<인공지능 팀 프로젝트 보고서\_DNN>

1. 과제 개요

-배송 지연 예측을 통한 사기 여부를 DNN을 이용하여 예측합니다.

1. 구현 환경

-Visual Studio Code/Google Colab

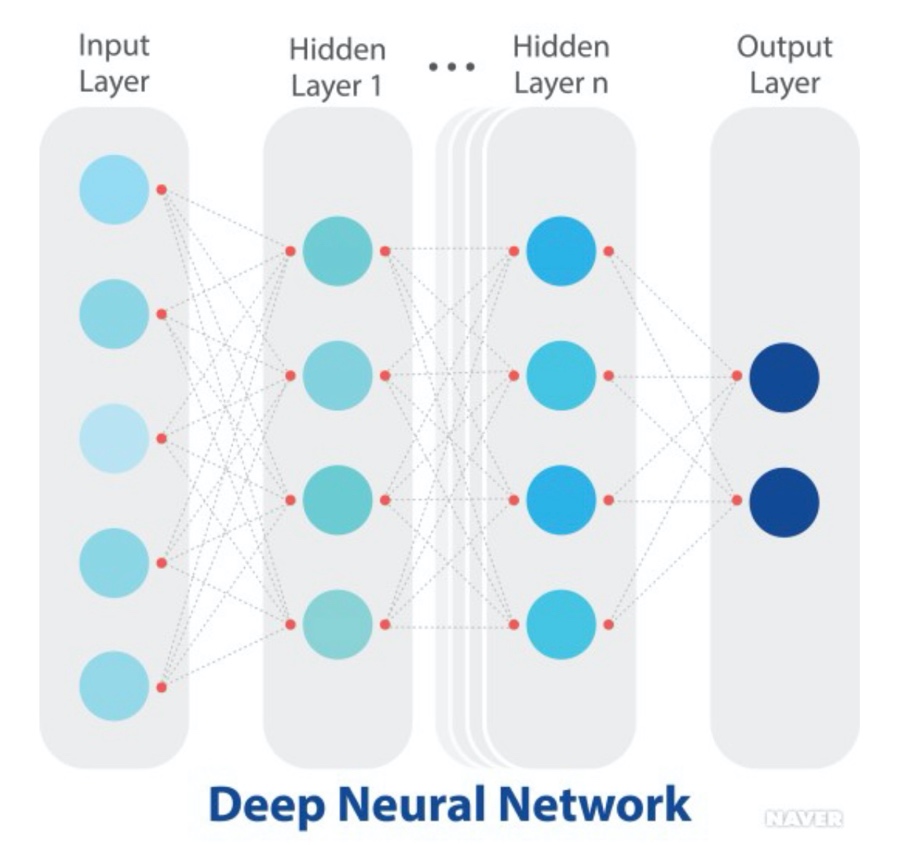
-python

-keras

1. 알고리즘에 대한 설명

3.1 심층 신경망(DNN: Deep Neural Network)이란?

DNN은 입력층(input layer)과 출력층(output layer) 사이에 다중의 은닉층(hidden layer)을 포함하는 인공 신경망입니다. 다중의 은닉층을 포함하여 다양한 비선형적 관계를 학습할 수 있습니다. 그러나 학습을 위해 많은 연산량을 필요로 하며, 과하게 학습하여 실제 데이터에 대해 오차가 증가하는 과적합(overfitting), 기울기값이 소실되는 문제(vanishing gradient problem) 등의 문제점이 발생할 수 있습니다. 이를 해결하기 위해 dropout, ReLU, batch normalization 등의 기법을 적용하여 딥러닝의 핵심 모델로 활용하고 있습니다.



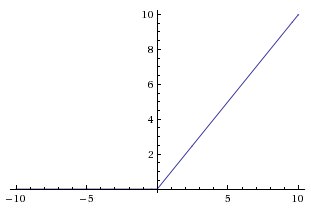
3.2 DNN과 활성화 함수

활성화 함수(activation function)는 DNN 구조에서 학습의 방향을 주도하는 역할을 합니다. 활성화 함수는 입력값이 최소값보다 작을 때는 출력 값을 작게, 일정한 값을 초과하면 출력 값을 증폭시킵니다.

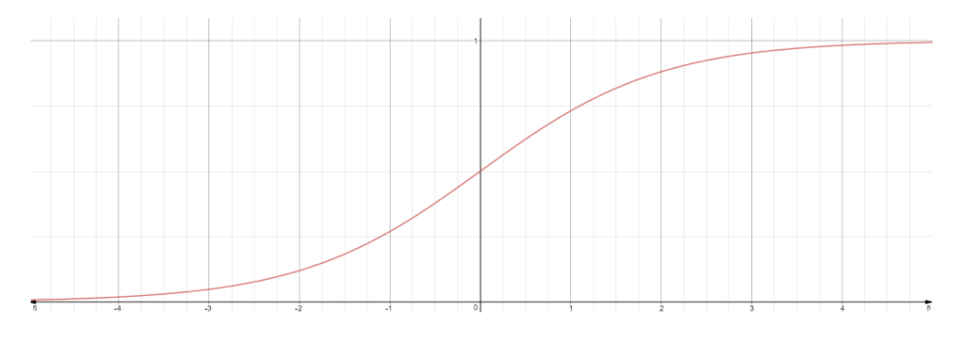
활성화 함수의 종류에는 step, sigmoid, ReLU, softmax 등이 있습니다.

