Deep Neural Networks

Chia-Chi Tsai (蔡家齊)
cctsai@gs.ncku.edu.tw
AI System Lab
Department of Electrical Engineering
National Cheng Kung University

Outline

- Deep Feedforward Networks
- Regularization
- Optimization
- Convolutional Neural Networks
- Practical Methods

Outline

- Deep Feedforward Networks
- Regularization
- Optimization
- Convolutional Neural Networks
- Practical Methods

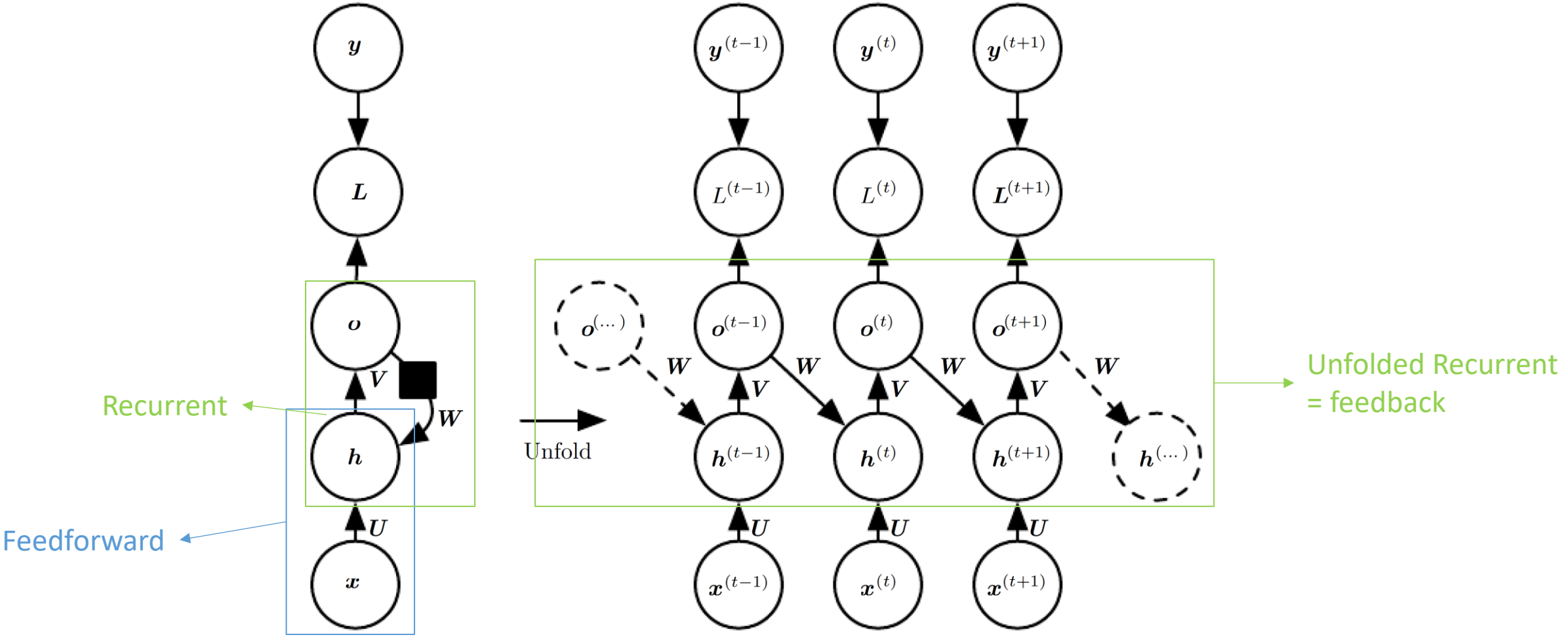
Deep Feedforward Networks

- Deep feedforward networks = feedforward neural networks = multilayer perceptrons (MLP)
  - Most influential Deep learning Models
  - Goal – approximate some function f^* by
    - Only **feedforward** connections
    - Information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.
- A classifier Example - $f^*(x)$ maps an input x to a category y
  - A feed forward network define a mapping function f

$$y = f(x; \theta)$$

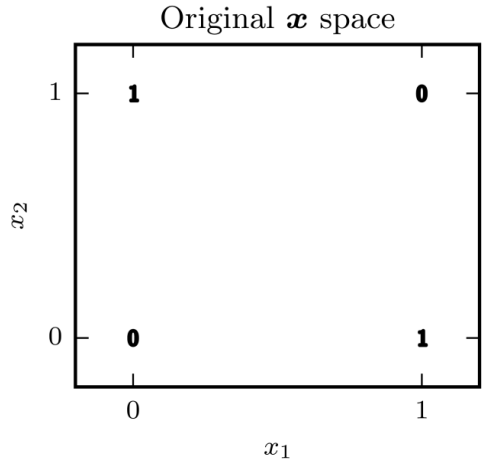
  - Goal: Learns the value of the parameters θ that result in the best function approximation $f(x; \theta) \cong f^*(x)$

Feed-forward v.s. Feed-back



A Simple Example: Learning XOR

- Problems
  - Input Data: $\mathbb{X} = \{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$
  - Target function: $\mathrm{y} = f^*(\mathbb{X}) = \{0,1,1,0\}$
- We will not be concerned with statistical generalization but perform correctly on the four point of $\mathbb{X}$.
- We will train the network on all four of these points $\mathbb{X}$. The only challenge is to fit the training set



Starting from Linear Models

- Linear model

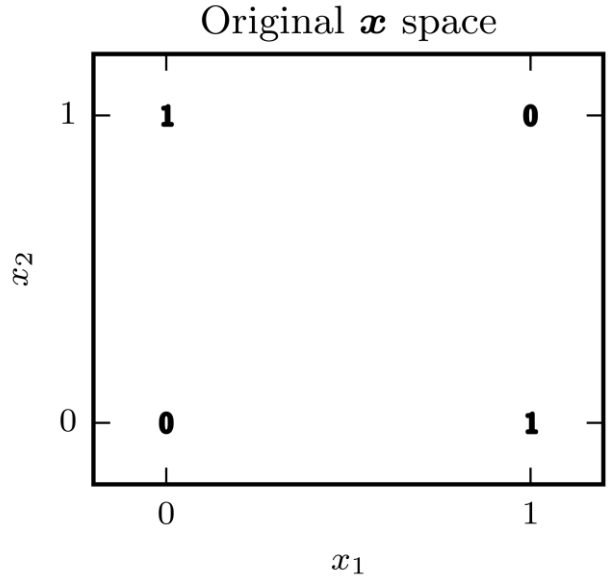
$$y = f(x; \theta = \{w, b\}) = x^T w + b$$

- Loss function - MSE

$$J(\theta) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(x) - f(x, \theta))^2$$

- Minimize $J(\theta)$ in closed form with respect to w and b using the normal equations
  - We can get $w = 0$ and $b = \frac{1}{2}$
  - Result in $y = \frac{1}{2}$ on every input sample

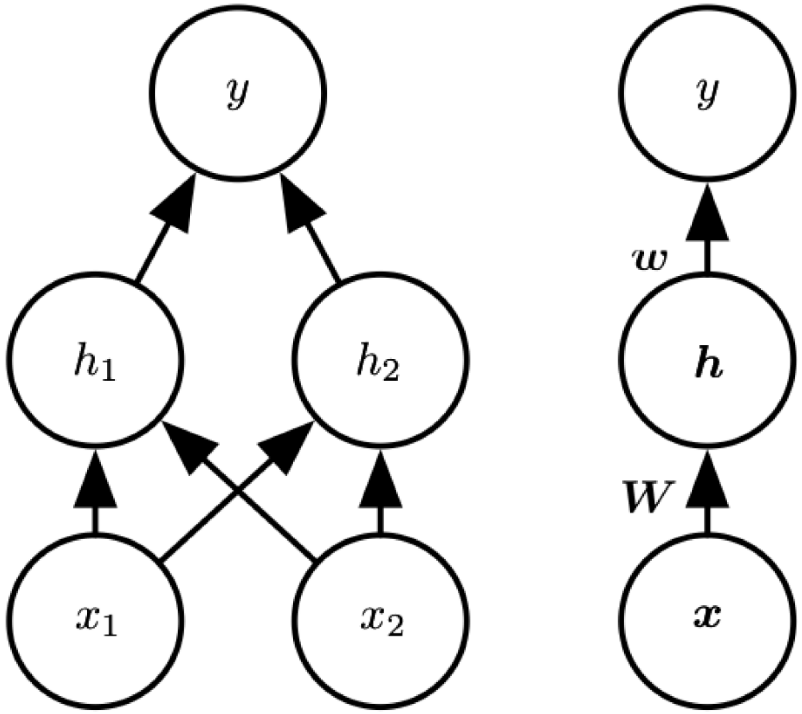
- XOR is not linearly separable
  - Linear model is **NOT** able to represent XOR function



Learning from Different Feature Space

- Use one hidden layer containing two hidden units to learn ϕ

$$y = f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x))$$



$$y = f^{(2)}(h; w, b) = f^{(2)}(f^{(1)}(x))$$

$$h = \phi(x) = f^{(1)}(x; W, c)$$

$$x$$

Affine Transformation

- Mapping input to a different feature space

$$\mathrm{h} = \phi(x) = w^T x + b$$

- Use kernel functions such as radial basis functions (RBFs), e.g.:

$$\phi(x) = e^{-(\epsilon\|x - x_i\|)^2}$$

- Manually engineer ϕ, that is, features in computer vision, speech recognition, etc.
- To learn ϕ from data:

$$y = f(x; \theta, w) = \phi(x; \theta)^T w$$

From Linear to Nonlinear

- To extend linear models to represent nonlinear function of x
  - Most neural networks do so using an affine transformation controlled by learned parameters, followed by a **fixed, nonlinear function** called an **activation function**

$$\mathrm{h} = \phi(x) = w^T x + b \rightarrow h = g\big(\phi(x)\big) = g(w^T x + b)$$

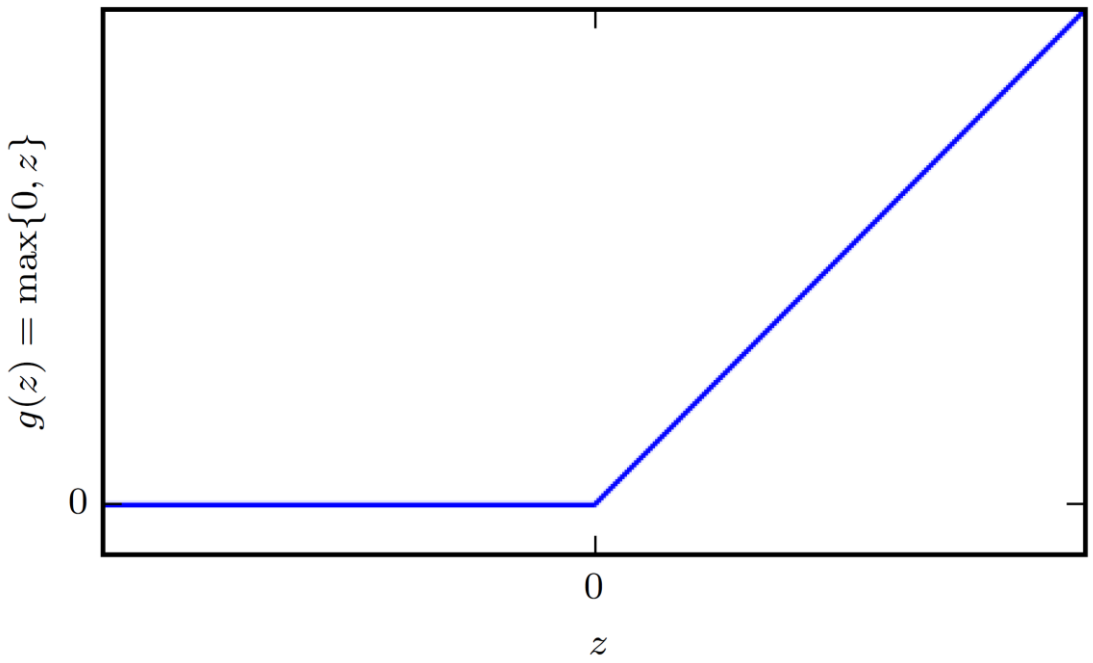
- Activation function g is typically chosen to be a function that is applied element-wise

$$h_i = g(x^T W_{:,i} + c_i)$$

Back to Learning XOR

- **Let there be nonlinearity!**
- ReLU: rectified linear unit: $g(z) = \max\{0, z\}$
- ReLU is applied element-wise to **h**:

$$h_i = g(x^T W_{:i} + c_i)$$



A Simple Example: Learning XOR

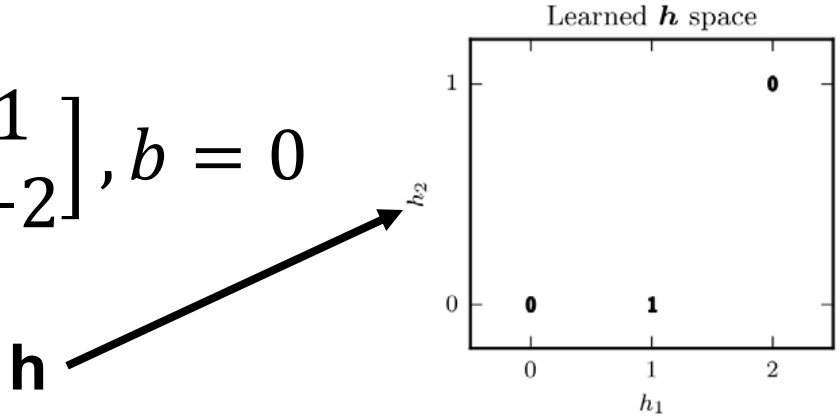
- Complete neural network model:

$$y = f(x; W, c, w, b) = f^{(2)}\left(f^{(1)}(x)\right) = w^T \max\{0, W^T x + c\} + b$$

- Obtain model parameters after training

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

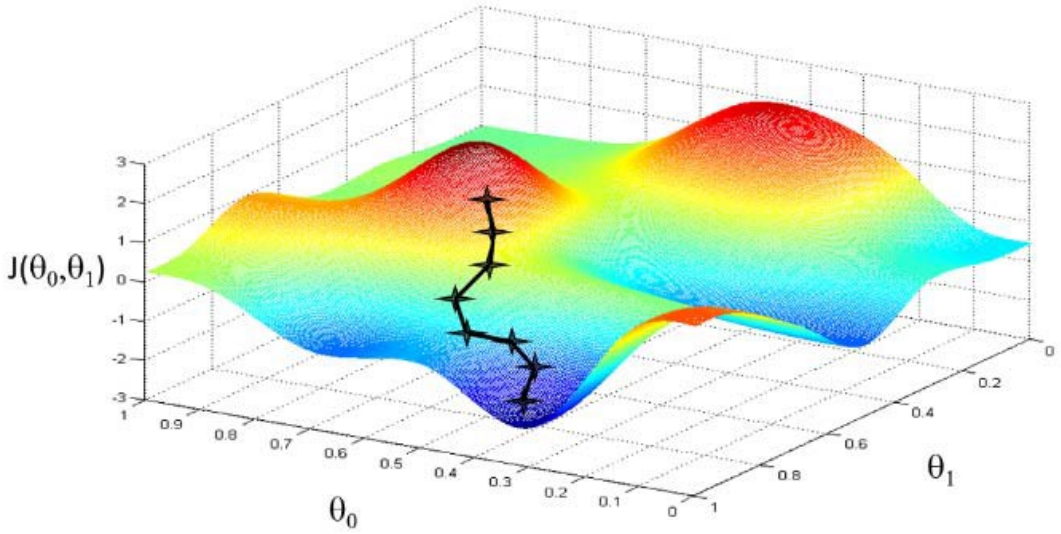
- Run the network



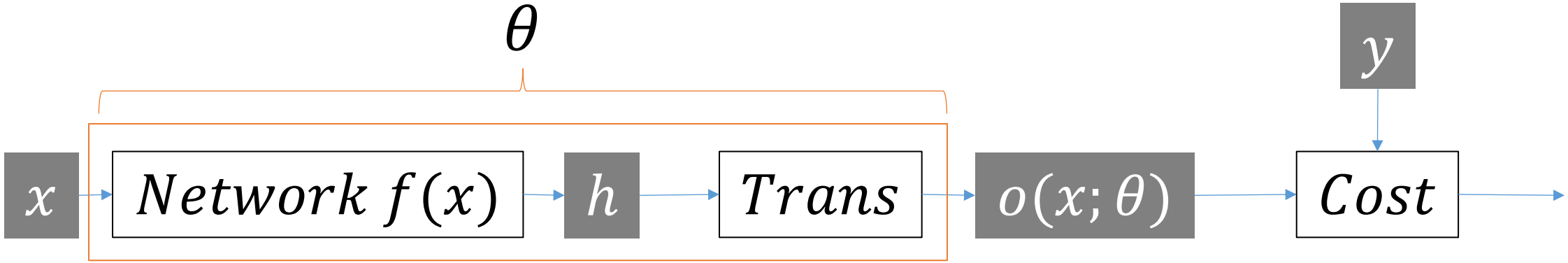
$$x = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \xrightarrow{W^T x + c} \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{w^T h + b} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Gradient-based Learning

- Nonlinearity of NN causes non-convex loss functions
- Means NNs are usually trained by using iterative, gradient-based optimizers, such as stochastic gradient descent methods
  - **No convergence guarantee**
  - Sensitive to initial values of parameters
    - Weight → small random values
    - Bias → zero of small positive values
  - Issues
    - Cost functions
    - Computation of gradients
    - Gradient-based optimization



General Setting of Gradient-based Learning



- x: inputs
- $f(x)$: Feedforward network
- h: Hidden units computed by $f(x)$
- $Trans.$: Output layer transforming h to output $o(x;\ \theta)$
- $o(x;\theta)$: Output units parameterized by model parameters θ
- $Cost$: A function of ground-truth y and output o to be minimized w.r.t. model parameters θ

