Machine learning homework - 3

<ResNet using Tensorflow>

컴퓨터정보공학부

2020202013

고가연

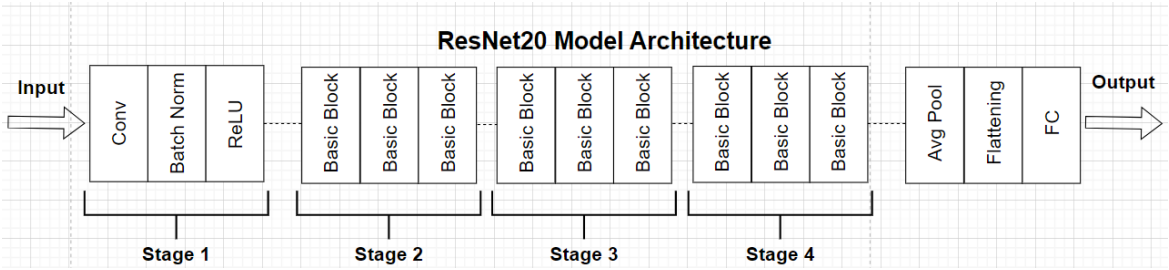
I. 데이터에 대한 설명

Kaggle에서 open dataset인 Food-101을 다운받아서 사용했다. Food-101은 음식 사진을 포함하는 대규모 데이터셋으로, 101개의 다양한 음식 범주를 가지고 있다. 각 범주에는 약 1,000개의 학습 이미지와 테스트 이미지가 포함되어 있으며, 총 이미지 수는 101,000장이다. 최상위 폴더 archive 폴더 하위에 images 폴더와 meta 폴더가 있고 images 폴더 밑에는 101개의 이미지 클래스 폴더들이 있다. 그 클래스 폴더에는 각 1,000개의 이미지들이 있다.

본 과제에서 구현한 ResNet20 모델에는 229 x 229 크기의 input으로 각 이미지가 들어가고 컬러 이미지이므로 RGB에 따라 3개의 채널로 입력된다. 따라서 해당 모델의 입력 데이터의 shape은 (none, 299, 299, 3)이다.

II. 네트워크 구조에 대한 설명

Tensorflow와 Keras로 구축한 ResNet20 모델의 네트워크 구조와 각 계층에 대한 설명을 하겠다. 먼저, 아래의 그림은 ResNet20 모델의 아키텍처를 설명하는 구조화한 그림(structural figure)과 table이다.



Layer name	Output size	20-layer
Conv1	299x299	3x3, 16, stride 1
Conv2_x	299x299	$\begin{pmatrix} 3\text{x}3, 16 \\ 3\text{x}3, 16 \end{pmatrix}$ x3
Conv3_x	150x150	$\begin{pmatrix} 3\text{x}3, 32 \\ 3\text{x}3, 32 \end{pmatrix}$ x3
Conv4_x	75x75	$\begin{pmatrix} 3\text{x}3, 64 \\ 3\text{x}3, 64 \end{pmatrix}$ x3
	1x1	Average pool, 1000-d fc, softmax

해당 네트워크의 전체적인 구조와 각 block에 대해 설명하겠다. 먼저, 입력 데이터의 shape은 (none, 299, 299, 3)이다. 입력 데이터는 크기가 299x299인 컬러 이미지라는 의미이다. 이때 none은 batch size를 말하고 이는 뒤의 소스코드 설명 부분에서 자세히 설명하겠다. 이러한 input 이미지가 모델에 들어가서 첫 번째 단계로 conv1 layer에 들어간다. 이는 합성곱 레이어와, Batch Norm(BN), ReLU(활성화 함수)를 차례로 거친다. 이때 3x3 kernel size와 16개의 filter, stride 1을 사용한다. 두 번째 단계로 conv2_x layer에서는 Basic Block에 들어간다. Basic Block은 ResNet20 모델에서 Residual Block이라고도 불리는 핵심 구성 요소이다. 이 구조는 네트워크의 깊이가 깊어질수록 gradient 소실 문제를 완화하고, 모델의 성능을 향상시키는 데 도움을 준다.

Basic Block은 두 개의 컨볼루션(Convolution) 계층으로 구성된다. 각 컨볼루션 계층 다음에는 배치 정규화(Batch Normalization)과 활성화 함수(ReLU)가 적용된다. 이를 통해 비선형성을 도입하고, 학습을 안정화시키며, 특징 추출을 강화한다. 또한, 스킵 연결(skip connection)도 사용됩니다. 스킵 연결은 입력 tensor를 바로 출력에 더해주는 방식으로, 학습 도중 정보의 손실을 최소화하고 gradient 흐름을 원활하게 한다. 이를 통해 네트워크가 더 깊어져도 성능 저하 없이 정보를 전달할 수 있다.

