Institute of Physics and Astronomy

Aarhus University

# Classifying Stellar Pulsations based on their Light Curves using Machine Learning

Bachelor Project

Spring 2018

Jonas Dornonville de la Cour

201510352

DATE OF SUBMISSION: 15/06/18

Project Supervisors: Kuldeep Verma and Mikkel Nørup Lund

## Contents

## Resumé

The Transiting Exoplanet Survey Satelite (TESS) forventes at levere tidsserie luminositets data fra flere millioner stjerner. Eftersom mængden af data er så stor, bliver det nødvendigt at automatisere meget af analyseprocessen. I dette projekt vil vi kategorisere simulerede lyskurver i 14 forskellige kategorier ved brug af machine learning. TESS' kameraer bruger en lukketid på 2 sekunder og bruger gennemsnittet af informationen fra hvert billede til at generere short cadence (gennemsnit over 2 min.) og long cadence (gennemsnit over 30 min.) lyskurver. Ved at træne et 4 lags Convolutional Neural Network (CNN) på 2256 long cadence lyskurver, med 565 lyskurver sat til side som validerings data, kan vi opnå en klassifiserings nøjagtighed på 81(1)%. Klassificeringsalgoritmen opnår en nøjagtighed på over 85% for 7 af de 14 kategorier, de resterende kategorier er dårligt representeret i validerings dataene. Ved træning af yderligere 11307 mere ligeligt fodelte (blandt kategorierne) lyskurver opnås en nøjagtighed på 68.0(2)% med samme netværk. Ved brug af en Residual Convolutional Neural Networks struktur opnås på samme data en nøjagtighed på 68(1)% med et bedste resultat på 69%. Når modellerne trænes på de yderligere dataset ser vi en klassificeringsnøjagtighed på over 70% for 7 af de 14 kategorier. Ydermere forbedres klassificeringsnøjagtigheden for solar-like kategorien fra 40(14)% til 58(2)%.

## Summary

The Transiting Exoplanet Survey Satelite (TESS) is set to deliver time series brightness data from several million stars. Due to the amount of data automatic analysis is key and in this project we apply machine learning techniques to classify simulated light curves into 14 categories. The TESS cameras use an exposure time of 2 seconds and averages the information from each picture in order to generate short cadence (average over 2 min.) and long cadence (average over 30 min.) light curves. By training a 4 layer Convolutional Neural Network (CNN) structure on 2257 long cadence data sets and using 565 validation sets, an accuracy of 81(1)% is achieved. The classifier has a classification accuracy above 85% on 7 of the 14 classes, the remaining classes have poor representation in the validation set. Using an additional 11307 diversely distributed long cadence light curves an accuracy of 68.0(2)% is achieved using the same architecture. Using Residual Convolutional Neural Network structure on the same data we achieve an accuracy of 68(1)% and a best accuracy of 69%. These classifiers show accuracy above 70% on 7 of the 14 classes with an improved solar-like classification accuracy of 58(2)% as opposed to 40(14)% without the additional light curves.

## 1 Introduction

On the 18th of April 2018, the Transiting Exoplanet Survey Satelite (TESS) was launched with the objective of observing more than 20 million stars.[6] TESS surveys large sections of the sky at a time taking pictures with short and long cadence (2 minutes and 30 minutes respectively) for 27 days. The brightness of each star in each picture is recorded and ultimately constitutes time resolved brightness data for the observed stars (lightcurves).[7]

Analysis of time resolved star brightness data has in the past been dominated by expert inspection of each individual dataset to initially find which stars are showing oscillations such as to pick stars for further observation by ground based observatories. The amount of stars that can be observed has however increased dramatically since the start of the century, particularly the K2 Galactic Archeology Program provided thousands of red giant candidates from each campaign. For this reason expert inspection becomes extremely time consuming, impractical and introduces inherently subjective decision making which limits reproducibility. The aforementioned problems have introduced the need for automated categorization and detection of stellar oscillation. Up until recent years, high order parametric model fitting was the state of the art method for oscillation detection. These models have yet to show the same detection rate as expert visual inspection.[9]

Recently however, machine learning techniques have found their way into this field and show promising results.[9] Inspired by these developments in machine learning driven asteroseismology and the unprecedented need for automation posed by TESS, this project aims to use a combination of various machine learning techniques to classify light curves which are simulated to resemble observational data from TESS.

## 2 Theory

The derivations of the gradient descent and backpropagation algorithms in sections 2.2 and 2.3 as well as the explanation of neural networks in section 2.1, are heavily inspired by the first and second chapter of Neural Networks and Deep Learning by Michael Nielsen.[1][2]

### 2.1 Artificial Neural Networks

There are a couple of different techniques that achieve machine learning but neural networks in particular are responsible for breakthroughs in speech recognition and most importantly, computer vision. The latter is the inspiration to use neural networks in this project.

An Artificial Neural Network (ANN) is a very complex mathematical model which approximates the complicated relationship between a given input and output. The ANN importantly consists of neurons. Neurons make decisions based on the input it receives as seen in figure 1. These decisions can either be binary as is the case for the 'perceptron' or it can be a continuous function of the input, as is the case for most neuron types.

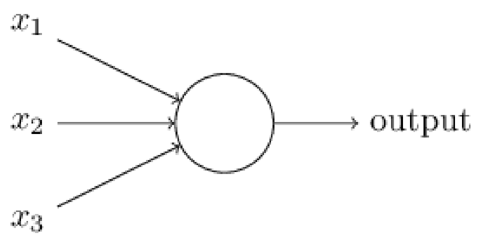


Figure 1: [1] Depicts the general operation of an artificial neuron

Different types of neurons are distinguished by their output function known as an activation function. For the case of the perceptron this activation function is defined as,

$$\text{output} \quad = \quad \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{1}$$

where the $w_j$'s are the weights corresponding to each input $x_j$. For notation purposes we redefine the threshold in terms of a bias, $b \equiv -threshold$ such that in vector notation we get,

$$\text{output} \quad = \quad \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{2}$$

If we connect multiple such neurons together we can in the perceptron case perceive it as a bunch of connected if-statements. To improve this intuition lets consider a binary example where you want to decide whether or not to eat ice cream. We might consider the factors to answer this question threefold as such:

- Am I overweight?
- Is the weather good?
- Do I have a craving for ice cream?

We can proceed to ascribe a value to each of these conditions, $x_1$, $x_2$ and $x_3$. Let us assume that $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ and the weights are $w_1 = 0$, $w_2 = 1$, $w_3 = 2$ lastly we need to choose a bias or threshold value $b_1 = 2$. In this scenario we're really prone to instant gratification and don't care whether we're overweight or if the weather is any good, all that matters is whether or not we crave the ice cream. The idea however is that the neuron should be able to learn the right weights and biases such that it makes the right decision, if for instance we increase the bias to $b_1 = 3$ we impose the additional criteria that both the weather has to be good AND the craving has to be present. Additionally if we cared about healthy living then $w_1$ should be non-zero. In this example we demonstrate that in spite of their biologically inspired name neurons are fundamentally nothing but math.

The binary nature of the perceptron however makes it a poor candidate for learning. In the ice cream example changing one of the weights by a small amount $\Delta w$ causes a huge change in the output $\Delta output$ In order for the network to learn, it needs to be able to adjust it's weights and biases gradually to achieve better results. For this reason, smooth analogues of the perceptron exists, namely sigmoid and tanh. In these cases the activation functions are,

$$\text{output} = \frac{1}{1+e^{-z}} \tag{3}$$

and

$$\text{output} = tanh(z) \tag{4}$$

If we plot these functions they look something like this,

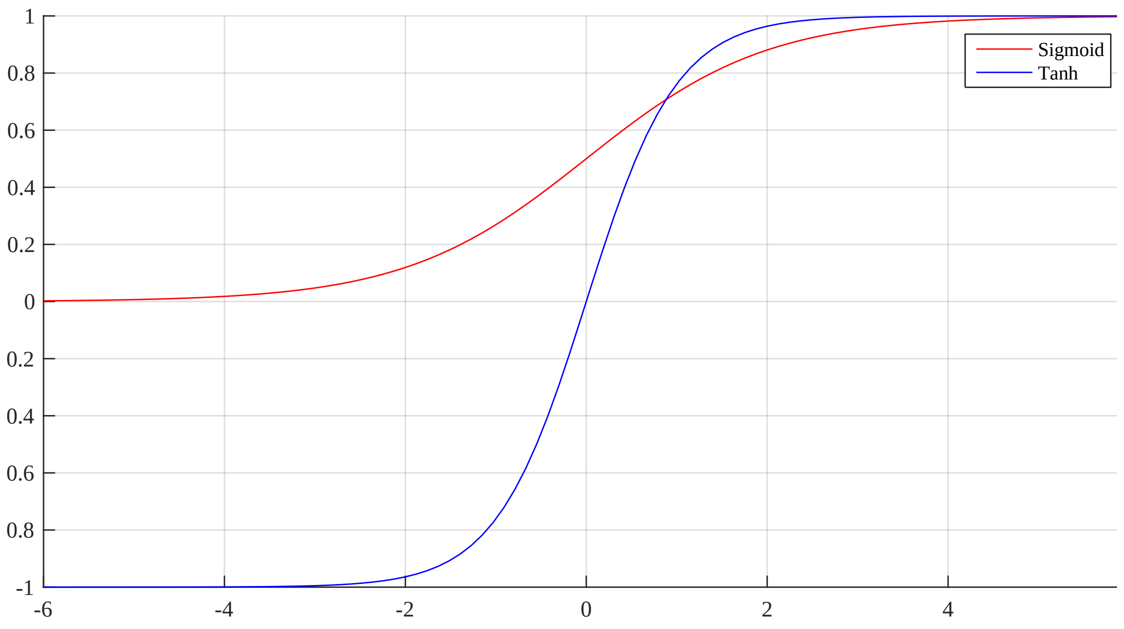


Figure 2: Illustration of smooth activation functions

It becomes immediately apparent how, in these cases, a small change in input, $z = w \cdot x + b$, causes a small change in output. Thus the process of optimizing weights and biases becomes gradual.

In order to make complicated decisions based on input, we simply increase the number of neurons and connect them in all possible combinations as such,

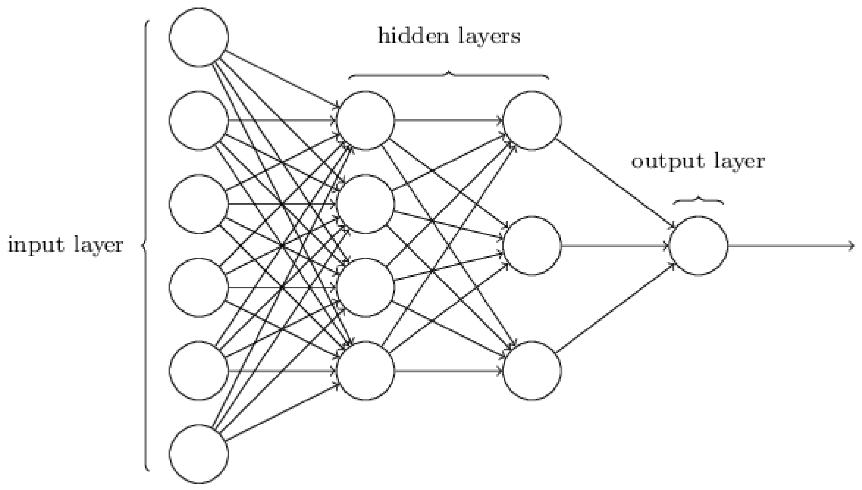


Figure 3: [1] Illustration of deeply connected layers

You could of course limit the connections such that it is more in the range of 1-2 connections/neurons but all that would achieve is limiting which connections can be made by the network and it is not possible to presuppose which connections are relevant and which are not. If you connect the neurons as depicted in figure 3, the layers are called deeply connected layers. If the network, like in figure 3, has multiple hidden layers (layers between the input and output layers) it is called a Deep Neural Network or DNN.

## 2.2 Gradient Descent

In order to understand how the network actually learns the weights and biases of each neuron and connection in the network we need gradient descent. Gradient descent is one of many optimization methods, but it is particularly useful because of its scalability to high dimensional problems as is the case for artificial neural networks.

As with any optimization problem we need a way to evaluate the performance of our model at all times. There are a couple of common functions which evaluate the performance, these are called cost or loss functions. Let us examine the quadratic loss function,

$$C(w,b) = \frac{1}{2n}\sum_x ||y_{actual}(x) - y_{out}(x,w,b)||^2 \tag{5}$$

Where $x$ , $w$ and $b$ are vectors, $x$ being the input vector, $y_{actual}$ is typically a standard basis vector with dimension equal to the possible outputs of the problem. For instance if we consider the popular classification problem of identifying handwritten digits, the $y_{actual}$ corresponding to figure 4 would be $y_{actual} = (1,0,0,0,0,0,0,0,0,0)^T$. Similarly $y_{out}$ has the same shape as $y_{actual}$ and denotes the resultant output vector of the network given the input vector $x$, weights $w$ and biases $b$. From equation (5) we see that as $y_{out}$ approaches $y_{actual}$ C(w,b) approaches 0 as intended.

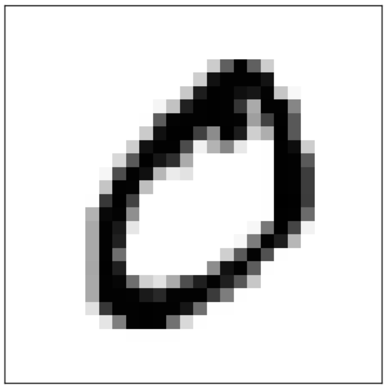


Figure 4: [1] Bitmap image of a handwritten zero

One might wonder why use quadratic cost function as a measure for performance? Why not try to minimize the number of incorrect classifications directly such that the cost is a sum of incorrect classifications? The issue with this is that such a function would not be a smooth function of the weights and biases. Since most changes to biases and weights will not affect the number of correct classifications, it is hard for the network to learn weights and biases.

We can think of the cost function C(w,b) as some arbitrary function C($v$) where $v = v_1, v_2, ..., v_n$ . In the case where $dim(v) = 2$, we can illustrate the function as a surface plot,

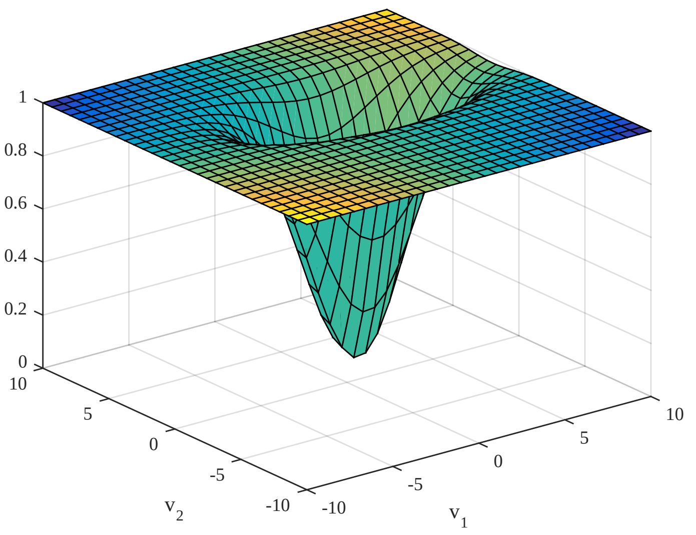


Figure 5: Surface plot of an arbitrary 2 dimensional gaussian distribution

In figure 5 the global minimum of the function is very well defined and easy to just eyeball but in most real cases the function is far more complicated than this and $dim(v)$ might be of the order $10^6$. This illustration is then of course a simplification regardless, it allows for the right intuition. One might try to calculate the partial derivatives of the cost function with respect to each variable analytically and then use those to find the local minimum. This is feasible for low dimensional problems but becomes virtually impossible for high dimensions. However we can find analytical partial derivatives with respect to each variable and call that the gradient of the cost function $\nabla C$,

$$\nabla C = (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_n}) \tag{6}$$

And the change of the cost function can then be calculated as,

$$\Delta C \approx \sum_i \frac{\partial C}{\partial v_i} \Delta v_i \tag{7}$$

or in dot product notation

$$\Delta C \approx \nabla C \cdot \Delta v \tag{8}$$

Equations (7) and (8) are only valid approximations for small changes $\Delta v$ because changes in the cost function with respect to one variable still depend on the other variables. Now we need a way to choose the change in parameters $\Delta v$ such that $\Delta C$ becomes negative, suppose that,

$$\Delta v = -\eta \nabla C \tag{9}$$

where $\eta$ is a positive parameter known as the learning rate. From equation (8) plugging in equation (9) yields $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2$ and since $||\nabla C||^2 \geq 0$ this guarantees that $\Delta C \leq 0$ within the bounds of equation (8). In order for the gradient descent algorithm to minimize the cost function it simply has to iteratively update the variables according to,

$$v \rightarrow v' = v - \eta \nabla C \tag{10}$$

which for weights and biases is equivalent to,

$$w_j \rightarrow w'_j = w_j - \eta \frac{\partial C}{\partial w_j} \tag{11}$$

$$b_k \rightarrow b'_k = b_k - \eta \frac{\partial C}{\partial b_k} \tag{12}$$

However the partial derivatives in equations (11) and (12) are of course averages over all the training inputs which, if say one has millions of training inputs, will make learning computationally slow. This can be amended by using stochastic gradient descent which makes the approximation that the average gradient of every input is roughly equal to the average gradient over a small subset (mini batch) of the inputs ,

$$\frac{\sum_{i=1}^{m} \nabla C_{X_i}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{13}$$

where $X_j$ is an element of the training inputs in a mini-batch and $m$ is the batch size. With this approximation we can reiterate equations 11 and 12,

$$w_j \rightarrow w'_j = w_j - \frac{\eta}{m} \sum_i \frac{\partial C_{X_i}}{\partial w_j} \tag{14}$$

$$b_k \to b'_k = b_k - \frac{\eta}{m}\sum_i \frac{\partial C_{X_i}}{\partial b_k} \tag{15}$$

When we iterate equations (14) and (15) enough to exhaust all mini-batches $\lceil \frac{n}{m} \rceil$ in the training data, we have completed what is referred to as an epoch. Now what is left is to compute the partial derivatives, which is where backpropagation comes in.

