



02.16_Pytorch

▼ TORCH.AUTOGRAD를 사용한 자동 미분

torch.autograd를 사용한 자동 미분

파이토치(PyTorch) 기본 익히기|| 빠른 시작|| 텐서(Tensor)|| Dataset과 Dataloader|| 변형(Transform)|| 신경망 모델 구성하기|| Autograd|| 최적화(Optimization)|| 모델 저장하고 불러오기 신경망을 학습할 때 가장 자주
➊ https://tutorials.pytorch.kr/beginner/basics/autogradqs_tutorial.html

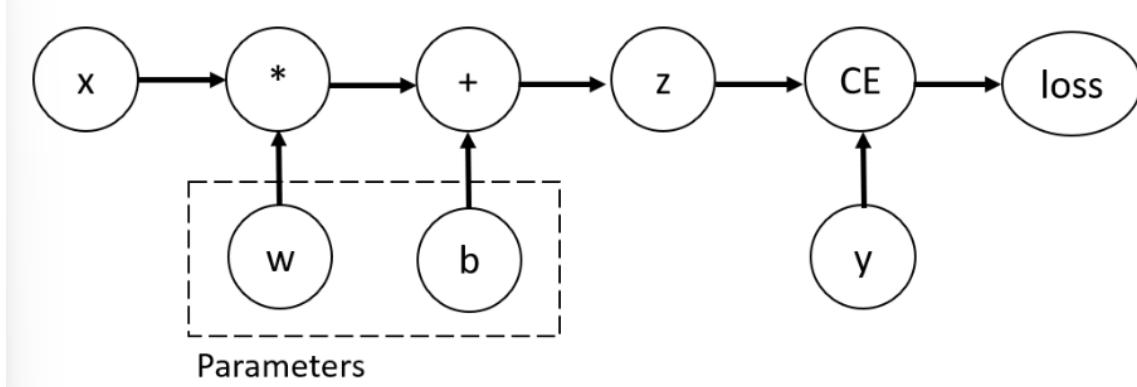
파이토치
한국 사용자

- 역전파에서 매개변수(모델 가중치)는 주어진 매개변수에 대한 손실 함수의 변화도(gradient)에 따라 조정
- 이러한 변화도를 계산하기 위해 pyTorch에는 torch.autograd라고 불리는 자동 미분 엔진 내장. 이는 모든 계산 그래프에 대한 변화도의 자동 계산 지원
- 입력 x, 매개변수 w와 b, 그리고 일부 손실 함수가 있는 가장 간단한 단일 계층 신경망을 가정

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

- Tensor, Function과 연산그래프(Computational graph)



- 이 신경망에서, w 와 b 는 최적화를 해야 하는 매개변수이다.
- 이러한 변수들에 대한 손실 함수의 변화도를 계산할 수 있어야 한다

이를 위해 해당 텐서에 `requires_grad` 속성을 설정

- 연산 그래프를 구성하기 위해 텐서에 적용하는 함수는 Function 클래스의 객체
- 이 객체는 순전파 방향으로 함수를 계산하는 방법과, 역전파 단계에서 도함수(**derivative**)를 계산하는 방법
- 역전파 함수에 대한 참조는 텐서의 `grad_fn` 속성에 저장

```

Gradient function for z = <AddBackward0 object at 0x7ff3df4e6280> # 메모리주
Gradient function for loss = <BinaryCrossEntropyWithLogitsBackward0 object at 0x7ff36616c940>
  
```

Out:

```

Gradient function for z = <AddBackward0 object at 0x7ff3df4e6280>
Gradient function for loss = <BinaryCrossEntropyWithLogitsBackward0 object at 0x7ff36616c940>
  
```

변화도(Gradient) 계산하기

- 신경망에서 매개변수의 가중치를 최적화하려면 매개변수에 대한 손실함수의 도함수(derivative)를 계산해야 한다. 즉, x 와 y 의 일부 고정값에서 $\frac{\partial \text{loss}}{\partial w}$ 와 $\frac{\partial \text{loss}}{\partial b}$ 필요
- 이러한 도함수를 계산하기 위해, `loss.backward()`를 호출한 다음, `w.grad`와 `b.grad`에서 값 출력

```

loss.backward()
print(w.grad)
  
```

```
print(b.grad)
```

Out:

```
tensor([[0.2162, 0.3280, 0.3170],  
       [0.2162, 0.3280, 0.3170],  
       [0.2162, 0.3280, 0.3170],  
       [0.2162, 0.3280, 0.3170],  
       [0.2162, 0.3280, 0.3170]])  
tensor([0.2162, 0.3280, 0.3170])
```

- 연산 그래프의 잎(leaf) 노드들 중 `requires_grad` 속성이 `True`로 설정된 노드들의 `grad` 속성만 구할 수 있습니다. 그래프의 다른 모든 노드에서는 변화도가 유효하지 않습니다.
- 성능 상의 이유로, 주어진 그래프에서의 `backward` 를 사용한 변화도 계산은 한 번만 수행할 수 있습니다. 만약 동일한 그래프에서 여러번의 `backward` 호출이 필요하면, `backward` 호출 시에 `retrain_graph=True` 를 전달해야 합니다.

변화도 추적 멈추기

- 기본적으로, `requires_grad=True`인 모든 텐서들은 연산 기록을 추적하고 변화도 계산을 지원
- 그러나 순전파 연산만 필요한 경우에는, 이러한 추적이나 지원이 필요 없을 수 있다. 이런 경우에는 연산 코드를 `torch.no_grad()` 블록으로 둘러싸서 연산 추적 중단. 이를 통해 순전파 단계만 수행할 때 연산 속도가 향상

```
z = torch.matmul(x, w)+b  
print(z.requires_grad)  
  
with torch.no_grad():  
    z = torch.matmul(x, w)+b  
print(z.requires_grad)
```

Out:

```
True  
False
```

- 변화도 추적을 하는 것은 느리다
- 테스트 코드에서는 필요 없음
- `with torch.no_grad()`: 이 코드 밑에는 gradient 계산 안함
- 다른 방법은 텐서에 `detach()` 메소드 사용

```
z = torch.matmul(x, w)+b
z_det = z.detach()
print(z_det.requires_grad)
```

Out: False

- 변화도 추적을 멈춰야 하는 이유
 - 신경망의 일부 매개변수를 고정된 매개변수(frozen parameter)로 표시합니다. 이는 사전 학습된 신경망을 미세조정 할 때 매우 일반적인 시나리오입니다.
 - 변화도를 추적하지 않는 텐서의 연산이 더 효율적이기 때문에, 순전파 단계만 수행할 때 연산 속도가 향상됩니다.

연산 그래프에 대한 추가 정보

- 개념적으로, autograd는 데이터(텐서)의 및 실행된 모든 연산들(및 연산 결과가 새로운 텐서인 경우도 포함하여)의 기록을 Function 객체로 구성된 방향성 비순환 그래프(DAG; Directed Acyclic Graph)에 저장(keep)합니다.
- 이 방향성 비순환 그래프(DAG)의 잎(leaf)은 입력 텐서이고, 뿌리(root)는 결과 텐서이다.
- 이 그래프를 뿌리에서부터 잎까지 추적하면 연쇄 법칙(chain rule)에 따라 변화도를 자동 계산 가능
- 순전파 단계에서, autograd는 다음 두 가지 작업을 동시에 수행합니다:
 - 요청된 연산을 수행하여 결과 텐서를 계산하고,
 - DAG에 연산의 변화도 기능(gradient function)를 유지(maintain)
- 역전파 단계는 DAG 뿌리(root)에서 `.backward()` 가 호출될 때 시작됩니다. `autograd` 는 이 때:
 - 각 `.grad_fn` 으로부터 변화도를 계산하고,
 - 각 텐서의 `.grad` 속성에 계산 결과를 쌓고(accumulate),
 - 연쇄 법칙을 사용하여, 모든 잎(leaf) 텐서들까지 전파(propagate)

PyTorch에서 DAG들은 동적(dynamic)입니다. 주목해야 할 중요한 점은 그래프가 처음부터(from scratch) 다시 생성된다는 것입니다; 매번 `.backward()` 가 호출되고 나면, autograd는 새로운 그래프를 채우기 (populate) 시작합니다. 이러한 점 덕분에 모델에서 흐름 제어(control

flow) 구문들을 사용할 수 있게 되는 것입니다; 매번 반복(iteration)할 때마다 필요하면 모양(shape)이나 크기(size), 연산(operation)을 바꿀 수 있습니다.

