Open in Colab

T81-558: Applications of Deep Neural Networks

Module 11: Natural Language Processing and Speech Recognition

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 11 Material

- Part 11.1: Introduction to Hugging Face [Video] [Notebook]
- Part 11.2: Hugging Face Tokenizers [Video] [Notebook]
- Part 11.3: Hugging Face Datasets [Video] [Notebook]
- Part 11.4: Training Hugging Face Models [Video] [Notebook]
- **Part 11.5: What are Embedding Layers in Keras** [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [ ]: try:
            %tensorflow_version 2.x
            COLAB = True
            print("Note: using Google CoLab")
        except:
            print("Note: not using Google CoLab")
            COLAB = False
```

```
Note: using Google CoLab
```

Part 11.5: What are Embedding Layers in Keras

Embedding Layers are a handy feature of Keras that allows the program to automatically insert additional information into the data flow of your neural network. In the previous section, you saw that Word2Vec could expand words to a 300 dimension vector. An embedding layer would automatically allow you to insert these 300-dimension vectors in the place of word indexes.

Programmers often use embedding layers with Natural Language Processing (NLP); however, you can use these layers when you wish to insert a lengthier vector in an index value place. In some ways, you can think of an embedding layer as dimension expansion. However, the hope is that these additional dimensions provide more information to the model and provide a better score.

Simple Embedding Layer Example

- **input_dim** = How large is the vocabulary? How many categories are you encoding? This parameter is the number of items in your "lookup table."
- **output_dim** = How many numbers in the vector you wish to return.
- **input_length** = How many items are in the input feature vector that you need to transform?

Now we create a neural network with a vocabulary size of 10, which will reduce those values between 0-9 to 4 number vectors. This neural network does nothing more than passing the embedding on to the output. But it does let us see what the embedding is doing. Each feature vector coming in will have two such features.

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding
import numpy as np

model = Sequential()
embedding_layer = Embedding(input_dim=10, output_dim=4, input_length=2)
model.add(embedding_layer)
model.compile('adam', 'mse')
```

Let's take a look at the structure of this neural network to see what is happening inside it.

In [ ]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 2, 4)              40

=================================================================
Total params: 40
Trainable params: 40
Non-trainable params: 0
_________________________________________________________________
```

For this neural network, which is just an embedding layer, the input is a vector of size 2. These two inputs are integer numbers from 0 to 9 (corresponding to the requested input_dim quantity of 10 values). Looking at the summary above, we see that the embedding layer has 40 parameters. This value comes from the embedded lookup table that contains four amounts (output_dim) for each of the 10 (input_dim) possible integer values for the two inputs. The output is 2 (input_length) length 4 (output_dim) vectors, resulting in a total output size of 8, which corresponds to the Output Shape given in the summary above.

Now, let us query the neural network with two rows. The input is two integer values, as was specified when we created the neural network.

In [ ]:
```
input_data = np.array([
    [1, 2]
])

pred = model.predict(input_data)

print(input_data.shape)
print(pred)
```

```
(1, 2)
[[[-0.04494917  0.01937468 -0.00152863  0.04808659]
  [-0.04002655  0.03441895  0.04462588 -0.01472597]]]
```

Here we see two length-4 vectors that Keras looked up for each input integer. Recall that Python arrays are zero-based. Keras replaced the value of 1 with the second row of the 10 x 4 lookup matrix. Similarly, Keras returned the value of 2 by the third row of the lookup matrix. The following code displays the lookup matrix in its entirety. The embedding layer performs no mathematical operations other than inserting the correct row from the lookup table.

In [ ]:
```
embedding_layer.get_weights()
```

Out[ ]:
```
[array([[-0.03164196,  0.02898774, -0.0273805 ,  0.01066511],
        [-0.04494917,  0.01937468, -0.00152863,  0.04808659],
        [-0.04002655,  0.03441895,  0.04462588, -0.01472597],
        [ 0.02480464, -0.02585896,  0.0099823 ,  0.02589831],
        [-0.02502655,  0.02517617, -0.03199299,  0.00127842],
        [-0.00205797,  0.02709344, -0.04335414, -0.01793201],
        [ 0.03926537,  0.0293855 ,  0.0445295 , -0.02160555],
        [-0.0075082 , -0.03241253,  0.04906586, -0.02384975],
        [ 0.00264529, -0.01921672, -0.0031809 ,  0.00151991],
        [-0.02407705, -0.04659952, -0.02667597, -0.04108504]],
       dtype=float32)]
```

The values above are random parameters that Keras generated as starting points. Generally, we will transfer an embedding or train these random values into something useful. The following section demonstrates how to embed a hand-coded embedding.

Transferring An Embedding

Now, we see how to hard-code an embedding lookup that performs a simple one-hot encoding. One-hot encoding would transform the input integer values of 0, 1, and 2 to the vectors $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ respectively. The following code replaced the random lookup values in the embedding layer with this one-hot coding-inspired lookup table.

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding
import numpy as np

embedding_lookup = np.array([
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]
])

model = Sequential()
embedding_layer = Embedding(input_dim=3, output_dim=3, input_length=2)
model.add(embedding_layer)
model.compile('adam', 'mse')

embedding_layer.set_weights([embedding_lookup])
```