<구현한 알고리즘에서 사용한 활성화 함수>

1. ReLU: hidden layer에서 사용



1. sigmoid: 마지막 layer 학습에 사용



3.3 Hyper Parameter

3.3.1 Hyper Parameter Related to Network Structure

(1) Number of Hidden Layer and Units

Input Layer와 Output Layer 사이의 계층인 Hidden Layer의 개수와 각 layer들의 노드의 개수 2가지가 변수가 됩니다. 보통 네트워크 구조에서 각 hidden layer의 뉴런들은 정규화 기술이 적용되며 모델의 정확도를 높이는데 기여합니다. 그리고 뉴런의 개수와 레이어의 깊이가 학습 시 Underfitting 또는 Overfitting에 영향을 줍니다.

ex1) hidden layer: 3개층, node 개수: 256, 128, 64

텍스트이(가) 표시된 사진

자동 생성된 설명

ex2) hidden layer: 3개층, node 개수: 200, 200, 200

텍스트이(가) 표시된 사진

자동 생성된 설명

ex3) hidden layer: 5개층, node 개수: 512, 512, 256, 256, 256

텍스트이(가) 표시된 사진

자동 생성된 설명

ex4) hidden layer: 5개층, node 개수: 512, 256, 256, 128, 128

텍스트이(가) 표시된 사진

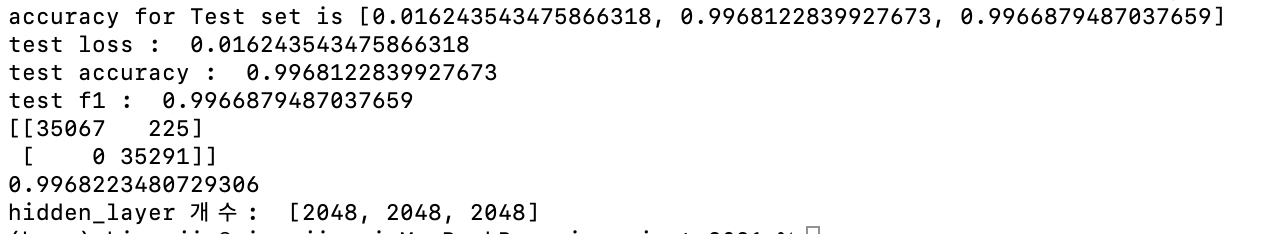
자동 생성된 설명

ex5) hidden layer: 3개층, node 개수: 2048, 1024, 1024

텍스트이(가) 표시된 사진

자동 생성된 설명

ex6) hidden layer: 3개층, node 개수: 2048, 2048, 2048



hidden layer와 node의 개수를 바꿔가며 학습을 시켜본 결과, hidden layer는 3개, node의 개수는 2048개 정도에서 효율이 좋게 나타나는 것을 확인했습니다.

(2) Dropout

Dropout은 overfitting을 피하기 위한 정규화 기법입니다. 학습 데이터에 과적합하게 학습된 model을 실제로 사용하기 위해 사용하며, hidden layer의 뉴런들을 일정 비율로 배제하고 학습을 합니다.

Dropout 값을 0.1~1.0 사이의 값으로 변경해가며 모델을 학습해 본 결과, 0.1~0.5 사이의 값을 가질 때 효율이 좋게 나타났으며, 0.4일 때 가장 효율이 좋았습니다.

(3) Weight Initialization in Network

각 레이어에 사용되는 Activation Function에 따라서 적절한 Weight 초기화 기법을 사용하는 것이 좋습니다. 1의 값을 갖는 class에는 0.9, 0의 값을 갖는 class에는 0.1의 Weight를 줬을 때 가장 좋은 효율을 나타냈습니다.

3.3.2 Hyper Parameters Related to Training Algorithm

(1) Number of Epochs

Epoch이란 model이 전체 dataset에 대해 한번의 학습을 완료한 상태를 말합니다. 학습을 많이 할수록 error가 낮아질 것이라고 예상할 수 있으나 지나치게 학습을 진행하면 overfitting이 발생하게 됩니다. 이 현상으로 인해 validation 단계에서 error가 감소하다가 어느 순간부터 다시 증가하게 될 수도 있습니다. 과적합을 방지하기 위해 early stopping을 통해 epoch의 경계값을 찾고자 하였고, 결과적으로 epoch값을 1000으로 결정하게 되었습니다.

(2) Batch Size

Batch size는 한 번의 batch마다 주는 데이터 샘플의 size입니다. 전체 dataset을 한번에 학습하려면 하드웨어 시스템에 부담이 있으며 시간이 오래 걸린다는 문제도 발생하게 됩니다. 따라서 전체 dataset을 일정 크기로 나누어 학습을 하게 됩니다.

Batch size로 나누어진 dataset 하나의 크기를 mini batch라고 합니다. 이때 Batch size로 나눠진 mini batch들을 전부 학습하여 전체 dataset을 1 epoch 학습하는데 실행한 횟수를 iteration이라고 합니다. 구현한 DNN 모델에서는 batch size를 512로 두어 하드웨어 시스템의 부담과 학습에 소요되는 시간을 줄였습니다.

3.4 스케일링

데이터를 모델링 하기 전에는 스케일링 과정을 통해 다차원의 값들을 비교 및 분석하기 쉽게 만들어주어야 합니다. 이 과정을 통해 자료의 오버플로우나 언더플로우를 방지하고, 최적화 과정에서의 안정성 및 수렴 속도를 향상시킵니다. 구현한 모델에서는 StandardScaling()을 통해 스케일링 과정을 거쳤습니다.

