

AutoEncoder with CNN

수업시간 : 수 11:00 ~ 13:00 목 15:00 ~ 17:00
2018008559 컴퓨터소프트웨어학부 신상윤

0. Oxford flower 102 dataset

Oxford flower 102의 train dataset은 총 102종류의 꽃이 10장씩 있는 dataset 이다. 우리의 목표는 이 train dataset으로 autoencoder를 만들어 encoder, decoder를 학습시키고, 이후 encoder, classifier를 학습시켜 102종류의 꽃을 분류하는 분류기를 만드는 것이다.

1. AutoEncoder 구조, 코드

가. Encoder

input : (3,100,100) -> convolution -> (16,100,100) -> activation, max pooling2 -> (16,50,50) -> convolution -> (64,50,50) -> activation, max pooling2 -> (64,25,25) -> convolution -> (144,25,25) -> activation, max pooling3 -> (144,13,13) -> convolution -> (256,13,13) -> activation, max pooling3 -> (256,7,7) -> convolution -> (400,7,7) -> activation, max pooling3 -> (400,4,4)

convolution은 모두 1 padding 1 stride (3,3) kernel
maxpooling2 : no padding 2 stride (2,2) kernel
maxpooling3 : 1 padding 2 stride (3,3) kernel
activation : ReLU function

flatten을 하지 않는 이유 : flatten 이후 분류기를 CNN으로 학습시키기 위해서는 다시 flatten한 행렬을 reshape하고 convolution layer를 적용해야 한다. 이때 행렬의 feature를 제대로 추출하지 못 할것이라 판단하여 Encoder에서 flatten을 시키지 않았다.

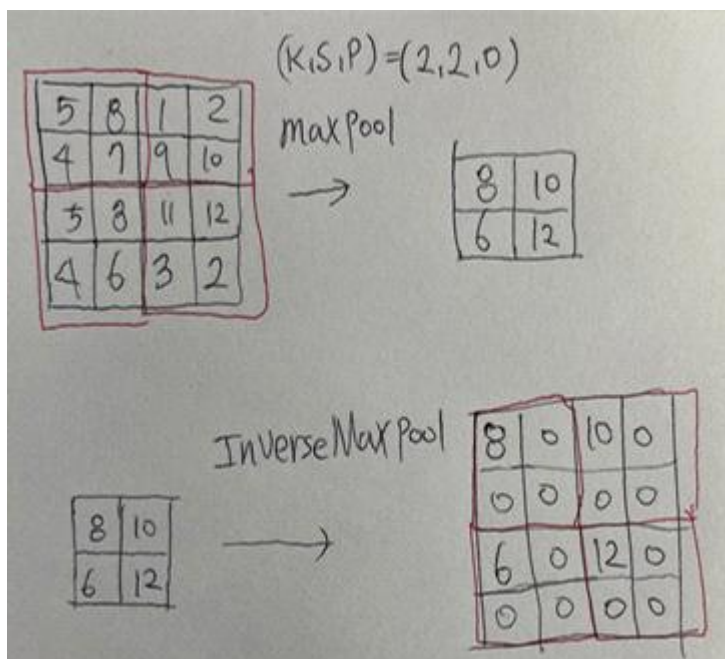
나. Decoder

input : (400,4,4) -> InverseMaxPool3 -> (400,7,7) -> deconvolution -> (256,7,7) -> activation -> InverseMaxPool3 -> (256,13,13) -> deconvolution -> (144,13,13) -> activation -> InverseMaxPool3 -> (144,25,25) -> deconvolution -> (64,25,25) -> activation -> InverseMaxPool2 -> (64,50,50) -> deconvolution -> (16,50,50) -> activation -> InverseMaxPool2 -> (16,100,100) -> deconvolution -> (3,100,100)

activation : ReLU function

1) InverseMaxPool

decoder를 구현하기 위해서는 maxpooling의 역연산(InverseMaxPool)을 구현할 필요가 있었다.



위 사진과 같이 kernel size 구역 안에 단순히 한 element만 최댓값을 채워주는 함수를 구현하고 InverseMaxPool의 역연산을 다시 살펴보면 결국 maxpooling 연산과 같은 것을 확인할 수 있다. 기능적으로 살펴보면 maxpooling은 결국 중요한 feature만 뽑아내는 과정이므로 역연산을 구현할 때 이 feature를 살리는 것이 가장 중요하다. 따라서 위와 같이 InverseMaxPool을 구현하는 것이 타당하다 할 수 있다.

