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 – needs an NN for each cell…
* Wordle – complex modeling…maybe later
* Kakuro – too difficult to model
* Mahjong – Turn based
* Minesweeper – Turn based

### 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

#### What if…

What we build an array of small networks that predict one cell value from 0-9 in a 3x3 grid…these can then become inputs to a 3x3 grid of 3x3 grids, ie a Sudoku…

Or we create 81 separate networks, each will be called for a separate cell if that cell is NOT populated;

Thus, the loss value will be an array of 81 numbers for each cell then…(to be initialized as -999 to double check that whether the training data had any cell consistently populated…)

We can then assign the most certain probabilities (anything more than 90%) to the cells and rerun the network…

Essentially each cell will need its own small neural network…since the grid won’t be completely empty, it will involve 81 networks… each one to be called

# Acknowledgements

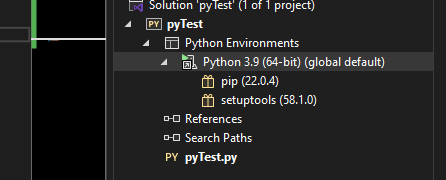
* <https://cs230.stanford.edu/files_winter_2018/projects/6939771.pdf> This paper was used as an inspiration
* <https://medium.com/analytics-vidhya/how-to-solve-sudoku-with-convolutional-neural-networks-cnn-c92be8830c52>
* Implementation guide on the above paper
* <https://www.youtube.com/watch?v=w8yWXqWQYmU>
* Samson Zhang offered motivation to build from scratch
* Andrej Karpathy offered key insights on fundamentals of complex networks and the underlying maths. <https://www.youtube.com/@AndrejKarpathy/videos>
* Andrew Ng series (long ago) helped with foundational understanding of machine learning.

# 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.

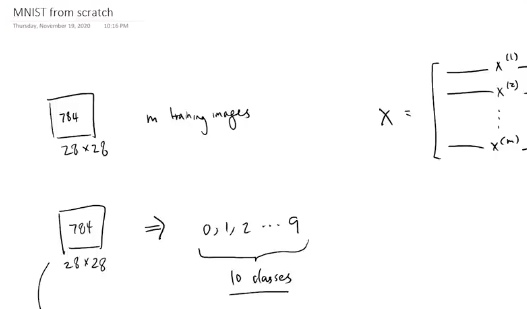
### Problem Statement

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

Input - 28x28 low res greyscale images.

Output – prediction of written number, from 0-9

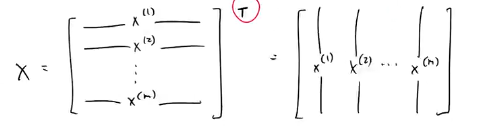
### The underlying Maths



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:

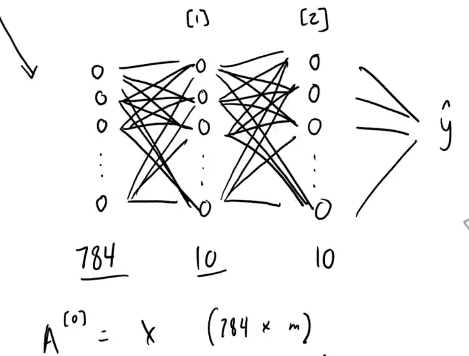


He transposed the [m x 784] matrix on the left.

<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.

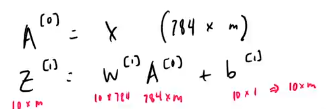
Hence, 2nd layer is the output layer, and we have 1 hidden layer.

)

## 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>

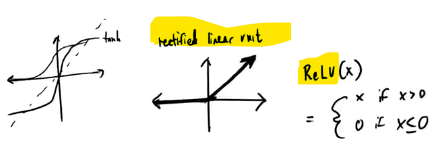
(“unactivated” because activation is not applied yet)

#### 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.

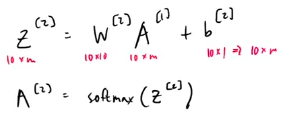
The activation function makes it a non-linear combination, rather than just a linear model. Allows mapping more complex datasets.

<ReLU is just semi-linear>

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

A1 = ReLU (Z1)

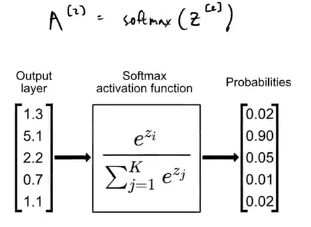
#### Same process for Layer 2



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.

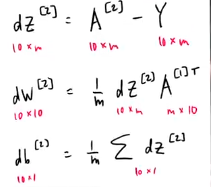
ML will help learn the optimum weights and biases.

(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.

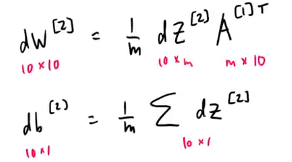
Does dZ2 correspond to the loss function ???

