REPORT



**인공지능 개인과제**

**201701794 최진영**

**목 차**

1. 서 론

1.1 문제설명

2. 본 론

2.1 코드설명

2.2 최적화 방법에 따른 차이

2.2.1 SGD

2.2.2 Adam

2.2.3 RMSprop

2.3 학습률과 epoch에 따른 차이

2.3.1 학습률 0.01 / epoch 200번

2.3.2 학습률 0.001 / epoch 200번

2.3.3 학습률 0.01 / epoch 500번

2.3.4 학습률 0.001 / epoch 1000번

2.3.5 학습률 0.01/ epoch 400번

2.4 손실함수에 따른 차이

2.4.1 MSE

2.4.2 크로스 엔트로피(categorical cross-entropy/binary cross-entropy)

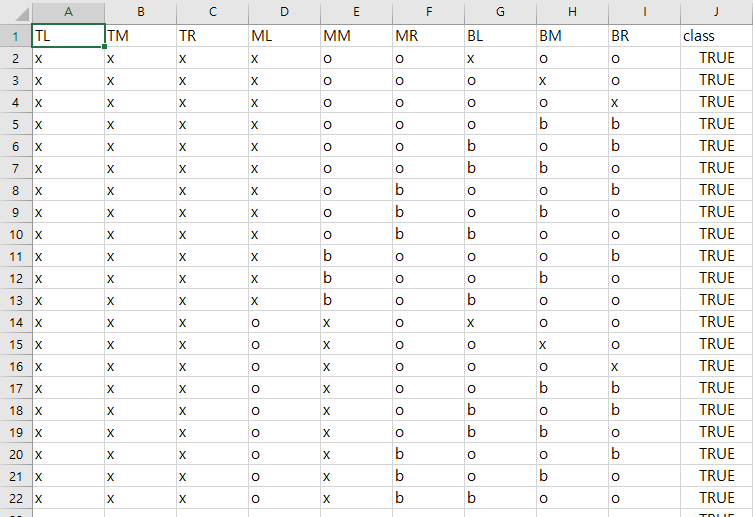
3. 결 론

1. **서론**

삼목 게임인 Tic Tac Toe를 딥러닝으로 학습시켜 결과를 본다. 여러가지 학습방법과 학습률과 epoch을 변동시켜 정확도와 손실률을 측정하고 가장 적절한 방법을 찾아본다. Keras에는 SGD와 Adam, RMSprop를 비교 손실함수 MSE와 categorical\_crossentropy 도 사용한다. 데이터셋은 링크 https://github.com/datasets/tic-tac-toe 에서 tic-tac-toe.csv를 가져왔다. 각각 박스의 위치는 표와 같이 표현되어 있으며 class에서 true는 x가 이긴 것 false는 x가 진 것이다.

|  |  |  |
| --- | --- | --- |
| **TL** | **TM** | **TR** |
| **ML** | **MM** | **MR** |
| **BL** | **BM** | **MR** |

tic-tac-toe.csv의 구성은 다음과 같다.



1. **본론**
   1. **코드 분석**
2. 처음 tensorflow와 numpy, matplotlib을 import 해준다.
3. 데이터 받아오는 함수

def load\_ttt(shuffle=False):

label={'x':0,'o':1,'b':2,'false':0,'true':1}

data = np.loadtxt("./Data/tic-tac-toe.csv", skiprows=1, delimiter=',',

converters={0:lambda name: label[name.decode()],

1:lambda name: label[name.decode()],

2:lambda name: label[name.decode()],

3:lambda name: label[name.decode()],

4:lambda name: label[name.decode()],

5:lambda name: label[name.decode()],

6:lambda name: label[name.decode()],

7:lambda name: label[name.decode()],

8:lambda name: label[name.decode()],

9:lambda name: label[name.decode()]})

if shuffle:

np.random.shuffle(data)

return datadata에 np.loadtxt를 사용하여 ./Data/tic-tac-toe.csv 파일을 불러온다. 첫 줄은 데이터가 아니므로 생략하고 ,로 구분하여 받아온다. x, o, b, TRUE, FALSE는 문자열 형태이기 때문에 lambda함수를 사용해서 문자열들을 각각 0, 1, 2, 0, 1로 바꾸어 주었다. shuffle은 True로 받아오면 데이터들을 랜덤으로 받아오게 된다. 데이터는 numpy 상태로 받아왔다.

def train\_test\_data\_set(ttt\_data, test\_rate=0.4):

n = int(ttt\_data.shape[0]\*(1-test\_rate))

x\_train = ttt\_data[:n,:-1]

y\_train = ttt\_data[:n, -1]

x\_test = ttt\_data[n:,:-1]

y\_test = ttt\_data[n:,-1]

return (x\_train, y\_train), (x\_test, y\_test)테스트 데이터와 학습시킬 데이터를 나눈다 test\_rate를 0.4로 지정해 테스트 데이터는 40%, 학습시킬 데이터는 60%를 사용한다. 불러온 데이터 중에 앞에서 자르는데 x\_train에는 맨마지막을 제외하여 자르고 y\_train은 맨마지막 컬럼만 받아온다. x\_test, y\_test는 테스트 데이터이다.

