where η is the *step size* parameter controlling how long a step $\triangle$w should be. This method is called *gradient descent* and is illustrated in Figure 2. Differentiating (5) with respect to the weights in the output and hidden layer gives

$$\triangle w_{kj}^{(2)} = -\eta \left(\sum_{n=1}^{N} (y_{nk} - t_{nk}) z_{nj} + \alpha w_{kj}^{(2)} \right) \tag{8}$$

$$\triangle w_{ji}^{(1)} = -\eta \left(\sum_{n=1}^{N} \left((1 - z_{nj}^2) \sum_{k=1}^{c} w_{kj}^{(2)} (y_{nk} - t_{nk}) \right) x_{ni} + \alpha w_{ji}^{(1)} \right) \tag{9}$$

using that the nonlinear function $g(a) = \tanh(a)$ shown in Figure 3.

In the following sections we want to investigate some of the parameters involved in the optimization.

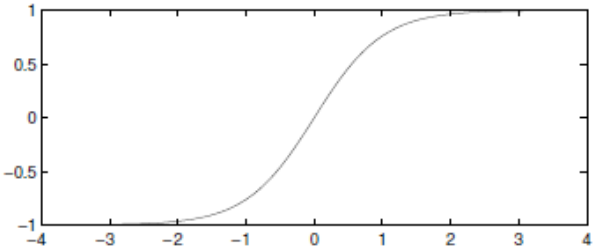


Figure 3: The nonlinear function $g(\cdot) = \tanh(\cdot)$.

Initialization range

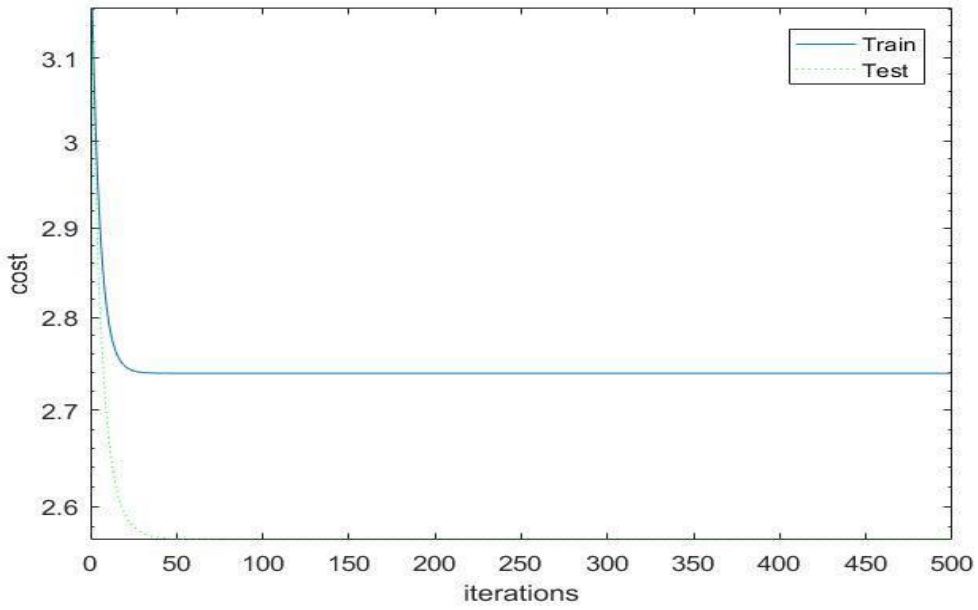
The gradient descent process starts from an initial set of values for the weights. The role of the initialization range is important for the convergence of the parameters. If the range is making the initial weights too small compared to the signal, the gradient will be small and learning slow initially. On the other hand, setting the range to high forces the nonlinear function close to the step function and again the gradients become small and learning slow.

Checkpoint 5.1:

Use the program main5a.m to create an NN training-set with a 4-dimensional input variable, 5-dimensional hidden variable and a 1-dimensional output variable ($4 \times 5 \times 1$). Observe the effects of setting the range of the initial weights to: 0, 0.5 and 10000. Use Eqs. (8,9) and Figure 3 to explain the effects you see.