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

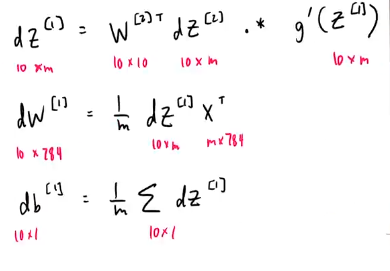
#### Gradient of loss in 2nd /output layer

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.

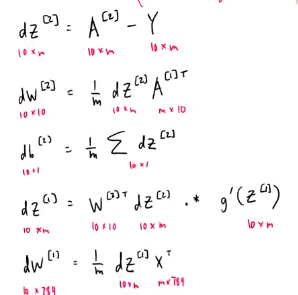
Therefore, in this case, **loss = (1/m) x sum [(-1) x log {A2 where Y=1}]**

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.w

**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(Y.size), 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

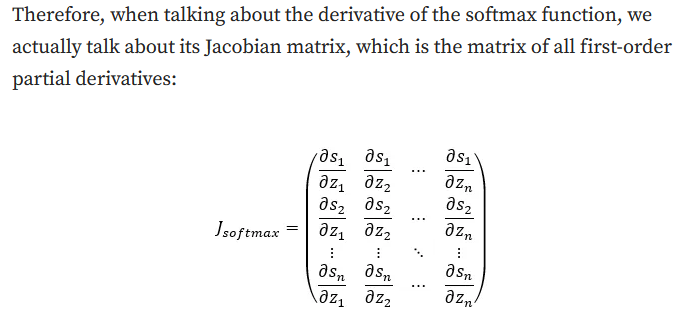
We can check that if y = Softmax (x), then d(y)/d(x) = ???

Someone raise the exact question that I was looking for ☺

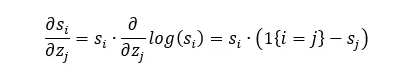
<https://stackoverflow.com/questions/58461808/understanding-backpropagation-with-softmax>

“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*

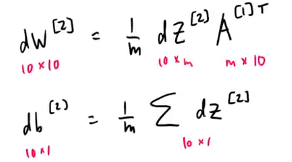
d(loss)/d(Z2) = d(loss)/d(A2y) = (-1/m) x A2y.

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.

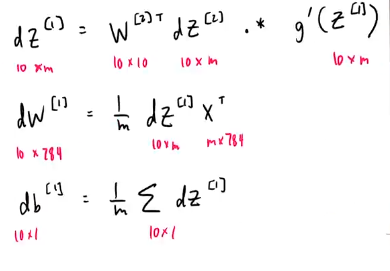
#### Derivations for Hidden Layer

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’(Z1)

(g’(Z) was implemented for ReLU separately. We can stick to tanh() in our code.)

(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,

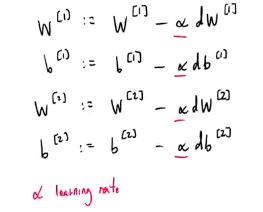
D(loss)/d(w1) = dW1 = **dZ1. X** (need to double check that I get the transpose right. The variable and the gradient need to have the same dimensions.)

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.

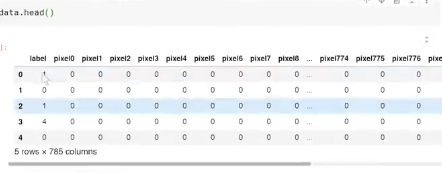
(12:00)

### 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)**

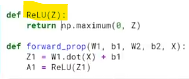
**W2 = np.random.randn(10,10)**

**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

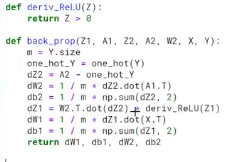
#### Derivative of ReLU()



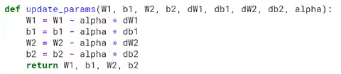
This makes sense since slope is 1 where z > 0 and slope = 0 where z <=0, so a simple binary return will be sufficient.

True = 1 in Python anyway.

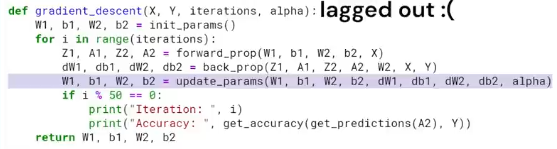
Finally…



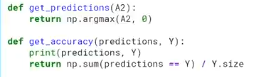
### Update Params



### Gradient Descent function - important



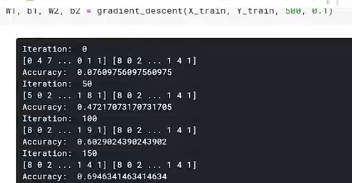
Every 50th iteration, print iteration number and accuracy against the output



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.



