**Assignment 1 Report**

심명진

Q1: k-Nearest Neighbor classifier

Q2: Training a Support Vector Machine

Q3: Implement a Softmax classifier

Q4: Two-Layer Neural Network

Q5: Higher Level Representations: Image Features

**[FULL CODE]** assignment 1

<https://github.com/myeongmy/deepLearning/tree/master/assignment1>

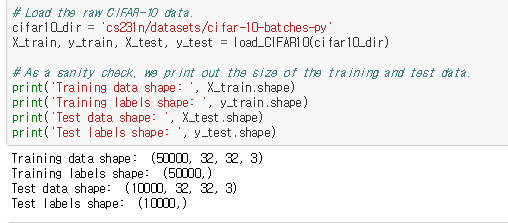
Q1 : k-Nearest Neighbor classifier

Knn 분류기는 다른 선형 분류기나 신경망 네트워크, CNN과는 다르게 데이터 중심 접근방법을 이용한다. 각 카테고리 별로 이미지들을 수집하여 이미지와 라벨 자체를 모델에 훈련시킨다. 훈련 시간은 적게 드는 반면, 새로운 이미지에 대한 prediction이 오래 걸린다는 단점이 있다.

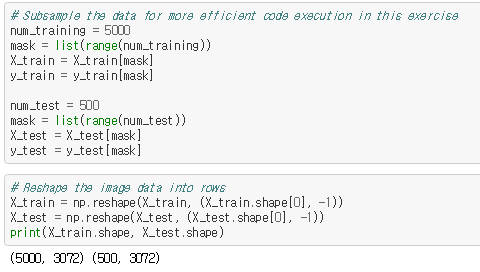
* 이 과제에서 구현해야 할 것

1. Image classification 과정 이해
2. 하이퍼파라미터인 k를 구하기 위한 cross-validation 시행

<knn.ipynb>



먼저 데이터셋(CIFAR-10)을 로드하여 training data와 그에 대한 라벨 데이터, testing data와 그에 대한 라벨 데이터를 마련한다. Training data는 50000개, testing data는 10000개에 해당한다.

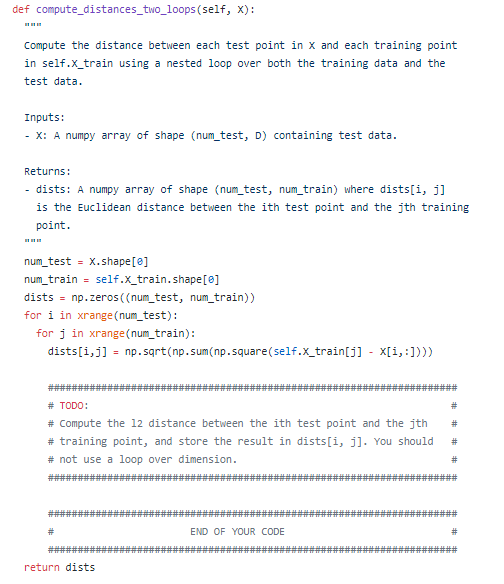


이 예제에서는 training data로 5000개, testing data로 500개만 이용하기로 한다. 그리고 각 이미지 데이터를 32\*32\*3 열로 길게 펼친다.

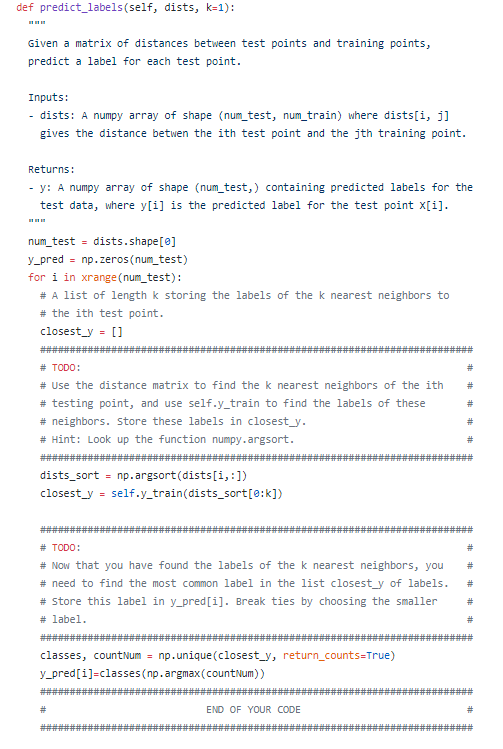
<k\_nearest\_neighbor.py>

Training data를 k\_nearest\_neighbor.py의 train 함수로 훈련시킨 뒤, 500개의 testing data와 5000개의 training data 간의 거리 비교를 하려고 한다.

먼저 두 개의 반복 loop를 이용한 방법이다.

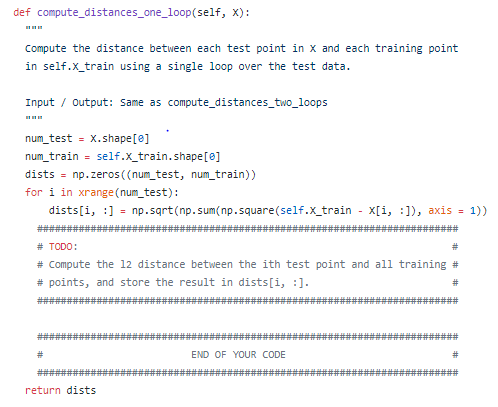


input으로 testing data X를 받아 testing data의 개수를 num\_test 변수에 저장하고 인스턴스 변수인 X\_train을 통해 training data의 개수를 num\_train에 저장한다. Dists 배열을 (500, 5000) shape의 모든 원소가 0인 배열로 초기화를 한 후, 각 for문을 돌면서 인덱스 하나하나씩 채워간다. 우리는 L2 distance를 구해야 하므로 500개의 testing data를 모든 training data와 한 번씩 비교를 해나가는데 각 셀 값의 차를 제곱하고 그 값들을 전부 더하여 루트 연산을 적용하면 해당 training data와 해당 testing data의 거리 차이가 계산된다. 그리고 그 dists 배열을 리턴한다.



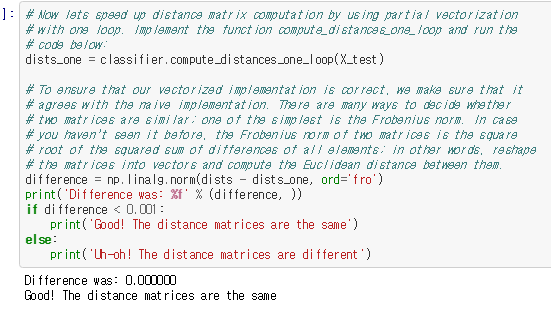
위에서 구한 dists 배열을 가지고 predict\_labels 함수를 이용해 testing data에 대한 라벨 값을 예측한다. Testing data 500개 하나하나 for문을 돌면서 예측되는 라벨을 구해 y\_pred 배열을 완성한다. Closest\_y에는 5000개의 training data와의 거리 데이터 중 가장 작은 거리에 해당하는 k개를 골라 그에 해당하는 라벨들을 저장한다. 그리고 그 closest\_y에 unique 연산을 실시해 각 클래스(라벨)별로 개수를 구한다. 그래서 개수가 가장 큰 클래스(라벨)을 해당 testing data의 라벨 값으로 최종 결정한다.

이제는 거리 측정 연산의 속도를 높이기 위해 loop를 한 번만 사용하기로 한다.

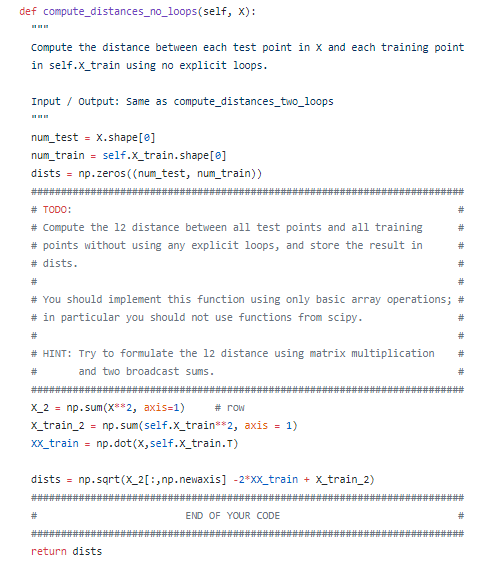


Loop는 testing data 500개에 대해서만 하나 돌며 dists[i]를 각각 구한다. Numpy는 vectorized 연산을 제공하므로 self.X\_train – X[I]를 하면 각 행별로 뺄셈을 한 결과가 나오게 되고 그것을 제곱하여 axis = 1로 덧셈을 하면 각 행별로 덧셈이 이루어진다. 그래서 그것에 루트 연산을 해주게 되면 하나의 testing data에 대한 거리 측정값을 구하게 된다.

* 결과 : loop를 두 개를 이용하나 하나를 이용하나 거리 측정 값에는 변화가 없음을 알 수 있다.

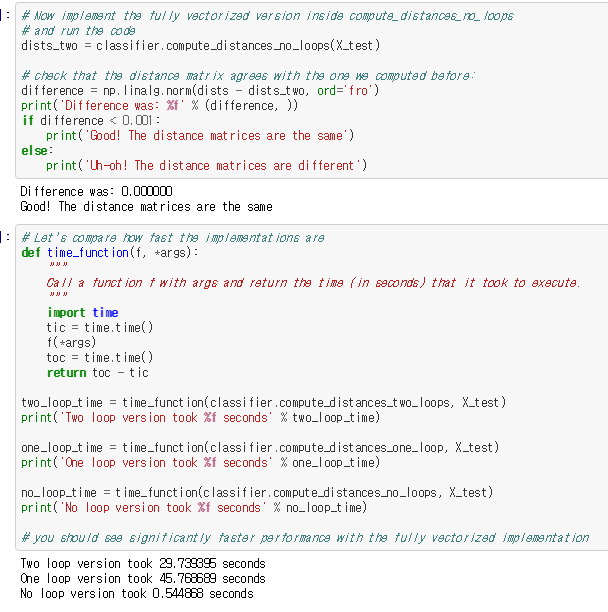


마지막으로 loop를 아예 사용하지 않고 fully vectorized하게 거리 측정 함수를 구현해 보면



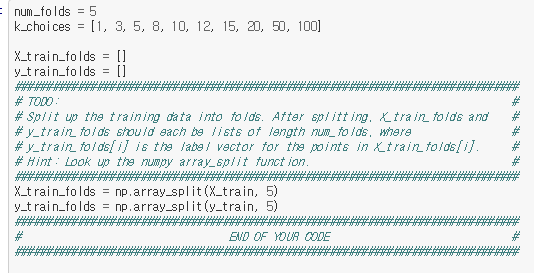
Loop를 쓰지 않고 구현하기 위해 l2 distance를 구하는 공식, 즉 완전제곱식의 형태를 풀어서 계산해보았다. 따라서, testing data의 제곱 값을 따로 구하고, training data의 제곱 값을 따로 구하고, 그 둘의 내적 곱을 따로 구해 마지막에 덧/뺄셈을 통해 dists를 구하였다.

* 결과 : 이것도 마찬가지로 앞에서 구한 two loop를 이용해서 구하는 방식과 결과 값은 다르지 않았다. 하지만 실행 시간 면에서는 훨씬 더 빠른 속도를 보였다. (왜 one loop가 더 오래 걸리는지 모르겠습니다..)

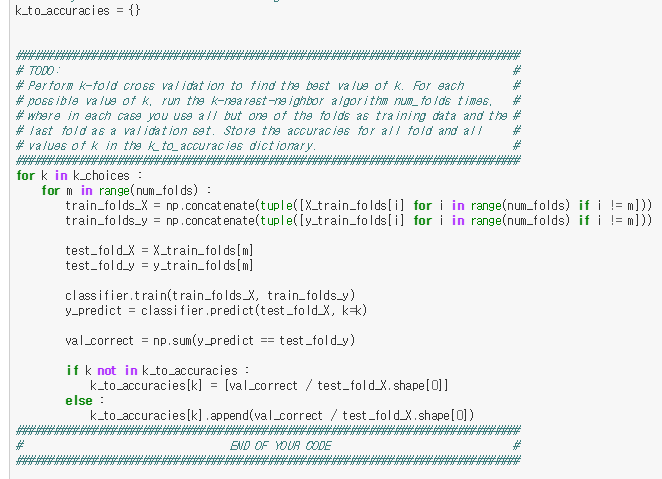


Cross-validation (좋은 k를 선정하기 위함)

<knn.ipynb>

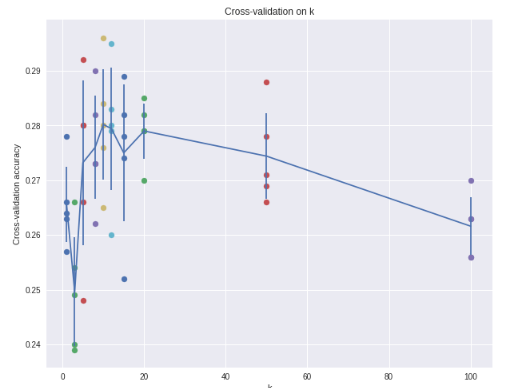


Fold의 개수는 5개라고 하고 training data를 다섯 등분해준다. Array\_split 연산을 이용하여 나눠주고 각각 X\_train\_folds, y\_train\_folds 배열에 담는다.

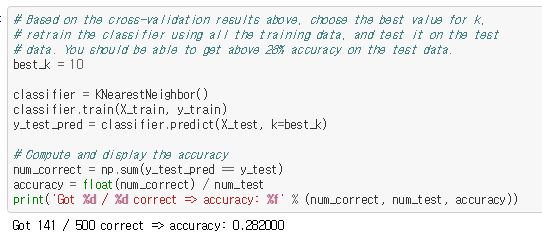


모든 k의 후보에 대해 knn 알고리즘을 적용하고 그 중 정확도가 제일 좋았던 k를 선택한다. 각 k마다 cross-validation 기법으로 훈련시키고 테스트 하기 때문에 하나의 k에 대하여 정확도 값은 5개가 나온다. (5개의 fold로 나누었으므로 각 fold가 한 번씩 testing data가 되는 것이다)

* 결과 : k=10일 때 정확도가 가장 좋아보인다.



따라서, k=10이라 하고 최종적으로 훈련시키고 정확도를 다시 계산하여보면 28.2%가 나온다.



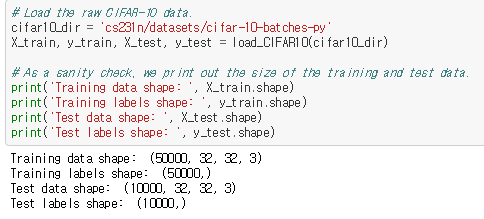
Q2: Training a Support Vector Machine

SVM classifier는 앞의 knn classifier와 다르게 데이터 중심의 접근 방법이 아닌 parametric 접근 방식이다. Training data를 이용하여 가중치 w를 최적화하여 이미지 분류를 가능하게 한다.

