Deep Learning Hyperparameters & models for Regression Problems

In [ ]:
```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

In [ ]:
```
# load the dataset
df = pd.read_csv('dataset.csv')
df
```

In [ ]:
```
#scale the data
# MinMaxScaler / StandardScaler
```

In [ ]:
```
#MinMaxScaler

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(traindata)
scaled_train = scaler.transform(traindata)
scaled_test = scaler.transform(testdata)
```

In [ ]:
```
#StandardScaler

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(traindata)
scaled_train = scaler.transform(traindata)
scaled_test = scaler.transform(testdata)
```

In [ ]:
```
# split into input (X) and output (Y) variables
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

In [ ]:
```
# determine the number of input features
n_features = X_train.shape[1]  #all columns as features
print(n_features)
```

Sequential models

In [ ]:
```
# simple dl model
model = Sequential()
model.add(Dense(10, kernel_initializer='he_normal', activation='relu', input_shape=(n_features,)))
model.add(Dense(1, kernel_initializer='he_normal'))
```

Hyperparameter tuning

says about all the ways that can be used for the optimization of our neural networks.

Important Parameters to be considered while tuning a Neural Network

- Number of Layers in a Neural Network
- The Dropout Value
- Learning Rate of the Neural Network
- The Batch Size

Choosing the Right Number of Hidden Layers in a Neural Network

- Choose 3 to 5 Hidden Layers for a Medium-Sized Dataset.
- Neural Networks Might Suffer from underfitting if the hidden layers are less than 2.
- Similarly, the Neural Networks can suffer from overfitting if the Number of Hidden layers is too high.
- For Bigger Problems, Slowly increase the Hidden Layers and Check to see if Increasing Hidden Layers optimizes the Neural Network.

Choosing the right Activation Function

----- 'relu' activation function is used for regression problems

- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoid and tanh functions are sometimes avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used in most cases these days
- If we encounter a case of dead neurons in our networks, the leaky ReLU function is the best choice
- The ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using the ReLU function and then move over to other activation functions in case ReLU doesn't provide optimum results.

In [ ]:
```
# dl model with more dense layers
model = Sequential()
model.add(Dense(20, input_shape=(n_features,), kernel_initializer='he_normal', activation='relu'))
model.add(Dense(10, kernel_initializer='he_normal', activation='relu'))
model.add(Dense(1, kernel_initializer='he_normal'))
```

REGULARIZATION Techniques

- when the model performs better on train dataset but prediction result is poor on test dataset
- regularization layers help to avoid overfitting and more efficient model
- regularization techniques improves model performance and converge faster
- regularization tools : early stopping, dropout, weight initialization techniques, and batch normalization

In [ ]:
```
model = Sequential()
model.add(Dense(50, kernel_initializer='he_normal', activation='relu', input_shape=(n_features,))))
model.add(BatchNormalization())
model.add(Dense(30, kernel_initializer='he_normal', activation='relu'))
model.add(BatchNormalization())
model.add(Dense(10, kernel_initializer='he_normal', activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1, kernel_initializer='he_normal'))
```

Batch normalization process

- Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch.
- faster training with better accuracy
- smoothens the loss function, optimizes model parameters
- appropriate way to add this layer before or after activation functions
- should not be used along with Dropout layer

- used in MLP, CNN, RNN architectures

Read more :

- https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/
- https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/
- https://towardsdatascience.com/different-normalization-layers-in-deep-learning-1a7214ff71d6
- https://www.kaggle.com/ryanholbrook/dropout-and-batch-normalization

In [ ]:
```
model = Sequential()
model.add(Dense(50, activation='relu', input_shape=(n_features,))))
model.add(BatchNormalization())
model.add(Dense(30, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(10, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1))
```

Using Dropout and Batch Normalization together

Red Wine model

- Dropout to control overfitting
- Batch normalization to speed up optimization
- when adding Dropout with Batch normalization, in the dense layer there will be higher units

Dropout

The term dropout refers to dropping or removing neurons at random from a hidden layer in a neural network to avoid overfitting.

What should be the optimum value of a dropout in a hidden layer?

- During the training, the activation is randomly set to 0. Typically 50% of activations in layers are dropped.
- Most of the time, the Value of 0.2 to 0.4 is the Most preferable for Dropout, which means deactivating 20 to 40% of the Neurons from the Hidden Layers.
- Always Try to Maintain a Dropout Value lesser than 0.5, As Higher than 0.5 Value for Dropout can lead to Overfitting.
- Dropout values Less than 0.2 value will have no or less Significance.

In [ ]:
```
model = Sequential()
model.add(Dense(1024, activation='relu', input_shape=(n_features,)))
model.add.Dropout(0.3)
model.add(BatchNormalization())
model.add(Dense(1024, activation='relu'))
model.add.Dropout(0.3)
model.add(BatchNormalization())
model.add(Dense(1024, activation='relu'))
model.add.Dropout(0.3)
model.add(BatchNormalization())
model.add(Dense(1024, activation='relu'))
model.add.Dropout(0.3)
model.add(BatchNormalization())
model.add(Dense(1))
```

Read more.....

- Book: https://b-ok.asia/book/12248872/86a2fc

Learning rates

- Tuning parameter for Optimization Algorithms
- Optimizers : Momentum, RMS Prop, Adagrad, Adadelta, Adam

read more:

- https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/

Adam :

Adam converges to a better loss, faster compared to the others. Adam Optimizer is quite tried and tested, and in many deep learning projects, Adam Optimizer is used mostly by practitioners. It is easy to implement, Computationally efficient, Little memory requirements.