## 2.3 Backpropagation

In order to calculate the derivatives with respect to each weight and bias in the network we need to work backwards from the last layer, the output layer, because it determines the cost $C(w, b)$. In order to describe derivatives we need some unambiguous notation. Let $w^l_{jk}$ denote the weight of the connection between the $k^{th}$ neuron in the $(l-1)^{th}$ layer to the $j^{th}$ neuron in the $l^{th}$ layer. More intuitively the bias $b^l_j$ and activation $a^l_j$ refer to the bias and activation of the $j^{th}$ neuron in the $l^{th}$ layer respectively.

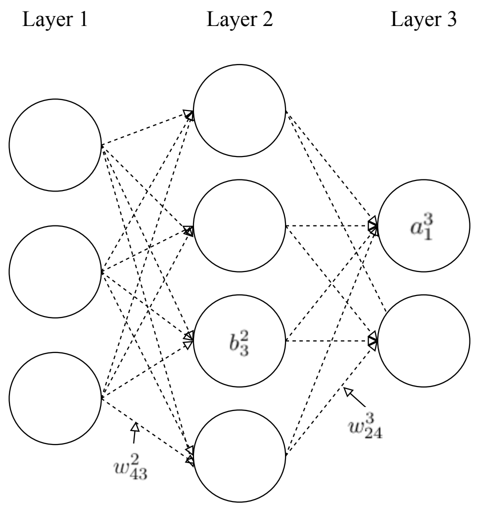


Figure 6: Illustration of the weight, bias and activation notation used in backpropagation

The activation illustrated in figure 6 is simply defined as,

$$a^l_j = \phi\left(\left(\sum_k w^l_{jk} a^{l-1}_k\right) + b^l_j\right) \tag{16}$$

or in vector notation as,

$$a^l = \phi(w^l a^{l-1} + b^l) = \phi a^l = \phi(z^l) \tag{17}$$

Where $z^l = w^l a^{l-1} + b^l$ and $\phi$ is the activation function of choice (sigmoid, tanh, ReLU etc.). To take the quadratic cost function as an example we can see how it depends on the activation of the last layer,

$$C = \frac{1}{2n}\sum_x ||y_{actual}(x) - a^L(x)||^2 \tag{18}$$

where L is the number of layers in the network i.e the last layer. Now we can express the gradient of the cost function with respect to the activations as $\nabla_a C = \frac{\partial C}{\partial a^L}$ and define the error on the $j^{th}$ neuron in the $L^{th}$ layer as,

$$\delta^L = \nabla_a C \circ \phi'(z^L) \tag{19}$$

where $\phi'$ is the derivative of the activation function and thus gives a measure for the rate of change of the activation function at $z_L$ and $\circ$ is notation for the entrywise product known as Hadamard product. The derivative of the quadratic cost function equation (18) is straight forward, $\nabla_a C = (a^L - y_{actual})$ thus equation (19) becomes,

$$\delta^L = (a^L - y_{actual}) \circ \phi'(z^L) \tag{20}$$

Now that we have an expression that determines the error on the last layer as a function of $a^L$ and $z^L$ we need the second part of backpropagation which is to determine the error on each layer in terms of the error in the next layer that is,

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \phi'(z^l) \tag{21}$$

This can intuitively be thought of as propagating the error backwards in the network by applying the weight matrix to the errors in the $(l+1)^{th}$ layer. By combining equations (20) and (21) we can recursively calculate the errors on the weighted input to each layer for an arbitrary number of layers. This is achieved by first calculating $\delta^L$ then $\delta^{L-1}$ and so on. The last thing we need is a way to express the partial derivatives of the cost with respect to any bias and weights in the network, in terms of $\delta$. For the bias it is incredibly simple because it is a constant term in the activation independent of the weights and activation of previous layers, thus,

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{22}$$

and in vector notation,

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{23}$$

For the activations it is slightly trickier because the $j^{th}$ activation in the $l^{th}$ layer is $\sum_k a^{l-1}_k w^l_{jk}$ so naturally the error on each weight is proportional to that same $a^{l-1}_k$ factor,

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{24}$$

Now we have all the components of backpropagation which can be described as the following sequence:

- Setting the activation of the input layer $a^1$ equal to the input $x$
- For each layer $l = 2, ..., L$ calculate the weighted inputs and activations, $z^l = w^l a^{l-1} + b^l$ and $a^l = \phi(x^l)$
- Calculate the output error $\delta^L$ as per equation (20)
- Backpropagate the error through the layers in order to calculate $\delta^l$ according to equation (21)
- Lastly using equations (23) and (24) we compute all the partial derivatives

When we have calculated the partial derivatives with respect to weights and biases we can finally apply gradient descent i.e. update weights and biases according to equations (14) and (15).

## 2.4 Overfitting and Regularization

In statistics an overfitted model is a model that contains more parameters than can be justified by the data. In essence overfitting in the example of polynomials of the same order as the number of data points.

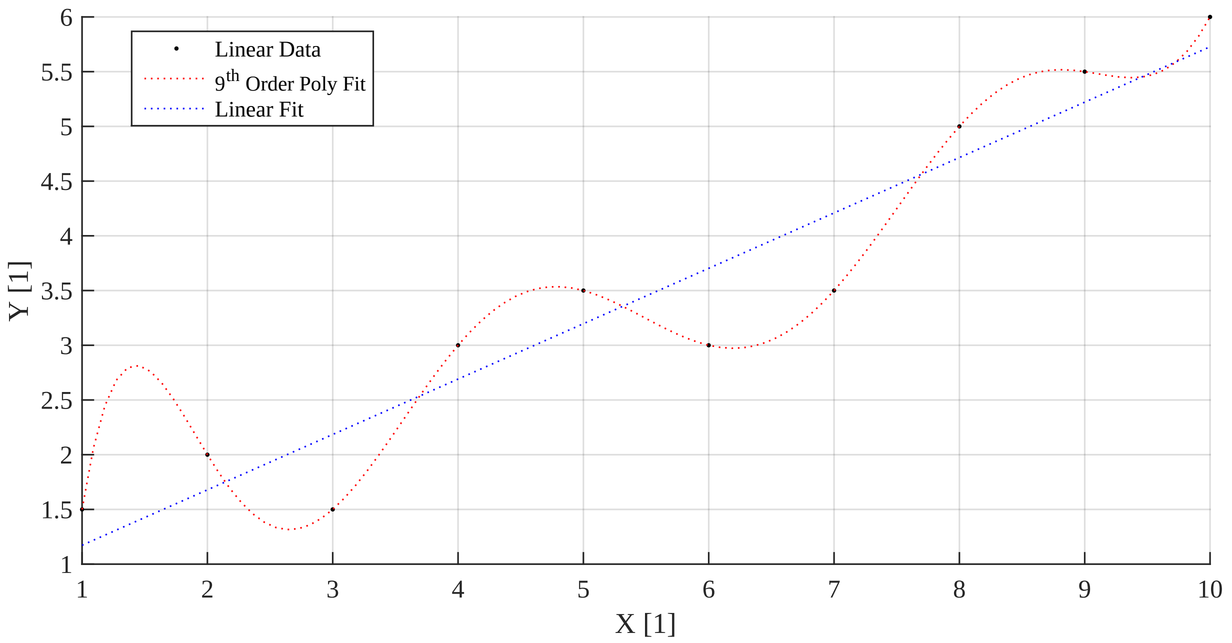


Figure 7: Example of linear data which is shifted by a random constant between 0 and 1. The data is then curvefitted to a 9th order polynomial and a 1st order polynomial.

In figure 7 the problem of overfitting is exaggerated to the point where the number of parameters matches the number of data points. The problem with this kind of fitting is that the values extrapolated from this fit do not generalize very well. In the case of machine learning we face similar problems when the number of parameters approaches the number of data points. More generally overfitting happens when the underlying structure of the problem has fewer parameters than the model.[3] The number of parameters in the underlying problem is unknown so some level of overfitting or underfitting is guaranteed.

Typically in machine learning the complexity of the model supersedes the complexity of the underlying problem and we need some way of reducing overfitting. The process of reducing overfitting in a model is called regularization. There are two main forms of regularization in machine learning namely dropout and L2-norm. Dropout randomly sets neuron activations in a layer equal to zero, the number of neurons that are nullified is the dropout rate. Hence dropout reduces the number of effective parameters in the network by a set percentage. L2-norm, also known as Tikhonov regularization, reduces overfitting by penalizing the cost function based on the norm sum of the weights in the network squared i.e. the cost function becomes,

$$C'(w,b) = C(w,b) + \lambda \left( \sum ||w||^2 + \sum ||b||^2 \right) \tag{25}$$

Here $\lambda$ denotes the L2-norm regularization constant factor which can be tuned to increase or decrease penalty to the cost function.

## 2.5 Hyperparameters

Using gradient descent in connection with backpropagation we can theoretically achieve learning in a neural network. In order to do that we need to design the actual neural networ,k which means we need to answer a couple of questions like:

- How many layers should the network consist of?
- How many neurons should be in each of the layers?
- What activation function should we use?
- What should the learning rate $\eta$ be?
- What should our batch size $m$ be?

Depending on the architecture there can be many more parameters to tune. So the question is how do we determine good values for these hyperparameters? The answer is that there are no rigorous rules that determine good hyperparameters because neural networks really are black boxes, therefore the best we can do is heuristics. For instance it helps to think of the number of neurons in a densely connected layer as features or subproblems. That is to say, if one wishes to recognize a face in an image you might want to divide it into cheeks, noses, eyes, foreheads, hair, ears, mouths etc. and then divide those problems into even tinier subproblems like eyebrows, pupils, irises, eyelashes and so on. Proceed to divide the subproblems into subproblems until you reach basic geometric shapes that can be determined by pixels that are grouped together. Now the way to choose learning rate is mostly trial and error but the more complex the function the smaller the learning rate probably needs to be. When all this is said and done most machine learning algorithms are built based on a combination of experience i.e. 'this worked in the past so it might work here' and trial and error - lots of trial and error. Similar problems often have similar solutions so if you built an algorithm to detect cars in images it probably works well for motorcycles or even boats (frankly most objects) too. A key ingredient in choosing good hyperparameters is getting quick feedback on how well the network is performing. This means one should not start of using all 1 million data sets, instead start with a fraction of that because all you are initially looking for is whether or not the network is learning or if it's just generating random classifications/noise. Additionally you should not let the network run for an absurd amount of epochs if there is no learning. This is known as early stopping.

There are a lot of ongoing efforts to automate the process of finding good hyperparameters. Algorithms such as grid search, random search, Bayesian optimization, gradient based optimization are all examples of this. They all have the same thing in common, it is extremely computationally costly to iteratively try network hyperparameters off in a systematic way without human intervention because each set of parameters requires you to train the network. However if the problem is small enough and certain pruning is done to the training process it might be feasible to use systematic hyperparameter tuning.

## 2.6 Convolutional Neural Networks

A variation of neural networks known as convolutional neural networks (CNNs) have proven to be exceptionally useful in machine learning problems such as computer vision, text and speech recognition among others. Convolution works in three steps:

- Convolution convolves so called filters over the input
- Activation applies the activation function to the stack generated by the convolution layer.
- Max Pooling downsamples the spatial dimension of the stack by picking out max values in discrete intervals.

If we imagine a 3 dimensional image (2 spatial dimensions and 1 color dimension RGB) of size 32x32x3 and decide the filters are going to be 5x5x3 then it can illustrated like this,

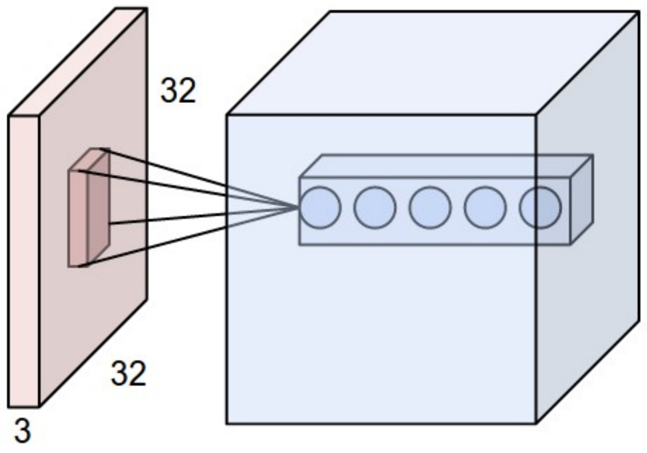


Figure 8: [4] Simple illustration of the convolution operation with inputsize 32x32x3 and filter size 5x5x3.

The example uses 5 filters and each filter generates its own so called feature map corresponding to the 5 neuron depth in figure 8. To further help this intuition of iterating a filter over an input image let us consider the individual slices and filters to understand the workings.

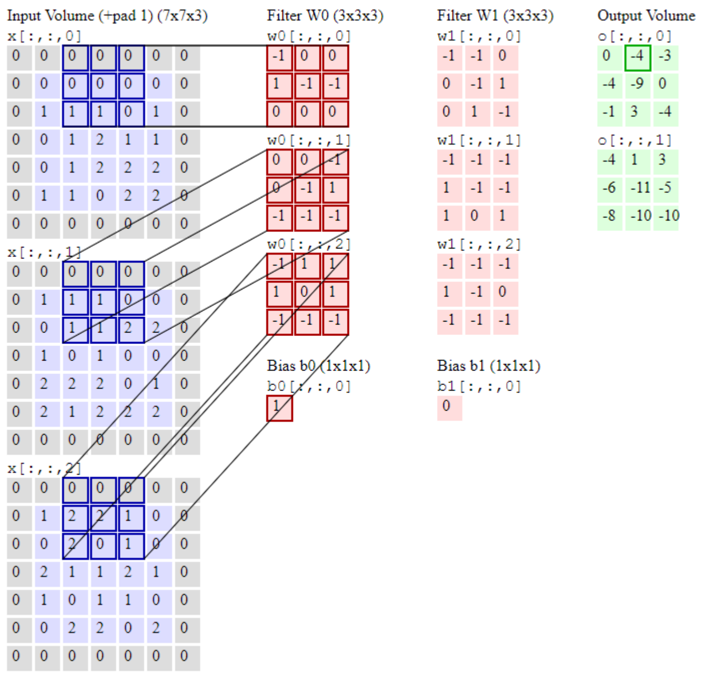


Figure 9: [4] Visualization of individual slices of the input volume and individual layers of the 2 filters as well as the 2 dimensional output corresponding to the filters.

