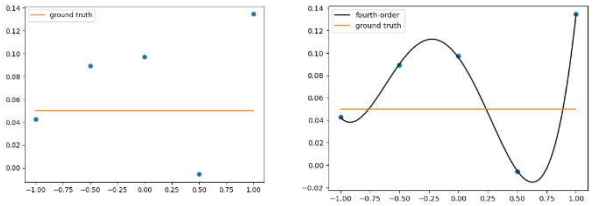
Summary Report [Week 10]. Regularization

20190106 KimByungjun (김병준)

● Exercises

이번 Exercise 에서는 Regularization(정규화) 중 L1, L2, dropout 정규화를 구현해보고자 한다. 정규화는 **overfitting 을 방지**하며, 일반화 성능을 높이는 데 효과적인 기법이다.

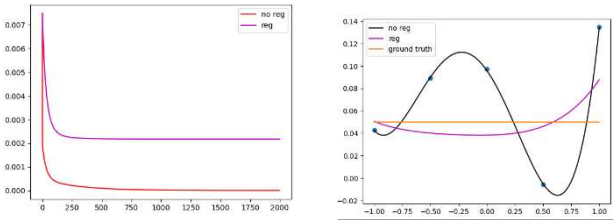
우선 우측 **[Fig 1-1]** 5 개의 점은 ground truth: y=0.05 의 기준선에서 noise(0.05×torch.randn(size=5))를 적용한 dataset 이다.



[Fig 1-1] (Left) Dataset for regression

[Fig 1-2] (Right)4th order – Polynomial Regression model

해당 dataset 에 대해 4 차 함수의 polynomial-regression 이 적용된 model 을 보면(**[Fig 1-2]**) 주어진 5 개의 점을 함수가 잘 통과하는 걸 보니 training 이 잘 되었다고 볼 수 있지만, 이 경우 해당 4 차 함수를 벗어나는 추가 data 가 주어지는 경우, model 이 잘 설명할 수 없다는 치명적인 단점이 생긴다. 여기서, training 에 쓰이는 SGD optimizer 의 weight_decay argument 에 값을 부여하여 L2 정규화를 적용하게 될 경우 **[Fig 2-1], [Fig 2-2]**와 같이 training loss 와 model 이 4th order model 과 대비된다. 이 때, 정규화가 적용된 그래프가 overfitting 을 훨씬 더 완화시킨 것을 알 수 있으며 ground truth 와도 보다 더 근접해졌음을 알 수 있으며, 대신 training loss 는 정규화가 없는 model 에 비해 상대적으로 높은 수치에 수렴해가는 것을 알 수 있다.



[Fig 2-1] (Left) Training loss with regularization (Purple)

[Fig 2-2] (Right) model with regularization (Purple)

아래는 Titanic dataset 에 대한 Binary classification model 구현 중 정규화를 적용시켰을 때의 영향을 알아보기 위한 Exercises 및 Implementation 이다. 이 때, 정규화의 영향을 보다 더 확실하게 파악하기 위해 training dataset 의 크기가 작은 (size = 44) 상황을 부여하였다.

1) Define functions compute_l1_norm and compute_l2_norm.

