CSC311, Summer 2023, Final Project Report

Canyang Wang, Xiangyu Tu

1 Part A

1. Question 1

   (a) Here is our plot for the accuracy on the validation data as a function of k.

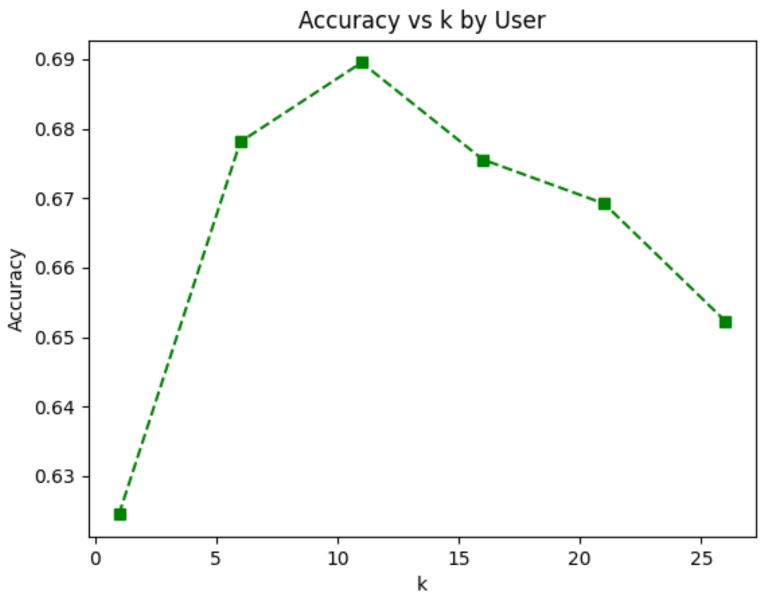


Figure 1: Accuracy plot by user on the validation data

   (b) Our choice for k* is 11 , and the final test accuracy as 0.6841659610499576 and validation accuracy as 0.6895286480383855

(c) For the item-based collaborative filtering, Our choice for k* is 21 , and the final test accuracy as 0.6816257408975445 and validation accuracy as 0.6922099915325995

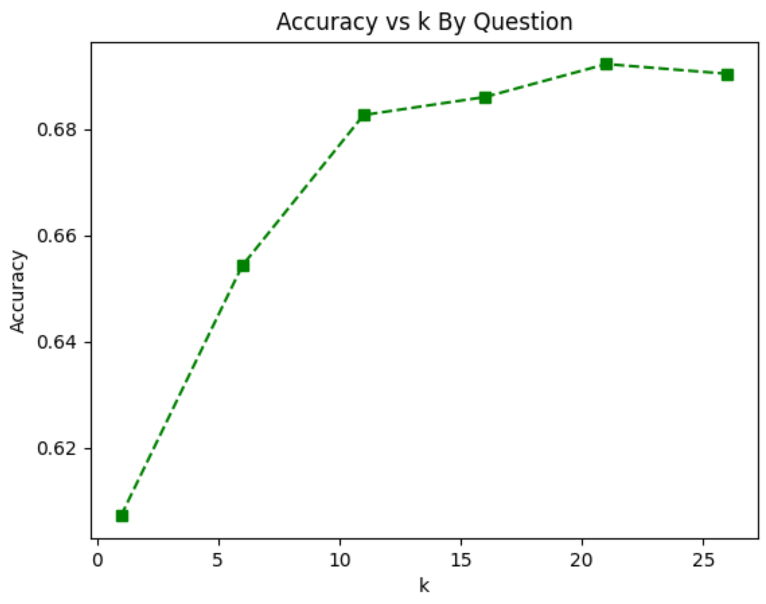


Figure 2: Accuracy plot by question on the validation data

(d) We think between the test performance between user- and item- based collaborative filtering, user based collaborative filtering is better since it has much better test accuracy at 0.6841659610499576 which is about 0.0025 more accurate than that of item-based collaborative filtering. Also, user-based collaborative filtering reaches the highest accuracy much faster than item-based collaborative filtering which is 11 and 21 separately.

(e) Here are the limitations for kNN algorithm for this task:

1. Computationally expense:
kNN algorithm can be very computationally expensive, especially when dealing with large datasets and large k values. This is because the distance from a test point to every other point in the dataset must be calculated.This problem gets worse in high-dimensional spaces. Moreover, we have to re-run the algorithm on the whole dataset after every changes which may cause more time compared with other methods in this task.

2. Dimensionality Issue (Curse of Dimensionality):
In high-dimensional spaces, the concept of ”nearest” may not hold the same significance as in lower dimensions. Thus, most points are nearly the same distance to be more specific: The points in high dimensions tend to be almost equidistant to the query point, which weakens the power of kNN.

3. Choose a k value proper or not:
A smaller 'k' value can capture noise and lead to overfitting, while a larger 'k' value can lead to underfitting. However, it is hard to choose a great k value without using techniques, thus it is critical to choose a right value of 'k'

2. Question 2

(a) Here is our derivation for the log-likelihood $log(p(C|\theta,\beta))$

Part A

2. (a) First, I assume $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$ represents all students' ability where $\theta_i \in \theta$, also assume $\beta = (\beta_1, \beta_2, \ldots, \beta_M)$ represents all probabilities that all questions are correctly answered by students.

