고재현_HW2_report

1. 전체 구조

모델

기본적으로 문제에서 주어진 조건을 만족하도록 구현하였다. 각각의 레이어에 dropout을 추가하여 overfitting을 방지하였다. batch normalization을 추가하여 학습 속도를 증가시켰다.

최종 모델의 학습 결과는 약 69%의 정확도를 보였다.

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10, dropout=0.2):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 6, 5, stride=1, padding=2),  # in_ch
            nn.BatchNorm2d(6),  # 32*32*6, out_channel = 6(#f
            nn.ReLU(),  # activation of 1st layer=ReLu
            nn.Dropout2d(0.1),
            nn.MaxPool2d(2, stride=2))  # out : 32*32*6 -> 16
        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, 5, stride=1, padding=2),  # in_c
            nn.BatchNorm2d(16),  # 16*16*16
            nn.Dropout2d(0.1),
            nn.ReLU(),
            nn.MaxPool2d(2, stride=2))  # out : 16*16*16-> 8*
        self.layer3 = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(1024, 120),
            nn.BatchNorm1d(120),
            nn.ReLU())
        self.layer4 = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(120, 84),
            nn.BatchNorm1d(84),
            nn.ReLU())
        self.output = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(84, num_classes),
            nn.BatchNorm1d(num_classes)
        )
```

전처리, 데이터셋 사용 방법

CIFAR10 data의 distribution을 계산하여 normalize에 사용하였다.

```
def calc_distribution():
    train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True)
    # use np.concatenate to stick all the images together to form a 1600000 X 32 X 3 array
    x = np.concatenate([np.asarray(train_data[i][0]) for i in range(len(train_data))])
    print(x.shape)
    train_mean = np.mean(x, axis=(0, 1))
    train_std = np.std(x, axis=(0, 1))
    print(train_mean / 255, train_std / 255)
```

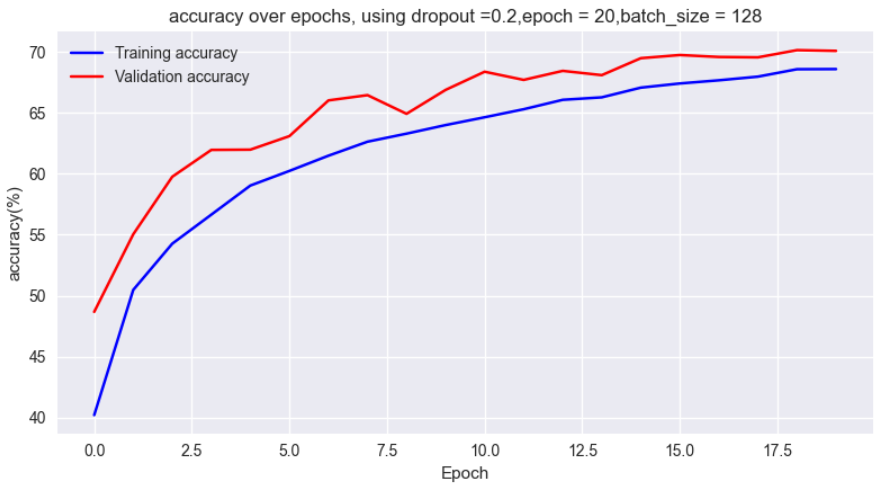
train dataset의 일부를 validation set으로 사용하였다.

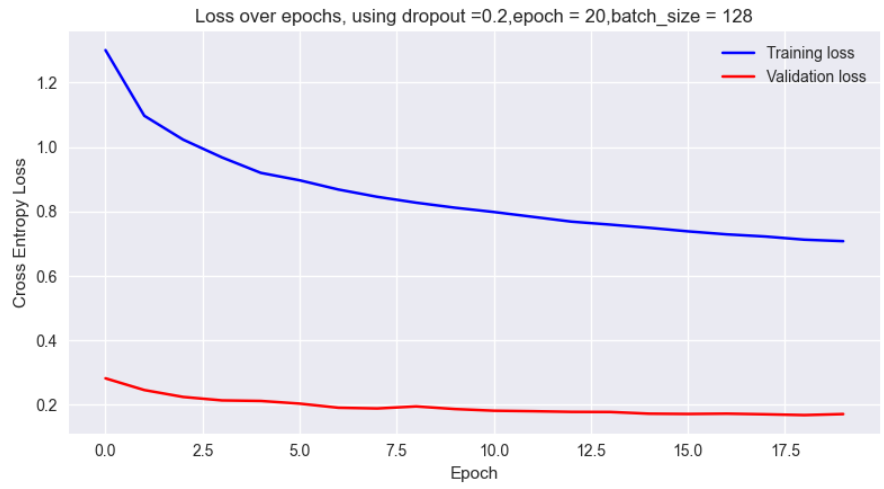
```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=False, transform=transform_test)

