# CS 5350/6350: Machine Learning Fall 2020

Homework 6

**Alejandro Palomino**

Handed out: 12 November, 2020
Due date: 03 December, 2020

## 1 Logistic Regression

1. [5 points] What is the derivative of the function $g(\mathbf{w}) = \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i))$ with respect to the weight vector?

   $\rightarrow$ Using the chain rule $\frac{d}{dz} f(h(z)) = f'(h(z)) \times h'(z)$ where $f(h(z)) = \log(A)$ and $h(z) = 1 + \exp(B\mathbf{w}^T)$.

$$f'(h(z)) = \frac{d}{dz} \log(A) = \frac{1}{A} = \frac{1}{1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)}$$

$$h'(z) = \frac{d}{dz} 1 + \exp(B\mathbf{w}^T) = B \times \exp(B\mathbf{w}^T) = -y_i \mathbf{x}_i \exp(-y_i \mathbf{w}^T \mathbf{x}_i)$$

   Putting these two together yields the final derivative and recalling the sigmoid function $\boldsymbol{\sigma}(z) = (1 + e^{-z})^{-1}$:

$$\frac{\partial g(\mathbf{w})}{\partial \mathbf{w}} = -y_i \mathbf{x}_i \frac{e^{-y_i \mathbf{w}^T \mathbf{x}_i}}{1 + e^{-y_i \mathbf{w}^T \mathbf{x}_i}} = -y_i \mathbf{x}_i \frac{1}{1 + e^{y_i \mathbf{w}^T \mathbf{x}_i}} = -y_i \mathbf{x}_i \ \boldsymbol{\sigma}(-y_i \mathbf{w}^T \mathbf{x}_i)$$

2. [5 points] The inner most step in the SGD algorihtm is the gradient update where we use a single randomly chosen example instead of the entire dataset to compute a stochastic estimate of the gradient. Write down the objective where the entire dataset is composed of a single example, say $(\mathbf{x}_i, y_i)$.

   $\rightarrow$ The objective function for the logistic regression over a dataset $S = \{(\mathbf{x}_i, y_i)\}, \mathbf{x} \in \mathfrak{R}^d, y \in \{-1, 1\}$ is:

$$\min_{\mathbf{w}} \left\{ \sum_{i=1}^{m} \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)) + \frac{1}{\sigma^2} \mathbf{w}^T \mathbf{w} \right\}$$

   With just a single data point $\mathbf{x}_0$, we no longer have a summation because we have just a single training sample yielding:

$$\min_{\mathbf{w}} \left\{ \log(1 + \exp(-y_0 \mathbf{w}^T \mathbf{x}_0)) + \frac{1}{\sigma^2} \mathbf{w}^T \mathbf{w} \right\}$$

3. [5 points] Derive the gradient of the SGD objective for a single example (from the previous question) with respect to the weight vector.

   → The weight vector update requires that we take the derivative of the logistic regression objective ($\nabla J$), indicating the steepest incline, and move in the opposite (negative direction). Given that we have just 1 data point, we exclude the summation in the objective function leaving:

   $$J(\mathbf{w}) = \log(1 + \exp(-y_0 \mathbf{w}^T \mathbf{x}_0)) + \frac{1}{\sigma^2} \mathbf{w}^T \mathbf{w}$$

   Taking the derivative:

   $$\frac{\partial J}{\partial \mathbf{w}} = -y_0 \mathbf{x}_0 \ \boldsymbol{\sigma}(-y_0 \mathbf{w}^T \mathbf{x}_0) + \frac{2\mathbf{w}}{\sigma^2}$$

   An updated weight is then $\mathbf{w}' = \mathbf{w} - \eta \frac{\partial J}{\partial \mathbf{w}}$ where $\eta$ is the learning rate or SGD step size and $\boldsymbol{\sigma}$ is the sigmoid function.

4. [15 points] Write down the pseudo code for the stochastic gradient algorithm using the gradient from previous part.

   Hint: The answer to this question will be an algorithm that is similar to the SGD based learner we developed in the class for SVMs.

