Machine Learning

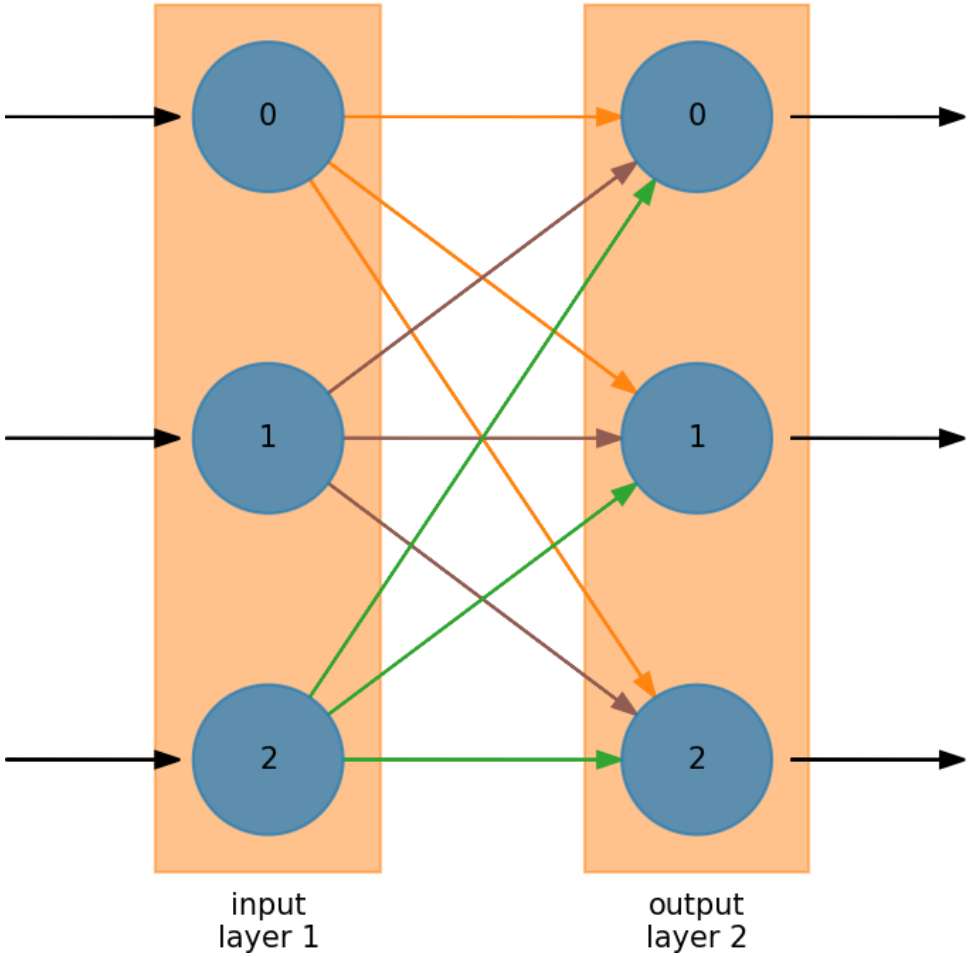
Neural Network Overview

- Computers are good at arithmetic but not great at pattern recognition
  - Neural nets attempt to model how neurons transmit information

(original source unknown)

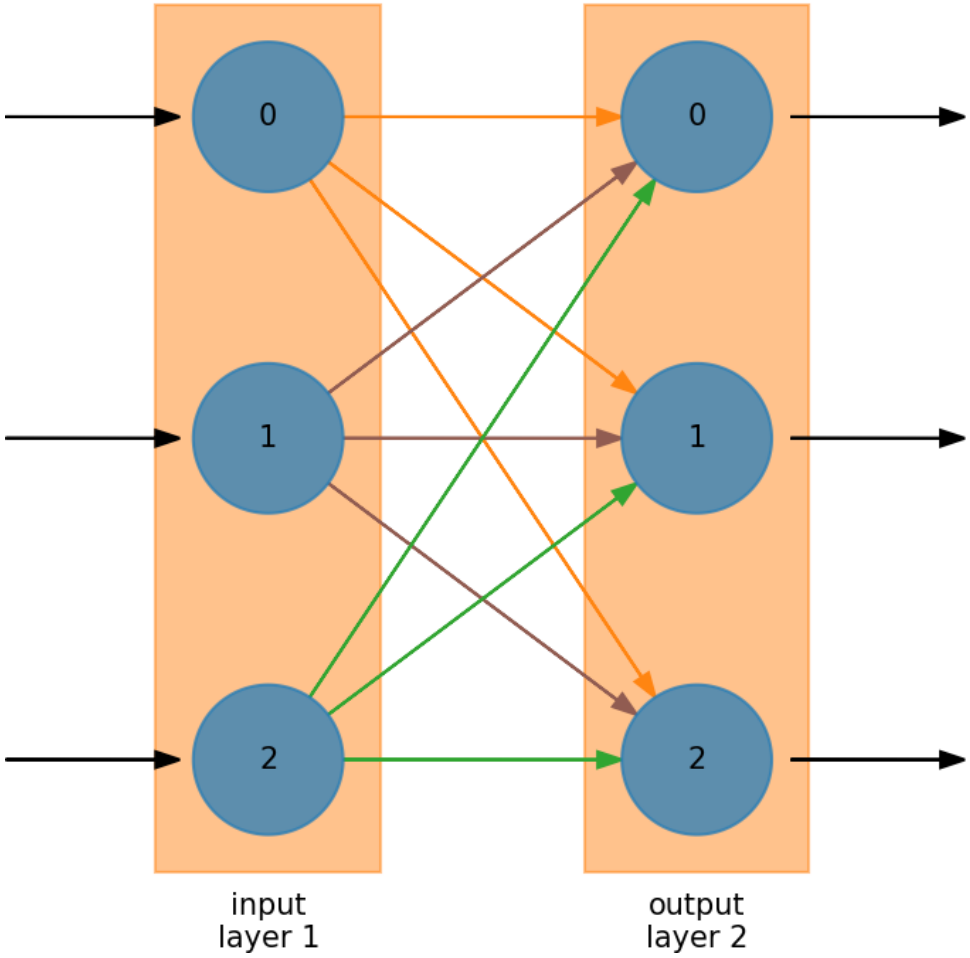
Neural Network Overview

- Some nomenclature:
  - Neural networks are divided into *layers*
    - There is always an input layer—it doesn't do any processing—just accepts the input
    - There is always an output layer
  - Within a layer, there are neurons or *nodes*
    - For input, there will be one node for each input variable
  - Every node in the first layer connects to every node in the next layer
    - The *weight* associated with the *connection* can vary—these are the matrix elements
  - In this example, the processing is done in layer 2 (output)



Neural Network Overview

- When you train a neural network, you are adjusting the weights connecting the nodes
- Some connections may have zero weight
- This mimics nature—a single neuron can connect to several (or lots) of other neurons
- Linear algebra problem:
  - Inputs: $\mathbf{x} \in \mathbb{R}^n$
  - Outputs: $\mathbf{z} \in \mathbb{R}^m$
  - Neutral network is a map, $\mathbb{R}^n \rightarrow \mathbb{R}^m$ that can be expressed as a matrix, $\mathbf{A}$
    - $\mathbf{z} = \mathbf{A}\mathbf{x}$
  - Given enough input, we could know all the matrix elements in $\mathbf{A}$



Nonlinear Model

- We'll use a nonlinear function, g(p), that acts on a vector:

$$g(\mathbf{x}) = \begin{pmatrix} g(x_0) \\ g(x_1) \\ \vdots \\ g(x_{n-1}) \end{pmatrix}$$

  - then **z** = g(**A x**)
- New procedure: set the entries of **A** via training, using a simple, nonlinear, g(p) that fits our training data
- From the graphical representation, the nonlinear function is applied on the output layer

