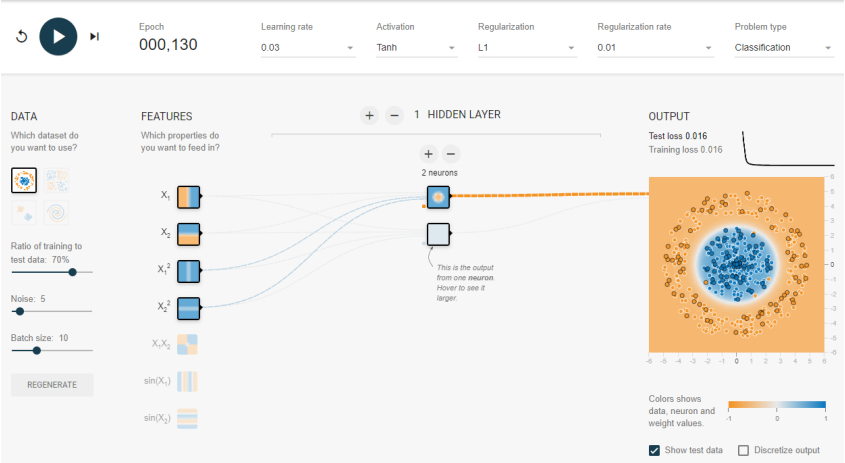
CS 577 - F22 - Assignment 1 Programming Portion

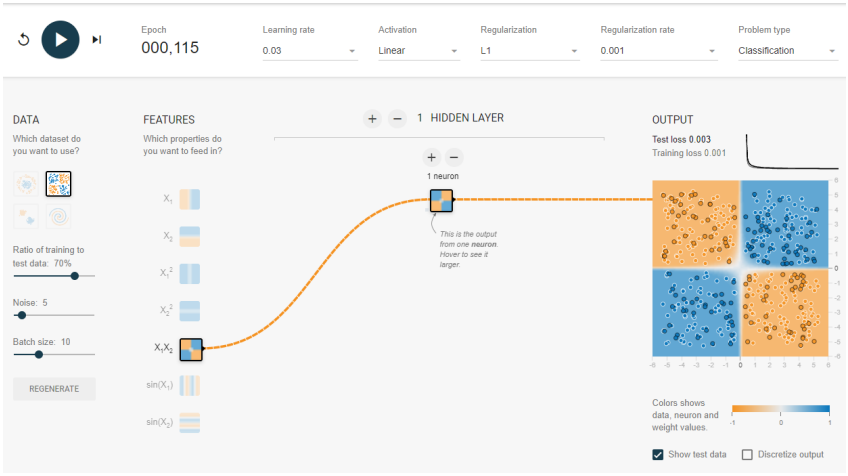
Due date: 09/20/2022

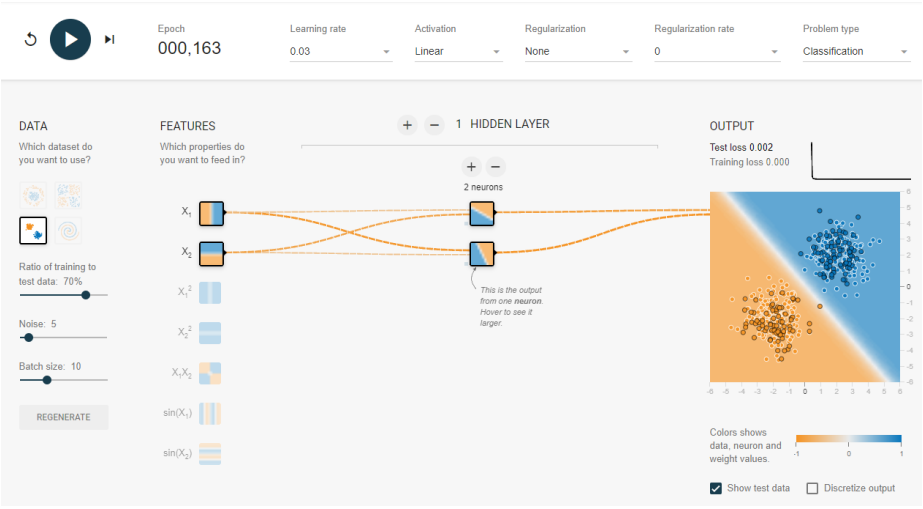
Anas Puthawala - A20416308
Professor Gady Agam



The model stops improving at around ~100 epochs. The hyperparameters are learning rate of 0.03, tanh activation, with L1 regularization at a rate of 0.01. Train test split is 70-30, with a little bit of noise and batch size of 10. There are 4 inputs with 1 hidden layer consisting of two units. The inputs are X1, X2, X1^2 and X2^2. Training loss was 0.016 with testing loss of 0.016.

