5. Regularization

AILab
Hanyang Univ.

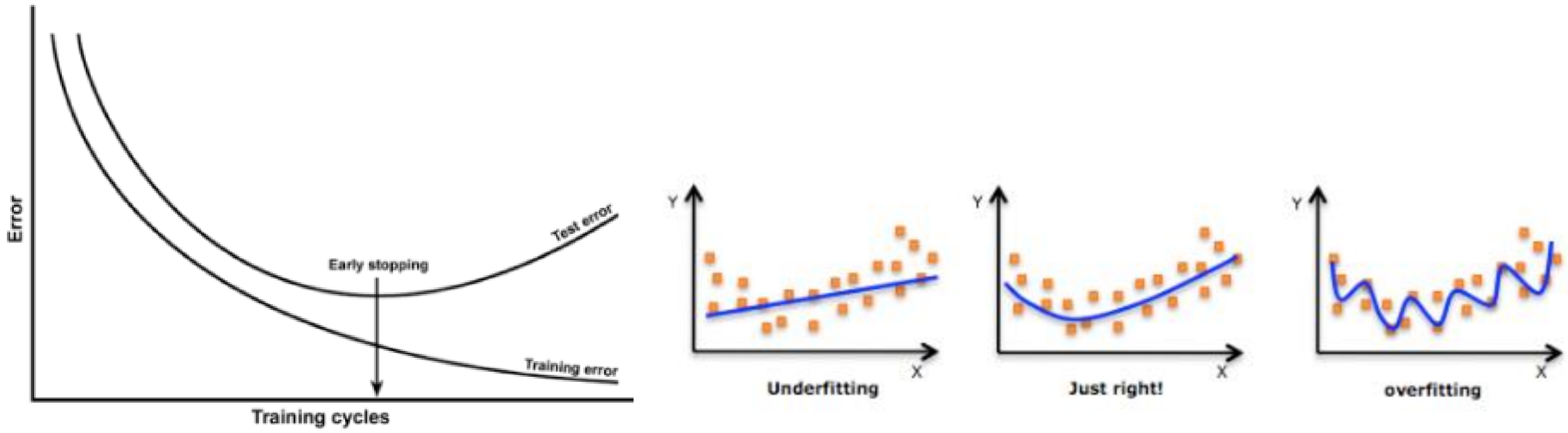
오늘 실습 내용

1. Overfitting
2. L1 L2 Regularization
3. Dropout
4. Normalization

1. Overfitting

Overfitting

- **Overfitting이란?**
  - 한 데이터셋에만 지나치게 최적화된 상태
  - 아래 그래프처럼 학습 데이터에 대해서는 오차가 감소하지만 실제 데이터에 대해서는 오차가 증가하는 지점이 존재할 수 있음
  - 즉, overfitting은 학습데이터에 대해 과하게 학습하여 실제 데이터에 대한 오차가 증가할 경우 발생



Overfitting

- **Overfitting 을 완화시키는 방법은?**
- Training Data 를 늘린다.
- Regularization
- Dropout
- Normalization

2. L1 L2 Regularization

Regularization

- 모델의 파라미터 확인하는 방법
  - Use named_parameters() or parameters()
  - https://pytorch.org/docs/stable/generated/torch.nn.Module.html?highlight=named_parameters#torch.nn.Module.named_parameters

```python
class LogisticRegression(nn.Module):
  def __init__(self, x_in, x_out):
    super(LogisticRegression, self).__init__()
    self.linear = nn.Linear(x_in, x_out)
    self.activation = nn.Sigmoid()
  def forward(self, x):
    z = self.linear(x)
    a = self.activation(z)
    return a
```

```python
for name, param in model.named_parameters():
  print('=========================')
  print(name)
  print(param.shape)
  print(param)
```

```
=========================
linear.weight
torch.Size([1, 2])
Parameter containing:
tensor([[ 0.0162, -0.1808]], requires_grad=True)
=========================
linear.bias
torch.Size([1])
Parameter containing:
tensor([-0.0776], requires_grad=True)
```

