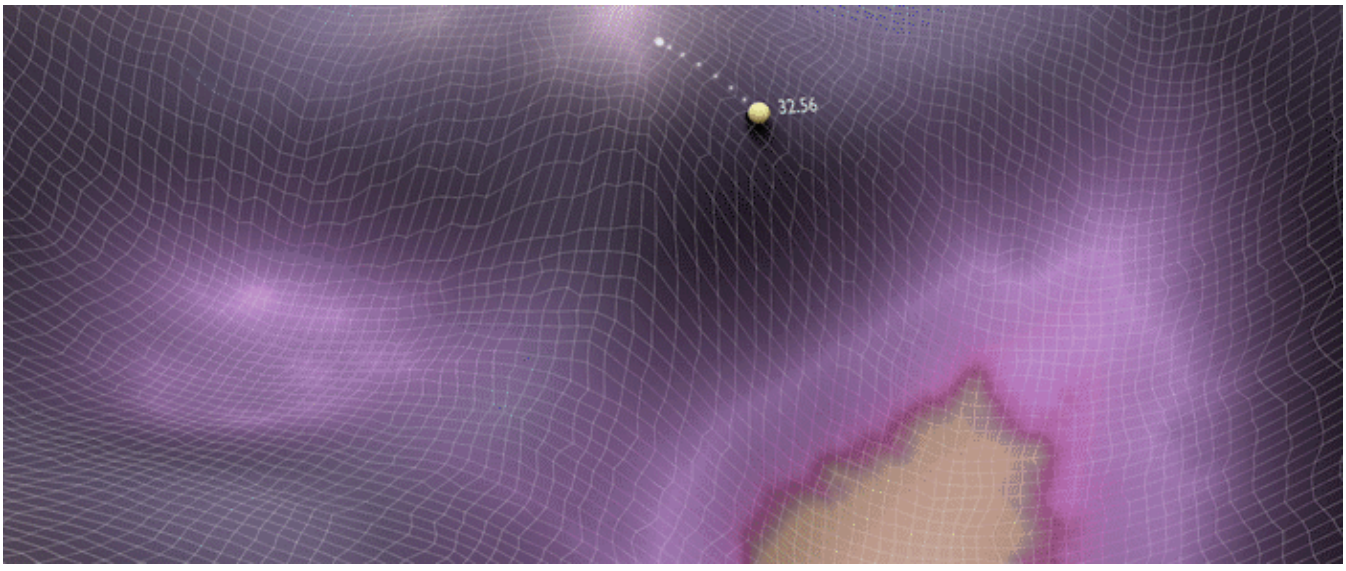
Published in Towards Data Science

Javier Ideami Follow

Feb 12, 2019 · 24 min read · ✨ · Listen

Predict malignancy in cancer tumors with your own neural network

In the final part of this series, we predict malignancy in breast cancer tumors using the network we coded from scratch.



In part 1 of this series, we understood in depth **the architecture of our neural network. In part 2, we built it using Python.** We also understood in depth back-propagation and the gradient descent optimization algorithm.

In the **final part 3,** we will use the **Wisconsin Cancer data-set.** We will learn to **prepare our data, run it through our network and analyze** the results.

It's time to explore the loss landscape of our network.

Navigating the Loss Landscape within deep learning training processes. Variations include: Std SGD, LR annealing, large LR or SGD+momentum. Loss values modified & scaled to facilitate visual contrast. Visuals by Javier Ideami@ideami.com

Switching on the network

To switch on our network, **we need some fuel,** we need **data.**

- We will use a real data-set connected to the detection of breast cancer tumors.
- The data comes from **The Wisconsin Cancer Data-set.**
- This data was gathered by the University of Wisconsin Hospitals, Madison and by Dr. William H. Wolberg.
- By request of the owners of the data, we mention one of the studies linked to the data-set: O. L. Mangasarian and W. H. Wolberg: "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, September 1990, pp 1 & 18.
- The data, in csv format, can be downloaded from this link
- At this **Github link**, you can access all the code and data of the project.

javismiles/Deep-Learning-predicting-breast-cancer-tumor-malignancy

Predicting Cancer Malignancy with a 2 layer neural network coded from scratch in Python. ...

github.com

First, we download the data to our machine. Then, we use pandas to create a **dataframe and we take a look at its first rows.**

```
df = pd.read_csv('wisconsin-cancer-dataset.csv',header=None)
df.head(5)
```

	0	1	2	3	4	5	6	7	8	9	10
0	1000025	5	1	1	1	2	1	3	1	1	2
1	1002945	5	4	4	5	7	10	3	2	1	2
2	1015425	3	1	1	1	2	2	3	1	1	2
3	1016277	6	8	8	1	3	4	3	7	1	2
4	1017023	4	1	1	3	2	1	3	1	1	2
5	1017122	8	10	10	8	7	10	9	7	1	4
6	1018099	1	1	1	1	2	10	3	1	1	2
7	1018561	2	1	2	1	2	1	3	1	1	2
8	1033078	2	1	1	1	2	1	1	1	5	2
9	1033078	4	2	1	1	2	1	2	1	1	2
10	1035283	1	1	1	1	1	1	3	1	1	2
11	1036172	2	1	1	1	2	1	2	1	1	2
12	1041801	5	3	3	3	2	3	4	4	1	4
13	1043999	1	1	1	1	2	3	3	1	1	2
14	1044572	8	7	5	10	7	9	5	5	4	4

A dataframe is a python data structure that allows us to work and visualize data very easily.

The first thing we need to do is to understand the structure of the data. We find key information about it on its website.

- **There are 699 rows in total, belonging to 699 patients**
- The first column is an **ID** that identifies each patient.
- The following **9 columns are features** that express different types of information connected to the detected tumors. They represent data related to: Clump Thickness, Uniformity of Cell Size, Uniformity of Cell Shape, Marginal

Adhesion, Single Epithelial Cell Size, Bare Nuclei, Bland Chromatin, Normal Nucleoli and Mitoses.

- The last column is the class of the tumor and it has two possible values: **2** means that the tumor was found to be **benign. 4** means that it was found to be **malignant.**
- We are told as well that there are a few rows that contain missing data. The **missing data** is represented in the data-set with the **?** character.
- Out of the 699 patients in the dataset, the class distribution is: **Benign: 458 (65.5%) and Malignant: 241 (34.5%)**

This is useful information that allows us to achieve some conclusions.

- Our objective will be to train our neural network to **predict if a tumor is benign or malignant,** based on the features provided by the data.
- The **input to the network** will be made of **the 9 features, the 9 columns** that express different **features** of the tumors.
- We will not make use of the first column that holds the ID of the patient.
- We will **eliminate** from the data-set any **rows that contain missing data** (identified with the ? character).
- In binary classification scenarios, It's good to have a good percentage of data from both classes. We have a **65%-35% distribution,** which is good enough.
- The benign and malignant **classes** are **identified with the digits 2 and 4.** The last layer of our network outputs values between 0 and 1 through its **Sigmoid** function. Furthermore, neural networks tend to work better when data is set in that range, from 0 to 1. We will therefore change the values of the class columns to hold a 0 instead of a 2 for benign cases and a 1 instead of a 4 for malignant cases. (we could also scale the output of the Sigmoid instead).

We proceed to do these changes. First, we change the class values (at the column number 10) from 2 to 0 and from 4 to 1

```
df.iloc[:,10].replace(2, 0,inplace=True)
```

```
df.iloc[:,10].replace(4, 1,inplace=True)
```

Then we proceed to eliminate all rows that hold missing values (represented by the ? character) at column 6, which we have identified as the column that holds them.

```
df = df[~df[6].isin(['?'])]
```

The '?' character causes Python to interpret column 6 as made of strings. Other columns are made of integers. We set the entire dataframe to be interpreted as made of float numbers. This helps our network perform complex computations.

```
df = df.astype(float)
```

Next, let's deal with the range of the values in our data. Notice how the data within the 9 features is made of numbers that go beyond the 0 to 1 range. Real data-sets are often messy and come with a great diversity of range in their values: negative numbers, huge range differences within columns, etc.

That's why **data normalization** is a key first step within the **feature engineering** phase of deep learning processes.

Normalizing the **data** means **preparing** it **in a way that is easier for the network to digest**. We are helping the network converge easier and faster to that minima we are looking for. Typically, neural networks respond well to numerical data set in the 0 to 1 range, and also to data that has a mean of 0 and a standard deviation of 1.

Feature engineering and normalization are not the focus of this article but let's quickly mention a couple of methods within this phase of the feature engineering process:

- An example of a normalization method would be re-scaling our data to fit within the 0 to 1 range by applying the **min-max** method to each feature column. **new_x= (x- min_x)/(max_x- min_x)**
- We can also apply **standardization**, which centers the values of each feature column, setting it to have a mean of 0 and a standard deviation of 1.

new_x = (x- mean)/std.dev

Some data-sets and scenarios will benefit more than others from each of these techniques. In our case and after some tests, we decide to apply min-max normalization using the sklearn library:

```
names = df.columns[0:10]
scaler = MinMaxScaler()
scaled_df = scaler.fit_transform(df.iloc[:,0:10])
scaled_df = pd.DataFrame(scaled_df, columns=names)
```

Let's take a look at the same 15 rows after all these changes.

	0	1	2	3	4	5	6	7	8	9	10
0	0.069946	0.444444	0.000000	0.000000	0.000000	0.111111	0.000000	0.222222	0.000000	0.000000	0.0
1	0.070164	0.444444	0.333333	0.333333	0.444444	0.666667	1.000000	0.222222	0.111111	0.000000	0.0
2	0.071096	0.222222	0.000000	0.000000	0.000000	0.111111	0.111111	0.222222	0.000000	0.000000	0.0
3	0.071160	0.555556	0.777778	0.777778	0.000000	0.222222	0.333333	0.222222	0.666667	0.000000	0.0
4	0.071216	0.333333	0.000000	0.000000	0.222222	0.111111	0.000000	0.222222	0.000000	0.000000	0.0
5	0.071223	0.777778	1.000000	1.000000	0.777778	0.666667	1.000000	0.888889	0.666667	0.000000	1.0
6	0.071296	0.000000	0.000000	0.000000	0.000000	0.111111	1.000000	0.222222	0.000000	0.000000	0.0
7	0.071331	0.111111	0.000000	0.111111	0.000000	0.111111	0.000000	0.222222	0.000000	0.000000	0.0
8	0.072415	0.111111	0.000000	0.000000	0.000000	0.111111	0.000000	0.000000	0.000000	0.444444	0.0
9	0.072415	0.333333	0.111111	0.000000	0.000000	0.111111	0.000000	0.111111	0.000000	0.000000	0.0
10	0.072579	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.222222	0.000000	0.000000	0.0
11	0.072646	0.111111	0.000000	0.000000	0.000000	0.111111	0.000000	0.111111	0.000000	0.000000	0.0
12	0.073066	0.444444	0.222222	0.222222	0.222222	0.111111	0.222222	0.333333	0.333333	0.000000	1.0
13	0.073230	0.000000	0.000000	0.000000	0.000000	0.111111	0.222222	0.222222	0.000000	0.000000	0.0
14	0.073273	0.777778	0.666667	0.444444	1.000000	0.666667	0.888889	0.444444	0.444444	0.333333	1.0

- After the changes we have 683 rows. 16 that held missing data have been eliminated.
- All the columns are now made of float numbers and their values are normalized between 0 and 1. (Column 0, the IDs, will be ignored when we build the training set later).
- The final column, the class, is now using a value of 0 for benign tumors and of 1 for malignant ones.
- Notice that we are not normalizing the class column because it already holds values in the 0 to 1 range and its values should remain set to 0 or 1.

- And notice that the final column, the one we will use as our target, doesn't need to be a float. It can be an integer because our output can only be 1 or 0. (when we train the network, we will pick that column from the original **df** dataframe, where it is set to 0 or 1).

- Therefore, our **scaled_df** dataframe contains all the normalized columns, and we will pick the class column from the **df** dataframe, the non normalized version of the data-set.

The process could continue as we explore more the data.

- Are all the 9 features essential? do we want to include them all in the training process?

