CSCI E-89 Final Project:
Finding Exoplanets Using FFT and CNNs
Tierney, Sean

Problem Statement:

Using data from NASA's Kepler mission, I tried to predict whether an exoplanet was orbiting a distant star based on the star's measured brightness over a period of 80 days. In theory, the brightness should dim periodically as planets occlude the star. Because of this potential periodicity of the data, I wanted to explore the benefits of using frequency space representation as an input to deep neural networks to perform the prediction.

Overview of Technology:

Dataset:
"Exoplanet Hunting in Deep Space, Kepler labelled time series data", available at Kaggle at the following URL: https://www.kaggle.com/keplersmachines/kepler-labelled-time-series-data

Dataset size: 57MB Format of data file: csv

High Level Overview of Steps:

1. Installed the following python 3 packages using pip: numpy, scipy, matplotlib, pandas, scikit-learn, tensorflow, keras
2. Downloaded data set as described above
3. Loaded data using pandas and performed the Fast Fourier Transform (FFT) using numpy
4. Ran a dense model and a 1D CNN model on both the time-series and frequency domain data
5. Provided test accuracy measurements and confusion plots for each of the four results

Hardware:

MacBook (Retina, 12-inch, Early 2016), 1.2 GHz Intel Core m5, 8 GB 1867 MHz LPDDR3, Intel HD Graphics 515 1536 MB, macOS High Sierra Version 10.13.4

Software:

Python 3.6 (https://www.python.org/downloads/)
Jupyter Lab (https://jupyter.org/install)
Keras with Tensorflow (https://keras.io/#installation)

Lessons Learned:

The Fourier transformed data actually underperformed the time series data. Peak validation accuracy was with the time-series CNN at 0.986. The CNN frequency and Dense time series had similar accuracy at about 0.85. Unfortunately, I have to reject my hypothesis that the periodic nature of the data would play well with a frequency domain transform.

YouTube Presentations:

2 minute: https://youtu.be/OLBq7vF3Gco

15 minute: https://youtu.be/7ToOJ9FdypU

The following console command installs the required python packages. I already had them installed so it doesn't do any extra work besides checking up to date versions.

```
$ pip3 install numpy scipy matplotlib pandas scikit-learn tensorflow keras
Requirement already satisfied: numpy in /Library/Frameworks/Python.framewo
rk/Versions/3.6/lib/python3.6/site-packages (1.13.3)
Requirement already satisfied: scipy in /Library/Frameworks/Python.framewo
rk/Versions/3.6/lib/python3.6/site-packages (1.0.0)
Requirement already satisfied: matplotlib in /Library/Frameworks/Python.fr
amework/Versions/3.6/lib/python3.6/site-packages (2.1.0)
Requirement already satisfied: pandas in /Library/Frameworks/Python.framew
ork/Versions/3.6/lib/python3.6/site-packages (0.21.0)
Requirement already satisfied: scikit-learn in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (0.19.1)
Requirement already satisfied: tensorflow in /Library/Frameworks/Python.fr
amework/Versions/3.6/lib/python3.6/site-packages (1.7.0)
Requirement already satisfied: keras in /Library/Frameworks/Python.framewo
rk/Versions/3.6/lib/python3.6/site-packages (2.1.4)
Requirement already satisfied: six>=1.10 in /Library/Frameworks/Python.fra
mework/Versions/3.6/lib/python3.6/site-packages (from matplotlib) (1.11.0)
Requirement already satisfied: pytz in /Library/Frameworks/Python.framewor
k/Versions/3.6/lib/python3.6/site-packages (from matplotlib) (2018.3)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packa
ges (from matplotlib) (2.2.0)
Requirement already satisfied: python-dateutil>=2.0 in /Library/Frameworks
/Python.framework/Versions/3.6/lib/python3.6/site-packages (from matplotli
b) (2.6.1)
Requirement already satisfied: cycler>=0.10 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from matplotlib) (0.10
.0)
Requirement already satisfied: wheel>=0.26 in /Library/Frameworks/Python.f
ramework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (0.30.
0)
Requirement already satisfied: astor>=0.6.0 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (0.6.
2)
Requirement already satisfied: protobuf>=3.4.0 in /Library/Frameworks/Pyth
```

```
on.framework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (3
.5.1)
Requirement already satisfied: gast>=0.2.0 in /Library/Frameworks/Python.f
ramework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (0.2.0
)
Requirement already satisfied: absl-py>=0.1.6 in /Library/Frameworks/Pytho
n.framework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (0.
1.10)
Requirement already satisfied: grpcio>=1.8.6 in /Library/Frameworks/Python
.framework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (1.1
1.0)
Requirement already satisfied: tensorboard<1.8.0,>=1.7.0 in /Library/Frame
works/Python.framework/Versions/3.6/lib/python3.6/site-packages (from tens
orflow) (1.7.0)
Requirement already satisfied: termcolor>=1.1.0 in /Library/Frameworks/Pyt
hon.framework/Versions/3.6/lib/python3.6/site-packages (from tensorflow) (
1.1.0)
Requirement already satisfied: pyyaml in /Library/Frameworks/Python.framew
ork/Versions/3.6/lib/python3.6/site-packages (from keras) (3.12)
Requirement already satisfied: setuptools in /Library/Frameworks/Python.fr
amework/Versions/3.6/lib/python3.6/site-packages (from protobuf>=3.4.0->te
nsorflow) (38.5.1)
Requirement already satisfied: markdown>=2.6.8 in /Library/Frameworks/Pyth
on.framework/Versions/3.6/lib/python3.6/site-packages (from tensorboard<1.
8.0,>=1.7.0->tensorflow) (2.6.11)
Requirement already satisfied: html5lib==0.9999999 in /Library/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/site-packages (from tensorboar
d<1.8.0,>=1.7.0->tensorflow) (0.9999999)
Requirement already satisfied: werkzeug>=0.11.10 in /Library/Frameworks/Py
thon.framework/Versions/3.6/lib/python3.6/site-packages (from tensorboard<
1.8.0,>=1.7.0->tensorflow) (0.14.1)
Requirement already satisfied: bleach==1.5.0 in /Library/Frameworks/Python
.framework/Versions/3.6/lib/python3.6/site-packages (from tensorboard<1.8.
0,>=1.7.0->tensorflow) (1.5.0)
```

