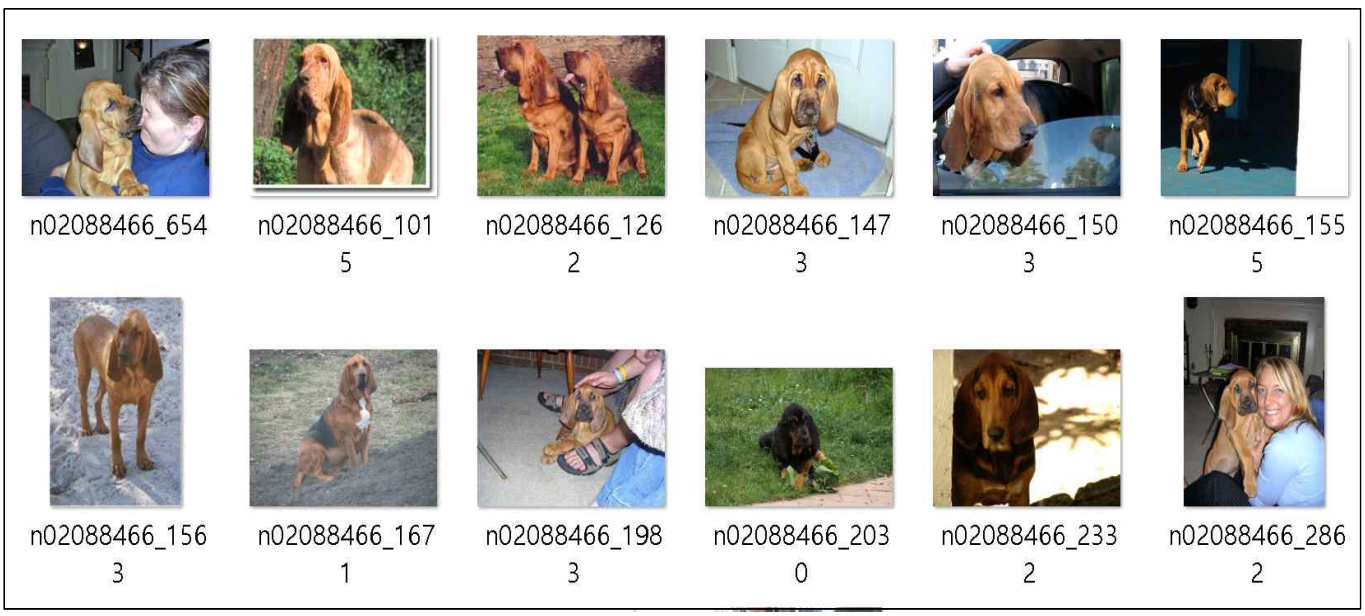
<Miami Housing Dataset> => <Dog Images> 코드 변경

# CNN vs FCN

20212245 김희서

2021.12.22


kaggle : https://www.kaggle.com/yaswanthgali/dog-images



## dataloader.py

```
from PIL import Image
import glob
import os

from torch.utils.data import Dataset
```

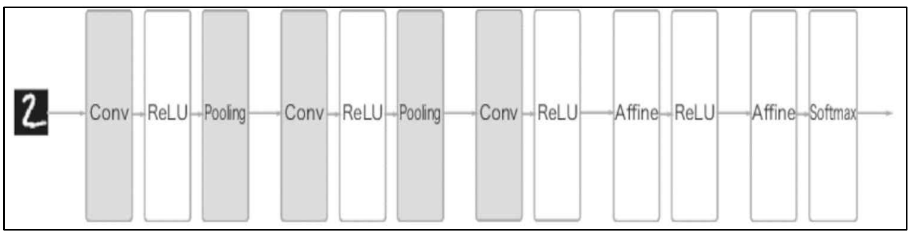
PIL : Python imaging Library
- 파이썬 인터프리터에 다양한 이미지 파일 형식을 지원하고 강력한 이미지 처리와 그래픽 기능을 제공하는 자유 오픈 소스 소프트웨어 라이브러리.

```
class TrainSet(Dataset):
    def __init__(self, paths, lab, preproc, lab2idx, indices=None):
        self.img_paths = paths
        self.labs = lab
        self.lab2idx = lab2idx
        self.preproc = preproc
        self.indices = list(range(len(self.labs))) if indices is None else indices

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
        idx = self.indices[idx]
        img_path = self.img_paths[idx]
        img = Image.open(img_path)
        lab = self.labs[idx]

        img = self.preproc(img)
        if img.shape[0] != 3:
            img = img[:3]
        lab = self.lab2idx[lab]

        return img, lab
```

해당 코드의 경우, "dataloader.py"로, 여기서 "path, lab, preproc, lab2idx"값은 다음을 나타냅니다.

path는 데이터 경로, lab은 정답, preproc은 전처리, lab2idx는 인덱스 인코딩에 대한 정보를 넣어주는 부분입니다.

