Report

Team Name: ***Urynbassarov Yerassyl***

Score: 0.629 (XGBoost)

Overview

Since there was a discussion on older version of dataset, I used exactly the old one.
Dataset contains 250 training samples and 19750 test samples, each with 300 features, plus an ID and target outcome. Each header/feature doesn't have name, so it's difficult to make features engineering or attributes selection measure or perform dimensionality reduction. Therefore, I though using Decision Tree isn't feasible in this case.
Similarly, I though lazy learning kNN also won't be feasible because of Curse of Dimensionality and huge testing set, which probably going to take quite a long time roughly 250*19750*300 operations. In terms of computational power it's not that big and works fast, but I think this scale of test/train is too large for using kNN.

Despite all said before, as it was said try kNN first since it might work very-well, I've tried kNN in various ways. Since, I did assumed that kNN won't suit to this solve this task, I made some research and found Approximate Nearest Neighbors.

Generally, I think for this kind of tasks where no information can't be gained by human the Neural Networks would become really handy. Their ability to grasp features, recognize patterns would be really helpful. Therefor I used NN and MLP.
Additionally I just tried how XGBoost works.

Libraries used

1. Sklearn:
   a) Used to preprocess data (MinMaxScaler, Normalization)
   b) Used ML models (KNeighboursClassifier)
   c) Used optimization methods (cross-validation)
2. Pandas and numpy:
   a) Manipulate data (divide features and target)
   b) Create output in required format
3. Pynndescent:
   a) Build an ANN model
4. Keras and Tenserflow
   a) Using to build NN and NLP, optimize and evaluate them.
   b) Adding hidden layers, dropout rates, activation functions and regularization
5. XGBoost
   a) Using to build simple XGBoost model.
6. Matplotlib
   a) Visualize and evaluate outcome of models

Data preprocessing

I used 3 data preprocessing methods from sklearn.preprocessing: MinMaxScaler, Noramalization, StandardScaler for all 300 features.

MinMaxScaler for kNN (metrics=Euclidean, Manhattan, Cosine):

Transform features by scaling them to fixed range between [0,1]

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Normalization for kNN (metrics = cosine) and ANN:

Normalize features by scaling them into their unit form

StandardScaler for NN and MLP, since backproprogation and optimization works better with standard scaled inputs:

$$Z = \frac{X - \bar{X}}{\sigma}$$

Models

kNN

I wanted to only try simple kNN with Euclidean distance and move on to next model of ANN, but after the lecture on text in ML and DL, I was inspired to make kNN with cosine distance. It turned out that kNN method already have cosine metric and other metrics, so additionally I've tried another discussed metric on lecture "Manhattan".

I've used MinMaxScaling for all of the metrics stated above, but I found out that it doesn't fit for cosine distance, therefore I normalized training data for cosine metric.

For deriving number k – I've used cross-validation on training data.

Metric	***Preprocessing***	***K***	***Score***	
			Private	***Public***
Euclidean	MinMaxScaling [0,1]	5	*0.534*	*0.547*
Euclidean	MinMaxScaling [0,1]	29	*0.534*	*0.536*
Manhattan	MinMaxScaling [0,1]	25	*0.540*	*0.539*
Cosine	MinMaxScaling [0,1]	25	*0.544*	*0.549*
Cosine	Normalization ("l2")	27	*0.535*	*0.551*