▼ Optimizing Model Params

모델 매개변수 최적화하기

파이토치(PyTorch) 기본 익히기|| 빠른 시작|| 텐서(Tensor)|| Dataset과 Dataloader|| 변형(Transform)|| 신경망 모델 구성하기|| Autograd|| 최적화(Optimization)|| 모델 저장하고 불러오기 이제 모델과 데이터가 준비되
👉 https://tutorials.pytorch.kr/beginner/basics/optimization_tutorial.html

파이토치
한국 사용자

하이퍼파라미터(Hyperparameter)

- 하이퍼파라미터 : 모델 최적화 과정을 제어할 수 있는 조절 가능한 매개변수
- 서로 다른 하이퍼파라미터 값은 모델 학습과 수렴률(convergence rate)에 영향을 미칠 수 있다.
- 학습 시에는 다음과 같은 하이퍼파라미터를 정의:
 - 에폭(epoch)수 : 데이터 셋을 반복하는 횟수
 - 배치 크기(batch size) : 매개변수가 갱신되기 전 신경망을 통해 전파된 데이터의 샘플 수
 - 학습률(learning rate) : 각 배치/에폭에서 모델의 매개변수를 조절하는 비율. 값이 작을 수록 학습 속도가 느려지고, 값이 크면 학습 중 예측할 수 없는 동작 가능

```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

최적화 단계(Optimization Loop)

- 하이퍼파라미터를 설정한 뒤에는 최적화 단계를 통해 모델을 학습하고 최적화
- 하나의 에폭은 다음 두 부분으로 구성
 - 학습 단계(train loop) : 학습용 데이터 셋을 반복하고 최적의 매개변수로 수렴
 - 검증/테스트 단계(validation/test loop) : 모델 성능이 개선되고 있는지를 확인하기 위해 테스트 데이터 셋을 반복

손실 함수(Loss Function)

- 손실 함수는 획득한 결과와 실제 값 사이의 틀린 정도(degree of dissimilarity)를 측정하며, 학습 중에 이 값을 최소화
- 주어진 데이터 샘플을 입력으로 계산한 예측과 정답(label)을 비교하여 손실(loss)을 계산
- 일반적인 손실함수에는 회귀 문제(regression task)에 사용하는 nn.MSELoss(평균 제곱 오차:Mean Square Error)나 분류(classification)에 사용하는 nn.NLLLoss(음의 로그 우도:Negative Log Likelihood), 그리고 nn.LogSoftmax와 nn.NLLLoss를 합친 nn.CrossEntropyLoss 등이 있다.
- 여기서는 모델의 출력 로짓(logit)을 nn.CrossEntropyLoss에 전달하여 logit을 정규화하고 예측 오류를 계산

```
# 손실 함수를 초기화
loss_fn = nn.CrossEntropyLoss()
```

$$loss(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right)$$

옵티마이저(Optimizer)

- 최적화 : 각 학습 단계에서 모델의 오류를 줄이기 위해 모델 매개변수를 조정하는 과정
- 여기서는 SGD(확률적 경사하강법;Stochastic Gradient Descent) 옵티마이저 사용
- 학습하려는 모델의 매개변수와 학습률 하이퍼파라미터를 등록하여 옵티마이저를 초기화

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- 학습 단계에서 최적화 단계:

- `optimizer.zero_grad()`를 호출하여 모델 매개변수의 변화도를 재설정. 기본적으로 변화도는 더해지기 때문에 중복 계산을 막기 위해 반복할 때마다 명시적으로 0으로 설정
 - `loss.backward()`를 호출하여 예측 손실(prediction loss)을 역전파.
 - 변화도를 계산한 뒤에는 `optimizer.step()`을 호출하여 역전파 단계에서 수집된 변화도로 매개변수를 조정
- 전체 구현 - Train Loop

```

def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # 예측(prediction)과 손실(loss) 계산
        pred = model(X)
        loss = loss_fn(pred, y)

        # 역전파
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:.7f} [{current:.5d}/{size:.5d}]")

def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")

```

```

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-----")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")

```

Out

```

Out: Epoch 1
-----
loss: 2.296443 [ 0/60000]
loss: 2.289541 [ 6400/60000]
loss: 2.273510 [12800/60000]
loss: 2.269320 [19200/60000]
loss: 2.252222 [25600/60000]
loss: 2.214578 [32000/60000]
loss: 2.230923 [38400/60000]
loss: 2.191731 [44800/60000]
loss: 2.193038 [51200/60000]
loss: 2.163834 [57600/60000]
Test Error:
Accuracy: 29.8%, Avg loss: 2.157976
...
Epoch 2
-----
loss: 2.166126 [ 0/60000]
loss: 2.159775 [ 6400/60000]
loss: 2.105813 [12800/60000]
loss: 2.122930 [19200/60000]
loss: 2.077600 [25600/60000]
loss: 2.005851 [32000/60000]
loss: 2.043375 [38400/60000]
loss: 1.962059 [44800/60000]
loss: 1.970146 [51200/60000]
loss: 1.904207 [57600/60000]
Test Error:
Accuracy: 57.6%, Avg loss: 1.901922
Epoch 9
-----
loss: 0.895599 [ 0/60000]
loss: 0.953716 [ 6400/60000]
loss: 0.735850 [12800/60000]
loss: 0.905481 [19200/60000]
loss: 0.781723 [25600/60000]
loss: 0.806867 [32000/60000]
loss: 0.886456 [38400/60000]
loss: 0.836664 [44800/60000]
loss: 0.855176 [51200/60000]
loss: 0.831358 [57600/60000]
Test Error:
Accuracy: 69.4%, Avg loss: 0.820203
Epoch 10
-----
loss: 0.841826 [ 0/60000]
loss: 0.913726 [ 6400/60000]
loss: 0.688676 [12800/60000]
loss: 0.867516 [19200/60000]
loss: 0.748552 [25600/60000]
loss: 0.766759 [32000/60000]
loss: 0.853791 [38400/60000]
loss: 0.811449 [44800/60000]
loss: 0.820697 [51200/60000]
loss: 0.801162 [57600/60000]
Test Error:
Accuracy: 70.8%, Avg loss: 0.788385
Done!

```

▼ Save and Load the Model

모델 저장하고 불러오기

파이토치(PyTorch) 기본 익히기|| 빠른 시작|| 텐서(Tensor)|| Dataset과 Dataloader|| 변형(Transform)|| 신경망 모델 구성하기|| Autograd|| 최적화(Optimization)|| 모델 저장하고 불러오기 이번 장에서는 저장하거나 불러오기하는 방법을 살펴보겠습니다.
🔗 https://tutorials.pytorch.kr/beginner/basics/saveloadrun_tutorial.html

파이토치
한국 사용자

