

HW 05 – REPORT

소속 : 정보컴퓨터공학부

학번 : 201925111

이름 : 김건호

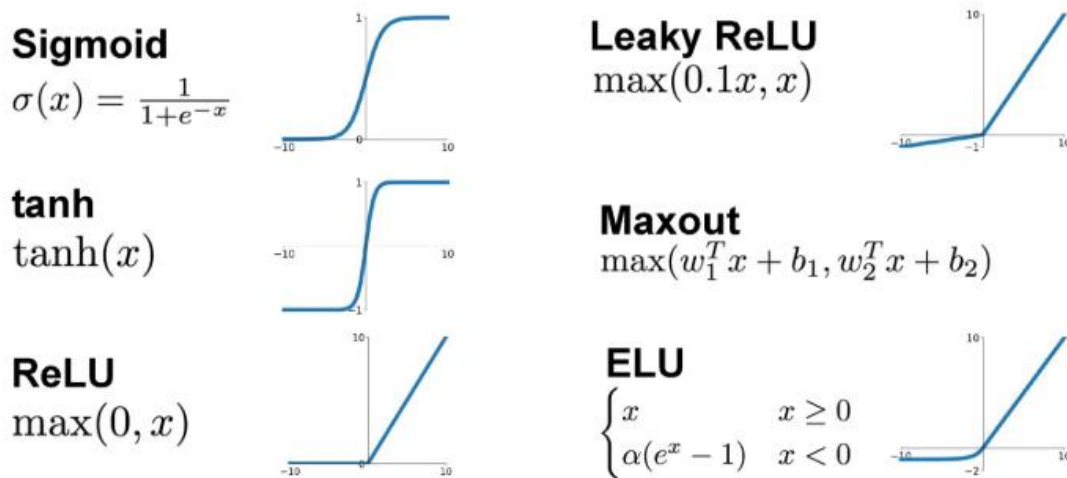
1. 서론

1. 실습 목표

- 1) pytorch 사용 방법을 익힐 수 있다.
- 2) Pytorch를 사용해서 convolution neural net를 구현할 수 있다.
- 3) pytorch를 사용해서 CIFAR-10 dataset을 classification하는 모델을 구현할 수 있다.

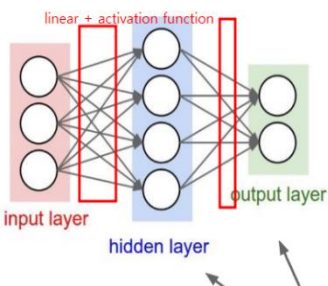
2. 이론적 배경

Neural Network란 인간의 신경(neuron)을 모방해서 만든 network입니다. 각 뉴런은 linear function에 대응됩니다. input에 대한 뉴런의 output이 신호의 활성화를 일으키는지를 판단하는 function이 activation function입니다. 입력 x 에 대해, linear function $f = Wx$ 를 지난 결과가 activation function의 새로운 input으로 들어갑니다. 아래는 많이 사용되는 activation function입니다.



sigmoid는 확률 값을 나타냅니다. ReLU function은 값이 0보다 작으면 0, 0보다 크면 자기 자신을 출력하는 activation function입니다. 2 layer – neural network를 예시로 들어보겠습니다.

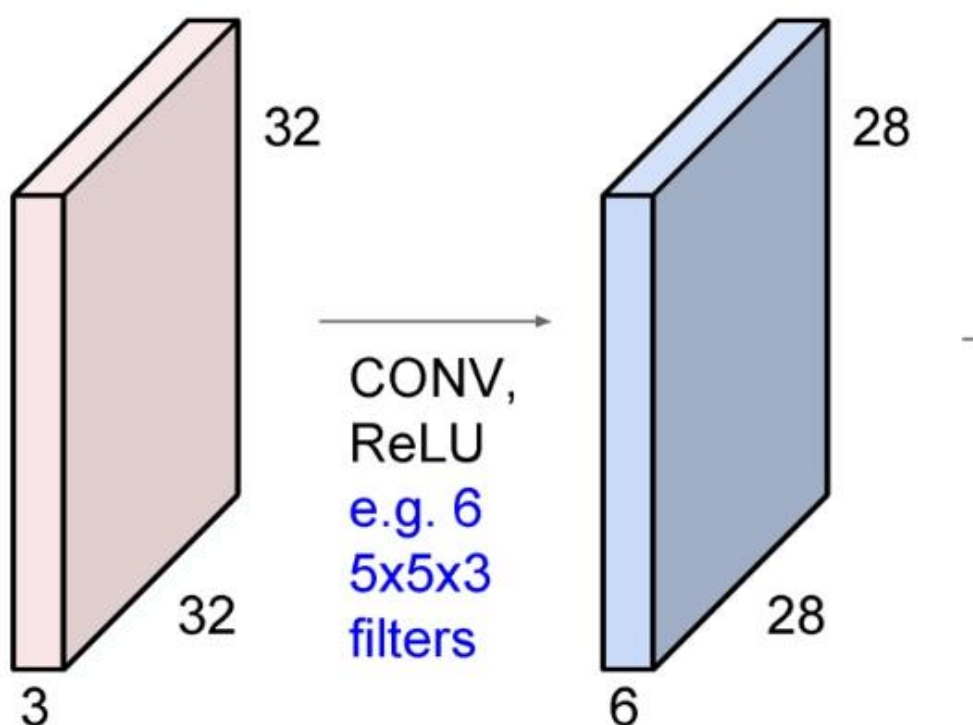
linear function과 activation function (선택사항)을 1 layer이라고 표현합니다. 옆 network는 총 2



