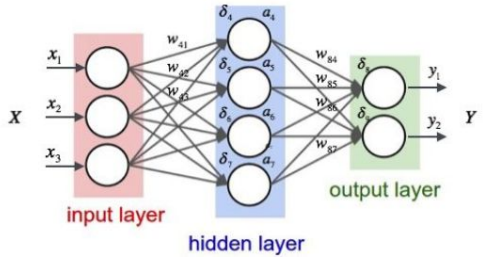
Exercise 5

Sindre Eik de Lange

1. Understanding Neural Networks



1.1

a. **X** = input, e.g. a photo, a text corpus, etc.
b. **Input layer** = Initial layer in the neural network
c. **Weights w** = The set of individual weighted inputs for each node in the neural network
d. **Sum of weights function δ** = An “adder”, which sums the input signals.
e. **Activation function α** = function that decides whether a neural fires for current inputs.
f. **Hidden Layer** = Set of layers between the input- and output layer.
g. **Output layer** = The last layer where the model calculates the appropriate values, e.g. the probabilities of the different classes in the data set.
h. **Y** = The actual output. Following the aforementioned example: the predicted class(es).

1.2

I am not sure I understand the question, because the weights are what actually “learns” because this is what research has deemed the most efficient for such networks. That is, to keep the activation functions static, while updating the weights (making them dynamic) using techniques such as (Stochastic) Gradient Descent (w. Restarts). However, if the question is “why is it that having dynamic weights works for neural networks?”, then this is because by using the aforementioned techniques one can calculate how the weights should be updated in order to minimize the function.

1.3

Randomization is used in order to avoid local minima of a function. An example where the network would not come to a good solution without it, is when decided value finds you in a local minima, which is (notably) worse than the global minima. If one is not using techniques

such as Momentum (which only helps to a certain degree, in this case), then one would be stuck in the same local minima for the entirety of the training.

1.4

The input of the “output”-node = (0,69*0,116)+(0,77*0,329)+(0,68*0,708) = 0,815.
The output of the “output”-node (i.e. after activation function = sigmoid) = 0,69.

1.5

w3 = 0,134
w6 = 0,234
w9 = 0,354

2. Simulation of Neural Networks

2.1

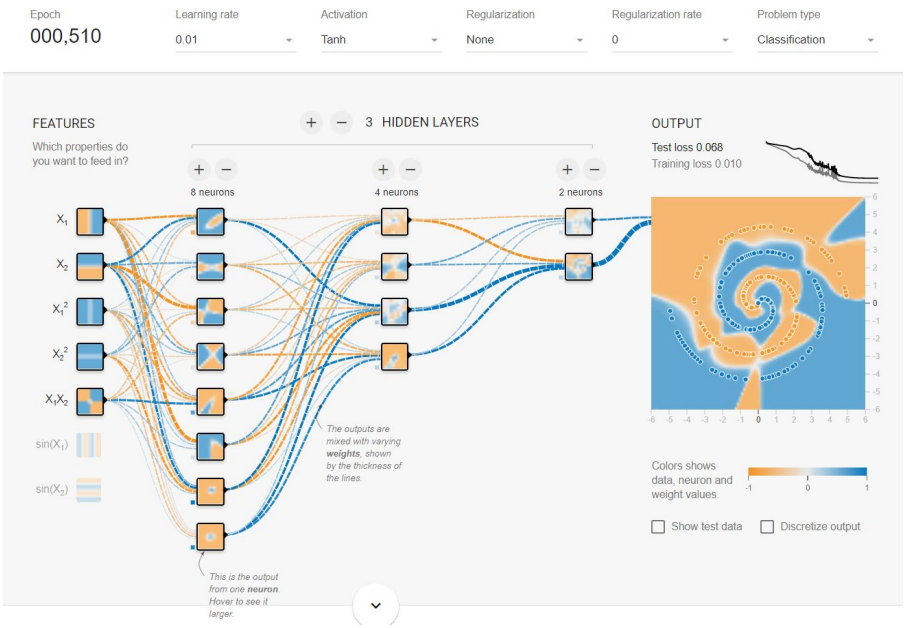
The neurons in the first hidden layer have simpler shapes than that of the neurons in the second hidden layer because that is how neural networks are built to learn: they learn the simpler shapes/relationships first, such as horizontal and vertical lines, then they learn more intricate shapes/relationships the deeper into the network one goes.

2.2

When changing the learning rate to 3 the model jumps all over the place - it is unable to find any sort of minima, and stay there, because when updating each weight it updates it far too much.

2.3

That is true, however, with more complex shapes one generally needs more complex/deeper models, so if one has 4 hidden layers, then the model obtains fairly good results:



2.4

This is because, as mentioned in *2.3*, when the shapes are more complex, so does the models need to be. This is because a simple model have no way of learning complex shapes and relationships. I.e. the more neurons (and hidden layers) the more updates to the weights, therefore less need for radical updates, hence converging faster to the minima.

2.5

Deep Learning

> Create the network

In [197]:
```
model = Sequential()
model.add(Dense(10, input_dim=4, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(6, activation='relu'))
model.add(Dense(3, activation='sigmoid'))
```

Justification for the number of hidden layers and neuors in each layer

As far as I understand there is not a set of rules specifying number of *hidden layers*, and *neurons* in those *hidden layers*. However, when using different dataset, with different sizes and complexity, one gains some heurestics about these choices. The *Iris dataset* is fairly non-complex, which is why one does not need to have too many hidden layers, nor many neurons on those layers. This is the reason I choose 2 hidden layers.

Furthermore, the number of *neurons* in each layer, as far as I understand, should be in a power of 2, when using GPUs. Apperantly they perform better if this is the case (which is the same for batch size, etc.). I can be wrong about this, so please correct me if that is the case. Nonetheless, I do not have GPU on my computer, so it does not really matter.

The number of *neurons* in the *hidden layer* is basically a product of trial and error, which is how one usaully has to do in order to obtain the best results. But, as stated earlier, the Iris dataset if fairly non-complex which means that one does not need a large amount of neurons in each layer, in order to obtain a good model.

Lastly, the only layer that is static is the *output layer*, which has 3 neurons, because of my *OneHotEncoding* of the *target variable*.

> Compile model, for speed

In [198]:
```
optimizer = Adam(lr=0.002)
```

In [199]:
```
# Categorial cross entropy lets you classify > two classes.
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

> Train, and evaluate the model (Should be > 95%)

In [200]:
```
model.fit(X_train, y_train, epochs=100, batch_size=5)
scores = model.evaluate(X_test, y_test)
```

```
Epoch 92/100
105/105 [==============================] - 0s 752us/step - loss: 0.1177 - acc: 0.9524
Epoch 93/100
105/105 [==============================] - 0s 728us/step - loss: 0.1002 - acc: 0.9524
Epoch 94/100
105/105 [==============================] - 0s 744us/step - loss: 0.0779 - acc: 0.9714
Epoch 95/100
105/105 [==============================] - 0s 743us/step - loss: 0.0781 - acc: 0.9810
Epoch 96/100
105/105 [==============================] - 0s 734us/step - loss: 0.0890 - acc: 0.9524
Epoch 97/100
105/105 [==============================] - 0s 784us/step - loss: 0.0809 - acc: 0.9714
Epoch 98/100
105/105 [==============================] - 0s 785us/step - loss: 0.0777 - acc: 0.9714
Epoch 99/100
105/105 [==============================] - 0s 739us/step - loss: 0.0867 - acc: 0.9714
Epoch 100/100
105/105 [==============================] - 0s 747us/step - loss: 0.0791 - acc: 0.9619
45/45 [==============================] - 1s 14ms/step
```

In [201]:
```
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
acc: 100.00%
```