```
import torch
import torchvision.models as models
```

모델 가중치 저장하고 불러오기

- PyTorch 모델은 학습한 매개변수를 `state_dict`라고 불리는 내부 상태 사전(internal state dictionary)에 저장. 이 상태 값들은 `torch.save` 메소드를 저장할 수 있다.

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

- 모델 가중치를 불러오기 위해서는, 먼저 동일한 모델의 인스턴스를 생성한 다음에 메소드를 사용하여 매개변수들을 불러온다.

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

모델의 형태를 포함하여 저장하고 불러오기

- 모델의 가중치를 불러올 때, 신경망의 구조를 정의하기 위해 모델 클래스를 먼저 생성해야 했다. 이 클래스의 구조를 모델과 함께 저장하고 싶으면 `model.state_dict()`가 아닌 `model`을 저장 함수에 전달

```
torch.save(model, 'model.pth')
```

- 다음과 같이 모델을 불러온다

```
model = torch.load('model.pth')
```

▼ Training a Classifier

Dataset and DataLoader

```
import torch
import torchvision
import torchvision.transforms as transforms

# Create datasets for training & validation, download if necessary
training_set = torchvision.datasets.FashionMNIST('./data', train=True, transform=transform, download=True)
validation_set = torchvision.datasets.FashionMNIST('./data', train=False, transform=transform, download=True)

# Create data loaders for our datasets; shuffle for training, not for validation
training_loader = torch.utils.data.DataLoader(training_set, batch_size=4, shuffle=True, num_workers=2)
```

```
transforms.Normalize((0.5, 0.5, 0.5))

# Create datasets for training & validation, download if necessary
training_set = torchvision.datasets.FashionMNIST('./data', train=True, transform=transform, download=True)
validation_set = torchvision.datasets.FashionMNIST('./data', train=False, transform=transform, download=True)

# Create data loaders for our datasets; shuffle for training, not for validation
training_loader = torch.utils.data.DataLoader(training_set, batch_size=4, shuffle=True, num_workers=2)
validation_loader = torch.utils.data.DataLoader(validation_set, batch_size=4, shuffle=False, num_workers=2)
```

```
validation_set = torchvision.datasets.FashionMNIST('./data', train=False, transform=transform, download=True)

# Create data loaders for our datasets; shuffle for training, not for validation
training_loader = torch.utils.data.DataLoader(training_set, batch_size=4, shuffle=True, num_workers=2)
validation_loader = torch.utils.data.DataLoader(validation_set, batch_size=4, shuffle=False, num_workers=2)
```

```
classes = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
           'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot')
```

`transforms.Compose` : 여러 단계로 변환해야 하는 경우, `Compose`를 통해 여러 단계를 묶음.

`transforms.ToTensor()` : 데이터를 tensor로 바꿈.

`transforms.Normalize(mean, std, inplace=False)` : 정규화

`datasets.FashionMNIST(root, train, transform, download)`

- `DataLoader`를 통해 데이터를 준비
- `num_workers` : 학습 도중 CPU의 작업을 몇 개의 코어를 사용해서 진행할지에 대한 설정 파라미터

```

# Class labels
classes = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
           'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot')

# Report split sizes
print('Training set has {} instances'.format(len(training_set)))
print('Validation set has {} instances'.format(len(validation_set)))

```

분류 label 정의 및 각 객체 개수 확인

Out:

Training set has 60000 instances
Validation set has 10000 instances

```

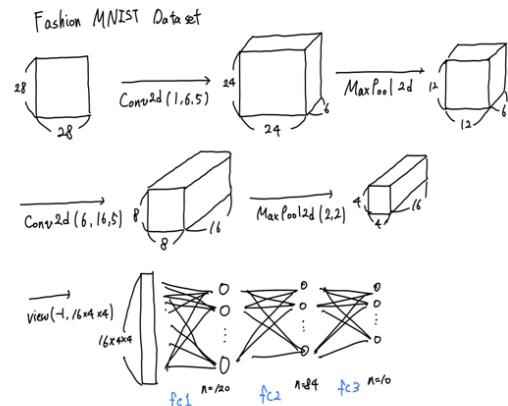
import torch.nn as nn
import torch.nn.functional as F

# PyTorch models inherit from torch.nn.Module
class GarmentClassifier(nn.Module):
    def __init__(self):
        super(GarmentClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = GarmentClassifier()

```



- 손실함수로 CrossEntropyLoss 사용

```
loss_fn = torch.nn.CrossEntropyLoss()
```

- Optimizer

```

# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

```

- The Training Loop

```

def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    for i, data in enumerate(training_loader):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = model(inputs)

        loss = loss_fn(outputs, labels)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000
            print(' batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss

```

- Per-Epoch Activity(Training)

```

from datetime import datetime
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
writer = SummaryWriter('runs/fashion_trainer_{}'.format(timestamp))
epoch_number = 0

EPOCHS = 5

best_vloss = 1_000_000.

for epoch in range(EPOCHS):
    print('EPOCH {}'.format(epoch_number + 1))

    model.train(True)
    avg_loss = train_one_epoch(epoch_number, writer)

    model.train(False)

    running_vloss = 0.0
    for i, vdata in enumerate(validation_loader):
        vinputs, vlabels = vdata
        voutputs = model(vinputs)
        vloss = loss_fn(voutputs, vlabels)
        running_vloss += vloss

    avg_vloss = running_vloss / (i + 1)
    print('LOSS train {} valid {}'.format(avg_loss, avg_vloss))

    writer.add_scalars('Training vs. Validation Loss',
                      {'Training': avg_loss, 'Validation': avg_vloss},
                      epoch_number + 1)
writer.flush()

```

```
if avg_vloss < best_vloss:  
    best_vloss = avg_vloss  
    model_path = 'model_{0}_{1}'.format(timestamp, epoch_number)  
    torch.save(model.state_dict(), model_path)  
  
