

# FYS-STK4155 H20 - Project 2:

## Neural nets

Olav Fønstelien

November 13, 2020

### Abstract

## 1 Introduction

In [1] we studied linear regression methods. We saw that for the Ordinary Least Squares (OLS) and Ridge methods we had to calculate a matrix product on the form  $(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top$  in order to obtain the coefficient estimates  $\hat{\beta}$  ( $\lambda = 0$  for OLS). As we saw, the matrix inversion operation may be numerically unstable, and if we need a large number of samples to approximate the function, this operation becomes computationally demanding or even infeasible.

The Stochastic Gradient Descent method (SGD) deals with both of these problems. It still requires calculating  $\mathbf{X}^\top \mathbf{X}$ , but since it determines the coefficient estimates  $\hat{\beta}$  iteratively on so-called *mini batches* which can be kept in the higher levels of cache, the operation becomes less computationally demanding. The inversion operation is avoided all-together.

SGD is useful in other methods as well. SGD can be used in logistic regression methods, which lets us solve *classification problems* where we predict the outcome of an event as the one most likely of several possible *classes* of outcomes. For these problems, linear regression methods are unsuitable. Logistic regression again is unsuitable for the class of problems where the decision boundary between features and outcomes is non-linear, like the *XOR* (exclusive or) logical operator. For these problems, Artificial Neural Networks (ANNs) can be used.

In this report we will ....

## 2 Methods

In this section we will develop methods for solving regression and classification problems with traditional linear and logistic regression methods, as well as Feed-Forward Neural Network, a simple type of ANN. In the outline I will use matrix notation where possible, since this closely resembles the implementation in a computer programming environment with array programming capabilities like native MATLAB, C++ with the `armadillo` lib or Python with the `numpy` module, which I will use.

### 2.1 Stochastic Gradient Descent

Stochastic Gradient Descent is a numerical method for minimizing the cost function of a mathematical optimization method like OLS or logistic regression, where the minimization is done in an iterative, stochastic approach on a selection of the full set of samples. That is; given a cost function  $\mathcal{C}(\beta)$ , where  $\beta \in \mathbb{R}^p$ , SGD lets us solve the problem

$$\frac{\partial \mathcal{C}(\beta)}{\partial \beta} = \mathbf{0}, \quad (1)$$

where a closed-form solution is otherwise not available, or computationally expensive, and by using only a sub-set of the samples in each iteration. It is based on the iterative application of Newton-Raphson's method on the coefficient vector  $\beta$ , such that

$$\beta^{(i+1)} = \beta^{(i)} - [\mathbf{H}^{(i)}]^{-1} \mathbf{g}^{(i)}, \quad (2)$$

where  $\mathbf{g}$  is the gradient of the cost function and  $\mathbf{H}$  is its Hessian matrix, each given by

$$\mathbf{g}^{(i)} = \frac{\partial \mathcal{C}(\beta^{(i)})}{\partial \beta}, \text{ and } \mathbf{H}^{(i)} = \frac{\partial^2 \mathcal{C}(\beta^{(i)})}{\partial \beta \partial \beta^\top}. \quad (3)$$

Equation (2) can be deduced from the Taylor expansion of  $\mathcal{C}(\beta)$  around  $\beta^{(i+1)} - \beta^{(i)}$ . See [2] for a thorough outline of this.

Calculating the Hessian and inverting it is an expensive and numerically volatile operation, so our approach will be to replace it by a scalar  $\eta$ , the so-called *learning rate*, such that  $\mathbf{H}^{-1} \rightarrow \eta$ . Equation (2) thus becomes

$$\beta^{(i+1)} = \beta^{(i)} - \eta \mathbf{g}^{(i)}. \quad (4)$$

Finding the *best*  $\eta = \hat{\eta}$ , such that  $\beta^{(i+1)} = \hat{\beta}$  again involves calculating the Hessian, unfortunately [2];

$$\hat{\eta} = \frac{\mathbf{g}^\top \mathbf{g}}{\mathbf{g}^\top \mathbf{H} \mathbf{g}}. \quad (5)$$

