

Deep Learning Personal Project: Symplectic Gradient Adjustment Implementation, Comparison and Evaluation

*** **

School of Data and Computer Science, Sun Yat-sen University
Guangzhou, China

*****@mail2.sysu.edu.cn

Abstract

The state-of-the-art deep learning model is guaranteed by the gradient descent method to converge to a local minimum. However, this guarantee fails in settings with multiple interactive loss functions, especially generative adversarial networks (GAN). Little research deepens into the mechanism of how interactive loss functions affect the performance of the gradient descent method, and only a few adjustments to the current gradient descent method are proved effective in GAN. The authors propose a new adjustment to the current gradient descent method called Symplectic Gradient Adjustment (SGA) to find the stable fixed point in n-player games e.g. GAN. The key to SGA is to decompose the second-order dynamics, which is also known as Hessian, into two components. Both are easy to solve and can converge to Nash equilibrium by gradient descent, and thus they build up mechanics to solve general n-players differentiable games. I (Rui Xi) conduct a series of experiments to evaluate its performance, including four major sections. Firstly, I make a comparison to classic gradient-descent-based optimizers in a Gaussian Mixture Model (GMM). Secondly, a state-of-the-art optimizer, optimistic mirror descent (OMD), is used to analyze the threshold of learning rate. Thirdly, we introduce a high dimensional GMM to identify the universality of the SGA. Finally, a Pytorch implementation of SGA is proposed. The experiments' result shows the effectiveness of SGA in GMM.

1. Introduction

Today's machine learning method strongly dependent on the gradient descent method, to optimize the parameter of the model to reduce a single loss (objective) function. However, there is more and more model which have more than one objective function, e.g. GAN [4]. This model trains the generative model and discriminate model at the same time, who compete with each other via a minimax game. Sim-

ple gradient descent failed to cope with these cases because they provide no guarantee to converge to equilibrium or any stable point, which means the parameters of the model will never stabilize. It has been proved that finding the equilibrium of a game, in general, is a PPAD hard problem [3].

When using gradient descent in games, there are three main difficulties of the problem. Firstly, the potential existence of cycles implies that there are no convergence guarantees. Secondly, even if converged, the rate may be slow due to rotational forces. This problem leads to long training time and small learning rate. Finally, there is no way to measure progress since there are more than one parameters. Current method strongly depends on looking at samples.

Considering all these defects, the authors of [1] propose Symplectic Gradient Adjustment (SGA) to find the stable fixed point in n-player games. In their work, the SGA algorithm outperforms any other optimizers, in both convergence speed and learning rate threshold. Therefore, I am curious about the details of the SGA and the situations it suits.

All these factors motivate me to choose this paper to implement and further investigate the experiments they did. In this paper, I will give a brief introduction to the principle of SGA, and a toy experiment to explain how it works. Then, a fully-functional implementation of SGA is given in Tensorflow, followed by a basic GAN model. After this, the experiments in the paper is repeated to prove the honesty of the authors. Next, a series of further investigations are done regarding its performance compared to other optimizers, both classic ones and state-of-the-art one included. Finally, an alternative implementation of SGA algorithm by Pytorch is proposed. To my best knowledge, it is the first SGA implementation in Pytorch. The main contributions of mine in this paper are listed:

1. Propose Symplectic Gradient Adjustment (SGA) algorithm, Basic multilayer perceptron (MLP) model, Generative Adversarial Network (GAN) and Optimistic Mirror Descent (OMD) implementation in Tensorflow,

as well as SGA implementation in PyTorch.

2. Make a further the comparison of optimizers, including Stochastic Gradient Descent (SGD), AdaGrad, RMSProp, Adam and SGA, when learning a 4 by 4 mixture of Gaussian in 2D.
3. Repeat the comparison of learning rate threshold of OMD and SGA in GMM.
4. Make the comparison of optimizers when learning a high dimensional Gaussian using RMSProp and SGA.

The rest of paper is organized as follows. In Section 2, I introduce previous works related to ours. Then, I briefly describe the principles of SGA algorithm in Section 3. In Section 4, I do a series of experiments, including repeat ones and newly-introduced ones to prove or represent the effectiveness of SGA, as well as some analysis of the result. The whole paper is concluded in Section 5. Acknowledgements, references and source code will be attached in the following sections.

2. Related Works

GAN is first introduced by Goodfellow *et al.* in [4], which trains generative model and discriminate model at the same time, which competes to each other via a min-max game. Such game requires a harsh convergence requirement. Convergence to Nash equilibria in two-player games was studied in Singh *et al.* [8].

To reach the convergence, a bunch of gradient-descent-based optimizers have been introduced. Stochastic Gradient Descent (SGD) [2] is now the most popular optimizer in both Tensorflow and PyTorch. With the adjustment of way decay and momentum, RMSProp [5] enables us to reach a faster and stabler convergence to local minimum, compared to SGD.

However, both of the two methods mentioned above fail to solve GMM problem in GAN, and thus researchers come up with another two methods. Consensus optimization [7] introduce empirical penalties to optimizing loss functions, leading to a better GAN result in a variety of GANs. OMD [6] analyze the behavior of mirror descent (MD) in a class of non-monotone problems, and apply a an "extra-gradient" step, called optimistic descent. They made it to reach convergence in all coherent problems.

Based on the previous works, Balduzzi *et al.* [1] introduce the second-order dynamics decompose into gradient descent method. Their results show that their method outperforms Consensus optimization and OMD in every aspect.

3. Algorithm

In this section, a brief introduction of the SGA algorithm is delivered, together with an example to learn about how SGA works intuitively.

3.1. Helmholtz decomposition and Games

To introduce the construction progress of SGA, we start with the formulate the n-player game. Let $w = \{w_1, w_2, \dots, w_n\}$ denote to the parameters of players, and players has twice continuously losses $l = \{l_1, l_2, \dots, l_n\}$. Player i control w_i . Let $\xi(w) = \{\nabla_{w_1} l_1, \nabla_{w_2} l_2, \dots, \nabla_{w_n} l_n\}$ denote to simultaneous gradient descent of the respective players. Then, use Helmholtz decomposition to divide the Hessian matrix $H(w)$ of w into a symmetric matrix $S(w)$ and anti-symmetric matrix $A(w)$.

Now introducing the concept of stable fixed point: A fix point w^* with $\xi(w^*) = 0$ is stable if $S(w) \succeq 0$ for w is a neighborhood of w^* . Note that a stable fixed point is also a local Nash equilibrium.

If $A(w) \equiv 0$, the game is a potential game. It follows the gradient descent on ξ to converge to fixed point. If $S(w) \equiv 0$, the game is a Hamiltonian game. It follows the gradient descent on $\mathcal{H}(w)$ to converge to fixed point, where $\mathcal{H}(w) = \frac{1}{2} \|\xi(w)\|_2^2$.

With all the previous definitions and theorems, we can draw a adjustment gradient descent ξ_λ instead of the original ξ , which has to satisfy five desiderata:

1. compatible with game dynamic:

$$\langle \xi_\lambda, \xi \rangle = \alpha_1 \|\xi\|^2 \quad (1)$$

, where $\alpha_1 > 0$

2. compatible with potential game:

$$\langle \xi_\lambda, \nabla \Phi \rangle = \alpha_2 \|\nabla \Phi\|^2 \quad (2)$$

, where $\alpha_2 > 0$

3. compatible with Hamiltonian game:

$$\langle \xi_\lambda, \nabla \mathcal{H} \rangle = \alpha_3 \|\nabla \mathcal{H}\|^2 \quad (3)$$

, where $\alpha_3 > 0$

4. attracted to stable fixed point (Nash equilibrium): in neighborhood where $S \succ 0$, require

$$\theta(\xi_\lambda, \nabla \mathcal{H}) \leq \theta(\xi, \nabla \mathcal{H}) \quad (4)$$

5. repelled by unstable fixed point: in neighborhood where $S \prec 0$, require

$$\theta(\xi_\lambda, \nabla \mathcal{H}) \geq \theta(\xi, \nabla \mathcal{H}) \quad (5)$$

To satisfy all the five desiderata, they build adjustment gradient descent:

$$\xi_\lambda \leftarrow \xi + \lambda \cdot A^T \xi \quad (6)$$

and its aligned version

$$\xi_\lambda \leftarrow \xi + |\lambda| \cdot \text{sign}(\langle \xi, \nabla \mathcal{H} \rangle) \cdot H^T \xi \quad (7)$$

The key to SGA is generalized Helmholtz decomposition. In classical Helmholtz decomposition, any vector field ξ in 3-dimensions is the sum of curl-free (gradient) and divergence-free (infinitesimal rotation) component:

$$\xi = \underbrace{\nabla \phi}_{\text{gradient}} + \underbrace{\text{curl}(B)}_{\text{rotation}} [\text{curl}(\cdot) := \nabla \times (\cdot)] \quad (8)$$

Remind that the rotation component is the antisymmetric part of the game Hessian. And thus the relation between classical Helmholtz decomposition and generalized Helmholtz decomposition follows.

3.2. An example

Why an adversarial game is hard to solve by gradient descent? The authors give us the answer that is because of the rotation force in the dynamic of the loss function. A simple adversarial game with a significant rotation force is shown then.

Let's have a look at these 2 function, which player1 control x , while player2 control y :

$$l_1(x, y) = 0.5x^2 + 10xy \quad (9)$$

$$l_2(x, y) = 0.5y^2 - 10xy \quad (10)$$

Obviously, the nash solution falls on $(0, 0)$, where $l_1 = l_2 = 0$. Aware that their dynamic imply that gradient descent will not lead them to $(0, 0)$ directly.

$$\frac{\partial l_1}{\partial x} = x + 10y \quad (11)$$

$$\frac{\partial l_2}{\partial y} = y - 10x \quad (12)$$

Let's denote we init $(x, y) = (1, 1)$ and learning rate $lr = 0.01$. After SGD optimizer steps once, (x, y) change to $(0.89, 1.09)$. When x converge to 0, y become larger surprisingly.

Fig. 1 shows the path of (x, y) when SGD steps 100 times. From left to right, the learning rate is $(0.001, 0.005, 0.01)$ respectively.

Reminded that the dynamic of loss function.

$$\frac{\partial l_1}{\partial x} = x + 10y$$

If there was only x but no $10y$, gradient descent in $\frac{\partial l_1}{\partial x}$ will bring x to 0 immediately. Here, $10y$ is the rotation force. This theorem follows when it comes to $\frac{\partial l_2}{\partial y}$. SGA will add an extra offset to the dynamic and make it better.

Fig. 2 shows the path of (x, y) when SGA steps 100 times. From left to right, the learning rate is $(0.001, 0.005, 0.01)$ respectively.

The code of SGA consists of 2 part. The first part calculate the transposed Hessian matrix H^T of loss function, and get the asymmetric part of it A^T .

Then, we apply this modification to adjust gradient using

$$\xi_\lambda \leftarrow \xi + \lambda \cdot A^T \xi$$

You can refer to attached python source code to have a close look at what it did.

4. Experiments

In this section, a series of experiments regarding the effectiveness of SGA have been conducted.

4.1. Environment Preparation

Due to the unreliability of the devices provided by the course, I turn to InPlusLab (SYSU) and CoLab (Google) for help. The PyTorch environment is enabled by 1 GTX1080Ti GPU from InPlusLab, while the Tensorflow environment is provided by CoLab without knowing the details of devices.

Package Name	Version
numpy	1.16.4
Tensorflow	1.12
kfac	0.1.4
dm-sonnet	1.33
PyTorch	1.0.1

Table 1. Table to some key environments

4.2. SGA algorithm implementation

The SGA algorithm is implement in Tensorflow as a plugin to an optimizer. Readers can go through the source code in the appended jupyter notebook file in **Experiment 1**.

The SGA optimizer **class SymplecticOptimizer** is inherit from **tf.train.Optimizer**, which is modified based on **tf.train.RMSPropOptimizer**. The logic of **class SymplecticOptimizer** is as followed.

First, in **compute gradients**, it calculates the gradients (in the code it is **gards**) of all the parameters (in the code it is **vars**) of the model exactly the same as a **tf.train.RMSPropOptimizer** do (because of a simply call of it). Then, in **apply gradients**, it calculates the asymmetric matrix of the Hessian matrix of all the parameter. In **reset and build graph**, it is automatically applied into the **train op**, then used in **session.run**.

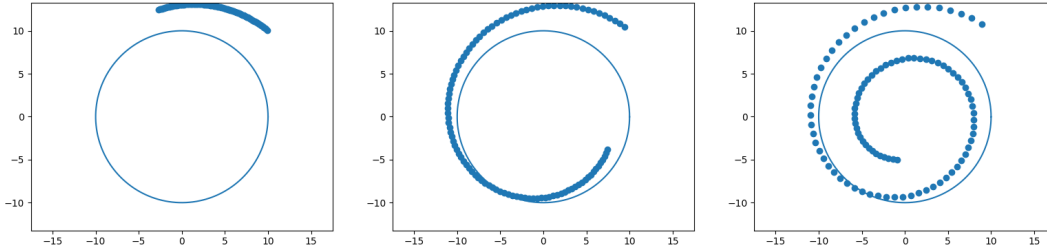


Figure 1. SGD steps 100 times with learning rate is (0.001, 0.005, 0.01) respectively.

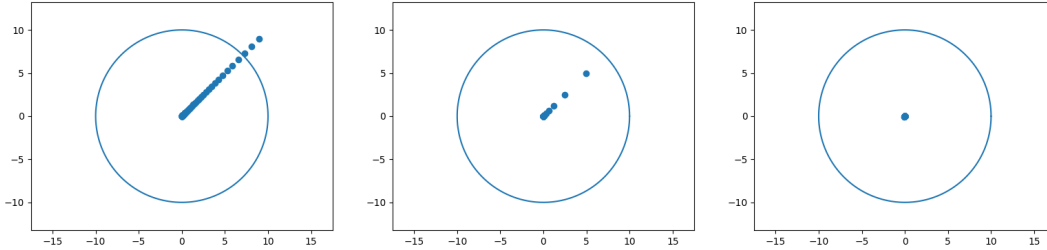


Figure 2. SGA steps 100 times with learning rate is (0.001, 0.005, 0.01) respectively.

4.3. Basic MLP model implementation

A basic multilayer perceptron (MLP) model is implemented in this section. It's a rather simple model compared to what I have learnt in the Deep Learning courses this term. The simple model is also the result of the trade-off between the complexity of calculation and the time investment. Readers can go through the source code in the appended jupyter notebook file in **Experiment 2**.

This simple network contains multiple full connect layers, enabled by `snt.Linear` (alias as `nn.Linear` in PyTorch). After each fc layer, `relu` is used for an activation function. The total depth of the model is defined by an input parameter called `depth`. It is six(6) in the following MLP instance.

4.4. GAN implementation

The GAN structure is almost the same as the GAN implementation we have learnt in the courses, which will be omitted to save space. Readers can also walk through it in **Experiment 2**. The only noteworthy thing is that I use cross entropy loss, enabled by `tf.nn.sigmoid_cross_entropy_with_logits`.

4.5. Learning Gaussian in 2D

This experiment is exactly the authors did in their paper. The authors only compare the performances of RMSProp and SGA, however. In this section, the input sample is $4 \cdot 4$ mixture of Gaussians. All the optimizers are

learning such distribution with $learning_rate = 1e - 4$ and $iter = 10000$. I not only simply repeat their jobs but also add three other optimizers (SGD, Adam, Adagrad) into comparison. The full results are available in the appended notebook in **Experiment 4**. To save space, only the results after 10000 iterations (a.k.a. ended) is presented in the paper's body.

As we can see from Fig. 3 to Fig. 7, the performances of optimizers vary. SGD, Adagrad and Adam fail completely in learning Gaussian in 2D, they obviously restrict themselves in a relatively small area. Even though they have a small loss in both discriminant and generator net, they miss the target of 2D GMM. RMSProp is still far from the expectation, but its performance is relatively better than the previous 3 optimizers. Compared to the previous ones, it successfully reaches 3 of 16 Gaussian instead. SGA is the best optimizer in Experiment 4, which is quite close to the actual $4 \cdot 4$ mixture of Gaussians. If only focus on the loss of discriminant and generator in Table 2, I can hardly tell the differences between them.

4.6. OMD algorithm implementation

To compare the difference of the performances of the state-of-the-art optimizer, an implementation of OMD is given in. However, the objective of the paper is to analyze the performance of SGA, while OMD is only a Control group. Therefore, details of OMD are skipped. If you find it interesting to learn more about, just turn to for further infor-

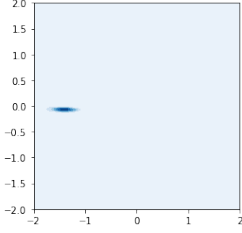


Figure 3. SGD

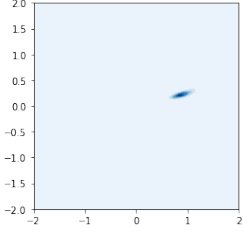


Figure 4. Adagrad

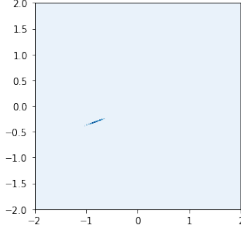


Figure 5. Adam

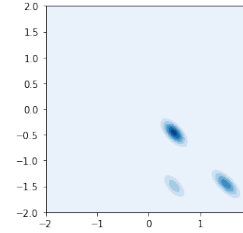


Figure 6. RMSProp

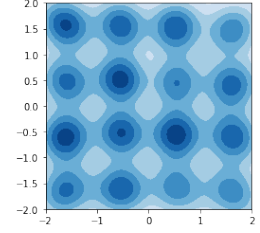


Figure 7. SGA

Optimizer	discriminant loss	generator loss
SGA	1.3853	0.6939
SGD	0.9195	0.8402
AdaGrad	0.9412	0.8080
Adam	0.7635	0.9369
RMSProp	0.5137	1.7146

Table 2. Table of loss in Experiment 4

mation.

4.7. Learning rate threshold of OMD and SGA in GMM

Learning rate threshold is an important metric to analyze the performance of an optimizer. In this section, I re-conduct 2D GMM experiments for several times to locate the learning rate threshold of SGA. Readers can go through the whole experiments in **Experiment 6**. Due to the limitation of time, only threshold of SGA is listed.

As we can see from Fig. 8 to Fig. 9, the approximate threshold lay on near $lr = 1e - 4$. A further investigation is conducted in the Fig. 12 to Fig. 15, which show us at least SGA work well in a threshold $[8e - 5, 1.6e - 4]$.

4.8. High dimensional Gaussian

Learning a high dimensional Gaussian using a GAN is an experiment that was proposed in A classification-based study of covariate shift in gan distributions by Santurkar *et al.* in ICML 2018. In this introduced analysis method, they judge whether the GAN learn characteristic from all the directions via walking through the distribution of eigenvalue number. Fig. 16 give a direct illustration of their method: the real dataset has a more gentle and horizontal distribution in eigenvalue, while generative dataset does not.

In the experiments below, the graphs plot all 75 singular values, in decreasing order, of the covariance matrix of the data generated by the GAN. To address this problem, we use $learning_rate = 2e - 4$ and $iter = 20000$. Readers can go through source code appended in **Experiment 7**.

I only analyze the performance of RMSProp and SGA using this method (the other optimizers are really disappointing, as I just omit them). Due to the restriction in

length, I only put the figures before and after training in here. Readers can check the rest up in the appended file.

As we can see it from Fig. 17 and Fig. 19, the starting point of RMSProp and SGA is almost the same. However, after 20000 times of iterations, the distribution of eigenvalue of the GAN using RMSProp deviates, which we can see in Fig. 18, while that of SGA becomes gentle, as we can see in Fig. 20. These results strongly suggest that the GAN using SGA have an advantage to that using RMSProp. The conclusion is drawn.

4.9. SGA implementation in PyTorch

To my best knowledge, there is no SGA implementation in PyTorch currently. And thus, after implementing it in Tensorflow, I try to build it in PyTorch as well. However, the difference between the mechanism of the optimizer in Tensorflow and PyTorch is considerably large, it took a long time to do this job.

The final implementation is given in the **Experiment 8** of the attached file. Be aware that, due to the limitation in time, I cannot fully test whether it meets all the needs as that implemented in Tensorflow. Use it at your own risk.

5. Conclusion

Applying Helmholtz decomposition, the authors propose a new adjustment to the current gradient descent method called Symplectic Gradient Adjustment (SGA) to find the stable fixed point in n-player games e.g. GAN, leading to a brand-new optimizer call SymplecticOptimizer. I (Rui Xi) repeat the experiments the authors conducted in their paper, as well as add a series of comparison experiments to prove the correctness and effectiveness of the SGA algorithm. I first compare SGA with 4 classic optimizers in a 2D GMM problem. The results imply that only SGA can cope with such a situation. Followed by this, I implement another state-of-the-art optimizer OMD and compare their learning rate threshold by a time-consuming experiment. The result indicates that SGA does have a wider threshold in the learning rate, which makes it outperforms OMD. Then, I evaluate the performance of GANs using SGA and RMSProp in a brand-new method. The results suggest that the dataset generated by GAN using SGA has a better outcome in learning

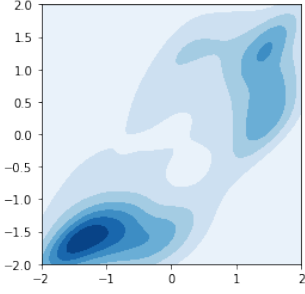


Figure 8. $lr = 5e - 5$

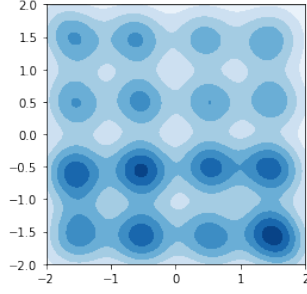


Figure 9. $lr = 1e - 4$

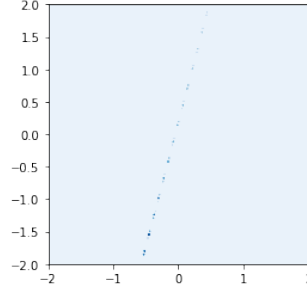


Figure 10. $lr = 5e - 4$

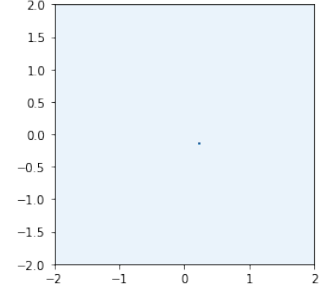


Figure 11. $lr = 1e - 3$

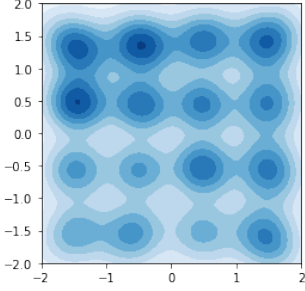


Figure 12. $lr = 8e - 5$

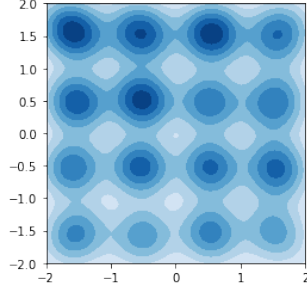


Figure 13. $lr = 1.2e - 4$

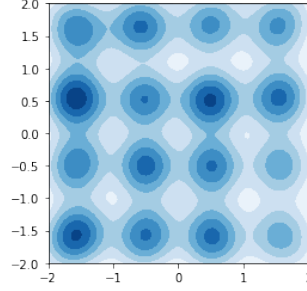


Figure 14. $lr = 1.4e - 3$

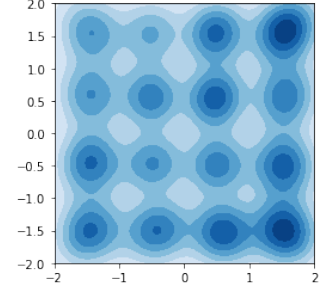


Figure 15. $lr = 1.6e - 4$

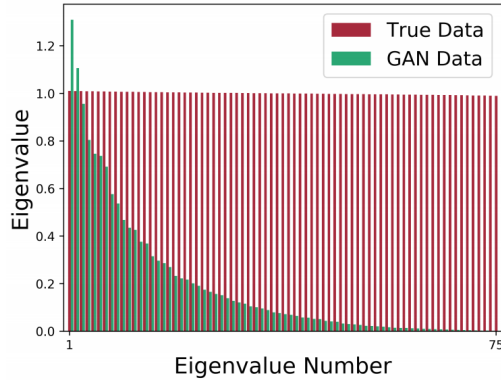


Figure 16. Explanation to Santurkar's work

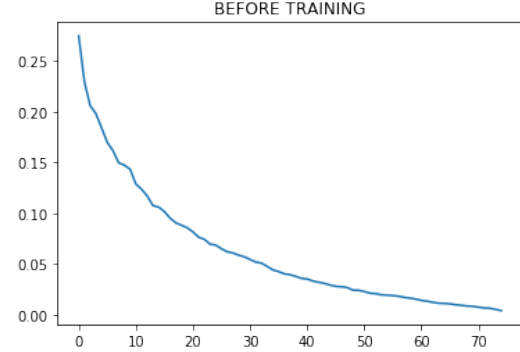


Figure 17. RMSProp in High dimensional Gaussian, before training

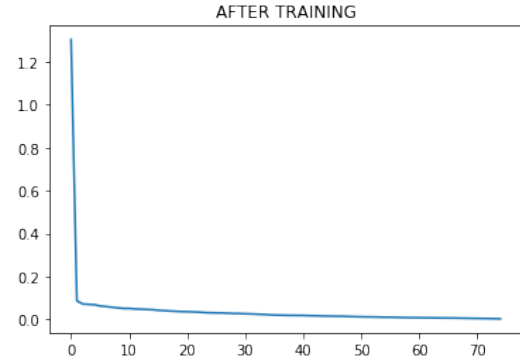


Figure 18. RMSProp in High dimensional Gaussian, after training

features from all the direction of the original dataset. All these results prove the effectiveness of SGA.

In the future work, I will try migrate SGA into PyTorch and make a comparison between Consensus optimization [7] and SGA, if possible.

6. Acknowledgements

My sincere thankfulness to the lecturer of Deep Learning course, Prof. Wang, who open the door of Neural Network to me. Of course, I appreciate it that all the teacher assistances who prepare bloody 19 pieces of courseware for us. Without their work, I could never have such an understanding of the nature of Neural Network. Last but not least, it is my girlfriend Dyla who helped me debug the PyTorch code

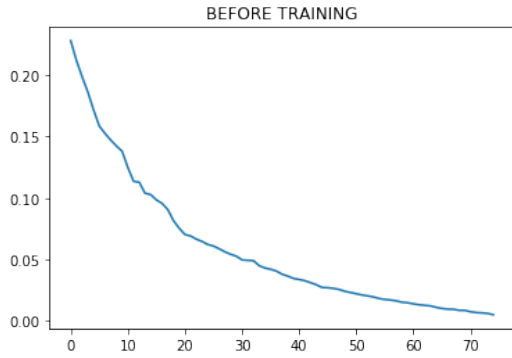


Figure 19. SGA in High dimensional Gaussian, before training

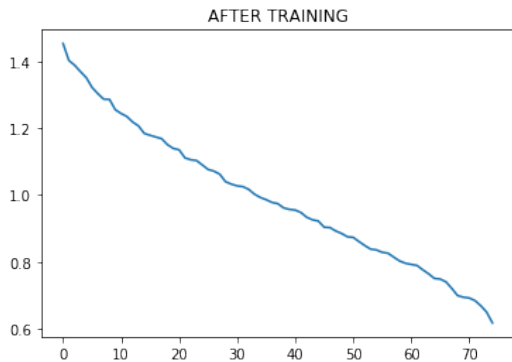


Figure 20. SGA in High dimensional Gaussian, after training

during the experiment. I will never forget her kindest help.

References

- [1] David Balduzzi, Sebastien Racaniere, James Martens, Jakob Foerster, Karl Tuyls, and Thore Graepel. The mechanics of n-player differentiable games. *arXiv preprint arXiv:1802.05642*, 2018.
- [2] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.
- [3] Constantinos Daskalakis, Paul W Goldberg, and Christos H Papadimitriou. The complexity of computing a nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.
- [4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.
- [5] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.
- [6] Panayotis Mertikopoulos, Bruno Lecuat, Houssam Zenati, Chuan-Sheng Foo, Vijay Chandrasekhar, and Georgios Pilioras. Optimistic mirror descent in saddle-point problems: Going the extra (gradient) mile. 2019.
- [7] Lars Mescheder. On the convergence properties of gan training. *arXiv preprint arXiv:1801.04406*, 1:16, 2018.
- [8] Satinder Singh, Michael Kearns, and Yishay Mansour. Nash convergence of gradient dynamics in general-sum games. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 541–548. Morgan Kaufmann Publishers Inc., 2000.

Deep_Learning_Personal_Project

July 12, 2019

1 Deep Learning Personal Project: SGA implementation, Comparison and Evaluation

- Code implemented by Sirius See, SDCS, SYSU
- Student Name: Xi Rui
- Student ID: 16340247

1.1 Acknowledgement

- The Symplectic Gradient Adjustment (SGA) algorithm is first introduced in [The Mechanics of n-Player Differentiable Games](#) by **David Balduzzi** in ICML 2018.
- The code is inspired by **Deep Mind's** [SGA algorithm](#) in 2018.
- The Optimistic Mirror Descent (OMD) is first introduced in [Optimistic mirror descent in saddle-point problems: Going the extra \(gradient\) mile](#) by **Panayotis Mertikopoulos** in ICLR 2019
- **A series of experiments is added by Sirius See (Xi Rui)** as the **solo project of Deep Learning course** in Sun Yat-sen University. Further detailed description of my work will be listed in the following section.
- **Google Colab** and **SYSU InPlusLab** provide the devices (mainly GPUs) for me to conduct the following experiments.

