실전 기계 학습 중간

SVM : 선형분류기

- 일반화 능력을 높이기 위해 여백을 최대화한다.
- C를 크게 하면 정확도는 높지만, 일반화 능력이 떨어지며, C를 작게 하면 정확도는 떨어지지만 일반화 능력은 높아진다.

혼동 행렬 : 부류별로 옳은 분류와 틀린 분류의 개수를 기록한 행렬

```
conf=np.zeros((10,10))
for i in range(len(res)):
    conf[res[i]][y_test[i]]+=1
print(conf)
```

- 참긍정(TP) / 거짓부정(FN) / 거짓긍정(FP) / 참부정(TN)

$$\text{정확률}=\frac{\text{맞힌 샘플 수}}{\text{전체 샘플 수}}=\frac{\text{대각선 샘플 수}}{\text{전체 샘플 수}} \quad \text{특이도}=\frac{TN}{TN+FP},\ \text{민감도}=\frac{TP}{TP+FN} \quad \text{정밀도}=\frac{TP}{TP+FP},\ \text{재현율}=\frac{TP}{TP+FN}$$

퍼셉트론

입력층 : d + 1 개(d개의 특징 벡터와 상수) 출력층 : 1개

$$o=\tau(s)=\tau\left(\sum_{i=1}^{d} w_i x_i + w_0\right)$$

계단 함수를 활성 함수로 사용하여 이진 분류

$$\text{이때 } \tau(s)=\begin{cases}+1, s>0\\ -1, s\le 0\end{cases} \qquad o=\tau\left(\mathbf{w}\mathbf{x}^{\mathrm{T}}\right)$$

퍼셉트론은 선형 분류로 국한된다.

손실함수

w에 따라 손실 함수 J(w)를 계산하고, 미분을 사용하여 손실 함수를 낮추는 Δw를 계산한다

➔ 미분을 통해 기울기를 계산하여 w값 업데이트 방향을 알아낸다.

➔ 미분을 통해 알아낸 값을 항상 최소점은 아니지만 극소점이 된다.

$$J(\mathbf{w})=\sum_{\mathbf{x}\in I} -y\left(\mathbf{w}\mathbf{x}^{\mathrm{T}}\right)$$

I는 틀린 샘플의 집합, $-y(wx^T)$는 항상 양수

틀린 값이 많을수록 손실함수는 항상 증가한다.

Epoch : 훈련 집합에 있는 샘플 전체를 한 번 처리하는 일

Optimizer

gradient descent 경사 하강법

w를 미분해서 미분 값 부호의 반대로 업데이트한다

$$w_1 = w_1 + \rho\left(-\frac{\partial J}{\partial w_1}\right)$$

업데이트될 때마다 J(w)는 감소한다. p는 학습률

사용 이유

- 데이터에 노이즈가 개입하기 때문에
- 매개 변수가 방대해서 과잉 적합될 수 있기 때문에

Stochastic gradient descent 스토캐스틱 경사 하강법

배치모드 : 틀린 샘플을 모은 다음 한꺼번에 매개 변수를 갱신한다.

- 전체 데이터의 평균을 내기 때문에 안정적이다. 계산량이 크다
- **미니 배치** : 일정한 크기의 부분 집합을 모아 처리한다. 배치/패턴의 중간 형태

패턴 모드 : 패턴 별로 매개 변수를 갱신한다. (세대를 시작할 때 샘플을 뒤섞어 랜덤 샘플링 효과 제공)

- 안정적인 데이터 결과를 내지 못한다. 계산량이 작다

미니 배치를 사용하여 부분 집합으로 나눌 때 랜덤 샘플링을 적용하기 때문에 SGD라고 부른다.

- 극소값이 아닌 flat한 부분을 만나면 gradient가 업데이트가 잘 되지 않을 수 있다

➔ 2차 미분 값을 추가로 사용하거나 학습률을 조정한다.

➔ 각각의 데이터마다 적절한 학습률이 다르다 -> 적응적 학습률 : ADAM

torch.optim.SGD

- data 의 실제값과 model 을 통한 예측값을 비교하여 loss 를 도출한 후, loss 를 줄이기 위하여 loss 의 미분을 통하여 loss 가 최소가 되는 지점을 찾는다.
- 전체 데이터의 일부인 batch 데이터를 이용한다.

torch.optim.Adagrad

- 가중치가 업데이트된 횟수에 따라 learning rate 를 다르게 업데이트 한다.
- 변동이 많이 있었던 가중치의 learning rate 는 감소시키고, 변동이 많이 없었던 가중치에 대해서는 learning rate 를 증가시켜서 학습한다.
- learning rate 를 줄이기 때문에 최적의 해에 가까워 지기전에 learning rate 가 0 에 가까워져 더이상 최적화가 이루어지지 않을 수 있다.

