What does $\boldsymbol{w_1}$ do?

Measuring loss for a linear unit

- Model's hypothesis $h(\mathbf{x})$ function outputs a label estimate $\hat{y}$, given its parameters θ. Let's call them the weights, $\mathbf{w}$.

$$\hat{y} = h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- Sum of squared errors loss function:

i is the index to the ith example $\mathbf{x}_i$ and its label $\mathbf{y}_i$

$$L(X, Y, \mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^{\mathbf{2}}$$

This 2 makes the derivative simpler

If we consider a single example, then…

$$L(X, Y, \mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_{\boldsymbol{i}})^{\mathbf{2}}$$

Setting the number of data points N = 1 results in…

$$L(\mathbf{x}, y, \mathbf{w}) = \frac{1}{2}(y - \hat{y})^{\mathbf{2}}$$

The example **x** is a *D* dimensional vector
The model weights **w** are also *D* dimensional
Our label *y* is a scalar

For each dimension *d*, take the partial derivative

$\frac{\partial L}{\partial w_d} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_d}$ gives the change of our loss function L with respect to weight w_d

Our loss function is : $L = \frac{1}{2}(y - \hat{y})^{\mathbf{2}}$

$$= \frac{y^2}{2} + \frac{\hat{y}^2}{2} - y\hat{y}$$

therefore… $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$

For each dimension *d*, take the partial derivative

$\frac{\partial L}{\partial w_d} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_d}$ gives the change of our loss function L with respect to weight w_d

From the previous slide.... $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$

Our estimator is a linear unit : $\hat{y} = \mathbf{w}^T \mathbf{x}$

therefore… $\frac{\partial L}{\partial \hat{y}} = \mathbf{w}^T \mathbf{x} - y$

Let’s calculate $\frac{\partial \hat{y}}{\partial w_d}$

D is the total number of dimensions
d is the current dimension
$\mathbf{w}$ is the *D* dimensional model weight vector
$\mathbf{x}$ is the *D* dimensional input example
w_d is the model weight for dimension *d*
x_d is the value for $\mathbf{x}$ at dimension *d*

Our estimator is : $\hat{y} = \mathbf{w}^T\mathbf{x} = w_0x_0$+… w_dx_d +… w_Dx_D

Now… w_d is the only parameter we’re varying right now.

So all w_j where $j \neq d$ are constant in this partial derivative.

Therefore, $\frac{\partial \hat{y}}{\partial w_d}$= x_d

The gradient for weight *d* **is…**

$$\frac{\partial L}{\partial w_d} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_d} = (\mathbf{w}^T\mathbf{x} - y)x_d$$
$$= -(y - \mathbf{w}^T\mathbf{x})x_d$$

So the gradient of the loss for all *D* weights is…

$$\nabla L(\mathbf{x}, y, \mathbf{w}) = \left[\frac{\partial L}{\partial w_0}, \ldots \frac{\partial L}{\partial w_d}, \ldots \frac{\partial L}{\partial w_D}\right]$$

$$= -(y - \mathbf{w}^T\mathbf{x})\mathbf{x}$$

We can now estimate the gradient for a whole set

$$\nabla L(X, Y, \mathbf{w}) \; = \frac{1}{N} \sum_{i=1}^{N} \nabla L(\mathbf{x}_i, y_i, \mathbf{w})$$

X and *Y* are the set of examples and labels.
N is the number of examples.
$\mathbf{x}_i, \mathbf{y}_i$ are a single pair of example and label.

The gradient can now be used here

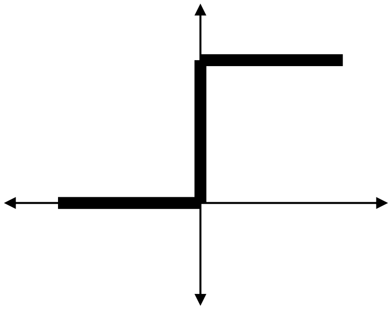
```
Initialize $\theta^{(0)}$
Repeat until stopping condition met:
    $\theta^{(t+1)} = \theta_t - \eta \nabla L(X, Y; \theta^{(t)})$
Return $\theta^{(t_{max})}$
```

