합성곱 신경망(Convolutional Neural Network)

기존의 다층 퍼셉트론은 완전 연결 계층으로 구성되어 있다. 완전 연결 계층이란 모든 퍼셉트론 쌍이 빠짐 없이 서로 연결되어 있는 것이다. 은닉 계층과 출력 계층 모두 완전 연결 계층이다. 완전 연결 계층의 문제점은

- 가중치 행렬 크기에 따른 메모리 부담
- 많은 파라미터로 인한 느린 학습 속도
- 많은 파라미터 학습을 위해 많은 양의 데이터 필요

등이 있다.

이런 메모리 부담과 기하급수적으로 증가하는 학습 시간을 줄이기 위한 방법으로 CNN(Convolutional Neural Network)라는 개념이 등장하였다. CNN은 작은 가중치 텐서를 이미지 모든 영역에 반복 적용하고, 풀링을 통해서 처리할 이미지 해상도를 단계적으로 감소시켜 신경망 파라미터 수를 줄이면서도 품질을 향상시킬 수 있는 기법이다.

패딩(Padding) : 입력 픽셀에 덧대는 기법.

패딩을 적용하지 않을 시에는 가장자리에 있는 픽셀들이 상대적으로 덜 적용되어 결과에 영향을 미칠 수 있는데 이를 공평하게 반영하기 위해서 주로 사용한다.

Same Padding :

- 출력 계산에 영향 없도록 입력 행렬 가장자리에 0으로 값을 덧댄다.
- 크기가 입력과 출력이 같아 Same Padding이라고 한다. 이는 연산을 복잡하게 할 수 있다.

Valid Padding

- 가장자리에 있는 픽셀이 덜 반영될 수 있다.
- 입력 크기에 비해 출력 크기가 줄어듬
- 계산에 반영되는 유효 입력 픽셀 수가 모든 출력 픽셀에 공평하게 적용됨.

Stride

- 일정 간격으로 건너 뛰면서 출력 픽셀 생성

풀링(Pooling)

- 커널 영역의 입력 픽셀값들의 대푯값 구해 출력 픽셀로 생성
- 대푯값 : 보통 최댓값(Max Pooling)이나 평균값(Avg Pooling) 이용

기대효과 : 정보 집약 효과, 패턴 포착의 강건성

입력, 커널, 보폭간의 관계

- 커널 크기와 보폭이 서로 다른 경우
  - 겹치거나 비는 부분 발생.
- 커널 크기 = 보폭
  - 입력 범위를 벗어날 수 있다. 가장자리 부분에서 조심스러운 처리 필요.
  - 효율적 처리 가능
- 커널 크기 = 보폭(입력 크기의 약수인 경우)
  - 깨끗하게 분할 가능(입력 범위를 벗어나지 않음)
  - 매우 간단하고 효율적인 처리 가능

특징맵 :

- 합성곱 계층에서 출력되는 픽셀 이미지
- 커널이 포착한 특정 패턴이나 특징의 표현
- 이미지에는 다양한 패턴과 특징 존재

채널

- 합성곱 신경망이 다루는 이미지 데이터에 추가되는 차원
- 최초 입력 이미지의 채널 (흑백 : [가로,세로,1], 컬러 : [가로,세로,3])
- 입력 채널 수는 정해져 있으나, 출력 채널 수는 신경망의 설계에 따라 변경 가능

정보량.

- 일반적으로 CNN layer를 거치면서 채널 수 증가(정보량 증가)
- 보통 Pooling 계층 이용하여 정보량 감축

코드 및 설명

```
%run ../chap06/adam_model.ipynb
```

```
class CnnBasicModel(AdamModel): # AdamModel을 상속받음
    def __init__(self, name, dataset, hconfigs, show_maps = False):
        if isinstance(hconfigs, list) and ₩
        not isinstance(hconfigs[0], (list, int)):
            hconfigs = [hconfigs]
        self.show_maps = show_maps
        self.need_maps = False
        self.kernels = []
        super(CnnBasicModel, self).__init__(name, dataset, hconfigs) # AdamMod
        self.use_adam = True
```

: AdamModel 을 상속받고 super()를 통해서 AdamModel의 메소드를 이용하여 초기화

```
def cnn_basic_alloc_layer_param(self, input_shape, hconfig):
    layer_type = get_layer_type(hconfig)

    m_name = 'alloc_{}_layer'.format(layer_type)
    method = getattr(self, m_name)
    pm, output_shape = method(input_shape, hconfig) # hconfig(은닉 계층 얻어내

    return pm, output_shape # 파라미터 정보 및 출력 계층

CnnBasicModel.alloc_layer_param = cnn_basic_alloc_layer_param
```

```
def cnn_basic_forward_layer(self, x, hconfig, pm): # 순전파 메소드 처리
    layer_type = get_layer_type(hconfig)

    m_name = 'forward_{}_layer'.format(layer_type)
    method = getattr(self, m_name)
    y, aux = method(x, hconfig, pm)

    return y, aux

CnnBasicModel.forward_layer = cnn_basic_forward_layer
```

```
def cnn_basic_backprop_layer(self, G_y, hconfig, pm, aux): # 역전파 메소드 처리
    layer_type = get_layer_type(hconfig)

    m_name = 'backprop_{}_layer'.format(layer_type)
    method = getattr(self, m_name)
    G_input = method(G_y, hconfig, pm, aux)

    return G_input

CnnBasicModel.backprop_layer = cnn_basic_backprop_layer
```

