Detecting Bitcoin Ransomware Transactions Using a Neural Network

Matt S, Eish K, and Kyle M

Motivation

Our primary motivation for this project was to learn new libraries and skills. Since we didn't create neural networks in the lab portion of this class, we decided to explore that with this project- from our domain knowledge, we have seen the rise of neural networks in industry. Next, we wanted our project to have a real world application, as we wanted it to serve a functional purpose. While looking through datasets, one that stuck out to us was a Bitcoin transaction dataset. We chose this dataset because Bitcoin and the blockchain are becoming increasingly relevant, with the acceptance of Bitcoin reaching its highest levels and the blockchain concept becoming the backbone of several applications today. Because of the Wild West nature of cryptocurrency transactions and the inherent decentralization of it, it has become a prime vehicle for ransomware payments[1]. In May of 2021, the Colonial Pipeline went offline as it was hacked and held for ransom. The owners ended up paying ~$4.4 million in cryptocurrency. This led to the first time the average gallon price of gas being $3 in seven years[2]. Several other high profile cases have occurred and are increasing in financial magnitude(Figure 1). Our goal was to create a neural network that could identify such transactions. We identified neural networks as a good use case for this problem, mainly because ransomware detection relies on making connections that humans usually cannot make, and the iterative aspect of the neural networks in fine-tuning itself worked well with the large number of available transaction data.

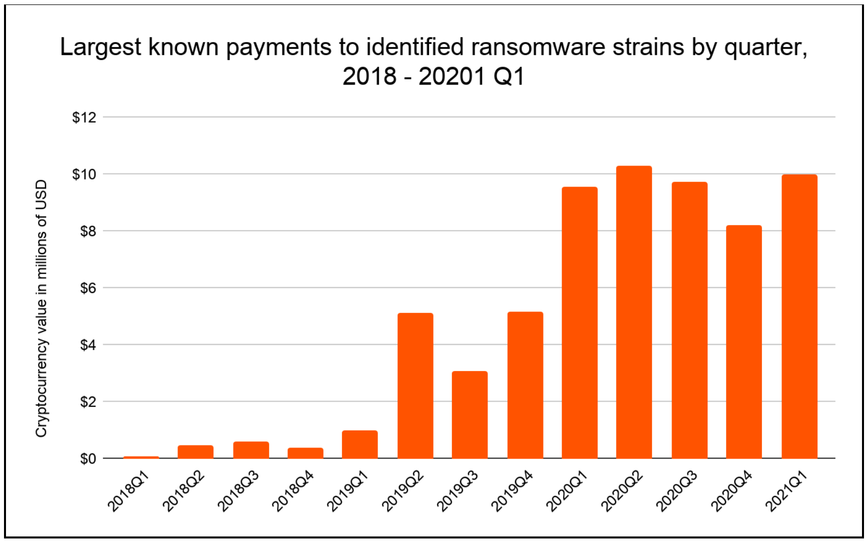
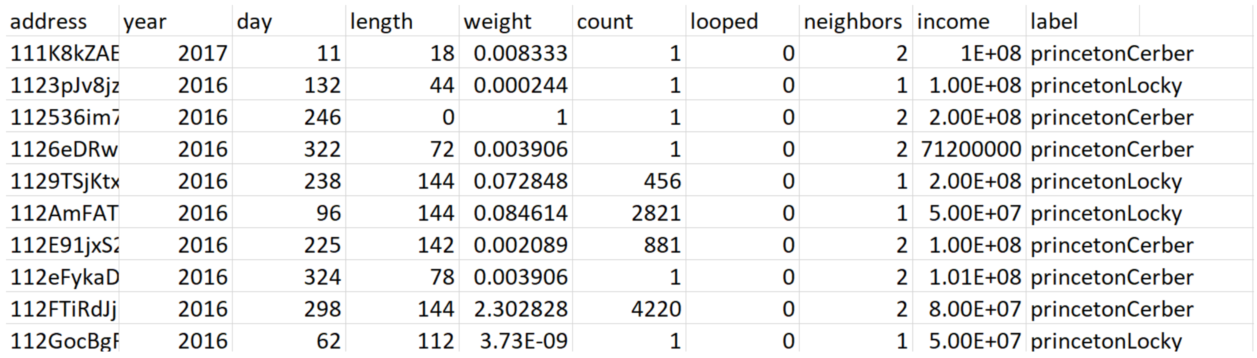


Figure 1: Largest Known Ransomware Payments by Quarter
Credit: (Ransomware 2021: Critical Mid-year Update [REPORT PREVIEW], 2021)

Problem Setting/Dataset Information

The data set we used for this project was the Bitcoin Heist Ransomware Address Dataset Data Set from the UCI Machine Learning Repository by Cuneyt Guncan Akcora, Yulia Gel, and Murat Lantarcioglu(Figure 2). The datasets contain transactions from the Bitcoin ledger, with some known ransomware payments labelled.

The authors downloaded and parsed the entire Bitcoin transaction graph from January 2009 to December 2018. Using a time interval of 24 hours, the authors extracted the daily transactions on the network and formed the Bitcoin graph and filtered out the network edges less than B0.3. Most ransomware amounts are rarely below this threshold.



address	year	day	length	weight	count	looped	neighbors	income	label
111K8kZAE	2017	11	18	0.008333	1	0	2	1E+08	princetonCerber
1123pJv8jz	2016	132	44	0.000244	1	0	1	1.00E+08	princetonLocky
112536im7	2016	246	0	1	1	0	2	2.00E+08	princetonCerber
1126eDRw	2016	322	72	0.003906	1	0	2	71200000	princetonCerber
1129TSjKtx	2016	238	144	0.072848	456	0	1	2.00E+08	princetonLocky
112AmFAT	2016	96	144	0.084614	2821	0	1	5.00E+07	princetonLocky
112E91jxS2	2016	225	142	0.002089	881	0	2	1.00E+08	princetonCerber
112eFykaD	2016	324	78	0.003906	1	0	2	1.01E+08	princetonCerber
112FTiRdJj	2016	298	144	2.302828	4220	0	2	8.00E+07	princetonCerber
112GocBgF	2016	62	112	3.73E-09	1	0	1	5.00E+07	princetonLocky