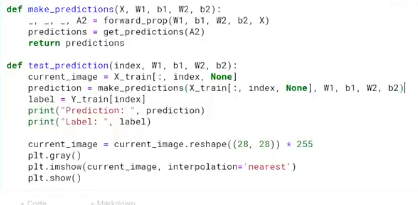
Training…



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…

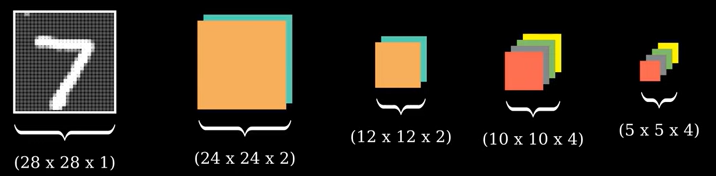
* Gradient descent with momentum
* RMS Prop
* Adam optimization

These are variations of gradient descent.

# Convolutional Neural Nets 101

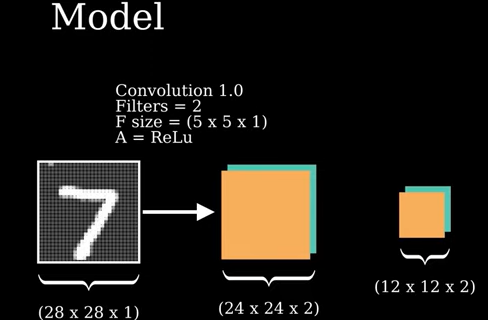
## Visualization

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



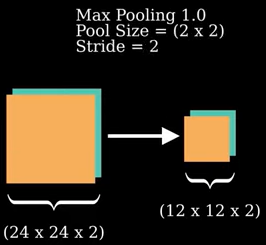
This is the visual representation of the entire model.

<https://www.youtube.com/watch?v=jDe5BAsT2-Y>



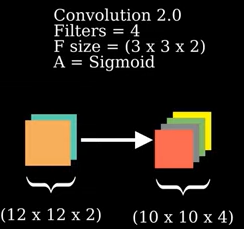
28x28x1 is the dimension of his input – same as Samson’s since he’s also using MNIST.

* This person is going to form a “convolution” using “2 filters or kernels” of filter size 5x5x1, and use ReLU activation function.
* Next step is a “Max pooling” with a “stride” = 2 and a “pool size” of 2x2.



This screenshot represents “the first layer” …

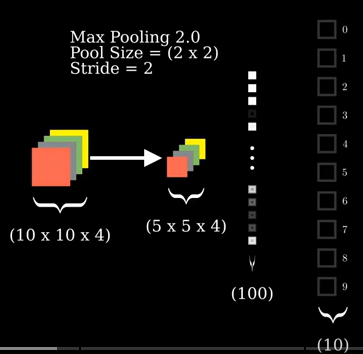
The “2nd convolution” is represented below:



This time, 4 filters will be used <??>

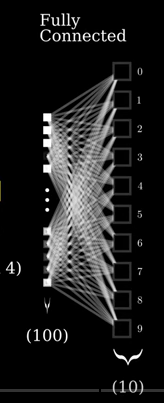
Filter size is 3x3x2. With **sigmoid** activation function…

Maxpooling will be done again, before “flattening the matrices to form a flattened layer of 100 nodes”



This is the 2nd layer

Final step:



The fully connected layer

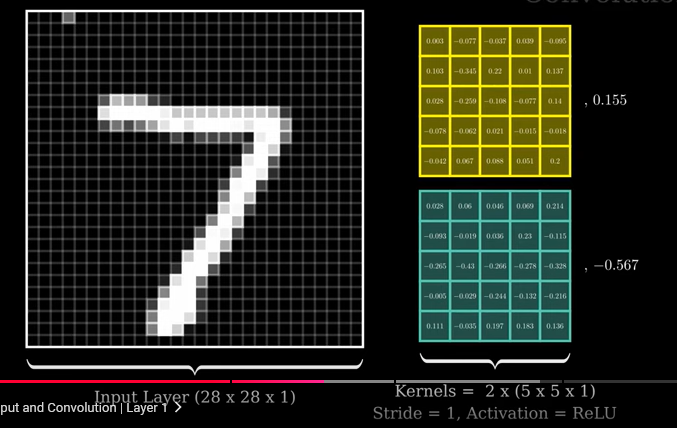
### Explanation

<https://www.youtube.com/watch?v=jDe5BAsT2-Y&t=91s>

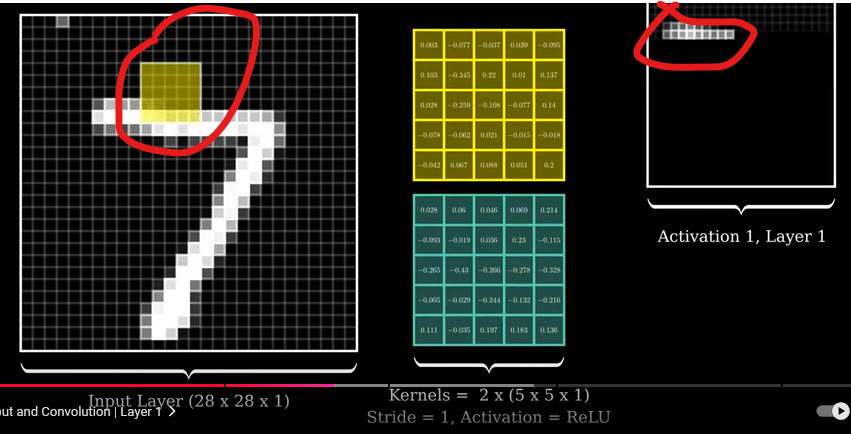
#### Convolution

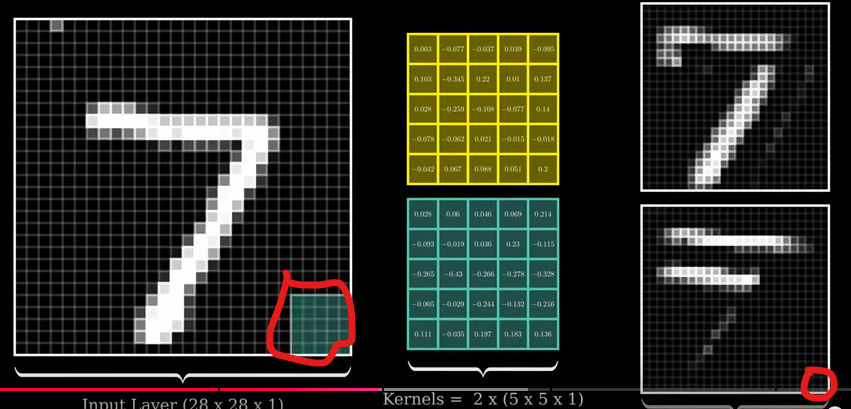
Input layer is 28x28x1 – since we have images of size 28x28 and these are greyscale images, not RGB, hence the color makes the 3rd hyper-dimension;

The first convolutional layer consists of 2 5x5x1 “filters” and add standard biases, one for each filter.



The filter is applied as illustrated below, starting from the top-left to the bottom right, and the activation function value is shown on the right side result.



Note here that the stride of the filters is 2, meaning that they’re applied from pixels 1-5, then 3-7, then 5-9…**then 19-24** and can’t go beyond that.. **that’s why with an input of 28 x 28 x 1 returns an output of 24x24x2**Both filters are applied like so…

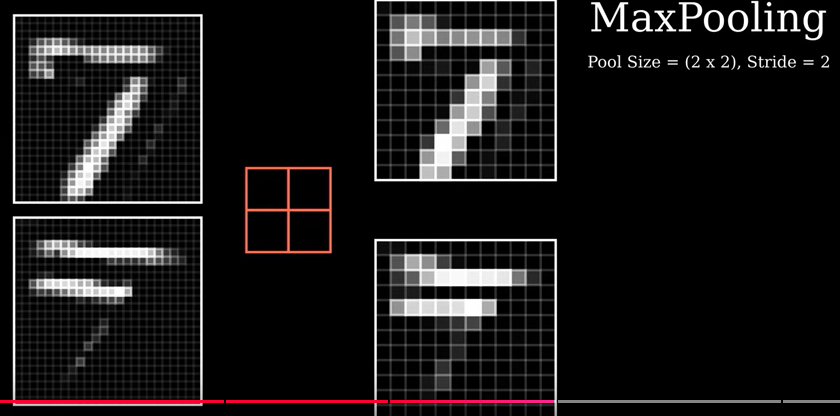
And the first convolution returns 2 different matrix values, shown on the right, since the 2 filters are different…

These 2 are called **channels**.

#### Max-pooling

This is meant to shrink the dimension of the 24x24x2 to 12x12x2 – you run a 2x2 window across the matrix of 24x24, and skip every 2nd pixel when you move the window (stride = 2)

The output will be the highest (“most activated”) pixel value

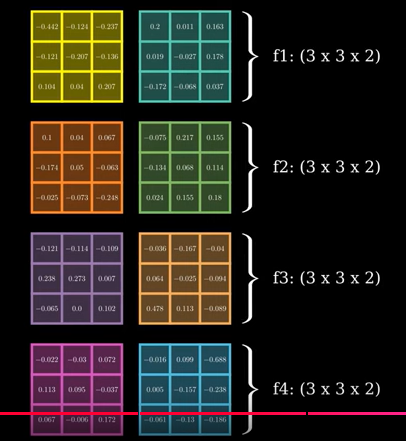


<not fully sure why this is needed for our case at least>

#### 2nd convolution layer..

4 filters, stride = 1, **sigmoid** activation

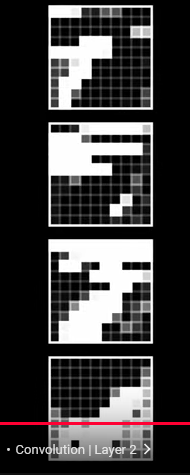
There are 4 filters of size 3x3x2, since we now have 2 channels from the previous activation, and one bias for each filter.