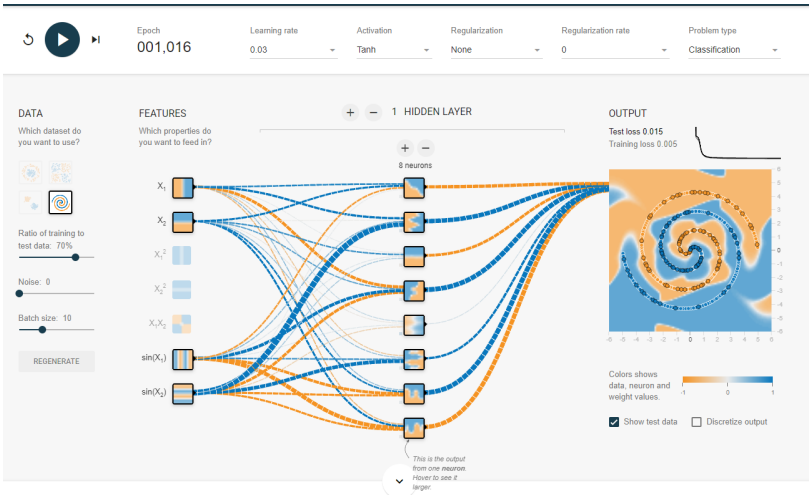
Similar to the previous case, the model doesn't improve much past ~100 epochs. The hyperparameters are learning rate of 0.03, linear activation, L1 regularization at a rate of 0.01. This is a very simple network with one input (X1*X2) and one hidden layer with one unit. The train-test split is 70-30, with noise of 5 and bath size of 10. I obtained pretty good results from this network on this specific data type. The training loss was 0.001 and testing loss was 0.003





The model stops improving after approximately 150 epochs. This network produces a test loss of 0.002 and training loss of 0.000. This is the simplest model that can produce this level of performance. If I were to get rid of the hidden layers, I'd get around 0.004 test loss and 0.001 training loss. The hyperparameters are ~150 epochs, learning rate of 0.03, linear activation, no regularization. Two input units, X1 and X2 with 1 hidden layer consisting of two neurons. Train-test split of 70-30, noise of 5 and a batch size of 10.

S



Lastly, the final spiral circle dataset was a bit interesting. It took around 1000 epochs to get to optimal performance. I had 4 input features, X1, X2, sin(X1) and sin(X2) with a hidden layer consisting of 8 units. The train-test-split was 70-30 with a batch size of 10. My learning rate was 0.03 with a tanh activation and no regularization. Best testing loss I observed was 0.015 with training loss of 0.005 only.

CIFAR-10 Dataset Analysis

The problem statement is to analyze images and develop a multi-class classification FCN for a subset of three out of the total 10 classes. Steps of the implementation for this task consisted of the following:

- Load the CIFAR-10 dataset using tf datasets
- Select a subset of three classes. The three classes I selected were images of airplanes, automobiles and dogs. This was accomplished using a variety of functions, namely np.take.
- Once the subset of the data was created, the data (X, y) was shuffled in unison because when I was extracting the classes I was pulling and simply appending the different classes onto each other, so my X (before shuffling) had all the airplanes, then all the automobiles, and lastly all the dogs in a row. With unison shuffling, the X and the labels respectively were randomized.
- The data, X and y was split into X_train, y_train, X_val and y_val using sklearn's train_test_split. The ratio for this problem was 70% train, 15% val and 15% test. The ratio can differ based on the amount of data available.
- Next I had to perform vectorization on the features array. Initially the features array was a 4D tensor because it had color. To perform vectorization, I took the mean of the color channel to collapse it from 4D into 3D and then used np.reshape to effectively 'flatten' the 3D array into a 2D tensor. This was a 2D tensor of the shape (# of samples, dim of each sample). The dimension of each sample was 32*32 = 1024 (again, because we reshaped).
- After this, I divided all the feature values by 255. to normalize the feature inputs.
- There were no missing values to take care of, and the same vectorization technique for the features was conducted on X_test and X_Val.
- Next, I had to categorically encode the the labels. However, it's important to note that before I categorically encode them I replaced all instances of a '5' in the labels array to a '2'. To explain, see the figure on the right hand side. Recall, that I extracted the airplanes, automobiles and dogs as my subset of three classes, which is the label '0', '1', and '5'. When I go to categorically encode this, it yields an output of dimension 6 because the encoding runs from 0-5 which is a total of 6 values. The extraneous values weren't important because we didn't want the model predicting a half automobile half dog hybrid of some sort so I collapsed the '5' and replaced it with a '2'.