이러한 Basic Block을 3번 거치도록 구성했다. 즉, 3x3 크기의 kernel과, 16개의 filter를 사용한 컨볼루션 계층을 2번 거치는 Basic Block을 stage 2에서 3번 거친다. 결과적으로 여태까지 7개의 convolution 계층을 거친 것이다. 그 다음 세 번째 단계에서도 마찬가지로 Basic Block을 3번 거치도록 구성했다. 이때 3x3 크기의 kernel과 32개의 filter를 사용했다. 따라서 총 13개의 convolution 계층을 거친 것을 알 수 있다. 그리고 마지막 단계에서도 Basic Block을 3번 거치고 3x3 크기의 kernel과 64개의 filter를 사용한다. 여기까지 총 19개의 convolution 계층을 거친 것이다. 마지막으로 flattening layer로, 데이터를 1차원으로 만들어주는 계층을 거쳐야 한다. 따라서 총 20개의 layer를 갖는 ResNet20 model의 구조를 확인해보았다.

위의 그림을 참고하여 ResNet20 모델 구축에 사용된 layer에 대해 자세히 설명하겠다. 먼저, convolution layer(Conv2D)은 2D 컨볼루션 연산을 수행하여 입력 tensor에 필터를 적용하여 특징 맵을 생성하는 기능을 수행한다. filters: 출력 특징 맵의 차원(깊이)을 결정합니다. 해당 함수의 인자로는 kernel_size, strides, padding, kernel_initializer가 있다. Kernel_size는 3x3 필터의 크기를 지정하였고,

strides는 1로 설정하여 필터가 이동하는 보폭을 지정했다. 그리고 padding은 경계 처리 방법을 결정하는데, 'same'으로 설정하여 출력 특징 맵의 크기가 입력과 동일하게 유지되도록 했다. 마지막으로 kernel_initializer는 필터의 초기화 방법을 지정하는 인자로, 'he_normal'을 사용했고 이는 He 초기화 방법을 사용한다.

두 번째로 BatchNormalization(BN)이 있다. 이는 입력 데이터를 정규화하여 안정적인 학습을 할 수 있게 도와주는 기능을 수행한다. 항상 Convolution layer 뒤에 쓰이도록 한다. 또한, 입력 데이터의 평균과 분산을 이용하여 정규화하고, 정규화된 데이터에 scale과 이동 파라미터를 적용해 스케일 조정하는 역할을 한다. 이를 통해서 학습이 더욱 안정화되고, 학습 속도가 향상되는 효과를 얻을 수 있다.

세 번째로 Activation을 사용했다. 활성화 함수를 적용해서 비선형성을 추가하기 위함이다. 활성화 함수 중 ReLU(Rectified Linear Unit)을 사용했고, 이는 음수를 0으로 양수는 그대로 출력하는 동작을 수행한다.

네 번째로 Add가 있다. 이는 입력 tensor와 stride가 1보다 큰 경우의 convolution block의 출력과 skip connection을 합치는 역할을 한다. 즉, 입력 tensor와 skip connection의 값을 더하여 출력을 생성하는 동작을 수행한다.

다섯 번째로, AveragePooling2D가 있다. 평균 풀링 연산을 수행하여 입력 특징 맵을 다운샘플링(크기를 줄이기)한다. 이때 pool_size 인자는 풀링 윈도우의 크기를 지정하는 변수로, 8x8로 설정하였다.

여섯 번째로, Flatten이 존재한다. 역할은 다차원 배열을 1차원으로 평탄화한다.

마지막으로, Dense는 완전 연결 계층(full-connected layer)을 생성하는 역할을 수행하여 출력 클래스의 개수(101)를 num_classes 인자를 통해 결정할 수 있다. 그리고 kernel_regularizer 인자를 통해 가중치 정규화를 적용하고 activation 인자를 통해 'softmax'라는 활성화 함수를 지정하였다. 소프트맥스 함수를 사용해 확률 분포를 생성한다.

III. 소스코드에 대한 설명

아래는 Food-101 이미지 데이터를 101가지 클래스로 분류하는 ResNet20 모델을 구현한 소스코드에 대한 전반적인 과정에 대해 설명하고 있다.