IMPORTANT - Each of the 2 dimensions in each of the 4 filters would have to be summed up together…hence the 8 filters illustrated above will be summed up into 4 outputs.



The convolution will return 4 output channels, hence a 10x10x4 output.



The filters now look nothing like the original image.

Note that This 2nd layer has a sigmoid activation function as well…



Here, for each input pixel, you are doing the following steps (Note that Stride = 1 here):

* Each of the 4 filters has 2 dimensions (3x3x**2**), which will be applied to the 2 channels created.
* The 2 resulting matrices will be added to each other to make one 3x3 matrix.
* Add all elements of this 3x3 matrix – this sum will be associated with the one pixel – THIS is where you illustrate the dependency of one datapoint being spatially dependent on its neighbours.
* Apply sigmoid on this final sum, to have an activation value for each pixel.

#### Max pooling 2

This will work the same way as before, reducing the dimension of 10x10x4 to 5x5x4, by running a 2x2 window across the 10x10 and outputting the max (most-activated.) pixel/feature value out of the 4 values in the window….

We have now extracted the most activated set of pixel values from each channel.

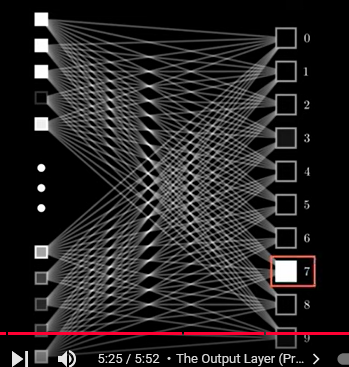
#### Flattening step



We take the 4 matrices and flatten them into one continuous set of 100 nodes (5x5x4 = 100)

#### Output layer

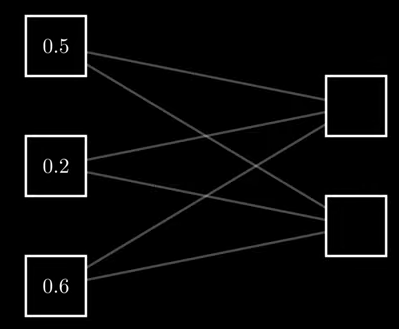
We “run activations” from the fully connected layer of 100 nodes upon the output layer of 10 possible predictions, to predict the output



<not super clear on how this particular activation piece works>

#### Explanation of the output layer – simplified

Let’s say you had 3 input nodes and 2 output nodes, all fully connected…



This connection is based on weights and biases as well…much like any layer in a neural network.

Z = W.X =B

You could add an activation here as well, after Z, but he didn’t add that, “because it was the last layer”.

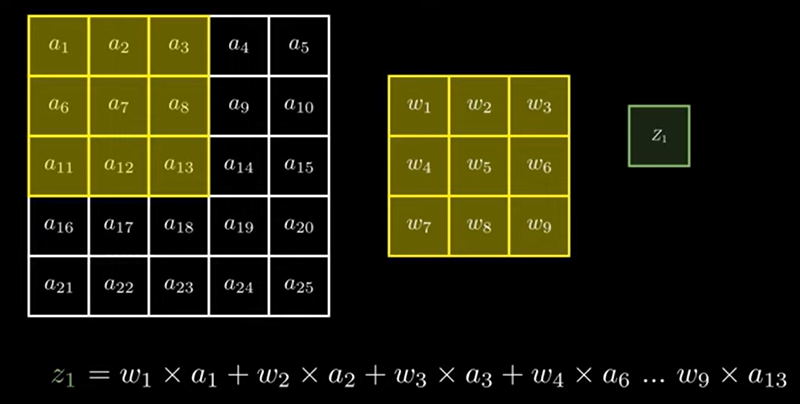
## ForwardProp in CNN

<https://www.youtube.com/watch?v=z9hJzduHToc>

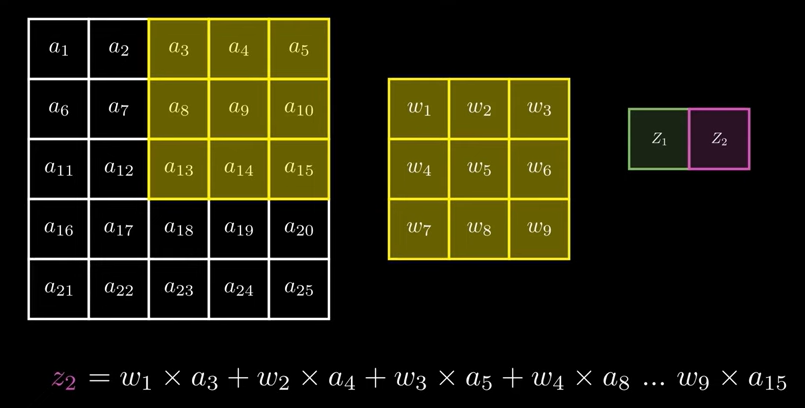
He’ll explain the maths from scratch.

