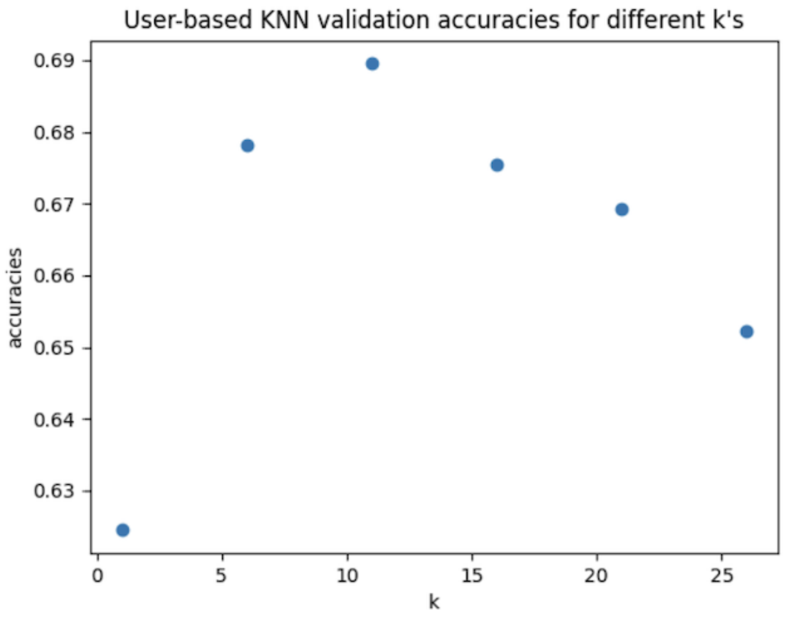
Correctness in Diagnostic Questions Matrix Completion in Personalized Education Platforms

Mason Hu | 1007880161 | mason.hu@mail.utoronto.ca
Yufei Liu | 1007746003 | yuf.liu@mail.utoronto.ca
Fucheng Zhuang | 1006320410 | fucheng.zhuang@mail.utoronto.ca

Mar 28th 2023

1 k-Nearest Neighbor

1.1 Tuning hyperparameters K for user based



User-based Validation Accuracy (approximately):

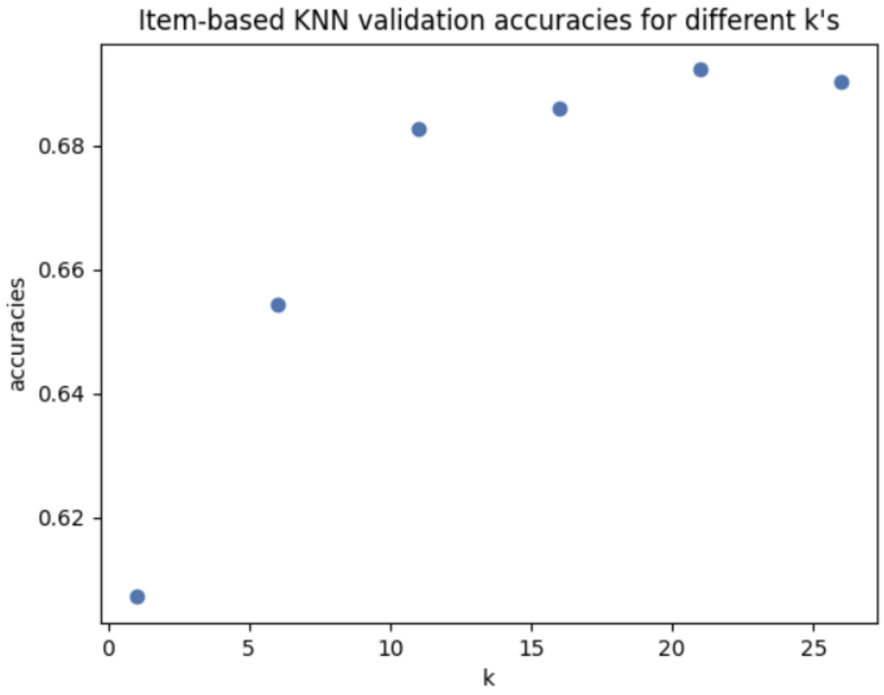
$$k = 1 : 0.6244; k = 6 : 0.678; k = 11 : 0.689;$$

$$k = 16 : 0.675; k = 21 : 0.669; k = 26 : 0.652.$$

1.2 Highest accuracy for user based

k^* has the highest validation accuracy when k = 11 and the final test accuracy is about 0.6841659610499576

1.3 Tuning hyperparameters K for item based



Item-based Validation Accuracy (approximately):

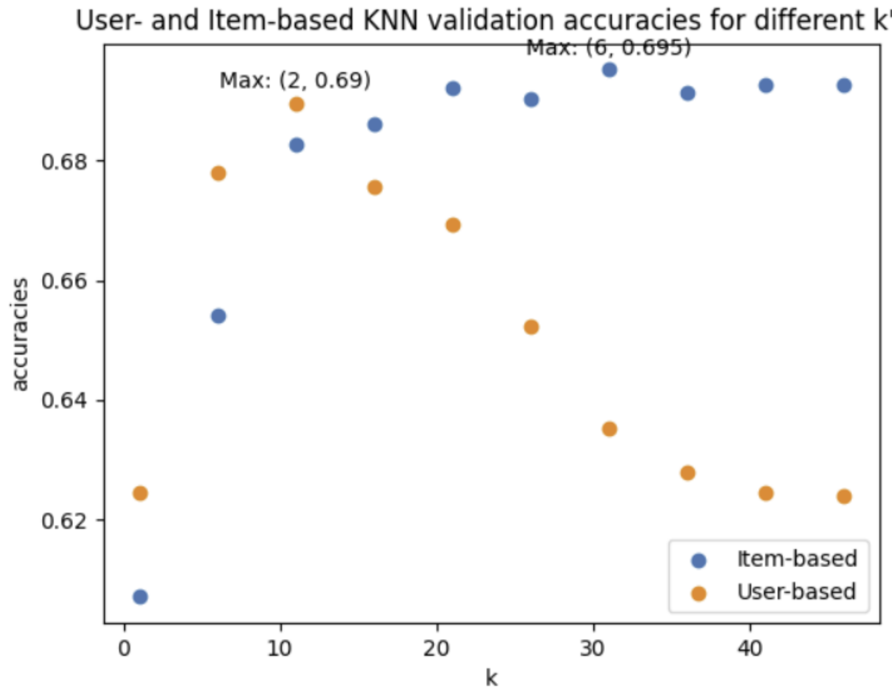
$$k = 1 : 0.607; k = 6 : 0.654; k = 11 : 0.682;$$

$$k = 16 : 0.686; k = 21 : 0.692; k = 26 : 0.690.$$

Highest accuracy for item based:
k^* has the highest validation accuracy when k = 21 and the final test accuracy is about 0.692

The underlying assumption on item-based collaborative filtering:
Users who have similar item preferences in the past will have similar item preferences in the future. That is to say, users who have answered the closest question in the past are likely to have the same answers for similar questions in the future. So, users tend to have consistent preferences over time, and their past answers are good indicators of their future answers.

1.4 Compare



From the graph, we can find out that item-based collaborative filtering performs better. The reason is that the validation accuracies from item-based collaborative filtering are slightly higher than that from user-based collaborative filtering for most values of k (especially for k ¿ 20). And user based filtering tends to overfit when k is greater than 11.

1.5 Potential limitations of kNN

1. There may not be a large amount of data available for training, especially if the question is very specific or niche. This can result in sparse data, where there are very few examples of similar questions and answers, which can make it difficult for the algorithm to accurately predict the answer.

2. The data have high dimensionality, which can make it difficult for the algorithm to identify relevant patterns and similarities between data points. In addition, high dimensionality can increase the computational complexity of the algorithm and make it more difficult to train and optimize. Also, many points in high dimensionality have similar distances even if they are spread out. So, the nearest distance may not be that helpful.