2) Deconvolution

Convolution 연산이 결국 filter를 통과하여 새로운 feature를 새롭게 더 많이 만든 것이기 때문에 Deconvolution은 새롭게 만들되 더 적게 만들어 filter가 학습을 통해 Deconvolution의 역할을 수행하도록 기대했다. 따라서 Convolution 함수와 똑같지만 channel을 줄어드는 방향으로 조절하였다.

다. 코드

1) Encoder

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, 1, 1)
        self.conv2 = nn.Conv2d(16, 64, 3, 1, 1)
        self.conv3 = nn.Conv2d(64, 144, 3, 1, 1)
        self.conv4 = nn.Conv2d(144, 256, 3, 1, 1)
        self.conv5 = nn.Conv2d(256, 400, 3, 1, 1)

        self.pool2 = nn.MaxPool2d(2, 2, 0)
        self.pool3 = nn.MaxPool2d(3, 2, 1)

        self.activation = nn.ReLU()

        self.bn1 = nn.BatchNorm2d(16)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(144)
        self.bn4 = nn.BatchNorm2d(256)
        self.bn5 = nn.BatchNorm2d(400)

    def forward(self, x):
        x = self.pool2(self.activation(self.bn1(self.conv1(x))))
        x = self.pool2(self.activation(self.bn2(self.conv2(x))))
        x = self.pool3(self.activation(self.bn3(self.conv3(x))))
        x = self.pool3(self.activation(self.bn4(self.conv4(x))))
        x = self.pool3(self.activation(self.bn5(self.conv5(x))))
        #print(np.shape(x))
        return x
```

2) Decoder, Autoencoder

```
IMP2 = Inverse_MaxPool2().apply
IMP3 = Inverse_MaxPool3().apply

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()

        self.deconv1 = nn.Conv2d(400, 256, 3, 1, 1)
        self.deconv2 = nn.Conv2d(256, 144, 3, 1, 1)
        self.deconv3 = nn.Conv2d(144, 64, 3, 1, 1)
        self.deconv4 = nn.Conv2d(64, 16, 3, 1, 1)
        self.deconv5 = nn.Conv2d(16, 3, 3, 1, 1)

        self.bn1 = nn.BatchNorm2d(256)
        self.bn2 = nn.BatchNorm2d(144)
        self.bn3 = nn.BatchNorm2d(64)
        self.bn4 = nn.BatchNorm2d(16)

        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.activation(self.bn1(self.deconv1(IMP3(x))))
        x = self.activation(self.bn2(self.deconv2(IMP3(x))))
        x = self.activation(self.bn3(self.deconv3(IMP3(x))))
        x = self.activation(self.bn4(self.deconv4(IMP2(x))))
        x = self.deconv5(IMP2(x))
        return x
```

InverseMaxPool을 구현하고 생성자로 선언하여 사용하였다.

```
class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, input):
        z = self.encoder(input)
        x_hat = self.decoder(z)
        return z, x_hat
```

Encoder와 Decoder를 합친 AutoEncoder이다.

3) InverseMaxPool

```
class Inverse_MaxPool2(torch.autograd.Function):
    def __init__(self):
        super(Inverse_MaxPool2, self).__init__()

    def forward(self, x_in):
        x_in_dim = x_in.size() #ex) x_in_dim = [128, 64, 50, 50]
        x_out = torch.zeros((x_in_dim[0],
                             x_in_dim[1],
                             x_in_dim[2]*2,
                             x_in_dim[3]*2)).to(device) #ex) x_out_dim = [128, 64, 100, 100]
        for k in range(x_in_dim[2]):
            for l in range(x_in_dim[3]):
                x_out[:, :, 2*k:2*k+1, 2*l:2*l+1] = x_in[:, :, k:k+1, l:l+1] #inverse MaxPool

        return x_out

    def backward(self, x_in):
        x_in_dim = x_in.size() #ex) x_in_dim = [128, 64, 100, 100]
        x_out = torch.zeros((x_in_dim[0],
                             x_in_dim[1],
                             int((x_in_dim[2])/2),
                             int((x_in_dim[3])/2))).to(device) #ex) x_out_dim = [128, 64, 50, 50]

        for k in range(int((x_in_dim[2])/2)):
            for l in range(int((x_in_dim[3])/2)):
                x_out[:, :, k:k+1, l:l+1] = x_in[:, :, 2*k:2*k+1, 2*l:2*l+1] #inverse MaxPool

        return x_out
```

forward 함수는 정의한 것처럼 kernel이 (2,2)일때는 왼쪽 위 위치에 (3,3)일때는 정 가운데에 Max값이 들어가고 나머지는 0으로 채우도록 정의했다. 먼저 모든 element가 0인 x_out을 만들고, x_in에서의 값들이 모두 Max값이 되도록 대입했다. InverseMaxPool2와 3의 차이는 kernel의 크기이며 (2,2) 일때는 x_out의 좌표가 x_in 좌표의 정확히 2배가 되고, (3,3) 일때도 x_out의 좌표가 x_in 좌표의 정확히 2배가 된다. 이는 encoder에서 정의할 때 두 경우 모두 stride를 2로 했기 때문이다.

forward 함수만 정의하고 실행하여도 torch가 알아서 계산을 해주기는 하는데, 1 epoch가 너무 느려(10분) torch.autograd.Function로 직접 backward를 정의해 주었다. update 되는 weight는 없으므로 단순히 역으로 대입해주면 된다. 예를 들어 kernel이 (2,2) 일때는 x_out의 좌표가 x_in 좌표의 정확히 1/2배가 된다.

2. Classifier 구조, 코드

input : (400,4,4) -> convolution -> (700,4,4) -> activation ->
convolution -> (1000,4,4) -> activation -> flatten -> 16000 -> linear
-> 8192 -> activation, linear-> 2048 -> activation, linear -> 512 ->
activation, linear -> 102

activation : ReLU function

flatten : nn.linear를 적용하기 위해 사용하였다.

maxpooling이 없는 이유 : CNN 구조로 작성하기 위해 처음에는 reshape 후 Encoder처럼 channel을 늘리고 maxpooling을 했었는데 reshape 후 추가로 가중치들의 개수가 계속 변하니 학습이 잘 안 되어서 아예 maxpooling을 제외했다. 또한 feature를 뽑아주는 Encoder와는 다르게 Classifier에서는 특성을 종합하여 분류하는 것이 목적이기 때문에 적당히 feature를 늘리고 full connected layer로 분류하는 것이 성능이 더 좋을 것이라 생각했다.

```
##### Classifier 모델 코드 #####

class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.conv1 = nn.Conv2d(400, 700, 3, 1, 1)
        self.conv2 = nn.Conv2d(700, 1000, 3, 1, 1)

        self.linear1 = nn.Linear(16*1000, 8192)
        self.linear2 = nn.Linear(8192, 2048)
        self.linear3 = nn.Linear(2048, 512)
        self.linear4 = nn.Linear(512, 102)

        self.bn1 = nn.BatchNorm2d(700)
        self.bn2 = nn.BatchNorm2d(1000)
        self.bn3 = nn.BatchNorm1d(8192)
        self.bn4 = nn.BatchNorm1d(2048)
        self.bn5 = nn.BatchNorm1d(512)

        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.activation(self.bn1(self.conv1(x)))
        x = self.activation(self.bn2(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = self.activation(self.bn3(self.linear1(x)))
        x = self.activation(self.bn4(self.linear2(x)))
        x = self.activation(self.bn5(self.linear3(x)))
        x = self.linear4(x)
        return x
```


최종적으로 102개의 값을 구하고, CrossEntropyLoss를 통해 softmax로 102종류의 꽃의 확률을 나타내고 분류를 할 수 있도록 하였다.

3. AutoEncoder, Classifier 학습 코드

```
##### AutoEncoder 학습 코드 #####
sample = train_dataset[1019][0].view(1,3,100,100).to(device)

autoencoder = AutoEncoder().to(device)
optimizer = optim.Adam(autoencoder.parameters(), lr = 0.001)
criterion = nn.MSELoss()

epochs = 100

autoencoder.train()
for epoch in range(epochs):
    autoencoder.train()
    avg_cost = 0
    total_batch_num = len(train_dataloader)

    for b_x, b_y in train_dataloader:
        b_x = b_x.to(device)
        z, b_x_hat = autoencoder(b_x)
        loss = criterion(b_x_hat, b_x)

        avg_cost += loss / total_batch_num
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print('Epoch : {} / {}, cost : {}'.format(epoch+1, epochs, avg_cost))
```

AutoEncoder 이기 때문에 MSE loss를 사용하였고, learning rate는 0.001을 사용하였다.


