CNN의 지도학습 기법 구현

이재형

국민대학교 전자공학부

harry8121@kookmin.ac.kr

요 약

Convolutional Nerual Network 모델을 구현하여 학습하고 검증해보았습니다.

I. 서론

이번에는 하나의 convolution layet를 stride=1, pooling없이 구현하고, ReLU함수의 적용, flatten시킨 후에 FC layer로 출력을 내보내는 구조의 네트워크를 구현해보았습니다.

이번 주제의 핵심이 되는 convolution layer의 forward연산과 backward연산을 중점적으로 어떻게 구현하였고, 내재되어 있는 수식들을 분석해보겠습니다.

또한, 학습 과정에 있어서 모델의 정확도를 높이기 위한 hyper parameter의 수정. 위의 2가지를 중점적으로 서술하도록 하겠습니다.

이전 코드를 차용한 부분이 많기에 변화된 점과 추가된 점들을 중심으로 서술하겠습니다.

II. 수행 내용

2. forward, backward 연산 구현

2.1 Convolution 연산을 위한 data reshape

3가지 class를 갖는 toy dataset은 모두 특정 패턴을 갖는 흑백이미지 입니다. 3가지 이미지로부터 data가 생성되는데, 이미지의 shape의 형태를 살펴보면 (이미지수, 가로 크기, 세로 크기)입니다.

convolution연산은 필터를 통해 이미지의 영역을 sliding하며 연산되는데, 이때 이미지의 채널 수와 필터의 채널 수가 동일해야합니다. 흑백 이미지의 채널 수는 1이므로 이미지로부터 생성된 data를 reshape하여
(이미지 수, 채널 수 = 1, 가로 크기, 세로 크기)와 같은 shape를 갖도록 하였습니다.

마찬가지로 필터의 역할을 하는 self.K1의 파라미터를 ConvNet2 class의 생성자에 초기화할 때
(커널 수, 채널 수 = 1, 가로 크기, 세로 크기)와 같은 shape를 갖도록 하여 동일한 채널 수를 기반으로 conv연산이 가능케 하였습니다.

```
# self.K1 : 필터(weight)를 의미, k_sz는 가로/세로 차원, k_num는 필터의 수
self.K1 = 0.001 * np.random.randn(self.kernel_num, 1, self.kernel_sz, self.
```

필터의 shape 형태

```
# sample data mini-batch
batch_data, batch_labels = get_mini_batch(batch_sz, db_data, db_labels, num_data)
batch_data = batch_data.reshape(batch_data.shape[0], 1, batch_data.shape[1], batch_dat
```

batch_data의 shape 형태

위는 필터와 data를 reshape하는 코드입니다. mini-batch단위로 batch_data로 학습 시키므로 batch_data를 뽑은 후에 reshape해주어서 forward 연산에 들어갈 수 있도록 구현한 것입니다.

추가적으로 학습 후에 test data를 통해서 모델의 성능을 검증할 때, test data또한 batch_data의 shape와 같은 형태로 만들어주기 위해 toy dataset을 train/test data로 분할 하는 셀의 부분에서 test_data또한 reshape 해주었고 그 코드는 아래와 같습니다.

```
te_data = te_data.reshape(te_data.shape[0], 1, te_data.shape[1], te_data.
```

test_data의 shape 형태

2.2 ConvNet2 클래스 정의

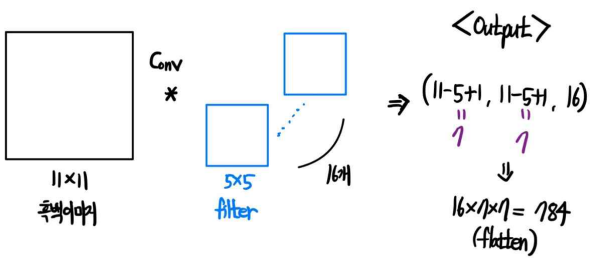
다음은 ConvNet2 class의 생성자 부분입니다.
이미지의 사이즈, 필터의 사이즈, 필터의 수, 클래의 수를 정의합니다.
또한, Conv연산과 FC layer에서 이루어지는 선형 연산에 사용되는 파라미터들의 초기화 합니다.