Thus, I can get $L(C|\theta,\beta) = \prod_i^N \prod_j^M P(C_{ij}|\theta_i,\beta_j)$

Since $P(C_{ij}=1|\theta_i,\beta_j) = \frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}$

Thus,

$$L(C|\theta,\beta) = \prod_i^N \prod_j^M \left(\frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}\right)^{C_{ij}} \left(1-\frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}\right)^{(1-C_{ij})}$$

$$= \prod_i^N \prod_j^M \left(\frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}\right)^{C_{ij}} \left(\frac{1}{1+\exp(\theta_i-\beta_j)}\right)^{(1-C_{ij})}$$

Then, take the log of $L(C|\theta,\beta)$ to get

$$l(C|\theta,\beta) = \log\left(\prod_{i=1}^N \prod_{j=1}^M \left(\frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}\right)^{C_{ij}} \left(\frac{1}{1+\exp(\theta_i-\beta_j)}\right)^{(1-C_{ij})}\right)$$

$$= \sum_{i=1}^N \sum_{j=1}^M C_{ij} \log\left(\frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}\right) + (1-C_{ij}) \log\left(\frac{1}{1+\exp(\theta_i-\beta_j)}\right)$$

$$= \sum_{i=1}^N \sum_{j=1}^M C_{ij} \log\left(\frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}\right) + \log\left(\frac{1}{1+\exp(\theta_i-\beta_j)}\right) - C_{ij} \log\left(\frac{1}{1+\exp(\theta_i-\beta_j)}\right)$$

$$= \sum_{i=1}^N \sum_{j=1}^M C_{ij} \left(\log(\exp(\theta_i-\beta_j))\right) + \log\left(\frac{1}{1+\exp(\theta_i-\beta_j)}\right)$$

$$= \sum_{i=1}^N \sum_{j=1}^M C_{ij} (\theta_i-\beta_j) - \log(1+\exp(\theta_i-\beta_j))$$

Now, take the derivative of $l(C|\theta,\beta)$ of θ_i to get the log-likelihood

$$\frac{\partial l}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \sum_{i=1}^N \sum_{j=1}^M C_{ij} (\theta_i-\beta_j) - \log(1+\exp(\theta_i-\beta_j))$$

$$= \sum_{j=1}^M \frac{\partial}{\partial \theta_i} \sum_{i=1}^N C_{ij}(\theta_i-\beta_j) - \log(1+\exp(\theta_i-\beta_j))$$

$$= \sum_{j=1}^M C_{ij} - \frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}$$

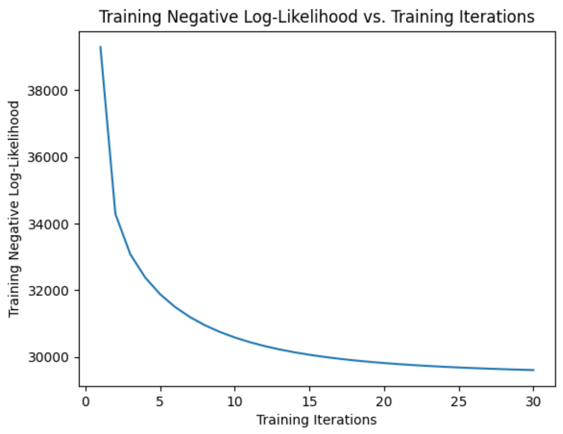
Take the derivative of $l(C|\theta,\beta)$ by β_j to get the log-likelihood:

$$\frac{\partial l}{\partial \beta_j} = \frac{\partial}{\partial \beta_j} \sum_{i=1}^N \sum_{j=1}^M C_{ij} (\theta_i-\beta_j) - \log(1+\exp(\theta_i-\beta_j))$$

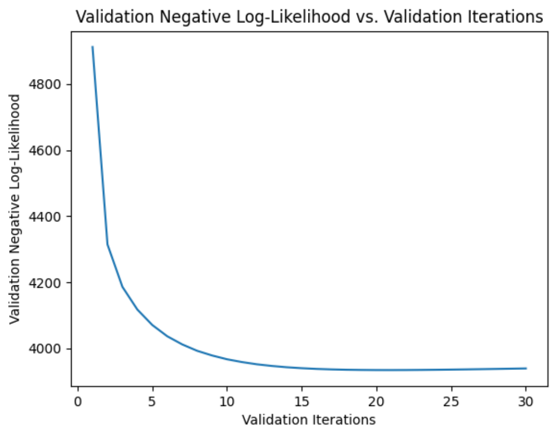
$$= \sum_{i=1}^N \frac{\partial}{\partial \beta_j} \sum_{j=1}^M C_{ij}(\theta_i-\beta_j) - \log(1+\exp(\theta_i-\beta_j))$$

$$= \sum_{i=1}^N -C_{ij} + \frac{\exp(\theta_i-\beta_j)}{1+\exp(\theta_i-\beta_j)}$$

(b) For this question our hyper-parameters are: learning rate is 0.015 and iteration is 30. Here are the training curve and validation curve:



(a) Training



(b) Validation

Figure 3: Training and validation figures.