: layer의 파라미터 생성, 순전파, 역전파 처리

```
def cnn_basic_alloc_full_layer(self, input_shape, hconfig): # Fully Connected
    input_cnt = np.prod(input_shape)
    output_cnt = get_conf_param(hconfig, 'width', hconfig)

    weight = np.random.normal(0, self.rand_std, [input_cnt, output_cnt])
    bias = np.zeros([output_cnt])

    return {'w':weight, 'b':bias}, [output_cnt]

def cnn_basic_alloc_conv_layer(self, input_shape, hconfig): # Convolution 연산
    assert len(input_shape) == 3 # 미니배치 크기 고려하지 않은 [가로, 세로, 채
    xh, xw, xchn = input_shape
    kh, kw = get_conf_param_2d(hconfig, 'ksize')
    ychn = get_conf_param(hconfig, 'chn')

    kernel = np.random.normal(0, self.rand_std, [kh, kw, xchn, ychn])
    bias = np.zeros([ychn])

    if self.show_maps: self.kernels.append(kernel)

    return {'k':kernel, 'b':bias}, [xh, xw, ychn] # 채널 수만 변경됨.

def cnn_basic_alloc_pool_layer(self, input_shape, hconfig): # max, avg pool
    assert len(input_shape) == 3
    xh, xw, xchn = input_shape
    sh, sw = get_conf_param_2d(hconfig, 'stride') # stride 처리

    assert xh % sh == 0
    assert xw % sw == 0

    return {}, [xh//sh, xw//sw, xchn]

CnnBasicModel.alloc_full_layer = cnn_basic_alloc_full_layer
CnnBasicModel.alloc_conv_layer = cnn_basic_alloc_conv_layer
CnnBasicModel.alloc_max_layer = cnn_basic_alloc_pool_layer
CnnBasicModel.alloc_avg_layer = cnn_basic_alloc_pool_layer
```

: Fully Connected layer, Convolution layer, Pooling layer관련 처리.

cnn_basic_alloc_pool_layer에서는 stride, avg pooling, max pooling 을 위한 처리를 하였다.

```
def get_layer_type(hconfig):
    if not isinstance(hconfig, list): return 'full'
    return hconfig[0]

def get_conf_param(hconfig, key, defval = None): # 키에 대한 값을 찾아 반환 ㅊ
    if not isinstance(hconfig, list): return defval
    if len(hconfig) <= 1: return defval
    if not key in hconfig[1]: return defval
    return hconfig[1][key]

def get_conf_param_2d(hconfig, key, defval = None): # 위의 get_conf_param의 2
    if len(hconfig) <= 1: return defval
    if not key in hconfig[1]: return defval
    val = hconfig[1][key]
    if isinstance(val, list): return val
    return [val, val]
```

: 은닉계층 구성 정보를 위한 것. get_conf_param을 통해 key에 대한 값을 반환. 예외 처리 되었거나, 없는 경우에는 defval을 반환. get_conf_param_2d는 get_conf_param의 2차원 버전.

```
def cnn_basic_forward_full_layer(self, x, hconfig, pm):
    if pm is None: return x, None

    x_org_shape = x.shape

# 차원 축소. 4차원인 경우
# mini-batch 크기에 해당하는 첫번째 차원을 제외하는 다른 차원을 하나로 축소
    if len(x.shape) != 2:
        mb_size = x.shape[0]
        x = x.reshape([mb_size, -1])

    affine = np.matmul(x, pm['w']) + pm['b']
    y = self.activate(affine, hconfig) # activation function 다양하게 적용 가능

    return y, [x, y, x_org_shape]

CnnBasicModel.forward_full_layer = cnn_basic_forward_full_layer
```

: 차원을 검사하여 2차원이 아닌 경우 -> 4차원. mini-batch 크기에 해당하는 첫번째 차원을 제외한 다른 차원을 하나로 축소(연산을 위한 작업), activation function을 다양하게 적용 가능케 함.

```
def cnn_basic_backprop_full_layer(self, G_y, hconfig, pm, aux):
    if pm is None: return G_y

    x, y, x_org_shape = aux

    G_affine = self.activate_derv(G_y, y, hconfig)

    g_affine_weight = x.transpose()
    g_affine_input = pm['w'].transpose()

    G_weight = np.matmul(g_affine_weight, G_affine)
    G_bias = np.sum(G_affine, axis = 0)
    G_input = np.matmul(G_affine, g_affine_input)

    self.update_param(pm, 'w', G_weight)
    self.update_param(pm, 'b', G_bias)

    return G_input.reshape(x_org_shape) # shape 복원.

CnnBasicModel.backprop_full_layer = cnn_basic_backprop_full_layer
```

: 기존의 backpropagation과 유사. 연산을 위해 차원 축소 및 기존의 차원을 반환하는 것만 차이.

```
def cnn_basic_activate(self, affine, hconfig): # 여러 비선형 활성화 함수 적용
    if hconfig is None: return affine

    func = get_conf_param(hconfig, 'actfunc', 'relu')

    if func == 'none':      return affine
    elif func == 'relu':    return relu(affine)
    elif func == 'sigmoid': return sigmoid(affine)
    elif func == 'tanh':    return tanh(affine)
    else:                   assert 0

def cnn_basic_activate_derv(self, G_y, y, hconfig):
    if hconfig is None: return G_y

    func = get_conf_param(hconfig, 'actfunc', 'relu')

    if func == 'none':      return G_y
    elif func == 'relu':    return relu_derv(y) * G_y
    elif func == 'sigmoid': return sigmoid_derv(y) * G_y
    elif func == 'tanh':    return tanh_derv(y) * G_y
    else:                   assert 0

CnnBasicModel.activate = cnn_basic_activate
CnnBasicModel.activate_derv = cnn_basic_activate_derv
```