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import keras
```

```
Using TensorFlow backend.
```

Now that I've loaded the fundamental python packages, I'll now load the star brightness intensity data as provided from Kaggle.

In [2]:
```
datadir = 'kepler-labelled-time-series-data'
train_file = 'exotrain.csv'
test_file = 'exotest.csv'

train_df = pd.read_csv(datadir + '/' + train_file, sep=',')
test_df = pd.read_csv(datadir + '/' + test_file, sep=',')
```

Let's take a look at what the raw training data looks like.

In [3]:
```
train_df.head()
```

Out[3]:

	LABEL	FLUX.1	FLUX.2	FLUX.3	FLUX.4	FLUX.5	FLUX.6	FLUX.7	FLUX.8
0	2	93.85	83.81	20.10	-26.98	-39.56	-124.71	-135.18	-96.27
1	2	-38.88	-33.83	-58.54	-40.09	-79.31	-72.81	-86.55	-85.33
2	2	532.64	535.92	513.73	496.92	456.45	466.00	464.50	486.39
3	2	326.52	347.39	302.35	298.13	317.74	312.70	322.33	311.31
4	2	-1107.21	-1112.59	-1118.95	-1095.10	-1057.55	-1034.48	-998.34	-1022.71

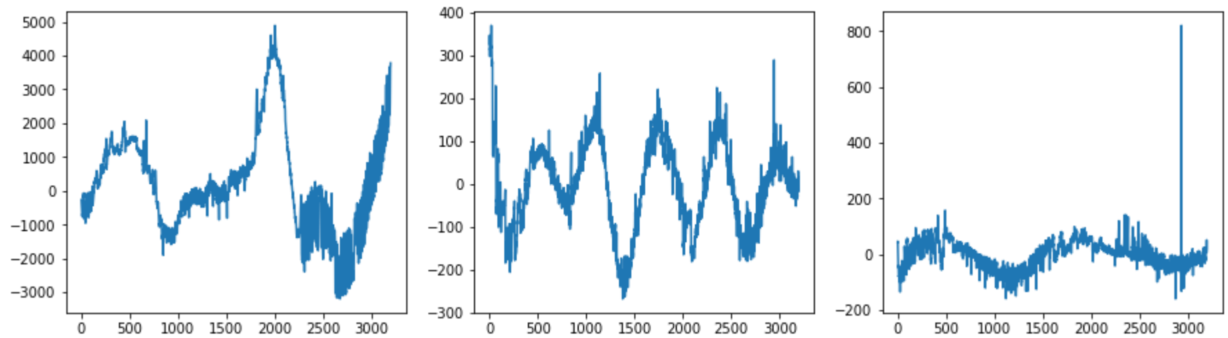
5 rows × 3198 columns

Each row has a label. 2 means there is an exoplanet present, and 1 means there is no exoplanet present. Now let's visualize what this intensity data looks like over time.

In [4]:
```
exo_pos = train_df[train_df['LABEL'] == 2].values[:, 1:]
exo_neg = train_df[train_df['LABEL'] == 1].values[:, 1:]
```

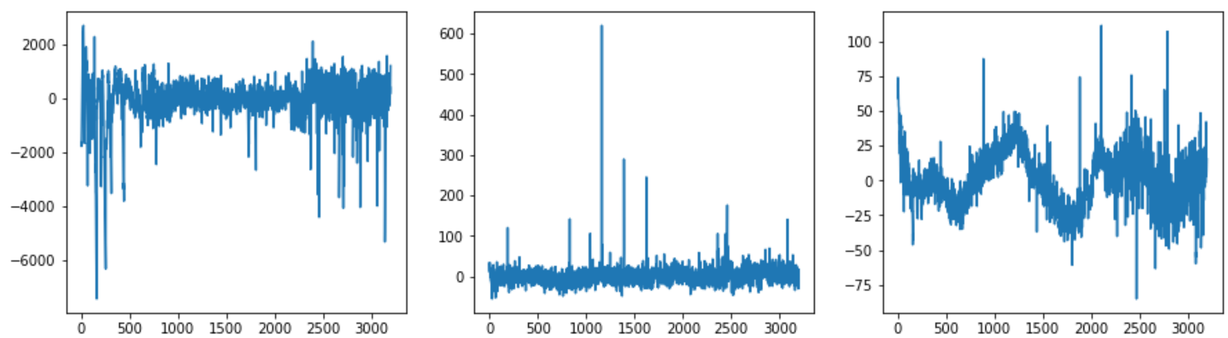
These are some stars with exoplanets.

In [5]:
```
pos_ex = exo_pos[np.random.choice(len(exo_pos), 3)]

fig, axes = plt.subplots(1, 3)
fig.set_figwidth(15)
for i, ax in enumerate(axes):
    ax.plot(pos_ex[i])
```



And these are stars without exoplanets.

In [6]:
```
neg_ex = exo_neg[np.random.choice(len(exo_neg), 3)]

