Probabilistic Deep Learning:

5. Probabilistic Neural Network 1

ESC 2024 Winter Session 1주차

Contents

1

Fully Connected Neural Networks (fcNNs)

Probablistic Neural Network 1

Various types of DL layers

딥러닝은 여러 Hidden layer들이 쌓여 이루어진다.
이때, 쓰일 수 있는 layer에는 여러 type이 존재한다.
1) fully-connected NNs, 2) convolutional NNs 3) 1D convolutional 등등…
실제 딥러닝 모델은 이러한 다양한 layer들과 그들의 결합으로 이루어진다.
따라서, 우리는 해결하고자 하는 문제에 따라 **적절한 type의 layer를 선택해야 한다**.

Ex)
-Excel sheet와 같이 local structrue가 없는 경우
→ fully-connected neural network(fcNN)이 적합

-이미지 데이터와 같이 local structure를 가지는 경우
→ convolutional neural network(CNN)이 적합

-text와 같이 sequential한 데이터의 경우
→ 1D-CNN이 적합

이어지는 1장과 2장에서는 가장 대표적인 fcNN, CNN의 구조와 사용하는 이유를 살펴보고 간단한 코딩을 통해 실습을 해볼 것!

Fully Connected Neural Networks(fcNNs)

- Hidden layer의 각각의 neuron(=hidden unit)이 다음 layer의 모든 neuron과 연결되기 때문에 Fully Connected NN으로 불림

-인간 뉴런 세포로부터 영감을 받았음
(뉴런들은 서로 다른 수많은 뉴런들과 연결되어 있으며, 이전 뉴런으로부터 신호를 받아 다음 뉴런으로 전달함. 신호를 전달하는 방식은 학습과 경험 등을 통해 바뀔 수 있음 등등...)

-이전 layer 값들에 weight들이 곱해져 선형적으로 더해지고, 마지막으로 비선형함수인 activation function을 통과하여 output이 됨. 수식적으로 나타내면,

$$z = x_1 * w_1 + x_2 * w_2 + \cdots + x_p * w_p + 1 * bias$$

$$y = activation(z)$$

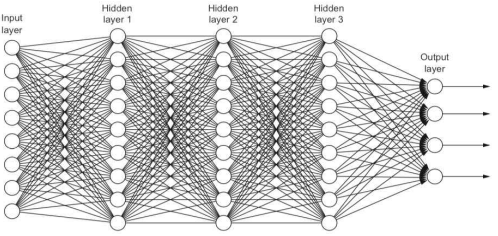


그림1. 3개의 hidden layer를 가진 fcNN을 도식화한 그림

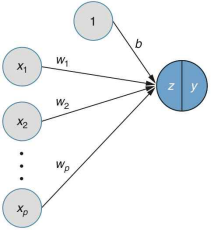
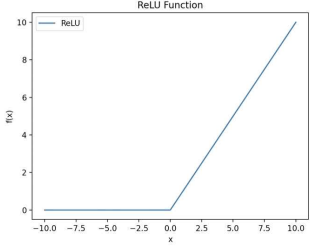


그림2. 이전 layer의 값으로부터 Output이 만들어지는 과정

Activation Function

activaion(z)가 activation function으로, 다양한 함수를 사용할 수 있다.

1. ReLU activtion

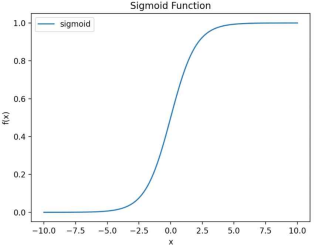


$$ReLU(z) = \max(0, z)$$

-함수에서 0보다 큰 부분만 출력함

-Sigmoid보다 계산이 빠르고, gradient의 소실이 없음

2. Sigmoid activtion



$$Sigmoid(z) = \frac{1}{1 + e^{z}}$$

-0과 1사이의 출력 범위를 가짐

-logistic regression, classifier등에 사용 가능

Fully Connected Neural Networks(fcNNs)

```
[1] #필요한 라이브러리 다운받기
    import numpy as np
    import matplotlib.pyplot as plt
    import math
```

```
[2] # ReLU function 정의하기
    def ReLU(preactivation):
      activation = np.clip(preactivation, 0, math.inf)
      return activation
```

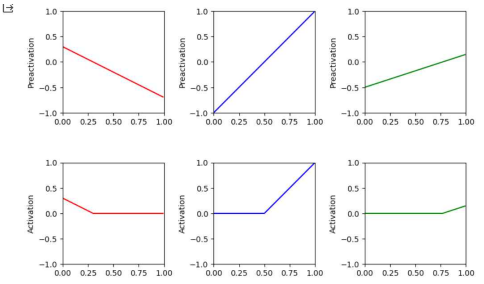
```
# 하나의 input, 하나의 Hidden layer를 가진 neural network 만들기
def shallow_1_1_3(x, activation_fn, phi_0,phi_1,phi_2,phi_3, theta_10, theta_11, theta_20, theta_21, theta_30, theta_31):

  pre_1 = theta_10+theta_11*x
  pre_2 = theta_20+theta_21*x
  pre_3 = theta_30+theta_31*x

  # Pass these through the ReLU function to compute the activations as in

  act_1 = activation_fn(pre_1)
  act_2 = activation_fn(pre_2)
  act_3 = activation_fn(pre_3)

  w_act_1 = phi_1*act_1
  w_act_2 = phi_2*act_2
  w_act_3 = phi_3*act_3

  y = phi_0+w_act_1+w_act_2+w_act_3

  return y, pre_1, pre_2, pre_3, act_1, act_2, act_3, w_act_1, w_act_2, w_act_3
```

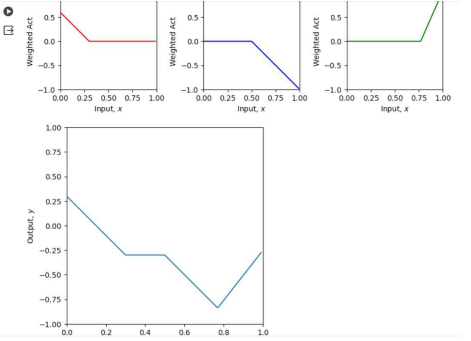
Fully Connected Neural Networks(fcNNs)

```
# Neural network를 plotting하는 함수 정의하기
def plot_neural(x, y, pre_1, pre_2, pre_3, act_1, act_2, act_3, w_act_1, w_act_2, w_act_3, plot_all=False, x_data=None, y_data=None):

  # Plot intermediate plots if flag set
  if plot_all:
    fig, ax = plt.subplots(3,3)
    fig.set_size_inches(8.5, 8.5)
    fig.tight_layout(pad=3.0)
    ax[0,0].plot(x,pre_1,'r-'); ax[0,0].set_ylabel('Preactivation')
    ax[0,1].plot(x,pre_2,'b-'); ax[0,1].set_ylabel('Preactivation')
    ax[0,2].plot(x,pre_3,'g-'); ax[0,2].set_ylabel('Preactivation')
    ax[1,0].plot(x,act_1,'r-'); ax[1,0].set_ylabel('Activation')
    ax[1,1].plot(x,act_2,'b-'); ax[1,1].set_ylabel('Activation')
    ax[1,2].plot(x,act_3,'g-'); ax[1,2].set_ylabel('Activation')
    ax[2,0].plot(x,w_act_1,'r-'); ax[2,0].set_ylabel('Weighted Act')
    ax[2,1].plot(x,w_act_2,'b-'); ax[2,1].set_ylabel('Weighted Act')
    ax[2,2].plot(x,w_act_3,'g-'); ax[2,2].set_ylabel('Weighted Act')

    for plot_y in range(3):
      for plot_x in range(3):
        ax[plot_y,plot_x].set_xlim([0,1]);ax[plot_x,plot_y].set_ylim([-1,1])
        ax[plot_y,plot_x].set_aspect(0.5)
      ax[2,plot_y].set_xlabel('Input, $x$');
    plt.show()

  fig, ax = plt.subplots()
  ax.plot(x,y)
  ax.set_xlabel('Input, $x$'); ax.set_ylabel('Output, $y$')
  ax.set_xlim([0,1]);ax.set_ylim([-1,1])
  ax.set_aspect(0.5)
  if x_data is not None:
    ax.plot(x_data, y_data, 'mo')
    for i in range(len(x_data)):
```

Fully Connected Neural Networks(fcNNs)

```
# parameter를 임의로 넣었음
theta_10 =  0.3 ; theta_11 = -1.0
theta_20 = -1.0  ; theta_21 = 2.0
theta_30 = -0.5  ; theta_31 = 0.65
phi_0 = -0.3; phi_1 = 2.0; phi_2 = -1.0; phi_3 = 7.0

# Define a range of input values
x = np.arange(0,1,0.01)

# We run the neural network for each of these input values
y, pre_1, pre_2, pre_3, act_1, act_2, act_3, w_act_1, w_act_2, w_act_3 = M
    shallow_1_1_3(x, ReLU, phi_0,phi_1,phi_2,phi_3, theta_10, theta_11, theta_20, theta_21, theta_30, theta_31)
# And then plot it
plot_neural(x, y, pre_1, pre_2, pre_3, act_1, act_2, act_3, w_act_1, w_act_2, w_act_3, plot_all=True)
```





Fully Connected Neural Networks(fcNNs)

Banknote가 faked인지 real인지 분류하는 classifier를
fcNN을 통해 구현해보자.
한 번은 Hidden layer 없이 단 하나의 neuron만을 이용해,
한 번은 Hidden layer를 이용해 network를 만들고
성능을 비교할 것이다.

