

# **SDSC 3006 L02**

## **Class 10. Deep Learning**

Name: Yiren Liu

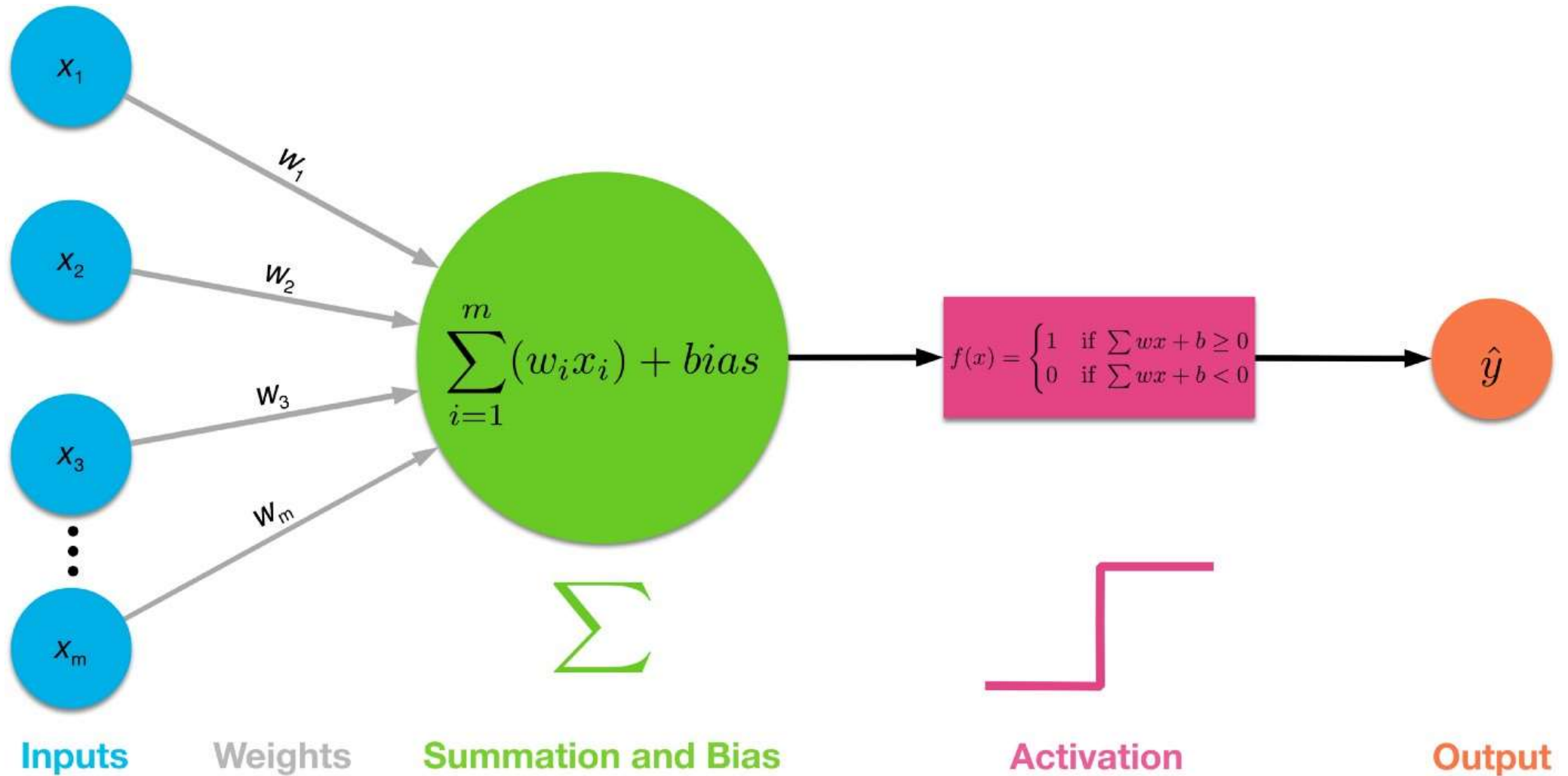
Email: [yirenliu2-c@my.cityu.edu.hk](mailto:yirenliu2-c@my.cityu.edu.hk)

School of Data Science  
City University of Hong Kong

# Outline

- **Single Layer Network (linear)**
- **Multilayer Network (non-linear)**
- **Convolution Neural Network (Image Analysis)**