Regularization

- L1 loss in LogisticRegression

reg = model.linear.weight.abs().sum()

$$\|w\|_1 = \sum_{j=1}^{n} |w_j|$$

```
print(model.linear.weight)
print(model.linear.weight.abs().sum())
```

```
Parameter containing:
tensor([[ 0.0162, -0.1808]], requires_grad=True)
tensor(0.1970, grad_fn=<SumBackward0>)
```

Regularization

- L2 loss in LogisticRegression

reg = model.linear.weight.pow(2.0).sum()

$$\|w\|_2^2 = \sum_{j=1}^{n} w_j^2 = w^T w$$

```
print(model.linear.weight)
print(model.linear.weight.pow(2.0).sum())
```

```
Parameter containing:
tensor([[ 0.0162, -0.1808]], requires_grad=True)
tensor(0.0330, grad_fn=<SumBackward0>)
```

Regularization

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

- L1 loss

```
reg = model.linear.weight.abs().sum()

loss = loss + lambda * reg/total_num/2.
```

- L2 loss

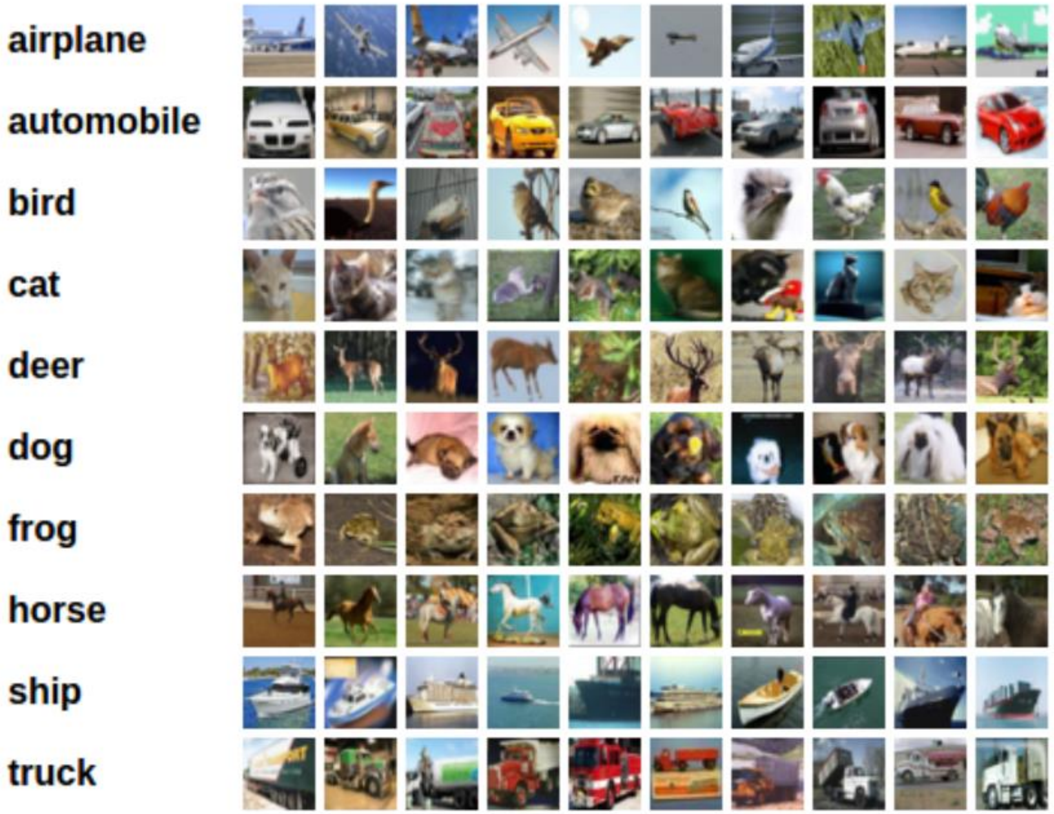
```
L2_norm = model.linear.weight.pow(2.0).sum()

loss = loss + lambda * reg/total_num/2.
```

