인공지능프로그래밍

Lab3

Optimizers for Deep Neural Networks

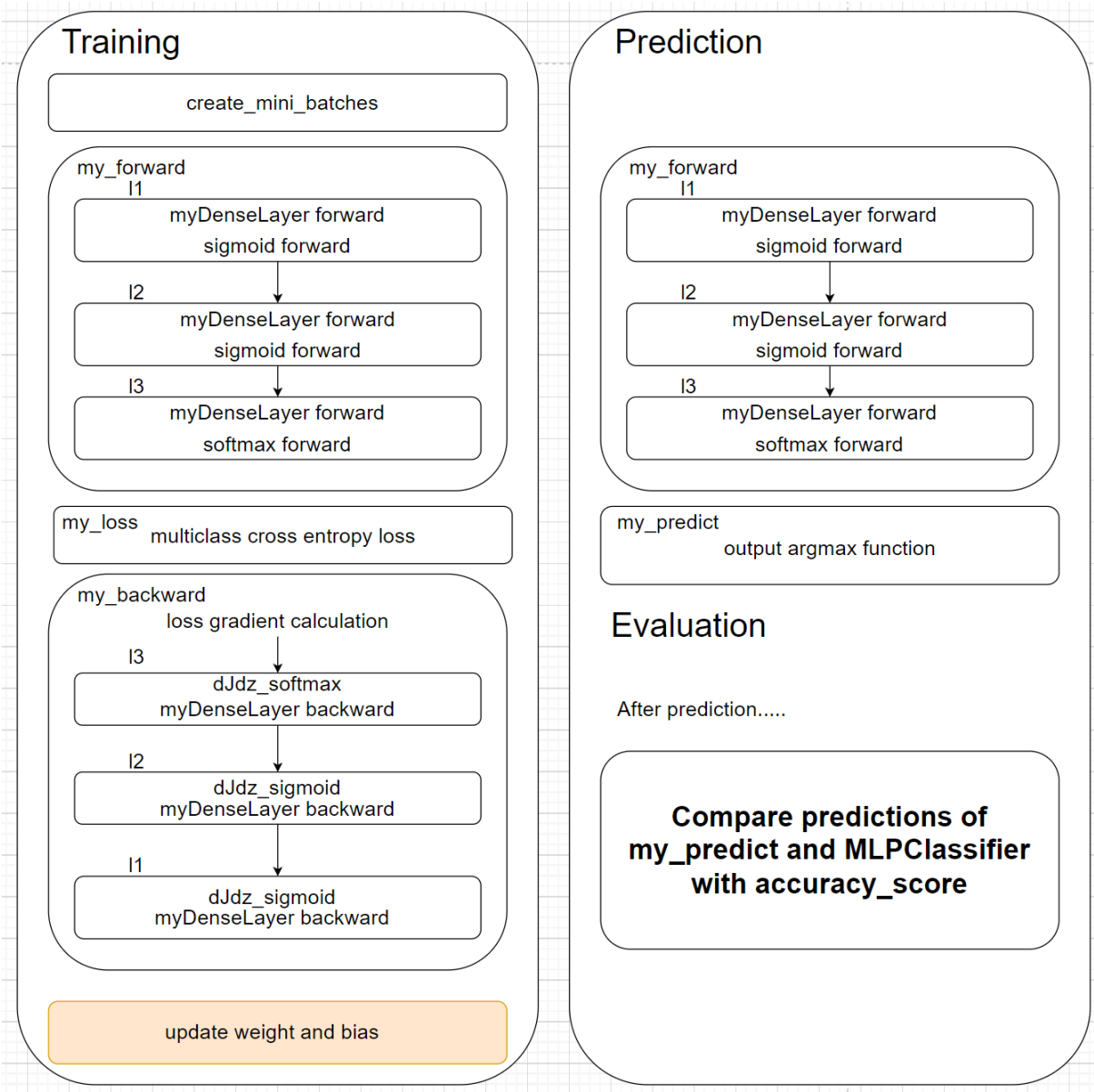
2024/10/07

2019202050 이강현

Lab Objective

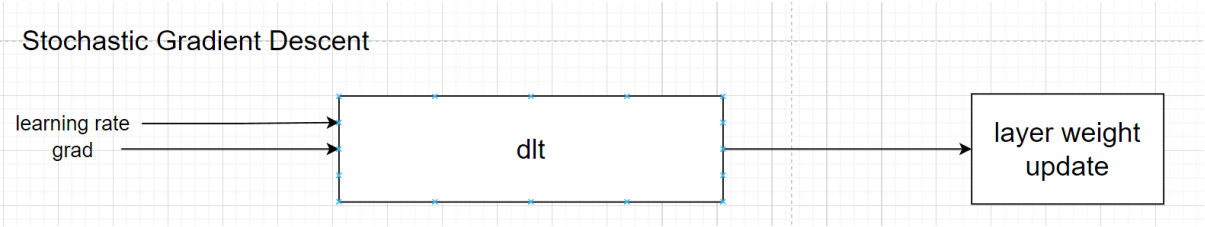
Lab2와 마찬가지로 MNIST Dataset를 이용하여 DNN을 직접 구현한다. DNN 구현 시 추가적으로 전체 데이터를 mini-batch로 나누고 optimizer를 직접 구현하여 사용해본다. 다양한 optimizer를 사용하여 DNN을 학습시키고 해당 결과를 비교해보며 구현한 네트워크에 대해 최적 파라미터를 찾는 과정을 익힌다.