* 이 과제에서 구현해야 할 것

1. Loss function과 그에 대한 gradient 구하기
2. 최적의 learning rate와 regularization strength 찾기
3. SGD 방법을 이용하여 손실 함수 최적화

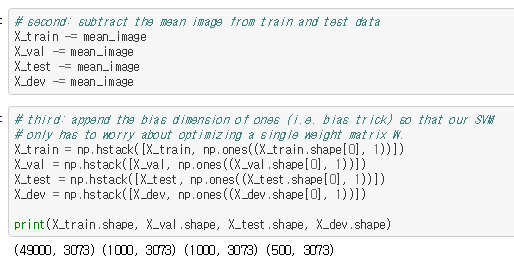
<svm.ipynb>



먼저 데이터셋(CIFAR-10)을 로드하여 training data와 그에 대한 라벨 데이터, testing data와 그에 대한 라벨 데이터를 마련한다. Training data는 50000개, testing data는 10000개에 해당한다.



데이터를 training, validation, testing set 세 가지로 나눈다. 그리고 training data 중 일부를 development set으로 가진다. 그리고 각 데이터셋들을 32\*32\*3 열로 길게 펼쳐준다.

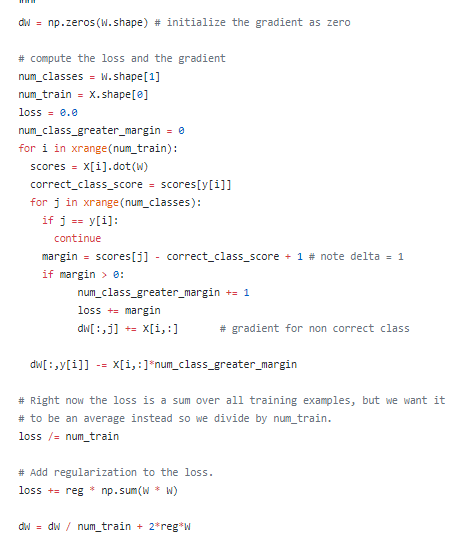


Training data를 모델에 훈련시키기 전에 데이터 전처리 과정이 필요하다. Training data와 testing data로부터 mean image를 추출하여 빼주고 bias 항을 추가해준다.

데이터 전처리가 완료되면 이제 svm loss function을 구현해야한다. 첫 번째는 loss function을 vectorized 연산을 이용하지 않고 naive하게 구현한다. 가중치 W는 작은 수로 초기화시켜두고 loss를 계산한다.

<linear\_svm.py>

Def svm\_loss\_naive

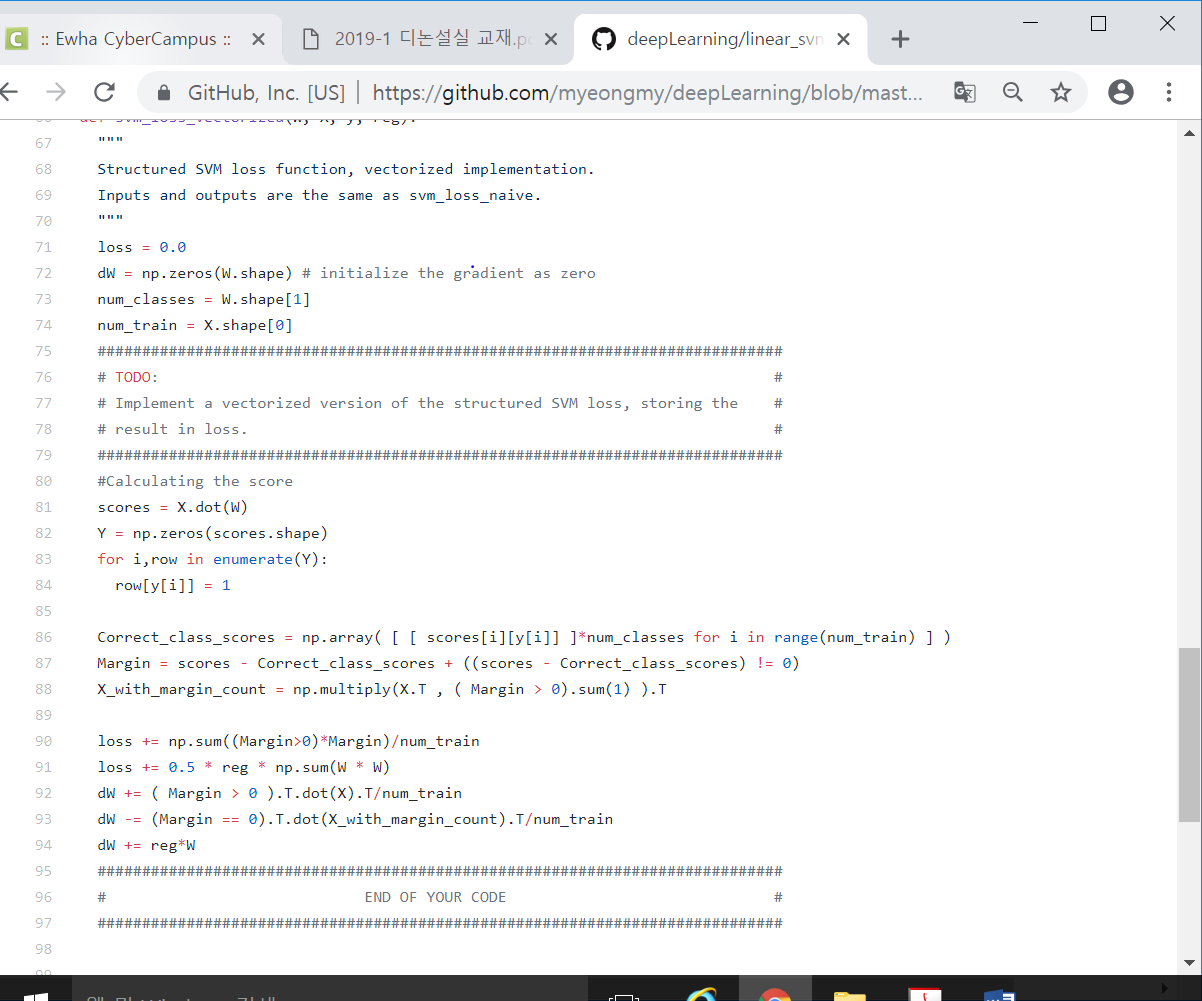


함수의 입력으로 가중치 w와 데이터와 데이터 라벨 배열을 받으면 각 이미지 데이터 별로 데이터 X와 W의 내적 곱을 구하여 각 클래스에 대한 score를 얻어낸다. 그리고 해당 이미지의 정답 클래스에 해당하는 점수를 따로 저장해둔 뒤, 모든 다른 클래스와 해당 점수를 비교하여 마진 값을 구한다. 이 때, 모든 클래스를 for문을 돌면서 정답 클래스에 해당하는 경우는 따로 마진 값을 계산하지 않고 넘어가고 그 이외의 클래스에 대해서는 마진 값을 계산한 뒤 해당 마진 값이 0보다 큰 경우에만 loss에 그 값을 더해준다. 이것이 svm loss를 구하는 방식이다. 이렇게 총 loss를 구해준 뒤 training data의 개수로 나눠주어 평균 loss를 얻고 마지막에 l2 regularization을 한다는 가정하에 regularization 항을 추가해준다.

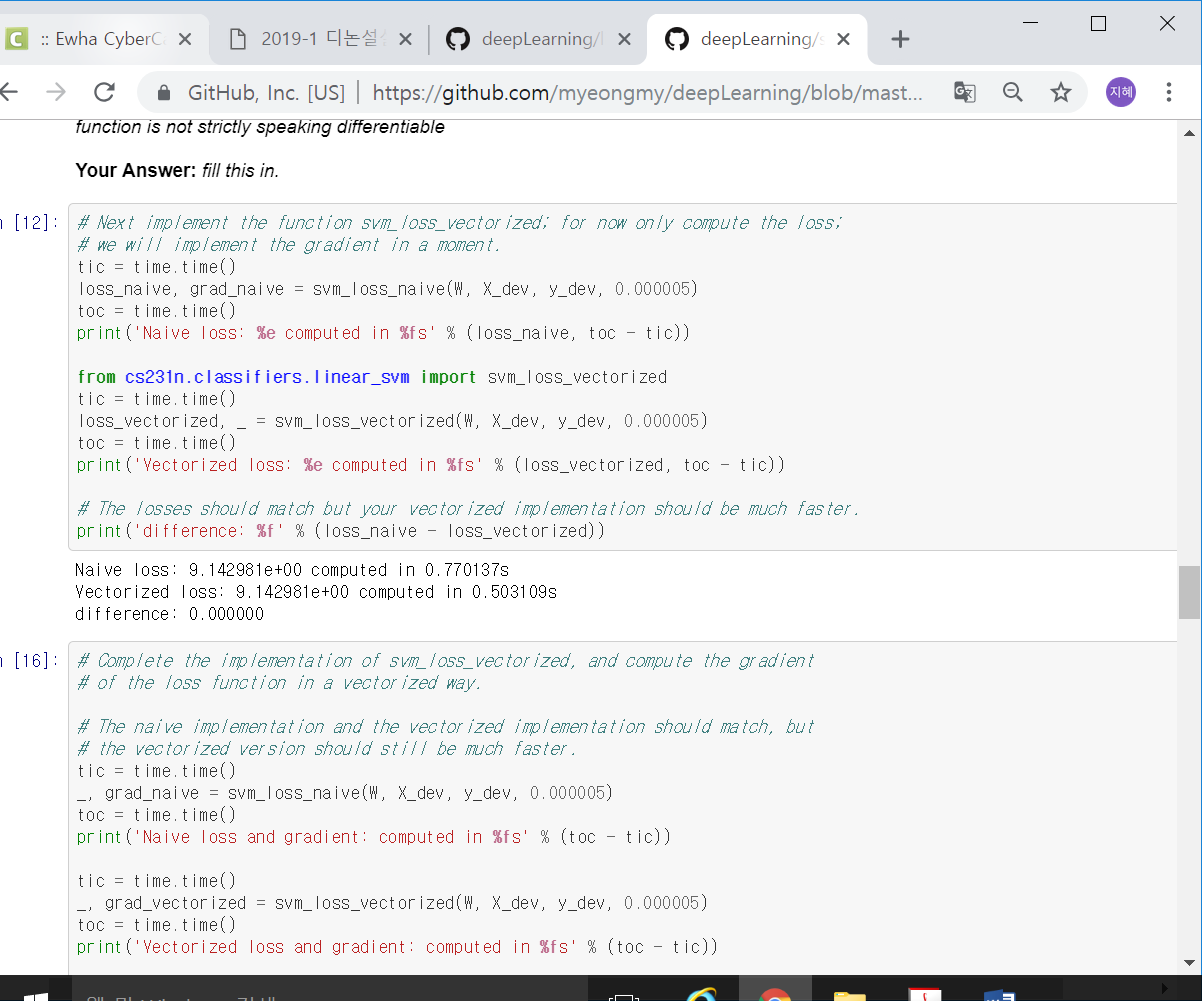
Gradient 같은 경우에는 우리가 가중치 W에 대한 gradient가 필요한 것이므로 loss 계산해주었던 수식을 토대로 차례차례 편미분을 통해 식을 얻어낼 수 있다. 앞서 loss를 구할 때 마진 값이 0보다 큰 경우에만 loss에 해당 마진 값을 더해주었으므로 dW에도 그 경우에만 X[i] 값을 더해주고 정답 클래스의 경우에는 마진 값이 0 이상이었던 클래스의 개수만큼 오히려 빼준다. 그리고 전체 loss를 training data의 개수로 나누어주었던 것처럼 dW도 똑같이 나누어주고, reg\*np.sum(W\*W)의 미분 값인 2\*reg\*W도 추가로 더해준다.

다음은 똑같은 svm loss를 vectorized하게 구현한다.

Def svm\_loss\_vectorized



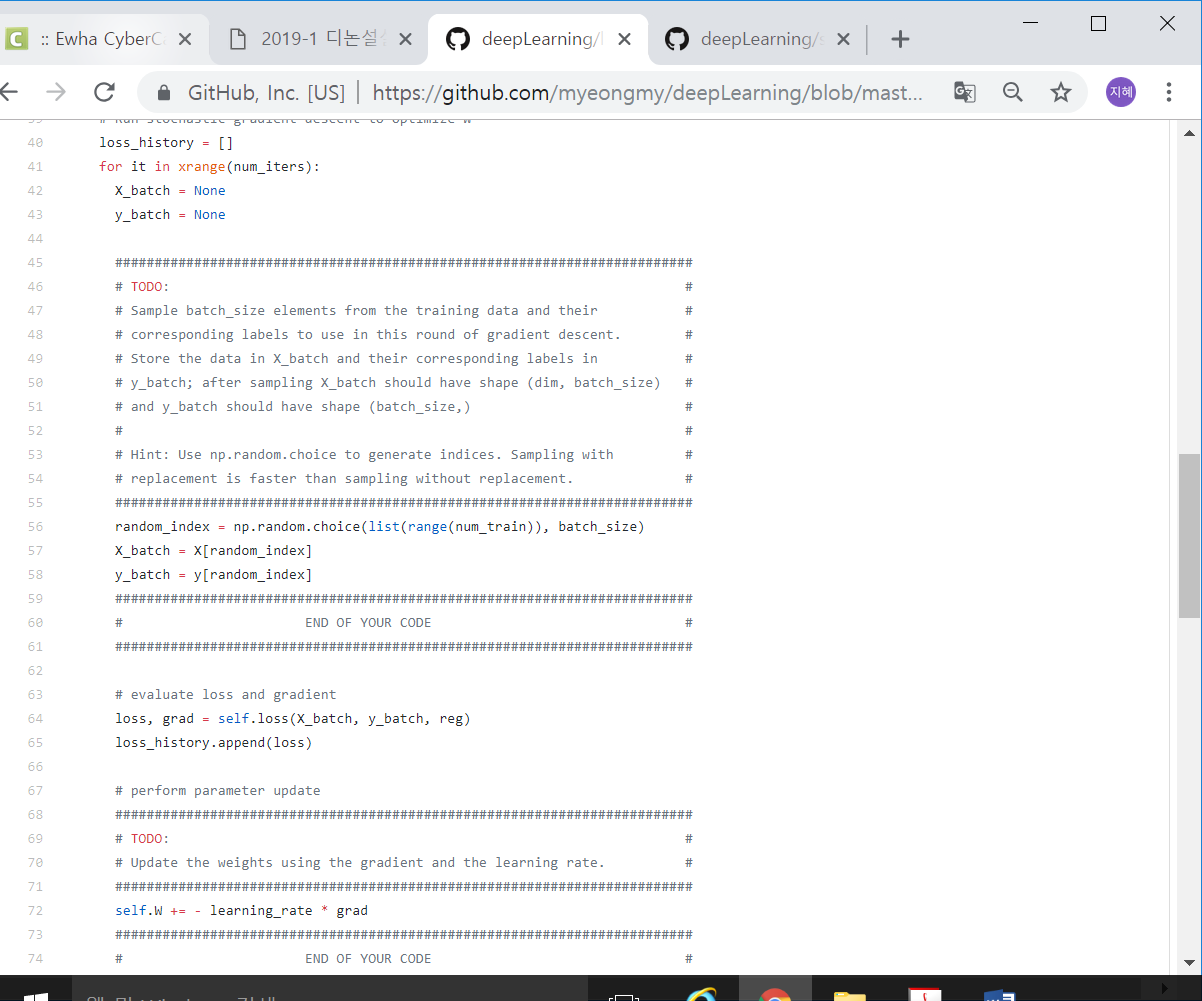
앞의 naive하게 구현한 것과 로직은 같다. 다만 앞에서는 for문을 돌며 각 testing image마다 각각 loss를 구하여 더해주었다면 이번에는 numpy에서 제공하는 vectorized 연산을 이용하여 한 번에 전체 testing images에 대한 margin 값을 구하고 loss를 구한다. 반복하지 않고 한 번에 loss를 계산하므로 이 방법이 실행 시간 면에서 더 빠르다.



