for classification

Deep Learning Theory and Applications

Losses and activations

We see sigmoidal function has low gradients in some places. Are there other ways a NN can lose gradient

① activation → 0

② many layers

③ wrong initialization of weight

Yale

AMTH/CBB 663

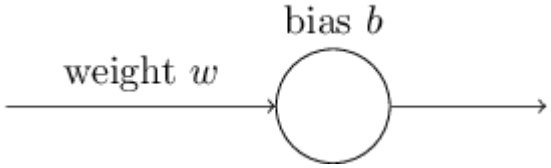
CPSC 452/552

Being wrong is unpleasant

- Suppose you give a piano concert before an audience and completely and obviously mess up the performance
- Humans get embarrassed and so you would likely learn from your mistake quickly
- In contrast, we learn more slowly when our errors are less-well defined
- What about neural networks?

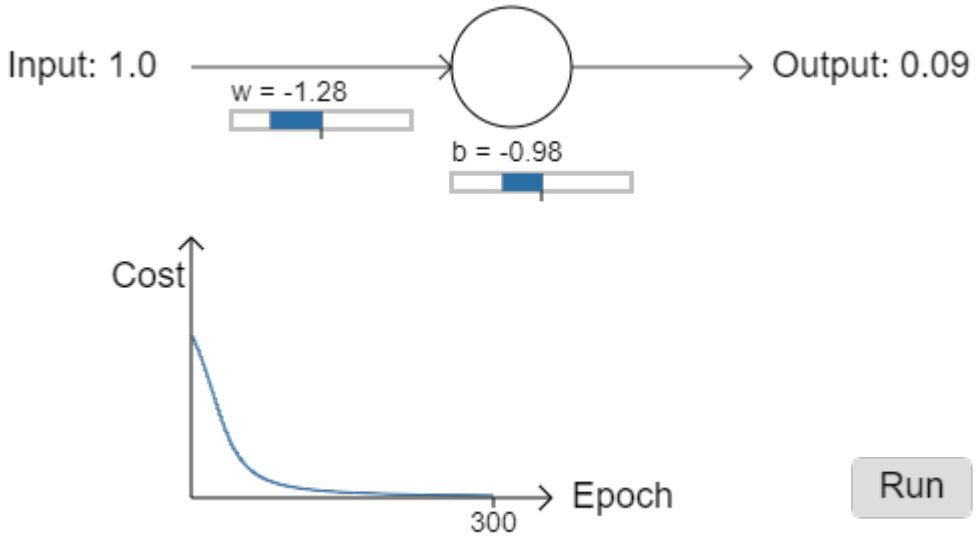
Do neural networks learn quickly?

- Example:



- Goal: train the neuron to map input 1 to output 0
  - Easy to do by hand, but let's see what gradient descent does
- Pick initial weight $w = 0.6$ and initial bias $b = 0.9$
  - Initial output is 0.82
  - A lot of learning is needed to get output to 0
  - Learning rate $\eta = 0.15$, quadratic cost function

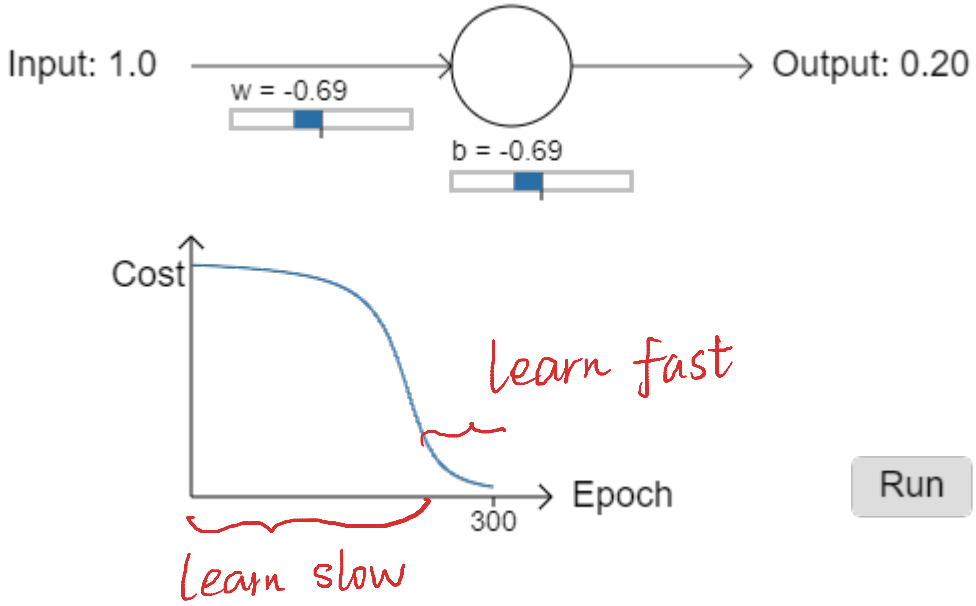
Do neural networks learn quickly?



