Discovering exoplanets using Convolutional Neural Networks

Philippe van Amerongen

30 June 2018

Abstract

The NASA Kepler Space mission had as main objective to discover extrasolar planets by observing over 300,000 stars (Barentsen, 2018). This results in light curves of those 300,000 stars, which need to be analyzed individually for presence of exoplanets. With the upcoming TESS mission more than 2,000,000 stars will be observed (Ricker et al., 2010). In order to efficiently analyze all that data this thesis puts forth an approach to automate this analysis using a machine learning algorithm, a convolutional neural network. The objective of this thesis is to explore the possibilities and accuracy of such an algorithm in the field of exoplanet detection. There will be made use of a high leven neural network framework called Keras, which is built upon the TensorFlow library. The code made for this thesis can be found at `https://github.com/phicoder/exoplanet-detection`.

Acknowledgements

I would first like to thank my thesis advisor drs. Martin Vogelaar of the Kapteyn Astronomical Institute at the University of Groningen. He gave me all the freedom I desired in making this thesis. More importantly, when I needed feedback or some assistance in using the facilities available at the institute, he was always pleased to help.

I would also like to thank MSc Enrico Petrillo for sharing his knowledge about neural networks in the field of astronomy, enabling me to start writing this thesis with a clear goal in mind.

I would also like to acknowledge prof. dr. Inga Kamp of the Kapteyn Astronomical Insitute at the University of Groningen as the second reader of this thesis, showing much interest in this topic.

Philippe van Amerongen

Contents

1 Introduction

Exoplanets are planets that are situated outside of our solar system. There is an abundance of exoplanets, which follows from the fact that with the Kepler mission as of April 2018 there have been discovered over 2,200 confirmed exoplanets (Johnson, 2018), while having analyzed over 300,000 stars (Barentsen, 2018). It can be seen that this ratio is slightly more than 1 exoplanet per 150 stars. If we extrapolate this to our own galaxy, which contains about 2.5×10^{11} stars (Masetti, 2018), we could potentially have 1.7×10^{9} exoplanets in our galaxy. Some of those planets may be earth-like and may even contain life. The main focus of exoplanet discovery has become finding those type of earth-like planets. Analyzing that much data is a quite complicated problem, since the options to do this using algorithms are limited. Therefore, certain initiatives like 'Planet Hunters' have been set up, where a preselection of exoplanet candidates gets made by volunteers, through a peer reviewing system. However, the new TESS satellite is estimated to observe 2,000,000 stars (Ricker et al., 2010), which will take much more time to analyze than the Kepler mission. Moreover, analyzing this data manually is also prone to errors. For these reasons it might be interesting to look at machine learning algorithms in order to speed up this process.

A popular approach to machine learning is by using neural networks. Neural networks are a collection of neurons (nodes), organized in matrices, which slightly resembles the functioning of the brain, a set of connected neurons. In most cases of neural networks the process consists of matrix multiplications. By adjusting the strength of the connections the neural network are able to learn from the data that is given, called training data. Neural network learn to recognize features in that data. Therefore, a neural network can be classified as a pattern recognizing algorithm. The more complex the network is, the more complex features the network is able to recognize. However, more complex features require more complex networks and more data, which in turn requires more computing power. The mathematics and techniques behind neural networks can be quite complex. However, several programming frameworks have been made to develop neural networks more easily. In this paper, the high level neural network library Keras for Python3 is used (reference: `https://keras.io/`). This library makes use of the lower level TensforFlow (reference: `https://www.tensorflow.org/`) library. There are many alternatives, but Keras is very well-documented and provides all the functionality required in this thesis. Moreover, code can be ported easily to different machines and GPUs can be used without many extra proceedings.

For exoplanet detection neural networks could potentially be able to detect the specific features present in the light curves indicating an exoplanet is present. In the case that an automated pipeline could be set up to detect exoplanets from light curves, this would decrease the need for volunteers and increase efficiency and accuracy of finding exoplanets. In this paper the goal is to explore the use of neural networks in exoplanet detection and create a low threshold introduction for astronomers with an interest in this topic. This paper will try to give an answer to the question, is it possible to accurately detect exoplanets from light curves using convolutional neural networks? The hypothesis is that this should be possible, since light curves of stars with orbiting exoplanets show certain features, which a neural network should be able to pick up.

2 Theory

2.1 Exoplanet detection

There are several methods to detect exoplanets of which the transit method (Deeg, 1998) is the most prominent. The focus of this paper will be on the transit method. The transit method looks at the brightness of a star at regular intervals. The result is a brightness versus the time plot, such a plot is called a light curve. An example of such a plot can be seen in figure 1.

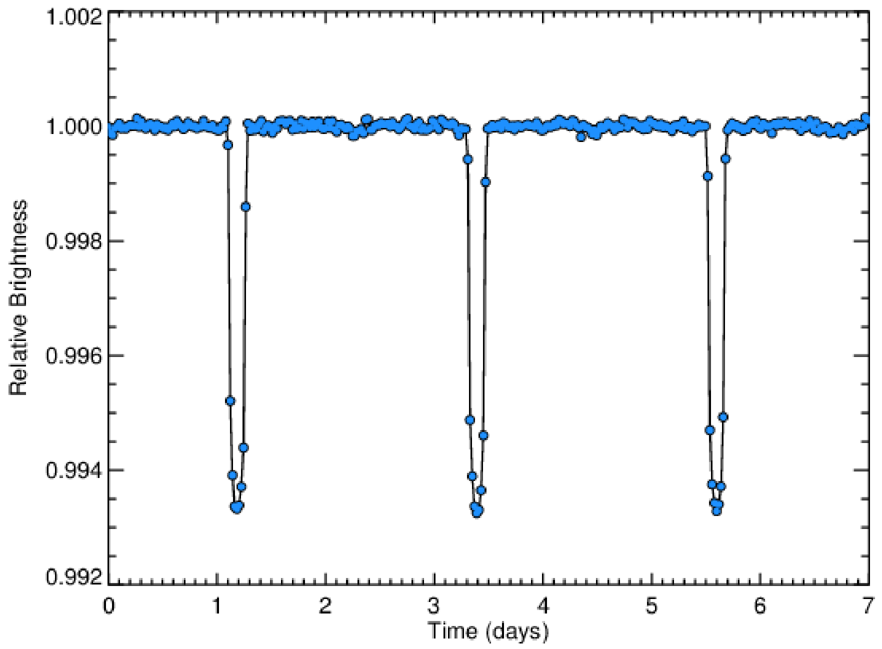


Figure 1: An idealized representation of a light curve, from (Vanderburg, 2017). It contains a transiting planet, which is characterized by the repeating dips in brightness.

If an exoplanet would transit the star, that is being in between the observer and the host star, the brightness shows a dip. The relative size of this dip only depends on the ratio between the frontal surfaces of the host star and the planet, assuming we observe from a distance much larger than the radius of the orbit of the planet. The brightness decreases roughly as much as the amount that the planet covers the surface of the host star. We can formulate this as $\Delta = \frac{r^2}{R^2}$, where Δ is the brightness decrease, r is the planet radius and R is the host star radius. If we take as an example the earth transiting the sun seen from another solar system we get $\Delta = \frac{(6\times10^3)^2}{(7\times10^5)^2} = 7 \times 10^{-5} = 0.007\%$. From this we can see that in order to detect exoplanets, especially earth-sized exoplanets, we need very precise measurements of the brightness of the star. This is the main reason why the popularity of the transit method increased with time; the technology has become more and more available to do these kind of precise measurements. With the launch of the Kepler mission, the use of this method vastly overtook other techniques.

The second most used technique is the radial velocity method. When an object orbits a star, the star also has an orbit. This implies that the star has a radial velocity. This velocity can be obtained by using the Doppler effect. Using high precision spectroscopes a shift in spectral lines of the star can be noticed and therefore its radial velocity can be deduced. This method is sometimes used to confirm exoplanet candidates found using the transit method.

There are two main disadvantages about the transit method. The first disadvantage is that the planet has to have an orbit aligned in such a way that the planet can be positioned between the host star and the earth. The probability for this to happen, assuming an eccentricity of 1, we get $P = \frac{R}{d}$, where d is the radius of the orbit and R is the radius of the host star, ignoring the size of the planet. Again, taking as an example the earth orbiting the sun we get $P = \frac{7\times10^5}{1.5\times10^8} = 4.7 \times 10^{-3} = 0.47\%$. A very small change, which means that it is very likely to see no exoplanets around the host star, but in reality there are. This disadvantage is counteracted with the possibility to monitor many stars at the same time.

The second disadvantage is that, because of the tiny dip in the light curve, it is easy to get false positives. A dip does not necessarily mean that an exoplanet is transiting. According to Santerne et al. (2012) the false positive rate for Kepler close-in giant candidates is 35%. For this reason, many exoplanet candidates require further inspection, with for example the radial velocity method, in order to confirm if the observed anomaly is truly an exoplanet.

2.2 Convolutional Neural Networks

Before starting to explain what convolutional neural networks are, I would like to refer you to the following article: `https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/`.

Neural networks are, in its fundament, a collection of weights, usually ordered in layers. The first layer is the input layer. This layer can receive a series of numerical values, the places of the weights are also called nodes, and in the case of the input layer we talk about input nodes. These values then get multiplied by the weights and get passed to the next layer, where they get multiplied by those weights. This goes on until the end is reached, the output layer. The output is again a set of numerical values, which can be interpreted by the user.

Lets start with the simplest form of a neural network. This type is called a perceptron, which consists of n input nodes and one output node. A sketch of a perceptron can be seen in Figure 2. Lets say we have the input values $X = [x_1, x_2, ..., x_n]^T$, which means we have n inputs and thus n weights. Now we have the weights $W = [w_1, w_2, ..., w_n]$. The output is the dot product of these two matrices, $z = X \cdot W = x_1 \times w_1 + x_2 \times w_2 + ... + x_n \times w_n$. The data is fed forward into the network, this is where the name feed forward network comes from. It can be seen that the basics of a neural network is nothing more than simple linear operations, therefore it is quite easy to compute by a computer. In order to make the data even more useful, by for example normalizing it, we have a last step. The value z goes through an activation function, which produces the final output a. This activation function can be any chosen function, but the most widely used function is the ReLU, which can be written as $a = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$. Normalization of output is very common practice, in order to do that for example a sigmoid function can be used. The sigmoid function is written as $a = \frac{1}{1+e^{-z}}$.

