INF283 - Compulsory Assignment 1

Mathias S. Grønstad, Oct. 2017

Task 1. Classifying digits

First I read the csv file using the *pandas* library and the *read_csv()* function. I set the the **input layer** to be the first 62 columns. We then have 62 independent variables as a numpy array, *X*. Similarly, the last column is set as the dependent variable, *Y*.
For the **output layer** I use 1-of-N encoding (p. 92) creating binary output values. I create a target matrix, called *target* with equal number of rows as in the dataset, and 10 columns. Each column represents a digit from 0 to 9 and its rows are either 0 or 1 depending on which digit its corresponding row in the dataset . E.g. if a row in the dataset represents 5, then column 5 in *target* is set to 1, and all other columns are 0.

I use the **softmax activation function**, which is commonly used for classification problems where 1-of-N encoding is used (p. 81). This is because unlike logistic, it normalizes the output values to a 0 to 1 interval. Using logistic would gives very small number with many decimal digits. These are rounded up making it seem like the the error stops decreasing. This is because when the small numbers are rounded up the differences are lost. I use the sum of squares **error function**, as suggested in mlp.py. I could also consider using the cross-entropy cost function.

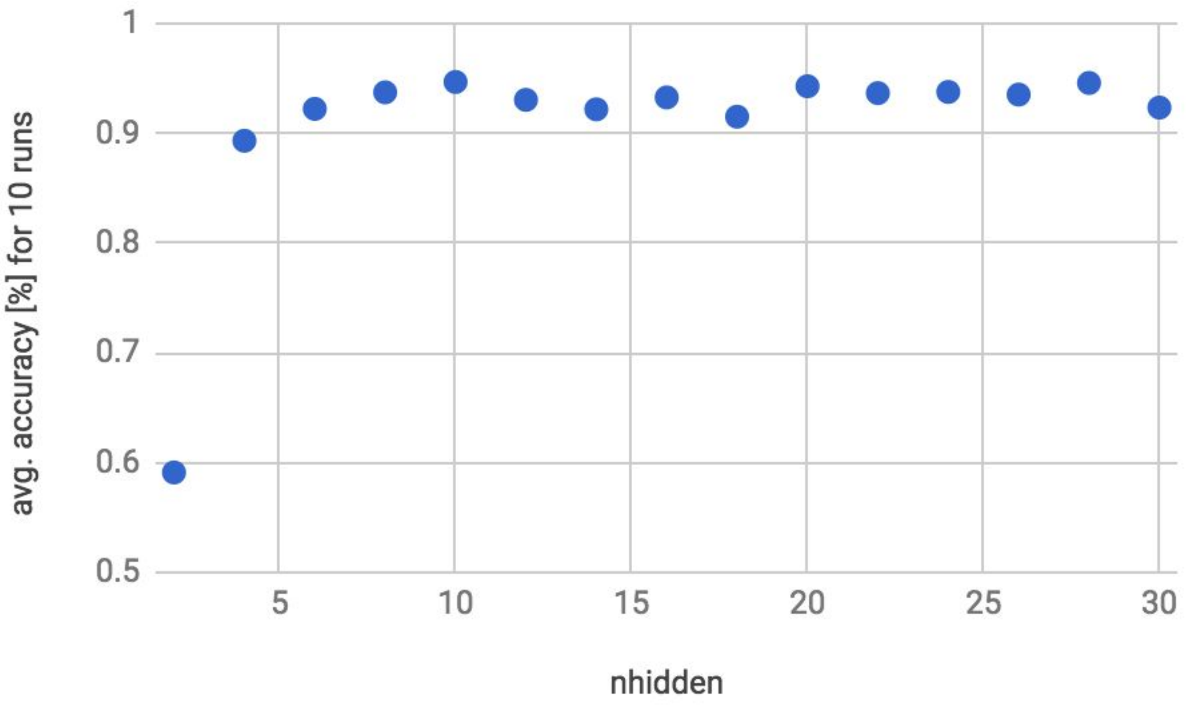
Momentum, α is set to α = 0.9, since it's what's commonly used and suggested (p. 84). This parameter adds some contribution of the previous weights to the current ones. I chose eta, η = 0.1 since we want a relatively low learning rate, or else we can end up in local minima. I investigate this parameter later.
I use 100 iterations since it was used in the iris.py program. Lastly, beta is set to β = 1. Beta effectively makes the learning rate faster since it updates the weights faster:

```
deltah = self.hidden**self.beta*
*(1.0-self.hidden)*(np.dot(deltao,np.transpose(self.weights2)))

updatew1 = eta*(np.dot(np.transpose(inputs),*deltah*[:,:-1])) +
self.momentum*updatew1
```

We don’t want it too high because it’s purpose is to shape the transfer function, not speed up the learning rate (we have η for that). Generally, too large weight updates can cause us to converge to a local minimum, instead of a global minimum. The former would yield a poorer accuracy. I don’t adjust beta (leave it at 1 for now) since this is a simple classification problem with a threshold of 0.5, so that values will be rounded to 0 or 1.