- **self.K1** : 필터의 가중치
- **self.b1** : conv layer의 각 필터에 대한 편향
- **self.K_outdim** :
  conv layer의 출력 차원, Conv연산 후에 FC layer로 전달되기 전에 flatten될 것으로 예상되는 차원 수를 의미
- **self.W2** : FC layer의 가중치
- **self.b2** : FC layer의 편향

이렇게 정의된 변수들을 바탕으로 구현된 forward 함수에 대해서 서술하겠습니다.

2.3 Forward 연산

먼저 Convolution연산이 현재 주어진 데이터의 형태와 필터의 형태로 인하여 어떻게 연산되는지 생각해보았습니다.



11x11 크기의 흑백이미지와 5x5크기의 필터 16개와 conv연산을 거치고 나면 (7, 7, 16) 크기의 feeture map이 도출되고 이를 flatten하면 7x7x16 = 784차원이 됩니다. 이를 self.K_outdim을 통해 위의 생성자 부분에서 표현하였습니다.

이제 conv연산할 준비가 되었으니 forward연산의 구현을 통해 어떻게 conv연산이 진행되고 flatten되어 FC layer를 통해 score가 계산되는지 서술하겠습니다.

1. data, output정의

- **self.data = data**
- **batch_sz = data.shape[0]** : 입력 데이터의 첫 번째 차원이 batch의 크기를 나타내므로 이를 batch_sz에 저장합니다.
- **self.conv_out = np.zeros(batch_sz, self.kernel_num, self.img_sz - self.kernel_sz + 1, self.img_sz - self.kernel_sz + 1))**
  : conv연산의 출력을 저장할 빈 행렬을 초기화합니다. 위의 그림에서 알 수 있듯이 conv연산결과는 img_sz - k_sz + 1의 가로, 세로 크기를 갖고 필터의 수만큼의 출력 차원이 도출됩니다. 추가적으로 batch_sz를 고려하여 코드를 작성했습니다.

2. conv 연산

```
for i in range(batch_sz):
    for k in range(self.kernel_num):
        for h in range(self.img_sz - self.kernel_sz + 1):
            h1 = h
            h2 = h1 + self.kernel_sz
            for w in range(self.img_sz - self.kernel_sz + 1):
                # Apply the kernel on the image patch and sum the result
                w1 = w
                w2 = w1 + self.kernel_sz
                self.conv_out[i, k, h, w] = np.sum(self.data[i, :, h1:h2, w1:w2] * self.K1
```

conv연산을 수행하는 for문

위의 4중 for문을 하나하나 서술하겠습니다.

1. i Loop
- i 루프는 batch내의 각 이미지에 대해 반복합니다.

2. k Loop
- k 루프는 conv layer 내의 각 필터를 순회합니다.

3. h, w Loop
- h, w의 루프는 이미지 내에서 필터가 이동할 수 있는 모든 위치를 순회합니다.

- h1, h2, w1, w2는 각각 필터가 적용되는 이미지 패치의 높이와 너비를 정의합니다.

4. Convolution 연산 수행
- 위에서 정의한 conv연산의 output이 담길 self.conv_out에 for문을 순회하며 현재 위치에서의 conv연산을 수행하여 저장합니다.

- self.data[i, :, h1:h2, w1:w2]는 입력 이미지에서 필터가 적용될 패치 부분을 가져오고 이를 현재 이미지 위를 순회 중인 필터인 self.K1[k]와 요소 별 곱 즉, 내적 연산을 하고 그 결과를 합산합니다.

3. ReLU 적용, FC layer 통과

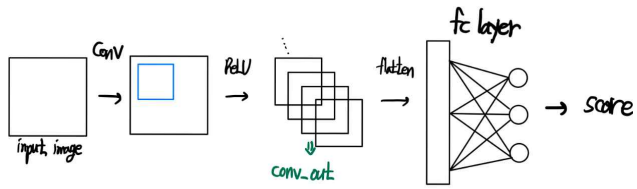
- conv 연산 결과에 ReLU함수를 통해 음수 값을 제거하며 비선형성을 도입하고, conv layer의 출력을 flatten시켜 1차원 벡터로 변환합니다.

