Machine Learning from 100000 feet

For a great intuitive look at this with beautiful animations, see https://www.youtube.com/watch?v=aircAruvnKk

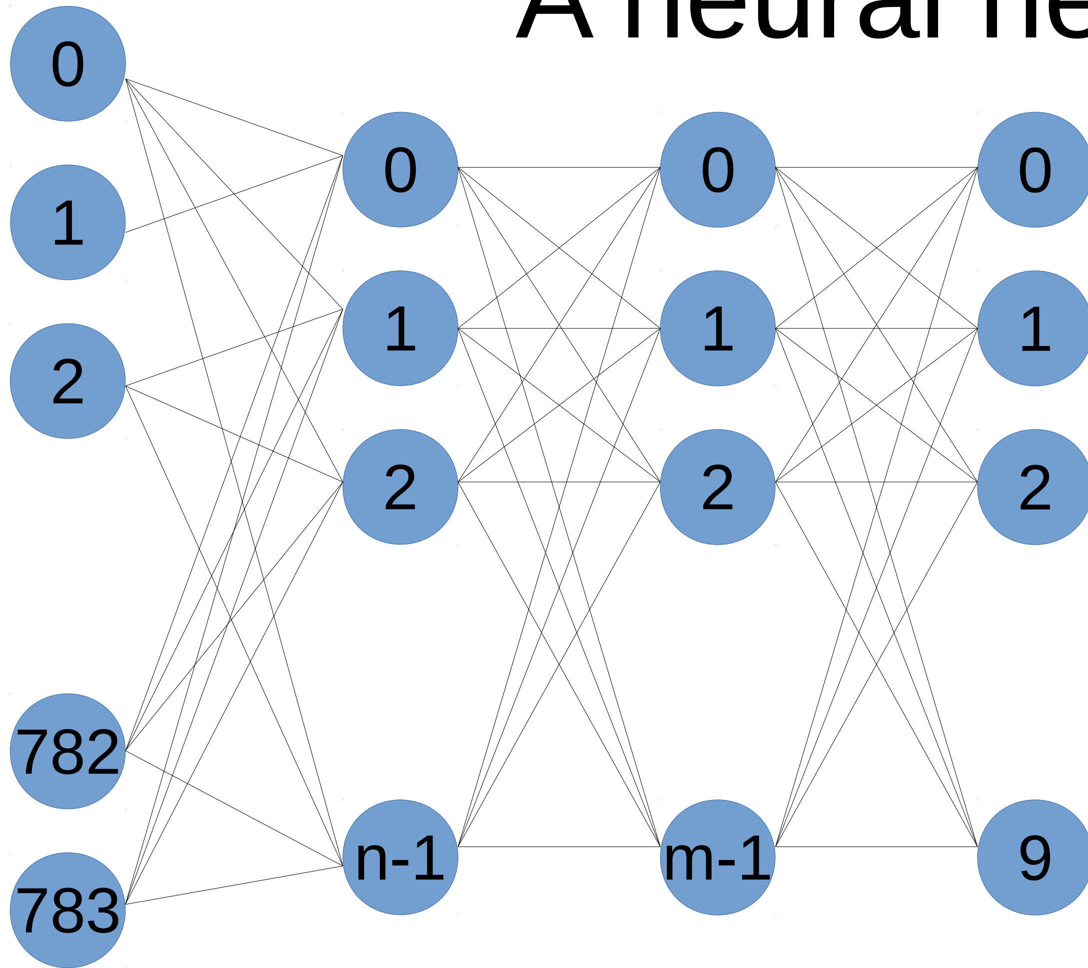
What is a neural network

- It’s not AI
- It’s basically a connected graph organized in layers
- By tuning the neural network it will match data to buckets established by training
- They are opaque

The problem we're going to show

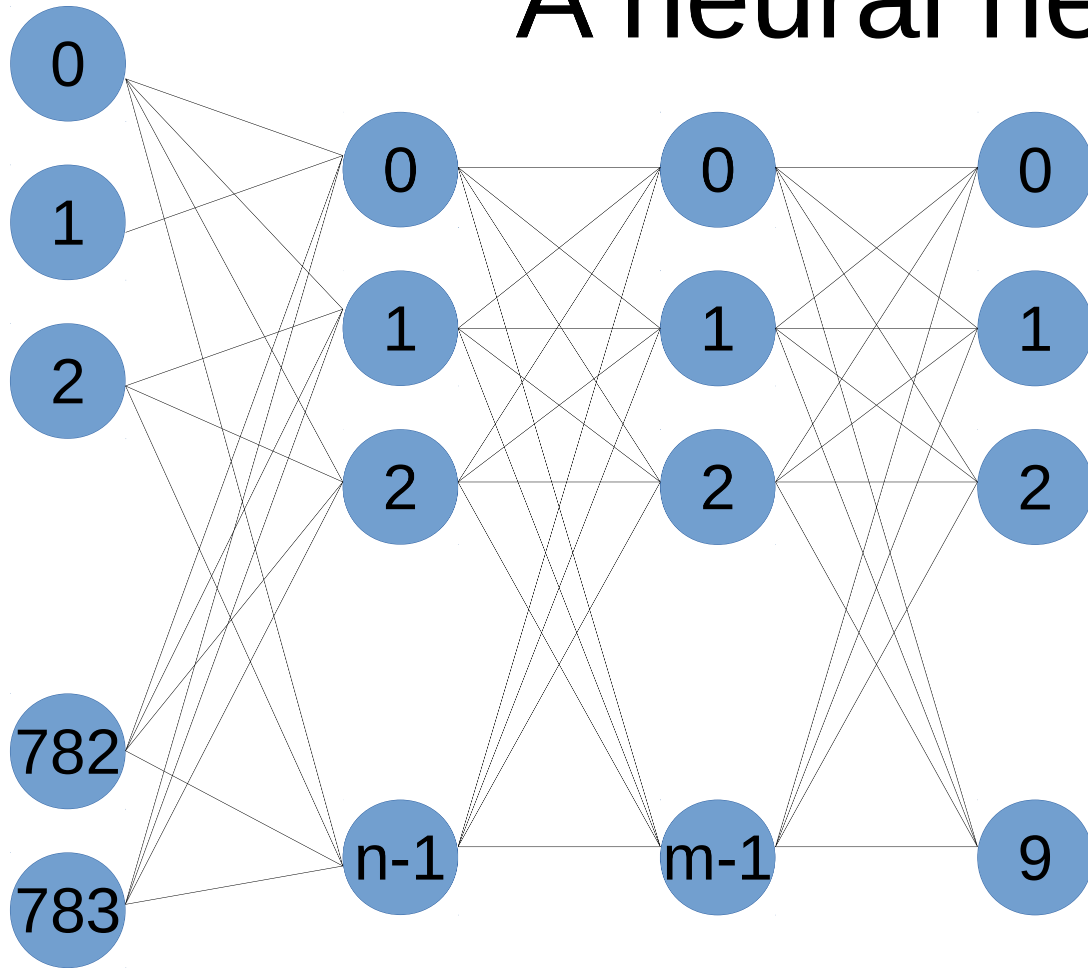
- MNIST is the "hello world" of machine learning
- The idea is to match take handwritten digits, represent them as pixels, and automatically recognize them.
- Each number is represented by a 28 x 28 array of pixels

