Deep Learning Assignment
ID: 582015

1 Perceptron

Our input to the perception model is $T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$. Here, $x_i \in \chi = R^{10}$,$y_i \in Y = \{+1, -1\}, i = 1, 2, \cdots, N$

Hence, our perception model is $f(x) = sign(w \cdot x + b)$, where sign function is the one classifying the sum of inputs as label +1 or -1. The process can be summarized as follows:
1. Choose initial values for weights and bias w_0, b_0
2. Loop through training dataset (x_i, y_i)
3. If $y_i (w \cdot x_i + b) \leq 0$, update w and b

$$\mathbf{w} \rightarrow \mathbf{w} + \frac{\epsilon_w}{2} (v^m - v(\mathbf{u}^m)) \mathbf{u}^m$$
$$\mathbf{b} \rightarrow b - \frac{\epsilon_w}{2} (v^m - v(\mathbf{u}^m))$$

For perception learning, we use $\epsilon = 0.01$ as the learning rate and use threshold 10^{-10}. We use the absolute value of the change of weights as error term. Once it becomes smaller than the threshold, we terminate the learning process. We use a randomly generated dataset of size 100 as training set and for test data, we generate a dataset of size 1000 and calculate its 'performance'(percentage of correctly classified data) over 20 trials.

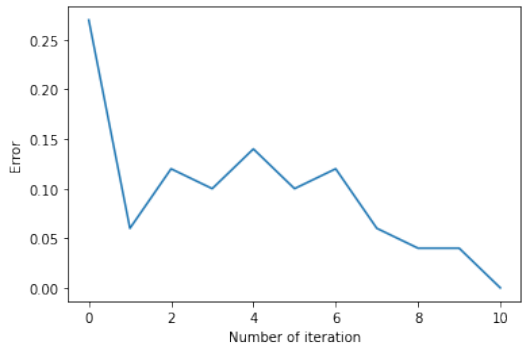

Figure 1: Error of training dataset
task: sum of inputs

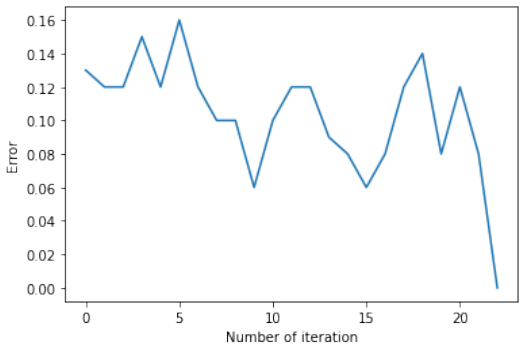

Figure 2: Error of testing dataset
task: product of inputs

Above is two pictures depicting the change of error term defined over iteration in two missions. Both two missions are well solved. In most cases, the perceptron learning converges rapid after a few epochs. Under a few circumstances, the error term cannot converge to zero and oscillate within a certain range. The network built can also be used for larger inputs with N¿10. The second task which calculates the product of inputs have a bigger fluctuation in the error term.

The plots of the performance versus the size of training vectors are shown as follows when doing the sum and parity tasks.

In Figure 3, we can see a clear convergence when $N > 200$. The preceptron learning process will work better when provided with larger training datasets and the performance will approximate 1.0 as expected. In Figure 4, we classify input vectors by the sign of their product $v = sign(\prod_i u_i)$. A very poor performance for the product classification task can be seen and the performance score mainly oscillate between 0.4 and 0.7. A basic explanation on the difference of performances is linear separability. If we simplify the sum task as two-dimensional input, which is the number of +1 and the number of -1. The output of the question can be easily classified into two categories in a 2-D plane

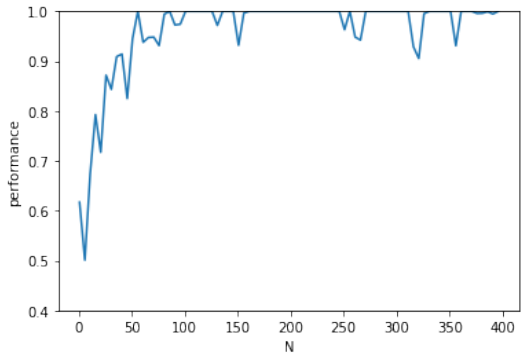

Figure 3: Performance as a function of size of training set in sum-classification

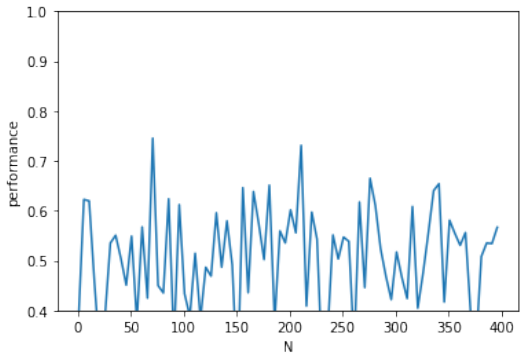

Figure 4: Performance as a function of size of training set in product-classification

with a hyperplane. However, in the following case, we cannot use one single line to separate the two classes and we have at most 75% accuracy. Actually, the perceptron model with threshold will not work well if the case is not linear separable. Our parity problem can be thought of as two inputs and is not linear separable. To better solve this problem, we could use multiple perceptrons or even better, non-linear kernal support vector machine.

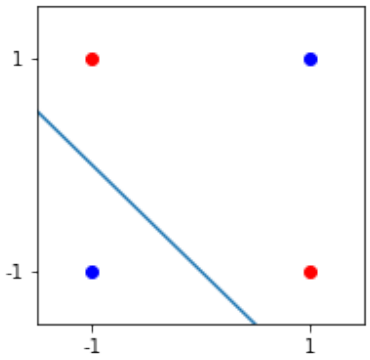

Figure 5: Parity question: linear inseparability

2 Self-supervised Learning

For the perceptron model I constructed, there will be eight input units, three hidden units and eight output units. The algorithm is similar to the backprop network with three layers which can be summarized as follows:

1. Input training data, here we use diagonal matrix of 8 dimensions as input. We use sigmoid function as the activation function as required. Mean squared error (MSE) is used as the cost function $C(\cdot)$.
2. Forward Propagation of inputs.

$$\begin{cases} z^{(l)} = w^{(l)} a^{(l-1)} + b \\ a^{(l)} = \sigma\left(z^{(l)}\right) \end{cases}$$

3. Calculate the error of the output layers.

$$\delta^{(L)} = \nabla_{a^{(L)}} C(\theta) \cdot \sigma'\left(z^{(L)}\right)$$

4. Calculate the backward propagation of errors

$$\delta^{(l)} = \left(\left(w^{(l+1)}\right)^T \delta^{(l+1)}\right) \cdot \sigma'\left(z^{(l)}\right)$$

$l = L-1, L-2, \cdots, 2$

5. Use gradient descending to update the value of weights and biases,$\alpha$is the learning rate, here we set it as 0.5.

$$\begin{cases} w_{jk}^{(l)} := w_{jk}^{(l)} - \alpha \frac{\partial C(\theta)}{\partial w_{jk}^{(l)}} \\ b_j^{(l)} := b_j^{(l)} - \alpha \frac{\partial C(\theta)}{\partial b_j^{(l)}} \end{cases}$$

Although this is a very simple model, it successfully reproduces the input vector with high accuracy. The training loss drops drastically in the first 100 iterations. After 3000 iterations, we could find a quite similar output matrix or vector.

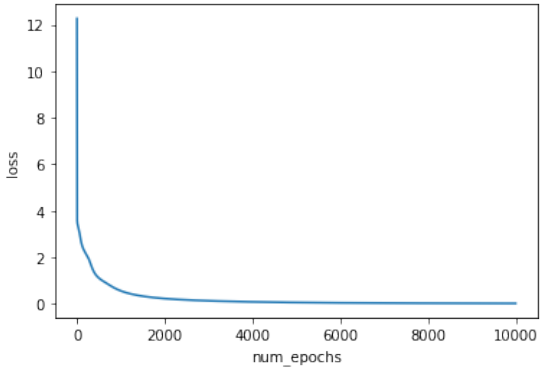

Figure 6: Training loss of eight 8-d vector as a function of number of epoches

Then we consider include momentum term in the change of weights. Momentum, in physics, the product of mass and velocity, enables a particular object with mass to continue in it's trajectory even when an external opposing force is applied. Momentum in neural networks is a variant of the stochastic gradient descent. It replaces the gradient with a momentum which is an aggregate of gradients. Usually, a momentum term can speed up learning and help if the training gets stuck at some local minima.

We use the following formula to update the delta term of the weights.

$$\Delta w_{ij} = \left(\eta * \frac{\partial E}{\partial w_{ij}}\right) + \left(\gamma * \Delta w_{ij}^{t-1}\right)$$

γ is the momentum factor, multiplying the weight increment from the last iteration. Here we take $\gamma = 0.5$

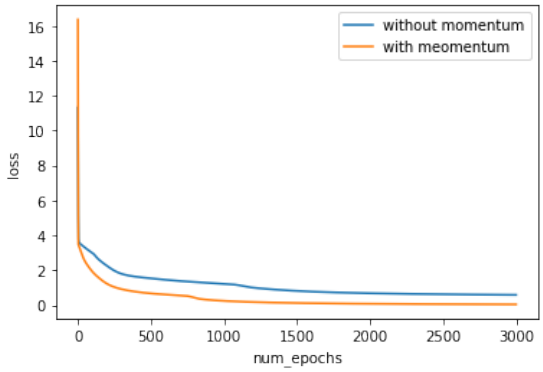

Figure 7: Training loss from two methods: with(orange) and without(blue) momentum term

Obviously, we can see by including momentum term we obtained expected results in fewer iterations. We can also set other values for momentum factor γ but it would make more sense if we keep $\eta + \gamma = 1, \gamma \in (0, 1)$.

A network with 16 inputs can also be trained like this.

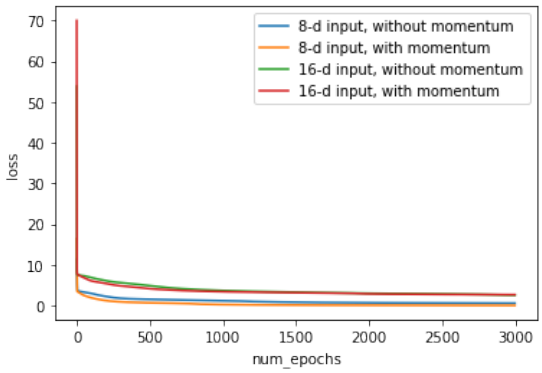

Figure 8: A comparsion of training loss across two methods on 8-d input and 16-d input

We can see a much bigger error in the training of 16-d inputs, but still, using momentum term will speed up training and have similar results. In this case, we did not try with different sets of η and γ, yet we still get fairly good training results. In the next section, we will talk more about hyperparameter optimization.