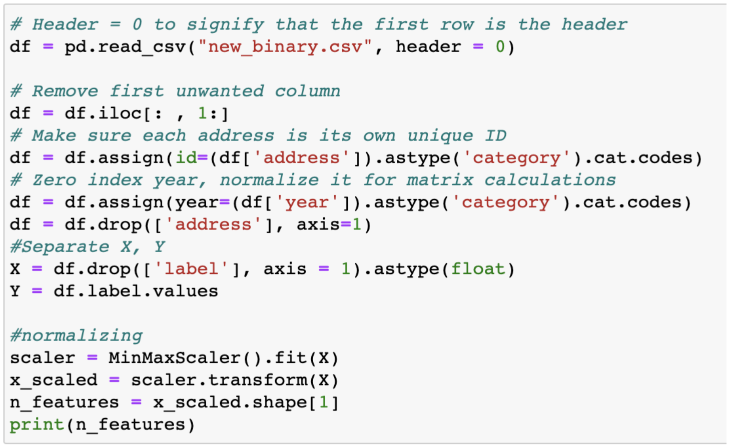
Figure 2: Dataset Data in Excel

The dataset we got had 2,916,697 instances and 10 attributes. The attributes are the bitcoin address, year, length, day, weight, count, looped, neighbors, income, and label. The graph features chosen were designed to quantify specific transaction patterns, and are as follows:

- **Looped** - intended to count how many transactions:
    - split their coins
    - move these coins in the network by using different paths
    - merge them in a single address
- **Weight** - quantifies if the transaction has more input addresses than output address(i.e. merge behavior)
    - Coins that are in multiple address are passed through a number of merging transactions, ending up in one final address
- **Count** - Also quantifies merging pattern
    - Rather than providing information on the amount of transactions that output like weight, provides information on number of of transactions
- **Length** - quantifies number of mixing rounds

    - Mixing rounds - coins are passed through newly created addresses, using the addresses as a mask while sending and receiving similar amount
- **Address** - Bitcoin address of possible ransomware
- **Year** - YYYY Format
- **Day** - Indexed 1-365
- **Income** - Satoshi amount(i.e. 1 bitcoin = 100 million satoshis)
- **Neighbors** - How many neighbors the address has in the Bitcoin network
- **Label** - Either white(clean) or name of affiliated ransomware family
    - The white labels are not known for certain if they are not ransomware
    - Ransomware comes from three studies:
        - Montreal
        - Princeton
        - Padua

There was a large amount of pre-processing done for our project. First was the dataset size. We decided that the 2 million instances were too much for us to handle, as we did not have the computational power and processing necessary to complete that task in a comfortable time. Thus, we decided to take 40,000 instances from the dataset. Of those 40,000 instances, ½ were ransomware and ½ were white. The reasoning behind this was because in our research, we found that balanced datasets led to the best neural network performances[3]. Since our intention with the project was to create a binary classifier rather than a multiclass classifier, we had to pre-process the labeling. We one hot encoded the labelling, boiling down 25 ransomware families to 1, and white to 0. Additionally, we mapped each unique bitcoin address to an unique 0-indexed ID and 0-indexed the years. Last, but not least, we normalized our data, using the MinMaxScaler from the sklearn library to fit all values into the range of 0 and 1 and prevent any bias in our model fitting[4](Figure 3).

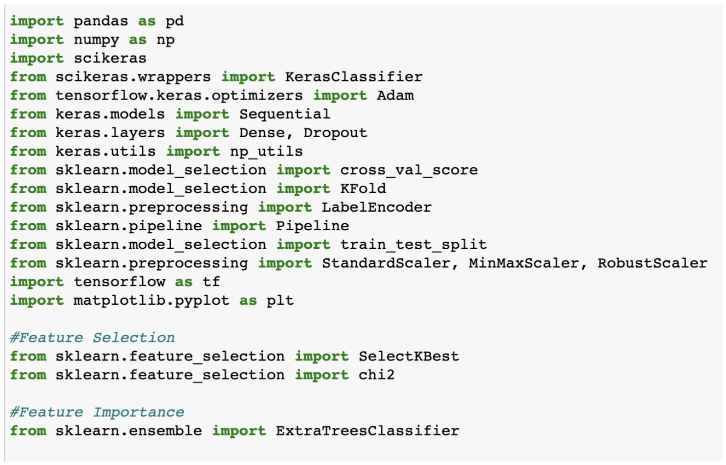


```
# Header = 0 to signify that the first row is the header
df = pd.read_csv("new_binary.csv", header = 0)

# Remove first unwanted column
df = df.iloc[: , 1:]
# Make sure each address is its own unique ID
df = df.assign(id=(df['address']).astype('category').cat.codes)
# Zero index year, normalize it for matrix calculations
df = df.assign(year=(df['year']).astype('category').cat.codes)
df = df.drop(['address'], axis=1)
#Separate X, Y
X = df.drop(['label'], axis = 1).astype(float)
Y = df.label.values

#normalizing
scaler = MinMaxScaler().fit(X)
x_scaled = scaler.transform(X)
n_features = x_scaled.shape[1]
print(n_features)
```

Figure 3: Data Preprocessing

Algorithm/Implementation

For our project environment, we decided to design our project with Python in Jupyter Notebook. We imported the reduced dataset and then pre-processed as listed in the earlier section. We used the pandas, scikeras, keras, and sklearn libraries. Pandas is used to read the data, and scikeras library allows us to seamlessly run our model in TensorFlow. Keras allows us to add some more functionality and flexibility in our model design, and the sklearn library helps with computing the metrics we will use to judge our progress. Our imports are below(Figure 4):



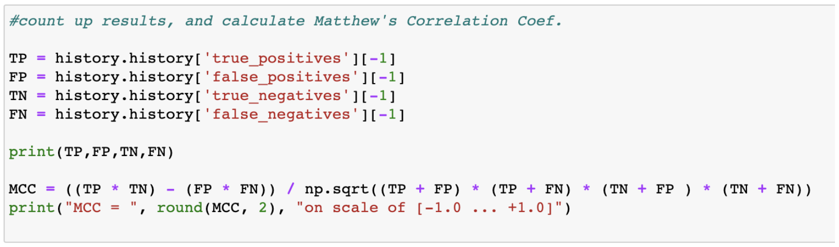
```
import pandas as pd
import numpy as np
import scikeras
from scikeras.wrappers import KerasClassifier
from tensorflow.keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.utils import np_utils
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
import tensorflow as tf
import matplotlib.pyplot as plt

#Feature Selection
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

#Feature Importance
from sklearn.ensemble import ExtraTreesClassifier
```