실제로 naive loss와 vectorized loss는 측정된 값은 같지만 계산하기 위해 걸린 시간은 다른 것을 알 수 있다.

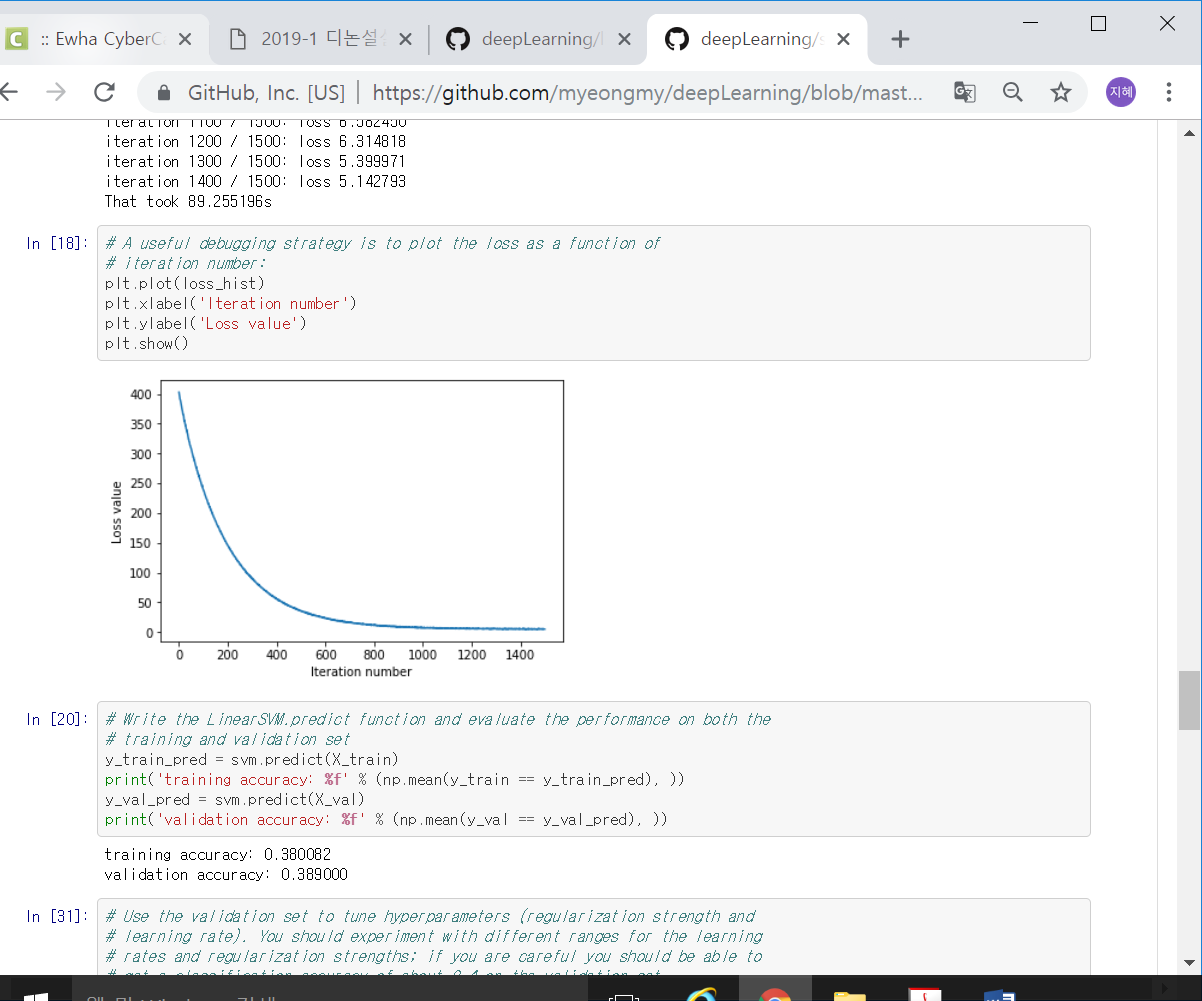
SGD(Stochastic Gradient Descent) – 가중치 W를 최적화하여 loss값을 최소화하기 위한 알고리즘

<linear\_classifier.py>

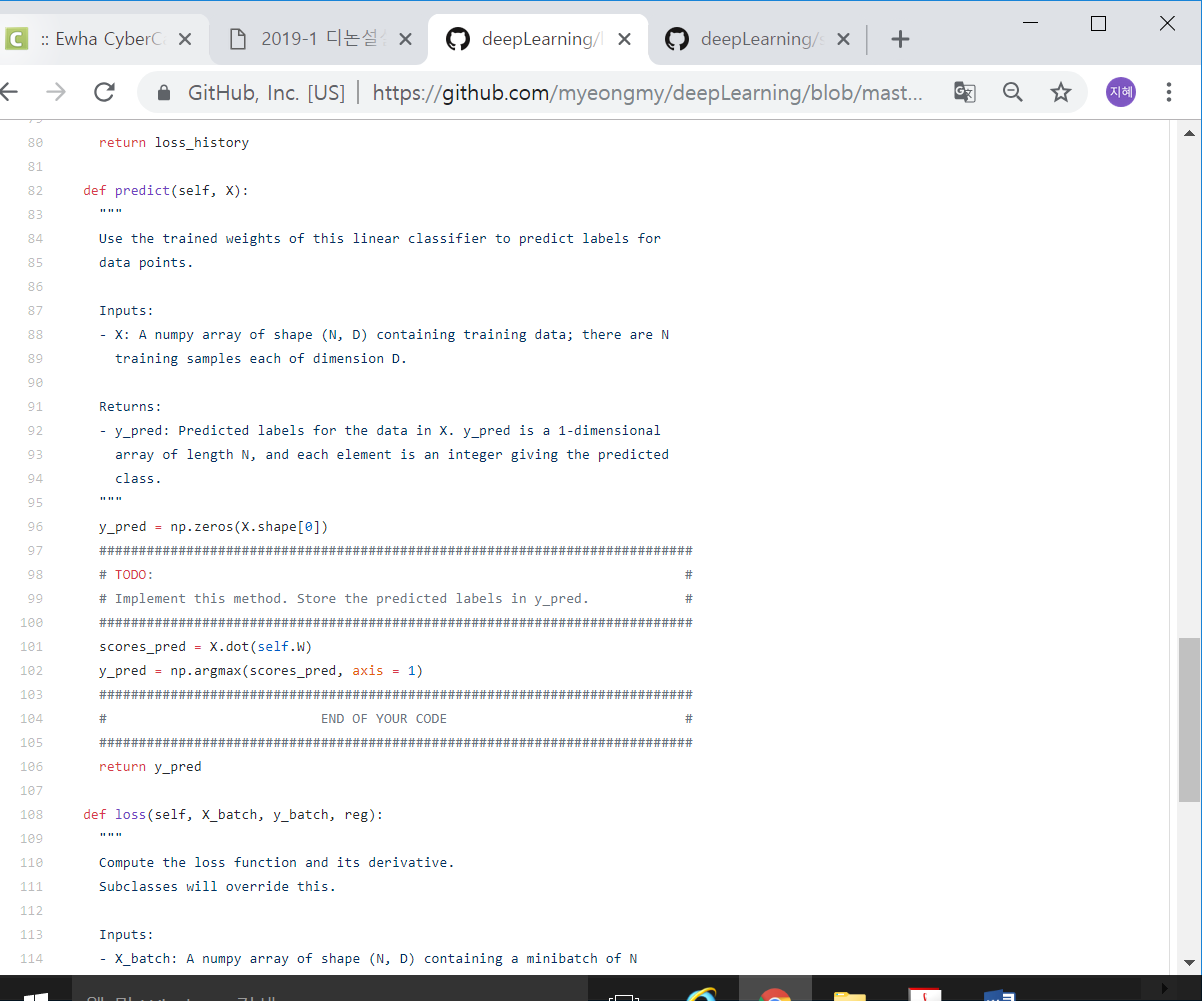


다음은 train() 함수의 일부를 캡쳐한 것이다. SGD 방법을 이용하여 훈련시키며 최적의 W를 찾아가는 과정인데 여기서는 mini-batch SGD를 이용하기로 한다. 이는 training data가 많으면 계속 반복하면서 계산을 하는데 시간이 너무 오래걸리므로 매 반복마다 랜덤으로 training data 중 일부를 뽑아서 그에 대한 loss를 계산하는 식으로 한다. 매 시행(반복)마다 랜덤하게 데이터를 추출하므로 이론적으로는 mini-batch SGD를 수행한 결과와 SGD를 수행한 결과는 크게 다르지 않다. 함수의 입력으로 training data X, 그에 대한 라벨 y, 반복 횟수 등을 받아 실행을 하게 된다. Training data에서 batch\_size만큼의 데이터를 추출하는 것은 np.random.choice() 연산을 이용한다. Training data의 인덱스 중 batch\_size만큼 랜덤하게 추출한 후 해당 인덱스를 이용하여 X\_batch, y\_batch 배열을 만든다. 이제 이 데이터를 이용해 loss와 gradient를 계산한다. 계산된 loss는 loss가 줄어드는 과정을 저장하기 위해 loss\_history 배열에 추가하고, 계산된 gradient를 이용해서는 가중치 W를 업데이트한다.

* 결과: 반복이 시행되면서 점점 loss가 줄어드는 것을 볼 수 있다.

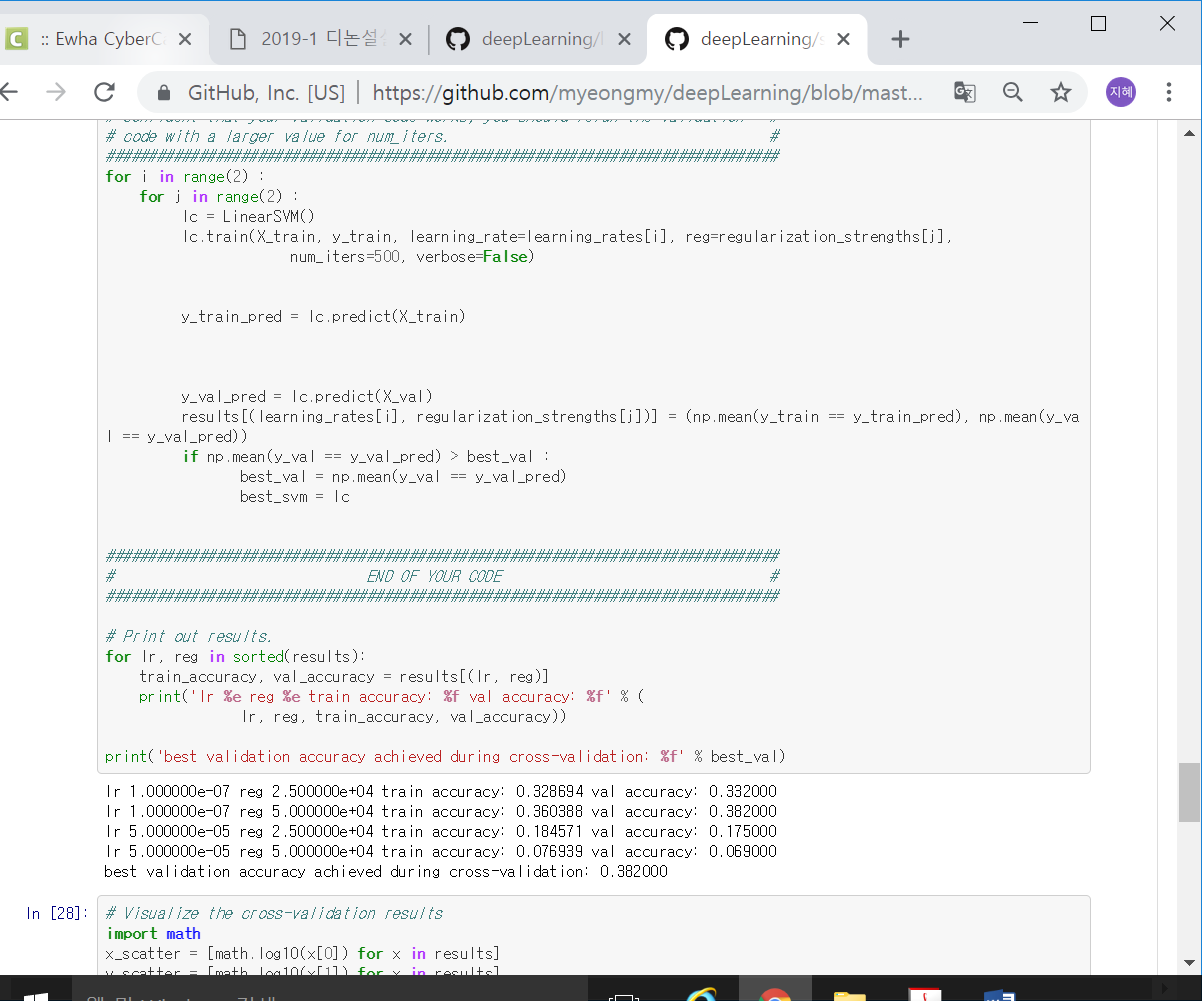


이제 훈련된 linear classifier를 가지고 새로운 testing data에 대해 라벨 예측을 시도한다.