Summary

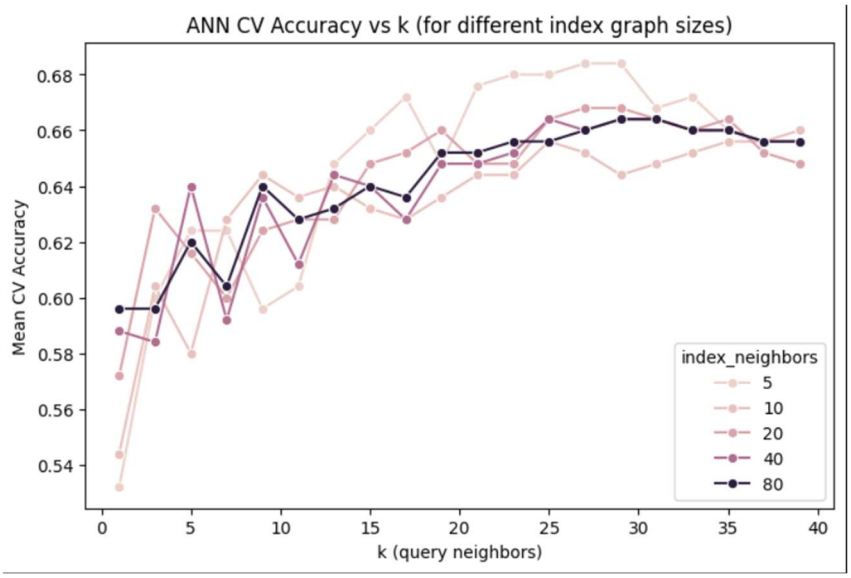
From observation of table, we can see that each model performed almost identically, with slight differences in scores. Also, it's difficult to assume whether if one of the models worked better than another, or could somehow improve the outcome.

ANN

ANN is little more complex version of kNN which works fast but with cost of accuracy. It creates indexes or graphs of probably closest neighbors, and makes decision based on them. It is helpful when working very huge amount data.
There are few libraries such annoy which was developed by Spotify, hnswlib. However, I couldn't import them since they require Microsoft IDE, so I instead I used pynndescent and wrote ANN classification from scratch with help of AI. Additionally, I wrote cross-validation method to find best k number of neighbors and index size.

Overall, I made 3 Submissions for ANN:

1) Random ANN with index size = 20, and k = 5: ***Private – 0.545; Public – 0.559***
2) ANN with index size 20, and best cross-val score k = 27: ***Private – 0.545; Public – 0.541***
3) Testing index sizes = [5, 10, 20, 40, 80] with cross-validation:



Here the best performance is showed by index=5, and k=27: ***Private – 0.546; Public – 0.564***

Neural Networks

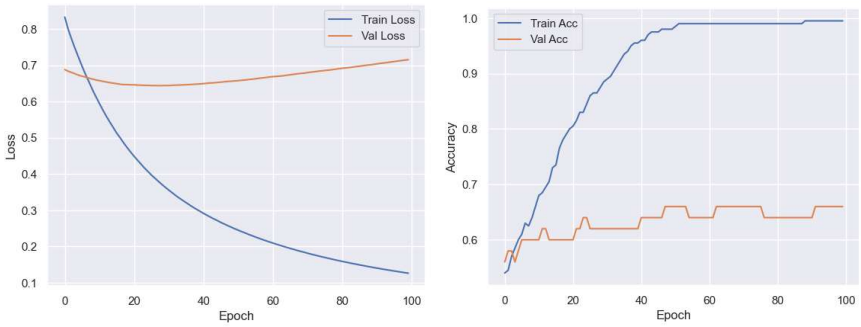
Doing some research, I found out that training and optimizing algorithm like algorithm descent performs worse in not scaled than in scaled. Therefore, I used Standard Scale for preprocessing both training and test data. And I Used tenserflow.keras's Sequential () model to build MLP.
Generally, I did different size of MLP for binary classification, where all hidden layers have RelU activation function and output node has Sigmoid activation function.
All the test were done manually, except keras-tuner.

- During fitting for 2- and 3-layer MLP, 100 epochs was enough.
- Everywhere I used training batch size of 40.
- Used built-in validation split (it didn't shuffle data, but in our case our data doesn't have order so it doesn't really affect; but I didn't do cross-validation). Mainly I used validation_split_ratio = [0.2-0.3]
- Training was done on loss function – "binary_crossentropy", optimizer= "adam", and metrics I evaluated were accuracy and loss of training and validation data.
  Gap between lines of train and validation shows whether model is overfitted or not.
  Bigger gap (Tr-L smaller, Val-L higher): model is overfitted.
  Almost identical: good model.

