목차

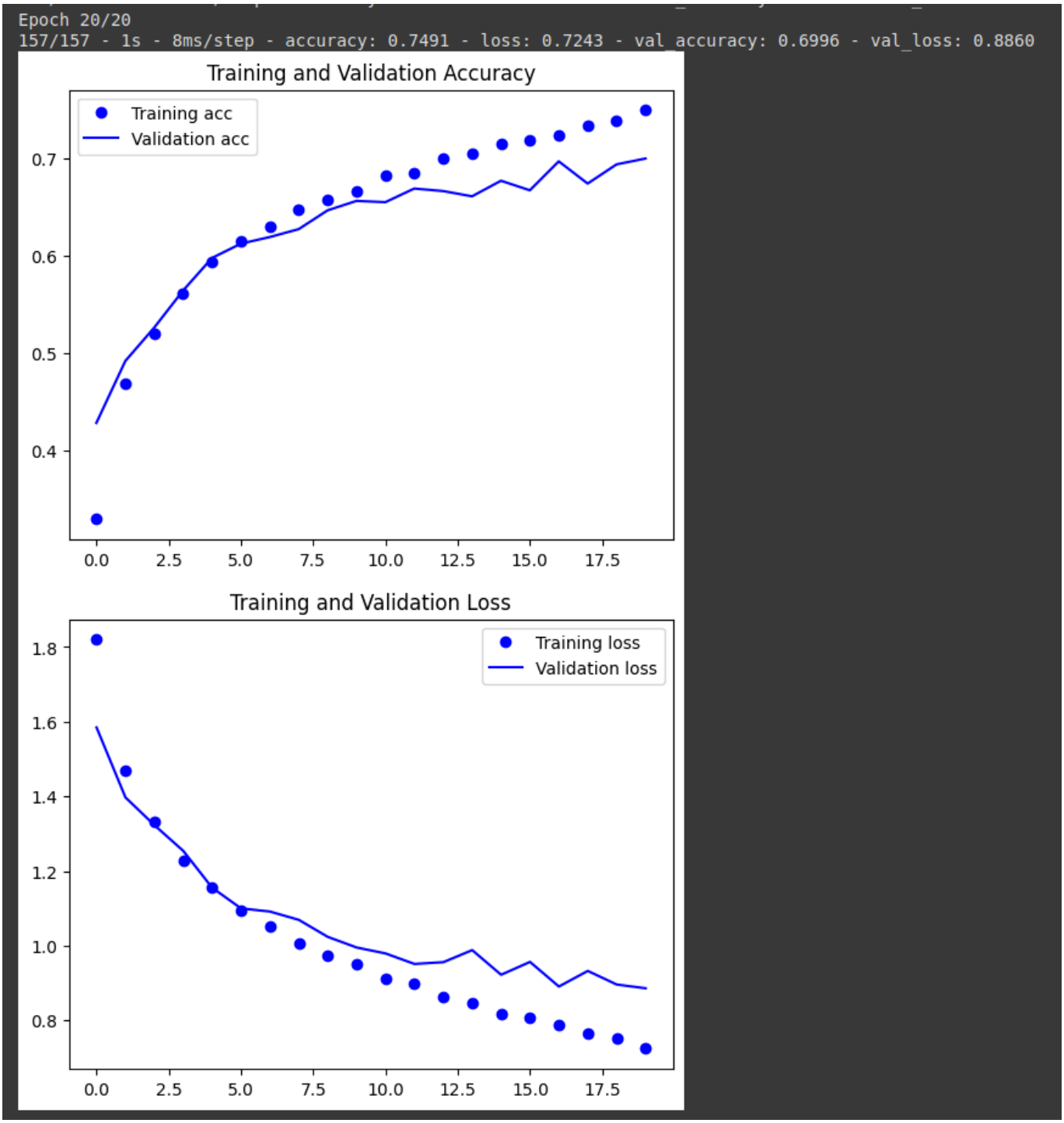
CIFAR10 - CNN을 이용한 학습

초기 Hyper Parameter 설정

- filters(필터 개수)
  - 1st `Conv2D`: 32
  - 2nd `Conv2D`: 64
  - 3rd `Conv2D`: 64
    - 필터 수가 더 다양한 특징 학습 가능
    - 대신 모델의 복잡도와 계산량 증가
- kernel_size
  - 모든 Conv2D 층: 3X3 사이즈
- Pool-size
  - 첫번째 MaxPool2D(2,2)
  - 두번째 MaxPool2D(2,2)
    - (2,2) pooling은 feature 맵의 픽셀 수를 1/4로 줄임
- units(뉴런 개수)
  - 1st `Dense`층: 64
  - 2nd `Dense`층: 10
  - FC layer의 뉴런 수 정의
  - 출력층의 뉴런 수는 분류 Class의 개수와 일치해야함
- activation
  - `Conv2D`, `Dense`: ReLU 사용

    - 마지막 `Dense`(출력층): softmax 사용
- Optimizer
    - rmsprop
- Loss function
    - `categorical_crossentropy`
- epochs
    - 20으로 세팅
- batch size
    - 256
- validation split
    - 0.2
    - 학습 데이터셋 일부를 떼어내어 검증용으로 사용

기존 성능



- 검증 정확도와 훈련 정확도의 차이가 크지 않다
  - 과적합도가 낮다
- 검증 및 훈련 손실율도 1%이하로 적은 것을 확인 가능
- Validation accuracy가 0.699수준으로 MLP에 비해 크게 향상된 것을 확인할 수 있다

Convolution Layer의 필터 개수 차이에 따른 성능 변화

> 첫번째 Conv2D 층의 필터 개수만 바꿈

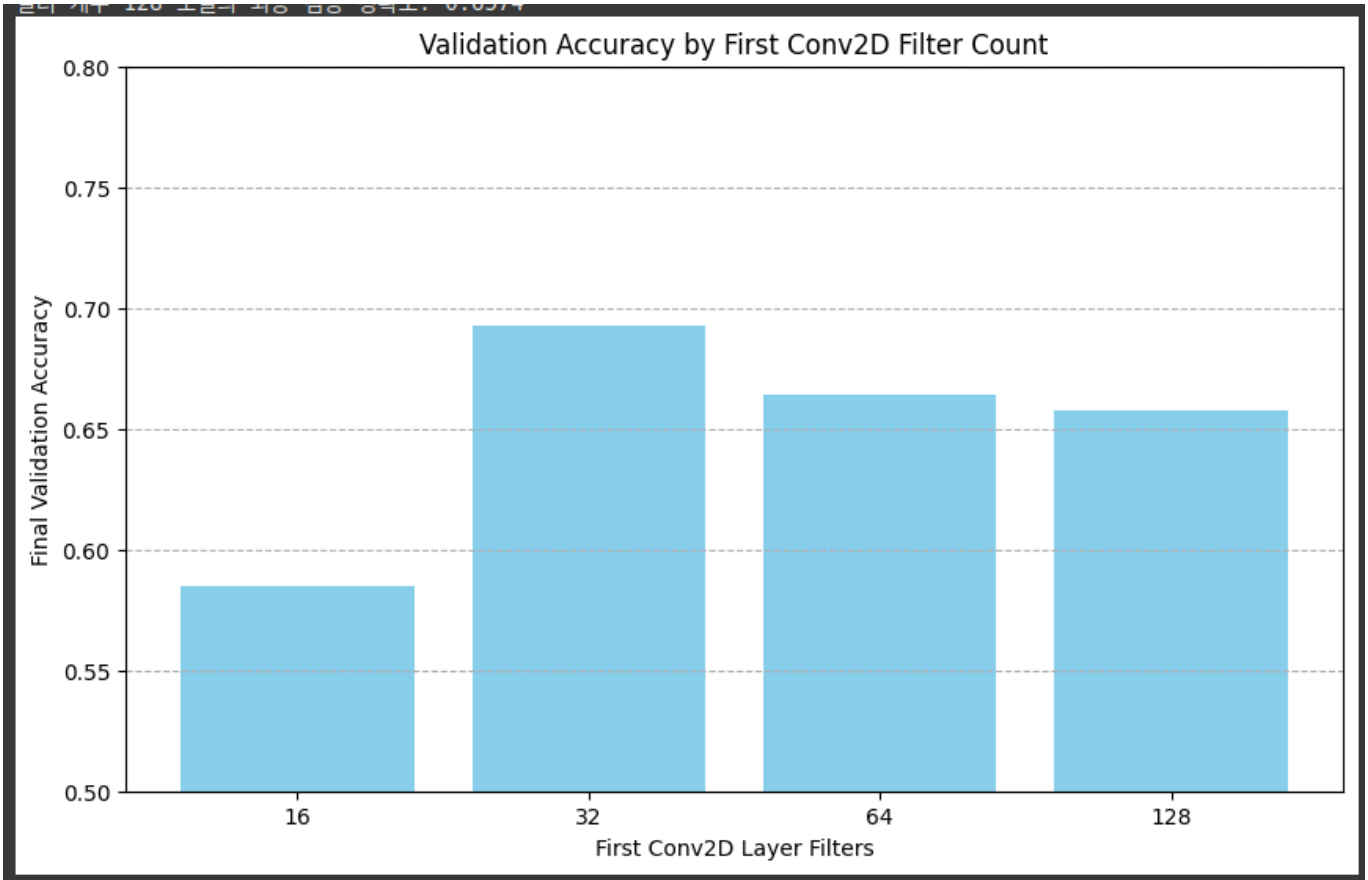
Code

```
# 필터 개수 차이에 따른 성능 변화
```

```
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers

# 1. 데이터 로드 및 전처리
(train_x, train_y), (test_x, test_y) = cifar10.load_data()
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

train_x = train_x / 255.
test_x = test_x / 255.

train_y = to_categorical(train_y)
test_y = to_categorical(test_y)

# 2. 비교할 Conv2D 필터 개수 리스트 정의
# 여기서는 첫 번째 Conv2D 층의 필터 개수를 변경하며 실험합니다.
# 다른 Conv2D 층의 필터 개수도 함께 조절해볼 수 있습니다.
filter_counts_to_test = [16, 32, 64, 128] # 다양한 필터 개수 시도
results = {} # 결과를 저장할 딕셔너리

print("Conv2D 필터 개수에 따른 모델 학습 시작...")

for filters_1st_conv in filter_counts_to_test:
    print(f"\n--- 필터 개수: {filters_1st_conv}로 모델 학습 ---")

    # 3. CNN 모델 디자인 (필터 개수 변경)
    model = models.Sequential()

    # 첫 번째 Conv2D 층의 필터 개수만 변경합니다.
    # (32, 32, 3) => (30, 30, filters_1st_conv)
    model.add(layers.Conv2D(filters=filters_1st_conv, kernel_size=(3, 3),
                            activation='relu',
                            input_shape=(32, 32, 3)))
    # (30, 30, filters_1st_conv) => (15, 15, filters_1st_conv)
    model.add(layers.MaxPool2D(pool_size=(2, 2)))

    # 나머지 Conv2D 층들은 고정된 필터 개수를 사용합니다 (원래 코드와 동일).
    # (15, 15, filters_1st_conv) => (13, 13, 64)
    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
                            activation='relu'))

    # (13, 13, 64) => (6, 6, 64)
    model.add(layers.MaxPool2D(pool_size=(2, 2)))

    # (6, 6, 64) => (4, 4, 64)
    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
                            activation='relu'))

    # 3D를 1D로 변환
    model.add(layers.Flatten())

    # Classification : Fully Connected Layer 추가
```

