2025 – ~~Sudoku~~ Solver Neural Network Project – Full notes

## Puzzle Problem statement

We want a problem statement with the following criteria:

* Should be something for which I can independently generate a training set
* Should be some type of puzzle which is easy to explain and solvable with an algo for performance comparison.
* Should involve a basic neural network, not genetic algos.
* Should be simple/famous enough to explain to an interviewer, eg. Sudoku instead of kakuro
* Should be sensible enough to an expert – eg. Does NN make sense for Sudoku??

### Famous puzzles to try

* Sudoku
* Wordle
* Kakuro – too difficult to model
* Mahjong
* Minesweeper

### Sudoku idea Dead in the Water?? – 21 Oct 2024

Apparently, Neural Networks are not the best way for Sudokus ☹

<https://stackoverflow.com/questions/44397123/neural-network-for-sudoku-solver>

A neural net probably wouldn't work well for a sudoku solver because NNs are best at pattern finding. Framing sudoku as a constraint satisfaction problem would work far better

Neural networks are better for multi-class Classification problems. Not this.

Genetic algorithms are recommended here

<https://github.com/dharm1k987/sudoku-solver-opencv/tree/master/models> This gave me some hope, but this guy is just using the neural network to READ the image, not solve the puzzle…

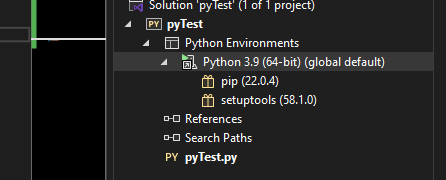
<https://cs230.stanford.edu/files_winter_2018/projects/6939771.pdf> Can we use this paper as a reference?

# Tutorials and 101s

## Setting up python on Visual Studio 2022

Reference Video - <https://www.youtube.com/watch?v=oUwz2mc4BFA>

Adding python dependencies like numpy in VS - <https://learn.microsoft.com/en-us/visualstudio/python/tutorial-working-with-python-in-visual-studio-step-05-installing-packages?view=vs-2022>



In the Visual Studio Solution Explorer, right click on Python 3.9 (or whatever python env you have), click on “Open Command Prompt here”

Enter the following command in the command prompt:

pip install numpy

Restart Visual Studio

# Samson Zhang - Building a neural network FROM SCRATCH…

…no TensorflowPytorch just numpy and math – I like this part

<https://www.youtube.com/watch?v=w8yWXqWQYmU>

better learning when building from scratch…definitely.

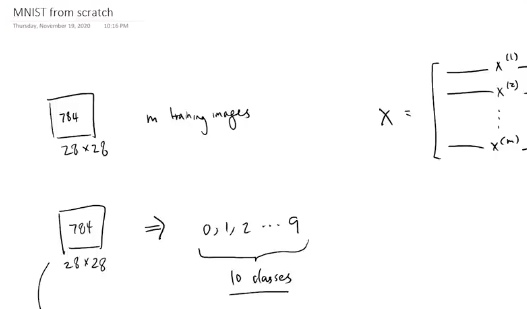
### Problem Statement

Digit classification using the “MNIST Dataset” of handwritten numbers.

Input - 28x28 low res greyscale images.

Output – prediction of written number, from 0-9

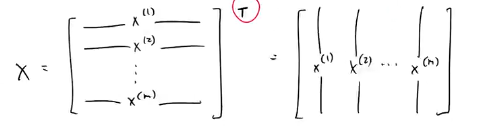
### The underlying Maths



28x28 = 784 pixels in each image – each pixel value ranging from [0,255] – 0 being black, 255 being white <aaah, nostalgia>

#### Representing one input vector (image)

If we have ‘m’ images, then each training example can be represented as:

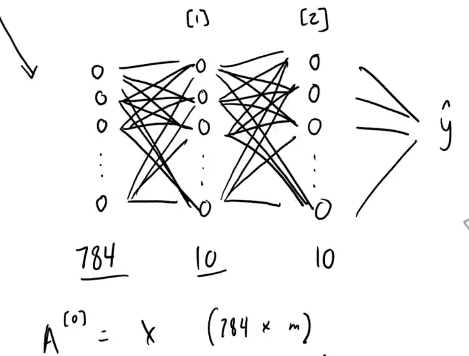


He transposed the [m x 784] matrix on the left.