```

if (epoch+1)%5 == 0:
    autoencoder.eval()
    fig, ax = plt.subplots(1,2)
    with torch.no_grad():
        test_z, test_output = autoencoder(sample)
    ax[0].set_title('x')
    ax[1].set_title('x_hat')

    ax[0].set_axis_off()
    ax[1].set_axis_off()
    ax[0].imshow(np.transpose(np.reshape(sample.cpu(),(3,100,100)),(1,2,0)))
    ax[1].imshow(np.transpose(np.reshape(test_output.cpu(),(3,100,100)),(1,2,0)))
    plt.show()

```

학습을하면서 AutoEncoder의 변화를 보고싶었으므로 epoch+1이 5의 배수일때마다 원본 x와, AutoEncoder를 통과후 나오는 x_hat을 비교하여 보여주도록 하였다.

```

##### Classifier 학습 코드 #####

classifier = Classifier().to(device)
autoencoder = AutoEncoder().to(device)
cls_criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(
    [
        {"params": autoencoder.parameters(), "lr": 0.001},
        {"params": classifier.parameters(), "lr": 0.001}
    ]
)
autoencoder.train()
classifier.train()
total_batch_num = len(train_dataloader)

epochs = 40

for epoch in range(epochs):
    autoencoder.train()
    classifier.train()
    avg_cost = 0

    for b_x, b_y in train_dataloader:
        b_x = b_x.to(device)
        z, b_x_hat = autoencoder(b_x)
        logits = classifier(z)
        loss = cls_criterion(logits, b_y.to(device))

        avg_cost += loss / total_batch_num
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print('Epoch : {} / {}, cost : {}'.format(epoch+1, epochs, avg_cost))

```

softmax 함수를 쓰므로 CrossEntropyLoss를 사용하였고, AutoEncoder, Classifier 모두 learning rate는 0.001을 사용하였다.

4. Classifier 정확도 측정 코드

```
##### Classifier 정확도 측정 코드 #####

correct = 0
total = 0
classifier.eval()
autoencoder.eval()
for b_x, b_y in train_dataloader:
    b_x = b_x.to(device)
    with torch.no_grad():
        z, b_x_hat = autoencoder(b_x)
        logits = classifier(z)
        predicts = torch.argmax(logits, dim = 1)

    total += len(b_y)
    correct += (predicts == b_y.to(device)).sum().item()

print(f'Accuracy of the network on train images: {100 * correct / total} %')

correct = 0
total = 0
for b_x, b_y in test_dataloader:
    b_x = b_x.to(device)
    with torch.no_grad():
        z, b_x_hat = autoencoder(b_x)
        logits = classifier(z)
        predicts = torch.argmax(logits, dim = 1)

    total += len(b_y)
    correct += (predicts == b_y.to(device)).sum().item()