Program Flow

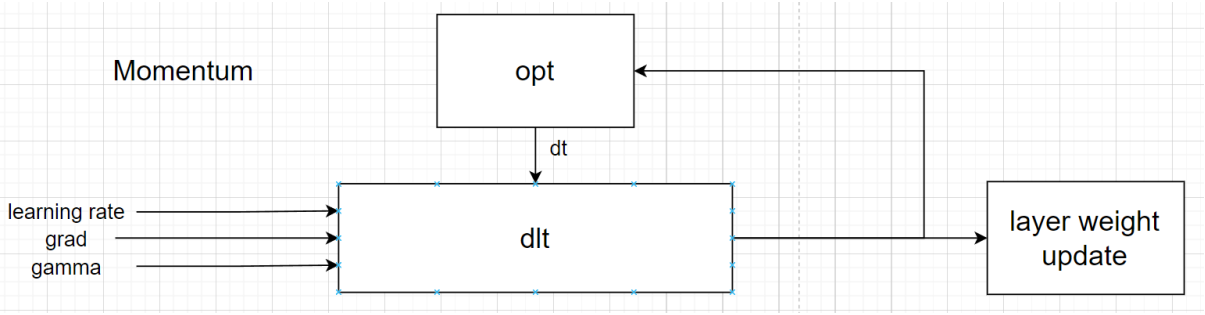


위 그림은 전체 프로세스를 도식화한 것이다.

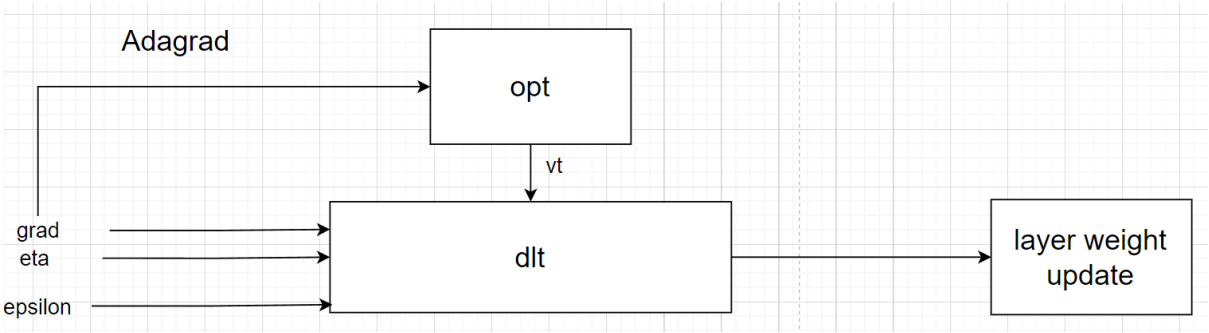
전체적인 프로세스는 Lab2과 유사하지만 훈련 시 mini-batch로 나누어 진행된다는 점과 weight와 bias를 update할때 추가적으로 다양한 optimizer를 사용한다는 점에 차이가 있다. 주황색으로 색칠한 부분에서 사용된 optimizer들의 흐름도를 아래와 같이 간략히 나타내었다.