fig, axes = plt.subplots(1, 3)
fig.set_figwidth(15)
for i, ax in enumerate(axes):
    ax.plot(neg_ex[i])
```



There were initially quite a few issues with overfitting, due to the fact that there are many many more examples of systems without exoplanets than examples with exoplanets. So, I've replicated the exoplanet data enough times in the dataset enough times where they're approximately equal.

In [7]:
```
oversample_rate = exo_neg.shape[0] // exo_pos.shape[0]
exo_pos = np.tile(exo_pos, (oversample_rate, 1))
```

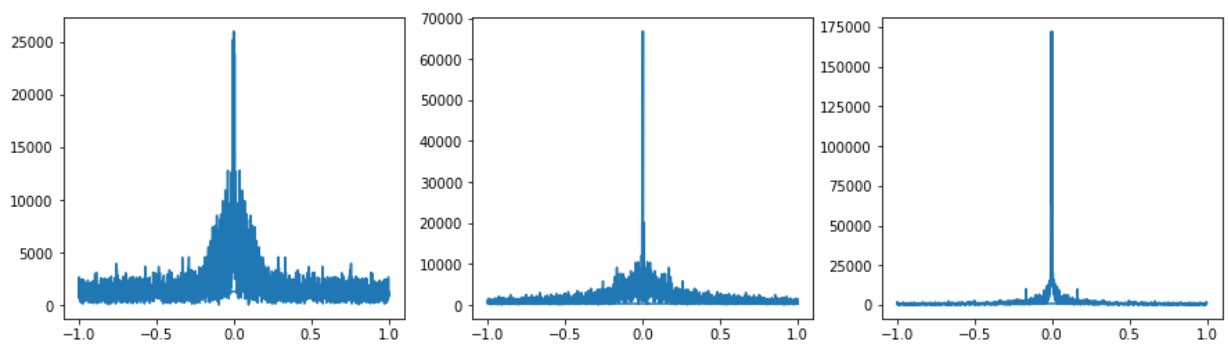
The Fourier transform is a way of changing time-series signals to the frequency domain. It's based on the notion that any function can be represented as an infinite sum of sinusoids of all frequencies. Because the data is inherently periodic, I wanted to explore the results of neural networks comparing time-series and frequency domain inputs.

Here is where I apply the Fast Fourier Transform (FFT) to the two time-series inputs.

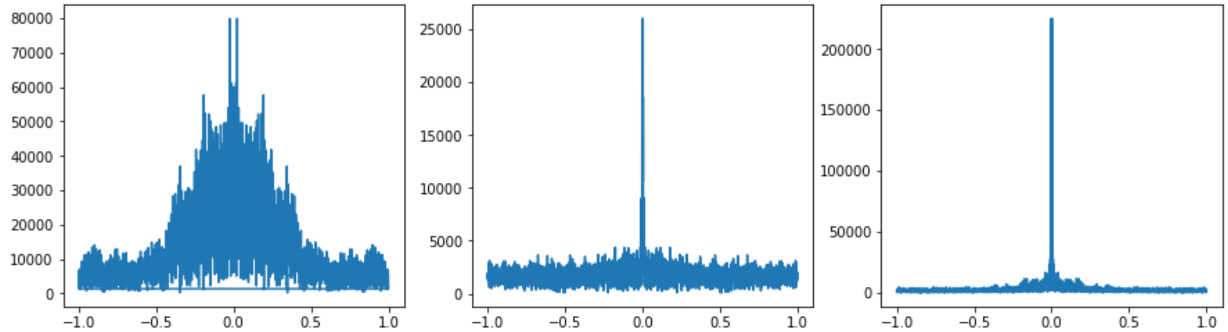
In [8]:
```
from numpy.fft import fft, fftfreq
f = fftfreq(exo_pos[0].shape[-1], d=0.5)

pos_freq = fft(exo_pos)
neg_freq = fft(exo_neg)
```

Let's take a look, again, at some examples of data, but this time in the frequency domain.

In [9]:
```
pos_ex = pos_freq[np.random.choice(len(pos_freq), 3)]

fig, axes = plt.subplots(1, 3)
fig.set_figwidth(15)
for i, ax in enumerate(axes):
    ax.plot(f, np.abs(pos_ex[i]))# .real, f, pos_ex[i].imag)
plt.show()
```



In [10]:
```
neg_ex = neg_freq[np.random.choice(len(neg_freq), 3)]

fig, axes = plt.subplots(1, 3)
fig.set_figwidth(15)
for i, ax in enumerate(axes):
    ax.plot(f, np.abs(neg_ex[i])) #.real, f, neg_ex[i].imag)
plt.show()
```



Here I combine the data into full tensors, ready for feeding into the neural networks. First training data, then test data.

In [11]:
```
ts_data = np.concatenate([exo_pos, exo_neg])
freq_data = np.abs(np.concatenate([pos_freq, neg_freq]))
labels = np.concatenate([np.ones(len(exo_pos)), np.zeros(len(exo_neg))
])
```

In [12]:
```
ts_test = test_df.values[:, 1:]
freq_test = np.abs(fft(test_df.values[:, 1:]))
labels_test = test_df['LABEL'].values - 1
```

Neural networks often perform better when scaled to have zero mean and unit variance. Here I use scikit-learn's robust scale function to scale my input data.

In [13]:
```
from sklearn.preprocessing import robust_scale
ts_data = robust_scale(ts_data)
ts_test = robust_scale(ts_test)

freq_data = robust_scale(freq_data)
freq_test = robust_scale(freq_test)
```

In [14]:
```
ts_data.shape
```

Out[14]: (10082, 3197)

The first model is a fairly simple Multi-Layer Perceptron (MLP). Only one hidden layer and with regularization and dropout to fight overfitting. I use binary cross-entropy as the loss function, and the efficient Adam optimizer.

In [15]:
```
from keras import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2

def dense_model():
    model = Sequential()
    model.add(Dense(256, activation='relu', kernel_regularizer=l2(1e-3
),
                    input_shape=(ts_data.shape[1], )))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    model.summary()
    return model
```

For the convolutional model, I use three convolutional layers, settled on by repeated testing. I also added some regularization to the final two convolutional layers. After those layers, a 64-wide linear layer and a final single neuron with sigmoid activation round out the network.

Again, I use the binary cross-entropy loss function and Adam optimizer.

In [16]:
```
from keras import Sequential
from keras.layers import Dense, Conv1D, MaxPooling1D, Flatten
from keras.regularizers import l2

def conv_model():
    model = Sequential()
    model.add(Conv1D(16, 16, activation='relu', input_shape=(freq_data
.shape[1], 1)))
    model.add(MaxPooling1D(4))
    model.add(Conv1D(32, 16, activation='relu', kernel_regularizer=l2(
1e-4)))
    model.add(MaxPooling1D(4))
    model.add(Conv1D(64, 16, activation='relu', kernel_regularizer=l2(
1e-3)))
    model.add(MaxPooling1D(4))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    model.summary()
    return model
```

Now that the networks have been set up, let's run each with time series and frequency domain data and see how they perform.

In [17]:
```
test_results = {}
```

In [18]:
```
model_name = 'Dense Time Series'
model = dense_model()
ts_dense_history = model.fit(ts_data, labels,
                             validation_data=(ts_test, labels_test),
                             batch_size=64, epochs=20, shuffle=True)
test_results[model_name] = model.predict(ts_test)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               818688
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
```

