Multi-Layer Perceptron구조의 Neural Network의 지도학습 기법 구현

이재형

국민대학교 전자공학부

harry8121@kookmin.ac.kr

요 약

MLP구조의 nerual network를 구현하고 MNIST data를 통해 모델을 학습하고 검증해보았습니다.

I. 서론

저번에는 하나의 layer를 갖는 단일 퍼셉트론 구조였다면, 이번에는 하나의 hidden layer를 갖는 MLP 구조의 nerual network를 구현하였습니다.

이번 주제의 핵심이 되는 오류 역전파를 계산하는 backward함수의 구현에 있어서 gradient연산 과정과 더불어 mini-batch를 이용한 것.

또한, 학습 과정에 있어서 모델의 정확도를 높이기 위한 hyper parameter의 수정. 위의 3가지를 중점적으로 서술하도록 하겠습니다.

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

II. 수행 내용

2.1 forwarrd, backward 연산 구현

2.1.1 forward연산

input layer부터 하나의 hidden layer를 거쳐 output layer까지의 연산과정을 구현하였습니다.

hidden layer의 출력에 ReLU 함수를 적용하였고, 그 후에 output layer의 출력을 구하였습니다.

- hidden layer : w1 * x + b1 수식 구현, ReLU함수 적용

```
# 첫번째 hidden layer
z1 = np.dot(data, self.W1) + self.b
a1 = np.maximum(0, z1)
self.hidden_layer_output = a1
```

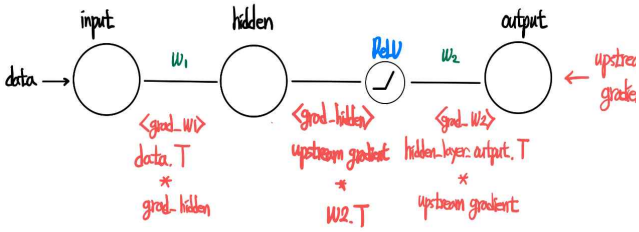
hidden layer의 forward연산

```
# Output layer
self.scores = np.dot(a1, self.W2) + self.
```

output layer forward연산

2.1.2 backward 연산

다음은 backward 연산과정의 구현입니다. 우선적으로 upstream gradient와 local gradient를 곱해가는 역전파 과정을 밑의 그림과 같이 나타낼 수 있습니다.



backward과정에서 나오는 self.hidden_layer_output은 forward연산에서 hidden layer의 출력값을 저장해놓은 변수로서 backward연산에서 이용됩니다.

w2에 대한 gradient는 hidden layer의 출력과 upstream gradient의 내적으로 나타낼 수 있습니다. 이 때, shape를 고려해보겠습니다.

- 은닉층의 출력의shape :N x num_hid_neurons
- upstream gradient의 shape : N x num_cls

각각의 shape는 위와 같고 이를 연산하기 위하여 은닉층의 출력을 전치하여 연산하였습니다.

```
grad_scores = upstream_grad
grad_W2 = np.dot(self.hidden_layer_output.T, grad_scores) / bat
grad_b2 = np.sum(grad_scores, axis=0) / batch_sz
```

다음은 은닉층의 gradient입니다. upstream gradient와 w2의 내적으로 표현됩니다. 이 또한 shape를 고려하여 w2를 전치하여 내적합니다.

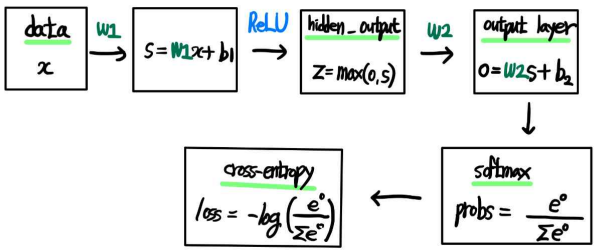
또한, ReLU함수를 고려하여 출력이 0이하인 요소들에 대하여 0으로 처리하여 gradient를 계산합니다.

```
# hidden layer 역전파 계산
grad_hidden = np.dot(grad_scores, self.W2.T)
grad_hidden[self.hidden_layer_output <= 0] =
```

마지막으로 w1에 대한 gradient입니다.
hidden layer의 gradient와 input data의 내적연산으로 표현됩니다.

```
grad_W1 = np.dot(self.data.T, grad_hidden) / batch_s
grad_b1 = np.sum(grad_hidden, axis=0) / batch_sz
```

위의 전 과정을 softmax와 cross entropy까지 적용하여 나타내면 아래의 그림과 같습니다.



2.2 mini-batch 단위의 gradient연산

위의 backward함수에서 gradient를 계산할 때, 계산된 gradient를 batch_sz로 나누어 미니 배치의 크기에 따른 평균을 사용하였습니다. 처음에 mini-batch로 나누지 않았을 때 모델의 정확도가 현저히 낮은 것을 확인할 수 있었습니다.

SGD를 고려하여 batch_sz로 나누어 gradient의 평균을 구해보았을 때 모델의 정확도가 향상되었습니다. 모든 train data를 사용하여 한 번에 가중치를 업데이트 하지 않고, 각 mini-batch에 대해 가중치를 업데이트하여 전체 데이터를 효과적으로 학습한 것입니다.

