Training DNN

HOML – chapter11

TAVE Research DL001
Changdae Oh

2021. 01. 10

Contents

The challenges of training Deep Neural Networks

- Vanishing gradient / Exploding gradient
- Insufficient data, Difficulties in labeling
- Training can be extremely slow
- Risk of overfitting

Contents

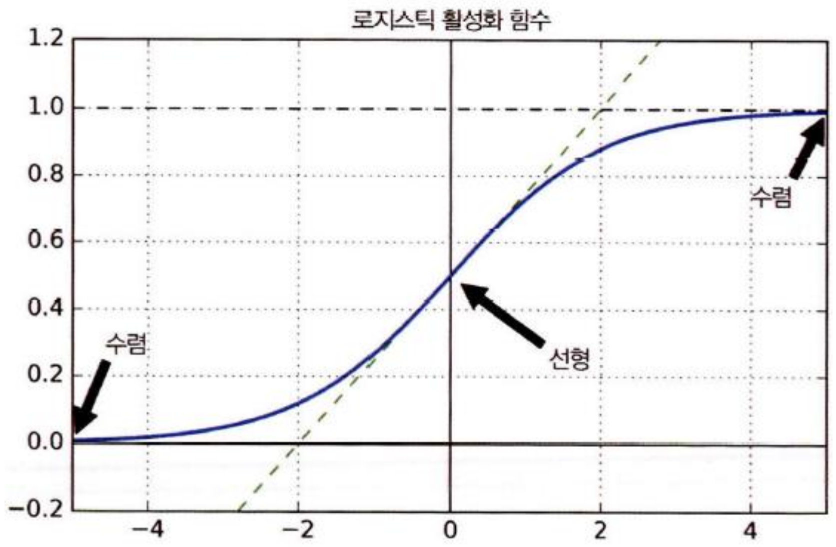
Gradient vanishing/exploding

Vanishing gradient

- When training DNN, **the gradient gradually decreases** as it gets closer to the input layer in the backpropagation process

Exploding gradient

- On the contrary, as it approaches the input layer, **the gradient gradually increases** and diverges.



How can we solve this problem?

1) Initializing strategy
2) Activation functions
3) Batch Normalization
4) Gradient Clipping

Gradient vanishing/exploding

1. Initializing strategy

Past weight initialization strategy

: Sampling from
Normal distribution (Mean : 0, Var : 1)

Glorot initialization (Xavier initialization)

Normal distribution (Mean : 0, Var : $\frac{1}{fan_{avg}}$)

uniform distribution ($-\sqrt{\frac{3}{fan_{avg}}}$, $+\sqrt{\frac{3}{fan_{avg}}}$)

LuCun initialization

Just replace fan-avg with fan-in

He initialization

Just double scale variance

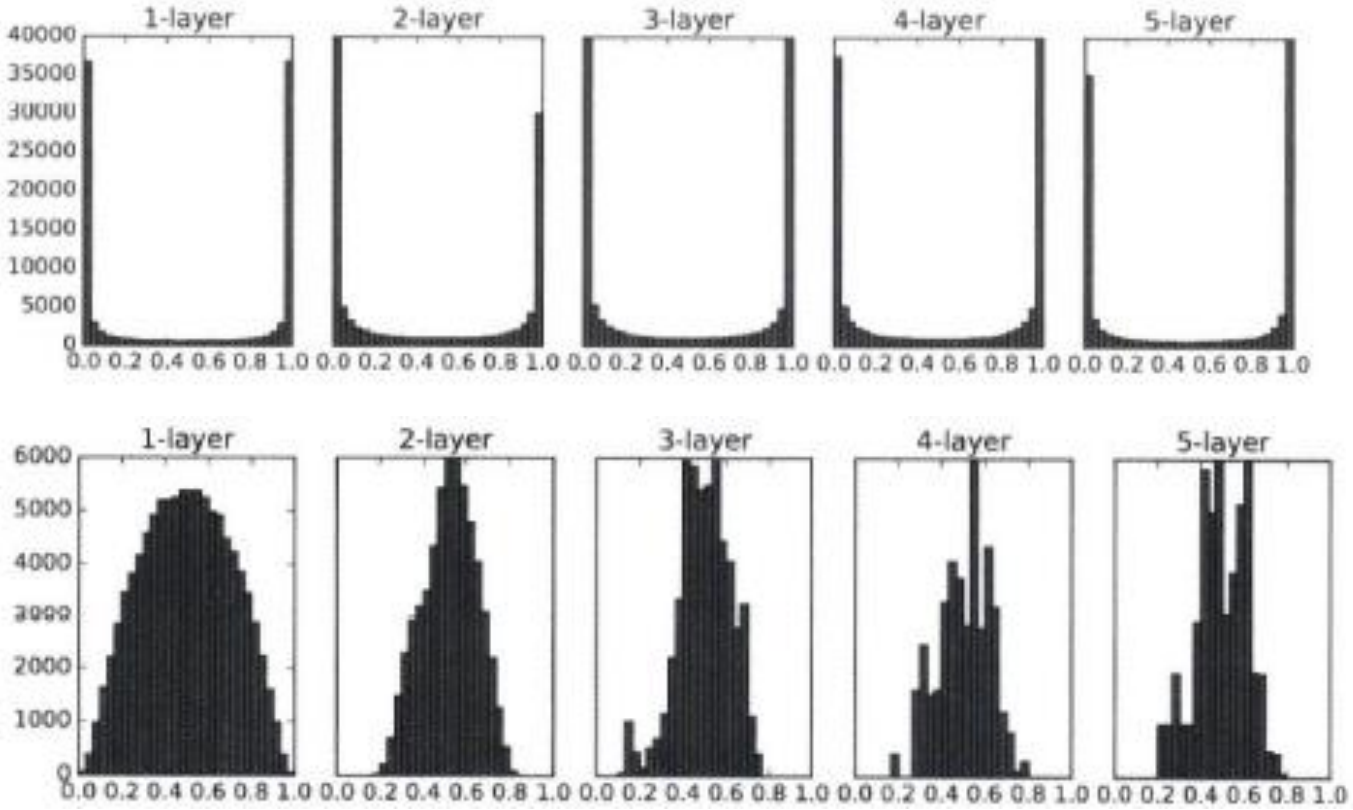
strategy	Activation	Variance(Normal dist.)
Glorot	sigmoid, tanh, softmax	1 / fan_avg
He	ReLU s	2 / fan_in
LuCun	SELU	1 / fan_avg