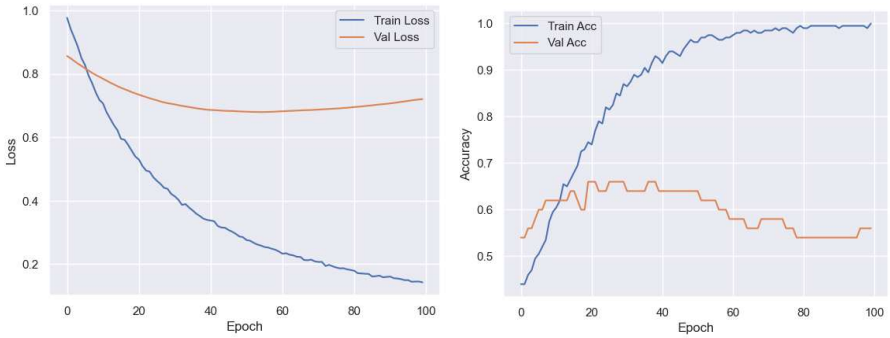
(Neural Networks codes with outputs are about 70 pages because of all epochs, especially for keras-tuner; And tests I've done manually I recorded only in this report, since I was just editing same code snippet for with different parameters)

2 layers

1 Input layer – 300 features;
1 output layer with Sigmoid activation function to classify 1 or 0;
Mini batch size = 40;
Epochs = 100;
Validation_split: 0.2



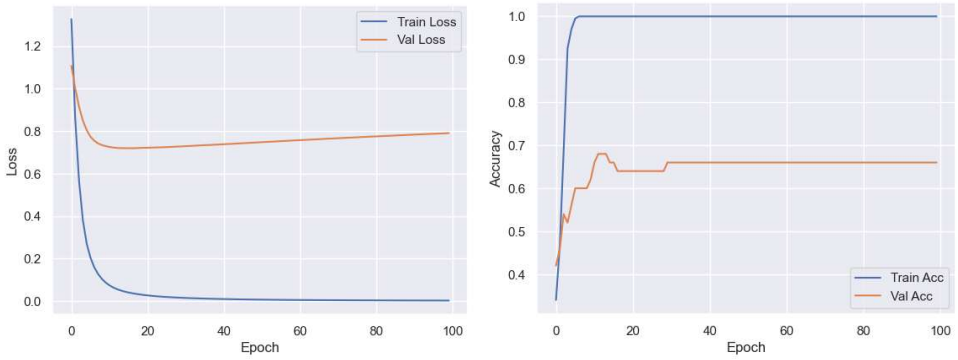
I also tried to dropout input features (dropout rate = 0.01)



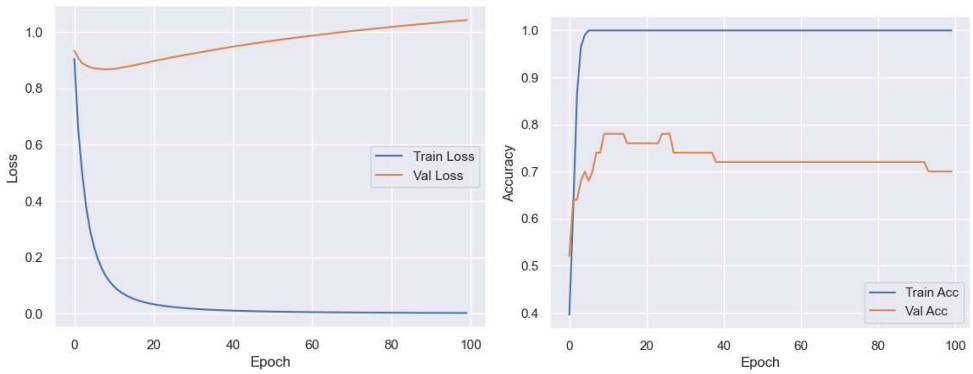
Summary:
2 Layers isn't sufficient for prediction.