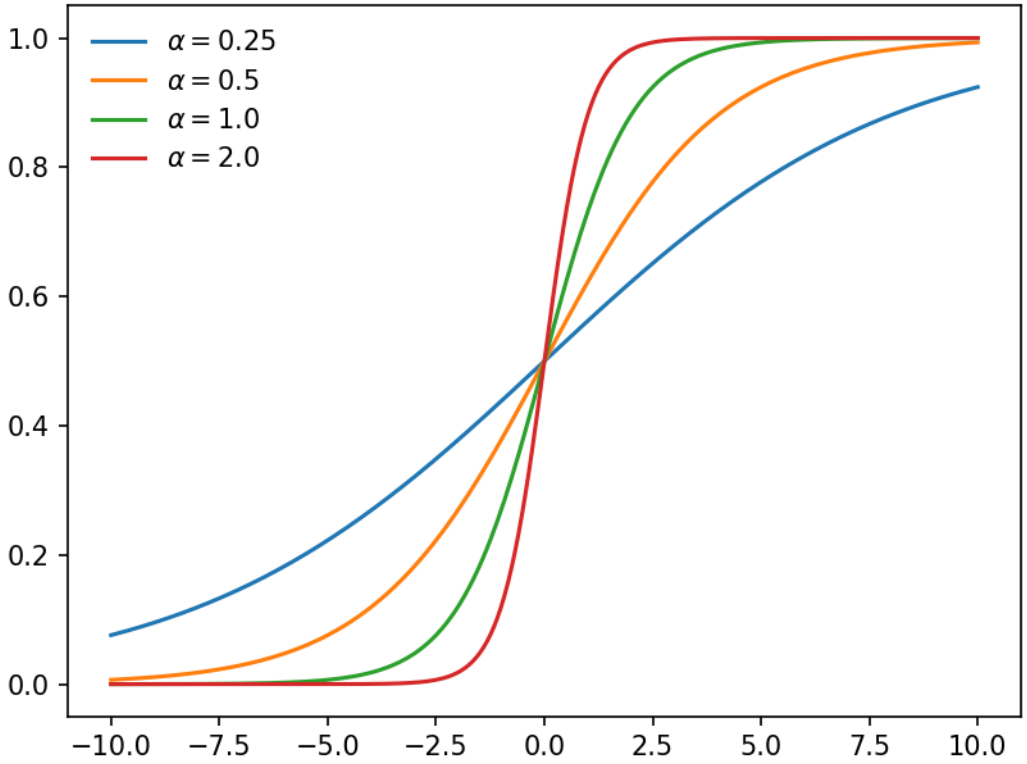
Nonlinear Model

- Again, this mirrors biology
  - Neurons don't act linearly
  - There is a threshold that needs to be reached before a neuron "fires"
- A step function would work, but we want something differentiable
- There are a lot of different choices in the literature

Sigmoid Function

- Common choice: sigmoid function

$$g(p) = \frac{1}{1 + e^{-\alpha p}}$$



- Note, all outputs are scaled to be $z_j \in (0, 1)$
- We'll take $\alpha = 1$

Scaling Output

- Note that since the sigmoid maps all output to (0, 1), we need to make sure that the output in our training set is likewise mapped to (0, 1)
  - If the data doesn't already fall in (0, 1), we can just use a linear transformation:

$$\tilde{x} = 0.9\,\frac{x - \min x_i}{\Delta x} + 0.05$$

    - here, Δx is the largest possible range of x_i in the inputs

Actually, (0, 1] works fine—we just need to avoid a 0, since that cancels out weights in the matrices

Implementation

- Basic operation
  - Train the model with known input/output to get all A_{ij}
  - Use $\mathbf{z} = g(\mathbf{A}\,\mathbf{x})$ to get output for a new input $\mathbf{x}$
- Training:
  - We have T pairs $(\mathbf{x}^k, \mathbf{y}^k)$ for k = 1, …, T
    - Important: remember that our **y**'s have to be scaled to be in (0, 1), so they are in the same range that our function g(p) maps to
  - We require that $g(\mathbf{A}\,\mathbf{x}^k) = \mathbf{y}^k$ for all k
    - Recall, that g(p) is a scalar function that works element-by-element:

$$z_i = g([\mathbf{A}\mathbf{x}]_i) = g\left(\left[\sum_j A_{ij} x_j\right]\right)$$

Implementation

- Training (cont.)
  - We find the elements of **A**
    - This can be expressed as a minimization problem, where we alter the matrix elements to achieve this agreement
    - There may not be a unique set of A_{ij}, so we will loop randomly over all training data multiple times to optimize **A**

$$f(A_{ij}) = \|g(\mathbf{A}\mathbf{x}^k) - \mathbf{y}^k\|^2$$

      - This looks like a least-squares minimization
- The function we minimize is called the *cost function*
  - There are other choices than the square of the error

Implementation

- Minimization
  - A common technique for minimization is *gradient descent* (sometimes called steepest descent)
    - This looks at the local derivative of the function f with respect to the parameters A_{ij} and moves a small distance *downhill*, and iterates…
  - We'll also compare to an external library for minimization
- Caveats
  - When you minimize with one set of training data, there is no guarantee that you are still minimized with respect to the previous sets
  - In practice, you feed the training data multiple times, in random order, to the minimizer—each pass is called an *epoch*

Aside: Minimization

- Steepest descent minimization
  - Start at a point $\mathbf{x}_0$ and evaluate the gradient
  - Move *downhill* by following the gradient by some amount η
  - Correct our initial guess and iterate

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \frac{\partial u}{\partial \mathbf{x}}$$

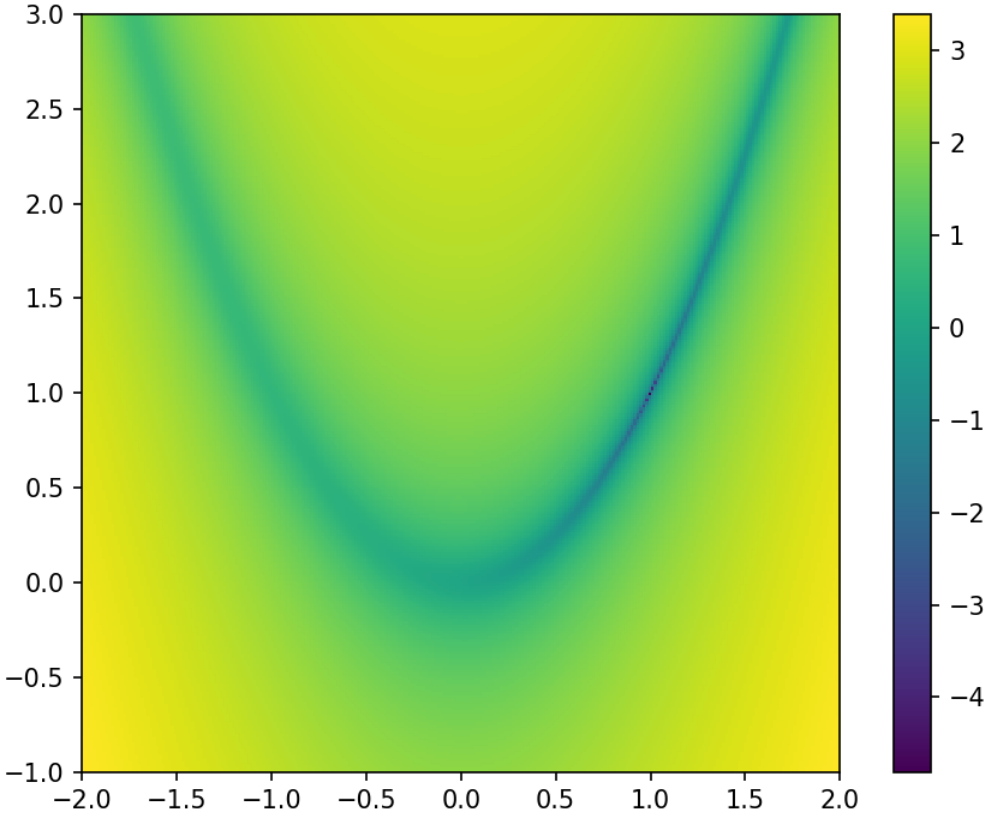
- Need to choose the amount to move each iteration
  - Sometimes we instead define a unit vector in the direction of the local gradient, and then η represents the distance to travel in that direction
- You can think about this as what happens if you put a marble on a surface—it rolls to a minimum
  - May not be the global minimum—we can get stuck in a local minimum

Aside: Minimization

- Example: Rosenbrock (banana) function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

  - This is a hard problem for optimization
  - Global minimum is at (a, a^2)

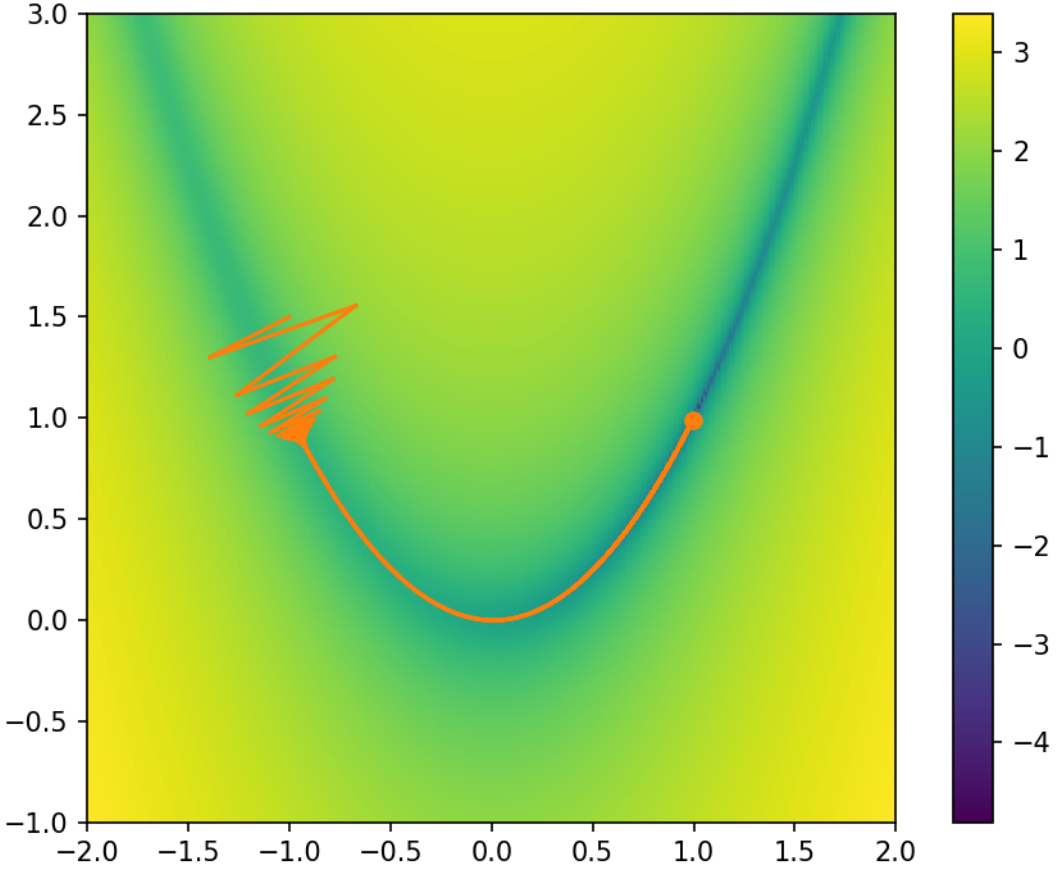


Note: this is the log of the function plotted

code: steepest_descent.py

Aside: Minimization

- Minimization with gradient descent is very sensitive to choice of η
  - Too large and you may shoot off far from the minimum
  - Too small and you do a lot of extra work



