Scalable Scientific Computing
CMDA 4634

DCGANs Optimization in PyTorch
Fall 2022

Sanil Jain

December 12, 2022

Contents

Part I

Final projects

Lecture 1

DCGANs Optimization in PyTorch

Sanil Jain

Contents

1.1 DCGANs

Deep Convolutional Generative Adversarial Neural Networks (DCGANs) is an interesting development in the Artificial Intelligence and Deep Learning space when it comes to image generation and classification. To understand this type of model, it is first important to understand neural networks and then to understand what a GAN is.

1.1.1 Neural Networks

Neural Networks at the most basic level are the replication of the human brain into a model. We do this by treating each neuron in the brain as a node in a graph and then giving each node an activation function to mimic how neurons are activated in the brain via electricity. On top of this, each node is assigned a function in order to make it useful for a certain process. Process-wise, the model has been able to analyze and generate any kind of data available with a focus on audio, image, and text. When looking at the structure of a neural network there are three components an input layer, hidden layers, and the output layer. The most important part that differentiates these networks from each other lies in the hidden layers which can vary depending on the purpose of the network. Each hidden layer as mentioned holds an activation function which may also have a weight attached to it as a layer may have many nodes per layer that range depending on the goal and data set size. Figure 1.1 highlights as there are about 5 nodes in the input and hidden layers.

These neural networks after creation are "trained" on data, which is fed through the input layer, then through the many hidden layers with it ending at the output layer which is known as a feed-forward. Depending on the accuracy of the predictions made by the output layer with respect to the training data, the model changes its weights in the hidden layers. The function used to calculate this is called the loss function and the goal of the model is usually to maximize the accuracy and minimize the loss function. This loss function is minimized through a process called gradient descent and, in some cases, a more specialized version of it depending on the data type. The process by which we go backward in the network to optimize the weight per iteration or epoch is called backpropagation.

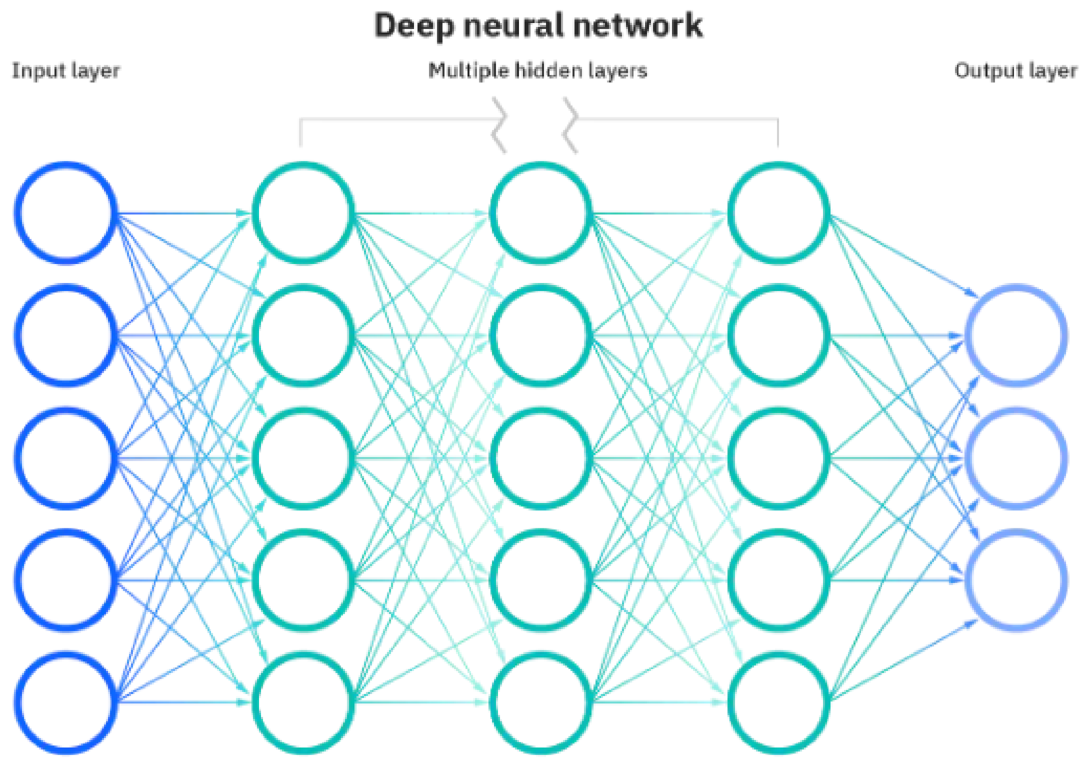

Figure 1.1: Deep Nueral Network Diagram

1.1.2 GANs

Generative Adversarial Neural Networks (GANs) are a type of model that relies on a conflict between two networks. The conflict is engineered in order to be able to generate and classify data. The algorithm relies on two parts, the Generator ($G(x)$) and the Discriminator ($D(x)$). $G(x)$'s goal is to generate fake data from the training data set in order to trick $D(x)$. $D(x)$'s goal is to classify $G(x)$'s data into two categories, fake and real. The relationship between the two algorithms is important as eventually the generation and classification will converge. This convergence occurs when $D(x)$ will reach a 50% chance of predicting $G(x)$'s data. This means that $G(x)$ is good enough at generating data that $D(x)$ cannot differentiate between the training and generated data meaning it must resort to guessing. Training-wise, we have a special loss function that is used to enforce the relationship between $D(x)$ and $G(x)$, and when training we have three gradients, we minimize which in turn increases the amount of time it takes to train per iteration. The loss function that shows the relationship is below where we want to find the place where $p_{data} = p_z$ but in practice we usually don't get that close. This is still an active area of research and depends heavily on the structure of both $G(x)$ and $D(x)$. The loss function is shown below.

$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_{z}(z)}[1 - \log D(G(z))]$

Looking outwards, this algorithm class is quite flexible in that it can generate most data. ChatGPT is a great example of the use of this model as it is able to generate answers from questions humans ask in a variety of domains. In the case of ChatGPT, the data that is given is chat text data and the output is what $G(x)$ thinks a human would say.

1.1.3 DCGANs

Deep Convolutional GANs are a further extension of the GANs algorithm by specifying the hidden layers that are used by each network. We set the $G(x)$ hidden layers to convolutions which makes them Convolutional Neural Networks (CNN). CNN's are a special class of Neural networks that is specialized for image data and uses a convolutional function in order to do this. Figure 1.2 shows this with the numbers below specifying the expected matrix size and image size.

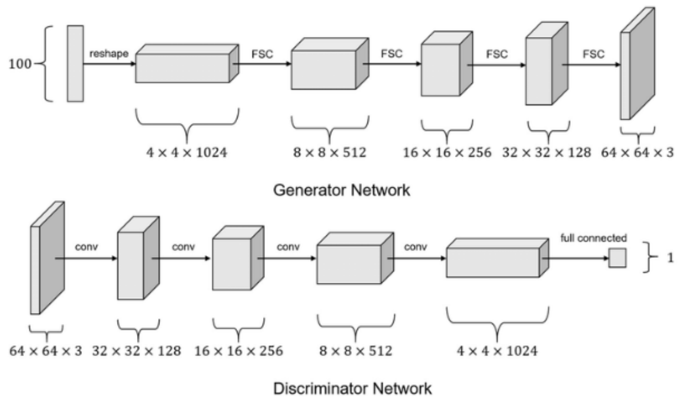

Figure 1.2: DCGANs Architecture

1.2 PyTorch Essentials and Implementation

PyTorch is a deep learning framework that is used widely in research and industry in order to develop and deploy deep learning algorithms. PyTorch‘s main features are its dynamic graph, extendibility, and special just-in-time (JIT) compiler. PyTorch is unique among frameworks as it has a C++ back and front-end API and uses Python as its user interface. Python allows for the development pace to be a lot faster than in C++ but also preserves the speed by allowing more experienced developers to be able to write code directly in C++ if optimizing for performance. This C++ compatibility also allows for easier integration with CUDA and the ability to extend classes in PyTorch in order to write special kernels and program directly in CUDA. Its integration with CUDA is further shown through its use of cuDNN, a special CUDA library written by NVIDIA to speed up Neural Networks at the GPU level. It also has the ability to make a C++ executable from Python code that can be directly converted to C++ as well which PyTorch calls TorchScript. This is done most of the time to avoid python dependency problems and provides some speed up.

1.2.1 Important Data Structures

Tensors are PyTorch's version of NumPy arrays and/or matrices that were created in order to leverage the GPU. This can be done by instantiating tensors directly on a GPU from Python code. These tensors are what PyTorch uses in order to do any matrix calculation and what is used in order to make gradient calculations. The cost of these gradient calculations can be quite significant so PyTorch made a special data structure to speed this up called Autograd. Autograd is an engine that automatically computes these gradients by creating a computational graph. These computational graphs are directed acyclic graphs (DAGs) that have leaf nodes as the input and the root as the output. This graph computes this gradient by using the chain rule with the nodes as portions of the chain rule. This graph I'm assuming allows for some parallelism as the input is a matrix in which each leaf node is only doing a part of the work.

Another important data structure that makes data loading quite a bit fast is the Data Loader object. This object loads the data that is given to the program in parallel in order to accommodate large data sets that we cannot fit in memory most of the time and takes care of all tasks associated with the process. When doing the data loading, the object is loading the data onto the GPU from the CPU.

1.2.2 Implementing the Generator and Discriminator

When implementing $G(x)$ and $D(x)$ we must prior to it prep the data by passing it through the DataLoader object. This object transforms the data into a tensor of File Objects that the model will read. After this, we have to set the initial weights for both networks. These initial weights have been specified in the DCGANs paper which is a normal distribution across all weights with the mean set to zero and the standard deviation at 0.02.

The networks themselves are quite easy to implement by themselves as they are the opposite of each other. $G(x)$ is implemented by first creating a class that is passed in a nn.Module object from PyTorch. This object is an interface in PyTorch that sets the parameters that a model must have which are the hidden layers and a forward function in order to specify what a feed-forward would look like for this model. The hidden layers that we specify for $G(x)$ are Convolutional Transpose 2D's which make the matrix go from a 4×4 to 8×8. This process is continued four more times until we get to a size of 64×64. After this, we pass the model through a Tanh function which helps us generate an image output in the form of a tensor. $D(x)$ follows the same principle in reverse as we start with a tensor of 64×64 size and transform it using convolutions to a 4×4 matrix. Afterward, we pass it through a Sigmoid function which helps us make a binary decision. Throughout this process, we keep the tensors in powers of two as these tensors are optimal for matrix multiplication on Tensor Cores.

Lastly, we must make sure to create the loss function, fixed noise, and optimizers. This loss function object is built into PyTorch so just have to call the nn.BCELoss() function. We create the fixed noise pretty easily using a tensor method. After this, we create the real and fake labels as one and zero and then we instantiate the optimizer objects. These optimizers are called Adam optimizers and are a replacement for stochastic gradient descent due to them being more optimal for image-based data when related to deep learning.

1.2.3 Implementing Training

To begin the training process when we are training two networks together, we must specify the number of epochs we want the model to do. After specifying, we pass in the data loader which then loads a tensor into $D(x)$. At the same time as this, we zero the autograd object's tensor and then we move the data to the device to prep it for a forward pass. We then label the data and do one iteration with $D(x)$ and recompute its own gradient and then backpropagates. After this process, we add some noise to the device and do the same steps as above on $G(x)$. After backpropagation occurs on $G(x)$, we evaluate its output against $D(x)$ which then computes the gradient of the loss function which specifies the relationship between the two models. We do this process for all training data in the data loader and collect the loss amount of $G(x)$, $D(x)$, and $D(G(x))$ in order to evaluate the training and loss of the functions per epoch. As we do this training process, after the end of each epoch we collect the mean of the images that $G(x)$ creates to check it for human inspection. Each epoch varies in time but on a GTX 1060, it took about 2 minutes and 43.4 seconds to run one epoch. The code in the repository is set to one epoch currently but 200 epochs seem to be the point where images that are somewhat "real" can be seen. More epochs are definitely needed to create non-blurry images. Figure 1.3 is an example of what it should look like.

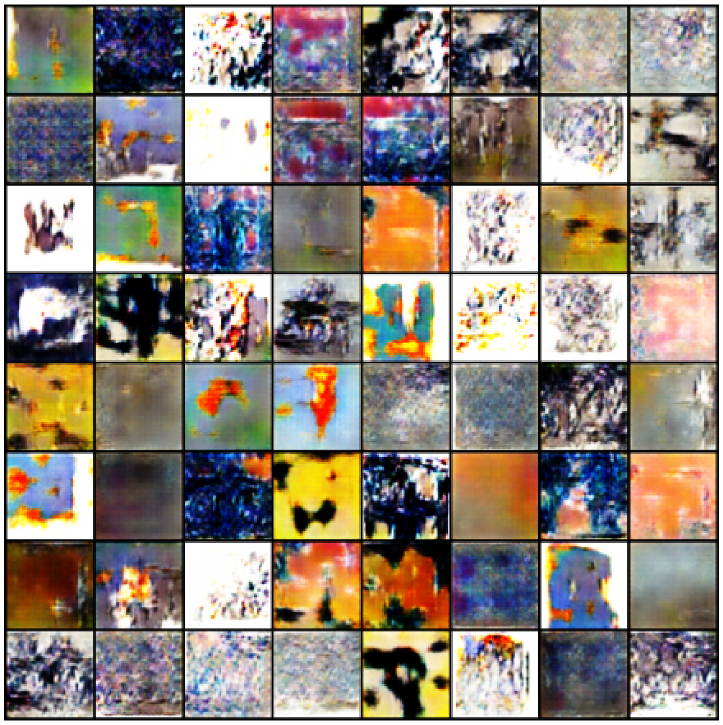

Figure 1.3: Output after 200 epochs

1.3 Optimizations

In order to further optimize a DCGAN for performance, we must do another round of parallelisms outside of just using tensors, autograd, and the data loader. These tools range from using a different compiler entirely to speed up python code and using some parallelism at the data level to perform other speed-ups. In the discussion below, the focus is on optimizing that are directly related to GPUs as there are a variety of tools that are available that can speed up algorithms. These techniques are focused more on the Neural Networks themselves and are out of the scope of the project.

1.3.1 Distributed Data-Parallel

Distributed Data-Parallel (DDP) is a module in PyTorch specifically for training large models that require multiple GPUs or devices. The module does this by splitting the training data into n slices that are then loaded in parallel into n devices. This loading is done in order to speed up the training process per epoch when it comes to the forward feed portion of the epoch. The module at the backpropagation stage finds the summation of the weights found at each device and then updates the new network weights at each device. An inferior version of this Data Parallel does the tasks above at the block level instead of the device level. Both modules are wrapper objects that wrap around the neural networks and do the tasks above automatically in the background. If a more manual approach is required, it can be specified what work each device is going to do using DDP. This has lots of intricacies and I wasn't able to implement it due to the prerequisites for utilizing DDP being eight devices and the process is not that well documented in general. An interesting thing to note is that by just replacing the Data Parallel with DDP, the epochs become slightly more efficient.

1.3.2 nvFuser and JIT compiler

Another major optimization tool used is the nvFuser. nvFuser is a special compiler specifically for Volta generation or later and is a CUDA accelerator that fuses different kernels together in order to provide some speed up. nvFuser is a just-in-time (JIT) compiler that allows for integration into PyTorch in prior generations by the use of a decorator. This decorator, @jit.script_method, essentially pre-compiles the Python code into TorchScript. The decorator's only caveat is that it can be only used on functions that use PyTorch objects.

1.4 Final Thoughts

GANs have been a model structure that has interested me for the longest time because of their use in graphics and this project combined that interest with GPUs quite well. Originally, this project was supposed to include benchmarking data but after writing the code in PyTorch, I realized that profiling my program is quite difficult without it running on more than one GPU as the optimization techniques that I wanted to use would require hardware that is not available to me. In addition to that, I would love to spend more time with DDP and write a PyTorch program completely in C++ in the future.

1.5 Resources

Github Repository
Slides Link
PyTorch Documentation

1.6 Bibliography

Radford, A., Metz, L. & Chintala, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. (arXiv,2015), https://arxiv.org/abs/1511.06434

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. & Bengio, Y. Generative Adversarial Networks. (arXiv,2014), https://arxiv.org/abs/1406.2661

By: IBM Cloud Education. (n.d.). What are neural networks? IBM. Retrieved December 11, 2022, from https://www.ibm.com/cloud/learn/neural-networks