3. The test time for kNN is very low since it needs to compute the distance between the point and all other possible training data points. This makes the computation expensive and requires a longer time in the testing time.

2 Item Response Theory

2.1 Log-likelihood and derivative

The Log-likelihood of the IRT

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\beta}|c_{ij}) &= \log\left(\prod_{i=1}^{n}\prod_{j=1}^{k}\sigma(\theta_i-\beta_j)^{c_{ij}}(1-\sigma(\theta_i-\beta_j))^{c_{ij}}\right)\\ &= \sum_{i=1}^{n}\sum_{j=1}^{k} c_{ij}\log(\sigma(\theta_i-\beta_j)) + (1-c_{ij})\log(1-\sigma(\theta_i-\beta_j))\end{aligned}$$

where $\sigma(x) = \frac{1}{1+e^x}$ is the sigmoid/expit function.

The derivative, using the fact that $\sigma'(x) = \sigma(x)(1-\sigma(x))$, we get

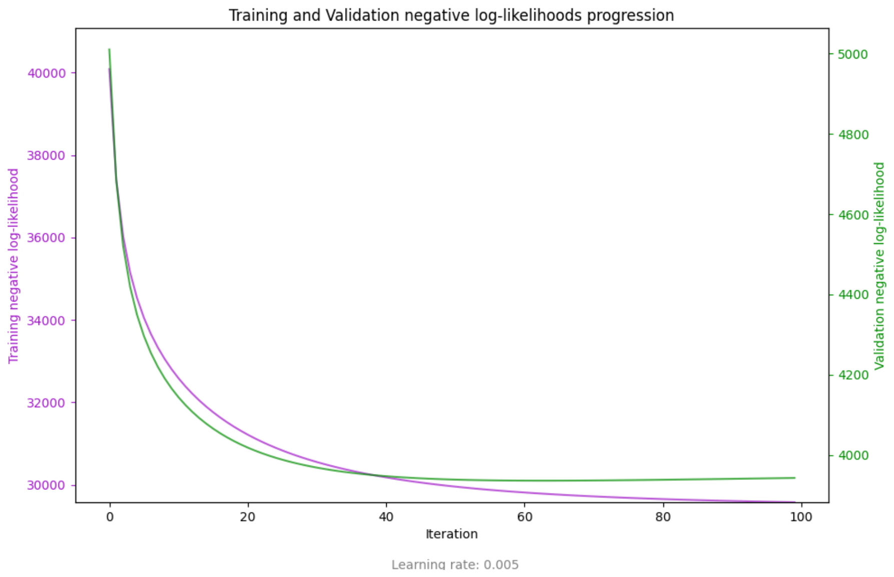
$$\begin{aligned}\frac{d}{d\theta_i}\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\beta}|c_{ij}) &= \frac{d}{d\theta_i}\sum_{j=1}^{k} c_{ij}\log(\sigma(\theta_i-\beta_j)) + (1-c_{ij})\log(1-\sigma(\theta_i-\beta_j))\\ &= \sum_{j=1}^{k} c_{ij}\frac{\sigma(\theta_i-\beta_j)(1-\sigma(\theta_i-\beta_j))}{\sigma(\theta_i-\beta_j)} + (1-c_{ij})\frac{-\sigma(\theta_i-\beta_j)(1-\sigma(\theta_i-\beta_j))}{1-\sigma(\theta_i-\beta_j)}\\ &= \sum_{j=1}^{k} c_{ij} - \sigma(\theta_i-\beta_j)\end{aligned}$$

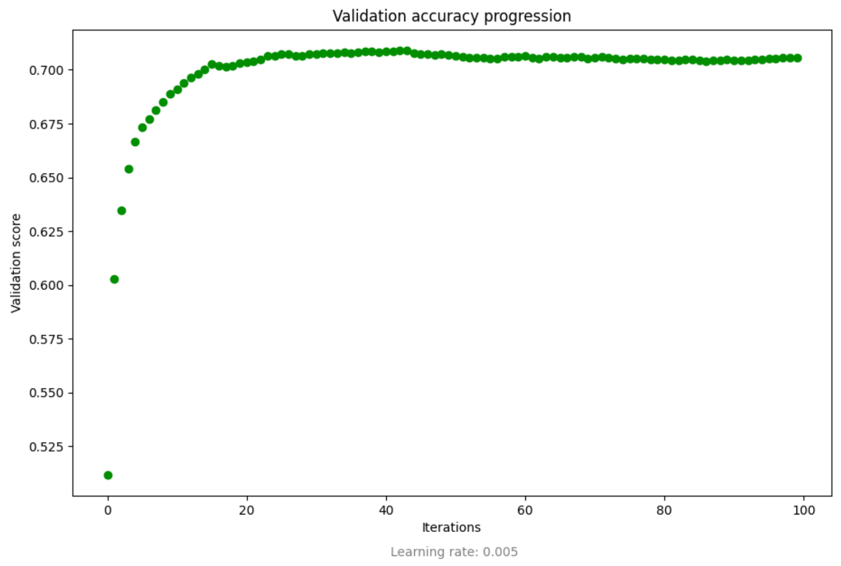
$$\begin{aligned}\frac{d}{d\beta_j}\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\beta}|c_{ij}) &= \frac{d}{d\beta_j}\sum_{i=1}^{n} c_{ij}\log(\sigma(\theta_i-\beta_j)) + (1-c_{ij})\log(1-\sigma(\theta_i-\beta_j))\\ &= \sum_{i=1}^{n} c_{ij}\frac{-\sigma(\theta_i-\beta_j)(1-\sigma(\theta_i-\beta_j))}{\sigma(\theta_i-\beta_j)} + (1-c_{ij})\frac{\sigma(\theta_i-\beta_j)(1-\sigma(\theta_i-\beta_j))}{1-\sigma(\theta_i-\beta_j)}\\ &= \sum_{i=1}^{n} \sigma(\theta_i-\beta_j) - c_{ij}\end{aligned}$$

Of course, in our context, we only partially have c_{ij} so we have to ignore the `NaN` values when we actually implement the gradient computing algorithm.

2.2 Train IRT

After running the model on different hyperparameters several times, I decided to use learning rate $lr = 0.005$ and iteration $ite = 100$ since these choices gives the most stable and fastest convergence.

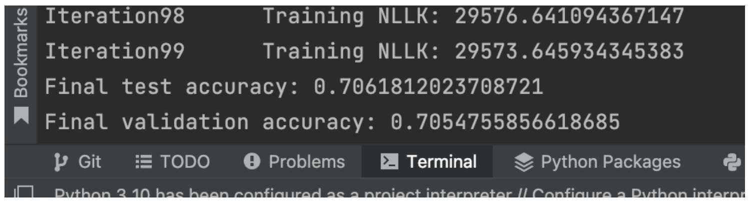




Above are the training and validation negative log-likelihoods curves as training progresses, and the validation accuracy curve as training progresses.

2.3 Final accuracy

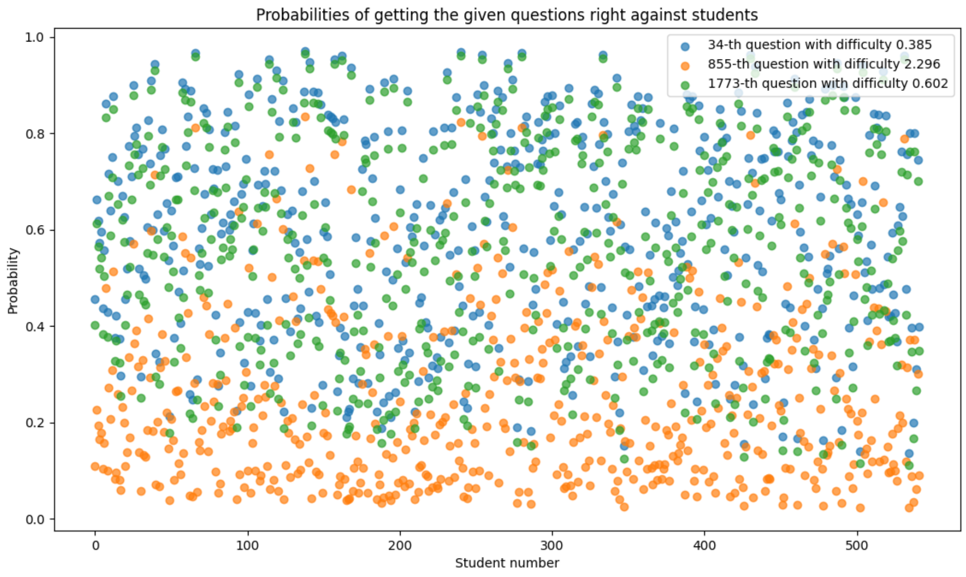
And the final test and validation accuracies are shown below.