(c) Our final validation accuracy is 0.70575783234547 , and test accuracy is: 0.7053344623200677 .

(d) Here are the curves generated by the plot that represent the probability of a student with a given ability level (theta) answering a particular question (j1, j2, or j3) correctly.

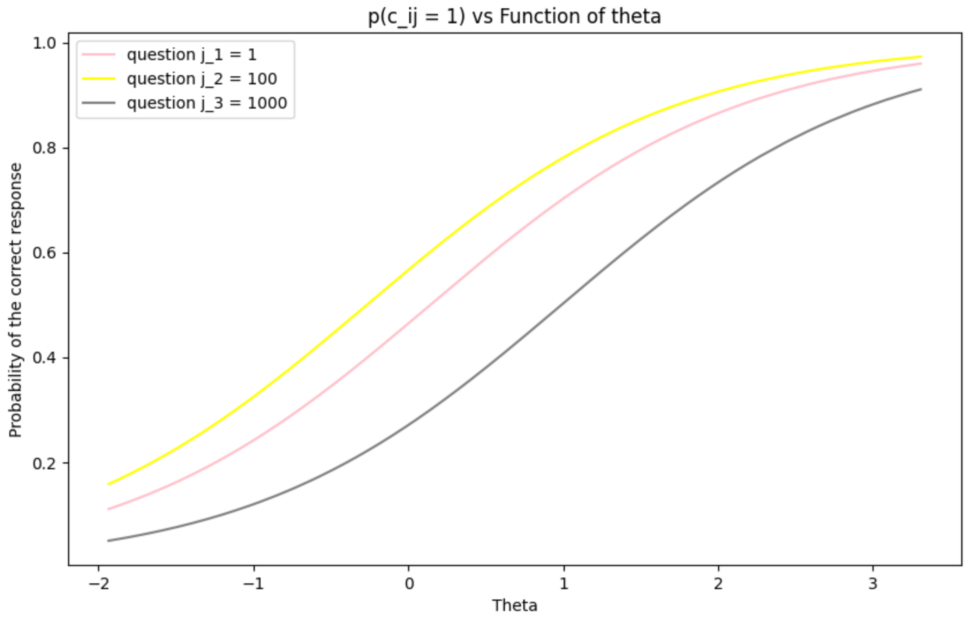


Figure 4: Probability of a student with a given ability level answering a particular question correctly

Each curve corresponds to a different question, with the difficulty level of the question represented by beta.
The shape of these curves follows the sigmoid function. This means that as the ability level of a student (theta) increases, the probability of the student answering the question correctly also increases. The curves also indicate they all form positive correlations.
For each curve:
A curve closer to the x-axis corresponds to a more difficult question.
A curve closer to the y-axis corresponds to an easier question.

3. Question 3 (Option 2: Neural Networks)

   (a) For the differences between ALS and neural networks:
   1. They are different in algorithm and model structure.
   Alternating Least Squares is an iterative optimization algorithm that is commonly used to solve matrix factorization problems. It works by fixing one factor and solving for the other in a least squares sense. This process alternates between the factors until convergence.
   On the opposite, for neural networks, it is a class of models that consists of interconnected layers of nodes with varying activation functions. Neural networks can be used for a wide variety of tasks, including classification, regression, and so on.
   2. They are different in complexity and flexibility.
   ALS's complexity is relatively low, and it generally has fewer hyperparameters to tune.
   Neural Networks are highly flexible and can be applied to a broad range of problems. They have a highly adaptable structure and can include many layers and thousands of connections between nodes. This makes the method extremely powerful but also more complex, it needs careful tuning of hyperparameters.

3. They are different in interpretability and transparency.
ALS is more interpretable since it always has a clear and tangible meaning. The process of ALS is also relatively transparent and can be understood as solving a series of linear equations.
However for neural networks, it seems like a black box which its predictions may be accurate, but understanding how they reach a particular conclusion can be difficult due to the complex interactions between many neurons across layers. It is lack of transparency and it can be harder to trust and understand.
4.They are different in optimization.
ALS uses linear equations to minimize where at each iteration solves the optimal solution efficiently and directly instead of by using gradient descent to find the optimal set of weights as what neural networks does.
5. They run for different models.
As for ALS, it always runs for linear model, using two matrix multiplication to predict the missing values, but for neural networks, according to its activation function, it can run for both linear or non-linear models.
6. Different number of parameters to minimize:
For ALS, it needs two parameters to optimize which are usually referred to as user and item matrices, but neural networks typically have many parameters that need to be optimized. These parameters include the weights and biases across all layers in the network.