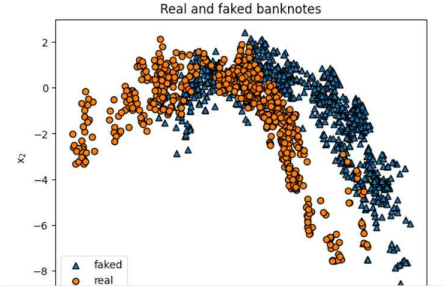
```
# 필요한 라이브러리 다운받기
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('default')

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers
```

```
# url로부터 데이터 읽어오기
from urllib.request import urlopen
url = 'http://archive.ics.uci.edu/ml/machine-learning
raw_data = urlopen(url)
dataset = np.loadtxt(raw_data, delimiter=",")
```

```
# 주어진 데이터 중 2개의 변수만 사용할 것
X=dataset[:,[1,3]]
# Y에는 True label(faked/real)을 저장
Y=dataset[:,4]
```

Fully Connected Neural Networks(fcNNs)

```
# 주어진 데이터를 시각화해보기
idx_f = [np.where(Y==1)]
idx_r = [np.where(Y==0)]
params = {'mathtext.default': 'regular' }  #Nicer Plotting (Latex Style)
plt.rcParams.update(params)
plt.scatter(X[idx_r,0],X[idx_r,1], alpha=1.0,marker='^',edgecolor='black')
plt.scatter(X[idx_f,0],X[idx_f,1], alpha=1.0,marker='o',edgecolor='black')
plt.title("Real and faked banknotes")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.legend(("faked","real"),loc='lower left',fontsize=10)
plt.show()
```

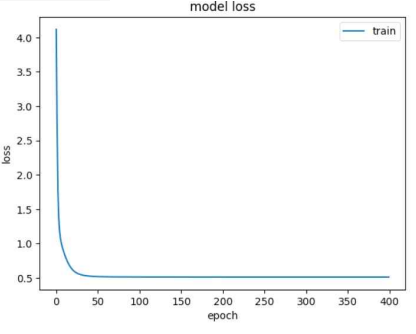


Fully Connected Neural Networks(fcNNs)

```
# 하나의 뉴런만을 가진 NN 만들기(Hidden layer가 없음)
model = Sequential()                          # sequential()로 모델 만들기 시작
model.add(Dense(1, batch_input_shape=(None, 2),    # 뉴런이 하나이기 때문에 Dense = 1, input size는 (batchsize, 2)
                activation='sigmoid'))        # activation function으로 sigmoid사용

sgd = optimizers.SGD(lr=0.15)                 # optimizer로 stochastic gradient descent사용, learning rate는 0.15로 설정

# 모델 컴파일링을 통해 모델 구축을 완료
model.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
```

```
# 네트워크 training 하기
history = model.fit(X, Y,       # model.fit을 통해 네트워크를 fitting
                    epochs=400,         # epoch(=training에서 iteration의 횟수)를 400으로 설정
                    batch_size=128,     # batch size를 128로 설정
                    verbose=0)
```

```
[10] # training이 진행되는 동안 accuracy와 loss의 변화 시각화하기
     plt.plot(history.history['accuracy'])
     plt.title('model accuracy')
     plt.ylabel('accuracy')
     plt.xlabel('epoch')
     plt.legend(['train'], loc='lower right')
     plt.show()
     plt.plot(history.history['loss'])
     plt.title('model loss')
     plt.ylabel('loss')
     plt.xlabel('epoch')
     plt.legend(['train'], loc='upper right')
     plt.show()
```



Fully Connected Neural Networks(fcNNs)

```
def plotModel(X,Y, model, t): # 학습된 decision boundary를 보여주는 함수 작성

    x1list = np.linspace(np.min(X[:,0])-2, np.max(X[:,0])+2, 50)
    x2list = np.linspace(np.min(X[:,1])-2, np.max(X[:,1])+2, 50)
    X1_grid, X2_grid = np.meshgrid(x1list, x2list) # np.meshgrid를 통해 그림을 그릴 2차원 공간 생성

    # 각각의 x1과 x2값에 대해 model의 예측값 생성
    p = np.array([model.predict(np.reshape(np.array([l1,l2]),(1,2))) for l1,l2 in zip(np.ravel(X1_grid), np.ravel(X2_grid))])
    print(p.shape)
    if len(p.shape) == 3 and p.shape[2]==2:
        p = p[:,:,1] # pick p for class 1 if there are more than 2 classes
    p = np.reshape(p,X1_grid.shape)

    # 2차원 공간 위에 decision boundary 그리기
    params = {'mathtext.default': 'regular' }
    plt.rcParams.update(params) #점점 update하며 그리기
    plt.figure(figsize=(16,4))

    plt.subplot(1,2,(1))
    cp = plt.contourf(X1_grid, X2_grid, p,cmap='viridis')
    plt.colorbar(cp)
    idx_f = [np.where(Y==1)]
    idx_r = [np.where(Y==0)]
    plt.scatter(X[idx_r,0],X[idx_r,1], alpha=1.0,marker='^',edgecolor='black') #위에 원자료 덧대기
    plt.scatter(X[idx_f,0],X[idx_f,1], alpha=1.0,marker='o',edgecolor='black')
    plt.title(t)
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')

plotModel(X, Y, model, 'fcnn separation without hidden layer') #위에서 정의한 함수 사용하여 시각화하기
```

Fully Connected Neural Networks(fcNNs)

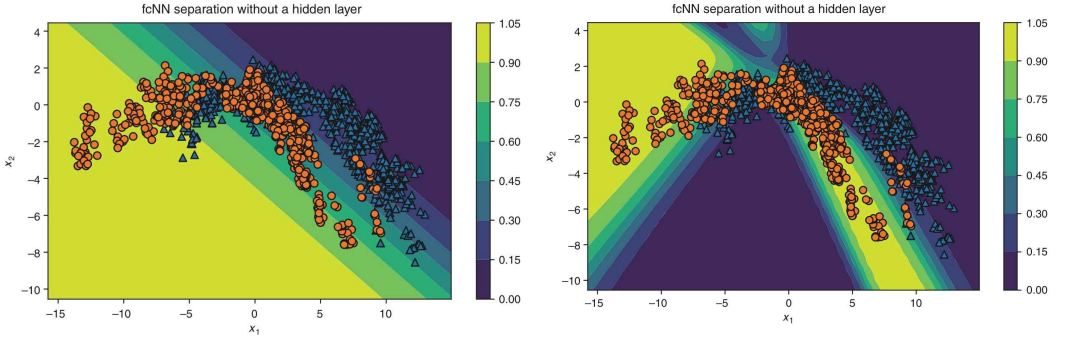
```
# Model을 training하기. 위와 동일하지만 이번에는 neuron이 8개인 Hidden layer와 outputlayer에도 2개의 neuron을 둘 것
model = Sequential() # outputlayer에도 2개의 neuron을 둘 것
model.add(Dense(8, batch_input_shape=(None, 2),activation='sigmoid')) # 이번에는 neron이 8개인 hidden layer이기 때문에 dense = 8
model.add(Dense(2, activation='softmax')) #나머지는 위의 환경과 동일하게 세팅

model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
```

```
# 이번에는 output layer에도 neuron이 두 개이기 때문에 Y의 형태를 바꿔줌
Y_c=to_categorical(Y,2)
Y[0:5], Y_c[0:5],Y[-5:-1],Y_c[-5:-1]
```

```
(array([0., 0., 0., 0., 0.]),
 array([[1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.],
        [1., 0.]], dtype=float32),
 array([1., 1., 1., 1.]),
 array([[0., 1.],
        [0., 1.],
        [0., 1.],
        [0., 1.]], dtype=float32))
```

```
# 위와 동일하게 network training
history = model.fit(X, Y_c,
          epochs=400,
          batch_size=128,
          verbose=0)
```

Fully Connected Neural Networks(fcNNs)



Hidden layer 없는 NN은 decision boundary가 선형인 것에 비해,

Hidden layer를 추가하니 비선형 decision boundary도 표현할 수 있는 것을 확인할 수 있다.