layer를 가졌습니다. 모든 입력 node가 모든 출력 node에 영향을 주는 network를 fully – connected layer network라고 합니다. 옆 network는 fully – connected layer이자, 2 layer – neural network입니다. 각 linear function은 입력에 대한 weight의 matrix로 이루어져 있습니다. 적절한 weight를 찾는 것이, model을 학습하는 과정입니다.

적절한 weight를 찾는 방법에 대해 알아보겠습니다. gradient descent(경사하강법)을 많이 사용합니다. 처음엔 random하게 weight를 초기화합니다. 또한 이에 대한 loss를 계산합니다. loss는 아래에서 자세하게 이야기하겠습니다. 계산된 loss를 미리 설정된 learning rate를 사용해서, loss를 줄이는 방향으로 weight를 update합니다. weight 변화가 거의 없는 지점까지 반복합니다.

위 과정에서 linear function을 사용하지 않고, 대신 사전에 정의된, filter(=kernel)를 사용해서 convolution하는 network를 CNN(convolution neural network)라고 합니다. CNN에서 input은 3개의 channel(R,G,B)로 구성됩니다. filter(=kernel)는 보통 가로, 세로의 크기가 같습니다. 또한, filter의 수가 n개 있다면, output의 channel 수도 n일 것입니다. 아래는 activation function을 ReLU로 설정한 후, 6개의 5x5x3 filter를 사용해 32x32x3 image를 convolution한 결과를 보여주는 예입니다.



filter 내부 적절하게 generalization된 값을 찾는 것이, 좋은 모델을 만드는 것입니다. 하지만 학습 data에 너무 기반하여, filter 내부 값을 찾는다면, 이는 모델이 학습 data만 의존하게 되는, 즉 overfitting model이 만들어질 수 있습니다. 따라서 이를 보완하고자 Regularization(정규화)을 합니다. 이는 loss function을 기반으로 진행합니다. 만약 weight가 너무 크다면, 즉 이는 모델이 너무 복잡함을 의미하고, 이는 모델이 overfitting되었다고 판단합니다. 따라서 loss function을 아래와 같이 정의할 수 있습니다. 이를 바탕으로 model을 Regularization합니다.

$$L = L_{\text{data}} + L_{\text{reg}} \qquad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$

2. 본론

Question 1 : Implement the forward pass for the three-layer ConvNet

첫 번째 layer는 zero padding이 2인, 32개의 5x5 filter를 convolution한 후, activation function으로 ReLU를 사용하는 layer입니다. 두 번째 layer는 zero padding이 1인, 16개의 3x3 filter를 convolution한 후, activation function으로 ReLU를 사용하는 layer입니다. 마지막 layer는 bias를 포함한, 10개의 class에 대한 score를 계산하는 Fully – connected layer입니다.

이를 구현하기 위해, torch.nn.functional을 사용합니다. 두 번째 layer의 결과를 fully connected layer의 input으로 넣기 위해, flatten function을 사용하고, pytorch의 matrix multiplication을 할 수 있는, mm method를 사용한 후, bias를 더해주었습니다. 아래는 three_layer_convnet_test()의 결과입니다.

```
[8] def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype) # minibatch size 64, image size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype) # [out_channel, in_channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,)) # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype) # [out_channel, in_channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,)) # out_channel

    # you must calculate the shape of the tensor after two conv layers, before the fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b])
    print(scores.size()) # you should see [64, 10]
    three_layer_convnet_test()

torch.Size([64, 10])
```

Question 2 : Initialize the parameters of a three-layer ConvNet.