He’ll exclude the bias term from the walkthrough for ease…

Say you have a 5x5 input layer and a 3x3 filter:

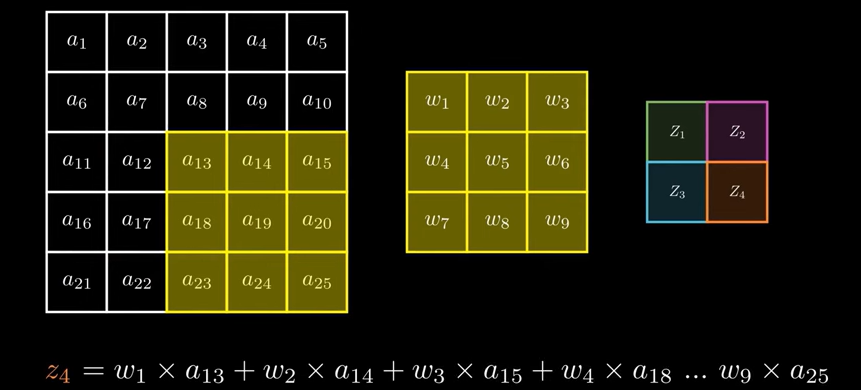


For this iteration, z1 is a dot product of the top-left subset of the input layer.



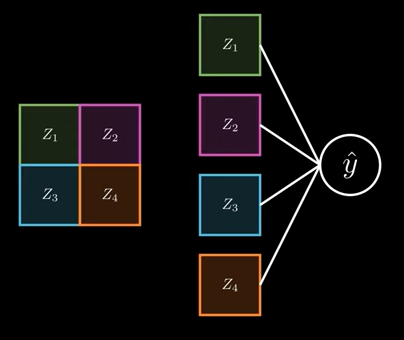
Now the kernel is moved 2 steps to the right, not one. This is because stride = 2.

Similarly, the kernel needs to be moved 2 steps down also…for the next set of iterations.



Thus, the convolution of a 5x5 input matrix with a 3x3 filter returns a 2x2 output at stride = 2

Similarly, 28x28 conv with 5x5 filter will return a 24x24 output at stride =2;



Layer 1 is then flattened and an output prediction is made. This prediction is y^. presumably, the output prediction is made via the activation function.

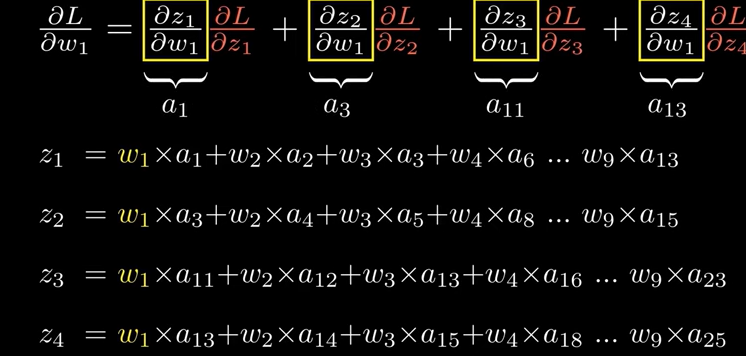
The prediction can then be used to calculate the loss.

## BackProp in CNN

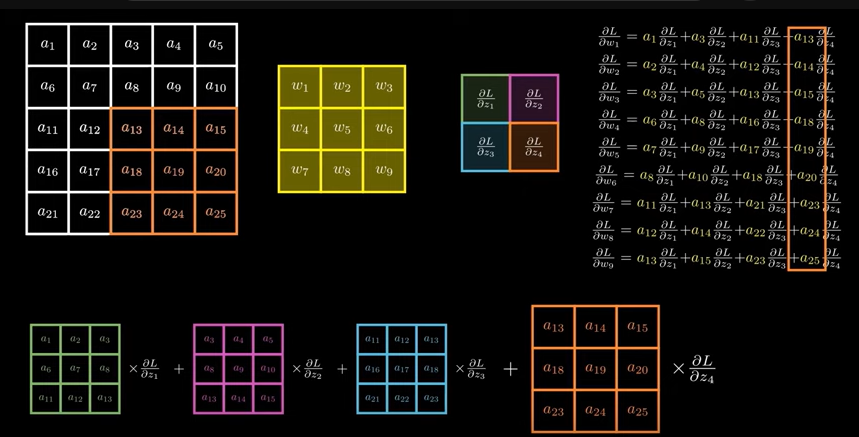
Weight update is same as before –> w\* = w – (a x dL/dw)

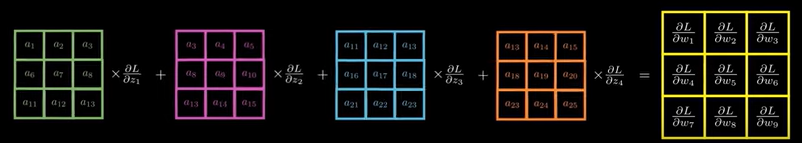
### Calculating Partial derivative of Loss.