1. 데이터 준비: prepare_data 함수

-> 학습, 검증, 테스트 데이터를 구분하여 복사한다.

-> train.txt, test.txt 파일을 읽어서 이미지 파일의 경로를 가져온다.

-> 클래스 별로 이미지 파일들을 그에 해당하는 폴더로 복사한다.

-> 이미지 파일의 개수를 카운트한다.

2. ResNet20 모델 정의: resnet20 함수

-> 입력 이미지의 크기와 클래스 수를 인자로 받는다.

-> 모델의 입력 레이어를 정의한다.

-> Conv2D 레이어와 BatchNormalization, Activation 등을 사용하여 기본 블록인 basic_block을 구성한다.

-> 기본 블록들을 쌓아 ResNet20 모델을 구성한다.

-> AveragePooling2D, Flatten, Dense 레이어를 추가하여 최종 분류를 수행하는 모델을 생성한다.

3. 모델 컴파일과 학습: model.compile, model.fit

-> 학습에 필요한 하이퍼파라미터와 옵티마이저를 설정한다.

-> 모델을 컴파일하고, 학습 데이터와 검증 데이터를 사용하여 모델을 학습시킨다.

-> ModelCheckpoint 콜백을 사용하여 검증 정확도가 가장 높은 모델을 저장한다.

-> 학습된 모델을 저장한다.

4. 이미지 예측: predict_class 함수

-> 학습된 모델과 이미지 파일의 경로를 인자로 받는다.

-> 이미지 파일을 모델이 요구하는 입력 형식으로 변환한다.

-> 모델을 사용하여 이미지를 예측하고, 예측된 클래스를 출력한다.

5. 혼동 행렬: Confusion Matrix

-> 테스트 데이터에 대한 실제 레이블과 모델의 예측 레이블을 비교

하여 혼동 행렬을 생성한다.

-> seaborn 라이브러리를 사용하여 혼동 행렬을 시각화한다.

6. 분류 보고서: classification_report, F1-score

-> 테스트 데이터에 대한 예측 결과를 기반으로 분류 보고서를 생성한다.

-> 각 클래스별 정밀도(precision), 재현율(recall), F1 스코어 등을 출력한다.

7. 정확도 계산: accuracy_score

-> 테스트 데이터에 대한 예측 결과와 실제 레이블을 비교하여 정확도를 계산한다.

위의 과정에서 forward(순전파) 과정과 back propagation(역전파) 과정에 대해 설명하겠다. 먼저, forward 과정은 입력 데이터를 모델에 주입하여 예측 값을 계산한다. 그러면 각 레이어의 가중치와 편향을 사용하여 계산된 예측 값이 생성된다. 그리고 손실 함수를 사용하여 예측 값과 실제 값 사이의 오차를 계산한다. 반면 back propagation 과정은 말그대로 손실 함수에서부터 역순으로 각 레이어를 거쳐 가중치를 업데이트한다. 그리고 손실 함수의 결과를 최소화하기 위해 각 레이어의 가중치와 편향을 조정한다. 경사 하강법(Gradient Descent) 알고리즘을 사용하여 가중치를 업데이트한다. 경사 하강법은 손실 함수의 기울기(gradient)를 사용하여 가중치를 조정하는 역할을 한다. 따라서, 해당 소스 코드에서는 model.compile 함수에 의해 손실 함수와 최적화 방법이 설정되었으며, model.fit 함수를 호출하여 학습 데이터와 검증 데이터로 모델을 학습시킨다. model.fit 함수는 주어진 데이터를 사용하여 순전파와 역전파를 자동으로 수행하여 모델의 가중치를 업데이트한다.

predict_class 함수를 통해 이미지를 예측하는 부분에서 forwarding이 수행된다. 예시 이미지는 다음과 같은 과정을 거쳐 변환됩니다:

1. 이미지 로드:

-> tf.keras.utils.load_img 함수를 사용하여 이미지 파일을 로드한다.

-> 로드된 이미지는 RGB 형식의 픽셀 값으로 표현된다.

-> 이미지의 크기는 target_size=(img_width, img_height)로 설정된 크기로 조정된다.

2. 이미지 변환:

-> tf.keras.utils.img_to_array 함수를 사용하여 이미지를 넘파이 배열로 변환한다.

-> 변환된 이미지는 0에서 255 사이의 값으로 표준화되지 않은 원시 RGB 픽셀 값을 가지게 된다.

