DTSA-5511: Week 6 - Final Project

October 6, 2022

Credit Card Fraud Detection using Neural Networks

Introduction

This is the Week 6 Final Project assignment for DTSA-5511, Introduction to Deep Learning. This applies an autoencoder neural network to a dataset of credit card transactions to determine if the transaction is fraudulent or not. The credit card transaction dataset was found on Kaggle and consists of 284,807 records, however only 492 of them are marked as fraudulent and all the remaining are marked valid.

Autoencoders are a special class of neural networks that consist of an encoder section and a decoder section. The encoder section essentially down samples the supplied data into a smaller number of dimensions (or features) and the decoder expands that back to the original dimension size. In this manner the autoencoder learns how to reduce the the number of dimensions necessary for modeling the data. This is a similar function to what Principal Component Analysis (PCA) does, but PCA works best when there are linear relationships between the dimensions and the target output (valid or fraud) while autoencoders are not subect to this restriction.

This distinction is important with the dataset used in this project as there are 28 columns in the dataset which are simply named V1 - V28 with no explanation as to what the values in these columns mean or represent. By using an autoencoder we should be able to achieve good results without needing to determine if linear relationships are present.

Finally, this Jupyter notebook can be downloaded from https://github.com/clayv/DTSA-5511_Final.

In [1]:
```
from imblearn.over_sampling import SMOTE, BorderlineSMOTE
import keras
from keras.callbacks import EarlyStopping, ReduceLROnPlateau
from keras.layers import Activation, Dense, Dropout
from keras.layers import Flatten, MaxPooling2D, Rescaling
from keras.models import Sequential
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
#import random
import seaborn as sns
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, f1_score, roc_curve, roc_auc_score
from sklearn.model_selection import train_test_split
import tensorflow as tf
```

In [2]:
```
print("Keras version:",keras.__version__)
print("TensorFlow version:",tf.__version__)
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
    print('Not connected to a GPU')
else:
    #print("\n", gpu_info)
    print("Connected to a GPU")
```

```
Keras version: 2.10.0
TensorFlow version: 2.10.0
Connected to a GPU
```

In [3]:
```
#Gloabals
DATA_DIR = "./data"
LABELS = ["Valid", "Fraud"]
ENCODER_ACTIVATION = "selu"
RANDOM_SEED = 42
METRICS = [
      keras.metrics.TruePositives(name='tp'),
      keras.metrics.FalsePositives(name='fp'),
      keras.metrics.TrueNegatives(name='tn'),
      keras.metrics.FalseNegatives(name='fn'),
      keras.metrics.BinaryAccuracy(name='accuracy'),
      keras.metrics.Precision(name='precision'),
      keras.metrics.Recall(name='recall'),
      keras.metrics.AUC(name='auc'),
]
```

The Dataset[1]

The dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, with only 492 fraudulent transactions out of 284,807 transactions.

It contains only numerical input variables which are the result of a PCA transformation. Due to confidentiality issues, the original features cannot be provided and more background information about the data is not available. Features V1, V2, ..., V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

References

[1]Kaggle. Credit Card Fraud Detection. https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

In [30]:
```
df_cc_all = pd.read_csv(os.path.join(DATA_DIR, "creditcard.csv"))
```

Exploratory Data Analysis

After the loading the data, well display the number of records in the the dataset and use the describe function to display some basic information about each feature/column and the head(2) function to display 2 rows.

In [31]:
```
print("Records in dataset: {0}\n".format(len(df_cc_all)))
print(df_cc_all.describe())
df_cc_all.head(2)
```

```
Records in dataset: 284807

                Time            V1            V2            V3            V4  \
count  284807.000000  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean    94813.859575  3.918649e-15  5.682686e-16 -8.761736e-15  2.811118e-15
std     47488.145955  1.958696e+00  1.651309e+00  1.516255e+00  1.415869e+00
min         0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00
25%     54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01
50%     84692.000000  1.810880e-02  6.548556e-02  1.798463e-01 -1.984653e-02
75%    139320.500000  1.315642e+00  8.037239e-01  1.027196e+00  7.433413e-01
max    172792.000000  2.454930e+00  2.205773e+01  9.382558e+00  1.687534e+01

                 V5            V6            V7            V8            V9  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean  -1.552103e-15  2.040130e-15 -1.698953e-15 -1.893285e-16 -3.147640e-15
std    1.380247e+00  1.332271e+00  1.237094e+00  1.194353e+00  1.098632e+00
min   -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01
25%   -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01
50%   -5.433583e-02 -2.741871e-01  4.010308e-02  2.235804e-02 -5.142873e-02
75%    6.119264e-01  3.985649e-01  5.704361e-01  3.273459e-01  5.971390e-01
max    3.480167e+01  7.330163e+01  1.205895e+02  2.000721e+01  1.559499e+01

       ...           V21           V22           V23           V24  \
count  ...  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   ...  1.473120e-16  8.042109e-16  5.282512e-16  4.456271e-15
std    ...  7.345240e-01  7.257016e-01  6.244603e-01  6.056471e-01
min    ... -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00
25%    ... -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01
50%    ... -2.945017e-02  6.781943e-03 -1.119293e-02  4.097606e-02
75%    ...  1.863772e-01  5.285536e-01  1.476421e-01  4.395266e-01
max    ...  2.720284e+01  1.050309e+01  2.252841e+01  4.584549e+00

                V25           V26           V27           V28         Amount  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  284807.000000
mean   1.426896e-15  1.701640e-15 -3.662252e-16 -1.217809e-16      88.349619
std    5.212781e-01  4.822270e-01  4.036325e-01  3.300833e-01     250.120109
min   -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01       0.000000
25%   -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02       5.600000
50%    1.659350e-02 -5.213911e-02  1.342146e-03  1.124383e-02      22.000000
75%    3.507156e-01  2.409522e-01  9.104512e-02  7.827995e-02      77.165000
max    7.519589e+00  3.517346e+00  3.161220e+01  3.384781e+01   25691.160000

              Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000

[8 rows x 31 columns]
```

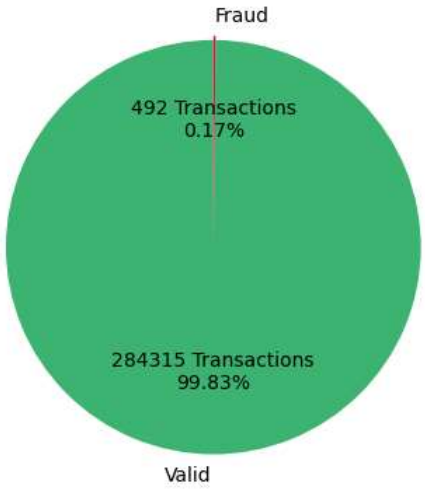
Out[31]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846

