Neural Networks and Deep Learning Lecture 4

Wei WANG

cs5242@comp.nus.edu.sg

National University of Singapore

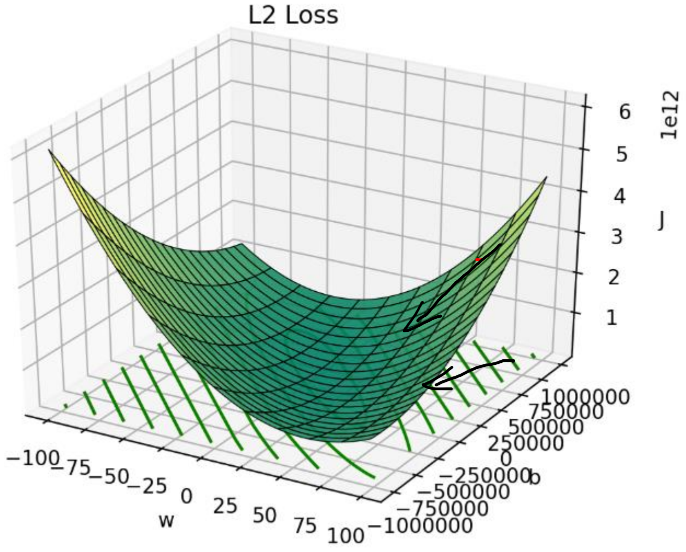
School of Computing

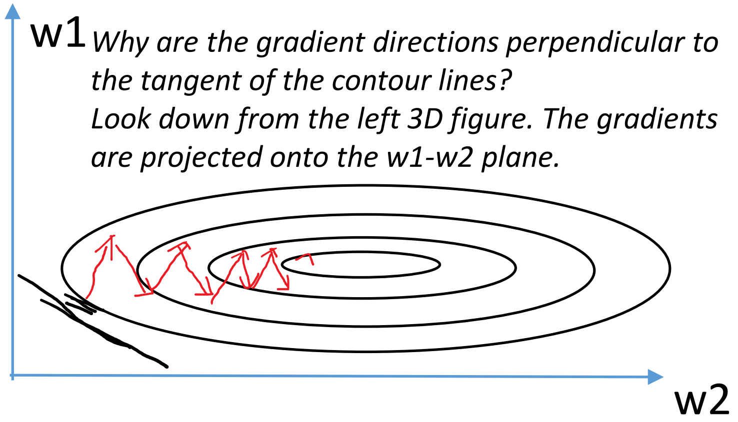
Administrative

- Refer to IVLE for the timeline.
- Assignment 1
- Final project
  - https://www.kaggle.com/t/bf0db238000f42e2bb010af37a3d5238
  - Register a Kaggle account using your nus email
  - Include group ID and nus ID (exxx) in your account name

Recap

Steepest gradient direction



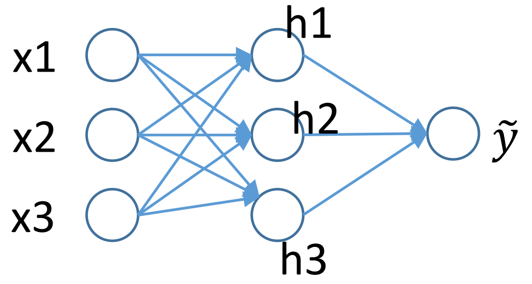


Why are the gradient directions perpendicular to the tangent of the contour lines?
Look down from the left 3D figure. The gradients are projected onto the w1-w2 plane.

single layer perceptron/ linear regression

Multilayer perceptron

$$\tilde{y} = W^T x + b, b \in R$$



- i-th layer consists of a **linear/affine** transformation function

$$z^{[i]} = a^{[i]}\left(h^{[i-1]}\right) = W^{[i]T} h^{[i-1]} + b^{[i]}$$

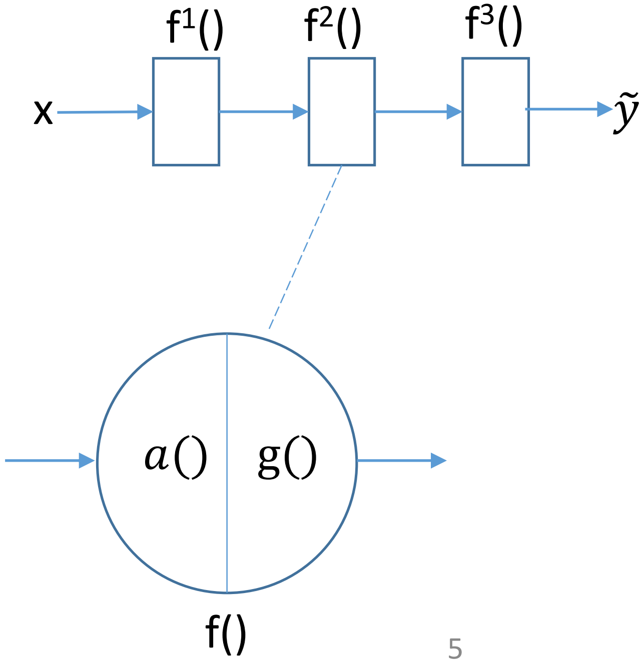
Different

$$W^{[i]} \in R^{d_{i-1} \times d_i}, b^i \in \boldsymbol{R}^{\boldsymbol{d_i}}$$

d_i is the number of hidden units at the i-th layer, which is a hyper-parameter to be tuned.

- followed by a **non-linear activation** function

$$h^{[i]} = g^{[i]}(z^{[i]}), \in R^{d_i}$$

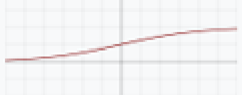
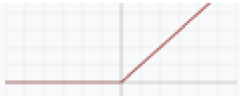


Why we need nonlinear activation:

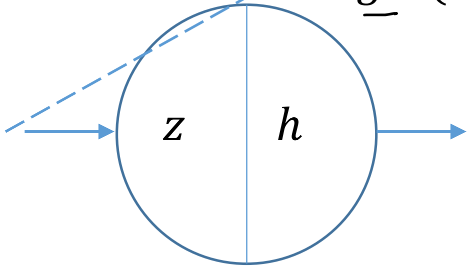
$$h^{[1]} = W^{[1]} x$$
$$h^{[2]} = W^{[2]} h^{[1]}$$
$$\cdots$$
$$h^{[k]} = W^{[k]} h^{[k-1]} = W^{[k]} W^{[k-1]} \ldots W^{[1]} x = \tilde{w}^T x$$

Activation functions

- Logistic (Sigmoid, σ) VS ReLU
  - Element-wise operation

Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = \overset{h_k}{f(x)}(1 - \overset{h_k}{f(x)})$	$(0, 1)$
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$

$h^{[i]} = g^{[i]}(z^{[i]})$



Logistic activation

If z_k is large, e.g. >10
Then h_k is near 1
If z_k is small, e.g. <-10
Then h_k is near 0

→ For both cases, $\frac{\partial h_k}{\partial z_k} \approx 0 \rightarrow \frac{\partial L}{\partial z_k} = \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial z_k} \approx 0$

→ gradients of $W_{:,k}$ (the k-th column, which contributes to the k-th element of z) are near 0, called gradient vanishing

ReLU activation

If z_k is positive, $\frac{\partial h_k}{\partial z_k} = 1$, no gradient vanishing

If z_k is negative, $\frac{\partial L}{\partial z_k} = \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial z_k} = 0$ gradients vanishing

Still better than Logistic as z_k has a larger working zone (domain).
Leaky ReLU resolves the gradient vanishing problem for negative z_k

(Binary) Cross-entropy loss

- Max log likelihood
  - logP$(correct|x) =$ log $P(predict = \text{Cat}|\ \text{x})$ = log $1 - \tilde{y}$ if y=0;

    log $\text{P}(predict = Dog|\ \text{x})$ = log $\tilde{y}$ if y=1;

    $= \text{ylog}\tilde{y} + (1 - y)\text{log}(1 - \tilde{y})$

→Minimize negative log likelihood

$L(x, y) = -\text{ylog}\tilde{y} - (1 - y)\text{log}(1 - \tilde{y})$

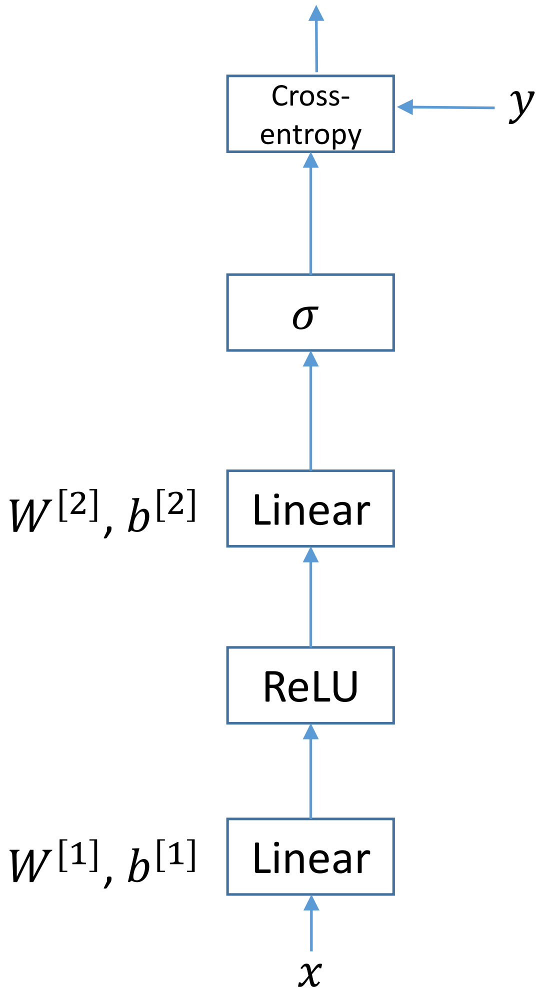
$\text{J}(X, Y) = -\frac{1}{n} Ylog\tilde{Y} - (1 - Y)\text{log}(1 - \tilde{Y})$