3 MNIST Classification

Training deep learning models usually takes time and requires computational resources. There are a lot of hyperparameters we need to deel with to get the best neural network, for example:

1. Number of iterations and training epochs, which activation function to use
2. Learning rate, momentum factor
3. Number of layers and units in each layer
4. Batch size, number of training samples

Mainly when we mention hyperparameter tuning, we are referring to hyperparameters to tune in a neural network like number of layers, and hyperparameters involved in the learning algorithm. Most of the time, increasing the units can improve the performance of a model but could also result in overfitting.Learning rate is also very important for an optimizer. A low learning rate will bring about slow convergence or even no optimal solution. A too high learning rate might result in divergence and the model cannot reach optimal solution.
There are many ways of hyperparameter tuning, like *GridSearch*, *RandomSearch* and *Bayesian Optimizers*. Here we will first talk about hyperparameter tuning for MNIST classification using *Random Forest* method.

Random Forest is a supervised machine learning algorithm made up of decision trees. It will draw a random bootstrap sample from the training set and grow a decision tree from the sample. Then it will aggregate the prediction by each tree to assign the class label by majority vote. We use *RandomForestClassifier* from the *sklearn* package to build the classifier. There are several important parameters like n_estimators(the number of trees) and random_state(the randomness of the bootstrapping of the samples used and the sampling of features when looking for best split of the nodes). Here we use $n\_estimators = 100$ and default setting first. We have a cross validation score of 0.96583, which is already a very good result since this is a dataset whose digits are easy to differentiate. We can see many cases 4 is predicted as 9, 3 is predicted as 2, 9 is predicted as 3 and 4, 5 is predicted as 3.
Now we need to adjust the hyperparameters step by step in this model.

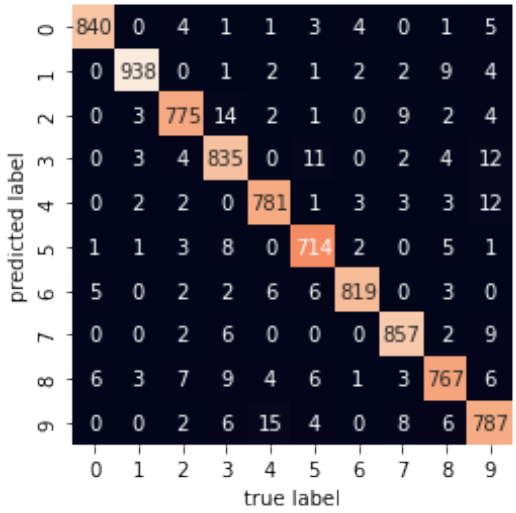

Figure 9: A confusion matrix of true labels and predicted labels of MNIST digits

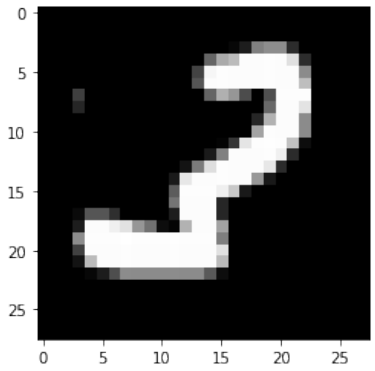

Figure 10: One of the common mistakes: 3 is predicted as 2

- Choose criterion Criterion is highly associated with how the optimum split of features is found and the parameter is used to measure the quality of a split.There are two commonly used criterion, *Entropy* and *Gini*.

$$Gini = 1 - \sum_j p_j^2$$

where p_j is the probability of class j.

$$Entropy = -\sum_j p_j \cdot \log_2 \cdot p_j$$

Entropy measures the disorder of the features with the target. The optimum split is chosen by the feature with less entropy. In this case, we choose Entropy with score of 0.9488 which is a little better than using Gini with score 0.9460. Usually, the obtained results with entropy criterion are slightly better. However, the Gini criterion is less computationally expensive and much faster. In our case, the criterion parameter will not make a difference to the result.

- Choose the number of trees: n_estimators(Figure 11)

  We run the classifier for $n\_estimators = 10, 20, 30...200$ and obtain the best parameter is 140 with score 0.95047796. And we run it for interval (130,150) and the optimum is found at 136.

- Choose max_depth: the maximum depth of the tree(Figure 12) The best parameter found is 10 with score 0.95213842.

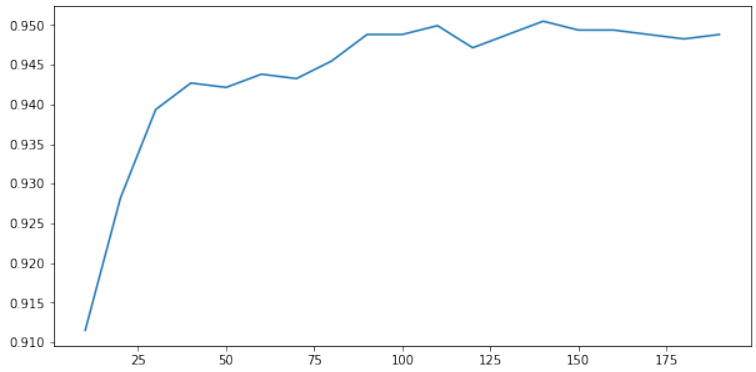

Figure 11: A plot of cross-validation score against n_estimators value

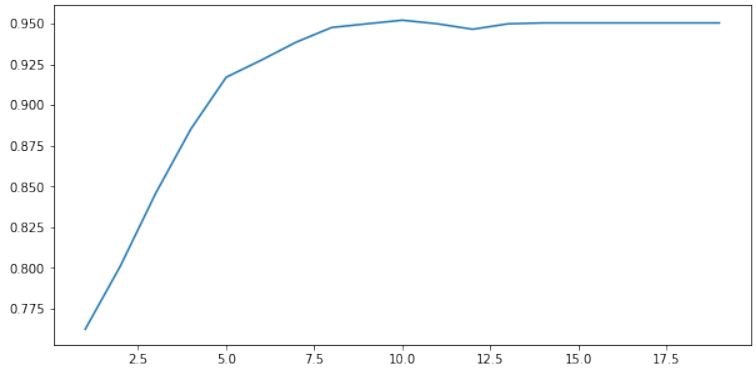

Figure 12: A plot of cross-validation score against max_depth value

- Choose min_samples_split: the minimum number of samples required to split an internal node(Figure 13)

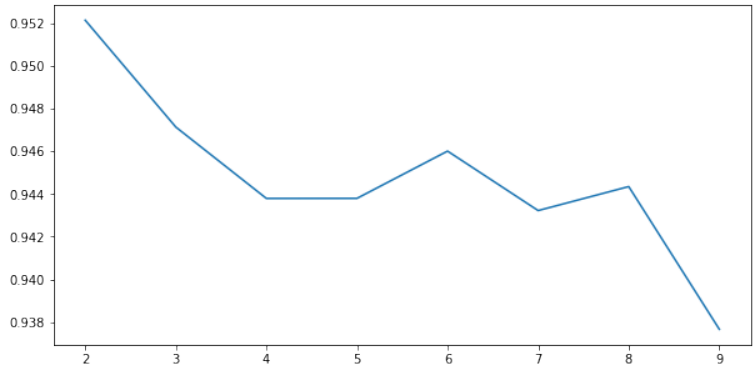

Figure 13: A plot of cross-validation score against min_samples_split value

  The best parameter found is 2 with score 0.95213842.

- Choose min_samples_leaf: the minimum number of samples required to be at a leaf node(Figure 14)
  The best parameter found is 1 with score 0.95213842.

In short summary, we have best parameters **n_estimators = 136, max_depth = 10, min_samples_leaf =1 , min_samples_split =2, criterion = 'entropy',max_features='sqrt'**

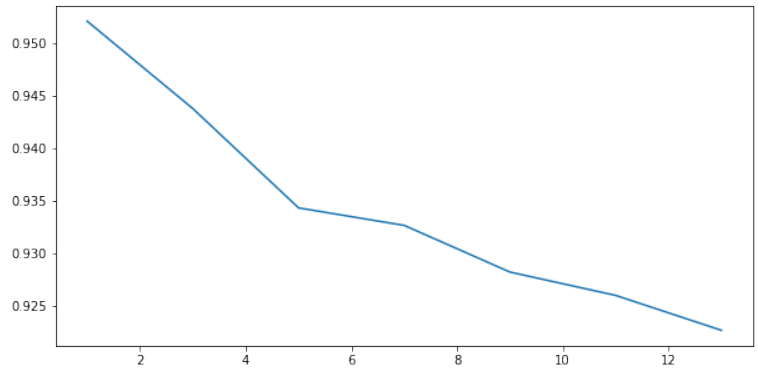

Figure 14: A plot of cross-validation score against min_samples_leaf value

We just implemented manual adjustment for the parameters one by one. However, this is just a temporary solution since we do not know how these parameters influence each other. Hence, we do a grid search in the neighborhood of the optimized parameters we already obtain. Furthermore, we could use Random Search Cross Validation in Scikit-Learn package to specify a grid of hyperparameter ranges and randomly sample from the grid. The result is a little different from above, **n_estimators = 300, max_depth = 14, min_samples_leaf =2 , min_samples_split =4, criterion = 'entropy', max_features='sqrt'**

We now move on to the hyperparameter tuning for using Convolutional Neural Network. There are many choices for CNN architecture. Our purpose is to choose the one with high accuracy while minimizing computational complexity. A typical CNN starts with feature extraction and finishes with classification. Convolutional layers convolve the input and pass its result to the next layer.Pooling layers reduce the dimensions of data by combining the outputs of neuron clusters. Fully connected layers connect every neuron in one layer to every neuron in another layer. The flattened matrix goes through a fully connected layer to classify the images. As mentioned before, there are a range of hyperparameters to be considered. Some are associated with the learning(optimization) algorithm and some are associated with the structure of the neural network. Still, a very explicit way of tuning parameters is to analyse their influence one by one especially when dealing with parameters involved in the CNN structure. First, we consider the number of subsampling pairs we use in the model. Here, pair means the combination of the use of convolution layer and pooling layer. We have the following three common structures starting with 784 units in input layer and ending with 256 units in fully connected dense layer and 10 in output layer.

```
A. [convolution layer: 24 feature maps(5x5 filter),max pooling(2x2 filter,stride 2)]
B. [convolution layer: 24 feature maps(5x5 filter),max pooling(2x2 filter,stride 2)]
   [convolution layer: 48 feature maps(5x5 filter),max pooling(2x2 filter,stride 2)]
C. [convolution layer: 24 feature maps(5x5 filter),max pooling(2x2 filter,stride 2)]
   [convolution layer: 48 feature maps(5x5 filter),max pooling(2x2 filter,stride 2)]
   [convolution layer: 64 feature maps(5x5 filter),max pooling(2x2 filter,stride 2)]
```

