Data Science Toolbox: Assignment 4

Junfan Huang
Sam Harding

March 2019

1 Abstract

In this task, our aim was to train our neural networks to learn the Caesar Cipher rule and be able to accurately model the decoding function. We chose this topic since the Caesar decoding function (addition modulo 26) is not a trivial function to map, and we wanted to see if a powerful prediction device like a neural network can correctly learn the rule. Our simulation is divided into 3 parts: Caesar Cipher modelling, Sequences of letters modelling, Noise Robustness testing.

In the first part, we start by modifying the dimensions of input and output space to give us 4 similar but fundamentally different types of neural network structure. We found that in some cases, our neural network cannot fit the function well. So we designed an artificial smooth function corresponding to Caesar Cipher rule and explored how it learned the function as time went by. We found that providing the network with more inputs information generally allows the model to decode better, which conforms with our understanding of neural networks.

As an extension of our model, we then decided to supply the network with more than one letter as input at a time. We achieved this by using a binary matrix named **letter position matrix** to inform the model of each letter in the input. After that, we hide some letter pairs, see if model can still learn to decode correctly.

Finally, we added different rates of random noise to our data pairs prediction model in order to test the robustness of our neural network and analyse how it reacts to error within the data. It is really exciting to see the visualisation of the Deep Learning algorithms noise robustness.

The code for this project can be found at `https://github.com/xihajun/Data-Science-Deep-learning-Sam-Jun`:

- `./project summary/project.ipynb` contains the code for running our simulations and producing the figures in this report.
- `./documentation/`contains rough versions of these and provides evidence of our work.

We have chosen to split the equity as follows:

- Junfan 1/2
- Sam 1/2

2 Neural network

A neural network is a form of deep learning, which attempts to mimic the human brain's natural excellence at a problem such as image detection (which is otherwise almost impossible to solve by algorithmic classification methods). The idea is to model and replicate computationally the neurons in a human brain (shown in figure 1).

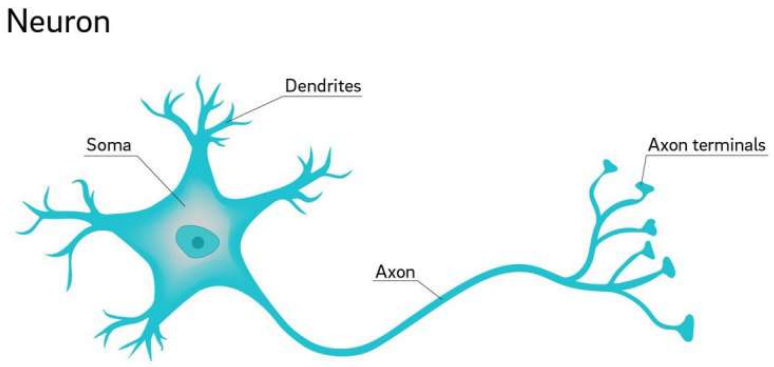


Figure 1: Biological neuron in the brain

The dendrites respond to electrical signals from other neurons, and send an electrical pulse down the axon to the terminals, which then pass on information to other neurons.

The computational neural network follows a similar pattern. We start with a series of inputs (similar to dendrites) which take in information. There are then a series of middle layers containing nodes, to which each of the nodes from the previous layer are connected. These links are weighted (the weights are updated by training the model), and only become activated based upon an activation function which the user selects. Finally, we reach the output nodes (the axon terminals in the biological model), giving the final values of the model. The weights of all the links are updated during training, with the intention to overfit the model (so that the network is as trained as possible). Then with new inputs, the model can predict the outcome.

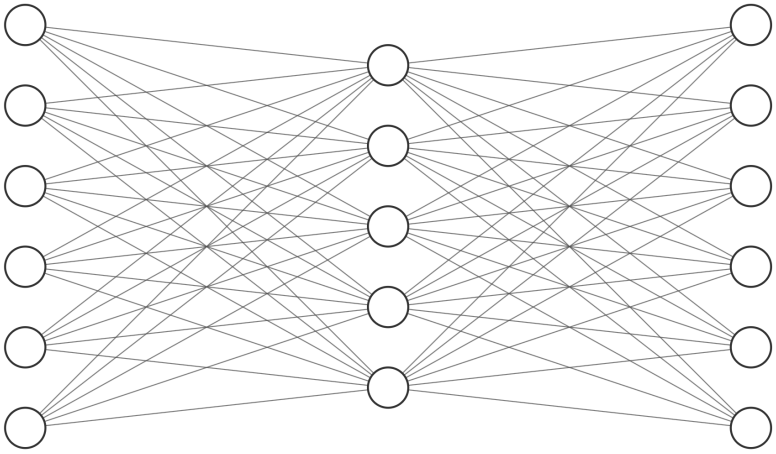


Figure 2: Example of neural network with one middle layer

2.1 Activation function and loss function for discrete output

Before we start to build our model, we should consider when should we use and how to use activation function.

2.1.1 ReLU function

One activation function we considered was the Rectified Linear Unit (or ReLU) function, which considers only positive values:

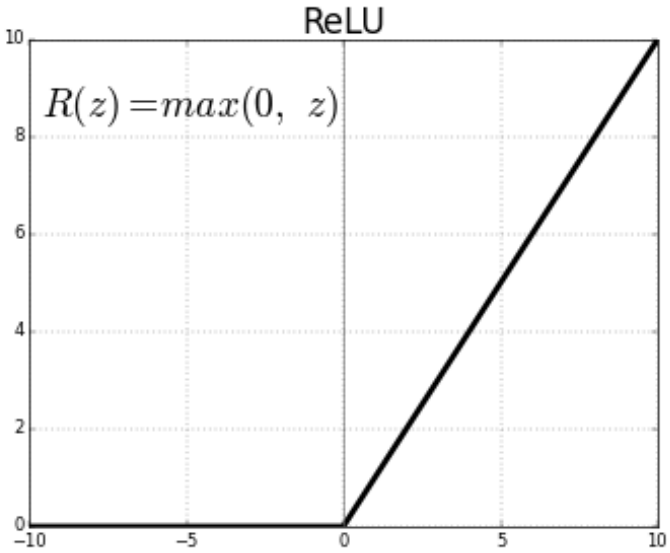


Figure 3: Rectified Linear Unit function

The first model is 1 input and 1 output, so our neural network acts like a function

$$f : \{0, \ldots, 25\} \longrightarrow \{0, \ldots, 25\}$$

In this case, the ReLU activation function may be more suitable as it acts very much like a linear function, which is the sort of thing we want our neural network to learn (albeit modulo 26).

2.1.2 Mean Squared Error

The choice of loss function is also important for a neural network, providing the model with a metric to determine how far the predicted output lies from the real output, and the model can adjust weights to minimise this "loss". Although our problem is more like classification than regression, we started with regression loss functions since our inputs and outputs are numeric.

Our first function was Mean Squared Error, or MSE. This is the standard loss function for neural networks, as it is very generic to apply and gives a good statistical measure of distance. The MSE of two vectors $\mathbf{x}$ and $\mathbf{y}$ of length N is calculated using the following formula:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - x_i)^2,$$

where y is the target, x is the output. In our base case, where we have one input and one output in the set $\{0, 1, ..., 25\}$, we simply calculate the distance of the predicted from known output. For example, if the predicted decoded output is 26 while the label is 24, then the MSE for that case is 4 (2^2). Our goal is to minimise the MSE value and get a better prediction.

2.1.3 Mean Absolute Error

There are still other loss function can be chosen. Another simplest one is Mean Absolute Error, or MAE, which similarly defined as follow

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - x_i|,$$

where y is the target, x is the output.

Our output is one dimension which means $N = 1$. Intuitively, these two loss function can be shown below:

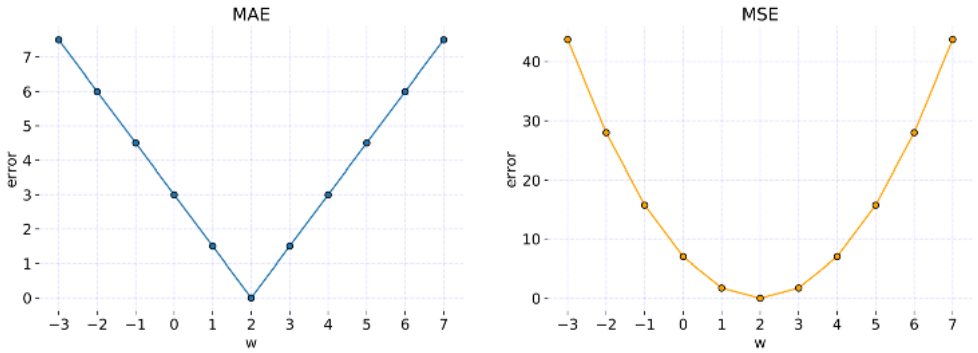


Figure 4: MSE and MAE when N=1

Compared with MSE, the MAE is more robust to outliers. If we make a single very bad decoding prediction, the squaring will make the error even worse and it may skew the metric towards overestimating the model's badness.[1] But in our case, we need a high accuracy, MSE is more suitable in theory as well as in our simulation.

However, when our output values turns to 0 and 1 as a classification problem, it is better to choose other loss functions. There are some loss functions which have a better performance in our simulation. To introduce this, we have to go back the activation function.

2.1.4 Sigmoid and Softmax activation function

Sigmoid function maps a real number into (0,1), which corresponds to our "classification" problem. But it is not suitable to be an activation function for middle layer which could be seen in the following paragraph.

As for Softmax activation function[3], it can be treat as a general sigmoid function which has the following form

$$p(c_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

It usually used for multi-classification problem which is similar to our simple Caesar cipher models in the first part.

2.1.5 Binary Cross Entropy

Sigmoid and softmax activation functions also correspond to some special loss function. For brevity, the binary cross entropy(BCE) loss is [2]

$$\text{loss}(x, y) = -\sum_{i=1}^{N}[x_i log(p(y_i)) + (1 - x_i) log(p(1 - y_i))]$$

where y is the target, x is the output.

This is a classification loss function, so this only works with our binary vector outputs (so $N = 26$). This function means that when a coefficient in the prediction vector is 0, the loss function adds $log(p(1-y_i))$ to the total loss. Conversely if the coefficient is 1, the loss function adds $log(p(y_i))$ to the total loss.

2.1.6 Categorical Cross Entropy

Categorical Cross Entropy Loss also called Softmax Loss. It is a Softmax activation plus a Cross-Entropy loss which has the following form.[2]

$$loss(x, y) = -\sum_j y_{ij} log(x_{ij})$$

where y is the target, x is the output, i denotes the data point and j denotes the class.

This is the loss function of choice for multi-class classification problems and softmax output units. For hard targets, i.e., targets that assign all of the probability to a single class per data point, providing a vector of int for the targets is usually slightly more efficient than providing a matrix with a single 1 or 0 per row.

Because of its property, it might not suitable for use in the multi-label problem in our second part, i.e., sequences of letters modelling (as there are more than one label).

3 Caesar Cipher Modelling

3.1 Caesar Cipher

We aim to test whether a neural network can figure out the underlying structure of a simple cryptographic cipher, say the Caesar cipher. This is a very basic method of encryption in which each letter of the intended message is shifted by three down the alphabet. For instance:

HELLO $\longrightarrow$ **KHOOR**

This method of encryption is easily broken with a basic knowledge of cryptography; one need only perform frequency analysis on the letters in the ciphertext (given that it is long enough), or even just cycle through all 25 possible shifts to recover the original message. However, we are interested to see if a neural network can learn the relationship between input and output as easily, without this meta-data approach.

We chose to focus on analysing the success of our model under different architectures. For instance, we can represent the initial input of each letter as a numeric value between 0 and 25 (where 0=a, 1=b, etc) or as a binary vector of length 26 (where the vector is all zero except for a 1 in the position corresponding to the appropriate input letter). We examined these kinds of variations, along with choices of activation and loss functions, to see how they affected the success of our decoding model.

3.2 Modelling

After considering how to choice the activation function and the loss function, we should to consider the structure of our neural network model.

3.2.1 One input vs One output

Our baseline model considered a model like this:

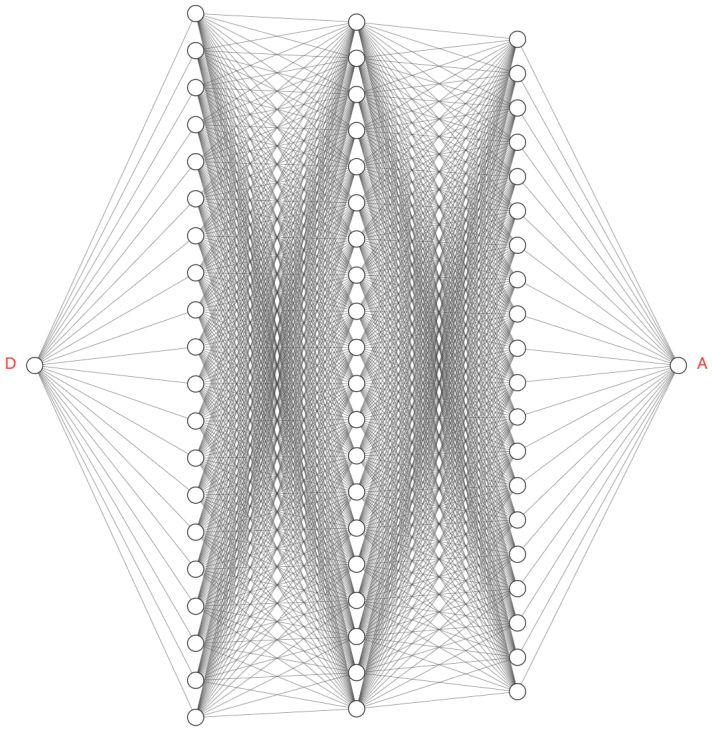
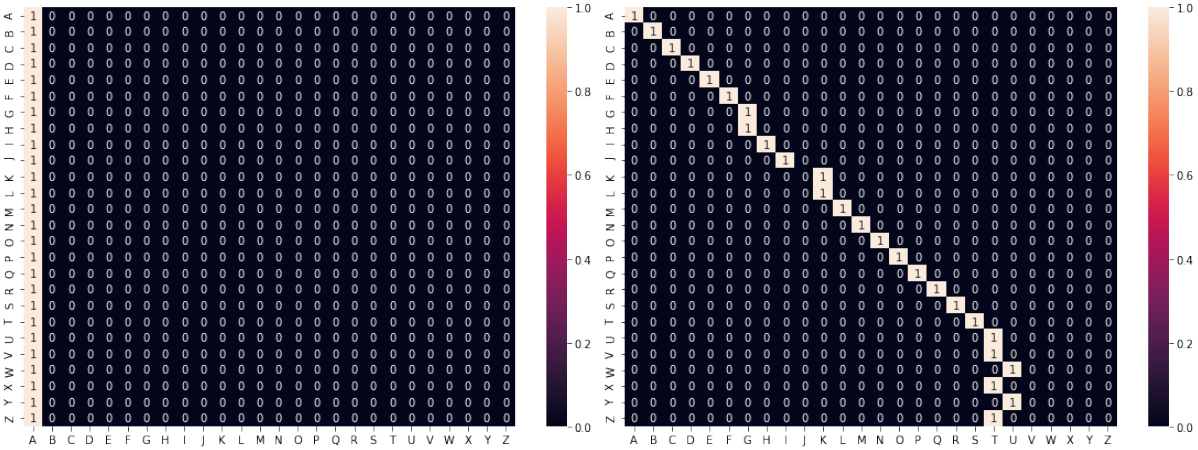


