

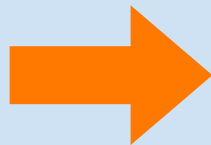
MNIST Dataset: Classification of Handwritten Numbers

Johnny Vo, Emma Gomez, Naomi Castro
CIC-PCUBED 2023
Faculty mentor: Dr. Doina Bein

Introduction

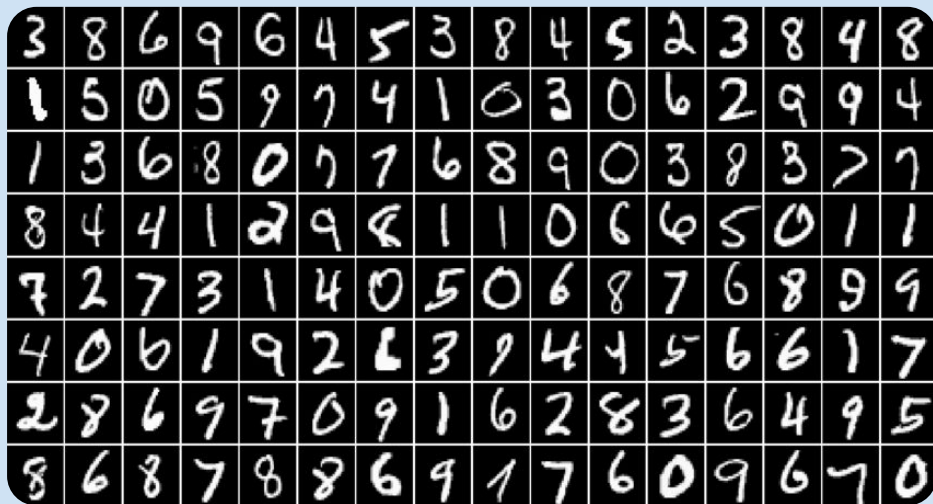
Problem

**How to train a
computer to see
and identify 10
different
handwritten digits?**



Approach

**Develop a
Convolutional
Neural Network to
perform image
processing**



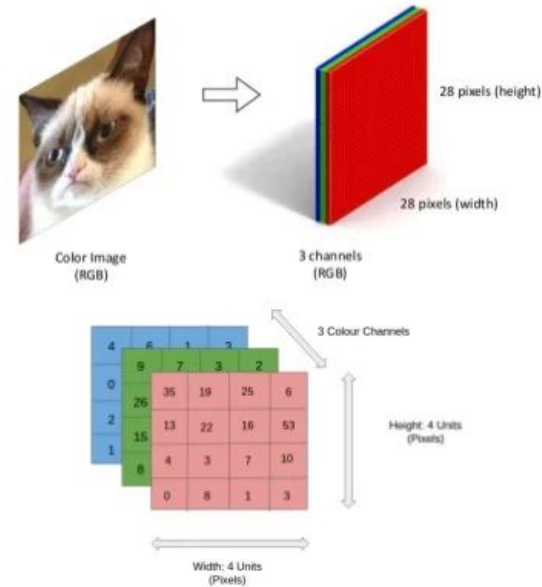
MNIST (Modified National Institute of Standards and Technology) Database

- A dataset of thousands of handwritten digits.
- 60,000 training digits and 10,000 testing digits.
- A centered, grayscale digit and their corresponding labels.

