Wide & Deep Learning for Recommender Systems

한석원

2023.07.25

Outline

- Introduction
- Background
- Wide and Deep model
- Joint training
- Model
- Implementation

Introduction

Recommender System

input query is a set of user and contextual information (such as what apps they have installed, what devices they use), and the output is a ranked list of items (such as applications in app store)

+

achieve both memorization and generalization

Introduction

Memorization (Wide) : frequent co-occurrence of items or features을 학습하고 correlation available in historical data을 활용
“Seagulls can fly”, “Pigeons can fly”

Generalization (Deep) : transitivity of correlation 에 근거하고 new feature combinations that have never or rarely occurred in past 탐색
“Animal with wings can fly”

Generalization + memorizing exceptions (Wide + Deep) : memorize what the users like + by using the previous information, recommend a new one
“Animals with wings can fly, but penguins cannot fly”

Introduction

Wide model (memorization)

- Logistic regression 사용
- Simple model
- One-hot encoding
- Cross product 사용
- Feature transformation 사용

Deep model (generalization)

- Deep neural network 사용
- Sparse data에 대해서 학습 어려움
- Feature data를 low dimension dense embedding vector로 학습

Introduction

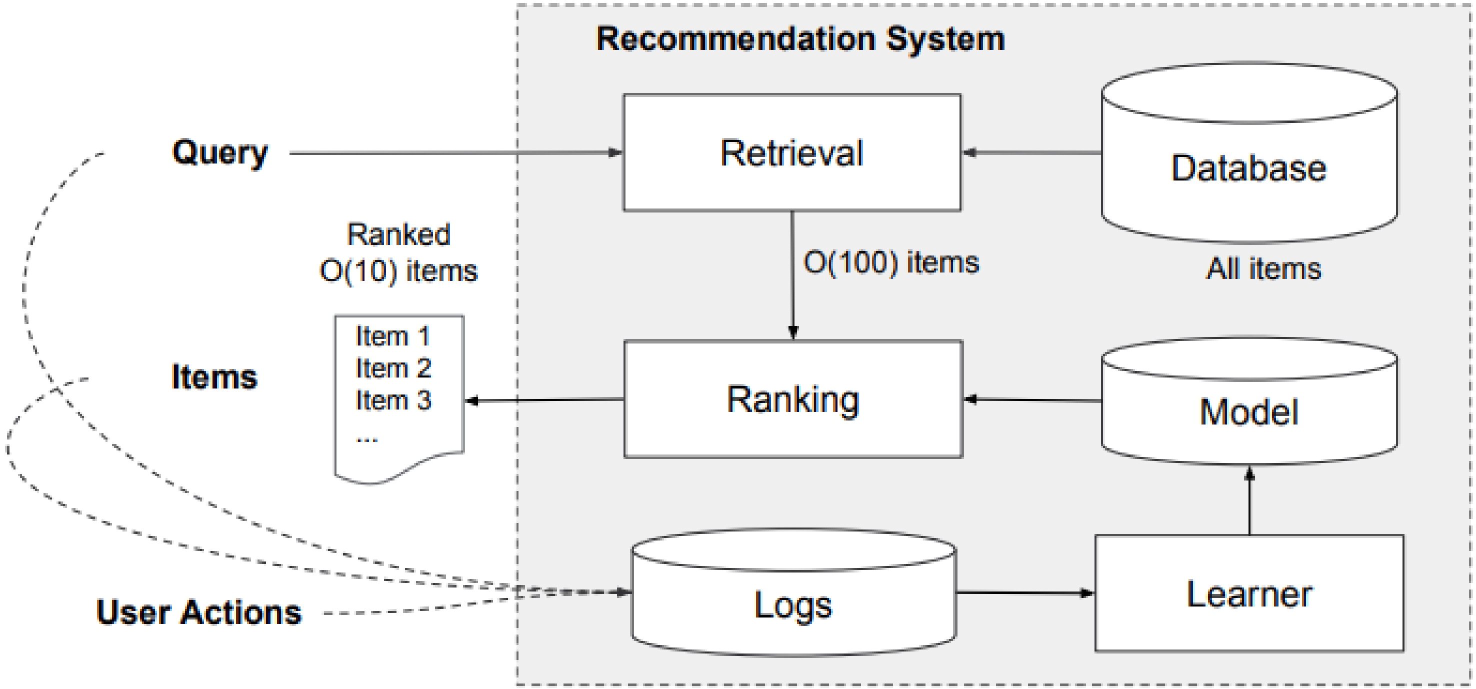


Figure 2: Overview of the recommender system.