: 여러 비선형 함수 적용 가능

```python
# 이해하기는 쉽지만 계산 실행 시간이 지나치게 오래 걸린다
# 1. 3차원 내적 연산을 1차원 내적 연산으로 하는 방법.
# 2. 한 번의 행렬 곱셈으로 전체 출력 일괄 생성 등의 방법으로 성능 향상 꾀할 수
def forward_conv_layer_adhoc(self, x, hconfig, pm):
    mb_size, xh, xw, xchn = x.shape
    kh, kw, _, ychn = pm['k'].shape

    conv = np.zeros((mb_size, xh, xw, ychn))

    for n in range(mb_size):
        for r in range(xh):
            for c in range(xw):
                for ym in range(ychn):
                    for i in range(kh):
                        for j in range(kw):
                            rx = r + i - (kh-1) // 2
                            cx = c + j - (kw-1) // 2
                            if rx < 0 or rx >= xh: continue
                            if cx < 0 or cx >= xw: continue
                            for xm in range(xchn):
                                kval = pm['k'][i][j][xm][ym]
                                ival = x[n][rx][cx][xm]
                                conv[n][r][c][ym] += kval * ival

    y = self.activate(conv + pm['b'], hconfig)

    return y, [x, y]
```

: 이런 식으로 하면 for문이 무려 7중으로 된다. 이해하기는 쉽지만 계산 실행시간이 지나치게 오래 걸리는 치명적인 단점이 있다.

다른 방법에는

1. 3차원 내적 연산을 1차원 내적 연산으로 하는 방법(약간의 성능 향상)
2. 한 번의 행렬 곱셈으로 전체 출력 일괄 생성 등의 방법으로 성능 향상 꾀할 수 있다. (수천 배 이상 빨라지기도 한다.)

```
def forward_conv_layer_better(self, x, hconfig, pm):
    mb_size, xh, xw, xchn = x.shape
    kh, kw, _, ychn = pm['k'].shape

    conv = np.zeros((mb_size, xh, xw, ychn))

    bh, bw = (kh-1)//2, (kw-1)//2
    eh, ew = xh + kh - 1, xw + kw - 1

    x_ext = np.zeros((mb_size, eh, ew, xchn))
    x_ext[:, bh:bh + xh, bw:bw + xw, :] = x

    k_flat = pm['k'].transpose([3, 0, 1, 2]).reshape([ychn, -1])

    for n in range(mb_size):
        for r in range(xh):
            for c in range(xw):
                for ym in range(ychn):
                    xe_flat = x_ext[n, r:r + kh, c:c + kw, :].flatten()
                    conv[n, r, c, ym] = (xe_flat*k_flat[ym]).sum()

    y = self.activate(conv + pm['b'], hconfig)

    return y, [x, y]
```

: 1번 방법을 적용하였다. forward_conv_layer_adhoc() 함수에 비해 수십 배 빠른 실행 속도를 보이기는 하지만 여전히 실행 효율이 충분치 않다.

```
def cnn_basic_forward_conv_layer(self, x, hconfig, pm):
    mb_size, xh, xw, xchn = x.shape
    kh, kw, _, ychn = pm['k'].shape

    # 차원 축소를 통한 matmul 연산
    x_flat = get_ext_regions_for_conv(x, kh, kw)
    k_flat = pm['k'].reshape([kh*kw*xchn, ychn])

    conv_flat = np.matmul(x_flat, k_flat)
    conv = conv_flat.reshape([mb_size, xh, xw, ychn])

    y = self.activate(conv + pm['b'], hconfig)

    if self.need_maps: self.maps.append(y)

    return y, [x_flat, k_flat, x, y]

CnnBasicModel.forward_conv_layer = cnn_basic_forward_conv_layer
```