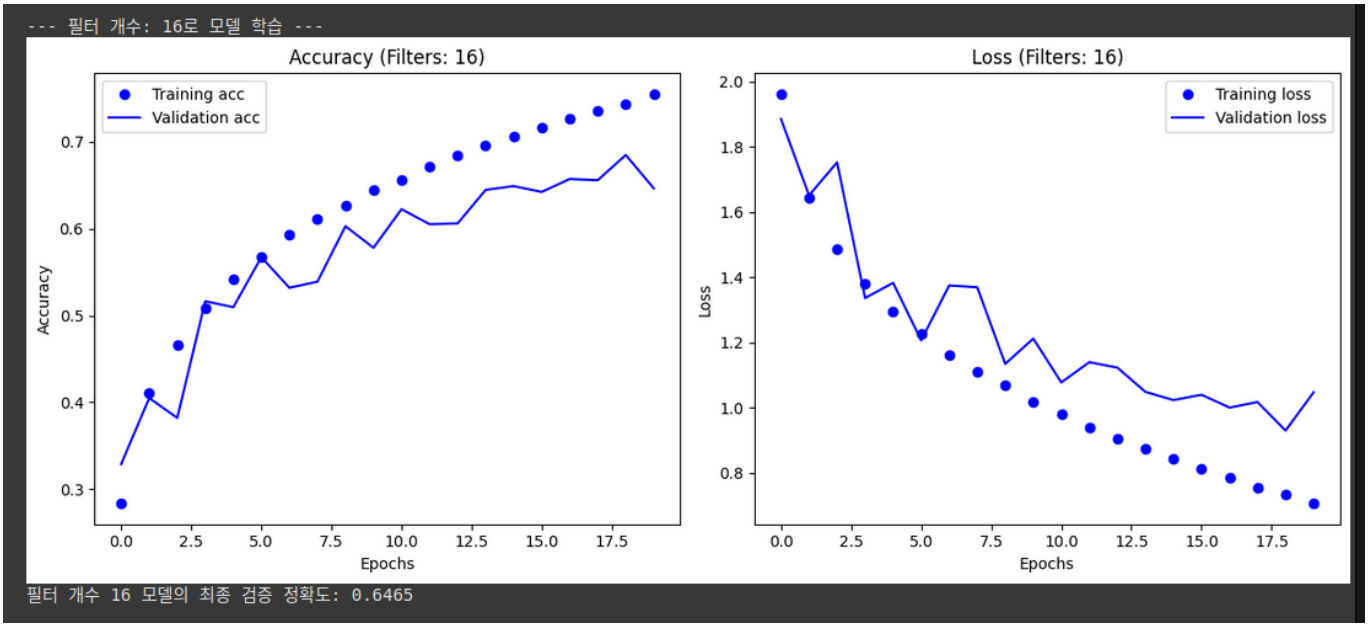
```
    model.add(layers.Dense(units=64, activation='relu'))
    model.add(layers.Dense(units=10, activation='softmax'))

    # 모델의 학습 정보 설정
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

    # 모델 학습
    history = model.fit(x=train_x, y=train_y, epochs=20, batch_size=256,
verbose=0, validation_split=0.2)
    # verbose=0으로 설정하여 가ㄱ 에포크의 출력을 생략하고, 최종 결과만 확인합니다.

    # 결과 저장
    results[filters_1st_conv] = history.history['val_accuracy'][-1] # 마지막
에포크의 검증 정확도
    print(f"필터 개수 {filters_1st_conv} 모델의 최종 검증 정확도:
{results[filters_1st_conv]:.4f}")

# 4. 결과 시가ㄱ화
plt.figure(figsize=(10, 6))
plt.bar([str(k) for k in results.keys()], results.values(),
color='skyblue')
plt.xlabel('First Conv2D Layer Filters')
plt.ylabel('Final Validation Accuracy')
plt.title('Validation Accuracy by First Conv2D Filter Count')
plt.ylim(0.5, 0.8) # y축 범위 조정 (정확도가 이 범위 내에 있을 것으로 예상)
plt.grid(axis='y', linestyle='--')
plt.show()

print("\n모든 모델 학습 및 비교 완료.")
for filters, acc in results.items():
    print(f"필터 {filters}개: 최종 검증 정확도 = {acc:.4f}")
```

결과

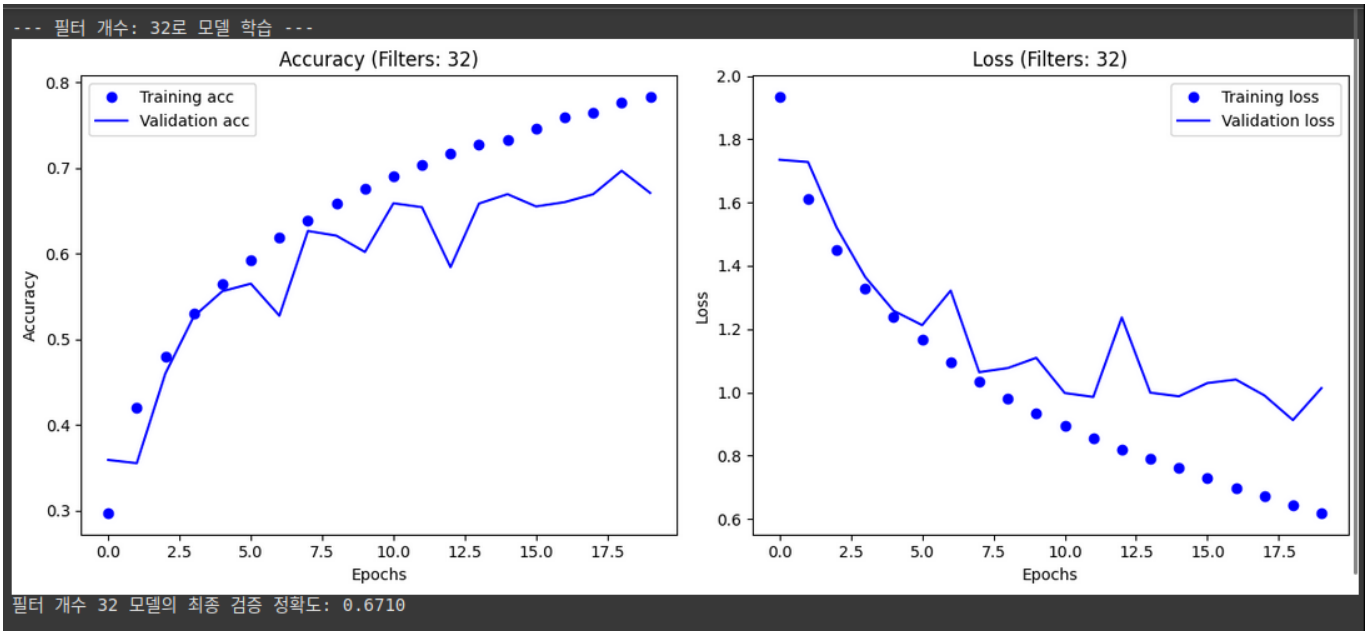



```
모든 모델 학습 및 비교 완료.
필터 16개: 최종 검증 정확도 = 0.5847
필터 32개: 최종 검증 정확도 = 0.6924
필터 64개: 최종 검증 정확도 = 0.6640
필터 128개: 최종 검증 정확도 = 0.6574
```

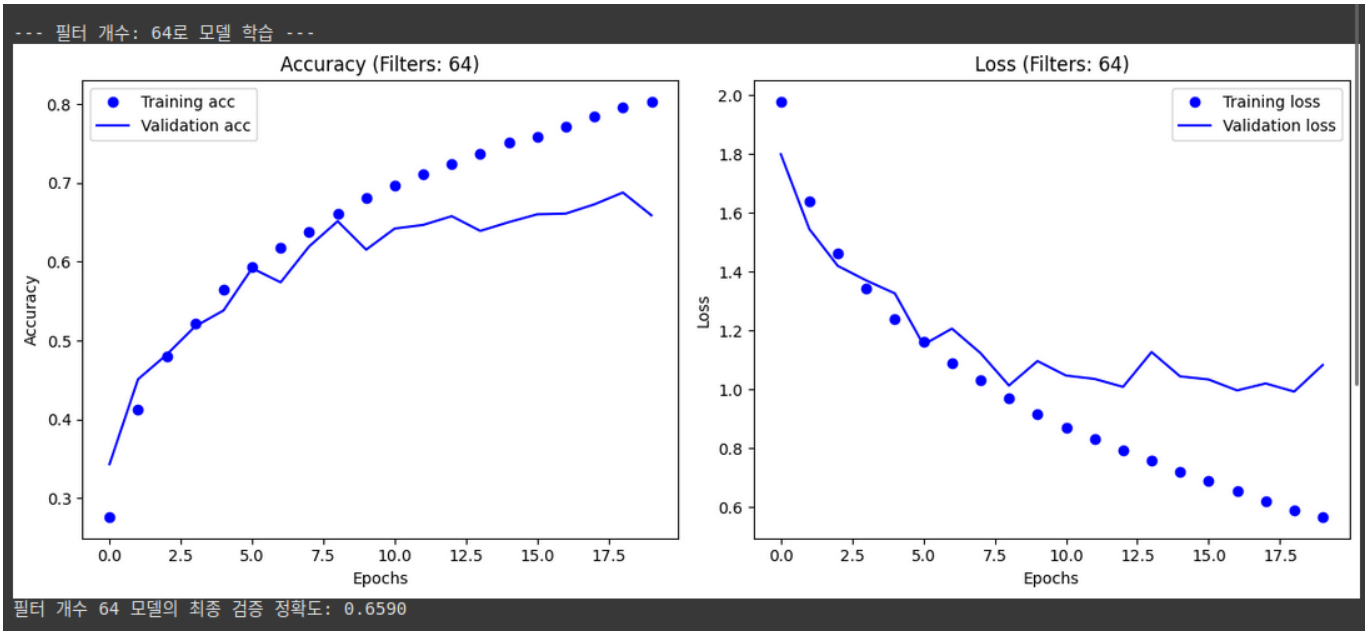
[필터 개수: 16]



