**Ashutosh Jha**

**(11811)**

**Ques - Discuss the steps involved in any machine learning task**

We generally follow the following steps to solve a particular machine learning task :

1. Gathering data (Collecting Relevant data)
2. Loading Datasets
3. Data Exploration (Understand what is in a dataset and the characteristics of the data)
4. Data Preparation (Cleaning Data and preparing it for training, Remove duplicates, correct errors, deal with missing values, etc.)
   1. Pre - processing
      1. StandardScalar
      2. MinMaxScalar
      3. Normalization
      4. Discretization
   2. Train test split (Splitting the dataset into training set and test set )
5. Setup Models (Choosing a model, different algorithms are for different tasks)
   1. Pipeline
   2. Supervised
      1. logistic/linear regression
      2. Support vector machine
      3. Naive bayes
      4. Decision trees
      5. KNN
      6. Ensemble methods
   3. Clustering (unsupervised)
      1. k means
      2. Affinity propagation
6. Training (We train the model for training set so as to answer a question or make a prediction correctly as often as possible)
7. Evaluation (Testing the model against previously unseen data, we measure the performance of model using metrics or combination of them)
   1. Classification Metrics
      1. Accuracy Score
      2. F1\_score
      3. Confusion matrix
      4. Precision recall curve
   2. Regression Metrics
      1. Mean absolute error
      2. Mean squared error
      3. R2 score
   3. Clustering Metrics
      1. Adjusted rand score
      2. V measure score
      3. Contingency matrix
8. Model tuning (Also called hyperparameter tuning, we find optimal combination of model parameters for improved performance)
   1. Manual evaluation
   2. Grid search
   3. Randomized search
   4. Cross validation (Also an evaluation method)
   5. Validation curves
9. Prediction

**Ques - Explain backpropagation with the help of pseudocode**

**Que - Explain backpropagation with the help of pseudocode**

Backpropagation is an algorithm commonly used to train neural networks. When we initialize a neural network, we set weights for its individual elements, called neurons. Inputs are loaded, and are passed through the network of neurons, and the network provides an output for each one, given the initial weights. So, Backpropagation helps to adjust the weights of the neurons so that the result comes closer and closer to the known true result.

Backpropagation is simply an algorithm which performs a highly efficient search for the optimal weight values, using the gradient descent technique. It allows to bring the error functions to a minimum with low computational resources, even in large, realistic models. Backpropagation tries to optimize the inter-neuron weights, and not the neurons themselves. Their functionality is pretty much decided when you decide the activation function.

Backpropagation reduces the error function E by taking help of the gradients with respect to individual weights,

Here the error function is : E = ½ (to - ao)2

Ao depends on weightages given by No to its inputs, the ultimate value of E depends on those weights too.

**Pseudocode :**

function BACK-PROP(*example*, *network*) returns a neural network

**inputs** *examples*, a set of examples, each with input vector x and output vector y

*network*, a multilayer network with *L* layers, weights *wi,j*, activation function (*g)*

**local variables**: Δ, a vector of errors, indexed by network node

**repeat**

**for each** weight *wi,j* in *network* do

*wi,j* ← a small random number

**for each** example (x, y) in *examples* do

     /\* *Propagate the inputs forward to compute the outputs* \*/

**for each** node *i* in the input layer do

*ai* ← *xi*

**for** *l* = 2 to *L* do

**for each** node *j* in layer *l* do

*inj* ← Σ*i* *wi,j* *ai*

*aj* ← *g*(*inj*)

     /\* *Propagate deltas backward from output layer to input layer* \*/

**for each** node *j* in the output layer do

       Δ[*j*] ← *g*′(*inj*) × (*yi* − *aj*)

**for** *l* = *L* − 1 to 1 do

**for each** node *i* in layer *l* do

         Δ[*i*] ← *g*′(*ini*) Σ*j* *wi,j* Δ[*j*]

     /\* *Update every weight in network using deltas* \*/

**for each** weight *wi,j* in *network* do

*wi,j* ← *wi,j* + *α* × *ai* × Δ[*j*]

**until** some stopping criterion is satisfied

**return** *network*

***Ques. 2:*** Design a CNN architecture where the input image is of size 28x28 and the output is a binary class. Using at least 4 convolution layers using at least 2 different kernel sizes and 2 different strides. You may pad according to need. Write the height, width, depth after every layer. *(4 marks)*

***Sol. 2:***

// Importing the libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten, Dense, Dropout

// Training data and Testing data is generated

...

...

...