Fully Connected Neural Networks(fcNNs)

fcNN을 image classification 에 쓸 때:

Simple neural network는 2D input을 받을 수 없기 때문에,

그림을 1D 벡터로 펼쳐서 넣어야 함.

예를 들어, 28x28 픽셀 크기의 이미지를 분류한다면,

이를 28*28=784의 크기를 가지는 1D 벡터로 바꾸어야 함.

→ 학습해야 하는 parameter의 수가 기하급수적으로 늘어남.

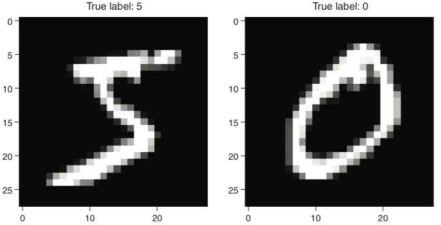


그림1. 분류하고자 하는 이미지(28*28 크기)

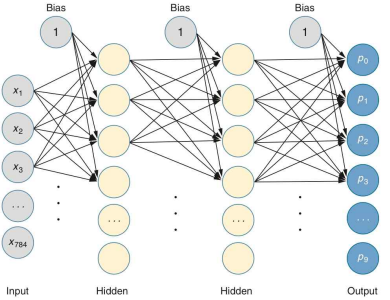


그림2. fcNN으로 연결한다면 parameter가 매우 많아진다.

Fully Connected Neural Networks(fcNNs)

FcNN을 통해 MNIST data set을 분석해 보자.

*MNIST data set: 아래 그림과 같이 0에서 9까지 10가지로 분류될 수 있는 손글씨 70,000개의 데이터.
이중 60,000개를 training에 사용하고 나머지 10,000개는 Test에 사용한다.

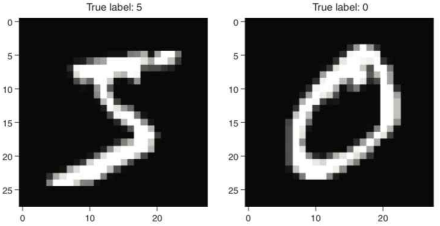


그림1. MNIST data set의 예시

```
# 마찬가지로 필요한 library 모두 다운받기
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('default')
from sklearn.metrics import confusion_matrix
import tensorflow as tf

import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Convolution2D, MaxPooling2D, Flatten , Activation
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers
```

다음으로 넘어가기 전, one-hot encoding이라는 개념을 알아야 한다.
최종적으로 결과는 해당 이미지가 0일 확률부터 9일 확률까지 총 10가지가 있기 때문에, 연산을 위해 True label 0에는 (1,0,0,…,0), 1에는 (0,1,0,…,0), …, 9에는 (0,0,0,…,1)을 부여하는 작업이 필요하다. 이것을 one-hot encoding이라고 한다.

Fully Connected Neural Networks(fcNNs)

```
from tensorflow.keras.datasets import mnist #MNIST로부터 data 불러오기
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# x_train에서 50000개까지를 training set,나머지 60000까지를 test set으로 한다.
X_train=x_train[0:50000] / 255 #모든 값을 255로 나누어 0과 1사이의 값으로 만들 것이다.
Y_train=keras.utils.to_categorical(y_train[0:50000],10) # one-hot encoding

X_val=x_train[50000:60000] / 255
Y_val=keras.utils.to_categorical(y_train[50000:60000],10)

X_test=x_test / 255
Y_test=keras.utils.to_categorical(y_test,10)

del x_train, y_train, x_test, y_test

X_train=np.reshape(X_train, (X_train.shape[0],28,28,1))
X_val=np.reshape(X_val, (X_val.shape[0],28,28,1))
X_test=np.reshape(X_test, (X_test.shape[0],28,28,1))
```

```
# fcNN을 사용하기 위해 데이터를 1차원 벡터 형태로 펼쳐야 함.

X_train_flat = X_train.reshape([X_train.shape[0], 784])
X_val_flat = X_val.reshape([X_val.shape[0], 784])
X_test_flat = X_test.reshape([X_test.shape[0], 784])
```

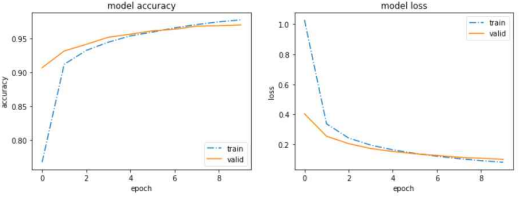
```
# 각각 100, 50개의 neuron으로 이루어진 hidden layer 2개를 가지는 fcNN을 만들 것.
model = Sequential()

model.add(Dense(100, batch_input_shape=(None, 784))) #100개의 neuron을 가지는 첫 번째 layer
model.add(Activation('sigmoid'))
model.add(Dense(50)) # 50개의 neuron을 가지는 두 번째 layer
model.add(Activation('sigmoid'))
model.add(Dense(10)) # output layer는 10 가지의 결과값을 가짐
model.add(Activation('softmax')) #확률들의 합을 1로 맞추기 위해 softmax함수를 사용

# compile model and intitialize weights
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
# 마찬가지로 모델 트레이닝
history=model.fit(X_train_flat, Y_train,
                  batch_size=128,
                  epochs=10,
                  verbose=2,
                  validation_data=(X_val_flat, Y_val)
                 )
```

Fully Connected Neural Networks(fcNNs)

```
# training 과정이 진행됨에 따라 accuracy와 loss의 변화를 시각적으로 확인해보자.
plt.figure(figsize=(12,4))
plt.subplot(1,2,(1))
plt.plot(history.history['accuracy'],linestyle='-.')
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'valid'], loc='lower right')
plt.subplot(1,2,(2))
plt.plot(history.history['loss'],linestyle='-.')
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'valid'], loc='upper right')
```

```
<matplotlib.legend.Legend at 0x7f6edc3a3908>
```




```
] pred=model.predict(X_test_flat) #이제, test data를 이용해 모델의 정확도를 확인해보자.
  accuracy = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
  print("예측 정확도 = " , accuracy)

  313/313 [==============================] - 0s 1ms/step
  예측 정확도 =  0.9678
```

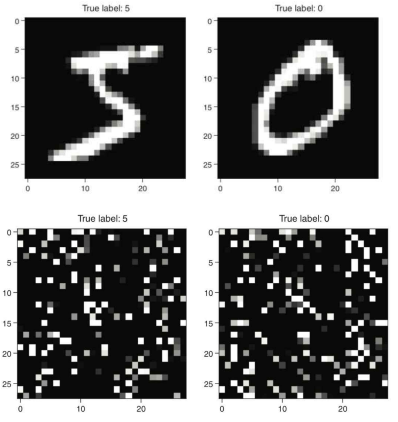
Fully Connected Neural Networks(fcNNs)

만약 그림의 픽셀이 무작위로 섞인다면?

오른쪽의 두 그림은 모두 이전 그림의 픽셀들을 동일한 방식으로 섞은 것.

하지만, fcNN에는 local structure가 없기 때문에, 원본 그림과 섞인 그림을 분류할 때 분류 성능에서 차이를 보이지 않음.

이것을 실제 코딩을 통해 확인해보자.