: 2번 방법을 적용하여 텐서 연산을 수행하였다. 반복문 수행 필요 없이 차원을 축소시켜 numpy의 matmul을 사용하는 행렬 곱셈 한번으로 연산을 끝낼 수 있다.

```
def cnn_basic_backprop_conv_layer(self, G_y, hconfig, pm, aux):
    x_flat, k_flat, x, y = aux

    kh, kw, xchn, ychn = pm['k'].shape
    mb_size, xh, xw, _ = G_y.shape

    G_conv = self.activate_derv(G_y, y, hconfig)

    G_conv_flat = G_conv.reshape(mb_size*xh*xw, ychn)

    g_conv_k_flat = x_flat.transpose()
    g_conv_x_flat = k_flat.transpose()

    G_k_flat = np.matmul(g_conv_k_flat, G_conv_flat)
    G_x_flat = np.matmul(G_conv_flat, g_conv_x_flat)
    G_bias = np.sum(G_conv_flat, axis = 0)

    G_kernel = G_k_flat.reshape([kh, kw, xchn, ychn])
    G_input = undo_ext_regions_for_conv(G_x_flat, x, kh, kw)

    # update parameter
    self.update_param(pm, 'k', G_kernel)
    self.update_param(pm, 'b', G_bias)

    return G_input

CnnBasicModel.backprop_conv_layer = cnn_basic_backprop_conv_layer
```

```
def get_ext_regions_for_conv(x, kh, kw):
    mb_size, xh, xw, xchn = x.shape

    regs = get_ext_regions(x, kh, kw, 0)
    regs = regs.transpose([2, 0, 1, 3, 4, 5])

    return regs.reshape([mb_size*xh*xw, kh*kw*xchn])

# 차원 축소
def get_ext_regions(x, kh, kw, fill):
    mb_size, xh, xw, xchn = x.shape

    eh, ew = xh + kh - 1, xw + kw - 1
    bh, bw = (kh-1)//2, (kw-1)//2

    x_ext = np.zeros((mb_size, eh, ew, xchn), dtype = 'float32') + fill
    x_ext[:, bh:bh + xh, bw:bw + xw, :] = x

    regs = np.zeros((xh, xw, mb_size*kh*kw*xchn), dtype = 'float32')

    for r in range(xh):
        for c in range(xw):
            regs[r, c, :] = x_ext[:, r:r + kh, c:c + kw, :].flatten()

    return regs.reshape([xh, xw, mb_size, kh, kw, xchn]) # 6차원의 텐서 형태로
```

: 차원 축소

```
def undo_ext_regions_for_conv(regs, x, kh, kw):
    mb_size, xh, xw, xchn = x.shape

    regs = regs.reshape([mb_size, xh, xw, kh, kw, xchn])
    regs = regs.transpose([1, 2, 0, 3, 4, 5])

    return undo_ext_regions(regs, kh, kw)

def undo_ext_regions(regs, kh, kw):
    xh, xw, mb_size, kh, kw, xchn = regs.shape

    eh, ew = xh + kh - 1, xw + kw - 1
    bh, bw = (kh-1)//2, (kw-1)//2

    gx_ext = np.zeros([mb_size, eh, ew, xchn], dtype = 'float32')

    for r in range(xh):
        for c in range(xw):
            gx_ext[:, r:r + kh, c:c + kw, :] += regs[r, c]

    return gx_ext[:, bh:bh + xh, bw:bw + xw, :] # 4차원
```

: 차원 축소를 원래대로 되돌리는 메소드

```
def cnn_basic_forward_avg_layer(self, x, hconfig, pm):
    mb_size, xh, xw, chn = x.shape
    sh, sw = get_conf_param_2d(hconfig, 'stride')
    yh, yw = xh // sh, xw // sw

    x1 = x.reshape([mb_size, yh, sh, yw, sw, chn])
    x2 = x1.transpose(0, 1, 3, 5, 2, 4) # 재배열
    x3 = x2.reshape([-1, sh*sw])

    y_flat = np.average(x3, 1)
    y = y_flat.reshape([mb_size, yh, yw, chn])

    if self.need_maps: self.maps.append(y) # 중간 결과에 대한 데이터 수집

    return y, None

def cnn_basic_backprop_avg_layer(self, G_y, hconfig, pm, aux):
    mb_size, yh, yw, chn = G_y.shape
    sh, sw = get_conf_param_2d(hconfig, 'stride')
    xh, xw = yh * sh, yw * sw

    gy_flat = G_y.flatten() / (sh * sw) # 차원 축소

    gx1 = np.zeros([mb_size*yh*yw*chn, sh*sw], dtype = 'float32')
    for i in range(sh*sw):
        gx1[:, i] = gy_flat # 모든 행 복사
    gx2 = gx1.reshape([mb_size, yh, yw, chn, sh, sw])
    gx3 = gx2.transpose([0, 1, 4, 2, 5, 3]) # 재배열

    G_input = gx3.reshape([mb_size, xh, xw, chn])

    return G_input

CnnBasicModel.forward_avg_layer = cnn_basic_forward_avg_layer
CnnBasicModel.backprop_avg_layer = cnn_basic_backprop_avg_layer
```

: Avg Pooling 에 대한 순전파 및 역전파 처리

```
def cnn_basic_forward_max_layer(self, x, hconfig, pm):
    mb_size, xh, xw, chn = x.shape
    sh, sw = get_conf_param_2d(hconfig, 'stride')
    yh, yw = xh // sh, xw // sw

    x1 = x.reshape([mb_size, yh, sh, yw, sw, chn])
    x2 = x1.transpose(0, 1, 3, 5, 2, 4)
    x3 = x2.reshape([-1, sh*sw])

    idxs = np.argmax(x3, axis = 1)
    y_flat = x3[np.arange(mb_size*yh*yw*chn), idxs]
    y = y_flat.reshape([mb_size, yh, yw, chn])

    if self.need_maps: self.maps.append(y)

    return y, idxs

def cnn_basic_backprop_max_layer(self, G_y, hconfig, pm, aux):
    idxs = aux

    mb_size, yh, yw, chn = G_y.shape
    sh, sw = get_conf_param_2d(hconfig, 'stride')
    xh, xw = yh * sh, yw * sw

    gy_flat = G_y.flatten()

    gx1 = np.zeros([mb_size*yh*yw*chn, sh*sw], dtype = 'float32')
    gx1[np.arange(mb_size*yh*yw*chn), idxs] = gy_flat[:]
    gx2 = gx1.reshape([mb_size, yh, yw, chn, sh, sw])
    gx3 = gx2.transpose([0, 1, 4, 2, 5, 3])

    G_input = gx3.reshape([mb_size, xh, xw, chn])

    return G_input

CnnBasicModel.forward_max_layer = cnn_basic_forward_max_layer
CnnBasicModel.backprop_max_layer = cnn_basic_backprop_max_layer
```