<Note – he did this because the MNIST dataset **itself** is configured such that each row is a separate training example, and he wanted each column to contain one example.>

He has constructed an [784 x m] vector – 784 rows, m columns, so each column is a separate training example.

#### The network structure



0th layer - 1 input layer of 784 nodes, for 784 inputs in each image.

Layer 1 – hidden – 10 neurons; output layer <Layer 2> of 10 neurons on the right – each neuron in output layer corresponds to one possible digit prediction.

(

he refers to the input layer as the 0th layer because there are no parameters in the input layer; it doesn’t “belong” to the network per se. the 1st hidden layer is where the network starts.

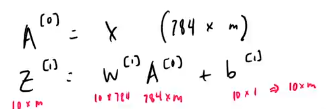
Hence, 2nd layer is the output layer, and we have 1 hidden layer.

)

## Training the network – 3 steps

### 1.Forward propagation

run an image through the network and compute the output.

* 

A0 in the image is the just the input X -> 784 x m

Z1 is the “unactivated” first layer – a [10,784] weight matrix w1 and [10x1] bias b1 will be applied on A[0] input to get Z1 of [10 x m]

Z1 = w1.A0 + b1 <dot product>

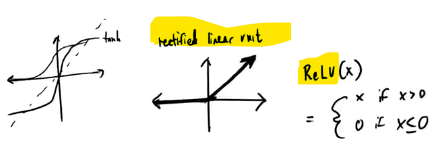
(“unactivated” because activation is not applied yet)

#### Activation

We’ll apply an activation function to each neuron in the first hidden layer



He is using ReLU (Rectified Linear Unit), but tanh(), sigmoid() can also work.



Without the activation, the 2nd layer is merely a linear combination of the 1st hidden layer, which in turn is a linear combination of the input layer – so you may as well not have the 1st layer.

What you end up doing is just linear regression with extra steps.

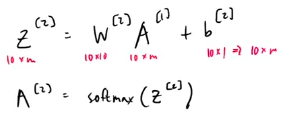
The activation function makes it a non-linear combination, rather than just a linear model. Allows mapping more complex datasets.

<ReLU is just semi-linear>

A1 is the “activated” output of the first hidden layer:

A1 = ReLU (Z1)

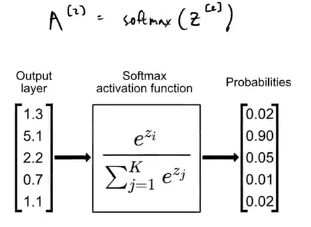
#### Same process for Layer 2



w2is 10x10 (10 rows because current layer has 10 neurons and 10 columns because previous layer 1 also had 10 neurons.) and b2 is 10x1 (bias is associated with the neurons of the current layer only.).

z2 = w2.A1 + b2

#### Softmax Activation Function



We “activate” the output layer using Softmax because we want probabilities from the output layer, corresponding to each of the possible outputs. The main benefit of Softmax (as opposed to Z/sum(Z)) is that it incorporates for negative preactivations as well.

Each of the output layer neurons corresponds to a possible prediction set of the network (eg. 0-9 in this case)’ >

We need good weights and biases to make the prediction accurate.

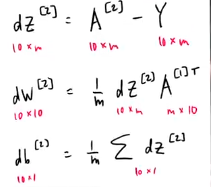
ML will help learn the optimum weights and biases.

(8:00)

## 2.Backward Propagation

During training, Start with prediction and check its “deviation” from the expected output, or “the actual label” – this deviation would make our **loss function**.

BackProp will tell us the contribution of each of the weights and biases to that error. (this is why we check the gradient of the loss function with respect to each weight and bias tensor).



In dZ2 = A2 – Y, he has done a simple difference between the prediction and the intended output. (Andrej had used pytorch.cross\_entropy())

Note – A2 will be of the form [0.0001, 0.03, 0.006…], ie a low/high probability for each of the 10 possible outputs, whereas Y will be a “one-hot” encoded vector – ie a single index will be 1 <corresponding to the correct output>, and the rest will be 0.

So the subtraction is between a probability vector (ranging from 0,1) and a binary vector.

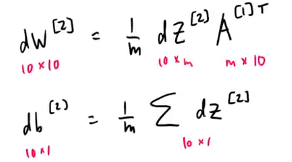
Does dZ2 correspond to the loss function ???