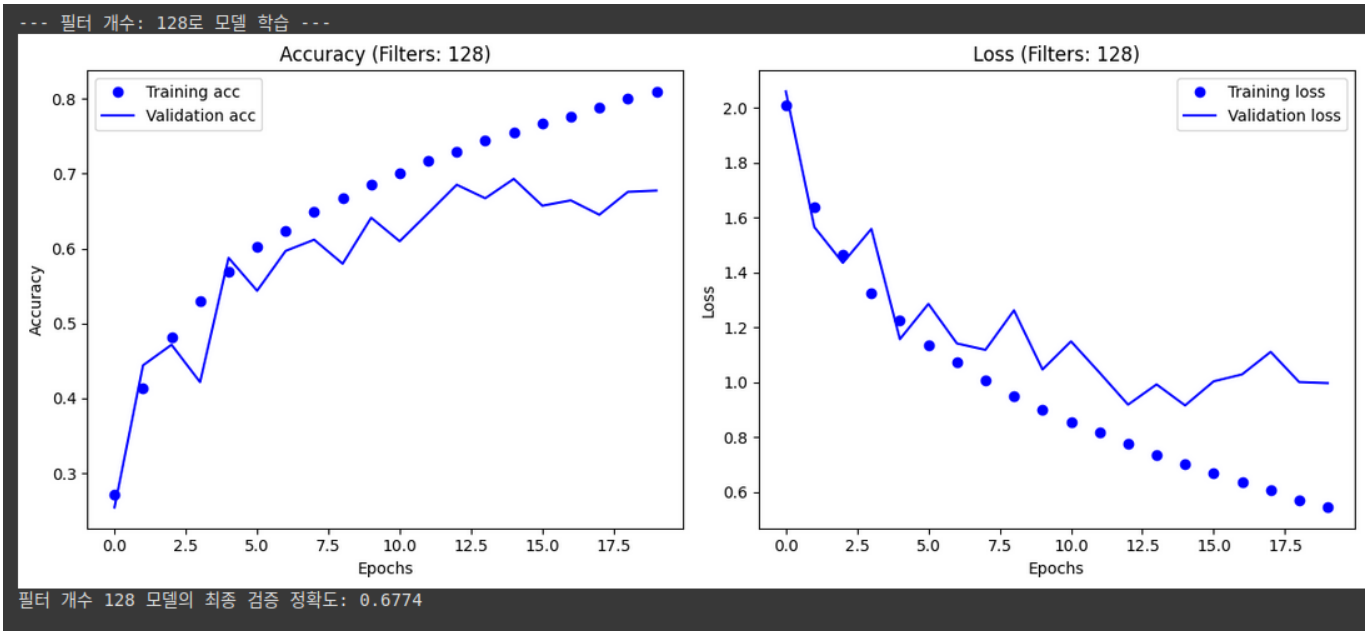
[필터 개수: 32]



[필터 개수: 64]



[필터 개수: 128]



- 첫번째 Convoltion Layer의 필터 개수가 32개일 때 가장 정확도가 높다.
- 필터의 개수가 낮을 때 성능이 제일 낮다
- 필터의 개수가 많다고 성능이 개선되는 것은 아니다
- 필터의 개수가 적을때 과적합도 제일 적고, 필터의 개수가 늘어날수록 과적합도가 커지는 양상을 보인다
- 손실율의 경우, 필터의 개수가 증가할수록 1%이하로 떨어지지 못하는 모습을 보인다

Kernel size에 따른 정확도 차이

Code

```
# 커널 사이즈에 따른 차이

import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers

# 1. 데이터 로드 및 전처리
(train_x, train_y), (test_x, test_y) = cifar10.load_data()
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

train_x = train_x / 255.
test_x = test_x / 255.

train_y = to_categorical(train_y)
test_y = to_categorical(test_y)

# 2. 비교할 Conv2D 커널 사이즈 리스트 정의
# 여기서는 첫 번째 Conv2D 층의 kernel_size를 변경하며 실험합니다.
# 튜플 형태로 (가로, 세로)를 명시합니다.
kernel_sizes_to_test = [(3, 3), (5, 5), (7, 7)] # 다양한 커널 사이즈 시도
final_val_accuracies = {} # 최종 검증 정확도를 저장할 딕셔너리
```

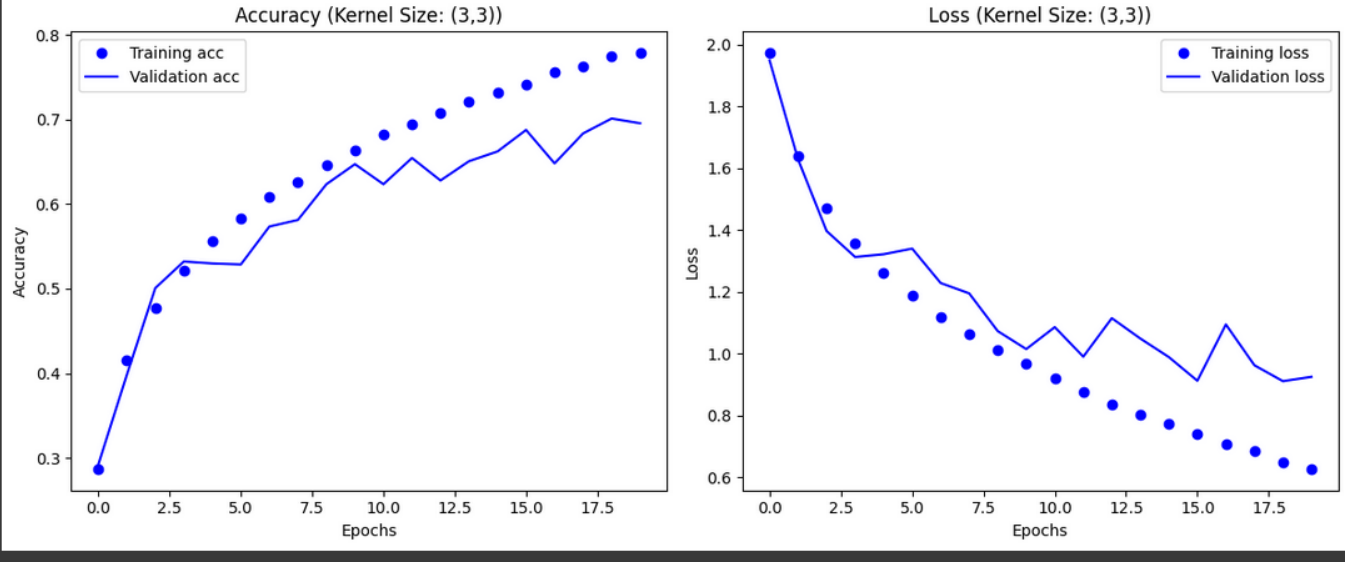
```
print("Conv2D 커널 사이즈에 따른 모델 학습 시작...")

for current_kernel_size in kernel_sizes_to_test:
    # 커널 사이즈를 문자열로 변환하여 출력 및 그래프 레이블에 사용
    kernel_size_str = f"({current_kernel_size[0]},
{current_kernel_size[1]})"
    print(f"\n--- 커널 사이즈: {kernel_size_str}로 모델 학습 ---")

    # 3. CNN 모델 디자인 (커널 사이즈 변경)
    model = models.Sequential()

    # 첫 번째 Conv2D 층의 kernel_size만 변경합니다.
    # input_shape는 (32, 32, 3)으로 동일
    # output_shape는 kernel_size에 따라 변함 (예: (32-k+1, 32-k+1, 32))
    model.add(layers.Conv2D(filters=32, kernel_size=current_kernel_size,
                            activation='relu',
                            input_shape=(32, 32, 3)))
    model.add(layers.MaxPool2D(pool_size=(2, 2)))

    # 나머지 Conv2D 층들은 고정된 필터 개수와 커널 사이즈를 사용합니다 (원본 코드와 동
일).
    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3), # 고정된 (3,3) 커
널
                            activation='relu'))
    model.add(layers.MaxPool2D(pool_size=(2, 2)))

    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3), # 고정된 (3,3) 커
널
                            activation='relu'))

    model.add(layers.Flatten())

    model.add(layers.Dense(units=64, activation='relu'))
    model.add(layers.Dense(units=10, activation='softmax'))

    # 모델의 학습 정보 설정
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

    # 모델 학습
    history = model.fit(x=train_x, y=train_y, epochs=20, batch_size=256,
verbose=0, validation_split=0.2)

    # 최종 검증 정확도 저장
    final_val_accuracies[kernel_size_str] = history.history['val_accuracy']
[-1]
    print(f"커널 사이즈 {kernel_size_str} 모델의 최종 검증 정확도:
{final_val_accuracies[kernel_size_str]:.4f}")


    # 4. 결과 시가ㄱ화 (오버피팅 확인)
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
```

```
        loss = history.history['loss']
        val_loss = history.history['val_loss']

        epochs = range(len(acc))

        plt.figure(figsize=(12, 5))

        # 정확도 그래프
        plt.subplot(1, 2, 1) # 1행 2열 중 첫 번째 플롯
        plt.plot(epochs, acc, 'bo', label='Training acc')
        plt.plot(epochs, val_acc, 'b', label='Validation acc')
        plt.title(f'Accuracy (Kernel Size: {kernel_size_str})')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.legend()

        # 손실 그래프
        plt.subplot(1, 2, 2) # 1행 2열 중 두 번째 플롯
        plt.plot(epochs, loss, 'bo', label='Training loss')
        plt.plot(epochs, val_loss, 'b', label='Validation loss')
        plt.title(f'Loss (Kernel Size: {kernel_size_str})')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

        plt.tight_layout() # 서브플롯 가ㄴ의 가ㄴ격 자동 조절
        plt.show()

# 5. 모든 모델의 최종 검증 정확도 막대그래프 시가ㄱ화
plt.figure(figsize=(10, 6))
plt.bar(final_val_accuracies.keys(), final_val_accuracies.values(),
color='lightgreen')
plt.xlabel('First Conv2D Layer Kernel Size')
plt.ylabel('Final Validation Accuracy')
plt.title('Validation Accuracy by First Conv2D Kernel Size')
plt.ylim(0.5, 0.8) # y축 범위 조정 (정확도가 이 범위 내에 있을 것으로 예상)
plt.grid(axis='y', linestyle='--')
plt.show()

print("\n모든 모델 학습 및 비교 완료.")
for k_size, acc in final_val_accuracies.items():
    print(f"커널 사이즈 {k_size}: 최종 검증 정확도 = {acc:.4f}")
```

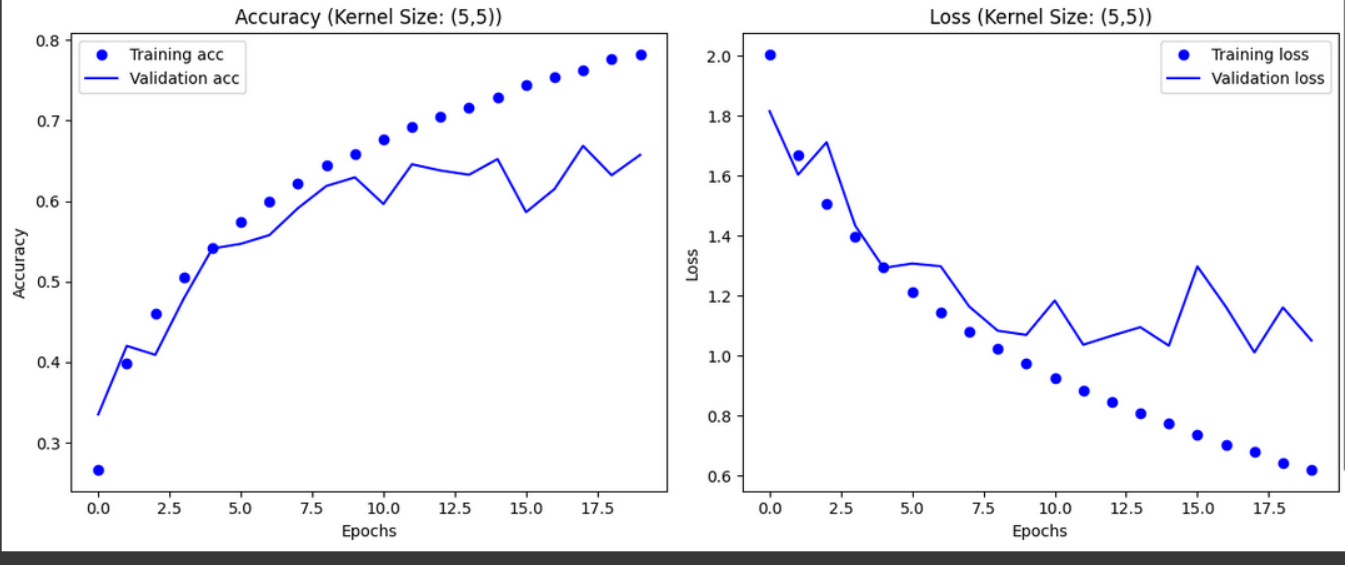
결과

[Kernel size = (3,3)]

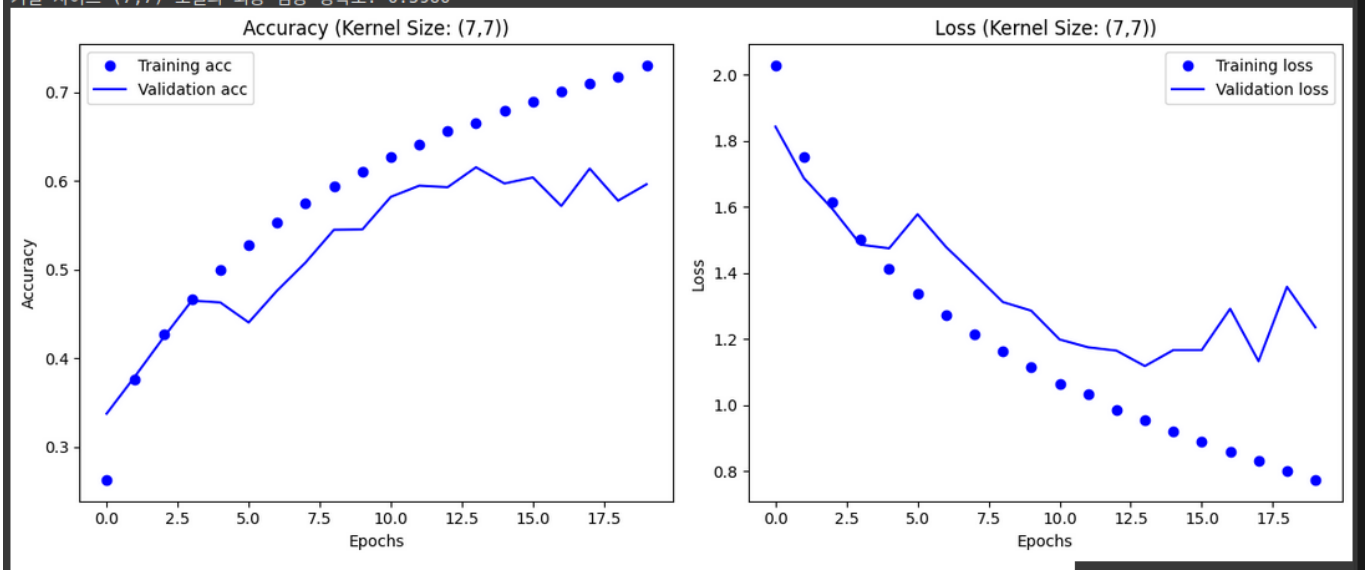
```
--- 커널 사이즈: (3,3)로 모델 학습 ---
커널 사이즈 (3,3) 모델의 최종 검증 정확도: 0.6954
```

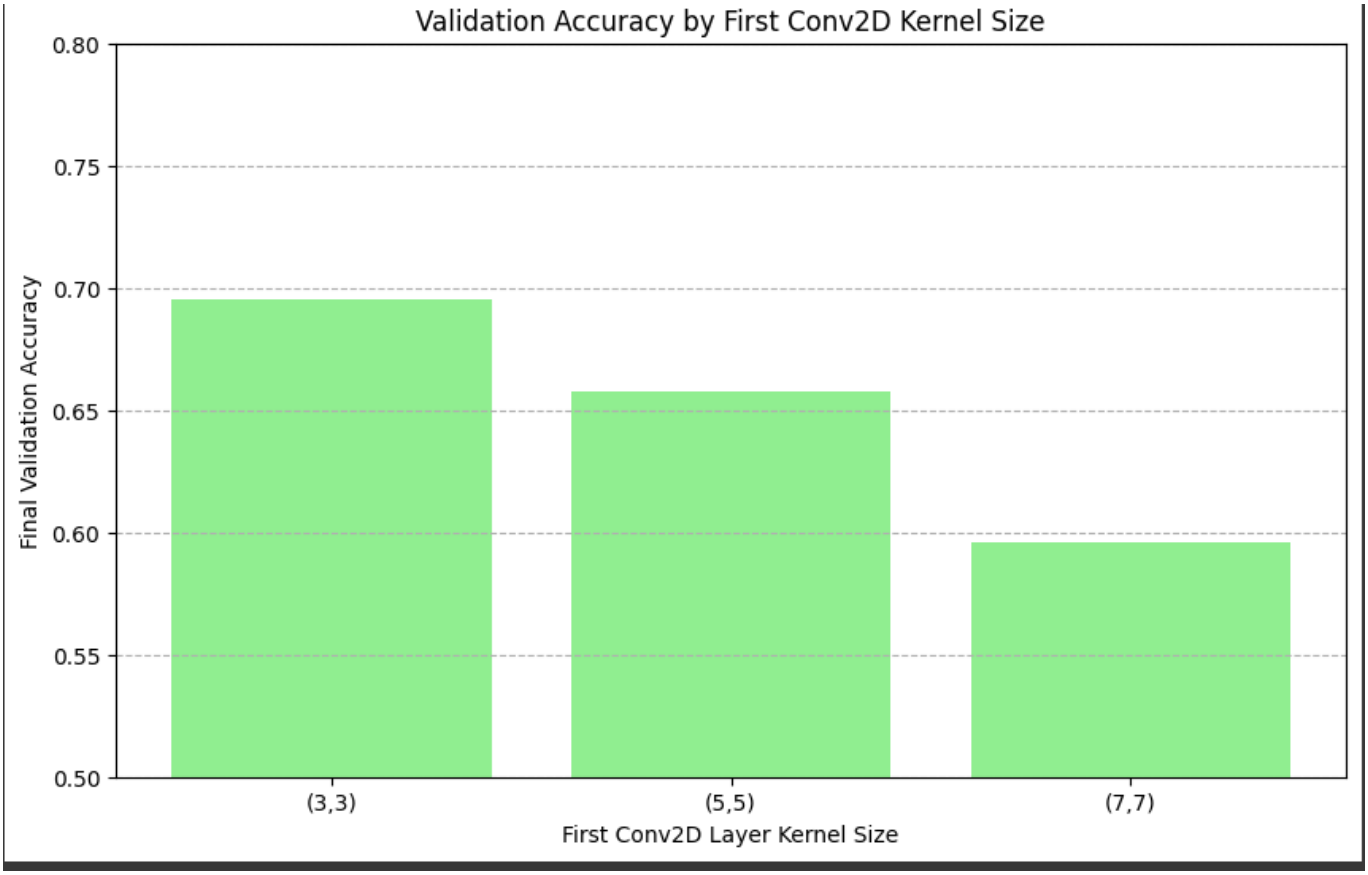



```
--- 커널 사이즈: (5,5)로 모델 학습 ---
커널 사이즈 (5,5) 모델의 최종 검증 정확도: 0.6574
```




```
--- 커널 사이즈: (7,7)로 모델 학습 ---
커널 사이즈 (7,7) 모델의 최종 검증 정확도: 0.5960
```