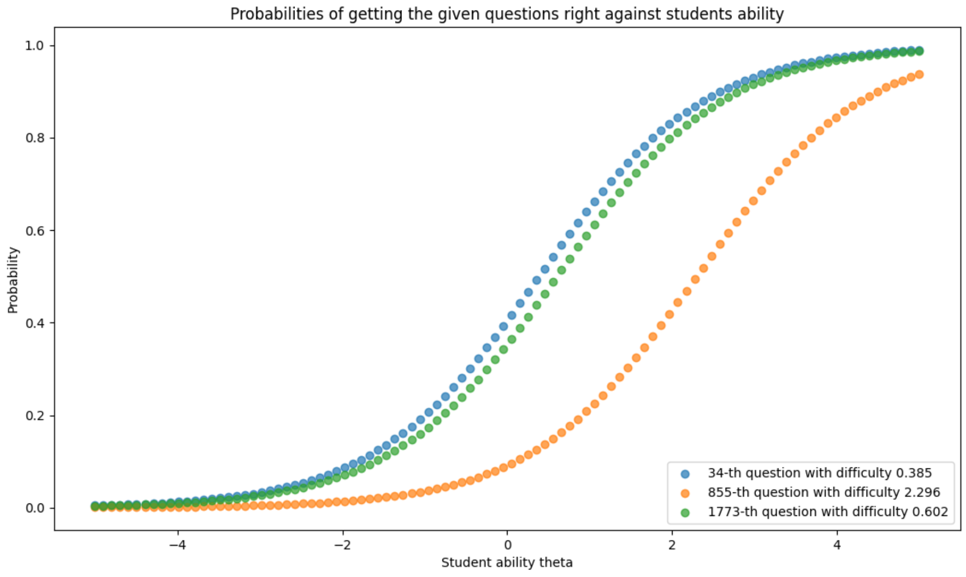


```
Iteration98      Training NLLK: 29576.641094367147
Iteration99      Training NLLK: 29573.645934345383
Final test accuracy: 0.7061812023708721
Final validation accuracy: 0.7054755856618685
```

Bookmarks
Git
TODO
Problems
Terminal
Python Packages
Python 3.10 has been configured as a project interpreter // Configure a Python interpr

2.4 Three curves





The above graphs are the probabilities of getting the given three questions right against students/students' abilities. In the first graph we can see that the more difficult (the higher the β) the question is, the lower probability students are going to them right.

In the second graph, the probabilities of getting the given questions right is positively related with the students ability to solve questions in general. In particular, this is an S-shaped expit curve.

This is in nature due to the logistic model conditioned on the βs we assumed for the Item Response Theory. The positional shift of these three curves represent the different difficulties of the questions: the more difficult the question is, the more the curve tends to the right.

3 Neural Networks

3.1 Describe differences

Difference 1 Alternating least squares and neural networks are both ways to solve complicated optimization problems. However, ALS assumes a linear relationship and factorizes a matrix into U, Z and minimized linear least squares. However, neural networks do not have strict assumptions, but instead learns the relationship of inputs and output by encoding the outputs into some form of cryptic knowledge and decoding them back into probabilities.

Difference 2 In ALS, we minimize the loss with respect to both Matrix U and Z. In particular, we minimize the objective

$$\min_{\mathbf{U},\mathbf{Z}} \frac{1}{2} \sum_{(i,j)\in O} (R_{ij} - \mathbf{u}_i^T \mathbf{z}_j)^2$$

where neural networks minimizes the squared error loss or cross entropy loss

$$\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}) = ||\mathbf{x} - \tilde{\mathbf{x}}||^2$$

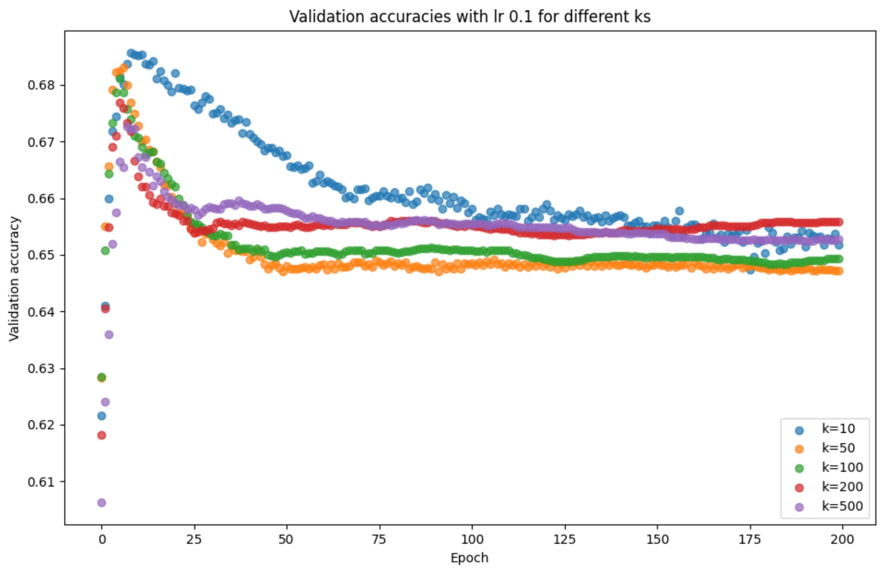
The first loss function is not jointly convex w.r.t matrices $\mathbf{U}, \mathbf{Z}$, which is why we have to fix one matrix and minimize the other, alternating this process among two matrices until convergence. However, in neural networks, we just have to use usual gradient descent to minimize the square error loss.

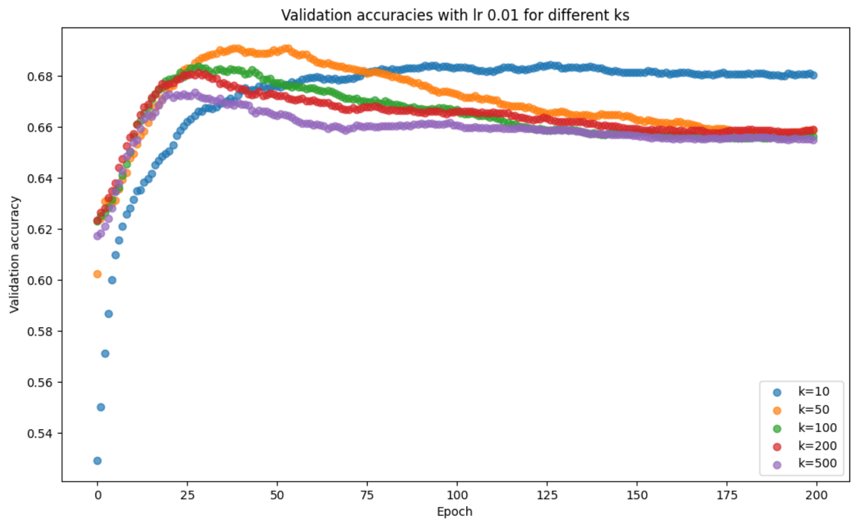
Difference 3 In general, ALS are designed to mainly solve problems sparse data problems in the context of matrix factorization. They are usually ideal for recommendation systems. But neural networks are more modern in a way that they can be used on sparse or non-sparse data and suit a wide range of complicated machine learning tasks, including natural language processing and image recognition. This also leads to their difference in computatability: ALS are usually more efficient in computation while neural networks requires more sophisticated optimization techniques.