2 rows × 31 columns

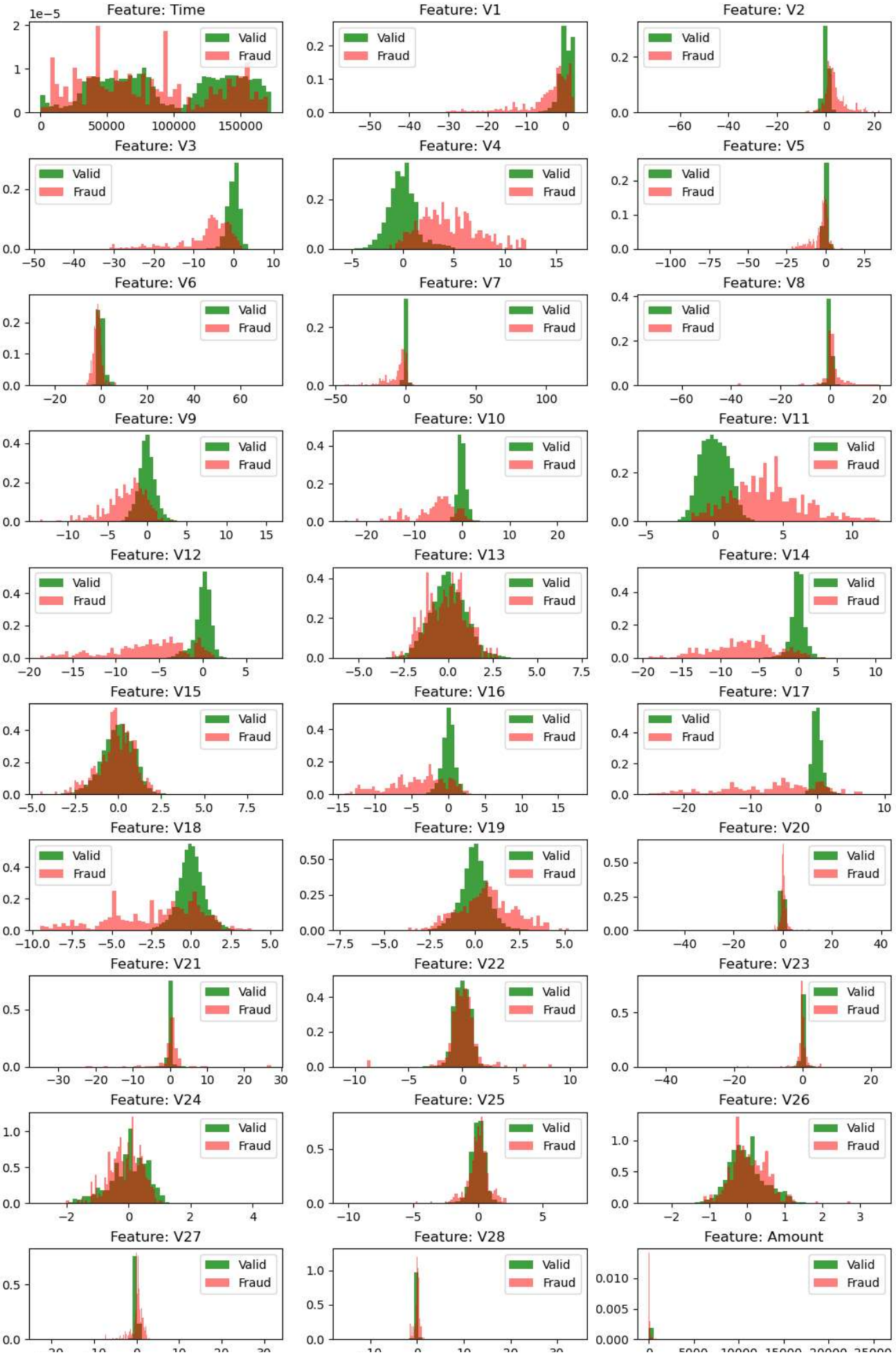
Next we'll make a pie plot to verify the classification counts as given in the data description on Kaggle. The plot confirms that of the 284, 807 records there are 492 transactions labeled as fraudulent and the remaining 284,315 as valid. This means that only ≈0.17% of the transactions are labled as fraudulent.

In [32]:
```
def makePiePlot(df):
    values = [len(df[df.Class == 0]), len(df[df.Class == 1])]
    plt.pie(values, explode = [.01, .01], labels = LABELS, colors = ['mediumseagreen', 'crimson'],
            autopct = (lambda value:f'{value * sum(values) / 100 :.0f} Transactions\n{value:.2f}%'),
            pctdistance = 0.6, shadow = False, labeldistance = 1.1, startangle = 90,
            radius = 1, counterclock = True, wedgeprops = None, textprops = None, center = (0, 0),
            frame = False, rotatelabels = False, normalize = True, data = None)
    plt.show()

makePiePlot(df_cc_all)
```



As we have no information regarding features V1 - V28, this next cell iterates over all the features, except for "Class", and displays the density histogram of both the valid and fraudulent transactions for each feature. This may be useful during the modeling portion of this assignment in shaping the data presented to the autoencoder to improve perfomance.

In [33]:
```
features = df_cc_all.columns[:-1].values

cols = 3
rows = len(features) // cols
fig, ax = plt.subplots(rows, cols, figsize=(13, 20))

for i, feature in enumerate(df_cc_all[features]):
    ax[i // cols, i % cols].hist(df_cc_all[feature][df_cc_all.Class == 0], bins=50,
            density = True, alpha = 0.75, color = 'green', label = 'Valid')
    ax[i // cols, i % cols].hist(df_cc_all[feature][df_cc_all.Class == 1], bins=50,
         density = True, alpha = 0.5, color = 'red', label = 'Fraud')
    ax[i // cols, i % cols].set_title('Feature: ' + str(feature))
    ax[i // cols, i % cols].legend()

plt.subplots_adjust(hspace = 0.5)
plt.show()
```


Feature: Time
1e−5
Feature: V1
Feature: V2
Feature: V3
Feature: V4
Feature: V5
Feature: V6
Feature: V7
Feature: V8
Feature: V9
Feature: V10
Feature: V11
Feature: V12
Feature: V13
Feature: V14
Feature: V15
Feature: V16
Feature: V17
Feature: V18
Feature: V19
Feature: V20
Feature: V21
Feature: V22
Feature: V23
Feature: V24
Feature: V25
Feature: V26
Feature: V27
Feature: V28
Feature: Amount
Valid
Fraud

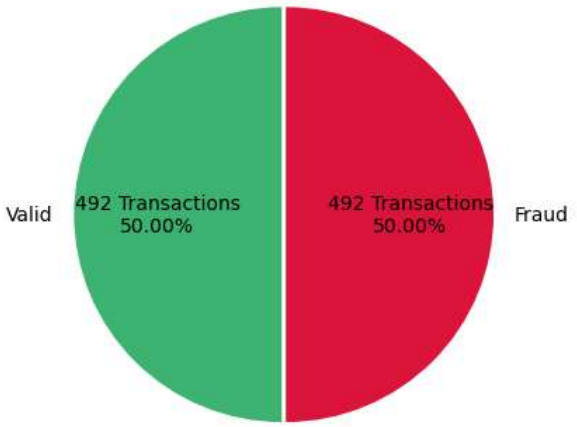
Data Wrangling

The time column will be dropped because from the data description it is merely the number of seconds passed since the first record in the dataset. Additionally, all remaining feature columns will have their values standardized as this was found to dramatically reduce the number of epochs required before the model converges.