$\theta^{(t)}$ are the parameters of the model at time step *t.*

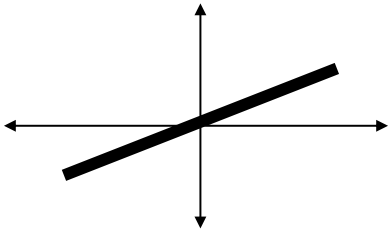
$(NOTE$: $\theta^{(t)}$ corresponds to the model weights **w** from the prev. slide)

Sigmoid (aka Logistic) function: best of both

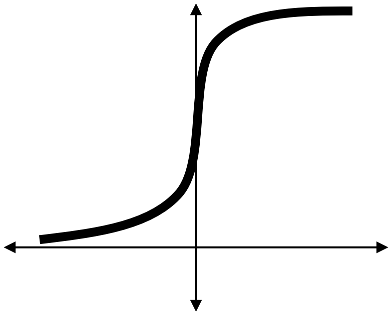
- Perceptron $f(x) = \begin{cases} 1\ if\ 0 < \sum_{i=0}^{n} w_i x_i \\ -1\ else \end{cases}$



- Linear $f(x) = \mathbf{w}^T\mathbf{x} = \sum_{i=0}^{n} w_i x_i$



- Sigmoid $f(x) = \sigma(x) = \dfrac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$



What’s cool about the sigmoid function

- It looks like a rounded step function, so we can build circuits of arbitrary functions like we can with perceptrons

- It has non-zero slope everywhere and no sharp corners

- The derivative of the function is this: $\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$

- …and it’s easy to plug into the gradient descent algorithm to get the learning rule.

For each dimension *i*, take the partial derivative

Our loss function: $L = \frac{1}{2}(y - \hat{y})^{\mathbf{2}}$ Our estimate: $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$, where $z = \mathbf{w}^T\mathbf{x}$

$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_i}$ gives the change of loss function L with respect to weight w_i

Therefore $\frac{\partial L}{\partial \hat{y}} = (y - \hat{y}) = \big(y - \sigma(z)\big)$

..and $\frac{\partial \hat{y}}{\partial z} = \sigma(z)(1 - \sigma(z))$, as was given to us.

...and $\frac{\partial z}{\partial w_i} = x_i$, since $z = \mathbf{w}^T\mathbf{x} = w_0 x_0 \ldots + w_i x_i \ldots + w_d x_d$

Therefore, $\frac{\partial L}{\partial w_i} = \big(y - \sigma(z)\big)\sigma(z)(1 - \sigma(z))x_i$

For each dimension *i*, take the partial derivative

From the previous slide: $\frac{\partial L}{\partial w_i} = \big(y - \sigma(z)\big)\sigma(z)(1 - \sigma(z))x_i$

$\mathrm{Let's\ compose}\ \sigma(z) = \frac{1}{1+e^{-z}}$ and $z = \mathbf{w}^T\mathbf{x}$ into one function (called $\sigma(\mathbf{x})$), to get the following:

$$\sigma(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

This lets us now write the change in loss as:

$$\frac{\partial L}{\partial w_i} = \big(y - \sigma(\mathbf{x})\big)\sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))x_i$$

Backpropagation of error

Where we left off

- We have the $\sigma(x)$ sigmoid function that we can train with gradient descent, because it's differentiable and has a non-zero gradient everywhere.
- We can plug multiple sigmoids together to form arbitrary Boolean functions, by just interpreting the last output with sign($\sigma(x)$)
- We now need a way to have error from the output sigmoid function to flow to the input, so we can adjust the parameters of every $\sigma(x)$ on the path from the input to the output when we do our gradient descent.