   → We previously derived the gradient update for the logistic SGD as $\nabla J(\mathbf{w}) = -y_i \mathbf{x}_i \ \sigma(-y_i \mathbf{w}^T \mathbf{x}_i) + \frac{2\mathbf{w}}{\sigma^2}$. The learning rate, or SGD step size, is the hyper-parameter $\eta$. Thus, the logistic SGD algorithm can be written in pseudocode as:

   (a) Given a training set $S = \{(\mathbf{x}_i, y_i)\}, \mathbf{x} \in \mathfrak{R}^d, y \in \{-1, 1\}$

   (b) Initialize weights $\mathbf{w} = 0 \in \mathfrak{R}^d$

   (c) FOR epochs $t = 1 \ldots T$ :

   - SHUFFLE training examples $S' \leftarrow (\mathbf{x}_i, y_i) \in S$
   - FOR each training example $(\mathbf{x}_i, y_i) \in S'$ :
     - Update weights $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J(\mathbf{w})$

   (d) RETURN $\mathbf{w}$

# 2 Experiments

## 2.1 What to report

1. briefly describe the design decisions that you have made in your implementation.

2. Report the best hyper-parameters, cross-validation accuracy and test set accuracy.

3. Plot corresponding loss function at each epoch. For the SVM over trees, you should plot the loss function for the SVM.

## 2.2 Algorithms to Compare

Table 1: Summary results table.

| | | Performance | | | |
|---|---|---|---|---|---|
| **Learner** | **Best hyper-parameters** | **CV Avg. Accuracy** | **Training Epochs** | **Training Accuracy** | **Testing Accuracy** |
| SVM | Lr = 0.0001, C = 1000; tau = 6.8e3 | 0.766 | 26 | 0.859 | 0.815 |
| Logistic Regression | Lr = 0.1, sig2 = 10000; tau = 7.3 | 0.766 | 14 | 0.835 | 0.789 |
| SVM over Trees | Lr = 0.001, C = 10; depth = 8; tau = 407 | 0.867 | 4 | 0.890 | 0.832 |

1. [20 points] **Support Vector Machine**

   → **Design**: This implementation of SVM (soft SVM) is programmed in Python 3.7. The following paragraphs briefly describe the principal modules and design decisions which comprise this implementation. Note elements of the SVM implementation are re-used for Logistic Regression and SVM over Trees.

   **loadData**: This function reads in cross-validation (CV), training, and testing data from the 'data/csv-format/' directory input and returns Padas dataframes. For the CV data, the first fold is read in as the validation set (446,206). The remaining folds are appended as the CV training set (1784,206). This process is repeated across all 5 folds and stored as a dictionary comprising 'dataCV[fold]['val']' and 'dataCV[fold]['trn']. The bias term is folded into the weights throughout data preparation.

   **runSVM-CV**: This function conducts SVM cross-validation (handles CV data, runs SVM algorithm, records performance). It takes in 'dataCV', an early stopping threshold (set to 'None' for CV), and returns the best algorithm performance across all hyper-parameters (HP). CV is conducted over a fixed learning period of 10 epochs for each combination of HP and each fold in 'dataCV.' Then, the 'svm' function is called returning the best weight learned with that combination of HP and CV fold. Given the best weight, the 'accuracy' function is called to evaluate the best weights' performance on the folds 'val' dataset. If this evaluation results in improved validation accuracy, the HP, validation accuracy, and objective function value are stored. Best hyper-parameters are selected as the mode of best algorithm performance across 3 trials and the early stopping threshold for training is defined as 1% of the average objective function value $J_{CV_{avg}}$ over all 5 folds and 3 trials. SVM CV results are presented in Table 2 with the best hyper-parameters $lr$ learning rate, $C$ tradeoff parameter, and early stopping threshold $\tau$

   **svm**: This function implements the SVM algorithm. It takes in data, HP (learning rate and tradeoff), early stopping threshold, and maximum epochs and returns best

weight, learning curve, objective function value and losses. For each epoch, the input dataset is shuffled, and weights are updated according to IF $y_i \mathbf{w}^T X_i \leq 1$, THEN $w' = (1 - \gamma_t)w + \gamma_t C y_i X_i$ ELSE $w' = (1 - \gamma_t)w$ for all shuffled $i \in X$. The learning rate is updated according to $\gamma_t = \gamma_0/(1 + t)$. At the end of each epoch, the resulting weights are evaluated in the function 'evalEP-SVM' which returns the epoch accuracy and training loss. If an epoch improves accuracy, its best accuracy and best weights are recorded. Training is stopped early, if the difference between 3 consecutive objective functions is less than the early stopping threshold $\tau = 0.01 * J_{CV_{avg}}$.