- 커널 사이즈에 따른 과적합도는 큰 차이를 보이진 않는다
- 다만 커널 사이즈가 증가할수록 과적합도가 커지는 모습을 보인다
- 성능 정확도의 경우 커널의 사이즈가 증가할수록 감소하는 모습을 보여준다
- 커널의 사이즈가 (3,3)일 때 가장 높은 정확도를 보여준다

Pool Size에 따른 성능비교

Code

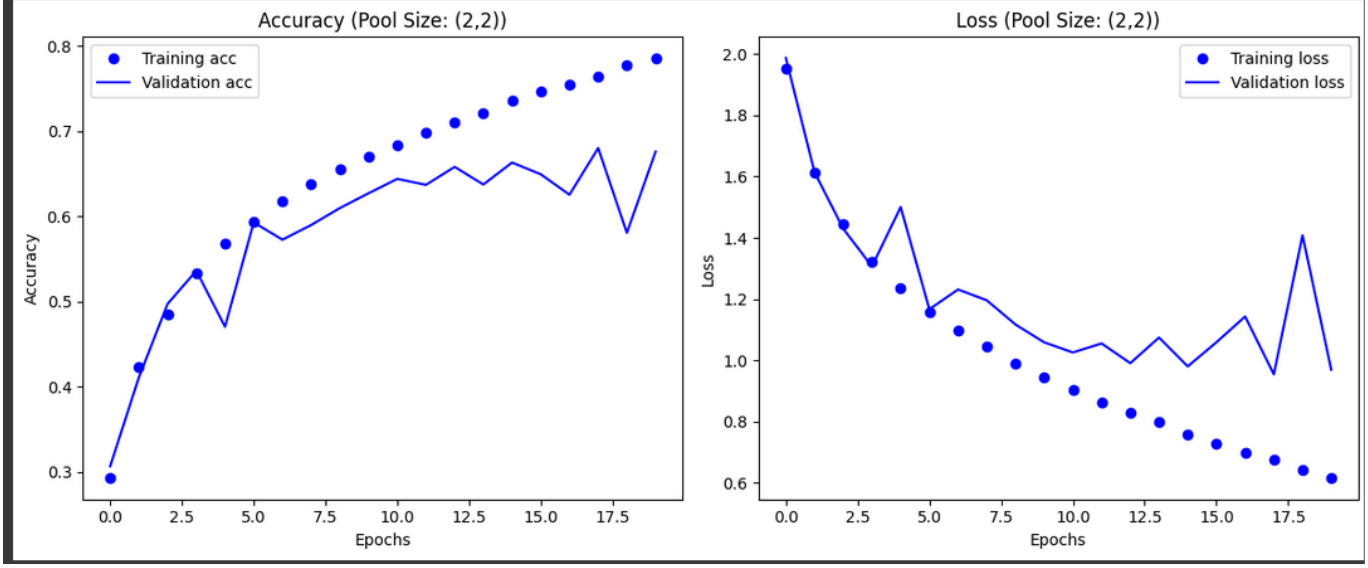
```
# MaxPooling 사이즈에 따른 성능비교
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers

# 1. 데이터 로드 및 전처리
(train_x, train_y), (test_x, test_y) = cifar10.load_data()
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

train_x = train_x / 255.
test_x = test_x / 255.

train_y = to_categorical(train_y)
test_y = to_categorical(test_y)

# 2. 비교할 MaxPool2D pool_size 리스트 정의
# 첫 번째 MaxPool2D 층의 pool_size를 변경하며 실험합니다.
```

```
pool_sizes_to_test = [(2, 2), (3, 3)] # 일반적으로 (2,2)가 많이 사용되지만,
(3,3)도 시도해 볼 수 있습니다.
final_val_accuracies = {} # 최종 검증 정확도를 저장할 딕셔너리

print("MaxPool2D pool_size에 따른 모델 학습 시작...")

for current_pool_size in pool_sizes_to_test:
    # pool_size를 문자열로 변환하여 출력 및 그래프 레이블에 사용
    pool_size_str = f"({current_pool_size[0]},{current_pool_size[1]})"
    print(f"\n--- Pool Size: {pool_size_str}로 모델 학습 ---")

    # 3. CNN 모델 디자인 (pool_size 변경)
    model = models.Sequential()

    # 첫 번째 Conv2D 층 (고정)
    model.add(layers.Conv2D(filters=32, kernel_size=(3, 3),
                            activation='relu',
                            input_shape=(32, 32, 3)))
    # 첫 번째 MaxPool2D 층의 pool_size만 변경합니다.
    # (30, 30, 32) => (ceil(30/p), ceil(30/p), 32)
    model.add(layers.MaxPool2D(pool_size=current_pool_size))

    # 나머지 Conv2D 및 MaxPool2D 층들은 고정된 설정 사용 (원본 코드와 동일).
    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
                            activation='relu'))
    model.add(layers.MaxPool2D(pool_size=(2, 2))) # 두 번째 풀링 층은 (2,2) 고
정

    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
                            activation='relu'))

    model.add(layers.Flatten())

    model.add(layers.Dense(units=64, activation='relu'))
    model.add(layers.Dense(units=10, activation='softmax'))

    # 모델의 학습 정보 설정
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

    # 모델 학습
    history = model.fit(x=train_x, y=train_y, epochs=20, batch_size=256,
verbose=0, validation_split=0.2)

    # 최종 검증 정확도 저장
    final_val_accuracies[pool_size_str] = history.history['val_accuracy']
[-1]
    print(f"Pool Size {pool_size_str} 모델의 최종 검증 정확도:
{final_val_accuracies[pool_size_str]:.4f}")

    # 4. 결과 시가ㄱ화 (오버피팅 확인)
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
```

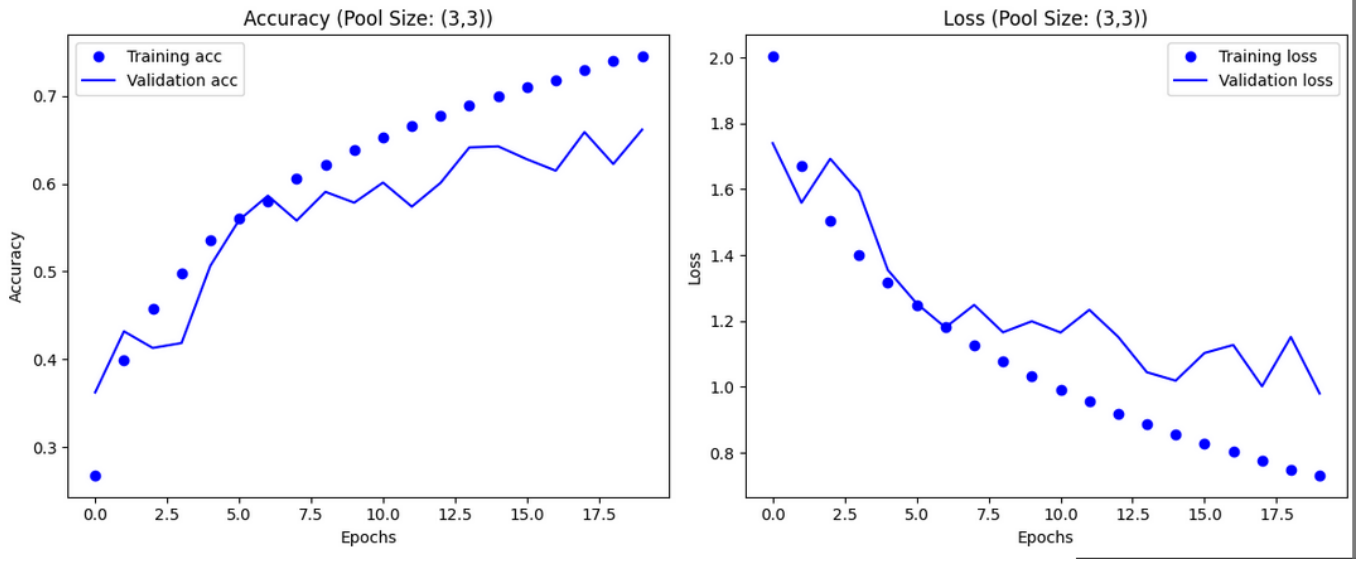
```
    val_loss = history.history['val_loss']

    epochs = range(len(acc))

    plt.figure(figsize=(12, 5))

    # 정확도 그래프
    plt.subplot(1, 2, 1) # 1행 2열 중 첫 번째 플롯
    plt.plot(epochs, acc, 'bo', label='Training acc')
    plt.plot(epochs, val_acc, 'b', label='Validation acc')
    plt.title(f'Accuracy (Pool Size: {pool_size_str})')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # 손실 그래프
    plt.subplot(1, 2, 2) # 1행 2열 중 두 번째 플롯
    plt.plot(epochs, loss, 'bo', label='Training loss')
    plt.plot(epochs, val_loss, 'b', label='Validation loss')
    plt.title(f'Loss (Pool Size: {pool_size_str})')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout() # 서브플롯 가ㄴ의 가ㄴ격 자동 조절
    plt.show()

# 5. 모든 모델의 최종 검증 정확도 막대그래프 시가ㄱ화
plt.figure(figsize=(10, 6))
plt.bar(final_val_accuracies.keys(), final_val_accuracies.values(),
color='lightcoral')
plt.xlabel('First MaxPool2D Layer Pool Size')
plt.ylabel('Final Validation Accuracy')
plt.title('Validation Accuracy by First MaxPool2D Pool Size')
plt.ylim(0.5, 0.8) # y축 범위 조정 (정확도가 이 범위 내에 있을 것으로 예상)
plt.grid(axis='y', linestyle='--')
plt.show()

print("\n모든 모델 학습 및 비교 완료.")
for p_size, acc in final_val_accuracies.items():
    print(f"풀링 사이즈 {p_size}: 최종 검증 정확도 = {acc:.4f}")
```