A neural network



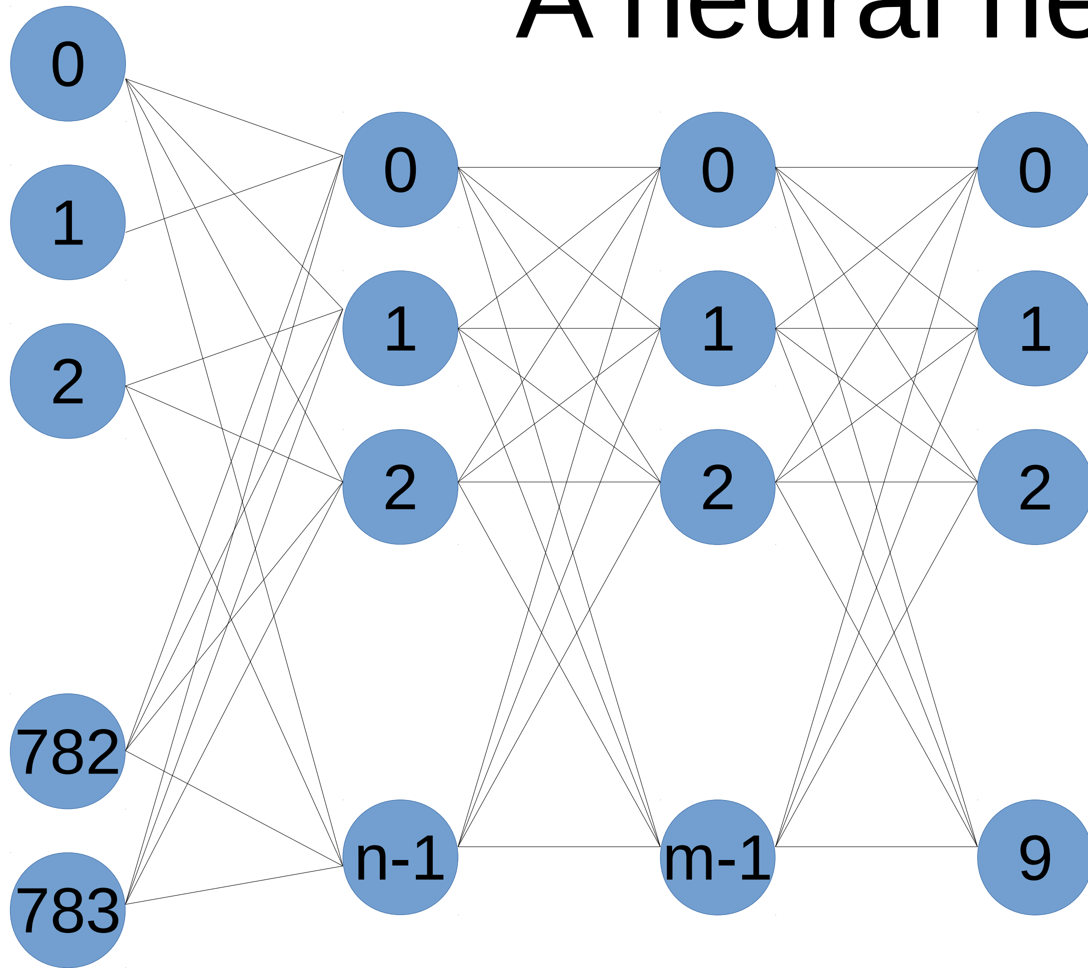
- 784 inputs correspond to the 784 (28 x 28) pixels in each image.
- 10 outputs correspond to the digits 0 .. 9

A neural network



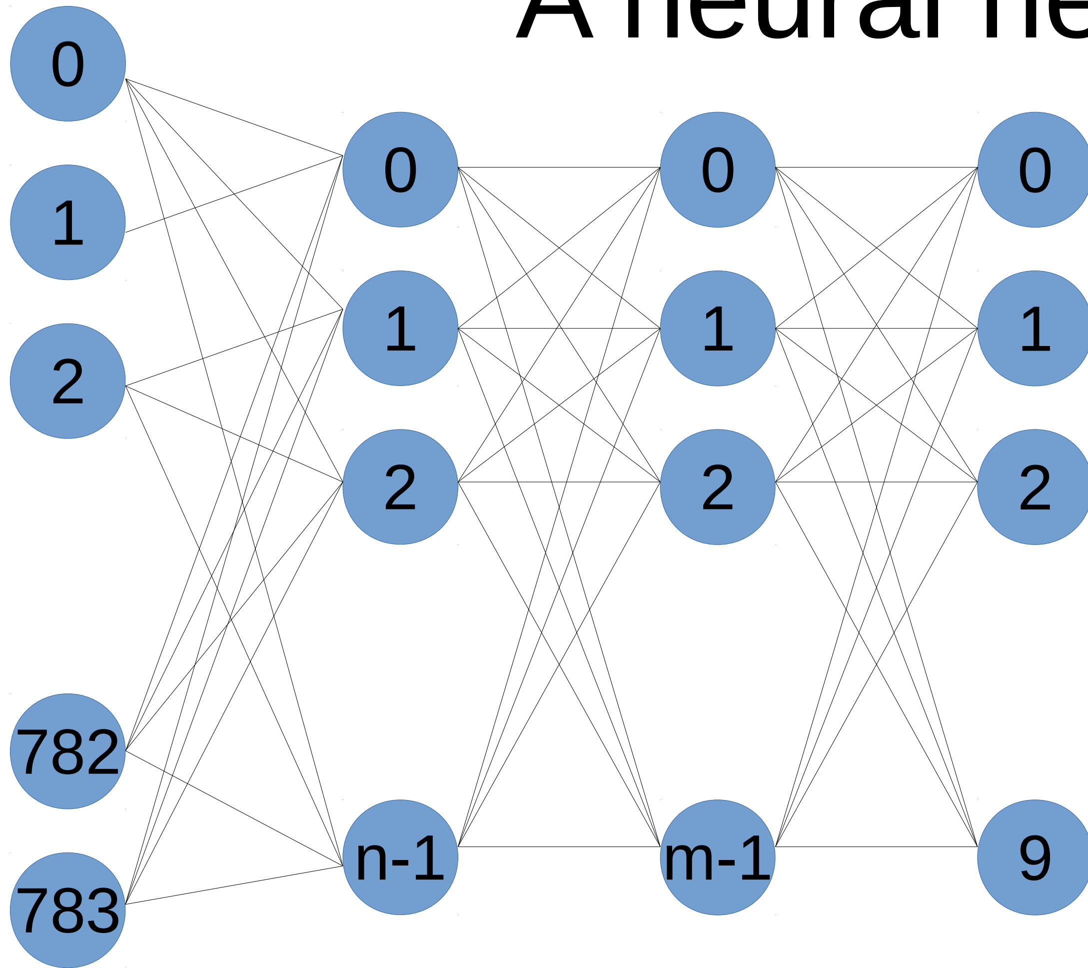
- Nodes or *neurons* values are *activations*
- Nodes are connected to other nodes that they can stimulate
- Analogous to brains and neurons

