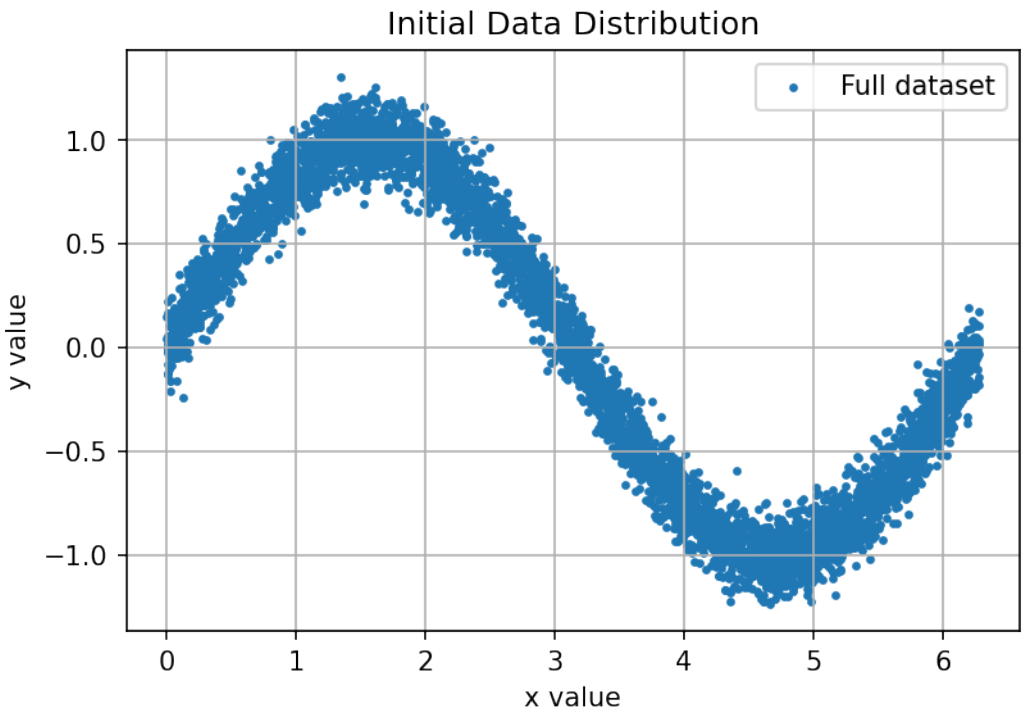
Homework_1_Weiss

May 25, 2023

[ ]:
```
import os
import struct
import numpy as np
import sys
import gzip
import shutil
import matplotlib.pyplot as plt
from sklearn.model_selection import ParameterGrid, train_test_split
from sklearn.neural_network import MLPRegressor
import warnings
warnings.filterwarnings("ignore")

import matplotlib as mpl
mpl.rcParams['figure.dpi']= 150
```

0.0.1 Preparation: Generate the noisy sin data and plot them

[ ]:
```
n_samples=5000
x_values=np.random.uniform(low=0,high=2*np.pi,size=n_samples)
sin_wave=np.sin(x_values)+(0.1*np.random.randn(x_values.shape[0]))
plt.scatter(x_values,sin_wave, label="Full dataset", s=5)
plt.xlabel("x value")
plt.ylabel("y value")
plt.grid(True)
plt.legend()
plt.title("Initial Data Distribution")
plt.show()
```



1 a)

Use 50%, 25% and 25% for the training, validation and testing data set and randomly select these data from the total data set. Make a scatter plot (x versus sin(x)) of these three data sets To define the MLP the class MLPRegressor from sklearn has been used (see https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html), the parameter n_iter_no_change has been set equal to the number of epochs to be sure that the training procedure will run till the end.