(b) See neural_network.py

(c) We choose the number of epoch as 20, k as 200 to get the best Validation accuracy is 0.6788032740615297

(d) Here is the final test accuracy: The test accuracy under the k value 200 is 0.6694891335026814

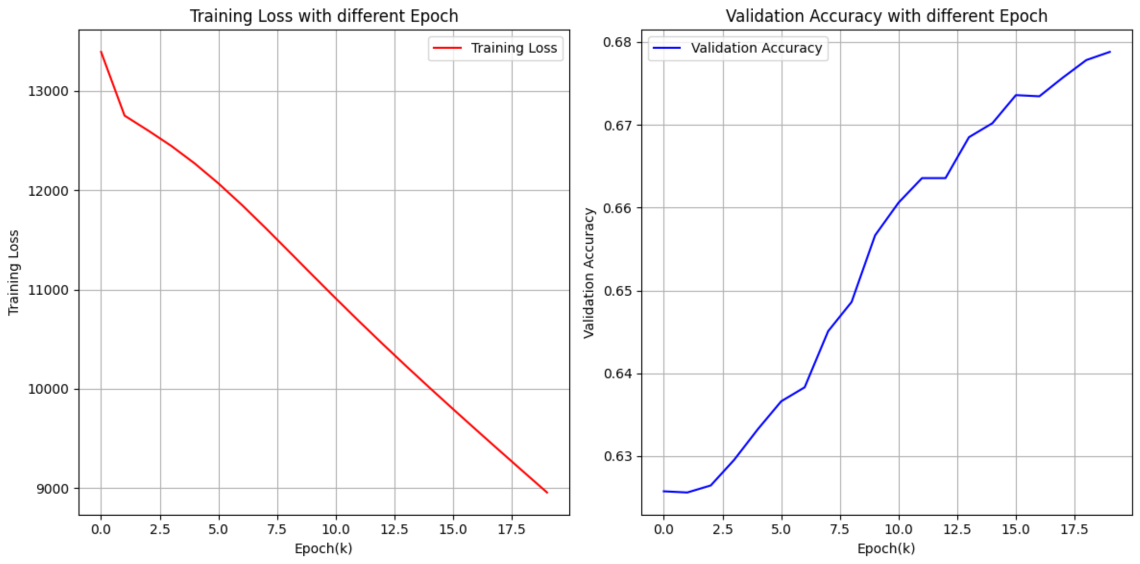


Figure 5: Training and validation curves

(e) Here is the graph for final validation and test accuracies:

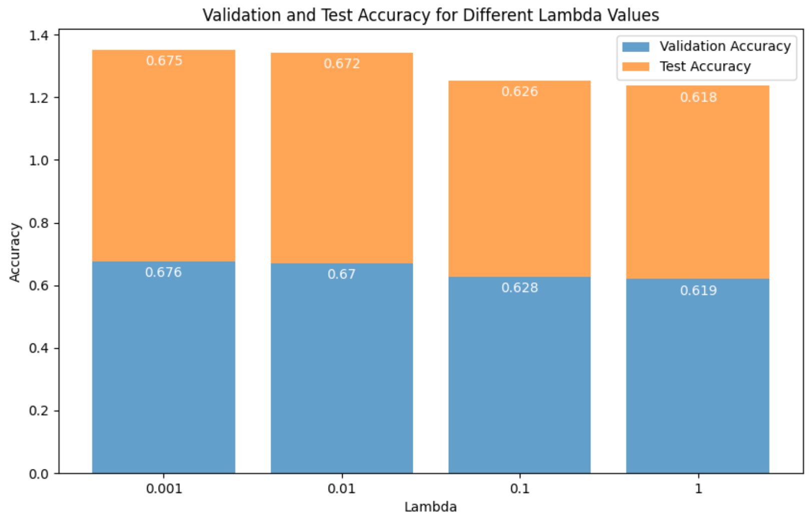


Figure 6: Training and validation curves

Under lambda = 0.001, it reaches the best validation and test accuracy, but for the validation accuracy under regularization, it performs a little bit worse which has about 0.002 lower but for the test accuracy under regularization it performs much better which has about 0.006 greater than the one without regularization.

4. Question 4

   (a) Our selected hyperparameters for knn are:
   k = 11
   Our selected hyperparameters for irt are:
   learning rate = 0.01
   iterations = 30
   Our selected hyperparameters for nn are:
   k = 200
   learning rate = 0.01
   lambda = 0
   epochs = 20

   (b) For our process of doing the ensembling, which are:
   Step 1: Resampling the Training Data (Bootstrap)
   Three base models are selected, and the training data is resampled for each model using the resample_data function. Bootstrapping, or resampling with replacement, is used to create unique datasets for training each base model.

Step 2: Base Model Training
Three base models are chosen: k-Nearest Neighbors (KNN), Item Response Theory (IRT), and a Neural Network (AutoEncoder).
KNN Model:

A user-question matrix is created from the resampled training data, and missing values are filled using KNN imputation.
IRT Model:

Item Response Theory is trained using the resampled training data to estimate user ability (theta) and question difficulty (beta).
Neural Network Model:

An AutoEncoder neural network is trained using the resampled training data. Missing values are handled by converting them to zeros.

Step 3: Ensemble Prediction

The ensemble_predict function is used to generate predictions from each of the base models for the given test data.

KNN Predictions: Extracted directly from the trained KNN matrix.

IRT Predictions: Calculated using the sigmoid function and the trained theta and beta values.

NN Predictions: Generated by passing the user-question matrix through the trained neural network.

The final ensemble prediction is computed by averaging the predictions from the three base models and then applying a threshold of 0.5.