torch.optim.RMSprop

- adagrad 의 gradient 가 0 에 수렴하는 문제를 가지고 있기 때문에, adagrad 의 문제점을 지수이동평균을 적용하여 개선한 방식이다. 학습의 최소 steo 을 유지할 수 있다.

torch.optim.Adam

- momentum 이란 이전 학습 결과도 가중치 변화에 영향을 주게 함으로써 최소값이 아닌 극소점을 만났을 때 빠져나올 수 있도록 해준다.
- Adam 은 RMSprop 와 Momentum 을 조합한 방식

torch.optim.Adamax

- Adam 의 계산식에서 v_t 에 다른 norm 을 넣어 사용하는 방법

torch.optim.ASGD

- 기존의 SGD 방식에서 평균화 기법을 적용함으로써 더 높은 속도를 얻고자 고안된 방법

torch.optim.LBFGS

- LBFGS 는 Limited-memory quasi-Newton methods 의 한 예시로써, Hessian 행렬을 계산하거나 저장하기 위한 비용이 합리적이지 않을 경우 유용하게 사용된다. 이 방법은 밀도가 높은 의 Hessian 행렬을 저장하는 대신 차원의 벡터 몇 개만을 유지하여 Hessian 행렬을 추정(approximation)하는 방식이다.

torch.optim.Rprop

- back propagation 과 비슷하지만 속도가 더 빠른 방식
- lr 과 같은 하이퍼 파리미터가 필요하지 않으며, 각 가중치는 독립적으로 계산되며 최적화중 가중치의 부호가 바뀌었을 때는 -1<mu<0 의 값을, 부호가 같을 때는 0<mu<1 의 값을 곱해주면 최적화를 진행한다.

다층 퍼셉트론 (MLP)

- 퍼센트론을 병렬로 결합하면 공간의 차원 상승, Sigmoid 주로 사용

```
mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001,batch_
size=32,max_iter=300,solver='sgd',verbose=True)
```

딥러닝(은닉층에 ReLU, 출력층에 Softmax를 주로 사용한다)

- 은닉층을 추가하여 비선형 분류를 가능하게 한다.
- 원래의 특징 공간을 임시공간(은닉층)으로 변환
- 새로운 공간은 성능을 높여주지만, 너무 많아지면 과잉 적합 가능성이 높아진다.
- Batch size는 32, 64를 많이 사용(특정 수를 넘어야 batch가 모집단의 경향과 비슷해진다)
- NAS : 데이터에 대한 최적의 모델을 자동으로 찾아준다.

오류 역전파 학습 알고리즘

출력층에서 시작하여 역방향으로 오류를 전파하는 뜻

- 입력층에서 출력층까지의 계산을 한다(전방 계산)
- 레이블과 예측값과의 오차를 계산한다
- 출력층에서 오류층으로 오류를 전파한다(오류 역전파, Back Propagation)

PyTorch

torch.ones(n, m) (n x m의 1행렬)　　torch.manual_seed(2) = 임의의 랜덤 시드

torch.randn(n, m) (n x m의 임의의 정수 행렬)　torch.tensor([[1,2], [3,4]])

param.view = reshape 명령어　y.add_(x) : y += x　　param.numpy() = to_numpy

Autograd 사용하기 위해

- param.torch.ones([n, m], **requires_grad = True**) 같이 requires_grad = True가 필요하다
- t.backward()　: back-propagation 실행
- param.grad : 각 변수별 미분 값 반환
- param.grad.data.zero_()를 통해 gradient값을 초기화 한다.

torch.nn.Linear(n, m) : n차원의 데이터를 m차원의 데이터를 가중치를 모두 곱하여 선형 변환

```
class Model(nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
```

optimizer.step() : 각 데이터마다 forward, backward를 수행 후 가중치 업데이트

```
for x_val, y_val in zip(x_data, y_data):
    ...
    w.data = w.data - 0.01 * w.grad.data
```

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

학습률이 너무 크면 발산하며, 너무 작으면 Update가 되지 않는다.