[ ]:
```
class MLP():

    def _compute_cost(self, y, output):

        squared_weights_sum=0
        weights=np.array(self.mlp.coefs_)

        for layer_weights in weights:
            squared_weights_sum+=np.sum(layer_weights**2)

        L2_term = (self.l2 *
                   (squared_weights_sum))

        cost = np.sum(np.square(output-y))/output.shape[0] + L2_term
```

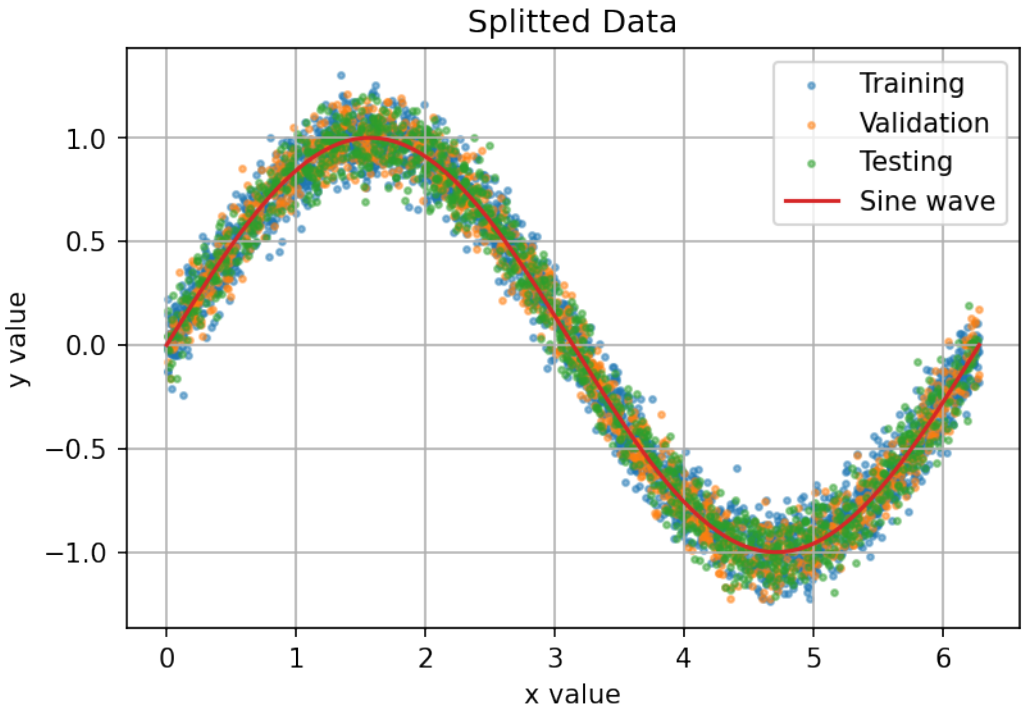
```
        return cost


    def __init__(self,n_hidden=100,seed=2,l2 = 0.001,epochs = 80,eta = 0.
↪005,shuffle = True, minibatch_size = 100):

        self.n_hidden=n_hidden
        self.seed=seed
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.shuffle = shuffle
        self.minibatch_size = minibatch_size
        self.mlp=None

    def train(self,X_train,y_train,X_validation,y_validation):
        # validation data is only used for rmse calculation, not for the␣
↪training itself

        loss_curve_training=[]
        loss_curve_validation=[]

        epoch_strlen = len(str(self.epochs))
        self.eval_ = {'cost': [], 'train_mse': [], 'valid_mse': []}

        self.mlp=MLPRegressor(hidden_layer_sizes=self.
↪n_hidden,activation='relu',max_iter=self.epochs,solver='adam',
                              shuffle=self.shuffle,
                              batch_size=self.
↪minibatch_size,learning_rate_init=self.eta,verbose=False,alpha=self.l2,
                              n_iter_no_change=self.epochs)

        for i in range(self.epochs):

            self.mlp.partial_fit(X_train,y_train)
            y_pred_training=self.predict(X_train)
            y_pred_validation=self.predict(X_validation)
            mse_training=np.mean(np.square(y_train-y_pred_training))
            mse_validation=np.mean(np.square(y_validation-y_pred_validation))

            loss_curve_training.append(mse_training)
            loss_curve_validation.append(mse_validation)

            cost = self._compute_cost(y=y_train,
                                      output=y_pred_training)
```

```
            sys.stderr.write('\r%0*d/%d | Cost: %.2f '
                             '| Train/Valid MSE.: %.2f/%.2f ' %
                             (epoch_strlen, i+1, self.epochs, cost,
                              mse_training, mse_validation))
            sys.stderr.flush()

            self.eval_['cost'].append(cost)
            self.eval_['train_mse'].append(mse_training)
            self.eval_['valid_mse'].append(mse_validation)

    def evaluate(self,X,y):

        y_pred=self.predict(X)
        mse=np.mean(np.square(y-y_pred))
        print("mean squared error:{}".format(mse))

    def predict(self,X):

        if(self.mlp==None):
            print("model not trained")
        else:
            predictions=self.mlp.predict(X)

        return predictions
```

Here, the data is split and plotted with a sine wave for reference.