```
Total params: 818,945
Trainable params: 818,945
Non-trainable params: 0
_________________________________________________________________
Train on 10082 samples, validate on 570 samples
Epoch 1/20
10082/10082 [==============================] - 3s 336us/step - loss: 1.4184 - acc: 0.9089 - val_loss: 1.3639 - val_acc: 0.8632
Epoch 2/20
10082/10082 [==============================] - 4s 367us/step - loss: 1.2568 - acc: 0.9479 - val_loss: 0.8095 - val_acc: 0.9404
Epoch 3/20
10082/10082 [==============================] - 4s 425us/step - loss: 1.3379 - acc: 0.9345 - val_loss: 1.1982 - val_acc: 0.9053
Epoch 4/20
10082/10082 [==============================] - 4s 359us/step - loss: 1.0171 - acc: 0.9606 - val_loss: 1.3528 - val_acc: 0.8702
Epoch 5/20
10082/10082 [==============================] - 4s 355us/step - loss: 0.9632 - acc: 0.9618 - val_loss: 0.6803 - val_acc: 0.9614
Epoch 6/20
10082/10082 [==============================] - 4s 371us/step - loss: 1.2671 - acc: 0.9382 - val_loss: 1.7544 - val_acc: 0.8439
Epoch 7/20
10082/10082 [==============================] - 3s 343us/step - loss: 1.6197 - acc: 0.9191 - val_loss: 1.9881 - val_acc: 0.8246
Epoch 8/20
10082/10082 [==============================] - 3s 306us/step - loss: 1.4426 - acc: 0.9338 - val_loss: 0.8022 - val_acc: 0.9544
Epoch 9/20
10082/10082 [==============================] - 3s 317us/step - loss: 1.3014 - acc: 0.9427 - val_loss: 0.7063 - val_acc: 0.9632
Epoch 10/20
10082/10082 [==============================] - 3s 330us/step - loss: 1.2723 - acc: 0.9402 - val_loss: 0.5847 - val_acc: 0.9632
Epoch 11/20
10082/10082 [==============================] - 3s 309us/step - loss: 1.2477 - acc: 0.9403 - val_loss: 0.5720 - val_acc: 0.9614
Epoch 12/20
10082/10082 [==============================] - 3s 303us/step - loss: 1.3749 - acc: 0.9286 - val_loss: 1.7692 - val_acc: 0.8561
Epoch 13/20
10082/10082 [==============================] - 4s 356us/step - loss: 1.3578 - acc: 0.9312 - val_loss: 0.6777 - val_acc: 0.9509
Epoch 14/20
10082/10082 [==============================] - 3s 324us/step - loss: 1.2275 - acc: 0.9400 - val_loss: 0.6945 - val_acc: 0.9526
Epoch 15/20
10082/10082 [==============================] - 3s 316us/step - loss: 1.2393 - acc: 0.9382 - val_loss: 0.5891 - val_acc: 0.9544
```

```
Epoch 16/20
10082/10082 [==============================] - 3s 336us/step - loss:
1.4433 - acc: 0.9230 - val_loss: 1.1134 - val_acc: 0.9158
Epoch 17/20
10082/10082 [==============================] - 4s 356us/step - loss:
1.5477 - acc: 0.9216 - val_loss: 0.6970 - val_acc: 0.9596
Epoch 18/20
10082/10082 [==============================] - 4s 386us/step - loss:
1.8665 - acc: 0.8974 - val_loss: 1.5582 - val_acc: 0.8702
Epoch 19/20
10082/10082 [==============================] - 5s 461us/step - loss:
1.5651 - acc: 0.9228 - val_loss: 0.7557 - val_acc: 0.9439
Epoch 20/20
10082/10082 [==============================] - 4s 443us/step - loss:
1.5078 - acc: 0.9252 - val_loss: 1.2747 - val_acc: 0.8842
```

In [19]:

```
model_name = 'CNN Time Series'
model = conv_model()
ts_conv_history = model.fit(ts_data[:, :, None], labels,
                            validation_data=(ts_test[:, :, None], labe
ls_test),
                            batch_size=64, epochs=10, shuffle=True)

test_results[model_name] = model.predict(ts_test[:, :, None])
```

```
WARNING:tensorflow:From /Library/Frameworks/Python.framework/Version
s/3.6/lib/python3.6/site-packages/tensorflow/python/util/deprecation
.py:497: calling conv1d (from tensorflow.python.ops.nn_ops) with dat
a_format=NHWC is deprecated and will be removed in a future version.
Instructions for updating:
`NHWC` for data_format is deprecated, use `NWC` instead
```

Layer (type)	Output Shape	Param #
conv1d_1 (Conv1D)	(None, 3182, 16)	272
max_pooling1d_1 (MaxPooling1	(None, 795, 16)	0
conv1d_2 (Conv1D)	(None, 780, 32)	8224
max_pooling1d_2 (MaxPooling1	(None, 195, 32)	0
conv1d_3 (Conv1D)	(None, 180, 64)	32832
max_pooling1d_3 (MaxPooling1	(None, 45, 64)	0
flatten_1 (Flatten)	(None, 2880)	0
dense_3 (Dense)	(None, 64)	184384