In figure 9 we depict the convolution operation in an input image of size 7x7x3, in this case using 2 filters that are 3x3x3 .These filter matrices are the weights of a CNN corresponding to the weights discussed in the gradient descent and backpropagation sections. When the filters are moved across the input layers we calculate the dot product of the 3 layered patch and 3-layered weight matrix.[4] The distance we move the filter for each iteration is called the stride-length and is usually chosen to be (1,1) such that the mapping is 1:1 spatially. In this particular example the stride length is chosen to be (2,2) such that the filter moves across the input in 9 strides. For this reason the output has the dimension 3x3x2, 1 layer for each filter.

The second stage of a convolution layer is exactly like it sounds. It applies the activation function to each element of the output. In convolutional neural networks the rectified linear unit activation function is predominantly used,

$$\phi(x) = max(0, x) \tag{26}$$

which comparably looks like,

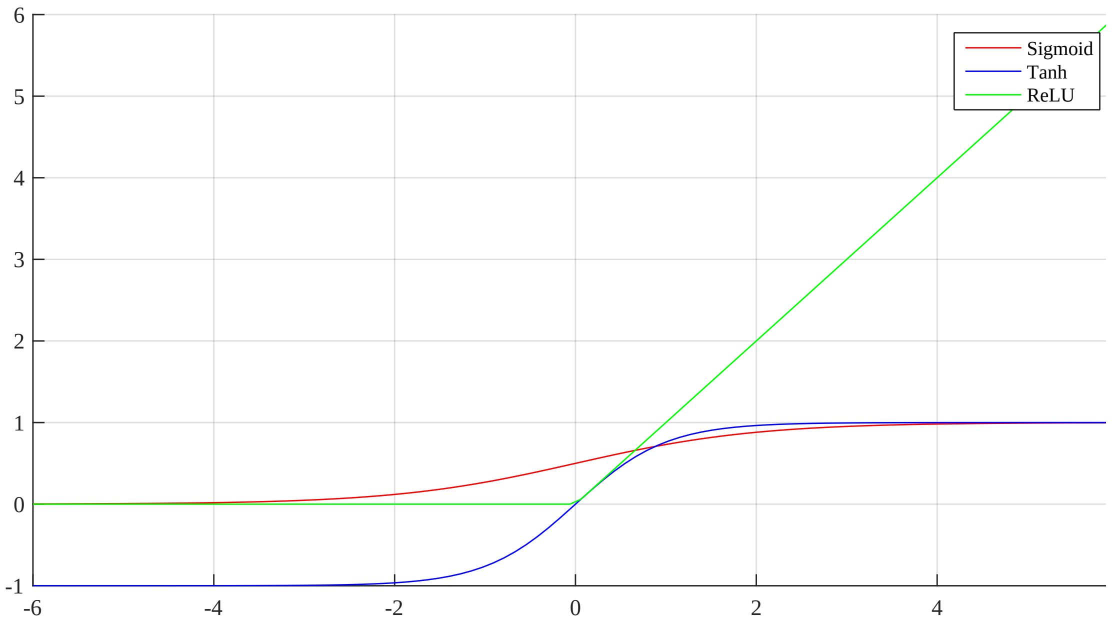


Figure 10: Plot of soft activation functions and ReLU.

The rationale behind ReLU as an activation function is that the neurons with soft activation functions stagnate/saturate, that is to say the gradient approaches 0 quickly and so does learning. Where ReLU activations do not. Furthermore, gradient computation is extremely fast on ReLU activations as it is either 0 or 1.

The last stage of a convolution layer condenses information and also improves spatial invariance by applying a pooling operation, either max pooling or average pooling. Max pooling works similarly to filters in that they can be considered patches that are moved across the layer. However, instead of computing anything it simply takes the maximum value of the patch. Two factors then determine the spatial dimensions of the output, the pool-size and the stride length. Usually the stride length is the same dimension as the pool-size such that there is no overlap. The pool-size is typically (2,2) thus the feature maps are compressed by a factor of 2 in each dimension. This can be illustrated as in figure 11,

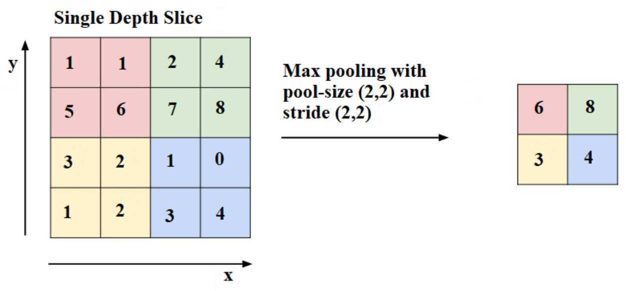


Figure 11: Simple 4x4 illustration of max pooling with 2,2 pool size and stride length

If we stack multiple layers of convolution we call it a deep convolutional neural network. Typically the number of filters used in convolution layers is ascending. This makes intuitive sense if you think about it as extracting low level features and combining them into higher level features. In the end the features should be complex enough to correspond to the constituents of classifier classes.

When we have generated a sufficiently complex stack of feature maps we need a way to turn this into a desired classification. This is sometimes achieved by simply using a deeply connected layer which connects to every single neuron in all of the feature maps. Sometimes however it is desirable to condense the information even further and in those cases global average pooling layers (GAP) are used. GAP layers, as the name suggests, simply averages the values of each feature map into a single value such that $n$ feature maps into a single n-element vector. [5] After this GAP layer, adding a deeply connected layer is less computationally costly. A typical illustration of a CNN can be see in figure 12,

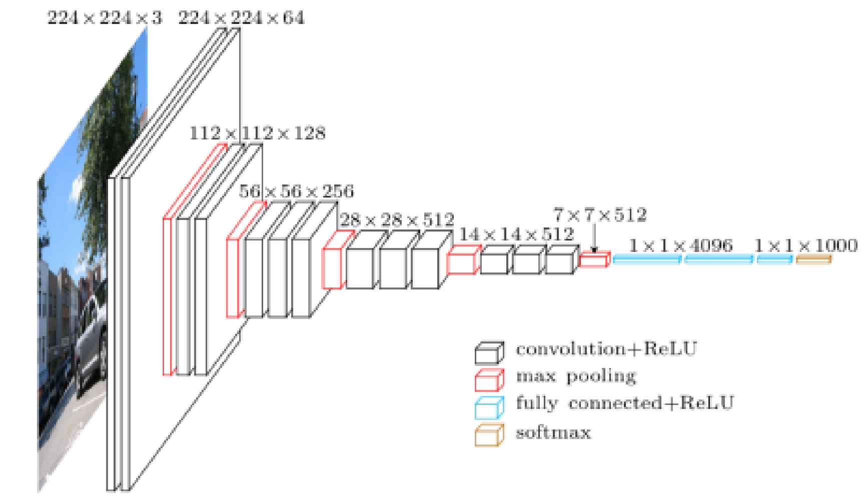


Figure 12: [5] 224x224 color image going through 5 layers of convolution and max pooling ending in 2 deeply connected layers.

## 3 Procedure

### 3.1 Data

The data modeled in this project consists of 4000 1-dimensional inputs corresponding to 4000 light curves simulated in accordance with observed stellar type distribution. Furthermore the length of the data sets varies in accordance to the TESS sky-coverage durations. Particularly 2822 long cadence light curves were simulated the remaining 1178 are short cadence. The supplied data has 14 different main categories with a bunch of subcategories. The main categories are displayed in figure 13,

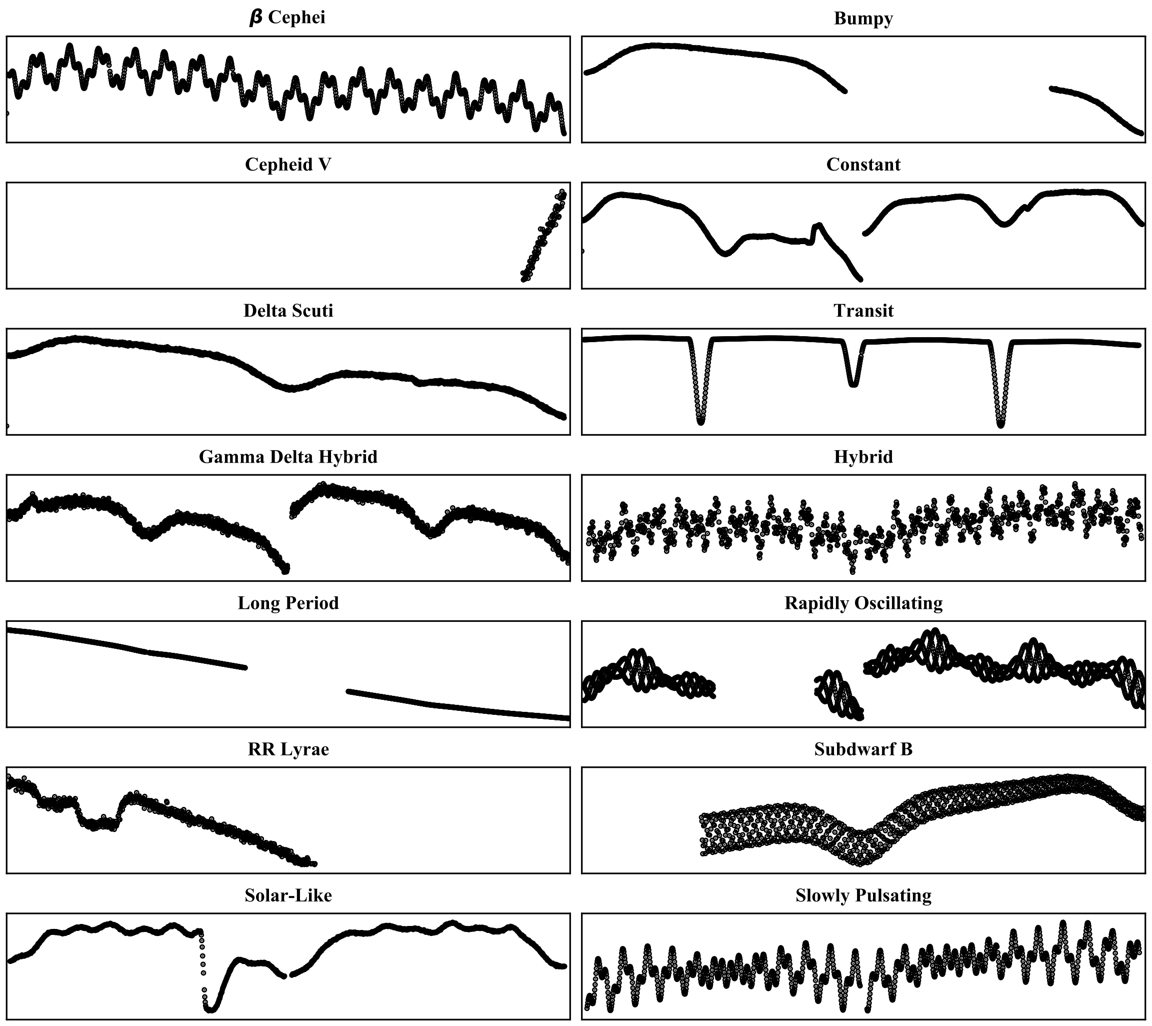


Figure 13: Light curves sampled from the 14 different categories. The unit of the x-axes is time in days and the unit of the y-axes is flux in counts, labels have been omitted because they bear no significance to the illustration of the data. They would only serve to complicate the figure unnecessarily.

## 3.2 Loading and Preprocessing

A common thing to do in machine learning is preprocessing data for various reasons. In this case the preprocessing is fivefold. Due to the variance in size of the data sets and a neural network's intrinsic need for the input size to be constant, the preprocessing pads smaller data sets with zeros to match the largest data sets. Secondly, in order to speed up convergence, we subtract the mean of the data from the data and divide by the standard deviation of the data such that,

$$x = \frac{x - mean(x)}{std(x)} \tag{27}$$

This way the data turns into values of the order of magnitude $10^0$. Thirdly we need to randomly shuffle the datasets such that we do not train a model in favor of certain data. Fourthly the data needs to be split into training data and validation data. The fraction of the data which is used for validation data needs to be representative of the collective datasets. If there are tens or hundreds of thousands of datasets, then a small fraction like 5% might be good enough to ensure this condition. In this case using 2822 datasets a split of 60% to 40% is almost necessary and is predominantly used throughout i.e,

$$X_{train} = X[1, ..., 0.6n] \tag{28}$$

$$X_{validation} = X[0.6n, ..., n] \tag{29}$$

where $X$ is the list of all datasets and n is the number of datasets. Next, in order to compute the cost function we need to generate so called label vectors or $y_{actual}$ in equation 5 the size of the label vectors are equal to the number of categories we want to classify. For all purposes the following 14 categories have been chosen as targets for the classifier.

- Beta Cephei
- Bumpy
- Cepheid Variable
- Constant
- Delta Scuti Variable
- Transit
- Gamma Doradus Variable + Delta Scuti Variable - Hybrid
- Slowly Pulsating B-type + Beta Cepheid Hybrid
- Long Period Variable
- Rapidly Oscillating AP Star
- RR Lyrae Variable
- Subdwarf B Variable
- Solar-Like
- Slowly Pulsating B-type Star

That is to say the algorithm generates a 14 element vector for each star like,

For i in n:

$$y_{actual}[i] = \begin{cases} 1 & \text{if type is } L[i] \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

Here $L[i]$ refers to the i$^{th}$ category.

## 3.3 Evolutionary Hyperparameter Tuning

In order to find a network that outperforms a simple CNN we turn to what is known as an evolutionary algorithm. Evolutionary algorithms work in the following way

- Generate the initial population, or in this case value sets for the hyperparameters of a neural network randomly.

- Evaluate the fitness of the parameters for each member in the population i.e. train the network with each set of hyperparameters for a set number of epochs or patience.
- Select the best members of the population for reproduction.
- Generate new population from best members of the previous population using mutation and crossover.
- Evaluate the fitness of the new members in the population i.e. train the network with the new sets of hyperparameters.
- Replace the worst performing members of the population with new members.

The algorithm described is a public repository by the data scientits, Joe Davison.[11]

Particularly in my case I was interested in gaining insight into a good structure of neural network for classifying my dataset. Particularly my algorithm optimizes the following parameters,

- Number of convolutional layers.
- Number of filters in convolution layers.
- Filter size in the convolution layers.
- Number of dropout layers.
- Dropout percentages.
- Number of pooling layers.
- Number of deeply connected layers.
- Whether to use flatten layer or global average pooling.

Training the network iteratively is very computationally costly so to prevent the computation from blowing up we use early stopping with a patience of 3 epochs i.e. if the network has no learning for 3 consecutive epochs it breaks and evaluates the next network. Additionally the maximum epochs is set to 50. This is actually a pretty large number of epochs but it turns out to be necessary in order to avoid putting too much preference on fast convergence. We also need a restriction on the number of dense nodes in the layers which is set to 512. Lastly, the highest number of convolutional layers is 4 and the number of dense layers is limited to 2. The genetic algorithm is set to generate 10 members in each population for 14 generations.