1. ttt\_data = load\_ttt(shuffle=True)

(x\_train, y\_train), (x\_test, y\_test) = train\_test\_data\_set(ttt\_data, test\_rate=0.4)

print("x\_train.shape:", x\_train.shape) #(574x9)

print("y\_train.shape:", y\_train.shape) #(574x1)

print("x\_test.shape:", x\_test.shape) #(384x9)

print("y\_test.shape:", y\_test.shape) #(384x1)

y\_train = tf.keras.utils.to\_categorical(y\_train)

y\_test= tf.keras.utils.to\_categorical(y\_test)

print("y\_train=", y\_train)

print("y\_test=", y\_test)

x, y의 train데이터와 test데이터의 shape를 보여준다.

1. n = 10

model = tf.keras.Sequential()

model.add(tf.keras.layers.Dense(units=n, input\_dim=9, activation='sigmoid'))

model.add(tf.keras.layers.Dense(units=n, activation='sigmoid'))

model.add(tf.keras.layers.Dense(units=2, activation='softmax'))

model.summary()

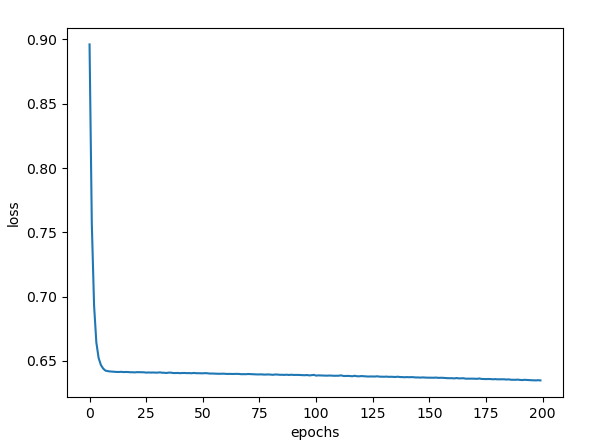
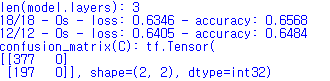
뉴런은 우선 10개로 잡고 모델을 세운다. 입력값은 9개 activation=’sigmoid’로 돌려보았다. 그리고 Dense층은 더 쌓을 수 있다. 입력값은 한번만 입력해주면 되고 마지막에 TRUE, FALSE 2가지로 나뉘니 units=2 이다. 우선 층은 3층을 쌓아 실험했다.

1. ret = model.fit(x\_train, y\_train, epochs=200, verbose=2)

epochs는 학습을 얼마나 시킬 것인가이고 verbose에 값이 있으면 학습되는 모양새를 보여준다.

기본적으로 epoches는 200 Dense층은 3층, categorical\_crossentropy으로 실험했다.

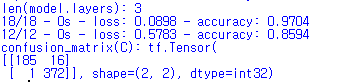
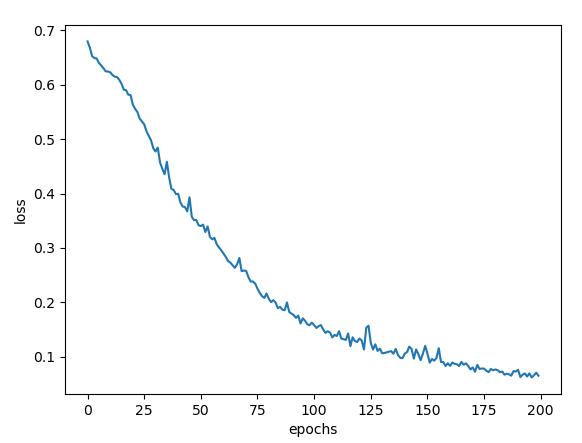
* + 1. **SGD**

SGD로 돌렸을 때는 모든 결과를 한쪽으로 분류하는 문제가 발생했다. 그래서 정확도는 66%로 덜어지게 되었다.