<he has not taken any negative log likelihood to estimate a numeric loss value >

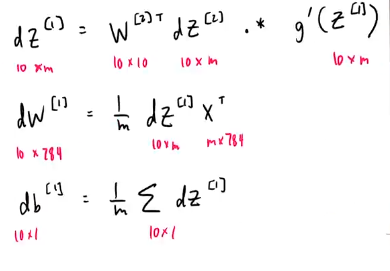
#### Gradient of loss in 2nd /output layer

dw2 is the derivative of the loss function with respect to output layer weights w2.

db2 is the derivative of the loss wrt bias layer 2. He termed it the “average of the absolute error”.

****

#### Gradient of loss in 1st /hidden layer

****

g’(Z) is the derivative of the activation function.

#### My deviation

He summarized these formulae as “fancy maths”, but I would prefer to dig further into how this can be derived…

### IMPORTANT - My Derivation of Backprop - notes for my understanding

Let’s connect this to my Backprop ninja notes from Andrej Karpathy… <https://www.youtube.com/watch?v=q8SA3rM6ckI>

Andrej took a batch size of 32; we can assume that Samson has taken a batch size of ‘m’, since he took the entire training dataset.

Therefore, A2 is a collection of m training examples, not just one…this makes sense since A2 is of size 10 x m.

This means that you should be getting m outputs, not just one.

Z2 is of size 10 x m -> 10 possible outputs, across m training examples. This is the “preactivation”

A2, thus contains the probability of occurrence of each of the 10 possible outputs (from 0-9), for each of the m training inputs. Each input is a 28x28 vector, so has 784 dimensions;

Y is also a one-hot encoded tensor of size 10 x m. there are m columns, each having only one index set to 1, corresponding to the correct (expected) output.

#### Introducing a single loss value for the model

He has taken a numeric subtraction between the 2. However, if you were to calculate the **loss** of the model as one number across the entire training set…

…you’ll take an **average-negative-log** of the predicted probabilities, only on the indices where Y =1, because while training, we’re only interested in the probabilities of the **expected** outputs. Negative log likelihood would give you a reasonable, positive value for the loss.

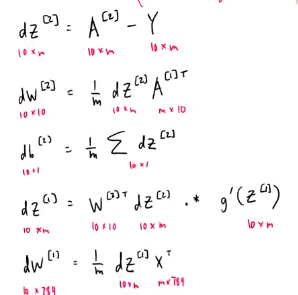
Therefore, in this case, **loss = (1/m) x sum [(-1) x log {A2 where Y=1}]**

Therefore, d(loss)/d(A2) represents the gradient of loss with respect to the Activations A2, and can be represented as:

d(loss)/d(A2) = (1/m) x …

<**this explains where the 1/m came from in Samson’s formulae**…kinda>

#### Diving deeper into the maths



Since his formulae are kiiinda high level, I want to derive them using Andrej’s guidance on loss.

**loss = (1/m) x sum [(-1) x log {A2 where Y=1}]**

in this case, both A2 and Y are of size 10 x m, meaning that each row corresponds to the predicted and actual output of one training examples, something like below:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Z2 (pre-activation) | 0.008 | 1.400 | 0.001 | 0.113 | **5.700** | 0.060 | 0.004 | 0.200 | 0.080 | 0.980 |
| **A2 (prediction)** | 0.001 | 0.164 | 0.000 | 0.013 | **0.667** | 0.007 | 0.000 | 0.023 | 0.009 | 0.115 |
| **Y (Output)** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **{A2 where Y=1}** | 0 | 0 | 0 | 0 | **0.667** | 0 | 0 | 0 | 0 | 0 |

The expression {A2 where Y=1} **cannot be numerically calculated** using matrix multiplication. We’ll have to stick to numpy and python here:



In our case…

**A2y = A2[range(m), Y]**

This is the python code that will give us the datapoint that we want, in a 10 x m matrix.

<I imagine that this is equivalent to the cross\_entropy in pytorch>

Hence,

**loss = (1/m) x sum [(-1) x log(A2y) ]**

say **L** = **log(A2y)**

Hence **d(loss)/d(L) = -1/m**

To transition from a single Loss number to an 10 x m matrix, This is equivalent to having an 10 x m matrix, that looks like below:

**L’ = { -1/m where Y = 1, 0 elsewhere }**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Y (Output)** | 0 | 0 | 0 | 0 | -1/m | 0 | 0 | 0 | 0 | 0 |

d(loss)/d(L) = L’ {L’ is also 10 x m}

Using chain rule, **d(loss)/d(A2y) = d(loss)/d(L) x d(L)/d(A2y)**

= L’ x 1 / A2y

In matrix multiplication terms, we want a [10 x m] matrix here. Therefore, we don’t need to construct L’ explicitly.