Logistic Regression

- 독립 변수의 선형 결합을 이용하여 사건 발생 가능성을 예측, 2진 변수로 결과값을 반환한다

Loss function

$\frac{\partial loss}{\partial w} = w.grad$　　`w.data = w.data - 0.01 * w.grad.item()`

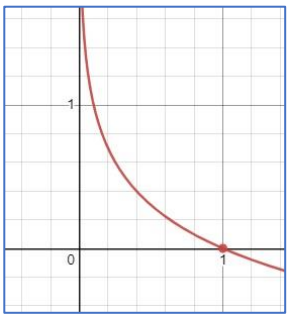
$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

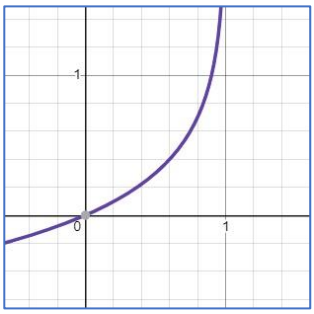
$$w = w - \alpha\frac{\partial loss}{\partial w}$$

$$\arg\min_{w} loss(w)$$

- **MSELoss**
- y를 원핫코드로 표현하고, 각각의 O_i와의 유클리드 거리를 구한다.
- 제곱을 하여 MSE를 구하여 제곱근을 제거한다.(미니 배치 단위로 처리)
-
```
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```
- **Cross Entropy Loss**
- $$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$
- criterion = nn.BCELoss(reduction = 'mean') : Binary Cross Entropy
- BCELoss는 SoftMax를 거친 후 Cross-Entropy Loss를 수행한다.



- true값이 1, 즉 y = 1 일 때 / loss가 1일 때 가장 작다
- y의 예측값을 강하게 1로 drive한다.



- true값이 0, 즉 y = 0 일 때 / loss가 0일 때 가장 작다
- y의 예측값을 강하게 0으로 drive한다.

활성함수

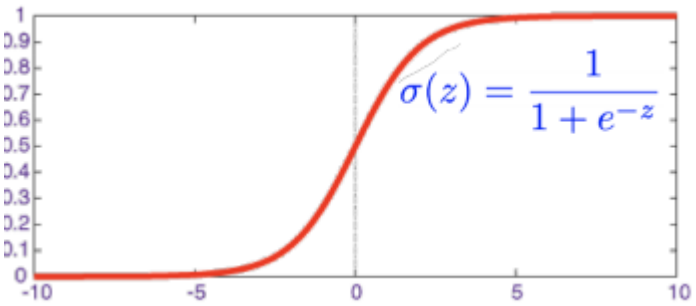
- 미분이 가능해야 한다. - Vanishing gradient문제를 최대한 억제해야 한다.
- Non-Linear이여야 한다.

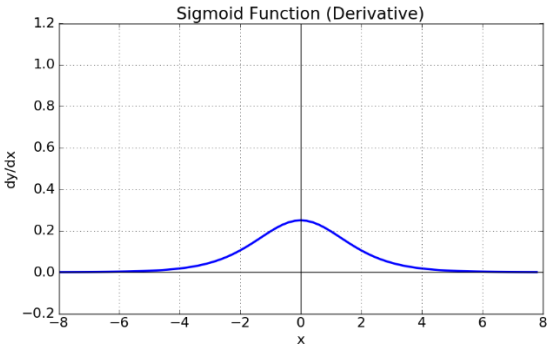
sigmoid

무한의 입력 범위를 유한 입력 범위로 줄여주면 입력이 클 때 1에 가까운 값을 반환한다.

모든 실수 입력 값을 0과 1사이의 미분 가능한 수로 변환한다.