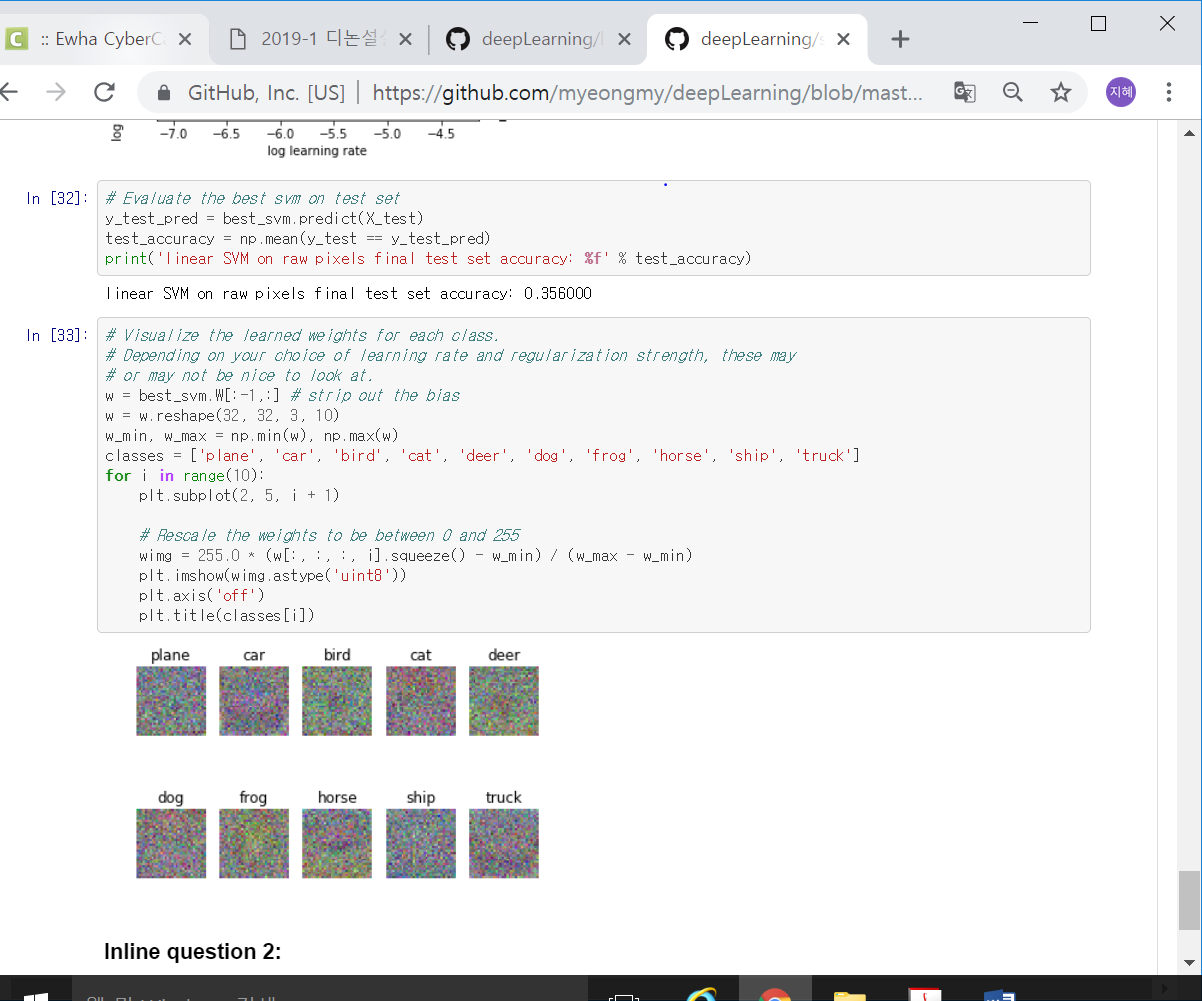


Input data와 가중치 W를 내적 곱하여 각 클래스에 대한 score값인 scores\_pred 배열을 얻는다. Scores\_pred의 shpae는 (data 개수, 클래스 수)가 될 것이다. 각 data별로 가장 높은 점수를 얻은 클래스를 구해야 하므로 np.argmax() 연산을 이용한다. 그래서 최종적으로 y\_pred(각 testing data에 대한 라벨 예측값)을 리턴한다.

하이퍼파라미터로 가중치 W가 해당 모델의 성능을 좌지우지하는 가장 중요한 요소이기는 하지만, 또 다른 하이퍼파라미터인 regularization strength, learning rate 등도 잘 결정해야 한다. 실험을 해보기 위해 몇 가지 learning rate와 regularization\_strength를 가지고 여러 번 훈련시켜보며 최적의 하이퍼파라미터를 찾아본다.



linearSVM을 하이퍼파라미터를 약간씩 조정하여 총 4번 훈련시킨다. 그리고 predict() 함수를 이용하여 각각 training data에 대한 예측 값과 validation data에 대한 예측 값을 구하여 분류기의 정확도를 구해본다. 가장 정확도가 높았던 모델과 정확도 값을 마지막에 출력하기 위하여 best\_val과 best\_svm에 저장한다.



위에서 가장 성능이 좋았던 모델을 사용하여 최종적으로 testing set을 가지고 예측을 해본 결과 35.6%의 정확도를 보였다.

Q3: Implement a Softmax classifier

Softmax classifier는 앞에서 구현한 svm classifier와 같은 linear classifier이다. 다만 다른 점은 손실 함수를 구하는 방법이 다르다는 것이다. Svm classifier는 정답 클래스의 점수 값과 얼마나 큰 차이를 보이느냐에 초점을 둔 반면, softmax classifier는 전체 클래스에 대한 정답 클래스의 확률 값을 계산한다.

* 이 과제에서 구현해야 할 것

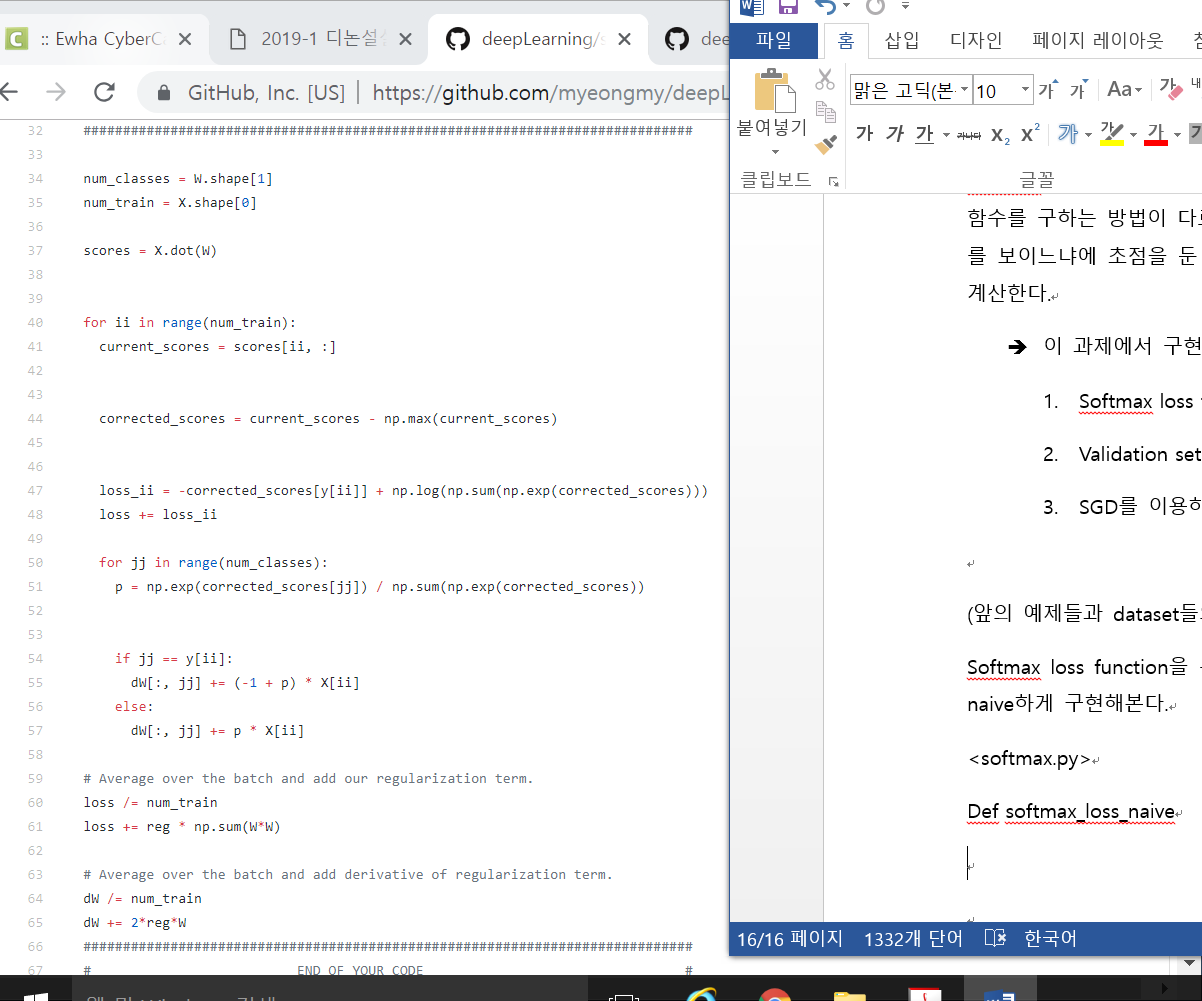
1. Softmax loss function을 구현하고 그에 대한 gradient 구하기
2. Validation set을 이용하여 hyperparameter 조정하기
3. SGD를 이용하여 loss function 최적화

(앞의 예제들과 dataset들의 shape와 값이 동일하므로 데이터 로드하는 부분은 생략한다.)

Softmax loss function을 두 가지 방법으로 구현한다. 첫 번째는 vectorized 연산을 사용하지 않고 naive하게 구현해본다.

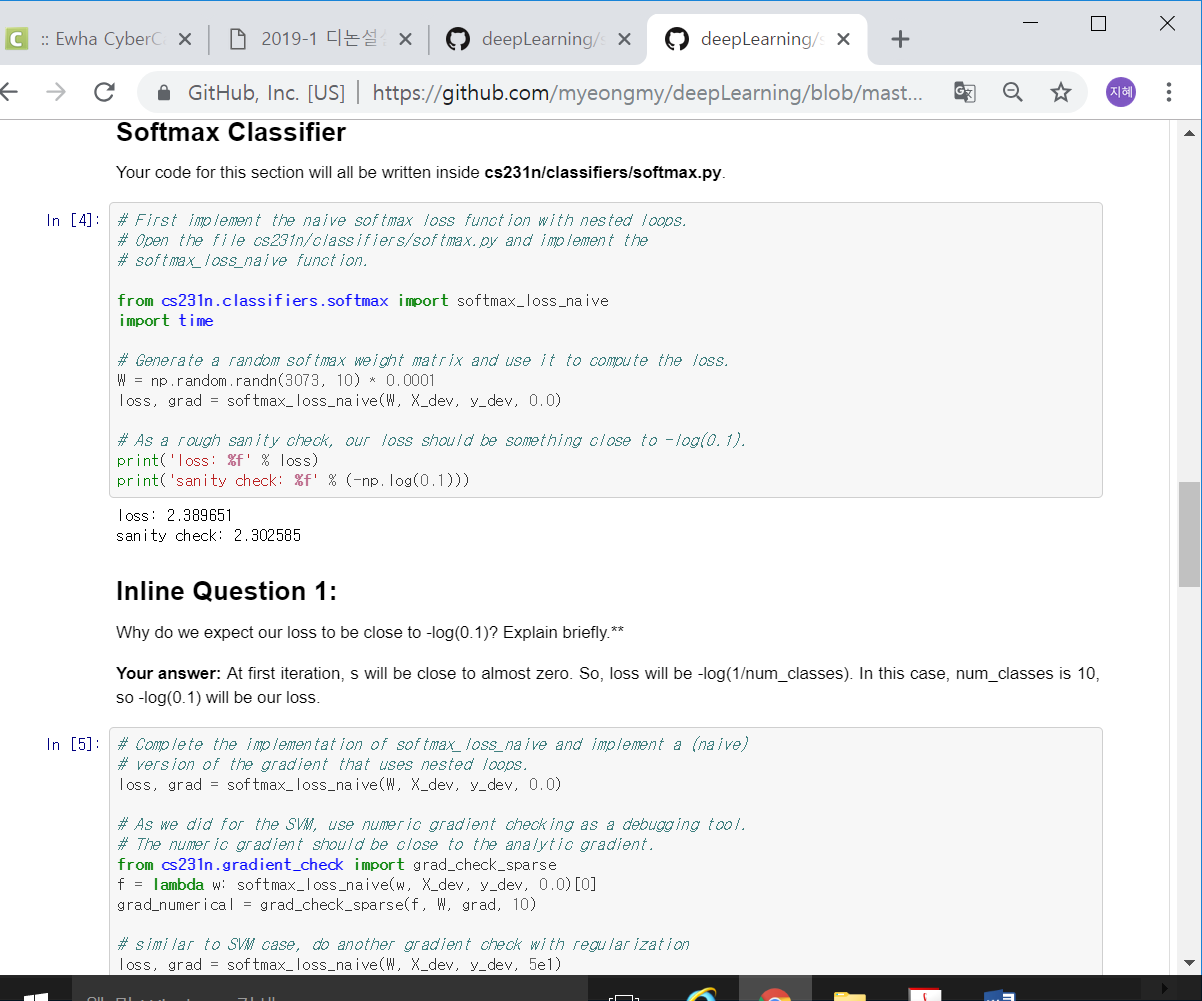
<softmax.py>

Def softmax\_loss\_naive



Softmax의 수식에 따라 loss를 계산하는데 testing data 하나하나 for문을 돌며 loss를 계산하여 더해간다. Input testing data들에 가중치 W를 내적 곱하여 나온 점수에 ii번째 행을 택하여 각 이미지에 대한 현재 current\_scores를 가져온다. 이 배열을 이용하여 loss\_ii를 수식에 따라 구한 뒤 전체 loss에 더하여 준다. Gradient 같은 경우에는 정답 클래스에 대한 gradient와 그렇지 않은 나머지 클래스들에 대한 gradient를 구하는 방식이 약간 다르다. Loss를 Wy에 대하여 편미분 하면 (py-1)x의 형태가 되고, Wk(k != y)에 대하여 편미분하면 pkx의 형태가 된다. (편미분 되는 부분은 교수님의 강의 프린트를 참고하였으나 원리는 이해하기 어려웠습니다..)

마지막으로 우리는 전체 loss를 data의 개수로 나누어주어 평균 loss를 구하고 overfitting을 막기 위한 regularization 항도 추가적으로 더해준다. Loss에 해준 연산에 대하여 dW에도 똑같이 반영한다.