<Explanation>

L1, L2 정규화란 기존 loss function 에 각각의 penalty term 을 더해주는 방식으로, 해당 예제에서는 $\lambda\Sigma_i|\theta_i|$, $\frac{\lambda}{2}\Sigma_i\theta_i^2$ 을 더하는 것으로 한다. 이 때, $\Sigma_i|\theta_i|$, $\Sigma_i\theta_i^2$ 는 l_p-norm $\|x_p\| = (\sum_{i=1}^{n}|x_i|^p)^{\frac{1}{p}}$ 의 형태에서 비롯되었으며 **$\Sigma_i|\theta_i|$, $\sqrt{\Sigma_i\theta_i^2}$ 를 각각 l_1-norm, l_2-norm** 이라고 한다. **[Fig 3]** 에서는 해당 norm 들을 계산하는 함수를 구현한 것이며, torch.tensor 에 대한 연산임에 유의하여, function parameter 인 tensor w 에 대해 l_1-norm 은 torch.sum(torch.abs(w))을, l_2-norm 은 torch.sqrt(torch.sum(w**2))을 반환하도록 구현하였다.

```
class NN(nn.Module):
  def __init__(self):
    super(NN,self).__init__()
    self.fc1 = nn.Linear(5,16)
    self.fc2 = nn.Linear(16,8)
    self.fc3 = nn.Linear(8,1)

  def forward(self,x):
    x = self.fc1(x)
    x = nn.ReLU()(x)
    x = self.fc2(x)
    x = nn.ReLU()(x)
    x = self.fc3(x)
    return x

  def compute_l1_norm(self, w):
    ######## Task 1 ########
    return torch.sum(torch.abs(w))
    ########################

  def compute_l2_norm(self, w):
    ######## Task 1 ########
    return torch.sqrt(torch.sum(w**2))
    ########################
```

[Fig 3] Code for Exercise 1

정규화가 없을 때의 classification 결과를 먼저 살펴보자. **[Fig 4]**와 같이 Loss function 은 cross entropy loss 를, optimizer 는 Adam(Adaptive Moment Estimation) Optimizer 를 사용하였다. Test accuracy 는 0.6536 정도가 나왔다.

```
[116] # No regularization

regmodel=NN() #regression model
regmodel.to(torch.float32)

# Define your loss function
criterion = nn.CrossEntropyLoss()

# Create an optimizer
#optimizer = optim.SGD(regmodel.parameters(), lr=0.01)
optimizer = optim.Adam(regmodel.parameters(), lr=0.01)

num_epochs = 3000

# Training loop
for epoch in range(num_epochs):
    optimizer.zero_grad()  # Zero the gradients

    out = regmodel(X_train_tensor)                    # Forward pass
    p=torch.cat((torch.zeros_like(out),out),dim=1)    # binary probability distribution
    loss = criterion(p, y_train_tensor)               # Compute the loss

    loss.backward()       # Backpropagate to compute gradients
    optimizer.step()      # Update the model's parameters
```

```
[118] # Predictions
y_pred = regmodel(X_test_tensor)
y_pred = [y.detach().numpy()[]>=0 for y in y_pred]

# Convert true labels tensor to numpy array
y_test_np = y_test_tensor.numpy()

# Evaluate the model
accuracy = accuracy_score(y_test_np, y_pred)
print("Accuracy:", accuracy)

Accuracy: 0.6536180308422301
```

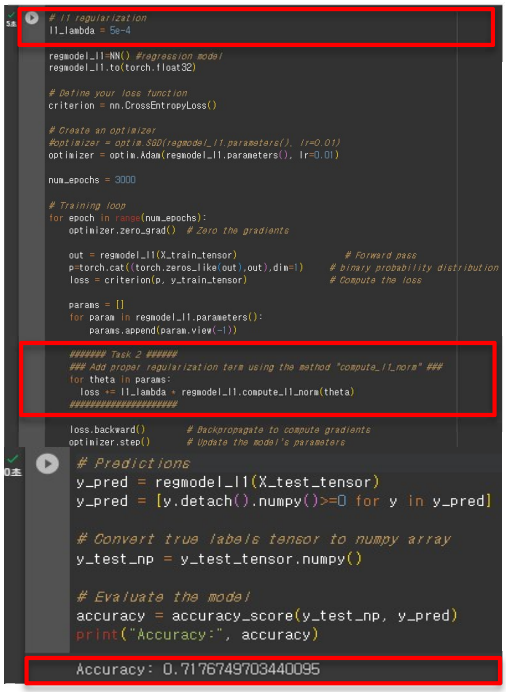
[Fig 4] Training / Test model without regularization

아래는 L1, L2, dropout 정규화 각각을 적용 시, test accuracy 의 변화를 알아보는 과정이다. 우선, L1 정규화를 먼저 진행해보자.

2) Define the loss function with l_1-regularization. Use *compute_l1_norm* method.

