CNN(convolution neural network) 구현

1123007 김현호

REQUIRE & USE

1. PYTHON 3.x

2. NUMPY

3. MATPLOTLIB

4. PIL/PILLOW

Network Model

1. VGGNET, ALEXNET 참고
2. 데이터 input size = n*3*32*32
3. (CONVOLUTION - RELU - POOLING) * 2 - (CONVOLUTION - RELU - DROPOUT - POOLING) * 2
   - FULLY CONNECTED - RELU - DROPOUT - FULLY CONNECTED - SOFTMAX

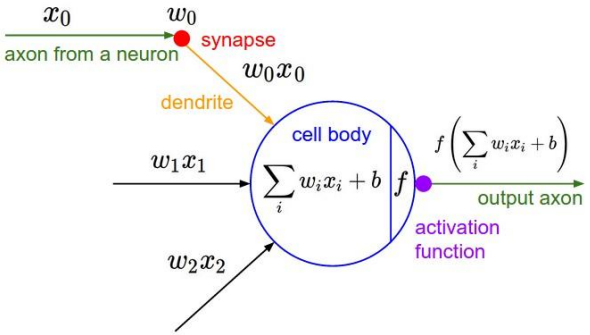
LAYER info

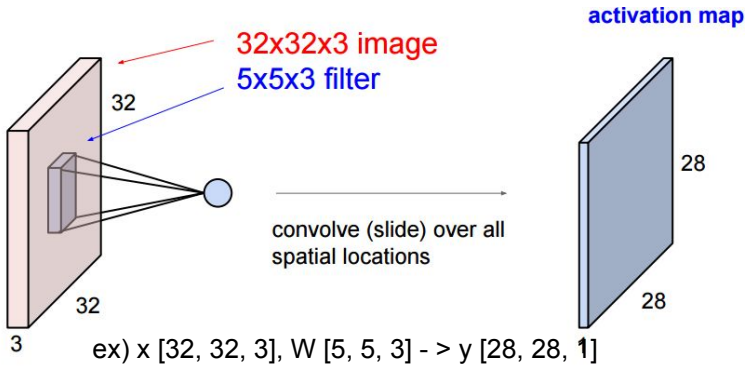
```
self.layers = OrderedDict()
self.layers['C1'] = Convolution(self.para['W1'], self.para['b1'], pad=1)
self.layers['R1'] = Relu()
self.layers['P1'] = Pooling(2, stride=2)

self.layers['C2'] = Convolution(self.para['W2'], self.para['b2'], pad=1)
self.layers['R2'] = Relu()
self.layers['P2'] = Pooling(2, stride=2)

self.layers['C3'] = Convolution(self.para['W3'], self.para['b3'], pad=1)
self.layers['R3'] = Relu()
self.layers['D3'] = Drop_out(self.drop_ratio)
self.layers['P3'] = Pooling(2, stride=2)
```

```
self.layers['C4'] = Convolution(self.para['W4'], self.para['b4'], pad=1)
self.layers['R4'] = Relu()
self.layers['D4'] = Drop_out(self.drop_ratio)
self.layers['P4'] = Pooling(2, stride=2)

self.layers['fc5'] = Affine(self.para['W5'], self.para['b5'])
self.layers['R5'] = Relu()
self.layers['D5'] = Drop_out(self.drop_ratio)

self.layers['fc6'] = Affine(self.para['W6'], self.para['b6'])

self.lastLayer = SoftmaxWithLoss()
```

CONVOLUTION LAYER

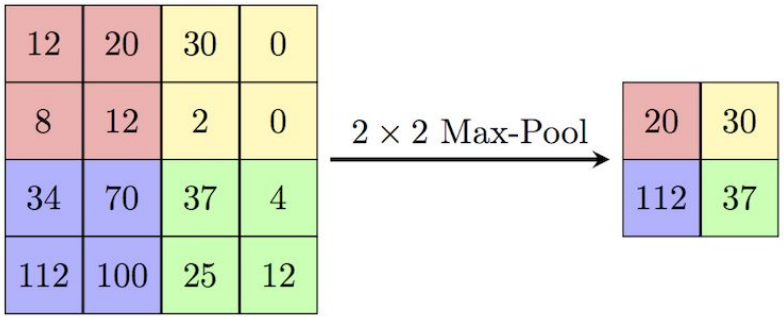
- initialize : 입력 x, 가중치 W, 편향 b, stride, pad 로 구성
- 입력과 가중치를 벡터로 변환하고 np.dot(input, weight.T) + bias 연산 (array.T = 전치행렬)
- 출력의 사이즈 : ((입력의 사이즈 + pad*2 - 가중치의 커널사이즈) / stride) + 1
- 여기서는 kernel size가 3 이고 pad가 1이기 때문에 출력사이즈의 변화는 없다.