[ ]:
```
# Splitting the data into training, validation and testing sets
x_train, x_test, y_train, y_test = train_test_split(x_values, sin_wave,␣
 ↪test_size=0.25, random_state=0)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=1/
 ↪3, random_state=0)

# Making scatter plots of the three data sets
plt.scatter(x_train, y_train, label='Training', s=5, alpha=0.5)
plt.scatter(x_val, y_val, label='Validation', s=5, alpha=0.5)
plt.scatter(x_test, y_test, label='Testing', s=5, alpha=0.5)

# Plotting a sine wave for reference
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)
plt.plot(x, y, label='Sine wave', color='tab:red')

plt.legend()
plt.xlabel("x value")
plt.ylabel("y value")
plt.grid(True)
```

```
plt.title("Splitted Data")
plt.show()
```



[ ]:
```
# reshape data
x_train, x_val, x_test = x_train.reshape(-1, 1), x_val.reshape(-1, 1), x_test.
↪reshape(-1, 1)
```

2 b)

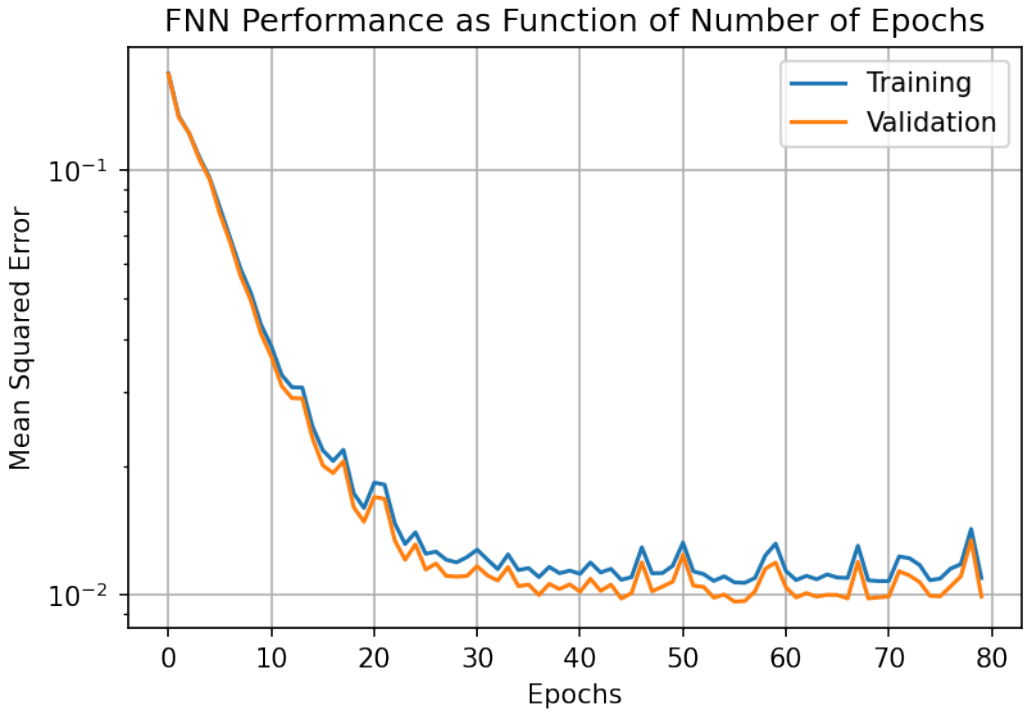
Plot the training and validation error (e.g. the mean squared error) as a function of the number of epochs. The plot can be found below. It is evident that the performance does not continue to increase after around 30 epochs. Before, the MSE shows an approximately exponential decrease with every epoch.