# Single Neural Network



# Breakthrough

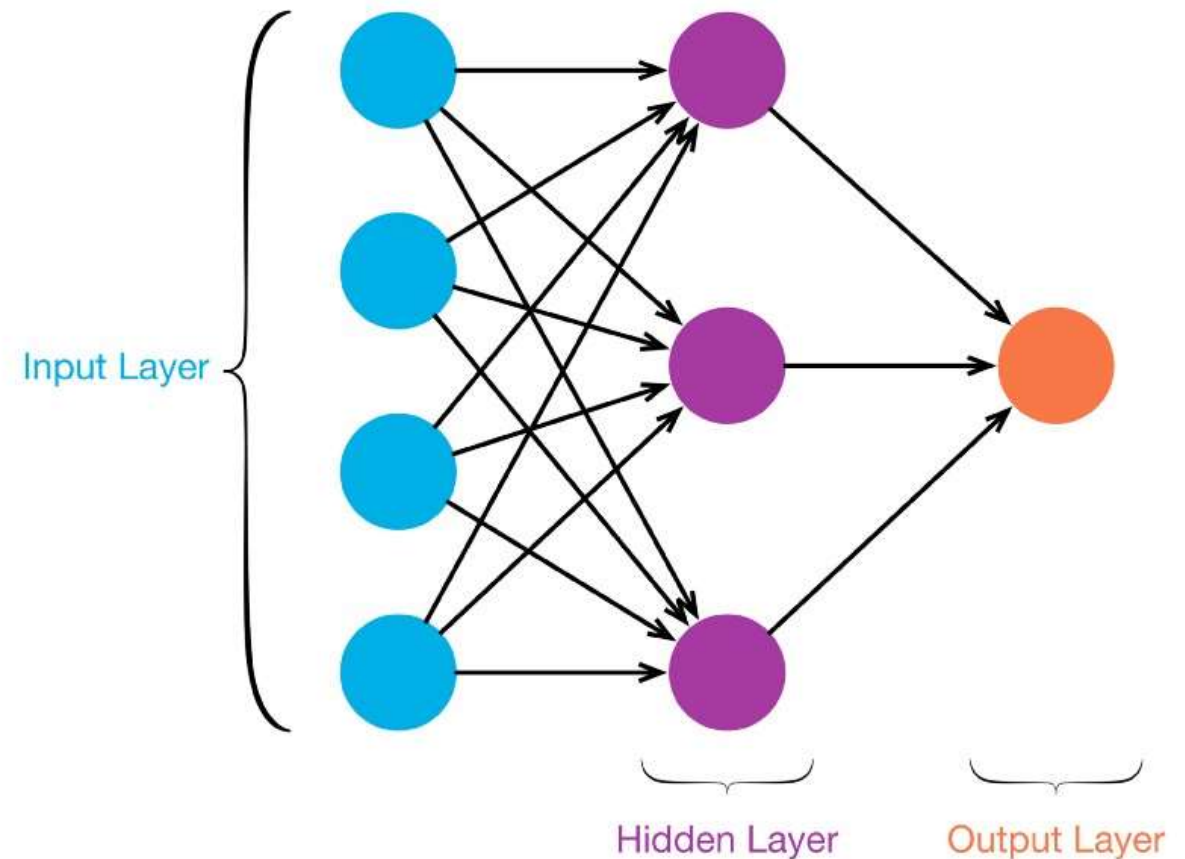
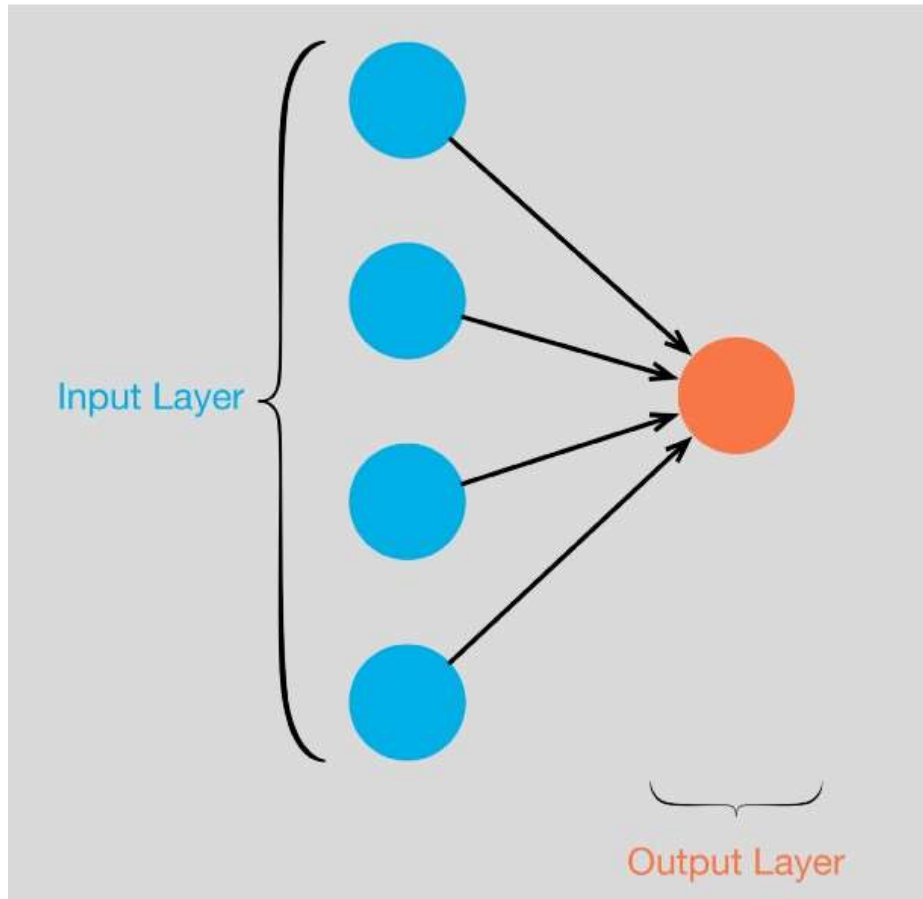
## Breakthrough: Multi-Layer Perceptron