A neural network



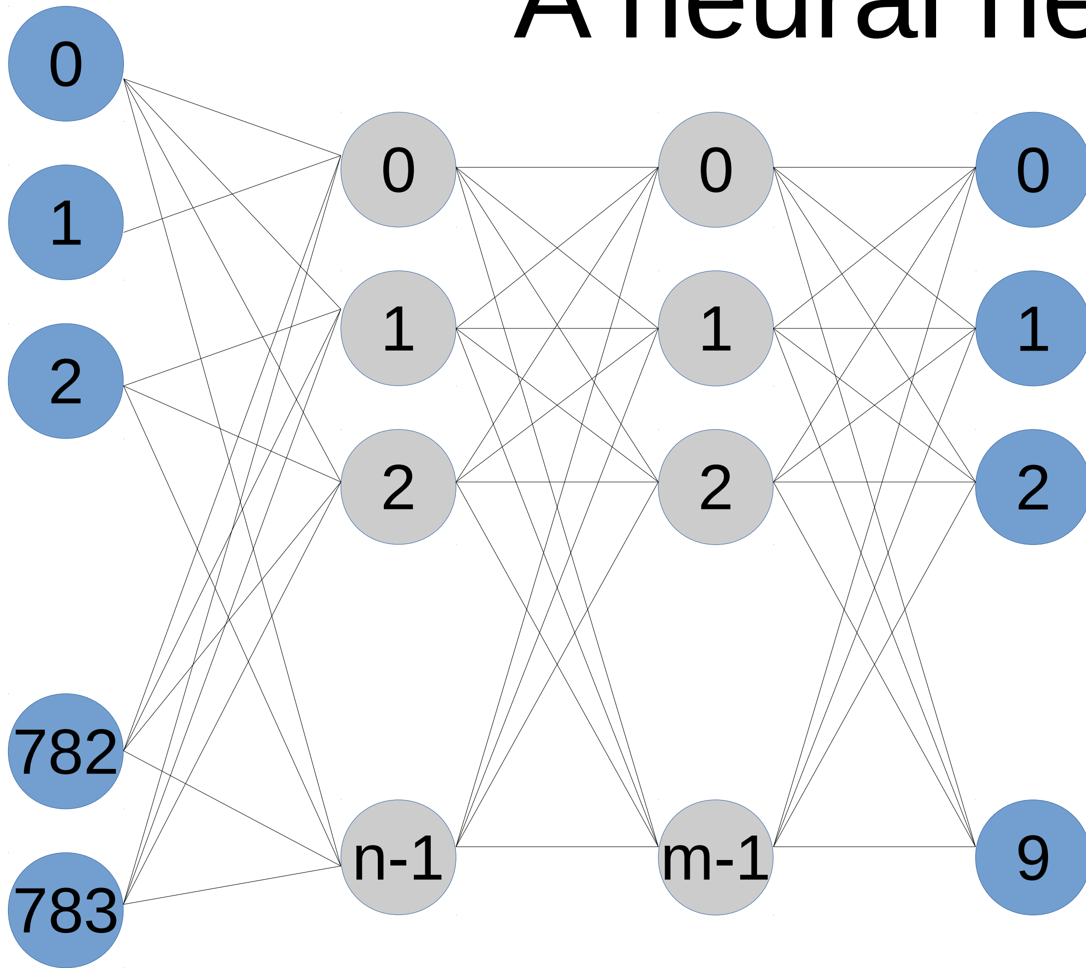
- Values of the input nodes are the value of the corresponding pixel
- Value of the output node is a numeric representation of the likelihood this is the number whose pixels are inputs.

A neural network



- Input values and values on nodes are *normalized* to be between 0 and 1
- Number of layers and number of neurons in a layer affect the performance of the Neural network.

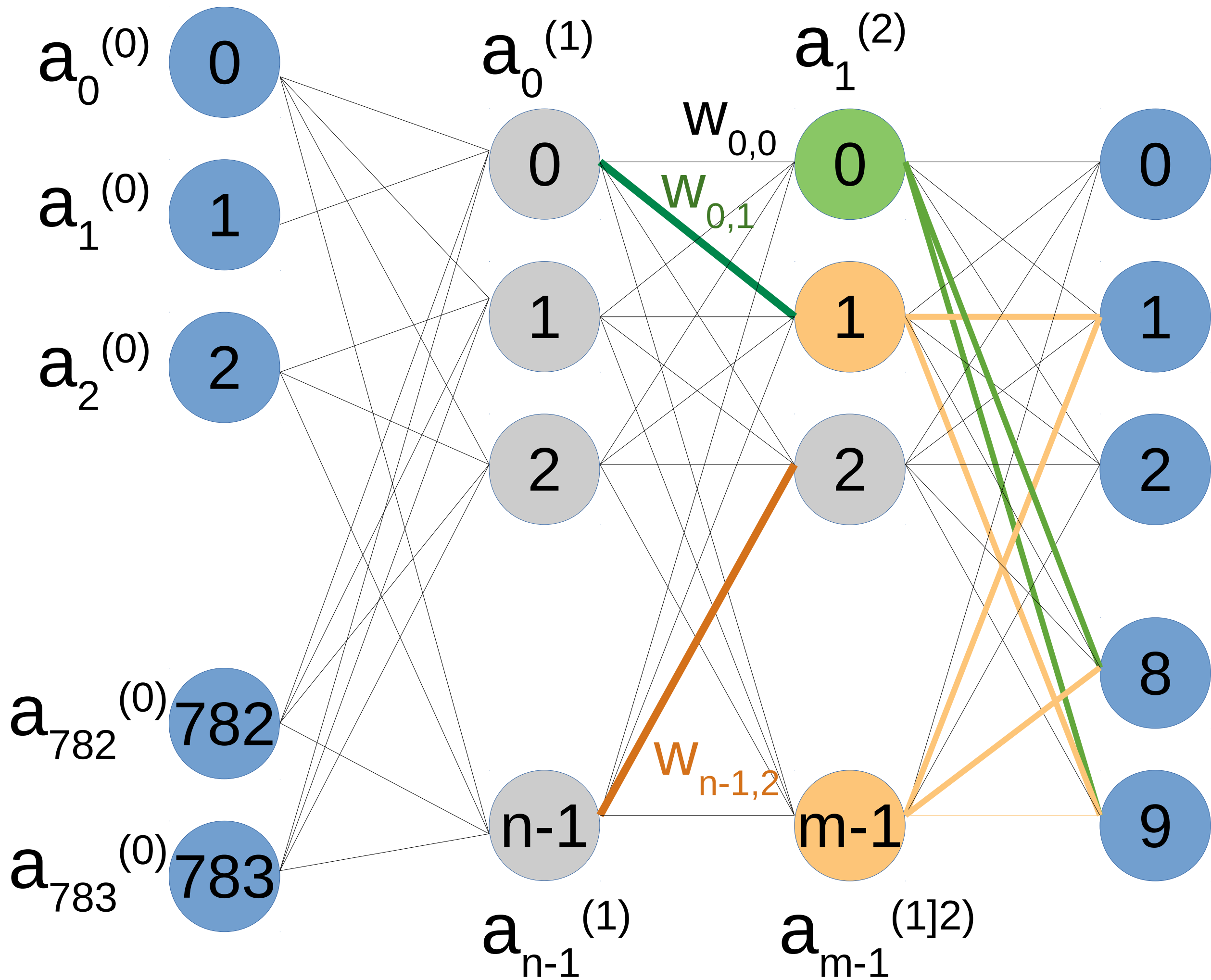
A neural network



- This is a *multilayer percepton*
- The gray nodes are *hidden layers*

Parameters of the neural network

- Some parameters of the neural network are
  - The number of layers,
  - The number of nodes,
  - How values are normalized to be between 0 and 1
- Selecting parameters is more art than science
- Initially just play with it.
- To small of a network leads to poor accuracy
- To large of a network leads to *overfitting* and poor accuracy.



- Activation values are represented as a_x^y, where x is the position with a layer and y is the layer.
- Each connection from some a_x^y to $a_z^{(y+1)}$ has a weight w_x^y, associated with the originating and destination nodes.

- To find the value for some node $a_r^{(c)}$, we use the formula $a'_r{}^{(c)} = \sigma(\mathbf{A}\mathbf{w}^{\mathbf{c-1}}+b)$, where **w** and **b** are vectors of weights and *biases.* $a'_0{}^{(2)}=\sum^{n}_{i=0}(a_i{}^{(1)}*w_{i,1})+b$
- To get the number between 0 and 1, a *regularizer* function is used. The sigmoid function is one such regularizer, i.e., $a_1^{(0)}=\sigma(1/(1+e^{-a'})$
- Biases can be used to ensure a value is greater than some other value, e.g., $a'_1{}^{(0)}=\sum_{i=0}^{783}(a_0{}^{(i)}*w_{0,i})-10$

This can be written as

$$\begin{bmatrix} w_{0,0}, w_{0,1} \dots w_{0,n} \\ W_{0,0,} w_{0,1} \dots w_{0,n} \\ \dots \\ W_{0,0,} w_{0,1} \dots w_{0,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \dots \\ a_n^{(0)} \end{bmatrix} = \begin{bmatrix} a_0^{(1)} \\ \\ \\ \\ \end{bmatrix}$$

- This computes one element
- A full matrix multiply computes all of the *a's* of row 1

We'll see the effect this has on TPU architectures.