~~Logistic Regression의 overfitting 확인하거나 성능 확인하는 test set~~

Cifar-10 Data

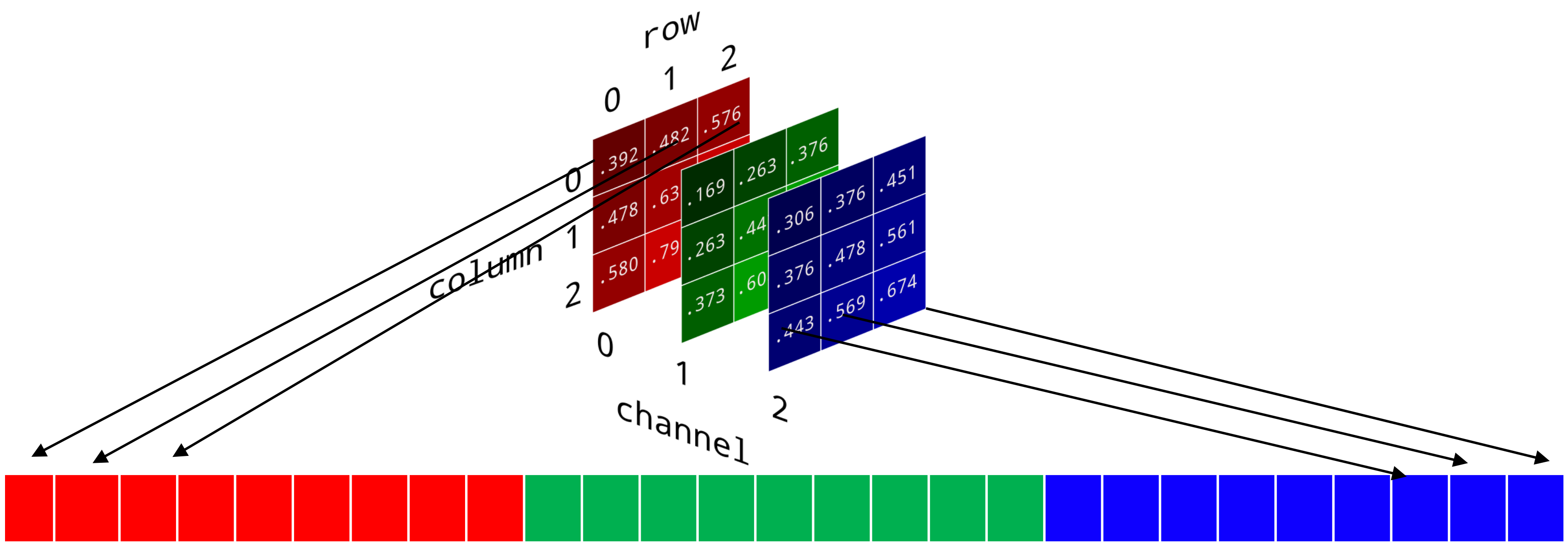
Cifar-10

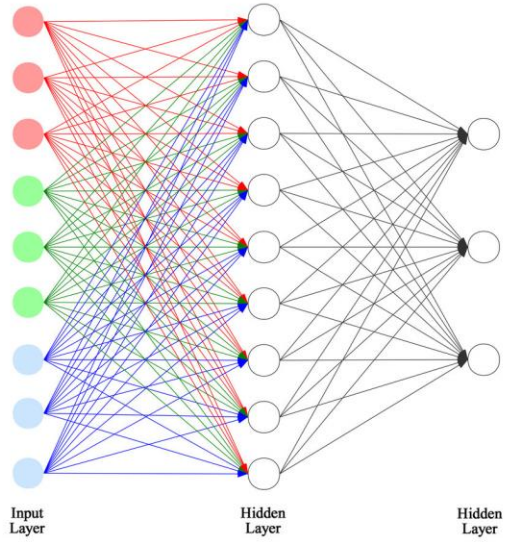
- Cifar-10 Dataset
  - 10개의 클래스로 분류된 RGB image
  - 6000 32x32 per class. 50000 training images, 10000 test images.