- The neuron rapidly learns a weight and bias that decreases cost using 300 epochs
- Achieves output of 0.09 (pretty good)

Do neural networks learn quickly?

- Change starting weights and bias to be 2.0
  - Initial output is 0.98



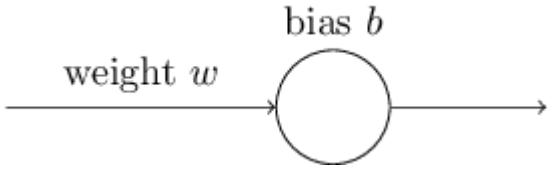
- Learning occurs much more slowly
  - Weight and bias change little for first 150 epochs

Neural networks aren't like humans

- Humans often learn fastest when we're very wrong
- The artificial neuron learns fastest when it's just a little wrong
  - True in more general networks as well

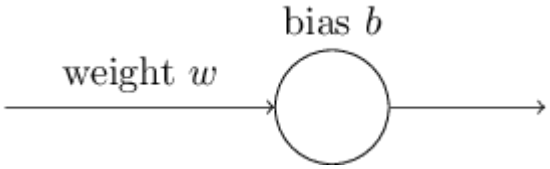
- Why is learning so slow?

Why is learning so slow?



- The neuron learns by changing the weight and bias based on the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$
- "Learning is slow" $= \frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ are small
- Why are the partial derivatives small?
- Let's compute them

Why is learning so slow?



- $C = \frac{(y-a)^2}{2}$ MSE
  - a = neuron output when $x = 1$
  - $y = 0$ is the desired output
  - Recall that $a = \sigma(z)$ and $z = wx + b$
- Using the chain rule gives

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z)$$

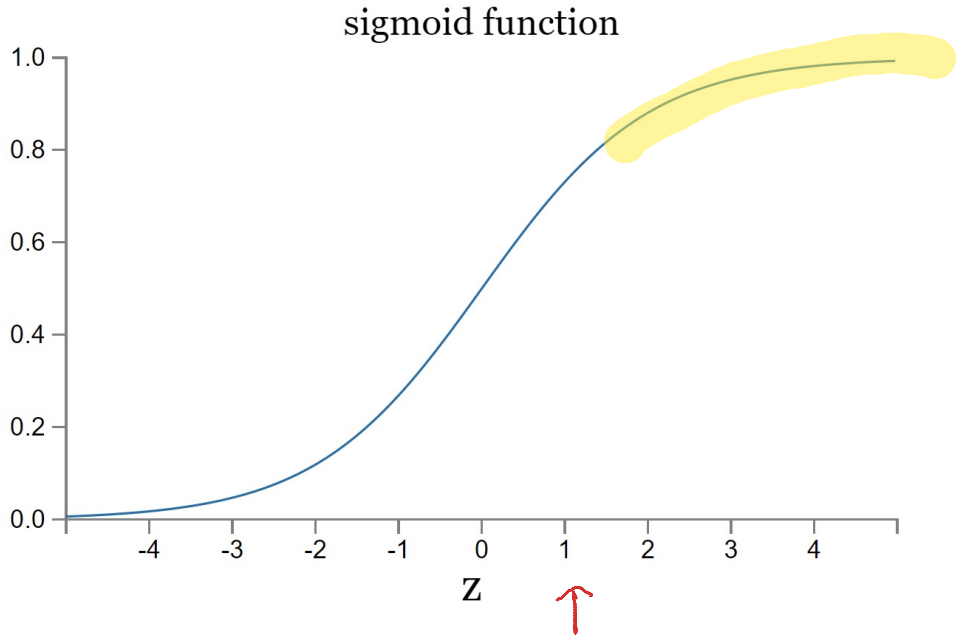
  - Recall that $x = 1$ and $y = 0$

derivative depends on activation function $\sigma(z)$

Why is learning so slow?