num_train = len(trainset)
indices = list(range(num_train))
train_idx, valid_idx = indices[10000:], indices[:10000]
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, sampler=train_sampler,
                                           num_workers=2)

test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                          shuffle=False, num_workers=2)

valid_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, sampler=valid_sampler,
                                           num_workers=2)
```

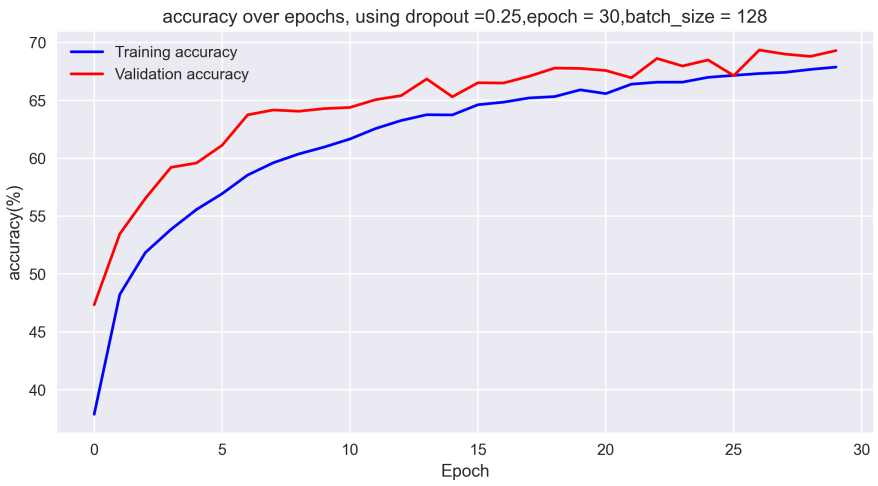
2. 모델의 구조 및 전처리 방법에 따른 학습 효과 변화

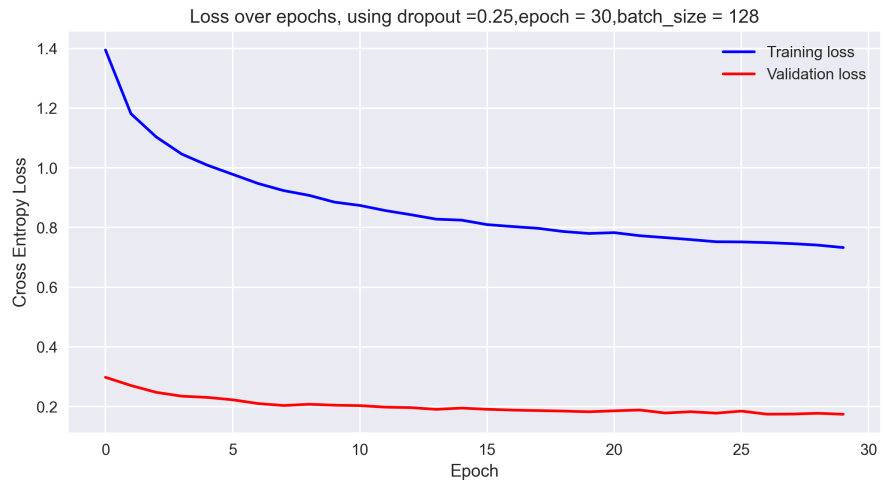
2.1 dropout, learning rate, batch size에 따른 학습 속도 변화

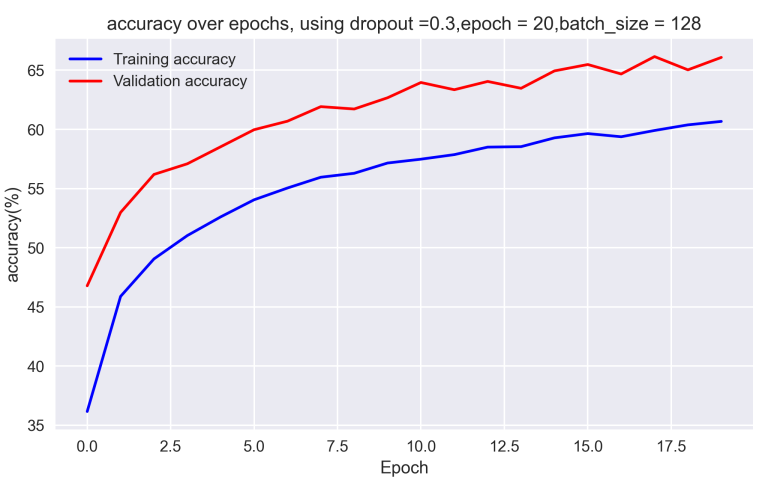
2.1.1 Dropout

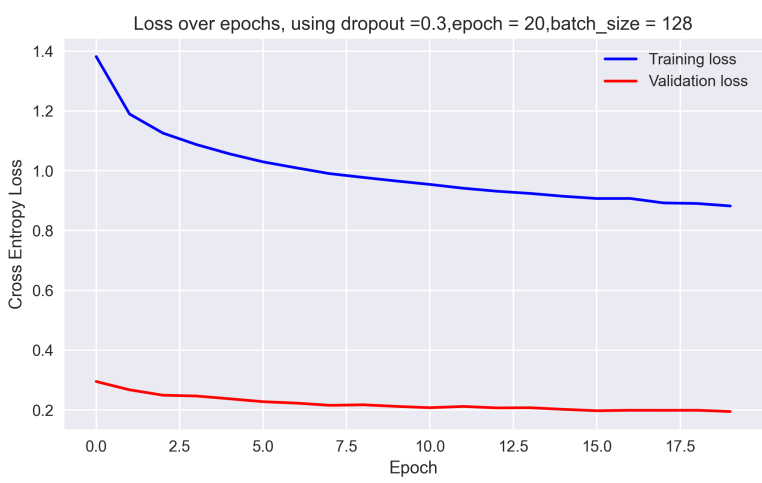












dropout=0.2로 적용했을 때 가장 효과적이었다.

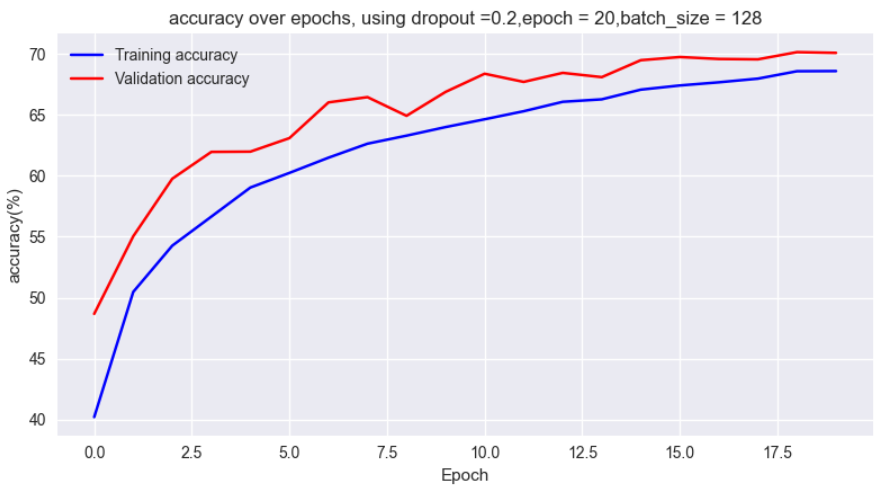
2.1.2 learning rate

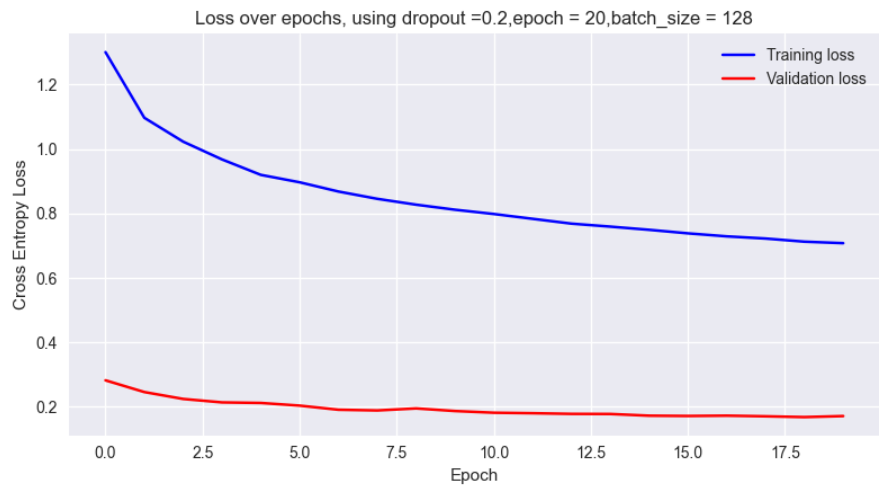
