

의 차이를 한번 살펴보겠습니다. 다음 사진은 순서대로 100 회, 20 회, 15 회 순으로 학습량을 다르게 설정한 후 특정사진에 대한 테스트 결과물의 차이를 보여줍니다.

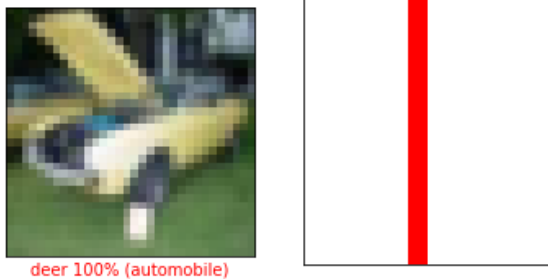


사진 2. 100 회(과대적합)

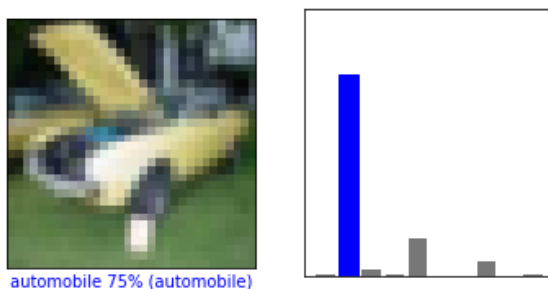


사진 3. 20 회

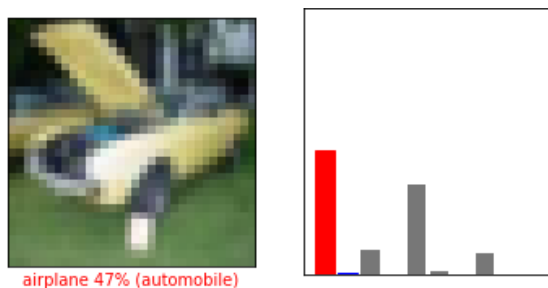


사진 4. 15 회(과소적합)

동일한 코드에서 학습량을 조절하여도 다른 결과를 보여주게 됩니다. 100 회 학습의 경우 너무 많은 학습을 하게 되면서 오차율이 매우 커지게 됩니다. 이를 과대적합이라고 합니다. 반면에 15 회의 학습을 했을 경우는 테스트에서 비록 틀린 결과를 보여주지만 그 오차율이 과대적합에 비해 낮음을 알 수 있습니다. 한편, 20 회의 학습을 했을 경우 오히려 올바른 값을 도출하고 있습니다.

표 1. 학습횟수에 따른 검증 정확성

학습횟수	검증 정확성
100 회(과대적합)	69.24%
20 회	71.56%
15 회(과소적합)	70.72%

물론 이러한 단적인 예시만으로 20 회의 학습량의 결과가 절대적으로 높은 것은 아닙니다. 검사 결과에 대한 정확성은 대략 $70\% \pm 1\%$ 로 오차범위 내에서 비슷한 수치를 보여줍니다. 따라서 일정한 학습횟수 이상으로 넘어갈 경우 검증 정확성에 큰 영향을 주지 않음을 알 수 있었습니다.

앞서 본 보고서의 요약에서도 기술했듯 최종적인 검증 정확성을 최대한 높여가는 것을 목적으로 추가적인 방법들을 제시하려고 합니다. 총 세 가지의 방법들을 추가하면서 설명하겠습니다.

첫번째로 신경망의 층을 두껍게 하는 방법입니다. 더 촘촘한 층을 쌓아서 학습을 하게 되면 학습되는 시간은 오래 걸리지만 그 만큼 더 결과적으로 좋은 성능을 보여줄 수 있습니다. 뒤쪽에서 더욱 더 자세하게 다루겠습니다.

두번째로는 현재 가지고 있는 CIFAR-10 DB의 사진들을 임의적으로 조절하여 데이터양을 증대하는 방법입니다. 비슷하지만 조금씩 다른 이미지들을 많이 학습시킴으로써 여러가지 상황에 맞추어 학습을 유도할 수 있습니다.

세번째로는 수리계획 혹은 학습 최적화라고 불리는 Optimizer 에 관한 내용입니다. 학습을 어떻게 진행할 것인지에 관한 내용이며, 여러가지 이론이 존재합니다. 따라서 제시된 이론을 그대로 적용해보기도 해보고 임의적으로 그 값을 조절하면서 수행하면 결론적으로 더 좋은 결과를 도출해낼 수 있습니다. 역시 뒤쪽에서 더 자세하게 다루겠습니다.

추가적으로 이미지를 활용한 기계 학습의 특성상 이미지의 각 픽셀 값들을 계산하는 과정이 매우 많기 때문에 CPU 를 활용한 학습속도와 GPU 를 활용한 학습속도는 매우 차이가 큼니다. GPU 는 행렬연산에 특화된 처리 장치로서 특히나 영상 데이터를 처리하는데 빠른 속도를 보입니다. 다음 링크는 이해를 돕기 위한 영상 링크입니다.

링크 1. CPU 와 GPU 의 차이

<https://www.youtube.com/watch?v=1BAZf3PsjWA>

해당 영상에서는 한 번에 이미지 데이터를 처리하여 결과물로 보여주는 것과 대비하여 한 개의 픽셀마다 계산해가는 CPU 와의 차이를 포인트 샷을 통해 직관적으로 설명한 영상입니다.