- Recall the sigmoid function

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



- Curve is flat when the output is close to 1
  - $\sigma'(z)$ gets very small
  - $\frac{\partial C}{\partial w} = a\sigma'(z) = \frac{\partial C}{\partial b}$ get very small
  - This occurs in more general neural networks as well

How can we fix this?

1. Change the cost function so that it doesn't depend on the activation function directly
2. Change the activation function

Lets try both!

derivative of cost function doesn't depend on derivative of activation

eg. Linear activation

$$\frac{\partial C}{\partial w} = a \cdot 1 = a$$

Changing the cost function

- A cost function needs to compare the neural network output with the actual output
  could be pr
- If the neural network output is a sigmoid then it gives a value between 0-1 that indicates the "probability of the output" (…in the case of one output)
  could be pr
- →We can use a method of comparing probability distributions
  - Cross entropy, KL Divergence , EMD (caveat: most formulations are undifferentiable)

max-min discrepency

Entropy from Information Theory

- Entropy is an information theoretic measure that measures the amount of information contained in a measurement
  - Information comes from the uncertainty about the outcome before hand

$H(p) = -2 \times \frac{1}{2} \log_2(\frac{1}{2}) = \log_2 2 = 1$

- A fair coin contains less information than a fair die
  - Only two possible outcomes in a fair coin
  - Six possible outcomes in a fair die
  - Fair die contains more information

0 or 1

1	000
2	001
⋮	⋮
6	110

- Formula for entropy: $H(P) = -\sum_i p_i \log_2(p_i)$
  - Probability distribution p
  - Over all possible outcomes I
  - Measured in units called "bits"

i is ith outcome

fair die $= -6 \times \frac{1}{6} \log_2(\frac{1}{6}) = \log_2 6$

Log base 2

def: How many bits need to transmit answer

Cross Entropy

- *Cross entropy* is the average number of bits needed to encode data coming from a source with distribution p when we use model q
- It is given by:
  - H(P,Q)= $-\sum_i p_i \log(q_i)$
- Consider the output of a single sigmoid a to be a binary probability, here it reduces to
  - H(P,Q)= $-\sum(a \log(y) + (1-a) \log(1-y))$
  - This is also called *binary cross entropy*

ground truth label

$y = \{0, 1\}$

Generalizing to many outputs

- Outputs from multiple sigmoid neurons do not form a probability distribution (why)
- Generalization of binary cross entropy to multi output functions is just by summing the binary cross entropy at each neuron

output is one-hot coding

$$C = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)]$$

Cross-entropy Cost function

characteristics of a cost function { have a minimum; continuous and differentiable

$$C = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)]$$

- Is this even a cost function?
  - $C > 0$
  - If the neuron's output is close to the desired output for all x, then $C \approx 0$

a y

- Example: $y = 0$ and $a \approx 0$

$$C = 0 - \ln(1-a) \approx 0$$

$= 0 - \ln(1-0)$

$= 0 - \ln 1 \quad = 0 - 0$

Seems to be a cost function!

Will it help vanishing gradients?

any activation function

- Set $a = \sigma(z)$ and apply chain rule twice:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\sigma'(z)x_j$$

The cross-entropy cost function

- Obtaining a common denominator gives:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)\big(1-\sigma(z)\big)}(\sigma(z)-y)$$

- Definition of sigmoid function $\sigma(z) = 1/(1+e^{-z})$

$$\sigma'(z) = \sigma(z)\big(1-\sigma(z)\big)$$

$$\Rightarrow \frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x x_j(\sigma(z)-y)$$

Cancelled out the derivative of the sigmoid!