Since there is a convolution step in the middle, the partial derivative can be calculated for weight matrix, via convolution output, as shown below:



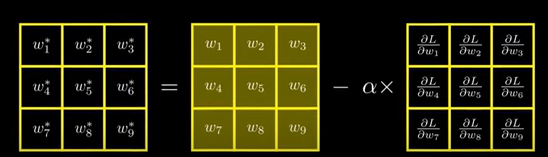
Note – we’ll have to figure out how to do this with matrices…as shown below:





So then it is possible to find derivatives at Matrix level.

### Updating Weights



# 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.
* There is a “Dancing links” Sudoku solver algorithm that can be used as a point of comparison.

# Notes from my implementation

## Important things to remember

* Double check matrix multiplications – where to or not to transpose
* Instead of using A + B, try using np.add(A,B), especially if you’re expecting a smaller matrix to get broadcasted.
* Start with smaller learning rates when updating parameters. I forgot adding the learning rate and my loss kept increasing steadily.

## Model accuracy – as on 11Nov24

My dataset only had 2 sudoku puzzles for the model to work with…I wanted to try to overfit the model first…

## 

model accuracy is defined as the number of cells that were predicted correctly…so after 500 iterations, my model textbook could only predict 1.2% of the cells correctly… ☹

So this model is currently unable to overfit even 2 training examples…bummer.

Normalization of x\_train from [1,9] to [1/9, 1] reduced the loss from 4.1 to 3.9 in certain training iterations…but it also increased the loss to 4.2-4.3 in certain cases…strange.

Would loss be reduced with more training data?? (No)

### Possible Next steps – 12 Nov 24

* Work on sudoku generator to create more training examples. – Increased from 2 to 100 on 17th nov 2024…

## Model update - as on 17Nov24

* Using 100 examples instead of 2 increased the loss from 4.11 to 8.44 in the first 500 iterations…
* Should I introduce another hidden layer then??
* The feeling of giving up is seeping in- <https://stackoverflow.com/questions/44397123/neural-network-for-sudoku-solver>
* In this example, they wanted one neural network for the whole puzzle, whereas I’m building one neural network for each cell…
* Interestingly, reducing the number of hidden layer neurons from 12 to 10 reduced the loss from 8.44 to 8.1.

Note – my model tends to predict one number multiple times and repeats it everytime.

* **Should we train the neural network with super simple examples?** Eg. 1000 examples that have only one cell filled etc…right now, the network has no intelligence about the fact that numbers cannot be repeated…

### Next strategy:

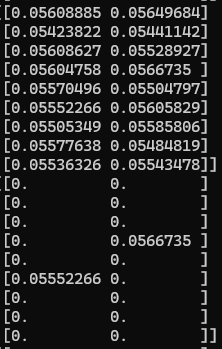
* Created 2000 training examples that only have 1-2 cells missing…accuracy should ideally be higher here…
* Loss went up to 11.1 (facepalm)

### Important check from Andrej:

* If my network were guessing randomly, then it would predict each of the numbers [1,9] with a probability 1/9, **meaning that the loss should be –(log(1/9)) = 0.95**…my loss is 11.1, so maybe I need to curb my initialization loss…
  + Note – Andrej’s model loss dropped from 27 to 4 after 10,000 iterations. I’m only doing 500
* Just initializing B2 (output layer Bias) at 0 reduced this loss from 11.1 to 10.7, and multiplying init weights by 0.1 reduced the loss to 9.8 (9.7 after multiplying them to 0.01), so there is definitely some confidently wrong stuff happening…
* In the training set of only 2 examples, the loss went down from 4.1 to 2.88…but still far away from 0.95

### Important discovery – softmax bugfix

If I look at the softmax function output, ie A2, I see that the probabilities don’t even sum up to 1:

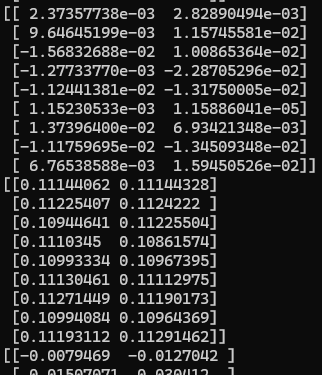


That could also explain the high loss, because while all probabilities are fairly uniform, each of them is half of the actual probability that it should be (0.111), if it were randomly guessing.

This is because this is the output from the 2 example set – np.sum(z) was summing up across BOTH examples, hence the denominator was 2, not 1.