Cost Function

- The Maximum likelihood (ML) principle provides a guide for designing cost function
- If we define a conditional distribution $p(y|x)$ as a distribution over y parameterized by the network outputs $\mathrm{o}(x;\ \theta)$

$$p(y|x) \triangleq p(y; o(x;\theta))$$

- The ML principle suggest we take the negative log-likelihood, or cross-entropy

$$-\log p\big(y; o(x;\theta)\big)$$

as the cost function to be minimized w.r.t. model parameter θ

$$J(\theta) = -E_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x;\theta)$$

Learning Conditional Statistics

- If we define

$$p_{model}(y|x;\theta) \triangleq N(y; o(x;\theta), I)$$

$$\Rightarrow J(\theta) = \frac{1}{2} E_{x,y\sim\hat{p}_{data}} \|y - o(x;\theta)\|^2 + const.$$

- Minimize the negative log-likelihood yields

$$\theta^* = \arg\min_{\theta} E_{x,y\sim p_{data}} \|y - o(x;\theta)\|^2$$

- With sufficient capacity, the network will learn the conditional mean

$$f(x;\theta^*) \approx E[y|x]$$

  which predicts the mean value of y for each x

Learning Conditional Statistics

- By the same token, if we define

$$p(y|\mathrm{x}) \triangleq Laplace(y; o(x;\theta), \gamma) = \frac{1}{2\gamma}\exp(-\frac{|y - o(x;\theta)|}{\gamma})$$

- Then, minimizing the negative log-likelihood yields

$$\theta^* = \arg\min_{\theta} E_{x,y\sim p_{data}}|y - o(x;\theta)|$$

- With sufficient capacity, the network will learn the conditional median

$$f(x;\theta^*) \approx Median[y|x]$$

  which predicts the median value of y for each x

- In other words, $o(x;\theta^*)$ satisfies

$$\int_{-\infty}^{o(x;\theta^*)} p(y|x)dy = \int_{o(x;\theta^*)}^{\infty} p(y|x)dy$$

Learning Conditional Statistics

- View the cost function as being a functional, rather than a function
  - A functional is a mapping from functions to real numbers
- Learning is to choose a function, not a set of parameters
  - We can design our cost functional to have its minimum occur at some specific function we desire
- For example – Mean square error

$$f^* = \arg\min_f E_{x,y\sim p_{data}} \|y - f(x)\|^2$$

yields

$$f^*(x) = E_{y\sim p_{data}(y|x)}[y] \Rightarrow a\ function\ that\ predict\ the\ mean\ of\ y\ for\ each\ value\ of\ x$$

- Another example – Mean absolute error

$$f^* = \arg\min_f E_{x,y\sim p_{data}} \|y - f(x)\|_1$$

yields

$$a\ function\ that\ predict\ the\ median\ value\ of\ y\ for\ each\ x$$

Cost Function Design Choice

- Some output units that saturate produce **very small gradients** when combined with mean squared error and mean absolute error
- While negative log-likelihood often with **large gradient** and predictable for learning
- ⇒ One reason that the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(y|x)$
- The choice of cost function is tightly coupled with **the choice of output unit**
  - Most of the time, we simply use the cross-entropy between the data distribution and the model distribution
  - The choice of how to represent the output then determines the form of the cross-entropy function

Output Units

- Any kind of neural network unit that may be used as an output
- We focus on the use of these units as **outputs of the model** only

- Common design choices
  - Linear Units for Gaussian Output Distributions
  - Sigmoid Units for Bernoulli Output Distributions
  - Softmax Units for Multinoulli Output Distributions

- Many other output units
  - The principle of maximum likelihood provides a guide for how to design a good cost function for nearly any kind of output layer

Learning Gaussian Output Distributions

- The n-dimensional Gaussian distribution $N(y; \mu, \Sigma)$ is given by

$$N(y; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(y-\mu)^T \Sigma^{-1}(y-\mu)\right)$$

- A conditional Gaussian can be learned by treating $o(x;\ \theta)$ as the means

$$p(y|x) = N(y; o(x;\theta), I)$$

- In this model, the outputs $o(x;\theta)$ often take a linear form

$$o(x;\theta) = W^T h(x) + b$$

- As such, they are known as **linear output units**
  - Maximizing the log-likelihood is equivalent to minimizing the mean squared error
- Linear units do not saturate
  - Pose little difficulty for gradient-based optimization algorithms
  - May be used with a wide variety of optimization algorithms

Learning Bernoulli Output Distributions

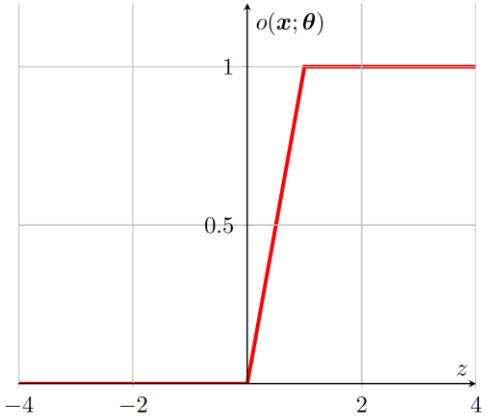
- Many tasks require predicting the value of a binary variable y.
  - Classification problems with two classes can be cast in this form.
- A conditional Bernoulli can be learned by having $o(x;\ \theta)$ as the very distribution parameter

$$p(y|x) = o(x;\theta)^y(1 - o(x;\theta))^{1-y}, y \in \{0,1\}$$

Learning Bernoulli Output Distributions

- The $o(x;\theta)$ must be in $\{0,1\}$ in order to be a valid parameter, one trivial implementation would be

$$o(x;\theta) = \max\{0, \min\{1, z\}\}, z = w^T h(x) + b$$



- This however may be problematic for the gradient of log-likelihood w.r.t. w, b is zero when $z = w^T h(x) + b$ is outside the unit interval

$$-\nabla_{w,b} \log p(y|x) = -\frac{\partial \log p(y|x)}{\partial o(x;\theta)} \boxed{\frac{\partial o(x;\theta)}{\partial z}} \nabla_{w.b} z$$

=0

- The gradient-based learning may fail to learn w, b properly

Sigmoid for Bernoulli Output

- An alternative approach is to use sigmoid units combined with the negative log-likelihood function

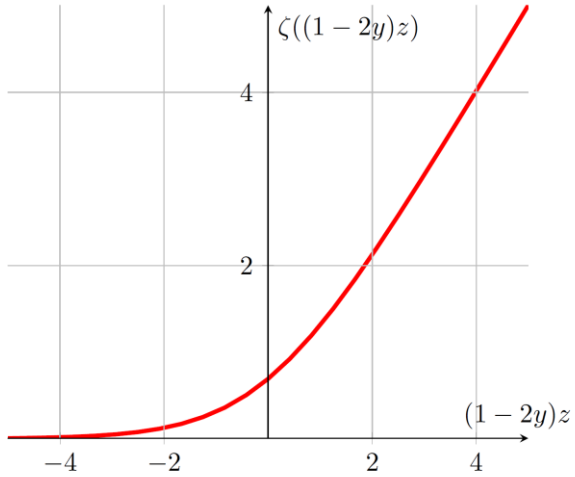
$$o(x;\theta) = \sigma(z), with\ z = w^T h(x) + b, \sigma(z) = \frac{e^z}{1+e^z}$$

- Be definition, the negative log-likelihood can be computed as

$$\begin{aligned} -\log p(y|x) &= -\log o(x:\theta)^y \left(1 - o(x;\theta)\right)^{1-y} \\ &= -\log \sigma\big((2y-1)z\big) \\ &= \zeta\big((1-2y)z\big) \end{aligned}$$

  - Where $y \in \{0,1\}$ and $\zeta(\cdot)$ is the softplus function

$$\zeta(x) = \log(1 + e^x)$$



Sigmoid for Bernoulli Output

- Ideally, we want $z = w^T h(x) + b$ to be very positive (respectively, negative) when the ground-true y = 1 (respectively, y = 0)
- This suggests that samples correctly classified have very negative $(1-2y)z \Rightarrow$ these samples will not contribute to the gradient computation

$$-\nabla_{w,b} \log p(y|x) = -\frac{\partial \log p(y|x)}{\partial \zeta} \frac{\partial \zeta}{\partial (1-2y)z} \frac{\partial (1-2y)z}{\partial z} \nabla_{w,b} z$$

$\frac{\partial \zeta}{\partial (1-2y)z}$ = 0

- On the other hand, samples incorrectly classified have very positive $(1-2y)z \Rightarrow$ leads to strong gradient
- When sigmoid units are combined with other cost functions, e.g. mean squared error, the gradient vanishing problem may occur

Learning Multinoulli Output Distributions

- A Multinoulli random variable x has n possible values with distribution

$$P(x;\alpha) = \prod_{i=1}^{n} (\alpha_i)^{1_{x=i}} \ , x = 1,2,\dots,n$$

  - Where

$$\alpha_i \in [0,1], \forall i$$

$$\sum_{i=1}^{n} \alpha_i = 1$$

$$1_{x=i} = \begin{cases} 1, if\ x = i \\ 0, if\ x \neq i \end{cases}$$

- When using a one-hot vector (or the 1-of-K coding) x, in which only one element equals to 1 with the others being 0, to represent the n possible Multinoulli distribution is given by

$$P(x;\alpha) = \prod_{i=1}^{n} (\alpha_i)^{x_i} \ , x_i \in \{0,1\}$$

- Observe that Bernoulli is a special case of Multinoulli with $n = 2$

Learning Multinoulli Output Distributions

- We are solving a probability distribution over a discrete variable with n possible values
  - Used as the output of a classifier for n classes
- Goal: a vector $\hat{y}, \hat{y}_i = P(y = i|x)$
  - Each $\hat{y}_i$ lies in the interval [0,1]
  - Entire vector $\hat{y}$ sum to 1

$$z = W^T h + b, \qquad z_i = \log \tilde{P}\,(y = i|x)$$

  - Softmax function - softened version of $\arg\max$

$$Softmax(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_i)}$$

Softmax for Multinoulli

- A conditional Multinoulli can be learned by having its distribution parameters be modeled by softmax output units $o(x;\theta)$

$$p(y|x) = \prod_{i=1}^{n} (o(x;\theta)_i)^{y_i}$$

  - Where y is a one-hot vector and

$$o(x;\theta)_i = softmax(z)_i = \frac{\exp(z_i)}{\sum_{j=i}^{n} \exp(z_j)}$$

$$z = W^T h(x) + b$$

- When the difference between the maximal element of z and the others becomes large, **softmax** becomes a form of **winner-take-all**, namely, one output is nearly 1 and the others are nearly 0
  - A way to create a form of competition between the units that participate in it
- As such, it is viewed as a softened version of argmax with a one-hot representation

Softmax for Multinoulli

- Softmax overparameterizes the distribution: the n-th probability may be obtained by subtracting the first $n-1$ probabilities from 1
  - only $n-1$ parameters are necessary
- One may require that one element of z be 0
  - $\Rightarrow$ leads to sigmoid when $n = 2$

$$\sigma(z) = \frac{\exp(z)}{\exp(z) + \exp(0)} = \frac{1}{1 + \exp(-z)}$$

- As with sigmoid units, softmax units almost always come with the log-likelihood function

$$-\log p(y|x) = -\sum_{i=1}^{n} y_i \log(o(x;\theta)_i) = -\sum_{i=1}^{n} y_i \left(z_i - log \sum_{j=1}^{n} \exp(z_j) \right)$$

Softmax for Multinoulli

- Assuming $y_i = 1$ and $y_j = 0, j \neq i$, we have

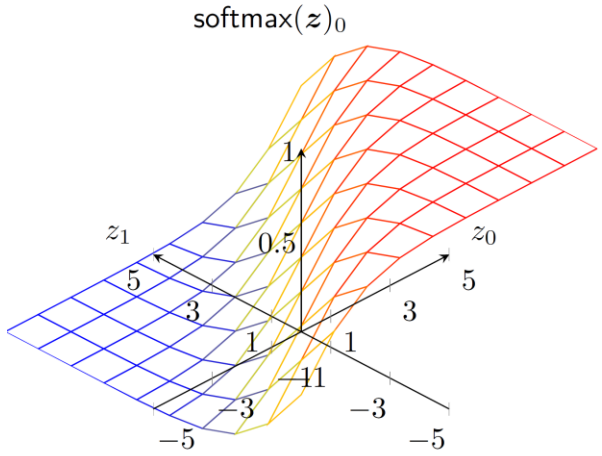
$$-\log p(y|x) = -(z_i - \log \sum_{j=1}^{n} exp(z_j))$$

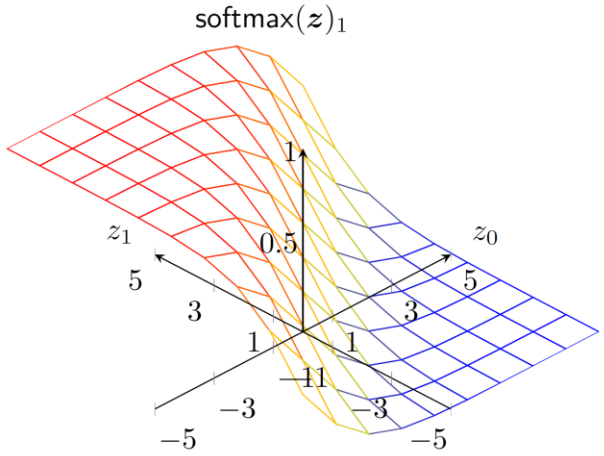
- This suggests that when the ground truth $y_i = 1$, minimizing the negative log-likelihood amounts to maximizing z_i and **penalizes the most active incorrect prediction** z_j if $z_j \gg z_k, k \neq j$
- On the other hand, samples correctly classified (i.e., $z_i \gg z_k, k \neq i$) will contribute little to the gradient computation
  - In this case

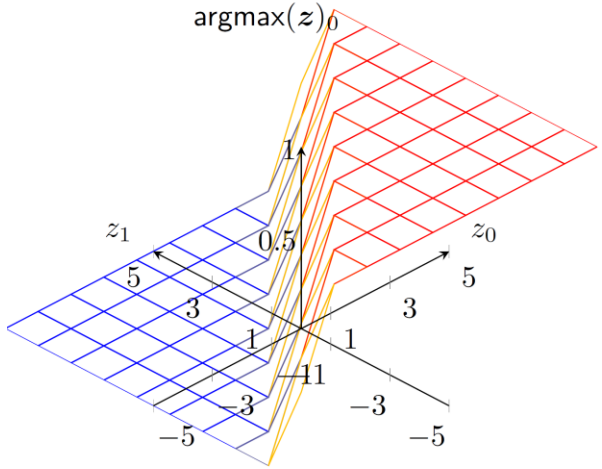
$$(z_i - \log \sum_{j=1}^{n} \exp(z_j)) \approx 0$$

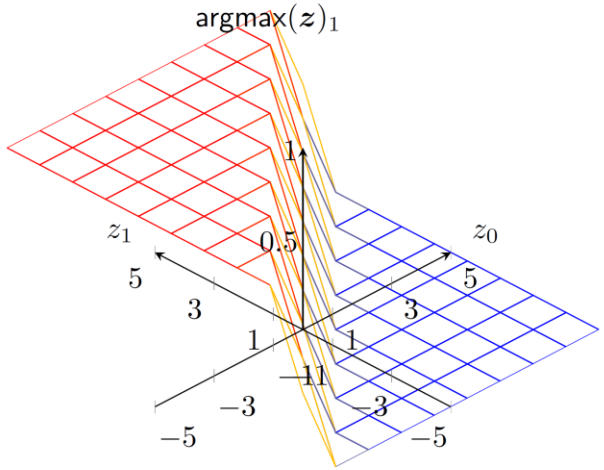
- Observe that the log undoes the saturating effect of the softmax

Softmax vs. Argmax









Common Design Choices

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	GAN, VAE, FVBN	Various

Relative Frequencies

- With sufficient capacity, minimizing the negative log-likelihood will

$$-E_{x,y\sim\hat{p}_{data}} \log p(y|x) = -E_{x,y\sim\hat{p}_{data}} \left[\sum_{i=1}^{n} y_i \log o(x;\theta)_i \right]$$

$$= -\sum_{j=1}^{m} \sum_{i=1}^{n} y_i^{(j)} \log o\left(x^{(j)};\theta\right)_i$$

  - Drive output units to approximate relative frequencies in training data

$$o(x;\theta)_i \approx \frac{\sum_{j=1}^{m} 1_{x^{(j)}=x, y_i^{(j)}=1}}{\sum_{j=1}^{m} 1_{x^{(j)}=x}}$$

Relative Frequencies

- To see this, solving the following constrained optimization problem:

$$\arg\min_{o(x;\theta)_i}\left(-\sum_{j;x^{(i)}=x}\sum_{i=1}^{n} y_i^{(j)}\log o(x;\theta)_i\right), s.t. \sum_{i=1}^{n} o(x;\theta)_i = 1$$

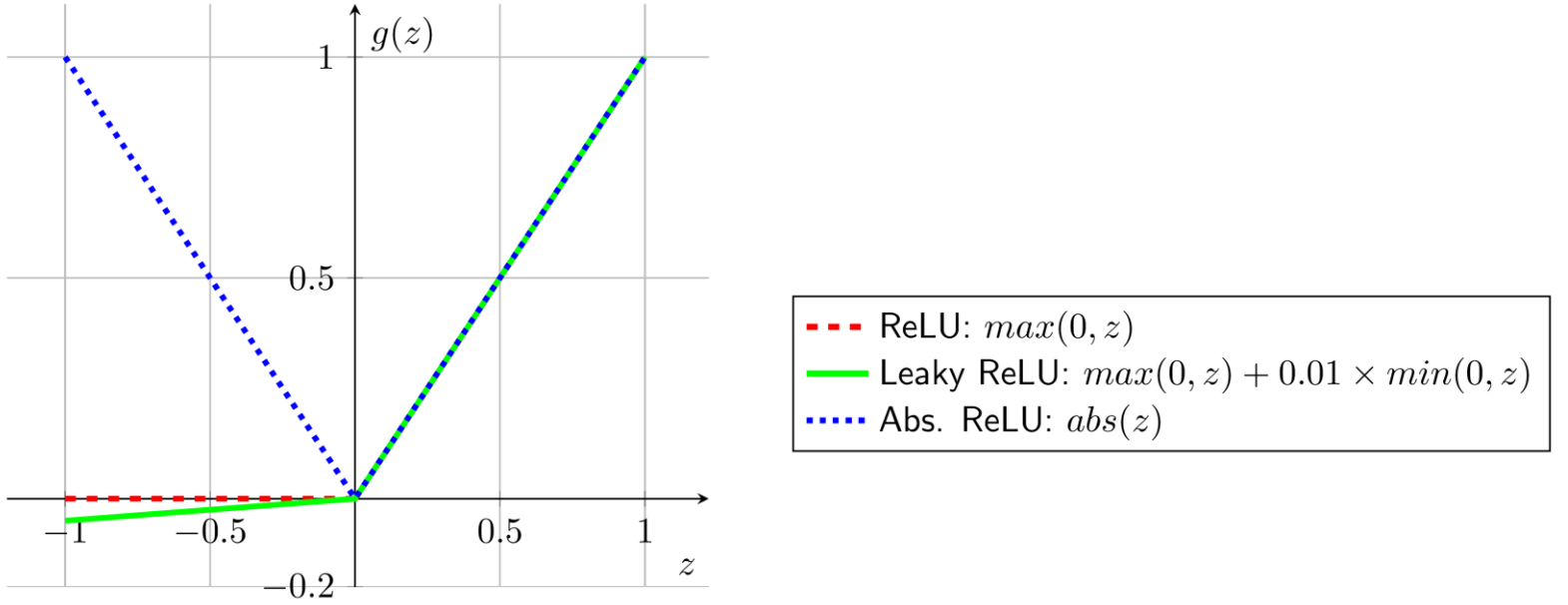
- Essentially, we are using the network to encode/approximate relative frequencies for different combinations of x, y
- How should we interpret the value of $o(x;\theta)$ if we input an x that is not included in training data?