**evalEP-svm**: This function evaluates the performance of the current training epoch. It takes in validation data, epoch weights, and the tradeoff parameter to calculate epoch accuracy, objective function, and empirical loss. Where the regularizer $R = 0.5\mathbf{w}^T\mathbf{w}$, empirical loss $L = max(0, 1 - y_i \mathbf{w}^T X_i)$, and objective function is then $J = R + C \times L$.

**runSVM-trn**: This function conducts SVM training by running the SVM algorithm over the training dataset with the best HP from CV, and returns performance metrics. Training results are recorded as a dictionary 'svmTrn['w'], svmTrn['Acc'], svmTrn['LC'], svmTrn['Obj'], svmTrn['Losses']'.

After 'runSVM-trn' the learning curve and training loss are plotted by 'plot-learning' and 'plot-loss' given the best weights and HP discovered previously.

**runSVM-test**: This function takes in the test dataset and best weights discovered during training to evaluate accuracy performance on the test data using the accuracy function.

**accuracy**: This function predicts labels $y'$ given a dataset $X$ and weights $w$ and evaluates those predicted labels against the true labels $y$. IF $\mathbf{w}^T X_i \geq 0$, then $y'_i = 1$ ELSE $y'_i = 0$. The number of correctly predicted labels is divided by the total tested to return a percent accuracy.

Table 2: SVM cross-validation results.

| Run 1 | | | | | | Run 2 | | | | | | Run 3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Fold** | **Ep** | **lr** | **C** | **acc** | **obj** | **Fold** | **Ep** | **lr** | **C** | **acc** | **obj** | **Fold** | **Ep** | **lr** | **C** | **acc** | **obj** |
| 1 | 10 | 0 | 1000 | 0.776 | 7.20E+05 | 1 | 10 | 0 | 1000 | 0.785 | 7.13E+05 | 1 | 10 | 0 | 1000 | 0.785 | 7.88E+05 |
| 2 | 10 | 0 | 1000 | 0.805 | 7.35E+05 | 2 | 10 | 0 | 1000 | 0.803 | 7.66E+05 | 2 | 10 | 0 | 1000 | 0.787 | 7.90E+05 |
| 3 | 10 | 0 | 1000 | 0.771 | 7.00E+05 | 3 | 10 | 0 | 1000 | 0.769 | 7.07E+05 | 3 | 10 | 0 | 1000 | 0.769 | 6.97E+05 |
| 4 | 10 | 0 | 1000 | 0.769 | 6.94E+05 | 4 | 10 | 0 | 1000 | 0.769 | 6.99E+05 | 4 | 10 | 0 | 1000 | 0.760 | 7.29E+05 |
| 5 | 10 | 0 | 1000 | 0.780 | 7.89E+05 | 5 | 10 | 0 | 100 | 0.749 | 9.80E+04 | 5 | 10 | 0 | 1000 | 0.769 | 7.08E+05 |

| | SVM Cross-Validation Performance | | | | | |
|---|---|---|---|---|---|---|
| | **Time** | **Ep** | **lr** | **C** | **accuracy** | **obj** |
| **Average** | 0.43 m | 10.00 | 0.0001 | 940 | 0.776 | 6.89E+05 |
| **Mode - HP** | -- | -- | 0.0001 | 1000 | -- | -- |

→ **Results**: Summary results for the best hyper-parameters, the cross-validation accuracy and the the test set accuracy are presented in Table 1. The SVM training loss plot is shown in Fig. 1.

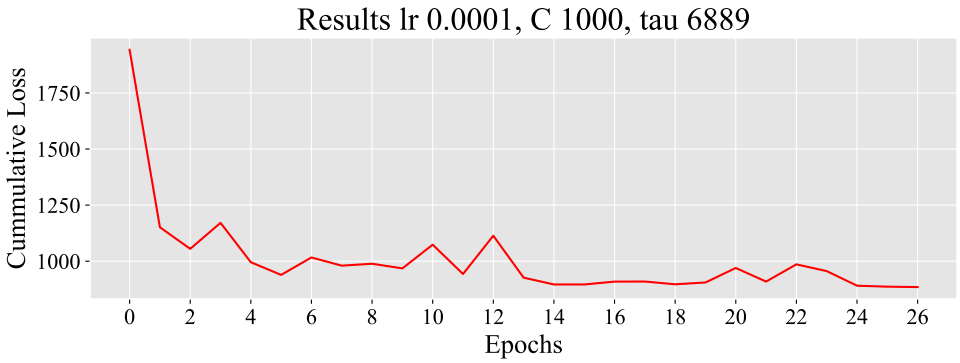


Figure 1: SVM training loss progress.

2. [20 points] **Logistic regression**

→ **Design**: This implementation of Logistic Regression is programmed in Python 3.7. The following paragraphs briefly describe the principal modules and design decisions which comprise this implementation. Logistic Regression employs the same 'loadData' data as described for the SVM algorithm.

**runLogReg-CV**: This function runs cross-validation (CV) for the Logistic Regression. It differs from the 'runSVM-CV' only in hyper-parameters (HP) and the learning algorithm. Namely, 'runLogReg' runs the 'logReg' Logistic Regression function. Logistic Regression CV results are presented in Table 3.

**logReg**: This function implements Logistic Regression. It takes in a dataset $[X, y]$, HP, early-stopping threshold, and maximum epochs and returns the best weights, accuracy, and losses discovered after training. The Logistic Regression weight update is implemented as presented by the pseudocode in question 1.4. Weight update relies on the 'sigmoid' function defined as $S(z) = 1/(1 + e^{-z})$ implemented as a NumPy equation. After weight update, epoch accuracy and losses are evaluated by function 'evalEp-LogReg.' Again, epoch weights which improve accuracy are recorded. Finally, the same early stop condition is used as that of 'svm' (but note that the threshold is a function of CV average objective value).

**evalEP-LogReg**: This function evaluates the performance of the current training epoch. It takes in validation data, epoch weights, and the tradeoff parameter to calculate epoch accuracy, objective function, and empirical loss. Where the regularizer $R = \mathbf{w}^T\mathbf{w}/\sigma^2$, empirical loss $L = \log(1 + e^{-y_i\mathbf{w}^T X_i})$, and objective function is then $J = R + L$.

**runLogReg-trn**: This function conducts Logistic Regression training by running the 'logReg' algorithm over the training dataset with the best HP from CV, and returns performance metrics. Again, training results are recorded as a dictionary.

After 'runLogReg' the learning curve and training loss are plotted by 'plot-learning' and 'plot-loss' given the best weights and HP discovered previously.

**runLogReg-test**: This function takes in the test dataset and best weights discovered

during training to evaluate accuracy performance on the test data using the accuracy function.

Table 3: Logistic regression cross-validation results.

| Run 1 | | | | | | Run 2 | | | | | | Run 3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Fold** | **Ep** | **lr** | **sig2** | **acc** | **obj** | **Fold** | **Ep** | **lr** | **sig2** | **acc** | **obj** | **Fold** | **Ep** | **lr** | **sig2** | **acc** | **obj** |
| 1 | 10 | 0.1 | 1000 | 0.778 | 737.11 | 1 | 10 | 0.1 | 1000 | 0.785 | 737.78 | 1 | 10 | 0.1 | 1000 | 0.769 | 751.50 |
| 2 | 10 | 0.1 | 10000 | 0.785 | 678.80 | 2 | 10 | 0.1 | 1000 | 0.787 | 741.32 | 2 | 10 | 0.1 | 1000 | 0.787 | 741.03 |
| 3 | 10 | 1 | 10000 | 0.762 | 686.17 | 3 | 10 | 0.1 | 10000 | 0.751 | 668.60 | 3 | 10 | 1 | 10000 | 0.776 | 694.09 |
| 4 | 10 | 1 | 10000 | 0.765 | 677.07 | 4 | 10 | 1 | 10000 | 0.742 | 731.37 | 4 | 10 | 1 | 10000 | 0.756 | 700.97 |
| 5 | 10 | 0.01 | 1000 | 0.738 | 948.99 | 5 | 10 | 1 | 10000 | 0.758 | 754.57 | 5 | 10 | 1 | 10000 | 0.753 | 703.36 |

