Tyler Bryk
CPE-695 Applied Machine Learning
Homework 5

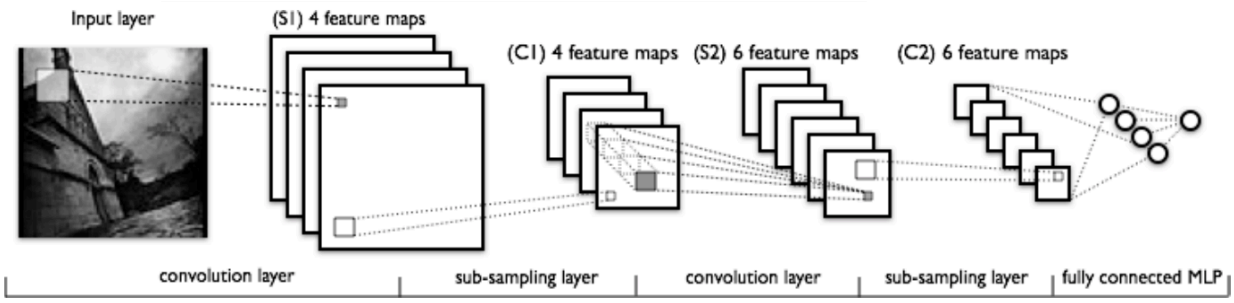
Problem 1.1:
Please explain the pros and cons of Instance-Based Learning and Model-Based Learning respectively.

Instance-based Learning
Typical machine learning algorithms often generalize or form an abstraction of different training data points that are supplied to the model. In instance-based learning, this is not the case. Here, instead of generalizing new data points as they come in, the model explicitly compares the new data instances with previous instances seen in past training data. This style of learning directly compares new data with old data, and it does not perform any operations on the data before or after making these comparisons. Because of this, there are no model parameters to tune, which can be seen as an advantage. Additionally, since no operations are performed, training is very fast, and complex target functions can be learned. However, since all of the calculations are done at query time, the model can be very slow when trying to retrieve new information. Lastly, the models can be easily fooled by irrelevant information because of the fact that they look at all features, not just the important ones.

Model-based Learning
Model-based learning, also known as reinforcement learning is an approach where actions are observed from an environment in order to stimulate learning. For example, in reinforcement learning, the model is given the option to take certain actions based on data that is inputted. Based on these actions, either a penalty or reward is given out which the model can learn from. Essentially, the optimal behavior of a system can be achieved by having the model learn from various outcomes that happen as a result from the models predictions / actions. One large advantage of this learning style is that it essentially represents a framework of models, where a large number of different model types can be utilized to solve a problem. Another advantage of model-based learning is that the model parameters do not change as the size of the training data increases because the models are parameterized with a fixed number. On the other hand, model-based learning is computationally expensive, and it does not learn well in the absence of sufficient training data.

Problem 1.2:
Please draw the diagram of Convolutional Neural Networks (CNN). Then explain the functionality of each layer of CNN. Name several latest algorithms of CNN (e.g., AlexNet etc.).

The picture below represents a diagram of the Convolutional Neural Network (CNN)



As shown in the diagram, all CNN's have very similar architectures. They all contain convolution layers, sampling/pooling layers, and fully-connected layers.

Convolution Layer:
The convolution layer performs a mathematical operation on the data where the input is convolved with a mask or filter. Merging the input data with a convolution filter will result in a feature map being created. This is typically done with a 3x3 or 5x5 convolution filter, and it is done many times in each step with different filters so that multiple feature maps are produced. The resultant feature maps are added together, and then passed through a ReLU to achieve non-linearity.

Sub-Sampling/Pooling Layer:
The pooling layers are used to reduce the dimensionality and number of parameters that result from the preceding convolution layer. The pooling layers look at each feature map independently, and they scale down the length and width, keeping the depth intact. This layer helps reduce the overall amount of weights that need to be calculated during training, which keeps training time and cost as low as possible.

Fully-Connected Layer:
The fully-connected layer is used to wrap up the architecture of the preceding convolution and pooling layers. The FC layer is no different in a CNN than it is in a normal ANN. Once the input data is convolved and scaled down, that result is used as the input to the FC layers which act like normal NN's. Lastly, the FC layer is trained like an ANN as well, using back-propagation and gradient descent.

CNN Algorithms:
Some of the latest CNN algorithms include: AlexNet (2012), ZFNet (2013), GoogLeNet/ Inception (2014), VGGNet (2014), and ResNet (2015).