Consider one output node

Let’s define a function…

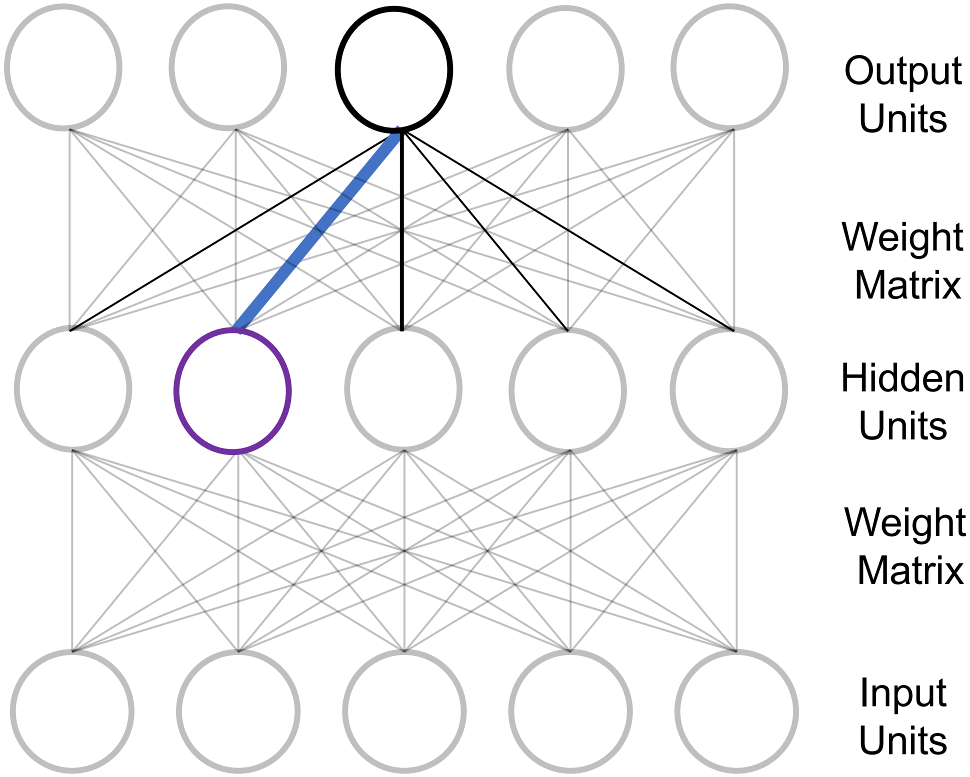
$$\delta = \big(y - \sigma(\mathbf{x})\big)\, \sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))$$

Now this…

$$\frac{\partial L}{\partial w_i} = \big(y - \sigma(\mathbf{x})\big)\sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))x_i$$

…becomes this: $\dfrac{\partial L}{\partial w_i} = \delta x_i$

For any output node k we just use this, as before.



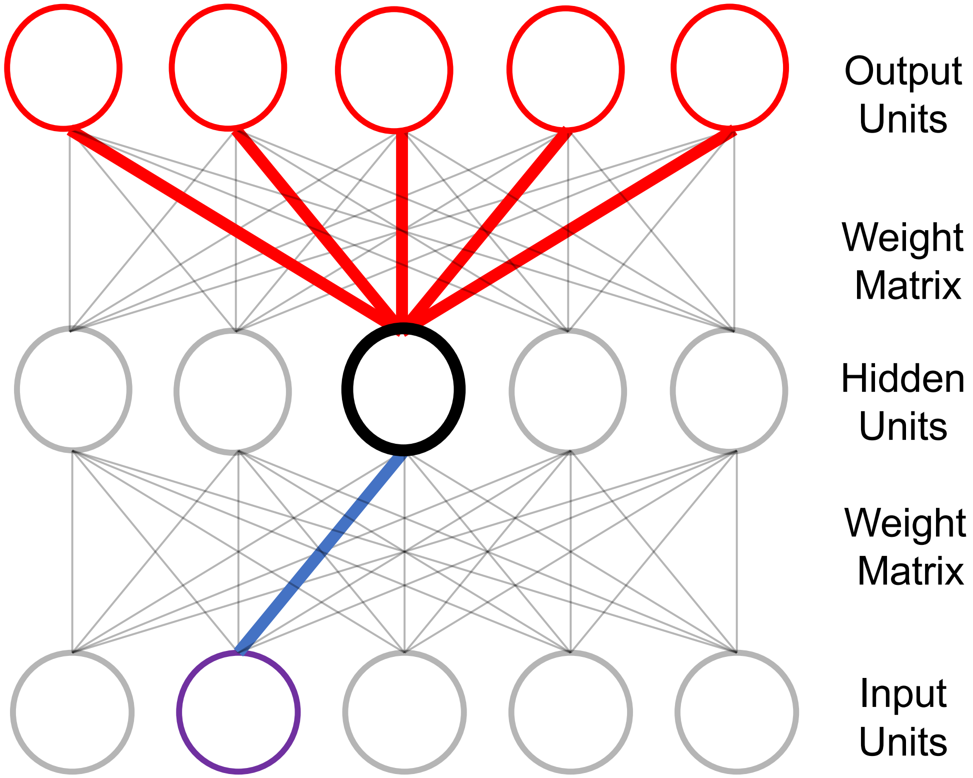
Consider one hidden node

For a hidden node h we need to redefine δ. Instead of comparing the output of the node to a known target output y, we look at its contribution to the output of the k nodes it is connected to at the next layer.

$$\delta = \left(\sum_k w_k\,\delta_k\right)\sigma(\mathbf{x})(1-\sigma(\mathbf{x}))$$

…and we then do: $\dfrac{\partial L}{\partial w_i} = \delta x_i$

We can do this repeatedly for multiple hidden layers.