조금 더 깊은 신경망과 데이터 증대를 통해 학습량이 많아 짐에 따라 그 시간이 비약적으로 늘어났습니다. 따라서 GPU 를 활용하여 개발환경을 구축하였습니다.

표 2. 개발환경

CPU	Ryzen5 2600 3.9Ghz
-----	--------------------

RAM	DDR4 16GB 3200Mhz
GPU	NVIDIA GTX 1080
OS	Windows 10 64bit

제가 가지고 있는 그래픽 카드는 NVIDIA 사의 GTX 1080 입니다. 해당 그래픽 카드를 활용한 텐서플로우 개발환경 구축을 위해 CUDA 10.1 cuDNN 10.1 v7 을 설치 후 tensorflow-gpu 2.1.0 버전을 사용하였습니다.

2.1 신경망 층 증대 및 이미지 데이터 보강

합성곱(Convolution) 신경망의 구성은 특징 추출(Feature Extraction)과 분류기(Classifier)로 나뉩니다. 특징 추출은 합성곱 레이어와 풀링 레이어를 반복하여 구성하고 분류기는 누락 레이어(Dropout layer)와 밀집 레이어(Dense layer)로 구성됩니다.

합성곱 연산은 이미지의 특징을 주변 픽셀과 조합하여 대치하는 방식입니다. 즉, 이미지의 특징을 찾는 과정입니다. 따라서 전체적인 이미지의 형상을 유지한다는 특징이 있습니다. 이미지의 특징을 파악하면 필터를 통해 특징들을 세부적으로 나눠서 걸러냅니다. 그리고 합성곱 레이어에서는 다음과 같은 특징들을 쌓아 둔 형태로 존재합니다.







Operation	Filter	Filtered image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

사진 5. 필터를 통한 이미지 특징 추출

풀링 레이어는 추출된 특징의 픽셀들 중에서 중요한 값들만 기록합니다. 최대 풀링 레이어와 평균 풀링 레이어가 있습니다. 최대 풀링 레이어는 가장 큰 특징을 나타내는 하나의 값을 기록하며, 평균 풀링 레이어는 평균값을 내어 기록합니다. 이미지의 전체적인 경향을 조금 가져온다는 특징이 있습니다.

플래튼 레이어는 위 두개의 레이어를 반복적으로 거치면서 주요한 특징을 추출합니다.

누락 레이어는 과대적합을 방지하기 위한 레이어입니다. 학습을 위한 데이터에만 과도하게 적응하

는 것을 막기 위해 무작위로 학습된 데이터 중에서 일부를 누락시킵니다.

해당 이론을 바탕으로 다음과 같은 코드를 구성하여 학습을 진행시켰습니다.

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(input_shape=(32, 32, 3), kernel_size=(3, 3), filters=32, padding='same', activation='relu'))
model.add(tf.keras.layers.Conv2D(kernel_size=(3, 3), filters=64, padding='same', activation='relu'))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(rate=0.5))

model.add(tf.keras.layers.Conv2D(kernel_size=(3, 3), filters=128, padding='same', activation='relu'))
model.add(tf.keras.layers.Conv2D(kernel_size=(3, 3), filters=256, padding='valid', activation='relu'))
model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(rate=0.5))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=512, activation='relu'))
model.add(tf.keras.layers.Dropout(rate=0.5))
model.add(tf.keras.layers.Dense(units=256, activation='relu'))
model.add(tf.keras.layers.Dropout(rate=0.5))
model.add(tf.keras.layers.Dense(units=10, activation='softmax'))
```

사진 6. 신경망 층 증대 코드

앞서 진행했던 것에 비해 더 많은 합성곱 층과 누락 층을 가집니다. 결과적으로 앞서 진행했던 실험 결과보다 더 좋은 정확성을 가지게 됩니다.

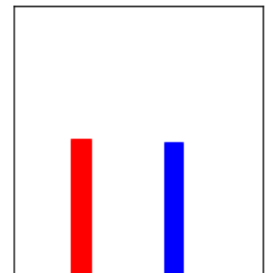


사진 7. 신경망 추가 전 20 회 학습

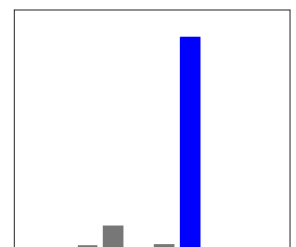


사진 8. 신경망 추가 후 20 회

표 3. 신경망 추가 전/후 정확성(각각 20 회 학습)

신경망 추가 전/후	검증 정확성
전	71.56%
후	76.3%

신경망을 깊게 구성하였으니 이제 가지고 있는 이미지를 적절히 편집 조절하여 다양한 상황의 이미지를 보강하여 추가적으로 학습시킵니다.