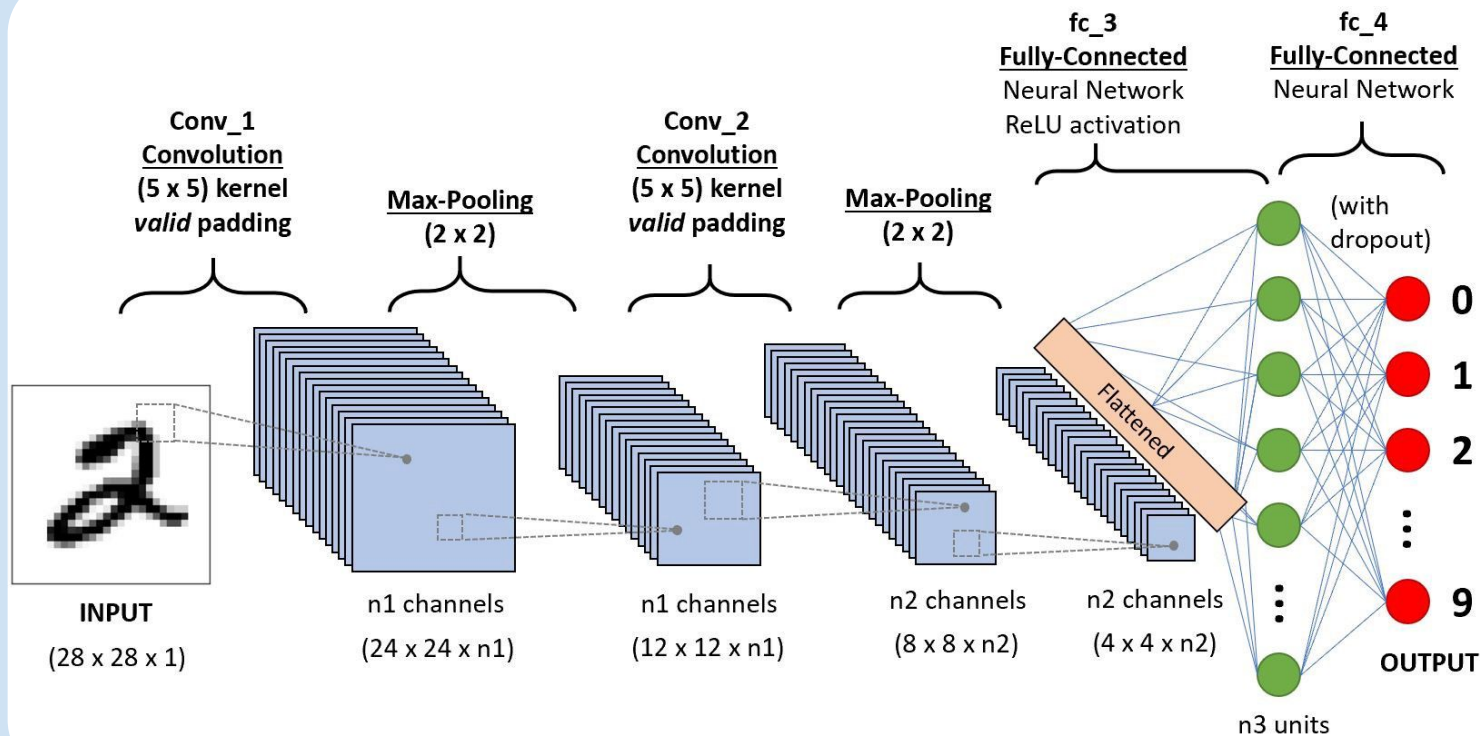
Neural Network Layout

- **Convolutional Neural Network**
- **Nature of mathematical convolution works with image processing and computer vision**
- **Image is a tensor where each element is a value that represents the amount of RGB or grayscale in each pixel**

color image is 3rd-order tensor



Neural Network Layout Example



Data Preparation

There were a few things we found that we had to do before building our model to make sure that everything would be compatible ● ● ●

The first thing we had to do was load the data and split it into training and testing groups, making sure that the images and the labels were in separate variables for each group.

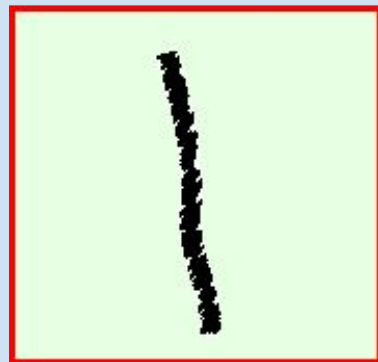
0	8	7	6	4	6	9	7	2	1	5	1	4	
0	1	2	3	4	4	6	2	9	3	0	1	2	3
0	1	2	3	4	5	6	7	0	1	2	3	4	5
7	4	2	0	9	1	2	8	9	1	4	0	9	5
0	2	7	8	4	8	0	7	7	1	1	2	9	3
5	3	9	4	2	7	2	3	8	1	2	9	8	8
2	9	1	6	0	1	7	1	1	0	3	4	2	6
7	7	6	3	6	7	4	2	7	4	9	1	0	6
2	4	1	8	3	5	5	5	3	5	9	7	4	8

0	8	7	6	4	6	
0	1	2	3	4	4	6
0	1	2	3	4	5	6
7	4	2	0	9	1	2
0	2	7	8	4	8	0
5	3	9	4	2	7	2
2	9	1	6	0	1	7
7	7	6	3	6	7	4
2	4	1	8	3	5	5

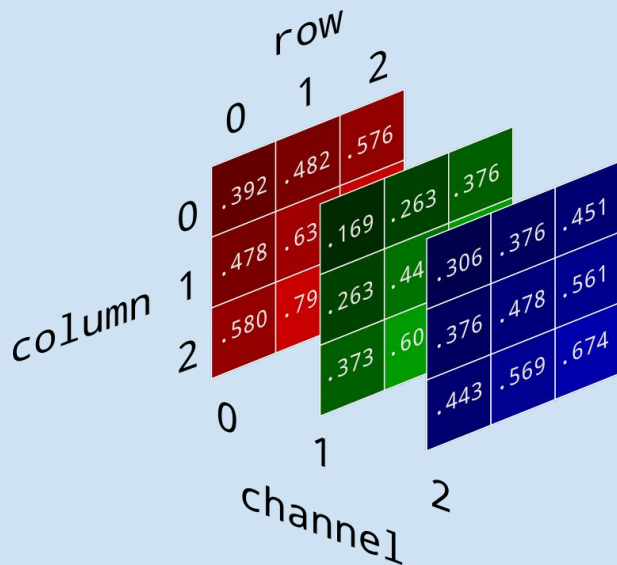
9	7	2	1	5	1	4
2	9	3	0	1	2	3
7	0	1	2	3	4	5
8	9	1	4	0	9	5
7	7	1	1	2	9	3
3	8	1	2	9	8	8
1	1	0	3	4	2	6
2	7	4	9	1	0	6
5	3	5	9	7	4	8

Data Preparation

- **Normalize the images.**
- **Divide each pixel in the image by 255 (so value range is from 0.0 to 1.0).**
- **Model will deal with smaller numbers**
- **Mitigates the possibility of an “exploding gradient”.**



Data Preparation



- **Reshape each image so that the layers of the network could understand its dimensions**
- **Each image in database made up of 28x28 pixels, grayscale**
- **Grayscale → 2-D tensor, shape: (28, 28)**
- **RGB image → 3-D, shape: (28, 28, 3)**
- **Need to explicitly tell the model that there is only one color value it needs to work with (gray)**
- **Reshape the training tensor to be (60000, 28, 28, 1), 60,000 is number of images in the the training group.**

Data Preparation

Finally, we one-hot encoded the validation data—the digit labels. This process essentially creates binary vectors to represent each digit from 0 to 9.

