1. X-or neural network
2. **def** **sigmoid**(z):
3. **return** 1 **/** (1 **+** np**.**exp(**-**z))

이번 실습의 X-or neural network를 구현할 때에는 sigmoid function을 사용하였다. Sigmoid function이란 밑의 수식을 가지고 있는 함수로써 위 코드에서 이 식을 return 하여 주는 것을 알 수 있다.

이 외의 activation function도 relu function, tanh function 등이 존재한다.

**def** **initialize\_parameters**(n\_x, n\_h, n\_y):

W1 **=** np**.**random**.**randn(n\_h, n\_x)

b1 **=** np**.**zeros((n\_h, 1))

W2 **=** np**.**random**.**randn(n\_y, n\_h)

b2 **=** np**.**zeros((n\_y, 1))

parameters **=** {"W1" : W1, "b1": b1,

"W2" : W2, "b2": b2}

**return** parameters

W1과 W2는 weight값이고 b1과 b2는 bias값과 같은 parameter들을 초기화 시키는 과정이다. 이 과정에서 쓰인 np.random.randn(a,b) 함수는 기댓 값이 0이고 표준편차가 1인 random값을 생성해내는 과정입니다. b x a 이중 list로 random값을 생성해준다. 그리고 생성한 random값들을 parameters라고 하는 dictionary형에 key와 value까지 하여 저장을 해주는 과정이다.

**def** **forward\_propagation**(X, Y, parameters):

m **=** X**.**shape[1] #m에 X의 열수를 저장한다

W1 **=** parameters["W1"]

W2 **=** parameters["W2"]

b1 **=** parameters["b1"]

b2 **=** parameters["b2"]

Z1 **=** np**.**dot(W1, X) **+** b1

A1 **=** sigmoid(Z1)

Z2 **=** np**.**dot(W2, A1) **+** b2

A2 **=** sigmoid(Z2)

cache **=** (Z1, A1, W1, b1, Z2, A2, W2, b2)

logprobs **=** np**.**multiply(np**.**log(A2), Y) **+** np**.**multiply(np**.**log(1 **-** A2), (1 **-** Y))

cost **=** **-**np**.**sum(logprobs) **/** m

**return** cost, cache, A2

이 다음 중요 함수로는 forward\_propagation을 하는 함수이다. 여기서 cache는 단순히 저장을 하기위한 변수로 사용이 되는 것 같고 이 과정을 통해서 sigmoid함수와 MLP의 기존 알고리즘들을 적용해 계산해주는 과정이다.

**def** **backward\_propagation**(X, Y, cache):

m **=** X**.**shape[1]

(Z1, A1, W1, b1, Z2, A2, W2, b2) **=** cache

dZ2 **=** A2 **-** Y

dW2 **=** np**.**dot(dZ2, A1**.**T) **/** m

db2 **=** np**.**sum(dZ2, axis **=** 1, keepdims**=**True)

dA1 **=** np**.**dot(W2**.**T, dZ2)

dZ1 **=** np**.**multiply(dA1, A1 **\*** (1**-** A1))

dW1 **=** np**.**dot(dZ1, X**.**T) **/** m

db1 **=** np**.**sum(dZ1, axis**=**1, keepdims**=**True) **/** m

gradients **=** {"dZ2": dZ2, "dW2": dW2, "db2": db2,

"dZ1": dZ1, "dW1": dW1, "db1": db1}

**return** gradients

backward\_propagation함수에서는 forward\_propagation함수에서 실행하였던 것을 반대로하여 gradient값을 구해주는 과정이다.

**def** **update\_parameters**(parameters, grads, learning\_rate):

parameters["W1"] **=** parameters["W1"] **-** learning\_rate **\*** grads["dW1"]

parameters["W2"] **=** parameters["W2"] **-** learning\_rate **\*** grads["dW2"]

parameters["b1"] **=** parameters["b1"] **-** learning\_rate **\*** grads["db1"]

parameters["b2"] **=** parameters["b2"] **-** learning\_rate **\*** grads["db2"]

**return** parameters

위 update\_parameters 함수는 위의 backward\_propagation에서 구해놓은 gradeint값을 토대로 parameter들을 update해주는 과정을 실행해주는 함수이다. 새로운 w를 기존 w에서 수정값을 빼는 과정이며 이러한 수정값을 얻기 위해 learning rate, gradient, cost function을 사용한다.