: Max Pooling 에 대한 순전파 및 역전파 처리

```
def cnn_basic_visualize(self, num):
    print('Model {} Visualization'.format(self.name))

    self.need_maps = self.show_maps
    self.maps = []

    deX, deY = self.dataset.get_visualize_data(num)
    est = self.get_estimate(deX)

    if self.show_maps:
        for kernel in self.kernels:
            kh, kw, xchn, ychn = kernel.shape
            grids = kernel.reshape([kh, kw, -1]).transpose(2, 0, 1)
            draw_images_horz(grids[0:5, :, :])

        for pmap in self.maps:
            draw_images_horz(pmap[:, :, :, 0])

    self.dataset.visualize(deX, est, deY)

    self.need_maps = False
    self.maps = None

CnnBasicModel.visualize = cnn_basic_visualize
```

: 시각화 과정

실행 결과

[Test1] default 설정, epoch 10으로 수행한 다층 퍼셉트론

```
fm1 = CnnBasicModel('flowers_model_1', fd, [30, 10])
fm1.exec_all(epoch_count = 10, report = 2)
```

```
Model flowers_model_1 train started:
    Epoch 2: cost=1.568, accuracy=0.311/0.360 (48/48 secs)
    Epoch 4: cost=1.424, accuracy=0.372/0.380 (48/96 secs)
    Epoch 6: cost=1.357, accuracy=0.419/0.380 (49/145 secs)
    Epoch 8: cost=1.314, accuracy=0.431/0.330 (47/192 secs)
    Epoch 10: cost=1.298, accuracy=0.424/0.410 (48/240 secs)
Model flowers_model_1 train ended in 240 secs:
Model flowers_model_1 test report: accuracy = 0.419, (0 secs)

Model flowers_model_1 Visualization
```

```
추정확률분포 [32,42, 0, 9, 7, 9] => 추정 dandelion : 정답 dandelion => O
추정확률분포 [ 9,25, 0,10,37,19] => 추정 sunflower : 정답 rose => X
추정확률분포 [16,15, 0,30, 8,30] => 추정 tulip : 정답 tulip => O
```

: 정확도 0.419로 2/3의 결과

[Test2] default 설정, epoch 10.

Fully Connected 를 이용하여 width 30, 10으로 순차적 수행, Adam 알고리즘 적용 X

```
fm2 = CnnBasicModel('flowers_model_2', fd,
                    [['full', {'width':30}],
                     ['full', {'width':10}]])
fm2.use_adam = False
fm2.exec_all(epoch_count = 10, report = 2)
```

```
Model flowers_model_2 train started:
    Epoch 2: cost=1.459, accuracy=0.350/0.250 (11/11 secs)
    Epoch 4: cost=1.357, accuracy=0.403/0.410 (13/24 secs)
    Epoch 6: cost=1.313, accuracy=0.426/0.430 (12/36 secs)
    Epoch 8: cost=1.277, accuracy=0.439/0.390 (12/48 secs)
    Epoch 10: cost=1.231, accuracy=0.463/0.450 (12/60 secs)
Model flowers_model_2 train ended in 60 secs:
Model flowers_model_2 test report: accuracy = 0.364, (0 secs)

Model flowers_model_2 Visualization
```

```
추정확률분포 [29,63, 0, 2, 1, 6] => 추정 dandelion : 정답 daisy => X
추정확률분포 [30,26, 0,13,11,20] => 추정 daisy : 정답 daisy => O
추정확률분포 [ 0, 0, 0,85, 0,15] => 추정 rose : 정답 rose => O
```

: 정확도 0.364로 2/3의 결과 앞의 모델보다 정확성이 떨어지는 것을 확인할 수 있었다.

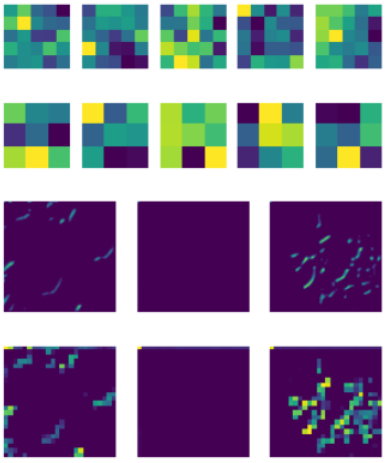
[Test3] default 설정, epoch 10.

