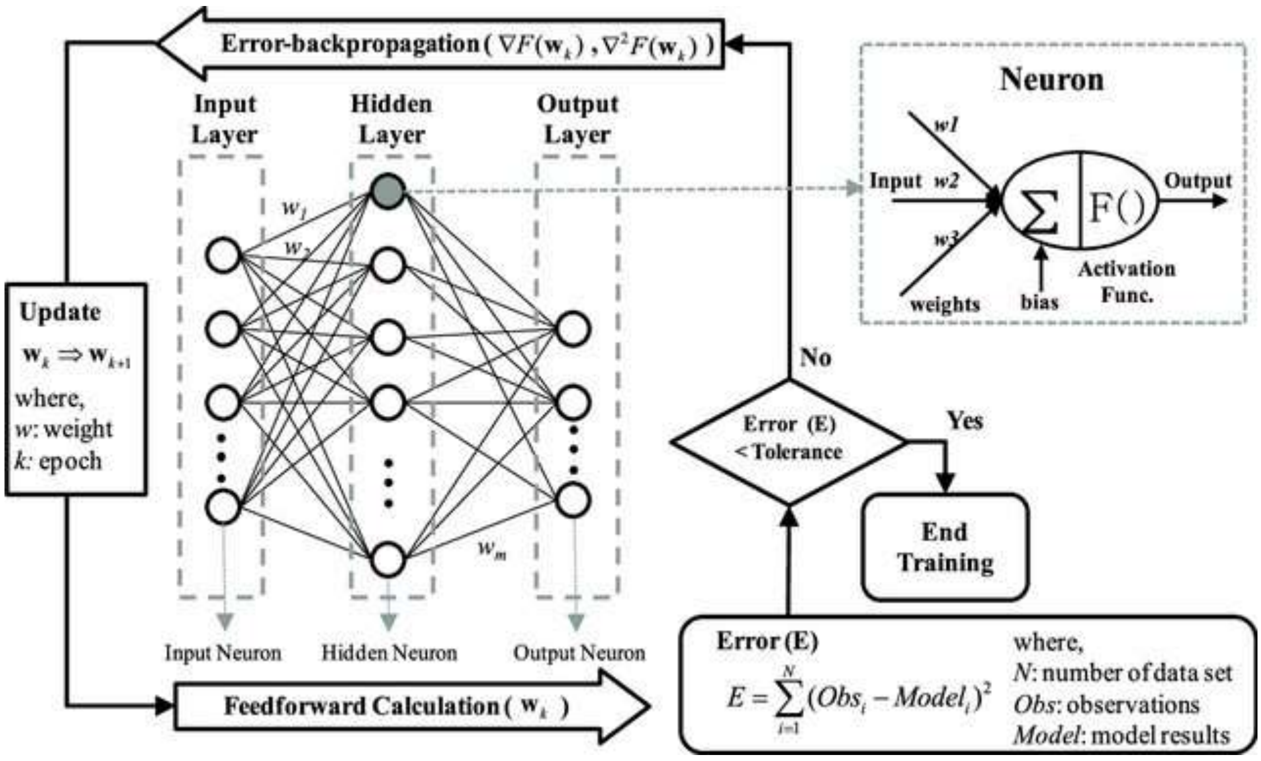
Vanishing/Exploding Gradients in Deep Neural Networks

This lecture explains the problem of *exploding* and *vanishing* gradients while training a deep neural network and the techniques that can be used to cleverly get past this impediment.

Table of Contents

A glimpse of the Backpropagation Algorithm

We know that the backpropagation algorithm is the heart of neural network training. Let's have a glimpse over this algorithm that has proved to be a harbinger in the evolution as well as the revolution of Deep Learning.



- After propagating the input features forward to the output layer through the various hidden layers consisting of different/same activation functions, we come up with a predicted probability of a sample belonging to the positive class *(generally, for classification tasks).*
- Now, the backpropagation algorithm propagates backward from the output layer to the input layer calculating the error gradients on the way.
- Once the computation for gradients of the cost function w.r.t each parameter (weights and biases) in the neural network is done, the algorithm takes a gradient descent step towards the minimum to update the value of each parameter in the network using these gradients.

Understanding the Problems

Vanishing –

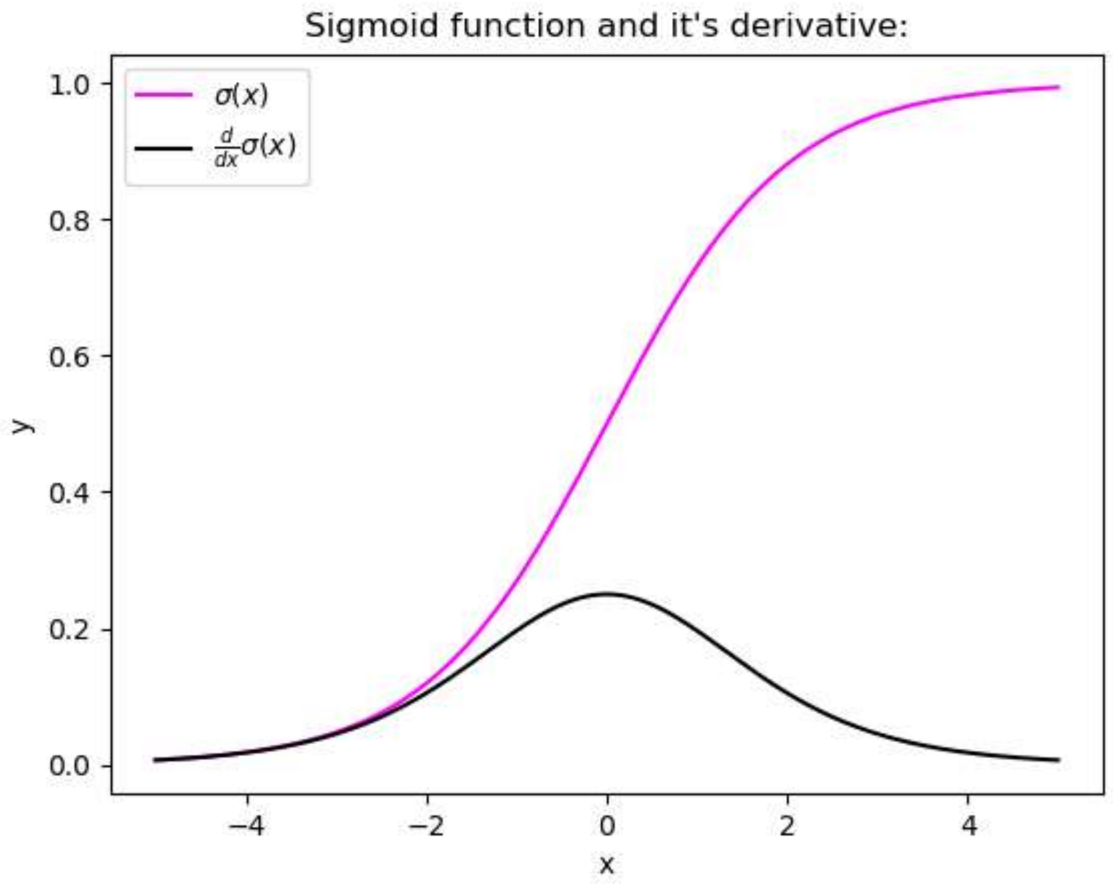
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the ***vanishing gradients*** problem.

Exploding –

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the ***exploding gradients*** problem.

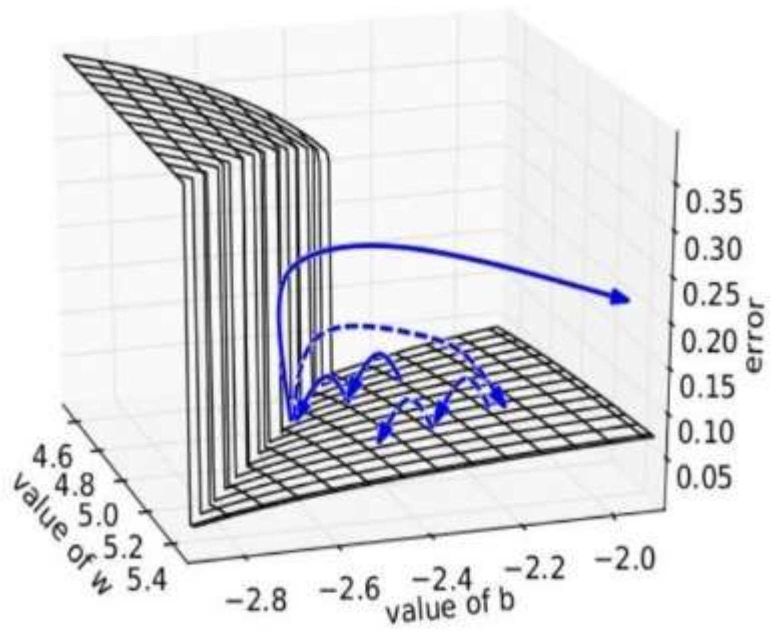
Why do the gradients even vanish/explode?

Certain activation functions, like the logistic function (sigmoid), have a very huge difference between the variance of their inputs and the outputs. In simpler words, they shrink and transform a larger input space into a smaller output space that lies between the range of [0,1].



Observing the above graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.



How to know if our model is suffering from the Exploding/Vanishing gradient problem?

Following are some signs that can indicate that our gradients are exploding/vanishing :

Exploding	**Vanishing**
There is an exponential growth in the model parameters.	The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
The model weights may become NaN during training.	The model weights may become 0 during training.
The model experiences avalanche learning.	The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

Certainly, neither do we want our signal to explode or saturate nor do we want it to die out. The signal needs to flow properly both in the forward direction when making predictions as well as in the backward direction while calculating gradients.

Solutions

Now that we are well aware of the vanishing/exploding gradients problems, it's time to learn some techniques that can be used to fix the respective problems.

1. Proper Weight Initialization

In their paper, researchers Xavier Glorot, Antoine Bordes, and Yoshua Bengio proposed a way to remarkably alleviate this problem.

For the proper flow of the signal, the authors argue that:

1. The variance of outputs of each layer should be equal to the variance of its inputs.
2. The gradients should have equal variance before and after flowing through a layer in the reverse direction.

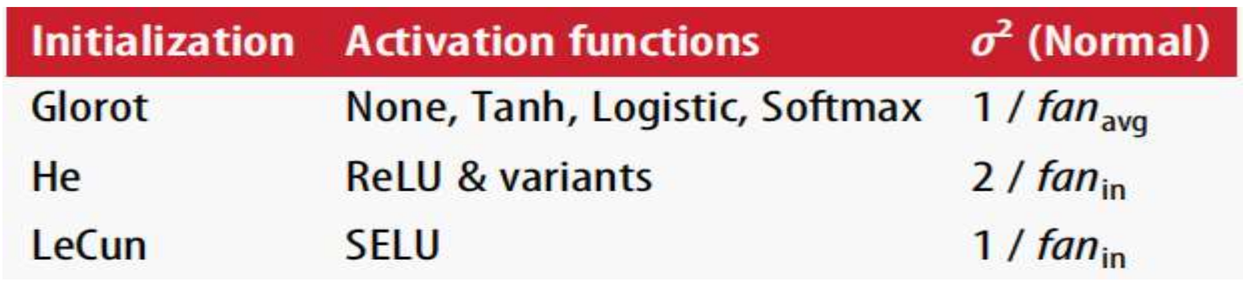
Although it is impossible for both conditions to hold for any layer in the network until and unless the number of inputs to the layer (fan_{in}) is equal to the number of neurons in the layer (fan_{out}), but they proposed a well-proven compromise that works incredibly well in practice: randomly initialize the connection weights for each layer in the network as described in the following equation which is popularly known as **Xavier initialization** (after the author's first name) or **Glorot initialization** (after his last name).

where $fan_{avg} = (fan_{in} + fan_{out}) / 2$

- Normal distribution with mean 0 and variance $\sigma^2 = 1/ fan_{avg}$
- Or a uniform distribution between $-r$ and $+r$, with $r = sqrt(3 / fan_{avg})$

Following are some more very popular weight initialization strategies for different activation functions, they only differ by the scale of variance and by the usage of either fan_{avg} or fan_{in}

for uniform distribution, calculate r as: $\boldsymbol{r = sqrt(3*\sigma^2)}$



Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Using the above initialization strategies can significantly speed up the training and increase the odds of gradient descent converging at a lower generalization error.

Wait, but how do we put these strategies into code ??

Relax! we will not need to hardcode anything, Keras does it for us.

- Keras uses Xavier's initialization strategy with uniform distribution.

- If we wish to use a different strategy than the default one, this can be done using the ***kernel_initializer*** parameter while creating the layer. For example :

```
keras.layer.Dense(25, activation = "relu", kernel_initializer="he_normal")
```

or

```
keras.layer.Dense(25, activation = "relu", kernel_initializer="he_uniform")
```

- If we wish to use use the initialization based on fan_{avg} rather than fan_{in} , we can use the VarianceScaling initializer like this :

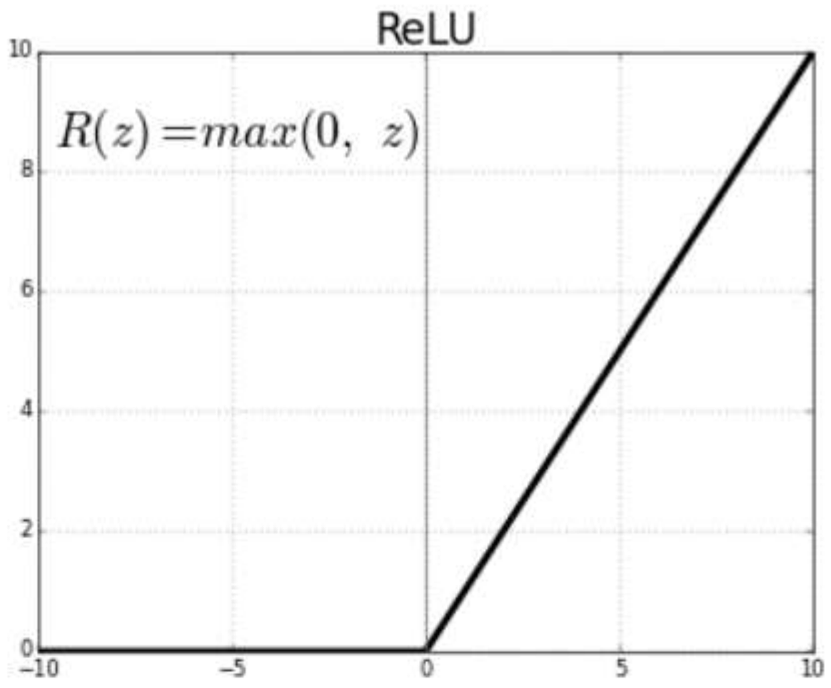
```
he_avg_init = keras.initializers.VarianceScaling(scale=2.,
mode='fan_avg', distribution='uniform')

keras.layers.Dense(20, activation="sigmoid", kernel_initializer=he_avg_init)
```

2. Using Non-saturating Activation Functions

In an earlier section, while studying the nature of sigmoid activation function, we observed that its nature of saturating for larger inputs (negative or positive) came out to be a major reason behind the vanishing of gradients thus making it non-recommendable to use in the hidden layers of the network.

So to tackle the issue regarding the saturation of activation functions like sigmoid and tanh, we must use some other non-saturating functions like ReLu and its alternatives.

ReLU (Rectified Linear Unit)

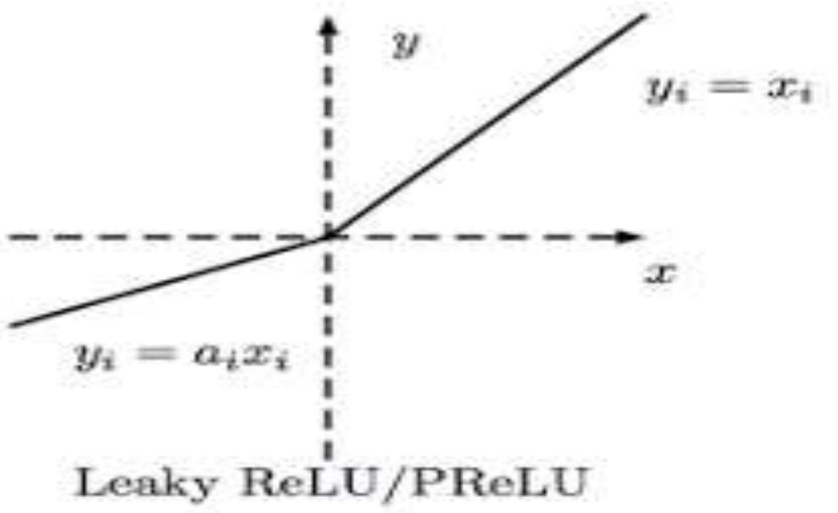


- *Relu(z) = max(0,z)*
- Outputs 0 for any negative input.
- Range: [0, infinity]

Unfortunately, the ReLu function is also not a perfect pick for the intermediate layers of the network "in some cases". It suffers from a problem known as *dying* ReLus wherein some neurons just die out, meaning they keep on throwing 0 as outputs with the advancement in training.

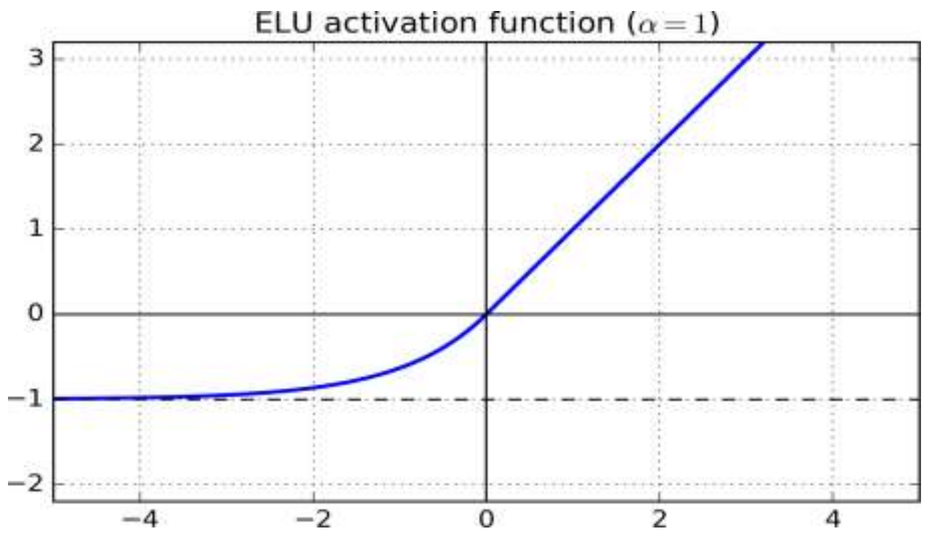
Some popular alternative functions of the ReLU that mitigates the problem of vanishing gradients when used as activation for the intermediate layers of the network are LReLU, PReLU, ELU, SELU :

LReLU (Leaky ReLU)



- *LeakyReLUα(z) = max(αz, z)*
- The amount of "leak" is controlled by the hyperparameter **α**, it is the slope of the function for z < 0.
- The smaller slope for the leak ensures that the neurons powered by leaky Relu never die; although they might venture into a state of coma for a long training phase they always have a chance to eventually wake up.
- ***α*** can also be trained, that is, the model learns the value of α during training. This variant wherein α is now considered a parameter rather than a hyperparameter is called parametric leaky ReLu (**PReLU**).

ELU (Exponential Linear Unit)



- For z < 0, it takes on negative values which allow the unit to have an average output closer to 0 thus alleviating the vanishing gradient problem
- For z < 0, the gradients are non zero. This avoids the dead neurons problem.
- For α = 1, the function is smooth everywhere, this speeds up the gradient descent since it does not bounce right and left around z=0.
- A scaled version of this function (**SELU:** *Scaled ELU*) is also used very often in Deep Learning.

3. Batch Normalization

Using He initialization along with any variant of the ReLU activation function can significantly reduce the chances of vanishing/exploding problems at the beginning. However, it does not guarantee that the problem won't reappear during training.

In 2015, Sergey Ioffe and Christian Szegedy proposed a paper in which they introduced a technique known as *Batch Normalization* to address the problem of vanishing/exploding gradients.

The Following key points explain the intuition behind BN and how it works:

- It consists of adding an operation in the model just before or after the activation function of each hidden layer.
- This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
- In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs.

- To zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation.
- It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization").

model = keras.models.Sequential([keras.layers.Flatten(input_shape=[28, 28]),

keras.layers.BatchNormalization(),

keras.layers.Dense(300, activation="relu"),

keras.layers.BatchNormalization(),

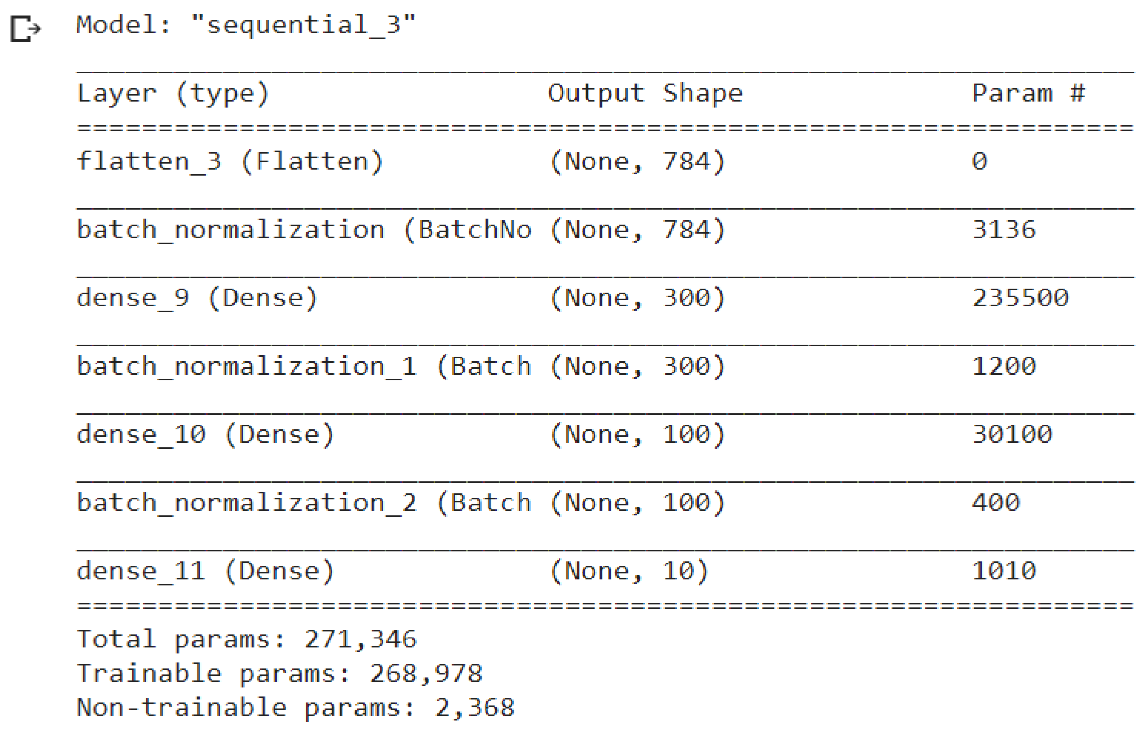
keras.layers.Dense(100, activation="relu"),

keras.layers.BatchNormalization(),

keras.layers.Dense(10, activation="softmax")])

we just added batch normalization after each layer (dataset : FMNIST)

model.summary()



```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_3 (Flatten)          (None, 784)               0         
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136      
_________________________________________________________________
dense_9 (Dense)              (None, 300)               235500    
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200      
_________________________________________________________________
dense_10 (Dense)             (None, 100)               30100     
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400       
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1010      
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

4. Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called Gradient Clipping.

- This optimizer will clip every component of the gradient vector to a value between – 1.0 and 1.0.

- Meaning, all the partial derivatives of the loss w.r.t each trainable parameter will be clipped between –1.0 and 1.0

```
optimizer = keras.optimizers.SGD(clipvalue = 1.0)
```

- The threshold is a hyperparameter we can tune.
- The orientation of the gradient vector may change due to this: for eg, let the original gradient vector be [0.9, 100.0] pointing mostly in the direction of the second axis, but once we clip it by some value, we get [0.9, 1.0] which now points somewhere around the diagonal between the two axes.
- To ensure that the orientation remains intact even after clipping, we should clip by norm rather than by value.

```
optimizer = keras.optimizers.SGD(clipnorm = 1.0)
```

- Now the whole gradient will be clipped if the threshold we picked is less than its ℓ2 norm. For eg: if *clipnorm=1* then the vector [0.9, 100.0] will be clipped to [0.00899, 0.999995] , thus preserving its orientation.