음수 값을 0에 가깝게 표현하기 때문에 계층이 많아질수록 vanishing gradient 문제가 발생한다.





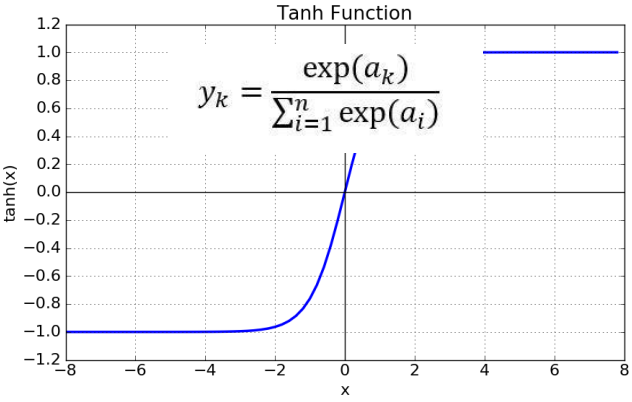
tanh

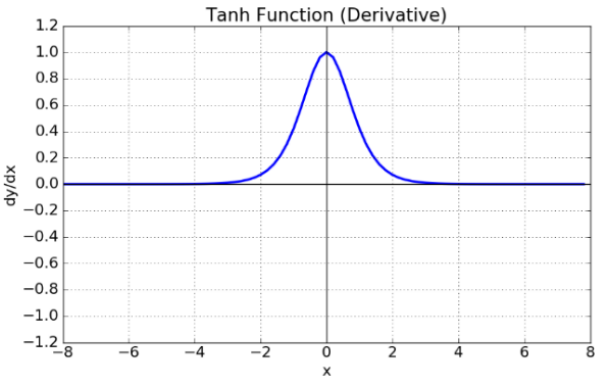
sigmoid 함수를 변형한 쌍곡선 함수

미분의 형태를 보면 sigmoid함수보다 4배 크기 때문에 non-zero centered 문제를 해결하였다.

vanishing gradient문제를 여전히 포함하고 있다.

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





SoftMax

- 큰 값은 더 크게, 작은 값은 더 작게 만든다 (출력층에서 많이 사용한다)
- 입력 값을 모두 [0,1]사이의 값으로 정규화하여 출력하며 출력값을 총합은 항상 1이다.

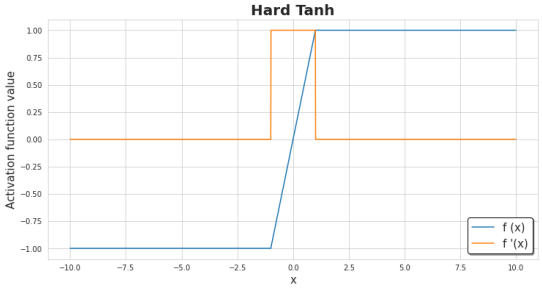
HardTanh

- Tanh 함수보다 각진 형태의 함수
- 입력값이 1 보다 크면 1 을 출력하고 입력값이 -1 보다 작으면 -1 을 출력하면 그 외에는 입력값을 출력한다.

$$f(x) = 1 \ \ if \ \ x > 1$$

$$f(x) = -1 \ \ if \ \ x < -1$$

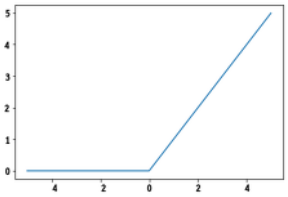
$$f(x) = x \ \ otherwise$$



-

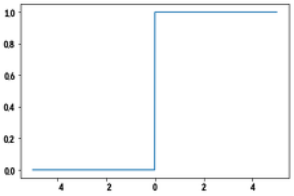
ReLU

- vanishing gradient 문제를 해결하기 위해 고안되었다.
- ReLU 의 도함수는 0 또는 1 의 값을 가지기 때문에 컴퓨팅 자원 면에서 효율적이며 최대값이 1 보다 큰 값도 가능하여 학습이 빠르다는 장점을 가지고 있다.
- 0 보다 작은 경우에는 0 을 반환하고 0 을 반환한 노드는 다시 값을 가지기 어렵다는 문제를 가지고 있다.
- 음수를 포기한 것은 양수가 더 중요하며 평균적으로 사용하면 0.5 정도의 gradient 를 얻음



$$ReLU(x) = \max(0,\ x)$$

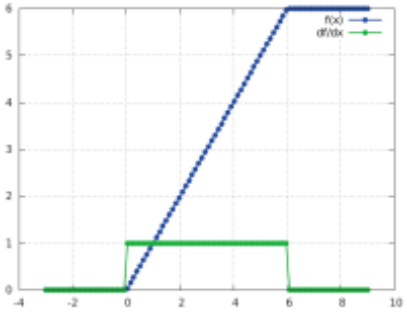
Rectified Linear Unit