Cifar-10

- RGB image
  - View 이용해서 shape 조정





3D data → 1D data

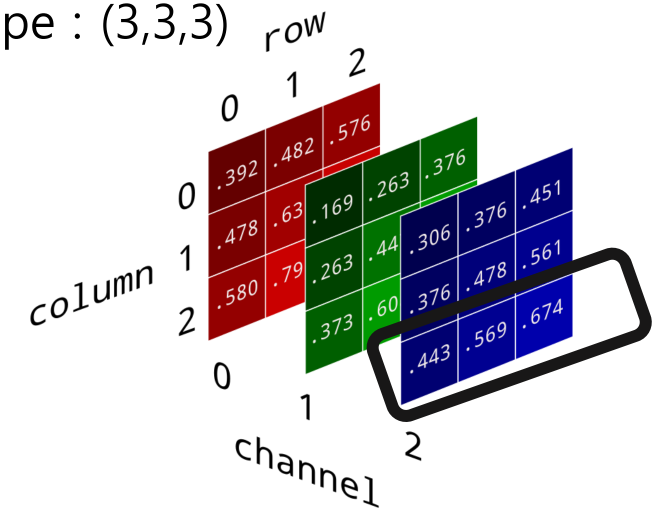
Cifar-10

- Image Data in Simple Neural Network
  - 실제로 view가 잘 실행되는지 확인

```
img = torch.FloatTensor([ [[.392,.482,.576],[.478,.639,.241], [.580,.790,.543]],
                          [[.169,.263,.376],[.263,.442,.823],[.373,.602,.165]],
                          [[.306,.376,.451],[.376,.478,.561],[.443,.569,.674]] ])
```

img

shape : (3,3,3)




```
print(img.shape)
new_img = img.view(3*3*3)
print(new_img.shape)
print(new_img)
print(new_img[3*3*3-3:3*3*3])
print(img[2,2,:])
```

```
torch.Size([3, 3, 3])
torch.Size([27])
tensor([0.3920, 0.4820, 0.5760, 0.4780, 0.6390, 0.2410, 0.5800, 0.7900, 0.5430,
        0.1690, 0.2630, 0.3760, 0.2630, 0.4420, 0.8230, 0.3730, 0.6020, 0.1650,
        0.3060, 0.3760, 0.4510, 0.3760, 0.4780, 0.5610, 0.4430, 0.5690, 0.6740])
tensor([0.4430, 0.5690, 0.6740])
tensor([0.4430, 0.5690, 0.6740])
```

new_img

shape : (27)

Cifar-10

- Torchvision 이용해서 data 불러오기

```
import torchvision
import torchvision.transforms as transforms

train_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                    train=True,
                    transform=transforms.ToTensor(),
                    download=True)
test_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                    train=False,
                    transform=transforms.ToTensor(),
                    download=True)
```

Regularization

- Random Seed 고정
  - 모델 weight가 생성될 때마다 random하게 생성됨
  - 성능 비교를 하기위해 값을 고정하는 것이 좋다.
    - 어떤 random한 값에서는 좋게 나오고 다른 값에서는 나쁘게 나올 수 있음

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.cuda.manual_seed_all(0)
```

```
print(model.linear1.weight[0,1])
```

```
tensor(-0.0298, device='cuda:0', grad_fn=<SelectBackward0>)
```

Regularization

- Cifar-10 Dataset
  - Model 구조
    - Input size : 32*32*3
    - Output size : 10