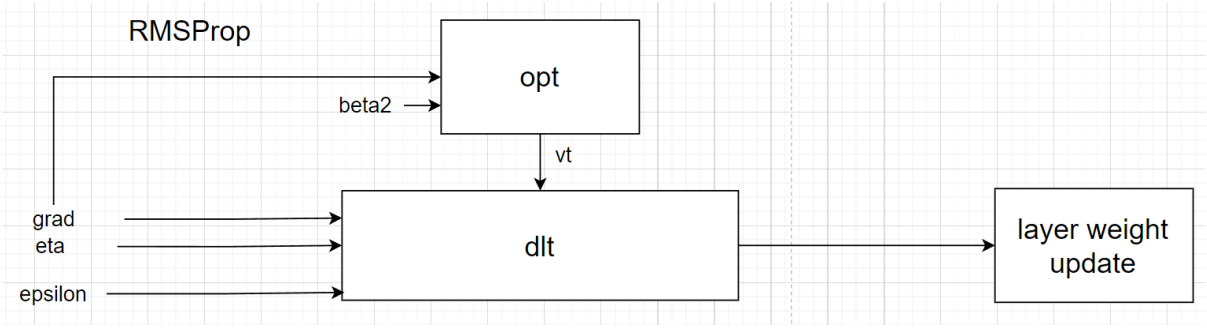
Stochastic Gradient Descent는 입력받은 gradient에 learning rate만 곱한 값으로 weight를 업데이트 한다.



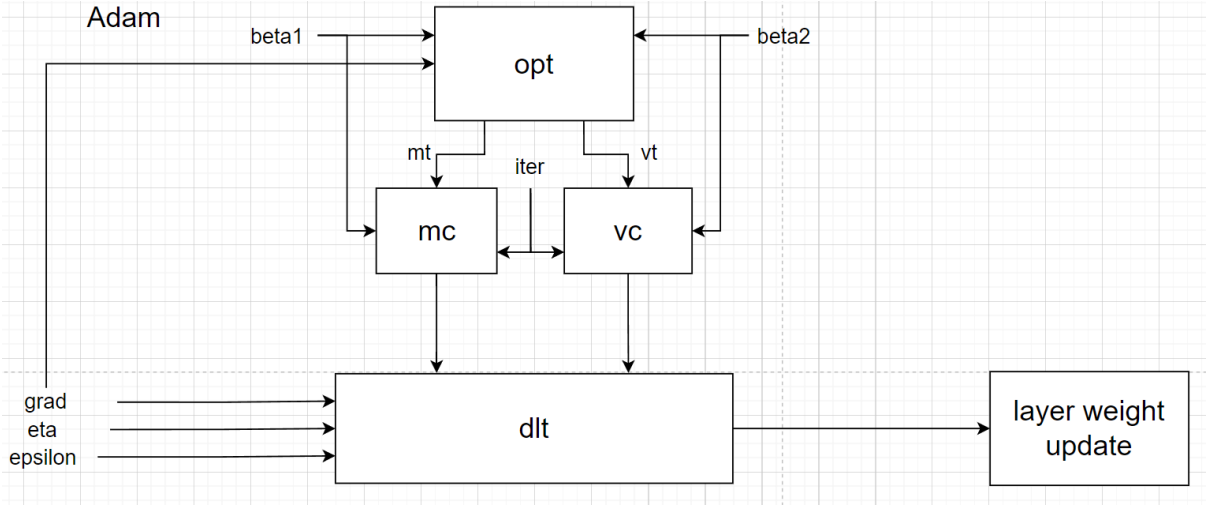
Momentum은 gamma값을 이용하여 이전 gradient에 gamma를 곱하고 현재 gradient에 learning rate를 곱한 값으로 weight를 업데이트한다.