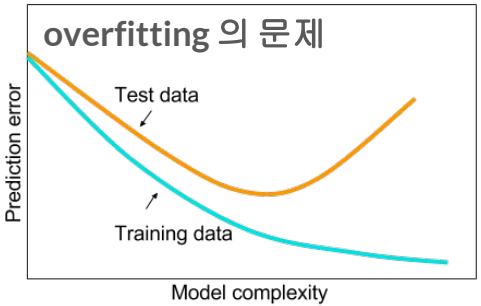
RELU and MAX - POOLING

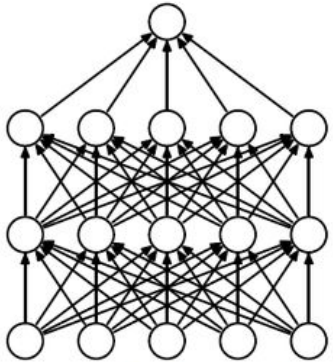
- activation function 으로 Relu, (x = max(0, x))
- max pooling 으로 size=2*2, stride=2 를 가지며 출력의 가로세로는 입력의 가로세로의 절반 (pool(x[32, 32,3]) = y[16, 16, 3]) 이며 2*2커널, stride 2 마다 가장큰 값을 구한다.



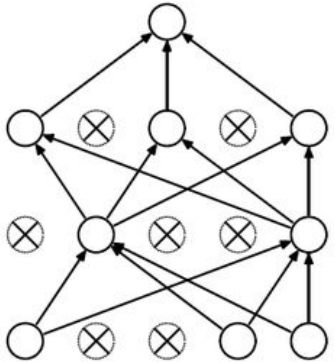
DROP-OUT LAYER

- 비율 0.5 의 드롭아웃 레이어
- 학습시 50%의 뉴런은 비활성화
- 과적합(over fitting)을 보완한다.





(a) Standard Neural Net



(b) After applying dropout.

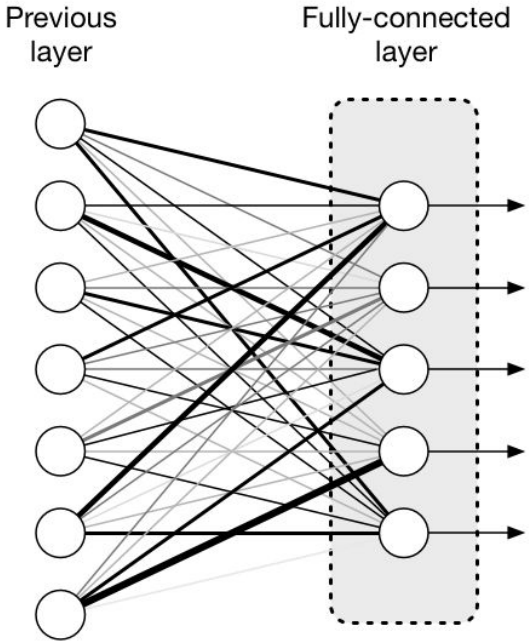
FULLY CONNECTED LAYER

- 이전레이어의 모든 뉴런이 연결되있는 AFFINE 레이어

- 입력과 가중치가 매트릭스곱이 가능한 형태여야함 ([100, 20] @ [20, 10])

- 마지막 FC layer에서 Softmax 함수를 사용한다.