3 Layers

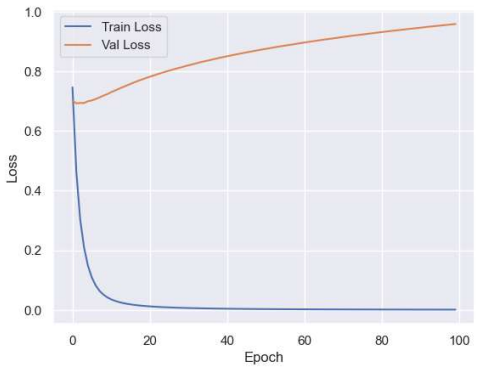
1 hidden layer – 128 nodes:

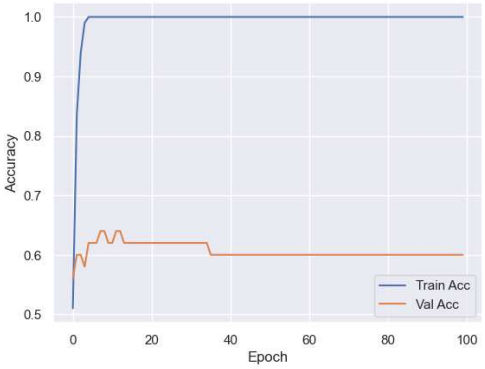


1 hidden layer – 64 nodes:



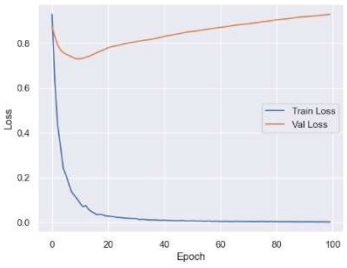
1 hidden layer – 158 nodes:



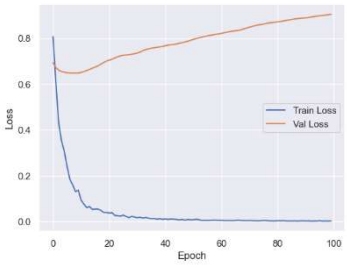


I have tried few more times tweaking width (50-200), validation split (0.2-0.5), and training batch size (20-80). All resulted almost same outcome as pictures above. So, in order to decrease overfitting in another way, I tried same model with dropout in hidden layer:

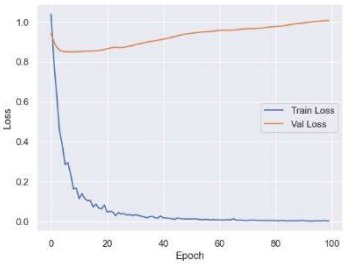
Dropout rate – 0.25:



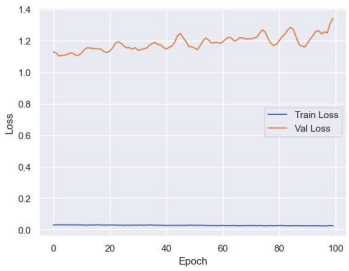
Dropout rate – 0.375



Dropout rate – 0.5:



Dropout couldn't help reducing overfitting.
Trying dropout (0.3) and regularization l2(0.01):

