USC University of Southern California

ECE 105: Introduction to Electrical Engineering

Lecture 17

Neural Networks

Yasser Khan

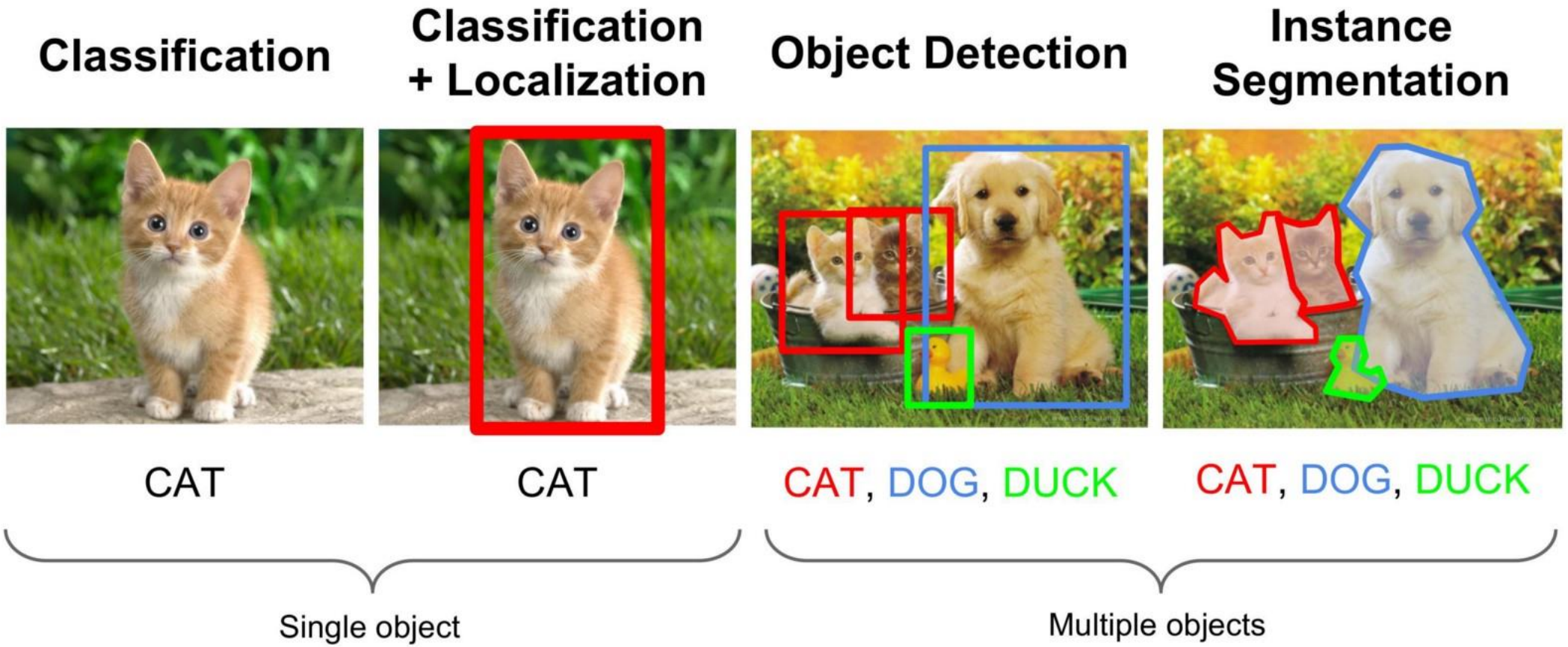
Rehan Kapadia

Timeframe

USC University of Southern California

- 1943: McCulloch–Pitts “neuron”
  - Started the field
- 1962: Rosenblatt’s perceptron
  - Learned its own weight values; convergence proof
- 1969: Minsky & Papert book on perceptrons
  - Proved limitations of single-layer perceptron networks
- 1982: Hopfield and convergence in symmetric networks
  - Introduced energy-function concept
- 1986: Backpropagation of errors
  - Method for training multilayer networks
- Present: Probabilistic interpretations, Bayesian and spiking networks

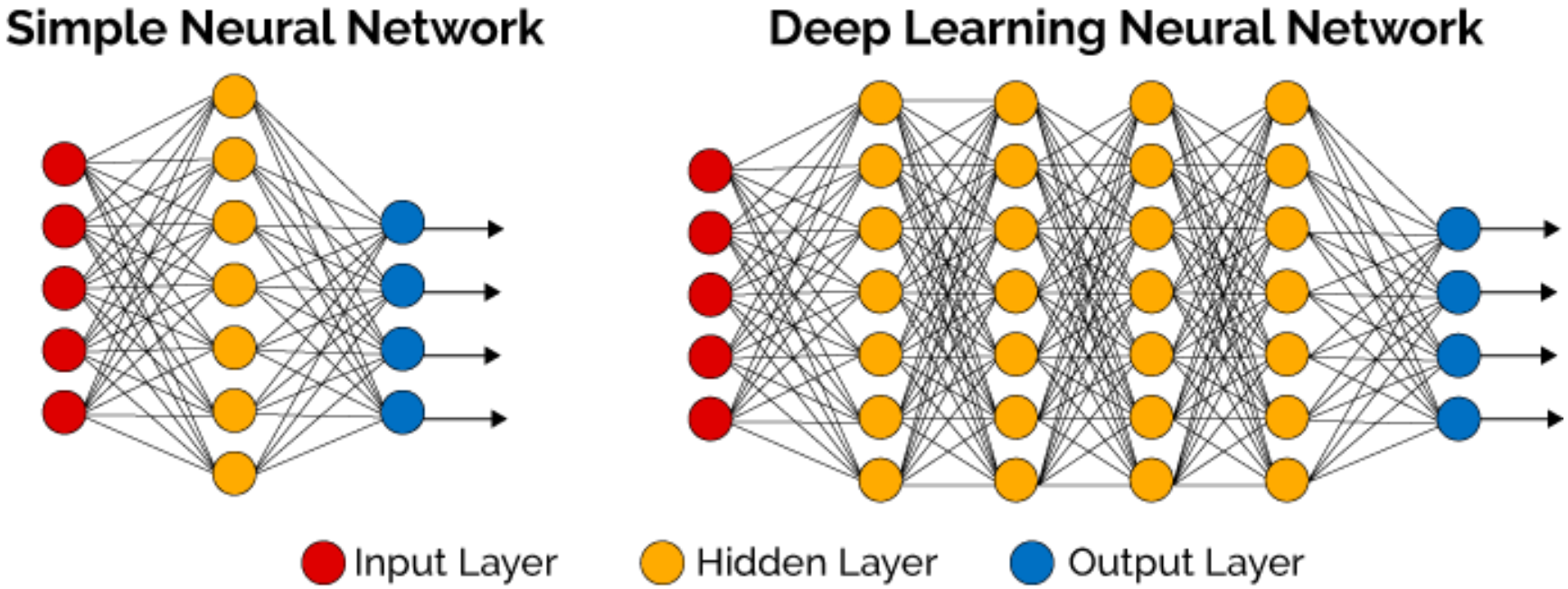
Why Deep Neural Networks?



- Deep neural networks are excellent for certain classes of problems, such as image recognition, speech recognition etc.
- Importantly, as the amount of data scales, the performance of deep learning continues to improve

Image from Medium: “Review of Deep Learning Algorithms for Object Detection”

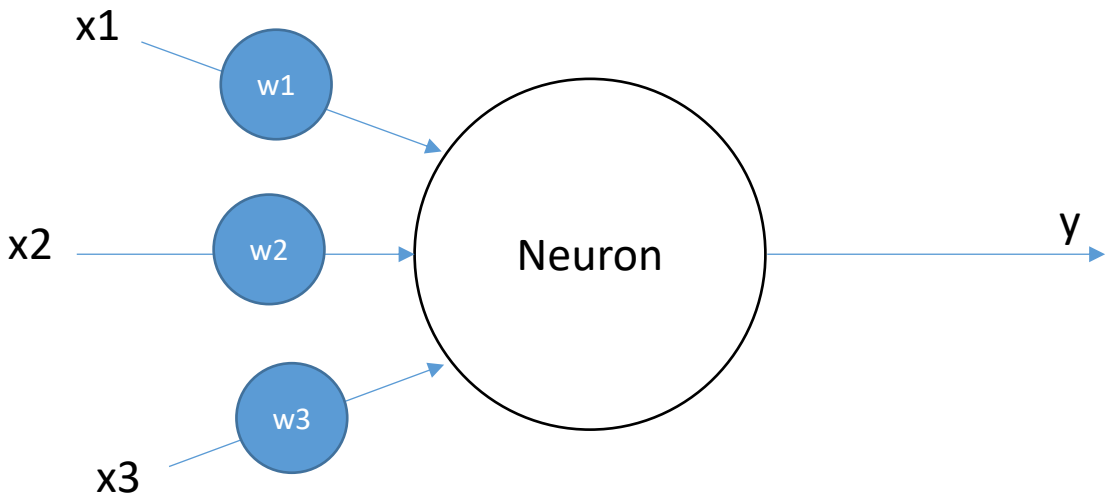
What's so deep about Deep Neural Networks?



- The "deep" refers to the number of layers of 'units' between the input layer and output layer.
- Since this is a neural network, the units are called neurons

https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351

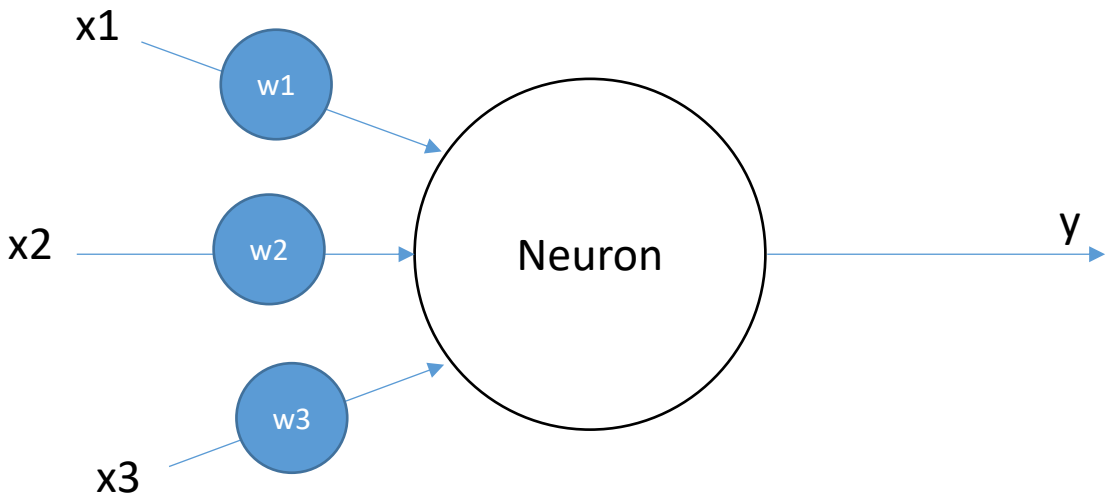
Fundamental Unit of a Neural Network



Mathematical Definition Y = f(w1*x1 + w2*x2 + w3*x3 + b)

- The above is the fundamental unit of a neural network
  - W_i refers to the weights, sometimes called synaptic weights
  - The neuron is the mathematical function that connects inputs to outputs
  - X_i refers to the inputs
  - Y refers to the output

Fundamental Unit of a Neural Network



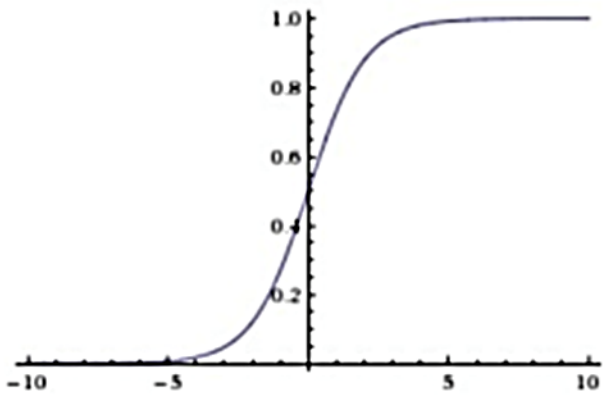
Mathematical Definition Y = f(w1*x1 + w2*x2 + w3*x3 + b)

- Many different functions are used for the neuron, but the three most popular are
  - Sigmoid: $\sigma(x) = 1/(1+\exp(-x))$
  - Rectified Linear Unit (ReLU): $f(x) = \max(0,x)$
  - tanh: $\tanh(x) = 2\sigma(2x) - 1$

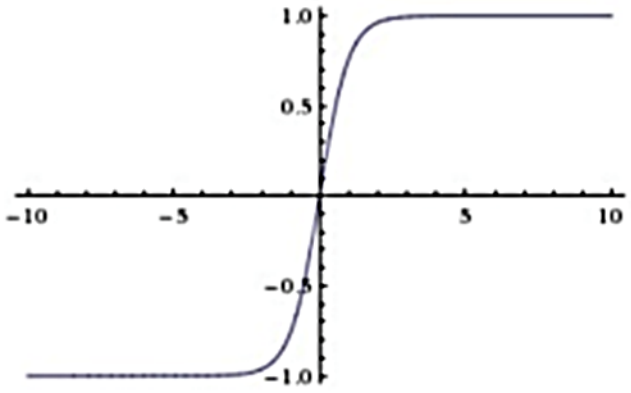
Fundamental Unit of a Neural Network

USC University of Southern California

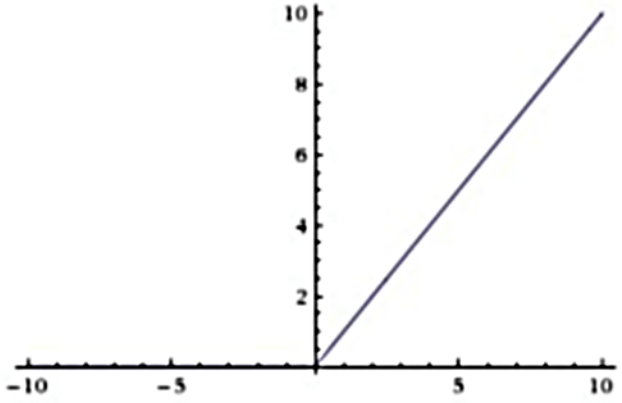
Sigmoid: $\sigma(x) = 1/(1+\exp(-x))$



$\tanh(x) = 2\sigma(2x) - 1$

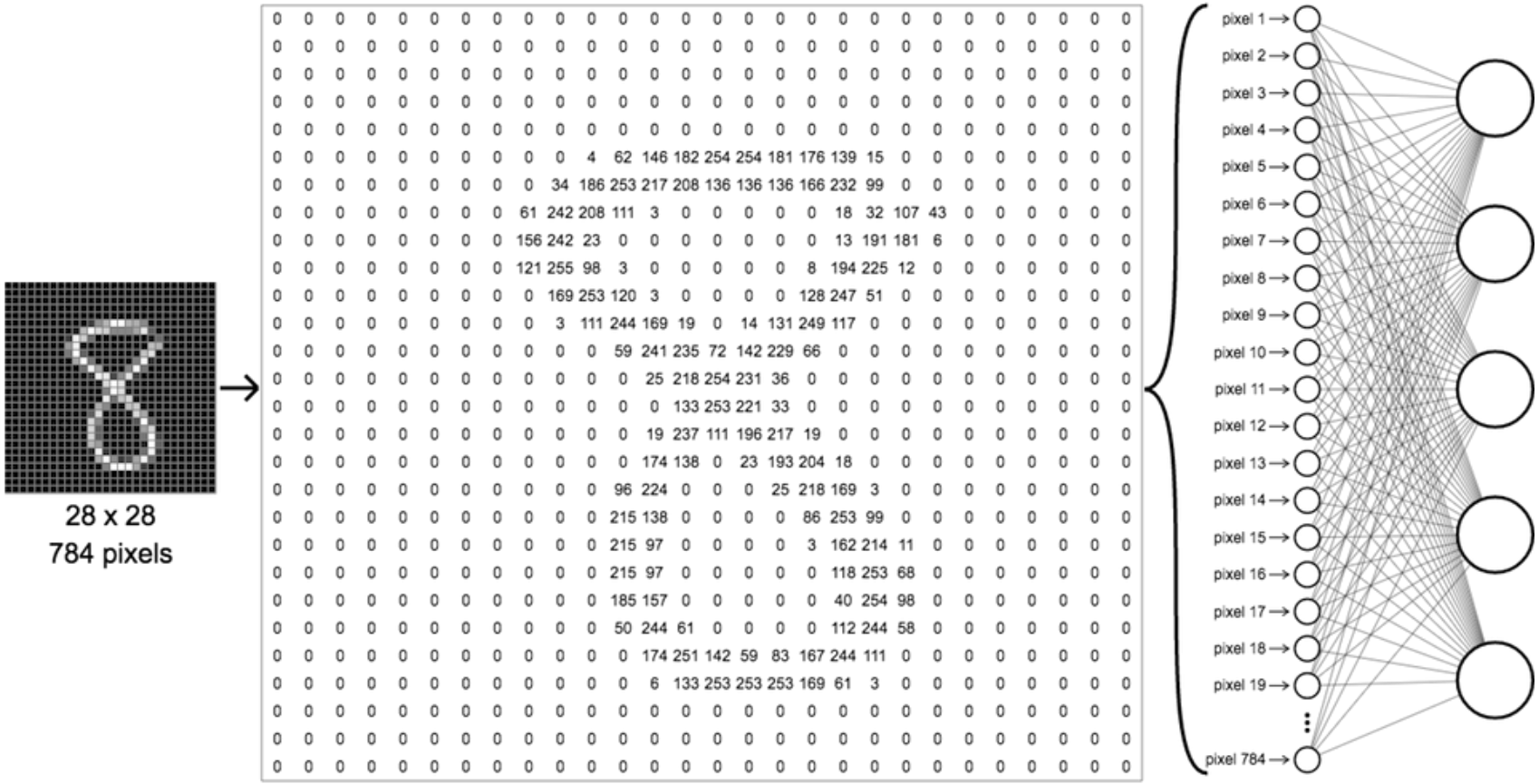


ReLU: $f(x) = \max(0,x)$



- Many different functions are used for the neuron, but the three most popular are
  - Sigmoid: $\sigma(x) = 1/(1+\exp(-x))$
  - Rectified Linear Unit (ReLU): $f(x) = \max(0,x)$
  - tanh: $\tanh(x) = 2\sigma(2x) - 1$