3. 이미지 스케일링:

-> img /= 255.를 사용하여 이미지의 모든 픽셀 값을 0과 1 사이로 스케일링한다.

-> 이렇게 함으로써 모델은 0과 1 사이의 값을 입력으로 받게 된다.

4. 예측:

-> model.predict 함수를 사용하여 이미지를 예측한다.

-> 모델은 입력 이미지를 받아 클래스별 예측 확률을 출력한다.

-> np.argmax 함수를 사용하여 예측 확률이 가장 높은 클래스의 인덱스를 얻는다.

5. 결과 출력:

-> 예측된 클래스의 인덱스를 사용하여 food_list에서 식품 이름을 가져온다.

-> 결과 이미지와 예측된 식품 이름을 함께 시각화하여 출력한다.

따라서, forwarding 단계에서는 이미지를 로드하고 변환한 후 모델에 주입하여 예측을 수행하며, 최종 예측 결과를 반환한다.

다음은 기존의 CNN 코드에서 변경한 점과 ResNet20 모델의 하이퍼파라미터에 대한 설명을 하겠다. 기존의 CNN 코드에서는 동일하게 Food-101 dataset을 사용했지만 train(80%)과 validation set(10%)으로만 나눠서 모델을 평가했다. 하지만 해당 과제에서 구현한 ResNet20 모델은 train(80%), validation(10%), test(10%)로 Dataset을 나누어서 사용했다. 이때 입력한 이미지 크기는 동일하게 299x299

이다. 또한, 기존의 CNN 코드에서는 MobileNetV2라는 CNN 모델을 사용해서 음식 이미지 데이터들을 101가지 클래스로 분류하였고 나는 이 모델 부분을 변경해서 ResNet20 모델을 구축했다. 그리고 batch size는 20에서 40으로, epoch 수는 10에서 50으로 변경했다. Loss function(손실함수)는 그대로 categorical_crossentropy (다중 분류 손실함수)를 사용했다. 마지막으로 Optimization function(최적화 함수)는 SGD에서 Adam으로 변경했다. 아래는

1. 배치 크기 변경: 20에서 40으로 변경

   역할 및 기능: 배치 크기는 한 번에 처리되는 샘플의 수를 의미합니다. 작은 배치 크기는 메모리 요구량을 줄이고 gradient의 정확도를 높이는 장점이 있다.

   수정한 이유: 배치 크기를 증가시킴으로써 한 번에 더 많은 샘플을 처리하고, 더 많은 파라미터 업데이트를 수행할 수 있다. 이는 학습 속도를 향상시킬 수 있다. 또한, 배치 크기를 적절히 조정하면 모델의 일반화 성능을 향상시킬 수 있다.

2. 에폭 수 변경: 10에서 50으로 변경

   역할 및 기능: 에폭은 전체 데이터셋을 한 번 학습하는 단위이다. 학습을 더 많이 할수록 모델이 데이터를 더 잘 이해하고 학습할 수 있다.

   수정한 이유: 에폭 수를 증가시킴으로써 모델이 데이터를 더 많이 보고 학습할 수 있다. 이는 모델의 성능을 향상시킬 수 있다. 또한, 에폭 수를 늘리면 모델이 더 오랫동안 학습할 수 있기 때문에 더 복잡한 패턴을 학습할 수 있을 것으로 기대할 수 있다.

3. 손실 함수: categorical_crossentropy (다중 분류 손실 함수)

   역할 및 기능: 손실 함수는 모델이 예측한 값과 실제 값 사이의 오차를 계산하는 함수이다. 다중 분류 작업에서 categorical_crossentropy는 일반적으로 사용되는 손실 함수이다.

   수정하지 않은 이유: 손실 함수는 모델의 학습과정에서 사용되는 중요한 요소이다. 이미지 분류 작업에서 categorical_crossentropy는 일반적으로 잘 동작하는 손실 함수로 알려져 있기 때문에 변경할 필요

성이 없다고 생각했다.

4. 최적화 함수: SGD에서 Adam으로 변경

역할 및 기능: 최적화 함수는 모델의 파라미터를 업데이트하는 데 사용되는 알고리즘이다. SGD와 Adam은 각각 다른 최적화 알고리즘이다.

수정한 이유: Adam은 SGD에 비해 더 빠르게 수렴할 수 있는 장점이 있다. Adam은 학습률을 조정하면서 그라디언트 업데이트를 수행하기 때문에 더 효율적인 학습이 가능하다. 따라서 Adam을 사용하여 모델의 학습 속도와 성능을 개선할 수 있다.