$$R'(y) = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

ReLU6

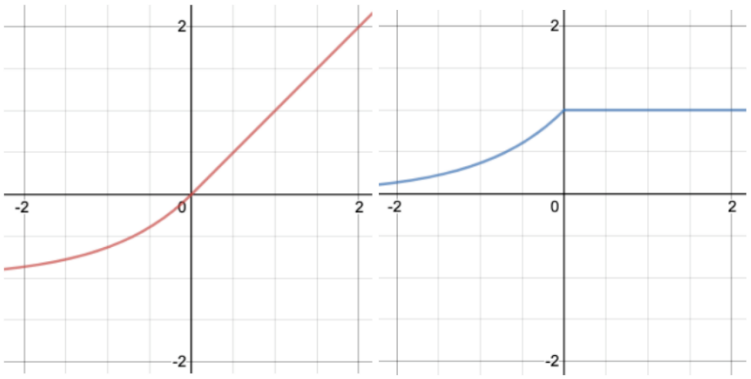
- ReLU 의 변형으로 6 의 상한값을 설정한다.
- 정확도 면에서 장점을 가지고 있으며, 상한값이 6 이기 때문에 3bit 만 사용하여 표현이 가능합니다.
- ReLU6 의 함수 및 도함수 형태



ELU & SELU

- ReLU와 비슷한 형태를 가지면 지수를 이용하여 음수일 경우 0이 아닌 완만한 곡선으로 표현합니다.
- 출력값이 0에 가까우며 exp함수의 비용이 별도로 발생합니다.
- 음수의 표현 a(e^x – 1)로 표현하며 a = 1일 때 ELU, 그 외의 값일 때 SELU라고 지칭합니다.

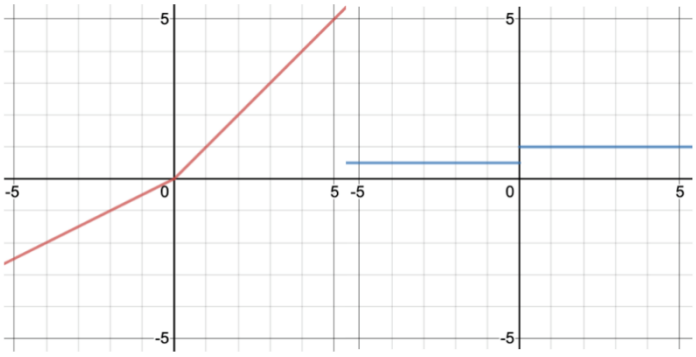
$$f(\alpha,x)=\begin{cases} x & (x>0) \\ \alpha(e^x-1) & (x\leq 0) \end{cases} \qquad f'(\alpha,x)=\begin{cases} 1 & (x>0) \\ f(\alpha,x)+\alpha & (x\leq 0) \end{cases}$$



LeakyReLU & PReLU

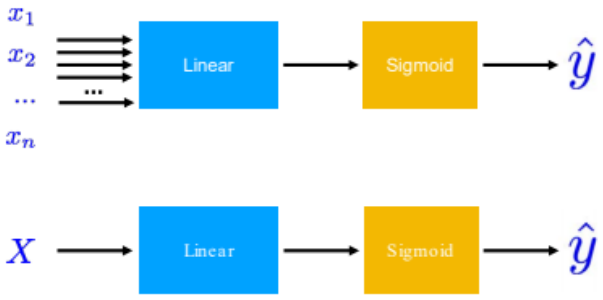
- LeakyReLU는 ReLU가 가지는 Dying ReLU현상을 해결하기 위해 고안되었습니다.
- LeakyReLU는 ReLU와 비슷한 형태이지만 음수의 표현에서 0이 아닌 ax형태로 표현함으로써 미분값이 0이 아닌 값인 작은 값으로 표현됩니다.
- LeakyReLU는 알파값으로 고정된 상수를 사용하며 PReLU는 알파값을 학습이 가능한 파라미터로 설정한 방법입니다.

$$f(\alpha,x)=\begin{cases}\alpha x & (x<0)\\ x & (x\geq 0)\end{cases} \qquad f'(x)=\begin{cases}\alpha & (x<0)\\ 1 & (x\geq 0)\end{cases}$$



Wide

- 한 층에서 입력과 출력이 굉장히 많은 경우



Deep

- 여러 개의 층을 쌓은 경우
- 활성화 함수를 사용하지 않는다면 하나의 층을 쌓은 것과 같은 동작이다.
- 활성화 함수를 사용하면 비선형 분류가 가능해진다.


