Computer Vision by Learning: Practical Assignments

University of Amsterdam – Spring 2019

April 3, 2019

In this practical assignment you will learn how to implement and train basic neural architectures like multi-layered perceptrons (MLPs), convolutional neural networks (CNNs), recurrent neural networks (RNNs) and generative adversarial networks (GANs). The document is organized in a number of sections, each containing its own set of questions and implementation assignments:

- **Section 1**: Multi-layered Perceptrons
- **Section 2**: Convolutional Neural Networks
- **Section 3**: Recurrent Neural Networks
- **Section 4**: Generative Adversarial Networks

What language to use? Python and PyTorch have a large community of people eager to help other people. If you have coding related questions: (1) read the documentation, (2) search on Google and StackOverflow, (3) ask your question on StackOverflow and finally (4) ask the teaching assistants.

Code repository? In order to get you started with these assignments, you're advised to start using the code provided in this repository.

Computation needed? All you need is your laptop and a connection to the internet. Then you can use CoLab to train your neural networks on GPUs for free. If your laptop has a GPU, then that's a plus. If not, CoLab is more than enough. **WiFi?** Please login to the free WiFi network `Amsterdam Science Park`.

What to hand-in? We expect each participant to write an individual report about *all* the questions in this practical assignment. Please clearly mark each answer by a heading indicating the question number. Use the NIPS LaTeX template as provided in this link. Create ZIP archive containing your report and all Python code. Please preserve the directory structure as provided in the Github repository for this assignment. Give the ZIP file the following name: `lastname.zip` where you insert your last name. Finally, please send the ZIP file to this e-mail cv-by-learning-2019@outlook.com.

The deadline for this assignment is April 30th at 23:59.

1 Multi-layer Perceptrons (MLPs)

The main goal of this part is to make you familiar with PyTorch. PyTorch is a deep learning framework for fast, flexible experimentation. It provides two high-level features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autodiff system

You can also reuse your favorite python packages such as NumPy, SciPy and Cython to extend PyTorch when needed.

There are several tutorials available for PyTorch:

- Deep Learning with PyTorch: A 60 Minute Blitz
- Learning PyTorch with Examples

In this assignment, you will study and implement a multilayer perceptron (MLP). The task will be to classify tiny images into 10 different categories. Implementing this task will get you familiar with the data processing pipeline as well as the hyper-parameters tuning required to train an MLP.

In general, MLPs are considered as universal approximators. This means they can learn to represent a wide variety of functions. In other words, MLPs are not restricted to images but could also be used for learning text or speech representations for example. Nevertheless, their capacity and their performance might not be optimal, that is why more specialized types of neural networks exist to process images, videos, text, or sound. We will explore in Section 2 how to exploit the spatial structure of images, and in Section 3 how to model sequential data.

Question 1.1

First, start with the code in the directory `code\part1_MLP_CNN` in this repository. Implement the MLP in `mlp_pytorch.py` file by following the instructions inside the file. The interface is similar to `mlp_numpy.py`. Implement training and testing procedures for your model in `train_mlp_pytorch.py` by following instructions inside the file. You should train your MLP on the CIFAR-10 dataset for which you can use the code in `cifar10_utils.py`.

Before proceeding with this question, convince yourself that your MLP implementation is correct. For this question you need to perform a number of experiments on your MLP to get familiar with several parameters and their effect on training and performance. For example you may want to try different regularization types, run your network for more iterations, add more layers, change the learning rate and other parameters as you like. Your goal is to get the best test accuracy you can. You should be able to get *at least 0.53* accuracy on the CIFAR-10 test set but we challenge you to improve this. List modifications that you have tried in the report with the results that you got using them. Explain in the report how you are choosing new modifications to test. Study your best model by plotting accuracy and loss curves.

2 Convolutional Neural Networks (CNNs)

At this point you should have already noticed that the accuracy of MLP networks is far from being perfect. A more suitable type of architecture to process image data is the

CNNs. They are now the widely use in both academia and industry for very broad image processing tasks.