Step 4: Evaluate the Ensemble
The ensemble's predictions are compared to the true labels to calculate the validation and test accuracies. Results are printed to the console.

(c) As for our conclusion of ensembling:
After running our ensemble.py, we get the validation accuracy as 0.6821902342647473 and test accuracy as 0.6906576347727914. This is much better than our results from the neural network, which the validation accuracy is 0.6788032740615297 and the test accuracy is 0.6694891335026814. Also for knn algorithms, when $k = 11$, we got test accuracy as 0.6841659610499576 which is lower than our ensembling test accuracy and as for the validation accuracy, the knn algorithm got as 0.6895286480383855 which is lower than that of ensembling validation accuracy. Only for using the irt algorithm, we got validation accuracy as 0.70575783234547 and test accuracy as 0.7053344623200677 which both are higher than the result of ensemble.
Thus, we can conclude that although by averaging the predictions, the ensemble can improve stability and potentially increase accuracy compared to several algorithm, it still cannot be improved

a lot. Since the ensemble aggregates different types of models, it might capture more complex patterns and relationships in the data, possibly leading to improved performance over individual models. Also, It is likely to perform better if the base models have complementary strengths and weaknesses.

2 Part B

1. Formal description

   (a) **Introduction**: In this part, we will enhance the prediction accuracy and address the limitations of the neural network model from part A. We introduce modifications by (1) adding an additional hidden layer, and (2) using mini batch gradient descent.

   (b) **Model Architecture**: In our augmented neural network model, we introduced a new hidden layer in our auto-encoder function. We constructed the **AugmentedAutoEncoder** with three linear layers. The initial layer transforms the input, which has the dimensionality of num_question, into a k-dimensional space. This output is then further transformed into an l-dimensional space by the second layer, which is the **new hidden layer**. Finally, the third layer returns the output to the num_question dimensionality, providing the predicted answers.

   (c) **Model Components**:
   Input Layer: num_question dimensions
   First Hidden Layer (g): num_question $\rightarrow$ k
   Second Hidden Layer (h): k $\rightarrow$ l
   Output Layer (i): l $\rightarrow$ num_question
   Here is the simplified diagram of the augmented neural network model:

   (d) **Model Equations**:
   Given a user $v \in \mathbb{R}^{N_{equations}}$ from a set of users S.

$$g(v) = f_1(W^{(1)} * v + b^{(1)})$$
$$h(g) = f_2(W^{(2)} * g(v) + b^{(2)})$$
$$i(h) = f_3(W^{(3)} * h(g) + b^{(3)})$$

   Where f_1, f_2, f_3 are activation functions, in this model we will try {sigmoid, ReLU, and Tanh} to find which one provide the best validation and test accuracies.
   W and b are the weight and bias for the respective layers.

   Regularization: Weight Norm $= ||W^{(1)}||_2^2 + ||W^{(2)}||_2^2 + ||W^{(3)}||_2^2$
   This helps prevent overfitting and ensures that the model weights don't grow unbounded.

   (e) **Optimization with Batch Gradient Descent**:
   To further improve the training process and convergence rate, our model employs the technique of batch gradient descent. This technique can be computationally expensive, but it offers the advantage of stable and less noisy weight updates. In our project, we will try batch sizes of {1, 50, 100, 200, 500, 1000}.
   The update equation for batch gradient descent is given by:

$$\theta = \theta - \alpha \cdot \nabla J(\theta)$$

   where θ represents the parameters (or weights) of the model, α is the learning rate, and $\nabla J(\theta)$ is the gradient of the loss function with respect to the parameters.
   Employing batch gradient descent ensures consistent and steady convergence towards the global minimum of the loss function, especially for convex loss surfaces. It eliminates the oscillations and noise commonly found in stochastic updates, making it a suitable choice for our augmented neural network.

(f) **Rationale for Model Enhancements**:

i. **Introduction of an Additional Hidden Layer**:

- *Increased Model Capacity*: A single-layer neural network has its limitations in terms of the complexities it can capture. By introducing another hidden layer, our model has a richer set of parameters and can learn more intricate patterns from the data. This can be particularly useful if the underlying relationship between the input and output is nonlinear.
- *Reduced Underfitting*: Underfitting occurs when the model is too simplistic to capture the underlying patterns of the dataset. By adding an additional hidden layer, we're giving our model the opportunity to learn deeper and potentially more accurate representations of the input data.

ii. **Utilization of Mini Batch Gradient Descent**:

- *Computational Efficiency*: Gradient descent using the entire dataset can be computationally intensive, especially when dealing with large datasets (more than 50,000 rows). Batch gradient descent offers a compromise by computing the gradient on a subset of the data, thereby speeding up the training process while still benefiting from the accuracy of using more than one data point.
- *Noise can Aid in Avoiding Local Minima*: While the gradients calculated in batch gradient descent are approximations of the true gradient (as calculated with the entire dataset), this noise can sometimes help the algorithm jump out of local minima.

In conclusion, our enhancements aim to address potential underfitting by increasing the model's capacity and ensuring stable and efficient optimization through batch gradient descent. Together, these modifications should help improve the model's overall predictive accuracy and robustness.