Back-propagation (Layer API)

```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```

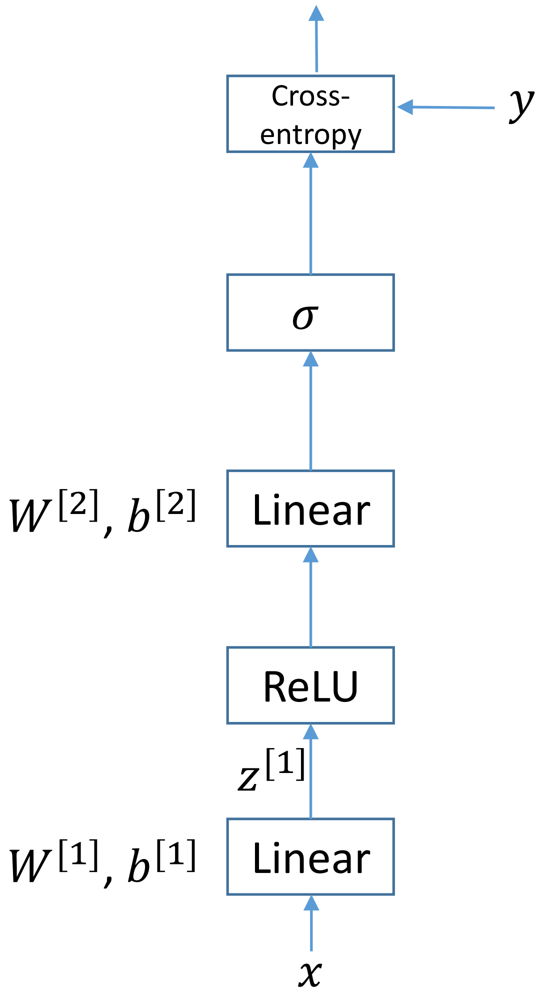
The gradients of W and b can be stored in the layer or returned together with the gradient of x.

The forward and backward must consider x and dy for a mini-batch of samples, i.e. the first dimension of x and dy should be the batch index



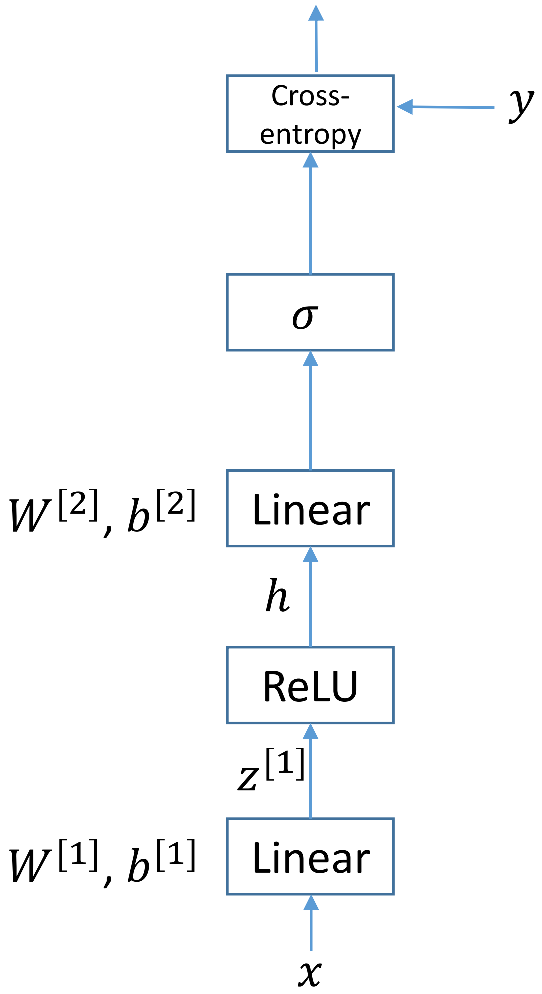
Back-propagation (Layer API)

```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



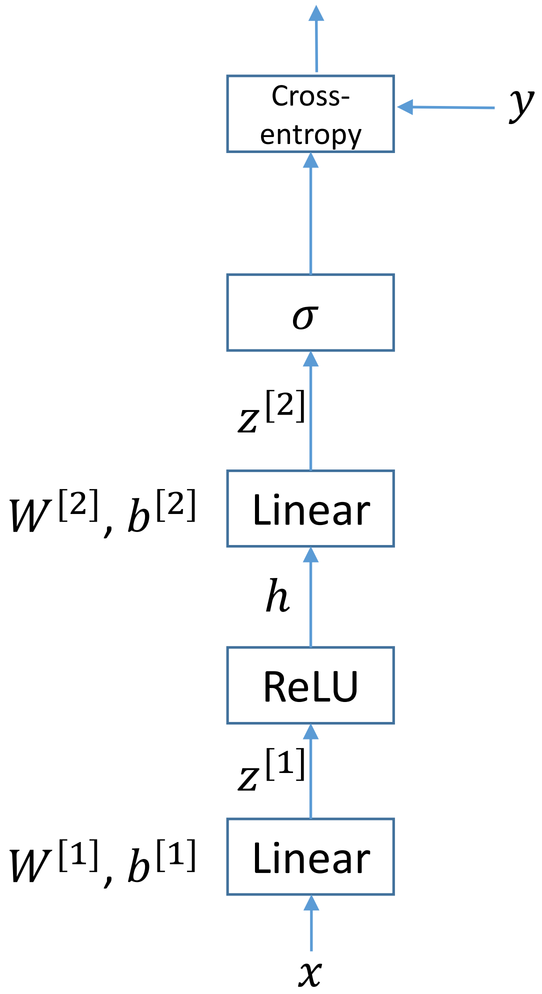
Back-propagation (Layer API)

```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



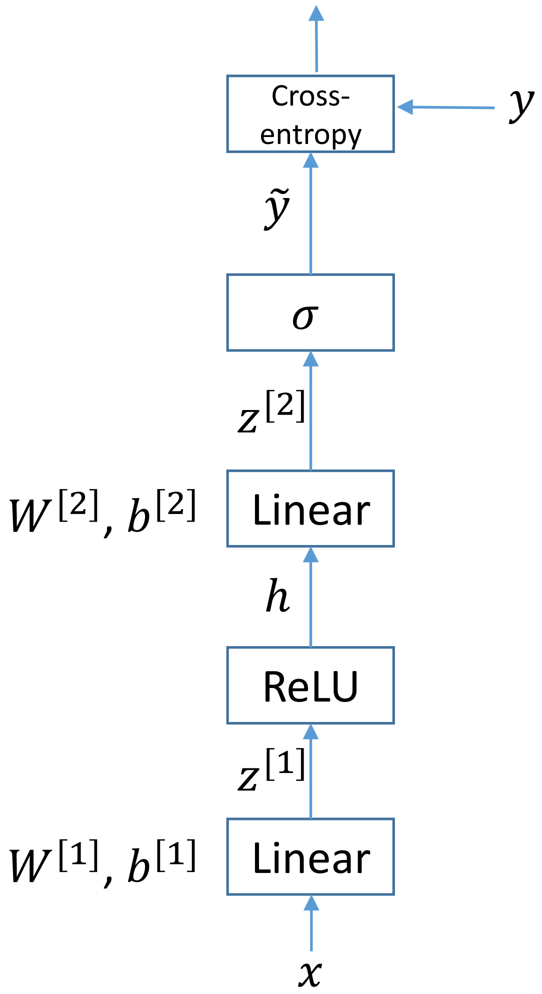
Back-propagation (Layer API)

```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



Back-propagation (Layer API)