In [34]:
```
df_cc_all = df_cc_all.drop('Time', axis=1)
#df_cc_all = df_cc_all.drop(['V13', 'V15', 'V20', 'V22', 'V23', 'V25', 'V26', 'V28'], axis=1)
#df_cc_all = df_cc_all.drop(['V2', 'V3', 'V5', 'V7', 'V9', 'V11', 'V12', 'V16', 'V17', 'V18'], axis=1)

features = df_cc_all.columns[:-1].values
numFeatures = len(features)

for feature in features:
    mean, std = df_cc_all[feature].mean(), df_cc_all[feature].std()
    df_cc_all.loc[:, feature] = (df_cc_all.loc[:, feature] - mean) / std
```

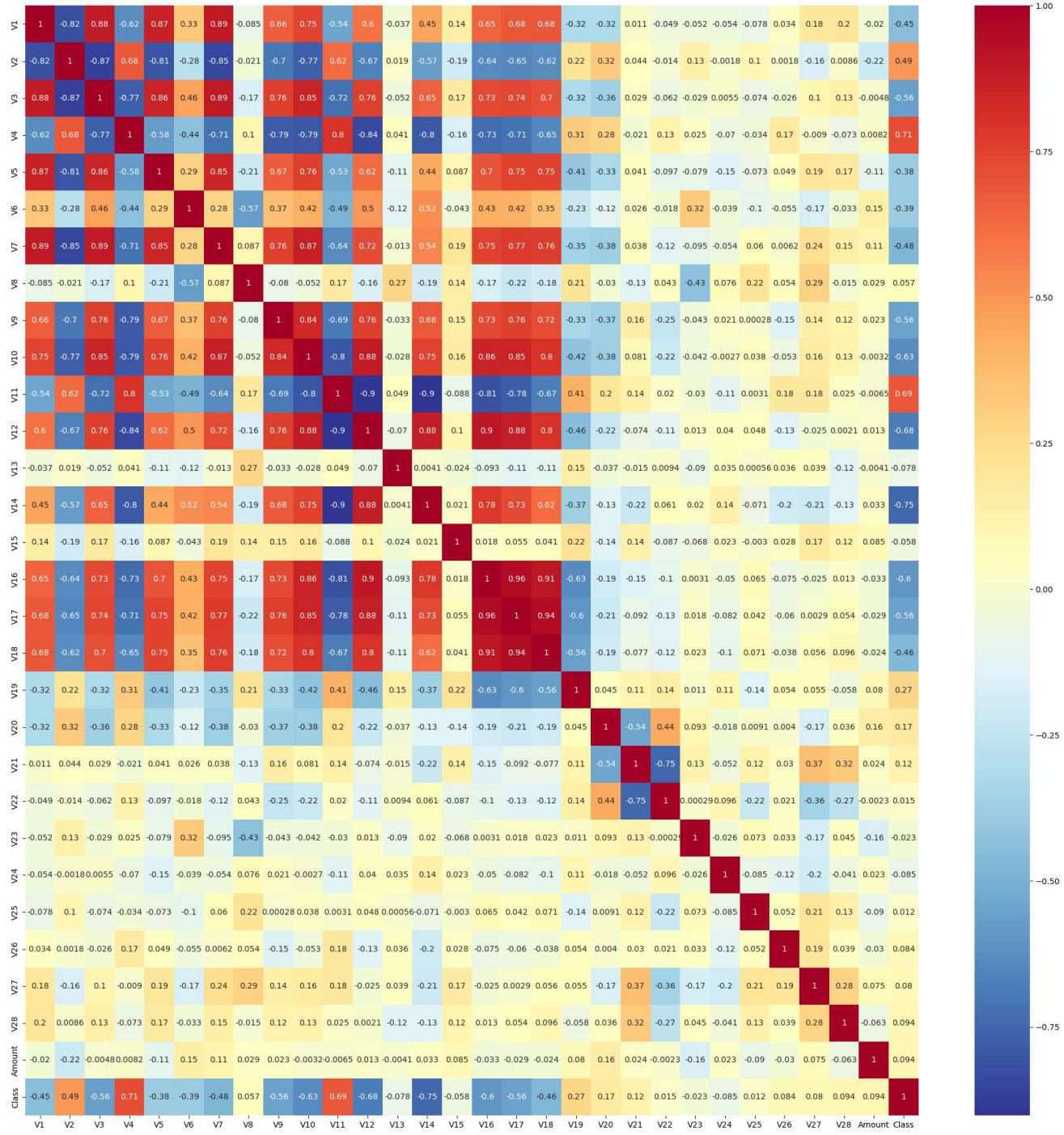
Next a count of the smaller of the two classes (fraud) is made and then a random sample is taken of the larger of the 2 classes to create a balanced dataset. Before using this new dataset, another pie plot is made to ensure it is now balanced.

In [35]:
```
def getSubset(df, count, rank):
    return df[df.Class == rank].sample(n = count, replace = False, random_state = RANDOM_SEED)

lowCount = min(len(df_cc_all[df_cc_all.Class == 0]), len(df_cc_all[df_cc_all.Class == 1]))

dfValid = getSubset(df_cc_all, lowCount, 0)
dfFraud = getSubset(df_cc_all, lowCount, 1)

df_balanced = pd.concat([dfValid, dfFraud])#.sample(frac = 1, random_state = RANDOM_SEED)

makePiePlot(df_balanced)
```



A coorelation matix is created in the next cell and is show a large number of coorelated features, specifically V1 - V18 (except for V6, V13, & v15) show darker colors. Again, although no action will be taken intitially based on this information again it may be useful during the modeling portion of this assignment in shaping the data presented to the autoencoder to improve perfomance.

In [36]:
```
plt.figure(figsize = (25, 25))
sns.heatmap(df_balanced.corr(), annot = True, cmap = "RdYlBu_r")
plt.show()
```



Modeling Setup

This next cell is a plotting helper function for use in creating visualization of the modeling process, how it progresses, and prediction results.

In [77]:

```
def plotResults(origin, history, actuals, predictions):
    dataName = "Validation Data"
    fig, ax = plt.subplots(2, 2, figsize=(13, 10))

    print("On the " + dataName)

    print("AUC: {0}".format(roc_auc_score(actuals, predictions)))

    binaryPredictions = np.where(predictions < .5, 0, 1)
    print("F1 Score: ", f1_score(actuals, binaryPredictions))

    upper = len(history['loss']) + 1

    ax[0, 0].plot(np.arange(origin + 1, upper), history['loss'][origin:], label = 'Training')
    ax[0, 0].plot(np.arange(origin + 1, upper), history['val_loss'][origin:], label = 'Validation', color = 'red')
    ax[0, 0].set(xticks=np.arange(origin, upper, upper // 10))
    ax[0, 0].set_xlabel('Epoch')
```

```
    ax[0, 0].set_ylabel('Loss')
    ax[0, 0].set_title("Loss w/" + dataName)
    ax[0, 0].legend()

    ax[0, 1].plot(np.arange(origin + 1, upper), history['accuracy'][origin:], label = 'Training')
    ax[0, 1].plot(np.arange(origin + 1, upper), history['val_accuracy'][origin:], label = 'Validation', color = 'red')
    ax[0, 1].set(xticks=np.arange(origin, upper, upper // 10))
    ax[0, 1].set_xlabel('Epoch')
    ax[0, 1].set_title("Accuracy w/" + dataName)
    ax[0, 1].legend()

    fpr, tpr, threshold = roc_curve(actuals, predictions)
    ax[1, 0].plot(fpr, tpr, label='ROC curve (area = %0.2f)' % history['val_auc'][-1])
    ax[1, 0].set_xlabel('False Positive Rate')
    ax[1, 0].set_ylabel('True Positive Rate')
    ax[1, 0].set_title(dataName + " ROC")
    ax[1, 0].legend(loc="lower right")

    cm = confusion_matrix(actuals, binaryPredictions)

    ax[1, 1].matshow(cm, cmap = plt.cm.Blues, alpha = 0.3)
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax[1, 1].text(x = j, y = i, s = cm[i, j], va='center', ha='center', size='xx-large')
    ax[1, 1].set_xlabel("Confusion Matrix on\n" + dataName + "\n(0 = Negative, 1 = Positive)", fontsize = 14)
    ax[1, 1].set_ylabel('Actuals', fontsize = 12)
    ax[1, 1].set_title('Predictions', fontsize = 12)

    plt.subplots_adjust(hspace = 0.4)
    plt.show()
```

A split of the balanced dataset is made with 20% of the dataset be held in reserve for validating the model.