epoch_number += 1
```

- Tensorboard

```
%load_ext tensorboard  
%tensorboard --logdir=runs
```

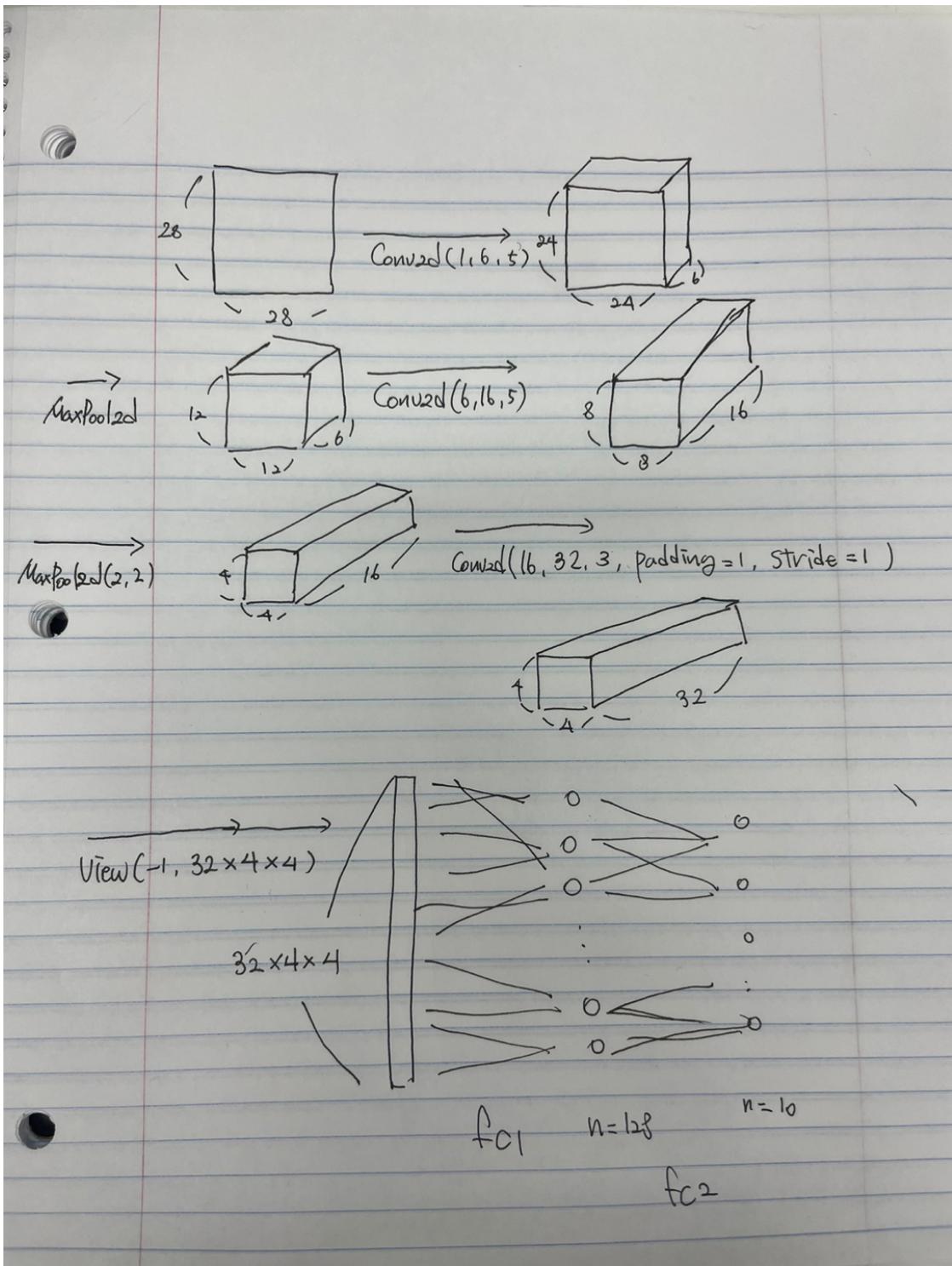
- $(N+2P-F)/S + 1$

▼ 실습(GarmentClassifier 바꿔보기)

Google Colaboratory

 <https://colab.research.google.com/drive/1OUGm4zM732VKWRd0Z34x6k-Pbl8Dun59?usp=sharing>



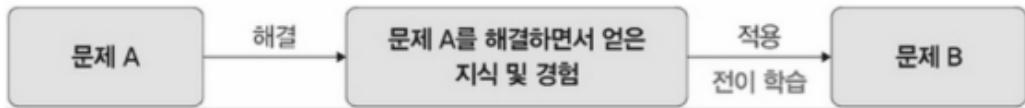


▼ Transfer_learning

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b85b87e1-860e-4730-8a8a-6fcc97fc26ba/Transfer_learning.pdf

데이터 구축은 고비용(시간 + 돈)!

- 이미 구축된 데이터를 활용하여 특징을 학습한 후 활용
- 예 : 이미지넷(120만 개의 이미지에 대하여 학습)을 활용하여 특징 학습 후에 적은 규모의 데이터셋(e.g. FashionMNIST)에 적용



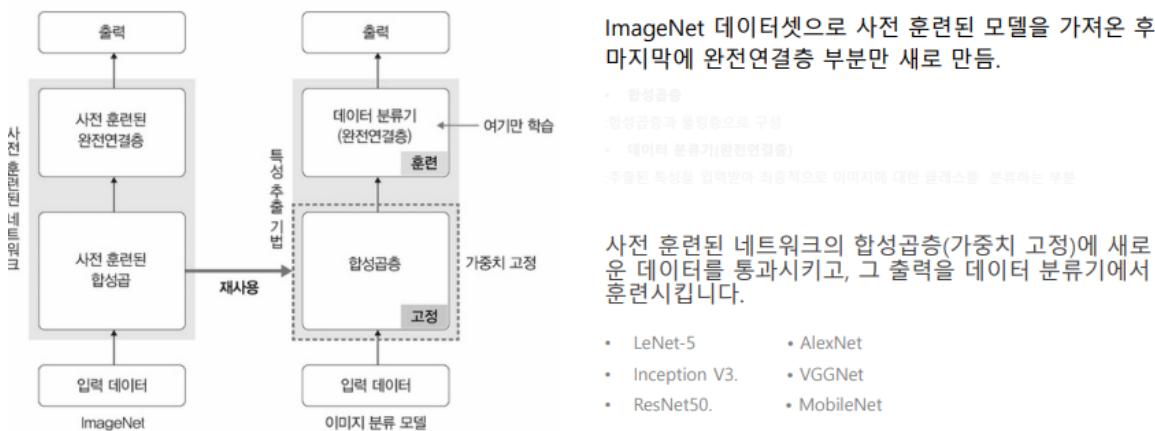
- 주로 세 가지 전략 사용

- ConvNet as fixed feature extractor : 마지막 FC 레이어만 학습
- Fine-tuning the ConvNet : 전체 네트워크를 재학습(소규모 데이터로)
- Pretrained models : 다른 연구자가 공개한 사전학습 모델 활용

• 고정된 특징 추출기로 써의 합성곱 신경망

ImageNet 데이터셋으로 사전 훈련된 모델을 가져온 후 마지막에 완전연결층 부분만 새로 만듭니다.

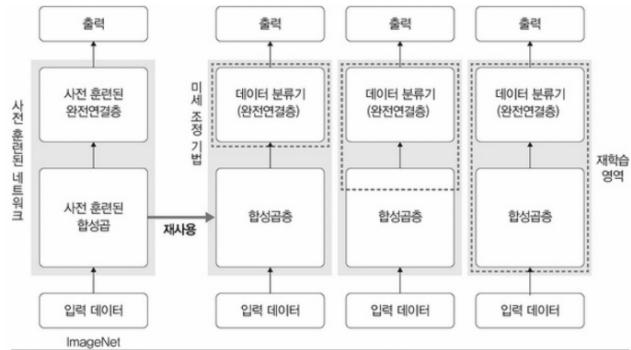
- 합성곱층
- 완전연결층과 풀링층으로 구성
- 데이터 분류기(완전연결층)
- 추출된 특성을 입력받아 최종적으로 이미지에 대한 클래스를 분류하는 부분



• 합성곱 신경망의 미세조정(finetuning)

사전 훈련된 네트워크를 미세 조정하여 분석하려는 데이터셋에 잘 맞도록 모델의 파라미터를 조정하는 기법

- 데이터셋이 크고 사전 훈련된 모델과 유사성이 작을 경우
- 모델 전체를 재학습
- 데이터셋 크기가 크기 때문에 재학습시키는 것이 좋은 전략
- 데이터셋이 크고 사전 훈련된 모델과 유사성이 높을 경우
- 합성곱층의 뒷부분(완전연결층과 가까운 부분)과 데이터 분류기를 학습
- 데이터셋이 유사하기 때문에 전체를 학습시키는 것보다는 강한 특징이 나타나는 합성곱층의 뒷부분과 데이터 분류기만 새로 학습하더라도 최적의 성능을 낼 수 있음
- 데이터셋이 작고 사전 훈련된 모델과 유사성이 작을 경우
- 합성곱층의 일부분과 데이터 분류기를 학습
- 데이터가 적기 때문에 일부 계층에 미세 조정 기법을 적용한다고 해도 효과가 없을 수 있음
- 데이터셋이 작고 사전 훈련된 모델과 유사성이 높을 경우
- 데이터 분류기만 학습
- 데이터가 적기 때문에 많은 계층에 미세 조정 기법을 적용하면 과적합이 발생할 수 있음



▼ 개미 / 벌 분류 실습

컴퓨터 비전(Vision)을 위한 전이학습(Transfer Learning)

Author: Sasank Chilamkurthy, 번역: 박정환,. 이 튜토리얼에서는 전이학습(Transfer Learning)을 이용하여 이미지 분류를 위한 합성곱 신경망을 어떻게 학습시키는지 배워보겠습니다. 전이학습에 대해서는

☞ https://tutorials.pytorch.kr/beginner/transfer_learning_tutorial

파이토
한국 사용자

Google Colaboratory

☞ <https://colab.research.google.com/drive/19HbKkEVTrnLDya4c3xC8TQybzgnRdNczg?usp=sharing>



1. Training 및 Test 데이터 셋을 불러오고 정규화