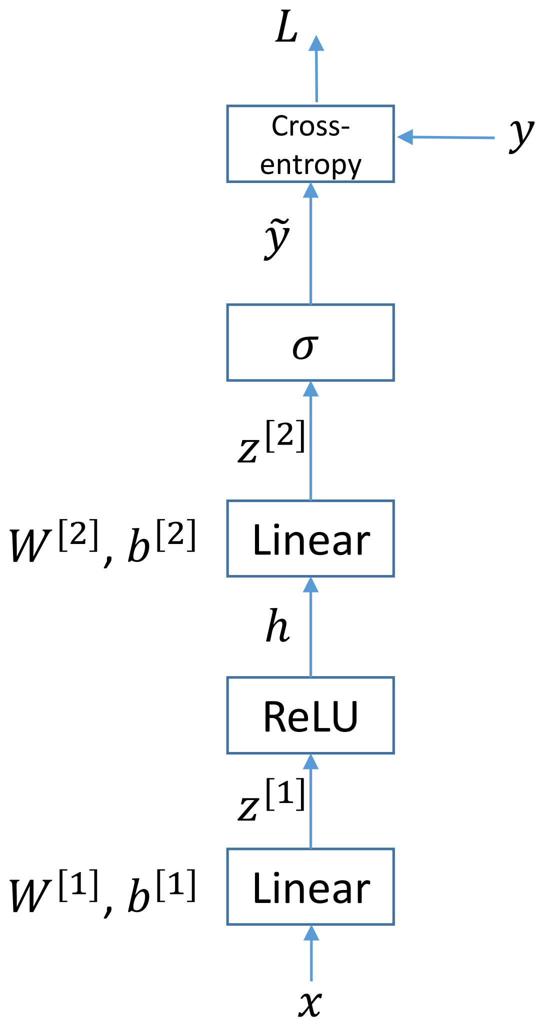
```
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



Back-propagation (Layer API)

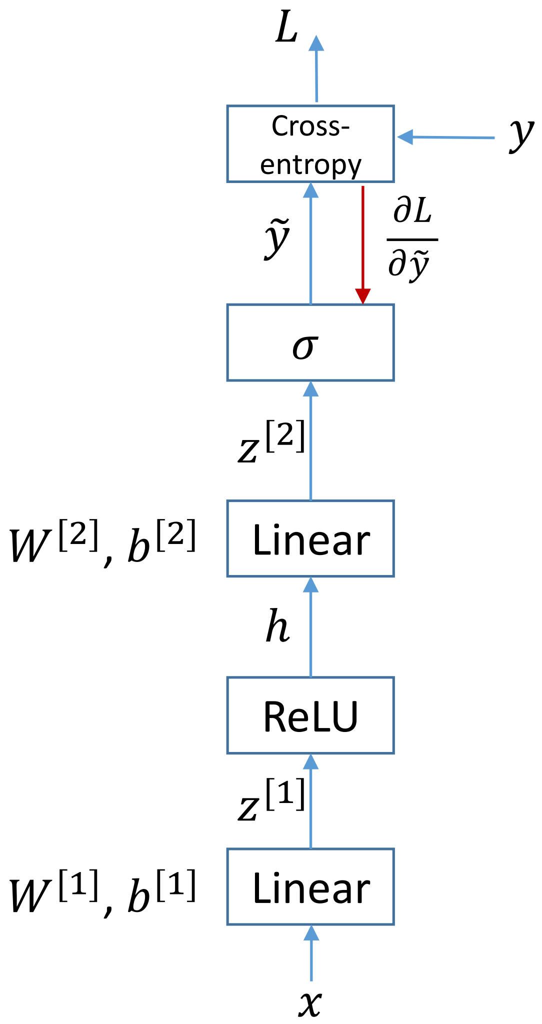
-

```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



Back-propagation (Layer API)

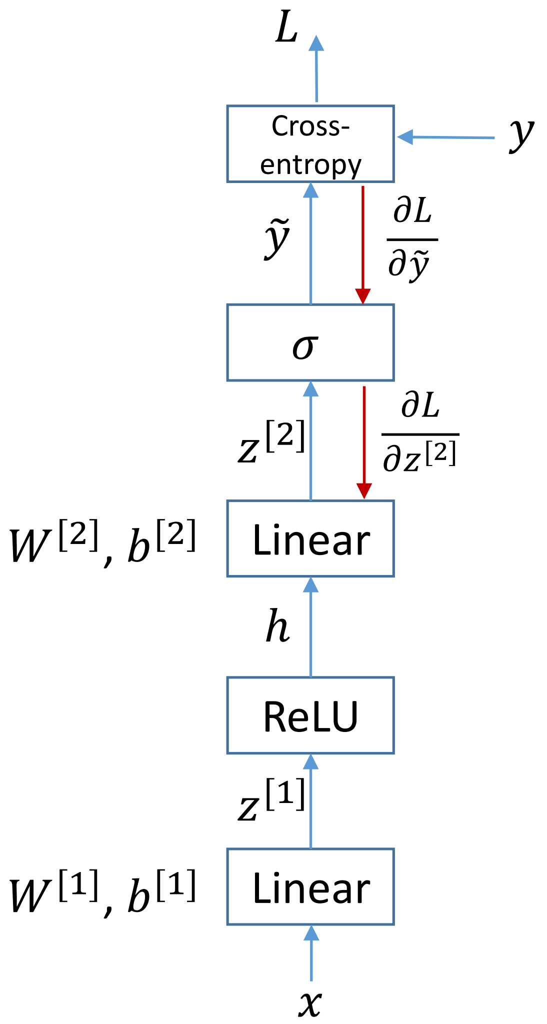
```
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



Back-propagation (Layer API)

-

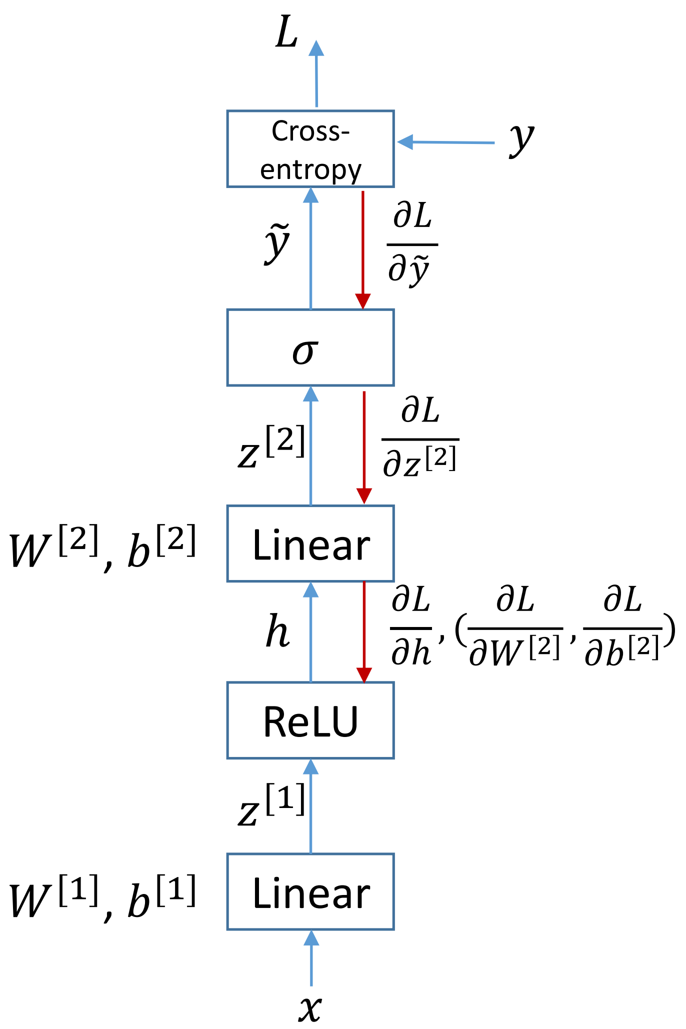
```
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



Back-propagation (Layer API)