## 3.4 Training and Evaluation

The final part of the algorithm concerns training the actual network on the data. The network is trained over a maximum 250 epochs with a patience of 30 epochs. The performance of the network is then evaluated based on absolute accuracy, learning curve and average individual class accuracy as well as a confusion matrix. Absolute accuracy is simply,

$$Accuracy_{abs} = \frac{n}{N} \tag{31}$$

where n is the number of correct classifications in the validation set and N is the number of datasets in the validation set. Average individual class accuracy is the average of individual class accuracies.

$$Accuarcy_{avg} = \frac{1}{n_{classes}} \sum_{class}^{n_{classes}} \frac{n_{class}}{N_{class}} \tag{32}$$

where $n_{classes}$ is the number of classes, $n_{class}$ and $N_{class}$ are the number of correct classifications and total number of datasets of the respective classes. Thus if there is an equal number of datasets of each class equation (31) and (32) are equivalent. Learning curves are the validation accuracies plotted against epoch numbers as in figure 14,

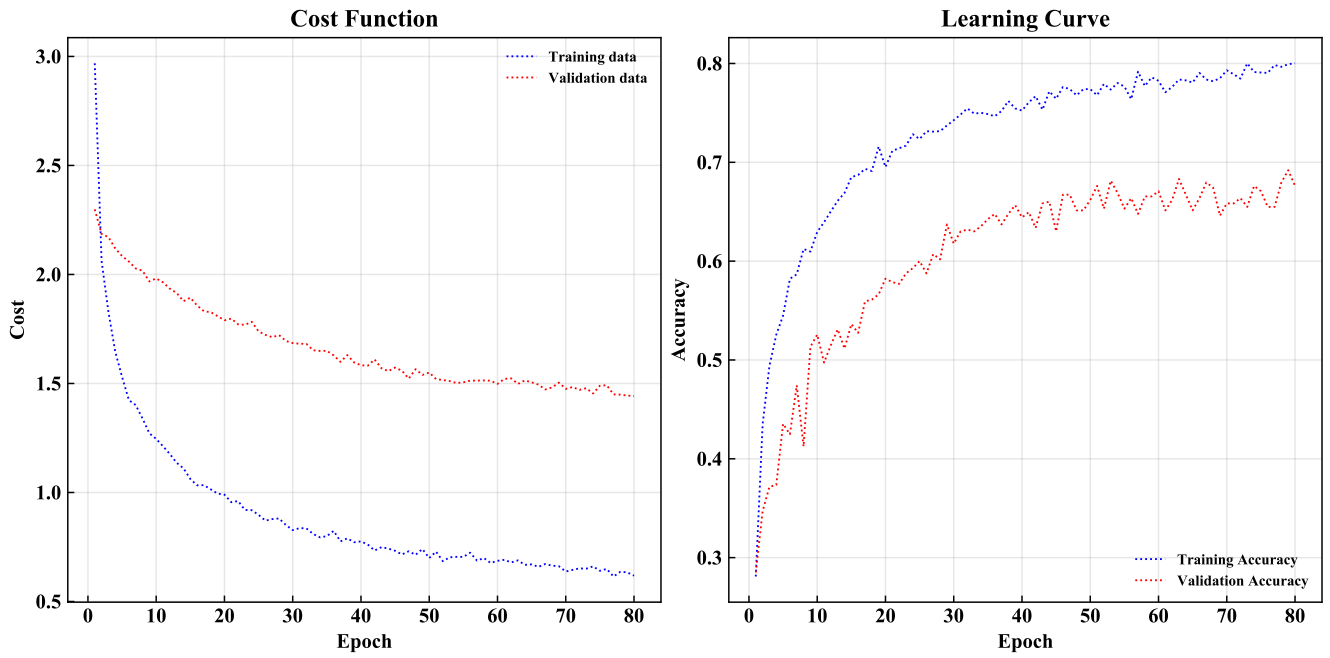
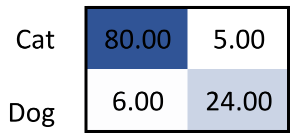


Figure 14: Example of Learning Curve and Cost function plots

In this example we see that the network is definitely converging on 70% accuracy. Learning curves are a good way of measuring whether or not the network is overfitting and if convergence is slow but consistent or fast but oscillating. The last metric and possibly most useful metric for understanding which classes are hard to classify is the confusion matrix. A confusion matrix is a matrix which contains the number of times each class is classified as each of the possible classes. That is to say the $(i, j)^{th}$ entry corresponds to the number of times the $j^{th}$ class is classified as the $i^{th}$ class.[10] Thus the diagonal entries of a confusion matrix contains the correct classifications. Below we use a 2 category example of dog/cat classification,



| | | |
|---|---|---|
| Cat | 80.00 | 5.00 |
| Dog | 6.00 | 24.00 |

Figure 15: Cat / Dog image classification confusion matrix example

In figure 15 we see that cat images were classified 80 times as cats and 5 times as dogs, conversely dog images were classified 6 times as cats and 24 times as dogs. This method of visualizing classification allows for easy recognition of tricky distinctions. In this example we see that dogs are trickier to discern from cats than cats are to discern from dogs.

## 4 Results

### 4.1 Model 1

For initial fitting of the data we use a simple convolutional neural network of the following structure,

- Input-Layer
- Dropout-Layer (50%)
- Convolution-Layer (filters: 12, size: 10)
- Dropout-Layer (50%)
- Convolution-Layer (filters: 24, size: 8)
- Dropout-Layer (50%)
- Convolution-Layer (filters: 48, size: 6)
- Dropout-Layer (50%)
- Convolution-Layer (filters: 96, size: 4)
- Global Average Pooling-Layer ()
- Dropout-Layer (50%)
- Dense-Layer (nodes: 14)

With these network parameters training with a patience of 60 epochs yields the following learning curve,

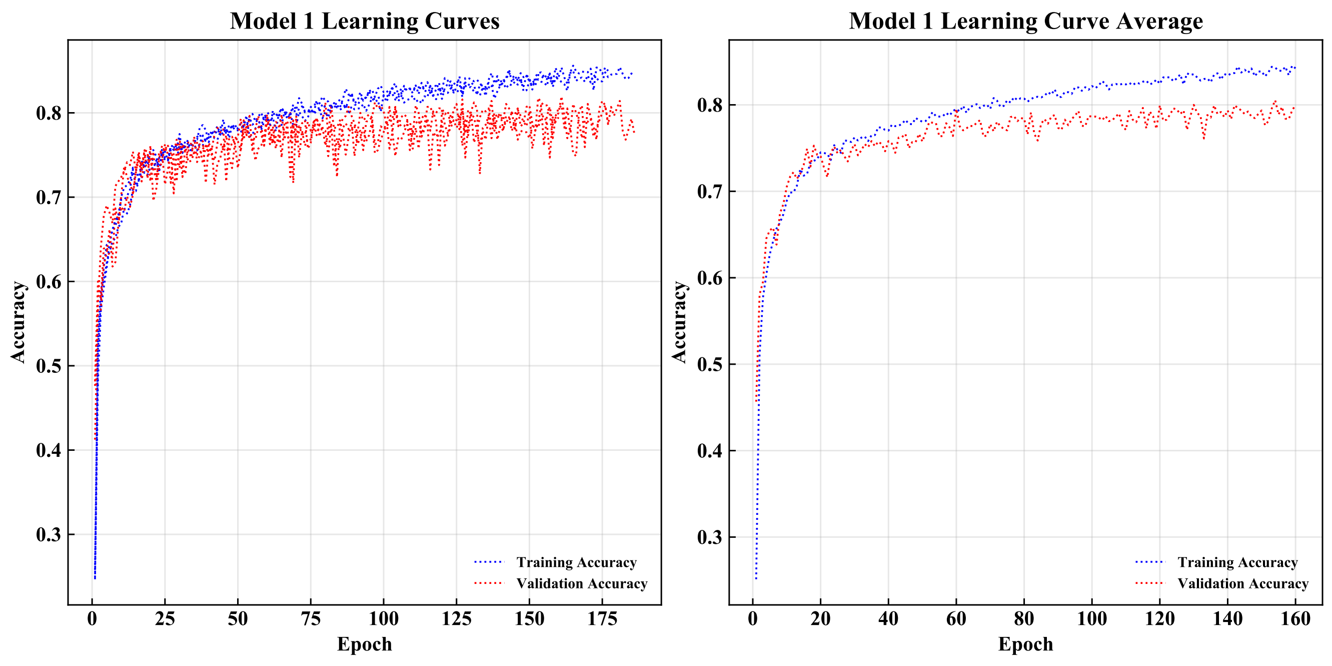


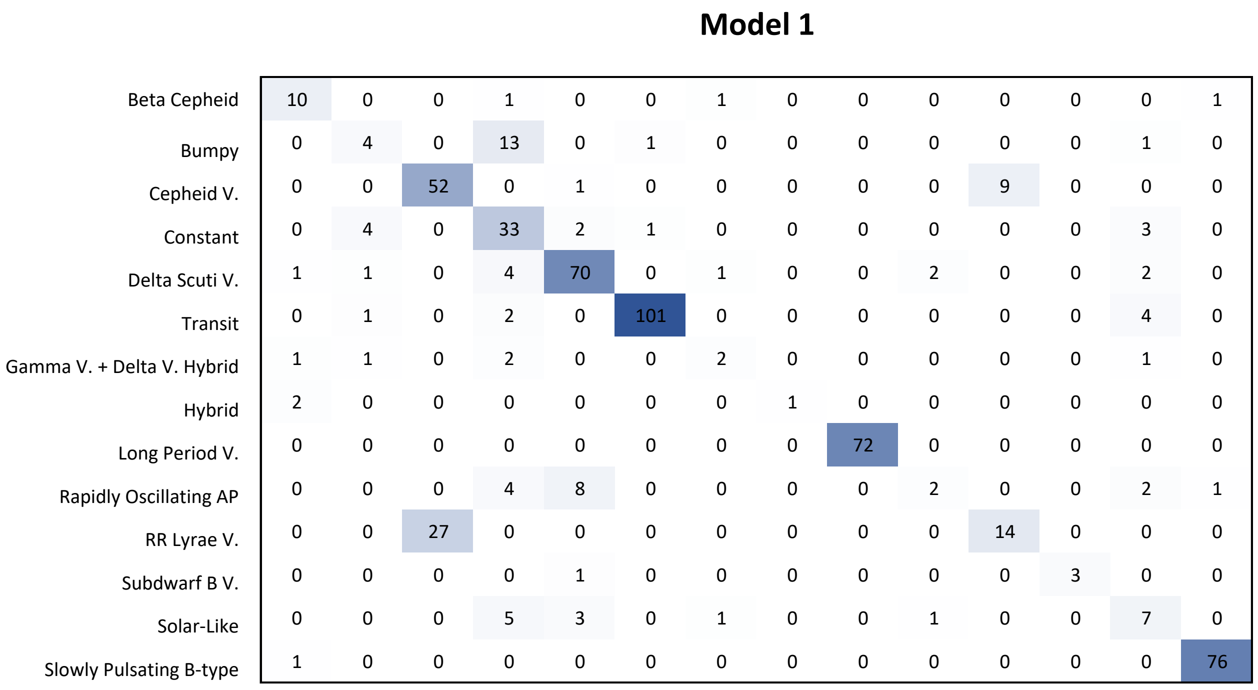
Figure 16: Learning curves for 3 retraining iterations of Model 1 on the left and the average learning curve plotted on the right.

The purpose of running 3 iterations of retraining, as plotted in figure 16, is to reduce oscillation in the learning curve thus getting a clearer picture of whether or not the model is still converging and if there is consistent overfitting as well as narrowing the standard deviation of the accuracy numbers. From these iterations we derive an absolute accuracy of 79(1)%. Table 1 shows how well each class is predicted by the model.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Average Accuracies 1-7** [%] | 77(14) | 19(3) | 84(9) | 75(6) | 86(1) | 93(4) | 27(16) |
| **Average Accuracies 8-14** [%] | 22(32) | 99.0(5) | 10(2) | 33(4) | 62(13) | 40(14) | 98(2) |

Table 1: Class accuracies averaged over 3 retraining sessions using model 1

From this the average class accuracy (each class weighted equally with no regard for the number of datasets in each class) is 59(4)%. Table 1 is useful for determining exactly how well each class can be predicted, however it does not lend much insight into what the class is being misinterpreted as. Therefore we generate an averaged confusion matrix as seen in figure 17,



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Beta Cepheid | 10 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Bumpy | 0 | 4 | 0 | 13 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Cepheid V. | 0 | 0 | 52 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 |
| Constant | 0 | 4 | 0 | 33 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 |
| Delta Scuti V. | 1 | 1 | 0 | 4 | 70 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| Transit | 0 | 1 | 0 | 2 | 0 | 101 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |
| Gamma V. + Delta V. Hybrid | 1 | 1 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Hybrid | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Long Period V. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 72 | 0 | 0 | 0 | 0 | 0 |
| Rapidly Oscillating AP | 0 | 0 | 0 | 4 | 8 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 2 | 1 |
| RR Lyrae V. | 0 | 0 | 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 |
| Subdwarf B V. | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| Solar-Like | 0 | 0 | 0 | 5 | 3 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 7 | 0 |
| Slowly Pulsating B-type | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 76 |

Figure 17: Confusion Matrix averaged over 3 training sessions of Model 1 with 60 epoch patience.

Figure 17 shows clearly that RR Lyrae Variables and Cepheid Variables are notoriously difficult for the model to distinguish. Similarly bumpy, constant and solar-like stars seem difficult to discern from one another. In order to shine light on this interchangeability we plot some samples of each star type,

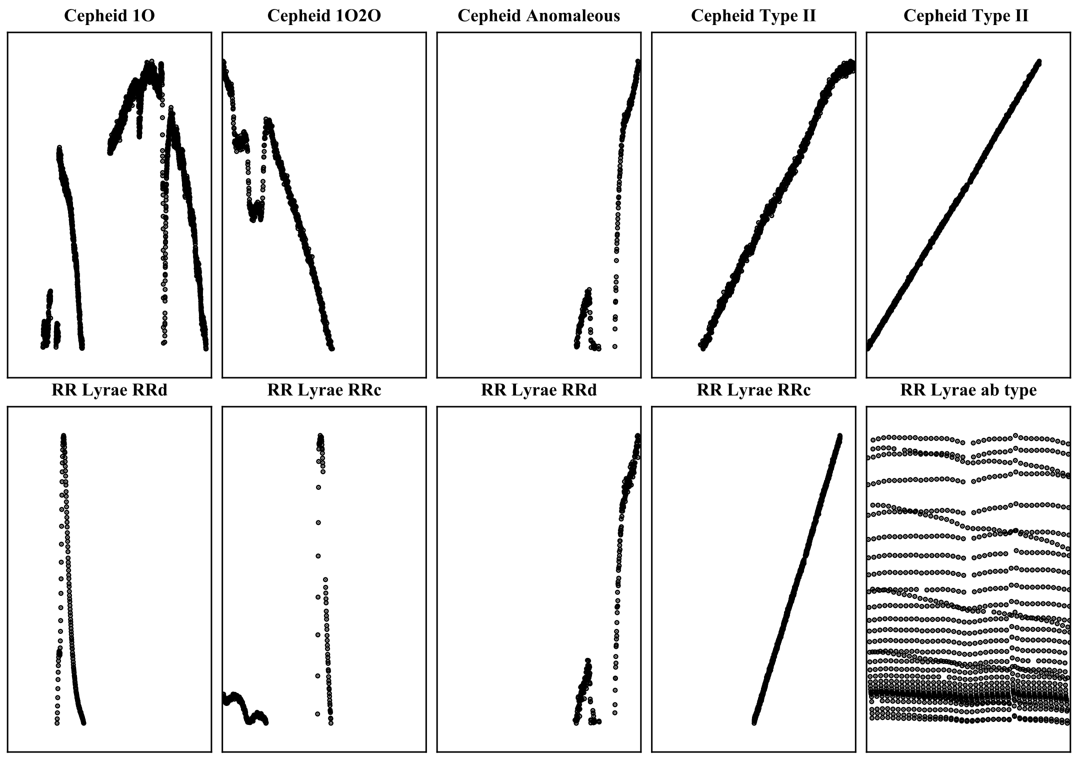


Figure 18: Random assortment of Cephei Variable and RR Lyrae light curve plots

From figure 18 it becomes immediately apparent that RR Lyrae RRc and RRd are difficult to distinguish from Cepheid Anomaleous and Cepheid Type II.

Model 1 was initially trained with a patience of 60 epochs using the cost function as a metric for learning. If instead we use accuracy as the metric we get an average accuracy over 3 retraining sessions of 81(1)% and the class accuracies are,

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Average Accuracies 1-7** [%] | 86(12) | 8(4) | 96(5) | 87(6) | 89(3) | 92(1) | 25(9) |
| **Average Accuracies 8-14** [%] | 67(24) | 97(1) | 24(17) | 33.3(5) | 67(38) | 44(6) | 99.7(5) |

Table 2: Class accuracies averaged over 3 retraining sessions using model 1 and accuracy as the metric

From table 2 we get an average class accuracy of 66(4)%.