| | Logistic Regression Cross-Validation Performance | | | | | |
|---|---|---|---|---|---|---|
| | **Time** | **Ep** | **lr** | **sig2** | **accuracy** | **obj** |
| **Average** | 1.20 | 10.00 | 0.51 | 6400 | 0.766 | 730.18 |
| **Mode - HP** | -- | -- | 0.10 | 10000 | -- | -- |

→ **Results**: Summary results for the best hyper-parameters, the cross-validation accuracy and the the test set accuracy are presented in Table 1. The Logistic Regression training loss plot is shown in Fig. 2 with the best hyper-parameters $lr$ learning rate, $C$ tradeoff (or $\sigma^2$), and early stopping threshold $\tau$

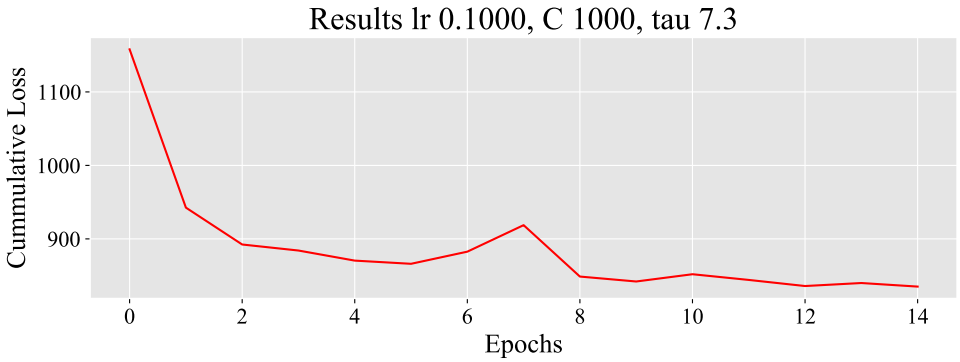


Figure 2: Logistic Regression training loss progress.

3. [20 points] **SVM over trees**

→ **Design**: This implementation of SVM over Trees is programmed in Python 3.7. The following paragraphs briefly describe the principal modules and design decisions which comprise this implementation. Although elements of the SVM implementation are re-used for this implementation, the first phase for SVM over Trees is a decision-tree based data transformation. For each depth $d = [1, 2, 4, 8]$ and then repeated 200 times, 10% of the training dataframe is randomly sampled with replacement. This data subset is then fed to the 'id3' algorithm to build trees. Trees are stored in a dictionary $t[d][i] \; \forall \; i \in 200$. These decision trees can then provide transformed versions the cross-validation, training, and testing datasets.

**id3**: This function implements the depth-limited ID3 algorithm as discussed in HW 2 (split on lowest entropy, recurse). It takes in data, the maximum depth limitation, initial depth = 0, and returns a decision tree. Depth limitation is included as a base

case in the recursion. Trees are constructed as nested dictionaries. Data is handled as NumPy arrays.

**transformData**: This function takes as an input the dictionary of trees at depths $d = [1, 2, 4, 8]$, a dataset and returns a transformed version of the input dataset as a dictionary with depths as the key. It does this by looping over all depths, all samples in the input data set, all trees at that depth (200), and predicting labels based on the current data sample and current tree using the function 'predictLbl.' The CV, training, and testing data are all transformed using this 200 decision approach in the 'transformData' function for all depths.

**predictLbl**: This function predicts a label $\{-1, +1\}$ given a data sample $X_i$ and decision tree. It does this by recursively traversing through the tree according to the values in $X_i$. IF the traversal discovers a dictionary, continue traversing, ELSE return the current value (i.e. the label).

Now, with decision tree transformed CV ('trn': (1784,201), 'val': (446,201)), training (2230,201), and testing data (558,201), SVM CV, training, and testing can proceed just as it did for the classical SVM implementation. Note: the column dimension shown above includes the label and results of the 200 trees. Cross-validation results for the SVM over trees is shown in Table 4.

Table 4: SVM over Trees cross-validation results.

| Run 1 | | | | | | |
|---|---|---|---|---|---|---|
| **Fold** | **Ep** | **d** | **lr** | **sig2** | **acc** | **obj** |
| 1 | 10 | 8 | 0.001 | 10 | 0.8834 | 5090 |
| 2 | 10 | 8 | 0.001 | 10 | 0.8789 | 4868 |
| 3 | 10 | 8 | 0.01 | 1 | 0.8543 | 687 |
| 4 | 10 | 8 | 0.0001 | 10 | 0.8475 | 4882 |
| 5 | 10 | 8 | 0.001 | 100 | 0.8722 | 55724 |

| Run 2 | | | | | | |
|---|---|---|---|---|---|---|
| **Fold** | **Ep** | | **lr** | **sig2** | **acc** | **obj** |
| 1 | 10 | 8 | 0.001 | 10 | 0.886 | 4949 |
| 2 | 10 | 8 | 0.0001 | 10 | 0.872 | 5124 |
| 3 | 10 | 8 | 0.001 | 10 | 0.863 | 4604 |
| 4 | 10 | 8 | 0.01 | 1 | 0.848 | 660 |
| 5 | 10 | 8 | 0.001 | 10 | 0.865 | 4821 |

| Run 3 | | | | | | |
|---|---|---|---|---|---|---|
| **Fold** | **Ep** | | **lr** | **sig2** | **acc** | **obj** |
| 1 | 10 | 8 | 0.001 | 10 | 0.890 | 5018 |
| 2 | 10 | 8 | 0.0001 | 100 | 0.872 | 41438 |
| 3 | 10 | 8 | 0.0001 | 10 | 0.850 | 4906 |
| 4 | 10 | 8 | 0.001 | 100 | 0.850 | 54256 |
| 5 | 10 | 8 | 1E-05 | 1000 | 0.868 | 414705 |

| SVM over Trees Cross-Validation Performance | | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Time** | **Ep** | **depth** | **lr** | **C** | **accuracy** | **obj** |
| **Average** | 1.76 | 10.00 | 8.00 | 0.0019 | 92.8 | 0.87 | 4.08E+04 |
| **Mode - HP** | -- | -- | 8.00 | 0.001 | 10.0 | | |

→ **Results**: Summary results for the best hyper-parameters, the cross-validation accuracy and the the test set accuracy are presented in Table 1. The SVM over trees training loss plot is shown in Fig. 3.

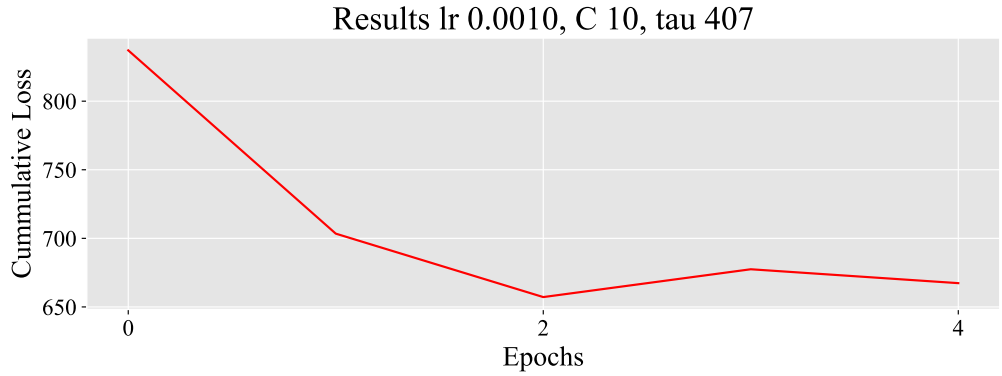


Figure 3: SVM over Trees training loss progress.

# 3 Neural Networks and the XOR function (30 points, extra credit)

Recall that linear classifiers cannot represent the XOR function defined over two inputs (shown below). Consider a two layer neural network, where all nodes have the *threshold activation*. That is, the activation at each node is simply the sign function. The hidden layer of the network has two units. The XOR data $(x_1, x_2)|y$ is $(-1,-1)|-1$, $(-1,+1)|+1$, $(+1,-1)|+1$, $(+1,+1) = -1$. Note: Answer to 3.2 on the next page.

1. Find a set of weights for this network such that it correctly classifies all the XOR examples.

   $\rightarrow$ The labels are calculated according to the forward propagation equations shown in equations (1)-(3) where $sgn$ is the sign, or linear threshold, function $(z \geq 0; +1)$. These equations are implemented in Excel to test weight values.

$$\mathrm{y} = sgn\left(w_{01}^{o} + w_{11}^{o} h_1 + w_{21}^{o} h_2\right) \tag{1}$$

$$h_2 = sgn\left(w_{02}^{h} + w_{12}^{h} x_1 + w_{22}^{h} x_2\right) \tag{2}$$

$$\mathrm{z}_1 = sgn\left(w_{01}^{h} + w_{11}^{h} x_1 + w_{21}^{h} x_2\right) \tag{3}$$

   The set of weights presented in Table 5 were discovered via trial and error in the Excel program shown.

Table 5: Weights for Threshold Activation NN.

**Weights**

| | 1 | 2 | b |
|---|---|---|---|
| w.out | 1.00 | -2.00 | -2.00 |
| w.1 | 1.00 | 0.50 | 1.00 |
| w.2 | 1.00 | 0.50 | -1.00 |

**Neural Network Calculations**

| Example | val_h1 | sgn(h1) | val_h2 | sgn(h2) | val_y' | y' | Loss |
|---|---|---|---|---|---|---|---|
| 1 | -0.50 | -1 | -2.50 | -1 | -1 | -1 | 0 |
| 2 | 0.50 | 1 | -1.50 | -1 | 1 | 1 | 0 |
| 3 | 1.50 | 1 | -0.50 | -1 | 1 | 1 | 0 |
| 4 | 2.50 | 1 | 0.50 | 1 | -3 | -1 | 0 |

2. Now suppose the activation functions in each node in the network is the sigmoid function. Is there a set of weights for this new network that correctly classifies the XOR examples?

   → The labels are calculated according to the forward propagation equations shown in equations (4)-(6) where $\sigma$ is the sigmoid activation function ($\sigma(z) = \frac{1}{1+e^{-x}}$) and $sgn^{0.5}$ is the threshold function ($z \geq 0.5; +1$). These equations are implemented in Excel to test weight values.

$$\mathrm{y} = sgn^{0.5}\left(w_{01}^{o} + w_{11}^{o} h_1 + w_{21}^{o} h_2\right) \tag{4}$$

$$h_2 = \sigma\left(w_{02}^{h} + w_{12}^{h} x_1 + w_{22}^{h} x_2\right) \tag{5}$$

$$\mathrm{z}_1 = \sigma\left(w_{01}^{h} + w_{11}^{h} x_1 + w_{21}^{h} x_2\right) \tag{6}$$

   The set of weights presented in Table 6 were discovered via trial and error in the Excel program shown.

Table 6: Weights for Sigmoid Activation NN.

**Weights**

| | 1 | 2 | b |
|---|---|---|---|
| w.out | 1.10 | -2.00 | 0.25 |
| w.1 | 1.20 | 0.60 | 1.00 |
| w.2 | 1.10 | 0.60 | -1.20 |

**Neural Network Calculations**

| | val_h1 | sig(h1) | val_h2 | sig(h2) | val_y' | y' | Loss |
|---|---|---|---|---|---|---|---|
| 1 | -0.80 | 0.31 | -2.90 | 0.05 | 0.4867 | -1 | 0 |
| 2 | 0.40 | 0.60 | -1.70 | 0.15 | 0.5996 | 1 | 0 |
| 3 | 1.60 | 0.83 | -0.70 | 0.33 | 0.5016 | 1 | 0 |
| 4 | 2.80 | 0.94 | 0.50 | 0.62 | 0.042 | -1 | 0 |