Problem 1.3:

When training deep networks using Back-propagation, one difficulty is so-called "diffusion of gradient", i.e., the error will attenuate as it propagates to early layers. Please explain how to address this problem.

The vanishing gradient problem, or 'diffusion of gradients' is a common phenomena that occurs when training deep networks with back-propagation. Namely, when using back-propagation to update the training weights, certain activation functions cause the gradient to be within the range of (-1,1) which results in the gradient being very small. As the gradient propagates through the layers, it becomes exponentially smaller each time, so the early network layers hardly get updated because of this small change in weight. This phenomena is known as the vanishing gradient problem.

One common solution for the vanishing gradient problem is to instead train the deep network following a multi-layer hierarchy, using something that is known as greedy layer-wise training. In this process, the network is trained one layer at a time with unlabeled data. The first layer is trained, then the first layer and its parameters are frozen while the second layer is trained. This process is repeated for any number of layers, and finally, the output of this process is fed into a supervised learning model which is fine tuned by using back-propagation. This process alleviates the vanishing gradient problem because it allows early layers to learn equally since each layer is trained in isolation.

Another simple solution to solve the vanishing gradient problem is to use an activation function that does not saturate the output. The ReLU activation function is particularly expressive because it only saturates the output in one direction, due to its linear form, and it actually speeds up the convergence of most SGD algorithms. Lastly, using long short-term memory (LSTM) and gated recurrent units (GRU's) can help with the vanishing gradient problem because they both help the model memorize a longer history. In typical back-propagation, the garden becomes small which means that longer history is forgotten in earlier layers. When using LSTM or GRU's, this is no longer the case because the recurrent structure allows the model to hold onto long term memory, making the vanishing gradient problem less applicable. Overall, the vanishing gradient problem can be solved by using multi-layer hierarchies, non-saturating activation functions, and long-term memory modules such as LSTM and GRU's.

CPE695_HW5

November 29, 2020

1 CPE695 HW5

By: Tyler Bryk

In this assignment, we will design a genetic algorithm to perform polynomial fitting on a randomly generated dataset. The genetic algorithm will create new offspring by using both mutation and crossover operations. First, the sample data will be created which follows the curve Y = x^3 + 20x^2 + 5x + Noise. Then the genetic algorithm will be implemented, and the results will be compared with the built-in polyfit() function.

[1]:
```
# Import Libraries
import math
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
plt.style.use('seaborn-whitegrid')
```

[2]:
```
# Define Simulated Data from HW1
noise = 100
samples = 50
x = 25 * (np.random.rand(samples, 1) - 0.8)
y = 5 * x + 20 * x**2 + 1 * x**3 + noise * np.random.randn(samples, 1)
x_train, x_test, y_train, y_test = train_test_split(x.squeeze(), y.squeeze(),␣
↪test_size=0.30, random_state=10413641)
```

[3]:
```
# Helper Functions
def generateChromos(n):
  chromosomes = []
  for i in range(n):
    chromosomes.append(np.random.randint(0,25,4))
  return chromosomes

def MSE(chromosomes):
  error = []
  for chr in chromosomes:
      poly = np.poly1d(chr)
      preds = []
      for x in x_train:
```

```
        preds.append(poly(x))
      error.append(np.square(y_train - preds).mean())
  return error

def findKMin(data, k):
  bestData = sorted(data)[:k]
  bestIdx  = []
  for d in bestData:
    bestIdx.append(data.index(d))
  return bestIdx
```

[4]:
```
def polyFitGA(iter=500, mutationRate=0.1, crossoverRate=1):
  chromosomes = generateChromos(100)
  for i in range(iter):
    # Compute Error of Chromosomes
    error = MSE(chromosomes)

    # Keep 10 Best Chromosomes
    keptChromos = []
    bestIdxs = findKMin(data=error, k=10)
    for idx in bestIdxs:
      keptChromos.append(chromosomes[idx])

    # Generate New Chromosomes
    chromosomes = []
    for chrIdx in range(0,len(keptChromos),2):
      chr1 = keptChromos[chrIdx]
      chr2 = keptChromos[chrIdx+1]
      chromosomes.append(chr1)
      chromosomes.append(chr2)
      while len(chromosomes) < (20*((chrIdx/2)+1)):
        #Crossover
        if (np.random.uniform(0.0, 1.0) <= crossoverRate):
          randIdx = np.random.randint(0,3)
          newChromo = []
          for i1 in range(randIdx+1):
            newChromo.append(chr1[i1])
          for i2 in range(randIdx+1, len(chr2)):
            newChromo.append(chr2[i2])
          chromosomes.append(np.array(newChromo))

        # Mutation
        if (np.random.uniform(0.0, 1.0) <= mutationRate):
          randIdx = np.random.randint(0,4)
          randMlt = np.random.uniform(0.0, 2.0)
          if np.random.randint(0,10) < 5: tmp = chr1.copy()
          else: tmp = chr2.copy()
```

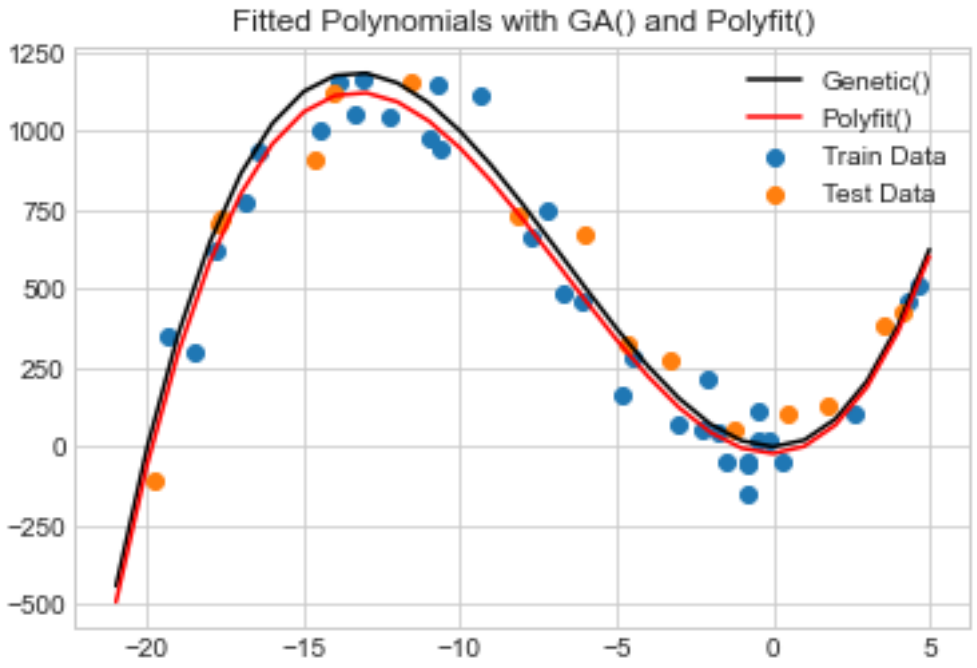
```
            tmp[randIdx] = tmp[randIdx] * randMlt
            chromosomes.append(tmp)

    # Pick Best Chromosome
    error = MSE(chromosomes)
    bestIdx = findKMin(data=error, k=1)
    return chromosomes[bestIdx[0]]
```

[5]:
```
# Train Both GA() and Polyfit() Models
polyPF = np.poly1d(np.polyfit(x_train, y_train, 3))
polyGA = np.poly1d(polyFitGA(iter=500, mutationRate=0.10, crossoverRate=1.00))
```

[6]:
```
# Plot Fitted Polynomials with GA() and Polyfit()
xfitPF, xfitGA, yfitPF, yfitGA = [],[],[],[]
for i in range(math.floor(min(x_train))-1, math.ceil(max(x_train))+1):
    xfitPF.append(i)
    yfitPF.append(polyPF(i))
    xfitGA.append(i)
    yfitGA.append(polyGA(i))

plt.title('Fitted Polynomials with GA() and Polyfit()')
plt.plot(xfitGA, yfitGA, 'k')
plt.plot(xfitPF, yfitPF, 'r')
plt.scatter(x_train, y_train)
plt.scatter(x_test, y_test)
plt.legend(['Genetic()', 'Polyfit()', 'Train Data', 'Test Data'])
plt.show()
```



[7]:
```
# Show Testing Error for GA() & Polyfit()
predsGA, predsPF = [],[]
for x in x_test:
  predsGA.append(polyGA(x))
  predsPF.append(polyPF(x))
errorGA = np.square(y_test - predsGA).mean()/100
errorPF = np.square(y_test - predsPF).mean()/100
print(f"Total Error of Genetic Algorithm: {int(errorGA)}")
print(f"Total Error of Polyfit Function:  {int(errorPF)}")
```

```
Total Error of Genetic Algorithm: 109
Total Error of Polyfit Function:  106
```

As shown in the results above, the error of the genetic algorithm is very close to that of the polyfit() function. We can also empirically visualize this based on the graph displayed above. The ideal parameters for the GA were mutation rate = 0.01 and crossover rate = 1.00. Overall, both models did a very good job of fitting polynomials to the raw sample data.