code: steepest_descent.py

Neural Net Minimization

- For our function,

$$f(A_{ij}) = \|g(\mathbf{A}\mathbf{x}^k) - \mathbf{y}^k\|^2$$

  - Note, this definition is for a single training pair, ($\mathbf{x}^k$, $\mathbf{y}^k$)

$$(\mathbf{x}^k, \mathbf{y}^k) = (\{x_1^k, x_2^k, \ldots, x_n^k\}, \{y_1^k, y_2^k, \ldots, y_m^k\})$$

  - Our update would be

$$A_{pq} = A_{pq} - \eta \frac{\partial f}{\partial A_{pq}}$$

  - where

$$f(A_{ij}) = \sum_{i=1}^{m} \left[g\left(\sum_{j=1}^{n} A_{ij} x_j \right) - y_i \right]^2$$

Neural Net Minimization

- Working out the derivative:

$$\frac{\partial f}{\partial A_{pq}} = 2(z_p - y_p)\alpha z_p(1 - z_p)x_q$$

  - We could then use steepest descent, looping over the matrix elements and doing the minimization on them one by one, iterating until we converge
    - Instead, we just do one push "downhill" following the gradient for a single training set and then move to the next.
    - η is often called the *learning rate*
- Gradient descent is often used for neural nets because it only requires the first derivative
  - Newton's method would require the second derivatives (Hessian matrix)

Neural Net Minimization

- Recall,
  - **A** is m × n matrix
  - **x** is n × 1 vector
  - **y** (and hence **z**) is m × 1 vector
- We can write our derivative as:

$$\frac{\partial f}{\partial \mathbf{A}} = \underbrace{2(\mathbf{z}-\mathbf{y}) \circ \alpha\mathbf{z} \circ (1-\mathbf{z})}_{m\times 1} \cdot \underbrace{\mathbf{x}^\mathsf{T}}_{1\times n}$$

- Then the correction to our matrix is:

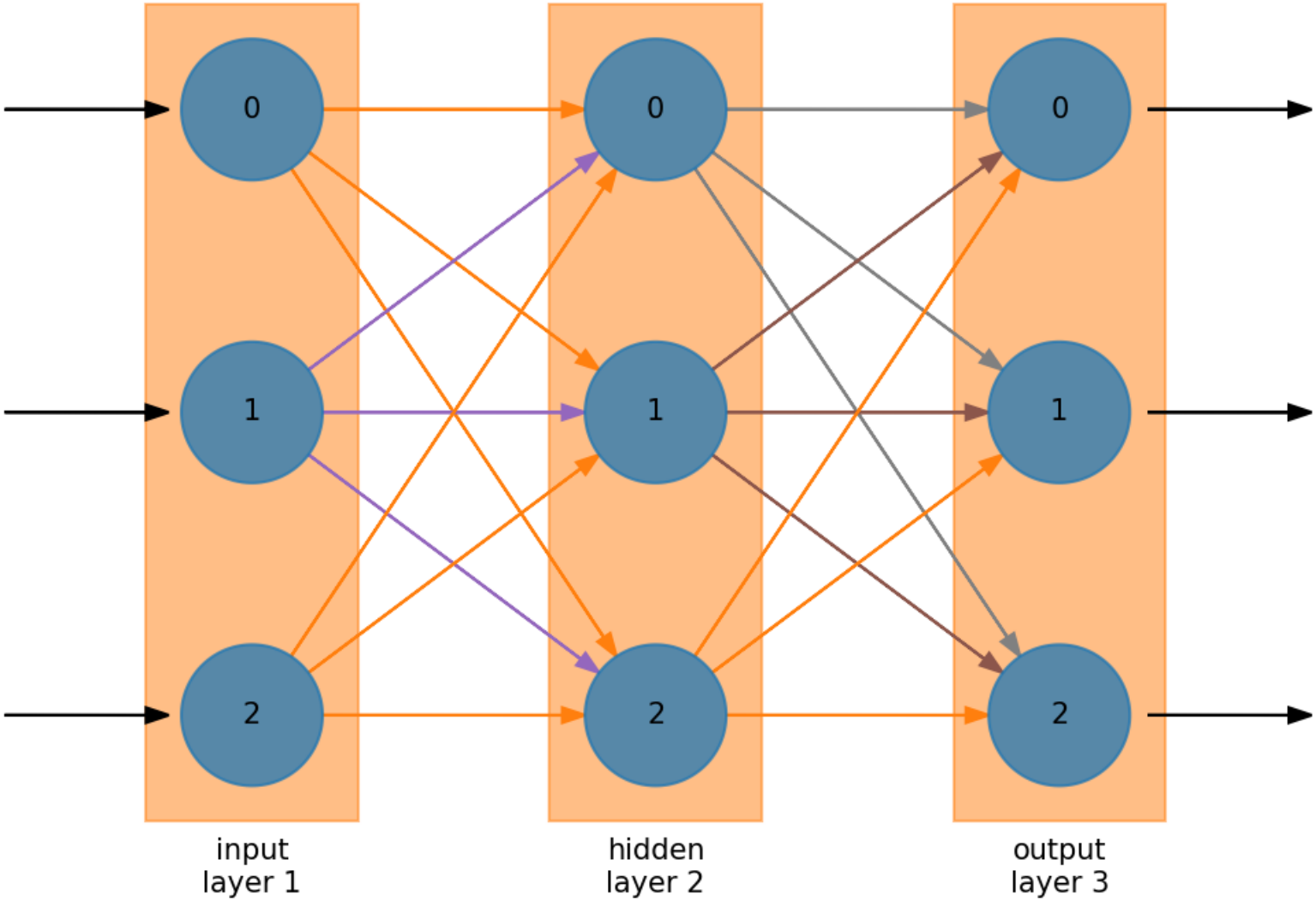
$$\Delta\mathbf{A} = -2\eta\,(\mathbf{z}-\mathbf{y}) \circ \alpha\mathbf{z} \circ (1-\mathbf{z}) \cdot \mathbf{x}^\mathsf{T}$$

