**Versions:**

Tensorflow==2.7

Pandas==1.4.4

Numpy==1.23.5

Keras==0.20.0

Sklearn==1.0.2

Csv==1.0

matplotlib==3.5.2

**Functions:**

**Define parameters For NN Model:**

**input\_dim = X\_train\_vectors.shape[1]** #sets the variable input\_dim equal to the number of features in the training data.'X\_train\_vectors.shape[1]'' retrieves the number of columns (i.e., features) in the feature matrix X\_train\_vectors. This value represents the number of input dimensions or features for the classification model.

**output\_dim = len(np.unique(y\_train))** #sets the variable output\_dim equal to the number of unique target labels in the training data. These variables are likely used to define the input and output dimensions of the neural network or classification model being built.np.unique(y\_train) returns the unique values of the target variable y\_train. The length of this array is equal to the number of distinct target labels in the dataset.len(np.unique(y\_train)) returns the number of unique target labels in the dataset. This value represents the number of output dimensions or classes for the classification model.

# Reshape the data for LSTM input

**X\_train\_lstm = np.reshape(X\_train\_vectors.toarray(), (X\_train\_vectors.shape[0], 1, X\_train\_vectors.shape[1]))** #X\_train\_vectors is the sparse matrix of shape (number of samples, number of features) that contains the training feature vectors. The toarray() method is used to convert this sparse matrix into a dense matrix.The np.reshape() function is then used to reshape the dense matrix into a 3-dimensional array with shape (number of samples, 1, number of features). The first dimension represents the number of samples, the second dimension represents the number of time steps in the LSTM (which is set to 1 in this case), and the third dimension represents the number of features in each input vector.The reason for reshaping the input data in this way is because LSTM models require 3-dimensional input data, where the first dimension is the number of samples, the second dimension is the number of time steps, and the third dimension is the number of features.this line of code reshapes the training data from a dense matrix of shape (number of samples, number of features) to a 3-dimensional array of shape (number of samples, 1, number of features) that is suitable for training an LSTM model.

**X\_test\_lstm = np.reshape(X\_test\_vectors.toarray(), (X\_test\_vectors.shape[0], 1, X\_test\_vectors.shape[1]))** #X\_test\_vectors is the sparse matrix of shape (number of samples, number of features) that contains the test feature vectors. The toarray() method is used to convert this sparse matrix into a dense matrix.The np.reshape() function is then used to reshape the dense matrix into a 3-dimensional array with shape.The reason for reshaping the test data in this way is because the LSTM model expects input data with the same shape as the training data.this line of code reshapes the test data from a dense matrix of shape (number of samples, number of features) to a 3-dimensional array of shape (number of samples, 1, number of features) that is suitable for testing the LSTM model.

**epochs = 1** #sets the number of training epochs to 10 and assigns this value to epochs. One epoch is a complete pass through the training data.

**batch\_size = 32** #sets the batch size to 32, which is the number of samples that will be processed in each training iteration.

**gamma = 0.99** # sets the value of gamma to 0.99, which is a discount factor used in the calculation of the discounted cumulative reward in some reinforcement learning algorithms.

**learning\_rate = 0.001** #sets the learning rate to 0.001, which is the step size used in the optimization algorithm during training.

**Class to build the NN model:**

**Name:** Neural Network model

**Purpose:** to define and create a neural network or classification model for a given dataset.

**Invariants:**

* The Input function defines the input layer of the neural network with shape=(input\_dim,), where input\_dim is the number of features in the training data.
  + Two Dense functions are used to define the hidden layers of the neural network. The first hidden layer has 64 neurons and relu activation function, while the second hidden layer has output\_dim neurons and softmax activation function. The output\_dim variable represents the number of unique target labels in the training data.
  + The Model function is used to create an instance of the neural network or classification model with inputs=inputs and outputs=x. The model takes the input layer and hidden layers as input and outputs the predicted target labels.
  + Finally, the Adam optimizer with a learning rate of 0.001 is set to update the weights of the neural network or classification model during training**.**

**CODE:**

**inputs = Input(shape=(input\_dim,))** #defines an input layer for the neural network or classification model with the specified number of features or input dimensions.The Input function takes shape=(input\_dim,) as an argument, where input\_dim is an integer representing the number of features in the training data. The shape parameter is a tuple that specifies the input shape of the layer.This line of code is important as it initializes the neural network or classification model and sets the input shape for the subsequent layers.

**x = Dense(128, activation='tanh')(inputs)** #defines a hidden layer in the neural network or classification model.The Dense function creates a fully connected layer with 64 neurons and relu activation function. The activation parameter specifies the activation function used to introduce non-linearity into the model.The (inputs) at the end of the line specifies that the input to this layer is the inputs layer that was previously defined.This line of code is important because it adds a layer of computation to the neural network or classification model, which helps it learn complex representations of the input data. The relu activation function is commonly used in deep learning models and helps in speeding up the training process by preventing vanishing gradients.

**x = Dense(output\_dim, activation='softmax')(x)** #defines the output layer of the neural network or classification model.The Dense function creates a fully connected layer with output\_dim neurons and softmax activation function. The output\_dim parameter specifies the number of unique target labels in the training data. The softmax activation function converts the output of the layer into a probability distribution over the target labels, where the highest probability is assigned to the predicted target label.The (x) at the end of the line specifies that the input to this layer is the output of the previous hidden layer x.This line of code is important because it is the final layer of the neural network or classification model, which produces the predicted target labels. The softmax activation function is commonly used for multi-class classification problems and ensures that the predicted probabilities sum up to 1.0 over all target labels.