Figure 4: Main Project Imports

In order for us to track our progress in the project, it was necessary to decide on the metrics that would help us gauge our direction as well as our model's efficacy. The metrics we decided on are:

- **Accuracy** - The most important, it is the proportion of true results among the total number of cases. In other words, how many the model correctly predicts.
- **Precision** - Tracks out of how many predicted as "positive" by our model truly are positive. In our case, how many times our model wrongly identifies a bitcoin transaction as ransomware.
- **Recall** - Tracks out of the known positives, how many the model correctly labelled as positive. In our case, how many of the ransomware transactions we successfully caught.
- **AUC** - Stands for Area Under the ROC Curve, it provides an aggregate measure of performance across numerous classification thresholds. From a range of 0 to 1, 0 being 100% wrong and 1 being 100% right.
- **MCC** - Stands for Matthews Correlation Coefficient. Produces a high score if prediction obtains good results across all four confusion matrix categories. From a range of -1 to 1, -1 being total disagreement between prediction and observation, 0 no better than random, and 1 meaning perfect agreement.

The first four metrics we were able to gather from the metrics built into the Keras API. MCC we manually computed, as shown in Figure 5.



```
#count up results, and calculate Matthew's Correlation Coef.

TP = history.history['true_positives'][-1]
FP = history.history['false_positives'][-1]
TN = history.history['true_negatives'][-1]
FN = history.history['false_negatives'][-1]

print(TP,FP,TN,FN)

MCC = ((TP * TN) - (FP * FN)) / np.sqrt((TP + FP) * (TP + FN) * (TN + FP ) * (TN + FN))
print("MCC = ", round(MCC, 2), "on scale of [-1.0 ... +1.0]")
```

Figure 5: Matthews Correlation Coefficient Math

To generate the models, we decided to use the Keras, an excellent Deep Learning API that wraps TensorFlow for Python. For the model type, we chose sequential over functional, as it was the simpler one to model. Although functional is more complex, the sequential model was far easier to configure and more aligned with our lectures in class. For the hidden layers we used ReLU and the sigmoid function for the final layer as the activation functions as discussed in class. ReLU because it is far more computationally efficient, while sigmoid is far more helpful for final results. As for the loss function, binary cross entropy was chosen because it is conceptually equivalent to model-fitting using maximum-likelihood estimation. Thus, it is best for minimizing the differences between the data's distribution and the model's distribution. This will prevent our model from being more generalizable, yet the loss function's logarithmic properties will help avoid gradient saturation. As for the optimization algorithm, we chose Adam because of its prevalent industry usage for deep learning.

Experiment Design and Delivery

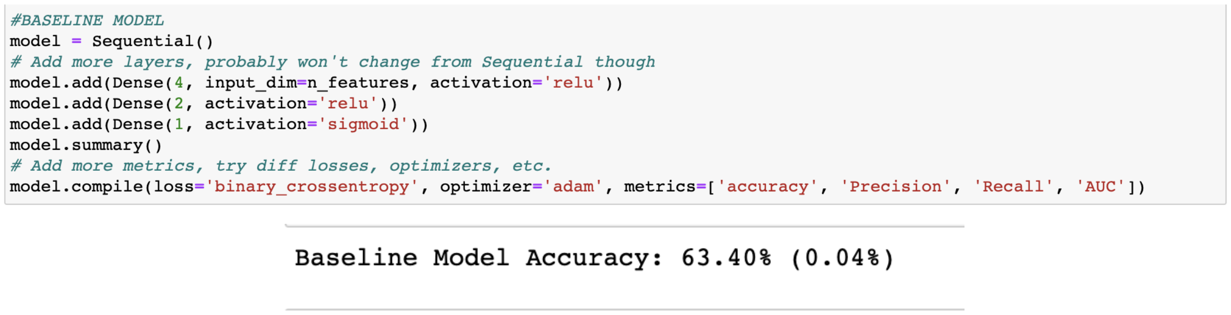
We identified 3 questions to guide our design:

1. **Can a neural net reliably identify ransomware transactions given a set of metadata from Bitcoin transactions?**
2. **Which metric is the model performing worst in? Why is that metric the lowest, and how can that be fixed in next steps?**
3. **Which features are most directly tied to identifying whether the transaction was affiliated with ransomware?**

BASELINE MODEL

For our first model, we attempted a cookie-cutter approach we devised from reviewing multiple introductory articles online. From our basic research, we found that finding the right number of layers and neurons in those layers would be a case of trial and error, but rule of thumb was that two hidden layers would suffice for most datasets. For the hidden layers we used ReLU and the sigmoid function for the final layer as the activation functions as discussed in class. ReLU allows for faster training in multi-layer models. Sigmoid is used as the final activation to give us a 0 or 1 answer to our problem.

We started with a basic sequential model that had 3 layers, 2 hidden, consisting of 4, 2, and 1 neurons respectively. They were selected because the neurons in layers are supposed to go down each layer, and that since the dataset had 9 features, the n/2 logic would be a good starting point. Our model had a modest 63.4% accuracy, a strong start with lots of space for optimization(Figure 6,7).



```
#BASELINE MODEL
model = Sequential()
# Add more layers, probably won't change from Sequential though
model.add(Dense(4, input_dim=n_features, activation='relu'))
model.add(Dense(2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
# Add more metrics, try diff losses, optimizers, etc.
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy', 'Precision', 'Recall', 'AUC'])
```

```
Baseline Model Accuracy: 63.40% (0.04%)
```