(Softmax : 결과에 대한 0~1까지의 수치, 수치의 합은 1)



LAYER info2

input data (n * 3 * 32 * 32)

conv&relu1 (kernel (16,3, 3), padding=1. stride=1) -> output (n,16, 32, 32)

pooling1(kernel (2*2), stride=2) -> output(n,16, 16, 16)

conv&relu2 (kernel (32, 3,3), padding=1. stride=1) -> output (n,32, 16, 16)

pooling2(kernel (2*2), stride=2) -> output(n,32, 8, 8)

conv&relu&dropout3 (kernel (64,3,3), padding=1. stride=1) -> output (n,64, 8, 8)

pooling1(kernel (2*2), stride=2) -> output(n,64, 4, 4)

conv&relu&dropout4 (kernel (128, 3, 3), padding=1. stride=1) -> output (n,128, 4, 4)

pooling1(kernel (2*2), stride=2) -> output(n,128, 2, 2)

fully connect5&relu&dropout kernel(1024, 512) -> output (n,1024)

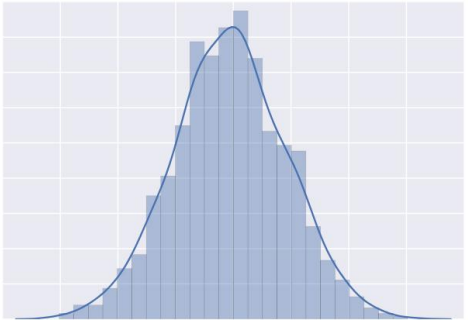
fully connect6&softmax kernel(num of class, 1024) -> output(n, num of class)

Weight initialize

- He initialization : numpy.random.randn(w.shape) * sqrt(2/input size)
- 평균이 sqrt(2/input size) 인 정규분포값
- Relu 와 잘 동작 한다고한다.

```
self.para = {}
self.para['W1'] = np.random.randn(16, 3 ,3, 3) / np.sqrt(3/2)
self.para['b1'] = np.zeros(16)

self.para['W2'] = np.random.randn(32, 16, 3, 3 ) / np.sqrt(16/2)
self.para['b2'] = np.zeros(32)

self.para['W3'] = np.random.randn(64, 32, 3, 3 ) / np.sqrt(32/2)
self.para['b3'] = np.zeros(64)
```

```
self.para['W4'] = np.random.randn(128, 64, 3, 3 ) /
np.sqrt(64/2)
self.para['b4'] = np.zeros(128)

self.para['W5'] = np.random.randn(1024, 128 * 2 *
2).transpose(1, 0) / np.sqrt((128 * 2 * 2)/2)
self.para['b5'] = np.zeros(1024)

self.para['W6'] = np.random.randn(n_class, 512 * 1 *
1).transpose(1, 0) / np.sqrt((1024 * 1 * 1)/2)
self.para['b6'] = np.zeros(n_class)
```



Optimizer : Adam optimizer

- Adam optimizer : 이전 기울기의 지수평균를 사용한 학습방법.
- gradient vanishing 및 local minima 에 대한 보완.

- 현재 네트워크에서 SGD와 SGD.Momentum 의 경우 학습시
  가중치가 inf 또는 none 또는 0(gradient vanishing)이 발생됨.
- Adam의 경우 자원을 많이 사용해서 조금 느리지만 안정적인 학습이된다.

Adam optimizer

```
class Adam:
def __init__(self, beta1=0.9, beta2=0.999, epsilon=1e-08, lr_rate=0.001):
        self.beta1 = beta1
        self.beta2 = beta2

        self.w_m = None
        self.w_v = None
        self.b_m = None
        self.b_v = None

        self.t = 0
        self.epsilon = epsilon
        self.lr_rate = lr_rate

def update(self, grads, network):
        if self.w_m == None:

        self.w_m = {}
        self.w_v = {}
        self.b_m = {}
        self.b_v = {}
```