## 4.2 Model 2

In order to develop some insight into the architecture of a model for solving the classification problem we turn to the evolutionary algorithm described in section 5.3. With these restrictions in place we find the best architecture suggested by the algorithm to have the following structure:

- Input-Layer
- Convolution-Layer (filters: 16, size: 3)
- Dropout-Layer (0%)
- Convolution-Layer (filters: 64, size: 3)

- Dropout-Layer (15%)
- Convolution-Layer (filters: 32, size: 3)
- Dropout-Layer (0%)
- Flatten-Layer ()
- Dense-Layer (nodes: 14)

Interestingly this architecture resembles a standard convolutional neural network with a pretty significant difference in that a flatten layer is favored at the end. Flatten converts the multiple layers of the last convolution layer into a single vector just like a global average pooling layer. The flatten layer does not average the elements in each layer though, it concatenates all elements of every layer into a single layer thus creating a huge feature vector. For this particular case it results in roughly $10^6$ trainable parameters. Considering that this a factor $10^3$ increase in parameters, comparing to model 1, the model is significantly slower to train. Training this model over 250 epochs over 3 training sessions yields the following learning curves,

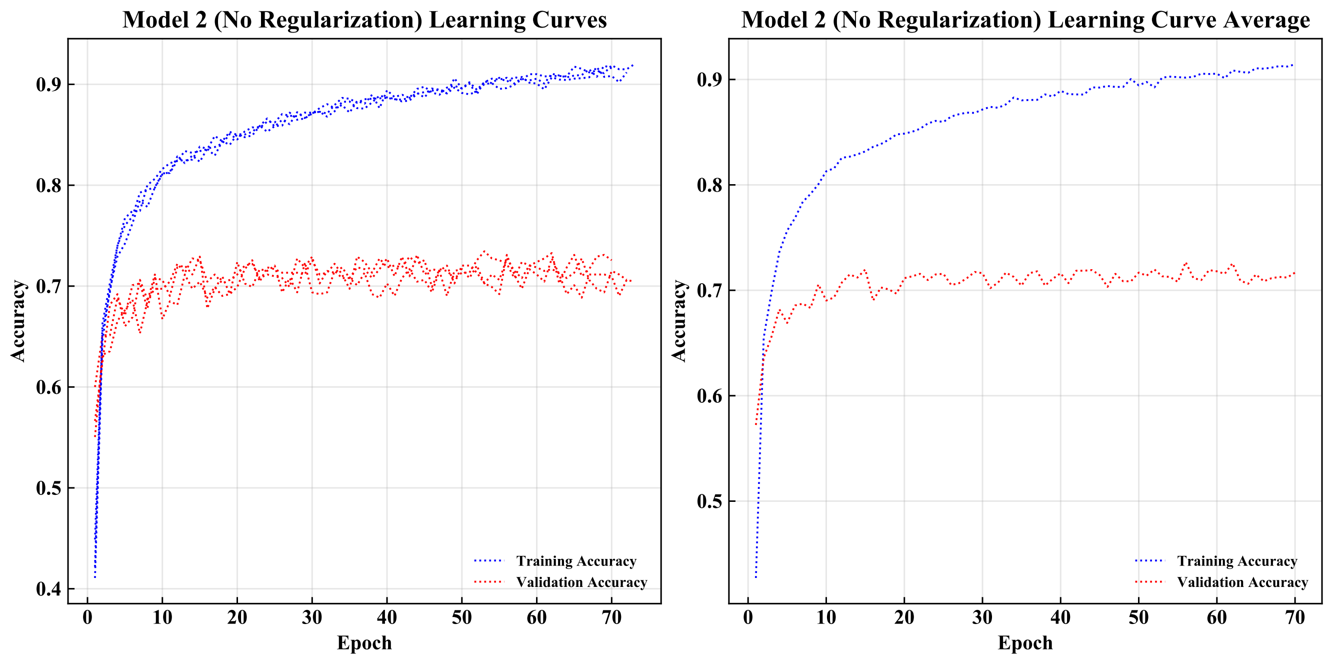


Figure 19: Learning curves from 3 retraining sessions using Model 2 without regularization.

The average accuracy derived from this model is then 71(1)% and the class accuracies can be found in table 3.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Average Accuracies 1-7** [%] | 55(5) | 26(17) | 83(4) | 75(11) | 79(7) | 83(1) | 0(0) |
| **Average Accuracies 8-14** [%] | 19(14) | 97(2) | 11(2) | 22(1) | 60(23) | 7(5) | 86(1) |

Table 3: Class accuracies averaged over 3 retraining sessions using model 1 and accuracy as the metric

From figure 3 we get an average class accuracy 50(1)%. From the learning curve the significant discrepancy between validation accuracy and training accuracy is due to overfitting. In order to reduce the overfitting we apply regularization in the form of dropout and L2 norm. The new architecture has the following structure,

- Input-Layer
- Convolution-Layer (filters: 16, size: 10)
- Dropout-Layer (50%)
- Convolution-Layer (filters: 32, size: 8)
- Dropout-Layer (65%)
- Convolution-Layer (filters: 64, size: 6)
- Dropout-Layer (50%)
- Convolution-Layer (filters: 128, size: 4)
- Dropout-Layer (50%)
- Flatten-Layer ()
- Dense-Layer (nodes: 14)

Training this model over 3 iterations yields the training curve in figure 20.

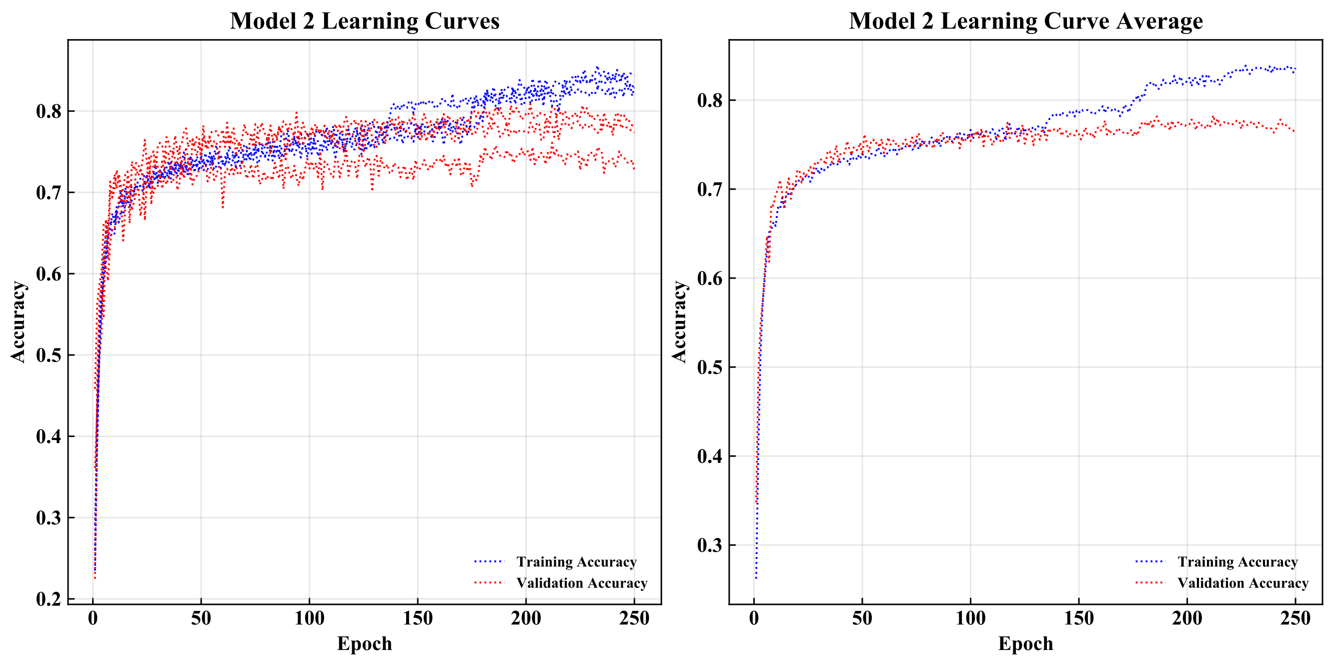
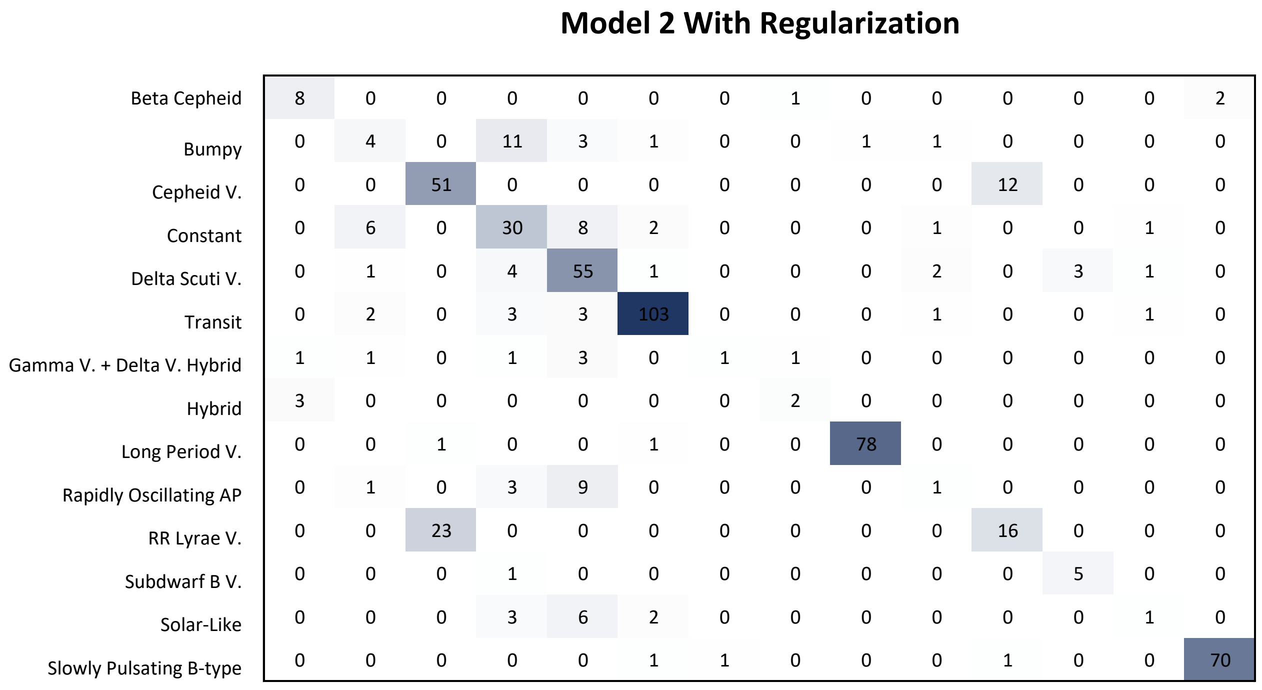


Figure 20: Model 2 (Regularized) Learning curves on the left with the averages plotted on the right.

The regularization in addition to the convolution layer modifications in the updated model 2 have reduced overfitting significantly. The accuracy from the update model 2 is 78.5(15)%, the class accuracies are shown in table 4 and the confusion matrix can be seen in figure 21

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Average Accuracies 1-7** [%] | 48(27) | 15(16) | 93(3) | 83(5) | 89(2) | 94(1) | 14(10) |
| **Average Accuracies 8-14** [%] | 11(16) | 99(1) | 9(7) | 21(4) | 58(12) | 22(2) | 99.3(5) |

Table 4: Class accuracies averaged over 3 retraining sessions using model 2 with regularization


Model 2 With Regularization

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Beta Cepheid | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 |
| Bumpy | 0 | 4 | 0 | 11 | 3 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| Cepheid V. | 0 | 0 | 51 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 |
| Constant | 0 | 6 | 0 | 30 | 8 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| Delta Scuti V. | 0 | 1 | 0 | 4 | 55 | 1 | 0 | 0 | 0 | 2 | 0 | 3 | 1 | 0 |
| Transit | 0 | 2 | 0 | 3 | 3 | 103 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| Gamma V. + Delta V. Hybrid | 1 | 1 | 0 | 1 | 3 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hybrid | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| Long Period V. | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 78 | 0 | 0 | 0 | 0 | 0 |
| Rapidly Oscillating AP | 0 | 1 | 0 | 3 | 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| RR Lyrae V. | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 0 | 0 | 0 |
| Subdwarf B V. | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 |
| Solar-Like | 0 | 0 | 0 | 3 | 6 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Slowly Pulsating B-type | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 70 |

Figure 21: Average confusion matrix over 3 retraining iterations using model 2 with regularization.

The average class accuracy is 54(2)%. From these figures we see obvious improvements but not enough to match the accuracy provided by model 1.

## 4.3 Model 1 Additional Data

Considering the confusion matrix in figure 17 it is pretty obvious that certain categories have too little supporting data sets to actually say with any confidence how well those classes can be predicted by the model. For this reason, an additional 15.000 light curves (11.307 long cadence light curves) have been simulated with an equal distribution in terms of the main categories (not the subcategories),

- A + F Classical:
  - Delta Scuti Variable
  - Gamma Doradus Variable
  - Gamma Doradus Variable + Delta Scuti Variable - Hybrid
  - Rapidly Oscillating AP Stars
  - Bumpy
- O + B Classical:
  - Slowly Pulsating B-type Stars

  - Beta Cepheid
  - Subdwarf B Variable
  - Slowly Pulsating B-type + Beta Cepheid Hybrid
- RR-Lyrae
  - Fundamental-Mode
  - First Overtone
  - Double-Mode
- Cepheid
  - Fundamental-Mode
  - First Overtone
  - First and Second Overtone
  - Fundamental and First Overtone
  - Type II
  - Anomalous
- Long Period Variables
  - Mira
  - Semi-regular Variables
- Transit
- Solar Like Oscillators

Training model 1 on this network three. Times gives us the learning curve in figure 22.

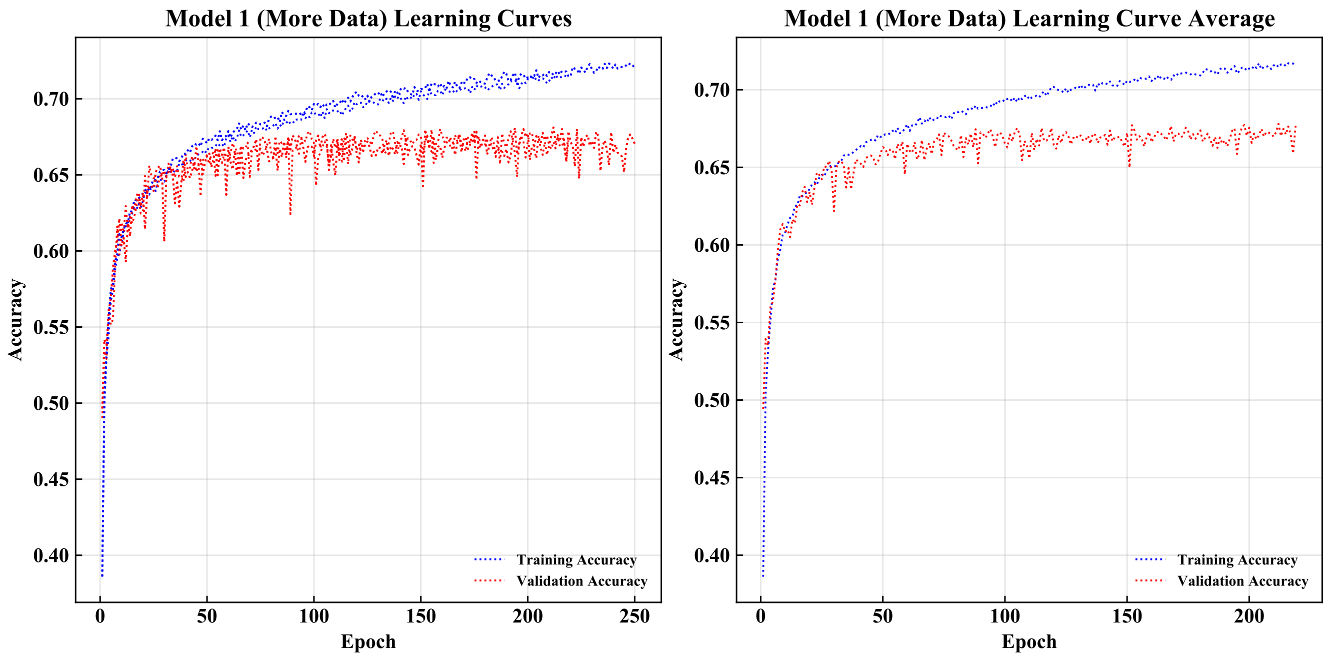


Figure 22: Learning curve for model 1 retrained 3 times on the additional data.