```
gen = ImageDataGenerator(rotation_range=20,|
                        shear_range=0.2,
                        width_shift_range=0.2,
                        height_shift_range=0.2,
                        horizontal_flip=True)
```

사진 8. 이미지 보강 코드 및 파라미터

이미지는 임의로 20 도 회전, 밀림정도 20%, 가로 세로 옮김 정도 20%, 사진 뒤집기 활성화 정도로 세팅하여 이미지를 보강하였습니다. 또한 보강된 이미지의 개수는 원본의 1.5 배; 즉 75000 장을 추가하여 총 학습데이터 125000 장을 학습시켰습니다.

동일한 학습횟수를 가지면서 이미지를 보강한 방법과 이전에 신경망 층만 깊게 구성했던 방법의 검증 정확성은 73.31%로 오히려 감소하였습니다. 그 이유는 학습되는 데이터양과 그 종류가 다양해지기 때문에 기존 20 회정도의 학습량으로는 충분하지 않다고 판단됩니다. 따라서 과대적합이 기존에 비해 쉽게 이루어지지 않을 것이라고 판단하고 학습량도 125 회로 기존에 비해 높게 설정하였습니다.

표 4. 이미지 데이터 보강 전/후 정확성

데이터 보강 전/후	검증 정확성
전(20 회 학습)	76.3%
후(125 회 학습)	79.53%

실제로 몇가지의 테스트 이미지를 시각화 하여 결과물들을 전체적으로 비교하면 의미 있는 변화를 관찰할 수 있었습니다.

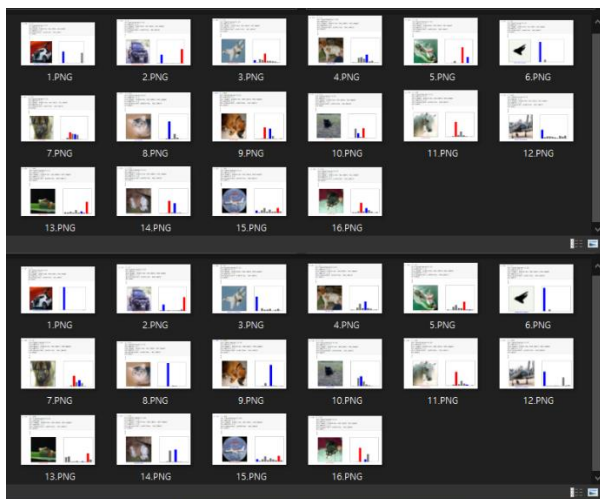


사진 9. 이미지 데이터 보강 전/후 비교

위쪽의 폴더가 보강 전, 아래쪽의 폴더가 보강 후입니다. 간단히 설명하면 이미지를 보강하기 전의 실험 결과 중에서 맞추지 못했던 사진들을 위주로 수집한 결과물과 보강후의 결과물들과 비교하였습니다. 파란색 그래프가 높으면 정답, 붉은 색 그래

프가 더 높으면 오답입니다. 즉, 16 개의 시료 중에서 맞추지 못했던 11 개에서 이미지 보강 후 7 개가량으로 줄어들었습니다.

그럼에도 불구하고 아직까지 검증 정확성이 80%를 넘기지는 못하고 있습니다. 따라서 보다 더 전문적이고 세부적인 구조를 조절해야 한다고 판단하여 몇가지의 자료들을 찾아보았습니다.

2.2 학습 최적화

데이터를 학습하는 과정에서 일부 값들을 수정합니다. 이 최적화를 하는 방식은 이미 많은 발전을 이루었고 여러가지 방법론들을 가지고 있습니다.

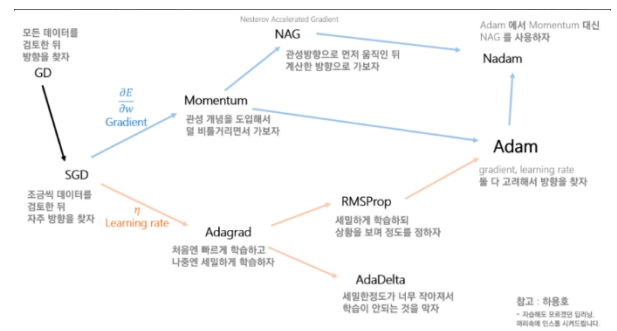


사진 10. 최적화 방법의 발전

현재까지 진행되었던 실험들은 전부 ‘Adam’기법을 사용하였습니다. 데이터의 종류와 특징들에 따라서 각 기법들이 다른 결과물들을 도출해 냅니다. 즉, 반드시 ‘Adam’만이 좋은 결과물을 낸다는 보장은 없습니다.

Cifar 10 DB 를 사용할 경우는 다음과 같이 기법별 검증 정확도 추적선으로 그래프로 도시하여 비교한 자료입니다.

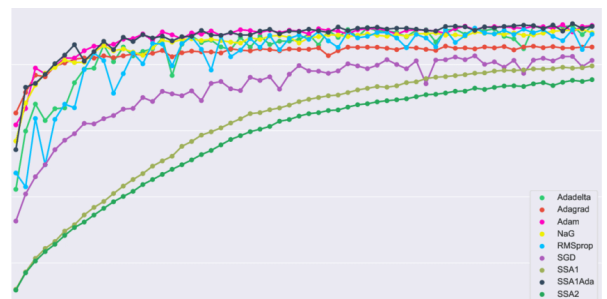


사진 11. 최적화 기법 별 정확도 추적선

Cifar 10 DB 를 활용할 경우에는 ‘Adam’ 혹은 ‘SSA1Ada’ 기법 이외에는 월등히 좋은 성능을 내는 기법은 보이지 않습니다. 따라서 현재까지 진행되었던 최적화 기법을 다른 기법으로 교체할 필요성은 없다고 생각합니다. 하지만 기법의 세부적인 학습률을 조절하여 조금 더 좋은 결과를 기대할 수 있습니다.

학습률에 대한 설명에 앞서 관련개념인 손실함수에 대해 잠시 짚고 넘어가겠습니다.

손실함수(loss)란, 확률로써 데이터를 분류하고 정확성을 도출하는 과정에서 나타나는 오차의 정도를 나타냅니다. 예를 들어 강아지 그림을 분류할 때 고양이가 96%, 강아지 4%로 분류가 되었다면 손실함수는 큰 것입니다. 다시 말해 훈련 손실과 검증 손실의 차이가 클 경우, 훈련 데이터 셋에만 적응을 과도하게 했다는 의미입니다.(과대적합).

학습률은 이 손실함수의 최저점에 수렴하는 방식을 조절하는 값입니다. 값이 너무 클 경우 손실함수에 최저점에 수렴하지 못하고 발산할 수도 있습니다. 반대로 너무 작을 경우에는 학습시간이 매우 오래 걸리며 최저점에 수렴을 하지 못할 수 있습니다.

추가적으로 Batch size(일괄적으로 처리되는 집단의 크기)를 수정하여 학습되는 방식을 조율할 수 있습니다. 이때 Batch size 를 크게 할 경우 학습률이 작아지는 것과 비슷한 효과를 냅니다. 일괄적으로 처리되는 집단의 크기가 커지게 되면 한 번에 처리해야 할 데이터의 양이 많아지게 됨으로 학습속도가 느려집니다.

그렇다면 학습률과 Batch size 가 서로 상관 관계를 가지고 있다면 적절히 두 값을 조절하는 것만으로도 다른 결과를 불러일으킬 수 있다는 판단이 됩니다. 관련 학술 자료들을 찾아보다 보니 다음과 같은 자료를 찾을 수 있었습니다.

Table 1. The results of the test AUC of the Adam optimizer.

Test AUC		
Batch size	Adam LR = 0.0001	Adam LR = 0.001
16	0.9677	0.9144
32	0.9636	0.9332
64	0.9616	0.9381
128	0.9567	0.9432
256	0.9585	0.9652

사진 12. 학습률과 Batch size 에 따른 정확성

해당 논문에서 가장 높은 정확성보이는 두가지 경우를 볼 수 있습니다. 큰 처리 집단 크기와 큰 학습률, 작은 처리 집단 크기와 낮은 학습률입니다. 표를 보게 되면 일단 낮은 학습률의 경우가 대체적으로 높은 정확성을 보입니다. 가장 높은 정확성을 보이는 처리 집단 크기 16, 학습률 0.0001 을 가지고 실험을 진행해보았습니다.

표 5. 학습률과 처리 집단 크기 조율에 따른 검증 정확성

학습량	검증 정확성
15 회	84.04%
125 회	85.88%

결과적으로 유의미하게 검증 정확성이 증가 했음을 확인할 수 있습니다. 하지만 학습결과에 대한 내용을 확인해보겠습니다.