THIS is why increasing the training size would increase the loss, because you’re shrinking the probabilities even further, by taking a sum across multiple training examples, which isn’t correct.

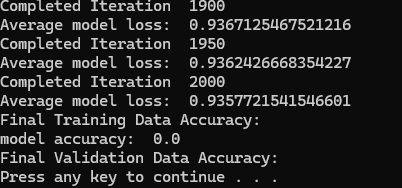
This critical bugfix reduced the loss in our 2-example training from 2.88 to 2.1, and the probabilities at initialization are DEFINITELY as good as random guessing…uniform:



Note – same is true for the training sample of 2000 examples…loss reduced to 2.91 from 9.88.

The other bug was the fact that negative log likelihood was taking natural log with e, instead of log10.

After these fixes, the model is starting to reduce training loss consistently as well…

\

Training the 2-example model for 2000 iterations reduced the training loss from 0.953 to 0.935…so clearly there are a LOT more iterations that the model needs to go through before it can start reducing loss..

This was at a learning rate of 0.0001

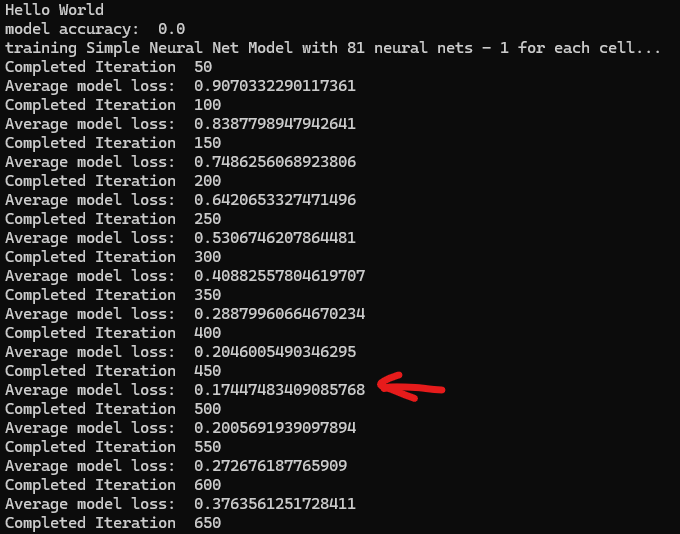
### Increasing learning rate…

Increased learning rate 10x -> from 0.0001 to 0.001

At learning rate = 0.001 -> In 2000 iterations, on 2 examples, loss decreased from 0.953 to **0.637** -> CONSISTENTLY

### Too high a learning rate…

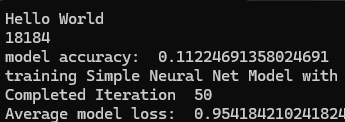
At learning rate = 0.01, the loss reduced for the first 400-500 iterations and then increased dramatically:



So we can create loss thresholds for adapting the learning rate then..

### Bugfix in accuracy computation

Note – there was also a minor bug in my test function, where I was comparing the y output [1,9],to the predicted INDEX [0,8], which is why my accuracy was ALWAYS 0.

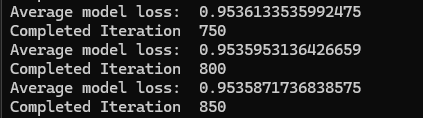


Even without training, the model had 11.22% accuracy on 2000 examples.

After 500 iterations, accuracy increased to 11.98% on training data and 10.63% on validation, so we’re definitely not overfitting…

So we now need more iterations, but we don’t have the compute for it in our laptop.

Note – I tried 1000 iterations on the 2000 training set – the minimum loss that I achieved was 0.9535, which was between iteration 700-800, after which the loss started increasing again.



### Possible next steps

* Scale down the training dataset from 2000 to 200, for a realistic training time.
* Track the prediction probabilities of each cell and pick the most certain one….
* Intuitively, I think my approach to have simple to complex examples is correct…so that the network is trained to be able to solve a Sudoku at any stage…now we need to scale this up.
* Create a layered NN class which can maintain an array of neurons as one object, and can determine its size based on the previous layer supplied to it…such that you only need to define the size of input layer (81xn) and output layer (10x?), and then you can easily create more layers in the middle…

# CV Points:

* Year when you made this - 2025/26
* Was built **without** Tensorflow or Pytorch.
* Sudoku solver AI based on Neural networks

### Interview points

* How many layers did the network have? Which neural network configuration performed best?
* Performance compared to algorithmic solver
* How many optimizations were tested?
* Built the Stanford paper algo from scratch.

#### Why Sudoku

* Unambiguous solution which can be algorithmically verified…this can’t be done for image classification
* Input dataset is easy to generate by oneself as a learning exercise, even though there were kaggle datasets available.
* Because of the algorithmic nature of input data, multiple different techniques, like CNN, RNN and genetic algorithms could be explored