```

from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy

cudnn.benchmark = True
plt.ioff()  # 대화형 모드

```

```

# 학습을 위해 데이터 증가(augmentation) 및 일반화(normalization)
# 검증을 위한 일반화
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

```

- Normalize는 RGB 값의 평균과 분산값 사용
- 위에 숫자는 ImageNet의 평균과 분산값

```

data_dir = 'data/hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                         data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
                                              shuffle=True, num_workers=4)
                  for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

```

- 보통 validation set에서는 shuffle을 False로 True도 가능
- 일부 이미지 시각화

```

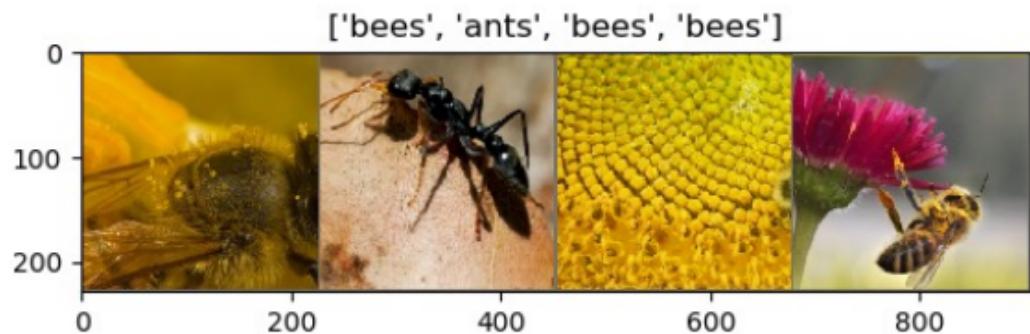
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # 갱신이 될 때까지 잠시 기다립니다.

# 학습 데이터의 배치를 얻습니다.
inputs, classes = next(iter(dataloaders['train']))

# 배치로부터 격자 형태의 이미지를 만듭니다.
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])

```



2. 모델 정의하기

```

model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# 새로 생성된 모듈의 매개변수는 기본값이 requires_grad=True 임
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

```

- `requires_grad` : 자동미분
 - `False`라면 FC 레이어에만 학습 적용
- `in_features`는 들어오는 차원수를 나타내줌
- `fc`는 모델마다 이름, 값이 다르기 때문에 각각의 모델에서 확인 필요

3. 손실함수와 Optimizer 정의

```

criterion = nn.CrossEntropyLoss()

# 모든 매개변수들이 최적화되었는지 관찰
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# 7 에폭마다 0.1씩 학습률 감소
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

```

4. Training set을 사용하여 신경망 학습

```

def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f'Epoch {epoch}/{num_epochs - 1}')
        print('-' * 10)

        # 각 에폭(epoch)은 학습 단계와 검증 단계를 갖습니다.
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train() # 모델을 학습 모드로 설정
            else:
                model.eval() # 모델을 평가 모드로 설정

            running_loss = 0.0
            running_corrects = 0

            # 데이터를 반복
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # 매개변수 경사도를 0으로 설정
                optimizer.zero_grad()

                # 순전파
                # 학습 시에만 연산 기록을 추적
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                # 학습 단계인 경우 역전파 + 최적화
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

```

- 모델을 저장하는 방법
 - 100번마다 저장하는 방법

- 되돌아가기 쉽지만 저장용량 많이 차지
 - 성능이 이전 모델보다 좋아졌을 때 저장하는 방법
- 학습 시에만 연산기록을 추적할 때
 - requires = True인 것만 추적

```

# 통계
running_loss += loss.item() * inputs.size(0)
running_corrects += torch.sum(preds == labels.data)
if phase == 'train':
    scheduler.step()

epoch_loss = running_loss / dataset_sizes[phase]
epoch_acc = running_corrects.double() / dataset_sizes[phase]

print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

# 모델을 깊은 복사(deep copy)함
if phase == 'val' and epoch_acc > best_acc:
    best_acc = epoch_acc
    best_model_wts = copy.deepcopy(model.state_dict())

print()

time_elapsed = time.time() - since
print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s')
print(f'Best val Acc: {best_acc:.4f}')

# 가장 나은 모델 가중치를 불러옴
model.load_state_dict(best_model_wts)
return model

```

5. Test set을 사용하여 신경망이 잘 훈련했는지 검사

```

model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                      num_epochs=25)

```

Out:

```
Epoch 0/24
-----
train Loss: 0.5926 Acc: 0.6639
val Loss: 0.2288 Acc: 0.9412

Epoch 1/24
-----
train Loss: 0.4602 Acc: 0.8238
val Loss: 0.2509 Acc: 0.9020
```

```
Epoch 24/24
-----
train Loss: 0.3253 Acc: 0.8648
val Loss: 0.2031 Acc: 0.9477