index	label
0	**airplane** (0)
1	**automobile** (1)
2	**bird** (2)
3	**cat** (3)
4	**deer** (4)
5	**dog** (5)
6	**frog** (6)
7	**horse** (7)
8	**ship** (8)
9	**truck** (9)
...	...
...	...

Fig. 1 - CIFAR-10 Labels

Again, the labels don't matter as much because ultimately the integers can be mapped back to a string with the text representing what the label is supposed to represent.

Cifar-10 Neural Network

The initial network was a sequential fully connected network. The input_shape was (32*32) = (1024,) and the activations used were 'relu' all throughout untill we go to the final dense layer (dense_3) which had a softmax activation with three classes, one for each of the labels; airplane, automobile, and dogs.

I compiled the network using the rmsprop optimizer, categorical crossentropy loss because we had categorical encoding to match the loss and the metrics to evaluate the model was accuracy. I fit the model on X_train, y_train and used the validation data as curated. I collected the history of training the model by saving the model.fit argument into a variable called 'history' and created a function to go through the history dictionary and plot the training and validation curves for loss and accuracy to evaluate model training and performance.
Further, I used a batch size of 128 and trained for 100 epochs as a starting point to gauge performance.

```
model = model()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               524800

 dense_1 (Dense)             (None, 256)               131328

 dense_2 (Dense)             (None, 56)                14392

 dense_3 (Dense)             (None, 3)                 171

=================================================================
Total params: 670,691
Trainable params: 670,691
Non-trainable params: 0
_________________________________________________________________
```

Fig. 2 - CIFAR10 Initial Network

Evaluating initial CIFAR-10 model performance - Results

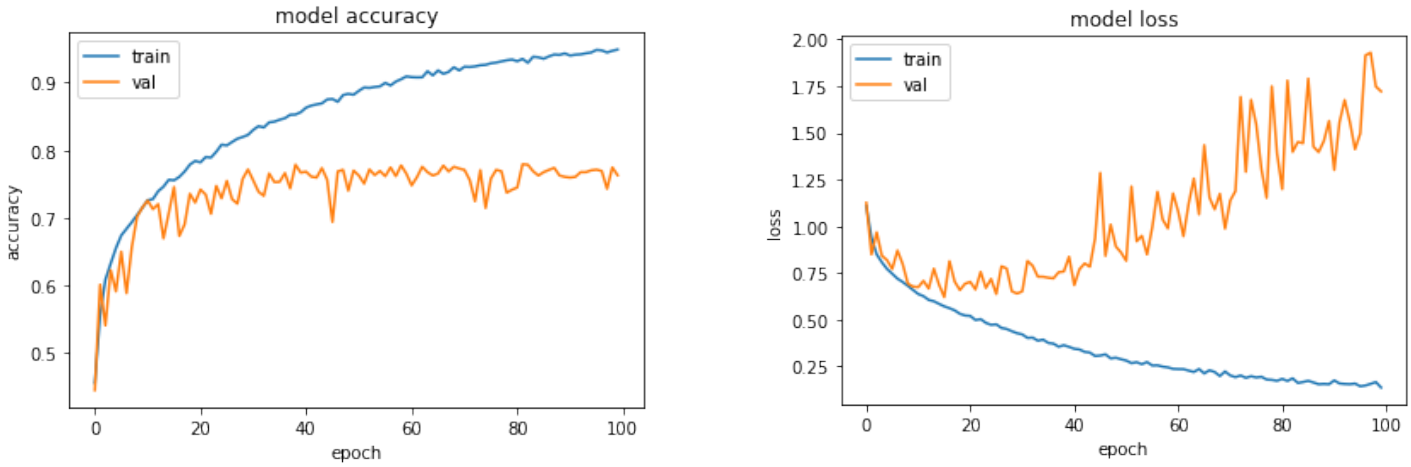


Fig. 3 - Accuracy & Loss curves for initial model for multi-class classification of CIFAR-10 dataset