In [93]:
```
y = df_balanced.Class.values
X = df_balanced.drop('Class', axis=1).values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20, random_state = RANDOM_SEED)
```

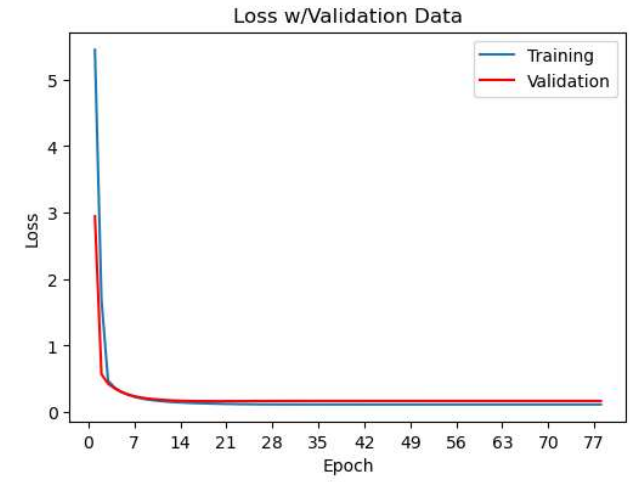
Model Architecture and Design

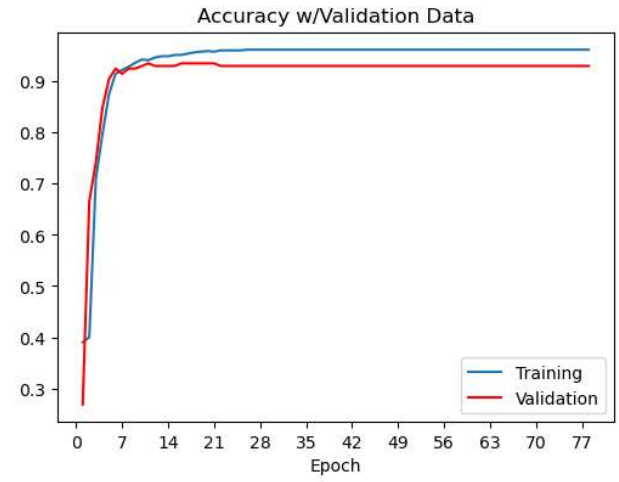
The first model will start simple with a single dense layer with 16 units in the encoder and the decoder simply expands this back to original number of features, 29, before outputting either a 0 or 1 from the final layer with the sigmoid activation function.

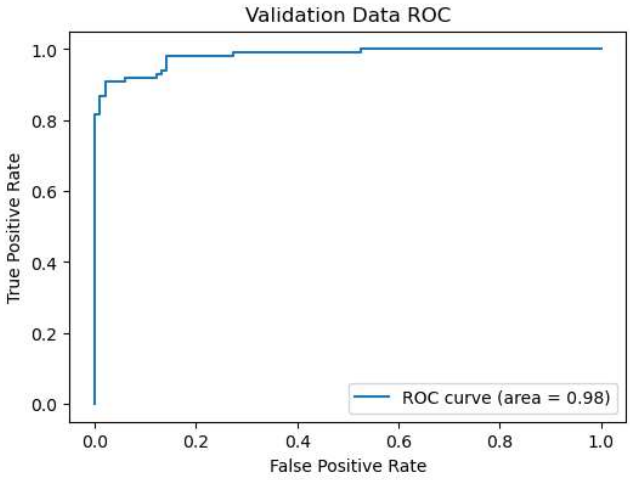
In [39]:
```
#Define learning rate scheduler and early stopping functions
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.5, patience = 3,
    verbose = False, mode = 'min', min_lr = 1e-20
)
earlyStop = EarlyStopping(monitor='loss', patience = 5)

callbacks_list = [reduce_lr, earlyStop]
```

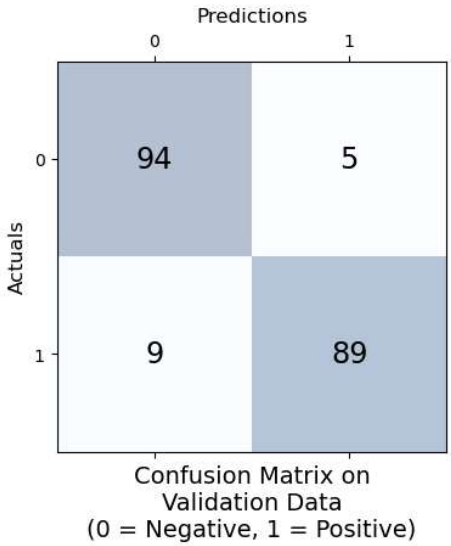
In [17]:
```
tf.keras.backend.clear_session()

stackedEncoder = keras.models.Sequential()
stackedEncoder.add(Dense(16, activation = ENCODER_ACTIVATION))

stackedDecoder = keras.models.Sequential()
stackedDecoder.add(Dense(numFeatures, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(1, activation = "sigmoid"))

model = keras.models.Sequential([stackedEncoder, stackedDecoder])
model.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.001),
      loss = keras.losses.BinaryCrossentropy(from_logits = False),
      metrics = METRICS
)

input_shape = (None, numFeatures)
model.build(input_shape)

histModel_1 = model.fit(X_train, y_train, epochs = 100, validation_data = [X_test, y_test],
    callbacks = callbacks_list, verbose = False
)

plotResults(0, histModel_1.history, y_test, model.predict(X_test))
```

```
7/7 [==============================] - 0s 1ms/step
On the Validation Data
AUC: 0.9815502166748047
F1 Score:  0.9270833333333334
```







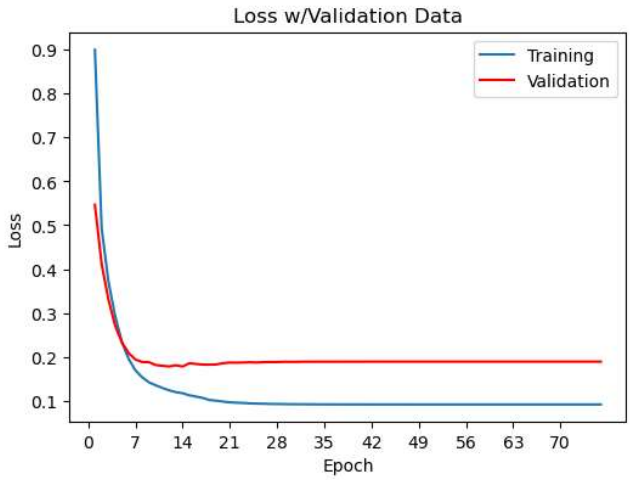


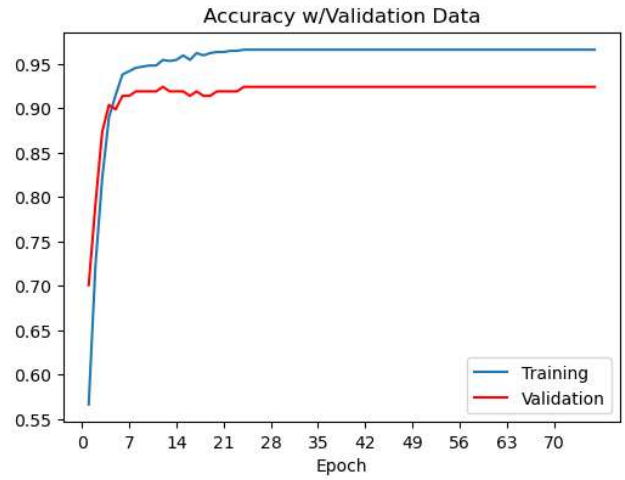
Even with a *very* simple autoencoder we're already seeing impressive results with an AUC of ≈0.9816 and a F1 of ≈0.9271