Apply the regularizer function to this to normalize (the sigmoid function, in our case

$$\begin{bmatrix} w_{0,0}, w_{0,1} \cdots w_{0,n} \\ W_{0,0,} w_{0,1} \cdots w_{0,n} \\ \cdots \\ W_{0,0,} w_{0,1} \cdots w_{0,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \cdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ \cdots \\ b_n^{(1)} \end{bmatrix} = \begin{bmatrix} a_0^{(1)} \\ \\ \\ \\ \end{bmatrix}$$

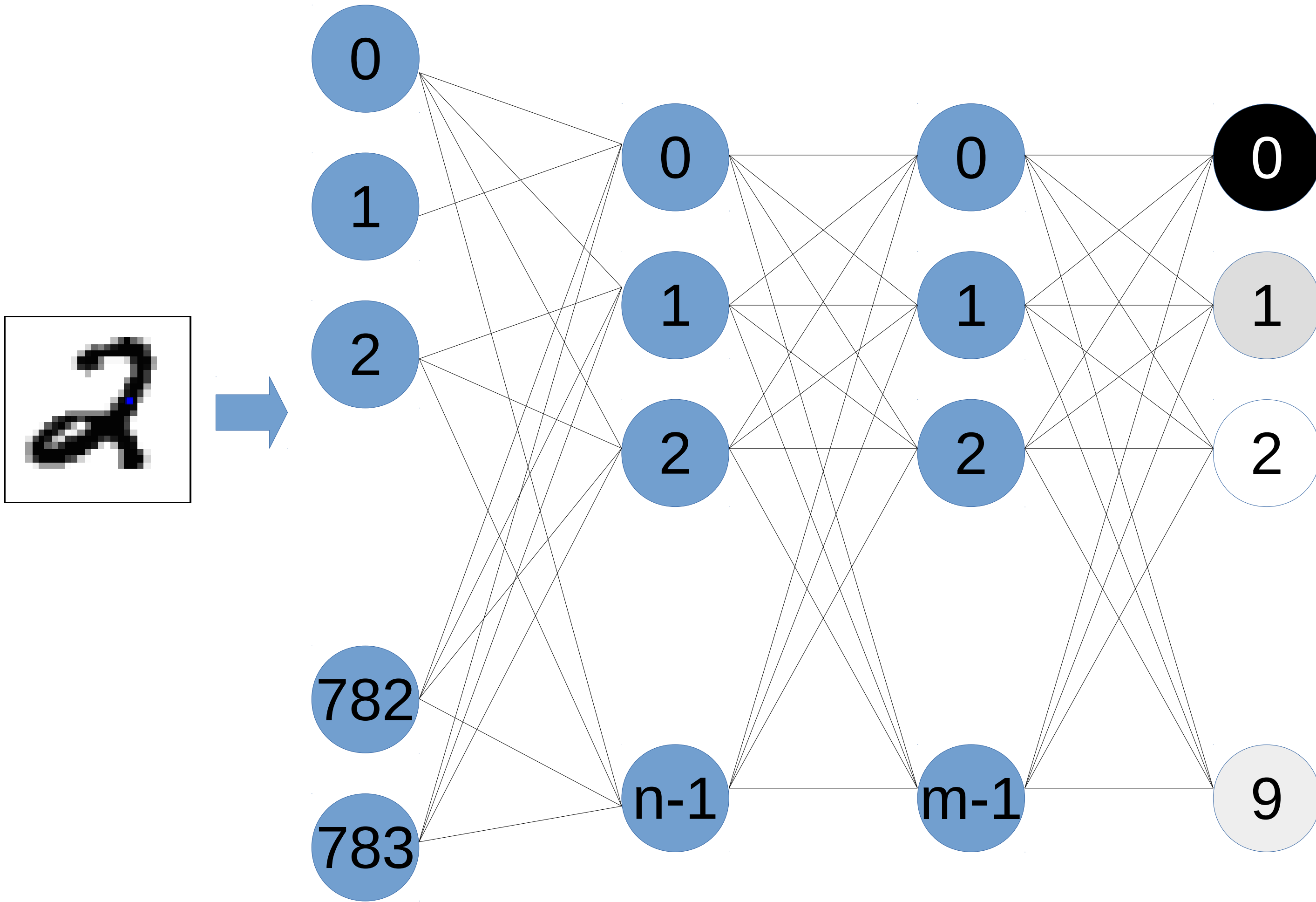
What does it mean to train the neural network?

- Training is simply the setting of the weights and biases appropriately.
- We can do this using *gradient descent* and *back propagation*, which we discuss next.
- To train the network using a data set with inputs and labels that are the correct answer.
- We train for a given number of *epochs* (passes over the training data) or until a *loss* function says we are good. In either case, the loss function is a measure of how good the algorithm recognizes the *training data.*
- We'll start out with random weights and biases and train them to something better.

The loss function (cost in the tutorial mentioned in the title slide)

- Many cost functions are available – we’ll discuss a little more with tensorflow
- We’ll use sum of squares of the error, because it is simple
- Let’s return to our number recognition problem.
  - If a 2 is the number to recognize, ideally the last layer will have 1 for node for 2, and 0 for everything else.
  - Loss is how far we deviate from this.