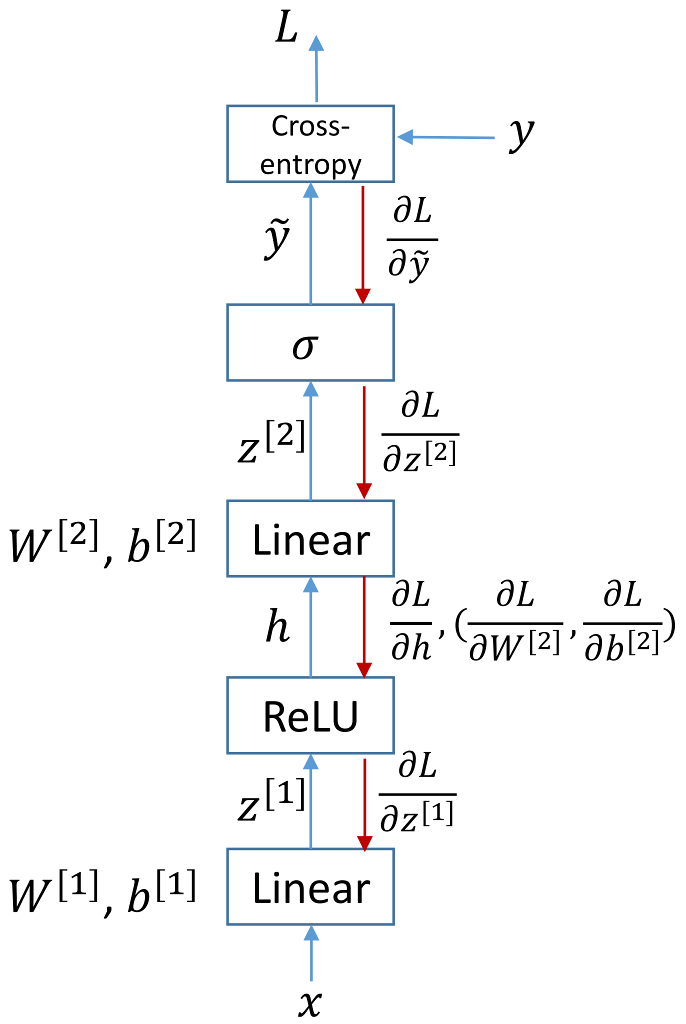
-

```
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



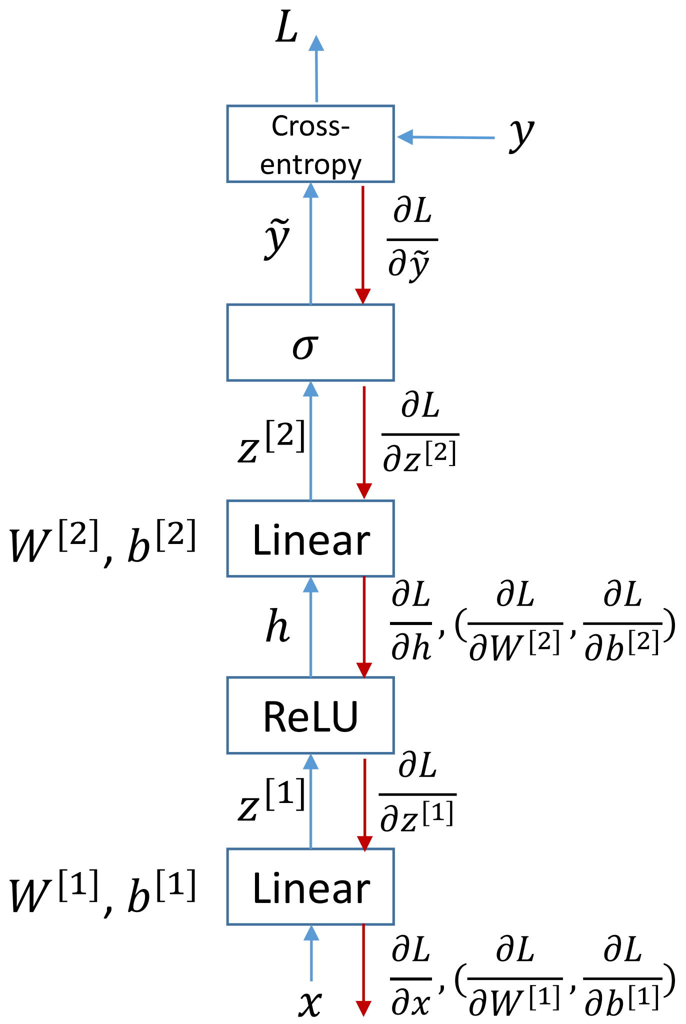
Back-propagation (Layer API)

```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



Back-propagation (Layer API)

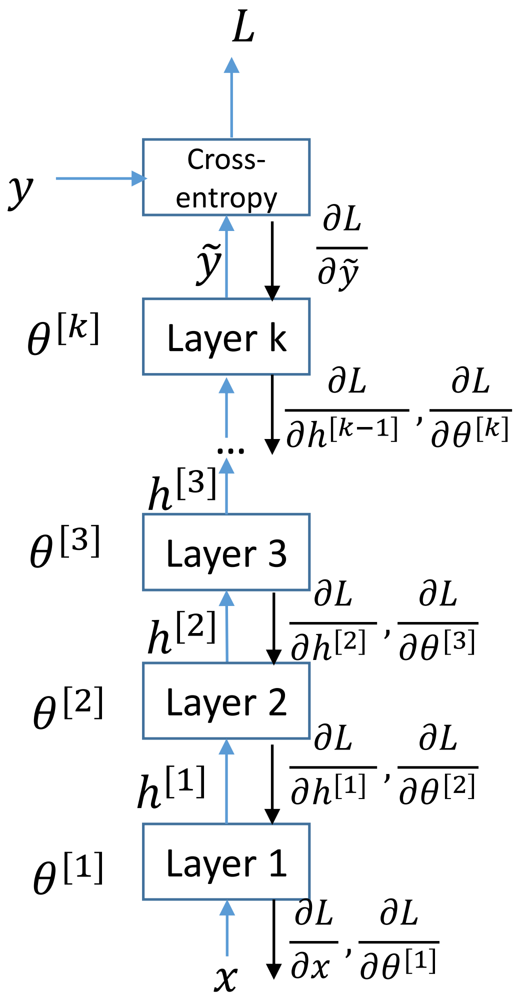
```python
class Layer(object):
    def __init__(self, name):
        self.name = name

    def forward(self, x, args=None):
        # return y, the output of this layer
        pass

    def backward(self, dy, args=None):
        # return gradients of the input x and parameters.
        pass
```



From shallow to deep

Layer i could be

1. A linear layer with parameters $\theta^{[i]} = \{W^{[i]}, b^{[i]}\}$
2. A ReLU layer, with $\theta^{[i]} = \emptyset$
3. A Logistic layer, with $\theta^{[i]} = \emptyset$
4. A convolution layer, pooling layer, etc. (to be introduced)



Universal Approximate Theory

1. MLP has great capacity
2. Difficult to optimize

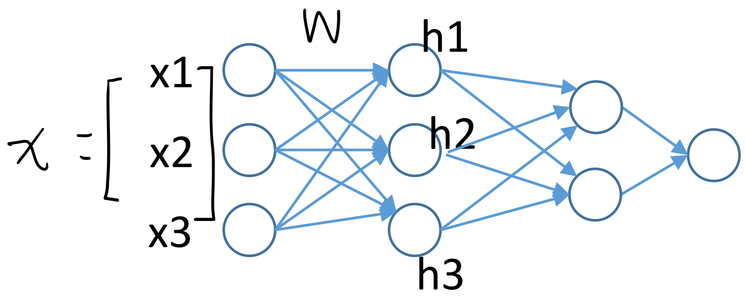
Training tricks for deep neural networks

Random parameter initialization

- All elements of W are the same (e.g. 0 or 1)
- → all hidden units are the same
- → derivatives of all hidden units are the same
- → derivatives of all columns of W are the same
- → W are updated in the same direction and length

? problems

- Repeat



1. If all neurons in one layer are the same, then only one neuron is enough and all others are redundant → a very simple model

2. W's elements are always the same → redundant parameters

Random parameter initialization

- Weight matrix (W) http://cs231n.github.io/neural-networks-2/#init
  - Gaussian, N(0, 0.01)
    - Too small variance → gradient vanishing
    - Too large variance → gradient exploding
  - Uniform, U(-0.05, 0.05)
  - Glorot/Xavier [20]
    - **Gaussian N(0, sqrt(2/(fan_in + fan_out))**
  - He/MSRA [21]
    - **Gaussian N(sqrt(2/fan_in))**
- Bias vector
  - **0**

```
W = np.random.rand(nb_y, nb_x) * math.sqrt(2.0/(nb_y + nb_x))
W = np.random.rand(nb_y, nb_x) * math.sqrt(2.0/nb_x)
```

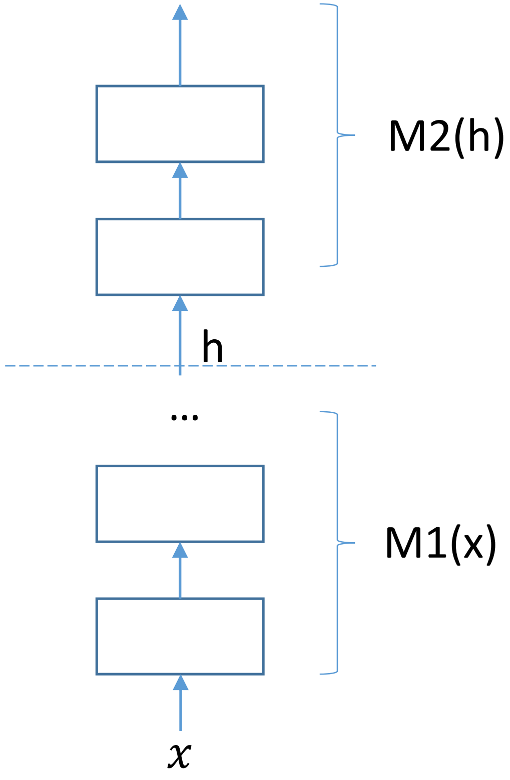
$$h^{[i-1]} \leftarrow g(z^{[i-1]})$$
$$z^{[i]} = {W^{[i]}}^T h^{[i-1]}$$

$$\frac{\partial J}{\partial h^{[i-1]}} \leftarrow W^{[i]} \frac{\partial J}{\partial z^{[i-1]}}$$
$$\frac{\partial J}{\partial z^{[i-1]}} \leftarrow \frac{\partial J}{\partial h^{[i-1]}}$$
$$\frac{\partial J}{\partial W^{[i-1]}} \leftarrow h^{[i-2]} \frac{\partial J}{\partial z^{[i-1]}}$$

