AML Exam Report

Domenico Villani
dovi@itu.dk

Anton Mølbjerg Eskildsen
aesk@itu.dk

December 20, 2019

Course code: KSADMAL1KU

The report contains 1694 *words counted using Overleaf's internal word-count. It likely automatically removes equations and extra spaces compared to a simple PDF word count.*

1 Introduction

The report is meant to present the results of the final project for the *Advanced Machine Learning* course. The project was the occasion to apply the theory learnt during the last semester and to share our knowledge with each other. We chose to implement the *Generative Adversarial Nets* among the paper proposed at the end of the course[4]. Generative models and *GANs* in particular are one of the most interesting topics in *Machine Learning*, they can be applied to different fields such as data augmentation in order to improve the number and the variability of a dataset and realistic graphic generation for gaming and design. The algorithm is based on two models, a *generator* that tries to capture the the data distribution generating images and a *discriminator* which estimates the probability that an image is real (comes from the training dataset) or a fake (generated by the *generator*), for this reason the training is performed as a competition between the two models until the generated images are indistinguishable from the real ones.

In the following sections we outline the theory behind the *GANs* and describe our implementation, then we show our experiments and discuss the results as required in the assignment.

2 Method

As defined in the assignment, our method should follow the source paper [4] as closely as possible. Any divergence from the original method is motivated and clearly explained.

A Generative adversarial network (GANs) or *Adversarial net* as it is referred to in the paper, is a form of generative model. Generative models approximate

a data distribution p_{data} by a *generator* $G(\mathbf{z})$ with, that implicitly defines a probability distribution p_g. Samples $\mathbf{z}$ are generated using a predefined prior distribution $p_{\mathbf{z}}$. Instead of trying to learn p_g directly, adversarial nets pits the generator against a *discriminator* $D(\mathbf{x})$ whose objective is to discriminate between samples from p_{data} and p_g. The goal of the generator is instead to try and fool the discriminator. The two networks can therefore be understood as two players playing the following min-max game with the value function $V(D, G)$

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}} \left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}} \left[\log\left(1 - D(G(\mathbf{z}))\right)\right]. \quad (1)$$

In practice, this optimisation is implemented using two neural networks and separate optimisation steps for $G(\mathbf{z})$ and $D(\mathbf{x})$. The loss function for the discriminator is

$$-\frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(\mathbf{x}^{(i)}\right) + \log\left(1 - D(G(\mathbf{z}^{(i)}))\right)\right], \quad (2)$$

and for the generator

$$\frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D(G(\mathbf{z}^{(i)})\right). \quad (3)$$

The discriminator loss is simply negated from the one presented in the source paper to allow its use with gradient descent optimisers.

The discriminator predicts a Bernoulli distribution over the probability of the input image being real or fake. For the implementation, we therefore rewrite the loss function as the combined negative log-likelihood and sigmoid which is more numerically stable [3, pp. 176-178].

2.1 Training

Training is performed mostly like a regular neural network, i.e. the dataset is partitioned into batches which are then forward-propagated through the network. The network parameters are then updated in the direction of steepest gradient descent using backward-propagation to calculate the gradient. This process continues for some amount of epochs either predefined or determined by a metric on the networks [3, pp. 239-245].

Adversarial nets are a bit different since two networks need to be updated. Additionally, data $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(n)}\}$ for the generator is drawn directly from the prior distribution $p_{\mathbf{z}}$ instead of from a sample. The discriminator is updated first and possibly multiple times before the generator is updated. We, like the authors of the source paper, chose to use the same number of updates for both nets.

2.2 Maxout layer

The source paper uses a network layer operation called maxout, which is motivated by the fact that any convex function can be approximated by taking the maximum of a set of lines. Additionally, a two-layer maxout network is, like a regular multilayer perceptron, a universal function approximator[5].

It is implemented by a simple modification to a typical linear layer with an activation function. Given a layer with M neurons/outputs, N inputs, and K functions per output, the layer is defined as

$$h_i(x) = \max_{j \in [iK, iK+K]} z_j, \tag{4}$$

where

$$z = \mathbf{x}^T W + \mathbf{b}. \tag{5}$$

The layer weights have size $W : \mathbb{R}^{N \times KM}$ and the biases $\mathbf{b} : \mathbb{R}^{KM}$. This formulation is a bit different to the one traditionally used but is closer to our actual implementation. It makes the computation easier to implement since z is just a regular linear combination. For convolution, maxout is implemented by having MK channels in the output feature-map produced by a two-dimensional convolution and applying the function

$$hc_{i,j,k}(x) = \max_{v \in [kK, kK+K]} z_{i,j,v}, \tag{6}$$

where $hc_{i,j,k}$ is the output feature vector at row i, column j, and channel k. Thus the max operation is performed on intervals of the channels of the convolution output z. The implementations are found in `util.py`.

2.3 Evaluation method

Evaluation is performed using *kernel density estimation* (Parzen Window fitting in the paper). The density estimate of samples is used to calculate the log-likelihood of the actual data. The Gaussian kernel density estimate of generated samples $\mu = \{G(\mathbf{z}_1), \ldots, G(\mathbf{z}_n)\}$ for samples $\mathbf{z}_1, \ldots, \mathbf{z}_n$ drawn from $p_{\mathbf{z}}$ is then

$$p\left(\mathbf{x}|\mu\right) = \frac{1}{n} \sum_j \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(\mathbf{x} - \mu_j)^2}{2\sigma^2}\right). \tag{7}$$

Taking the log of this expression (for use in log-likelihood) gives us

$$\log\left(p\left(\mathbf{x}|\mu\right)\right) = \log\left(\sum_j \exp\left(\frac{-(\mathbf{x} - \mu_j)^2}{2\sigma^2}\right)\right) - \log\left(\sum_i \sigma\sqrt{2\pi}\right). \tag{8}$$

Finally, we find the log-likelihood by the negative cross-entropy between the density estimate and data

$$\ell(\mathbf{X}|\mu) = \mathbb{E}_{\mathbf{X}\sim\hat{p}_{data}} \log\left(p\left(\mathbf{X}_i|\mu\right)\right). \tag{9}$$

The implementation (found in `likelihood.py`) uses a few convenience methods to enable these calculations without the use of loops. It otherwise follows this exact formulation. Note that the measure is not standardised and therefore cannot be used to compare performance across datasets.

3 Experiments

3.1 Implementation

GANs can be implemented in several different ways. We experimented with some different architectures in order to find an equilibrium for every dataset. As asked in the assignment we tried to replicate the results from the original paper as close as possible. We used the implementation from the author of the paper as inspiration and guidance on architectural details[2]. The first is a fully connected architecture using two *maxout* layers and an output layer implemented using a *sigmoid* function for the *discriminator*. The *generator* has two $ReLU$ layers with a *sigmoid* function as output layer. The second architecture uses convolutional layers. The *discriminator* uses three convolutional layers designed for using convolutional filters, then the data is reshaped to apply a maxout layer and an output layer using a *sigmoid* function. The *generator* uses three layers, two fully connected using a $ReLU$, a *sigmoid* as activation function, and an output layer which is a transposed convolution.

The models are implemented in `mnist.py`, `cifar10_dense.py`, and `cifar10_cnn.py`. They are trained in `train.py`. The training revealed the limits of these architectures, in fact we generated plausible results only for the *MNIST* dataset trained using the fully connected architecture. For this reason we chose to implement a new training (`train_2.py`) combining the sigmoid and cross entropy as mentioned in section 2 for the *generator* and the *Discriminator* losses. Moreover we modified the *CNN* architecture that uses $LeakyReLU$ activation for the convolutional layers in the *discriminator* and two layers of *transposed convolution*, $LeakyReLU$ and *dropout* for the *generator*. We found this implementation gives us better results on the *cifar10* dataset.

3.2 Results

The models were trained for a different amount of epochs depending on their performance. *RMSProp* [3, p. 299] was used in place of stochastic gradient descent due to the paper implementing learning rate decay which isn't directly supported by Tensorflow. It is also widely used today as the standard algorithm for back-propagation alongside *Adam* [3, p. 299, 413]. Generated samples are

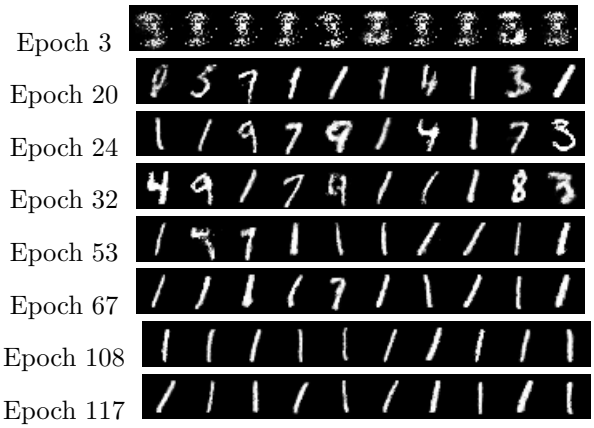

Figure 1: Images generated using MNIST dataset

Figure 2: Images generated using Cifar10 dataset and the fully connected architecture

shown for the MNIST model in Figure 1, the dense CIFAR10 model in Figure 2, and the cnn CIFAR10 model in Figure 3.

The highest observed likelihoods for each model are shown in Table 1. The log-likelihood estimate recorded at each epoch for each model, is shown in Figure 4.

4 Discussion

Considering the MNIST model, we notice how the model starts to generate visually coherent images at epoch 24, then from epoch 53, we can see how

Figure 3: Images generated using Cifar10 dataset and the *CNN* architecture

Source	MNIST	CIFAR (dense)	CIFAR (CNN)
Goodfellow et. al.[4]	225 ± 2		
Ours	97 ± 6	847 ± 12	558 ± 12

Table 1: Highest observed log-likelihood estimates for the different models.

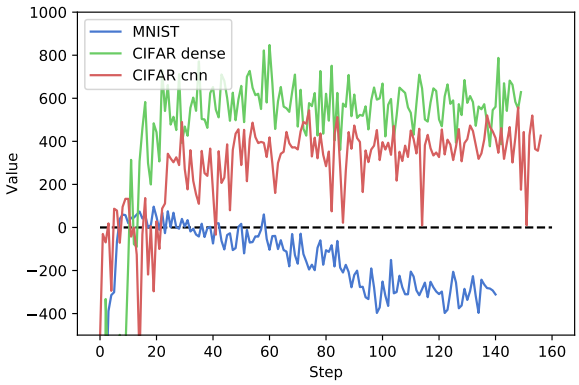

Figure 4: Likelihood estimate as a function of epochs.

the model is effected by a *mode collapse* which happens when the *Generator* finds a particularly good output which fools the *Disciminator* more than other generated samples and starts to generate only that kind of image[1]. In our case, a complete mode collapse to only 1s happens after epoch 67. The other models don't show this kind of behavior as is clear from the samples shown in Figure 2 and Figure 3. We suspect this is partly due to the simplicity of the MNIST data making it easier for the generator to be caught in a local minimum.

The models compare quite favourably against the ones presented in the source paper [4]. For the CIFAR10 dataset, the model using fully connected layers produces much more blurry images. This should be expected since convolutional neural networks are translation invariant, making them more spatially aware [6]. We believe results would be even more believable if multiple upscaling layers where used for the generator and swapping transposed convolution with regular convolution combined with some other upsampling method. Using deconvolution leads to a number of undesirable side effects as explained in [7] and is therefore better replaced with regular convolution and image upsampling.

The source paper only reports results for the MNIST and TFD datasets, the latter of which is not publicly available and therefore not used. The likelihood of the source paper is 131% higher than ours which is clearly a significant difference. This is likely due to their model being more finely tuned and tested for more hyperparameters. Additionally, they did not disclose the exact parameters used for their log-likelihood estimation. The variance of the Gaussian kernel has a significant impact on the log-likelihood result. We modelled our method after

theirs, using a small subset of the training data for finding the optimal variance in the interval $[0.1, 1]$. We do not, however, know their final chosen range for testing the variance.

The MNIST model quickly deteriorates which is likely caused by the observed mode collapse. The other models seem to increase steadily and then level off, reaching what seems to be some equilibrium between the discriminator and generator. The models are thus either limited by capacity or by being stuck in a local optimum. It is likely a combination due to the models relative simplicity compared to more modern architectures as well as the use of a simple cross entropy loss.

5 Conclusion

This report presented the results of our project. We implemented models close to the original paper implementation a few modifications modifications in order to achieve better results. We presented the theory and our implementation of different parts of the network, we showed our experiments, evaluated and discussed our results. The project improved our understanding of such a complex neural network and showed us the difficulties of getting a stable training.

References

[1] Common problems — generative adversarial networks — google developers.

[2] Ian Goodfellow. adversarial. `https://github.com/goodfeli/adversarial`, 2014.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[5] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks, 2013.

[6] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015.

[7] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016.