```
In [36]: import pandas as pd
pd.DataFrame(history.history).plot()
```

Out [36]: <AxesSubplot:>

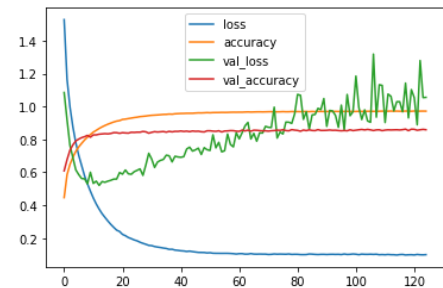


사진 13. 실험 내용 그래프

그래프를 확인해보면 검증 손실함수와 학습 손실함수의 그래프가 벌어지는 것을 확인할 수 있습니다. 즉, 과대적합이 발생한다는 결론을 내릴 수 있습니다.

앞서 Dropout(누락 함수)가 과대적합을 방지한다는 것을 알았기 때문에 이번에는 이 누락함수를 조절하여 성능 향상의 기대를 할 수 있습니다. 누락함수 층을 두텁게 하기 위해서는 기존의 신경망 층을 전체적으로 깊게 만들어야 합니다. 하나의 신경망 층을 구성하는 요소에서 하나의 층에 특징 추출과 분류기로 나뉘기 때문입니다.

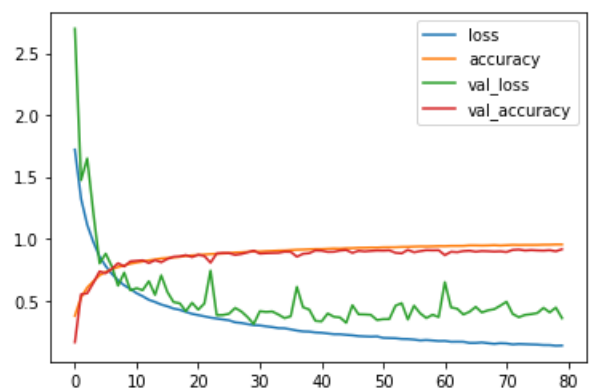


사진 14. 누락함수 추가 후 그래프

누락함수와 더불어 신경망을 깊게 만들어 전반적으로 검증함수가 훈련함수를 잘 따라가고 있음을 확인할 수 있습니다. 더불어 과대적합을 줄이기 위해 신경망 층을 더욱 깊게 만들었기 때문에 검증 정확성 역시 91%로써 본 보고서의 목표치에 다 달았습니다.

그럼에도 일부 불안정한 모습을 보입니다. 특히 20 회 초반 부분에서 크게 누락율과 정확성이 안 좋

게 나옵니다. 40 회 이상의 학습 구간부터는 검증 정확성이 가장 높게 측정되는 부분은 92%를 조금 넘기기도 하지만 89%까지도 내려가기도 합니다. 이때 손실함수를 학습이 진행됨에 따라 증가시켜주면 해당 문제점을 줄일 수 있습니다.

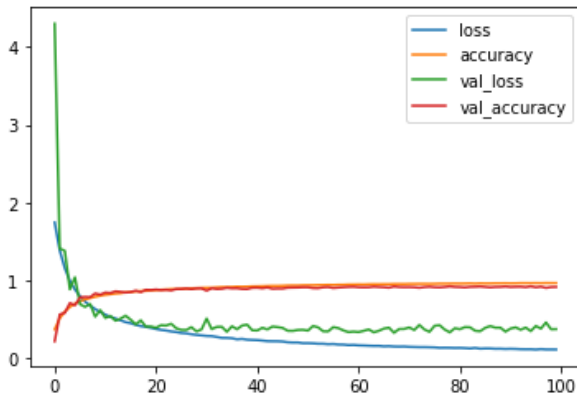


사진 15. 누락함수를 증가시키며 실험한 결과

최종 검증 정확성의 유의미한 증가 없이 91% 정도에 머물렀지만 대체적으로 검증함수들이 훈련함수를 굉장히 잘 따라가고 있음을 확인할 수 있습니다.

이때 반드시 누락율을 학습이 과정에서 점진적으로 증가시키는 것 만이 가장 높은 결과를 도출할지에 대한 의문이 생겼습니다. 과대적합을 방지하기 위해 일부러 학습된 데이터 일부를 누락시키는 것이라면 여러 갈래로 나뉘게 되는 분기점에서 좀 더 많은 양의 학습데이터를 누락시킨다면 더 좋은 결과가 나올 것이라는 가설을 세운 뒤 실험을 진행하였습니다.