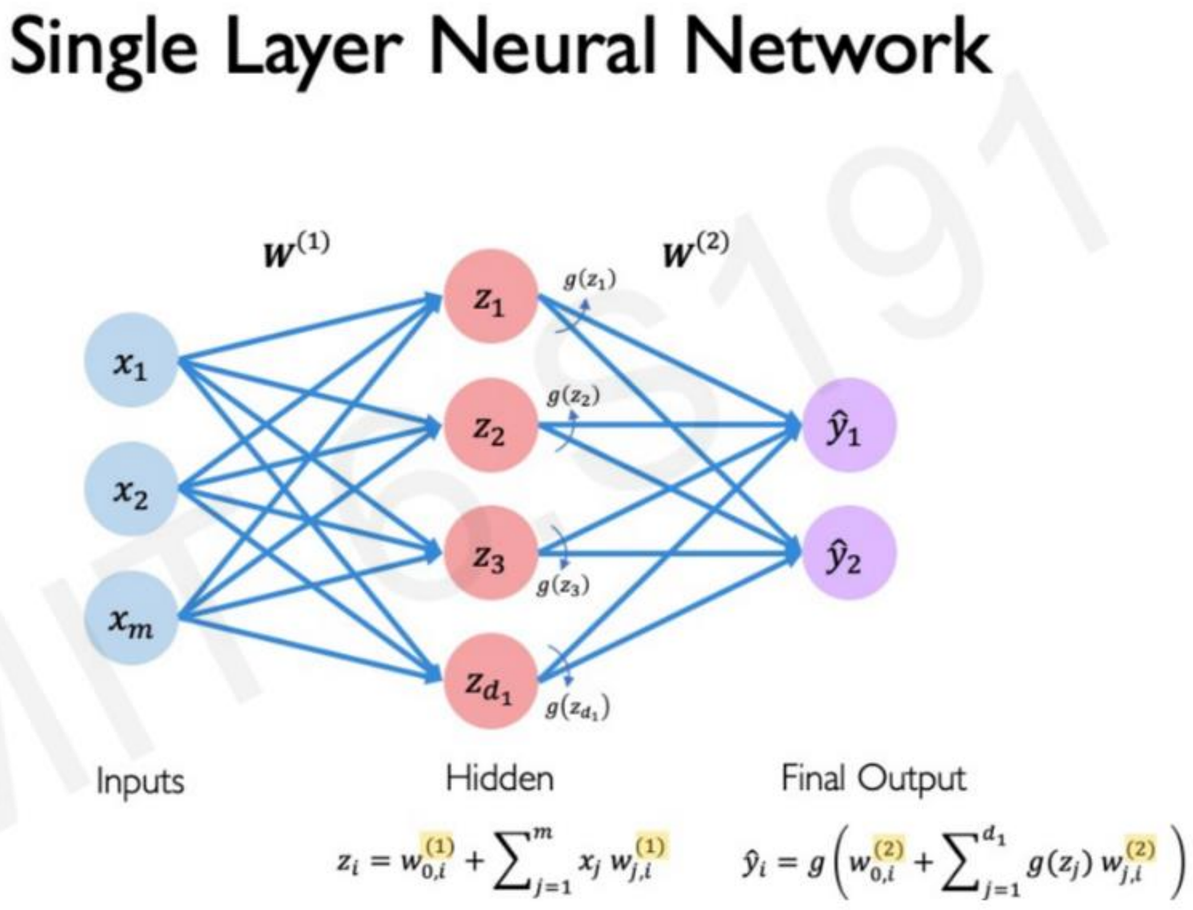
1. Get a query including various user and contextual features (query generated when a user visits the app store)

