# Network Analysis

## Individual Assignment 1

## Hans Alberto Franke

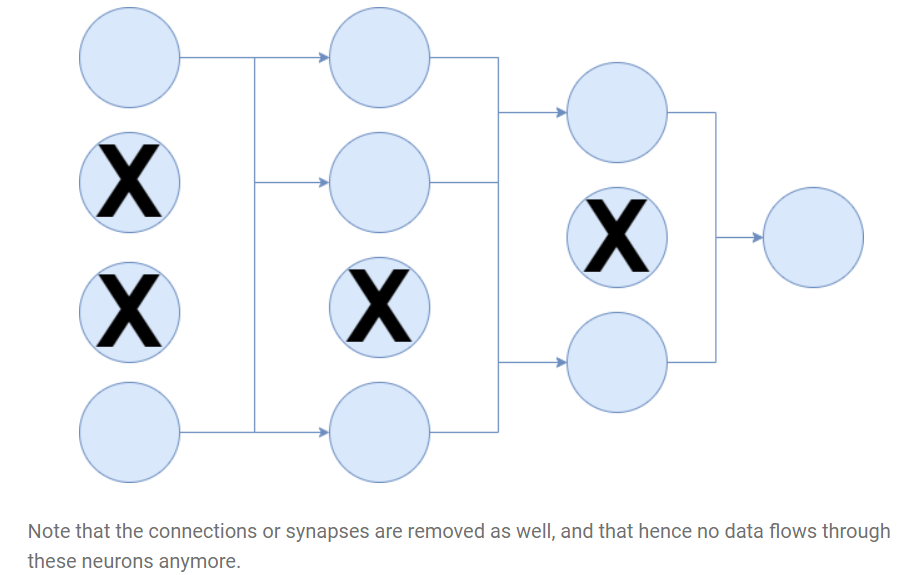
## Feb/21

## Describe the principles of overfitting and how dropout can reduce this (Question 1, 5 points)

### **Answer**:

**Overfitting,** being too sensitive to your training data. In other words, your model predict “too good” in training set, but not so well in real data or even in test set. No only overfitting are to be avoided, but underfitting as well, as your model will perform worse than it could perform theoretically. Fortunately, certain techniques – called regularizes – can be used to reduce the impact of overfitting. **Dropout** is one of them.

Using **Dropout**, the training process essentially drops out neurons in a neural network. They are temporarily removed from the network, which can be visualized as follows (image bellow). This removal neurons and synapses during training is performed at random, with a parameter that is tunable (p.e % of neurons dropped).



**Source**: <https://www.machinecurve.com/index.php/2019/12/16/what-is-dropout-reduce-overfitting-in-your-neural-networks/>

As (Srivastava et al., 2014) defines, evaluating the gradient is done with respect to the error, but also with respect to what all other units are doing. This means that neurons may fix the mistakes of other neurons changing their weights. This behavior may lead to co-adaptations that may not generalize to new data, resulting in overfitting. **Dropout**, then, prevents these co-adaptations by making the presence of other hidden [neurons] unreliable. That way neurons cannot depend on other units to fix their mistakes, which minimize the number of co-adaptations that do not generalize to unseen data and therefore reduces overfitting as well.

**Dropout** prevents **overfitting** due to a layer's "over-reliance" on a few of its inputs, once these inputs aren't always present during training (i.e. they are dropped at random), the layer learns to use all of its inputs, improving generalization.

## Write a short (~500 word) summary of the experimental approach and results. (Question 2, 10 points)

**Answer:**

The article compares a neural network with biological functions of humans in task of easily recognize objects in scenes. This ability is known to be supported by a network of hierarchically interconnected brain areas. The authors looks to use computational techniques to identify a neural network that matches human performance. First, they construct a scenario considering of natural categories, the objects are putted on random natural scenes to ensure the background is uncorrelated with object. Using multiple electrode arrays, they collected responses from 168 IT neurons to each image, then using high-throughput computational methods to evaluate thousands of candidate neural network model on these images, measuring object categorization performance x IT neural predictive for each model.

