NeuralNetworks

September 25, 2025

The most basic neural network is called a **multi-layered perception**. It is a neural network where the neurons in each layer are connected to **all** the neurons in the next layer. For this reason we call it a **dense** network.

In this notebook we use `numpy` to manually create and train a neural network. We do this mostly so we can build some intuition around what happens behind the scene when we train a neural network.

[ ]:
```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

The data we use is manually created. We want to have a very small data set so that we can look at intermediate results as we build our neural network.

[ ]:
```
X1 = np.array(np.arange(0.1, 0.7, 0.1))
X1 = np.exp(X1 * 1.1 + 0.75)
X2 = np.array(np.arange(0.6, 1.2, 0.1))
X2 = np.exp(X2 * 0.4 + 0.75)
X3 = np.random.random(6)
X3 = np.exp(X3 * 0.4 + 0.75)

X_train = np.array([X1, X2, X3]).T
y_train = (X_train[:,:1] > 3).all(axis = 1).reshape(6, 1)

print(np.hstack([X_train, y_train]))
del X1, X2, X3
```

0.1 Using a logistic regression

Before we train a neural network, it might be worthwhile asking what we would do if we had to solve this using the tools we already have at our disposal. Since our target is binary, using a `LogisticRegression` is one easy option. So let's try it:

[ ]:
```
logmod = LogisticRegression()
logmod.fit(X_train, y_train.ravel())
y_hat = logmod.predict(X_train)
```

Let's evaluate our model. Usually we would evaluate the model on the training data. We'll worry about test data later. For now that's besides the point.

[ ]:
```
confusion_matrix(y_train, y_hat)
```

So with logistic regression, we can train a model that seems to quickly find the decision boundary. How does logistic regression make its prediction? It uses the following formula to get raw predictions .

$$\text{raw\_predictions} = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3$$

In previous lectures, we referred to b_0, b_1 and b_2 as the model's **parameters**: b_0 is called the **intercept** and b_1, b_2 and b_3 are called **coefficients**. These raw predictions represent our confidence about how likely it is that any row of the data would belong to the positive class. But the scale of these raw predictions are somewhat arbitrary.

[ ]:
```
print("Model intercept (bias): ")
print(logmod.intercept_)
print("Model coefficients (weights): ")
print(logmod.coef_.T)

pred = logmod.intercept_ + np.dot(X_train, logmod.coef_.T)
pred
```

Let's see an example of an **activation function**. Here we use the **sigmoid** activation function, also called the **logistic** activation function, given by $\sigma(z) = \frac{1}{1+e^{-z}}$. It forces the activations to be between 0 and 1. Before passing the input to this function, use `np.clip` to trim it between -500 and 500.

[ ]:
```
def sigmoid(x):
    x = np.clip(x, -500, 500)
    return 1/(1 + np.exp(-x))
```

We can take the raw predictions and pass them to a **sigmoid** function and get predictions that are rescaled to be between 0 and 1. We interpret these scaled predictions as the probability that a given row belongs to the positive class.

[ ]:
```
np.hstack([sigmoid(pred), y_train])
```

By the way, the above is what we obtain when we run the `predict_proba` method of the trained model.

[ ]:
```
logmod.predict_proba(X_train) # the second column shows the probability of Y = 1
```

Notice that when the prediction is below 0.50 the labels are 0 and otherwise the labels are 1. The reason we started with `LogisticRegression` is because the way that it trains is very similar to a neural network. In fact, a logistic regression model is a neural network with **no hidden layer**. So let's now manually create our neural network and see how we can get a result similar to what `LogisticRegression` obtained above.

0.2 Neural network

Let's return to our prediction equation:

$$\text{raw\_predictions} = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3$$

In neural networks, we prefer to use the word **bias** for the intercept and **weights** for the coefficients. We saw how logistic regression found its parameters. Now we want to see how a neural network finds its parameters? It starts with some random values for them. We call this **random initialization**. We usually generate numbers that are random but close to 0.

[ ]:
```
def init_parameters(dim1, dim2 = 1, std = 1e-1, random = True):
    if(random):
        return(np.random.random([dim1, dim2]) * std)
    else:
        return(np.zeros([dim1, dim2]))
```

Once we have values for the parameters, we can now run a **forward pass**, which ultimately ends in **predictions**. Of course, because we randomly initialized our parameters, the first time around the predictions are as good as coin tosses.

Note that our forward pass consists of a matrix multiplication, for which we use `np.dot`. The forward pass takes the input data and multiplies it by weights and adds the bias, the result of which is called a **weighted sum**, called `Z1` below. It then applies the **activation function** to the weighted sum, we get the **activations**, called `A1` here.

In this example, we don't have any hidden layers, so our forward pass will take us directly from the data to the predictions. But if we had hidden layers, we would run this same calculation once for each hidden layer, finally finishing with the prediction.

[ ]:
```
def forward(W1, bias, X):
    Z1 = np.dot(X, W1) + bias
    A1 = sigmoid(Z1)
    return(A1)
```

Let's test our function to make sure it works.

[ ]:
```
_, input_cols = X_train.shape
_, output_cols = y_train.shape

