Assignment 4

TDT4173 Machine Learning and Case-Based Reasoning

Olav Markussen and Geir Kulia

April 6, 2017

1 Theory

1.1 What is the core idea of deep learning? How does it differ from shallow learning?

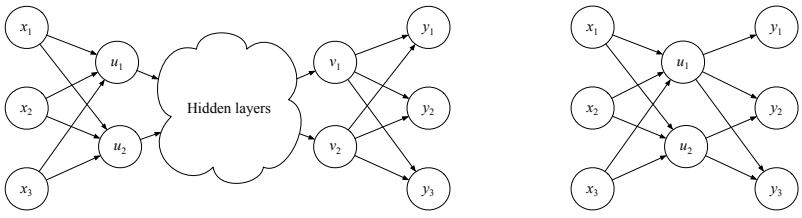
The core idea of deep learning is to have several hidden layers to describe complicated functions. This enables more complex representations, with the disadvantage of more computing power needed, as well as overfitting. Figure 1a shows a diagram of a deep neural network, where $\mathbf{u} = \{u_1, u_2\}$, $\mathbf{v} = \{v_1, v_2\}$, and the cloud denotes hidden layers.

A shallow network contains fewer, if any, hidden layers. It can represent all functions, but the number of parameters expands rapidly with the complexity of the function. It is less prone to overfitting. A shallow network is displayed in Figure 1b, where $\mathbf{u} = \{u_1, u_2\}$ is the only hidden layer.

1.2 Logistic regression as neural network

One can define the logistic regression implemented in Assignment 1 [3] as a neural network, where we have one hidden layer with one neuron. One first define the input to the neuron as

$$z = h(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} \tag{1}$$



(a) Example of a deep network

(b) Example of a shallow network

Figure 1: Deep and shallow networks.

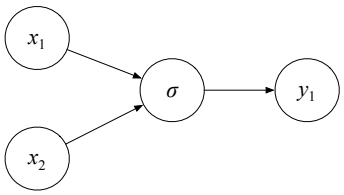


Figure 2: Example of a logistic regression neural network

The activation function is then defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

and the backpropagation algorithm is then

$$\frac{\partial E_{\text{ec}}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial E_{\text{ec}}(\mathbf{w})}{\partial \sigma(z)} \cdot \frac{\partial \sigma(z)}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{w}}. \tag{3}$$

The training will then be

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \eta \sum_{i=1}^{N} (\sigma(z) - y_i)\mathbf{x}_i \tag{4}$$

The final parameter is given by

$$y_1 = \sigma(z). \tag{5}$$

The topology of the network is shown in Figure 2.

1.3 Artificial neural network with only linear activation functions

Lets assume we have a linear activation function on the form

$$a(z) = C_1 \cdot z + C_2 \tag{6}$$

where C_1 and C_2 are constant. Traditionally, the backprogation algorithm would be defined as

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial a(z_j)} \cdot \frac{\partial a(z_j)}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_j} \tag{7}$$

where

$$E = \frac{1}{2} \sum_j (y_j - a_j) \tag{8}$$

and z_j is defined by Equation 1. The first differential is then given by

$$\frac{\partial E}{\partial a(z_j)} = -(y_j - a_j) \tag{9}$$

The second differential will be constant, as

$$\frac{\partial a(z_j)}{\partial z_j} = \frac{\partial}{\partial z_j}\big(C_1 \cdot z + C_2\big) = C_1 \tag{10}$$

The last differential is then

$$\frac{\partial z_j}{\partial w_j} = \frac{\partial}{\partial w_j}\big(w_j \cdot x_j\big) = x_j \tag{11}$$

This gives a backpropagation algorithm on the form

$$\frac{\partial E}{\partial w_j} = -C_1 x_j (y_j - a_j) = \big(C_1^2 x_j^2\big) w_j - \big(C_1 x_j y_j + C_1 C_2 x_j\big) \tag{12}$$

where $C_1^2 x_j^2$ and $-\big(C_1 x_j y_j + C_1 C_2 x_j\big)$ are constant with respect to w_j, so that

$$\frac{\partial E}{\partial w_j} = K_1 w_j + K_2 \tag{13}$$

Aforementioned implies that any transition through any node will be linear. A cascade of linear operations always results in a linear system, leading to the transfer function of the neural network to be linear as well.

1.4 Forward Pass for Network Topology

The first layer $\mathbf{u}$ is given by

$$\mathbf{u} = \sigma(\mathbf{W}^{(1)\mathrm{T}}\mathbf{x}) = [0.98, 0.67, 0.04]^{\mathrm{T}} \tag{14}$$

and the second layer will be

$$\mathbf{v} = \sigma(\mathbf{W}^{(2)\mathrm{T}}\mathbf{u}) = [0.57, 0.34, 0.56, 0.63]^{\mathrm{T}} \tag{15}$$

resulting in the output value

$$\mathbf{y} = \mathbf{W}^{(3)\mathrm{T}}\sigma(\mathbf{v}) = [0.66]^{\mathrm{T}} \tag{16}$$

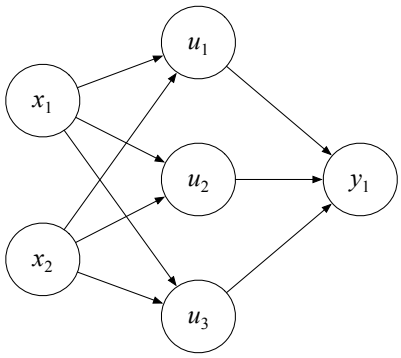
The values have been calculated using an attached python module.

2 Multilayer Perceptron

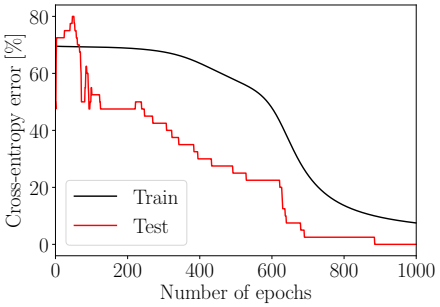
A multilayer perceptron was implemented by modifying the appended Tensorflow code for a XOR gate. The authors wanted to use as few neurons possible, so the Tensorflow Playground [4] was applied. An outline of the neural net implemented is shown in Figure 3a. The authors were able to achieve decent results with only one hidden layer! A learning rate chosen was

$$\alpha = 2.0 \tag{17}$$

The resulting cross-entropy error for training and testing is shown in Figure 3b.

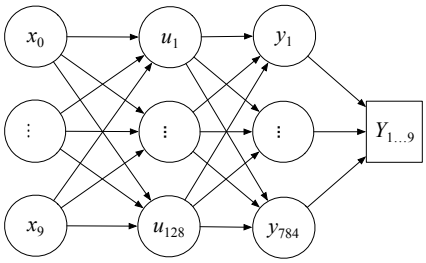


(a) Neural network used for Multilayer Perceptron

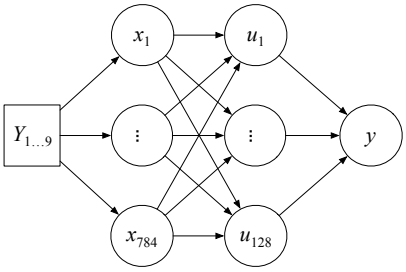


(b) Cross entropy error of training and test data

Figure 3: Neural network graph and cross entropy error for a Multilayer Perceptron.

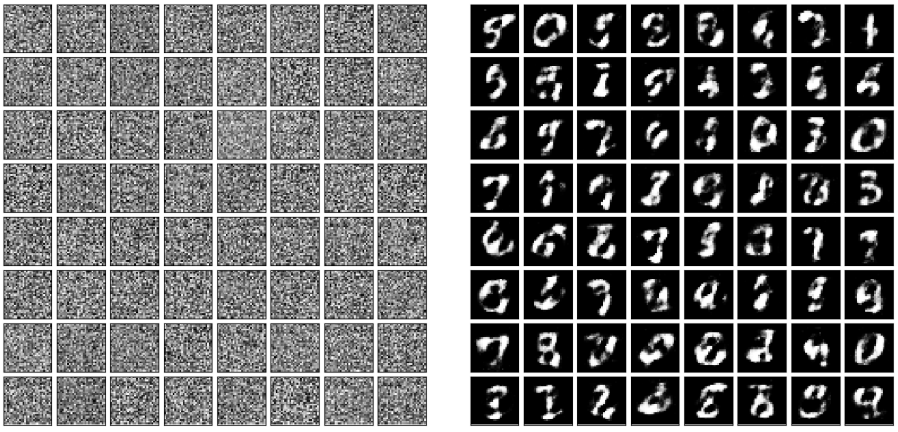


(a) Neural network used for the generator network **G**.



(b) Neural network used for the discriminator network **D**.

Figure 4: The Neural network of **G** and **D**.



(a) The output image of generator network **G** after 100 epochs.

(b) The output image of the generator network **G** after 40 000 epochs.

Figure 5: The output of a neural network **G** that generates digits based on Generative Adversarial Networks.

3 Generative Adversarial Networks

Searches for the previous implementation was used for the generator **G** and discriminator **D**. We were inspired by the tutorial. We tried with several layers but achieved the best result with two hidden layers. The width of the first hidden layer **u** was chosen to 128 [2]. Because the output picture has dimensions of 28 px times 28 px, the last layer **y** had to be $28^2 = 784$. The topology of the network G is shown in Figure 4b. The input **x** is given so that

$$\mathbf{x}(0) = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] \tag{18}$$

and

$$\mathbf{x}(4) = [0, 0, 0, 0, 1, 0, 0, 0, 0] \tag{19}$$

The output $\mathbf{Y_{0...9}}$ is a $28 \cdot 28$ picture that ideally illustrates a number corresponding to **x**. After several trials and errors it was decided to apply different activation functions. There are neither a matrix multiplication nor a bias correction between **y** and **Y**, only reorganization to a picture. The activation function for $\mathbf{u_G}$ is ReLU and Sigmoid for $\mathbf{y_G}$.

The discriminator **D** is applying the opposite action. It takes a picture $\mathbf{Y_{0...9}}$ as an input and then reorganizes it to **x**. It has only one hidden layer $\mathbf{u_D}$ of width 128. The discriminator **D** uses the activation function ReLU.

The discriminator **D** is trained on the MNIST dataset five times for each time the generator **G** train against **G**. It's important to use a sufficiently small number of training of **D** for each time **G** is trained. If this number is too large, then **G** will not be able to improve as **D** will discard all attempts by **G**. The

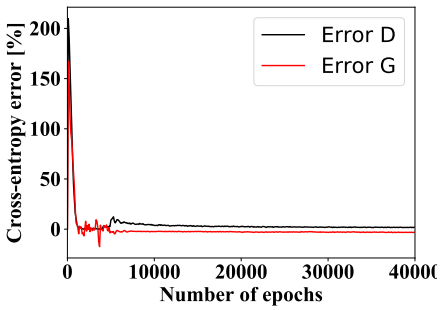


Figure 6: Cross-entropy error of **G** and **D**.

initial learning rate applied was chosen to

$$\alpha_0 = 1 \cdot 10^{-4} \tag{20}$$

The learning rate then decay exponentially so that

$$\alpha(i) = \alpha_0 \cdot 0.99^{\frac{i}{1000}} \tag{21}$$

where i is the epoch index and is increasing by 1 for each iteration. The batch size is 64. The final results for epochs of 100 and 40 000 is shown in Figure 5.

3.1 Cross-entropy error for G and D.

The cross-entropy error function for **G** is given by

$$E_{\mathbf{G}} = -E_{\text{rm}}(\hat{y}_f) \tag{22}$$

where subscript f indicates the estimated y by **D** for picture generated by **G**. f is short for fake. The cross-entropy error function for **D** is

$$E_{\mathbf{D}} = E_{\text{rm}}(\hat{y}_r) + E_{\mathbf{G}} \tag{23}$$

where $E_{\text{rm}}(\cdot)$ is the reduced mean error. Subscript r indicates estimation on real images from the discrimination training set. The cross-entropy errors $E_{\mathbf{G}}$ and $E_{\mathbf{D}}$ are shown in Figure 6. The error starts relatively low, as **D** is not yet well trained in discriminating generated photos. The error then decreases dramatically, as shown below

```
Epoch:   0 lr   0.0001
          Generator error:          0.0063119
          Discriminator error:     -0.0009150
Epoch:   100 lr   0.0001
          Generator error:          1.6718566
          Discriminator error:      2.0977683
```

After the initial rise, the error falls dramatically, as both **G** and **D** becomes better. **G** oscillates around **D** for a time as shown below

```
Epoch: 200 lr 0.0001
        Generator error:        1.4765003
        Discriminator error:    1.9077775
Epoch: 300 lr 0.0001
        Generator error:        1.1227739
        Discriminator error:    1.6191332
Epoch: 400 lr 0.0001
        Generator error:        0.9557722
        Discriminator error:    1.3049076
```

This is because $E_{\mathbf{G}}$ is highly dependent on the discrimination capabilities of **D**. Both of the cross-entropy errors E stabilizes around 0. The whole trend is shown in Figure 6.

3.2 What type of learning is GAN?

According to OpenAI, which the authors find as a credible source, the GAN is a unsupervised learning algorithm [1]. **G** is unsupervised as it doesn't get any feedback from real cases, as they does not apply. However, the authors of this report will argue that **D** is semisuprvised as it gets a boolean value from several real examples yet adjusted by the cross-entropy error of **G**, as shown in Equation 23.

References

[1] greg brockman peter chen vicki cheung rocky duan ian goodfellow durk kingma jonathan ho rein houthooft tim salimans john schulman ilya sutskever wojciech zaremba Andrej Karpathy, pieter abbeel. Generative Models. `https://blog.openai.com/generative-models/`, 2016. [Online; accessed 3 April 2017].

[2] Agustinus Kristiadi. Wasserstein GAN implementation in TensorFlow and Pytorch. `http://wiseodd.github.io/techblog/2017/02/04/wasserstein-gan/`, 2017. [Online; accessed 1 April 2017].

[3] Aleksander Rognhaugen. TDT4173: Machine learning and case-based reasoning - assignment 1. *NTNU*, 2017.

[4] Daniel Smilkov and Shan Carter. Tensorflow Playground. `http://bit.ly/2ongVBd`, 2016. [Online; accessed 1 April 2017].