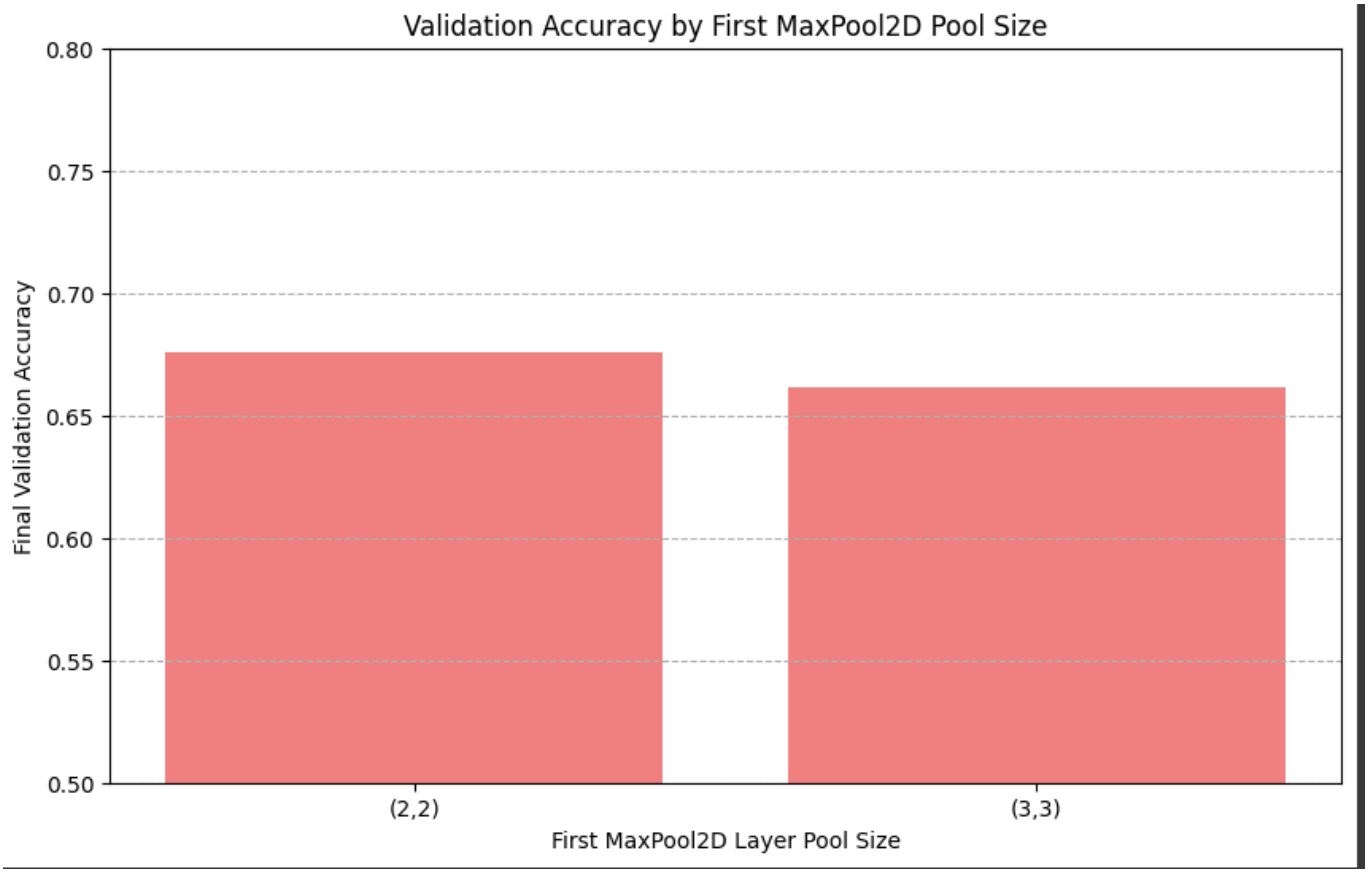
결과

```
--- Pool Size: (2,2)로 모델 학습 ---
Pool Size (2,2) 모델의 최종 검증 정확도: 0.6760
```




```
--- Pool Size: (3,3)로 모델 학습 ---
Pool Size (3,3) 모델의 최종 검증 정확도: 0.6615
```



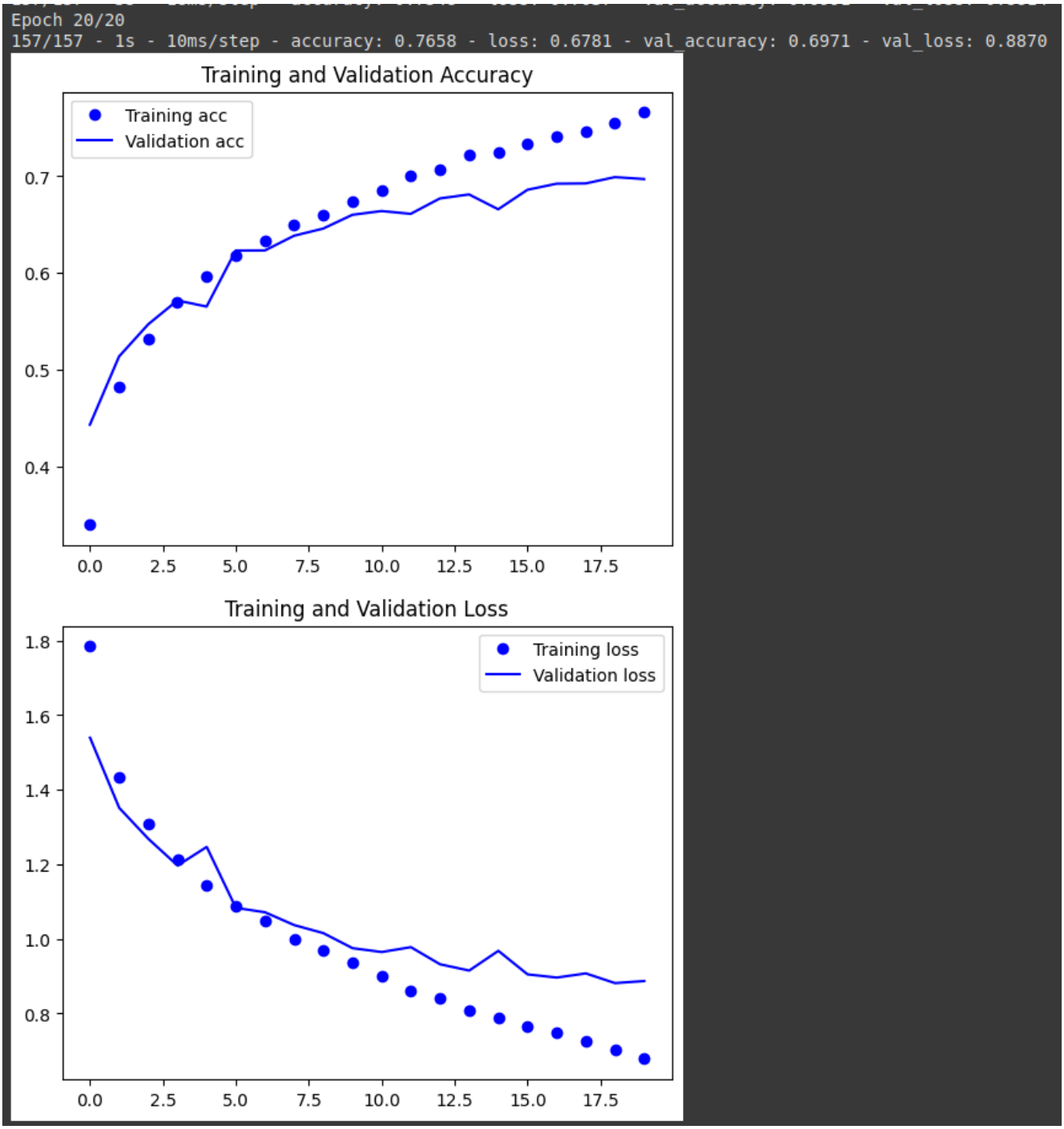


- Pool size가 증가할수록 검증 및 훈련 정확도의 차이가 적은 것으로 보여진다
- 다만 pool size가 증가할수록, 정확도가 감소하는 것으로 보인다

Optimizer에 따른 성능 차이

> 기존 `rmsprop`에서 `adam`으로 변경

결과



- 기존 optimizer를 사용했을 때, 정확도가 0.699가 나온 반면 adam을 사용하니 0.697수준으로 떨어졌다

DropOut을 추가했을때 성능 차이

Code

```
# Drop Out을 추가했을때
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers

# 1. 데이터 로드 및 전처리
```