2. Relevant Figure and Diagram

(a) **Model diagram**: Here is the diagram of the new augmented neural network:

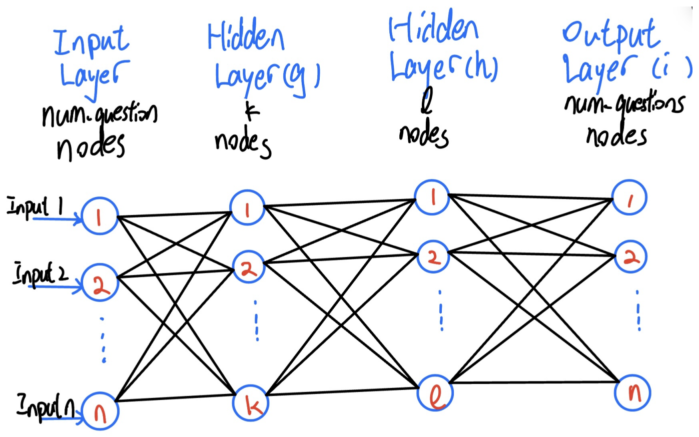


Figure 7: 'h' is the new hidden layer we added in our model with 'l' nodes.

(b) **Activation functions**: Here are the graph of the activation functions we are using for our augmented neural network {sigmoid, Tanh, and ReLU} for f_1, f_2, f_3.

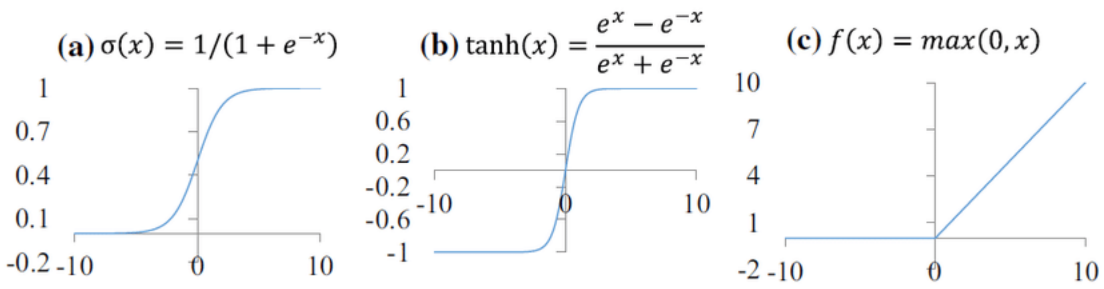


Figure 8: Left plot: sigmoid, Middle plot: Tanh, Right plot: ReLU[2].

(c) Difference between **batch gradient descent**, **mini-batch gradient descent**, and **stochastic gradient descent**:

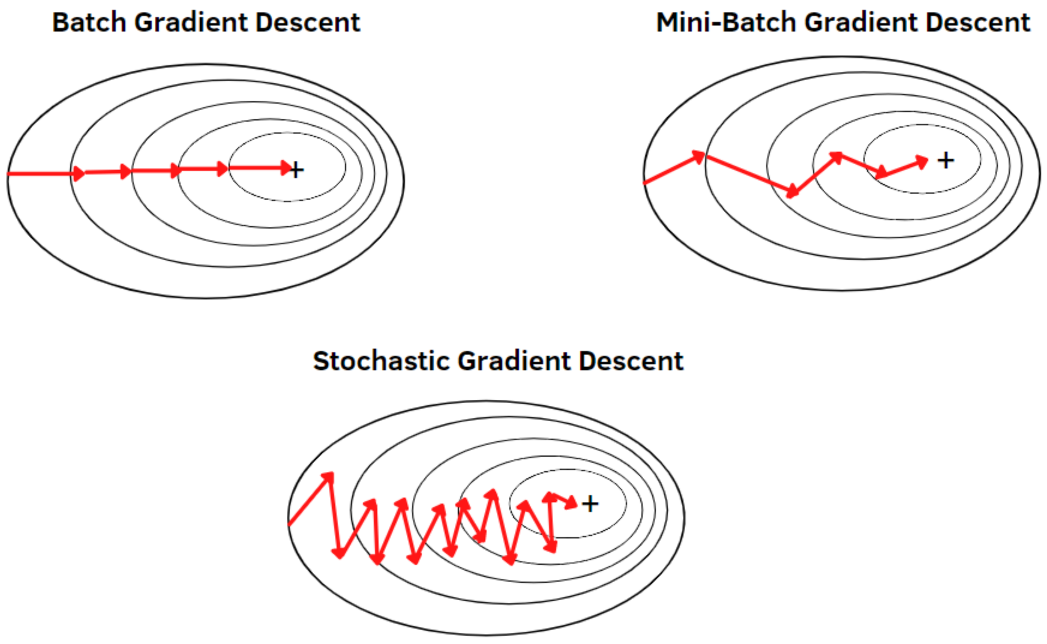


Figure 9: Effect on training curve of different types of gradient descent.

