Chapter 10: Programming Exercises

Exercise: Binary Classification with a MLP

In this exercise we are going to classify (preprocessed) movie reviews as either positive or negative. We will use the built-in IMDB Keras dataset for this.

Step 1: Load the Data

- Use `tf.keras.datasets.imdb.load_data()` to easily download the IMDB dataset.
  - Limit the vocabulary to the 10000 most frequent words, by using the `num_words` argument.
  - See this page for information on how to use this function.
- A possible way of calling this function is:

```
(train_data, train_labels), (test_data, test_labels) = YOUR CODE HERE
```

Note: check that both `train_data` and `test_data` contain 25000 examples.

Print the features of the first training example. You should see something like:

```
[1, 14, 22, 16, ..., 178, 32]
```

Hence, each example is a lists of integers.

You can use the following function to convert a list of integers back into English text:

```
def convert_to_english(list_of_integers):
    word_index = tf.keras.datasets.imdb.get_word_index()
    # reverse the word index
    reverse_word_index = {idx : word for (word, idx) in word_index.items() }
    # map each integer to a word and join all words together
    # Index 0, 1 and 2 are reserved for 'padding', 'start of sequence', and 'unknown'
    return " ".join(reverse_word_index.get(idx - 3, "?") for idx in list_of_integers)
```

Step 2: Convert the Data

It is not very convenient for a neural network to ingest list of integers.

Instead, we will use multi-hot encoding to convert the lists of integers into vectors of 0s and 1s.

Note: multi-hot encoding does lose some information: - The order of the words is lost. - We also lose information about how often a word occurs in a review. We will later see better ways of encoding text.

Write a function `convert_to_multi_hot(sequences, dimension)` that takes a numpy array of sequences (lists of integers) and returns a numpy array of shape `(len(sequences), dimension)` where each row is a multi-hot encoded

vector of length `dimension`. You can write this function using "ordinary" loops. Later, we will see how you can make this type of function more efficient.

```
def convert_to_multi_hot(sequences, dimension):
    output = np.zeros(shape=(sequences.shape[0], dimension), dtype=np.float32)

    # YOUR CODE HERE

    return output
```

Example usage:

```
convert_to_multi_hot(np.array([[1,2,3,1],[0,4,5,4]]), dimension=6)
```

should yield

```
array([[0., 1., 1., 1., 0., 0.],
       [1., 0., 0., 0., 1., 1.]], dtype=float32)
```

Next, convert the training and test data to multi-hot encoded vectors. Use a dimension of 10000.

```
X_train = convert_to_multi_hot(train_data, dimension=10_000)
X_test = convert_to_multi_hot(test_data, dimension=10_000)
# Rename the labels
y_train = train_labels
y_test = test_labels
```

We will also need validation data. We will use the first 10,000 examples from the test data as our validation data. The last 15,000 examples will be used as our test data. Write code to create the validation and test data.

```
X_val = YOUR CODE HERE
y_val = YOUR CODE HERE
X_test = YOUR CODE HERE
y_test = YOUR CODE HERE
```

Step 3: Build a Model

Build a model using the Sequential API. Use the following architecture:

- A `Dense` layer with 16 units and the `relu` activation function.
- A `Dense` layer with 16 units and the `relu` activation function.
- A `Dense` output layer.
  - How many units should this layer have?
  - What is the most appropriate activation function for this layer, given that we are doing binary classification?

Write a function `get_model()` that returns this model. Ask for the `summary` of the returned model. You should see something like:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                160016

 dense_1 (Dense)             (None, 16)                272

 dense_2 (Dense)             (None, 1)                 17

=================================================================
Total params: 160,305
Trainable params: 160,305
Non-trainable params: 0
_________________________________________________________________
```

Step 4: Compile the Model

Next, compile the model.

- Use `rmsprop` as the optimizer with all the default parameter settings.
- Specify the correct loss for a binary classification problem.
- Track the `accuracy` metric.

Step 5: Train the Model

- Train the model.
  - Train it for 20 epochs.
  - Use a batch size of 512.
  - Be sure to use the validation data.
- Use the `history` object returned by the `fit` method to plot the learning curves. You can use the code given below to plot the learning curves.