Figure 5: The baseline model structure

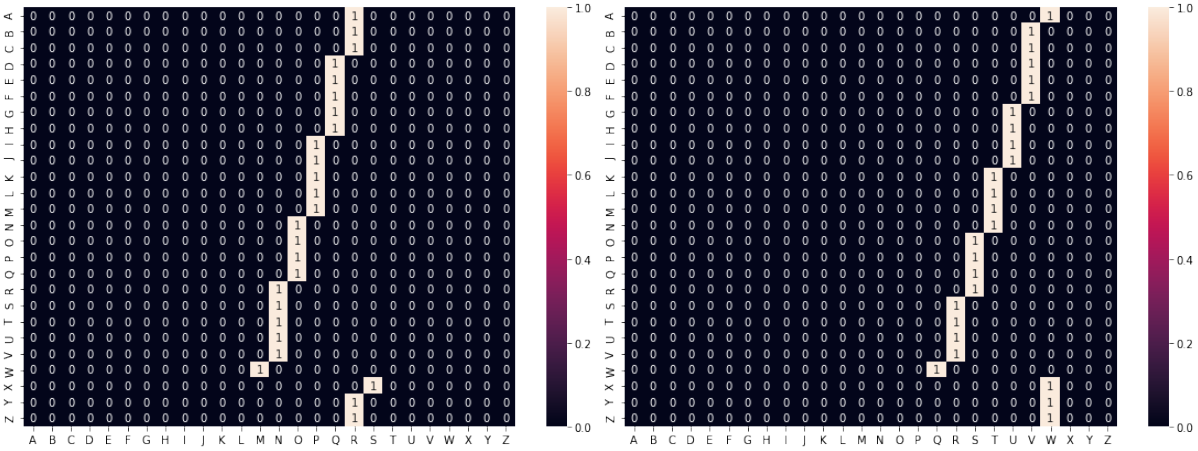
Ideally, the well trained model are able to do a precisely decoding. However, no matter how we changed the structure of the activation function, loss function, optimiser, number of neurons as well as number of layers, the outputs were always unsatisfactory which can be seen in the confusion matrix below.



(a) Without Batch Normalisation

(b) With Batch Normalisation

Figure 6: Normalisation comparison



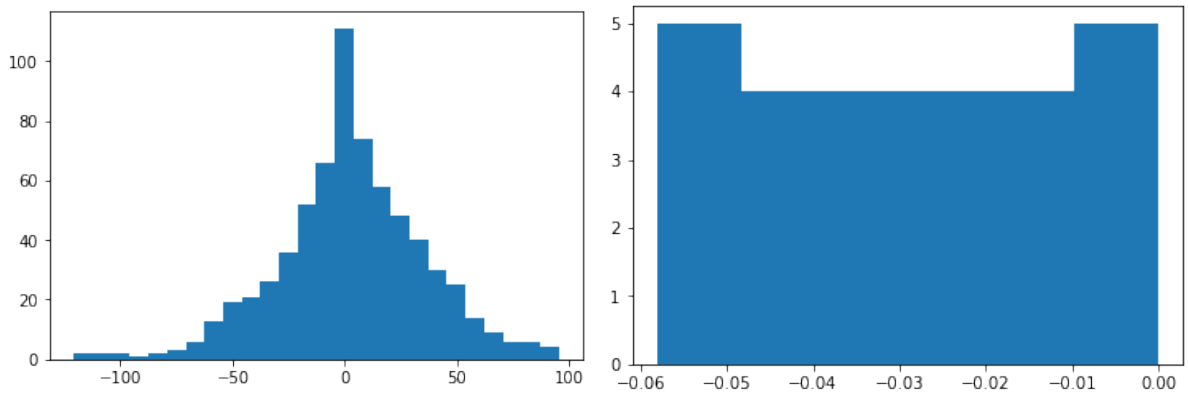
(a) Non-linear activation function

(b) Non-linear with large number of neurons

Figure 7: Using Non-linear activation function

As we can clearly see, the model with the best performance is the one with batch normalisation and ReLU activation function which means our activation judgement might be right. However, although the performance is better than without batch normalisation and a non-linear activation function, the performance is terrible in general. Let's figure out what happened by using our designed function.

Here we show the distribution of values in one of the middle layers for one input instance, with and without batch normalisation. From this we can see that compared with the model without it, batch normalisation does in some way improve the model's capacity and retains information given by the inputs (which is clearly lost in the model without BN). This clearly suggests that BN is a useful addition to our model.



(a) Middle layer output with BN

(b) Middle layer output without BN

Figure 8: Comparison of Batch normalisation

3.2.2 Artificial Caesar function

In fact, Caesar Cipher has this form $E_n(x) = (x + n) \bmod 26$. Our goal is to design a smooth function to fit it and see given more data, if our neural networks can be better and how it works, what did it learn. (Basically, the decoding process and encoding process are the same, so we tried to fit encoding function this time.) For mod operation, it is hard to get a function to fit it. Luckily, it can be treat as two function, and if we see in this way, it is easy to get a function

$$y = \mathbb{1}_{x<23}(x) \cdot (x + 3) + \mathbb{1}_{x\geq 23}(x) \cdot (x - 23) \quad \text{where } x \in \{0, 1, \dots, 25\}$$