$$\mathbf{A} \leftarrow \mathbf{A} + \Delta\mathbf{A}$$

Here, **a** ◦ **b** is an element-wise product

Hidden Layers

- We can add more more parameters by another layer of nodes/neurons



Hidden Layers

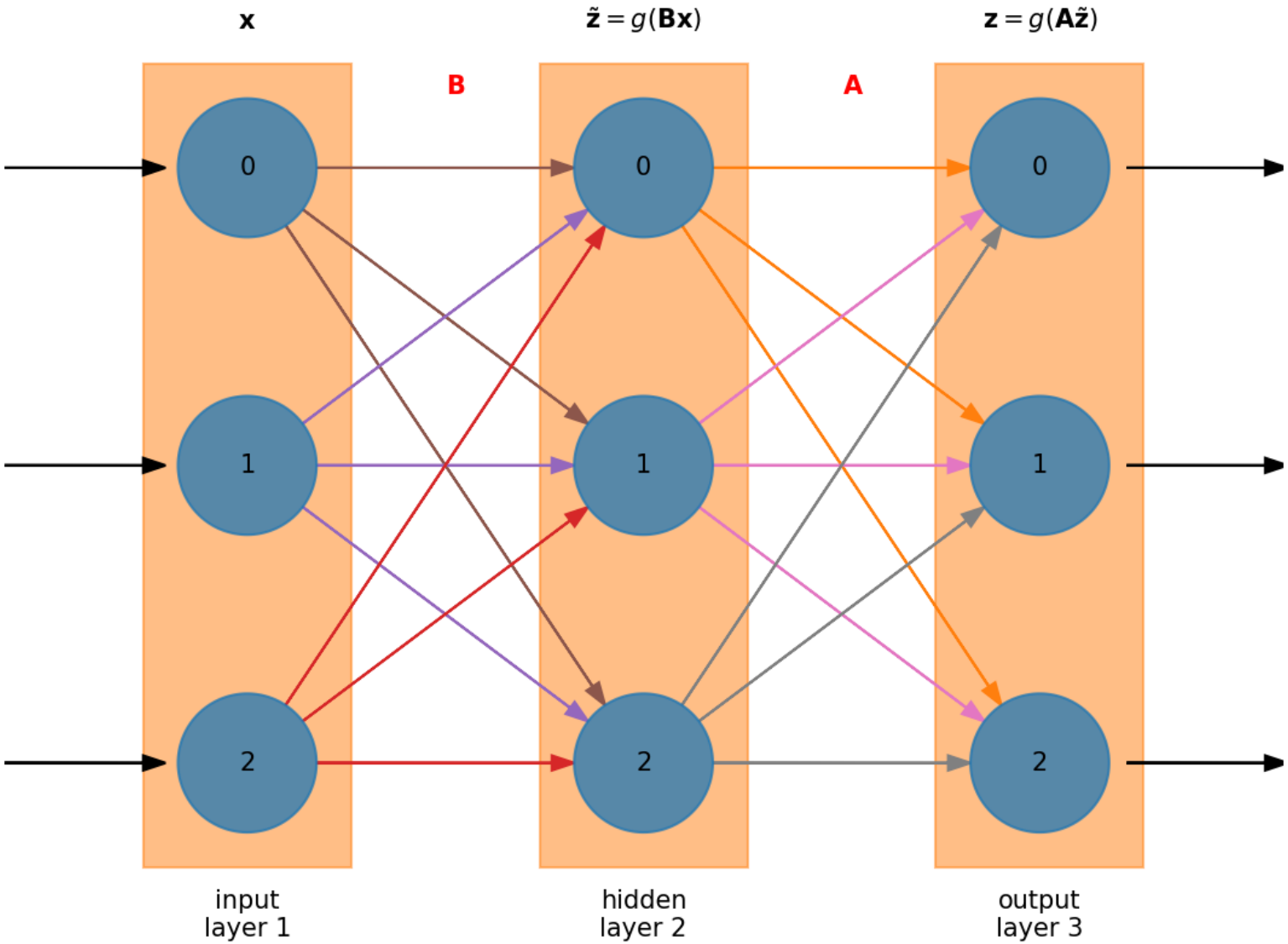
- *Hidden layers* sit between the input and output
- For hidden layer of dimension k:
  - Inputs: **x** $\in \mathbb{R}^n$
  - Outputs: **z** $\in \mathbb{R}^m$
  - **A** is an m × k matrix
  - **B** is an k × n matrix
  - The product **AB** is m × n, as we had before
- *Universal approximation theorem*: single layer network can represent any continuous function
- We transform the input in two steps:

$$\tilde{\mathbf{z}} = g(\mathbf{B}\mathbf{x})$$

$$\mathbf{z} = g(\mathbf{A}\tilde{\mathbf{z}})$$

Hidden Layers

- Graphically this appears as:



Hidden Layers

- Now we minimize:

$$f(A_{ls}, B_{ij}) = \sum_{l=1}^{m} (z_l - y_l)^2$$

$$\tilde{z}_i = g\left(\sum_{j=1}^{n} B_{ij} x_j\right)$$

$$z_l = g\left(\sum_{s=1}^{k} A_{ls} \tilde{z}_s\right)$$

Minimization

- We need to do the minimization now for both sets of weights (matricies)
- In practice, we do them one at a time, with each seeing the result from its layer
  - This process is also called *backpropagation* in neural networks—we are using the errors at the end to change the weights that came earlier in the network

Gradient Descent

- We can do our gradient descent on **A** and **B** separately now
  - This is the strength of backpropagation and gradient descent vs. some "canned" minimization routine—we are not optimizing the entire system all together
- Differentiating our error and lots of chain rule gives:

$$\Delta A = -2\eta\, \mathbf{e} \circ \mathbf{z} \circ (1 - \mathbf{z}) \cdot \tilde{\mathbf{z}}^\intercal$$
$$\Delta B = -2\eta\, \tilde{\mathbf{e}} \circ \tilde{\mathbf{z}} \circ (1 - \tilde{\mathbf{z}}) \cdot \mathbf{x}^\intercal$$

Note: this is a single dot product, the combination of vectors on the left are multiplied element-by-element (the Hadamard product)

  - With

$$\tilde{\mathbf{e}} = \mathbf{A}^\intercal \mathbf{e} \circ \mathbf{z} \circ (1 - \mathbf{z}) \approx \mathbf{A}^\intercal \mathbf{e}$$

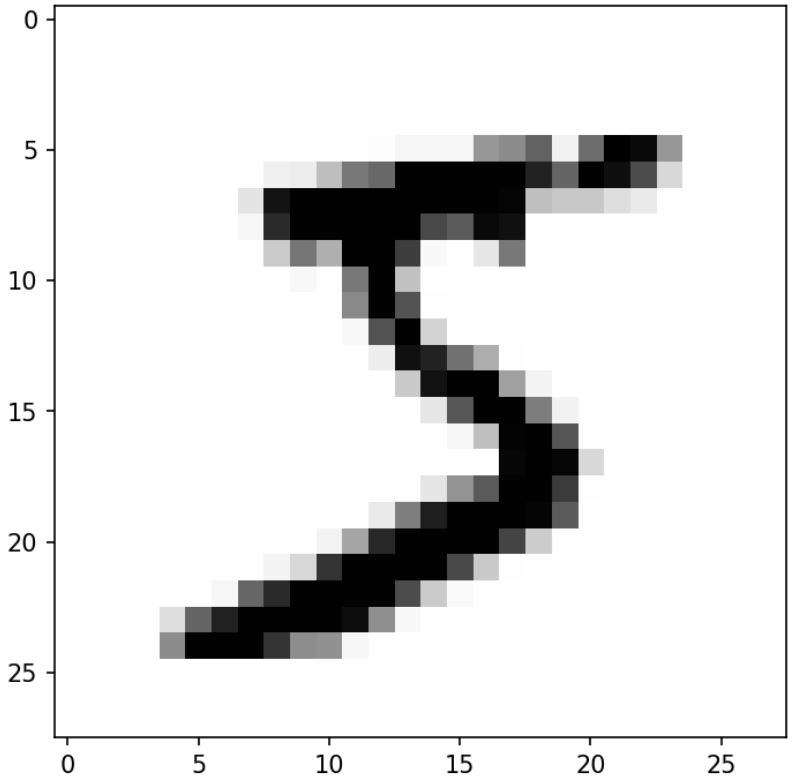
This approximation seems to be commonly made and supposedly doesn't affect convergence much

Image Classification

- We'll try to recognize a digit (0 – 9) from an image of a handwritten digit.
  - MNIST dataset (http://yann.lecun.com/exdb/mnist/)
    - Popular dataset for testing out maching learning techniques
    - Training set is 60,000 images
      - Approximately 250 different writers
    - Test set is 10,000 images
    - Correct answer is know for both sets so we can test our performance
- Image details:
  - 28 × 28 pixels, grayscale (0 – 255 intensity)
- The best learning algorithms can get accuracy > 99%

Image Classification

- Neural network characteristics:
  - Input layer will be 784 nodes
    - One for each pixel in the input image
  - Output layer will be 10 nodes
    - An array with an entry for each possible digit
    - "3" would be represented as: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
  - We'll start with a hidden layer size of 100
- We'll train on the training set, using up to 60000 images
  - Rescale the input to be in [0.01, 1]
- We'll test on the test set of 10000 images