1)def __getitem__(self, idx):
데이터셋에서 특정 1개의 샘플을 가져오는 함수




CNN 구조

## custom_cnn.py

```
class CustomCNN(nn.Module):
    def __init__(self, num_cls=100, img_size=32):
        super(CustomCNN, self).__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2),
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2),
        )
        self.block3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2),
        )
        self.block4 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2),
        )
        self.block5 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2),
        )
        self.fc1 = nn.Linear(512, 256)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(256, num_cls)
```

Convolution Neural Network(CNN)의 기본구조는 아래의 사진과 같이 "Convolution layer - Pooling layer - FC layer" 순서로 되어있습니다.

따라서 "custom_cnn.py"도 그에 맞춰 작성하였습니다.
nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1) =>

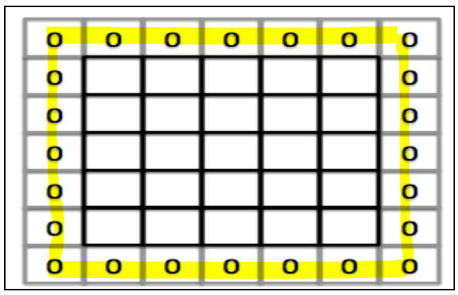
class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)

- 첫 번째 "3"은 in_channels(int)의 값으로, input image의 channel수를 나타내고, 3인 이유는 "RGB" 이미지이기 때문입니다.

두 번째 "32"인 out_channels(int)의 값으로,
convolution에 의해서 생성 된 channel의 수입니다.

nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
"kernel_size=3" : 필터의 사이즈를 나타내고, "stride=1" : 필터가 1칸씩 이동하는 것을 나타내고, "padding=1" : zero padding을 input의 양쪽에 인자만큼 해줍니다.

1) 출처 : https://wikidocs.net/57165



padding=1

이때 padding이 1이기 때문에, 위의 사진처럼 0이 양쪽에 1개씩 붙습니다.

"nn.[2]BatchNorm2d(32)"

- torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None)

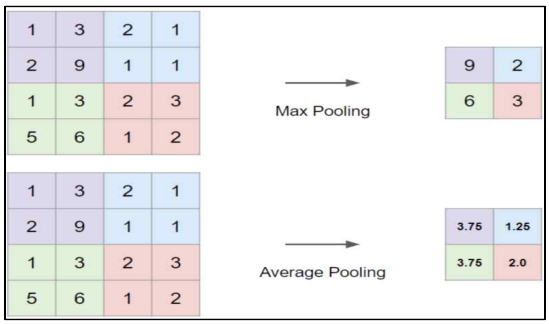
정규화를 입력에 적용하여, 평균 및 단위 분산은 0이 되고, 네트워크 정확도를 높입니다.
(학습과정에서 각 배치 단위 별로 데이터가 다양한 분포를 가지더라도, 각 배치별로 평균과 분산을 이용해 정규화 하는 것을 뜻합니다.) 해당 코드에서 32인 이유는 앞서 "nn.Conv2d"에서 out_channels의 값이 "32"가 되기 때문입니다.

"nn.ReLU()"

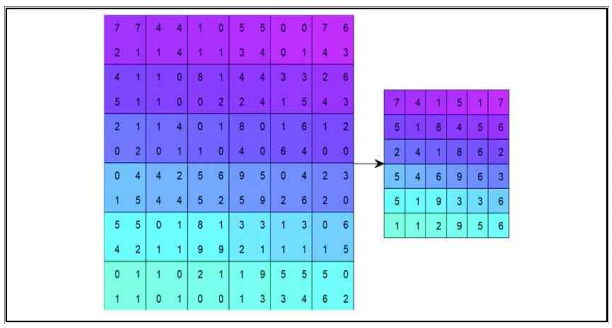
- ReLU함수의 경우, 위의 CNN구조에서 볼 수 있는 것처럼 ReLU함수를 사용하였습니다.

"nn.[3]AvgPool2d(2, stride=2)"

- Pooling(풀링)의 종류에는 Max Pooling, Average Pooling이 있는데, 이때 Max Pooling은 대상 영역에서 최댓값을 취하는 것이고 Average Pooling의 경우, 평균을 계산합니다.




Max Pooling vs Average Pooling



① stride = 2, kernels size = 2

위의 그림을 보면 Max Pooling의 경우, 보라색 부분에는 "1, 2, 3, 9"가 있는 데, 이 중 가장 큰 값인 "9"가 되고, "Average Pooling"의 경우 (1+2+3+9)/4=3.75가 됩니다. 이를 통해, 4*4가 2*2가 되어, 2차원 데이터의 가로, 세로 방향의 공간을 줄이는 Pooling을 하였습니다.

torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)

①의 경우, block1의 AvgPool2d와 같은 조건인데, 필터의 크기는 2*2이고, 그 필터는 2칸씩 이동하면서 컨볼루션을 진행하기 때문에, 가로, 세로 모두 1/2이 되는 것을 확인할 수 있습니다.

+) 2D Average Pooling : This block reduces the 1. size of the data, 2. the number of parameters, 3. the amount of computation needed, and 4. it also controls overfitting.