Hidden Units

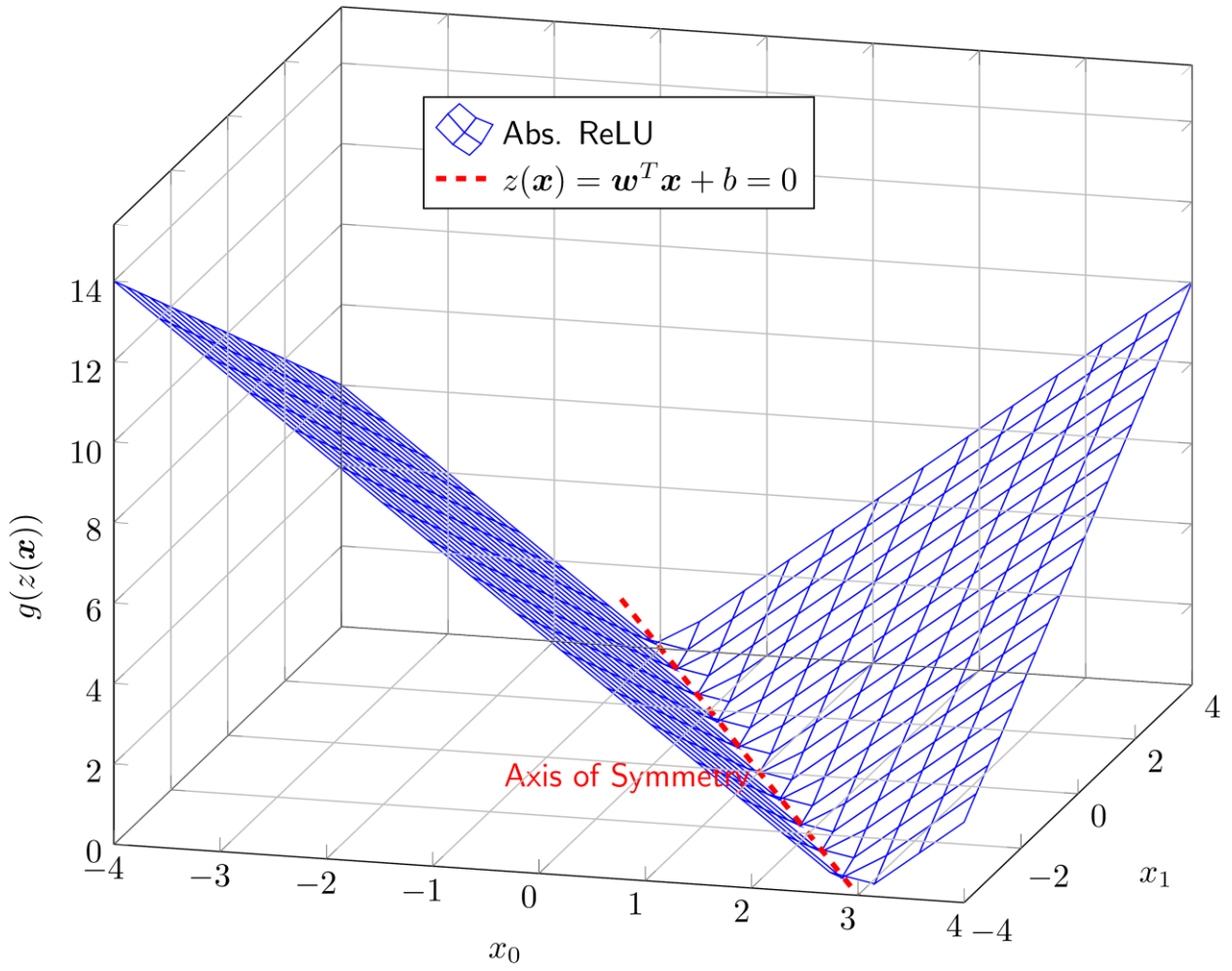
- Design of hidden units in an extremely active area of research
- ReLUs are an excellent default choice
  - 90% of the time
- For some research projects, get creative
- Many hidden units perform comparably to ReLUs. New hidden units that perform comparably are rarely interesting
- Although not differentiable at all point, it is still okay to use for gradient-based learning algorithm.
  - Use left or right derivative, instead

Hidden Units

- Computation of hidden units h
  1. Accepting input x
  2. Applying affine transformation $z = W^T x + b$
  3. Applying element-wise non-linear mapping $h = g(z)$
- Typical activation functions $g(z)$



Activation Functions in Input Space



ReLUs

- **Re**ctified **L**inear **U**nits (ReLU): $g(z) = \max\{0, z\}$
- Easy to optimize because the derivatives through ReLU remain large whenever the unit is active
- When initializing the parameters of the affine transformation, it can be a good practice to set all elements of b to a small, positive value, such as 0.1

$$h = g(W^T x + b)$$

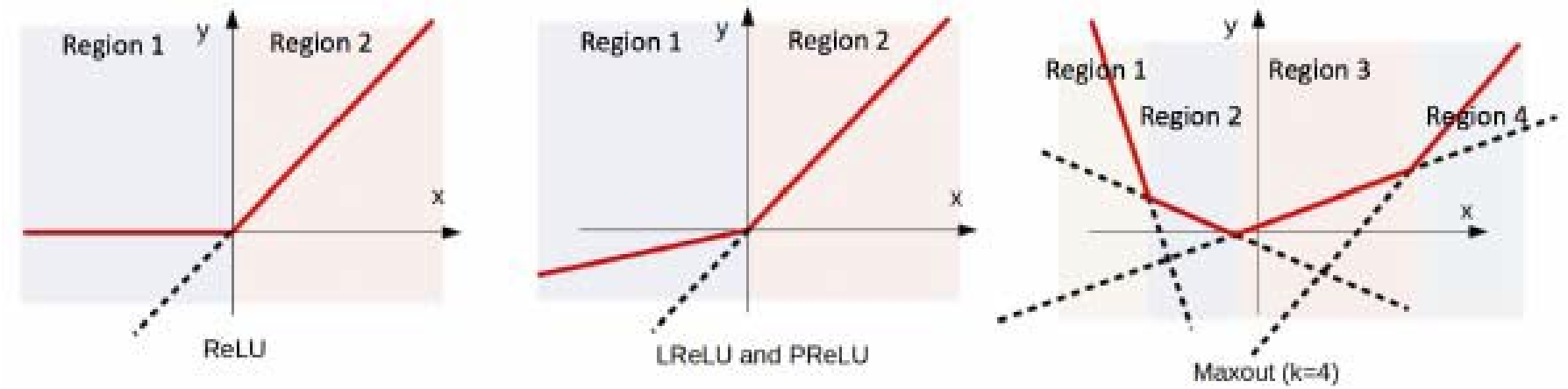
- Drawback
  - ReLUs cannot learn via gradient-based methods on examples with zero activation

Generalizations of ReLUs

- Use a non-zero slope α_i when $z_i < 0$

$$h_i = g(z, \alpha)_i = \max\{o, z_i\} + \alpha_i \min\{0, z_i\}$$

- Absolute value rectification fixes $\alpha_i = -1 : g(z) = |z|$
- Leaky ReLU fixes α_i to a small value like 0.01
- Parametric ReLU or PReLU treats α_i as a learnable parameter



Maxout Units

- Formulation

$$g(z)_i = \max_{j \in \mathbb{G}(i)} z_j$$

  - Where

$$\mathbb{G}(i) = \{(i-1)k + 1, (i-1)k + 2, \ldots, (i-1)k + k\}$$

- Maxout can implement many typical activation functions
  - Given

$$z_1 = W_1^T x + b_1$$
$$z_2 = W_2^T x + b_2$$

  - ReLU

$$g(z_1) = \max(z_1, z_2)$$
$$with\ z_2 = 0\ by\ (W_2, b_2) = 0$$

  - Leaky ReLU

$$g(z_1) = \max(z_1, z_2)$$
$$with\ z_2 = 0.01 \times z_1\ by\ (W_2, b_2) = 0.01 \times (W_1, b_1)$$

Maxout Units

- Maxout can be seen as **learning the activation function**, in the sense that $(w_1, b_1)$and $(\mathrm{w}_2, \mathrm{b}_2)$ can both be learned
- Maxout can approximate any piecewise linear, convex function in input space x, when more pieces $\{z_i\}$ are input
- ReLU and all these variants exhibit linear behavior

Why Linearity?

- Toy problem: Training network with only one hidden unit

$$x \xrightarrow[Affine]{f(x)=w^T x+b} z \xrightarrow[Activation]{g(z)} h \xrightarrow[Likelihood]{c(h)} J$$

- Partial derivative of J w.r.t. model parameter w_i is given by

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial h} \times \frac{\partial h}{\partial z} \times \frac{\partial z}{\partial w_i} = c'(h) \times g'(z) \times x_i$$

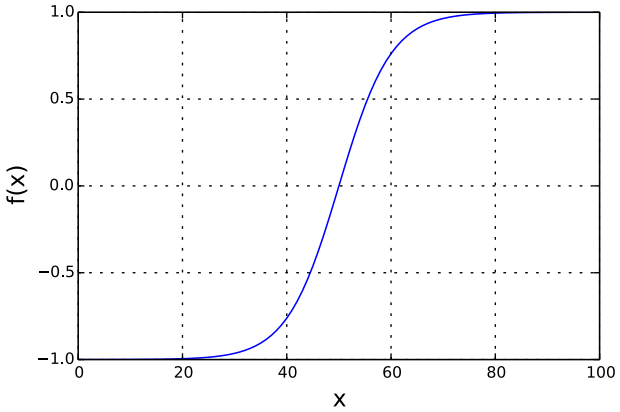
- Linearity in activation (i.e. $\frac{\partial h}{\partial z}$ is some non-zeros constant) helps ensure a well-behaved gradient w.r.t. model parameters

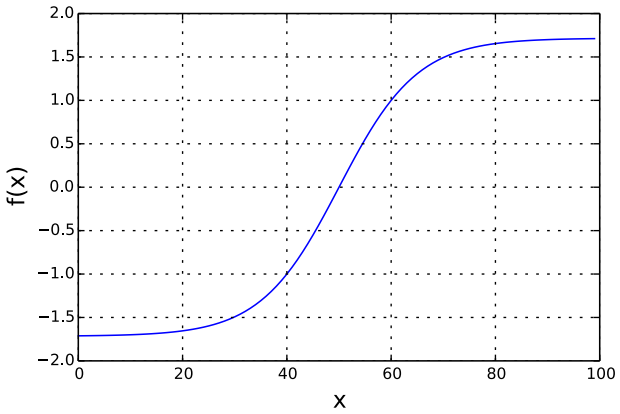
A Few Other Reasonable Hidden Units

- Radial basis function (RBF) unit
  - A real-valued function whose value depends only on the distance between the input and some fixed point, either the origin
  - Gaussian RBF example
    - $h_i = \exp\left(-\frac{1}{\sigma_i^2}\left\|W_{:,i} - x\right\|^2\right)$
    - It become more active as x approaches a template $W_{:,i}$
    - Difficult to optimize - because it saturates to 0 for most x

A Few Other Reasonable Hidden Units

- Tanh
  - $g(a) = \tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$
  - Non-linearity compresses the input in the range $[-1,1]$
  - Zero-centered
  - The gradients for tanh are steeper than sigmoid
    - Suffers from the vanishing gradient problem
- LeCun's Tanh
  - Introduced in Yann LeCun
  - $f(a) = 1.7159 \times \tanh(\frac{2}{3}a)$
  - The constants have been chosen to keep the variance of the output close to 1
    - Because the gain of the sigmoid is roughly 1 over its useful range

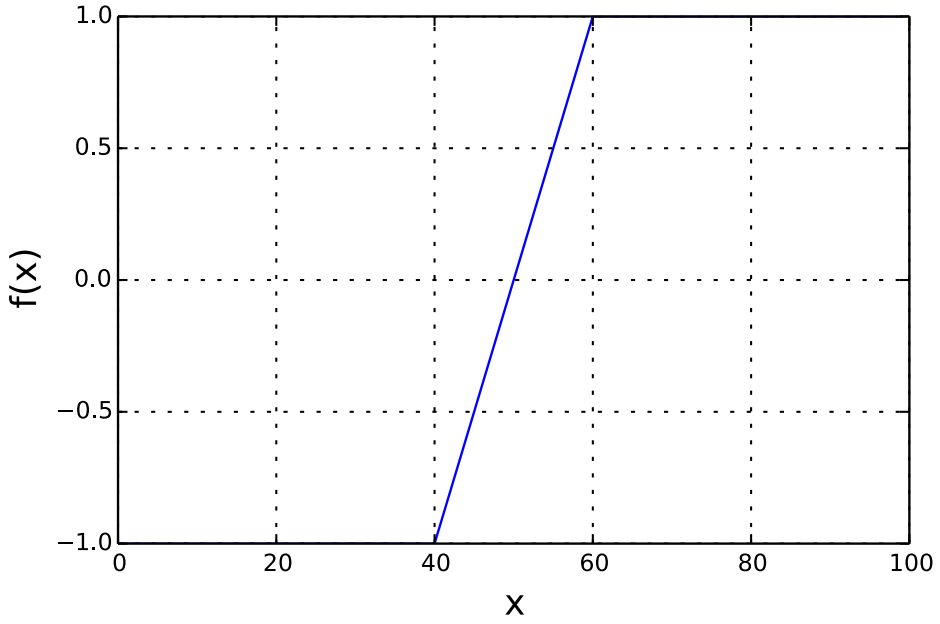




LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K. R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9-48). Springer, Berlin, Heidelberg.

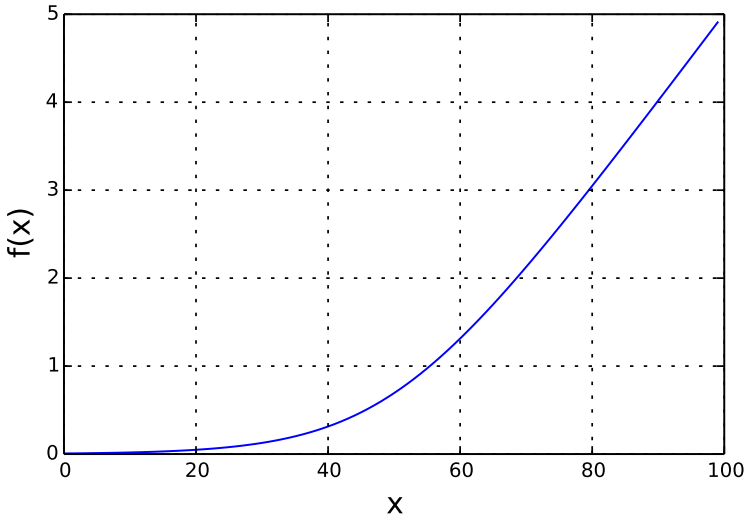
A Few Other Reasonable Hidden Units

- Hard Tanh
  - Shaped similarly to the tanh and the rectifier
  - Unlike rectifier, it's bounded
  - $g(a) = \max(-1, \min(1, a))$



A Few Other Reasonable Hidden Units

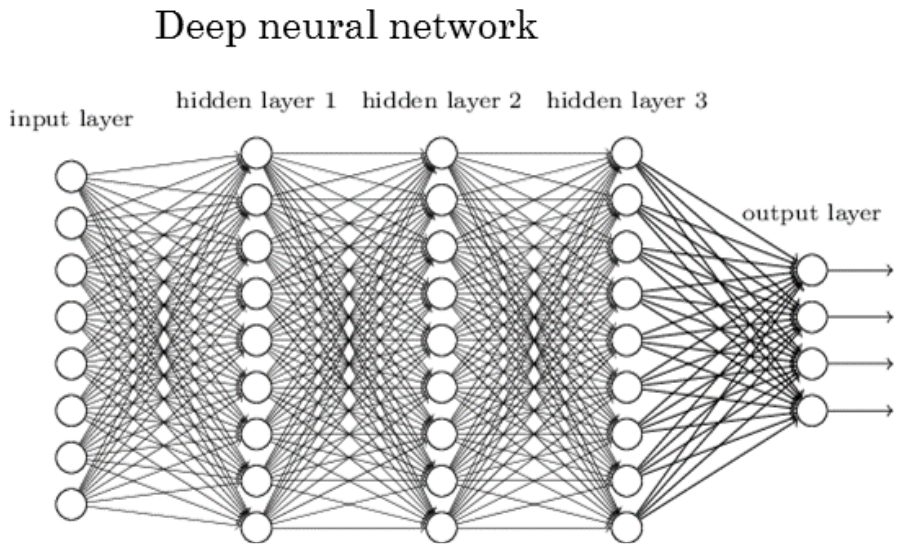
- Softplus - a smooth version of the rectifier
  - $g(a) = \zeta(a) = \log(1 + e^a)$
  - Key characteristic
    - Differentiable everywhere
    - Saturating less completely
  - Does Softplus unit have an advantage over the rectifier ?
    - **Empirically, it does not**
  - Generally discouraged