Figure 6: Baseline Code & Model Results

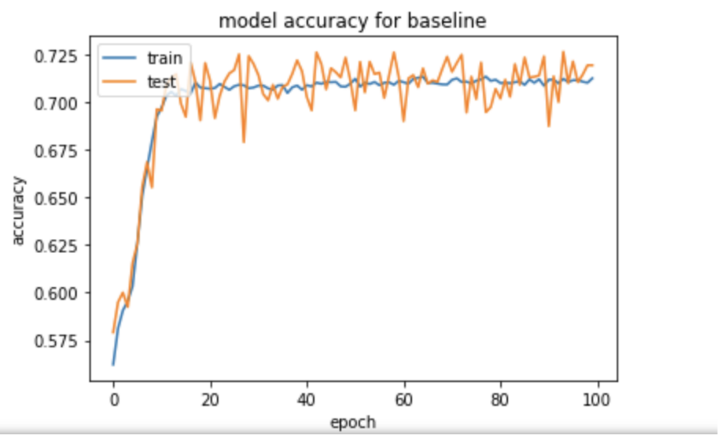
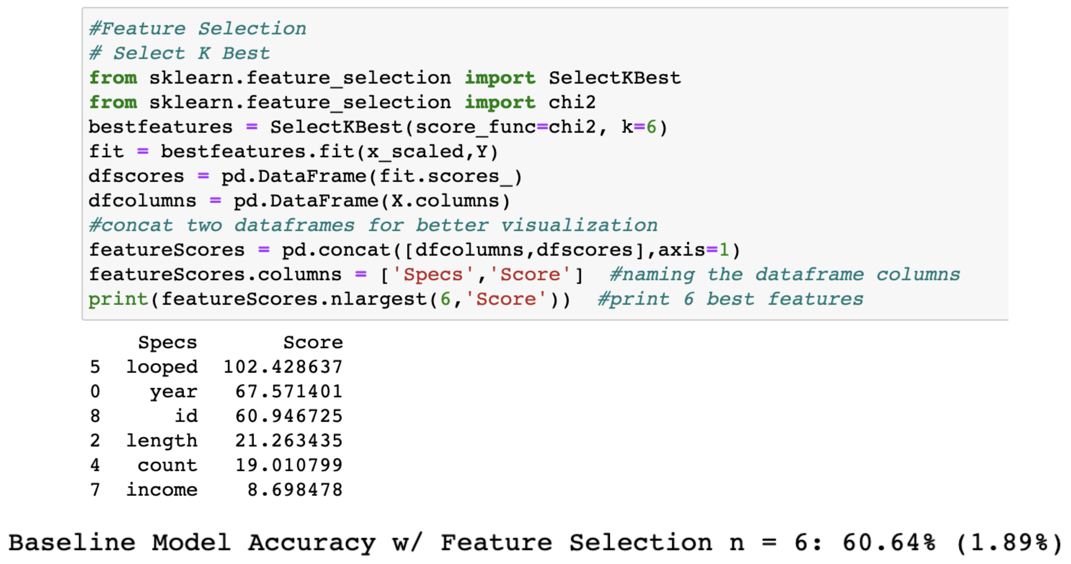


Figure 7: Graph of Baseline Model Accuracy

APPLYING FEATURE SELECTION

We did not implement any dimensionality reduction yet. We were passing all nine parameters into the model, which was computationally expensive and possibly redundant. We also intended to reduce the degree of freedom in our measurements and predictions. First, we tried feature selection using the sklearn SelectKBest algorithm, using the chi squared score function. We selected the chi squared function because it is commonly used for classification tasks, and because it tests for goodness of fit of the observed distribution to the theoretical one. We tried our baseline model with different amounts of the K best taken, with varying success(Figure 8).



```
#Feature Selection
# Select K Best
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
bestfeatures = SelectKBest(score_func=chi2, k=6)
fit = bestfeatures.fit(x_scaled,Y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']  #naming the dataframe columns
print(featureScores.nlargest(6,'Score'))  #print 6 best features
```

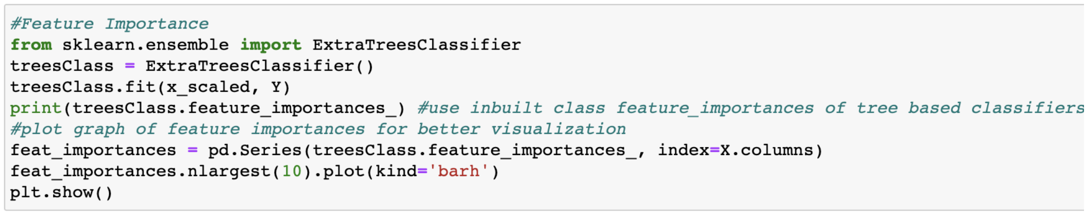
```
    Specs       Score
5  looped  102.428637
0    year   67.571401
8      id   60.946725
2  length   21.263435
4   count   19.010799
7  income    8.698478
```

```
Baseline Model Accuracy w/ Feature Selection n = 6: 60.64% (1.89%)
```

Figure 8: Feature Selection Code & Model Results

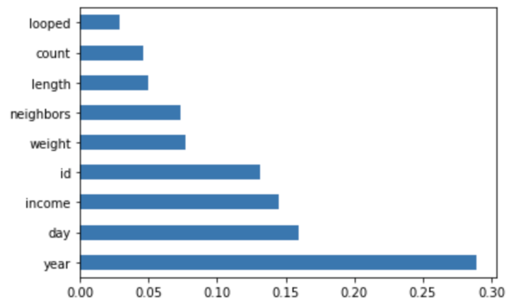
APPLYING FEATURE IMPORTANCE

In our runs, it seemed that n=6 resulted in the best runs with the baseline model. We wanted to see another dimensionality reduction technique applied to the dataset as well to compare, so we used decision trees to find the features with the most importance. To do so, we imported the ExtraTreesClassifier from sklearn, and the reason we chose ExtraTrees is because it builds multiple trees using random feature subsets, samples without replacement, and are split randomly, not on best splits. This increases the randomness of the classifier, giving the best result for understanding how valuable each input feature is to the target variable. With the results, we culled the dataset to different n most important features, and eventually settled on n = 6, which gave us a 68.34% accuracy(Figure 9).



```
#Feature Importance
from sklearn.ensemble import ExtraTreesClassifier
treesClass = ExtraTreesClassifier()
treesClass.fit(x_scaled, Y)
print(treesClass.feature_importances_) #use inbuilt class feature_importances of tree based classifiers
#plot graph of feature importances for better visualization
feat_importances = pd.Series(treesClass.feature_importances_, index=X.columns)
feat_importances.nlargest(10).plot(kind='barh')
plt.show()
```

```
[0.28887412 0.15968086 0.04986905 0.07680054 0.04617727 0.02935006
 0.07305028 0.14486663 0.13133119]
```