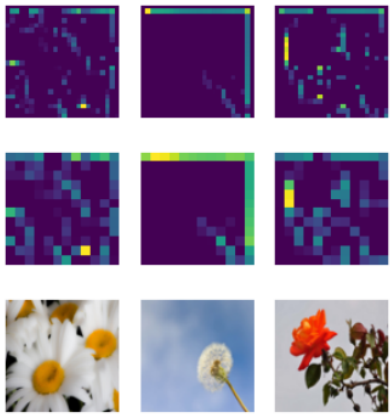
conv -> max -> conv -> avg 순차적으로 수행하였다. (채널 6 -> 12)

```
fm3 = CnnBasicModel('flowers_model_3', fd,
              [['conv', {'ksize':5, 'chn':6}],
               ['max', {'stride':4}],
               ['conv', {'ksize':3, 'chn':12}],
               ['avg', {'stride':2}]],
              True)
fm3.exec_all(epoch_count = 10, report = 2)
```

```
Model flowers_model_3 train started:
    Epoch 2: cost=1.129, accuracy=0.546/0.540 (383/383 secs)
    Epoch 4: cost=0.899, accuracy=0.664/0.530 (373/756 secs)
    Epoch 6: cost=0.761, accuracy=0.714/0.660 (370/1126 secs)
    Epoch 8: cost=0.624, accuracy=0.763/0.560 (340/1466 secs)
    Epoch 10: cost=0.489, accuracy=0.821/0.540 (364/1830 secs)
Model flowers_model_3 train ended in 1830 secs:
Model flowers_model_3 test report: accuracy = 0.516, (8 secs)

Model flowers_model_3 Visualization
```





```
추정확률분포 [ 1, 1, 0.23, 0.75] => 추정 tulip : 정답 daisy => X
추정확률분포 [20.55, 0, 5.13, 7] => 추정 dandelion : 정답 dandelion => O
추정확률분포 [ 0, 0, 0.60, 0.40] => 추정 rose : 정답 rose => O
```

: 이전의 모델들보다 성능이 향상된 것을 확인할 수 있다.

0.516의 정확도, 2/3

[Test3] default 설정, epoch 10.

conv -> max -> conv -> max -> conv -> avg 순차적으로 수행하였다.(채널 6 -> 12 -> 24, stride 3)

```
fm4 = CnnBasicModel('flowers_model_4', fd,
                [['conv', {'ksize':3, 'chn':6}],
                 ['max', {'stride':2}],
                 ['conv', {'ksize':3, 'chn':12}],
                 ['max', {'stride':2}],
                 ['conv', {'ksize':3, 'chn':24}],
                 ['avg', {'stride':3}]])
fm4.exec_all(epoch_count = 10, report = 2)
```

```
Model flowers_model_4 train started:
    Epoch 2: cost=1.123, accuracy=0.555/0.550 (359/359 secs)
    Epoch 4: cost=0.908, accuracy=0.661/0.650 (358/717 secs)
    Epoch 6: cost=0.758, accuracy=0.714/0.600 (370/1087 secs)
    Epoch 8: cost=0.646, accuracy=0.761/0.630 (378/1465 secs)
    Epoch 10: cost=0.544, accuracy=0.797/0.640 (417/1882 secs)
Model flowers_model_4 train ended in 1882 secs:
Model flowers_model_4 test report: accuracy = 0.627, (8 secs)

Model flowers_model_4 Visualization
```

```
추정확률분포 [ 0, 0, 0, 5, 0,95] => 추정 tulip : 정답 daisy => X
추정확률분포 [44, 6, 0, 5, 2,43] => 추정 daisy : 정답 rose => X
추정확률분포 [25, 9, 0, 1, 1,65] => 추정 tulip : 정답 tulip => O
```

: 0.627의 정확도, 1/3 conv -> pooling의 여러 단계를 거쳐서 이전보다 성능이 향상된 것을 확인할 수 있었다.

결론

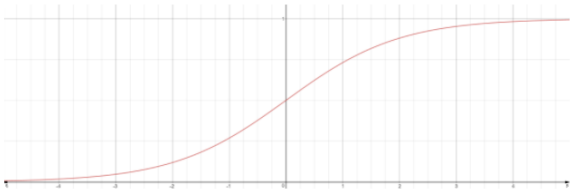
- 차원 축소를 통해서 연산을 간단하게 수행할 수 있도록 하였으며 효율적인 수행이 가능하였다.
- CNN 관련 라이브러리를 사용할 때는 이렇게 Avg Pooling, Max Pooling 등 각각에 forward progration, backpropagation이 별도로 존재하는 지를 모르고 그냥 이용하였는데 내부적으로 어떻게 동작하는지를 알 수 있어 도움이 되었다.
- 원래 합성곱 연산을 미니배치, 텐서를 통해서 한번에 수행한다는 것을 알고 있었으나 다른 방법에 대해서는 알지 못했는데 왜 행렬곱으로 한번에 이런 방식으로 처리하는지에 대해서 다른 방법과 비교하여 알 수 있었다.
- getattr() 함수를 통해서 특정 이름의 메소드를 탐색할 수 있다는 것을 알게 되었고, 메소드 이름을 통해 후킹 기법을 사용할 수 있다는 것을 알게 되었다.
- 라이브러리를 이용할 때는 알 수 없었던 많은 것을 알게되어 매우 유익하였다.
- Conv -> Pool -> Conv -> Pool ··· FC -> Softmax 의 과정에 대해서 이해할 수 있게 되었다.