```
class Model(nn.Module):
  def __init__(self):
    super(Model, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 256)
    self.linear2 = nn.Linear(256, 128)
    self.linear3 = nn.Linear(128, 10)

    self.activation = nn.Sigmoid()

  def forward(self, x):
    z1 = self.linear1(x)
    a1 = self.activation(z1)

    z2 = self.linear2(a1)
    a2 = self.activation(z2)

    z3 = self.linear3(a2)

    return z3
```

Regularization

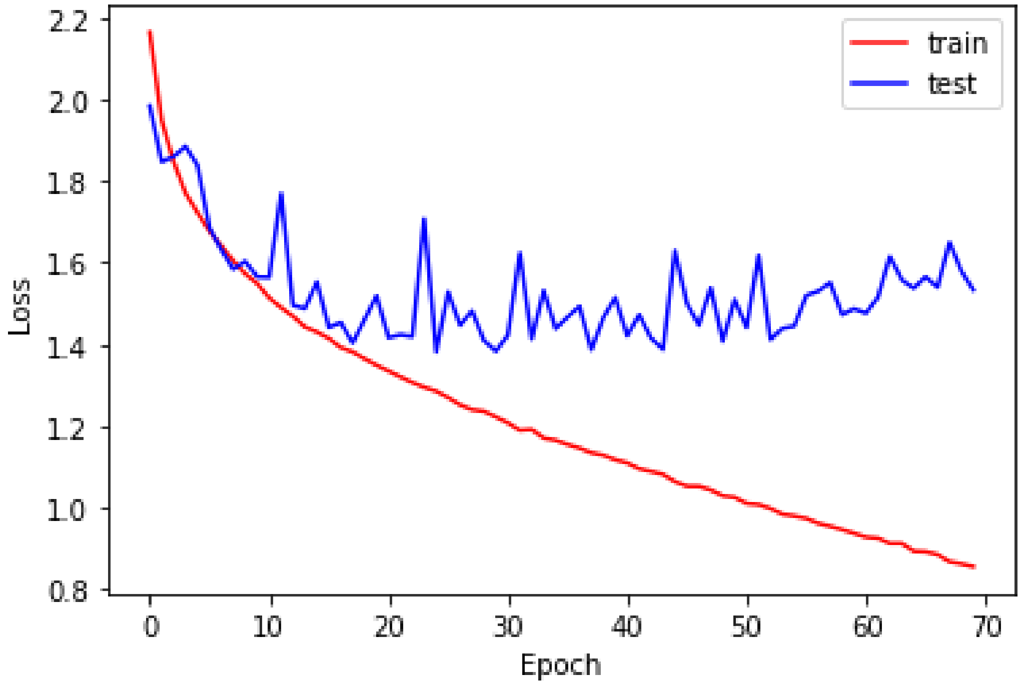
- 실제로 학습이 overfitting인지 확인
  - Matplotlib 활용
  - detach()
    - 기존 텐서 복사
  - cpu()
    - GPU에 있는 tensor CPU로 이동

```
epochs = 70
train_avg_costs = []
test_avg_costs = []

test_total_batch = len(test_dataloader)
total_batch_num = len(train_dataloader)

for epoch in range(epochs):
    avg_cost = 0
    model.train()
    for b_x, b_y in train_dataloader:
      b_x = b_x.view(-1, 32*32*3).to(device)
      logits = model(b_x)  # forward propagation
      loss = criterion(logits, b_y.to(device)) # get cost

      optimizer.zero_grad()
      loss.backward() # backward propagation
      optimizer.step() # update parameters

      avg_cost += loss / total_batch_num
    train_avg_costs.append(avg_cost.detach().cpu())
    print('Epoch : {} / {}, cost : {}'.format(epoch+1, epochs, avg_cost))

    test_avg_cost=0
    model.eval()
    for b_x, b_y in test_dataloader:
      b_x = b_x.view(-1, 32*32*3).to(device)
      with torch.no_grad():
        logits = model(b_x)
        test_loss = criterion(logits, b_y.to(device)) # get cost
      test_avg_cost += test_loss / test_total_batch

    test_avg_costs.append(test_avg_cost.detach().cpu())
```

Regularization

- 실제로 학습이 overfitting인지 확인
  - Matplotlib 활용
  - Test Loss 증가 확인