The softplus demonstrates that the performance of hidden unit types can be very counterintuitive

Architecture Design

- Architecture: overall structure of the network
  - How many units it should have
  - How these units should be connected to each other
  - How to choose the depth and width of each layer
- Deeper networks often
  - Use far fewer units per layer and far fewer parameters
  - Generalize to the test set
  - Are harder to optimize



Universal Approximation Theorem

- Given enough hidden units, a feedforward network can approximate any Borel measurable function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ with any degree of accuracy
  - Any continuous function $f: \mathbb{R}^n \rightarrow \mathbb{R}$ on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable
  - (Hornik et al., 1989; Cybenko, 1989)
- A feedforward network with a single layer is sufficient to represent any function
  - But the layer may be infeasibly large and may fail to learn and generalize correctly
- The theorem holds true for networks with rectified linear activation functions
  - (Leshno et al., 1993)

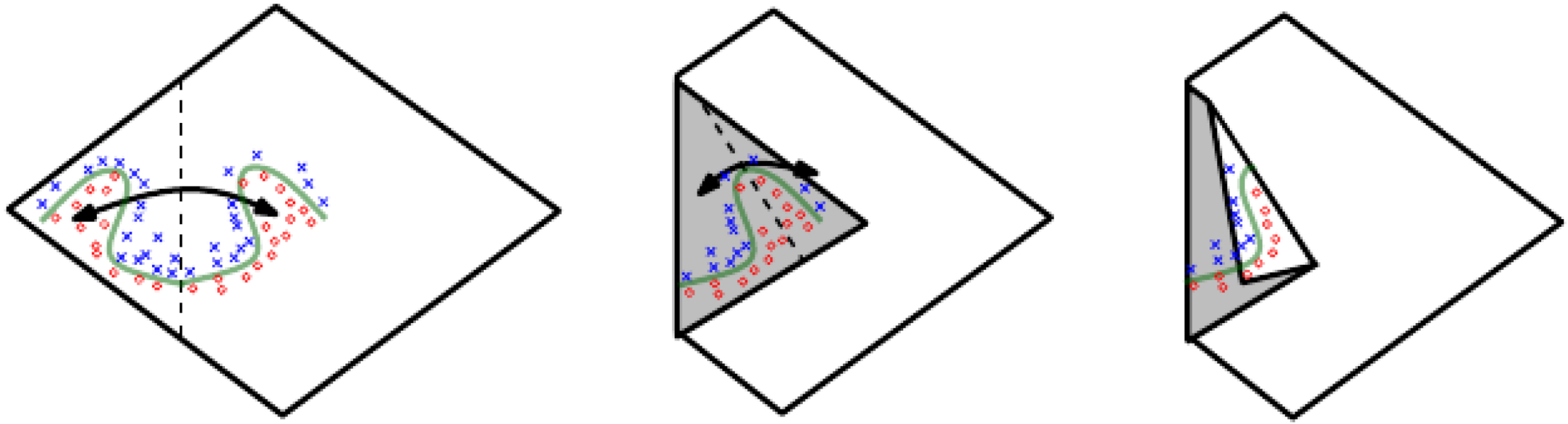
How Many Are Enough

- Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow network of one hidden layer
  - (Montufar et al., 2014)
- Empirically, deeper models can use fewer units to represent the desired function and generalize better

- Using deeper models can
  - Reduce the number of units required to represent the desired function
  - Reduce the generalization error

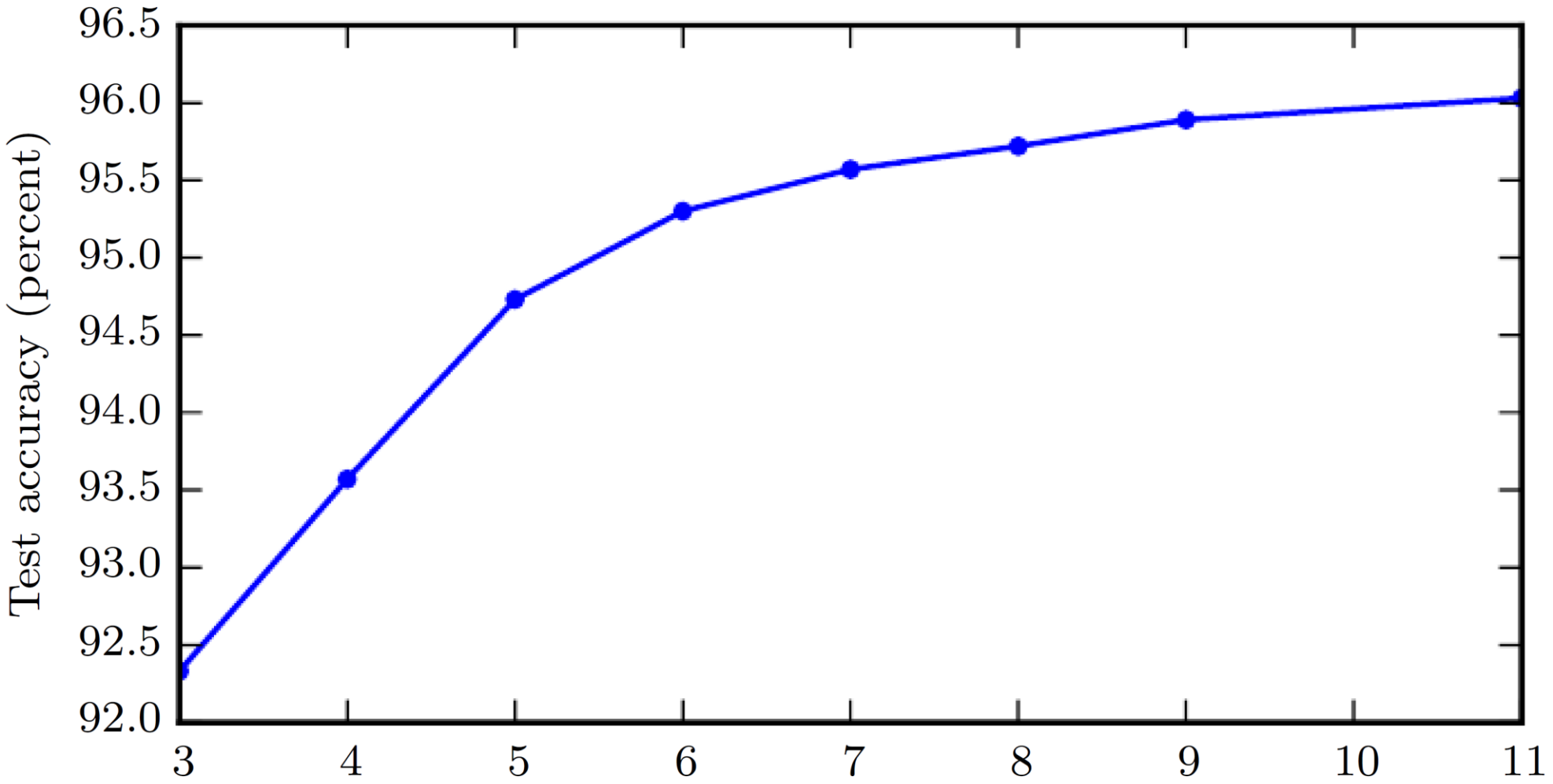
Universal Approximation Properties and Depth

- A network with absolute value rectification creates mirror images of the function computed on top of hidden units

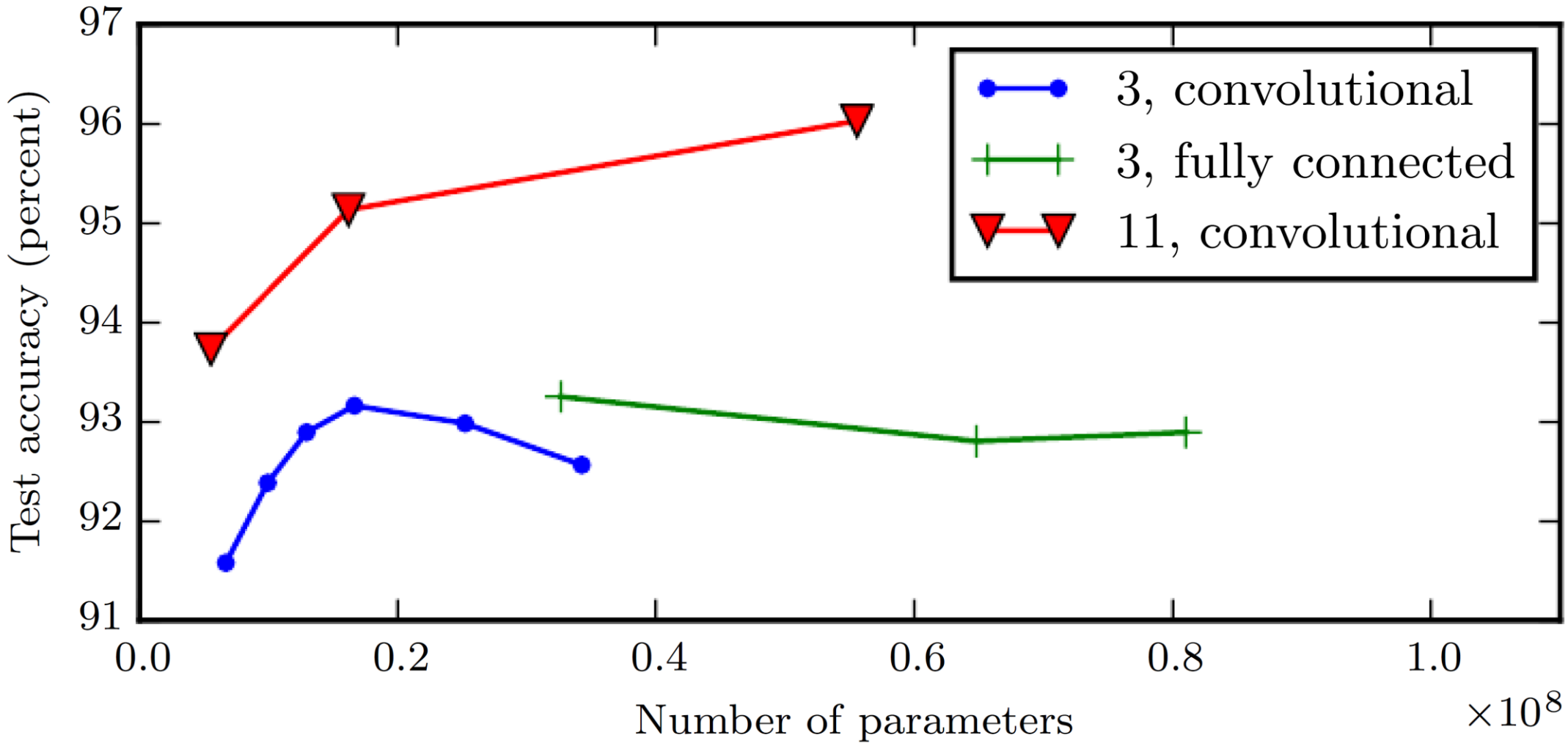


Better Generalization with Greater Depth

- SVHN dataset



Large, Shallow Models Overfit More

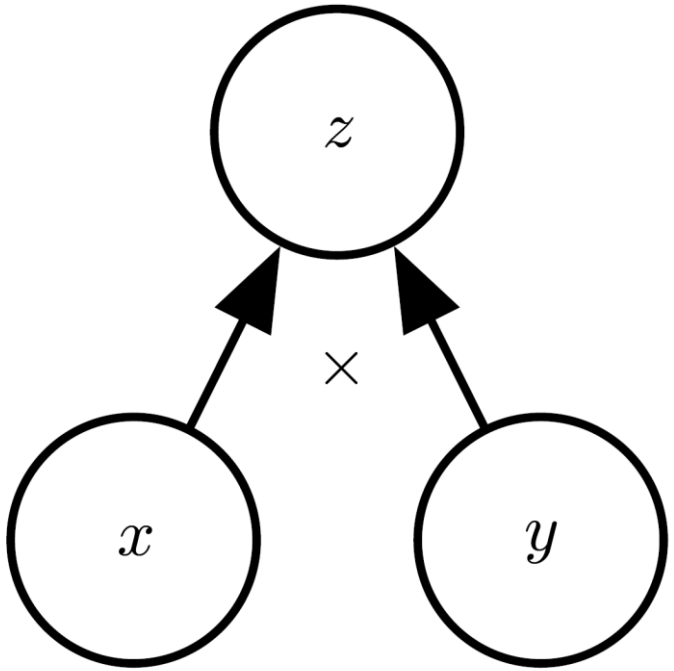


Computational Graph

- To formalize computation as a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$
  - A node $v \in \mathcal{V}$ indicates a variable
    - Scaler
    - Vector
    - Matrix
    - Tensor
  - A directed edge $e \in \mathcal{E}$ from x to y indicates that y is computed by applying an operation to x
  - An operation
    - A simple function of one or more variables
    - Return only a single output variable
- Observe that some nodes represent model parameters w, b