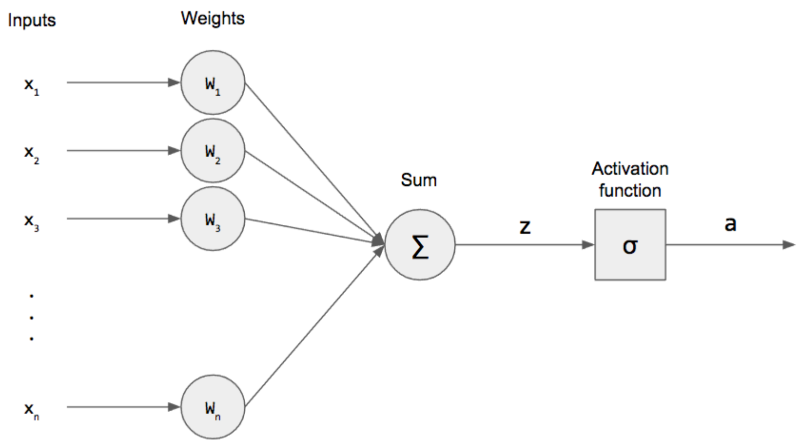


Figure 2: Interpretation of a perceptron, from Deshpande (2017). The inputs get multiplied with each assigned weight, the results get added, so we get one numerical output z. This output then goes through an activation function, which gives the final useful result a.

In order for the neural network to give useful output, the weights have to be changed for the output to match the expected value, the neural network has to be trained. This usually happens through a process called *back propagation*, for which we need labeled data. Labeled data is a set of numerical inputs for which the output is known. Before training we need to initialize the weights, this can be done by plugging in random values, but it is possible to have more tactical initialization values. For example; it is possible to use the weights of another, already trained network. Now the network has been setup, the training can start. The labeled data goes through the network after which the output is compared to the expected output. For this comparison a squared error can be used, but there are many different mathematical error calculation methods, the function used to calculate the error is called a *loss function.* For binary problems, one best uses a binary cross entropy loss function, which is a cross entropy loss function (De Boer et al., 2005) tailored for binary problems. This method is provided by the Keras library. Now the error is known, this can be used to adjust the weights. It is desired to minimize the error, using a so-called *optimizer*. There are many optimizers, but the Adam optimizer has proven to be a very efficient optimizer for most problems (Kingma and Ba, 2014). This optimizer is also provided by the Keras library. If we have a weight w_i, the new weight becomes $w_i = w_i(1 - \delta)$, where δ is the given error. In order to back propagate the error to previous layer, a backwardpass is required which can be seen in Figure 3.

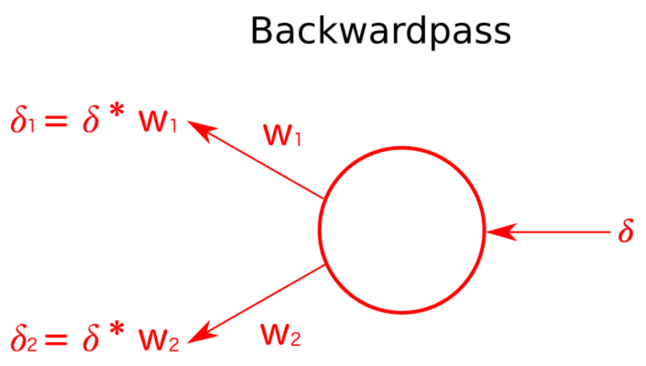


Figure 3: Abstraction of a backwardpass, for a two input and one output node, used in back propagation. The given error δ is multiplied with the weights, which give the errors for those weights δ_1, δ_2, which can be used to adjust those weights respectively.

In order to not let the network stagnate, converging to just one solution that may not be most appropriate, dropout layers can be added. Moreover, these layers can prevent over-fitting, which will be discussed later in this paper. A dropout layer deactivates randomly selected connections during the training period. This will cause the nodes to be less dependent on each other, since clusters of nodes will be trained to work by themselves. When finally activating all the connections, the network consists of clusters of nodes who have been trained together. The final result is a more robust network that should not be able to over-fit easily. (Srivastava et al., 2014)

For more complex problems, a bigger neural network is required. A neural network is very modular, so extra layers can be added and more output nodes can be used. Take as example a handwritten digit classifier, as input a flattened matrix (converting a $m \times p$ matrix to a $n \times 1$ matrix, where $n = m \times p$) of grey values representing the image is given. There will be some layers using the ReLU activation function and a dropout layer. At the end the output gets normalized by a layer with a sigmoid activation function. Since neural networks work best using binary output, called one hot encoding, the output would be 10 nodes, where node 1 represents the number 0, node 2 represents the number 1, etcetera. The output will then be 10 numbers each between 0 and 1, the highest number (probability) is the best guess of the network.

The efficiency of the handwritten digit classifier can be greatly increased by using a convolutional neural network. A convolutional neural network can take as input a 2-dimensional (or a higher dimensional) matrix instead of an $n \times 1$

matrix. Since the above example, as well as the network in this thesis, have as input an image, the input will be 2-dimensional which implies a convolutional neural network can be used. A convolutional neural network usually makes use of two techniques, called pooling and convolutions. At the end of the network the matrix is flattened to a 1-dimensional shape and a regular feed foward neural network can be applied. *Pooling* is a process with which the size of the 2-dimensional input can be drastically reduced depending on the pooling size. The image is divided into pieces of the pooling size and for every pool (collection of pixels) a new output pixel is made. The value for the output pixel can be the minimum, average or maximum of the pool, called minpooling, average pooling and maxpooling respectively. If we would take a pooling size of 2×2, this means that the resulting image would be 4 times as small as the input, because it is halved vertically and horizontally. The part that has the intelligence of the network are the convolutional layers, in which convolutions happen. For a *convolution* a piece of the image is taken, say a $k \times k$ fragment, the weights are applied and a single output is generated. This process is done scanning the entire image, moving the fragment pixel per pixel across the image. This process reduces the image size on each side by $\lceil \frac{k}{2} \rceil - 1$ with $k > 2$. The ouput images are called feature maps, since they are supposed to represent a certain feature that the convolutional neural network learned. A representation of a convolutional neural network can be seen in Figure 4.

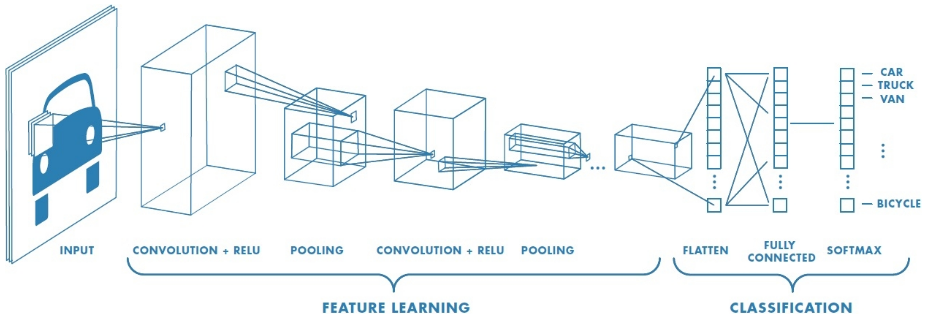


Figure 4: Interpretation of a convolutional neural network by MathWorks (2018).

Designing a CNN is a non-trivial task. However, one can apply certain rules and techniques. When setting up a CNN, the first layer will be a convolutional layer. A convolutional layer generates feature maps which are able to detect certain patterns. Depending on the number of features and granularity of the features that need to be detected, the designer can adjust the number of feature maps and their size. If the features are generally small, the designer can choose for small maps, e.g. 4x4 maps. If there are a lot of different features to be detected, the designer can choose to let the layer generate a lot of feature maps. The more of these maps there are and the bigger they are, the more computing power they will require, this can also weigh into making design choices. If the input is very large, the choice can be made to add some pooling layers. However, with these layers detail falls away, which can make the network less accurate. In the end a 1-dimensional matrix is preferred, so the designer can apply a final feed forward neural network. This last part is mostly used for regularization and normalization purposes. However, if the task is very complex, the designer can append a more complex feed forward network. Depending on the amount of training data the designer should make the network more or less complex; less data, less complex, otherwise the network will not be able to accurately learn features. A common issue with neural networks is *over-fitting*, which means that the neural network memorizes the training data. This causes the neural network to get 100% accuracy on the training data, but is not necessarily good at recognizing the features that the network is required to detect. In order to reduce over-fitting a dropout layer can be added to the network.

2.3 Programming a neural network

This section will show how to get started with programming a neural network by making one using the Keras library. First off a little table with all the layers and their descriptions.

Layer	Description
Dense	A fully connected layer, meaning that all the nodes are connected to all the nodes in the next layer. In Keras the first input parameter (x) indicates the number of nodes, like so `Dense(`x`)`
Conv2D	A 2-dimensional convolutional layer. This layer makes x feature maps of the images of size $n \times m$. In Keras the syntax is used `Conv2D(`x`, (`n, m`))`
MaxPooling2D	A 2-dimensional pooling layer. A pooling layer takes $n \times m$ squares of pixels (pools) and transforms them into one pixel. In the case of a maxpooling layer, the pixel with the highest value in the pool determines the value of the single output pixel. Keras uses the following syntax `MaxPooling2D(poolsize = (`n, m`))`
Flatten	A layer that transforms a multidimensional matrix into a 1-dimensional matrix.
Dropout	A layer that deactivates connections between layers randomly with probability p during training. During prediction all the connections are activated again. This layer can make the network more robust and prevent over-fitting. Keras uses the following syntax `Dropout(`p`)`

At the end of this section the simple network we are going to make is compared to the network that was made for this thesis. We will program a simple perceptron which can learn the rules of linear regression. We will be able to let the network 'understand' relations in the form of

condition	output
$ax \leq y$	0 or 1
$ax > y$	0 or 1

It can be seen that this is a binary classification problem. Also, the advantage of a neural network is immediately visible; we can solve a general problem without having to edit the code (if we desire different results, we just need to train on different data).