$$loss=\Sigma^{10}_{i=0}(a_i(3)\text{-expected})^2$$

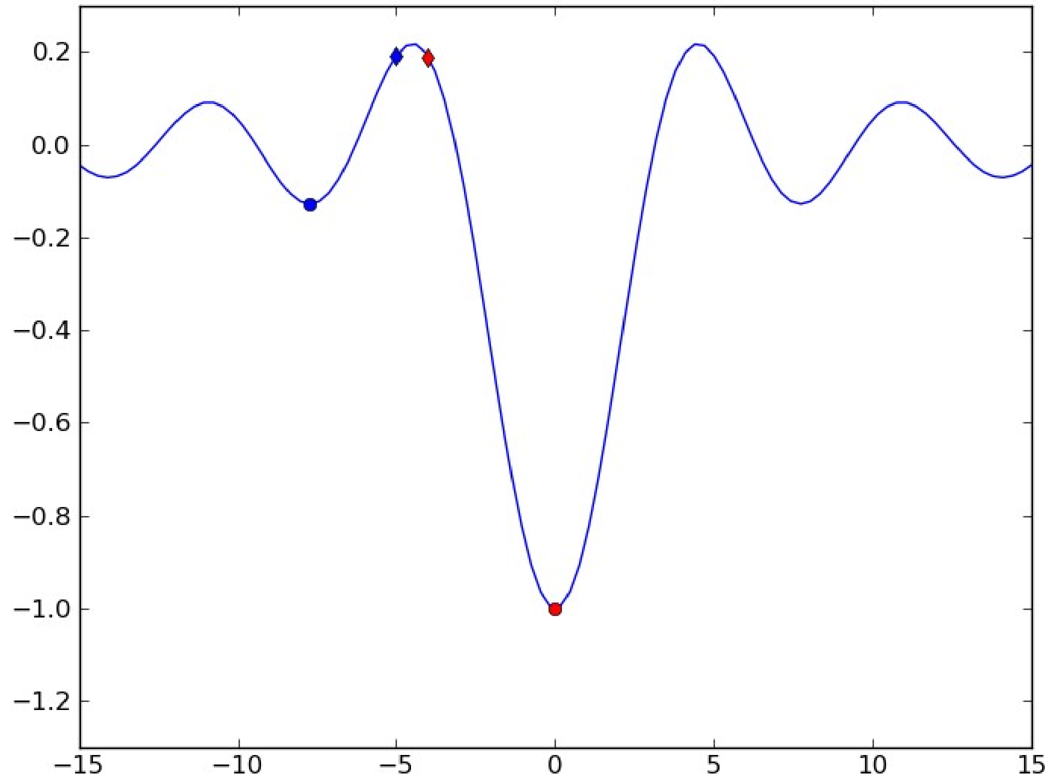


Basic training strategy

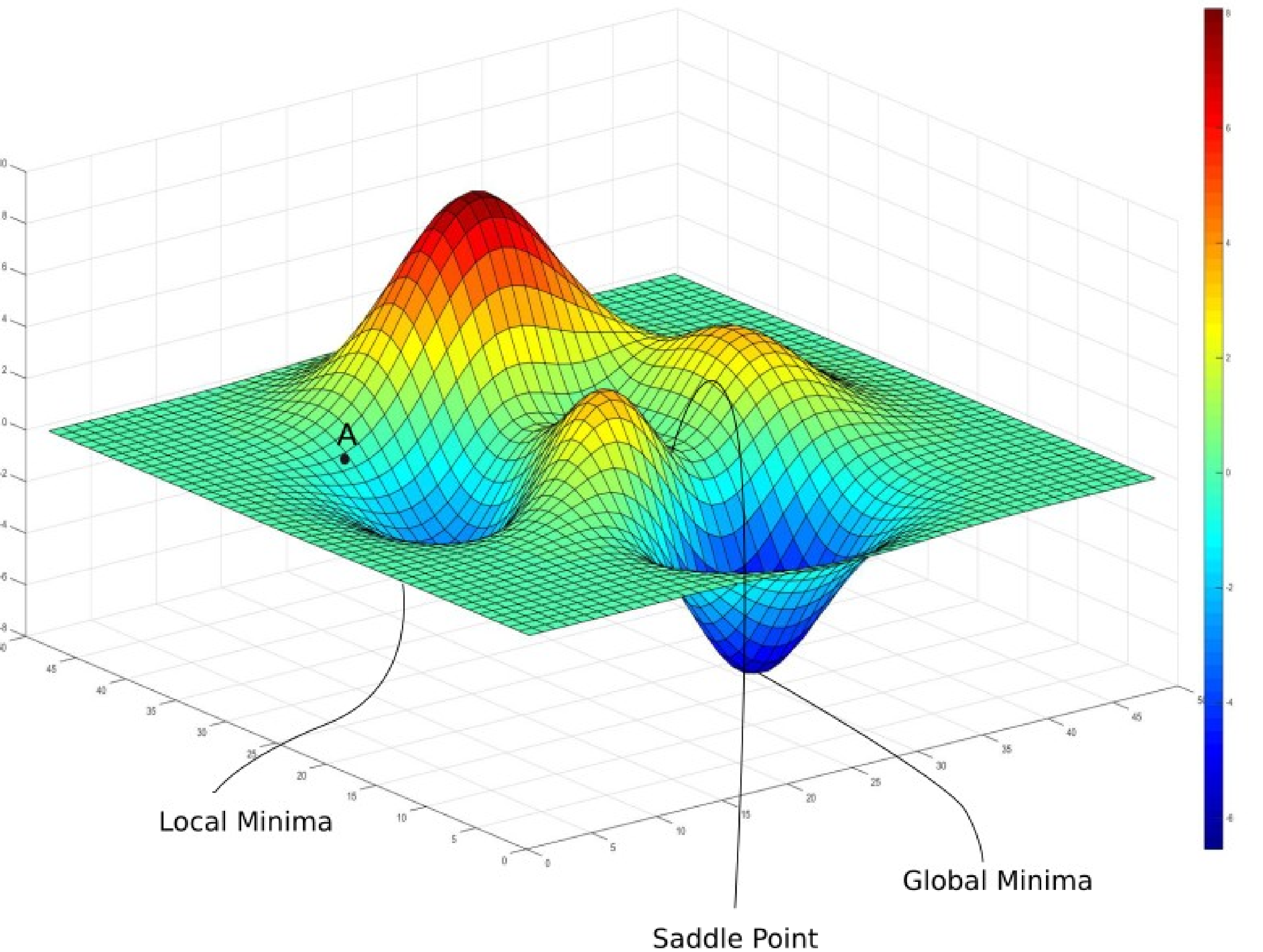
- Feed the training data into the randomly initialized neural network
- Compute the loss function
- Use gradient descent, or another optimizer, to tune the weights and biases
- Repeat until satisfied with the level of training

A neural network is a function

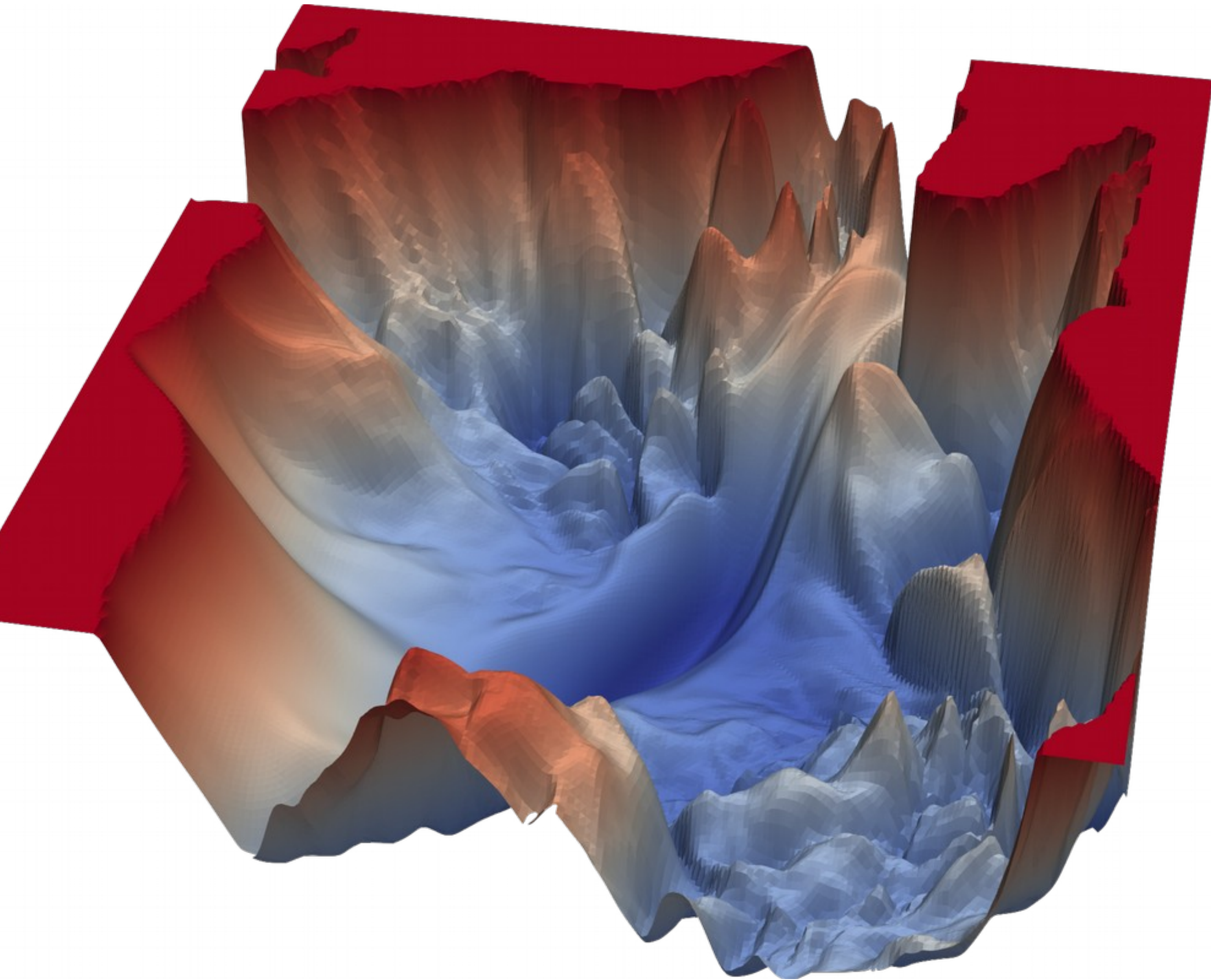
- We have 13002 weights and biases
- The neural network is a function of these weights and biases
- We want to adjust the weights and balances to minimize the loss function



- A function in one variable
- Minimum found using derivative of the function
- Local minima are an issue.



