AI Q4
Homework Documentation

Tamás Gyenis

Tamás Gyenis

+36301951330

tamgyen@gmail.com

TABLE OF CONTENTS

1 APPROACH

The task to map a purely numerical feature space with continuous data onto the same type of space, only with less dimensions can be categorized as multiple regression. For the general model type I propose a neural network based regressor. Below, I present the steps of data processing, model building and training pipeline development focusing on intuitions behind the decisions, and also highlighting other possible options.

2 DATA PROCESSING

- parser.py

We first separate the target columns from the target dataset. This is done with the *parseTarget* function. After this, the target column values are evaluated based on the -.5 threshold value and corresponding logical columns are created for each cell line (rows), e.g. we set value 1 if the current cell line gene effect is greater than the threshold and set 0 otherwise. We then upshift the effect values by .5 (reasons described later). Then the target columns are concatenated.

The next step is filtering the feature datasets for corresponding cell lines. E.g., we only need data for cell lines that we have targets for. This is done in the *filterByIndex* function, that is ran for the expressions dataframe, as well as for the copy numbers dataframe. The function also removes unmatched cell lines from the target dataframe. We can now inspect the data.

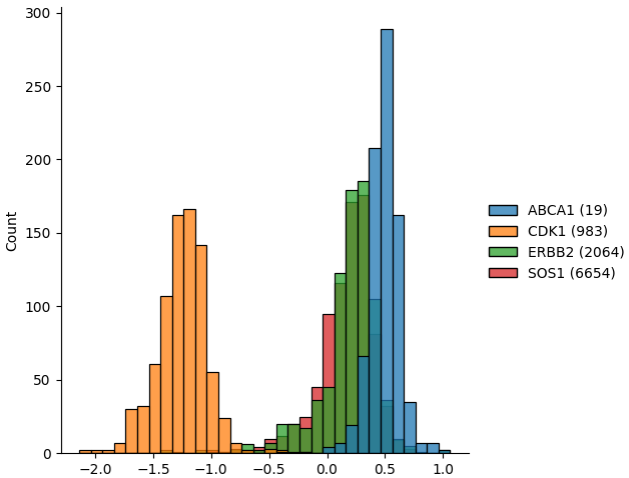

Figure 1: Target data distribution

On Figure 1, we can see the target data distributions along cell lines. The data has normal distribution with no outliers but is differently cantered for the different genes. This should not pose a problem though.

NOTE: We have the option to apply column-wise min-max scaling between [0-1] in the *parseTarget* fnc though. The effects of this offline target data rescaling should be investigated later.

Regarding feature data, see the distribution row wise (values for different genes) see Figure 2 and column-wise (cell lines) on Figure 3.