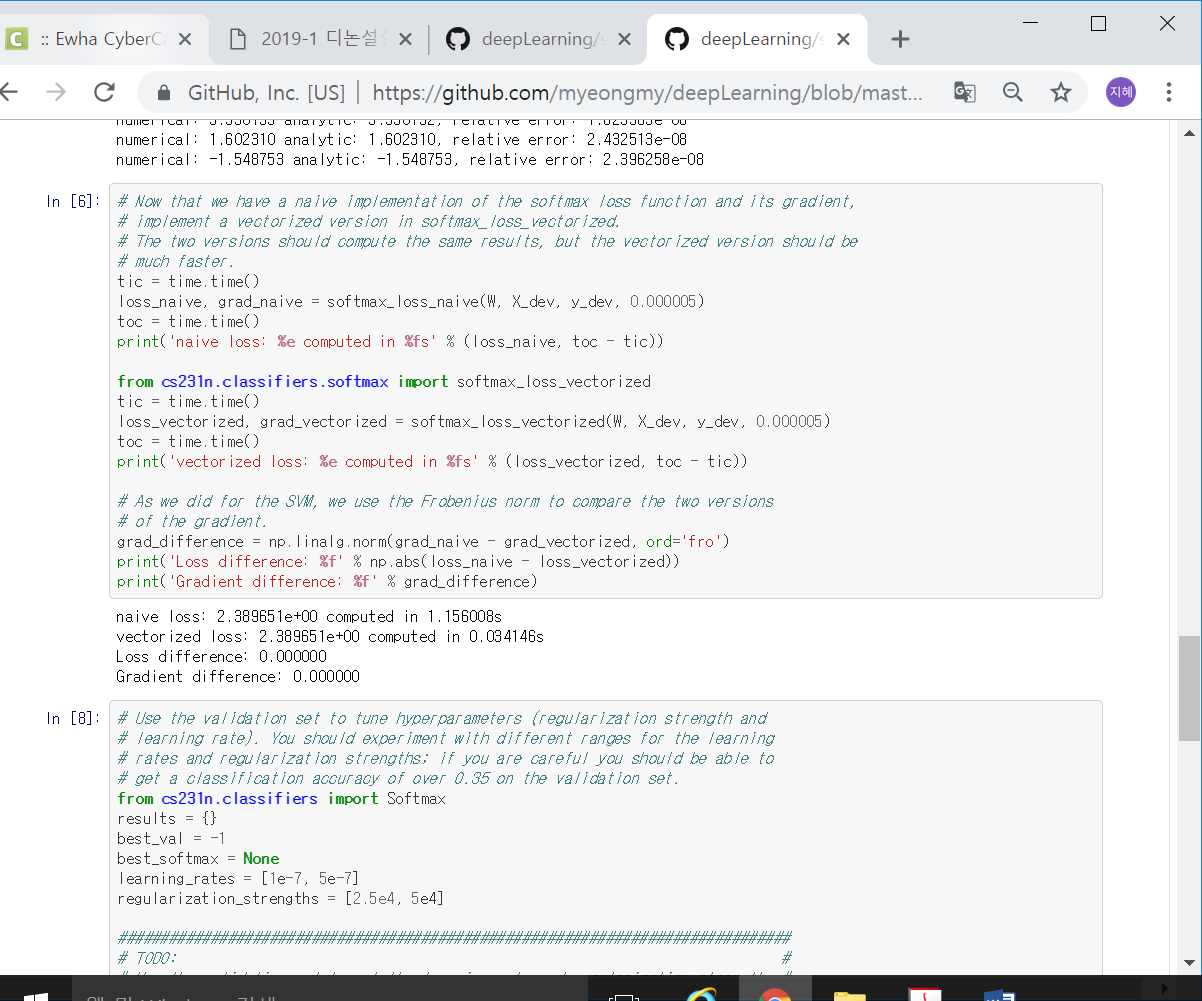
가중치를 아주 작은 수로 초기화한 후 위에서 구현한 함수로 loss를 계산하여보면 –log(0.1)에 가까운 수가 나온다. 왜냐하면 첫 번째 iteration에서는 W가 아주 작은 수이므로 W와 data를 내적 곱한 결과 score가 거의 모두 0에 가까운 수가 출력된다. 따라서 이 예제에서는 클래스의 수가 10개이니 –log(1/10) loss가 나오는 것이다. 처음 iteration 때 이 값이 안 나오면 모델에 문제가 있는 것이니 sanity check용도로 돌려보는 것이 좋다.

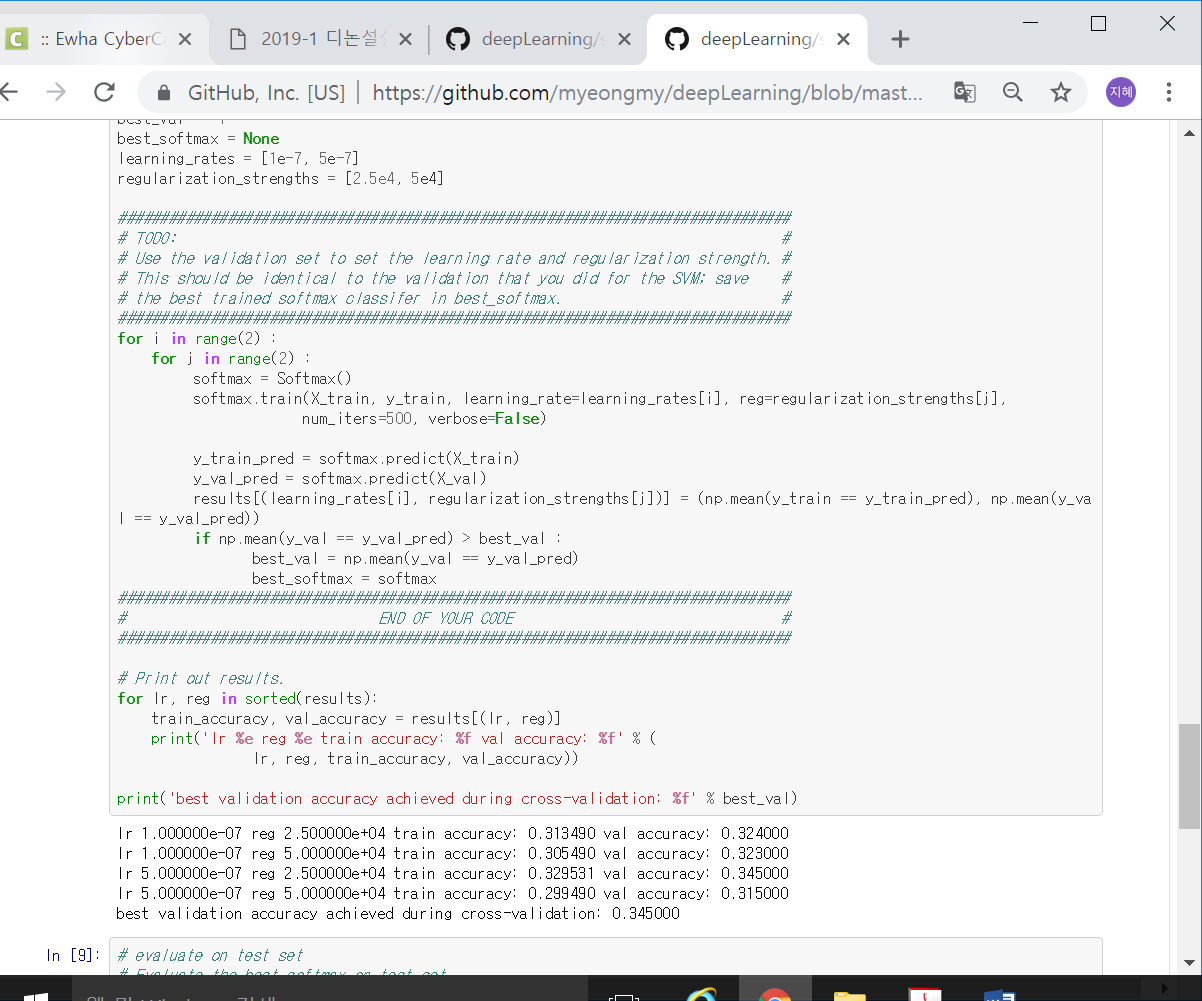
이번에는 위에서 구현한 softmax loss function을 vectorized하게 구현한다.

Def softmax\_loss\_vectorized

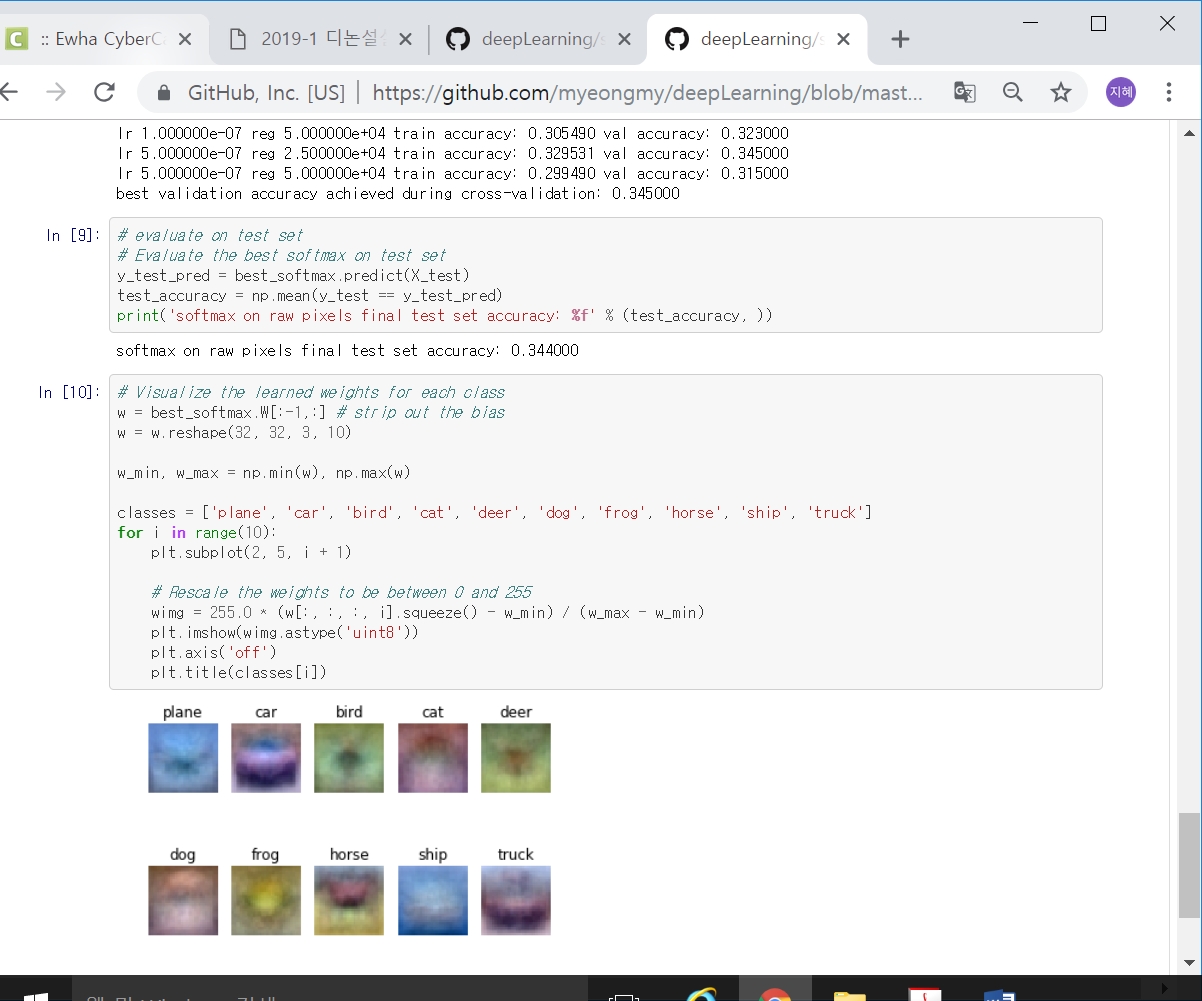


위와 동일한 방식으로 작성했는데 이번에는 for문을 돌리지 않고 numpy의 다양한 vectorized 연산을 이용하여 구현하였다. 위와 달리 추가된 “axis=1”부분은 각 행 별로 최대값, 합 등을 구하기 위함이다.



naive하게 구현한 loss나 vectorized하게 구현한 loss가 결과 값에 차이는 없지만 vectorized loss가 실행 속도가 훨씬 빠른 것을 알 수 있다

Q2의 svm classifier와 마찬가지로 learning rate, regularization strength 등 하이퍼파라미터들을 조정하기 위해 여러 가지 경우의 수를 두고 여러 번 모델을 훈련시킨다. (이 예제에서는 4번 훈련시킨다.) 방법은 svm loss 때 했던 방식과 같고 결과를 보면 lr이 5.0e-7, reg이 2.5e+4일 때 가장 정확도가 좋은 것을 알 수 있다.



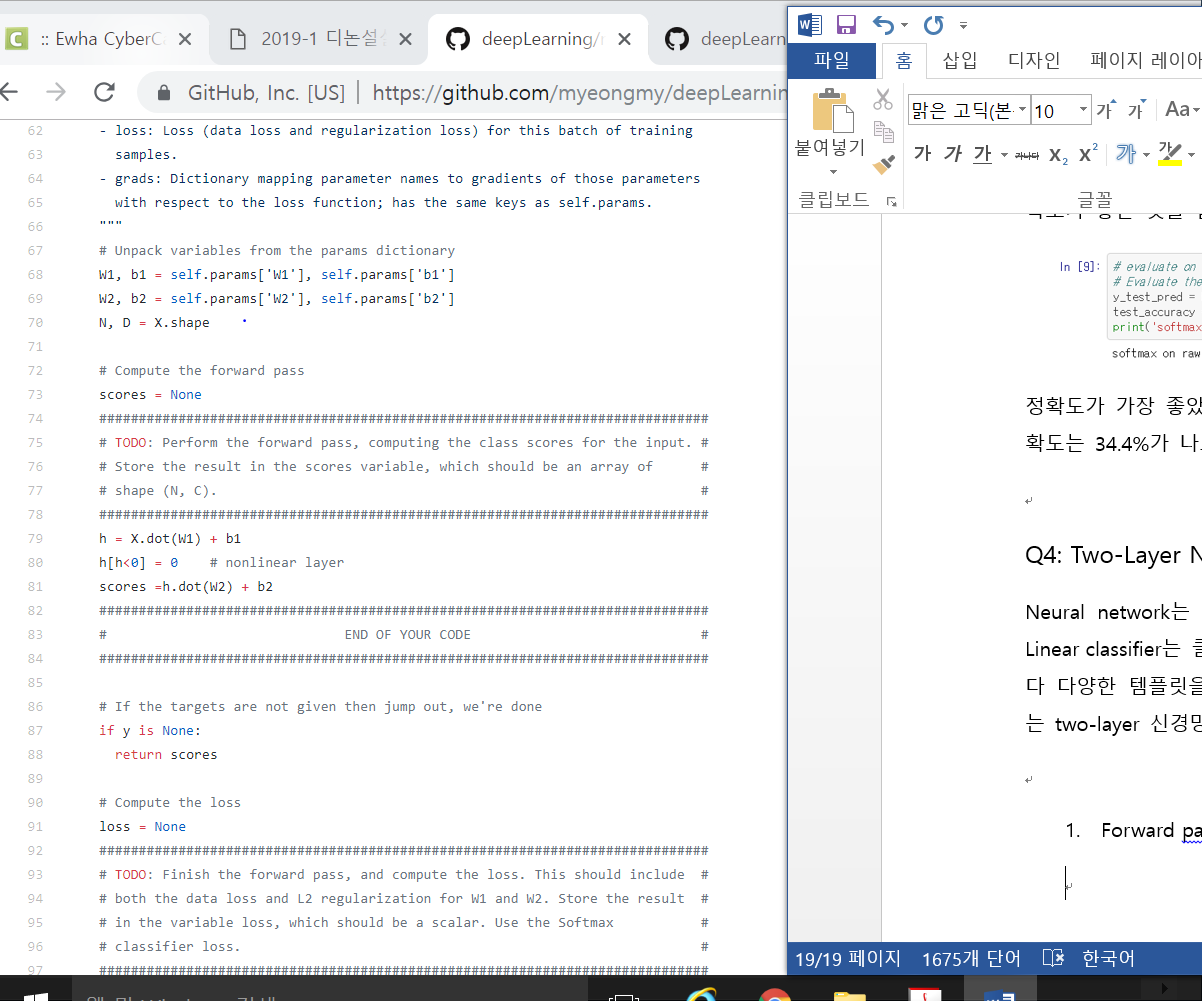
정확도가 가장 좋았던 모델을 이용하여 최종적으로 test set을 가지고 돌려본다. Test set에서의 정확도는 34.4%가 나오는 것을 알 수 있다.