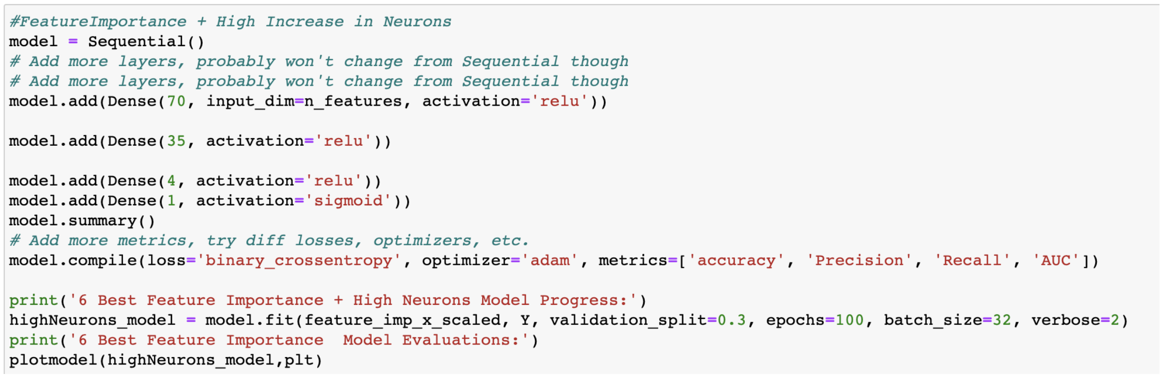
```
Baseline Model Accuracy w/ Feature Importance n = 6: 68.36% (2.34%)
```

Figure 9: Feature Importance Model Code & Metrics

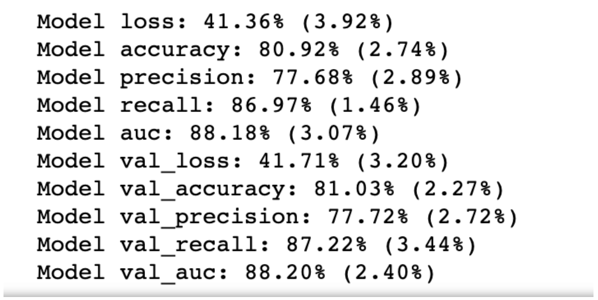
When including this into our Baseline model, we achieved markedly higher accuracy. Most importantly it was more accurate than feature selection, so we decided to continue using feature importance with our next models.

FEATURE IMPORTANCE W/ HIGH NEURON COUNT

We then began revisiting the question of hidden layers and nodes[5]. Broadly speaking, we knew adding more neurons would make our model far more specific and fine tuned to the data, and we also knew that the downside would possibly be extreme overfitting. We also found similar research in ransomware identification in Bitcoin networks, and in their system they in part had a classification model with 50 neurons in their first hidden layer and 25 in their second[]. Thus, we decided to take the same approach. Additionally, we thought since the original data had 25 different ransomware from 4 broad families, we should add another hidden layer. The first hidden layer would have a large number of neurons for pattern tracking, and would decrease to numbers such as 25 & 4. We arbitrarily chose 70 & 35, and achieved significantly greater results. The original metric is for training data, and the val_metric is for testing data. We achieved 80.92% accuracy on the training data and 81.03% on the test(Figure 10). Our precision and recall rates stayed fairly similar between train and test, and the 1+% standard deviations made it clear that the model was overfitting(Figure 11).



```
#FeatureImportance + High Increase in Neurons
model = Sequential()
# Add more layers, probably won't change from Sequential though
# Add more layers, probably won't change from Sequential though
model.add(Dense(70, input_dim=n_features, activation='relu'))

model.add(Dense(35, activation='relu'))

model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
# Add more metrics, try diff losses, optimizers, etc.
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy', 'Precision', 'Recall', 'AUC'])

print('6 Best Feature Importance + High Neurons Model Progress:')
highNeurons_model = model.fit(feature_imp_x_scaled, Y, validation_split=0.3, epochs=100, batch_size=32, verbose=2)
print('6 Best Feature Importance  Model Evaluations:')
plotmodel(highNeurons_model,plt)
```



```
Model loss: 41.36% (3.92%)
Model accuracy: 80.92% (2.74%)
Model precision: 77.68% (2.89%)
Model recall: 86.97% (1.46%)
Model auc: 88.18% (3.07%)
Model val_loss: 41.71% (3.20%)
Model val_accuracy: 81.03% (2.27%)
Model val_precision: 77.72% (2.72%)
Model val_recall: 87.22% (3.44%)
Model val_auc: 88.20% (2.40%)
```