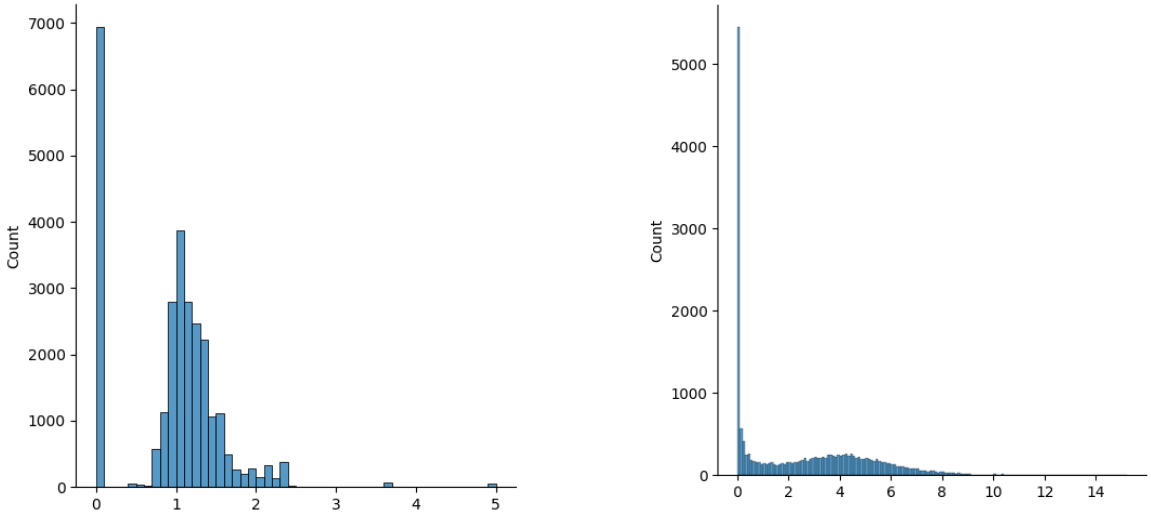

Figure 2: Gene values distribution of row 0: copy number (left) expression(right)

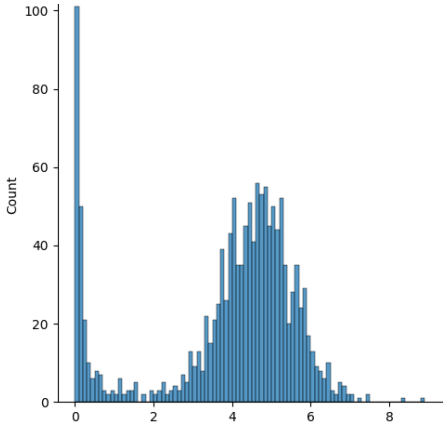

Figure 3: expression cell line data for a gene

The data in both cases is normally distributed with few outliers but is heavily zero inflated. This shouldn't be a problem either if we chose activations correctly. We will see result of training time feature data normalization later though. We finally combine target and feature data and save the dataset.

3 MODEL BUILDING

3.1 AUTOKERAS

- autoKeras.py

For this type of purely numerical data its hard to have initial intuitions about the semantical difficulty of mapping. E.g. for image data we can usually get better intuitions by inspecting image features (mnist classification vs imageNet classification for example). Also, without no previous results from similar tasks, it can be hard to decide on initial model architecture. Although, generally purely dense models work well for regression tasks on unstructured numerical data.

We should try autoKeras [1] first to approximate a model structure. After running autoKras overnight (autoKeras.py) it returned a dense model as expected, that also contained a BatchNormalization layer and a CategoriyEncoding layer (from tf.experimental). See the autogenerated model on Figure 4.

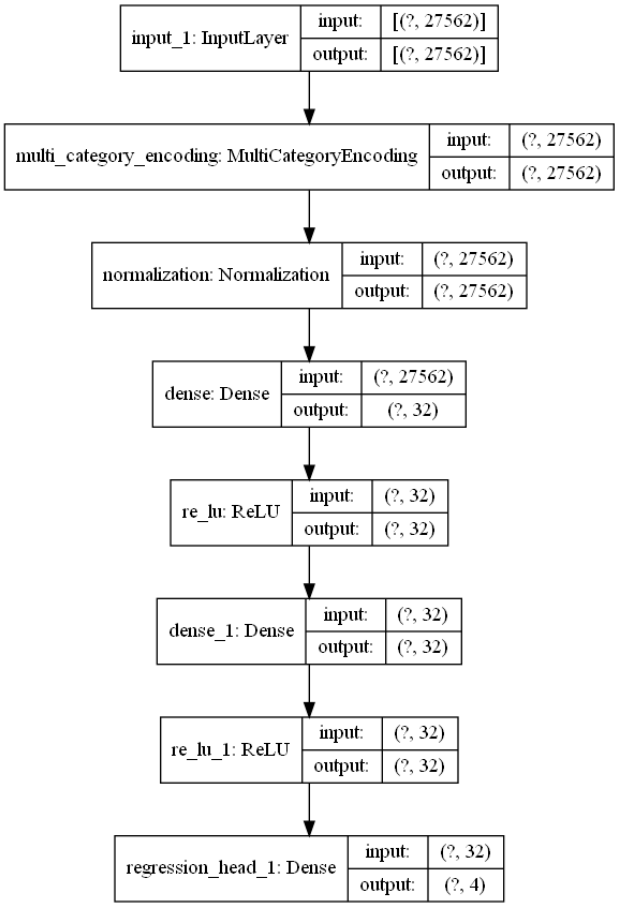

Figure 4: Model generated by AutoKeras

The model has 900k parameters (which seems too much instantly) and achieved a MAE of 0.48. It was only trained on copy number data. We could go on by further searching with AutoKeras, but running searches is timely since the initial graph search uses CPU (for some reason) and only the training is done on GPU after searches. Nevertheless, this validates the idea of using a shallow dense network for the task. We should continue with the evaluation of custom models.