weights = init_parameters(input_cols, output_cols)
bias = init_parameters(output_cols)

print("Checking dimensions: {} * {} + {}".format(X_train.shape, weights.shape,␣
 ↪bias.shape))

pred = forward(weights, bias, X_train)
pred
```

With the forward pass, we now have a prediction. Our next question is how can we improve our prediction? The answer is that we need to calculate our error (called **loss**) and from that derive the **derivative of loss w.r.t. weights and biases**. In multivariate calculus, this derivative is called the **gradient**.

In previous lectures, we learned that for classification model, we can measure the error by looking at **accuracy** (or precision and recall for imbalanced data). However, as it turns out these metrics are not going to work well here, because in order to get derivatives in calculus we need **continuous functions**, and accuracy, precision or recall are not continuous functions of our weights and biases. Another problem is that these metrics are obtained **after** we define our threshold, and can change if we change our threshold. So we need something else.

One loss function that works well with classification is the **cross-entropy loss**. For binary classification, cross-entropy for the ith data point is defined as $y_i * log(\hat{y}_i) + (1 - y_i) * log(1 - \hat{y}_i)$, where y_i is our binary target, and $\hat{y}_i$ is the prediction (activation at the output layer) at row i. Cross-entropy for the whole data is just the average of the cross-entropies at each row.

While we don't show the derivation here, once we define our loss function, we can get the derivative of loss w.r.t. the activations `A1`, and then (using the chain rule) get the derivative of loss w.r.t. to `Z1`, and finally w.r.t. weights and biases.

[ ]:
```
def backward(A1, W1, bias, X, Y):

    m = np.shape(X)[0] # used the calculate the cost by the number of inputs -1/m

    loss = Y * np.log(A1) + (1 - Y)*np.log(1 - A1)            # loss at each row
    cost = (-1/m) * np.sum(loss)                              # loss across all rows
    dZ1 = A1 - Y                                              # derivative of loss wrt Z1
    dW1 = (1/m) * np.dot(X.T, dZ1)                            # derivative of loss wrt weights
    dBias = (1/m) * np.sum(dZ1, axis = 0, keepdims = True) # derivative of loss wrt bias

    grads = {"dW1": dW1, "dB1": dBias}                        # updated weights and biases

    return(grads, cost)
```

Let's once again test the output to make sure it's working.

[ ]:
```
gradients, _ = backward(pred, weights, bias, X_train, y_train)
gradients
```

We now have all we need to start running our optimization routine: a simple implementation of **gradient descent**. This consists of iteratively running forward propagation to get predictions, the backpropagation to get the gradient of loss w.r.t. weights and biases, and finally moving weights

and biases in the direction of their gradient. For the latter, we control the size of the step using a constant we call the **learning rate**. As we do this, we record loss at each iteration so that we can plot it and make sure that loss is decreasing at the end of each iteration.

[ ]:
```
def run_grad_desc(num_epochs, learning_rate, X, Y):

    m, input_cols = X.shape

    W1 = init_parameters(input_cols, output_cols)
    B1 = init_parameters(output_cols)

    loss_array = np.ones([num_epochs])*np.nan      # place-holder of keeping␣
↪track of loss

    for i in np.arange(num_epochs):
        A1 = forward(W1, B1, X)                    # get activations in final␣
↪layer
        grads, cost = backward(A1, W1, B1, X, Y)  # get gradient and the cost␣
↪from BP

        W1 = W1 - learning_rate*grads["dW1"]       # update weights
        B1 = B1 - learning_rate*grads["dB1"]       # update bias

        loss_array[i] = cost                       # record loss for current␣
↪iteration

        parameter = {"W1": W1, "B1": B1}           # record parameters for␣
↪current iteration

    return(parameter, loss_array)
```

That's it. Let's now run our gradient descent function for 1000 iterations.

[ ]:
```
num_epochs = 500
learning_rate = 0.01
params, loss_array = run_grad_desc(num_epochs, learning_rate, X_train, y_train)
```

After letting the network train for many iterations, these are the final parameters we have.

[ ]:
```
print(params['B1'][0])
print(params['W1'].ravel())
```

And these are the parameters we got when we trained a logistic regression model.

[ ]:
```
print(logmod.intercept_)
print(logmod.coef_)
```

We can see that the parameters don't necessarily look similar. There can be a lot of reasons for

that. Because our data is close to linearly separable, there are a lot of possible solutions. There could also be differences between the `sklearn` logistic regression and our implementation of neural networks. So instead of comparing the parameters, let's compare the predictions: we can put the predictions we get from using the parameters for the neural network and logistic regression side by side.

[ ]:
```
Y_pred_nn = params['B1'] + np.dot(X_train, params['W1'])
y_pred_logit = logmod.intercept_ + np.dot(X_train, logmod.coef_.T)

np.hstack([y_pred_logit, Y_pred_nn, y_train])
```

Recall that these are raw predictions. So it might be best to pass these to a sigmoid function to turn them into probabilities.

[ ]:
```
np.hstack([sigmoid(y_pred_logit), sigmoid(Y_pred_nn), y_train])
```

We can see that in either case if we use 0.50 as the cut-off both models predict correctly.

0.2.1 Exercise

- Run the neural network for 10K iterations instead of 1000 and look at the predictions.
- Run the neural network for 100K iterations instead of 1000 and look at the predictions.
- Do you see a trend?
- Return to where you run `run_grad_desc` and prior to running run the following code: `y_train = np.hstack([y_train, ~y_train])`. Careful! This will break the earlier logistic regression code. Train the network and look at the results that follow. Can you explain what happened? This result has important consequences for our earlier claim that you can do multi-class classification with neural networks using a single model (without resorting to **one-vs-rest** or **one-vs-one** models).

0.2.2 End of exercise

Let's show how loss drops iteration over iteration that we train this. This means that in a real-world scenario, if we let training continue indefinitely, eventually we will reach a point where we begin over-fitting to the training data. So it's important to have test data set aside that we use for knowing when that happens so we can stop training. This is called **early stopping**.

[ ]:
```
sns.lineplot(data = loss_array[::10]); # we only plot every 50th point so plot␣
↪renders fast
```

So we saw how a neural network works. Of course a real neural network would have at least one hidden layer, but hidden layers only add the amount of computation we have to do. The principle stays the same.

[ ]:
```
Load the wine.csv data and prepare the data for analysis: Split the data into␣
↪training and testing and normalize the features.
```

[1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import MinMaxScaler

# Load the data from the CSV file
df = pd.read_csv('wine.csv')

# Separate features (X) and target (y)
X = df.drop('Class', axis=1)
y = df['Class']

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Normalize the features using MinMaxScaler
scaler = MinMaxScaler()
X_train_normalized = scaler.fit_transform(X_train)
X_test_normalized = scaler.transform(X_test)

# Convert the normalized arrays back to DataFrames for better readability and␣
 ↪saving to CSV
X_train_normalized = pd.DataFrame(X_train_normalized, columns=X_train.columns)
X_test_normalized = pd.DataFrame(X_test_normalized, columns=X_test.columns)

# Save the processed data to CSV files
X_train_normalized.to_csv('X_train_normalized.csv', index=False)
X_test_normalized.to_csv('X_test_normalized.csv', index=False)
y_train.to_csv('y_train.csv', index=False)
y_test.to_csv('y_test.csv', index=False)

# Display the first few rows of the normalized training and testing sets
print('Normalized Training Data:')
print(X_train_normalized.head())
print('\nNormalized Testing Data:')
print(X_test_normalized.head())
```

```
Normalized Training Data:
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0       0.237288          0.106667     0.284553        0.108896   0.036545
1       0.381356          0.133333     0.390244        0.023006   0.139535
2       0.330508          0.423333     0.008130        0.023006   0.091362
3       0.118644          0.193333     0.268293        0.009202   0.031561
4       0.237288          0.120000     0.455285        0.226994   0.073090

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0             0.240550              0.381657  0.128976  0.354331   0.084270
1             0.034364              0.017751  0.134374  0.409449   0.224719
2             0.206186              0.109467  0.127241  0.527559   0.196629
```

```
3             0.082474              0.221893  0.059572  0.496063   0.089888
4             0.213058              0.399408  0.211876  0.291339   0.151685

    alcohol   quality
0  0.403226  0.333333
1  0.709677  0.666667
2  0.612903  0.500000
3  0.596774  0.500000
4  0.209677  0.333333

Normalized Testing Data:
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0       0.271186          0.060000     0.601626        0.187117   0.059801
1       0.330508          0.373333     0.170732        0.024540   0.112957
2       0.254237          0.206667     0.276423        0.104294   0.018272
3       0.211864          0.133333     0.382114        0.162577   0.051495
4       0.305085          0.180000     0.162602        0.203988   0.074751

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0             0.158076              0.355030  0.136688  0.409449   0.089888
1             0.213058              0.375740  0.163678  0.417323   0.129213
2             0.254296              0.375740  0.096588  0.346457   0.123596
3             0.412371              0.523669  0.169848  0.299213   0.162921
4             0.426117              0.659763  0.226913  0.291339   0.157303

    alcohol   quality
0  0.677419  0.833333
1  0.306452  0.333333
2  0.645161  0.666667
3  0.241935  0.500000
4  0.145161  0.500000
```

[ ]:
```
Train a logistic regression classifier to predict the type of wine (red vs.␣
↪white). Report the accuracy of the model.
```

[2]:
```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the pre-processed data from the CSV files
X_train_normalized = pd.read_csv('X_train_normalized.csv')
X_test_normalized = pd.read_csv('X_test_normalized.csv')
y_train = pd.read_csv('y_train.csv')
y_test = pd.read_csv('y_test.csv')

# Initialize the Logistic Regression classifier
log_reg_model = LogisticRegression(solver='liblinear', random_state=42)
```

```
# Train the model using the normalized training data
log_reg_model.fit(X_train_normalized, y_train.values.ravel())

# Make predictions on the normalized testing data
y_pred = log_reg_model.predict(X_test_normalized)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy
print(f"Model Accuracy: {accuracy:.4f}")
```

```
Model Accuracy: 0.9823
```

[ ]:
```
Train a multi-layer feed-forward neural network to predict the type of wine.␣
↪Your network should have one hidden layer. You are free to choose how many␣
↪neurons you want in the hidden layer.
```

[3]:
```
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Load the pre-processed data
X_train_normalized = pd.read_csv('X_train_normalized.csv')
X_test_normalized = pd.read_csv('X_test_normalized.csv')
y_train = pd.read_csv('y_train.csv')
y_test = pd.read_csv('y_test.csv')

# Initialize the Multi-layer Perceptron (MLP) Classifier with two hidden layers
# For example, two layers with 100 and 50 neurons respectively
mlp_classifier = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500,␣
↪random_state=42)

# Train the model
mlp_classifier.fit(X_train_normalized, y_train.values.ravel())

# Make predictions on the testing data
y_pred = mlp_classifier.predict(X_test_normalized)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy
print(f"Neural Network Model Accuracy: {accuracy:.4f}")
```

```
Neural Network Model Accuracy: 0.9954
```

[ ]:
```
Tune your neural network by trying different values for the learning rate and␣
↪the number of neurons in the hidden layer.
```

[3]:
```
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

# Load the pre-processed data
X_train_normalized = pd.read_csv('X_train_normalized.csv')
X_test_normalized = pd.read_csv('X_test_normalized.csv')
y_train = pd.read_csv('y_train.csv').values.ravel()
y_test = pd.read_csv('y_test.csv').values.ravel()

# Define the model to be tuned
mlp = MLPClassifier(max_iter=500, random_state=42)

# Define the hyperparameters to search over
param_grid = {
    'hidden_layer_sizes': [(50, 25), (100, 50), (100, 100)],
    'learning_rate_init': [0.001, 0.01, 0.1]
}

# Initialize GridSearchCV with the model and parameter grid
grid_search = GridSearchCV(mlp, param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Perform the grid search on the training data
grid_search.fit(X_train_normalized, y_train)

# Get the best parameters and the best estimator (model)
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred_best = best_model.predict(X_test_normalized)

# Calculate the accuracy of the best model on the test set
best_model_accuracy = accuracy_score(y_test, y_pred_best)

# Print the results
print("Best Hyperparameters:", best_params)
print(f"Accuracy of the best model on the test set: {best_model_accuracy:.4f}")
```

```
Best Hyperparameters: {'hidden_layer_sizes': (50, 25), 'learning_rate_init':
0.001}
Accuracy of the best model on the test set: 0.9923
```

[ ]:
```
Report the accuracy of the best model you obtained in the previous step.
```

[ ]:
```
hidden layer sizes: (50, 25)
learning rate init: 0.001

# The accuracy of the tuned neural network model on the test set is 99.23%.
# This result is only a minor change from the initial model's accuracy,
# showing that the default parameters were already quite effective for this␣
 ↪dataset.
```

[ ]:
```

```