To start programming the network, we first import TensorFlow and NumPy and set up the model of a perceptron:

```python
from tensorflow import keras
import numpy as np

# Set up a sequential model, with as input an array representing the
# layers of the model
model = keras.Sequential([
    keras.layers.Dense(2),
    keras.layers.Dense(1, activation = 'sigmoid')
])
```

As we can see the above model has two input nodes and one output node. A dense layer represents a fully connected layer, meaning that all the nodes in the layer are connected to all the nodes in the next layer. In the final layer we use a sigmoid activation function in order to normalize the output, since we have a binary classification problem we want output values between 0 and 1. In this case a sequential model is chosen, since it is a simple feed forward network. One can also use the functional API that Keras provides to make more complex models such as "multi-output models, directed acyclic graphs, or models with shared layers" (`https://keras.io/getting-started/functional-api-guide/`).

Neural networks made with Keras run on highly optimized C/C++ code, therefore networks first need to be compiled. One can do that as follows:

```python
model.compile(
    optimizer = 'Adam',
    loss = 'binary_crossentropy',
    metrics = ['accuracy']
)
```

The compiler can take several input parameters, but the ones above are most important. As has been said about *optimizers*, the Adam optimizer seems to work best, so we implement that one by giving it to the compiler as an input parameter. Since this is a binary classification problem, we can use the binary cross entropy *loss* function. The metrics are the output the model will print to the standard output when training. The two most interesting *metrics* are loss and accuracy and since loss is provided by default, only accuracy has to be put in the array.

Now the model has been compiled, we can start training the network. For this we first need to have training data and labels. In this example, the goal is to teach the network the following ruleset

condition	output
$x > y$	1
$x \leq y$	0

Since we have an equation, we can generate training data with NumPy using the following code

```python
# generate 10000 training samples of form (x, y)
data = np.random.rand(10000, 2)

# generate the training data labels
labels = np.array([])
for n in data:
    if n[0] > n[1]: # if x > y, output a 1
        labels = np.append(labels, [1])
    else:
        labels = np.append(labels, [0])
```

The more training data, the better, but 10,000 samples seems more than reasonable for this problem.

The following line of code will train the perceptron

```python
model.fit(data, labels,
    epochs = 8,
    batch_size = 20,
    validation_split = 0.1
)
```

The three named input parameters are very important, because they have a big impact on training performance. *Epochs* represent the number of iterations the network has to train on the same data, each time shuffling this data. The more epochs, the longer the training phase will take, but each epoch the network will get more accurate. However, too many epochs can result in over-fitting. Since we are solving a very trivial problem, we actually want the network to over-fit. However, after some epochs the network will stop improving its accuracy. In the case of this example that seems to happen after about 8 epochs. Complex problems may require hundreds of epochs. *Batch size* indicates on how many samples the network should predict before it backpropagates the cumulative error. A bigger batch size means less processing power is required, but also reduces the accuracy. In order to keep the training time reasonable for this trivial example problem, a batch size of 20 is chosen. The *validation split* parameter can tell the model on how much of the training data the network should not train, but predict only. The number represents the fraction of the training data used as validation data. The accuracy and loss results from these predictions give a good indication if a network is learning well, one can for example detect over-fitting or see the progress the network has made. The preferred result is when the accuracy gradually (over several epochs) increases and the loss gradually decreases. This is an indication that the network is learning steadily. Just as important is that the validation loss and accuracy follow the training loss and accuracy, perhaps with some delay. This is an indication that the network is not over-fitting.

The output looks like the following

```
Train on 9000 samples, validate on 1000 samples
Epoch 1/8
9000/9000 [==============================] - 1s 92us/step - loss:
    0.7154 - acc: 0.4287 - val_loss: 0.6405 - val_acc: 0.7140
Epoch 2/8
9000/9000 [==============================] - 1s 66us/step - loss:
    0.5711 - acc: 0.8072 - val_loss: 0.5035 - val_acc: 0.8680
Epoch 3/8
9000/9000 [==============================] - 1s 66us/step - loss:
    0.4422 - acc: 0.8748 - val_loss: 0.3877 - val_acc: 0.9080
Epoch 4/8
9000/9000 [==============================] - 1s 63us/step - loss:
    0.3433 - acc: 0.9088 - val_loss: 0.3028 - val_acc: 0.9290
Epoch 5/8
9000/9000 [==============================] - 1s 63us/step - loss:
    0.2719 - acc: 0.9368 - val_loss: 0.2409 - val_acc: 0.9560
Epoch 6/8
9000/9000 [==============================] - 1s 65us/step - loss:
    0.2204 - acc: 0.9578 - val_loss: 0.1957 - val_acc: 0.9730
Epoch 7/8
9000/9000 [==============================] - 1s 64us/step - loss:
    0.1833 - acc: 0.9739 - val_loss: 0.1646 - val_acc: 0.9800
Epoch 8/8
9000/9000 [==============================] - 1s 63us/step - loss:
    0.1567 - acc: 0.9882 - val_loss: 0.1405 - val_acc: 0.9950
```

We can see that the model is using 1000 samples (10%) as validation data. Also we see that the model did train for 8 epochs. Furthermore, as well as the validation accuracy as the training accuracy gradually increase. The validation and training loss gradually decrease. The network's final validation accuracy is 99.5%, which means that it is wrong in 1 in 200 cases. This is acceptable for this example since not all the possible techniques to improve the model have been applied in order to keep it more accessible. A completely optimized network would be able to train faster and would be able to get near 100% validation accuracy.

Now the network has been trained, it can be used to predict on data. We can do this as follows

```python
# generate some prediction data manually
predication_data = np.array([
    [0.35, 0.71], # -> 0
    [0.58, 0.22], # -> 1
    [0.98, 0.80], # -> 1
    [0.55, 0.59], # -> 0, stress test, how close can values be?
    [10, 20]      # -> 0, network has not been trained on numbers
                  #       bigger than 1.
])

# Lets predict
result = model.predict(predication_data)

print(result)
```

The result is the following

```
[[0.02836107]
 [0.9758824 ]
 [0.85824233]
 [0.41625893]
 [0.        ]]
```

We can see that if we would round all the probabilities, they are all correct. The fourth entry is quite undecided, since it is hitting the limits of the network's accuracy. The last entry shows why neural networks can be so useful. The last entry consists of numbers bigger than 1, which the network has never been trained on, but it still can give a certain and correct result.

If we would like to reuse the (trained) network, we can save its weights or the entire trained model. Saving the entire model usually takes up much more disk space, but it saves the following things:

- Architecture of the model, allowing to re-create the model
- Weights of the model
- Training configuration (loss, optimizer)
- State of the optimizer, allowing to resume training exactly where you left off

The network can be saved as follows

```
# Saves the entire model
model.save('model.hdf5')
# or
# Saves the weights of the model
model.save_weights('model.hdf5')
```

The model can be loaded using one of the following lines

```
# Load an entire model
model.load('model.hdf5')
# or
# Load the weights of a model
model.load_weights('model.hdf5')
```

Now we have taken our first steps in programming a neural network in Python3 using Keras, we will have a look at the final model used in this thesis. The code for the final model is the following

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(5, (50, 50), activation = 'relu',
        input_shape = (
            self.dimensions[0],
            self.dimensions[1],
            1
        )
    ),
    tf.keras.layers.MaxPooling2D(pool_size = (2, 2)),
    tf.keras.layers.Conv2D(10, (4, 4), activation = 'relu'),
    tf.keras.layers.MaxPooling2D(pool_size = (2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation = 'relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(1, activation = 'sigmoid'),
])

model.compile(optimizer = 'adam',
                loss = 'binary_crossentropy',
                metrics = ['accuracy'])
```

We can immediately see similarities between the simple example network and this network. Moreover, the compiler is exactly the same. The difference lies in the layers of the sequential model. We can directly see that it is a convolutional neural network because of the presence of convolutional layers. Also, maxpooling, flattening, dense layers and a dropout have been used. In order to make the network more accurate, many ReLU activation functions have been used. At the end, again, a sigmoid activation function is used for normalization.

2.4 Neural networks on exoplanet detection

With the vast amounts of light curves and detected exoplanets, applying a neural network to this problem could be very useful. The light curves that represent a star which hosts an exoplanet can be used as training data. There are several ways to process light curves using neural networks. One can for example bin the data points to a predefined number of points and feed this data into the network. Research into this technique has been done by Shallue and Vanderburg (2018), producing useful results. Other research into exoplanet detection using neural networks has, to date, not been done. In this thesis a different approach is used; convolutional neural network will be used to try to spot potential exoplanets. The light curve can be transformed into an image of a predefined size, after which the convolutional neural network processes it, resulting a possible detection of an exoplanet. The advantage of this technique is that the neural network will always be presented with all the data, no binning is needed. On the other hand, same-size images of light curves will need to be generated, since the input size of the convolutional neural network is fixed. If a lot of data is present, with big ranges in brightness and time, details will become less visible, since more data is compressed in a smaller frame. This could cause the network to overlook possible exoplanet candidates.

3 Methods

3.1 Data pipeline

In this thesis the MAST database, (NASA and STScI, 2018), was used to obtain the light curves. The database contains light curves created by the Kepler space mission. In order to automate the data retrieval process a Python class for MAST was made, which can be found in the public github repository `https://github.com/phicoder/exoplanet-detection`, specifically the file `MAST.py` . The class is able to retrieve data from all the available data sets provided by MAST as well as fetching the light curves given a Kepler ID. This last functionality was used by the program to fetch all the desired light curves, these are FITS files (with file extension `.fits`).

Once the FITS files were retrieved, the data was processed. The FITS files contain a column for time and a column for adjusted flux. These points were then plotted into a graph as small white dots on a black background, an example can been seen in figure 5. These graphs have equal width and height and each of them has the same dimensions. The images having the same dimensions is a requirement, since the neural network has a predetermined and fixed amount of input nodes.

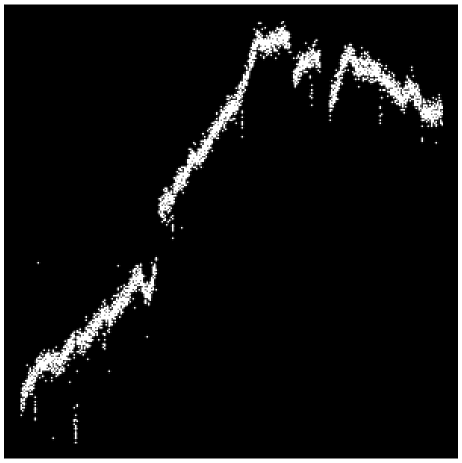


Figure 5: An image of a light curve as used by the neural network. This particular light curve contains regular dips in brightness, which is the core feature sought after in light curves in this paper. This feature can signify an exoplanet. The size of the image is 300x300 pixels.

3.2 Training

In order to make the neural network learn to differentiate between light curves of star systems with exoplanets from star systems without exoplanets the neural network needs to be trained. The neural network was trained using light curves with (positive) and without exoplanet presence (negative). This data was selected by hand, so that the positive training data completely consist of light curves with regular dips in brightness, as seen in figure 5. The MAST database has a data set containing all the Kepler IDs of star systems containing confirmed exoplanets. These IDs were then used to fetch the appropriate light curve data. After the data was processed and hand selected, the Neural Network was trained by feeding all the images as 2-dimensional matrices into the CNN.

The errors were then back propagated to minimize the loss. After the training was finished, the neural network was ready to be used to make predictions for exoplanet presence.

3.3 Convolutional Neural Network

After the data pipeline had been set up, the Neural Network was programmed. First a basic CNN was made for testing purposes. Since this is a binary classification problem, the binary cross entropy loss function was used. Also the Adam optimizer was implemented, mainly due to its efficiency. Both of these methods are provided by the Keras library. Using training data several iterations to the neural network were made to see how the CNN behaved. After several revisions of the network, a final version of the network was made. The final version of the neural has the following flow:

300x300 image

⇓

5@50x50 feature maps (convolution)

⇓

2x2 maxpooling

⇓

10@4x4 feature maps (convolution)

⇓

2x2 maxpooling

⇓

flatten to 1d matrix

⇓

dense layer 128 nodes, ReLU

⇓

dropout of 0.1

⇓

dense layer 1 node, Sigmoid

The convergence to a design as above can be explained. There was not a lot of training data available and there was a lack of experience, so the setup of the network was kept as simple as possible. In order to reduce the computing power needed during the test phase, several maxpooling layers were applied. In general, the number of features that were tried to be detected was not plenty, therefore the first convolutional layer was made using just 5 feature maps. The size of these features varies a lot, so a relatively large size was chosen. For the second convolutional layer much smaller feature maps were chosen, mostly because of the pooling. Since the maps were chosen to be smaller in size, the choice was made to use more feature maps. One dense layer was added to add slightly more intelligence to the network using the ReLU activation function due to its good reputation. Since over-fitting was found to be an issue, as can be seen in 6, a dropout was added at the end of the network. The last layer is the node that makes the verdict if an exoplanet is present, or more accurately, features of the training data were recognized. This output is binary, exoplanet present or not, so the sigmoid activation function was chosen.

3.4 Predicting

In order to make predictions, the `./data` directory was filled with images of exoplanets. For the retrieval of the light curve data a Python script was made, which can be found in the github repository under the name `data_import.py`. The light curves per star for every time period were put each in its own subdirectory in the data directory. Once the data directory was filled up, the `./main` script was executed. For every star each of the time periods were analyzed by the neural network. For the stars where at least one of the light curves reached a certain threshold, a new line in a log file was created in order to keep the log file as short as possible. After the network had processed all the light curves, the log file was used to check for potential exoplanets.

3.5 Computing

In the testing phase, while developing the network, a 2.6 GHz Intel Core i7 QuadCore CPU was used. Training the several networks took between 10-180 minutes. The final network took 180 minutes to train and 120 minutes to perform the predictions. This was done to demonstrate that it is possible to perform these kind of tasks on high-end consumer computers, so it is not necessary to have continuous access to a high-performance (super)computer. Once having finished the testing phase, the network was also trained using the norma4 computer cluster at the Kapteyn Astronomical Institute at the University of Groningen. Using this machine the training and predicting using the final network took significantly less time, 80 and 20 minutes respectively. Even though this machine performed much better, the training time can be sped up significantly by using a GPU. The performance increase of using a GPU could be in the order of a magnitude.

4 Results

4.1 Training data

The training data was obtained using a modification on the `data_import.py` script. This set contained the data of a few hundred light curves with and without exoplanet presence. This data was then cast into images containing the data as black dots on a white background. However, the neural network could not increase its accuracy and decrease its loss. The reason for this could be that, since most of the image is white (encoded as a 1), the weights need to be tuned down to near 0 in order to get a 0 from the output node. Most of the images should classify as a 0, since most real images will not contain exoplanets, and therefore this approach does not work. This was fixed by changing the preprocessing stage, so that the generated images would be white data points on a black background. Now a new problem arose, the training data with exoplanets (positive) seemed too random and no specific patterns could be recognized. Therefore, the concession was made to focus on a specific feature for exoplanets that just some of the graphs contained; very visible regular dips in brightness. The positive training data had to be selected by hand for these features. After a thorough selection, 228 images remained for the positive data. After this a random selection of 685 images of light curves not containing exoplanets was added. No specific data reduction was applied on the images, which also contributed to having to make the decision to just detect the regular dip features.

4.2 Data

To save disk space and reduce prediction time in the test phase, the light curves of 3,000 stars were used, which is just 1% of the total number of stars observed by the Kepler mission. A selection of light curve images can be seen in figure 6. It can be seen that light curve data is not constant; many features are present. In order to let the neural network give useful results, there has been chosen to only look for dip features as seen in figure 5.

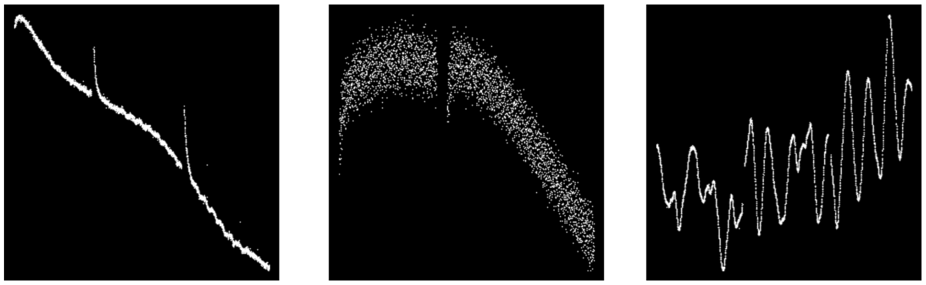


Figure 6: A random selection of three light curves from three different stars. One can see how erratic this light curves can be. On the *left image* there are two peaks, this feature may be explained by recalibration of the Kepler telescope. On the *middle image* a gap can be seen. This is due to downtime of the Kepler telescope, maybe because the star light was blocked by the earth. In the most *right image* a very erratic pattern can be seen, this is not caused by an exoplanet, but could be a binary star.

4.3 Neural Network training

Training the neural network was a big part of this thesis. Many iterations of the neural network were made and for every iteration the network had to be trained in order to see its effectiveness. The training sessions were done using 913 images doing 8 epochs, 8 times training on the training set, and using 10% of the training data as validation data. This validation data is data on which the network did not train, but just did predictions. This was used to check if the network learned features or if it was over-fitting on the training data. Training the network on 913 images for 8 epochs roughly took between 10-180 minutes, depending on the complexity of the neural network. The loss of the final version can been seen in 7. A smooth descent in both validation and training loss can be seen. This most likely implies no over-fitting. The accuracy on the training data compared to the validation data can be seen in figure 8. The accuracy of both the training data and validation data raised gradually in roughly the same pace, meaning that the network is properly and steadily learning to recognize features. After having obtained these results, the designing of the neural network was concluded. The downside of this model is that it took by far the most time to train, but the other models showed signs of over-fitting. Therefore, this model was chosen above the others. Accordingly, the produced model was used for doing predictions. The results of other versions of the network can be found in Appendix A.

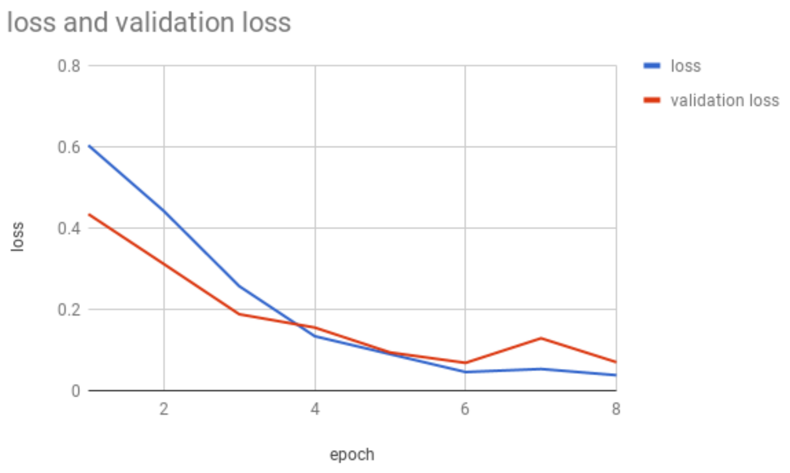


Figure 7: The training loss and validation loss per epoch for the final version of the model. It can be noted that the validation and training loss decrease gradually at roughly the same pace. This is a sign that the neural network is not over-fitting and gradually learning to recognize features.

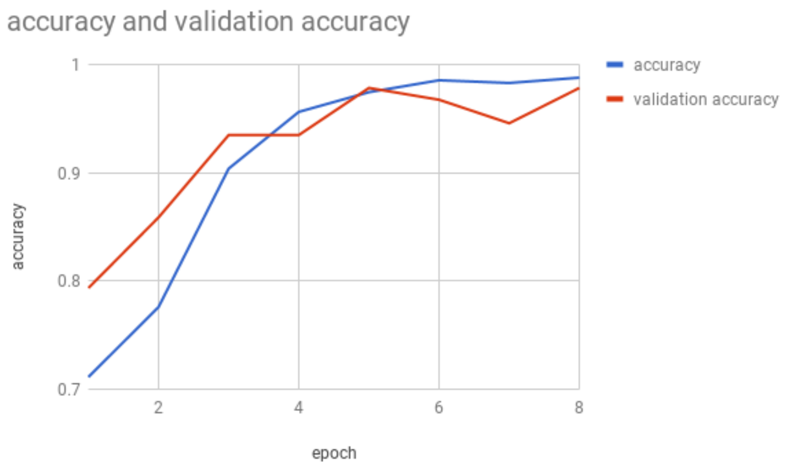


Figure 8: The training accuracy and validation accuracy per epoch for the final version of the model. It can be seen that the training and validation accuracy gradually increase at roughly the same pace. This is a sign that the neural network is not over-fitting and gradually learning to recognize features. In the first epoch the validation accuracy is higher than the training accuracy, this is simply coincidence, later on the accuracies converge towards each other.

4.4 Neural Network predictions

The final model was used to do predictions on 3,000 stars, with each roughly 18 quarters, so 54,000 images of light curves. The network took 120 minutes to analyze all the images on the i7 processor. However, this process could have been sped up many times using a GPU. The program was made so that if the network was more than 90% sure about that dips in the light curve appeared, it would count it as a positive. The KIC ID got written in the log file if there was at least one positive in all the light curves of the star. These are the first few lines of the log file:

```
KIC ID,surpassed threshold,total,ratio
2969216,5,18,0.2777777778
2707758,1,18,0.05555555556
2576692,13,18,0.7222222222
3113351,1,18,0.05555555556
```

The complete log file can be found and downloaded at `https://github.com/phicoder/exoplanet-detection/blob/master/logs/log.csv`. After the neural network finished predicting, the log file consisted of 1,338 lines, which means that for 1,338 stars at least one image superseded the 90% threshold. This log file can now be used to find stars with many images that superseded the threshold. The best candidate found is KIC ID 2708156, with a score of 16/18, which appears to be an eclipsing binary. In figure 9 it becomes clear why the network labeled most of its images as positive, each of the images contains very significant dips in brightness.

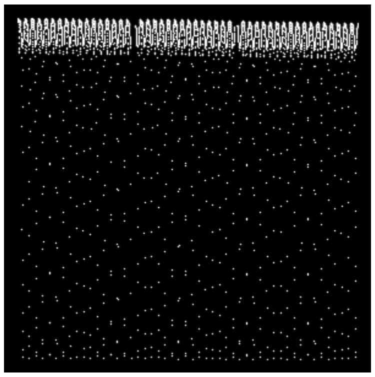


Figure 9: Quarter 5 of KIC ID 2708156. This star is classified as an eclipsing binary and was detected by the convolutional neural network. Since the neural network mainly focused on detecting dips in light curves this can be considered a positive result.

Looking through the top results it appears that most results are eclipsing binaries and just some of them contain exoplanet candidates. This is due to the fact that the neural network was trained on recognizing dips in light curves. The network has proven to recognize those features extremely well, even if the dips were barely noticeable. However, sometimes the network accepts less obvious light curves, an example is KIC ID 2718252. This star is not flagged as to have any anomalies and no very clear dips were observed in the light curves. Two interesting stars were found in the top 25 results; 2163434 and 2854994. These stars are not flagged to have any anomalies, but they do show dips in the light curves as shown in figure 10.

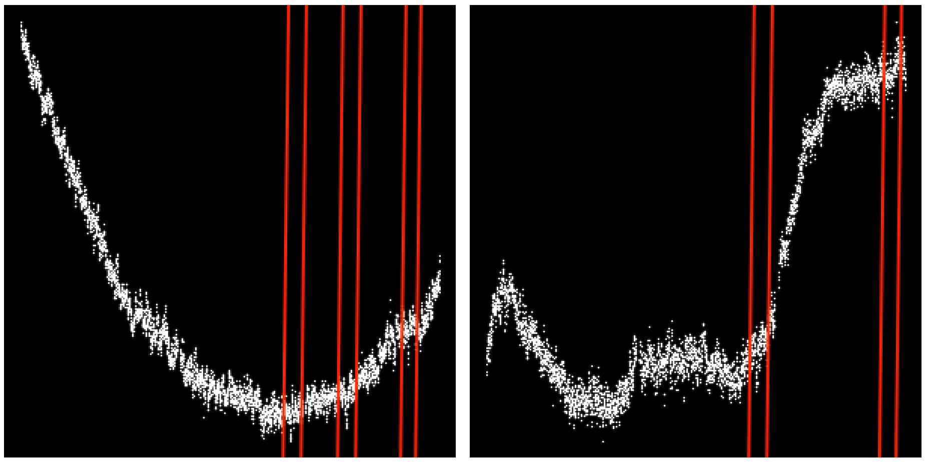


Figure 10: On the left KIC ID 2163434 and on the right KIC ID 2854994. These were both present in the top 25 made by the convolutional neural network. In both graphs dips in intensity can be noticed, which probably triggered the network.The most prominent dips are marked between red lines. Neither of these stars are flagged having an anomaly, e.g. an exoplanet.

It has become clear that the network is able to detect features, in this case dips in light curves, very well. Even with a selection of 3,000 stars, or 1% of the total number of stars analyzed by the Kepler mission, some interesting results appear. Unfortunately it mostly detected eclipsing binaries, but this can be explained through the lack of preprocessing of the light curve data and inadequate training data. The network did make few mistakes, but it is not known how many positives it has skipped. All in all, the technology looks promising. As already said by Miller (1993) neural networks could be useful in astronomy in the fields of adaptive telescope optics, object classification and matching, and detector event filtering, satellite system applications. This thesis could be seen as an indication that in the category of object classification, Miller might be correct. The expectations are that by using neural networks in astronomy many tasks in those fields could be sped up and accuracy could be increased due to less manual work.

4.5 Analysis of two stars

Two stars have been found using the neural network which have not been classified as anything, but do show some interesting features. In figure 10 we can see that both light curves show some small dips in brightness. *KIC ID 2163434* In Appendix C we can see all the available light curves for this star. In quarter 7, 9 and 10 dips are visible in the light curves. They are most prominent in the lowest part of the light curve in quarter 9. Since the dips are visible in so many places, it was excluded that the features are caused by measurement errors. The dips are very small in size, which usually does not occur with eclipsing binaries.

Furthermore, the dips do not have a specific recurring pattern. Since there is no pattern visible, but there are multiple dips which are small and different in size, it could be multiple exoplanet candidates. However, there is a lack of light curves, so the result is not very significant. Nonetheless, it may be interesting to better investigate this star.

KIC ID 2854994
In Appendix D we can see all the available light curves for this star. In quarter 0, 1, 2, 6, 7, 10, 12, 13 and 17 clear brightness dips can be observed. In some quarters very prominent dips can be seen, even more so than for KIC ID 2163434. Again there is the same situation in which there are several dips per light curve, but not with regular intervals. This could again imply multiple exoplanet candidates. Unlike KIC ID 2163434, this star has much more evidence for an anomaly, many more light curves that contain features pointing to an anomaly. Therefore, it is recommended that more research is done on this star.

5 Discussion

As concluded through the results, the neural network seems to be able to detect the features it was trained to find very well. However, these features do apparently not have a one-to-one correlation with exoplanet presence, but are more likely to occur with eclipsing binaries. For this reason it is important to consider doing some data preprocessing on the light curves, to make exoplanet features stand out more in the images. Due to the lack of knowledge on exoplanets and data reduction methods in this field, this was not possible in this paper. For that reason there has been chosen to let the neural network recognize dips in the light curves. A recommendation for follow up research is to bring more expertise in exoplanet detection. Besides this, a suggestion for data reduction could be subtracting a base spline from the light curves in order to reduce the low frequency noise and applying some normalization so that the y-scale can be conserved, which is not the case in this paper.

The final model used seems to work quite well; low validation loss, high validation accuracy and no apparent over-fitting. Training on 913 images for 8 epochs took roughly 180 minutes on an Intel i7 QuadCore CPU and 80 minutes on the norma4 computer cluster at the Kapteyn Astronomical Institute. Considering both machines used a CPU, this is a very reasonable training time. However, the training times were simply too long to make even more complex networks, which could have improved the accuracy even more. Also because of this, just several iterations of a neural network have been made. Therefore, it is not guaranteed that the current model is the best model for the task. In order to be less limited by this factor, one can use GPUs instead. These could reduce the training and prediction time by an order of a magnitude. The current code is compatible with Nvidia GPUs, so using these should not take any extra proceedings. It is also possible to run the code in the cloud using Google's TPUs (Tesla Processing Units), where as much power as needed can be provided (please refer to: `https://cloud.google.com/ml-engine/`).

In order to make more adequate use of neural networks in astronomy, the community will have to collaborate. It has been noticed that in this thesis there was a consistent lack of knowledge about exoplanet detection, which in turn limited the scope of the thesis to mainly focus on the neural network. This led to missed potential for the thesis. A suggestion is to create a close collaboration between a team of exoplanet experts and convolutional neural network experts. However, the aim of this thesis was mostly to gain experience in using these techniques and to create an entry for astronomers in the topic of neural networks. Getting used to work with neural networks took a lot of time. Therefore, that got the priority in this thesis.

In science in general it is of utmost importance to log everything done during the thesis. However, while making this thesis there has been lost track of logging progress. An example is the iterations of the neural network; many iterations have been made, but just the last four versions were accurately logged. Logging the iterations of the neural network and their results are extremely important for follow-up research, since with those results new iterations can be more easily made. In this thesis github has been used for version control, but too few commits have been made. In follow-up research a commit should be done for every newly made iteration of the neural network. Using such a repository system is not enough, since it is important to keep track of the results as well. A spreadsheet as found in Appendix B would suffice for each iteration of the neural network.