The cross-entropy cost function

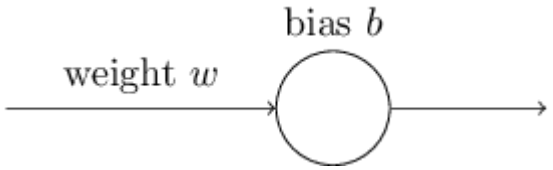
$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$$

Takeaway: the rate the weight is learned is controlled by $\sigma(z) - y$, i.e. the ***error in the output***

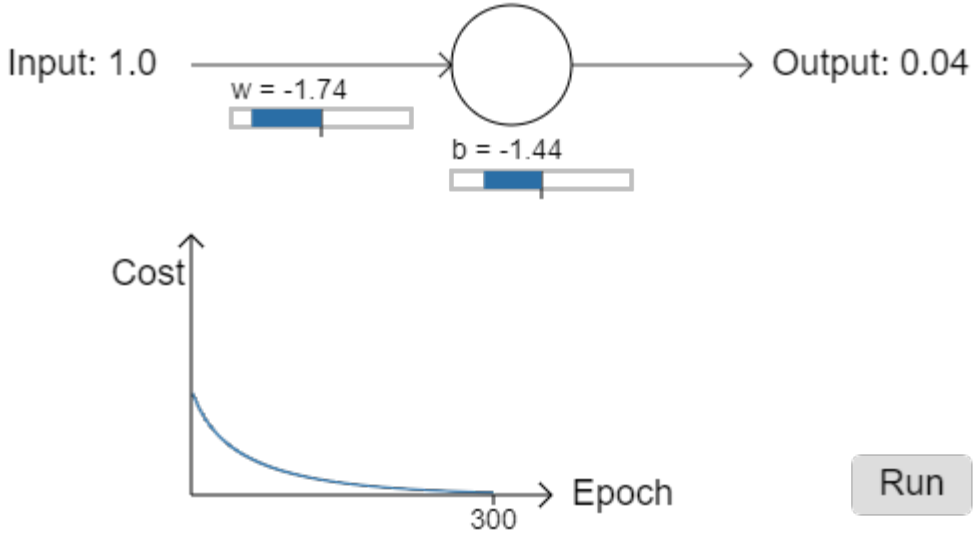
- The larger the error, the faster the neuron learns
- Avoids the learning slowdown for the quadratic cost
  - Recall that $\frac{\partial C}{\partial w} = a\sigma'(z)$ in that case
  - For cross entropy, the $\sigma'(z)$ term is cancelled
- Similar result for the bias:

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z) - y)$$

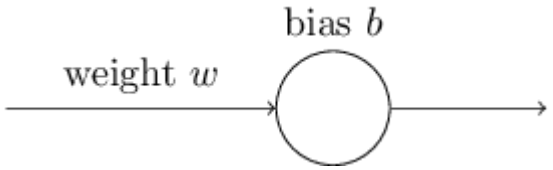
The cross-entropy cost function



- Use cross-entropy instead of quadratic cost
- First, the case where quadratic cost was fine: $w = 0.6$ and $b = 0.9$

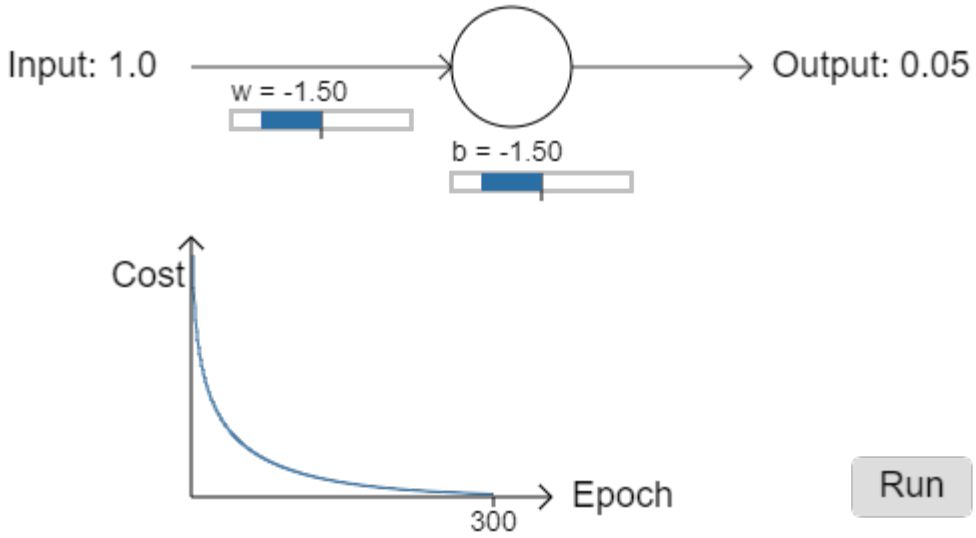


The cross-entropy cost function



- Use cross-entropy instead of quadratic cost
- Now, the case where quadratic cost got stuck: $w = 2.0$ and $b = 2$.