---

2) 출처 : https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html
3) 출처 : https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html

## resnet.py

```python
def conv3x3(in_planes: int, out_planes: int, stride: int = 1, groups: int = 1, dilation: int = 1) -> nn.Conv2d:
    """3x3 convolution with padding"""
    return nn.Conv2d(
        in_planes,
        out_planes,
        kernel_size=3,
        stride=stride,
        padding=dilation,
        groups=groups,
        bias=False,
        dilation=dilation,
    )
```

```python
def conv1x1(in_planes: int, out_planes: int, stride: int = 1) -> nn.Conv2d:
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False)
```

resnet.py의 경우, "https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py"의 코드를 사용하였고, "resnet"의 경우, 이번 프로젝트에서 사용한 네트워크입니다. => CNN 활용코드

우선 resnet 코드를 이해하기 위해서는 Residual block에 대해서 이해해야 합니다.
위 두 코드에서 "in_planes : int"는 입력 필터 개수, "out_planes : int"는 출력 필터 개수를 의미합니다.
- "conv3x3"의 [4]"groups"는 입력과 출력의 관계를 제어하는 것으로 default값은 1입니다.

- groups값이 1일 경우, 모든 입력들이 모든 출력에 컨볼루션.

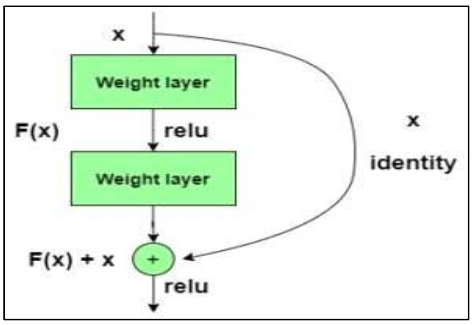
  groups=in_channels일 경우, 각각의 입력 채널은 자체 필터 세트로 컨볼루션. size : $\frac{out\_channels}{in\_channels}$

- "dilation" : 커널 원소간의 거리(kernel points 사이의 간격 제어). 값이 클수록 같은 파라미터수로 더 넓은 범위를 파악.

```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError("BasicBlock only supports groups=1 and base_width=64")
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # self.conv1, self.downsample layers 둘다 stride가 1이 아닐 때, input을 downsample한다.
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
```

```python
def forward(self, x: Tensor) -> Tensor:
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```

(expansion: int =1 : 이 부분은 시뮬레이션 돌리다가 가장 좋았던 값이 나와 설정한 값입니다.)
"class BasicBlock(nn.Module)" : __init__는 거의 모든 코드에서 볼 수 있는 것처럼 초기 설정으로, Normalization layer가 없을 때 ("if norm_layer is None" 구문으로) "nn.BatchNorm2d"가 norm_layer에 입력됩니다.



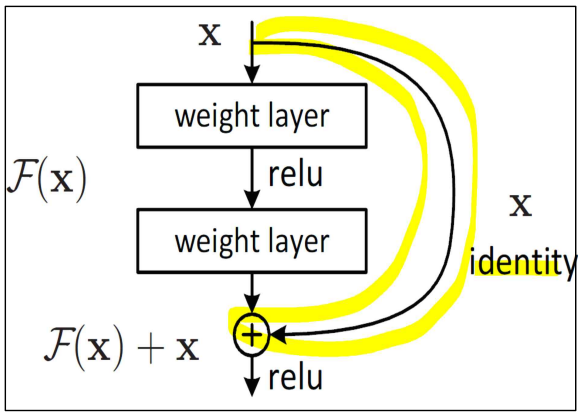

"if groups !=1 or base_width!=64"와 "if dilation >1"의 구문은 오류 발생 시 오류 원인을 알기 위해서 설정.

그리고 "BasicBlock"의 나머지 코드는 "conv1 - bn1 - relu - conv2 - bn2"의 순서로 forward에 필요한 layer를 설정합니다. 이때 "self.bn1=norm_layer(planes)"의 경우, if문으로 설정한 기본 값으로 "nn.BatchNorm2d"을 사용.

bn1이"self.conv2=conv3x3(planes, planes)"의 경우, conv2가 conv3x3을 사용한다는 것을 의미합니다.
"downsample"은 신호처리에서 자주 등장하는 개념으로, stride가 1이 아닌 경우, 사이즈를 맞춰주는 역할.
즉, forward 할 경우, 아래 그림의 "F(x)+x"의 residual을 구현할 때, F(x)와 x의 tensor 사이즈가 다를 때 사용.

4) 출처 : https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

```python
def forward(self, x: Tensor) -> Tensor:
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```




"def forward(self, x: Tensor) -> Tensor :"의 경우, layer를 앞서 언급한 "conv1 - bn1 - relu - conv2 - bn2" 순서로 진행되는 것을 볼 수 있고, "out +=identity"의 경우, 오른쪽의 그림의 표시 된 부분을 나타내며, ResNet의 block을 통과한 feature map의 값인 F(x)와 identity값인 x값을 더한 것을 의미합니다.

```python
class Bottleneck(nn.Module):
    expansion: int = 4

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        width = int(planes * (base_width / 64.0)) * groups
        # Both self.conv2 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv1x1(inplanes, width)
        self.bn1 = norm_layer(width)
        self.conv2 = conv3x3(width, width, stride, groups, dilation)
        self.bn2 = norm_layer(width)
        self.conv3 = conv1x1(width, planes * self.expansion)
        self.bn3 = norm_layer(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride
```