The development MSE of training and validation data behaves very similarly. This is because, essentially, both data sets contain random data of a sine. Due to the large number of datapoints in each set, the randomness is almost the same in both sets. This also implies that the model is able to use its training from the training data to make predictions of similar quality with the validation data (which is not suprising given the similarity of the two datasets).

[ ]:
```
# Create an instance of the MLP class
mlp = MLP()
```

```python
# Train the model
mlp.train(x_train, y_train, x_val, y_val)

# Plot the training and validation error
plt.plot(mlp.eval_['train_mse'], label='Training')
plt.plot(mlp.eval_['valid_mse'], label='Validation')
plt.xlabel('Epochs')
plt.ylabel('Mean Squared Error')
plt.legend()
plt.grid(True)
plt.yscale("log")
plt.title("FNN Performance as Function of Number of Epochs")
plt.show()
```

```
80/80 | Cost: 0.03 | Train/Valid MSE.: 0.01/0.01
```



3 c)

Describe the effect of the learning rate on the behaviour of the loss function. What is the optimal value of this hyper parameter? The learning rate is a hyperparameter that controls the step size of the weight updates during training. It determines how quickly or slowly the model learns from the data. The loss function itself is not directly influenced by the learning rate. Only the weights itself play a part in the loss function but they are being adjusted using learning rate.

The learning rate is specified by the eta parameter when creating an instance of the MLP class and is passed to the MLPRegressor from scikit-learn when creating an instance of this class in the train method. The MLPRegressor uses the learning rate to update the weights of the model during training. Specifically, it uses an optimization algorithm called Adam to update the weights. The Adam algorithm computes adaptive learning rates for each weight based on the first and second moments of the gradients. The learning rate specified by the eta parameter is used as an initial step size for these adaptive learning rates.

A common approach to finding a good value for the learning rate is to perform a grid search over a range of possible values and select the one that results in the best performance on a validation set. This is what is being performed in the following code. Here, for each learning rate, the FNN is trained and its respective performance is assessed. For any step size and especially the large one, it could be possible that a minimum of the cost function is found with just a few steps if, by coincidence, the starting value is optimal. To avoid incorrectly choosing this learning rate because of a convenient starting value, the FNN is trained multiple times for each hyperparameter and the performance is then averaged.

The optimal learning rate is chosen as the value that minimizes the MSE of the validation data. This data is not used to train the model itself and thus represents an independent data set. This is plotted in blue in the graph below and hereby we find an optimal learning rate of η which is printed below the code. In green, the corresponding cost function is plotted which essentially not only incorporates the MSE but also an L2 parameter to avoid overfitting. This parameter depends on the weights which are being found with influence of the learning rate. For the most part, validation MSE and cost function are closely connected which implies a relatively small L2 parameter. For large learning rates, due to the large step size of the corrector step, an optimal value for the weights could not be found. The weights become too large and let the L2 parameter blow up which leads to a deviation of the cost function from the validation MSE.

The learning rate influences how quickly or slowly the model learns from the data. A high learning rate can result in large weight updates and rapid convergence, but may also cause the model to overshoot the optimal values of the weights and result in poor performance. This is visible on the right end of the plot below. A low learning rate can result in small weight updates and slow convergence, but may also allow the model to find better solutions by taking smaller steps in the weight space. Especially because the number of epochs is fixed in this analysis, a smaller stepsize does not necessarily provide a better performance as the network then takes too long to minimize the loss function. This is what can be seen on the left end of the plot, where the error rises with smaller learning rates. The optimum can thus be found in the middle of these extremes.