When looking at model accuracy, we can see that the model stopped improving at around an accuracy of 75% (the exact value using model.evaluate is **.7703**). Compare this to the loss-curve and you can see that the model is over-fitting because the training loss is continuing to decrease whereas the validation loss grows. It's at around the 15-20 epoch range that the model began to overfit. To combat this issue of the model not improving in accuracy and overfitting, I added in some Dropout layers which help combat the issue of overfitting. Dropout is a regularization technique and helps prevent overfitting by randomly dropping a specific rate of neurons in the network during training in each iteration. This way, the model will not be learning the features of the training set too deeply as with the inclusion of Dropout layers the model will 'hide' / 'drop' some neurons in the network.
I also compiled the model this time around with the adam optimizer, current gold-standard for many cases in the industry. The new model is as follows:

```
model_2.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_18 (Dense)            (None, 512)               524800

 dense_19 (Dense)            (None, 256)               131328

 dropout_1 (Dropout)         (None, 256)               0

 dense_20 (Dense)            (None, 128)               32896

 dropout_2 (Dropout)         (None, 128)               0

 dense_21 (Dense)            (None, 56)                7224

 dense_22 (Dense)            (None, 3)                 171

=================================================================
Total params: 696,419
Trainable params: 696,419
Non-trainable params: 0
_________________________________________________________________
```

```
# compile the network
model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics='accuracy')
```

Fig. 4 - Optimized model for CIFAR10 dataset

This time around, I trained for a total of 20 epochs and increased the batch size from 128 to 256 to allow for a more enhanced update in the hopes that the optimizer will A. converge faster and B. learn the decision function better.

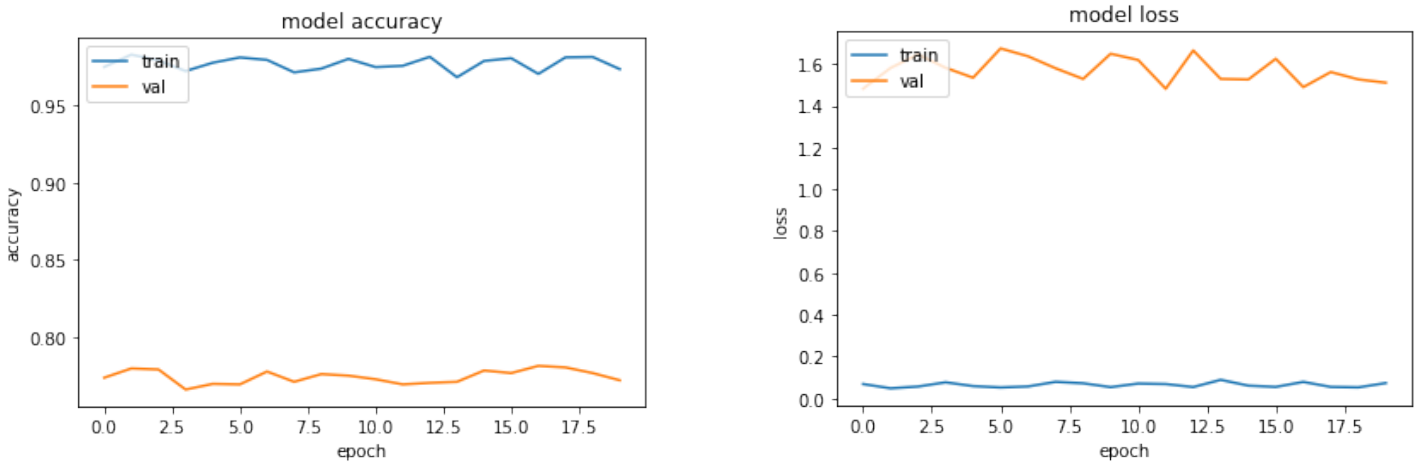


Fig. 5 - Accuracy and loss curves for CIFAR-10 with the improved model.

Deriving conclusions gets challenging as the number of epochs decrease, the curves generally start and end at around the same point, with very slight (if any increases). One thing for sure is that the issue of over-fitting is definitely taken care of with the help of the dropout layers and evaluating the performance of this new model results in an accuracy of **.7817** so there's an increase in performance by changing the epochs, the batch size, the optimizer, and network architecture.