첫 번째 layer는 zero padding이 2인, 32개의 5x5 filter를 convolution한 후, activation function으로 ReLU를 사용하는 layer입니다. 두 번째 layer는 zero padding이 1인, 16개의 3x3 filter를 convolution한 후, activation function으로 ReLU를 사용하는 layer입니다. 마지막 layer는 bias를 포함한, 10개의 class에 대한 score를 계산하는 Fully – connected layer입니다. 이러한 architecture의 convolution network를 구현하기 위해, weight matrix를 random_weight function을 사용하여 초기화해주고, bias matrix를 zero_weight function을 사용하여 초기화해줍니다. 두 함수는 parameter로 shape라는 tuple을 사용합니다. 이는 (출력 채널 수이자 convolution layer의 필터 수, 입력 이미지의 채널 수, 초기는 RGB이므로 3, filter의 row 크기, filter의 col 크기) 형태를 가집니다.

따라서, 첫 번째 convolution layer는 (convolution weight는 32개의 출력이자 32개의 필터, 3개의 RGB 채널, filter의 row size=5, filter의 col size=5)인 shape를 갖고, 두 번째는 convolution layer는 (convolution weight는 16개의 출력이자 16개의 필터, 32개의 입력 채널 수, filter의 row size=3, filter의 col size=3)인 shape를 갖습니다. 또한 첫 번째 convolution layer는 32개의 channel을 출력하므로, bias shape는 (32,)를 갖고, 두 번째 bias shape는 (16,)을 갖습니다.

마지막 layer인 fully connected layer는 16개의 출력 채널을 가지며, 32x32의 matrix에 대한 32x32x16개의 weight가 필요하고, output으로 10개의 class에 대해 score를 측정해야 하므로, shape는 (32x32x16, 10)으로 설정하였습니다. 또한 10개의 출력에 대해 각각의 bias가 필요하므로 bias shape는 (10,)으로 설정하였습니다.

```
conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_w2 = random_weight((channel_2, channel_1, 3, 3))

conv_b1 = zero_weight((channel_1,))
conv_b2 = zero_weight((channel_2,))

fc_w = random_weight((32 * 32 * 16, 10))
fc_b = zero_weight((10, ))
```

Question 3 : Set up the layers you need for a three-layer ConvNet with the architecture defined above. Question 4 : Implement the forward function for a 3-layer ConvNet. you should use the layers you defined in `__init__` and specify the connectivity of those layers in `forward()`.

3 layer convnet의 class와 forward function을 만드는 것이 목적입니다. 이는 torch.nn의 Conv2d method를 사용하여 구현할 수 있습니다. nn.init.kaiming_normal_의 parameter로 convnet의 function을 사용할 수 있습니다. 이는 신경망의 초기 가중치를 적절하게 초기화해주는 method입니다.

```
self.conv1 = nn.Conv2d(in_channel, channel_1, 5, padding=2)
self.conv2 = nn.Conv2d(channel_1, channel_2, 3, padding=1)
self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)

nn.init.kaiming_normal_(self.conv1.weight)
nn.init.kaiming_normal_(self.conv2.weight)
nn.init.kaiming_normal_(self.fc.weight)
```

forward function은 위를 활용하여, forwarding을 하는 함수이므로, 아래와 같이 구현할 수 있습니다.

```
scores = F.relu(self.conv1(x))
scores = F.relu(self.conv2(scores))
scores = self.fc(flatten(scores))
```

아래는 결과입니다.

```
def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype) # minibatch size 64, image size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes=10)
    scores = model(x)
    print(scores.size()) # you should see [64, 10]
test_ThreeLayerConvNet()

torch.Size([64, 10])
```

Question 5 : Instantiate your ThreeLayerConvNet model and a corresponding optimizer

위 threeLayerConvNet class를 optimization하는 부분입니다. 이를 위해 optim.SGD method를 사용하였습니다. 이는 parameter로 model의 parameter를 첫 번째 매개변수로 가지고, 두 번째로 learning rate를 갖습니다. momentum은 0으로 설정하였습니다. momentum이란 이전 단계의 gradient를 적절히 반영하여, 현재 단계의 기울기에 추가하는 hyperparameter입니다.

아래는 구현 결과입니다.

```
model = ThreeLayerConvNet(in_channel=3, channel_1 = 32, channel_2 = 16, num_classes=10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                               END OF YOUR CODE                               #
#####

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.3036
Checking accuracy on validation set
Got 127 / 1000 correct (12.70)
```

```
Iteration 100, loss = 1.8361
Checking accuracy on validation set
Got 368 / 1000 correct (36.80)
```

```
Iteration 200, loss = 1.9236
Checking accuracy on validation set
Got 391 / 1000 correct (39.10)
```

```
Iteration 300, loss = 1.7893
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)
```

```
Iteration 400, loss = 1.6868
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)
```

```
Iteration 500, loss = 1.5876
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)
```

```
Iteration 600, loss = 1.7067
Checking accuracy on validation set
Got 475 / 1000 correct (47.50)
```

```
Iteration 700, loss = 1.7773
Checking accuracy on validation set
Got 472 / 1000 correct (47.20)
```

Question 6 : Rewrite the 2-layer ConvNet with bias from Part III with the Sequential API.