also learns very fast



The cross-entropy cost function

- Cross-entropy generalizes well to networks with more layers and neurons
- Let $y = y_1, y_2, \ldots$ be desired values at the output neurons
- $a_1^L, a_2^L, \ldots$ the actual outputs

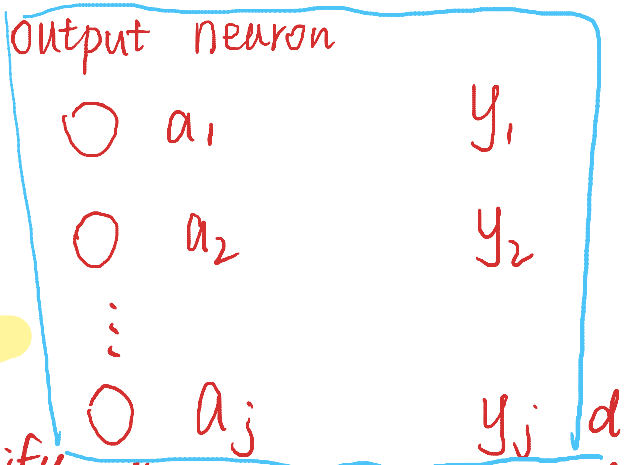
$$C = -\frac{1}{n}\sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j)\ln(1 - a_j^L)\right]$$

n examples → x j ↘ jth neuron

- Learning slowdown is avoided in this case as well

CE Loss can be used for multi-class classification

each output neuron is a binary classifier eg. neuron 2 for classify digit 9, neuron 10 for digit 0



Cross-entropy vs quadratic loss

- Cross-entropy is almost always better ***IF*** the output neurons are sigmoid neurons

near 1

result in saturated output

- Random initialization of weights and biases may result in incorrect output for some training input
  - E.g., saturated output near 1 when it should be 0 or vice versa
  - This will slow down learning with the quadratic cost but not cross-entropy

Cross-entropy applied to MNIST

- Same network as before with 1 hidden layer
- 30 hidden neurons
- Mini-batch size of 10
- $\eta = 0.5$
- Train for 30 epochs
- Resulting accuracy of 95.49%
  - Similar to 95.42% accuracy using quadratic loss
- Increase # hidden neurons to 100
- Resulting accuracy of 96.82%
  - Some improvement over 96.59% accuracy with quadratic loss

Learning rate { too high oscillate
too low long time to converge

Different derivation of cross entropy

倒推

- Suppose we discovered the learning slowdown was due to the $\sigma'(z)$ terms
- We might have wondered if it's possible to choose a cost function so those terms disappeared
- This would give for a single training input x

$$\frac{\partial C}{\partial w_j} = x_j(a - y)$$

$$\frac{\partial C}{\partial b} = (a - y)$$

error = a-y

- We can derive cross entropy from this

Where does cross-entropy come from?

$$\frac{\partial C}{\partial b} = (a - y)$$

- From the chain rule:

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma'(z)$$

- Since $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big) = a(1 - a)$,

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}a(1 - a)$$

- Comparing these equations gives

$$\frac{\partial C}{\partial a} = \frac{a - y}{a(1 - a)}$$

- Integrating wrt a gives

$$C = -[y \ln a + (1 - y)\ln(1 - a)]$$

Some intuition to cross-entropy

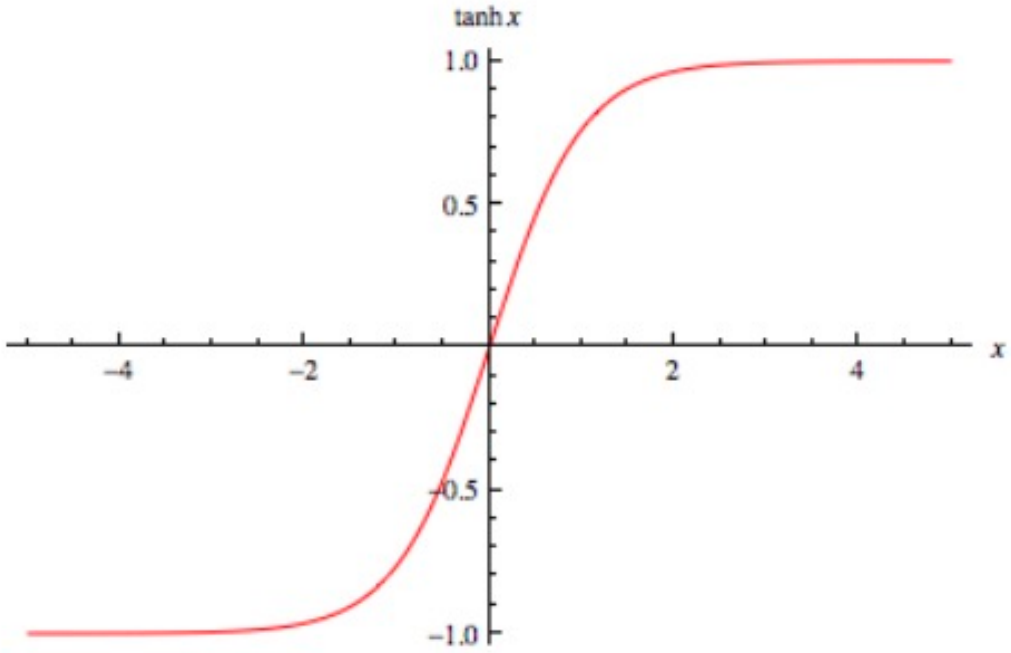
- Some intuition comes from information theory
- Cross-entropy is a measure of surprise
- Our neuron is trying to compute the function $x \rightarrow y = y(x)$
- Instead it computes $x \rightarrow a = a(x)$
  - Can think of a as the neuron's estimated probability that $y = 1$
  - Similarly, $1 - a$ is like the estimated probability that $y = 0$
- The cross-entropy measures how "surprised" we are on average when we learn the true value of y
  - Low surprise if the output is what we expect
  - High surprise if the output is unexpected

Back to the design

- We tried to tweak the cost function to have high gradients when far away from the right answer
- Can we change activation function instead?
- Thus far we have talked about sigmoidal activations
- There are many other possibilities

tanh

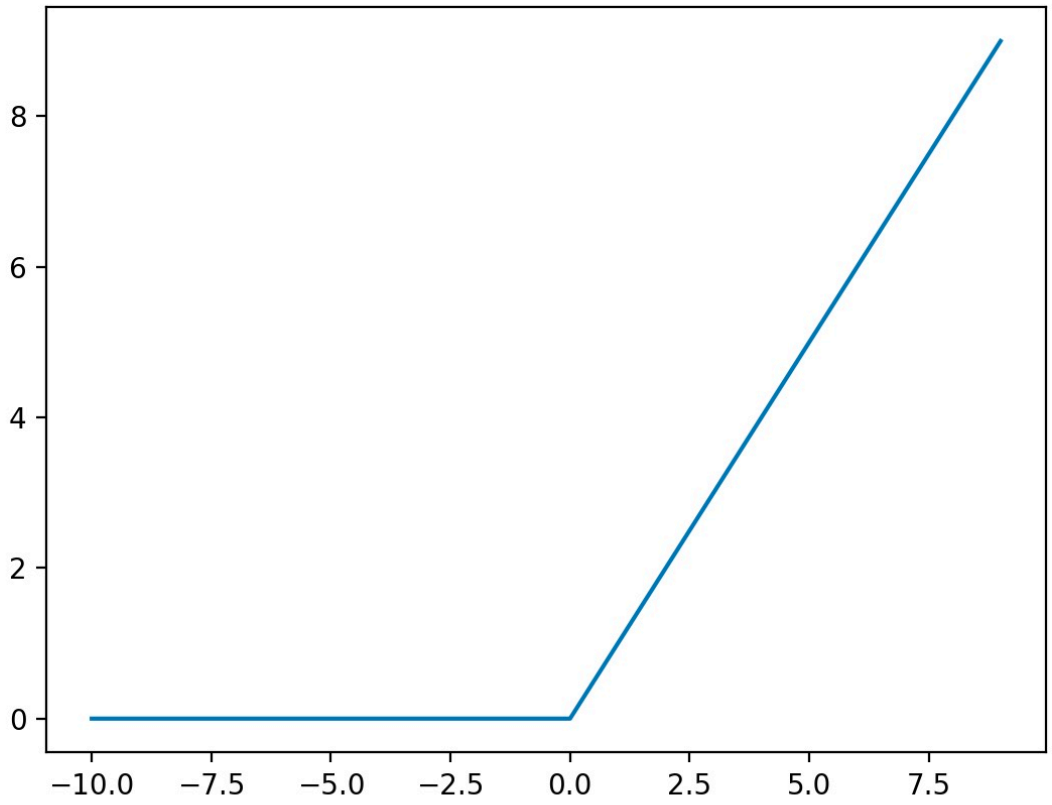
$$\tanh x = \frac{e^x - e^{-x}}{2} \div \frac{e^x + e^{-x}}{2} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Has the same problem as sigmoid (flattening gradients)