print(f'Accuracy of the network on test images: {100 * correct / total} %')
```

train_set과 test_set에 대한 정확도를 모두 출력하도록 하여 과적합인지, 학습이 부족한지 등등을 확인했다.

5. 실험결과, 성능비교

가. Batch Normalization(이하 BN)

autoencoder, classifier 모두 30epoch씩 실행

BN O	1	2	3	4	5
정확도 (train set)	93.24%	98.43%	99.6%	100%	95.29%
정확도 (test set)	23.27%	26.65%	26.65%	32.27%	23.96%
결린시간(초)	288.3	285.3	286.1	287.9	290.8

BN X	1	2	3	4	5
정확도 (train set)	3.2%	6.5%	6.47%	4.41%	0.98%
정확도 (test set)	2.47%	3.4%	3.24%	3.3%	1.06%
결린시간(초)	289.17	286.89	286	285.4	283.6

코드 실행시간은 비슷했지만, cost의 수렴 속도가 BN을 사용하였을 때 압도적으로 빨랐다. 즉 적은 epoch로도 고효율을 보여주었다. BN을 사용하지 않을 때도 epoch만 늘린다면 정확도는 계속 오를 것이다.

나. Optimizer

autoencoder, classifier 모두 30epoch씩 실행, BN적용, 5번씩 실행

1) SGD

정확도 (train set)	100%	100%	100%	100%	100%
정확도 (test set)	14.9%	15.12%	15.03%	15.29%	15%

2) Adagrad(best!)

정확도 (train set)	100%	100%	100%	100%	100%
정확도 (test set)	36.98%	36.84%	36.87%	37.42%	37.63%

3) RMSprop

정확도 (train set)	30.78%	19.21%	27.65%	25.69%	47.35%
정확도 (test set)	11.09%	8.93%	9.46%	8.52%	14.2%

4) Adam

정확도 (train set)	100%	97.45%	99.11%	99.7%	98.33%
정확도 (test set)	30.49%	25.45%	27.65%	28.77%	27.11%

Adam optimizer가 가장 좋을 줄 알았지만 의외로 Adagrad optimizer가 가장 정확도가 높았다.

RMSprop가 성능이 가장 안 좋았는데 step size와 방향 모두 계속 잘못 잡고 있는 것 같았다.

SGD는 train set에서는 잘 맞고, test set에서 성능이 부족한 것으로 보아 step size가 많이 작아진 것 같다.

마지막으로 Adam과 Adagrad를 비교해보자면 Adagrad가 가장 잘 맞는 방향으로 큰 step size로 잘 이동하여 빠르게 잘 수렴한 것 같다.

다. Regularization

autoencoder, classifier 모두 30epoch씩 실행, BN적용,

Adagrad 적용, 5번씩 실행

위의 결과를 보면 train set에서는 100%, test set에서는 40% 이하의 정확도를 보여주었다. 따라서 자연스럽게 model이 과적합 되어있다고 생각하여 정규화(Regularization)를 시도해보았다.

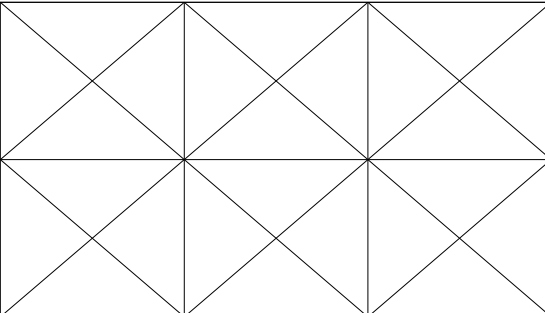
1) Dropout(drop_prob = 0.25)

정확도 (train set)	100%	100%	100%	100%	100%
정확도 (test set)	37.62%	38.07%	37.96%	37.76%	38.1%

2) Dropout(drop_prob = 0.5)

정확도 (train set)	100%	100%	100%	100%	100%
정확도 (test set)	35.14%	35.63%	35.22%	35.08%	35.65%

3) Dropout(drop_prob = 0.75)

정확도 (train set)	32.35%	16.08%			
정확도 (test set)	13.4%	9.6%			

4) L2 Regularization(lambda = 0.007)

정확도 (train set)	100%	100%	100%	100%	100%
정확도 (test set)	37.44%	37.65%	37.73%	37.29%	37.4%

5) L2 Regularization($\lambda = 0.01$)

정확도 (train set)	100%	100%	100%	100%	100%
정확도 (test set)	37.52%	37.66%	37.12%	37.43%	37.58%

L2 Regularization, dropout 모두 최대 1% 정도 정확도가 상승하였다. 결국 Regularization을 통해 극적인 변화는 없었는데, 가장 큰 이유는 train dataset이 1020개로 매우 data가 작아 과적합될 수밖에 없었기 때문이다. 물론 약간의 상승량은 있었지만 가장 좋은 방법은 train data를 늘리는 것이다.

라. 최종 Model, 결과

autoencoder 40epoch, classifier 40epoch 실행, BN적용
40epoch 이상부터는 큰 cost 변화가 없었기 때문에 40epoch로 설정

drop_prob = 0.25 적용

autoencoder학습에 Adagrad를 적용했을 때 실제 학습과정을 사진으로 봤는데 Adam을 썼을때보다 형태가 불완전했다. 따라서 autoencoder 학습에는 Adam을 사용하고 classifier 학습에는 Adagrad를 사용했다.

```
Accuracy of the network on train images: 100.0 %  
Accuracy of the network on test images: 41.16116441697837 %
```

6. Reference

[1] https://tutorials.pytorch.kr/beginner/examples_autograd/polynomial_custom_function.html