	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

Digit Class

One hot encoded vector

Layers

We used a combination of the following layers:

 **Convolution**

 **Dense**

 **Max Pooling**

 **Dropout**

 **Flatten**

And a few different activation functions:

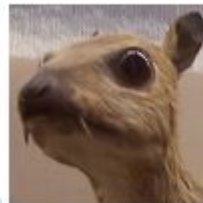
 **ReLU**

 **Softmax**

Convolution

- **Each sample of pixels multiplied by filter**
- **Filter → tensor of weights**
- **Each product saved into feature map**
- **Important in our network, pulls out features of digits computer should focus on**
- **Extracts edges, corners, and different ways the lines bend or slant**
- **Multiplying image pixels values with weights transforms pixels into representing and accentuating edges, corners, and curves**
- **Computer now able to categorize different features, how they relate to certain digits in further layers.**

Input image



*

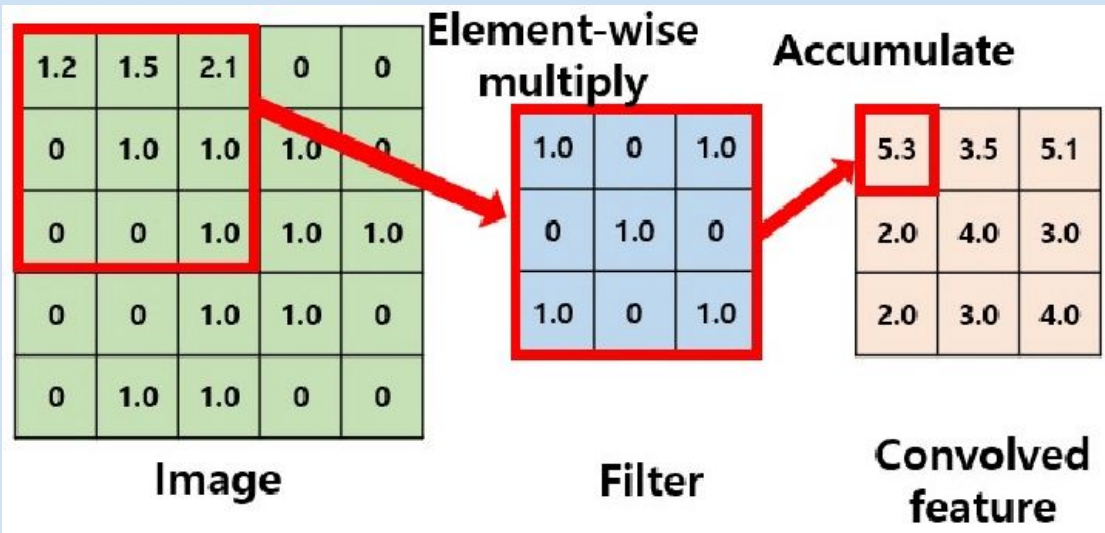
Convolution
Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map



Convolution



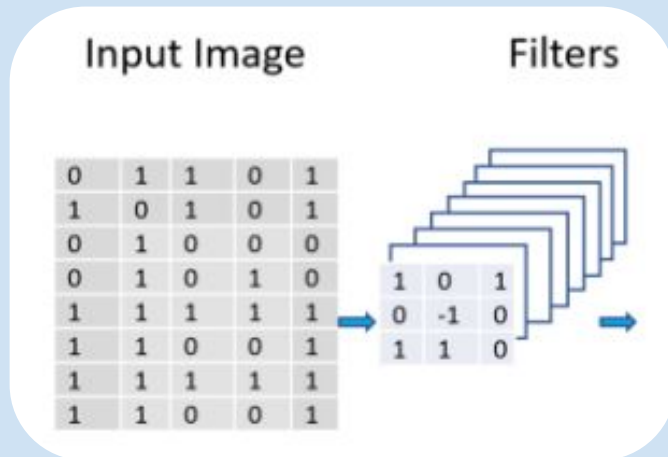
After convolution is computed, the feature map gets passed through a delinearization function

Convolution Hyperparameters

We also had some hyperparameters to consider for these layers

Number of Filters:

- The number of filters used to convolute depends on the amount of pattern combinations we wanted our model to see. The deeper we go into the model, the more filters are used at each convoluting layer. Each layer is able to extract more meaningful data than the last.
- It is also a power of 2 due to convention.
- We chose our first convolution layer to have 32 filters, and the second, 64.



Convolution Hyperparameters

1	2	1
2	4	2
1	2	1

3x3

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

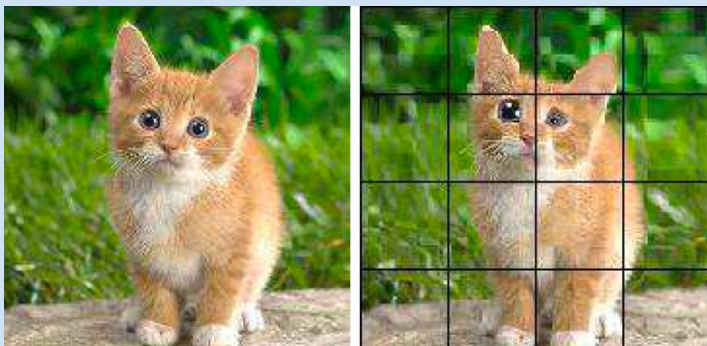
5x5

Size of Filters:

- On the other hand, the size of these filters decreases as you go deeper into the model. In addition, these sizes are usually odd to retain symmetry.
- 5x5 is typically the biggest size you should use if you want to complete training in a reasonable amount of time.
- A 3x3 is then the next smallest. It is able to provide meaningful information (unlike a 1x1 which only looks at one pixel at a time and produces no patterns), and makes cost function minimization simpler for the model due to its low number of weights.
- 5x5 was used in our first layer, and 3x3 was used in the second.