Q4: Two-Layer Neural Network

Neural network는 linear classifier의 선형 레이어를 여러 번 통과시켜 성능을 더 높일 수 있다. Linear classifier는 클래스 당 하나의 템플릿만을 두고 이미지를 판별하였다면, neural network는 보다 다양한 템플릿을 두게 되어 이미지의 다양한 형태를 고려하여 성능이 향상된다. 이 과제에서는 two-layer 신경망을 구현한다.

1. Forward pass : loss 계산

<neural\_net.py>

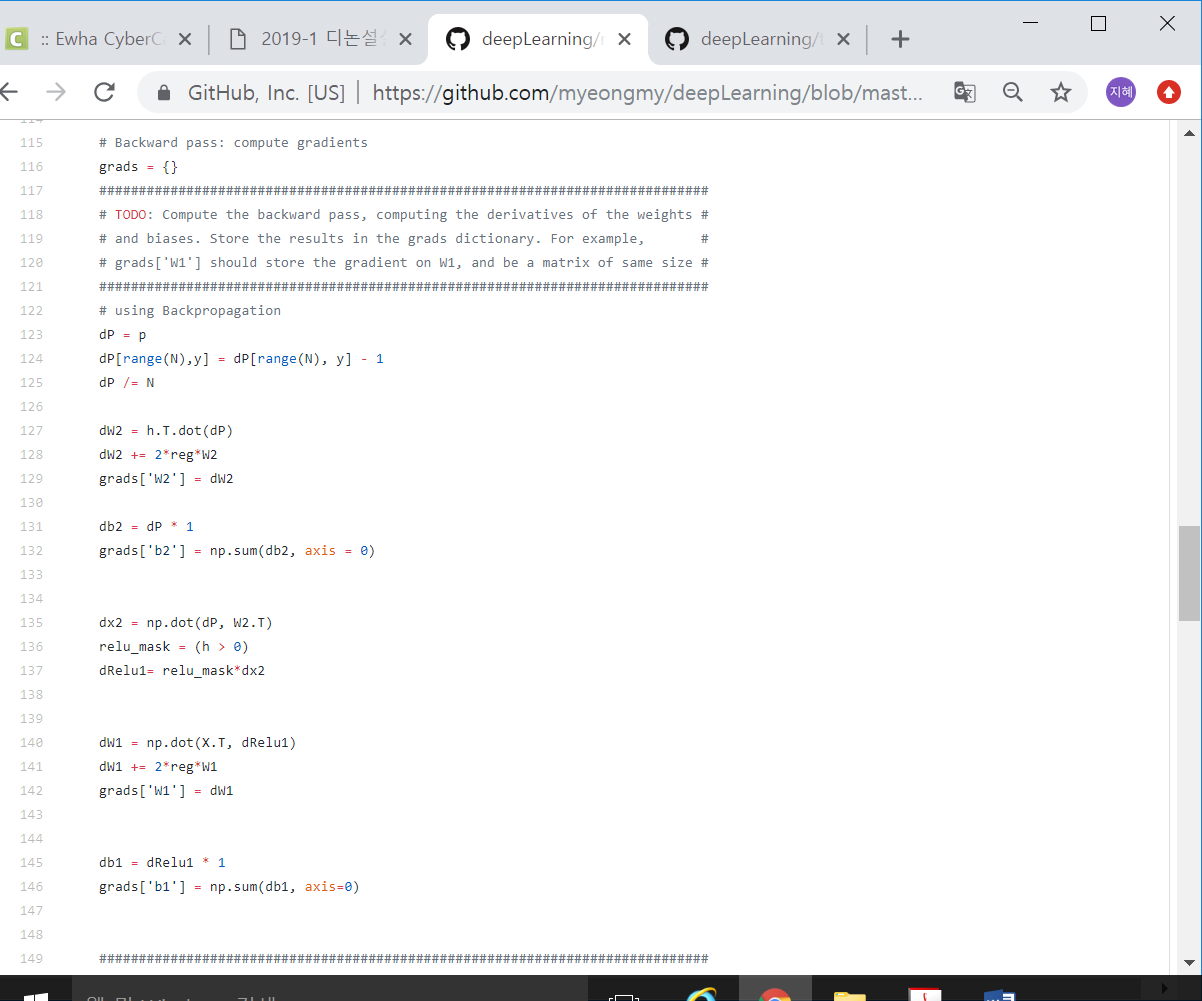


다음은 loss() 함수의 일부를 캡쳐한 것이다. 입력 값으로 training data를 받으면 먼저 첫 번째 선형레이어(W1\*X+b1)를 통과하여 중간 점수를 얻고 여기에 ReLu 활성화 함수를 적용하여 0보다 작은 점수를 가진 부분은 0으로 바꾸어준다. 이렇게 비 선형 레이어를 거친 h는 다시 한 번 2차 선형 레이어(W2\*X+b2)를 통과하여 최종 점수를 얻는다.



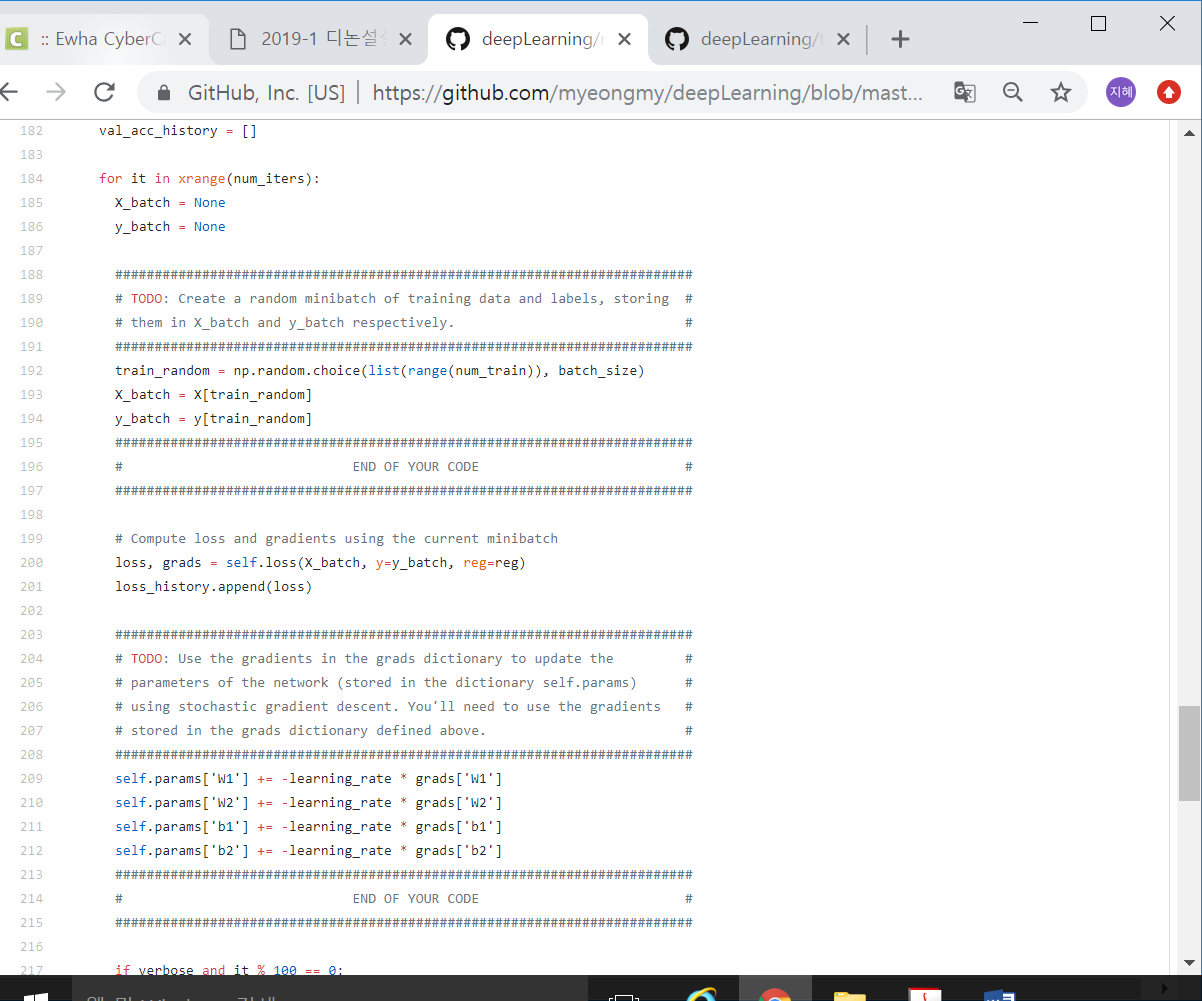
위에서 구한 score를 이용하여 loss를 구한다. Softmax loss function의 방식으로 vectorized하게 구현하였고, regularization은 l2 regularization을 사용하여 더해주었다. 위의 코드는 loss를 구하는 과정이라 앞의 Q3에서의 loss function 계산부분과 일치한다.

1. Backward pass : gradient 계산



Gradient는 차례차례 뒤에서부터 backpropagation을 진행하여 각각 W2, b2, W1, b1에 대한 gradient는 얻고 그것을 grads라는 dictionary에 저장하였다. (수식은 교수님 강의자료 Lecture 9 슬라이드 55에 있는 2-layer neural net(Softmax classifier)의 편미분 수식을 이용하였습니다.)

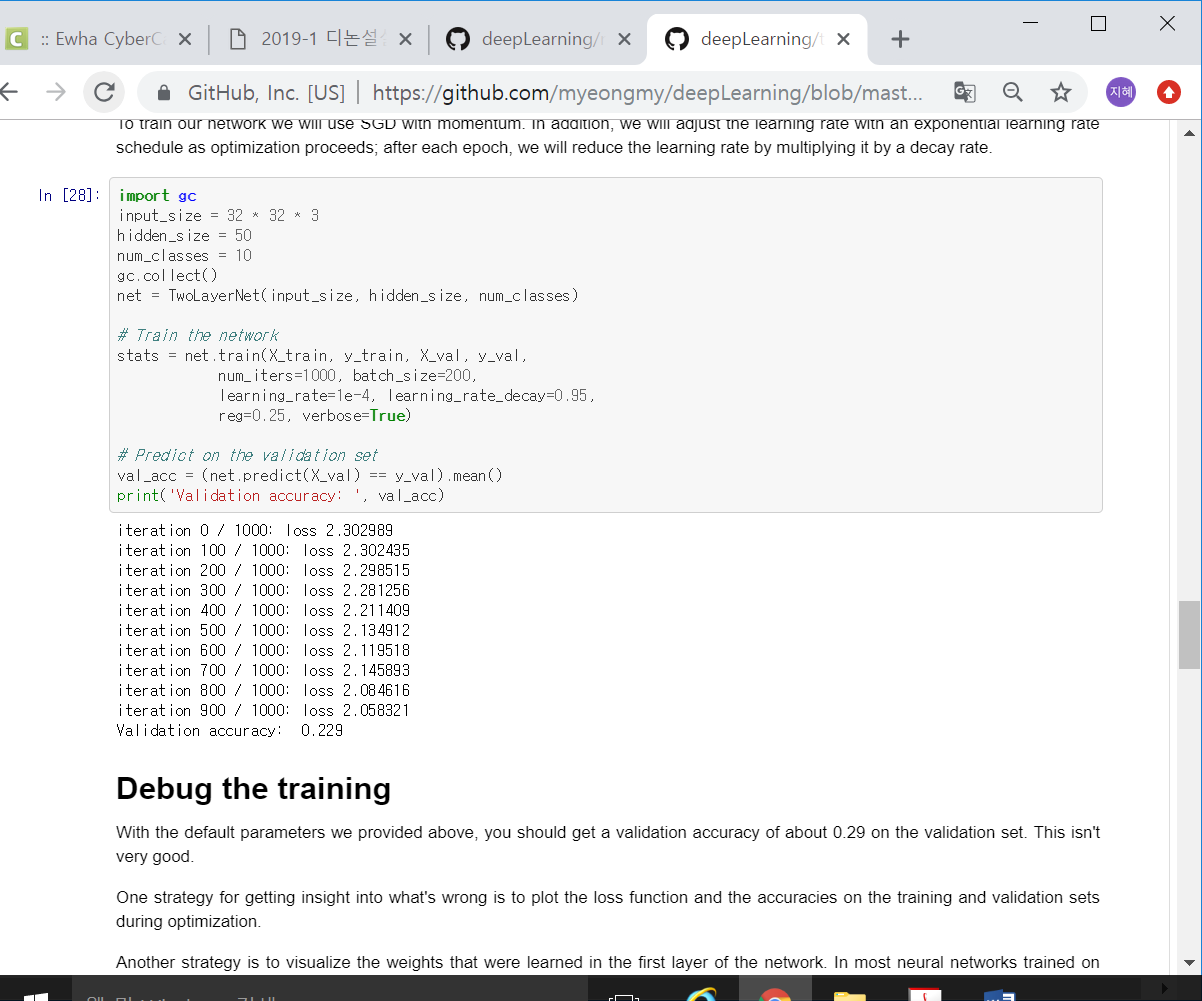
Loss와 gradient를 구하는 부분에 대한 구현을 마쳤으니 이제 모델을 train하는 부분을 살펴본다.



여기서도 앞선 문제의 softmax classifier를 훈련시킬 때와 마찬가지로 mini-batch SGD 최적화 알고리즘을 사용한다. 그러므로 매 iteration마다 training data로부터 임의로 batch\_size만큼의 데이터를 뽑아 X\_batch와 그에 대한 라벨을 y\_batch 변수에 저장한다. 그리고 그 X\_batch를 가지고 loss와 gradient를 구하여 가중치 W1, W2와 bias 항 b1, b2를 업데이트한다.