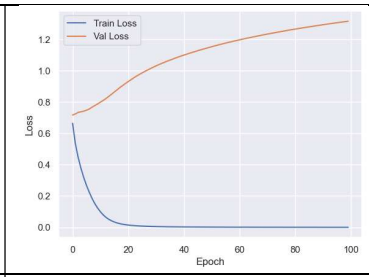
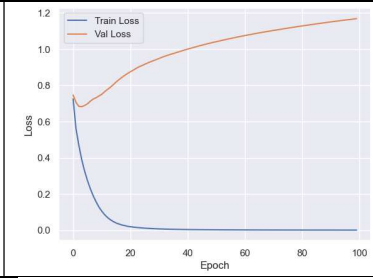
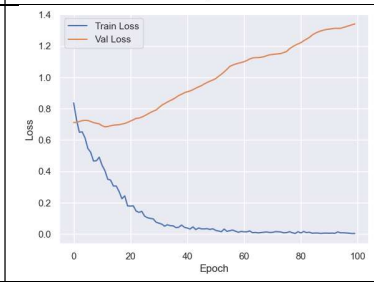
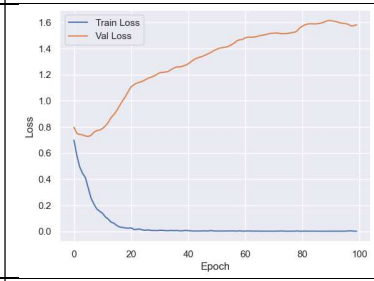
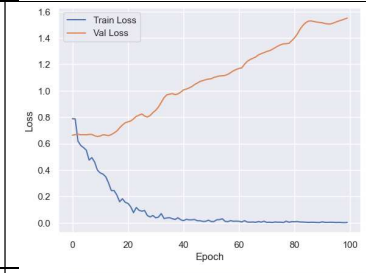
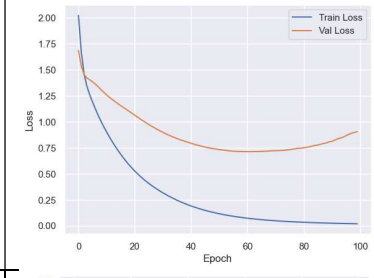
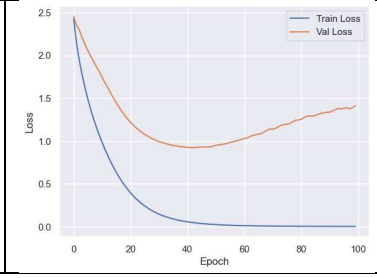


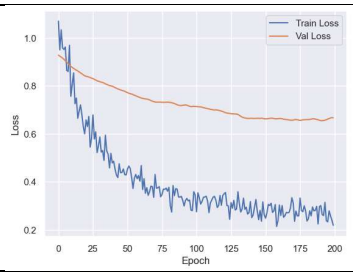
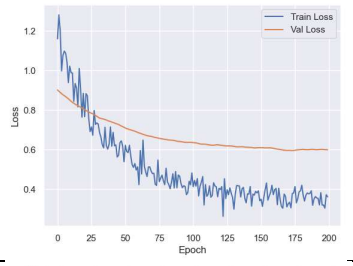
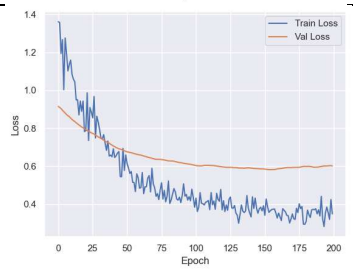
Summary:
3 Layers were not sufficient to predict.

4 layers

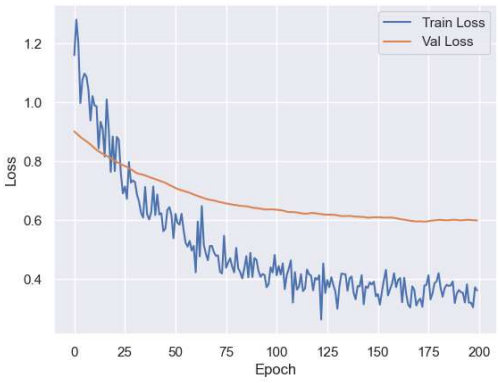
2 hidden layers with drop rates and regularization showed in table below.
Validation split 0.2 or 0.3; generally, 0.3 has better model, less overfit.