1. 데이터에 대한 설명

4.1 Input Feature

-fraud: 배송 지연 예측을 통한 사기 여부에 대한 정보

-n\_in: 입력 데이터의 크기(41 columns)

4.2 Target Output

-n\_out: 출력 데이터의 크기(1)

-fraud인지 아닌지 정보를 출력

-1: fraud

-0: fraud가 아님

1. 소스코드에 대한 설명

"""사용할 패키지 불러오기

"""

import numpy as np

import tensorflow as tf

import os

import matplotlib.pyplot as plt

from sklearn import metrics

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

from sklearn import datasets

from keras import backend as K

from keras.models import Sequential

from keras.layers import Dense, Activation, Dropout, BatchNormalization

from keras.optimizers import SGD

from tensorflow.python.keras import activations

from tensorflow import keras

from sklearn.model\_selection import train\_test\_split, cross\_val\_score, cross\_val\_predict

from sklearn.preprocessing import MinMaxScaler,StandardScaler

from sklearn.utils import class\_weight

from sklearn.metrics import confusion\_matrix

from keras.callbacks import EarlyStopping

from sklearn.metrics import classification\_report

from keras.models import load\_model

from IPython.display import SVG

# macro\_f1\_score

from sklearn.metrics import f1\_score, recall\_score, precision\_score

# loader.py 파일 import

from utils.loader import get\_fraud

def f1(y\_true, y\_pred):

def recall(y\_true, y\_pred):

"""Recall metric.

Only computes a batch-wise average of recall.

Computes the recall, a metric for multi-label classification of

how many relevant items are selected.

"""

true\_positives = K.sum(K.round(K.clip(y\_true \* y\_pred, 0, 1)))

possible\_positives = K.sum(K.round(K.clip(y\_true, 0, 1)))

recall = true\_positives / (possible\_positives + K.epsilon())

return recall

def precision(y\_true, y\_pred):

"""Precision metric.

Only computes a batch-wise average of precision.

Computes the precision, a metric for multi-label classification of

how many selected items are relevant.

"""

true\_positives = K.sum(K.round(K.clip(y\_true \* y\_pred, 0, 1)))

predicted\_positives = K.sum(K.round(K.clip(y\_pred, 0, 1)))

precision = true\_positives / (predicted\_positives + K.epsilon())

return precision

precision = precision(y\_true, y\_pred)

recall = recall(y\_true, y\_pred)

return 2\*((precision\*recall)/(precision+recall+K.epsilon()))

class NN:

def \_\_init\_\_(self):

"""Create Dataset

원본 데이터를 불러옵니다.

불러 온 데이터로부터 training set과 test set을 생성합니다.

효과적으로 학습할 수 있도록 데이터셋을 scaling 합니다.

model 생성에 이용될 input layer, hidden layer, 그리고 output layer의 형태를 결정하고,

마지막 layer에 이용할 activation 함수와 dropout 확률의 비율을 설정합니다.

Arguments:

X:

y:

X\_train: X의 training dataset(80%)

X\_test: X의 test dataset(20%)

y\_train: y의 training dataset(80%)

y\_test: y의 test dataset(20%)

n\_in: input data의 개수(column 크기)

n\_hiddens: hidden layer의 개수 및 각 layer의 뉴런 개수

n\_out: output data의 개수

activation: last layer의 activation function

p\_keep: dropout 확률의 비율

Keyword Arguments:

Raises:

Returns: [none] -- [dataset 준비]

"""

self.X, self.y = get\_fraud()

self.X\_train, self.X\_test, self.y\_train, self.y\_test = train\_test\_split(self.X, self.y, test\_size=0.2, random\_state=42, stratify=self.y) # 학습 데이터: 80%, 검증 데이터: 20%

# print(len(self.X\_train)) # 282331

# print(len(self.y\_train)) # 282331

# print(len(self.X\_test)) # 70583

sc = StandardScaler()

self.X\_train=sc.fit\_transform(self.X\_train)

self.X\_test=sc.fit\_transform(self.X\_test)

self.n\_in = 41 # 입력 데이터의 크기: 41 columns

# print('input 데이터의 크기: ', self.n\_in)

self.n\_hiddens = [2048, 2048, 2048] # 각 은닉층의 뉴런 개수

# print('각 은닉층의 뉴런 개수: ', self.n\_hiddens)

self.n\_out = 1 # 출력 데이터의 개수: 1개(0 또는 1)

# print('output 데이터의 개수', self.n\_out)

self.activation = 'sigmoid'

self.p\_keep = 0.4 # dropout 확률의 비율

# DNN 수행

def create\_model(self):

"""Create Dnn Model

Arguments:

Keyword Arguments:

Raises:

Returns: [model] -- DNN model

"""

model = Sequential()

model.add(BatchNormalization()) # 배치 정규화

# hidden layer만큼 Neural-Network 반복

for i, input\_dim in enumerate([self.n\_in] + self.n\_hiddens[:-1]):

model.add(Dense(input\_dim = input\_dim, units = self.n\_hiddens[i], kernel\_initializer='random\_uniform'))

model.add(Activation('relu')) # activation: relu, sigmoid, softmax 등

# model.add() # 가중치 초기화

model.add(Dropout(self.p\_keep))

model.add(Dense(units = self.n\_out))

model.add(Activation(self.activation))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy', f1])

# model.compile(optimizer='sgd', loss='binary\_crossentropy', metrics=[keras.metrics.Precision(), keras.metrics.Recall()])

return model

if \_\_name\_\_ == "\_\_main\_\_" :

"""main

Arguments:

Keyword Arguments:

Raises:

Returns:

"""

n = NN()

model = n.create\_model()

batch\_size = 512

epochs = 1000

class\_weights = {1: 0.9, 0: 0.1}

callbacks = EarlyStopping(monitor='f1', patience=50, verbose=1)