- Do we have enough quality data to produce good training, validation and test sets? (more about this later)

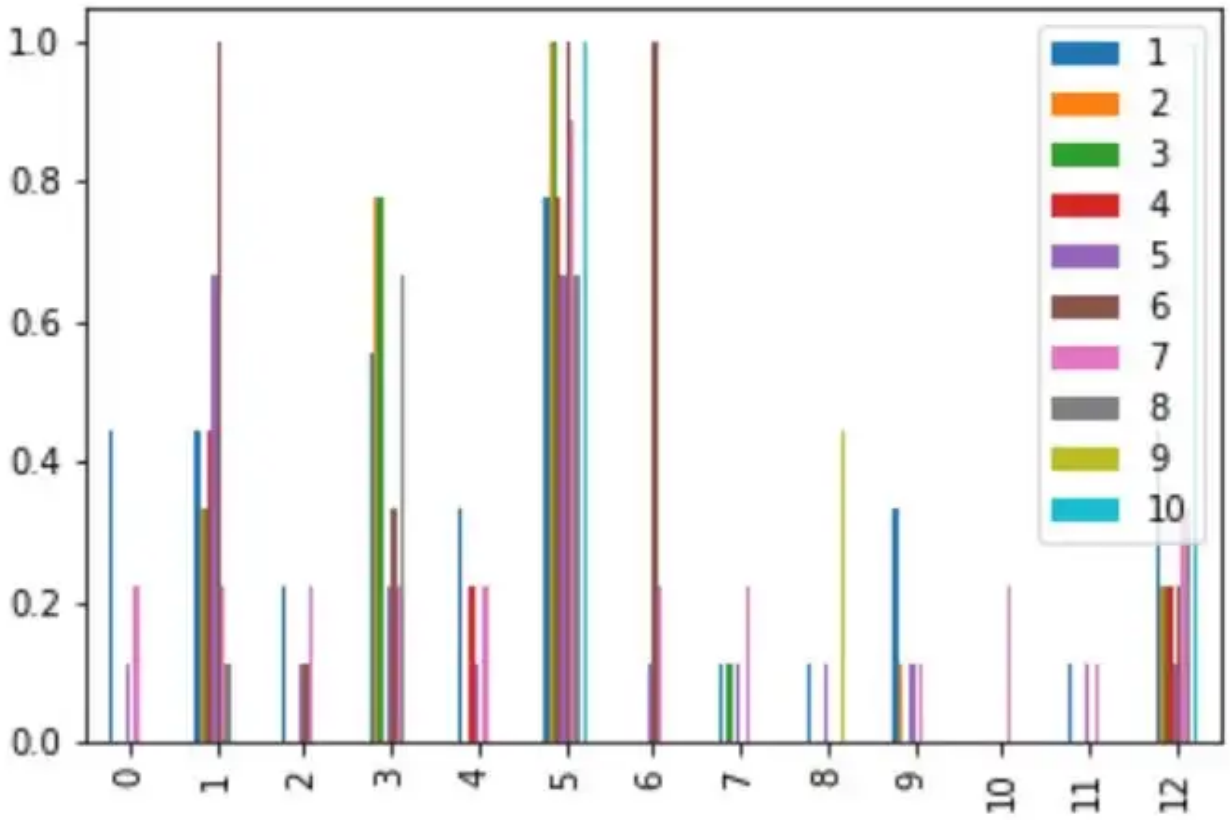
- Studying the data, do we find any meaningful and useful insights that may help us train the network more efficiently?

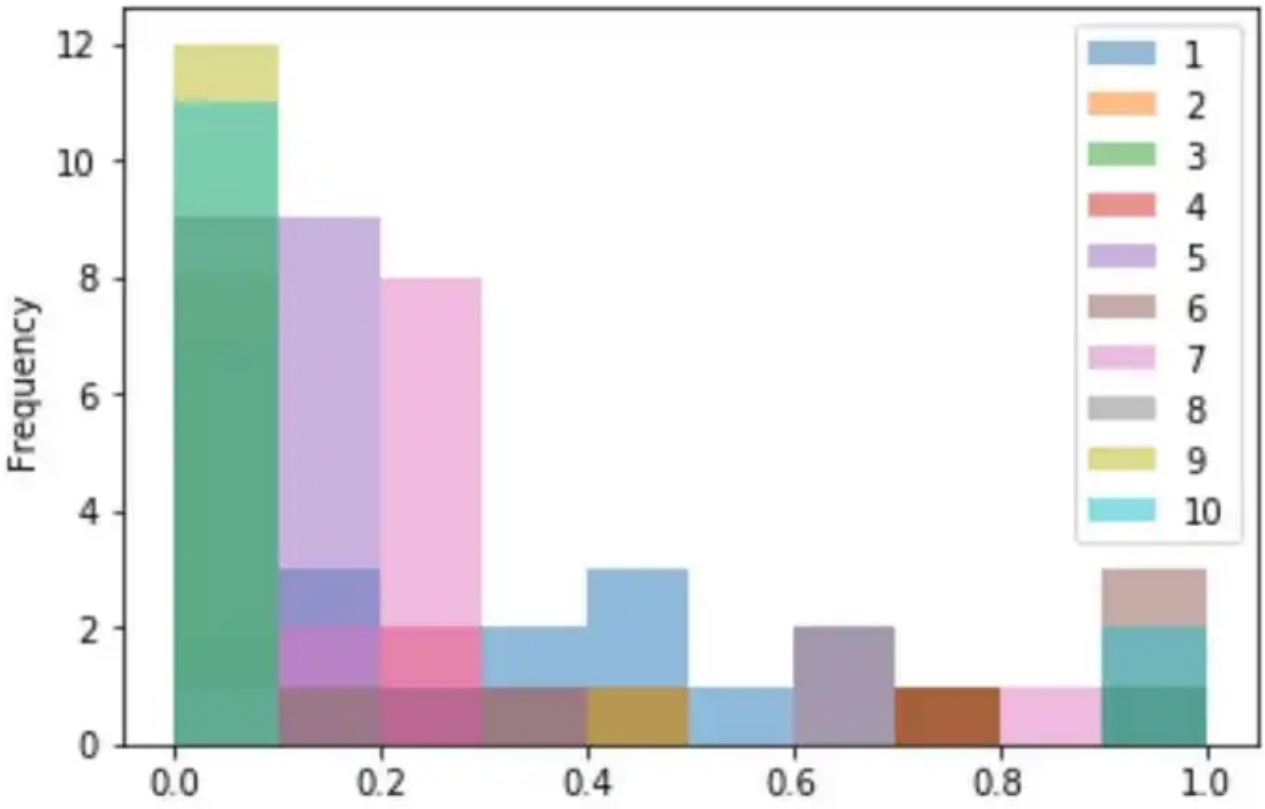
These and more are part of the **feature engineering process** that takes place before the training begins.

Another useful thing to do is to build charts to analyze the data in different ways. The myplotlib python library helps us study the data through different kinds of graphs.

We first combine the normalized columns we want to study with the class column, and then begin to explore.

```
scaled_df[10]= df[10]

scaled_df.iloc[0:13,1:11].plot.bar();
scaled_df.iloc[0:13,1:11].plot.hist(alpha=0.5)
```





Explore the full range of visualization options offered by Panda with this link

To speed up the article, in our case we conclude that the 9 features are useful. **Our objective is to predict the "class" column** with precision.

So, **how complex would the function be** that would describe the connection between our 683 samples and their output?

The relationship between the 9 features and the output is clearly multi-dimensional and non-linear.

Let's run the data through the network and see what happens.

Before we do that, we need to consider a key topic:

- If we use the 683 samples, all our samples, to create our training set and we get good results, we will have to face a key question.
- **What if** the network sets its weights in a way that matches perfectly the samples used during training, and yet **fails to generalize to new samples outside the training set**?
- Is **our final objective** achieving great **accuracy** when using the training data, or when **using data it has never seen before**? Obviously, the second case.

This is **the reason** why the deep learning practitioner typically takes into account **three kinds of data-sets**:

- **Training set:** the data you use to train the network. It contains the input features and the target labels.
- **Validation set:** a separate, different batch of data, which should ideally come from the same distribution than the training set. You will use it to verify the quality of the training. The validation set has as well target labels.
- **Test set**: another separate batch of data, used to test the network with fresh related data that ideally comes from the same distribution than the validation set. Typically, the test set doesn't come with target labels.

The size of the different sets in relation to each other is another topic that would take some time to describe. For our purposes, consider that most of the data forms the training set and a small percentage of it is typically extracted (and eliminated from the training set) to become the validation set.

20% is a typical number that is often chosen as the percentage of the data that will form our validation set.

To estimate the quality of the training of your network, it is useful to compare the performance of your training and validation sets:

- If the **loss value achieved on the validation set improves and then starts to get worse**, the network is **over-fitting**, meaning the network has learnt a function that fits very well the training data, yet **does not generalize well enough** to the validation set.
- The opposite of over-fitting is **under-fitting**, when the training performance of the network is not good enough and we obtain loss values that are too high in both the training and validation sets (and the training loss is worse than the validation loss, for example).
- Ideally you want to get similar performance in both data-sets.
- When we have **over-fitting**, we can apply **regularization**. Regularization is a technique that applies changes to the optimization algorithm that allow the network to generalize better. Regularization techniques include Dropout, L1 and L2 regularization, early stopping and data augmentation techniques.

In general, realize that **success with the validation set is your real target**. Having the network perform fantastically well with the training data serves no purpose if it fails to perform well with new data it hasn't seen before.

So your real target is to reach a good loss value and achieve good accuracy with your validation set.

To get there, over-fitting is one of the most important issues we need to prevent, and that's why regularization is so important. Let's quickly and briefly recap **4 widely used regularization techniques.**

Dropout: During each training pass, we randomly disable some of the hidden units of our network. This prevents the network from putting too much emphasis on any specific weight and helps the network generalize better. It is as if we were running the data through different network architectures and then averaging their impact, which helps prevent over-fitting.

L1 and L2: We add extra terms to the cost function that penalize the network when weights become too large. These techniques encourage the network to find a good balance between the loss value and the scale of the weights.

Early stopping: Over-fitting can be a consequence of training for too long. If we monitor our validation error, we can stop the training process when the validation error stops improving.

Data augmentation: Typically, more training data means a better network performance, but obtaining more data is not always possible. Instead, we can augment the existing data by artificially creating variations of it. For example, in the case of images, we can apply rotations, translations, cropping and other techniques to produce new variations of them.

Back to our data. It's time to pick our training and validation sets. We will select part of the 683 rows as the training set and a different part of the data-set as our **validation set.**

After the training, **we will validate the quality** of our network by running the process again through the validation set.

```
x=scaled_df.iloc[0:500,1:10].values.transpose()
y=df.iloc[0:500,10:].values.transpose()

xval=scaled_df.iloc[501:683,1:10].values.transpose()
yval=df.iloc[501:683,10:].values.transpose()
```

We decide to build our training set with 500 of the 683 rows, and we pick them from the normalized **scaled_df** dataframe. We also make sure to eliminate the first column (the IDs) and to not include the last column (the class) in the **input x** to the network

We declare **the target output y** using the class column that corresponds to the same 500 rows. We pick the class column from the original non-normalized **df** dataframe (as the class value should remain as a 0 or a 1).

We then select the next 183 rows for our validation set, and store them in the variables **xval** and **yval.**

We are ready. We will **first train the network** with the 500 rows of our x,y training set. **Afterwards, we will test the trained network with the 183 rows of our xval,yval validation set**, to see how well the network generalizes to data it has never seen before.

```
nn = dlnet(x,y)
nn.lr=0.01
nn.dims = [9, 15, 1]

nn.gd(x, y, iter = 15000)
```

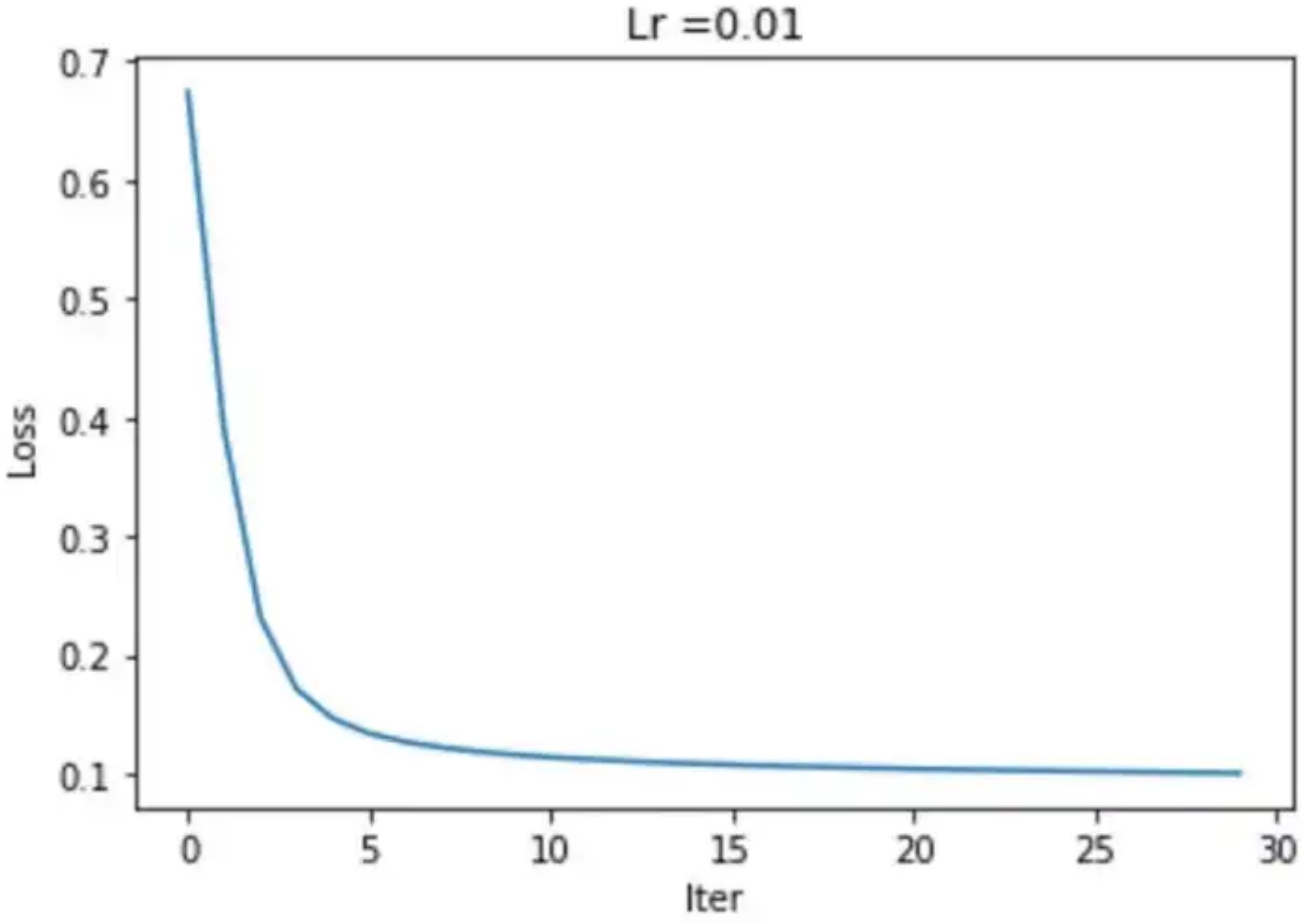
We declare our network, set a learning rate and the number of nodes at each layer (the input has 9 nodes because we are using 9 features, and it's not counted as a layer of the network. The first hidden layer has 15 hidden units and the second and final layer has a single output node).

We then run the gradient descent algorithm through a few thousand iterations. **Let's get a feel of how well the network trains with a few seconds of gradient descent.**

Every x iterations, we display the loss value of the network. If the training proceeds well, **the loss value should decrease after every cycle.**

```
Cost after iteration 0: 0.673967
Cost after iteration 500: 0.388928
Cost after iteration 1000: 0.231340
Cost after iteration 1500: 0.171447
Cost after iteration 2000: 0.146433
Cost after iteration 2500: 0.133993
Cost after iteration 3000: 0.126808
Cost after iteration 3500: 0.122107
Cost after iteration 12500: 0.101980
Cost after iteration 13000: 0.101604
Cost after iteration 14500: 0.100592
```

After a number of iterations our loss begins to stabilize at a low level. We plot a chart that follows the loss of the network through the iterations.



Our network seems to have trained quite well, reaching a low loss value (the distance between our predictions and the target outputs is low). But, **how good is it?** and most importantly, how good is it, not just on the whole training set, but way more important, **on our validation set**?

To find out, we create a new function, **pred()**, that runs a set of inputs through the network and then compares systematically every obtained output to its corresponding target output in order to **produce an average accuracy value.**

Notice below how the function studies if the prediction is above or below 0.5. We are doing binary classification and by default we consider that output values that are above 0.5 mean that the result belongs to one of the classes, and vice-versa.

In this case, because 1 is the class value for malignant tumors, we consider that outputs above 0.5 predict a malignant result, and below 0.5 the opposite. **We will talk in a bit about how, when and why we would want to change this 0.5 threshold value.**

```
def pred(self,x, y):
        self.X=x
        self.Y=y
        comp = np.zeros((1,x.shape[1]))
        pred, loss= self.forward()

        for i in range(0, pred.shape[1]):
            if pred[0,i] > 0.5: comp[0,i] = 1
            else: comp[0,i] = 0

        print("Acc: " + str(np.sum((comp == y)/x.shape[1])))

        return comp
```

We now proceed to compare the accuracy of the network when using the training and validation sets, by calling the **pred** function twice, once with our training set, and another time with our validation set.

```
pred_train = nn.pred(x, y)
pred_test = nn.pred(xval, yval)
```

And we get these 2 results.

```
Acc: 0.9620000000000003
Acc: 1.0
```

The network has an accuracy of a 96% on the training set (first 500 rows) and of 100% when using the validation set (next 183 rows).

The accuracy on the validation set is higher. This means that the network is not over-fitting and is **generalizing well enough** to be able to adapt to data it has never seen before.

We can now use the nn.forward() function to compare directly the first few values of the validation set output in relation to the target output:

```
nn.X,nn.Y=xval, yval
yvalh, loss = nn.forward()
print("\ny",np.around(yval[:,0:50,], decimals=0).astype(np.int))
```

```
print("\nyh",np.around(yvalh[:,0:50,],
decimals=0).astype(np.int),"\n")
```

And we get

```
y [[0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 1]]

yh [[0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

Both **match perfectly**, because we have achieved **100% accuracy** on our validation set.

Therefore, the function learnt pretty well to **adapt to both the training and validation sets.**

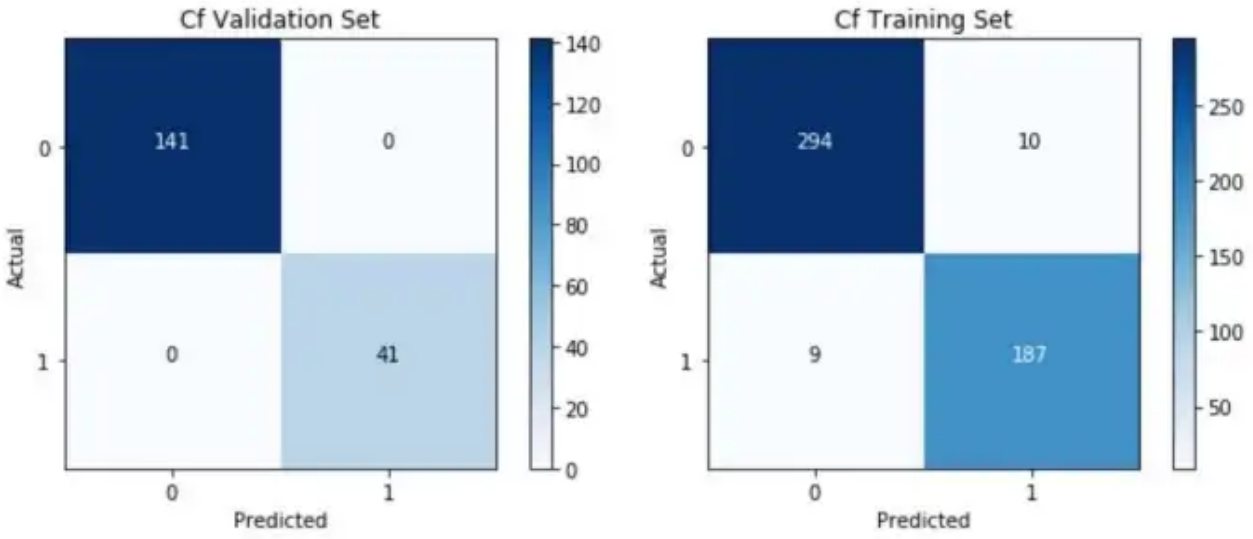
One great way to analyze the accuracy is by plotting a confusion matrix. First, we declare a custom plotting function.

```
def plotCf(a,b,t):
    cf =confusion_matrix(a,b)
    plt.imshow(cf,cmap=plt.cm.Blues,interpolation='nearest')
    plt.colorbar()
    plt.title(t)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    tick_marks = np.arange(len(set(expected))) # length of classes
    class_labels = ['0','1']
    tick_marks
    plt.xticks(tick_marks,class_labels)
    plt.yticks(tick_marks,class_labels)
    # plotting text value inside cells
    thresh = cf.max() / 2.
    for i,j in
itertools.product(range(cf.shape[0]),range(cf.shape[1])):

plt.text(j,i,format(cf[i,j],'d'),horizontalalignment='center',color=
'white' if cf[i,j] >thresh else 'black')
    plt.show();
```

(This custom confusion matrix function comes from this public Kaggle created by JP)

Then, we run the pred function again twice, and plot confusion matrices for both the training and validation sets.

```
nn.X,nn.Y=x, y
target=np.around(np.squeeze(y), decimals=0).astype(np.int)
predicted=np.around(np.squeeze(nn.pred(x,y)),
decimals=0).astype(np.int)
plotCf(target,predicted,'Cf Training Set')

nn.X,nn.Y=xval, yval
target=np.around(np.squeeze(yval), decimals=0).astype(np.int)
predicted=np.around(np.squeeze(nn.pred(xval,yval)),
decimals=0).astype(np.int)
plotCf(target,predicted,'Cf Validation Set')
```



We can see even more clearly that our validation set has perfect accuracy on its 183 samples. As for the training set, there are 19 mistakes among the 500 samples.

Now, at this point you may say that in a topic as delicate as diagnosing a tumor, setting our prediction to be 1 if the sigmoid output gives a value above 0.5 is not really good. **The network should be really confident before giving a prediction of malignancy.**

I totally agree, that's very correct. And these are the kinds of decisions that you need to take depending on the nature of the challenge and topic you are dealing with.

Let's then create a new variable called **threshold**. It will control our **confidence threshold, how close to 1 the output of the network needs to be** before we decide that a tumor is malignant. By default we set it to 0.5

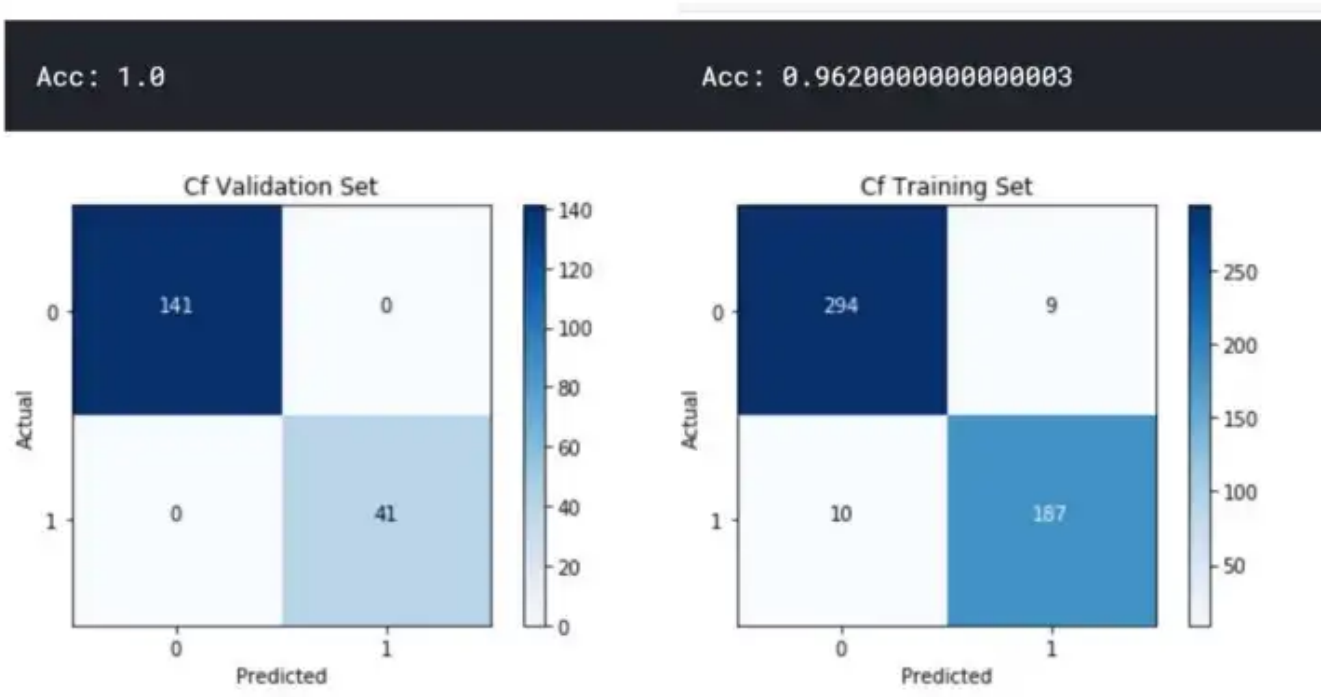
```
self.threshold=0.5
```

Out prediction function is now updated to use that **confidence threshold.**

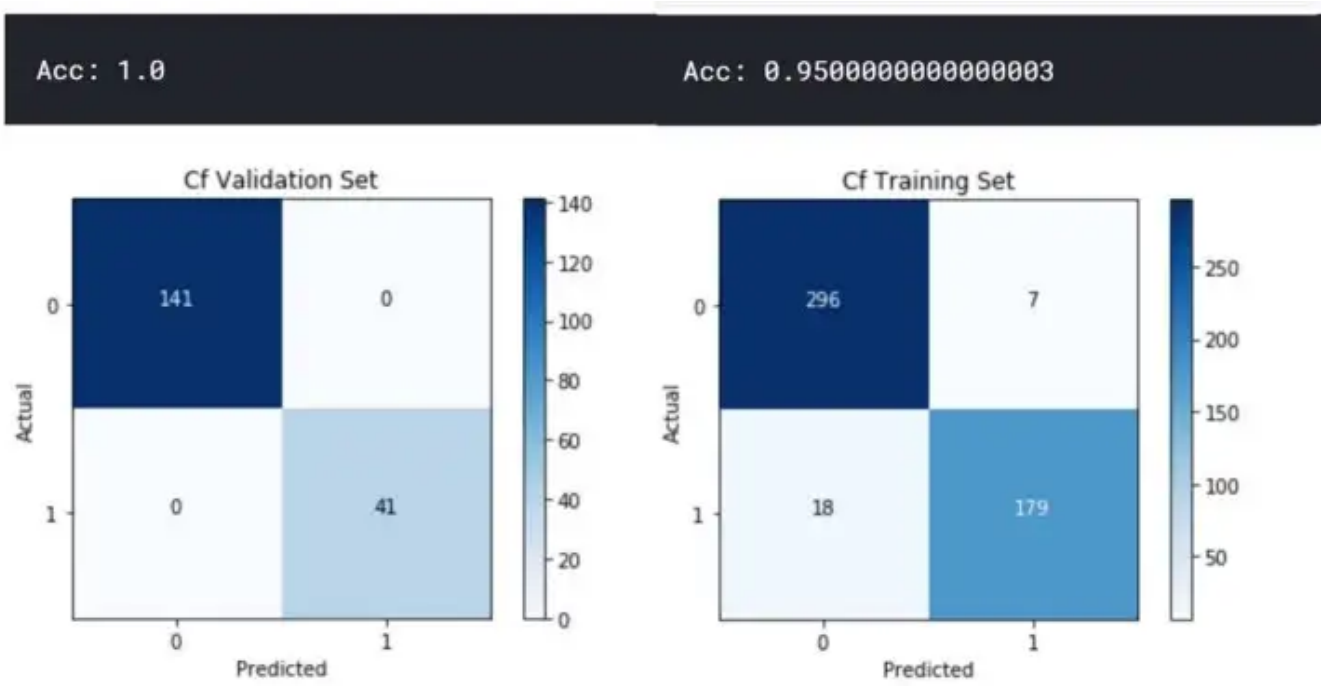
```
def pred(self,x, y):
        self.X=x
        self.Y=y
        comp = np.zeros((1,x.shape[1]))
        pred, loss= self.forward()

        for i in range(0, pred.shape[1]):
            if pred[0,i] > self.threshold: comp[0,i] = 1
            else: comp[0,i] = 0

        print("Acc: " + str(np.sum((comp == y)/x.shape[1])))

        return comp
```

Let's now compare our results as we gradually raise the confidence threshold.

Confidence threshold: 0.5 . Output values need to be higher than 0.5 for the output to be considered malignant. As seen previously, the validation accuracy is 100%, the training one is 96%.

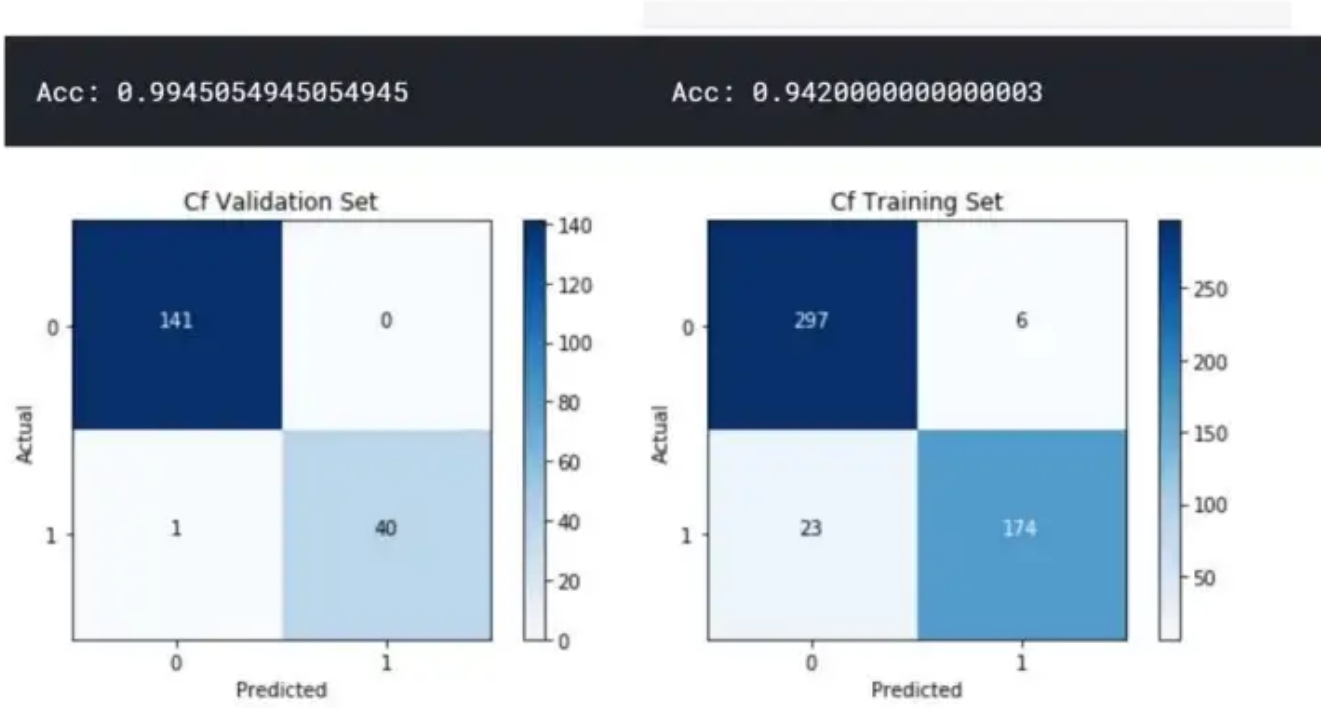


Confidence threshold: 0.7 . Output values need to be higher than 0.7 for the output to be considered malignant. The validation accuracy remains at 100%, the training one decreases a bit to 95%.



Confidence threshold: 0.8 . Output values need to be higher than 0.8 for the output to be considered malignant. The validation accuracy for the first time decreases very, very slightly to 99.45%. In the confusion matrix we see that 1 single sample of

the 183 is not recognized correctly. The training accuracy decreases a bit more till 94.2%



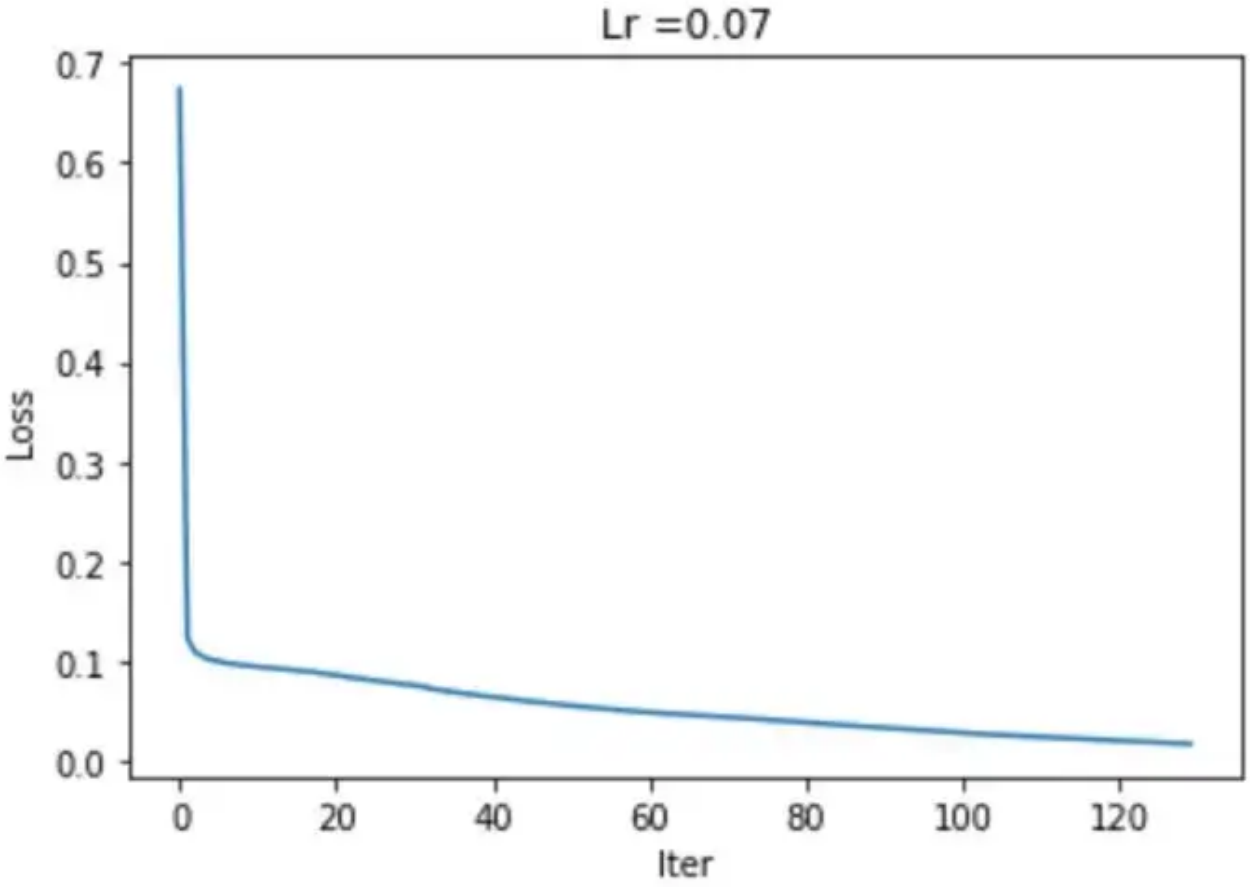
Confidence threshold: 0.9. Finally, in the case of 0.9, output values need to be higher than 0.9 for the output to be considered malignant. We are looking for almost complete confidence. The validation accuracy decreases a bit more till 98.9%. In the confusion matrix we see that 2 samples of the 183 were not recognized correctly. The training accuracy decreases further till 92.6%.

Therefore, by controlling the confidence threshold, we adapt to the specific needs of our challenge.

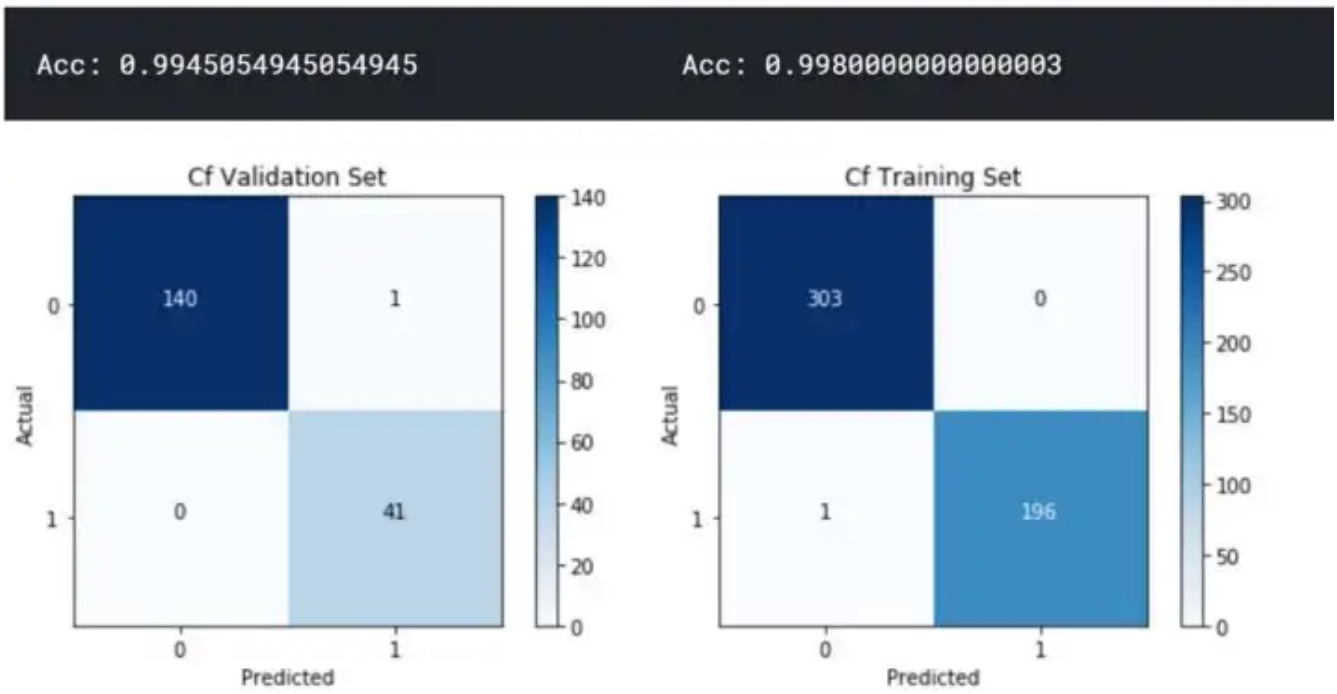
If we want to lower the loss value related to our training set (because we are failing to recognize a small percentage of the training samples), we can try to train for longer, and also use different learning rates.

For example, if we set the learning rate to 0.07 and train for 65000 iterations, we obtain:

```
Cost after iteration 63500: 0.017076
Cost after iteration 64000: 0.016762
Cost after iteration 64500: 0.016443
Acc: 0.9980000000000003
Acc: 0.9945054945054945
```



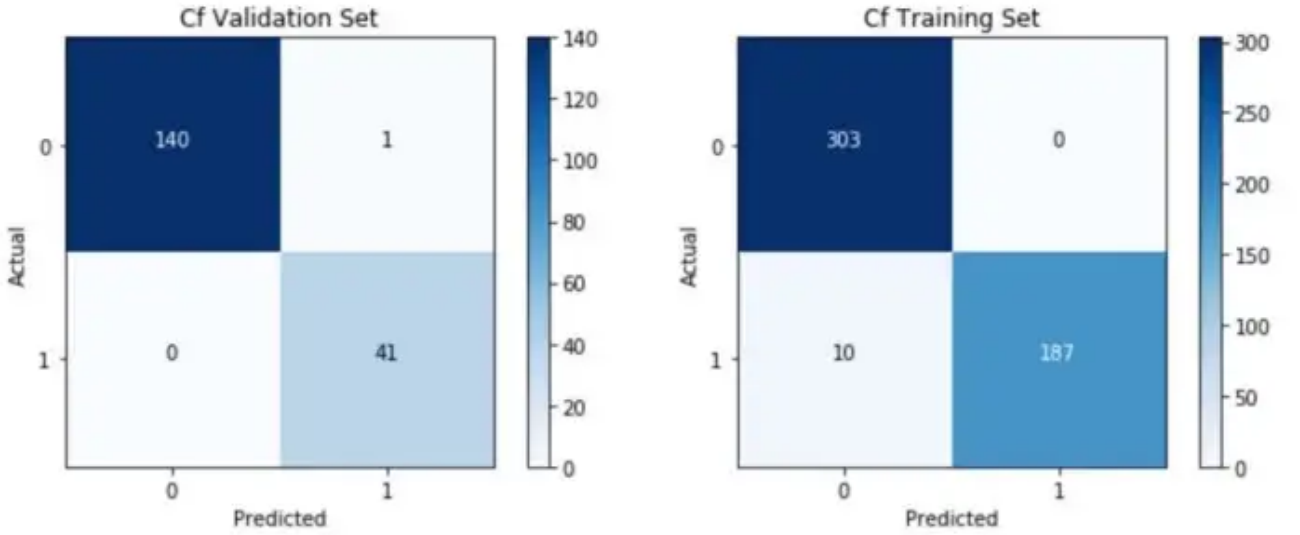
Now, with our confidence threshold set to 0.5, the network is accurate with every sample in both sets, except with one of each.



If we raise the confidence threshold to 0.7, performance is still excellent, only 1 validation sample and 2 training samples are not predicted correctly.

Finally, if we are really demanding and set the confidence threshold to 0.9, the network fails to guess correctly 1 of the validation samples and 10 of the training ones.

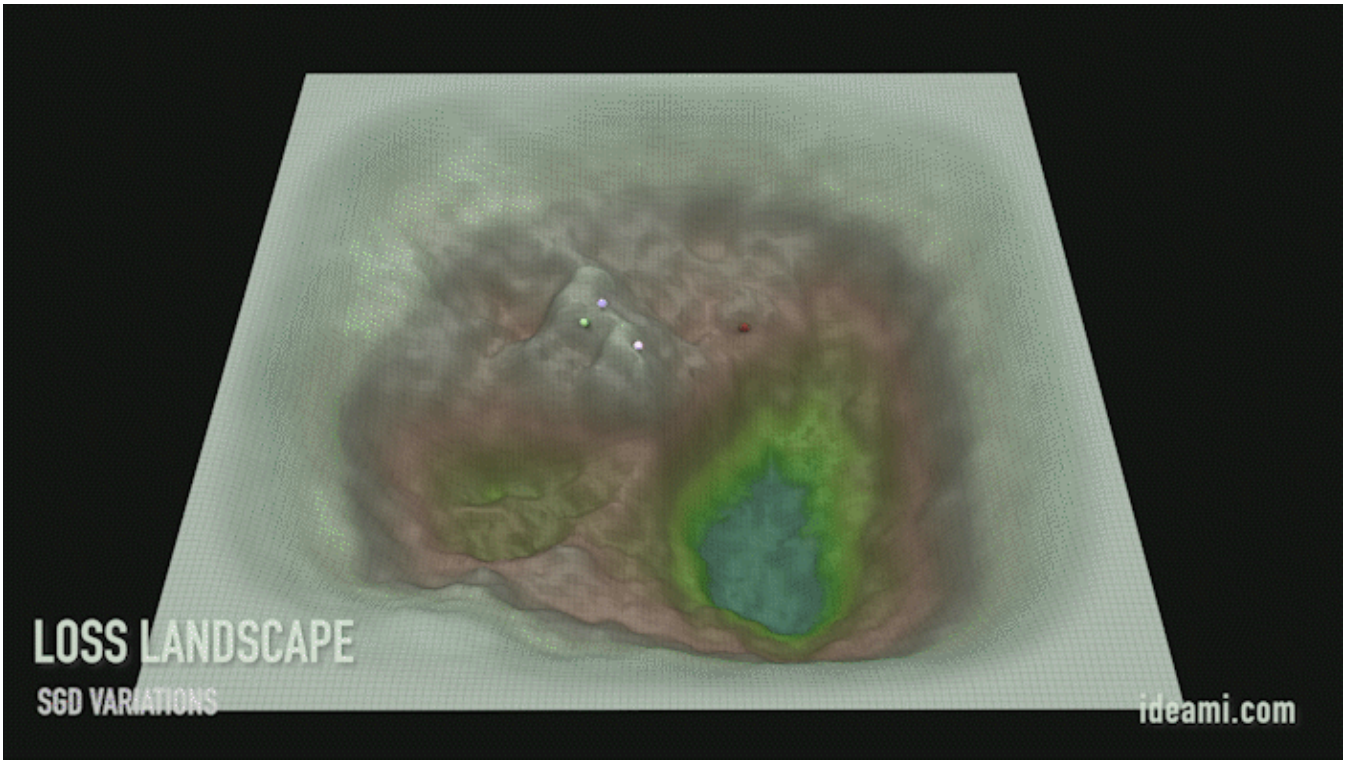
```
Acc: 0.9945054945054945          Acc: 0.9800000000000003
```



Although we have done quite well, considering that we are using a basic network without regularization, it is typical for things to get much harder when you are dealing with more complex data.

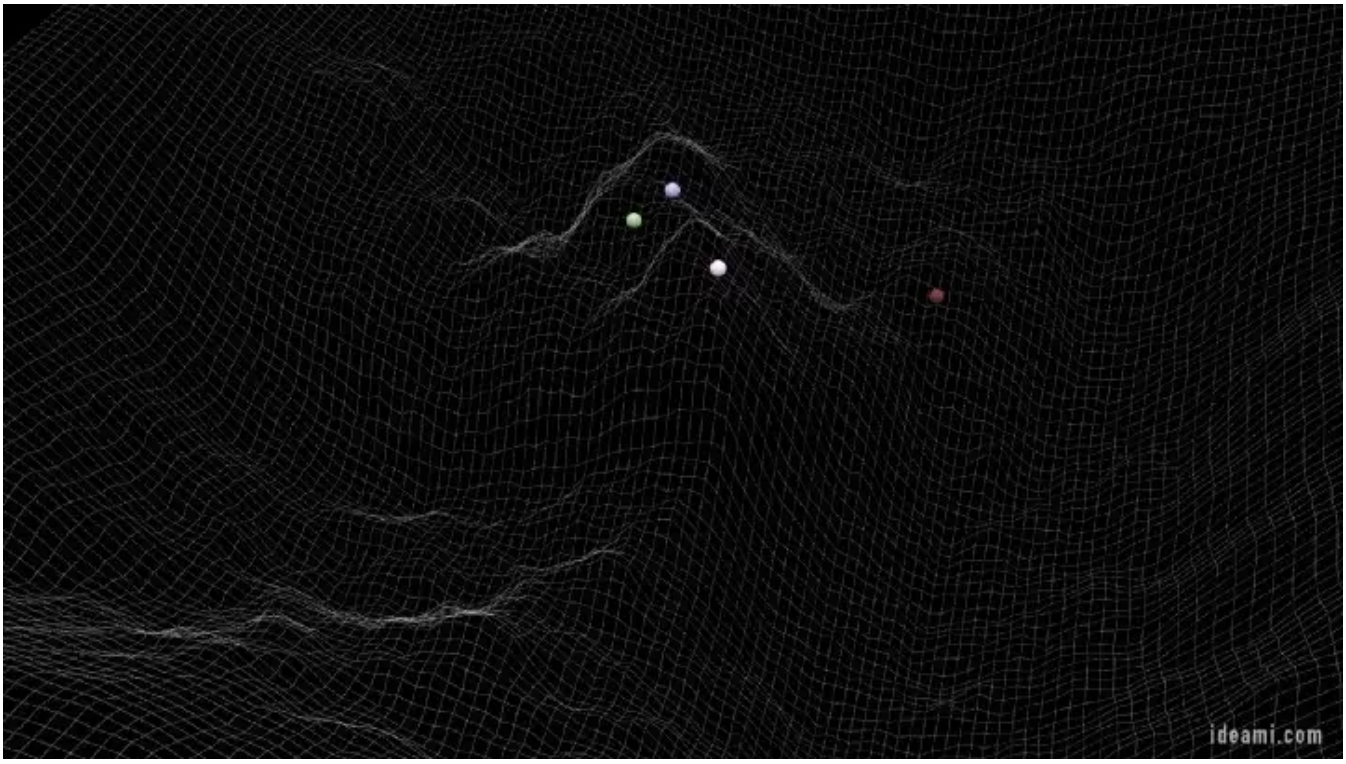
Often, the loss landscape **gets very complex** and it's **easier** to fall in the wrong local minima or **fail to converge** to a good enough loss.

Also, depending on the initial conditions of the network, we may converge to a good minima or we may get stuck at a plateau somewhere and fail to get out of it. It's useful at this stage to picture again our initial animation.



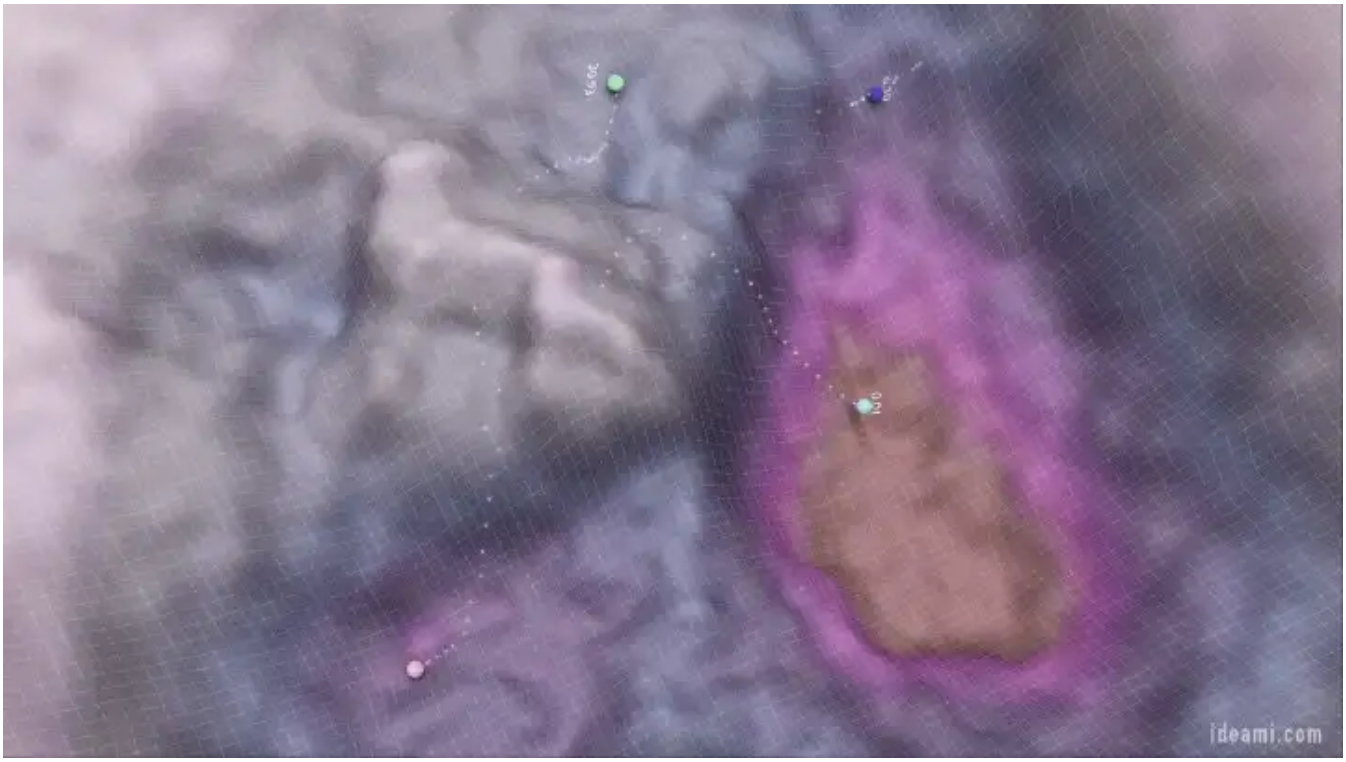
Navigating the Loss Landscape. Values have been modified and scaled up to facilitate visual contrast.

Picture that landscape, full of hills and valleys, places where the loss is really high, and places where the loss gets very low. The landscape of the loss function related to a complex scenario is often not uniform (though it can be made more smooth using different methods, but that's a whole different topic).



It's full of **hills and valleys** of different depths and angles. **The way you move around the landscape** is by changing the loss value of the network when you run the

gradient descent algorithm.

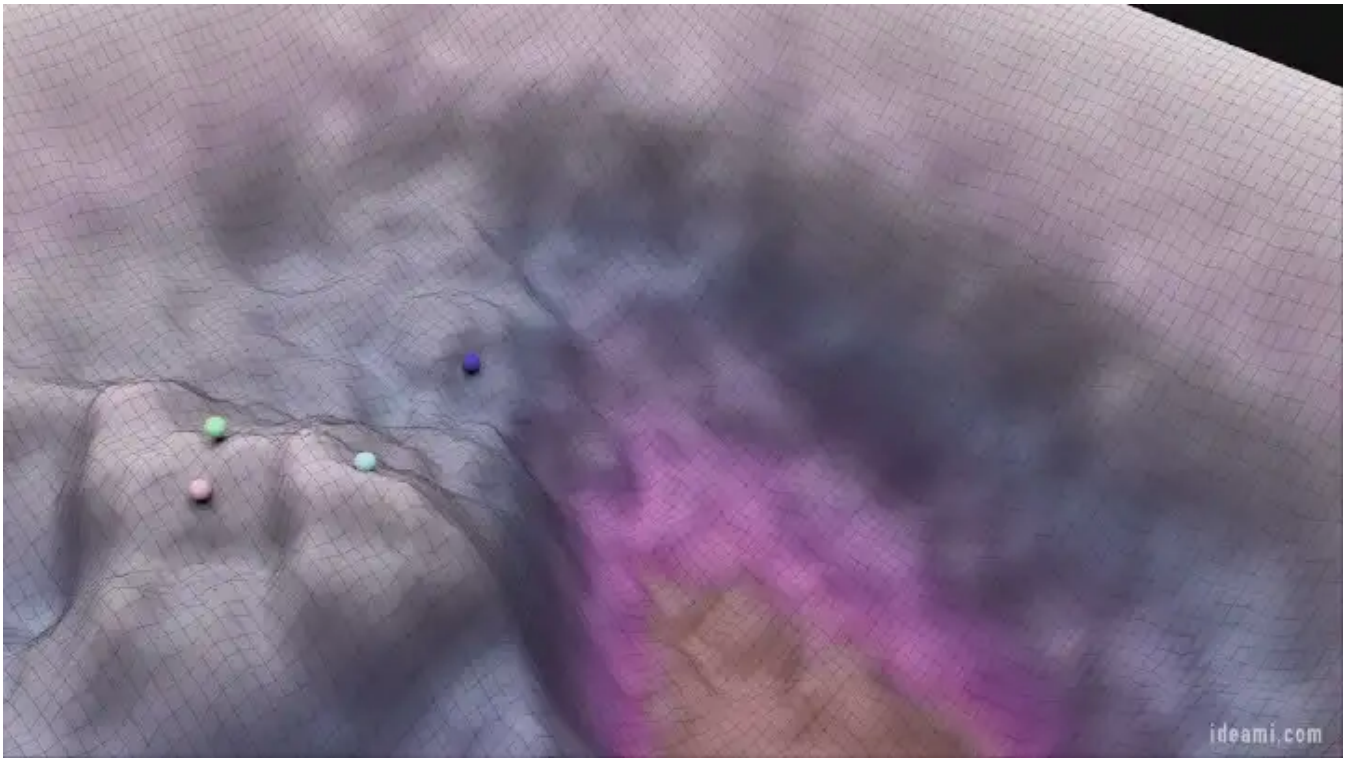


And the speed at which you move is controlled by the **learning rate:**

- If you are moving very slowly and somehow arrive to a plateau or a valley that is not low enough, you may get stuck there.
- If you move too fast, you may arrive to a low enough valley but rush through it and move away from it just as fast.

So there are some very delicate issues that have an enormous impact on how your network will perform.

- **The initial conditions:** in what part of the landscape do you drop the ball at the beginning of the process?



- **The speed** at which you move the ball, the learning rate.

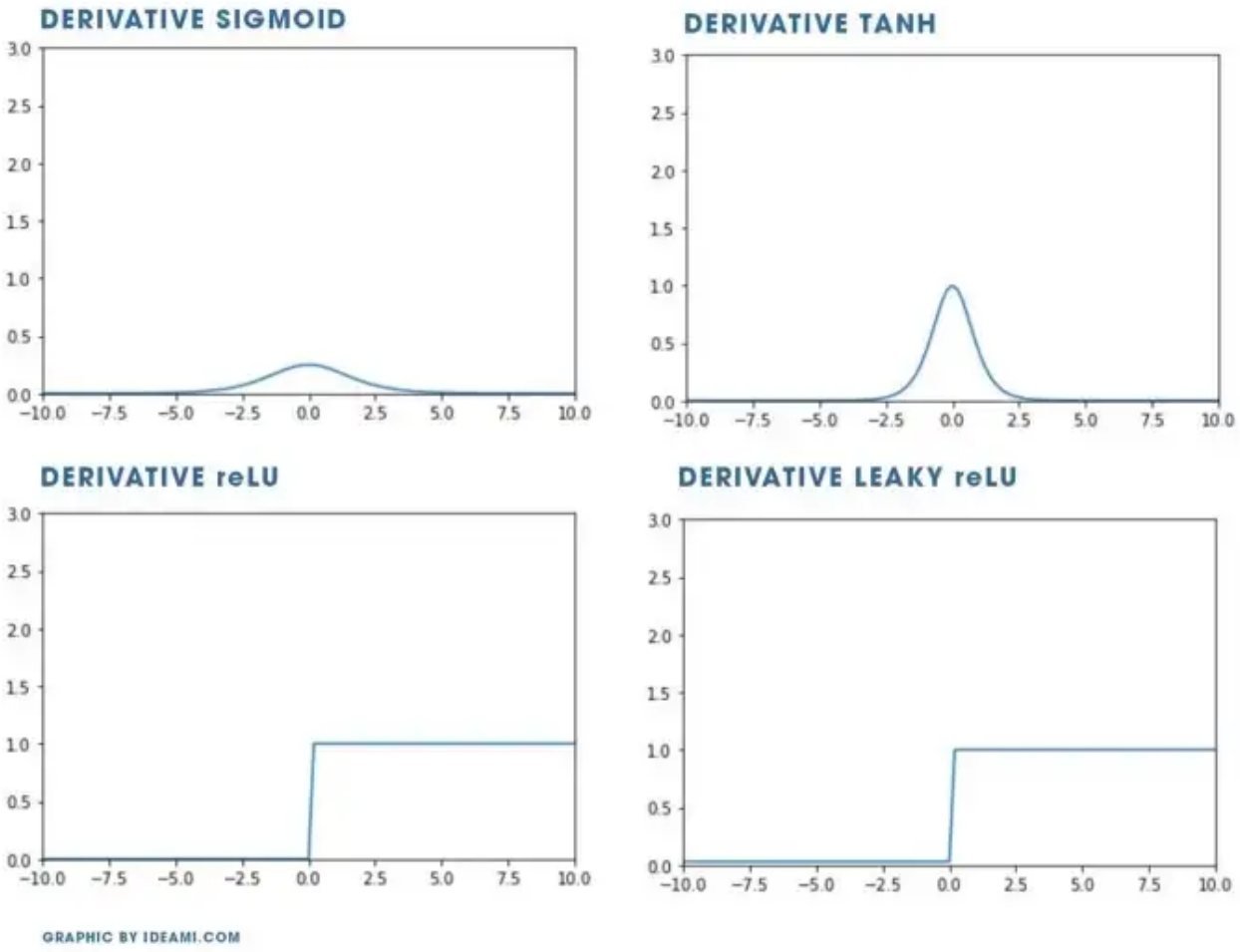
A lot of the **progress achieved recently** in improving the speed with which neural networks train is connected to different techniques that **dynamically manage the learning rate** and also to new ways of setting those initial conditions in better ways.

Regarding the initial conditions:

- Remember that each layer computes a combination of the weights and the inputs of the preceding layer (**weighted sum** of the inputs) and pass that computation to that layer's activation functions.
- Those activation functions have shapes that can either accelerate or stop all together the dynamics of the neurons, depending on the combination between the range of the inputs and the way they respond to that range.
- If the sigmoid function, for example, receives values that trigger a result that is close to the extremes of its output range, the output of the activation function on that part of its range becomes really flat. If it stays flat for some time, the derivative, the rate of change at that point becomes zero or very small.
- Recall that **it is the derivative what helps us decide in what direction to move next.** Therefore, if the derivative is not giving us meaningful information, it will be very difficult for the network to know in what direction to move next from that point.

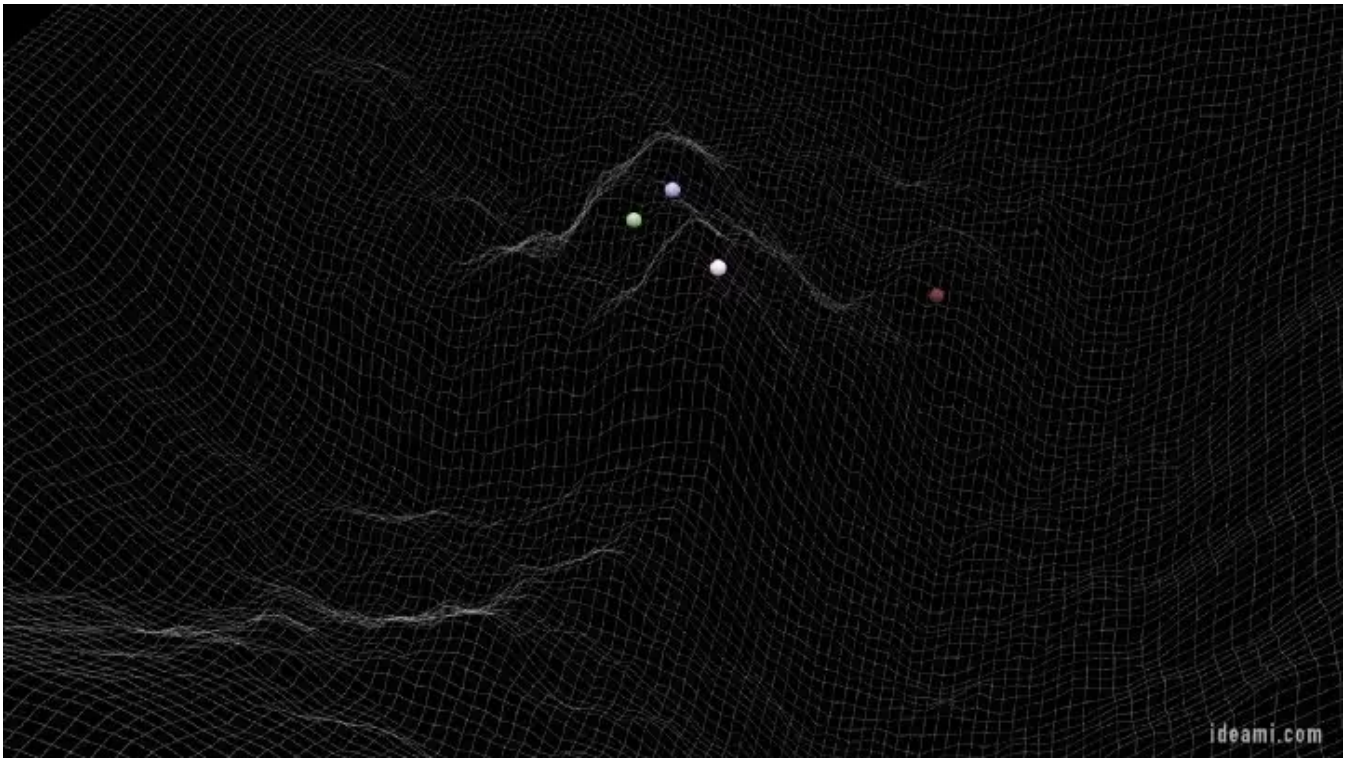
- It is as if you had reached a plateau in the landscape and you were really confused as to where to go next, and you just kept moving in circles around that point.

- This may happen also with ReLU, although ReLU has only 1 flat side as opposed to the 2 of Sigmoid and Tanh. **Leaky-ReLU** is a variation of ReLU that slightly modifies that side of the function (the flat one) to try to prevent vanishing gradients.



It is therefore **critical to set the initial values of our weights in the best way** possible so that the computations of the units at the start of the training process produce outputs that fall within the best possible range of our activation functions.

That could make the whole **difference between beginning at** a really high hill of the loss landscape or way lower.



Managing the learning rate to prevent the training process from being too slow or too fast, and to **adapt its value to the changing conditions** of the process and of each parameter, is another complex challenge

Talking about the many ways of dealing with the initial conditions and the learning rate would take a few articles. I will briefly describe some of them to give an idea of some of the methods experts use to deal with these challenges.

- **Xavier initialization:** A way of initializing our weights so that neuron's won't start in a saturated state (trapped at the delicate parts of their output ranges, where derivatives cannot provide enough information for the network to know where to go next).
- **Learning rate annealing**: high learning rates can push the algorithm to bypass and miss good minima at the loss landscape. A gradual decrease of the learning rate can prevent that. There are different ways to implement this decrease, including: exponential decay, step decay and 1/t decay.
- **Fast.ai Lr_find()**: An algorithm of the fast.ai library that finds the ideal range of values for the learning rate. **Lr_find** trains the model through a few iterations. It first tries to use a very low learning rate, and at each mini batch it changes the rate gradually until it reaches a very high value. The loss is recorded at each iteration and a chart helps us visualize the loss against the learning rate. We can

then decide what are the optimal values of the learning rate that decrease the loss in the most efficient way.

- **Differential learning rates**: Using different learning rates in different parts of our network.

- **SGDR, Stochastic Gradient Descent with Restarts**: Resetting our learning rate every x iterations. This can help us get out of plateaus or local minima that are not low enough, if we get stuck in one of them. A typical process is to start with a high learning rate. You then decrease it gradually at each mini batch. After x number of Epochs you reset it back to its initial high value and the same process repeats again. The concept is that moving gradually from a high rate to a lower one makes sense because we first quickly move down from the high points of the landscape (initial high loss value) and then move slower to prevent bypassing the minima of the landscape (low loss value areas). But if we get stuck at some plateau or a valley that is not low enough, restarting our rate to a high value every x iterations will help us jump out of that situation and continue exploring the landscape.

- **1 Cycle Policy:** A way of dynamically changing the learning rate proposed by Leslie N. Smith, in which we begin with a low rate value and gradually increase it until we reach a maximum. Then, we proceed to gradually decrease it till the end of the process. The initial gradual increase allows us to explore large areas of the loss landscape, increasing our chances of reaching a low area that is not bumpy; in the second part of the cycle, we settle in the low, flat area we have reached.

- **Momentum**: A variation of stochastic gradient descent that helps accelerate the path through the loss landscape while keeping the overall direction controlled. Recall that SGD can be noisy. Momentum averages the changes in the path, smooths that path and accelerates the movement towards the goal.

- **Adaptive learning rates:** Methods that calculate and use different learning rates for different parameters of the network.

- **AdaGrad (Adaptive Gradient Algorithm):**
Connecting with the previous point, AdaGrad is a variation of SGD that instead of using a single learning rate for all the parameters, uses a different rate for each parameter.

- **Root Mean Square Propagation** (RMSProp): Like Adagrad, RMSProp uses different learning rates for each parameter, and adapts those rates depending on the average of how fast they are changing (this helps when dealing with noisy contexts).

- **Adam**: It combines some aspects of RMSprop and SGDR with momentum. Like RMSprop, it uses squared gradients to scale the learning rate, and it also uses the average of the gradient to make use of momentum.

If you are new to all these names, don't get overwhelmed. **Behind most of them are the very same roots: back-propagation and gradient descent.**

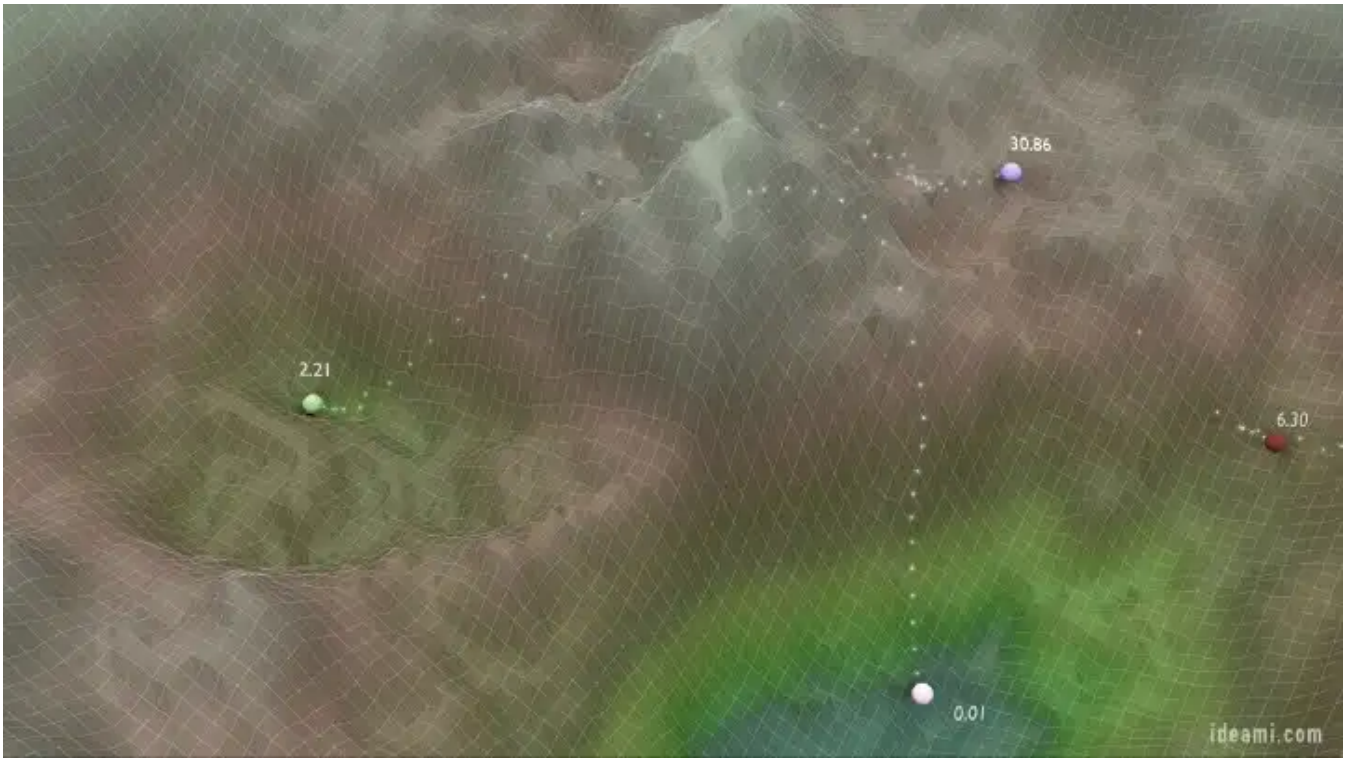
Also, a lot of these methods are selected automatically for you within modern frameworks such as the fast.ai library. It is though really useful to understand how they work, as you are then in a better position to take your own decisions and even to research and test different variations and options.

Understanding means more options

When we understand the core of the network, the basic back-propagation algorithm and the basic gradient descent process, we have more options to explore and experiment whenever we face hard challenges.

Because we understand the process, we realize for example that in deep learning, **the initial place where we drop the ball within the loss landscape is key.**

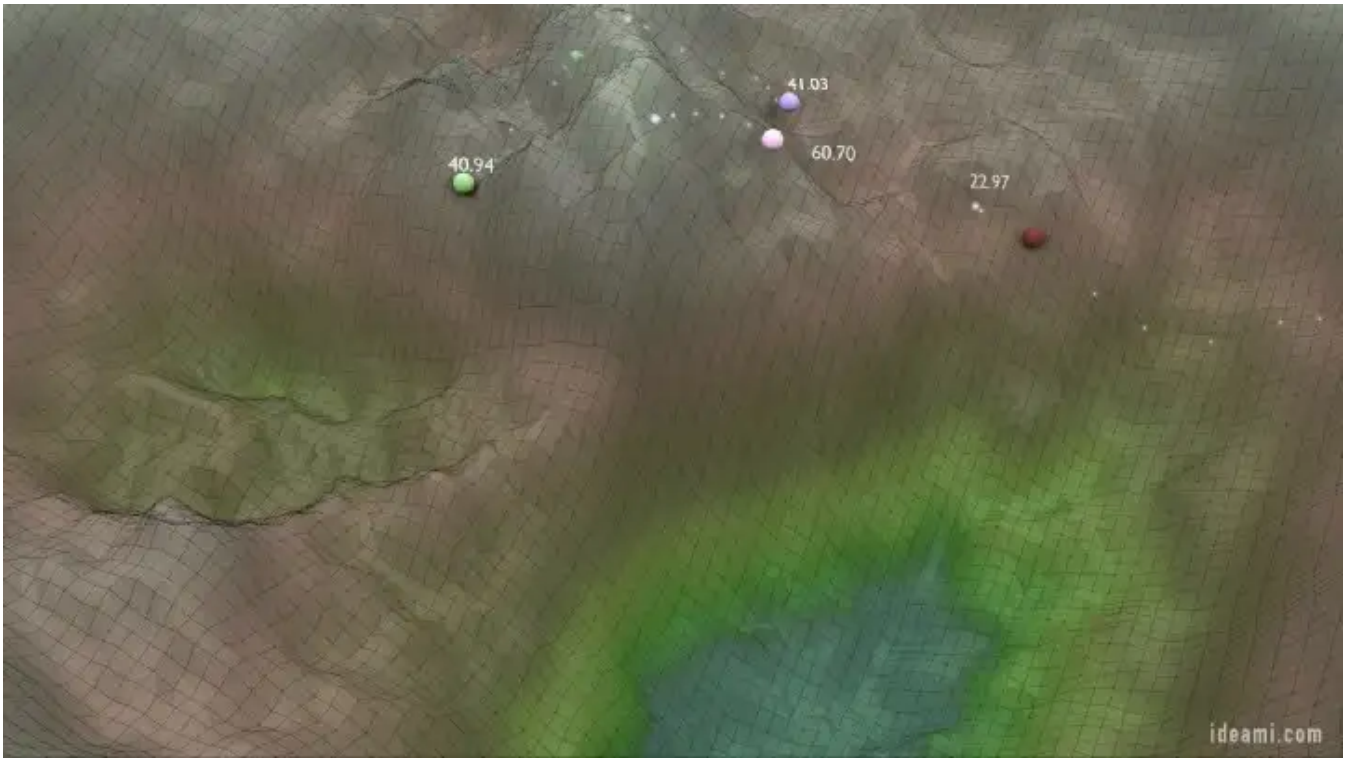
Some initial positions will soon push the ball (the training process) to get stuck in some part of the landscape. Others will quickly drive us to a good minima.

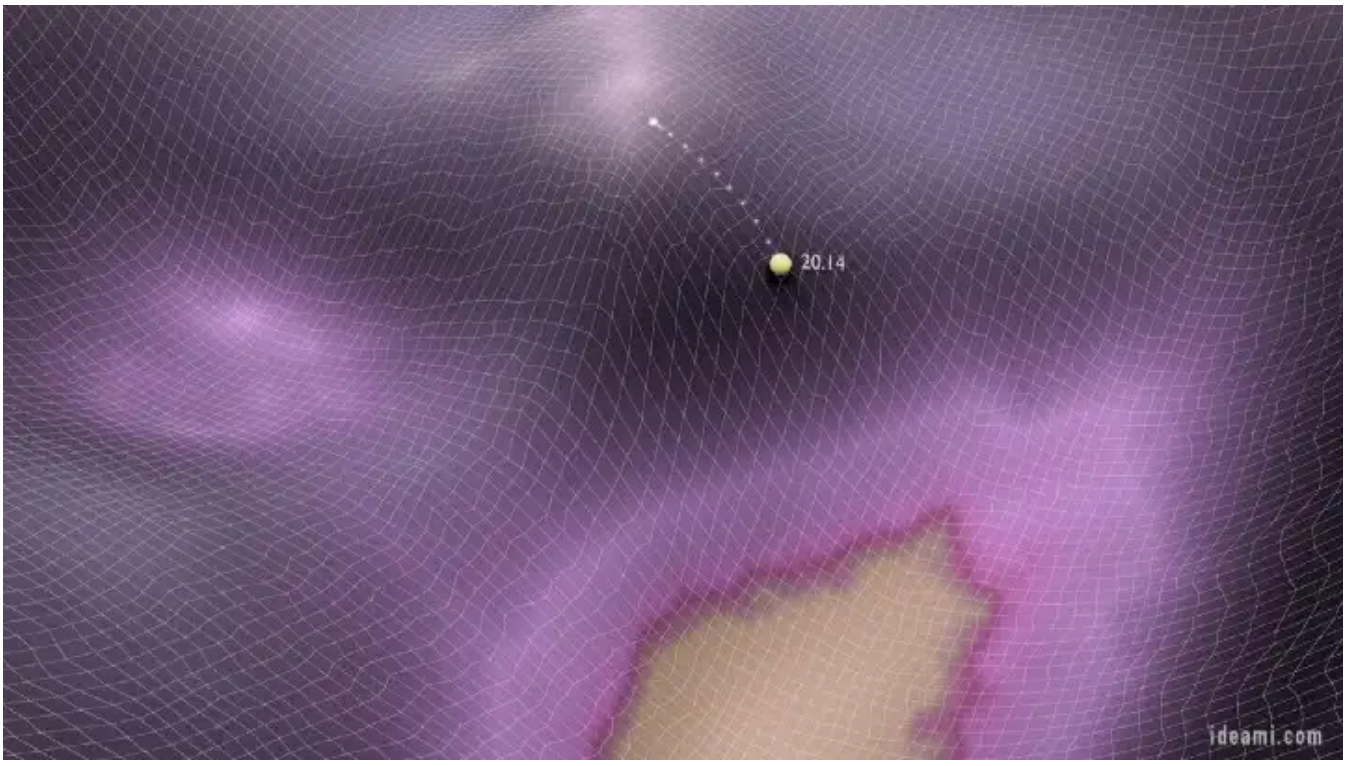


When the mystery function becomes more complex, it is the time to incorporate some of the advanced solutions I mentioned earlier. It is also time to study in more depth the architecture of the entire network and to go deeper into the different hyper-parameters.

Navigating the landscape

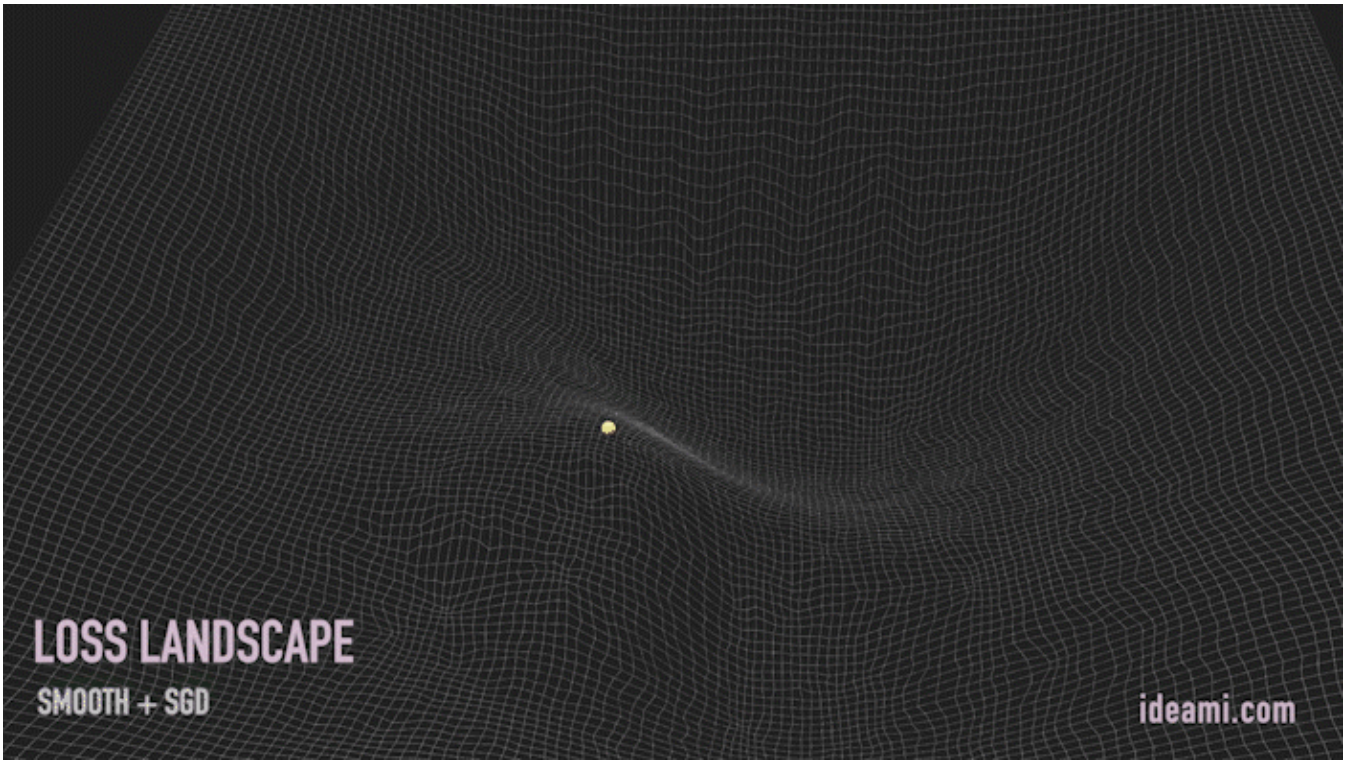
The shape of our loss landscape is very much influenced by the design of the architecture of our networks as well as hyper-parameters like the learning rate, the size of our batches, the optimizer algorithm we use, etc.





For a discussion about those influences, check the paper: Visualizing the Loss Landscape of Neural Nets by Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein.

A very interesting point coming out of recent research is how the **skip connections** model in neural nets can smooth our loss landscape and make it dramatically simpler and more convex, increasing our chances to converge to a good result.

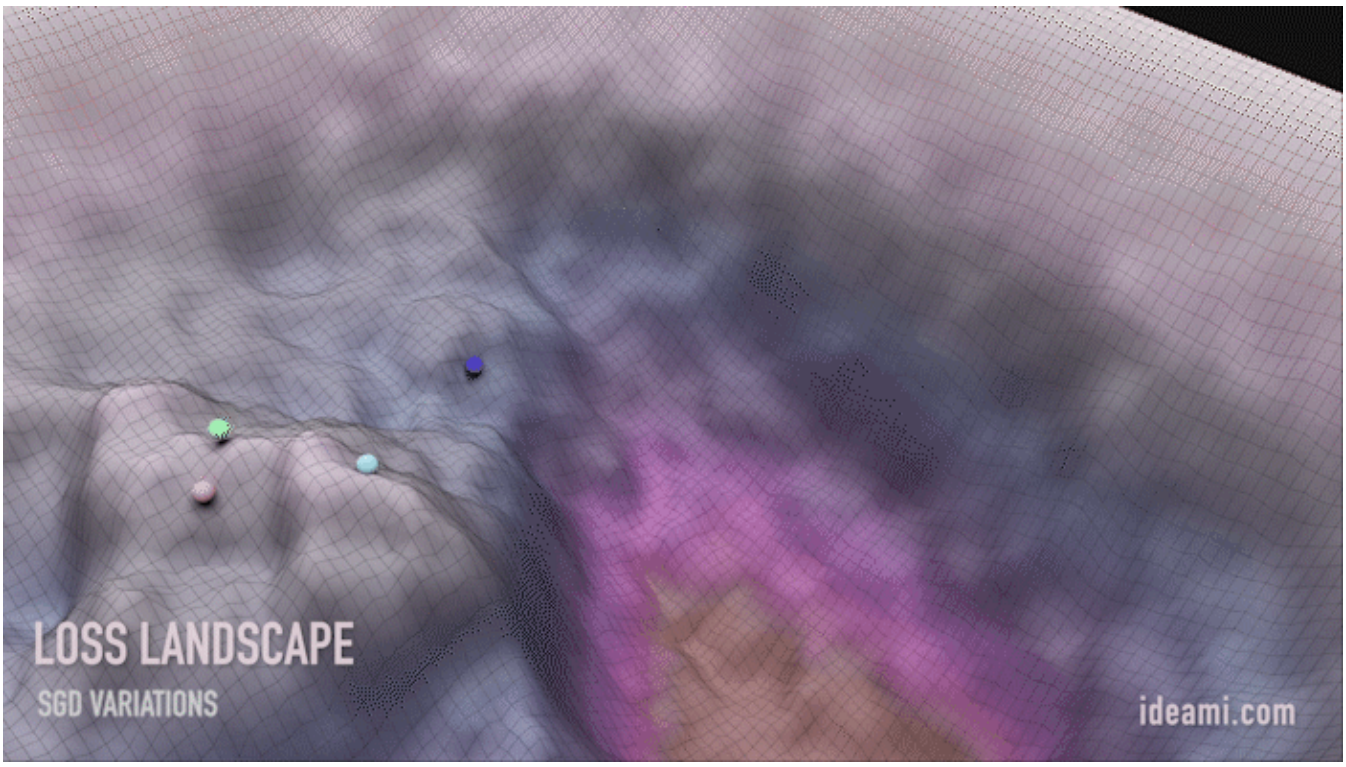


Navigating the Loss Landscape. Values have been modified and scaled up to facilitate visual contrast.

Skip connections have helped a lot to train very deep networks. Basically, skip connections are extra connections that link nodes of separate layers, skipping one or more non-linear layers in between.

As we **experiment** with different architectures and parameters, we are **modifying our loss landscape**, making it more rugged or smooth, increasing or decreasing the number of local optima. And as we optimize the way we initialize the parameters of the network, we are improving our starting position.

Let's keep on exploring new ways to navigate the loss landscapes of the most fascinating challenges in the world.



Navigating the Loss Landscape. Values have been modified and scaled up to facilitate visual contrast.

This article covered the basics and from here, **the sky is the limit**!

Links to the 3 parts of this article:

Part 1 | Part 2 | Part 3

Github Repository with all the code of this project

javismiles/Deep-Learning-predicting-breast-cancer-tumor-malignancy

Predicting Cancer Malignancy with a 2 layer neural network coded from scratch in Python. ...

github.com

Machine Learning Deep Learning Artificial Intelligence Data Science

Towards Data Science