```
sigmoid = torch.nn.Sigmoid()

l1 = torch.nn.Linear(2, 4)
l2 = torch.nn.Linear(4, 3)
l3 = torch.nn.Linear(3, 1)

out1    = sigmoid(l1(x_data))
out2    = sigmoid(l2(out1))
y_pred = sigmoid(l3(out2)
```

Vanishing Gradient Problem

Sigmoid의 경우

- 입력 값이 크거나 작은 경우 미분 값이 0에 가까워진다.
- 층이 많아질수록 기울기가 0에 가까워질 확률이 커지므로 가중치가 0에 가까워진다.

```
from torch import nn, optim, from_numpy
import numpy as np
from google.colab import drive

drive.mount('/content/gdrive')

xy = np.loadtxt('/content/gdrive/My Drive/Colab Notebooks/data/diabetes.csv.gz', delimiter=',', dtype=np.float32)
x_data = from_numpy(xy[:, 0:-1])
y_data = from_numpy(xy[:, [-1]])
print(f'X데's shape: {x_data.shape} | Y데's shape: {y_data.shape}')
```

```
class Model(nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn
        """
        super(Model, self).__init__()
        self.l1 = nn.Linear(8, 6)
        self.l2 = nn.Linear(6, 4)
        self.l3 = nn.Linear(4, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Vari
        a Variable of output data. We can use Mo
        well as arbitrary operators on Variables
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        return y_pred

model = Model()
```

```
criterion = nn.BCELoss(reduction='mean')
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

```
for epoch in range(100):
    # Forward pass: Compute predicted y by passing x to the mode
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(f'Epoch: {epoch + 1}/100 | Loss: {loss.item():.4f}')

    # Zero gradients, perform a backward pass, and update the we
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Data Loader

- 데이터를 불러오고 효과적으로 batch를 만들어 학습에
- 효율적으로 만든다.
- mini batch를 적절하게 만들어 준다.
- 각 epoch마다 데이터의 순서를 섞어준다.

먼저 Dataset의 상속을 받는 정형화된 데이터 객체를 만든다.

- 생성자, len, getitem과 같은 함수를 오버라이딩한다.

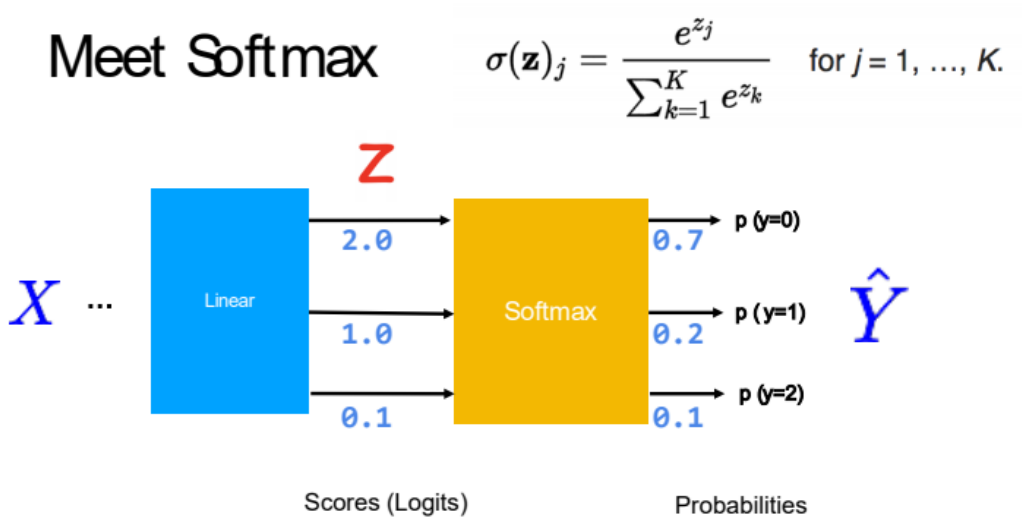
만들어진 dataset 객체를 DataLoader의 매개변수로 사용한다.

num_worker : 하드웨어에서 CPU로 보내는 전송 프로세서 수

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('/content/gdrive/My Drive/Colab N
                        delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = from_numpy(xy[:, 0:-1])
        self.y_data = from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

```
for epoch in range(2):
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = tensor(inputs), tensor(labels)