Ignoring L’ and replacing it with constant multiplication…

d(loss)/d(A2y) = (-1/m) x A2y.

**<double check matrix division> [A] / [B] = [A].[B\_inv]**

#### Derivatives by Output layer pre-activations

A2 = Softmax(Z2);

Z2 = w2.A1 + b2

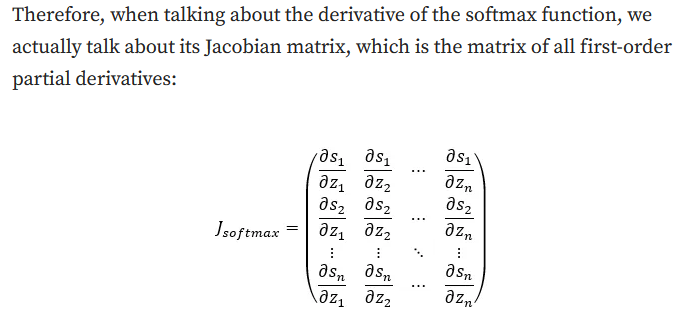
We can check that if y = Softmax (x), then d(y)/d(x) = ???

Someone raise the exact question that I was looking for ☺

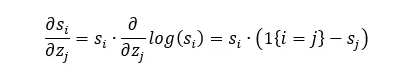
<https://stackoverflow.com/questions/58461808/understanding-backpropagation-with-softmax>

“Softmax accepts a vector as an input and gives a vector as an output, hence it is meaningless to define a "gradient" for softmax. The derivative of softmax is given by its Jacobian Matrix, which is just a neat way of writing all the combinations of derivatives of outputs with respect to all inputs.”

<https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1> also suggests the same.



<no wonder Andrej didn’t bpther with it>



<O…kay? Now what?>

*Assuming softmax derivative to be a pass through for now…sigh…*

Hence,

d(loss)/d(Z2) = **d(loss)/d(A2) x d(A2)/d(Z2)**

assuming that d(A2)/d(Z2) = 1

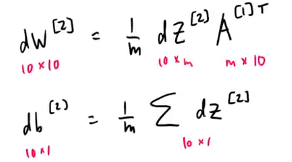
d(loss)/d(Z2) = d(loss)/d(A2y) = (-1/m) x A2y.

Now if Z2 = w2.A1 + b2 (A1 is the activation from the hidden layer.

Then d(loss)/d(w2) = **dW2 = d(loss)/d(Z2) x d(Z2)/d(w2)** **= (-1/m) x A2y x A1**

Similarly d(loss)/d(A1) = **dA1 = d(loss)/d(Z2) x d(Z2)/d(A1)** **= (-1/m) x A2y x w2**

And d(loss)/d(b2) = db2 = dZ2 anyway = (**-1/m) x A2y**

****

Structurally, my derivation is almost identical to his…the only nuance to point out is the fact that my (-1/m) component emerged from dZ2 itself, and his didn’t…and my expression is negative, because I have computed loss value based on negative log likelihood.

<so while I can stick with his expressions, I would love to try mine as well, especially the negative part.>

It could be entirely possible that we arrive at the same result, because he used a simple difference between prediction A2 and expectation Y, whereas I am using a negative log likelihood loss value.

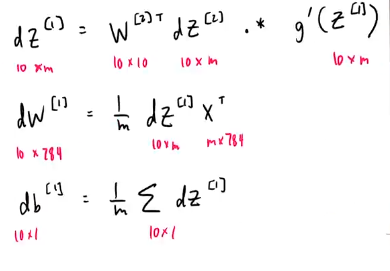
#### Derivations for Hidden Layer

A1 = g(Z1) = ReLU(Z1)

So A1 = Z1 if Z1 > 0 and A1 = 0 if Z1 <=0

Hence d(A1)/d(Z1) = g’(Z1) that he mentioned in his formulae, can be represented as follows:

g’(Z1) = 1 if Z1 >0, else 0

****

Therefore,

D(loss)/d(Z1) = **dZ1** = d(loss)/d(A1) x d(A1)/d(Z1)

**= (-1/m) x A2y x w2** x g’(Z)

(we don’t know what ReLU derivative will turn out to be.)