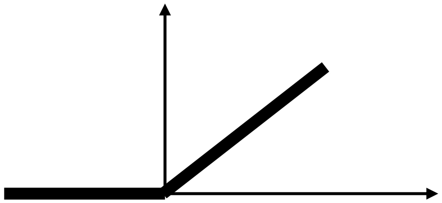
Some stuff I should mention

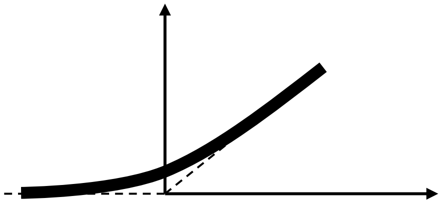
Sigmoid + SSE are not your only choices

- Pick an activation function
- Pick a loss function
- Make sure they’re both differentiable (or sub-differentiable)
- You can now do backpropagation of error

Rectified Linear Unit (ReLU) & Soft Plus :

- ReLU $f(x) = \max(0, \mathbf{w}^T\mathbf{x})$



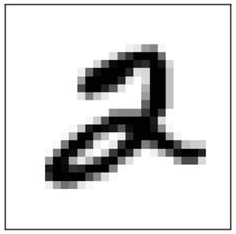


- Soft Plus $f(x) = \ln(1 + e^{\mathbf{w}^T\mathbf{x}})$

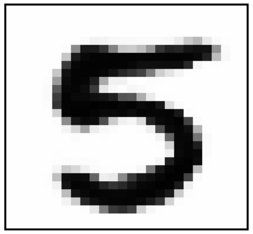
- Both can be combined in layers to make non-linear functions

“One Hot” Encoding

- A vector of values where a single element is 1 and all the rest are 0
- Common way to encode the true label, y, in a multi-class labeling problem
- Can be interpreted as a probability distribution



y = 0 0 1 0 0 0 0 0 0 0



y = 0 0 0 0 0 1 0 0 0 0

Probability distribution

* Discrete random variable X represents some experiment.

* $P(X)$ is the probability distributions over $\{x_1,...,x_n\}$, the set of possible outcomes for X.

* These outcomes are mutually exclusive.

* Their probabilities sum to one : $\sum_{i=1}^{n} P(x_i) = 1$

Soft Max Function

- Turns an N-dimensional vector of real numbers into a probability distribution, even if the numbers are both pos
- For a deep net, a_i is the output of the ith node in the output layer

$$p_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$$

Why softmax?

Why do I need this? $p_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$

Wouldn’t taking the absolute value and averaging do just as well?