fan-in : # of input units to weight matrix

fan-out : # of output units from weight matrix

Gradient vanishing/exploding

1. Initializing strategy



Ex) sigmoid & Glorot init

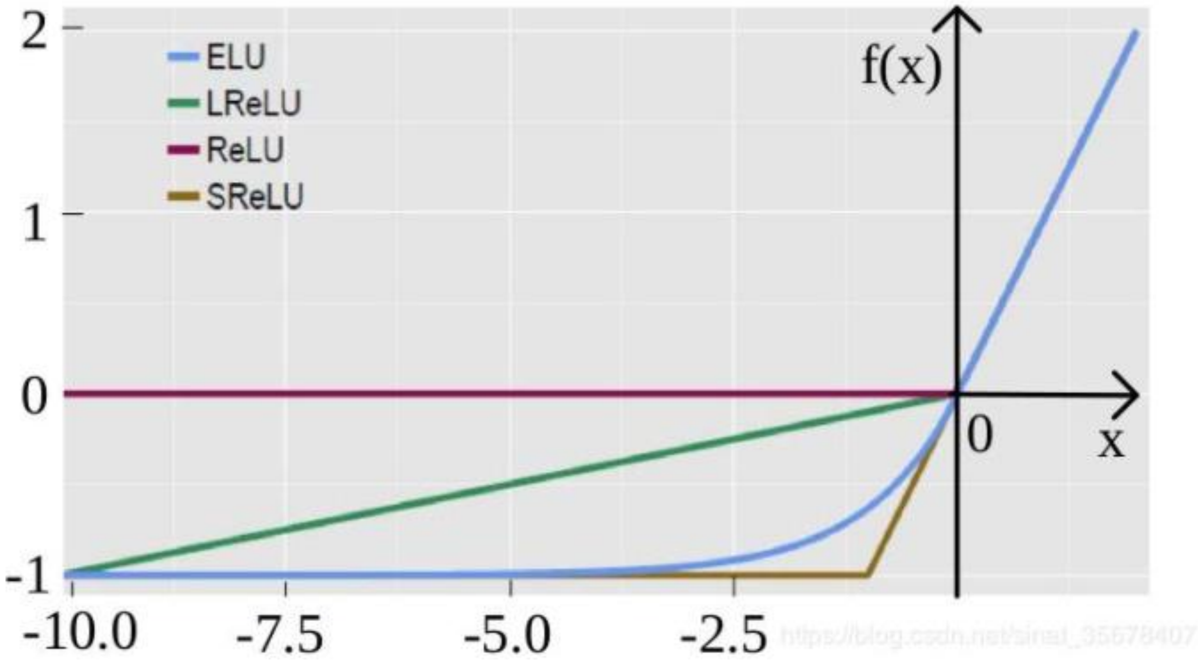
Gradient vanishing/exploding

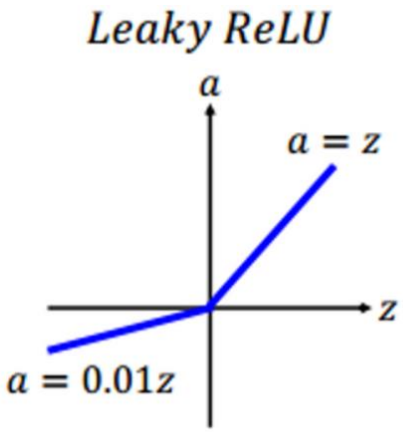
2. non-convergent activation function

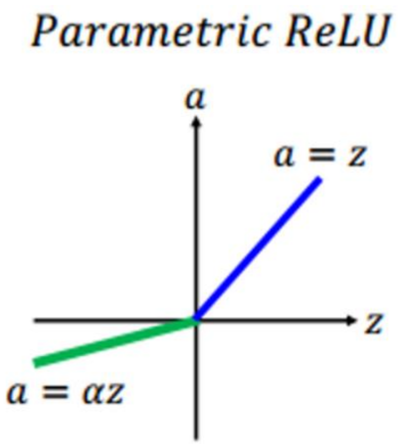
< ReLU >

- Not converged to specific positive value
- dying ReLU

- LeakyReLU / RReLU / PReLU
- ELU / SELU





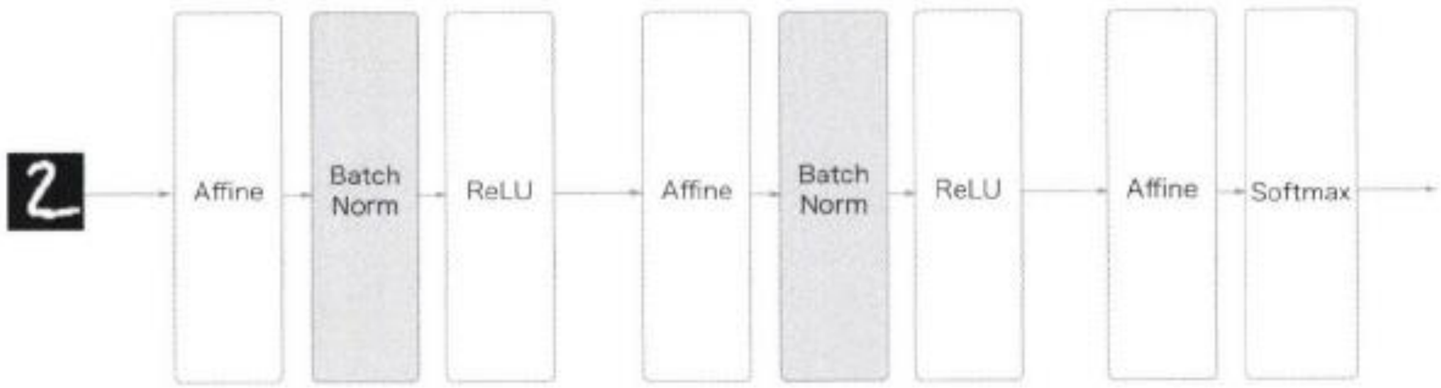


Gradient vanishing/exploding

3. Batch Normalization

- Align the input to the origin and normalize it
- In each layer, two new parameters are used to scale and shift the results.

➡ *Force each layer to spread the activations moderately*



algorithm

1. $\boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$
2. $\boldsymbol{\sigma}_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^{\,2} + \varepsilon}}$