Training complete in 0m 32s
Best val Acc: 0.954248
```

▼ CNN

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/15632812-6a4f-459d-8a62-cd55be034aad/CNN_Practice.pdf

▼ Batch Normalization

배치 정규화(Batch Normalization)

gaussian37's blog

 <https://gaussian37.github.io/dl-concept-batchnorm/>

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad // \text{mini-batch variance}$$

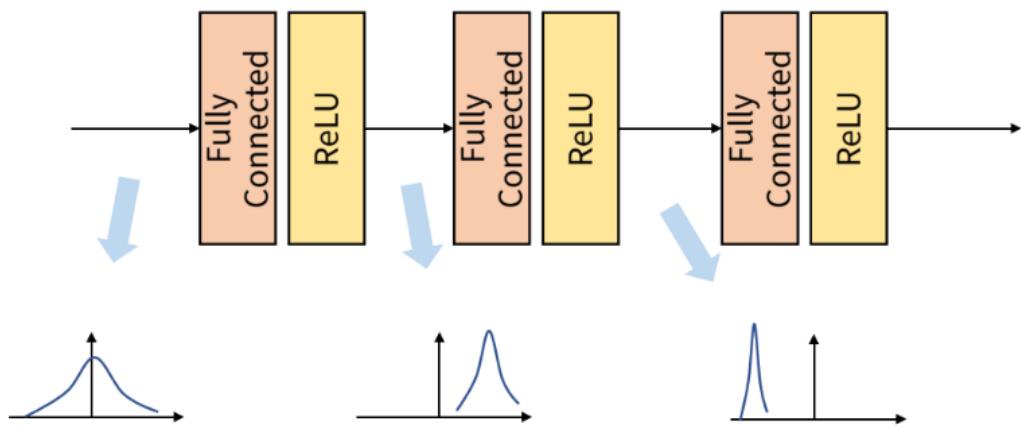
$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- Normalization 하는 이유 :

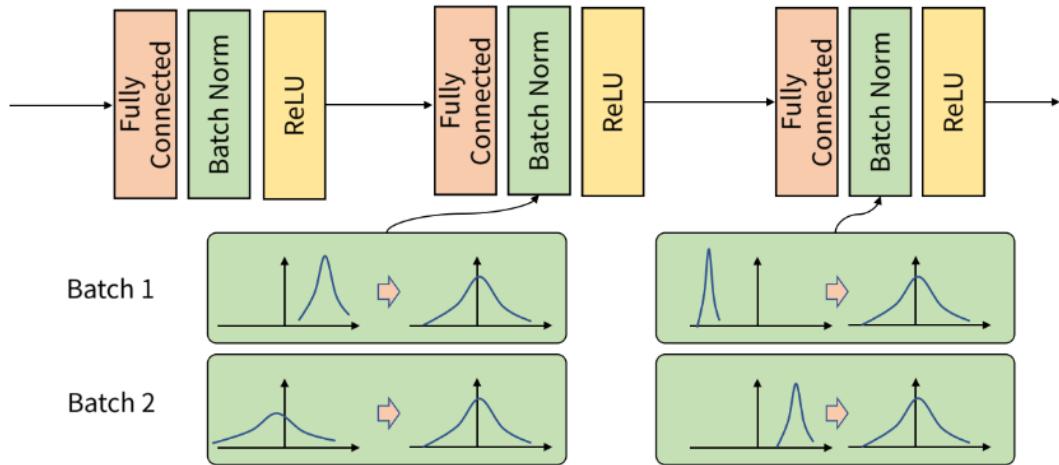
- 데이터가 크다면 전량을 조사할 수 없다
- 사용하는 데이터가 normal distribution을 뽑을 수 없을 만큼 작다면 사용하는 것이 batch normalization이다

- 아웃라이어가 학습을 방해하기 때문에 batch normalization
- batch 단위로 하기 때문에 Batch Normalization
- Internal Covariant Shift



- **Batch** 단위로 학습을 하게 되면 발생하는 문제점이 있는데 이것이 **Internal Covariant Shift** 이다.
- 먼저 Internal Covariant Shift 의미를 알아보면 위 그림과 같이 **학습 과정에서 계층 별로 입력의 데이터 분포가 달라지는 현상을 말한다.**
- 각 계층에서 입력으로 feature를 받게 되고 그 feature는 convolution이나 위와 같이 fully connected 연산을 거친 뒤 activation function을 적용
- 그러면 **연산 전/후에 데이터 간 분포가 달라질 수가 있습니다.**
- 이와 유사하게 Batch 단위로 학습을 하게 되면 **Batch 단위간에 데이터 분포의 차이**가 발생할 수 있음
- 즉, **Batch 간의 데이터가 상이하다고 말할 수 있는데 위에서 말한 Internal Covariant Shift 문제**
- 이 문제를 개선하기 위한 개념이 **Batch Normalization** 개념이 적용

Batch Normalization



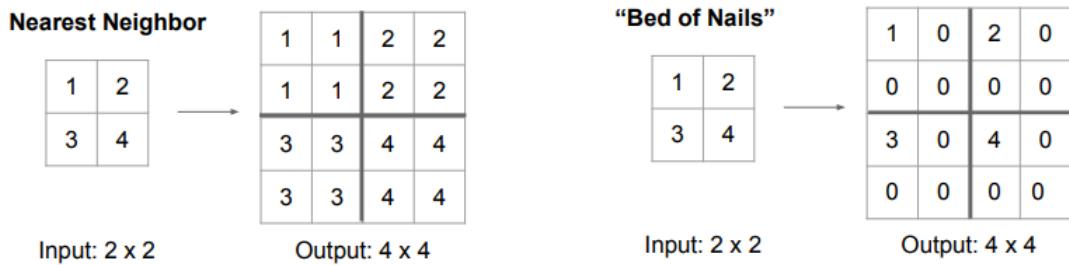
- batch normalization은 학습 과정에서 각 배치 단위 별로 데이터가 다양한 분포를 가지더라도 **각 배치별로 평균과 분산을 이용해 정규화**하는 것.
- 위 그림을 보면 batch 단위나 layer에 따라서 입력 값의 분포가 모두 다르지만 정규화를 통하여 분포를 zero mean gaussian 형태로 만든다.
- 그러면 평균은 0, 표준 편차는 1로 데이터의 분포를 조정할 수 있음.
- 여기서 중요한 것은 Batch Normalization은 학습 단계와 추론 단계에서 조금 다르게 적용되어야 한다.

▼ Object Detection and Image Segmentation

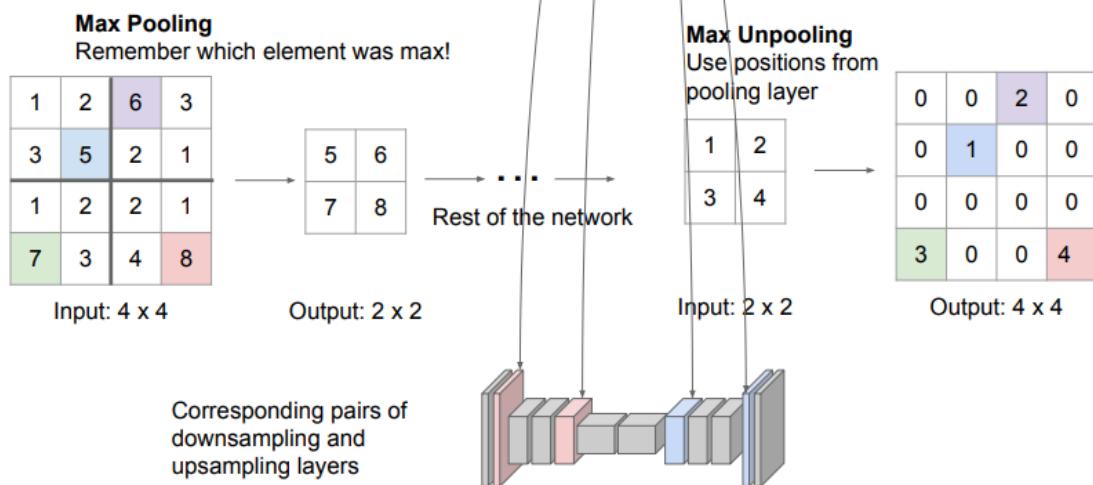
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b23b2162-78e9-4bbd-82ae-69c7e46b9e31/lecture_9_jiajun.pdf

- 이미지 맵이 줄어드는 이유
 - 연산량을 줄이기 위해
- 줄여가는 것 > Downsample, Subsampling
- 키워가는 것 > Upsampling

In-Network upsampling: “Unpooling”

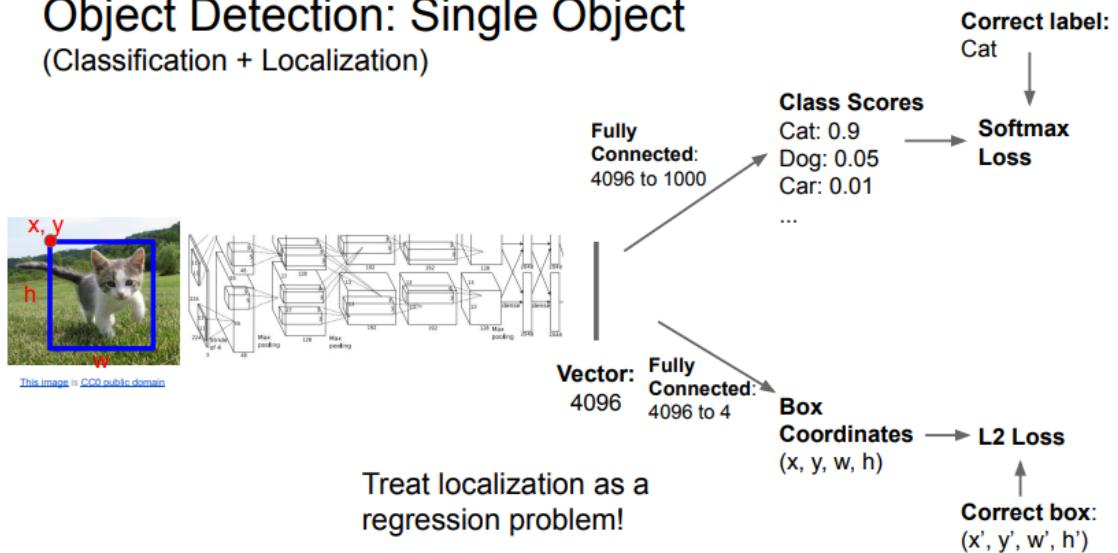


In-Network upsampling: “Max Unpooling”



- maxpooling 시의 위치를 기억
- 그 위치에 unpooling

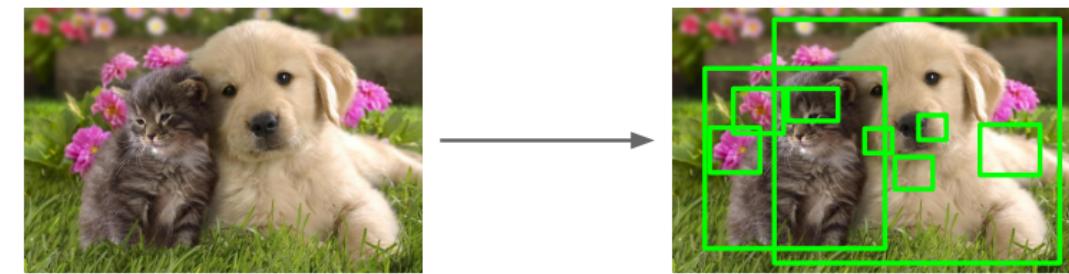
Object Detection: Single Object (Classification + Localization)



- Object 할 객체의 개수를 모르는 문제 발생
 - 이것을 알기 위한 방법