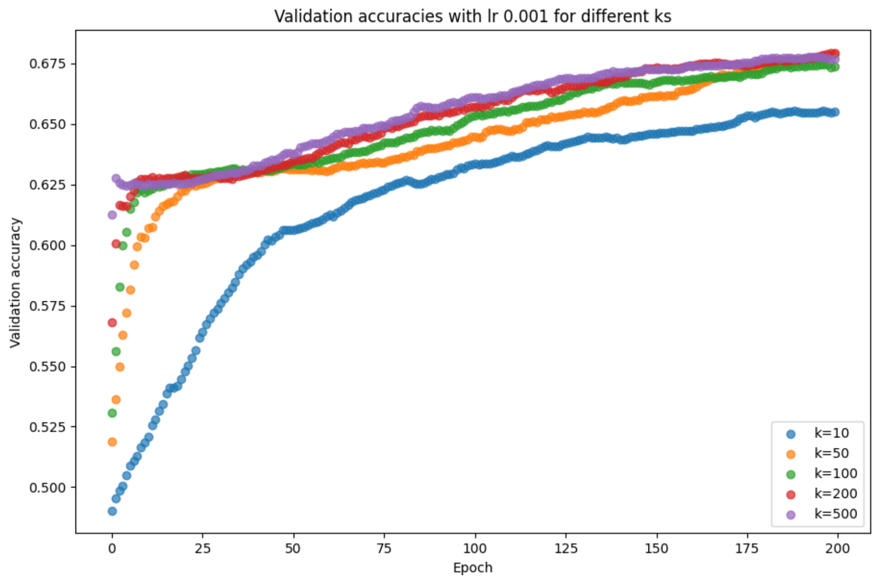
3.2 Implement `AutoEncoder()`

Completed in code

3.3 Train and Tune



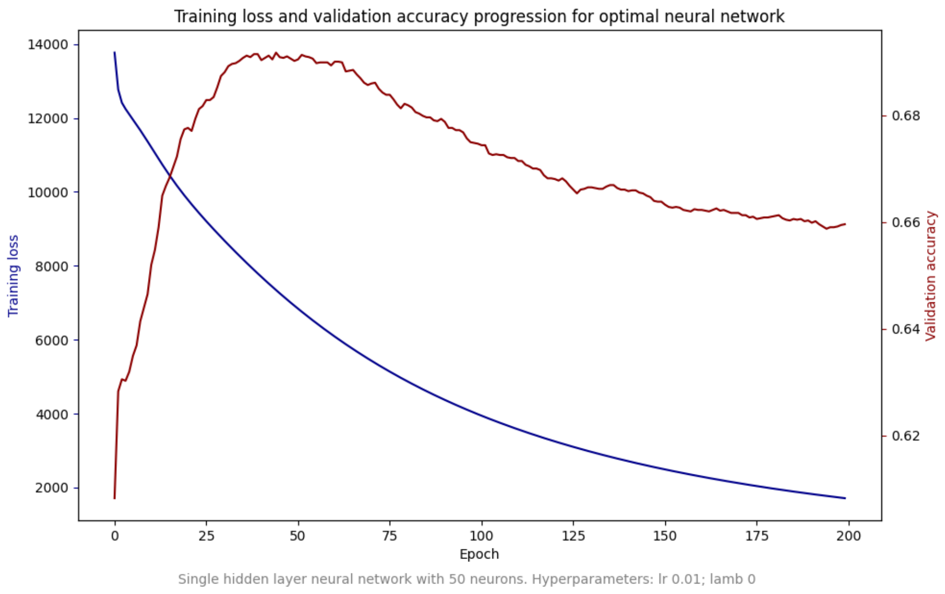




The above are the training performance of the models based on validation accuracy, of different number of neurons in the hidden layer and different learning rate, plotted against epoch.

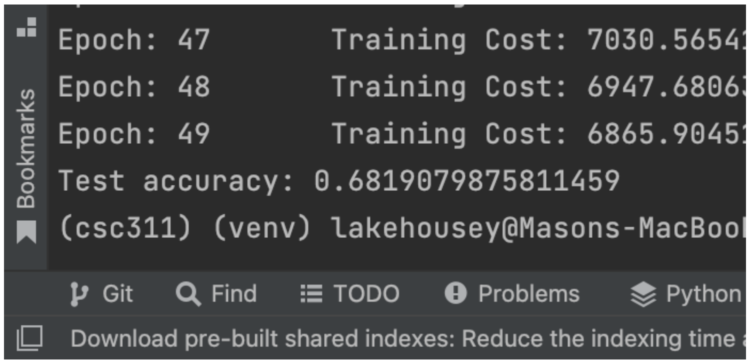
From the graphs we can see that the highest validation accuracy throughout all three graphs occur at approximately the 50th training epoch for $k^* = 50, lr = 0.01$. The validation accuracy there is approximately 0.69.

3.4 Plot and Report



As we can see from the plot, the training loss decreases consistently as training progresses, but the validation accuracy peaked at about 50 epochs and started to decrease afterwards.

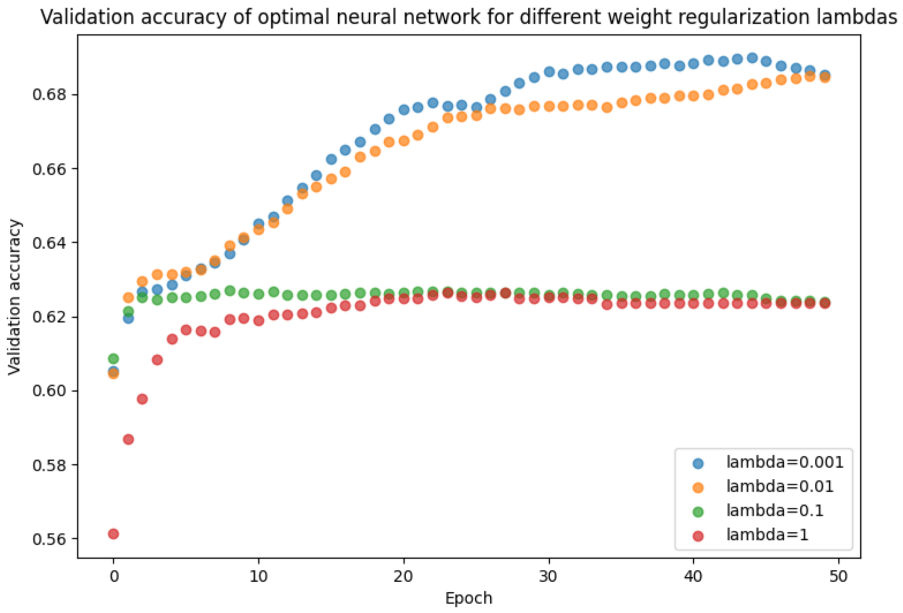
This is an obvious suggestion of overfitting since as the training goes on the model is getting worse at generalizing to new data.