(in his case, he has taken W2 transpose. Not sure how I can derive this. Would be difficult to ascertain in this case since W2 is 10x10 anyway.)

Z1 = w1. X + b1

Therefore,

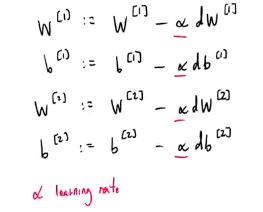
D(loss)/d(w1) = dW1 = **dZ1. X** (need to double check that I get the transpose right. The variable and the gradient need to have the same dimensions.)

And db1 = dZ1

So basically my derivations seems to be on similar lines as his, at least, if not identical…I am reasonably confident that using Andrej’s loss value optimization will give me a reasonably correct optimization, unless I mess up the matrix multiplication.

## 3. Update Parameters

(10:00)



Alpha (a) is the learning rate – subtract the derivative from the weights and biases. If the weights have to increase, then dW will be negative.

Learning rate is a “hyper-parameter”, meaning that it is not optimized by the model:

* Parameters like weights and biases are trained by the model.
* Hyper-parameters like learning rate, network size are set by us to configure the network

#### <Gradient Descent 101>

Picture the parabola loss = X^2. If you increase x and loss increases, that means that you’re on the RHS of the parabola, and the gradient will be positive. Therefore, while updating parameters, you are subtracting a net positive value from x, to ensure that loss reduces correspondingly.

Similarly, If you increase x and loss decreases, that means that you’re on the LHS of the parabola, and the gradient will be negative. Therefore, in the subtraction step above, you’re subtracting a net negative value from x, leading to a net increase in x, to ensure that loss reduces correspondingly.

The objective is to find a local minima.

Once the weights and biases are updated, we rerun the same loop over and over again. Until we arrive at a point where d(loss)/d(weight) = 0, meaning that there would be no more update, and the loss reaches a local minimum.

This the point where the prediction will be closest to the correct answer.

## The python Code



Numpy is for linear algebra, pandas is for reading data and pyplot is for showing plots/images.

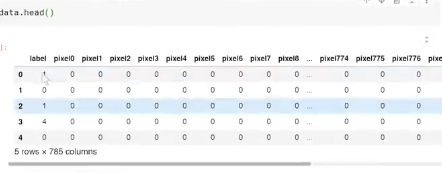
(12:00)

### Loading data



He loaded the MNIST digit data from his online kaggle notebook. We can use offline csvs that we make ourselves.

Data is loaded as a pandas dataframe, on which can call .head() function:



Each row is one training example, with values from pixel 0 to pixel 783 – 784 in total.

We now want to transfer the pandas dataframes to numpy arrays so that we can do linear algebra on them.



(12:30)

Then splitting data into training and test – avoiding overfitting.

Hyperparameters are tested on Test data, Parameters are optimized on Training Data.

#### Basic numpy syntax.

m,n = data.shape #rows, columns + 1 (including the output label; output label is Y, ie the expected output.)

np.random.shuffle(data)



He took the first 1000 examples as developmentva;validation data -

.T is the transpose function, such that each column is now a separate training example. This step is not mandatory, but makes it easier to extract data from this **particular** dataset.

Y\_dev = data\_dev[0} now represents the first column (equivalent to row once the transpose is done) in the dataframe above, ie, the output labels.

X\_dev represents the corresponding input data – from 1:n because the 0th row has the output labels.

First 1000 examples are the dev/validation set. The remaining will be used to train the model:



X\_train[0].shape is (41000,). This is the shape of our first row

This tells us that there are 41000 training examples, since each column is one example, and we are getting 41000 rows.

Shape of our first column can be returned by

X\_train[:,0].shape -- output is (704)

(15:00)

### The neural network code

#### Initializing Parameters – W1, B1, W2, B2

He is defining those in the init\_params() function.

**W1 = np.random.randn(10,784)**



rand() provides random values between 0 and 1. randn() provides values between 0.5 and -0.5.