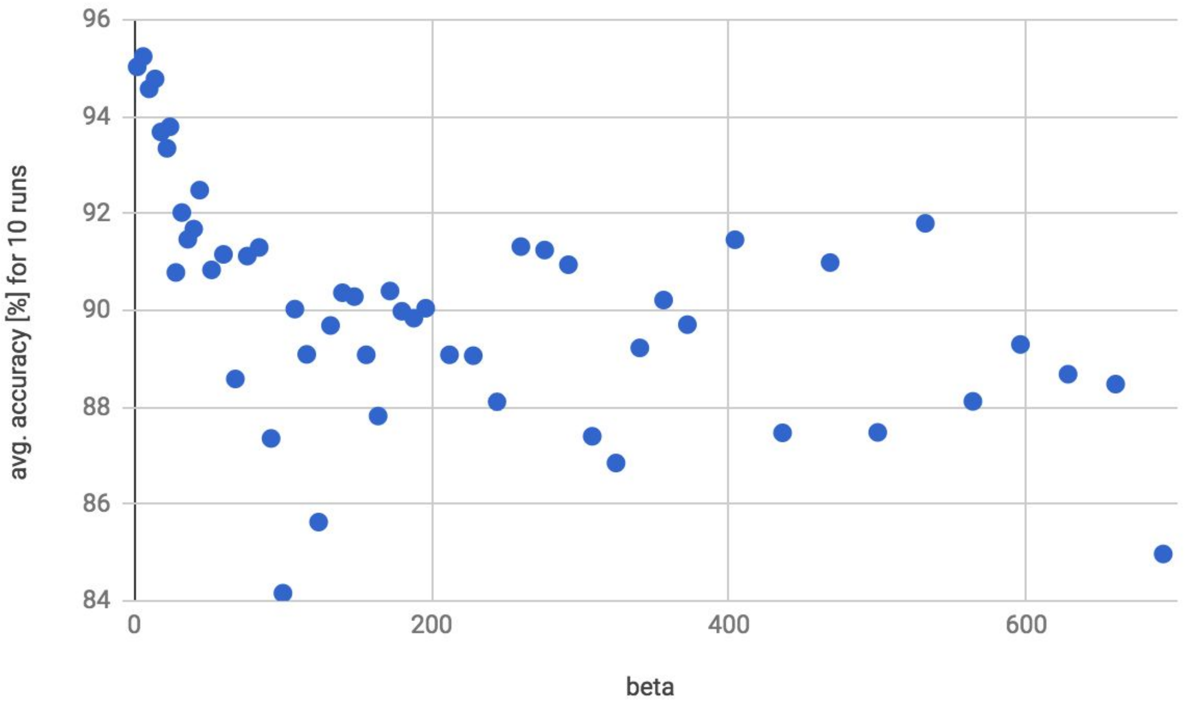
To test how many hidden nodes I should include in the **hidden layer** (HL) I do many several (10) runs for each chosen *nhidden* and average the result. I must do this due to variance as a result of pseudo-random number generation. The variance is quite significant and not taking averages would cause bad estimates since the accuracy can vary +/- 1% or more. Using the above mentioned parameters I run the program for a range of choices of *nhidden* as seen in the chart below:



It seems like *nhidden* = 10 could be a good choice, giving about 94.7% accuracy. The accuracy increase seems to flatten out after this.

As the plot demonstrates, runtime increases (seemingly) linearly with *nhidden*. We therefore use *nhidden* = 10, since after that the increase in accuracy doesn’t make up for the loss in runtime.