**model = Model(inputs=inputs, outputs=x)** #creates an instance of the neural network or classification model with the specified input and output layers.The Model function takes inputs=inputs and outputs=x as arguments. The inputs parameter specifies the input layer of the model, which was previously defined using the Input function. The outputs parameter specifies the output layer of the model, which was defined using the Dense function.This line of code is important because it connects the input and output layers to create a neural network or classification model. The model is an instance of the Model class from Keras, which provides high-level APIs for building and training deep learning models. The model can be trained using various optimization algorithms and loss functions to minimize the difference between the predicted and actual target labels.

**optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)** #creates an instance of the Adam optimization algorithm for the neural network or classification model.The optimizers.Adam function is a popular optimization algorithm used for training deep neural networks. The learning\_rate parameter specifies the step size or the size of the update made to the model weights during each iteration of the optimization algorithm.This line of code is important because it initializes the optimizer used to update the weights of the neural network or classification model during training. The choice of optimizer can significantly affect the performance of the model, and Adam is a popular choice due to its fast convergence and adaptive learning rate.

#optimizer = tf.keras.optimizers.RMSprop(learning\_rate=0.001) #RMSprop optimizer

**model.compile(loss='categorical\_crossentropy', optimizer=optimizer, metrics=['accuracy'])** # is an important line of code in the process of building a deep learning model using Keras. It compiles the model by specifying the loss function, optimizer, and metrics to be used during training.In particular, the loss parameter specifies the loss function that the model will use to evaluate its performance on the training data. In this case, the 'categorical\_crossentropy' loss function is used, which is commonly used for multiclass classification problems.The optimizer parameter specifies the optimization algorithm that will be used to adjust the weights of the model during training in order to minimize the loss function. Here, the optimizer variable is passed in, which should be an instance of a pre-defined optimizer class from Keras, such as Adam or RMSprop.Finally, the metrics parameter specifies the evaluation metrics that will be used to monitor the model's performance during training and testing. In this case, 'accuracy' is the metric used, which is commonly used for classification problems.Overall, model.compile is a crucial step in the process of building and training a deep learning model, as it sets up the model for optimization by specifying the necessary components for the training process.

#model.compile(loss='mean\_squared\_error', optimizer=optimizer, metrics=['accuracy']) #loss:mean\_squared\_error

**Define REINFORCEMENT LEARNING Technique for NN:**

**NAME:** policy\_gradient

**PURPOSE:** To implement the policy gradient algorithm for training a neural network model (model) using given input data (x) and target labels (y) with a specified optimizer (optimizer). The algorithm is trained for a given number of epochs (epochs) and batch size (batch\_size), and uses a discount factor (gamma) for computing the model loss. The purpose of this code is to train a neural network model using the policy gradient algorithm, where the model's weights are updated based on the computed policy gradient and model loss, and the training process is performed in batches for efficiency.

**INVARIANTS:**

* Calculates the length of the input data (x) using getnnz() method.
* Iterates over each epoch.
* For each epoch, iterates over the input data (x) in batches of size batch\_size.
* For each batch, computes the logits (raw output) of the model for the input data using model(x\_batch).
* Computes the log probabilities of the logits using tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0)).
* Creates one-hot encoded labels from the target labels (y\_batch) using tf.one\_hot() method.
* Computes the policy gradient loss by multiplying the log probabilities with the one-hot encoded labels and taking the negative mean using tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1)).
* Computes the gradients of the policy gradient loss with respect to the model's trainable weights using tape.gradient().
* Applies the gradients to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* Computes the average reward for the current batch and uses it as a baseline.
* Computes the model loss by multiplying the policy gradient loss with the average reward and the discount factor (gamma).
* Computes the gradients of the model loss with respect to the model's trainable weights using tape.gradient().Applies the gradients to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* The purpose of this code is to train a neural network model using the policy gradient algorithm, where the model's weights are updated based on the computed policy gradient and model loss, and the training process is performed in batches for efficiency.

**CODE:**

**def policy\_gradient(x, y, model, optimizer, epochs, batch\_size, gamma):** #class to define the policy\_gradient RL technique

**length = x.getnnz()**  #Calculates the length of the input data (x) using getnnz() method.

for epoch in range(epochs): #Iterates over each epoch.

**epoch\_rewards = []** #empty lists that are likely intended to store the rewards for each epoch during the training of a neural network.

**epoch\_losses = []** #empty lists that are likely intended to store the losses for each epoch during the training of a neural network.

**for batch\_start in range(0, length, batch\_size)**: #It is used to iterate over the input data (x) in batches of size batch\_size during the training process. It starts from the beginning of the input data and increments in steps of batch\_size until it reaches the end of the data.

**batch\_end = min(batch\_start + batch\_size, length)** # It calculates the ending index (batch\_end) of the current batch during the iteration over the input data (x) in batches. It ensures that the ending index does not exceed the total length of the data (length) to avoid accessing data beyond the available range.

**x\_batch = x[batch\_start:batch\_end].toarray()** # Extracting a batch of data from the array 'x' using the start and end indices of the batch.toarray() converters the batch to an array, if it's in sparse format, for further processing

**y\_batch = y[batch\_start:batch\_end]** #Extracting a batch of labels from the array 'y' using the start and end indices of the batch

**with tf.GradientTape() as tape:** #tf.GradientTape() is a TensorFlow API that provides a mechanism for automatic differentiation, which is a key technique used in machine learning optimization algorithms, such as gradient descent. It allows you to compute gradients of a computation with respect to its input variables, which can then be used to update the values of those variables during optimization."tape" refers to a mechanism provided by TensorFlow that records operations for the purpose of computing gradients. The tape acts as a context within which computations are recorded, and these computations can later be used to compute gradients using the tape.gradient() method.

**logits = model(x\_batch)** #Passing the batch of input data 'x\_batch' through the model to obtain logits.Logits are the output of the model before applying any activation function, typically used for classification tasks Logits represent the raw, unnormalized scores for each class, which can be used for further processing or prediction.'model' is the trained model that takes 'x\_batch' as input and produces logits as output

**log\_probs = tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0))** # calculates the log probabilities by taking the natural logarithm (tf.math.log()) of the model's predicted logits (logits). The tf.clip\_by\_value() function is used to clip the logits to a specific range to avoid numerical instability. In this case, the minimum value is set to 1e-10 and the maximum value is set to 1.0. The resulting log probabilities are stored in the log\_probs variable.