Operation of a neural network

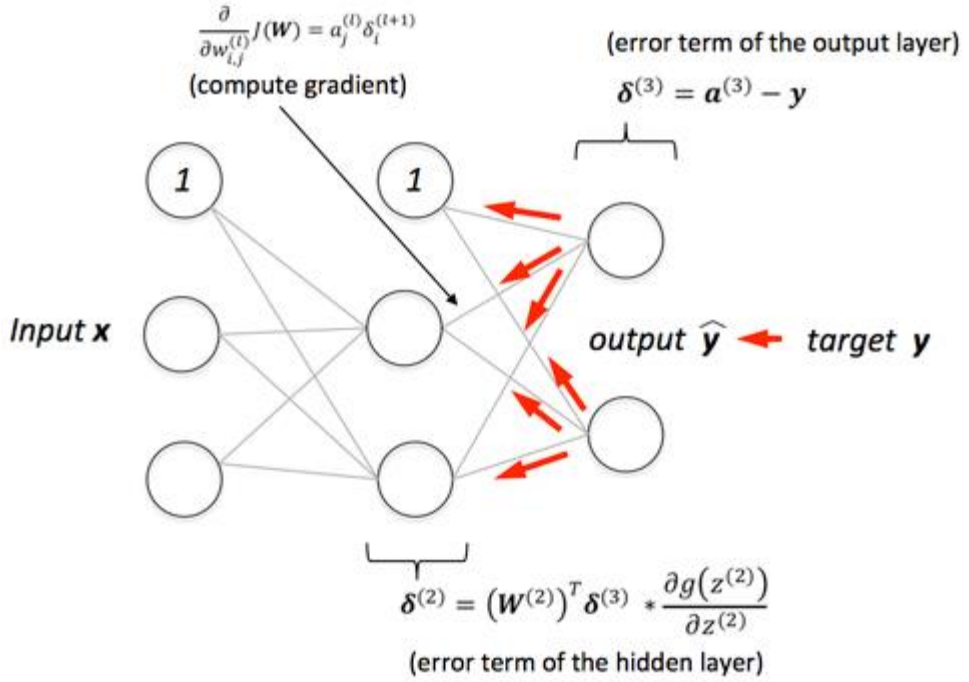


- Image is made up of a set number of pixels
  - Each pixel becomes an input
  - Example here: each pixel is from 0 to 256 in greyscale
  - Each input is connected to all the neurons of the hidden layer
  - Adjusting the weights will then eventually allow recognition of the digits

https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351

Training of an Artificial Neural Network

USC University of Southern California



- Backpropagation algorithm a form of supervised learning
  - We initialize all the weights of the ANN, and then feed the data into it
  - Each training example generates an output.
  - We know the correct output for those training samples, so for each output, we can define an error
  - Then, we can essentially go backwards, assigning some part of that error to the weights between output neuron and previous neurons
  - Adjusting the weights allows us to move towards the correct answer, as defined by us

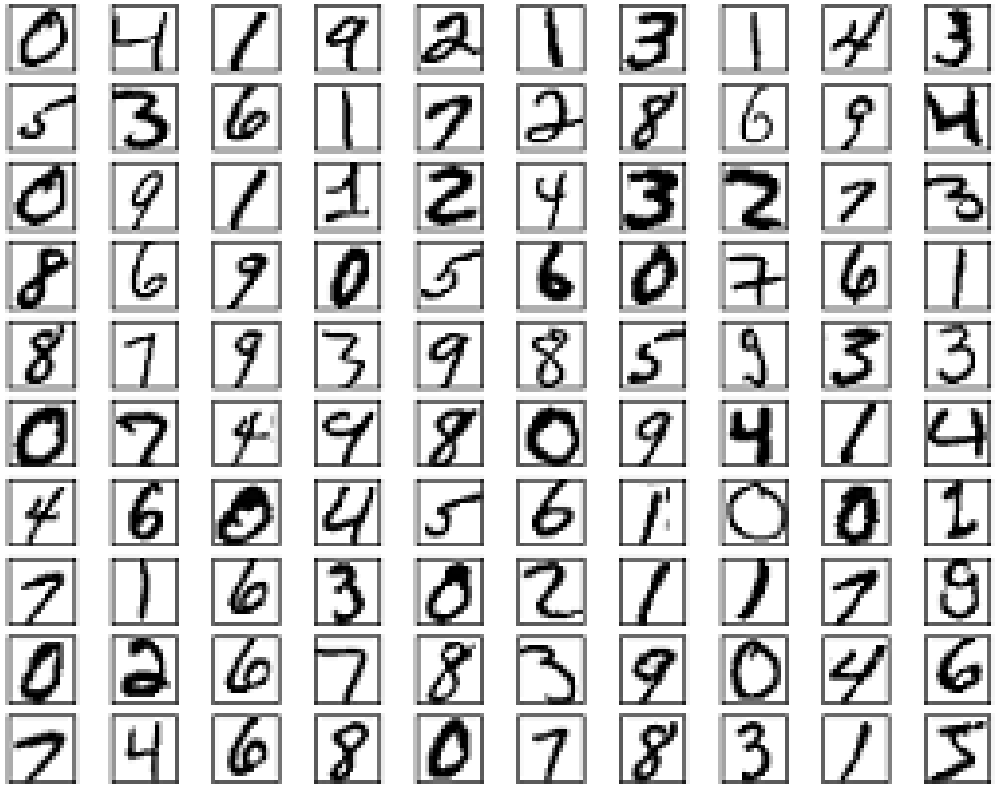
https://github.com/rasbt/python-machine-learning-book/blob/master/faq/visual-backpropagation.md

Handwriting Recognition

USC University of Southern California

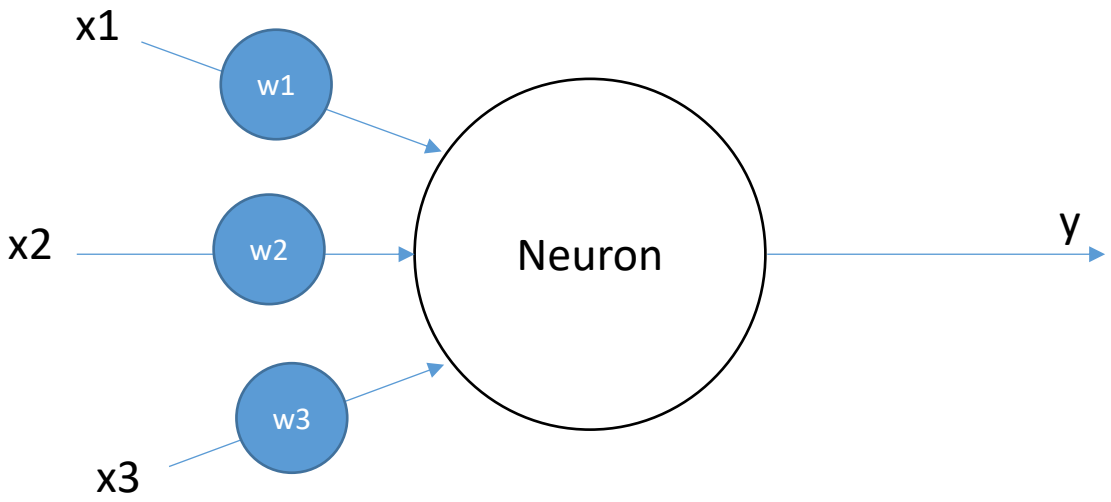
- We can easily recognize these numbers
  - But this uses a huge number of neurons (~billion or more) and an even larger number of connections/synapses (100's of billions)

Approach



- Need a large set of training data
  - Also need a set on which to test your trained model

Perceptron



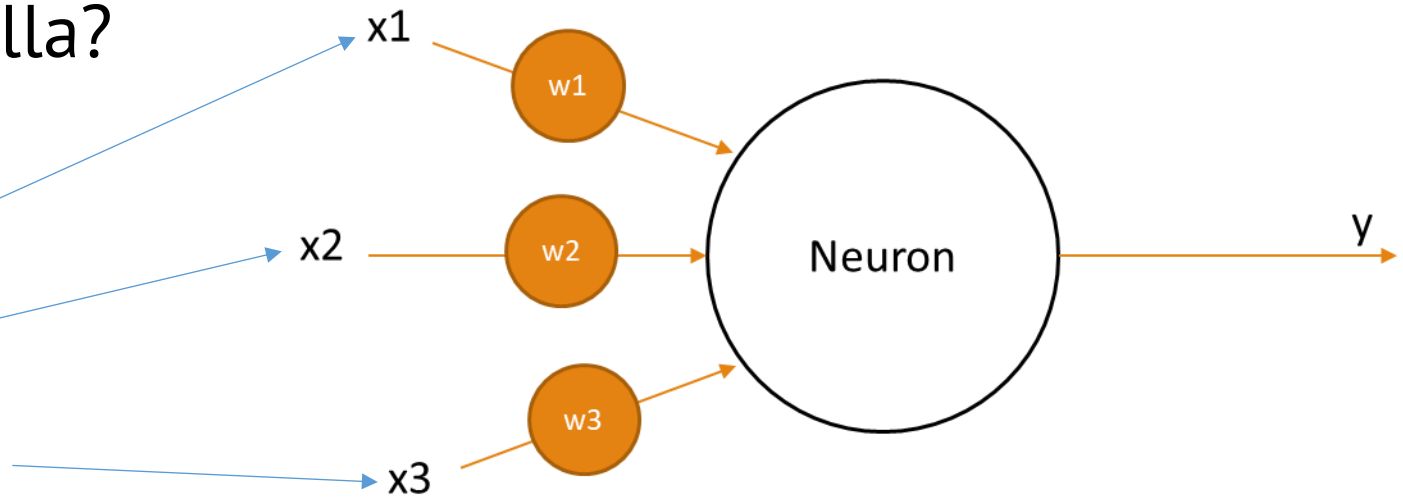
Mathematical Definition Y = f(w1*x1 + w2*x2 + w3*x3 + b)

- For the perceptron:
  - Output = 0 if $\Sigma w^{*}x \leqslant$threshold
  - Output = 1 if $\Sigma w^{*}x >$threshold

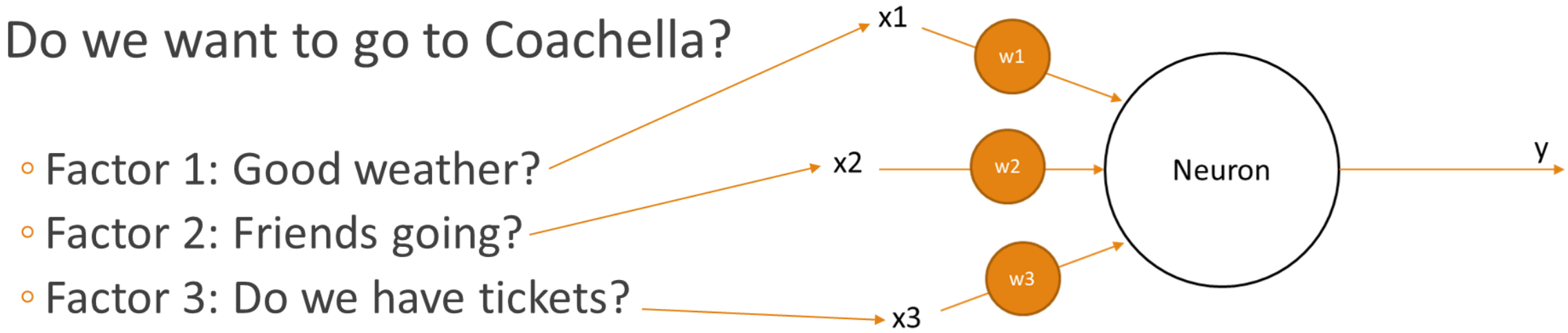
What can we answer with a perceptron?

USC University of Southern California

- Do we want to go to Coachella?
  - Factor 1: Good weather?
  - Factor 2: Friends going?
  - Factor 3: Do we have tickets?



How would we decide this?

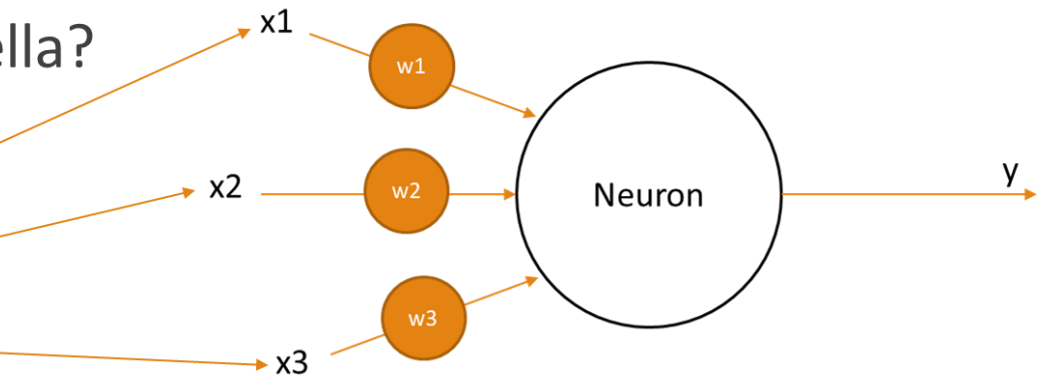


- Each variable is either a 1 or 0
  - The decision making happens in the weights
  - Here we have to assign weights and a threshold value

How would we decide this?

Do we want to go to Coachella?

- Factor 1: Good weather?
- Factor 2: Friends going?
- Factor 3: Do we have tickets?



Threshold = 10

w1 = 5

w2 = 5

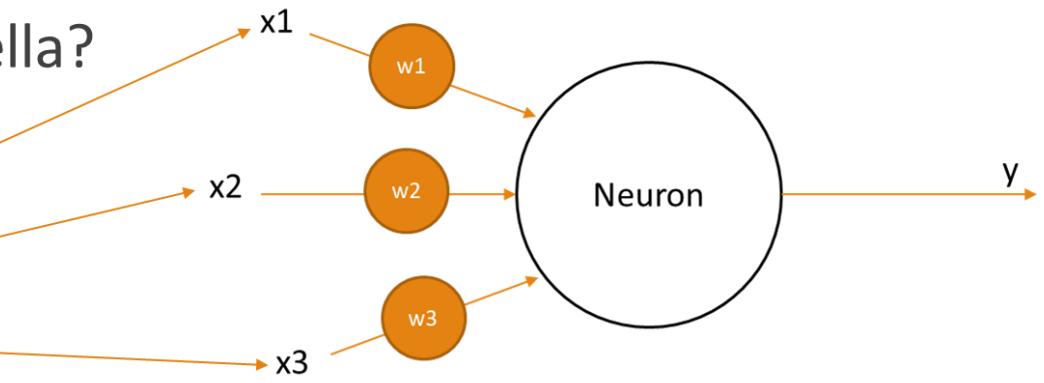
w3 = 11

This is a silly distribution. Either you have tickets and you're going, or you don't and you aren't.

How would we decide this?

Do we want to go to Coachella?

- Factor 1: Good weather?
- Factor 2: Friends going?
- Factor 3: Do we have tickets?



