Computer Vision (7315)

2020 Computer Vision
Assignment 3: Deep Learning

Vandit Jyotindra Gajjar
School of Computer Science, The University of Adelaide
vanditjyotindra.gajjar@student.adelaide.edu.au
Student ID: a1779153

Abstract

Computer Vision, Deep Learning, and Machine Learning are those fields that are closely related to one another. Recently, Deep Learning and Machine Learning have improved Computer Vision about Image Classification, Localization, Segmentation, and Object Tracking, Detection, etc. [1]. This assignment/report will mainly focus on the Machine Learning parts which improve the Computer Vision tasks. In general, we will work on the fundamentals of the Machine Learning apparatus used in Computer Vision tasks. We will explore the simplest learning algorithm called 'A Perceptron' [2]. Further, we will work on the extension of this Perceptron to work with more than 2 classes and solves the classification problem. Lastly, we will work on one of the applications of Deep Learning where we detect the counterfeit banknotes. This report focuses to solve the assignment questions with detailed analysis and in-depth context relating to Machine Learning and Deep Learning apparatus used in Computer Vision tasks. The source code is publicly available at *https://github.com/Vanditg/COMP-SCI-7315---Computer-Vision*.

Task 1
Answer (a)

We have implemented the perceptron training algorithm for binary classification (i.e. class 0 or 1). The input to this perceptron is 2D points and by the end of the learning algorithm, this perceptron classifies this point to either 0 or 1. The source code of the task-1 implementation is publicly available at *https://github.com/Vanditg/COMP-SCI-7315---Computer-Vision/tree/master/Assignment%20-%203/Solution/task_1*. Figure 1 shows the single perceptron functionalities. As per the lecture notes, to build the perceptron models it requires the following steps:

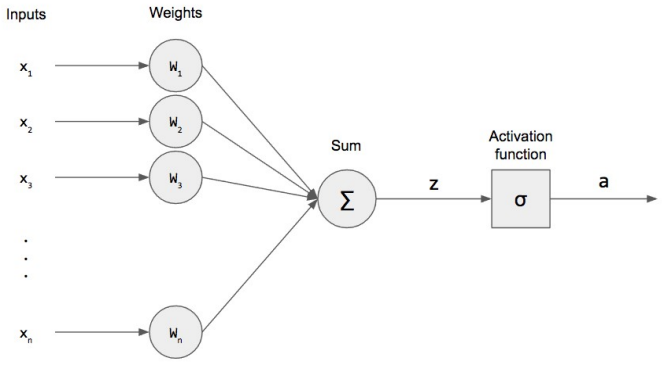


Figure 1. Single perceptron model. Best viewed in color and magnification. (Source - Google Image Search)

- Perceptron parameters initialization:

The perceptron model takes 2D input data points. The weight parameters of the model then multiply with the input data points to get some output result. Here, the weight size heavily depends on the input data points. Initially, we have been provided with the random initialization strategy to initialize our weights. We change the strategy to gradient descent as per lecture slides and algorithms. Figure 2 shows the necessary changes made in the notebook file to convert to gradient descent.

```
if res!=Y[i]:
    converged = False
    #Changing the stratergy to gradient descent
    self.w = self.w + (Y[i] - res) * l_r * X[i].reshape(-1, 1)
```

Figure 2. Gradient Descent Weight Initialization.

- Perceptron learning algorithm training
  - Matrix multiplication of weights and input 2D data points
  - Weight update

Here, we perform matrix multiplication between row and column, further we apply the Perceptron learning algorithm condition (Figure 3) on the output and update the weights correspondingly using gradient descent [3].

Algorithm: Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end
//the algorithm converges when all the inputs are classified correctly

Figure 3. Perceptron Learning Algorithm. (Source - Lecture Slides)

The activation function make value 1 or 0 based on the given condition. Figure 4 shows the necessary changes made on the iterative loop.

```
while not converged and epochs < max_epochs :

    # TODO
    # 1. add training code here that updates self.w
    # 2.  a criteria to set converged to True under the correct circumstances.
    converged, l_r = True, 0.01
    for i in range(len(X)):
        #Matrix Multiplication
        res = np.matmul(X[i],self.w)
        if res >= 0:
            res = 1.0
        else:
            res = 0.0
        if res!=Y[i]:
            converged = False
            #Changing the stratergy to gradient descent
            self.w = self.w + (Y[i] - res) * l_r * X[i].reshape(-1, 1)

    # curent strategy is random search (not good!)
    # self.w = np.random.randn(self.input_size,1)
```

Figure 4. Implementing the perceptron training algorithm as taught in the lecture.

In the notebook by changing the *number_of_samples* parameter, we can generate various sample sizes for both the class. Figure 5 shows the various sample sizes (i.e. 10, 50, 100) for both the classes.

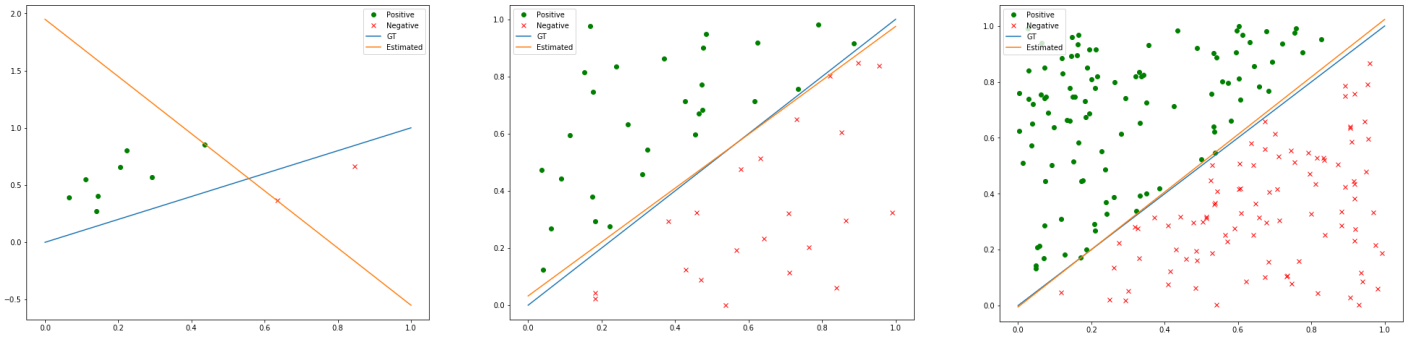


Figure 5. Various sample sizes generated by changing the *number_of_samples* parameter to 10, 50, and 100 (left to right). Best viewed in color and magnification.

Answer (b)

After implementing the perceptron learning algorithm, we see that in certain cases the estimated line can correctly classify all the samples, however, it does not agree with the ground-truth boundary. The core reason behind this situation is that in our learning algorithm, we set the condition that whenever the model converges, we will exit the loop thus training stops at that point. Depending on the variational number_of_samples parameter, we can observe such cases as shown in Figure 6. Whenever we increase the samples the distance between data points becomes less, and thus less space is available for convergence. So, due to the number of data points, this type of behavior seems to be normal. Furthermore, the Perceptron learning algorithm will not be able to classify all the samples, but eventually, it will attempt to find a line that possibly separates them. The possible disadvantage will be that whenever unseen data points take place in the data for classification, the perceptron learning algorithm will try to separate them in the best possible way which leads to good accuracy. This leads to some misclassification in some examples and leads to a mismatch in the estimated line and ground truth boundary. Figure 6 has several examples, which show this type of behavior.

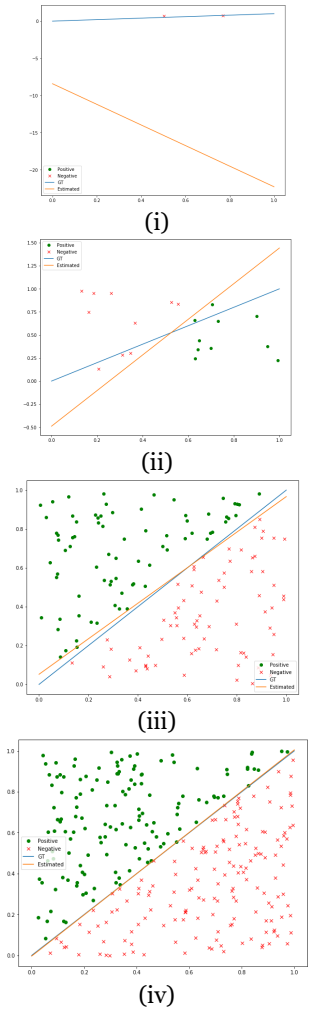


Figure 6. Comparison of estimated line and ground-truth boundary. Only the exact match in the last sub-image. (Sample size – 2, 20, 150, and 300 up to down). Best viewed in color and magnification.