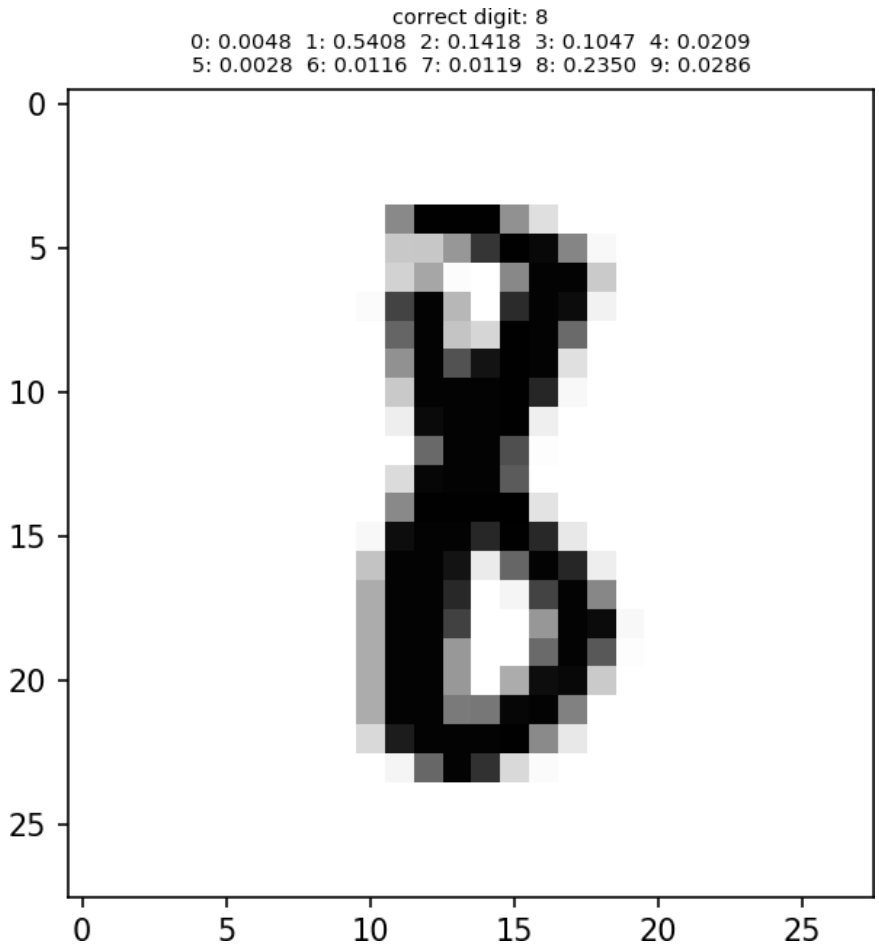
First digit MNIST in the training set

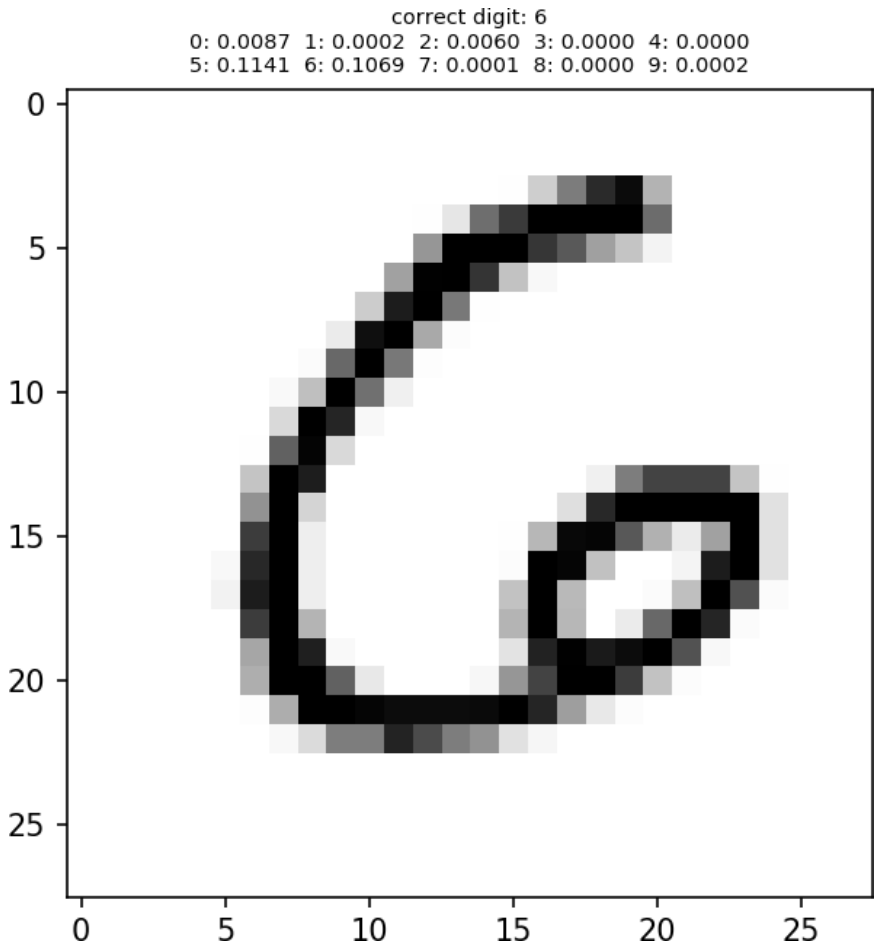
Image Classification

- Default configuration:
  - The full training set (60000 images)
  - Hidden layer of 100 nodes
  - 5 epochs of training
  - Learning rate of 0.1
- We achieve 95 – 96% accuracy