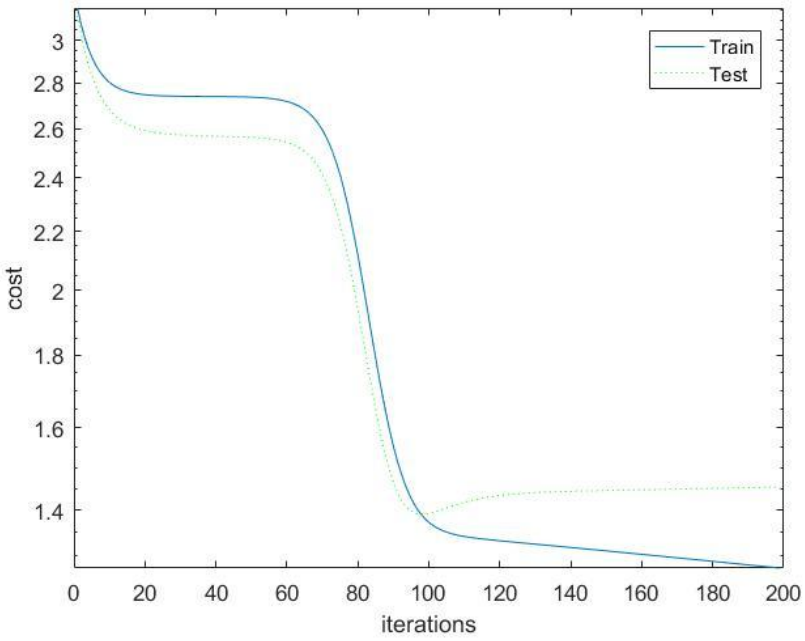
When we don't initialize the weights appropiately, the training process of our neural network is compromised.It is important to note that we are not applying weight decay in this exercise, i.e. alpha=0.

Let's start by the case when the initial weights are equal to 0, where the graph looks as follows.

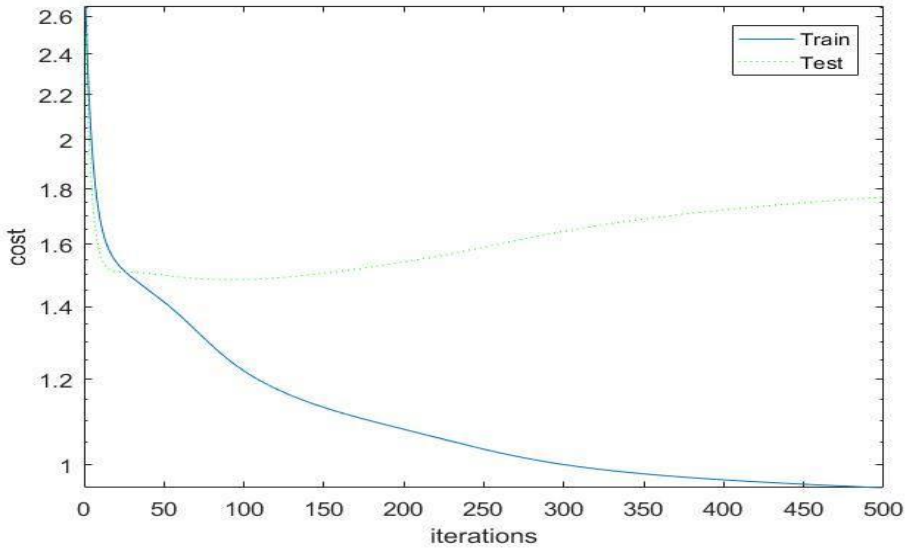


First of all, it is important to realize that the case when the initial weights are equal to 0 is an extreme case. We can note that when we input the 0 weights in the equations, equation 8 will only be non-zero for the weight of the bias between the hidden and output layer. The rest of the weights will always stay zero, which reduces the optimization problem to just one dimension, were we optimize that weight of the bias. And this is what we can see from the graph, the weight is being modified to go towards the mean of the output. As all the neurons from the hidden layer are "disconnected" and the value of the bias will always be 1, the input from the data won't affect at all the value of the weight. This also explains why the test cost is lower than the train cost. Based on this model, it just means that the training data is more spread out than the test data, and as we are only learning the mean value this cost will be lower.