We have the following parameters for the Embedding layer:

- input_dim=3 - There are three different integer categorical values allowed.
- output_dim=3 - Three columns represent a categorical value with three possible values per one-hot encoding.
- input_length=2 - The input vector has two of these categorical values.

We query the neural network with two categorical values to see the lookup performed.

In [ ]:
```
input_data = np.array([
    [0, 1]
])

pred = model.predict(input_data)

print(input_data.shape)
print(pred)
```

```
(1, 2)
[[[1. 0. 0.]
  [0. 1. 0.]]]
```

The given output shows that we provided the program with two rows from the one-hot encoding table. This encoding is a correct one-hot encoding for the values 0 and 1, where there are up to 3 unique values possible.

The following section demonstrates how to train this embedding lookup table.

Training an Embedding

First, we make use of the following imports.

In [ ]:
```
from numpy import array
from tensorflow.keras.preprocessing.text import one_hot
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Embedding, Dense
```

We create a neural network that classifies restaurant reviews according to positive or negative. This neural network can accept strings as input, such as given here. This code also includes positive or negative labels for each review.

In [ ]:
```
# Define 10 resturant reviews.
reviews = [
    'Never coming back!',
    'Horrible service',
    'Rude waitress',
    'Cold food.',
    'Horrible food!',
    'Awesome',
    'Awesome service!',
    'Rocks!',
    'poor work',
    'Couldn\'t have done better']

# Define labels (1=negative, 0=positive)
labels = array([1, 1, 1, 1, 1, 0, 0, 0, 0, 0])
```

Notice that the second to the last label is incorrect. Errors such as this are not too out of the ordinary, as most training data could have some noise.

We define a vocabulary size of 50 words. Though we do not have 50 words, it is okay to use a value larger than needed. If there are more than 50 words, the least frequently used words in the training set are automatically dropped by the embedding layer during training. For input, we one-hot encode the strings. We use the TensorFlow one-hot encoding method here rather than Scikit-Learn. Scikit-learn would expand these strings to the 0's and 1's as we would typically see for dummy variables. TensorFlow translates all words to index values and replaces each word with that index.

In [ ]:
```
VOCAB_SIZE = 50
encoded_reviews = [one_hot(d, VOCAB_SIZE) for d in reviews]
print(f"Encoded reviews: {encoded_reviews}")
```

```
Encoded reviews: [[40, 43, 7], [27, 31], [49, 46], [2, 28], [27, 28], [20],
[20, 31], [39], [18, 39], [11, 3, 18, 11]]
```

The program one-hot encodes these reviews to word indexes; however, their lengths are different. We pad these reviews to 4 words and truncate any words beyond the fourth word.

In [ ]:
```
MAX_LENGTH = 4

padded_reviews = pad_sequences(encoded_reviews, maxlen=MAX_LENGTH,
                               padding='post')
print(padded_reviews)
```

```
[[40 43  7  0]
 [27 31  0  0]
 [49 46  0  0]
 [ 2 28  0  0]
 [27 28  0  0]
 [20  0  0  0]
 [20 31  0  0]
 [39  0  0  0]
 [18 39  0  0]
 [11  3 18 11]]
```

As specified by the **padding=post** setting, each review is padded by appending zeros at the end, as specified by the **padding=post** setting.

Next, we create a neural network to learn to classify these reviews.

In [ ]:
```
model = Sequential()
embedding_layer = Embedding(VOCAB_SIZE, 8, input_length=MAX_LENGTH)
model.add(embedding_layer)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['acc'])

print(model.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 4, 8)              400

 flatten (Flatten)           (None, 32)                0

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 433
Trainable params: 433
Non-trainable params: 0
_________________________________________________________________
None
```

This network accepts four integer inputs that specify the indexes of a padded movie review. The first embedding layer converts these four indexes into four length vectors 8. These vectors come from the lookup table that contains 50 (VOCAB_SIZE) rows of vectors of length 8. This encoding is evident by the 400 (8 times 50) parameters in the embedding layer. The output size from the embedding layer is 32 (4 words expressed as 8-number embedded vectors). A single output neuron is connected to the embedding layer by 33 weights (32 from the embedding layer and a single bias neuron). Because this is a single-class classification network, we use the sigmoid activation function and binary_crossentropy.