6 Conclusion

First off, it has been a very instructive experience trying to apply convolutional neural networks in the field of astronomy, specifically in the field of exoplanet detection. Combining the front lines of computer science with one of the most widely discussed topics in astronomy opens up doors for many other neural network applications in astronomy. A main goal of this paper is to create a low threshold introduction for astronomers into neural networks. Considering the results of the paper, the hope is to have made a positive impression of neural networks and that with even little knowledge impressive results can be achieved.

Although this paper did not produce a perfect solution to detect exoplanets, it made a good basis for follow-up research to start from, including open source code. Moreover, two interesting objects, which are not classified yet, have been detected analyzing 3,000 stars, or 1% of the total number of stars analyzed by the Kepler mission, namely: KIC ID 2163434 and KIC ID 2854994. A closer inspection on these stars pointed out that they both contained irregular dips in brightness, which might implicate exoplanet presence. Further research might be useful to confirm this result. Furthermore, the neural network seems to be able to detect eclipsing binaries with ease, which could make it useful to search for those objects for the TESS mission. The network is very adjustable and reusable, thus having the potential, with some modifications, to be put to use for finding exoplanets.

To conclude, it is possible to accurately detect specific features in images of light curves using convolutional neural networks. For this reason the research question remains unanswered. However, if one is able to bring forward more features of exoplanet presence by applying a certain (automated) data reduction algorithm on the images, it is definitely possible to accurately detect exoplanets from light curves using convolutional neural networks.

Appendix A

The following are all the accuracy and loss results for the 4 iterations of the neural network.

300x300 image

$\Downarrow$

10@10x10 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

40@3x3 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

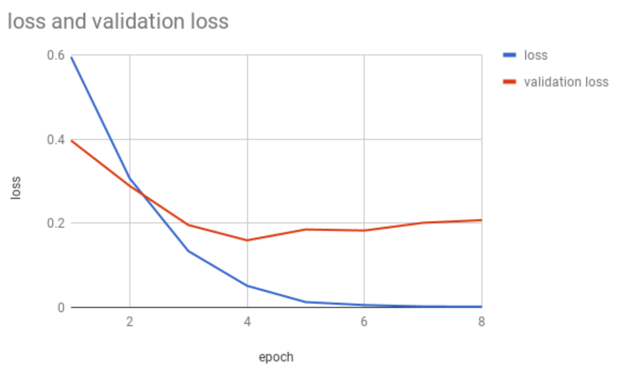
flatten to 1d matrix

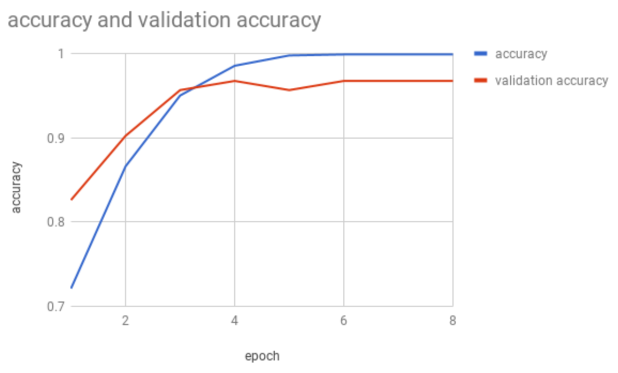
$\Downarrow$

dense layer 50 nodes, ReLU

$\Downarrow$

dense layer 1 node, Sigmoid





300x300 image

$\Downarrow$

10@10x10 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

40@3x3 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

flatten to 1d matrix

$\Downarrow$

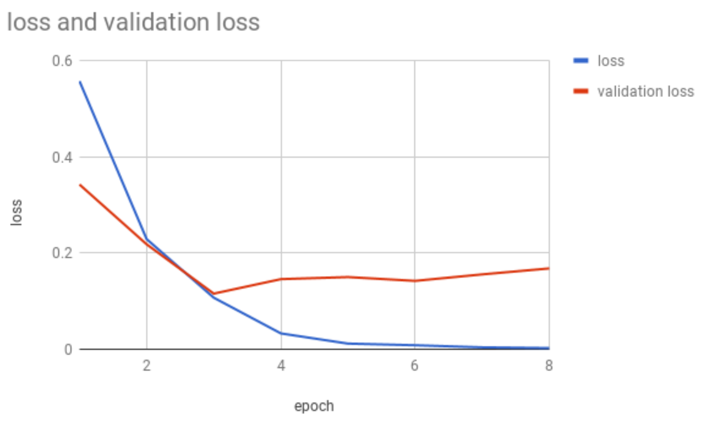
dense layer 50 nodes, ReLU

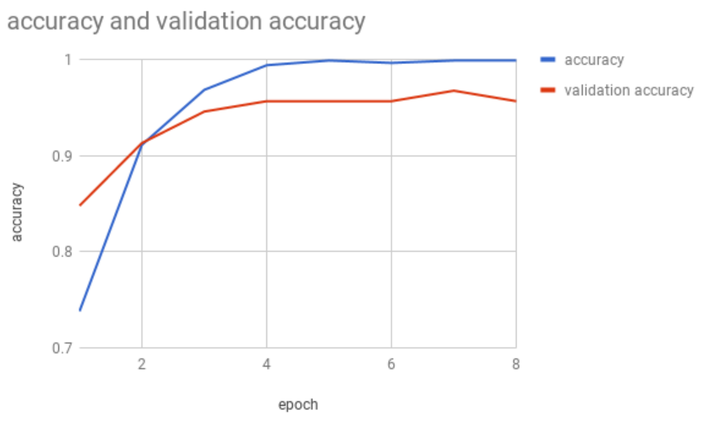
$\Downarrow$

dropout of 0.1

$\Downarrow$

dense layer 1 node, Sigmoid





300x300 image

$\Downarrow$

30@4x4 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

20@3x3 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

flatten to 1d matrix

$\Downarrow$

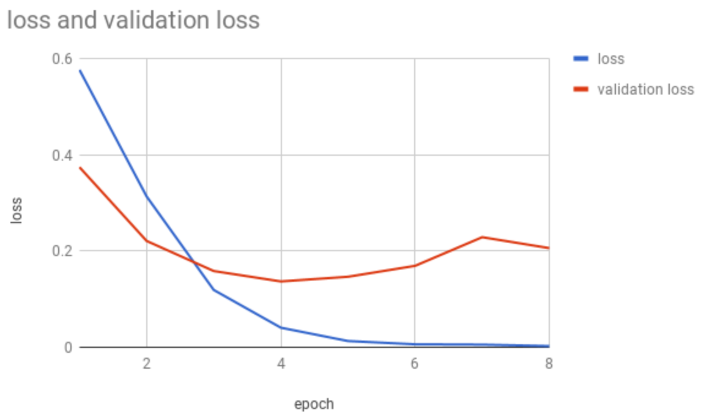
dense layer 50 nodes, ReLU

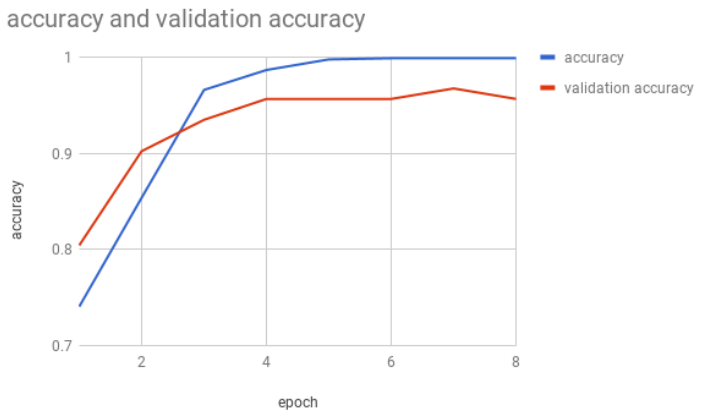
$\Downarrow$

dropout of 0.1

$\Downarrow$

dense layer 1 node, Sigmoid





Final model

300x300 image

$\Downarrow$

5@50x50 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

10@4x4 feature maps (convolution)

$\Downarrow$

2x2 maxpooling

$\Downarrow$

flatten to 1d matrix

$\Downarrow$

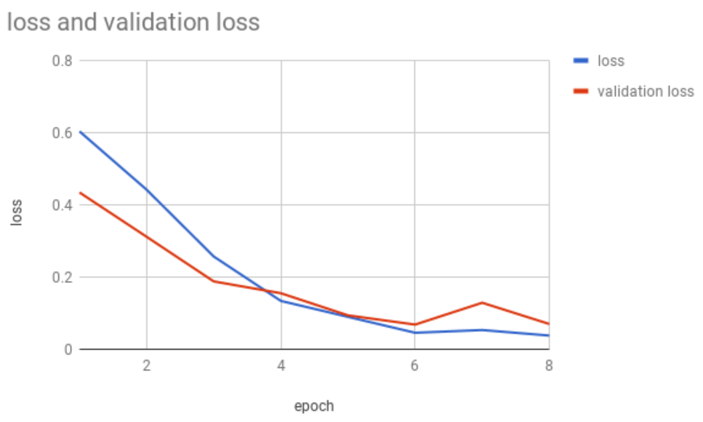
dense layer 128 nodes, ReLU

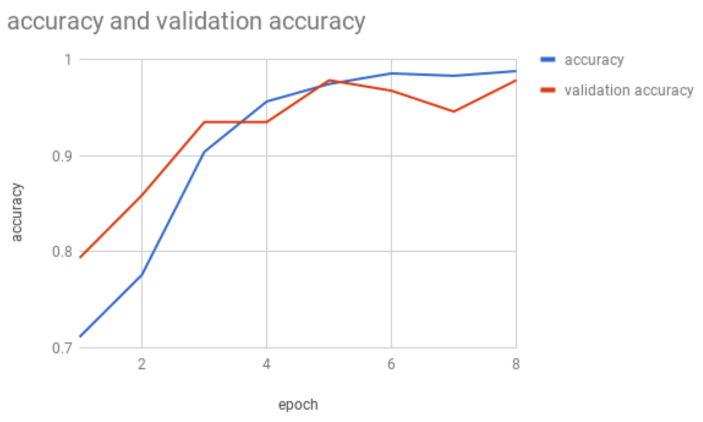
$\Downarrow$

dropout of 0.1

$\Downarrow$

dense layer 1 node, Sigmoid



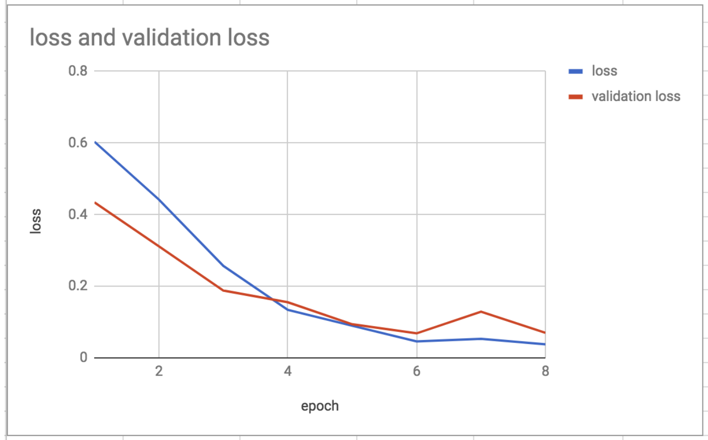


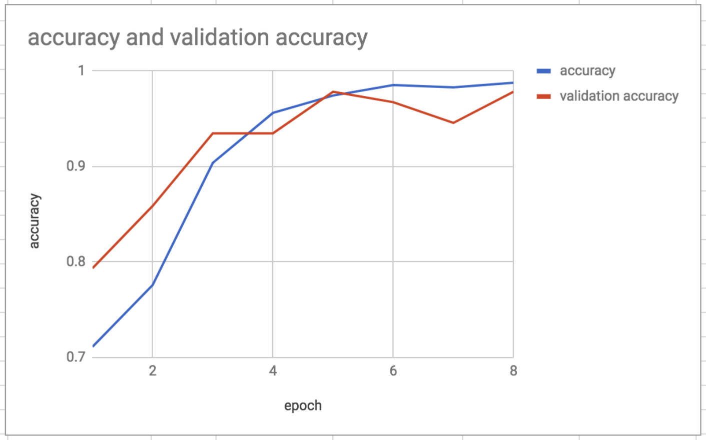
Appendix B

epoch	loss	accuracy	validation loss	validation accuracy
1	0.6036	0.7113	0.4344	0.7935
2	0.442	0.7759	0.3116	0.8587
3	0.257	0.9038	0.188	0.9348
4	0.134	0.9562	0.1553	0.9348
5	0.0898	0.9744	0.0939	0.9783
6	0.0458	0.9854	0.0684	0.9674
7	0.0533	0.9829	0.1289	0.9457
8	0.038	0.9878	0.07	0.9783

batch size	40
epochs	8
training samples	821
validation samples	92