Threshold = 12

w1 = 5

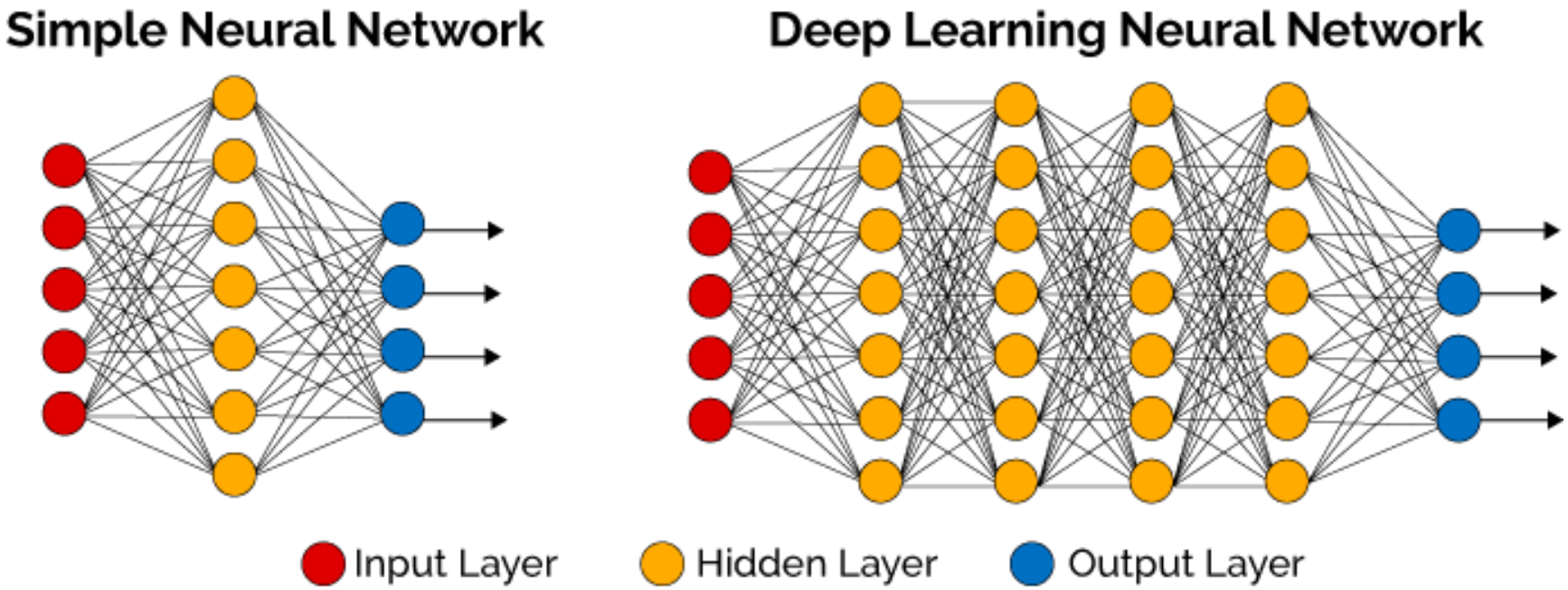
w2 = 5

w3 = 11

Now this basically says that we only go if we have tickets AND one of the other conditions are true, but not both of them.

This is a silly example, but a reasonable starting point to think about how we build up complex decision making.

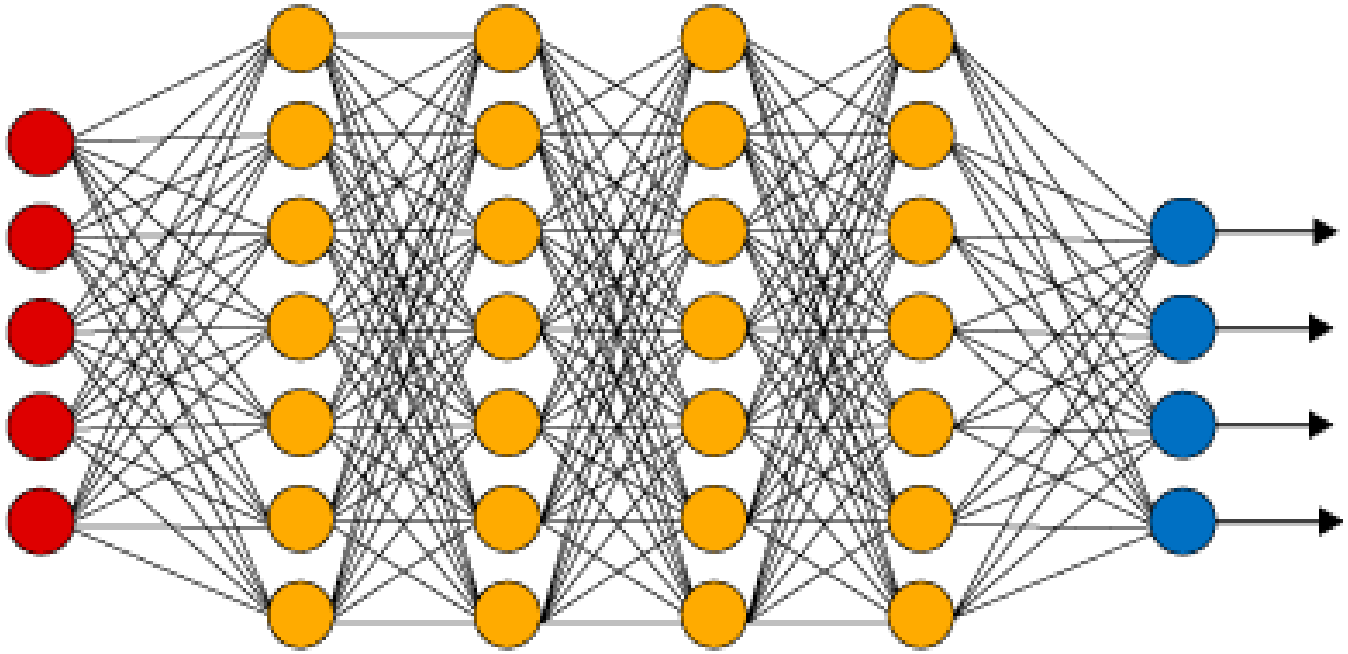
Addition of other nodes increases complexity



- We can now make decisions based on more inputs, and have a more nuanced way of getting to the outputs.

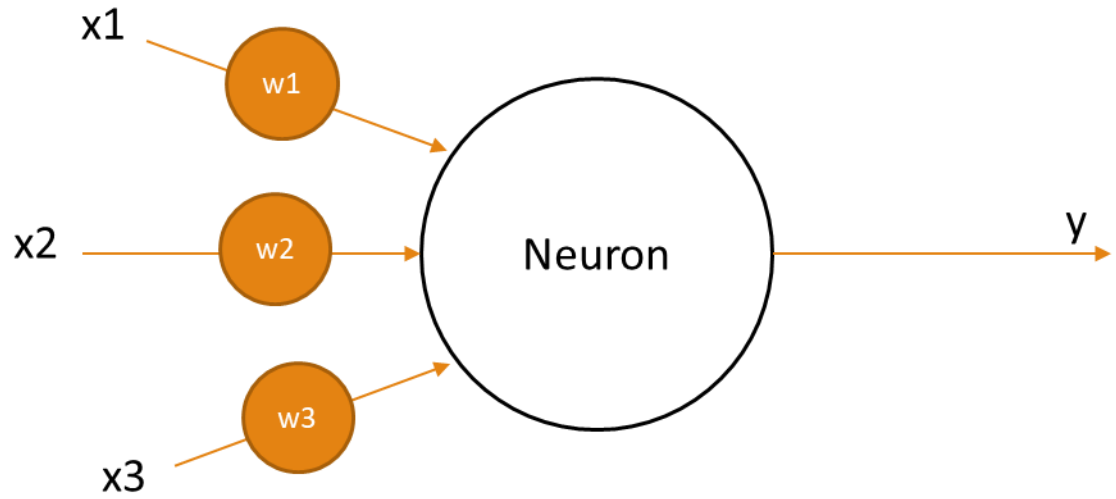
How does information change through layers?

USC University of Southern California



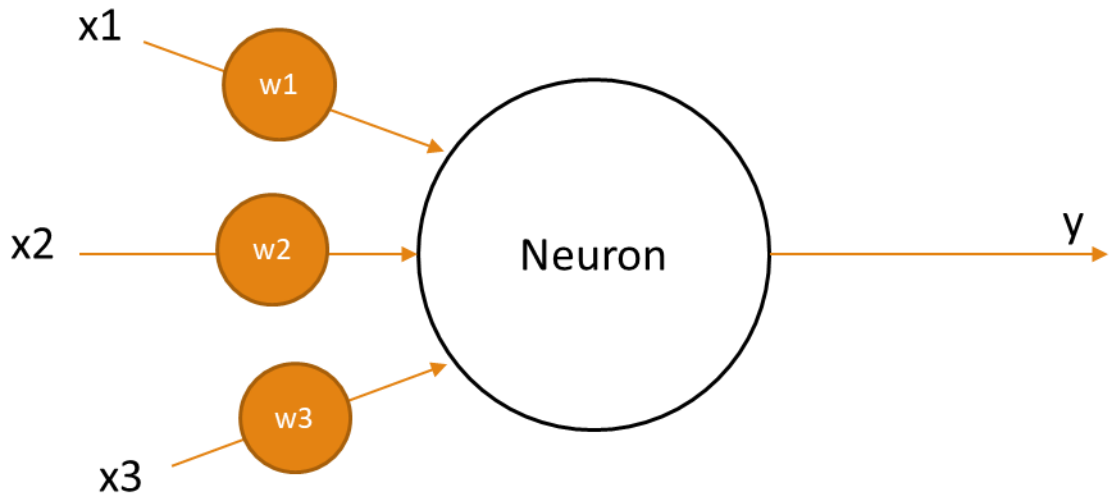
- As we move through the layers of the DNN, we are somewhat making decisions on more and more abstract items

Perceptrons



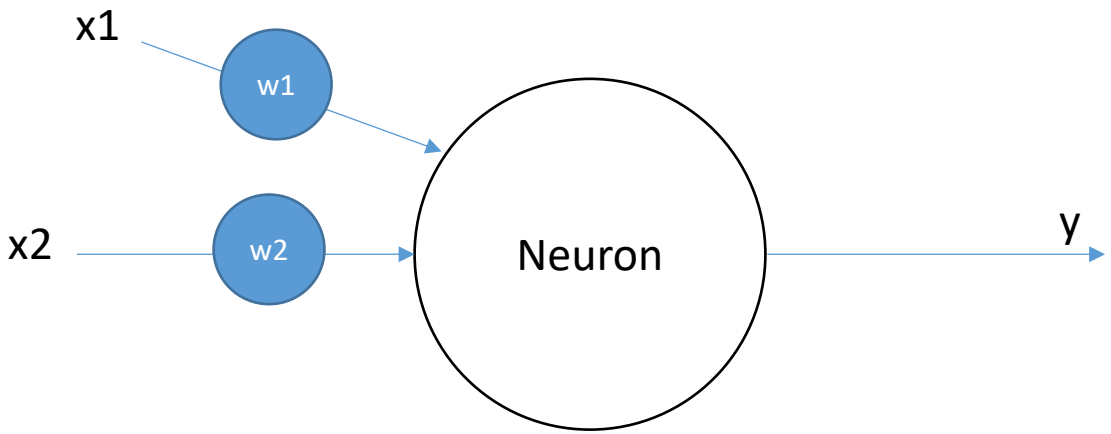
- We can assume
  - Inputs as a vector x = {x1, x2, x3}
  - Weights as vector w = {w1, w2,w3}
  - Then sum is just the dot product, and the become
    - 0 when when $x \cdot w + b \leq 0$
    - 1 when when $x \cdot w + b > 0$

Perceptrons



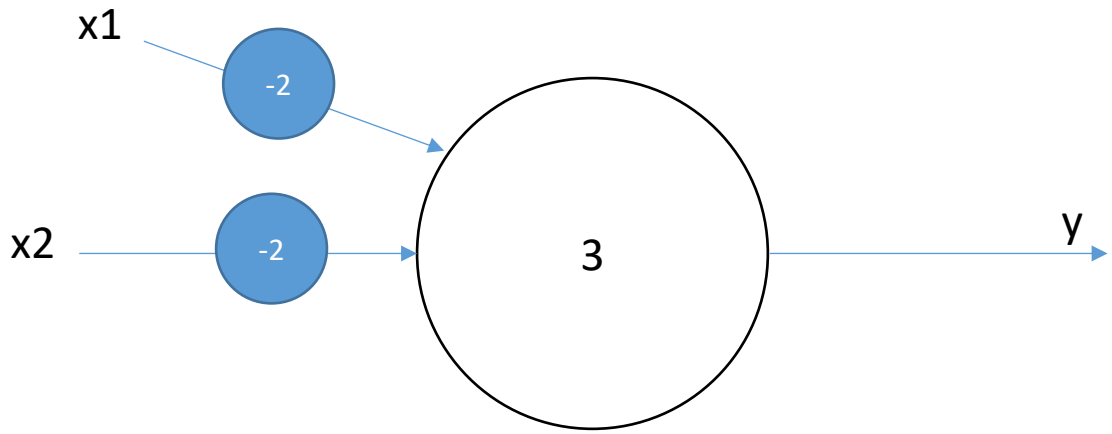
- 0 when when $x \cdot w + b \leq 0$
- 1 when when $x \cdot w + b > 0$
  - b can be thought of the bias
  - Large positive b -> neuron that fires on most inputs
  - Negative b-> neuron that rarely fires

Perceptrons for logical functions?



- How could we implement
  - AND
  - OR
  - NAND

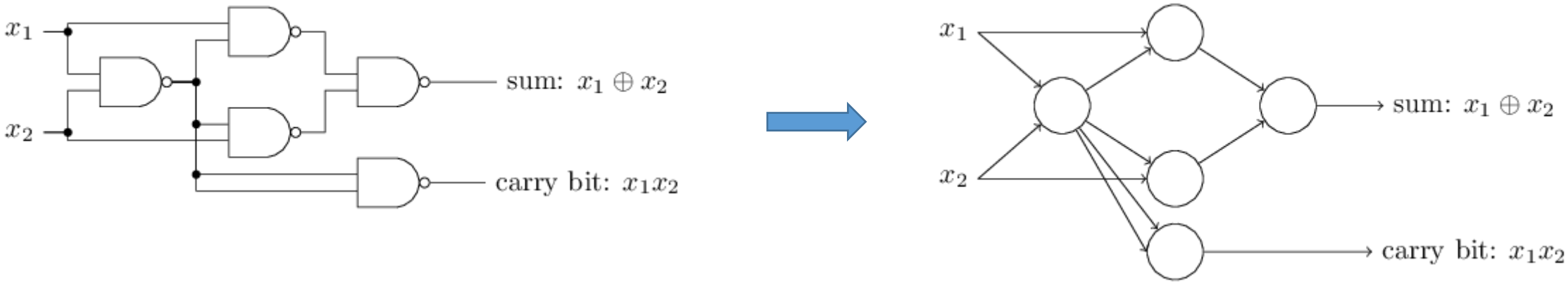
Perceptrons for logical functions?



X_1	X_2	Y
0	0	1
0	1	1
1	0	1
1	1	0

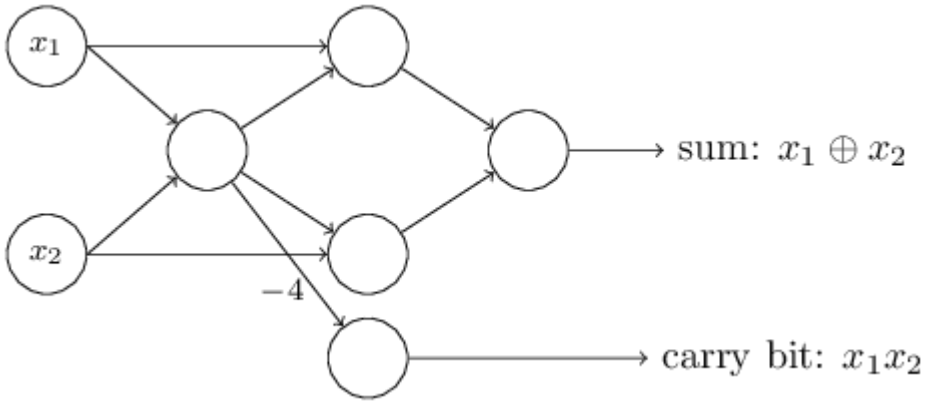
- NAND function
  - Since NAND is logically complete, we can actually implement any logical function using a perceptron

Adder



- We can actually build an adder using perceptrons!

Notation for perceptrons



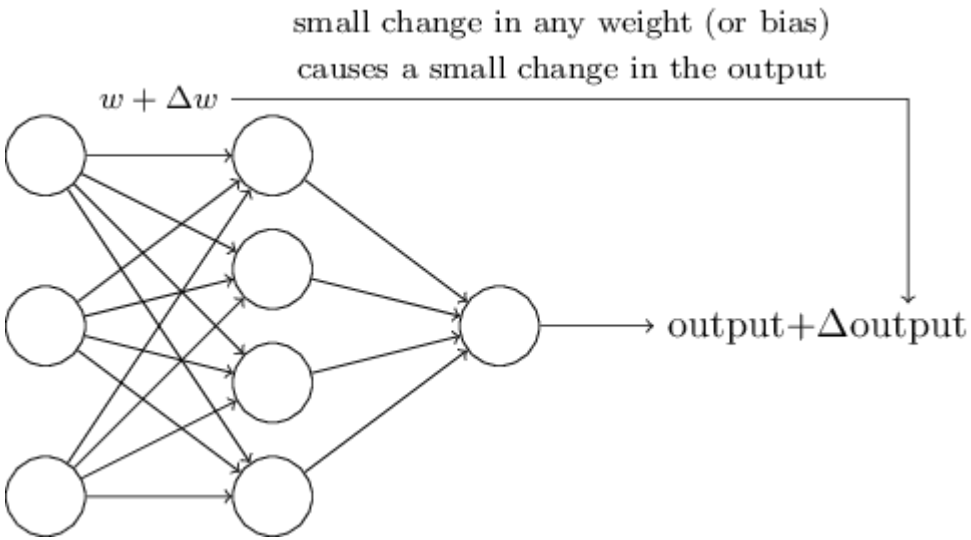
- Note that inputs in ANNs are also usually drawn as circles
  - But no inputs, only an output

So are perceptrons just more complex NAND gates?