However, the cost function  $\mathcal{C}(\beta)$  is guaranteed to be convex for linear and logistic regression problems, implying a positive definite  $\mathbf{H}$  [3]. Therefore, as long as we select  $\eta < 2/\lambda_{max}$ , where  $\lambda_{max}$  is the largest eigenvalue of the Hessian matrix, Equation (4) will converge towards  $\hat{\beta}$ . Again,  $\lambda_{max}$  will remain unknown to us;

but knowing that if we select a  $\eta$  which is not too large, a solution will be found, we base our approach on trial and error.

The next problem then becomes to decide when to stop the search for  $\hat{\beta}$ . One solution is to stop when the difference between the iterations becomes smaller than some defined value,  $\|\beta^{(n+1)} - \beta^{(n)}\|_2 \leq \varepsilon$ . Another, which we will use here, is simply to define a number of iterations, and extract the solution thereafter. This might seem a little inaccurate, maybe, but works well since the direction of  $\mathbf{g}$  is always towards the global minimum, and  $|\mathbf{g}| \rightarrow 0$  when  $\beta \rightarrow \hat{\beta}$ , which means that if we come close to  $\hat{\beta}$ , the final approximation  $\beta^{(final)} \approx \hat{\beta}$ .

A further refinement to this is to incrementally decrease  $\eta$  for each iteration towards  $\hat{\beta}$ ; that is to let  $\eta \rightarrow \eta^{(i)} = f(i)$ , such that  $\eta^{(i+1)} < \eta^{(i)}$ . This increases the stability of the method since it allows us to even have  $\eta^{(0)} > 2/\lambda_{max}$  and still get convergence.

Two such *learning schedules*  $\eta^{(i)} = f(i)$  are the `invscaling` schedule in Python's `scikit-learn` module [4];

$$\eta^{(i)} = \frac{\eta^{(0)}}{i^k}, \quad (6)$$

and another one suggested by Geron in [5];

$$\eta^{(i)} = \frac{t_0}{i + t_1}. \quad (7)$$

Here the factors  $k, t_0, t_1$  are user defined. Figure 1 shows each learning schedule with some selections for the user-defined factors over the 100 first iterations.

Calculating the gradient of the cost function  $\mathbf{g}$  in Equation (4) may prove computationally demanding if the data set is large. The gradient of the logistic regression method cost function, for instance, would require that we calculate the matrix product  $\mathbf{X}^T \mathbf{P}$  in each iteration. In SGD, we select a subset of the samples by a stochastic process, and calculated an approximated gradient based on this subset or mini batch. The standard deviation of the gradient's expectation value relative to the number of samples  $n$  follows the relation [2]

$$\sigma_g \sim \frac{1}{\sqrt{n}}, \quad (8)$$

which means that if we collect a mini batch of 100 from a data set of 1000 samples, the standard deviation will only increase by a factor of 3, but the computation of each iteration will run 10 times faster.

Another great advantage of only approximating the gradient on a mini sample is the decreased risk of getting stuck in a local minimum or saddle point, where  $\mathbf{g} \rightarrow 0$ , but  $\beta \neq \hat{\beta}$ . This is especially useful in neural network applications, where it is not given that we have a convex cost function.

The implementation of SGD is outlined in Listing 1. An `epoch` is defined as a full run through the full set of samples. The `learning_schedule()`s could be as in Equations (6) and (7).

The `cost_function_gradient()`s that we will use in this report are Mean Squared Error (MSE), used in OLS and Ridge regression methods; and Cross Entropy (CE), used in the logistic regression method. MSE cost function gradient is given by

$$\mathbf{g}_{mse}^{(n)} = \frac{\partial \mathcal{C}_{mse}^{(n)}}{\partial \beta} = \frac{1}{n}(\mathbf{X}^\top \mathbf{X})\beta^{(n)} - \mathbf{X}^\top \mathbf{y} + \lambda \beta^{(n)}, \quad (9)$$

where  $\mathbf{y}$  is the response variable vector and  $\lambda$  is the  $L_2$  regularization parameter [1].