The program now trains the neural network. The embedding lookup and dense 33 weights are updated to produce a better score.

In [ ]:
```
# fit the model
model.fit(padded_reviews, labels, epochs=100, verbose=0)
```

Out[ ]:
```
<keras.callbacks.History at 0x7fd87794c4d0>
```

We can see the learned embeddings. Think of each word's vector as a location in the 8 dimension space where words associated with positive reviews are close to other words. Similarly, training places negative reviews close to each other. In addition to the training setting these embeddings, the 33 weights between the embedding layer and output neuron similarly learn to transform these embeddings into an actual prediction. You can see these embeddings here.

In [ ]:
```
print(embedding_layer.get_weights()[0].shape)
print(embedding_layer.get_weights())
```

```
(50, 8)
[array([[-0.11389559, -0.04778124,  0.10034387,  0.12887037,  0.05670259,
        -0.09982903, -0.15423775, -0.06774805],
       [-0.04839246,  0.00527745,  0.0084306 , -0.03498586,  0.010772  ,
         0.04015711,  0.03564452, -0.00849336],
       [-0.11003157, -0.05829103,  0.12370535, -0.07124459, -0.0667479 ,
        -0.14339209, -0.13791779, -0.13947721],
       [-0.15395765, -0.08560142, -0.15915371, -0.0882007 ,  0.15756004,
        -0.10337664, -0.12412377, -0.10282961],
       [ 0.04919637, -0.00870635, -0.02393281,  0.04445953,  0.0124351 ,
         0.02354855, -0.02476437,  0.04543931],
       [-0.00503131,  0.01302261, -0.02866241,  0.04487506, -0.04427315,
         0.00651342, -0.02796236,  0.03458978],
       [-0.03738759, -0.00135366,  0.04961893, -0.04076886, -0.0007545 ,
         0.03454826,  0.03419926, -0.00689811],
       [ 0.14487585,  0.14052217, -0.08246625, -0.08622362,  0.10270283,
        -0.06439426, -0.16649802, -0.11733696],
       [-0.01337775,  0.00189237, -0.04226214, -0.02981731, -0.04849073,
         0.0464913 , -0.04499427, -0.04841725],
       [-0.01929135, -0.02657523, -0.0335291 ,  0.04808146,  0.02409947,
        -0.03780599,  0.03453754,  0.00598647],
       [ 0.03076488, -0.03929596,  0.00840779, -0.03980947,  0.04209021,
        -0.00642526,  0.03741593,  0.04605447],
       [ 0.11537231, -0.10763969, -0.06139125,  0.07191044,  0.05322507,
         0.15153708, -0.14278722,  0.11250742],
       [-0.04048342, -0.02535482, -0.01568266, -0.02351468,  0.00865855,
         0.04086712, -0.03859865,  0.0365578 ],
       [-0.0009298 , -0.0311846 , -0.03491043, -0.00289371,  0.00757905,
        -0.03187181, -0.02323085, -0.01488547],
       [ 0.0320026 ,  0.03818611,  0.00219003, -0.03297286, -0.03609738,
        -0.00905116, -0.00735079, -0.0369678 ],
       [ 0.04876169,  0.04988963, -0.01918377,  0.02061111, -0.03650783,
         0.00809064,  0.00043495, -0.02308334],
       [-0.02140537,  0.02220272,  0.00469884,  0.0342283 ,  0.01847946,
         0.02940113, -0.04855499,  0.02044804],
       [-0.00828004, -0.0079689 ,  0.01667002,  0.0414703 , -0.01305557,
         0.04526286, -0.01467935,  0.01147614],
       [-0.14282468, -0.08361981, -0.11100344,  0.1147782 ,  0.13931683,
         0.05983332,  0.16483088,  0.09642172],
       [-0.04617438,  0.04929153,  0.0485074 , -0.02250378,  0.01294557,
        -0.0425485 , -0.01274359,  0.00403596],
       [ 0.08578632,  0.10722891, -0.10169367,  0.05640666,  0.13935997,
         0.07905768,  0.0912255 ,  0.14614286],
       [-0.02422597, -0.02895569,  0.02458526, -0.02941357,  0.03783615,
         0.0217586 ,  0.04737884,  0.03385517],
       [-0.01605659,  0.02846745, -0.04217149,  0.00933688, -0.015615  ,
        -0.0185383 ,  0.03455376,  0.0217413 ],
       [-0.02496419, -0.01964381, -0.01747011, -0.0086274 , -0.00279769,
        -0.00473202,  0.04959089, -0.02818167],
       [-0.01308316,  0.0437695 , -0.01201218, -0.00937818, -0.03936937,
         0.03369248,  0.01404865,  0.01300433],
       [-0.03047577, -0.04215126, -0.03603753, -0.01572833,  0.04595536,