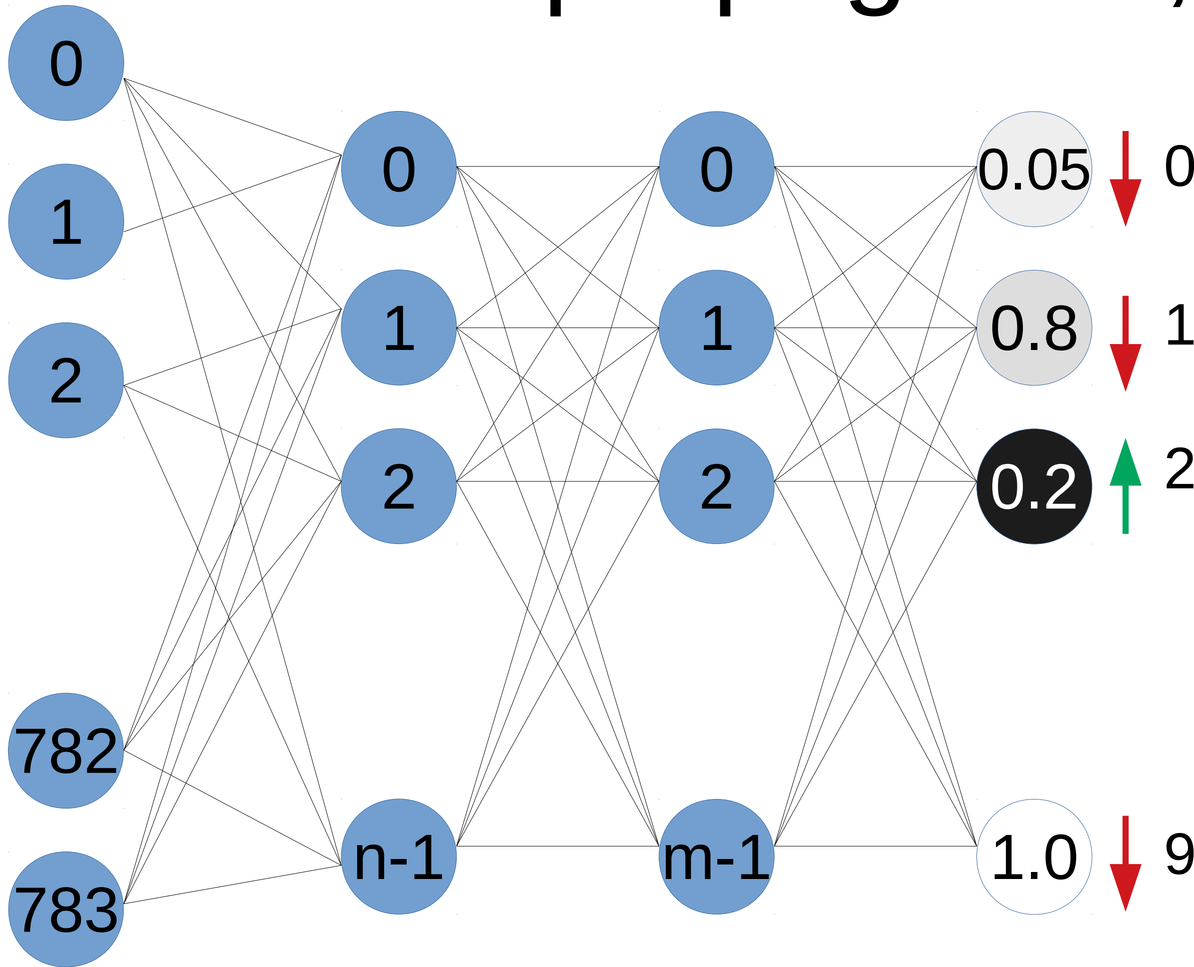
- A fairly nice function in 2 variables



- Visualization of a function represented by some neural network

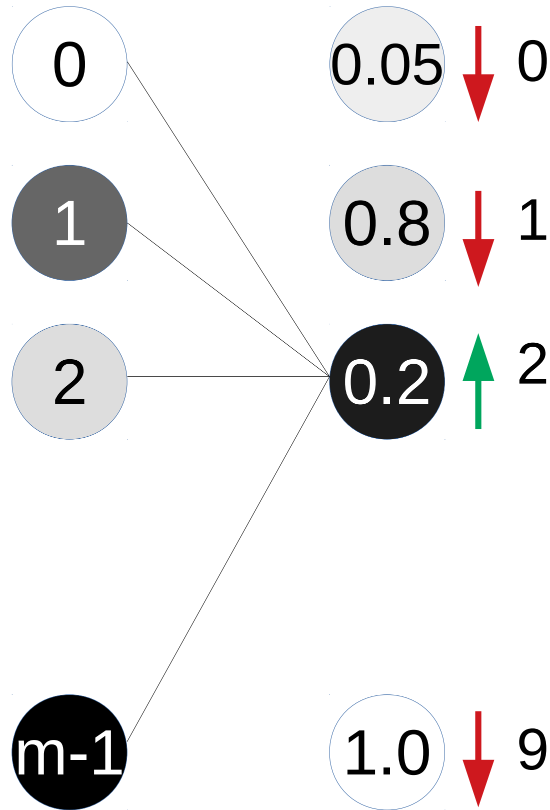
- We have thousands of inputs, 13002 weights and biases of our function, X variables, one output (the loss)
- We have local minima that should be avoided
- The negative of the gradient gives us the direction of steepest descent, drives us to the closest (local or global) minimum by giving us the changes in each of the 13002 weights and biases to move towards the local or global minimum.
- Having continuous activations is necessary to make this work, whereas biological neurons are more binary

Back propagation, input is 2

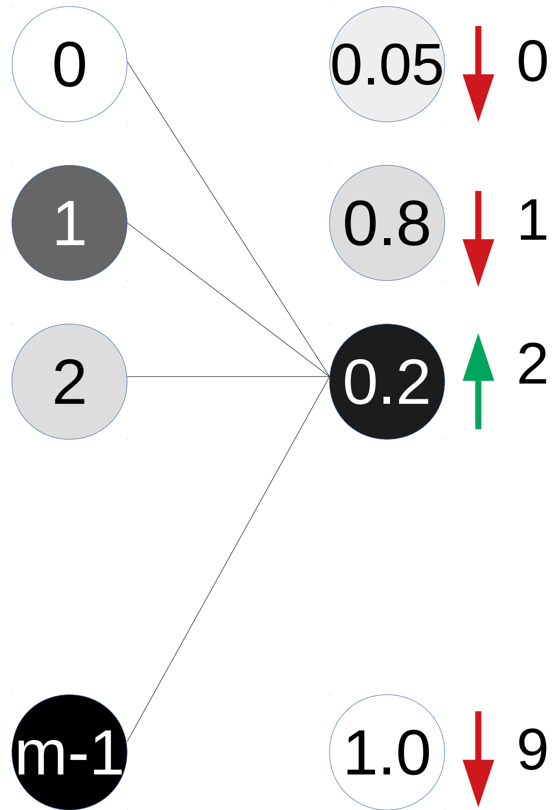


The 0 output is pretty close, but the contribution of the 9 output, is very high and contributes most to the error.

But let’s focus on the neuron we want to increase.



- $a'^{(4)}_2=\Sigma^{n}_{i=0}(a_i^{(3)*}w_{i,3})+b$
- Three ways t change the value of 2's neuron:
  - Change the value of the bias, b
  - Increase $w_{i,3}$
  - Change the value of $a_i^{(3)}$
- Changing the weights associated with brighter, high valued neurons feeding into 2 has more of an effect than changing the value of darker low-valued neurons.;



- Changing the values of the activations, i.e., the a values, associated with the nodes feeding into two will change the value of 2
- Increasing a values with positive weights, and decreasing those with negative weights, will increase the value of two.
- Again, changes of values associated with with weights with a larger magnitude will have a larger effect.