In order to understand better this phenomenon, let's set the weights to 0.0001. Which gives us the following graph:

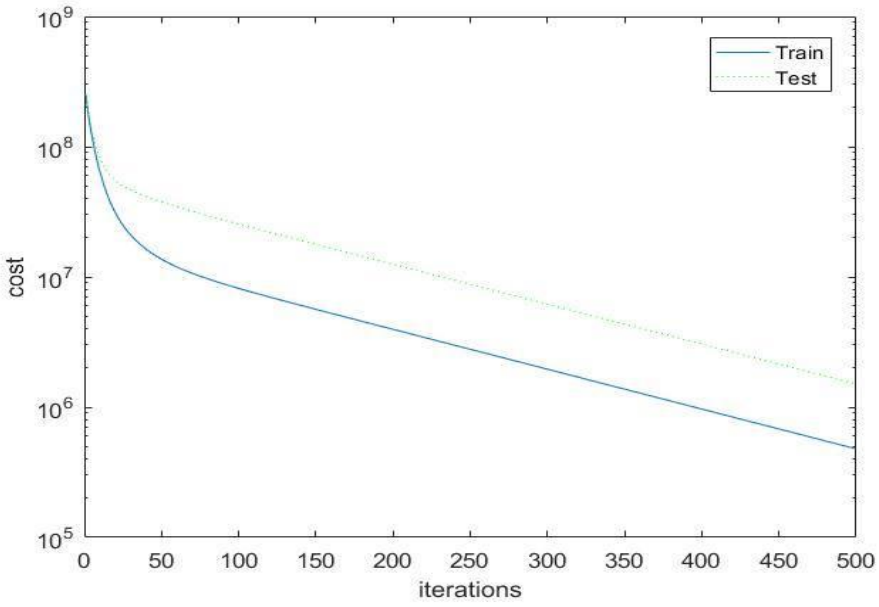


As we can see, at the beginning the graph looks really similar to the one before, but afterwards the weights are big enough to leave that local minima and continue learning from other dimensions. But, why does the test cost grows? Let's look at the case when we initialize the weights in the range of 0.5 to understand this better.



As we can see, this weight initialization is more appropiate for this model, and we quickly learn the training data. The problem is that after around 100 iterations the model start to overfit to the training data, causing the test cost to grow.

Finally, we will take a look at the situation when we initialize our weights with a very large value, as presented in the following graph:



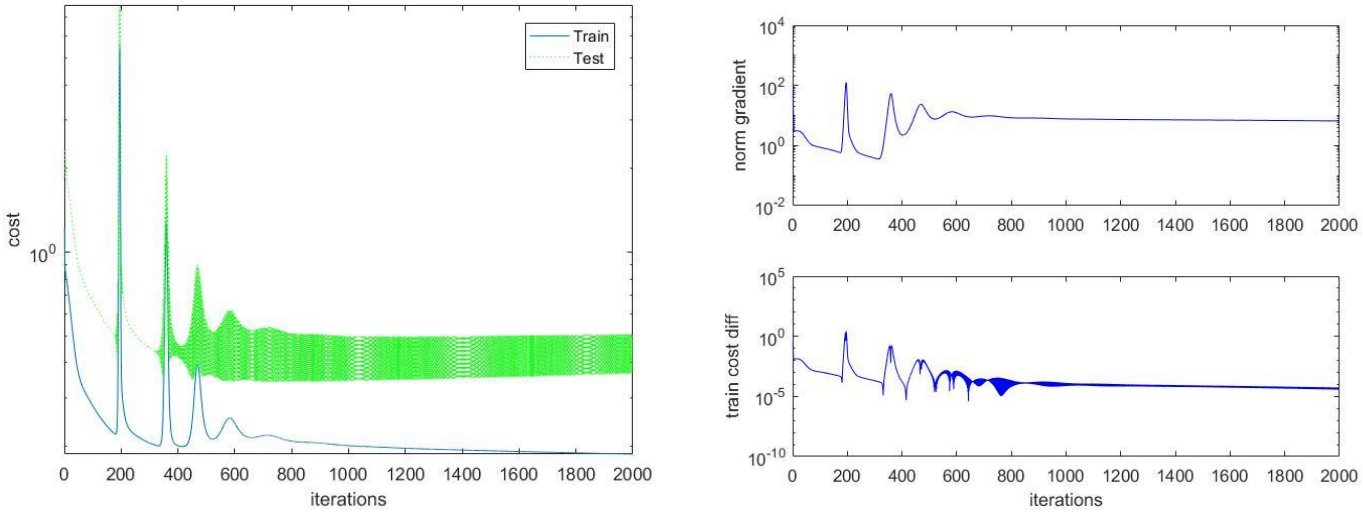
If we look at the cost we quickly see the difference in the scale, and how, even though it tries to learn, it does so really slowly. In order to understand this let's take a look at the equations 8 and 9. If we take a look at equation 8 the z term will be really close to 1, as due to the shape of tanh function and the large