**labels = tf.one\_hot(y\_batch, depth=output\_dim)** #Converting the batch of labels 'y\_batch' into one-hot encoding using 'tf.one\_hot' function One-hot encoding represents categorical labels as binary vectors with a single '1' and remaining '0's.'y\_batch' is the input tensor containing the batch of labels to be converted to one-hot encoding.'output\_dim' specifies the depth of the one-hot encoding, which should be equal to the number of classes in the classification task

**loss = -tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1))** #Compute the cross-entropy loss. labels: Ground truth labels. log\_probs: Log probabilities predicted by the model

**grads = tape.gradient(loss, model.trainable\_weights)** #grads typically refers to the computed gradients of the loss function with respect to the trainable weights of a machine learning model.The tape.gradient() function in TensorFlow is used to compute the gradients of a given function (in this case, the loss function) with respect to a list of variables (in this case, the model.trainable\_weights). These gradients can then be used in an optimization algorithm, such as gradient descent, to update the model weights and improve the model's performance during training.Compute the gradients of the loss with respect to the trainable weights of the model.loss: The computed loss value.model.trainable\_weights: List of trainable weights of the model

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))** #It applies the computed gradients (grads) to update the model weights (model.trainable\_weights) using an optimizer. The zip() function is used to create pairs of gradients and corresponding model weights, which are then passed to the apply\_gradients() method of the optimizer to perform the weight update step. This step is a key part of the optimization process in training machine learning models, as it helps to adjust the model weights based on the gradients of the loss function, with the goal of minimizing the loss and improving the model's performance.

**avg\_reward = np.mean((logits.numpy().argmax(axis=1) == y\_batch).astype(int))** #calculates the average reward, which is computed as the mean accuracy of the model's predicted logits compared to the ground truth labels (y\_batch). It mentions that logits represents the model's predicted logits, and y\_batch represents the ground truth labels. The result of the comparison between the predicted logits and ground truth labels is cast to an integer array using astype(int), and then the mean is calculated using np.mean(). The calculated average reward is stored in the avg\_reward variable.

**rewards = avg\_reward** #Rewards for the model

**print("The rewards are:",rewards)** #prints the rewards

epoch\_rewards.append(rewards) #appends the rewards obtained by the network during the current epoch to the epoch\_rewards list. This allows us to keep track of the rewards obtained by the network during each epoch of training.

**model\_loss = loss \* rewards \* gamma** # calculates the model loss, which is obtained by multiplying the original loss (loss) with the rewards for the current step (rewards) and the discount factor (gamma). The loss variable represents the original loss value, while rewards represents the rewards obtained for the current step, and gamma is the discount factor used in the computation. The resulting model loss is stored in the model\_loss variable.

**print("The model loss is:",model\_loss)** #prints the model loss

**epoch\_losses.append(model\_loss.numpy())** # appends the loss obtained by the network during the current epoch to the epoch\_losses list. This allows us to keep track of the loss obtained by the network during each epoch of training.The numpy() method is used to extract the numerical value of the TensorFlow loss object, which is a symbolic representation of the loss function used to train the network. This numerical value is then appended to the epoch\_losses list.

#Now we computing the gradient using rewards and model loss

**with tf.GradientTape() as tape:** # #tf.GradientTape() is a TensorFlow API that provides a mechanism for automatic differentiation, which is a key technique used in machine learning optimization algorithms, such as gradient descent. It allows you to compute gradients of a computation with respect to its input variables, which can then be used to update the values of those variables during optimization."tape" refers to a mechanism provided by TensorFlow that records operations for the purpose of computing gradients. The tape acts as a context within which computations are recorded, and these computations can later be used to compute gradients using the tape.gradient() method.

**logits = model(x\_batch)** #Passing the batch of input data 'x\_batch' through the model to obtain logits.Logits are the output of the model before applying any activation function, typically used for classification tasks Logits represent the raw, unnormalized scores for each class, which can be used for further processing or prediction.'model' is the trained model that takes 'x\_batch' as input and produces logits as output

**log\_probs = tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0))** # calculates the log probabilities by taking the natural logarithm (tf.math.log()) of the model's predicted logits (logits). The tf.clip\_by\_value() function is used to clip the logits to a specific range to avoid numerical instability. In this case, the minimum value is set to 1e-10 and the maximum value is set to 1.0. The resulting log probabilities are stored in the log\_probs variable.

**model\_loss = tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs \* model\_loss, axis=1))** #calculates the model loss by taking the element-wise multiplication (\*) of the ground truth labels (labels), the logarithm of the clipped logits representing the predicted probabilities (log\_probs), and the model loss calculated as the product of the original loss, rewards, and gamma (model\_loss). Then, the tf.reduce\_sum() function is used to compute the sum along the appropriate axis, and the tf.reduce\_mean() function is used to compute the mean of the resulting values. The final computed model loss is stored in the model\_loss variable.

**grads = tape.gradient(model\_loss, model.trainable\_weights)** #calculates the gradients of the computed model loss (model\_loss) with respect to the trainable weights of the model (model.trainable\_weights). The tape.gradient() function is used to compute these gradients, and the resulting gradients are stored in the grads variable.

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))** #It applies the computed gradients (grads) to update the model weights (model.trainable\_weights) using an optimizer. The zip() function is used to create pairs of gradients and corresponding model weights, which are then passed to the apply\_gradients() method of the optimizer to perform the weight update step. This step is a key part of the optimization process in training machine learning models, as it helps to adjust the model weights based on the gradients of the loss function, with the goal of minimizing the loss and improving the model's performance.