2. neural-network\_mnist

이 코드는 keras나 tensor flow를 이용하지 않고 neural network를 구현한것이다.

init해주는 함수에서

self.lr = learningrate

##learningrate를 초기화 해주는 코드이다. 이 learning rate는 밑의 함수인 train함수에서 사용하게 되고 밑에 코드가 사용하는 부분이다

# 은닉 계층과 출력 계층 간의 가중치 업데이트

self.who += self.lr \* np.dot((output\_errors \* final\_outputs \* (1.0 - final\_outputs)), np.transpose(hidden\_outputs))

# 입력 계층과 은닉 계층 간의 가충치 업데이트

self.wih += self.lr \* np.dot((hidden\_errors \* hidden\_outputs \* (1.0- hidden\_outputs)), np.transpose(inputs))

pass

self.activation\_function = lambda x: scipy.special.expit(x)

self.inverse\_activation\_function = lambda x: scipy.special.logit(x)

pass

이 부분에서 우선 activation function으로는 sigmoid function을 이용한다.

def query(self, inputs\_list):

query 함수는 보통 forward propagation역할을 하는 함수이다

def backquery(self, targets\_list):

backquery 함수는 보통 backward propagation역할을 하는 함수이다

3.Keras - mnist

from keras.datasets import mnist

from keras.utils import np\_utils

from keras.models import Sequential

from keras.layers import Dense

from keras.callbacks import ModelCheckpoint, EarlyStopping

import os

MODEL\_SAVE\_FOLDER\_PATH = './model/'

if not os.path.exists(MODEL\_SAVE\_FOLDER\_PATH):

os.mkdir(MODEL\_SAVE\_FOLDER\_PATH)

model\_path = MODEL\_SAVE\_FOLDER\_PATH + 'mnist-' + '{epoch:02d}-{val\_loss:.4f}.hdf5'

cb\_checkpoint = ModelCheckpoint(filepath=model\_path, monitor='val\_loss',

verbose=1, save\_best\_only=True)

cb\_early\_stopping = EarlyStopping(monitor='val\_loss', patience=10)

(X\_train, Y\_train), (X\_validation, Y\_validation) = mnist.load\_data()

X\_train = X\_train.reshape(X\_train.shape[0], 784).astype('float64') / 255

X\_validation = X\_validation.reshape(X\_validation.shape[0], 784).astype('float64') / 255

Y\_train = np\_utils.to\_categorical(Y\_train, 10)

Y\_validation = np\_utils.to\_categorical(Y\_validation, 10)

model = Sequential()

model.add(Dense(512, input\_dim=784, activation='relu'))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam',

metrics=['accuracy'])

model.fit(X\_train, Y\_train, validation\_data=(X\_validation, Y\_validation),

epochs=30, batch\_size=200, verbose=0,

callbacks=[cb\_checkpoint, cb\_early\_stopping])

print('\nAccuracy: {:.4f}'.format(model.evaluate(X\_validation, Y\_validation)[1]))

이 코드는 홈페이지에도 써있듯이 keras를 이용하여 손글씨를 인식하는 기능을 하는 코드이다.

MNIST 데이터를 네트워크에서 다운받아서 각각의 변수에 불러오도록 수행을 하며 MNIST데이터는 이미지로 되어 있고 28 \* 28 = 784개의 픽셀로 이루어져 있다.

model = Sequential()

model.add(Dense(512, input\_dim=784, activation='relu'))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam',

metrics=['accuracy'])

model.fit(X\_train, Y\_train, validation\_data=(X\_validation, Y\_validation),

epochs=30, batch\_size=200, verbose=0,

callbacks=[cb\_checkpoint, cb\_early\_stopping])

model = sequential() 의 부분에서는 model이 sequential classification을 사용한다는 뜻이고 model.add(~~)는 MLP의 layer을 한층 씩 추가하는 과정이고 이 코드 같은 경우에는 activation function을 relu를 이용 하는것과 softmax를 사용하는 layer을 이용하는 것을 알 수 있다. 그리고 보통의 rnn에서 layer을 추가할 때 relu함수가 최근들어 뉴럴 네트워크에서 가장 많이 쓰이는 활성함수이다. 많이 쓰이는 만큼 accuracy면에서도 좋은 편이고 계산이 매우 효율적이다. softmax는 수업시간에 하지 않은 activation함수라서 조금 조사를 해보니 결과를 전체 합계가 1이 되는 0과 1 사이의 값으로 변경해 주며 변환된 결과의 합계가 1이 되도록 만들어주는 역할을 주로 수행한다고 한다.

1. Auto encoder

Autoencoder란 비지도 학습에서 주로 사용하는 구조로 인코더를 통해 입력 데이터에 대한 특징을 추출하고 디코더를 통해 원본 데이터를 재구성하는 학습 방식이다. 오토인코더의 구조는 다음 그림과 같다

텍스트이(가) 표시된 사진

자동 생성된 설명

중간에 은닉층이 있고 은닉층에 제약을 두어 단순히 입력을 바로 출력으로 복사하지 못하도록 방지한다.