The optimized model is saved as *my_model_2.h5*.

Spam Email Data Analysis (Spambase)

The goal of this problem was to acquire data from the UCI ML Repository and develop a binary classifier to classify wether an email with various features is considered spam or not. In addition, we had to perform preprocessing to be able to pass the tensors as suitable for a network, and perform hyper-parameter optimization to improve performance as completed in the Cifar-10 dataset. The details for the implementation are as follows:

- Use pd.read_csv to read the .data file as a pandas dataframe
- There were various features, but the last column in the dataframe was our prediction goal. This was a column representing 1s and 0s wether or not an email given the various other features was spam or not.
- With that in mind, I separated the features from the labels (created X and y). X is simply the dataframe that does not have the last column (y) and y is the last column.
- I checked for any missing values and there were none
- I proceeded to split the features and the predictor (X, y) into training, validation and test sets. For this dataset I used a 70, 20, 10 split respectively in order to optimize the training and validation portion.

- Lastly, I noticed some features were drastically higher than others (like for example, there were some features which columns had values such as .15, .20, etc. and others which had 100, 200, etc.) and when training a model with this type of variance it can read to vanishing / exploding gradients. To counter this, I normalized the features by using Sklearn's Normalizer which basically normalizes the samples individually (each column) to unit norm. Each sample (i.e. each row of the data matrix) with at least one non zero component is rescaled independently of other samples so that its norm (l1, l2 or inf) equals one.
- It's important to note that normalization was only conducted on the features, the labels (y) were left as they are in binary format as that's valid for training.

Spambase Neural Network

The network utilized is a sequential network consisting of fully connected dense layers of 56 neurons, 32 neurons and the final output neuron of 1. The input_shape is the dimension of one feature of X_train. The activations used are 'relu' activations except for the final neuron which contains a sigmoid activation. The optimizer used is Adam with a learning rate of 0.01. The loss used is binary crossentropy because we've got two classes for our prediction, spam or not spam. In other words, this is a binary classification problem. The metrics used are 'binary_accuracy'. The initial model was trained for 100 epochs with a batch size of 32 and the history of training was recorded to generate the following loss curves.

Evaluating spam binary classifier model performance - Results

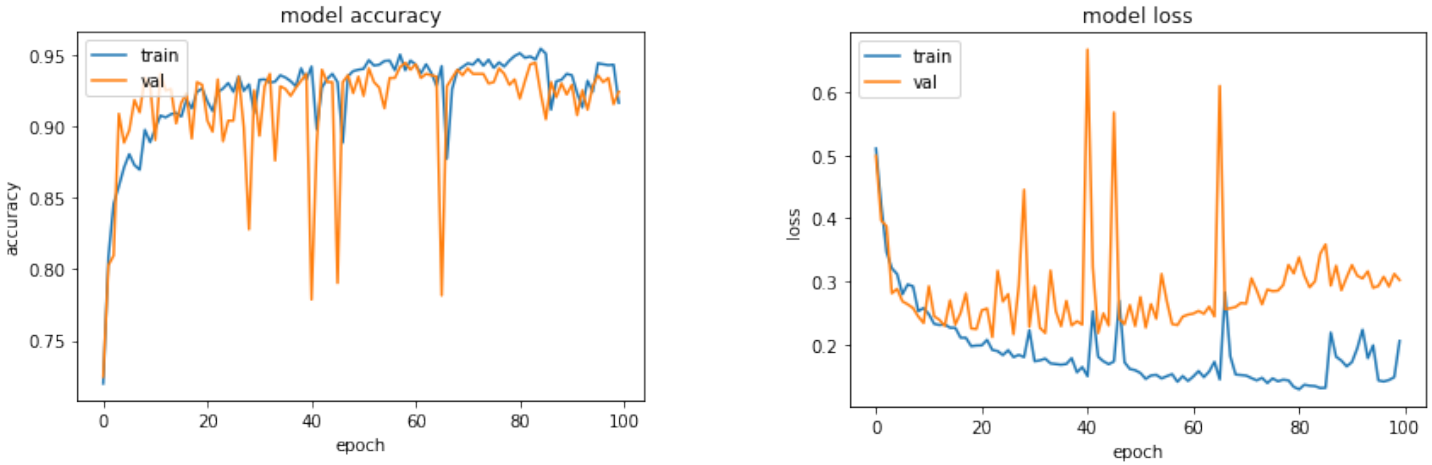


Fig. 6 - Accuracy & Loss curves for initial binary classifier

We can see that overall it's a pretty decent fit. One thing to note is that there is very slight over-fitting as seen from the model loss curve on the right. As we reach around 60-70 epochs the model validation loss begins increasing ever so slightly whereas the model training loss continues decreasing; a tell-tale sign that a model is beginning to overfit. To combat this issue of overfitting I'll add in some dropout layers in the hyper-parameter tuning portion. Lastly, when evaluating this initial classifier on the testing set we get a binary accuracy of **.9111**.

