The implementation of DNN

by Minyan Zhang

March 2020

Content

- The Programming Language and Software Libraries.
- The Installation of Tensorflow 2.0.
- The Construction of Models
- The Regression Problem.
- The Solution to the Regression Problem.
- The Image Recognition Problem.
- The Solution.

The Programming Language and Software Libraries

- Python & Anaconda
- Tensorflow 1.0 & Tensorflow 2.0
  https://www.tensorflow.org/
- Pytorch
  https://pytorch.org/
- Other useful libraries: pandas, numpy, matplotlib, seaborn

The Installation of Tensorflow 2.0. – CPU & GPU

- Pip + Virtual Environment
  https://www.tensorflow.org/install/pip
- Anaconda(CPU)
  - Construct an virtual environment: conda create -n the_name python=the_version_of_python(3.X)
  - Enter the virtual environment: conda activate the_name
  - Install the tensorflow: pip install tensorflow==the_version_of_tensorflow (2.1.0)
- For GPU version, you need to chech whether the GPU on your computer supports CUDA. Generally, the NVIDIA GPU will support CUDA.

Remark: All the sentences above are entered in "cmd".

The Construction of Models

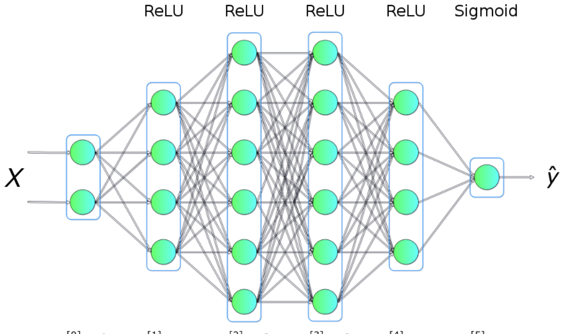
- layers
- optimizer
- loss
- metrics, Connection, compile
- Training
- Testing

Dense

layer=tf.keras.layers.
Dense(units,input_shape(batch_size,input_dim),activation=func_name)

$$output = activation(kernel \cdot input + bias).$$

The dimension of kernel: units$\times$input_dim; the dimension of bias: units$\times 1$.
The example of activation: 'relu', 'sigmoid', 'softmax', 'tanh'.



Conv2D

layer=
tf.keras.layers.Conv2D(input_shape(rows,cols,channels),filters=num, kernel_size=(rows,cols),strides=(s_1,s_2),padding,activation)

"filters" is the number of output filters;

"padding" has two values: 'valid' and 'same'. If 'valid', left columns or rows will be abandoned. If 'same', zeros will be supplied to make the sizes match.

Example: Assume input_shape(2,3,1), filters=1, kernel_size=(2,2), strides=(1,1). If padding='valid', the size of output is (1,2,1); if padding='same', the size of output is (1,3,1).

1	2	3
4	5	6

1	2	3	0
4	5	6	0

MaxPool2D

layer=tf.keras.layers.MaxPool2D(pool_size=(rows,cols),strides=(s_1,s_2),padding)
"pool_size" is similar to "kernel_size"; "strides" and "padding" are the same in "Conv2D".

Flatten

Flattens the input.

Example: input_shape of Flatten=(None,32,32,3),
output_shape=$32 \times 32 \times 3 = 3072$

optimizer=tf.keras.optimizers.schedules_namespace()
Generally used schedules_namespace:

- Adagrad
  learning_rate=0.001, initial_accumulator_value=0.1, epsilon=10^{-07}

$$accum_{n+1} = accum_n + g_n^2; accum_0 = initial\_accumulator\_value;$$
$$\theta_{n+1} = \theta_n - learning\_rate \frac{g_n}{\sqrt{accum_{n+1}} + epsilon}$$

- RMSprop
  learning_rate=0.001, rho=0.9, epsilon=10^{-07}

$$mon_{n+1} = rho \cdot mon_n + (1-\rho) g_n \odot g_n;$$
$$\theta_{n+1} = \theta_n - learning\_rate \frac{g_n}{\sqrt{mon_{n+1} + epsilon}}$$

- Adam
  learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=10^{-07}

$$m_{n+1} = beta\_1 m_n + (1 - beta\_1) g_n;$$
$$v_{n+1} = beta\_2 v_n + (1 - beta\_2) g_n \odot g_n;$$
$$\hat{m}_{n+1} = \frac{m_{n+1}}{1 - beta\_1^{n+1}}; \hat{v}_{n+1} = \frac{v_{n+1}}{1 - beta\_2^{n+1}};$$
$$\theta_{n+1} = \theta_n - learning\_rate \frac{\hat{m}_{n+1}}{\sqrt{\hat{v}_{n+1}} + epsilon}$$

- SGD
  learning_rate=0.01, momentum=0.0, nesterov=False

$$v_{n+1} = momentum \cdot v_n - learning\_rate \cdot g_n$$
$$\theta_{n+1} = \theta_n + v_{n+1}$$

  - g_n is evaluated at θ_n if nesterov=False;
  - g_n is evaluated at $\theta_n + momentum \cdot v_n$ if nesterov=True

The Construction of Models – loss

loss=tf.keras.losses.class_name() or 'func_name'

Class_name	func_name
CategoricalCrossentropy	categorical_crossentropy
MeanAbsoluteError	mae
MeanSquaredError	mse

Table: Generally used built-in loss class_name and func_name

- Self-defined loss function

```
def custom_loss(y_actual,y_pred):
    custom_loss=f(y_actual,y_pred)
    return custom_loss
```

The Construction of Models – metrics, Connection, compile

- metrics
  metrics=['metrics1', 'metrics2',...]
  Generally used built-in metrics:
  'accuracy', 'precision', 'recall', 'mean', 'mae', 'mse'
- Connection
  - model=tf.keras.Sequential([layer1,layer2,...])
  - model.add(layer)
- compile
  model.compile(optimizer=, loss=, metrics=)

The Construction of Models – Training

model.fit(train_data, true_result, epoch=num, verbose=,validation_split=,callback=[])
"**epoch**" is the count that you train the entire training dataset.
"**verbose**" has three values: 0=silent; 1(default)=process bar; 2=one line per epoch.
"**validation_split**" is the fraction of the training data to be used as validation data and takes value between 0 and 1.
"**callback**" can be many classes: EarlyStopping, ModelCheckpoint, Tensorboard or self-defined class.

Example: EarlyStopping: Stop training when a monitored quantity has stopped improving.
tf.keras.callbacks.EarlyStopping(monitor=, patience=num)
"**monitor**"='loss','val_loss'(default),'metric1','val_metric1','metric2' or 'val_metric2',...
"**patience**": Number of epochs with no improvement after which training will be stopped.

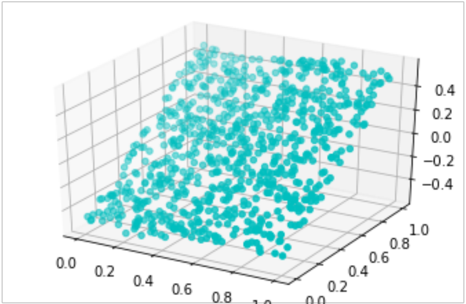
The Construction of Models – Testing

- model.evaluate(test_data,test_labels) Given the test_data, the trained model will return predicted values. Use the "loss" and "metrics" of this model to compare the predicted labels and test_labels.
- model.predict(x_test) Generates output predictions for the input x_test after training the model.

Comparison to model.fit:
model.fit is to train the model by using the given data. The biases and kernels of all layers will be changed based during the training.

model.evaluate and model.predict are used to do the calculations for the given data based on the model. The parameters for each layer will not change.

The Regression Problem



For a set of given 3-d data (x, y, z). (x, y, z) are 2-d data on $[0, 1] \times [0, 1]$. z is the value $y - (x^3 - 3/2x^2 + x/2 + 1/2)$. We will firstly use deep learning to study the relationship between the set of 2-d data (x, z) and y. Then, we test another set of (x, z) on the model by comparing the predictions with the true results y. Finally, we use the model to build up the predict function for $f(x) = x^3 - 3/2x^2 + x/2 + 1/2$.

The Solution to the Regression Problem

- Get Dataset
- Preprocess Data
- Construct Mode
- Train Mode
- Predict

The Solution to the Regression Problem – Get Dataset

https://blog.tensorflow.org/2019/02/introducing-tensorflow-datasets.html
https://www.tensorflow.org/datasets/catalog/overview (include MNIST)

```
import random
import xlwt

N=1000
wb=xlwt.Workbook()
sh1=wb.add_sheet('random')
sh1.write(0,0,'x')
sh1.write(0,1,'y')
sh1.write(0,2,'y\'s location')
for n in range(N):
    x=random.uniform(0,1)
    y=random.uniform(0,1)
    sh1.write(n+1,0,x)
    sh1.write(n+1,1,y)
    sh1.write(n+1,2,y-(x**3-3/2*x**2+x/2+1/2))
wb.save('data.xls')

column_names=['x','y','y\'s location']
raw_dataset=pd.read_excel('data.xls',
    names=column_names,na_values='?',comment='\t',
    sep=' ', skipinitialspace=True)
dataset=raw_dataset.copy()
dataset.tail()
```

To get the data, we firstly generate 1000 (any larger number) 2-d data in $[0,1]\times[0,1]$. Using the given function $f(x)=x^3-3/2x^2+x/2+1/2$, we can calculate out the distance $y-f(x)$ for each 2-d data. For future use, we can store this set of data in an excel document (data.xls).

Use pandas to get the data from the excel document.

	x	y	y's location
995	0.572571	0.055339	-0.426900
996	0.305493	0.869970	0.328702
997	0.213493	0.290798	-0.257310
998	0.824746	0.179505	-0.273556
999	0.853694	0.039483	-0.416341

```
dataset.head()
```

	x	y	y's location
0	0.893394	0.268033	-0.194500
1	0.513924	0.638644	0.142122
2	0.945637	0.082966	-0.394125
3	0.083798	0.809073	0.277119
4	0.152412	0.265328	-0.279575

The Solution to the Regression Problem – Preprocess Data

- Cleanse data

```
print(dataset.isna().sum())
```

```
x              0
y              0
y's location   0
dtype: int64
```

If one number is 0, use "dropna()" to get rid of this data.

- Divide data into training data and test data.

```
train_set=dataset.sample(frac=0.8,random_state=0)
test_set=dataset.drop(train_set.index)
```

```
train_set
```

	x	y	y's location
993	0.681438	0.228846	-0.231767
859	0.821453	0.278823	-0.174031
298	0.572378	0.907255	0.424970
553	0.247500	0.366957	-0.180070
672	0.959936	0.873960	0.391649
...	...	...	...
117	0.436139	0.415489	-0.100215
464	0.491365	0.160529	-0.341629
25	0.429366	0.986881	0.469574
110	0.208062	0.973866	0.425763
149	0.079867	0.301288	-0.229587

800 rows × 3 columns

```
test_set
```

	x	y	y's location
9	0.580650	0.052879	-0.427483
11	0.153235	0.849716	0.304721
19	0.593909	0.436602	-0.040749
23	0.506576	0.515446	0.017089
28	0.514244	0.516358	0.019916
...	...	...	...
962	0.692922	0.245064	-0.213886
966	0.515542	0.296375	-0.199743
976	0.785837	0.363851	-0.088044
980	0.274649	0.752311	0.207418
983	0.668194	0.672113	0.209404

200 rows × 3 columns

- Choose one variable as label and take labels

```
train_labels=train_set.pop('y')
test_labels=test_set.pop('y')
```

- Normalize data

```
train_stats=train_set.describe()
train_stats=train_stats.transpose()
train_stats
```

	count	mean	std	min	25%	50%	75%	max
x	800.0	0.491393	0.286322	0.001064	0.238717	0.494064	0.731351	0.999930
y's location	800.0	0.004010	0.292721	-0.540594	-0.247227	0.015060	0.255846	0.544757

Figure: Get the statistic information of x, y

```
def norm(x):
    return (x-train_stats['mean'])/train_stats['std']
norm_train_set=norm(train_set)
norm_test_set=norm(test_set)
```

Figure: Normalize data

Why do we normalize the data before training?

The Solution to the Regression Problem – Construct Mode

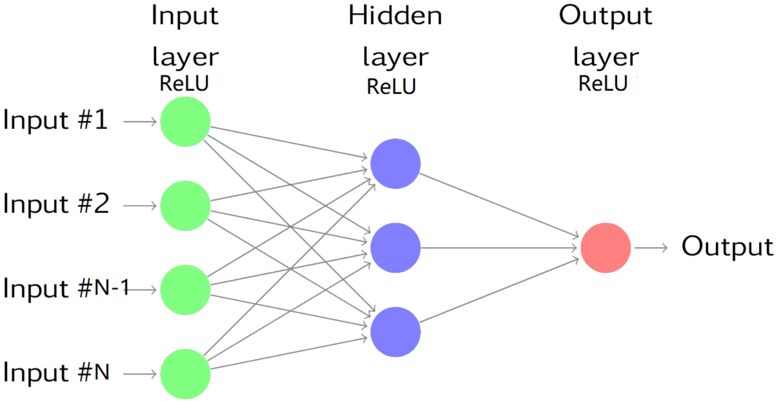


Figure: The basic model of neural network

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

def build_model():
    model=keras.Sequential([
    layers.Dense(64,activation='relu',input_shape=[len(train_set.keys())]),
    layers.Dense(64,activation='relu'),
    layers.Dense(1,activation='relu')])
    opt=keras.optimizers.Adam(0.001)
    model.compile(loss='mse',optimizer=opt,metrics=['mse'])
    return model
model=build_model()
model.summary()
```

Building up the deep learning model. Commonly, we start from 3-layer network.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                192
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 4,417
Trainable params: 4,417
Non-trainable params: 0
_________________________________________________________________
```

Check the construction details of model.

Q: How to calculate the number of the parameters in each layer?

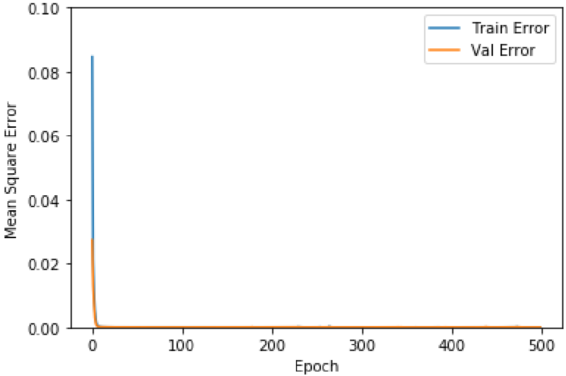
Self-defined callback function: A '·' will be output every epoch end. Every 100 '·' will start a new line.

```
class PrintDot(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        if epoch%100==0: print('\n')
        print('.', end='')
```

- 1st train

```
>>> Epochs=500
>>> History=model.fit(norm_train_set, train_labels, epochs=Epochs,
...                     validation_split=0.2, verbose=0, callbacks=[PrintDot()])

....................................................................................................

....................................................................................................

....................................................................................................

....................................................................................................

....................................................................................................
```

```python
hist=pd.DataFrame(History.history)
hist['epoch']=History.epoch

def plot_history(hist):
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Mean Square Error')
    plt.plot(hist['epoch'],hist['mse'],label='Train Error')
    plt.plot(hist['epoch'],hist['val_mse'],label='Val Error')
    plt.ylim([0,0.2])
    plt.legend()
    plt.show()

plot_history(hist)
```



- 2nd train

  From the graph, we can see that after epoch=20, the Train Error will keep small but the Val Error will raise. So, we should stop train earlier or try other neural networks. If we choose to stop earlier, we will use callback function "EarlyStopping".

```
early_stop=keras.callbacks.EarlyStopping(monitor='val_loss',patience=20)
History=model.fit(norm_train_set,train_labels,epochs=Epochs,
                  validation_split=0.2,verbose=0,callbacks=[PrintDot(),early_stop])
```

  But in our case, we just simply change epoch from 500 to 150.

```
model=build_model()
Epochs=150
History=model.fit(norm_train_set,train_labels,epochs=Epochs, validation_split=0.2,verbose=0,callbacks=[PrintDot()])
hist=pd.DataFrame(History.history)
hist['epoch']=History.epoch
hist.tail()
```

```
....................................................................................................
..................................................
```

	loss	mse	val_loss	val_mse	epoch
145	0.000006	0.000006	0.000008	0.000008	145
146	0.000004	0.000004	0.000006	0.000006	146
147	0.000004	0.000004	0.000005	0.000005	147
148	0.000003	0.000003	0.000004	0.000004	148
149	0.000003	0.000003	0.000007	0.000007	149

The Solution to the Regression Problem – Predict

```python
def prediction(norm_test_set, test_labels):
    test_predictions=model.predict(norm_test_set).flatten()
    plt.scatter(test_labels, test_predictions)
    plt.xlabel('True Value')
    plt.ylabel('Predictions')
    plt.axis('equal')
    plt.axis('square')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    _ = plt.plot([-100, 100], [-100, 100])
    plt.show()
prediction(norm_test_set, test_labels)
```

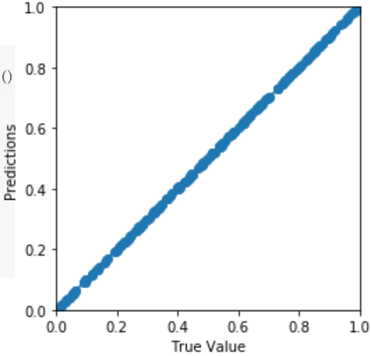


Figure: The comparison between predicted value for y with the real value for y

We use our trained model to find the predicted labels for the normalized test data and then compare it with the real labels.

We can also paint the error distribution.

```
test_predictions=model.predict(norm_test_set).flatten()
error=test_predictions-test_labels
plt.hist(error,bins=50)
plt.xlabel('Prediction error')
_=plt.ylabel('Count')
plt.show()
```

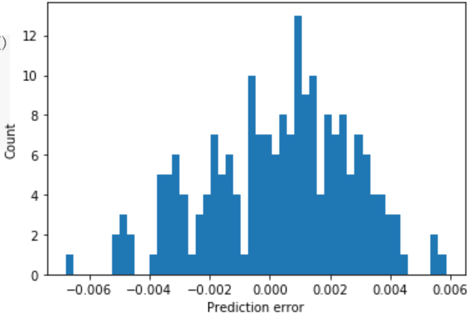


Figure: The error distribution between the predicted labels and the real labels for test data.

We can also use this model to get an approximate function of $f(x) = x^3 - 3/2x^2 + x/2 + 1/2$.

```
def func_y_x(x):
    x_new=(x-train_stats['mean']['x'])/train_stats['std']['x']
    c_new=(0-train_stats['mean']['y\'s location'])/train_stats['std']['y\'s location']
    xd=pd.DataFrame({'x':pd.Series([x_new],index=[0]),'y\'s location':pd.Series([c_new],index=[0])})
    y_pre=model.predict(xd).flatten()
    return [y_pre[0],x**3-3/2*x**2+x/2+1/2,abs(y_pre[0]-(x**3-3/2*x**2+x/2+1/2))]
```

Figure: We firstly normalize $(x, y'slocation)$ and then use the trained model to predict y.

x	func_y_x	f(x)	Absolute Error
0	0.5041	0.5	0.0041
1	0.4963	0.5	0.0037
0.5	0.5023	0.5	0.0023
0.25	0.5465	0.5469	0.0004
0.333	0.5404	0.5371	0.0034
0.45	0.5099	0.5124	0.0025
0.245	0.5465	0.5472	0.0007
0.765	0.4519	0.4524	0.0005

The Solution to the Regression Problem – Conclusion

1. The regression problem is very similar to the classification problems. If we choose "y's location" as the labels, use 'a' to represent the point (x, y) above or on the graph of $f(x)$ and 'b' to represent (x, y) below the $f(x)$ and hope to classify given 2-d (x, y) into 'a' or 'b' class, our problem is a classification problem. While, if we choose y as the labels, it will become a regression problem.
2. There is no restriction about the amounts of layers and nodes for each hidden layers. Generally, the deeper the network is, the better the result is. For the amount of nodes in each layer, we just let it smaller than the capacity of the training data.

The Image Recognition Problem

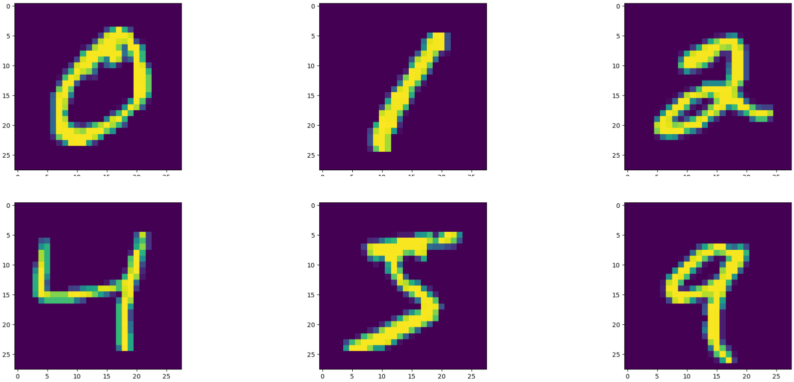


Figure: MNIST dataset contains lots of images of numbers. The neural network will recognize the numbers on the images by learning the data from this set.

The Solution

- Prepare Dataset

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

>>> (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
>>> print(x_train.shape, ' ', y_train.shape)
(60000, 28, 28)   (60000,)
>>> print(x_test.shape, ' ', y_test.shape)
(10000, 28, 28)   (10000,)
```

Get data from MINST. The data from MINST doesn't have the dimension for channel.

```
>>> x_train = x_train.reshape((-1, 28, 28, 1))
>>> x_test = x_test.reshape((-1, 28, 28, 1))
>>> print(x_train.shape, ' ', y_train.shape)
(60000, 28, 28, 1)   (60000,)
```

Transfer the raw data from MINST to the data whose form can be processed by Con2D.

Why don't we normalize the data here?

- Construct Model

```
model = keras.Sequential()
model.add(layers.Conv2D(input_shape=(x_train.shape[1], x_train.shape[2], x_train.shape[3]),
                        filters=32, kernel_size=(3,3), strides=(1,1), padding='valid',
                       activation='relu'))
model.add(layers.MaxPool2D(pool_size=(2,2)))
model.add(layers.Flatten())
model.add(layers.Dense(32, activation='relu'))

model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer=keras.optimizers.Adam(),
             loss=keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 5408)              0
_________________________________________________________________
dense (Dense)                (None, 32)                173088
_________________________________________________________________
dense_1 (Dense)              (None, 10)                330
=================================================================
Total params: 173,738
Trainable params: 173,738
Non-trainable params: 0
_________________________________________________________________
```

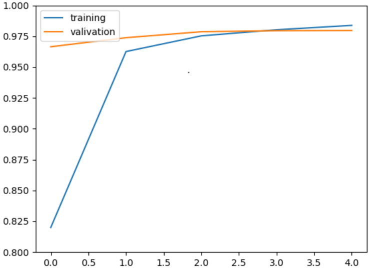
Q:Why the output shape of Conv2D is (None, 26, 26, 32)?

Real Ques: Why the amount of parameters of Conv2D is 320?

- Training

```
>>> history = model.fit(x_train, y_train, batch_size=64, epochs=5, validation_split=0.1,verbose=2)
Train on 54000 samples, validate on 6000 samples
Epoch 1/5
54000/54000 - 16s - loss: 0.7054 - accuracy: 0.8199 - val_loss: 0.1530 - val_accuracy: 0.9665
Epoch 2/5
54000/54000 - 15s - loss: 0.1431 - accuracy: 0.9626 - val_loss: 0.1077 - val_accuracy: 0.9738
Epoch 3/5
54000/54000 - 15s - loss: 0.0883 - accuracy: 0.9753 - val_loss: 0.0881 - val_accuracy: 0.9787
Epoch 4/5
54000/54000 - 15s - loss: 0.0657 - accuracy: 0.9803 - val_loss: 0.0818 - val_accuracy: 0.9795
Epoch 5/5
54000/54000 - 15s - loss: 0.0523 - accuracy: 0.9838 - val_loss: 0.0909 - val_accuracy: 0.9797
```

```
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['training', 'valivation'], loc='upper left')
plt.ylim([0.8,1])
plt.show()
```



- Testing

```
>>> res = model.evaluate(x_test, y_test)
10000/10000 [==============================] - 1s 113us/sample - loss: 0.1026 - accuracy: 0.9737
```

The End