# reconstructed\_model = keras.models.load\_model("my\_model.h5")

# np.testing.assert\_allclose(

# model.predict(n.X\_test), reconstructed\_model.predict(n.X\_test)

# )

# # 훈련 단계

# model.fit(n.X\_train, n.y\_train, batch\_size=batch\_size, epochs=epochs, callbacks=callbacks, class\_weight=class\_weights, verbose=1)

# model.fit(n.X\_test, n.y\_test, batch\_size=batch\_size, epochs=epochs, callbacks=callbacks, class\_weight=class\_weights, verbose=1)

plot\_model(model, show\_shapes=True, to\_file='DNN\_model.png')

# 정확도 평가 단계

# train\_evaluate = model.evaluate(n.X\_train, n.y\_train)

test\_evaluate = model.evaluate(n.X\_test, n.y\_test)

# print('accuracy for Train set is', train\_evaluate)

print('accuracy for Test set is', test\_evaluate)

# print(model.predict(n.X\_test))

y\_pred = model.predict(n.X\_test)

# y\_pred = np.argmax(y\_pred1, axis = 1)

# print(y\_pred1)

cm = confusion\_matrix(n.y\_test, y\_pred.round())

model.save("my\_model.h5")

# print('test evaluate : ', test\_evaluate)

print('test loss : ', test\_evaluate[0])

print('test accuracy : ', test\_evaluate[1])

print('test f1 : ', test\_evaluate[2])

print(classification\_report(n.y\_test, y\_pred.round()))

print(cm)

print(f1\_score(n.y\_test, y\_pred.round()))

print('hidden\_layer 개수: ', n.n\_hiddens)

5.1 학습 과정에 대한 설명

(1) loader파일의 get\_fraud()를 통해 fraud data정보를 가져옵니다.

(2) 가져온 fraud 데이터를 학습 데이터 80%, 테스트 데이터 20%의 비율로 split 합니다.

(3) 데이터를 모델링 하기 전에 StandardScaler()를 사용하여 스케일링 과정을 거칩니다.

데이터를 모델링 하기 전에는 스케일링 과정을 통해 다차원의 값들을 비교 및 분석하기 쉽게 만들어주어야 합니다. 이 과정을 통해 자료의 오버플로우나 언더플로우를 방지하고, 최적화 과정에서의 안정성 및 수렴 속도를 향상시킵니다.

구현한 모델에서는 StandardScaler(), MinMaxScaler(), Normalizer() 등을 통해 스케일링을 한 후 코드를 실행해보았습니다. 그 결과 StandardScaler()에서 가장 좋은 결과를 보였고, 스케일링 함수로 StandardScaler()를 채택하였습니다.

StandardScaler()는 feature를 리스케일링하여 각 특성의 평균을 0, 분산을 1로 변경하여 특성의 스케일을 맞춥니다. 궁극적으로 모든 feature들을 공통의 척도로 변경해주는 것이 목표입니다.

그런 다음, scikit-learn에서 제공하는 라이브러리 등을 통해 fit\_transform()과 transform() 메소드를 적용하였습니다. Fit\_transform()은 train dataset에서만 사용되며, 모델이 train data에 있는 평균과 분산을 학습하게 됩니다. 또한 이렇게 학습된 파라미터는 test data를 scale하는데 사용됩니다. 결국 train data로 학습된 스케일러의 파라미터를 통해 test data의 feature 값들이 스케일 되는 것입니다. Train data로부터 학습된 평균과 분산 값을 test data에 적용하기 위해 transform() 메소드를 사용합니다.

만약에 fit\_transform을 test data에도 적용하게 된다면 test data로부터 새로운 평균과 분산 값을 얻게 되어 모델이 test data도 학습하게 됩니다. 그러나 test data는 모델이 학습된 후에 평가할 때만 사용되어야 하므로 test data에는 fit\_transform을 적용하지 않습니다.

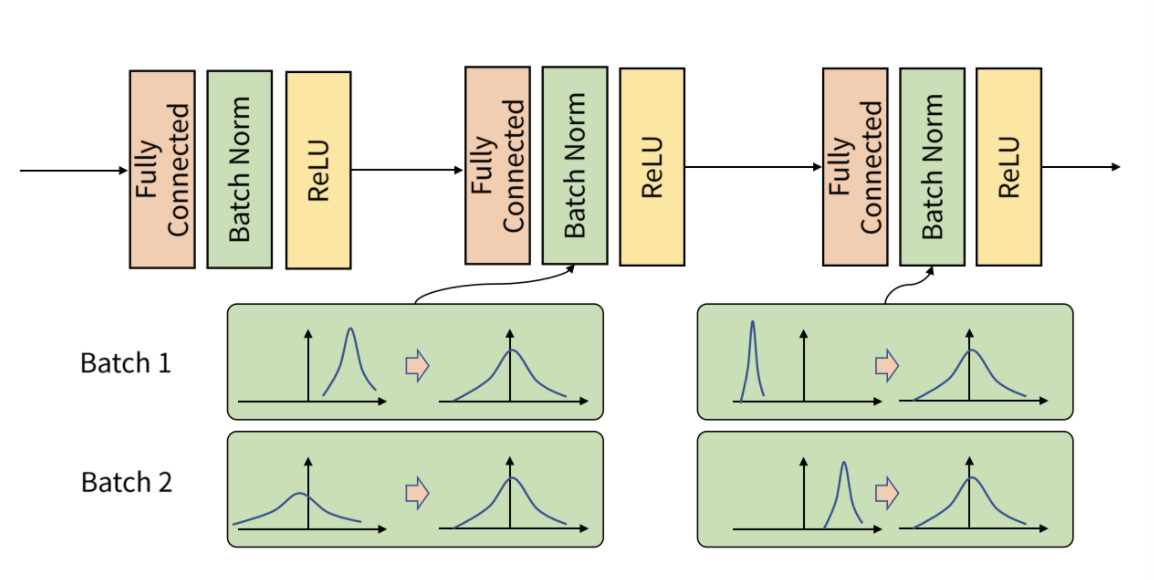
(4) create\_model(): 41개의 columns를 input data로 하고, 각 hidden layer의 뉴런 개수를 각각 2048개로 하여 은닉층을 3개 갖고, 1개의 출력값을 갖는 DNN 알고리즘을 구현합니다.