Max Pooling

- Takes the maximum values from each size-specified subsection of the feature map.
- Creates a smaller tensor of those maximum values.
- Suitable for convolution layers by only taking the maximum values from each section.
- Main features of image are highlighted.
- Lessens computational complexity by reducing dimensionality of the tensors.



Max Pooling

12	1	5	1
6	9	0	0
7	5	1	8
0	3	6	5

max pool
2×2 filter
stride 1

12	9	5	1
9	9	8	8
7	6	8	8
3	6	6	5

Deterministic
Downsampling

12	5
7	8

Max Pooling

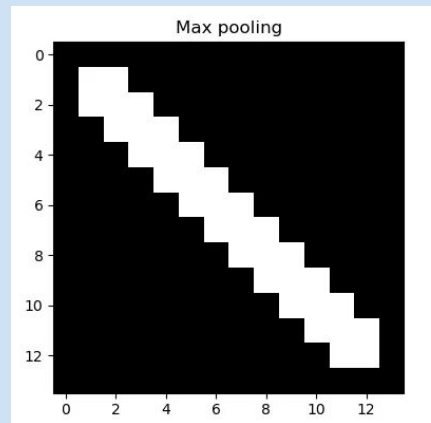
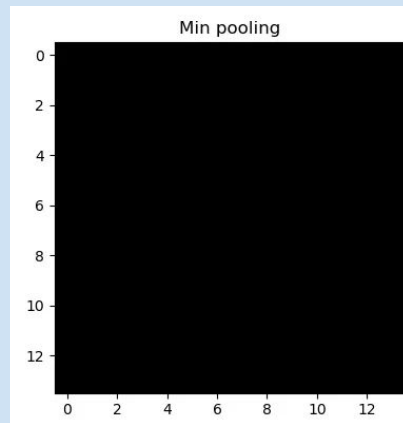
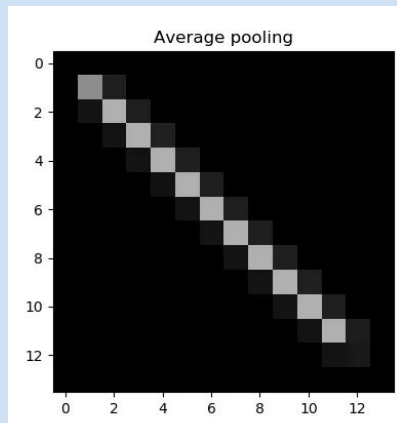
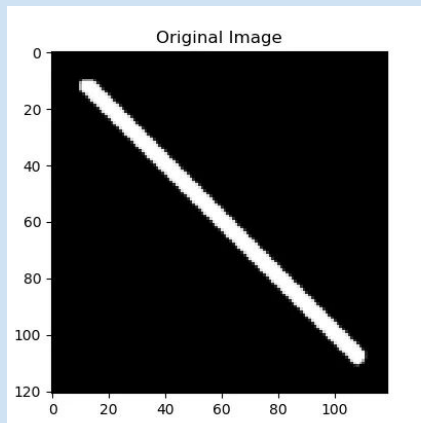
12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

2×2 Max-Pool

20	30
112	37

Max Pooling

- **Dataset made up of images consisting of a black pixels representing the background and a combination of gray and white pixels representing the digit**
- **Use max pooling as opposed to average or min pooling**
- **Pixels with gray or white pixels have higher numerical value than black ones, want model to focus on digit itself, not background**
- **Max pooling picks out the highest value from subsection of feature map tensor**



Flatten

- Takes multidimensional layer (N-D), flattens into single dimension (1-D)
- Takes the last feature map tensor max pooled, flattens into one dimensional array.