이는 train_mlp_neuralnet함수에서 gradient를 계산할 때 호출되는 compute_grad를 거쳐 backward까지 batch_sz의 매개변수를 전달하여 backward에서 SGD를 통한 gradient를 계산할 수 있도록 구현하였습니다.

```
# compute gradient (backpropagation)
grad = loss_fn.compute_grad(batch_labels, batch
```

compute_grad에 batch_sz 매개변수 추가하여 호출

```
def compute_grad(self, labels, batch_sz):
    # score dimensions should be N x M
    z = self.probs
    # grad_z dimensions should also be N x M
    grad_z = z - make_labels_onehot(self.labels,self.nu

    return self.model.backward(grad_z, batch_sz)
```

compute_grad내에서 backward호출시 batch_sz전달

```
def backward(self, upstream_grad, batch_
```

batch_sz를 매개변수로 전달받아서 grad연산 실행

3. 실험 결과 및 분석

이렇게 구현된 MLP구조의 NN을 MNIST data를 통해서 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.

```
te_scores = model.forward(te_data)
# 판별 결과를 클래스로 변환 (가장 높은 점수를 갖는 클래스 선택
te_predictions = np.argmax(te_scores, axis=1)

# 정확도 계산
# 예측값과 실제 레이블이 일치하는 경우 True
correct_predictions = (te_predictions == te_labels)
# 정확한 예측의 비율 계산
Acc = np.mean(correct_predictions.astype(float))

print('Classification accuracy= {}%'.format(Acc*100))
```

learning rate는 2e-4로 고정하고, batch_sz와 epoch 수, num_hid_neurons 수를 조정해가며 학습에 대한 정확도를 측정해보았습니다.

batch_sz = 256으로 설정하고, num_hid_neurons = 128을 설정한 후에, epoch를 증가시키면서 정확도를 살펴보았습니다. epoch수를 증가시킬수록 정확도가 상승되는 것을 확인할 수 있었습니다.

epoch수를 키우는 것이 모델의 일반화 성능을 향상시킨다고는 할 수 없습니다. epoch수를 무작정 키우게 되면 train data에 대한 정확도는 계속하여 증가할 가능성이 크지만, 이는 overfitting의 가능성을 높일 수 있기 때문입니다. 하지만, 위의 결과를 통해서 test_data에 대한 검증 정확도가 일반적으로 epoch수를 증가시킴으로서 증가한다는 점을 확인할 수 있었습니다.

그렇다면, 학습에 영향을 주는 하이퍼파라미터의 종류와 각각이 학습에 갖는 의의에 대해서 알아보았습니다.

1. 학습률(Learning Rate)
 - 가중치를 업데이트 시 얼마나 큰 보폭으로 이동할지 결정합니다. 너무 작으면 학습 속도를 늦출 수 있고, 너무 큰 학습률은 발산할 수 있기에 적절한 학습률을 찾는 것이 중요합니다.

2. 미니 배치 크기(Mini-batch size)
 - 학습 시 각 반복에서 사용되는 train data의 샘플 수를 결정합니다. 작은 미니 배치 크기는 메모리를 절약하고 빠른 학습을 가능하게 하지만, 큰 미니 배치 크기는 일반화에 도움이 됩니다.

3. 에포크 수(Number of Epochs)
 - 전체 train dataset을 몇 번 반복할지 결정합니다. 에포크 수가 적으면 모델이 데이터에 덜 fitting될 수 있지만, 너무 많다면 overfitting의 위험이 있습니다.

4. 은닉층 수와 뉴런 수
 - 은닉층수와 뉴런 수에 의해 신경망의 구조를 결정합니다. 더 깊은 네트워크나 더 많은 뉴런은 모델의 표현력을 증가시킬 수 있찌만, 더 많은 계산량이 필요하여 overfitting의 위험이 있습니다.

5. 가중치 초기화 방법
 - 가중치를 초기화하는 방법은 다양합니다. 적절한 초기화는 학습 속도와 수렴을 향상시킬 수 있습니다.

6. 활성화 함수
 - 각 뉴런의 출력을 결정하는 함수입니다. 비선형 함수를 사용하여 모델의 표현력과 학습 속도에 영향을 미칩니다.

7. 배치 정규화(Batch Normalization)
 - 각 층의 입력을 정규화하여 학습을 안정화합니다 학습 속도를 향상시키고 초기화에 덜 민감하게 만들 수 있습니다.

8. 드롭아웃 비율(Dropout Rate)
 - 뉴런을 랜덤하게 제거하여 과적합을 방지하는 방법입니다. overfitting을 방지하고 일반화능력을 향상시킬 수 있습니다.

위와 같은 하이퍼파라미터들을 적절하게 조절하고 튜닝하는 것이 모델을 효과적으로 학습시키는데 중요한 역할을 합니다. 수많은 시행착오와 실험, 검증을 통해 최적의 하이퍼파라미터를 찾는 것이 중요합니다.

4. 결론

단순한 MLP구조의 NN을 구현하여 MNIST data를 통해 학습해보았습니다.
학습 알고리즘인 경사하강법을 mini-batch를 통한 SGD로 구현하고, 학습 시의 에포크 수, 배치사이즈, 은닉층 내의 뉴런 수 등의 하이퍼파라미터를 조정해가며 test data에 대한 모델의 일반화 성능을 향상시키는 작업을 진행하면서 하이퍼파라미터의 중요성을 깨닫게 되었습니다.

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) : 학습 시 영향을 미치는 하이퍼파라미터의 종류와 그것들의 역할, 의의에 대한 개념을 얻었습니다.
[2] 강의자료 - Linear Classifier & Linear Regression : computational graph를 작성하고 수식을 작성하는데 활용하였습니다.