- 이를 FC layer로 전달하여 self.score를 계산합니다. 이때 차원의 형태를 살펴보면 flattened는 (batch_sz, self.K_outdim)의 차원을 갖고, self.W2는 (self.K_outdim, num_cls)의 차원을 갖기에 선형연산이 가능하며 그 연산의 결과 차원은 (batch_sz, num_cls)가 됩니다. 이는 self.score에 저장됩니다.

```
# ReLU activation function
self.conv_out = np.maximum(self.conv_out, 0)

# Flatten the convolution output
flattened = self.conv_out.reshape(batch_sz, -1)
#print(flattened.shape)

# Fully Connected Layer
self.scores = np.dot(flattened, self.W2) + self.b
self.num_data = batch_sz

return self.scores
```

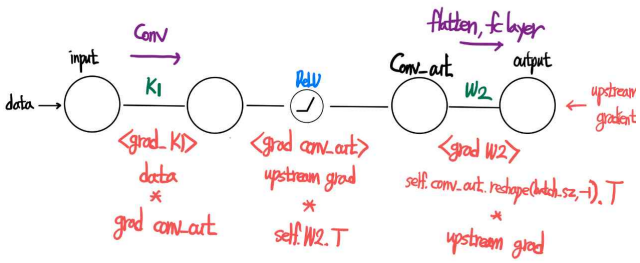
ReLU함수의 적용과 FC layer

forward연산의 전 과정을 그림으로 나타내면 아래 그림과 같습니다.



2.4 backward 연산

앞선 forward연산을 구현하였고, 이제 backward연산을 구현한 함수에 대한 설명과 backpropagation이 어떻게 일어나는지 서술하겠습니다.



backward과정을 computational graph로 나타내면 위와 같습니다. 순서대로 살펴보겠습니다.

1. grad_W2

- FC layer에 대한 grad_W2는 self.con_out을 flatten하여 FC layer의 입력 형태로 변환한 후에 upstream_grad와의 행렬 곱을 통해 계산합니다.

2. grad_b2

- FC layer의 편향에 대한 gradient로 각 클래스에 대한 gradient를 모두 더하여 편향에 대한 gradient 값을 얻습니다.

3. grad_conv_out

- 이는 conv layer의 출력에 대한 gradient를 나타내며, FC layer의 upstream_grad와 전치 된 self.W2의 행렬 곱 연산을 통해 계산합니다.

- 이후에 이 gradient를 원래 형태인 self.conv_out의 형태로 다시 변환해줍니다. 이는 신경망의 역전파 과정에서 각 레이어에 해당하는 gradient를 올바르게 전달하기 위함입니다.

4. ReLU 함수의 gradient

- 이전 레이어의 출력이 음수일 때 gradient를 0으로 설정하여 음수 영역의 gradient를 제거합니다.

5. grad_K1

- 이는 Conv layer의 필터에 대한 gradient로 입력 이미지와 Conv layer의 출력의 gradient를 곱하여 필터에 대한 gradient를 얻습니다.

- 이를 연산하는 for문의 형태는 forward의 for문 형태와 아주 흡사합니다. 데이터와 Conv layer의 출력에 대한 gradient를 곱하고 이를 누적하여 grad_K1에 저장함을 알 수 있습니다.

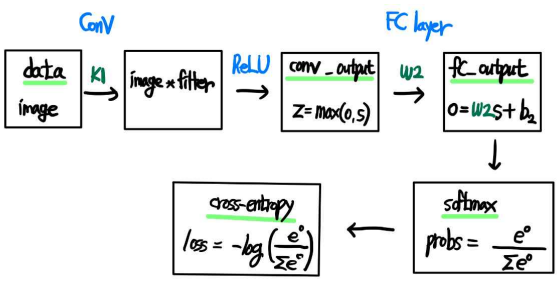
6. grad_b1

- 이는 Conv layer의 편향에 대한 gradient로 각 필터에 대한 gradient를 더하여 편향에 대한 gradient를 구합니다.

- 여기서 axis = (0, 2, 3)인 이유는 axis = 0은 배치 차원에 대한 합을 의미하여 각 샘플에 대한 gradient를 합산하는 것이고, axis = 2, 3은 높이와 너비 축에 대한 합을 의마하는 것입니다. 이는 grad_conv_out의 shape가
(batch_sz, kernel_num, conv_out_height, conv_out_width)
의 형태이기 때문입니다.

이후 softmax와 cross-entropy까지 적용한 것을 도식화하면 아래와 같습니다.