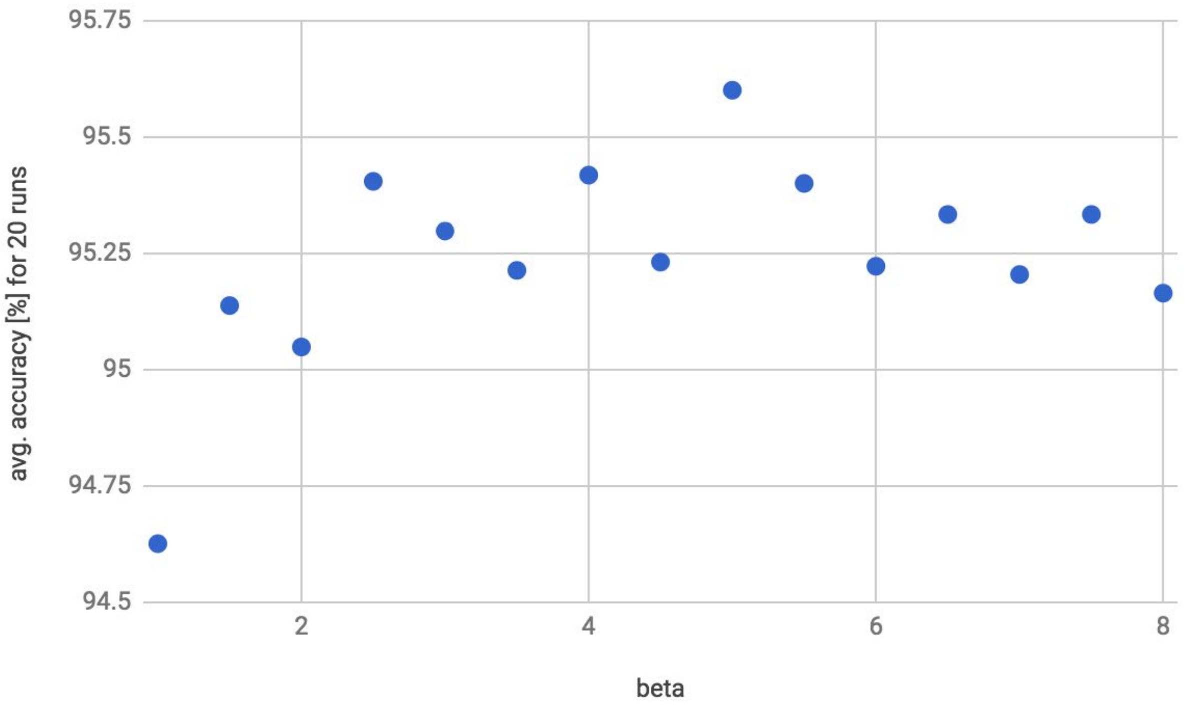
I use 10 nodes to further investigate the parameter beta. The reason for this is that I noticed that larger betas made the program run faster (since it affects learning rate), but it didn’t seem to cause poorer accuracy (unless it’s very high). Out of curiosity I wanted to see how far I could take this. How high beta can I use before it causes trouble? Below is a plot of beta vs accuracy, each taking the average of 10 runs, using *nhidden* = 10:



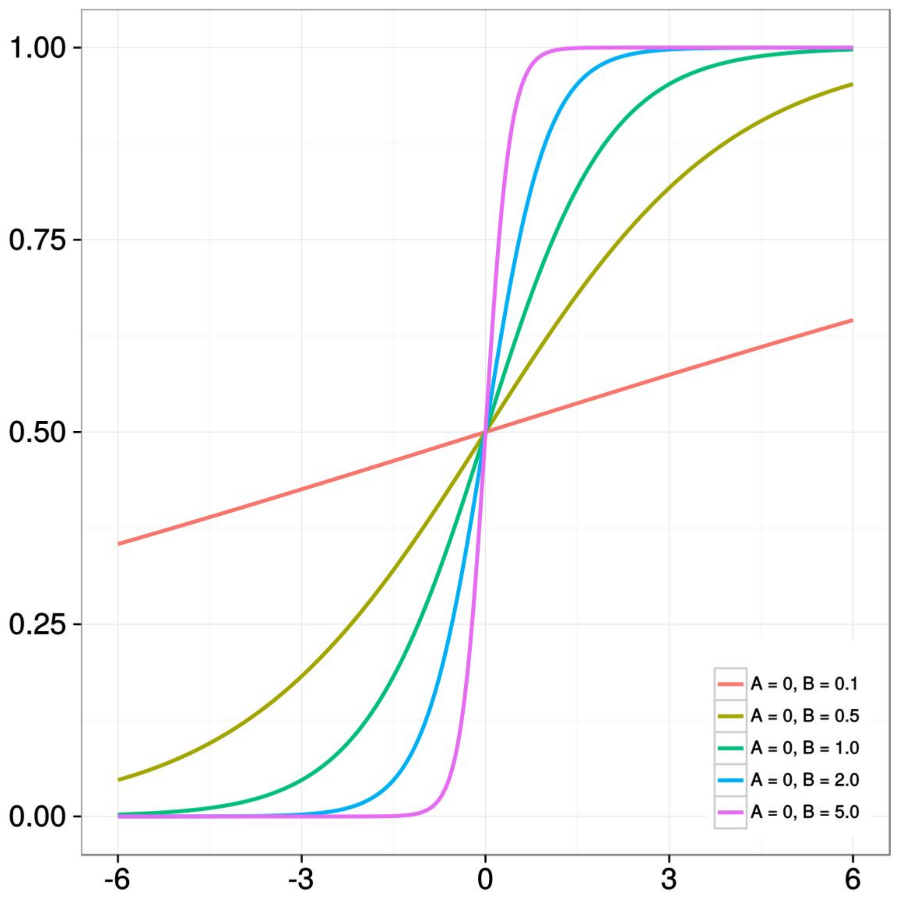
It seems that that using β > 24 starts to yield quite fluctuating, and poor accuracies. Using too high β causes problems since it's involved in an exponent:

RuntimeWarning: overflow encountered in exp
  *self.hidden = 1.0/(1.0+np.exp(-self.beta*self.hidden))*

I assume that this is what causes the big fluctuations, we deal with very small numbers (due to the exponent) in some instances. The higher the β the more often these occur. To find a good value for β I took the average of 20 runs for various β using *nhidden*=10

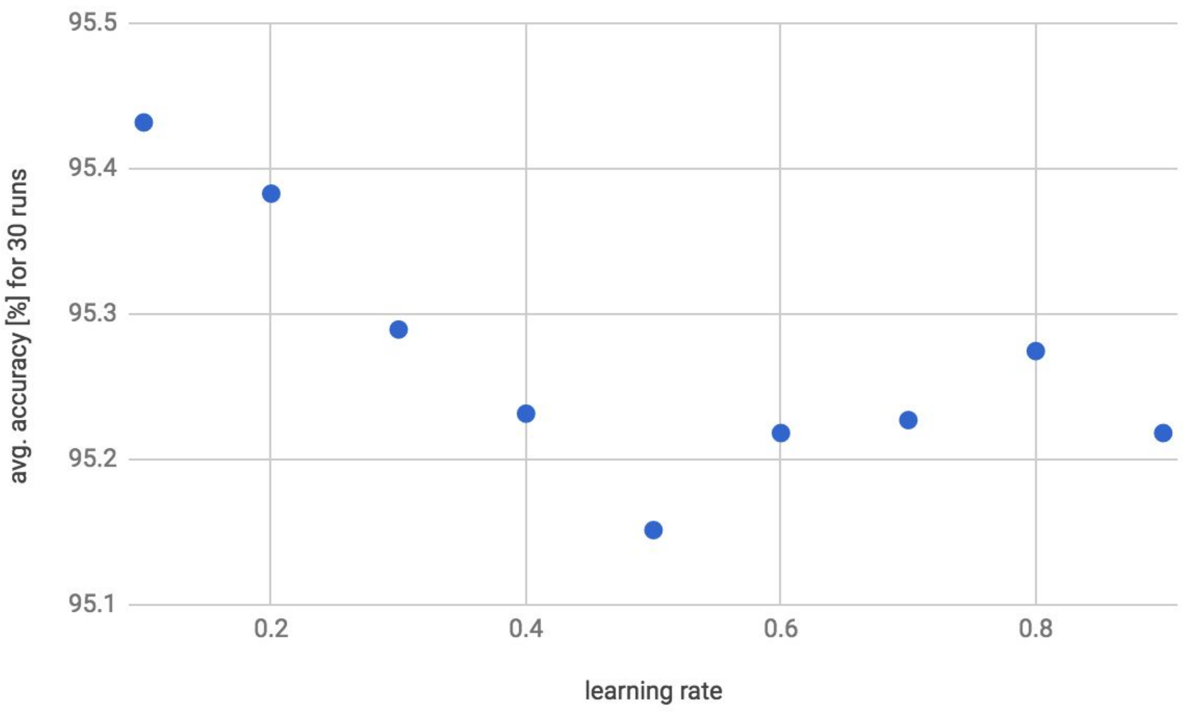


Even taking an average of 20 runs gave quite noticeable variance, but from the plot it seems like the suggested β=1 is suboptimal. For the rest of this assignment, a β=5 will be used. In general this is not recommended value, but for this particular dataset it works fine. In general it should be less than 3, preferably around 1. Beta influences the sigmoid function so that a higher β gives a steeper curve (more similar to the threshold function used by the perceptron).