1.2 Table of Content

0. Tensorflow Environment Preparation
1. Experiment 1: Symplectic Gradient Adjustment (SGA) algorithm implementation
2. Experiment 2: Basic multilayer perceptron (MLP) model implementation
3. Experiment 3: Generative Adversarial Network (GAN) implementation
4. Experiment 4: Comparison of optimizers when learning a 4 by 4 mixture of Gaussian in 2D
5. Using Stochastic Gradient Descent (SGD)
6. Using AdaGrad
7. Using RMSProp
8. Using Adam
9. Using SGA
10. Experiment 5: Optimistic Mirror Descent (OMD) algorithm implementation

11. Experiment 6: Comparison of learning rate threshold of OMD and SGA in Gaussian mixture model
12. Using OMD
13. Using SGA
14. Experiment 7: Comparison of optimizers when learning a high dimensional Gaussian
15. Using RMSProp
16. Using SGA
17. Experiment 8: An SGA implementation in PyTorch

1.3 Tensorflow Environment Preparation

```
In [0]: !pip install dm-sonnet
        !pip install 'kfac[tensorflow]'
```

[illegible]

Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (from keras-applications>=0.1.2)
Installing collected packages: kfac
Successfully installed kfac-0.1.4

```
In [0]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        import math
        import os
        import numpy as np
        import sonnet as snt
        import tensorflow as tf
        import kfac

        import matplotlib.pyplot as plt
        import scipy as sp
```

W0711 01:50:12.525845 140060646823808 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/resource_variable_ops.py:435: DeprecationWarning: tf.nn.conv2d is deprecated and will be removed in a future version. Please use tf.nn.conv2d_v2 instead.

W0711 01:50:12.527219 140060646823808 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/resource_variable_ops.py:435: DeprecationWarning: tf.nn.conv2d is deprecated and will be removed in a future version. Please use tf.nn.conv2d_v2 instead.

1.4 Experiment 1: Symplectic Gradient Adjustment (SGA) algorithm implementation

```
In [0]: # help function of SymplecticOptimizer
```

```
def list_divide_scalar(xs, y):
    return [x / y for x in xs]

def list_subtract(xs, ys):
    return [x - y for (x, y) in zip(xs, ys)]

def jacobian_vec(ys, xs, vs):
    return kfac.utils.fwd_gradients(
        ys, xs, grad_xs=vs, stop_gradients=xs)

def jacobian_transpose_vec(ys, xs, vs):
    dydxs = tf.gradients(ys, xs, grad_ys=vs, stop_gradients=xs)
    dydxs = [
        tf.zeros_like(x) if dydx is None else dydx for x, dydx in zip(xs, dydxs)
    ]
```

```

    return dydxs

def _dot(x, y):
    dot_list = []
    for xx, yy in zip(x, y):
        dot_list.append(tf.reduce_sum(xx * yy))
    return tf.add_n(dot_list)

In [0]: #@title Default title text
        # impl of SGA optimizer

class SymplecticOptimizer(tf.train.Optimizer):
    """Optimizer that corrects for rotational components in gradients."""

    def __init__(self,
                  learning_rate,
                  reg_params=1.,
                  use_signs=True,
                  use_locking=False,
                  name='symplectic_optimizer'):
        super(SymplecticOptimizer, self).__init__(
            use_locking=use_locking, name=name)
        self._gd = tf.train.RMSPropOptimizer(learning_rate)
        self._reg_params = reg_params
        self._use_signs = use_signs

    def compute_gradients(self,
                        loss,
                        var_list=None,
                        gate_gradients=tf.train.Optimizer.GATE_OP,
                        aggregation_method=None,
                        colocate_gradients_with_ops=False,
                        grad_loss=None):
        return self._gd.compute_gradients(loss, var_list, gate_gradients,
                                         aggregation_method,
                                         colocate_gradients_with_ops, grad_loss)

    def apply_gradients(self, grads_and_vars, global_step=None, name=None):
        grads, vars_ = zip(*grads_and_vars)
        n = len(vars_)
        h_v = jacobian_vec(grads, vars_, grads)
        ht_v = jacobian_transpose_vec(grads, vars_, grads)
        at_v = list_divide_scalar(list_subtract(ht_v, h_v), 2.)
        if self._use_signs:
            grad_dot_h = _dot(grads, ht_v)
            at_v_dot_h = _dot(at_v, ht_v)
            mult = grad_dot_h * at_v_dot_h

```

```

        lambda_ = tf.sign(mult / n + 0.1) * self._reg_params
    else:
        lambda_ = self._reg_params
    apply_vec = [(g + lambda_ * ag, x)
                  for (g, ag, x) in zip(grads, at_v, vars_)
                  if at_v is not None]
    return self._gd.apply_gradients(apply_vec, global_step, name)

```

1.5 Experiment 2: Basic multilayer perceptron (MLP) model implementation

In [0]: `class MLP(snt.AbstractModule):`

```

    def __init__(self, depth, hidden_size, out_dim, name='SimpleNet'):
        super(MLP, self).__init__(name=name)
        self._depth = depth
        self._hidden_size = hidden_size
        self._out_dim = out_dim

    def _build(self, input):
        h = input
        for i in range(self._depth):
            h = tf.nn.relu(snt.Linear(self._hidden_size)(h))
        return snt.Linear(self._out_dim)(h)

```

In [0]: `def reset_and_build_graph(depth, width, x_real_builder, z_dim, batch_size, learning_rate):`
`tf.reset_default_graph()`

```

x_real = x_real_builder(batch_size)
x_dim = x_real.get_shape().as_list()[1]
generator = MLP(depth, width, x_dim, 'generator')
discriminator = MLP(depth, width, 1, 'discriminator')
z = tf.random_normal([batch_size, z_dim])
x_fake = generator(z)
disc_out_real = discriminator(x_real)
disc_out_fake = discriminator(x_fake)

# Loss
disc_loss_real = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        logits=disc_out_real, labels=tf.ones_like(disc_out_real)))
disc_loss_fake = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        logits=disc_out_fake, labels=tf.zeros_like(disc_out_fake)))
disc_loss = disc_loss_real + disc_loss_fake

gen_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        logits=disc_out_fake, labels=tf.ones_like(disc_out_fake)))

```

```

gen_vars = generator.variable_scope.trainable_variables()
disc_vars = discriminator.variable_scope.trainable_variables()
# Compute gradients
xs = disc_vars + gen_vars
disc_grads = tf.gradients(disc_loss, disc_vars)
gen_grads = tf.gradients(gen_loss, gen_vars)
Xi = disc_grads + gen_grads
apply_vec = list(zip(Xi, xs))

if mode == 'RMS':
    optimizer = tf.train.RMSPropOptimizer(learning_rate)
elif mode == 'SGD':
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
elif mode == 'ADG':
    optimizer = tf.train.AdagradOptimizer(learning_rate)
elif mode == 'ADA':
    optimizer = tf.train.AdamOptimizer(learning_rate)
elif mode == 'SGA':
    optimizer = SymplecticOptimizer(learning_rate)
else:
    raise ValueError('Mode %s not recognised' % mode)

with tf.control_dependencies([g for (g, v) in apply_vec]):
    train_op = optimizer.apply_gradients(apply_vec)

init = tf.global_variables_initializer()

return train_op, x_fake, z, init, disc_loss, gen_loss

```

In [0]: # visualization

```

def kde(mu, tau, bbox=None, xlabel="", ylabel="", cmap='Blues'):
    values = np.vstack([mu, tau])
    kernel = sp.stats.gaussian_kde(values)

    fig, ax = plt.subplots()
    ax.axis(bbox)
    ax.set_aspect(abs(bbox[1]-bbox[0])/abs(bbox[3]-bbox[2]))
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)

    xx, yy = np.mgrid[bbox[0]:bbox[1]:300j, bbox[2]:bbox[3]:300j]
    positions = np.vstack([xx.ravel(), yy.ravel()])
    f = np.reshape(kernel(positions).T, xx.shape)
    cfset = ax.contourf(xx, yy, f, cmap=cmap)
    plt.show()

```

1.6 Experiment 4: Comparison of optimizers when learning a 4 by 4 mixture of Gaussian in 2D

```
In [0]: def train(train_op, x_fake, z, init, disc_loss, gen_loss, z_dim,
                n_iter=10001, n_save=2000):
    bbox = [-2, 2, -2, 2]
    batch_size = x_fake.get_shape()[0].value
    ztest = [np.random.randn(batch_size, z_dim) for i in range(10)]

    with tf.Session() as sess:
        sess.run(init)

        for i in range(n_iter):
            disc_loss_out, gen_loss_out, _ = sess.run(
                [disc_loss, gen_loss, train_op])
            if i % n_save == 0:
                print('i = %d, discriminant loss = %.4f, generator loss = %.4f' %
                      (i, disc_loss_out, gen_loss_out))
                x_out = np.concatenate(
                    [sess.run(x_fake, feed_dict={z: zt}) for zt in ztest], axis=0)
                kde(x_out[:, 0], x_out[:, 1], bbox=bbox)

def learn_mixture_of_gaussians(mode):
    print(mode)
    def x_real_builder(batch_size):
        sigma = 0.1
        skel = np.array([
            [ 1.50,  1.50],
            [ 1.50,  0.50],
            [ 1.50, -0.50],
            [ 1.50, -1.50],
            [ 0.50,  1.50],
            [ 0.50,  0.50],
            [ 0.50, -0.50],
            [ 0.50, -1.50],
            [-1.50,  1.50],
            [-1.50,  0.50],
            [-1.50, -0.50],
            [-1.50, -1.50],
            [-0.50,  1.50],
            [-0.50,  0.50],
            [-0.50, -0.50],
            [-0.50, -1.50],
        ])
        temp = np.tile(skel, (batch_size // 16 + 1, 1))
        mus = temp[0:batch_size, :]
        return mus + sigma*tf.random_normal([batch_size, 2])*0.2
```

```
z_dim = 64
train_op, x_fake, z, init, disc_loss, gen_loss = reset_and_build_graph(
    depth=6, width=384, x_real_builder=x_real_builder, z_dim=z_dim,
    batch_size=256, learning_rate=1e-4, mode=mode)

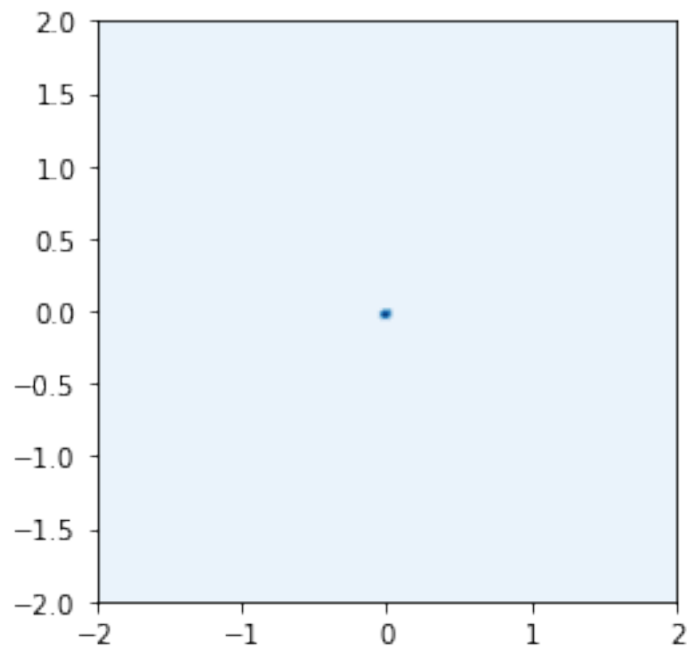
train(train_op, x_fake, z, init, disc_loss, gen_loss, z_dim)
```

1.6.1 Using SGD

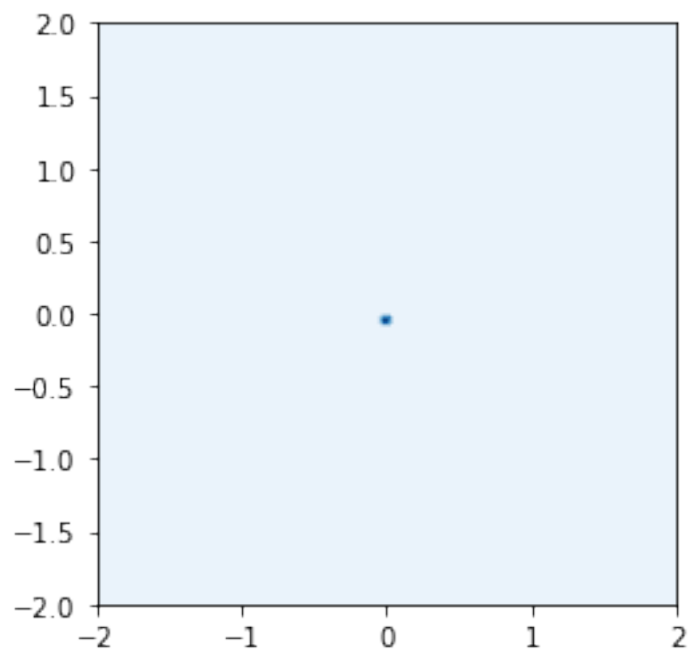
```
In [0]: learn_mixture_of_gaussians('SGD')
```

SGD

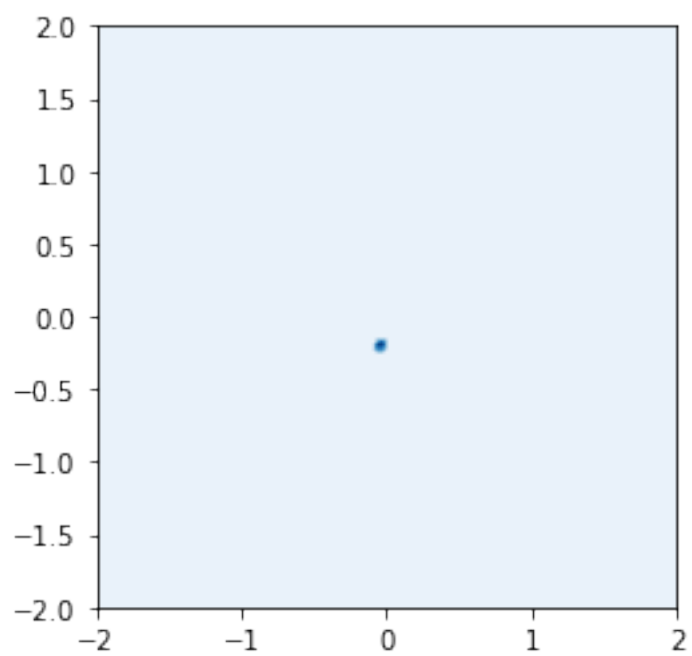
i = 0, discriminant loss = 1.4003, generator loss =0.6933



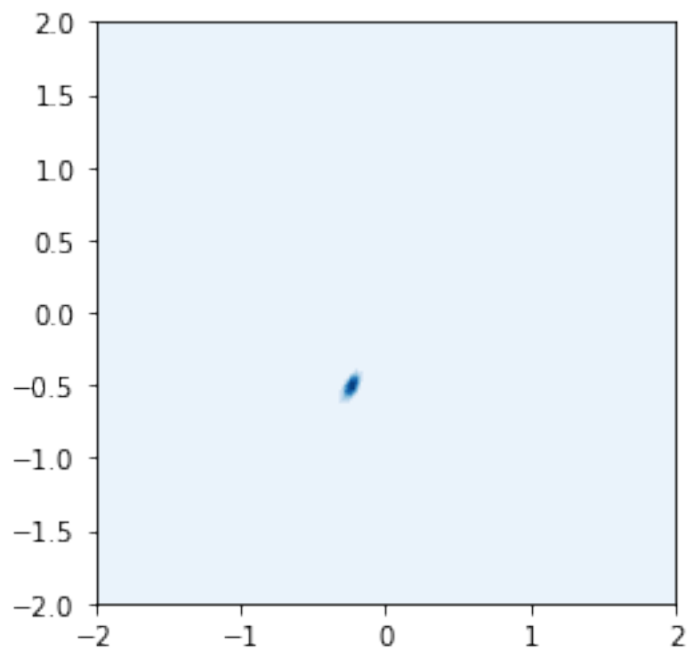
i = 2000, discriminant loss = 1.2123, generator loss =0.6958



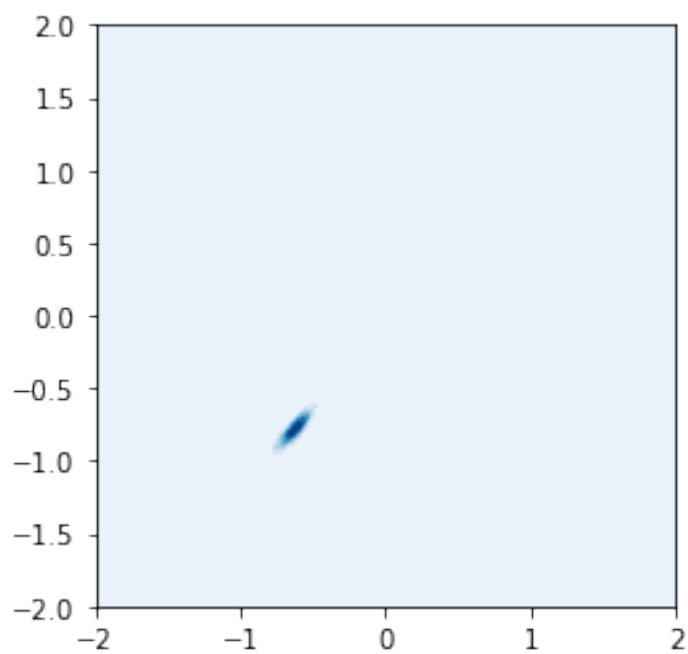
i = 4000, discriminant loss = 1.0371, generator loss = 0.6724



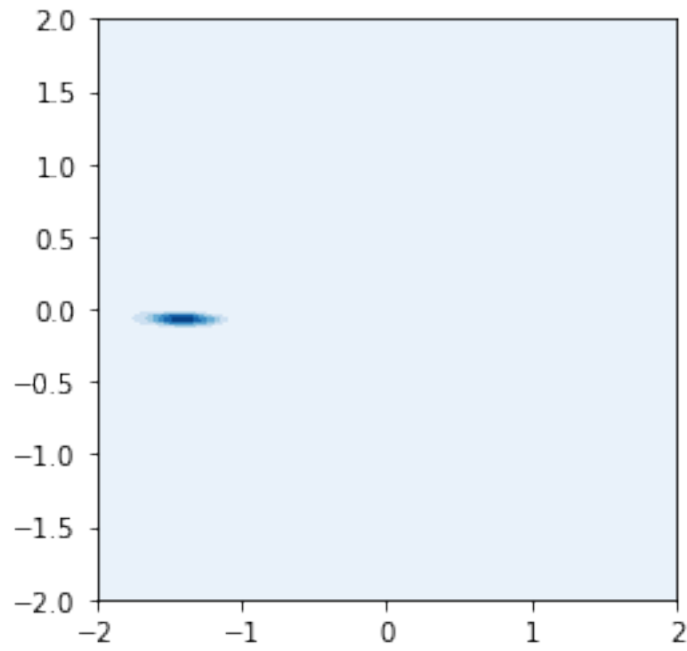
i = 6000, discriminant loss = 1.0122, generator loss =0.6308



i = 8000, discriminant loss = 0.9194, generator loss =0.7399



```
i = 10000, discriminant loss = 0.9195, generator loss =0.8402
```



1.6.2 Using AdaGrad

```
In [0]: learn_mixture_of_gaussians('ADG')
```

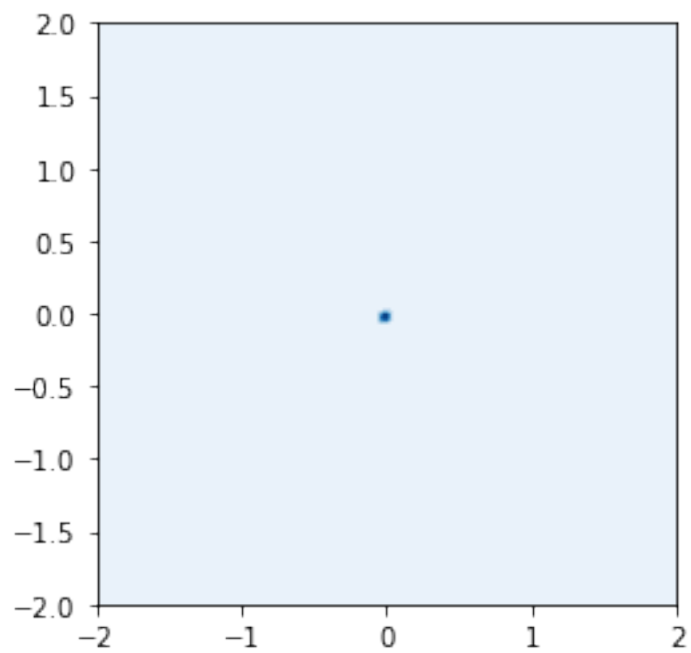
ADG

```
W0710 13:55:39.310465 140150887180160 deprecation.py:506] From /usr/local/lib/python3.6/dist-p
```

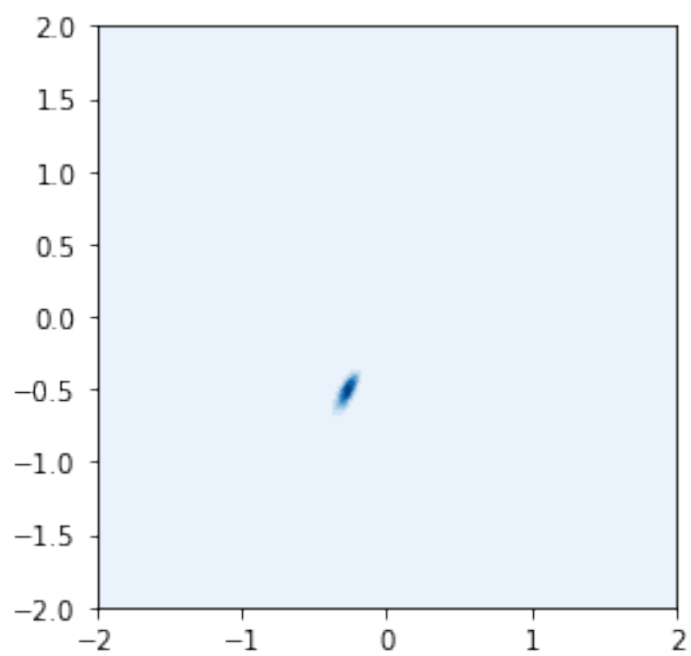
Instructions for updating:

Call initializer instance with the dtype argument instead of passing it to the constructor

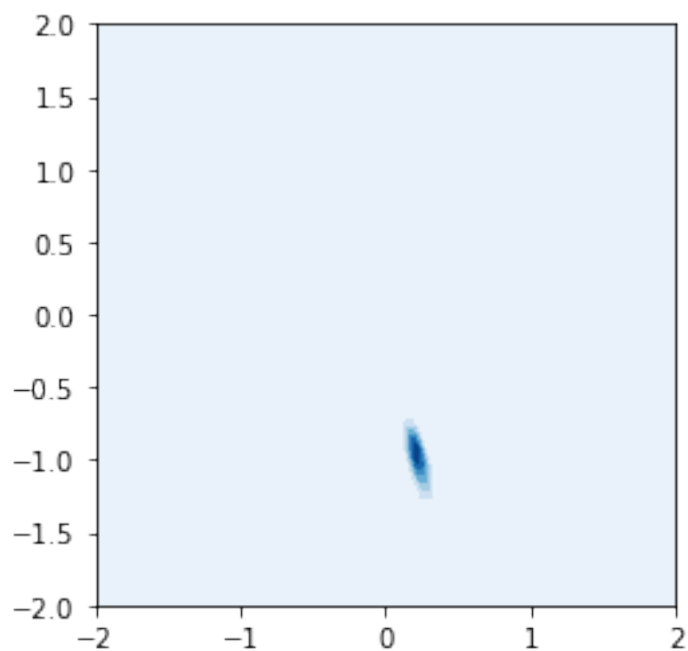
```
i = 0, discriminant loss = 1.3714, generator loss =0.6926
```



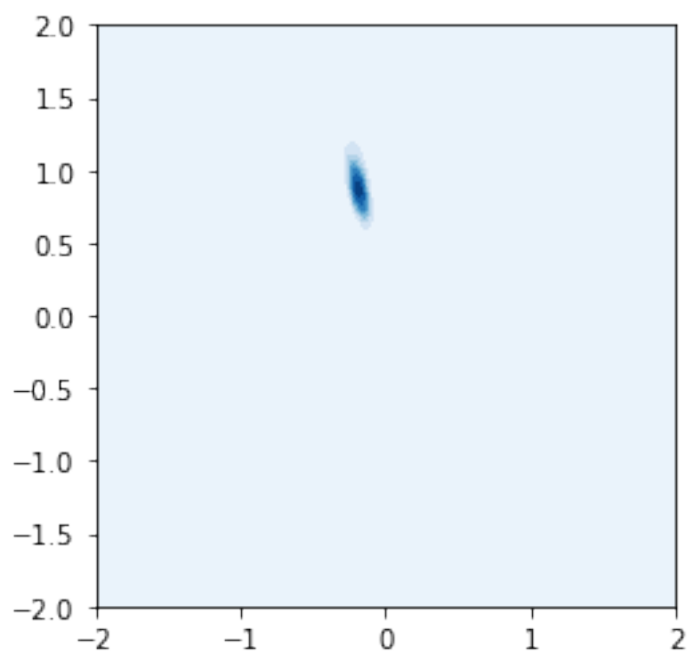
i = 2000, discriminant loss = 1.0970, generator loss = 0.6096



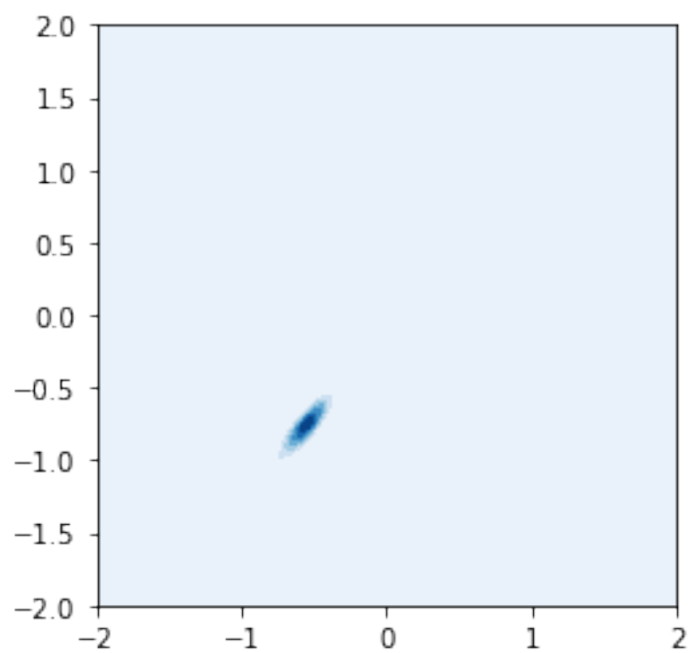
i = 4000, discriminant loss = 0.9673, generator loss = 0.6934



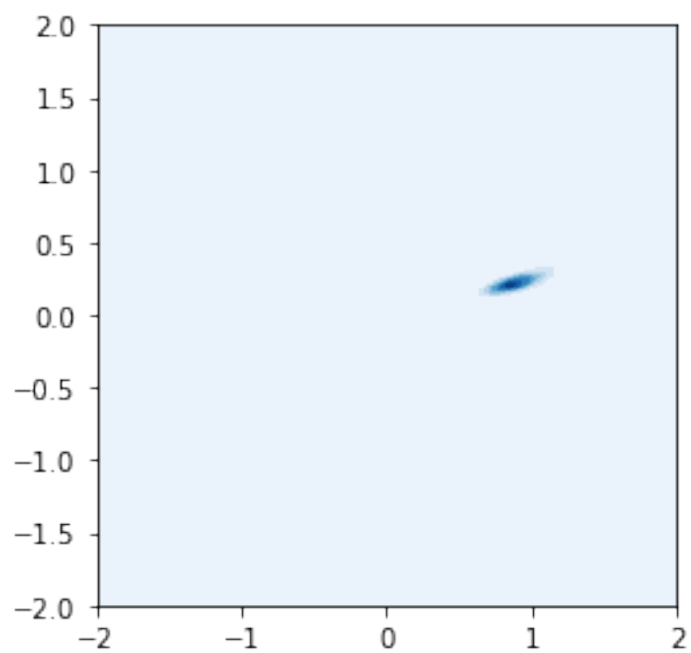
i = 6000, discriminant loss = 1.1352, generator loss = 0.8187



i = 8000, discriminant loss = 1.0060, generator loss = 0.7855



i = 10000, discriminant loss = 0.9412, generator loss = 0.8080

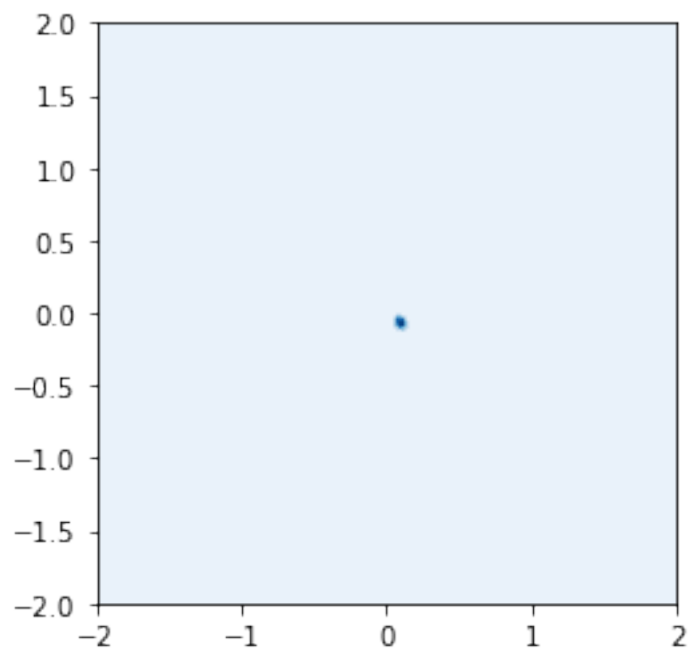


1.6.3 Using Adam

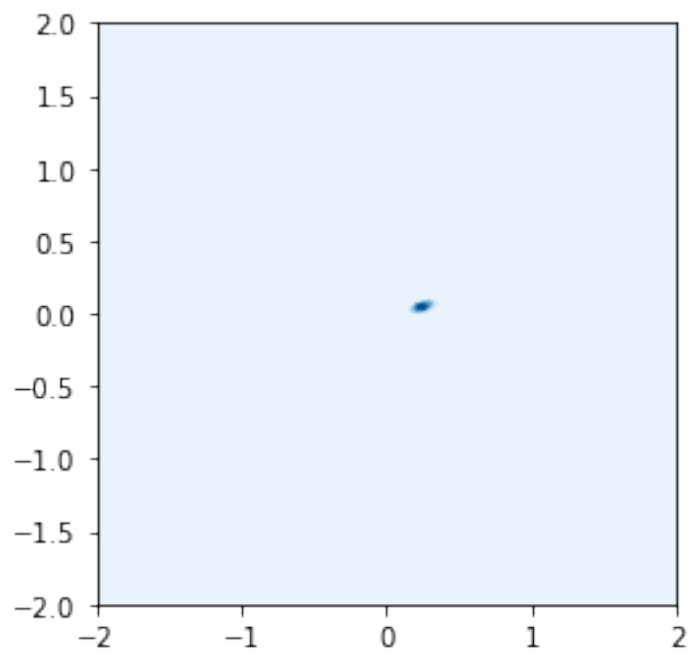
```
In [0]: learn_mixture_of_gaussians('ADA')
```

ADA

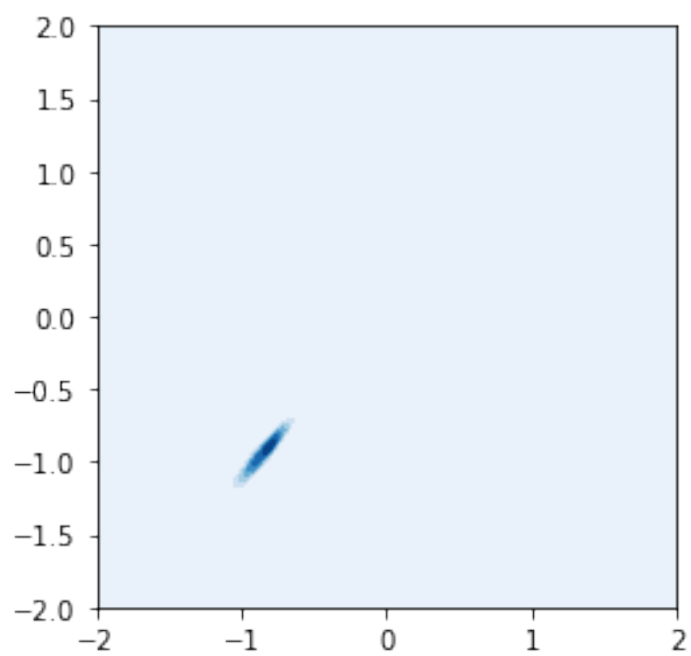
i = 0, discriminant loss = 1.3380, generator loss = 0.6903



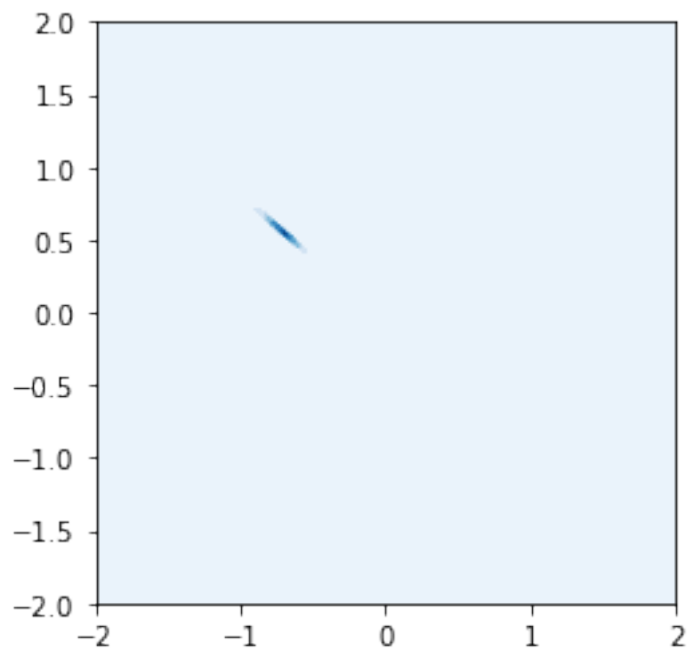
i = 2000, discriminant loss = 1.1390, generator loss = 0.8283



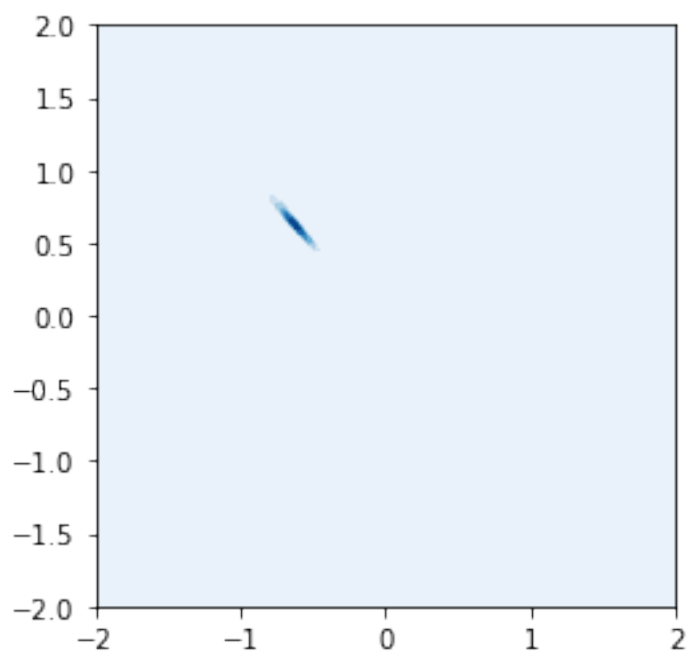
i = 4000, discriminant loss = 0.8575, generator loss =1.1069



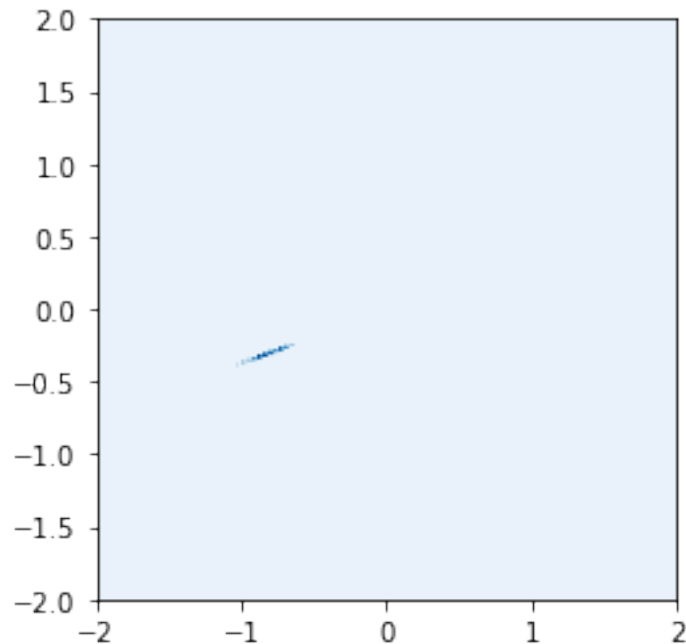
i = 6000, discriminant loss = 1.4832, generator loss =0.9082



i = 8000, discriminant loss = 1.3655, generator loss =0.9416




```
i = 10000, discriminant loss = 0.7635, generator loss =0.9369
```



1.6.4 Using RMSProp

```
In [0]: learn_mixture_of_gaussians('RMS')
```

```
W0710 11:44:01.528981 140150887180160 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/monitored_session.py:110: DeprecationWarning: <code>tf.nn.initialize_all_variables</code> is deprecated and will be removed in a future version. Please use <code>tf.nn.init.initialize_all_variables_v2</code> instead.
```

```
W0710 11:44:01.554638 140150887180160 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/monitored_session.py:110: DeprecationWarning: <code>tf.nn.initialize_all_variables</code> is deprecated and will be removed in a future version. Please use <code>tf.nn.init.initialize_all_variables_v2</code> instead.
```

```
W0710 11:44:01.561922 140150887180160 deprecation.py:506] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/monitored_session.py:110: DeprecationWarning: <code>tf.nn.initialize_all_variables</code> is deprecated and will be removed in a future version. Please use <code>tf.nn.init.initialize_all_variables_v2</code> instead.
```

```
Instructions for updating:
```

```
Call initializer instance with the dtype argument instead of passing it to the constructor
```

```
W0710 11:44:01.562939 140150887180160 deprecation.py:506] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/monitored_session.py:110: DeprecationWarning: <code>tf.nn.initialize_all_variables</code> is deprecated and will be removed in a future version. Please use <code>tf.nn.init.initialize_all_variables_v2</code> instead.
```

```
Instructions for updating:
```

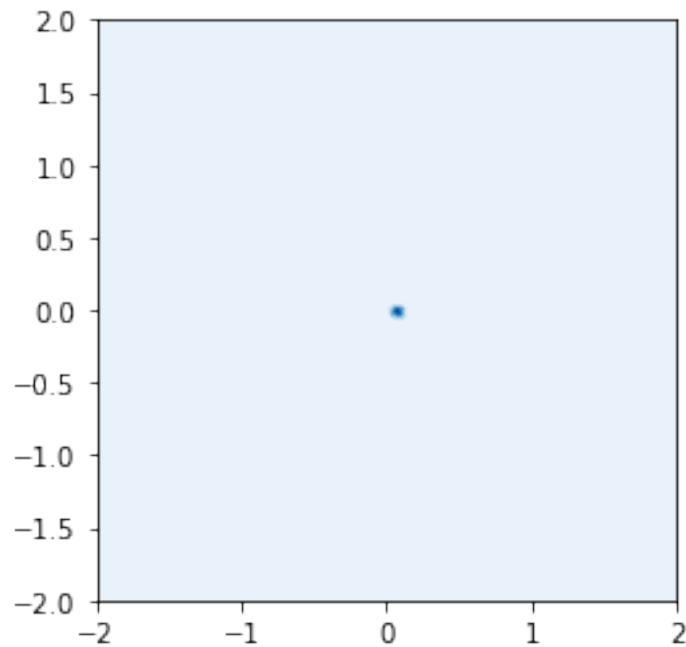
```
Call initializer instance with the dtype argument instead of passing it to the constructor
```

```
W0710 11:44:01.564574 140150887180160 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/monitored_session.py:110: DeprecationWarning: <code>tf.nn.initialize_all_variables</code> is deprecated and will be removed in a future version. Please use <code>tf.nn.init.initialize_all_variables_v2</code> instead.
```

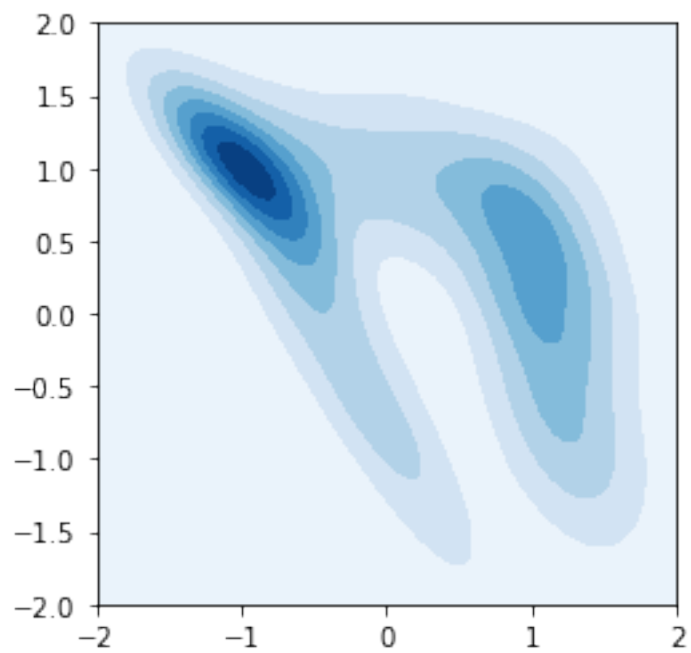
RMS

```
W0710 11:44:01.825356 140150887180160 deprecation.py:323] From /usr/local/lib/python3.6/dist-p
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
W0710 11:44:02.175057 140150887180160 deprecation.py:506] From /usr/local/lib/python3.6/dist-p
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
```

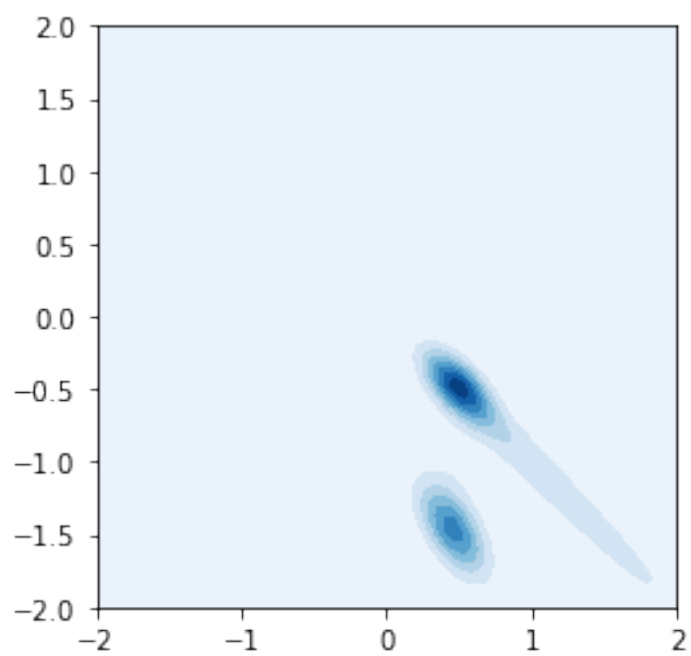
i = 0, discriminant loss = 1.4136, generator loss =0.6941



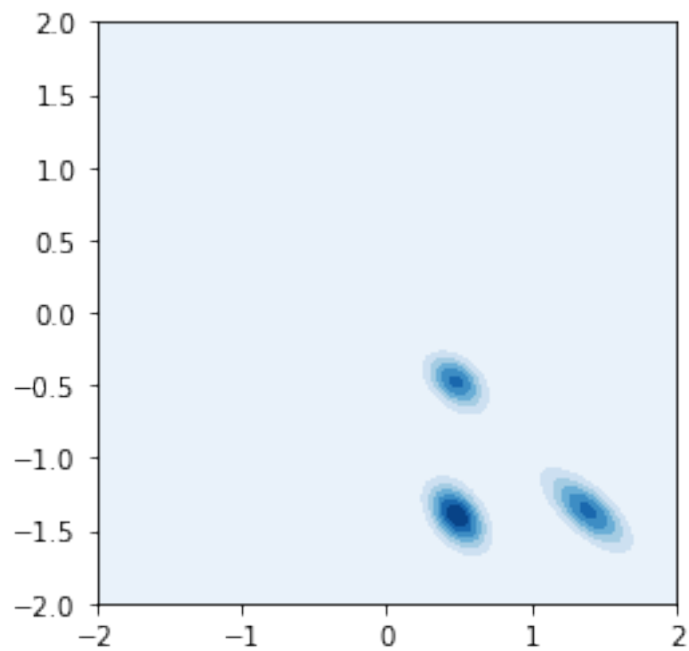
i = 2000, discriminant loss = 1.0527, generator loss =1.5996



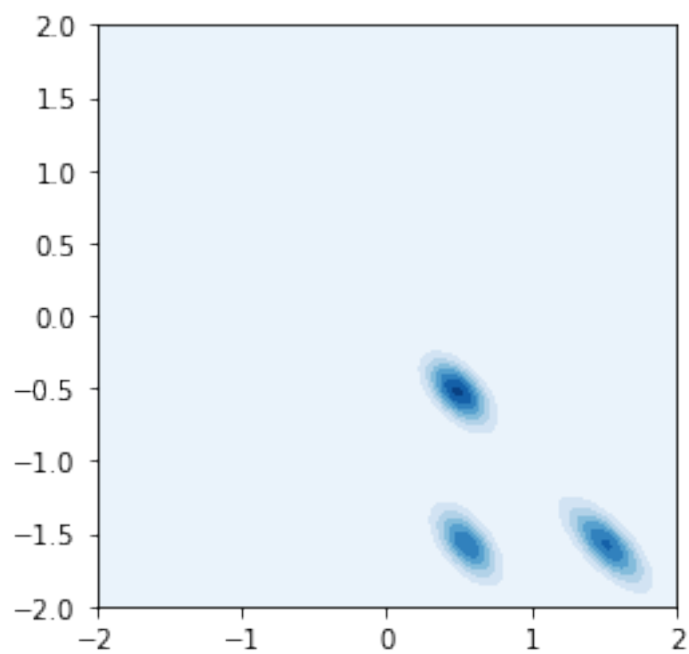
i = 4000, discriminant loss = 0.4317, generator loss =2.0905



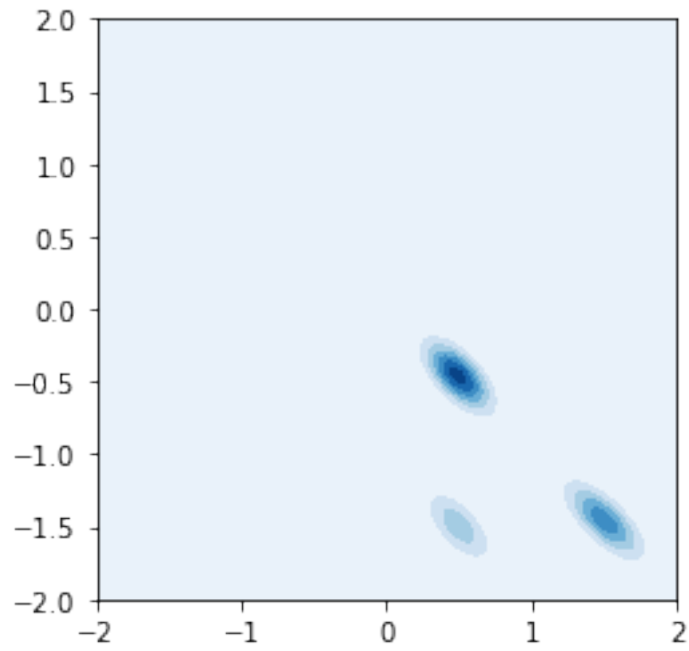
i = 6000, discriminant loss = 0.4968, generator loss =2.1840



i = 8000, discriminant loss = 0.4876, generator loss =1.7091



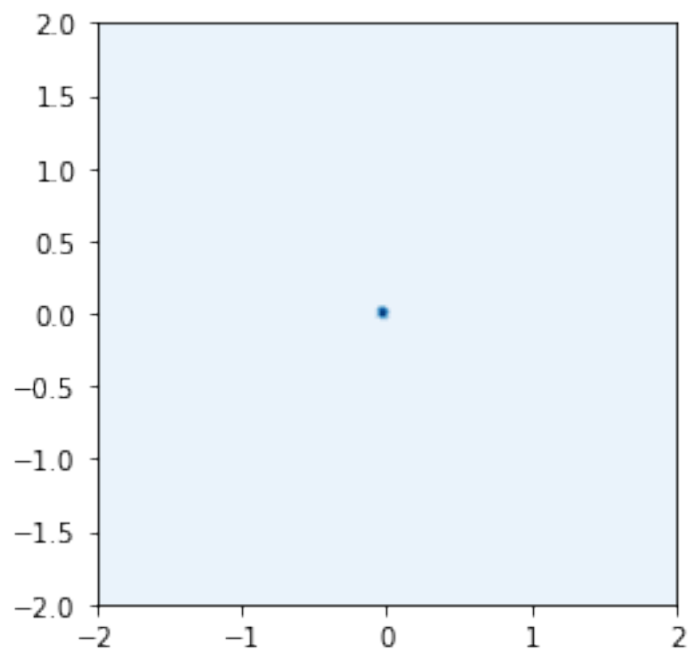
```
i = 10000, discriminant loss = 0.5137, generator loss =1.7146
```



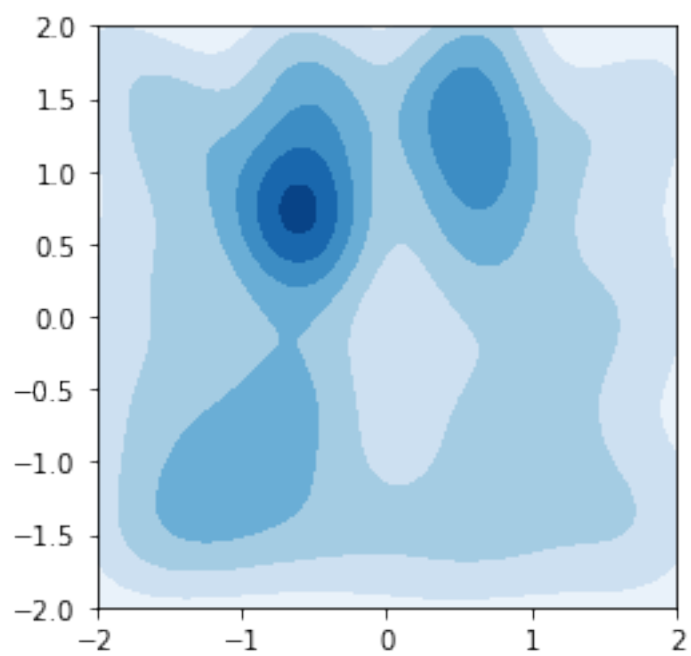
```
In [0]: # Use Symplectic Gradient Adjustment to optimise the GAN parameters.  
        # With SGA, all modes are produced by the trained GAN.  
        learn_mixture_of_gaussians('SGA')
```

SGA

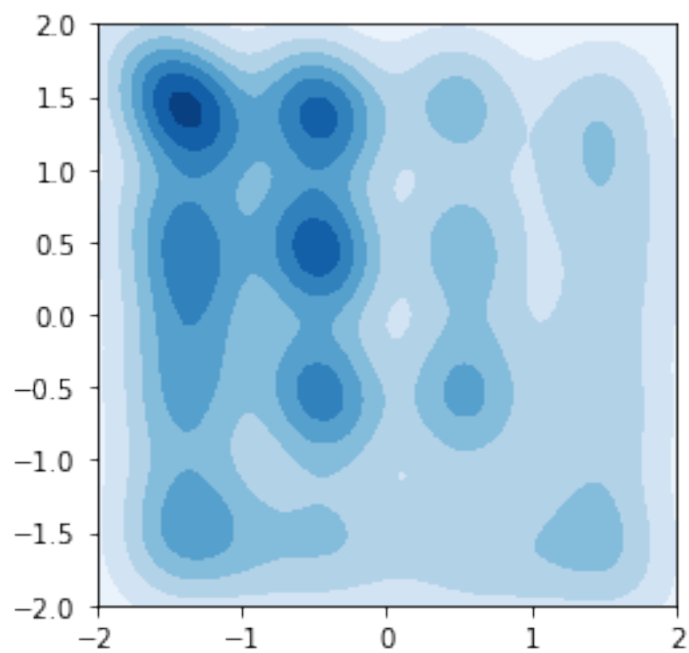
```
i = 0, discriminant loss = 1.4156, generator loss =0.6938
```



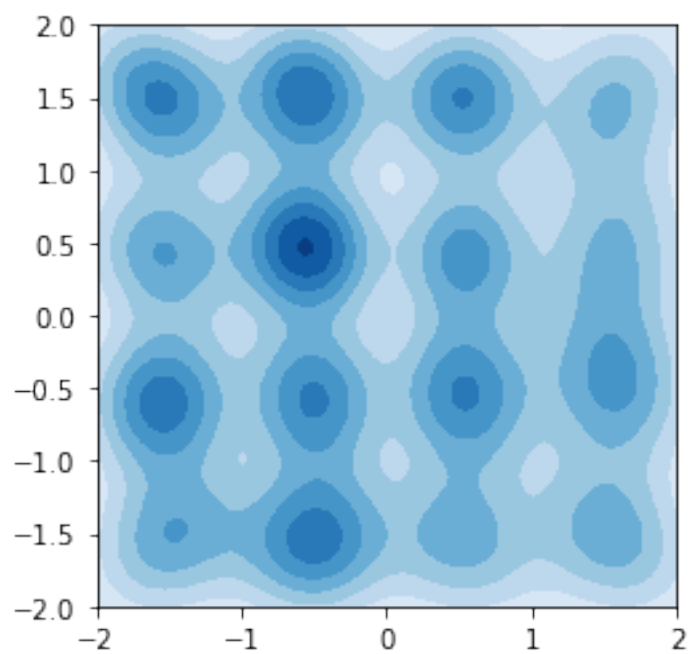
i = 2000, discriminant loss = 1.3443, generator loss =0.7208



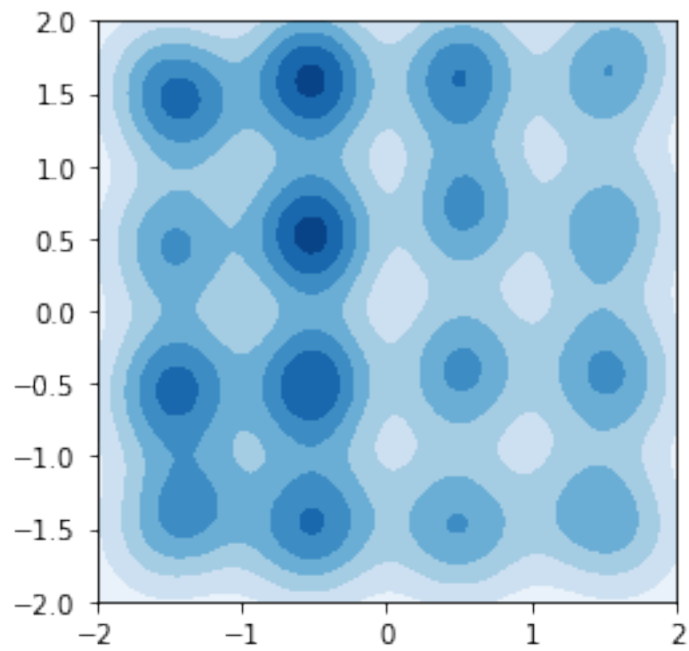
i = 4000, discriminant loss = 1.3732, generator loss =0.7006



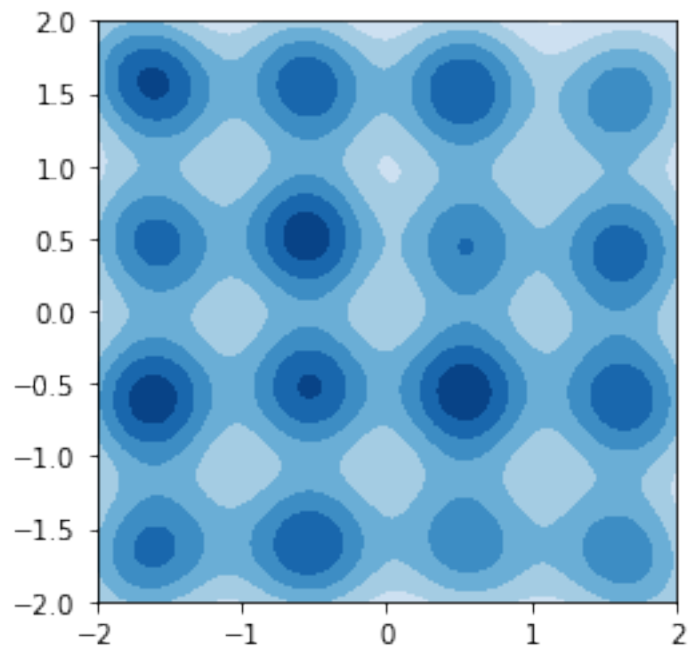
i = 6000, discriminant loss = 1.3827, generator loss =0.6935



i = 8000, discriminant loss = 1.3852, generator loss = 0.6940



i = 10000, discriminant loss = 1.3853, generator loss = 0.6939



1.7 Experiment 5: Optimistic Mirror Descent (OMD) algorithm implementation

```
In [0]: from tensorflow.python.ops import control_flow_ops
        from tensorflow.python.ops import math_ops
        from tensorflow.python.ops import state_ops
        from tensorflow.python.framework import ops
        from tensorflow.python.training import optimizer

class AdamirrorOptimizer(optimizer.Optimizer):

    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8,
                 use_locking=False, name="Adamirror"):

        super(AdamirrorOptimizer, self).__init__(use_locking, name)
        self._lr = learning_rate
        self._beta1 = beta1
        self._beta2 = beta2

        self._lr_t = None
        self._beta1_t = None
        self._beta2_t = None
        self._t = None

    def _prepare(self):
        self._lr_t = ops.convert_to_tensor(self._lr, name="learning_rate")
        self._beta1_t = ops.convert_to_tensor(self._beta1, name="beta1")
        self._beta2_t = ops.convert_to_tensor(self._beta2, name="beta2")
        self._t = ops.convert_to_tensor(0, name="t")

    def _create_slots(self, var_list):
        for v in var_list:
            self._zeros_slot(v, "m", self._name)
            self._zeros_slot(v, "v", self._name)
            self._zeros_slot(v, "g", self._name)

    def _apply_dense(self, grad, var):
        lr_t = math_ops.cast(self._lr_t, var.dtype.base_dtype)
        beta1_t = math_ops.cast(self._beta1_t, var.dtype.base_dtype)
        beta2_t = math_ops.cast(self._beta2_t, var.dtype.base_dtype)
        if var.dtype.base_dtype == tf.float16:
            eps = 1e-7
```

```

else:
    eps = 1e-8
    t = self._t

    t = t+1
    v = self.get_slot(var, "v")
    v_t = v.assign(beta2_t * v + (1. - beta2_t) * tf.square(grad))
    m = self.get_slot(var, "m")
    m_t = m.assign( beta1_t * m + (1. - beta1_t) * grad )
    v_t_hat = tf.div(v_t, 1. - tf.pow(beta2_t, t))
    m_t_hat = tf.div(m_t, 1. - tf.pow(beta1_t, t))

    g_t = tf.div( m_t_hat, tf.sqrt(v_t_hat)+eps )
    g_t_1 = self.get_slot(var, "g")
    g_t = g_t_1.assign( g_t )

    t_t = self._t.assign(t)

    var_update = state_ops.assign_sub(var, 2. * lr_t * g_t - lr_t * g_t_1) #Adam would
    return control_flow_ops.group(*[var_update, m_t, v_t, g_t, t_t])

def _apply_sparse(self, grad, var):
    raise NotImplementedError("Sparse gradient updates are not supported.")

```

```

In [0]: x = tf.constant([[2, 2.0], [3, 3]])
        y = tf.constant([[8, 16], [2, 3]])
        print(tf.pow(x, y)) # [[256, 65536], [9, 27]]

```

```

-----

ValueError                                Traceback (most recent call last)

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py in
526             as_ref=input_arg.is_ref,
--> 527             preferred_dtype=default_dtype)
528         except TypeError as err:

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/ops.py in internal_
1223     if ret is None:
-> 1224         ret = conversion_func(value, dtype=dtype, name=name, as_ref=as_ref)
1225

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/ops.py in _TensorTer
1017     "Tensor conversion requested dtype %s for Tensor with dtype %s: %r" %
-> 1018     (dtype.name, t.dtype.name, str(t)))

```

```
1019     return t
```

ValueError: Tensor conversion requested dtype float32 for Tensor with dtype int32: 'Tensor'

During handling of the above exception, another exception occurred:

```
TypeError                                Traceback (most recent call last)

<ipython-input-29-dce0577ce282> in <module>()
      1 x = tf.constant([[2, 2.0], [3, 3]])
      2 y = tf.constant([[8, 16], [2, 3]])
----> 3 print(tf.pow(x, y)) # [[256, 65536], [9, 27]]

/usr/local/lib/python3.6/dist-packages/tensorflow/python/util/dispatch.py in wrapper(*args, **kwargs)
    178     """Call target, and fall back on dispatchers if there is a TypeError."""
    179     try:
--> 180         return target(*args, **kwargs)
    181     except (TypeError, ValueError):
    182         # Note: convert_to_eager_tensor currently raises a ValueError, not a

/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py in pow(x, y, name)
    448     """
    449     with ops.name_scope(name, "Pow", [x]) as name:
--> 450         return gen_math_ops._pow(x, y, name=name)
    451
    452

/usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/gen_math_ops.py in _pow(x, y, name)
    6970     # Add nodes to the TensorFlow graph.
    6971     _, _, _op = _op_def_lib._apply_op_helper(
-> 6972         "Pow", x=x, y=y, name=name)
    6973     _result = _op.outputs[:]
    6974     _inputs_flat = _op.inputs

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py in _apply_op_helper(op_name, prefix, dtype, attrs, inputs, kwargs)
    561         "%s type %s of argument '%s'." %
    562         (prefix, dtypes.as_dtype(attrs[input_arg.type_attr]).name,
--> 563         inferred_from[input_arg.type_attr]))
    564
    565         types = [values.dtype]
```

TypeError: Input 'y' of 'Pow' Op has type int32 that does not match type float32 of arg

1.8 Experiment 6: Comparison of learning rate threshold of OMD and SGA in Gaussian mixture model

```
In [0]: def reset_and_build_graph_with_lr(depth, width, x_real_builder, z_dim, batch_size, learning_rate):
    tf.reset_default_graph()

    x_real = x_real_builder(batch_size)
    x_dim = x_real.get_shape().as_list()[1]
    generator = MLP(depth, width, x_dim, 'generator')
    discriminator = MLP(depth, width, 1, 'discriminator')
    z = tf.random_normal([batch_size, z_dim])
    x_fake = generator(z)
    disc_out_real = discriminator(x_real)
    disc_out_fake = discriminator(x_fake)

    # Loss
    disc_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=disc_out_real, labels=tf.ones_like(disc_out_real)))
    disc_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=disc_out_fake, labels=tf.zeros_like(disc_out_fake)))
    disc_loss = disc_loss_real + disc_loss_fake

    gen_loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(
            logits=disc_out_fake, labels=tf.ones_like(disc_out_fake)))
    gen_vars = generator.variable_scope.trainable_variables()
    disc_vars = discriminator.variable_scope.trainable_variables()
    # Compute gradients
    xs = disc_vars + gen_vars
    disc_grads = tf.gradients(disc_loss, disc_vars)
    gen_grads = tf.gradients(gen_loss, gen_vars)
    Xi = disc_grads + gen_grads
    apply_vec = list(zip(Xi, xs))

    if mode == 'RMS':
        optimizer = tf.train.RMSPropOptimizer(learning_rate)
    elif mode == 'OMD':
        optimizer = AdamirrorOptimizer(learning_rate)
    elif mode == 'SGA':
        optimizer = SymplecticOptimizer(learning_rate)
    else:
        raise ValueError('Mode %s not recognised' % mode)
```

```

with tf.control_dependencies([g for (g, v) in apply_vec]):
    train_op = optimizer.apply_gradients(apply_vec)

init = tf.global_variables_initializer()

return train_op, x_fake, z, init, disc_loss, gen_loss

In [0]: def learn_mixture_of_gaussians_with_lr(mode, learning_rate):
    print(mode)
    def x_real_builder(batch_size):
        sigma = 0.1
        skel = np.array([
            [ 1.50,  1.50],
            [ 1.50,  0.50],
            [ 1.50, -0.50],
            [ 1.50, -1.50],
            [ 0.50,  1.50],
            [ 0.50,  0.50],
            [ 0.50, -0.50],
            [ 0.50, -1.50],
            [-1.50,  1.50],
            [-1.50,  0.50],
            [-1.50, -0.50],
            [-1.50, -1.50],
            [-0.50,  1.50],
            [-0.50,  0.50],
            [-0.50, -0.50],
            [-0.50, -1.50],
        ])
        temp = np.tile(skel, (batch_size // 16 + 1, 1))
        mus = temp[0:batch_size,:]
        return mus + sigma*tf.random_normal([batch_size, 2])*0.2

    z_dim = 64
    train_op, x_fake, z, init, disc_loss, gen_loss = reset_and_build_graph_with_lr(
        depth=6, width=384, x_real_builder=x_real_builder, z_dim=z_dim,
        batch_size=256, learning_rate=learning_rate, mode=mode)

    train(train_op, x_fake, z, init, disc_loss, gen_loss, z_dim)

In [0]: lrs = [1e-5, 5e-5, 1e-4, 5e-4, 1e-3]

```

1.8.1 Using OMD

```

In [0]: for lr in lrs:
    learn_mixture_of_gaussians_with_lr('OMD', lr)

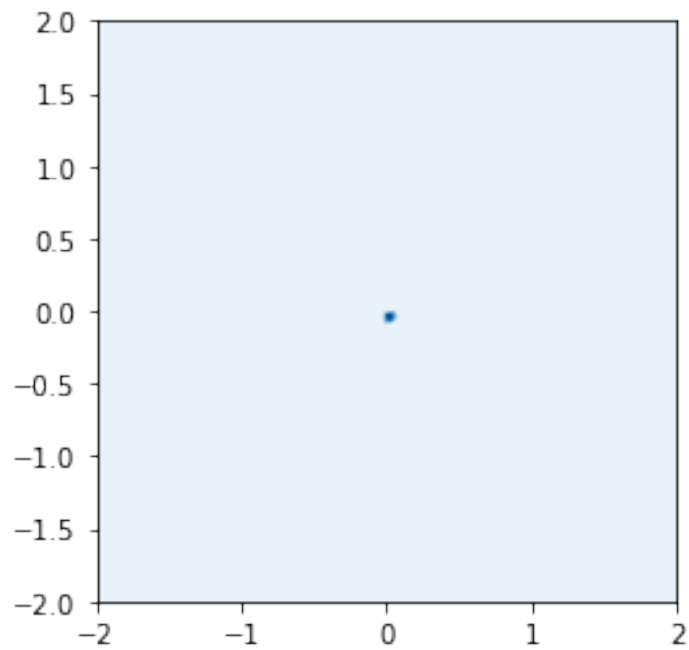
```

1.8.2 Using SGA

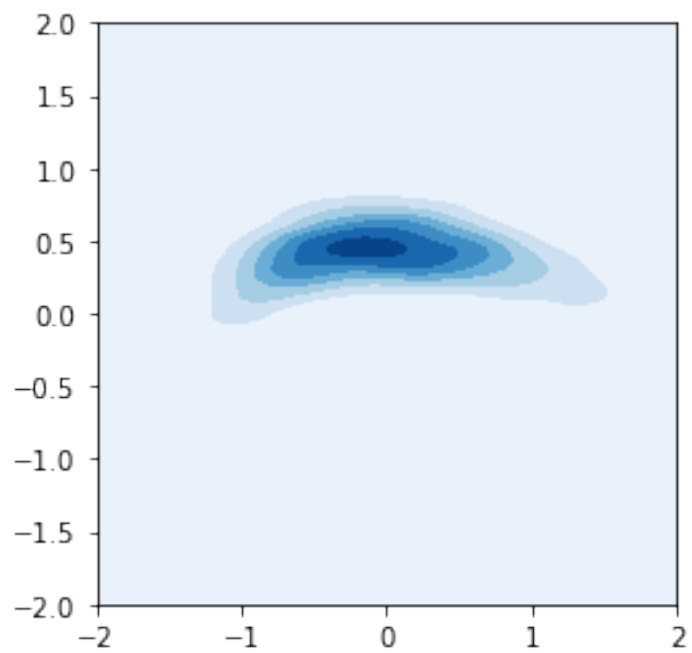
```
In [0]: for lr in lrs:
        learn_mixture_of_gaussians_with_lr('SGA', lr)
```

SGA

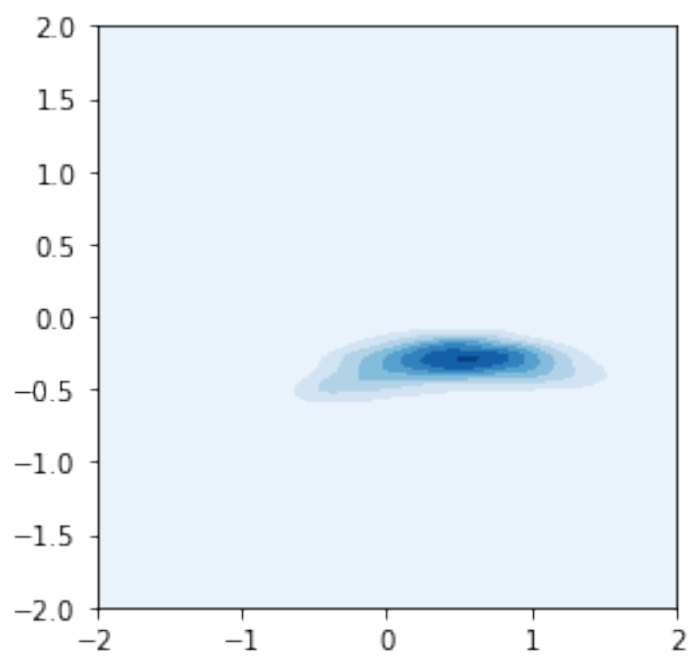
i = 0, discriminant loss = 1.3660, generator loss =0.6924



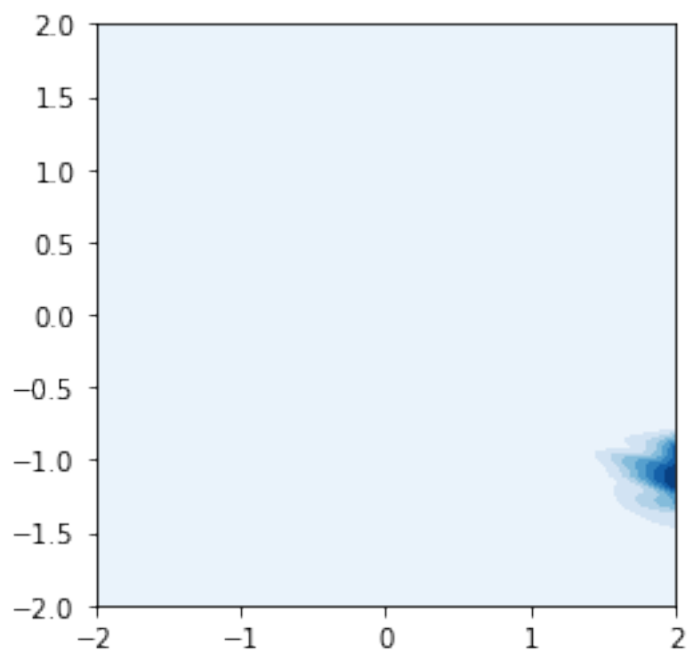
i = 2000, discriminant loss = 0.9693, generator loss =0.7837



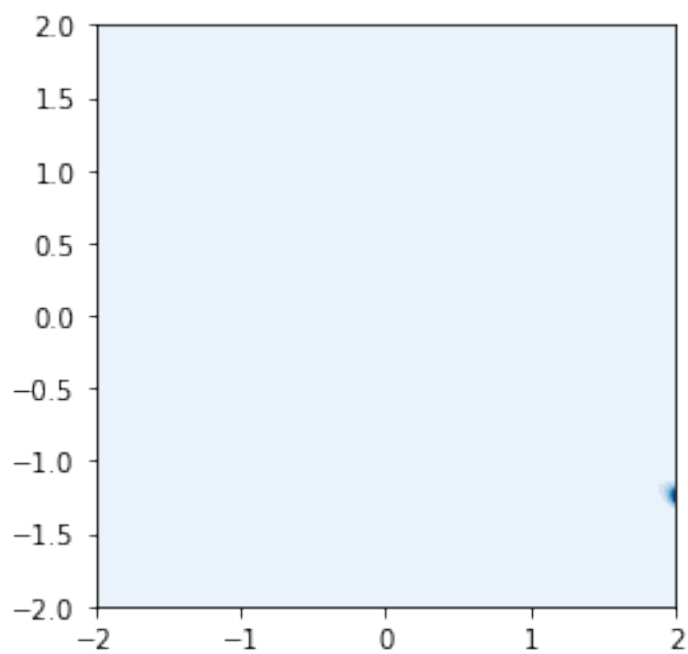
i = 4000, discriminant loss = 5.6121, generator loss =0.0429



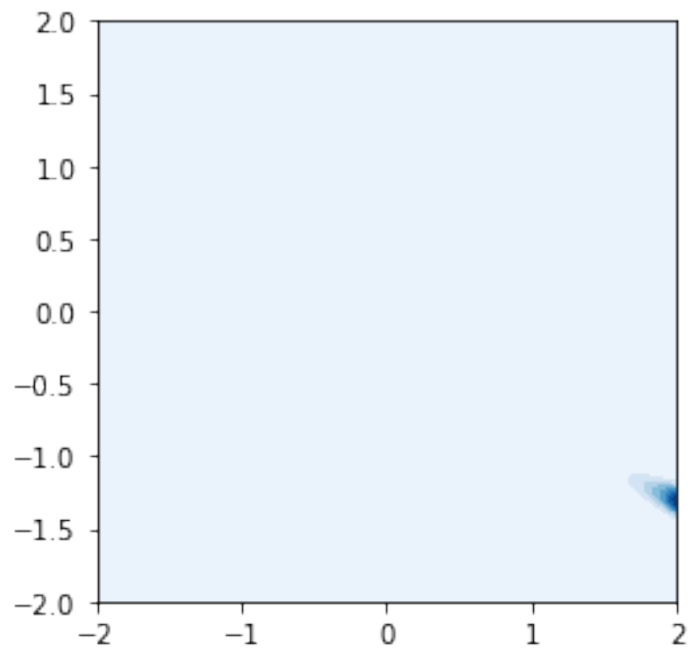
i = 6000, discriminant loss = 5.6527, generator loss =0.3095



i = 8000, discriminant loss = 49.1593, generator loss =1.2838

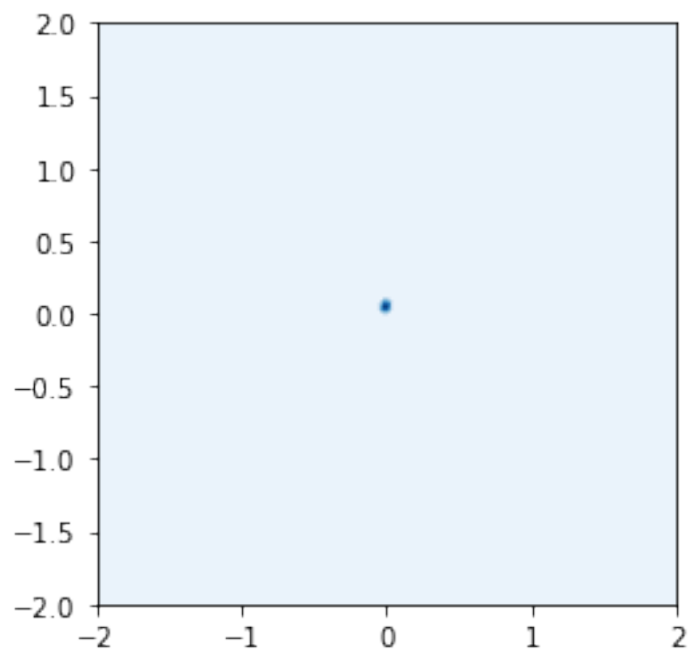


i = 10000, discriminant loss = 294.4630, generator loss =12.8131

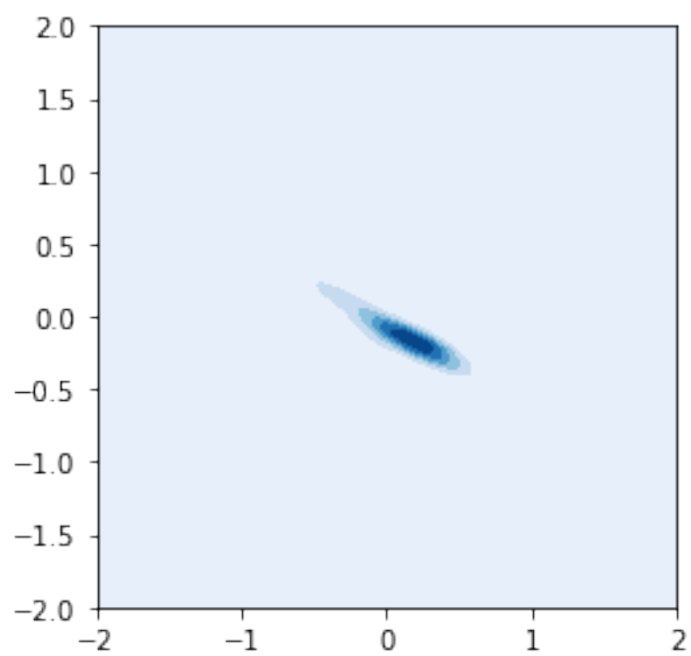


SGA

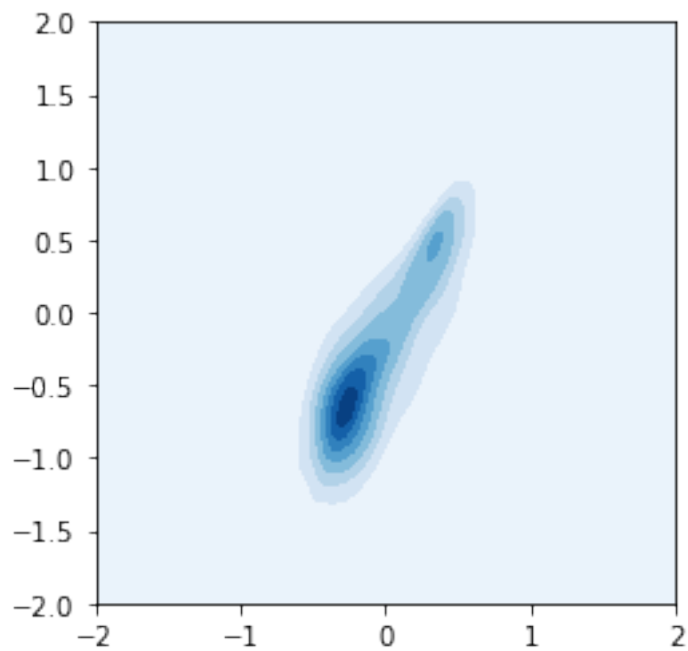
i = 0, discriminant loss = 1.3483, generator loss =0.6920



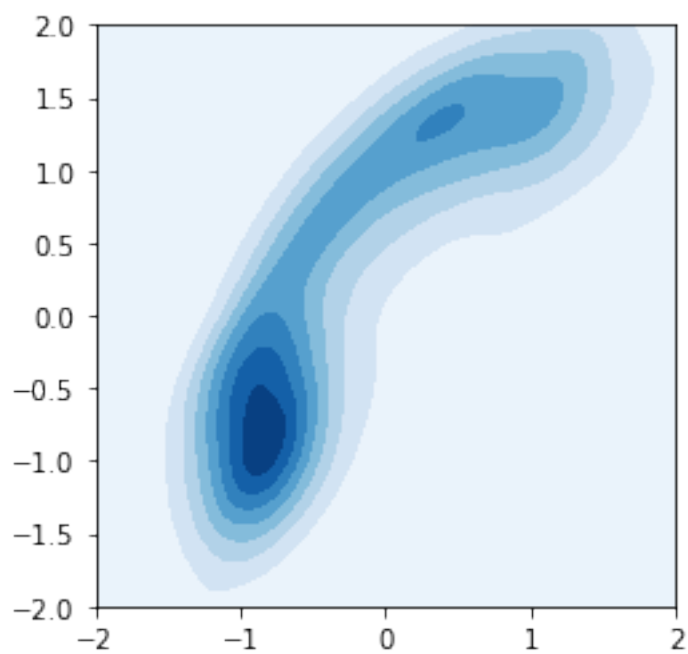
i = 2000, discriminant loss = 0.8996, generator loss = 0.7104



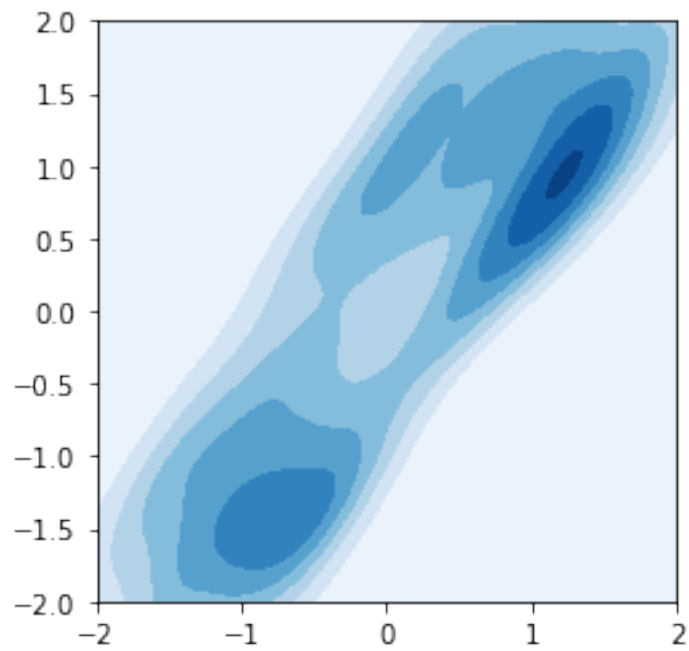
i = 4000, discriminant loss = 0.6597, generator loss =1.3435



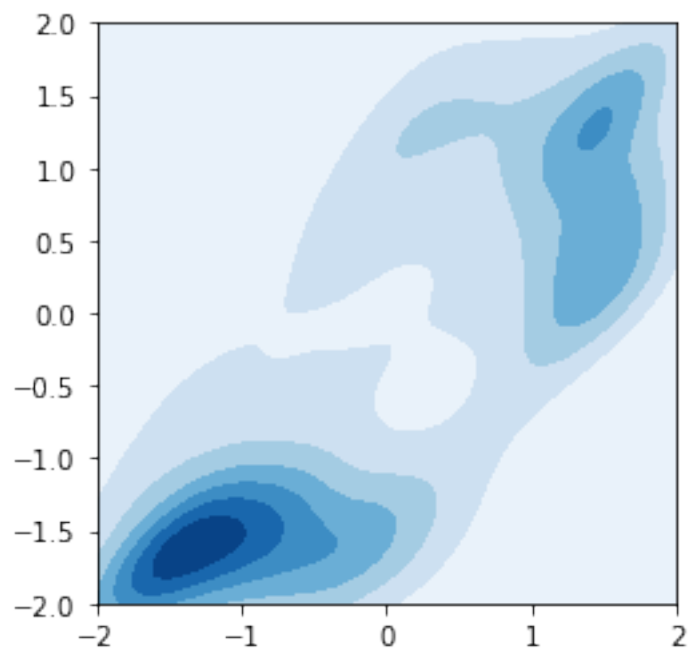
i = 6000, discriminant loss = 1.1008, generator loss =0.9895



i = 8000, discriminant loss = 0.9388, generator loss =1.0736

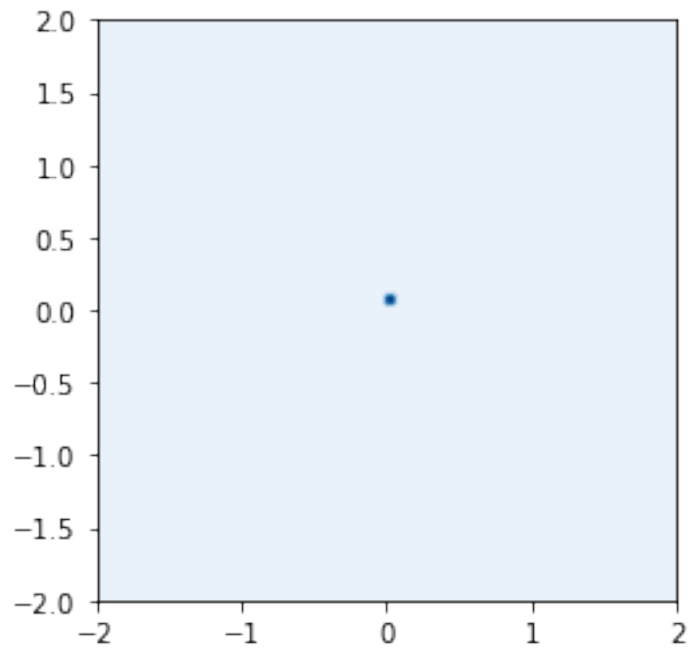


i = 10000, discriminant loss = 0.9927, generator loss =1.1249

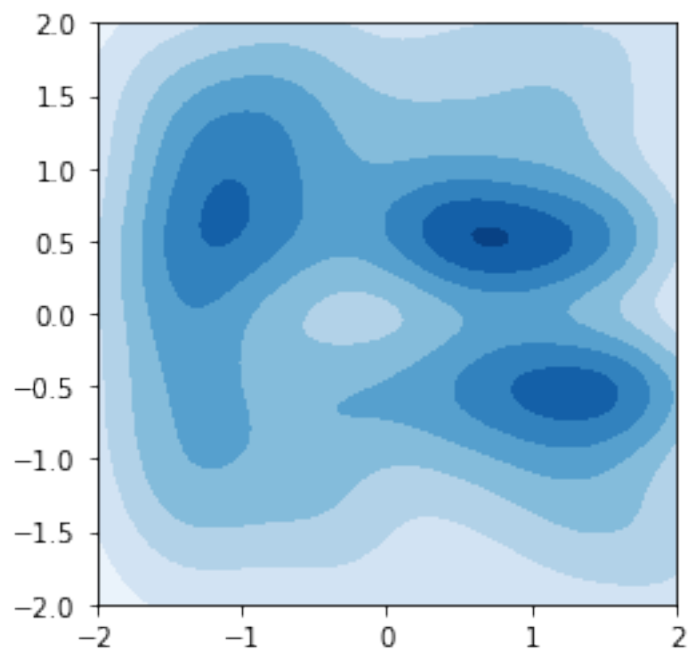


SGA

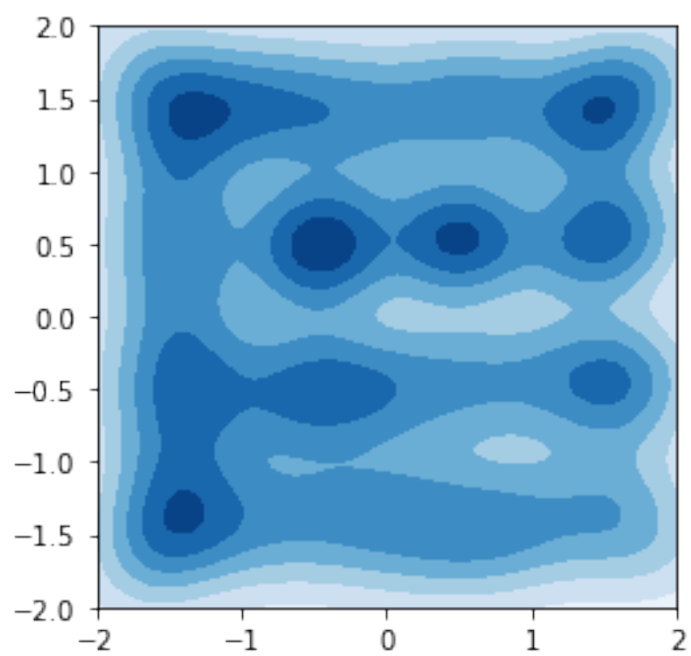
i = 0, discriminant loss = 1.3930, generator loss = 0.6940



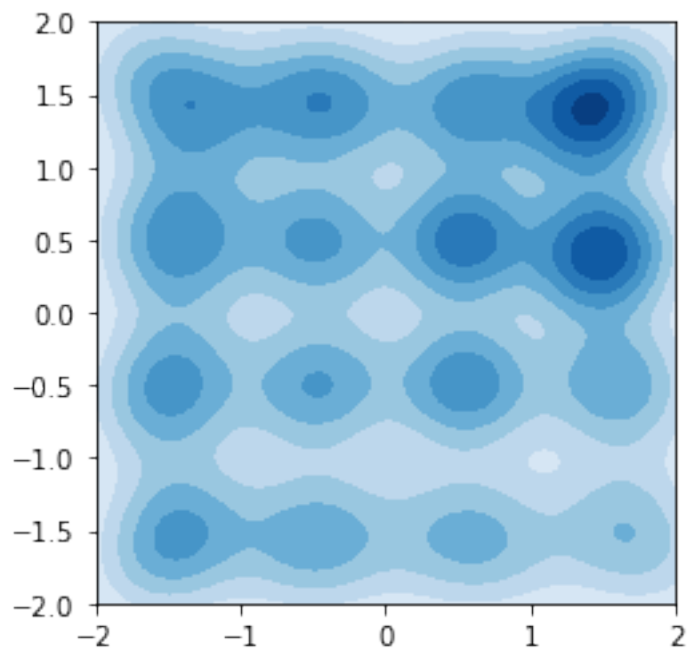
i = 2000, discriminant loss = 1.3426, generator loss = 0.7182



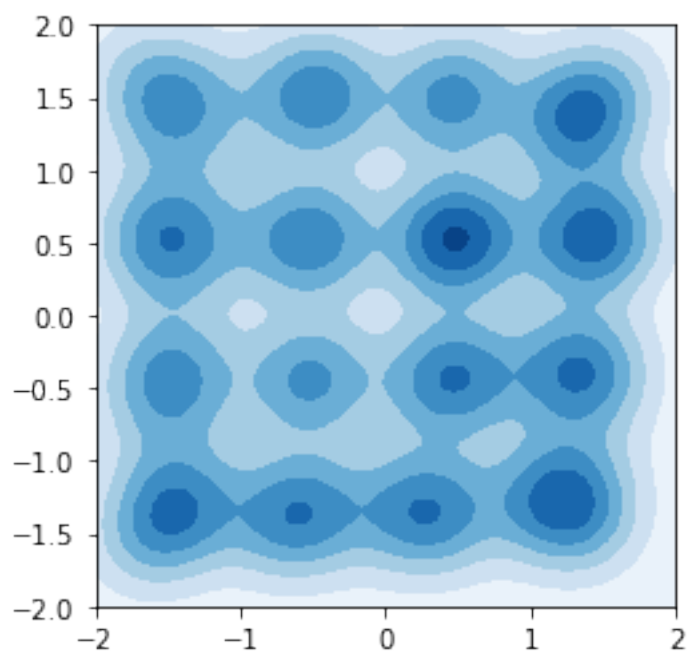
i = 4000, discriminant loss = 1.3656, generator loss = 0.7059



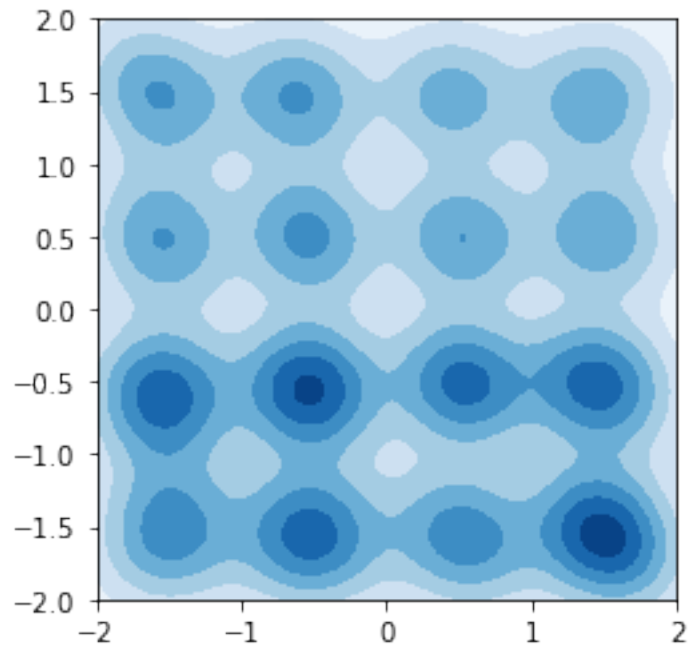
i = 6000, discriminator loss = 1.3823, generator loss = 0.6944



i = 8000, discriminator loss = 1.3850, generator loss = 0.6939

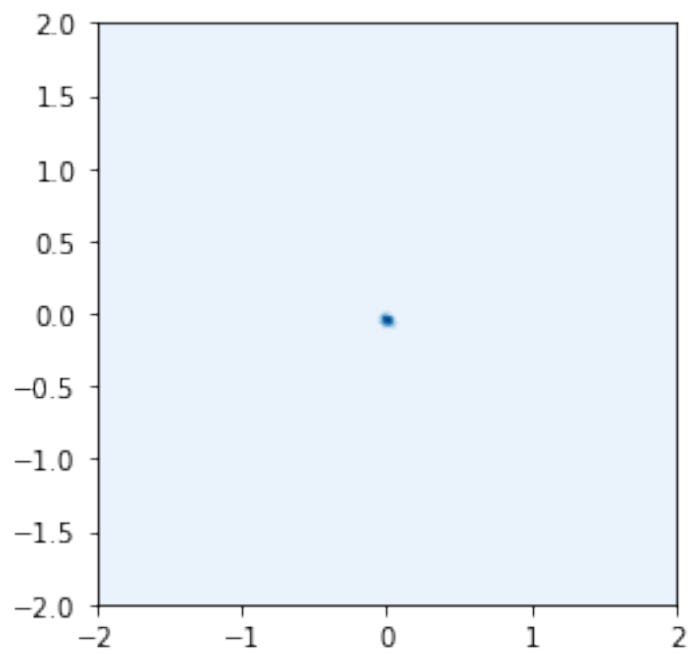


i = 10000, discriminant loss = 1.3854, generator loss =0.6939

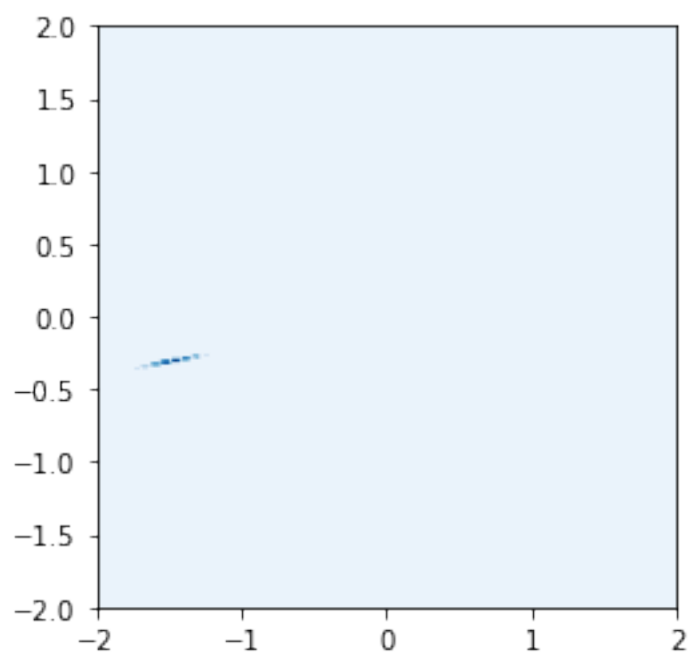


SGA

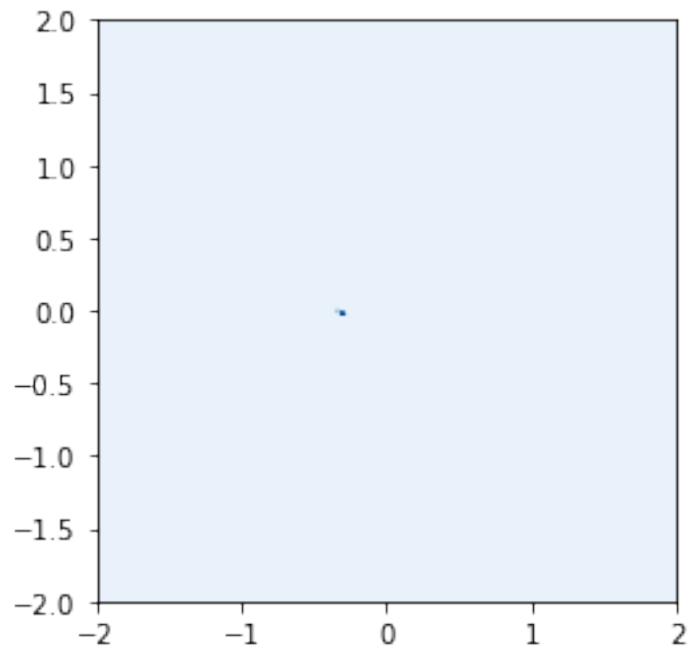
i = 0, discriminant loss = 1.3845, generator loss =0.6932



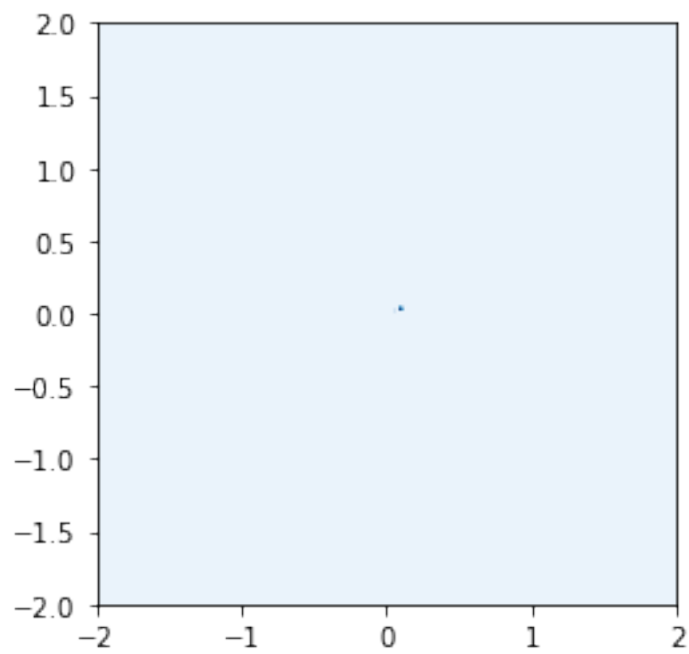
i = 2000, discriminant loss = 0.2664, generator loss =1.9103



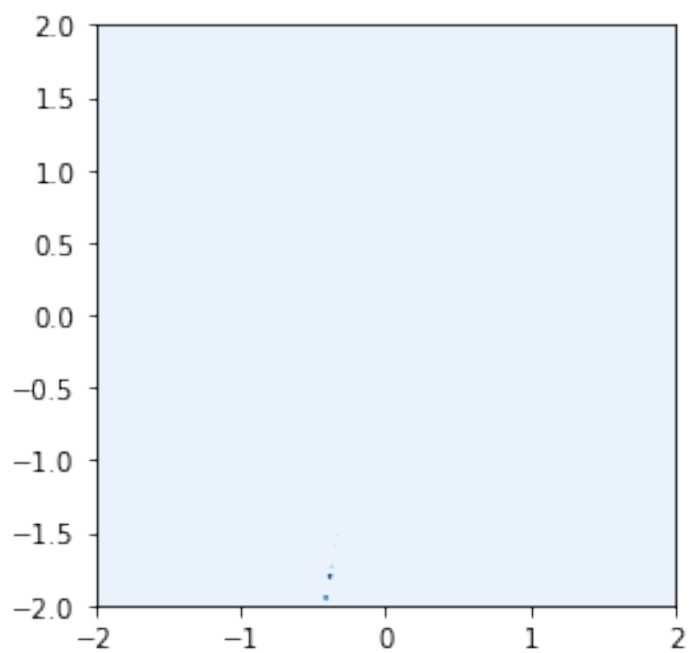
i = 4000, discriminant loss = 0.1573, generator loss =2.3711



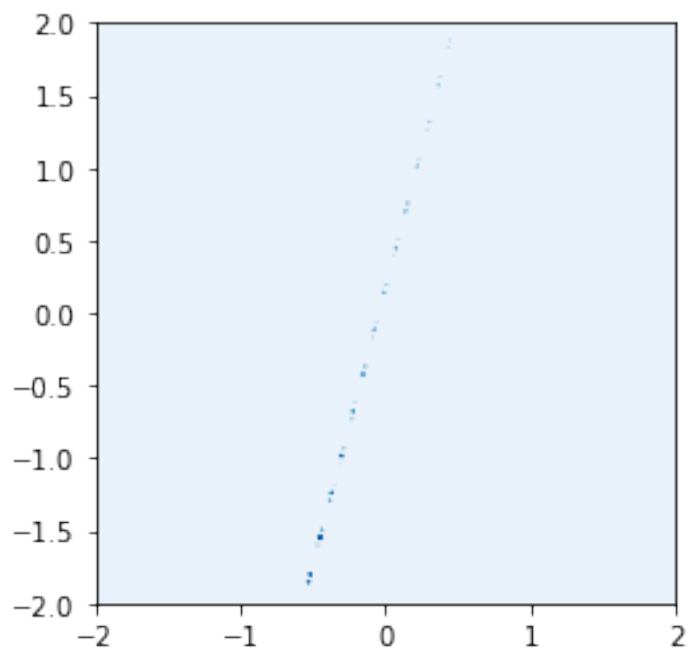
i = 6000, discriminant loss = 0.1241, generator loss =2.2053



i = 8000, discriminant loss = 0.0150, generator loss =4.2232

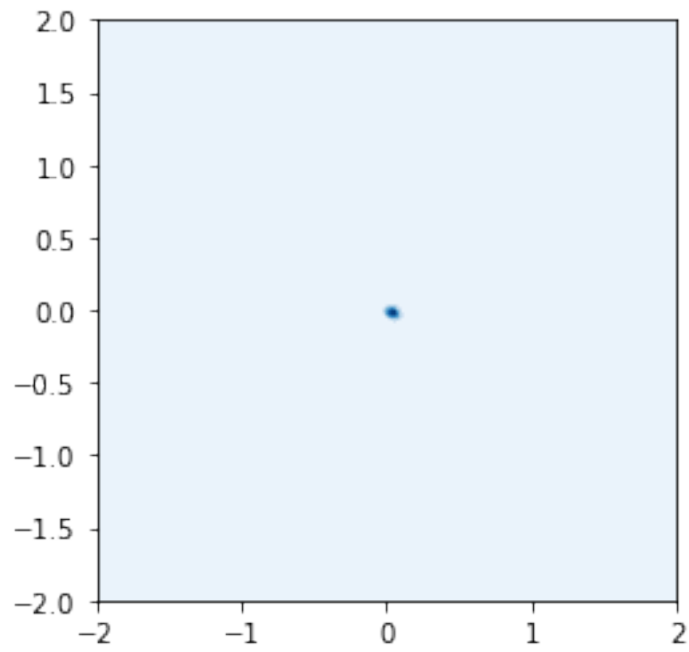


i = 10000, discriminant loss = 0.0006, generator loss =7.6488

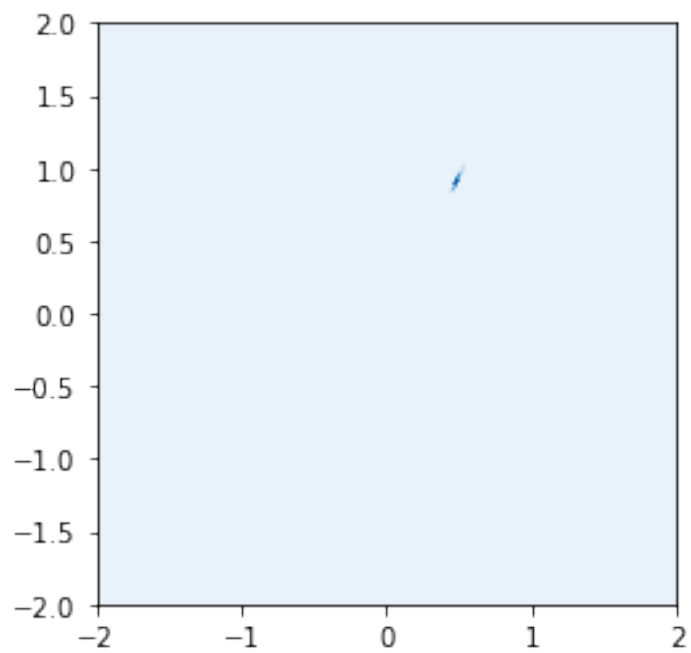


SGA

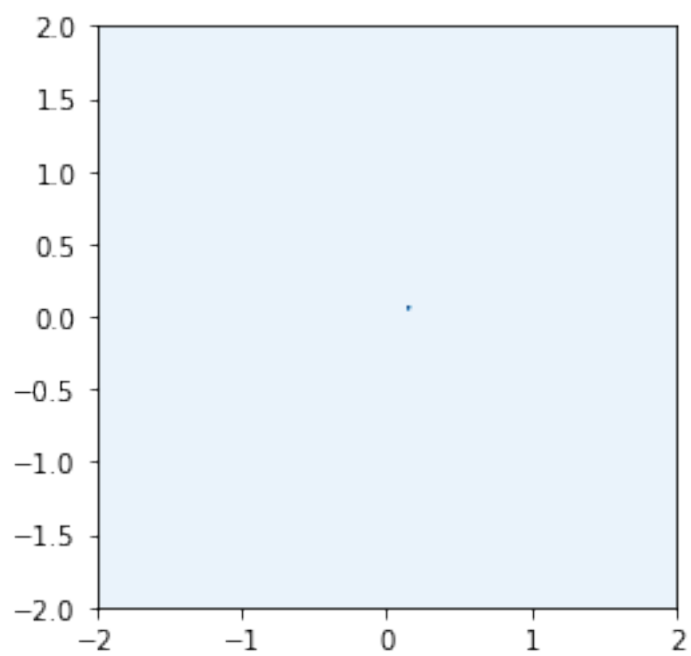
i = 0, discriminant loss = 1.3746, generator loss = 0.6926



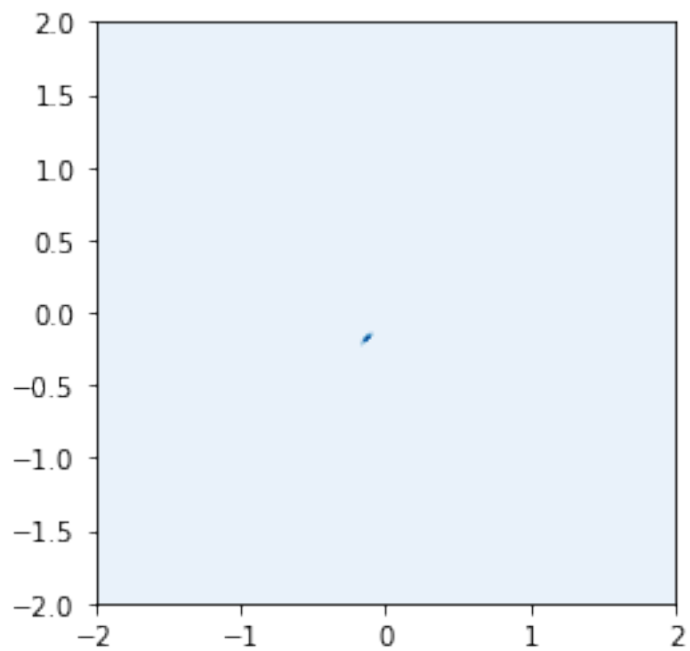
i = 2000, discriminant loss = 0.1350, generator loss = 2.0975



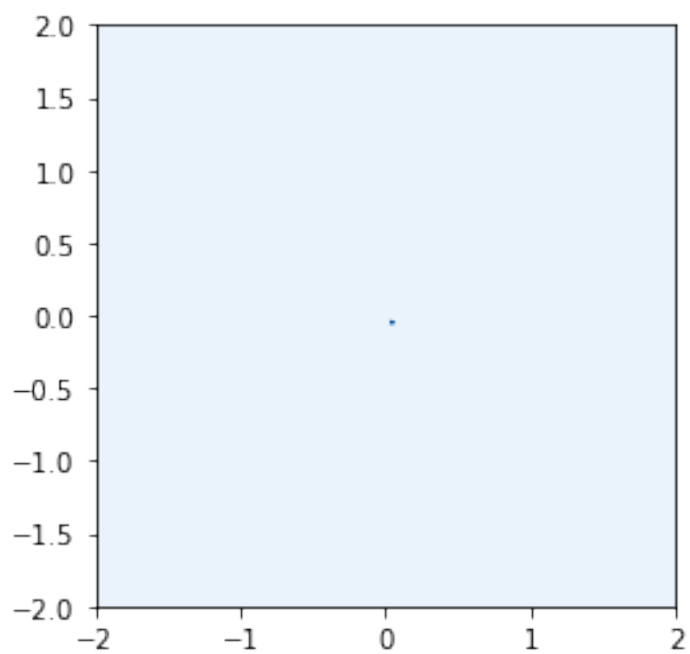
i = 4000, discriminant loss = 0.0353, generator loss =3.3729



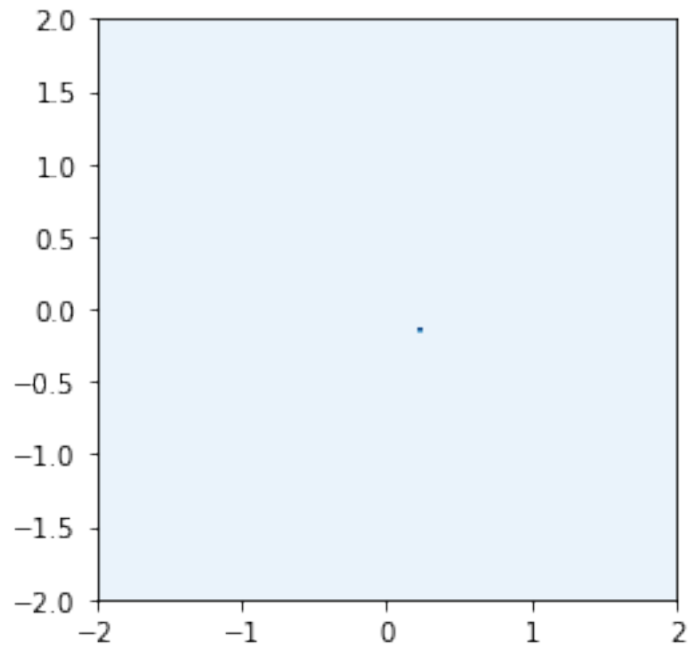
i = 6000, discriminant loss = 0.4416, generator loss =2.4975



i = 8000, discriminant loss = 0.0646, generator loss =2.8549



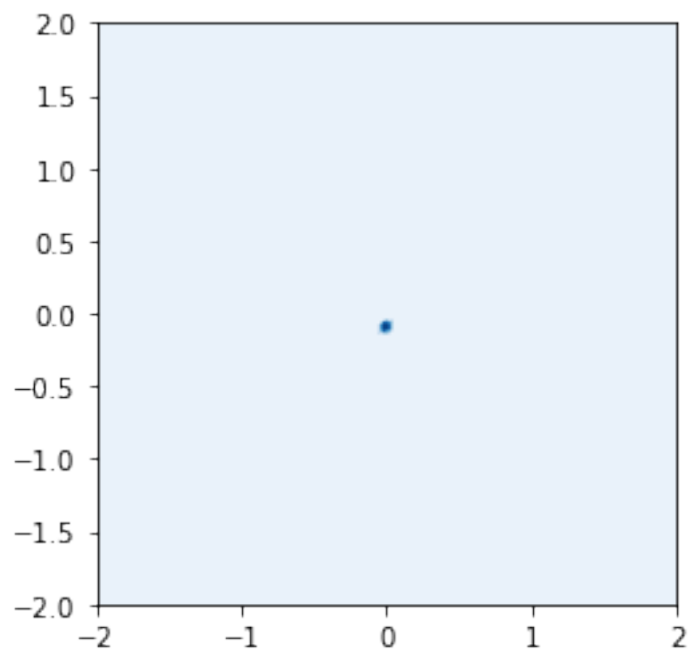
```
i = 10000, discriminant loss = 67803.4141, generator loss =0.0000
```



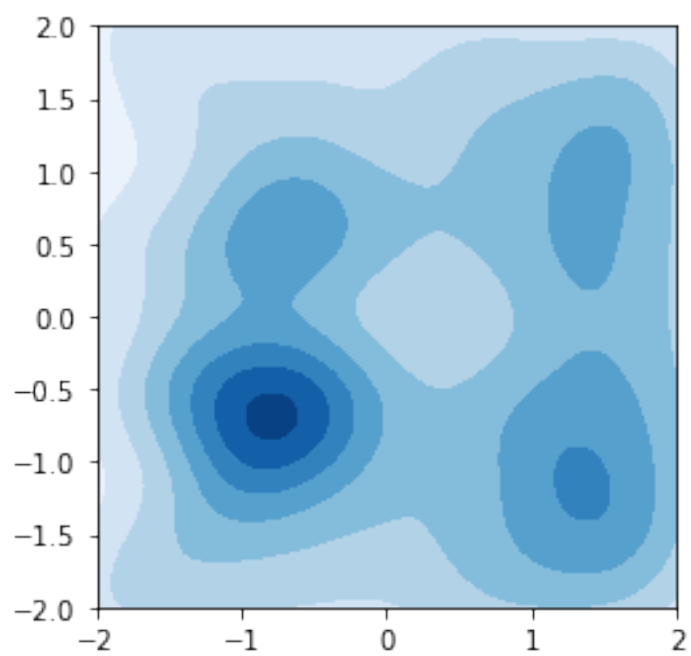
```
In [0]: lrs = [8e-5, 1e-4, 1.2e-4, 1.4e-4, 1.6e-4]
        for lr in lrs:
            learn_mixture_of_gaussians_with_lr('SGA', lr)
```

SGA

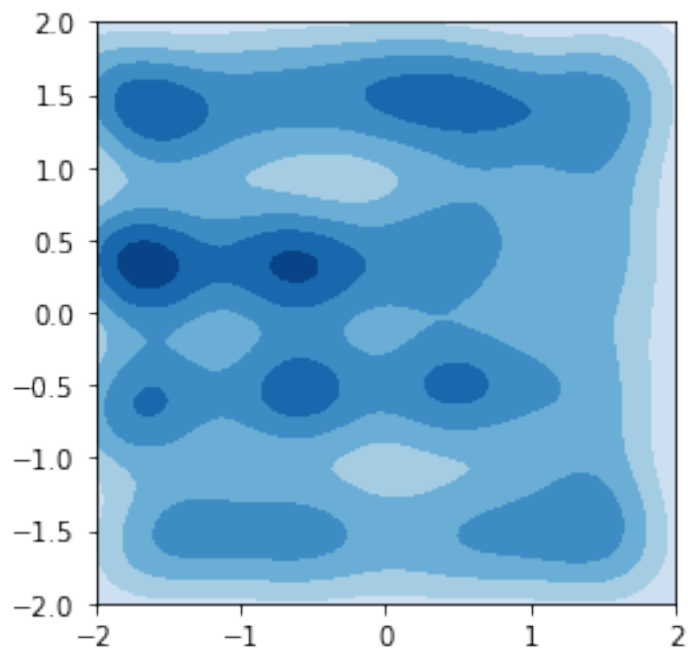
```
i = 0, discriminant loss = 1.3353, generator loss =0.6883
```



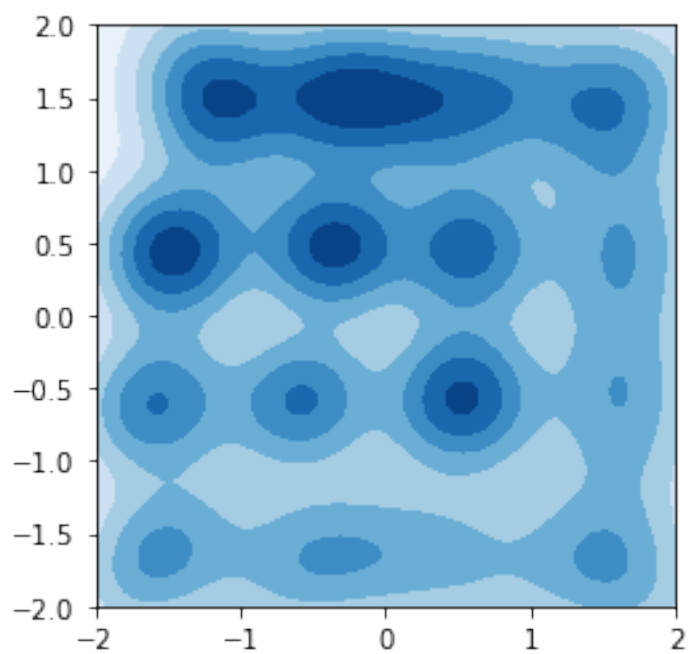
i = 2000, discriminant loss = 1.3481, generator loss = 0.7221



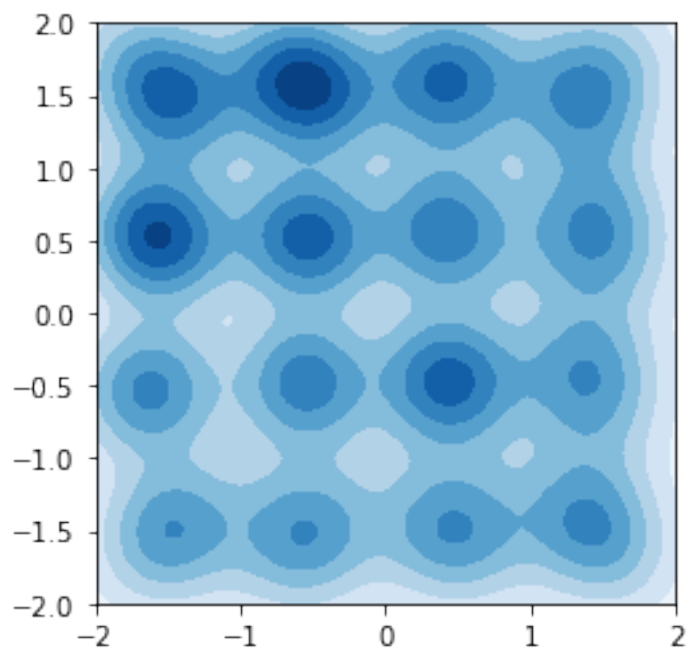
i = 4000, discriminant loss = 1.3642, generator loss =0.7061



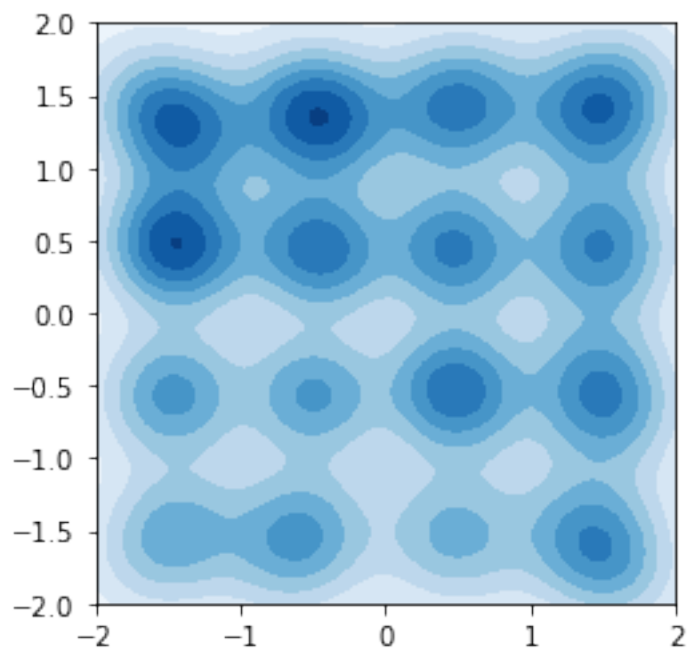
i = 6000, discriminant loss = 1.3772, generator loss =0.6963



i = 8000, discriminant loss = 1.3843, generator loss = 0.6940

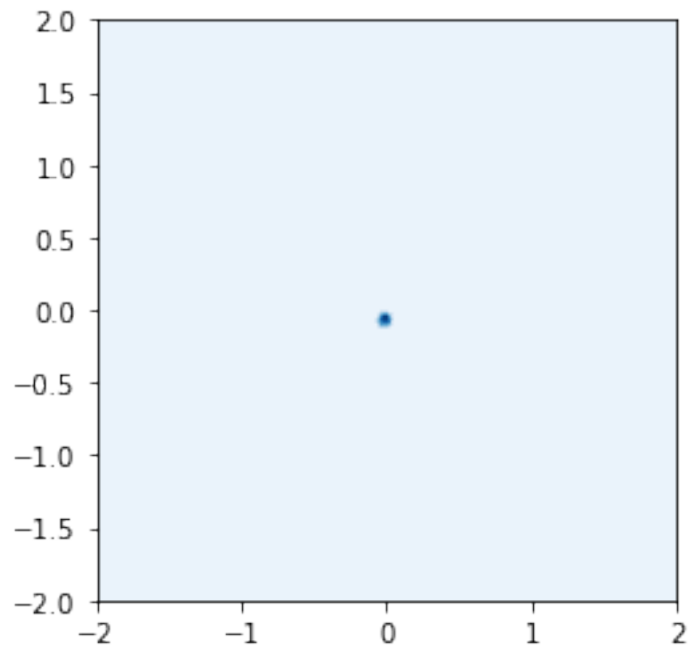


i = 10000, discriminant loss = 1.3854, generator loss = 0.6938

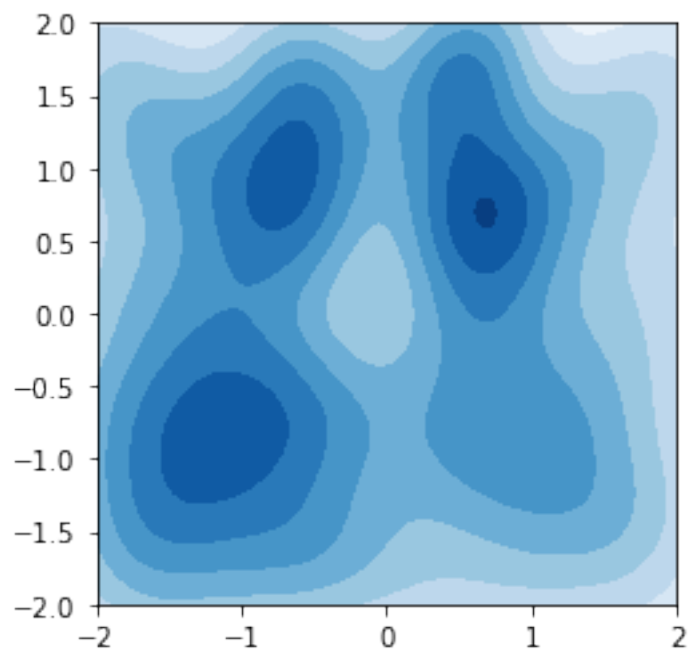


SGA

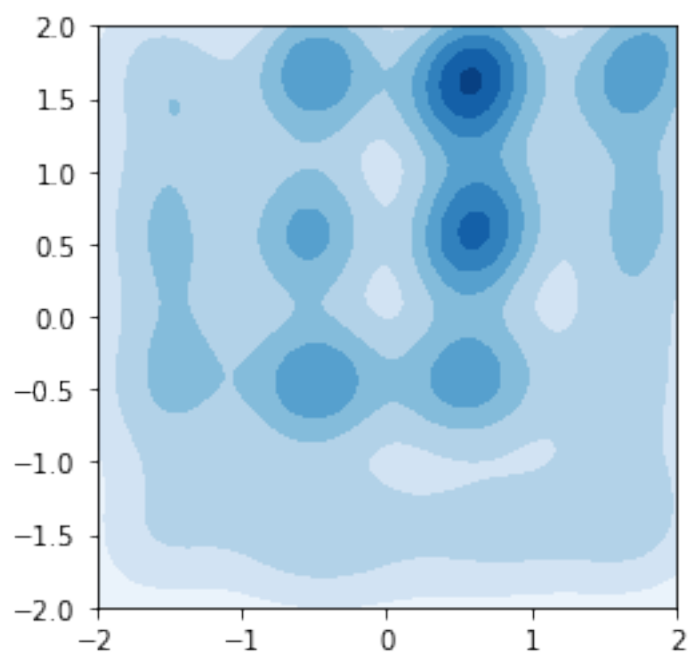
i = 0, discriminant loss = 1.3819, generator loss = 0.6926



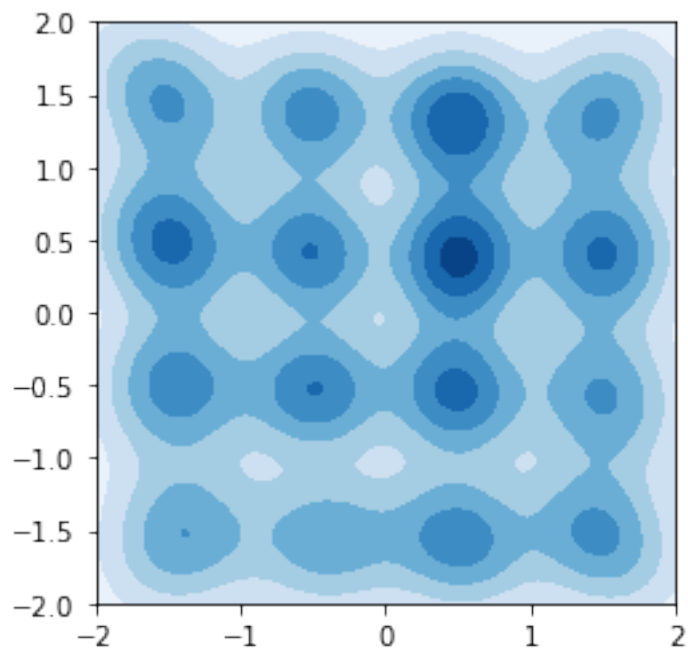
i = 2000, discriminant loss = 1.3461, generator loss = 0.7409



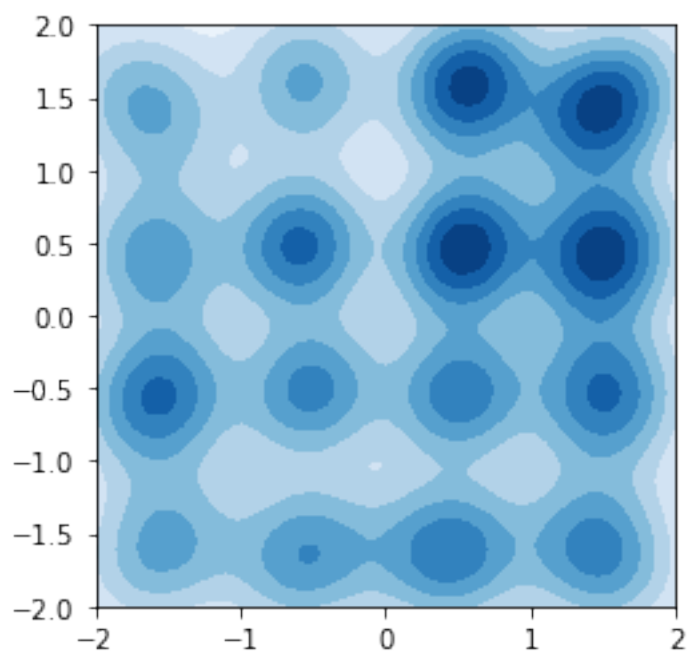
i = 4000, discriminant loss = 1.3761, generator loss =0.6976



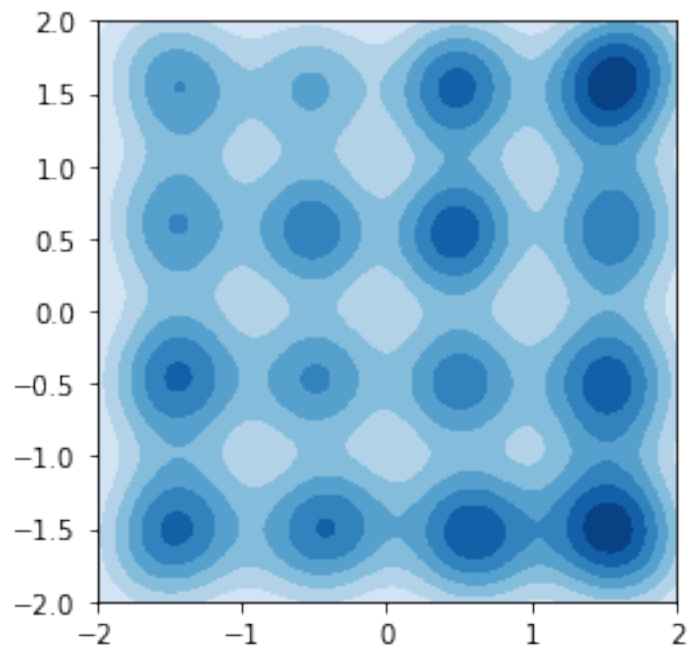
i = 6000, discriminant loss = 1.3846, generator loss = 0.6938



i = 8000, discriminant loss = 1.3851, generator loss = 0.6940

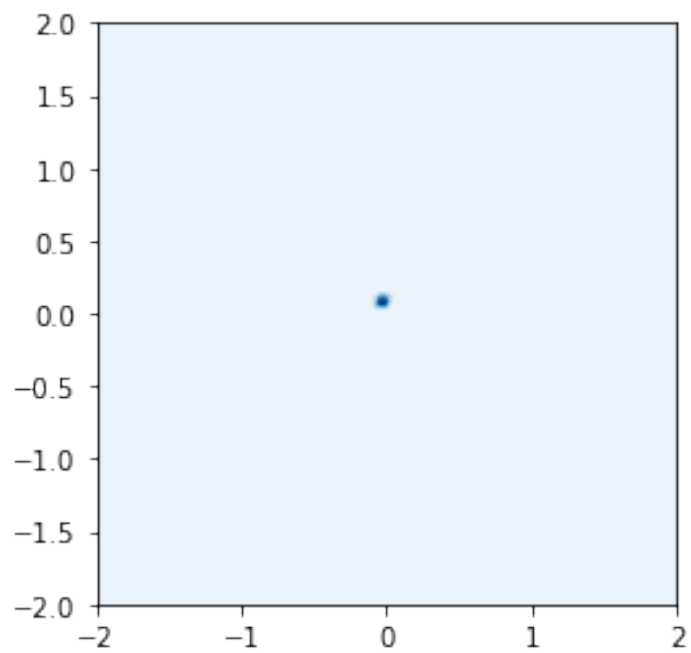


i = 10000, discriminant loss = 1.3854, generator loss = 0.6940

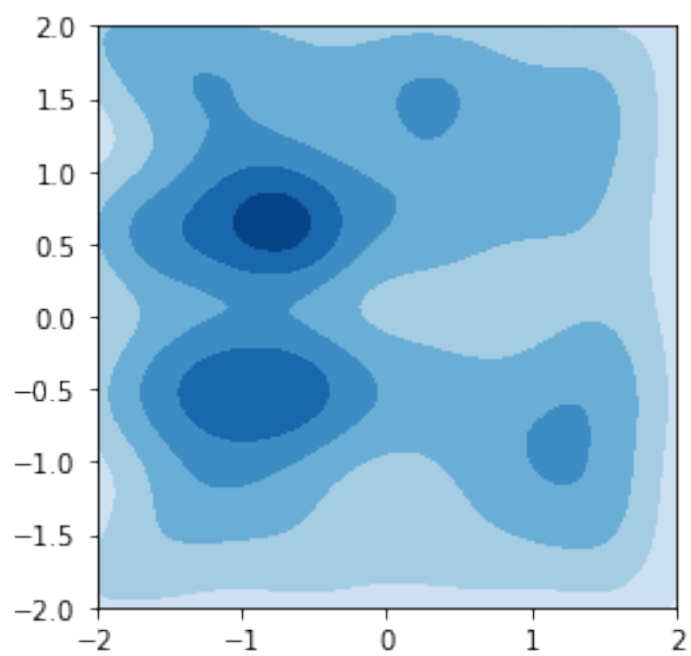


SGA

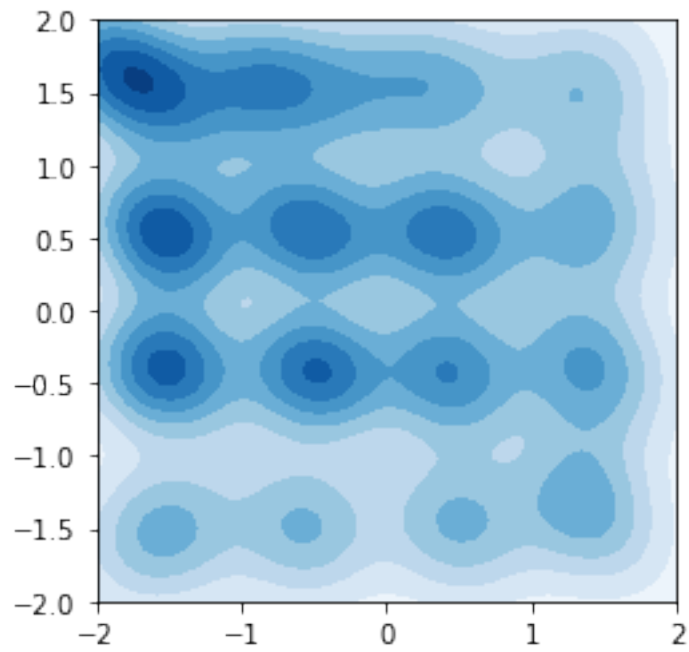
i = 0, discriminant loss = 1.3610, generator loss = 0.6920



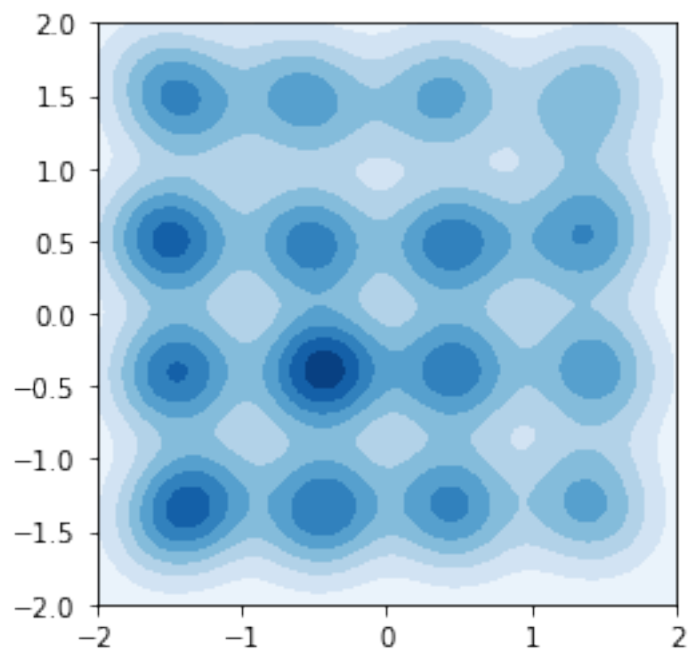
i = 2000, discriminant loss = 1.3472, generator loss = 0.7152



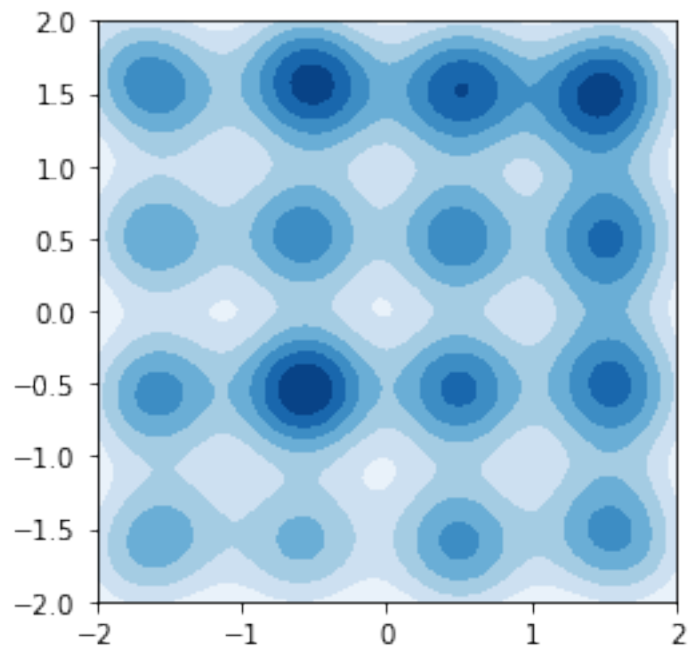
i = 4000, discriminant loss = 1.3788, generator loss = 0.6979



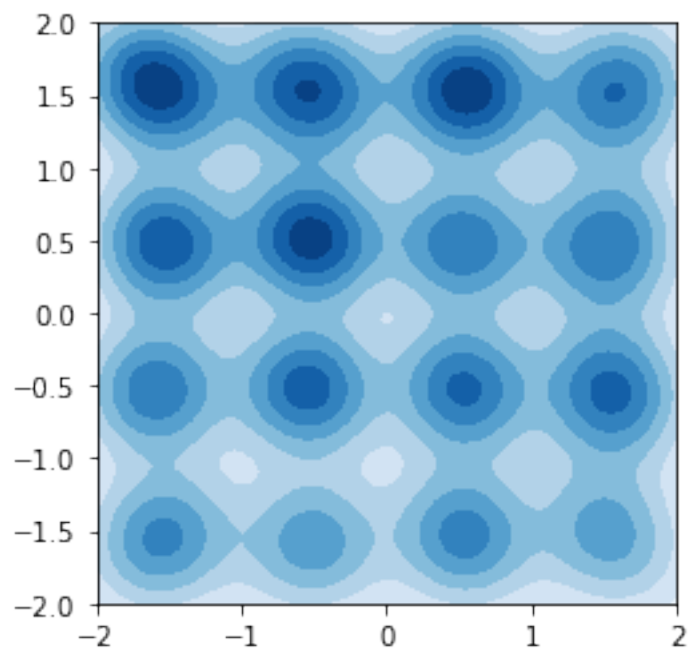
i = 6000, discriminant loss = 1.3850, generator loss = 0.6937



i = 8000, discriminant loss = 1.3852, generator loss = 0.6936

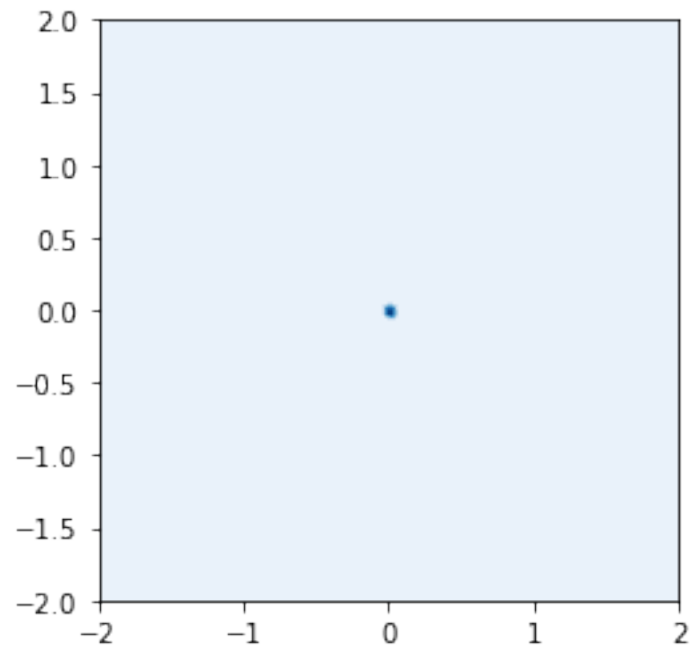


i = 10000, discriminant loss = 1.3857, generator loss = 0.6931

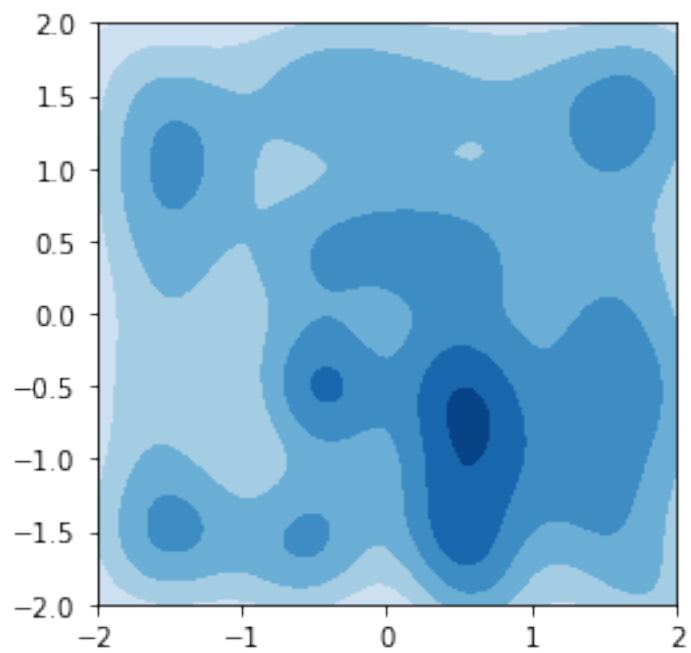


SGA

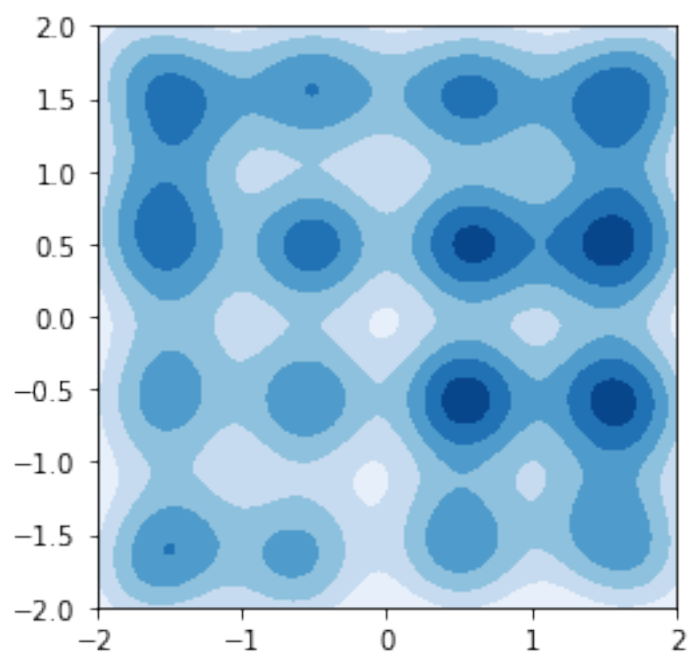
i = 0, discriminant loss = 1.3879, generator loss = 0.6932



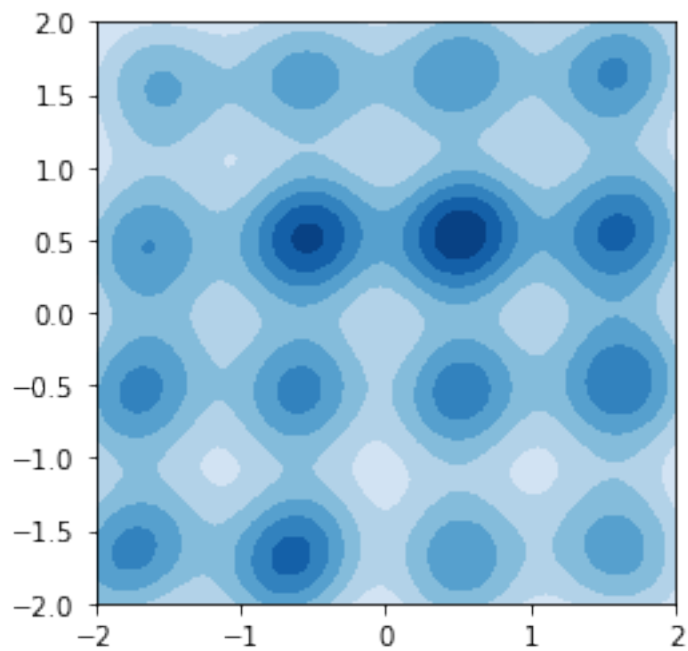
i = 2000, discriminant loss = 1.3501, generator loss = 0.7214



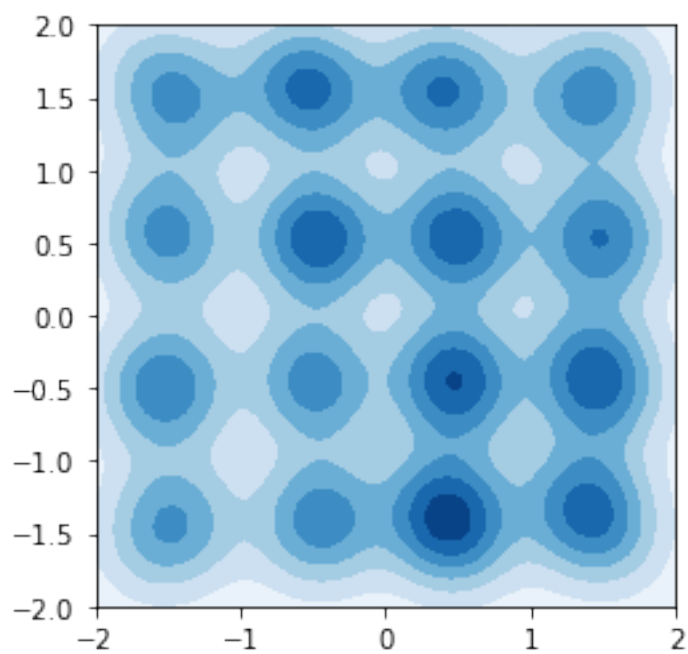
i = 4000, discriminant loss = 1.3834, generator loss = 0.6938



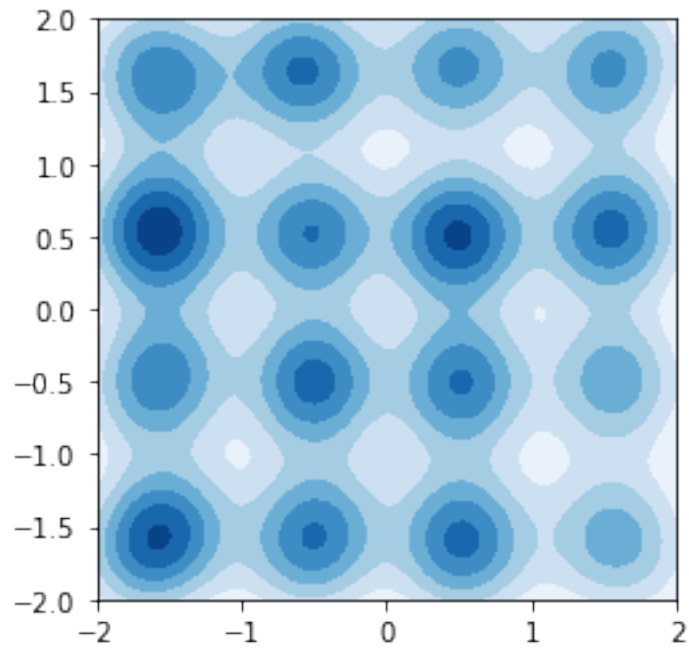
i = 6000, discriminator loss = 1.3856, generator loss = 0.6934



i = 8000, discriminator loss = 1.3858, generator loss = 0.6933

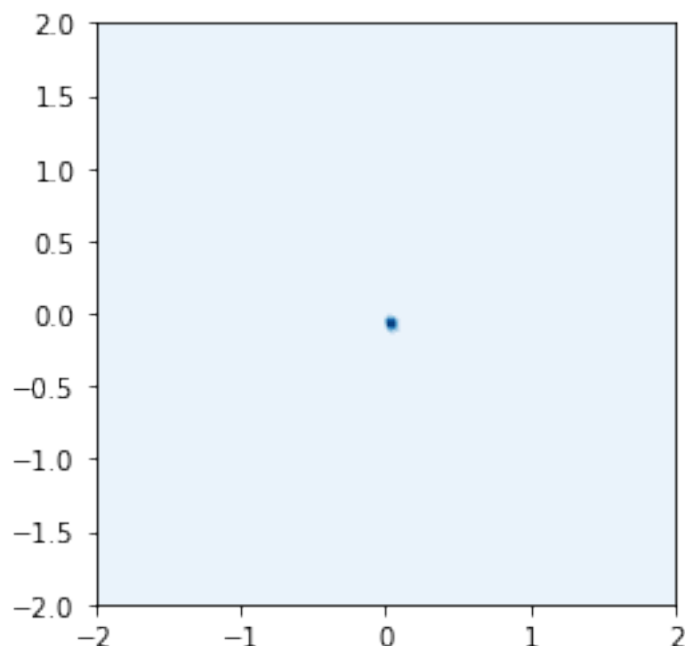


i = 10000, discriminant loss = 1.3855, generator loss =0.6939



SGA

i = 0, discriminant loss = 1.3549, generator loss =0.6912



1.9 Experiment 7: Comparison of optimizers when learning a high dimensional Gaussian

Learning a high dimensional Gaussian (dimension 75 in the experiment below) using a GAN is an experiment that was proposed in [A Classification-Based Study of Covariate Shift in GAN Distributions](#) by Santurkar in ICML 2018 .

In the experiments below, the graphs plot all 75 singular values, in decreasing order, of the covariance matrix of the data generated by the GAN.

```
In [0]: def compute_eigenvalue(sess, x, n_pts, title):
        """Computes the singular values of the covariance matrix of x.

        The singular values are displayed in decreasing order in a plot.

        Args:
            sess: a Session object.
            x: a Tensor of shape ``(batch_size, x_dim)``
            n_pts: an int; the number of points used to compute the covariance matrix
            title: a string; the title of the displayed plot
        """

        batch_size, x_dim = x.get_shape().as_list()
        # Round n_pts to the next multiple of batch_size
        n_runs = (n_pts + batch_size - 1) // batch_size
        n_pts = n_runs * batch_size
        mean = np.zeros([x_dim])
        moment = np.zeros([x_dim, x_dim])
```

```

for _ in range(n_runs):
    x_out = sess.run(x)
    mean += np.sum(x_out, axis=0)
    moment += np.matmul(x_out.transpose(), x_out)
mean /= n_pts
moment /= n_pts
mean_2 = np.expand_dims(mean, 0)
cov = moment - np.matmul(mean_2.transpose(), mean_2)
u, s, vh = np.linalg.svd(cov)
plt.plot(s)
plt.title(title)
plt.show()

def train(train_op, x_fake, init, disc_loss, gen_loss):
    n_iter = 20001
    n_save = 2000

    with tf.Session() as sess:
        sess.run(init)

        compute_eigenvalue(sess, x_fake, 2**20, 'BEFORE TRAINING')

        for i in range(n_iter):
            sess.run(train_op)
            disc_loss_out, gen_loss_out = sess.run([disc_loss, gen_loss])
            if i % n_save == 0:
                print('i = %d, discriminant loss = %.4f, generator loss = %.4f' %
                      (i, disc_loss_out, gen_loss_out))
                compute_eigenvalue(sess, x_fake, 2**15, 'iter %d' % i)

        compute_eigenvalue(sess, x_fake, 2**20, 'AFTER TRAINING')

def high_dim_gaussian_experiment(mode):
    print(mode)

    x_dim = 75
    def x_real_builder(batch_size):
        return tf.random_normal([batch_size, x_dim])

    train_op, x_fake, unused_z, init, disc_loss, gen_loss = reset_and_build_graph(
        depth=2, width=200, x_real_builder=x_real_builder, z_dim=200,
        batch_size=64, learning_rate=2e-4, mode=mode)

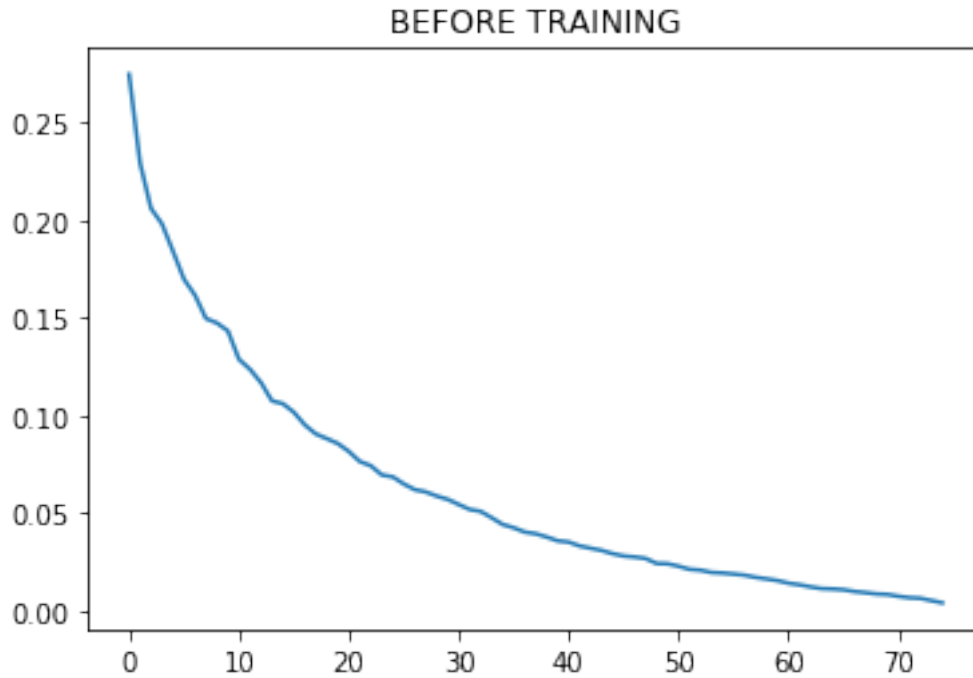
    train(train_op, x_fake, init, disc_loss, gen_loss)

```

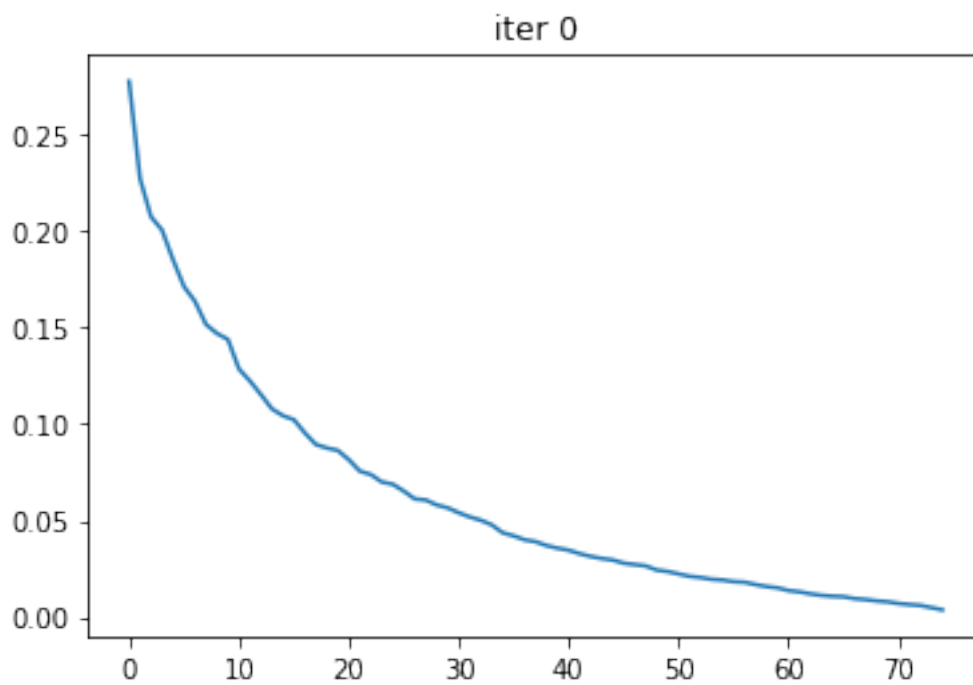
1.9.1 Using RMS

```
In [0]: # Use plain RmsProp to optimise the GAN parameters.  
        # This experiment demonstrate how traditional GAN training fails to learn all  
        # the directions of a high dimensional Gaussian.  
        high_dim_gaussian_experiment('RMS')
```

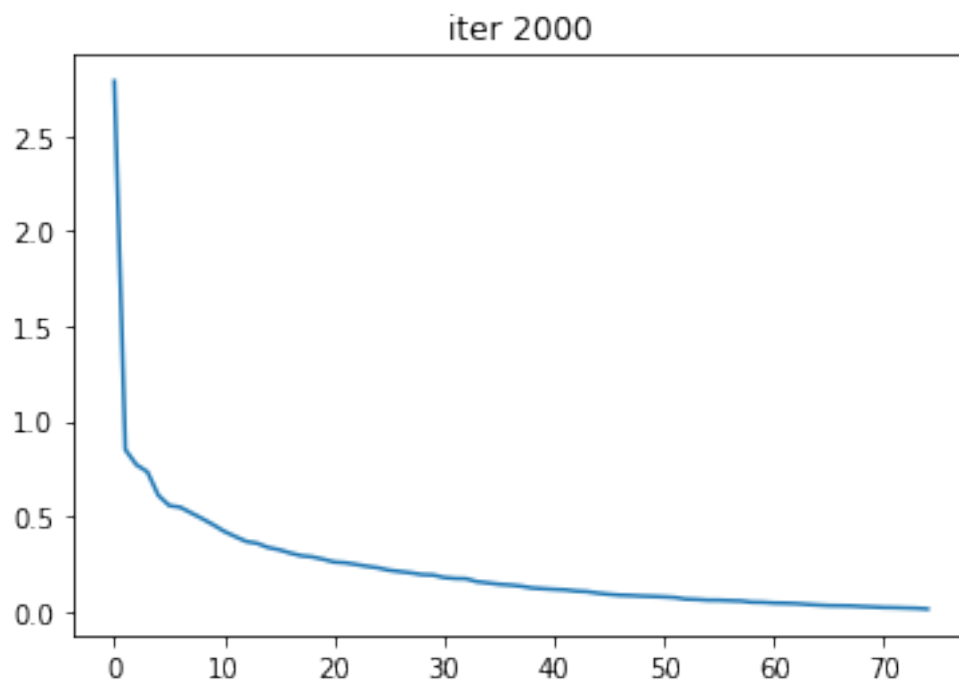
RMS



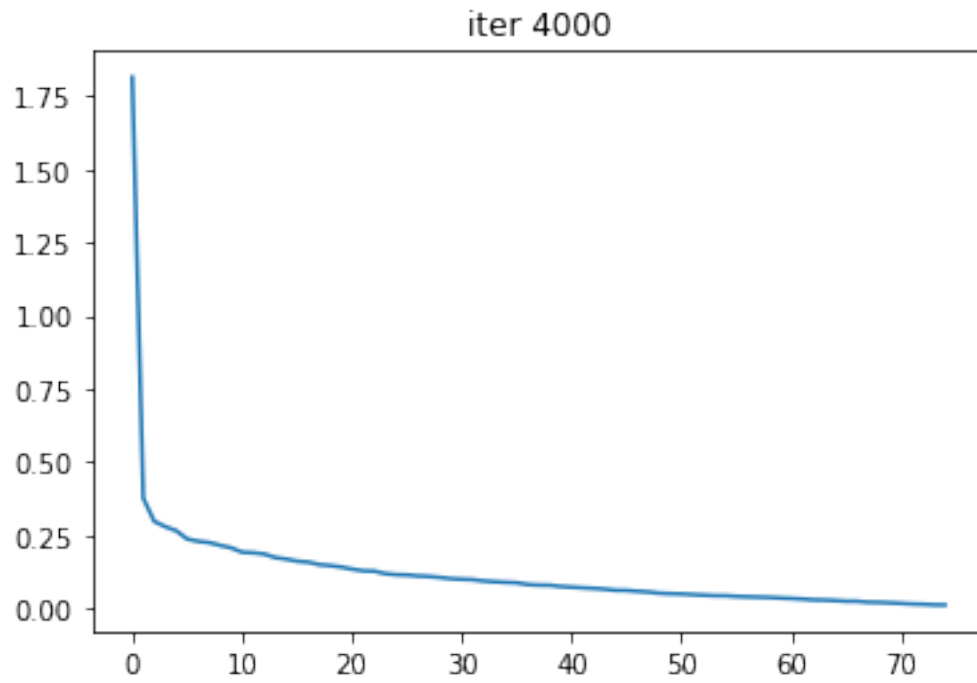
i = 0, discriminant loss = 1.3822, generator loss =0.6249



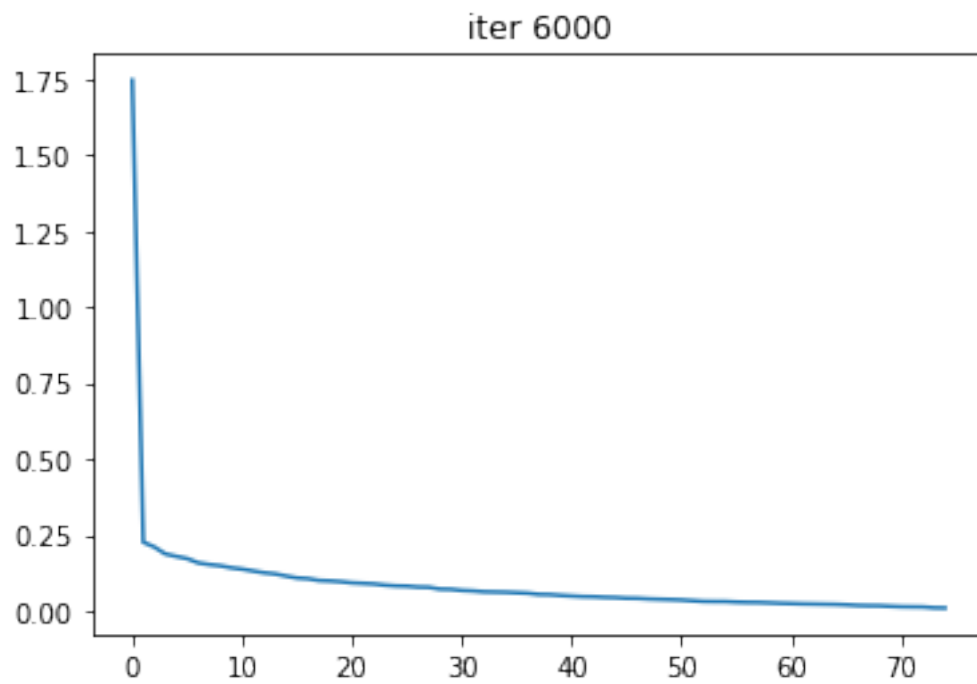
i = 2000, discriminant loss = 1.4030, generator loss = 0.6816



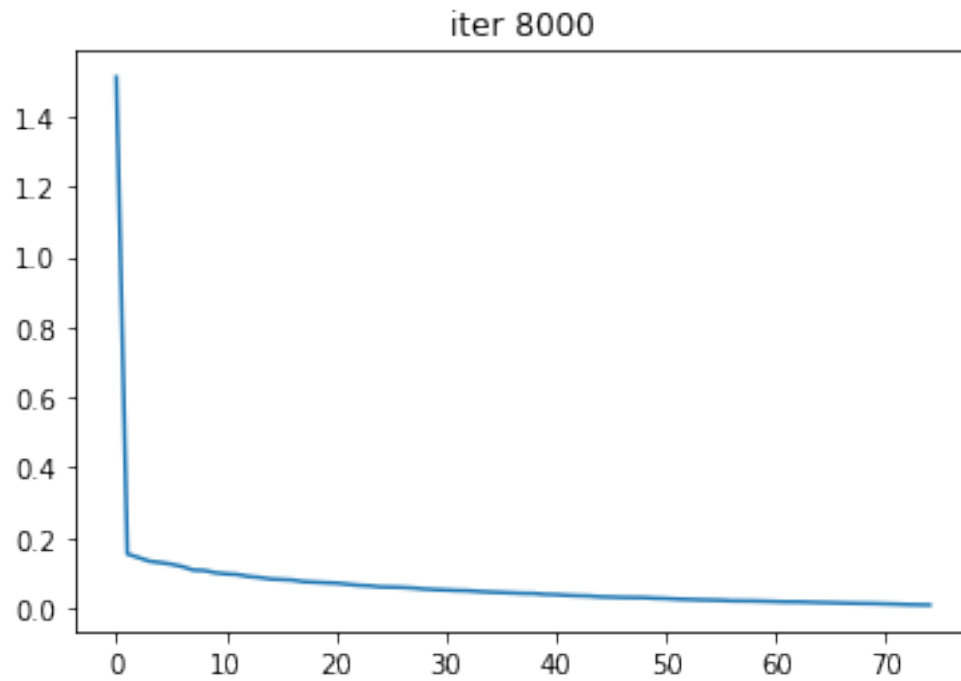
i = 4000, discriminant loss = 1.4097, generator loss = 0.6523



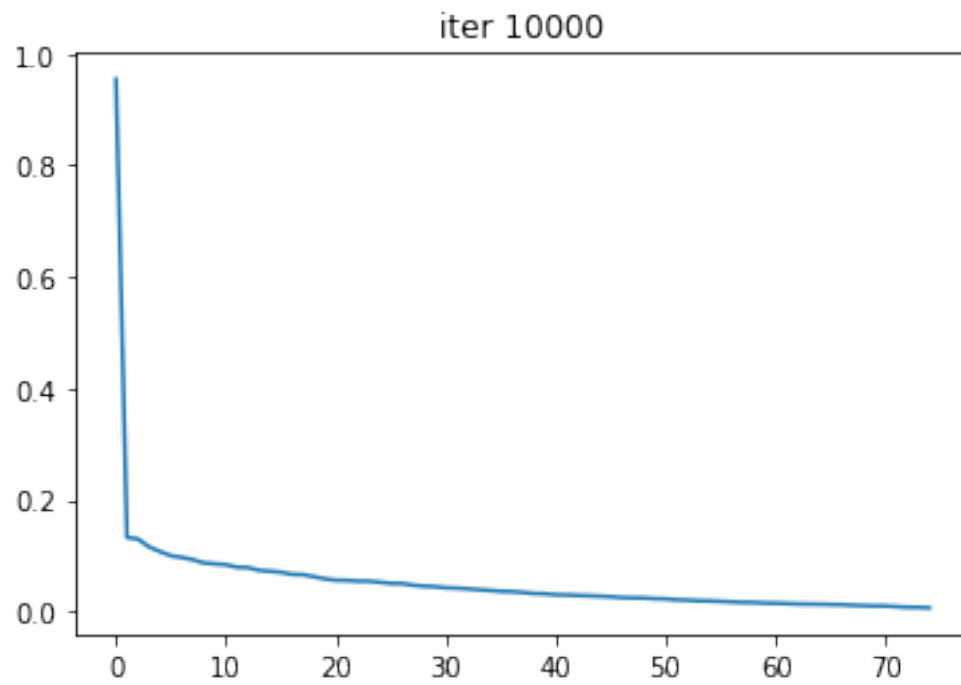
i = 6000, discriminant loss = 1.4174, generator loss = 0.6912



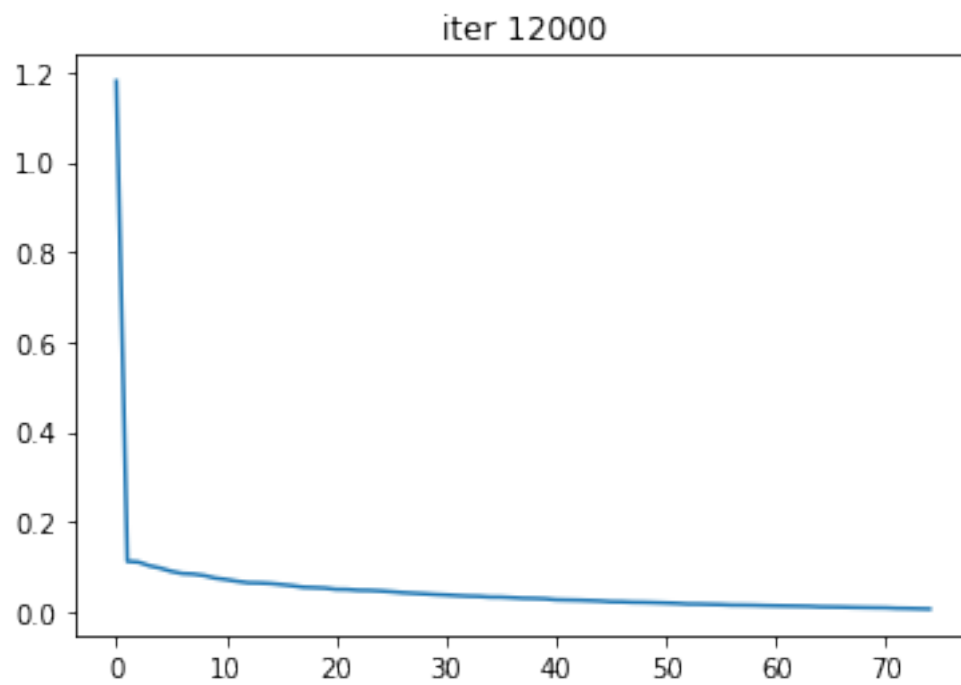
i = 8000, discriminant loss = 1.4118, generator loss =0.6706



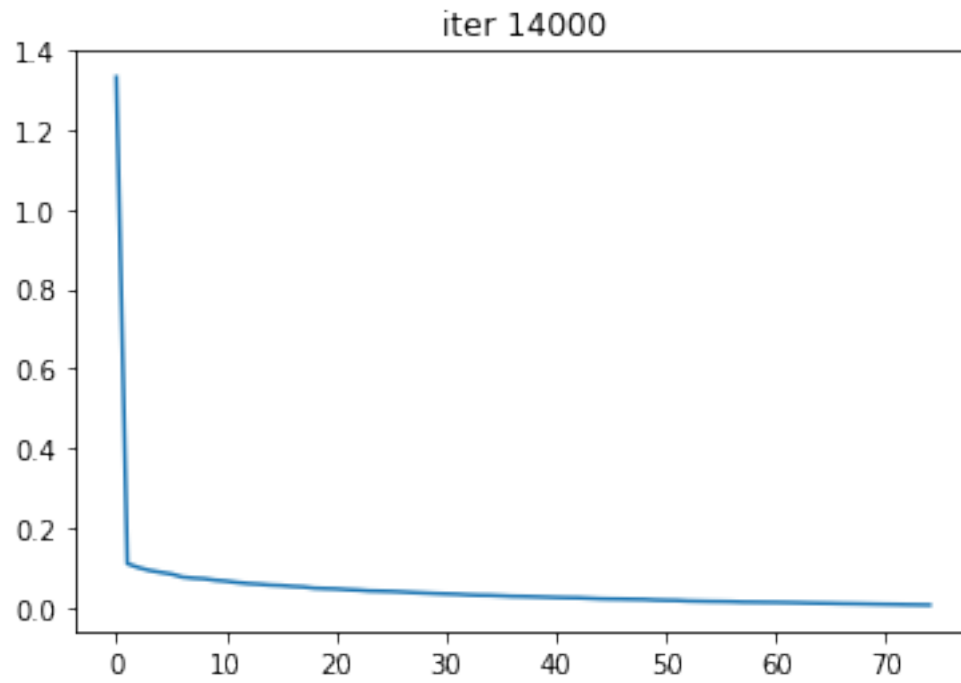
i = 10000, discriminant loss = 1.3706, generator loss =0.7215



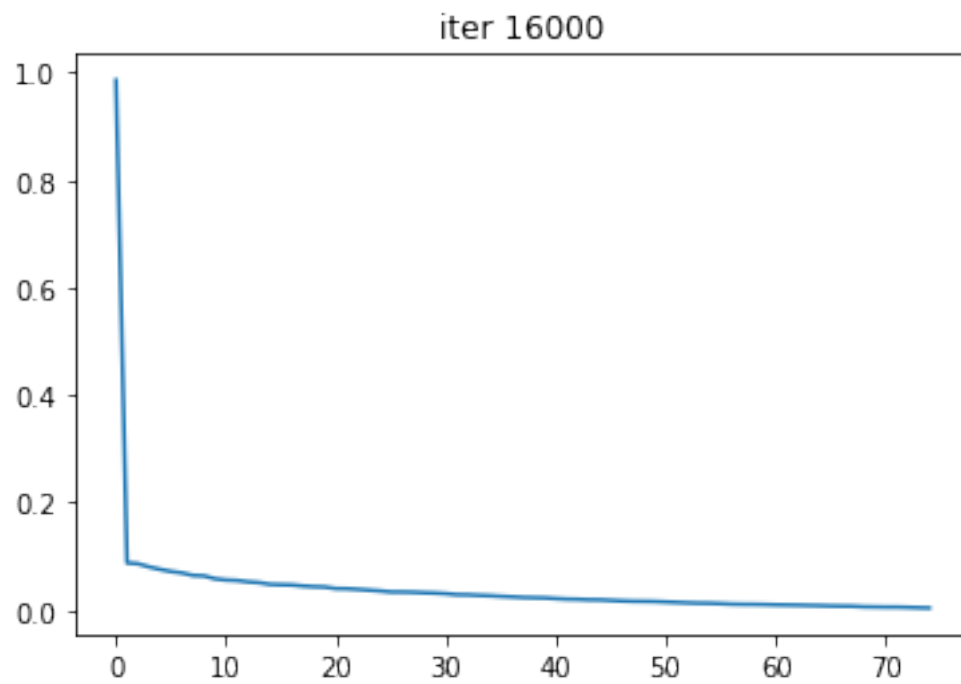
i = 12000, discriminant loss = 1.3736, generator loss = 0.7113



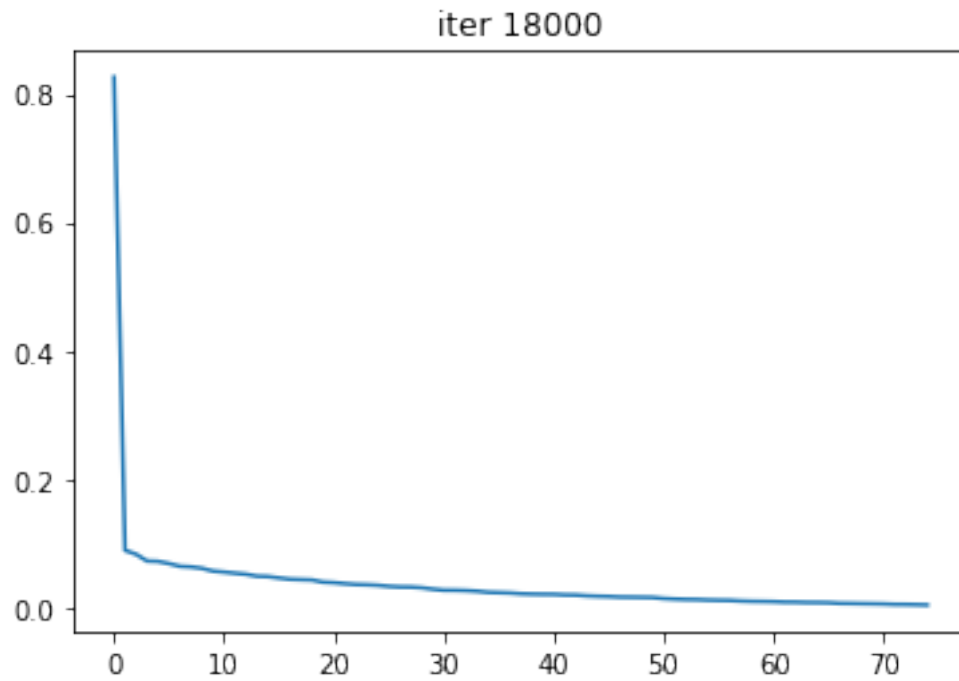
i = 14000, discriminant loss = 1.3803, generator loss = 0.7034



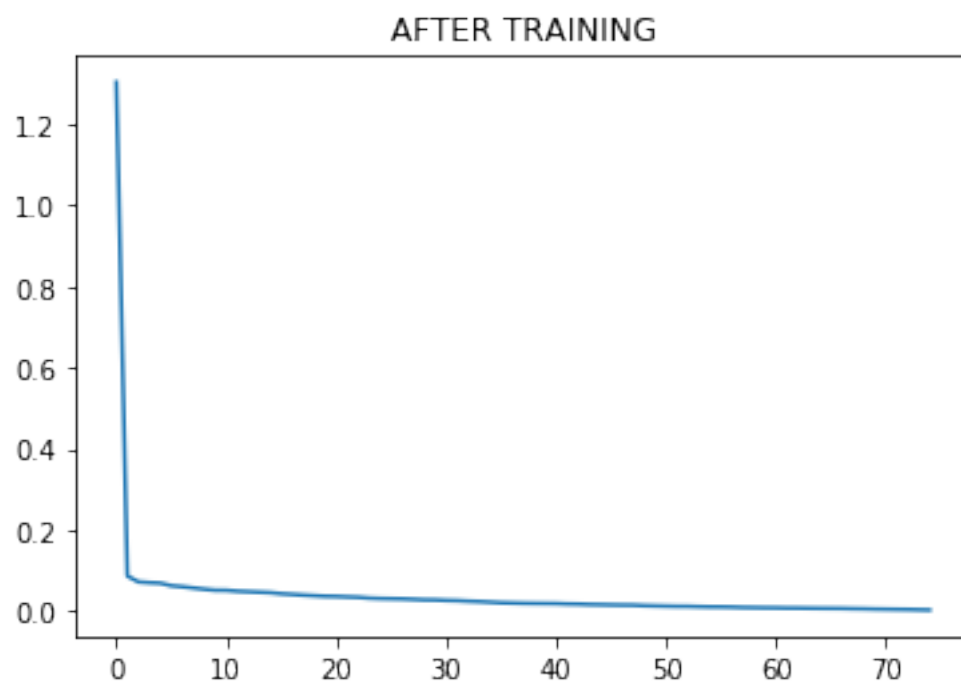
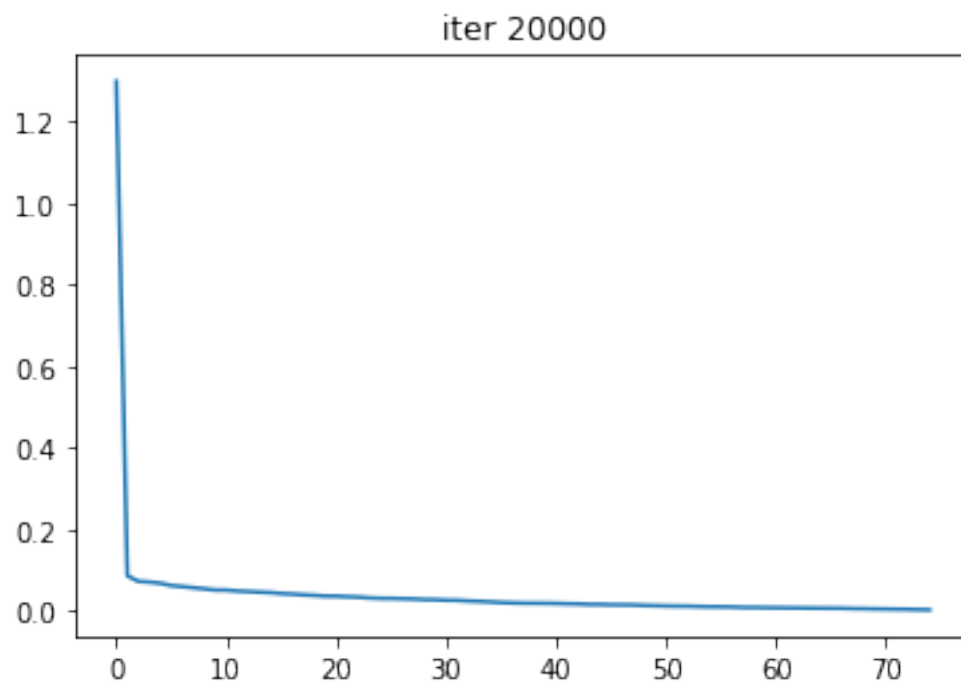
i = 16000, discriminant loss = 1.4275, generator loss = 0.7320



i = 18000, discriminant loss = 1.3801, generator loss = 0.6883



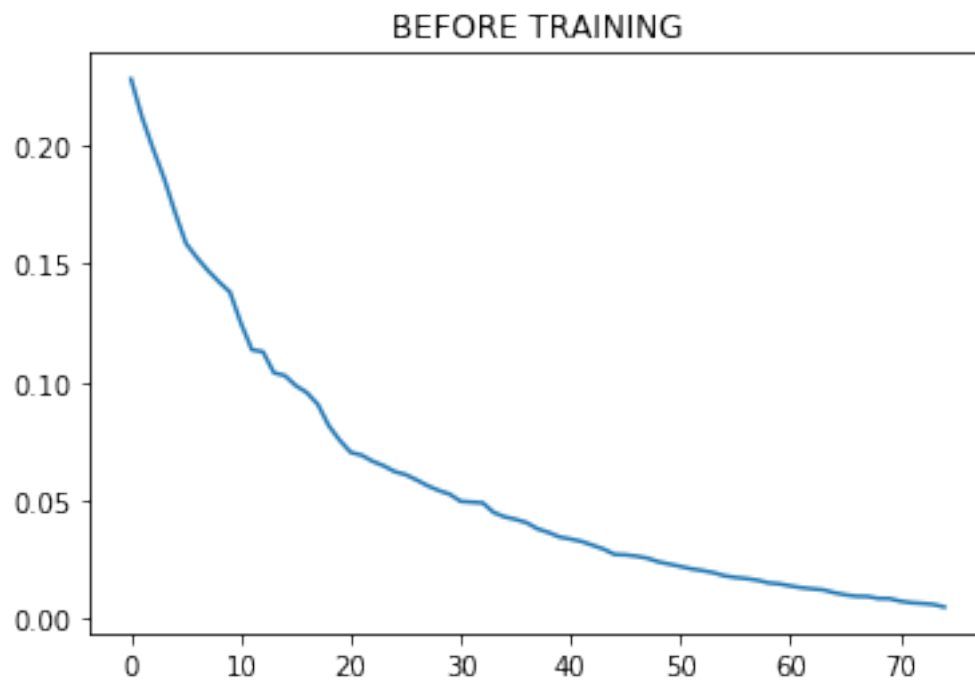
i = 20000, discriminant loss = 1.4164, generator loss = 0.7083



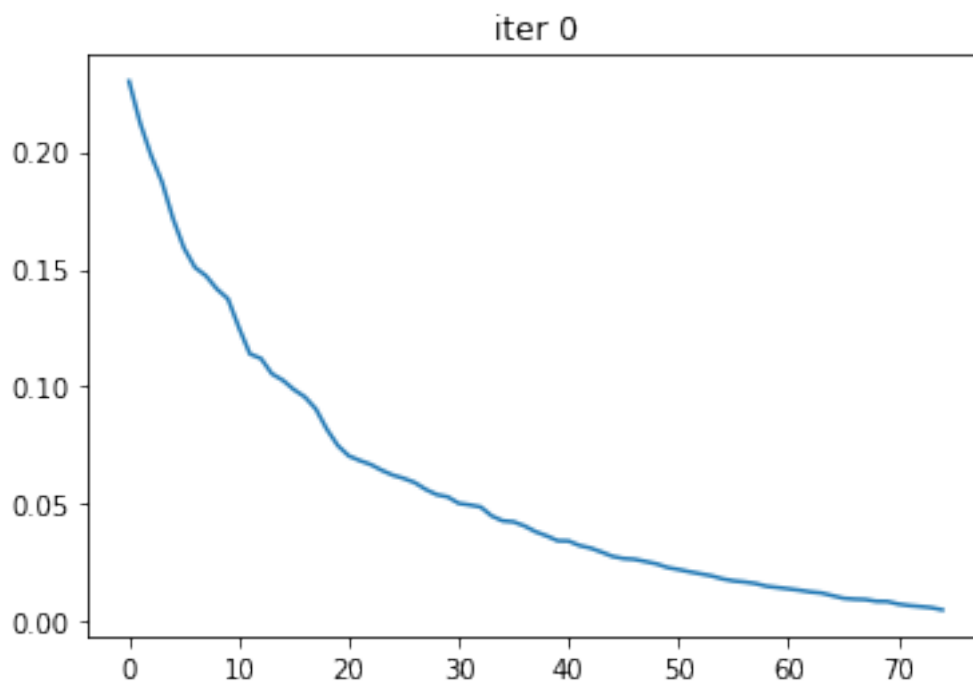
1.9.2 Using SGA

```
In [0]: # Use Symplectic Gradient Adjustment to optimise the GAN parameters  
        high_dim_gaussian_experiment('SGA')
```

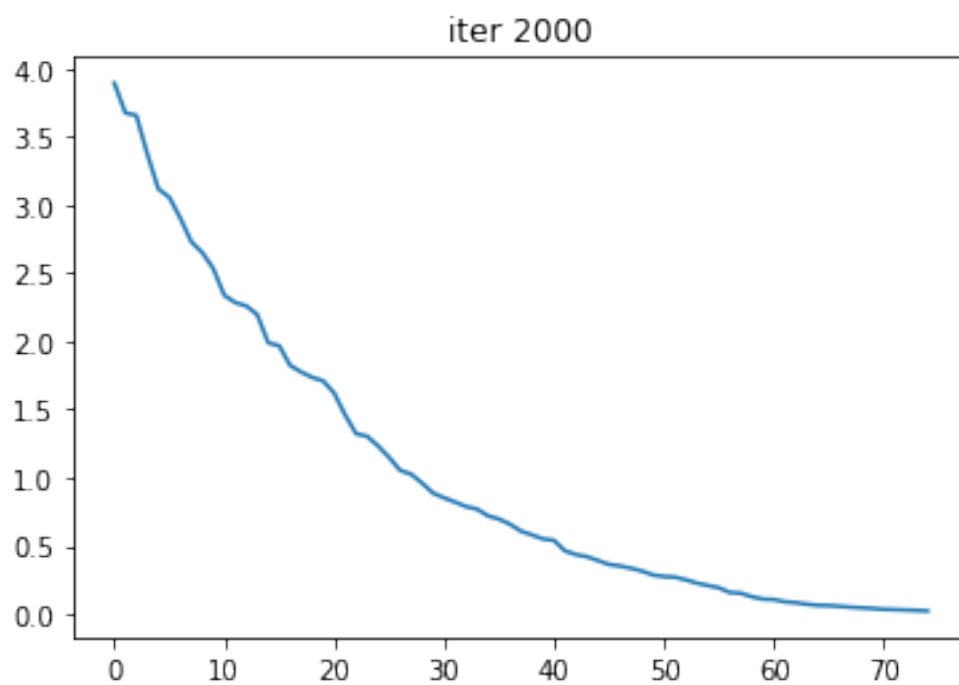
SGA



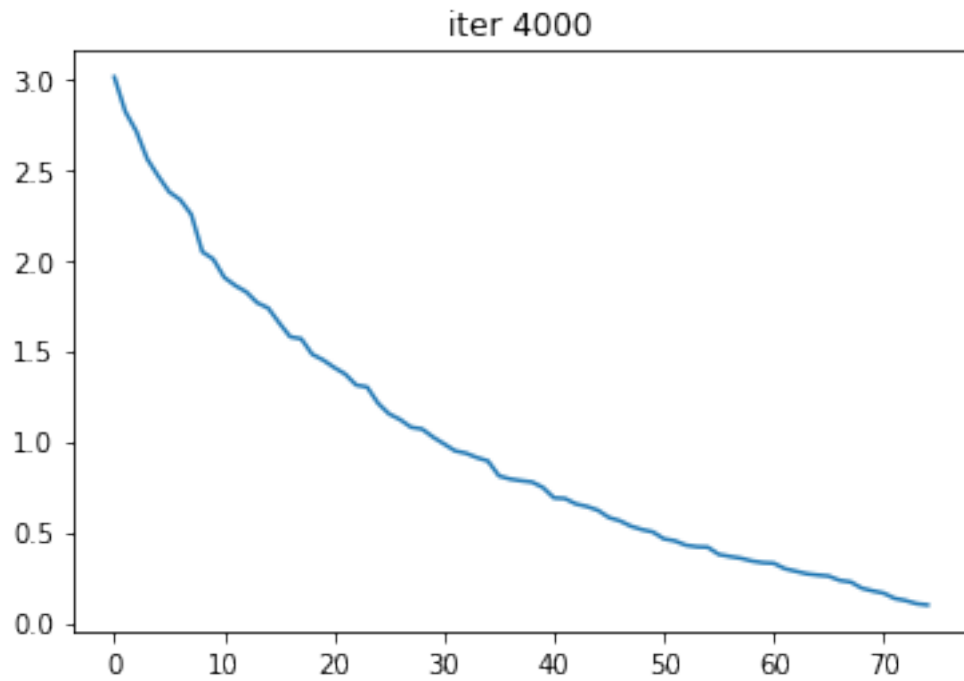
i = 0, discriminant loss = 1.5273, generator loss = 0.7279



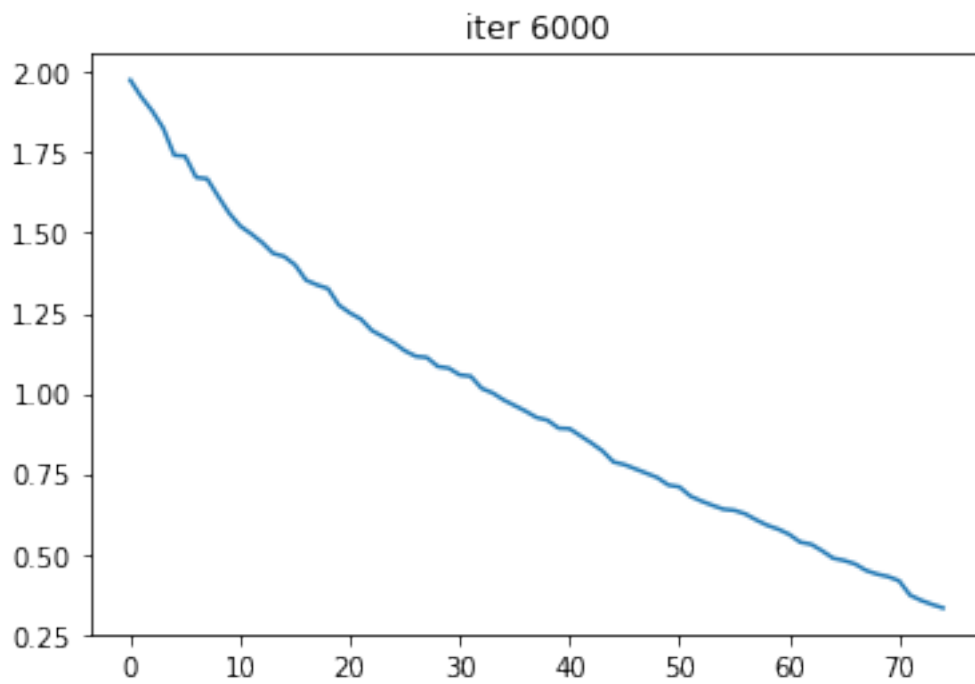
i = 2000, discriminant loss = 1.2478, generator loss = 0.7696



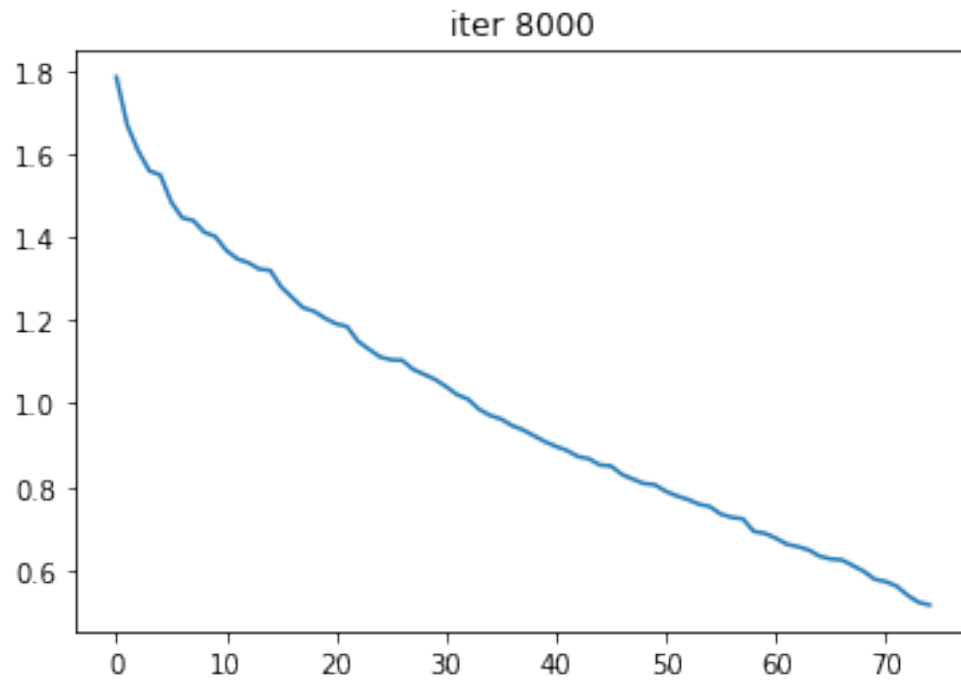
i = 4000, discriminant loss = 1.3265, generator loss = 0.7274



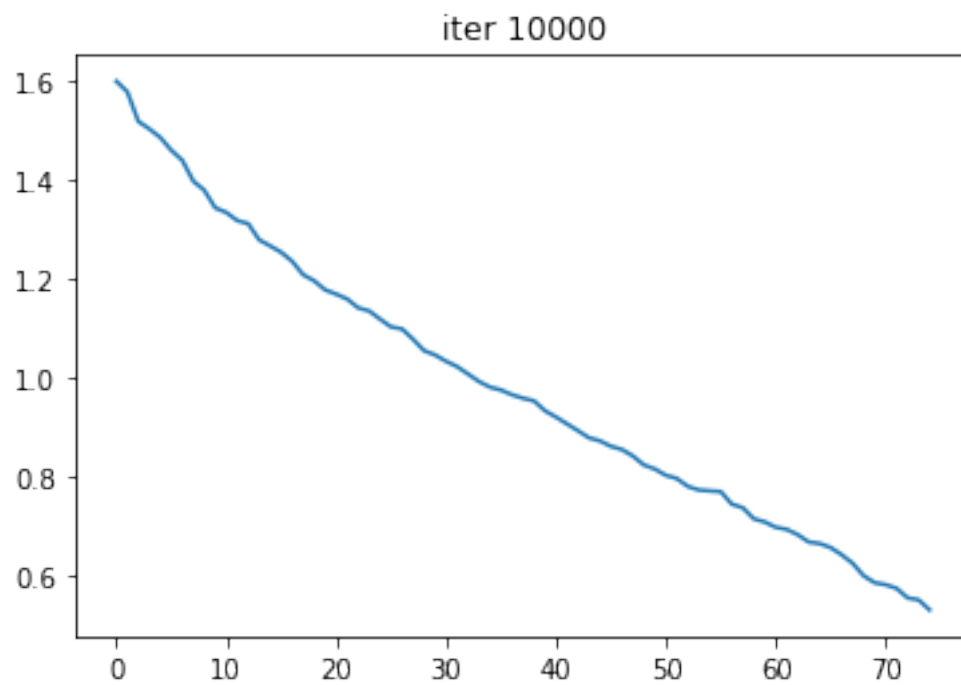
i = 6000, discriminant loss = 1.3730, generator loss = 0.6980



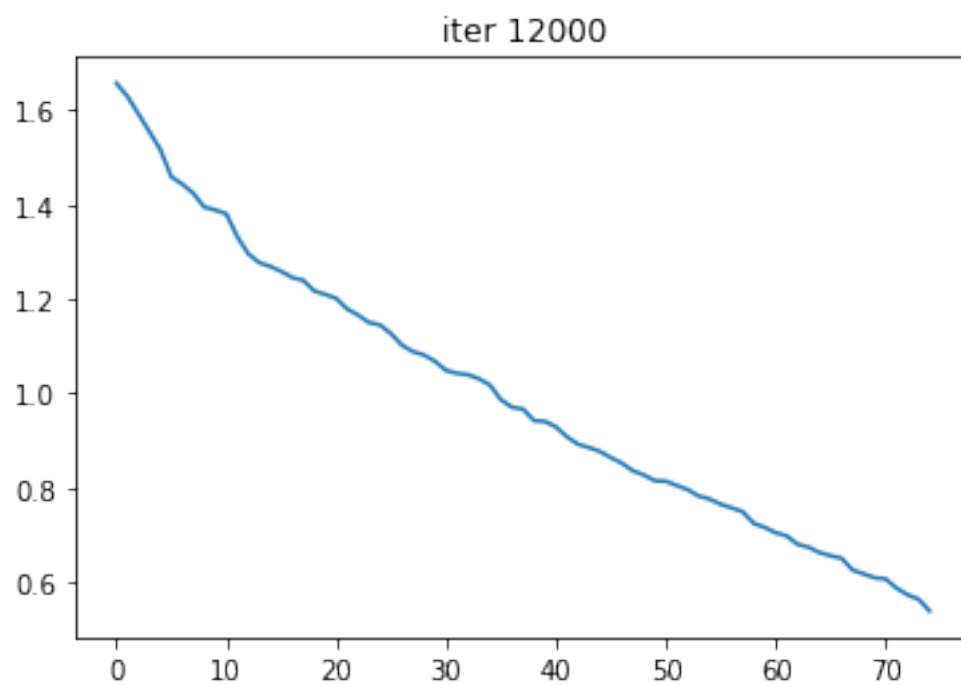
i = 8000, discriminant loss = 1.3824, generator loss =0.6947



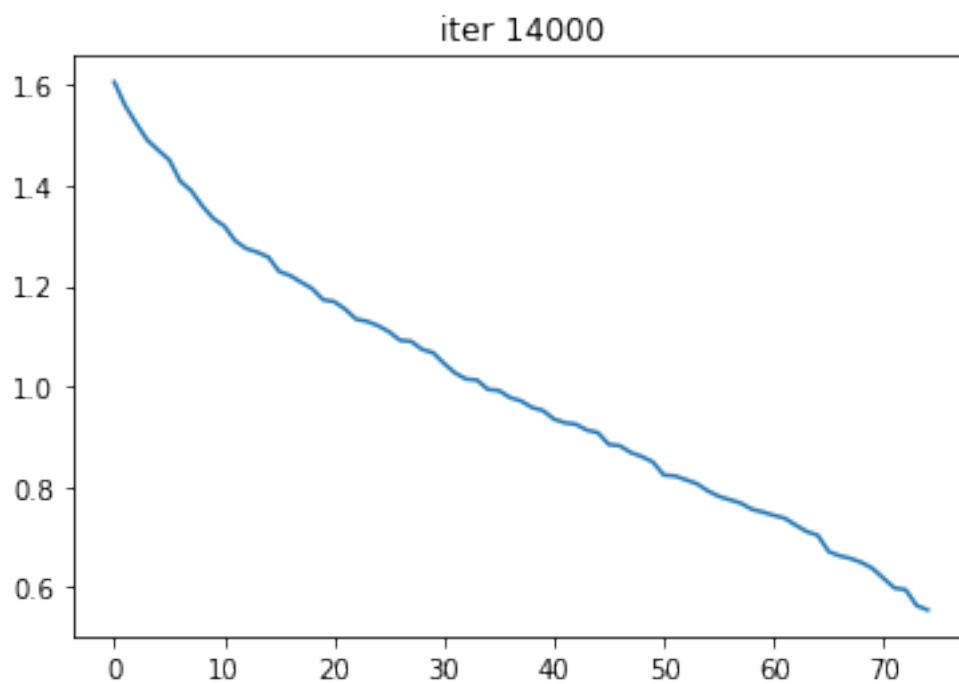
i = 10000, discriminant loss = 1.3806, generator loss =0.6989



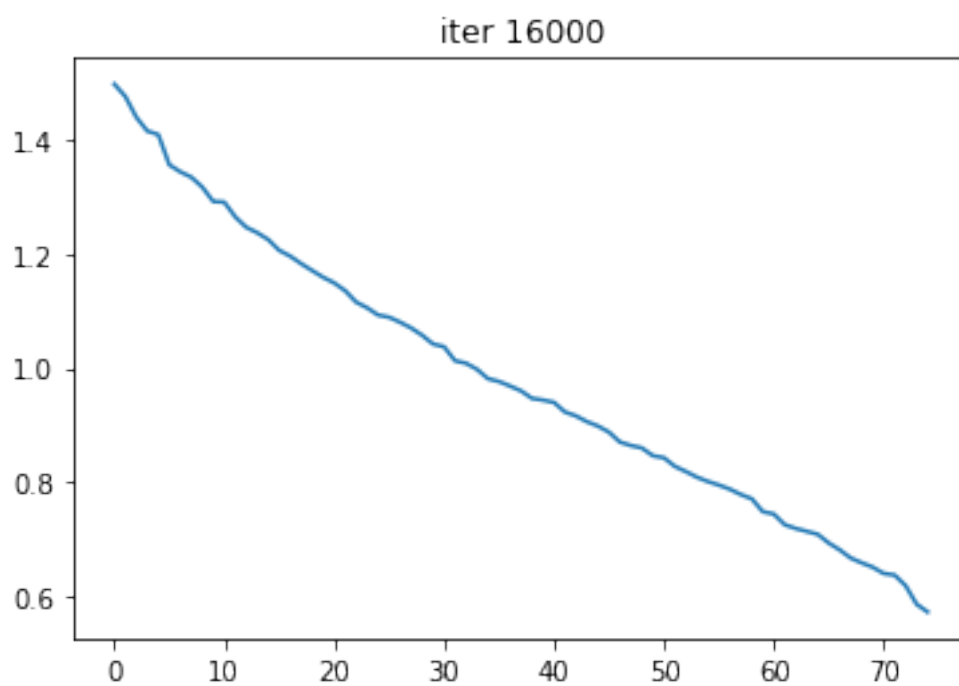
i = 12000, discriminant loss = 1.3703, generator loss = 0.7037



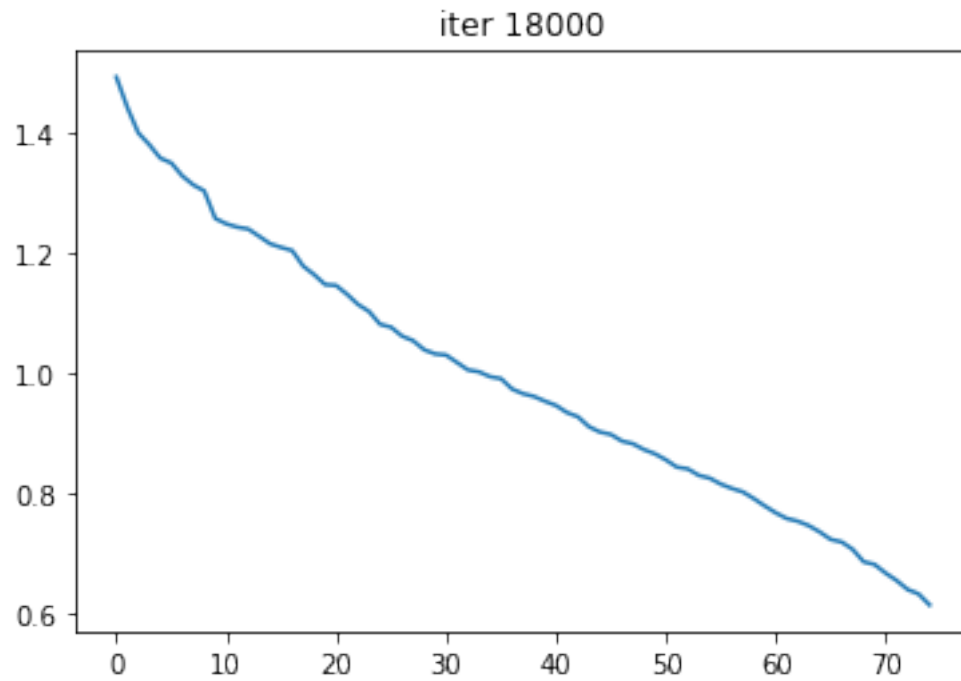
i = 14000, discriminant loss = 1.3645, generator loss =0.7058



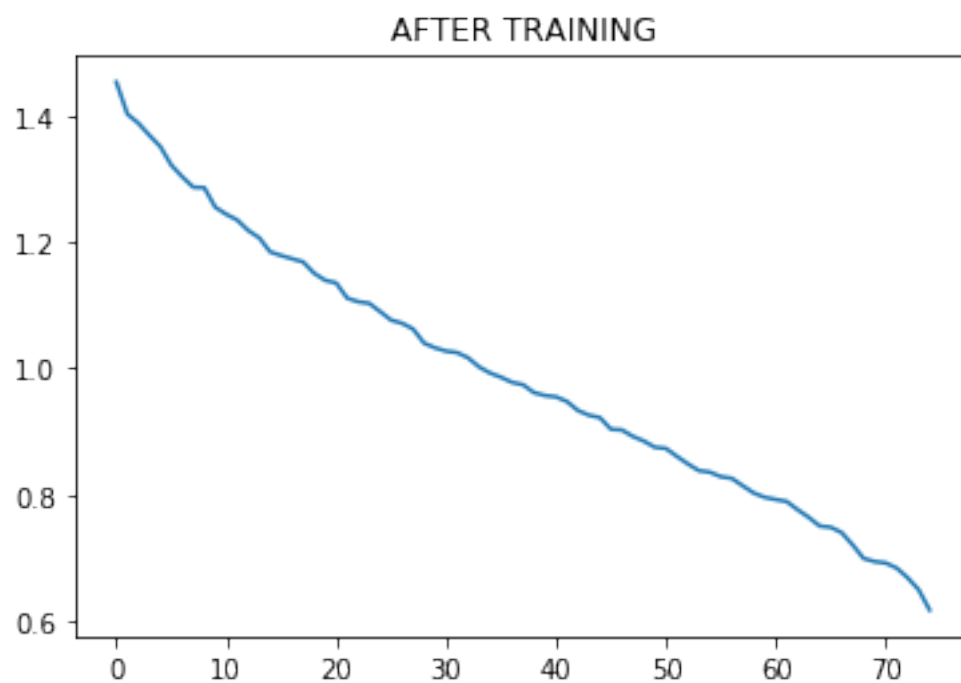
i = 16000, discriminant loss = 1.3513, generator loss =0.7180



i = 18000, discriminant loss = 1.3613, generator loss = 0.7066



i = 20000, discriminant loss = 1.3681, generator loss = 0.6979



1.10 Experiment 8: An SGA implementation in PyTorch

```
In [0]: def list_divide_scalar(xs, y):
        return [x / y for x in xs]

def list_subtract(xs, ys):
    return [x - y for (x, y) in zip(xs, ys)]

def jacobian_vec(ys, xs, vs):
    us = [torch.zeros_like(y) + float("nan") for y in ys]
    #print(us)
    dydxs = torch.autograd.grad(ys, xs, grad_outputs=us)
    print(dydxs)
    dydxs = [torch.zeros_like(x) if dydx is None else dydx for x, dydx in zip(xs, dydxs)]
    dysdx = torch.autograd.grad(dydxs, us, grad_outputs=vs)
    return dysdx

def jacobian_transpose_vec(ys, xs, vs):
    dydxs = torch.autograd.grad(ys, xs, grad_outputs=vs)
    dydxs = [torch.zeros_like(x) if dydx is None else dydx for x, dydx in zip(xs, dydxs)]
    return dydxs

def _dot(x, y):
    return torch.mm(x, y)

In [0]: class my_SGA(Optimizer):
        def __init__(self, params, lr=required, reg_params=1., use_signs=True, use_locking=False):
            if lr is not required and lr < 0.0:
                raise ValueError("Invalid learning rate: {}".format(lr))

            defaults = dict(lr=lr, reg_params=reg_params, use_signs=use_signs,
                             use_locking=use_locking)
            super(my_SGA, self).__init__(params, defaults)

        def __setstate__(self, state):
            super(my_SGA, self).__setstate__(state)

        def step(self, closure=None):

            loss = None
            if closure is not None:
                loss = closure()

            grads = []
            vars_ = []
            for group in self.param_groups:
                for p in group['params']:
                    print(p)
                    grads.append(p.grad.data)
```



```

        print(p.grad.data)
        vars_.append(p.data)
    #print(grads)
    #print(vars_)
    n = len(vars_)
    h_v = jacobian_vec(grads, vars_, grads)
    ht_v = jacobian_transpose_vec(grads, vars_, grads)
    at_v = list_divide_scalar(list_subtract(ht_v, h_v), 2.)
    if group['use_signs']:
        grad_dot_h = _dot(grads, ht_v)
        at_v_dot_h = _dot(at_v, ht_v)
        mult = grad_dot_h * at_v_dot_h
        lambda_ = torch.sign(mult / n + 0.1) * group['reg_params']
    else:
        lambda_ = group['reg_params']

    apply_vec = [(g + lambda_ * ag, x) for (g, ag, x) in zip(grads, at_v, vars_)]

    for group in self.param_groups:
        for p in group['params']:
            if p.grad is None:
                continue
            d_p = p.grad.data
            p.data.add_(-group['lr'], d_p)

    return loss

```