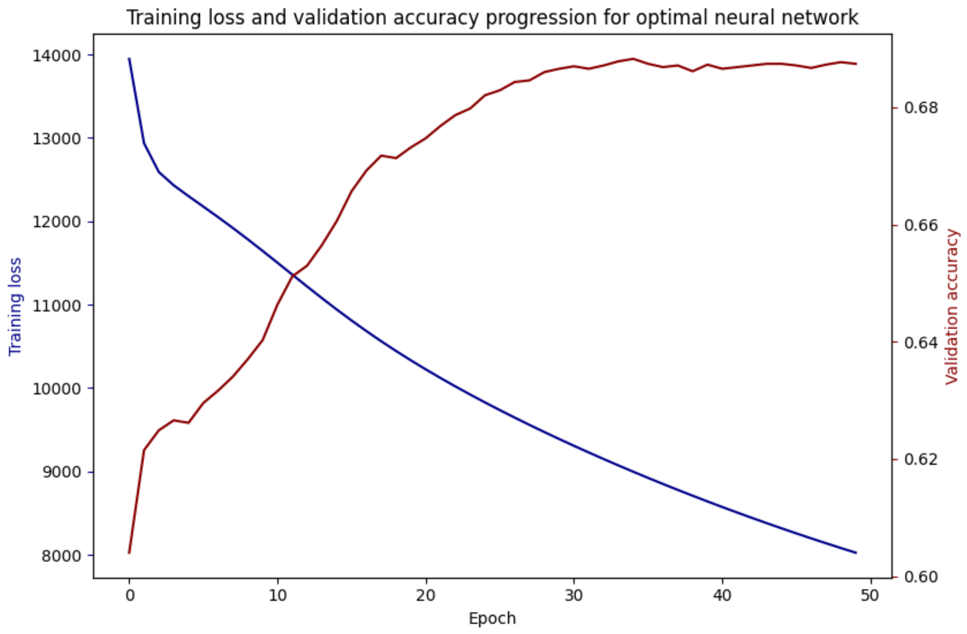
```
Epoch: 47       Training Cost: 7030.5654
Epoch: 48       Training Cost: 6947.6806
Epoch: 49       Training Cost: 6865.9045
Test accuracy: 0.6819079875811459
(csc311) (venv) lakehousey@Masons-MacBoo
```

The final test accuracy of our model is 0.682 as shown in the terminal output.

3.5 Weight regularization

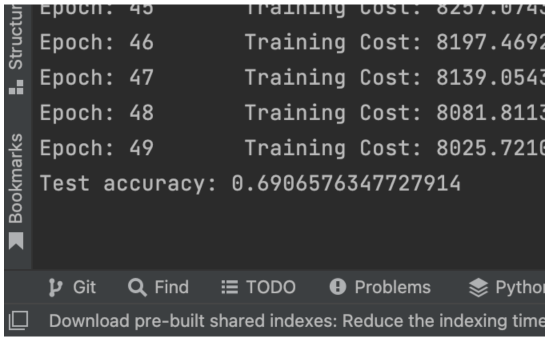


Using our optimal k^* and other optimal hyperparameters, we trained the optimal model with weight regularizations added. As we can see from the plot above, $\lambda = 0.001$ performs best on validation accuracy.



Single hidden layer neural network with 50 neurons. Hyperparameters: lr 0.01; lamb 0.001

The final test accuracy is shown as in the terminal output.

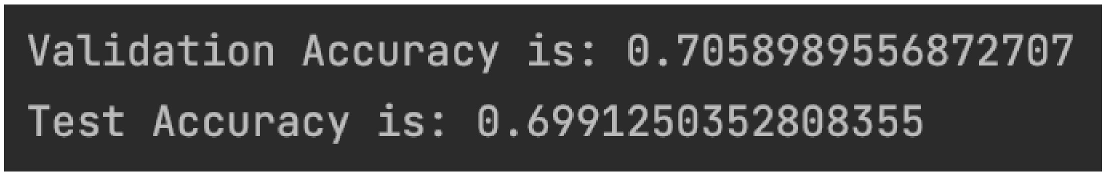


```
Epoch: 45        Training Cost: 8257.074
Epoch: 46        Training Cost: 8197.4692
Epoch: 47        Training Cost: 8139.0543
Epoch: 48        Training Cost: 8081.8113
Epoch: 49        Training Cost: 8025.7216
Test accuracy: 0.6906576347727914
```

The test accuracy is higher than without the weight regularization. Suggesting evidence that the model with regularization performs better. Additionally, overall the validation accuracies also starts declining later and decline less prominently in models with weight regularization.

4 Ensemble

4.1 Final validation and test accuracy



```
Validation Accuracy is: 0.7058989556872707
Test Accuracy is: 0.6991250352808355
```

The validation and test accuracy of the model is shown above.

4.2 Explain the ensemble process

To implement the bagging ensemble, we performed the following operations:
1. Bootstrap the training set (randomly sample training data of the original size with replacement).
2. Train the IRT model using the bootstrapped data.
3. Generate and store the predictions by using the trained IRT model given the validation and test data.
4. Repeat steps 1-3 for 3 times.
5. Average the 3 predictions made as the final prediction.
6. Calculate the validation accuracy and test accuracy by comparing the final prediction with the given labels.

4.3 Better performance?

No, we didn't obtain a better performance using the ensemble. The ensembled accuracy (approximately 0.7059 for validation and 0.6991 for test) is similar to the highest accuracy of a single IRT model (approximately 0.7062). This is probably because the data is sufficient to train the model so additional data may not be that helpful to get a model with better performance.

5 Algorithm extension

5.1 Introduction:

In our final model of neural networks in Part A, we reached a test accuracy of 69%, which in some sense is significant, but has to be extensively improved for the algorithm to have a impact on society.

In this section, I proposed three methods that can potentially optimize the training process and algorithm, and alleviate the extent of overfit. I first implemented a dropout algorithm and achieved a 2% validation accuracy increase. Then I proposed cross-entropy loss instead of MSE for the loss function criterion. Furthermore, while modifying the training algorithm, I came across a potential misimplementation of function train() that would lead to bias, and modified the code and data structure for a better training result.

5.2 Dropout

Formal description The problem with neural networks, especially deep neural networks, is that overfitting comes too fast, a random noise in the data could be readily incorporated into the model given the model has enough hidden layers and neurons. In fact this process is inevitable as the training progresses. As long as they have enough neurons, neural networks are just too powerful to neglect details.

It turns out that dropout is effective procedure to decrease the number of neurons and alleviate overfitting. Dropout is basically dropping out randomly a subset of neurons in the model and only train on the remaining subset. The idea of dropout comes from ensembling. By combining the prediction of results from several separately trained models, we reach an averaged prediction that suffer less from the baises in the training process. By training the dataset on a randomly chosen subset of neurons, we will achieve the same thing: in every iteration, a different subset of neurons come into use. This unique allocation will make them have their own unique access to the whole dataset, which means their training process will be less influenced by or dependent on the existence of other neurons. This in turn leads to every neuron training on more robust features and have better generalizability.

More formally: say originally the forward pass algorithm is implemented as such:

$$z_i^{l+1} = f(\mathbf{w}_i^{l+1}\mathbf{z}^l + b_i^{l+1})$$

where z_i^{l+1} is the i-th neuron in the $l+1$st hidden layer, $\mathbf{w}_i^{l+1}$ is the associated weights of that neuron with, $\mathbf{z}^l$ is the neurons in the previous layer, b_i^{l+1} is the bias, and f is the activation function on the $l+1$st layer. We will initiate a set of Bernoulli random variable

$$r_1^l, r_2^l, \cdots r_{j_l}^l \sim iid \text{ Bernoulli}(p)$$

for each layer where j_l is the number of neurons in the l-th layer.

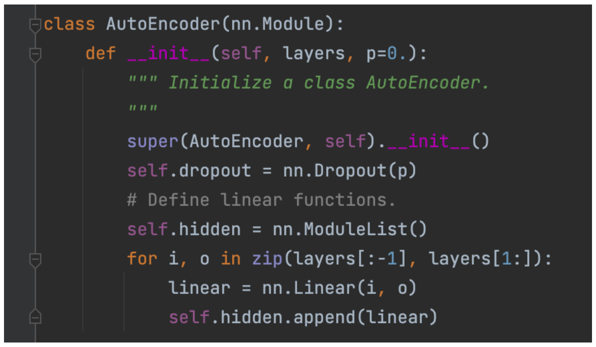
The resulting modified forward pass algorithm with dropout would be

$$z_i^{l+1} = f(\mathbf{w}_i^{l+1}\tilde{\mathbf{z}}^l + b_i^{l+1})$$

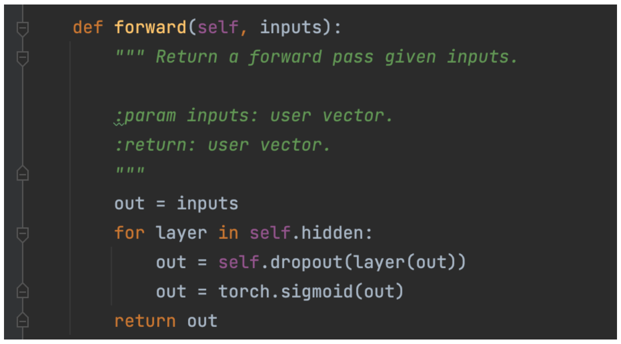
$$\tilde{\mathbf{z}}^l = \mathbf{z}^l * \mathbf{r}^l$$

where $*$ performs the element-wise multiplication.