```
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65
=================================================================
Total params: 225,777
Trainable params: 225,777
Non-trainable params: 0
_________________________________________________________________
Train on 10082 samples, validate on 570 samples
Epoch 1/20
10082/10082 [==============================] - 43s 4ms/step - loss:
0.5577 - acc: 0.8163 - val_loss: 0.3933 - val_acc: 0.9351
Epoch 2/20
10082/10082 [==============================] - 43s 4ms/step - loss:
0.1957 - acc: 0.9633 - val_loss: 0.2449 - val_acc: 0.9667
Epoch 3/20
10082/10082 [==============================] - 42s 4ms/step - loss:
0.0753 - acc: 0.9960 - val_loss: 0.1918 - val_acc: 0.9807
Epoch 4/20
10082/10082 [==============================] - 42s 4ms/step - loss:
0.0637 - acc: 0.9977 - val_loss: 0.1578 - val_acc: 0.9825
Epoch 5/20
10082/10082 [==============================] - 45s 4ms/step - loss:
0.0550 - acc: 0.9988 - val_loss: 0.1982 - val_acc: 0.9789
Epoch 6/20
10082/10082 [==============================] - 45s 4ms/step - loss:
0.0517 - acc: 0.9983 - val_loss: 0.4095 - val_acc: 0.8737
Epoch 7/20
10082/10082 [==============================] - 43s 4ms/step - loss:
0.0600 - acc: 0.9974 - val_loss: 0.1308 - val_acc: 0.9912
Epoch 8/20
10082/10082 [==============================] - 43s 4ms/step - loss:
0.0581 - acc: 0.9974 - val_loss: 0.1400 - val_acc: 0.9877
Epoch 9/20
10082/10082 [==============================] - 44s 4ms/step - loss:
0.0449 - acc: 0.9992 - val_loss: 0.1763 - val_acc: 0.9860
Epoch 10/20
10082/10082 [==============================] - 48s 5ms/step - loss:
0.0407 - acc: 0.9991 - val_loss: 0.1544 - val_acc: 0.9860
Epoch 11/20
10082/10082 [==============================] - 60s 6ms/step - loss:
0.0351 - acc: 0.9996 - val_loss: 0.1322 - val_acc: 0.9912
Epoch 12/20
10082/10082 [==============================] - 45s 5ms/step - loss:
0.0419 - acc: 0.9982 - val_loss: 0.1057 - val_acc: 0.9895
Epoch 13/20
10082/10082 [==============================] - 55s 5ms/step - loss:
0.1842 - acc: 0.9767 - val_loss: 0.3085 - val_acc: 0.9544
Epoch 14/20
10082/10082 [==============================] - 45s 4ms/step - loss:
```

```
0.0822 - acc: 0.9946 - val_loss: 0.1611 - val_acc: 0.9912
Epoch 15/20
10082/10082 [==============================] - 53s 5ms/step - loss:
0.0431 - acc: 0.9996 - val_loss: 0.1338 - val_acc: 0.9930
Epoch 16/20
10082/10082 [==============================] - 53s 5ms/step - loss:
0.0421 - acc: 0.9993 - val_loss: 0.1082 - val_acc: 0.9930
Epoch 17/20
10082/10082 [==============================] - 51s 5ms/step - loss:
0.0354 - acc: 0.9998 - val_loss: 0.1417 - val_acc: 0.9895
Epoch 18/20
10082/10082 [==============================] - 50s 5ms/step - loss:
0.0331 - acc: 0.9997 - val_loss: 0.1296 - val_acc: 0.9895
Epoch 19/20
10082/10082 [==============================] - 51s 5ms/step - loss:
0.0320 - acc: 0.9997 - val_loss: 0.1109 - val_acc: 0.9895
Epoch 20/20
10082/10082 [==============================] - 50s 5ms/step - loss:
0.0278 - acc: 0.9998 - val_loss: 0.1288 - val_acc: 0.9895
```

In [20]:

```
model_name = 'Dense Frequency'

model = dense_model()
freq_dense_history = model.fit(freq_data, labels,
                               validation_data=(freq_test, labels_test
),
                               batch_size=64, epochs=20, shuffle=True)

test_results[model_name] = model.predict(freq_test)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 256)               818688
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 257
=================================================================
Total params: 818,945
Trainable params: 818,945
Non-trainable params: 0
_________________________________________________________________
Train on 10082 samples, validate on 570 samples
Epoch 1/20
10082/10082 [==============================] - 5s 456us/step - loss:
1.1122 - acc: 0.9165 - val_loss: 8.4813 - val_acc: 0.4561
Epoch 2/20
10082/10082 [==============================] - 4s 413us/step - loss:
1.9578 - acc: 0.8864 - val_loss: 8.9229 - val_acc: 0.4544
```

```
Epoch 3/20
10082/10082 [==============================] - 5s 473us/step - loss:
2.7023 - acc: 0.8470 - val_loss: 5.3557 - val_acc: 0.6018
Epoch 4/20
10082/10082 [==============================] - 4s 383us/step - loss:
0.8678 - acc: 0.9710 - val_loss: 4.3277 - val_acc: 0.6842
Epoch 5/20
10082/10082 [==============================] - 4s 384us/step - loss:
0.7419 - acc: 0.9819 - val_loss: 2.3452 - val_acc: 0.8211
Epoch 6/20
10082/10082 [==============================] - 4s 427us/step - loss:
0.5953 - acc: 0.9888 - val_loss: 1.2290 - val_acc: 0.8877
Epoch 7/20
10082/10082 [==============================] - 4s 434us/step - loss:
0.5482 - acc: 0.9891 - val_loss: 1.3281 - val_acc: 0.8737
Epoch 8/20
10082/10082 [==============================] - 4s 421us/step - loss:
0.4969 - acc: 0.9896 - val_loss: 3.6115 - val_acc: 0.7035
Epoch 9/20
10082/10082 [==============================] - 4s 441us/step - loss:
0.4163 - acc: 0.9936 - val_loss: 2.4507 - val_acc: 0.7754
Epoch 10/20
10082/10082 [==============================] - 5s 458us/step - loss:
0.4837 - acc: 0.9838 - val_loss: 1.8003 - val_acc: 0.8561
Epoch 11/20
10082/10082 [==============================] - 5s 452us/step - loss:
0.4075 - acc: 0.9917 - val_loss: 3.5983 - val_acc: 0.7035
Epoch 12/20
10082/10082 [==============================] - 4s 427us/step - loss:
0.5500 - acc: 0.9792 - val_loss: 6.4827 - val_acc: 0.5263
Epoch 13/20
10082/10082 [==============================] - 5s 468us/step - loss:
0.4215 - acc: 0.9897 - val_loss: 2.5218 - val_acc: 0.7684
Epoch 14/20
10082/10082 [==============================] - 4s 419us/step - loss:
0.3727 - acc: 0.9920 - val_loss: 1.0648 - val_acc: 0.8860
Epoch 15/20
10082/10082 [==============================] - 4s 404us/step - loss:
0.3261 - acc: 0.9924 - val_loss: 1.2116 - val_acc: 0.8737
Epoch 16/20
10082/10082 [==============================] - 4s 405us/step - loss:
0.3083 - acc: 0.9919 - val_loss: 2.6885 - val_acc: 0.7368
Epoch 17/20
10082/10082 [==============================] - 4s 401us/step - loss:
1.7417 - acc: 0.8868 - val_loss: 7.1596 - val_acc: 0.4930
Epoch 18/20
10082/10082 [==============================] - 4s 400us/step - loss:
0.7377 - acc: 0.9694 - val_loss: 2.8006 - val_acc: 0.7579
Epoch 19/20
10082/10082 [==============================] - 4s 403us/step - loss:
```