**2.2.2 Adam**

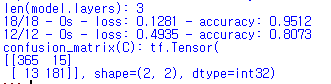
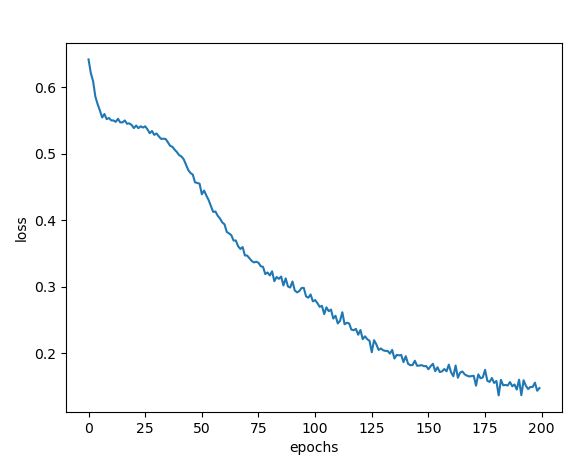
opt = tf.keras.optimizers.Adam(0.01)



Adam은 약 97%의 정확도를 보였다.

**2.2.3 RMSprop**

opt = tf.keras.optimizers.RMSprop(learning\_rate=0.01)



RMSprop은 약 95%의 정확도를 보였다.

가장 적절한 방법은 Adam을 사용하는 것이라는 결과가 나왔다.

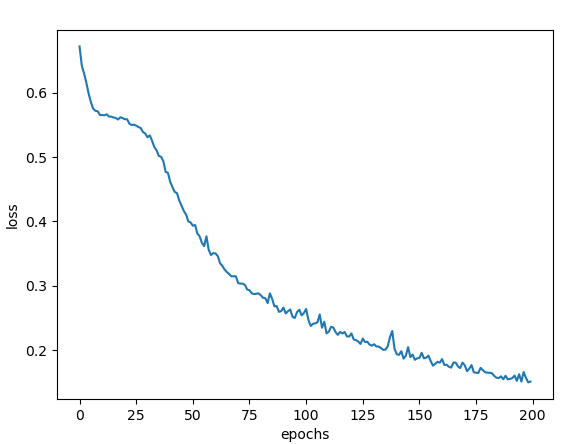
**2.3 학습률과 epoch에 따른 차이**

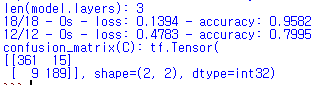
여기서는 위에서 가장 높은 정확도를 보였던 Adam을 사용하여 실험해 보였다.

**2.3.1 학습률 0.01 / epoch 200번**

opt = tf.keras.optimizers.Adam(learning\_rate=0.01)

ret = model.fit(x\_train, y\_train, epochs=200, verbose=2)



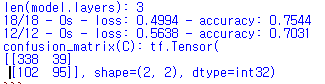
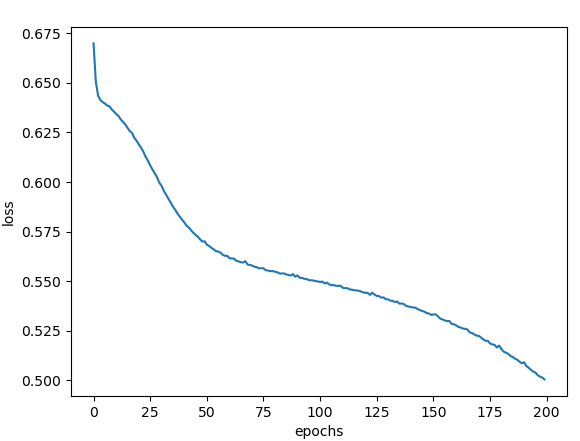


학습율 0.01에 epoches를 200으로 돌렸을 떄 약 96%의 정확도를 보였다.

**2.3.2 학습률 0.001 / epoch 200번**

opt = tf.keras.optimizers.Adam(learning\_rate=0.001)

ret = model.fit(x\_train, y\_train, epochs=200, verbose=2)

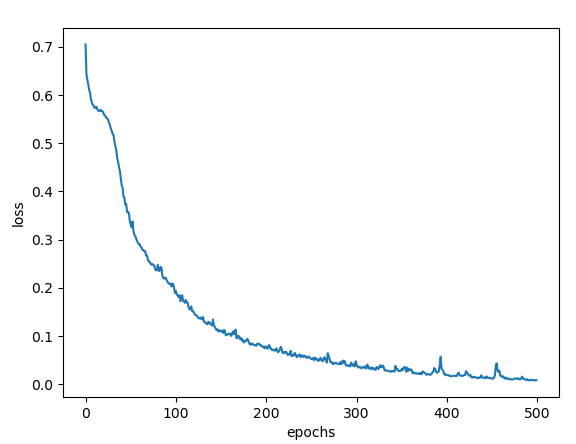


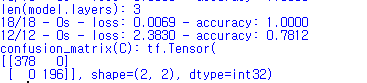
학습율 0.001에 epoches를 200으로 돌렸을 떄 약 75%의 정확도를 보였다.