Figure 10: Feature Importance Code & Model with Higher Neuron Count

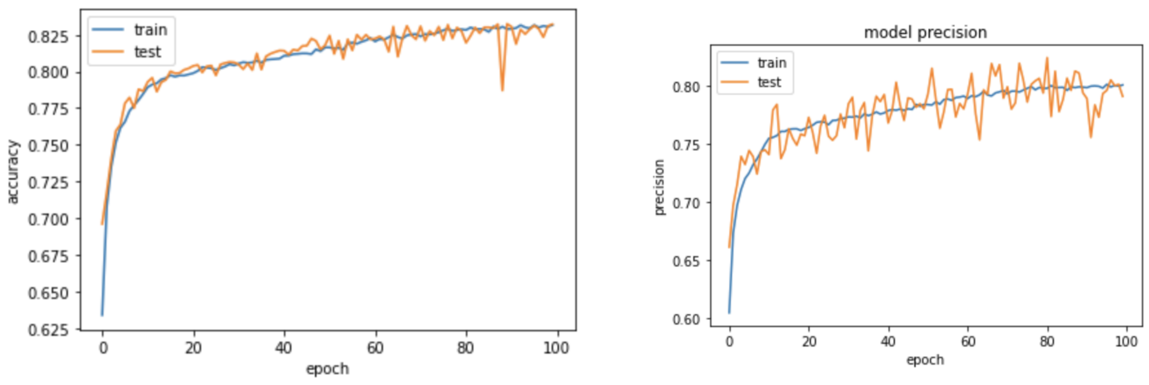
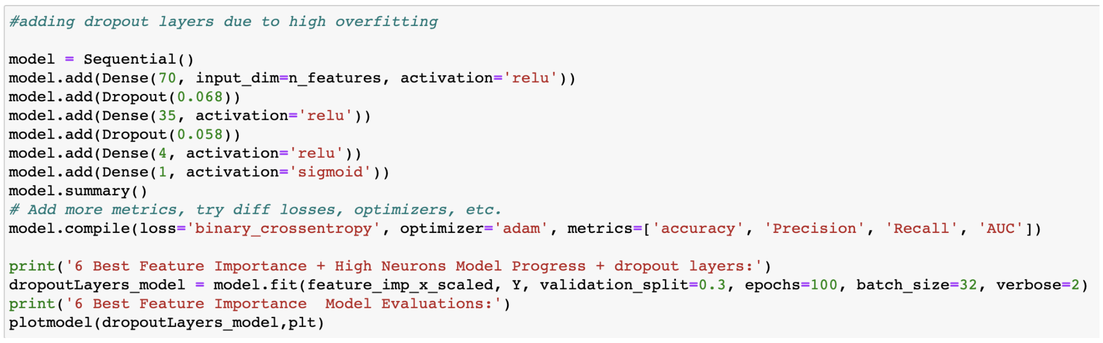


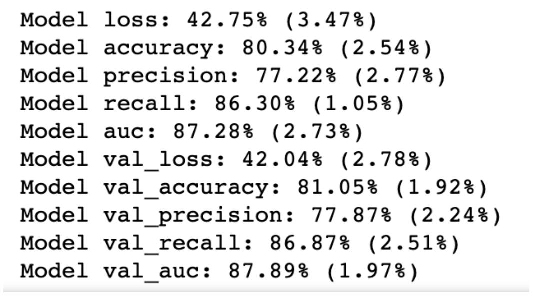
Figure 11: Accuracy and Precision Graphs for High Neuron Count

FEATURE IMPORTANCE W/ HIGH NEURON COUNT & DROPOUT LAYERS

Although our accuracy had sky-rocketed, the vast increase of neurons did create the side-effect of overfitting as we anticipated. Thus, we added dropout layers. This regularization technique is generally used to combat overfitting, as it randomly drops some neurons from the network to prevent network co-adaption among similar layer nodes. Sometimes, network layers will also react jointly to previous layers' mistakes, so the process of randomly dropping neurons makes the model far more robust. We added a dropout layer between each hidden layer, arbitrarily choosing the number from a range we found generally accepted. Our results did not improve much on the accuracy front or any other metric, with around the same. The standard deviation in the results, however, did decrease(Figure 12).



```
#adding dropout layers due to high overfitting

model = Sequential()
model.add(Dense(70, input_dim=n_features, activation='relu'))
model.add(Dropout(0.068))
model.add(Dense(35, activation='relu'))
model.add(Dropout(0.058))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
# Add more metrics, try diff losses, optimizers, etc.
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy', 'Precision', 'Recall', 'AUC'])

print('6 Best Feature Importance + High Neurons Model Progress + dropout layers:')
dropoutLayers_model = model.fit(feature_imp_x_scaled, Y, validation_split=0.3, epochs=100, batch_size=32, verbose=2)
print('6 Best Feature Importance  Model Evaluations:')
plotmodel(dropoutLayers_model,plt)
```



```
Model loss: 42.75% (3.47%)
Model accuracy: 80.34% (2.54%)
Model precision: 77.22% (2.77%)
Model recall: 86.30% (1.05%)
Model auc: 87.28% (2.73%)
Model val_loss: 42.04% (2.78%)
Model val_accuracy: 81.05% (1.92%)
Model val_precision: 77.87% (2.24%)
Model val_recall: 86.87% (2.51%)
Model val_auc: 87.89% (1.97%)
```

Figure 12: Adding Dropout Layers to High Neuron Code & Model Results

We noticed that our precision rate was consistently staying at rates lower than the other metrics, which meant that our model had a tendency to identify a transaction as ransomware more than it was. Although the test precision levels fluctuate around the train precision levels, this precision issue seems to arise from overfitting and the usage of a balanced dataset(Figure 14). We also theorized that because we were using Dense layers in which neurons are all connected to each other in the layers, which perhaps wasn't the most ideal modeling because some of these ransomware were part of unique families, so the dense connections were overfitting.

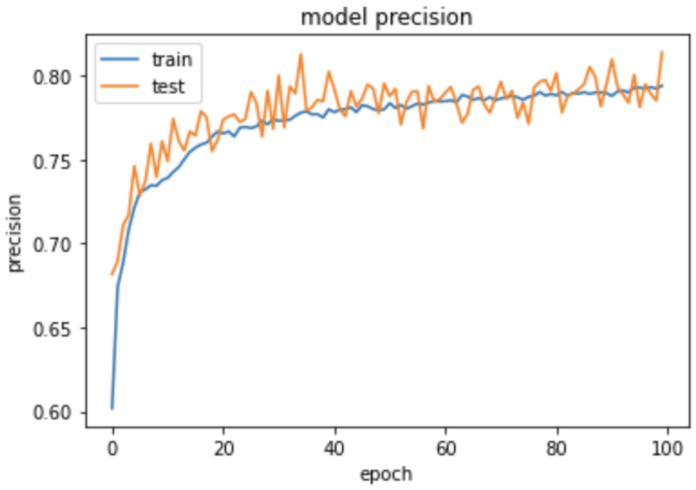
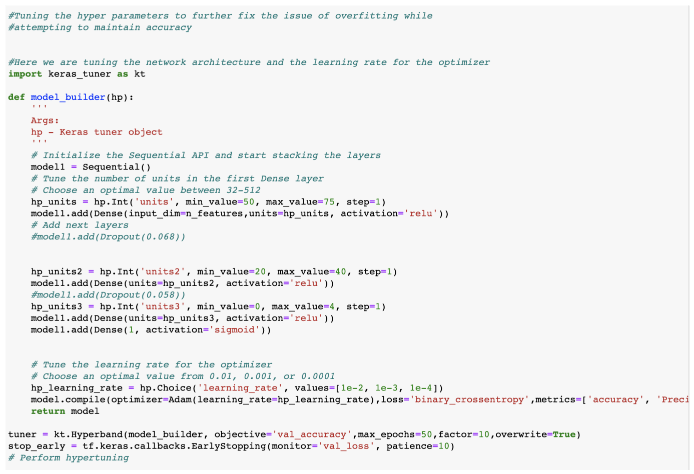