3.2 ADAPTIVE BUILDER

- modelBuilder.py

The training of these shallow, relatively low capacity (param. count) models is generally fast on data with few examples (800 cell lines). Because of this, we can implement an automatic training-testing loop. In this loop a Keras functional API model is built in every iteration using a builder function with changing passed parameters for the different architectural elements. The *buildDenseAuto* fnc builds, compiles, and returns the model based on the passed params.

In order to emphasize on the described error weighing ("Errors in prediction that go over this threshold indicate a significantly worse performance than having similar absolute errors not crossing the threshold."), all the models are built with prediction heads that have 4 output neurons for predicting the logical thold passing values (mapped with the logical columns generated in the target parser), next to the 4 neurons predicting the effect scores. The logical outputs (l1…l4) have sigmoid activation and calculate binary crossentropy loss, while the regression outputs (r1…r4) have linear activations and calculate "closs".

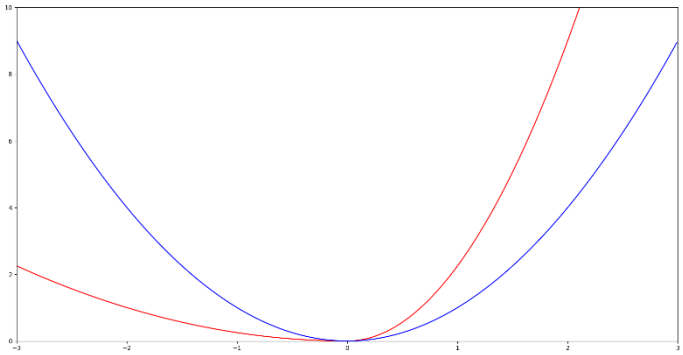

Figure 5: MSE loss (blue) and closs (red)

Closs (for custom loss) is implemented to punish regression neuron predictions more that go over the threshold (which is now shifted to 0 from -.5). See the closs function on Figure 5, and the calculation of the loss in the following snippet:

```
def closs(y_true, y_pred):
    loss = ((y_true - y_pred) ** 2) * (bk.sign(y_true - y_pred) + 0.5) ** 2
    return loss
```

Loss weighing is set to be 1 for closses and .2 for bce losses. For the dense activations we use reLU. For the optimizer adam should be a good first choice.

4 TRAINING PIPELINE

- train.py

Since we want to train and evaluate multiple models, an accelerated pipeline should be used. We first load the dataframe, shuffle it and split to train validation and test sets. Then we initialize tf.data datasets using the *from_tensor_slices* built in fnc. If we were to feed samples in their current form to the multiple output network, the losses would be calculated instantly as an average. This is unsatisfactory, since we want to have different gradients (and losses) at the output. To achieve this, we split the target sample tensors from shape [1,8] into a list of scalar tensors in the *loadDataVector* function. A model graph can accept sample tensors in many forms. From datasets generally tuples are passed (feature, target). Now to preserve the separate scalar tensor input samples for the separate model outputs, we use dictionary keys. If feeded a dictionary, the keys are matched with the model outputs with the corresponding name. (This is why we have to name our layers in the model head). The features can be returned as a rank 1 tensor (shape [1, num_feature_cols]) matching with the Input layer key “features”. After this we map the loader function to the dataset objects. Note that a @tf.function decorator is used in order not to calculate the loading eagerly, but as a graph, speeding up the execution.

After this, the datasets are shuffled (not the test set) and batched. We define the different model building parameters in lists and train the generated model inside a nested loop to try every combination. We train until the val_loss does not improve

for 20 epochs. Also, we save the resulting test error values and model configurations, as well as the training history loss plots.