The modelling of CNNs try to approximate the general retinotopic organization of the ventral stream using spatial convolution, with calculations in any one region of the visual field identical to those elsewhere. Each of this convolutional layer is composed of simple and neuronally plausible basic operations, including *linear filtering, thresholding, pooling, and normalization*. So these layers can be stacked hierarchically to construct deep neural networks.

The numbers of layers in Network depth from one to three, and filter weights for each layer were chosen randomly from bounded uniform distributions whose bounds were model parameters. Models were selected for evaluation by one of three procedures: (i) random sampling of the uniform distribution over parameter space. (ii) optimization for performance on the high-variation eight-way categorization task and (iii) optimization directly for IT neural predictivity.

They evaluate the performance of the network in different difficult scenarios, from simple image position to high(p.e 180° rotations on all axes, 2.5× dilation, and full-frame translations..). The comparison was made with human performance and other networks, to proper evaluate performance and find clues where the model can be improved.

After comparison, they propose extend network performance with a mixture of Deep Neural Networks which correspond intuitively to architecturally specialized subregions like those observed in the ventral visual stream. They used hierarchical modular optimization (HMO) procedure embodies a conceptually simple hypothesis for how high-performing combinations of functionally specialized hierarchical architectures can be efficiently discovered and hierarchically combined, without needing to prespecify the subtasks ahead of time. Algorithmically, HMO is analogous to an adaptive boosting procedure interleaved with hyperparameter optimization.

To gain further insight, an exploratory analysis of the parameters of the learned HMO model, assessing each parameter both for how diverse it was between model mixture components and how sensitively it was tuned. Two classes of model parameters were especially sensitive and diverse: (i) filter statistics, including filter mean and spread, and (ii) the exponent trading off between max-pooling and average-pooling.

The results can be added in neuroscience, because is a common understanding in it, the tuning curves of neurons in lower cortical areas will be a necessary precursor to explaining higher visual cortex, the results indicate that it is useful to complement this bottom-up approach with a top-down perspective characterizing IT as the product of an evolutionary/developmental process that selected for high performance on recognition on tasks like those used in our optimization.

## Play around with these settings and see how they affect your ability to learn classification of different data sets. Write down what you found and how you interpret the effects of these settings. This question is intentionally open to allow you to explore the process. (Question 3, 8 points)

The website is a visual representation of a formal neural network. First, as question states we can select only **classification** problems. Besides classification you can select different parameters, the same you can chose in a model on Keras, for example, thus this visual is more clear to see the impact of changing each one in real time, therefore one can have a better understanding of the meaning of each of this parameters and how they can impact in performance (speed/epochs) and accuracy (loss function).

Bellow, I define of each of this parameters:

* **Parameters**:
  + Learning rate
    - The amount that the weights are updated during **training** is referred to as the step size
    - It is important parameters because if it too large it can be jumping around local optimal or NOT converge at all.
  + Activation function
    - is a **function** that is added into an artificial **neural network** in order to help the network **learn** complex patterns in the data. When comparing with a neuron-based model that is in our brains, the **activation function** is at the end deciding what is to be fired to the next neuron
    - Funcions: Rectified Linear Units (ReLU), Linear Activation (Linear), a hyperbolic activation function (Tanh) and Sigmoid.
  + Regularization:
    - As we can see from the formula of **L1 and L2 regularization**, **L1 regularization** adds the penalty term in cost function by adding the absolute value of weight(Wj) parameters, while **L2 regularization** adds the squared value of weights(Wj) in the cost function
    - **Regularization**, significantly reduces the variance of the model, without substantial increase in its bias. ... As the value of λ rises, it reduces the value of coefficients and thus reducing the variance. It is user to prevent **overfitting.**
  + Regularization Rate
    - Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda** (also called the **regularization rate**).
  + Ratio of training and test
    - Split between train and test , or size of each one in the model. P.e 80% training and 20% from test
  + Input Features
    - Or feature selection / transformation => (normal or quadratic, interaction x1 \* x2, sin(x1), etc.. )
  + Batch size
    - How many rows per batch to be used in training iteration (update weights).
* **Modeling**:
  + Number of hidden layers
  + Number of neurons in hidden and output layers
* **Extra**:
  + Noise => to simulate real input problems (f.e. wrong inputs, outliers, ..), data patterns become more **irregular** as the noise increase
* **Output**:
  + Loss: in train and test (accuracy performance, this loss value is used in backpropagation to update the weights of layers/filters)
  + Visual plot with predict values (and if you click the real values (“show test data”)