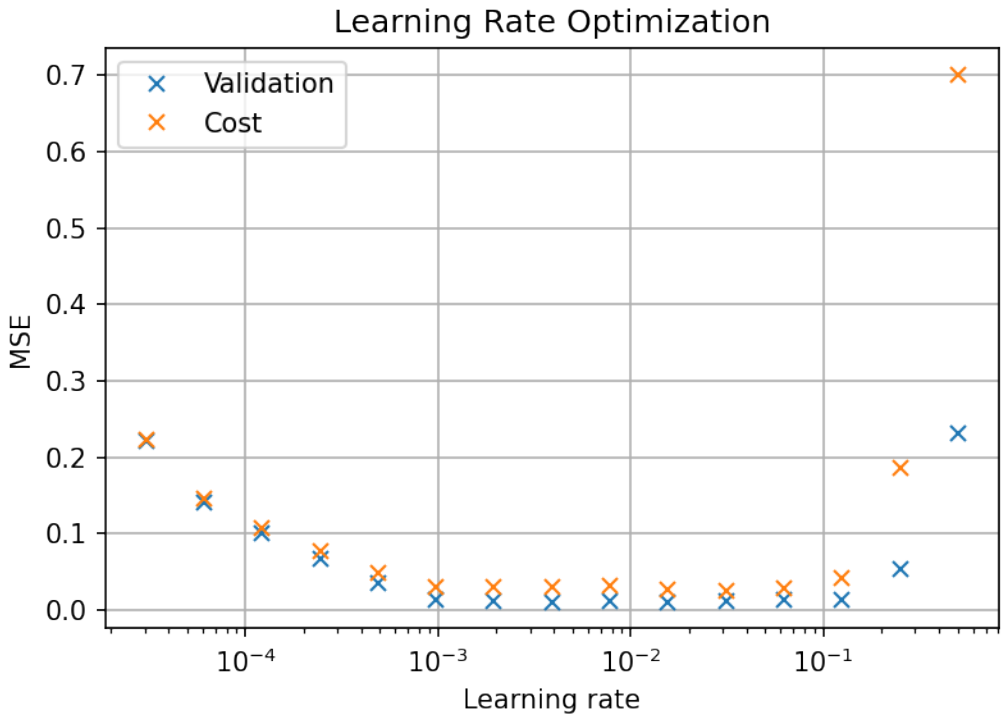
[ ]:
```
learning_rates = [0.5]
for i in range(1, 15):
    learning_rates.append(learning_rates[i-1]/2)
learning_rates
n_runs = 5

mse_values = [] # of validation data
best_mse = float('inf')
best_lr = None
```

```
cost_values = []

# loop through different learning rates
for lr in learning_rates:
    mse_sum = 0
    cost_sum = 0

    # train the model multiple times to avoid coincidentally good result
    for run in range(n_runs):
        mlp = MLP(n_hidden=100, seed=2, l2=0.001, epochs=80, eta=lr,␣
 ↪shuffle=True, minibatch_size=100)

        # train on training data with learning rate (validation data not used␣
 ↪for training)
        mlp.train(x_train, y_train, x_val, y_val)

        # test on validation data
        y_pred = mlp.predict(x_val)

        cost = mlp.eval_['cost'][-1]
        mse = mlp.eval_['valid_mse'][-1]

        mse_sum += mse
        cost_sum += cost

    mse_avg = mse_sum / n_runs
    mse_values.append(mse_avg)

    cost_avg = cost_sum / n_runs
    cost_values.append(cost_avg)

    if mse_avg < best_mse:
        best_mse = mse_avg
        best_lr = lr

print(f"The best learning rate is: {best_lr}")
```

```
80/80 | Cost: 0.22 | Train/Valid MSE.: 0.21/0.22

The best learning rate is: 0.015625
```

[ ]:
```
plt.plot(learning_rates, mse_values, "x", label="Validation")
plt.plot(learning_rates, cost_values, "x", label="Cost")
plt.xscale('log')
plt.xlabel('Learning rate')
plt.ylabel('MSE')
plt.legend()
plt.grid(True)
```

```
plt.title("Learning Rate Optimization")
plt.show()
```



4 d)

Determine the performance of the FNN on the test data. What is the accuracy of the predictions for the chosen hyper parameters? Accuracy is a metric used to evaluate the performance of classification models. It is calculated as the ratio of the number of correct predictions to the total number of predictions. However, in this case, we are working on a regression problem, where the goal is to predict a continuous value rather than a class label. In this case, accuracy is not an appropriate metric to use.