**Train model using REINFORCEMENT LEARNING Technique for NN:**

**NAME:** policy\_gradient

**PARAMETERS:**Call the policy\_gradient function with the following parameters:

* X\_train\_vectors: Training input data (features)
* y\_train: Training target labels
* model: The trained neural network model
* Adam(learning\_rate): Adam optimizer with specified learning rate
* epochs: Number of training epochs
* batch\_size: Batch size for training
* gamma: Discount factor for rewards

**PURPOSE:** This is used to call the function named policy\_gradient with the provided arguments as input parameters.The purpose of this line of code is to invoke the policy\_gradient function and pass in the required input parameters for it to train the model with the reinforcement technique .

**PRECONDITION:** X\_train\_vectors, y\_train, model1, epochs, batch\_size, gamma, and learning\_rate should be appropriately defined and initialized before this function call.

**POSTCONDITION:** The model's weights and biases being updated based on the policy gradient algorithm, the function is designed to update the model's parameters during training.

**CODE:**

policy\_gradient(X\_train\_vectors, y\_train, model1, Adam(learning\_rate), epochs, batch\_size, gamma)

**Define Q-Learning Technique for NN:**

**NAME**: q\_learning

**PURPOSE**: to implement the Q-learning technique for training a neural network model (model) using given input data (x) and target labels (y) with a specified optimizer (optimizer). The algorithm is trained for a given number of epochs (epochs) and batch size (batch\_size), and uses a discount factor (gamma) for computing the Q-values and rewards. The Q-values and rewards are used to modify the loss function and update the model's weights. The code also includes the visualization of reward and loss histograms for analysis. The purpose of this code is to define the Q-learning function for a neural network model, which can be used for reinforcement learning tasks.

**INVARIANTS**:

* The input data x should be a sparse matrix, and length is calculated using the getnnz() method of x.
* The input data y should be a numpy array of target labels for each sample in x.
* The model should be a neural network model implemented using TensorFlow.
* The optimizer should be an instance of a TensorFlow optimizer (e.g. Adam optimizer).
* Epochs should be an integer specifying the number of times to iterate over the input data.
* batch\_size should be an integer specifying the size of each batch during training.
* gamma should be a float specifying the discount factor for future rewards in the Q-learning algorithm.
* The training process involves iterating over the input data in batches of size batch\_size, and for each batch:
* The logits (raw output) of the model for the input data x\_batch are computed using model(x\_batch).
* The log probabilities of the logits are computed using tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0)).
* One-hot encoded labels are created from the target labels y\_batch using tf.one\_hot() method.
* The maximum Q-value for each sample in the batch is computed using tf.reduce\_max(logits, axis=1).
* The maximum Q-value is used as the reward and is converted to a numpy array using q\_values.numpy().
* The policy gradient loss is computed by multiplying the log probabilities with the one-hot encoded labels and taking the negative mean using tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1) \* rewards).
* The gradients of the policy gradient loss with respect to the model's trainable weights are computed using tape.gradient().
* The gradients are applied to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* The average reward for the current batch is computed and used as a baseline.
* The model loss is computed by multiplying the policy gradient loss with the average reward and the discount factor (gamma).
* The gradients of the model loss with respect to the model's trainable weights are computed using tape.gradient().
* The gradients are applied to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* The rewards and losses for each epoch are stored in epoch\_rewards and epoch\_losses, respectively.
* After training, the reward and loss histograms for all epochs are plotted using plt.hist().

**CODE:**

**def q\_learning(x, y, model, optimizer, epochs, batch\_size, gamma):** #class to define the Q-Learning technique

**length = x.getnnz()** #Calculates the length of the input data (x) using getnnz() method.

**for epoch in range(epochs):** #Iterates over each epoch.

**epoch\_rewards = []** #empty lists that are likely intended to store the rewards for each epoch during the training of a neural network.

**epoch\_losses = []** #empty lists that are likely intended to store the losses for each epoch during the training of a neural network.

**for batch\_start in range(0, length, batch\_size)**: #It is used to iterate over the input data (x) in batches of size batch\_size during the training process. It starts from the beginning of the input data and increments in steps of batch\_size until it reaches the end of the data.

**batch\_end = min(batch\_start + batch\_size, length)** # # It calculates the ending index (batch\_end) of the current batch during the iteration over the input data (x) in batches. It ensures that the ending index does not exceed the total length of the data (length) to avoid accessing data beyond the available range.

**x\_batch = x[batch\_start:batch\_end].toarray()** #Extracting a batch of data from the array 'x' using the start and end indices of the batch.toarray() converters the batch to an array, if it's in sparse format, for further processing

**y\_batch = y[batch\_start:batch\_end]** #Extracting a batch of labels from the array 'y' using the start and end indices of the batch

**with tf.GradientTape() as tape: #tf.GradientTape()** is a TensorFlow API that provides a mechanism for automatic differentiation, which is a key technique used in machine learning optimization algorithms, such as gradient descent. It allows you to compute gradients of a computation with respect to its input variables, which can then be used to update the values of those variables during optimization."tape" refers to a mechanism provided by TensorFlow that records operations for the purpose of computing gradients. The tape acts as a context within which computations are recorded, and these computations can later be used to compute gradients using the tape.gradient() method.

**logits = model(x\_batch)** #Passing the batch of input data 'x\_batch' through the model to obtain logits.Logits are the output of the model before applying any activation function, typically used for classification tasks Logits represent the raw, unnormalized scores for each class, which can be used for further processing or prediction.'model' is the trained model that takes 'x\_batch' as input and produces logits as output

**log\_probs = tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0))** # calculates the log probabilities by taking the natural logarithm (tf.math.log()) of the model's predicted logits (logits). The tf.clip\_by\_value() function is used to clip the logits to a specific range to avoid numerical instability. In this case, the minimum value is set to 1e-10 and the maximum value is set to 1.0. The resulting log probabilities are stored in the log\_probs variable.