The CE cost function gradient is given by

$$\mathbf{g}_{ce}^{(n)} = \frac{\partial \mathcal{C}_{ce}^{(n)}}{\partial \beta} = \frac{1}{n}(\mathbf{X}^\top \mathbf{P}^{(n)} - \mathbf{X}^\top \mathbf{Y}) + \lambda \beta^{(n)}. \quad (10)$$

Here,  $\mathbf{Y} \in \mathbb{R}^{n \times c}$  is the response variable matrix with  $n$  samples and  $c$  different classes of outcome. Each row  $\mathbf{y}_i$  has a single 1, and the rest 0s. Likewise,  $\mathbf{P} \in \mathbb{R}^{n \times c}$  is the probability matrix for the  $n$  samples and  $c$  classes following the *softmax* function:

$$\mathbf{P} = \text{softmax}(\mathbf{X}'\mathbf{B}). \quad (11)$$

$\mathbf{X}'$  is the design matrix with an intercept column added in front  $\mathbf{X}' = [\mathbf{1} \ \mathbf{X}]$ . The coefficient matrix  $\mathbf{B} \in \mathbb{R}^{(p+1) \times c}$ , accounting for the intercept, i.e. the variation in outcomes not explained by the features in  $\mathbf{X}$ .  $\mathbf{B}$  is given by

$$\mathbf{B} = [\beta_1 \ \beta_2 \ \cdots \ \beta_c], \quad \text{where} \quad \beta_j = [\beta_{0j} \ \beta_{1j} \ \cdots \ \beta_{pj}]^\top. \quad (12)$$

Each element  $p_{ij}$  in  $\mathbf{P}$  is given by

$$p_{ij} = \text{softmax}(\mathbf{X}'\mathbf{B})_{ij} = \frac{\exp(\mathbf{x}'_i \beta_j)}{\sum_c \exp(\mathbf{x}'_i \beta_c)}, \quad (13)$$

and the sum of each row  $\mathbf{p}_i$  in  $\mathbf{P}$  is 1;  $\sum_j p_{ij} = 1$ .

In Figure 2 we see a demonstration on some mock-up cases for each of the regression methods OLS and Ridge with SGD, and logistic with SGD. For the linear regression cases, a simple linear function with some noise has been used;  $y_i = 1 + 2x_i + \varepsilon_i$ , where  $\varepsilon \sim \mathcal{N}(0, 0.75^2)$ , and samples  $n = 100$ . We see that both methods make the fit quite well after 20 epochs and 10 mini batches with constant learning schedule with  $\eta = 0.5$  for OLS and  $\eta = 0.7, \lambda = 0.1$  for Ridge.

The logistic regression classification problem mock-up is shown in Table 1. The design matrix  $\mathbf{X}'$  is generated by adding some noise to an identity matrix of dimension  $3 \times 3$ , and adding the intercepts. Each of the three cases represent distinct outcomes,  $\mathbf{Y}$ . To the right in Figure 2 we see that when we use the result of the logistic regression  $\mathbf{B}$  to fit to  $\mathbf{X}'$ , correct outcomes are predicted with high confidence in each case.

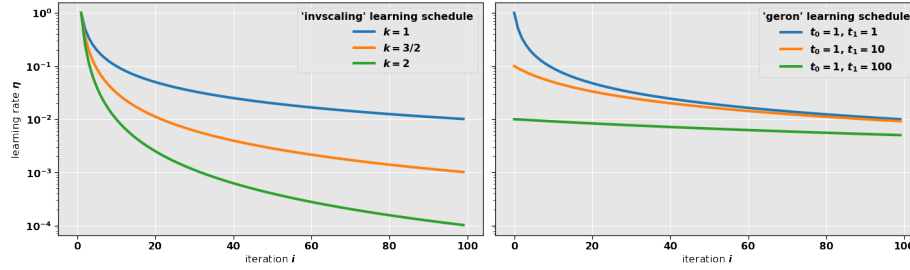


Figure 1: Two different learning schedules over the 100 first iterations. To the left we see the `invscaling` schedule in Python's `scikit-learn` module, as given in Equation (6). To the right we see the simple schedule suggested by Geron, as given in Equation (7).

