


Appendix B Multi-Layer Perceptron Applied to the Fuel Economy Use Case

 **Note:** This appendix is also available as a colored version of PDF from this book's download website.

This appendix demonstrates how one can apply the multi-layer perceptron model to solve regression machine learning problems. We choose the *sklearn.neural_network.MLPRegressor* model and apply it to our fuel economy machine learning use case we studied in the first few chapters of this book. Although it is a very simple implementation, it actually demonstrates many aspects of a feed-forward neural network model in terms of how multiple layers are composed, how various activation functions work, how various solvers work, etc.

Check out this web link for the details of the MLP model we use for this case: http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html#sklearn.neural_network.MLPRegressor.

B.1 THE BASELINE

First, let's establish a baseline.

Figure 2.8 of the main text illustrated how a linear regressor works when applied to the fuel economy machine learning use case. I copied the script `ch02.vehicles_linear_regression_mean.py` for that use case and renamed it `vehicles_mlp_mean.py`. The changes I applied are as follows:

```
regr = MLPRegressor(hidden_layer_sizes=(60), max_iter=200, alpha=1e-4,  
                   solver='lbfgs', verbose=10, tol=1e-4, random_state=1,  
                   learning_rate_init=.1)
```

```
# Train the model using the training sets
regr.fit(first_half_x, first_half_y.ravel())
```

As you see, it's so easy to change to a different model after you have understood all the basic concepts associated with various machine learning models. Now if you look at the parameters for the `MLRegressor` model shown above, it should be obvious to you if you have studied chapters 1-9 in the main text. Here, we only need to check out how the hidden layers are supposed to be specified and what solvers are available.

As of now, we specified `hidden_layer_sizes=(60)` and `solver='lbfgs'` for the MLP model we are going to try out. In this case, the hidden layer is a single layer with 60 neurons. The hidden layers are specified in the format of (m, n, \dots) , which is a Python tuple, with each number specifying the number of neurons for that particular layer. The solver *lbfgs* is an optimizer in the family of quasi-Newton methods.

Figure B.1 shows the result obtained with executing the above MLP model. It is close to the quadratic model result shown in Figure 2.9 in the main text. The MSE, RMSE and R-squared score are 16.75, 4.09 and 0.81, respectively, comparable to the corresponding metrics of 16.51, 4.06, and 0.82 obtained for Figure 2.9. However, this simple example shows many aspects of typical neural network models. If you are interested, you can even check out how the back propagation algorithm is implemented exactly in *sklearn*.

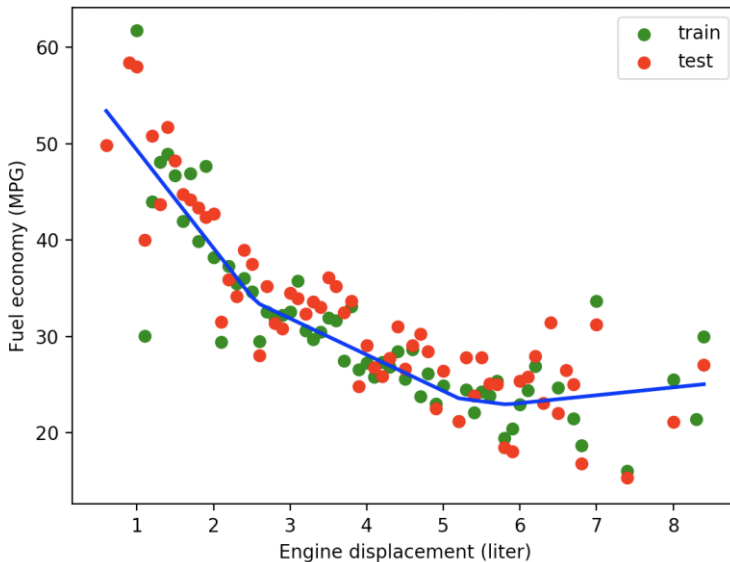


Figure B.1 The multi-layer perceptron model applied to the fuel economy machine learning use case.