learning rate가 너무 작을 때는 학습 효과가 너무 낮았다. 20~30 epoch 기준으로 learning rate는 1e-03 일 때 가장 효율적이었다.

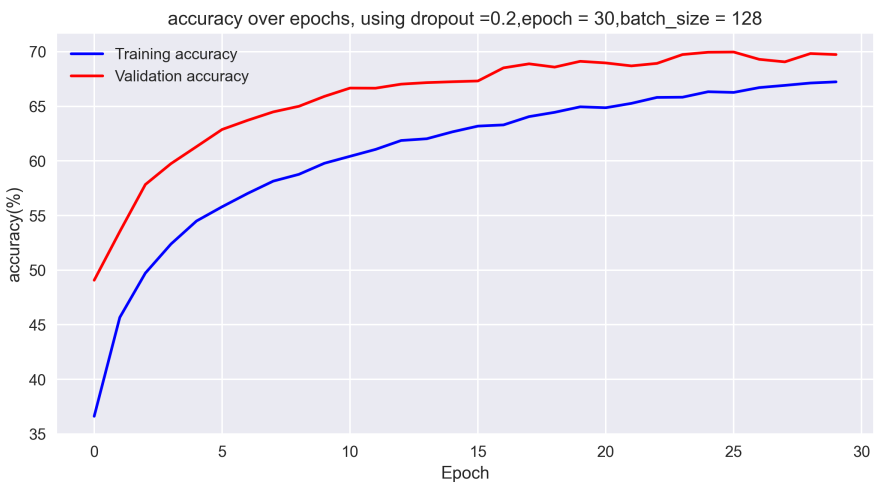
2.1.3 batch size

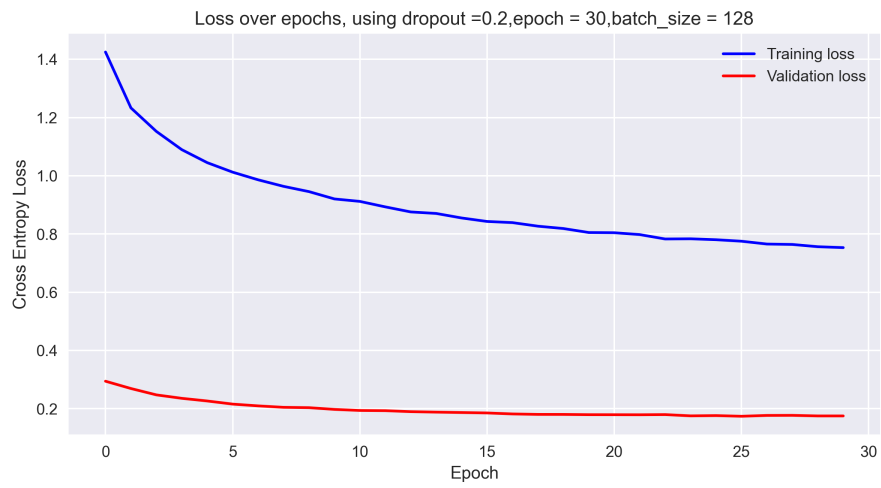
batch size는 캐싱 효율을 위해 2^n 꼴인 32,64,128,256으로 테스트하였다. dropout=0.2, epoch=30에서 128이 학습 속도 및 학습 결과면에서 가장 효과적이었다.

2.Batch normalization에 따른 학습 형태 변화









batch normalization을 적용한 경우(아래) 가 accuracy 면에서 적게 진동하였다.

3. Train의 전처리에 image processing 방법들을 적용할 경우

너무 큰 transform 값을 사용할 경우 학습 효율이 떨어졌다.

```
transform_train = transforms.Compose([transforms. RandomHorizontalFlip(),
                                      transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4),
                                      transforms.RandomAffine(30, shear=10, scale=(0.8,1.2)),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
                                      ])
```

```
09:36:33 --- Epoch: 11  Train loss: 1.2148  Valid loss: 0.2712  Train accuracy(%): 45.78    Valid accuracy(%): 52.19
09:37:12 --- Epoch: 12  Train loss: 1.2068  Valid loss: 0.2699  Train accuracy(%): 46.24    Valid accuracy(%): 52.50
09:37:44 --- Epoch: 13  Train loss: 1.2050  Valid loss: 0.2655  Train accuracy(%): 46.26    Valid accuracy(%): 53.43
09:38:14 --- Epoch: 14  Train loss: 1.1959  Valid loss: 0.2614  Train accuracy(%): 46.58    Valid accuracy(%): 53.88
```

```
transform_train = transforms.Compose([transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1),
                                      transforms.RandomAffine(30, shear=10, scale=(0.9,1.1)),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
                                      ])
```

```
10:20:10 --- Epoch: 24  Train loss: 1.0384  Valid loss: 0.2195  Train accuracy(%): 54.13    Valid accuracy(%): 61.45
10:20:37 --- Epoch: 25  Train loss: 1.0345  Valid loss: 0.2167  Train accuracy(%): 54.25    Valid accuracy(%): 62.14
10:21:04 --- Epoch: 26  Train loss: 1.0298  Valid loss: 0.2218  Train accuracy(%): 54.54    Valid accuracy(%): 61.66
10:21:32 --- Epoch: 27  Train loss: 1.0272  Valid loss: 0.2161  Train accuracy(%): 54.55    Valid accuracy(%): 62.29
10:22:00 --- Epoch: 28  Train loss: 1.0252  Valid loss: 0.2179  Train accuracy(%): 54.39    Valid accuracy(%): 61.15
10:22:28 --- Epoch: 29  Train loss: 1.0203  Valid loss: 0.2146  Train accuracy(%): 54.52    Valid accuracy(%): 62.63
10:22:55 --- Epoch: 30  Train loss: 1.0184  Valid loss: 0.2162  Train accuracy(%): 54.70    Valid accuracy(%): 62.46
```

이외에도 다양한 transform을 적용하였으나 학습 효율 면에서 크게 효과적이지 않았다. 적절한 transform parameter 값을 찾기는 어려웠다. transform을 적용할 경우 더 많은 epoch이 필요할 듯하다.