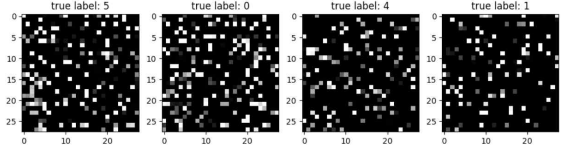


Fully Connected Neural Networks(fcNNs)

```
# 이미지의 픽셀을 무작위로 섞기 위한 함수
def shuffel_pixels(idx, data):
  data_new=np.zeros((data.shape))
  for i,img in enumerate(data):
    data_new[i] = img.flatten()[idx].reshape((28,28,1))
  return data_new

shuffel_idx = np.random.permutation(np.arange(28*28))
X_train_shuffle = shuffel_pixels(shuffel_idx, X_train)
X_val_shuffle = shuffel_pixels(shuffel_idx, X_val)
X_test_shuffle = shuffel_pixels(shuffel_idx, X_test)

# fcNN을 위해 데이터를 1차원 벡터로 펼쳐주는 작업도 필수
X_train_shuffle_flat = X_train_shuffle.reshape([X_train_shuffle.shape[0], 784])
X_val_shuffle_flat = X_val_shuffle.reshape([X_val_shuffle.shape[0], 784])
X_test_shuffle_flat = X_test_shuffle.reshape([X_test_shuffle.shape[0], 784])
```

```
# 섞인 이미지가 어떻게 생겼는지 확인해보자.
plt.figure(figsize=(12,12))
for i in range(0,4):
    plt.subplot(1,4,(i+1))
    plt.imshow((X_train_shuffle[i,:,:,0]),cmap="gray")
    plt.title('true label: '+np.str(np.argmax(Y_train,axis=1)[i]))
```

```
<ipython-input-8-cfaf763a0c23>:6: DeprecationWarning: `np.str` is a deprecated alias for the builtin `str`. To silence this warning, use `str` by itself. Doing this will not modify any behavior and is safe.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  plt.title('true label: '+np.str(np.argmax(Y_train,axis=1)[i]))
```



Fully Connected Neural Networks(fcNNs)

```
# 아까와 마찬가지로 모델을 만들고
model = Sequential()

model.add(Dense(100, batch_input_shape=(None, 784)))
model.add(Activation('sigmoid'))
model.add(Dense(50))
model.add(Activation('sigmoid'))
model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
# 이번엔 shuffle된 데이터로 학습을 시켜보자.
history=model.fit(X_train_shuffle_flat, Y_train,
                  batch_size=128,
                  epochs=10,
                  verbose=2,
                  validation_data=(X_val_shuffle_flat, Y_val)
                 )
```

```
pred=model.predict(X_test_shuffle_flat)
love_ESC = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
print("모델 정확도 = " , love_ESC)
```

```
313/313 [==============================] - 1s 2ms/step
모델 정확도 =  0.9694
```

섞인 이미지를 이용하여 training된 모델의 정확도 역시
원본 이미지를 이용하여 training된 모델의 정확도와 비슷하게
96% 수준인 것을 확인할 수 있다.

2

Convoultional Neural Networks (CNNs)

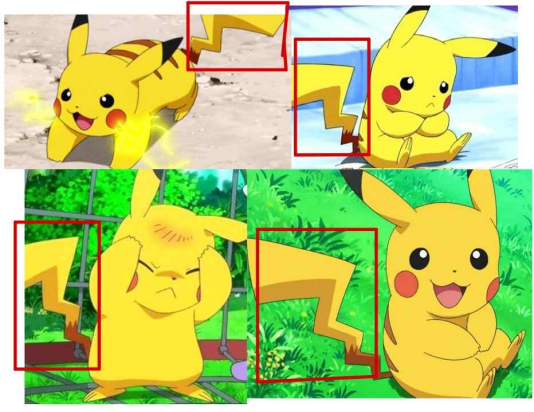
Probablistic Neural Network 1

Convolutional Neural Networks(CNNs)

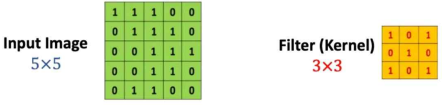
-fcNN의 한계:

1. input variable이 많아질수록, 하나의 Hidden layer만 더해도 parameter가 기하급수적으로 늘어남
→ computing cost가 매우매우 높음.
2. 2차원 형태의 이미지를 1차원으로 펼치는 과정에서 이미지의 공간적 정보가 손실됨.

-우리가 이미지를 분류할 때, fcNN처럼 전체를 보기보단, 이미지의 중요한 부분적 특성을 살리는 것이 더 효율적이지 않을까?
또한, 인접한 픽셀들 사이의 관련성도 고려한다면 어떨까?
→ **이러한 아이디어에서 시작한 것이 CNN**



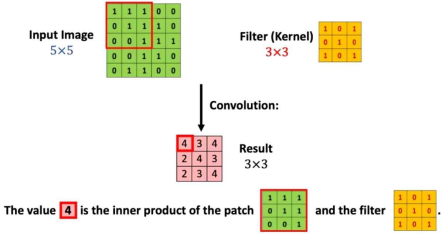
Convolutional Neural Networks(CNNs)



간단한 설명을 위해 5x5 크기의 흑백사진을 사용하자.

위 그림에서 픽샐값이 1이면 검정색, 0이면 하얀색을 의미한다.

CNN에서는 오른쪽과 같은 필터가 이미지 전체를 훑으며 패턴을 찾는다.

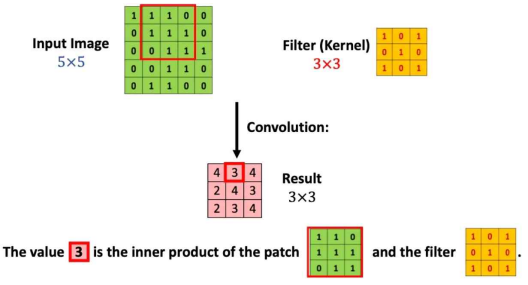


필터는 이미지를 훑으면서, matrix 의 같은 자리에 있는 원소끼리 곱하여 그 합을 resulting matrix 에 넣는다.

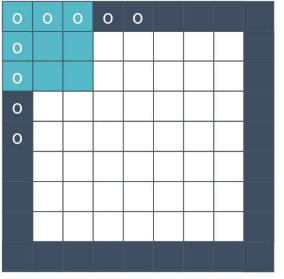
이때, 이 resulting matrix 를 feature map, activation map이라고 한다.

이 과정을 input image의 모든 자리에 대해 반복하면, 원래의 input image보다 작아진 activation map이 나온다.

Convolutional Neural Networks(CNNs)



이 연산은 결국 matrix간의 inner product이기 때문에,
Filter의 패턴과 해당 부분의 패턴이 일치할수록 큰 값이 저장된다.
이러한 과정을 거쳐 input image에서 특정한 패턴을 찾아낸다.



이때, activation map은 기존 이미지에 비해 정보량 손실이
발생하는데, 이것을 막고 싶다면 기존 이미지 주변을 의미 없는
0으로 둘러싸 주는 **zero padding** 을 할 수 있다.
또한, 필터를 꼭 한 칸 씩 움직일 필요는 없는데, 필터가 움직이는
크기를 **stride**라고 한다.

Convolutional Neural Networks(CNNs)

Input image가 AxB 사이즈이고, filter가 axb 사이즈이며, stride의 값이 s일때, activation map의 크기는 다음과 같이 나타낼 수 있다.

((A-a)/s+1) x ((B-b)/s +1)

지금까지 살펴본 흑백 이미지는 2차원 이미지이다.
그러나, 컬러 이미지의 경우 한 픽셀에 R,G,B 세 가지의 값을 가지는 3차원 이미지이기 때문에, 이 경우에는 3차원 필터를 사용해야 한다.

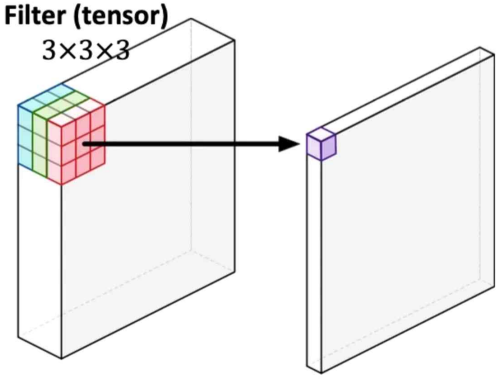
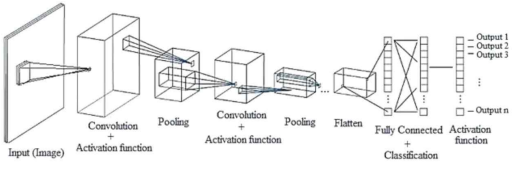
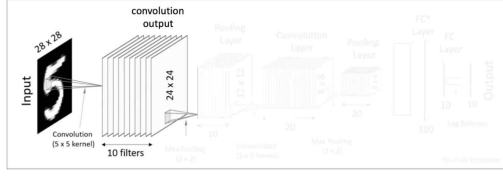


그림1. 우리가 흔히 접하는 컬러 이미지는 3차원 데이터이다

Convolutional Neural Networks(CNNs)

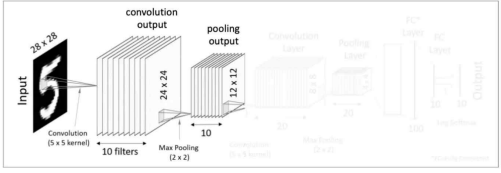




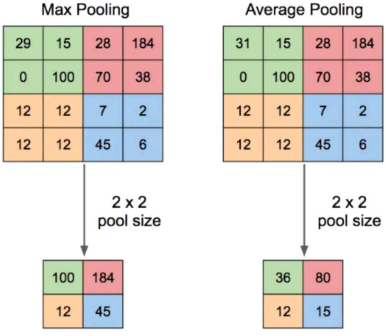
CNN의 전체 과정을 도식화한 그림. Input image에 대해 convolution layer와 pooling layer를 반복하여 적용하고, 마지막에 데이터를 펼쳐서 fcNN으로 분석하는 것을 확인할 수 있다.
각 과정을 자세히 살펴보자.