```

In [9]: m1 = layers.Conv2D(64, 3, padding='same', activation='relu', name='B1_Conv_1')(input)
        m1 = layers.BatchNormalization(name='B1_Norm_1')(m1)
        m1 = layers.Conv2D(64, 3, padding='same', activation='relu', name='B1_Conv_2')(m1)
        m1 = layers.BatchNormalization(name='B1_Norm_2')(m1)
        m1 = layers.MaxPool2D(2, strides=2, name='B1_Pool')(m1)
        m1 = layers.Dropout(0.2, name='B1_Drop')(m1)

In [10]: m2 = layers.Conv2D(128, 3, padding='same', activation='relu', name='B2_Conv_1')(m1)
        m2 = layers.BatchNormalization(name='B2_Norm_1')(m2)
        m2 = layers.Conv2D(128, 3, dilation_rate=2, activation='relu', name='B2_Conv_2')(m2)
        m2 = layers.BatchNormalization(name='B2_Norm_2')(m2)
        m2 = layers.MaxPool2D(2, strides=2, name='B2_Pool')(m2)
        m2 = layers.Dropout(0.25, name='B2_Drop')(m2)

In [11]: m3 = layers.Conv2D(128, 3, padding='same', activation='relu', name='B3_Conv_1')(m2)
        m3 = layers.BatchNormalization(name='B3_Norm_1')(m3)
        m3 = layers.Conv2D(256, 3, padding='same', dilation_rate=2, activation='relu', name='B3_Conv_2')(m3)
        m3 = layers.BatchNormalization(name='B3_Norm_2')(m3)
        m3 = layers.MaxPool2D(2, strides=2, name='B3_Pool')(m3)
        m3 = layers.Dropout(0.3, name='B3_Drop')(m3)

        m4 = layers.Conv2D(128, 3, padding='same', activation='relu', name='B4_Conv_1')(m3)
        m4 = layers.BatchNormalization(name='B4_Norm_1')(m4)
        m4 = layers.Conv2D(256, 3, padding='same', activation='relu', name='B4_Conv_2')(m4)
        m4 = layers.add([m4, m3Pool], name='B4_Add')
        m4 = layers.BatchNormalization(name='B4_Norm_2')(m4)
        m4 = layers.Dropout(0.5, name='B4_Drop')(m4)
        m4 = layers.GlobalAveragePooling2D(name='B4_Global')(m4)

        m5 = layers.Conv2D(128, 3, dilation_rate=2, padding='same', activation='relu', name='B5_Conv_1')(m4)
        m5 = layers.BatchNormalization(name='B5_Norm_1')(m5)
        m5 = layers.Conv2D(256, 3, dilation_rate=2, padding='same', activation='relu', name='B5_Conv_2')(m5)
        m5 = layers.BatchNormalization(name='B5_Norm_2')(m5)
        m5 = layers.Conv2D(256, 3, padding='same', activation='relu', name='B5_Conv_3')(m5)
        m5 = layers.BatchNormalization(name='B5_Norm_3')(m5)
        m5 = layers.MaxPool2D(2, name='B5_Pool')(m5)
        m5 = layers.Dropout(0.4, name='B5_Drop')(m5)

        m6 = layers.Conv2D(128, 3, padding='same', activation='relu', name='B6_Conv_1')(m5)
        m6 = layers.BatchNormalization(name='B6_Norm_1')(m6)
        m6 = layers.Conv2D(256, 3, padding='same', dilation_rate=2, activation='relu', name='B6_Conv_2')(m6)
        m6 = layers.add([m6, m5Pool], name='B6_Add')
        m6 = layers.BatchNormalization(name='B6_Norm_2')(m6)
        m6 = layers.Dropout(0.3, name='B6_Drop')(m6)

        m7 = layers.Conv2D(128, 3, padding='same', activation='relu', name='B7_Conv_1')(m6)
        m7 = layers.BatchNormalization(name='B7_Norm_1')(m7)
        m7 = layers.Conv2D(256, 3, padding='same', activation='relu', name='B7_Conv_2')(m7)
        m7 = layers.add([m7, m6Pool], name='B7_Add')
        m7 = layers.BatchNormalization(name='B7_Norm_2')(m7)
        m7 = layers.Dropout(0.4, name='B7_Drop')(m7)

        m8 = layers.Conv2D(128, 3, padding='same', activation='relu', name='B8_Conv_1')(m7)
        m8 = layers.BatchNormalization(name='B8_Norm_1')(m8)
        m8 = layers.Conv2D(256, 3, padding='same', dilation_rate=2, activation='relu', name='B8_Conv_2')(m8)
        m8 = layers.add([m8, m7Pool], name='B8_Add')
        m8 = layers.BatchNormalization(name='B8_Norm_2')(m8)
        m8 = layers.Dropout(0.3, name='B8_Drop')(m8)

        m9 = layers.Conv2D(256, 3, activation='relu', name='B9_Conv_1')(m8)
        m9 = layers.BatchNormalization(name='B9_Norm_1')(m9)
        m9 = layers.Conv2D(256, 3, dilation_rate=2, activation='relu', name='B9_Conv_2')(m9)
        m9 = layers.add([m9, m8Pool], name='B9_Add')
        m9 = layers.BatchNormalization(name='B9_Norm_2')(m9)
        m9 = layers.Dropout(0.2, name='B9_Drop')(m9)

        m10 = layers.Conv2D(512, 3, padding='same', activation='relu', name='B10_Conv_1')(m9)
        m10 = layers.GlobalAveragePooling2D(name='B10_Pool')(m10)
        m10 = layers.BatchNormalization(name='B10_Norm')(m10)

```

사진 16. 누락함수 조정

따라서 누락율을 0.2->0.25->0.3->0.5->0.4->0.3-

>0.2 의 형식으로 늘였다가 다시 줄이는 형식을 사용했습니다.

3. 실험 결과 및 분석

연구 결과를 독자들이 이해하기 쉽게 다양한 대상에 대해 기존의 다른 방법들과의 성능도 객관적으로 비교하고 분석하기 바랍니다.

먼저 진행되었던 실험을 일련적으로 표로 정리하면 다음과 같습니다.

표 6. 추가되는 방법에 따른 검증 정확성

추가되는 방법/학습횟수	검증 정확성
기본 CNN/20 회	71.56%
신경망 추가/20 회	76.3%
데이터 보강/125 회	79.53%
학습 최적화/125 회	85.88%
더 깊은 신경망/80 회	91.48%
점진적증가 누락함수 /100 회	91.12% 92.04%(maximum)
누락함수 조정/150 회	93.01%

우선적으로 학습량이 계속해서 많아지는 이유는 데이터의 보강으로 인해 양이 증가하고, 신경망을 깊게 만들면서 과대학습을 방지하는 함수들을 많이 넣었기 때문입니다. 실제로 어느정도 학습수준 이상으로 넘어가게 되면 크게 증가하지 않는 영역에 들어오게 됩니다. 이때 학습결과물과 검증결과물을 그래프로 시각화하여 전체적으로 어떤 문제점이 발생하는지를 파악할 수 있습니다. 가장 좋은 결과물은 검증결과물이 학습결과를 잘 따라가는 것입니다. 다시 말해 학습데이터에 적응하지 않고, 학습데이터를 기반으로 일반화를 잘 진행하였다는 의미입니다.

앞서 실험 내용에서 초반 방법의 추가에 따른 예시 사진을 보여주어 그 부분은 생략하고 최종적으로 가장 높은 검증 정확성을 가지는 코드를 기준으로 결과 분석을 해보겠습니다.

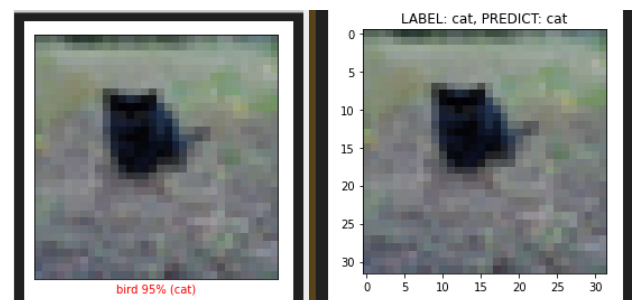


사진 16. 최종본과 비교(검증 정확성 = 우: 85%, 좌: 91%)

91%정도의 검증 정확성을 가지는 코드의 경우 대

부분의 사진에서 비교적 잘 맞추는 것을 확인할 수 있었습니다. 특히하게도 실험을 할 때마다 다른 결과물을 보이는 그림도 있었습니다.

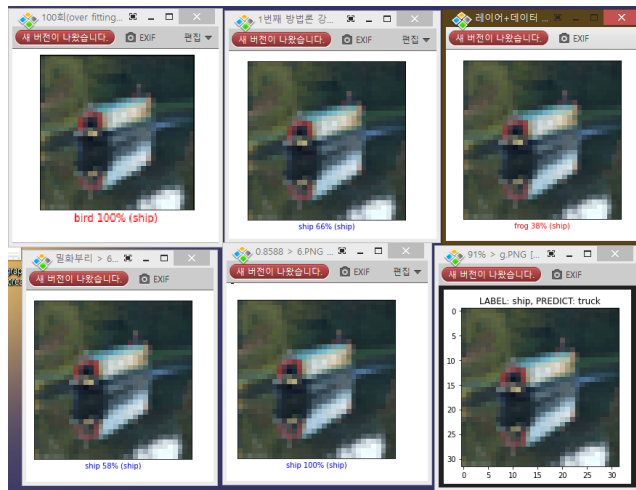


사진 17. 매 실험결과 다른 결과물(좌->우 방향으로 최신 방법론 적용)

반드시 검증 정확성이 높아진다고 맞추지 못했던 것을 맞추게 되는 것은 아니라는 것입니다. 각각의 방법에서 새, 트럭, 배, 개구리로 다양하게 분류됨을 확인할 수 있습니다. 사진의 모양이 특징을 추출하는데 있어서 물살이 날개처럼 보이기도 하고, 물에 반사된 모양이 얼핏 트럭처럼 보이기도 하기 때문에 분류에 다양한 결과물이 나왔을 것이라고 추측합니다. 이는 이미지 보강보다는 더욱 다양한 예시와 고화질의 사진을 사용한다면 해소될 문제로 보입니다. 이를 뒷받침해줄 예시를 보겠습니다.

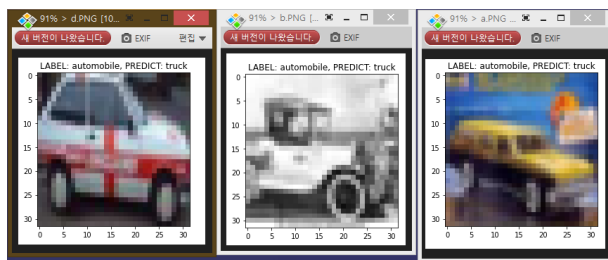


사진 18. 동일한 경우의 예측실패(검증 정확성 91%)

자동차 범주의 이미지를 트럭으로 분류한 경우입니다. 생각보다 비슷한 경우로 분류가 된 경우가 많이 있었습니다. 따라서 CIFAR 10 DB 에서 트럭에 대한 더 다양한 사진 혹은 고화질로 이미지를 보강하여 학습시킨다면 개선될 문제라고 판단됩니다. 덧붙여 두번째 사진의 경우 옛날 자동차로써 최근 차들에 비해 둔탁한 느낌이기 때문에 더욱 트럭으로 분류했을 것이라고 판단됩니다. 화질에 대한 아쉬움은 다음 사진에서도 보입니다.

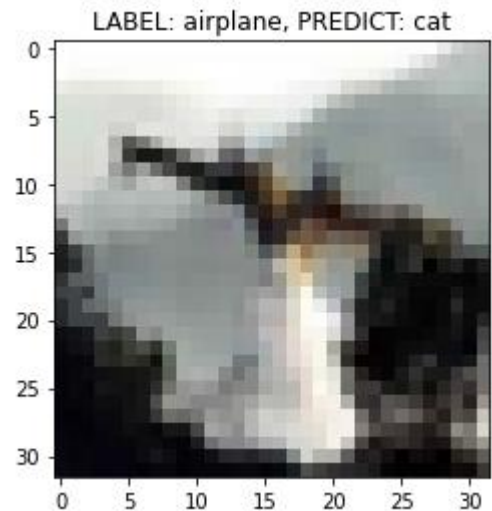


사진 19. 아쉬운 화질 예시(검증 정확성 91%)

처음 사진을 보는 사람의 입장에서 어떤 사진인지 감이 잘 잡히지 않습니다. 배경이나 주제가 되는 물체와 주변 환경이 제대로 식별이 되지 않습니다. 화소의 부족으로 인해 음영도 쉽게 판단이 되지 않아 주제가 되는 물체와 주변환경의 구별이 잘 되지 않아 고양이와 뒤의 물체를 합성으로 날개로 판단하여 비행기로 예측했을 것 같습니다.

반면에 완벽하게 아쉬웠던 부분도 존재했습니다.

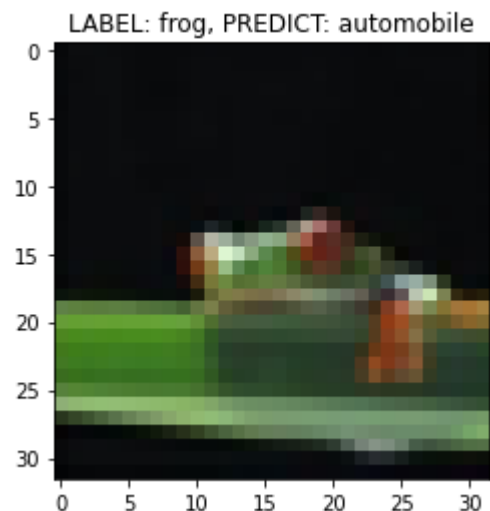


사진 20. 분류 실패 예시(검증 정확성 91%)

확실하게 개구리처럼 보임에도 자동차로 분류한 경우입니다. 이러한 부분은 학습과정의 방식을 조금 더 세밀하게 완성한다면 해소될 것으로 보이나 여러 방면에서 추가적인 실험을 했음에도 제대로 분리시키기가 힘들었습니다. 가장 아쉬운 부분입니다. CIFAR 10 DB 를 가지고 검증 정확성을 99%까지 올린 학술자료도 있기 때문에 충분히 돌파구가 있다고 판단됩니다.

4. 결론

본 보고서에서는 CIFAR 10 DB 를 CNN 을 활용하여 10가지의 각주로 분류하는 시스템을 만들었습니다. 기본적인 CNN 구조를 만든 후 새로운 방법을 도입하면서 각 방법별로 약 5%의 검증 정확성을 올려냈습니다.

표 6. 추가되는 방법에 따른 검증 정확성

추가되는 방법/학습횟수	검증 정확성
기본 CNN/20 회	71.56%
신경망 추가/20 회	76.3%
데이터 보강/125 회	79.53%
학습 최적화/125 회	85.88%
더 깊은 신경망/80 회	91.48%
점진적증가 누락함수 /100 회	91.12%
	92.04%(maximum)
누락함수 조정/150 회	93.01%

학습되는 과정의 신경망을 추가하고 학습 데이터를 보강 및 양을 증대하여 8%의 검증 정확성을 높였습니다. 학습의 과정에서 학습횟수가 증대함에 따라 과대적합을 피할 수 없었고, 이를 극복하기 위해 학습 최적화를 하여 과대적합을 최대한 피하면서 한 번에 학습되는 처리집단을 축소하여 보다 촘촘한 학습을 진행시켰습니다. 결과적으로 12%의 검증 정확성을 향상시켰습니다. 추가적으로 결과를 분석하면서 더욱 극복할 여지가 충분히 있어 보였고 여러 학술지에서도 99%의 검증 정확성을 가지는 결과물도 있었습니다. 그럼에도 데이터의 화질과 다양성에 아쉬움을 보이는 분석 결과도 존재하였습니다.

학습방법을 추가하고 수정하는 과정에서 학습량이 증가함에 따라 학습하는 시간이 비약적으로 증가합니다. 이를 극복하기 위해 GPU 를 활용하여 학습을 하게 되었고 결과적으로 CPU 대비 하여 동일한 학습프로그램을 기준으로 1 회 학습 당 약 72%의 시간을 단축시켰습니다. 기계학습이라는 개념이 등장한 이후 많은 연산을 한 번에 그리고 동시에 처리하는 시스템이 요구하는 하드웨어 시장이 더욱 개척되면서 GPU, MPU 와 같은 새로운 연산 처리 장치의 시대가 도래한 것입니다.

참고문헌

[1] <https://bskyvision.com/700>, [Anaconda+python]CIFAR-10 데이터셋으로 이미지 분류기 만들기 (컨볼루션 신경망).txt (2020-02-06), 코드자료 참고

[2] 도전적인 HarimKang, <https://davinci-ai.tistory.com/29>, 딥러닝(6)-CNN(Convolution Neural Network)-2020.2.24 22:12, CNN 의 구성 및 내용 참고, 사진 5. 인용

[3] <https://www.youtube.com/watch?v=1BAZf3PsjWA>, NVIDIA 의 과학시간 - GPU 와 CPU 의 차이, 링크 1. 출처

[4] REALSTUDY_NET, <https://www.youtube.com/watch?v=6oe01JBzFs>, [실용 AI 010]cifar10 CNN, 코드 분석 참조

[5] NeoWizard, <https://www.youtube.com/watch?v=pKZmmO32Fpc>, [TensorFlow 2.x 강의 17]CNN CIFAR 10 Example v2, 코드 분석 참조

[6] Neowizard, https://github.com/neowizard2018/neowizard/blob/master/TensorFlow2/TF_2_x_LEC_17_Example.ipynb, TF_2_x_LEC_17_Example.ipynb, 이미지 보강 파라미터 참고

[7] Neowizard, <https://blog.naver.com/PostView.naver?blogId=beyondlegend&logNo=222345469147&parentCategoryNo=&categoryNo=93&viewDate=&isShowPopularPosts=true&from=search>, CNN 성능향상 (CIFAR 10 정확도), 2021.5.10 20:48, 이미지 데이터 보강 코드 참고

[8] Gom Guard, <https://gomguard.tistory.com/187>, [딥러닝] 뉴럴 네트워크 Part.8 - 옵티마이저 (Optimizer), 2018.3.29, 옵티마이저 이론 및 발전 과정 참고, 사진 10. 인용

[9] ResearchGate, https://www.researchgate.net/figure/Accuracy-comparison-for-different-optimizers-on-test-CIFAR-10-dataset-using_fig6_332780505, New optimization algorithms for neural network training using operator splitting techniques, 2019.4, 옵티마이저 별 검증 정확도 추적성, 사진 11. 인용

[10] Inhovation, <https://inhovation97.tistory.com/32>, Learning rate & batch size best 조합 찾기(feat.논문 리뷰와 실험결과), 2021.3.31, 내용 및 논문 참조

[11] Ibrahem Kandel, Mauro Castelli, The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset, ICT Express 6 (2020) 312-315, 314 page, Adam LR 과 Batch size 의 상관 관계 참조, 사진 12 인용

[12] (<https://www.sciencedirect.com/science/article/pii/S2405959519303455#fig2>, 위 논문의 접근사이트)

[13] BioinformaticsAndMe, <https://bioinformaticsandme.tistory.com/130>, 학습률 (Learning rate), 2019.9.24, 학습률 뜻 참조