**2.3.3 학습률 0.01 / epoch 500번**

opt = tf.keras.optimizers.Adam(learning\_rate=0.01)

ret = model.fit(x\_train, y\_train, epochs=500, verbose=2)



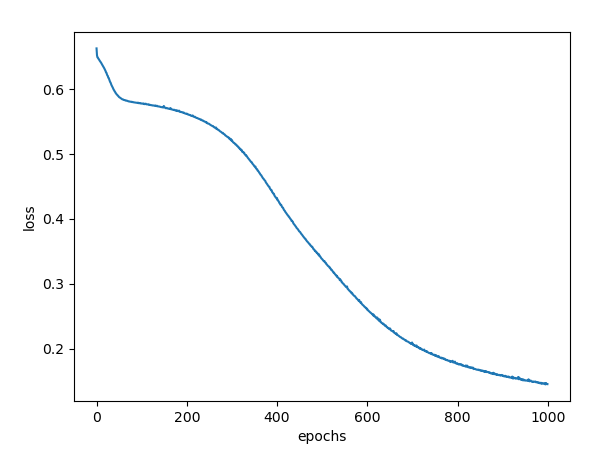


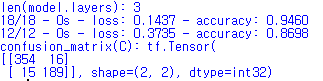
학습율 0.01에 epoches를 500으로 돌렸을 떄 약 100%의 정확도를 보였다.

**2.3.4 학습률 0.001 / epoch 1000번**

opt = tf.keras.optimizers.Adam(learning\_rate=0.001)

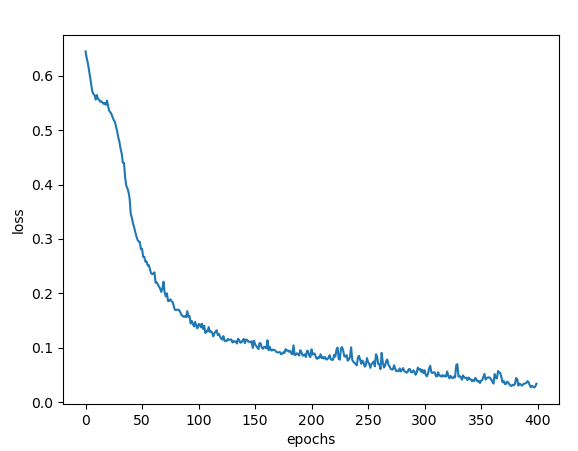
ret = model.fit(x\_train, y\_train, epochs=1000, verbose=2)

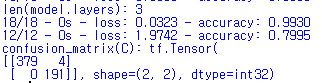




학습율 0.001에 epoches를 1000으로 돌렸을 떄 약 95%의 정확도를 보였다.

**2.3.5 학습률 0.01/ epoch 400번**





학습율 0.01에 epoches를 400으로 돌렸을 떄 약 99%의 정확도를 보였다.

학습율 0.01에 epoches를 400으로 돌리는 정도가 적절할 거 같다는 결과가 나왔다.

**2.4 손실함수에 따른 차이**

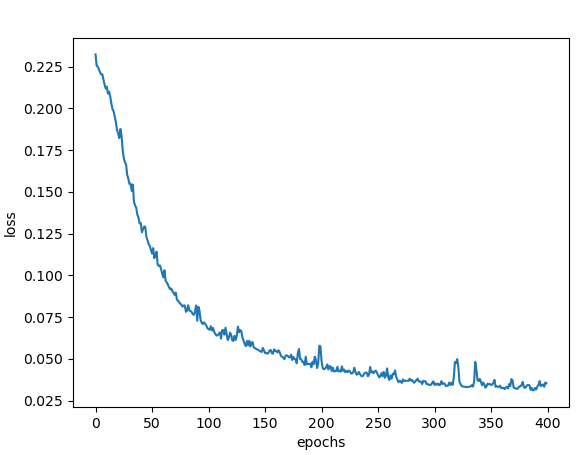
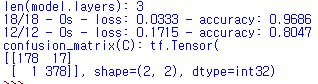
위에서 가장 정확도가 좋았던 epochs=500에서 살짝 줄여 epochs=400과 learning\_rate = 0.01로 실험해보았다.