Fortunately, there is this built-in Dropout class in PyTorch that eases our construction of dropout.



```
class AutoEncoder(nn.Module):
    def __init__(self, layers, p=0.):
        """ Initialize a class AutoEncoder.
        """
        super(AutoEncoder, self).__init__()
        self.dropout = nn.Dropout(p)
        # Define linear functions.
        self.hidden = nn.ModuleList()
        for i, o in zip(layers[:-1], layers[1:]):
            linear = nn.Linear(i, o)
            self.hidden.append(linear)
```



```
    def forward(self, inputs):
        """ Return a forward pass given inputs.

        :param inputs: user vector.
        :return: user vector.
        """
        out = inputs
        for layer in self.hidden:
            out = self.dropout(layer(out))
            out = torch.sigmoid(out)
        return out
```

Above is my modified AutoEncoder class and forward method. I also needed to add a line of model.train() in train() function to distinguish from the evaluation state. I also changed the initializer to take in multiple hidden linear layers. (Acknowledgements to Deep Learning in PyTorch - IBM, a really good online course)

In this algorithm improvement section I did some preliminary comparisons and decided to stick to only two hidden layers, each with 50 neurons, as the best neural network to make further improvements on.

Diagram for intuition

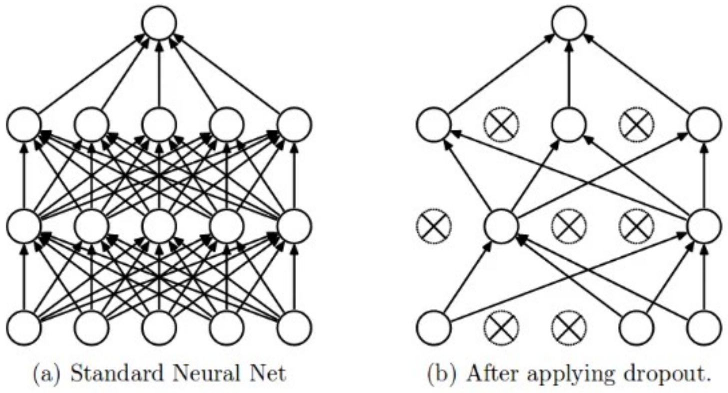
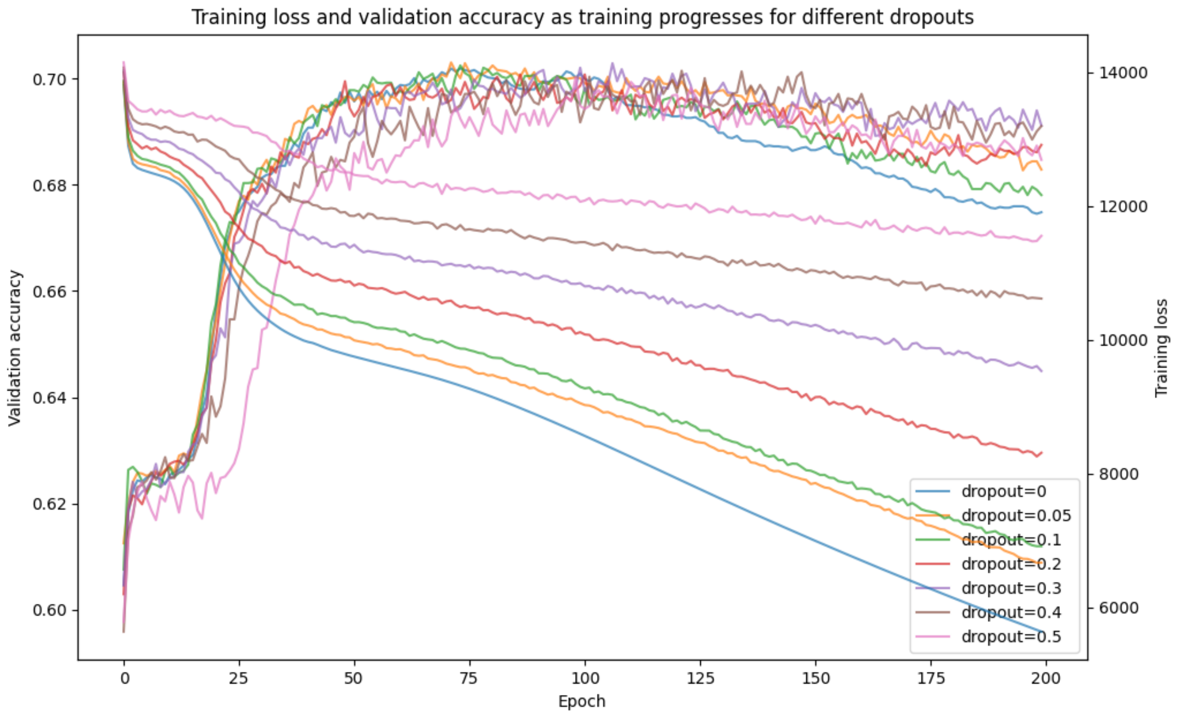


Image by Nitish

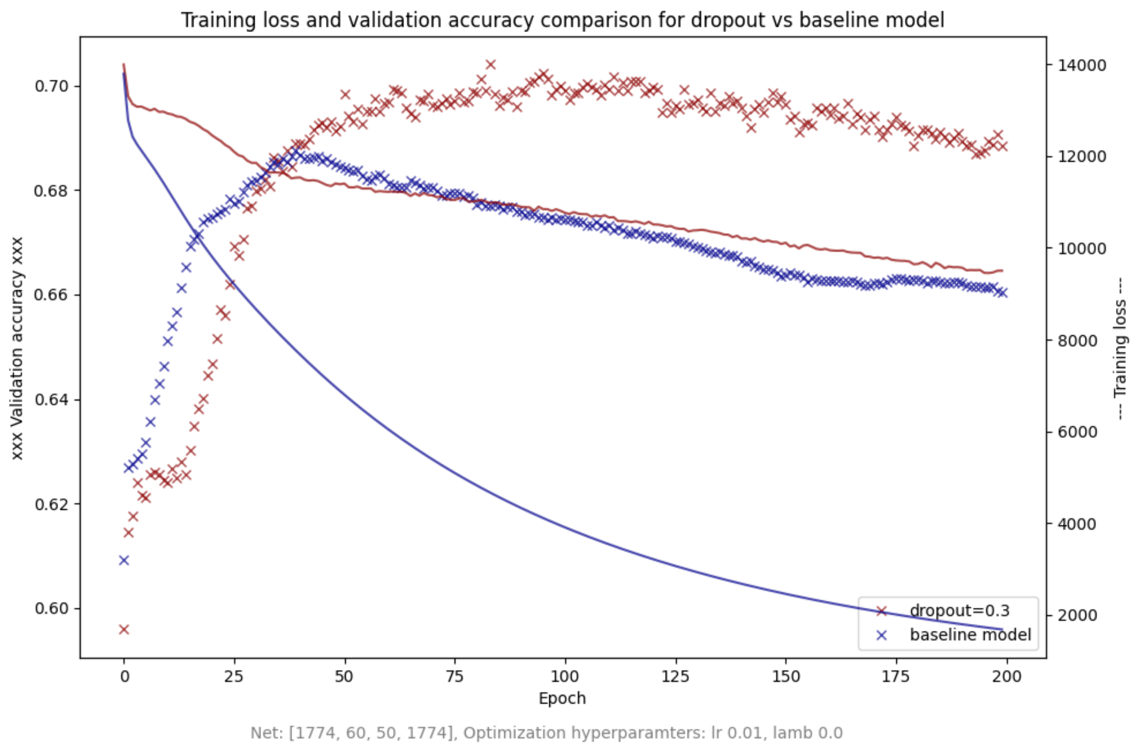
As we can see in the figure, the nodes in the neural network are "turned off" when we apply dropout. And by turning off, I mean completely disregard the node, its link to the previous layer, its effect on the next layer, and its value. The proposed algorithm works well like this because by setting and applying a Bernoulli(p) random variable to the neuron, we have a random chance that the neuron will have a value of 0, effectively turning the node "off" completely.