우선 input image에 대해서 convolution을 시행함. 실전에서는 하나의 필터가 아닌 여러 개의 필터를 적용시키고, 각 결과값을 하나로 쌓아 3차원 데이터를 만든다.
그 후, 결과값에 activation function을 통과시켜 output을 만든다.

Convolutional Neural Networks(CNNs)

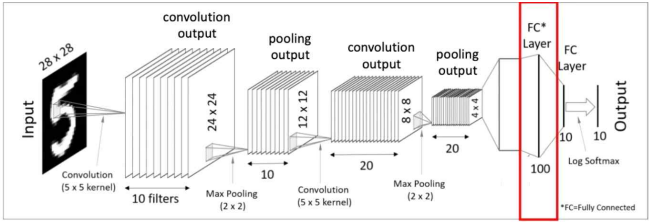


그 후, pooling layer를 거쳐 data의 dimension 을 줄이는 작업을 한다. Convolution을 통해 생성된 결과값을 줄이기 위한 것인데, pooling을 거치면 data의 크기가 줄어든다.



Pooling에는 Max Pooling과 Average pooling이 있는데, 각각의 grid 내에서 가장 큰 값만 도출하거나, grid 값들 간의 평균값을 도출한다.

Convolutional Neural Networks(CNNs)



이처럼 convolution layer와 pooling layer를 여러 번 거친 후,

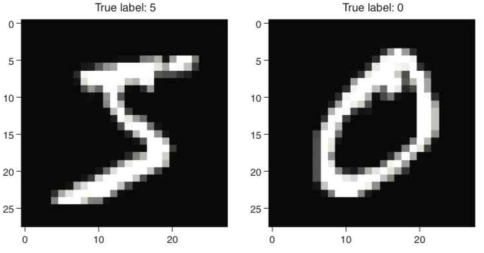
최종적으로는 데이터를 1차원으로 펼쳐 fcNN을 통해 분석하게 된다.

마지막으로 sofrmax activation function을 통과시켜

각 확률들의 합이 1이 되도록 조정하면 마무리이다.

Convolutional Neural Networks(CNNs)

이번에는 CNN을 통해 MNIST data set을 분석해 보자.




```
# 2개의 convolution layer와 fc layer를 가진 CNN을 만들 것
model = Sequential()

model.add(Convolution2D(8,(3, 3),padding='same',input_shape=(28, 28, 1)))
# 8개의 filter를 가진 convolution layer, 3*3 필터 사용
model.add(Activation('relu')) #이번에는 activation으로 ReLU를 써보자.
model.add(Convolution2D(8, (3, 3),padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2))) #2*2d의 MaxPooling을 사용

model.add(Convolution2D(16, (3, 3),padding='same'))
#이번에는 16개의 filter를 가진 convolution layer
model.add(Activation('relu'))
model.add(Convolution2D(16,(3, 3),padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2))) #마찬가지로 Maxpooling

model.add(Flatten()) #이제 fc Layer를 위해 모든 데이터를 펼치자.
model.add(Dense(40)) #40개의 neuron을 가진 fc Layer 추가
model.add(Activation('relu'))
model.add(Dense(10)) #Output에는 0~9까지 10가지의 class가 있음
model.add(Activation('softmax')) #확률의 합을 1로 맞추기 위해 softmax함수 사용

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Fully Connected Neural Networks(fcNNs)

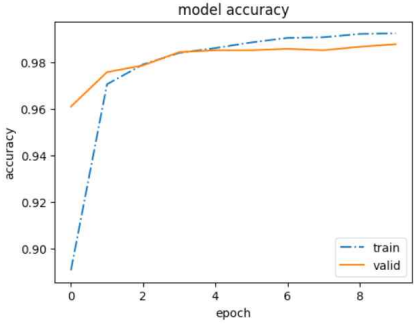
```
# 모델을 트레이닝 하고..
history=model.fit(X_train, Y_train,
                  batch_size=128,
                  epochs=10,
                  verbose=2,
                  validation_data=(X_val, Y_val)
                 )
```

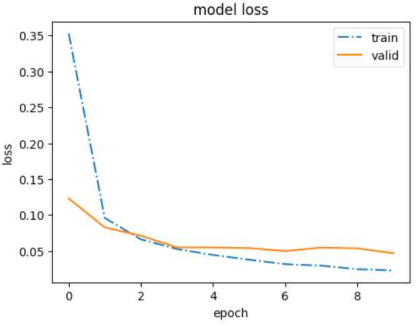
```
Epoch 1/10
391/391 - 42s - loss: 0.3530 - accuracy: 0.8905 - val_loss: 0.1232 - val_accuracy: 0.9609 - 42s/epoch - 109ms/step
Epoch 2/10
391/391 - 55s - loss: 0.0960 - accuracy: 0.9705 - val_loss: 0.0829 - val_accuracy: 0.9757 - 55s/epoch - 141ms/step
Epoch 3/10
391/391 - 47s - loss: 0.0662 - accuracy: 0.9790 - val_loss: 0.0713 - val_accuracy: 0.9786 - 47s/epoch - 121ms/step
Epoch 4/10
391/391 - 45s - loss: 0.0528 - accuracy: 0.9840 - val_loss: 0.0551 - val_accuracy: 0.9843 - 45s/epoch - 114ms/step
Epoch 5/10
391/391 - 55s - loss: 0.0445 - accuracy: 0.9860 - val_loss: 0.0549 - val_accuracy: 0.9851 - 55s/epoch - 140ms/step
Epoch 6/10
391/391 - 42s - loss: 0.0379 - accuracy: 0.9885 - val_loss: 0.0541 - val_accuracy: 0.9851 - 42s/epoch - 108ms/step
Epoch 7/10
391/391 - 41s - loss: 0.0317 - accuracy: 0.9904 - val_loss: 0.0499 - val_accuracy: 0.9857 - 41s/epoch - 106ms/step
Epoch 8/10
391/391 - 44s - loss: 0.0297 - accuracy: 0.9907 - val_loss: 0.0548 - val_accuracy: 0.9851 - 44s/epoch - 112ms/step
Epoch 9/10
391/391 - 41s - loss: 0.0247 - accuracy: 0.9921 - val_loss: 0.0537 - val_accuracy: 0.9866 - 41s/epoch - 104ms/step
Epoch 10/10
391/391 - 43s - loss: 0.0230 - accuracy: 0.9924 - val_loss: 0.0469 - val_accuracy: 0.9877 - 43s/epoch - 110ms/step
```

```
# 마찬가지로, training 진행에 따른 accuracy와 loss변화를 시각적으로 확인해보자.
plt.figure(figsize=(12,4))
plt.subplot(1,2,(1))
plt.plot(history.history['accuracy'],linestyle='-.')
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'valid'], loc='lower right')
plt.subplot(1,2,(2))
plt.plot(history.history['loss'],linestyle='-.')
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'valid'], loc='upper right')
```

Fully Connected Neural Networks(fcNNs)

<matplotlib.legend.Legend at 0x79ad5af92f80>






```
pred=model.predict(X_test) #모델의 정확도를 확인해보자.
fighting_ESC = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
print("모델 정확도 = " , fighting_ESC)
```