weights all the neurons will stay in saturation. Meanwhile, the difference between target and signal is expected to be large, so the first gradient will be large. However, if we take a look at eq.9 que see that as z will be really close to 1, the result will be really close to 0, making the non-linear part of the model to learn really slowly, while maintaning the high cost. If er turn on alpha solve the problem after a lot of iterations.

Checkpoint 5.2:

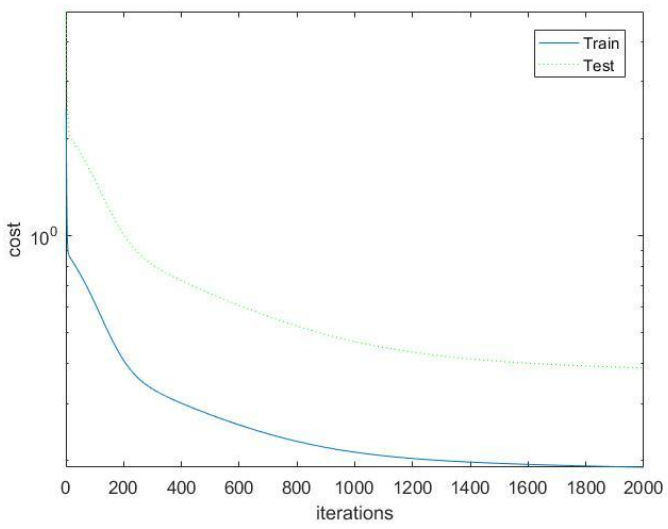
Use the program main5a.m to create an NN training-set with a 4-dimensional input variable, 5-dimensional hidden variable and a 1-dimensional output variable ($4 \times 5 \times 1$). Set the step size to different values and observe the behavior. What is a good step size for the data-set?

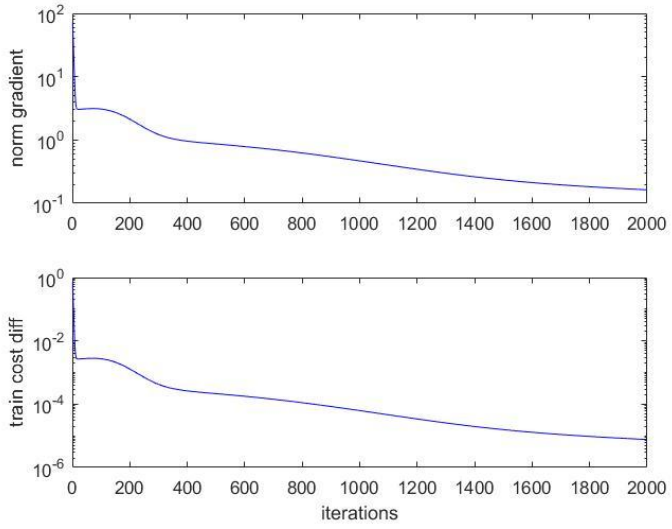
In order to see the behaviour of a NN for different learning step sizes we are going to use main5b.m, as it also shows the difference of costs and the norm of gradients. The first graph shows this variables for a learning step of 0.005