$$p_i = \frac{|a_i|}{\sum_{j=1}^{N} |a_j|}$$

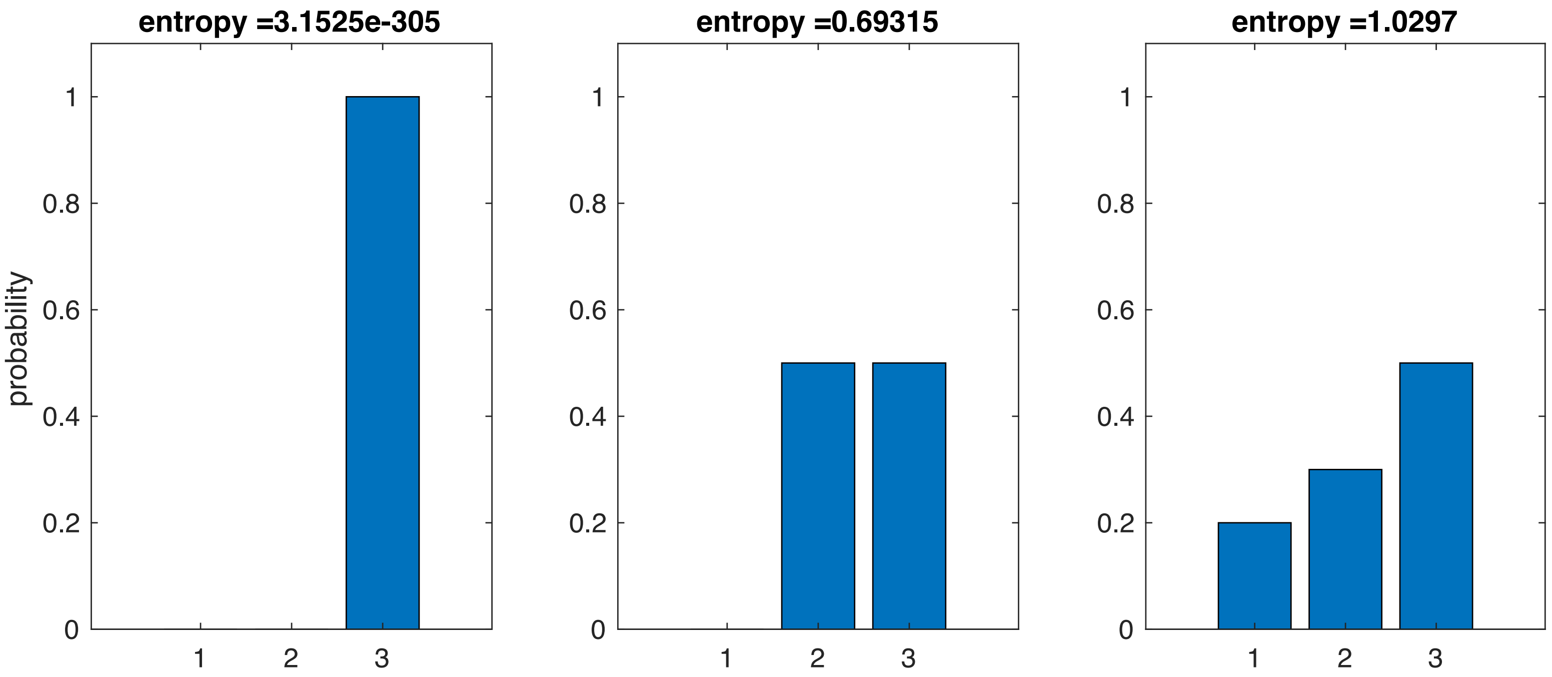
- Softmax is a multivariate extension of the sigmoid (logistic) function
- When combined with cross entropy loss function, the resulting derivative is a very nice one.

Entropy

- Entropy is the measure of the skewedness of a distribution
- The higher the entropy, the harder it is to guess the value a random variable will take when we draw from the distribution.
- Here,

$$H(P) = -\sum_{i=1}^{N} P(i)\log(P(i))$$

Some examples



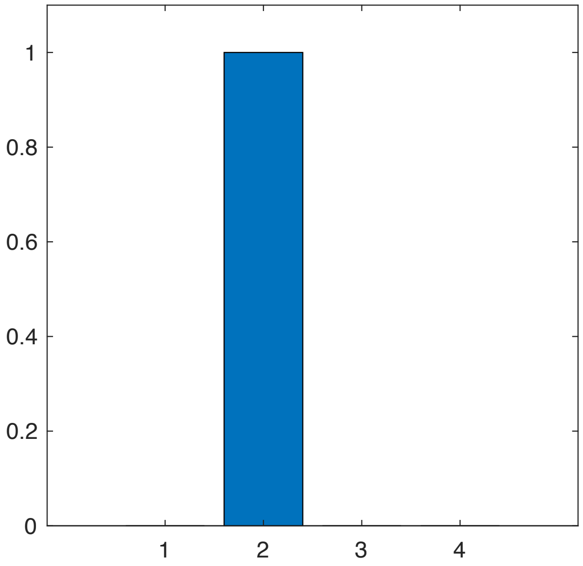
Cross Entropy

- Cross entropy is a measure of the similarity between distributions
- It is *NOT* symmetric.