Flatten

1	1	0
4	2	1
0	2	1

Pooled Feature Map

Flattening

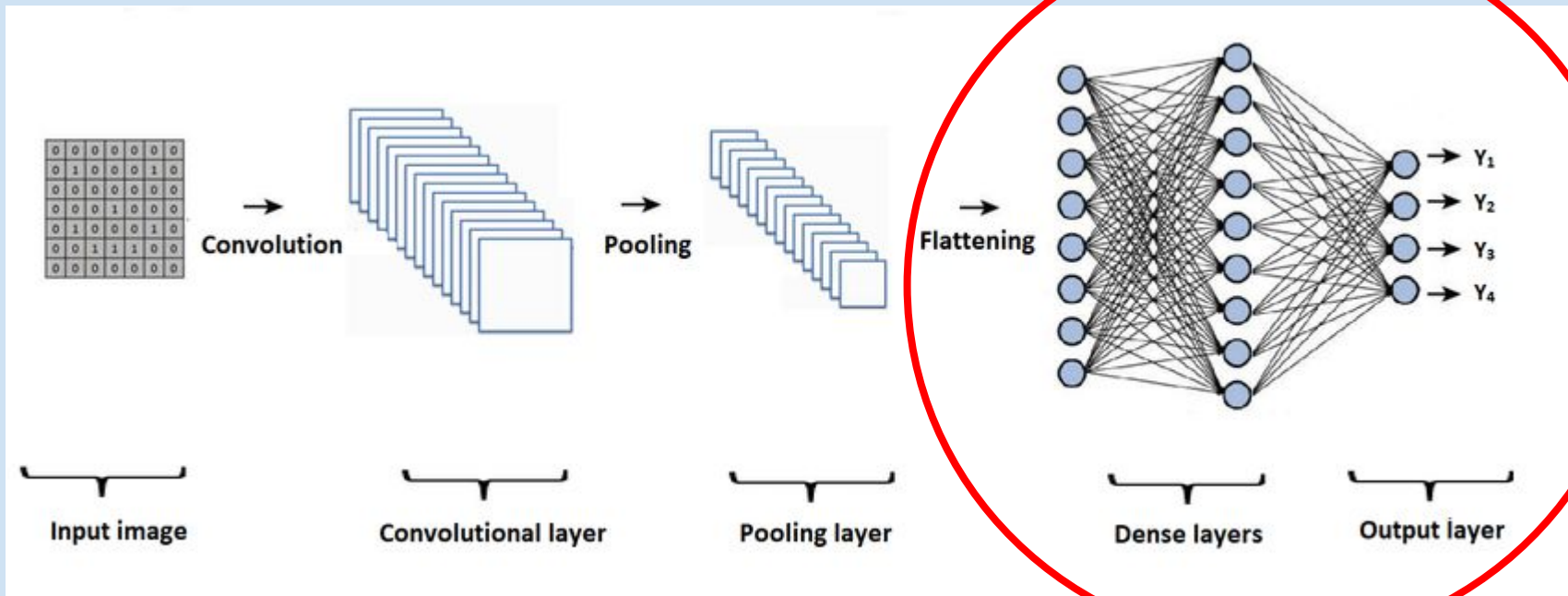


1
1
0
4
2
1
0
2
1

The Dense layers are one dimensional and are what's known as fully or deeply connected layers. In these layers, each node in one layer is connected to every other node in the next layer. The information from one layer is passed to the next by multiplying each node in the former layer by a weight and then adding the products together. This is done n number of times where n is the number of nodes in the latter layer. Each node in the latter layer is then comprised of each sum of products. There is an option to add a number to the product called a bias, but we did not include it. After the products are summed, an activation function is applied to delinearize the value

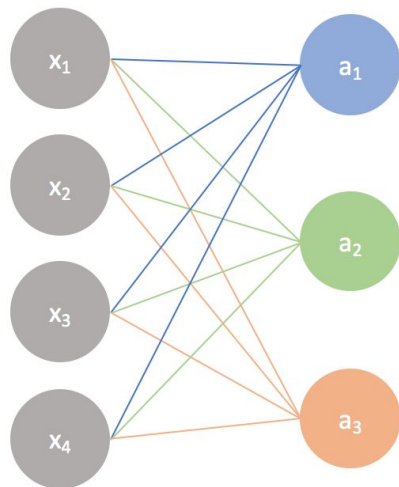
We found that it is important to have Dense layers because they are the ones actually doing the work of classifying the image. They receive the information about the image extracted by the convolution and categorize the features. In addition, for our model to have 10 output nodes that ultimately provide the probabilities that the image is a certain digit, we needed at least one other one dimensional layer before that understands the data from the convolutional layers before it passes its inference into the output.

Dense



Input layer

Output layer

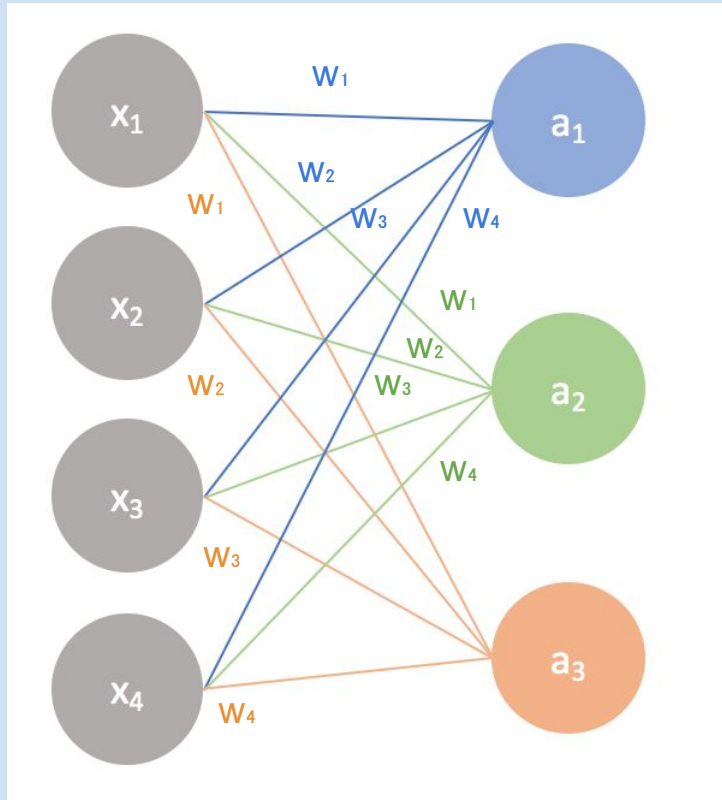


Using multiple observations

weights

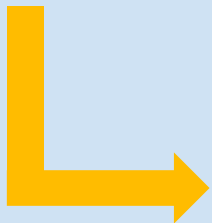
$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} \text{Observation 1} & \text{Observation 2} & \text{Observation 3} & \text{Observation 4} \\ x_1 & x_1 & x_1 & x_1 \\ x_2 & x_2 & x_2 & x_2 \\ x_3 & x_3 & x_3 & x_3 \\ x_4 & x_4 & x_4 & x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} \xrightarrow{\text{activation}} \begin{bmatrix} \text{Observation 1} & \text{Observation 2} & \text{Observation 3} & \text{Observation 4} \\ a_1 & a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 & a_3 \end{bmatrix}$$

Dense



Dropout

General consensus: at least one Dropout layer is important to avoid overfitting.



Co-adaptation: a phenomenon where two or more neurons and their weights compute to be near identical.

