<48시간 vision task 보고서>

-Image classification with dataset

of noisy label version of CIFAR-100 dataset-

작성자: 광운대학교 컴퓨터정보공학부 고가연

작성시간: 2023.07.20 10 PM ~ 2023.07.22 10 PM

I. Methodology

1. Architecture overview

해당 과제를 해결하기 위해서 4개의 합성곱 층(Conv2d)과 3개의 완전 연결 층(Linear)으로 구성된 간단한 CNN(Convolutional Neural Network) 모델을 사용했습니다. 모델의 입력으로는 100개의 클래스로 구성된 이미지 데이터를 받고, 일부 Noisy label을 갖고 있는 CIFAR-100 데이터셋의 이미지 분류 문제를 해결하기 위해 설계되었습니다. 각 입력 이미지에는 하나의 클래스 레이블이 있습니다. 모델은 합성곱 층을 통해 입력 이미지의 특징을 추출하고, 완전 연결 층을 통해 추출된 특징을 기반으로 각 클래스에 대한 확률 분포를 출력합니다.

모델의 구조는 다음과 같습니다.

① self.conv1: 첫 번째 합성곱 층으로, 입력 채널 수가 3이고 출력 채널 수가 64인 필터(커널)를 사용합니다. 입력 이미지에 3x3 크기의 필터를 적용하며, 경계를 따라 2픽셀로 패딩하여 입력과 출력의 크기를 동일하게 유지합니다.

② self.conv2: 두 번째 합성곱 층으로, 입력 채널 수가 64이고 출력 채널 수가 256인 필터를 사용합니다. 마찬가지로 3x3 크기의 필터를 사용하며, 패딩을 사용하여 입력과 출력의 크기를 유지합니다.

③ self.conv3: 세 번째 합성곱 층으로, 입력 채널 수가 256이고 출력 채널 수가 1024인 필터를 사용합니다. 여기서도 3x3 크기의 필터와 패딩을 사용합니다.

④ self.conv4: 네 번째 합성곱 층으로, 입력 채널 수가 1024이고 출력 채널 수가 4096인 필터를 사용합니다. 여기서도 3x3 크기의 필터와 패딩을 사용합니다.

⑤ self.fc1: 첫 번째 완전 연결 층으로, 합성곱 층에서 나온 특성 맵을 1차원으로 펼쳐서 36864개의 뉴런과 연결합니다. 이 층은 36864차원의 입력을 받아 800차원의 출력을 내보냅니다.

⑥ self.fc2: 두 번째 완전 연결 층으로, 800차원의 입력을 받아 500차원의 출력을 내보냅니다.

⑦ self.fc3: 세 번째 완전 연결 층으로, 500차원의 입력을 받아 100차원의 출력을 내보냅니다. CIFAR-100 데이터셋은 100개의 클래스로 구성되어 있기 때문에 100차원의 출력으로 입력 이미지들을 분류할 수 있도록 합니다.

⑧ self.relu: ReLU(ReLU(Rectified Linear Unit)) 활성화 함수를 사용하여 각 층의 출력에 비선형성을 부여합니다.

⑨ self.max_pool2d: 2x2 크기의 최대 풀링(Max Pooling)을 수행하여 공간 해상도를 줄이고 특성을 강조합니다.

⑩ self.dropout: Dropout을 적용하여 과적합을 방지하고 모델의 일반화 성능을 향상시킵니다.

데이터셋은 kaggle에서 제공하는 noisy label 버전의 CIFAR-100 데이터셋을 다운받은 후, 전처리하는 과정을 거쳤습니다. './dataset'이라는 디렉토리에 저장되어 있는 'cifar100_nl.csv' 파일을 열어서 반복문을 통해 한 행씩 읽으면 이미지 파일이 저장되어 있는 경로와 해당 이미지 파일의 클래스(라벨)가 ','로 구분되어 저장되어 있는 것을 알 수 있습니다. 결과적으로 훈련 데이터셋 49999개의 이미지 파일과 검증 데이터셋 9999개의 이미지 파일로 제공되는 것을 확인했습니다. 추후에 발생할 수 있는 오버피팅(과적합, overfitting) 문제를 방지하기 위해 49999개의 훈련 이미지 데이터셋을 나누어서 10000개의 validation dataset도 구성해 두었습니다. 따라서, CNN 모델에 입력한 데이터셋 구성은 아래와 같습니다.

training : validation : test = 39999 : 10000 : 9999

다음은 설계한 아키텍처에 대해 설명하겠습니다.