4. $\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$

advantage

- Learning faster
- Rubust to initializing
- Restraint overfitting

disadvantage

- Increase complexity
- Computing resource

Gradient vanishing/exploding

4. Gradient Clipping

- If the norm of the gradient is greater than some threshold, scale it down before applying SGD update

➡ *Take a step in the same direction, but a smaller step*

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$

if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**

$\quad \hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$

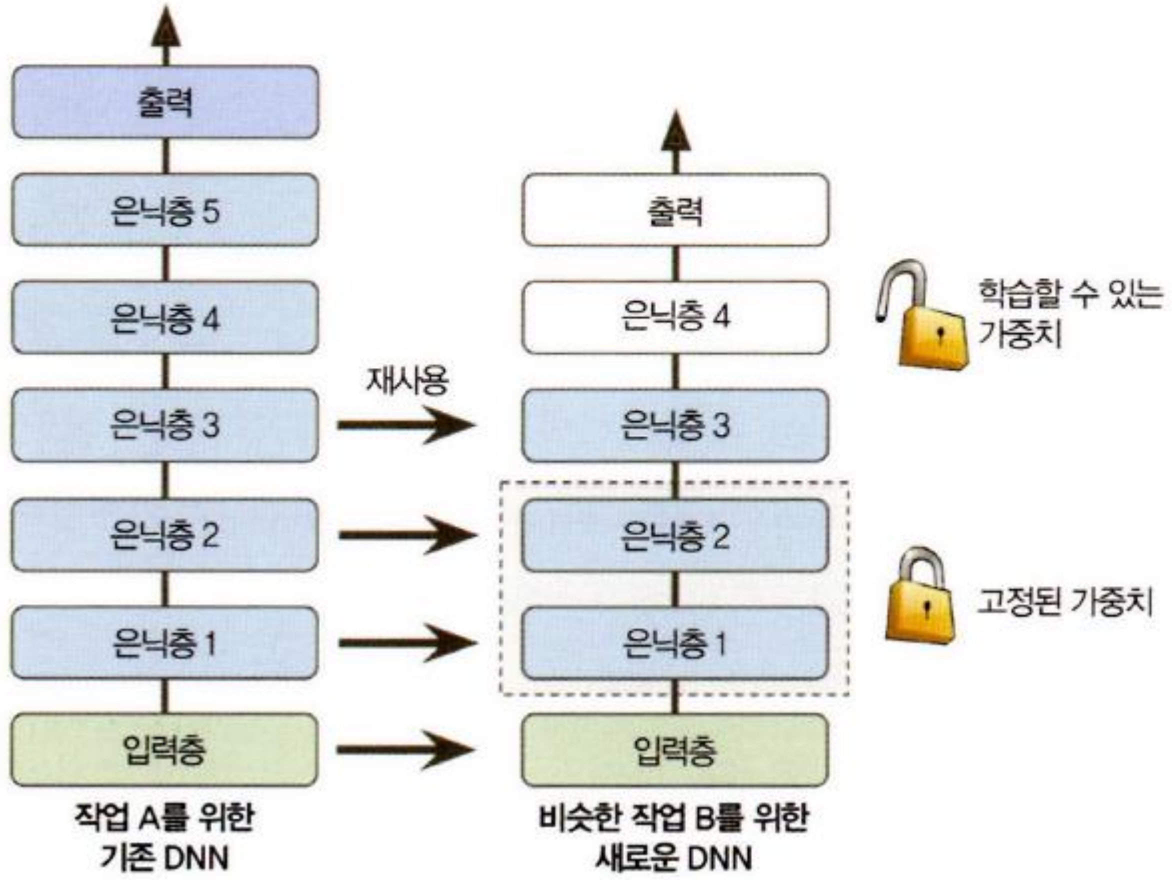
end if

Contents

Reusing pre-trained layer

Transfer learning

- Find out if there is a system that handles a **similar type of problem to be solved**
- **Reuse the lower layers** of the neural network.