```
0.5396 - acc: 0.9843 - val_loss: 1.4611 - val_acc: 0.8544
Epoch 20/20
10082/10082 [==============================] - 4s 400us/step - loss:
0.4548 - acc: 0.9885 - val_loss: 4.9244 - val_acc: 0.6175
```

In [21]:
```
model_name = 'CNN Frequency'

model = conv_model()
freq_conv_history = model.fit(freq_data[:, :, None], labels,
                              validation_data=(freq_test[:, :, None],
labels_test),
                              batch_size=64, epochs=10, shuffle=True)

test_results[model_name] = model.predict(freq_test[:, :, None])
```

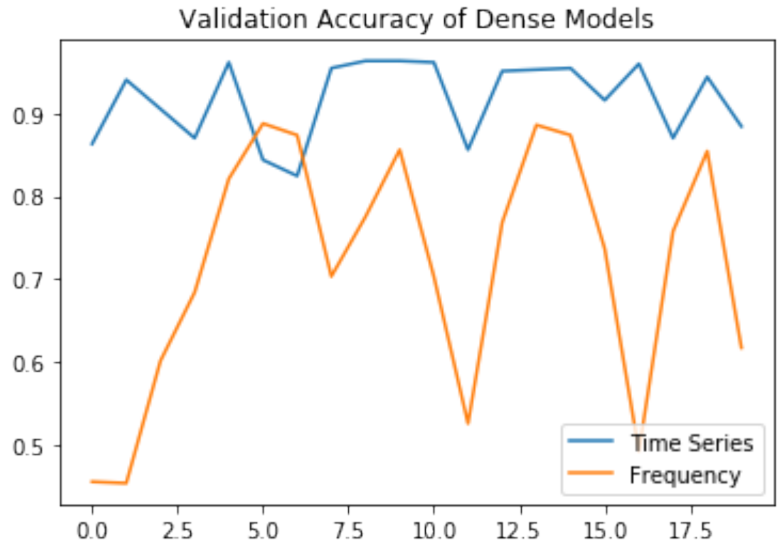
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_4 (Conv1D)            (None, 3182, 16)          272
_________________________________________________________________
max_pooling1d_4 (MaxPooling1 (None, 795, 16)           0
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 780, 32)           8224
_________________________________________________________________
max_pooling1d_5 (MaxPooling1 (None, 195, 32)           0
_________________________________________________________________
conv1d_6 (Conv1D)            (None, 180, 64)           32832
_________________________________________________________________
max_pooling1d_6 (MaxPooling1 (None, 45, 64)            0
_________________________________________________________________
flatten_2 (Flatten)          (None, 2880)              0
_________________________________________________________________
dense_7 (Dense)              (None, 64)                184384
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 65
=================================================================
Total params: 225,777
Trainable params: 225,777
Non-trainable params: 0
_________________________________________________________________
Train on 10082 samples, validate on 570 samples
Epoch 1/20
10082/10082 [==============================] - 55s 5ms/step - loss:
0.5647 - acc: 0.7801 - val_loss: 0.8280 - val_acc: 0.6123
Epoch 2/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.1854 - acc: 0.9669 - val_loss: 0.4172 - val_acc: 0.8368
Epoch 3/20
```

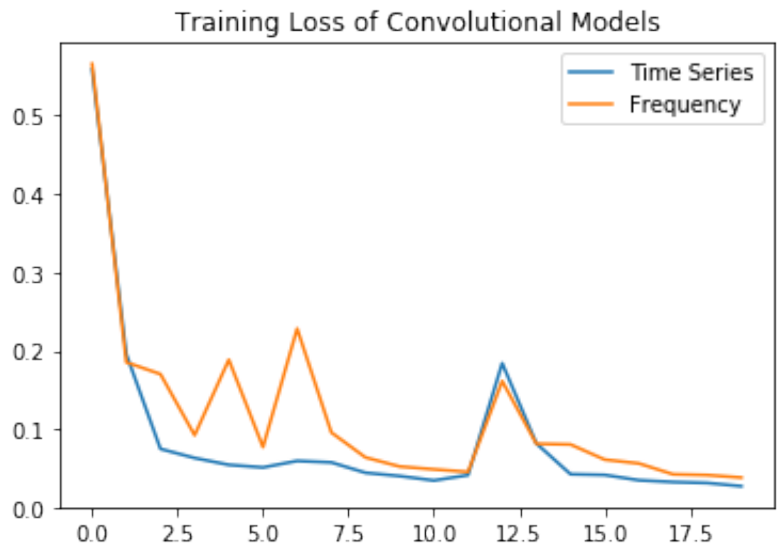
```
10082/10082 [==============================] - 53s 5ms/step - loss:
0.1703 - acc: 0.9731 - val_loss: 0.6776 - val_acc: 0.7667
Epoch 4/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.0927 - acc: 0.9914 - val_loss: 0.1228 - val_acc: 0.9895
Epoch 5/20
10082/10082 [==============================] - 51s 5ms/step - loss:
0.1886 - acc: 0.9604 - val_loss: 0.3189 - val_acc: 0.8754
Epoch 6/20
10082/10082 [==============================] - 53s 5ms/step - loss:
0.0776 - acc: 0.9948 - val_loss: 0.2250 - val_acc: 0.9211
Epoch 7/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.2282 - acc: 0.9633 - val_loss: 0.2143 - val_acc: 0.9579
Epoch 8/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.0961 - acc: 0.9885 - val_loss: 0.3574 - val_acc: 0.8842
Epoch 9/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.0644 - acc: 0.9962 - val_loss: 0.2332 - val_acc: 0.9228
Epoch 10/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.0529 - acc: 0.9976 - val_loss: 0.2568 - val_acc: 0.9158
Epoch 11/20
10082/10082 [==============================] - 53s 5ms/step - loss:
0.0493 - acc: 0.9982 - val_loss: 0.2223 - val_acc: 0.9368
Epoch 12/20
10082/10082 [==============================] - 51s 5ms/step - loss:
0.0460 - acc: 0.9989 - val_loss: 0.2073 - val_acc: 0.9439
Epoch 13/20
10082/10082 [==============================] - 53s 5ms/step - loss:
0.1614 - acc: 0.9714 - val_loss: 0.5003 - val_acc: 0.7772
Epoch 14/20
10082/10082 [==============================] - 54s 5ms/step - loss:
0.0817 - acc: 0.9931 - val_loss: 0.3351 - val_acc: 0.8772
Epoch 15/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.0809 - acc: 0.9909 - val_loss: 0.3915 - val_acc: 0.8737
Epoch 16/20
10082/10082 [==============================] - 52s 5ms/step - loss:
0.0616 - acc: 0.9966 - val_loss: 0.3428 - val_acc: 0.8825
Epoch 17/20
10082/10082 [==============================] - 53s 5ms/step - loss:
0.0568 - acc: 0.9961 - val_loss: 0.2129 - val_acc: 0.9263
Epoch 18/20
10082/10082 [==============================] - 48s 5ms/step - loss:
0.0429 - acc: 0.9993 - val_loss: 0.1723 - val_acc: 0.9421
Epoch 19/20
10082/10082 [==============================] - 44s 4ms/step - loss:
0.0420 - acc: 0.9990 - val_loss: 0.2194 - val_acc: 0.9193
```