2. Recommender system returns a list of apps

3. Once we recommend apps, get user actions and store them on the training data

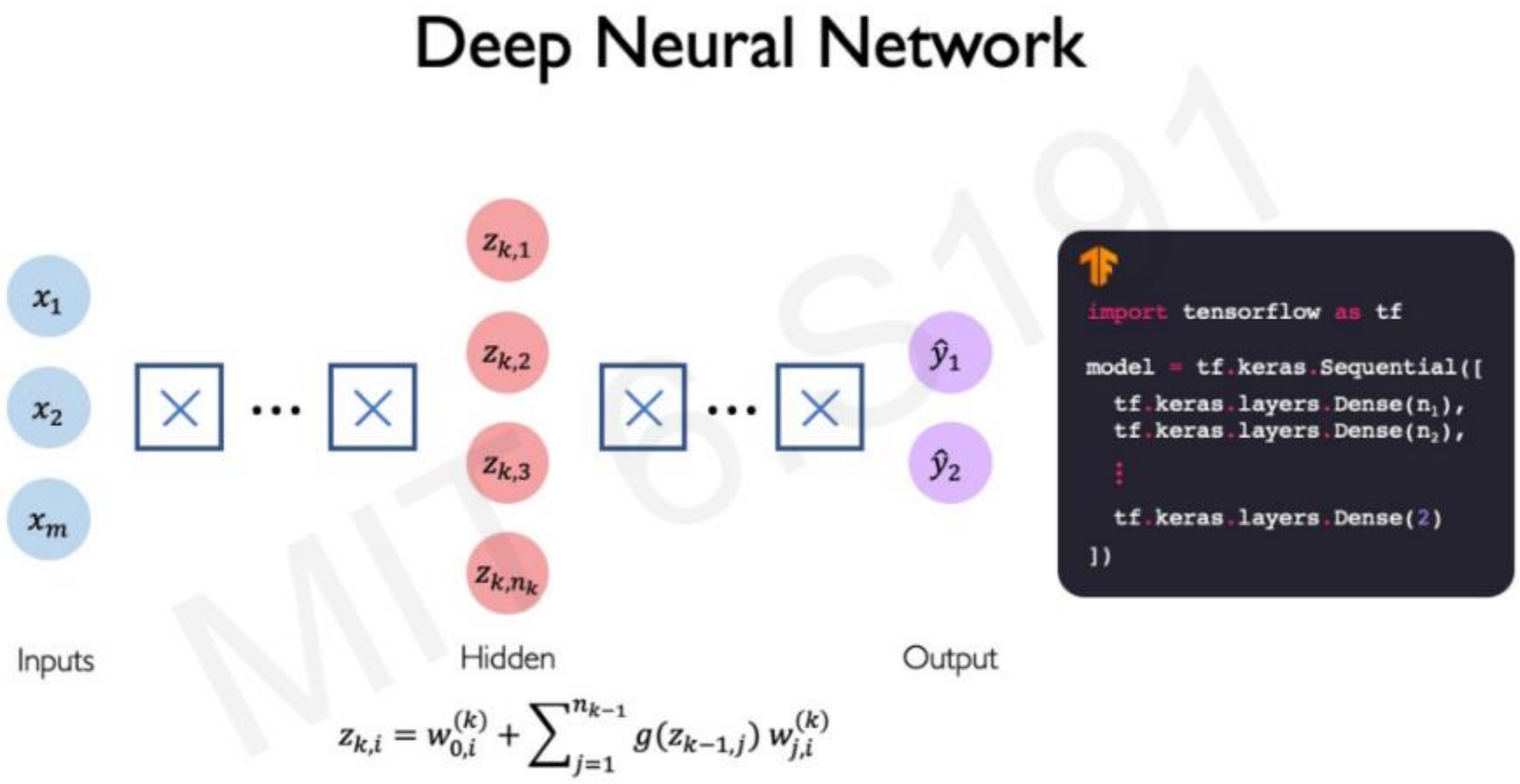
P(y|x) 를 기반으로 앱의 순위를 결정

y: probability of user action
x: features



hidden layer output is obtained by dot product → adding a bias → applying the non linearity
the difference between z_1 and z_2 is that the weight vectors we dot product are different

Background



We can create a deep neural network by stacking layers

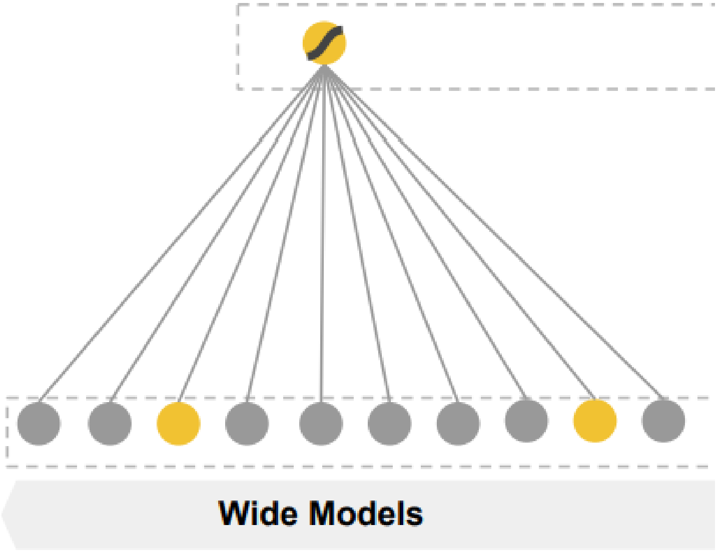
Wide Model

Wide component is a generalized linear model of the form $y = w^T x + b$.

y is the prediction and $x = [x_1, x_2, ..., x_d]$ is a vector of d features. The feature set x includes raw input features and also transformed features.