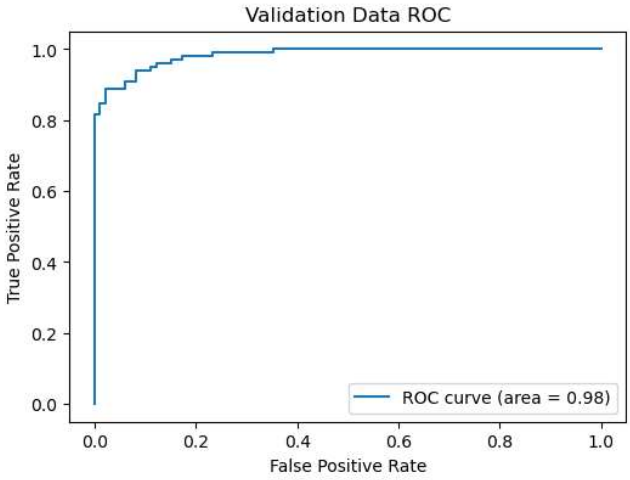
Adding an additional dense layer with 4 units in the encoder and a 16 unit one in the decoder (to create a mirroring of the encoder) improves the AUC to ≈0.9837 but with a slight reduction in F1 to ≈0.9223

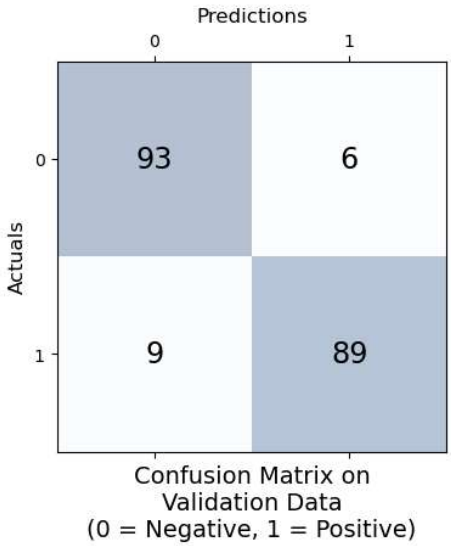
In [16]:
```
tf.keras.backend.clear_session()

stackedEncoder = keras.models.Sequential()
stackedEncoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(4, activation = ENCODER_ACTIVATION))

stackedDecoder = keras.models.Sequential()
stackedDecoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(numFeatures, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(1, activation = "sigmoid"))

model = keras.models.Sequential([stackedEncoder, stackedDecoder])
model.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.001),
      loss = keras.losses.BinaryCrossentropy(from_logits = False),
      metrics = METRICS
)

input_shape = (None, numFeatures)
model.build(input_shape)

histModel_2 = model.fit(X_train, y_train, epochs = 100, validation_data = [X_test, y_test],
    callbacks = callbacks_list, verbose = False
)

plotResults(0, histModel_2.history, y_test, model.predict(X_test))
```

```
7/7 [==============================] - 0s 1ms/step
On the Validation Data
AUC: 0.9837148785591125
F1 Score:  0.922279792746114
```





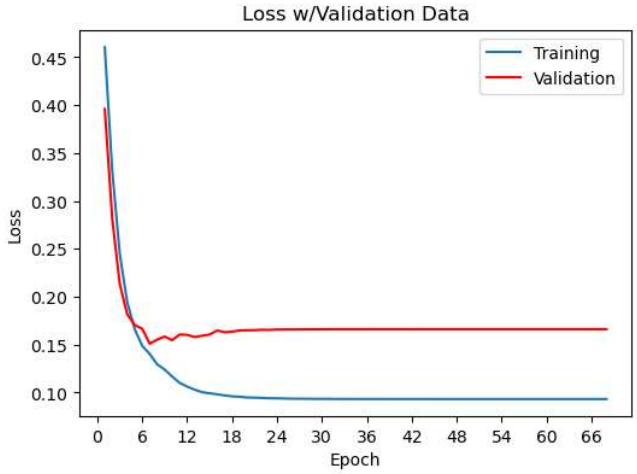


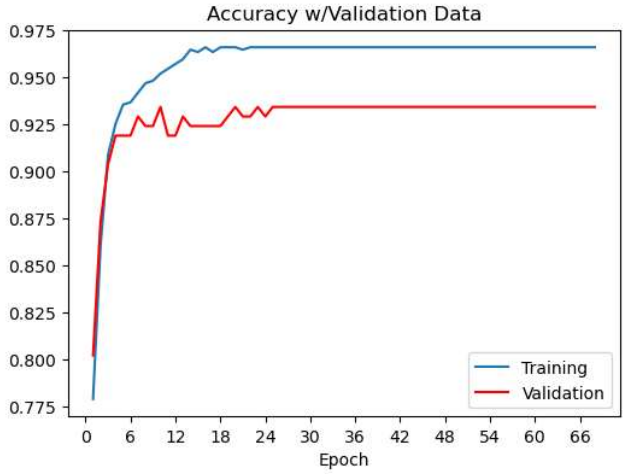


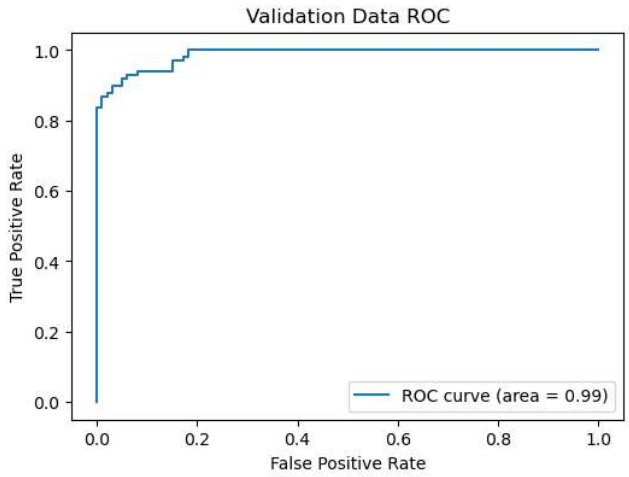
A 3 layer autoencoder is tried next which improves both the AUC to ≈0.9863 and the F1 to ≈0.9326

