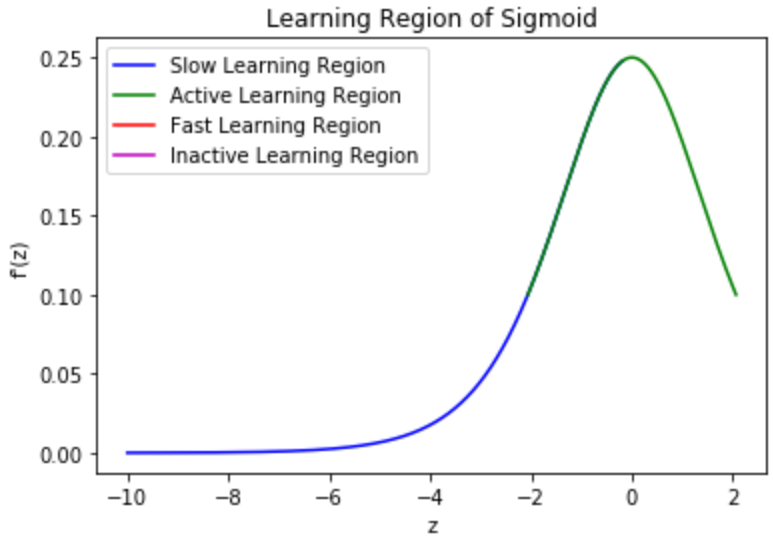
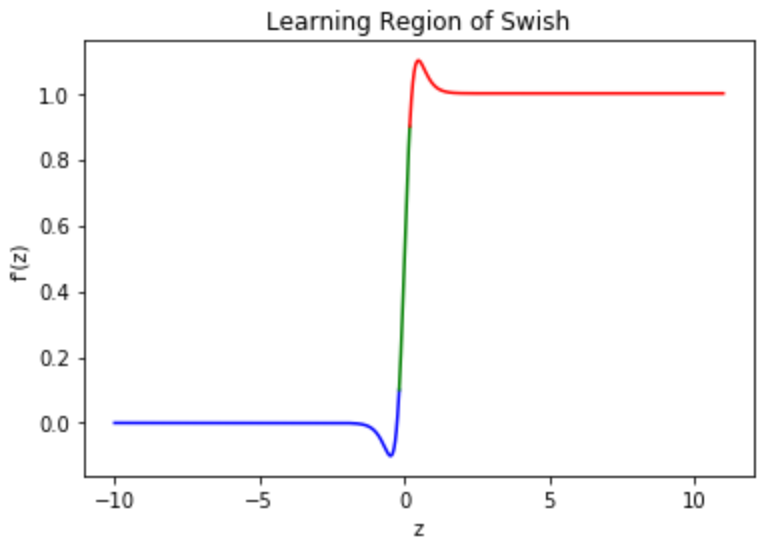
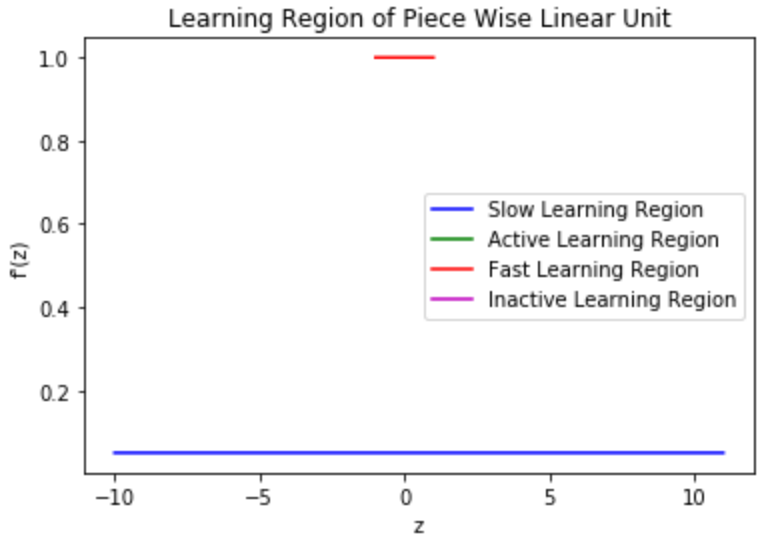
Problem 1:

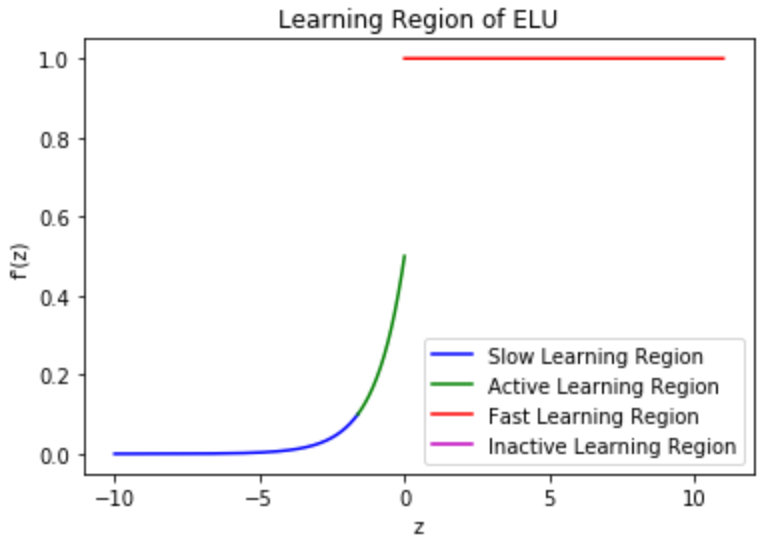
In this problem we plot the Gradient Descent of some activation functions that are commonly used in Deep Learning. According the question, the domain of the activation function is divided into 4 types of regions.

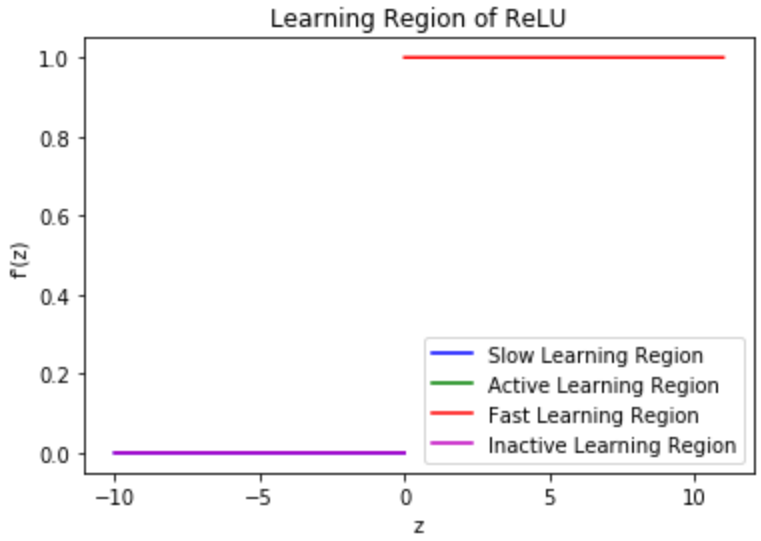
Activation	**f(z)**	**f'(z)**	**Number of Regions**
Rectified Linear Unit	f(z) = z { if z >= 0 0 otherwise }	f'(z) = 1 { if z>=0 0 otherwise }	2; Inactive and Fast
Sigmoid	f(z) = 1/(1 + exp(-z))	f'(z) = f(z)(1 - f(z))	2; Slow and Active
Piecewise Linear Unit	f(z) = 0.05z + 0.95 if z > 1 = z if |z| <= 1 = 0.05z - 0.95 if z < 1	f'(z) = 0.05 if |z| > 1 = 1 if |z| <= 1	2; Fast and Slow
Swish	f(z) = (z) sig(5z)	f'(z) = 5(z)exp(-5z)/(sig^2(5z) + sig(5z))	3; Fast, Active and Slow
ELU	f(z) = z if z >= 0, 0.5(e^z - 1) otherwise	f'(z) = 1 if z>= 0 0.5e^x	3; Active, Slow and Fast




Learning Region of Swish
f'(z)
z
1.0
0.8
0.6
0.4
0.2
0.0
−10
−5
0
5
10


Learning Region of Piece Wise Linear Unit
f'(z)
z
Slow Learning Region
Active Learning Region
Fast Learning Region
Inactive Learning Region
1.0
0.8
0.6
0.4
0.2
−10
−5
0
5
10





Problem 2

There are two main computations that take place in neural networks. These are forward propagation and backward propagation. The algorithm forward propagation is given here:

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

In this algorithm, a sample X is considered where the feature vector is n_i. We consider our neural network as a graph and each node of the graph is a neuron which is considered as the node. The k^{th} node is numbered as u_k here. Now, this numbering starts from the input layer of the neural network and goes on till all neurons of that layer are given a particular u_k, after which the numbering method moves to the next layer, and so on. In this way, if there are n_i input features in the neural network, we compute u_{ni} nodes corresponding to the first n_i neurons of the network and the output of u_k is given as $u^{(k)}$.

In the first loop we take all the nodes of the first layer as the input layer of the graph. In this way again, u_1 gets the value of X_1, u_2 gets the value X_2, and so on.