IV. 실행결과 & Plot

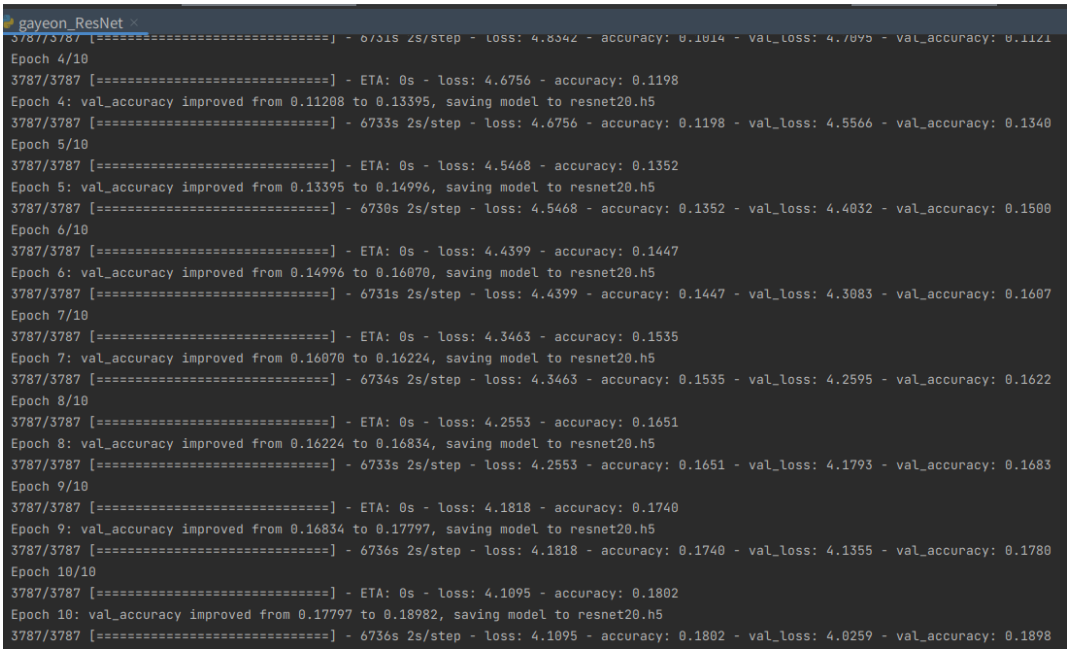
1. 실험결과

-> ResNet20 모델의 filter size를 16, 32, 64로 설정함.

-> 입력 데이터 shape = (none, 299, 299, 3)

-> 하이퍼파라미터: Optimizer(SGD), learning rate(0.0001), batch size(20), epoch(10)

-> 첫 번째 실행결과는 아래의 캡처 화면과 같다. 10 에폭을 다 학습했을 때 train accuracy는 18.02%이고, validation accuracy는 18.98%이다.



2. 실험결과

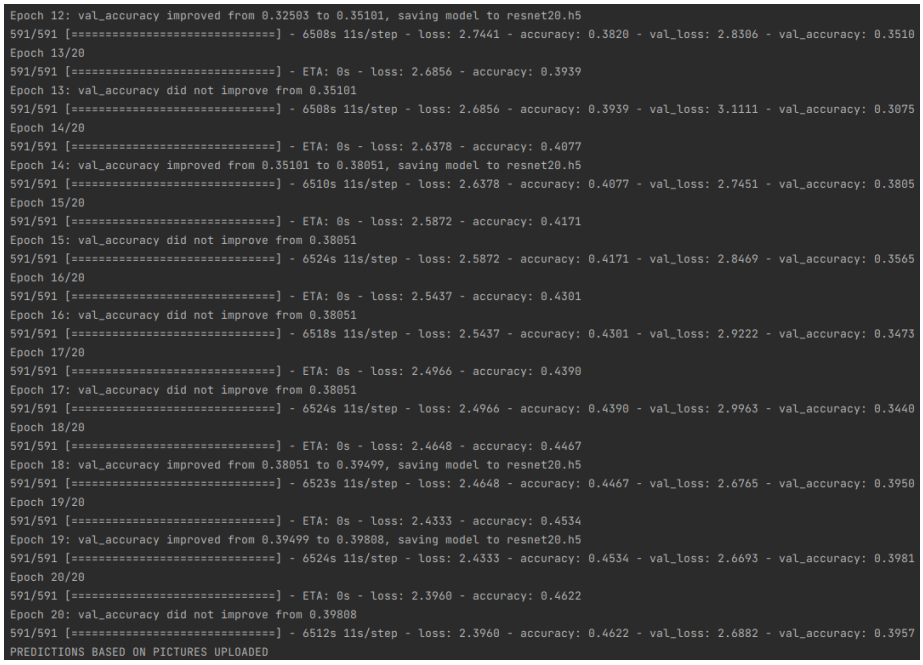
-> ResNet20 모델의 filter size를 16, 32, 64로 설정함.