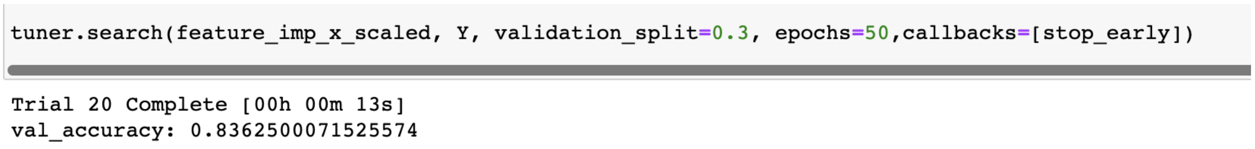
Figure 13: Adding Dropout Layers to High Neuron Model Precision Graph

FEATURE IMPORTANCE W/ HIGH NEURON COUNT, DROPOUT LAYERS, & HYPERPARAMETER TUNING

Given that configuring the number of neurons and hidden layers led to the greatest improvement in our model, I thought we should try to find the best number of neurons and layers for our model. To systematically figure this out, we turned them into hyper parameters in our model. We made use of the keras_tuner library to do this. In our model below we set the number of neurons in each layer as the hyper parameter, the tuner then tries to find the best number of neurons and whether or not a third layer is necessary that yields the best accuracy(Figure 14).



```
#Tuning the hyper parameters to further fix the issue of overfitting while
#attempting to maintain accuracy


#Here we are tuning the network architecture and the learning rate for the optimizer
import keras_tuner as kt

def model_builder(hp):
    '''
    Args:
    hp - Keras tuner object
    '''
    # Initialize the Sequential API and start stacking the layers
    model1 = Sequential()
    # Tune the number of units in the first Dense layer
    # Choose an optimal value between 32-512
    hp_units = hp.Int('units', min_value=50, max_value=75, step=1)
    model1.add(Dense(input_dim=n_features,units=hp_units, activation='relu'))
    # Add next layers
    #model1.add(Dropout(0.068))


    hp_units2 = hp.Int('units2', min_value=20, max_value=40, step=1)
    model1.add(Dense(units=hp_units2, activation='relu'))
    #model1.add(Dropout(0.058))
    hp_units3 = hp.Int('units3', min_value=0, max_value=4, step=1)
    model1.add(Dense(units=hp_units3, activation='relu'))
    model1.add(Dense(1, activation='sigmoid'))


    # Tune the learning rate for the optimizer
    # Choose an optimal value from 0.01, 0.001, or 0.0001
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
    model.compile(optimizer=Adam(learning_rate=hp_learning_rate),loss='binary_crossentropy',metrics=['accuracy', 'Preci
    return model

tuner = kt.Hyperband(model_builder, objective='val_accuracy',max_epochs=50,factor=10,overwrite=True)
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)
# Perform hypertuning
```

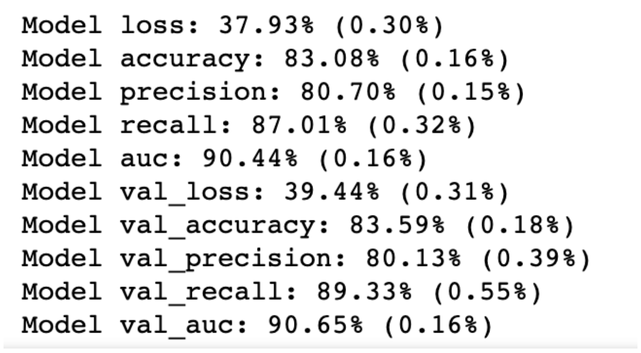


```
tuner.search(feature_imp_x_scaled, Y, validation_split=0.3, epochs=50,callbacks=[stop_early])
```

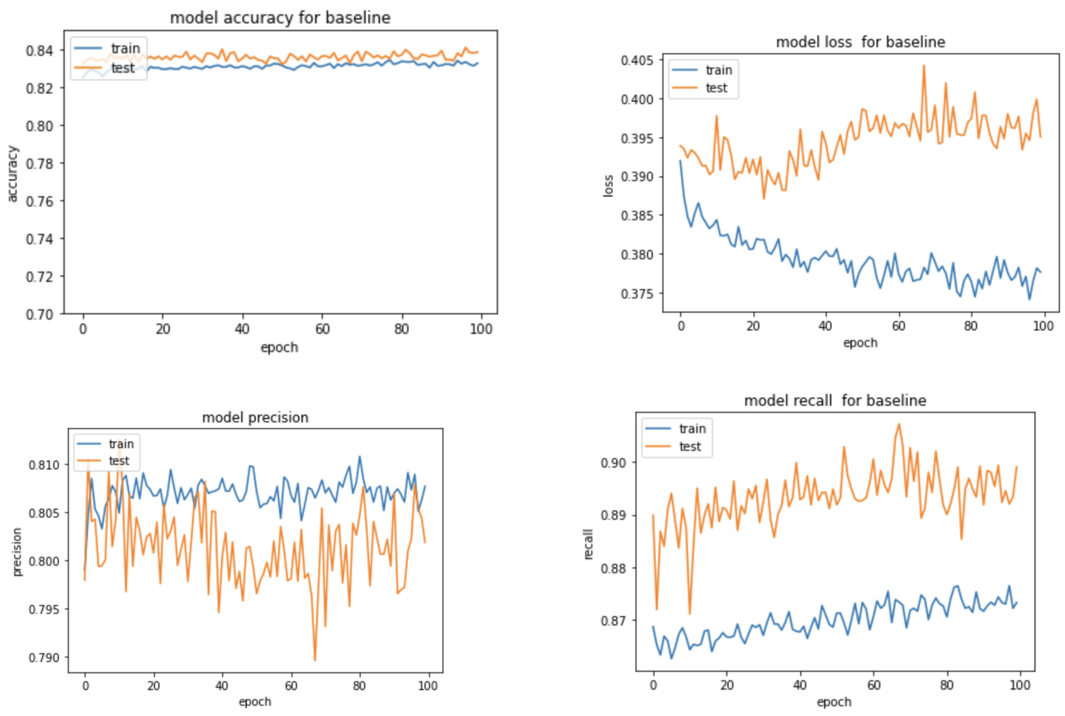
```
Trial 20 Complete [00h 00m 13s]
val_accuracy: 0.8362500071525574
```