Region Proposals: Selective Search

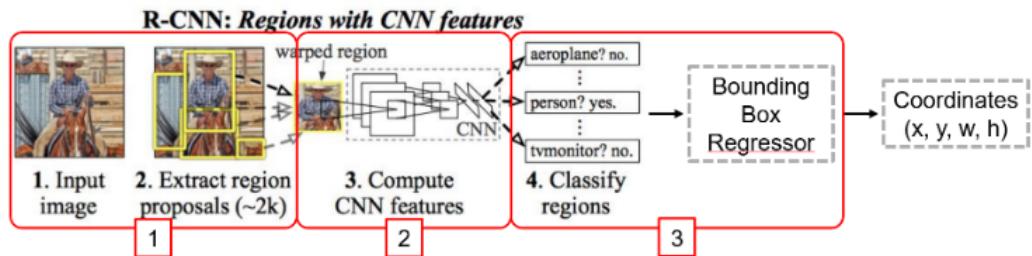
- Find “blobby” image regions that are likely to contain objects
- Relatively fast to run; e.g. Selective Search gives 2000 region proposals in a few seconds on CPU



© et al. "Measuring the objectness of image windows", TPAMI 2012
© et al. "Selective Search for Object Recognition", IJCV 2011

- 색이 바뀌는 지점에 바운딩 박스
- 바운딩 박스 후보를 줄일 수 있음

▼ R-CNN



1. Hypothesize Bounding Boxes (Proposals)

- Image로부터 Object가 존재할 적절한 위치에 Bounding Box Proposal (Selective Search)
- 2000개의 Proposal이 생성됨.

2. Resampling pixels / features for each boxes

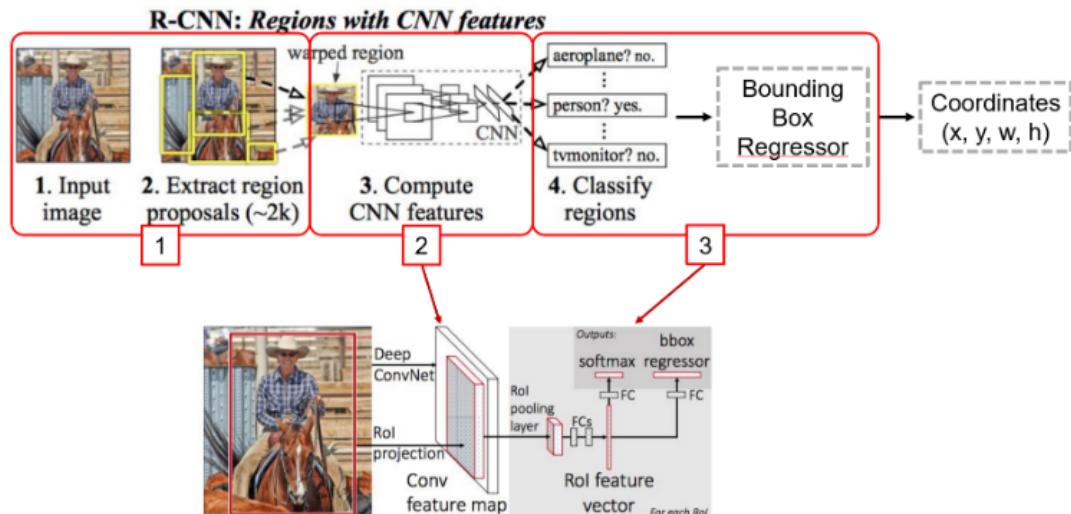
- 모든 Proposal을 Crop 후 동일한 크기로 만듦 (224x224x3)

3. Classifier / Bounding Box Regressor

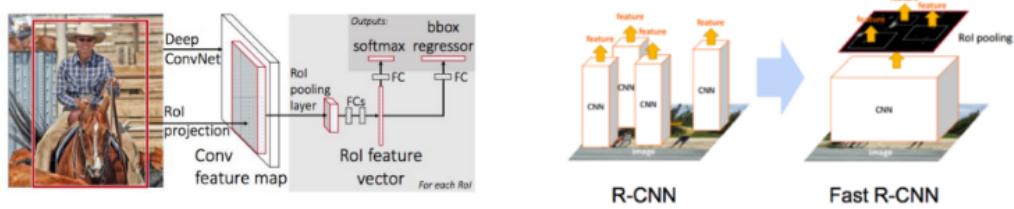
- 위의 영상을 Classifier와 Bounding Box Regressor로 처리
- 하지만 모든 Proposal에 대해 CNN을 거쳐야 하므로 연산량이 매우 많은 단점이 존재

▼ Fast R-CNN

- R-CNN 구조



- Fast R-CNN 구조

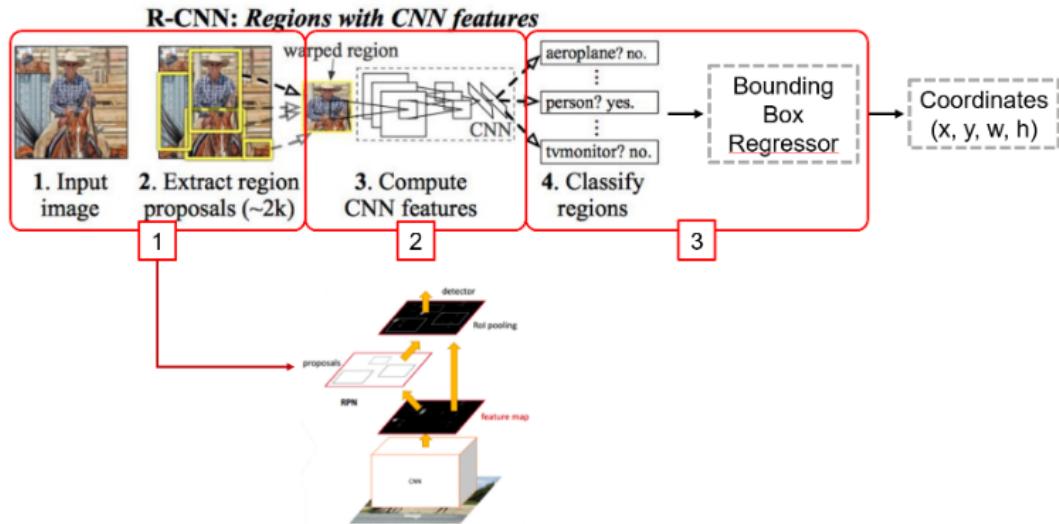


Fast R-CNN 구조

- Fast R-CNN은 모든 Proposal이 네트워크를 거쳐야 하는 R-CNN의 병목(bottleneck) 구조의 단점을 개선하고자 제안 된 방식
- 가장 큰 차이점은, 각 Proposal들이 CNN을 거치는것이 아니라 전체 이미지에 대해 CNN을 한번 거친 후 출력 된 특징 맵(Feature map)단에서 객체 탐지를 수행
- R-CNN
 - Extract image regions
 - 1 CNN per region(2000 CNNs)
 - Classify region-based features
 - Complexity: $\sim 224 \times 224 \times 2000$
- Fast R-CNN
 - 1 CNN on the entire image
 - Extract features from feature map regions
 - Classify region-based features
 - Complexity: $\sim 600 \times 1000 \times 1$
 - ~ 160 x faster than R-CNN
- 하지만 Fast R-CNN에서 Region Proposal을 CNN Network가 아닌 Selective search 외부 알고리즘으로 수행하여 병목현상 발생

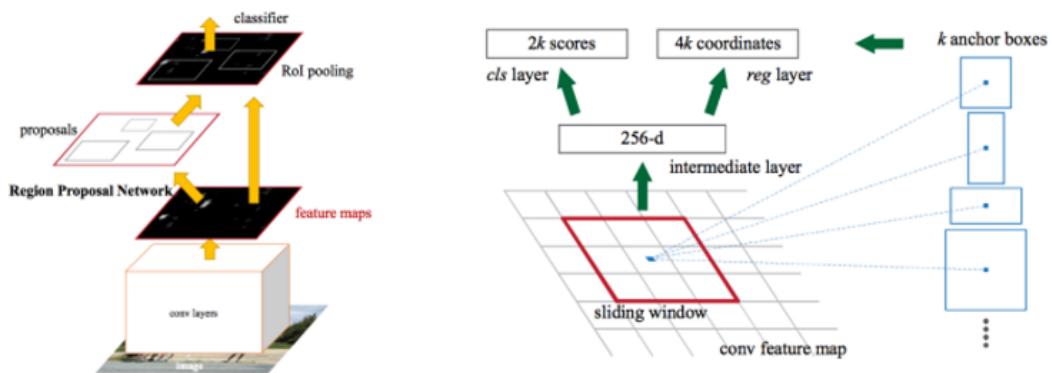
▼ Faster R-CNN

- R-CNN 구조

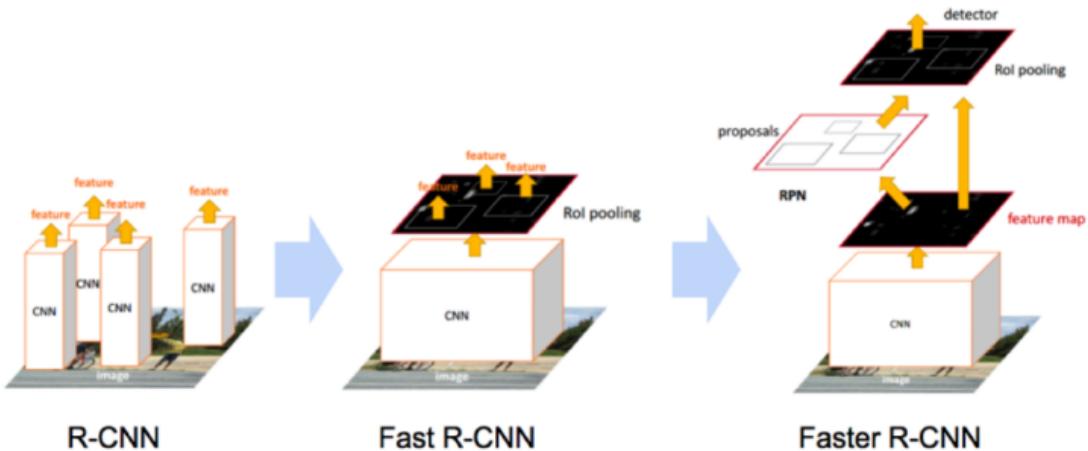


- Region Proposal을 RPN이라는 네트워크를 이용하여 수행(병목현상 해소)

- Faster R-CNN 구조



- Region Proposal 단계에서의 bottleneck 현상 제거
 - 해당 단계를 기존의 Selective search 가 아닌 CNN(RPN)으로 해결
- CNN을 통과한 Feature map에서 슬라이딩 윈도우를 이용해 각 지점(anchor)마다 가능한 바운딩 박스의 좌표와 그 점수를 계산
- 2:1, 1:1, 1:2의 종횡비(Aspect ratio)로 객체를 탐색



R-CNN 계열 구조 비교

System	Time	07 data	07 + 12 data
R-CNN	~ 50s	66.0	-
Fast R-CNN	~ 2s	66.9	70.0
Faster R-CNN	~ 198ms	69.9	73.2

Detection mAP on PASCAL VOC 2007 and 2012, with VGG-16 pre-trained on ImageNet Dataset