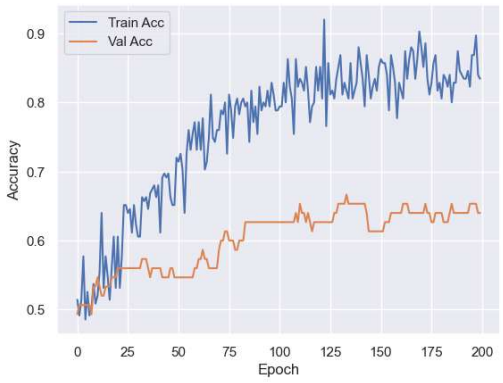
1st hidden layer size	1st hidden layer dropout	2nd hid. layer	2nd hid. Layer dropout	Reg.	Graph: Loss~Epoch Train Loss (Blue)	Last Val error, Val acc

50	0	50	0	No		
128	0	128	0	No		
50	0.3	50	0.3	No		
128	0.3	128	0.3	No		
128	0.5	128	0.5	No		
50	0	50	0	L2 (1st lay.) L2(0.01)		
128	0	128	0	L2 (1st lay.) L2(0.01)		

128	0.3	128	0.3	L2 (1st and 2nd lay.) L2(0.01)		val_loss: 0.6668
128	0.3	64	0.5	L2 (1st and 2nd lay.) L2(0.001)		val_loss: 0.5992; Val_acc: 0.6400
128	0.3	64	0.5	L2 (1st and 2nd lay.) L2(0.01)		val_loss: 0.6017; Val_acc: 0.6267

Summary:
Second to last row is the best model I could test: size of nodes is: [300 -> 128 -> 64 -> 1]; dropout rate for corresponding hidden layers is: [0.3 -> 0.5], and regularization for both hidden layers are L2(0.001). Gap between train and validation become much smaller than before.





Submission of this model gave me Score: ***Private: 0.603; Public: 0.635***.

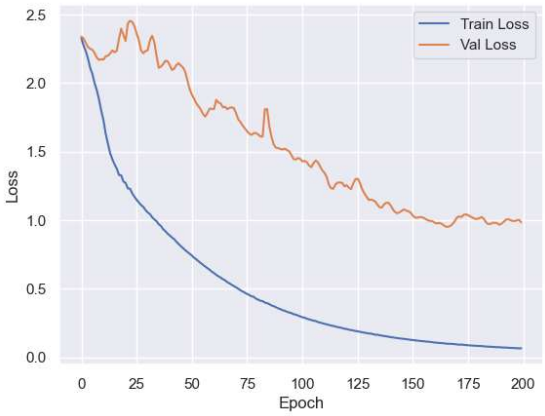
Keras-tuning

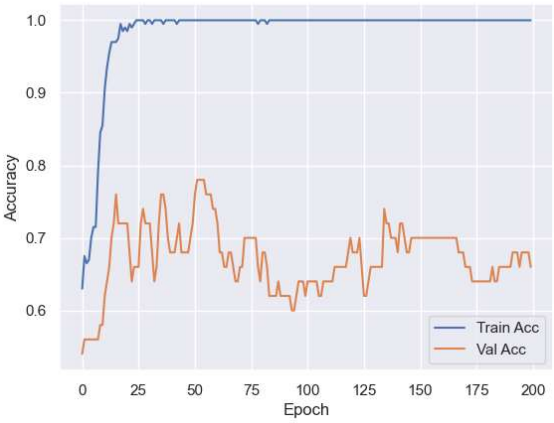
With help of AI, I automated finding best hyper parameters for MLP:

- Decides how many hidden layers to include (in range of 1-4)
- Finds best batch size, and node sizes for layer
- Finds dropout rate and after which layer to use

```
best_hp.values
✓ 0.0s
{'num_layers': 4,
 'l2': 0.0035576956131721355,
 'units_0': 224,
 'dropout_0': True,
 'lr': 0.0005020473830325016,
 'batch_size': 32,
 'drop_rate_0': 0.5,
 'units_1': 32,
 'dropout_1': False,
 'units_2': 128,
 'dropout_2': True,
 'units_3': 96,
 'dropout_3': False,
 'drop_rate_1': 0.5,
 'drop_rate_2': 0.1}
```