        # Run your training process
        print(f'Epoch: {i} | Inputs {inputs.data} | Labels {labels.data}')
```

Softmax



Logits : softmax에 입력되는 입력값

Logits을 받아 큰 값은 더 크게, 작은 값은 더 작게 확률 값으로 변환해준다. (미분 가능 하게)

그 후 Loss function으로 cross entropy를 사용하고 one-hot 데이터로 변환한다.

NLLLoss : 적은 개수의 데이터에 해당하는 클래스에 높은 가중치를 주어 학습양을 늘리는 방법

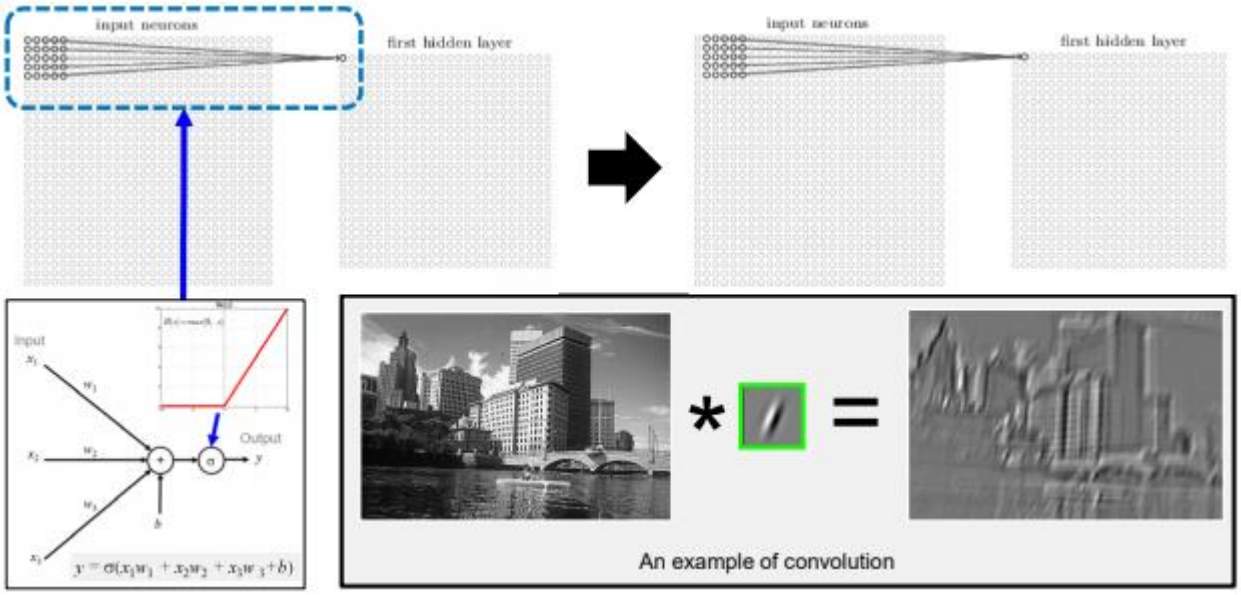
criterion = nn.CrossEntropyLoss() : 이진분류가 아닐 때 사용

momentum : 가중치가 0에 가까워져 업데이트가 잘 이루어지지 않을 때, 이전의 가중치를 고려하여 업데이트 폭을 늘린다.

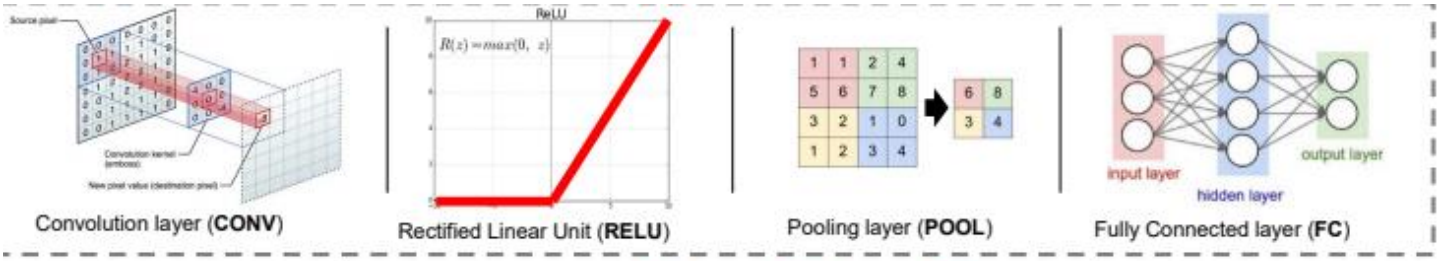
DNN(Deep Neural Networks) : 사람의 뇌의 동작을 모사

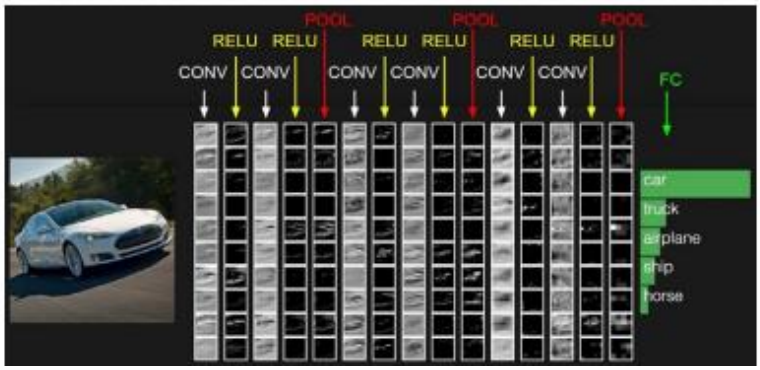
Supervised learning / Unsupervised Learning / Reinforcement Learning

CNN(Convolutional Neural Network)