In this case, it seems that we reach a global minima, but the learning process produces a few overshoots. This is caused because the step that we take is too big, which can end up in this kind of behaviour in the cost function. Furthermore, when we reach around 800 iterations, even though there are no big changes in the training cost (see train cost difference graph) the test cost suffers from an oscillatory behaviour. This can be explained by the norm gradient graph, where we can note that even though our test gradient is being reduced steadily the norm gradient stays quite big. Thus, there are always gradients being backpropagated, but with inverted signs in each iteration. This changes does not have a great effect on the training cost, but, as we are not optimizing for the test, we can note those jumps over a minimum of the cost function.

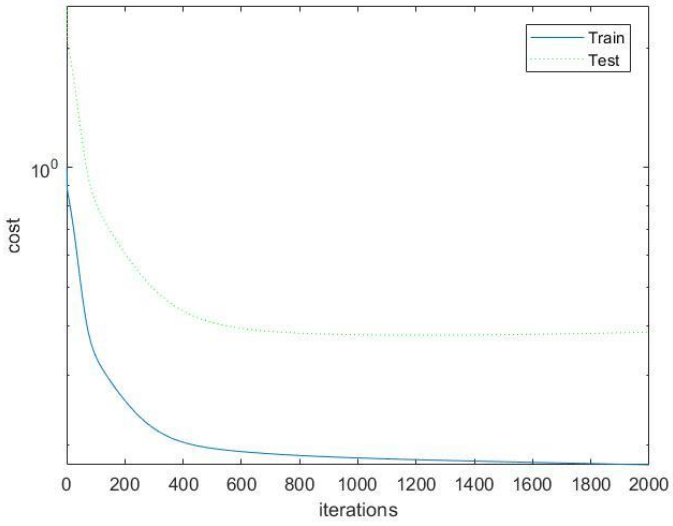
The previous learning rate of 0.005 is definitely too big, so we will try with a smaller one, 0.001.

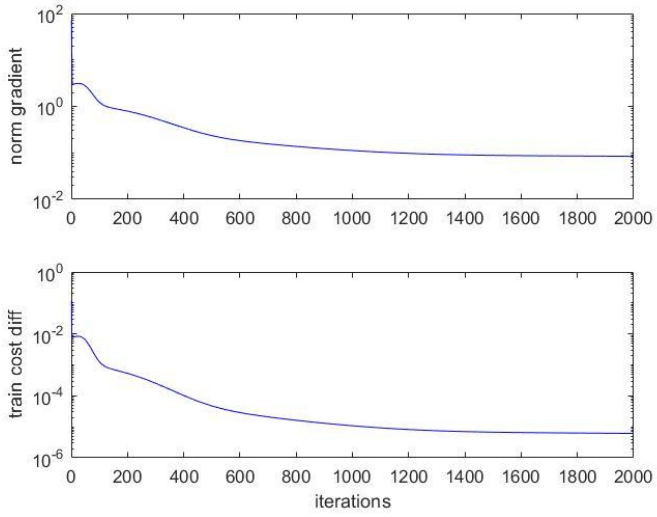




With this learning rate we can note a more smooth cost function, without big jumps between iterations, as we can also observe from the train cost diff graph. It is important to see also the difference between this norm gradient graph and the previous one, where the order of magnitude was two times higher.