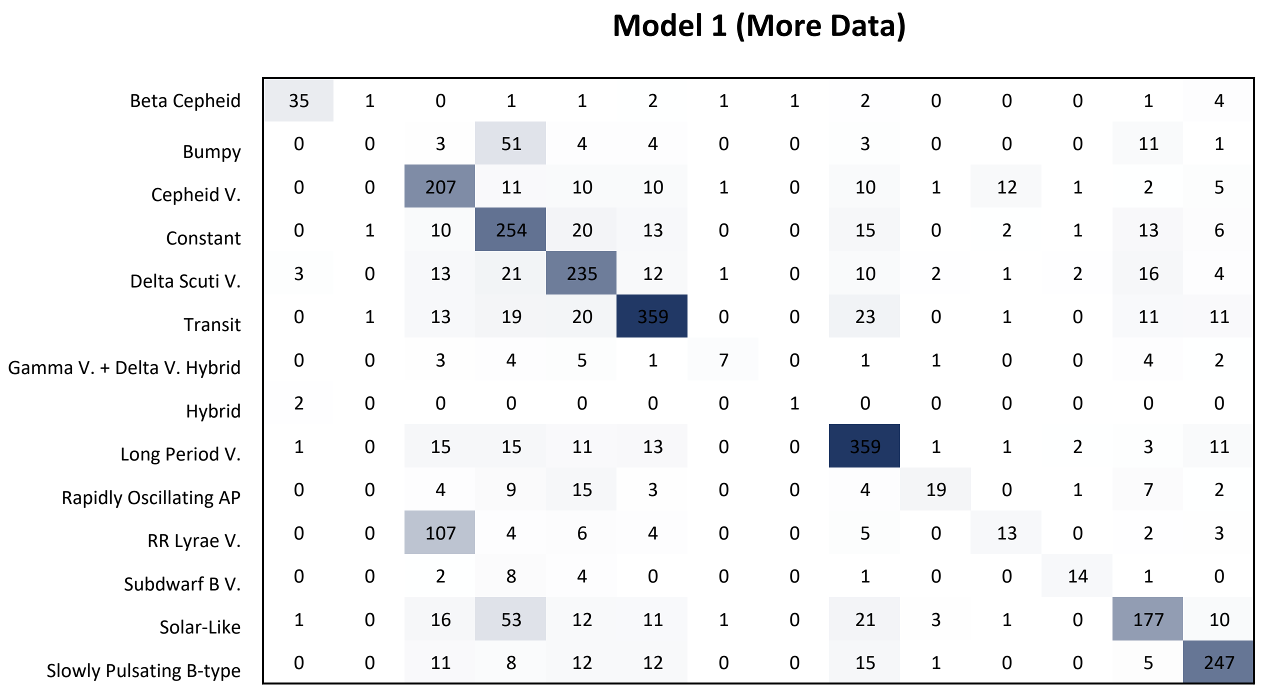
The average accuracy gathered from this is 68.0(2)% and the average class accuracy is found to be 52(3)%.

The class accuracies are shown in table 5.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Average Accuracies 1-7** [%] | 72(4) | 0(0) | 77(1) | 76(3) | 73(3) | 78(3) | 24(4) |
| **Average Accuracies 8-14** [%] | 28(21) | 83.3(5) | 30(6) | 9.3(5) | 46(9) | 58(2) | 80(1) |

Table 5: Class accuracies averaged over 3 retraining sessions using model 1 with additional data

These class accuracies are a lot more reliable since the sheer number of datasets has increased and thus the validation set is more complete in terms of representation of smaller classes. Looking at the confusion matrix in figure 23 we see trends similar to the confusion matrices of figures 17 and 21, however solar-like oscillation has improved significantly with the added data. We see that the added diversity has increased confidence in our class accuracies but also caused a decrease in overall accuracy.


Model 1 (More Data)

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Beta Cepheid | 35 | 1 | 0 | 1 | 1 | 2 | 1 | 1 | 2 | 0 | 0 | 0 | 1 | 4 |
| Bumpy | 0 | 0 | 3 | 51 | 4 | 4 | 0 | 0 | 3 | 0 | 0 | 0 | 11 | 1 |
| Cepheid V. | 0 | 0 | 207 | 11 | 10 | 10 | 1 | 0 | 10 | 1 | 12 | 1 | 2 | 5 |
| Constant | 0 | 1 | 10 | 254 | 20 | 13 | 0 | 0 | 15 | 0 | 2 | 1 | 13 | 6 |
| Delta Scuti V. | 3 | 0 | 13 | 21 | 235 | 12 | 1 | 0 | 10 | 2 | 1 | 2 | 16 | 4 |
| Transit | 0 | 1 | 13 | 19 | 20 | 359 | 0 | 0 | 23 | 0 | 1 | 0 | 11 | 11 |
| Gamma V. + Delta V. Hybrid | 0 | 0 | 3 | 4 | 5 | 1 | 7 | 0 | 1 | 1 | 0 | 0 | 4 | 2 |
| Hybrid | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Long Period V. | 1 | 0 | 15 | 15 | 11 | 13 | 0 | 0 | 359 | 1 | 1 | 2 | 3 | 11 |
| Rapidly Oscillating AP | 0 | 0 | 4 | 9 | 15 | 3 | 0 | 0 | 4 | 19 | 0 | 1 | 7 | 2 |
| RR Lyrae V. | 0 | 0 | 107 | 4 | 6 | 4 | 0 | 0 | 5 | 0 | 13 | 0 | 2 | 3 |
| Subdwarf B V. | 0 | 0 | 2 | 8 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 14 | 1 | 0 |
| Solar-Like | 1 | 0 | 16 | 53 | 12 | 11 | 1 | 0 | 21 | 3 | 1 | 0 | 177 | 10 |
| Slowly Pulsating B-type | 0 | 0 | 11 | 8 | 12 | 12 | 0 | 0 | 15 | 1 | 0 | 0 | 5 | 247 |

Figure 23: Confusion Matrix averaged over 3 training sessions using Model 1 with additional data.

## 4.4 Model 3 Additional Data

In convolutional neural networks results on ImageNet the best networks have the same things in common; they all use very deep architectures. CNNs with many layers eventually experience saturation of accuracy ie. adding additional layers does not improve accuracy. The accuracy has been shown to decrease rapidly after saturation. This is known as the degradation problem.[8] In order to increase the depth of a CNN without degradation, a team at microsoft developed a technique called residual neural networks.[8] The residual neural network combats the degredation problem by allowing the network to learn to skip certain convolution layers or place less emphasis on them.

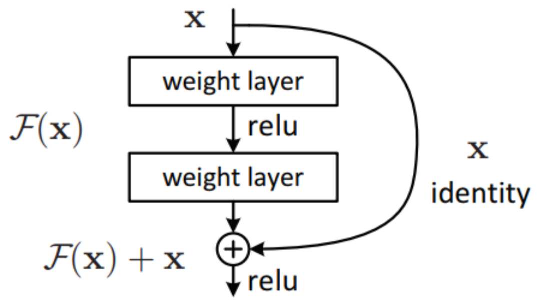

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 24: [8] Illustration of a residual building block

Figure 24 shows how shortcuts in the network can be integrated by adding merge layers that sums the output of an arbitrary number of layers and the output from the layer before those. By integrating residual blocks in our network, we achieve 8 layers of convolution with the following structure,

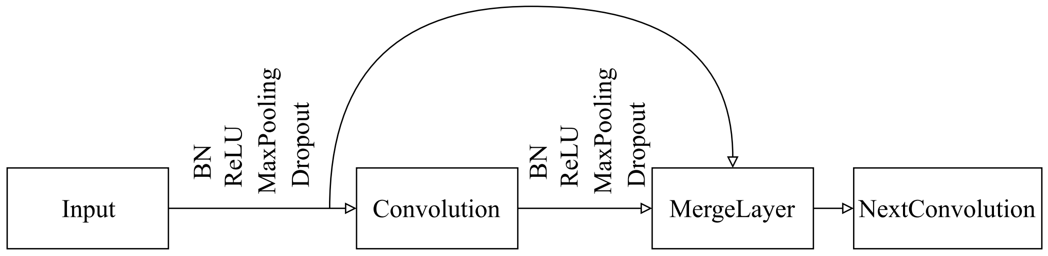


Figure 25: Visualization of the residual structure of the individual layers in Model 3. BN is an abbreviation for batch normalization.

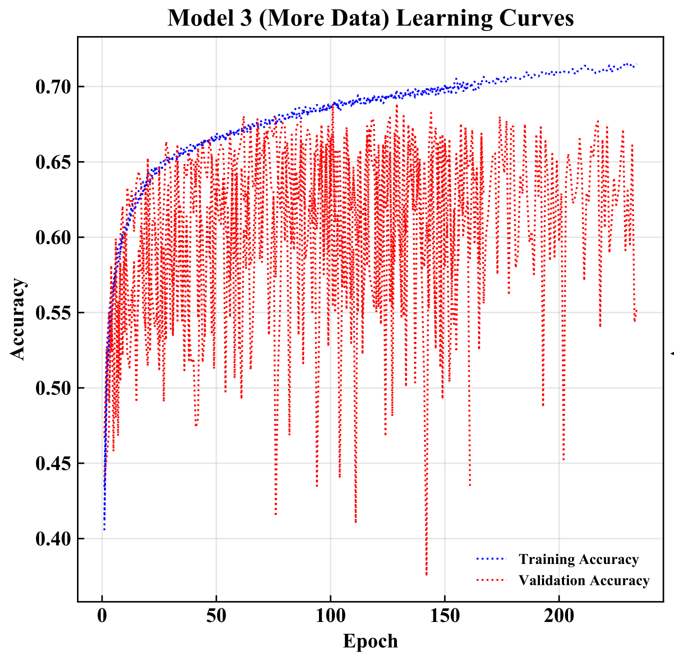
The model uses the following number of filters, (16, 16, 32, 32, 64, 64, 128, 256), and filter sizes, (16, 14, 14, 12, 12, 10, 10, 8). Max pooling is only used on every second layer to prevent too much loss of information. From the residual CNN we get an average accuracy of 68(1)% the learning curves are shown in figure 26.

The class accuracies are shown in table 6.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Average Accuracies 1-7** [%] | 84(6) | 0(0) | 80(1) | 72(1) | 69(1) | 81(3) | 1(1) |
| **Average Accuracies 8-14** [%] | 0(0) | 83(2) | 42(7) | 6(2) | 57(13) | 59(1) | 80(2) |

Table 6: Class accuracies averaged over 3 retraining sessions using model 1 with additional data.

The average class accuracy using the residual model is 51(2)%. These results are remarkably similar to the Model 1 results so it seems reasonable to suggest that in the convolutional neural network case the accuracy is close to saturated.



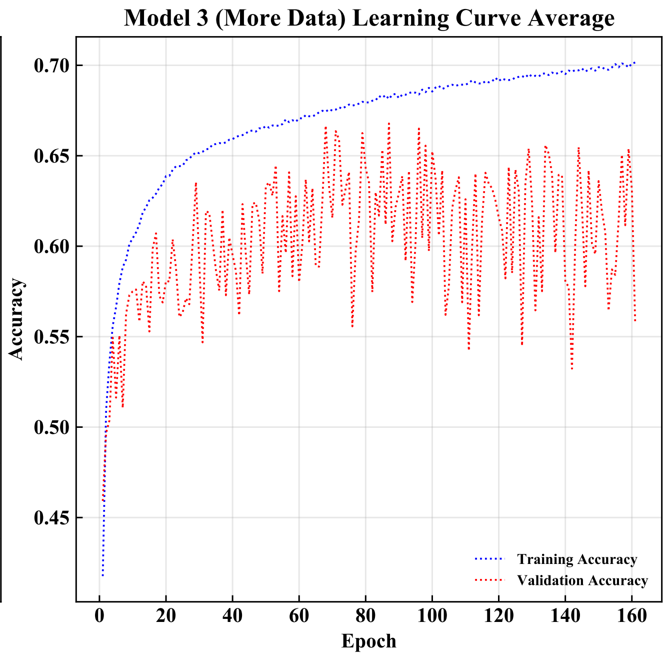


Figure 26: Learning curve from 3 retraining sessions using the residual model and the additional data.

## 5 Discussion

For unevenly distributed data a simple 4 layer CNN structure proved superior to the model generated through the evolutionary algorithm. The evolutionary algorithm inherently favors fast convergence because the learning patience was set to 3 epochs and the learning rate was fixed. In figure 19 the fast convergence is evident and it seems reasonable that this model was favored during evolution.

In the case of the more diversely distributed datasets we find from figures 23 and tables 5, 6 that the class accuracies initially provided in table 2 are misleading due to the uneven distribution of datasets. We find from model 3 that the accuracy is seemingly invariant to the increase in CNN depth. This suggests that either CNN is a model architecture or that the accuracy is limited by the nature of the data and the choice of categories in the classifier.

One plausible bottleneck is that light curves in and of themselves do not necessarily carry enough information to distinguish one star from another. Indeed due to the many parameters responsible for the oscillations in star luminosity two different categories of stars might produce very similar light curves. Information like the spectrum of the stars would certainly improve classification confidence.

There is a fundamental issue with machine learning algorithms in the lack of concrete solutions or heuristics. The number of possible combinations of neural networks is infinite and thus systematic testing can only be done in a well defined parameter space with discrete intervals. These kinds of tests are sometimes helpful but are often extremely expensive computationally.

Lastly, in this project the choice of classification categories is rather arbitrary. In hindsight one might have chosen the categories sorted by the number of datasets in each category. A problem with the 14 categories chosen in this project is that 5 of the categories haver disproportionately poor representation in the data and hence the overall classification accuracy suffers.

## 6 Conclusion

Through a 4 layer convolutional neural network structure (Model 1), an accuracy of 81(1)% was achieved with above 85% accuracy on 7 of the 14 target classes.

Flatten layers and L2-norm regularization in Model 2, made no improvements on the results of Model 1, achieving an accuracy of 78.5(15)% and an average class accuracy of 54(2)%.

With more diversely distributed data Model 1, achieved an accuracy of 68.0(2)% and an average class accuracy of 52(3)%. In an attempt to imitate the success of residual neural networks in object recognition, residual neural network structure with 8 layers was used to achieve accuracy of 68(1)% and an average class accuracy of 51(2)%. Both of these models achieved an accuracy above 70% for 7 of the 14 classes, with an improved solar-like oscillation classification accuracy of 58(2)%, as opposed to 40(14)% achieved without additional light curves.

## Acknowledgements

In any machine learning problem data is at the core so thank you to Mikkel N. Lund for performing the light curve simulations for this project. The evolutionary algorithm was taken from a public repository by Joe Davison [11]. Thanks to Sara for proofreading the project. Lastly I would like to thank Kuldeep Verma for his patience and time in explaining much of the subject matter to me and for the insightful discussions about my models.

## References

[1] Nielsen, M. (2017). Neural Networks and Deep Learning. Retrieved from http://neuralnetworksanddeeplearning.com/chap1.html

[2] Nielsen, M. (2017). Neural Networks and Deep Learning. Retrieved from http://neuralnetworksanddeeplearning.com/chap2.html

[3] M. Hawkins, D. (2004). The Problem of Overfitting. Journal of Chemical Information and Modelling, 44(1), 1-12.