-> 위의 backward코드는 ChatGPT를 활용하여 도출되었습니다.[1]



3. 실험 결과 및 분석

이렇게 구현된 CNN모델을 통해 test data의 판별 결과를 도출해보았습니다.

- te_scores = model.forward(te_data)
- te_predictions = np.argmax(te_scores, axis=1)
- test data의 score를 계산하고, 판별 결과를 가장 높은 score를 갖는 클래스로 선택하여 변환하였습니다.

- correct_predictions = (te_predictions == test_labels)
- 예측값과 실제 레이블이 일치하는 경우 True

- Acc = np.mean(correct_predictions.astype(float))
- print('Classification accuracy= {}%'.format(Acc*100))
- 정확한 예측의 비율을 계산하여 백분율로 환산하였습니다.

하이퍼파라미터를 설정해가면서 모델의 정확도를 측정해보았습니다. batch_sz와 필터의 크기, 수는 각각 8, 5, 16으로 유지한 상태에서 epoch와 lr를 바꾸어가며 정확도를 관찰하였습니다.

- lr : 0.001 : 에포크 수를 아무리 늘려도 모델의 정확도가 33.333%에 머물렀고, loss값을 출력해보았을 때 loss가 업데이트 되긴하지만 크게 줄어들지 않음을 확인하였습니다.

- lr : 0.0001 : 0.001과 마찬가지의 결과가 도출되었습니다

- lr : 2e - 4 : 마찬가지로 loss값이 줄어들지 않았습니다.

아래 사진은 위의 lr로 학습하였을 때의 loss값의 변화와 정확도 입니다.

```
loss= 1.0980629475899533
loss= 1.0990669046654422
loss= 1.0968232976020142
loss= 1.09649661265101
loss= 1.0975124275403831
loss= 1.1025065068098039
loss= 1.097706112745372
loss= 1.0969954479559219
loss= 1.0999153542130862
...
loss= 1.1029734580589077
loss= 1.0991129423232167
loss= 1.099091380531506
loss= 1.0966940060728767
```

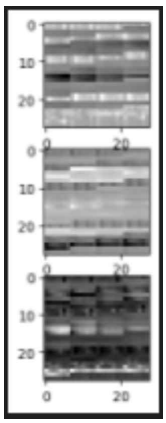
```
Classification accuracy= 33.33333333333333%
```

위의 lr를 기준으로 epoch를 아무리 늘려도 loss가 크게 변화하지 않음을 확인했고, batch수와 필터 사이즈, 수를 아무리 변화시켜도 모델의 정확도가 향상되지 않았습니다.

그래서 lr를 0.01로 증가시키고 나머지는 동일한 조건하에 학습을 진행했습니다. 그 결과 loss값이 크게 감소함을 확인할 수 있었고, 학습 정확도도 100%가 도출되었습니다. 밑의 표는 위의 결과들을 정리한 것입니다.

lr	batch	f_s	f_n	epoch	acc
0.001	8	5	16	500	33.333%
0.0001	8	5	16	500	33.333%
2e-4	8	5	16	500	33.333%
0.01	8	5	16	500	100%

```
loss= 1.0975351120076717
loss= 1.108293878743838
loss= 1.106936581983778
loss= 1.094901336487872
loss= 1.0942735455651325
loss= 1.098380747947758
...
loss= 0.011163278749573792
loss= 0.01337994943264047
loss= 0.030473918136481315
loss= 0.016161256508049934
```



```
Classification accuracy= 100.0%
```

lr를 0.01로 증가시켰을 때의 결과

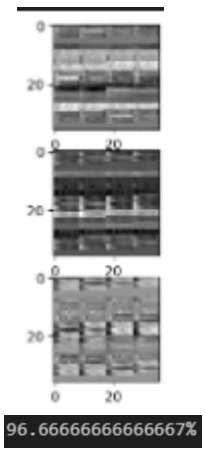
lr를 증가시켰을 때 학습의 정확도가 올라간 결과의 의미에 대해서 생각해보았습니다.

일반적으로 lr는 학습 속도와 학습 과정의 안정성에 큰 영향을 미칩니다. 너무 크면 학습을 불안정하게 하며 발산할 가능성이 존재하고, 너무 작다면 학습에 걸리는 시간이 너무 오래 소모되므로 적절한 학습률을 정하는 것은 학습에 있어서 아주 중요한 부분입니다.

