Final Project

Jinwoo Kim

Wide and Deep Learning for Recommender Systems

Introduction

This project will try to use an algorithm, Wide and Deep Learning, which is published by Google in 2016. This algorithm will extend the basic networks that we have explored. This project will focus on applying network knowledge that we covered in class. This algorithm is used to recommend Google Play's apps and is specifically applied to sort out candy apps created based on user's search query.

Abstract

Often, when solving regression or classification problems, a linear model is used. At this time, the cross-product between features is effective in remembering the characteristics of data. However, for generalization, it is necessary to go through an engineering process that requires a lot of labor and management. On the other hand, the natural net model using embedding requires less engineering effort and is excellent in learning combination between features. However, there is a problem that the characteristics of the data cannot be remembered in detail due to excessive generalization.

Wide and Deep Model

First, let's briefly look at the characteristics of the wide model and deep model. Here, we will see the difference between the input of the two models. Suppose that the interaction between the app feature installed by the user and the app feature viewed is used as input. At this time, only three apps exist, A, B, and C, and the apps installed by the user and the apps clicked are as follows.

$$user\_install\_app = [A, B]$$

$$user\_impression\_app = [A, C]$$

In the wide model, interaction is expressed through cross-product between the installed app and the viewed app. For example, if the user installed the A app and watched the C app at

the same time, it can be expressed as (A, C) = (1,1), and multiplied by the two values will be 1. In this way, there are a total of nine combinations of all apps, and there are only four cases where 1. This method is strong against memory because it learns all cases that become 1, and is excellent at learning niche combination that reflects the user's unique taste, while a pair that becomes zero has the disadvantage that learning is impossible.

On the other hand, the deep model expresses the A, B, and C apps in the same embedding space (here, 2D assumption). However, Niche combination rarely appears in other users, so there is not enough information to express the app. Therefore, these apps are likely to have embedding vectors that do not properly express their relationship with other apps.

Wide and Deep Learning

Wide Component

```
class Wide:

    def __init__(self, X_train, y_train):

        wide_input = Input(shape=(X_train.shape[1],))
        output_layer = Dense(y_train.shape[1], activation='sigmoid')(wide_input)

        # Model
        self.wide_model = Model(wide_input, output_layer)
        self.wide_model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer='Adam')

    def get_model(self):
        return self.wide_model
```

Figure #1, Wide class in Model.py

```
def Wide(self):

    load = Data()
    X_train, y_train, X_test, y_test = load.get_wide_model_data(self.df_train, self.df_test)

    model = Wide(X_train, y_train)
    model = model.get_model()
    model.fit(X_train, y_train, epochs=10,  batch_size=64)

    print('wide model accuracy:', model.evaluate(X_test, y_test)[1])
```

Figure #2, Wide class in Run.py

Deep Component

```
class Deep:

    def __init__(self, X_train, y_train, embeddings_tensors, continuous_tensors):

        deep_input = [et[0] for et in embeddings_tensors] + [ct[0] for ct in continuous_tensors]
        deep_embedding = [et[1] for et in embeddings_tensors] + [ct[1] for ct in continuous_tensors]
        deep_embedding = Flatten()(concatenate(deep_embedding))

        layer_1 = Dense(100, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(deep_embedding)
        layer_1_dropout = Dropout(0.5)(layer_1)

        layer_2 = Dense(50, activation='relu')(layer_1_dropout)
        layer_2_dropout = Dropout(0.5)(layer_2)

        output_layer = Dense(y_train.shape[1], activation='sigmoid')(layer_2_dropout)

        self.deep_model = Model(deep_input, output_layer)
        self.deep_model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer='Adam')

    def get_model(self):
        return self.deep_model
```