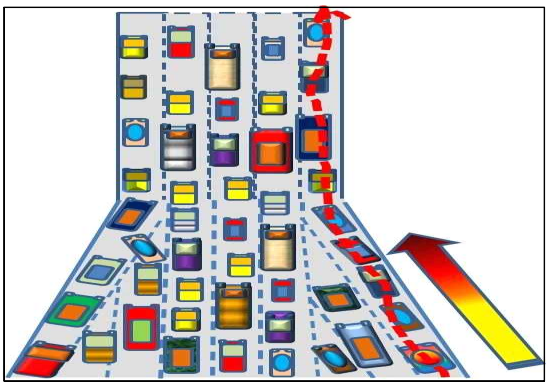
```python
def forward(self, x: Tensor) -> Tensor:
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)

    out = self.conv3(out)
    out = self.bn3(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)
```

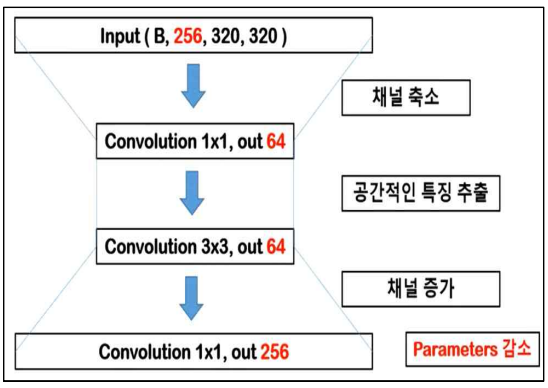
"class Bottleneck(nn.Module)" : "__init__"의 "base_width : int =64, dilation : int = 1, groups : int =1"를 통해, base_width, dilation, groups의 값을 설정.

- "width = int(planes * (base_width/64.0)) * groups" : 채널을 얼마나 줄일지 width값을 설정합니다.

- "self.conv1 = conv1x1(inplanes, width)" : inplanes인 입력 채널을 width channel로 낮추는 "conv1x1".
- "self.conv2 = conv3x3(width, width, stride, groups, dilation)" : 앞서 낮춰진 채널을 "conv3x3".
- "self.conv3 = conv1x1(width, planes * self.expansion)" : width 채널을 output channel(planes) * self.expansion (expansion channel)로 확장.

conv1x1 : "1x1 convolution layer"의 경우, 1) channel 수 조절, 2) 연산 량 감소, 3) 비선형성.
- 1x1 convolution : 직관적으로 1x1 크기를 가지는 convolution filter를 사용한 convolution layer. (차원 축소)





- class의 “Bottleneck”라는 이름이 붙인 이유 : 이 부분에서 동작하는 방법을 위의 그림처럼 보인 것입니다.
- “convolution 1x1, out 64”로 채널을 압축(축소)합니다. 이를 진행하는 이유는 연산량을 줄이기 위해서입니다.

- 그리고 공간 특성을 추출하기 위해 1x1 convolution보다 9배 연산량이 많은 3x3 convolution을 합니다.
- 마지막으로 1x1 convolution으로 채널을 다시 증가시켜, 연산량을 줄입니다.

- 이 과정의 모습이 차가 막히는 병목현상의 모습과 비슷하다고 하여 “Bottleneck”이라고 붙여졌습니다.

```
class ResNet(nn.Module):
    def __init__(
        self,
        block: Type[Union[BasicBlock, Bottleneck]],
        layers: List[int],
        num_classes: int = 1000,
        zero_init_residual: bool = False,
        groups: int = 1,
        width_per_group: int = 64,
        replace_stride_with_dilation: Optional[List[bool]] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
        **kwargs,
    ) -> None:
        super().__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError(
                "replace_stride_with_dilation should be None "
                f"or a 3-element tuple, got {replace_stride_with_dilation}"
            )
```

```
self.groups = groups
self.base_width = width_per_group
self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3, bias=False)
self.bn1 = norm_layer(self.inplanes)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2, dilate=replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2, dilate=replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2, dilate=replace_stride_with_dilation[2])
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)
```

“class RseNet(nn.Module)” : “num_classes : int = 1000” = 최종 출력 피쳐는 1000, “zero_init_residual : bool = False” = Block을 지날 때, weight값을 0으로 초기화.

- “self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3, bias=False)”, “self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)” = 7*7 conv와 3*3 maxpool을사용한 것을 볼 수 있습니다.

```
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2, dilate=replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2, dilate=replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2, dilate=replace_stride_with_dilation[2])
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)
```

위의 코드는 레이어를 만드는 함수인 “_make_layer”를 사용하여, residual block을 나타낸 것으로, “64-128-256-512”을 통해, layer를 지날 때마다 2배씩 늘어나도록 설정하였고, layer4를 지난 후, “AdaptiveAvgPool2d”를 통해, (n, 512, 1,1)의 텐서를 만들고, 마지막으로 “nn.Linear”을 연결합니다.

- “self.layer2=self._make_layer(block, 128, layers[1], stride=2, dilate=replace_stride_width_dilation[0])” = 출력 채널을 128 * expansion을 할 것이고, layers[2]의 개수만큼 block을 쌓는 것을 의미합니다.

- “AdaptiveAvgPool2d”를 하는 이유는 각 채널의 이미지를 (1, 1)사이즈로 Average Pooling하기 위한 것입니다.

```
for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

# Zero-initialize the last BN in each residual branch,
# so that the residual branch starts with zeros, and each residual block behaves like an identity.
# This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
if zero_init_residual:
    for m in self.modules():
        if isinstance(m, Bottleneck):
            nn.init.constant_(m.bn3.weight, 0)  # type: ignore[arg-type]
        elif isinstance(m, BasicBlock):
            nn.init.constant_(m.bn2.weight, 0)  # type: ignore[arg-type]
```

```
def _make_layer(
    self,
    block: Type[Union[BasicBlock, Bottleneck]],
    planes: int,
    blocks: int,
    stride: int = 1,
    dilate: bool = False,
) -> nn.Sequential:
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(
        block(
            self.inplanes, planes, stride, downsample, self.groups, self.base_width, previous_dilation, norm_layer
        )
    )
    self.inplanes = planes * block.expansion
```

```python
for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

# Zero-initialize the last BN in each residual branch,
# so that the residual branch starts with zeros, and each residual block behaves like an identity.
# This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
if zero_init_residual:
    for m in self.modules():
        if isinstance(m, Bottleneck):
            nn.init.constant_(m.bn3.weight, 0)  # type: ignore[arg-type]
        elif isinstance(m, BasicBlock):
            nn.init.constant_(m.bn2.weight, 0)  # type: ignore[arg-type]
```

```python
def _make_layer(
    self,
    block: Type[Union[BasicBlock, Bottleneck]],
    planes: int,
    blocks: int,
    stride: int = 1,
    dilate: bool = False,
) -> nn.Sequential:
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(
        block(
            self.inplanes, planes, stride, downsample, self.groups, self.base_width, previous_dilation, norm_layer
        )
    )
    self.inplanes = planes * block.expansion
```

“nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")” = “fan_out”의 경우, 역전파할 때, 가중치 전체의 크기를 보존시킵니다.

- “nn.init.constant-(m.bn3.weight,0)” : zero_init_residual을 설정할 경우, 마지막 BN에서 weight를 0으로 초기화.

- “def _make_layer” : make_layer는 레이어를 만드는 함수로, 최종 출력, 최초 입력사이즈를 맞도록 “downsample”을 합니다.

- “if stride != 1 or self.inplanes != planes * block.expansion:” : downsample이 필요할 때 layer 생성.
즉, stride가 1이 아닌 경우, 크기가 줄어들거나 input planes(self.inplanes)가 planes * block.expansion의 크기와 같지 않을 때, stride는 1이 아닌 값으로 down sample 합니다.

-“layers.append” : 이를 통해 layer에 hyper parameter의 block을 추가하고, “self.inplanes = planes * block.expansion”를 통해, 다음 block을 추가하기 위해서, 차원을 맞춥니다.

-“return nn.Sequential(*layers)” : 이때 layers는 그동안 쌓은 block들을 나타내고, 이를 NN로 나열하기 위해서 Sequential을 사용하였습니다.

```python
def _forward_impl(self, x: Tensor) -> Tensor:
    # See note [TorchScript super()]
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)

    return x
```

```python
def resnet18(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> ResNet:
    r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.03385.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    resnet18 = _resnet("resnet18", BasicBlock, [2, 2, 2, 2], pretrained, progress, **kwargs)
    resnet18_pretrained = torchvision.models.resnet18(pretrained=True)
    resnet18.load_state_dict(resnet18_pretrained.state_dict())
    return resnet18
```

- 이때 layer2는 128에서 512채널이 되는 데, strie=2에 의해 (512*28*28)이 됩니다.
과정이 반복되어 “layer4”를 지나고 “avgpool”를 지나면, “2048*1*1”이 되고, “x=self.fc(x)”의 값은 2048이 됩니다.

이때 “return x”의 값은 “1000”이 되는데, 그 이유는 앞서 최종출력 피쳐 값으로(num_classes) 1000을 설정하였기 때문입니다.

오른쪽 코드 : 조건에 따라 값을 출력하기 위해서, resnet에 하이퍼 파라미터 값을 정의합니다. : resnet18
(18, 50은 레이어 개수를 나타냅니다.)

코드를 실행하기 위해서 “python train.py --train_dir ./images/images --test_dir ./images/images --device cpu --arch resfcn18”는 fcn, “python train.py --train_dir ./images/images --test_dir ./images/images --device cpu --arch resnet18”는 cnn을 사용합니다.

# resfcn.py

```
class ResNet18(nn.Module):
    def __init__(self, num_classes=4):
        super(ResNet18, self).__init__()
        #torchvision에서 사용가능한 미리 학습된 ResNet 불러오기
        resnet18 = torchvision.models.resnet18(pretrained=True)
        resnet18.fc = nn.Linear(512, num_classes)
        self.resnet18 = resnet18

    def forward(self, input):
        output = self.resnet18(input)  # (N, feature_map_channels, feature_map_w, feature_map_h)
        return output
```

1)**ResNet18** : 18개 층으로 이루어진 ResNet.

- **ResNet** : 2015년 ILSVRC에서 우승한 **CNN 네트워크**

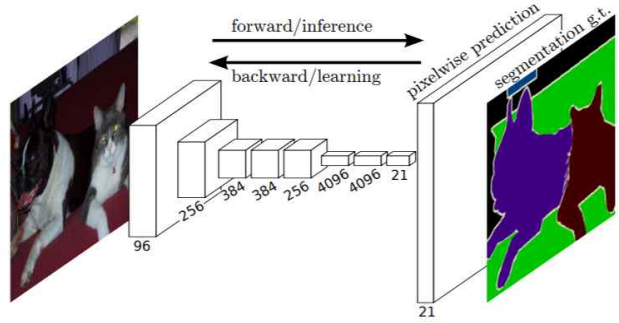
**torchvision** : torchvision은 파이토치와 함께 사용되는 컴퓨터 비전용 라이브러리로, 입력 값이 이미지이기에 사용하였습니다.

- 이때 torchvision은 Pytorch와 함께 제공되므로, 자동으로 GPU가 지원됩니다.

이 코드에서 "torchvision.models.resnet18 (**pretrained=True**)"의 경우, **torchvision**에서 미리 학습된 **ResNet**을 불러오기 위해 사용하였습니다.

```
class BasicBlock(nn.Module):
    expansion: int = 1
    def __init__(
        self,
        in_features = None,
        norm_layer = None,
    ):
        super().__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm1d
        #self.conv1와 self.downsample layer는 stride가 1이 아닐 때, input값
        self.fc1 = nn.Linear(in_features, in_features//2)
        self.bn1 = norm_layer(in_features//2)
        self.relu = nn.ReLU(inplace=True)
        self.fc2 = nn.Linear(in_features//2, in_features//2)
        self.bn2 = norm_layer(in_features//2)
        self.downsample = nn.AvgPool1d(kernel_size=3, stride=2, padding=1)

    def forward(self, x):
        identity = x
        batch_size = x.shape[0]

        out = self.fc1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.fc2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            x = x.view(batch_size, 1, -1)
            identity = self.downsample(x)
            identity = identity.view(batch_size, -1)

        out += identity
        out = self.relu(out)

        return out
```

다음 코드에서 나오는 함수들은 이미 이전에 언급한 것이고, 중요한 것은 "in_features//2"인데, 이는 fcn의 구조를 보면 알 수 있습니다.




2)Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

```
class resfcn18(nn.Module):
    def __init__(self, num_classes=100, img_size=224):
        super(resfcn18, self).__init__()
        self.img_size = img_size
        self.resize = 192
        target_size = 2048
        self.fc1 = nn.Linear((self.resize ** 2) * 3, target_size * 2)
        self.bn1 = nn.BatchNorm1d(target_size * 2)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.AvgPool1d(kernel_size=3, stride=2, padding=1)
        self.block1 = nn.Sequential(BasicBlock(target_size//(2**0)),
                                    BasicBlock(target_size//(2**1)),
                                    )
        self.block2 = nn.Sequential(BasicBlock(target_size//(2**2)),
                                    BasicBlock(target_size//(2**3)),
                                    )
        self.block3 = nn.Sequential(BasicBlock(target_size//(2**4)),
                                    BasicBlock(target_size//(2**5)),
                                    )
        self.block4 = nn.Sequential(BasicBlock(target_size//(2**6)),
                                    BasicBlock(target_size//(2**7)),
                                    )
        self.fc2 = nn.Linear(target_size//(2**8), num_classes)
```

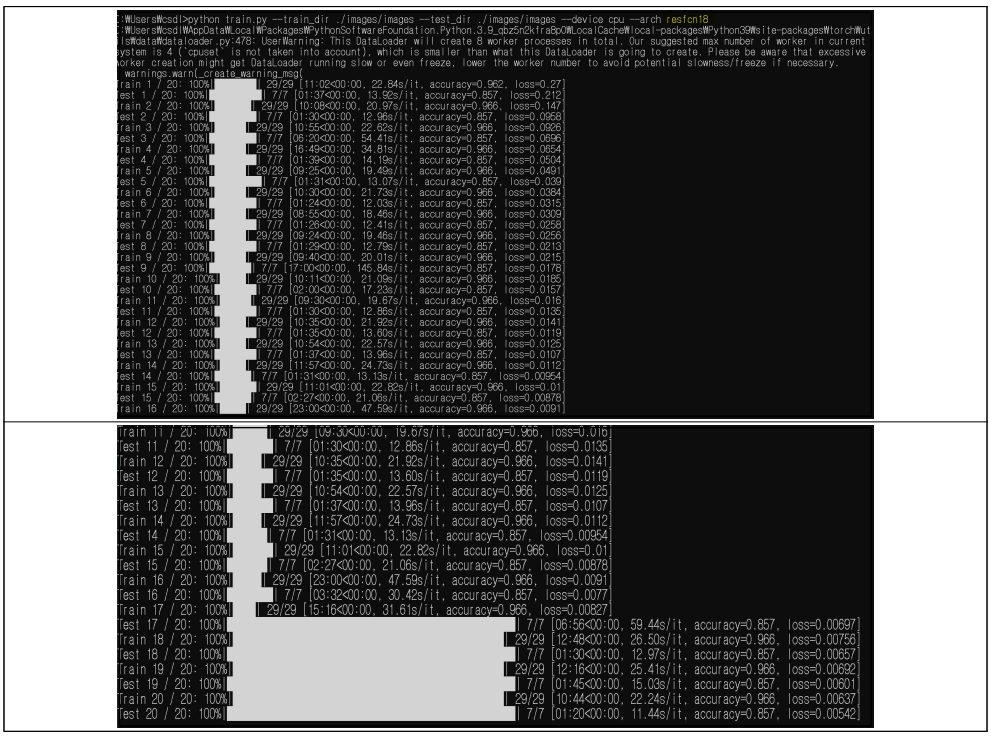
위의 코드는 "BasicBlock"과 달리 기존 architecture를 바꿔 옵션을 주어 설정한 값입니다.
이때 "//(2**3)", "//(2**7)"은 BasicBlock과 달리 임의로 설정한 값 중 성능이 잘 나왔기에 사용한 값들입니다.