From Figure 15, we can see using 2 and 3 pairs of convolution layer and pooling layer achieve almost same accuracy. In order to minimize computational complexity, we choose 2 pairs of (C-P). Then, among different values of feature maps, we choose 32 and 64 in the two convolution layers among the following settings:(8,16),(16,32),(24,48),(32,64),(48,96),(64,128). In the training result, (32,64)(64,128) achieved similar accuracy after 15 epochs. Hence we choose (32,64) as numbers of feature maps. Furthermoafter the dense layers of the networkre, following the same process, we choose 128 units in the fully connected dense layer. 256 and more units will only perform slightly better than 128 units after 15 epochs, hence they are not worth the extra computing cost.
Dropout value is another parameter which plays an important part in preventing overfitting on the

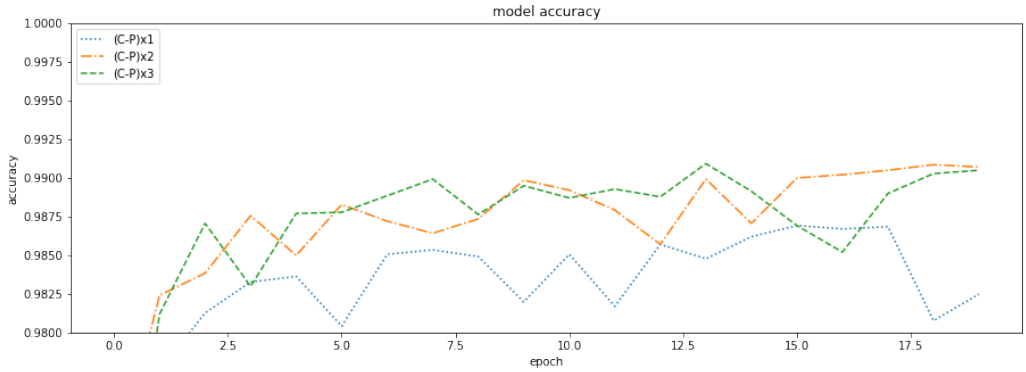

Figure 15: A plot of validation accuracy using three CNN structures

training data. Dropout randomly deactivates some neurons of a layer, thus removing their contribution to the output. Dropouts are mainly used after the dense layers of the network. Here we consider dropout threshold from 0 to 0.5 since taking dropout bigger than 0.5 will lead to poor performance and inaccurate prediction. From Figure 16, we can see 30% dropout is the best. In short summary, we

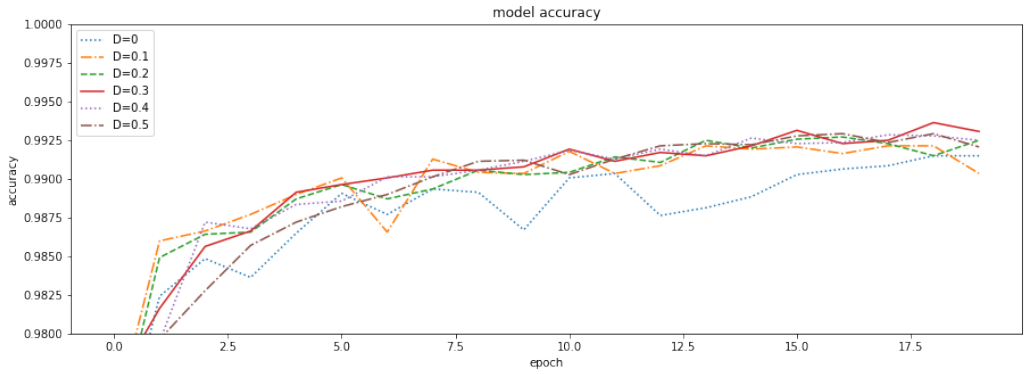

Figure 16: A plot of validation accuracy against models with different dropout values from 0 to 0.5

use two pairs of (C-P) in the CNN model with 32 and 64 feature maps relatively in each convolution layer. After each (C-P), we implement a dropout of 30% before the dense layer. In the fully connected dense layer before the output layer, we set 128 units.
Before we move on to learning rate, there are many available optimizers. They are the algorithm used to minimize the loss/cost. Optimizers in neural networks work by finding the gradient/derivative of the loss with respect to the parameters. The various optimizers differ in how they change the weights. Most popular optimizers are Stochastic Gradient Descent (SGD), Momentum, AdaGrad, RMSProp and Adam. Of all the models constructed above, we use Adam which is in nature a combination of RMSProp and momentum. There are three important hyperparameters involved. One is learning rateϵ. The other two β_1, β_2 are decay rates which are close to 1. We use the following combination of hyperparameters to compile the model. The result is shown as follows:

Learning-Rate	Beta1	Beta2	Optimum-Epoch	Val-Loss
0.1	0.9	0.99	3	0.02544233202934265
0.1	0.9	0.999	2	0.028278963640332222
0.1	0.99	0.99	4	0.029036937281489372
0.1	0.99	0.999	16	0.031150540336966515
0.01	0.9	0.99	15	0.03613648936152458
0.01	0.9	0.999	1	0.03395235165953636
0.01	0.99	0.99	11	0.036354441195726395

0.01	0.99	0.999	10	0.037416618317365646
0.001	0.9	0.99	7	0.04145916923880577
0.001	0.9	0.999	12	0.0379871241748333
0.001	0.99	0.99	16	0.04299500212073326
0.001	0.99	0.999	16	0.043123263865709305

The optimized hyperparameters are $\epsilon = 0.1, \beta_1 = 0.9, \beta_2 = 0.99$
Model loss and accuracy over epochs can be seen from Figure 17 and 18. An accuracy of 0.99857 on the validation sets are obtained, indicating that this optimized model has good performance on accuracy with reasonable computational cost.

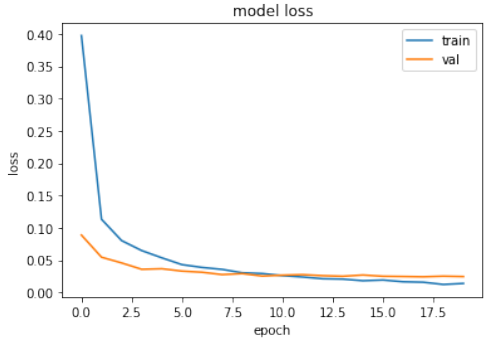

Figure 17: A plot of model loss on training and validation sets with optimized parameter

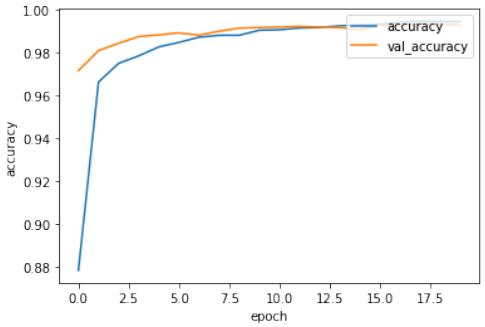

Figure 18: A plot of accuracy and val_accuracy of the optimized CNN model

4 Question 4 Hopfield Network

A Hopfield network is a simple assembly of perceptrons that is able to overcome the XOR problem. It is a fully interconnected neural network where each unit is connected to every other unit. The network stores information in their connectivity matrix W. Matrix W is symmetric and if $w_{ij} = 0$, node i and j are disconnected. The weights indicate how strong the link between nodes are. The higher of the value of the weight, the more likely the two connected nodes will activate simultaneously.
The state s_i takes value in +1,-1. The activation rule is as follows:

$s_i = +1$ if $\sum_j w_{ij} s_j \geq \theta_i$
$s_i = +1$ otherwise

The concept of energy is introduced to measure the performance of the hopfield network.

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j - \sum_i \theta_i s_i == -\frac{1}{2}\vec{s}^T \mathbf{W} \vec{s}^T \vec{\theta}$$

$\vec{s}$ is the input and state vector and $\vec{\theta}$ is the bias vector. The storage capacity of memory is the number of patterns stored in the network. The estimated upper bound depends on the strategy for updating weights. With Hebbian learning, the estimate is around $N \leq 0.15K$, where K is the number of nodes in the network.

We first initialize the network with $\vec{x}_p = (x_{p,1}, x_{p,2}, \cdots, x_{p,J})^T, x_{p,i} = \pm 1$. N patterns are stored in the network.
$w_{ij} = \frac{1}{J}\sum_{p=1}^{N} x_{p,i} x_{p,j}$ for all $i \neq j$. And $w_{ii} = 0$. The Hebbian learning rule can be written in the following form:

$$w_{ij}(t) = w_{ij}(t-1) + \eta x_{t,i} x_{t,j}, \forall i \neq j$$

$t = 1, \cdots, N, w_{ij}(0) = 0, \eta = \frac{1}{J}$

The storage capacity of an associate memory network is $N_{max} = \frac{J}{2 \ln J}$ In our example , $J = 25$ and we generate N patterns($N < J$) to reproduce the input bipolar vectors. We use 5x5 binary bitmap to visualize the plots. In order to measure its performance, we use overall memory accuracy, which is the mean value of the accuracy of recalling all of the bipolar vectors using the Hopfield network. We

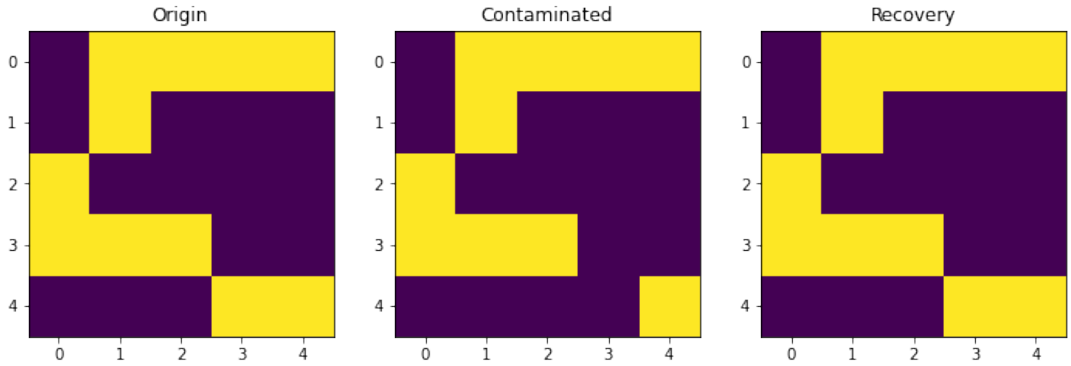

Figure 19: A plot of original pattern, contaminated pattern and recovered pattern

found perfect accuracy when $N \leq 4$ in this case. This proves the upperbound $N_{max} = \frac{J}{2 \ln J}$ mentioned before. All of the input patterns can be restored, even when a contaminated pattern is used, one of whose element is taken the opposite number(from +1 to -1 or from -1 to +1). The model has been tested on different noisy inputs. Overall, when there are 30% distortion, the hebbian learning rule can still almost restore the original pattern. When the distortion rate goes up to 50%, the model shows very bad recall accuracy.
The sparseness of the bipolar pattern does not have big influence on the storage capacity but will affect the overall accuracy. We take p, the proportion of +1, from 0.1 to 0.9 and N from 2 to 10.