Comparison and Demonstration



Net: [1774, 60, 50, 1774], Optimization hyperparamters: lr 0.01, lamb 0.0

These are the training results obtained from tuning the dropouts with probabilities 0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5. As we can see the validation accuracies here are pretty much the same even though the training loss drastically varies. This is one of the traits of dropout. We will elaborate on this after we present the comparison between our chosen dropout hyperparameter $p = 0.3$ vs. our baseline model in part A.



This is the comparison of $p = 0.3$ dropout on a $[50, 50]$ hidden layer network with that of no dropout on a single [50] hidden layer neural network. Apparently, although the training loss is high on the model with dropout, the validation accuracy is also unsurprisingly higher. The max validation accuracy in training process is 70.5% for dropout model and 68.7% on the baseline model.

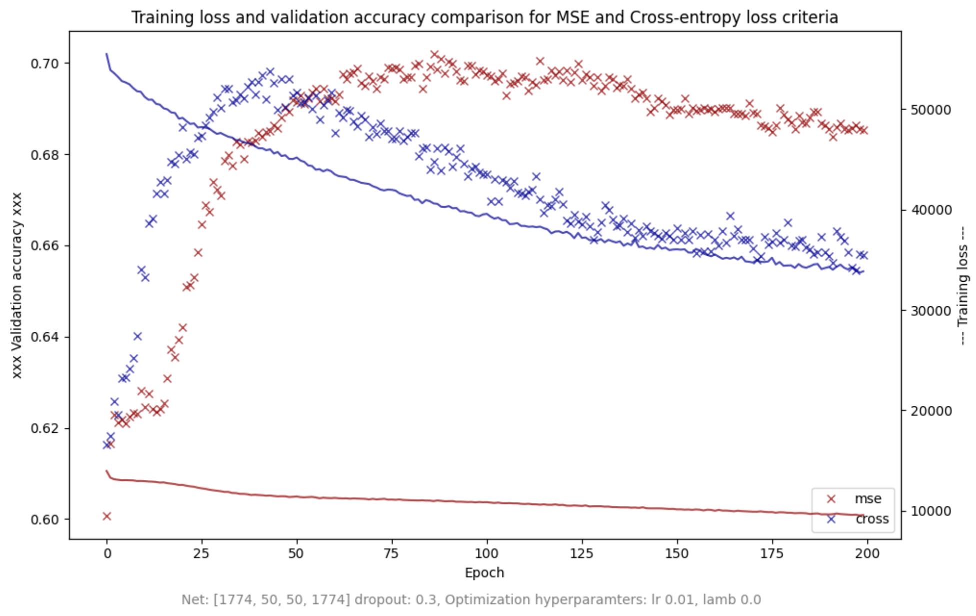
Since the training loss is so high, you cannot argue that the added hidden layer with 50 neurons and the increased model complexity is the reason for this improvement. Hence, evidence is that dropout significantly improved our model performance by making the model more generalizable and robust. Dropout is typical of a model that uses the idea of ensembling as regularization. The reason for this improvement is also extensively discussed in the formal description sections.

Additionally, the validation accuracy plateaued around 70% for a while in the dropout model. This is also contrasts the rapidly decreasing validation accuracy for no dropout model.

5.3 Cross-entropy loss

Cross-entropy loss is a natural criterion for 0-1 loss. It captures the nature of probability and punishes the wrong outputs based on how ridiculous the prediction is. The loss increases exponentially as the probability deviates from the true target. In this report I am not going to elaborate on the idea of cross-entropy since it is already discussed as course material.

However, as a result, there does not seem to be any improvement in validation accuracy when I changed my loss criterion to cross-entropy loss.



This is mainly due to the fact that cross-entropy loss is more prone to overfitting than MSE. Additionally, cross-entropy makes the assumption of a linear decision boundary, which is not required by MSE. Since the loss function criteria are not the main focus of our algorithm improvement (and did not achieve better results), I will not elaborate on that.

However, while I am modifying the algorithm of train function, I encountered the implementation of "zero_train_matrix" and had the following doubt.

5.4 Alternative data structure proposal

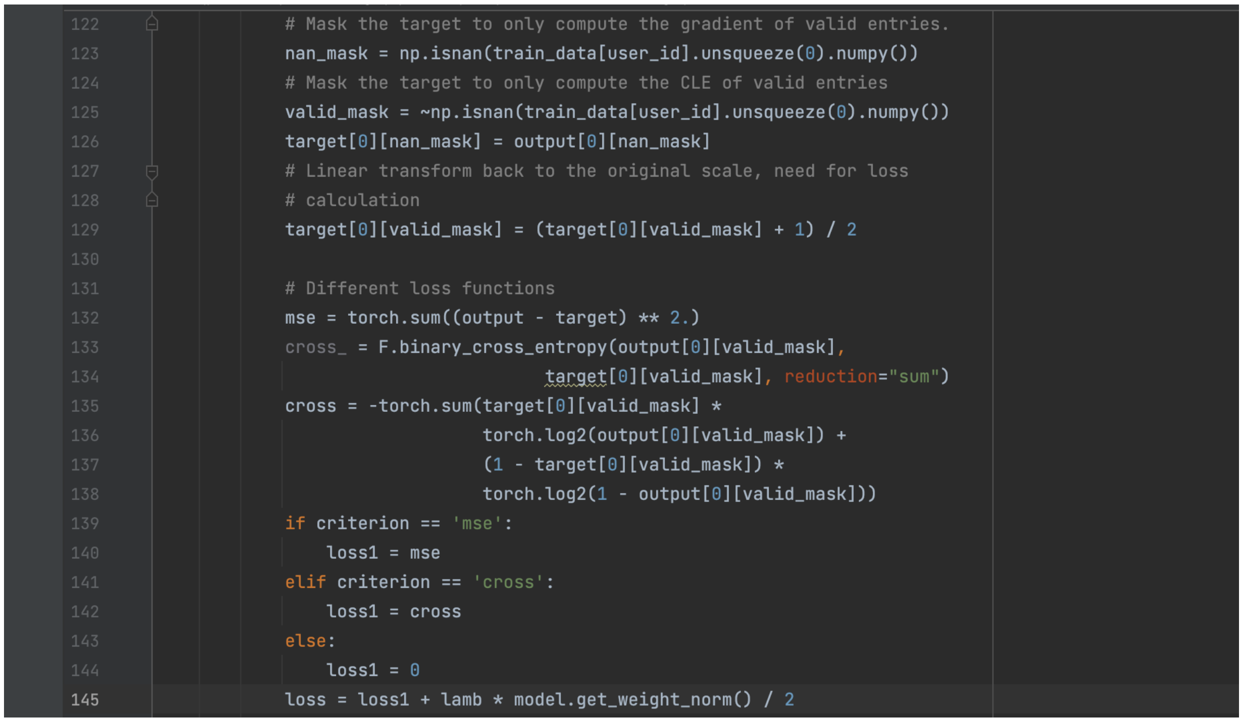
However, while I am modifying the algorithm of train function hoping cross entropy loss would work, I encountered the implementation of "zero_train_matrix" and had the following doubt.