I went back and edited a few things such as:

- Adding dropout layers with rate of 0.10 (meaning that it'll drop 10% of neurons randomly during the training process to prevent over-fitting)
- Changing the number of neurons per dense layer from 56, 32, 1 to 32, 32 , 1
- Changing the learning rate for Adam to 0.015 instead of 0.01
- Reduced the number of epochs to 50

I kept the batch size, the loss, and the metrics for evaluation the same. And trained the model, plotted the same curves and evaluated the results from there.

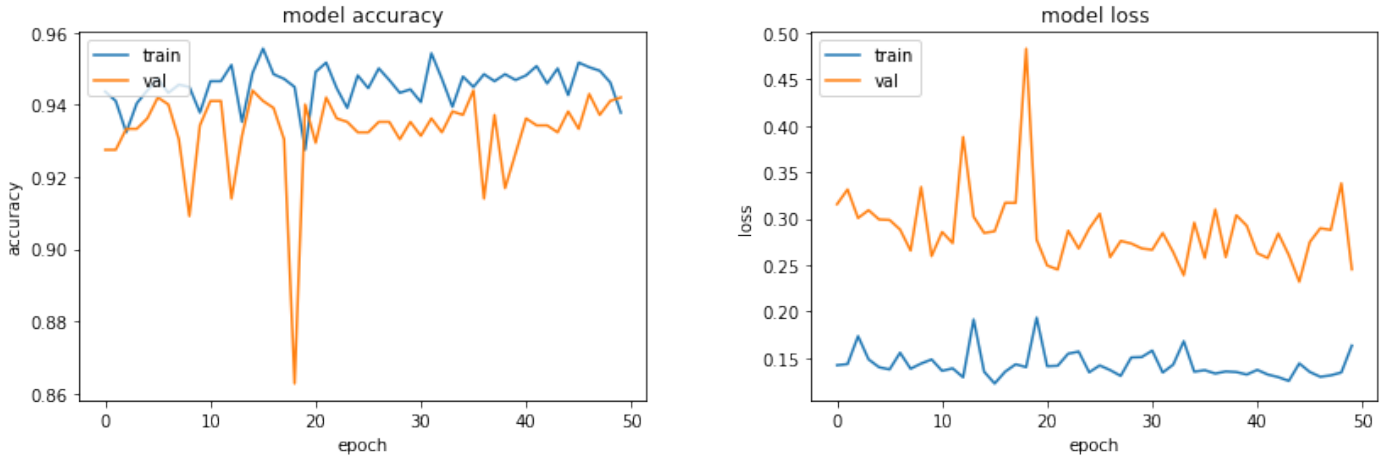


Fig. 7 - Accuracy and loss curves for optimized binary classifier - Spambase

Few things to note:

1. The curves are not as jaggedy with extreme spikes upwards and downwards when compared to the initial classifier
2. Next, looking at the loss curves from the initial model to this model we can see a definite improvement in that there's no distinct sign of over-fitting occurring. 50 epochs seems to be the ideal amount to train because after around the 50 epochs it seems that it might begin to spike and go upwards.

Last but not least, when evaluating the new models performance, I got an accuracy of **.9349**. The older model had an accuracy of **.9111** so there's definitely an increase from adding dropout layers, increasing the learning rate, and changing the amount of neurons in each dense layer. This model is saved as *spambase_model_2.h5*.

Communities & Crime Data Analysis

The goal for this dataset is to predict 'ViolentCrimesPerPop' which is a numerical (decimal) value. This is a regression task, unlike the previous tasks which were multi-class classification and binary classification. The data was acquired from the UCI ML repository and loaded into python using pd.read_csv. To gauge the amount of missing values I replaced the '?' with np.nan's and used df.isna().sum() to see a count of missing values per column.