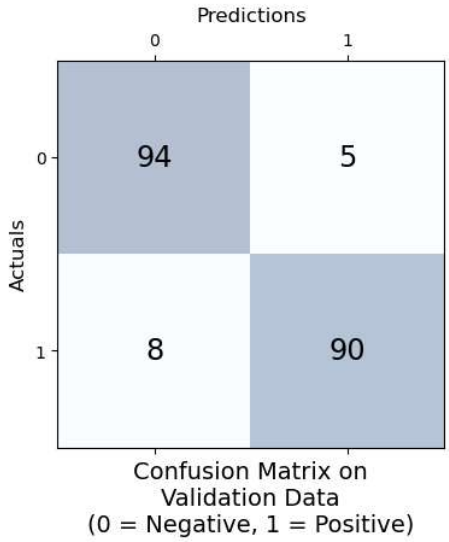
In [15]:
```
tf.keras.backend.clear_session()

stackedEncoder = keras.models.Sequential()
stackedEncoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(8, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(4, activation = ENCODER_ACTIVATION))

stackedDecoder = keras.models.Sequential()
stackedDecoder.add(Dense(8, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(numFeatures, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(1, activation = "sigmoid"))

model = keras.models.Sequential([stackedEncoder, stackedDecoder])
model.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.001),
      loss = keras.losses.BinaryCrossentropy(from_logits = False),
      metrics = METRICS
)

input_shape = (None, numFeatures)
model.build(input_shape)

histModel_3 = model.fit(X_train, y_train, epochs = 100, validation_data = [X_test, y_test],
    callbacks = callbacks_list, verbose = False
)

plotResults(0, histModel_3.history, y_test, model.predict(X_test))
```

```
7/7 [==============================] - 0s 1ms/step
On the Validation Data
AUC: 0.9863430857658386
F1 Score:  0.9326424870466321
```









The results from the 3 layer model look **very** good and this will be the model architecture that the hyperparameters will be tuned with. It will be saved in the next cell for potential use against the full dataset.

In [ ]:
```
model.save(os.path.join("./Models/", "final_model.h5"))
```

Hyperparameter Tuning

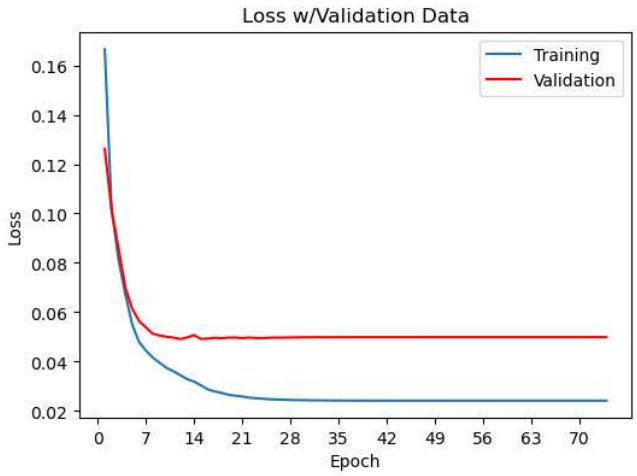
An autoencoder neural network does not have many parameters to tune, particularly if comprised solely of Dense layers. So we'll try changing the loss function and adding layers back. The model below, Model 4, is the same as Model 3, but with Mean Squared Error being used as the loss function.

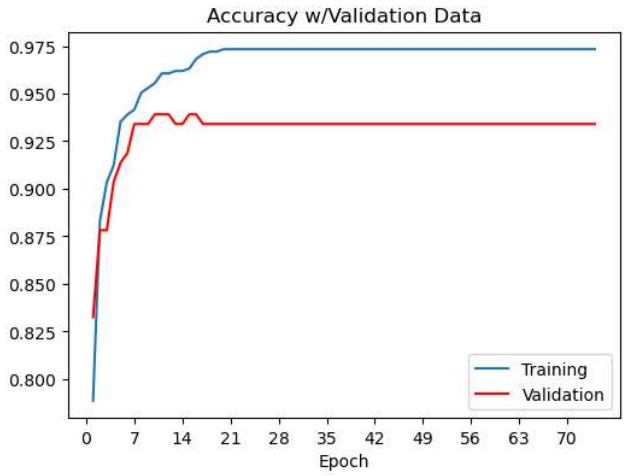
Switching to MSE did improve the F1 score, but at the cost of a reduction in AUC. As the actual dataset is highly unbalanced, the higher AUC acheived with Model 3 means that Model 3's architecture will be carried forward.

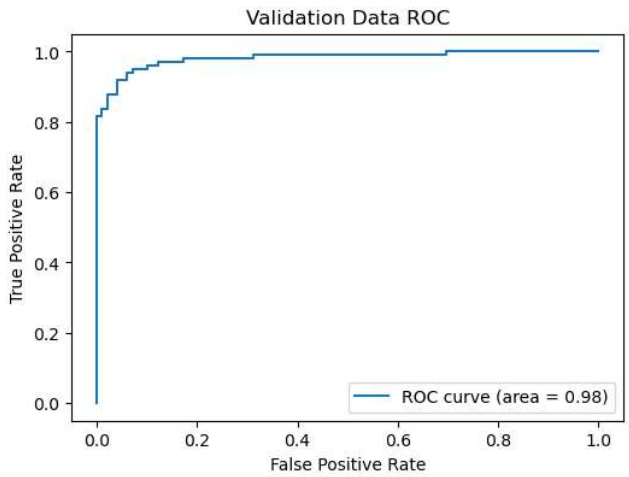
In [78]:
```
tf.keras.backend.clear_session()

stackedEncoder = keras.models.Sequential()
stackedEncoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(8, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(4, activation = ENCODER_ACTIVATION))

stackedDecoder = keras.models.Sequential()
stackedDecoder.add(Dense(8, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(numFeatures, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(1, activation = "sigmoid"))

model = keras.models.Sequential([stackedEncoder, stackedDecoder])
model.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.001),
      loss = "mse",
      metrics = METRICS
)

input_shape = (None, numFeatures)
```

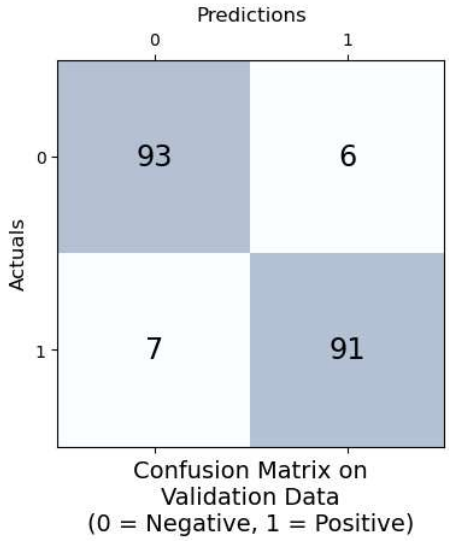
```
model.build(input_shape)

histModel_4 = model.fit(X_train, y_train, epochs = 100, validation_data = [X_test, y_test],
    callbacks = callbacks_list, verbose = False
)

plotResults(0, histModel_4.history, y_test, model.predict(X_test))
```

```
7/7 [==============================] - 0s 1ms/step
On the Validation Data
AUC: 0.9810348381776953
F1 Score:  0.9333333333333333
```