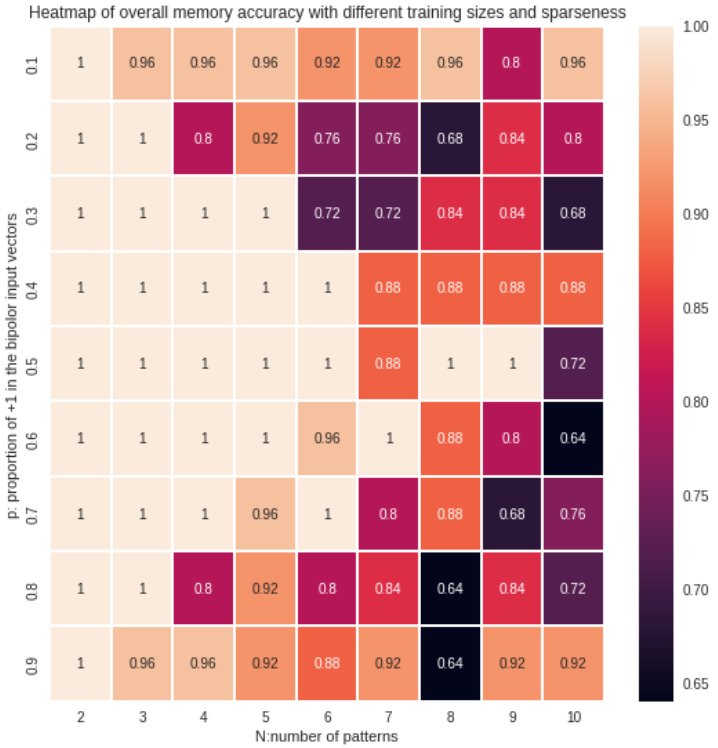

Figure 20: A heatmap of overall accuracy with related to N and p

We can see that when the proportions of +1 and -1 get close to 1:1, the network show better accuracy. Spurious states, which are other stable states different from the fundamental memories, are one major restriction for the Hopfield network being used as associated memory[11]. There are many methods

to improve this by using modified learning rules, like weighted Hebbian learning and perceptron-type learning rule and pseudoinverse learning.
The pseudoinverse rule aims at minimizing the crosstalk between the stored patterns[3]. Based on the hebbian rule weight matrix W, we calculate the pseudoinverse as

$$W_{pinv} = \mathrm{W}^{\mathrm{T}} \cdot \left(W \cdot W^{T}\right)^{-1}$$

The pseudoinverse matrix is calculated using *numpy.linalg.pinv*. This method can to a much extent improve the storage capacity and recall accuracy over the hebbian rule. We mentioned the storage capacity of Hebbian rule is around 0.14J. By using this rule, we achieved an upperbound of J, which is a great improvement in memory capacity. For pattern recall accuracy, no error is detected when trying to recall the original pattern. When a distortion rate of 50% is set, we can still have nearest neighbor of original pattern in the recovered output, which is also much better than the Hebbian rule.
In short summary, the Hopfield network built works with different sizes of patterns and try to recall the input patterns with different learning rules. With Hebbian learning rule, the maximum storage capacity is around 4 when J=25. The accuracy is quite low when working with noisy patterns. But this can be greatly improved by using pseudoinverse rule to update the weight matrix. With this method the storage capacity can be increased to J.

5 AlphaFold2 one year on: a deep learning based model solved protein prediction problem

Introduction

In 2020, DeepMind team won the 14th Critical Assessment of Structural Prediction Competition (CASP14), where scientists strive to predict the structure of several proteins whose structure has not been released. Their model achieved the most accurate of all the submissions for 92.5% of the targets. The deep learning algorithms were first used and developed by DeepMind to tackle the most difficult and complex human game Go, The success of *AlphaGo* could be viewed as a prelude of the team moving forward to work on applicable fields like protein structure prediction problem.

The argument for structure prediction is based on Anfinsen's 'thermodynamic hypothesis,' which asserts that in a physiological context, a protein's lowest free-energy state is unique, and hence its corresponding 3D structure is also unique. Before, laboratory experiments have been the primary source of good protein structure. Over the past decades, Cryo-EM has become the favoured tool of many structural-biology labs[2]. After an initial try at CASP13, DeepMind team applied the latest attention system Transformer instead of Convolutional Neural Network. In addition, instead of using the distances between amino acids directly this time, they used another neural network to generate the 3D coordinates of the atoms directly from the Transformer's output, says JinBo Xu in ISICDM, who is a computational biologiest in Toyota Technological Institute at Chicago. *AlphaFold2* had very high confidence in 36% of the new human protein predictions and some level in 58% of them. Over half of the prediction results are comparable to experimental results, achieving GDT-TS score over 0.9, which is a scoring system of CASP competition to asses predicton models[6]. Also, it is worth mentioning that even atoms in the side chains can be accurately provides.

The creativity of developing *AlphaFold2* lies in the application of novel deep learning approach that combines physical and biological knowledge[5]. The startling accuracy and robustness to a certain extent make protein structure prediction almost a solved problem. The method has triggered off great focus in life sciences community, with promising applications in proteomics, therapeutics and many other fields[4]. As claimed in the Nature new article[2], this program will change everything and empower a new generation of molecular biologists to ask more advanced questions. This essay will briefly discuss about the archietecture of *AlphaFold2* and its influence along with concerns and limitations.

Methodology

AlphaFold achieves predicting the protein structure primarily by predicting the distribution of distances between each pair of amino acids in a protein, and the angles between the chemical bonds that connect them, and then aggregating the measurements for all amino acid pairs into a 2D histogram of distances. And then, it learns from these pictures through CNN to create 3D structure. *AlphaFold2* instead makes use of attention to focus on more details to improve completeness and accuracy. Specifically, it composes of two main parts, which are *EvoFormer* and Structure module[5]. The two modules are optimized jointly, adopted by many recent advances in machine learning[1].

The attention mechanism is the core of both modules. Attention allows neural network to guide information flow by selecting which components of the input must interact with others. In 2017, the original model architecture based on attention mechanism, Transformers, was proposed. In the model, every input token, for example a residue of protein sequence can attend to every other input token, Each token generates a key-query-value triplet and attention function maps a query and key-value pairrs to an output[10].

In the first step, *AlphaFold2* uses input amino acid sequence to query database of protein sequences to obtain multiple sequence alignment(MSA), thus constructing an initial representation of the structure based on the results of MSA, which is called pair representation. The basic idea of MSA is that some parts of the protein is highly conserved. If two amino acids are in close contact, the mutation in the first one might lead the second one to mutate. The feature extracted from MSA gives a contact map model between two amino acids, which is intended to provide more an better reference and priori knowledge to AlphaFold's structure prediction model.

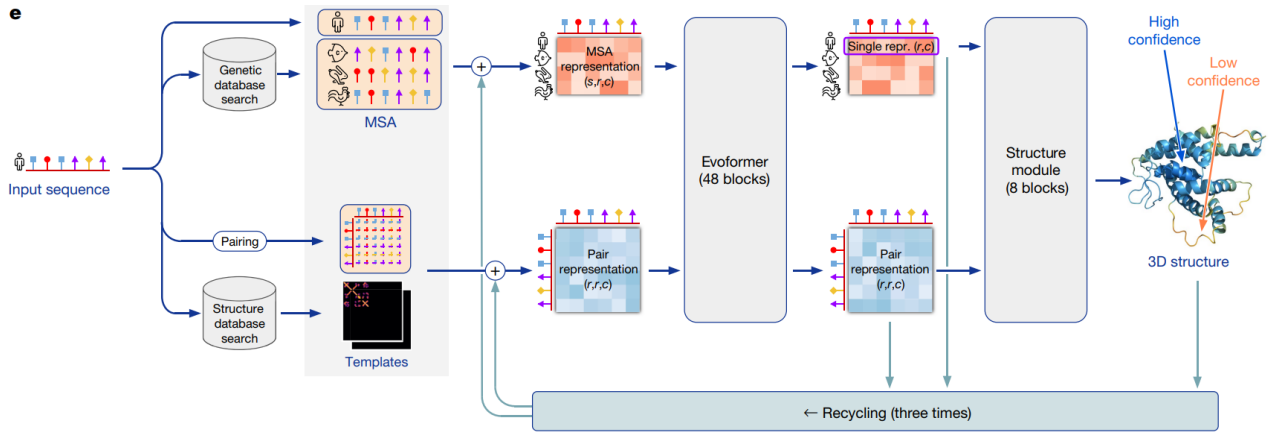

Figure 21: A model architecture flow chart of AlphaFold2 as published in the Nature paper(Jumper et al.,2021)[5]

In *EvoFormer*, it mainly combines graph network and multiple sequence alignment(MSA). The former calculates the distance among amino acids, which indicating the relationship among them, using a designed 'Triangular self-attension network'[5]. Through MSA, the similarity in structures and functions among protein sequences can be inferred. The attention system is able to connect information learnt in the triangle attention to what is learnt in MSA, which gives a broader information than the previous *AlphaFold*. In *Evoformer*, the pair representation, which captures with residues are likely to interact with each other works as both a product and an intermediate layer. The coevolution information from MSA will be passed back to the pair representation, so another structure hypothesis can be generated. Both representations, sequence and structure, exchange information until the network reaches a solid inference.

In Structure Module , every part of the protein can be calculated with 'Invariant Point Attention'(IPA). The structure module considers the protein as a 'residue gas'. Every amino acid is modelled as a triangle, representing the three atoms of the backbone. Starting from some certain atom, a 3D framework as a reference field can be created, which is characterized by numerous independent rotations and translations[5]. An explanation of IPA is well described in the section 1.8 of Supplementary Information in the original Nature paper[5].

Influence, Concerns and Further Work

With an explicit goal of replacing crystallography as a method for determineing protein structure, *AlphaFold2* did a great job in many aspects. Especially, the representations of protein structure is fed through the networks multiple times to refine the structure. Transformer architecture is broadly used in *AlphaFold2*, attention mechanism is well used in updating information from MSA and pair representations.

The advent of Alphafold2 enables better prediction of the probability of protein-molecule binding, thus greatly accelerating the efficiency of new drug development. The availability of high-accuracy predictions will hopefully benefit the biopharmaceutical industry to use AlphaFold's model for development. Furthermore, some applications, such as protein evolutionary analysis, are expected to thrive[2]. *AlphaFold2* brought biophysical insight into the process of protein folding and the intricate structure of proteins. DeepMind researcher Tunyasuvunakool said that there are already cases where AlphaFold has already enabled experimental scientists to work out structures that had eluded them for years. This will not totally replace experimental techniques. AlphaFold can be combined with laboratory work to better interpret protein structures, while saving much time and effort. The great work to a certain extent well solved the protein structure prediction problem. Scientists can move forward to how to make use of this to interpret many prion diseases, such as Creutzfeldt–Jakob disease and Alzheimer's disease.