For regression problems, there are several other metrics that you can use to evaluate the performance of your model. A common one includes mean squared error (MSE) which is calculated below. This measure is very close to the MSE of the datapoint distribution to an actual sine, from which the data was created using random deviations with a gaussian distribution, which has been calculated one cell further and is very close to 0.01. This basically sets the boundary for the performance of any prediction.

[ ]:
```
# Create an instance of the MLP class with the optimized learningrate
mlp = MLP(eta=best_lr)

# Train the model
```

```
mlp.train(x_train, y_train, x_val, y_val)

# Evaluate the model on the test data
mlp.evaluate(x_test, y_test)
```

```
80/80 | Cost: 0.03 | Train/Valid MSE.: 0.01/0.01
```

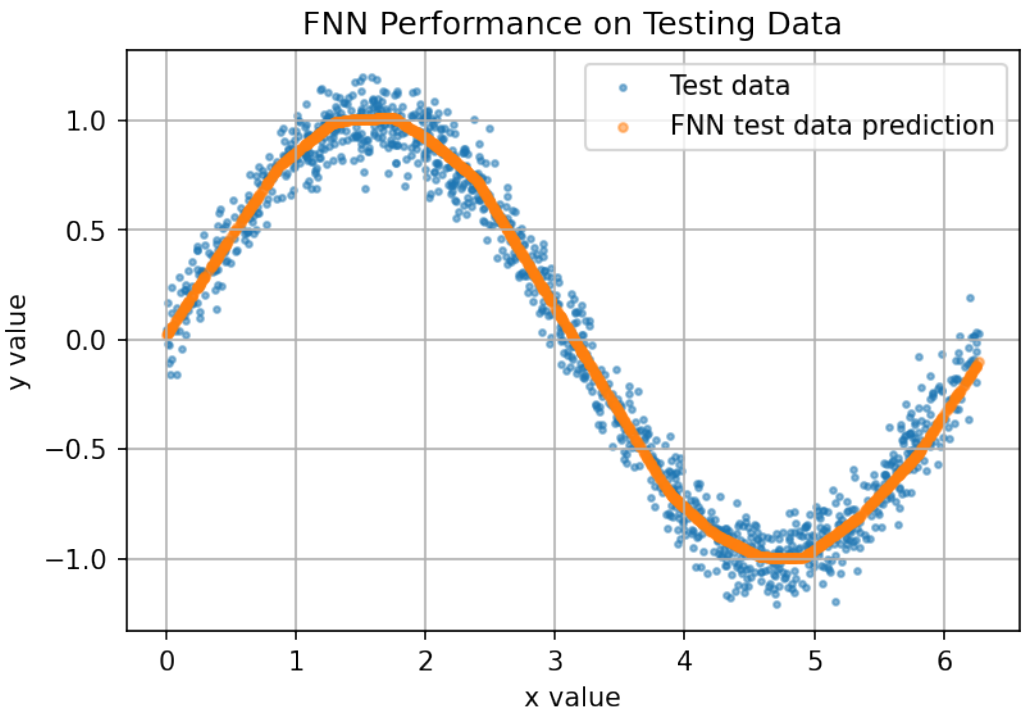
```
mean squared error:0.010805988234896077
```

[ ]:
```
# calculate MSE from sine distribution, to which random gaussian deviations␣
↪where added to create initial datasets
np.mean((np.sin(x_values) - sin_wave)**2)
```

[ ]:
```
0.009981646630062532
```

You can assess the performance of the FNN visually by looking at the graph below. The FNN predicts the shape of the sine really well but does not represent the random noise in the original test data. The FNN does not try to predict the randomness of the points which shows that the model was not overfitted.

[ ]:
```
# use FNN for regression on test data
predictions = mlp.predict(x_test)
plt.scatter(x_test, y_test, label='Test data', s=5, alpha=0.5)
plt.scatter(x_test, predictions, s=10, alpha=0.5, label="FNN test data␣
↪prediction")
plt.legend()
plt.grid(True)
plt.xlabel("x value")
plt.ylabel("y value")
plt.title("FNN Performance on Testing Data")
plt.show()
```


FNN Performance on Testing Data
Test data
FNN test data prediction
y value
x value
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6