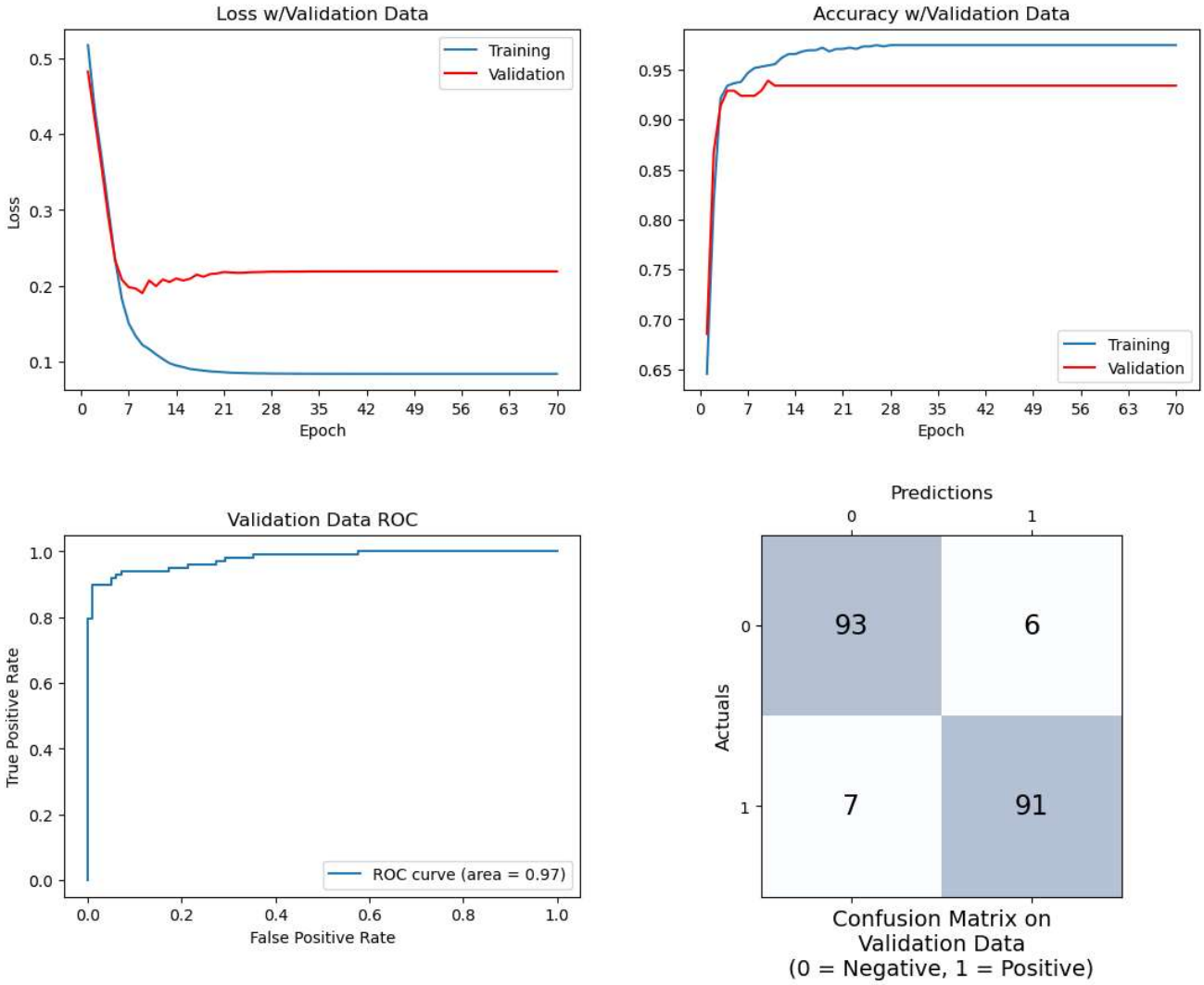


Switching to a relu activation function also did not improve the results.

In [80]:
```
tf.keras.backend.clear_session()

stackedEncoder = keras.models.Sequential()
stackedEncoder.add(Dense(16, activation = "relu"))
stackedEncoder.add(Dense(8, activation = "relu"))
stackedEncoder.add(Dense(4, activation = "relu"))

stackedDecoder = keras.models.Sequential()
stackedDecoder.add(Dense(8, activation = "relu"))
stackedDecoder.add(Dense(16, activation = "relu"))
stackedDecoder.add(Dense(numFeatures, activation = "relu"))
stackedDecoder.add(Dense(1, activation = "sigmoid"))

model = keras.models.Sequential([stackedEncoder, stackedDecoder])
model.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.001),
      loss = keras.losses.BinaryCrossentropy(from_logits = False),
      metrics = METRICS
)

input_shape = (None, numFeatures)
model.build(input_shape)

histModel_4 = model.fit(X_train, y_train, epochs = 100, validation_data = [X_test, y_test],
```

```
    callbacks = callbacks_list, verbose = False
)

plotResults(0, histModel_4.history, y_test, model.predict(X_test))
```

```
7/7 [==============================] - 0s 2ms/step
On the Validation Data
AUC: 0.9774273345701916
F1 Score:  0.9333333333333333
```



The last thing we'll try is to remove some columns with from model which we can visually see in the feature density plots above where there is a very similar distribution between valid and fraudulent transactions.

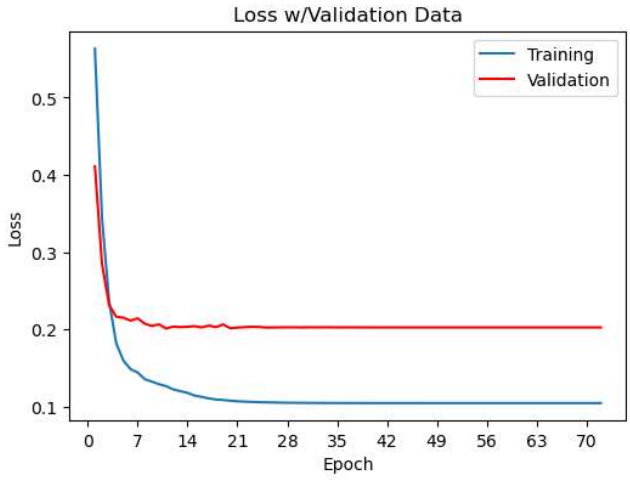
However, again the slight increase in F1 comes at the expense of AUC, so this modification will also be discarded.

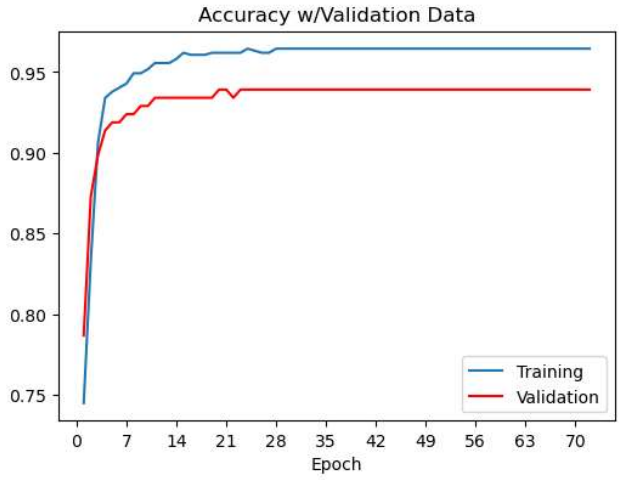
In [81]:
```
dropColumns = ['V13', 'V15', 'V20', 'V22', 'V23', 'V25', 'V26', 'V28', 'Class']
X_drop = df_balanced.drop(dropColumns, axis = 1).values
numFeatures -= len(dropColumns) - 1
X_train, X_test, y_train, y_test = train_test_split(X_drop, y, test_size = 0.20, random_state = RANDOM_SEED)
```