<Explanation>

기존 loss function 이 $L(\theta)$일 때, L1 정규화가 적용된 loss function 은 $L_1(\theta) = L(\theta) + \lambda\Sigma_i|\theta_i|$ 이며, 단순히 기존 loss function 에다가 위에서 정의한 **compute_l1_norm** 함수의 반환값에 λ를 곱해 더해주면 된다. 이에 대한 code 는 **[Fig 5-1]**와 같으며($\lambda = 5 \times 10^{-4}$), 해당 modified loss function $L_1(\theta)$으로 training 한 model 의 test accuracy 는 **[Fig 5-2]**와 같다. **정확도는 0.7177 정도가 나왔으며, 정규화가 없는 model 의 정확도인 0.6536 보다 더 높은 수치가 나왔다**. 주의할 점은, compute_l1_norm 함수는 NN class 내 정의된 함수이므로, norm 계산 시, regmodel_l1.compute_l1_norm (regmodel_l1 이 NN class 내 object) 으로 호출해야 한다.



```
# l1 regularization
l1_lambda = 5e-4

regmodel_l1=NN() #regression model
regmodel_l1.to(torch.float32)

# Define your loss function
criterion = nn.CrossEntropyLoss()

# Create an optimizer
#optimizer = optim.SGD(regmodel_l1.parameters(), lr=0.01)
optimizer = optim.Adam(regmodel_l1.parameters(), lr=0.01)

num_epochs = 3000

# Training loop
for epoch in range(num_epochs):
    optimizer.zero_grad()  # Zero the gradients

    out = regmodel_l1(X_train_tensor)                    # Forward pass
    p=torch.cat((torch.zeros_like(out),out),dim=1)       # binary probability distribution
    loss = criterion(p, y_train_tensor)                  # Compute the loss

    params = []
    for param in regmodel_l1.parameters():
        params.append(param.view(-1))

    ####### Task 2 ######
    ### Add proper regularization term using the method "compute_l1_norm" ###
    for theta in params:
        loss += l1_lambda * regmodel_l1.compute_l1_norm(theta)
    ######################

    loss.backward()      # Backpropagate to compute gradients
    optimizer.step()     # Update the model's parameters

# Predictions
y_pred = regmodel_l1(X_test_tensor)
y_pred = [y.detach().numpy()>=0 for y in y_pred]

# Convert true labels tensor to numpy array
y_test_np = y_test_tensor.numpy()

# Evaluate the model
accuracy = accuracy_score(y_test_np, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.7176749703440095
```

[Fig 5-1] (Above) Code for Exercise *2*

[Fig 5-2] (Below) Test accuracy for model with l_1-regularization

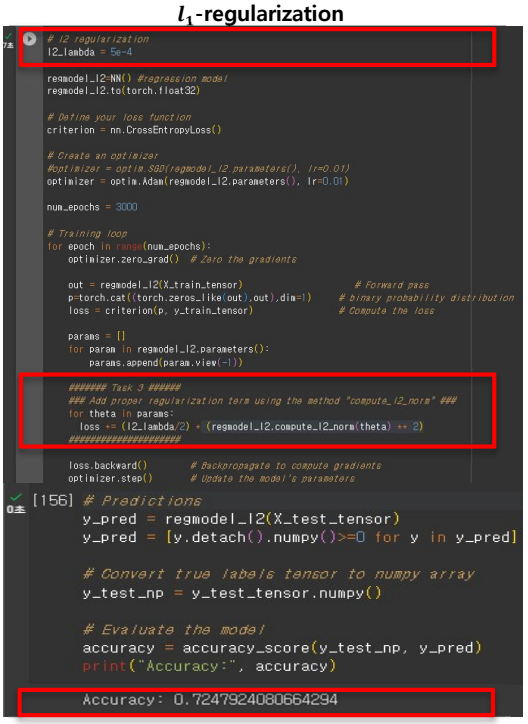
3) Define the loss function with l_2-regularization. Use *compute_l2_norm* method.