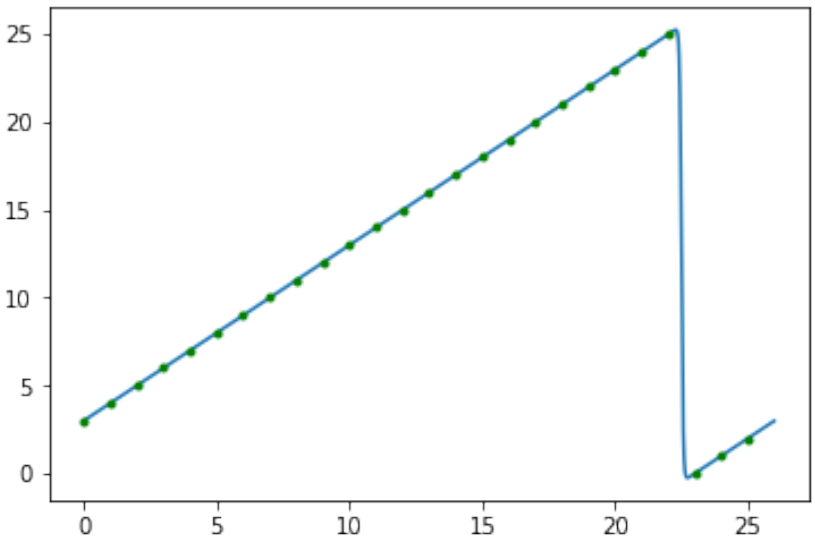


Figure 9: Caesar Cipher

As the sigmoid function looks like this, which can be treat as a step function if do a small modification.

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}. \quad \longrightarrow \quad S_1(x) = S(-100(x-22.5)), S_2(x) = S(100(x-22.5))$$

It is easy to see that sigmoid function can be a replacement as our indicator function above. Then we get

$$y = S_1(x) \cdot (x+3) + S_2(x) \cdot (x-23) \quad \text{where } x \in \{0, 1, \dots, 25\}$$

Given this function, we are able to generate more data and have a deep exploration what our model actually learned.

We chose the simplest model with 500 neurons in the only one middle layer and the three pictures below show what our model actually learned.

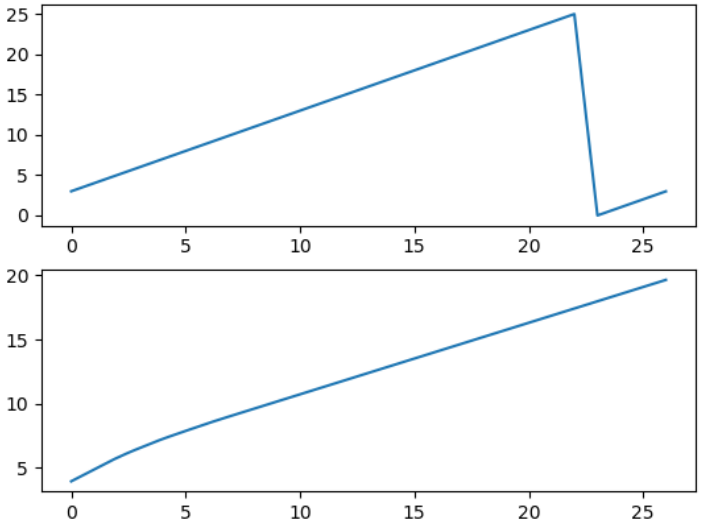


Figure 10: 500 neurons with 100 epochs

After 100 epochs, networks learned a linear line to fit the curve.

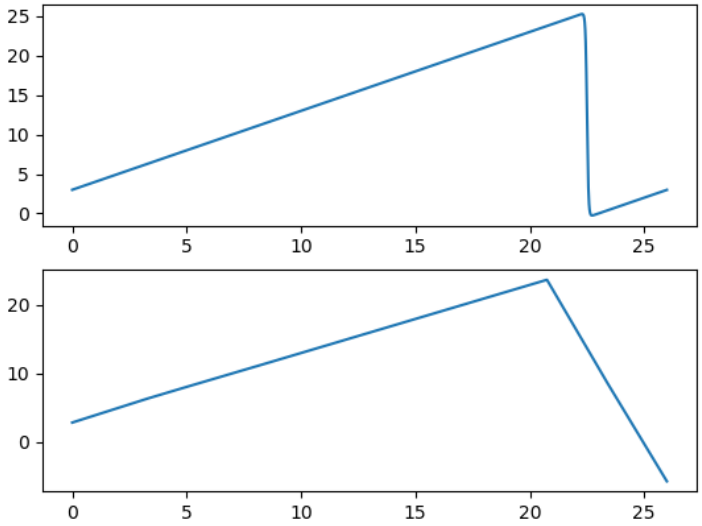


Figure 11: 500 neurons with 1000 epochs

After 1000 epochs, our model realised that there are some mistakes and tried to fix it.

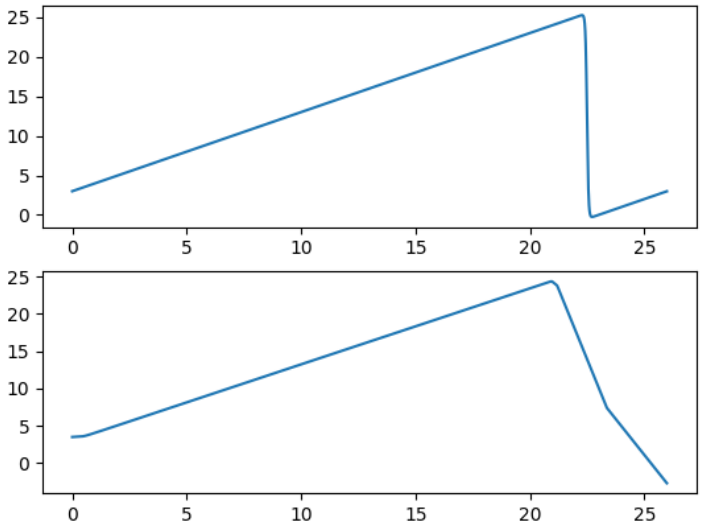


Figure 12: 500 neurons with 5000 epochs

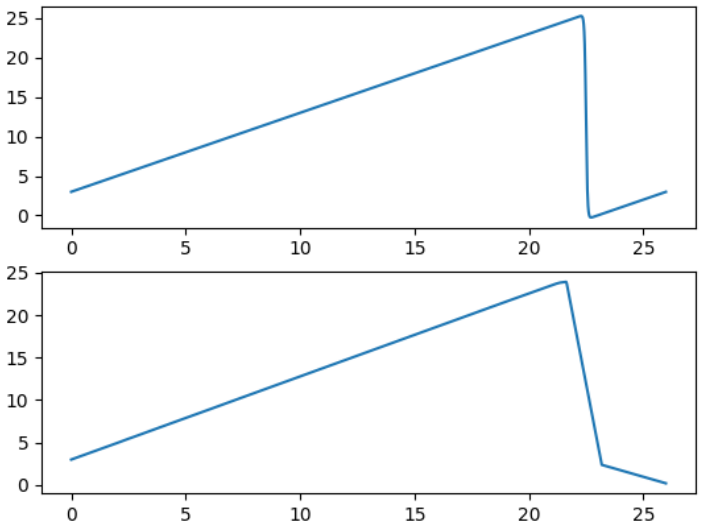


Figure 13: 500 neurons with 20000 epochs

These figures show that even with many epochs, the model still cannot fully learn the required function (shown above each plot). We can conclude that in this case the ”step” function for Caesar cipher decoding is simply too hard or strange for a neural network to learn.