1. Adam이외의 최적화 함수에는 Sigmoid, tanh, ReLU, Leaky ReLU, PReLu, ELU, Maxout 등이 있다.

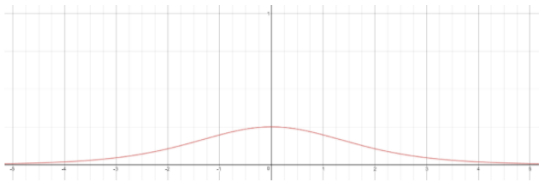
Sigmoid (logistic function)

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

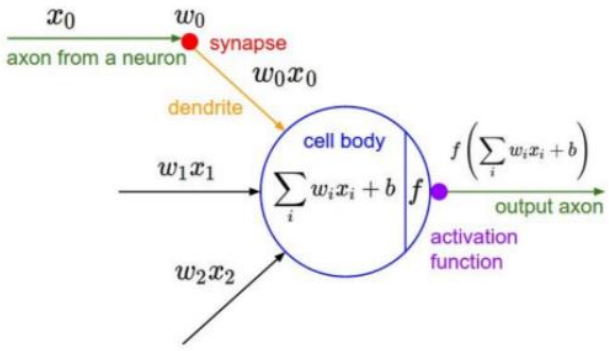


[Sigmoid]



[Sigmoid 1차 미분]

- S자 커브 형태로 자연스럽게 활성화 가능함.
- 입력 값에 따라 gradient 값(sigmoid 1차 미분값)이 지나치게 작아질 수 있다(학습이 잘 안됨)
- Exponential 연산 자체가 다소 무겁다. (학습이 느려짐)



$x0, x1, x2$는 모두 0 이상의 값을 갖는데, 직전 층에서 sigmoid로 인해서 양수([0,1])로 활성화된 출력이 입력으로 주어지기 때문이다.

$$\frac{\partial L}{\partial w_i} = x_i \times \delta$$

: Loss(L)를 weight(w)에 따라 미분한 것.

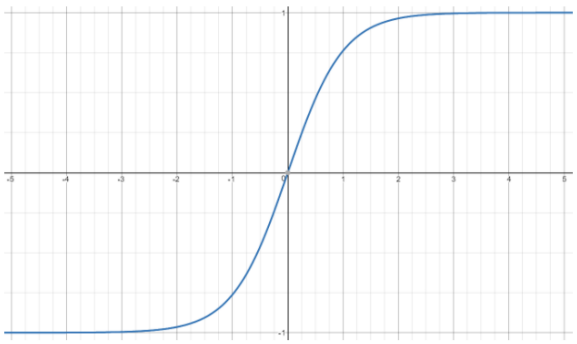
δ가 양수라면 Loss에 대한 $w0, w1, w2$ 각각의 gradient가 모두 양수 혹은 모두 음수가 된다. 따라서 데이터 x와 파라미터 w를 2차원 벡터로 가정한다면 1사분면과 3사분면에만 속하고, 2사분면과 4사분면에는 속할 수 없다. 결과적으로 허용되는 방향에 제약이 가해져(1사분면과 3사분면에만 가능) 학습 속도가 늦거나 수렴이 어렵게 된다. 이 문제는 하이퍼볼릭탄젠트(tanh)를 이용하여 해결할 수 있다.

tanh (하이퍼볼릭탄젠트)

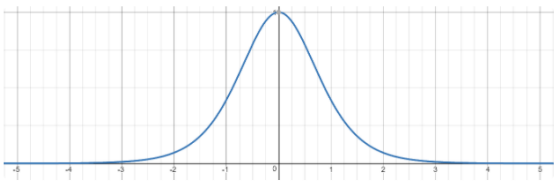
하이퍼볼릭탄젠트는 **sigmoid 함수의 크기와 위치를 조절(rescale and shift)한 함수이다.**

sigmoid 함수와의 관계식과 미분식이다.

$$\begin{aligned} tanh(x) &= 2\sigma(2x) - 1 \\ &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ tanh'(x) &= 1 - tanh^2(x) \end{aligned}$$



[tanh]



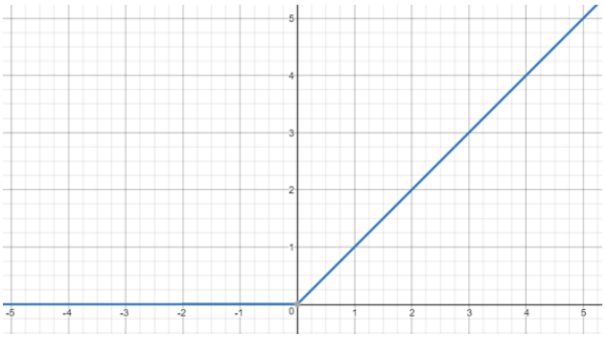
[tanh 1차 미분]

- sigmoid 함수[0, 1]와 달리 tanh는 -1과 1사이의 값을 가진다([-1, 1]

- 그래프의 모양이 sigmoid 함수와 달리 0을 기준으로 대칭인 점을 확인할 수 있다. -> sigmoid를 활성화 함수로 사용할 때보다 학습 수렴 속도가 빠르다.
- 입력 값에 따라 gradient 값(tanh1차 미분값)이 지나치게 작아질 수 있다(학습이 잘 안됨)

ReLU(Rectified Linear Unit)

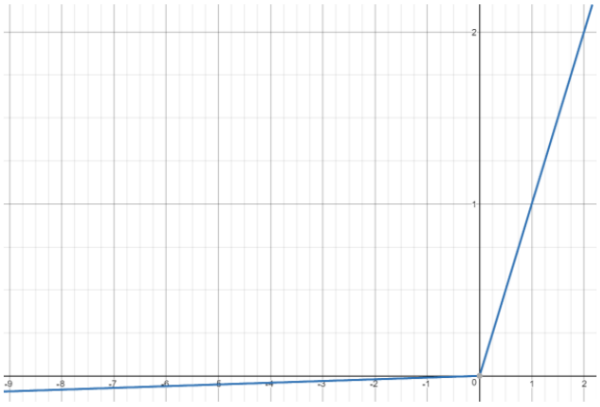
$$f(x) = max(0, x)$$



- 음수일 때 무조건 gradient 가 0이 되는 단점이 있다. -> 이를 해결하기 위해서 Leaky ReLu를 사용한다.
- x = 0에서 미분이 불가능하다. 이는 보통 x = 0에서 미분 값을 0으로 만들어주는 방식을 통해 따로 처리한다. -> x <= 0에서 미분 값이 0이 된다.
- 앞서 소개된 활성화 함수에 비해 학습 속도가 훨씬 빠르다

Leaky ReLU

$$f(x) = max(0.01x, x)$$



- **x 가 음수일 때 gradient가 0.01**이라는 점을 제외하고는 ReLU와 같다.