Figure 14: Adding Hyperparameter Tuning Code & Model Accuracy

This led to our best accuracy result, a result of 83.6%. Our precision finally reached 80+% levels, a huge reason for our increase in accuracy. Our AUC metric surpassed 90% for the first time, and our standard deviations for all recorded metrics was sub-1%(Figure 15).



```
Model loss: 37.93% (0.30%)
Model accuracy: 83.08% (0.16%)
Model precision: 80.70% (0.15%)
Model recall: 87.01% (0.32%)
Model auc: 90.44% (0.16%)
Model val_loss: 39.44% (0.31%)
Model val_accuracy: 83.59% (0.18%)
Model val_precision: 80.13% (0.39%)
Model val_recall: 89.33% (0.55%)
Model val_auc: 90.65% (0.16%)
```



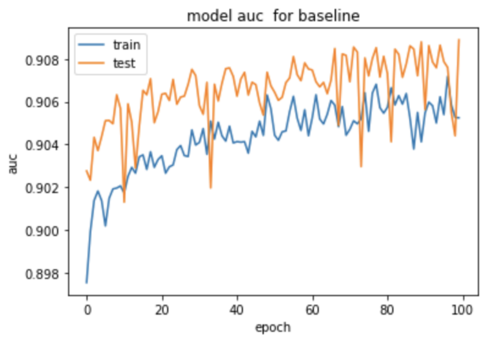


Figure 9: Metrics of Final Model Design

Thus, we were able to answer our three experiment questions:

1. **Can a neural net reliably identify ransomware transactions given a set of metadata from Bitcoin transactions?**

From our project, the answer seems to be yes. With our two weeks of work, we were able to create a binary classification classifier that was able to predict w/ a 84% accuracy rate. This is also taking into account that we modified our dataset from a multiple classification problem to a binary classification problem, which makes us believe if we continued the work as a multiple classification problem, the accuracy would increase due to the classifier being able to find strong patterns within certain ransomware families. With access the more data, the sky's the limit.

2. **Which metric is the model performing worst in? Why is that metric the lowest, and how can that be fixed in next steps?**

The model performed worst in precision, which measures how precise the model is in its flagging, or its false positive rate. This means our models had a tendency to think a transaction was ransomware when it wasn't. This could partly be due to the dataset, which is 50/50 in terms of ransomware and clean, which doesn't correctly reflect reality. But this also may be due to a lack of dropout magnitude. This can be fixed by further hypertuning our model parameters, culling outlier values, adding more data, or merging feature engineering and selection.

3. **Which features are most directly tied to identifying whether the transaction was affiliated with ransomware?**

It's interesting because the two algorithms we used to identify the features of most correlation/causation pointed to fairly different results. For SelectKBest that used the chi squared loss function, the top five were looped, year, id, length and count. For our ExtraTreesClassifier the top five were year, day, income, id and weight. Because our model tested better with the latter, we used that. However, we should look at using other methods of feature selection and engineering, as it wouldn't be wise to use year or day for future predictions for example. There seems to be consensus on year and id only.

Conclusion

In this project we set out to design a neural model that was able to identify possible ransomware payments on the Bitcoin blockchain. The process was an eye-opening experience into the world of deep learning, and a fascinating foray into the ransomware and blockchain world. Through our usage of Keras API on Python, we constantly iterated our model creation, molding it with data pre-processing, feature importance, dropout layers, and hyper parameterization. Although it had its kinks, we were able to reliably predict 83% of the inputs. Given the limited computing power, time, and data, this is a reasonable rate. With a larger network and more inputs we might have been able to get a much higher accuracy. The tuning of the hyperparameters made an enormous difference in the performance of our model. A tuner function helped give us a higher accuracy. If we explored larger networks with more complicated features we might also improve our accuracy. We could train these over many epochs with more computing power. For next steps, it would be interesting to gather more data, testing different classifiers and different hyperparameter tunings for the optimal design. It would be especially interesting to cross apply this neural network application with another deep learning application, such as a decision tree or a general adversarial network[6]. With the rise of blockchain and cryptocurrency, the problem our project aims to solve will only become more pressing.

Works Cited

1. Blog.chainalysis.com. 2021. *Ransomware 2021: Critical Mid-year Update [REPORT PREVIEW]*. [online] Available at: <https://blog.chainalysis.com/reports/ransomware-update-may-2021> [Accessed 8 December 2021].
2. Dossett, J., 2021. *A timeline of the biggest ransomware attacks*. [online] CNET. Available at: <https://www.cnet.com/personal-finance/crypto/a-timeline-of-the-biggest-ransomware-attacks/> [Accessed 8 December 2021].
3. Amruthnath, N. (2020, June 25). Why balancing your data set is important? R-Bloggers. https://www.r-bloggers.com/2020/06/why-balancing-your-data-set-is-important/
4. Hale, J. (2021, September 1). Scale, standardize, or normalize with Scikit-Learn. Medium. Retrieved December 8, 2021, from https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02.
5. Heaton. (2020, July 20). The Number of Hidden Layers. Heaton Research. https://www.heatonresearch.com/2017/06/01/hidden-layers.html
6. Al-Haija, Q. A., & Alsulami, A. A. (2021). High Performance Classification Model to Identify Ransomware Payments for Heterogeneous Bitcoin Networks. Electronics, 10(17), 2113. https://doi.org/10.3390/electronics10172113