모든 픽셀에 가중치를 주는 것이 아닌 특정 크기의 kernel에 가중치를 곱한 뒤 활성함수를 통과 시켜 출력값과 mapping하여 feature map을 만든다. **Local connectivity**





Pooling : 임의의 stride크기만큼의 픽셀을 max, avg와 같은 방식으로 중요한 값을 도출해내어 차원을 축소시킨다. (채널의 축은 차원축소 시키지 않는다)

Padding : convolution 수행 시 출력되는 차원의 수를 조절한다.

torch.nn.Conv2d (in_channels, out_channels, kernel_siz)

nn.MaxPool2d (kernel_size)

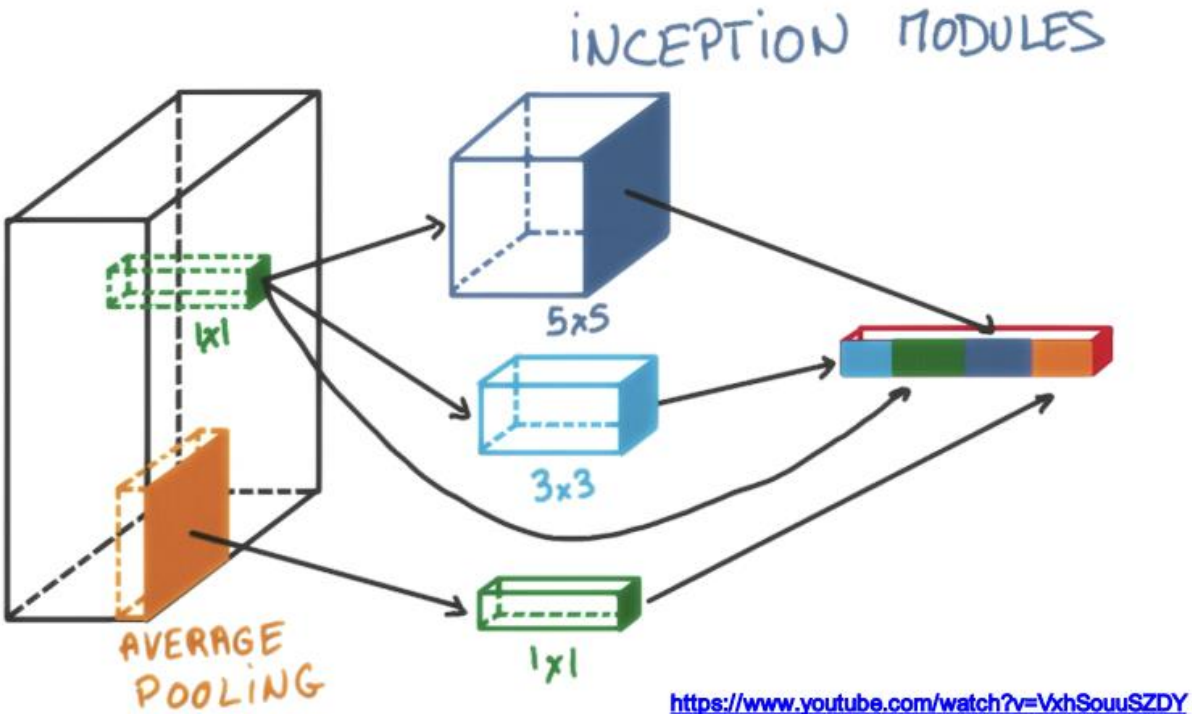
```
self.mp = nn.MaxPool2d(2)
```

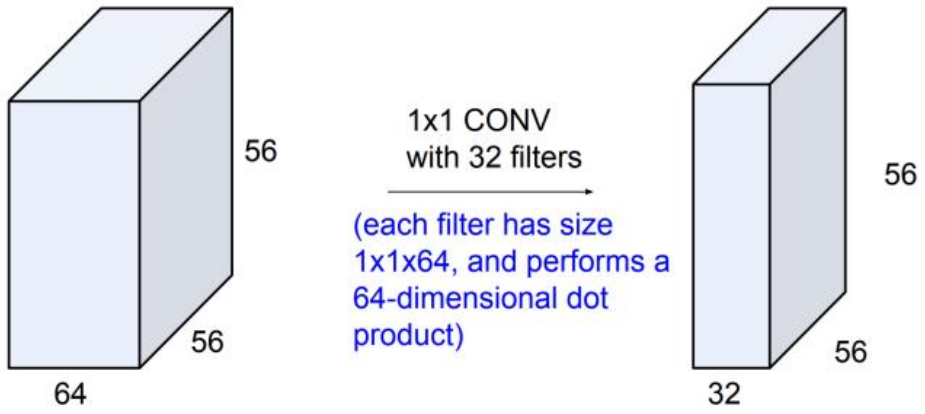
```
self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
```

```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

Inception Modules

여러 개의 convolution layer을 사용함으로써 복잡도는 늘어나지만 비선형성도 늘어난다.





1 x 1의 커널이 채널 전체를 돌며 필터 수를 줄여 차원을 줄이는 것이 목적이다.