TDT4173: Machine Learning and Case-Based Reasoning

Assignment 4

March 17, 2017

Aleksander Rognhaugen
Department of Computer Science
Norwegian University of Science and Technology (NTNU)

- **Delivery deadline: April 06, 2017** by 22:00.
- **This assignment counts towards 4 % of your final grade.**
- You can work on your own or in groups of two people.
- Deliver your solution on *itslearning* before the deadline.
- Please upload your report as a PDF file, and package your code into an archive (e.g. zip, rar, tar).
- Your code is part of your delivery, so please make sure that your code is well-documented and as readable as possible.
- For each programming task you need to give a brief explanation of what you did, answer any questions in the task text, and show any results, e.g. plots, in the report.

Learning Objectives: Gain insight into (a) core ideas of deep learning, and (b) how simple multilayer perceptrons can be implemented using TensorFlow.

1 Theory [1 point]

a) **[0.4 points]** What is the core idea of *deep* learning? How does it differ from *shallow* learning?

b) **[0.2 points]** Explain how logistic regression can be defined as an articial neural network. Your explanation must include how many neurons there are, how the neurons are organised, and how this neural network can be trained. It may be helpful to create a drawing of the network topology in the report.

c) **[0.2 points]** Explain why an artificial neural network with only linear activation functions, i.e $g(\mathbf{x}) = \mathbf{x}$, cannot be used to model nonlinear data, regardless of how many hidden layers there are. *Tip. Envision what a single linear neuron is learning, and then extrapolate this knowledge to arbitrary numbers of hidden layers and neurons.*

For the next question we will focus on the feedforward neural network in Figure 1. As indicated by the caption, this network consist of two hidden layers $\mathbf{u}$ and $\mathbf{v}$, where the input $\mathbf{x}$ is connected to $\mathbf{u}$, and $\mathbf{v}$ is connected to the output $\mathbf{y}$. These types of hidden layers are typically referred to as *fully-connected*[1] because every neuron in a layer is connected to every neuron in the preceding layer.

The output of a single neuron typically comprise of two steps: (i) integrate the input by taking a linear combination of the input and associated weights, and (ii) feeding this scalar into an activation function. This can be summarised as a dot product between inputs and weights wrapped in a function, such as the logistic function. To simplify all of these dot products, we can gather all weights linking neurons from one layer to another into a matrix called $W^{(l)}$ for layer l, where element $\mathbf{W}_{ij}^{(l)}$ indicates the weight from neuron i to neuron j for layer l.

With this representation, the output activation $\mathbf{a}^{(l)}$ for layer l is simply the matrix-vector product $g(\mathbf{a}^{(l-1)}\mathbf{W}^{(l)})$, where $g(.)$ is an activation function and is applied *element-wise*. That is, before we apply the activation function we take the activations from the preceding layer $(l-1)$ and multiply it with the weight matrix for the current layer. For example, for the network topology in Figure 1 we can calculate the activations for layer $\mathbf{u}$ by computing $g(\mathbf{x}\mathbf{W}^{(1)})$, assuming $\mathbf{x}$ is a row vector. This makes sense, because $\mathbf{x}$ is a 1×4 row vector and $\mathbf{W}^{(1)}$ is 4×3 matrix. Thus, the output activation for layer $\mathbf{u}$ has the size 1×3, i.e. there are three neurons with one value each.

d) **[0.2 points]** Using the matrix multiplication approach outline above, perform the forward pass (inference step) on the network topology in Figure 1 using the weight matrices in Equation 1 on the input vector $\mathbf{x} = [1, -4, 0, 7]$. The input vector is a *row* vector. Every layer – that is $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{y}$ – uses the logistic function as its activation function, i.e. $g = \sigma(z) = \frac{1}{1+e^{-z}}$. Record the final output of the network as well as the intermediate activations for layer $\mathbf{u}$ and $\mathbf{v}$ in the report. Additionally, write down the full expression for calculating the output of the network in matrix notation, i.e. $y(\mathbf{x}) = \sigma(\ldots)$. *Tip. We recommend that you use a programming language with a linear algebra package to do these calculations.*

$$\mathbf{W}^{(1)} = \begin{bmatrix} -0.1 & 0.2 & 0.1 \\ 0.1 & 0.4 & 0.1 \\ 0.0 & -0.7 & 0.2 \\ 0.6 & 0.3 & -0.4 \end{bmatrix} \quad \mathbf{W}^{(2)} = \begin{bmatrix} 0.3 & -0.8 & 0.1 & 0.0 \\ 0.0 & 0.1 & 0.2 & 0.8 \\ -0.2 & 0.7 & 0.4 & 0.1 \end{bmatrix} \quad \mathbf{W}^{(3)} = \begin{bmatrix} 0.2 \\ 0.1 \\ 0.5 \\ 0.4 \end{bmatrix} \tag{1}$$

[1] Sometimes also called *densely connected*.

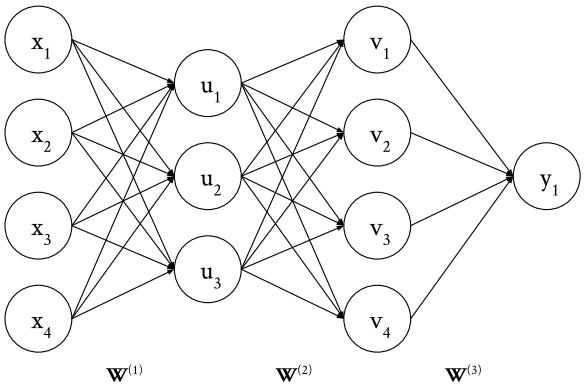

Figure 1: A simple feedforward neural network with four input neurons $\mathbf{x}$, two hidden layers $\mathbf{u}$ and $\mathbf{v}$, and one output neuron $\mathbf{y}$. The weight matrices connecting each of the layers have been indicated as $\mathbf{W}^{(l)}$. The network does not have any *bias* neurons.

2 Programming [3 points]

Basic Introduction to TensorFlow

TensorFlow[2] is an open source library by Google for distributed large-scale machine learning [1, 2]. It is one of many recent machine learning libraries, such as Theano, MXNet, and PyTorch [3, 4, 5, 6], that focus on building artificial neural networks (ANNs). The main thing that these, and other modern ANN libraries, have in common is that they all formulate models as computational graphs.

Generally, a computational graph consists of a set of nodes indicating variables, and a set of operations that connect them. A variable is generally a tensor, which, for our purposes, is an array of numbers organised in a regular grid with an arbitrary number of axes. For example, a tensor with two axes is a matrix. Operations are simplified functions of one or more variables that can return one or more variables. Two examples of mathematical expressions that have been formulated as computational graphs can be seen in Figure 2.

TensorFlow, and other computational graph libraries, may feel a little bit arcane at first; so, to facilitate the learning we will do a step-by-step walkthrough on how to solve a simple nonlinear problem. The TensorFlow API can be accessed through multiple languages, but Python is currently the only language which support the whole API. This walkthrough assumes some previous knowledge of Python; however, if you have any previous programming experience, then learning the bare minimum should be a breeze.

The XOR function (exclusive or) is a logical operation over two binary inputs that return 1 (*true*) when exactly one of its inputs is 1, otherwise it returns 0 (*false*). The truth table for the XOR function can be seen in Table 1. It is inherently a nonlinear problem because a line cannot separate the two output classes. I urge the reader to try and plot the

[2]URL: `https://www.tensorflow.org/`

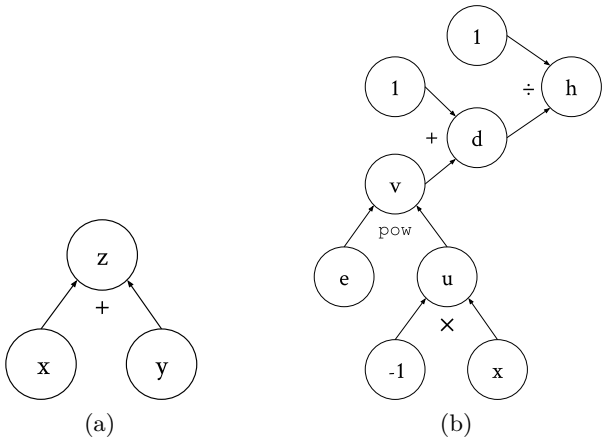

Figure 2: The expression $z = x + y$ can be seen as a computational graph in (a), while (b) illustrates $h = \frac{1}{1+e^{-x}}$ as a computational graph.

input to the XOR function as points in a rectangular coordinate system, and then try to separate points in the two classes by a line.

Table 1: Truth table for the XOR function. The input to the function are labelled x_1 and x_2.

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Thankfully, a simple feedforward neural network with one hidden layer containing two neurons can solve the problem. The ANN we will implement in TensorFlow can be seen in Figure 3. For completeness, the network includes two bias neurons just so you can see how this can be incorporated in a neural network model.

The first thing we have to do is import the libraries we are going to use. Naturally, we are going to use Tensorflow, but we are also going to import the NumPy package to hold our data[3].

```
import numpy as np
import tensorflow as tf
```

Next, let us define the XOR dataset. This is simply the truth table in Table 1 converted to NumPy arrays. `DATA_X` is a 4×2 array, while `DATA_y` is a 4×1 array. `DATA_X` is the

[3]This is not strictly necessary as we could have used Python lists, but for other problems you may want to use NumPy for data pre-processing.

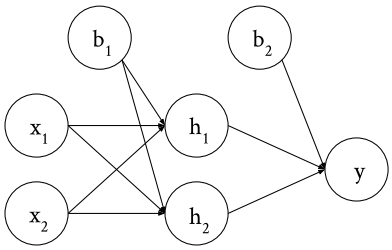

Figure 3: A feedforward neural network that can be used to learn the XOR function. It includes one hidden layer with two neurons, as well as two bias neurons: one for the hidden layer and another for the output neuron. All activation functions are assumed to be the logistic function.

input to our network, while `DATA_y` is the desired output For the rest of the code we will need to use TensorFlow.

```
DATA_X = np.array([[0,0], [0,1], [1,0], [1,1]], dtype=np.float32)
DATA_y = np.array([[0], [1], [1], [0]], dtype=np.float32)
```

Variables in TensorFlow must be instantiated to some value when they are created. A `tf.placeholder` is a special type of variable that allows us to instantiate its content at a later time. We will use these variables, one for the two inputs and one for the desired output, as entry points to our computational graph. The first argument is the data type and the second is the shape (i.e. matrix size). The first item of the shape will always be the batch size, i.e. the number of data points we feed to the model at the same time. We have set it to `None`, which essentially means that TensorFlow will figure out the size based on the amount of data we send to the model.

```
X = tf.placeholder(tf.float32, shape=(None, 2))
y = tf.placeholder(tf.float32, shape=(None, 1))
```

Looking at Figure 3, we can see that there are 2×2 weights from the input layer to the hidden layer, 2 weights from the bias to the hidden layer, 2×1 weights from the hidden layer to the output neuron, and, lastly, 1 weight from the bias to the output neuron. Not only do we have to create all of these weights as TensorFlow variables using `tf.Variable`, but we have to initialise them as well. In the code snippet below we can see the weights initialised and put in a Python dictionary[4]. The weights have been initialised by sampling values uniformly $\in [-1, 1]$, while the bias weights have simply been set to zero. The important thing to notice here, is how both `tf.random_uniform` and `tf.zeros` expect the first argument to be the size of the weight arrays.

[4]Python dictionaries consist of key-value pairs that allow for easy access of data.

```
weights = {'w1': tf.Variable(tf.random_uniform([2, 2], -1, 1)),
           'b1': tf.Variable(tf.zeros([2])),
           'w2': tf.Variable(tf.random_uniform([2, 1], -1, 1)),
           'b2': tf.Variable(tf.zeros([1]))}
```

Having a slew of numbers organised in matrices and vectors do not a neural network make, so we need to combine them in some way. This is the point where we define our artifical neural network model as a computational graph. In the following code snippet we apply quite a few TensorFlow operations. In order of usage, `tf.add` performs element-wise addition, `tf.matmul` performs matrix multiplication, and `tf.sigmoid` is the well-known logistic function. The weights are connected together using the aforementioned operations with the help of the keys we defined for the weights above. We have elected to call the output of the model `y_hat`[5]. Please convice yourself that this definition correspond to the illustration in Figure 3.

```
z1 = tf.add(tf.matmul(X, weights['w1']), weights['b1'])
h1 = tf.sigmoid(z1)
z2 = tf.add(tf.matmul(h1, weights['w2']), weights['b2'])
y_hat = tf.sigmoid(z2)
```

Just as we did for linear and logistic regression, we need to specify an error function. To simplify matters, we are going to let the error function be the same cross-entropy error function we used in assignment 1. Intuitively, this makes sense because the output of the XOR function is binary. `tf.reduce_mean` is an operation that *reduces* the size of tensor dimensions by taking the mean value. We do not specify any reduction indices, so all dimensions are reduced and we get a scalar value.

```
error = - tf.reduce_mean(tf.multiply(y, tf.log(y_hat)) +
                         tf.multiply(1 - y, tf.log(1 - y_hat)))
```

One of the main advantages of using computational graph libraries is that they typically perform automatic differentiation, which means that the library will find all the derivatives it needs by itself. Therefore, the only thing we have to do in order to run backpropagation is to specify which optimiser to use. We will use the all too familiar gradient descent optimiser. This specific optimiser only has one argument, the learning rate (`lr`), which we have set to 10.0. Immediately after the creation of the optimiser object, we apply the `minimize` method, which we tell to minimise the cross-entropy error function we defined above. The `var_list` argument is a way to explicitly define which TensorFlow variables (the values in the Python dictionary) that should be included during optimisation. This is not strictly necessary for this problem, but is useful when we have several ANN models.

```
lr = 10.0
optimiser = tf.train.GradientDescentOptimizer(lr).minimize(
    error, var_list=weights.values())
```

[5] Y-hat or $\hat{y}$ is a common name for an estimated value.

We now have everything we need: data, model, and optimiser. This means that we can create the training loop. The code for doing so can be seen below. There are three aspects of the loop that need closer inspection. First, TensorFlow requires us to run the `tf.global_variables_initializer` function. This is, unsurprisingly, an operation that initialises global variables in the graph. Second, TensorFlow uses what is known as a session to allocate memory and execute computational graphs. We can start a session using the `with` statement, which can be thought of as a try-catch-finally block defined by the session. Instantiated variables *only* live inside a session, so as soon as it is closed they are deallocated. Lastly, we use the aforementioned session to operate on the computational graph via the `run` method.

```
init = tf.global_variables_initializer()

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

    nb_epochs = 1000
    for epoch in range(nb_epochs):
        sess.run(optimiser, feed_dict={X: DATA_X, y: DATA_y})
```

As we can see above, the `run` method is first used to initialise all global variables. Then, we define a training loop that runs for 1000 epochs, or walkthroughs, of the data. For each epoch, we execute the `run` method, where the first argument is one or more operation, over the variables we have initialised, that we want to run. The second argument, `feed_dict`, is used to feed the `placeholder` variables we defined earlier with data. The `X` data will be used as input to the model, while the `y` values will be used when calculating the error. `X` and `y` are the variable names we gave to our `placeholder` variables.

It is important to realise that `feed_dict` is only required when the operation we want to run needs data to execute. For example, if we want to see the current weights between the input layer and hidden layer, we can run `sess.run(weights['w1'])`. This does not depend on any data, so we do not need to use the `feed_dict` argument.

The full code for learning the XOR function using TensorFlow can be found attached with the assignment in `tf_xor_tdt4173.py`. It includes a few additional lines of code at the end for printing the model error every 100th epoch, as well as the final prediction. Please run the code to make sure that everything works correctly. The learning is quite sensitive to weight initialisation, so you might have to run it a few times if the optimisation converges to a bad local optimum.

Task 1: Multilayer Perceptron [1 point]

A multilayer perceptron, or MLP, is a feedforward neural network and defines a mapping from inputs to outputs. An example of an MLP can be seen in Figure 1.

As we saw in the first assignment, ordinary logistic regression could not capture the data illustrated in Figure 4, unless we altered the input data by either replacing or augmenting them. In this assignment task, you will use an MLP to classify the data in Figure 4 ***without*** modifying the input data.

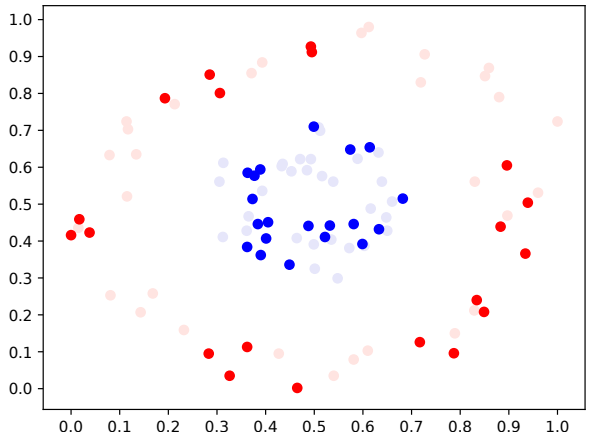

Figure 4: The nonlinear dataset we attempted to classify in assignment 1.

The assignment comes with a series of utility functions in `helpers.py`. They are meant to ease the transition to Python for those of you that are not as familiar with its functions.

a) **[0.3 points]** Modify the code we used to learn the XOR function so that it can be used to train an MLP on the data in `cl-train.csv` and `cl-test.csv`[6]. There are mainly three things that need to be altered: (i) loading of data, (ii) weights and model, and (iii) learning rate. The assignment comes with a Python function for loading the data in `helpers.py`.

b) **[0.5 points]** Experiment with different number of hidden layers, hidden neurons etc. until you are able to solve the problem. Please explain the ANN topology you defined, in the report, i.e. the number of hidden layers, number of hidden neurons per layer, which activation function(s) you used, as well as the learning rate. Take a look at the documentation for TensorFlow[7] to find information about different kinds of activation functions etc.

c) **[0.2 points]** Include a plot of the cross-entropy error function for both the training and test set over 1000 iterations.

[6]Depending on how comfortable you are with Python and TensorFlow, please feel free to write your own script.

[7]URL: `https://www.tensorflow.org/`

Task 2: Generative Adversarial Networks [2 points]

Generative adversarial networks, or GANs, is a generative modelling methodology by Goodfellow et al. [7] from 2014 that has garnered much interest these past few years. It is based on the idea of transforming samples of latent variables $\mathbf{z}$ to samples $\mathbf{x}$ of a probability distribution that we would like to learn. The transformation is done via a differentiable function, which typically is defined as an artifical neural network.

When viewed through the lens of game theory, a GAN consists of a *generator* and an adversary called the *discriminator*. The generator network $\mathbf{G}$ produces samples $\mathbf{x}$ by transforming latent variables $\mathbf{z}$ with the help of a neural network. The adversary, the discriminator network $\mathbf{D}$, attempts to discriminate between the samples $\mathbf{x}$ generated by $\mathbf{G}$ and the training data. In other words, the discriminator seeks to detect whether the input data is *fake* or *real*. At the same time, the generator attempts to *fool* the discriminator by generating plausible samples. A GAN has converged when the discriminator can no longer differentiate between real data and samples generated by the generator.

Distinguishing between fake and real data sounds like something we have done several times before; indeed, it is a binary classification problem. The original formulation of GANs as a zero-sum game can be seen in Equation 2. We can see that the discriminator tries to maximise the log-likelihood of giving the correct prediction, whilst the generator tries to minimise this quantity. In practice, the training is done by alternating optimisation between the generator and the discriminator.

$$\arg\min_{\mathbf{G}} \max_{\mathbf{D}} \frac{1}{N} \sum_{i=1}^{N} \ln \mathbf{D}(x_i) + \ln(1 - \mathbf{D}(\mathbf{G}(z_i))) \tag{2}$$

GANs are notoriously difficult to train, so for this assignment we are going to have to do some simplifications. First, we are going to train on the MNIST database of handwritten digits; a set of images with a number between 0 to 9 per image. Secondly, most of the implementation will already be done for you; the focus will be on testing out different kinds of network definitions for the generator and the discriminator. Lastly, we are going to use a recent type of GAN called a wasserstein GAN, which will make the training easier.

Wasserstein GAN, or WGAN, is a recent GAN algorithm by Arjovsky et al. [8] from earlier this year (2017) that attempts to improve the stability of GANs by reducing mode collapse, i.e. less diverse samples are being generated, and make it easier to debug by providing meaningful learning curves. While the paper can be quite tough to read, the modifications are simple. There are three main differences: (i) remove the logarithm from the error function (output is linearly activated), which means we must do something about the weights so that they do not explode (they use weight clipping); (ii) use a different gradient descent variant, they use root mean square propagation, or RMSProp; and (iii) train the discriminator more than the generator in order to stabilise gradients.

As previously mentioned, in this task you will get an implementation of a WGAN. Loading the MNIST dataset, error and optimiser definitions, as well as the training loop is already implemented. The main work that is missing is to explore different choices of

neural networks for the generator and discriminator. The implementation can be found in `tf_wgan_tdt4173.py`.

a) **[1.5 point]** Improve the neural network for the generator and discriminator by exploring different number of layers and neurons as well as internal activation functions until your generator is generating meaningful samples[8]. You may have to alter other hyperparameters such as the learning rate as well. Record the definition of your best generator and discriminator network in the report. Explain the thought process behind your decisions. Make sure to write down any of the changes you might have done to the code, such as the number of epochs, batch size, and learning rate. Additionally, show a few plots of what the best generator ended up generating.

For those of you that are feeling adventurous, know that you can improve the models in any way you want, e.g. transposed convolutions, dropout, and so forth. *Note. If your generator is not generating meaningful samples at epoch* 20000*, then it is time to alter the model in some way.*

b) **[0.4 points]** Using the best configuration you discovered in the previous task, observe how the error for the generator and the discriminator evolve over time. Does the error decrease monotonically for both networks? Explain your reasoning. In the context of the error, explain the relationship between the generator and the discriminator. Use parts of the error that is printed out by the script in your explanation.

c) **[0.1 points]** There are several different *types* of learning. What *type* of learning are GANs doing? Explain your reasoning.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org 2015 URL: `http://tensorflow.org/` (cit. on p. 3)

[8] In our case, by meaningful samples we mean that we can distinguish the digit within the generated images. It is common for GANs and other similar methods to generate blurry images, so do not worry too much about this.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "TensorFlow: A system for large-scale machine learning" in: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA* 2016 (cit. on p. 3)

[3] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio "Theano: new features and speed improvements" in: *arXiv preprint arXiv:1211.5590* (2012) (cit. on p. 3)

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio "Theano: a CPU and GPU math expression compiler" in: *Proceedings of the Python for scientific computing conference (SciPy)* vol. 4 Austin, TX 2010, p. 3 (cit. on p. 3)

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems" in: *arXiv preprint arXiv:1512.01274* (2015) (cit. on p. 3)

[6] PyTorch core team *PyTorch* 2016 url: `http://pytorch.org/` (cit. on p. 3)

[7] 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* 2014, pp. 2672–2680 (cit. on p. 9)

[8] Martin Arjovsky, Soumith Chintala, and Léon Bottou "Wasserstein GAN" in: *arXiv preprint arXiv:1701.07875* (2017) (cit. on p. 9)