DEALING WITH MISSING VALUES

- You can impute missing values with the mean, median, mode, or you can use a model such as KMean or KNN or even learn a classifier to impute values.
- Main caveat to this approach is that sufficient data is needed. In this dataset, there was an insufficient amount of data to accurately be able to impute the missing values. If we had sufficient data and imputing was needed, I would've followed the methods mentioned in the first bullet point.

In addition, There was a column containing county names which also needed to be dropped. If I went to categorically OHE the county names then it would blow up the dimensionality of the feature set considering that there were 1828 unique county names in a dataframe of around 1994 columns to start with. Blowing up the dimensionality to categorically encode the county names wasn't a wise decision. To better justify why I am dropping the entire column imagine there are 100 rows and one of the columns has an ID feature, and practically each ID is unique. If we include that ID feature in our model when we train, then #1 it most likely will lead to collinearity and #2 it simply won't hold much value when trying to fit a decision boundary for that feature.

After the missing values were taken care of, the entire dataframe was split into X (features) and y (labels), the labels were the column 'ViolentCrimesPerPop' and the features were everything except for that column. The train_test_split function from sklearn was utilized to split up the data from X, y to training set, validation set, and testing set with a split of 70, 20, and 10 respectively. I also noticed some features which had values in the range of the thousands and some who had decimal values and to counteract this issue I performed normalization using sklearn.preprocessing.normalizer function to bring each column to its unit norm.

K-FOLD CROSS VALIDATION

In addition, a technique called K-Fold cross validation was utilized to more accurately gauge the performance of the model during the training stage. The way K-Fold cross validation works is that it partitions the entire training data-set into 'K' folds (in this case it was 5). It initially holds out one of the K folds as a validation set and trains on the remaining k-1 partitions. It uses the validation set to assess the performance and the loss of the model at the given epoch. In the next iteration, another fold (not the same one) gets taken out as a validation set and the other k-1 folds are used as training set. This way, we achieve a more robust evaluation of the models performance as it exposes itself to the entire dataset rather than just one specific partition of the training data and one specific partition of the validation data. Oftetimes, K-Fold Cross Validation is used when there is lesser data to assess a model's performance / loss.

Crime Data Neural Network

The initial model consisted of a sequential model with three dense layers. The first and second layers consisted of 32 neurons both of which had 'relu' activations followed by the final dense layer which had no activation. The model was compiled with the rmsprop optimizer, mean square error loss (as it is a regression task) and the evaluation metric as mean absolute error. Note that another acceptable evaluation metric is root mean square error. During cross-validation, the 'K' chosen was 5 and the number of epochs I initially chose to train for was 75 with a batch size of 16.