Moreover, all the original codes of *AlphaFold2* is publicly released on Github and there is a *ColabFold*

version, which is a free and accessible platform for protein folding running on Google Colaboratory[7]. This makes high quality protein structure prediction accessible and in addition gives features to explore the full potential of *AlphaFold2*[7].

Currently, *AlphaFold2* is still expensive in computational cost. This kind of limits its spread among individual researchers. Another important problem is that it much depends on known structures, hence we cannot safely draw the conclusion that the challenge of predicting protein structures has been already tackled. Since it is a deep learning based work, it still much depends on the volume of training samples. Hence, it shows much lower accuracy in predicting some irregular structure. Since its samples from Protein Data Bank(PDB), for proteins without experimental results from PDB, the prediction results are much worse than those PDB resolved ones. A comparison of distribution of average confidence scores for *AlphaFold2* models of human proteins with and without homologs in PDB reveals that predictions for available PDB structures are heavily skewed to higher scores[4]. In contrast, those without homologs in PDB, the predicted local-distance different test(pLDDT)[8] scores are much lower and indicate a very broad range of predicted reliability.

Another outstanding challenge is that, the *AlphaFold2* models cannot be explained or externally proved[4]. Besides experimental results from the lab, the models cannot be independently validated by existing computer programs or methods. There are still many concerns remain unaddressed and await further work. But in all, this is still an incredible achievement for biology researchers, who will no longer need to invest huge amount of fundings and long time to understand the structures of proteins, but can refer to results from *AlphaFold2*.

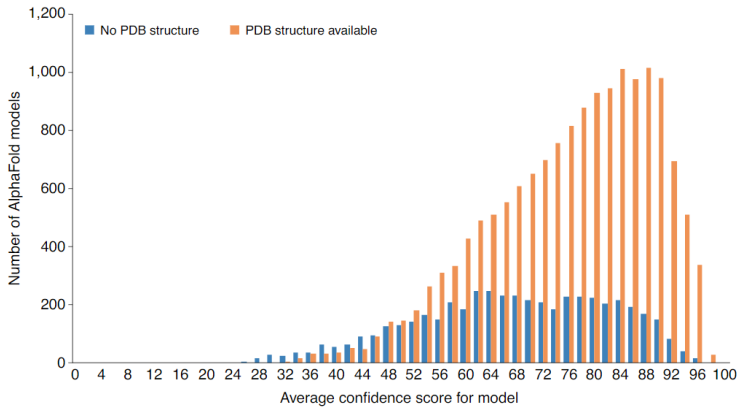

Figure 22: Distribution of average confidence scores for AlphaFold2 models of human proteins with and without PDB homologs[9]

Summary

In summary, *AlphaFold2*, created by DeepMind team, proves to be a milestone in predicting protein structure and resolving protein folding problem. It has already been used to predict structures for some of the proteins in SARS-CoV-2 to better help understand how the mutations have affected the proteins. In the very recent research in Omicron S protein, *AlphaFold2* is used to predict the structure of S, M and N proteins of the Omicron variant and a significant difference is observed which might result in weak recognition by antibodies, explaining potential immune escape and the invalidation of some existing vaccines[12]. The development of new techniques will not come along with the conclusion that a problem to which researchers have developed uncountable time and effort has already been solved. On the contrary, protein structure problem is still an unsolved problem. More novel subproblems will come up and await to be studied on. In the meanwhile, more new facts about biology can be discovered and more effort will be devoted to the application based on already developed methods. It still remains to be seen how much *AlphaFold2* and relevant inspired models can help design proteinss with new functions in the recent future.

References

[1] Nazim Bouatta, Peter Sorger, and Mohammed AlQuraishi. Protein structure prediction by alphafold2: Are attention and symmetries all you need? *Acta Crystallographica Section D: Structural Biology*, 77:982–991, 8 2021.

[2] Ewen Callaway. 'it will change everything': Deepmind's ai makes gigantic leap in solving protein structures. *Nature*, 588:203–204, 12 2020.

[3] R Chandra, Somesh Kumar, and Puneet Goswami. 'implementation of hopfield neural network for its capacity with finger print images. *Int. J. Comput. Appl*, 141(5):44–49, 2016.

[4] David T Jones and Janet M Thornton. Protein structure predictions to atomic accuracy with alphafold, 2022.

[5] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A.A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *undefined*, 596:583–589, 8 2021.

[6] Andriy Kryshtafovych, Torsten Schwede, Maya Topf, Krzysztof Fidelis, and John Moult. Critical assessment of methods of protein structure prediction (casp)-round xiii. *Proteins*, 87(12):1011—1020, December 2019.

[7] Milot Mirdita, Konstantin Schütze, Yoshitaka Moriwaki, Lim Heo, Sergey Ovchinnikov, and Martin Steinegger. Colabfold - making protein folding accessible to all. *bioRxiv*, page 2021.08.15.456425, 10 2021.

[8] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[9] Janet M Thornton, Roman A Laskowski, and Neera Borkakoti. Alphafold heralds a data-driven revolution in biology and medicine. *Nature Medicine*, 27(10):1666–1669, 2021.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[11] Yue Wu, Jianqing Hu, Wei Wu, Yong Zhou, and K.-L Du. Storage capacity of the hopfield network associative memory. 01 2012.

[12] Qiangzhen Yang, Ali Alamdar Shah Syed, Aamir Fahira, and Yongyong Shi. Structural analysis of the sars-cov-2 omicron variant proteins. *Research*, 2021, 2021.

6 Appendix

6.1 Question 1

```
1
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import copy,random
6
7  class perceptron:
8
9      def __init__(self, N=10, gamma=-0.5):
10
11         self.ws = np.zeros(N)
12         self.gamma = gamma
13         self.dim = N
14         self.gamma = 0
15         self.err=[]
16         self.E=[]
17         return
18
19
20     def train(self, N = 100, Print = False, task = 'sum',a=123):
21         np.random.seed(a)
22         train_data = np.random.choice([1,-1], (self.dim,N)) #training
               patterns
23         if task == 'sum': vs = np.sign(np.sum(train_data,0)+0.01)
24         elif task == 'product': vs = np.sign(np.prod(train_data, 0))
25         else:
26             print('Fail!')
27             return
28
29         if Print: print(vs)
30         self.data = train_data
31         self.truth = vs
32
33         self.train_perceptron()
34
35
36         if Print: print(self.ws)
37         return
38
39     def train_perceptron_prop(self, epsilon, thresh = 10**(-10), Print=
           False):
40         oldws = copy.copy(self.ws)
41
42         for i in range(self.data.shape[1]):
43             u = self.data[:,i]
44             self.ws += epsilon/2*(
45                         self.truth[i] -
46                         np.sign(np.dot(self.ws,u)-self.gamma)) * u
47             self.gamma -= epsilon/2*(
48                         self.truth[i] -
```

```
                        np.sign(np.dot(self.ws,u)-self.gamma))

        err=np.sum((np.abs(self.ws-oldws))**1)

        Err=0
        for i in range(self.data.shape[1]):
            u = self.data[:,i]
            Err=Err+(self.truth[i] - np.sign(np.dot(self.ws,u)-self.gamma
                ))

        self.err.append(err)
        self.E.append(Err)

        if Print: print('err:', err)
        if err < thresh: #converged
            return False
        else: return True #should still train

    def train_perceptron(self, epsilon = 0.01, nlim=1000):
        self.ws = np.zeros(self.dim)
        train = True
        n = 0
        error=[]
        while train and n < nlim:
            n += 1
            train = self.train_perceptron_prop(epsilon)


        #print('final gamma:', self.gamma,'final weights:',self.ws)
        #print('Error:',self.err)

        # plt.figure()
        # plt.plot(self.err)
        # plt.xlabel('Number of iteration')
        # plt.ylabel('Error')
        # plt.show()

        return
    def calculate(self, u, Print = False):

        v = np.sign(np.dot(self.ws, u)-self.gamma)
        self.state = v
        if Print:
            print('sum:', np.sum(u))
            print('state:', v)
        return v
    def queries(self, n, plusses='random'):

        if plusses == 'random': queries = np.random.choice([-1, 1], (10,n
            ))

        else:
            queries = np.zeros((10, n))
```

```
            pool = [1 for i in range(plusses)]+[-1 for i in range(10-
                plusses)]
            #print(pool)
            for i in range(n):
                queries[:,i] = np.random.choice(pool, 10, replace=False)
                #print(queries[:,i])


        truth = np.sign(np.sum(queries, 0)+0.001)
        vs = np.sign(np.dot(self.ws, queries)-self.gamma)
        performance = sum(np.array(vs) == np.array(truth))/n

        return performance
    def test_performance(self, Ns = np.arange(1,400, 5), task = 'sum',
                         ntrial = 20, n=1000, Plot=True, plusses='random'
                             ):
        #plusses is nubmer of +1s in input
        performances = []
        for N in Ns:

            performance = []
            for j in range(10):
                self.train(N=N, task = task)
                for i in range(ntrial):
                    performance.append(self.queries(n, plusses=plusses))
            performance = np.mean(performance)
            performances.append(performance)

        if Plot:
            plt.figure()
            plt.plot(Ns, performances)
            plt.xlabel('N')
            plt.ylabel('performance')
            plt.ylim(0.4,1)
            plt.show()

        return Ns, performances

s = perceptron()

s.train(N=100, task='sum',a=1238)

s.test_performance(task='sum')

p = perceptron()
p.train(N=100, task='product',a=2)

p.test_performance(task='product')
```