n\_in = 41

n\_hiddens = [2048, 2048, 2048]

n\_out = 1

(4)-1 배치 정규화를 위해 BatchNormalization() 적용

Batch normalization은 학습 과정에서 각 배치 단위 별로 데이터가 다양한 분포를 가지더라도 각 배치 별로 평균과 분산을 이용해 정규화 하는 것을 뜻합니다.

위 그림을 보면 batch 단위나 layer에 따라서 입력 값의 분포가 모두 다르지만, 정규화를 통해 분포를 zero mean gaussian 형태로 만들 수 있습니다. Zero mean gaussian은 평균이 0, 표준 편차는 1인 데이터의 분포를 가지는 형태입니다.

DNN 모델에 배치 정규화 함수를 적용하지 않았을 경우 검증 손실 값이 0.27이었고, 적용하였을 경우 검증 손실 값이 0.19로 낮아지는 것을 확인할 수 있었습니다. Accuracy와 F1 score 값에는 변화가 거의 없었으나, 손실 값에서 더욱 효율적인 결과를 얻을 수 있기 때문에 BatchNormalization()을 적용하였습니다.

(4)-2 각 layer에 activation function으로 ReLU를 적용하고, 마지막 layer에는 sigmoid를 적용합니다.

Perceptron은 fully-connected layer와 sigmoid로 이루어져 있으며, perceptron을 아주 많이 쌓게 되면 그것을 DNN이라고 부릅니다. 만약 DNN을 이루고 있는 요소 중 sigmoid 함수가 없다면 fully-connected layer를 100개 쌓아도 10000개를 쌓아도 선형 함수 하나로 표현이 가능하기 때문에, 한 층을 쌓은 것과 다를 바 없게 됩니다. 그렇기 때문에 여러 층을 쌓는 것이 의미를 갖기 위해서는 hidden layer에서 좀 더 복잡한 형태의 activation function을 사용해야 합니다. 이때 고려할 수 있는 세 가지 일반적인 activation function에는 ReLU, sigmoid, 그리고 tanh가 있습니다.

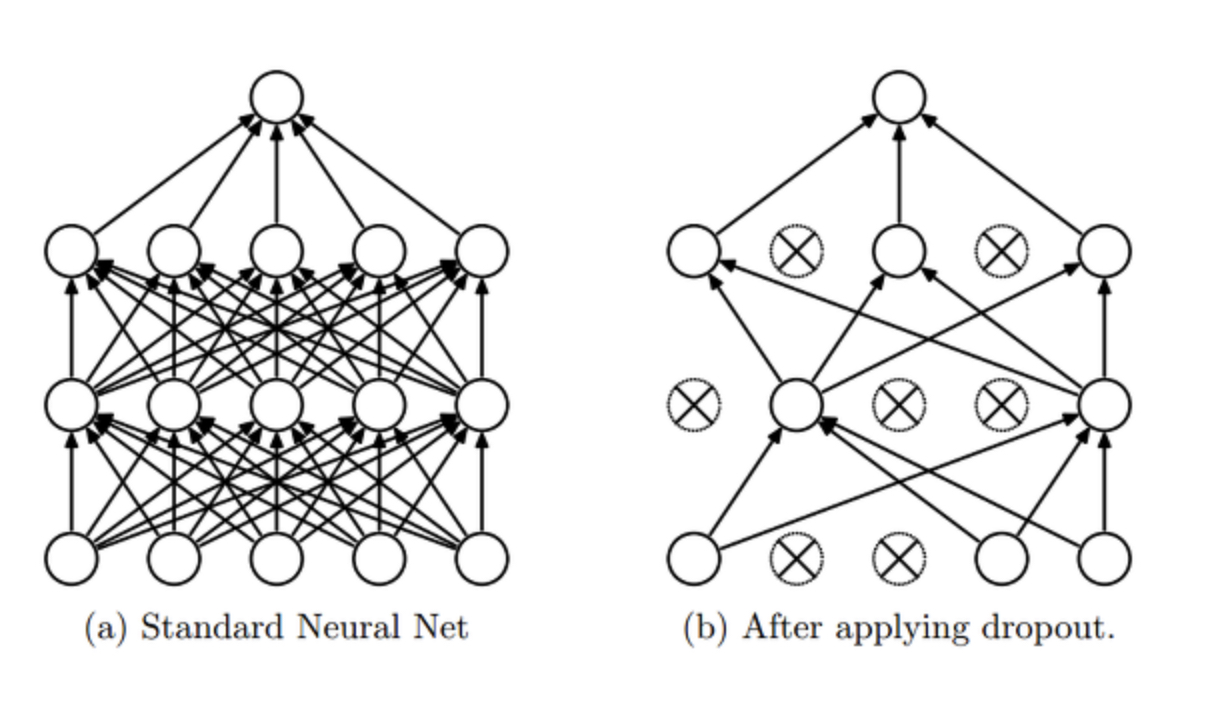
우선 Sigmoid 함수는 값이 어느정도 작으면 0으로 수렴하고 어느정도 크면 1로 수렴합니다. 이는 Node를 켜고 끄는 개념을 위해서 도입되었으나, 실제로 DNN에 적용하면 문제가 발생합니다. 함수가 0에서 1 사이의 값만 가지기 때문에 Chain rule을 적용하면 값이 계속 작아져서 gradient가 사라지는 vanishing gradient 문제가 발생하여 결론적으로 학습이 되지 않습니다. 또한 계산량이 많아서 복잡하다는 문제도 있습니다.