USC University of Southern California

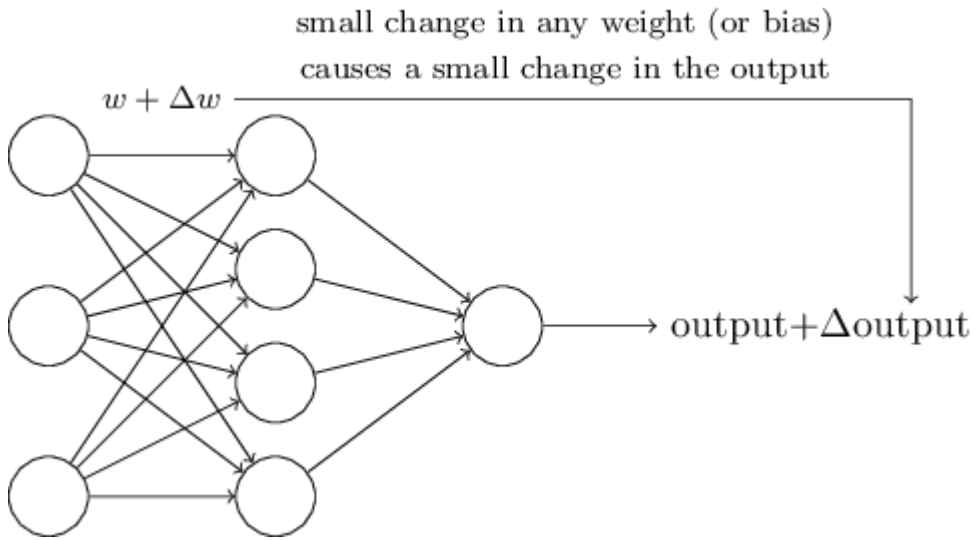
- No
  - The key strength lies in the algorithms which allow us to tune the weights
  - It also needs to be something that's done with just the input, and not something that relies on a human
- What's so important about this?
  - When we create a circuit, we basically presolve some set of problems
  - Then we lay out the circuit to solve those
  - However, with our 'learning' approach, we don't have to do that. We just create some basic architecture, and let it automatically tune itself to get the right answer

Sigmoid Neurons



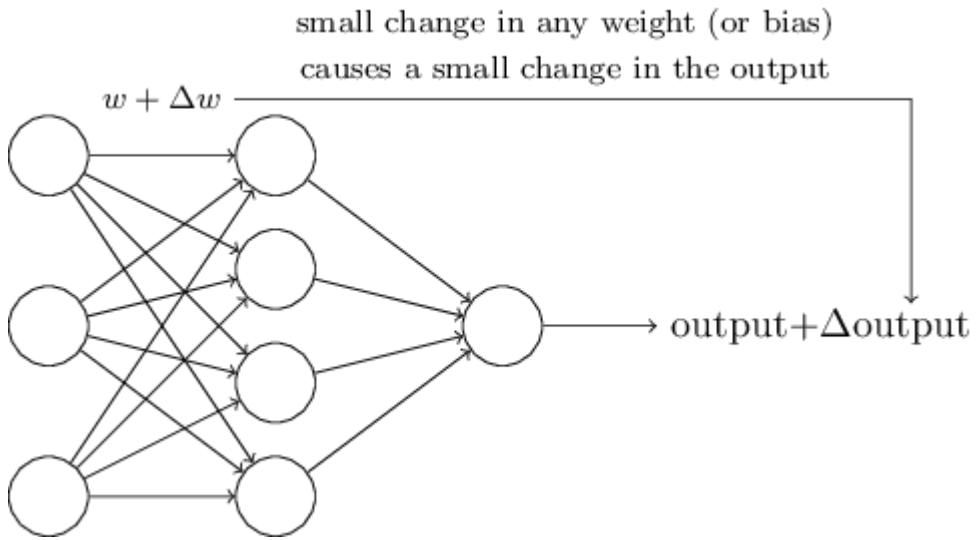
- If we want to learn, then we should have a small change in weight in some synapse to have a corresponding small change in the output

Sigmoid Neurons



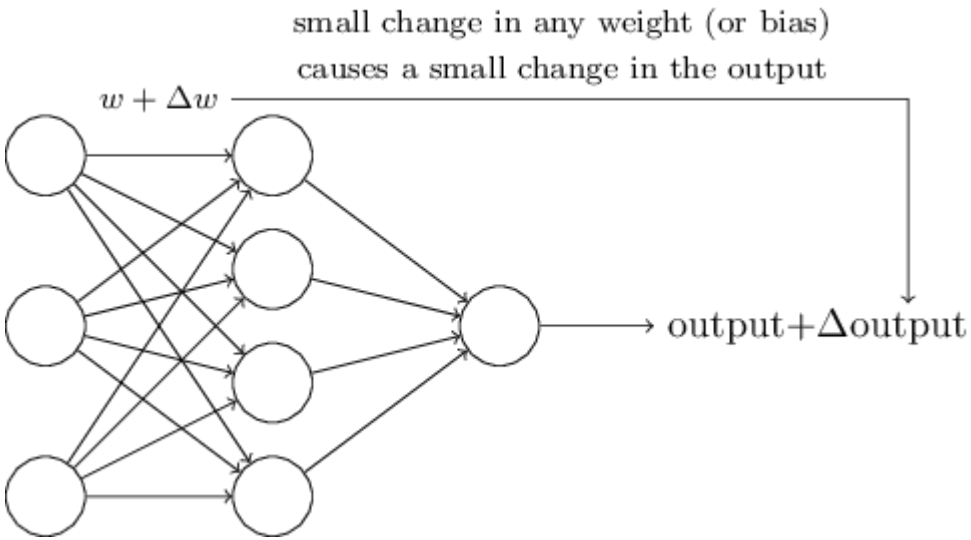
- Why is this important?
  - Let's say we feed an input which is a handwritten digit
    - It's a 3
  - Our NN classifies it as a 4…
    - It would be great if we could just change it slightly and have a small change in our output

Sigmoid Neurons



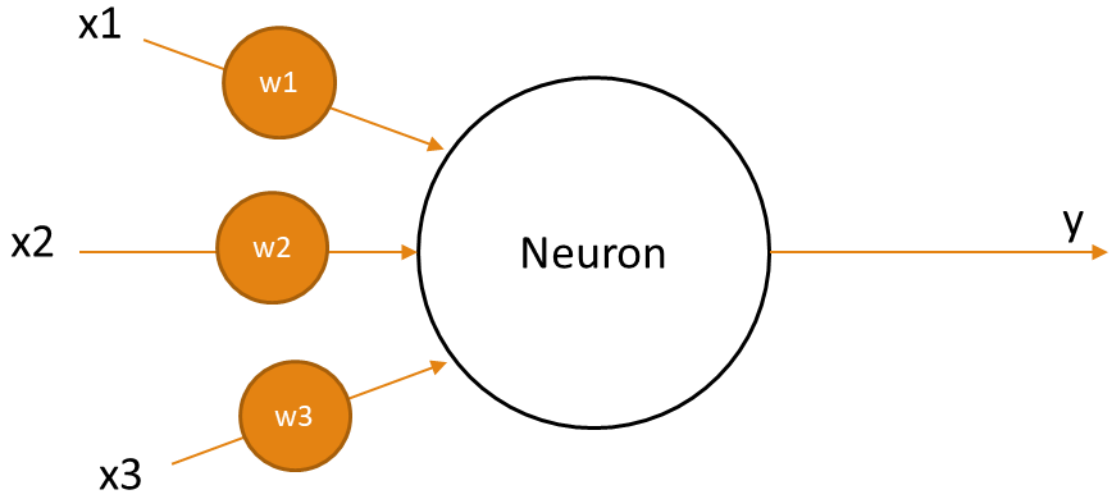
- Why is this important?
  - But if we have a perceptron, then any small weight change can invert the neuron output
    - This then propagates down the NN and can radically change everything in a fashion that is complex to precalculate
  - You fixed your incorrect classification of the 3, but messed up others

Sigmoid Neurons



- Why is this important?
  - So with perceptrons, it is challenging to have an algorithm that will allow you to easily learn complex data
  - To circumvent this, the sigmoid neuron was introduced

Sigmoid Neurons

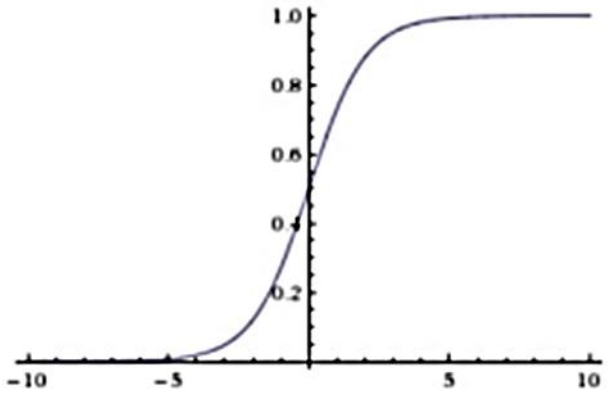


- Looks the same as before…
  - However, inputs can now be anything between 0 and 1, not just 0 or 1.
  - Equation is now more complex:
    - $\sigma(x \cdot w+b)=\sigma(z)=\frac{1}{1+e^{-z}}$

Similarities between Perceptrons and Sigmoids

USC University of Southern California

- $\sigma(x \cdot w+b)=\sigma(z)=\frac{1}{1+e^{-z}}$
  - When z is large positive number, then $\sigma(z) \sim 1$
  - When z is large negative number, then $\sigma(z) \sim 0$
  - Key difference is that we have smoothed out any sudden changes…



Why is smoothness crucial?

USC University of Southern California

- $\sigma(x \cdot w+b)=\sigma(z)=\frac{1}{1+e^{-z}}$
  - Now, when we have a small change in a specific weight Δw_i
  - We also have a small change in the output

$$\Delta out \sim \sum_j \frac{\partial out}{\partial w_j} \Delta w_j + \frac{\partial out}{\partial b} \Delta b$$

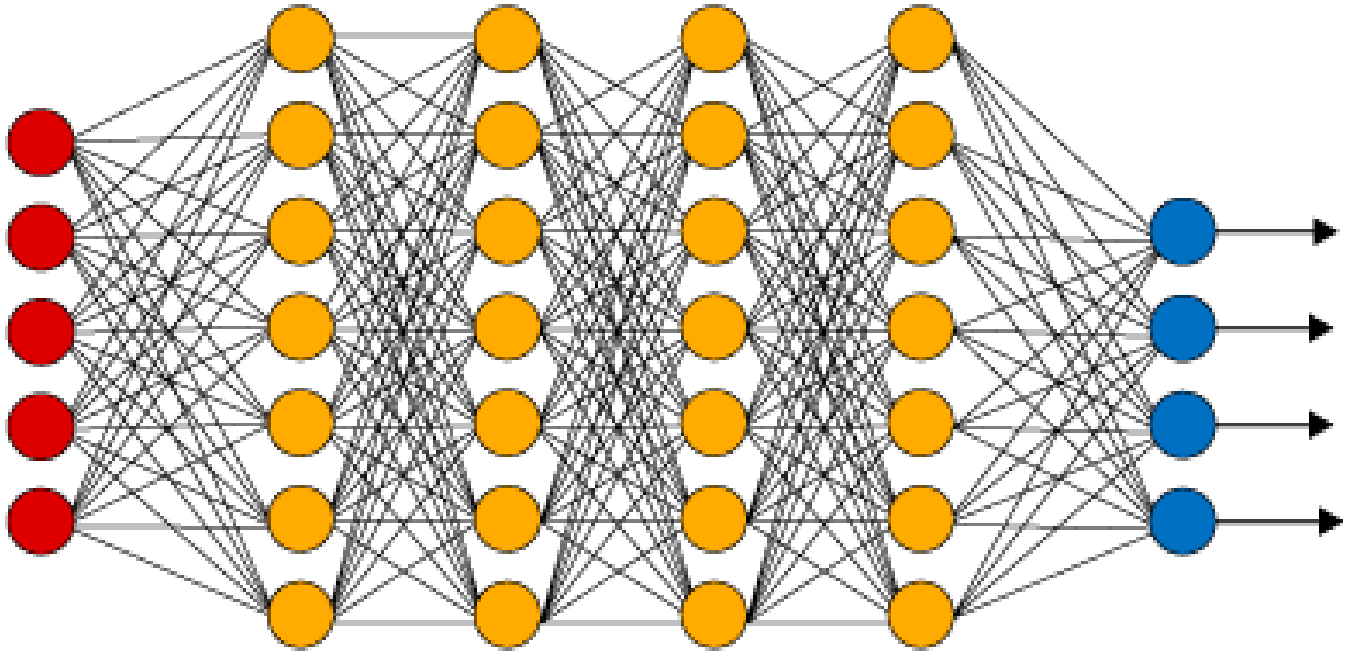
  - The sum is over all the weights, w_j, and $\frac{\partial out}{\partial w_j}$, $\frac{\partial out}{\partial b}$ represent the partial derivatives of the output with respect to the weight or bias.
  - So in this range, we essentially have a linear equation approximating our output

Sigmoid Neurons

USC University of Southern California

- In the end the actual neuron doesn't matter that much
  - Smoothness lets us use the approximation given in the last slide
  - It's a bonus if computing the derivatives are easy to do, because we will be doing a lot of that when training.