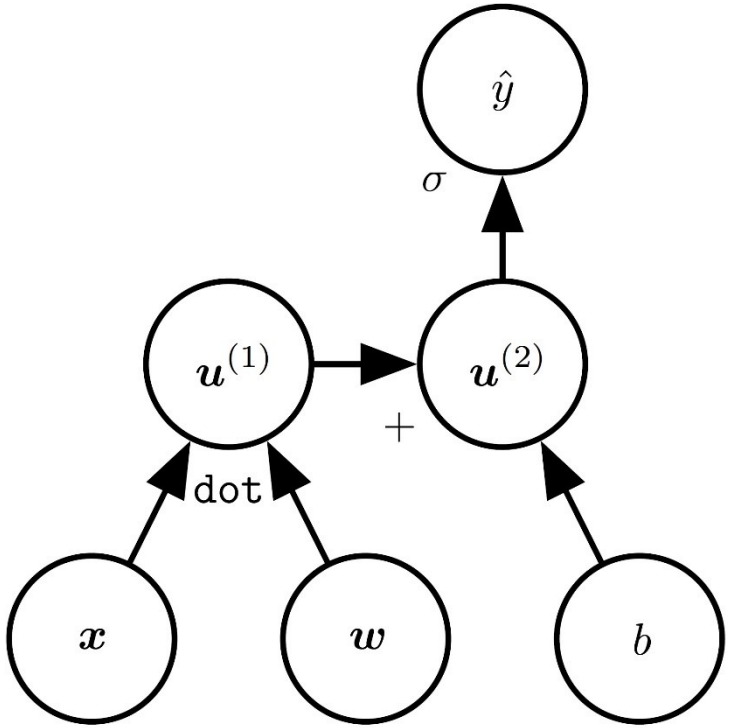
Computation Graph - Multiplication

- $z = x \times y$



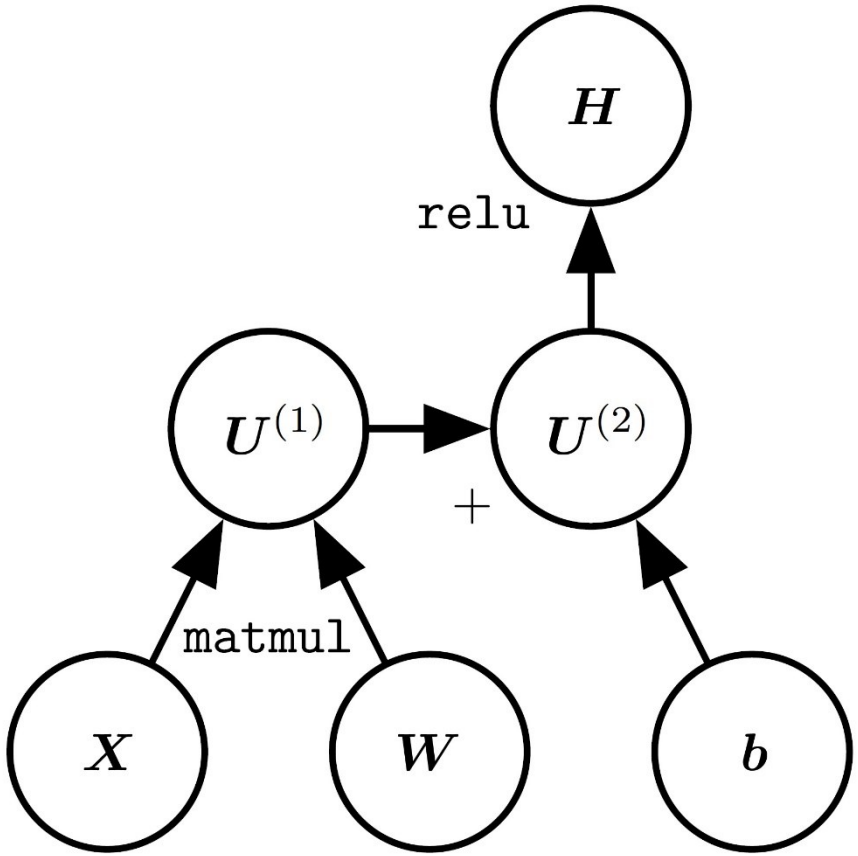
Computation Graph - Logistic Regression

- $\hat{y} = \sigma(w^T x + b)$



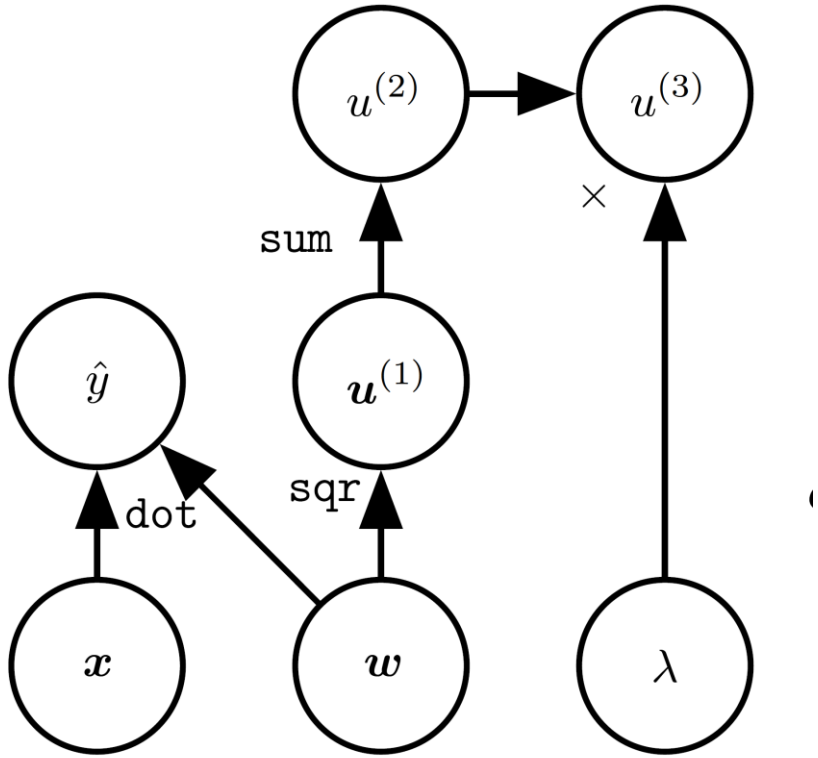
Computation Graph – ReLU

- $H = \max\{0, XW + b\}$

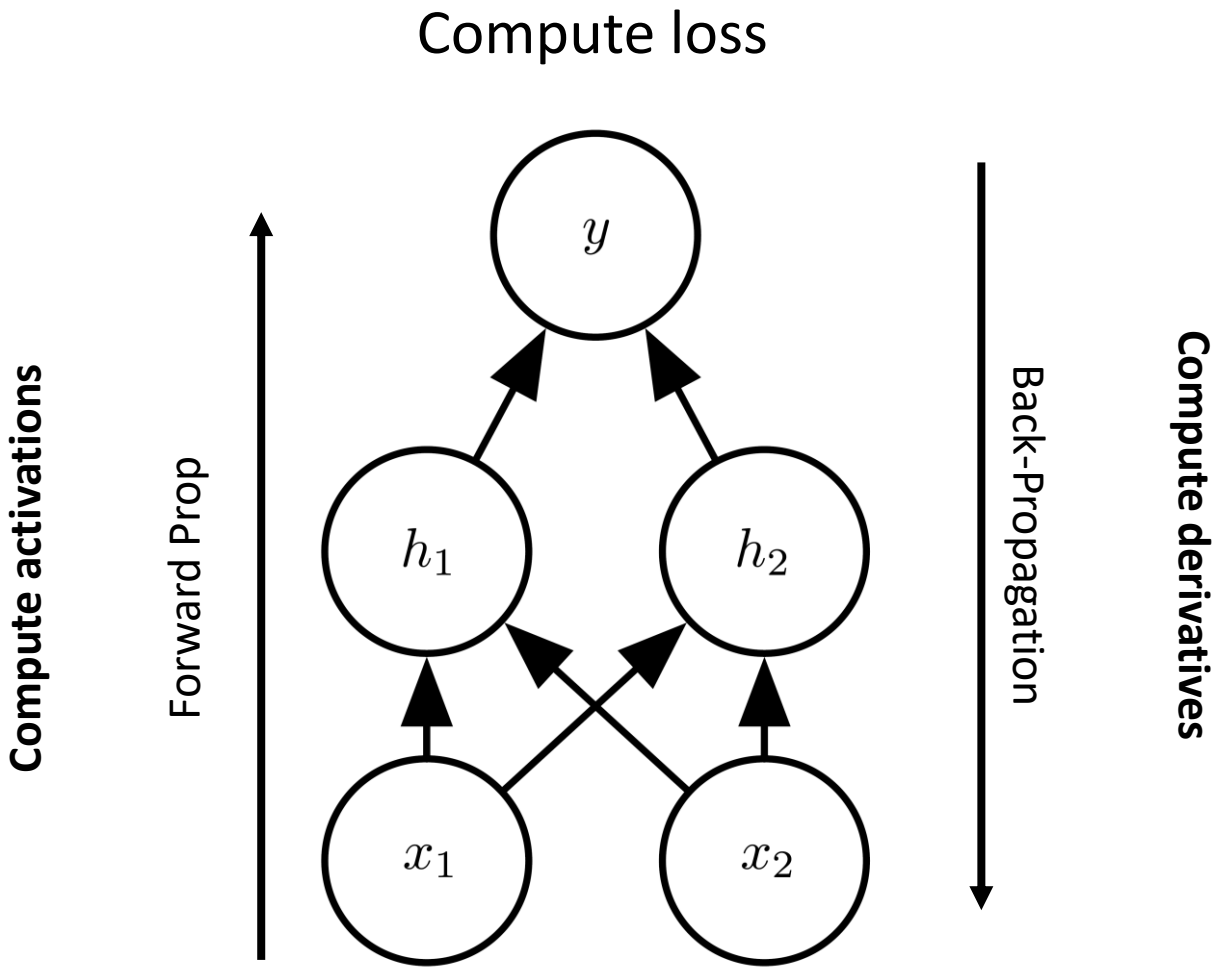


Computation Graph – Linear Regression and Weight Decay

- Weights w are used by computing $\hat{y}$ and weight decay penalty
  - Prediction output $\hat{y} = w^T x$
  - Weight decay penalty $\lambda \sum_i w_i^2$



Simple Example of Back-Prop



Back-Propagation

- Back-propagation is **just the chain rule** of calculus

$$\frac{\mathrm{dz}}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

- But it's a particular implementation of the chain rule
  - Use dynamic programming (table filling)
  - Avoid recomputing repeated subexpressions
  - Speed vs. memory tradeoff

Chain Rule of Calculus

- To compute the derivative $\frac{dz}{dx}$ of a function $z(x)$ formed by the composition of functions $z(x) = f\big(g(x)\big)$

$$x \xrightarrow{g(x)} y \xrightarrow{f(y)} z$$

  - $x \in \mathbb{R}$ is a real number
  - $f, g: \mathbb{R} \to \mathbb{R}$ are real-valued functions of single variable

- The chain rule states that

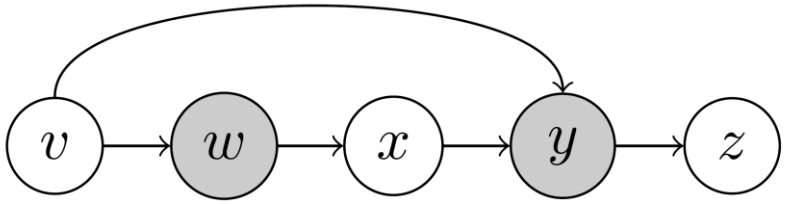
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Chain Rule of Calculus

- As an extension, we have for the following graph

$$\frac{dz}{dv} = \frac{dz}{dy}\frac{dy}{dv} + \frac{dz}{dw}\frac{dw}{dv} = \sum_{n:v \in Pa(n)} \frac{dz}{dn}\frac{dn}{dv}$$

  - Where $\mathrm{P}a(n)$ is the set of nodes that are parents of n



- To verify the result requires another chain rule from calculus

$$z = f(y_1, y_2), \qquad y1 = g_1(x), y2 = g_2(x)$$

$$\frac{dz}{dx} = \frac{dz}{dy_1}\frac{dy_1}{dx} + \frac{dz}{dy_2}\frac{dy_2}{dx}$$

Vector Case

- z is a scaler, and x, y are vector

$$x_{m\times 1} \xrightarrow{g(x)} y_{n\times 1} \xrightarrow{f(y)} z_{1\times 1}$$

- Applying the chain rules lead to

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}, i = 1,2,\dots,m$$

- In matrix notation,

$$\begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{bmatrix}$$

Vector Case

- This is recognized as a **Jacobian-gradient product**

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

  - Where $\frac{\partial y}{\partial x}$ (abbrev. $J_{y,x}$) is known as **Jacobian matrix** with

$$\left(\frac{\partial y}{\partial x}\right)_{i,j} = \frac{\partial y_i}{\partial x_j}$$

- As an example, when $y = g(x) = Wx$ (i.e. $y_i = \sum_j w_{i,j} x_j$),

$$\frac{\partial y}{\partial x} = W$$

Matrix Case

- z is a scaler, and X, Y are matrices

$$X_{m\times n} \xrightarrow{g(X)} Y_{s\times k} \xrightarrow{f(Y)} z_{1\times 1}$$

- The chain rule suggests that

$$\frac{\partial z}{\partial x_{i,j}} = \sum_{s,k} \frac{\partial z}{\partial y_{s,k}} \frac{\partial y_{s,k}}{\partial x_{i,j}}, \forall i, j$$

- More generally, when X, Y are tensors (high dimensional arrays)

$$\nabla_X z = \sum_j \left(\frac{\partial z}{\partial Y_j}\right) \nabla_X Y_j$$

  - Where $Y = g(X), z = f(Y)$ and X is treated as if it were a vector

Matrix Example

- Example 1: Assume $Y = g(X) = WX$
  - Let
    - $\nabla_Y z$ denote a matrix with its element (i, j) given by $\frac{\partial z}{\partial Y_{i,j}}$
    - $\nabla_X z$ be a matrix with its element (i, j) given by $\frac{\partial z}{\partial X_{i,j}}$
  - Observe that each **column** $Y_{:,j}$ of Y is a function of the **corresponding column** $X_{:,j}$ in X
    - i.e. $Y_{:,j} = WX_{:,j}$
  - We apply the Jacobian-gradient product of vector form to obtain

$$\nabla_{X_{:,j}} z = W^T \nabla_{Y_{:,j}} z \Rightarrow \nabla_X z = W^T \nabla_Y z$$

Matrix Example

- Example 2: Assume $Y = g(X) = XW$
  - Let
    - $\nabla_Y z$ denote a matrix with its element (i, j) given by $\frac{\partial z}{\partial Y_{i,j}}$
    - $\nabla_X z$ be a matrix with its element (i, j) given by $\frac{\partial z}{\partial X_{i,j}}$
  - Observe that each **row** $Y_{i,:}$ of Y is a function of the **corresponding row** $X_{i,:}$ in X
    - i.e. $Y_{i,:} = X_{i,:}W$
    - Or equivalently, $Y_{i,:}^T = W^T X_{i,:}^T$
  - We apply the Jacobian-gradient product of vector form yields

$$\nabla_{X_{i,:}^T} z = W^T \nabla_{Y_{i,:}^T} z \Rightarrow \nabla_X z = (\nabla_Y z) W^T$$

Back to Back-Propagation

- Toy problem: To compute the derivative of J w.r.t x

$$x \xrightarrow{f(x)} z \xrightarrow{g(z)} h \xrightarrow{c(h)} J$$

- From the chain rule, we have

$$\begin{aligned}\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial x} \\ &= c'(h)g'(z)f'(x) \\ &= c'\Big(g\big(f(x)\big)\Big)\,g'\big(f(x)\big)\,f'(x)\end{aligned}$$

Back-Propagation Implementation

- There are two possible implementation
  - One based on the last equality incurs redundant subcomputation (e.g. $f(x)$)
  - The other follow the penultimate equality requires z, h be pre-computed through forward propagation
- Assuming z, h have been-precomputed, one way to compute the derivatives of J w.r.t all variables x, z, h is to process in the order of
  1. $\frac{\partial J}{\partial h} = c'(h)$
  2. $\frac{\partial J}{\partial z} = \frac{\partial J}{\partial h} g'(z)$
  3. $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial z} f'(x)$
- This technique is known as **the back-propagation** method

Implementing Back-Prop

- Forward-prop of computing a computational graph
  - $x^{(1)}, x^{(2)}, \cdots, x^{(n_i)}$: $input$
  - $u^{(1)}, u^{(2)}, \cdots, u^{(n_i)}$: input units
  - $u^{(n_i+1)}, u^{(n_i+2)}, \cdots, u^{(n-1)}$: hidden units
  - $u^{(n)}$: output units
  - Each nodes computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$
- Pseudo forward-prop procedure

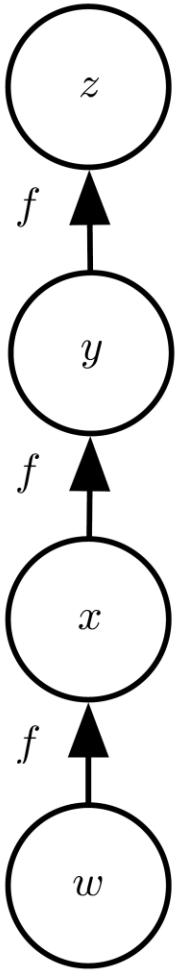
1: for $i = 1, \ldots, n_i$ do
2: $\quad u^{(i)} \leftarrow x_i$
3: endfor
4: for $i = n_i + 1, \ldots n$ do
5: $\quad \mathbb{A}^{(i)} \leftarrow \{u^{(i)} | Pa(u^{(i)})\}$
6: $\quad u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
7: endfor
8: return $u^{(n)}$

Implementing Back-Prop

- Back-Prop of computing a computational graph
  - Computing derivative of $u^{(n)}$ with respect to the variables in the graph
  - $gradTable$: a data structure that will store the derivative that have been compute
    - $gradTable[u^{(i)}]$: store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$
- Pseudo Back-prop Procedure

1: run forward-prop
2: Initialize $gradTable$
3: $gradTable[u^{(n)}] \leftarrow 1$
4: for $j = n - 1$ to 1 do
5: $gradTable[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})} gradTable[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
6: endfor
7: return $\{gradTable[u^{(i)}] | i = 1, \ldots, n_i\}$

Repeated Subexpressions



$$\frac{\partial z}{\partial w}$$
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

Back-prop avoids computing this twice

Implementing Back-Prop in MLP

- Forward-prop of computing a MLP computational graph with a single input x
  - $L(\hat{y}, y)$: Loss depends on the network output $\hat{y}$ and target y
  - J: total cost
  - $\Omega(\theta)$: regularizer
  - θ: all parameters(weights W and biases b)
  - $W^{(i)}, i \in \{1, \dots, l\}$: the weight matrices of the model
  - $b^{(i)}, i \in \{1, \dots, l\}$: the bias parameters of the model
- Pseudo forward-prop procedure

1: $h^{(0)} = x$
2: for $k = 1, \dots, l$ do
3: $\quad a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
4: $\quad h^{(k)} = f\left(a^{(k)}\right)$
5: endfor
6: $\hat{y} = h^{(l)}$
7: $J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Implementing Back-Prop in MLP

- Back-prop of computing a MLP computational graph with a single input x
  - Yields the gradient on the activation $a^{(k)}$ for each layer k
  - The gradients on weights and biases can be used as part of SGD update
- Pseudo back-prop procedure

1: run forward-prop

2: $g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$

3: for $k = l, l-1, \ldots, 1$ do

4: $g \leftarrow \nabla_{a^{(k)}} J = g \odot f'\left(a^{(k)}\right)$

5: $\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

6: $\nabla_{W^{(k)}} J = g h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

7: $g \leftarrow \nabla_{h^{(k-1)}} J = W^{(k)T} g$

8: endfor

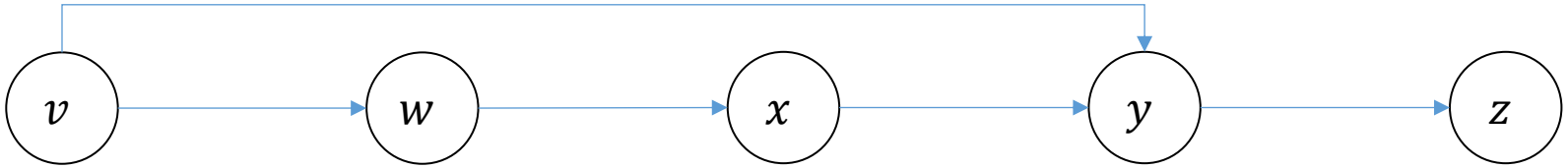
Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise)

Compute gradients on weights and biases (including the regularization term)

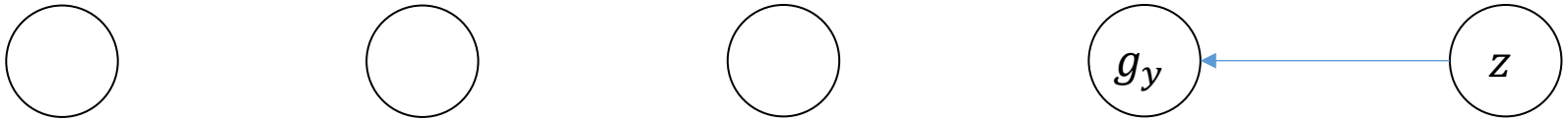
Propagate the gradients w.r.t. the next lower-level hidden layer's activations

General Back-propagation

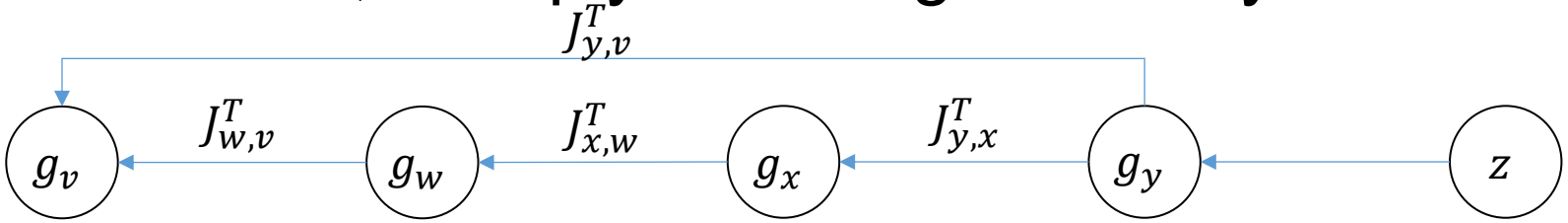
- To compute the gradient of z w.r.t all its ancestors y, x, w, v