또 다른 non-linear activation function에는 tanh 함수(쌍곡 탄젠트 함수)가 있습니다. 이 함수는 sigmoid와 매우 유사하며, 동일한 S자 모양을 갖습니다. 임의의 실수 값을 입력으로 취하고 -1에서 1까지의 범위를 출력 값으로 갖습니다. Tanh 함수도 마찬가지로 vanishing gradient 문제가 발생하여 기울기가 사라지는 문제가 발생하게 됩니다.

이를 해결하기 위한 activation function이 ReLU입니다. ReLU는 0보다 작은 값이 나오면 노드를 아예 꺼버리고, 그렇지 않으면 그 값을 그대로 가져가도록 합니다. ReLU는 선형 함수이지만, multi-layer의 activation function으로 ReLU를 사용하게 되면 linear한 부분 부분이 결합된 합성 함수가 만들어지게 됩니다. 따라서 최종적으로 봤을 때 non-linear한 성질을 가지게 됩니다. 또한 간단한 구현이 가능하기 때문에 계산의 복잡성이 크게 감소하여 훈련 시간도 단축됩니다.

그러나 ReLU는 0보다 작은 값을 가질 때 아예 0으로 만들어버린다는 문제가 있습니다. 이러한 문제를 해결하기 위해 activation function으로 ReLU를 사용하다가 마지막 layer에서 sigmoid를 한 번 사용해주는 복합적인 방식을 채택하였습니다.

(4)-3 p\_keep을 dropout의 확률로 정하여 dropout을 적용했습니다.



Dropout은 신경망의 뉴런을 부분적으로 생략하여 모델의 과적합(overfitting)을 해결해주기 위한 방법 중 하나입니다. 전체 weight를 계산에 참여시키는 것이 아니라, Layer에 포함된 weight 중에서 일부만 참여시키고 나머지 뉴런은 0으로 만듭니다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **dropout** | **0.0** | **0.2** | **0.4** | **0.6** | **0.8** |
| **Loss** | 0.13 | 0.20 | 0.30 | 0.20 | 0.22 |
| **Accuracy** | 0.95 | 0.94 | 0.94 | 0.94 | 0.94 |
| **F1 score** | 0.95 | 0.94 | 0.94 | 0.94 | 0.94 |

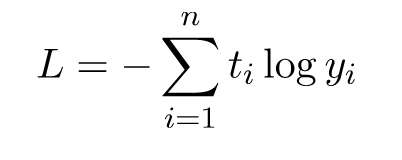
p\_keep은 0.0~1.0 사이의 값을 가지며 1.0은 dropout을 하지 않은 것과 같은 결과입니다. 따라서 overfitting이 발생하는 경우 dropout 값을 1.0에 가깝게 정할수록 error가 높아진다고 합니다. 그런데 p\_keep의 값을 0.0에서 1.0 미만까지 0.2씩 증가시키며 모델을 학습시켜본 결과, loss, accuracy, 그리고 f1 score 모두 거의 유사한 값을 갖는 것을 확인할 수 있었습니다.

(4)-4 model.compile()을 이용하여 모델의 optimizer, loss, 그리고 metrics를 선택합니다.

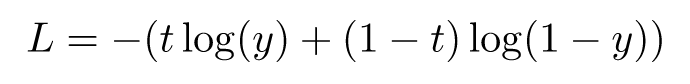
Optimizer는 adam, loss는 binary\_crossentropy, 그리고 metrics는 accuracy와 f1을 선택하였습니다.

딥러닝에서 학습 속도를 개선하기 위해서는 weight parameter들을 최적화하기 위해 optimizer를 사용할 수 있습니다. Optimizer의 종류에는 Gradient Descent(GD), Stochastic Gradient Descent(SGD), Momentum, Adagrad, Adam 등 다양한 것들이 있습니다. 기존의 Gradient Descent 방식은 최적값을 찾아가는 것은 정확하지만 너무 느리다는 문제가 있습니다. 그래서 속도를 개선하기 위해 등장한 것이 SGD 입니다. 그러나 SGD는 최적의 값을 찾아가는 방향이 뒤죽박죽이고, 한 스텝을 나아가기 위한 사이즈를 정하기 어렵습니다. 방향과 스텝 사이즈(또는 learning rate)를 고려한 새로운 optimizer들이 많이 등장하였는데, adam은 방향성과 스텝 사이즈(step size, learning rate)를 모두 고려한 것으로 최근에 가장 많이 사용하고 있는 optimizer입니다.

Cross entropy는 loss function중 하나로, 수식은 아래와 같습니다.



t는 정답 값이고, y는 추론 값입니다. 정답의 개수와 추론의 개수는 같고, 이 개수가 2개이면 이진 분류, 2개보다 많으면 다중 분류입니다. y값은 신경망 여러 개를 거쳐 산출된 값이 최종적으로 어떤 activation function의 입력 값이 되어 산출된 결과입니다.

Binary Cross entropy는 추론 값과 정답 값이 2개(참, 거짓)인 손실 함수입니다. 

참은 1, 거짓은 0을 갖습니다. 구현한 DNN 모델에서의 결과가 0과 1 두 가지 값을 갖기 때문에 Binary Cross Entropy를 사용했습니다.