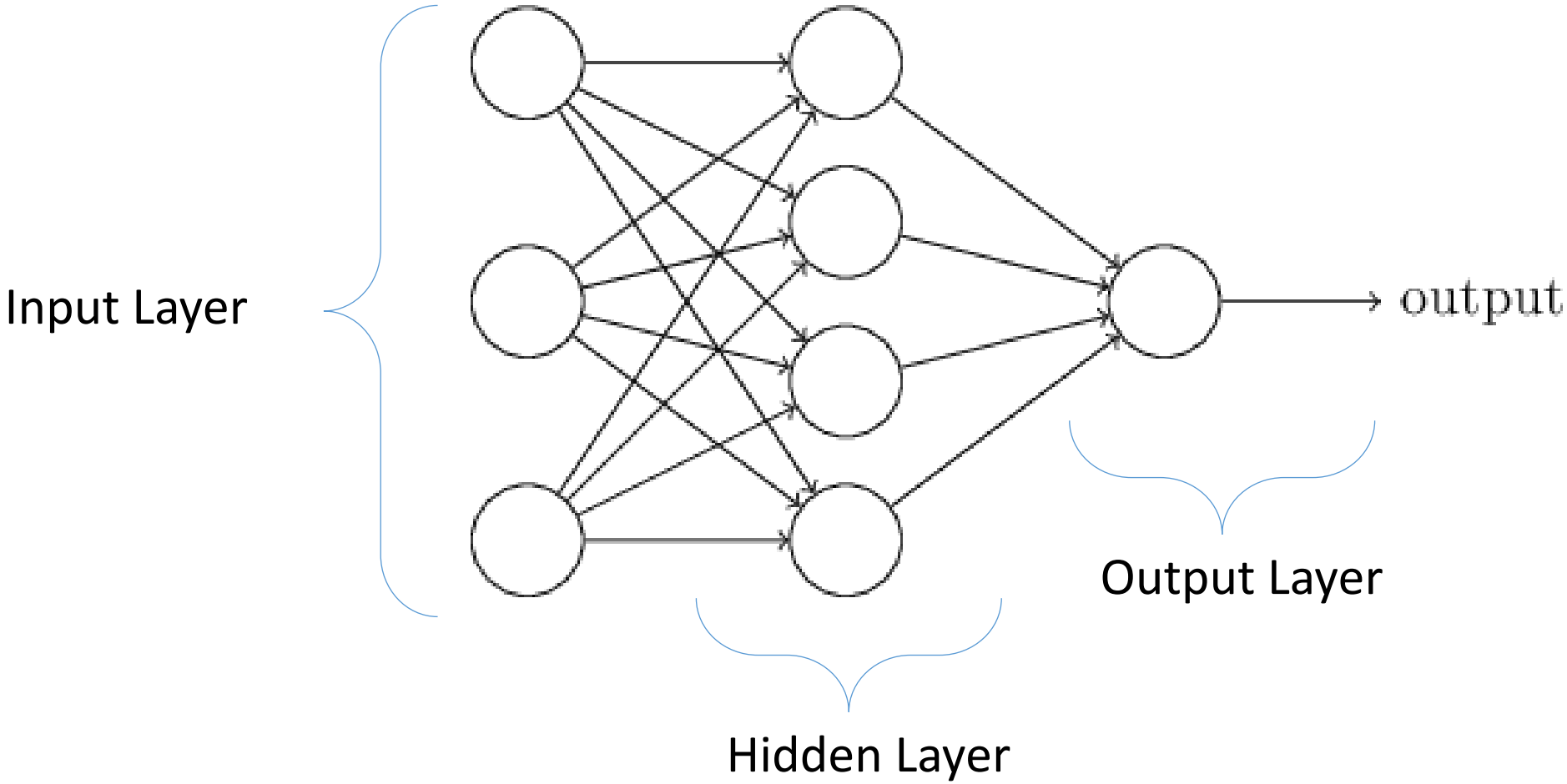
Sigmoid Neurons



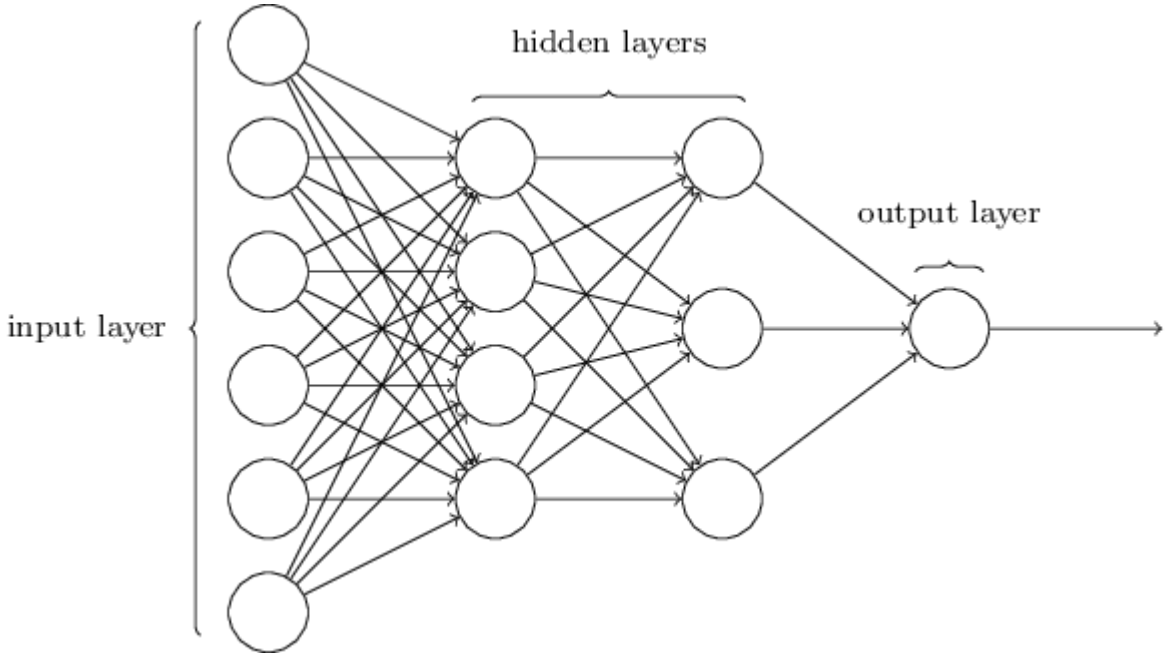
- Now, if we have our NN
  - Perceptron neurons are easy to evaluate
    - 0 or 1; yes, or no;
  - Since sigmoids are continuous, we have to setup some criteria to decide what that output neuron is saying
    - Maybe if a value is >0.5, then that input is a 9.

Neural Network Architecture

Three Layer Neural Network

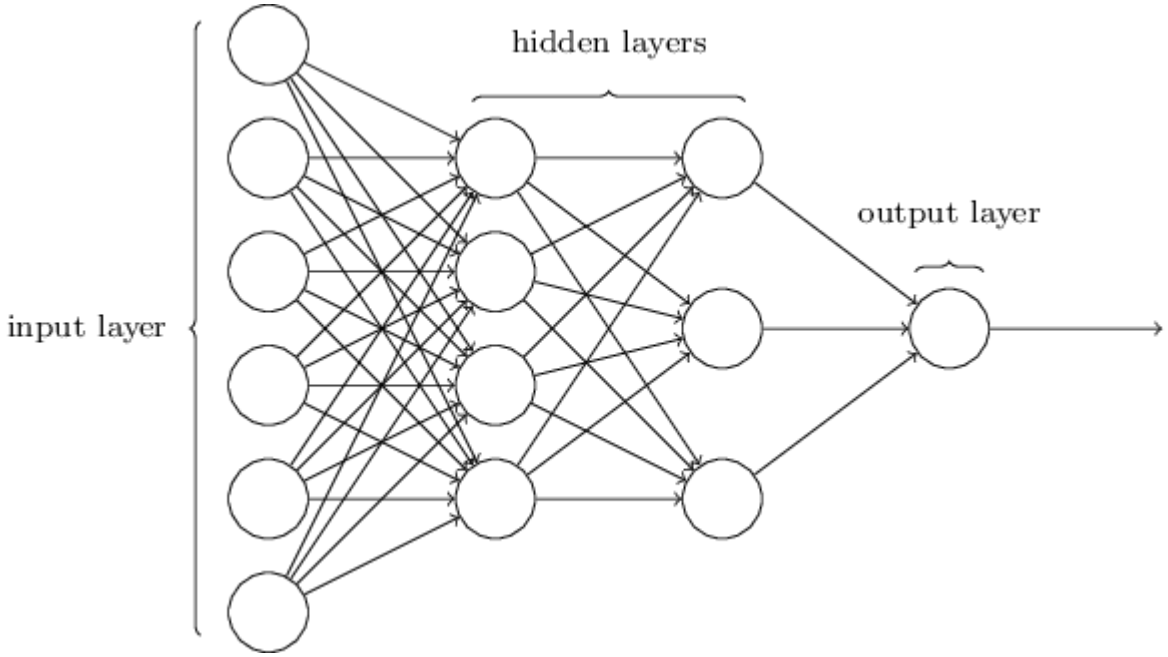


Four Layer Neural Network



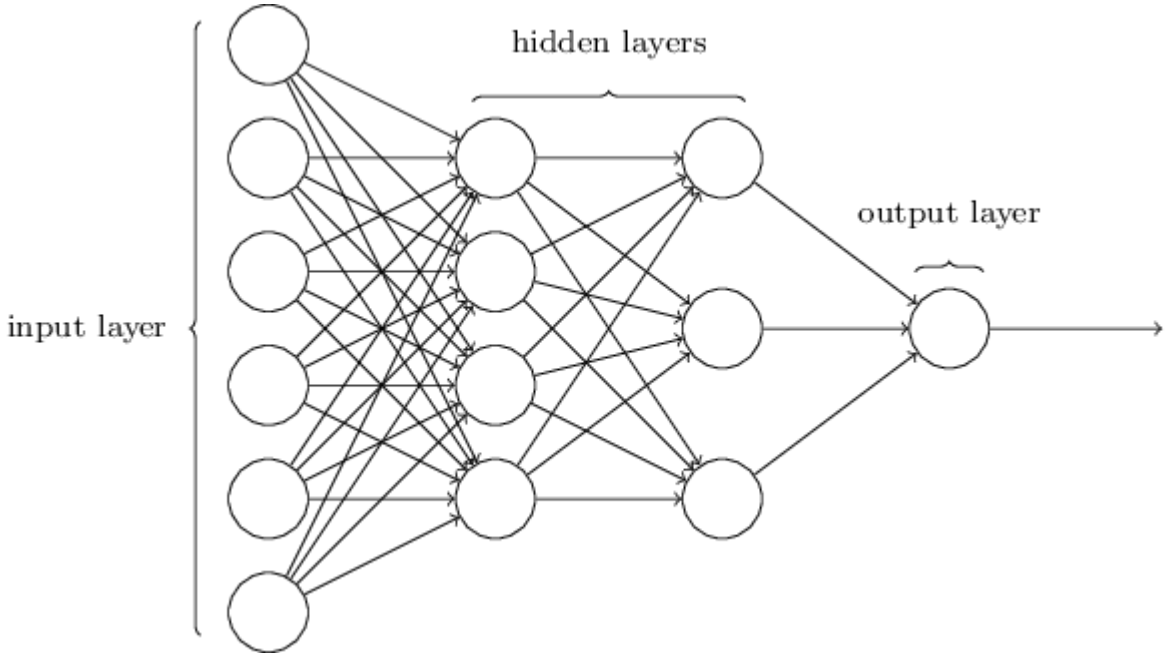
- This is a multi-layer perceptron (MLP)

How do we design our Neural Network Topology?



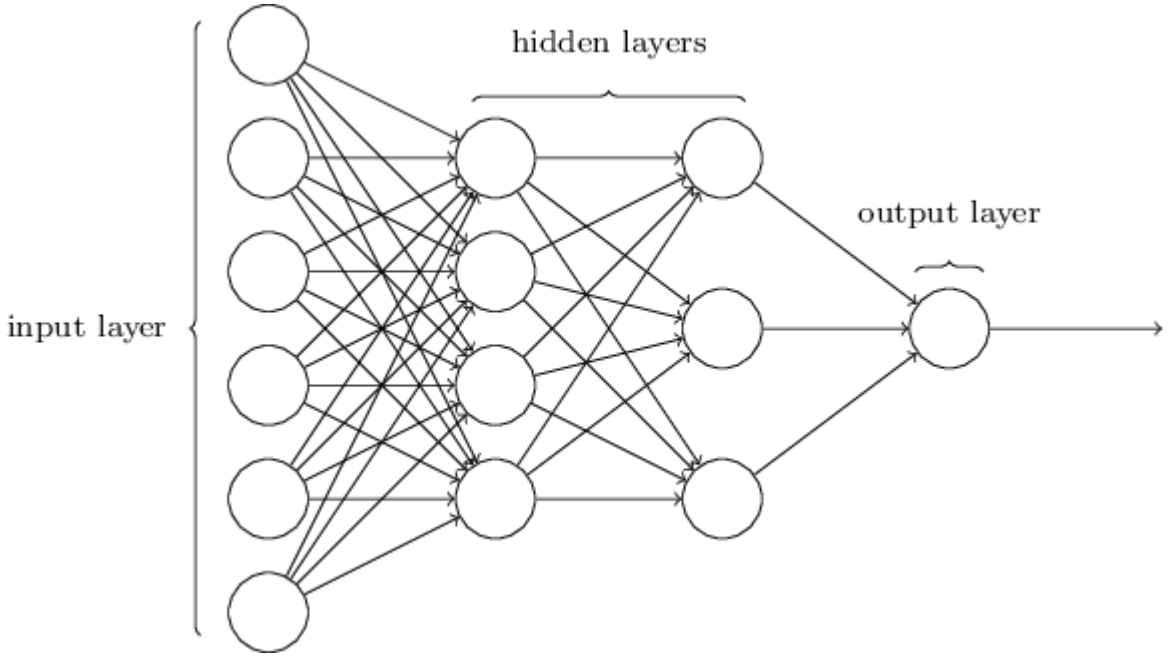
- Inputs
  - Depends on the number and type of inputs
- Outputs
  - If we are trying to do handwritten digit recognition, then we could use 10 outputs (0, 1, … , 9)

How do we design our Neural Network Topology?



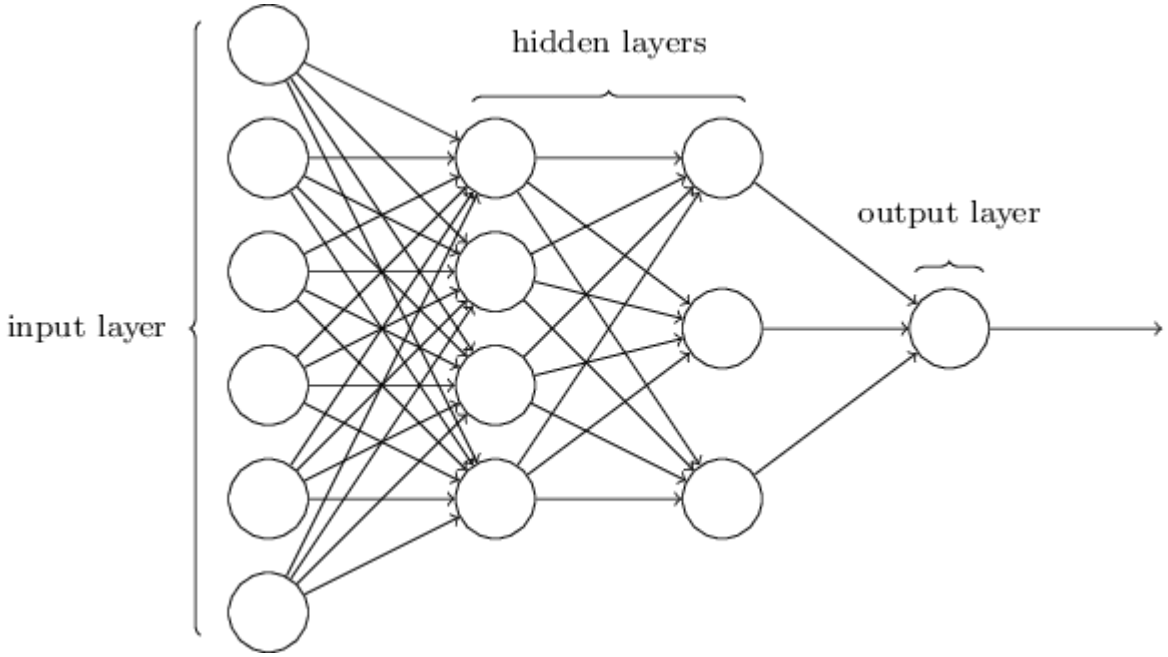
- Input: 64 x 64 pixel greyscale image
  - We could have 4096 input neurons, each with values between 0 and 1, which represent the greyscale intensity of the pixel
  - Here each input represents a pixel

How do we design our Neural Network Topology?



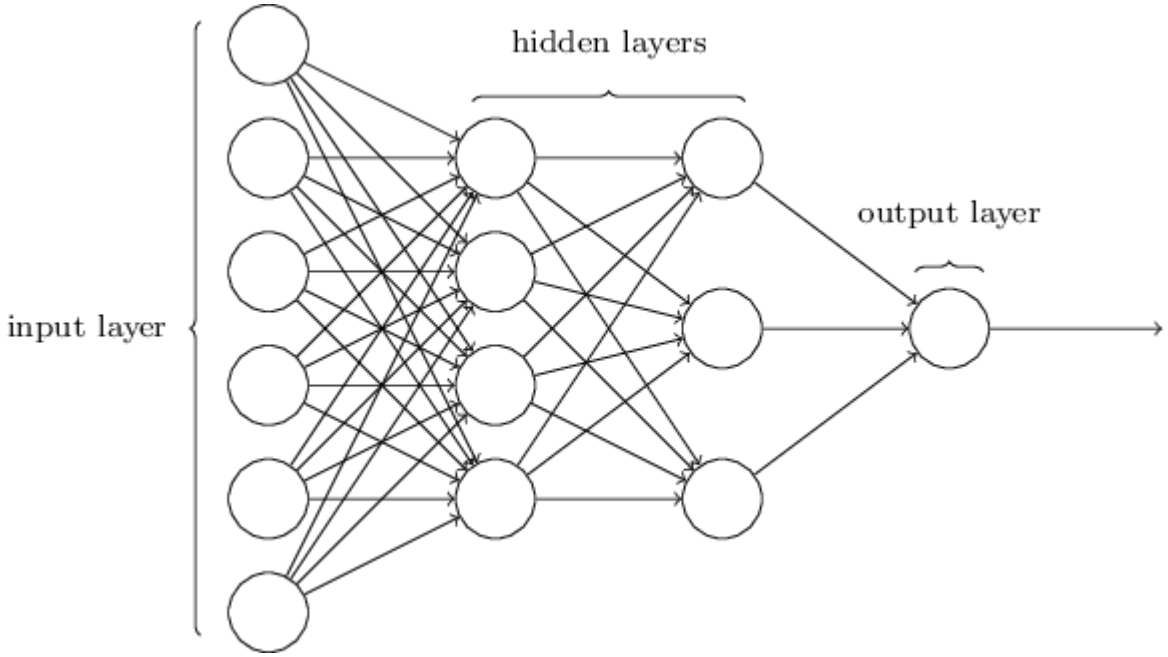
- Output: 0-9
  - With 10 output neurons, we could assign each one a digit, and then depending on the outputs, only 1 of those neurons should output a 'yes'
    - Recall 'yes' is defined as output being greater than some value (e.g. output>0.5)

How do we design our Neural Network Topology?