Answer (c)

When we have performed various training sessions, we noticed unseen behavior of the boundary that was oscillating between two solutions and not able to reach 100% accuracy until max iteration reached. Figure 7 shows one of the sessions when we observe this type of behavior. The main reason behind this behavior is the gradient descent step size and this has been defined by the learning rate parameter, which was introduced in the modified algorithm. When the learning rate is high, the gradient descent takes big steps and constantly oscillates between positive and negative sides and not able to provide convergence and good accuracy results as it traps in the local minima. Figure 8 shows the general concept of gradient descent steps and learning rate. In our experiments, the training algorithm has a learning rate of 0.01, while the modified algorithm has a learning rate of 0.001.

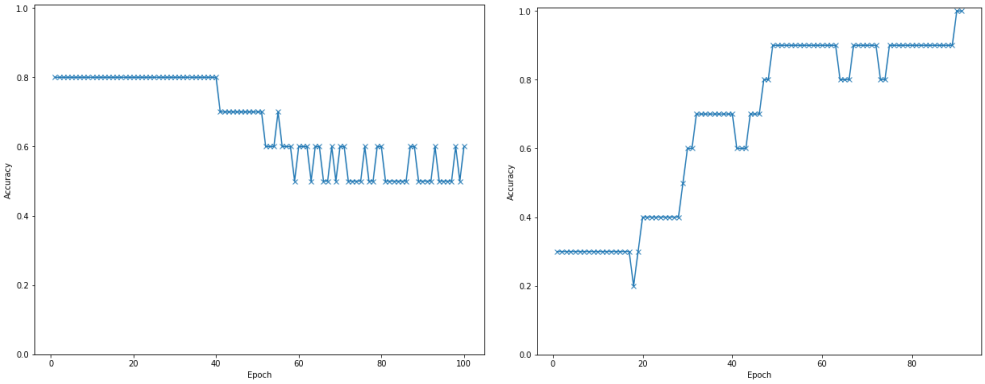


Figure 7. Left - Oscillation between positive and negative side and not reaching to 100% accuracy even max iteration reached. High learning rate - 0.01 leads model to not converge. Right – Model converges after changing the learning rate to 0.001. Best viewed in color and magnification.

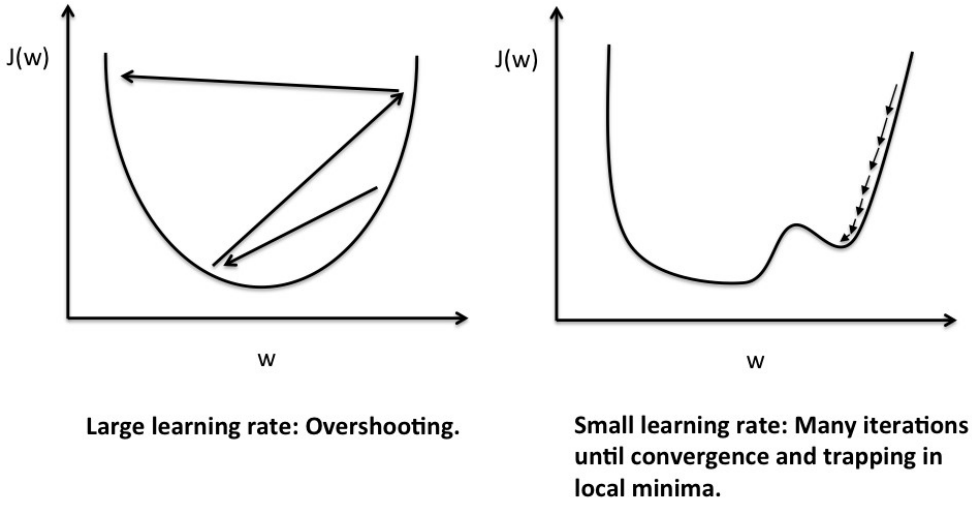


Figure 8. Gradient descent steps and the effect of learning rate. Best viewed in color and magnification. (Source - Google Image Search)

Answer (d)

The strategy of random initialization leads the perceptron learning algorithm to converge in a different number of epochs. However, in our experiments, we fix the number of epochs to 1000 and changes the sample size and learning rate to get the mean number of epochs. We run the algorithm for five times and take the mean of it for various combinations. We found in our experiments that if there are less number of input data points than the perceptron training learning algorithm works better. If we increase the data points, then the perceptron modified learning algorithm works better. In general, from the experiments, we claim that the perceptron modified learning algorithm works better as the model takes even more steps but ultimately it will converge at a certain point. If we increase the number of data points than we need to look at the learning rate and choose a smaller learning rate for convergence. In Table 1, we show the mean number of epochs required to converge the perceptron training learning algorithm and perceptron modified learning algorithm for 10, 50, 500, and 5000 sample sizes.

Table 1. The mean number of epochs required to converge the perceptron training and modified learning algorithm for sample sizes – 10, 50, 500, and 5000.

Sample Sizes	Perceptron training learning algorithm (learning rate - 0.01)	Perceptron modified learning algorithm (learning rate - 0.001)
10	73	226
50	44	249
500	130	75
5000	No convergence	17

Computer Vision (7315)

Task 2
Answer (a)

We extend the single perceptron learning algorithm from classifying 2 classes (i.e. 0 or 1) to 3 classes classification. Figure 9 shows the image of the dataset on which we have performed our experiments. It represents different classes represented by different colors.

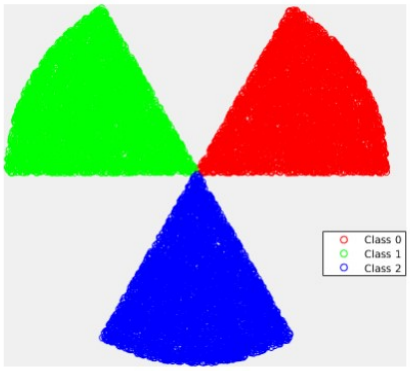


Figure 9. Dataset containing three classes. Best viewed in color and magnification. (Source - Assignment Problem PDF)

A hint has been provided to use one vs all classification approach [4], we modify the task 1 algorithm correspondingly. The source code of the task-2 implementation is publicly available at *https://github.com/Vanditg/COMP-SCI-7315---Computer-Vision/tree/master/Assignment%20-%203/Solution/task_2*. In perceptron learning, the model can be able to differentiate between 2 classes one at a time. We extend this by setting 1 class true while the other 2 classes false at a certain point. The basic difference between one vs all classification approach is shown in Figure 10. By this approach, we have 3 different base-cases in which we train 3 different model weight. The base-cases are shown in Table 2. In the implementation, we have changed the False values to zero while True value to 1 and train it as positive. The major changes in the implementation for all 3 cases are shown in Figure 11.

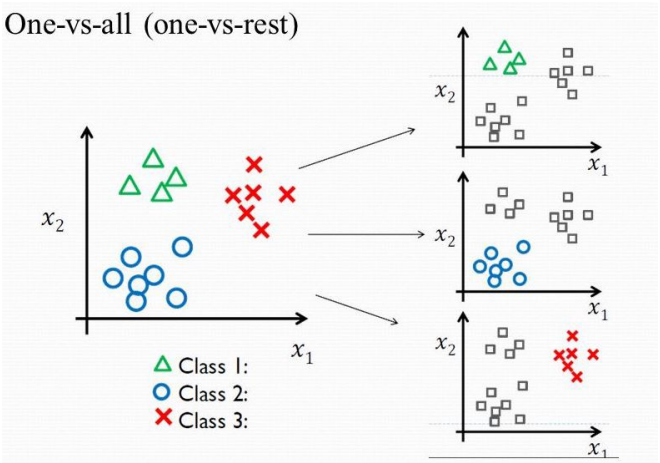


Figure 10. One vs all classification approach for multi-class classification. Best viewed in color and magnification. (Source - Google Image Search)

Table 2. Base-cases for 3 class one vs all classification approach.