- 데이터 전처리 방법: 입력 이미지에 대해 train set, validation set, test set 총 3가지 데이터셋으로 분류했습니다. 그리고나서 각 이미지 파일을 하나씩 불러와서 32x32의 target size로 resize를 수행하고 numpy 배열로 변환했습니다. 그후 이미지 데이터를 0과 1 사이로 정규화 함으로써 모델이 학습할 시 안정성을 높여주도록 만들었습니다. 따라서 각 데이터셋의 .npy 파일을 생성해서 따로 저장해 두었습니다.

  이미지 파일의 정보가 담긴 x_data와 순서대로 라벨을 저장해둔 y_data를 각 인덱스에 맞게 (x, y) 형식으로 저장했습니다. 이때 x_data는 모델에 입력하기 위해서 4차원의 데이터이어야 합니다. 예를 들어, train set의 x_data는 (39999, 3, 32, 32)의 shape을 가집니다. 3은 channel의 수를 말하는 것으로, RGB 컬러의 이미지임을 뜻합니다. 만약 흑백의 이미지라면 channel의 수는 1로 저장이 되어야 합니다. 그리고 y_data는 1차원의 리스트 형태로 저장했고 100개의 클래스이므로 각 문자형으로 저장되어 있던 클래스명을 임의로 0~99까지의 정수형으로 아래의 그림과 같이 변환했습니다.

```
Class to Label Mapping:
{'apple': 0, 'aquarium_fish': 1, 'baby': 2, 'bear': 3, 'beaver': 4, 'bed': 5, 'bee': 6,
'beetle': 7, 'bicycle': 8, 'bottle': 9, 'bowl': 10, 'boy': 11, 'bridge': 12, 'bus': 13,
'butterfly': 14, 'camel': 15, 'can': 16, 'castle': 17, 'caterpillar': 18, 'cattle': 19,
'chair': 20, 'chimpanzee': 21, 'clock': 22, 'cloud': 23, 'cockroach': 24, 'couch': 25,
'crab': 26, 'crocodile': 27, 'cup': 28, 'dinosaur': 29, 'dolphin': 30, 'elephant': 31,
'flatfish': 32, 'forest': 33, 'fox': 34, 'girl': 35, 'hamster': 36, 'house': 37,
'kangaroo': 38, 'keyboard': 39, 'lamp': 40, 'lawn_mower': 41, 'leopard': 42, 'lion':
43, 'lizard': 44, 'lobster': 45, 'man': 46, 'maple_tree': 47, 'motorcycle': 48,
'mountain': 49, 'mouse': 50, 'mushroom': 51, 'oak_tree': 52, 'orange': 53, 'orchid':
54, 'otter': 55, 'palm_tree': 56, 'pear': 57, 'pickup_truck': 58, 'pine_tree': 59,
'plain': 60, 'plate': 61, 'poppy': 62, 'porcupine': 63, 'possum': 64, 'rabbit': 65,
'raccoon': 66, 'ray': 67, 'road': 68, 'rocket': 69, 'rose': 70, 'sea': 71, 'seal': 72,
'shark': 73, 'shrew': 74, 'skunk': 75, 'skyscraper': 76, 'snail': 77, 'snake': 78,
'spider': 79, 'squirrel': 80, 'streetcar': 81, 'sunflower': 82, 'sweet_pepper': 83,
'table': 84, 'tank': 85, 'telephone': 86, 'television': 87, 'tiger': 88, 'tractor': 89,
'train': 90, 'trout': 91, 'tulip': 92, 'turtle': 93, 'wardrobe': 94, 'whale': 95,
'willow_tree': 96, 'wolf': 97, 'woman': 98, 'worm': 99}
```