✓ Learning faster

✓ Significantly reduce the amount of train data required

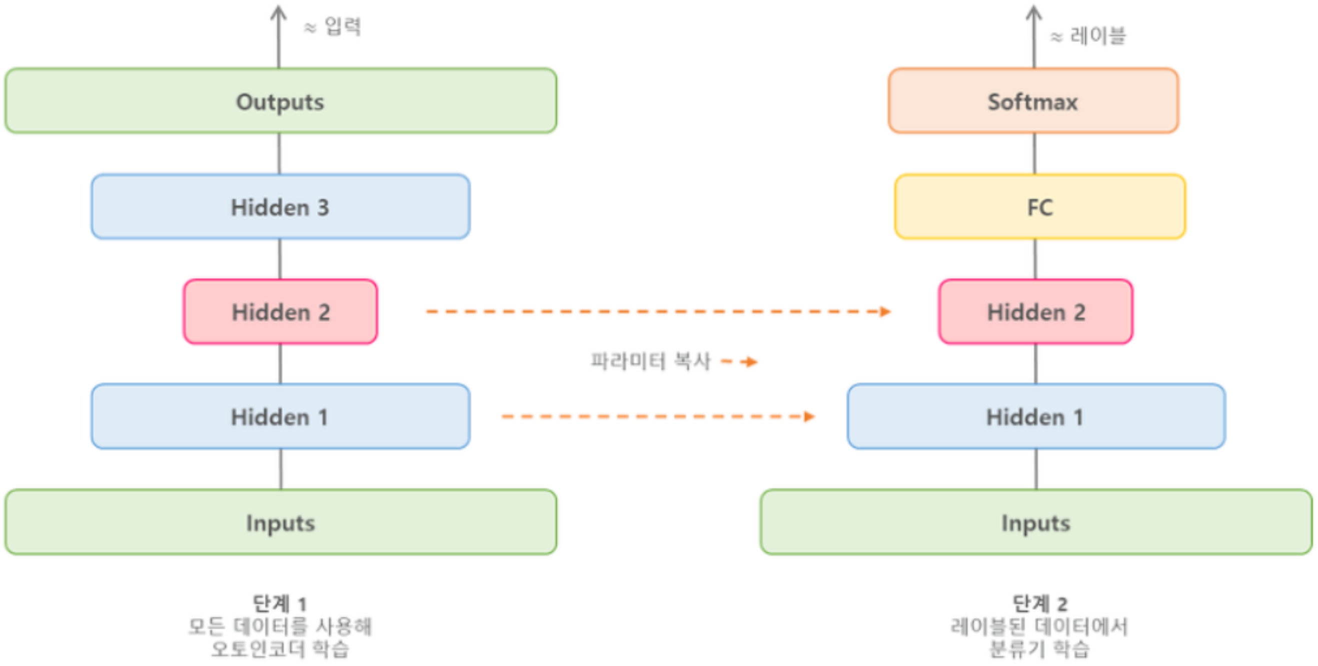
< Process >

1. Freeze all reusable layers.
2. Fitting the model, evaluate performance
3. Compare performance by melting the freezing of one or two layers at the top of the frozen layers and updating the weights via backpropagation

Reusing pre-trained layer

Unsupervised pretraining

- don't have enough labeled training data?



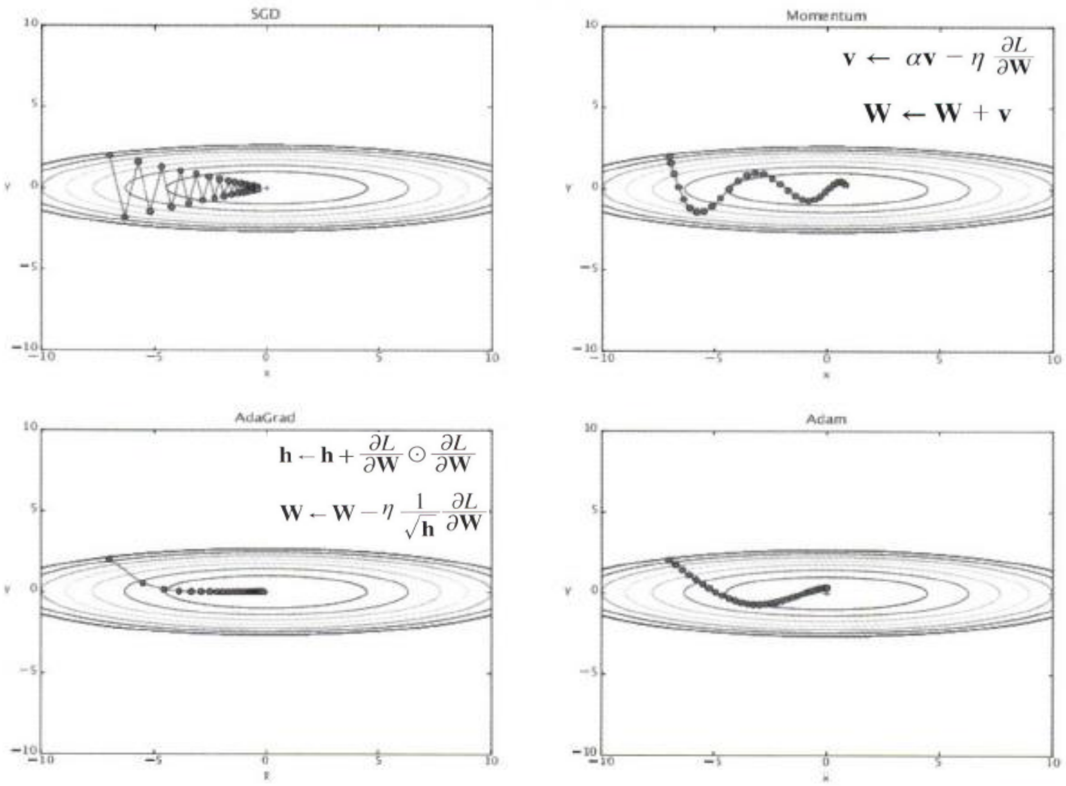
< Process >

1. Train an unsupervised learning model on unlabeled data. (auto encoder, GAN)
2. Reuse lower layers of those models, On top of that, Add hidden layers and an output layer suitable for the new task.
3. Fine-tuning the final network with supervised learning using labeled training samples

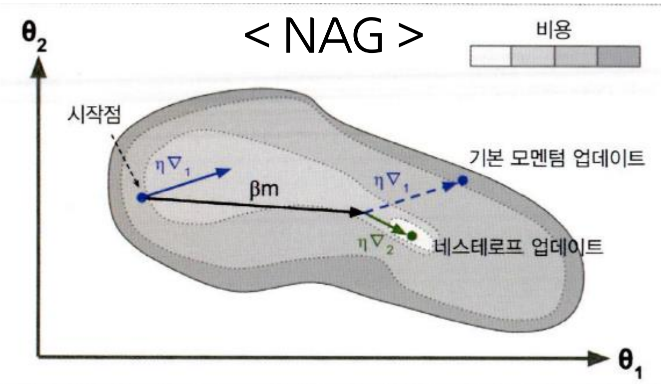
or Pretrain in sub-task !!

Contents

Fast optimizer



- ➤ Momentum / Nesterov accelerated gradient
- ➤ Adagrad / RMSProp
- ➤ Adam / AdaMax / Nadam

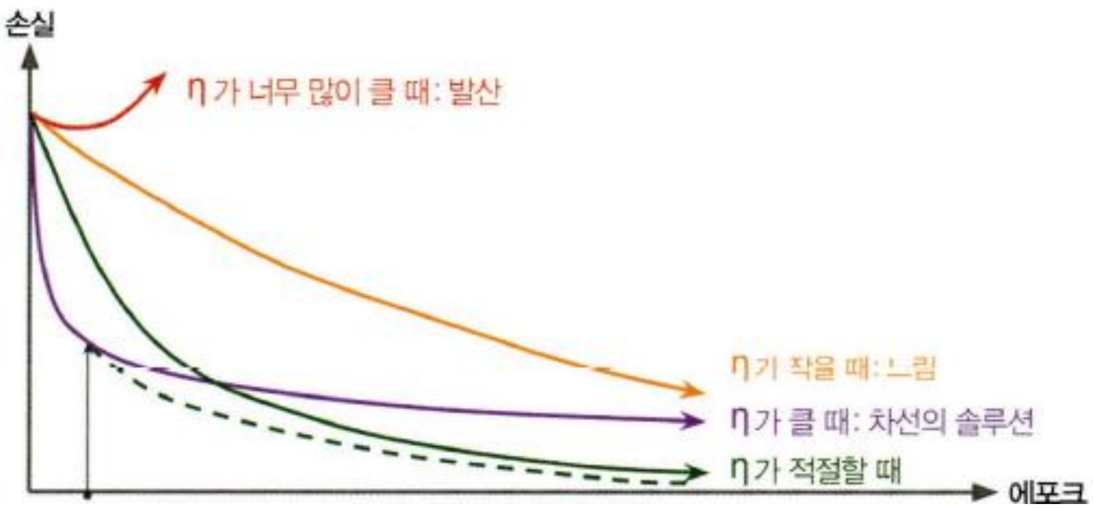


Optimizer	Convergence Speed	Convergence Quality
SGD	*	***
Momentum	**	***
NAG	**	***
Adagrad	***	* (stop too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

Fast optimizer

Learning rate Scheduling – LR decay method

> Starting with a large learning rate
> and decaying the learning rate when the learning rate is slow,
> the solution can be found faster than the optimal fixed learning rate



- Power scheduling
- Exponential scheduling
- Piecewise constant scheduling
- Performance scheduling
- 1 cycle scheduling

Contents

Avoid overfitting by regularization

L1, L2 Regularization

Norm term $\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$

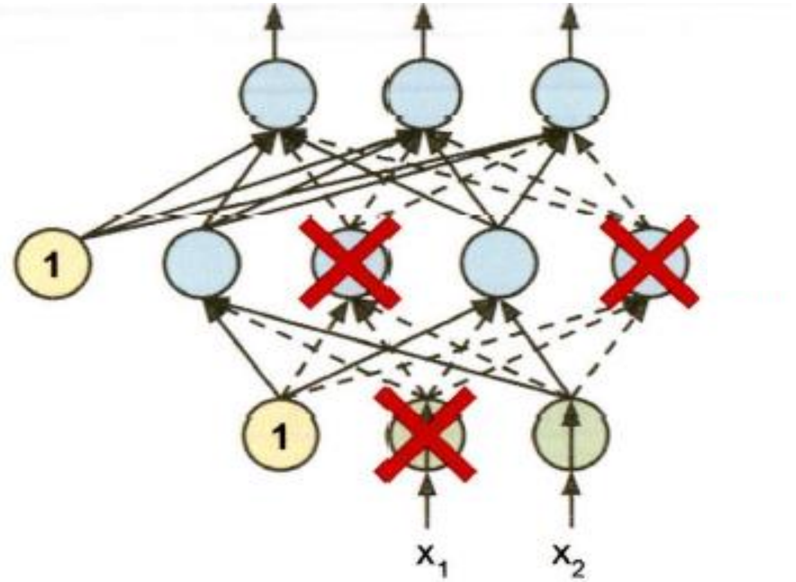
L1 $$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y}_i) + \frac{\lambda}{2}|w|\}$$