-> 입력 데이터 shape = (none, 299, 299, 3)

-> 하이퍼파라미터: Optimizer(SGD), **learning rate(0.01), batch size(128), epoch(20)**

-> 첫 번째 실험에서 변경한 점은 학습률(lr)과 배치 크기, 에폭 수이다. 왜냐하면 첫 번째 실험에서 학습률이 0.0001로 너무 낮아서 학습하는 속도가 너무 느리고 시간이 많이 소요되는 문제점을 해결하고자 학습률을 0.01로 증가시켰다. 일반적으로 학습률과 배치 크기가 상호연관성이 있어서 학습률이 클수록 배치 크기를 키우는 것이 local minima에 빠질 확률이 적고 optimal minima에 수렴할 수 있다고 알려져 있다. 이를 통해서 배치 크기를 128로 증가시켜보았다. 또한, 에폭 수를 10에서 20으로 늘린 이유는 첫 번째 실험에서 loss는 계속 감소하는 추세를 보였고 accuracy는 계속 증가하는 추세를 보였기 때문에 10 에폭만으로는 학습이 완료되었다고 할 수 없었다.

-> 두 번째 실행결과는 아래의 캡처 화면과 같다. 20 에폭을 다 학습했을 때 train accuracy는 46.22%이고, validation accuracy는 39.57%이다. 첫 번째 실험과 결과를 비교해보았을 때, learning rate와 batch size를 키웠기 때문에 학습 속도가 훨씬 빨라져서 12에폭일 때 validation 정확도는 이미 35%에 도달한 것을 볼 수 있다. 하지만 12에폭일 때부터 overfitting(과적합)이 발생하는 것을 확인하였다. 오버피팅은 train set에 대해서만 학습이 잘 된 모델을 말하는데 쉽게 말해 validation 정확도가 train 정확도보다 낮은 경우를 말한다.



3. 실험결과

-> ResNet20 모델의 filter size를 16, 32, 64로 설정함.

-> 입력 데이터 shape = (none, 299, 299, 3)

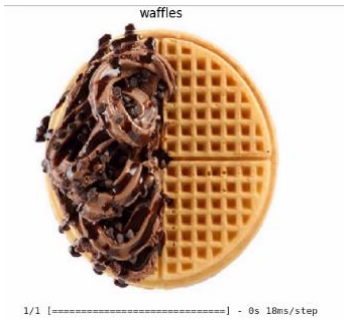
-> 하이퍼파라미터: **Optimizer(Adam), learning rate(0.001), batch size(64), epoch(40)**

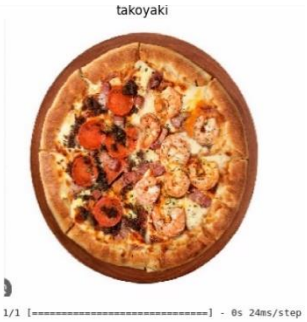
-> 두 번째 실험에서 바뀐 요소는 optimizer와 학습률(learning rate), 배치 크기, 그리고 에폭 수이다. 먼저, optimizer 중 Adam (Adaptive Moment Estimation)은 SGD (Stochastic Gradient Descent)의 변형된 형태인 최적화 알고리즘으로 당연히 모든 상황에서 Adam이 더 우수하다고 할 수는 없지만 두 개의 옵티마이저 중에 해당 데이터와 모델에 더 적합한 최적화 함수를 찾기 위해서 Adam으로 변경해보았다. 일반적으로 Adam은 학습률을 자동으로 조정하는 기능을 갖고 있으며, 모멘텀(momentum)을 사용하여 이전 gradient들의 이동 평균을 계산한다. 그리고 효율적인 메모리 사용을 한다고 알려져 있다. 그리고 이전 실험에서 learning rate는 0.01을 사용하고 batch size는 128을 사용했는데 확실히 학습 속도가 매우 빨라졌지만 overfitting이 발생하는 것을 발견했었다. 그래서 학습률과 배치 크기를 감소시켜서 조금 더 세세하게 학습할 수 있도록 조정했다. 그리고 에폭도 40으로 늘려서 정확도가 더 올