2-layer ConvNet을 nn.Sequential API를 사용해서 재작성하는 부분입니다. nn.Sequential API는 여러 layer를 쉽게 순차적으로 연결할 수 있는 API입니다. momentum을 0.9로 설정하였습니다.

아래는 구현 결과입니다.

```
model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                       momentum=0.9, nesterov=True)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                               END OF YOUR CODE                               #
#####

train_part34(model, optimizer)

Iteration 0, loss = 2.3029
Checking accuracy on validation set
Got 170 / 1000 correct (17.00)

Iteration 100, loss = 1.8585
Checking accuracy on validation set
Got 377 / 1000 correct (37.70)

Iteration 200, loss = 1.8157
Checking accuracy on validation set
Got 381 / 1000 correct (38.10)

Iteration 300, loss = 1.6115
Checking accuracy on validation set
Got 402 / 1000 correct (40.20)

Iteration 400, loss = 1.9049
Checking accuracy on validation set
Got 419 / 1000 correct (41.90)

Iteration 500, loss = 1.7511
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)

Iteration 600, loss = 1.9490
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Iteration 700, loss = 1.6900
Checking accuracy on validation set
Got 424 / 1000 correct (42.40)
```

Question 7: Experiment with any architectures, optimizers, and hyperparameters. Achieve AT LEAST 70% accuracy on the validation set within 10 epochs. Note that you can use the `check_accuracy` function to evaluate on either the test set or the validation set, by passing either `loader_test` or `loader_val` as the second argument to `check_accuracy`. You should not touch the test set until you have finished your architecture and hyperparameter tuning, and only run the test set once at the end to report a final value.

10 epochs 내에서, validation set의 정확도를 70% 달성하면 됩니다. epoch는 모델 학습의 반복 횟수를 조절하는 parameter인데, 10으로 설정한다면, 10번 반복한다는 의미입니다. epoch가 늘어나면 모델의 정확도는 향상되지만, 이는 overfitting을 야기하고, 또한 학습 시간이 길어진다는 단점이 있습니다. overfitting 상태를 방지하기 위해, MaxPool2d와 Dropout을 사용하였습니다. dropout은 overfitting을 방지하기 위해, parameter 비율만큼 node를 drop하는 방식입니다. 또한 activation function은 ReLU function을 적용하였습니다. 또한, optimizer로 Adam optimizer를 사용하였습니다.


아래는 model 구현입니다.

```
model = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),
    nn.Conv2d(64, 128, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),
    nn.Flatten(),
    nn.Linear(128 * 4 * 4, 256),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(256, 10)
)

# Initialize the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

아래는 최종 결과입니다.

```
[ ] best_model = model
    check_accuracy_part34(loader_test, best_model)
```

 Checking accuracy on test set
Got 7596 / 10000 correct (75.96)

3. 결론

좋은 모델을 만들기 위해 필수적인 것은 overfitting을 막는 것입니다. 하지만 좋은 모델을 만들기 위해, data를 preprocessing하는 것도 중요합니다. 두 번째로, 어떤 architecture를 사용할지 정합니다. 보통 image classification problem에선 ResNet을 사용합니다. architecture를 결정하였다면, filter 내부 적절한 값을 찾기 위해 보통 먼저 random하게 값을 정한 후, 학습을 진행하면서, 적절한 값으로 update합니다. 또한, 각 linear function의 output에 대해 적절한 scale로 변환해주는 과정을 Normalization(정규화)이라고 하는데, 대표적으로 batch normalization이 존재합니다. 이는 평균은 0, 분산은 1인 데이터의 분포로 scaling해주는 기법입니다. 이후 output을 activation function의 input으로 사용합니다. 이를 마지막 layer까지 반복합니다.

이 때, overfitting을 막기 위해, Regularization을 진행합니다. 위 실습에선, torch.optim module의 SGD(경사하강법) method와 Adam method를 사용했습니다. 이를 통해, 적절한 weight를 계산할 수 있습니다. Adaptive Moment Estimation (Adam) optimizing은 현재 weight의 gradient 평균과, 제곱의 지수 평균을 사용해서 weight를 update하는 방식입니다. 또한, weight로 학습률을 조절하여 weight를 update하는 Adagrad 방식도 존재합니다. 이러한 방식으로 overfitting을 막을 수 있습니다. 또한 dropout을 통해 random하게 특정 node를 고려하지 않고 학습할 수도 있습니다.