- Compute gradient w.r.t. every parent of z



- Trave backward, multiply current gradient by Jacobian recursively



- Sum gradient from different path

General Back-propagation

- In symbols, we have

$$g_y = \nabla_y z$$
$$g_x = \nabla_x z = J_{y,x}^T g_y$$
$$g_w = \nabla_w z = J_{x,w}^T g_x$$
$$g_v = \nabla_v z = J_{w,v}^T g_w + J_{y,v}^T g_y$$

  - Where

$$J_{y,x} = \frac{\partial y}{\partial x}$$
$$J_{x,w} = \frac{\partial x}{\partial w}$$
$$J_{w,v} = \frac{\partial w}{\partial v}$$
$$J_{y,v} = \frac{\partial y}{\partial v}$$

- The spirit of this procedure can extend to case where y, x, w, v are matrices, tensors, vectors, scalers, or their mixing combinations

Implementing the Back-Prop

- Given
  - $\mathcal{G}$: the computational graph
  - V: a tensor representing variables in $\mathcal{G}$
- Assuming each variable V is associated with the following subroutines
  - $getOperation(V)$
    - Returns the operation that computes V (edges coming into V)
  - $getConsumers(V, \mathcal{G})$
    - Returns the list of variables that are children of V in the computational graph $\mathcal{G}$
  - $getInputs(V, \mathcal{G})$
    - Returns the list of variables that are parents of V in the computational graph $\mathcal{G}$

Implementing the Back-Prop

- Given
  - $\mathbb{T}$: the target set of variables whose gradients must be computed
  - z: the variable to be differentiated
- Pseudo Procedure

1: $\mathcal{G}' \leftarrow$ pruned from $\mathcal{G}$ to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$
2: Initialize $gradTable$
3: $gradTable[z] \leftarrow 1$
4: for V in $\mathbb{T}$ do
5: $buildGrad(V, \mathcal{G}, \mathcal{G}', gradTable)$
6: endfor
7: return $gradTable$ restricted to $\mathbb{T}$

Implement $buildGrad(V, \mathcal{G}, \mathcal{G}', gradTable)$

- Given
  - V: the variable whose gradient should be added to $\mathcal{G}$ and $gradTable$
  - $\mathcal{G}$: the graph to modify
  - $\mathcal{G}'$: the restriction of $\mathcal{G}$ to nodes that participate in the gradient
  - $gradTable$: a data structure mapping nodes to their gradients
- Pseudo Procedure

1: if V is in $gradTable$ then
2: return $gradTable[V]$
3: endif
4: $i \leftarrow 1$
5: for C in $getConsumers(V, \mathcal{G}')$ do
6: $op \leftarrow getOperation(\mathrm{C})$
7: $\mathrm{D} \leftarrow buildGrad(C, \mathcal{G}, \mathcal{G}', gradTable)$
8: $G^{(i)} \leftarrow op.bprop(getInputs(C, \mathcal{G}'), V, D)$
9: $i \leftarrow i + 1$
10: endfor
11: $G \leftarrow \sum_i G^{(i)}$
12: $gradTable[V] = G$
13: Insert G and the operations creating it into $\mathcal{G}$
14: Return G

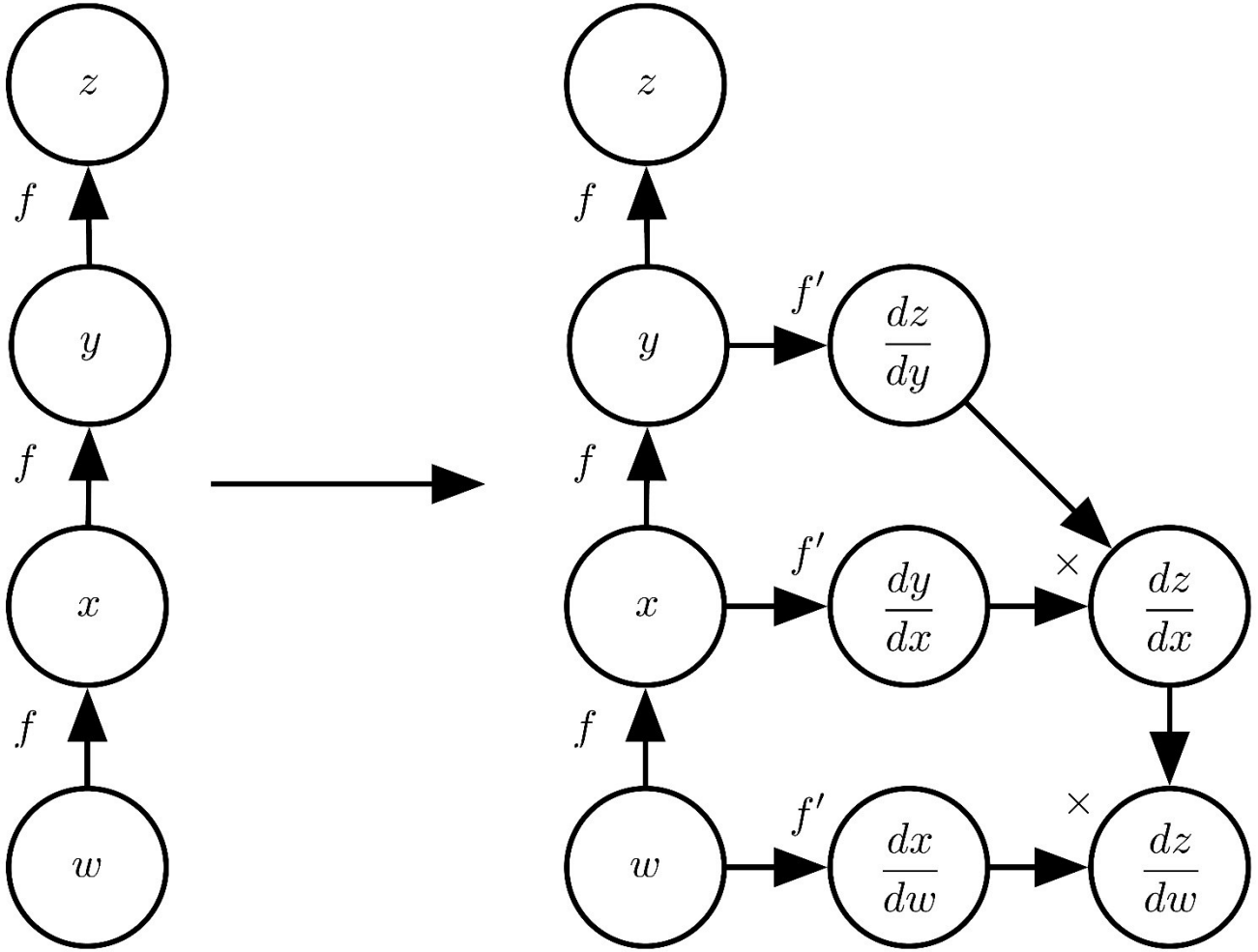
Define $op.bprop(inputs, X, G) = \sum_i (\nabla_X op.f(inputs)_i) G_i$

$op.f$: the mathematical function that the operation implements

X: the input whose gradient we wish to compute

G: the gradient on the output of the operation

Backprop in Computational Graphs



Backprop for MLP training

- Input X in mini-batch form

$$X = \begin{bmatrix} x_0^T \\ x_1^T \\ \vdots \\ x_{m-1}^T \end{bmatrix}$$

- One layer of hidden feature H with ReLU

$$U^{(1)} = XW^{(1)}$$
$$H = \max\{0, U^{(1)}\}$$

- One layer of outputs $U^{(2)}$ (before normalization)

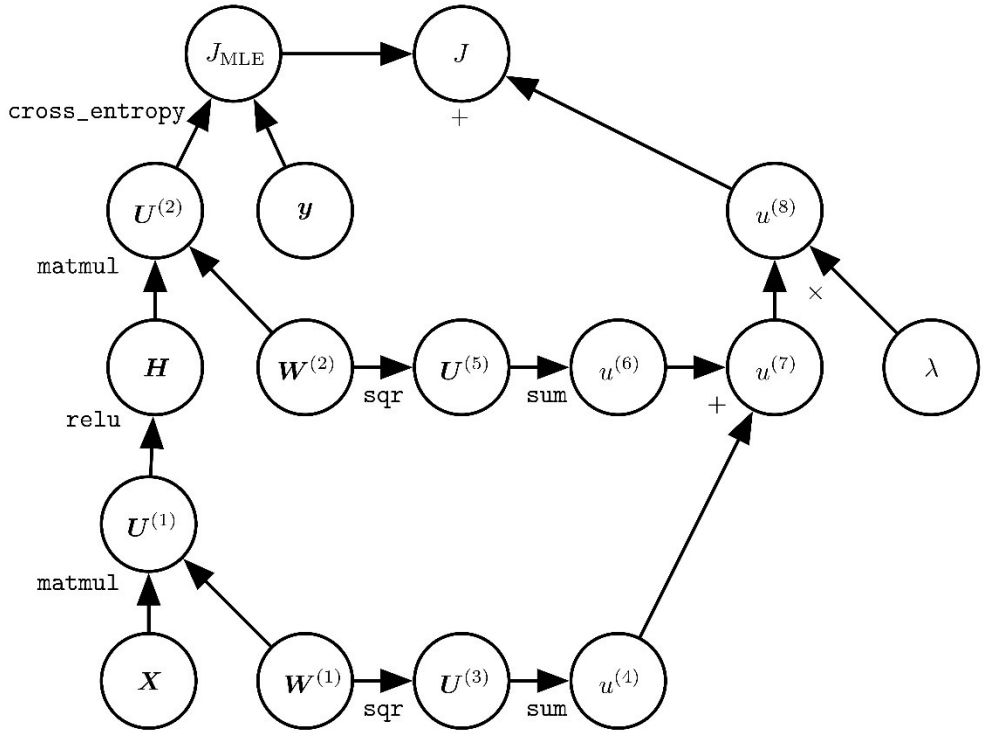
$$U^{(2)} = HW^{(2)}$$

Backprop for MLP training

- Objective: To minimize the cross-entropy with weight decay

$$J = J_{MLE} + \lambda\left(\sum_{i,j}\left(W_{i,j}^{(1)}\right)^2 + \sum_{i,j}\left(W_{i,j}^{(2)}\right)^2\right)$$

- Computational graph



Backprop for MLP training

- Backprop: To compute $\nabla_{W^{(1)}} J$ and $\nabla_{W^{(2)}} J$
  - Two path from J to the weights (only one path illustrated)
  - Assume $\nabla_{U^{(2)}} J = G$
  - Then $\nabla_{W^{(2)}} J = H^T G$
    - cf. Example 1 in Matrix Case
  - Similarly, $\nabla_H J = G W^{(2)T}$
    - cf. Example 2 in Matrix Case
  - Tracing back further, we have $\nabla_{U^{(1)}} J = G'$ by zeroing out element in $\nabla_H J$ corresponding to entries of $U^{(1)}$ less than zero
  - Again, $\nabla_{W^{(1)}} J = X^T G'$
    - cg. Example 1 in Matrix Case

Outline

- Deep Feedforward Networks
- Regularization
- Optimization
- Convolutional Neural Networks
- Practical Methods

Definition of Regularization

Regularization is **any modification** we make to a learning algorithm that is intended to **reduce its generalization error** but not its training error

Regularization

- In machine learning, we optimize a cost function defined w.r.t. the training set

$$J(\theta) = E_{x,y\sim\hat{p}_{data}} L(f(x;\theta), y)$$

  Where

  L: the per-example loss function

  $f(x;\theta)$: the model prediction

  y: the target output

  $\hat{p}_{data}$: the empirical distribution

- We however hope that doing so will minimize the expected loss over the true data-generating distribution $p_{data}(x, y)$

$$J^*(\theta) = E_{x,y\sim p_{data}}(f(x;\theta), y)$$

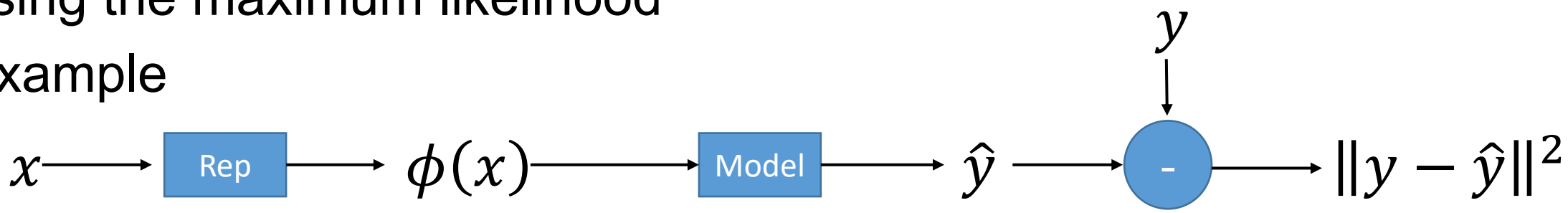
Regularization

- If the true distribution $p_{data}(x, y)$ is known
  - Minimizing $J^*(\theta)$ would become a pure optimization problem
- However, if only the empirical distribution $\hat{p}_{data}(x, y)$ over the training data is known
  - We have machine learning problem
- One central problem in machine learning is how to make an algorithm work well not just only on training data, but also on new inputs

- **Regularization** is strategies used to reduce test error, possibly at the expense of increased training error

Revisit Capacity, Underfitting and Overfitting

- To characterize analytically the relationship between a model's capacity and the phenomena of underfitting and overfitting when it is trained using the maximum likelihood
- Example



  - To predict y from x, we construct a model of the form
$$\hat{y}(x) = f(x; w) = w^T \phi(x)$$
  - And make a point estimate of the parameters w by minimizing
$$E_{x,y\sim\hat{p}_{data}} \|y - \hat{y}(x)\|^2$$

Revisit Capacity, Underfitting and Overfitting

- This is equivalent to maximizing the expected likelihood function $E_{x,y\sim\hat{p}_{data}}p(y|x)$ by assuming the following data model

$$p(y|x) = N(y; \hat{y}(x), \sigma^2)$$

- The optimal prediction which achieves the minimum expected squared generalization error

$$g^*(x) = \arg\min_{g(\cdot)} E_{x,y\sim p_{data}} \|y - g(x)\|^2$$

- is given by the conditional mean

$$g^*(x) = E_{y\sim P_{data}(y|x)}[y]$$

Revisit Capacity, Underfitting and Overfitting

- The expected generalization error of the model $\hat{y}(x)$ is then seen as the sum of Bayes error and expected error between the optimal and the model predictions

$$
\begin{aligned}
& E_{x,y\sim p_{data}} \|y - \hat{y}(x)\|^2 \\
&= E_{x,y\sim p_{data}} \|y - g^*(x) + g^*(x) - \hat{y}(x)\|^2 \\
&= \boxed{E_{x,y\sim p_{data}} \|y - g^*(x)\|^2} + E_{x\sim p_{data}} \|g^*(x) - \hat{y}(x)\|^2
\end{aligned}
$$

Bayes error

- Where the cross-term

$$
\begin{aligned}
& E_{x,y\sim p_{data}} \left[2\left(y - g^*(x)\right)\left(g^*(x) - \hat{y}(x)\right)\right] \\
&= E_{x\sim p_{data}} E_{y\sim p_{data}(y|x)} \left[2\left(y - g^*(x)\right)\left(g^*(x) - \hat{y}(x)\right)\right] \\
&= E_{x\sim p_{data(x)}} \left[2E_{y\sim p_{data}(y|x)} [y - g^*(x)]\left(g^*(x) - \hat{y}(x)\right)\right] \\
&= 0
\end{aligned}
$$

Bayes Error and Expected Error

- The Bayes error represents the minimum achievable value of the expected generalization error
  - Arises from the intrinsic noise on data
  - Independent of $\hat{y}(x)$ and training data

- On the other hand, the expected error between the optimal and the model predictions has to do with training data
  - Because $\hat{y}(x)$ is obtained by making a point estimate of w based on a particular training data set $\mathcal{D} = \{X^{(train)}, y^{(train)}\}$

$$E_{x \sim p_{data}} \| g^*(x) - \hat{y}(x) \|^2$$

Revisit Capacity, Underfitting and Overfitting

- Assume we are concerned with how the model performs over an ensemble of training data sets
  - Denote the model $\hat{y}(x)$ trained with a particular data set $\mathcal{D}$ as $\hat{y}(x;\mathcal{D})$
- For a given x, we then evaluate the expected error between optimal the model predictions w.r.t. the distribution of training data to be

$$E_{\mathcal{D}}\|g^*(x) - \hat{y}(x;\mathcal{D})\|^2$$
$$= E_{\mathcal{D}}\left\|g^*(x) - E_{\mathcal{D}}\left(\hat{y}(x;\mathcal{D})\right) + E_{\mathcal{D}}\left(\hat{y}(x;\mathcal{D})\right) - \hat{y}(x;\mathcal{D})\right\|^2$$
$$= \underbrace{E_{\mathcal{D}}\left\|g^*(x) - E_{\mathcal{D}}\left(\hat{y}(x;\mathcal{D})\right)\right\|^2}_{(bias)^2} + \underbrace{E_{\mathcal{D}}\left\|E_{\mathcal{D}}\left(\hat{y}(x;\mathcal{D})\right) - \hat{y}(x;\mathcal{D})\right\|^2}_{variance}$$