CNNs rely on the convolution operation, which has many advantages. It reduces the number of parameters, compared to an MLP, to make CNNs less prone to overfitting. Additionally, this parameter sharing makes the features equivariant to scale and translations. For a more in-depth understanding of the CNNs, we highly encourage you to read these CS231n course notes from Stanford University.

In this part of the assignment you are going to implement a small version of the popular VGG network, which has won the ImageNet 2014 challenge.

Name	Kernel	Stride	Padding	Channels In/Out
conv1	3×3	1	1	3/64
maxpool1	3×3	2	1	64/64
conv2	3×3	1	1	64/128
maxpool2	3×3	2	1	128/128
conv3_a	3×3	1	1	128/256
conv3_b	3×3	1	1	256/256
maxpool3	3×3	2	1	256/256
conv4_a	3×3	1	1	256/512
conv4_b	3×3	1	1	512/512
maxpool4	3×3	2	1	512/512
conv5_a	3×3	1	1	512/512
conv5_b	3×3	1	1	512/512
maxpool5	3×3	2	1	512/512
avgpool	1×1	1	0	512/512
linear	–	-	-	512/10

Table 1. Specification of ConvNet architecture. All *conv* blocks consist of 2D-convolutional layer, followed by Batch Normalization layer and ReLU layer.

Question 2.1

Implement the ConvNet specified in Table 1 inside `convnet_pytorch.py` file by following the instructions inside the file. Implement training and testing procedures for your model in `train_convnet_pytorch.py` by following instructions inside the file. Use Adam optimizer with default learning rate. Use default PyTorch parameters to initialize convolutional and linear layers. With default parameters you should get around *0.75* accuracy on the test set. Study the model by plotting accuracy and loss curves.

3 Recurrent Neural Networks (RNNs)

In this assignment you will study and implement recurrent neural networks (RNNs). This type of neural network is best suited for sequential processing of data, such as a sequence of characters, words or video frames. Its applications are mostly in neural machine translation, speech analysis and video understanding. These networks are very powerful and have found their way into many production environments. For example **Google's neural machine translation system** relies on Long-Short Term Networks (LSTMs).

Before continuing with the first assignment, we highly recommend each student to read this excellent blogpost by Chris Olah on recurrence neural networks: **Understanding**

LSTM Networks. For the second part of the assignment, you also might want to have a look at the **PyTorch recurrent network documentation**.

For the first task, you will compare vanilla Recurrent Neural Networks (RNN) with Long-Short Term Networks (LSTM). PyTorch has a large amount of building blocks for recurrent neural networks. However, to get you familiar with the concept of recurrent connections, in this first part of this assignment you will implement a vanilla RNN and LSTM from scratch. The use of high-level operations such as `torch.nn.RNN`, `torch.nn.LSTM` and `torch.nn.Linear` is not allowed until the second part of this assignment. Convolutional Neural Networks

3.1 Toy Problem: Palindrome Numbers

In this first assignment, we will focus on very simple sequential training data for understanding the memorization capability of recurrent neural networks. More specifically, we will study *palindrome* numbers. Palindromes are numbers which read the same backward as forward, such as:

$$\begin{gathered}303\\4224\\175282571\\682846747648286\end{gathered}$$

We can use a recurrent neural network to predict the next digit of the palindrome at every timestep. While very simple for short palindromes, this task becomes increasingly difficult for longer palindromes. For example when the network is given the input 68284674764828_ and the task is to predict the digit on the _ position, the network has to remember information from 14 timesteps earlier. If the task is to predict the last digit only, the intermediate digits are irrelevant. However, they may affect the evolution of the dynamic system and possibly erase the internally stored information about the initial values of input. In short, this simple problem enables studying the memorization capability of recurrent networks.

For the coding assignment, in the file `main_part/dataset.py`, we have prepared the class `PalindromeDataset` which inherits from `torch.utils.data.Dataset` and contains the function `generate_palindrome` to randomly generate palindrome numbers. You can use this dataset directly in PyTorch and you do not need to modify contents of this file. Note that for short palindromes the number of possible numbers is rather small, but we ignore this sampling collision problem for the purpose of this assignment.