6.2 Question 2

```

```

```
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  def sigmoid(x):
7      return 1 / (1 + np.exp(-x))
8
9
10 def derivative_sigmoid(x):
11     return sigmoid(x) * (1 - sigmoid(x))
12
13
14 def MSE(y, y_hat):
15     return sum(sum((y - y_hat) ** 2) / 2)
16
17 def accuracy(y, y_hat):
18     cnt = 0
19     y_hat = [1 if i > 0.5 else 0 for i in y_hat]
20     for i in range(len(y)):
21         if y[i] == y_hat[i]:
22             cnt += 1
23     return cnt / len(y)
24
25 if __name__ == '__main__':
26
27     w1 = np.random.normal(size=(3, 8))
28     w2 = np.random.normal(size=(8, 3))
29     b1 = np.ones((3, 8))
30     b2 = np.ones((8, 8))
31
32     X = np.diag([1,1,1,1,1,1,1,1])
33     Y = np.diag([1,1,1,1,1,1,1,1])
34     #Hyperparameter
35     num_epochs = 10000
36     lr = 0.5
37
38     loss_list = []
39     acc_list = []
40     for epoch in range(num_epochs):
41         # forward layer1
42         z1 = np.matmul(w1,X) + b1
43         a1 = sigmoid(z1)
44
45         # forward layer2
46         z2 = np.matmul(w2,a1) + b2
47         a2 = sigmoid(z2)
48
49         # loss function
50         loss = MSE(Y, a2)
51
52
53
54
55
```

```
56          loss_list.append(loss)
57
58          if epoch % 500 == 0:
59              print(f'epoch {epoch}: loss = {loss}')
60
61          delta = (Y - a2) * derivative_sigmoid(z2)
62          w2_grad = np.matmul(delta,a1.T)
63          b2_grad = np.sum(delta, axis=0)
64
65          delta =  derivative_sigmoid(z1)* np.matmul(w2.T,delta)
66          w1_grad = np.matmul(delta,X)
67          b1_grad = np.sum(delta, axis=0)
68
69          w1 += lr * w1_grad
70          b1 += lr * b1_grad
71          w2 += lr * w2_grad
72          b2 += lr * b2_grad
73
74      # predict
75      y = sigmoid(np.matmul(w2,sigmoid(np.matmul(w1,X) + b1))+ b2)
76      print(y)
77
78  loss=loss_list
79  line1 = plt.plot(list(range(num_epochs)), loss_list)
80  plt.ylabel('loss')
81  plt.xlabel('num_epochs')
82
83  if __name__ == '__main__':
84
85      w1 = np.random.normal(size=(3, 8))
86      w2 = np.random.normal(size=(8, 3))
87      b1 = np.ones((3, 8))
88      b2 = np.ones((8, 8))
89
90      X = np.diag([1,1,1,1,1,1,1,1])
91      Y = np.diag([1,1,1,1,1,1,1,1])
92
93      num_epochs = 10000
94      lr = 0.5
95      gamma=0.5
96
97
98      # train
99
100     loss_list2 = []
101
102     for epoch in range(num_epochs):
103         # forward layer1
104         z1 = np.matmul(w1,X) + b1
105         a1 = sigmoid(z1)
106
107         # forward layer2
108         z2 = np.matmul(w2,a1) + b2
```

```
109             a2 = sigmoid(z2)
110
111             # loss function
112             loss = MSE(Y, a2)
113             loss_list2.append(loss)
114
115             if epoch % 500 == 0:
116                 print(f'epoch {epoch}: loss = {loss}')
117
118             delta = (Y - a2) * derivative_sigmoid(z2)
119             last_w2_grad=w2_grad
120             w2_grad = np.matmul(delta,a1.T)
121             b2_grad = np.sum(delta, axis=0)
122
123             delta =  derivative_sigmoid(z1)* np.matmul(w2.T,delta)
124             last_w1_grad=w1_grad
125             w1_grad = np.matmul(delta,X)
126             b1_grad = np.sum(delta, axis=0)
127
128             w1 += lr * w1_grad+gamma*last_w1_grad
129             b1 += lr * b1_grad
130             w2 += lr * w2_grad+gamma*last_w2_grad
131             b2 += lr * b2_grad
132         # predict
133         y = sigmoid(np.matmul(w2,sigmoid(np.matmul(w1,X) + b1))+ b2)
134         print(y)
135
136 fig= plt.figure()
137 line = plt.plot(list(range(3000)), loss_list[:3000],label='without
        momentum')
138 line2 = plt.plot(list(range(3000)), loss_list2[:3000],label='with
        meomentum')
139 #line3 = plt.plot(list(range(3000)), loss_list3[:3000])
140 plt.ylabel('loss')
141 plt.xlabel('num_epochs')
142 plt.legend()
143 plt.show()
144
145 if __name__ == '__main__':
146
147     w1 = np.random.normal(size=(3, 16))
148     w2 = np.random.normal(size=(16, 3))
149     b1 = np.ones((3, 16))
150     b2 = np.ones((16, 16))
151     X = np.diag([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
152     Y = np.diag([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
153
154
155     num_epochs = 10000
156     lr = 0.5
157     gamma=0.5
158
159
```

```
    loss_list16 = []

    for epoch in range(num_epochs):
        # forward layer1
        z1 = np.matmul(w1,X) + b1
        a1 = sigmoid(z1)

        # forward layer2
        z2 = np.matmul(w2,a1) + b2
        a2 = sigmoid(z2)

        # loss function
        loss = MSE(Y, a2)

        loss_list16.append(loss)

        if epoch % 500 == 0:
            print(f'epoch {epoch}: loss = {loss}')


        delta = (Y - a2) * derivative_sigmoid(z2)

        w2_grad = np.matmul(delta,a1.T)
        b2_grad = np.sum(delta, axis=0)

        delta =  derivative_sigmoid(z1)* np.matmul(w2.T,delta)

        w1_grad = np.matmul(delta,X)
        b1_grad = np.sum(delta, axis=0)


        w1 += lr * w1_grad
        b1 += lr * b1_grad
        w2 += lr * w2_grad
        b2 += lr * b2_grad

    # predict
    y = sigmoid(np.matmul(w2,sigmoid(np.matmul(w1,X) + b1))+ b2)
    print(y)

fig= plt.figure()
line = plt.plot(list(range(3000)), loss_list[:3000],label='8-d input, without momentum')
line2 = plt.plot(list(range(3000)), loss_list2[:3000],label='8-d input, with momentum')
line16= plt.plot(list(range(3000)), loss_list16[:3000],label='16-d input, without momentum')
line_16= plt.plot(list(range(3000)), loss_list_16[:3000],label='16-d input, with momentum')
#line3 = plt.plot(list(range(3000)), loss_list3[:3000])
plt.ylabel('loss')
plt.xlabel('num_epochs')
plt.legend()
```

```
209 plt.show()
```