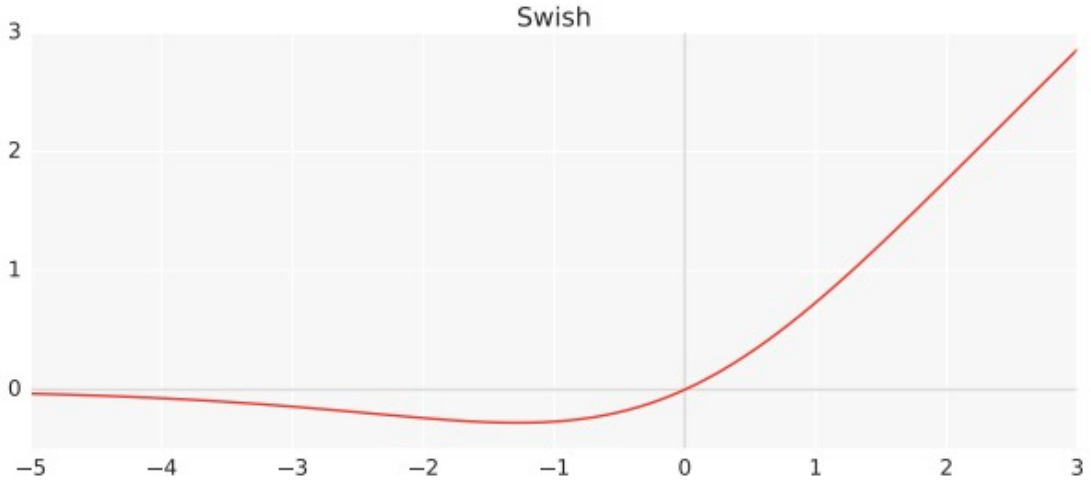
RelU

$$f(x) = x^+ = \max(0, x)$$



Swish

$$\text{swish}(x) := x \times \text{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}.$$



Softmax

preserve probability

Softmax

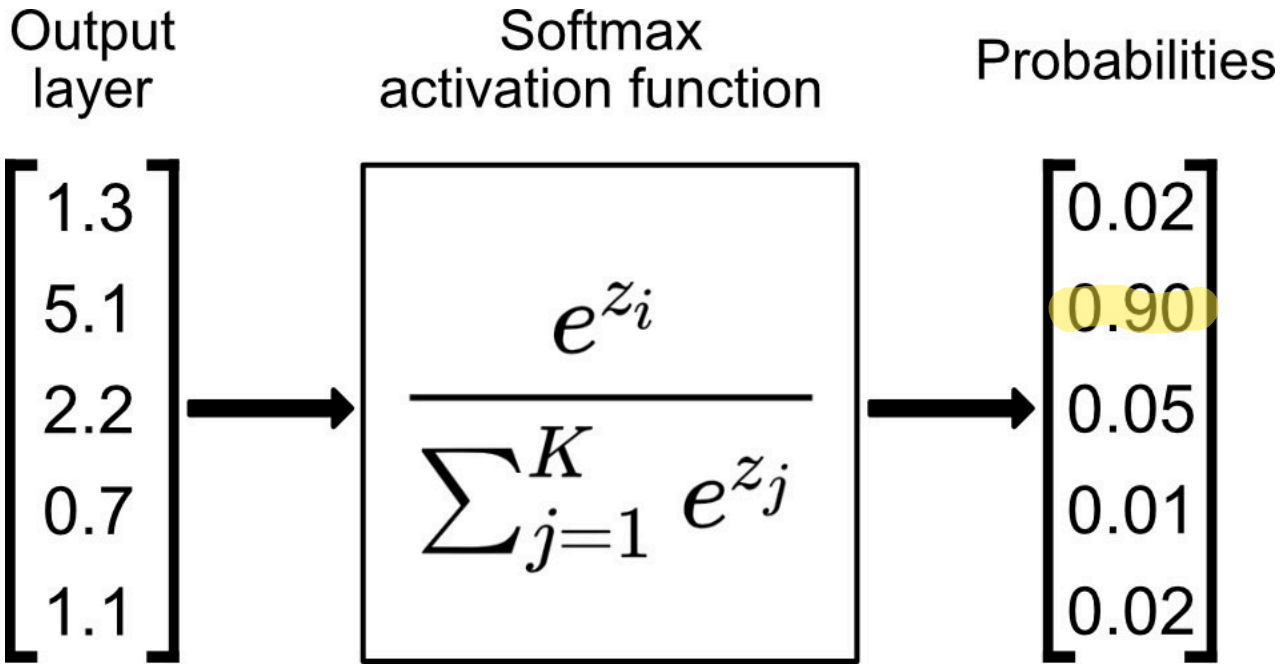
- Use the same weighted inputs $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$
- Instead of the sigmoid function, we apply the ***softmax function*** to the z_j^L:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \qquad \in [0,1]$$