- 학습 방법: 모델 학습에 사용한 손실 함수는 LabelSmoothingLoss()를 직접 구현해 사용했습니다. 왜냐하면 noisy label을 갖고 있는 데이터셋을 사용해야 하기 때문에 Label Smoothing을 적용했습니다. 그리고 최적화 기법은 Adam을 사용했습니다. 그리고 오버피팅(overfitting)을 방지하기 위해서 5-fold cross-validation을 적용했습니다.
- 성능 평가: 모델의 성능 평가를 위해 정확도(Accuracy)와 손실(Loss) 지표를 사용했습니다. Train set과 validation set의 지표를 비교하며 과적합(overfitting/underfitting) 문제의 발생 여부를 판단할 수 있었습니다. 만약 validation set의 loss가 에폭 수가 증가할수록 train set의 loss보다 더 커진다면 오버피팅이 발생한 것으로 판단했고 이 문제를 해결하고자 노력했습니다.
- 하이퍼파라미터: 모델 구조와 학습에 사용된 하이퍼파라미터에 대해 설명하겠습니다. 최적화 기법은 Adam, 학습률(learning rate)은 0.0001, batch size는 32, epoch 수는 23으로 설정하여 학습을 진행했습니다. 기본적으로 학습률이 작을수록 batch size도 작은 것이 학습을 좀 더 안정적으로 할 수 있다고 합니다. 또한, 작은 학습률은 Local Minima에 빠지는 가능성을 줄여줄 수 있습니다. 결과적으로, 여러 번의 실험을 통해서 가장 좋은 성능을 내는 하이퍼파라미터로 설정했습니다.

2. Any specialty of this model

① 5-fold cross validation 기법 적용: 기존에는 k-fold cross validation 기법을 적용하지 않고 단지 train set과 validation set, test set으로만 나눠서 과적합 문제가 발생하는지 여부를 판단하였습니다. 즉, 에폭 수가 증가할수록 train set의 loss와 validation set의 loss를 비교해보면 어느 순간 validation loss가 증가하는 시점이 발생하면 이를 overfitting 문제가 발생했다고 판단했습니다. 이를 해결하기 위해 k-fold cross validation 기법을 적용했습니다. 결과적으로 오버피팅 문제는 해결되었고 해당 모델의 일반화 성능도 추정

할 수 있었습니다.

② Label Smoothing 적용: 해당 과제에서 주어진 데이터셋은 noisy label을 갖고 있는 100개의 클래스로 구성된 이미지 데이터셋입니다. 기존에는 다중 클래스 분류 모델에 사용하는 손실 함수는 주로 크로스 엔트로피 손실(Cross Entropy Loss) 함수가 많이 사용됩니다. 따라서 처음에는 CrossEntropyLoss 손실 함수를 사용했다가 성능이 잘 나오지 않아서 데이터셋의 특성을 고려해서 Label Smoothing을 적용한 손실 함수로 변경했습니다. Noisy label 데이터셋의 오류에 대한 더 큰 유연성을 제공하기 위해 Label Smoothing을 적용했고, 이는 정답 레이블 대신에 약간의 오차를 가지도록 함으로써 더 많은 정규화 효과를 얻을 수 있습니다.

③ He initialization 적용: 가중치 초기화 방법 중 ReLU(Rectified Linear Unit)와 같은 활성화 함수를 사용하는 신경망에서 효과적으로 동작하는 He 초기화를 사용했습니다. 이를 통해 모델의 성능을 더 올릴 수 있었습니다.

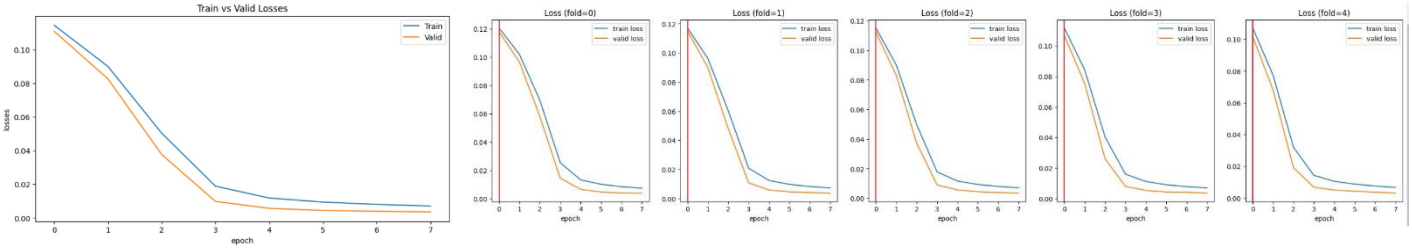
II. Results & Visualization

다음은 결과 분석과 시각화한 정확도와 손실 plot에 대해 설명하겠습니다. 먼저 아래의 2개의 캡처 사진 중 왼쪽 사진은 에폭 수 8 동안 학습하면서 출력한 train loss와 validation loss의 변화를 보여주고 있습니다. 적절하게 loss가 감소하는 것을 확인함으로써 학습이 잘 되고 있음을 알 수 있습니다. 또한, validation loss가 train loss보다 증가하는 구간이 없기 때문에 오버피팅이 발생하지 않는 것을 볼 수 있습니다. 그리고 오른쪽의 사진은 confusion matrix를 출력한 결과 창입니다. 클래스가 100개이기 때문에 100x100 행렬을 출력하기에 크기가 너무 커서 생략되었지만 바로 아래에 각 클래스별 정확도를 계산한 2차원 배열을 출력하도록 작성했습니다. 이를 통해서 class imbalance 문제가 발생했는지에 대한 여부를 판단할 수 있습니다. 대부분의 클래스가 약 60%의 정확도를 갖는 것을 확인함으로써 클래스 불균형 문제는 심각하지 않은 것으로 분석했습니다. 결과적으로 test dataset에서의 정확도는 64.298%, loss는 0.046으로 나왔습니다.

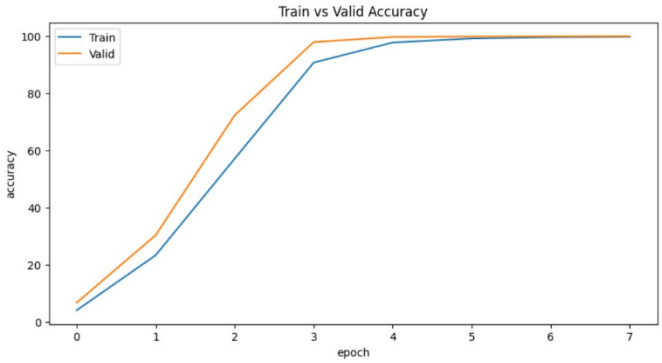
```
[4] fold=0, train loss: 0.025421, valid loss: 0.014510
[4] fold=1, train loss: 0.020840, valid loss: 0.010735
[4] fold=2, train loss: 0.017852, valid loss: 0.009080
[4] fold=3, train loss: 0.015959, valid loss: 0.007908
[4] fold=4, train loss: 0.014283, valid loss: 0.006839
[5] fold=0, train loss: 0.013298, valid loss: 0.006614
[5] fold=1, train loss: 0.012391, valid loss: 0.005906
[5] fold=2, train loss: 0.011666, valid loss: 0.005635
[5] fold=3, train loss: 0.011143, valid loss: 0.005284
[5] fold=4, train loss: 0.010535, valid loss: 0.005017
[6] fold=0, train loss: 0.010192, valid loss: 0.004729
[6] fold=1, train loss: 0.009736, valid loss: 0.004557
[6] fold=2, train loss: 0.009332, valid loss: 0.004427
[6] fold=3, train loss: 0.009030, valid loss: 0.004264
[6] fold=4, train loss: 0.008758, valid loss: 0.004267
[7] fold=0, train loss: 0.008486, valid loss: 0.004058
[7] fold=1, train loss: 0.008278, valid loss: 0.004094
[7] fold=2, train loss: 0.008048, valid loss: 0.003852
[7] fold=3, train loss: 0.007804, valid loss: 0.004047
[7] fold=4, train loss: 0.007581, valid loss: 0.003684
[8] fold=0, train loss: 0.007410, valid loss: 0.003951
[8] fold=1, train loss: 0.007273, valid loss: 0.003717
[8] fold=2, train loss: 0.007059, valid loss: 0.003484
[8] fold=3, train loss: 0.006911, valid loss: 0.003531
[8] fold=4, train loss: 0.006665, valid loss: 0.003096
Test Loss: 0.046 | Accuracy: 64.298
```

```
[0.77202073 0.73869347 0.50526316 0.47715736 0.55102041 0.67821782
 0.6763285  0.57837838 0.67010309 0.70149254 0.50531915 0.51041667
 0.64615385 0.65151515 0.58767773 0.58375635 0.62801932 0.77142857
 0.61458333 0.61214953 0.84951456 0.74611399 0.62176166 0.84183673
 0.75935829 0.55080214 0.51933702 0.51282051 0.75510204 0.6728972
 0.61702128 0.56435644 0.58080808 0.62135922 0.67692308 0.54395604
 0.61904762 0.48108108 0.595      0.657277   0.55801105 0.76923077
 0.67804878 0.71       0.4973822  0.60913706 0.48128342 0.70646766
 0.83756345 0.64179104 0.50246305 0.61       0.77777778 0.78846154
 0.70918367 0.51243781 0.8        0.67179487 0.68341709 0.60091743
 0.84729064 0.71014493 0.79812207 0.58252427 0.49758454 0.55660377
 0.57286432 0.57920792 0.88151659 0.75129534 0.64321608 0.78199052
 0.43137255 0.54545455 0.53365385 0.77040816 0.82352941 0.50738916
 0.51832461 0.55050505 0.50943396 0.665      0.89285714 0.63461538
 0.61928934 0.67924528 0.66666667 0.6937799  0.7038835  0.70560748
 0.60204082 0.73195876 0.50970874 0.57345972 0.76719577 0.62176166
 0.57142857 0.63546798 0.525      0.6372549 ]
```