<Explanation>

L2 정규화가 적용된 loss function 은 $L_2(\theta) = L(\theta) + \frac{\lambda}{2}\Sigma_i\theta_i^2$ 이며, $\Sigma_i\theta_i^2$ 항은 위에서 정의한 **compute_l2_norm** 함수의 반환값에 제곱을 해야한다. 그 결과값에 $\frac{\lambda}{2}$를 곱해 기존 loss function $L(\theta)$에 더해주면 된다. 이에 대한 code 는 **[Fig 6-1]**와 같으며($\lambda = 5 \times 10^{-4}$), 해당 modified loss function $L_2(\theta)$으로 training 한 model 의 test accuracy 는 **[Fig 6-2]**와 같다. **정확도는 0.7248 정도가 나왔으며, 정규화가 없는 model 의 정확도인 0.6536 보다 더 높은 수치가 나왔다**.

Note. [Fig 6-1]의 list 형 변수 params 내에는 여러 개의 tensor elements 가 존재하며, 각 tensor 마다의 $\Sigma_i\theta_i^2$를 계산하여 합한 것은, 모든 tensor 를 하나의 single tensor 로 concatenate(연결)하고 그 단일 tensor 자체의 $\Sigma_i\theta_i^2$를 계산한 것과 동일하다(Linearity 성립).

즉, params list 를 변화시키지 않고, list 내 각각의 tensor 원소를 theta 변수에 저장 후, theta 마다 regmodel_l2.compute_l2_norm(theta) ** 2 (수식으로 표현하는 경우, $\left(\sqrt{\Sigma_i\theta_i^2}\right)^2 = \Sigma_i\theta_i^2$) 을 연산하면 된다.



```
# l2 regularization
l2_lambda = 5e-4

regmodel_l2=NN() #regression model
regmodel_l2.to(torch.float32)

# Define your loss function
criterion = nn.CrossEntropyLoss()

# Create an optimizer
#optimizer = optim.SGD(regmodel_l2.parameters(), lr=0.01)
optimizer = optim.Adam(regmodel_l2.parameters(), lr=0.01)

num_epochs = 3000

# Training loop
for epoch in range(num_epochs):
    optimizer.zero_grad()  # Zero the gradients

    out = regmodel_l2(X_train_tensor)                    # Forward pass
    p=torch.cat((torch.zeros_like(out),out),dim=1)       # binary probability distribution
    loss = criterion(p, y_train_tensor)                  # Compute the loss

    params = []
    for param in regmodel_l2.parameters():
        params.append(param.view(-1))

    ####### Task 3 ######
    ### Add proper regularization term using the method "compute_l2_norm" ###
    for theta in params:
        loss += (l2_lambda/2) * (regmodel_l2.compute_l2_norm(theta) ** 2)
    ######################

    loss.backward()      # Backpropagate to compute gradients
    optimizer.step()     # Update the model's parameters

[156] # Predictions
y_pred = regmodel_l2(X_test_tensor)
y_pred = [y.detach().numpy()>=0 for y in y_pred]

# Convert true labels tensor to numpy array
y_test_np = y_test_tensor.numpy()

# Evaluate the model
accuracy = accuracy_score(y_test_np, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.7247924080664294
```

[Fig 6-1] (Above) Code for Exercise *3*

[Fig 6-2] (Below) Test accuracy for model with l_2-regularization

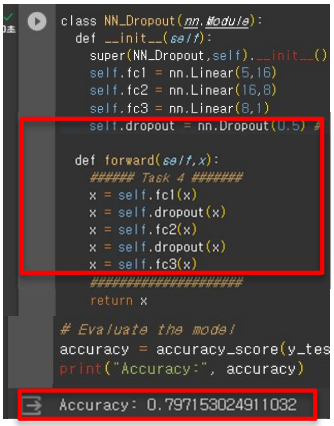
Dropout 이란, neutral network 에서 dropout layer 내 각 neuron 마다 dropped out 이 될 확률을 부여하여, neuron 이 확률적으로 꺼지도록(off) 만드는 정규화 방식이다. 이는 특정 neuron 이 input feature 에 대해 강하게 의존하는 것을 방지하여 model 이 더 일반화될 수 있도록 하는 것을 목적으로 둔다.