The colors of the plot represent positive (blue) or negative (orange) impact. In the hidden layers, the lines are colored by the weights of the connections between neurons. Blue shows a positive weight, which means the network is using that output of the neuron as given. An orange line shows that the network is assigning a negative weight. One can look at lines to understand how backpropagation are updating the weights of the network based on the loss function. If one put the mouse over a node, it shows the impact of that node on final output. The size of the line represents the weights in a visual way.

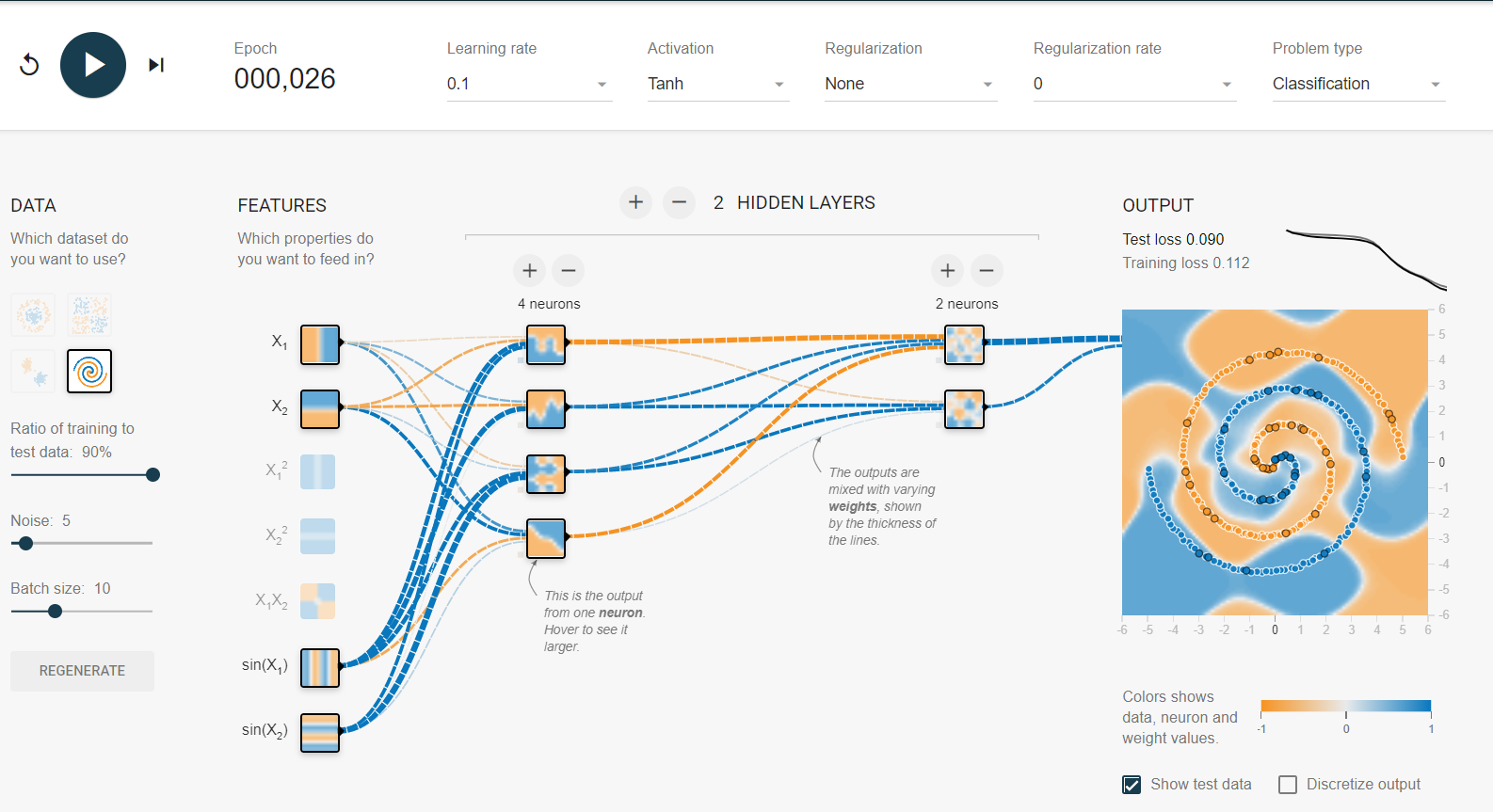
The interface of the website allow one to change parameters and simulate a network. Therefore one can compare how fast a network converge (number of epochs) to a desired loss output.

## What is the minimum you need in the network to classify the spiral shape with a test set loss of below 0.1? (Question 4, 7 points)

I’m considering **minimum cost** achieve if, the loss threshold (<0.1) is reached with the minimum number of epochs and simpler modelling (f.e minimum number of nodes).

**First step, was to simulate create new features**. The best features I can come up with is to transform the coordinates to [spherical coordinates](https://en.wikipedia.org/wiki/Spherical_coordinate_system) using the sin(x1) and sin(x2). So, with ~30 epochs it achieve test loss < 0.09. **Activation** **function**: I chose the Tanh.

* Number of neurons:
  + First layer: 4 (less than that the converge rate almost double)
  + 2nd layer: 2 (if not the converge rate takes almost more 100 epochs, 4 neurons don’t affect performance too much so avoided to turn network simpler)
* I don not use a **regularization rate** because the model don’t seems overfitted (difference in plot from training to test loss)
* For smaller batch sizes like 1, 0.1 is too high **learning rate** as the model fails to converge as it jumps around the global minima. So, if you would like to keep a high **learning rate**(0.1), keep the batch size high(10) as well. This usually gives a slow yet smoother convergence.
* **Ratio of training** to test data: 90% it converge fast and almost same overfitting than with 70% (difference from training loss to test loss)
* **Noise** => 5 (higher than that the model takes long to converge and have a lot of overfitting almost double loss in test)



Source:

<http://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.1&regularizationRate=0&noise=5&networkShape=4,2&seed=0.19939&showTestData=true&discretize=false&percTrainData=90&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=true&cosY=false&sinY=true&collectStats=false&problem=classification&initZero=false&hideText=false>

## Explain the principle of backpropagation of error in plain English in about 500 words. This can be answered with minimal mathematical content and should be IN YOUR OWN WORDS. What is backpropagation trying to achieve, and how does it do so? (Question 5, 8 points)

One would be interested for example, what will be my grade in Network Analysis as a function of grade in individual assignment and group assignment? The algorithm needs training data, p.e. grades of the other students. Then one can try to predict my grade (y^) with my own inputs, like: my grade on individual assignment (x1) and my grade on group assignment(x2).

In any ANN(artificial neural network) we have the following items: Neurons and Weights. Neurons store the values that will be calculated to define the Weights, where these weights are the "key" to the functioning of all ANN, it is by the weight that ANN can identify, for example, that object is round and not square or what would be my predicted grade. I will illustrate the effect of backpropagation in a simple multiple-linear regression.

Every time you submit a input( Student 1 (x1 = 5, x2 = 6) ) and a certain output for this (student 1 y = 6), it regulates the weights (which are the lines that connect the neurons) to try to get as close as possible to the real result y. This relation can be mapped as (f(y) = W1.x1 + W2.x2 + B), where B = intercept or bias. The way that ANN learns it **weights**, or solve this equation, can be expressed by gradient descend for example, or in another words, it minimizes the **loss function** of y for y^. What **backpropagation** does, is propagate this error backwards on network, therefore the layers weights are adjusted based on their contribution on the overall error, or update network parameters in small amount in the direction opposite to the error gradient.

**As a formal definition**, when the neural network is initialized, weights are set for its individual elements, called neurons. Inputs are loaded, they are passed through the network of neurons, and the network provides an output for each one, given the initial weights. The initial weights are random numbers, essentially, backpropagation evaluates the expression for the derivative of the cost function as a product of derivatives between each layer from left to right – "backwards" – with the gradient of the weights between each layer being a simple modification of the partial products (the "backwards propagated error"). *Back-propagation* is the essence of neural network training. It is the practice of fine-tuning the weights of a neural net based on the error rate (i.e. loss) obtained in the previous epoch (i.e. iteration). Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization.

**Example: See figure1**

* **Inputs**:
  + Individual assigment = x1
  + Group assigment = x2
* **Hidden** Layer of 2 **nodes** (N1 and N2) and 4 **weights** where **a** = outputs
  + W11 = link from x1 to N1
  + W21 = link from x2 to N1
  + W12 = link from x1 to N2
  + W22 = link from x2 to N2
  + **N1 =>** a1 **=**  w11\*x1 + w21\*x2
    - **F(a1) = a1** (using activation function)
  + **N2 =>** a2 **=** w12\*x1 + w22\*x2
    - **F(a2) = a2**
* **Output** layer (O1) = 1 node => predicted grade (**y^**)
  + W5 = link from N1 to O1
  + W6 = link from N2 to O1
  + **y^ =** a1\*w5 + a2\*w6
    - F(y^) = y^
* **Error / Cost Function** 
  + **Error/Loss: E =** y^- y
* **activation** **function**that determines the activation value at every node in the neural net. For simplicity, let’s choose an identity activation function: **f(a) = a**
* **Formula of partial derivative or *delta*:**
* delta\_0 = w \* delta\_1 \* f`(z) , where *delta = total\_loss = E*, and *delta\_1* is on the right, and *f`(z)* is the derivate of activation function, as identity *f`(z) = f(z)*

**Example data:**

* W5 = W5 – alpha \* y^ \* delta(y^)
* W1 = W1 – alpha \* a1\* delta(a1)
* Or, Wi = Wi – alpha \* ai \* delta(ai), where ai is always the result in the node in the **right** of the Wi.

W11

W12

W21

W22

W5

W6

Backpropagation

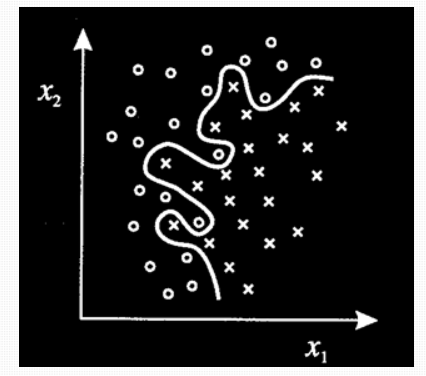
Feedforward Information

**Figure1**. **Backpropagation**: The sizes of yellow circles represents the contribution of each weight(W) based on the overall error

Then there is another round of calibration, and the training, evaluating and backpropagation is done until the model “converge”, for instance the loss function is minimized until some threshold or hit the defined number of epochs. Each round is define or by one row, or by a batch of rows (batch gradient).

These assumptions are made trying to simulate neuron of human brain, or in other words, adjusting weights so that neurons fire or not depends on the added value that they have to the network.

The main advantage of using Backpropagation is that it works with multilayers and solves "nonlinearly separable" problems and some algorithms do not solve. In summary, a "nonlinearly separable" problem is one where we cannot separate 2 distinct classes on the two-dimensional Cartesian axis just by tracing a line.



Another important feature is that Backpropagation is feedforward, that is, the connection between neurons is not cyclic, going from start to finish you will not find a cycle.

**BONUS QUESTION:**

## Describe the process of backpropagation in mathematical terms. Here, explain (in English, in about 500 words) what each equation you give does, and relate this to the answers given in Question 5. You are welcome to express equations in your own R or python code rather than using equation layout, but you need to make clear you understand what each line is doing. (Question 6, 5 points).

* **import** numpy as np
* # define the sigmoid function
* # Backpropagation is actually a major motivating factor in the historical use of sigmoid activation functions due to its convenient derivative:
* **def** sigmoid(x, derivative=False):
* **if** (derivative == True):
* **return** sigmoid(x,derivative=False) \* (1 - sigmoid(x,derivative=False))
* **else**:
* **return** 1 / (1 + np.exp(-x))
* # choose a random seed for reproducible results
* #np.random.seed(1)
* # learning rate
* alpha = .1
* # number of nodes in the hidden layer
* num\_hidden = 2
* # inputs (grade individual assigment, grade of group assigment)
* #Student 1 to student n (number of rows)
* number\_students = 15
* X = np.random.randint(10, size=(number\_students, 2))

* # outputs
* # y.T is the transpose of y, making this a column vector
* y = np.random.randint(10, size=(number\_students, 1)).T

* # initialize weights randomly with mean 0 and range [-1, 1]
* # the +1 in the 1st dimension of the weight matrices is for the bias weight
* hidden\_weights = 2\*np.random.random((X.shape[1] + 1, num\_hidden)) - 1
* output\_weights = 2\*np.random.random((num\_hidden + 1, y.shape[1])) - 1
* # number of iterations of gradient descent
* num\_iterations = 10000
* # for each iteration of gradient descent
* **for** i **in** range(num\_iterations):
* # forward phase
* # np.hstack((np.ones(...), X) adds a fixed input of 1 for the bias weight
* input\_layer\_outputs = np.hstack((np.ones((X.shape[0], 1)), X))
* hidden\_layer\_outputs = np.hstack((np.ones((X.shape[0], 1)), sigmoid(np.dot(input\_layer\_outputs, hidden\_weights))))
* output\_layer\_outputs = np.dot(hidden\_layer\_outputs, output\_weights)
* # backward phase
* # output layer error term
* output\_error = output\_layer\_outputs - y
* # hidden layer error term
* # [:, 1:] removes the bias term from the backpropagation
* hidden\_error = hidden\_layer\_outputs[:, 1:] \* (1 - hidden\_layer\_outputs[:, 1:]) \* np.dot(output\_error, output\_weights.T[:, 1:])
* # partial derivatives
* hidden\_pd = input\_layer\_outputs[:, :, np.newaxis] \* hidden\_error[: , np.newaxis, :]
* output\_pd = hidden\_layer\_outputs[:, :, np.newaxis] \* output\_error[:, np.newaxis, :]
* # average for total gradients
* total\_hidden\_gradient = np.average(hidden\_pd, axis=0)
* total\_output\_gradient = np.average(output\_pd, axis=0)
* # update weights
* hidden\_weights += - alpha \* total\_hidden\_gradient
* output\_weights += - alpha \* total\_output\_gradient
* # print the final outputs of the neural network on the inputs X
* #transform output to predict values
* y\_hat = np.mean(output\_layer\_outputs, axis=0)
* y\_hat = y\_hat.reshape(1,number\_students)

* #print("Output After Training: \n{}".format(output\_layer\_outputs))
* **print**("Matrix of grades-assigments:X1,X2 \n")
* **print**("Matrix of grades-EXAMs: Exam or Y \n")
* **print**("Number of students:", number\_students)
* **print**("\n X1 , X2, EXAM, Y\_HAT")
* **print**(np.hstack((X, y.T, y\_hat.T)))
* #Total error after 1000 iterations
* **print**('Total error',np.sum(output\_error))