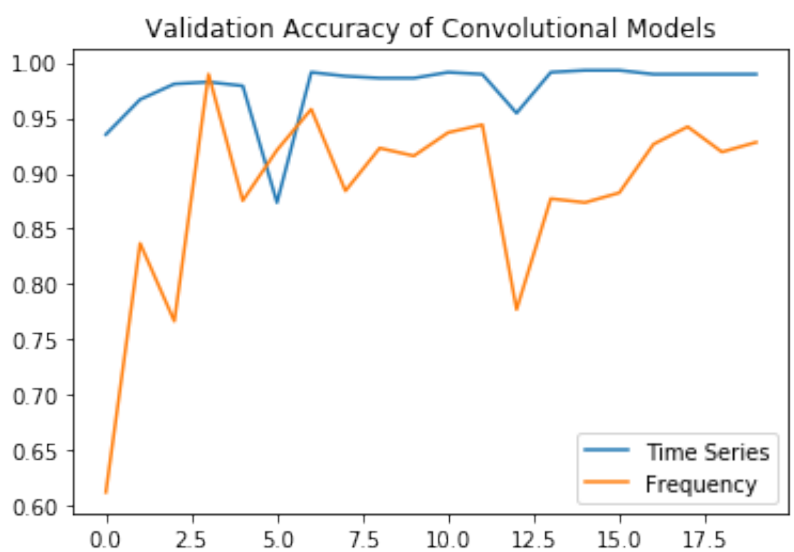
```
Epoch 20/20
10082/10082 [==============================] - 47s 5ms/step - loss:
0.0388 - acc: 0.9993 - val_loss: 0.2045 - val_acc: 0.9281
```

Let's plot the training loss and validation accuracy to get a sense for how well the models performed.

In [23]:
```
plt.plot(ts_dense_history.history['loss'], label='Time Series')
plt.plot(freq_dense_history.history['loss'], label='Frequency')
plt.legend()
plt.title('Training Loss of Dense Models')
plt.show()

plt.plot(ts_dense_history.history['val_acc'], label='Time Series')
plt.plot(freq_dense_history.history['val_acc'], label='Frequency')
plt.legend()
plt.title('Validation Accuracy of Dense Models')
plt.show()

plt.plot(ts_conv_history.history['loss'], label='Time Series')
plt.plot(freq_conv_history.history['loss'], label='Frequency')
plt.legend()
plt.title('Training Loss of Convolutional Models')
plt.show()

plt.plot(ts_conv_history.history['val_acc'], label='Time Series')
plt.plot(freq_conv_history.history['val_acc'], label='Frequency')
plt.legend()
plt.title('Validation Accuracy of Convolutional Models')
plt.show()
```


Validation Accuracy of Dense Models
0.9
0.8
0.7
0.6
0.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Time Series
Frequency


Training Loss of Convolutional Models
0.5
0.4
0.3
0.2
0.1
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Time Series
Frequency


Validation Accuracy of Convolutional Models
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Time Series
Frequency

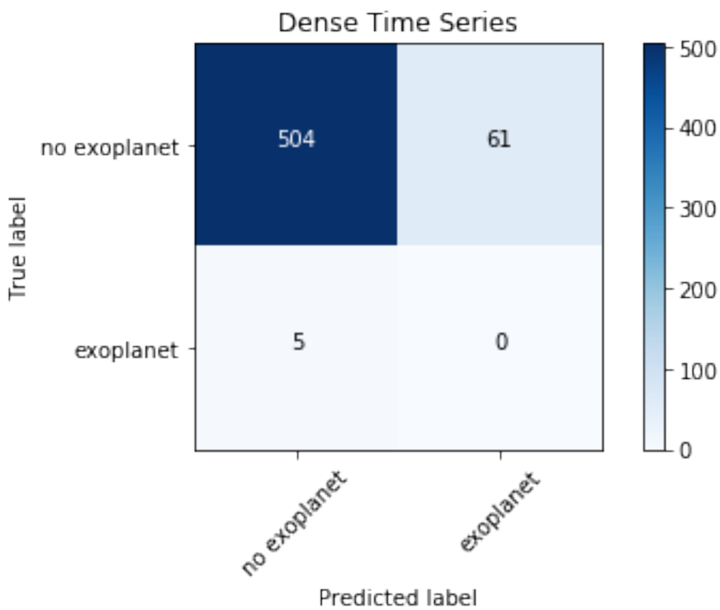
Based on this, the clear winner is the time series data, and it seems that the CNN outperforms the dense model.