// CNN Architecture

// 1. CNN is initialised

model = Sequential()

// 2. First Convolution layer and a Pooling layer is added

model.add(Conv2D(filters = 512, kernel\_size = 5, strides=(1,1), activation=’relu’, input\_shape = [28, 28, 3]))

// (Height, Width, Depth) = ()

model.add(MaxPool2D(pool\_size=(2,2), strides=2))

// 3. Second Convolution layer and a Pooling layer is added

model.add(Conv2D(filters = 512, kernel\_size = 5, strides=(2,2), activation=’relu’))

model.add(MaxPool2D(pool\_size=(2,2), strides=2))

// 4. Third Convolution layer and a Pooling layer is added

model.add(Conv2D(filters = 256, kernel\_size = 3, strides=(1,1), activation=’relu’))

model.add(MaxPool2D(pool\_size=(2,2), strides=2))

// 5. Fourth Convolution layer and a Pooling layer is added

model.add(Conv2D(filters = 128, kernel\_size = 3, strides=(2,2), activation=’relu’))

model.add(MaxPool2D(pool\_size=(2,2), strides=2))

// 6. Flattening the layers

model.add(Flatten())

// 7. Two Hidden layers with respective Dropout layers (to prevent overfitting) are added

model.add(Dense(units = 128, activation = ‘relu’))

model.add(Dropout(0.5))

model.add(Dense(units = 64, activation = ‘relu’))

model.add(Dropout(0.25))

// 8. Output Layer - Binary output is required

model.add(Dense(units = 1, activation=’sigmoid’))

// Compiling the model

model.compile(optimizer=’adam’, loss=’binary\_crossentropy’, metrics=[‘accuracy’])

// optimizer is set to ‘adam’ in purpose to use Stochastic Gradient Descent while training the model, loss is set to ‘binary\_crossentropy’ as the output is binary and metrics is set to ‘accuracy’ in order to train and backpropagate the model by accuracy.

// Model is trained

...

...

...

// Model is saved

model.save(‘model.h5’)

***Ques. 3:*** Discuss the following in the context of reinforcement learning:

1. exploration vs exploitation;
2. State value function vs action-value function;
3. discuss 2 challenges and solutions with a design in which an agent is being trained to play chess where the reward for taking(killing) a piece is +1 and the reward for winning the game is +10. *(4 marks)*

***Sol. 3:***

1. The exploration-exploitation dilemma is a fundamental problem in reinforcement learning as well as in real life which we frequently face when choosing between options, would you rather:
   1. pick something you are familiar with in order to maximise the chance of getting what you wanted. ***(Exploitation)***
   2. or pick something you have not tried and possibly learning more, which may (or may not) result in you making better decisions in future. ***(Exploration)***

Exploration means that your algorithm is searching the space of possible solutions as well, while exploitation takes advantage of existing best solutions/schema in the population.

1. The Value Function represents the value for the agent to be in a certain state. More specifically, the state value function describes the expected return G\_t from a given state. In general, a ***state value function*** is defined concerning a specific policy, since the expected return depends on the policy:

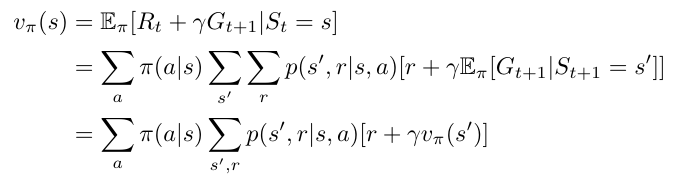


The index π indicates the dependency on the policy. Furthermore an ***action-value function*** can be defined. The action-value of a state is the expected return if the agent chooses action a according to a policy π.

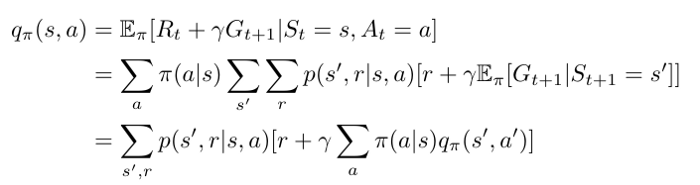


Value functions are critical to Reinforcement Learning. They allow an agent to query the quality of his current situation rather than waiting for the long-term result. This has a dual benefit. First, the return is not immediately available, and second, the return can be random due to the stochasticity of the policy as well as the dynamics of the environment. The value function summarizes all future possibilities by averaging the returns. Thus, the value function allows an assessment of the quality of different policies.

A fundamental property of value functions used throughout RL is that they satisfy recursive relationships. For each policy and state s, the following consistency condition applies between the value of s and the value of its possible subsequent states:



This equation is also called the Bellman equation. For the Value Function the Bellman equation defines a relation of the value of State s and its following State s′. The Bellman equation is also used for the Action-Value function. Accordingly, the Action-Value can be calculated from the following state:



In the Bellman equations the structure of the MDP formulation is used to reduce this infinite sum to a system of linear equations. By directly solving the equation, the exact state values can then be determined.