Batch Gradient Descent updates weights using the whole dataset, providing a smooth training curve but may be slow for large datasets. Mini-Batch Gradient Descent uses a subset of the data, balancing speed and stability, and is often preferred due to its efficiency and hardware optimization benefits. Stochastic Gradient Descent (SGD), with a batch size of 1, offers faster updates but produces a noisier curve and may oscillate near the optimal point. The chosen method can significantly shape the training curve and overall model performance.[2]

3. Demonstration and Comparison:

   (a) **Training loss with different batch sizes**:
   Hyper parameter setting: learning_rate=0.1, num_epoch = 20, k=200, l=50, activation_function = sigmoid

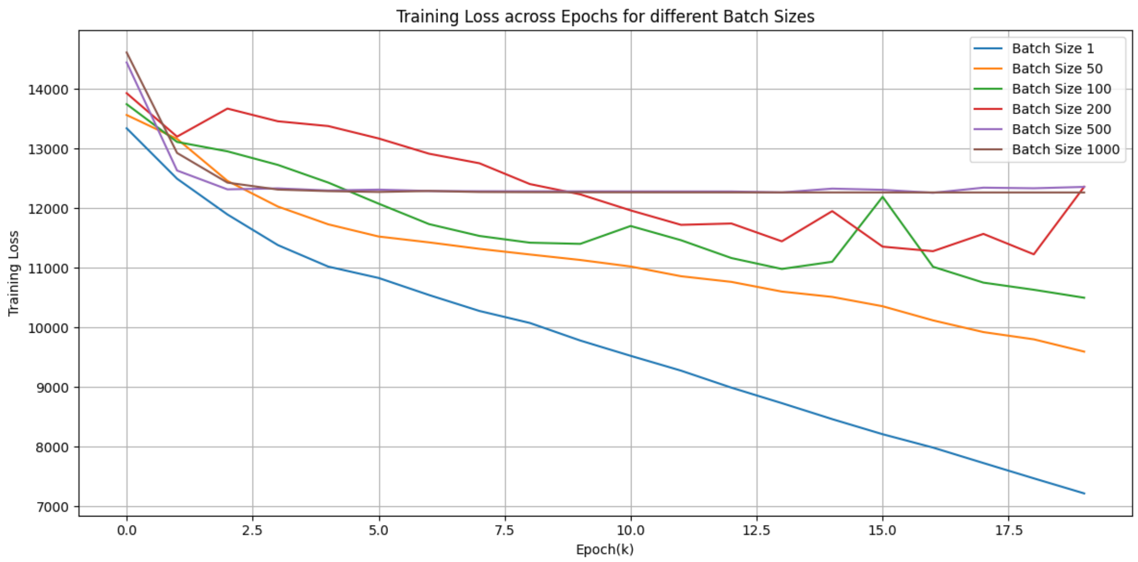
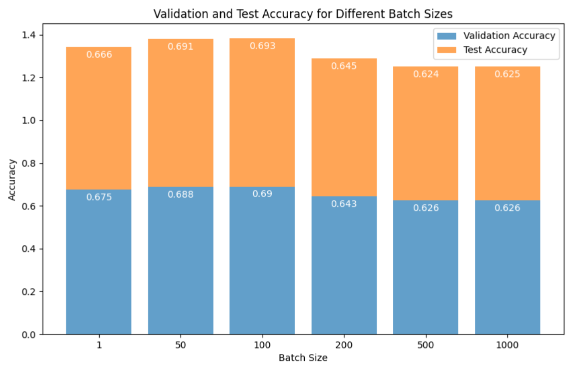


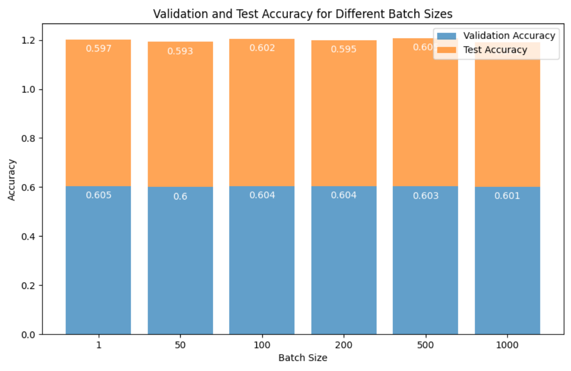
Figure 10: Plot for training loss curve with different batch sizes

From the above plot, we can see that when the batch size is 1, the model updates its weights after every single training example. This frequent updating can often allow the model to converge to a lower training loss. On the other hand, using a larger batch size, such as 500 or 1000, makes the model update its weights based on the average gradient derived from a greater number of samples. This might prevent the model from exploring potential beneficial paths in the loss landscape that smaller batches or individual samples might uncover. This behaves in the line for 500 and 1000 converged much early and at a much higher training loss compared with other batch sizes. Intermediate batch sizes like 50, 100, and 200 find their performance nestled between the extremes of 1 and 1000. This is evidence that my interpretation of the batch size and its effect on training loss is correct.