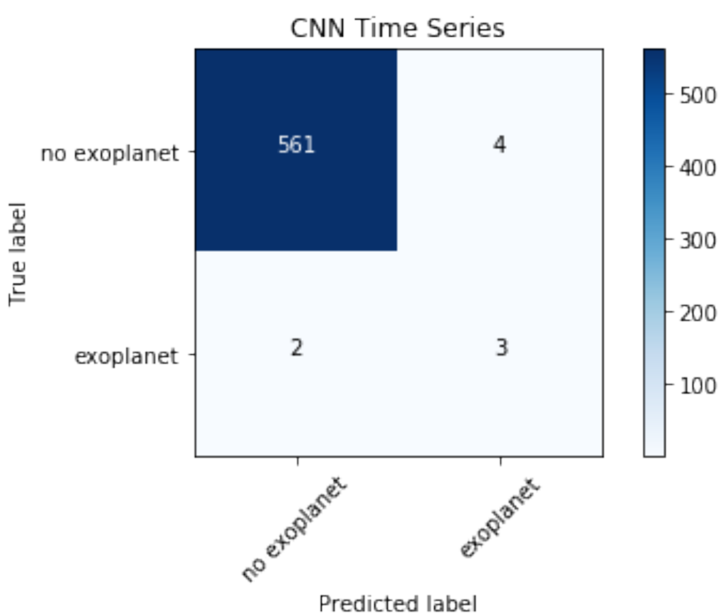
However, to get another take, let's see what the confusion plots look like.

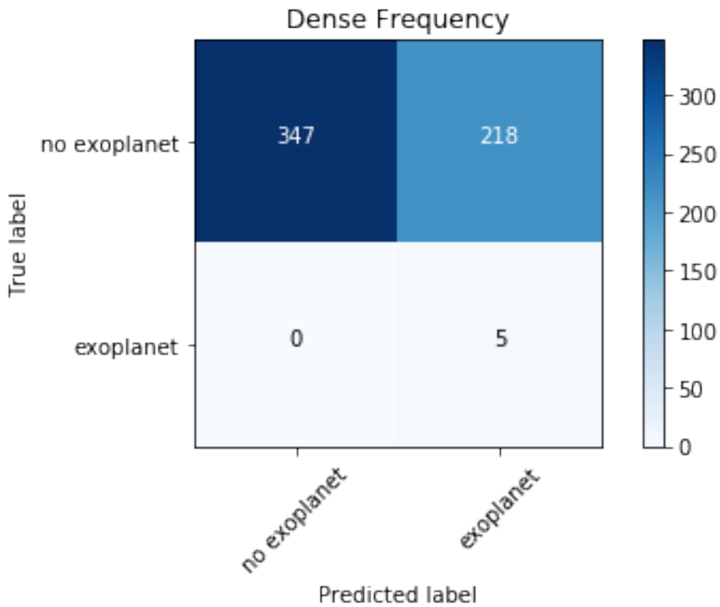
In [24]:
```
# add method of graphically showing correct and incorrect classificati
ons
from itertools import product
from sklearn.metrics import confusion_matrix

# confusion matrix plotting method taken from the sklearn docs
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

def visualize_accuracy(model, x, y, plot_title='Confusion Matrix'):
    y_hat = model.predict(x)

    cm = confusion_matrix(y, np.round(y_hat))
    plot_confusion_matrix(cm, ['no exoplanet', 'exoplanet'], title=plo
t_title)

def visualize_accuracy_new(y, y_hat, plot_title='Confusion Matrix'):
    cm = confusion_matrix(y, np.round(y_hat))
    plt.figure()
    plot_confusion_matrix(cm, ['no exoplanet', 'exoplanet'], title=plo
t_title)
    plt.plot()
```

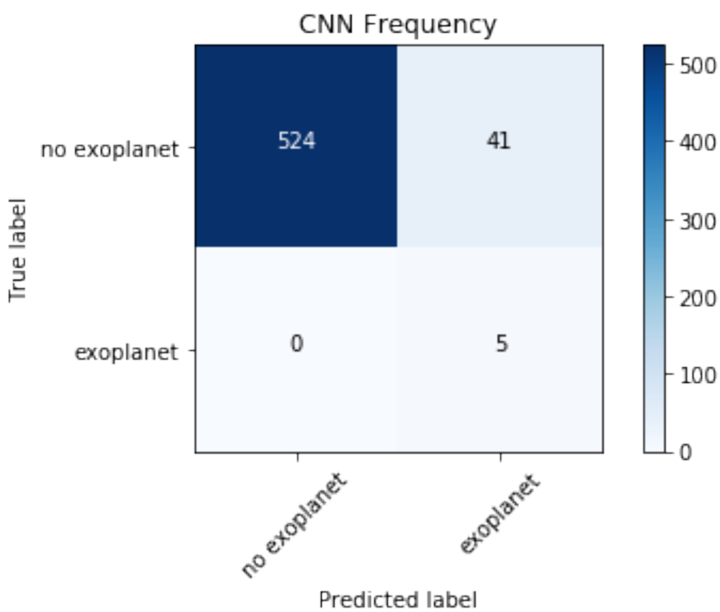
In [25]:
```
for model_name, y_hat in test_results.items():
    acc = np.sum(np.abs(y_hat[:, 0] - labels_test) < 0.1) / len(labels
_test)
    print('{} accuracy: {}'.format(model_name, acc))
    visualize_accuracy_new(labels_test, y_hat, model_name)
```

```
Dense Time Series accuracy: 0.8543859649122807
CNN Time Series accuracy: 0.9859649122807017
Dense Frequency accuracy: 0.5543859649122806
CNN Frequency accuracy: 0.8491228070175438
```






Dense Frequency
no exoplanet
exoplanet
True label
347
218
0
5
no exoplanet
exoplanet
Predicted label
300
250
200
150
100
50
0


CNN Frequency
no exoplanet
exoplanet
True label
524
41
0
5
no exoplanet
exoplanet
Predicted label
500
400
300
200
100
0

Clearly, the time series models are superior, as are the CNNs. Peak validation accuracy was with the time-series CNN at 0.986. The CNN frequency and Dense time series had similar accuracy at about 0.85. Unfortunately, I have to reject my hypothesis that the periodic nature of the data would play well with a frequency domain transform.

That said, one potential benefit of the frequency-domain models could be that they do not classify exoplanet stars as non-exoplanet stars. Such a model could be used with other methods to ensure no promising results go unseen. In this case, it may have been prudent to write a loss function that weighs heavier a misclassification as such, rather than marking no exoplanet as exoplanet.

For future improvements, it may als be worthwhile using a RNN on the time-series data as well to compare results. RNNs are very good at detecting patterns in time-series data, so this may be a good application.