Cross tranformation

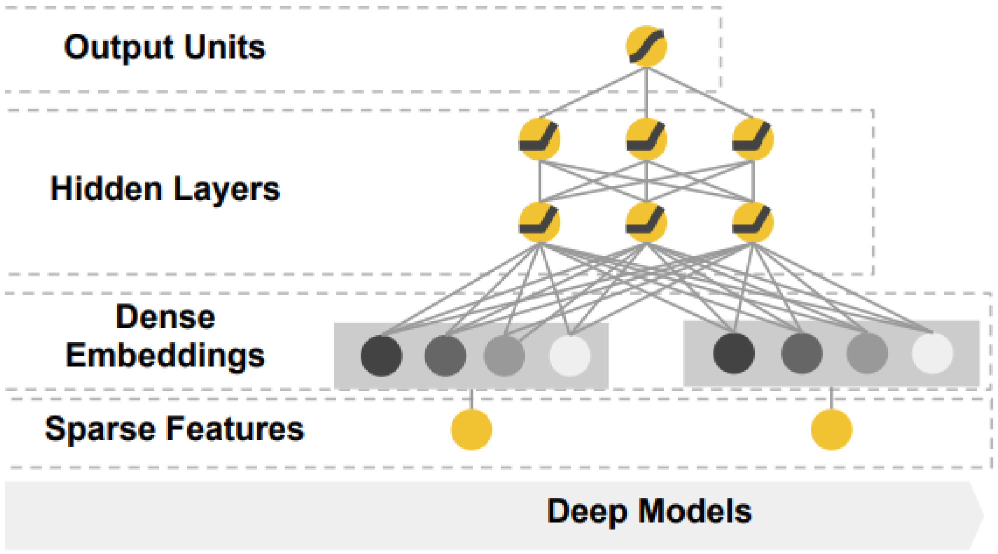
$$\phi_k(\mathbf{x}) = \prod_{i=1}^{d} x_i^{c_{ki}} \quad c_{ki} \in \{0, 1\}$$



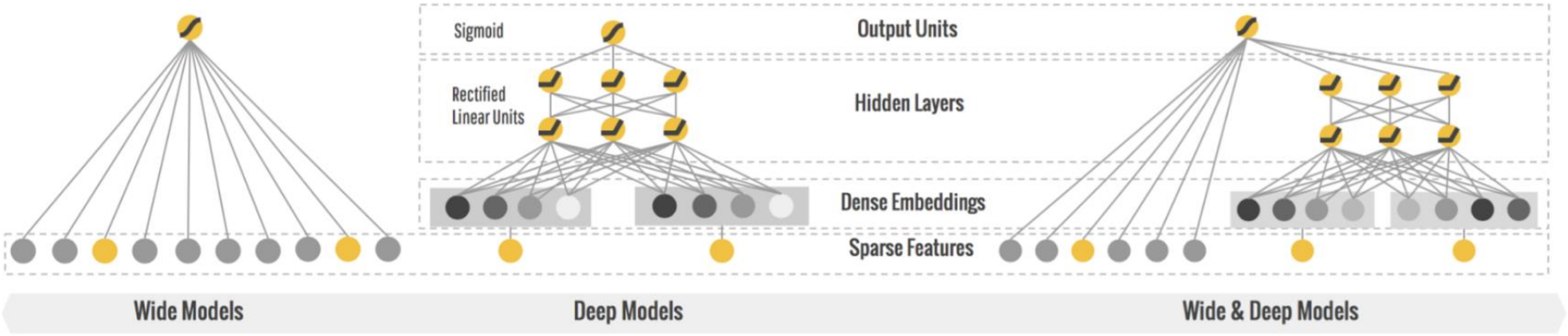
Deep Model

Deep component is a feed-forward neural network. $a^{(l+1)} = f(X^{(l)}a^{(l)} + b^{(l)})$

l is the layer number and f is the activation function, often rectified linear units (ReLU).
$a^{(l)}, b^{(l)}, W^{(l)}$ are activations, bias, and model-weights at l-th layer.