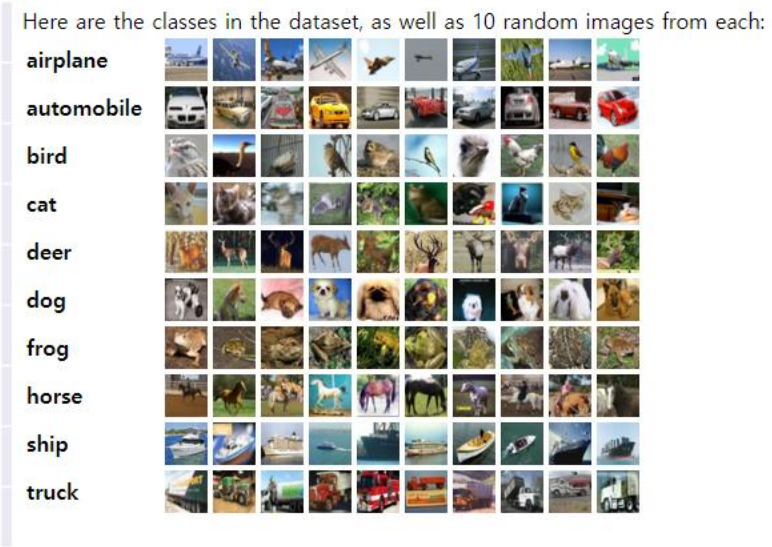
```
for i in range(1, network.layer_len+1):
        self.w_m[i] = np.zeros((grads['W' + str(i)].shape))
        self.w_v[i] = np.zeros((grads['W' + str(i)].shape))
        self.b_m[i] = np.zeros((grads['b' + str(i)].shape))
        self.b_v[i] = np.zeros((grads['b' + str(i)].shape))

        self.t += 1
        lr_t = self.lr_rate * np.sqrt(1 - self.beta2**self.t) / (1 - self.beta1**self.t)

        for i in range(1, network.layer_len+1):
                self.w_m[i] = self.beta1 * self.w_m[i] + (1 - self.beta1) * grads['W' + str(i)]
                self.b_m[i] = self.beta1 * self.b_m[i] + (1 - self.beta1) * grads['b' + str(i)]
                self.w_v[i] = self.beta2 * self.w_v[i] + (1 - self.beta2) * grads['W' + str(i)] * grads['W' + str(i)]
                self.b_v[i] = self.beta2 * self.b_v[i] + (1 - self.beta2) * grads['b' + str(i)] * grads['b' + str(i)]

        if (i<network.con_len+1):
                network.layers["C" + str(i)].W -= lr_t * self.w_m[i] / (np.sqrt(self.w_v[i]) + self.epsilon)
                network.layers["C" + str(i)].b -= lr_t * self.b_m[i] / (np.sqrt(self.b_v[i]) + self.epsilon)
        else:
                network.layers["fc" + str(i)].W -= lr_t * self.w_m[i] / (np.sqrt(self.w_v[i]) + self.epsilon)
                network.layers["fc" + str(i)].b -= lr_t * self.b_m[i] / (np.sqrt(self.b_v[i]) + self.epsilon)
```

USED Data 1 : Cifar-10

https://www.cs.toronto.edu/~kriz/cifar.html

- 초기 학습 데이터로 Cifar - 10 을 사용한다.
- class 가 10 개이고 각 클래스당 6000장을 갖으며 총 60000장의 데이터 세트이며 이중 10000장은 테스트세트이다.

- 여기서 사용은 50000장을쓰고 그중 1000장은 검증용으로 쓴다.



USED Data 2 : my dataset

- Class가 4 이고 각 클래스당 55장의 총 220장의 데이터 세트.
- 웹크라울러, 구글 VISION API 사용하여 얼굴부분 추출

(https://github.com/bwcho75/facerecognition/blob/master/com/terry/face/extract/crop_face.py)

- 추출된 사진으로 3*32*32사이즈로 리사이즈
- 각 55장중 10장씩 검증용으로 둔다.

-> 총 220중 180장이 학습용, 40장이 검증용

- 랜덤으로 학습용과 검증용을 나누어 n번씩 학습하며 최종 Accuracy는 n번의 학습의 Accuracy의 평균으로 둔다.

USED Data 3 : Dog vs Cat

- 클래스가 2개인 이진 분류 데이터세트

- 총 25000장의 개와 고양이 사진을 사용한다.

- 3* 32 * 32 로 이미지 리사이즈.

- 각 12500 장중 500장을 검증용으로 둔다.

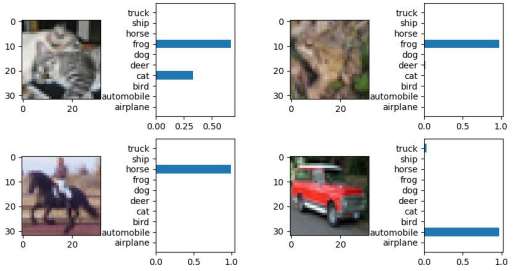
-> 전체 24000장이 학습용 1000장이 검증용.