B.2 THE EFFECTS OF THE NUMBER OF NEURONS

Now let's check out the effects of the number of neurons in this use case. I re-used the previous script `vehicles_mlp_mean.py` and renamed it `vehicles_mlp_mean_neurons.py`, with the following changes made:

```

colors = ['green', 'blue', 'red', 'purple', 'yellow']

for i, neuron in enumerate([10, 50, 100, 200, 400]):
    regr = MLPRegressor(hidden_layer_sizes=(neuron,), max_iter=200, alpha=1e-4,
                        solver='lbfgs', tol=1e-4, random_state=1, learning_rate_init=.1)

    # Train the model using the training sets
    regr.fit(first_half_x, first_half_y.ravel())

    # Make predictions using the testing set
    second_half_y_pred = regr.predict(second_half_x)

    print("\tModel parameters: ", regr.get_params(deep=False))

    # mean squared error
    mse = mean_squared_error(second_half_y, second_half_y_pred)
    rmse = np.sqrt(mse)
    print("Mean squared error: %.2f" % mse)
    print("Root mean squared error: %.2f" % rmse)
    # Explained variance score: 1 is perfect prediction
    print('R-squared score: %.2f' % r2_score(second_half_y, second_half_y_pred))

    plt.plot(second_half_x, second_half_y_pred, linewidth=2, color=colors[i],
             label="neurons = %s" % str(neuron))

    plt.scatter(first_half_x, first_half_y, color='green', label = "training set")
    plt.scatter(second_half_x, second_half_y, color='red', label = "testing set")
    plt.xlabel ("Engine displacement (liter)")
    plt.ylabel ("Fuel economy (MPG)")
    plt.legend(loc='upper right')
    plt.show()

```

The above script loops through five settings for the number of neurons equal to 10, 50, 100, 200, and 400, respectively. The effects of the number of neurons are shown in Figure B.2. As is seen, fittings are differentiable from 10 to 50 and 100 neurons, but once surpassing 100 neurons, results for 100, 200, and 400 neurons are not quite distinguishable. This means that 100 neurons are sufficient for this single-layer perceptron model.

Next, we explore how the results would change with the number of layers.

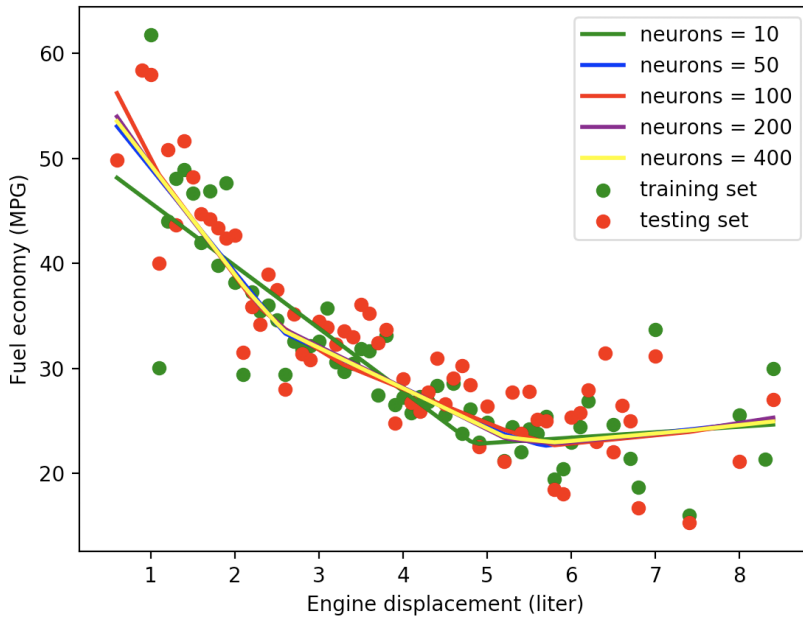


Figure B.2 Effects of varying number of neurons with a single layer perceptron.

B.3 THE EFFECTS OF THE NUMBER OF LAYERS

Now let's check out the effects of the number of layers for the MLP model we use in this use case. I reused the previous script `vehicles_mlp_mean_neurons.py` and renamed it `vehicles_mlp_mean_layers.py`, with the following changes made:

```

layer = (100,)
colors = ['green', 'blue', 'red', 'purple']
for i in range(0, 4):
    print("layer = ", layer, "\n")
    regr = MLPRegressor(hidden_layer_sizes=layer, max_iter=200, alpha=1e-4,
                        solver='lbfgs', tol=1e-4, random_state=1, learning_rate_init=.1)

# Train the model using the training sets
    regr.fit(first_half_x, first_half_y.ravel())

    ....

    plt.plot(second_half_x, second_half_y_pred, linewidth=2, color=colors[i],
             label="mlp_layers %s" % str(layer))

    layer = layer + layer

```

.....

As is seen, we started with a single layer with 100 neurons. Then we doubled the number of layers from 1 to 2, 4, and 8, for a total of four runs. Figure B.3 shows the results. We start to see overfitting with eight layers indicated by the purple curve with the increasing model complexity. However, the MSE, RMSE and R squared score did not change much, as that overfitting occurred at the lower end only.

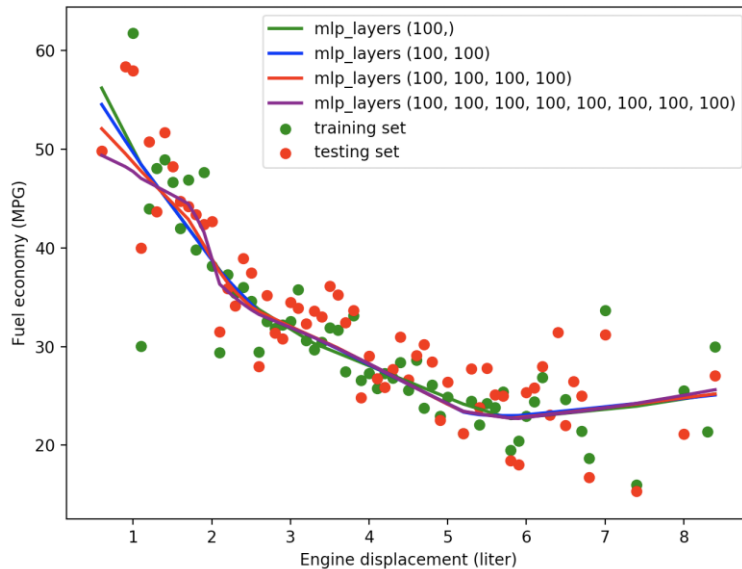


Figure B.3 Effects of varying number of layers with a multi-layer perceptron.

B.4 THE EFFECTS THE ACTIVATION FUNCTIONS

Now let's check out the effects of the activation functions for the MLP model used in this use case. I re-used the previous script `vehicles_mlp_mean_layerss.py` and renamed it `vehicles_mlp_mean_activations.py`, with the following changes made:

```

activations = ['identity', 'logistic', 'tanh', 'relu']
for activation in activations:
    regr = MLPRegressor(hidden_layer_sizes=(100,), activation=activation,
                        max_iter=200, alpha=1e-4, solver='lbfgs', verbose=10, tol=1e-4,
                        random_state=1, learning_rate_init=.1)

```

.....

As shown in Figure B.4, the *identity* activation function is equivalent to a linear model, while the *tanh* and *logistic* activations performed poorly against the ReLU activation function. This is why the ReLU activation function has been used unanimously for various ANNs, such as CNNs, RNNs and AEs introduced in Chapters 10, 11 and 12, respectively.

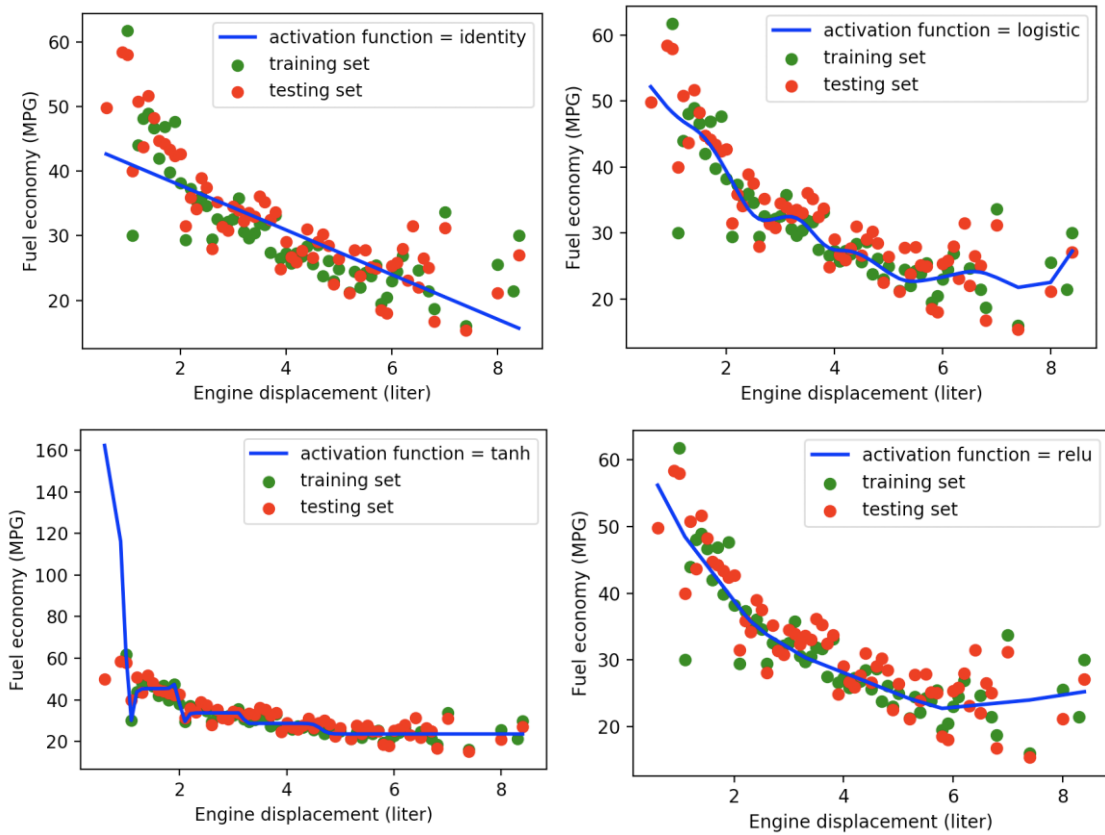


Figure B.4 Single-layer perceptron with different activation functions.

B.5 THE EFFECTS THE SOLVERS

Now let's check out the effects of the solvers for the MLP model we choose in this use case. I re-used the previous script `vehicles_mlp_mean_activations.py` and renamed it `vehicles_mlp_mean_solvers.py`, with the following changes made:

```
solvers = ['lbfgs', 'sgd', 'adam']
for solver in solvers:
    regr = MLPRegressor(hidden_layer_sizes=(100,), max_iter=200, alpha=1e-4,
                        solver=solver, verbose=10, tol=1e-4,
                        random_state=1, learning_rate_init=.1)
    .....
```

Fig B.5 shows the result. It is seen that the `sgd` and `adam` solvers are totally off the track, compared with the `lbfgs` solver. It should not be a surprise as we already learnt in Part I for the conventional models that the `lbfgs` solver is more stable in general.

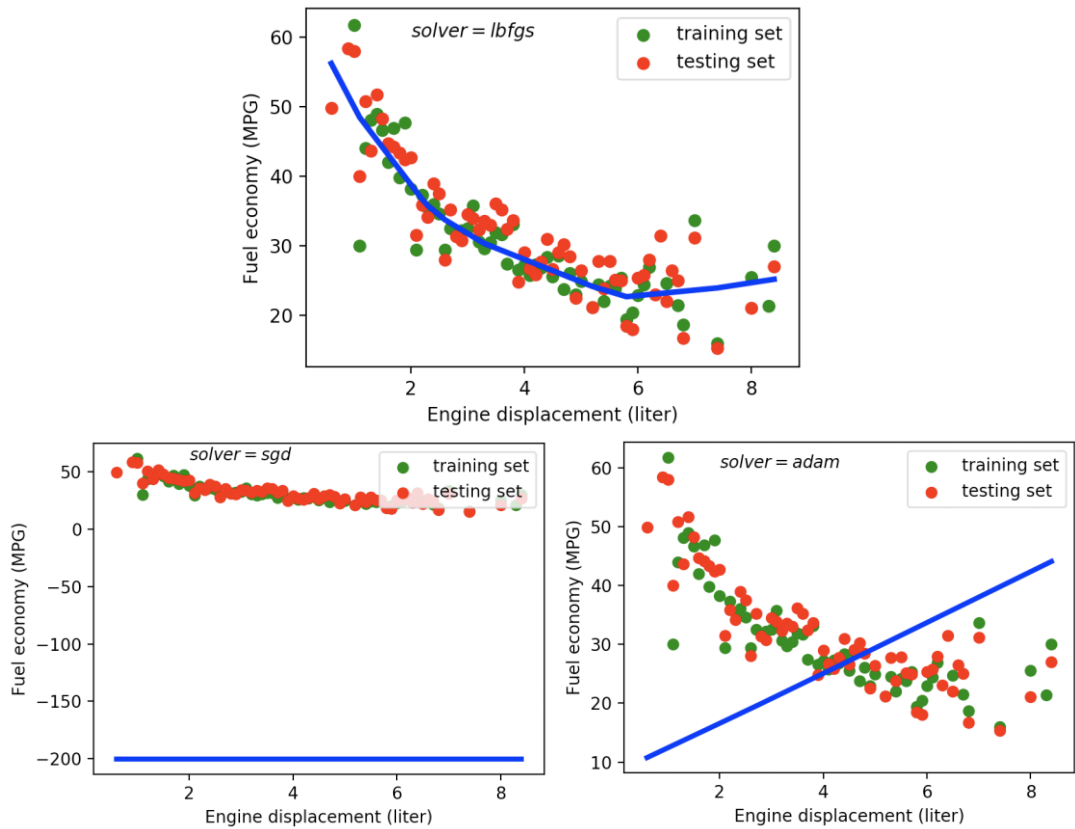


Figure B.5 Single-layer perceptron with different solvers.

B.6 THE MLP MODEL FOR CLASSIFICATION TASKS

The MLP model can also be used for classification tasks. If you are interested, check out an example at http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html. Given what you have learnt so far, you should be able to understand that example easily.