이렇게 구현한 two-layer 신경망 모델을 직접 CIFAR-10의 dataset을 사용하여 훈련시켜보도록 한다.

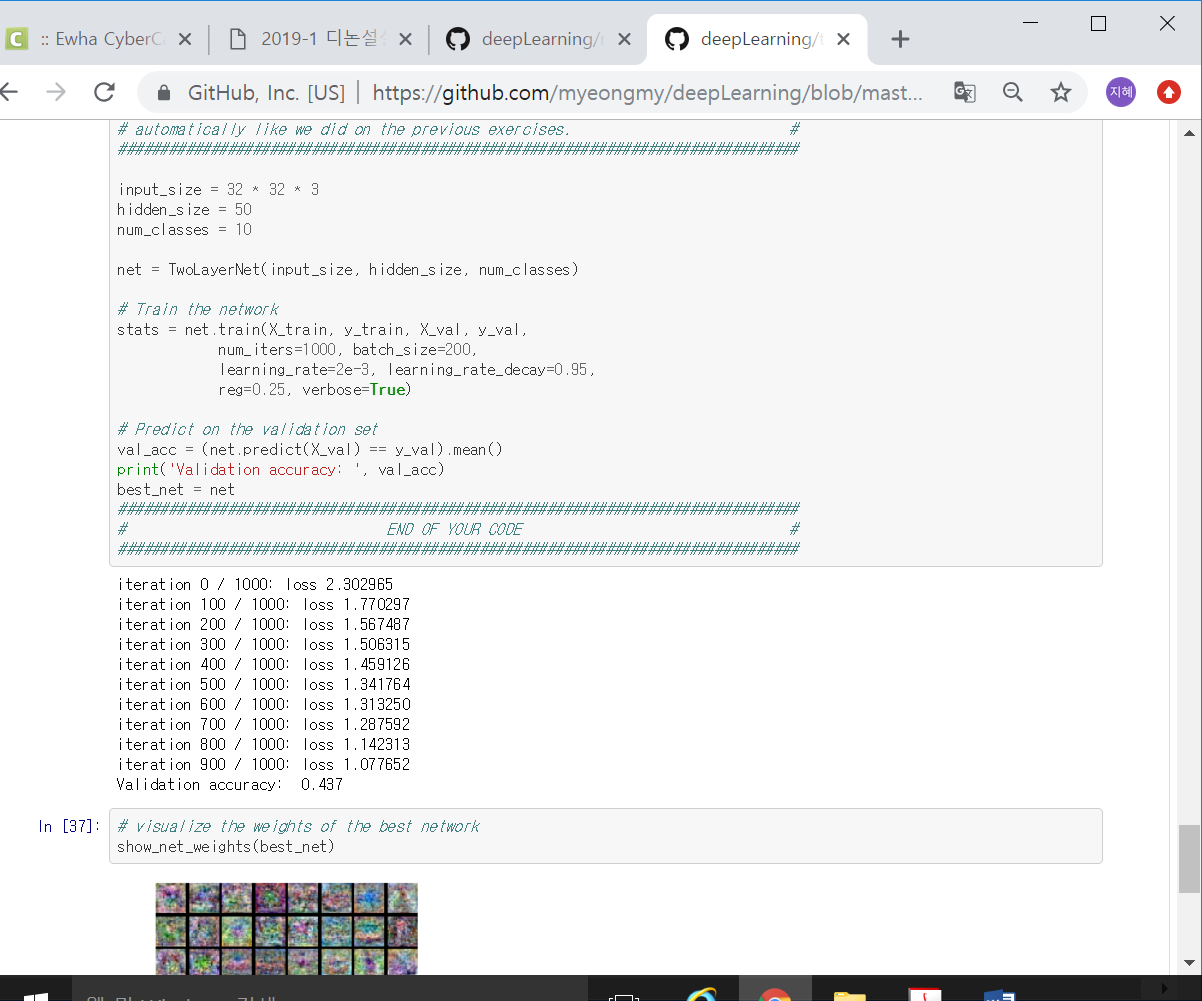
* 결과: iteration이 진행될수록 loss가 점점 최소화 되는 것을 볼 수 있다. 정확도는 22.9%



1. Tune your hyperparameters

위에서 훈련시킨 모델은 loss가 점점 줄고 있는 방향이긴 하지만 줄어드는 속도가 느리고 선형적이다. 이는 learning rate가 너무 작아서 그런 것 같다고 판단하여 learning rate를 2e-3으로 증가시키고, training data에 대한 overfitting을 막기 위해 reg 항을 좀 더 크게 주고 다시 훈련시켜보았다.

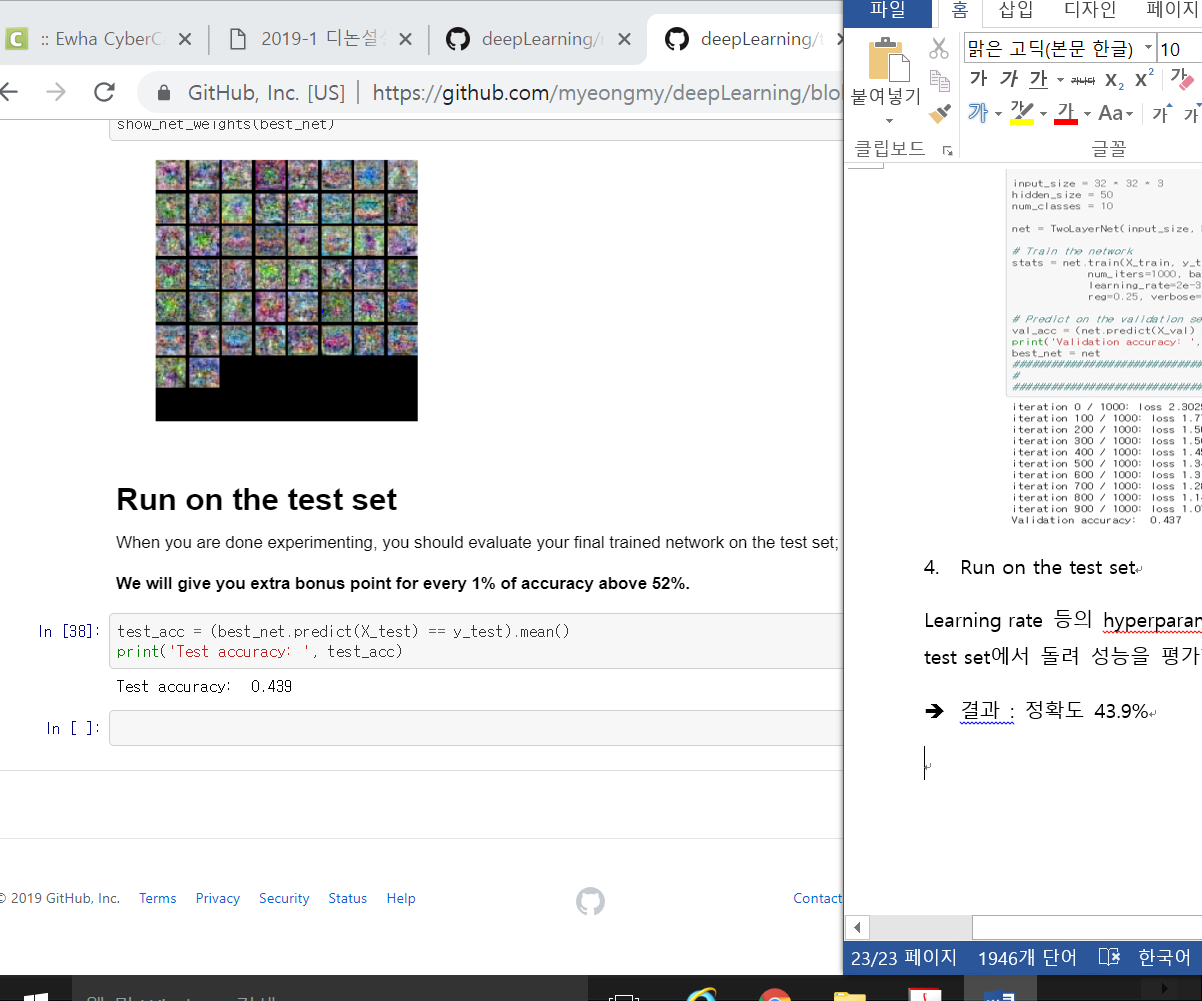
* 결과 : 정확도가 향상되었다. (43.7%)



1. Run on the test set

Learning rate 등의 hyperparameter들을 조정하여 정확도를 높인 모델을 이용하여 최종적으로 test set에서 돌려 성능을 평가한다.

* 결과 : 정확도 43.9%

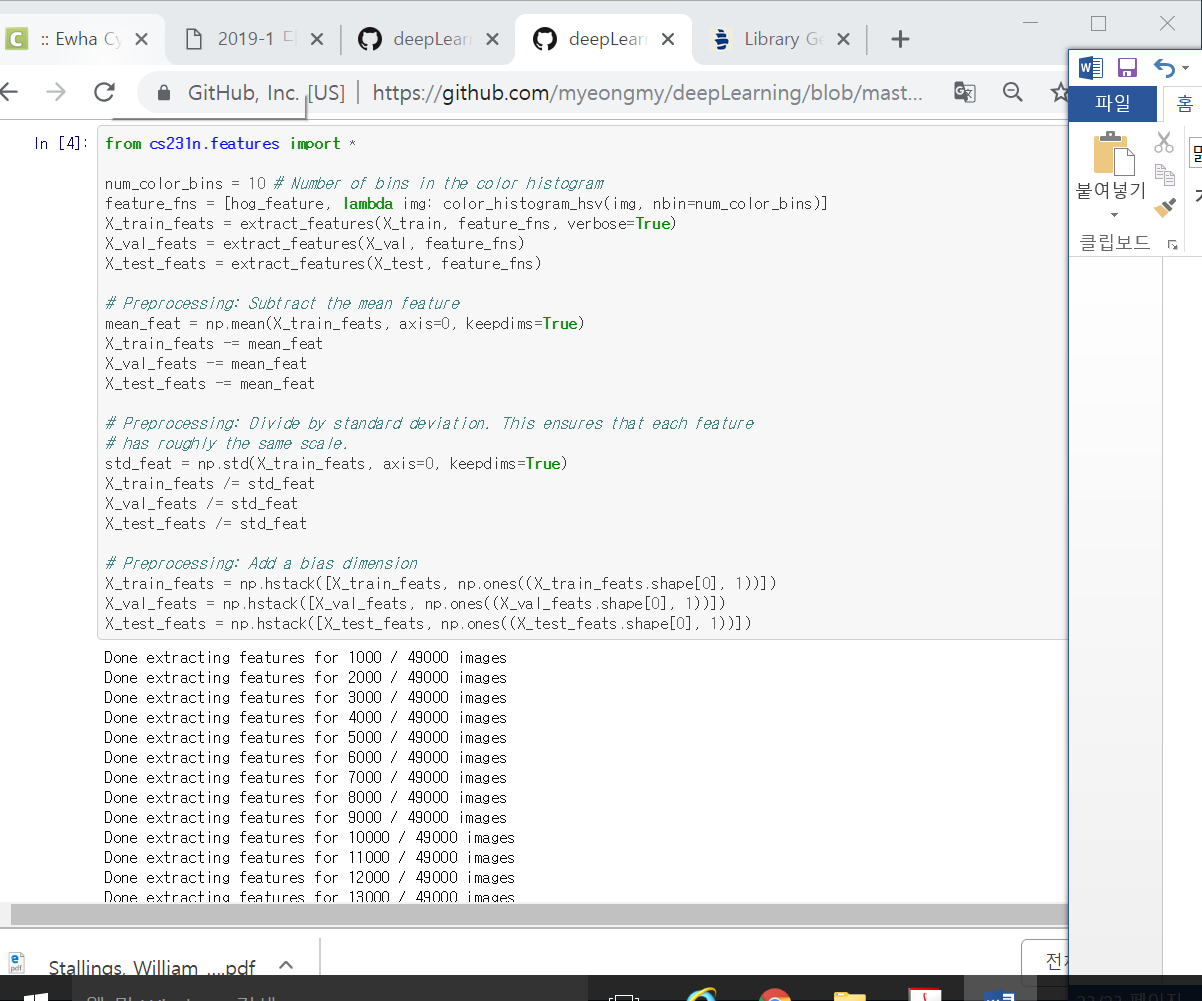


Q5: Higher Level Representations: Image Features

앞선 문제들을 통해 image classification 작업을 linear classifier, neural network 모델을 이용하여 수행했다. 이는 입력 이미지의 픽셀 값들 하나하나에 모델을 적용했다면 이번 과제에서는 픽셀 값으로부터 계산된 특징들에 모델을 적용해본다.

1. Extract Features

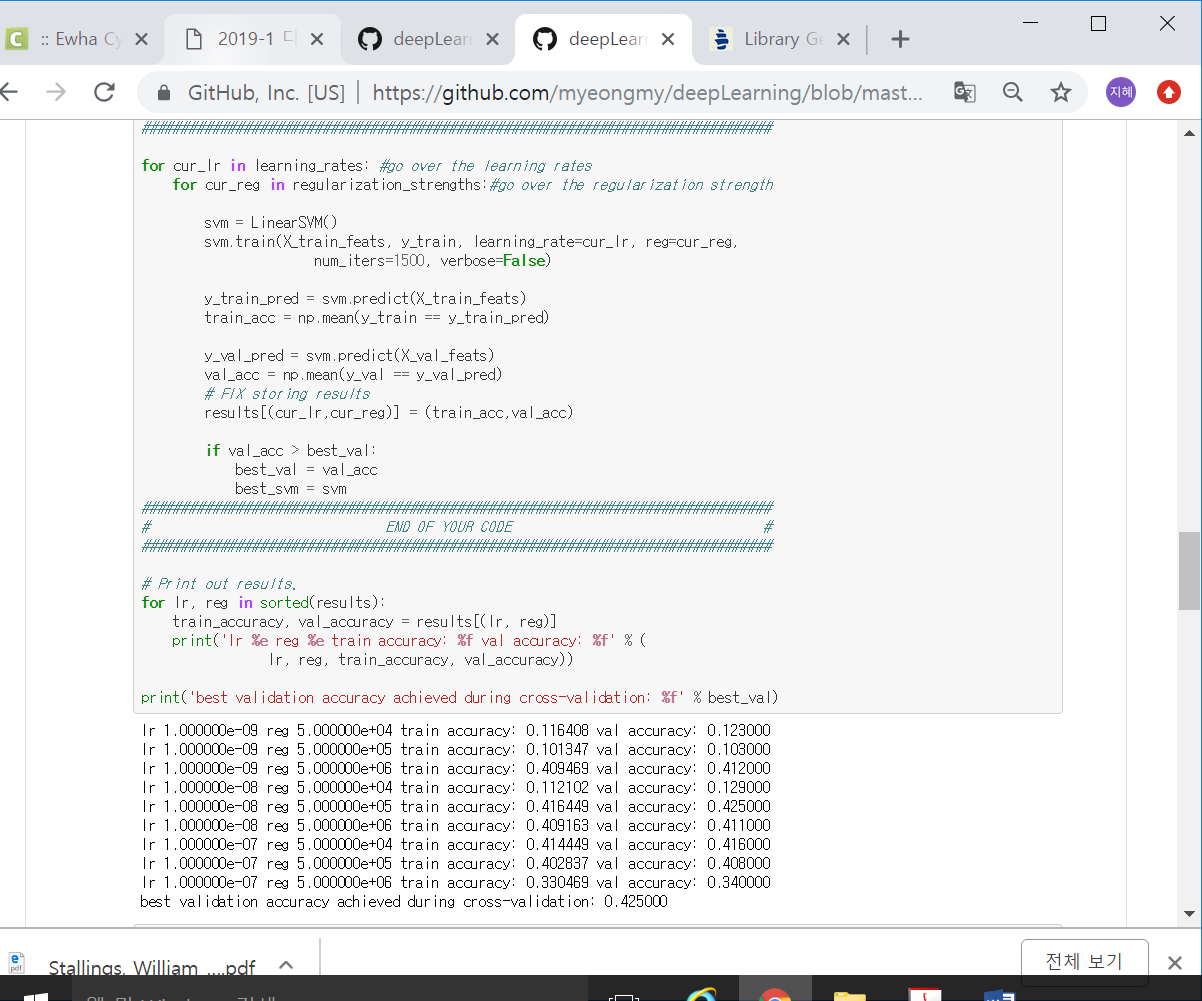
각 이미지로부터 HOG와 color histogram을 계산한다. HOG는 이미지의 컬러 정보는 무시하고 이미지의 질감을 추출해주고, color histogram은 반대로 질감을 무시하고 색깔에 대한 정보를 나타낸다.



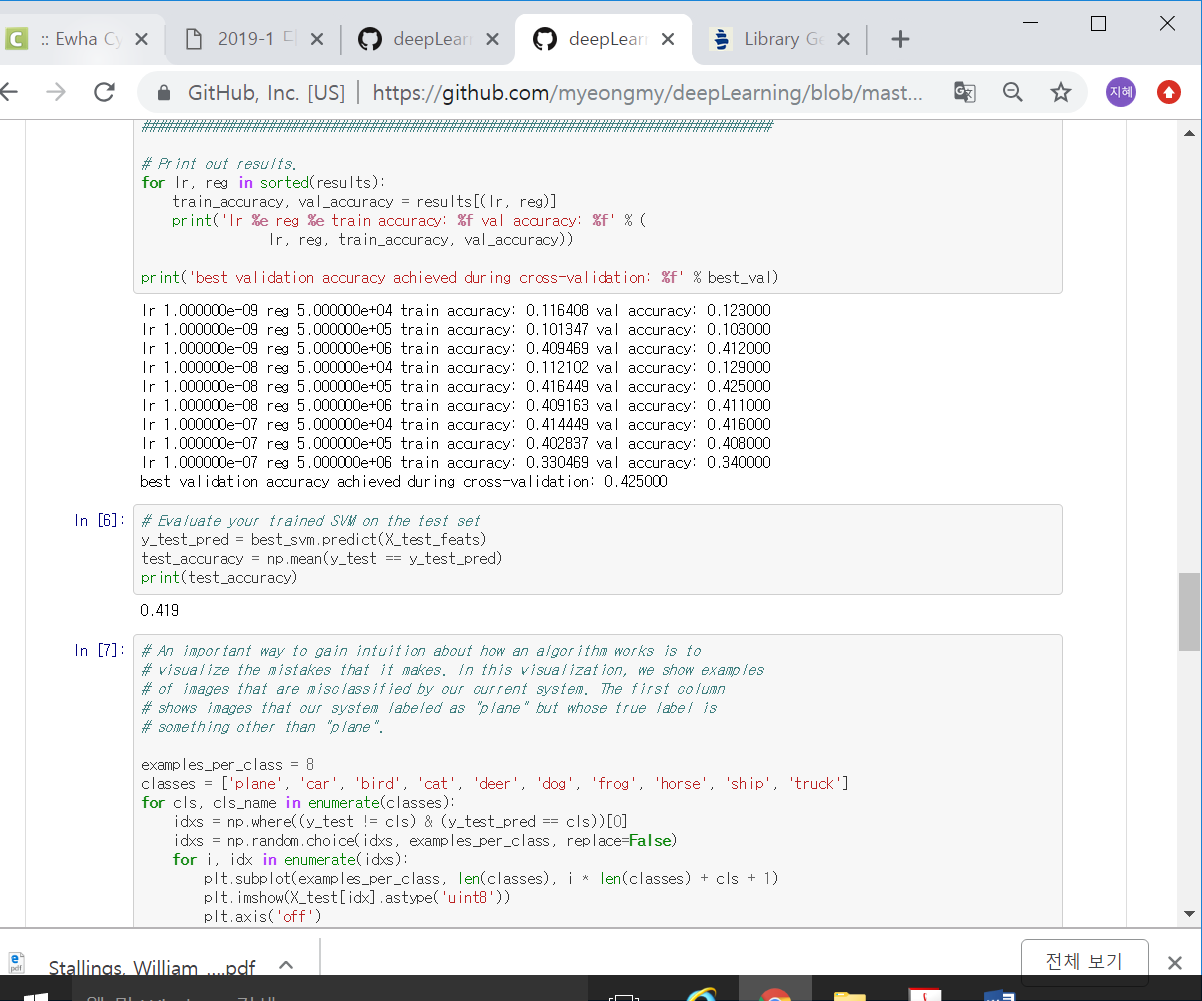
그래서 hog\_feature와 color\_histogram\_hsv 함수는 각각 질감에 대한 특징 벡터, 색깔에 대한 특징 벡터를 리턴한다. 따라서 우리는 extract\_features 함수를 통해 training data, validation set data, test set data에 대한 특징들을 추출한다. 특징을 추출한 뒤 몇 가지 전처리 과정을 해준다. Mean feature를 구해서 빼주고, 표준 편차로 나누어주고, bias 항을 추가해준다.

1. Train SVM on features

위에서 추출한 특징들을 가지고 SVM classifier를 훈련시킨다.



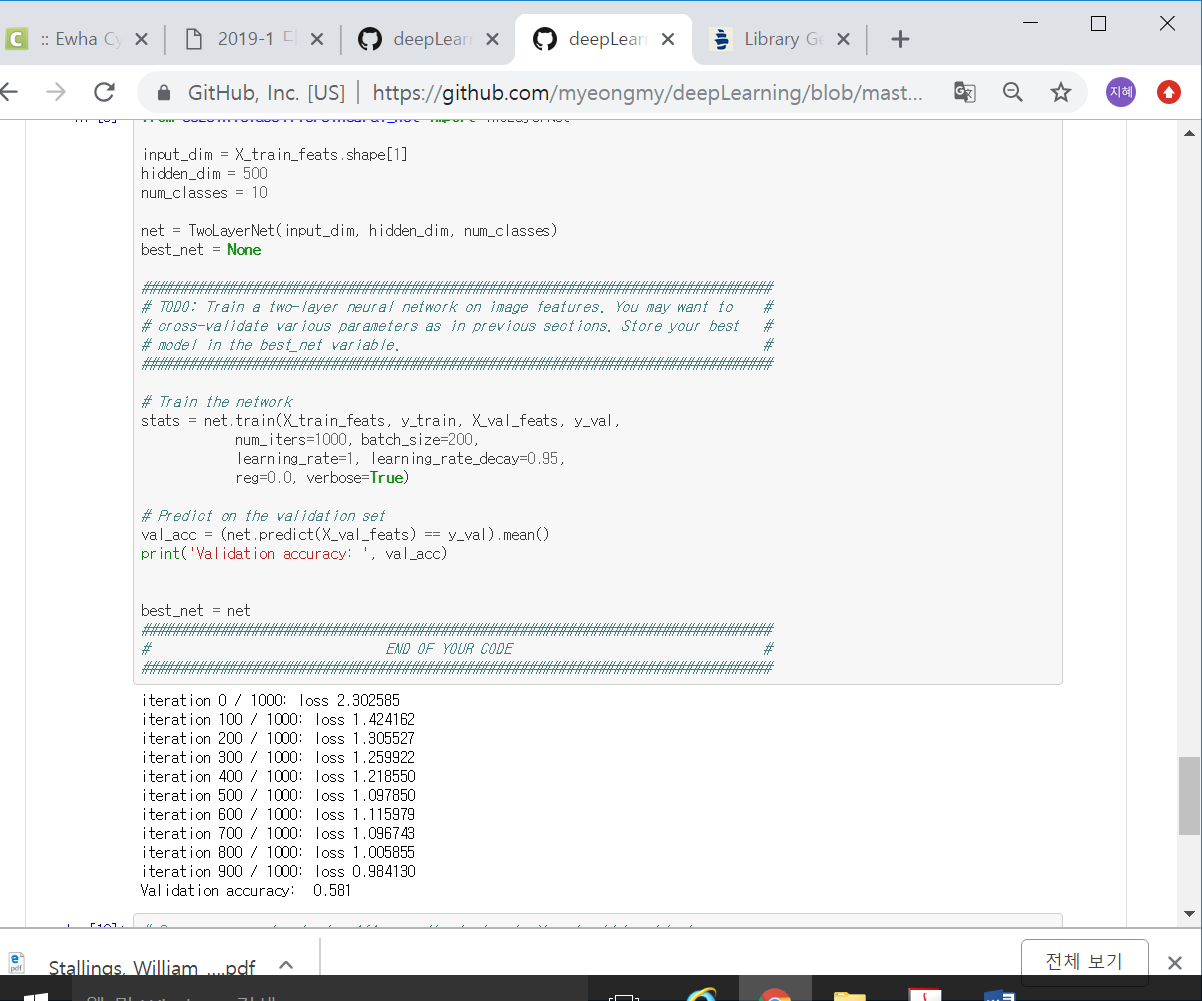
Validation set을 이용해 하이퍼파라미터인 learning rate와 regularization strength를 조정한다. Learning rate 후보와 regularization\_strength 후보를 각각 배열에 저장해 두고 for문을 돌며 하나씩 선택하여 모든 경우의 수(총 9개)만큼 모델을 훈련시킨다. 위에서 추출한 특징들을 가지고 훈련시키고 가장 정확도가 좋은 모델을 구하기 위해 best\_svm 변수에 해당 모델을 저장하도록 한다. 위의 캡처 결과를 보면 알 수 있듯이 가장 좋은 정확도를 보인 모델은 42.5%이다.



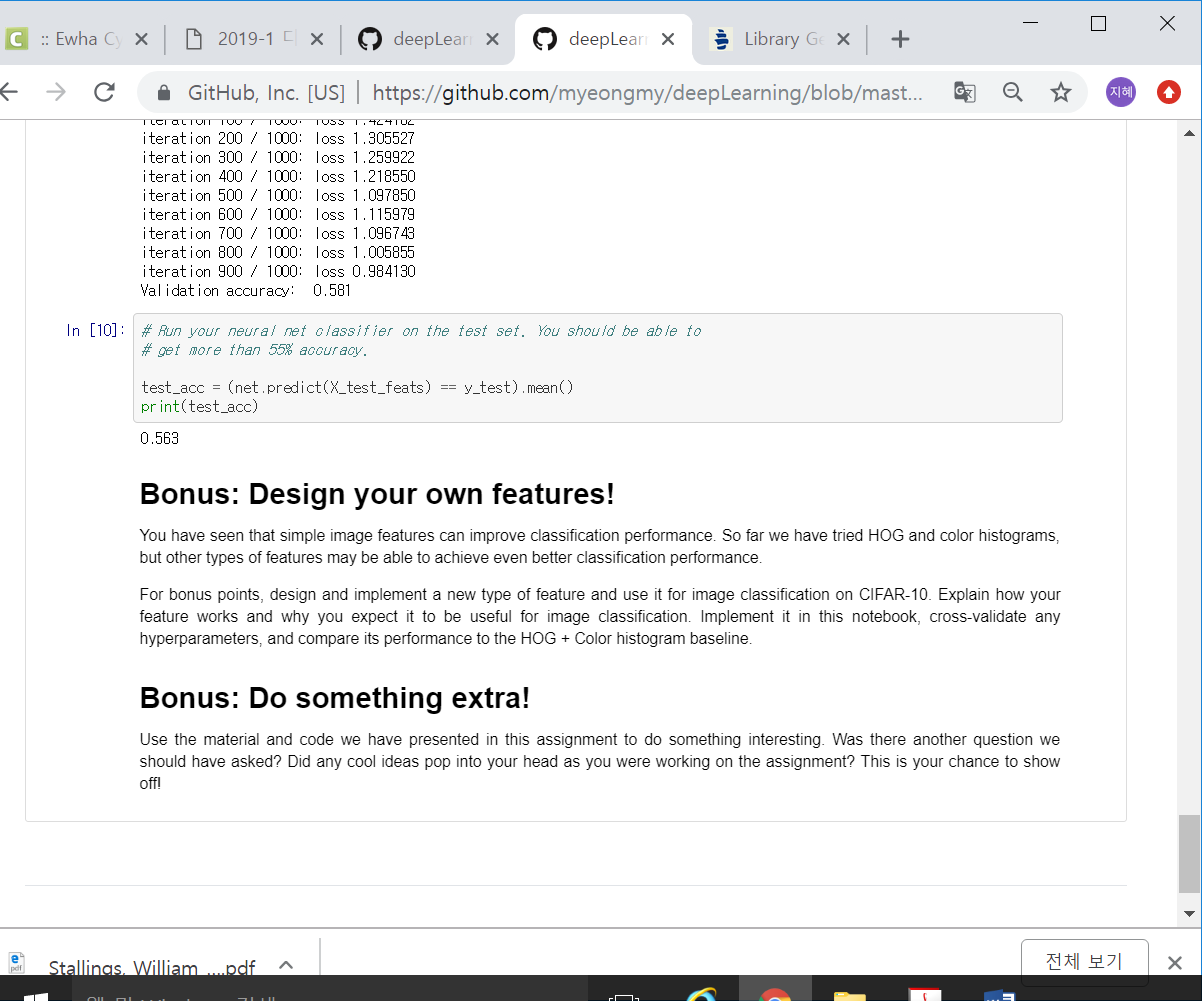
앞서 구한 best\_svm 모델을 이용하여 test set에서 돌린 결과 41.9%의 정확도가 나왔다.

1. Neural Network on image features

이번에는 이미지로부터 추출한 특징들을 이용해서 two-layer neural network를 훈련시켜본다.



다양한 hyperparameter를 가지고 실험을 해보았을 때 learning rate를 1, reg을 0으로 두고 훈련시켰을 때 정확도가 가장 높게 나왔다. 그래서 해당 모델을 best\_net 변수에 저장하였다.



위에서 구한 best\_net을 이용해 최종적으로 test set에서 돌린 결과 56.3%의 정확도가 나왔다.