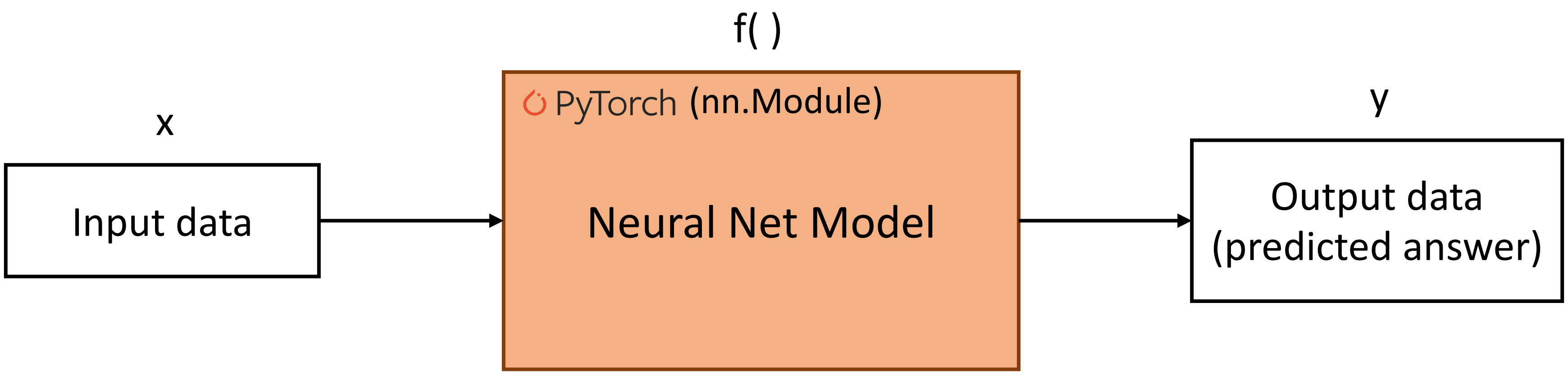
Neural Network with PyTorch module

2019. 09. 26.

2019-2 오픈소스SW프로젝트2

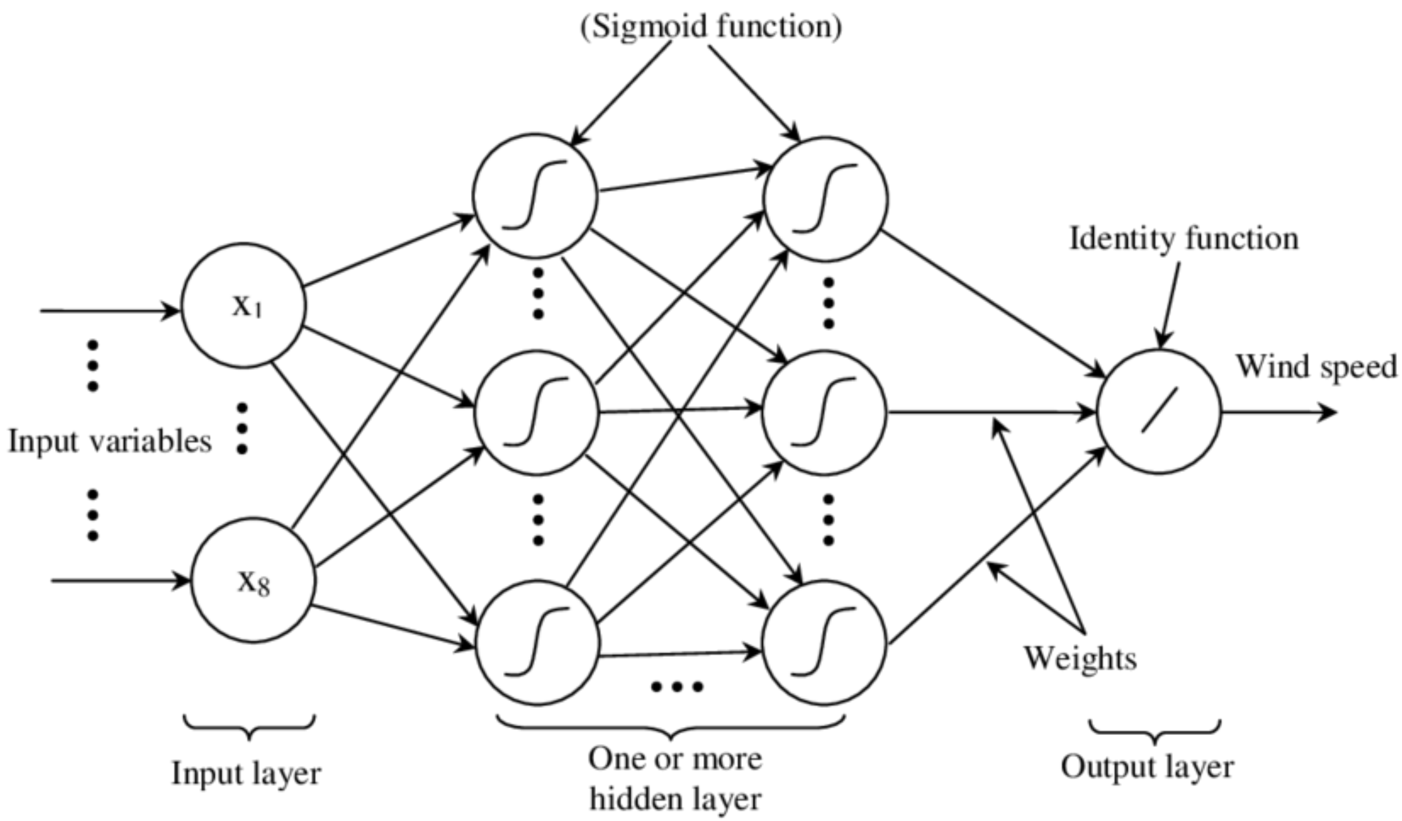
Module

- You can define your own NN model based on nn.Module class.
- nn.Module is base class for all neural network modules in PyTorch.



Module

- Architecture of simple neural network : Multi Layer Perceptron



Architecture of MLP - https://www.researchgate.net/figure/Architecture-of-a-multilayer-perceptron-neural-network_fig5_316351306

Module

- There are load of neural network layers in PyTorch package.
- Don't have to make neural networks or cells from the scratch.
- Some useful layers are provided like Linear, RNN, CNN, Normalization … etc.

class torch.nn.Linear(*in_features, out_features, bias=True*) [source]

class torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True*) [source]

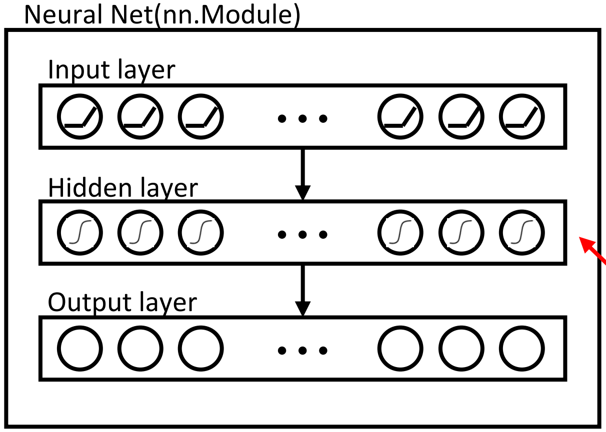
class torch.nn.RNN(**args, **kwargs*) [source]

class torch.nn.Dropout(*p=0.5, inplace=False*) [source]

class torch.nn.BatchNorm1d(*num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True*) [source]

$+\ \alpha$

Module



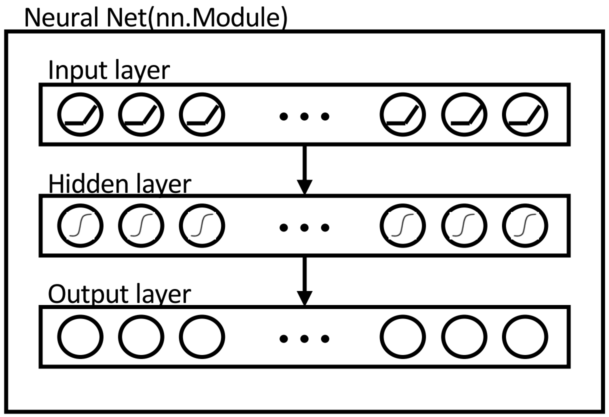

```python
# design neural network with pytorch module
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(NeuralNet, self).__init__()

        self.input_layer = nn.Linear(input_size, hidden_size)
        self.hidden_layer = nn.Linear(hidden_size, hidden_size)
        self.output_layer = nn.Linear(hidden_size, output_size)

    # Must implement forward() for every subclass of nn.Module
    def forward(self, x):
        x = F.relu(self.input_layer(x))
        x = F.relu(self.hidden_layer(x))
        x = self.output_layer(x)
        return x
```

Module

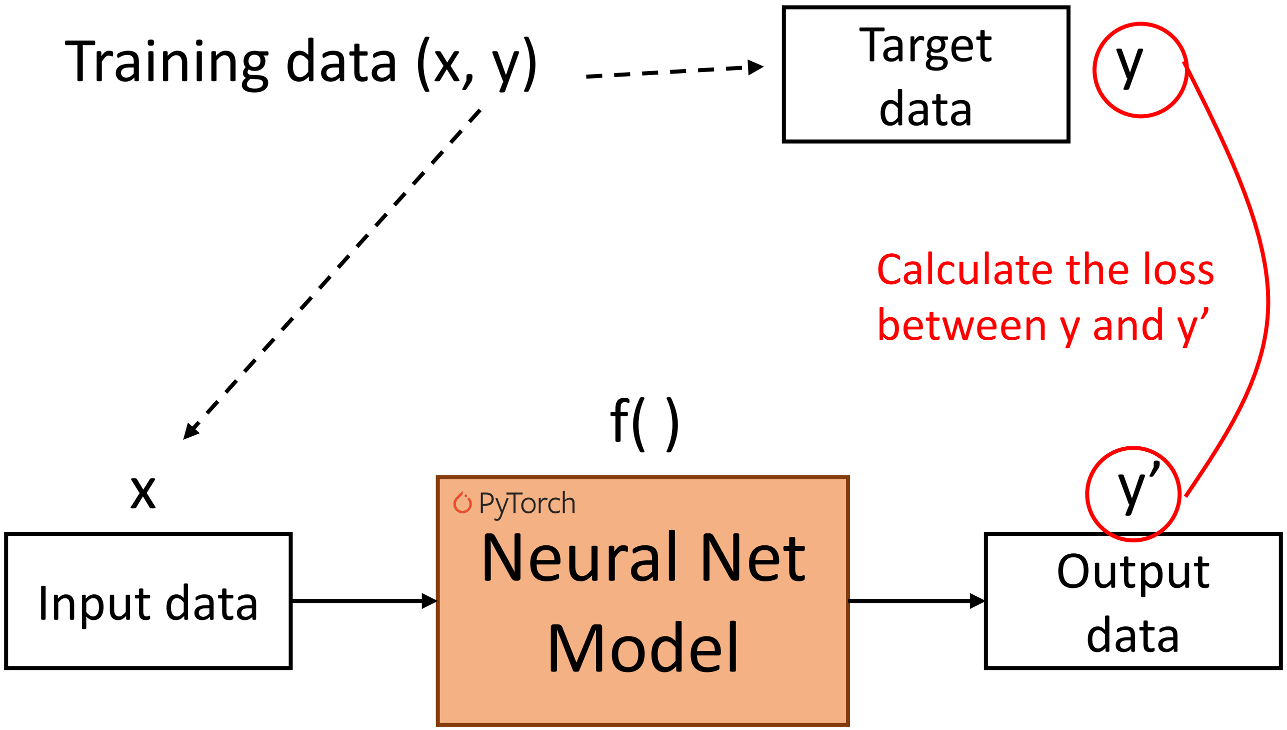
- You can see your model structure simply use '*print(model)*' command.




```python
# create model with defined neuralnet module
model = NeuralNet(100, 50, 100)
print("\nModel Structure: ")
print(model)
```

```
Model Structure:
NeuralNet(
  (input_layer): Linear(in_features=100, out_features=50, bias=True)
  (hidden_layer): Linear(in_features=50, out_features=50, bias=True)
  (output_layer): Linear(in_features=50, out_features=100, bias=True)
)
```

Define Loss Function



- Assume that we have Training pair (***x, y***).
- Our model represents f( ), and returns f(***x***) which means ***y'***.
- What we want is that ***y'*** becomes ***y***. (same as f(***x***) == ***y***)
- So we have to define appropriate loss function for the model and minimize the loss(error).

Define Loss Function

- Use simple Mean Squared Error(MSE) loss function in here.

```
# make dummy input & target data
input_data = torch.randn(100, requires_grad=True)
target_data = torch.full((100,), 30, requires_grad=False) # ground truth
```

```
# forward propagation(the two lines below are functionally identical)
output_data = model(input_data)
#output_data = model.forward(input_data)
```

```
# define loss function
criterion = nn.MSELoss()
loss = criterion(output_data, target_data)
print("Loss: %F" %loss.item())
```

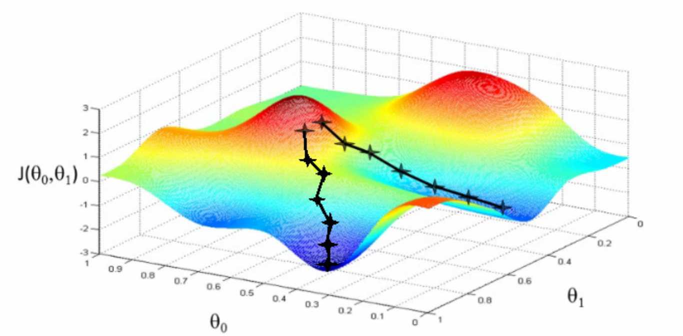
Back prop. and Model Training

- Update parameters simply calculate eq. for every parameters.

*weight_new = weight_old – (learning_rate * gradient)*

- But PyTorch provides some famous & useful optimizing methods in nn.optim package.

- PyTorch already has well defined optimizers such as SGD, Adam, AdaDelta, RMSprop …



Gradient Descent : http://blog.datumbox.com/tuning-the-learning-rate-in-gradient-descent/

Back prop. and Model Training

- Repeat the step until your model can express wanted function.

Typical training step

```
# forward propagation(the two lines below are functionally identical)
output_data = model(input_data)
#output_data = model.forward(input_data)
```

1. Forward propagation

```
# define loss function
criterion = nn.MSELoss()
loss = criterion(output_data, target_data)
print("Loss: %F" %loss.item())
```

2. Calculate loss(error)

```
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```
# calculate gradient on autograd backpropagation
# if you call backward(), you will get accumulated gradient for all data in graph
# so you have to call zero_grad() before calling backward() to erase all buffered d
model.zero_grad()
optimizer.zero_grad()
loss.backward()
# update the parameters in model
optimizer.step()
```

3. Back propagation

4. Update parametes

Check Model Parameters(Weights)

- If you want to check model's parameters, just see objects' attributes(layers) weight.
- You can access parameter tensors whenever you want.

Before updating step

```
print(model.input_layer.weight)
```

```
# calculate gradient on autograd backpropagation
# if you call backward(), you will get accumulated gradient for all da
# so you have to call zero_grad() be
model.zero_grad()
optimizer.zero_grad()
loss.backward()
# update the parameters in model
optimizer.step()
```

After updating step

```
print(model.input_layer.weight)
```

```
Parameter containing:
tensor([[-0.0436, -0.0579,  0.0661,  ..., -0.0139, -0.0383,  0.0531],
        [ 0.0055, -0.0015, -0.0363,  ...,  0.0310,  0.0455, -0.0943],
        [ 0.0226,  0.0655, -0.0408,  ...,  0.0188, -0.0233, -0.0968],
        ...,
        [-0.0380, -0.0347,  0.0190,  ..., -0.0483, -0.0047,  0.0718],
        [-0.0280,  0.0785, -0.0471,  ..., -0.0744,  0.0014,  0.0512],
        [ 0.0924,  0.0954,  0.0069,  ..., -0.0917, -0.0099, -0.0450]],
       requires_grad=True)
```

```
Parameter containing:
tensor([[-0.0438, -0.0573,  0.0661,  ..., -0.0137, -0.0380,  0.0532],
        [ 0.0060, -0.0031, -0.0364,  ...,  0.0306,  0.0448, -0.0946],
        [ 0.0226,  0.0655, -0.0408,  ...,  0.0188, -0.0233, -0.0968],
        ...,
        [-0.0380, -0.0347,  0.0190,  ..., -0.0483, -0.0047,  0.0718],
        [-0.0280,  0.0785, -0.0471,  ..., -0.0744,  0.0014,  0.0512],
        [ 0.0908,  0.1003,  0.0072,  ..., -0.0905, -0.0079, -0.0439]],
       requires_grad=True)
```

Code

- https://github.com/leechaeyoon/pytorch-tutorial
  - tensor_creation_and_operation.py
  - neural_network_with_pytorch_module.py