```
def plot_learning_curves(history):
    plt.figure(figsize=(8, 5))
    for key, style in zip(history.history, ["r-o", "r-*", "b-o", "b-*"]):
        epochs = np.array(history.epoch)
        plt.plot(epochs + 1, history.history[key], style, label=key)
    plt.xlabel("Epoch")
    plt.axis([1, len(history.history['loss']), 0., 1])
    plt.legend(loc="lower left")
    plt.grid()
```

You should notice that the model is overfitting.

Initialize and compile the model again, but train for fewer epochs, so that the model does not overfit.

Step 6: Evaluate the Model

- Use the `evaluate` method to evaluate the model on the test set.
  - What is the performance of the model on the test set?

Note: You should get around 89% accuracy on the test set.

Finally, we are going to find the reviews that the model is most **confidently wrong** about.

E.g. suppose we have a positive review, e.g. the true label is 1, then we would expect the model to output a value close to 1 as its prediction for this review. The formula for the binary crossentropy is

$$-\big(y \times \ln(\hat{y}) + (1-y) \times \ln(1-\hat{y})\big)$$

where y is the true label (a 0 or a 1) and $\hat{y}$ is the predicted label (a value between 0 and 1). E.g. if $y = 1$ and $\hat{y} = 0.95$, then the loss is

$$-\big(1 \times \ln(0.95) + (1-1) \times \ln(1-0.95)\big) = -\ln(0.95) = 0.0513$$

However, if $y = 1$ and $\hat{y} = 0.05$, then the loss is

$$-\big(1 \times \ln(0.05) + (1-1) \times \ln(1-0.05)\big) = -\ln(0.05) = 2.9957$$

We can thus find the reviews that the model is most confidently wrong about by looking for the reviews with the highest losses.

Complete the following Python function:

```python
def find_confidently_wrong(y_true, y_pred, top=10):
    """
    y_true: the true labels (0/1). Shape (n, 1)
    y_pred: the predictions (floats). Shape(n, 1)

    Returns: list of indices, such that these indices have the highest loss
    (and they are actually misclassified)
    """
    assert len(y_true.shape) == len(y_pred.shape) == 2, "Rank should be 2"
    assert y_true.shape[0] == y_pred.shape[0], "Not the same length"
    assert y_true.shape[1] == y_pred.shape[1] == 1, "Second dimension should be 1"

    bce = tf.keras.losses.BinaryCrossentropy(reduction=tf.keras.losses.Reduction.NONE)

    # YOUR CODE HERE
```

See this page for more information on the `BinaryCrossentropy` loss function.

You can then use the following code to print the top 10 reviews that the model is most confidently wrong about:

```
import pprint

y_test_true = y_test.reshape(-1,1)
for idx in find_confidently_wrong(y_test_true, y_test_pred, top=10):
    actual_sentiment = 'POSITIVE' if y_test_true[idx][0] == 1 else 'NEGATIVE'
    predicted_sentiment = 'POSITIVE' if y_test_pred[idx][0] > 0.5 else 'NEGATIVE'
    # Add 10_000 because we used the first 10_000 elements as validation data
    pprint.pprint(convert_to_english(test_data[10_000 + idx]), width=80)
    print(f"is a {actual_sentiment} review but is was classified as {predicted_sentiment}.")
    print("**********************************")
```

Exercise: MNIST with a Deep MLP

Note: this exercise is based on exercise 10 from chapter 10 of the book "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" by Aurélien Géron. However, it contains more detailed instructions.

Step 1: Load and Preprocess the MNIST dataset

- Use `tf.keras.datasets.mnist_load_data()` to easily download the MNIST dataset.
- Use the last 10,000 images of the training set as your validation set.
  - This will give you 50,000 training images, 10,000 validation images, and 10,000 test images.
- The pixel values are stored as integers (from 0 to 255). Convert them to floats between 0.0 and 1.0.
- Reshape the tensors so that they have rank 2 instead of rank 3.

Step 2: Build a MLP

Use the `Sequential` API to build to build a model with two hidden layers and one output layer. Use the ReLU activation function for the hidden layers (with 300 and 100 units each) and the appropriate activation function and number of units for the output layer, given that we are doing multiclass classification. Create a function `get_model()` that returns this model. You should see something like this:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 300)               235500

 dense_1 (Dense)             (None, 100)               30100

 dense_2 (Dense)             (None, 10)                1010
```

```
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Step 3: Compile the model

- Create an `SGD` optimizer with a learning rate of 1e-2.
- `compile` the model.
  - Use the appropriate loss function for multiclass classification, also taking into account that the labels are not one-hot encoded.
  - Track the accuracy metric.

Step 4: Train the Model

- Use a batch size of 256.
- Train the model for a *large* number of epochs but use the `EarlyStoppingCallback` to stop training when the validation accuracy has not improved for 5 epochs.
- Also use the `ModelCheckpoint` callback to save the model with the best validation loss.
- Use the `TensorBoard` callback to log the training process.

Step 5: Evaluate the Model

- Use the `evaluate` method to evaluate the model on the train, validation and test set.

Step 6: Try Different Learning Rates

- Try different learning rates (e.g. `[0.01, 0.05, 0.1, 0.5]`) and batch sizes (e.g. `[64, 128, 256, 512]`) and see how they affect the performance of the model.
- Check the learning curves in TensorBoard to inspect how the training process went.
- Save all the different models and try to find the best one. Evaluate its performance on the test set.