▼ YOLO

You Only Look Once

You Only Look Once. YOLO

너드팩토리에서 운영하는 블로그입니다.

☞ <https://blog.nerdfactory.ai/2021/05/06/You-Only-Look-Once.-YOLO.html>



Loss Function (sum-squared error)

loss function:

$$\begin{aligned}
 & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
 & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
 & + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
 & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
 & + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \quad (3)
 \end{aligned}$$

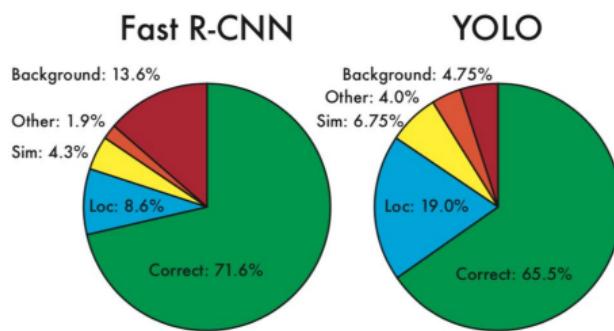
model. We use sum-squared error because it is easy to optimize, however it does not perfectly align with our goal of maximizing average precision. It weights localization error equally with classification error which may not be ideal. Also, in every image many grid cells do not contain any object. This pushes the “confidence” scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, causing training to diverge early on.

To remedy this, we increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that don’t contain objects. We use two parameters, λ_{coord} and λ_{noobj} to accomplish this. We set $\lambda_{\text{coord}} = 5$ and $\lambda_{\text{noobj}} = .5$.

$$\lambda_{\text{coord}} = 5, \lambda_{\text{noobj}} = 0.5$$

- 큰 박스와 작은 박스의 차이를 줄이기 위해
 - 루트 사용

Error Analysis



Loc: Localization Error

Correct class,

.1 < IOU < .5

Background:

IOU < 0.1

Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

- YOLO는 localization 많이 틀림
- 그리드 형식을 사용하고
- 1-Stage이기 때문

▼ 실습(객체 검출 미세조정(Finetuning))

TorchVision 객체 검출 미세조정(Finetuning) 튜토리얼

본 튜토리얼에서는 Penn-Fudan Database for Pedestrian Detection and Segmentation 데이터셋으로 미리 학습된 Mask R-CNN 모델을 미세조정 해 볼 것입니다. 이 데이터셋에는 보행자 인스턴스

🔗 https://tutorials.pytorch.kr/intermediate/torchvision_tutorial.html

파이토
한국 사용자

Google Colaboratory

🔗 https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/torchvision_finetuning_instance_segmentation.ipynb



▼ 실습

Google Colaboratory

🔗 https://colab.research.google.com/drive/1xO5_8cDHpJ0KamE-42xV7_qccZaUaK_J?usp=sharing



https://s3-us-west-2.amazonaws.com/secure.notion-static.com/572401d5-35ef-4739-9d75-49c4ae432dea/Object_Detection_Practice.pdf