```
import matplotlib.pyplot as plt
import numpy as np
epoch = range(epochs)
plt.plot(epoch,train_avg_costs,'r-')
plt.plot(epoch,test_avg_costs,'b-')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(['train','test'])
plt.show()
```



Regularization

- 전체 코드 1/3

```
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.cuda.manual_seed_all(0)
```

```
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

```
import torchvision
import torchvision.transforms as transforms

train_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                    train=True,
                    transform=transforms.ToTensor(),
                    download=True)
test_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                    train=False,
                    transform=transforms.ToTensor(),
                    download=True)
```

```
batch_size = 128

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batc
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_
```

```
class Model(nn.Module):
  def __init__(self, drop_prob):
    super(Model, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 256)
    self.linear2 = nn.Linear(256, 128)
    self.linear3 = nn.Linear(128, 10)

    self.activation = nn.Sigmoid()

  def forward(self, x):
    z1 = self.linear1(x)
    a1 = self.activation(z1)

    z2 = self.linear2(a1)
    a2 = self.activation(z2)

    z3 = self.linear3(a2)

    return z3
```

Regularization

- 전체 코드 2/3

```
[ ] model = Model().to(device).train()

[ ] optimizer = optim.SGD(model.parameters(), lr=1) # set optimizer

[ ] criterion = nn.CrossEntropyLoss()

[ ] epochs = 70

    train_avg_costs = []
    test_avg_costs = []

    test_total_batch = len(test_dataloader)
    total_batch_num = len(train_dataloader)

    for epoch in range(epochs):
        avg_cost = 0
        model.train()
        for b_x, b_y in train_dataloader:
          b_x = b_x.view(-1, 32*32*3).to(device)
          logits = model(b_x)  # forward propagation
          loss = criterion(logits, b_y.to(device)) # get cost

          optimizer.zero_grad()
          loss.backward() # backward propagation
          optimizer.step() # update parameters

          avg_cost += loss / total_batch_num
        train_avg_costs.append(avg_cost.detach())
        print('Epoch : {} / {}, cost : {}'.format(epoch+1, epochs, avg_cost))

        test_avg_cost=0
        model.eval()
        for b_x, b_y in test_dataloader:
          b_x = b_x.view(-1, 32*32*3).to(device)
          with torch.no_grad():
            logits = model(b_x)
            test_loss = criterion(logits, b_y.to(device)) # get cost
          test_avg_cost += test_loss / test_total_batch

        test_avg_costs.append(test_avg_cost.detach())
```

Regularization

- 전체 코드 3/3

```
import matplotlib.pyplot as plt
import numpy as np
epoch = range(epochs)
plt.plot(epoch,train_avg_costs,'r-')
plt.plot(epoch,test_avg_costs,'b-')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(['train','test'])
plt.show()
```