Metrics는 모델의 성능을 평가하는데 사용되는 함수로, loss function과 비슷하지만 metrics를 통한 성능 평가 결과는 모델을 학습시키는데 사용되지 않는다는 점에서 다릅니다. 모델의 성능 평가 지표로 f1 score를 적용했기 때문에 metrics에 f1을 사용했습니다.

(4)-5 learning rate

Gradient Descent 알고리즘은 Learning rate 또는 Step size라 불리는 스칼라를 곱하여 다음 지점을 결정합니다. 이때 너무 크지도, 작지도 않은 적절한 learning rate를 세팅하여 minimum 값에 효율적으로 도달할 수 있도록 해야합니다.



Learning rate가 너무 큰 경우에는 데이터가 무질서하게 이탈하며, 최저점에 수렴하지 못하게 됩니다. 반대로 learning rate가 너무 작은 경우에는 학습 시간이 매우 오래 걸리며, 최저점에 도달하지 못하게 될 수도 있습니다.

구현한 모델에서 사용한 경사하강법 알고리즘의 종류는 adam입니다. Adam에서는 learning rate의 기본값을 0.001로 갖고 있습니다. 이 값을 변화시키며 모델을 학습시켜본 결과, 0.001보다 작은 learning rate를 갖는 경우 학습 시간이 오래 걸렸고, 0.001보다 큰 learning rate를 갖는 경우 정확도가 작은 폭으로 감소하는 것을 확인했습니다. 따라서 Learning rate 값은 기본값인 0.001을 그대로 적용하였습니다.

(5) batch\_size: 512, epochs: 50 으로 DNN 모델을 학습시킵니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Batch size** | **64** | **128** | **256** | **512** |
| **Loss** | 0.038 | 0.030 | 0.025 | 0.025 |
| **Accuracy** | 0.989 | 0.993 | 0.996 | 0.995 |
| **F1 score** | 0.989 | 0.993 | 0.996 | 0.995 |

Batch size를 바꿔가며 학습시켜 본 결과, batch size가 256 또는 512인 경우 학습 효율이 좋은 것을 발견했습니다. 그리고 Batch size가 크면 학습 속도가 개선되기 때문에 둘 중 512를 최종 batch size로 결정했습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **epochs** | **50** | **100** | **500** | **1000** |
| **Loss** | 0.024 | 0.025 | 0.025 | 0.025 |
| **Accuracy** | 0.995 | 0.995 | 0.995 | 0.995 |
| **F1 score** | 0.995 | 0.995 | 0.995 | 0.995 |

Epoch 값에 변화를 줘 가며 모델을 학습시켜 본 결과, loss, accuracy, 그리고 f1 score 모두 값에 변화가 없음을 확인할 수 있었습니다. 그래서 빠른 결과 도출을 위해 epoch 값을 50으로 두어 학습을 진행하기로 했습니다.

(6) output 값에 대해 class\_weight를 적용합니다. (1: 0.9, 0: 0.1)

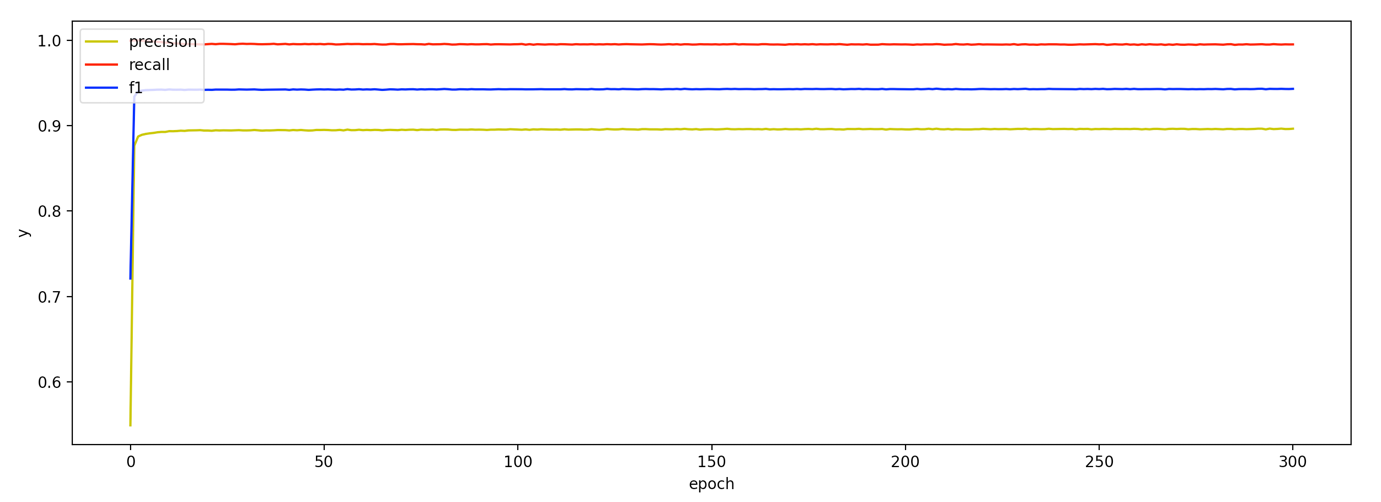
(7) EarlyStopping을 적용하여 f1 값을 관찰하고, 더 이상 training이 필요하지 않은 경우 학습을 조기 종료하도록 합니다.