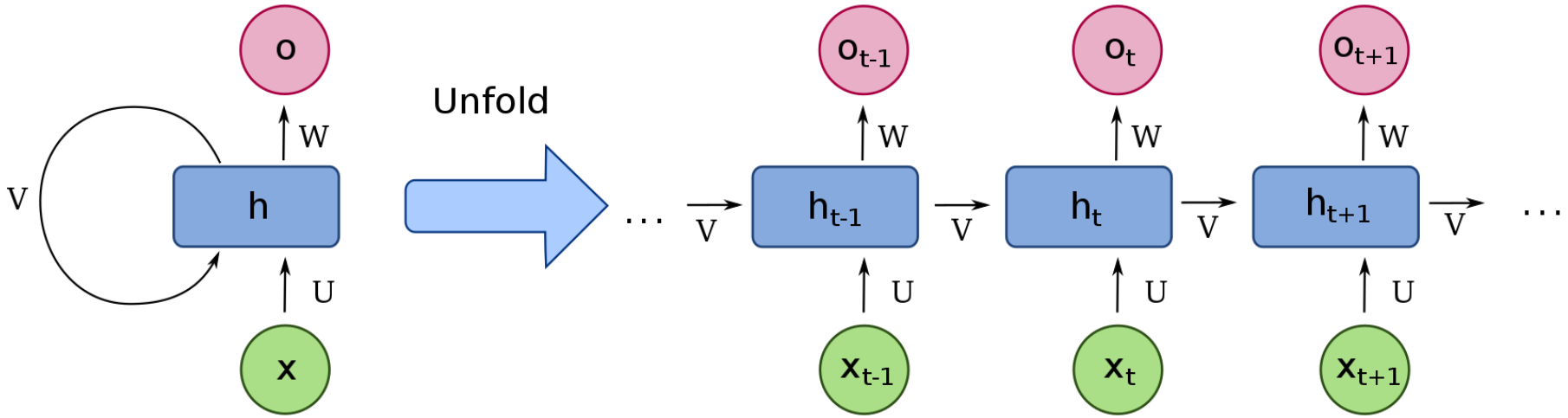
- Hidden layers?
  - This part is art, not science
  - Driven by heuristics depending on problem

How do we design our Neural Network Topology?



- Note:
  - These are feed-forward neural networks
  - All inputs move forward, never backward, during operation

Aside: Recurrent Neural Networks



- Note:
  - The output of a neuron actually is an input
  - There is some kind of time dependence, where the behavior can evolve over time
  - This is necessary as the output of a neuron cant be fed back in if it were to be evaluated intantaneously

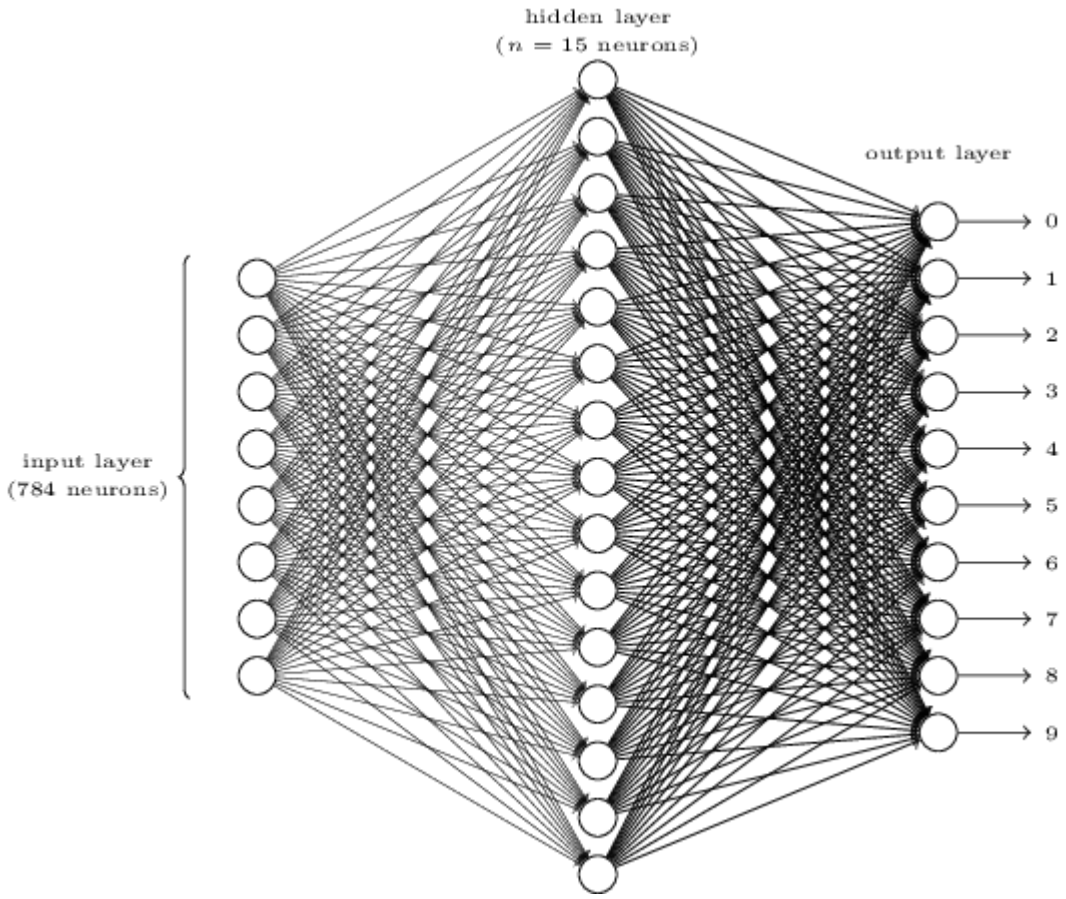
Getting back to handwriting

USC University of Southern California

- This number is 6 different images
  - Building an algorithm to automatically recognize and segment is a bit outside the scope of what we cover here and relatively do-able.
  - So we will focus on a single digit
  - Once we have that, segmenting can be done by taking a single image, and segmenting it into many different portions, and then using the handwritten digit classifier to assign scores to the segmentation and use the most highly scored version

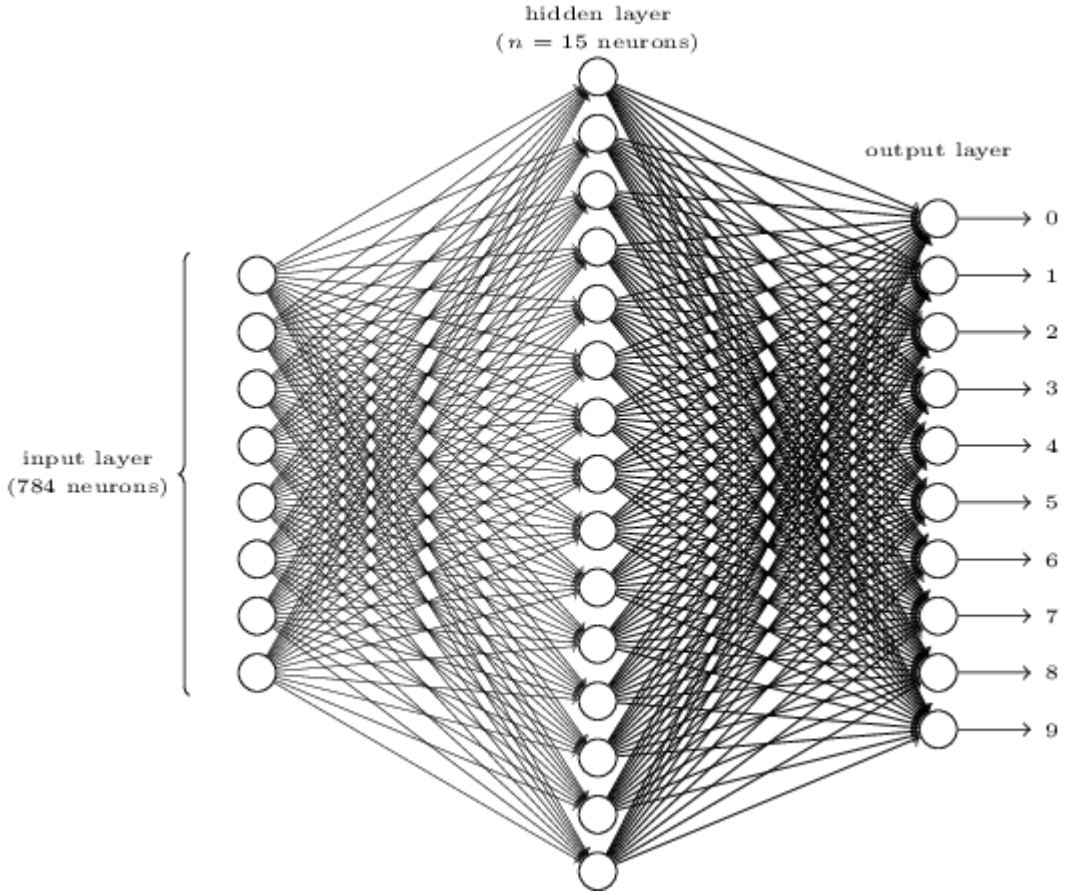
An ANN for Handwriting

USC
University of
Southern California



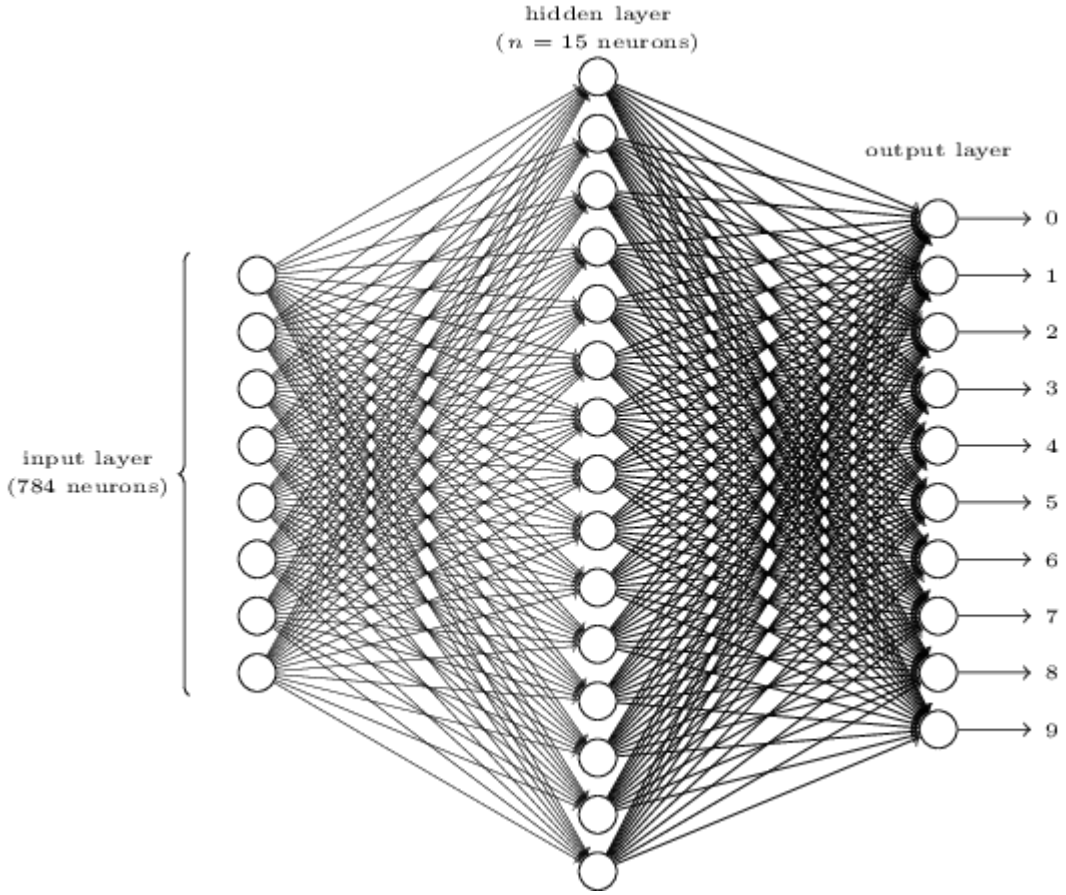
- What are the inputs?
  - 28 x 28 pixels images of scanned handwritten digits
  - Input layer has 784 neurons
  - Input pixels are greyscale with 0 representing white and 1 representing black
- One hidden layer
  - This only has 15 neurons, need to try different numbers of neurons to figure out how many hidden layers

An ANN for Handwriting



- Output layer
  - 10 neurons, each representing a digit
  - For a given digit, one neuron should be 1, while the others should be 0.
  - In reality, we find the neuron with the highest value, and say that is the one which activated

An ANN for Handwriting

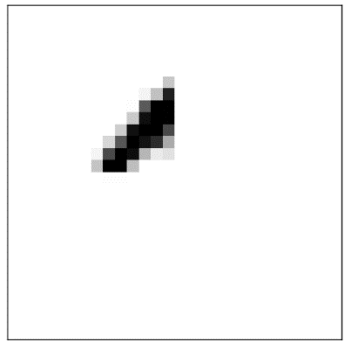


- Output layer
  - Why not use just 4 digits with binary encoded digits instead of 10 neurons?
  - If we do it, we will see that our neural network will be better with 10
  - Why?

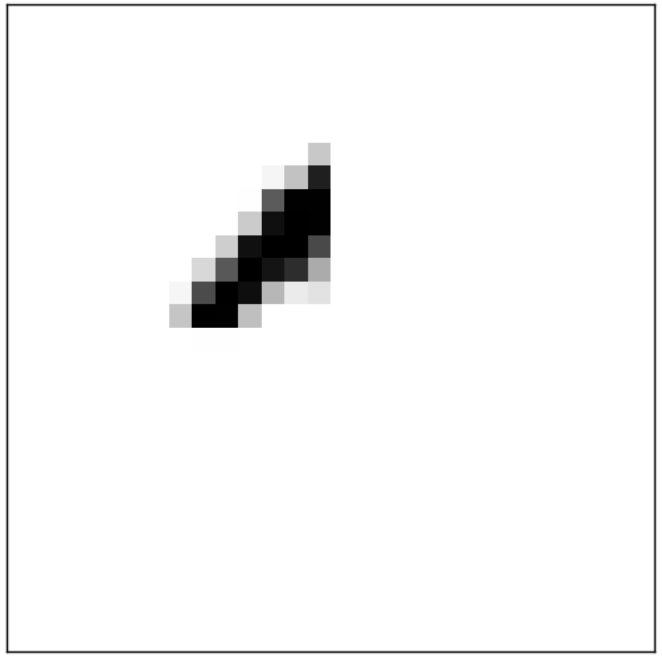
An ANN for Handwriting

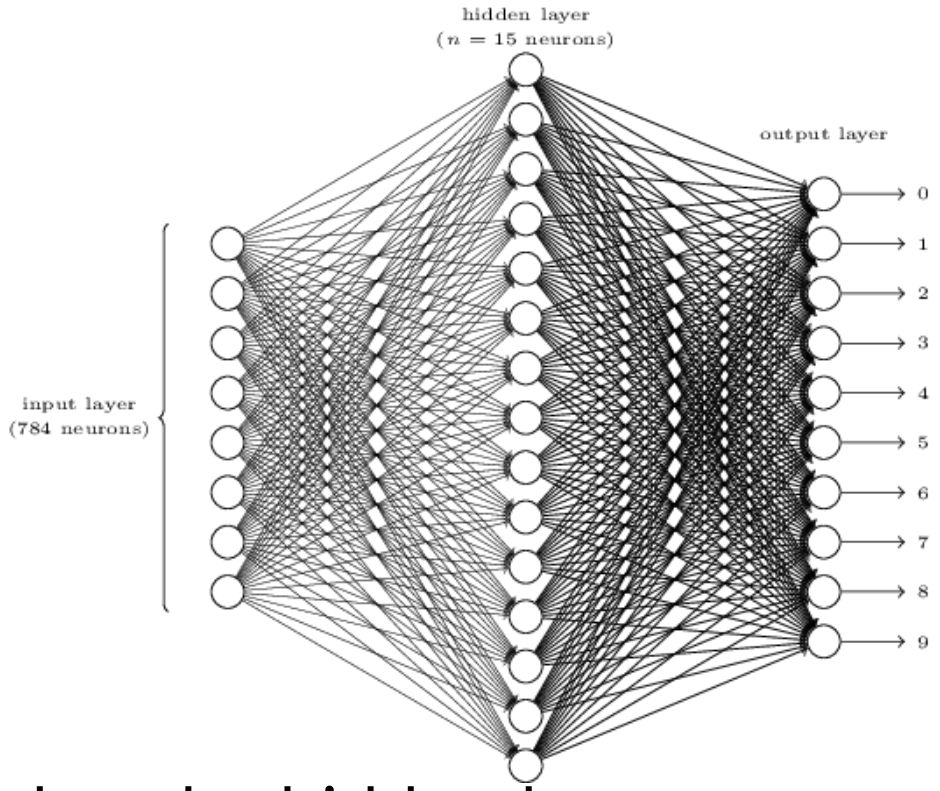
USC University of Southern California

- Why do 10 neurons work better than 4?
  - Need to consider how this network is working
    - The '0' output neuron is basically summing up all the inputs from the hidden layer and trying to decide whether the digit is a 0.
    - So then, what are the hidden layer neurons doing?
  - As a heuristic, let's assume that the first hidden neuron detects whether a certain pattern is present in the image:



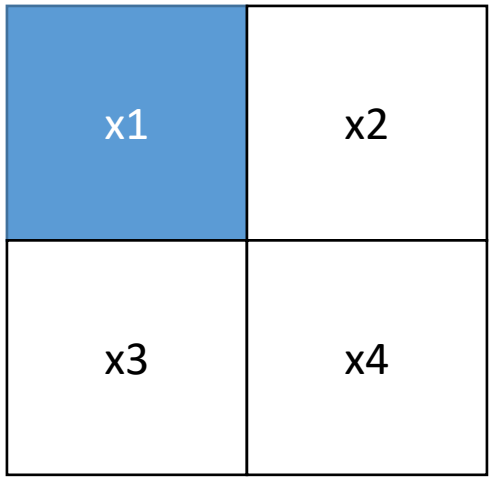
How does the hidden layer neuron find a specific image?

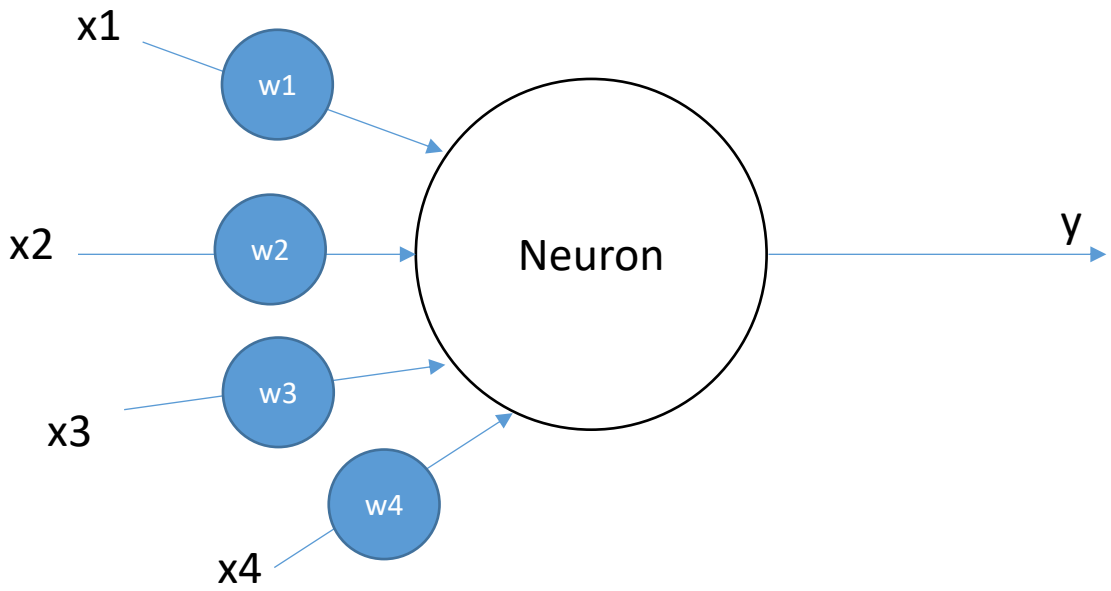




- Recall, every single input is connected to the hidden layer
  - So if I want the first hidden layer neuron to find that pattern, how would I bias it?

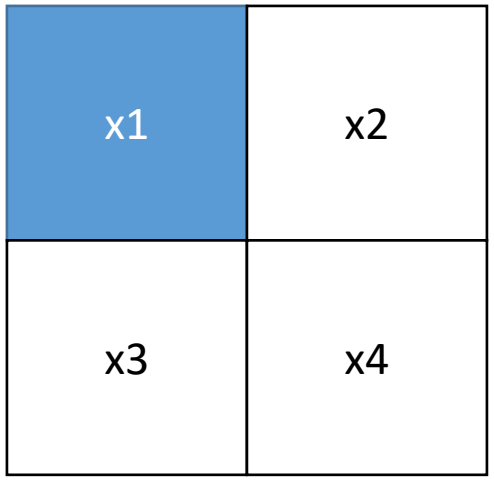
Let's do a simple example

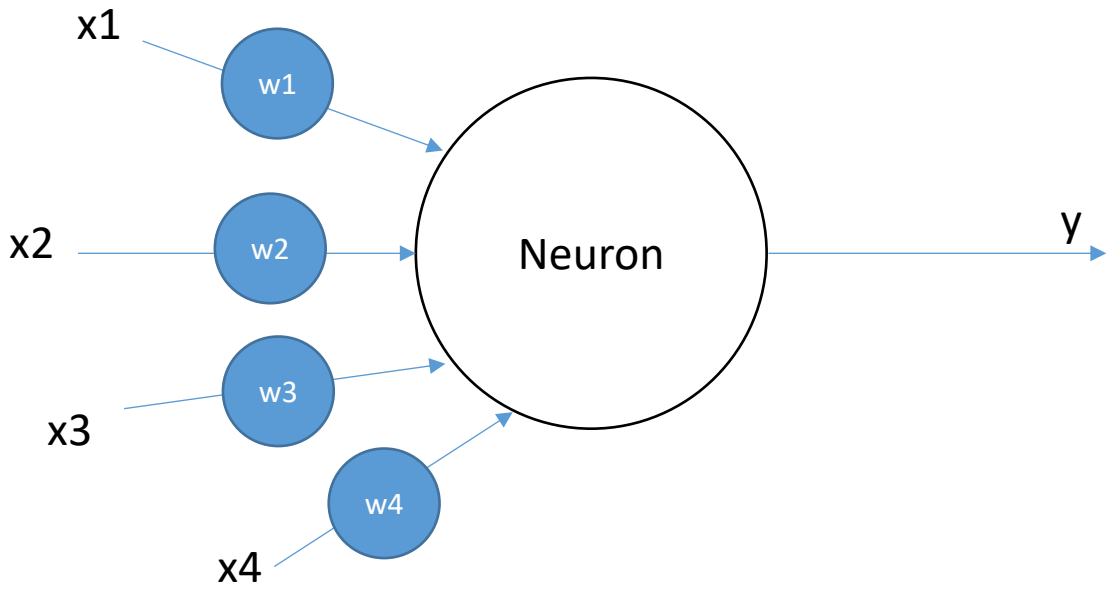




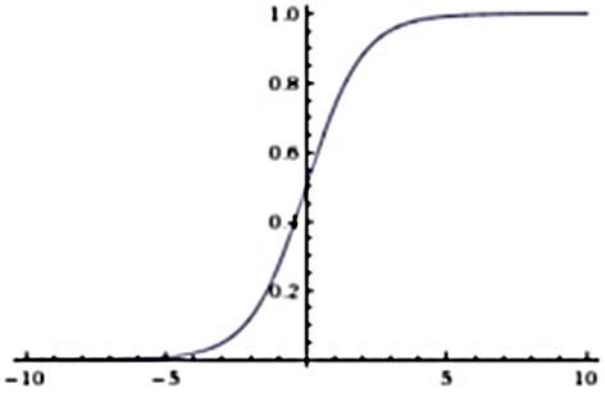
- I want to design a hidden layer neuron which tells me if the top left box is filled
  - So, we have four inputs
  - We want the output y to be high if x1 is dark and x2-4 are light

Let's do a simple example

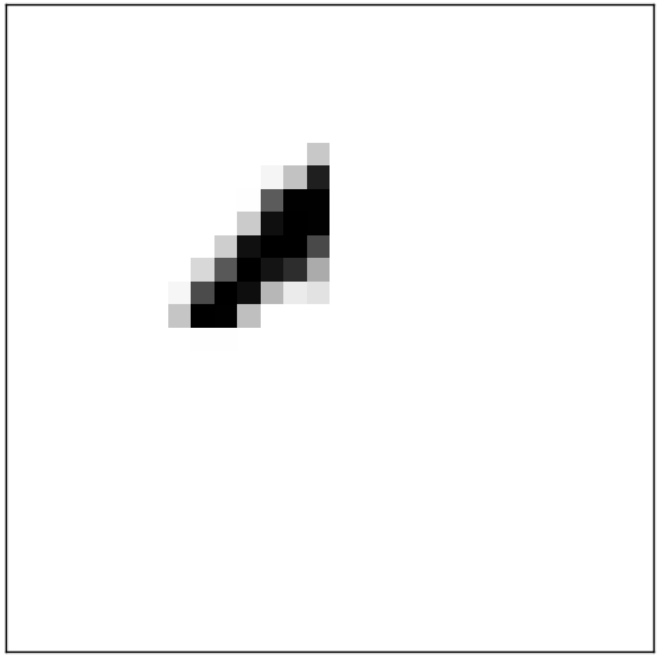


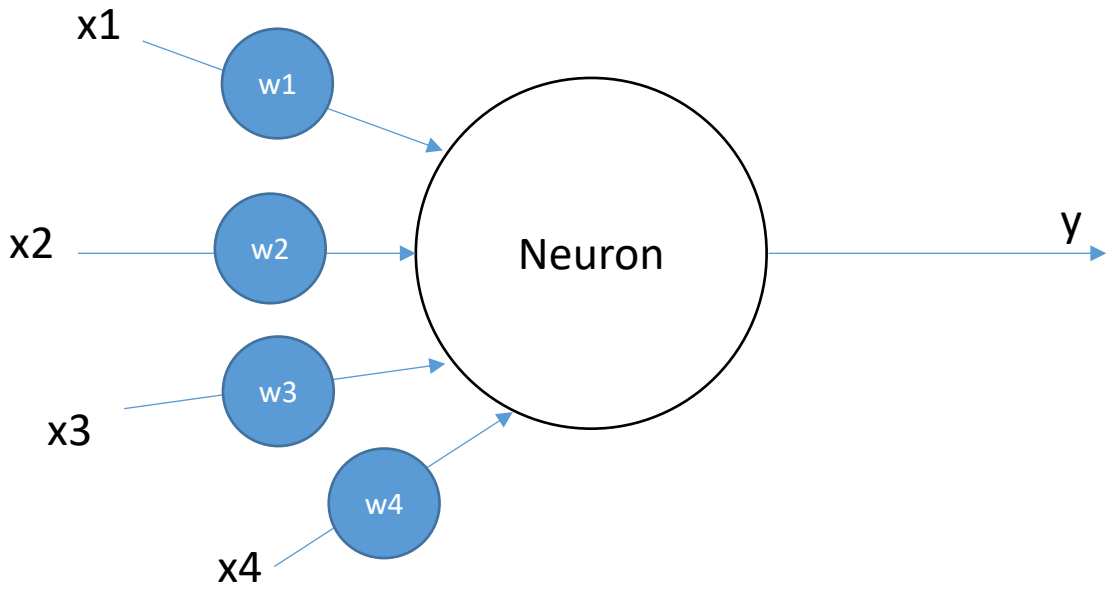


- Recall: $\sigma(x \cdot w+b)=\sigma(z)=\frac{1}{1+e^{-z}}$
  - So, want to weight w1 heavily, while dramatically lowering the weights of x2-x4
  - We want the output y to be high if x1 is dark and x2-4 are light

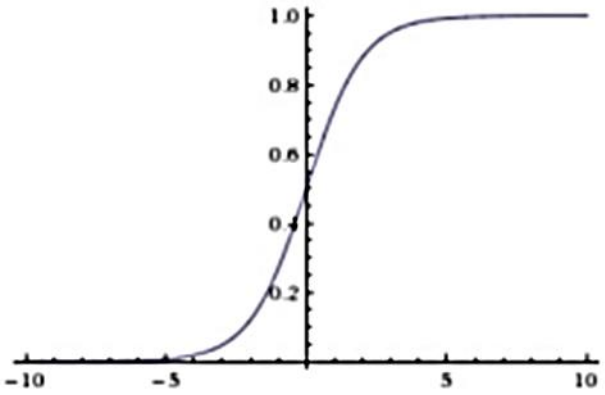


Back to the task at hand

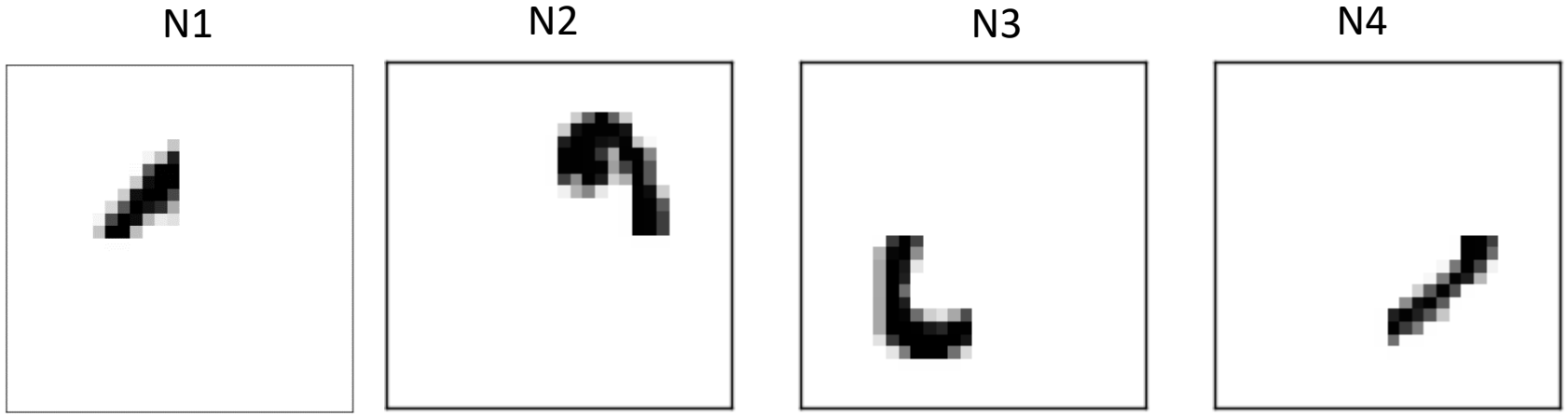


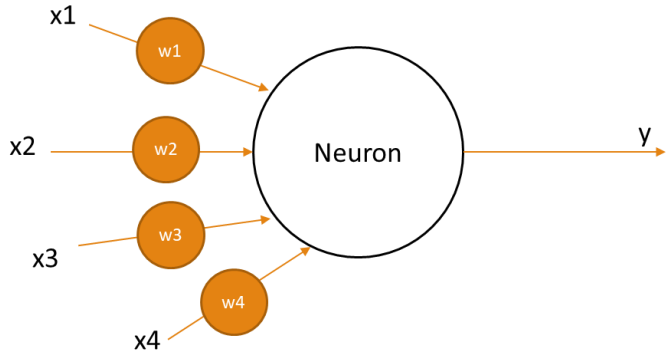


- So, want to weight the pixels which make up this pattern heavily, while reduce the weights of the other pixels

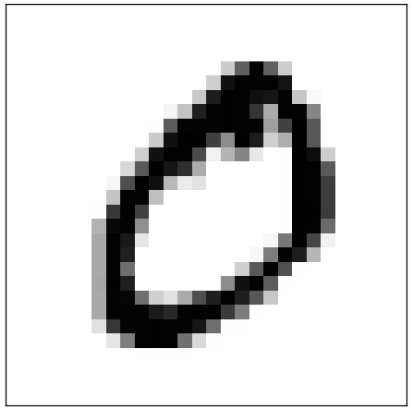


Other hidden layer neurons could then be detecting other images

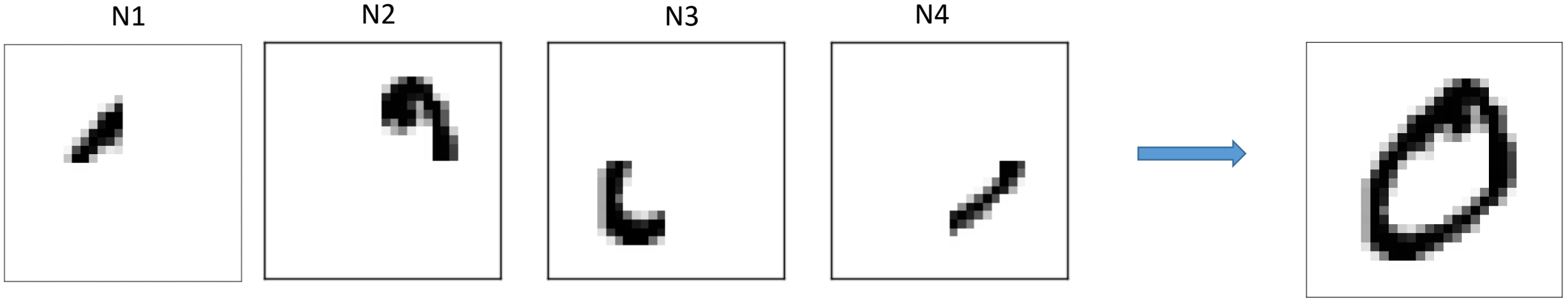




- Then neuron 2 and 3 and 4 could be detecting these other patterns
- Together these 4 images make up a zero

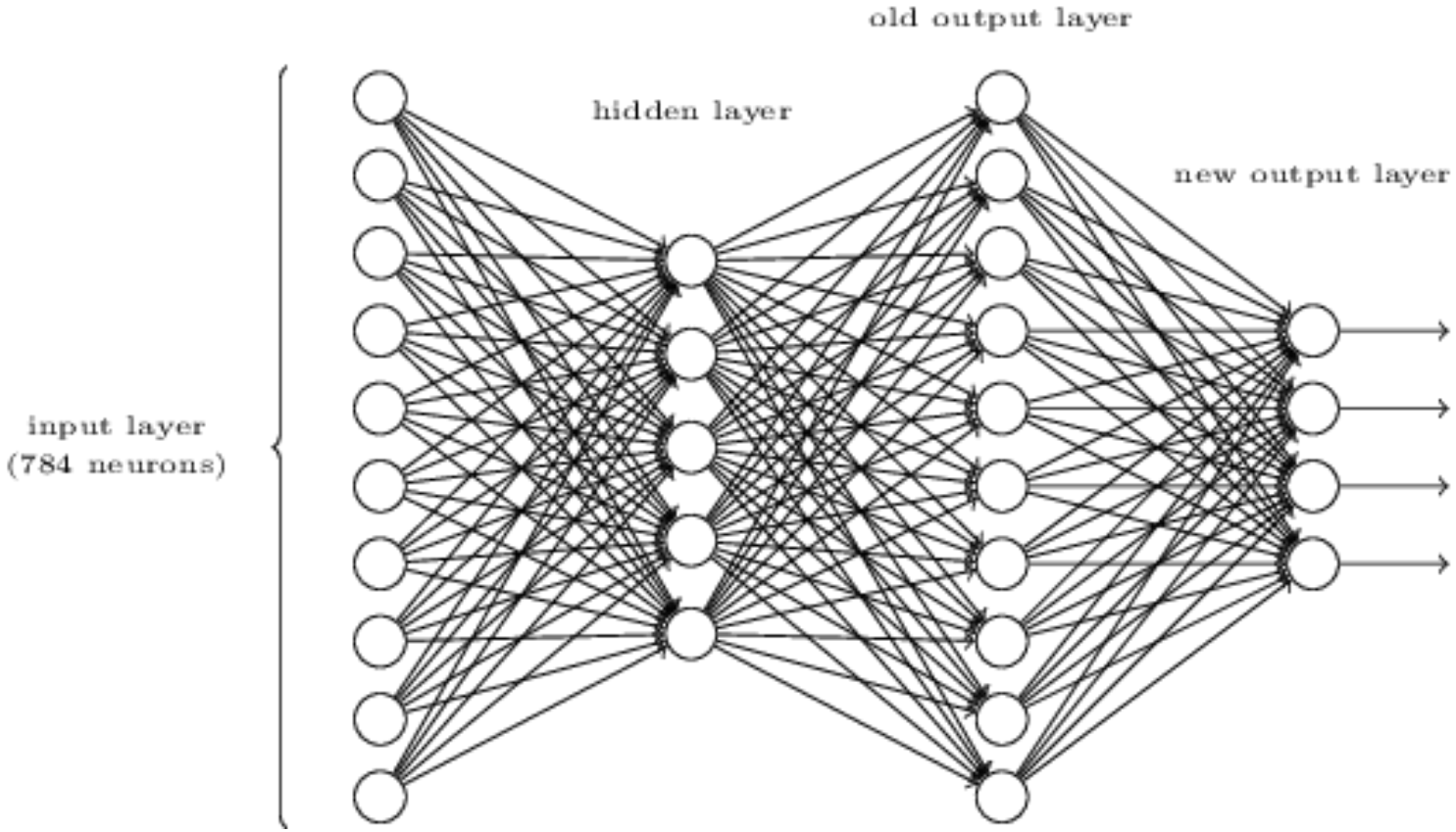


Other hidden layer neurons could then be detecting other images



- If we see that all 4 of these neurons are firing, then we can make the assumption that we are seeing a zero.
  - Obviously this is a toy example, don't get carried away trying to poke holes into it
- Now, we can possibly see why it's better to have 10 outputs over 4
- With the above example, the 0 output neuron could just put the most weight on the outputs from the hidden neurons N1 through N4, and weaken the others
- But, if we had only 4 output neurons, our neural network would have to essentially do 2 calculations
  - First, we would need to figure out what the number was
  - Then we would need to transform that into binary

So what if we want the bitwise representation



- Why would this topology work?
  - Well, when we add in a new output layer, we now have a new layer of abstraction.
    - First, the hidden layer identifies certain patterns/features
    - Second, the old output layer transforms those patterns into the basis set of digits
    - Finally, the new output layer would transform the basis set of digits into binary

Learning how Artificial Neural Networks Learn with your Biological Neural network

- We will use the MNIST Data set
  - http://yann.lecun.com/exdb/mnist/
  - Two parts
    - 60,000 Training images (28 x 28 pixels, greyscale)
    - They were scanned samples from >200 people, ½ government employees, ½ high school students
    - 10,000 Test images
    - These test data were collected from different people

How to train your artificial neural network

- Use x as the annotation for the input data vector, a 784 long string of digits between 0 and 1 representing the greyscale values.
- The output is then a vector y=y(x) with 10 digits.
- As an example:
  - If input image x represents a 1
  - Output vector y(x) = (0,1,0,0,0,0,0,0,0,0)
  - What we need to do is develop an algorithm that let's us find the weights and biases so output is as close to y(x) given above as possible.

How to train your artificial neural network

USC University of Southern California

- We need to figure out what we want to optimize
  - Obviously, the number of images correctly recognized, duh….
- But what if we used that as the objective function?
  - C(w,b)=Number of correctly classified images
  - Could we easily optimize with this?
  - Not really, this is not a smooth function of w (all the weights in the NN) or b (all the biases in the NN)

How to train your artificial neural network

- So what if we defined another ‘cost function’?
  - C(w,b)=Number of correctly classified images
  - Could we easily optimize with this?
  - Not really, this is not a smooth function of w (all the weights in the NN) or b (all the biases in the NN)

Cost Function

$$C(w,b) = \frac{1}{2n}\sum_{x}\|y(x)-a\|^2$$

- w represents all the weights in the network
- b represents all the biases in the network
- a is the output vector when input vector x is input
- y(x) is the desired output vector when x is the input vector
- || y-a|| is just the magnitude of the difference between what we want and what we have.
- Essentially, we take the square of the error
- This is called the quadratic cost function, or mean squared error (MSE)

Cost Function

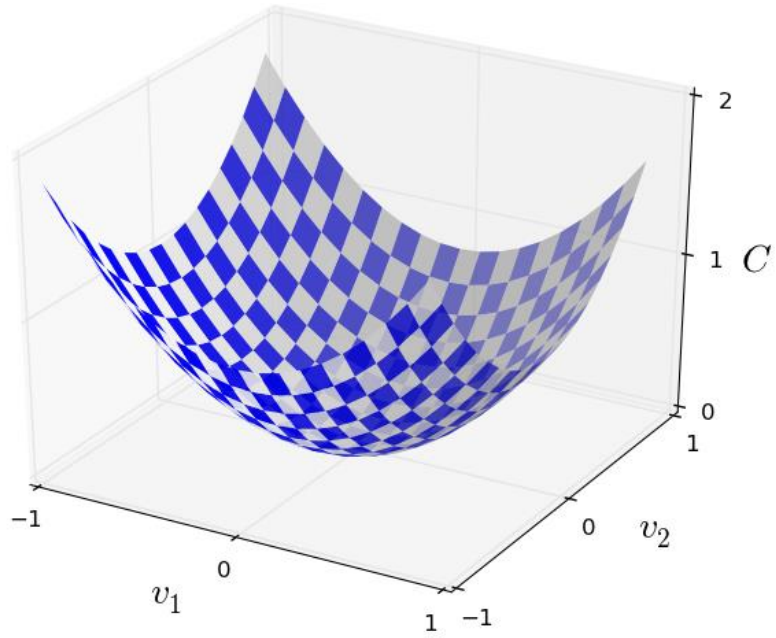
$$C(w,b) = \frac{1}{2n}\sum_{x}\|y(x)-a\|^2$$

- C is going to be positive or zero
- If C is small, then the actual network outputs are close to the inputs
- If C is large, our NN is outputting garbage
- We need an algorithm to minimize C by tuning w, b
- The chosen algorithm is gradient descent

Gradient Descent

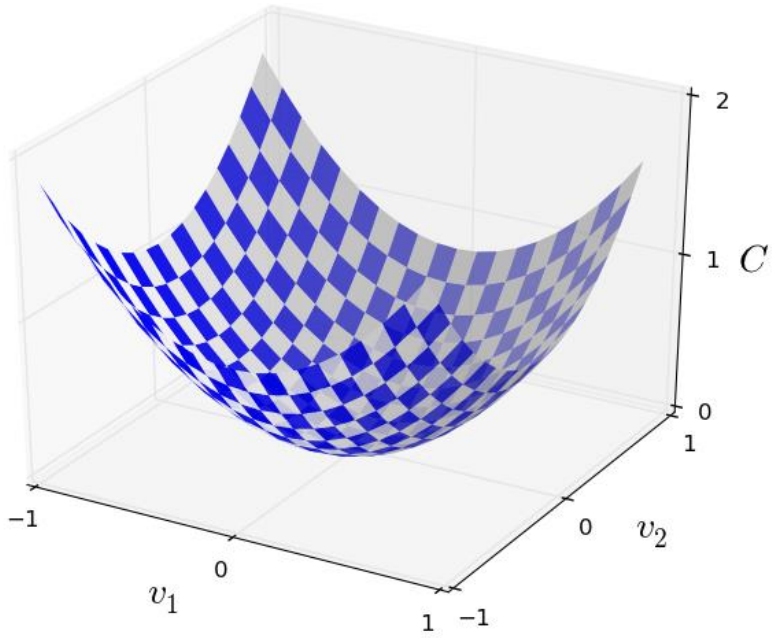
USC University of Southern California

- Let's assume we want to minimize function C(v), where v is the vector of inputs.
  - This could be a single variable, with v being a number, or a million variables, with v being a vector with a length of 1,000,000.
  - For ease of visualization, let's just say it has 2 variables

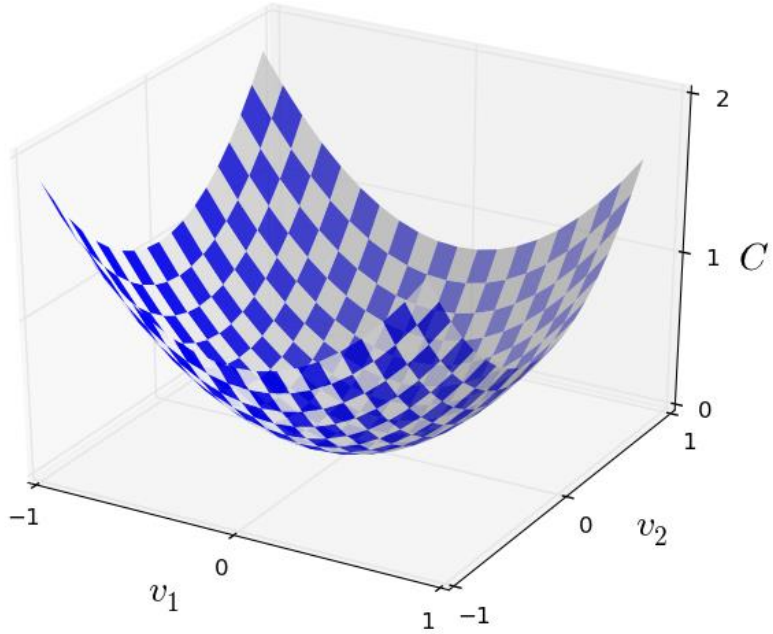


Minimization

- This seems easy enough…
  - I can just see it, what's the big deal?
  - Not so easy with 10^6 variables
  - Calculus also not tractable with huge dimensionalities
  - However we can still use these principles
    - Imagine that we want to start at some point here ar roll down to the bottom of the valley



Minimization



- Let's use some math
  - Assume ball starts at some location (v_1, v_2)
  - Then the ball rolls Δv_1 and Δv_2
  - We know:

$$\Delta C \sim \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

  - As long as change is small
  - We want to make ΔC negative
  - Let's define Δv as the vector of changes $\Delta v=(\Delta v_1, \Delta v_2,\ldots)^T$
  - The T transpose operator is just there to say it's a column vector

Minimization

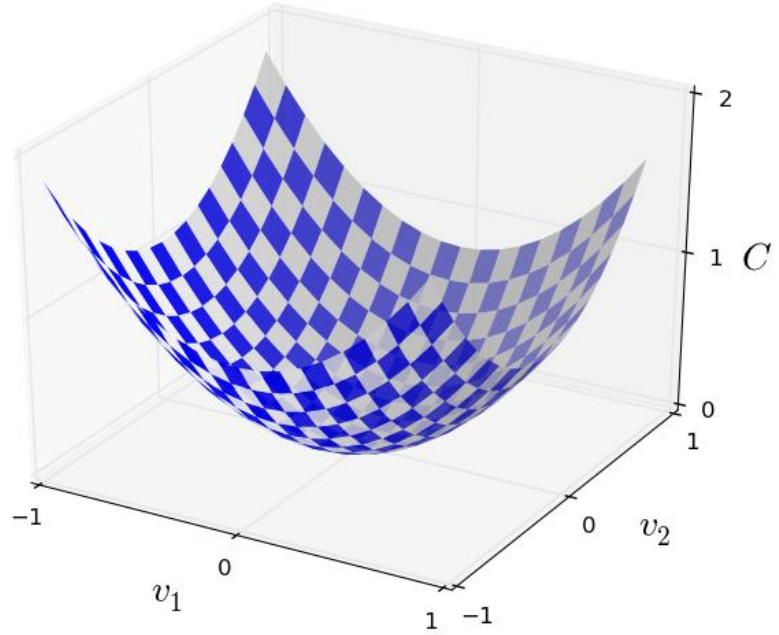
$$\Delta C \sim \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- Also define a gradient vector:

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

- Now this let's us write:

$$\Delta C \sim \nabla C \cdot \Delta v$$



Minimization

- Now, how do we choose Δv to make ΔC negative?

$$\Delta v = -\gamma \nabla C$$

  - Where g is a small, positive number
  - If we use the above approach, then:

$$\Delta C \sim -\gamma \nabla C \cdot \nabla C = -\gamma \|\nabla C\|^2$$

  - But remember, $\|\nabla C\|^2 > 0$, so multiplying by a negative number means this will always be negative…
  - γ is called the "learning rate"
  - So we will use $\Delta v = -\gamma \nabla C$ as the rule for how we change our inputs

Minimization

$$v \rightarrow v' = v - \gamma \nabla C$$

- Using the above update rule, we can now keep moving along some path, decreasing the value of C until we reach a minimum
  - This requires that we repeatedly compute the gradient, and then use that to define motion
- But, in order for it to work, the learning rate needs to be small enough for the linear approximation to work

Minimization for more than 2 variables

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \ldots \frac{\partial C}{\partial v_n}\right)^T$$

- $\Delta v = (\Delta v_1, \Delta v_2, \ldots, \Delta v_n)^T d$
- We use the same approach, and we are guaranteed that we will be moving down the hill, so to speak
  - The update rule essentially defines gradient descent
  - We only need the first derivative
  - Some variations on gradient descent use second derivatives, however, that blows up in complexity
  - If you have 10^6 inputs, then you might need 10^{12} second derivatives…

Using Gradient Descent in a neural network

USC University of Southern California

- In our neural network, we have some number of weights (w) and some number of biases (b)

$$w_k \rightarrow w_k' = w_k - \gamma \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \gamma \frac{\partial C}{\partial b_l}$$

Using Gradient Descent in a neural network

USC University of Southern California

- How would we actually do this?
  - Remember our cost function

$$C(w,b) = \frac{1}{2n}\sum_{x} \|y(x) - a\|^2$$

  - It's a function of all the inputs over the training examples set
  - This means to compute the gradient, we need to first compute gradients for all inputs, and then average them
    - We've got a ton of inputs…this could take a long time, slowing down learning.
  - So what can we do?
    - Train with fewer inputs

Stochastic Gradient Descent

- Stochastic gradient descent says that the
  - Let's estimate the gradient by only averaging over a small set of randomly chosen inputs
  - We choose some subset m of random training inputs
    - Label them $X_1,\ldots,X_m$ and call them the mini-batch
  - We expect that as long as m is not too small, the average value of the gradient of the mini batch will be roughly the same as the average value of the whole input set

$$\frac{\sum_{j=1}^{m} \nabla C x_j}{m} \sim \frac{\sum_x \nabla C_x}{n} = \nabla C$$

Stochastic Gradient Descent

$$\frac{\sum_{j=1}^{m} \nabla C x_j}{m} \sim \frac{\sum_x \nabla C_x}{n} = \nabla C$$

- If we rearrange, we get

$$\nabla C \sim \frac{1}{m} \sum_{j=1}^{m} \nabla C x_j$$

Using Stochastic Gradient Descent in our Networks

$$\frac{\sum_{j=1}^{m} \nabla C x_j}{m} \sim \frac{\sum_x \nabla C_x}{n} = \nabla C$$

- If we rearrange, we get

$$\nabla C \sim \frac{1}{m} \sum_{j=1}^{m} \nabla C x_j$$

- So when we train, we can pick out from the mini batch

$$w_k \rightarrow w_k' = w_k - \gamma \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \gamma \frac{\partial C}{\partial b_l}$$