**labels = tf.one\_hot(y\_batch, depth=output\_dim)** #Converting the batch of labels 'y\_batch' into one-hot encoding using 'tf.one\_hot' function One-hot encoding represents categorical labels as binary vectors with a single '1' and remaining '0's.'y\_batch' is the input tensor containing the batch of labels to be converted to one-hot encoding.'output\_dim' specifies the depth of the one-hot encoding, which should be equal to the number of classes in the classification task

**q\_values = tf.reduce\_max(logits, axis=1)** # Compute the maximum Q-value for each sample in the batch

**rewards = q\_values.numpy()** # Use the maximum Q-value as the reward

**loss = -tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1) \* rewards)** # Modify the loss function to include the reward

**grads = tape.gradient(loss, model.trainable\_weights)** # #grads typically refers to the computed gradients of the loss function with respect to the trainable weights of a machine learning model.The tape.gradient() function in TensorFlow is used to compute the gradients of a given function (in this case, the loss function) with respect to a list of variables (in this case, the model.trainable\_weights). These gradients can then be used in an optimization algorithm, such as gradient descent, to update the model weights and improve the model's performance during training.Compute the gradients of the loss with respect to the trainable weights of the model.loss: The computed loss value.model.trainable\_weights: List of trainable weights of the model

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))** #It applies the computed gradients (grads) to update the model weights (model.trainable\_weights) using an optimizer. The zip() function is used to create pairs of gradients and corresponding model weights, which are then passed to the apply\_gradients() method of the optimizer to perform the weight update step. This step is a key part of the optimization process in training machine learning models, as it helps to adjust the model weights based on the gradients of the loss function, with the goal of minimizing the loss and improving the model's performance.

**epoch\_rewards.append(rewards)** #appends the rewards obtained by the network during the current epoch to the epoch\_rewards list. This allows us to keep track of the rewards obtained by the network during each epoch of training.

**epoch\_losses.append(loss.numpy())** # appends the loss obtained by the network during the current epoch to the epoch\_losses list. This allows us to keep track of the loss obtained by the network during each epoch of training.The numpy() method is used to extract the numerical value of the TensorFlow loss object, which is a symbolic representation of the loss function used to train the network. This numerical value is then appended to the epoch\_losses list.

**print("The rewards are:", rewards)** #prints the rewards

**Train model using Q Learning Technique for NN:**

**NAME:** Q\_learning

**PARAMETERS:**

* Call the Q\_learning function with the following parameters:
* X\_train\_vectors: Training input data (features)
* y\_train: Training target labels
* model: The trained neural network model
* Adam(learning\_rate): Adam optimizer with specified learning rate
* epochs: Number of training epochs
* batch\_size: Batch size for training
* gamma: Discount factor for rewards

**PURPOSE:** This is used to call the function named Q\_learning with the provided arguments as input parameters.The purpose of this line of code is to invoke the Q\_learning function and pass in the required input parameters for it to train the model with the reinforcement technique .

**PRECONDITION:** X\_train\_vectors, y\_train, model, epochs, batch\_size, gamma, and learning\_rate should be appropriately defined and initialized before this function call.

**POSTCONDITION:** The model's weights and biases being updated based on the policy gradient algorithm, the function is designed to update the model's parameters during training.

**Code:**

q\_learning(X\_train\_vectors, y\_train, model, optimizer, epochs=1, batch\_size=32, gamma=0.95)

**Class to build the LSTM model:**

**Name:** Neural Network model

**Purpose:** to define and create a LSTM neural network or classification model for a given dataset.

**Invariants:**

* The input\_dim variable contains the number of features in the input data.
* The output\_dim variable contains the number of classes in the target variable.
* The inputs variable is an input layer that takes a 3-dimensional input of shape (number of samples, 1, input\_dim).
* The x variable is a hidden layer that applies a Long Short-Term Memory (LSTM) operation with 64 units and a Rectified Linear Unit (ReLU) activation function to the input.
* The model2 variable is a Keras Model that takes inputs as input and x as output.
* The optimizer variable is an Adam optimizer with a learning rate of 0.001 that is used to optimize the model during training.

**Code:**

**inputs = Input(shape=(1, input\_dim,))** #creates an input layer in Keras with 3 dimensions, defined by the shape parameter. The first dimension is set to 1, indicating that each input sample will have a single time step. The second dimension is set to input\_dim, indicating the number of features in each time step. The inputs variable stores this input layer, which will be used as the input to the subsequent layers in the model.

**x = LSTM(128, activation='tanh')(inputs)** # creates a Long Short-Term Memory (LSTM) layer in Keras with 64 units and the ReLU activation function. The inputs variable is passed as the input to this layer. The LSTM layer is a type of recurrent neural network layer that is commonly used for processing sequential data, such as text or time series data. The output of the LSTM layer is stored in the x variable and will be used as input to the subsequent layer in the model.

**x = Dense(output\_dim, activation='softmax')(x)** #creates a dense layer with output\_dim number of units and the softmax activation function. The x variable is passed as the input to this layer. The purpose of this layer is to perform the final classification of the input data, with each unit representing a different class. The output of the dense layer will be a probability distribution over the different classes, with the sum of the probabilities equal to 1. The output of this layer is stored in the x variable and will be used as the output of the model.

**model2 = Model(inputs=inputs, outputs=x)** # creates a Keras Model object that specifies the input and output layers of the neural network. The inputs variable represents the input layer of the model, which takes as input a tensor of shape (batch\_size, 1, input\_dim). The x variable represents the output layer of the model, which is the output of the last dense layer with softmax activation. The Model constructor takes two arguments: the input tensor and the output tensor. These tensors define the input and output of the model and all the layers in between. The resulting model2 object can be used to train and evaluate the neural network.

**optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)** # initializes an instance of the Adam optimizer from the Keras API with a learning rate of 0.001. The Adam optimizer is a popular gradient descent optimization algorithm that is commonly used in deep learning.

#optimizer = tf.keras.optimizers.RMSprop(learning\_rate=0.001) #RMSprop optimizer

**model.compile(loss='categorical\_crossentropy', optimizer=optimizer, metrics=['accuracy']**) # is an important line of code in the process of building a deep learning model using Keras. It compiles the model by specifying the loss function, optimizer, and metrics to be used during training.In particular, the loss parameter specifies the loss function that the model will use to evaluate its performance on the training data. In this case, the 'categorical\_crossentropy' loss function is used, which is commonly used for multiclass classification problems.The optimizer parameter specifies the optimization algorithm that will be used to adjust the weights of the model during training in order to minimize the loss function. Here, the optimizer variable is passed in, which should be an instance of a pre-defined optimizer class from Keras, such as Adam or RMSprop.Finally, the metrics parameter specifies the evaluation metrics that will be used to monitor the model's performance during training and testing. In this case, 'accuracy' is the metric used, which is commonly used for classification problems.Overall, model.compile is a crucial step in the process of building and training a deep learning model, as it sets up the model for optimization by specifying the necessary components for the training process.

#model.compile(loss='mean\_squared\_error', optimizer=optimizer, metrics=['accuracy']) #loss:mean\_squared\_error

**Define REINFORCEMENT LEARNING Technique for LSTM:**

**NAME:** policy\_gradient

**PURPOSE:** To implement the policy gradient algorithm for training a LSTM model (model) using given input data (x) and target labels (y) with a specified optimizer (optimizer). The algorithm is trained for a given number of epochs (epochs) and batch size (batch\_size), and uses a discount factor (gamma) for computing the model loss.The purpose of this code is to train a LSTM model using the policy gradient algorithm, where the model's weights are updated based on the computed policy gradient and model loss, and the training process is performed in batches for efficiency.

**INVARIANTS:**

* Calculates the length of the input data (x) using shape() method.length is an integer variable representing the number of rows in the input data x, used to determine the batch sizes and loop iterations in the training process.
* Iterates over each epoch.
* For each epoch, iterates over the input data (x) in batches of size batch\_size.
* For each batch, computes the logits (raw output) of the model for the input data using model(x\_batch).
* Computes the log probabilities of the logits using tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0)).
* Creates one-hot encoded labels from the target labels (y\_batch) using tf.one\_hot() method.
* Computes the policy gradient loss by multiplying the log probabilities with the one-hot encoded labels and taking the negative mean using tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1)).
* Computes the gradients of the policy gradient loss with respect to the model's trainable weights using tape.gradient().
* Applies the gradients to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* Computes the average reward for the current batch and uses it as a baseline.
* Computes the model loss by multiplying the policy gradient loss with the average reward and the discount factor (gamma).
* Computes the gradients of the model loss with respect to the model's trainable weights using tape.gradient().
* Applies the gradients to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* The purpose of this code is to train a neural network model using the policy gradient algorithm, where the model's weights are updated based on the computed policy gradient and model loss, and the training process is performed in batches for efficiency.

**Code:**

**def policy\_gradient(x, y, model, optimizer, epochs, batch\_size, gamma):** #class to define the policy\_gradient RL technique

**length = x.shape[0]** #Calculates the length of the input data (x) using shape() method.

for epoch in range(epochs): #Iterates over each epoch.

**epoch\_rewards = []** #empty lists that are likely intended to store the rewards for each epoch during the training of a neural network.

**epoch\_losses = []** #empty lists that are likely intended to store the losses for each epoch during the training of a neural network.

**for batch\_start in range(0, length, batch\_size):** #It is used to iterate over the input data (x) in batches of size batch\_size during the training process. It starts from the beginning of the input data and increments in steps of batch\_size until it reaches the end of the data.

**batch\_end = min(batch\_start + batch\_size, length)** # It calculates the ending index (batch\_end) of the current batch during the iteration over the input data (x) in batches. It ensures that the ending index does not exceed the total length of the data (length) to avoid accessing data beyond the available range.

**x\_batch = x[batch\_start:batch\_end]** #Extracting a batch of data from the array 'x' using the start and end indices of the batch

**y\_batch = y[batch\_start:batch\_end]** #Extracting a batch of labels from the array 'y' using the start and end indices of the batch

**with tf.GradientTape() as tape: #tf.GradientTape()** is a TensorFlow API that provides a mechanism for automatic differentiation, which is a key technique used in machine learning optimization algorithms, such as gradient descent. It allows you to compute gradients of a computation with respect to its input variables, which can then be used to update the values of those variables during optimization."tape" refers to a mechanism provided by TensorFlow that records operations for the purpose of computing gradients. The tape acts as a context within which computations are recorded, and these computations can later be used to compute gradients using the tape.gradient() method.

**logits = model(x\_batch)** #Passing the batch of input data 'x\_batch' through the model to obtain logits.Logits are the output of the model before applying any activation function, typically used for classification tasks Logits represent the raw, unnormalized scores for each class, which can be used for further processing or prediction.'model' is the trained model that takes 'x\_batch' as input and produces logits as output

**log\_probs = tf.nn.log\_softmax(logits)** #code applies the softmax function to the output logits of the neural network model and then takes the logarithm of the resulting probabilities.The tf.nn.log\_softmax function is a method from the TensorFlow library that applies the softmax function to the logits, which are the raw outputs of the neural network before the activation function is applied. The softmax function converts the logits into probabilities that sum to 1, allowing the outputs to be interpreted as probabilities of each class. the tf.nn.log\_softmax function is used to convert the raw outputs of the neural network into probabilities that can be used for classification, and the logarithm is taken for numerical stability during the calculation of the loss function.