[4] CS231n Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved from http://cs231n.github.io/convolutional-networks/

[5] Cook, A. (2017, April 9). Global Average Pooling Layers for Object Localization. Retrieved from https://alexisbcook.github.io/2017/global-average-pooling-layers-for-object-localization/

[6] N. Lund, M., Handberg, R., Kjeldsen, H., J. Chaplin, W., & Christian-Dalsgaard, J. (2016, October 9). Data preperation for asteroseismology with TESS. arXiv, 160, 1. Retrieved from https://arxiv.org/pdf/1610.02702.pdf

[7] TESS. (n.d.). Retrieved from https://tess.mit.edu/data/

[8] He, K., Zhang, X., Ren, S., & Sun, J. (2015, December 10). Deep Residual Learning for Image Recognition. arXiv, 1-2. Retrieved from https://arxiv.org/pdf/1512.03385.pdf

[9] Hon, M., Stello, D., & C. Zinn, J. (2018, April 18). Detecting Solar-Like Oscillations in Red Giants with Deep Learning. arXiv, 1. Retrieved from https://arxiv.org/pdf/1804.07495.pdf

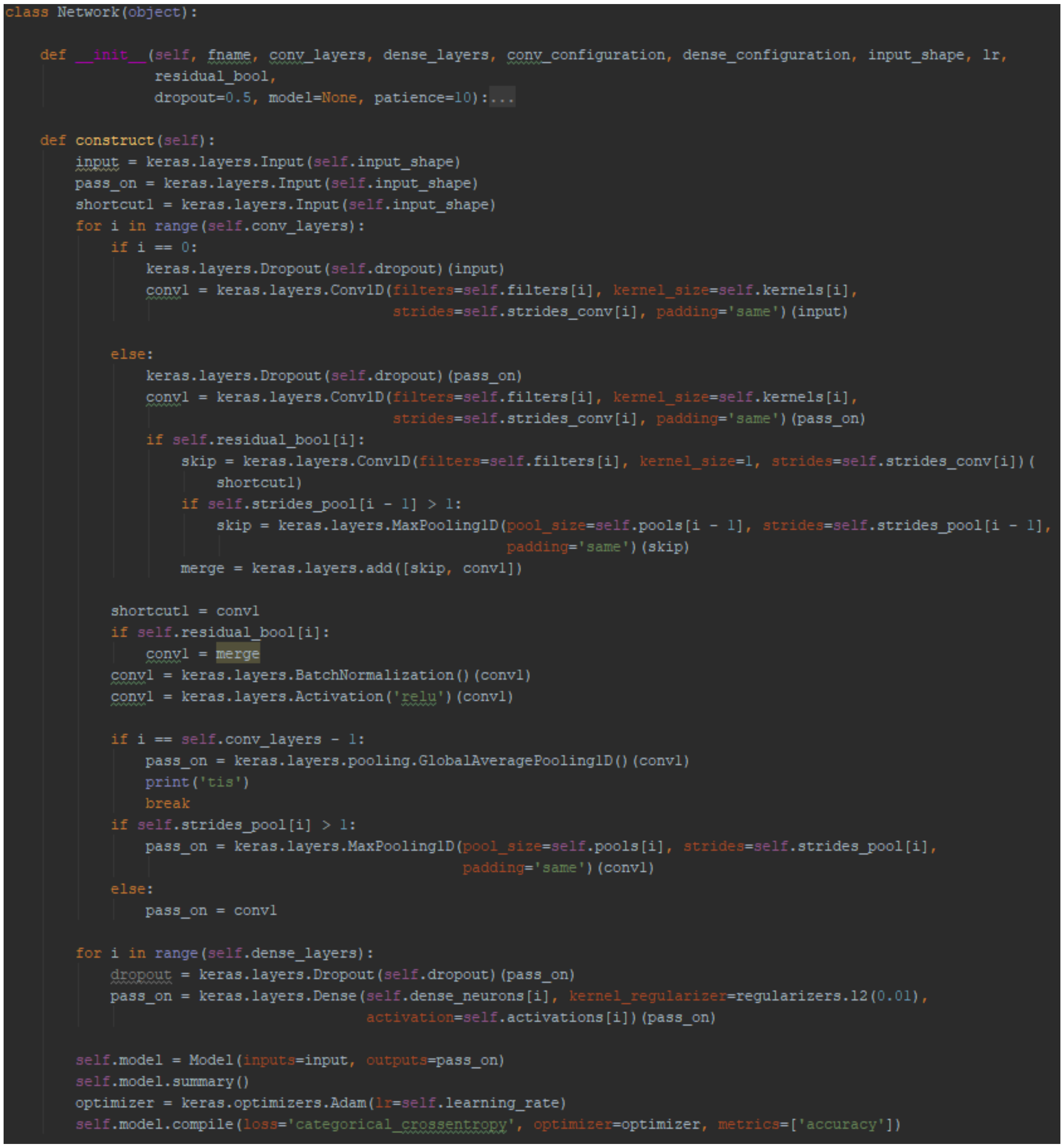
[10] M. W. Powers, D. (2007). Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Adelaide: School of Informatics and Engineering. Retrieved from

[11] Davison, J. (2017). DEvol - Deep Neural Network Evolution. Retrieved April 3, 2018, from https://github.com/joeddav/devol

# Appendices

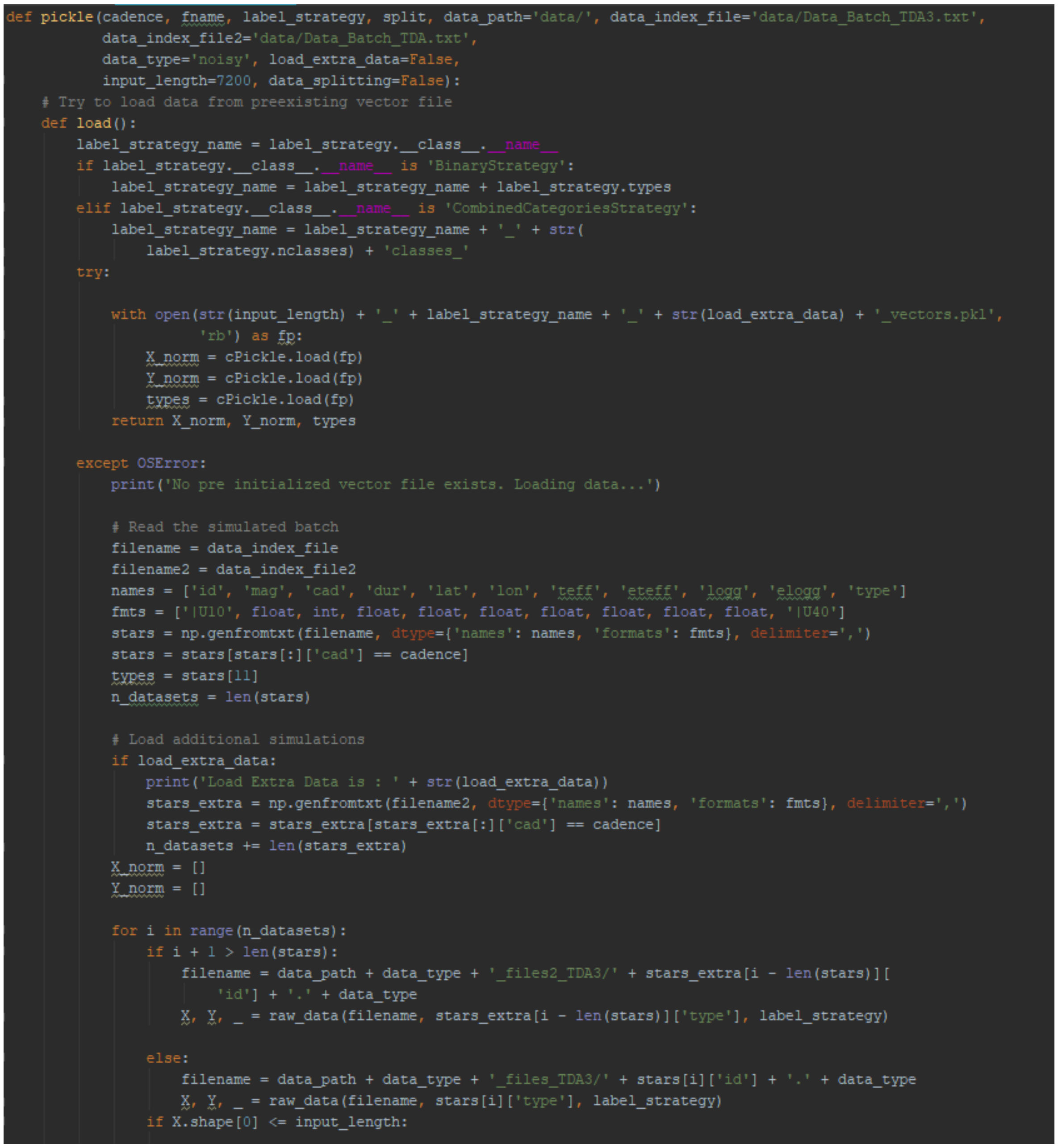
In order to make the machine learning analysis part of this project more transparent a few key components of the python project code has been included below.

### Adjustable Residual Neural Network Python implementation

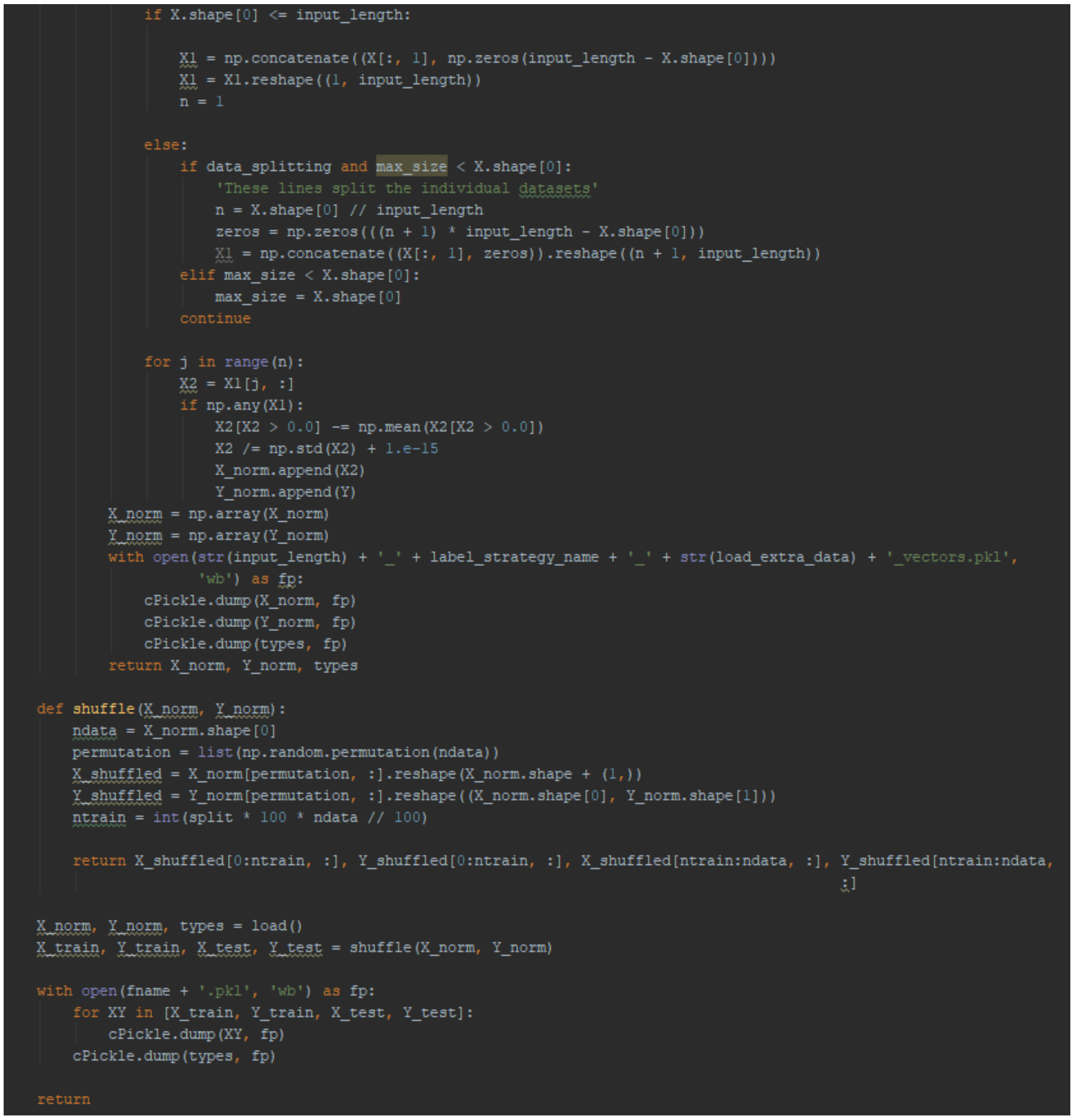


```python
class Network(object):

    def __init__(self, fname, conv_layers, dense_layers, conv_configuration, dense_configuration, input_shape, lr,
                 residual_bool,
                 dropout=0.5, model=None, patience=10):...

    def construct(self):
        input = keras.layers.Input(self.input_shape)
        pass_on = keras.layers.Input(self.input_shape)
        shortcut1 = keras.layers.Input(self.input_shape)
        for i in range(self.conv_layers):
            if i == 0:
                keras.layers.Dropout(self.dropout)(input)
                conv1 = keras.layers.Conv1D(filters=self.filters[i], kernel_size=self.kernels[i],
                                            strides=self.strides_conv[i], padding='same')(input)

            else:
                keras.layers.Dropout(self.dropout)(pass_on)
                conv1 = keras.layers.Conv1D(filters=self.filters[i], kernel_size=self.kernels[i],
                                            strides=self.strides_conv[i], padding='same')(pass_on)
                if self.residual_bool[i]:
                    skip = keras.layers.Conv1D(filters=self.filters[i], kernel_size=1, strides=self.strides_conv[i])(
                        shortcut1)
                    if self.strides_pool[i - 1] > 1:
                        skip = keras.layers.MaxPooling1D(pool_size=self.pools[i - 1], strides=self.strides_pool[i - 1],
                                                         padding='same')(skip)
                    merge = keras.layers.add([skip, conv1])

            shortcut1 = conv1
            if self.residual_bool[i]:
                conv1 = merge
            conv1 = keras.layers.BatchNormalization()(conv1)
            conv1 = keras.layers.Activation('relu')(conv1)

            if i == self.conv_layers - 1:
                pass_on = keras.layers.pooling.GlobalAveragePooling1D()(conv1)
                print('tis')
                break
            if self.strides_pool[i] > 1:
                pass_on = keras.layers.MaxPooling1D(pool_size=self.pools[i], strides=self.strides_pool[i],
                                                    padding='same')(conv1)
            else:
                pass_on = conv1

        for i in range(self.dense_layers):
            dropout = keras.layers.Dropout(self.dropout)(pass_on)
            pass_on = keras.layers.Dense(self.dense_neurons[i], kernel_regularizer=regularizers.l2(0.01),
                                         activation=self.activations[i])(pass_on)

        self.model = Model(inputs=input, outputs=pass_on)
        self.model.summary()
        optimizer = keras.optimizers.Adam(lr=self.learning_rate)
        self.model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

## Data loading implementation

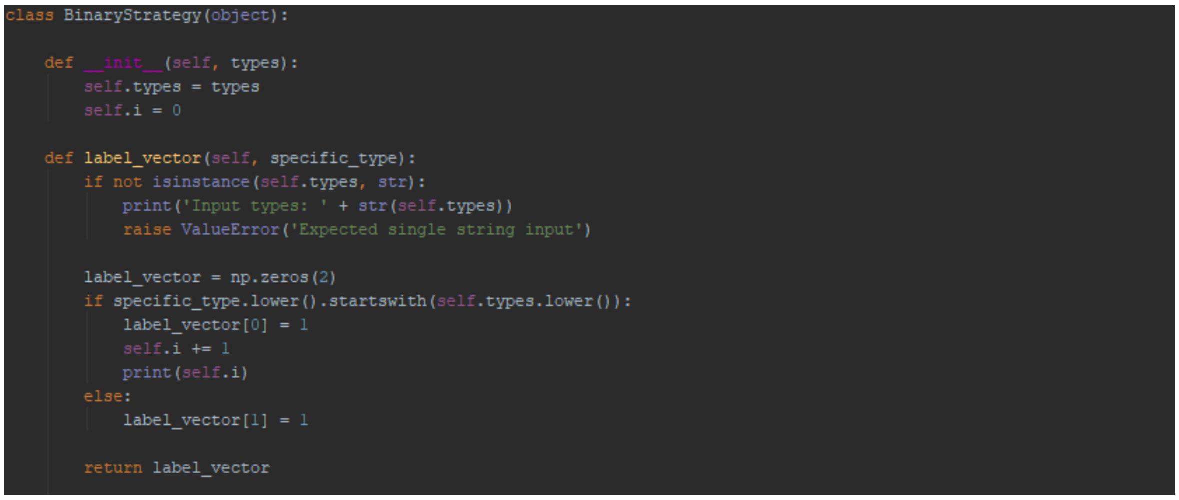


```python
def pickle(cadence, fname, label_strategy, split, data_path='data/', data_index_file='data/Data_Batch_TDA3.txt',
           data_index_file2='data/Data_Batch_TDA.txt',
           data_type='noisy', load_extra_data=False,
           input_length=7200, data_splitting=False):
    # Try to load data from preexisting vector file
    def load():
        label_strategy_name = label_strategy.__class__.__name__
        if label_strategy.__class__.__name__ is 'BinaryStrategy':
            label_strategy_name = label_strategy_name + label_strategy.types
        elif label_strategy.__class__.__name__ is 'CombinedCategoriesStrategy':
            label_strategy_name = label_strategy_name + '_' + str(
                label_strategy.nclasses) + 'classes_'
        try:

            with open(str(input_length) + '_' + label_strategy_name + '_' + str(load_extra_data) + '_vectors.pkl',
                      'rb') as fp:
                X_norm = cPickle.load(fp)
                Y_norm = cPickle.load(fp)
                types = cPickle.load(fp)
            return X_norm, Y_norm, types

        except OSError:
            print('No pre initialized vector file exists. Loading data...')