<He makes an error here of taking randn() and subtracting 0.5 from it, assuming that it’s [0,1] the above initialization is sufficient.>

Similarly, the correct initializations for b1,w2 and b2:

**b1 = np.random.randn(10,1)**

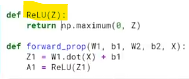
**W2 = np.random.randn(10,10)**

**b1 = np.random.randn(10,1)**



#### Forward prop method

This will calculate Z1 and A1:



np.maximum in ReLU() is an element-wise operation – each element of Z1 is compared to 0;



Then finally-> return Z1, A1, Z2, A2



Replace exp(Z) with np.exp(Z);

np.exp(Z) is also an element wise operation.

np.sum(Z) preserves the number of columns of the tensor, and collapses the rows into the sum.

<https://numpy.org/doc/stable/reference/generated/numpy.sum.html>

the documentation states that I can specify the axis on which I want the sum…so even if I don’t do the transpose that he did earlier to align with the MNIST dataset, I should be fine.

The idea is to ensure that softmax is being summed up across the axis for one training example, because the output probabilities will be on the single example.

(19:00)

### Backprop function

\

Need to one-hot encode Y;



Y.max() will be 9 in the data, so it will create 10 dimensions in each element. There are a total number of Y.size elements in this one\_hot vector.

We set the bit to 1 by arranging Y, but I guess it’s easier to just set the index??

he took a transpose because he wants each example as a column, not a row.…need to ensure that the matrix multiplication checks out.

He implemented backprop as per the formulae he shared earlier

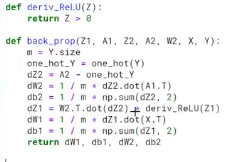
#### Derivative of ReLU()



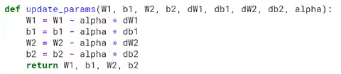
This makes sense since slope is 1 where z > 0 and slope = 0 where z <=0, so a simple binary return will be sufficient.

True = 1 in Python anyway.

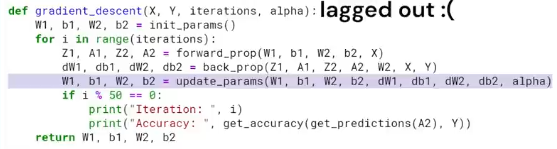
Finally…



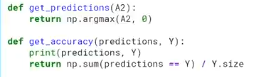
### Update Params



### Gradient Descent function - important



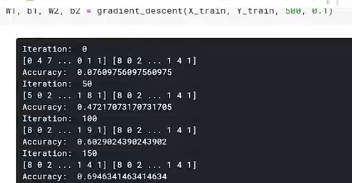
Every 50th iteration, print iteration number and accuracy against the output



IMPORTANT Note – the iterator I is running over the ENTIRE dataset over and over in each iteration i. Andrej’s code was iterating over a different batch in each iteration.



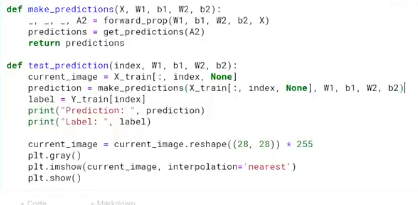
Training…



By iteraton 450, his accuracy was 84%; Not bad.

On test data, his accuracy was 85.5%.

## Making and testing predictions



Other optimizations;

### Instead of gradient descent…

* Gradient descent with momentum
* RMS Prop
* Adam optimization

These are variations of gradient descent.

# Python Lists - Push/Pop 101

<https://www.freecodecamp.org/news/python-pop-how-to-pop-from-a-list-or-an-array-in-python/>

if item in my\_list:

list\_name.pop(index)

# Project Plan

Need to build the following:

* Sudoku generator - populates a 9x9 grid with valid sudoku solutions - uses 9 row lists, 9 column lists and 9 grid lists.
* Sudoku validator - given a 9x9 sudoku grid, is it a valid sudoku?
* Sudoku puzzle maker - given a 9x9 valid sudoku grid, remove x numbers from it to make a puzzle - x can vary based on Easy/Med/Hard
* Sudoku Dataset - csv files that save training and validation data - 90% and 10%
* Neural network Training module - takes valid outputs of generator as outputs and puzzle maker's outputs as inputs.

# CV Points:

- Year when you made this - 2025/26

- Was built **without** Tensorflow or Pytorch.

- Sudoku solver AI based on Neural networks

- How many layers did the network have? Which neural network configuration performed best?

- Performance compared to algorithmic solver

- How many optimizations were tested?