Adagrad는 2차 미분값의 크기를 이용하여 gradient의 변화가 급격할때는 천천히 완만할때는 빠르게 가게끔 weight를 업데이트한다. 또 epsilon을 넣어 division by zero가 발생하지 않도록 한다.



RMSProp은 AdaGrad의 2차미분의 momentum을 추가한 형태로 이전 변화를 조금 반영한 업데이트 방식이다.



Adam은 1차,2차 미분을 모두 momentum과 함께 사용하고 추가로 epoch을 사용하여 weight가 학습의 초반에는 빠르게 나중엔 momentum을 그대로 사용하게끔 한하여 weight를 업데이트한다.

Result

코드의 주요 부분과 함께 결과를 살펴보자.

먼저 DNN의 forward와 backward에 대한 내용은 Lab2의 내용을 복사해서 구현하라고 제시되어 생략하고 추가로 mini-batch를 생성하는 부분과 optimizer를 구현한 부분에 집중하여 설명을 진행한다.

```
def create_mini_batches(X, y, batch_size=64):
    mini_batches = []
    n_minibatches = (X.shape[0] // batch_size)
    n_variables = X.shape[1]
    ### START CODE HERE ###

    data = np.hstack((X,y))      # concatenate X and y with np.hstack
    np.random.shuffle(data)      # then shuffle it

    for i in range(n_minibatches):
        mini_batch = data[i*batch_size:(i+1)*batch_size,:]        # get a slice of mini-batch
        X_mini, y_mini = np.split(mini_batch,[-y.shape[1]],axis=1)    # split mini-batch into X & y
        mini_batches.append((X_mini, y_mini))

    if data.shape[0] % batch_size != 0:
        mini_batch = data[n_minibatches*batch_size:,:]        # process the remaining data
        X_mini, y_mini = np.split(mini_batch,[-y.shape[1]],axis=1)    # split mini-batch into X & y
        mini_batches.append((X_mini, y_mini))

    ### END CODE HERE ###
    return mini_batches
```

먼저 X라는 훈련데이터와 y라는 정답데이터를 서로 열을 기준으로 이어 붙인다. 그러면 전체 샘플 수는 변하지 않고 훈련데이터의 샘플마다 variable과 label이 함께 존재하는 모양이 된다. 그 후 샘플들을 섞어 순서가 훈련에 영향을 미치지 않게 만든다. 이후 사용할 미니배치의 크기만큼 전체 데이터를 나눠주고 그 값만큼 반복하여 미니 배치로 전체 데이터를 split한다. split한 데이터에서 합쳐두었던 정답 데이터를 분리해주고 튜플로 묶어 list에 넣어준다. 추가적으로 나누어 떨어지지 않고 남은 데이터에 대한 코드를 if문으로 작성한다.

```
np.random.seed(1)
#test code
a = np.arange(20).reshape(10,2)
b = -np.arange(10,20).reshape(10,1)
c = create_mini_batches(a, b, 4)
for mini_X, mini_y in c:
    print(mini_X)
    print(mini_y, '\n')
```

```
[[ 4  5]
 [18 19]
 [12 13]
 [ 8  9]]
[[-12]
 [-19]
 [-16]
 [-14]]

[[ 0  1]
 [ 6  7]
 [ 2  3]
 [14 15]]
[[-10]
 [-13]
 [-11]
 [-17]]

[[16 17]
 [10 11]]
[[-18]
 [-15]]
```

샘플 수가 10개인 데이터를 4개씩 미니배치로 만들었을때 위와 같이 나오는 것을 확인한다.