3.2 Vanilla RNN

The vanilla RNN is formalized as follows. Given a sequence of input vectors $\mathbf{x}^{(t)}$ for $t = 1, \dots T$, the network computes a sequence of hidden states $\mathbf{h}^{(t)}$ and a sequence of output vectors $\mathbf{p}^{(t)}$ using the following equations for timesteps $t = 1, \dots, T$:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h) \tag{1}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{2}$$

As you can see, there are several trainable weight matrices and bias vectors. $\mathbf{W}_{hx}$ denotes the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden (or recurrent) weight matrix, $\mathbf{W}_{ph}$ represents the hidden-to-output weight matrix and the $\mathbf{b}_h$ and $\mathbf{b}_p$ vectors denote the biases. For the first timestep $t = 1$, the expression $\mathbf{h}^{(t-1)} = \mathbf{h}^{(0)}$ is replaced with a special vector $\mathbf{h}_{init}$ that is commonly initialized to a vector of zeros. The output value $\mathbf{p}^{(t)}$ depends on the state of the hidden layer $\mathbf{h}^{(t)}$ which in its turn depends on all previous state of the hidden layer. Therefore, a recurrent neural network can be seen as a (deep) feed-forward network with shared weights.

To optimize the trainable weights, the gradients of the RNN are computed via back-propagation through time (BPTT). The goal is to calculate the gradients of the loss $\mathcal{L}$ with respect to the model parameters $\mathbf{W}_{hx}, \mathbf{W}_{hh}$ and $\mathbf{W}_{ph}$ (biases omitted). Similar to training a feed-forward network, the weights and baises are updated using SGD or one of its variants. Different from feed-forward networks, recurrent networks can give output logits $\hat{\mathbf{y}}^{(t)}$ at every timestep. In this assignment the outputs will be given by the softmax function, *i.e.* $\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)})$. For the task of predicting the final palindrome number, we compute the standard cross-entropy loss *only* over the last timestep:

$$\mathcal{L} = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{3}$$

Where k runs over the number of classes ($K = 10$ because we have ten digits). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels.

Question 3.1

Implement the vanilla recurrent neural network as specified by the equations above in the file `vanilla_rnn.py`. For the *forward* pass you will need Python's `for`-loop to step through time. You need to initialize the variables and matrix multiplications yourself without using high-level PyTorch building blocks. The weights and biases can be initialized using `torch.nn.Parameter`. The *backward* pass does not need to be implemented by hand, instead you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. We have prepared boilerplate code in `main_part/train.py` which you should use for implementing the optimization procedure.

Question 3.2

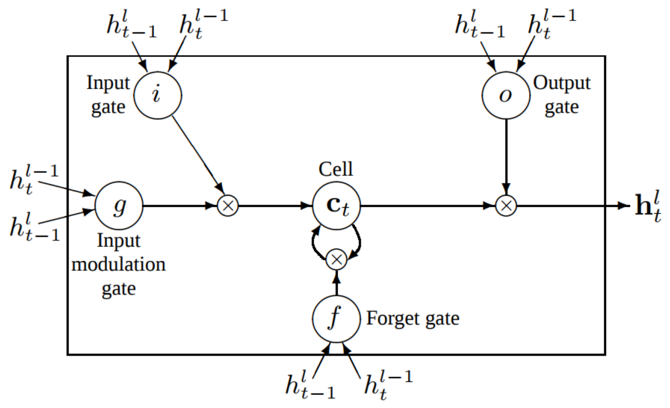
As the recurrent network is implemented, you are ready to experiment with the memorization capability of the vanilla RNN. Given a palindrome of length T, use the first $T-1$ digits as input and make the network predict the last digit. The network is *successful* if it correctly predicts the last digit and thus was capable of memorizing a small amount of information for T timesteps.

Start with short palindromes ($T = 5$), train the network until convergence and record the accuracy. Repeat this by gradually increasing the sequence length and create a plot that shows the accuracy versus palindrome length. As a sanity check, make sure that you obtain a near-perfect accuracy for $T = 5$ with the default parameters provided in `main_part/train.py`.

3.3 Long-Short Term Network (LSTM)

As you have observed after implementing the previous questions, training a vanilla RNN for remembering its inputs for an increasing number of timesteps is difficult. The problem is that the influence of a given input on the hidden layer (and therefore on the output layer), either decays or blows up exponentially as it unrolls the network. In practice, the *vanishing gradient problem* is the main shortcoming of vanilla RNNs. As a result, training

Figure 1. A graphical representation of LSTM memory cells (Zaremba *et al.* (ICLR, 2015))



a vanilla RNNs to consistently learn tasks containing delays of more than ~ 10 timesteps between relevant input and target is difficult. To overcome this problem, many different RNN architectures have been suggested. The most widely used variant is the Long-Short Term Network (LSTMs). An LSTM (Figure 1) introduces a number of gating mechanisms to improve gradient flow for better training. Before continuing, please read the following blogpost: **Understanding LSTM Networks**.

In this assignment we will use the following LSTM definition:

$$\mathbf{g}^{(t)} = \tanh(\mathbf{W}_{gx}\mathbf{x}^{(t)} + \mathbf{W}_{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g) \tag{4}$$
$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i) \tag{5}$$
$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \tag{6}$$
$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o) \tag{7}$$
$$\mathbf{c}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{8}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)} \tag{9}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{10}$$
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}). \tag{11}$$

In these equations $\odot$ is element-wise multiplication and $\sigma(\cdot)$ is the sigmoid function. The first six equations are the LSTM's core part whereas the last two equations are just the linear output mapping. Note that the LSTM has more weight matrices than the vanilla RNN. As the forward pass of the LSTM is relatively intricate, writing down the correct gradients for the LSTM would involve a lot of derivatives. Fortunately, LSTMs can easily be implemented in PyTorch and automatic differentiation takes care of the derivatives.

Question 3.3

(a) The LSTM extends the vanilla RNN cell by adding four gating mechanisms. Those gating mechanisms are crucial for successfully training recurrent neural networks. The LSTM has an *input modulation gate* $\mathbf{g}^{(t)}$, *input gate* $\mathbf{i}^{(t)}$, *forget gate* $\mathbf{f}^{(t)}$ and *output gate* $\mathbf{o}^{(t)}$. For each of these gates, write down a brief explanation of their purpose; explicitly discuss the non-linearity they use and motivate why this is a good choice.

(b) Given the LSTM cell as defined by the equations above and an input sample $\mathbf{x} \in \mathbb{R}^{T \times d}$ where T denotes the sequence length and d is the feature dimensionality. Let n denote the number of units in the LSTM and m represents the batch size. Write down the formula for the *total number* of trainable parameters in the *LSTM cell* as defined above.

Question 3.4

Implement the LSTM network as specified by the equations above in the file `lstm.py`. Just like for the Vanilla RNN, you are required to implement the model without any high-level PyTorch functions. You do not need to implement the *backward* pass yourself, but instead you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. For the optimization part we have prepared the code in `train.py`.

Using the palindromes as input, perform the same experiment you have done in *Question 3.1*. Train the network until convergence. You might need to adjust the learning rate when increasing the sequence length. The initial parameters in the prepared code provide a starting point. Again, create a plot of your results by gradually increasing the sequence length. Write down a comparison with the vanilla RNN and think of reasons for the different behavior. As a sanity check, your LSTM should obtain near-perfect accuracy for $T = 5$.

3.4 Bonus Part 1: MNIST Classification using LSTM

As we learned before, LSTM is a neural network that can learn a to represent a sequence of items, be it sequence of characters in a string, or a sequence of words in a sentence. In this experiment, we will use it to represent digit images of MNIST. But the problem is that each image in MNIST is not a sequence, but rather a two-dimensional tensor, i.e. a matrix. To overcome this, we reshape the image to become a one-dimensional sequence of pixels. So, each image is reshaped from 28×28 matrix into 784×1 vector.

Question 3.5

First, start by the code snippet `bonus_part1/lstm_mnist.py` and implements a 2-layer LSTM network. What kind of non-linearities will you use? Also, what is the non-linearity after the last layer? How many timesteps your are going to use?

Second, after you implemented your LSTM, now you can start with the code snippet `bonus_part1/train_lstm_mnist.py` to start training your LSTM. How many training epochs will you need? Can you compare how good or bad the LSTM for classifying the MNIST in comparison with the MLP in section 1 or the CNN in section 2? Is the LSTM better or worse than CNN? and why?

3.5 Bonus Part 2: RNNs as Generative Model

This is a bonus question. Students of the graduate Deep Learning course at the University of Amsterdam have typically considered this to be a fun and educative assignment. However, training the network requires some GPU resources which we cannot guarantee each course participant to have. Therefore, we have decided to make it a bonus question.

In this assignment you will build an LSTM for generation of text. By training an LSTM to predict the next character in a sentence, the network will learn local structure in text. You will train a two-layer LSTM on sentences from a book and use the model to generate new next. Before starting, we recommend reading the blog post **The Unreasonable Effectiveness of Recurrent Neural Networks**.

Given a training sequence $\mathbf{x} = (x^1, \dots, x^T)$, a recurrent neural network can use its output vectors $\mathbf{p} = (p^1, \dots, p^T)$ to obtain a sequence of predictions $\mathbf{y}^{(t)}$. In the first part of this assignment you have used the recurrent network as *sequence-to-one* mapping, here we use the recurrent network as *sequence-to-sequence* mapping. The total cross-entropy loss can be computed by averaging over all timesteps using the target labels $\mathbf{y}^{(t)}$.

$$\mathcal{L}^{(t)} = -\sum_{k=1}^{K} \mathbf{y}_k^{(t)} \log \hat{\mathbf{y}}_k^{(t)} \tag{12}$$

$$\mathcal{L} = \frac{1}{T} \sum_{t} \mathcal{L}^{(t)} \tag{13}$$

Again, k runs over the number of classes (vocabulary size). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels. Using this sequential loss, you can train a recurrent network to make a prediction at every timestep. The LSTM can be used to generate text, character by character that will look similar to the original text. Just like multi-layer perceptrons, LSTM cells can be stacked to create deeper layers for increased expressiveness. Each recurrent layer can be unrolled in time.

For training you can use a large text corpus such as publicly available books. We provide a number of books in the `assets` directory. However, you are also free to download other books, we recommend Project Gutenberg as good source. Make sure you download the books in plain text (.txt) for easy file reading in Python. We provide the `TextDataset` class for loading the text corpus and drawing batches of example sentences from the text.

The files `train_text.py` and `model.py` provide a starting point for your implementation. The sequence length specifies the length of training sentences which also limits the number of timesteps for backpropagation in time. When setting the sequence length to 30 steps, the gradient signal will never backpropagate more than 30 timesteps. As a result, the network cannot learn text dependencies longer than this number of characters.

Question 3.6

We recommend reading **PyTorch's documentation on RNNs** before you start. Study the code and its outputs for `bonus_part2/dataset.py` to sample sentences from the book to train with. Also, have a look at the parameters defined in `bonus_part2/train.py` and implement the corresponding PyTorch code to make the features work. We obtained good results with the default parameters as specified, but you may need to tune them depending on your own implementation.

- Implement a *two*-layer LSTM network to predict the next character in a sentence by training on sentences from a book. Train the model on sentences of length $T = 30$ from your book of choice. Define the total loss as average of cross-entropy loss over all timesteps (Equation 13).
- Make the network generate new sentences of length $T = 30$ now and then by randomly setting the first character of the sentence. Report 5 text samples generated by the network over evenly spaced intervals during training. Carefully study the text generated by your network. What changes do you observe when the training process evolves?
- Given the trained network, make your model generate sentences longer than $T = 30$ characters. In order to do so, you need to save the LSTM's hidden state and memory state after unrolling the network for 30 timesteps and then passing it as input for the next 30 characters. In particular, it could be fun to make your network finish some sentences that make relate to the book that your are training on. For example, if you are training on Grimm's Fairytales, you could make the network finish the sentence "*Sleeping beauty is ...*". Be creative and test the capabilities of your model.

4 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GAN) are a type of deep generative models. Similar to VAEs, GANs can generate images that mimick images from the dataset by sampling an encoding from a noise distribution. In constract to VAEs, in vanilla GANs there is no inference mechanism to determine an encoding or latent vector that corresponds to a given data point (or image). Figure 2 shows a schematic overview of a GAN. One thing to notice is that a GAN consists of two separate networks (i.e., there is no parameter sharing, or the like) called the generator and the discriminator. Training a GAN leverages uses a adversarial training scheme. In short, that means that instead of defining a loss function by hand (e.g., cross entropy or mean squared error), we train a network that acts as a loss function. In the case of a GAN this network is trained to discriminate between real images and fake (or generated) images, hence the name *discriminator*. The discriminator (together with the training data) then serves as a loss function for our *generator* network that will learn to generate images to are similar to those in the training set. Both the generator and discriminator are trained jointly. In this assignment we will focus on obtaining a generator network that can generate images that are similar to those in the training set.

4.1 Training objective: A Minimax Game

In order to train a GAN we have to decide on a noise distribution $p(z)$, in this case we will use a standard Normal distribution. Given this noise distribution, the GAN training

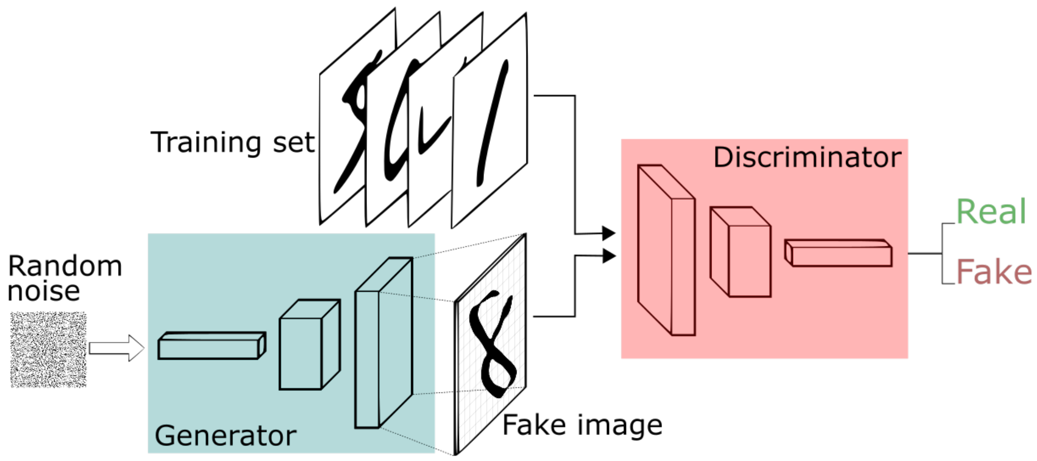


Figure 2. Schematic overview of a GAN. (Reproduced from skymind.ai)

procedure is a minimax game between the generator and discriminator. This is best seen by inspecting the loss (or optimization objective):

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{\text{data}}(x)}[\log D(X)] + \mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))] \tag{14}$$

4.2 Building a GAN

Not that the objective is specified and it is clear how the generator and discriminator should behave, we are ready to implement a GAN. In this part of the assignment you will implement a GAN in PyTorch.

Question 4.1

Build a GAN in PyTorch, and train it on the MNIST data. Start with the template in `a3_gan_template.py` in the `code` directory for this assignment.

Provide a short (no more than ten lines) description of your implementation.

Question 4.2

Sample 25 images from your trained GAN and include these in your report (see Figure 2a in [1] for an example. Do this at the start of training, halfway through training and after training has terminated.

Question 4.3

Sample 2 images from your GAN (make sure that they are of different classes). Interpolate between these two digits in latent space and include the results in your report. Use 7 interpolation steps, resulting in 9 images (including start and end point).

Report

We expect each participant to write an individual report about *all* the questions in this practical assignment. Please clearly mark each answer by a heading indicating the question number. Use the NIPS LaTeX template as provided here. Create ZIP archive containing your report and all Python code. Please preserve the directory structure as provided in the Github repository for this assignment. Give the ZIP file the following name: `lastname.zip` where you insert your last name. Finally, please send the ZIP file to this e-mail cv-by-learning-2019@outlook.com.

The deadline for this assignment is April 30th at 23:59.

References

[1] 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. 10