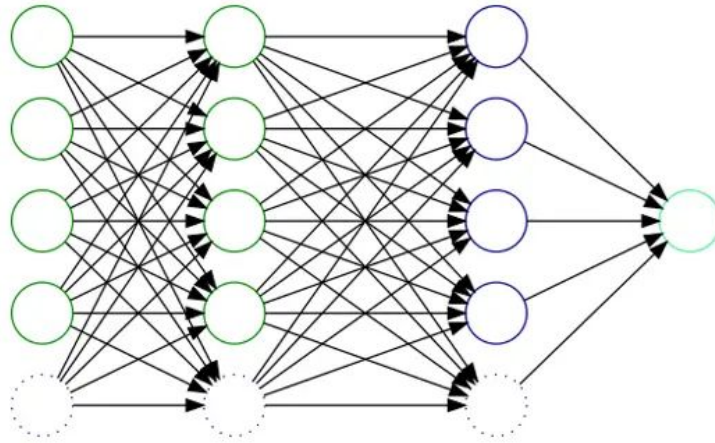
- Likely to happen in a dense layer with many neurons
- Can cause certain patterns

Dropout decreases the chances of this problem by zeroing out the values of randomly chosen neurons at a specified rate.

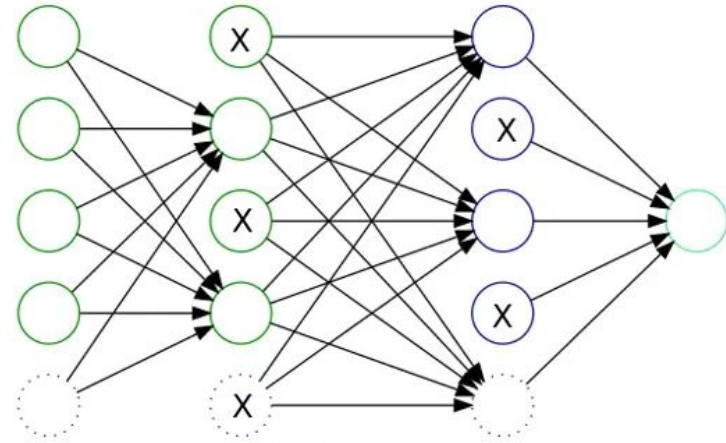
Our dropout rate: 0.5—or 50% of the neurons in that layer.

- Rate yielding the most regulation.

Dropout



Without dropout



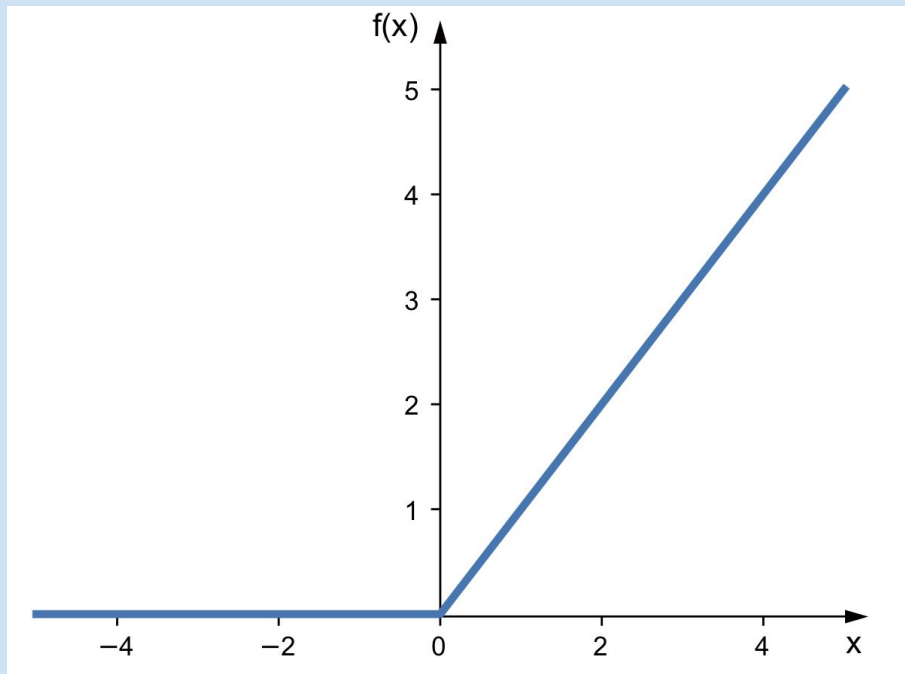
With dropout

ReLU

ReLU is a method of delinearizing that takes in a value and returns 0 if that value is negative or returns the input value if nonnegative. This reduces the nodes to only the ones that contain relevant information about the current digit.

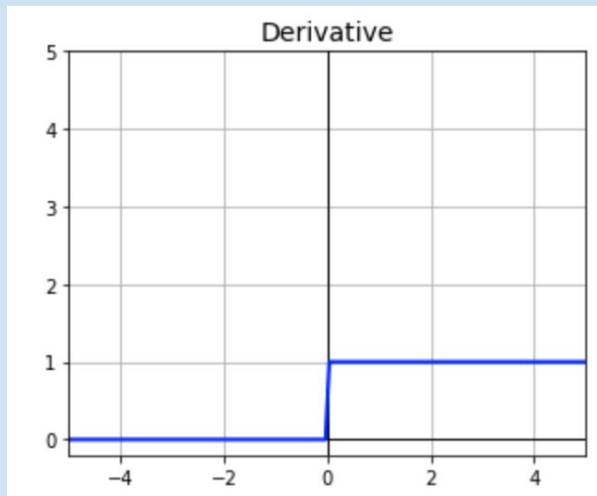
For example, if we gave the machine an 8, the nodes that pertain to straight lines would be either close to zero or zeroed out completely.

There is the issue of dead neurons where the information might be relevant, but we found our model to be very accurate regardless.