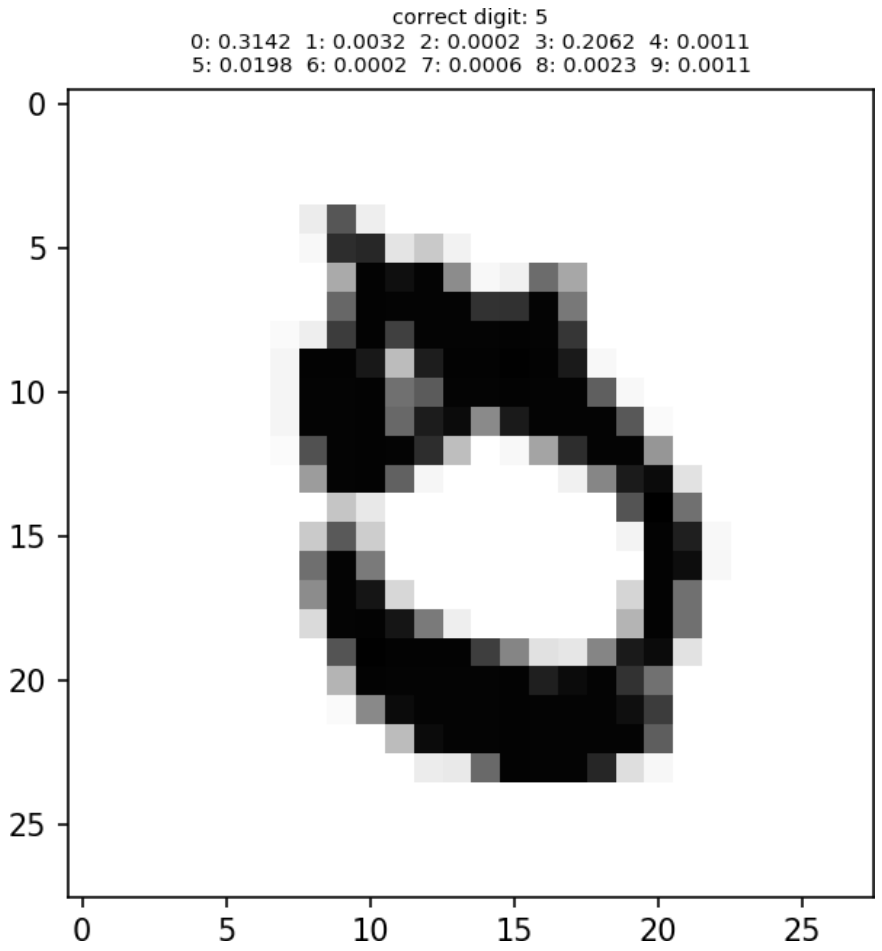
code: char_recognition.py

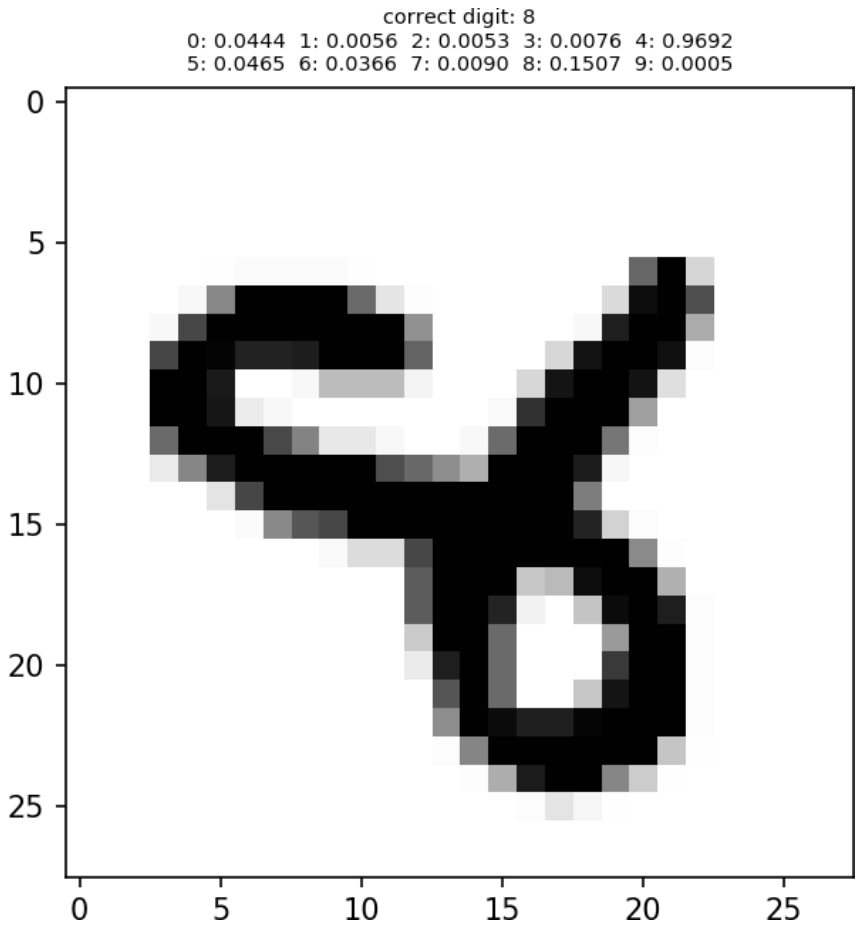
Some Image Classification Failures



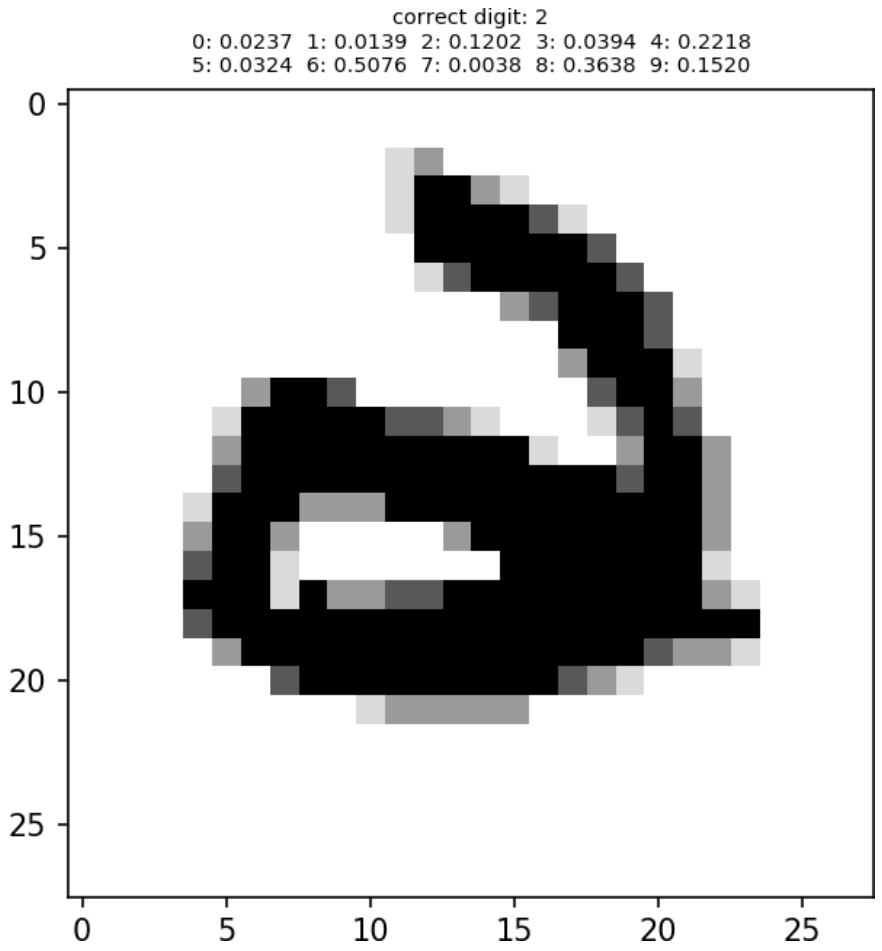


Some Image Classification Failures





Some Image Classification Failures



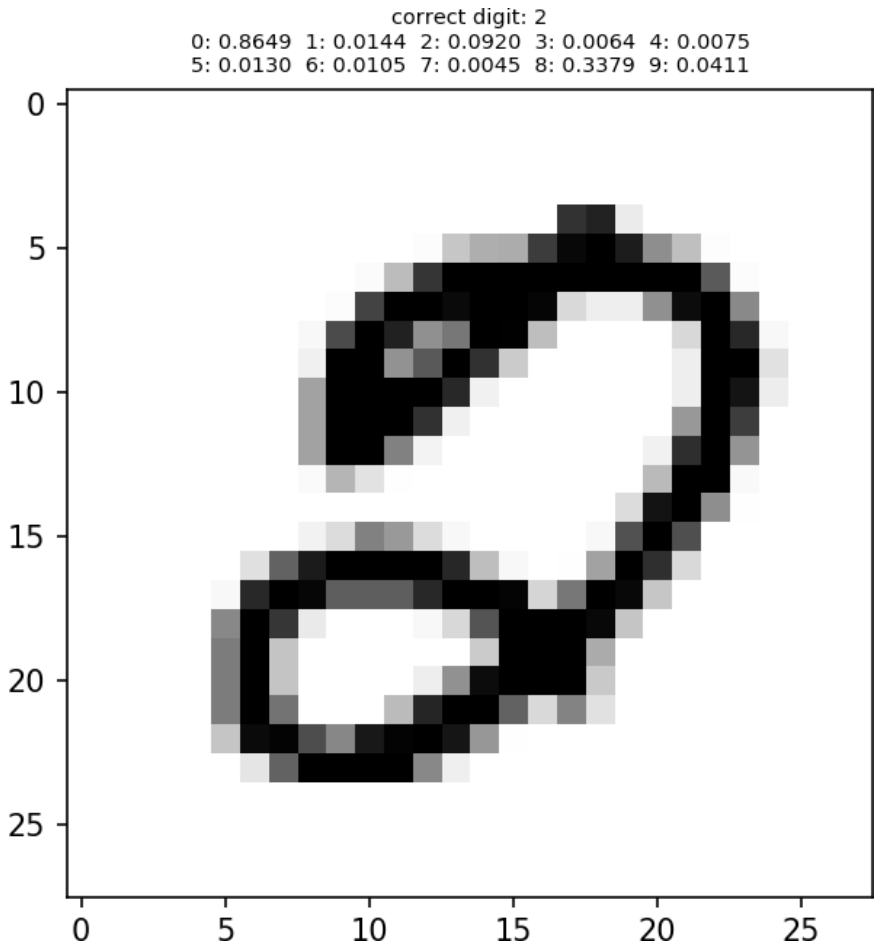
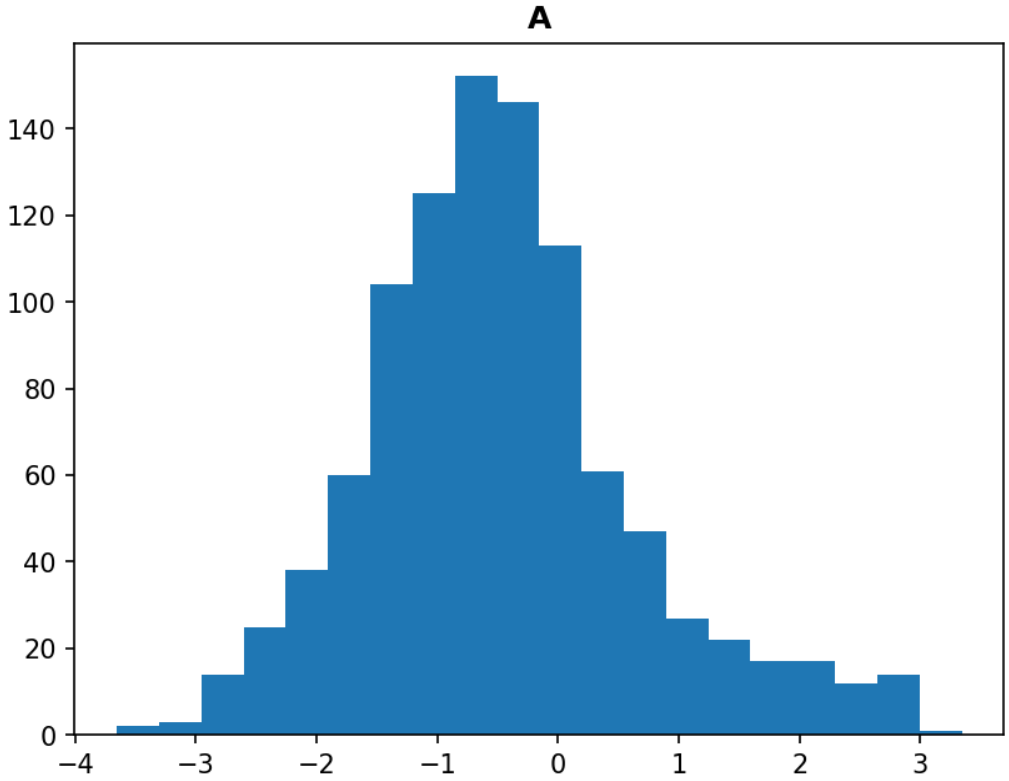
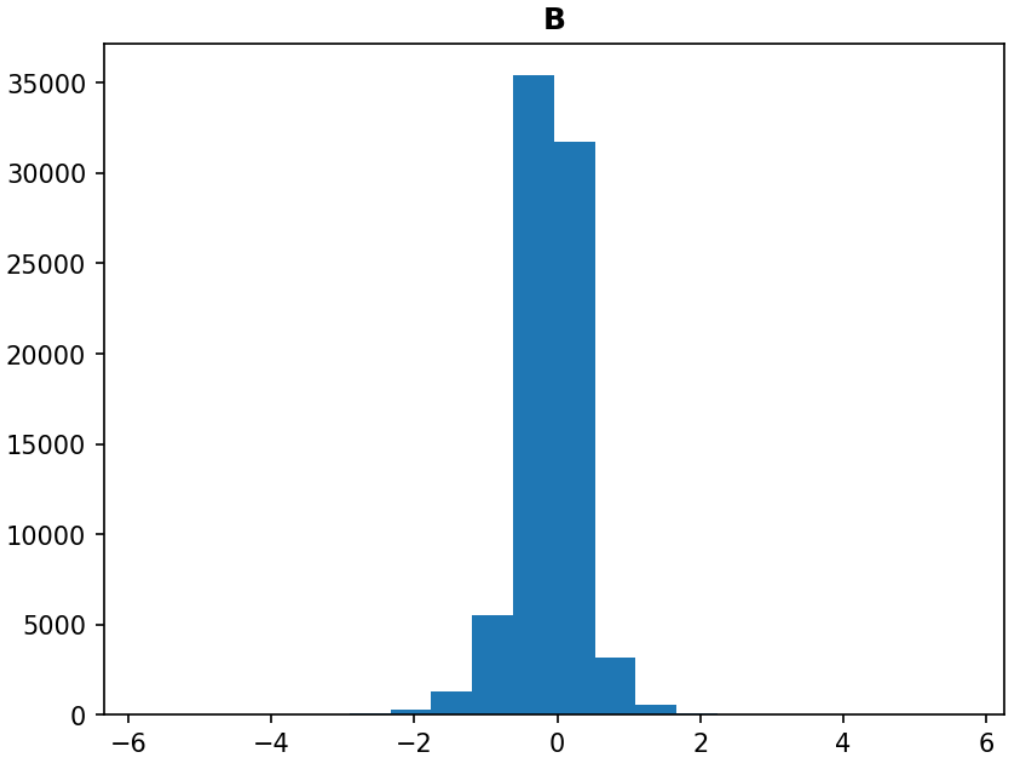