Base-cases	True	False
1	Class 0	Class 1 and Class 2
2	Class 1	Class 0 and Class 3
3	Class 2	Class 0 and Class 1

```
while not converged and epochs < max_epochs :

    # TODO
    # 1. add training code here that updates self.w
    # 2.  a criteria to set converged to True under the correct circumstances.
    converged, l_r = True, 0.001
    for i in range(len(X)):
        #----------------------Base-Case-1--------------#
        res = np.matmul(X[i], self.w_0) #matrix row and weight multiplication
        if res >= 0:
            res = 1.0
        else:
            res = 0.0
        if res != Y_0[i]:
            converged = False
            self.w_0 = self.w_0 + (Y_0[i] - res) * l_r * X[i].reshape(-1,1)
        #----------------------Base-Case-2--------------#
        res = np.matmul(X[i], self.w_1) #matrix row and weight multiplication
        if res >= 0:
            res = 1.0
        else:
            res = 0.0
        if res != Y_1[i]:
            converged = False
            self.w_1 = self.w_1 + (Y_1[i] - res) * l_r * X[i].reshape(-1,1)
        #----------------------Base-Case-3--------------#
        res = np.matmul(X[i], self.w_2) #matrix row and weight multiplication
        if res >= 0:
            res = 1.0
        else:
            res = 0.0
        if res != Y_2[i]:
            converged = False
            self.w_2 = self.w_2 + (Y_2[i] - res) * l_r * X[i].reshape(-1,1)
```

Figure 11. Major changes in the implementation - one vs all classification approach.

Answer (b)

We modify the *compute_accuracy* function and plot the accuracy over time for the training data. Here, we measure the accuracy of each class (i.e. class 0, class 1, and class 2) in which model predicts the right value, we then sum up all those values and divide over the total number of values. The correct implementation of the function is shown in Figure 12. The accuracy plot is shown in Figure 13, where the learning rate has been changed from 0.01 to 0.001.

```python
# This computes the accuracy of our current estimate
def compute_train_accuracy(self,X,Y):
    res_0 = np.matmul(X,self.w_0)
    Y_bar0 = (res_0 >= 0)

    res_1 = np.matmul(X,self.w_1)
    Y_bar1 = (res_1 >= 0)

    res_2 = np.matmul(X,self.w_2)
    Y_bar2 = (res_2 >= 0)

    Y_bar = Y_bar0 + Y_bar1 + Y_bar2

    accuracy = np.count_nonzero(Y_bar) / np.float(Y_bar.shape[0])
    self.history.append(accuracy)
    print("Accuracy : %f " % (accuracy))
    self.draw(X)
```

Figure 12. Implementation of *compute_accuracy* function.

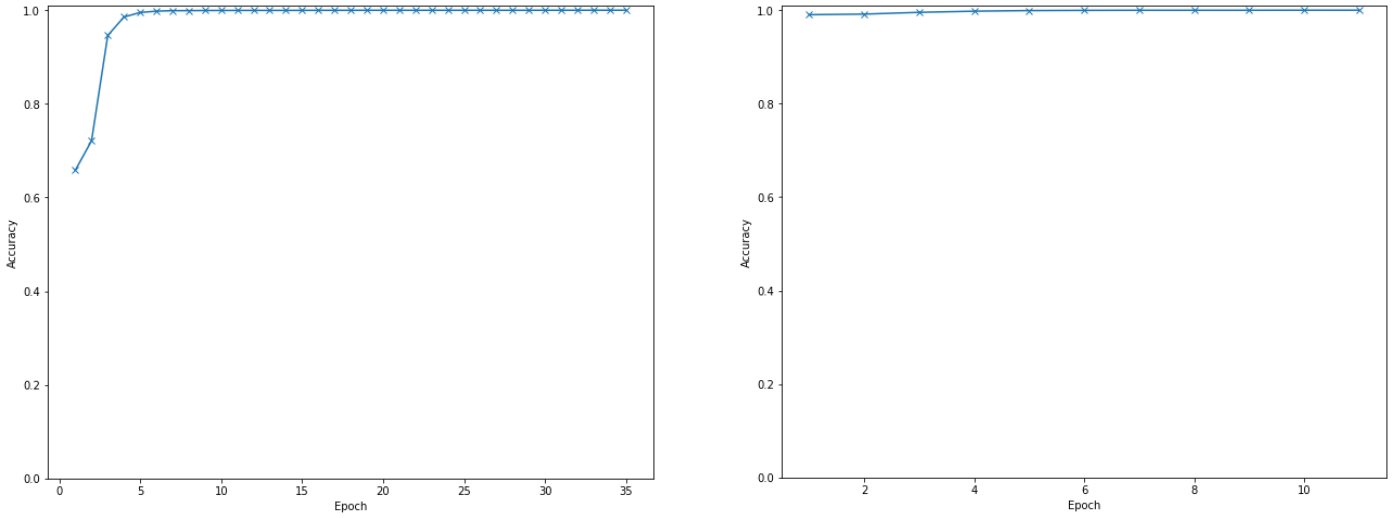


Figure 13. Accuracy vs epoch plot of the model. Left - learning rate 0.001, Right - learning rate 0.01. Best viewed in color and magnification.

Answer (c)

We have implemented the *draw()* function to visualize the decision boundaries on the given dataset for the perceptron modified learning algorithm shown in Figure 14. We will further visualize the steps of convergence in Figure 15.

```
def draw(self,X):

    pl.close()
    pl.figure(figsize=[10,8])
    pl.xlim((0,1))
    pl.ylim((0,1))
    res_1, res_2, res_3 = np.matmul(X,self.w_0).squeeze(), np.matmul(X,self.w_1).squeeze(), np.matmul(X,self.w_2).squeeze()
    P1, P2, P3 = X[res_1 >= 0,:], X[res_2 >= 0,:], X[res_3 >= 0,:]

    pl.plot(P1[:,0],P1[:,1],'ro', label = 'Class 0')
    pl.plot(P2[:,0],P2[:,1],'go', label = 'Class 1')
    pl.plot(P3[:,0],P3[:,1],'bo', label = 'Class 2')

    x_1, x_3, x_2 = np.linspace(0.55,-0.48), np.linspace(-1,1), np.linspace(-0.55,0.48)
    pl.plot(x_3, np.linspace(-0.00,-0.00), label = 'Ground-Truth-1')
    pl.plot(x_2, np.linspace(-0.9,0.9), label = 'Ground-Truth-2')
    pl.plot(x_1,np.linspace(-0.9,0.9), label = 'Ground-Truth-3')

    a1, b1, c1 = self.w_0[0], self.w_0[1], self.w_0[2]
    pl.plot(x_1, -a1/b1 * x_1 - c1/b1, label = 'Estimated-Line-1')

    a2, b2, c2 = self.w_1[0], self.w_1[1], self.w_1[2]
    pl.plot(x_2, -a2/b2 * x_2 - c2/b2, label = 'Estimated-Line-2')

    a3, b3, c3 = self.w_2[0], self.w_2[1], self.w_2[2]
    pl.plot(x_3, -a3/b3 * x_3 - c3/b3, label = 'Estimated-Line-3')

    pl.axis('tight')
    pl.legend()
    display.clear_output(wait=True)
    display.display(pl.gcf())
    time.sleep(0.1)
```

Figure 14. Implementation of the *draw* function in the perceptron modified learning algorithm.

