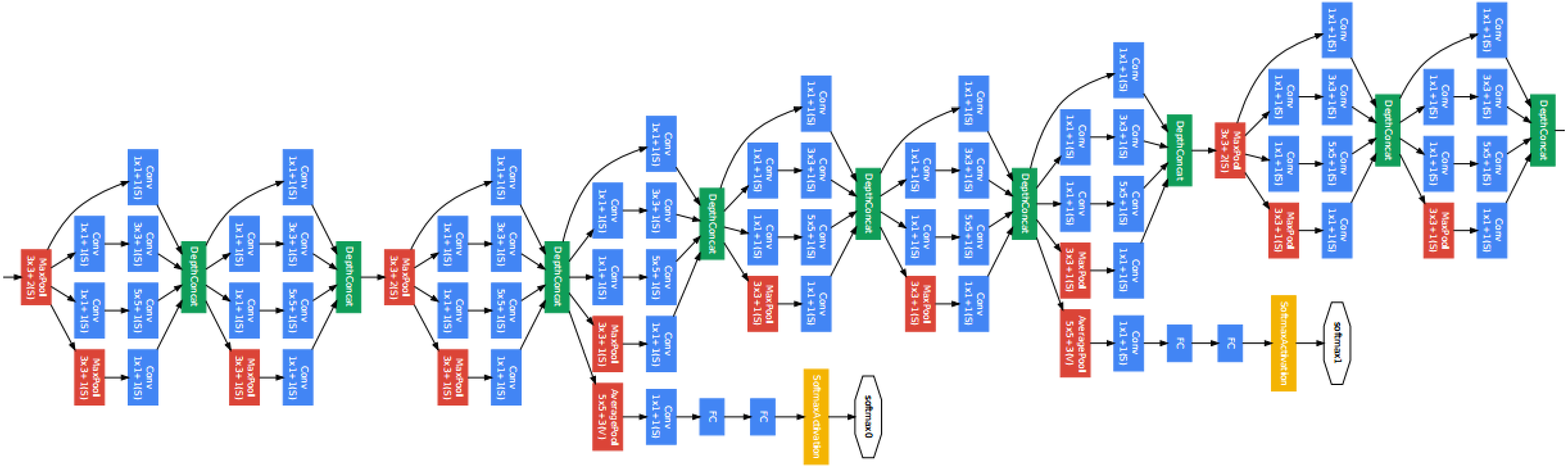
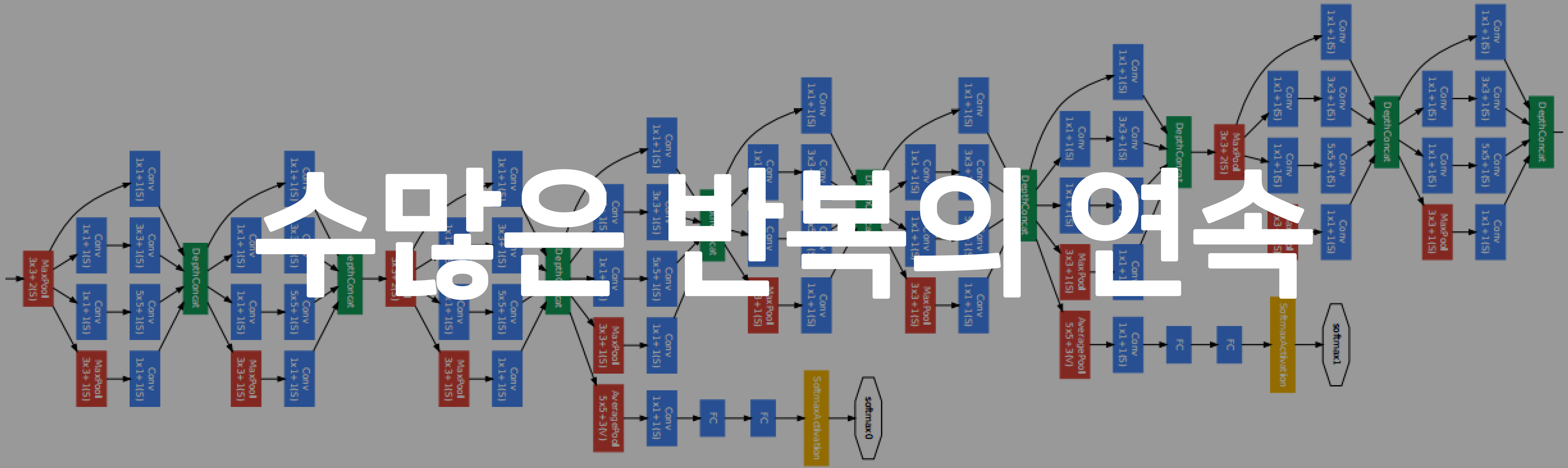
AutoGrad & Optimizer

TEAMLAB director
최성철

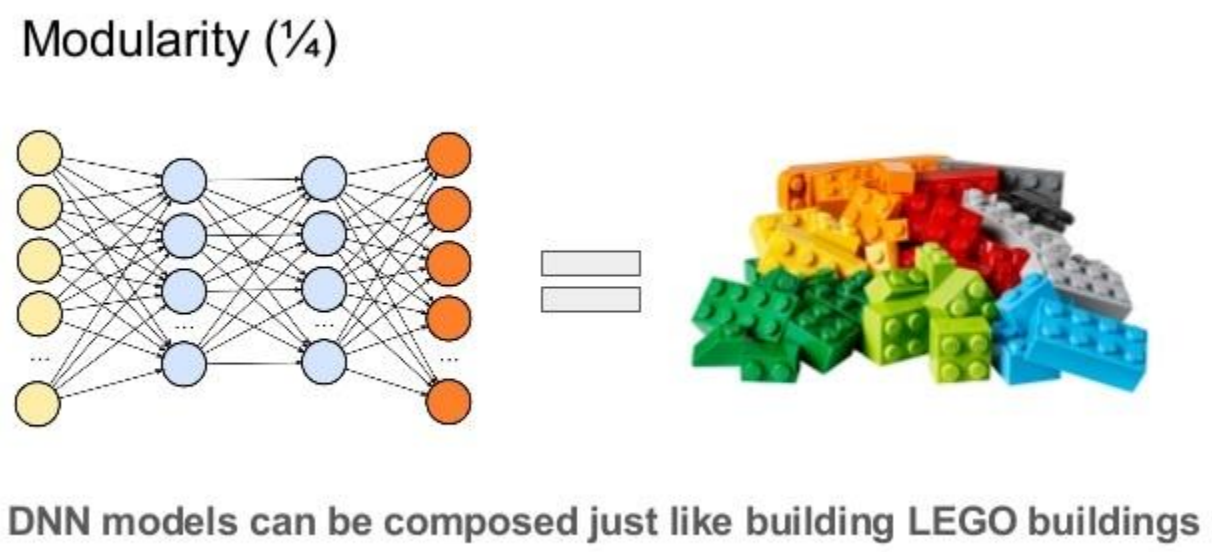
논문을 구현해 보자!


MaxPool 3x3+2(S)
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
DepthConcat
AveragePool 5x5+3(V)
FC
SoftmaxActivation
softmax0
softmax1

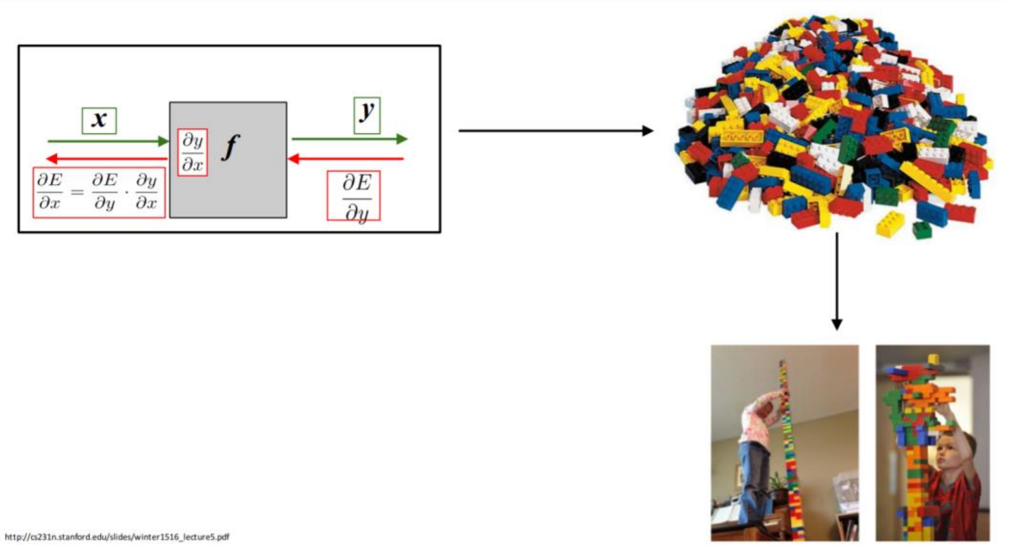


수많은 반복의 연속

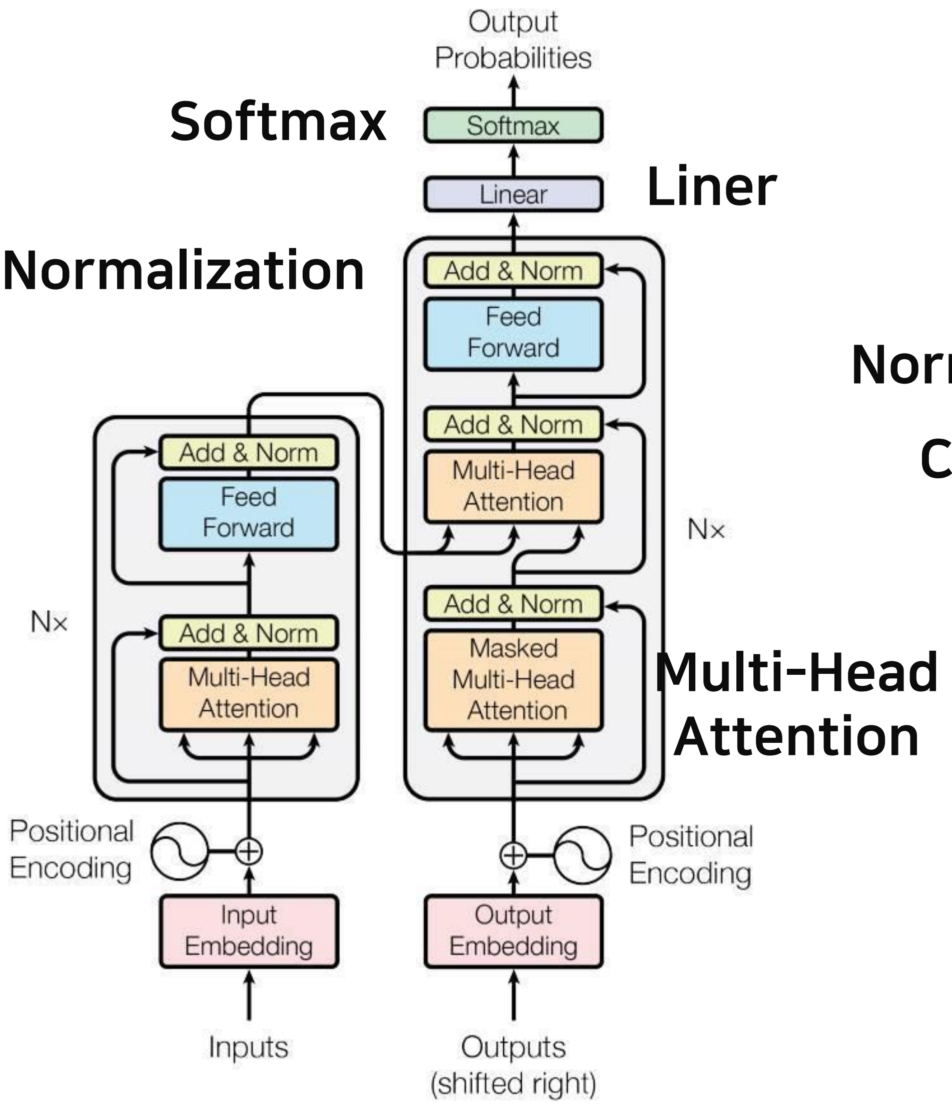
Layer = Block

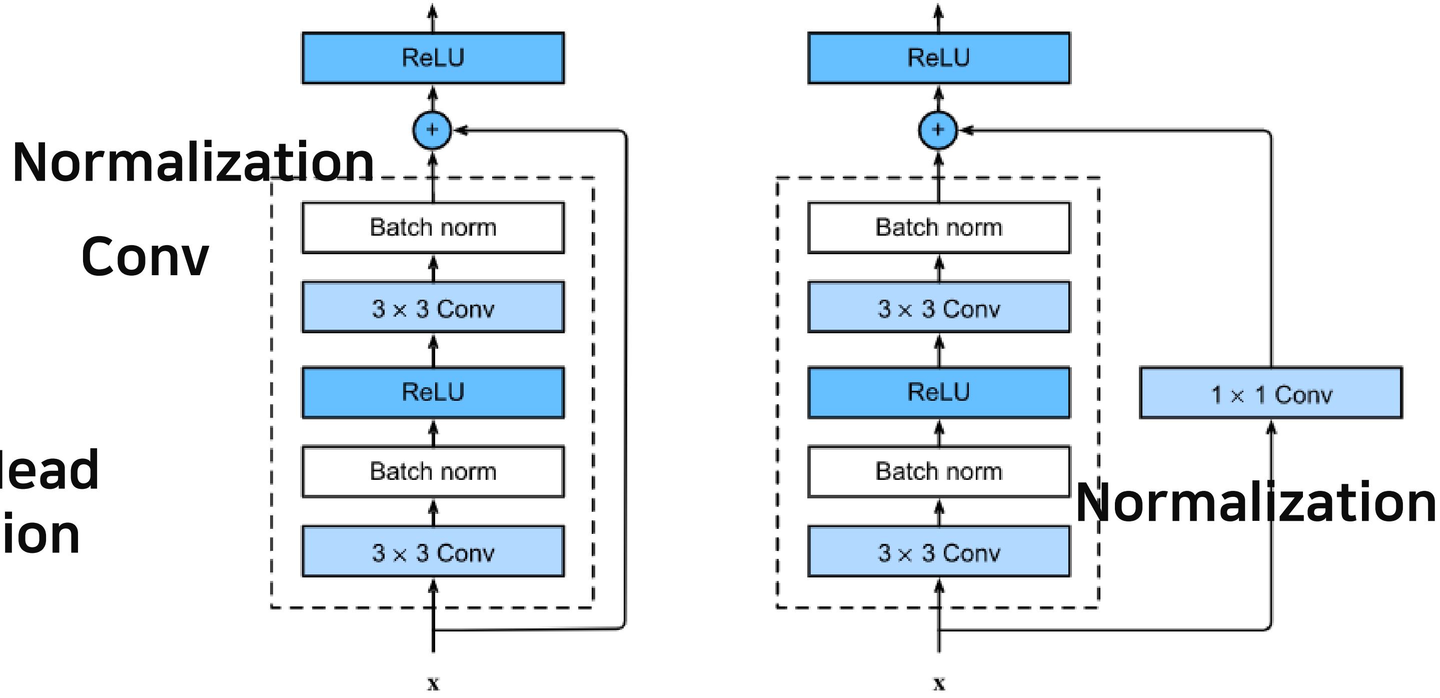


https://bit.ly/3lv0eAJ



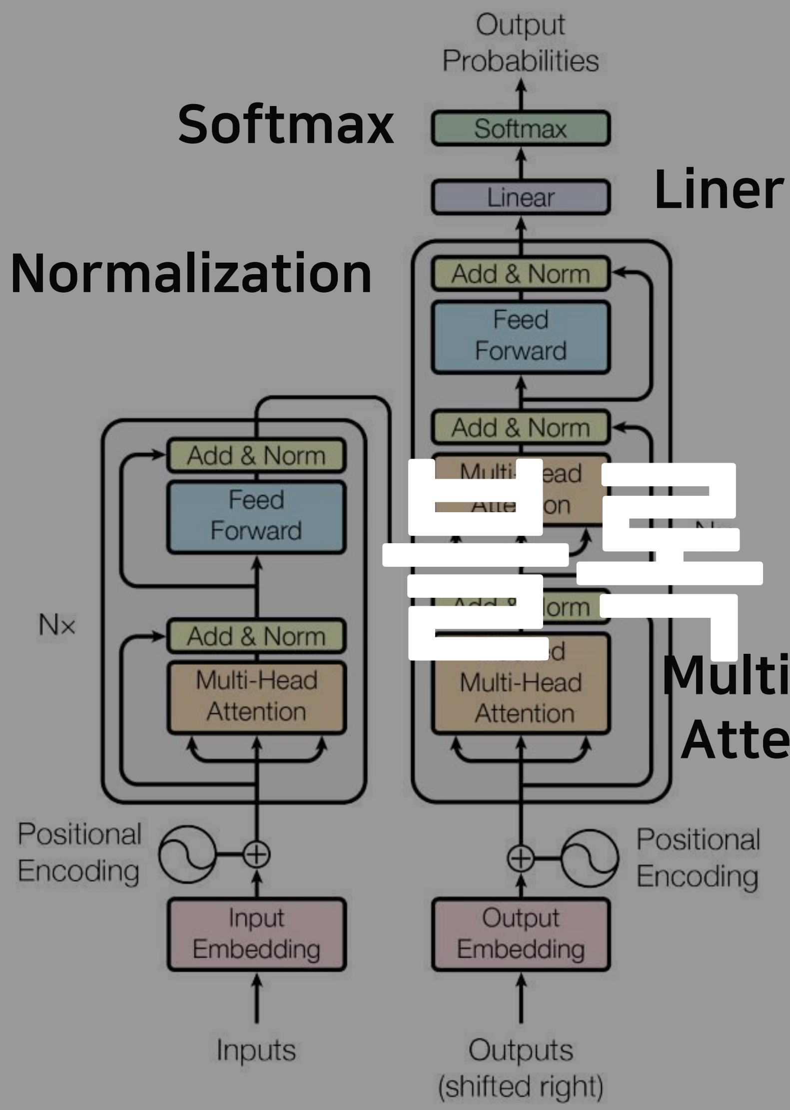
https://bit.ly/3lxGTPe

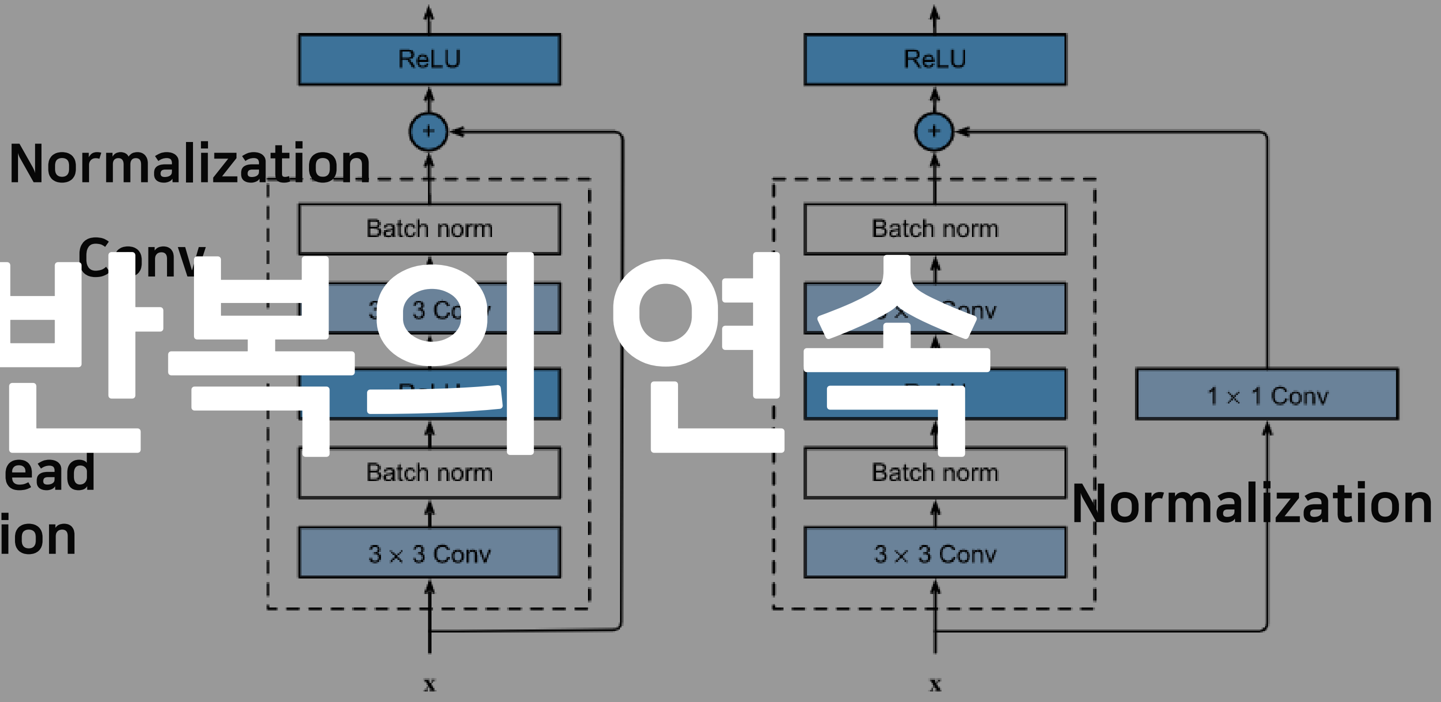




boostcamp ai tech

블록 반복의 연속


Output Probabilities
Softmax
Softmax
Linear
Liner
Normalization
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Multi-Head Attention
Multi-Head Attention
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

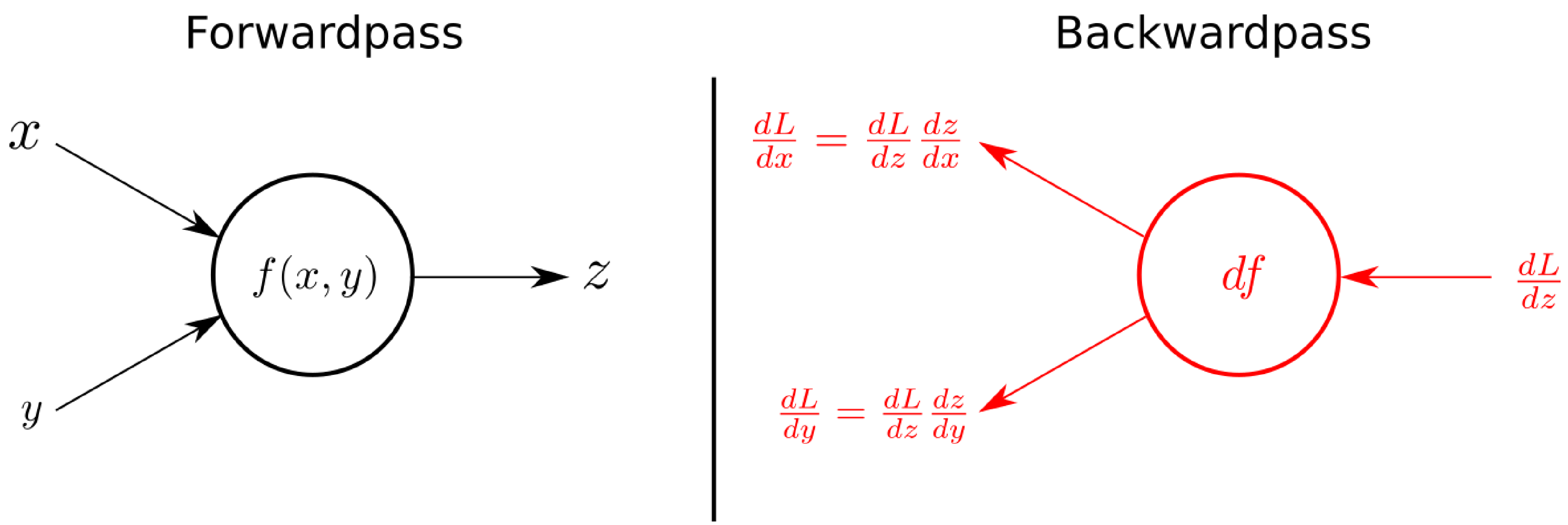

Normalization
Conv
ReLU
Batch norm
3 × 3 Conv
Batch norm
3 × 3 Conv
x
ReLU
Batch norm
Batch norm
3 × 3 Conv
1 × 1 Conv
Normalization
x

torch.nn.Module

torch.nn.Module

- 딥러닝을 구성하는 Layer의 base class
- Input, Output, Forward, Backward 정의
- 학습의 대상이 되는 parameter(tensor) 정의

torch.nn.Module



https://github.com/Vercaca/NN-Backpropagation

nn.Parameter

- Tensor 객체의 상속 객체
- nn.Module 내에 attribute가 될 때는 required_grad=True 로 지정되어 학습 대상이 되는 Tensor
- 우리가 직접 지정할 일은 잘 없음

  : 대부분의 layer에는 weights 값들이 지정되어 있음

nn.Parameter


```python
class MyLiner(nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features

        self.weights = nn.Parameter(
                torch.randn(in_features, out_features))

        self.bias = nn.Parameter(torch.randn(out_features))


    def forward(self, x : Tensor):
        return x @ self.weights + self.bias
```

Backward

- Layer에 있는 Parameter들의 미분을 수행
- Forward의 결과값 (model의 output=예측치)과 실제값간의 차이(loss) 에 대해 미분을 수행
- 해당 값으로 Parameter 업데이트

Backward


```
for epoch in range(epochs):
    ……
    # Clear gradient buffers because we don't want any gradient from previous epoch to
carry forward
    optimizer.zero_grad()

    # get output from the model, given the inputs
    outputs = model(inputs)

    # get loss for the predicted output
    loss = criterion(outputs, labels)
    print(loss)
    # get gradients w.r.t to parameters
    loss.backward()

    # update parameters
    optimizer.step()
    ………
```

Backward from the scratch

- 실제 backward는 Module 단계에서 직접 지정가능
- Module에서 backward 와 optimizer 오버라이딩
- 사용자가 직접 미분 수식을 써야하는 부담

  → 쓸일은 없으나 순서는 이해할 필요는 있음

```python
class LR(nn.Module):
    def __init__(self, dim, lr=torch.scalar_tensor(0.01)):
        super(LR, self).__init__()
        # intialize parameters
        self.w = torch.zeros(dim, 1, dtype=torch.float).to(device)
        self.b = torch.scalar_tensor(0).to(device)
        self.grads = {"dw": torch.zeros(dim, 1, dtype=torch.float).to(device),
                      "db": torch.scalar_tensor(0).to(device)}
        self.lr = lr.to(device)

    def forward(self, x):
        ## compute forward
        z = torch.mm(self.w.T, x)
        a = self.sigmoid(z)
        return a
```

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

```python
    def sigmoid(self, z):
        return 1/(1 + torch.exp(-z))

    def backward(self, x, yhat, y):
        ## compute backward
        self.grads["dw"] = (1/x.shape[1]) * torch.mm(x, (yhat - y).T)
        self.grads["db"] = (1/x.shape[1]) * torch.sum(yhat - y)
```

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

```python
    def optimize(self):
        ## optimization step
        self.w = self.w - self.lr * self.grads["dw"]
        self.b = self.b - self.lr * self.grads["db"]
```

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$:= \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

End of Document
Thank You.