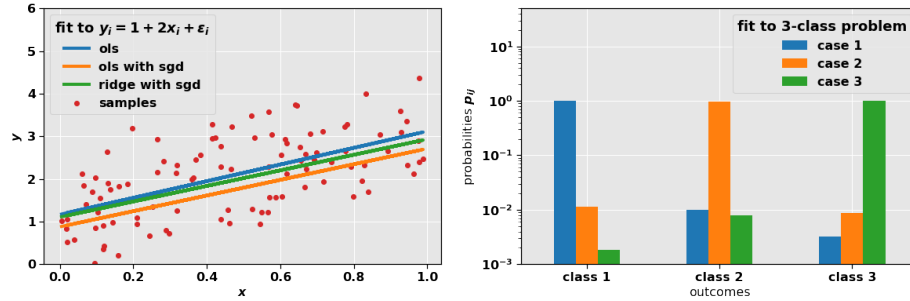


Figure 2: Demonstration of linear regression to the left, and logistic regression to the right using the SGD method. For linear regression, the fit is guaranteed to be best for the simple OLS case since the model we try to fit corresponds exactly to the function which generated the samples, but we see that both OLS with SGD and Ridge with SGD approach simple OLS. For logistic regression, a mock-up model of a  $3 \times 3$  design matrix  $\mathbf{X}$  plus intercept is fitted to a  $3 \times 3$  one-hot matrix  $\mathbf{Y} = \mathbf{I}$ . See Table 1. To generate the result,  $\mathbf{X}$  has been fed into the softmax function together with the result of the regression fit  $\mathbf{B}$  (Equation (13)). We see that we pick the right class with high confidence in each case.

Listing 1: Stochastic Gradient Descent algorithm. Cost function gradients and learning schedules may be defined independently of the implementation of SGD. This implementation runs for `epochs*batches` iterations, and does not sense if or when it converges.

```
// Declarations
betas = "initial coefficient values"
epochs = "number of runs through data set"
batches = "number of mini batches"
iteration_number = 0

// Inputs
data = "the design matrix"
targets = "the response variables"

// SGD process
FOR i = 1...epochs DO
  shuffle(data, targets)
  FOR j = 1...batches DO
    data_batch, targets_batch = "draw mini batch from full set w/o replacement"
    iteration_number++
    eta = learning_schedule(iteration_number, args...)
```

Table 1: Mock-up classification problem for demonstration of logistic regression. The table has three cases or samples 1,2,3, which are represented with the design matrix  $\mathbf{X}'$  in the middle and the one-hot response variable matrix  $\mathbf{Y}$  to the right. See the result of the demo to the right in Figure 2.

case	intercept	$p_1$	$p_2$	$p_3$	class 1	class 2	class 3
1	1	1.548814	0.715189	0.602763	1	0	0
2	1	0.544883	1.423655	0.645894	0	1	0
3	1	0.437587	0.891773	1.963663	0	0	1

## 2.2 Feed-Forward Neural Network

The linear and logistic regression methods each have their well-defined set of problems where they come to use; regression and classification, respectively. Artificial Neural Networks is another method, which is can be used for both types of problems, including classification problems with non-linear decision boundaries, like the XOR logical operator, or binary number representation. In this report we study a type of ANN called Feed-Forward Neural Network (FFNN).

Figure 3 gives an illustration of a typical FFNN. The layers consist of one or more *nodes* with a *activation function*. Each node-to-node connection has a *weight*, and the input to each node’s activation function is the weighted sum of the inputs from all the nodes in the former layer, plus a node-specific *bias*. Except for in the input layer, where the design matrix  $\mathbf{X}$  is fed in, all the activation functions are non-linear. This gives the network a highly non-linear behavior. The output of the first hidden layer is thus a non-linear transformation of the information in the input layer, and this process is repeated in the succeeding layers, including the output. At the output layer, the output is evaluated against a suitable cost function, and the error is back-propagated up through the network, which regulates the incremental update of weights and biases. This *training* process is then repeated for a given number of times, or until we have convergence.

What follows now is a mathematical derivation of the feed-forward, back-propagation and variable update of an FFNN. It is in large part based on [2], but translated into full matrix notation for simpler implementation in an array programming environment. The reader is encouraged to refer to this source for a thorough, general outline.

As indicated above, the input layer in our FFNN does not have an activation function, it only passes the design matrix  $\mathbf{X}$  on to the first hidden layer. If we let  $\mathbf{X} \in \mathbb{R}^{n \times p}$ , the input layer will have  $p$  nodes which are all connected to the  $m$  nodes of layer 1, the first hidden layer. Layer 1’s weight matrix  $\mathbf{W}_1 \in \mathbb{R}^{p \times m}$

is then given as

$$\mathbf{W}_1 = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ w_{p1} & w_{p2} & \cdots & w_{pm} \end{bmatrix}_{(1)}, \quad (14)$$

and its biases are

$$\mathbf{b}_1 = [b_1 \quad b_2 \quad \cdots \quad b_m]_{(1)}. \quad (15)$$

The resulting inputs  $\mathbf{Z}_1$  to layer 1's activation function  $f_1$  then become

$$\mathbf{Z}_1 = \mathbf{X}\mathbf{W}_1 + \mathbf{B}_1, \quad \text{where } \mathbf{Z}_1, \mathbf{B}_1 \in \mathbb{R}^{n \times m}. \quad (16)$$

Here,  $\mathbf{B}_1$  is the bias matrix; an  $n \times m$  matrix with  $\mathbf{b}_1 \in \mathbb{R}^m$  repeated in all its  $n$  rows.

Layer 1's activation function output  $\mathbf{A}_1$  is given by

$$\mathbf{A}_1 = f_1(\mathbf{Z}_1) \in \mathbb{R}^{n \times m}, \quad (17)$$

which is fed into the preceding layer, where this process is repeated. Here we note that this output  $\mathbf{A}_1$  is a non-linear transformation of the input  $\mathbf{X}$  to the network, which is true also for the remaining layers, including the output layer.

Let  $L$  denote the output layer in our FFNN and  $\mathcal{C}(\theta)$  the cost function of our model, where  $\theta$  may be any variable which influences its value. We evaluate the output of our FFNN against this cost function and update the output layer's weights and biases according to the gradient of the cost function with respect to each set of variables  $\mathbf{W}_L, \mathbf{b}_L$ ;

$$\begin{aligned} \mathbf{W}_L &\leftarrow \mathbf{W}_L - \eta \frac{\partial \mathcal{C}}{\partial \mathbf{W}_L}, \\ \mathbf{b}_L &\leftarrow \mathbf{b}_L - \eta \frac{\partial \mathcal{C}}{\partial \mathbf{b}_L}, \end{aligned} \quad (18)$$

where  $\eta$  is the learning rate, a hyperparameter.

By the chain rule, it can be shown that

$$\begin{aligned} \frac{\partial \mathcal{C}}{\partial \mathbf{W}_L} &= \mathbf{A}_{L-1}^T \left( \frac{\partial \mathcal{C}}{\partial \mathbf{A}_L} \circ \frac{df_L}{d\mathbf{Z}_L} \right), \\ \frac{\partial \mathcal{C}}{\partial \mathbf{b}_L} &= \frac{\partial \mathcal{C}}{\partial \mathbf{A}_L} \circ \frac{df_L}{d\mathbf{Z}_L} \end{aligned} \quad (19)$$

where  $\circ$  denotes the element-wise Hadamard product. Here we will let  $\Delta_L$  denote this Hadamard product or error term in Equation (19);

$$\Delta_L = \frac{\partial \mathcal{C}}{\partial \mathbf{A}_L} \circ \frac{df_L}{d\mathbf{Z}_L}. \quad (20)$$