Another few reasons we chose to use the ReLU activation function:

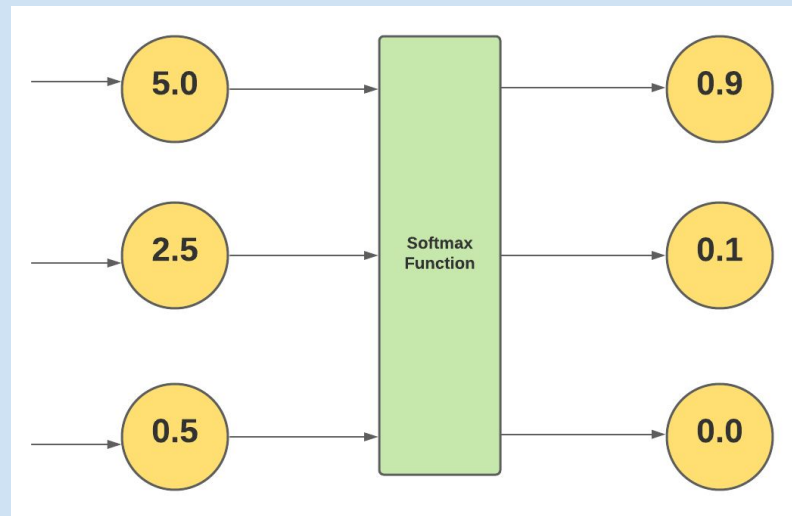
- **ReLU is computationally efficient as it is remarkably simple.**
- **It does not have the problem of the vanishing gradient, as opposed to other activation functions such as tanh or sigmoid. This is because the derivatives of those functions are between 0-1 and 0-0.25 respectively. Whereas the derivative of ReLU function is 0 OR 1, so, during backpropagation, the gradient will never get so small that it can no longer effectively adjust the weights.**



Softmax

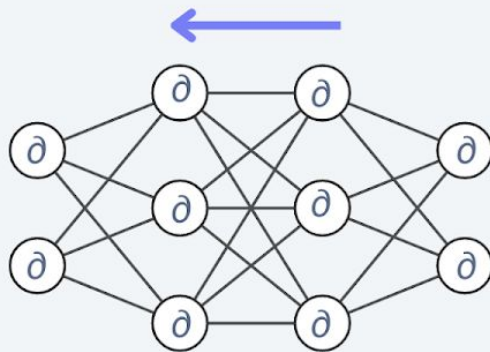
Another delinearization function that turns a vector of values into another vector of values that sum to 1.

- **Creates a distribution of numbers that can be used for probability.**
- **Most useful in the output layer**
- **Each output node represents a fractional number that represents how certain the computer is that the image is or is not that digit.**



Learning Method

In most neural networks, the actual learning is done using **backpropagation**. It uses a cost function to take partial derivatives with respect to the weights at each layer starting with the output and working back. In mathematics, this is called the gradient, and minimizing the function, gradient descent. By utilizing the information it gets from taking the partial derivatives, the model can see which weights are making the most impact. Then, it aims to minimize that function as best it can by adjusting the weights, so long as the gradient continues to be negative. With each weight update, the model is taking a step down the hill to get to a minimum in the cost function. These steps are called epochs.



Learning Method

There are 3 types of gradient descent we looked at while deciding which to use:

Batch

- Backpropagation done for every training example
- desired weight changes averaged to take one step downhill
- Travels down hill most efficiently getting closer to the minimum, takes computer longer amount of time.

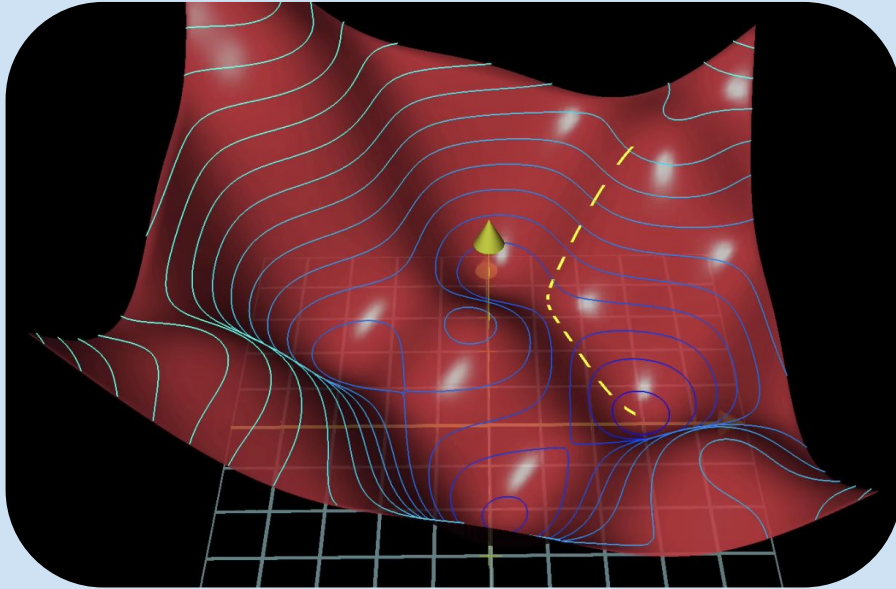
Stochastic

- Backpropagation done at single training examples
- Desired weight changes used to take one step downhill for each example
- Does not travel downhill as efficiently as Batch, reduces time taken for computing.

Mini-Batch


- Backpropagation done for every single training example in small batches
- Desired weight changes from batch averaged to take one step downhill, repeated for each batch
- Does not travel down the hill as efficiently as Batch, but reduces time greatly.

Learning Method




We ultimately decided to use Mini-Batch Gradient Descent due to its time advantages to reduce the load on our machines.


Other Hyperparameters Considered

A yellow starburst icon with six points, positioned to the left of the text box.

Chose to use about 10 epochs so there was enough accuracy being achieved, but not so much as to overfit the model.