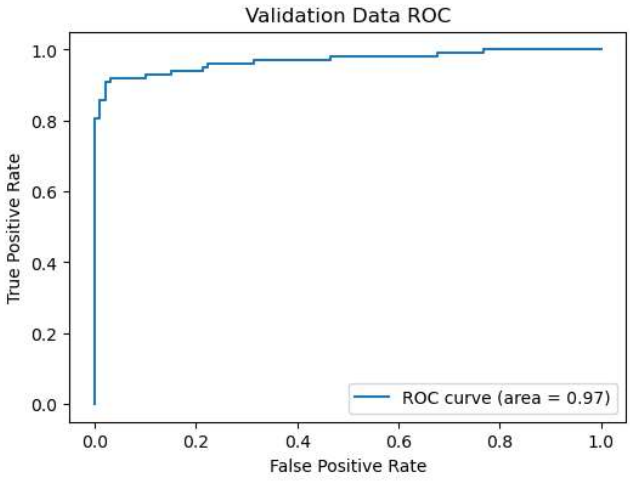
In [83]:
```
tf.keras.backend.clear_session()

stackedEncoder = keras.models.Sequential()
stackedEncoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(8, activation = ENCODER_ACTIVATION))
stackedEncoder.add(Dense(4, activation = ENCODER_ACTIVATION))

stackedDecoder = keras.models.Sequential()
stackedDecoder.add(Dense(8, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(16, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(numFeatures, activation = ENCODER_ACTIVATION))
stackedDecoder.add(Dense(1, activation = "sigmoid"))

model = keras.models.Sequential([stackedEncoder, stackedDecoder])
model.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.001),
      loss = keras.losses.BinaryCrossentropy(from_logits = False),
      metrics = METRICS
```

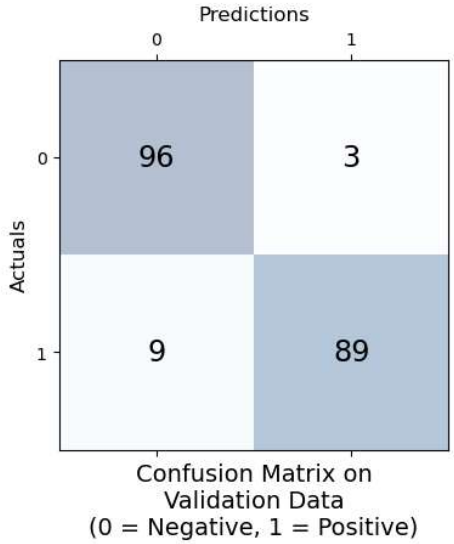
```
)

input_shape = (None, numFeatures)
model.build(input_shape)

histModel_7 = model.fit(X_train, y_train, epochs = 150, validation_data = [X_test, y_test],
    callbacks = callbacks_list, verbose = False
)

plotResults(0, histModel_7.history, y_test, model.predict(X_test))
```

```
7/7 [==============================] - 0s 1ms/step
On the Validation Data
AUC: 0.9684601113172542
F1 Score:  0.9368421052631578
```









Testing the FInal Model Against the Full Dataset

The model previously saved is loaded and its summary is output by the next cell.

In [146…

```
loadedModel = tf.keras.models.load_model(os.path.join("./Models/", "final_model.h5"))
loadedModel.summary()
```

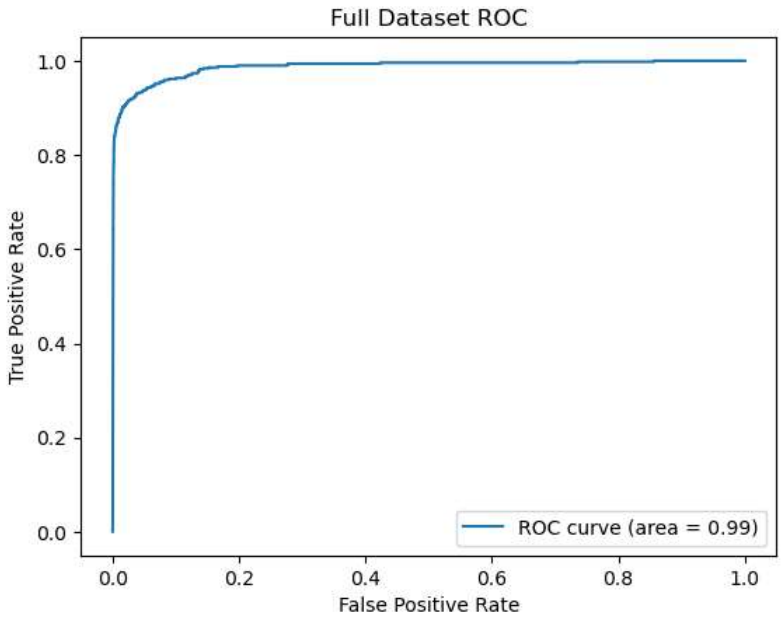
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential (Sequential)     (None, 4)                 652

 sequential_1 (Sequential)   (None, 1)                 707

=================================================================
Total params: 1,359
Trainable params: 1,359
Non-trainable params: 0
_________________________________________________________________
```

Because confusion matrix accuracy is not meaningful for unbalanced datasets, only the receiver operating characteristic curve is displayed below and even on the full dataset our model maintains excellent perfomance with an AUC of ≈0.9871.

In [148…

```
y_full = df_cc_all.Class.values
X_full = df_cc_all.drop('Class', axis = 1).values

preds = loadedModel.predict(X_full)

fullDatasetAUC = roc_auc_score(y_full, preds)
print("AUC:", fullDatasetAUC)

fpr, tpr, threshold = roc_curve(y_full, preds)
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % fullDatasetAUC)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("Full Dataset ROC")
plt.legend(loc="lower right")

plt.show()
```

```
8901/8901 [==============================] - 9s 976us/step
AUC: 0.9871420382951522
```



Results and Analysis

The best result after model and hyperparameter tuning was achieved with a 3 layer autoencoder architecture. The original 29 features were downsampled 29 -> 16 -> 8 -> 4 and then upsampled 4 -> 8 -> 16 -> 29. This model achieved an AUC score on the full dataset of ≈0.9871 and a recall of ≈.9411. Results of all the models and architectures on the balanced dataset are shown below:

Model Description	AUC Score	F1 Score
One Layer	0.9816	0.9271
Two Layer	0.9837	0.9223
Three Layer	0.9863	0.9326
Three Layer w/MSE	0.9810	0.9333
Three Layer w/ReLU	0.9774	0.9333
Three Layer & Drop Cols	0.9685	0.9368

It should be noted that **all** of these results are very tightly grouped. Indeed a single model architecture when re-fit against the balanced dataset could return scores that would vary ±2% sometimes. This is significant because attempting to recreate these results may have varying levels of success. As such, others attempting running this notebook may see results that are worse or better than the results posted here. But these differences, if any, should be minor.

Conclusion and Summary

The ability of even a simple one layer encoder to predict the whether a credit card transaction was fraudulent or not was surprsing. Although the final model was a three layer architecture, the increase in AUC and F1 were only ≈0.5% (0.005). The initial results simply adding a couple layers had the model performing so well that attempts to tune it further were discarded.

An area for improvement in creating and evaluating these models would be to also evaluate the Precision-Recall score of each model against the entire dataset after each training set. The Precision-Recall score is a more recent statistic that does a better job at evaluating a model where there is a severe imbalance in the data. Reevaluating each of the model architectures and tunings against this statistic could result in a different model being chosen.