라갈 수 있도록 조정했다.

```
Epoch 30: val_accuracy improved from 0.50198 to 0.50785, saving model to resnet20.h5
1183/1183 [==============================] - 967s 817ms/step - loss: 2.1550 - accuracy: 0.5410 - val_loss: 2.2557 - val_accuracy: 0.5079
Epoch 31/40
1183/1183 [==============================] - ETA: 0s - loss: 2.1386 - accuracy: 0.5416
Epoch 31: val_accuracy did not improve from 0.50785
1183/1183 [==============================] - 963s 814ms/step - loss: 2.1386 - accuracy: 0.5416 - val_loss: 2.5847 - val_accuracy: 0.4418
Epoch 32/40
1183/1183 [==============================] - ETA: 0s - loss: 2.1241 - accuracy: 0.5468
Epoch 32: val_accuracy did not improve from 0.50785
1183/1183 [==============================] - 964s 815ms/step - loss: 2.1241 - accuracy: 0.5468 - val_loss: 2.5282 - val_accuracy: 0.4525
Epoch 33/40
1183/1183 [==============================] - ETA: 0s - loss: 2.1097 - accuracy: 0.5494
Epoch 33: val_accuracy did not improve from 0.50785
1183/1183 [==============================] - 960s 811ms/step - loss: 2.1097 - accuracy: 0.5494 - val_loss: 2.3348 - val_accuracy: 0.4926
Epoch 34/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0847 - accuracy: 0.5537
Epoch 34: val_accuracy did not improve from 0.50785
1183/1183 [==============================] - 961s 812ms/step - loss: 2.0847 - accuracy: 0.5537 - val_loss: 2.3786 - val_accuracy: 0.4823
Epoch 35/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0772 - accuracy: 0.5543
Epoch 35: val_accuracy did not improve from 0.50785
1183/1183 [==============================] - 959s 810ms/step - loss: 2.0772 - accuracy: 0.5543 - val_loss: 3.0569 - val_accuracy: 0.4073
Epoch 36/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0553 - accuracy: 0.5590
Epoch 36: val_accuracy improved from 0.50785 to 0.51721, saving model to resnet20.h5
1183/1183 [==============================] - 962s 813ms/step - loss: 2.0553 - accuracy: 0.5590 - val_loss: 2.2282 - val_accuracy: 0.5172
Epoch 37/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0454 - accuracy: 0.5618
Epoch 37: val_accuracy did not improve from 0.51721
1183/1183 [==============================] - 963s 814ms/step - loss: 2.0454 - accuracy: 0.5618 - val_loss: 2.3350 - val_accuracy: 0.4934
Epoch 38/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0312 - accuracy: 0.5620
Epoch 38: val_accuracy did not improve from 0.51721
1183/1183 [==============================] - 959s 811ms/step - loss: 2.0312 - accuracy: 0.5620 - val_loss: 2.2734 - val_accuracy: 0.4983
Epoch 39/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0151 - accuracy: 0.5683
Epoch 39: val_accuracy did not improve from 0.51721
1183/1183 [==============================] - 965s 816ms/step - loss: 2.0151 - accuracy: 0.5683 - val_loss: 2.4337 - val_accuracy: 0.4745
Epoch 40/40
1183/1183 [==============================] - ETA: 0s - loss: 2.0018 - accuracy: 0.5686
Epoch 40: val_accuracy did not improve from 0.51721
1183/1183 [==============================] - 963s 813ms/step - loss: 2.0018 - accuracy: 0.5686 - val_loss: 2.4375 - val_accuracy: 0.4700
PREDICTIONS BASED ON PICTURES UPLOADED
1/1 [==============================] - 0s 420ms/step
```

-> 아래의 사진들은 사전에 저장해 둔 사진을 불러와서 학습을 완료한 모델을 통해 음식 사진을 예측한 결과이다. 첫 번째 사진인 와플만 맞춘 것을 볼 수 있다.





-> 세 번째 실행결과는 40 에폭을 다 학습했을 때 train accuracy는 56.86%이고, validation accuracy는 51.72%이다. 두 번째 실험과 결과를

비교해보았을 때, 정확도가 약 10% 정도 향상되었지만 오버피팅은 여전히 해결되지 않았다. 정확도 향상의 원인을 생각해보았을 때, 에폭 수를 두배로 늘린 것과 적절한 학습률과 배치 크기가 있었던 것 같다.