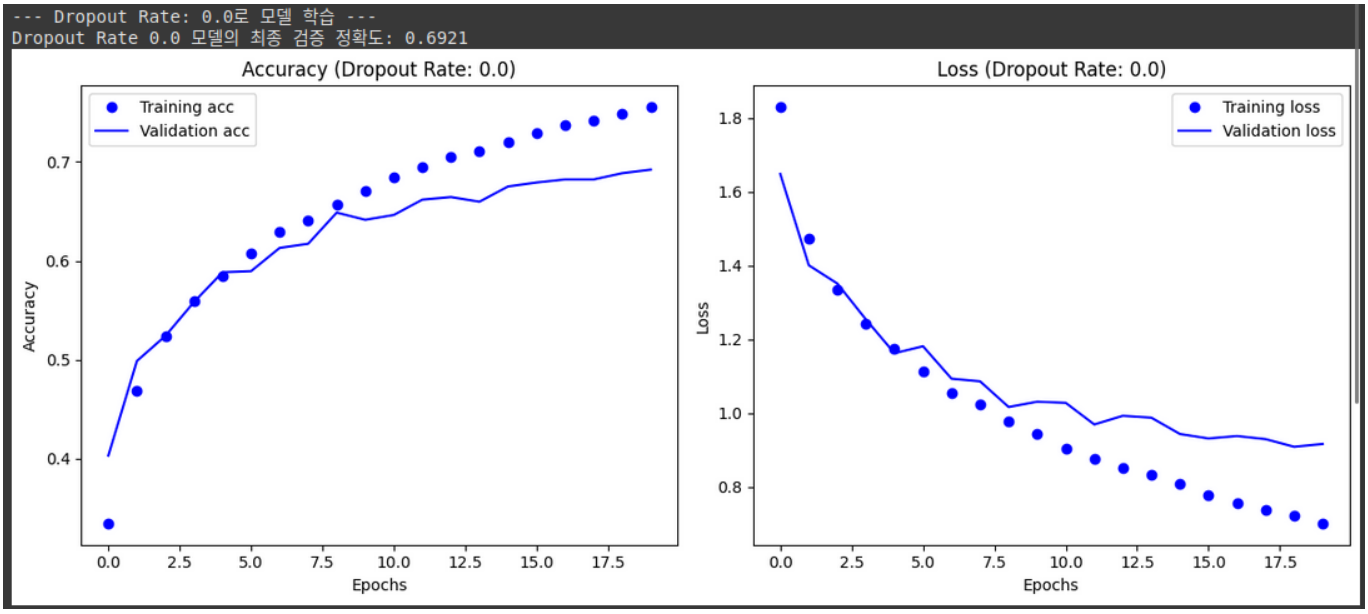
```
(train_x, train_y), (test_x, test_y) = cifar10.load_data()
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

train_x = train_x / 255.
test_x = test_x / 255.

train_y = to_categorical(train_y)
test_y = to_categorical(test_y)

# 2. 비교할 Dropout 비율 리스트 정의
# 다양한 드롭아웃 비율을 시도해봅니다. 0.0은 드롭아웃을 사용하지 않는 경우와 가ㅌ습니다.
dropout_rates_to_test = [0.0, 0.2, 0.4, 0.5]
final_val_accuracies = {} # 최종 검증 정확도를 저장할 딕셔너리

print("Dropout 비율에 따른 모델 학습 시작...")

for dropout_rate in dropout_rates_to_test:
    print(f"\n--- Dropout Rate: {dropout_rate}로 모델 학습 ---")

    # 3. CNN 모델 디자인 (Dropout 추가 및 비율 변경)
    model = models.Sequential()

    # (32, 32, 3) => (30, 30, 32)
    model.add(layers.Conv2D(filters=32, kernel_size=(3, 3),
                            activation='relu',
                            input_shape=(32, 32, 3)))
    # (30, 30, 32) => (15, 15, 32)
    model.add(layers.MaxPool2D(pool_size=(2, 2)))

    # (15, 15, 32) => (13, 13, 64)
    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
                            activation='relu'))

    # (13, 13, 64) => (6, 6, 64)
    model.add(layers.MaxPool2D(pool_size=(2, 2)))

    # (6, 6, 64) => (4, 4, 64)
    model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
                            activation='relu'))

    # 3D를 1D로 변환
    model.add(layers.Flatten())

    # Dropout 층 추가: 첫 번째 Dense 층 직전에 적용
    if dropout_rate > 0: # 드롭아웃 비율이 0보다 클 때만 층을 추가
        model.add(layers.Dropout(dropout_rate))

    # Classification : Fully Connected Layer 추가
    model.add(layers.Dense(units=64, activation='relu'))
    model.add(layers.Dense(units=10, activation='softmax'))

    # 모델의 학습 정보 설정 (옵티마이저를 'adam'으로 통일)
    model.compile(optimizer='adam', loss='categorical_crossentropy',
```

```
metrics=['accuracy'])

    # 모델 학습
    history = model.fit(x=train_x, y=train_y, epochs=20, batch_size=256,
verbose=0, validation_split=0.2)

    # 최종 검증 정확도 저장
    final_val_accuracies[dropout_rate] = history.history['val_accuracy']
[-1]
    print(f"Dropout Rate {dropout_rate} 모델의 최종 검증 정확도:
{final_val_accuracies[dropout_rate]:.4f}")

    # 4. 결과 시가ㄱ화 (오버피팅 확인)
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs = range(len(acc))

    plt.figure(figsize=(12, 5))

    # 정확도 그래프
    plt.subplot(1, 2, 1) # 1행 2열 중 첫 번째 플롯
    plt.plot(epochs, acc, 'bo', label='Training acc')
    plt.plot(epochs, val_acc, 'b', label='Validation acc')
    plt.title(f'Accuracy (Dropout Rate: {dropout_rate})')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # 손실 그래프
    plt.subplot(1, 2, 2) # 1행 2열 중 두 번째 플롯
    plt.plot(epochs, loss, 'bo', label='Training loss')
    plt.plot(epochs, val_loss, 'b', label='Validation loss')
    plt.title(f'Loss (Dropout Rate: {dropout_rate})')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout() # 서브플롯 가ㄴ의 가ㄴ격 자동 조절
    plt.show()

# 5. 모든 모델의 최종 검증 정확도 막대그래프 시가ㄱ화
plt.figure(figsize=(10, 6))
# x축 레이블을 문자열로 변환하여 표시
plt.bar([str(k) for k in final_val_accuracies.keys()],
final_val_accuracies.values(), color='darkblue')
plt.xlabel('Dropout Rate')
plt.ylabel('Final Validation Accuracy')
plt.title('Validation Accuracy by Dropout Rate')
plt.ylim(0.5, 0.8) # y축 범위 조정 (정확도가 이 범위 내에 있을 것으로 예상)
plt.grid(axis='y', linestyle='--')
plt.show()
```

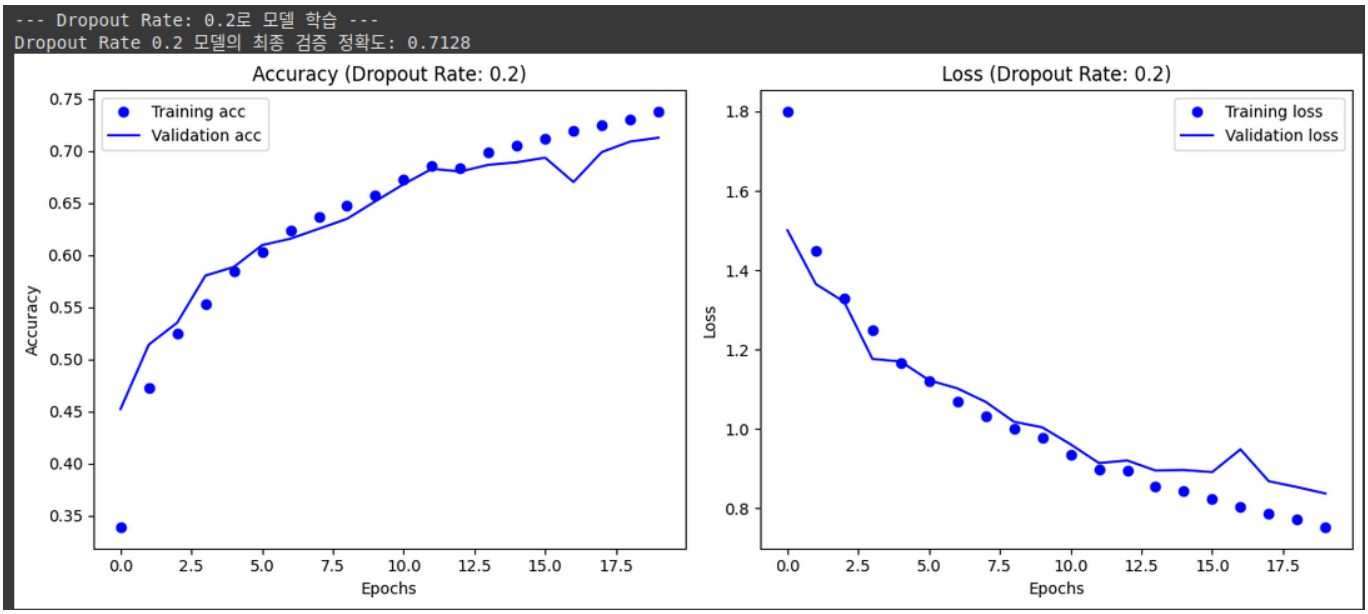
```
print("\n모든 모델 학습 및 비교 완료.")
for rate, acc in final_val_accuracies.items():
    print(f"Dropout Rate {rate}: 최종 검증 정확도 = {acc:.4f}")
```

결과

[DropOut = 0.0]

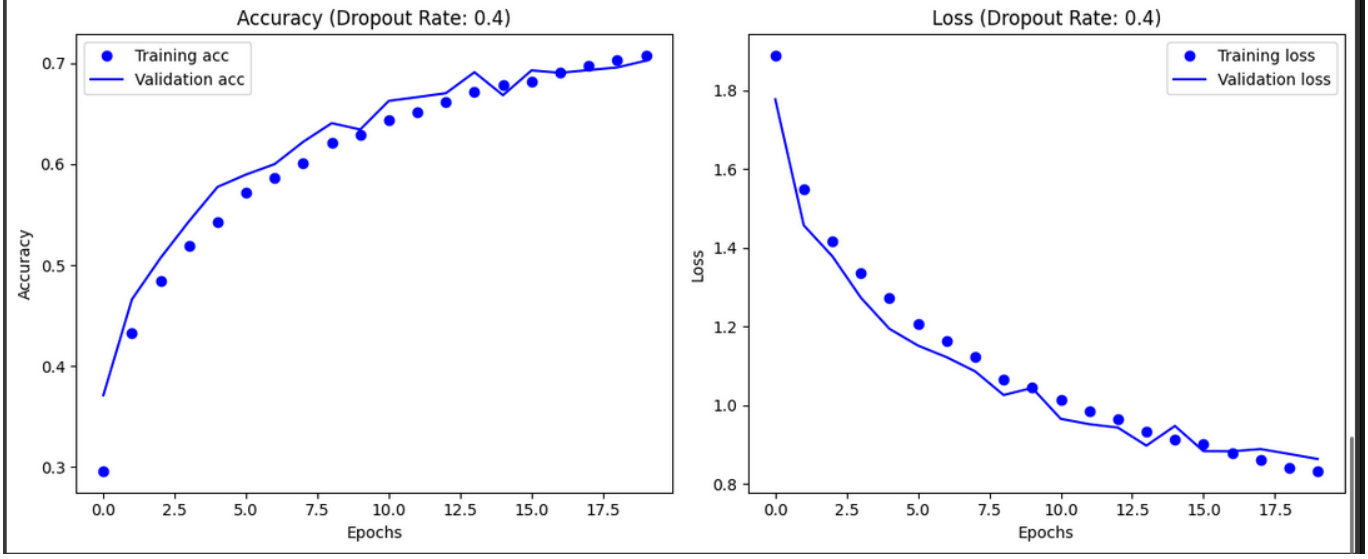


[DropOut = 0.2]



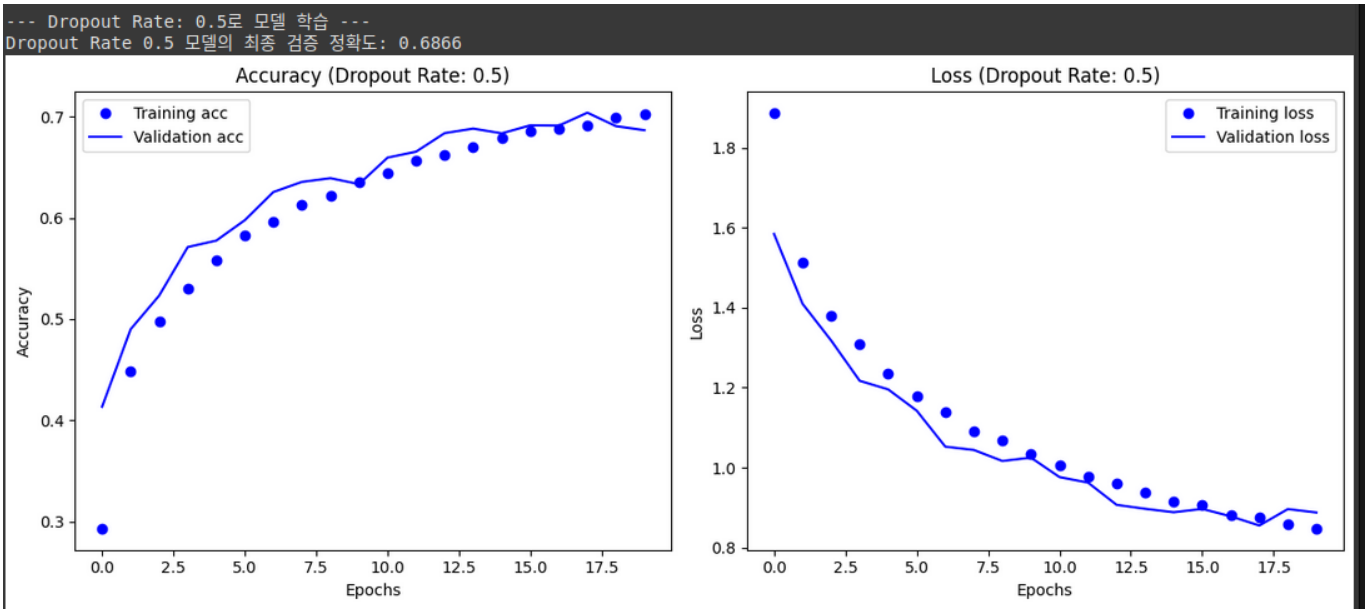
[DropOut = 0.4]

```
--- Dropout Rate: 0.4로 모델 학습 ---
Dropout Rate 0.4 모델의 최종 검증 정확도: 0.7026
```

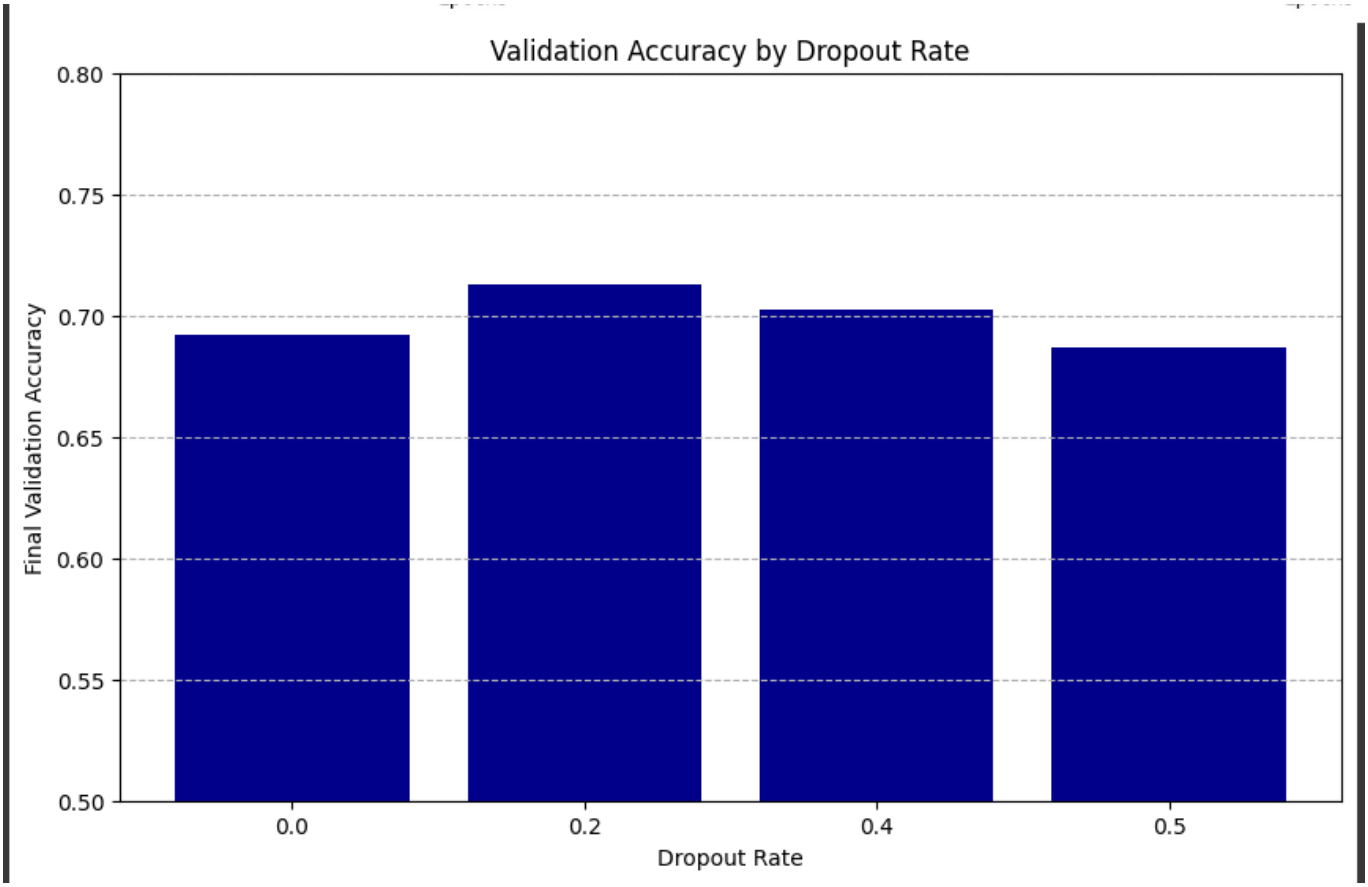


[DropOut = 0.5]

```
--- Dropout Rate: 0.5로 모델 학습 ---
Dropout Rate 0.5 모델의 최종 검증 정확도: 0.6866
```



[전체 비교]



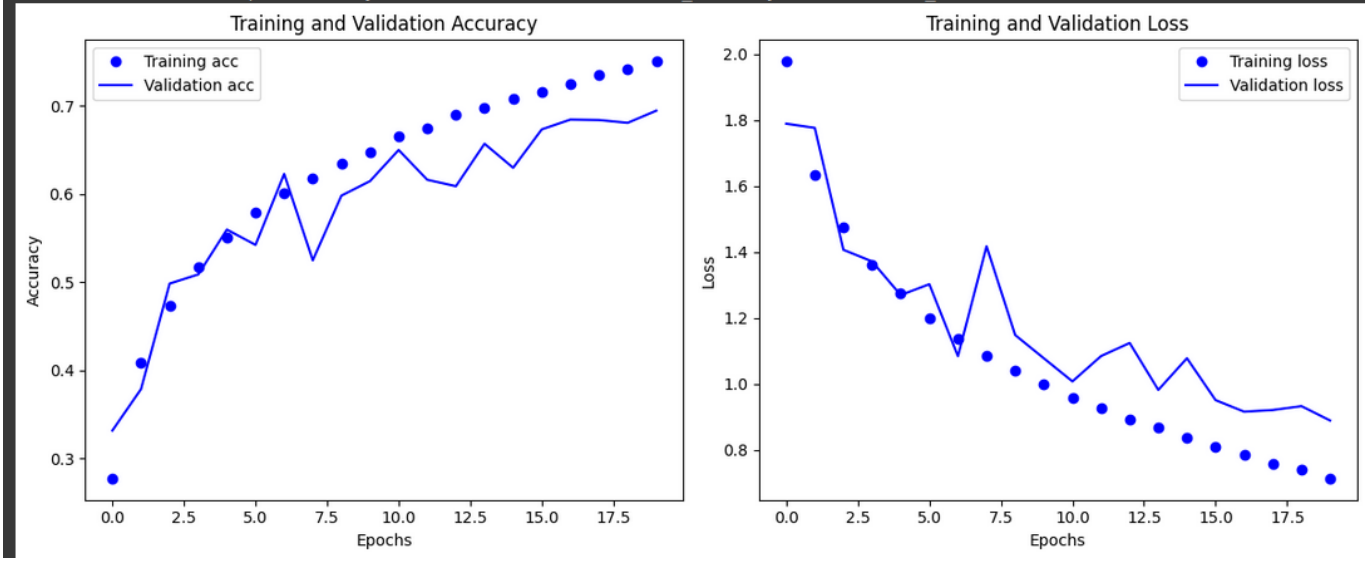
- DropOut의 비율이 0.2일 때, 성능이 0.71수준으로 기존보다 0.02정도 향상하였다
- 확실히 DropOut을 적용하니 과적합도가 낮아지는 것을 확인할 수 있다
- 또한 손실율도 DropOut의 비율이 커질수록 감소하는 추세를 보인다

종합 적용

> 각 파라미터에 대해 최적의 경우들만 추출하여 적용해봄

- Conv2D 필터 개수: 32
- 커널 사이즈: (3,3)
- pool size: (2,2)
- Optimizer: rmsprop
- Drop Out: 0.2

```
Epoch 20/20
157/157 - 1s - 9ms/step - accuracy: 0.7512 - loss: 0.7117 - val_accuracy: 0.6947 - val_loss: 0.8891
```



- 정확도가 0.694정도로 처음 코드와 유사한 정확도를 보여준다
  - 더 나은 파라미터들을 종합한다고 성능이 개선되는 것은 아니다
- 대신 Drop Out을 적용한 관계로, 검증 정확도 및 훈련 정확도의 차이가 줄어든 것을 확인할 수 있다
  - 과적합도가 낮아진 것을 확인할 수 있다

실험 결론

- 필터 개수와 DropOut의 비율은 증감에 따라 비례하여 성능이 개선되진 않는다
- kernel size의 경우 (3,3)일 때 가장 성능이 좋았으며, 커널 크기가 증가할수록 성능이 줄어든다
  - 커널의 사이즈가 커질수록 세부 특징을 포착하는데 어려움
- Pool size가 증가할수록 성능이 감소한다
  - Pooling size가 커질수록 해상도가 급감한다
  - pooling이 클수록 feature의 위치정보를 모호하게 만든다
  - 이미지에서 충분한 의미있는 특징을 학습하지 못하므로
    - underfitting문제가 발생할 수 있다