# 이 모델은 입력을 입력의 인코딩된 입력의 표현으로 매핑

encoder = Model(input\_img, encoded)

# 인코딩된 입력을 위한 플레이스 홀더

encoded\_input = Input(shape=(encoding\_dim,))

# 오토인코더 모델의 마지막 레이어 얻기

decoder\_layer = autoencoder.layers[-1]

# 디코더 모델 생성

decoder = Model(encoded\_input, decoder\_layer(encoded\_input))

->인코더와 디코더 모델을 생성하는 과정이다

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

data들을 train과 test로 나누어서 크기 784의 벡터로 만드는 과정이다.

1. CIFAR-10

이 코드는 CIFAR-10데이터셋을 검증하는데 사용하는 코드이다. 약간 조사를 해보니 CIFAR-10 데이터셋은 각각의 레이블마다 32x32 크기 이미지인 40000개의 training데이터셋, 10000개의 test 데이터셋이 존재하고, 총합하여 50000개의 데이터셋으로 구성 되어있다. 주로 맨 처음으로 앞에서 하였던 mnist데이터를 처리하고 그다음으로 cifar-10 데이터를 처리 한다고 한다.

# DNN\_CIFAR-10 / MNIST보다 복잡한 데이터의 처리(R,G,B)

import numpy as np

from keras import datasets

from keras.utils import np\_utils

from keras import layers, models

class DNN(models.Sequential):

    def \_\_init\_\_(self, Nin, Nh\_l, Pd\_l, Nout):

        super().\_\_init\_\_()

        # 첫 번째 은닉층

        self.add(layers.Dense(Nh\_l[0], activation='relu',input\_shape=(Nin,), name='Hidden-1'))

        # Dropout 확률을 정한다.

        # Dropout: 랜덤으로 몇개의 노드를 비활성화 한다.(오버피팅 방지)

        self.add(layers.Dropout(Pd\_l[0]))

        self.add(layers.Dense(Nh\_l[1], activation='relu', name='Hidden-2'))

        self.add(layers.Dropout(Pd\_l[1]))

        self.add(layers.Dense(Nout, activation='softmax'))

        self.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

* 여기에서는 데이터를 MLP에서 여러 개의 layer을 추가해주는 과정인데 여기서 안쪽에 dropout이라는 것은 tensor flow에서는 이미 구현되어있고 deep learning에서 overfitting을 줄이는 방법으로 그림으로 나타내면 layer안에서 몇 개의 노드를 아예 참여를 안시켜 버리고 나머지 노드들을 통해서만 훈련을 하게하는 방법이다. 그리고 보통 쉬게하는 노드는 random으로 설정해 주는 편이다

지도, 텍스트이(가) 표시된 사진

자동 생성된 설명

구글링에서 가져온 이미지인데 여기서 (b)를 한 것이 dropout을 한것이다

# 데이터 정리

def Data\_func():

    (X\_train, y\_train), (X\_test, y\_test) = datasets.cifar10.load\_data()

    # 0~9 사이의 정수값을 10개의 원소를 가지는 벡터로 변환

    # 1 -> 0100000000, 2 -> 0010000000, 9 -> 0000000001

    Y\_train = np\_utils.to\_categorical(y\_train)

    Y\_test = np\_utils.to\_categorical(y\_test)

    L, W, H, C = X\_train.shape

    X\_train = X\_train.reshape(-1, W \* H \* C)

    X\_test = X\_test.reshape(-1, W \* H \* C)

    X\_train = X\_train / 255.0

    X\_test = X\_test / 255.0

    return (X\_train, Y\_train), (X\_test, Y\_test)

위의 코드는 데이터를 정리해주는 과정이다. 데이터를 load한후에 train set과 test set으로 정리를 하여 return을 해주는 과정이다.

from ex\_2\_1\_keras\_ann import plot\_acc, plot\_loss

import matplotlib.pyplot as plt

# 모델 테스팅

def main():

    Nh\_l = [100, 50]

    Pd\_l = [0.0, 0.0]

    number\_of\_class = 10

    Nout = number\_of\_class

    (X\_train, Y\_train), (X\_test, Y\_test) = Data\_func()

    model = DNN(X\_train.shape[1], Nh\_l, Pd\_l, Nout)

    history = model.fit(X\_train, Y\_train, epochs=10, batch\_size=100, validation\_split=0.2)

##모델을 테스팅 하는 코드인데 여기서 epoch라는 것과 batch size라는 용어가 나온다. epoch라는 것은 한 번의 epoch는 인공 신경망에서 전체 데이터 셋에 대해 forward pass/backward pass 과정을 거친 것을 말함. 즉, 전체 데이터 셋에 대해 한 번 학습을 완료한 상태를 의미한다.

batch size는 한 번의 batch마다 주는 데이터 샘플의 size를 의미한다.