PReLU

$$f(x) = max(\alpha x, x)$$

- Leaky ReLU와 거의 유사하지만 **새로운 파라미터** α를 추가하여 x < 0에서 기울기를 학습할 수 있게 하였다.

ELU(Exponential Linear Unit)

$$f(x) = x \quad if \quad x > 0$$
$$f(x) = \alpha(e^x - 1) \quad if \quad x \leq 0$$

- ReLU의 특성을 공유하고, ***Dying ReLU 문제가 발생하지 않는다.**
- 출력값이 거의 zero-centered 에 가깝다.
- 일반적인 ReLU와 달리 exp 함수를 계산하는 비용이 발생한다.

Maxout 함수

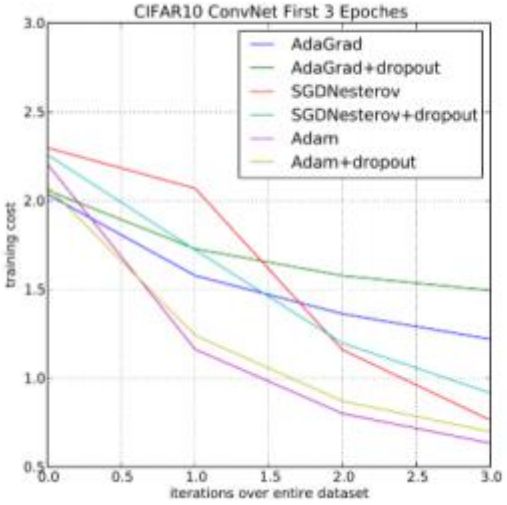
$$f(x) = max(w_1^T x + b_1, w_2^T x + b_2)$$

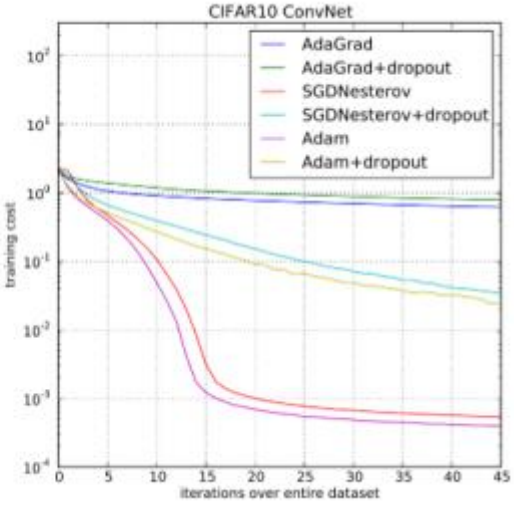
- ReLU의 특성을 공유하고, **Dying ReLU 문제가 발생하지 않는다.**
- 계산량이 복잡하다

---

*Dying ReLu : Forward propagation 과정에서 어떤 뉴런에서든 음수 값이 들어오게 되면 BackPropagation 시에 0이라는 real value 가 가중치에 곱해지면서 해당 노드가 통째로 죽어버리는 현상

무조건 좋지 않은 현상은 아닌 것으로 보인다. 오히려 이 점이 ReLU가 가지는 큰 강점으로 작용한다는 것.(Spacrse Repersentation을 부여함으로써) 거시적 관점에서 일종의 일반화 방식으로 dropout과 유사하게 overfitting 을 줄이는 역할을 한다. **단, 신경망이 깊고, 넓으며, 충분한 Epoch가 주어진 상황에서만 일반화에 도움이 된다.**





CNN에서 Adam을 적용하였을 때 다른 activation function을 적용하는 것보다 높은 정확도를 보이는 것을 확인할 수 있다.

2. CNN에서 이미지의 부분들을 잘라서 학습하는데 어떤 부분인지, 왜 그렇게 되는지를 설명하시오

: cnn_basic_forward_conv_layer에서 픽셀 각각에 대해 내적 연산을 수행하는 것을 알 수 있다. mini-batch를 이용하여 텐서 연산을 수행하여(numpy.matmul 행렬곱 이용) 성능 향상을 위한 것이다.

3. CNN은 이미지 외에 자연어 처리에도 적용할 수 있는데 어떤 함수를 수정하여 적용해야 가능한지?

: 대부분의 NLP 문제의 입력은 행렬로 표현되는 문장이나 문서이다. 10개의 단어로 이루어진 문장이 있으면 ont-hot vector와 같이 100차원 imbedding 을 사용하여 10 * 100의 행렬을 입력으로 하여 이를 다룰 수 있다. 처음에 이미지를 가져오는 부분을 문자를 가져온다고 가정하면 그 함수를 변경시켜 rearrange를 적용하면 될 것이다.