Batch normalization

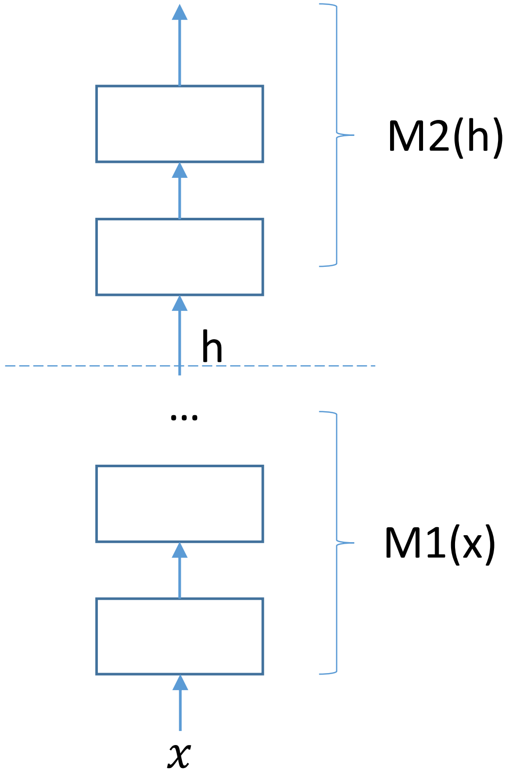
- Intuition 1
  - Train a model over dataset D
  - If D's distribution shifts (e.g. by adding new data samples)
  - The model should be updated to get good performance

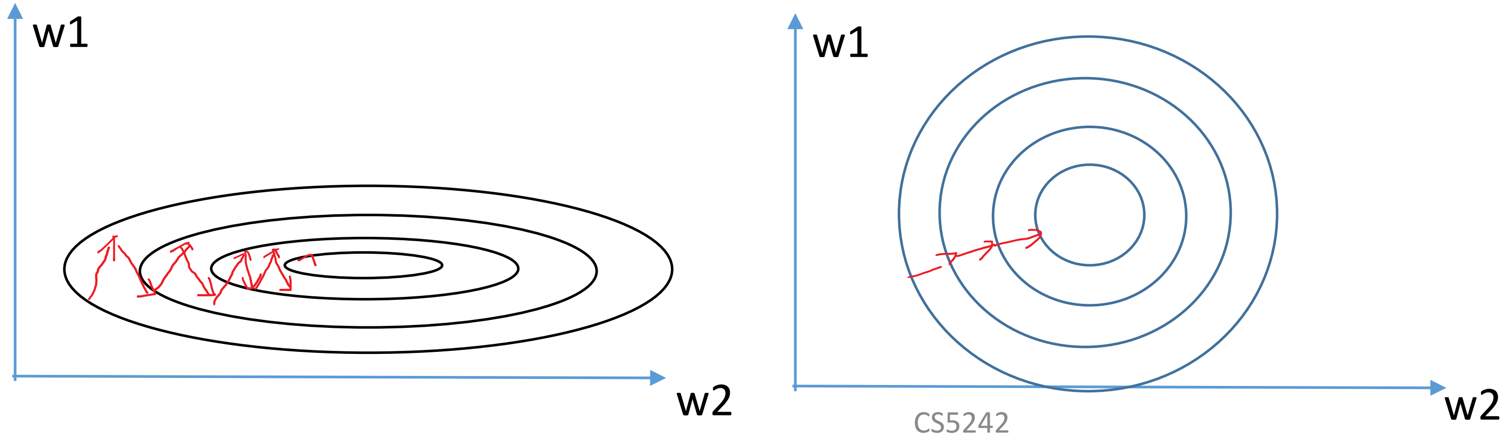
  - M1(x)'s output is M2(h)'s input
  - The distribution of h is keeping changing as parameters of M1() are updated.
  - Difficult to optimize M2 → covariate shift



Batch normalization

- Intuition 2
  - Normalize input features x (e.g. standardization)
    - → good loss contour for M1
  - But h is not normalized → ellipse loss contour for M2
  - We need to normalize M2





Batch normalization during training

- Normalization per unit across all samples in one mini-batch
  - Applied **after linear transformation and before activation**
  - $\widehat{z_k} = \frac{z_k - E[z_k]}{\sqrt{var[z_k]}}$, $\overline{z_k} = \gamma_k \widehat{z_k} + \beta_k$, k enumerates over all units of the layer
  - γ_k and β_k are parameters to be learned like weight matrix and bias
  - $E[z_k]$ and $var[z_k]$ are computed over one mini-batch samples

one mini-batch

$$\begin{cases} x^{(1)} \\ x^{(2)} \\ \dots \\ x^{(n)} \end{cases} \rightarrow \begin{cases} z^{(1)} \\ z^{(2)} \\ \dots \\ z^{(n)} \end{cases} \qquad \begin{cases} \mu_k = E[z_k] = \frac{z_k^{(1)} + z_k^{(2)} + \ldots + z_k^{(n)}}{n} \\ \\ \sigma_k = Var[z_k] \end{cases}$$

- Converge faster

Batch normalization during test

- Applied per testing sample
- $\widehat{z_k} = \frac{z_k - E[z_k]}{\sqrt{var[z_k]}}$, $\overline{z_k} = \gamma_k \widehat{z_k} + \beta_k$
- $E[z_k]$ and $var[z_k]$ are accumulated during training by exponential averaging

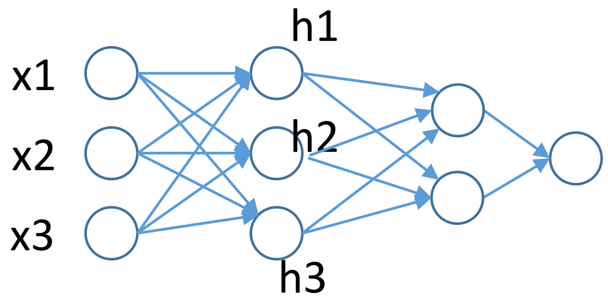
μ_k: accumulated mean

$$\mu_k = (1 - \beta)\mu_k + \beta E[z_k]$$

$E[z_k]$ is the expectation from the current training iteration

Dropout

- Training
  - Randomly set some neurons to 0 with probability p (0.5, 0.4, 1/3 etc.)
  - Multiple the outputs (h) with scale 1/(1-p) ?
  - Different layers may have different dropout rate
- During inference
  - Do nothing



without dropout/testing: $h_1 \, , h_2 \, , h_3 \, \rightarrow \, z_1$

p=1/3

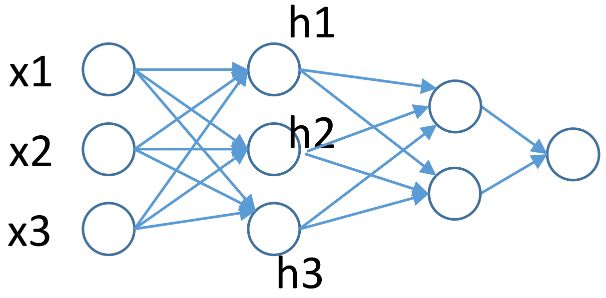
training with dropout: $h_1 \, , h_3 \, \rightarrow \, \bar{z_1}$