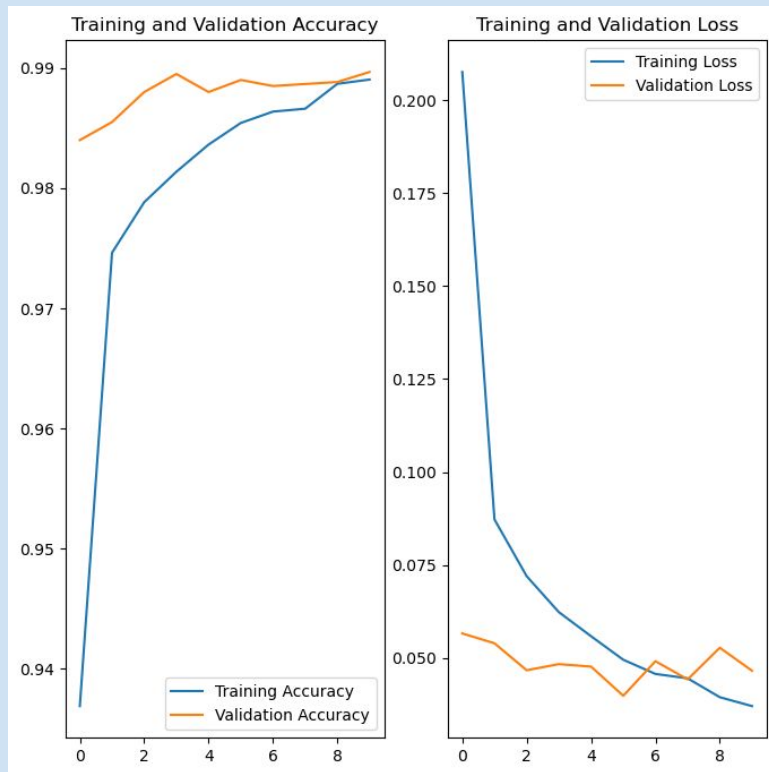
A yellow starburst icon with six points, positioned to the left of the text box.

Categorical Cross-Entropy as our cost function. This function is for multi-class classification. It was able to use probability-based output that the model predicted and essentially compare it to a vector with true probability—where only one output neuron is 1 and the rest are zero. Minimizing (gradient descent) the Cross-Entropy loss aims to shorten the distance between those two probabilities.

A yellow starburst icon with six points, positioned to the left of the text box.

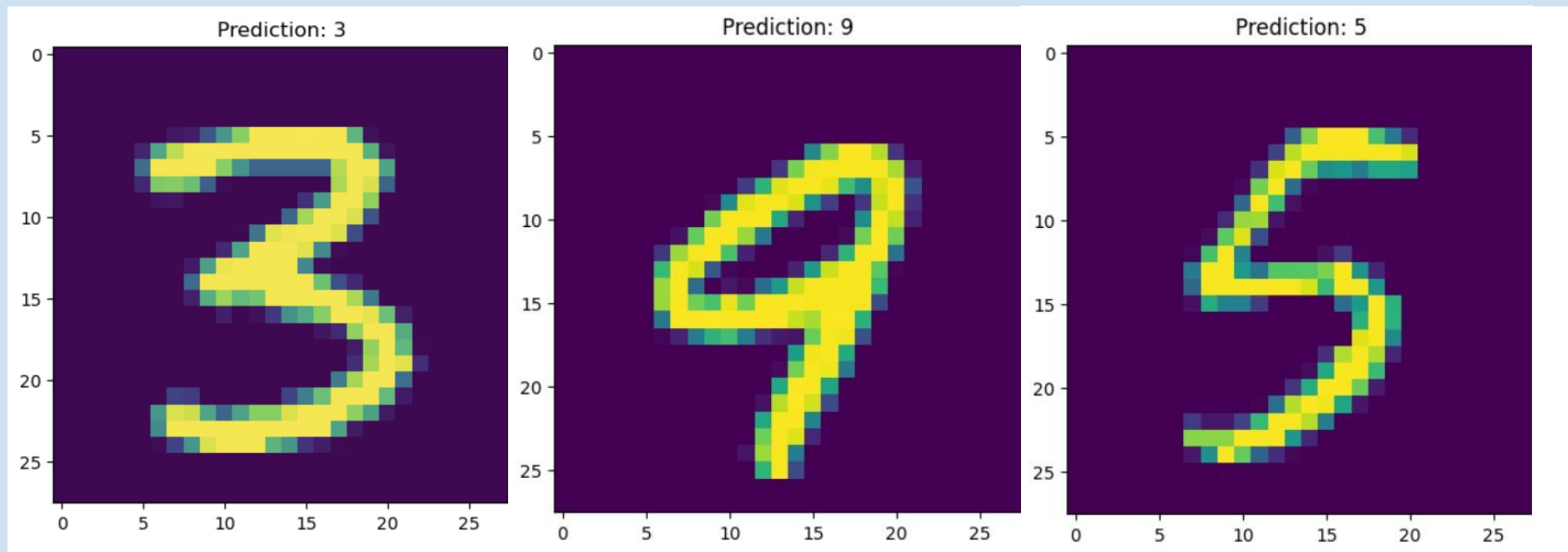
We found that CNN's need at least 3 hidden layers (Convolution, Pooling, and Dense), but that 5-10 was a good rule of thumb.

Results



Our model yielded ~98% accuracy and was able to make correct predictions on random example data

Prediction Examples



References

1. [Neural networks - YouTube](#)
2. [tensorflow.org](#)
3. [medium](#)
4. [towardsdatascience](#)
5. [machinelearningmastery](#)
6. [linkedin](#)
7. [kaggle](#)