다음은 optimizer에 대한 설명이다.

```
    # optimizer routines
    if  solver=='sgd':
        W_dlt = alpha * W_grad
        B_dlt = alpha * B_grad
```

sgd는 backpropagation을 이용해 구해낸 gradient에 learning rate만 곱해 업데이트하는 것을 확인할 수 있다.

```
    elif solver=='momentum':
        gamma = 0.9                # default setting
        ### START CODE HERE ###

        W_dlt = gamma*opt.W_dt + alpha*W_grad              # momentum for previous delta
        B_dlt = gamma*opt.B_dt + alpha*B_grad              # same goes for bias
        opt.W_dt = W_dlt           # keep data for later use
        opt.B_dt = B_dlt           # for bias, too
```

momentum은 먼저 이전 gradient와 현재 gradient를 각각 hyperparameter인 gamma와 alpha를 곱하고 더하는 방식으로 업데이트에 사용할 dlt들을 만들고 이들로 다시 미래에 사용할 현재 gradient를 저장해두는 것을 확인할 수 있다.

```
        ### END CODE HERE ###
    elif solver=='adagrad':
        ### START CODE HERE ###

        opt.W_vt = opt.W_vt + (W_grad*W_grad)           # accumulate delta square (2nd momentum)
        opt.B_vt = opt.B_vt + (B_grad*B_grad)           # accumulater for bias term
        W_dlt = (eta/np.sqrt(opt.W_vt+epsilon))*W_grad              # calculate new delta for weight
        B_dlt = (eta/np.sqrt(opt.B_vt+epsilon))*B_grad              # and for bias
```

adagrad는 2차 미분값을 과거의 값과 현재 gradient을 사용하여 먼저 업데이트하고 그 값을 이용하여 dlt들을 update하는 것을 확인한다.

```
        ### END CODE HERE ###
    elif solver=='rmsprop':
        beta2 = 0.9               # default setting
        ### START CODE HERE ###

        opt.W_vt = beta2*opt.W_vt + (1-beta2)*(W_grad*W_grad)         # blending with second momentum
        opt.B_vt = beta2*opt.B_vt + (1-beta2)*(B_grad*B_grad)           # also doging samething for bias
        W_dlt = (eta/np.sqrt(opt.W_vt+epsilon))*W_grad              # calculate new delta for weight
        B_dlt = (eta/np.sqrt(opt.B_vt+epsilon))*B_grad             # and for bias
```

rmsprop은 adagrad에 second momentum이 추가된 형태이다.

```
        ### END CODE HERE ###
    elif solver=='adam':
        beta1, beta2 = 0.9, 0.99  # default setting
        ### START CODE HERE ###

        opt.W_mt = beta1*opt.W_mt + (1-beta1)*W_grad           # blending with first momentum
        opt.B_mt = beta1*opt.B_mt + (1-beta1)*B_grad           # first momentum for bias
        opt.W_vt = beta2*opt.W_vt + (1-beta2)*(W_grad*W_grad)          # blending with second momentum
        opt.B_vt = beta2*opt.B_vt + (1-beta2)*(B_grad*B_grad)          # second momentum for bias
        W_mc = opt.W_mt / (1-np.power(beta1,iter))                # bias correction of first momentum for weight
        B_mc = opt.B_mt / (1-np.power(beta1,iter))                # and for bias term
        W_vc = opt.W_vt / (1-np.power(beta2,iter))                # bias correction of second momentum for weight
        B_vc = opt.B_vt / (1-np.power(beta2,iter))                # and for bias term
        W_dlt = (eta/(np.sqrt(W_vc)+epsilon))*W_mc              # calculate new delat for weight
        B_dlt = (eta/(np.sqrt(B_vc)+epsilon))*B_mc              # and for bias
```

adam은 1차 미분값의 momentum, 2차 미분값의 momentum, 학습 횟수에 따른 momentum 반영정도를 모두 고려하여 weight를 업데이트하는 것을 확인한다.

```
    # Adjust weight
    lyr.wegt = lyr.wegt - W_dlt
    lyr.bias = lyr.bias - B_dlt

    return
```