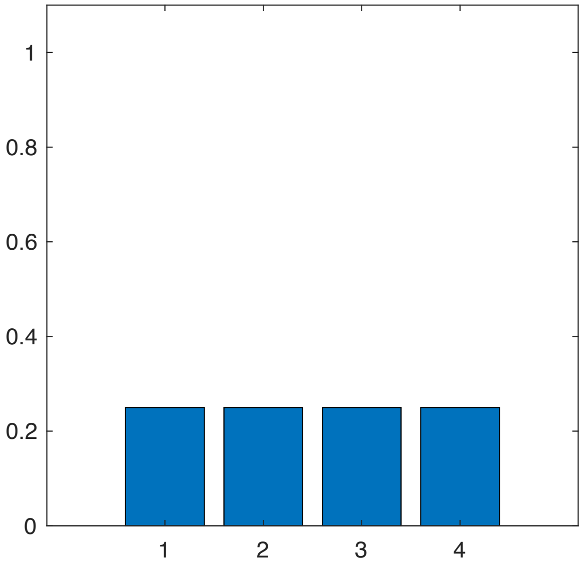
$$H(P,Q) = -\sum_{i=1}^{N} P(i)\log(Q(i))$$

An example

Distribution P

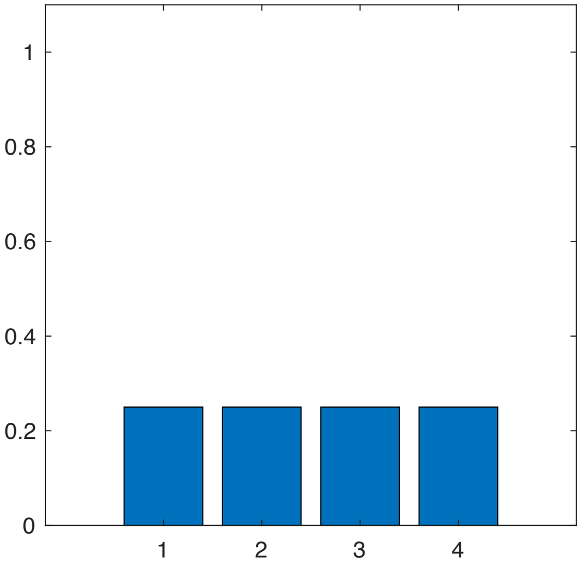


Distribution Q

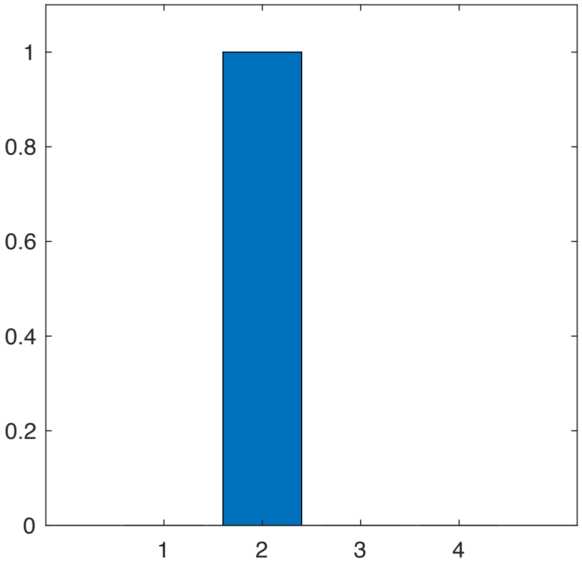


$$H(P,Q) = -\sum_{i=1}^{N} P(i)\log(Q(i)) = 1.39$$

An example



Distribution P



Distribution Q

$$H(P,Q) = -\sum_{i=1}^{N} P(i)\log(Q(i)) = \infty$$

Cross Entropy Loss Function

Given: “true” distribution $y = \{y_1,y_2, \dots y_N\}$ <-often a one-hot encoding

and estimated distribution $\hat{y} = \{\hat{y}_1,\hat{y}_2, \dots \hat{y}_N\}$ <-soft max over the last layer

Define cross entropy loss between 2 distributions as

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

A common approach…

- Define labels with a one-hot vector encoding
- Make the last layer have n nodes for an n-way classification problem
- Apply soft max to the last layer
- Use a cross-entropy loss function
- The resulting derivative of the loss function is wonderfully simple:

$$\frac{\partial L}{\partial a_i} = \hat{y}_i - y_i$$

L is the loss, i is the index to a node, a is the output of the last layer, $\hat{y}$ is the softmax probability distribution over the output layer of the network and y is the one-hot-encoding label.

There are many activation & loss functions

- As a system designer, you need to consider what activation function make sense for your problem

- The right loss function makes the difference between a learnable problem and an unlearnable one

- Different layers may have different activation functions

- Multiple loss functions may be used when teaching the network