머신러닝 모델의 한 가지 중요한 점은 적절한 epoch 값을 찾는 것입니다. 너무 많은 epoch은 overfitting을 일으키고, 너무 적은 epoch은 underfitting을 일으킵니다. 이런 상황에서 epoch을 설정하기 위해 EarlyStopping을 사용합니다. EarlyStopping은 Epoch을 많이 돌린 후, 특정 시점에서

5.2 결과 및 분석

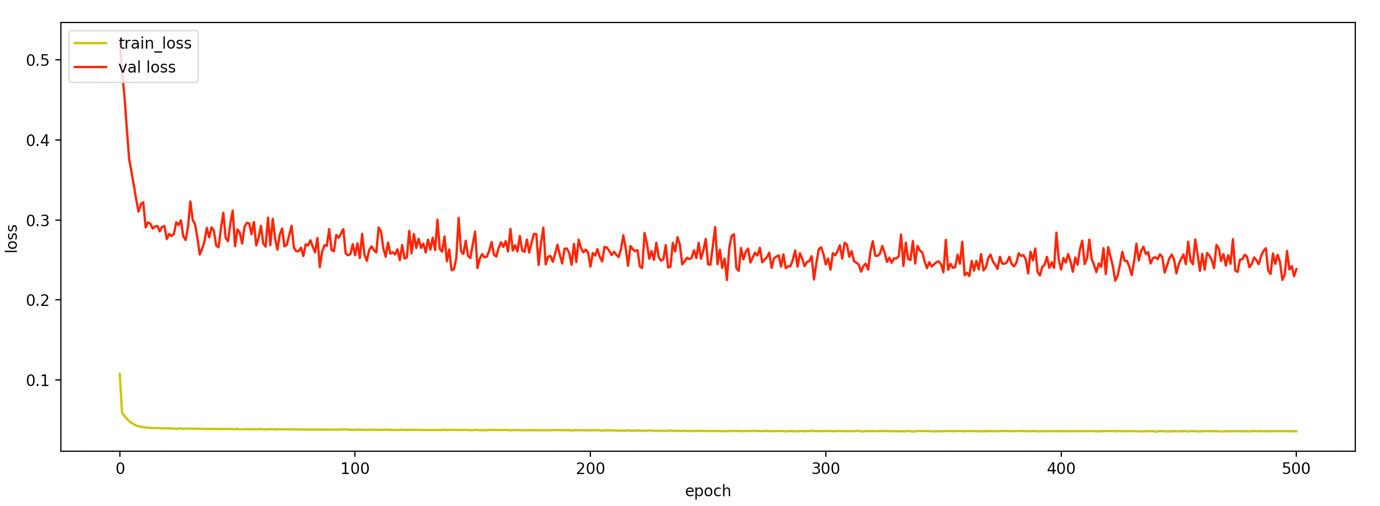
5.2.1 F1 Score, recall, precision

Classifier의 성능을 평가하는 기준으로 주로 사용되는 것에는 Accuracy와 F1 Score가 있습니다. Accuracy는 각 클래스 별로 input되는 데이터의 양이 동일할 때 효과적이고, 조화 평균을 이용하는 F1 Score는 imbalanced data가 주어진 상황에서 효과적입니다. 사용한 데이터가 imbalanced data이기 때문에 성능 평가 지표로 F1 Score를 채택하였습니다.



성능 평가 지표인 f1 score, 그리고 f1 score의 값을 결정짓는 precision과 recall 값의 변화를 시각화하여 나타내었습니다. DNN에서 F1 Score의 값이 0.96, precision의 값이 0.9, 그리고 recall 값은 거의 1.0에 수렴하고 있다는 것을 확인할 수 있습니다.

5.2.2 Loss, val\_loss



Loss는 하나의 모델을 컴파일하기 위해 필요한 매개변수 중 하나로, 훈련 손실 값이라고 합니다. Val\_loss는 검증 손실 값으로 검증 데이터에 대한 손실 값을 의미합니다. Epoch이 진행될수록 loss값은 0.1정도로부터 줄어들어 0에 수렴하고, val\_loss 값은 0.6 정도로부터 줄어들어 0.2에 수렴하는 것을 확인할 수 있습니다.

5.2.3 Confusion Matrix

Confusion Matrix는 분류 모델의 학습 결과, 주어진 데이터를 의도에 맞게 잘 분류했는지 평가하는 기준이 되는 것입니다. Confusion Matrix는 모델이 얼마나 정밀한지, 얼마나 실용적인 분류를 해냈는지, 얼마나 정확한 분류를 했는지를 모두 포함하고 있습니다.

|  |  |
| --- | --- |
| 35067 | 225 |
| 0 | 35291 |

테이블이(가) 표시된 사진

자동 생성된 설명

사기일 경우 레이블 1, 사기가 아닐 경우 레이블 0을 가진 데이터를 분류하는데, 여기서 관심 범주는 레이블 1이라고 할 수 있습니다.

True Positives(TP): 1인 레이블을 1이라 분류한 경우로, 관심 범주를 정확하게 분류한 값입니다.

False Negatives(FN): 1인 레이블을 0이라 분류한 경우로, 관심 범주를 관심 범주가 아닌 것으로 잘못 분류한 값입니다.

False Positives(FP): 0인 레이블을 1이라 하는 경우로, 관심 범주가 아닌 것을 관심 범주라고 잘못 분류한 값입니다.

True Negatives(TN): 0인 레이블을 0이라 하는 경우로, 관심 범주가 아닌 것을 정확하게 분류한 값입니다.

Hyper Parameter 값을 적절히 조절하여 학습시킨 결과, FN값은 전체 35292 중 225개, FP값은 0개로 나타났습니다. 사기 위험도가 높은 제품에 대해서는 일부 정상 제품이 사기라고 분류되고 있으나, 사기인 것은 완벽히 탐지해내고 있습니다.