Equation (18) now becomes

$$\begin{aligned}\mathbf{W}_L &\leftarrow \mathbf{W}_L - \eta \mathbf{A}_{L-1}^\top \Delta_L \\ \mathbf{b}_L &\leftarrow \mathbf{b}_L - \eta \Delta_L\end{aligned}\quad (21)$$

The preceding layers up to but not including the input are then updated in a similar fashion by back-propagation of this error up through the network. Again by the chain rule, it can be shown that  $\Delta_{L-1}$  is given by

$$\Delta_{L-1} = (\Delta_L \mathbf{W}_L^\top) \circ \frac{df_{L-1}}{d\mathbf{Z}_{L-1}}. \quad (22)$$

For the general layer  $l \geq 1$ ,  $\Delta_l$  becomes

$$\Delta_l = (\Delta_{l+1} \mathbf{W}_{l+1}^\top) \circ \frac{df_l}{d\mathbf{Z}_l}, \quad (23)$$

and the rule for updating layer  $l$ 's weights and biases becomes

$$\begin{aligned}\mathbf{W}_l &\leftarrow (1 - \eta\lambda) \mathbf{W}_l - \eta \mathbf{A}_{l-1}^\top \Delta_l \\ \mathbf{b}_l &\leftarrow \mathbf{b}_l - \eta \Delta_l\end{aligned}\quad (24)$$

Here, we have added  $\lambda$ , the  $L_2$  regularization parameter, which follows by the cost function derivation with respect to  $\mathbf{W}_L$ ;

$$\mathcal{C}' = \mathcal{C} + \frac{\lambda}{2} \|\mathbf{W}_L\|_2^2 \rightarrow \frac{\partial \mathcal{C}'}{\partial \mathbf{W}_L} = \frac{\partial \mathcal{C}}{\partial \mathbf{W}_L} + \lambda \mathbf{W}_L. \quad (25)$$

Like for the Stochastic Gradient Descent method discussed in the previous section, the ANN training process is then repeated until convergence, when  $\|\mathcal{C}(\theta)\|_2 \leq \varepsilon$ , or for a defined number of times, which we will do in this report.

Neural networks are generally data hungry [2], so to make the ANN training process as efficient as possible, we will use the Stochastic Gradient Descent method here as well. Another advantage of this approach is that the scattered descent reduces the chance of having the learning process getting stuck in local minima or saddle points. This only involves putting the training process inside the double epochs-batches loop in Listing 1. An outline of the training process is given in Listing 2.

The `activation_function()`s and `activation_function_derivative()`s in Listing 2 are very simple non-linear functions. Many types and variants for each type exist, but in this project we limit ourselves to those listed in Table 2.

The `cost_function_derivative()`s in Listing 2 which we will use in this report are the derivatives with respect to the outputs  $\mathbf{A}_L$  of the Mean Squared



Error cost function (MSE) and the Cross Entropy cost function (CE). The MSE derivative is given by

$$\frac{\partial \mathcal{C}_{mse}}{\partial \mathbf{A}_L} = (\mathbf{A}_L - \mathbf{Y}), \quad (26)$$

and the CE derivative by

$$\frac{\partial \mathcal{C}_{ce}}{\partial a_L^{(ij)}} = \frac{a_L^{(ij)} - y^{(ij)}}{a_L^{(ij)}(a_L^{(ij)} - y^{(ij)})}. \quad (27)$$

Equation (27) is due to [2].

Figure 4 shows the simple mock-up regression and classification problems used for SGD demonstration in the previous chapter (Figure 2) solved by our FFNN.

For the regression problem we fit a network with 1 hidden layer. It has 2 nodes in the input and hidden layers, and 1 node in the output. The activation function is always sigmoid, and the cost function MSE. We have trained this network on  $n = 100$  samples  $y_i = 1 + 2x_i + \varepsilon_i$  for 100 epochs with mini batches of 10 samples and use  $\eta = 0.5$  and the constant training schedule. The 100 samples are then fed back in to make a prediction. We see that as expected the prediction is a non-linear curve, and that it closely follows the OLS fit, but with a slight bend.