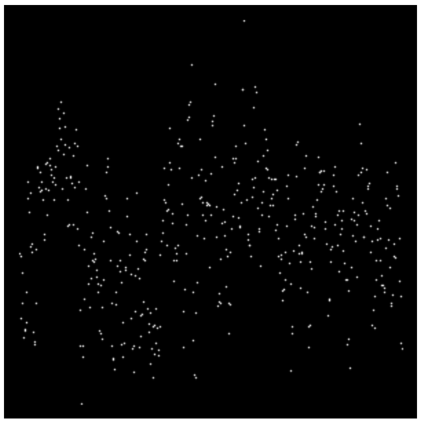
layout			training time
type	specs		
convolutional2D	5@50x50		
maxpooling2D	2x2		
convolutional2D	10@4x4		
maxpooling2D	2x2		
flatten			
dense	128, ReLU		
dropout	0.1		
dense	1, Sigmoid		

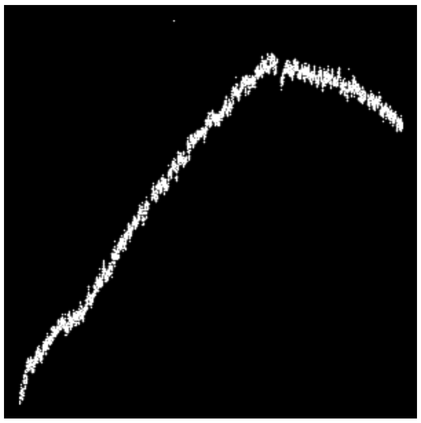


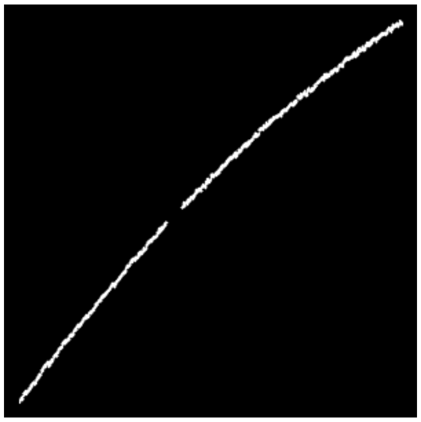


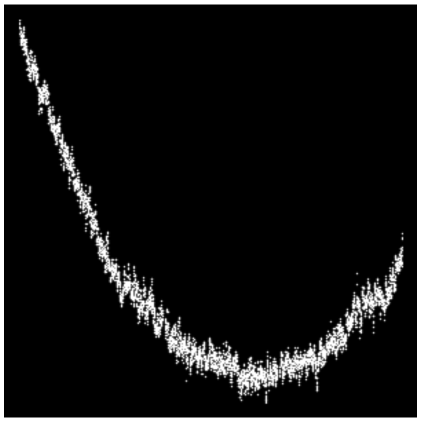
7 Appendix C

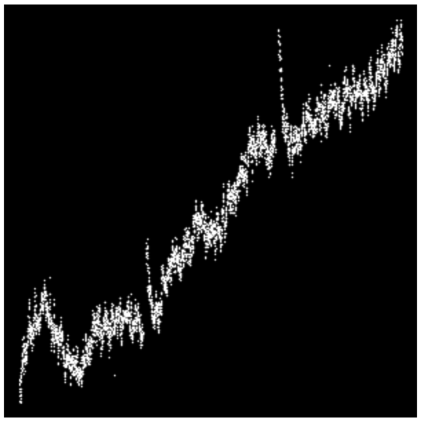
Light curves of the star with KIC ID 2163434


(a) Quarter 0


(b) Quarter 7

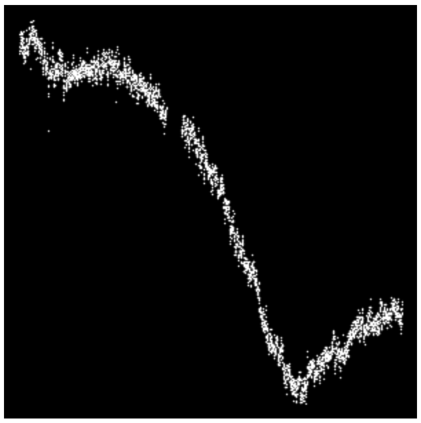

(c) Quarter 8

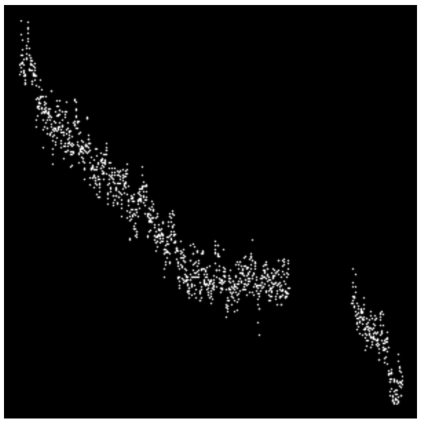

(d) Quarter 9

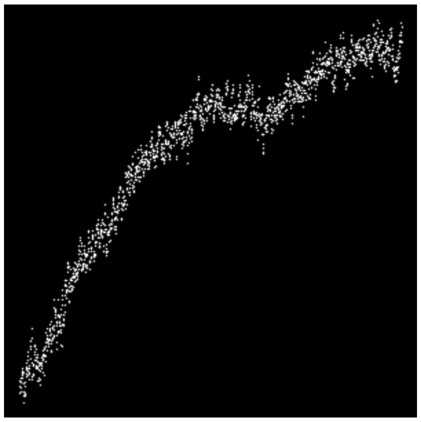

(e) Quarter 10

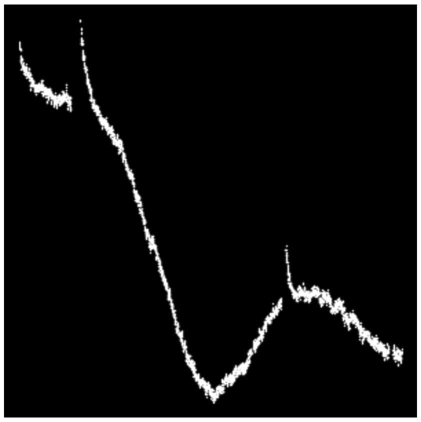
8 Appendix D

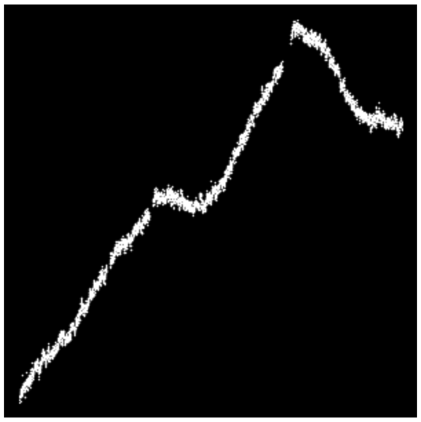
Light curves of the star with KIC ID 2854994

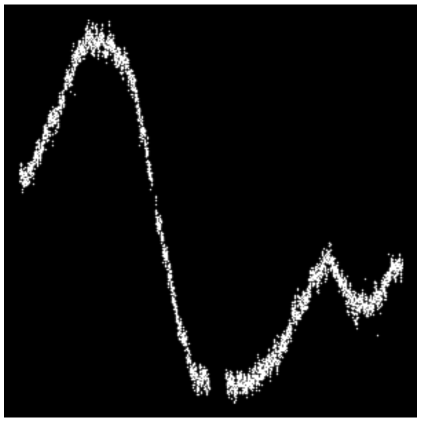

(a) Quarter 0

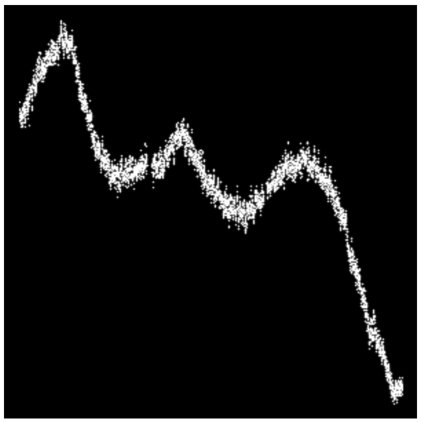

(b) Quarter 1

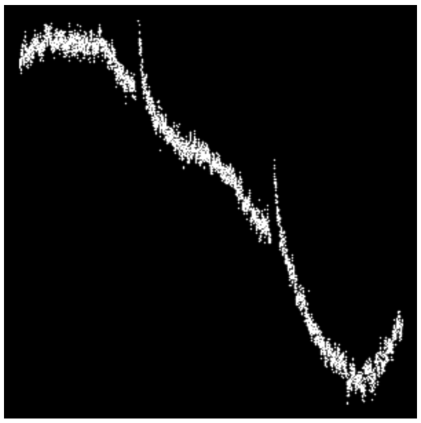

(c) Quarter 2

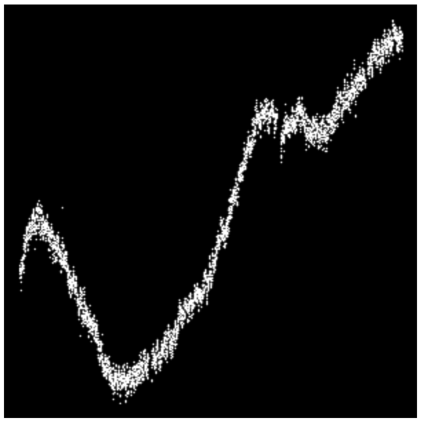

(d) Quarter 3

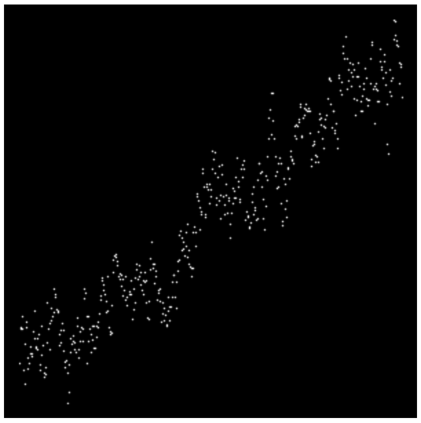

(e) Quarter 4

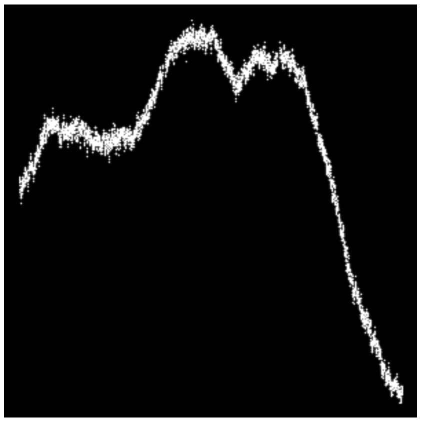

(f) Quarter 5

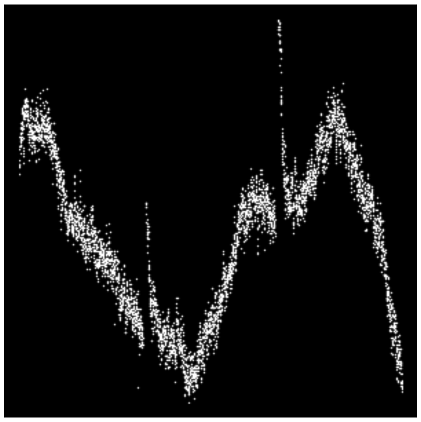

(a) Quarter 6

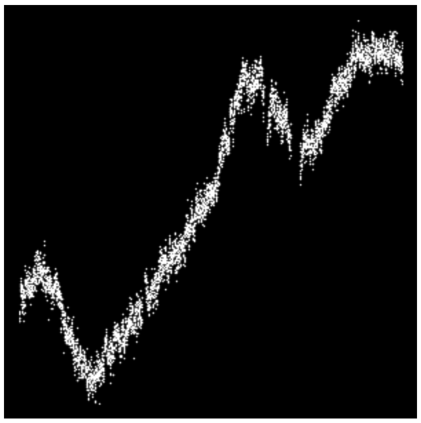

(b) Quarter 7

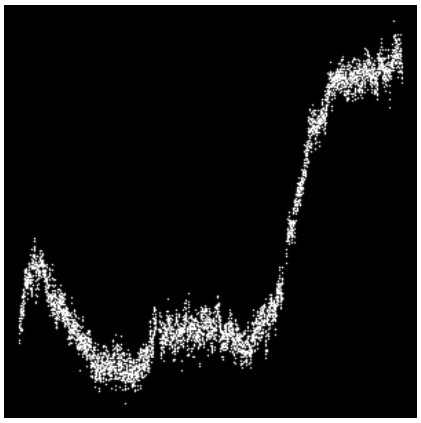

(c) Quarter 8

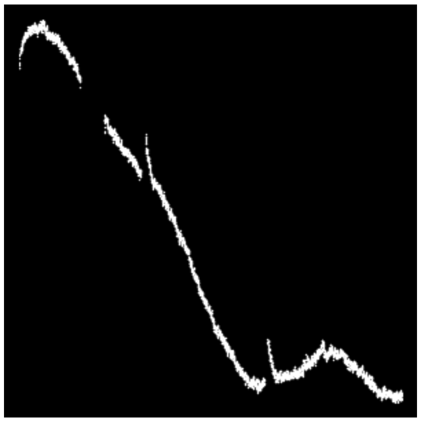

(d) Quarter 9

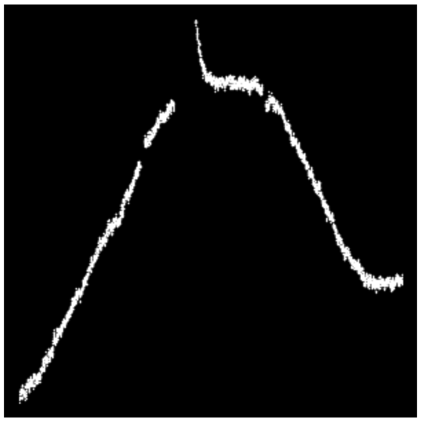

(e) Quarter 10

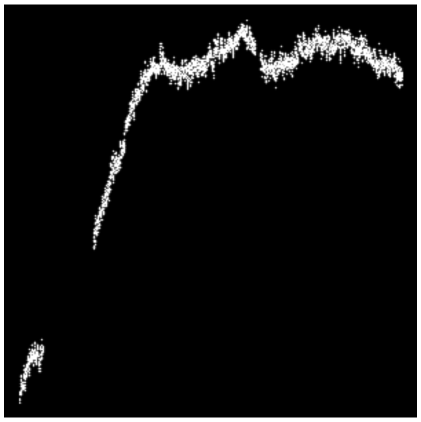

(f) Quarter 11

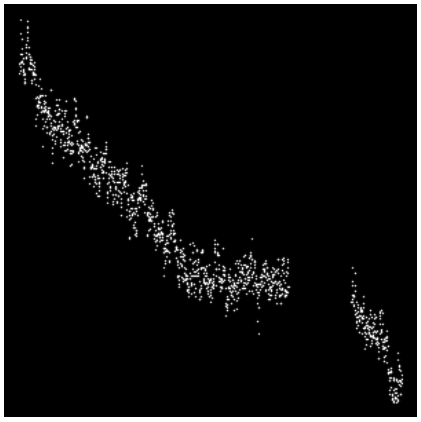

(a) Quarter 12


(b) Quarter 13


(c) Quarter 14


(d) Quarter 15


(e) Quarter 16


(f) Quarter 17

References

Barentsen, G. (2018). Science from kepler. Retrieved April 26, 2018, from `https://keplergo.arc.nasa.gov/science.html#science-from-kepler`.

De Boer, P.-T., Kroese, D. P., Mannor, S., and Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67.

Deeg, H. (1998). Photometric Detection of Extrasolar Planets by the Transit-Method. In Rebolo, R., Martin, E. L., and Zapatero Osorio, M. R., editors, *Brown Dwarfs and Extrasolar Planets*, volume 134 of *Astronomical Society of the Pacific Conference Series*, page 216.

Deshpande, M. (2017). Perceptrons: The first neural networks. Retrieved May 12, 2018, from `https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/`.

Johnson, M. (2018). Kepler discoveries. Retrieved April 26, 2018, from `https://www.nasa.gov/kepler/discoveries`.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Masetti, M. (2018). How many stars in the milky way? Retrieved April 26, 2018, from `https://asd.gsfc.nasa.gov/blueshift/index.php/2015/07/22/how-many-stars-in-the-milky-way/`.

MathWorks (2018). Convolutional neural network. Retrieved May 12, 2018, from `https://nl.mathworks.com/discovery/convolutional-neural-network.html`.

Miller, A. (1993). A review of neural network applications in astronomy. *Vistas in astronomy*, 36:141–161.

NASA and STScI (2018). Mast database. Retrieved May 14, 2018, from `https://archive.stsci.edu/`.

Ricker, G. R., Latham, D. W., Vanderspek, R. K., Ennico, K. A., Bakos, G., Brown, T. M., Burgasser, A. J., Charbonneau, D., Clampin, M., Deming, L. D., Doty, J. P., Dunham, E. W., Elliot, J. L., Holman, M. J., Ida, S., Jenkins, J. M., Jernigan, J. G., Kawai, N., Laughlin, G. P., Lissauer, J. J., Martel, F., Sasselov, D. D., Schingler, R. H., Seager, S., Torres, G., Udry, S., Villasenor, J. N., Winn, J. N., and Worden, S. P. (2010). Transiting Exoplanet Survey Satellite (TESS). In *American Astronomical Society Meeting Abstracts #215*, volume 42 of *Bulletin of the American Astronomical Society*, page 459.

Santerne, A., Díaz, R. F., Moutou, C., Bouchy, F., Hébrard, G., Almenara, J.-M., Bonomo, A. S., Deleuil, M., and Santos, N. C. (2012). SOPHIE velocimetry of Kepler transit candidates. VII. A false-positive rate of 35% for Kepler close-in giant candidates. *A&A*, 545:A76.

Shallue, C. J. and Vanderburg, A. (2018). Identifying Exoplanets with Deep Learning: A Five-planet Resonant Chain around Kepler-80 and an Eighth Planet around Kepler-90. *AJ*, 155:94.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Vanderburg, A. (2017). Transit light curve tutorial. Retrieved June 30, 2018, from `https://www.cfa.harvard.edu/~avanderb/tutorial/tutorial2.html`.