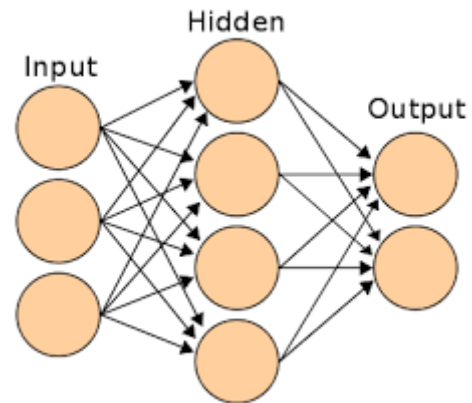


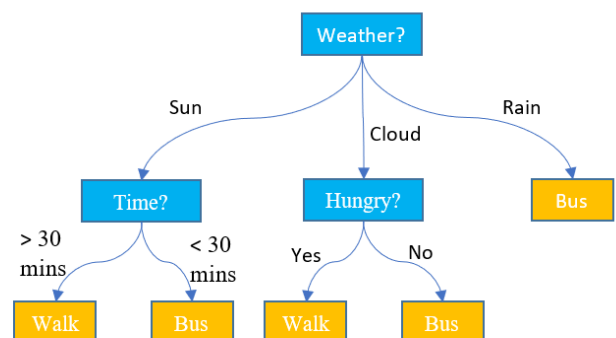
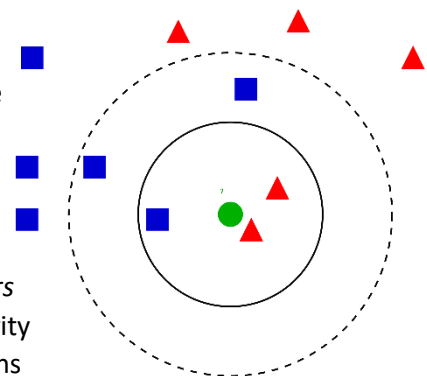
Assignment 3

Theory

1. Deep learning is a paradigm within the machine learning domain. It is heavily inspired by the human brain as seen from the perspective of a neuroscientist. In deep learning one constructs neural networks, they can be represented like the image to the right. Deep Learning is characterized by taking raw data and learning based on that, whereas shallow learning is dependent on the programmer to code it with some prior knowledge. Shallow learning, often as symbolic methods, needs explicit definitions of parameters and properties of the domain, where deep learning algorithms might just take a raw image and make its own understanding of properties like shapes and objects over time. For the sake of an example, it's hard to learn a shallow-learning model to classify a giraffe based on a picture, this is a domain where a deep neural network (preferably a convolutional network) would thrive. But if one is trying to build a model for deciding whether or not to go to the cinema today, a symbolic based method with coded parameters would probably outperform a neural network. Often one wants symbolic, shallow-learning models to take choices which we believe should be taken consciously, while implicit knowledge like object detection is often done better with deep learning models.

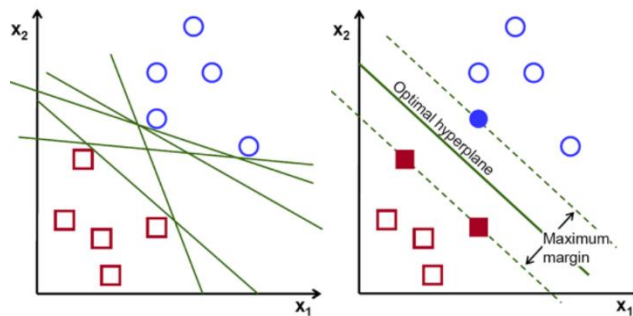


2. **k-NN** is short for **k Nearest Neighbours**. The idea behind the algorithm is that you have training samples where you know the classification of each element. One has to find the attributes that are of interest for all elements and place them in a mathematical multidimensional room. When the algorithm is classifying a new element, it places that new element in the same multidimensional room according to the attributes and finds the *k nearest neighbours* to that element. Among the *k nearest neighbours*, the majority class is probably the same class that the element in question belongs to. In the example to the right, the green circle is the unknown element to classify. *K* is set to be 3, so the algorithm finds the 3 nearest neighbours. From the image, one can see that out of the 3 neighbours, there are 2 triangles, so the unknown element is likely to be a triangle. Some side notes: might be smart to have *k* assigned as an odd number to avoid draws between majorities, though this can happen anyway when there are more than 2 classes. This algorithm is also called *lazy algorithm* because it does all



calculation during prediction, it does not need any training because it has to look at all the sample data upon prediction anyway.

Decision Trees can be represented like in the image to the right. By declaring attributes for elements in the domain and looking at historical data one can construct decision trees in order to classify. Each attribute has a set of legal values and when the value of this element has a certain value, that leads the decision tree into either a concluding stage or another decision node. The constructing of these decision trees is not done *lazy* like in k-nn, because



one can construct the trees from just looking at historical data, and when predicting for a new element one just has to follow the path from the root of the tree down throughout the tree in order to find a decision.

SVM is short for **Support Vector Machine**. Its about finding a way to differentiate 2 classes apart from each

other, but in a more efficient way than looking at all samples, by doing a trick of finding *support vectors*. Below is an illustration that shows how the SVM architecture is build. The idea behind SVM is also that one doesn't only want to find a hyperplane that can divide the two object-classes for the training samples, but one that maximizes the probability for correct classifications for unseen objects as well. So finding the hyperplane that creates the largest margin to both classes is the general strategy. In task one I already defined **Deep Learning** with neural networks.

So all of the mentioned techniques use some sort of ahead of time computation/training except k-nn, which is only active during the prediction. SVM is different from the other techniques in the regard that it is the only one that has a limitation to only be able to differentiate between 2 classes, while all the other techniques can distinguish between arbitrary number of different classes. Deep learning thrives with raw data whereas the other is designed to work with explicitly encoded parameters.

3. There are quite a few paradigms within machine learning and a lot of different algorithms within each paradigm. They all have advantages and disadvantages as is reflected in the previous section. Some domains are atomic and contained enough to only apply a single algorithm, others are so complex that the advantages of several algorithms and paradigms are needed to do the correct action. So to summarize, we can apply ensembled methods to create models that are more robust for noise and even use them to make more complex models which can handle all from raw data to cognitive choices in a single pass.

Below is 3 examples of ensembled methods¹

- a. Combine multiple algorithms and select the prediction that was the most popular. Lets say you cobine 10 algorithms that all will perform a prediction on a given problem. When they all had there say, select the option that received the most votes (for example in a classification problem).
- b. Combine multiple algorithms and select the average of the predictions. This solution does not fit for a classification problem, but can be used in other types of predictions. Lets say the domain is to figure out wether a written review of a movie is positive or negative on scale of 1 to 10. If we combine two different algorithms and they end up with predicting respectively that the reviews scored 7 and 9, then averaging the two the model will say the review is probably an 8.

- c. Combining methods and assign weights. This method can be used as an extra effort with both of the previously mentioned methods. We can have stronger belief in some of the algorithms than other, but if many of the weaker algorithms suggest a different prediction than the one strong, perhaps they are right. To obtain this kind of behaviour we can assign weights to the different algorithms so they have different impact on the final result.

More complex ensembled methods can be made, for instance with autonomous vehicles one might want to use sub symbolic models like neural networks to perform the visual impressions, but let symbolic, cognitive models decide the actions to perform based on the visual interpretation that the neural network did.

Programming

We chose to do 2 of the neural network tasks and the kNN task in this assignment. The tasks share quite a lot of the code since we tried to develop a generic system where it is easy to almost declaratively create new networks and train on arbitrary tasks, much like Keras and pytorch.

Task 2.4 – The XOR-Problem

Since the XOR-Problem is nonlinear it is not possible to solve with a single layer. We believe that it is enough with a network made up from 2 hidden nodes and 1 output node to solve the problem. Though neural networks make up their reasoning in weird ways there are multiple ways we can argue that a network with said characteristics can manage to solve the xor-problem. For instance, 1 of the hidden nodes can learn to discriminate where both input signals are 0, and the other can learn to discriminate when both signals are 1, the output node can then learn that when none of the previous nodes fire an high output, the values are neither [0,0] or [1,1] and then the output should be 1. As mentioned, though this seems like one logical solution to a human, it is probably not like this the neural network will evolve, but it serves as a proof that the chosen architecture is capable of learning XOR-classification.

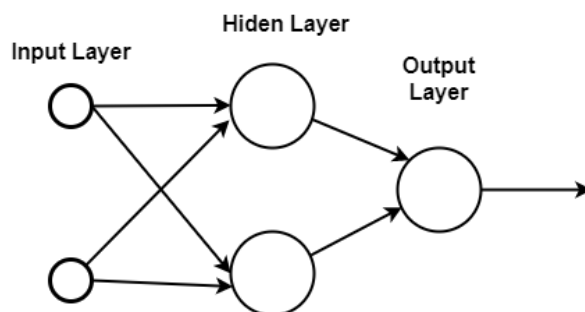


Figure 1 XOR-Neural Network Architecture

Implementation

The code in its entirety is attached to the assignment delivery.

During training we used the cross-entropy loss function. Every adjustment to the network during training is done by differentiating the loss function with respect to each component in the network. Doing this differentiation for every component on its own would be very computational heavy, luckily all these differentiations are possible to do vectorized as well.

$$\frac{dL}{dw^{k-1}} = \underbrace{\left(\frac{dL}{da^k} \cdot \frac{da^k}{dz^k} \cdot \frac{dz^k}{da^{k-1}} \cdot \frac{da^{k-1}}{dz^{k-1}} \right)}_{\delta^{k-1}} \cdot \frac{dz^{k-1}}{dw^{k-1}}$$

The example above shows how the deltas from the layers deeper in the network is used over and over in the chain rule differentiation in the backpropagation. Therefore we calculate the delta for each layer and store them before adjusting the weights and biases in order to reuse the calculation of each delta.

```
last_delta = hidden_error * network[i].activation_derivative(activations[i+1])
```

$$L(a^k) = \frac{1}{2}(t - y)^2 \Rightarrow \frac{dL}{da^k} = (y - t)$$

The equation above shows the loss function as well as its derivative. 't' is the truth value and 'y' is the prediction done by the network.

$$a(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \Rightarrow \frac{da^k}{dz^k} = a(z)(1 - a(z))$$

The Equation above shows the activation function which is used in every layer, it is called the sigmoid function. The equation on the right side shows the derivative of the sigmoid.

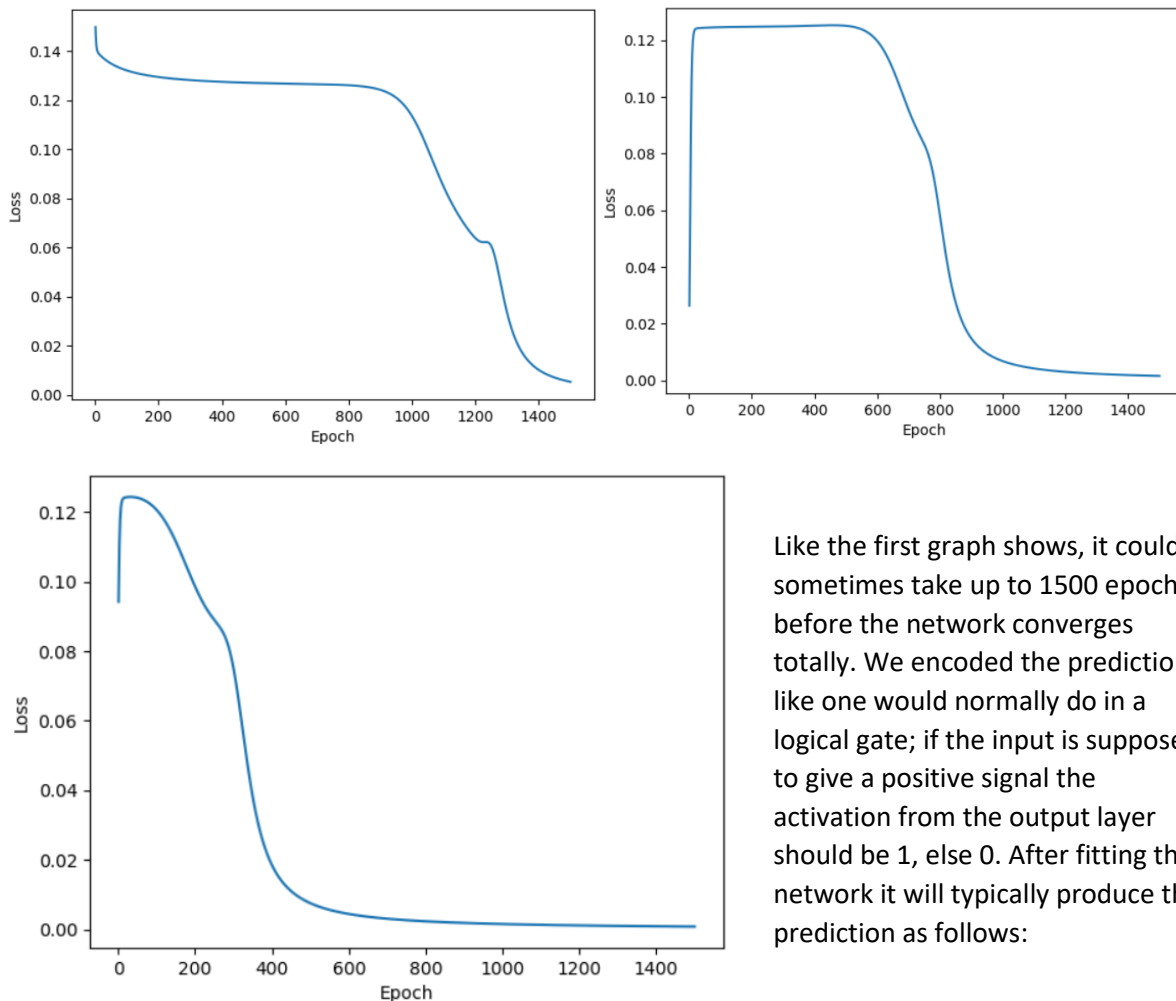
$$z^k(w, b, a^{k-1}) = a^{k-1} \cdot w^k + b^k \Rightarrow \frac{dz^k}{da^{k-1}} = w^k$$

The last equation is the messiest one. It is the derivative of the aggregate with respect to the activation from the previous layer. Like mentioned before these differentiations works well within the vectorized version.

```
#Train the network
learning_rate = 0.8
epochs = 1500

print("Training the network\nlearning rate: {}\nepochs: {}\nTraining...\n".format(learning_rate, epochs))
losses = fit(net, train_data, train_target, cross_entropy, cross_entropy_derivative, learning_rate, epochs)
```

To our great surprise we found that the network converged fastest with a learning rate around 0.8. It was also a surprise to us that it sometimes needed around 1000 epochs before convergence. The loss graph came in a bunch of varieties, here are some examples.



Like the first graph shows, it could sometimes take up to 1500 epochs before the network converges totally. We encoded the prediction like one would normally do in a logical gate; if the input is supposed to give a positive signal the activation from the output layer should be 1, else 0. After fitting the network it will typically produce the prediction as follows:

(0,0) : 0.04054398

(0,1) : 0.95478159

(1,0) : 0.96108093

(1,1) : 0.03614393

If this were to be used in any hypothetical deployed version, it would be satisfactory because the 1s are far above 0.5 and the 0s are far less than 0.5 and we could easily apply a filter to round the final prediction either way. This implementation proved that our network was sufficient and smart enough to learn the XOR-calculation by itself.

Task 2.5 – Handwritten Digits Classification

Most of the code written to fulfil task 2.4 is reused in this task. All the code is attached in the same file as task 2.4.

We implemented batches in order to speed up the training. This means that we only adjust the weights after predicting and evaluating a whole batch of data. We found great success when setting the batch-size to 128.

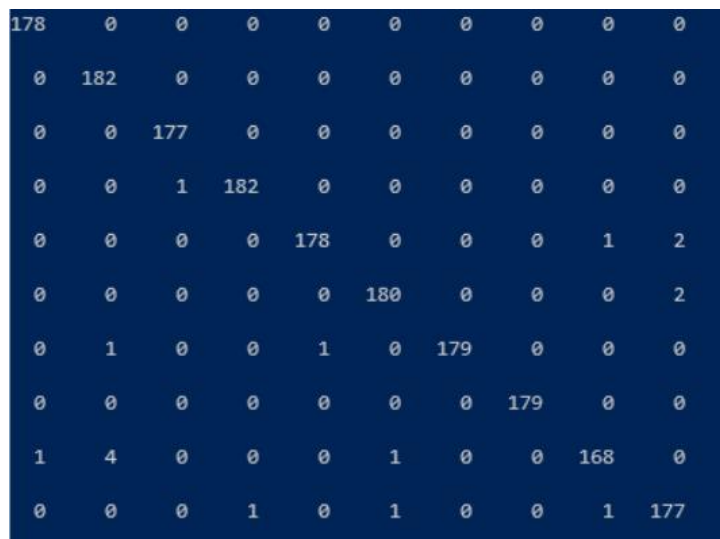
The hyper parameters are as follows;

Learning Rate : **0.001**

Epochs: **2000**

Batch Size: **128**

Since the images consisted of 8x8 grayscale pixels, the input layer has 64 nodes. We chose to have a hidden layer consisting of 32 nodes, fully connected. Since the network should classify 10 different classes (numbers from 0 through 9) the output layer consists of 10 nodes. Even though this is a small network it performed quite well on the dataset.



178	0	0	0	0	0	0	0	0	0
0	182	0	0	0	0	0	0	0	0
0	0	177	0	0	0	0	0	0	0
0	0	1	182	0	0	0	0	0	0
0	0	0	0	178	0	0	0	1	2
0	0	0	0	0	180	0	0	0	2
0	1	0	0	1	0	179	0	0	0
0	0	0	0	0	0	0	179	0	0
1	4	0	0	0	1	0	0	168	0
0	0	0	1	0	1	0	0	1	177

The image to the left shows the confusion-matrix for the network done on the training data. It is near perfect performance on this data.

Task 2.1 – K-Nearest Neighbour

All code is attached in the delivery. Our classification is done by voting and the regression is done by simple averaging the k nearest neighbours.

```

Fetching element 124 and performing classification...
10 nearest neighbours in random order:
[6.2, 2.8, 4.8, 1.8, 2.0]
[6.3, 2.8, 5.1, 1.5, 2.0]
[6.3, 2.7, 4.9, 1.8, 2.0]
[6.0, 2.7, 5.1, 1.6, 1.0]
[6.4, 2.7, 5.3, 1.9, 2.0]
[6.5, 2.8, 4.6, 1.5, 1.0]
[6.1, 3.0, 4.9, 1.8, 2.0]
[6.3, 2.5, 5.0, 1.9, 2.0]
[6.0, 3.0, 4.8, 1.8, 2.0]
[6.3, 2.5, 4.9, 1.5, 1.0]
Classification by vote shows 2.0

```

Image above shows the result of running classification on element 124.

```
Fetching element 124 and performing regression...
10 nearest neighbours in random order:
[6.3, 2.5, 4.9, 1.5]
[6.1, 2.8, 4.7, 1.2]
[6.3, 2.8, 5.1, 1.5]
[6.1, 3.0, 4.9, 1.8]
[6.1, 2.9, 4.7, 1.4]
[6.5, 2.8, 4.6, 1.5]
[6.3, 2.5, 5.0, 1.9]
[6.3, 2.7, 4.9, 1.8]
[6.2, 2.8, 4.8, 1.8]
[6.0, 2.7, 5.1, 1.6]
Regression by average shows: 1.6000000000000003
```

Image above shows the results from running regression on element 124.

ⁱ <https://towardsdatascience.com/simple-guide-for-ensemble-learning-methods-d87cc68705a2>