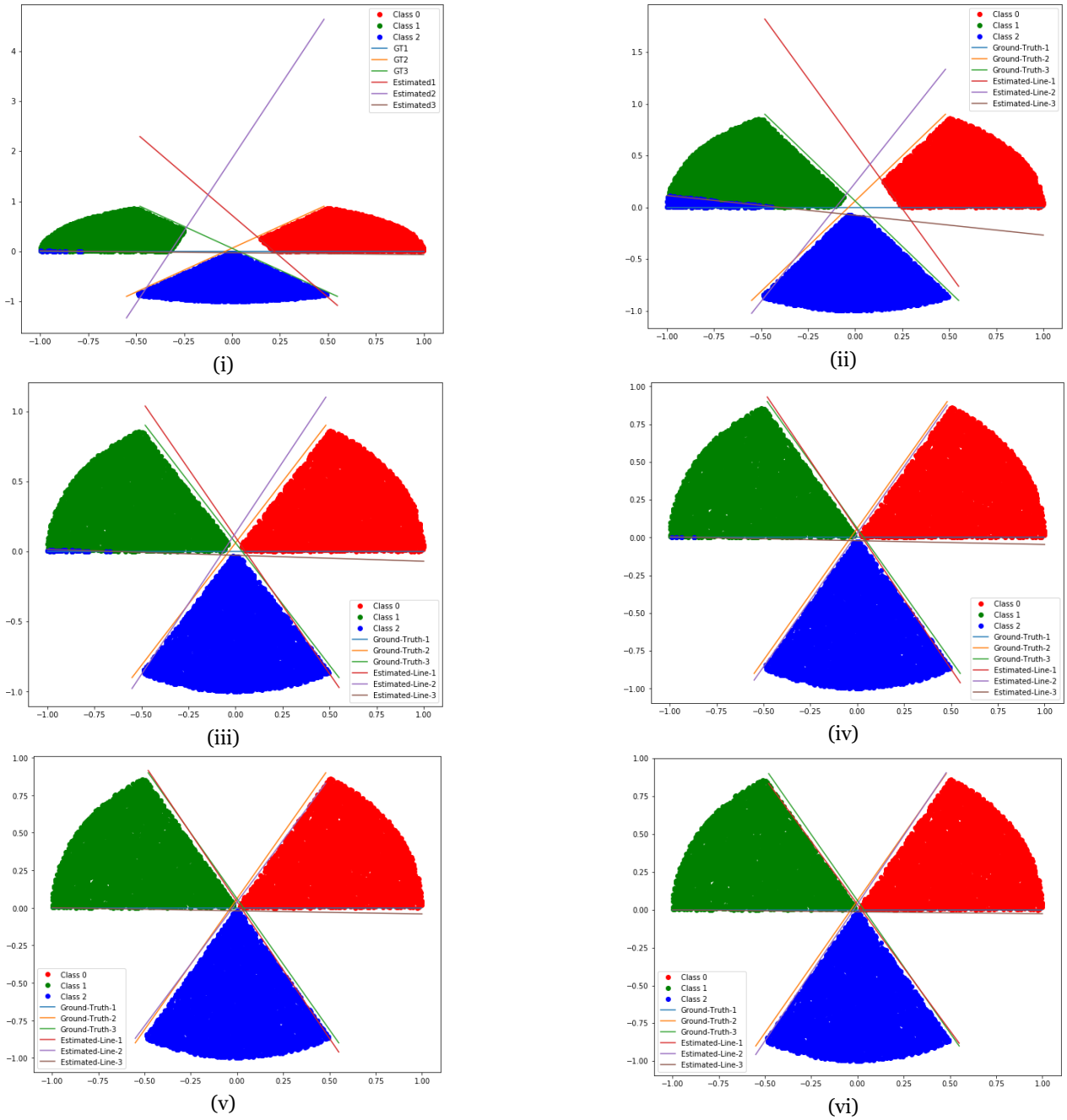


Figure 15. Steps of convergence. (i) Epoch - 1 (ii) Epoch - 5 (iii) Epoch - 10 (iv) Epoch - 15 (v) Epoch - 20 (vi) Epoch - 21. Best viewed in color and magnification.

From figure 15, we can see that the different classes can be separated by line based on a linear classifier. Linear classifiers do not have any curve while plotting. So the different classes can be differentiating by a line from each other leads linear classifier to remain a good choice and by using this in the given multi-class classification we can achieve maximum accuracy. We can conclude that if a problem is linear as our task and its class boundaries can be approximated well with linear hyperplanes, then linear classifiers are often more accurate than non-linear classifiers. If a problem is linear, it is certainly good to use a simple linear classifier rather than a non-linear classifier.

Computer Vision (7315)

Task 3

Answer (a)

We have implemented a 3 - layer neural network for classification of the non-linear separable XOR gate output. The source code of the task-3 implementation is publicly available at https://github.com/Vanditg/COMP-SCI-7315---Computer-Vision/tree/master/Assignment%20-%203/Solution/task_3. Figure 16 shows the different architecture based on hidden units' parameter. As per the lecture notes, to build the neural network it requires the following steps:

- Neural Network Structure:

  We have built a 3-layer neural network consisting input layer, hidden layer, and output layer.

  - Input layer: Two inputs A, B, and a bias. Thus, it has a size of 3.
  - Hidden layer: Variations in the number of neurons in the layer.
  - Output layer: It has a size of 1 as we will generate a single number at every input.

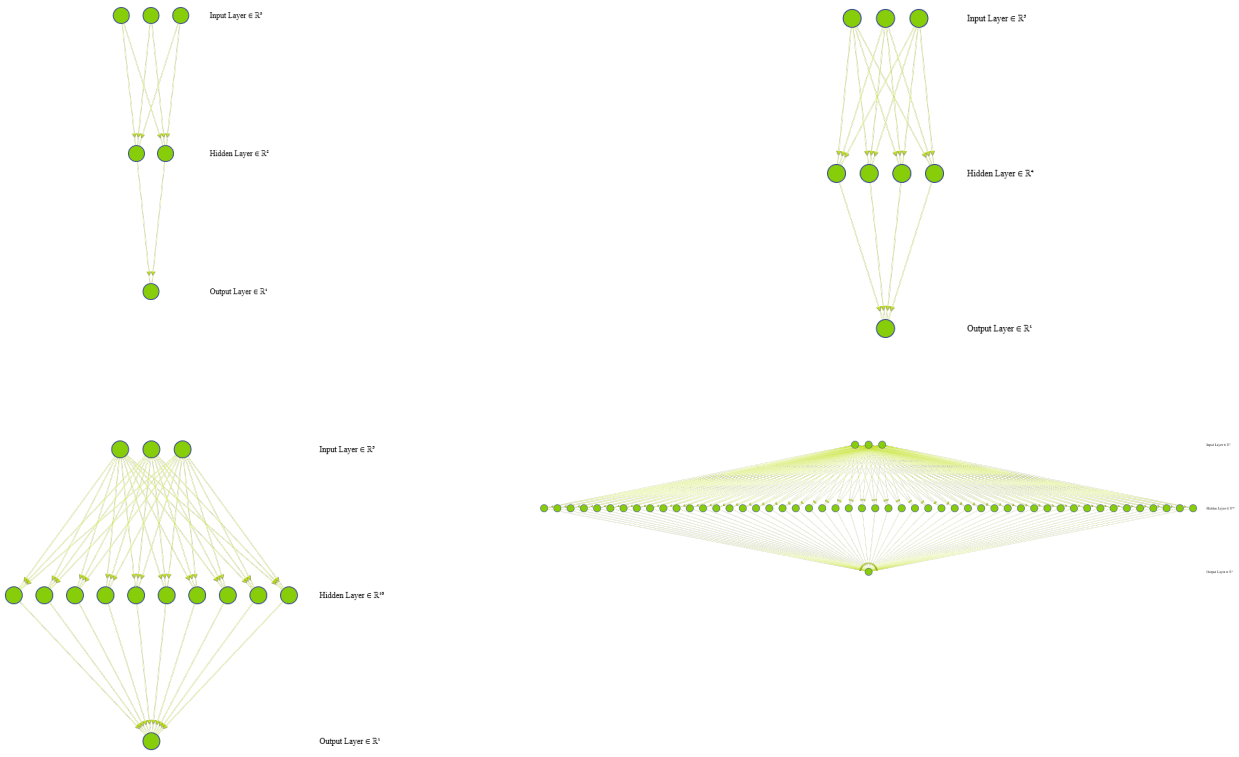


Figure 16. Different architecture based on hidden unit parameter - left top: 2, right top: 4, left bottom: 10, and right bottom: 50. Best viewed in color and magnification. (Source - http://alexlenail.me/NN-SVG/index.html)

In our model and model's implementation, there are 2 weight parameters available. One for hidden units and one for output calculation. Here, the size of weight is completely dependent on input data points/features and the number of hidden units/neurons in the hidden layer. Initially, we start with a random initialization strategy. Figure 17 shows the implementation of random weight initialization.

```
self.W_1 = np.random.rand(self.input.shape[1],h_size)

self.W_2 = np.random.rand(h_size,1)
```

Figure 17. Random weight initialization implementation in the codebase.

