TDT4173: Machine Learning and Case-Based Reasoning

Assignment 4

Amar Jaiswal

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

March 7, 2019

- **Delivery deadline: April 05, 2019** by 23:59.
- **This assignment counts towards 4 % of your final grade.**
- You can **work on your own or in a group** (maximum two persons) but must submit the assignment individually on Blackboard. Don't forget to mention your and your group member's **name and username** on the top of your report.
- Deliver your solution on *Blackboard* before the deadline.
- Please upload your report as a separate PDF file, and package your project file (code) and screenshots into an archive (e.g. zip, rar, tar).

Objective: Gain insight into (a) core concepts of ensemble learning methods, (b) learning which method to choose, (c) implementation of basic algorithms (k-NN, AdaBoost and basic Neural Network) from scratch.

1 Theory [1.9 points]

1.1) **[0.5 points]**

What is the core idea of **deep** learning? How does it differ from **shallow** learning? Support your arguments with relevant examples.

1.2) **[0.8 points]**

Describe the comparison between following machine learning techniques: **k-NN**, **decision tree**, **SVM** and **deep learning**. Also, discuss the situations, for each of these techniques, as **why** and **when NOT** to prefer them.

1.3) **[0.6 points]**

Discuss when should we use ensemble methods in context of machine learning? Explain briefly any 3 types of ensemble machine learning methods.

2 Programming [2.1 points]

Solve any 3 out of 6 problems described below.

2.1) **[0.7 points] k-NN implementation from scratch**

Implement **k-nearest neighbour** (k-NN) algorithm from scratch. Reuse this k-NN program to implement k-NN regression (could be based on simple mean) and classification (could be based on voting). Display the results (including all 10 nearest neighbours), where *k=10* for the 124$^{\text{th}}$ example from the given dataset (i.e. $X[124]$ = [6.3, 2.7, 4.91] for regression and [6.3, 2.7, 4.91, 1.8] for classification) as your test sample. The data set for regression and classification are in *knn_regression.csv* and *knn_classification.csv* files respectively. **The data set is not indexed, and has column names for features as $[x_1, x_{x2}, ..., x_n]$, and the target as *y*.**

Input :

- A training set $S = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m))$.

Initialization :

- Maximum number of iterations T;
- initialize the weight distribution $\forall i \in \{1, \ldots, m\}, D^{(1)}(i) = \frac{1}{m}$.

for $t = 1, \ldots, T$ **do**

- Learn a classifier $f_t : \mathbb{R}^d \to \{-1, +1\}$ using distribution $D^{(t)}$
- Set $\epsilon_t = \sum_{i: f_t(\mathbf{x}_i) \neq y_i} D^{(t)}(i)$
- Choose $a_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$
- Update the weight distribution over examples

$$\forall i \in \{1, \ldots, m\}, D^{(t+1)}(i) = \frac{D^{(t)}(i) e^{-a_t y_i f_t(\mathbf{x}_i)}}{Z^{(t)}}$$

where $Z^{(t)} = \sum_{i=1}^{m} D^{(t)}(i) e^{-a_t y_i f_t(\mathbf{x}_i)}$ is a normalization factor such that $D^{(t+1)}$ remains a distribution.

Output : The voted classifier $\forall \mathbf{x}, F(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} a_t f_t(\mathbf{x})\right)$

Figure 1: Pseudo-code for AdaBoost from Lecture slides.

2.2) **[0.7 points] AdaBoost implementation from scratch**

Implement the AdaBoost algorithm from scratch for binary classification. Pseudo-code is displayed in Figure 1. You may choose any classifier (initially prefer *DecisionTreeClassifier* from sklearn[1] with *maximum_depth* = 1) from the library to be used by your AdaBoost implementation, provided it should accept *sample_weight* as one of the parameters. The training and test data set are in *adaboost_train.csv* and *adaboost_test.csv* files respectively. The data set is indexed. Target values are under column name *y* and feature or attribute values are under column names x_1, x_2, ..., x_{10}. Display and comment on the error rate with respect to the number of iterations used for your AdaBoost algorithm. ***For pseudo-code details, please refer to Lecture 10 slides, by Hai Nguyen, it is available on the Blackboard after the respective lecture.***

[1] http://scikit-learn.org/

The exercises 2.3), 2.5) and 2.6) are centered around the toy data set of classifying hand written digits. The data set can be retrieved from *load_digits*[2] using sklearn. The data set contains 1797 greyscale 8×8 images and corresponding labels (a number between 0 to 9 per image). Although images usually are represented in two dimensions (height and width), with *load_digits* each image constitutes a single feature vector of 64 elements.

In order to obtain training and test data, we first have to split the data set between training and test data. One can use the *train_test_split*[3] API from python's sklearn for this purpose. By default, for *train_test_split*, we will keep 25% of the data as test set.

2.3) **[0.7 points] k-NN vs SVM vs Random Forest on sklearn digit data set**

In this task, you are allowed to use library implementations of [**k-NN**, **SVM**, and **RandomForest**] classifiers and train them on the data set from sklearn for *load_digits*. Do the prediction on the test data and plot the confusion matrix for "target test data" vs "predicted values" for every classifier listed above. Where "target test data" is y label from the test sample and "predicted values" are the result of prediction from individual classifiers. Include the confusion matrix and the used classifier's signature in the report. Document any exciting findings you may have encountered.

In the upcoming exercises, we are going to get hands-on experience with implementing artificial neural networks. Question 2.4) will address the problem of defining the core functionality of a neural network. Having implemented this, we are going to analyse how it works, specifically by solving the XOR problem. In part 2.5) and 2.6), we are moving towards deep learning by considering a more advanced problem, requiring a network architecture that is more complex. We will consider the same digits data set as in task 2.3) and are going to compare two different approaches with respect to this task. Ideally, the methods developed in 2.4) should be as generic as possible such that they can be reused when implementing the logic needed for solving questions 2.5) and 2.6).

To begin with, we will briefly discuss the equations that are needed for the optimisation of a neural network. In general, we are concerned with two main sub-processes. The forward pass (inference step), with propagation of numerical values from the input layer to the output layer of the network, and the backward pass dealing with computations of gradients and backpropagation of these in order to update the parameters (i.e. weights and biases) of the different layers of the network.

For the next questions, we will focus on a feedforward neural network conceptually similar to the one that is displayed in Figure 2. As indicated by the caption, this network consists 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* 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 $\mathbf{W}^{(l)}$, 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 $g(\mathbf{a}^{(l\text{-}1)} \cdot \mathbf{W}^{(l)})$, 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 2 we can calculate the activations for layer $\mathbf{u}$ by computing $g(\mathbf{x} \cdot \mathbf{W}^{(1)})$, assuming $\mathbf{x}$ is a row vector. This makes sense, because $\mathbf{x}$ is a 1×4 row vector and $\mathbf{W}^{(1)}$ is a 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.

Based on the output $\mathbf{y}$ of the network, we would like to get some intuition of whether the predictions of the neural network are right or wrong. A loss function computes the error with respect to the predicted output values $\mathbf{y}$ and the target values (desired output) $\mathbf{y}_\mathrm{t}$.

[2] http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html
[3] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

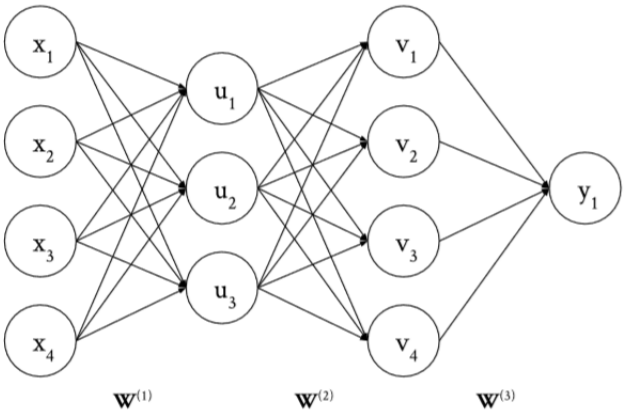


Figure 2: A simple feedforward neural network with four input neurons $\mathbf{x}$, two hidden layers $\mathbf{u}$ and $\mathbf{v}$ and one output neuron $\mathbf{y}$. The weight matrices connecting each of the layers have been indicated as $\mathbf{W}^{(l)}$. The network does not have any *bias* neurons.

The loss determines how the weights of the network should be updated. More specifically, if we are computing the gradients of the loss with respect to the weights, we would be able to estimate the direction of steepest increase in the loss based on these specific weight values. In order to minimise the loss, we therefore adjust the values of the weights a small step α (i.e. learning rate) in the opposite direction of the steepest increase.

This process of calculating the gradients of the loss with respect to the weights of layer l also requires the gradients of the layers later in the network to be computed. Hence, we need to backpropagate the error such that we in the end are able to obtain the gradient of the loss with respect to the weights. To illustrate this principle, we consider the network architecture in Figure 2. $\nabla_{\mathbf{W}^{(3)}} L$, the gradient of the loss (L) with respect to weights $\mathbf{W}^{(3)}$ is by the definition of the chain rule composed of the gradient of the loss L with respect to the outputs $\mathbf{y}$, the gradient of $\mathbf{y}$ with respect to the weighted input $\mathbf{v} \cdot \mathbf{W}^{(3)}$ and finally the gradient of $\mathbf{v} \cdot \mathbf{W}^{(3)}$ with respect to the weights $\mathbf{W}^{(3)}$. If we denote $\mathbf{v} \cdot \mathbf{W}^{(3)}$ as $\mathbf{z}^{(3)}$ and at the same time consider $\mathbf{y}$ as the activations of the final layer, in this case $\mathbf{y} = \mathbf{a}^{(3)}$, then this chain of gradients can be compactly represented as:

$$\nabla_{\mathbf{W}^{(3)}} L = (\nabla_{\mathbf{W}^{(3)}} \mathbf{z}^{(3)})^T \cdot (\nabla_{\mathbf{a}^{(3)}} L \nabla_{\mathbf{z}^{(3)}} \mathbf{a}^{(3)})$$

Generally speaking, if k is the final layer of the neural network the formula can be transformed to:

$$\nabla_{\mathbf{W}^{(k)}} L = (\nabla_{\mathbf{W}^{(k)}} \mathbf{z}^{(k)})^T \cdot (\nabla_{\mathbf{a}^{(k)}} L \nabla_{\mathbf{z}^{(k)}} \mathbf{a}^{(k)})$$

The term $\nabla_{\mathbf{a}^{(k)}} L \nabla_{\mathbf{z}^{(k)}} \mathbf{a}^{(k)}$ is often referred to as $\delta^{(k)}$ (Equation 1) and can be interpreted as expressing the error of layer k. The vector of $\delta^{(k)}$ has the same shape as the vectors $\mathbf{z}^{(k)}$ and $\mathbf{a}^{(k)}$.

$$\delta^{(k)} = \nabla_{\mathbf{a}^{(k)}} L \nabla_{\mathbf{z}^{(k)}} \mathbf{a}^{(k)} \tag{1}$$

For a layer m that is not the output layer, $\delta^{(m)}$ can be computed from the gradients higher up (closer to the output) in the network.

$$\delta^{(m)} = (\delta^{(m+1)} \cdot (\nabla_{\mathbf{a}^{(m)}} \mathbf{z}^{(m+1)})^T) \nabla_{\mathbf{z}^{(m)}} \mathbf{a}^{(m)} \tag{2}$$

The gradient of the loss with respect to the weights of an arbitrary layer l can thus be calculated from Equation 3 and new weights $\mathbf{W}^{(l)\prime}$ are obtained according to Equation 4.

$$\nabla_{\mathbf{W}^{(l)}} L = (\nabla_{\mathbf{W}^{(l)}} \mathbf{z}^{(l)})^T \cdot \delta^{(l)} \tag{3}$$

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

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

$$\mathbf{W}^{(l)\prime} = \mathbf{W}^{(l)} - \alpha \nabla_{\mathbf{W}^{(l)}} L \tag{4}$$

For simplicity of derivation, the activation function will be defined as the logistic function $g = \sigma(z) = \frac{1}{1+e^{-z}}$ while we will be using the squared loss $L = \frac{1}{2}(\mathbf{y}_\mathrm{t} - \mathbf{y})^2$ for computing the error of the output of the network.

In this exercise, you will be implementing the neural network logic from scratch, meaning that you are not allowed to utilise any deep learning frameworks such as TensorFlow[4], Keras[5] or Caffe[6] where this functionality is pre-defined. In other words, you are going to develop code for the operations required in the forward and backward pass of the optimisation process of a neural network. It is only allowed to use libraries that perform the low-level math and matrix operations, meaning the exponential function e^x, element-wise multiplication and the dot product of two matrices. The robustness of the implementation will be tested and verified by solving the XOR problem.

In short terms, 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 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.

2.4) **[0.7 points] Neural Network from scratch**

A fully-connected neural network is able to solve the nonlinear XOR problem, but what is the minimal number of layers and neurons within these layers necessary? Back up your answer with an illustration of the network architecture and an explanation why you believe this is the correct network architecture. Subsequently, develop the functionality required for training this neural network based on the theory that we discussed earlier. As part of this you will need to derive functions for the gradients expressed in Equations 1 to 4. Finally, train the network for the amount of rounds (epochs) that you feel appropriate and plot how the loss develops during training. Both the network architecture and graphical plot of error should be included in the report.

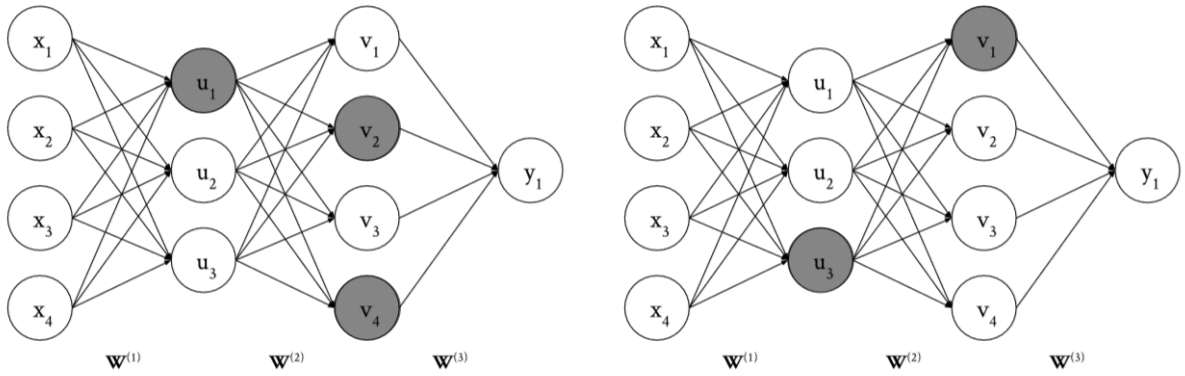
The subsequent exercises are based on the functionality developed in task 2.4).

2.5) **[0.7 points] Deeper Neural Network**

Develop a fully-connected neural network for solving the problem of classifying handwritten digits. The parameters should be tuned using the aforementioned digits training set. The implementation should be developed from scratch and you are urged to reuse functionality developed in 2.4). In the report, document the neural network architecture that you are utilising, what hyperparameters (i.e. learning rate, number of epochs etc.)) you used and explain any interesting findings you may have. Additionally, display how well your neural network performs on the test data. For this purpose, a confusion matrix would be a suitable option.

---

[4] https://tensorflow.org/
[5] https://keras.io/
[6] https://caffe.berkeleyvision.org/



(a) When the first training example is displayed to the network, neurons u_1, v_2 and v_4 are randomly selected to be dropped out.

(b) In the case of the second example being passed to the network, neurons u_3 and v_1 are not participating in the forward pass by being deactivated.

Figure 3: A simple feedforward neural network performing dropout. The network architecture is the same as of the network in Figure 2. We can observe that different subsets of neurons are activated for each of the two examples that are presented to the network.

In the final exercise, we will be extending the neural network logic of the two former tasks. We are going to apply a deep learning technique called dropout[7]. By utilising dropout we aim to construct a more robust classifier than the neural networks built in the former tasks.

As we have seen earlier in the course, it might happen that a model overfits a training set by not only learning patterns relevant for the task but rather memorises the cases of the training data. Particularly, in deep learning overfitting often occurs as a result of the large pool of parameters that should be optimised.

Dropout addresses this issue by deactivating neurons of the hidden layers of the neural network randomly such that some unlucky neurons are not allowed to participate in the forward pass during that particular iteration. For each training example passed to the neural network each hidden neuron is deactivated by a probability equal to the *dropout_rate*. In other words, a randomly selected subset of hidden neurons is deactivated each time and correspondingly the part of the neural network that plays the role of predicting output values varies from time to time. As a result, dropout can be considered a way to perform ensemble learning with artificial neural networks. It is important to note that dropout only is applied during training, meaning that all neurons participate in producing output in the testing phase.

To get an intuition of how dropout works, consider the neural networks of Figure 3. The hidden neurons that are coloured grey are deactivated while all other neurons contribute in producing the output $\mathbf{y}_1$. Let $\mathbf{d}^{(l)}$ denote the dropout vector defining which neurons of layer l that are activated (1) and which that are deactivated (0). Vector $\mathbf{d}^{(l)}$ is of the same shape as $\mathbf{a}^{(l)}$. When the first training example is displayed to the network, the network of Figure 3a is applied, yielding dropout vectors of $\mathbf{d}^{(1)} = [1 \;\; 0 \;\; 0]$ and $\mathbf{d}^{(2)} = [0 \;\; 1 \;\; 0 \;\; 1]$. For the neural network of the second example (Figure 3b), dropout vectors are given as $\mathbf{d}^{(1)} = [0 \;\; 0 \;\; 1]$ and $\mathbf{d}^{(2)} = [1 \;\; 0 \;\; 0 \;\; 0]$.

Compared to ordinary neural networks, neural networks applying dropout only require minor modifications. First of all, the output activation $\mathbf{a}^{(l)}$ must take into account the dropout vector. This ensures that only the desired neurons are activated during the forward pass. Additionally, the output activation is multiplied by a factor of $\frac{1}{1-dropout\_rate}$ to ensure that the expected scaling of the outputs is maintained when test data is presented to the network. The modified formula of the output activation $\mathbf{a}^{(l)}$ is presented in Equation 5.

$$\mathbf{a}^{(l)} = \frac{1}{1 - dropout\_rate}(\mathbf{d}^{(l)}(g(\mathbf{a}^{(l\text{-}1)} \cdot \mathbf{W}^{(l)}))) \tag{5}$$

[7]A more extensive introduction to the dropout technique is provided by Andrew Ng at https://coursera.org/learn/deep-neural-network/lecture/eM33A/dropout-regularization

2.6) **[0.7 points] Dropout Technique**

Extend the neural network implemented in programming task 2.5) by utilising the dropout technique. Apart from modifying the output activation $\mathbf{a}^{(l)}$ and implementing logic for dropout vectors $\mathbf{d}^{(l)}$, the neural network should remain the same. Although, you are free to choose whatever *dropout_rate* above 0 that you find appropriate, we recommend you to start with a *dropout_rate* of 0.2. Explain the hyperparameters (learning rate, number of epochs and dropout rate) that you ended up utilising for this task. Train the network on the aforementioned training data and record the performance of the model on the test images. The resulting confusion matrix, or other suitable evaluation measure, should be provided in the report. How did this network perform compared to the network of exercise 2.5)? Document any interesting findings you may have.

Acknowledgment

The programming tasks 2.4, 2.5, and 2.6 are based on the original work of Daniel Groos (for TDT4173 in year 2018).