Image Classification Weights

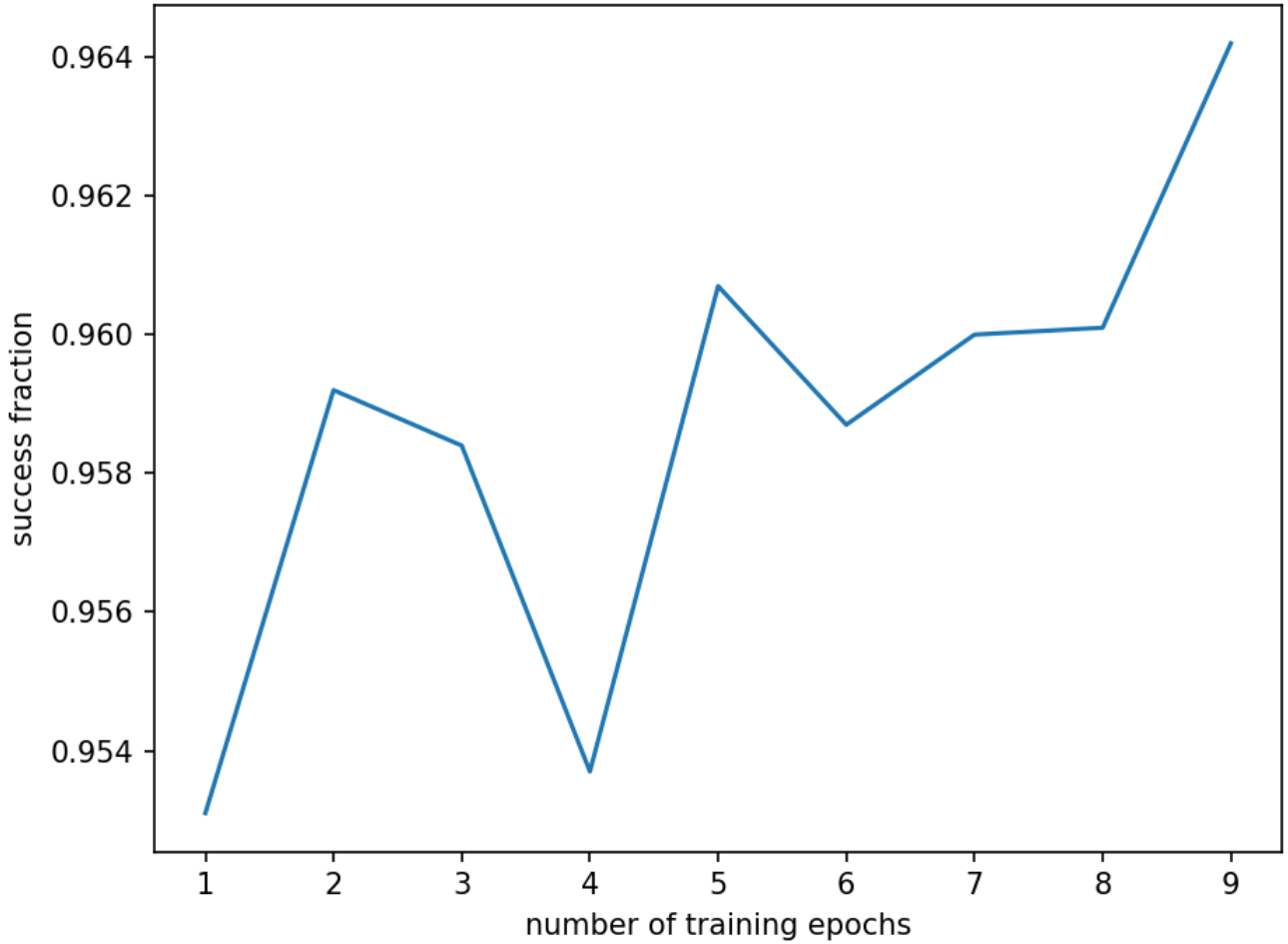
- Weights (matrix elements of **A** and **B**) seem symmetric about 0
  - Interestingly, with more training, the width of the distribution seems to grow





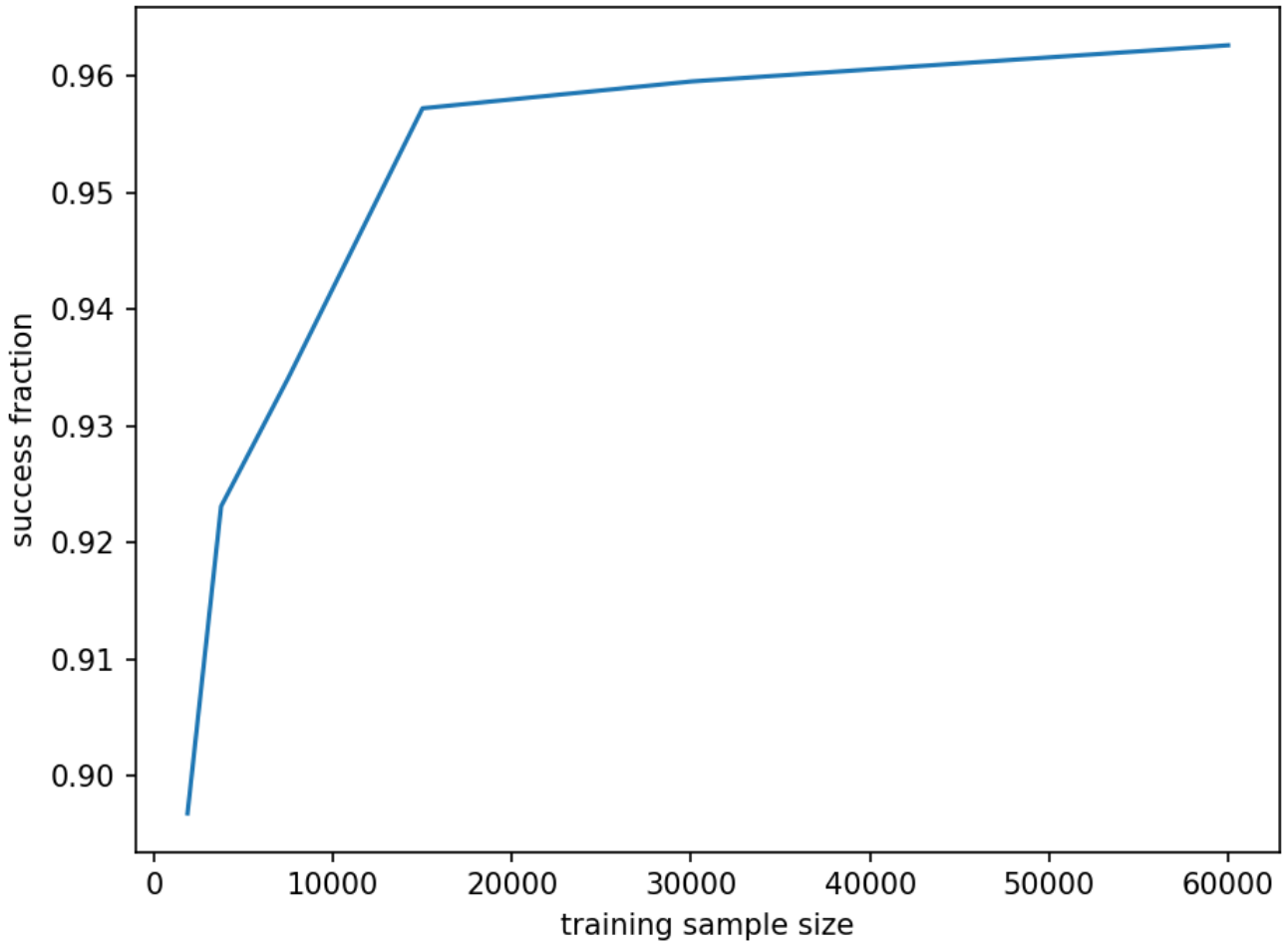
Effect of Number of Epochs

- When we use the full training set (60000 images) the number of epochs (passes through the training data) doesn't seem to matter much