Wide and Deep model



Joint training

Ensemble 과 달리 optimize all parameter simultaneously

한번에 Both wide and deep part backpropagation

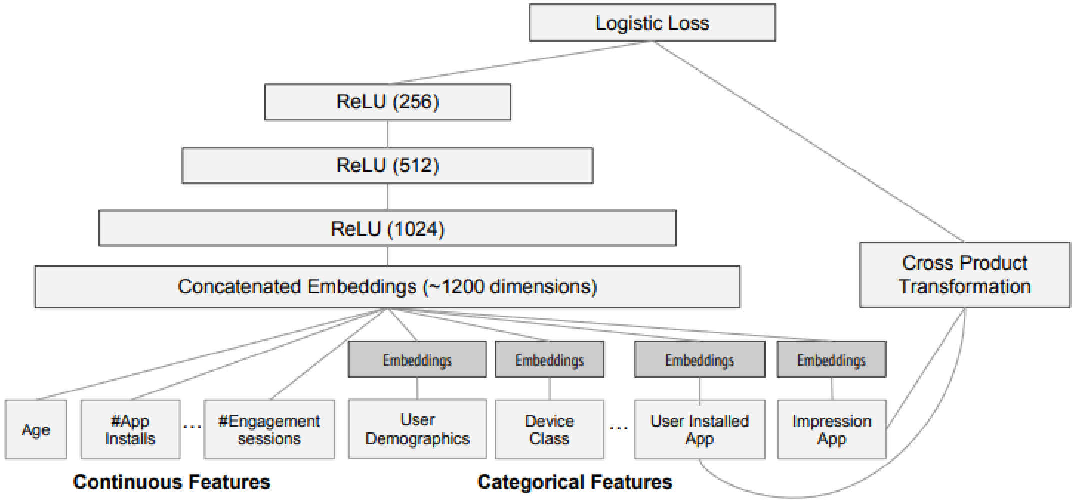


Figure 4: Wide & Deep model structure for apps recommendation.

Joint training

For a logistic regression problem, the model's prediction is :

$$P(Y = 1|x) = \sigma(w_{wide}^T[x, \phi(x)] + w_{deep}^T a^{(l_f)} + b)$$

where Y is the binary class label, $\sigma(\cdot)$ is the sigmoid function, $\phi(x)$ are the cross product transformations of the original features x, and b is the bias term. w_{wide} is the vector of all wide model weights, and w_{deep} are the weights applied on the final activations $a^{(l_f)}$.

Data Generalization

In this stage, user and app impression data within a period of time are used to generate training data. Each example corresponds to one impression. The label is app acquisition: 1 if the impressed app was installed, and 0 otherwise.

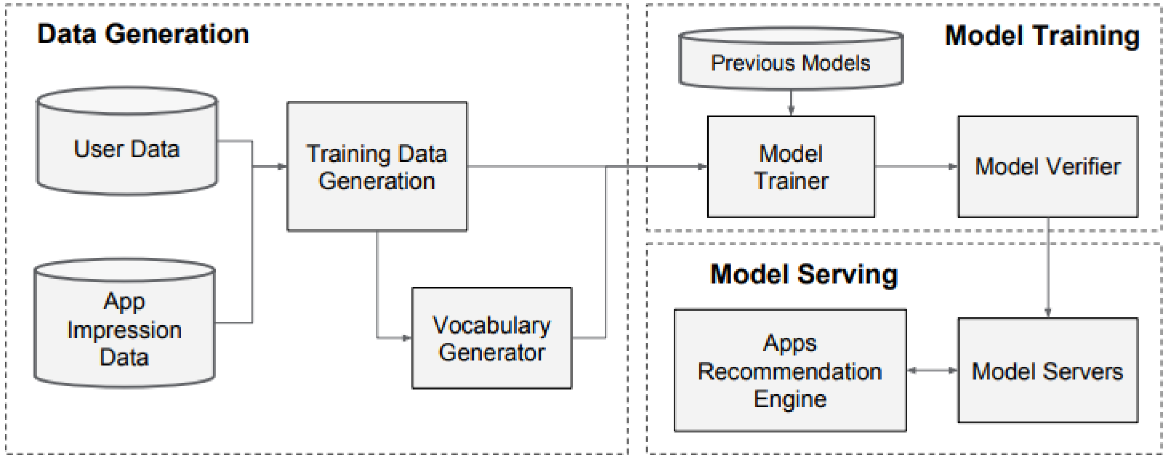


Figure 3: Apps recommendation pipeline overview.