5 RESULTS

Trainings were done on both the feature datasets separately and also combined, with different model depths widths and batch normalization on or off. With this 90 models were generated and evaluated. The best performing one was "dense_exp_7" that was trained only on gene expression data, with a final MAE of the 4 gene effect predictions of 0.1802. The model has 154k params. The best prediction performance was on gene ABCA1 with the MAE of 0.135, and threshold passing accuracy of 100%. Other results can be seen in the attached test_results_dense.csv.

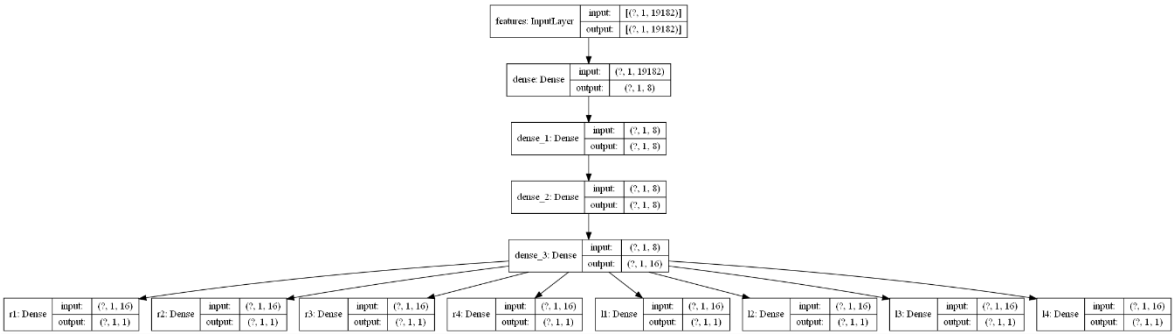

Figure 6: dense_exp_7 architecture

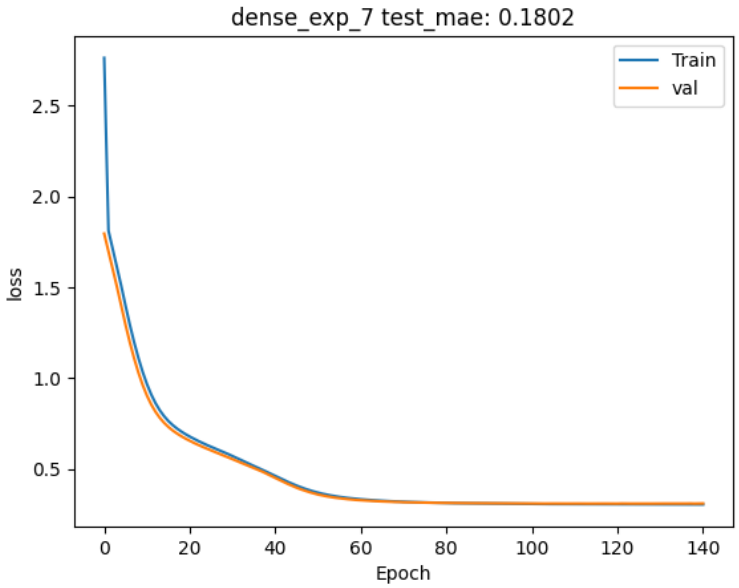

Figure 7: dense_exp_7 training curves

Generally, prediction performance was better for expressions dataset training by about 18%. Normalization does not tend to improve predictions. Performance increase over the autoKeras model is about 45% with a parameter count reduction of 83%.

6 REMARKS AND CONCLUSION

1) In order to try different mappings of the flat input tensor for information extraction some CNNs were also implemented. These had slightly worse performances. A proptotype fnc for further looped cnn testing was also crated in the modelBuilder.py. CNN testing would require more time.
2) Autoencoder models should also be tested
3) Different target data representations should also be tested

At this point a robust model development pipeline has been implemented, that can be used to process measurement data from studies and then train and evaluate different types of regression models with an accelerated pipeline. The following step is an experimentation-based model tuning and different data representation try-outs. Since the goal of the homework was to build the pipeline, I hand over the work in this current stage.

7 REFRENCES

[1] https://autokeras.com/