---

1) 출처 : https://hnsuk.tistory.com/31
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - "Deep Residual Learning for Image Recognition"
https://arxiv.org/abs/1512.03385
2) Jonathan Long - "Fully Convolutional Networks for Semantic Segmentation"

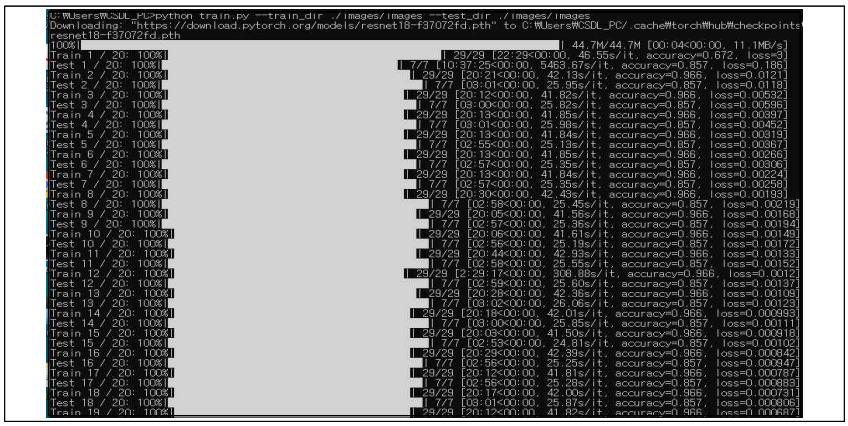
<시뮬레이션 값>

1. python train.py --train_dir ./images/images --test_dir ./images/images --device cpu --arch resfcn18



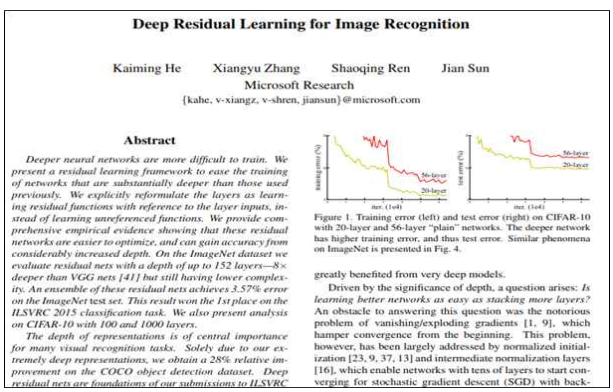
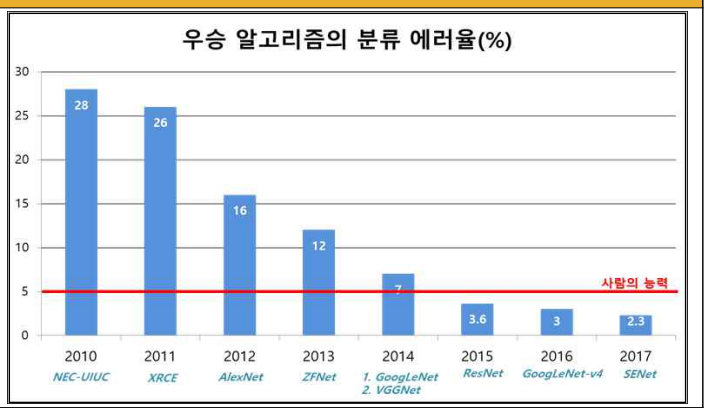