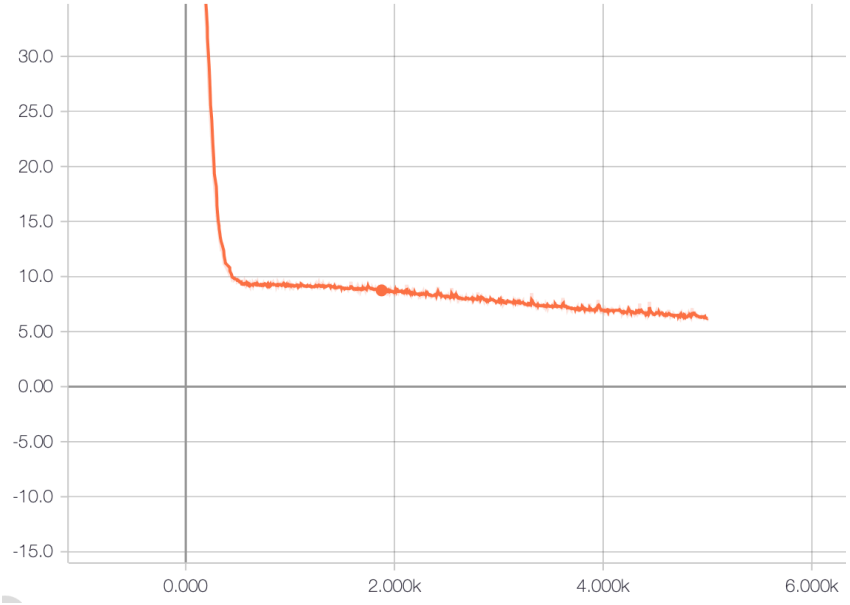


Figure 14: Loss reducing situation in this model

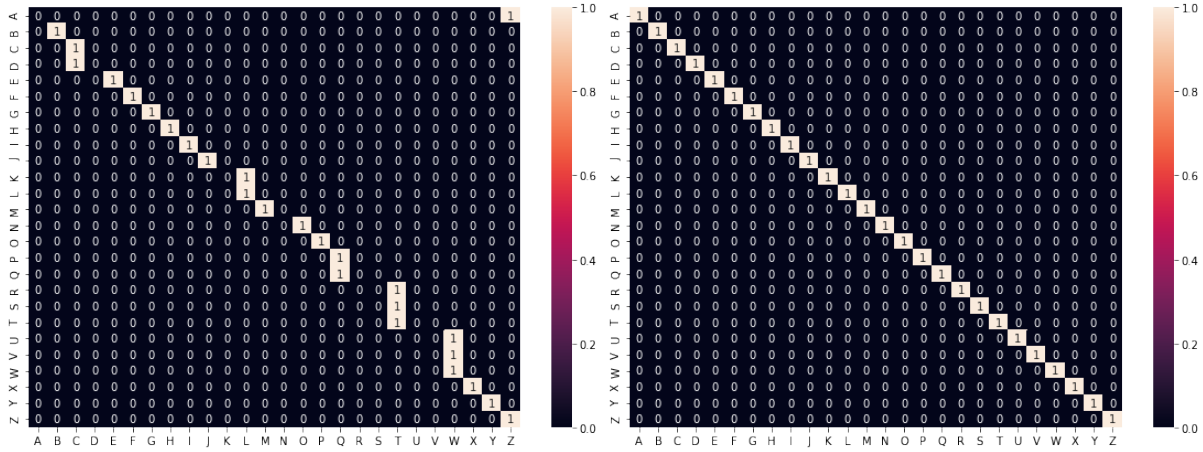
Because it is hard for our model to learn the angle, the loss function stayed at 9 (which corresponds to the last 3 shifts) for a long time and decreased really slowly.

3.2.3 1 input vs 26 output

When we tried to train our model to learn 26 output, we considered which activation function should be used in the discrete situation. We also were able to use a classification loss function, as our output

became a binary distribution in this case.

To our surprise, given one input and enough epochs, neural networks have a good performance now. Also, a nonlinear activation function inside seems helpful to improve the performance.



(a) 300 epochs

(b) 600 epochs

Figure 15: 1 input vs 26 output

3.2.4 26 input vs 1 output

After considering output structure, we can also modify the input structure. Given more information to the input, we are able to get two layers neural network and fit model well.

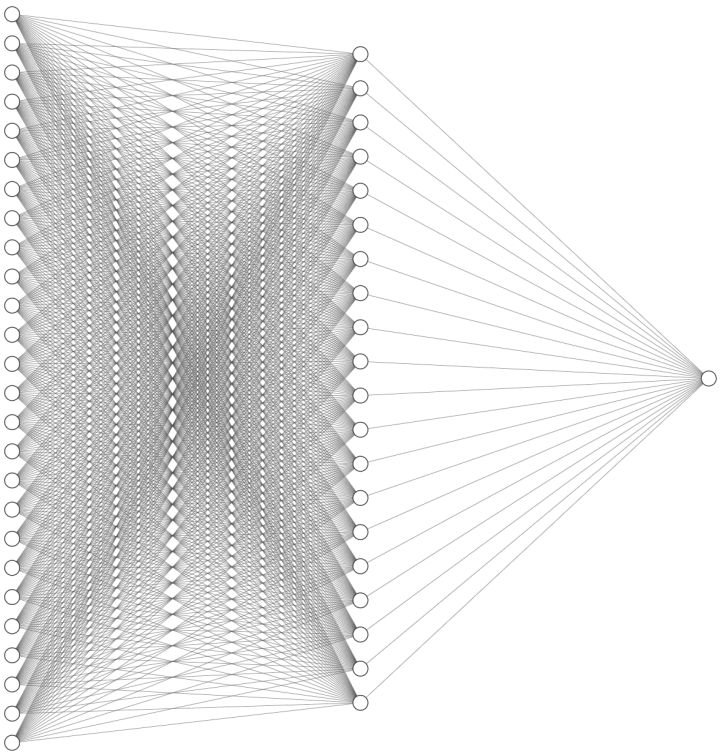


Figure 16: Neural network structure

The idea is to change the input letter to a vector. For example, the input letter $A = (1, 0, 0, ..., 0), B = (0, 1, 0, ..., 0), ..., Z = (0, 0, 0, ..., 1)$.