Model training

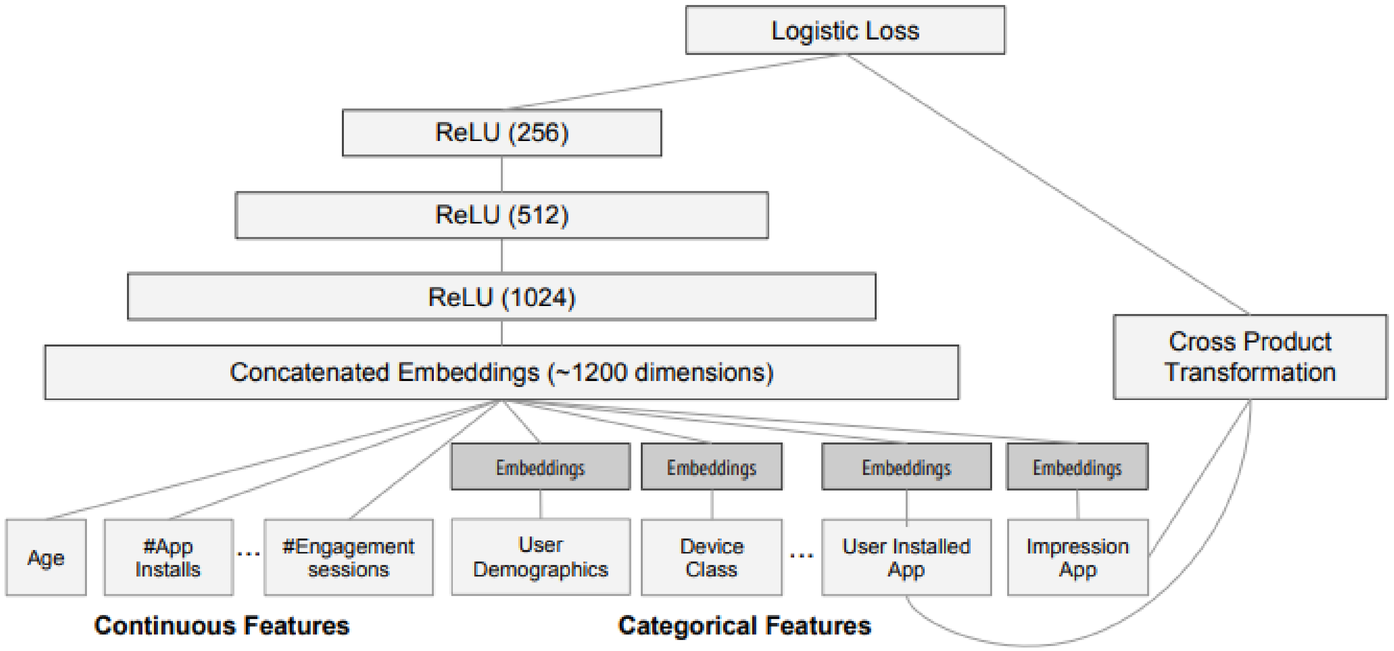


Figure 4: Wide & Deep model structure for apps recommendation.

Result

Table 1: Offline & online metrics of different models. Online Acquisition Gain is relative to the control.

Model	Offline AUC	Online Acquisition Gain
Wide (control)	0.726	0%
Deep	0.722	+2.9%
Wide & Deep	0.728	+3.9%

Table 2: Serving latency vs. batch size and threads.

Batch size	Number of Threads	Serving Latency (ms)
200	1	31
100	2	17
50	4	14

Implementation

```
estimator = DNNLinearCombinedClassifier(
    linear_feature_columns=my_wide_features,
    dnn_feature_columns=my_deep_features,
    dnn_hidden_units=[256, 64, 16],
    ...)

estimator.fit(...)
estimator.evaluate(...)
```