6.3 Question 3

```
1
2  #MNIST Random Forest
3  import numpy as np
4  import pandas as pd
5  import matplotlib.pyplot as plt
6  # %matplotlib inline
7  from sklearn.ensemble import RandomForestClassifier
8  from sklearn.metrics import accuracy_score
9  from sklearn.metrics import confusion_matrix
10
11 from google.colab import drive
12 drive.mount('/content/drive')
13
14 path='/content/drive/My Drive/mnist'
15
16 import os
17 os.chdir(path)
18 os.listdir(path)
19
20 train_file = pd.read_csv('train.csv')
21 test_file = pd.read_csv('test.csv')
22
23 np.sort(train_file.label.unique())
24
25 #define the number of samples for training set and for validation set
26 num_train,num_validation = int(len(train_file)*0.8),int(len(train_file)
       *0.2)
27
28 num_train,num_validation
29
30 #generate training data from train_file
31 x_train,y_train=train_file.iloc[:num_train,1:].values,train_file.iloc[:
       num_train,0].values
32 x_validation,y_validation=train_file.iloc[num_train:,1:].values,
       train_file.iloc[num_train:,0].values
33
34 print(x_train.shape)
35 print(y_train.shape)
36 print(x_validation.shape)
37 print(y_validation.shape)
38
39 index=3
40 print("Label: " + str(y_train[index]))
41 plt.imshow(x_train[index].reshape((28,28)),cmap='gray')
42 plt.show()
43
44 clf=RandomForestClassifier()
45 clf.fit(x_train,y_train)
46
```

```
47 prediction_validation = clf.predict(x_validation)
48 print("Validation Accuracy: " + str(accuracy_score(y_validation,
       prediction_validation)))
49
50 import seaborn as sns
51 from sklearn.metrics import confusion_matrix
52 mat = confusion_matrix(y_validation,prediction_validation)
53 sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
54 plt.xlabel('true label')
55 plt.ylabel('predicted label')
56
57 index=3
58 print("Predicted " + str(y_validation[y_validation!=prediction_validation
       ][index]) + " as " +
59       str(prediction_validation[y_validation!=prediction_validation][index
           ]))
60 plt.imshow(x_validation[y_validation!=prediction_validation][index].
       reshape((28,28)),cmap='gray')
61
62 from sklearn.datasets import load_digits
63 digits = load_digits()
64
65 fig = plt.figure(figsize=(6,6))
66 fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace
       =0.05)
67 for i in range(64):
68     ax = fig.add_subplot(8,8,i+1, xticks=[], yticks=[])
69     ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='
           nearest')
70     ax.text(0,7,str(digits.target[i]))
71
72 from sklearn.ensemble import RandomForestClassifier
73 from sklearn.datasets import load_digits
74 from sklearn.model_selection import train_test_split,GridSearchCV,
       cross_val_score
75 from sklearn.metrics import accuracy_score
76 import matplotlib.pyplot as plt
77 import numpy as np
78 data = load_digits()
79 x = data.data
80 y = data.target
81 RF = RandomForestClassifier(random_state = 66)
82 score = cross_val_score(RF,x,y,cv=10).mean()
83 print('cross_val_score is: %.4f'%score)
84
85 RF.fit(x,y)
86 y_=RF.predict(x)
87
88 RF = RandomForestClassifier(random_state = 66)
89 score = cross_val_score(RF,x,y,cv=10).mean()
90 print('Gini: %.4f'%score)
91 RF = RandomForestClassifier(criterion = 'entropy',random_state = 66)
92 score = cross_val_score(RF,x,y,cv=10).mean()
```

```
93  print('Entropy: %.4f'%score)
94
95  ScoreAll = []
96  for i in range(10,200,10):
97      DT = RandomForestClassifier(n_estimators = i,criterion = 'entropy',
            random_state = 66) #,
98      score = cross_val_score(DT,data.data,data.target,cv=10).mean()
99      ScoreAll.append([i,score])
100 ScoreAll = np.array(ScoreAll)
101
102 max_score = np.where(ScoreAll==np.max(ScoreAll[:,1]))[0][0]
103 print("best parameters with score:",ScoreAll[max_score])
104
105 plt.figure(figsize=[10,5])
106 plt.plot(ScoreAll[:,0],ScoreAll[:,1])
107 plt.show()
108
109 ScoreAll = []
110 for i in range(130,150):
111     DT = RandomForestClassifier(n_estimators = i,random_state = 66,
            criterion = 'entropy')
112     score = cross_val_score(DT,data.data,data.target,cv=10).mean()
113     ScoreAll.append([i,score])
114 ScoreAll = np.array(ScoreAll)
115
116 max_score = np.where(ScoreAll==np.max(ScoreAll[:,1]))[0][0]
117 print("best parameters with score:",ScoreAll[max_score])
118 plt.figure(figsize=[10,5])
119 plt.plot(ScoreAll[:,0],ScoreAll[:,1])
120 plt.show()
121
122 ScoreAll = []
123 for i in range(1,20):
124     DT = RandomForestClassifier(n_estimators = 136,random_state = 66,
            max_depth =i  ,criterion = 'entropy') #,criterion = 'entropy'
125     score = cross_val_score(DT,data.data,data.target,cv=10).mean()
126     ScoreAll.append([i,score])
127 ScoreAll = np.array(ScoreAll)
128
129 max_score = np.where(ScoreAll==np.max(ScoreAll[:,1]))[0][0]
130 print("best parameters with score:",ScoreAll[max_score])
131 plt.figure(figsize=[10,5])
132 plt.plot(ScoreAll[:,0],ScoreAll[:,1])
133 plt.show()
134
135 ScoreAll = []
136 for i in range(2,10):
137     RF = RandomForestClassifier(n_estimators = 136,random_state = 66,
            max_depth =10,min_samples_split = i,criterion = 'entropy' ) #,
            criterion = 'entropy'
138     score = cross_val_score(RF,data.data,data.target,cv=10).mean()
139     ScoreAll.append([i,score])
140 ScoreAll = np.array(ScoreAll)
```

```
141
142 max_score = np.where(ScoreAll==np.max(ScoreAll[:,1]))[0][0]
143 print("best parameters with score:",ScoreAll[max_score])
144 plt.figure(figsize=[10,5])
145 plt.plot(ScoreAll[:,0],ScoreAll[:,1])
146 plt.show()
147
148 ScoreAll = []
149 for i in range(1,15,2):
150     DT = RandomForestClassifier(n_estimators = 136,random_state = 66,
        max_depth =10,min_samples_leaf = i,min_samples_split = 2,criterion
        = 'entropy' )
151     score = cross_val_score(DT,data.data,data.target,cv=10).mean()
152     ScoreAll.append([i,score])
153 ScoreAll = np.array(ScoreAll)
154
155 max_score = np.where(ScoreAll==np.max(ScoreAll[:,1]))[0][0]
156 print("best parameters with score:",ScoreAll[max_score])
157 plt.figure(figsize=[10,5])
158 plt.plot(ScoreAll[:,0],ScoreAll[:,1])
159 plt.show()
160
161 param_grid = {
162     'max_features':np.arange(0.1, 1)}
163
164 rfc = RandomForestClassifier(random_state=66,n_estimators = 136,max_depth
    = 10,min_samples_leaf =1 ,min_samples_split =2,criterion = 'entropy'
    )
165 GS = GridSearchCV(rfc,param_grid,cv=10)
166 GS.fit(data.data,data.target)
167 print(GS.best_params_)
168 print(GS.best_score_)
169
170 param_grid = {
171   'n_estimators':np.arange(130, 140),
172   'max_depth':np.arange(5, 15),
173   'min_samples_leaf':np.arange(1, 3),
174   'min_samples_split':np.arange(2, 5),
175
176 }
177
178 rfc = RandomForestClassifier(random_state=66)
179 GS = GridSearchCV(rfc,param_grid,cv=10)
180 GS.fit(data.data,data.target)
181
182 print(GS.best_params_)
183 print(GS.best_score_)
```

```
1
2
3
4 import os
5 import matplotlib.pyplot as plt
6 import numpy as np
```

```
7  import tensorflow as tf
8  from tensorflow import keras
9
10 # Load MNIST dataset.
11 mnist = keras.datasets.mnist
12 (X_train, y_train), (X_test, y_test) = mnist.load_data()
13
14 print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
15
16 # Reshape the dataset into a 2D array.
17 X_train = X_train.reshape(60000, 28*28)
18 X_test = X_test.reshape(10000, 28*28)
19
20 # Reduce the data size to save time for training.
21 datasize_train = 10000;
22 datasize_test = 3000;
23 X_train = X_train[0:datasize_train, :]
24 X_test = X_test[0:datasize_test, :]
25 y_train = y_train[0:datasize_train]
26 y_test = y_test[0:datasize_test]
27 print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
28
29 from sklearn.ensemble import RandomForestClassifier
30 from sklearn.model_selection import RandomizedSearchCV
31
32 param_space = {"bootstrap": [True],
33         "max_depth": [6, 8, 10, 12, 14],
34         "max_features": ['auto', 'sqrt','log2'],
35         "min_samples_leaf": [2, 3, 4],
36         "min_samples_split": [2, 3, 4, 5],
37         "n_estimators": [100, 200, 300, 400, 500, 600, 700, 800, 900,
               1000]
38 }
39
40 forest_clf = RandomForestClassifier()
41
42 forest_rand_search = RandomizedSearchCV(forest_clf, param_space, n_iter
       =32,
43                                         scoring="accuracy", verbose=True,
                                               cv=5,
44                                         n_jobs=-1, random_state=42)
45
46 forest_rand_search.fit(X_train, y_train)
47
48 # Use Random Forest model.
49 RandomizedSearchCV(cv=5, error_score='raise-deprecating',
50                    estimator=RandomForestClassifier(bootstrap=True,
                           class_weight=None,
51                                                     criterion='gini', max_depth
                                                           =None,
52                                                     max_features='auto',
53                                                     max_leaf_nodes=None,
54                                                     min_impurity_decrease=0.0,
```

```
                                                min_samples_leaf=1,
                                                min_samples_split=2,
                                                min_weight_fraction_leaf
                                                    =0.0,
                                                n_estimators='warn', n_jobs
                                                    =None,
                                                oob_score=False,
                                                random_state=None, verbose
                                                    =0,
                                                warm_start=False),
                                   n_jobs=-1,
                                   param_distributions={'bootstrap': [True],
                                                        'max_depth': [6, 8, 10, 12, 14],
                                                        'max_features': ['auto', 'sqrt',
                                                                         'log2'],
                                                        'min_samples_leaf': [2, 3, 4],
                                                        'min_samples_split': [2, 3, 4,
                                                            5],
                                                        'n_estimators': [100, 200, 300,
                                                            400,
                                                                         500, 600, 700,
                                                                             800,
                                                                         900, 1000]},
                                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                                   return_train_score=False, scoring='accuracy', verbose=
                                       True)

forest_rand_search.best_params_

forest_rand_search.best_estimator_
```

```
# MNIST CNN


from google.colab import drive
drive.mount('/content/drive')

path='/content/drive/My Drive/mnist'

import os
os.chdir(path)
os.listdir(path)

# LOAD LIBRARIES
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D,
    AvgPool2D, BatchNormalization, Reshape
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import LearningRateScheduler
```

```
22 import matplotlib.pyplot as plt
23
24 # LOAD THE DATA
25 train = pd.read_csv("train.csv")
26 test = pd.read_csv("test.csv")
27
28 # PREPARE DATA FOR NEURAL NETWORK
29 Y_train = train["label"]
30 X_train = train.drop(labels = ["label"],axis = 1)
31 X_train = X_train / 255.0
32 X_test = test / 255.0
33 X_train = X_train.values.reshape(-1,28,28,1)
34 X_test = X_test.values.reshape(-1,28,28,1)
35 Y_train = to_categorical(Y_train, num_classes = 10)
36
37 # GLOBAL VARIABLES
38 annealer = LearningRateScheduler(lambda x: 1e-3 * 0.95 ** x, verbose=0)
39
40 nets = 3
41 model = [0] *nets
42
43 for j in range(3):
44     model[j] = Sequential()
45     model[j].add(Conv2D(24,kernel_size=5,padding='same',activation='relu'
         ,
46                 input_shape=(28,28,1)))
47     model[j].add(MaxPool2D())
48     if j>0:
49         model[j].add(Conv2D(48,kernel_size=5,padding='same',activation='
             relu'))
50         model[j].add(MaxPool2D())
51     if j>1:
52         model[j].add(Conv2D(64,kernel_size=5,padding='same',activation='
             relu'))
53         model[j].add(MaxPool2D(padding='same'))
54     model[j].add(Flatten())
55     model[j].add(Dense(256, activation='relu'))
56     model[j].add(Dense(10, activation='softmax'))
57     model[j].compile(optimizer="adam", loss="categorical_crossentropy",
         metrics=["accuracy"])
58
59 X_train2, X_val2, Y_train2, Y_val2 = train_test_split(X_train, Y_train,
     test_size = 0.333)
60 # TRAIN NETWORKS
61 history = [0] * nets
62 names = ["(C-P)x1","(C-P)x2","(C-P)x3"]
63 epochs = 20
64 for j in range(nets):
65     history[j] = model[j].fit(X_train2,Y_train2, batch_size=80, epochs =
         epochs,
66         validation_data = (X_val2,Y_val2), callbacks=[annealer], verbose
             =0)
67     print("CNN {0}: Epochs={1:d}, Train accuracy={2:.5f}, Validation
```

```
        accuracy={3:.5f}".format(
68          names[j],epochs,max(history[j].history['accuracy']),max(history[j
                ].history['val_accuracy']) ))
69
70  styles=[':','-.','--','-',':','-.','--','-',':','-.','--','-']
71  plt.figure(figsize=(15,5))
72  for i in range(nets):
73      plt.plot(history[i].history['val_accuracy'],linestyle=styles[i])
74  plt.title('model accuracy')
75  plt.ylabel('accuracy')
76  plt.xlabel('epoch')
77  plt.legend(["A",'B','C'], loc='upper left')
78  axes = plt.gca()
79  axes.set_ylim([0.98,1])
80  plt.show()
81
82  nets = 6
83  model = [0] *nets
84
85  for j in range(6):
86      model[j] = Sequential()
87      model[j].add(Conv2D(32,kernel_size=5,activation='relu',input_shape
            =(28,28,1)))
88      model[j].add(MaxPool2D())
89      model[j].add(Conv2D(64,kernel_size=5,activation='relu'))
90      model[j].add(MaxPool2D())
91      model[j].add(Flatten())
92      if j>0:
93          model[j].add(Dense(2**(j+4), activation='relu'))
94      model[j].add(Dense(10, activation='softmax'))
95      model[j].compile(optimizer="adam", loss="categorical_crossentropy",
            metrics=["accuracy"])
96
97  X_train2, X_val2, Y_train2, Y_val2 = train_test_split(X_train, Y_train,
        test_size = 0.333)
98  # TRAIN NETWORKS
99  history = [0] * nets
100 names = ["0N","32N","64N","128N","256N","512N"]
101 epochs = 20
102 for j in range(nets):
103     history[j] = model[j].fit(X_train2,Y_train2, batch_size=80, epochs =
            epochs,
104         validation_data = (X_val2,Y_val2), callbacks=[annealer], verbose
                =0)
105     print("CNN {0}: Epochs={1:d}, Train accuracy={2:.5f}, Validation
            accuracy={3:.5f}".format(
106         names[j],epochs,max(history[j].history['accuracy']),max(history[j
                ].history['val_accuracy']) ))
107
108 plt.figure(figsize=(15,5))
109 for i in range(nets):
110     plt.plot(history[i].history['val_accuracy'],linestyle=styles[i])
111 plt.title('model accuracy')
```

```
112 plt.ylabel('accuracy')
113 plt.xlabel('epoch')
114 plt.legend(names, loc='upper left')
115 axes = plt.gca()
116 axes.set_ylim([0.98,1])
117 plt.show()
118
119 nets = 6
120 model = [0] *nets
121
122 for j in range(6):
123     model[j] = Sequential()
124     model[j].add(Conv2D(32,kernel_size=5,activation='relu',input_shape
        =(28,28,1)))
125     model[j].add(MaxPool2D())
126     model[j].add(Dropout(j*0.1))
127     model[j].add(Conv2D(64,kernel_size=5,activation='relu'))
128     model[j].add(MaxPool2D())
129     model[j].add(Dropout(j*0.1))
130     model[j].add(Flatten())
131     model[j].add(Dense(128, activation='relu'))
132     model[j].add(Dropout(j*0.1))
133     model[j].add(Dense(10, activation='softmax'))
134     model[j].compile(optimizer="adam", loss="categorical_crossentropy",
        metrics=["accuracy"])
135
136 # CREATE VALIDATION SET
137 X_train2, X_val2, Y_train2, Y_val2 = train_test_split(X_train, Y_train,
    test_size = 0.333)
138 # TRAIN NETWORKS
139 history = [0] * nets
140 names = ["D=0","D=0.1","D=0.2","D=0.3","D=0.4","D=0.5"]
141 epochs = 20
142 for j in range(nets):
143     history[j] = model[j].fit(X_train2,Y_train2, batch_size=80, epochs =
        epochs,
144         validation_data = (X_val2,Y_val2), callbacks=[annealer], verbose
            =0)
145     print("CNN {0}: Epochs={1:d}, Train accuracy={2:.5f}, Validation
        accuracy={3:.5f}".format(
146         names[j],epochs,max(history[j].history['accuracy']),max(history[j
            ].history['val_accuracy']) ))
147
148 # PLOT ACCURACIES
149 plt.figure(figsize=(15,5))
150 for i in range(nets):
151     plt.plot(history[i].history['val_accuracy'],linestyle=styles[i])
152 plt.title('model accuracy')
153 plt.ylabel('accuracy')
154 plt.xlabel('epoch')
155 plt.legend(names, loc='upper left')
156 axes = plt.gca()
157 axes.set_ylim([0.98,1])
```

```
158 plt.show()
159
160 #ADAM Parameter tuning
161 import itertools
162
163 from tensorflow.keras.optimizers import Adam
164 lr_list = [0.1, 0.01, 0.001]
165 beta1_list = [0.9, 0.99]
166 beta2_list = [0.99, 0.999]
167
168 grid = [lr_list, beta1_list, beta2_list]
169
170 print("{:<14} {:<14} {:<14} {:<14} {:<14}".format('Learning-Rate', 'Beta1
        ', 'Beta2', 'Optimum-Epoch', 'Val-Loss'))
171
172 for lr, beta1, beta2 in itertools.product(*grid):
173     adam = Adam(lr=lr, decay=1e-6, beta_1=beta1, beta_2=beta2)
174     model.compile(loss='categorical_crossentropy', optimizer=adam)
175     history = model.fit(X_train2,Y_train2, batch_size=80, epochs = 20,
176         validation_data = (X_val2,Y_val2), callbacks=[annealer], verbose
            =0)
177
178     best_epoch = np.argsort(history.history['val_loss'])[0]
179     print("{:<14} {:<14} {:<14} {:<14} {:<14}".format(lr,beta1, beta2,
        best_epoch, history.history['val_loss'][best_epoch]))
180
181 #Best model
182 model = Sequential()
183 model.add(Conv2D(32,kernel_size=5,activation='relu',input_shape=(28,28,1)
    ))
184 model.add(MaxPool2D())
185 model.add(Dropout(0.3))
186 model.add(Conv2D(64,kernel_size=5,activation='relu'))
187 model.add(MaxPool2D())
188 model.add(Dropout(0.3))
189 model.add(Flatten())
190 model.add(Dense(128, activation='relu'))
191 model.add(Dropout(0.3))
192 model.add(Dense(10, activation='softmax'))
193 adam = Adam(lr=0.1, decay=1e-6, beta_1=0.9, beta_2=0.99)
194 model.compile(loss='categorical_crossentropy', optimizer=adam)
195
196 history = model.fit(X_train2,Y_train2, batch_size=80, epochs = 20,
197         validation_data = (X_val2,Y_val2), callbacks=[annealer], verbose
            =0)
198
199 test_loss = model.evaluate(X_val2,Y_val2, batch_size=80)
200 predictions = model.predict(X_val2, batch_size=80)
201 predictions = np.argmax(predictions, axis=1) # change encoding again
202
203 Y_pred=np.zeros((predictions.shape[0],10))
204
205 Y_pred[range(predictions.shape[0]),predictions]=1
```

```
206
207 print('Accuracy:', (Y_val2 == Y_pred).sum() / (Y_pred.shape[0]*10))
208
209 plt.plot(history.history['loss'])
210 plt.plot(history.history['val_loss'])
211 plt.title('model loss')
212 plt.ylabel('loss')
213 plt.xlabel('epoch')
214 plt.legend(['train', 'val'], loc='upper right')
215 plt.show()
216
217 plt.plot(history.history['accuracy'])
218 plt.plot(history.history['val_accuracy'])
219
220 plt.ylabel('accuracy')
221 plt.xlabel('epoch')
222 plt.legend(['accuracy', 'val_accuracy'], loc='upper right')
223 plt.show()
```