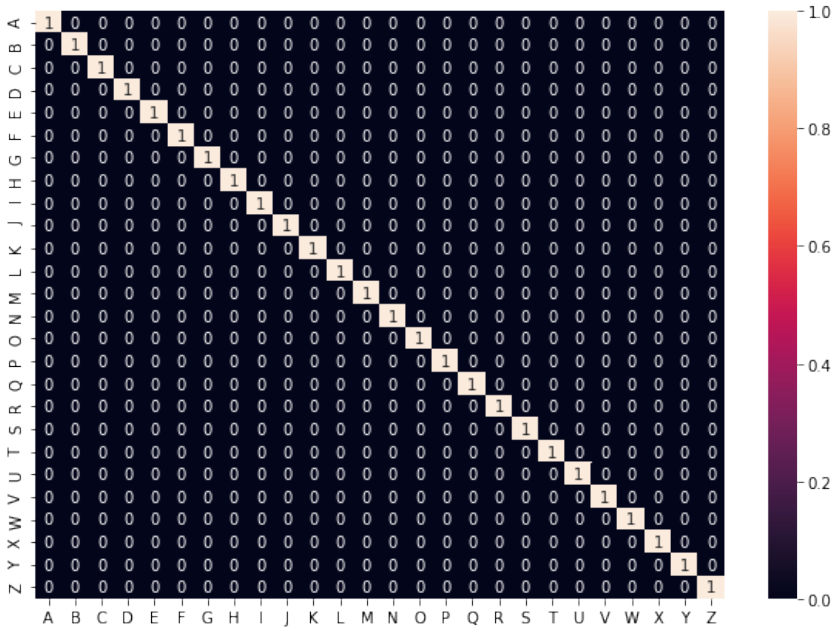


Figure 17: Confusion matrix

3.2.5 26 input vs 26 output

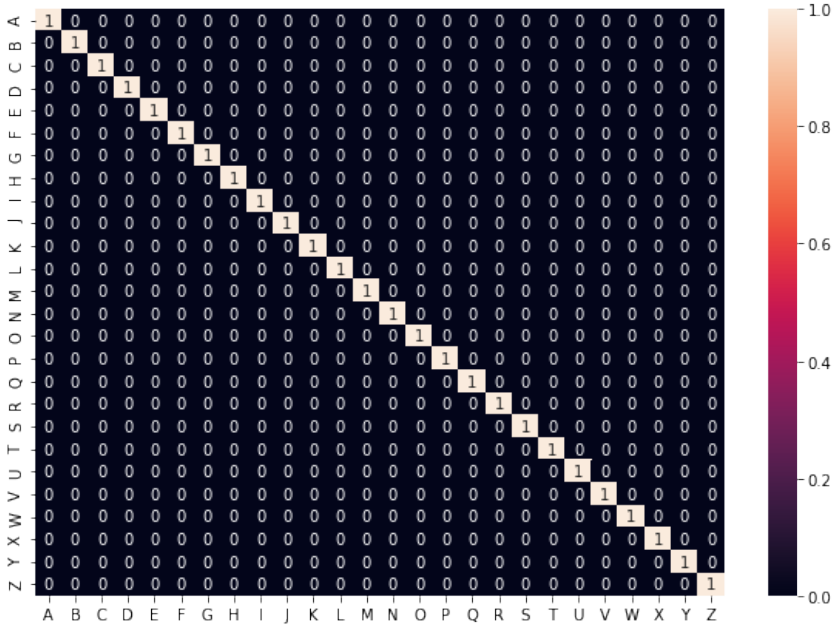
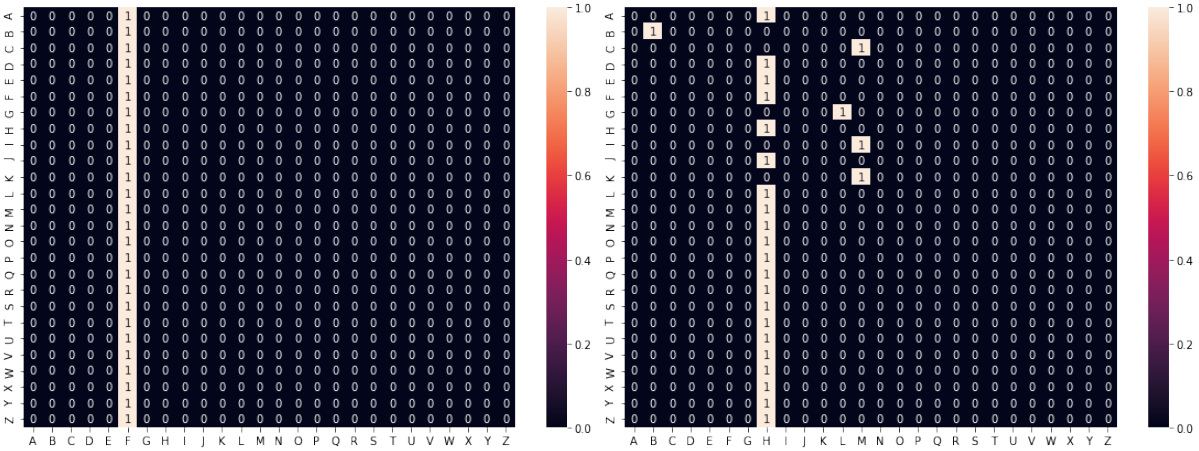


Figure 18: Confusion matrix

Even without middle layers, the model could learn things really well, as it is just an one-to-one correspondence. Furthermore, adding hidden layers to the neural network only decreases accuracy, which possibly proves an inefficiency in using neural networks to crack a Caesar cipher, as they are designed to model far more complex problems, and tend to ”overthink” simple functions.

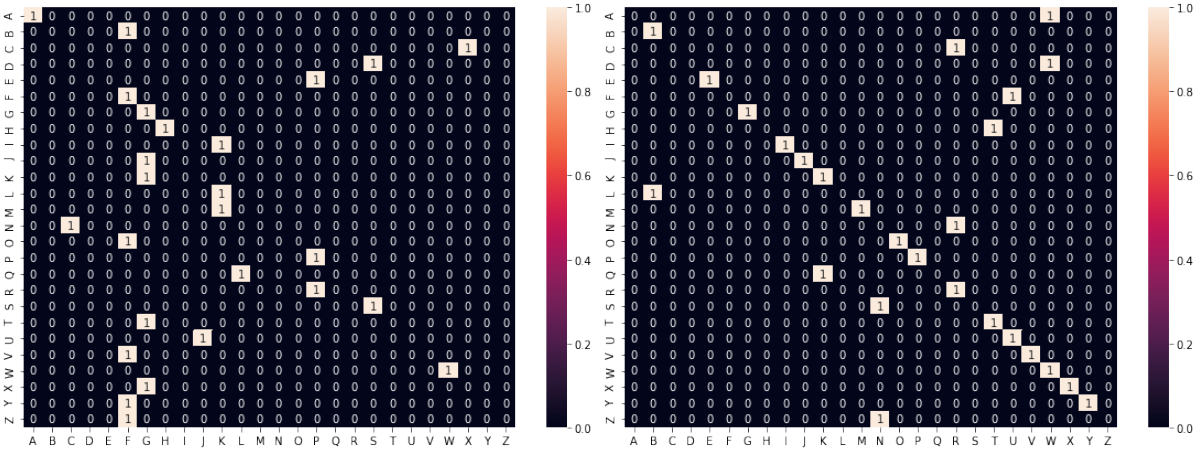
We now aim to change our network architecture to really examine how the model's accuracy varies. We use our binary vector for input and output, meaning that we have 26 nodes on each end of our network. We allow one hidden layer in our model, starting with one node, and increasing the number of nodes until the network reaches 100% prediction accuracy. We call this model the "funnel model", as we start by trying to funnel our input through a single node in the hidden layer. The following confusion matrices show the results:



(a) 1 neurons in the middle

(b) 5 neurons in the middle

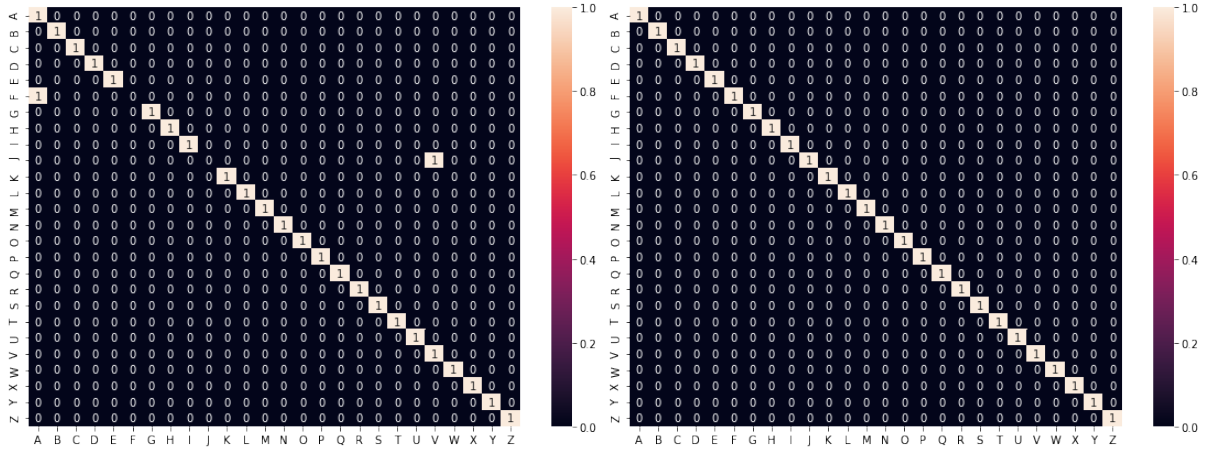
Figure 19: 1 input vs 26 output



(a) 10 neurons in the middle

(b) 15 neurons in the middle

Figure 20: 1 input vs 26 output



(a) 20 neurons in the middle

(b) 25 neurons in the middle

Figure 21: Different number of neurons in the middle layer

As expected, these results show that increasing the number of nodes in our hidden layer increases our decoding accuracy. With only one node, our model is very poor indeed, only being able to predict the letter **F** for any input. As we gradually increase the complexity of our model, the accuracy too increases until we reach full decoding accuracy by 25 nodes.

From this, we can conclude that this sort of network structure is not a suitable choice for our problem. Although this model achieves accuracy with enough nodes in the hidden layer, we aim to examine the drawbacks in this model. Clearly many nodes are needed to achieve good accuracy, which requires more computational power. We could also attribute the failures in our model to our choice of activation function.

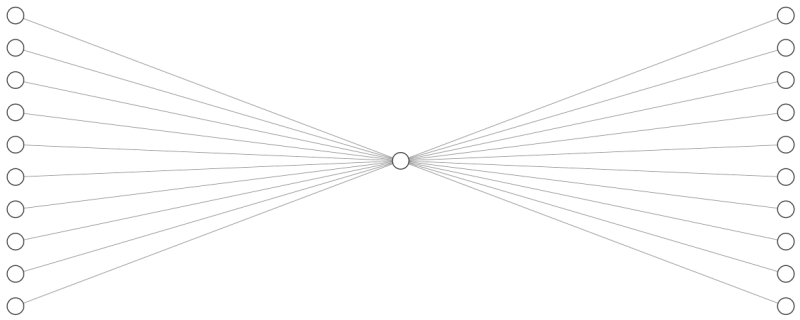


Figure 22: "Funnel" structure for deep learning

4 Sequences of letters modelling

Having investigated into the effect of network architecture on the success of our model, we looked for ways to improve it by considering more letters at a time, in order to see if this had positive or negative consequences on model accuracy.

Up to now, we only supplied the neural network with one of 26 inputs (members of our alphabet), and expected one of 26 outputs. This gave us very small training and testing sets.

In an attempt to improve our modelling approach, we decided to try training our model on two letters at once (for example providing **AB** as an input instead of just **A** or **B**). Again, there was more than one way of representing this tuple mathematically.

Given our alphabet $\mathbb{A}$, we can either represent our letter tuple as a vector of length two (that is, as an element of $\mathbb{A}^2$), or alternatively as a 2×26 binary matrix (where the position of the 1 in the top row corresponds to the position of the first letter in the alphabet, and the second row corresponds to the second letter). So our definition of the **letter position matrix** as follows:

- Given an alphabet with N elements, and sequence with length M, our letter position matrix L have M rows, N columns, that is L_{MN} and $L_{ij} \in \{0, 1\}, i \in \{0, 1, ..M\}, j \in \{0, 1, ..., N\}$.
- Each column represents a unique letter in the alphabet.
- Each row represents the position of the letter in the sequence.
- If $L_{ij} = 1$ means that the jth element in the alphabet occurs in the ith position in the sequence. For example, the sequence AB can be written as $\begin{bmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 \end{bmatrix}$

We start with pairs of letters. Firstly, we reshaped the matrix to a single vector as our model's input. In this case, we have 676 total possible letter combinations. Limited to the time, we just focus on one particular neural network structure - where input space is the same as output space (using the matrix method of input and output described above) and we selected sigmoid activation function and binary cross entropy loss for our neural network.

Because of the speciality of the matrix, we cannot use the softmax activation function in theory as it is not a multi-classification problem anymore. Softmax is designed to predict one output distribution, but here we are trying to obtain 2 outputs, so softmax is unsuitable. To verify this in simulation, we tried to use the softmax activation function combined with categorical cross entropy loss function. The results shows that it is really bad the output space can be treat as an distribution while it is not a classification problem, our model need to learn to decode two letters.

Our model have learned decoding rule quite well. From the error table, we can find an interesting phenomenon.

Table 1: Errors in decoding

Original Plaintext	Incorrect Prediction
PB	PF
WU	WM
GR	HE
WW	JW
RR	XY
SM	VX
QW	CW
VN	VX

Even though the model made the wrong prediction, it is clear that it still learned something correctly. This is especially visible for the letter W, as regardless of its position in the plaintext, our model seems to easily recognise it even with unlearned information (ie. U,J,Q). This phenomenon is quite interesting which might mean that our neural network learned that our inputs are from two different letters even though we merged the information into a single vector.

This approach can be easily generalised to more letters, and we could investigate further with three or more letters at a time. We focus on two letters, and briefly consider three and four letter combinations later in our simulation, but these took much larger computational power (as there are $26^3, 26^4$ total possible combinations respectively) and were much less effective at decoding encrypted text. However it is interesting to note that only a very small number of these combinations of letters will appear in natural English text, and some will appear much more frequently than others, so there could still be some success of this method if trained on real-life text.

4.1 Model simplify

To better examine the effect of changes in network architecture on success, we thought it might help to use a similar cipher on a much smaller alphabet. Our thinking is that eliminating complexity in the underlying problem will help us to focus on the structure of the network.

Our new alphabet set is $\mathbb{A} = \{A, B, C\}$, and our new encryption rule is a shift of 1 place in the alphabet. In other words, the encoding space is a bijection permuting $\mathbb{A}$ to $\mathbb{A}' = \{B, C, A\}$. This is no longer a true Caesar cipher, as this is by definition a shift of three letters, which would be an identity map and not an encryption in this case.