As per the task requirement, we have implemented *forward()* and *backward()* functions for the 3-layer neural network. First, we implement *forward()* function. We perform forward propagation in-network by multiplying the weights with the input data points/features at both the layers and we apply non-linear activation function (in our case Sigmoid). The forward function takes one parameter, and which is input data points/features x and performs the following operation as shown in Figure 18 and Figure 19.

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix} \qquad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}$$

$$Y = XW$$

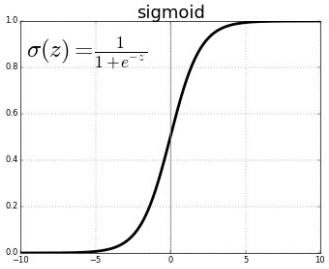
$$= \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix}$$

Figure 18. Forward propagation without activation function applied. (Source - Reading Material).

$$\hat{y} = \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1\mathbf{x})))$$

Figure 19. Forward propagation after applying the sigmoid activation function. (Source - Reading Material).

Sigmoid is a mathematical function [5] having a characteristic of the “S” - shaped curve. It is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point. Figure 20 shows the “S” - shaped curve and its implementation in code-base including forward propagation.




```
def sigmoid(x):
    return 1.0/(1+ np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1.0 - sigmoid(x))
```

```
def forward(self):
    # TODO:
    # implement the forward function that takes through each layer and
    # the corresponding activation function, this will generate the
    # output that should be stored in self.output
    Z_1 = np.dot(self.input, self.W_1)
    A_1 = sigmoid(Z_1)
    Z_2 = np.dot(A_1, self.W_2)
    self.output = sigmoid(Z_2)
    return np.dot((self.y - self.output).T, (self.y - self.output))
```

Figure 20. Left - Sigmoid function and “S”-shape curve. Top Right - Sigmoid implementation in the codebase. Bottom Right – Forward propagation implementation in the codebase. (Source - Google Image Search).

We have used squared error as the loss function which is defined as below in Figure 21.

$$L = \sum_{i}^{n} ||y - \hat{y}||^2$$

Figure 21. Squared error as the loss function implemented in our codebase. (Source - Reading Material)

After implementing *forward()* function, we now implement *backward()* function. We perform backward propagation to update weight parameters using gradient descent. We use the chain rule to compute the derivative of a squared error loss function. Figure 22 shows the compute of derivative and implementation of backward propagation.

$$\boxed{\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}}$$

```
def backward(self):
    # TODO:
    # apply the chain rule to find derivative of the loss function
    # with respect to W2 and W1

    example, l_r = self.input.shape[0], 0.01

    dZ_2 = self.y - self.output
    d_W2 = ((np.dot(sigmoid(np.dot(self.input,self.W_1)).T,dZ_2))/example)*(l_r)

    dZ_1 = np.multiply(np.dot(self.W_2,dZ_2.T).T,sigmoid_derivative(np.dot(self.input,self.W_1)))
    d_W1 = (np.dot(self.input.T,dZ_1)/example)*(l_r)#chain rule.

    self.W_2 +=  d_W2
    self.W_1 +=  d_W1
```

Figure 22. Top: Compute of derivative. Bottom: Implementation of *backward()* function in codebase.

We plot the graph of loss vs epoch for different numbers of neurons in Figure 23. We can observe that the different number of neurons in hidden layer leads to uncertainty in time to converge to the lowest loss value. Here the learning rate has been set to 0.001.

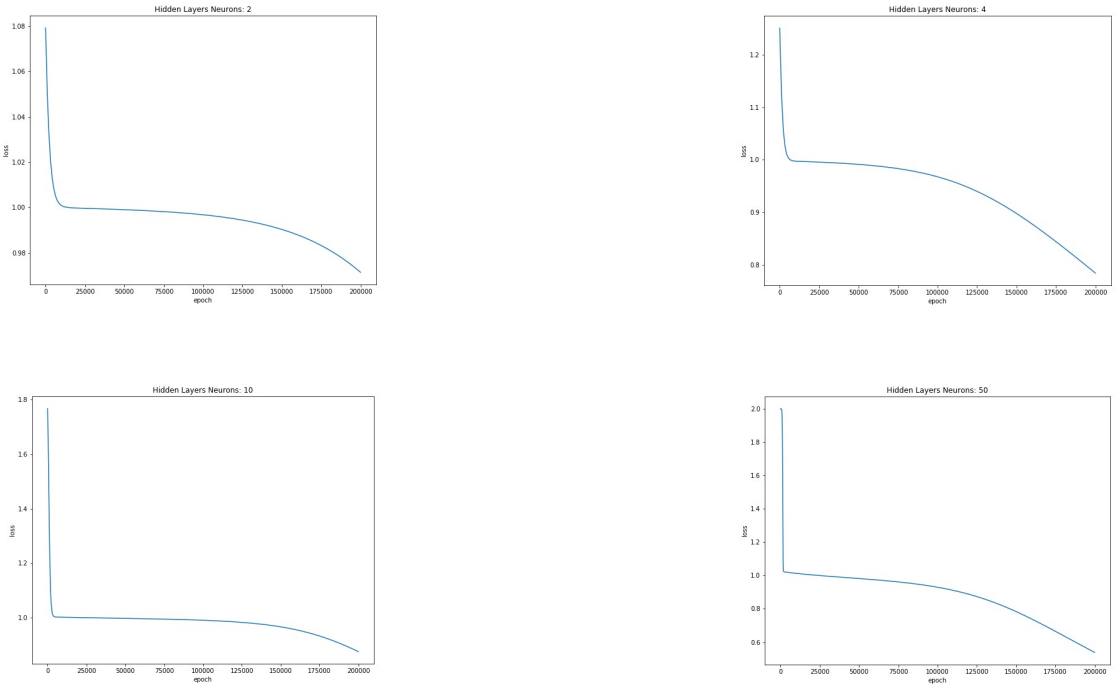


Figure 23. Loss vs Epoch graph for the different number of hidden neurons in the hidden layer. Top left: hidden neurons 2, Top right: hidden neurons 4, Bottom left: hidden neurons 10, and Bottom right: hidden neurons 50. Best viewed in color and magnification.

As from the different plot, we notice that as the number of hidden neurons increases the model try to overfit on the solution. Even the loss decreases correspondingly with less number of hidden neurons, thus the convergence of the model becomes faster. However, as we increase the number of hidden neurons in the hidden layer, it also increases the computing time. To claim this hypothesis we perform experiments based on the different numbers of hidden neurons vs iteration and states the loss for the combinations in Table 3 and Table 4.

Table 3. Loss vs Iterations vs Different number of hidden neurons.

	Loss			
Iterations	Neurons: 2	Neurons: 4	Neurons: 10	Neurons: 50
1	1.16655889	1.21904514	1.8211097	1.99999567
1000	1.12627743	1.27977869	1.67908833	1.97752716
10000	1.0009274	0.99939504	1.00549549	1.00943843
50000	1.00010251	0.99064128	0.9907075	0.97172736
100000	0.99963771	0.98437013	0.98436562	0.93758611
150000	0.99659896	0.99774613	0.94902522	0.63104036
200000	0.99967645	0.9197607	0.66349638	0.4832511

Table 4. Computing time for 200000 iterations vs different number of hidden neurons.

Computing time for 200000 iterations	
Different number of hidden neurons	Computing time (in seconds)
2	7.76
4	8.37
10	8.57
50	9.9

After performing several experiments, we can conclude that for successful training, we at least need 1 neuron in a hidden layer. However, when there are too many neurons in the hidden layer the following effects take place.

- Computing increases
- Increase in training time
- Faster convergence of model on training data
- Learning of model from fitting to overfitting

There have been constructive comments on the computer vision community on the selection of hidden units in the hidden layer. One of the notable feedbacks from Heaton *et al.* [6] is as follows:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Computer Vision (7315)

Task 4

In this part, we are focusing on detecting fake banknotes using Keras [7] - a high-level machine learning library. Instead of directly using images, in our experiment, we have been provided with the features which have already been extracted by the forensic department. The class labels are 0 (fake) and 1 (real) is provided with the data. We have to design a network for this binary classification task. First, we discuss the implementation details and attempt the questions in sequential order with in-depth discussion.