The next for loop works with the neurons in the next layers. Now, all neurons in a network, apart from the input layer, would do some computation on the inputs they get from the parent node and produce an output which is then passed to their child nodes. This is done inside this loop. For any random neuron u_i in any random layer, the neuron first takes the output of its parent nodes $u^{(j)}$ as arguments $A^{(i)}$ for computing a function $f^{(i)}$ (given to it as per architecture), to produce output $u^{(i)}$. This process continues till all neurons have finished their computation. The output $u^{(n)}$ of the last neuron u_n is given as output.

Algorithm 2

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

The second major part of the computation which is done in a neural network is the backpropagation. We need to calculate the derivatives of a particular neuron's output with respect to its previous layers for backpropagation.

We initialize a *grad_table* data structure, in order to lessen the load of computation. The *grad_table* will hold all the partial derivatives of a neuron. Then we go from the last neuron to the first, traversing in the opposite direction to the forward-propagation method. Here, for the last neuron, we see that the partial derivative of the last neuron to itself is always equal to 1. In this way, we initialize the last entry of the *grad_table* as 1. Then again, inside the loop, we calculate the partial derivative of all the nodes, in a backwards way. For each node, we get the partial derivative of that node w.r.t its parent. Now, for a particular child-parent neuron pair, their computation graph may have more than one path between them. This is the reason why taking

into consideration all the paths that the parent has with the child and use chain rule in traversing through all of them. Then, for optimization, we put inside the partial derivative of a node, w.r.t its children. This is possible since we are computing backwards, and hence the partial derivative of a node w.r.t its children will be computed before it is stored in the table. Hence, the values of all the derivatives which are required for the current computation should already have been computed.

Part 2: In case of forward propagation, the network needs to compute the output of all the parent nodes before going to the child, thus it has to traverse through every edge once. Similarly, backpropagation calculates the partial derivative of 2 nodes by visiting the edge between them, and since it only calculates the partial derivative of any 2 nodes (which are connected) once, thus, the total number of computations it requires is of the order of O(E) as well.
Both the algorithms take the same order of time since the computational complexity of both are of the order of O(E) where E is the number of edges of the computational graph respectively.

Problem 3

In this problem, we train a neural network to approximate the function
$g(x) = 1 + \sin(3\pi x/2), x \in [-1,1]$.

We generated 100 samples. These samples were used for both training as well as validation or testing. We trained the network using these 100 samples, after which we used the same model to predict the testing samples. The error between the predicted y and actual y value is taken, and their absolute difference is calculated.
Finally, the maximum difference between the 2 values are printed. The networks's architecture is given below:

Loss: Mean Squared Error
Optimizer: Adam (learning_rate = 0.003)
Epochs: 10000

Metrics: Mean Absolute Error

Type	No. Of Neurons	Activation Function
Dense	7	tanh
Dense	1	Linear

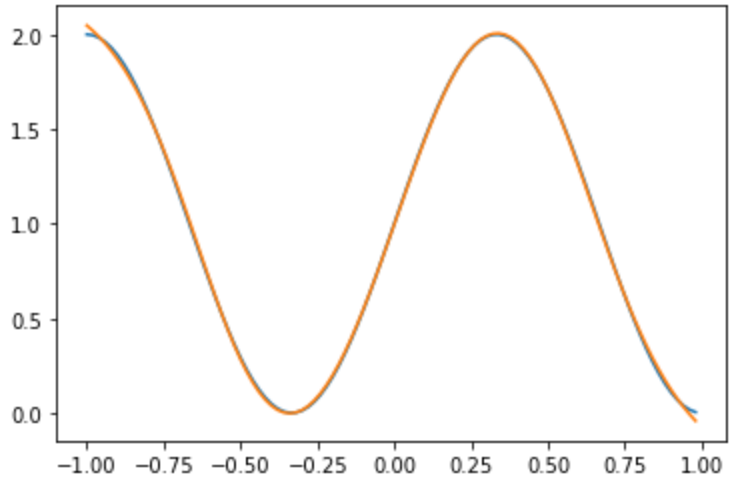


Figure: The blue and brown graph have almost converged.
Max Absolute Error was ~ 0.04 (< 0.05)

Problem 4

Part 1: The architecture of the softmax layer is-

- 20 neurons since the size of the bias array is 20
- The sample vector size will be 2000/ 20 = 100. Hence, there is 100 input features since the sample vector is of size 100
- Total number of connections is 100 X 20 = 2000

Part 2: The given sample is predicted and classified as class 4

Part 3: Using cross-entropy loss, keeping learning rate as 0.1, and considering the correct label of the sample is Class 1, we perform gradient descent on the model. The resulting changes in weights and biases are as follows-

Parameters	# Increased	# Decreased	# Unchanged
Weights	10	0	1990
Biases	1	0	19