Regularization

- L2 Regularization

Regularization for Neural Network

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|W^{[l]}\|_F^2$$

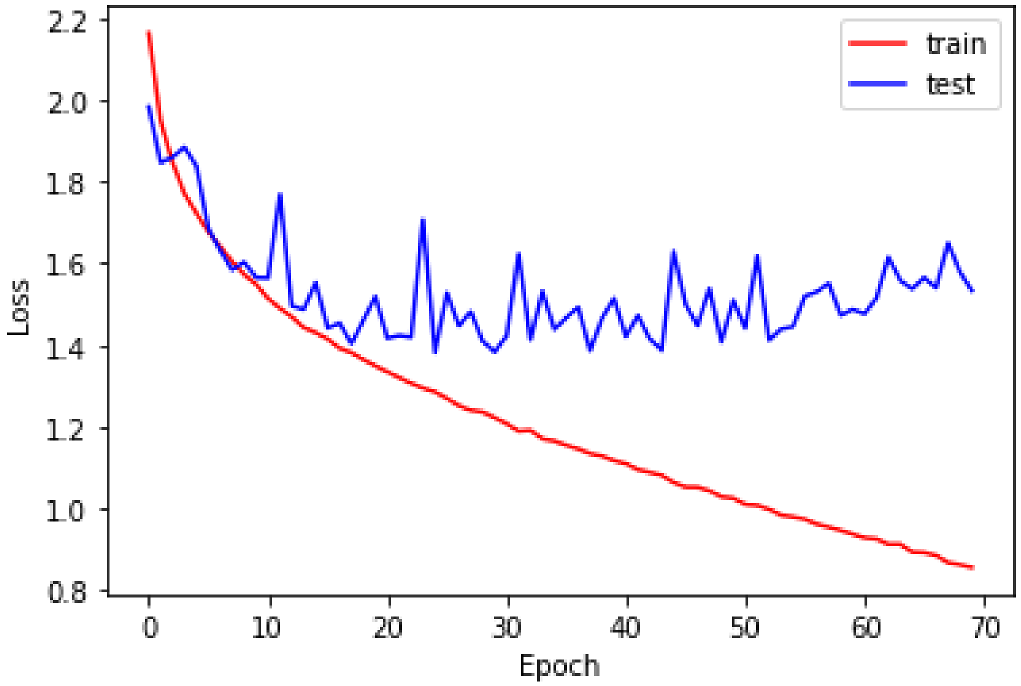
Frobenius norm: $\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(w_{ij}^{[l]}\right)^2 \qquad W^{[l]}: (n^{[l]}, n^{[l-1]})$

```python
epochs = 70
lmbd = 0.003

train_avg_costs = []
test_avg_costs = []

test_total_batch = len(test_dataloader)
total_batch_num = len(train_dataloader)

for epoch in range(epochs):
    avg_cost = 0
    model.train()
    for b_x, b_y in train_dataloader:
      b_x = b_x.view(-1, 32*32*3).to(device)
      logits = model(b_x)  # forward propagation
      loss = criterion(logits, b_y.to(device)) # get cost

      reg = model.linear1.weight.pow(2.0).sum()
      reg += model.linear2.weight.pow(2.0).sum()
      reg += model.linear3.weight.pow(2.0).sum()

      loss += lmbd*reg/len(b_x)/2.

      optimizer.zero_grad()
      loss.backward() # backward propagation
      optimizer.step() # update parameters

      avg_cost += loss / total_batch_num
    train_avg_costs.append(avg_cost.detach())
    print('Epoch : {} / {}, cost : {}'.format(epoch+1, epochs, avg_cost))

    test_avg_cost=0
    model.eval()
    for b_x, b_y in test_dataloader:
      b_x = b_x.view(-1, 32*32*3).to(device)
      with torch.no_grad():
        logits = model(b_x)
        test_loss = criterion(logits, b_y.to(device)) # get cost
      test_avg_cost += test_loss / test_total_batch

    test_avg_costs.append(test_avg_cost.detach())
```