Implementation details

In our experiment, we are performing the training on a workstation with the Intel i-7 core processor and accelerated by NVIDIA GTX 1060 6 GB GPU. All the experiments run in Keras 2.1.3 using the TensorFlow backend. The training and testing dataset split are 80% and 20% respectively. Data normalization provided by using min-max normalization. We are using Binary Cross-Entropy loss function as the problem is a binary classification problem. The network design consists of an input layer, a single hidden layer, and the final output layer. The activation function is Sigmoid. The learning rate for the base model is set to 0.01. We iterate for 500 epochs for initial runs. Further, the experiments will be performed by many variants amongst the characteristics described above. The source code of the task-4 implementation is publicly available at *https://github.com/Vanditg/COMP-SCI-7315---Computer-Vision/tree/master/Assignment%20-%203/Solution/task_4*.

Answer (a)

First of all, we will tune the number of neurons in the single hidden layer with huge variation and plot the graphs. Furthermore, we will add one more hidden layer to check the network performance on the test set and plot the graphs. Finally, we will discuss the performance between these two approaches.

- Change in the number of neurons in a single hidden layer: We change the number of neurons to 5, 10, 20, 50, and 100. In Figure 24, we plot the loss vs epoch and accuracy vs epoch graph for the different configuration. Also in Table 5, we summarize the result with different number of hidden units, loss at the last iteration, and accuracy at that point.

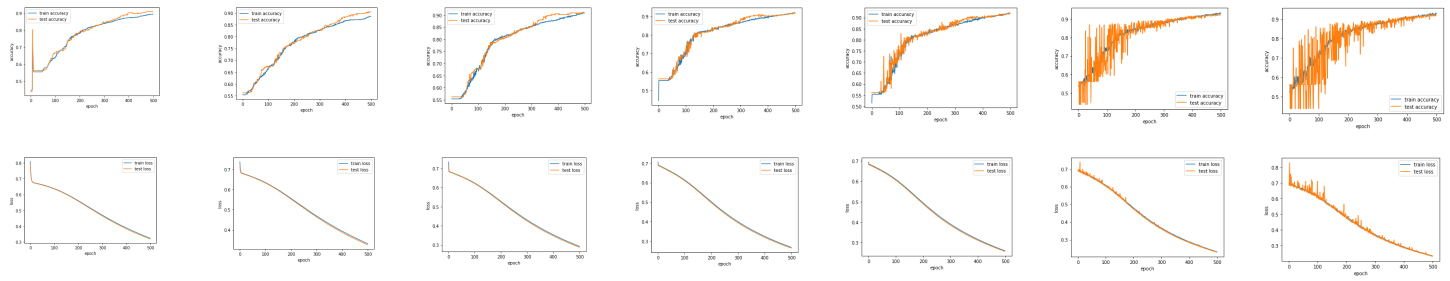


Figure 24. Top left - right: accuracy vs epoch for different number of neurons in hidden layer 5, 10, 20, 50, 100, 500, and 1000. Bottom left - right: loss vs epoch for different number of neurons in hidden layer 5, 10, 20, 50, 100, 500, and 1000. Best viewed in color and magnification.

Table 5. Summary of different number of hidden units, loss, and accuracy at the last iteration.

Hidden neurons	Loss	Accuracy
5	0.3181	90.88%
10	0.3266	90.51%
20	0.2881	90.88%
50	0.2647	91.97%
100	0.2571	92.34%
500	0.2297	92.70%
1000	0.2304	92.34%

- After performing experiments by changing the hidden units, we now explore the idea of adding one more hidden layer to our network. In Figure 25, we plot the loss vs epoch and accuracy vs epoch graph for the different configuration. Also in Table 6, we summarize the result with the different numbers of hidden units in the second layer, loss at the last iteration, and accuracy at that point.

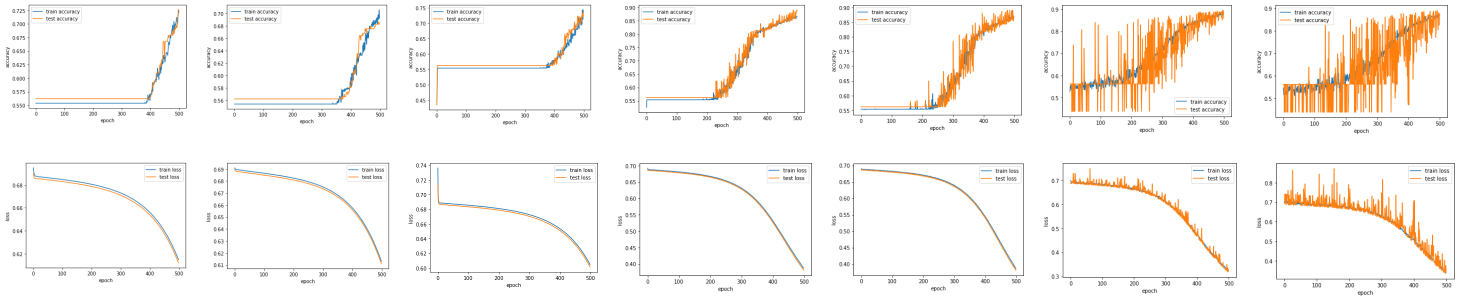


Figure 25. Top left - right: accuracy vs epoch for different number of neurons in 2nd hidden layer 5, 10, 20, 50, 100, 500, and 1000. Bottom left - right: loss vs epoch for different number of neurons in 2nd hidden layer 5, 10, 20, 50, 100, 500, and 1000. Best viewed in color and magnification.

Table 6. Summary of the different number of hidden units in 2nd hidden layer, loss, and accuracy at the last iteration.

Hidden neurons	Loss	Accuracy
5	0.6120	72.63%
10	0.6111	68.25%
20	0.6013	72.26%
50	0.3806	88.69%
100	0.3814	86.86%
500	0.3304	85.77%
1000	0.3476	87.59%

From the experiments, we can conclude that going deeper will make the neural network more expressive. It means that the network can capture variations of the data better. Further, this is known to yield expressiveness more efficiently than increasing the width of the hidden layers. However, the tradeoff for more expressiveness increases the tendency to overfit the training data. So, the network could be as deep as the training data allows, however, we can only determine a suitable depth by performing experiments. In general, by increasing the depth it will also lead to more computing time. From, the experiment we can see that increasing the depth leads to poor results as well compare to increasing the width in neural networks.

Answer (b)

After performing experiments on the width and depth of neural networks, we will focus on the activation function parameter. Instead of using Sigmoid, we will change it with the ReLU. Sigmoid activation function has a very sensitive issue which is when sigmoidal units saturate across most of their domain - they saturate to a high value when z is very positive, saturate to a low when z is very negative, and are only strongly sensitive to their input when z is near 0 [8]. The function in the neural network must provide more sensitivity to the activation sum input and avoid easy saturation, and a node or unit that implements are referred to Rectified Linear Activation Unit (ReLU). Figure 26 shows the characteristics of the ReLU function. A sigmoid function will transform an input value into an output between 0 and 1. Unlike sigmoid, ReLU is called a piecewise function, because half of the output is linear while another half is nonlinear. The ReLU function has also much less computation cost than sigmoid. Thus, ReLU learns faster than sigmoid. In Figure 27, we plot the loss vs epoch and accuracy vs epoch graph to showcase the convergence by changing the hidden units in a single layer. In Table 7, we summarize the number of hidden units, loss, accuracy, and number of epochs to convergence.