Gradient Descent Algorithms as Optimization techniques :

- Gradient Descent is also known as 'Backpropagation' is an optimization technique that is used to improve deep learning and neural network-based models by minimizing the loss function.
- In simple terms, Gradient Descent is used for training a deep neural network.
- Two important terms here - 1) Minimization 2) Loss Function

-Loss Function: --- 'mse'(mean squared error) is used for regression problems.

The other most common loss functions:

- 'binary_crossentropy'--- binary classification.
- 'sparse_categorical_crossentropy' ---- multi-class classification.

- Loss function quantifies the amount of error in your predictions, It is a metric that is directly related to the performance of the model.

- If Predictions are Good, then the Loss function would output a Lower Number and if your predictions are totally off, the Loss function will output a higher number.
- Our Loss Function should always justify two things while generating,
- It should always be low or negative-oriented: it is always a better option
- It should be differentiable: should be able to apply derivative on it

-Minimization:

- As loss function is negative-oriented, the neural network always strives for making it less, this process is called Minimization.

In [ ]:
```
# learning rate added in a simple MLP model
model = Sequential()
model.add(Dense(50, input_dim =2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Batch gradient descent algorithm We take the whole Training Data as a batch and pass it to the network. The network calculates the Gradient for the whole Training Data and Updates the weights. If we observe closely an epoch in batch gradient descent, we just update our weights once.

- Advantage:
- Speed As the whole data is passed as a batch, the network would process the data at once in a vectorized way. We all know modern GPUs love batch processing, so the speed at which we train the model is super high.
- Disadvantage:
- Poor Learning in Initial Phases: As the weights are updated just once. For every epoch, there is a high chance of the model underfitting, given our problem statement is complex to solve.
- High Memory Consumption: Fitting the whole data at once is impossible unless your data is too small. Imagine you want to train a model on data of size 1000GB. So while using batch gradient descent you would have to bring this 1000GB onto your GPU or RAM, which is impossible.
- It is recommended to use batch gradient descent when the data set is small and the problem statement is not too complex.

Stochastic gradient descent algorithm (SGD) Here we take each data point and calculate its gradient. Next, we update weights and repeat this process. So if our dataset has 1000 data points using stochastic gradient descent, we calculate gradients and update weight 1000 times in one epoch.

- Advantage: Low memory consumption: as we just load one data point at a time. There is no restriction on how big our data can be.
- Disadvantage: Low speed: Here we do not have any batch processing nor vectorization, which means it is very slow to train.
- Over learning: As the weights are updating very frequently, there is a chance for our model to overfit over simple problem statements as well.
- If the data set is small and the problem statement is complex, then try stochastic gradient descent. **Mini Batch Gradient Descent:** Here, we internally match the training data. We then calculate gradient and update weights for each batch. We repeat this process until gradients over the whole training data are calculated and weights are updated. So, if our dataset has 1000 examples and our batch size is supposed 100, then we update the weights 10 times for each epoch.
- Advantages: Low memory consumption, high speed Better memory optimization compared to batch gradient descent due to many batches as weights are updated less frequently compared to Stochastic gradient descent. It has fewer chances of overfitting. If your dataset is large, then mini-batch gradient descent should be used.

In [ ]:
```
from keras.callbacks import EarlyStopping, ModelCheckpoint

model = Sequential()
# Add fully connected layer with a ReLU activation function
model.add(Dense(512, activation='relu', input_shape=(10,)))
# Add fully connected layer with a ReLU activation function
model.add(Dense(256, activation='relu'))
# Add fully connected layer with a sigmoid activation function
model.add(Dense(128, activation='sigmoid'))
#Compile neural network
model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=['accuracy'])
callbacks = [EarlyStopping(monitor='val_loss', patience=2),
             ModelCheckpoint(filepath='MNIST_pred', monitor='val_loss', save_best_only=True)]

print(callbacks)

# Train neural network
history = network.fit(X_train, y_train, epochs=20, callbacks=callbacks, verbose=0, batch_size=100, validation_data=(X_test, y_test))
```

Early Stopping Regularization

- Early stopping is basically stopping the training once you reached the minimum of your losses or errors.
- When we use too many epochs it leads to overfitting, too less epochs leads to underfitting of the model.This method allows us to specify a large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

In [ ]:
```
# Compile model
model.compile(optimizer='Adam', loss='mean_squared_error',  metrics=['accuracy'])
```

In [ ]:
```
# fit the model
model.fit(X_train, y_train, epochs=250, batch_size=128, verbose=2)
```

Model Evaluation matrices :

Read More:

- https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/
- https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234

Batch size & Epochs

Read more :

- https://deeplizard.com/learn/video/U4WB9p6OOjM

Batch Size:

The selection of batch size depends on the RAM or GPU Memory. The more RAM we have, the higher Batch size can be fitted into Memory.

How to select the Batch Size in Neural Networks?

- Generally The Batch Size of 32 is Good, But we should also try the Batch Sizes of 64, 128, and 256.
- If the problem statement is complex, having a low batch size works better as low batch size and mini-batch gradient descent we will have more weight updates which ensures our weights are better tuned.
- If your problem statement is simple, then try to have the maximum batch size that can fit your memory.

Epochs:

- The number of epochs is a hyperparameter that defines the number of times that the learning algorithm will work through the entire training dataset.