마지막으로 다양한 optimizer들이 생성한 W_dlt,B_dlt를 이용하여 layer의 weight와 bias를 업데이트하는 것으로 마무리된다.

```
np.random.seed(101)
#random initializing
lyr = myDenseLayer(2,3)
opt = myOptParam(2,3)

lyr.wegt = np.random.randn(2,3)
lyr.bias = np.random.randn(2)
opt.W_dt = np.random.randn(2,3)
opt.B_dt = np.random.randn(2)
opt.W_mt = np.random.randn(2,3)
opt.B_mt = np.random.randn(2)
opt.W_vt = np.abs(np.random.randn(2,3))
opt.B_vt = np.abs(np.random.randn(2))

W_grad = np.random.randn(2,3)
B_grad = np.random.randn(2)

# optimizer settings are: 'sgd', 'momentum', 'adagrad', 'rmsprop', 'adam'
opts = ['sgd', 'momentum', 'adagrad', 'rmsprop', 'adam']
expt = [[ 7.67789007,  8.16882972, 10.34203348, -3.22934657],
        [14.46528172,  15.04341688, 19.30016537, -4.77070266],
        [22.50872929,  22.74302212, 28.47667875, -7.62607443],
        [30.69802889,  30.60433129, 37.72651766, -10.62235939],
        [29.41774022,  19.27573813, 23.68071186,  1.52919472]]
test_passed = True
for i, sol in enumerate(opts):
    my_optimizer(lyr, opt, W_grad, B_grad, sol, 10, 3)
    print('For '+sol+':')
    res = np.concatenate((lyr.wegt[0], lyr.bias[0:1]), axis=0)
    print(res)
    if not np.allclose(res, expt[i]):
        print(sol+' failed.')
        test_passed = False
if test_passed: print('test passed.')
else: print('test failed.')
```

```
For sgd:
[ 7.67789007  8.16882972 10.34203348 -3.22934657]
For momentum:
[14.46528172 15.04341688 19.30016537 -4.77070266]
For adagrad:
[22.50872929 22.74302212 28.47667875 -7.62607443]
For rmsprop:
[ 30.69802889  30.60433129  37.72651766 -10.62235939]
For adam:
[29.41774022 19.27573813 23.68071186  1.52919472]
test passed.
```

각각의 optimizer들이 잘 구현되었는지 확인하는 코드이다. 랜덤한 값들로 초기화를 진행한 후 예상치와 동일하게 결과를 얻는지 확인한다.

```
# optimizer settings are: 'sgd', 'momentum', 'adagrad', 'rmsprop', 'adam'
# alpha is learning rate
optimizer ='sgd'
alpha = 0.01
n_epochs = 1000

for epoch in range(n_epochs):

    batches = create_mini_batches(X_train, y_train, batch_size=64)
    for one_batch in batches:
        X_mini, y_mini = one_batch
        batch_len = X_mini.shape[0]  # last batch might have different length

        # Forward Path
        a_1, a_2, a_3 = my_forward(layers, X_mini)

        # Backward Path
        d_1, d_2, d_3 = my_backward(layers, a_1, a_2, a_3, X_mini, y_mini)

        dw_1, db_1 = d_1
        dw_2, db_2 = d_2
        dw_3, db_3 = d_3

        # Update weights and biases
        my_optimizer(l1, o1, dw_1, db_1, solver=optimizer, learning_rate=alpha, iter=epoch+1)
        my_optimizer(l2, o2, dw_2, db_2, solver=optimizer, learning_rate=alpha, iter=epoch+1)
        my_optimizer(l3, o3, dw_3, db_3, solver=optimizer, learning_rate=alpha, iter=epoch+1)

    if ((epoch+1)%100==0):
        loss_J = my_loss(layers, X_train, y_train)
        print('Epoch: %4d,  loss: %10.8f' % (epoch+1, loss_J))
```