**2.4.1 MSE**

def MSE(y, t):

return tf.reduce\_mean(tf.square(y - t))

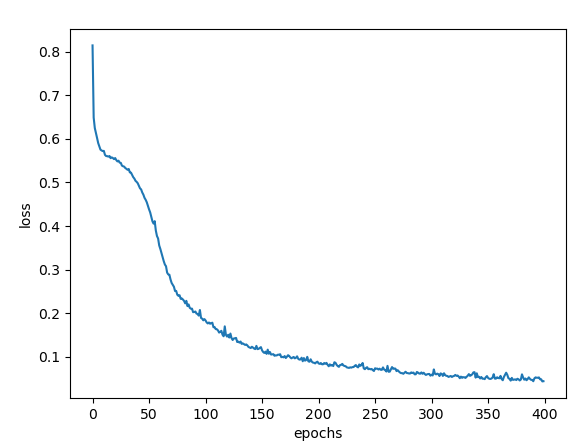
model.compile(optimizer=opt, loss= MSE, metrics=['accuracy'])

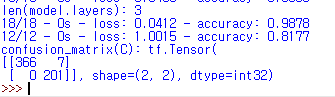
 

MSE로 돌렸을 때 약 98%의 정확도를 보였다.

**2.4.2 categorical\_crossentropy**

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





categorical\_crossentropy로 했을 때 약 99%의 정확도를 보인다.

categorical\_crossentropy로 했을 때가 더 높은 정확도를 보였다.

1. **결론**

여러 번 돌려보고 실험해 봤을 때 keras는 Adam, learning\_rate는 0.01, epoches는 400번, 손실함수는 categorical\_crossentropy로 하는 것이 가장 이상적인 것 같다는 결론을 내렸다. 이렇게 되는 것이 가장 효율적이고 높은 정확도를 얻게 되는 것인 거 같다.

learning\_rate와 epoches에서는 learning\_rate가 낮을수록 epoches는 높게 돌려야 좀더 섬세하고 높은 정확도가 나올 것이다. 하지만 컴퓨터에 따라 낮은 learning\_rate와 높은 epoches를 힘들어하는 컴퓨터도 있을 것이다. 오히려 너무 낮은 learning\_rate와 높은 epoches로 돌렸을 때 정확도에 그만큼의 변화가 없었고 100% 정확도를 보이고 있는 상태에서는 오히려 떨어지거나 쓸떼없이 학습하고 있는 모습을 보였다. 적절한 learning\_rate와 epoches를 찾고 설정하는 것이 중요하다.

최적화 함수와 손실함수도 여러 조건에 따라 잘 맞을 때도 있고 아닐 때도 있는 것 같다. 어떤 함수에는 이정도만 돌려도 높은 정확도를 보이지만 다른 함수는 더 높게 돌려야 그 정도의 정확도를 얻는다든가 등 여러 상황을 고려해야 하는 것 같다. 그렇기에 최적의 모델을 구성하는 것은 어렵다고 느껴졌다. 세상에는 많은 신경망들이 있고 그 신경망들을 효율적으로 사용하기 위해서는 정말 많은 실험들과 많은 상황을 고려해야 한다. 그렇게 과정을 거친 후에 가장 효율이 좋은 조건을 찾고 우리는 그 신경망을 가져와 사용하게 되는 것이다. 이러한 과정들이 있었기 때문에 우리들은 그래도 원하는 결과를 쉽게 얻을 수 있게 되는 것 같다고 생각이 들었다. 이번 인공지능을 들으면서 여러가지로 웹에 서치해보고 찾아보고 했는데 이미 만들어져 있는 것들을 불러와 자신의 주제에 맞게 변형시킨 후 결과를 도출하는 경우가 많아 보였다. 우리는 그냥 불러와서 쓰는 거지만 사실 그 신경망들을 만들기 위해 얼마나 많은 시간과 실험을 쏟아 부었을까 라는 생각이 들었다. 지금 배우는 것들도 많은 수학들과 이해해야 하는 것들이 많았기 때문에 정말 대단하다는 생각들이 들었다.

이제 막 인공지능을 배우고 이해해가는 상황이기 때문에 아직은 많은 것들을 이해할 순 없지만 이런 신경망이라던가 모델들을 배우고 연구해서 나중에 내가 만들고 싶은 웹이나 앱, 또 다른 것들이 있을 때 거기에 인공지능을 더해보고 싶다. 지금은 AI가 활성화된 시대이다. 인공지능은 점점 더 발전해 갈 것이고 우리의 생활 곳곳에 인공지능들이 있을 것이다. 나도 그 시대에 맞게 인공지능을 공부하고 개발에 임하고 싶다.