resfcn18 : fcn

2. python train.py --train_dir ./images/images --test_dir ./images/images



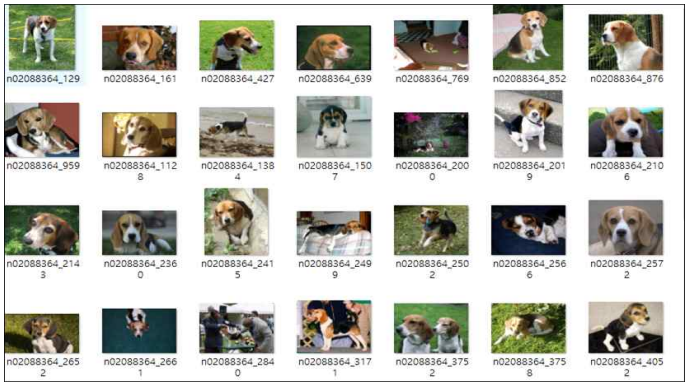

"python train.py --train_dir ./images/images --test_dir ./images/images --device cpu --arch resnet18"는 CNN을 사용합니다. => 결과 : "resnet18"을 사용한 CNN이 성능이 좋았습니다. loss 값이 더 좋은 결과 값을 보여주었습니다.

## <결론>

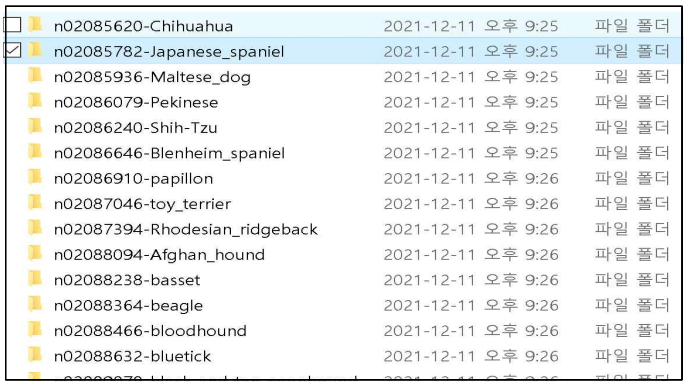
| CNN | ILSVRc -resnet 2015 |
|---|---|
|  |  |

이번 프로젝트는 CNN과 이전 프로젝트 FCN의 성능차이를 비교하는 프로젝트입니다.
따라서 기존의 FCN프로젝트의 주제로 했던 "Miami Housing Dataset 데이터"에서 "Dog Images"로 kaggle 데이터를 바꿨습니다. 이미지분석에 CNN이 적합하기 때문에.

LSTM과 비교하면, 이 주제의 프로젝트의 경우, 아쉬웠던 점이 많이 있습니다.



kaggle 데이터 값 (1)




kaggle 데이터 값 (2)

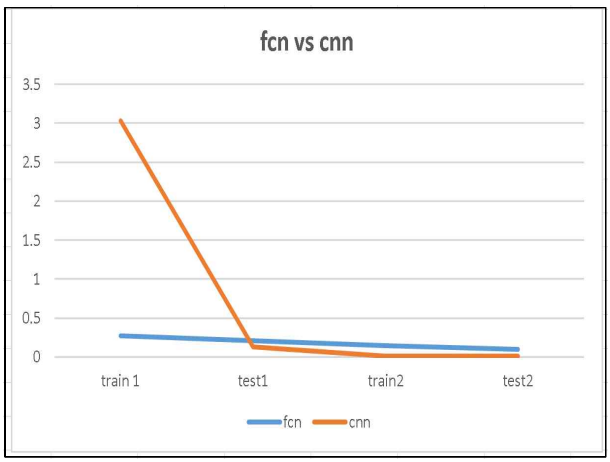
- 입력 값을 잘못 선정했다는 점입니다. 주가 분석을 했던 LSTM의 경우, 주가 예측을 어느 정도 구색을 갖춘 결과가 나와, LSTM을 잘 활용했다는 것을 말할 수 있지만, CNN의 경우, 이미지 값과 주제를 잘못 정한 것 같습니다.

우선, 1. CNN을 사용하였기에 입력 값으로 이미지를 사용하는 것은 어쩔 수 없었지만, "dog vs cat"과 같은 결과 값을 단순화하였다면 학습시간도 덜 걸리고, 결과 값도 더 정확하지 않았을까? 싶었습니다.

그리고 2. 주가데이터와 같은 숫자 csv파일이나, 글자와 같은 입력과 달리 이미지 입력 값은 시간이 너무 오래 걸려, 파라미터를 바꿔가면서, 성능 분석하기에 물리적인 시간이 부족했습니다.

그 점이 많이 아쉬웠고, 입력 값에 맞춰 코드를 수정하기는 했지만, 큰 틀은 기존에 있던 유명 코드인 "ResNet"을 사용하였기에, COPY가 되지 않을까? 하는 점도 완성한 후에 약간 걱정되는 부분이기도 합니다.

+) CNN인 resnet18의 시뮬레이션 값을 출력 후, 캡처를 하지 않아, 보고서에는 첨부하지 못했습니다.
그러나 아래의 excel값처럼 train, test 1, 2에서도 cnn의 값이 성능이 좋다는 것을 알 수 있습니다. 만일 학습이 다 되면, 추가적으로 사진을 제출하도록 하겠습니다. (2021.12.22. 오후11시16분 기준)




python train.py --train_dir ./images/images --test_dir ./images/images --device cpu --arch resnet18s