Computing softmax activation

1. Exponentiate every output
2. Normalize the sum so that its equal to 1

Softmax activation



Properties of softmax

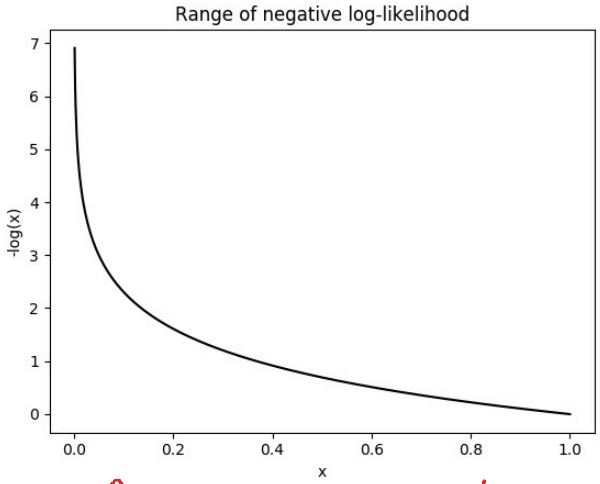
$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$$

- The output of the softmax layer can be thought of as a probability distribution
  - Can interpret the activation a_j^L as the network's estimate of the probability that the correct output is j
  - This does not occur with a sigmoid layer
- Activations always positive even if z is negative (don't have to shift or take absolute value)

Softmax enables probabilistic cost

- Recall Negative log likelihood from probability theory
- Since softmax gives the probability of the input being in many classes
- We can look at the negative log likelihood of the correct class ---always positive value
  - Measures the "unhappiness" of the network

happy = get expected output = output = ground truth



↑ small x high cost

Softmax with log-likelihood

- Log-likelihood cost of a training input x with output y:

$$C = -\ln a_y^L$$

- Example: a 7 from MNIST gives a cost of $-\ln a_7^L$
- If the network does well and is confident, it will estimate a value of a_7^L close to 1
  - Low cost

$C = -\ln 1 = 0$

- In contrast, if the network doesn't do well, a_7^L will be smaller
  - High cost

$a_7^L \to 0 \qquad -\ln a_y^L \to -\ln 0 = -(-\infty) = \infty$

Avoiding slowdown with softmax

- Learning slowdown is based on the behavior of $\partial C/\partial w_{jk}^{L}$ and $\partial C/\partial b_{j}^{L}$
- For softmax with log-likelihood cost:

$$\frac{\partial C}{\partial w_{jk}^{L}} = a_{k}^{L-1}\left(a_{j}^{L} - y_{j}\right)$$

$$\frac{\partial C}{\partial b_{j}^{L}} = a_{j}^{L} - y_{j}$$

- Same expressions when using cross-entropy!
- So which should you use?
  - In many situations both approaches work well
  - The softmax approach can be useful when interpreting the outputs as probabilities

Further Reading

- Nielsen book, chapter 3