            # Read the simulated batch
            filename = data_index_file
            filename2 = data_index_file2
            names = ['id', 'mag', 'cad', 'dur', 'lat', 'lon', 'teff', 'eteff', 'logg', 'elogg', 'type']
            fmts = ['|U10', float, int, float, float, float, float, float, float, float, '|U40']
            stars = np.genfromtxt(filename, dtype={'names': names, 'formats': fmts}, delimiter=',')
            stars = stars[stars[:]['cad'] == cadence]
            types = stars[11]
            n_datasets = len(stars)

            # Load additional simulations
            if load_extra_data:
                print('Load Extra Data is : ' + str(load_extra_data))
                stars_extra = np.genfromtxt(filename2, dtype={'names': names, 'formats': fmts}, delimiter=',')
                stars_extra = stars_extra[stars_extra[:]['cad'] == cadence]
                n_datasets += len(stars_extra)
            X_norm = []
            Y_norm = []

            for i in range(n_datasets):
                if i + 1 > len(stars):
                    filename = data_path + data_type + '_files2_TDA3/' + stars_extra[i - len(stars)][
                        'id'] + '.' + data_type
                    X, Y, _ = raw_data(filename, stars_extra[i - len(stars)]['type'], label_strategy)

                else:
                    filename = data_path + data_type + '_files_TDA3/' + stars[i]['id'] + '.' + data_type
                    X, Y, _ = raw_data(filename, stars[i]['type'], label_strategy)
                if X.shape[0] <= input_length:
```

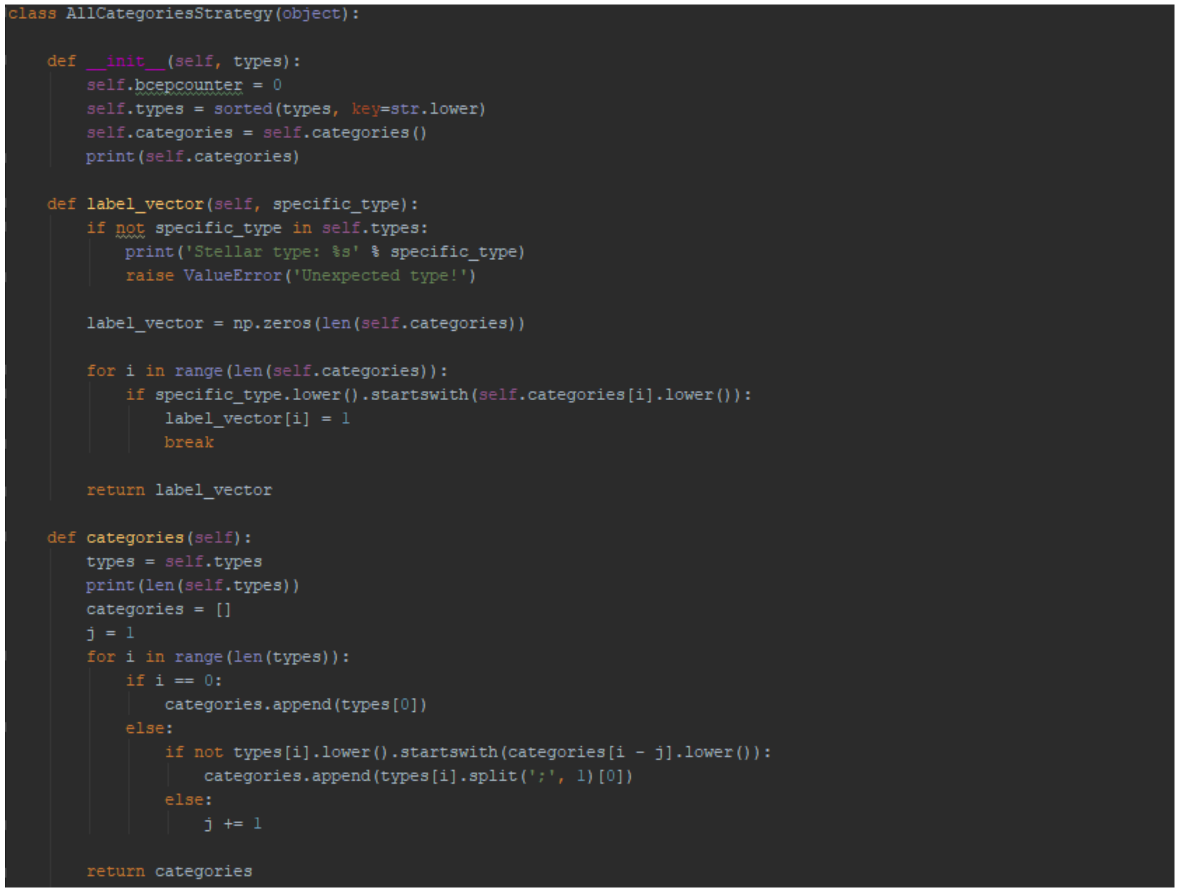


```
            if X.shape[0] <= input_length:

                X1 = np.concatenate((X[:, 1], np.zeros(input_length - X.shape[0])))
                X1 = X1.reshape((1, input_length))
                n = 1

            else:
                if data_splitting and max_size < X.shape[0]:
                    'These lines split the individual datasets'
                    n = X.shape[0] // input_length
                    zeros = np.zeros(((n + 1) * input_length - X.shape[0]))
                    X1 = np.concatenate((X[:, 1], zeros)).reshape((n + 1, input_length))
                elif max_size < X.shape[0]:
                    max_size = X.shape[0]
                continue

            for j in range(n):
                X2 = X1[j, :]
                if np.any(X1):
                    X2[X2 > 0.0] -= np.mean(X2[X2 > 0.0])
                    X2 /= np.std(X2) + 1.e-15
                    X_norm.append(X2)
                    Y_norm.append(Y)
        X_norm = np.array(X_norm)
        Y_norm = np.array(Y_norm)
        with open(str(input_length) + '_' + label_strategy_name + '_' + str(load_extra_data) + '_vectors.pkl',
                  'wb') as fp:
            cPickle.dump(X_norm, fp)
            cPickle.dump(Y_norm, fp)
            cPickle.dump(types, fp)
        return X_norm, Y_norm, types

def shuffle(X_norm, Y_norm):
    ndata = X_norm.shape[0]
    permutation = list(np.random.permutation(ndata))
    X_shuffled = X_norm[permutation, :].reshape(X_norm.shape + (1,))
    Y_shuffled = Y_norm[permutation, :].reshape((X_norm.shape[0], Y_norm.shape[1]))
    ntrain = int(split * 100 * ndata // 100)

    return X_shuffled[0:ntrain, :], Y_shuffled[0:ntrain, :], X_shuffled[ntrain:ndata, :], Y_shuffled[ntrain:ndata,
                                                                                           :]

X_norm, Y_norm, types = load()
X_train, Y_train, X_test, Y_test = shuffle(X_norm, Y_norm)

with open(fname + '.pkl', 'wb') as fp:
    for XY in [X_train, Y_train, X_test, Y_test]:
        cPickle.dump(XY, fp)
    cPickle.dump(types, fp)

return
```

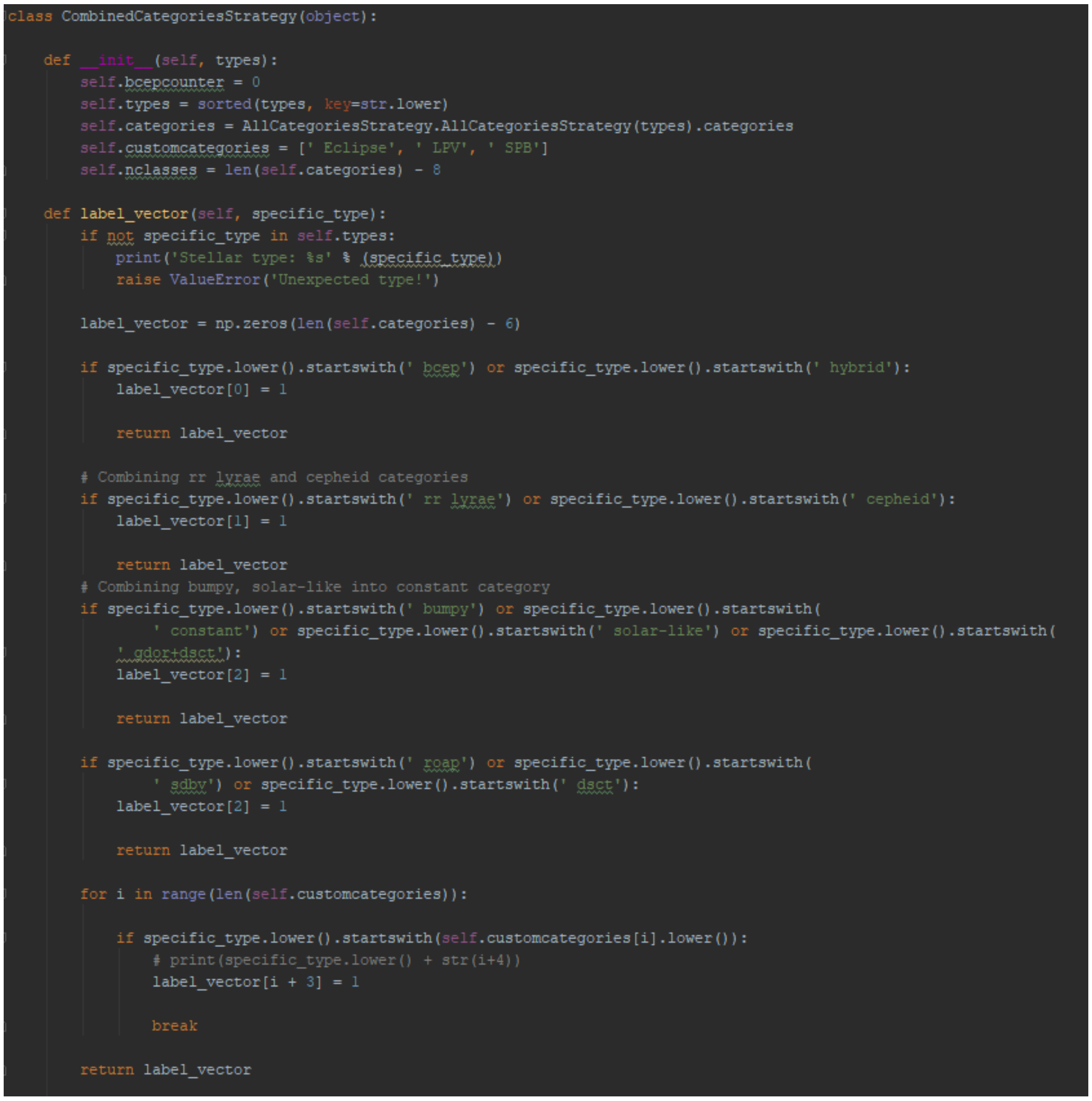
## Polymorphic Labeling Strategy implementation



```python
class BinaryStrategy(object):

    def __init__(self, types):
        self.types = types
        self.i = 0

    def label_vector(self, specific_type):
        if not isinstance(self.types, str):
            print('Input types: ' + str(self.types))
            raise ValueError('Expected single string input')

        label_vector = np.zeros(2)
        if specific_type.lower().startswith(self.types.lower()):
            label_vector[0] = 1
            self.i += 1
            print(self.i)
        else:
            label_vector[1] = 1

        return label_vector
```



```python
class AllCategoriesStrategy(object):

    def __init__(self, types):
        self.bcepcounter = 0
        self.types = sorted(types, key=str.lower)
        self.categories = self.categories()
        print(self.categories)

    def label_vector(self, specific_type):
        if not specific_type in self.types:
            print('Stellar type: %s' % specific_type)
            raise ValueError('Unexpected type!')

        label_vector = np.zeros(len(self.categories))

        for i in range(len(self.categories)):
            if specific_type.lower().startswith(self.categories[i].lower()):
                label_vector[i] = 1
                break

        return label_vector

    def categories(self):
        types = self.types
        print(len(self.types))
        categories = []
        j = 1
        for i in range(len(types)):
            if i == 0:
                categories.append(types[0])
            else:
                if not types[i].lower().startswith(categories[i - j].lower()):
                    categories.append(types[i].split(';', 1)[0])
                else:
                    j += 1

        return categories
```



```
class CombinedCategoriesStrategy(object):

    def __init__(self, types):
        self.bcepcounter = 0
        self.types = sorted(types, key=str.lower)
        self.categories = AllCategoriesStrategy.AllCategoriesStrategy(types).categories
        self.customcategories = [' Eclipse', ' LPV', ' SPB']
        self.nclasses = len(self.categories) - 8

    def label_vector(self, specific_type):
        if not specific_type in self.types:
            print('Stellar type: %s' % (specific_type))
            raise ValueError('Unexpected type!')

        label_vector = np.zeros(len(self.categories) - 6)

        if specific_type.lower().startswith(' bcep') or specific_type.lower().startswith(' hybrid'):
            label_vector[0] = 1

            return label_vector

        # Combining rr lyrae and cepheid categories
        if specific_type.lower().startswith(' rr lyrae') or specific_type.lower().startswith(' cepheid'):
            label_vector[1] = 1

            return label_vector
        # Combining bumpy, solar-like into constant category
        if specific_type.lower().startswith(' bumpy') or specific_type.lower().startswith(
                ' constant') or specific_type.lower().startswith(' solar-like') or specific_type.lower().startswith(
            ' gdor+dsct'):
            label_vector[2] = 1

            return label_vector

        if specific_type.lower().startswith(' roap') or specific_type.lower().startswith(
                ' sdbv') or specific_type.lower().startswith(' dsct'):
            label_vector[2] = 1

            return label_vector

        for i in range(len(self.customcategories)):

            if specific_type.lower().startswith(self.customcategories[i].lower()):
                # print(specific_type.lower() + str(i+4))
                label_vector[i + 3] = 1

                break

        return label_vector
```