  - Where the cross-term is again computed to be zero

Revisit Capacity, Underfitting and Overfitting

- The $(bias)^2$ represents the extent to which the average model prediction over all training data set differs from the optimal prediction
- The variance measures the extent to which the model $y(x;\mathcal{D})$ is sensitive to the particular choice of training set
- Both term can be further averaged over different $x's$ to obtain the expected generalization error of the model $\hat{y}(x)$

$$Bayes\ error + E_{x\sim p_{data}}[(bias)^2] + E_{x\sim p_{data}}[variance]$$

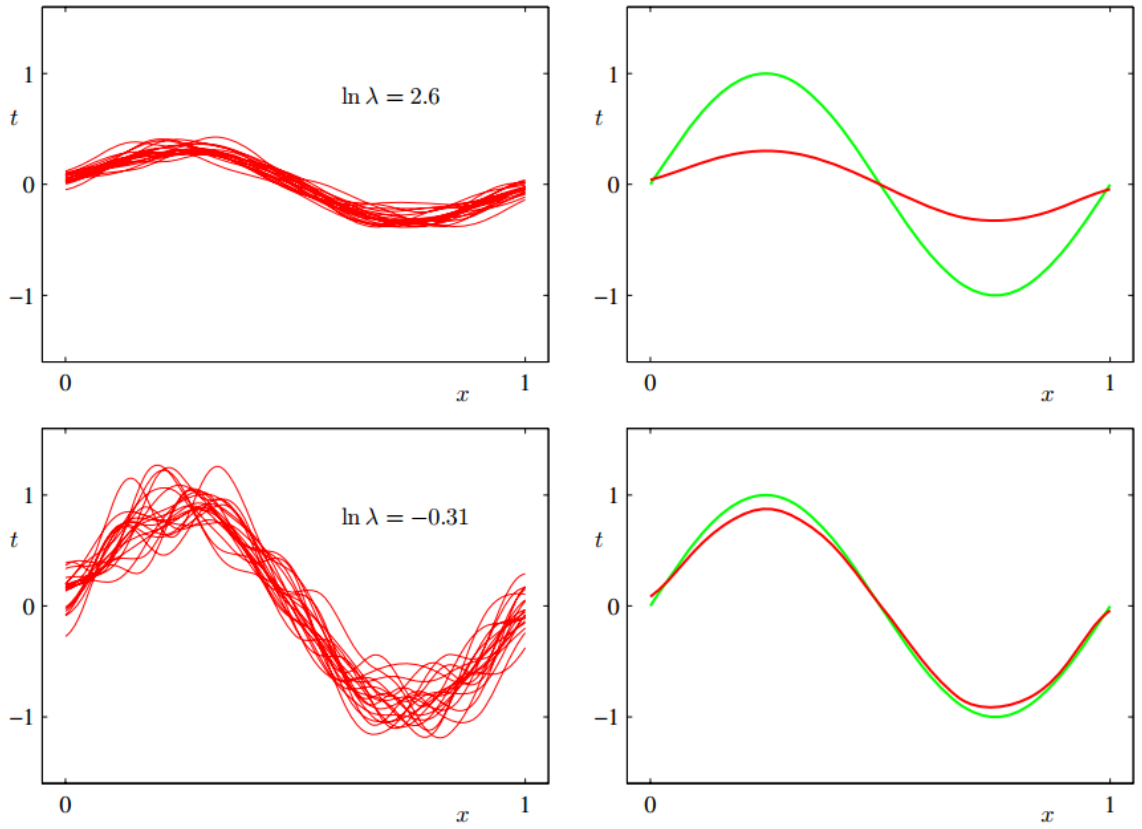
Fitting Sinusoidal Functions

- Given
  - Data: $y = \sin(2\pi x) + \epsilon,\ p(\epsilon) = \mathcal{N}(\epsilon; 0, \sigma^2)$
  - Model: $\hat{y} = w^T \phi(x),\ \phi(x)$ is a Gaussian basis
  - 100 training data set, each have 25 data point (x, y)
- Training

$$\min E_{x,y\sim\hat{p}_{data}} \|y - \hat{y}(x)\|^2 + \lambda w^T w$$

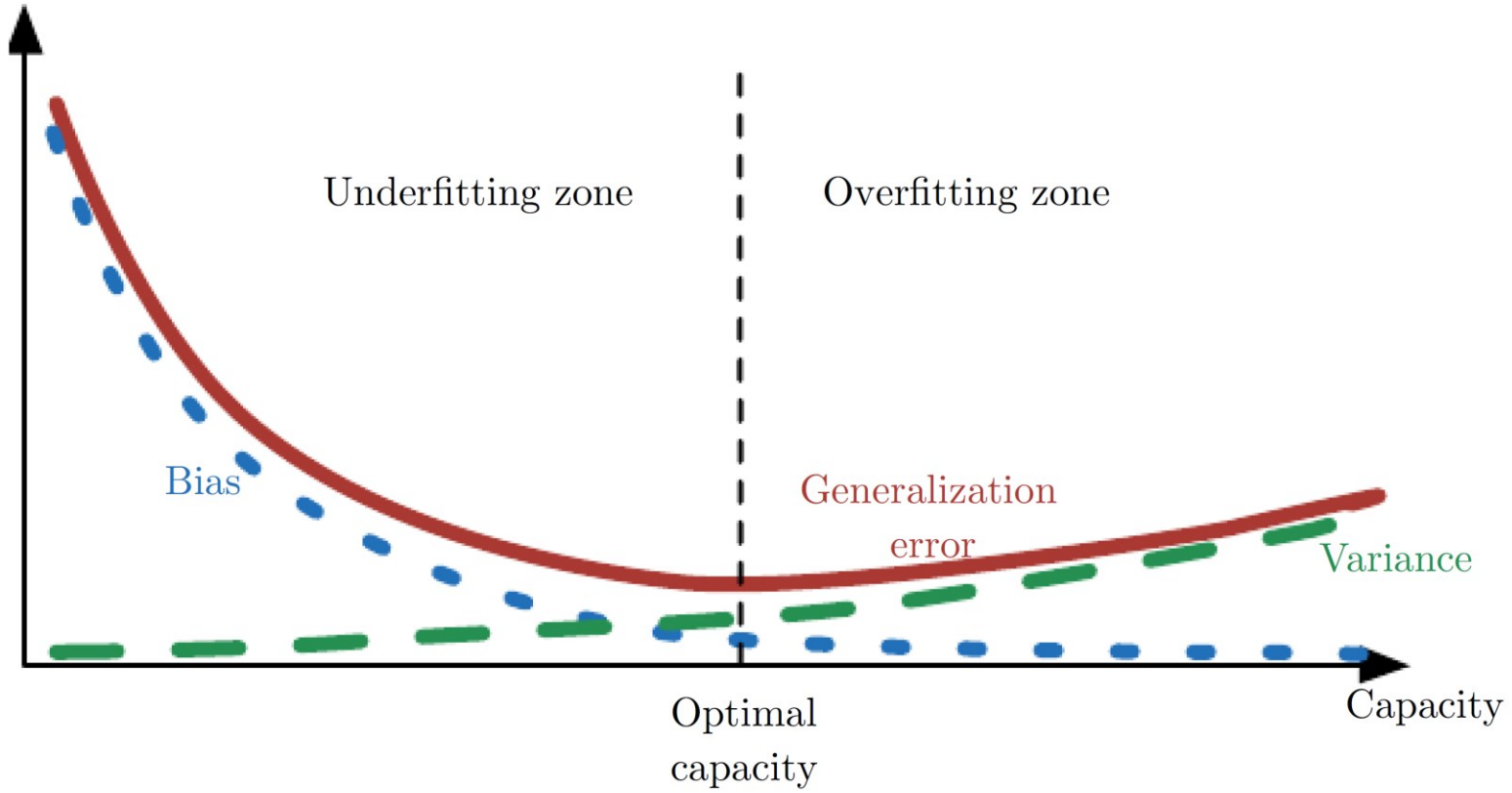
Fitting Sinusoidal Function

Left
$\hat{y}(x;\mathcal{D})$ with different training sets



Right
$g^*(x) = \sin(2\pi x)$ (Green)
$E_{\mathcal{D}}[y(x;\mathcal{D})]$ (Red)

Recap: Trade off Bias and Variance



In general, models of high capacity have low bias and high variance, whereas models of low capacity have high bias and low variance

Parameter Norm Penalties

- Limiting the model capacity by adding a norm penalty $\Omega(\theta)$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

  - Where X, y are training data and $\alpha \in [0, \infty)$ weights the relative contribution of the norm penalty to the objective function
- Generally, for neural networks, only the weights w of the affine transformation $w^T x + b$ are penalized
  - With b often left unregularized
  - Because regularizing the bias can introduce a significant amount of underfitting
    - E.g., in the linear regression problem
- We denote w weights as that should be regularized
  - Whereas θ are all the parameters $\{w. b\}$

L^2 Regularization

- The L^2 parameter regularization drive the weights closer to the origin by adding a L^2-norm penalty $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$ (i.e., weight decay)

$$\tilde{J}(w; X, y) = J(w; X, y) + \frac{\alpha}{2} w^T w$$

- The gradient $\tilde{J}(w, X, y)$ w.r.t w is

$$\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \alpha w$$

- To gain insight into the behavior of weight decay, we make a quadratic approximation to J around $w^* = \arg\min_w J(w)$

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w - w^*)^T H (w - w^*)$$

  - Where H is the Hessian matrix of J evaluated as w^*

L^2 Regularization

- We then solver for the minimum of $\tilde{J}(w; X, y)$ by substituting $\hat{J}$ for J, and setting to zero its gradient w.r.t w

$$\nabla_w \tilde{J}(w; X, y) \approx H(w - w^*) + \alpha w = 0$$

- The regularized solution $\tilde{w}$ is given by

$$\tilde{w} = (H + \alpha I)^{-1} H w^*$$

- Going further, we know that H must have the factorization

$$H = Q \Lambda Q^T$$

  - Because the Hessian matrix is real and symmetric, and is positive semi-definition when evaluated at w^*

- We have

$$\begin{aligned} \tilde{w} &= (Q \Lambda Q^T + \alpha I)^{-1} Q \Lambda Q^T w^* \\ &= (Q(\Lambda + \alpha I) Q^T)^{-1} Q \Lambda Q^T w^* \\ &= Q(\Lambda + \alpha I)^{-1} \Lambda Q^T w^* \end{aligned}$$

- From the above, the component of w that is aligned with the i-th eigenvector is re-scaled by $\frac{\lambda_i}{\lambda_i + \alpha}$

L^2 Regularization

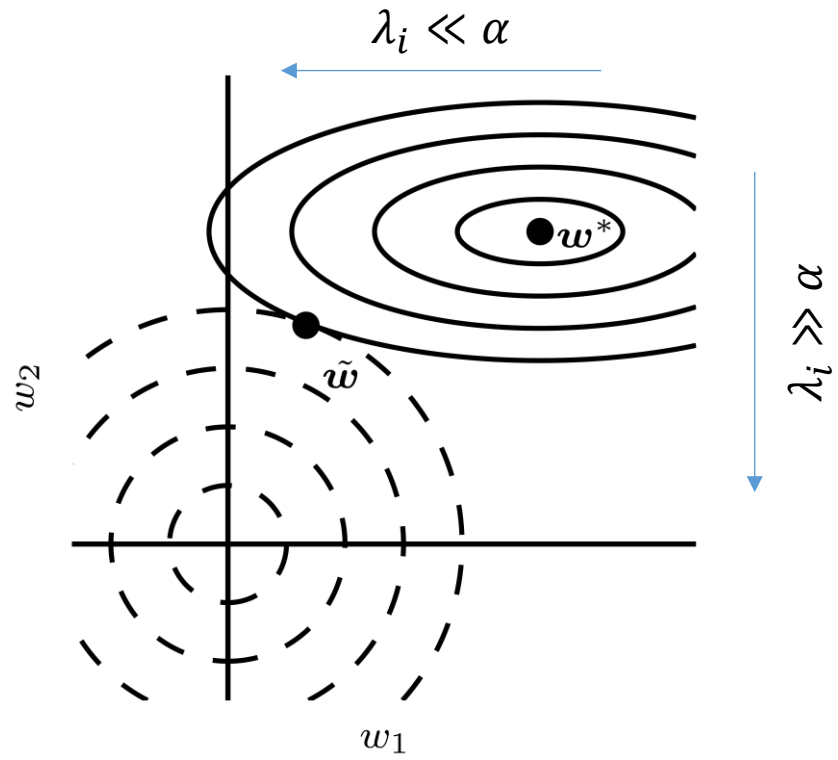
- Recall that

$$J(W; X, y)$$
$$\approx J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*)$$
$$= J(w^*) + \frac{1}{2}(w - w^*)^T Q \Lambda Q^T (w - w^*)$$

  - where along directions corresponding to large λ_i a further deviation from w^* contributes significantly to increasing the objective function

Effect of L^2 Regularization

- The effect of L^2 regularization is to decay away components of w^* along unimportant directions with $\lambda \ll \alpha$



L^1 Regularization

- Another popular parameter norm regularization is to add L^1-norm penalty $\Omega(\theta) = \|w\|_1 = \sum|w_i|$

$$\tilde{J}(w; X, y) = J(w; X, y) + \alpha\|w\|_1$$

- As the L^2 regularization, we hope to analyze the effect of L^1 regularization by making a quadratic approximation to J at w^*

$$\tilde{J}(w; X, y) \approx J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) + \alpha\|w\|_1$$

- It is however noticed that the full general Hessian does not admit a clean algebraic solution to the following problem

$$\nabla_w \tilde{J}(w; X, y) \approx H(w - w^*) + \alpha\, sign(w) = 0$$

L^1 Regularization

- We then make a further assumption that H is diagonal

$$H = \begin{bmatrix} H_{1,1} & 0 & \cdots & 0 \\ 0 & H_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & H_{n,n} \end{bmatrix}$$

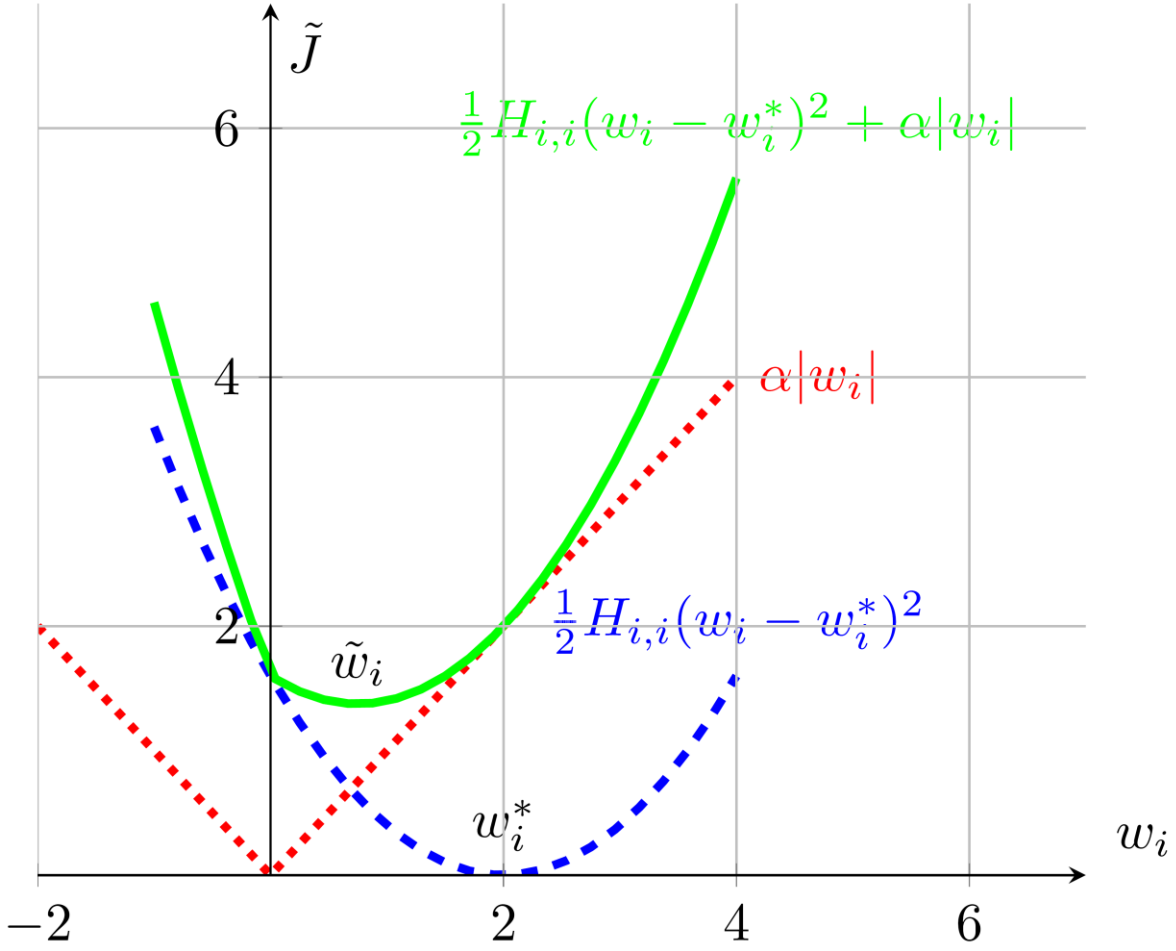
  - To arrive at

$$\tilde{J}(w; X, y) \approx J(w^*) + \sum_i \left[\frac{1}{2} H_{i,i} (w_i - w_i^*)^2 + \alpha |w_i| \right]$$

- Without loss of generality, let us assume $w_i^* > 0$
  - It can then be seen that the optimal w_i which minimizes $\tilde{J}$ lies in $[0, w_i^*]$

L^1 Regularization

It can then be seen that the optimal w_i which minimizes $\tilde{J}$ lies in $[0, w_i^*]$