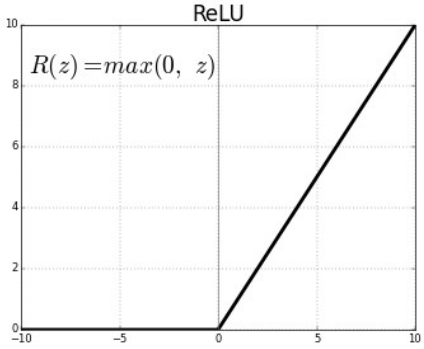


Figure 26. ReLU function. The output of ReLU is the maximum value between zero and the input value based on the equation. Best viewed in color and magnification. (Source – Google Image Search)

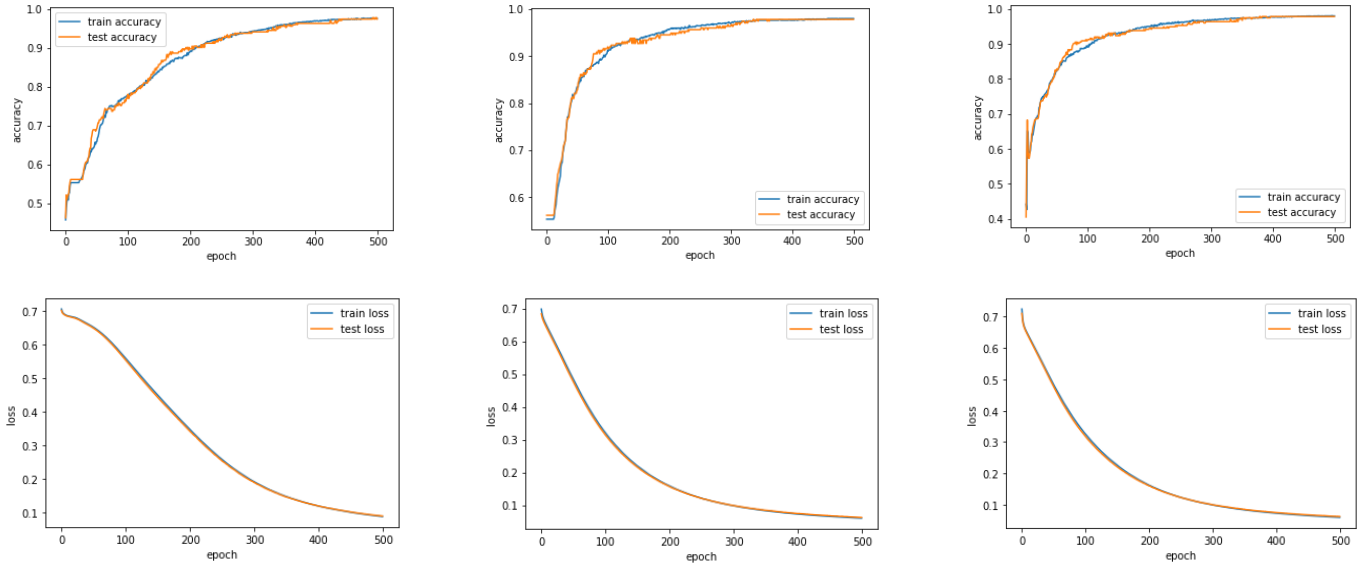


Figure 27. Top left - right: accuracy vs epoch for the different number of neurons followed by ReLU activation function in hidden layers 5, 10, and 20. Bottom left - right: loss vs epoch for the different number of neurons followed by ReLU activation function in hidden layers 5, 10, and 20. Best viewed in color and magnification.

Table 7. Summary of the number of hidden units, loss, accuracy, and number of epochs to convergence using the ReLU activation function.

Hidden Units	Loss	Accuracy	Number of Epoch Require for Convergence
5	0.0902	97.45%	480-500
10	0.0625	97.81%	370-400
50	0.0622	97.81%	410-440

Answer (c)

After performing experiments on ReLU, we will focus on one of the important parameters of neural networks, a learning rate. The amount that the weights are updated during training is referred to as the step size or the learning rate. In general, the learning rate is a hyperparameter used in the training of a neural network that has a small positive value often in the range between 0.0 and 1.0. The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs/iterations given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck. In Figure 28, we show the effect of the learning rate. Lecun *et al.* [8] noted that the learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate. In Figure 29, we showcase the loss vs epoch and accuracy vs epoch graph, where we set the hidden unit to 5, activation function as ReLU, and changes the learning rate to 0.0001, 0.001, 0.01, 0.05, and 0.01. In Table 8, we summarize the loss, accuracy, and learning rate while fixing the number of hidden units.

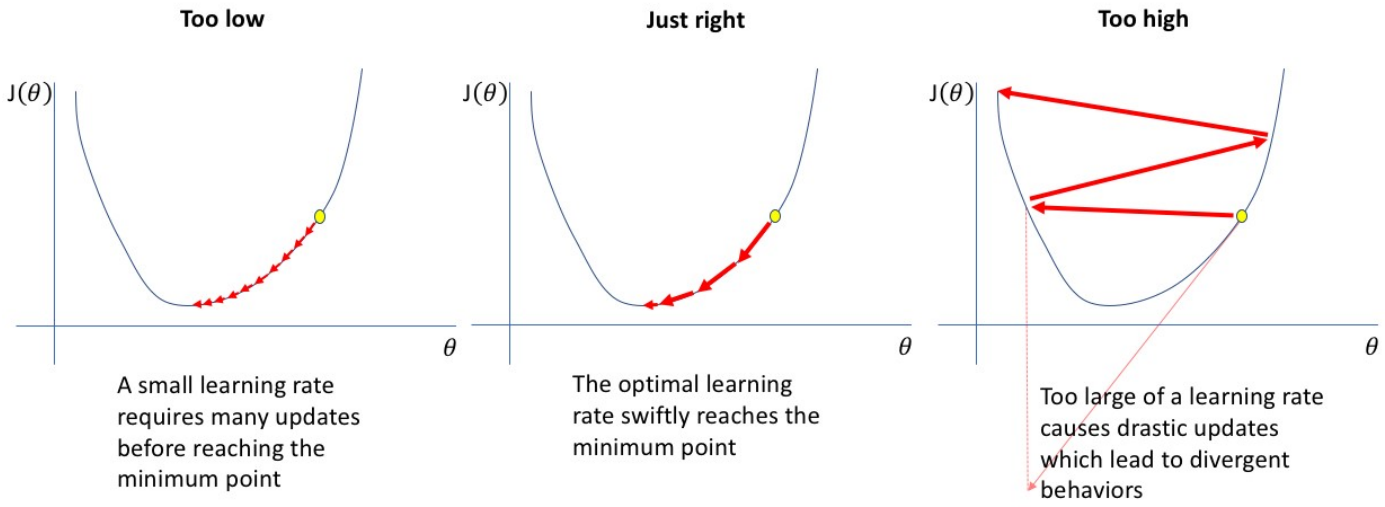


Figure 28. Effect of different learning rate too low, too high, and just right. Best viewed in color and magnification. (Source - Google Image Search).

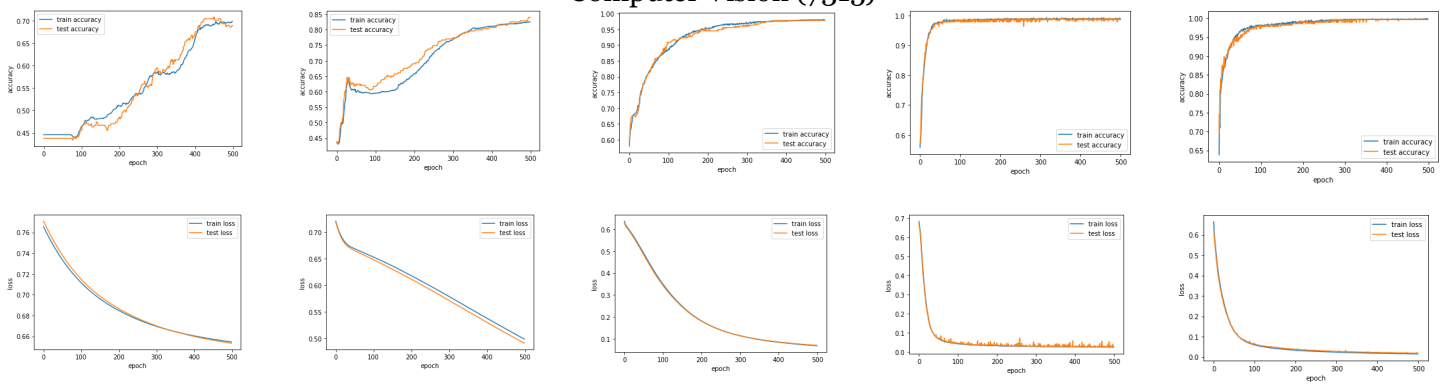


Figure 29. Top left - right: accuracy vs epoch for different learning rates 0.0001, 0.001, 0.01, 0.1, and 0.005. Bottom left - right: loss vs epoch for different learning rates 0.0001, 0.001, 0.01, 0.1, and 0.005. Best viewed in color and magnification.

Table 8. Summary of the loss and accuracy while changing the learning rate.