**labels = tf.one\_hot(y\_batch, depth=output\_dim)** #Converting the batch of labels 'y\_batch' into one-hot encoding using 'tf.one\_hot' function One-hot encoding represents categorical labels as binary vectors with a single '1' and remaining '0's.'y\_batch' is the input tensor containing the batch of labels to be converted to one-hot encoding.'output\_dim' specifies the depth of the one-hot encoding, which should be equal to the number of classes in the classification task

**rewards = tf.reduce\_sum(labels \* log\_probs, axis=1)** # Compute the reward using the log probability of the correct label

**loss = -tf.reduce\_mean(rewards)** # Maximize the expected reward

**grads = tape.gradient(loss, model.trainable\_weights)** #grads typically refers to the computed gradients of the loss function with respect to the trainable weights of a machine learning model.The tape.gradient() function in TensorFlow is used to compute the gradients of a given function (in this case, the loss function) with respect to a list of variables (in this case, the model.trainable\_weights). These gradients can then be used in an optimization algorithm, such as gradient descent, to update the model weights and improve the model's performance during training.Compute the gradients of the loss with respect to the trainable weights of the model.loss: The computed loss value.model.trainable\_weights: List of trainable weights of the model

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))** #It applies the computed gradients (grads) to update the model weights (model.trainable\_weights) using an optimizer. The zip() function is used to create pairs of gradients and corresponding model weights, which are then passed to the apply\_gradients() method of the optimizer to perform the weight update step. This step is a key part of the optimization process in training machine learning models, as it helps to adjust the model weights based on the gradients of the loss function, with the goal of minimizing the loss and improving the model's performance.

**epoch\_rewards.append(rewards)** #appends the rewards obtained by the network during the current epoch to the epoch\_rewards list. This allows us to keep track of the rewards obtained by the network during each epoch of training.

**epoch\_losses.append(loss.numpy())** # appends the loss obtained by the network during the current epoch to the epoch\_losses list. This allows us to keep track of the loss obtained by the network during each epoch of training.The numpy() method is used to extract the numerical value of the TensorFlow loss object, which is a symbolic representation of the loss function used to train the network. This numerical value is then appended to the epoch\_losses list.

**print("The rewards are:", rewards)** #prints the rewards

**Train model using REINFORCEMENT LEARNING Technique for LSTM:**

**NAME:** policy\_gradient

**PARAMETERS:**

* Call the policy\_gradient function with the following parameters:
* X\_train\_vectors: Training input data (features)
* y\_train: Training target labels
* model: The trained neural network model
* Adam(learning\_rate): Adam optimizer with specified learning rate
* epochs: Number of training epochs
* batch\_size: Batch size for training
* gamma: Discount factor for rewards

**PURPOSE:** This is used to call the function named policy\_gradient with the provided arguments as input parameters.The purpose of this line of code is to invoke the policy\_gradient function and pass in the required input parameters for it to train the model with the reinforcement technique .

**PRECONDITION:** X\_train\_lstm, y\_train, model2, epochs, batch\_size, gamma, and learning\_rate should be appropriately defined and initialized before this function call.

**POSTCONDITION:** The model's weights and biases being updated based on the policy gradient algorithm, the function is designed to update the model's parameters during training.

**Code:**

policy\_gradient(X\_train\_lstm, y\_train, model2, optimizer, epochs=1, batch\_size=32, gamma=0.95)

**Define Q-Learning Technique for LSTM:**

**NAME:** q\_learning

**PURPOSE:** to implement the Q-learning technique for training a LSTM model (model) using given input data (x) and target labels (y) with a specified optimizer (optimizer). The algorithm is trained for a given number of epochs (epochs) and batch size (batch\_size), and uses a discount factor (gamma) for computing the Q-values and rewards. The Q-values and rewards are used to modify the loss function and update the model's weights. The code also includes the visualization of reward and loss histograms for analysis. The purpose of this code is to define the Q-learning function for a LSTM model, which can be used for reinforcement learning tasks.

**INVARIANTS:**

* Calculates the length of the input data (x) using shape() method.length is an integer variable representing the number of rows in the input data x, used to determine the batch sizes and loop iterations in the training process.
* The input data y should be a numpy array of target labels for each sample in x.
* The model should be a neural network model implemented using TensorFlow.
* The optimizer should be an instance of a TensorFlow optimizer (e.g. Adam optimizer).
* Epochs should be an integer specifying the number of times to iterate over the input data.
* batch\_size should be an integer specifying the size of each batch during training.
* gamma should be a float specifying the discount factor for future rewards in the Q-learning algorithm.
* The training process involves iterating over the input data in batches of size batch\_size, and for each batch:
* The logits (raw output) of the model for the input data x\_batch are computed using model(x\_batch).
* The log probabilities of the logits are computed using tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0)).
* One-hot encoded labels are created from the target labels y\_batch using tf.one\_hot() method.
* The maximum Q-value for each sample in the batch is computed using tf.reduce\_max(logits, axis=1).
* The maximum Q-value is used as the reward and is converted to a numpy array using q\_values.numpy().
* The policy gradient loss is computed by multiplying the log probabilities with the one-hot encoded labels and taking the negative mean using tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1) \* rewards).
* The gradients of the policy gradient loss with respect to the model's trainable weights are computed using tape.gradient().
* The gradients are applied to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* The average reward for the current batch is computed and used as a baseline.
* The model loss is computed by multiplying the policy gradient loss with the average reward and the discount factor (gamma).
* The gradients of the model loss with respect to the model's trainable weights are computed using tape.gradient().
* The gradients are applied to the optimizer using optimizer.apply\_gradients() to update the model's weights.
* The rewards and losses for each epoch are stored in epoch\_rewards and epoch\_losses, respectively.
* After training, the reward and loss histograms for all epochs are plotted using plt.hist().