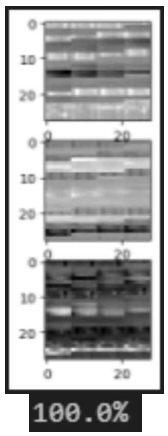
위의 경우는 lr을 증가시켰을 때 학습이 더 잘되었습니다. 이는 조금 더 높은 학습률로 인하여 local minimum에 빠지지 않고, global minimum에 더 빨리 수렴된 것이라고 생각할 수 있습니다. 정확하게 어떠한 요소와 원인으로 인하여 lr을 증가시켰을 때 학습이 더 잘되었는지에 대하여 상세히 서술할 수는 없지만, 때로는 조금 더 높은 학습률이 빠른 수렴 속도와 학습의 안정성을 보장해준다는 것을 알 수 있게 되었습니다.

다음으로는 학습률과 배치 사이즈, 에포크 등은 유지한 채 커널의 크기만을 변화시키면서 학습의 정확도를 관찰해보았습니다.

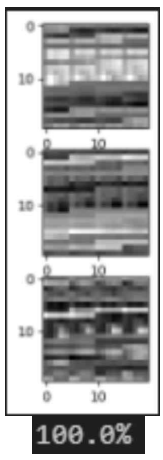
lr	batch	f_s	f_n	epoch	acc
0.01	8	3	16	500	96.667%
0.01	8	5	16	500	100%
0.01	8	7	16	500	100%
0.01	8	9	16	500	100%



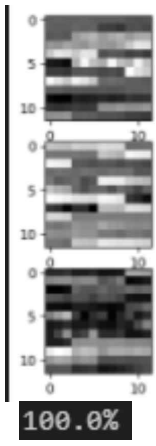
3x3 kernel



5x5 kernel



7x7 kernel



9x9 kernel

위는 표의 수치를 기반으로 필터의 크기만 변경했을 때 학습 결과와 도출된 파라미터의 사진입니다. 위의 결과를 통해 CNN에서 필터의 크기가 학습에 어떠한 영향을 미치는지 생각해보았습니다.

CNN에서 필터의 크기는 크게 2가지로 학습에 영향을 미친다는 것을 알 수 있습니다.

1. 특성 추출 능력
 - 작은 커널은 더 많은 세부 정보를 캡쳐할 수 있지만, 고차원적인 추상 정보를 잡아내기는 한계가 존재합니다. 반면에 큰 필터는 보다 넓은 영역의 정보를 캡쳐하여 추상 정보를 더 학습할 수 있습니다.

2. 계산 비용
 - 큰 커널은 작은 커널보다 더 많은 파라미터 수를 갖고 있으므로 연산 비용이 더 높을 수 있습니다. 이는 모델 학습에 계산량을 증가시킵니다.

정해진 해답은 없지만, 데이터셋의 특징과 복잡성에 따라 최적의 필터의 크기를 선택하는 것 또한 모델의 학습 성능에 큰 영향을 미친다는 것을 알게 되었습니다.

4. 결론

단순한 convolution layer지만 그 과정에서 이미지 데이터와 필터 간의 연산과정을 살펴보면서 CNN의 forward연산과 backward연산의 세부과정을 파악할 수 있게 되었습니다.

또한, 학습 알고리즘을 잘 구현한 상태에서도 모델의 안정적인 학습 과정과 좋은 성능을 얻기 위해서는 데이터의 특징에 맞는 적절한 하이퍼파라미터를 선정하는 것이 매우 중요함을 깨달았습니다.

결국 주어진 data를 기반으로 어떠한 task에 대한 performance를 극대화하는 전체적인 과정에 고려할 점들이 아주 많다고 느꼈습니다. 그러한 과정 속에서 특정 알고리즘이나 하이퍼파라미터 등이 어떠한 작용을 하는지를 잘 파악하고 있는 것이 모델 학습 시 모델의 일반화 성능을 극대화하는 것에 큰 영향을 미친다는 것을 알 수 있었습니다.

참고문헌

[1] chat-gpt(ver 3.5) : 2023.11.25접속
[2] 강의자료 - Linear Classifier & Linear Regression : computational graph를 작성하고 수식을 작성하는데 활용하였습니다.
[3] https://yjjo.tistory.com/9 : CNN의 backpropagation 연산을 이해하는데 도움이 되었습니다.