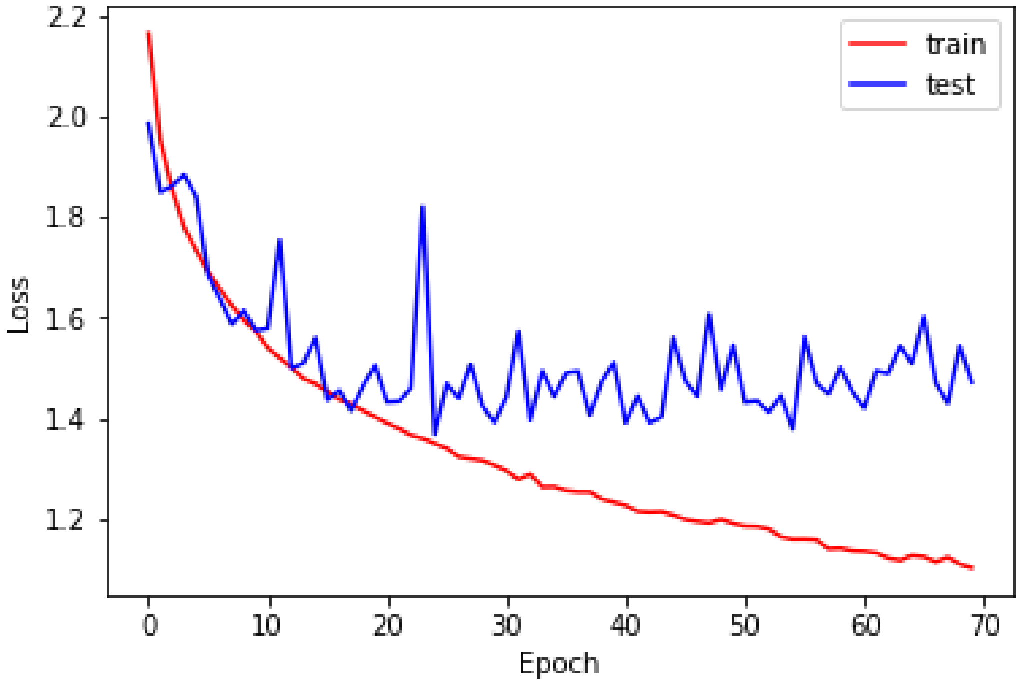
Regularization

• Loss Comparison

Base Model



L2 Regularization



3. Dropout

Dropout

- Dropout
  - https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

Parameters

- **p** – probability of an element to be zeroed. Default: 0.5
- **inplace** – If set to `True` , will do this operation in-place. Default: `False`

얼만큼 0으로 만들 것의냐의 확률

- model.eval()로 모드를 바꿔야 dropout이 작동하지 않음

Dropout

- Dropout

```
class Model(nn.Module):
  def __init__(self, drop_prob):
    super(Model, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 256)
    self.linear2 = nn.Linear(256, 128)
    self.linear3 = nn.Linear(128, 10)

    self.dropout = nn.Dropout(drop_prob)
    self.activation = nn.Sigmoid()

  def forward(self, x):
    z1 = self.linear1(x)
    a1 = self.activation(z1)
    al = self.dropout(a1)

    z2 = self.linear2(a1)
    a2 = self.activation(z2)
    a2 = self.dropout(a2)

    z3 = self.linear3(a2)

    return z3

model = Model(0.1).to(device).train()
```

4. Normalization

Normalization

- torchvision.transformers.Normalize
  - https://pytorch.org/vision/stable/generated/torchvision.transforms.Normalize.html#torchvision.transforms.Normalize

NORMALIZE

CLASS `torchvision.transforms.Normalize(mean, std, inplace=False)` [SOURCE]

Normalize a tensor image with mean and standard deviation. This transform does not support PIL Image. Given mean: `(mean[1],...,mean[n])` and std: `(std[1],..,std[n])` for `n` channels, this transform will normalize each channel of the input `torch.*Tensor` i.e., `output[channel] = (input[channel] - mean[channel]) / std[channel]`

1. subtract mean:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x^{(i)} := x^{(i)} - \mu$$

2. normalize variance:

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m} \left\{x_j^{(i)}\right\}^2$$

$$x_j^{(i)} := x_j^{(i)} / \sigma_j$$

Normalization

- torchvision.transformers.Normalize
  - Cifar-10 mean : (0.4914, 0.4822, 0.4465) (rgb)
  - Cifar-10 std : (0.247, 0.243, 0.261)

```
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))])

train_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                    train=True,
                    transform=transform,
                    download=True)
test_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                    train=False,
                    transform=transform,
                    download=True)
```

오늘 실습 내용

CIFAR-10에 L2 Regularization, Dropout, Normalization 모두 적용해서
test accuracy 확인

→ 만약 예상과 다른 결과가 나온다면 hyperparameter 조정해보기

epoch, learning rate, …