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 32)                3264

 dense_10 (Dense)            (None, 32)                1056

 dense_11 (Dense)            (None, 1)                 33

=================================================================
Total params: 4,353
Trainable params: 4,353
Non-trainable params: 0
_________________________________________________________________
```

Fig. 8 - Crime Regressor Initial Model

K-Fold Cross Validation Process & Results

The function np.array_split() was used to split the training and validation sets into 'k' amounts. There is also another function, np.split() which performs the same thing, but the number of samples for np.split() to work must be exactly divisible by 'k', if it is not then the function will not work. np.array_split takes care of the divisibility aspect by the last fold having the remainder of the samples. In this case it wasn't a huge difference as the last fold had simple 1 less sample when compared to the other folds. Following the procedure for K-Fold Cross Validation, the model was fit on the X_train, y_train partition and evaluated on the X_val and y_val. I collected the history of each training and validation sample, and made a function called *plot_validation_curves()* which simply took the input of the array of losses or MAE's, and took the mean column-wise. Since there were 5 folds, there were a total of 5 iterations in which the loss and the MAE was collected and the mean was taken by just adding up the column-wise value and dividing by the number of folds. This was all done through one function call, np.mean(). Lastly, I plotted the averaged out error's and the evaluation metric's (mae) with respect to the number of epochs to yield:

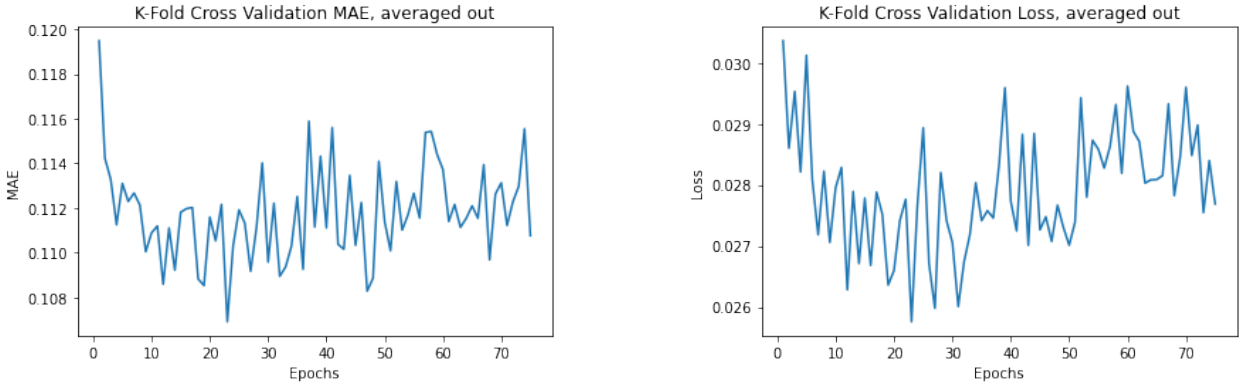


Fig. 9 - K-Fold Cross Validation Results

Few things we can note:
- Very jaggedy, lots of spikes which can mean changing the learning rate a bit
- The MAE seems to be at its lowest at around 22 - 23 epochs, with a value of approximately **0.106**
- After around 20 - 25 epochs, the model starts to get worse and show signs of over-fitting.

We can optimize our final model now based off these results in the following manner:
1. Add some dropout layers with a rate of .10 to combat the over-fitting
2. Change the learning rate (try increasing and decreasing) to 0.015
3. Try a different optimizer, specifically Adam
4. Change the number of neurons in each dense layer (went from 32, 32, 1 to 64, 32, 1)
5. Alter the number of epochs (reduced them from 75 to 50)

I kept the batch size, the loss function, and the metric for evaluation the same. And lastly, I trained the model on the entire training set and evaluate the model using the entire validation set and my results are as follows:

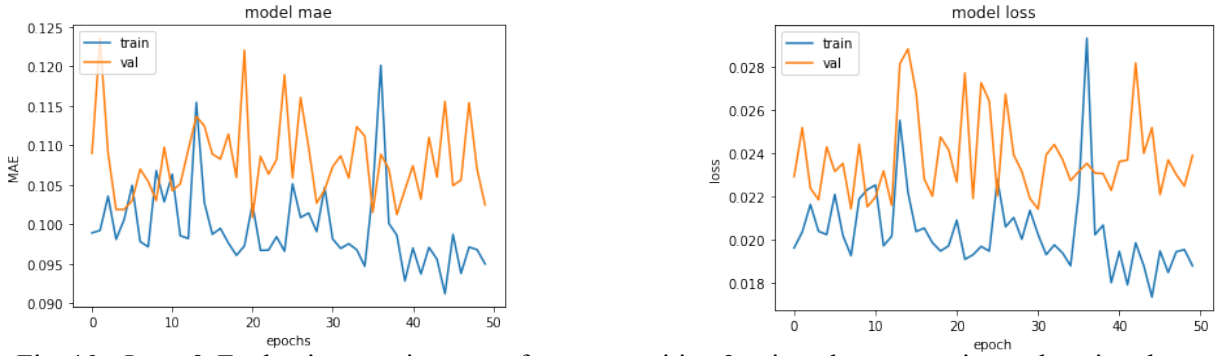


Fig. 10 - Loss & Evaluation metric curves for communities & crime data regression task, using the optimized network.

Few things to note:
- Data is still a bit jaggedy, learning rate adjustment could help with that, changing the optimizer to Adam helped the model learn a bit better (overall slight improvement in MAE) and decrease in Loss.
- When evaluating the model performance, it lead to a MAE of **0.0926.** So with the hyperparameter optimization there is an improvement. Further work could be directed in adjusting / tuning the learning rate and batch size in order to produce smoother loss / metric curves and produce more 'polished' learning.
- The model is saved as '*model_crime_2.h5*'.

References

1. https://www.tensorflow.org/api_docs/python/tf/all_symbols
2. https://numpy.org/doc/stable/reference/generated/numpy.mean.html
3. https://numpy.org/doc/stable/reference/generated/numpy.split.html
4. https://numpy.org/doc/stable/reference/generated/numpy.array_split.html
5. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
6. https://keras.io
7. https://www.tensorflow.org