

VAN GAN

Training Deep Convolutional Generative Adversarial Neural Networks
to Classify and Generate Van Gogh's Paintings

Michael Ge

michaelge@college.harvard.edu

Anni Wang

anni.wang@g.harvard.edu

December 6, 2017

Abstract

In this report, we discuss a deep-learning approach to image classification and generation based on sparse image corpora. We present a variety of black-box neural network approaches to training deep convolutional generative adversarial neural networks (DCGANs). We report the highlight of our generated images and classification results, compare these results with traditional machine learning models, and discuss the limitations of the model.

1 Introduction

Neural networks have become a standard method of solving non-convex regression and classification problems due to the increase in computing power and ease of implementation with regularly updated programming libraries. Their combination with zero-sum game strategies found in economics and artificial intelligence has led to the creation of generative adversarial neural networks (GANs), a mini-max model consists of a system of two neural networks: a discriminator takes the input image and classifies the images, and a generator forges increasingly better data to “fool” and as a result improve the accuracy of the discriminator.

A limitation of neural network-based models, however, is that they typically function under a large corpus of data and require large-scale computation. In this project, we explore techniques and limitations of applying DCGANs to sparsely populated datasets. We will show that:

- a limited corpus of data provides little information about the desired generated images
- a limited corpus of data quickly leads to overfitting in more complex models
- despite the sensitivity of the model, DCGANs are able to learn general traits about a sparse dataset relatively quickly.

We finally provide our code and instructions for replicating our results in the appendix.

2 Project Objective

The purpose of our project is first and foremost to explore recent machine learning techniques by implementing a functional, parameterized deep convolutional generative adversarial neural networks (DCGAN). Beginning this project, we were constrained by limited computational resources, restricting us to small datasets. With this in

mind, we set relatively simple goals of beating our baseline model by any nonzero threshold and generating images that are better than pure noise.

To achieve these goals, we opted not to use preset packages such as scikit-learn and instead to fully customize the model using PyTorch, an experimental imperative Python-based machine learning library published by Facebook Research. This allowed us to abstract many of the statistical details while still having the flexibility to experiment with our model.

3 Related Resources

Several publications have helped us better understand the process of building and tuning DCGANs for our project.

3.1 CNNs

The ImageNet classification paper by Krizhevsky et al. [5] provides many of the fundamental principles required for building convolutional neural networks (CNNs). Most notably, strategies to reduce overfitting including data augmentation and dropout are mentioned as well as a discussion on incorporating ReLU nonlinearity and normalization.

3.2 DCGANs

A paper by Radford et al. on DCGANs [7] suggests a novel layer architecture suitable for training on unclassified data that differs from traditional methods of neural network convolutions. The paper suggests the use of the LeakyReLU activation, batch normalization in both the discriminator and generator, and removing the use of pooling layers in exchange for strided convolutions in both the discriminator and generator.

3.3 Loss Function

Janocha and Czarnecki wrote a useful paper [3] on the use cases of many loss functions. In particular, we are interested in the binary cross-entropy loss since we would like to detect how far our parameter distribution deviates from the true distribution of a binary classification. Specifically, cross-entropy loss function is calculated as:

$$H(y, \hat{y}) = - \sum_i y_i \log \hat{y}_i$$

3.4 PyTorch

It is worth mentioning our use of PyTorch, a Python library for writing machine learning programs under an imperative paradigm. Unlike TensorFlow, PyTorch provides access to intermediate calculations in a neural network computation graph. Unfortunately, the library at this time is quite unstable as much of its functionality is still being regularly added to this day. Nevertheless, the neural network package was more than sufficient to build our model’s architecture as most of the tools mentioned in our relevant resources already had implementations in the library.

3.5 Odyssey

We later got access to Harvard’s cluster computing service, Odyssey. By SSHing into a remote server, we were able to batch run numerous jobs at once. One major pitfall of Odyssey, however, was that its graphics drivers are currently too old for PyTorch to use its GPU library, so we were left with running exclusively CPU-based implementations on Odyssey.

4 Datasets

4.1 Van Gogh Dataset

Our data was collected from the ICIP Van Gogh image dataset [1] which contained 333 high-resolution images of paintings by various painters. 124 of the paintings are definitively made by Van Gogh, while two others are either controversially by Van Gogh or disputed to be by Van Gogh. For simplicity, we decided to classify these two paintings as paintings by Van Gogh, totaling 126 samples in our positive class and 207 samples in our negative class.

Despite the small dataset, image processing was still intractable due to the large and irregular dimensionality of the images. Our solution was to resize the images to be 64×64 . With the reduced image size, our three-channel RGB images can be expressed as a vector $\mathbf{x} \in \mathbb{R}^{3 \times 64 \times 64}$ with $x_i \in [0, 255]$.

After working through several iterations of models, we then decided to normalize each pixel’s channel to be between -1 and 1. Thus, the range of some feature x_i would now lie in the range $[-1, 1]$. Doing so minimizes the scale of mathematical operations, reducing imprecision from floating point calculations.

4.2 Kaggle Cat-Dog Dataset

As we gained access to more powerful computing power through Odyssey, we also began to experiment with a secondary dataset of cat and dog images from Kaggle. [4] This dataset is comprised of 25,000 labeled images of cats and dogs in a variety of poses and breeds. Like the Van Gogh paintings, these images were also downsized to be 64×64 when fed into the DCGAN.

5 Model Architecture

Our DCGAN consists of two neural networks. The discriminator, known in the literature as the detective, is a multi-layer convolutional neural network that takes an image as an input and classifies it as “Van Gogh” or “Not Van Gogh.” The generator, also known as the forger, is a deconvolutional neural network (DeCNN) that produces a random input seed vector and generates an image.

Over several iterations, as the discriminator becomes better at both classifying the training data and forged data from the generator, the generator learns to create better forgeries that are able to trick the discriminator. The generator’s training is done without ever having access to the true training data.

5.1 Layer Design

The CNN and DeCNN are roughly inverses of one another. The CNN consists of a set of convolutional layers followed by a final linear projection onto two weights representing the unnormalized probabilities of each class. Each convolutional layer consists of the following:

- Convolution: selects weights for the next layer based on a filter of weights from the previous layer
- Dropout: zeros out weights randomly to prevent overfitting
- Batch Normalization: normalizes the weights as a form of regularization
- Leaky ReLU: allows for nonzero gradients to prevent overfitting
- Max Pooling: dimensionality reduction

In contrast, the DeCNN consists of several deconvolutional layers followed by an output Tanh layer. Each of these convolutional layers consists of:

- Transpose Convolution: “deconvolution”
- ReLU
- Batch Normalization
- Max Pooling

Note that regular ReLU is used instead of Leaky ReLU, preventing exploration toward negative weights. Furthermore, there is no dropout since we would like to preserve all information during image generation, and the final layer is a Tanh which maps pixel channel values to a number between -1 and 1. During image generation, these values are scaled between 0 and 255 to represent RGB colors.

5.2 Parameter Space

In terms of these functional components in each convolutional layer, all of the layers are identical within each neural network. However, the parameters that define the convolutional layers may vary widely from model to model. The following are the parameters to our convolutional layers:

- number of layers
- kernel dimensions at each layer: size of the filter used to perform convolutions
- channels: number of features learned at every layer. The more features, the stronger but the more computationally expensive the model becomes.
- strides: distance a filter travels for each weight calculation
- padding: adjustments for dimensionality
- pooling: scale of dimensionality reduction

Furthermore, the neural networks themselves differ in initialization as well. We design our neural networks to have the following hyperparameters:

- input/output image dimensions. This is set to 64×64
- the number of generated samples when comparing models and training the generator
- batch size: the number of samples to process at a time. With smaller batch sizes, gradient updates tend to perform more quickly at the cost of sample skew due to stochasticity.
- learning rate: step size when gradients are updated
- train/test split: 40% of our data is reserved for testing, while the rest is for training

When tuning these parameters, it is important to note that the output size at each layer is a function of the strides, padding, pooling, and kernel dimensions. Since the CNN expects a fixed size input, it must be the case that the parameter configuration for the DeCNN yields the same image size.

6 Model Training

6.1 Loss and Accuracy

We measure the performance of our DCGAN in comparison with the regular CNN classifier. To do so, we ran different configurations of our model over up to 1000 epochs. The model alternates training between training the discriminator and the generator based on the outputs of one another. A training epoch is defined by the following pseudocode:

```
def train(d, g):
    # train discriminator on real data
    d.train(training_data)
    # Train discriminator on forgeries
    d.train(g.forge())
    # Train generator to forge images
    # that trick discriminator
    g.train(d)
```

Thus, for each epoch, three gradient updates are made, one on discriminating the original training data, one on discriminating forged data, and one on generating forgeries. We will refer to each of these losses as “Detective”, “Compare”, and “Generator”, respectively. We use the Adam Optimizer set at a learning rate of 0.001 with the loss function being defined as the per-unit cross entropy. Using 8 cores of CPU with 5GB of RAM, we are able to complete one epoch of a reasonably dense 5-layer CNN/7-layer DeCNN GAN in about half an hour.

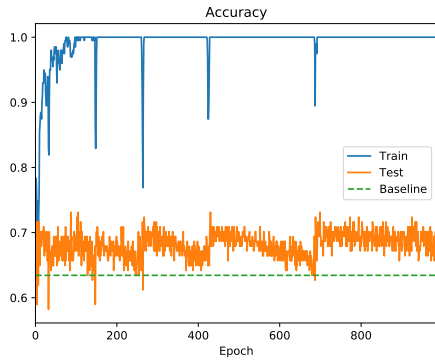
7 Results

For our initial exploration, we run several baseline models with low channel density and a few number of layers to ensure that our model functions without error. From the results in Appendix B, we can see that the generated images are, as expected, lacking recognizable patterns and complexity. Already, the similarity between images over epochs indicate signs of mode collapse. The problem and solution are discussed in Section 8.2 and 9.1.

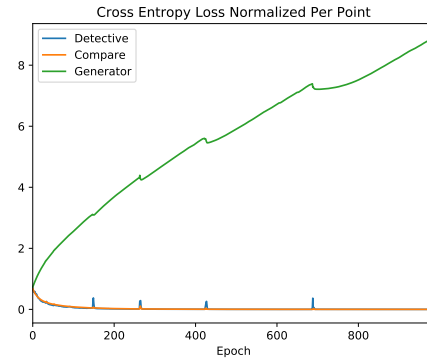
We now present some of the highlights of our results and iterative model development.

Table 1: Many-Layered Image Generation

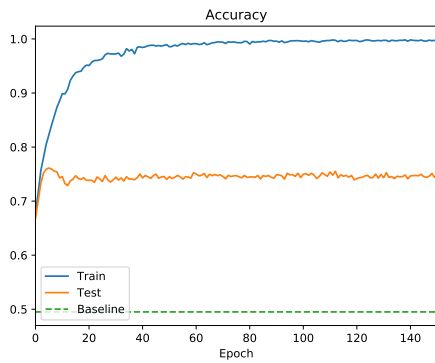
	Initial Epoch	Middle Epoch	Final Epoch
2-layer CNN/10-layer DeCNN			
5-layer CNN/15-layer DeCNN			
5-layer CNN/15-layer DeCNN, catdog			



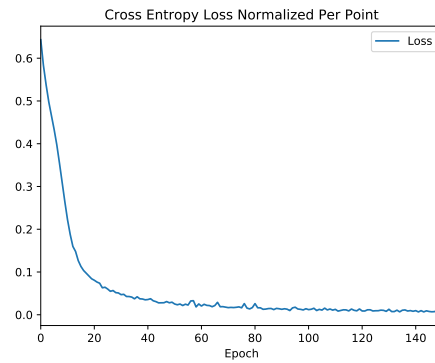
5-layer CNN/15-layer DeCNN Accuracy



5-layer CNN/15-layer DeCNN Loss



5-layer CNN Loss, catdog



5-layer CNN Loss, catdog

Table 1 summarizes the generated images from the best DCGAN model (high channel density) so far, and potential improvements from adding more layers and exercising on a larger dataset (i.e., cat-dog).

Comparing the first and the second row in the table, we see an increase in detail, structure, and tonal complexity that far surpass the baseline generators. Regarding the accuracy curves, the train accuracy quickly goes to one in the first 200 epochs. This indicates that, due to our small dataset, our model quickly to overfit our training data. As for the loss curve, the trend of the generator loss (the generator should create images that fool the discriminator) and compare loss (expected to increase; the generator should trick the discriminator) are inverted.

The results in the third row from cat-dog datasets are more promising. The images depicted are produced by the model run over 7 days but only over 10 epochs. Due to these computational demands of the model, we were unable to generate more detailed images. The plots displayed show a model with the discriminator training alone. From the loss plot, we see that the loss converges to 0, showing again that our model overfits the data. The test accuracy reflects this overfitting with a high training accuracy and a relatively low test accuracy, though the results are still significantly better than the baseline model. Since most of the literature on deep learning suggest training DCGAN model with hundreds of layers and millions of data points, while we were only able to train 15 layers with 333 images for the Van Gogh dataset and 25,000 for the cat-dog dataset, these results are not surprising. Nonetheless, we believe the model has the potential to generate better images with a larger dataset and more computational power.

8 Analysis

We now explore the shortcomings of our model.

8.1 Overfitting

In some of our initial experiments, we trained the generator against an equally strong discriminator. However, the generated images were almost indistinguishable from random noise, and the discriminator and comparison loss quickly approached one after a few iterations.

As several research paper discussed, with such small corpus of training data, it is very likely that the discriminator is overfitting the data. As a result, the discriminator is effectively “too strong,” understanding its training set too well to allow the generator the chance of misclassification and therefore information about how to generate better images. As a solution, we tried to weaken the discriminator by decreasing the number of layers in the CNN. After training several models with weak CNN, we then began to generate more meaningful images. This

is not to say, however, that weakening the CNN strictly increases the performance of the generator. If the CNN is too weak, then the predictive power will be weak, giving the generator the impression that all images it generates are good. In our experiment, we found that even with a single layer CNN, discrimination between forgeries and true images was good, though our generated images became weaker. This too would be a byproduct of the small dataset. Results of some of these models are included in Appendix B.

8.2 Mode Collapse

Mode collapse is a common problem in training generative adversarial neural networks. When the discriminator tends to outperform the generator as in the case of having small training datasets, the generator would end up finding one “mode” with a high probability of discriminator error during an epoch and latch onto these generations. The discriminator then learns to respond to solely these types of images, until the generator finds another such mode and generates those images instead. The generator then hops back and forth between these two types of images, decreasing the diversity in our generated images.

A post from Nibali ^[6] suggested several possible solutions to the problem, most of which require more complex parametrization and larger GPU capacity. As discussed in Section 7, we can improve the diversity of images by introducing model complexity and increasing the training set size. Alternative solutions to solve mode collapse are discussed in Section 9.1.

8.3 Model Comparison

To explore the best performance in classification without the concern of over-fitting, and possibility to classify paintings accurately with any model, we train a strong CNN with 10 layers, and compare its accuracy with traditional statistical and machine learning models, including logistic regression with regularization, random forests and gradient boosting.

In logistic regression, due to the high dimensionality in the data, we apply l_2 regularization and choose λ according to 5-fold cross-validation. In random forest and gradient boosting, we apply fine tuning on all flexible parameters. Details in parameter tuning are included in code files.

From table 2, we can see that, because of the inconsistency of artists’ style in every paintings, none of the models achieves significantly better accuracy than the baseline model (i.e., classify all images as non-Van Gogh). Both neural network models have a similar performance with common tree-based machine learning models. Like the neural networks, the training accuracy of gradient boosting approach 1 very soon.

Table 2: Model Comparison

	Train Accuracy	Test Accuracy	Cross Entropy
Baseline	0.609	0.609	N/A
Logistic Regression	0.628	0.608	0.675
Random Forests	0.803	0.635	0.653
Gradient Boosting	1	0.670	0.631
CNN only	1	≈ 0.67	0.01
DCGAN	1	≈ 0.67	Detective: < 0.01 , Compare: < 0.01 , Generator: > 3.6

9 Discussion

We now discuss methods that may improve future iterations of this project.

9.1 Encouraging Diversity

In a post on mode collapse,^[6] the author introduces several ways to deal with the issue in GANs. Directly encourage diversity by minibatch discrimination and feature mapping may be the most applicable method in our case.

Specifically, we can use batches of samples to directly assess diversity in the generated images. With minibatch discrimination, the discriminator compares images across a batch to determine whether the batch is real or fake, instead of evaluating individual samples. In contrast, feature mapping modifies the cost function in the generator to generate more diverse samples. Other solutions include using unrolled GANs, showing old forgeries to the discriminator repeatedly and multiple GANs which would have differing initial seeds to generate more diverse images at the cost of computation.

9.2 Data Augmentation

Considering the limited number of work by an individual artist, we propose two possible ways to augment the training data for Van Gogh dataset. First, we can “duplicate” each painting by rotating, squeezing, stretching and slightly tinting an original image. Doing this not only enlarges the corpus of training data, but may also increase the diversity of the shade and brightness in generated paintings. Another solution would be to tile the input images. The uniqueness of a painter is not found in the content of the image, but rather in the texture and techniques applied to generate the image. Thus, we could cut our high-resolution images into multiple small pieces rather than resizing them directly. In this way, all information in the original image is kept, giving us

more information in model training. This is a common technique used in style transfer learning, though we were concerned about generating textures rather than paintings, so we decided not to attempt this technique here.

9.3 Alternate Datasets

Other than the volume of data, another deficit in the nature of Van Gogh data set is the inconsistency of features in the paintings of the same class. It is very hard to identify Van Gogh’s work among all paintings even for human beings. As we can see in Section 7, the results from cat-dog images are more promising, because the features within each class are more consistent and the volume of data is significantly larger.

Hence, we believe that GANs are more suitable for images with more consistency and is only satisfying with a large amounts of training data.

9.4 GPU Computing

Had we had access to a proper GPU computing cluster, we would have been able to test more complex models which would have been essential for the scope of this project. Most of the literature on image generation requires neural networks with over one hundred layers. Because we were restricted to such a small number of layers, our images lacked the detail and complexity found in the real images.

9.5 DeLiGAN

As Gurumurthy Sarvadevabhatla and Radhakrishnan propose in their recent paper^[2], DeLiGAN is a novel GAN-based architecture for diverse and limited training data scenarios. By reparameterizing the latent generative space as a mixture model, the performance of the model under limited data significantly improves. However, even with this new method, the model is trained with 60,000 images, while we only have 333 in this project.

References

- [1] G. Folego, O. Gomes, and A. Rocha, “From impressionism to expressionism: Automatically identifying van gogh’s paintings,” in *2016 IEEE International Conference on Image Processing (ICIP)*, Sep. 2016, pp. 141–145. DOI: 10.1109/icip.2016.7532335.
- [2] S. Gurumurthy, R. K. Sarvadevabhatla, and V. B. Radhakrishnan, “Deligan : Generative adversarial networks for diverse and limited data,” *CoRR*, vol. abs/1706.02071, 2017. arXiv: 1706.02071. [Online]. Available: <http://arxiv.org/abs/1706.02071>.
- [3] K. Janocha and W. M. Czarnecki, “On loss functions for deep neural networks in classification,” *CoRR*, vol. abs/1702.05659, 2017. arXiv: 1702.05659. [Online]. Available: <http://arxiv.org/abs/1702.05659>.
- [4] *Kaggle dogs vs. cats dataset*. [Online]. Available: <https://www.kaggle.com/c/dogs-vs-cats/data>.
- [5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [6] A. Nibali, *Mode collapse in gans*, 2017. [Online]. Available: <http://aiden.nibali.org/blog/2017-01-18-mode-collapse-gans/>.
- [7] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *CoRR*, vol. abs/1511.06434, 2015. arXiv: 1511.06434. [Online]. Available: <http://arxiv.org/abs/1511.06434>.

Appendix A Running Instructions

To run the code, please download the released version linked here: https://www.dropbox.com/s/a4fp1480w7xhwlu/cs182_vg_gan.zip?dl=0. The file contains a code folder as well as datasets. The code is written in Python 3.6. To run the GAN training process, you can type:

```
python main.py [generated_file_name] [ vg | catdog ] [test]
```

The generated file name is the file name affix of all images generated by the generator. These images can be found in the created results folder. `vg` and `catdog` are two dataset keywords that define whether to use the Van Gogh or cat-dog dataset. Finally, by including the word “test” at the end of your command, you can test the GAN’s functionality on a small subset of the data with a tiny model.

Here are some examples:

```
python main.py generatedVanGoghs vg
python main.py test vg test
python main.py generated_catdogs catdog
```

Additionally, there is a `main_cnn.py` file which only runs the convolutional neural network. The test keyword does not work, though you can run it on the datasets with the same format as above.

If for some reason the above does not work, you may view just the code in a repository here: https://github.com/hahakumquat/vg_gan. The repository includes a CSV file containing URLs to Van Gogh images as well as a README containing a link to the Kaggle cat-dog dataset. To download each of the images from the CSV, you can use the `jpg_extract` function in `utils.py` to download the images into a folder directory `data_vg/raw/*`.

For sklearn models, simply run the file `sklearn.py` in the code folder. Parameters of the final random forests are chosen based on generated plot in the same folder “foo.png” and “foo2.png”. Parameter tuning and model training for gradient boosting and logistic regression are done automatically in the file.

Appendix B Work Distribution

Michael Ge was in charge of writing the PyTorch code for the DCGAN, DataLoader, plotter file, and utils file and wrote up sections 1-7 of the final writeup.

Anni Wang was in charge of tuning sklearn models(including logistic regression, random forests and gradient boosting), design of poster, and section 7-9 in the final writeup.

Appendix C Additional Images

Below are more images generated by the GAN for other configurations:

Table 3: Low Channel Density Image Generation


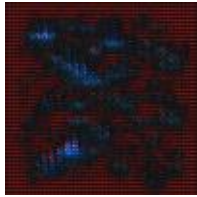
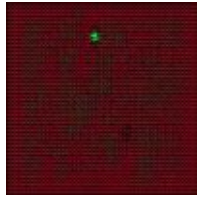
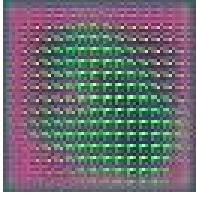
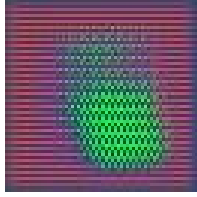
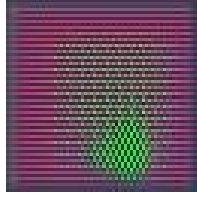
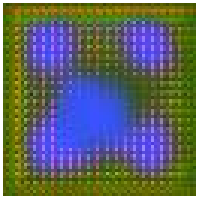
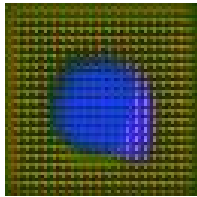
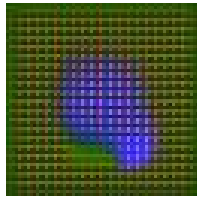
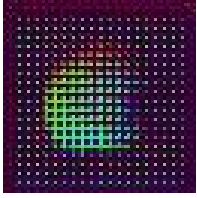
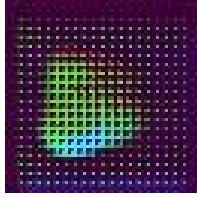
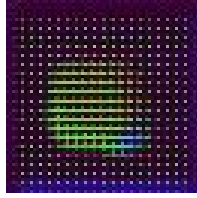
	Initial Epoch	Middle Epoch	Final Epoch
1-layer CNN/1-layer DeCNN			
2-layer CNN/7-layer DeCNN			

Table 4: Tanh-normalized High Channel Density Image Generation

	Initial Epoch	Middle Epoch	Final Epoch
1-layer CNN/7-layer DeCNN			
5-layer CNN/7-layer DeCNN			

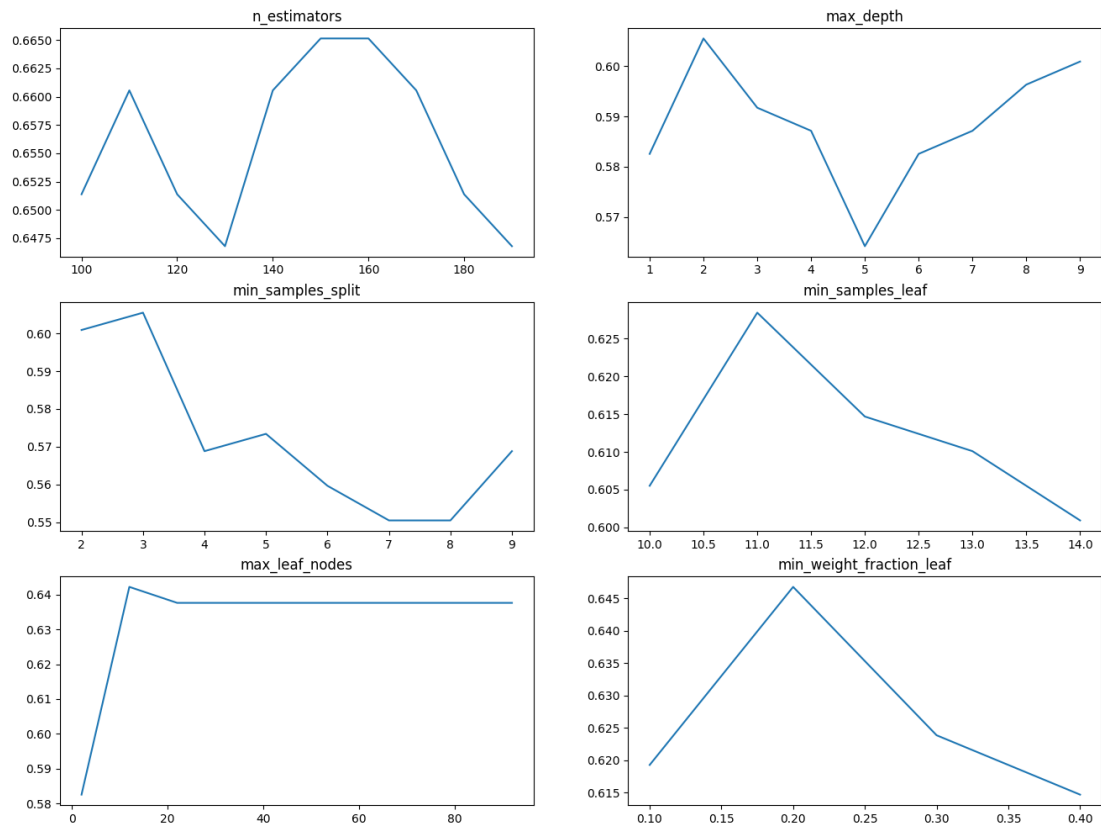


Figure 1: Parameter Tuning of Random Forest