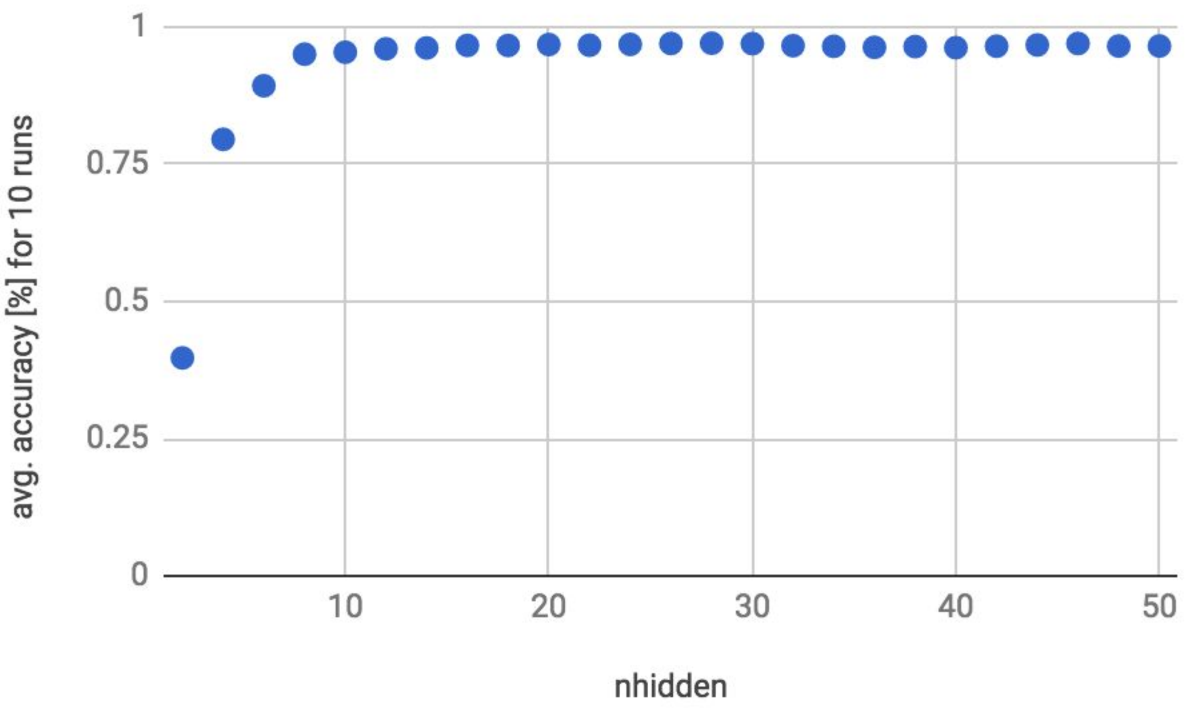
Source: https://en.wikipedia.org/wiki/File:GeneralizedLogisticB.svg

I also wanted to investigate the effect of η. I took the average of 30 runs for different learning rates (*nhidden* = 10, β = 5)

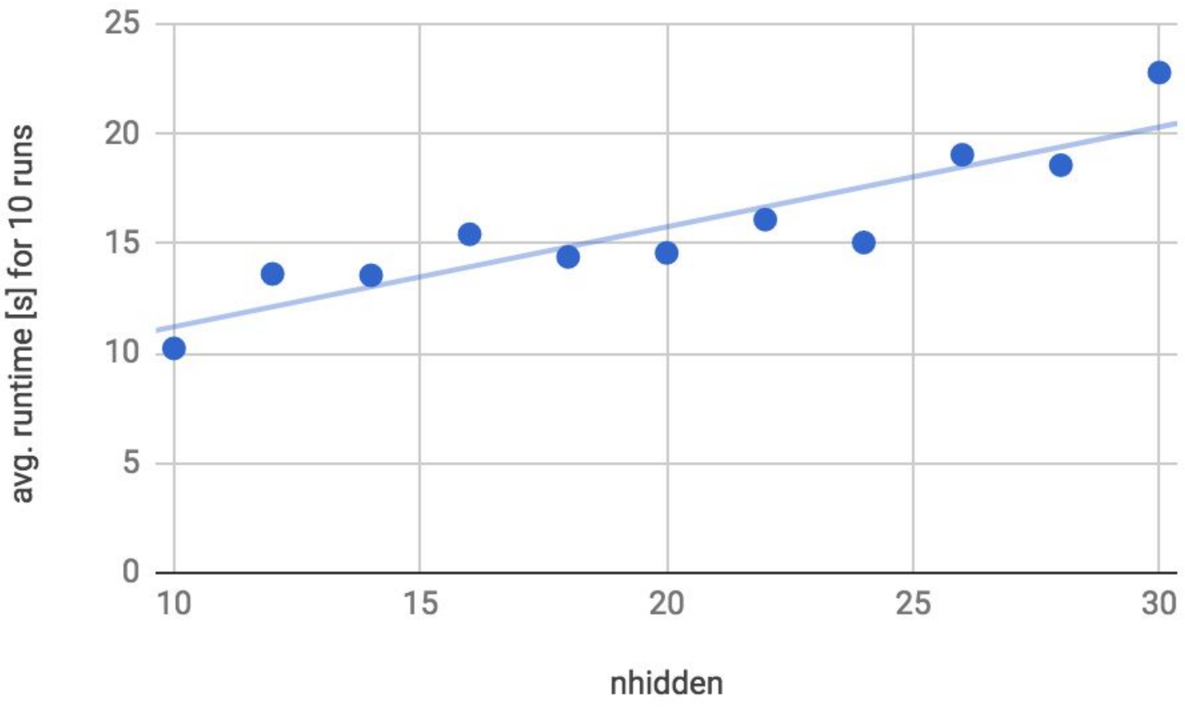


It seems like we should use η = 0.1. Why the accuracy suddenly increases after η = 0.5 is unexplainable for me, and I can only guess that it's due to randomness. But taking the average of 30 runs makes it seem odd. Maybe 0.8 is the best learning rate for the particular dataset (if you require high speed).

I want to test again if the optimal choice of *nhidden* using β = 5.



The highest accuracy I get is 97.04% for 28 nodes, but 20 nodes gives 96.80% and everything above 20 gives almost 97%. Let's compare runtime differences vs number of nodes:



As the plot demonstrates, runtime increases (seemingly) linearly with *nhidden*. We therefore use *nhidden* = 10, since after that the increase in accuracy doesn't make up for the loss in runtime.

Task 2 - A) Multiple hidden layers

In the MLP, learning happens in the weights, so adding a second HL meant modifying the weights in mlp.py. I made a new file called mlp2.py. First I must add a new set of weights:

*self.weights3 = (np.random.rand(self.nhidden+1,self.nout)-0.5)*2/np.sqrt(self.nhidden)*

This set of weights are from the second HL to the output layer. I also modify the *weights2* so that they are between the first HL to the second HL. The *weights1* are still from the input layer to the HL layer. I have to make changes in all places where the weights are involved. Next I created the second HL:

self.hidden2 = np.dot(self.hidden,self.weights2);
*self.hidden2 = 1.0/(1.0+np.exp(-self.beta*self.hidden2))*

I removed the following in both layers since it caused problems with dimensions, and I found it to be redundant:

self.hidden2 = np.concatenate((self.hidden2,-np.ones((np.shape(self.hidden)[0],1))),axis=1)

For the back propagation and weight, update I need to create an additional delta function:

*deltah2 = self.hidden*self.beta*(1.0-self.hidden)*(np.dot(deltah,np.transpose(self.weights2)))*

which goes from HL2 to HL1. I modify *deltah* so that it goes from output to HL2. The weights will be updated using:

updatew1 = eta(np.dot(np.transpose(inputs),deltah2)) + self.momentum*updatew1*
updatew2 = eta(np.dot(np.transpose(self.hidden),deltah)) + self.momentum*updatew2*
updatew3 = eta(np.dot(np.transpose(self.hidden2),deltao)) + self.momentum*updatew3*

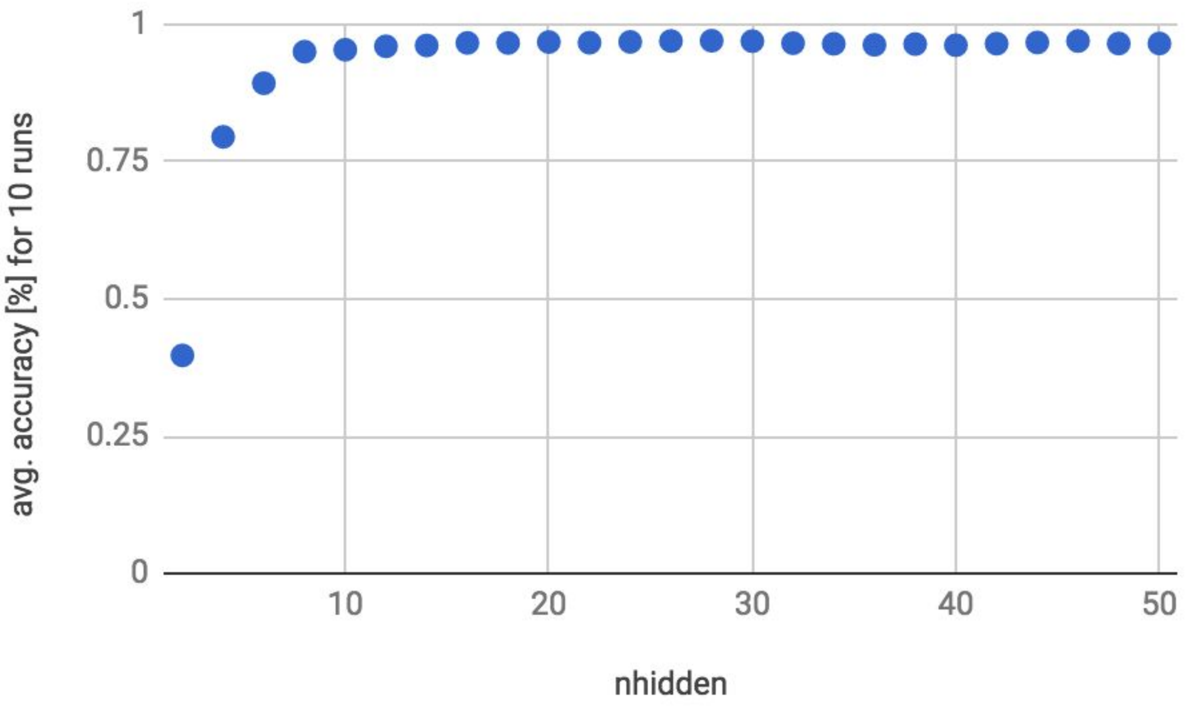
In the *mlpfwd* function i must make modifications such that HL1 "forwards" to HL2 and HL2 "forwards" to output:

self.hidden = np.dot(inputs,self.weights1);
*self.hidden = 1.0/(1.0+np.exp(-self.beta*self.hidden))*
self.hidden2 = np.dot(self.hidden,self.weights2);
*self.hidden2 = 1.0/(1.0+np.exp(-self.beta*self.hidden2))*

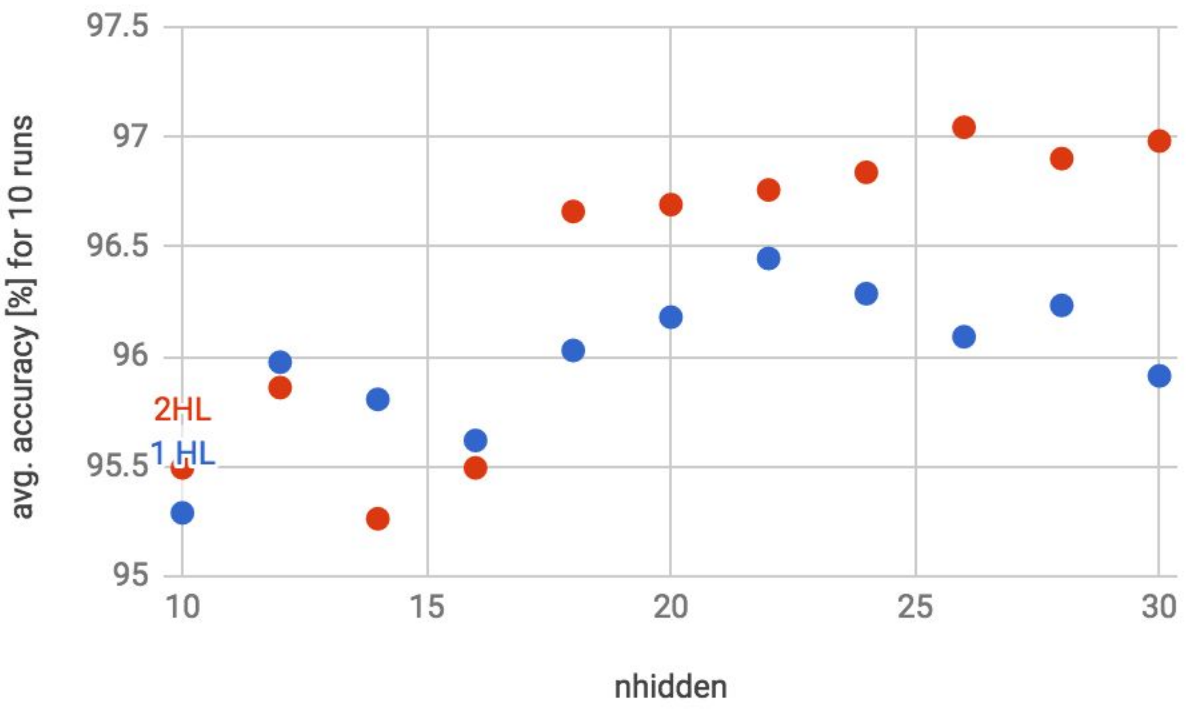
outputs = np.dot(self.hidden2,self.weights3);

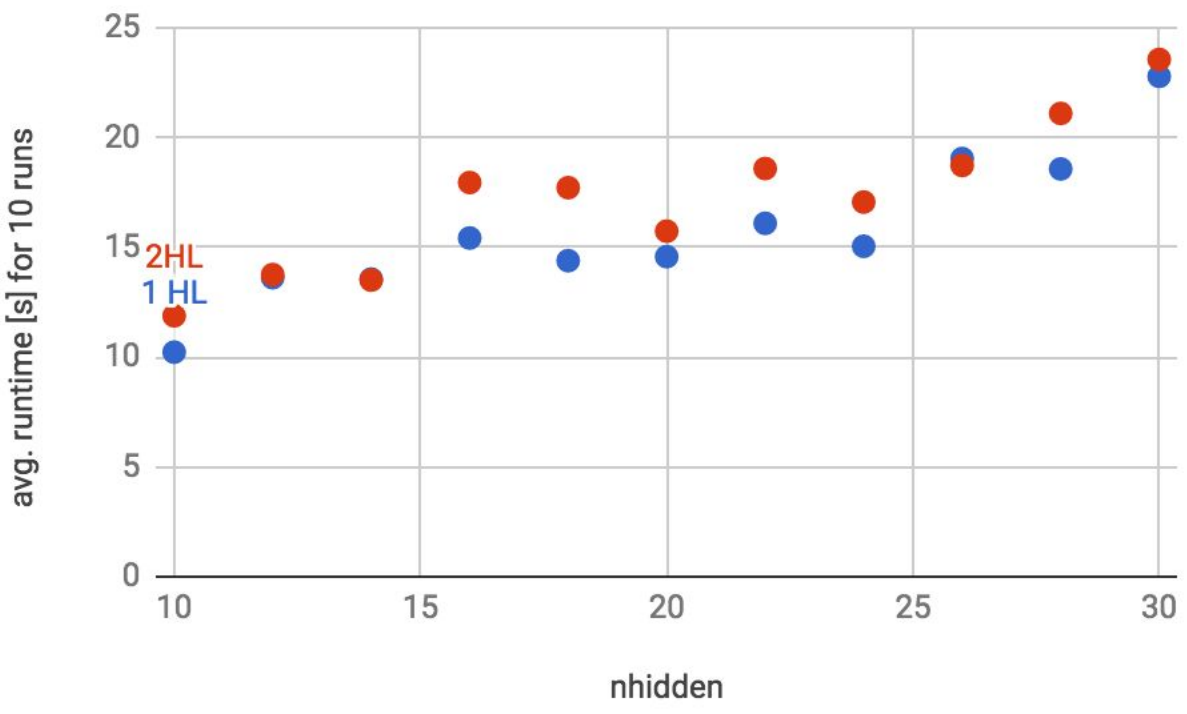
Adding a third HL would simply require the same process as above.

To find an optimal number for *nhidden* using 2 HLs I use the paraments from the previous task:
η = 0.1
β = 5 niterations = 100



Again, *nhidden* = 10 seems like a good choice which gives about 95.4% accuracy, while for 1 HL we got 95.3%. This difference insignificant, and could be mostly due to randomness. Nevertheless, there is very small increase in accuracy, so lets to an overall comparison of accuracy and runtime when using 1 HL (blue) vs. 2 HL (red).


97.5
97
96.5
96
95.5
95
10
15
20
25
30
avg. accuracy [%] for 10 runs
nhidden
2HL
1 HL


25
20
15
10
5
0
10
15
20
25
30
avg. runtime [s] for 10 runs
nhidden
2HL
1 HL

The runtime is pretty much identical whether we use 1 or 2 HLs, so is accuracy, especially for small *nhidden*.
When using *nhidden* > 15, using 2 HLs does seem to give slightly better accuracy than using 1 HL. If you can sacrifice runtime for accuracy, then multiple hidden layers might be better. But, according to the book (p. 86) we normally only need one hidden layer, but it may need an arbitrarily large number of hidden nodes (universal approximation theorem). Using two hidden layers can be advantageous since it approximates using linear combinations of the sigmoid functions in each layers.

Extra:
Using predefined libraries can make adding multiple layers a lot easier. I did so with the Titanic dataset using the Keras library to add a second layer. This is the code for adding two HLs and an output layer.

```
from keras.models import Sequential
from keras.layers import Dense

# Initialising the ANN
# Defining it as a sequence of layers
classifier = Sequential()

# Adding the input layer and the first hidden layer
# using the rectifier activation function for hidden layers.
classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu', input_dim = 11))

# Adding the second hidden layer
classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))

# Adding the output layer with a sigmoid activation function
classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))
```