Implementation

	user_id	movie_id	timestamp	title	release_date	video_release_date	unknown	action	adventure	animation	...	romance	sci_fi	thriller	war	western	age	gender	occupation	zip_code	label
0	298	474	884182806	Dr. Strangelove or: How I Learned to Stop Worr...	1963	NaN	0	0	0	0	...	0	1	0	1	0	44	0	executive	01581	1
1	298	257	884126240	Men in Black (1997)	1997	NaN	0	1	1	0	...	0	1	0	0	0	44	0	executive	01581	1
2	298	118	884183016	Twister (1996)	1996	NaN	0	1	1	0	...	0	0	1	0	0	44	0	executive	01581	1
3	298	546	884184098	Broken Arrow (1996)	1996	NaN	0	1	0	0	...	0	0	1	0	0	44	0	executive	01581	0
4	298	181	884125629	Return of the Jedi (1983)	1997	NaN	0	1	1	0	...	1	1	0	1	0	44	0	executive	01581	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
11633	662	276	880570080	Leaving Las Vegas (1995)	1995	NaN	0	0	0	0	...	1	0	0	0	0	55	0	librarian	19102	0
11634	662	319	880569520	Everyone Says I Love You (1996)	1996	NaN	0	0	0	0	...	1	0	0	0	0	55	0	librarian	19102	0
11635	662	6	880571006	Shanghai Triad (Yao a yao yao dao waipo qiao) ...	1995	NaN	0	0	0	0	...	0	0	0	0	0	55	0	librarian	19102	1
11636	662	985	880571006	Blood & Wine (1997)	1996	NaN	0	0	0	0	...	0	0	0	0	0	55	0	librarian	19102	1
11637	662	1511	880570301	Children of the Revolution (1996)	1997	NaN	0	0	0	0	...	0	0	0	0	0	55	0	librarian	19102	1

11638 rows × 30 columns

Dataset : movielens 100k (rating >= 4 : 1, else 0)

Implementation

```python
COLUMNS = ['user_id', 'movie_id', 'gender', 'age', 'unknown', 'action', 'adventure', 'animation', 'children', 'comedy', 'crime', 'documentary',
                'drama', 'fantasy', 'film_noir', 'horror', 'musical', 'mystery', 'romance', 'sci_fi', 'thriller',
                'war', 'western', 'release_date', 'video_release_date', 'occupation', 'zip_code', 'timestamp']

CATEGORICAL_COLUMNS = ['gender', 'unknown', 'action', 'adventure', 'animation', 'children', 'comedy', 'crime', 'documentary',
                'drama', 'fantasy', 'film_noir', 'horror', 'musical', 'mystery', 'romance', 'sci_fi', 'thriller',
                'war', 'western', 'video_release_date', 'occupation', 'zip_code']

CONTINUOUS_COLUMNS = ['timestamp', 'age', 'release_date']

labels = data['label'].values
```

```python
def process_categorical_columns(data, columns):
    for col in columns:
        if col in data.columns:
            data[col] = LabelEncoder().fit_transform(data[col])
    return data

data = process_categorical_columns(data, CATEGORICAL_COLUMNS)

def process_continuous_columns(data, columns):
    data[columns] = StandardScaler().fit_transform(data[columns])
    return data

data = process_continuous_columns(data, CONTINUOUS_COLUMNS)
```

Implementation

```
[164] class WideAndDeep(tf.keras.Model):
          def __init__(self, feature_columns, deep_feature_columns, hidden_units, output_dim, l1_regularization_strength):
              super(WideAndDeep, self).__init__()
              self.wide_feature_layer = tf.keras.layers.DenseFeatures(feature_columns)
              self.deep_feature_layer = tf.keras.layers.DenseFeatures(deep_feature_columns)
              self.deep_layers = [tf.keras.layers.Dense(units=units, activation='relu') for units in hidden_units]
              self.output_layer = tf.keras.layers.Dense(units=output_dim, activation='sigmoid')
              self.l1_regularizer = tf.keras.regularizers.l1(l1_regularization_strength)

          def call(self, inputs):
              wide_outputs = self.wide_feature_layer(inputs)
              deep_outputs = self.deep_feature_layer(inputs)
              for layer in self.deep_layers:
                  deep_outputs = layer(deep_outputs)
              concat_outputs = tf.concat([wide_outputs, deep_outputs], axis=1)
              return self.output_layer(concat_outputs)
```

```
[165] hidden_units = [32, 64, 128]
      output_dim = 1
      l1_regularization_strength = 0.001

      model = WideAndDeep(feature_columns, deep_feature_columns, hidden_units, output_dim, l1_regularization_strength)

      optimizer = tf.keras.optimizers.Adam()
      model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

      batch_size = 50
      epochs = 10

      model.fit(train_data_dict, train_labels, batch_size=batch_size, epochs=epochs)
```