The other output neurons affect this

- The non-two neurons need to be considered
- Add together of all the desired effects on non-two nodes and the two-node tells us how to nudge weights and biases from the previous layer
- Apply this recursively to more previous layers
- These nudges are, roughly proportional to the negative gradient discussed previously
- This is back propagation.

Computational issues

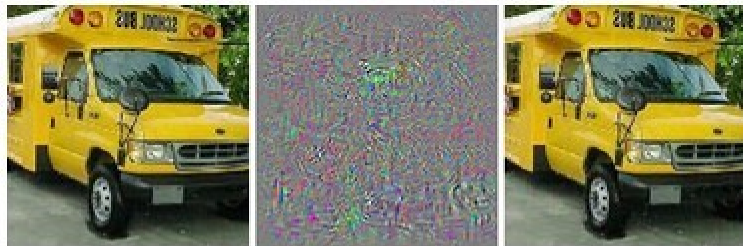
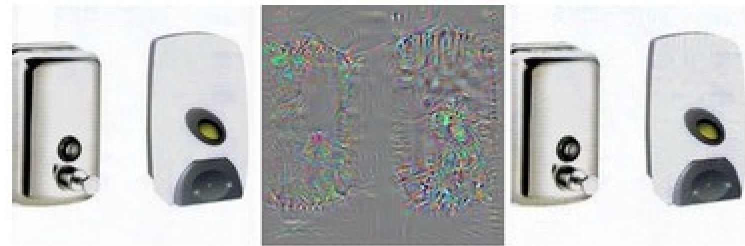
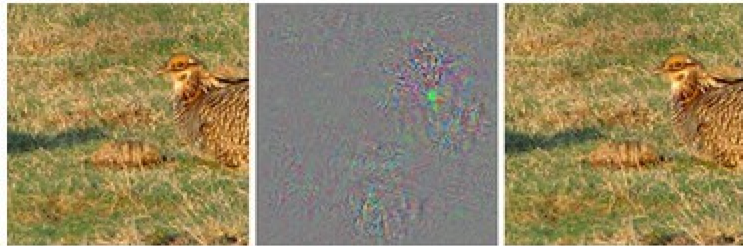
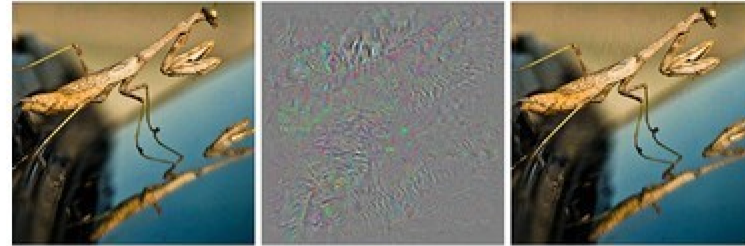
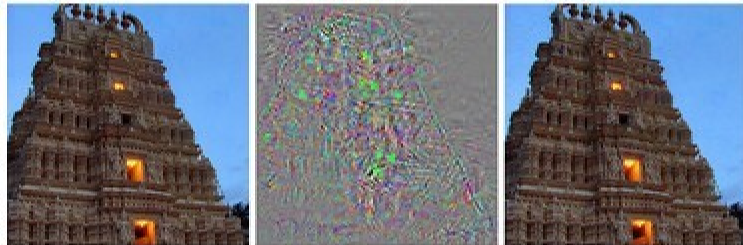
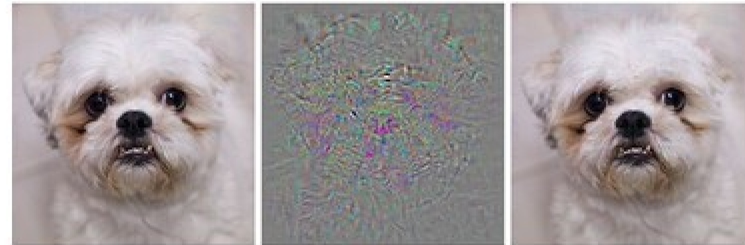
- Doing this for every input data point on every training step (epoch) is computationally complex.
- Solution:
  - *Batch* the data into chunks of data
  - In each epoch, train on one batch at a time

A problem with neural networks

- You might think that different layers begin to identify characteristics of the network, the next layers puts these together into larger parts of the number, and finally it identifies a 2
  - That’s not what happens
  - State of a layer looks pretty random compared to what it is recognizing
- Random patterns will often be strongly identified as a number.

Adversarial networks

https://arxiv.org/pdf/1712.09665.pdf

correct +distort ostrich correct +distort ostrich

Perturbed images are pasted onto signs

https://spectrum.ieee.org/cars-that-think/transportation/sensors/slight-street-sign-modifications-can-fool-machine-learning-algorithms

- Stop signs identified as speed limit 45 signs, right turn as stop signs

TPU Architecture

- Training is expensive – hours, days and weeks
- A result of real neural networks being complicated, and training data sets needing to be large (tens to hundreds of thousands of elements for classifiers). MNIST is ~10K images, and is small in overall size.
- Training involves lots of matrix multiplies
- So build a processor to do that

- Google had an ASIC (application specific integrated circuit) in 2006

A convolution

- Weights = $\{w_1, w_2, \ldots, w_k\}$, inputs $x = \{x_1, x_2, \ldots, x_k\}$ and outputs y $\{y_1, y_2, \ldots, y_k\}$
- $y_i = w_i x_i + w_{i+1} x_{i+1} + w_{i+2} x_{i+2} + \ldots + w_k x_k$
- As an example, let $k = 3$
- $y_1 = w_1 x_1 + w_2 x_2 + w_2 x_2$
- $y_2 = w_2 x_2 + w_3 x_3 + 0$
- $y_3 = w_3 x_3 + 0 + 0$

Computing this on a simple processor

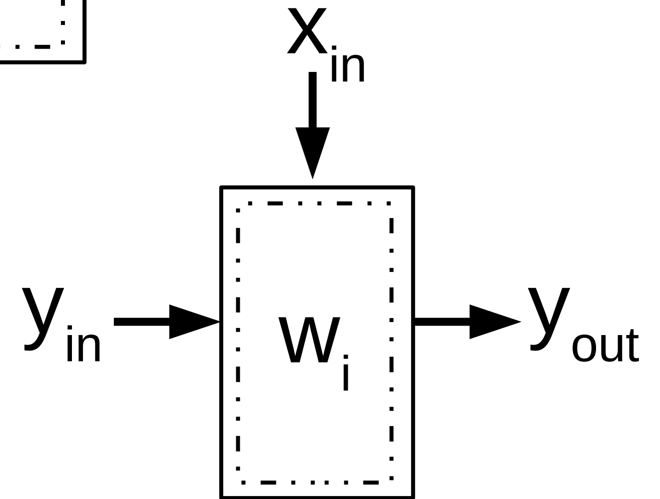
- Assume each input is read for each operation.
- 12 input values read for 3 results, bad I/O from memory balance
- $y_1 = w_1x_1 + w_2x_2 + w_2x_2$
- $y_2 = w_2x_2 + w_3x_3 + 0$
- $y_3 = w_3x_3 + 0 + 0$
- Systolic arrays, which “pump” data through the processor, can help

A simple systolic array

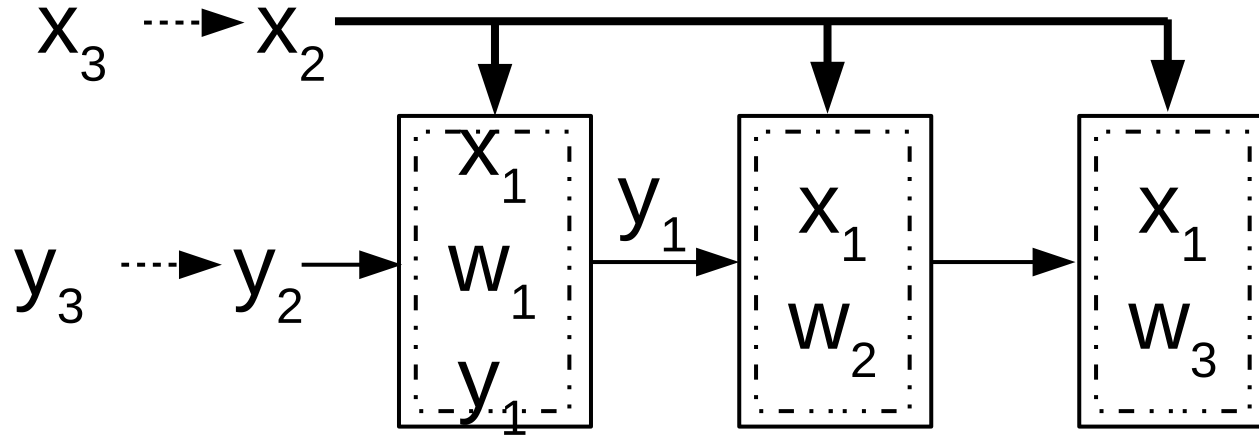
x_3 → x_2 → x_1

y_3 → y_2 → y_1 → w_1 → w_2 → w_3

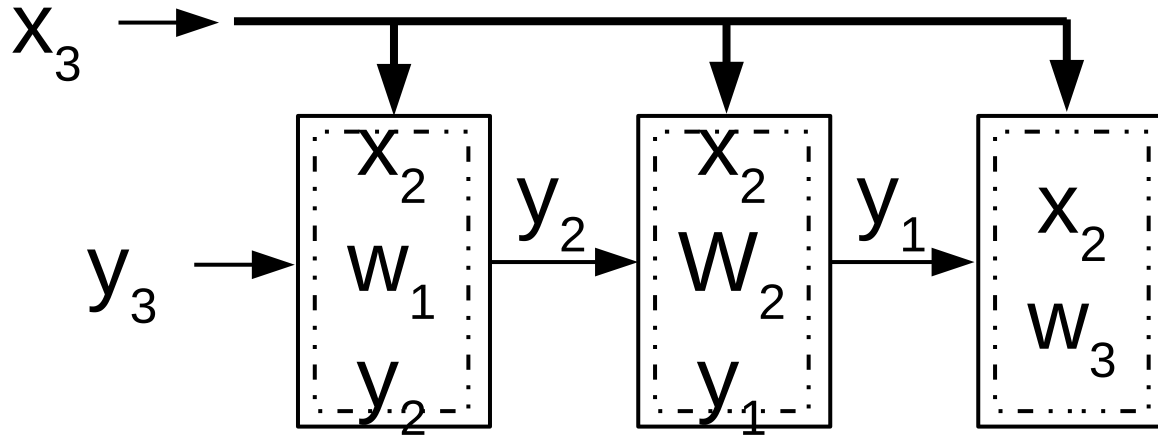
6 data elements are fetched to do the computation. Even for this small problem, 50% less data



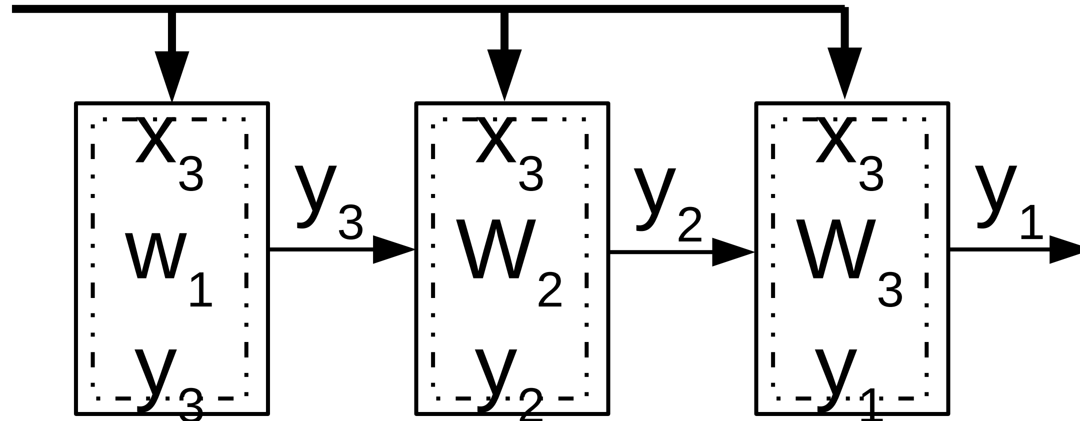
Step 1



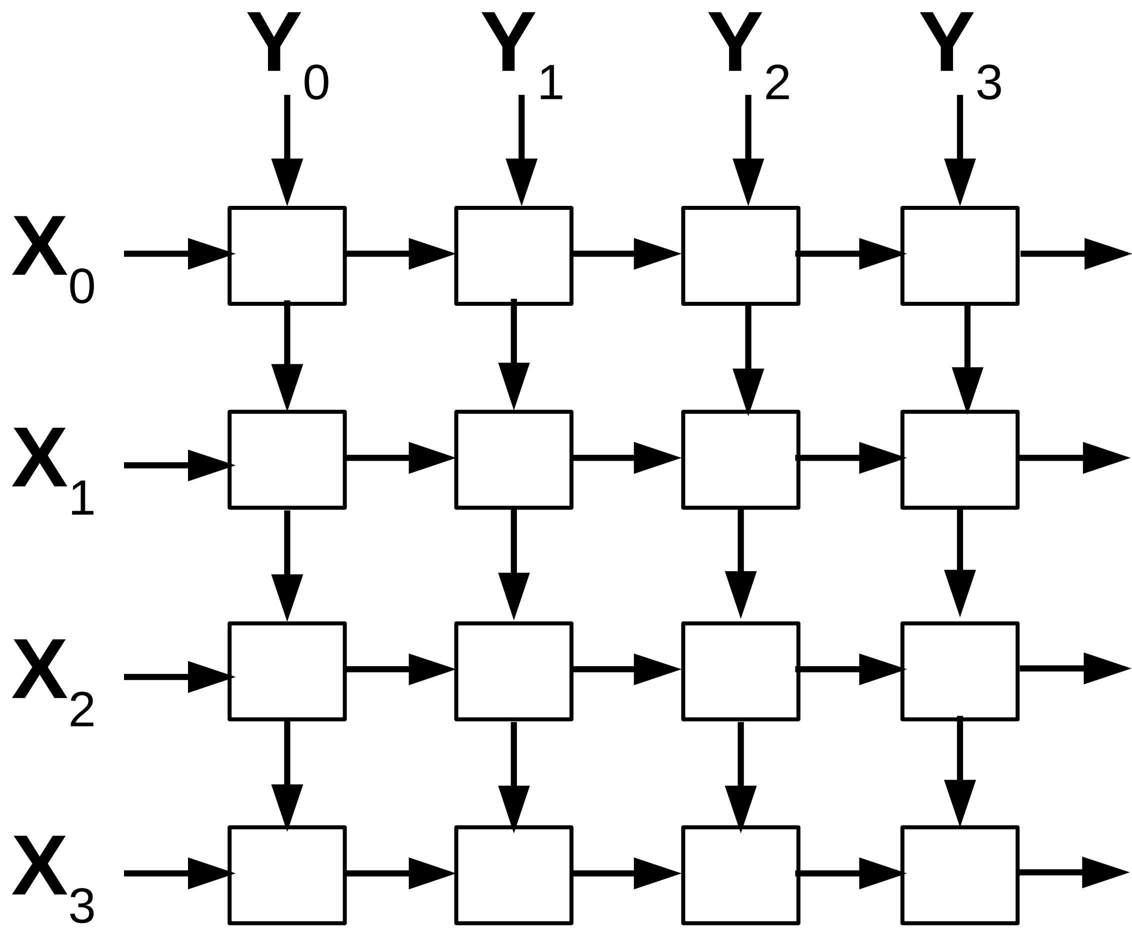
Step 2



Step 3



After step 3, in this small example, pump out the values for y_2 and y_3



Can do the same thing in 2 dimensions

TPUs

- https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu