	1	2	3	Σ
Benedikt Hopf Alireza Ketabdari				

Exercise Sheet Nr. 5

(Deadline January 25, 2021)

Problem 1

a) A Clique $C = (V', E')$ in an undirected Graph $G = (V, E)$ is a subgraph of G such that

$$V' \subseteq V \text{ and } E' = \{\{v, w\} | v, w \in V'\} \subseteq V$$

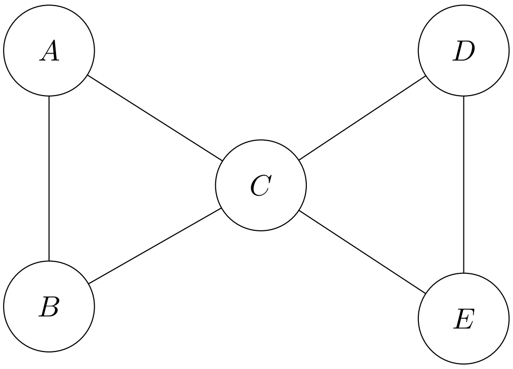
So a Clique is a subset ob nodes such that every one of these nodes is connected to every other.

A maximal Clique is a Clique, such that adding any other node from V to V', would cause it to not be a Clique anymore. So if $C = (V', E')$ is a Clique, but

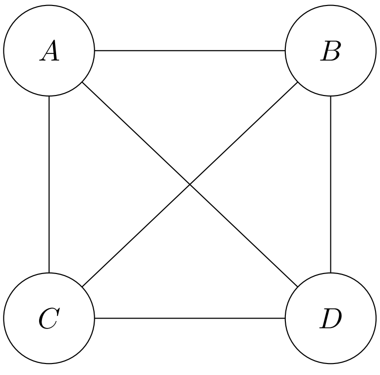
$$\forall x \in V \setminus V' : V' \cup \{x\} \text{ and } E' = \{\{v, w\} | v, w \in V' \cup \{x\}\} \not\subseteq E$$

is not a Clique (since $E' \not\subseteq E$).

The Graph below has two maximal Cliques, given by the nodes $\{A, B, C\}$ and $\{C, D, E\}$. Larger Cliques are not possible since A and B are not connected to D and E. Therefore the size of the Cliques is 3 for both maximal Cliques. Note that, while these are the only **maximal** Cliques, there are more non-maximal Cliques, such as for example $\{A, B\}$.



The next graph is completely one Clique (which is therefore obviously maximal), which consists of all four nodes.

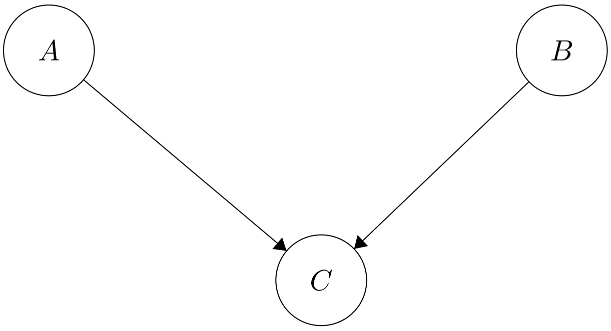


b) In general we cannot necessarily do that without loss of information since directed and undirected models cannot represent all the same distributions (see slide 11). This makes sense, since a directed model imposes a direct factorization on the distribution, whereas an undirected does not. We can still convert, the graph, though, as follows:

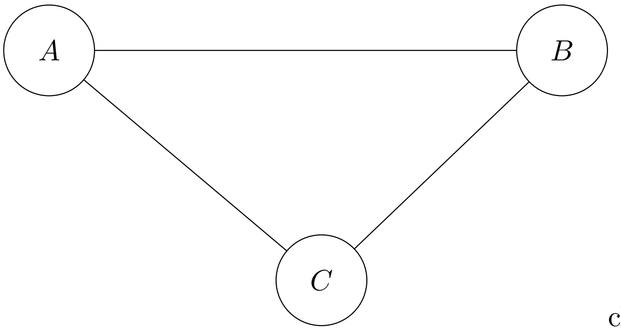
To avoid loss of information there are some steps we should consider to transform directed graph to the in-directed one:

- In the graphical model we should remove all arrows which represent the direction of the relation between two vertices.
- In the graphical model We should connect all parents who have a connection with a single specific vertex (marrying the parents).
- Then, we should recognize the cliques which are existing in the in-directed produced graph.
- After that, we should change the probabilistic distribution from multiplied conditional probabilistic of each vertex to Joint Distribution of potential functions which can be created according to cliques we recognized before (each potential function belongs to one clique).
- In the end, the result should be multiplication of potential functions divided by Z which is normalization

Assume the following graph:

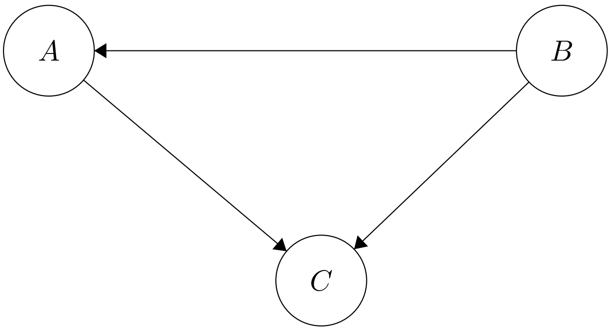


due to marrying the parents we would get this undirected graph:



ck]

which is the same one would get from this directed graph:



So loss of information is not completely avoidable.

c) In the subject of the joint distribution of un-directed graphs, there can be a huge amount of computation necessary if there is a long chain of related dependent variables to compute the probability of a single variable. The formula creates many nested sum calculations for every single variable. To prevent this disaster, we need to introduce the μ variable. μ variable can be interpreted as the impact of all ancestors and descendants on the distinct variable. There is need to compute μ one time from the first variable to the end in each step and then for every probability of a single variable can be reached easily by multiplying μ from both sides (ancestors and descendants, if we look at is as if there was some direction in the model).
In the case of normalization, we need to divide the result by Z. Z can be calculated as

$$Z = \sum_{x_n} \mu_\alpha(x_n)\mu_\beta(x_n)$$

at node x_n, where μ_α is the message passed from one side and μ_β is the message passed from the other side of the chain. Z is the normalization constant, which is necessary, to ensure that $\sum_{x_n} p(x_n) = 1$, which is necessary for $p(x_n)$ to be a valid probability distribution.

d) Both concepts are showing the connection between variables in a graph while they are useed in different types of graph representations (Cliques in un-directed graphs and factors in factor graph). Factors show us how variables connect and in which form. So if we have a Clique in an undirected graph, this means, that these variables can only be represented correctly by one joint distribution (i.e. they do not factorize). In factor graphs, we use the factor nodes to define what functions are used to connect the variable nodes (i.e. what the joint distribution is). Therefore one Clique, sharing a joint distribution needs to be connected to one factor/function.

So one can say, that a Clique in an undirected graphs directly corresponds to the one factor in the corresponding factor graph which connects all the members of that Clique.

e) The sum-product algorithm solves the problem of computing the marginals, but we might also be interested to find a way to maximize the distribution $p(x)$ (e.g. for maximum likelihood estimation). The procedure for this follows directly from the Sum-Product algorithm, but, in each step, we are looking for the value of the variable which makes the potential function or factor maximum.

$$p(X^{max}) = \max_X p(X) = \frac{1}{Z} \max_{x_1} \left[\max_{x_2} \left[\psi_{1,2}(x_1, x_2) \left[... \max_{x_n} \psi_{N-1,N}(x_{N-1}, x_{N-2}) \right] ... \right] \right]$$

Which is very similar to the sum-product algorithm. The basis of both algorithms is the distributive laws for products and sums/max-operations:

$$a \cdot b + a \cdot c = a \cdot (b + c) \text{ and } \max\{a \cdot b, a \cdot c\} = a \cdot \max\{b, c\}$$

The only difference is, that by summing one calculates the marginal and by maximizing the maximum likelihood value. Note also that this primarily gives rise to the **max-product**-algorithm, and **not** the **max-sum**-algorithm. This step happens, since (due to numerical stability) one prefers to use *log*s of probabilities and at this point products turn into sums

$$\log(a \cdot b) = \log(a) + \log(b)$$

f) Gaussian Graphical Models (GGMs) are a special type of undirected graphical model which assumes, that all variables are jointly gaussian disributed

$$x = (x_1, ..., x_n) \sim \mathcal{N}(\mu, \Sigma)$$

which makes computation easier.

GCMs can be used as tools to infer dependencies between biological variables. Popular applications are the reconstruction of gene, protein, and metabolite association networks. GGMs are an exploratory research tool that can be useful to discover interesting relations between genes (functional clusters) or to identify therapeutically interesting genes, but do not necessarily infer a network in the mechanistic sense.

In this model amount of dependency between two variables is represented by $\Sigma^{-1} = \Theta$. $\Theta_{ij} = 0$ means, that x_i and x_j are independent conditional on everything else. This is different from a 0 in the covariance matrix which means marginal independence. On the other hand, if Θ_{ij} gets large, it means strong dependency on each other.

g) The main improvement in the capability of Deep Learning is down to improvements in computational power (especially of GPUs). This enables training models that are actually *deep* in reasonable time. Much of what is currently being done in Deep Learning would have been possible thirty years ago, if they had had our current computers/graphics cards back then. By making Deep Leaning models much larger, their capability increases a lot (if there is enough training data and computational power available). This is how in 2012 AlexNet could beat any other competing model by a large margin in the ILSVRC.

There are several applications of Deep Learning. The most obvious one is probably image classification, since this is what AlexNet did. These days there are much more complex applications using images, like semanic segmentaion, image denoising, image upscaling and many more. A some of these techniques are for example used in self-driving cars. Furthermore applications include working with text, such as text summarization, machine translation and text generation, or even working with music.

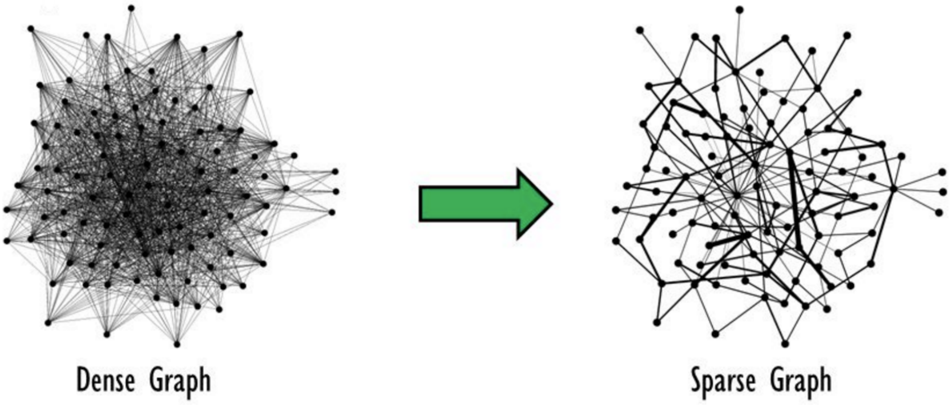
h) The primary goal of sparse coding is to find a (possibly overcomplete) set of basis vectors, such that the training data can be represented using only a few of these basis functions, either perfectly or 'well' by some (usually quadratic) measure.

Sparse coding aims to learn a useful sparse representation of any given data. Each datum will then be encoded as a sparse code:

- The algorithm only needs input data to learn the sparse representation. This is very useful since you can apply it directly to any kind of data, it is called unsupervised learning.
- It tries to find a sparse representation without losing much information, by leveraging the structure in the dataset.

Given a number of dimensions, sparse coding tries to learn an over-complete basis to represent data efficiently. To do so, you must have provided at first enough dimensions to learn this over-complete basis. An over-complete basis means redundancy in your basis, and vectors (while training) will be able to "compete" to represent data more efficiently. Also, you are assured, you won't need all dimensions to represent any data point: In an over-complete basis, you always can set some dimensions to 0.
Features will have multiple representations in the obtained basis, but by adding the sparsity constraint you can enforce a unique representation as each feature will get encoded with the most sparse representation.

'Sparse' just means, that a lot of factors (in this case) are 0, and only few bases are used. An example of 'sparse' can be seen below[1], by the example of a graph, where 'sparse' refers to only few edges being used.



i) An Autoencoder is made up of two parts: the first one is called the encoder, the second one the decoder. The encoder is used to transform the input (image) into some latent space. The decoder used that latent representation to recover the input image. Usually an encoder consists of some comvolutional layers either with some stride ≥ 1 or followed by pooling (and an activation function of course). The decoder can either consist of transpose-convolutions with stride ≥ 1 or upscaling and convolutions with stride 1 (and activation functions). Of course the architecture can also be different (especially if the input is not an image.

The goal here is not, to just get back to the input image (this would be pointless, since we already have it as an input), but rather to use the learned latent representation. One could for example use this as a way to compress the image, is the latent space is smaller, than the input. Another usage would be to feed in noisy images as the input, and train on the unaltered image as output, in order to train a denoising model.

The identity mapping can be prevented by making the latent space smaller, than

[1] https://miro.medium.com/max/5000/1*fDK6MfQvcliQ-u5lEmjc8Q.png

the input. Therefore the identity cannot be learned, as the latent representation is not large enough to hold the entire input.

j) Neural networks are made up of several layers, where the output of one is fed into the input of the next. The main operation of each layer is usually a matrix multiplication or convolution, which are both linear. Now we know, that the composition of two linear functions is still linear, so chaining these layers does not increase the capability. In order to get this to be an advantage, we need to break the linearity in between the layers. This is the purpose of an activation function (and therefore a activation function that is linear does not really make sense). So now instead of feeding the output from one layer into the input of the next, we feed it through the activation function and after that into the next layer. This breaks the linearity and enables the model to learn non-linear mappings.

Technically any function can be used as an activation function (even though some, like linear or constant ones, do not make much sense). Some popular activation functions include the logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$, $tanh$ and ReLU $r(x) = \max\{0, x\}$ (see Figure 1).

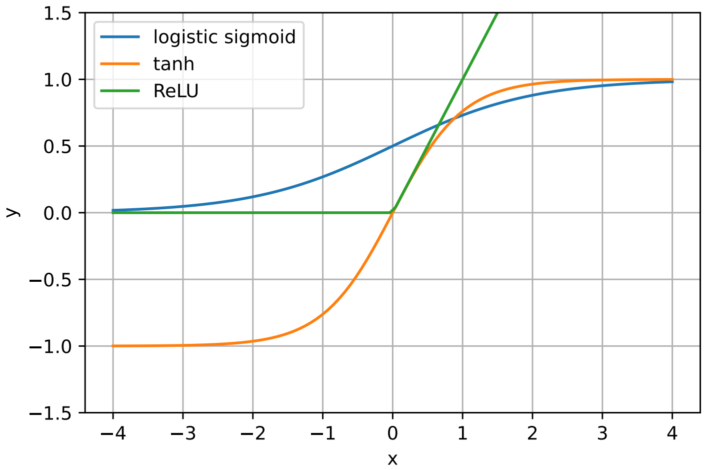


Figure 1: Plot of the activation functions discussed above.

Marginals

January 24, 2022

1 Problem 2

We will calculate the required distributions here in python, to avoid calculation errors:

[1]:
```
import numpy as np
```

First we recreate the given distribution

[2]:
```
p_dcs = np.array([
    0.06,
    0.13,
    0.04,
    0.23,
    0.06,
    0.17,
    0.04,
    0.27
]).reshape(2,2,2)
```

Next we can do a sanity check, to see if we have entered the values incorrectly:

[3]:
```
print(f"Sum of probabilities (should be 1): {p_dcs.sum()}")
```

```
Sum of probabilities (should be 1): 1.0
```

We can also define constants for the letters corresonding to the dimensions of the distribution, to make marginalization more readable:

[4]:
```
d = 0
c = 1
s = 2
```

1.1 Task a) $p(c)$

This can be easily done, by summing over everything but c:

[5]:
```
p_c = p_dcs.sum(axis=(d,s))
print(p_c)
```

```
[0.42 0.58]
```

So we have $p(c = 0) = 0.42$ and $p(c = 1) = 0.58$

1.2 Task b) $p(d)$

This can be done exactly the same way as before, but now summing over everything but d:

[6]:
```
p_d = p_dcs.sum(axis=(c,s))
print(p_d)
```

```
[0.46 0.54]
```

So we have $p(d = 0) = 0.46$ and $p(d = 1) = 0.54$

1.3 Task c) $p(d, c)$

This can also be done as before, except, now we have to only sum over s, since we want to keep c and d:

[7]:
```
p_dc = p_dcs.sum(axis=s)
print(p_dc)
```

```
[[0.19 0.27]
 [0.23 0.31]]
```

So we have: $p(c = d, c = 0) = 0.19$, $p(d = 0, c = 1) = 0.27$, $p(d = 1, c = 0) = 0.23$, and $p(d = 1, c = 1) = 0.31$.

1.4 Task d) $p(s|d, c)$

This is going to look similar to the full joint $p(d, c, s)$, except, that it is normalized differently, since it is only a probability distribution in s and **not** in c and d. So we can calculate this by dividing this by corresponding marginal from $p(d, c)$:

We could use the results from above, but for numpy to know what to do, it makes sense to keep the dimensons (so keep $p(d, c)$ three dimensional, since $p(d, c, s)$ is also three dimensional). This is why we recompute it:

[8]:
```
p_dc_keepdims = p_dcs.sum(axis=s, keepdims=True)
print(p_dc_keepdims)
```

```
[[[0.19]
  [0.27]]

 [[0.23]
  [0.31]]]
```

Note that this is the same distribution as above, just in a different format for numpy broadcasting to work nicely.

Now we can devide by this:

[9]:
```
p_s_dc = p_dcs / p_dc_keepdims
print(p_s_dc)
```

```
[[[0.31578947 0.68421053]
  [0.14814815 0.85185185]]

 [[0.26086957 0.73913043]
  [0.12903226 0.87096774]]]
```

This gives us the following results:

[10]:
```
d_max, c_max, s_max =  p_s_dc.shape

for d_value in range(d_max):
    for c_value in range(c_max):
        for s_value in range(s_max):
            print(f"p(d = {d_value}, c = {c_value}, s = {s_value}) =␣
 ↪{p_s_dc[d_value, c_value, s_value]}")
```

```
p(d = 0, c = 0, s = 0) = 0.3157894736842105
p(d = 0, c = 0, s = 1) = 0.6842105263157895
p(d = 0, c = 1, s = 0) = 0.14814814814814814
p(d = 0, c = 1, s = 1) = 0.8518518518518519
p(d = 1, c = 0, s = 0) = 0.2608695652173913
p(d = 1, c = 0, s = 1) = 0.7391304347826088
p(d = 1, c = 1, s = 0) = 0.12903225806451613
p(d = 1, c = 1, s = 1) = 0.870967741935484
```

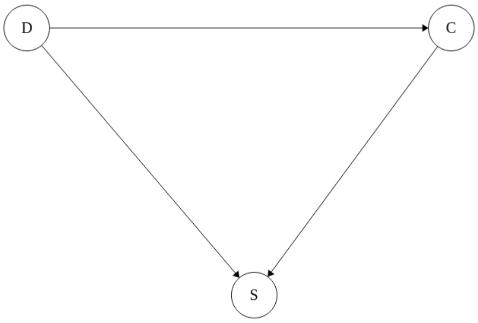
Note, that this is a probability distribution in s only, which can be seen since summing over s gives 1 in every case:

[11]:
```
print(p_s_dc.sum(axis=s))
```

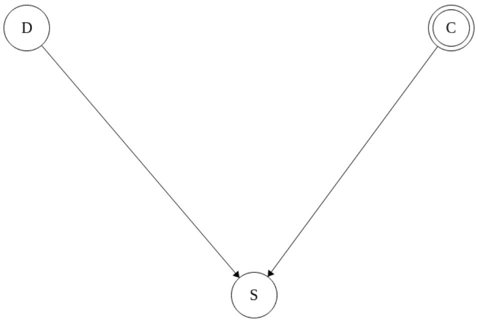
```
[[1. 1.]
 [1. 1.]]
```

1.5 Task d) using *do* calculus

Here now the graphical model becomes important: The original graphical model is:



By using the do-operator on whether the chief surgeon does the surgery, we have do delete the arrow from d to c. This results in the following new graph (where a double circle refers to the usage of the do-operator):



This means, that the original factorization

$$p(d,c,s) = p(d) \cdot p(c|d) \cdot p(s|d,c)$$

now changes to

$$p(d,c,s) = p(d) \cdot p(c) \cdot p(s|d,c)$$

since c is now independent of d. This way we can easily calculate the new joint, since we already have all the required distributions. As above, however, we will have to recalculate them with dimensions kept, for numpy to work:

```
[12]: p_c_keepdims = p_dcs.sum(axis=(d,s), keepdims=True)
      p_d_keepdims = p_dcs.sum(axis=(c,s), keepdims=True)
      # p_s_dc can be left as is, since it is already three dimensional
```

Now we can calculate the new joint after using the do-operator by simply multiplying as given in the formula above (this is automatically done correcly by numpy, since we have the correct dimensions):

[13]:

```
p_dcs_new = p_d_keepdims * p_c_keepdims * p_s_dc
print(p_dcs_new)
```

```
[[[0.06101053 0.13218947]
  [0.03952593 0.22727407]]

 [[0.05916522 0.16763478]
  [0.0404129  0.2727871 ]]]
```

Since this is mathematically not necessarily a probability distribution yet, we have to check if it sums to 1 and force it to if necessary:

[14]:

```
print(p_dcs_new.sum())
```

```
1.0
```

This seems to be fine, so we can leave it as it is.

[15]:

```
d_max, c_max, s_max =  p_dcs_new.shape

for d_value in range(d_max):
    for c_value in range(c_max):
        for s_value in range(s_max):
            print(f"p(d = {d_value}, c = {c_value}, s = {s_value}) =␣
 ↪{p_dcs_new[d_value, c_value, s_value]}")
```

```
p(d = 0, c = 0, s = 0) = 0.06101052631578948
p(d = 0, c = 0, s = 1) = 0.13218947368421055
p(d = 0, c = 1, s = 0) = 0.03952592592592593
p(d = 0, c = 1, s = 1) = 0.2272740740740741
p(d = 1, c = 0, s = 0) = 0.059165217391304356
p(d = 1, c = 0, s = 1) = 0.1676347826086957
p(d = 1, c = 1, s = 0) = 0.04041290322580646
p(d = 1, c = 1, s = 1) = 0.2727870967741936
```

For better readability, we can also convert the results to percentages and round to three digits:

[16]:

```
d_max, c_max, s_max =  p_dcs_new.shape

for d_value in range(d_max):
    for c_value in range(c_max):
        for s_value in range(s_max):
            print(f"p_new(d = {d_value}, c = {c_value}, s = {s_value}) =␣
 ↪{p_dcs_new[d_value, c_value, s_value]:2.1%}")
```

```
p_new(d = 0, c = 0, s = 0) = 6.1%
p_new(d = 0, c = 0, s = 1) = 13.2%
p_new(d = 0, c = 1, s = 0) = 4.0%
p_new(d = 0, c = 1, s = 1) = 22.7%
p_new(d = 1, c = 0, s = 0) = 5.9%
p_new(d = 1, c = 0, s = 1) = 16.8%
p_new(d = 1, c = 1, s = 0) = 4.0%
p_new(d = 1, c = 1, s = 1) = 27.3%
```

We can also look at the difference to before:

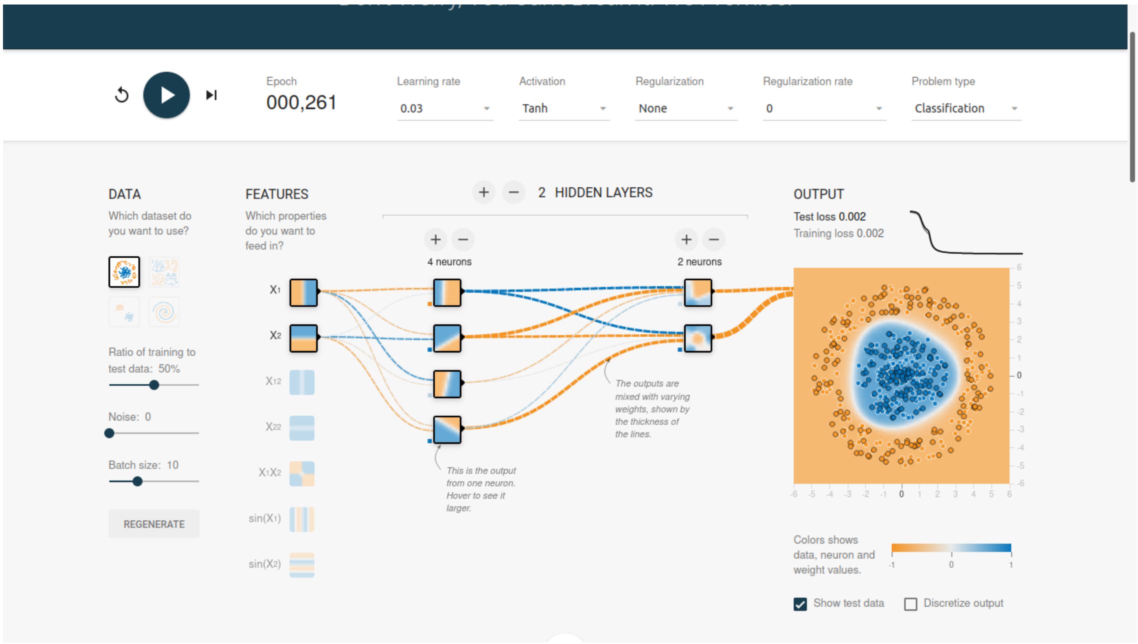
[17]:
```
d_max, c_max, s_max =  p_dcs_new.shape

for d_value in range(d_max):
    for c_value in range(c_max):
        for s_value in range(s_max):
            print(f"p(d = {d_value}, c = {c_value}, s = {s_value}) - p_new(d =␣
 ↪{d_value}, c = {c_value}, s = {s_value}) = {(p_dcs[d_value, c_value, s_value]␣
 ↪- p_dcs_new[d_value, c_value, s_value]):2.1%}")
```

```
p(d = 0, c = 0, s = 0) - p_new(d = 0, c = 0, s = 0) = -0.1%
p(d = 0, c = 0, s = 1) - p_new(d = 0, c = 0, s = 1) = -0.2%
p(d = 0, c = 1, s = 0) - p_new(d = 0, c = 1, s = 0) = 0.0%
p(d = 0, c = 1, s = 1) - p_new(d = 0, c = 1, s = 1) = 0.3%
p(d = 1, c = 0, s = 0) - p_new(d = 1, c = 0, s = 0) = 0.1%
p(d = 1, c = 0, s = 1) - p_new(d = 1, c = 0, s = 1) = 0.2%
p(d = 1, c = 1, s = 0) - p_new(d = 1, c = 1, s = 0) = -0.0%
p(d = 1, c = 1, s = 1) - p_new(d = 1, c = 1, s = 1) = -0.3%
```

Here we can see, that overall the change has been pretty small. The difference is, that the probability of the chief operating during daytime has been decreased, as has the probability of someone else operating at night. Inversely, the probability of the chief operaing at night has been slightly increased as has someone else operating during daytime.

Problem 3



Epoch 000,261

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons

2 neurons

This is the output from one neuron. Hover to see it larger.

The outputs are mixed with varying weights, shown by the thickness of the lines.

OUTPUT

Test loss 0.002

Training loss 0.002

Colors shows data, neuron and weight values.

☑ Show test data

□ Discretize output

Figure 2: Result of training the network on the circle dataset

a) The result of this task can be seen in Figure 2. Only the features x_1 and x_2 have been used.

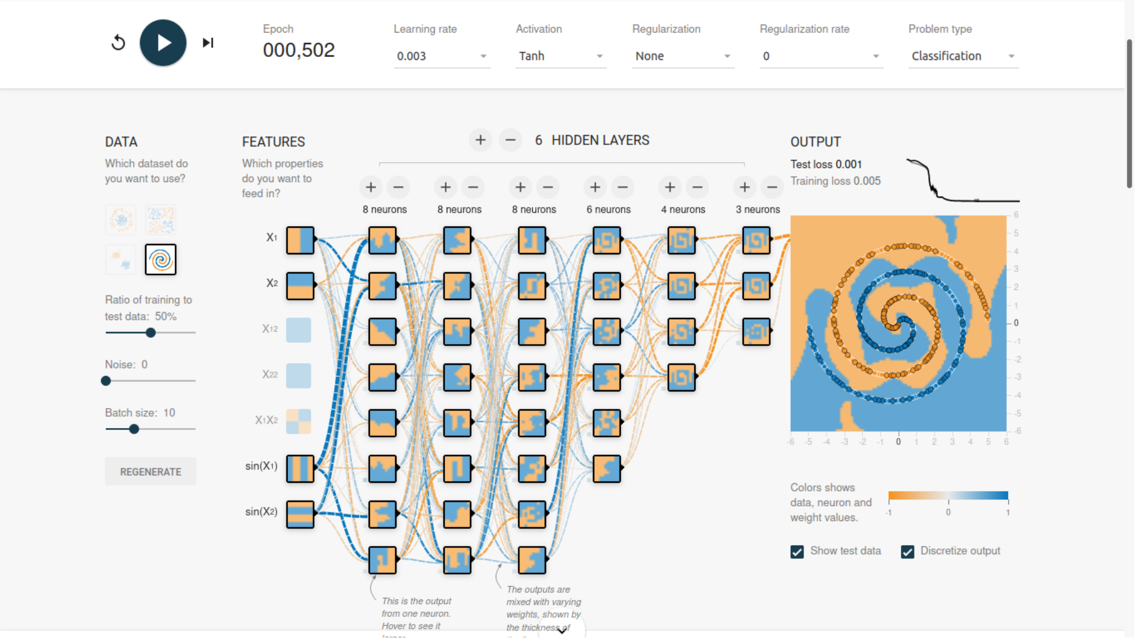
b) This is a rather easy problem. One can see, that the first hidden layer learns some straight decision boundaries (they have to be straight, since a single neuron is linear) at different angles. The second hidden layer combines them into something that already looks roughly like two partial circles. Finally these partial circles are combined to the final result.

c) The result of this task can be seen in Figure 3. Note that the learning rate has been reduced during training (starting at 0.1). This enables faster training in the beginning. The same result could have been achieved by training with the final learning rate (0.003) the entire time, but the it would have taken much longer.

The features x_1, x_2, $sin(x_1)$, $sin(x_2)$ have been used.

d) This is a much harder problem and requires a lot more layers. The first couple hidden layers seem to learn pretty arbitrary decision boundaries (note that these do not have to be linear since they can use the non-linear features $sin(x_1)$ and $sin(x_2)$). Starting from the third and definitely in the fourth hidden layer on can see something that resembles the spiral shape. How that is exacly made up is hard to describe, since the spiral is such a complex shape. On layers five and six on can see shapes that now resemble the spiral quite well, indicating, that maybe one or two layers less would have been sufficient.

Another thing to note is that the spiral shapes in layers five and six are quite square, different from the original square which is round. This is likely because the model relies heavily on the sin inputs, which are axis-aligned and thus square in combination. This claim is backed, by the fact, that in the connections from the input to



Epoch 000,502

Learning rate 0.003

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

Test loss 0.001

Training loss 0.005

☑ Show test data ☑ Discretize output

DATA
Which dataset do you want to use?
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X1)
sin(X2)
+ − 6 HIDDEN LAYERS
8 neurons
8 neurons
8 neurons
6 neurons
4 neurons
3 neurons
OUTPUT
Colors shows data, neuron and weight values.
-1
0
1
This is the output from one neuron. Hover to see it
The outputs are mixed with varying weights, shown by the thickness of

Figure 3: Result of training the model on the spiral dataset.

the first hidden layer on can clearly see (from the thickness of the lines), that the *sin*-features are used in a strong way.