The train_matrix has 0s and 1s and `NaN` values. In the zero_train_matrix however, the `NaN`s are also replaced with 0s, which makes the original ground truth 0s indistinguishable from the place holder/dummy zeros. I thought this was a genuine problem and posted on Piazza (see post @517). However, no one responded.

I decided to try testing out if my doubt is legitimate as well as potentially further improve my model by again modifying the data structure myself.

I first replaced the original 0s with -1s. Then I replace the `NaN`s with 0s. I did this because the original binary values being additive inverses of each other, while the `NaN` being 0s are the only rational assumptions here to make.

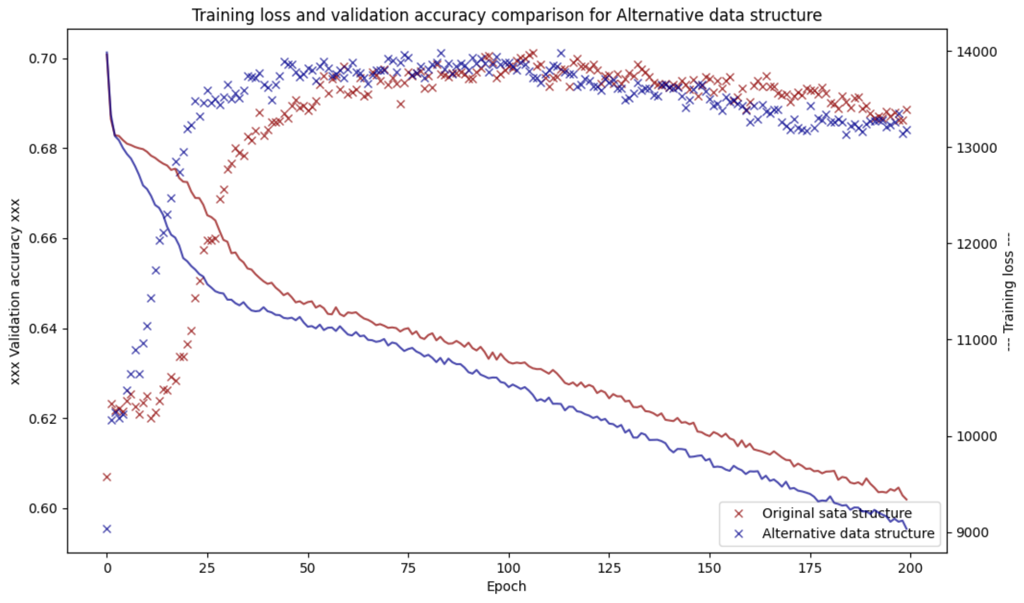
Then, as the output of the model, I changed the output valid_mask values back to 0/1 using the linear transformation $x \leftarrow \frac{x+1}{2}$ to evaluate the loss adequately. The following is my code along with the implementation of cross-entropy loss for the previous subsection.



```
# Mask the target to only compute the gradient of valid entries.
nan_mask = np.isnan(train_data[user_id].unsqueeze(0).numpy())
# Mask the target to only compute the CLE of valid entries
valid_mask = ~np.isnan(train_data[user_id].unsqueeze(0).numpy())
target[0][nan_mask] = output[0][nan_mask]
# Linear transform back to the original scale, need for loss
# calculation
target[0][valid_mask] = (target[0][valid_mask] + 1) / 2

# Different loss functions
mse = torch.sum((output - target) ** 2.)
cross_ = F.binary_cross_entropy(output[0][valid_mask],
                               target[0][valid_mask], reduction="sum")
cross = -torch.sum(target[0][valid_mask] *
                   torch.log2(output[0][valid_mask]) +
                   (1 - target[0][valid_mask]) *
                   torch.log2(1 - output[0][valid_mask]))
if criterion == 'mse':
    loss1 = mse
elif criterion == 'cross':
    loss1 = cross
else:
    loss1 = 0
loss = loss1 + lamb * model.get_weight_norm() / 2
```

The results of this alternative approach:



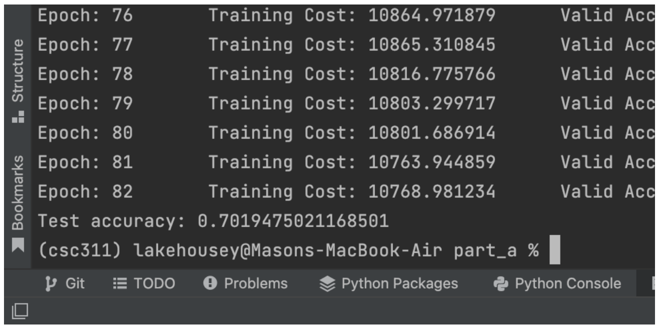
Net: [1774, 60, 50, 1774] dropout: 0.3, Optimization hyperparamters: lr 0.01, lamb 0.0

This result does not show a increase in accuracy of the alternative data structure. But the training loss is in general lower. I conclude that this alternative approach lowers the training loss by providing a more accurate training procedure, but the overall improvement affect on the model goodness is not obvious. It could have minor improvement that is shadowed by the fact that the original data structure is already good enough so that and further improvement would result in overfit. Or it could be that this adjustment has no improvement on the model goodness at all.

Nonetheless, this alternative approach did evidently reduce the training loss. And this leads to the end of my algorithm improvement sections.

5.5 Limitations and challenges

The final testing accuracy on our best model with [50, 50] hidden layers and, dropout $p = 0.3$, criterion MSE and alternative data structure is: 70.2%. More then 1% higher than our baseline model.



```
Epoch: 76      Training Cost: 10864.971879      Valid Acc
Epoch: 77      Training Cost: 10865.310845      Valid Acc
Epoch: 78      Training Cost: 10816.775766      Valid Acc
Epoch: 79      Training Cost: 10803.299717      Valid Acc
Epoch: 80      Training Cost: 10801.686914      Valid Acc
Epoch: 81      Training Cost: 10763.944859      Valid Acc
Epoch: 82      Training Cost: 10768.981234      Valid Acc
Test accuracy: 0.7019475021168501
(csc311) lakehousey@Masons-MacBook-Air part_a %
```

Limitations Dropout is indeed a powerful regularization technique. However, if my research problem gets more complex, or if the pattern in the data gets more complicated, dropout will reduce the model's capacity to make meaningful predictions. This is in nature always a tradeoff between model's generalizability and capacity.

In general, one universal limitation of neural networks is their interpretability. There are over $1774 \times 50 \times 2 + 1774 \times 50 \times 2 + 50 \times 50 \times 2$ weights tuned in this network. And if anyone ask me what these weights represent, how they affect each neuron and our prediction, and how they work for our model, I could not tell them. Neural networks are just trained to work, not to explain.

One other limitation is our data set. Due to the dataset's sparsity, I don't think any model with similar architecture as neural networks would generate an accuracy of over 75%. This due to the lack of feature representations that are expressed in the dataset. Overall, neural networks just need a large amount of data to train on and be generalized to new patterns.

Challenges In this project however, I did more than just dropping out neurons, trying different loss function and alternating this dubious data structure. I also tried out different activation functions, specialized weight initializations, and utilized the concept of momentum in optimizing SGD. These tools or concepts however, do not seem to be useful to any extent in our models goodness. Check out our github repo here for more detailed implementations.

Another challenge I encountered is the computability. In the future, if given enough computability, I would try to apply different levels of attention to our neural network and fit transformers based on the given metadata. This would potentially be able to address the limitations in my research problem.

6 Appendix

6.1 Group member contribution

Mason Hu is in charge of Item-Response Theory and Neural Networks. He designed and carried out the algorithm extension analysis in Part B and also compiled the Latex file.

Yufei Liu is in charge of Ensemble and KNN, also helped with neural networks coding and compiling Latex.

Fucheng Zhuang is in charge of KNN. He also helped with creating Github Repo and documenting our progress.