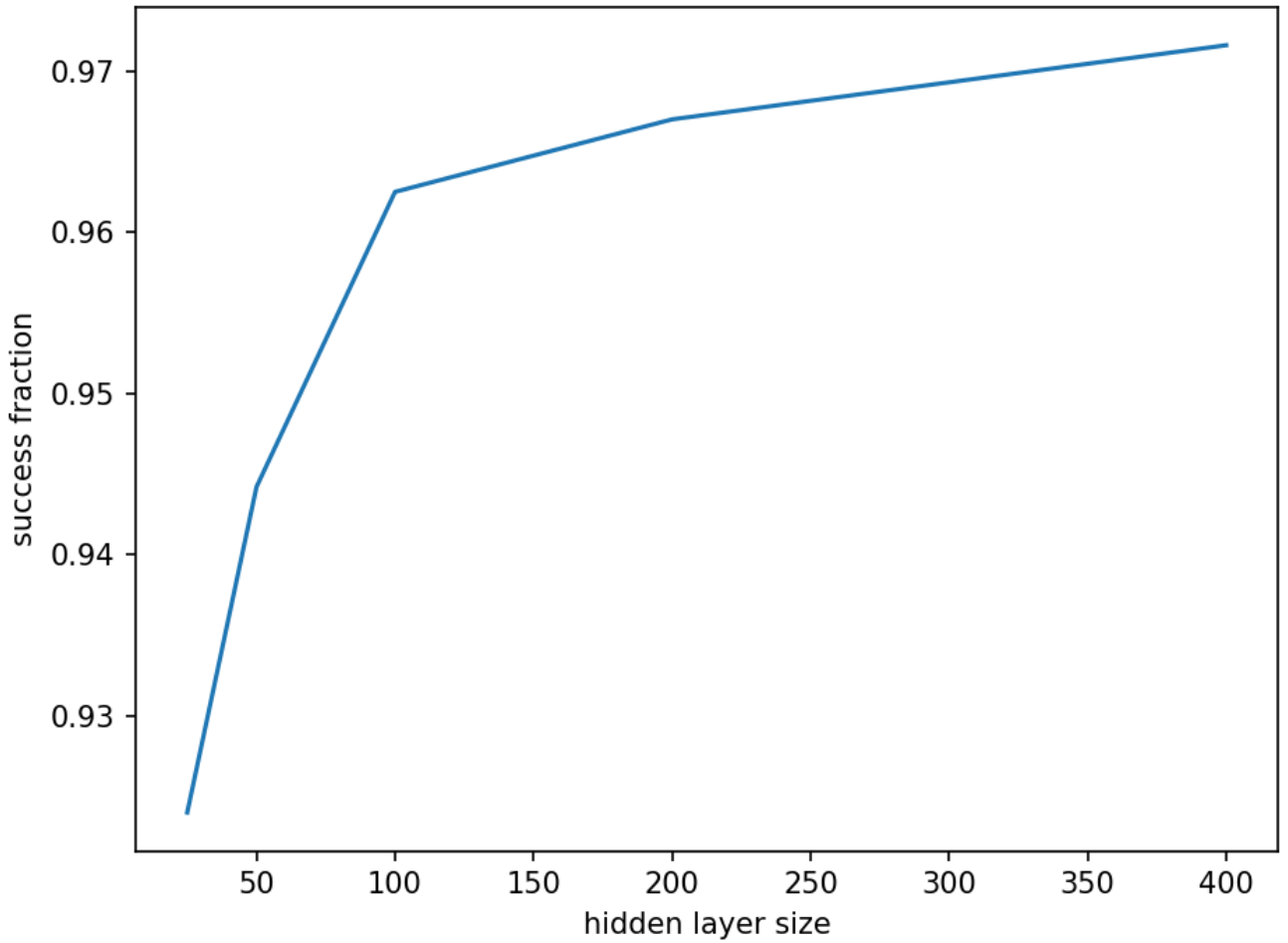
Effect of Training Set Size

- No surprise: the larger the training set, the better we do



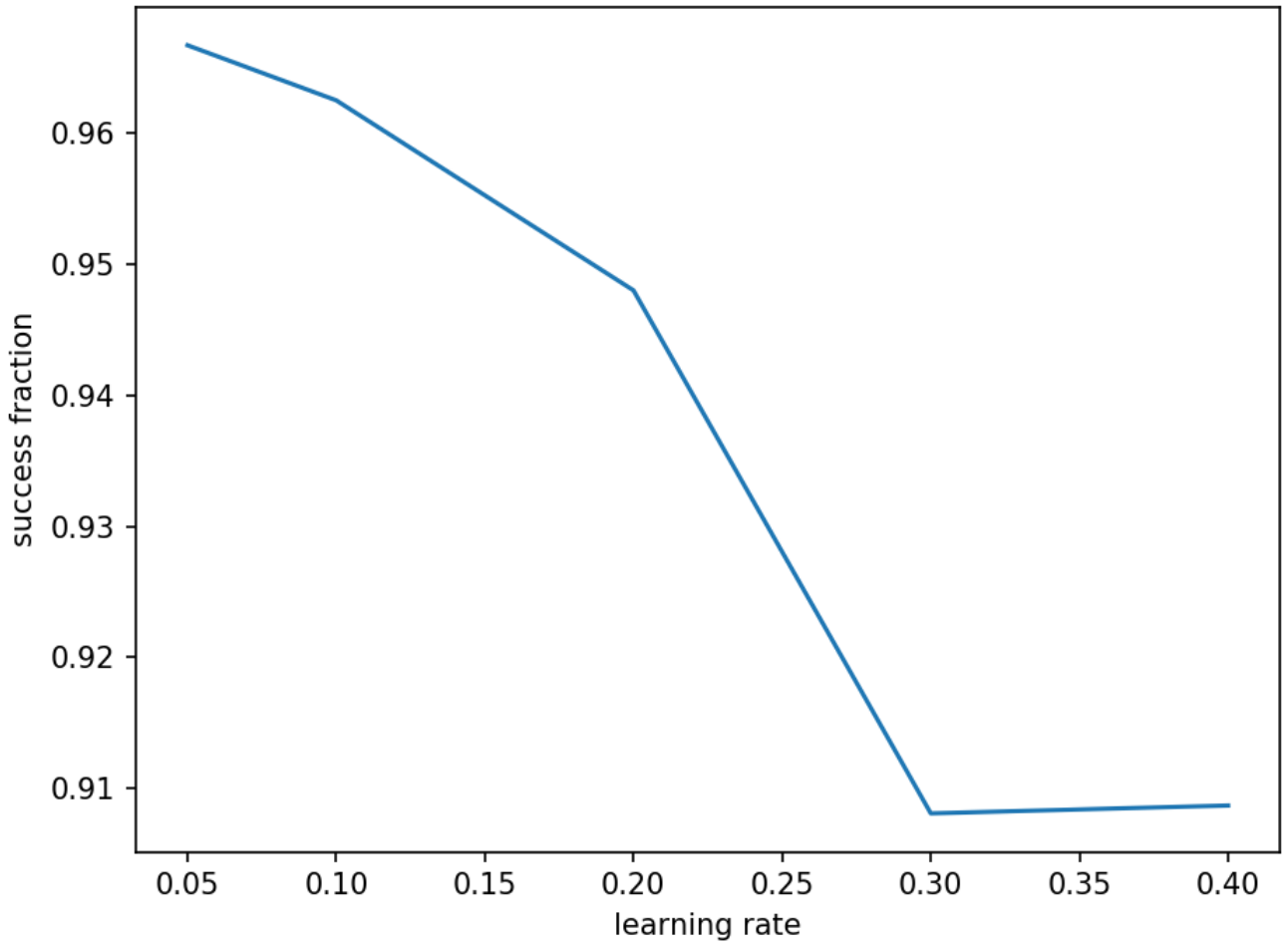
Effect of Hidden Layer Size

- Also not unexpected: the larger the hidden layer the better we do



Effect of Learning Rate

- A smaller learning rate seems to do better



CONVOLUTIONAL NN

- A convolutional-neural-network is a subclass of neural-networks which have at least one convolution layer. They are great for capturing local information (e.g. neighbor pixels in an image or surrounding words in a text) as well as reducing the complexity of the model (faster training, needs fewer samples, reduces the chance of overfitting).
- The main difference between CNNs and multilayer perceptron (MLPs) is that MLPs require vectors as inputs and when we have to deal with images, we flatten he image pixels into a 1D vector and then pass it. What we don't realize is that during flattening the image, we lose some valuable information about the image which later would have been useful in classifying the inputs. On the other hand, CNNs do not require vectors as inputs, rather they accept whole matrices. Hence why, we can input a picture, because after all the image is just a 2D array of its RGB values and the fact that it inputs a whole matrix can in itself become the prime reason to choose CNNs over MLPs while having image inputs.