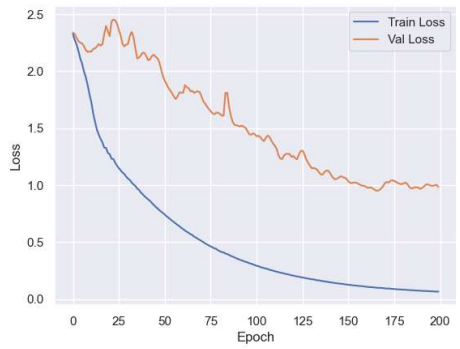
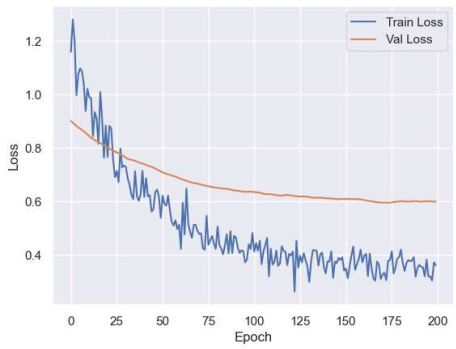
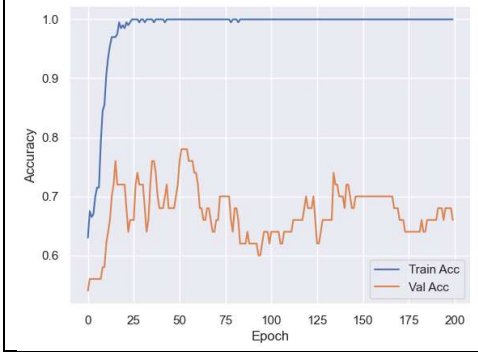
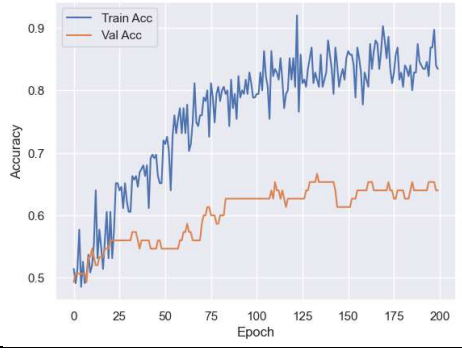
Result shows next:





Score: ***Private – 0.603; Public – 0.641***

Compare Keras-tuned model and Best Manually found model:

Keras-tuned *(model1)*	**Found manually *(model2)***
	
	

Private – 0.603 ***Public – 0.643***	***Private – 0.603*** ***Public – 0.635***

Model1 is more overfitted than model2, but more accurate for training data. In model2, both train and val. accuracy were steadily rising, while in model1 validation accuracy were chaotic or plummeting. Despite everything, both models' prediction of test data is identical.

XGBoost

I just wanted to try how would XGBoost work, and ran simple XGBoost Classifier model for binary classification. And so far, it has the best Private score.

Scores: ***Private – 0.629; Public – 0.641***.

Summary

As I assumed kNN couldn't really predict correctly target data, despite any metrics I have tried. And somehow ANN worked little better than overall kNN, but it was said ANN works faster but with a cost of accuracy. Probably it is because that kNN was to overfitted, and ANN initially won't check all neighbors which made it slightly better than kNN.

My favorite NN and MLP besides performed not very well as I expected, though it still preformed better than kNN and ANN. Generalization of MLP was really difficult and long process, and keras-tuner purpose was to maximize val_accuracy, therefore it couldn't directly solve overfitting problem, and that was the problem. Manually found model was so far the least overfitted model, but somehow they both preformed identically in competition.

And at the last and for me unexpectedly XGBoost performed the best among all models that I've tried, because I assumed that we can't make feature engineering or attribute selection on this dataset.

References:

https://stats.stackexchange.com/questions/286522/is-it-necessary-to-standardize-data-for-neural-networks – Why Standard Scale is better for NN

https://medium.com/@piyush.kailash.chaudhari/importance-of-feature-scaling-for-artificial-neural-networks-and-k-nearest-neighbors-4b7aa618d5ea – Why Standard Scale is better for NN

Applied Machine Learning and AI for Engineers – Used to learn how to write MLP with keras.

https://www.elastic.co/blog/understanding-ann - Understanding ANN

https://www.mongodb.com/resources/basics/ann-search - What is ANN search