Hidden Units	Loss	Accuracy	Learning rate
5	0.6533	68.98%	0.0001
5	0.4911	83.94%	0.001
5	0.0695	97.81%	0.01
5	0.0185	99.64%	0.05
5	0.0246	98.54%	0.1

From the experiments and results demonstrated here, we can conclude that the learning rate is one of the important parameters which leads to significant changes in the performance and convergence of the model. When we increase the learning rate the model converges faster. To achieve faster convergence, prevent oscillations, and getting stuck in undesirable local minima the learning rate is often varied during training either by an adaptive learning rate.

Answer (d)

After performing experiments on the learning rate, we will now focus on the number of epochs required in the training of a neural network. In general, the number of epochs is not that significant, in fact, more important is the validation and training error, and the convergence of the model. As long as it keeps dropping training should need to continue. For example, if the validation error starts increasing that might be an indication of overfitting. One should set the number of epochs as high as possible and terminate training based on the error rates. To be more specific on this hypothesis, an epoch is one learning cycle where the model sees the whole training data set. So we claim this hypothesis from the experiments that the only way to know the optimal number of epochs by plotting the accuracy vs epoch and loss vs epoch curve. In Figure 30, we showcase the accuracy vs epoch and loss vs epoch graphs, and in Table 9, we summarize the loss, accuracy, change of learning rate, additional hidden layers, and the number of epochs for convergence. In all the experiments we use ReLU as activation function and we set epochs to 300 and 1000.

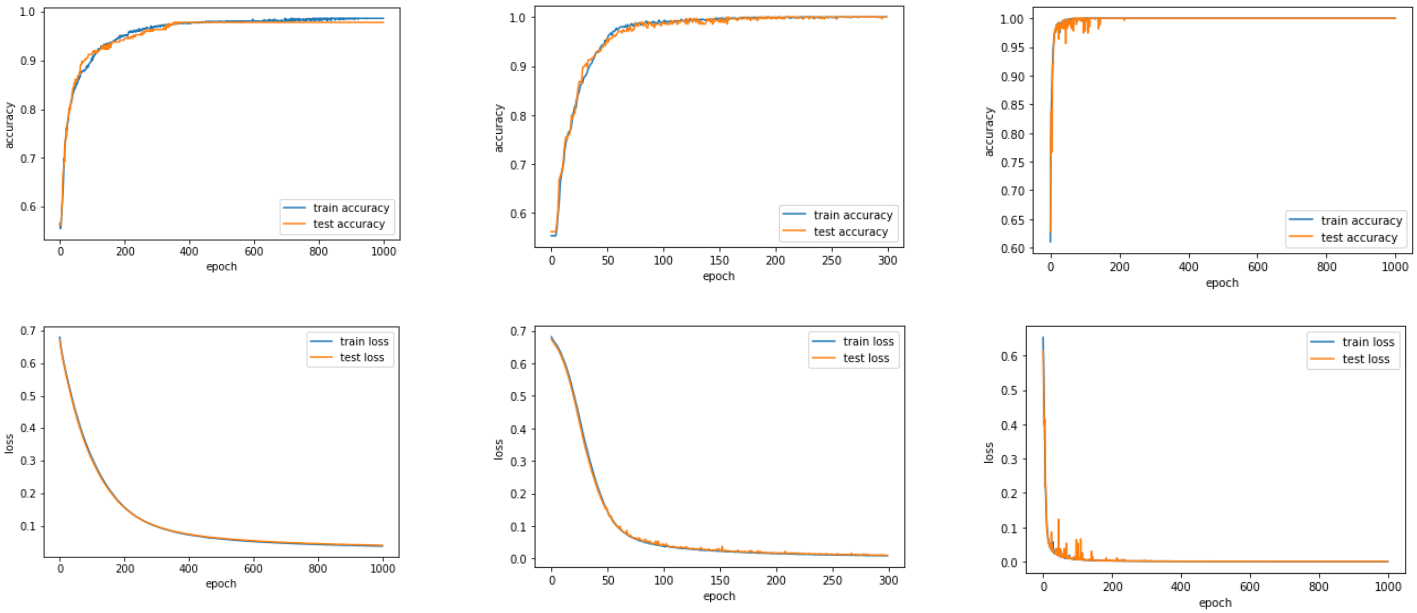


Figure 30. Top left - right: accuracy vs epoch for different learning rates 0.1, 0.1, and 0.01. Epoch: 1000, 300, and 1000. Hidden layer: One, Two, and Three. Bottom left - right: loss vs epoch for different learning rates 0.1, 0.1, and 0.01. Epoch: 1000, 300, and 1000. Hidden layer: One, Two, and Three. Best viewed in color and magnification.

Table 9. Summary of the loss, accuracy, number of epochs for convergence while changing the learning rate and adding hidden layers.

Hidden Units	Loss	Accuracy	Learning Rate	Hidden Layers	Number of Epochs for Convergence/Total Number of Epochs
50	0.0397	97.81%	0.1	One	350-400/1000
50	0.0003	100% (Overfit)	0.1	Two	70-90/300
50	0.0095	100% (Overfit)	0.01	Three	170-200/1000

From the experiments and summary of the result, we can conclude that the number of epochs does not matter in the case but what matters is the training error and convergence of the model. For instance, one of the configuration models converges even faster than the total number of epochs, thus leading to faster convergence and more training error after a certain point. Thus by looking at the training error and training-testing loss of the model we can configure the required number of epochs for the same. Finally, the best model has the following configurations and achieves good results without overfitting. We plot the loss vs epoch and accuracy vs epoch graph in Figure 31.

- Hidden Unit: 200
- Learning Rate: 0.01
- Number of Epochs: 400
- Activation Function: ReLU
- Loss: 0.0239
- Hidden Layers: Two Hidden Layers
- Accuracy: 99.64%

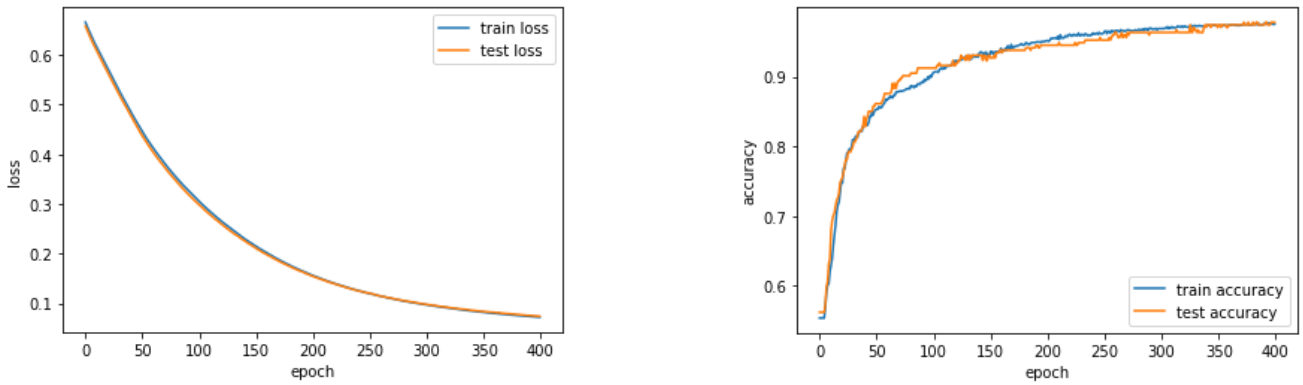


Figure 31. Left: Loss vs Epoch for the best model configuration. Right: Accuracy vs Epoch for the best model configuration. Best viewed in color and magnification.

References:

[1] Szeliski, Richard. Computer vision: algorithms and applications. Springer Science & Business Media, 2010.

[2] Stephen, I. "Perceptron-based learning algorithms." IEEE Transactions on neural networks 50.2 (1990): 179.

[3] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

[4] Rifkin, Ryan, and Aldebaro Klautau. "In defense of one-vs-all classification." Journal of machine learning research 5.Jan (2004): 101-141.

[5] Han, Jun, and Claudio Moraga. "The influence of the sigmoid function parameters on the speed of backpropagation learning." International Workshop on Artificial Neural Networks. Springer, Berlin, Heidelberg, 1995.

[6] Heaton, Jeff. "The number of hidden layers." Heaton Research Inc (2008).

[7] Chollet, François. "Keras: The python deep learning library." Astrophysics Source Code Library (2018).

[8] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." nature 521.7553 (2015): 436-444.