Classification is done on the 3-class problem described in Table 1. Again the FFNN has 1 hidden layer, but with 4 nodes in the input and hidden layers, and 3 nodes in the output layer. The hidden layer activation function is ReLU, and the output layer function is softmax, which is suitable for multiclass problems. The cost function is Cross Entropy. We have trained the network for 100 epochs with batch size 1,  $\eta = 0.1$  and  $L_2$  regularization  $\lambda = 0.001$ . In the input data cases 1,2,3 correspond to classes 1,2,3, which is also the prediction of our FFNN when the training data is fed back into the network for testing.

Listing 2: Outline of the Feed-Forward Neural Network training process using a Stochastic Gradient Descent scheme. The weights and biases in each layer must be initiated with some value.

---

```
// Declarations
layers = "collection of layers in the FFNN"
epochs = "number of runs through data set"
batches = "number of mini batches"
iteration_number = 0
lmd = "L2 regularization parameter"

// Inputs
data = "the design matrix"
targets = "the response variables"

// SGD learning process
```

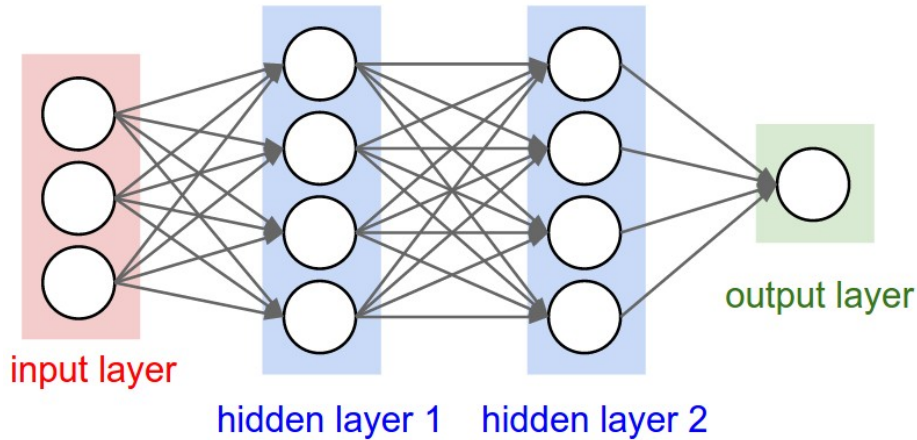


Figure 3: ANN with one input layer, two hidden layers, and an output layer. The design matrix is fed into the input layer, and is fed through the hidden layers and the output layer value is evaluated against a cost function. The error is then back-propagated and regulates the adjustment of the weights associated with each node-to-node connection and the bias of each node are in order to make a better fit in the next iteration. To introduce non-linearity, the output of each node is decided by the inputs and bias fed into a non-linear activation function. This network could be used for either regression, or binary classification. Illustration borrowed from [2].

```

FOR i = 1...epochs DO
  shuffle(data, targets)
  FOR j = 1...batches DO
    data_batch, targets_batch = "draw mini batch from full set w/o replacement"

    // Feed forward
    layer_outputs = data_batch
    FOR EACH layer IN layers DO
      "set layer's inputs by Equation (16)"
      "set layer's outputs by activation_function() Equation (17)"
    END FOR EACH

    // Calculate the cost function derivative with respect to output
    // layer outputs and targets
    error = cost_function_derivative("output layer's output", targets_batch)

    // Back-propagate the error by Equations (20) and (23)
    FOR EACH layer IN REVERSED(layers) DO
      "set layer's activation function_derivative() with respect to its inputs"
      "set layer's deltas by the back-propagated error"
      error = "error to back-propagate by this layer's new delta and weights"
    END FOR EACH
  END FOR j
END FOR i

```

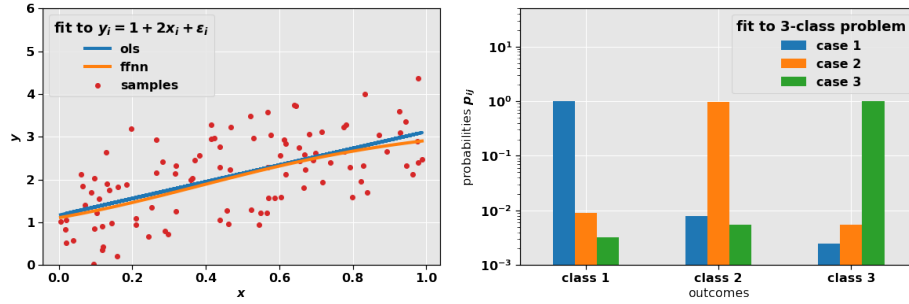


Figure 4: Demonstration of FFNN on a simple regression problem to the left, and a simple classification problem to the right. For linear regression, the fit is guaranteed to be best for the simple OLS case since the model we try to fit corresponds exactly to the function which generated the samples, but we see that the FFNN finds a non-linear fit which closely follows the OLS line. For the classification problem, we predict three cases with three features and three classes of outcome. In the input data cases 1,2,3 correspond to classes 1,2,3, respectively. See Table 1 for details. We see that we pick the right class with high confidence in each case. Compare with Figure 2, which shows the same set of problems solved by linear and logistic regression methods.

```

END FOR EACH

// Update weights and biases
iteration_number++
eta = learning_schedule(iteration_number, args...)
FOR EACH layer IN layers DO
    "update layer's weights by Equation (24) with step eta, reg. lmd"
    "update layer's biases by Equation (24) with step eta"
END FOR EACH
END FOR
END FOR

```

---

Table 2: Various activation functions and their derivatives. More variants and types exist, but these are the ones used in this report. ReLU, leaky-ReLU and unit-step derivatives are not defined at  $z = 0$ , but we approximate them as 0 here. Also, the derivative of unit-step is of course always 0, but to progress the learning process it is defined as shown here. The unit-step is usually best suited for classification problems, and softmax especially for multiclass problems. For linear regression problems, sigmoid or one of the ReLUs can be used.

name	$f(z)$	$f'(z)$	asymptote
ReLU	$\max(0, z)$	1 if $z > 0$ ; 0 else	$0, +\infty$
leaky-ReLU	$\max(0.01z, z)$	1 if $z > 0$ ; 0.01 else	$-\infty, +\infty$
unit-step	1 if $z > 0$ ; 0 else	set to 1 if $z > 0$ ; 0 else	$0, +1$
sigmoid	$1/(1 + e^{-z})$	$\text{sigmoid}(z)(1 - \text{sigmoid}(z))$	$0, +1$
softmax	$e^{z_{ij}} / \sum_c e^{z_{ic}}$	$\text{softmax}(z_{ij})(1 - \text{softmax}(z_{ij}))$	$0, +1$

### **3 Results**

- 3.1 Stochastic Gradient Descent: Linear Regression**
- 3.2 Stochastic Gradient Descent: Logistic Regression**
- 3.3 Feed-Forward Neural Network: Regression**
- 3.4 Feed-Forward Neural Network: Classification**

## 4 Discussion and Conclusion

## References

- [1] Olav Fønstelien. FYS-STK4155 H20 - Project1; An Assessment of OLS, Ridge and LASSO Regression on Areally Distributed Data. <https://github.com/fonstelien/FYS-STK4155/blob/master/project1/project1.pdf>, [Online; accessed 11-November-2020].
- [2] Morten Hjort-Jensen. FYS-STK4155 Applied Data Analysis and Machine Learning, Lecture notes Fall 2020. <https://compphysics.github.io/MachineLearning/doc/web/course.html>, [Online; accessed 11-November-2020].
- [3] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, Cambridge, Massachusetts, 2012.
- [4] scikit-learn, SGDRegressor class. [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.SGDRegressor.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html), [Online; accessed 11-November-2020].
- [5] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, California, 2019.