Figure #3, Deep class in Model.py

```
def Deep(self):

    load = Data()
    X_train, y_train, X_test, y_test, \
    embeddings_tensors, continuous_tensors = load.get_deep_model_data(self.df_train, self.df_test)

    model = Deep(X_train, y_train, embeddings_tensors, continuous_tensors)
    model = model.get_model()
    model.fit(X_train, y_train, batch_size=64, epochs=10)

    print('deep model accuracy:', model.evaluate(X_test, y_test)[1])
```

Figure #4, Deep class in Run.py

Now, let's look at how to point train the wide model and deep model. Here, unlike ensembles that combine several models, point training learns by simultaneously backpropagating the gradient of output to the wide and deep models.

```python
class Wide_Deep:

    def __init__(self, X_train_wide, y_train_wide,
                       X_train_deep, y_train_deep,
                       embeddings_tensors, continuous_tensors):
        wide_input = Input(shape=(X_train_wide.shape[1],), dtype='float32', name='wide')

        deep_input = [et[0] for et in embeddings_tensors] + [ct[0] for ct in continuous_tensors]
        deep_embedding = [et[1] for et in embeddings_tensors] + [ct[1] for ct in continuous_tensors]

        deep_embedding = Flatten()(concatenate(deep_embedding))
        layer_1 = Dense(50, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(deep_embedding)
        layer_1_dropout = Dropout(0.5)(layer_1)
        layer_2 = Dense(20, activation='relu', name='deep')(layer_1_dropout)
        layer_2_dropout = Dropout(0.5)(layer_2)

        wd_input = concatenate([wide_input, layer_2_dropout])
        wd_output = Dense(y_train_deep.shape[1], activation='sigmoid', name='wide_deep')(wd_input)

        self.wide_deep_model = Model([wide_input, deep_input], wd_output)
        self.wide_deep_model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Figure #5, Wide_Deep class in Model.py

```python
def Wide_and_Deep(self):

    load = Data()
    X_train_wide, y_train_wide, X_test_wide, y_test_wide = load.get_wide_model_data(self.df_train, self.df_test)
    X_train_deep, y_train_deep, X_test_deep, y_test_deep, \
    embeddings_tensors, continuous_tensors = load.get_deep_model_data(self.df_train, self.df_test)

    X_tr_wd = [X_train_wide] + X_train_deep
    y_tr_wd = y_train_deep

    X_te_wd = [X_test_wide] + X_test_deep
    y_te_wd = y_test_deep

    model = Wide_Deep(X_train_wide, y_train_wide, X_train_deep, y_train_deep, embeddings_tensors, continuous_tensors)
    model = model.get_model()
    model.fit(X_tr_wd, y_tr_wd, epochs=5, batch_size=128)

    print('wide and deep model accuracy:', model.evaluate(X_te_wd, y_te_wd)[1])
```

Figure #6, Wide_and_Deep class in Run.py

```
Epoch 1/5
255/255 [==============================] - 1s 1ms/step - loss: 1.8278 - accuracy: 0.7613
Epoch 2/5
255/255 [==============================] - 0s 1ms/step - loss: 0.4128 - accuracy: 0.8260
Epoch 3/5
255/255 [==============================] - 0s 1ms/step - loss: 0.3790 - accuracy: 0.8393
Epoch 4/5
255/255 [==============================] - 0s 1ms/step - loss: 0.3634 - accuracy: 0.8439
Epoch 5/5
255/255 [==============================] - 0s 1ms/step - loss: 0.3537 - accuracy: 0.8457
509/509 [==============================] - 0s 613us/step - loss: 0.6665 - accuracy: 0.7359
wide and deep model accuracy: 0.7359498739242554
```

Figure #7, Result Console

Conclusion

The wide and deep model is a combination of a wide model specialized in memorization and a deep model specialized in generalization. It is said to have originated from the idea of factorization machine (Fm). Fm is a linear model, and for generalization, engineering that requires a lot of labor and management is required. Wide and deep is an algorithm that enables complex engineering in the deep part while using the linear model in the wide part, and it can be understood that both the advantages and limitations of Fm are implemented fm