Fast forward almost two decades to 1986, Geoffrey Hinton, David Rumelhart, and Ronald Williams published a paper “*Learning representations by back-propagating errors*”, which introduced:

1. **Backpropagation**, a procedure to *repeatedly adjust the weights* so as to minimize the difference between actual output and desired output
2. **Hidden Layers**, which are *neuron nodes stacked in between inputs and outputs*, allowing neural networks to learn more complicated features (such as XOR logic)

Ref: <https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f>

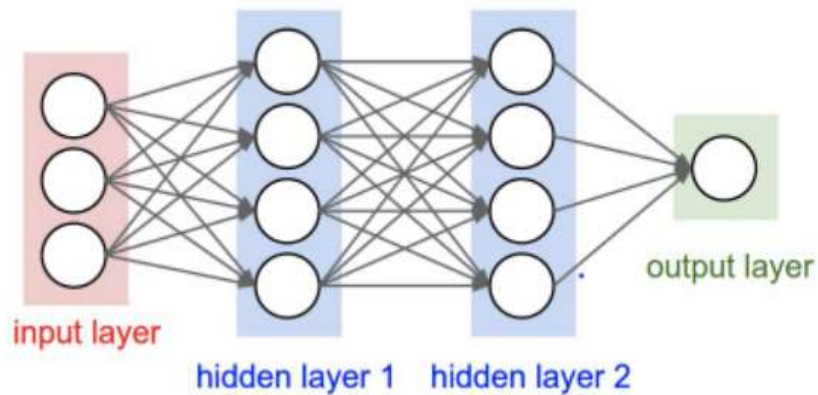
# Single layer to Multilayer Network



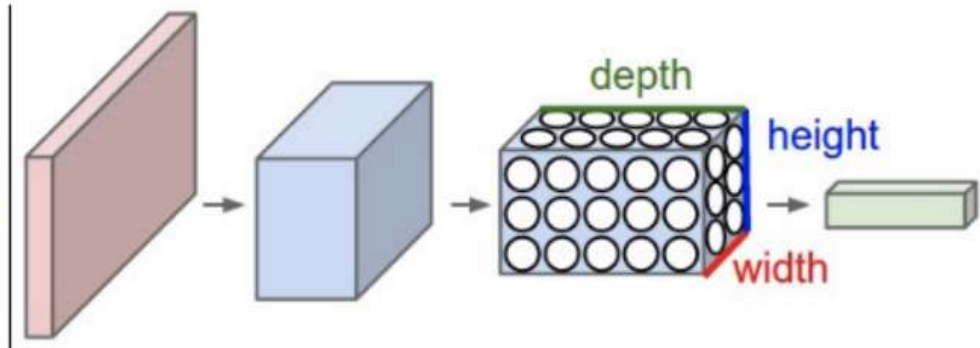
# Drawbacks of MLP

- There are several drawbacks of MLP's, especially when it comes to image processing. MLPs use one perceptron for each input (e.g. pixel in an image, multiplied by 3 in RGB case). The amount of weights rapidly becomes unmanageable for large images, e.g. 224 x 224 pixel image with 3 color channels there are around 150,000 weights that must be trained!
- Another common problem is that MLPs react differently to an input (images) and its shifted version — they are not translation invariant.

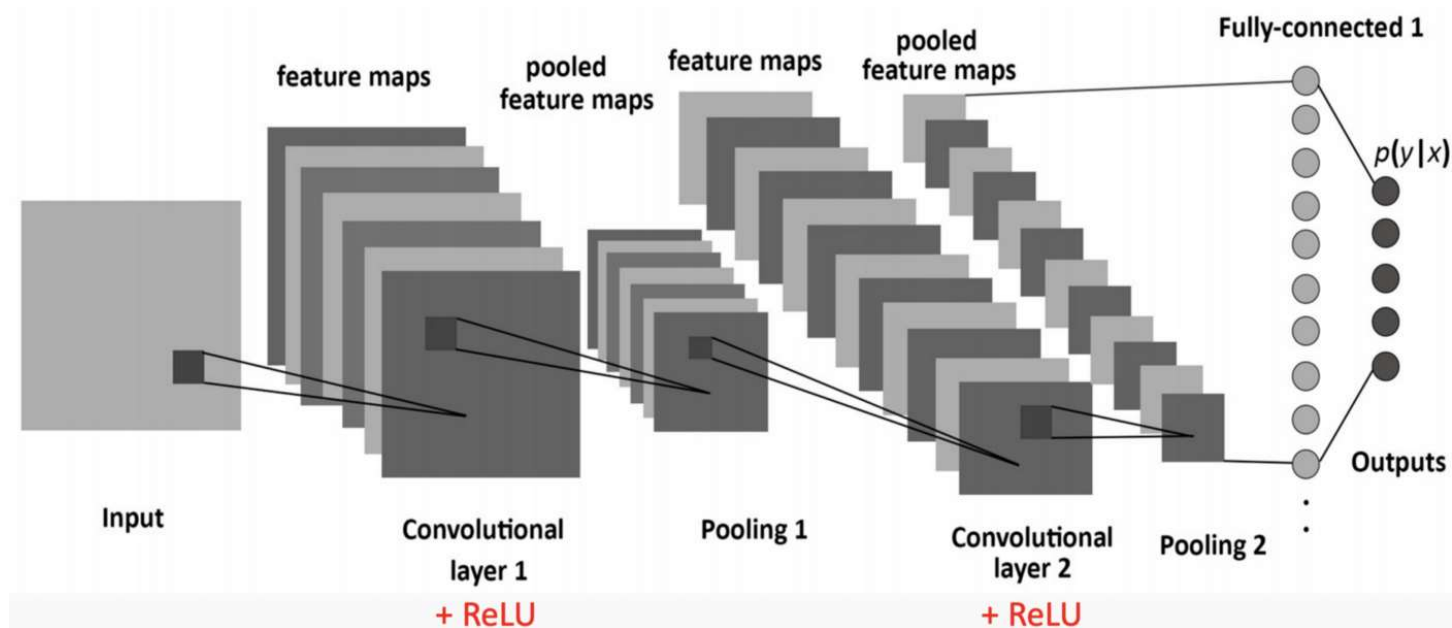
# MLP to CNN



MLP



CNN





# Keys of CNN

- There are three types of layers in a convolutional neural network (Each of these layers has different parameters that can be optimized):
- **Convolutional layers** are the layers where filters are applied to the original image, or to other feature maps in a deep CNN. This is where most of the user-specified parameters are in the network. The most important parameters are the number of **kernels** and the size of the kernels.
- **Pooling layers** are similar to convolutional layers, but they perform a specific function such as max pooling, which takes the maximum value in a certain filter region, or average pooling, which takes the average value in a filter region. These are typically used to **reduce the dimensionality** of the network.
- **Fully connected layers** are placed before the classification output of a CNN and are used to flatten the results before classification. This is similar to the output layer of an MLP.

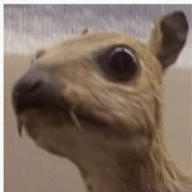
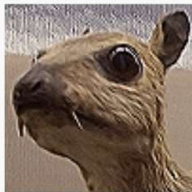


Convolution: made up of a large number of convolution filters(a little matrix).

*Edge detection*

 \*  $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$  =  ← Kernel

*Sharpen*

 \*  $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$  = 

Max Pooling: condense a large image into a smaller pooling summary image.

$$\text{Max Pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

# How to Work Deep Learning Networks

- A deep learning model is composed of one input layer, two or more hidden layers, and one final layer. The input layer receives input data and passes the inputs to the first hidden layer. The hidden layers perform mathematical computations on inputs.

# Details

The “Deep” in Deep Learning refers to having more than one hidden layer. The output layer returns the output data. Each connection between neurons is associated with a weight. This weight shows the importance of the input value. The initial weights are set randomly. Each neuron has an activation function. Once a batch of input data has passed through all the layers of the neural network, it returns the output data through the output layer.

**The loss or cost function** measures the difference between the actual result and the predicted result. To build a good model, you want your loss function to be close to zero. To find the minimum cost function, you use the gradient descent technique. Gradient descent works by changing the weights in small increments after each data set iteration.

Reference: <https://tensorflow.rstudio.com/tutorials/quickstart/beginner>  
[https://keras.io/getting\\_started/](https://keras.io/getting_started/)

# Deep Learning in R

- Using TensorFlow with *keras* (*seems not work on MacOS with M1 chip, please watch video of Lab01 for details*)
- The '*deepnet*' package
- The '*nnet*' package
- The '*neuralnet*' package

**Watch Lab01 Video for the  
implementation of following  
codes**

# Single Layer Network

# Introduction

- Layers: Input(1), Hidden(?), Output(1).
- Activation function: a non-linear function, such as sigmoid, tanh and ReLU.
- Loss: a measurement of NN, less is better. For instance, squared-error and cross-entropy.
- One-hot: a vector with one in the position related to its class and zeros elsewhere.

# Implementation

```
#set up data and separate it
library (ISLR2)
Gitters = na.omit(Hitters)
n = nrow(Gitters)
set.seed (13)
ntest = trunc(n/3)
testid = sample(1:n, ntest)
x = scale(model.matrix(Salary~.-1, data = Gitters))
y = Gitters$Salary
#linear model for comparison
lfit = lm(Salary~., data = Gitters[-testid , ])
lpred = predict (lfit , Gitters[testid , ])
with (Gitters[testid , ], mean (abs (lpred - Salary)))
```



# Implementation

##installation steps of tensorflow and keras: see lab02

##create model structure of NN using keras

```
library(keras)
```

```
modnn = keras_model_sequential() %>%
```

```
  layer_dense(units=50, activation="relu", input_shape=ncol(x)) %>%
```

```
  layer_dropout(rate=0.4) %>%
```

```
  layer_dense(units=1)
```

##pipe operator %>% passes the previous term as the first argument to the next function

##dropout: randomly drop the activations from previous layer(set to 0)

```
modnn %>% compile (loss = "mse", optimizer = optimizer_rmsprop(), metrics = list  
("mean_absolute_error"))
```

**##compile: not change R object, but relates to python**

# Implementation

##plot the history to display the mae for the training and test data

```
history = modnn %>% fit(x[-testid,], y[-testid], epochs=200, batch_size=32,  
validation_data = list(x[testid,], y[testid]))
```

##batch\_size: randomly choose training obs for SGD

```
install.packages('ggplots')  
plot (history)
```

##calculate

```
mae npred =predict (modnn , x[testid , ])  
mean ( abs (y[testid] - npred))
```

# **Multilayer Network**

# Implementation

```
mnist = dataset_mnist()
```

```
librag_train = mnist$train$y
```

```
x_train = mnist$train$x
```

```
g_train = mnist$train$y
```

```
x_test = mnist$test$x
```

```
g_test = mnist$test$y
```

```
dim(x_train)
```

```
dim(x_test)
```

```
# reshape the array into matrix and "one-hot"
```

```
x_train = array_reshape(x_train, c(nrow(x_train), 784))
```

```
x_test = array_reshape(x_test, c(nrow(x_test), 784))
```

```
y_train = to_categorical(g_train, 10)
```

```
y_test = to_categorical(g_test, 10)
```

```
# rescale the original data(0~255) into unit interval
```

```
x_train = x_train / 255
```

```
x_test = x_test / 255
```

# Implementation

## ##structure of NN

```
modelnn = keras_model_sequential()  
modelnn %>%  
  layer_dense(units=256, activation="relu", input_shape=c(784)) %>%  
  layer_dropout(rate=0.4) %>%  
  layer_dense(units=128, activation="relu") %>%  
  layer_dropout(rate=0.3) %>%  
  layer_dense(units=10, activation="softmax")
```

## ##summary(modelnn)

# Implementation

## ##structure of NN

```
modelnn %>% compile(loss="categorical_crossentropy",  
optimizer=optimizer_rmsprop(), metrics=c("accuracy"))
```

## ##supply training data, and fit the model

```
system.time(history <- modelnn %>%  
  fit(x_train, y_train, epochs=30, batch_size=128, validation_split=0.2))  
plot(history, smooth=FALSE)
```

## ##evaluate the model by caculate the accuracy

```
accuracy = function (pred, truth)  
  mean(drop (pred) == drop (truth))  
modelnn %>% predict_classes(x_test) %>% accuracy(g_test)
```