        -0.01445602,  0.02598487, -0.03712183],
       [ 0.04174629,  0.030602  , -0.01565778,  0.01411921, -0.03829115,
         0.02699218, -0.03978034, -0.00037332],
       [-0.05509803, -0.12121415,  0.12930614, -0.14208739, -0.05467908,
        -0.10421305, -0.1347957 , -0.09714746],
       [ 0.14368567,  0.14523256,  0.15996216,  0.07271292, -0.10887505,
         0.07155557,  0.10750765,  0.14647684],
       [-0.04667553, -0.00594231, -0.04209081, -0.01220823, -0.02044651,
         0.02359882,  0.01033651, -0.01691378],
       [ 0.02788267, -0.0466502 , -0.04354659, -0.04944308,  0.00530468,
         0.03017677,  0.01628789,  0.00456915],
       [ 0.09592342,  0.05642203,  0.03576508,  0.06546731, -0.03308697,
         0.03154759,  0.00280966,  0.03369548],
       [-0.00399817, -0.02812622, -0.00763954, -0.003208  ,  0.04371027,
        -0.03186812,  0.01646887, -0.04135863],
       [ 0.00120915,  0.00111195,  0.01940939,  0.0100676 ,  0.02689103,
        -0.02420806,  0.04829462, -0.00500059],
       [-0.00374997,  0.00533805,  0.01584294, -0.01231242, -0.02583057,
        -0.00426785, -0.01593303,  0.03316021],
       [-0.00542512, -0.02522955,  0.01944559,  0.04694534,  0.01956921,
        -0.04743705,  0.01203604, -0.04249186],
       [ 0.04021386, -0.00147871, -0.03729609, -0.04367838, -0.02620382,
        -0.03366937, -0.04764401,  0.01843042],
       [-0.04885202, -0.04030935, -0.02691921, -0.04069231,  0.00133073,
         0.04187706,  0.01700257, -0.0269224 ],
       [-0.04759267, -0.02806743, -0.02340071,  0.04413268,  0.04873205,
        -0.02571398,  0.02112493,  0.01220033],
       [ 0.03645799,  0.04670727, -0.14964601,  0.06317957,  0.12738568,
        -0.05583218, -0.07265829, -0.11887868],
       [-0.0461492 , -0.14710744,  0.14215472, -0.08502222, -0.11263344,
        -0.10313905, -0.09941045, -0.0613514 ],
       [-0.01235803, -0.03596945,  0.04333005, -0.02633744,  0.0076986 ,
        -0.02331397, -0.02244077,  0.02170218],
       [ 0.02890852, -0.02253481, -0.04383245, -0.00917351,  0.01134578,
         0.0413558 , -0.00813813,  0.03958623],
       [ 0.13829918,  0.0676541 ,  0.16875601,  0.04536283, -0.12547925,
         0.13549416,  0.06408142,  0.1365626 ],
       [ 0.02720174,  0.02317807, -0.01934367,  0.03661523, -0.00081351,
        -0.00664594, -0.01546872,  0.00292607],
       [ 0.03418565, -0.02236365, -0.03703803,  0.01724467, -0.02788099,
        -0.01143361, -0.00885036, -0.00753104],
       [ 0.11629202,  0.08401583,  0.12823549,  0.04578856, -0.10711329,
         0.12236115,  0.12761551,  0.12674938],
       [-0.01328101,  0.01608239, -0.02894524,  0.03419088,  0.04457787,
         0.02493219,  0.04973162,  0.03453101],
       [-0.00029699, -0.0425287 ,  0.02509956, -0.00861088,  0.04153964,
        -0.04445877, -0.00612149, -0.03430663],
       [-0.08493928, -0.10910758,  0.0605178 , -0.10072854, -0.11677803,
        -0.05648913, -0.13342443, -0.08516318]], dtype=float32)]
```

We can now evaluate this neural network's accuracy, including the embeddings and the learned dense layer.

In [ ]:
```python
loss, accuracy = model.evaluate(padded_reviews, labels, verbose=0)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 1.0
```

The accuracy is a perfect 1.0, indicating there is likely overfitting. It would be good to use early stopping to not overfit for a more complex data set.

In [ ]:
```
print(f'Log-loss: {loss}')
```

```
Log-loss: 0.48446863889694214
```

However, the loss is not perfect. Even though the predicted probabilities indicated a correct prediction in every case, the program did not achieve absolute confidence in each correct answer. The lack of confidence was likely due to the small amount of noise (previously discussed) in the data set. Some words that appeared in both positive and negative reviews contributed to this lack of absolute certainty.

In [ ]: