**Winter Project**

**Implementation of DCGAN**

Initial Research

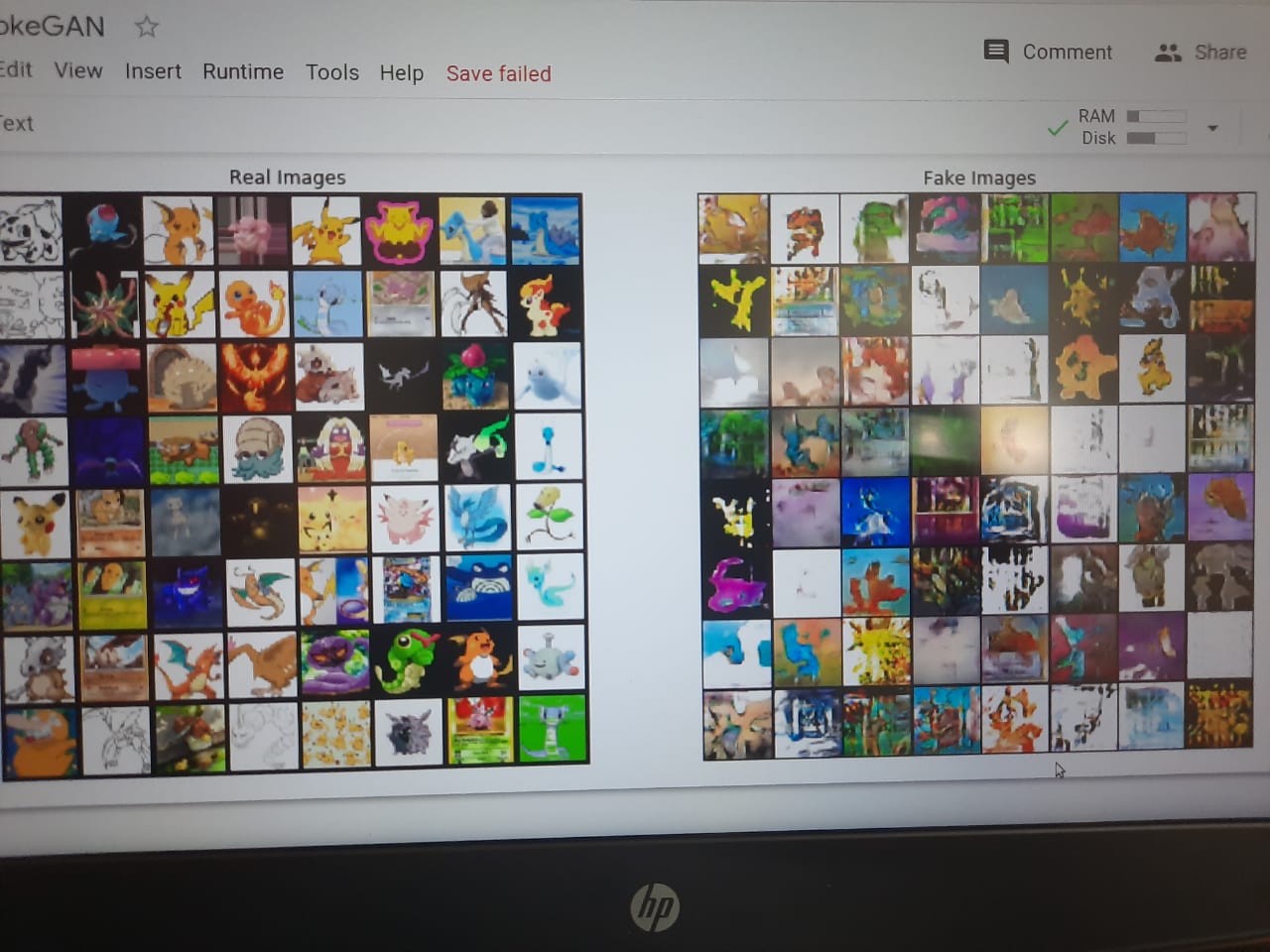
A simple google search on how to generate new Pokémon from an existing dataset yielded multiple references to Generative Adversarial Networks (GANs). I started reading the original GAN paper [1] published by Ian Goodfellow to gain an insight into how GANs actually work.