L2 $$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y}_i) + \frac{\lambda}{2}|w|^2\}$$

https://light-tree.tistory.com/125

Dropout

With hyperparameter p



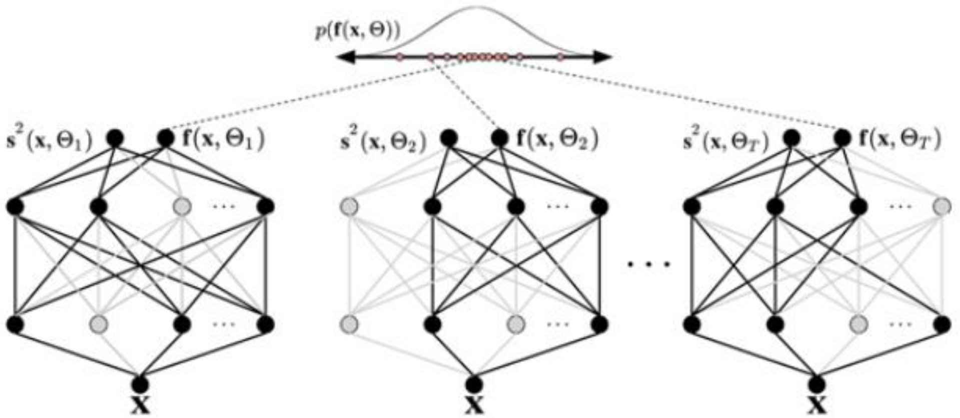
- Make a model **more stable**
- Can be seen as a kind of **ensemble** of individual NNs

End of training, the remaining connection weights are **multiplied by the keep probability**.

There is no action during the testing.

Avoid overfitting by regularization

Monte Carlo Dropout



Typical dropout techniques are applied only during training, and predictions are made using all neurons in the model during testing.

⬇

What if dropout is applied to both training and testing processes?

Max-norm regularization

- For each neuron, norm of the input connection weight is limited

$$\| \mathbf{w} \|_2 \leq r$$

- At the end of each training step, the norm of weights is calculated and scaled.

$$\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\|w\|_2}$$

Summary

hyper-parameter	nice default value
initializing	He
activation	ELU
normalizing	if network is deep - BN
regularization	early stopping / L2
optimizer	momentum(or RMSProp, Nadam)
LR schedule	1-cycle

Exception

- If you need a 'sparse' model, use L1 regularization.
- If a model with a fast response is needed, reduce the number of layers and merge the batch normalization layer into the previous layer. or use LeakyReLU / ReLU
- Use MCdropout for applications that are risk-sensitive, and whose speed is not very important.

DL from Scratch

Step 8 ~ 16

- Big Picture
- [goal 1] Automatic Differentiation
- [goal 2] Code Improvement

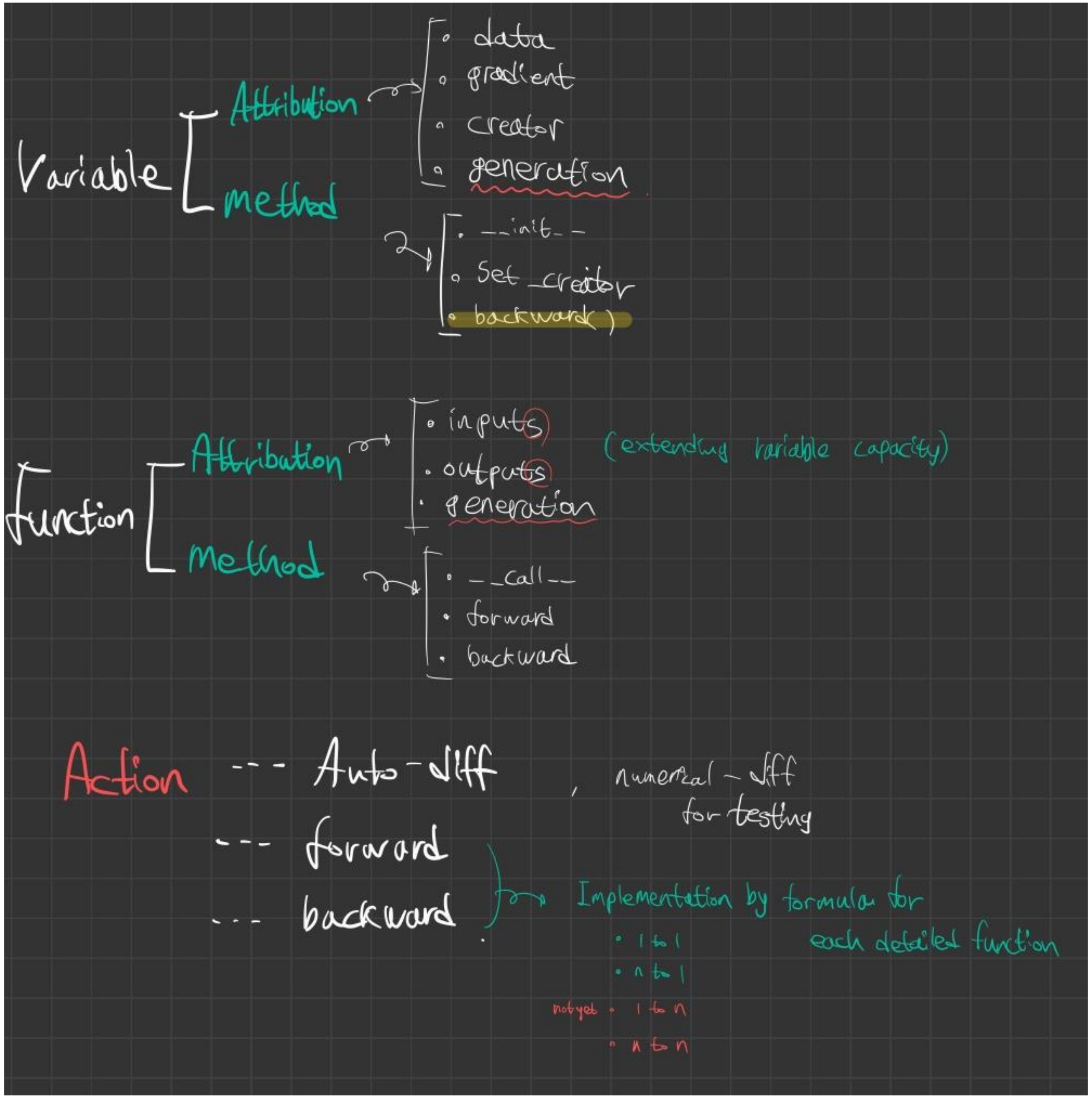
Review & Big Picture

So far. (~ step 16)

What subcomponents are needed
to implement neural networks?

Variable

Function



Automatic Differentiation

< Logic >

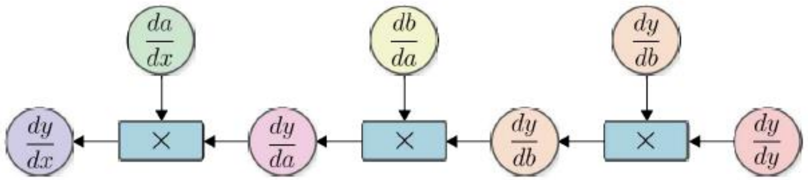
$$\frac{dy}{dx} = \left(\left(\frac{dy}{dy}\frac{dy}{db}\right)\frac{db}{da}\right)\frac{da}{dx}$$

① $\frac{dy}{dy}\frac{dy}{db} = \frac{dy}{db}$

② $\frac{dy}{db}\frac{db}{da} = \frac{dy}{da}$

③ $\frac{dy}{da}\frac{da}{dx} = \frac{dy}{dx}$

< Computation graph >




```python
class Variable:
    def __init__(self, data):
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError('{} is not supported'.format(type(data)))

        self.data = data
        self.grad = None
        self.creator = None

    def set_creator(self, func):
        self.creator = func

    def backward(self):
        if self.grad is None:
            self.grad = np.ones_like(self.data)

        funcs = [self.creator]
        while funcs:
            f = funcs.pop()
            x, y = f.input, f.output
            x.grad = f.backward(y.grad)

            if x.creator is not None:
                funcs.append(x.creator)
```

```python
def as_array(x):
    if np.isscalar(x):
        return np.array(x)
    return x

class Function:
    def __call__(self, input):
        x = input.data
        y = self.forward(x)
        output = Variable(as_array(y))
        output.set_creator(self)
        self.input = input
        self.output = output
        return output

    def forward(self, x):
        raise NotImplementedError()

    def backward(self, gy):
        raise NotImplementedError()
```

```python
class Square(Function):
    def forward(self, x):
        y = x ** 2
        return y

    def backward(self, gy):
        x = self.input.data
        gx = 2 * x * gy
        return gx
```

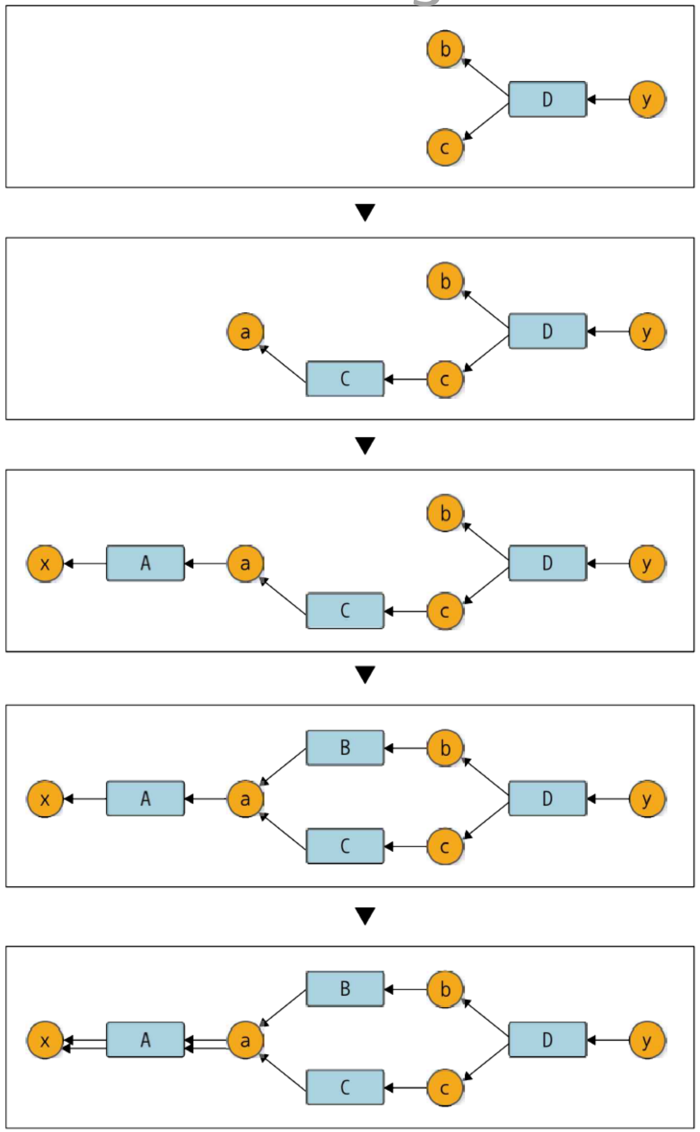
```python
class Exp(Function):
    def forward(self, x):
        y = np.exp(x)
        return y

    def backward(self, gy):
        x = self.input.data
        gx = np.exp(x) * gy
        return gx
```

Code Improvement

< problems >

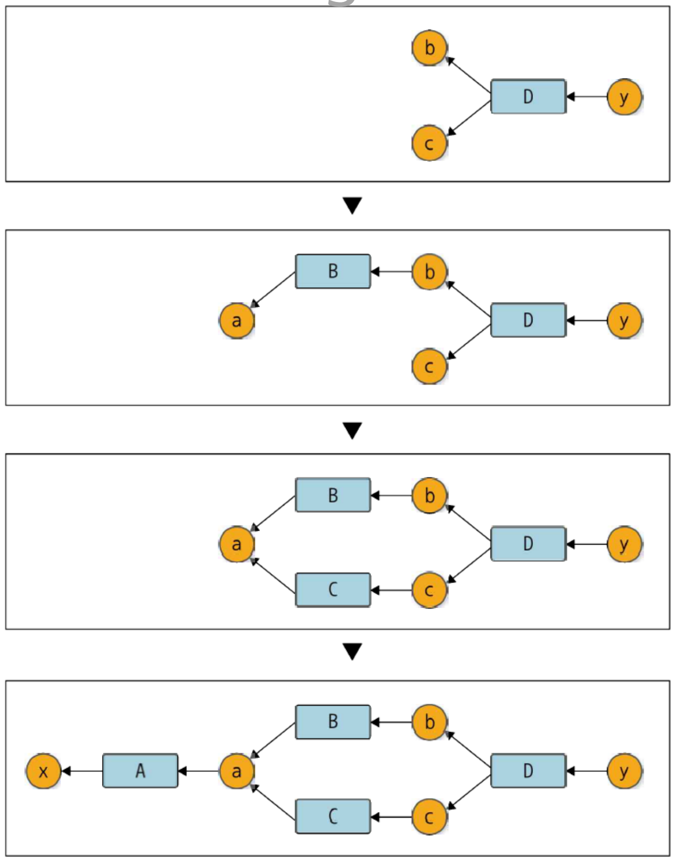
1. Single input & output
2. Reuse same variable
3. Order of calculation

wrong

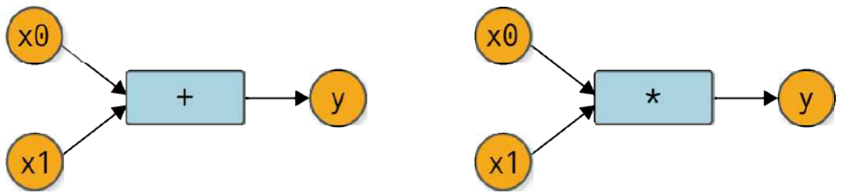


(3)

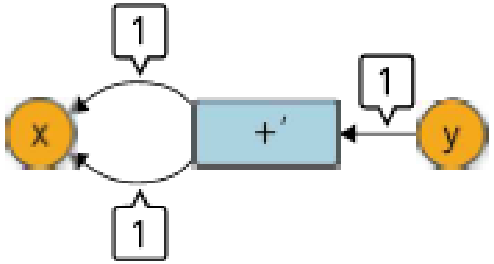
right



(1)



(2)



Code Improvement

```python
class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs] # multiple inputs
        ys = self.forward(*xs)
        if not isinstance(ys, tuple): # Tupleizating outputs
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        self.generation = max([x.generation for x in inputs]) # setting 'generation' of function
        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs
        return outputs if len(outputs) > 1 else outputs[0] # to return 1 or many outputs correctly

    def forward(self, xs):
        raise NotImplementedError()

    def backward(self, gys):
        raise NotImplementedError()


class Add(Function):
    # multiple intputs
    def forward(self, x0, x1):
        y = x0 + x1
        return y

    # multiple backward gradients
    def backward(self, gy):
        return gy, gy

def add(x0, x1):
    return Add()(x0, x1)


class Square(Function):
    def forward(self, x):
        y = x ** 2
        return y

    def backward(self, gy):
        x = self.inputs[0].data # to deal with tupleized single input
        gx = 2 * x * gy
        return gx
```

Tupleizating

All single elements

forward

2 → 1

backward

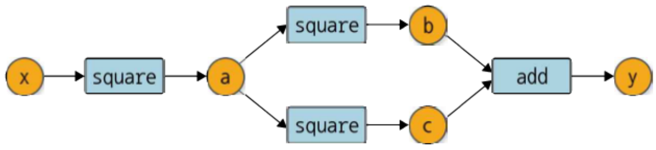
2 ← 1

```python
class Variable:
    def __init__(self, data):
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError('{} is not supported'.format(type(data)))

        self.data = data
        self.grad = None
        self.creator = None
        # use 'generation' variable to control the flow of backward gradients
        self.generation = 0

    def set_creator(self, func):
        self.creator = func
        # variable & function both have 'generation'
        self.generation = func.generation + 1

    def backward(self):
        if self.grad is None:
            self.grad = np.ones_like(self.data)

        # funcs & seen_set : space that functions will be stacked
        funcs = []
        seen_set = set() # set data type -> with no duplicates elements

        def add_func(f):
            if f not in seen_set:
                funcs.append(f)
                seen_set.add(f)
                funcs.sort(key = lambda x: x.generation)# sorting by generation
        add_func(self.creator)

        while funcs:
            f = funcs.pop() # taken out sequentially from the higher geneeration
            gys = [output.grad for output in f.outputs] # multiple gradients
            gxs = f.backward(*gys)
            if not isinstance(gxs, tuple):
                gxs = (gxs,)

            for x, gx in zip(f.inputs, gxs):
                # prevent overwriting of gradient when using the same variable instance
                if x.grad is None:
                    x.grad = gx
                else :
                    x.grad = x.grad + gx

                if x.creator is not None:
                    add_func(x.creator)

    # prevent accumalting of gradient when using the same variable instance
    def cleargrad(self):
        self.grad = None
```

ex) $y = add(x, x)$

$= 2x$

$\therefore \frac{dy}{dx} = 2$



$$y = (x^2)^2 + (x^2)^2 = 2x^4$$

```python
x = Variable(np.array(2.0))
a = square(x)
y = add(square(a), square(a))
y.backward()

print(y.data)
print(x.grad)
```

```
32.0
64.0
```