https://www.kaggle.com/c/dogs-vs-cats

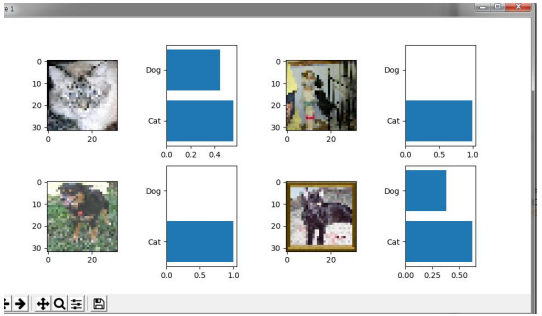
RESULT

	Clfar 10	my data	dog vs cat
Validation Accuracy	75%	78%	70%
Time & Epoch	약 2시간 epoch : 20	약 5분 epoch : 20 x5	약 40분 epoch : 10

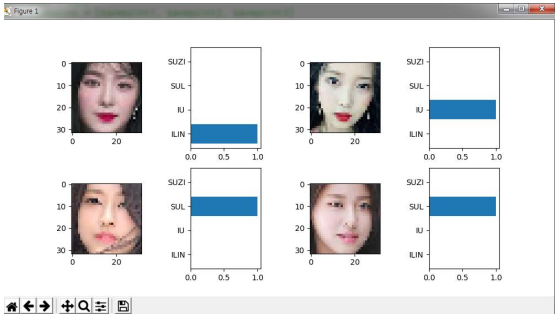
RESULT .random choice from validation data



cifar10

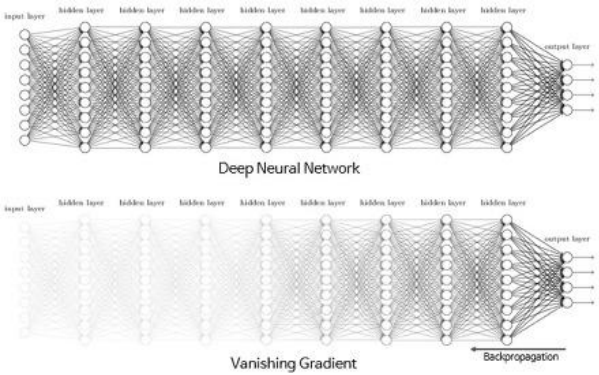


Dog vs Cat



mydata

PROBLEM & REVIEW



1. Gradient Vanishing

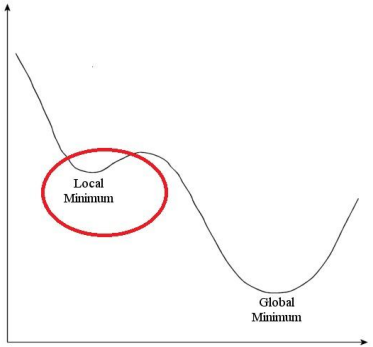
- 이번프로젝트에 아주 빈번하게 발생된 문제로 학습을 하는 수치인 기울기값(미분값)이 0으로 수렴하여 생기는 문제이다.

-> 데이터를 0~1사이로 정규화 하고 Relu와 He initialize 사용으로 상당부분 개선되었다.

2. local minima

- Gradient vanishing 문제 와 함께 발생되는 문제로 Gradient vanishing 원인이기도 한 현상으로 학습이 올바른 (이경우에는 Loss가 0과 가깝게되는)쪽으로 가지못하고 local minima에 갇혀 버리는 현상이다. 예를들면 맹인이 산을 내려갈때 발에 느껴지는 경사만 생각하고 내려가는 경우이다.

-> Adam optimizer로 상당부분 개선되었다.




```
============================
train acc : 0.74
============================
step 152 train loss : 1.058299097066743
step 153 train loss : 0.9210758568776672
step 154 train loss : 1.0611324216530746
step 155 train loss : 1.0336334259617763
step 156 train loss : 0.9768140871480026
step 157 train loss : 1.0268290705191456
step 158 train loss : 1.1324479388430764
step 159 train loss : 1.0266800669942047
step 160 train loss : 0.8458956553316175
step 161 train loss : 0.9337012670514103
step 162 train loss : 0.9967148992974398
step 163 train loss : 1.0067810826479775
step 164 train loss : 0.9471877123649945
step 165 train loss : 1.0554025421400963
step 166 train loss : 0.9456731475358892
step 167 train loss : 1.0326462843259727
step 168 train loss : 1.0536619966259073
step 169 train loss : 0.949582877379569
step 170 train loss : 0.9910259028377228
step 171 train loss : 0.8889007612117112
step 172 train loss : 0.813130303773432
step 173 train loss : 0.8554457266525735
step 174 train loss : 1.0526123672195729
```