$\bar{z_1} \, < \, z_1$ training and test are quite different

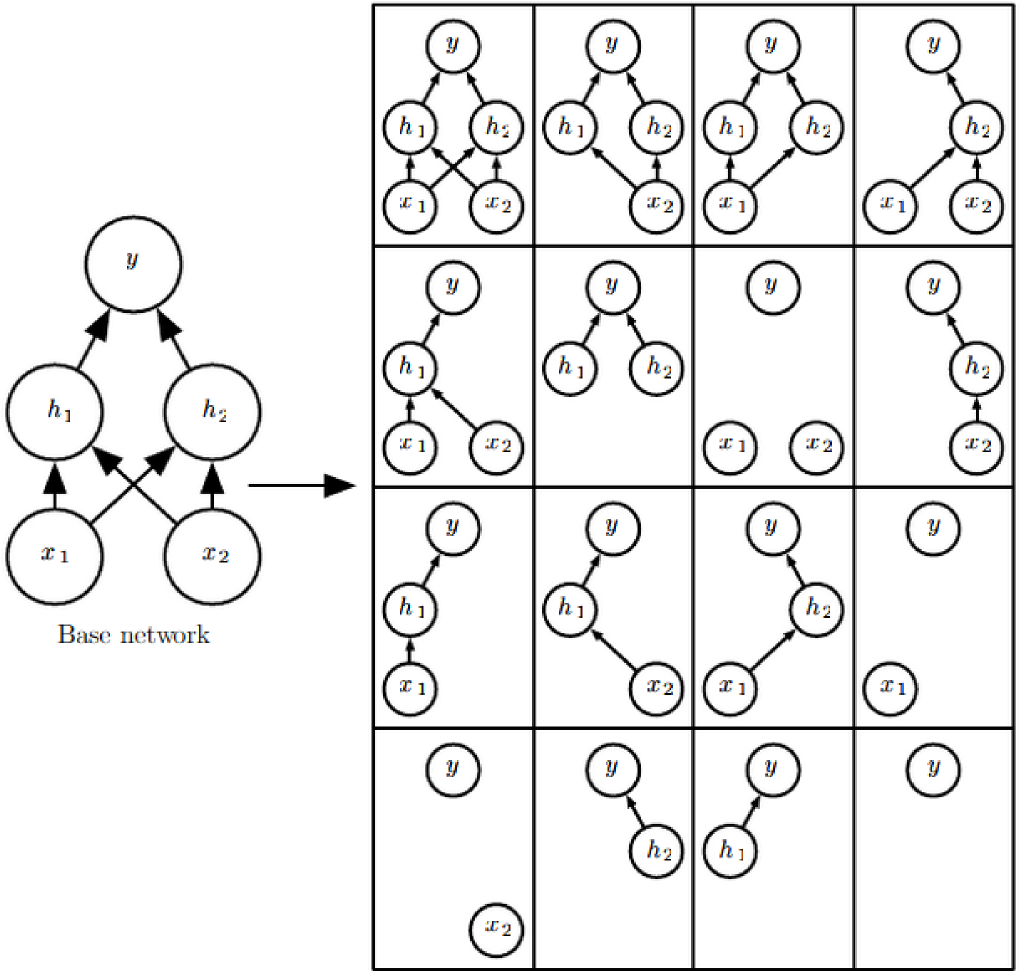
Rescale by 3/2

Dropout

- Intuition 1
  - Regularization
  - Similar to L2 norm
- Intuition 2
  - Ensemble modeling
  - $Pensemble(y|x) = \sum_z P(z)P(y|x,z)$

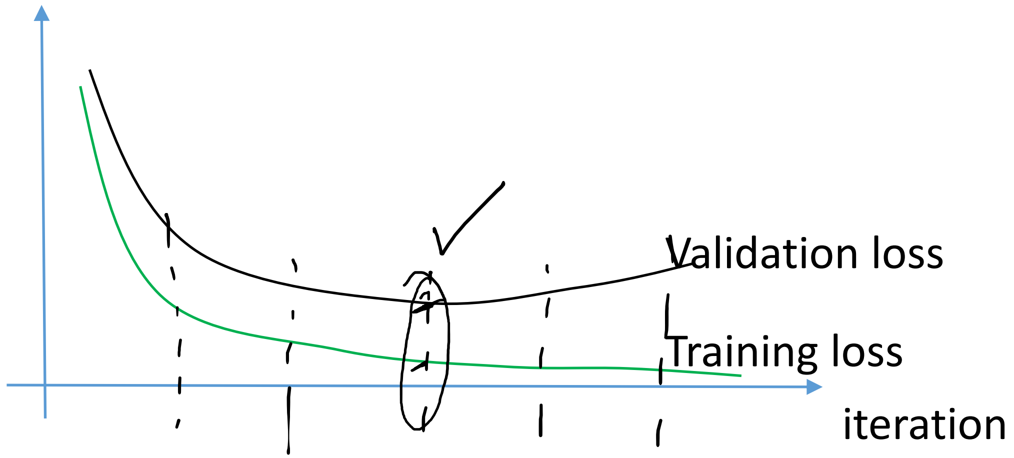


If adding dropout after the hidden layer h, then there are 2^3 different dropout cases, 2^3 different networks for assemble.



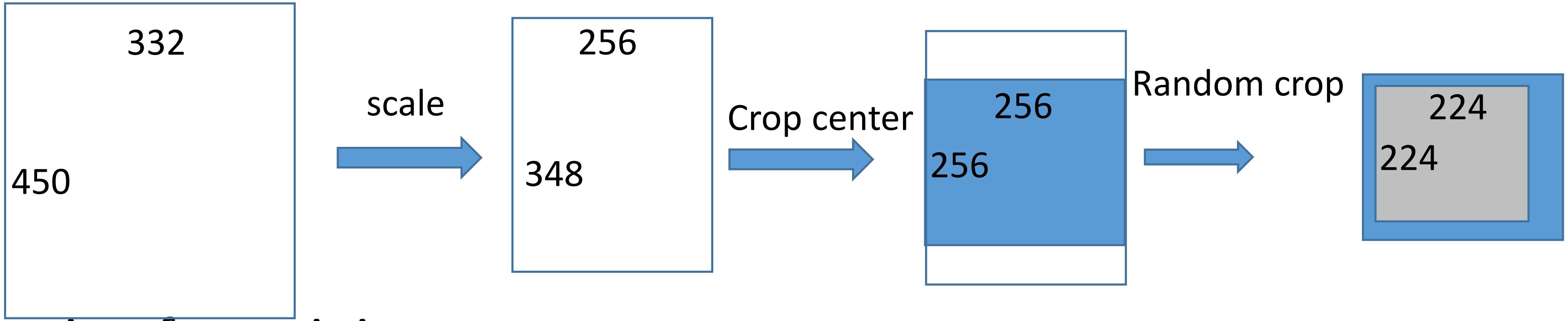
Source from:
http://www.deeplearningbook.org/contents/regularization.html

Regularization --- early stopping



Dataset augmentation

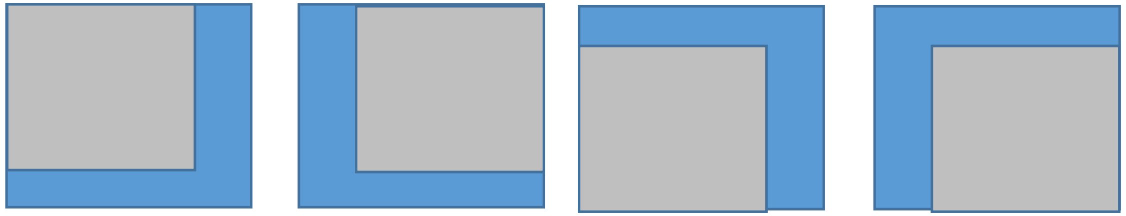
- Increase the training dataset to prevent overfitting
  - Scaling
  - Cropping
  - Flipping
  - Intensity
  - Rotation



- Random operation for training
  - Sample rotation angle from a range, e.g. [-15, 15]
  - Crop at random position (offsets)
  - Resize to random size (and then crop)
  - Etc.

Test time augmentation

- No random operations
- Crop
  - Training time: random crop (position)
  - Test time: crop at fixed position



- Rotation
  - Training time: random angle from [-15, 15]
  - Test time: fixed angles, 15, 0 and -15
- Resize
  - Training time: random size, e.g. between [224, 384]
  - Test time: fix sizes, [224, 256, 288, 384]
- Make predictions by aggregating the results from all augmented images.

Convolutional neural network (CNN)

From MLP to CNN

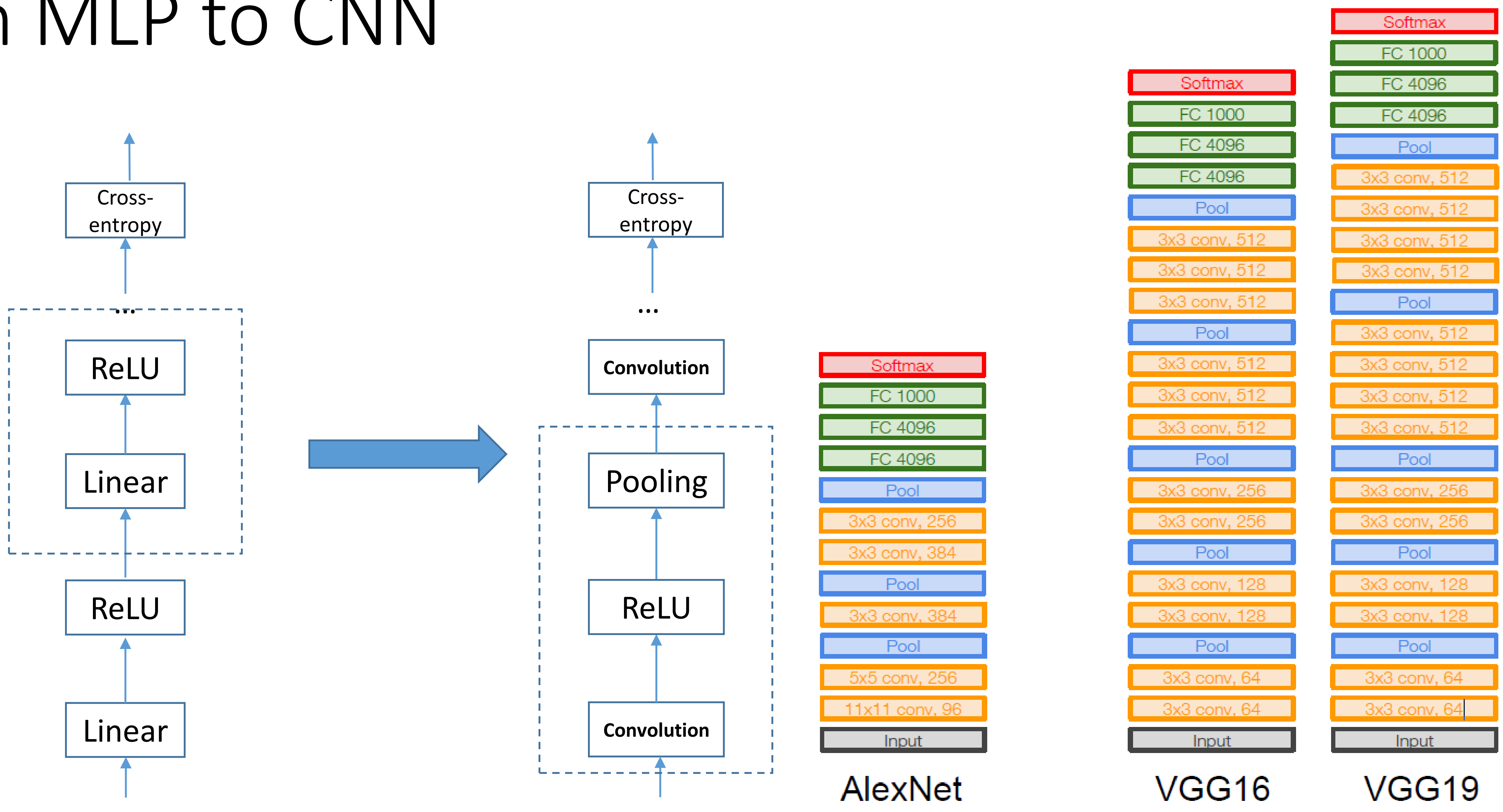
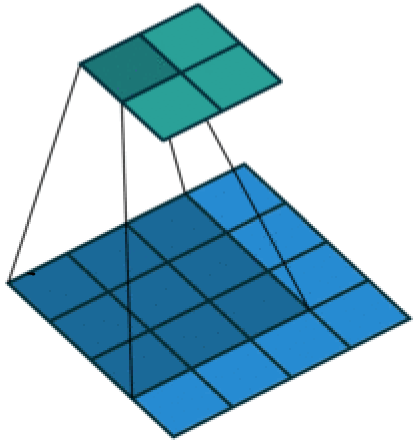
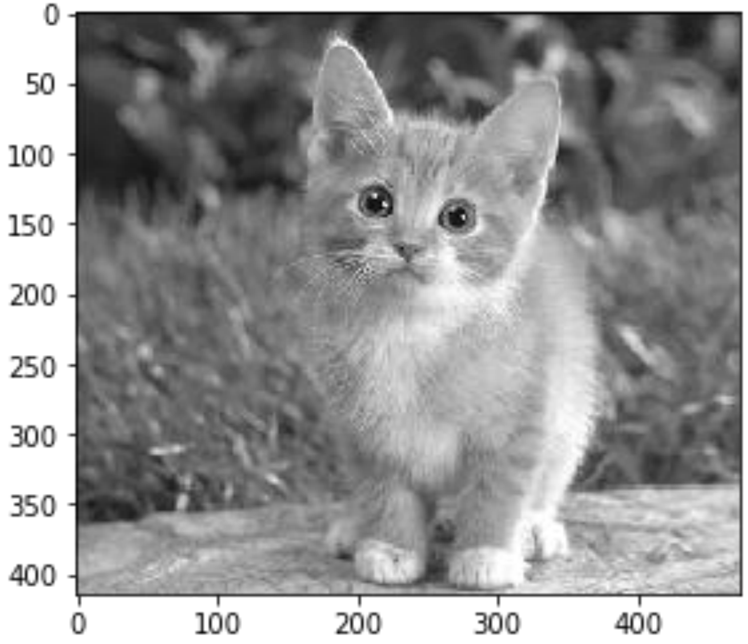


Image source: Stanford cs231n

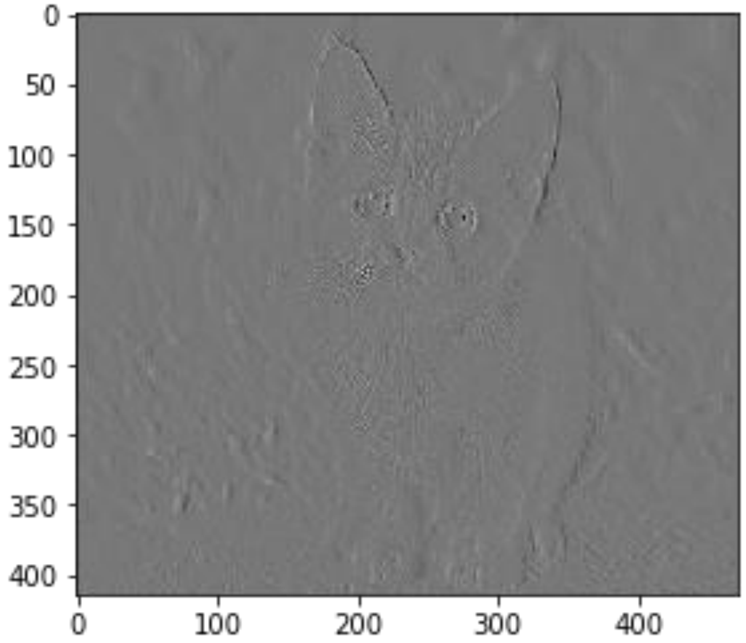
Convolution

http://setosa.io/ev/image-kernels/

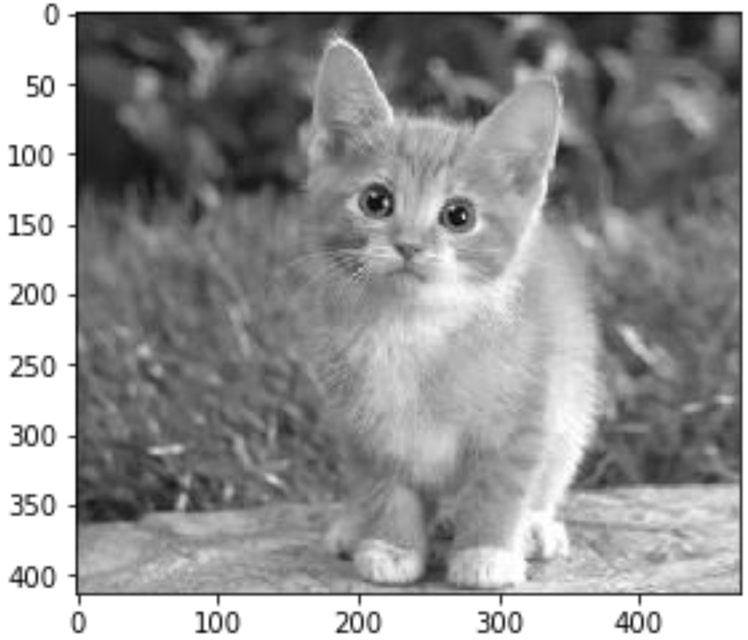




$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$



Why CNN is better than MLP?

- $W^i \in R^{|\boldsymbol{h}^{i-1}| \times |\boldsymbol{h}^i|}$
  - 2500x2000=5,000,000

Perceptron

Perceptron is too simple → underfitting→ add more layers → MLP

MLP

MLP has too many parameters → High dimension → difficult to optimize and overfitting → CNN (with more regularization)

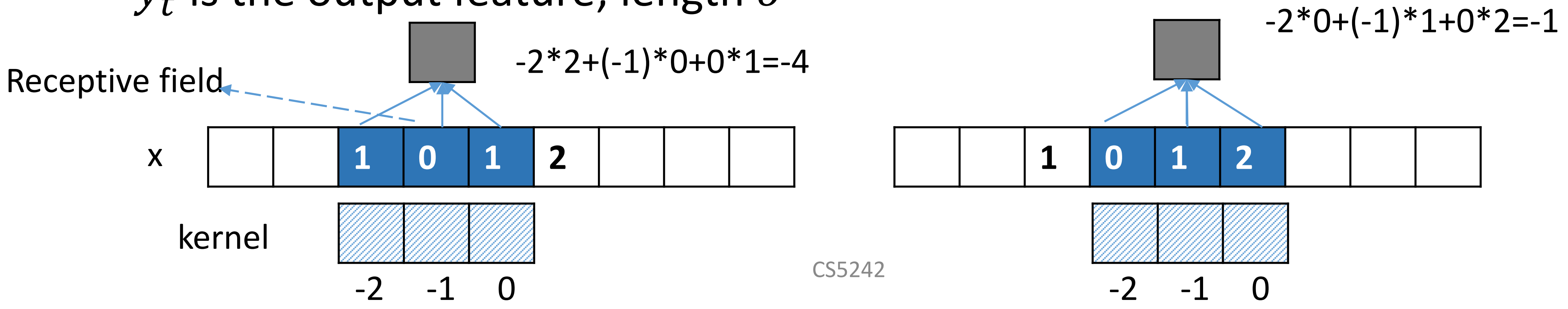
CNN

NN architecture	Dataset	Distortions	Test Error [%]
MLP:2500-2000-1500-1000-500-10	MNIST	no	1.47
MLP:2000-2000-2000-2000-2000-2000-2000-10	MNIST	no	1.531 ± 0.051
MLP:1500-1500-1500-1500-1500-1500-1500-10	MNIST	no	1.513 ± 0.052
MLP:1000-1000-1000-1000-1000-1000-1000-1000-1000-10	MNIST	no	1.628 ± 0.035
MLP:1000-1000-1000-1000-1000-1000-1000-1000-10	MNIST	no	1.542 ± 0.052
MLP:1000-1000-1000-1000-1000-1000-1000-10	MNIST	no	1.517 ± 0.069
MLP:1000-1000-1000-1000-1000-1000-10	MNIST	no	1.529 ± 0.078
MLP:1000-1000-1000-1000-1000-10	MNIST	no	1.571 ± 0.046
MLP:1000-1000-1000-1000-10	MNIST	no	1.549 ± 0.038
MLP:1000-1000-1000-10	MNIST	no	1.650 ± 0.030
MLP:500-500-500-500-500-500-500-10	MNIST	no	1.744 ± 0.038
MLP:500-500-500-500-500-500-10	MNIST	no	1.702 ± 0.064
MLP:500-500-500-500-500-10	MNIST	no	1.719 ± 0.069
MLP:500-500-500-500-10	MNIST	no	1.728 ± 0.028
MLP:500-500-500-10	MNIST	no	1.765±0.040
MLP:2000-1500-1000-500-10	MNIST	5% translation	0.94
MLP:2500-2000-1500-1000-500-10	MNIST	affine + elastic	0.35
MLP committee:2500-2000-1500-1000-500-10	MNIST	affine + elastic	0.31
CNN 20M-40M-60M-80M-100M-120M-150N	MNIST	affine + elastic	0.35

Source from: http://people.idsia.ch/~ciresan/results.htm

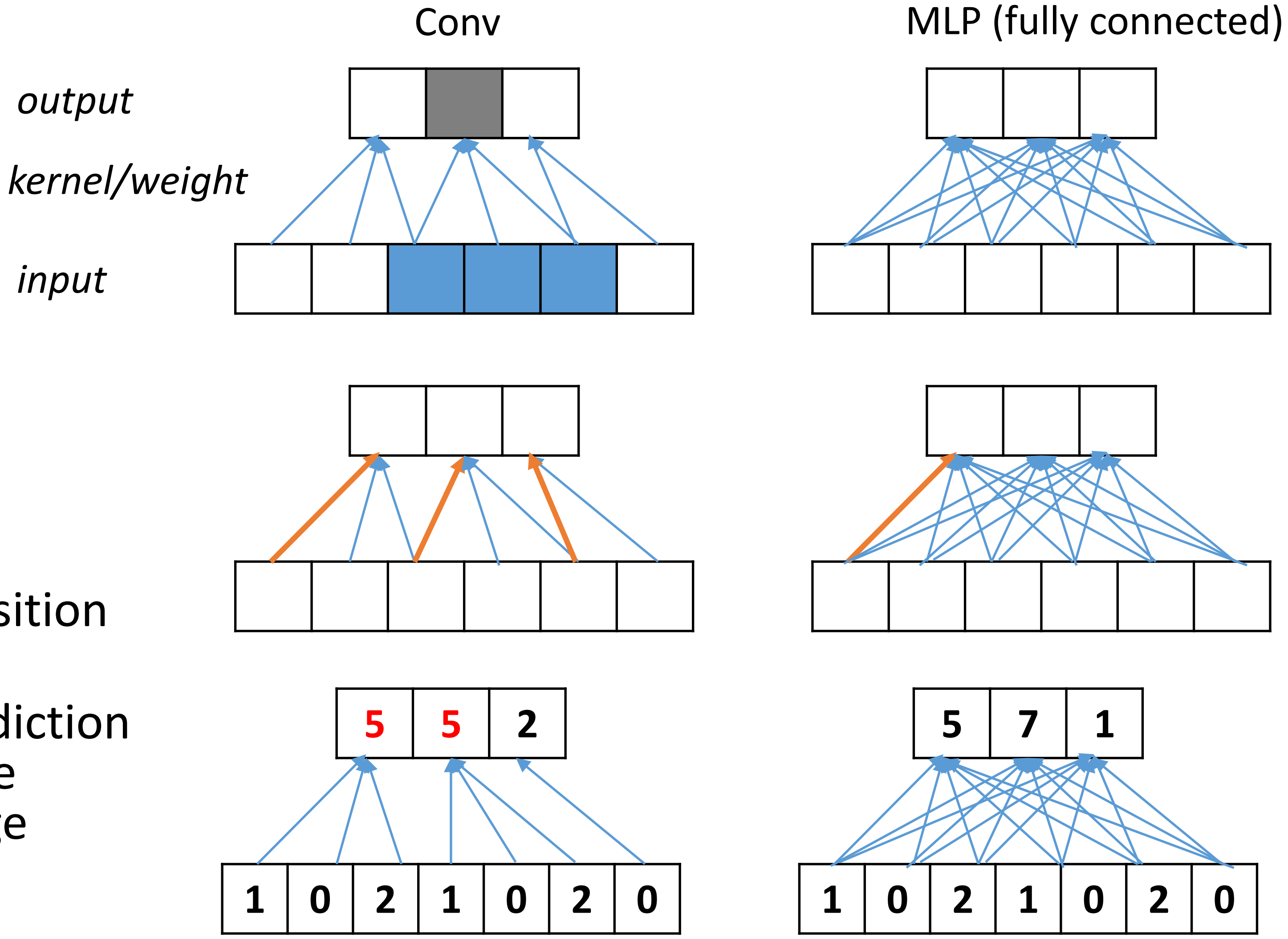
Convolution and Cross-Correlation

- Cross-correlation (https://en.wikipedia.org/wiki/Cross-correlation)
  - $y_t = \sum_{i=0}^{k-1} w_i \times x_{t+i}$
  - **In CNN, convolution refers to cross-correlation**
  - w is called kernel/filter; the parameters to be trained; length k
  - x is the input; length l
  - the input area, i.e. t-(k-1),…, t-1, t is called the receptive field
    - One receptive field generates one output value
  - y_t is the output feature; length o

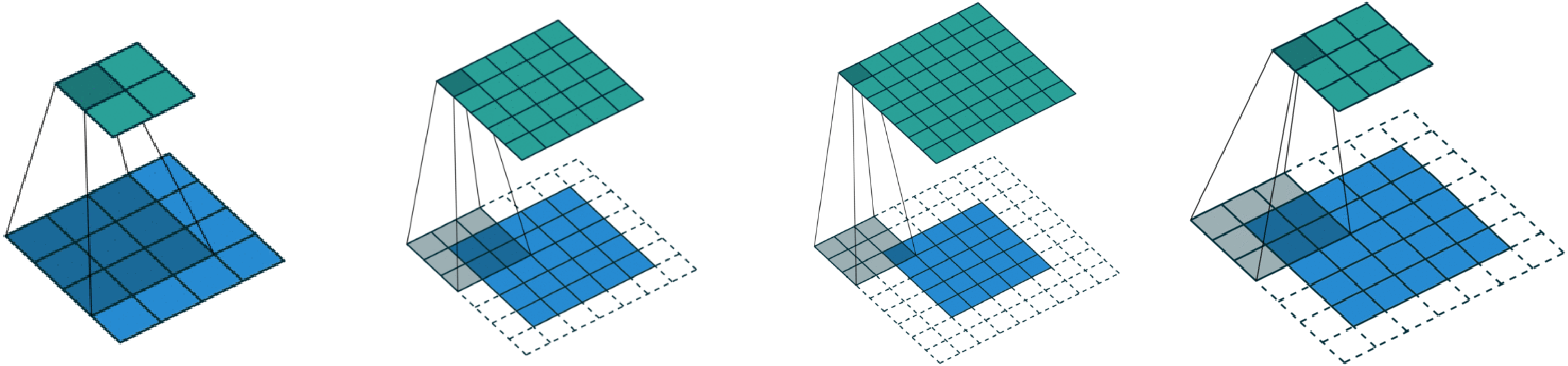


Properties (Why Convolution better?)

- Sparse connection
  - Fewer parameters
  - Less overfitting
- Weight sharing
  - Regularization
  - Less overfitting
- Location invariant
  - Robust to object position in the image
  - Make the same prediction no matter where the object is in the image



2D Convolution



k=3, p=0, s=1 (Valid) k=3, p=2, s=1 (Same) k=3, p=4, s=1 (Full) k=3, p=2, s=2

Source: http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html