4. 실험 결과

-> ResNet20 모델의 filter size를 16, 32, 64로 설정함.

-> 입력 데이터 shape = (none, 299, 299, 3)

-> 하이퍼파라미터: Optimizer(Adam), learning rate(0.001), **batch size(40), epoch(50)**

-> 세 번째 실험에서 바뀐 요소는 배치 크기, 그리고 에폭 수이다. 왜냐하면 세 번째 실험의 가장 큰 문제점은 오버피팅 문제와 낮은 정확도였다. 먼저 오버피팅 문제를 해결하기 위해서 배치 크기를 40으로 낮추었다. 배치 크기를 낮춰서 얻을 수 있는 효과는 더 정확한 그래디언트(gradient)를 추정할 수 있어서 오버피팅을 완화할 수 있다고 한다. 그리고 에폭 수를 50으로 늘린 이유는 이전 실험에서 정확도가 50%까지 밖에 나오지 않았기 때문에 학습을 더 많이 해서 정확도를 늘리고자 했다. 아래의 캡처 화면은 실행 결과화면이다.

```
Epoch 45/50
1893/1893 [==============================] - ETA: 0s - loss: 2.0071 - accuracy: 0.5588
Epoch 45: val_accuracy improved from 0.52929 to 0.53413, saving model to resnet20.h5
1893/1893 [==============================] - 967s 511ms/step - loss: 2.0071 - accuracy: 0.5588 - val_loss: 2.0833 - val_accuracy: 0.5341
Epoch 46/50
1893/1893 [==============================] - ETA: 0s - loss: 1.9973 - accuracy: 0.5616
Epoch 46: val_accuracy did not improve from 0.53413
1893/1893 [==============================] - 957s 505ms/step - loss: 1.9973 - accuracy: 0.5616 - val_loss: 2.1739 - val_accuracy: 0.5176
Epoch 47/50
1893/1893 [==============================] - ETA: 0s - loss: 1.9852 - accuracy: 0.5642
Epoch 47: val_accuracy did not improve from 0.53413
1893/1893 [==============================] - 970s 512ms/step - loss: 1.9852 - accuracy: 0.5642 - val_loss: 2.1374 - val_accuracy: 0.5241
Epoch 48/50
1893/1893 [==============================] - ETA: 0s - loss: 1.9774 - accuracy: 0.5648
Epoch 48: val_accuracy improved from 0.53413 to 0.54500, saving model to resnet20.h5
1893/1893 [==============================] - 971s 513ms/step - loss: 1.9774 - accuracy: 0.5648 - val_loss: 2.0608 - val_accuracy: 0.5450
Epoch 49/50
1893/1893 [==============================] - ETA: 0s - loss: 1.9627 - accuracy: 0.5685
Epoch 49: val_accuracy did not improve from 0.54500
1893/1893 [==============================] - 966s 510ms/step - loss: 1.9627 - accuracy: 0.5685 - val_loss: 2.2090 - val_accuracy: 0.5102
Epoch 50/50
1893/1893 [==============================] - ETA: 0s - loss: 1.9459 - accuracy: 0.5714
Epoch 50: val_accuracy did not improve from 0.54500
1893/1893 [==============================] - 965s 510ms/step - loss: 1.9459 - accuracy: 0.5714 - val_loss: 2.1665 - val_accuracy: 0.5163
PREDICTIONS BASED ON PICTURES UPLOADED
1/1 [==============================] - 0s 228ms/step
```

-> 네 번째 실행결과는 50 에폭을 다 학습했을 때 train accuracy는 57.14%이고, validation accuracy는 54.5%이다. 앞선 실험과 결과를 비교해보았을 때는 오버피팅(overfitting)이 완화되었고 정확도도 조금씩 더 올라간 것을 확인할 수 있다.

V. 참고문헌

https://velog.io/@vector13/Food101-%EB%8D%B0%EC%9D%B4%ED%84%B0%EC%85%8B%EC%9D%84-%EC%9D%B4%EC%9A%A9%ED%95%9C-%EC%9D%8C%EC%8B%9D-%EC%9D%B4%EB%AF%B8%EC%A7%80-%EB%B6%84%EB%A5%98%EA%B8%B0-%EB%A7%8C%EB%93%A4%EA%B8%B0

https://inhovation97.tistory.com/32