아래의 그림은 train set과 validation set의 loss(손실)를 출력한 결과입니다. 오른쪽의 loss는 각 fold마다 출력한 loss plot입니다. 이를 통해서 noisy data를 갖고 있는 fold가 있는지 확인할 수 있습니다.



마지막으로, train set과 validation set의 accuracy(정확도)를 출력한 결과입니다. 안정적으로 학습이 되는 것을 볼 수 있고, overfitting 문제는 확실히 사라진 것을 알 수 있습니다. 하지만, 오히려 underfitting 문제가 의심되었습니다. 왜냐하면 validation set의 정확도가 100에 거의 가깝게 되었지만 결과적으로 test set의 정확도는 64.298%로 차이가 크게 나고, 아래의 plot에서 볼 수 있듯이 train accuracy와 validation accuracy의 차이의 폭이 커지는 것으로 보아 언더피팅(underfitting, 과소적합) 문제가 의심된다고 결과를 분석해보았습니다.



III. Discussion(토의)

해당 CNN 모델의 정확도를 더욱 높이기 위해서 아래와 같은 아이디어를 생각해보았습니다.

1. 더 깊은 네트워크 구조: 더 많은 합성곱 층과 완전 연결 층을 추가하여 모델을 더 깊게 만들 수 있습니다. 깊은 네트워크는 더 복잡한 특징을 학습할 수 있으며, 높은 수준의 추상적인 특성을 잘 포착할 수 있습니다.
2. 더 많은 필터 사용: 각 합성곱 층에 사용되는 필터의 개수를 늘릴 수 있습니다. 더 많은 필터를 사용하면 모델이 다양한 특징을 감지하는 데 도움이 됩니다.
3. Batch Normalization: 각 층의 입력을 정규화하여 학습과정을 안정화시킬 수 있습니

다. Batch Normalization은 학습 속도를 높이고, 과적합을 방지하며, 초기 가중치 설정에 덜 민감하게 만듭니다.

4. Data Augmentation: 데이터 증강을 통해 학습 데이터를 인위적으로 늘릴 수 있습니다. 회전, 이동, 반전 등의 변환을 적용하여 모델이 다양한 상황에서도 잘 동작하도록 도와줍니다.

5. Learning Rate Scheduling: 학습률을 동적으로 조정하는 Learning Rate Scheduling을 적용할 수 있습니다. 학습 초기에는 큰 학습률로 빠르게 학습하고, 점진적으로 작은 학습률로 수렴하도록 조정합니다.

6. 다양한 Optimizer 시도: 다양한 최적화 기법(Optimizer)을 사용하여 모델의 성능을 비교해볼 수 있습니다. Adam, RMSprop, SGD 등 다양한 옵티마이저를 사용하여 최적화 성능을 향상시킬 수 있습니다.

7. 앙상블: 여러 개의 모델을 학습하고, 그 예측 결과를 평균 또는 투표하여 최종 예측을 수행하는 앙상블 방법을 사용할 수 있습니다. 서로 다른 모델들의 다양성을 활용하여 성능을 향상시키는 효과가 있습니다.

8. 하이퍼파라미터 튜닝: 학습률, 배치 크기, 에포크 수 등과 같은 하이퍼파라미터들을 조정하여 최적의 설정을 찾아낼 수 있습니다.

이러한 아이디어들을 활용하여 CNN 모델의 성능을 더욱 향상시킬 수 있습니다. 모델의 성능을 평가하고 개선하기 위해 실험과 비교를 통해 가장 적합한 방법을 찾아내는 것이 중요하다고 생각했습니다.