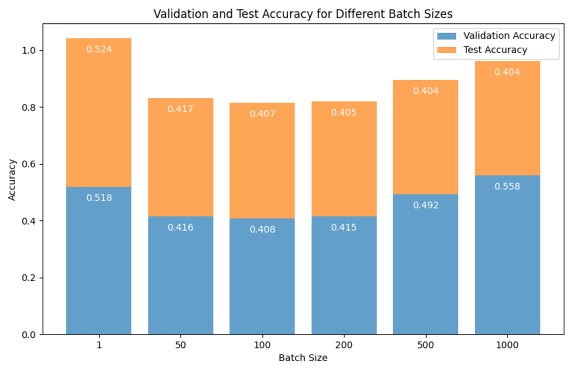
(b) **Test and validation accuracy for different activation functions**:
Hyperparameter setting: learning_rate=0.1, num_epoch = 20, k=200, l=50



(a) Sigmoid Activation



(b) Tanh Activation



(c) ReLU Activation

Figure 11: Comparison of different activation functions with varying batch sizes

After repeating the test several times, we found that the sigmoid activation function consistently produces the best validation and test accuracy after 20 epochs. Under this setting, the model will the highest test and validation accuracy when the batch size is 100. Where ReLU activation function will always produce the lowest validation and test accuracy.

(c) **Effect of learning rate**:

Hyperparameter setting: num_epoch = 20, k=200, l=50, activation_function = sigmoid

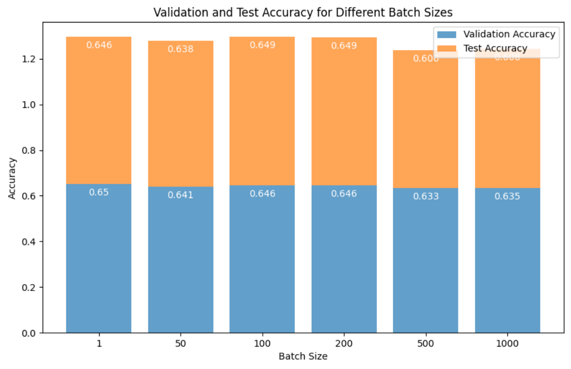


Figure 12: Learning rate = 0.01

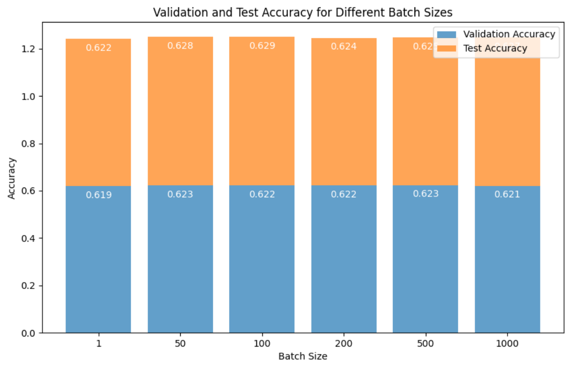


Figure 13: Learning rate = 0.001

During the course of this project, we evaluated the influence of different learning rates on batch sizes. In this experiment, we set all hyperparameters the same, with the exception of the learning rate. The plots indicate that when the learning rate is set very low, the effectiveness of batch size becomes less significant in comparison to models with higher learning rates. This behaviour is due to a smaller learning rate results in a smoother training curve, making it less susceptible to the noise introduced by varying batch sizes.

(d) **Result of best validation accuracy**: In order to achieve the best hyperparameter combination, we created a file called augmented_neural_network_parameter.py to test out different combinations. The best test Accuracy is 0.693 or **(69.3%)** with hyperparameters k=200, l=50, learning rate=0.1, B=100, and activation function=sigmoid. Compared with the best test accuracy of the base neural network model which is 0.669 or **(66.9%)**.

4. Limitation for this approach:

(a) **Choosing the best hyper-parameter:** In `augmented_neural_network_parameter.py`, We only tested a small set of combinations for the hyper-parameters. It takes a significant amount of time to train the models (around 30 minutes for a turn). It is very computational expansive to include more tests. The best parameter combination might not been included during our test.

(b) **Susceptibility to Overfitting:** With the added complexity due to augmentation and the introduction of more parameters, the neural network can easily overfit to the training data, especially if the training dataset isn't large enough or regularization isn't properly applied.

(c) **Potential for Underfitting:** Despite the augmentations, there's a chance the model might underfit the data. This suggests that the architecture may need further enhancement, such as adding more layers or even changing the type of layers, to capture the underlying patterns effectively.

(d) **Dependence on Activation Functions:** The performance of the augmented neural network can vary significantly based on the choice of activation functions. Some activation functions, like ReLU, can cause dead neurons during training, while others might not introduce enough non-linearity.

3 Contribution

1. Canyang Wang: Finish part A and proof read for part B, project structure set up

2. Xiangyu Tu: Finish part B and proof read for part A, finnished install.sh script for environment set up

4 Reference

1 A. Wagh, “Gradient descent and its types,” Analytics Vidhya, https://www.analyticsvidhya.com/blog/2022/07/gradient descent-and-its-types/ (accessed Aug. 16, 2023).

2 Figure 8. activation function. (a) sigmoid, (b) tanh, (c) relu., https://www.researchgate.net/figure/Activation-function-a-Sigmoid-b-tanh-c-ReLU_fig6_342831065 (accessed Aug. 16, 2023).