4) Design your own network with dropout (use at least once)

<Explanation>

[Fig 7]은 임의로 구성한 NN_Dropout neural network 로, fc1(nn.Linear(5,16)), fc2(nn.Linear(16,8)) 사이 한 번, fc2, fc3(nn.Linear(8,1)) 사이 한 번 dropout rate 0.5 의 dropout layer 가 추가되었다.

해당 model 을 동일 loss function(cross entropy), optimizer(Adam)로 training 시킨 뒤, test accuracy 를 구한 결과, **0.7971** 의 타 정규화의 경우보다 높은 정확도를 보였다.



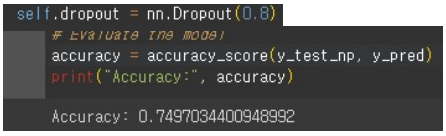
[Fig 7-1] (Above) Code for Exercise 4
[Fig 7-2] (Below) Test accuracy for model with dropout layers

● Discussion

1) Change of dropout rate

만일 **[Fig 7-1]**에서 Dropout rate 가 변한다면 어떤 결과가 나타날까? Dropout rate 를 0.5 에서 0.8 로 올려 model 을 훈련 시킨 결과, test accuracy 가 감소한 것을 확인할 수 있다. Dropout rate 가 높아졌다는 것은 사용하지 않는 neuron 의 비율이 높아진다는 것이며, training 시 더 적은 unit 이 학습된다는 것과 같다. 이는 overfitting 으로부터 벗어나는 데는 도움이 될 수 있지만, model 이 너무 학습을 하지 못해 underfitting 이 일어날 수도 있으며 **[Fig 8]**과 같이 test accuracy 가 감소하는 결과를 낳을 수도 있다.

반대로, 0.5 에서 0.2 로 낮춘 경우, test accuracy 가 0.79~0.8 사이의 값을 보이며, 0.5 일 때의 경우 여러 번의 test 를 실행해보면 0.79 이하의 정확도를 보이는 경우를 빈번히 관찰할 수 있다. 즉, 0.2 부근의 dropout rate 가 test accuracy 를 높이는 데 도움이 되며, 만일 rate 가 너무 높다면 오히려 model 의 정확도가 낮아지는 결과를 초래한다.



[Fig 8] Test accuracy for model with dropout layers (dropout rate = 0.8)

2) Number of Layers

Exercise 4 에서는 Dropout Layer 을 fc1, fc2 사이, 그리고 fc2, fc3 사이에 총 2 개 사용하였다. 만일, layer 를 fc2, fc3 사이 하나만 추가하고, fc1 와 fc2 사이엔 추가하지 않으면 어떻게 될까? Dropout rate 를 0.5 로 설정하고 확인한 결과, 반복적으로 test 를 진행해보면 큰 test accuracy 차이를 보이지 않았다. Layer 의 개수가 단지 하나 차이 밖에 안나기도 하며, training epoch 가 3000 인만큼 반복적으로 training 을 하면 정규화 사용 유무만큼의 정확도에 큰 차이를 보이지 않았다.

하지만, 가장 중요한 것은 output layer 후에 dropout layer 을 두면 안된다. **[Fig 9]**와 같이 마지막 linear layer 인 fc3 이후에 dropout layer 을 두게 되면, 상대적으로 test accuracy 가 매우 낮아지는 것을 확인할 수 있다. Dropout 이 마지막에 이루어진다는 것은, 이전까지의 연산 결과와 상관없이 마지막 output neuron 의 결과를 0 으로 만들어버리는 것과 같으므로, update 에 크게 잘못된 영향을 주게 된다. Dropout 은 여러 개의 neuron 으로 구성된 non-output layer 에 적용되어야 하며, 위에서 보았듯이 적절한 dropout rate 를 여러 trial 끝에 찾아내어 model 을 구현하는 게 중요하다.

[Fig 9] Test accuracy for model with dropout layer after the output layer