Implementation

```
Epoch 1/10
269/269 [==============================] - 11s 7ms/step - loss: 0.6653 - accuracy: 0.5970
Epoch 2/10
269/269 [==============================] - 2s 7ms/step - loss: 0.6477 - accuracy: 0.6269
Epoch 3/10
269/269 [==============================] - 2s 7ms/step - loss: 0.6319 - accuracy: 0.6423
Epoch 4/10
269/269 [==============================] - 2s 8ms/step - loss: 0.6156 - accuracy: 0.6546
Epoch 5/10
269/269 [==============================] - 3s 13ms/step - loss: 0.6035 - accuracy: 0.6694
Epoch 6/10
269/269 [==============================] - 2s 8ms/step - loss: 0.5977 - accuracy: 0.6732
Epoch 7/10
269/269 [==============================] - 2s 7ms/step - loss: 0.5931 - accuracy: 0.6774
Epoch 8/10
269/269 [==============================] - 2s 7ms/step - loss: 0.5902 - accuracy: 0.6805
Epoch 9/10
269/269 [==============================] - 2s 7ms/step - loss: 0.5894 - accuracy: 0.6805
Epoch 10/10
269/269 [==============================] - 2s 7ms/step - loss: 0.5877 - accuracy: 0.6813
<keras.callbacks.History at 0x789bb599fb50>
```

```
106/106 [==============================] - 2s 4ms/step
Wide and Deep AUC: 0.7194
```

```
Epoch 1/10
269/269 [==============================] - 13s 8ms/step - loss: 0.6684 - accuracy: 0.5958
Epoch 2/10
269/269 [==============================] - 2s 8ms/step - loss: 0.6500 - accuracy: 0.6278
Epoch 3/10
269/269 [==============================] - 2s 9ms/step - loss: 0.6346 - accuracy: 0.6454
Epoch 4/10
269/269 [==============================] - 3s 12ms/step - loss: 0.6200 - accuracy: 0.6602
Epoch 5/10
269/269 [==============================] - 2s 7ms/step - loss: 0.6099 - accuracy: 0.6708
Epoch 6/10
269/269 [==============================] - 2s 7ms/step - loss: 0.6037 - accuracy: 0.6730
Epoch 7/10
269/269 [==============================] - 2s 7ms/step - loss: 0.6000 - accuracy: 0.6754
Epoch 8/10
269/269 [==============================] - 2s 7ms/step - loss: 0.5981 - accuracy: 0.6759
Epoch 9/10
269/269 [==============================] - 2s 8ms/step - loss: 0.5961 - accuracy: 0.6767
Epoch 10/10
269/269 [==============================] - 3s 12ms/step - loss: 0.5952 - accuracy: 0.6744
<keras.callbacks.History at 0x789b9f683b20>
```

```
106/106 [==============================] - 3s 5ms/step
Wide model AUC: 0.7052
```

```
Epoch 1/10
269/269 [==============================] - 15s 9ms/step - loss: 0.6622 - accuracy: 0.6011
Epoch 2/10
269/269 [==============================] - 2s 8ms/step - loss: 0.6383 - accuracy: 0.6295
Epoch 3/10
269/269 [==============================] - 2s 8ms/step - loss: 0.6176 - accuracy: 0.6562
Epoch 4/10
269/269 [==============================] - 2s 9ms/step - loss: 0.6038 - accuracy: 0.6684
Epoch 5/10
269/269 [==============================] - 3s 13ms/step - loss: 0.5952 - accuracy: 0.6761
Epoch 6/10
269/269 [==============================] - 2s 8ms/step - loss: 0.5867 - accuracy: 0.6828
Epoch 7/10
269/269 [==============================] - 2s 8ms/step - loss: 0.5782 - accuracy: 0.6879
Epoch 8/10
269/269 [==============================] - 2s 8ms/step - loss: 0.5746 - accuracy: 0.6959
Epoch 9/10
269/269 [==============================] - 2s 8ms/step - loss: 0.5685 - accuracy: 0.6988
Epoch 10/10
269/269 [==============================] - 2s 9ms/step - loss: 0.5608 - accuracy: 0.7020
<keras.callbacks.History at 0x789badc28820>
```

```
106/106 [==============================] - 3s 5ms/step
Deep model AUC: 0.7092
```

	Wide	**Deep**	**Wide and Deep**
Paper AUC	0.726	0.722	0.728
My AUC	0.7052	0.7092	0.7194

Future improvement for my implementation

Wide model : Follow-the-regularized-leader (FTRL) algorithm with L1 regularization
Cross product of all features

Deep model : optimizer Adagrad 대신 Adam 사용

Dataset 차이로 인한 Model 간소화