```
313/313 [==============================] - 3s 10ms/step
모델 정확도 =  0.9894
```

Fully Connected Neural Networks(fcNNs)

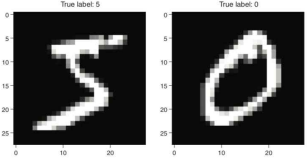
```
# 이번엔 shuffle된 이미지를 이용하여 모델을 학습시켜보자.
history=model.fit(X_train_shuffle, Y_train,
                  batch_size=128,
                  epochs=10,
                  verbose=2,
                  validation_data=(X_val_shuffle, Y_val)
                 )
```

```
Epoch 1/10
391/391 - 58s - loss: 0.7647 - accuracy: 0.7549 - val_loss: 0.3676 - val_accuracy: 0.8879 - 58s/epoch - 149ms/step
Epoch 2/10
391/391 - 45s - loss: 0.3415 - accuracy: 0.8969 - val_loss: 0.2658 - val_accuracy: 0.9173 - 45s/epoch - 115ms/step
Epoch 3/10
391/391 - 43s - loss: 0.2617 - accuracy: 0.9208 - val_loss: 0.2208 - val_accuracy: 0.9310 - 43s/epoch - 111ms/step
Epoch 4/10
391/391 - 44s - loss: 0.2175 - accuracy: 0.9335 - val_loss: 0.1951 - val_accuracy: 0.9392 - 44s/epoch - 112ms/step
Epoch 5/10
391/391 - 42s - loss: 0.1847 - accuracy: 0.9433 - val_loss: 0.1804 - val_accuracy: 0.9446 - 42s/epoch - 108ms/step
Epoch 6/10
391/391 - 46s - loss: 0.1608 - accuracy: 0.9500 - val_loss: 0.1691 - val_accuracy: 0.9452 - 46s/epoch - 116ms/step
Epoch 7/10
391/391 - 43s - loss: 0.1412 - accuracy: 0.9555 - val_loss: 0.1635 - val_accuracy: 0.9500 - 43s/epoch - 111ms/step
Epoch 8/10
391/391 - 53s - loss: 0.1288 - accuracy: 0.9592 - val_loss: 0.1663 - val_accuracy: 0.9500 - 53s/epoch - 135ms/step
Epoch 9/10
391/391 - 44s - loss: 0.1153 - accuracy: 0.9635 - val_loss: 0.1513 - val_accuracy: 0.9538 - 44s/epoch - 113ms/step
Epoch 10/10
391/391 - 44s - loss: 0.1035 - accuracy: 0.9673 - val_loss: 0.1493 - val_accuracy: 0.9558 - 44s/epoch - 111ms/step
```

```
pred=model.predict(X_test_shuffle) #shuffle된 그림으로 학습한 모델의 정확도를 확인해보자
accuracy_ESC = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
print("모델 정확도 = " , accuracy_ESC)
```

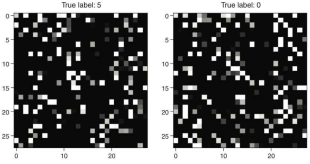
```
313/313 [==============================] - 5s 15ms/step
모델 정확도 =  0.9546
```

Convolutional Neural Networks(CNNs)




```
pred=model.predict(X_test) #모델의 정확도를 확인해보자.
fighting_ESC = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
print("모델 정확도 = " , fighting_ESC)
```

```
313/313 [==============================] - 3s 10ms/step
모델 정확도 =  0.9894
```




```
pred=model.predict(X_test_shuffle) #shuffle된 그림으로 학습한 모델의 정확도를 확인해보자
accuracy_ESC = np.sum(np.argmax(Y_test,axis=1)==np.argmax(pred,axis=1))/len(pred)
print("모델 정확도 = " , accuracy_ESC)
```

```
313/313 [==============================] - 5s 15ms/step
모델 정확도 =  0.9546
```

CNN의 경우 원본 이미지를 사용할 때와 변형된 이미지를 사용할 때 분류 성능이 달라지는 것을 확인할 수 있다.

즉, CNN은 이미지의 local structure를 활용하기 때문에 permutation 등으로 local structure 가 변하면 성능에 영향을 받는다.

3

Deriving a Loss Function for a Classification Problem

Probablistic Neural Network 1

Introduction to the MaxLike principle

Figure 4.2 A die with one side showing a dollar sign and the others displaying a dot

$.........

$.........
.$........
..$.......
...$......
....$.....
.....$....
......$...
.......$..
........$.
.........$

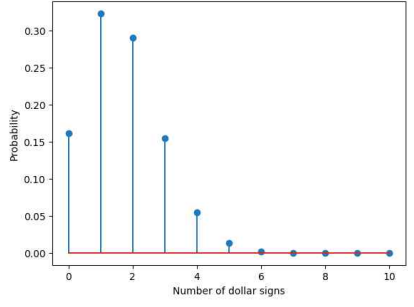
Introduction to the MaxLike principle

$\frac{5}{6} \cdot \frac{5}{6} \cdot \frac{5}{6} = \frac{1}{6} \cdot \frac{5}{6}^9 = 0.032$ or with $p = \frac{1}{6}$ as $p^1 \cdot (1-p)^{10-1}$.

$10 \cdot p \cdot (1-p)^9$

$45 \cdot (\frac{1}{6})^2 \cdot (\frac{5}{6})^8 = 0.2907$.

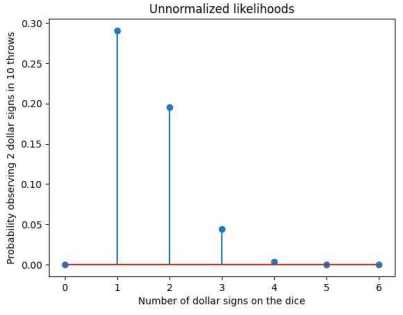
```
[7] from scipy.stats import binom
    import numpy as np
    import matplotlib.pyplot as plt
    # Define the numbers of possible successes (0 to 10)
    ndollar = np.asarray(np.linspace(0,10,11), dtype='int')
    # Calculate the probability to get the different number of possible successes
    pdollar_sign = binom.pmf(k=ndollar, n=10, p=1/6)  #B
    plt.stem(ndollar, pdollar_sign)
    plt.xlabel('Number of dollar signs')
    plt.ylabel('Probability')

    Text(0, 0.5, 'Probability')
```



Introduction to the MaxLike principle

```
# Solution
from scipy.stats import binom
# define the considered numbers of dollar signs on the die (zero to six):
ndollar = np.asarray(np.linspace(0,6,7), dtype='int')
# calculate the corresponding probability of getting 2 $-sings in 10 throws
pdollar = binom.pmf(k=2, n=10, p=ndollar/6)                    #8
plt.stem(ndollar, pdollar)
plt.xlabel('Number of dollar signs on the dice')
plt.ylabel('Probability observing 2 dollar signs in 10 throws')
plt.title('Unnormalized likelihoods')
```

```
Text(0.5, 1.0, 'Unnormalized likelihoods')
```



- Observed fixed data
- Assumed model
- Parameter => 0/6,1/6, … , 6/6

- Unnormalized probabilities
- Likelihoods

Deriving a loss function for a classification problem

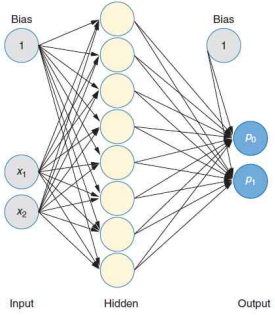


Figure 4.5 The classification network for banknotes described by the features x_1 and x_2 with two outputs yielding the probability for a real banknote (p_0) or a fake banknote (p_1). This is the same as figure 2.8 in chapter 2.

- P0 : 주어진 x가 class 0에 포함될 것이라고 "모델"이 예측한 확률
- P1 : 주어진 x가 class 1 (fake class)에 포함될 것으로 예측한 확률
- P0 + P0 = 1 (softmax 활성화 함수로 보장됨)

Deriving a loss function for a classification problem

- X = [0 , 1] , [1 , 0]
- Class : 0 / 1

- All training are independent
- Ex) Five banknotes (3 real / 2 fake)

$$P(\text{Training}) = p_0(x_1) \cdot p_0(x_2) \cdot p_0(x_3) \cdot p_1(x_4) \cdot p_1(x_5) = \prod_{j=1}^{3} p_0(x_j) \cdot \prod_{j=4}^{5} p_1(x_j)$$

$$P(\text{Training}) = \prod_{j \text{ with } y=0} p_0(x_j) \prod_{j \text{ with } y=1} p_1(x_j)$$

Deriving a loss function for a classification problem

$$P(Y = k|X = x, W = w) = \begin{cases} p_0(x, w) & \text{for } k = 0 \\ p_1(x, w) & \text{for } k = 1 \end{cases} \text{ with } \sum p_i = 1$$

- NN with fixed weight , input X => output Y CPD

$$P(\text{Training}) = \prod_{j \text{ with } y=0} p_0(x_j) \prod_{j \text{ with } y=1} p_1(x_j)$$

$$w = \text{argmax}_w \left\{ \prod_{i=1}^{n} P(Y = y_i \ x_i, w) \right\}$$

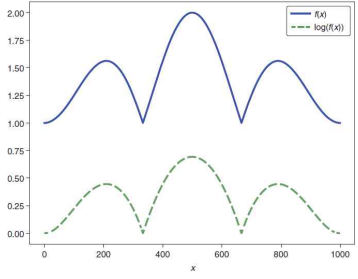
$$= \text{argmax}_w \left\{ \prod_{i \text{ with } y_i=0}^{n} P(Y = 0 \ x_i, w) \prod_{i \text{ with } y_i=1}^{n} P(Y = 1 \ x_i, w) \right\}$$

- Maxlike principle ~ weights W의 값을 likelihood 값이 최대가 되게 설정

Deriving a loss function for a classification problem

```
vals100 = np.random.uniform(0,1,100)
vals1000 = np.random.uniform(0,1,1000)
x100 = np.product(vals100)
x1000 = np.product(vals1000)
print(x100, x1000)
```

```
5.271944420831115e-40 0.0
```



- 0~1 사이의 확률 값들을 여러 번 곱하는 과정 ~ 데이터 개수가 많아질수록 0에 수렴하게 되는 문제가 생김

- P(Training)에 log를 얹어서 활용하게 됨

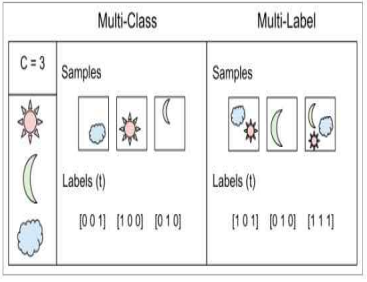
(log 함수는 단조 증가함수이다 보니 함수의 상대적 위치가 변하지 않음)

- 확률이 0이 나오게 되면 log 값이 음의 무한대로 발산하게 되는 문제가 생겨서 10E-20같은 매우 작은 값을 넣어서 보정해줌

$$\log(P(\text{Training})) = \sum_{j \text{ with } y_j=0} \log(p_0(x_j)) + \sum_{j \text{ with } y_j=1} \log(p_1(x_j)) \quad \text{Equation 4.2}$$

$$\text{crossentropy} = -\frac{1}{n}\Big(\sum_{j \text{ with } y_j=0} \log(p_0(x_j)) + \sum_{j \text{ with } y_j=1} \log(p_1(x_j)) \Big) \quad \text{Equation 4.3}$$

Deriving a loss function for a classification problem



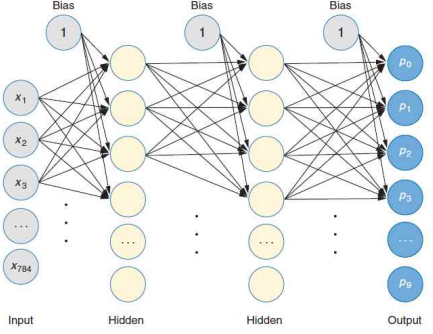


Figure 4.7 A fully connected NN (fcNN) with two hidden layers. For the MNIST example, the input layer has 784 values for the 28 × 28 pixels. The output layer has ten nodes, one for each class.

Deriving a loss function for a classification problem

```
import torch
import torch.nn as nn

torch.manual_seed(77)

### Multi class Setting ###

print(f"{'Setting up multiclass case':-^80}")

z = torch.randn(5,3) # z = f(x)
y_hat = torch.softmax(z, dim=1) # y_hat = softmax(f(x))

# ground truth : each element in target has to have 0 <= value < C
y = torch.tensor([0,1,0,2,1])

print(f"z = {z}\ny_hat = {y_hat}\ny = {y}\n{'':-^80}")

# Negative Log-likelihoods
#     - masking the correct entries
loss_NLL_scratch = -y_hat.log()[torch.arange(5),y.long()]
print(f"Negative Log-likelihoods\n    {loss_NLL_scratch}")
print(f"Loss Summantion : {loss_NLL_scratch.sum()}")
```

```
----------------------------Setting up multiclass case----------------------------
z = tensor([[-0.3568,  0.6007, -0.6988],
        [-0.5242,  0.9087, -1.6423],
        [ 0.4583, -0.1266,  0.2302],
        [ 0.0024, -0.8097,  1.3568],
        [-0.6798, -0.0881, -1.2044]])
y_hat = tensor([[0.2316, 0.6035, 0.1649],
        [0.1812, 0.7595, 0.0592],
        [0.4250, 0.2368, 0.3383],
        [0.1880, 0.0835, 0.7285],
        [0.2942, 0.5317, 0.1741]])
y = tensor([0, 1, 0, 2, 1])
--------------------------------------------------------------------------------
Negative Log-likelihoods
    tensor([1.4626, 0.2751, 0.8558, 0.3168, 0.6317])
Loss Summantion : 3.5419459342956543
```

$$\mathrm{NLL} = -\Big(\sum_{j \text{ with } y_j=0} \log(p_0(x_j)) + \sum_{j \text{ with } y_j=1} \log(p_1(x_j)) + \cdots + \sum_{j \text{ with } y_j=K-1} \log(p_{k-1}(x_j))\Big)$$

Deriving a loss function for a classification problem

$$\log \hat{y} = \begin{bmatrix} -1.24 & -1.5 & -0.72 \\ -1.04 & -1.68 & -0.78 \\ -0.61 & -0.96 & -2.61 \\ -1.89 & -1.24 & -0.58 \\ -2.5 & -0.39 & -1.42 \end{bmatrix}$$

$$y = \begin{bmatrix} 0 \\ 2 \\ 2 \\ 0 \\ 1 \end{bmatrix} \quad \log \hat{y} = \begin{bmatrix} \log \hat{y}^{(0)} & \log \hat{y}^{(1)} & \log \hat{y}^{(2)} & \log \hat{y}^{(y_i)} \\ \boxed{-1.24} & -1.5 & -0.72 & -1.24 \\ -1.04 & -1.68 & \boxed{-0.78} & -0.78 \\ -0.61 & -0.96 & \boxed{-2.61} & -2.61 \\ \boxed{-1.89} & -1.24 & -0.58 & -1.89 \\ -2.5 & \boxed{-0.39} & -1.42 & -0.39 \end{bmatrix}$$

True labels

Log predicted probabilities

$$y = \begin{bmatrix} 0 \\ 2 \\ 2 \\ 0 \\ 1 \end{bmatrix} \quad \log \hat{y} = \begin{bmatrix} \log \hat{y}^{(0)} & \log \hat{y}^{(1)} & \log \hat{y}^{(2)} & \log \hat{y}^{(y_i)} \\ \boxed{-1.24} & -1.5 & -0.72 & -1.24 \\ -1.04 & -1.68 & \boxed{-0.78} & -0.78 \\ -0.61 & -0.96 & \boxed{-2.61} & -2.61 \\ \boxed{-1.89} & -1.24 & -0.58 & -1.89 \\ -2.5 & \boxed{-0.39} & -1.42 & -0.39 \end{bmatrix} \quad -6.91$$

True labels

Log predicted probabilities

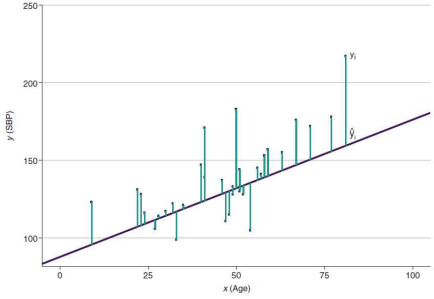
4

Deriving a Loss Function for a Regression Problem

Probablistic Neural Network 1

Deriving a loss function for regression problems

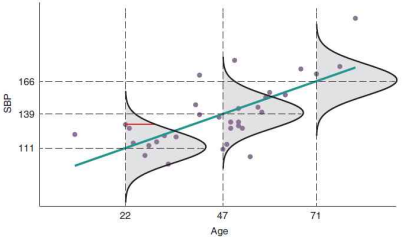
$$\text{loss} = \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - (a \cdot x_i + b))^2$$



- 회귀의 경우 앞선 분류에서와 마찬가지로, NN의 결과값은 특정 y값을 의미하는 것이 아닌, 특정 확률함수의 parameter들을 의미하게 된다.

- Classification에선 각각의 관측값에 대한 확률값이 나온 반면에 Regression에선 관측값 Y가 연속적이라 정규분포를 사용한다.
따라서, 1개의 paramete만 사용했던 베르누이분포와는 다르게 2개의 paramete를 적용한다.
다만, 우린 분산은 고정된 값으로 생각하고 NN이 평균을 조절한다고 가정한다.

Deriving a loss function for regression problems



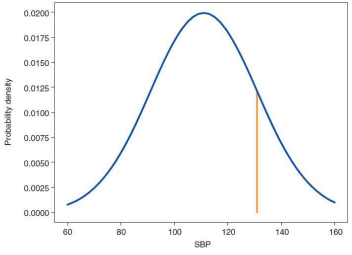


Figure 4.13 The conditional Normal density function *f*. The height of the vertical bar indicates the likelihood of the specific value under this model.

$$f(y = 131; \mu = 111, \sigma = 20) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi \cdot 400}} e^{-\frac{(131-111)^2}{2 \cdot 400}} \approx 0.01209$$

Deriving a loss function for regression problems

Listing 4.4: Estimating the maximum likelihood solution

```
# custom loss function
def my_loss(y_true,y_pred): #A
  loss = -tf.reduce_sum(tf.math.log(f(y_true,y_pred)))
  return loss

# setup NN w/o hidden layer and with output node
model = Sequential()
model.add(Dense(1, activation='linear',
                batch_input_shape=(None, 1)))
model.compile(loss=my_loss,optimizer="adam") # use custom loss
```

```
# fit the model and give out the two weights of the NN (slope and intercept)
for i in range(0,int(120000/5000)):
    model.fit(x=x,y=y,batch_size=33,
                      epochs=5000,
                      verbose = 0)
    a,b=model.get_weights()
    mse=np.mean(np.square(model.predict(x).reshape(len(x),)-y))
    print("Epoch:",i*5000+5000,"slope=",a[0][0],"intercept=",b[0],"MSE=",mse)

#Should reach (you might need to do more iteration)
#1.   optimal value for slope: 1.1050216
#2.   optimal value for intercept: 87.67143
#3.   minimal MSE: 349.200787168560
```

Deriving a loss function for regression problems

```
Epoch: 5000 slope= 2.6768923 intercept= 3.2861192 MSE= 1146.3853
Epoch: 10000 slope= 2.6660204 intercept= 6.7347355 MSE= 1075.9537
Epoch: 15000 slope= 2.570664 intercept= 11.642588 MSE= 990.4867
Epoch: 20000 slope= 2.4754844 intercept= 16.576927 MSE= 909.94775
Epoch: 25000 slope= 2.3804352 intercept= 21.507423 MSE= 834.8677
Epoch: 30000 slope= 2.2854965 intercept= 26.43024 MSE= 765.2859
Epoch: 35000 slope= 2.190644 intercept= 31.348665 MSE= 701.1361
Epoch: 40000 slope= 2.0961258 intercept= 36.256508 MSE= 642.47455
Epoch: 45000 slope= 2.001636 intercept= 41.156394 MSE= 589.21906
Epoch: 50000 slope= 1.9073676 intercept= 46.04868 MSE= 541.40216
Epoch: 55000 slope= 1.813271 intercept= 50.92797 MSE= 498.98123
Epoch: 60000 slope= 1.7195925 intercept= 55.78917 MSE= 461.97067
Epoch: 65000 slope= 1.6263069 intercept= 60.627724 MSE= 430.3394
Epoch: 70000 slope= 1.5335361 intercept= 65.440414 MSE= 404.09018
Epoch: 75000 slope= 1.4417399 intercept= 70.20454 MSE= 383.04834
Epoch: 80000 slope= 1.3511539 intercept= 74.909 MSE= 367.28796
Epoch: 85000 slope= 1.2628688 intercept= 79.46206 MSE= 356.64124
Epoch: 90000 slope= 1.1798283 intercept= 83.792145 MSE= 350.87042
Epoch: 95000 slope= 1.1150823 intercept= 87.151825 MSE= 349.23093
Epoch: 100000 slope= 1.1050111 intercept= 87.67141 MSE= 349.2008
Epoch: 105000 slope= 1.1050212 intercept= 87.671425 MSE= 349.2008
Epoch: 110000 slope= 1.10502 intercept= 87.67142 MSE= 349.20087
Epoch: 115000 slope= 1.1050212 intercept= 87.671425 MSE= 349.2008
Epoch: 120000 slope= 1.1050217 intercept= 87.671425 MSE= 349.20084
```

When we minimize the negativ log likelihood, the optimal values for the slope and intercept are the same as in chapter 3, where we minized the MSE!

Deriving a loss function for regression problems

- NN을 활용하여 확률밀도함수의 parameter을 결정하였고
- 주어진 data set에 대한 모델로 정규분포를 선택하였다.

$$Y_{x_i} \sim N(\mu_{x_i} = a \cdot x_i + b, \sigma^2)$$

- 정규분포를 일반적으로 쓰는 것은 맞지만, 실수 전체에서 정의되는 정규분포인 만큼, 개수와 같이 0이상의 수를 다루는 경우에는 적합하지 않다.
- 이때 사용할 다른 분포들은 5장에서 배운다.
- 또한, 분산을 상수로 고정하고 진행했지만 이 역시 다른 분포를 적용하여 확률변수로 취급할 수도 있다.

Deriving a loss function for regression problems

In [13]:
```
# set up a NN with 3 hidden layers and only one output node
# This allows that the mean of the CPD depends non-linearly on x
# and the standard deviation of the CPD is constant and not modeled
model = Sequential()
model.add(Dense(20, activation='relu',
                batch_input_shape=(None, 1)))
model.add(Dense(50, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(loss='mean_squared_error',optimizer="adam")
```

In [14]:
```
# summarize model along with number of model weights
model.summary()
```

Deriving a loss function for regression problems

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 20)                40
_________________________________________________________________
dense_2 (Dense)              (None, 50)                1050
_________________________________________________________________
dense_3 (Dense)              (None, 20)                1020
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 21
=================================================================
Total params: 2,131
Trainable params: 2,131
Non-trainable params: 0
_________________________________________________________________
```

Train the second model and show the fit along with the data

```
In [15]: # train the model
         history=model.fit(x, y,
                           batch_size=16,
                           epochs=1000,
                           verbose=0,
                          )
```

```
In [18]: # evaluation of the MSE loss
         model.evaluate(x,y,verbose=2)
```

```
420/1 - 0s - loss: 0.0372
Out[18]: 0.07425664712597306
```

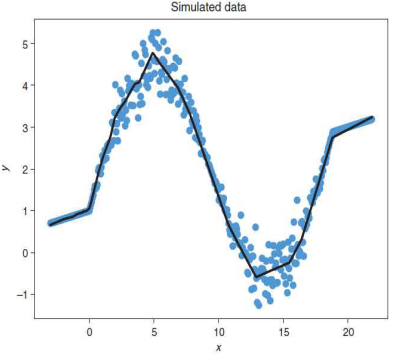


Figure 4.14 A sinus-shaped curve (solid line) fitted to the data points (see dots) by using an fcNN with three hidden layers and an MSE loss

Deriving a loss function for regression problems

- 앞선 얘기들에 더해서 정규분포의 2번째 parameter인 분산 역시 input data set에 종속된다고 하자
- 이렇게 되면 우린 NN에 또다른 output node를 추가해야 한다. 이에 대한 얘기를 할 것이다.

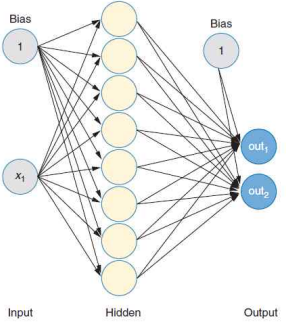


Figure 4.16 You can use an NN with two output nodes to control the parameters μ_x and σ_x of the conditional outcome distribution $N(\mu_x, \sigma_x)$ for regression tasks with nonconstant variance.

Deriving a loss function for regression problems

$$w = \text{argmin}_w \left\{ \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma(x_i, w)^2}} \right) + \frac{(\mu(x_i, w) - y_i)^2}{2\sigma(x_i, w)^2} \right\}$$

$$\text{loss} = \text{NLL} = \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma_{x_i}{}^2}} \right) + \frac{(\mu_{x_i} - y_i)^2}{2\sigma_{x_i}{}^2}$$

감사합니다