**Code:**

**def q\_learning(x, y, model, optimizer, epochs, batch\_size, gamma):**#class to define the Q-Learning technique

**length = x.shape[0]** #Iterates over each epoch.

**for epoch in range(epochs):**

**epoch\_rewards = []** #empty lists that are likely intended to store the rewards for each epoch during the training of a neural network.

**epoch\_losses = []** #empty lists that are likely intended to store the losses for each epoch during the training of a neural network.

**for batch\_start in range(0, length, batch\_size):** #It is used to iterate over the input data (x) in batches of size batch\_size during the training process. It starts from the beginning of the input data and increments in steps of batch\_size until it reaches the end of the data.

**batch\_end = min(batch\_start + batch\_size, length)** #It calculates the ending index (batch\_end) of the current batch during the iteration over the input data (x) in batches. It ensures that the ending index does not exceed the total length of the data (length) to avoid accessing data beyond the available range.

**x\_batch = x[batch\_start:batch\_end]** #Extracting a batch of data from the array 'x' using the start and end indices of the batch

**y\_batch = y[batch\_start:batch\_end]** #Extracting a batch of labels from the array 'y' using the start and end indices of the batch

**with tf.GradientTape() as tape:** #tf.GradientTape() is a TensorFlow API that provides a mechanism for automatic differentiation, which is a key technique used in machine learning optimization algorithms, such as gradient descent. It allows you to compute gradients of a computation with respect to its input variables, which can then be used to update the values of those variables during optimization."tape" refers to a mechanism provided by TensorFlow that records operations for the purpose of computing gradients. The tape acts as a context within which computations are recorded, and these computations can later be used to compute gradients using the tape.gradient() method.

**logits = model(x\_batch)** #Passing the batch of input data 'x\_batch' through the model to obtain logits.Logits are the output of the model before applying any activation function, typically used for classification tasks Logits represent the raw, unnormalized scores for each class, which can be used for further processing or prediction.'model' is the trained model that takes 'x\_batch' as input and produces logits as output.

**log\_probs = tf.math.log(tf.clip\_by\_value(logits, 1e-10, 1.0))** # calculates the log probabilities by taking the natural logarithm (tf.math.log()) of the model's predicted logits (logits). The tf.clip\_by\_value() function is used to clip the logits to a specific range to avoid numerical instability. In this case, the minimum value is set to 1e-10 and the maximum value is set to 1.0. The resulting log probabilities are stored in the log\_probs variable.

**labels = tf.one\_hot(y\_batch, depth=output\_dim)** #Converting the batch of labels 'y\_batch' into one-hot encoding using 'tf.one\_hot' function One-hot encoding represents categorical labels as binary vectors with a single '1' and remaining '0's.'y\_batch' is the input tensor containing the batch of labels to be converted to one-hot encoding.'output\_dim' specifies the depth of the one-hot encoding, which should be equal to the number of classes in the classification task

**q\_values = tf.reduce\_max(logits, axis=1)** # Compute the maximum Q-value for each sample in the batch

**rewards = q\_values.numpy()** # Use the maximum Q-value as the reward

**loss = -tf.reduce\_mean(tf.reduce\_sum(labels \* log\_probs, axis=1) \* rewards)** # Modify the loss function to include the reward

**grads = tape.gradient(loss, model.trainable\_weights)** #grads typically refers to the computed gradients of the loss function with respect to the trainable weights of a machine learning model.The tape.gradient() function in TensorFlow is used to compute the gradients of a given function (in this case, the loss function) with respect to a list of variables (in this case, the model.trainable\_weights). These gradients can then be used in an optimization algorithm, such as gradient descent, to update the model weights and improve the model's performance during training.Compute the gradients of the loss with respect to the trainable weights of the model.loss: The computed loss value.model.trainable\_weights: List of trainable weights of the model

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))** #It applies the computed gradients (grads) to update the model weights (model.trainable\_weights) using an optimizer. The zip() function is used to create pairs of gradients and corresponding model weights, which are then passed to the apply\_gradients() method of the optimizer to perform the weight update step. This step is a key part of the optimization process in training machine learning models, as it helps to adjust the model weights based on the gradients of the loss function, with the goal of minimizing the loss and improving the model's performance.

**print("The rewards are:", rewards)** #prints the rewards

**epoch\_rewards.append(rewards)** #appends the rewards obtained by the network during the current epoch to the epoch\_rewards list. This allows us to keep track of the rewards obtained by the network during each epoch of training.

**epoch\_losses.append(loss.numpy())**  # appends the loss obtained by the network during the current epoch to the epoch\_losses list. This allows us to keep track of the loss obtained by the network during each epoch of training.The numpy() method is used to extract the numerical value of the TensorFlow loss object, which is a symbolic representation of the loss function used to train the network. This numerical value is then appended to the epoch\_losses list.

**Train model using Q Learning Technique for LSTM:**

**NAME:** Q\_learning

**PARAMETERS:**

* Call the Q\_learning function with the following parameters:
* X\_train\_vectors: Training input data (features)
* y\_train: Training target labels
* model: The trained neural network model
* Adam(learning\_rate): Adam optimizer with specified learning rate
* epochs: Number of training epochs
* batch\_size: Batch size for training
* gamma: Discount factor for rewards

**PURPOSE:** This is used to call the function named Q\_learning with the provided arguments as input parameters.The purpose of this line of code is to invoke the Q\_learning function and pass in the required input parameters for it to train the model with the reinforcement technique .

**PRECONDITION:** X\_train\_lstm, y\_train, model3, epochs, batch\_size, gamma, and learning\_rate should be appropriately defined and initialized before this function call.

**POSTCONDITION:** The model's weights and biases being updated based on the policy gradient algorithm, the function is designed to update the model's parameters during training.

**Code:**

q\_learning(X\_train\_lstm, y\_train, model3, optimizer, epochs=1, batch\_size=32, gamma=0.95)