As the problem is really simple, we can use a small size of neural network to train it and see all the weights and output distributions.

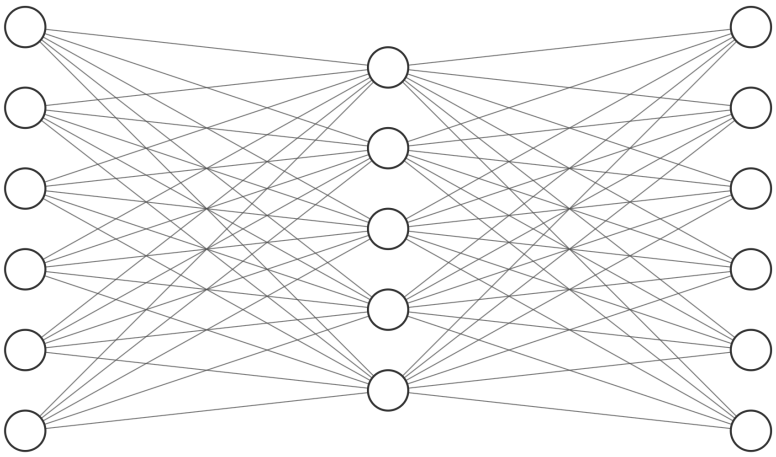


Figure 23: Structure of our model

By using this equation below, we are able to get every layers outputs and get the middle layer output distributions which shown in the **project.ipynb**.

$$\begin{aligned} y' &= Wx^T + b \\ y &= sigmoid(y') \end{aligned} \tag{1}$$

As the result of our investigation, we saw that the model was very bad at predicting for a letter pair it had not seen before. For example when we hide a pair AB for training, and then try to test the model with AB, it failed to correctly decode.

5 Noise Robustness testing

Since our model has high performance under standard testing, we decided to make the problem more complex by investigating how well our model deals with random noise. That is, we allowed increasingly more random error into our training labels. For example, if normally

$$\mathbf{A} \longrightarrow \mathbf{D}$$

then there is now a chance that

$$\mathbf{A} \longrightarrow \mathbf{C} \quad \text{or} \quad \mathbf{A} \longrightarrow \mathbf{Z}$$

The erroneous label is chosen uniformly from the set of incorrect letters.

The picture below is a noise robustness testing for two sequence.

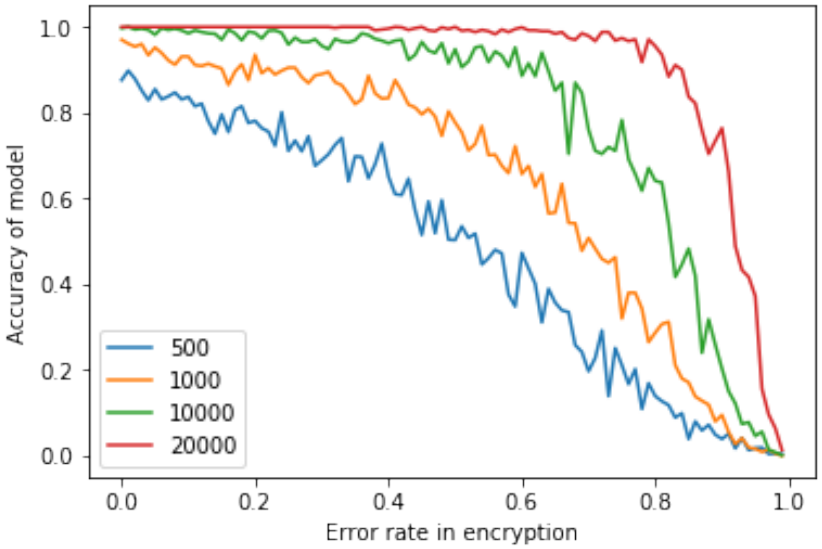


Figure 24: Noise rate vs. Model accuracy, for different training data sizes

As represented by the different coloured lines, we attempted this method with several variations of our model. As noted in the legend, the lines represent different sizes of training set used to tune the model (500 data points, 1000 data points, etc). Naturally with more instances of each letter to train on, the overall accuracy was better with more data points. It is very interesting to note the elbow in the graphs: The higher-data models are still very accurate even with 60-70% random noise overall.

To sum up, this test shows that Deep Learning algorithms are quite **robust** to **random errors** in the training set especially for the **sufficiently large training set**. This result agrees with research in paper [4]. However, for systematic errors (e.g. every error maps the given letter to **Z**) it is predictable that the model will not keep the same success rate.

6 Discussion and Conclusion

Our work suggests that a powerful computational tool like deep learning may simply be unsuitable for a task like decoding a Caesar cipher, since it requires high computational power and finds it very difficult to learn the basic modulo addition function. However, this topic allowed us to explore the effects of network architecture on efficiency and accuracy of a neural network, which we believe is a useful insight.

In the first part, after playing around with the architecture of the network, we found that the structure played an important role in the success of our decoding model. When we used our vectorised inputs and outputs, the model had a better ability to decode. However in the 1-input v 1-output case, our model (without Batch Normalisation) was very ineffective at decoding the encrypted letters. In addition, even for vectorised input and outputs, the hidden layer structure is also important in training an accurate model. For example, with the Funnel structure in Figure 22, we saw that when the hidden layer contains very few nodes the resulting performance was disappointing.

Next we tried Sequence Modelling, which proved successful. The model could decode sequences of letters as easily as just one letter, provided it had seen the data beforehand. However, when we tried to hide some combinations and test on these unseen data the model was much less effective. It could sometimes still correctly predict one letter, but most often failed at identifying both.

We also saw that the neural network model is robust under the inclusion of random noise, given a sufficiently large size of training set in our last part.

In short, we thought that deep learning can fit any function by using any structure, however, world not always goes smoothly. The structure design as well as activation and loss function choices take a important role.

Future work:

- Try a more complicate cipher, for example Substitution Cipher, and apply it into a longer paragraph.
- Use Differentiable Inductive Logic Programming method to guide the neural network to learn unseen data. For example, given AB, can it learn BA.
- Use Convolutional Neural Networks for letter position matrix which might remain more information and be able to learn unseen data, just like it works for image.[5]
- For the robustness testing, we haven't tried the performance given the unbalanced data. But we can predict the performance would be worse as some parts have smaller size but bigger noise.

References

[1] Georgios Drakos. How to select the right evaluation metric for machine learning models: Part 1 regression metrics. `https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics` Accessed: 2019-03.

[2] Raúl Gómez. Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing names. `https://gombru.github.io/2018/05/23/cross_entropy_loss/`. Accessed: 2019-03.

[3] Daniel Lawson. Neural nets and the perceptron. Accessed: 2019-03.

[4] David Rolnick, Andreas Veit, Serge Belongie, and Nir Shavit. Deep learning is robust to massive label noise. *arXiv preprint arXiv:1705.10694*, 2017.

[5] Pete Warden. How do CNNs deal with position differences? `https://petewarden.com/2017/10/29/how-do-cnns-deal-with-position-differences/`. Accessed: 2019-03.