구현한 함수들을 모두 이용해 mini-batch를 생성하고 forward연산, backward연산을 한 후 optimizer들을 각 레이어마다 사용하여 weight를 업데이트하는 것으로 코드가 구성되어 있다. 아래는 optimizer에 따른 결과이다.

```
Epoch:  100,  loss: 0.00449811
Epoch:  200,  loss: 0.00203723
Epoch:  300,  loss: 0.00129607
Epoch:  400,  loss: 0.00094257
Epoch:  500,  loss: 0.00073698
Epoch:  600,  loss: 0.00060322
Epoch:  700,  loss: 0.00050957
Epoch:  800,  loss: 0.00044048
Epoch:  900,  loss: 0.00038747
Epoch: 1000,  loss: 0.00034556
```

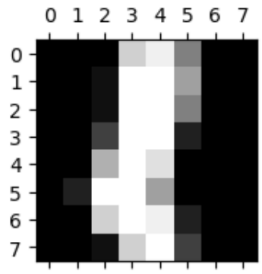
Evaluate Model Performance

```
from sklearn.metrics import accuracy_score

y_pred = my_predict(layers, X_test)

accuracy_score(y_pred, y_test)
```

```
0.9666666666666667
```



```
My prediction is 1
sk prediction is 1
Actual number is 1
```

-SGD-

```
Epoch:  100,  loss: 0.00022031
Epoch:  200,  loss: 0.00011587
Epoch:  300,  loss: 0.00007929
Epoch:  400,  loss: 0.00006038
Epoch:  500,  loss: 0.00004875
Epoch:  600,  loss: 0.00004085
Epoch:  700,  loss: 0.00003513
Epoch:  800,  loss: 0.00003080
Epoch:  900,  loss: 0.00002741
Epoch: 1000,  loss: 0.00002469
```

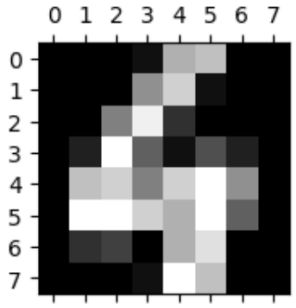
Evaluate Model Performance

```
from sklearn.metrics import accuracy_score

y_pred = my_predict(layers, X_test)

accuracy_score(y_pred, y_test)
```

```
0.975
```



```
My prediction is 4
sk prediction is 4
Actual number is 4
```

-momentum-

```
Epoch:  100,  loss: 0.07038199
Epoch:  200,  loss: 0.03139687
Epoch:  300,  loss: 0.01973312
Epoch:  400,  loss: 0.01442180
Epoch:  500,  loss: 0.01136267
Epoch:  600,  loss: 0.00937024
Epoch:  700,  loss: 0.00797034
Epoch:  800,  loss: 0.00693234
Epoch:  900,  loss: 0.00612384
Epoch: 1000,  loss: 0.00549021
```

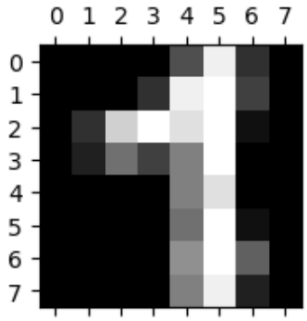
Evaluate Model Performance

```
from sklearn.metrics import accuracy_score

y_pred = my_predict(layers, X_test)

accuracy_score(y_pred, y_test)
```

```
0.95
```



```
My prediction is 1
sk prediction is 1
Actual number is 1
```

-adagrad-

```
Epoch:  100,  loss: 0.00000026
Epoch:  200,  loss: 0.00000012
Epoch:  300,  loss: 0.00000007
Epoch:  400,  loss: 0.00000005
Epoch:  500,  loss: 0.00000004
Epoch:  600,  loss: 0.00000004
Epoch:  700,  loss: 0.00000003
Epoch:  800,  loss: 0.00000003
Epoch:  900,  loss: 0.00000002
Epoch: 1000,  loss: 0.00000002
```

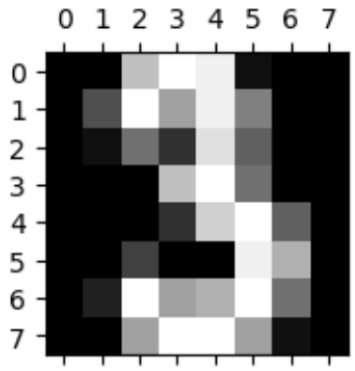
Evaluate Model Performance

```
from sklearn.metrics import accuracy_score

y_pred = my_predict(layers, X_test)

accuracy_score(y_pred, y_test)
```

```
0.9722222222222222
```




```
My prediction is 3
sk prediction is 3
Actual number is 3
```

-rmsprop-

```
Epoch:  100,  loss: 0.00000021
Epoch:  200,  loss: 0.00000000
Epoch:  300,  loss: 0.00000000
Epoch:  400,  loss: 0.00000000
Epoch:  500,  loss: 0.00000000
Epoch:  600,  loss: 0.00000000
Epoch:  700,  loss: 0.00000000
Epoch:  800,  loss: 0.00000000
Epoch:  900,  loss: 0.00000000
Epoch: 1000,  loss: 0.00000000
```

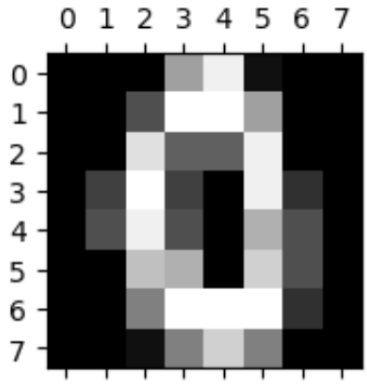
Evaluate Model Performance

```
from sklearn.metrics import accuracy_score

y_pred = my_predict(layers, X_test)

accuracy_score(y_pred, y_test)
```

```
0.9777777777777777
```




```
My prediction is 0
sk prediction is 0
Actual number is 0
```

-adam-

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(80, 70, ), activation='logistic', solver='sgd', \
                    alpha=0.01, learning_rate_init=0.01, max_iter=1000)

# Training/Fitting the Model
mlp.fit(X_train, y_train_num)

# Making Predictions
s_pred = mlp.predict(X_test)
accuracy_score(s_pred, y_test)
```

```
0.9777777777777777
```

-MLPClassifer-

결과적으로 각 optimizer들을 모두 비교한 결과 adam의 정확도가 가장 높았다.

많이 실행해보고 평균과 표준오차도 뽑아보아야 더 일반적인 결과를 얻을 수 있으나 대략적으로 모든 optimizer들이 준수한 성능을 보여준다. loss값은 running everage를 고려한 optimizer들이 압도적으로 낮은 것을 확인하였고 이는 최적점으로 향하는 방향이 샘플마다 조금씩 달라 왔다갔다하면서 학습되는 것보다 방향성을 유지하면서 학습되는 것이 더 빠르게 최적점에 도달하지 않았나라고 예상한다. 다만 낮은 loss값에도 불구하고 학습을 계속 진행했기에 validation 데이터를 통한 overfitting여부를 확인하고 적절한 epoch수를 찾아내는 추가 작업을 할 수 있으나 test data에 대해 높은 정확도를 보이는 것으로 보아 overfitting 위험은 다소 낮을 것으로 예상한다.

Discussion

DNN에 대해 mini-batch를 이용하여 학습을 진행해보고 다양한 optimizer를 직접 구현해보면서 네트워크가 학습되는 양상이나 그에 따른 성능을 loss, accuracy와 같은 지표로 확인해보았다. 강의자료에 나와있는 수식들을 직접 코드로 구현해보니 실제로 어떤 성능차이가 있는지 분석할 수 있어 학습에 용이했다. 하지만 딥러닝 특성상 내부적으로 가중치를 특정 방식으로 학습하는 것이 어떻게 학습이 되고 있는지 시각화하기 어려워 단순한 metric들만 이용해서 예상할 수 밖에 없어 다소 아쉬웠다. 하지만 내 데이터셋이 어떤 특성을 가지는지 모델이 좋은 모델로써 작동하기 위해 어떻게 학습되어야 하는지와 같은 문제를 해결하기 위한 방향성을 제시한다는 측면에서 이번 과제가 유의미하다고 생각한다.