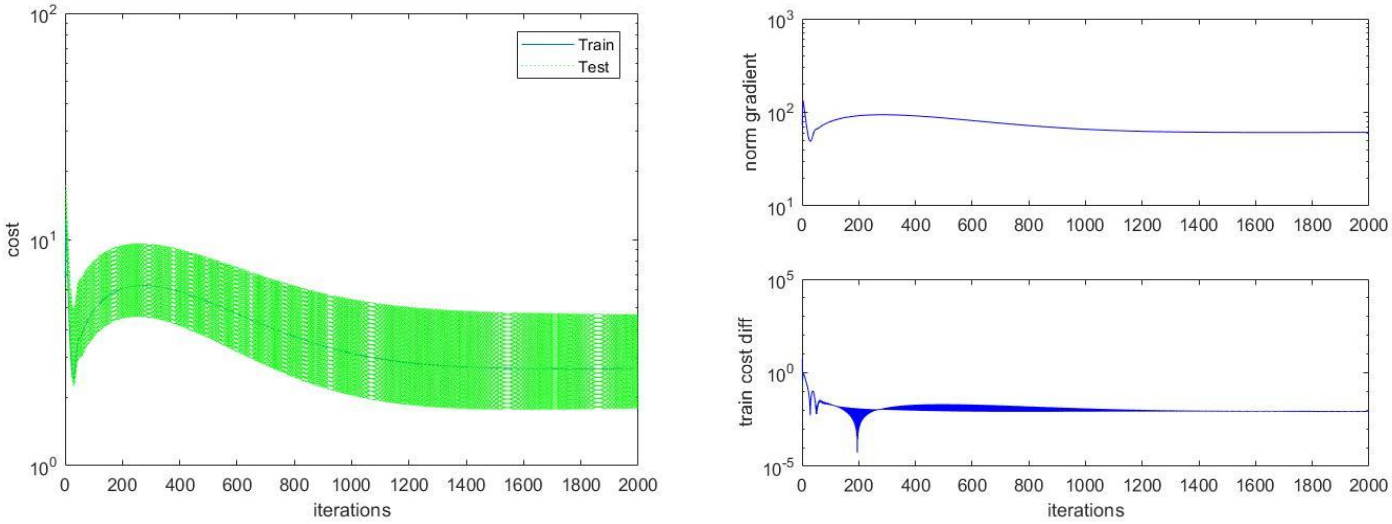
However, the learning seem rather slow, which is not an issue for a network of this size, but it matter in a more complex model.For this reason we experimented also with eta = 0.003.





We can see a small improvement in terms of speed of the learning as well as the final cost after the 2000 iterations, due to the faster learning rate.

Finally, for the seek of experimentation we tried with an even bigger learning rate than from the previous example, in order to see the effect of it in our training. Thus, with a learning rate of 0.009 we obtain the following results:



In this case we can see, we also have the oscillatory behaviour presented in the previous experiment. However, now this jumps over the minimum also have an effect on the training cost, which becomes significantly larger than in the previous experiments. In this case, as the learning rate is so big, not even the training cost reaches that minima, and keep backpropagating a series of positive and negative gradients.

Given this results, we consider that the best result is the one obtained by the learning rate of 0.003

Stopping criteria

Various methods can be used to stop the iteration process. The stopping criterion depends on the application. Here we will consider two possible candidates.

Checkpoint 5.3:

Use the program main5b.m to plot the test and training errors, the length of the gradient vector, and the training error difference between iterations. Comment on the plots and how they could be used as a stopping-criteria.

The first idea that one come up with is usually to use the training/test cost graph in order to determine the end of the training. However, this approach is not optimal. If we take a look at the graphs when the learning rate is big, we encounter two problems with thresholding this value. In the case of eta = 0.005 we can see that we may set a threshold and then, if we would have continue training the cost would have suffer a lot of changes. This can be solved by setting a number of consecutive iteretions for which the training cost should be below the threshold. But even then, we are not taking into count the oscillations of the test error.

However, this approach will led to the problem of which threshold cost should you select. For example, in the case with a really big eta the final cost after 2000 iterations is way higher than in the other cases, so if we set this threshold based on the other examples the training will never finish in this case.

Thus, we can argue that taking a look at just the plain cost function is not the best option, so let's try with the norm of gradients graph. In this graph we can note what the network's gradient size is, which can be a useful stopping criteria. Nevertheless, this criteria alone can present some issues, mainly because

one can not conclude that the learning stopped just because the norm gradient is small, as the training cost can be still decreasing substancially.

Now let's take a look at the cost difference graph. This variable provides a good insight about weather if the training cost is actually being reduced over time or not. However, it is not a good criteria on its own, because as it was explained previously, it ignores completely the fault case when the gradients are still big.

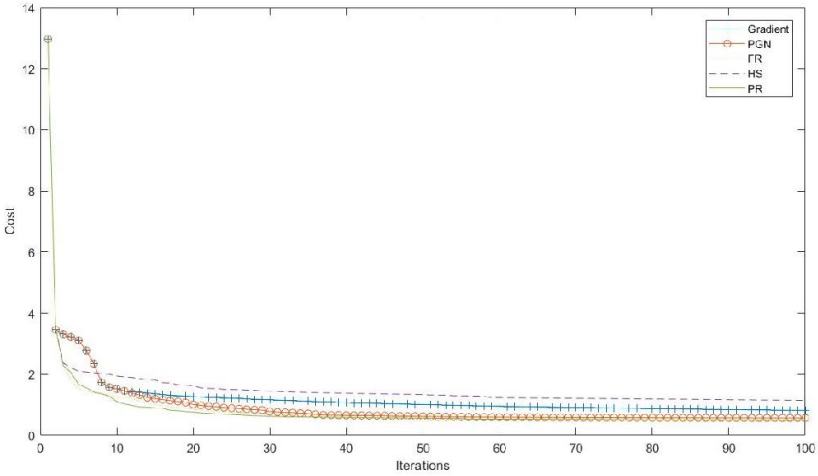
Based on all this thoughts, we can conclude that the best stopping criteria would be to threshold both the norm gradients and the cost difference, ensuring this way that the network has been trained correctly.

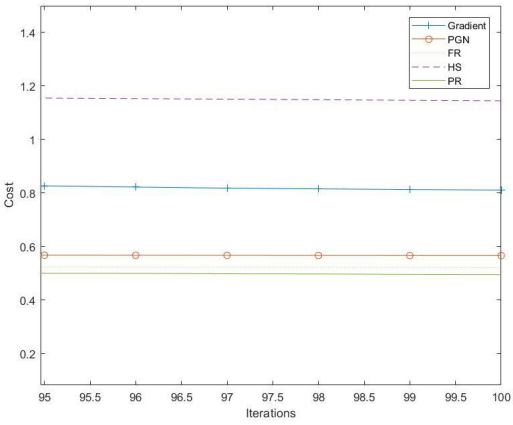
Checkpoint 5.4

Use the program `main5c.m` to investigate the speed of convergence for the different algorithms. The program runs the algorithms a couple of times (each with a different seed for the initialization of the weights) and computes the average of the cost function value. The evolution of the cost functions is available in the variable `E`.

Determine which is the best algorithm and comment on the shapes of the curves.