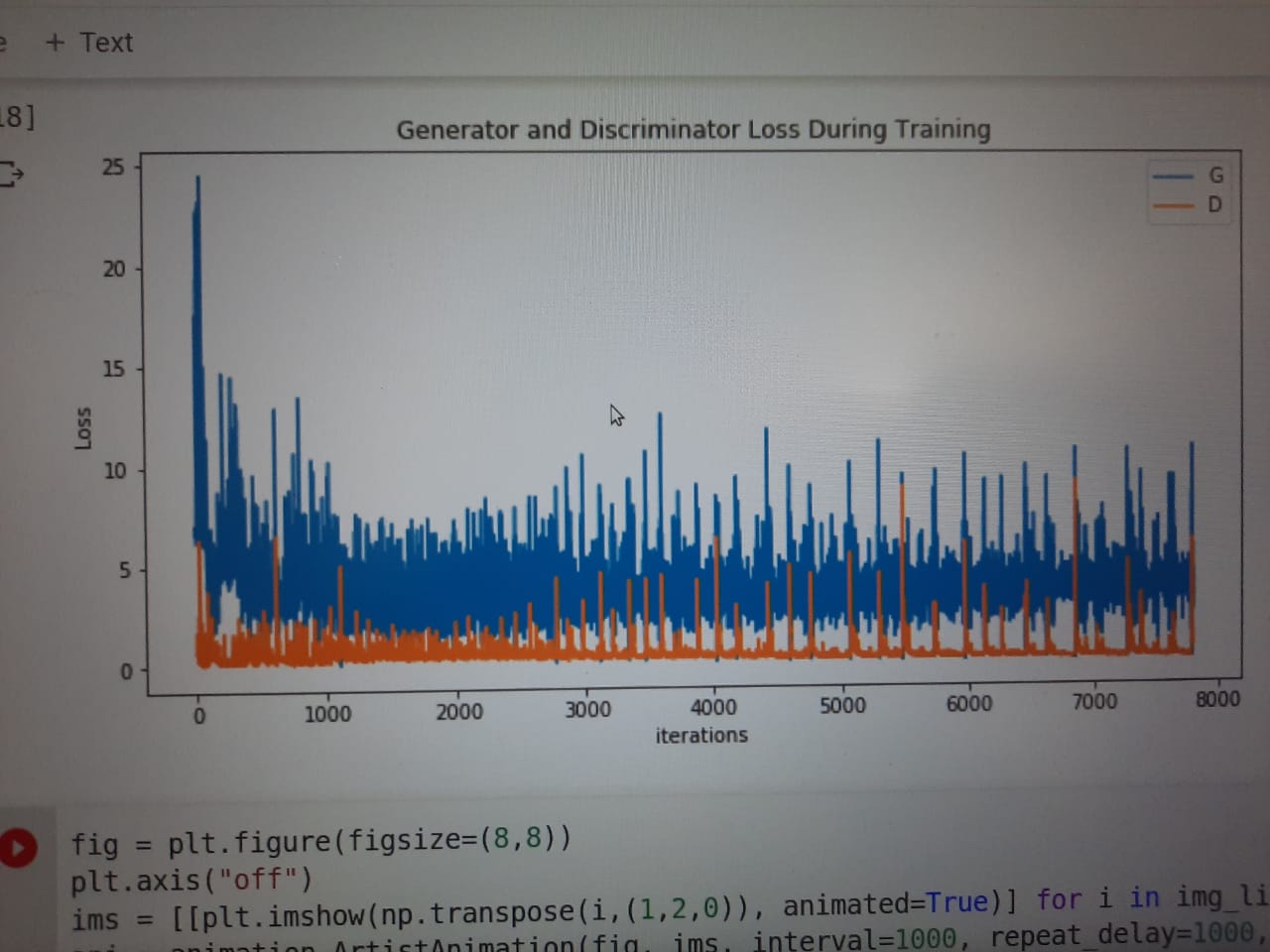
To further grasp the concept of generating new Pokémon I watched a few videos on YouTube [2] that described the entire process in detail. After that, I read the Deep Convolutional GAN paper [3] which is what I finally implemented on this project.

To implement the project on the PyTorch framework I undertook the PyTorch tutorial available on the official PyTorch website and read about the different loss functions, modules and functions the framework provided.

Architecture

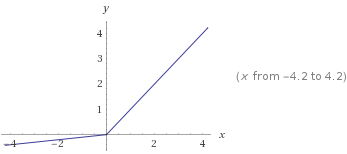
For my first attempt I used the DCGAN architecture provided on the tutorial page of the PyTorch page and after training for 100 epochs I got the following results:





In this architecture, the generator has four transpose convolutional layers, each followed by batch normalization and the ReLU activation function. The fully connected layer has a transpose convolutional layer followed by a tanh function.

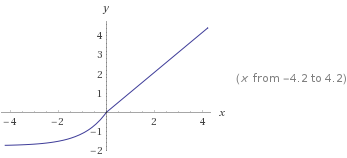
The discriminator’s first layer has a convolutional layer and a LeakyReLU activation function. There are three hidden layers each with a convolutional layer, batch normalization and a LeakyReLU activation function. The last layer has a convolutional layer a sigmoid function. Using the LeakyReLU instead of ReLU in the discriminator solves the problem of dying ReLUs. ReLUs can get trapped in a dead state. That is, the weights’ change is so high and the resulting *z* in the next iteration so small that the activation function is stuck at the left side of zero. The affected cell cannot contribute to the learning of the network anymore, and its gradient stays zero. If this happens to many cells in the network, the power of the trained network stays below its theoretical capabilities. The LeakyReLU on the other hand doesn’t have this problem.



As seen in the above graph which represents LeakyReLU, there is a slope on the left side. This slope is usually tiny (e.g., 0.01), but it is there, so there is always learning happening and the cell cannot die. Leaky ReLUs keep the simplicity of computation since there are only two different and constant gradients possible (1 and 0.01).

To improve upon these results and get clearer images, I tried changing the learning rate, the activation functions used in the generator and discriminator, and also tried a few optimization techniques I read in the paper “Improved techniques for training GANs” [4].

1. Trying out SELU activation function in the generator instead of ReLU.



The right half of the graph represents ReLUs but in the left half, the slope appears to near zero. Still, the problem of vanishing gradients doesn’t reappear in this case. This is because instead of trying to control the gradient, SELUs take the normalization approach. SELUs implement internal normalization.  The main idea is that each layer preserves the mean and variance from the previous layer.

The gradients can be used to adjust the variance. The activation function needs a region with a gradient larger than one to increase it. The formula behind SELU is :

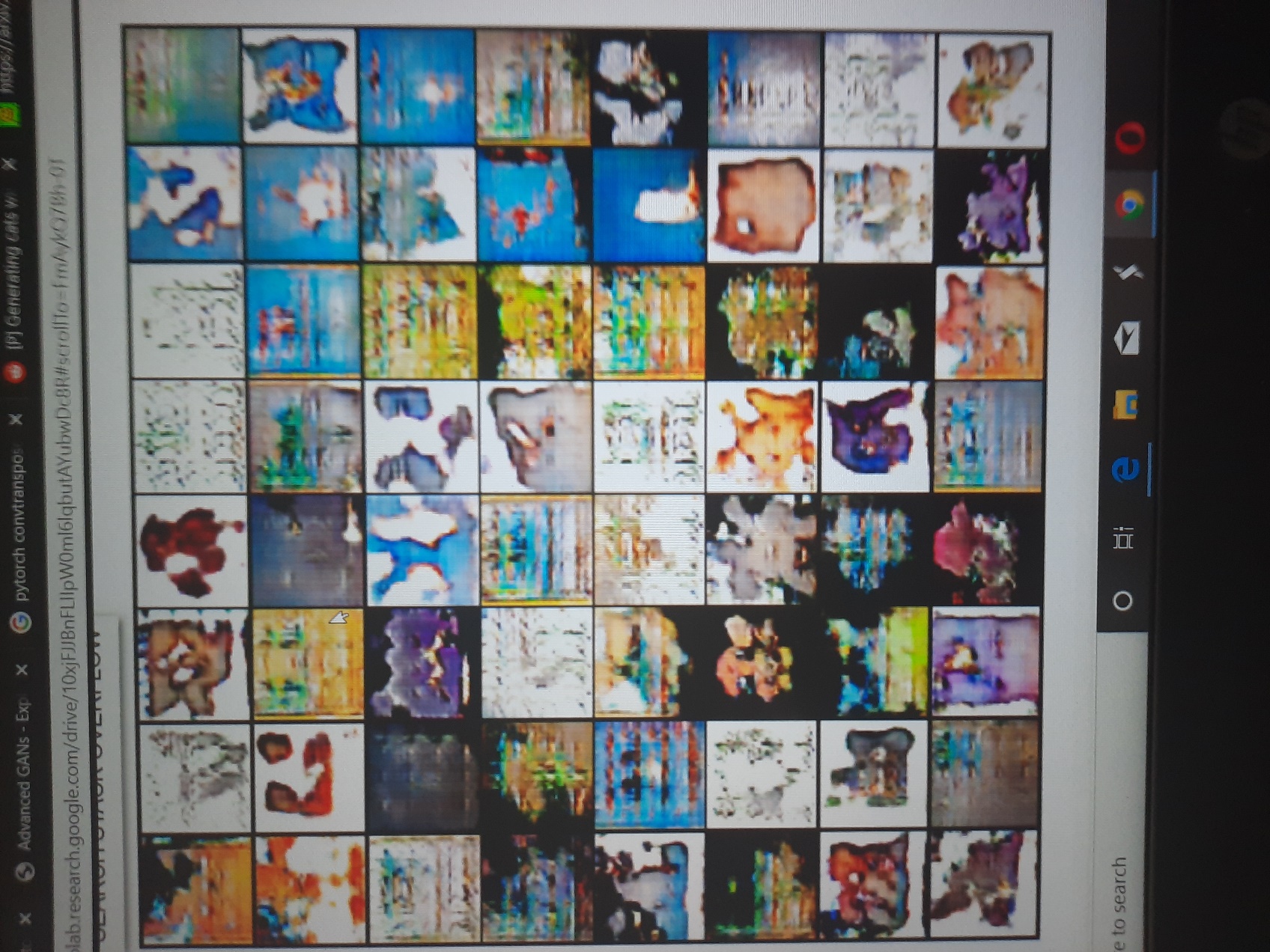


Although it looks like a ReLU for values larger than zero, there is an extra parameter involved: λ. This parameter is the reason for the **S**(scaled) in SELU. When it is larger than one, the gradient is also larger than one, and the activation function can increase the variance. A gradient very close to zero can be used to decrease the variance. The cause for vanishing gradients in other activation functions is a necessary characteristic for internal normalization.  SELUs have this excellent quality of self-normalization and the problem of vanishing gradients is solved indefinitely. The advantages of using SELUs in place of ReLUs are:

1. Similar to ReLUs, SELUs enable deep neural networks since there is no problem with vanishing gradients.
2. In contrast to ReLUs, SELUs cannot die.
3. SELUs on their own learn faster and betterthan other activation functions, even if they are combined with batch normalization.

Although I found a lot of articles claiming that SELUs produce much better results than ReLUs, the results I got weren’t so encouraging. There could still be a variety of reasons why I didn’t get better results than when I used ReLUs for the same dataset and for the same number of epochs. I tried training for a larger number of epochs but the colab notebook kept getting disconnected and thus I couldn’t get the results.

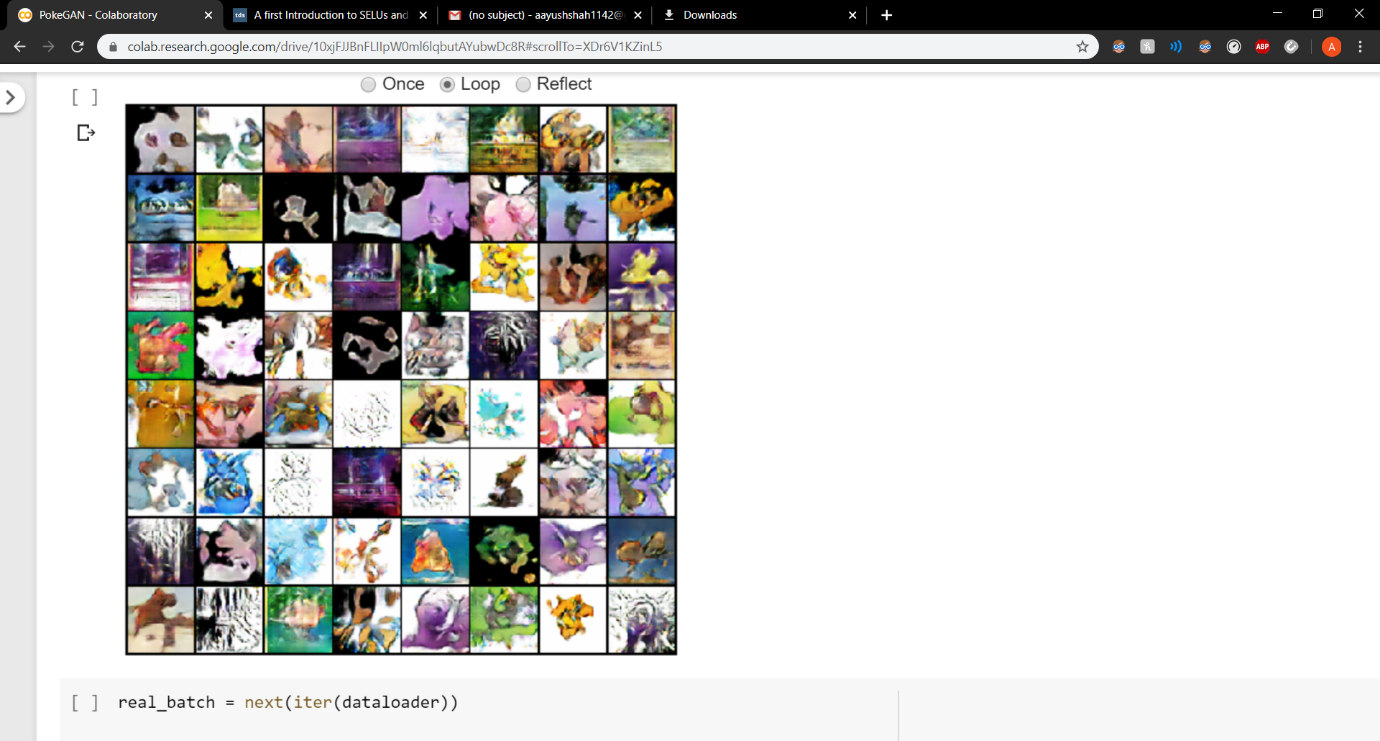
Here are the results I got after 100 epochs:

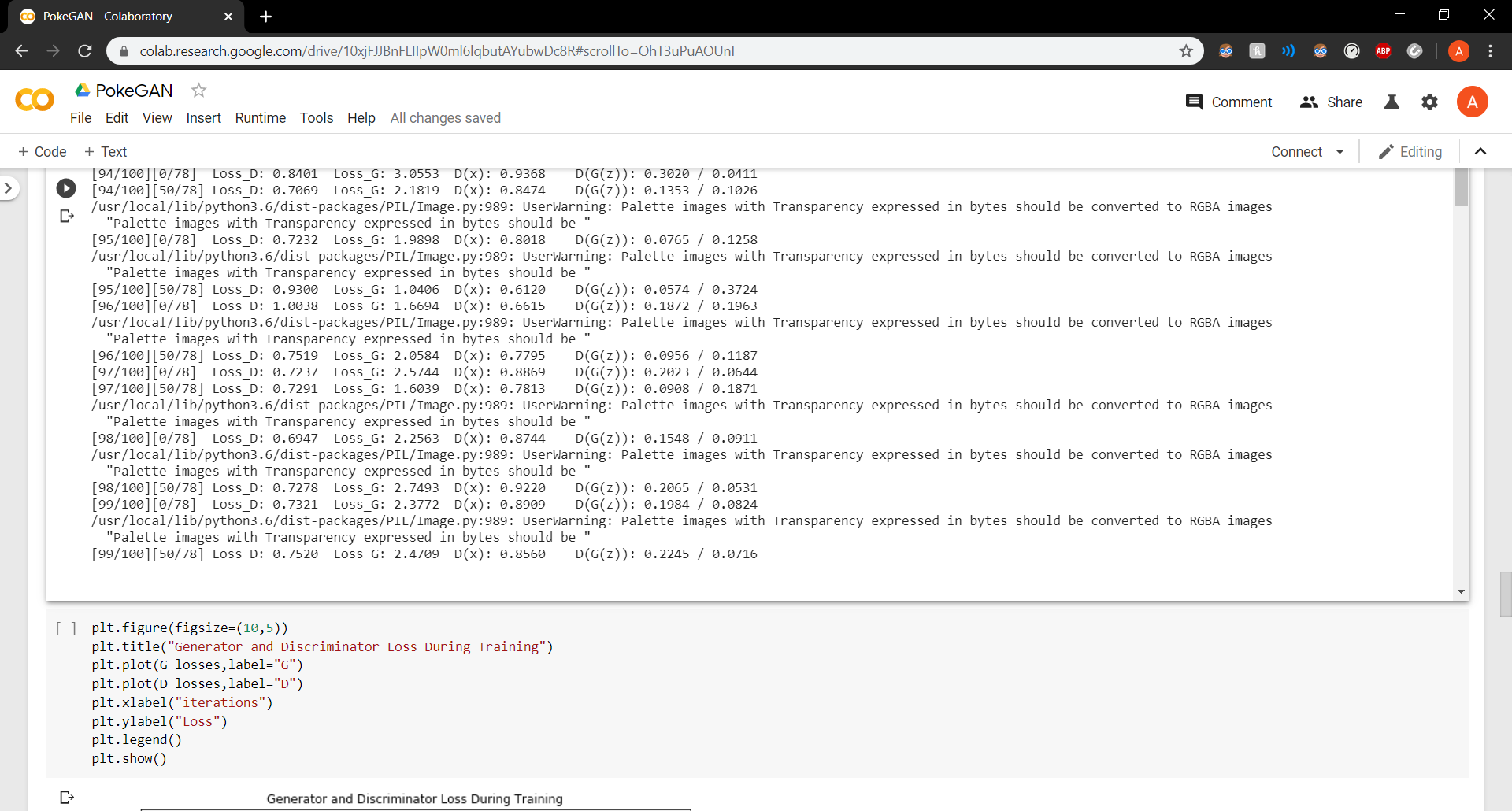


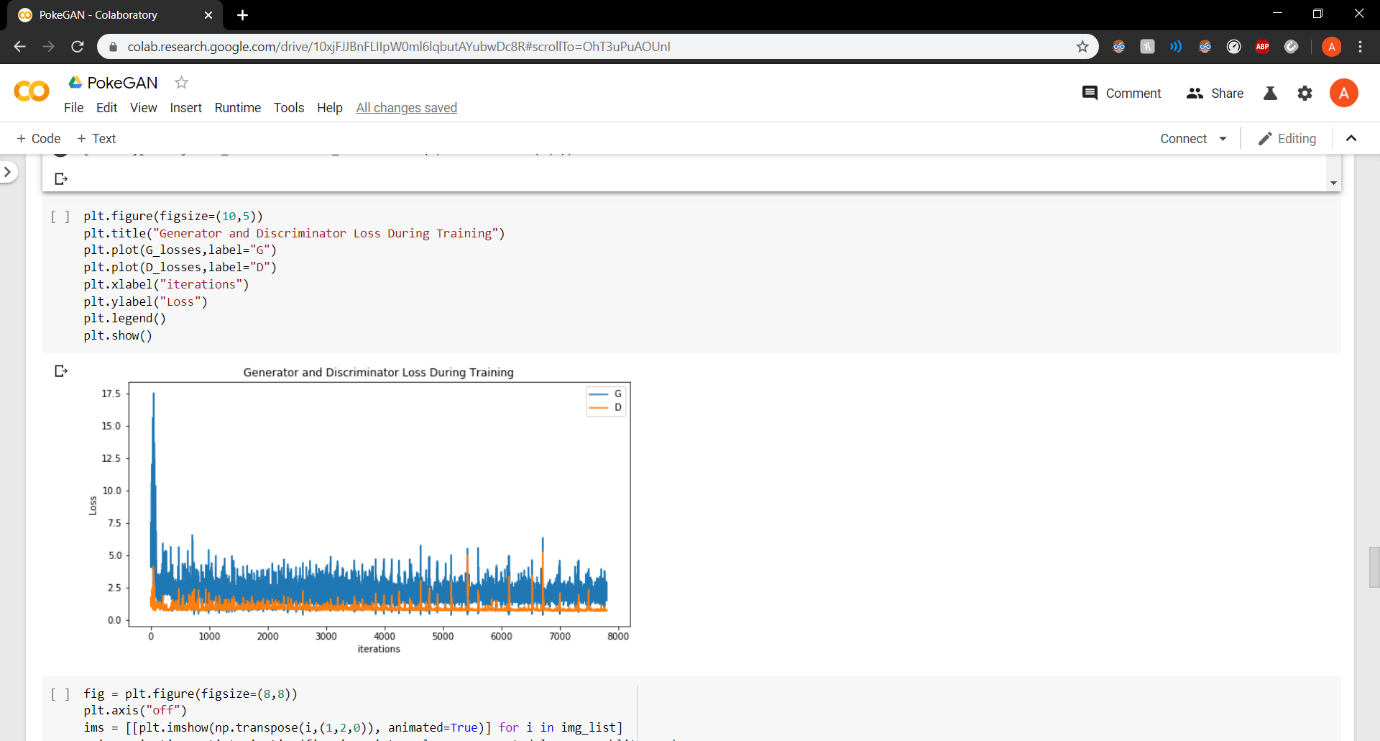
Although the images I obtained weren’t clearer, one advantage I did observe while using SELUs was that training time per epoch reduced a little. But overall, I concluded that unless I have a system capable of training a GAN for at least 1500 epochs, using SELU instead of ReLU doesn’t really have any real advantage.

2. I also tried tuning the learning rate. I started with a large learning rate and reduced it slowly as the model started to converge. However, when I combined this technique with the SELU activation function the model collapsed completely. The discriminator loss reduced to 0.0001 within 10 epochs and the images produced were just random noise.

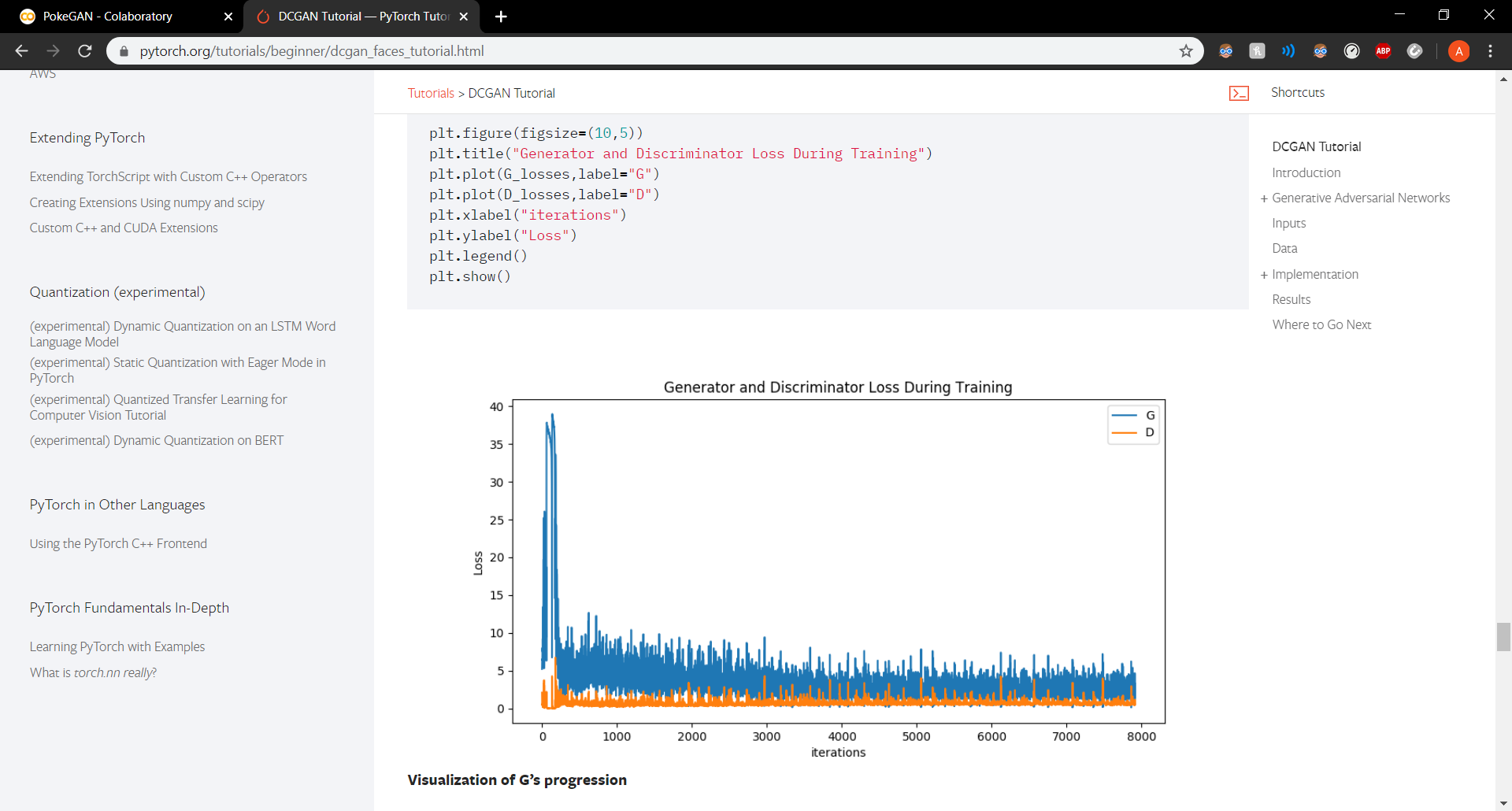
3. The most promising method I came across was **one sided label smoothing** [4]. In this method, the 0 and 1 values for the classifier are “smoothed” and replaced with 0.9 and 0.1. When I ran the model after applying this method combined with the ReLU activation in the generator and LeakyReLU in the discriminator along with an increased learning rate, the results I got after 100 epochs weren’t that clear but the generator and discriminator loss was varying very well. I couldn’t train the model for more than 100 epochs so I can’t post clear results but here are the ones I got after 100 epochs along with the loss:







In the DCGAN tutorial provided on the pytorch website, the graph of the generator and discriminator loss was:



Both the graphs are similar and this led me to believe that training for a higher number of epochs could lead to much better results.

References

[1] <http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>

[2] <https://www.youtube.com/watch?v=yz6dNf7X7SA>

[3] <https://arxiv.org/pdf/1511.06434.pdf>

[4] <http://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.pdf>