6.4 Question 4

```
1
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib.lines as mlines
6 import matplotlib.patches as mpatches
7 import copy
8 # patterns = np.array([
9 #     [-1,-1,-1,-1,1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,-1,-1,1.],
      # Letter D
10 #     [-1,-1,-1,-1,-1,1,1,1,-1,1,1,1,1,-1,1,-1,1,1,-1,1,-1,-1,-1,1,1.],
       # Letter J
11 #     [1,-1,-1,-1,-1,-1,1,1,1,1,-1,1,1,1,1,-1,1,1,1,1,1,-1,-1,-1,-1.],
        # Letter C
12 #     [-1,1,1,1,-1,-1,-1,1,-1,-1,-1,1,-1,1,-1,-1,1,1,1,-1,-1,1,1,1,-1.],
13 #     [-1,1,-1,-1,-1,-1,-1,1,-1,1,1,-1,1,1,1,1,-1,1,-1,-1,1,1,-1,1,1.]],
14
15
16 #     dtype=np.float)
17 result=np.zeros((9,9))
18
19 def gen_pattern(N,J,p):
20     patterns=[]
21     for i in range(N):
22       patterns.append(np.random.choice([-1,1],J,p=[1-0.1*p,0.1*p]))
23     patterns=np.array(patterns)
24     return patterns
25
26 def gen(N,J):
27     patterns=[]
28     for i in range(N):
29       patterns.append(np.random.choice([-1,1],J))
```

```
30      patterns=np.array(patterns)
31      return patterns
32
33  def hebbian(m, patterns):
34          weights = np.zeros((m,m))
35          for i in range(m-1):
36              for j in range(i+1,m):
37                  weights[i,j] = eta*np.dot(patterns[:,i], patterns[:,j])
38                  weights[j,i] = weights[i,j]
39          return weights
40
41  for p in range(1,10):
42      for N in range(2,11):
43          patterns=gen_pattern(N,25,p)
44          n = patterns.shape[0]
45          m = patterns.shape[1]
46          eta = 1./n
47
48          # training
49          weights= hebbian(m, patterns)
50          #weights=np.linalg.pinv(weights)
51
52
53
54          for a in range(len(patterns)):
55  #a=np.random.choice(range(len(patterns)))
56              accuracy=[]
57              states = np.array(patterns[a], dtype=np.float)
58              #states=np.array
                    ([-1,-1,-1,-1,1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,-1,-1,1.],
                      dtype=np.float)
59              origin=copy.deepcopy(states)
60              energy_list = [-0.5 * np.sum(weights.dot(states) * states)]
61              # recalling
62              for itr in range(10):
63                  for i in np.random.permutation(m):
64                      activations[i] = np.dot(weights[i,:], states)
65                      states[i]=np.sign(activations[i])
66                      energy = -0.5 * np.sum(weights.dot(states) * states)
67                      energy_list.append(energy)
68
69              accuracy.append(sum(states==origin)/len(states))
70
71          mean_acc=0
72          mean_acc=np.mean(accuracy)
73
74          result[p-1][N-2]=mean_acc
75
76  print(result)
77
78  def display(X):
79      plt.imshow(X.reshape((5,5)))
80
```

```
81 weight=0
82
83 patterns=gen(4,25)
84 n = patterns.shape[0]
85 m = patterns.shape[1]
86 eta = 1./n
87
88       # training
89 weight= hebbian(m, patterns)
90 weights=np.linalg.pinv(weight)
91
92
93 a=np.random.choice(range(len(patterns)))
94 print('a____',a)
95
96 states = np.array(patterns[a], dtype=np.float)
97           #states=np.array
                ([-1,-1,-1,-1,1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,1,1,-1,1,-1,-1,-1,1.],
                  dtype=np.float)
98 origin=copy.deepcopy(states)
99 print(origin)
100          # recalling
101 for itr in range(10):
102               for i in np.random.permutation(m):
103                   activations[i] = np.dot(weights[i,:], states)
104                   states[i]=np.sign(activations[i])
105
106 print(states)
107 print(sum(states==origin)/len(states))
108
109 fig = plt.figure(figsize=(12,4))
110 ax1=fig.add_subplot(1, 3, 1)
111 display(origin)
112 ax1.set_title('Origin')
113
114
115 ax2=fig.add_subplot(1, 3, 2)
116 test=copy.deepcopy(origin)
117
118 list=range(24)
119 sample=np.random.choice(list,3,replace=False)
120 for j in sample:
121     test[j]=-test[j]
122 print(sample)
123
124 display(test)
125 ax2.set_title('Contaminated')
126
127 print(origin)
128 print(test)
129
130
131
```

```
for itr in range(10):
    for i in np.random.permutation(m): # asynchronous activation
        activations[i] = np.dot(weights[i,:], test)
        test[i]=np.sign(activations[i])
print(test)


ax3=fig.add_subplot(1, 3, 3)
display(test)
ax3.set_title('Recovery')

import numpy as np
import seaborn as sns
import matplotlib.pylab as plt
plt.style.use("seaborn")



# 3. Plot the heatmap
plt.figure(figsize=(9,9))
x_axis_labels = [2,3,4,5,6,7,8,9,10] # labels for x-axis
y_axis_labels = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
s=heat_map = sns.heatmap( result, linewidth = 1 , annot = True,
    xticklabels=x_axis_labels, yticklabels=y_axis_labels)
plt.title( "Heatmap of overall memory accuracy with different training
    sizes and sparseness" )
s.set(xlabel='N:number of patterns', ylabel='p: proportion of +1 in the
    bipolor input vectors')

plt.show()
```