Let's start by taking a look at the output graph of the program



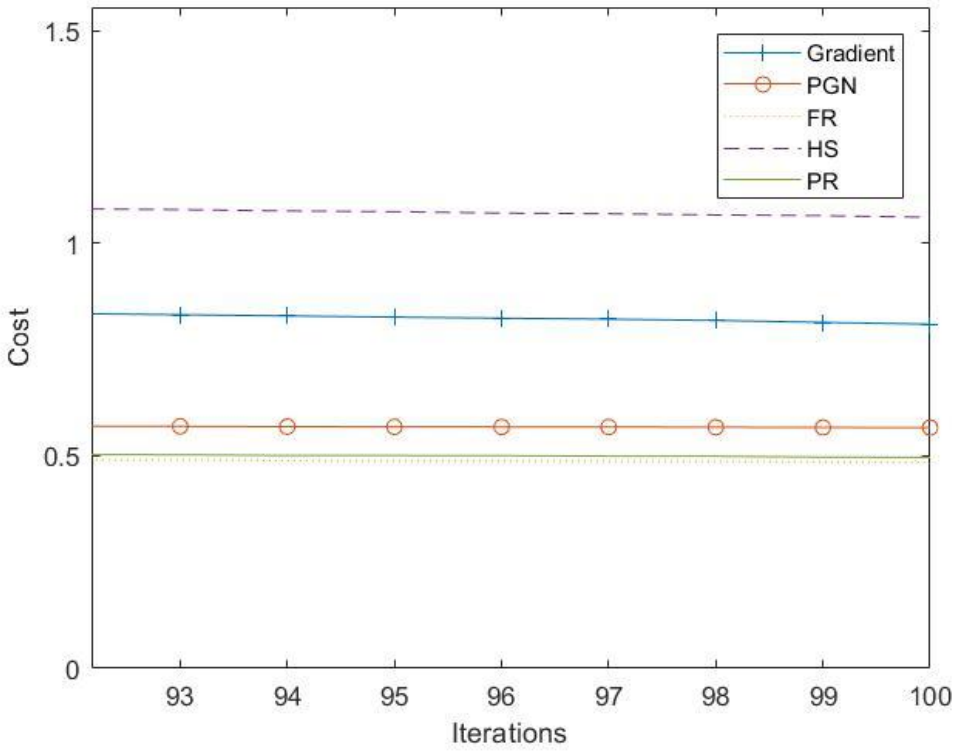


From this graph we can extract two important indicators that defines the performance of an optimization algorithm: Final cost after 100 iterations and speed of convergence.

Based on the first indicator we see that the algorithm that throws better results is Polak-Ribiere, followed closely by Fletcher-Reeves. The worst result in this case is the one obtained by using Hestenes-Stiefel.

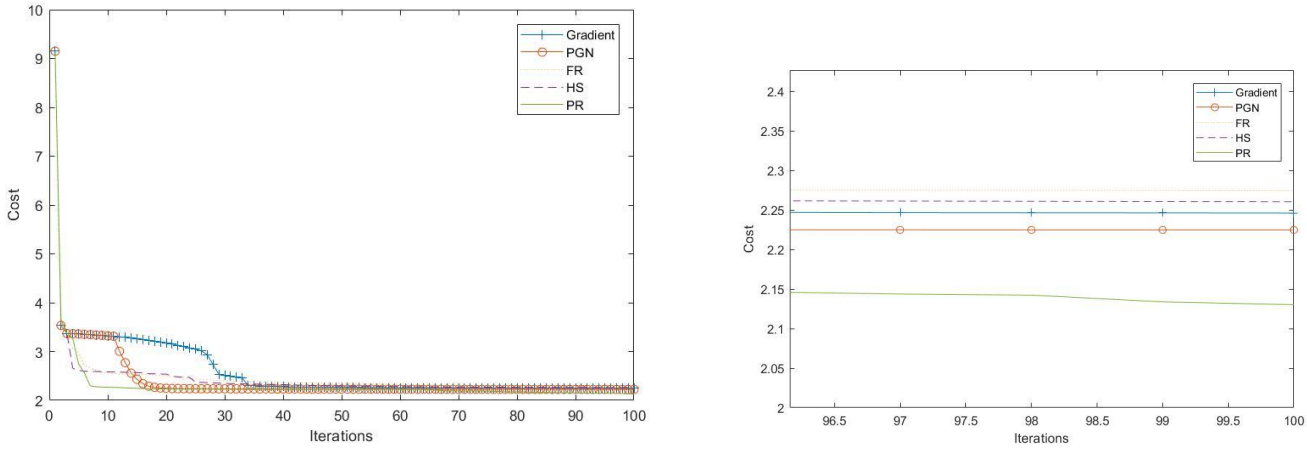
The second indicator shows also a really good performance of Polak-Ribiere algorithm, which led us to the conclusion that it is the best algorithm for this datset and neural network.

However, it is important to highlight that there is a reason why so many algorithms exist and are used, and it is because they work better in different scenarios. For example, we tried to change the weight initialization from 0.5 to 0.9, and we obtained the following final cost results:



In this case, and just with that subtle change, the FR algorithm achieves slightly better results than the PR algorithm.

In the case when the network is completely changed, the results are even more different. Let's take now a NN with 5 input neurons and 100 hidden neurons:



As we can see, the performance of gradient descent in this case is way worse than the one in the previous case. At the same time, if we take a look at the final cost we can see that the FR algorithm, which gave us the best results in the previous case, is giving now the higher cost.

Thus, we can conclude that there is no method better than another just based on this particular example. With different datasets we would probably obtain different results, so the only solution is to try different algorithms for the particular problem on hand.