L^1 Regularization

- Setting to zero the partial derivative of $\tilde{J}$ w.r.t. w_i yields

$$w_i = w_i^* - \frac{\alpha}{H_{i,i}}$$

- The regularized solution is then given by

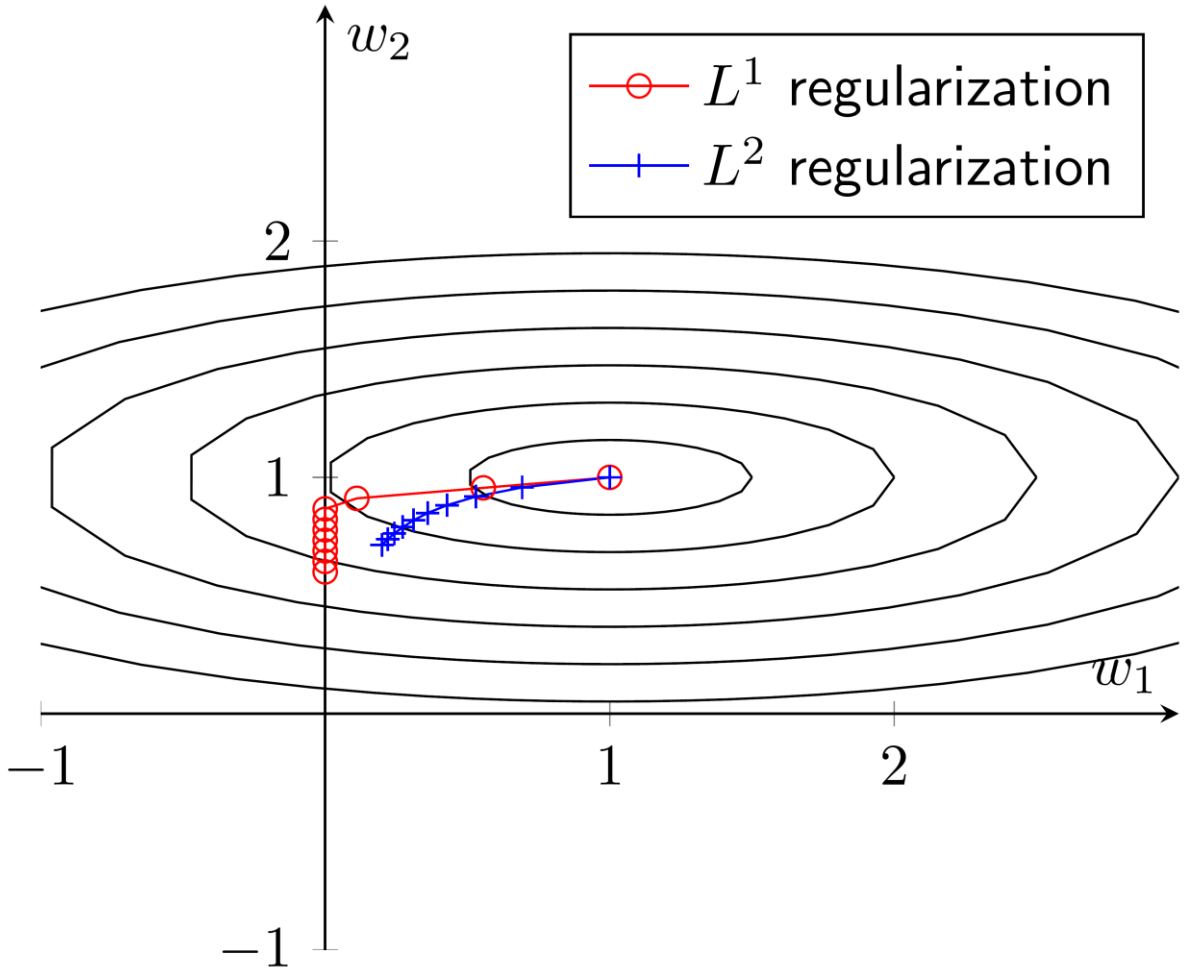
$$\widetilde{w}_i = \begin{cases} w_i^* - \dfrac{\alpha}{H_{i,i}}, & if\ w_i^* \geq \dfrac{\alpha}{H_{i,i}} \\ 0, & otherwise \end{cases}$$

- It is seen that L^1 regularization results in a solution that is more sparse
  - i.e., having more zero weights
  - A similar result occurs when $w_i^* < 0$
- In contrast, L^2 regularization in the present case does not cause the parameter to become sparse

$$\widetilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^*$$

- Lease absolute shrinkage and selection operator (LASSO): a feature selection mechanism based on L^1 penalty + linear model + least-square cost

L^1 Regularization vs. L^2 Regularization



Norm Penalty as Constrained Optimization

- Regularized training problem of minimizing $\tilde{J}$

$$arg\min_{\theta} \tilde{J}(\theta; X, y) = arg\min_{\theta} J(\theta; X, y) + \alpha\Omega(\theta)$$

  - Can be thought of constrained optimization with a weight constraint

$$arg\min_{\theta} J(\theta; X, y) \ \ s.t. \Omega(\theta) \leq K$$

- To solve the problem, we construct the Lagragian function

$$\mathcal{L}(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k)$$

- The solution is then given by

$$\theta^* = arg\min_{\theta} \max_{\alpha, \alpha > 0} \mathcal{L}(\theta, \alpha; X, y)$$

- When α is fixed at its optimal value α^*, θ is found by minimizing

$$J(\theta; X, y) + \alpha^*\Omega(\theta)$$

  - Which is exactly the same form as $\tilde{J}$

Norm Penalty as Constrained Optimization

- The optimal solution θ^* must satisfy

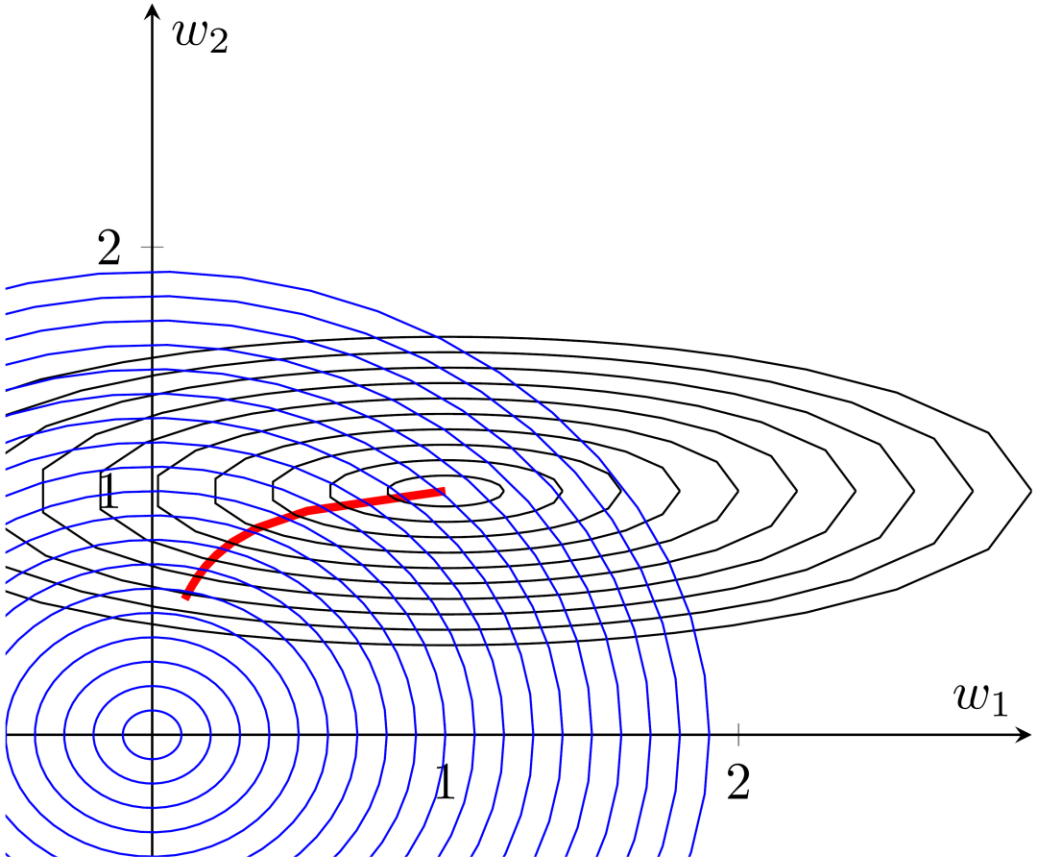
$$\nabla_\theta J(\theta^*; X, y) = -\alpha^* \nabla_\theta \Omega(\theta^*)$$

  - Note that the value of α^* does not directly tell us the value of k

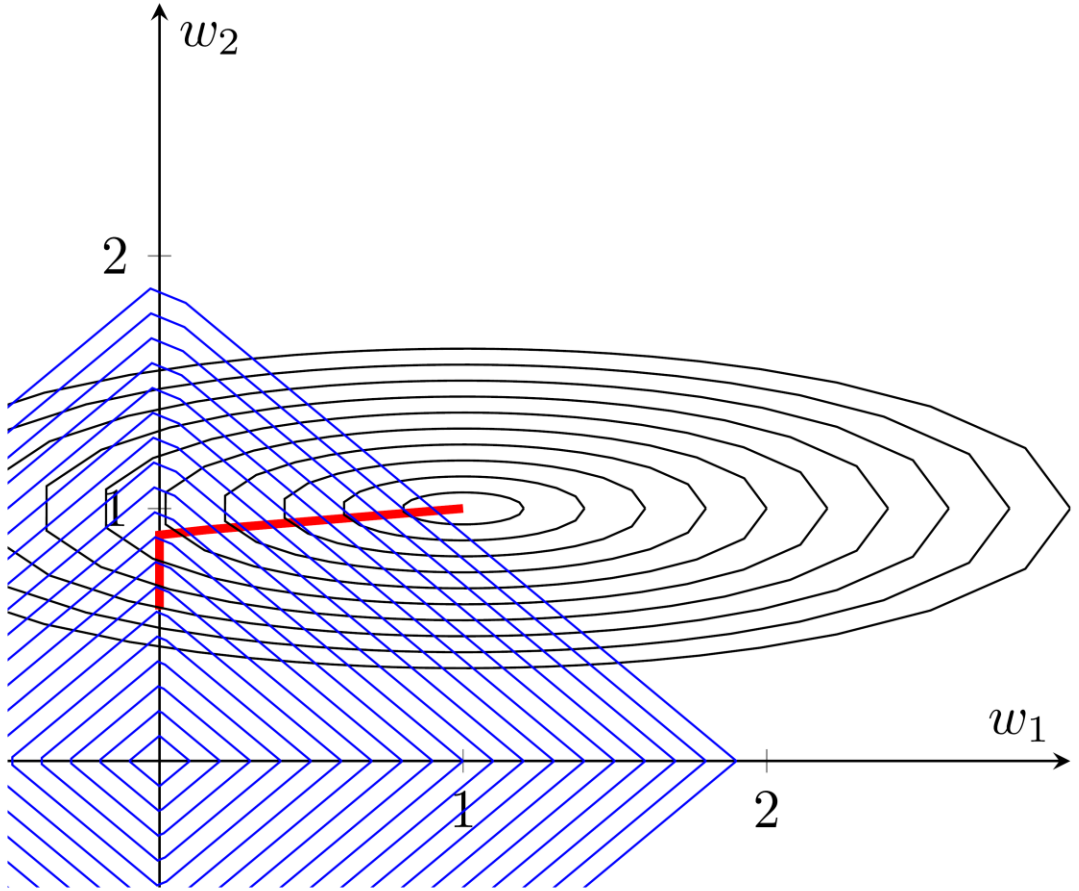
- Some use explicit constraints rather then penalties
  1. Take a step downhill on $J(\theta)$
  2. Project θ back to the set $\{\theta : \Omega(\theta) < k\}$
  3. Repeat step 1 and 2 until some stopping criterion is satisfied

Trajectory of L^1/L^2 Regularization

L^2 Regularization



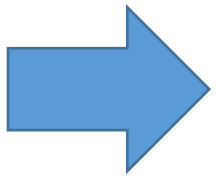
L^1 Regularization



Data Augmentation

- Idea: To add fake data to make the model generalize better
- Effective for some specific task
  - e.g. image recognition
  - Translation, rotation, scaling, etc. of training images
  - Noise injection in inputs, hidden units, outputs and weights
- Not as readily application to many other task
  - E.g. generate new fake data for density estimation

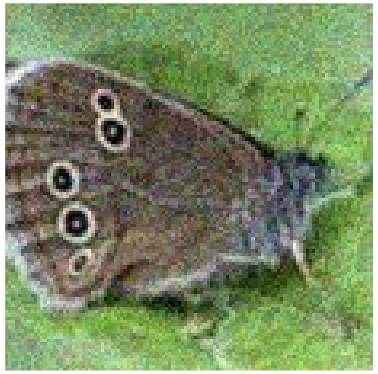
Dataset Augmentation



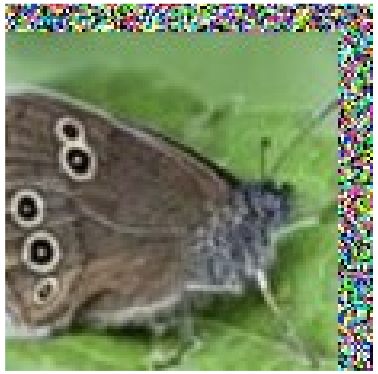
Affine Distortion

Horizontal Flip

Noise



Random Translation



Elastic Deformation

Hue Shift

Adversarial Training

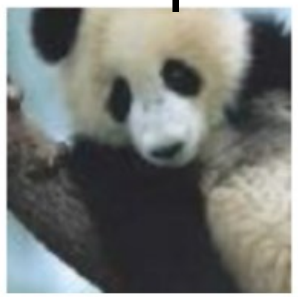
- Idea: To encourage the model $\hat{y}(x)$ to be locally constant in the vicinity of training data x by including adversarial example for training

$$x' \to x, \qquad \hat{y}(x') \to \hat{y}(x)$$

- This can be easily violated with simple linear model $\hat{y}(x) = w^T x$

$$|\hat{y}(x-\epsilon) - \hat{y}(x)| \le |w|^T |\epsilon| \approx \|w\|_1 c, \qquad with\ |\epsilon_i| = c$$

- Adversarial examples



$\boldsymbol{x}$
y = panda
with 57.7%
confidence

$+ .007 \times$

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
y = nematode
with8.2%
confidence

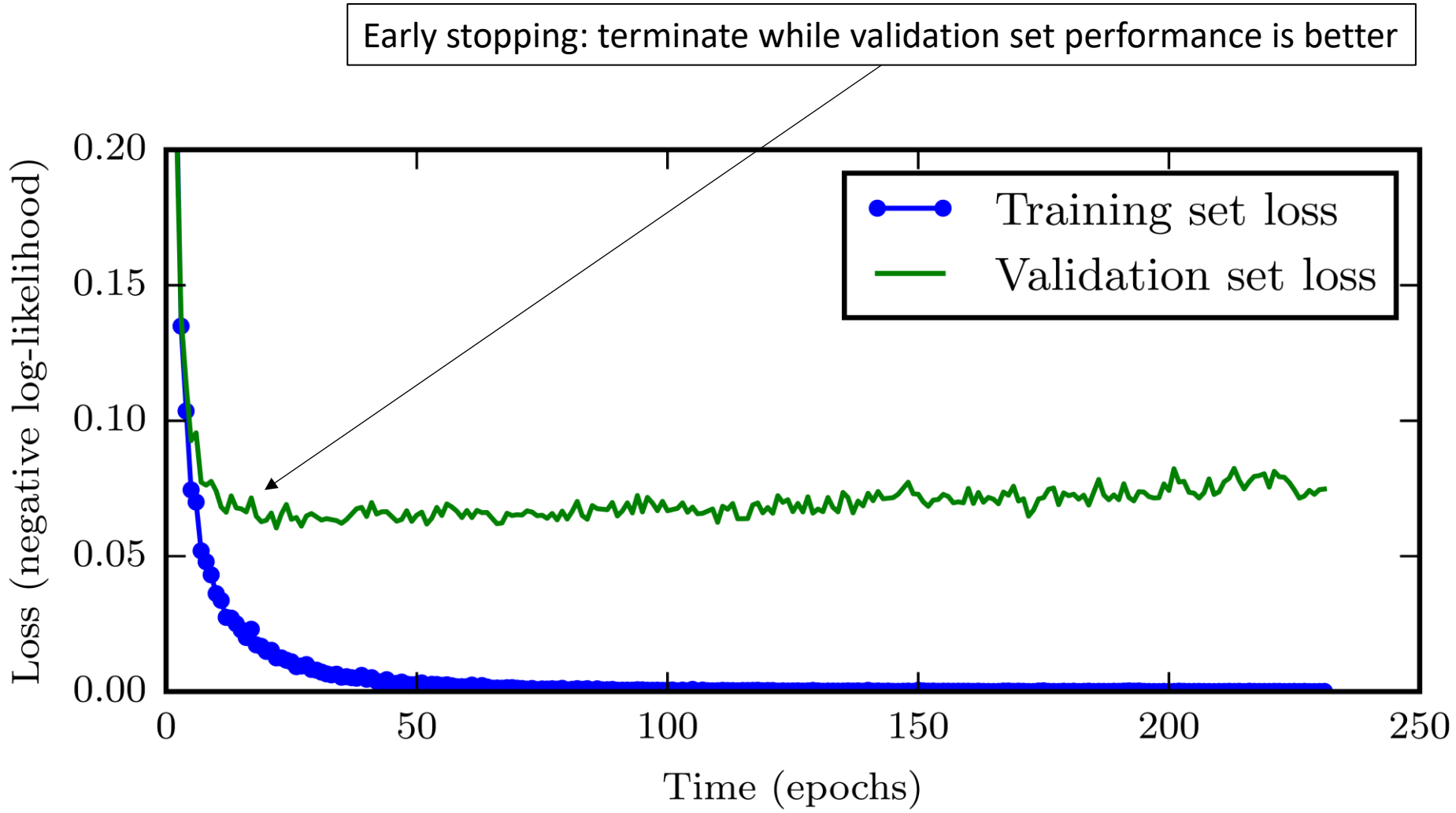
$=$

$\boldsymbol{x} + \epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
y = gibbon
with 99.3%
confidence

Early Stopping

- One effective way of determining when the training process should stop
- Error stopping procedure
  1. Run training for n steps
  2. Check validation error
  3. Store model parameters if validation error reduces
  4. Repeat 1~3 until validation error does not improve after a few trials
  5. Return model parameters

Learning Curves



Early Stopping as Regularization

- In a sense, early stopping restricts the training to a small volume of parameter space around the initial w_0
- Given the training iterations τ and learning rate ϵ, the reachable volume is determined by the product $\tau\epsilon$
- The product $\tau\epsilon$ plays a similar role to α^{-1} in L^2 regularization

$$arg\min_w J(w; X, y) + \frac{\alpha}{2} w^T w$$

  - Which is equivalent to

$$\min_w J(w; X, y) \quad s.t. w^T w \leq k_\alpha$$

- Early stopping however has the advantage of automatically determining the correct amount of regularization
  - i.e. the value of $\tau\epsilon \approx \alpha^{-1}$

Bootstrap Aggregating (Bagging)

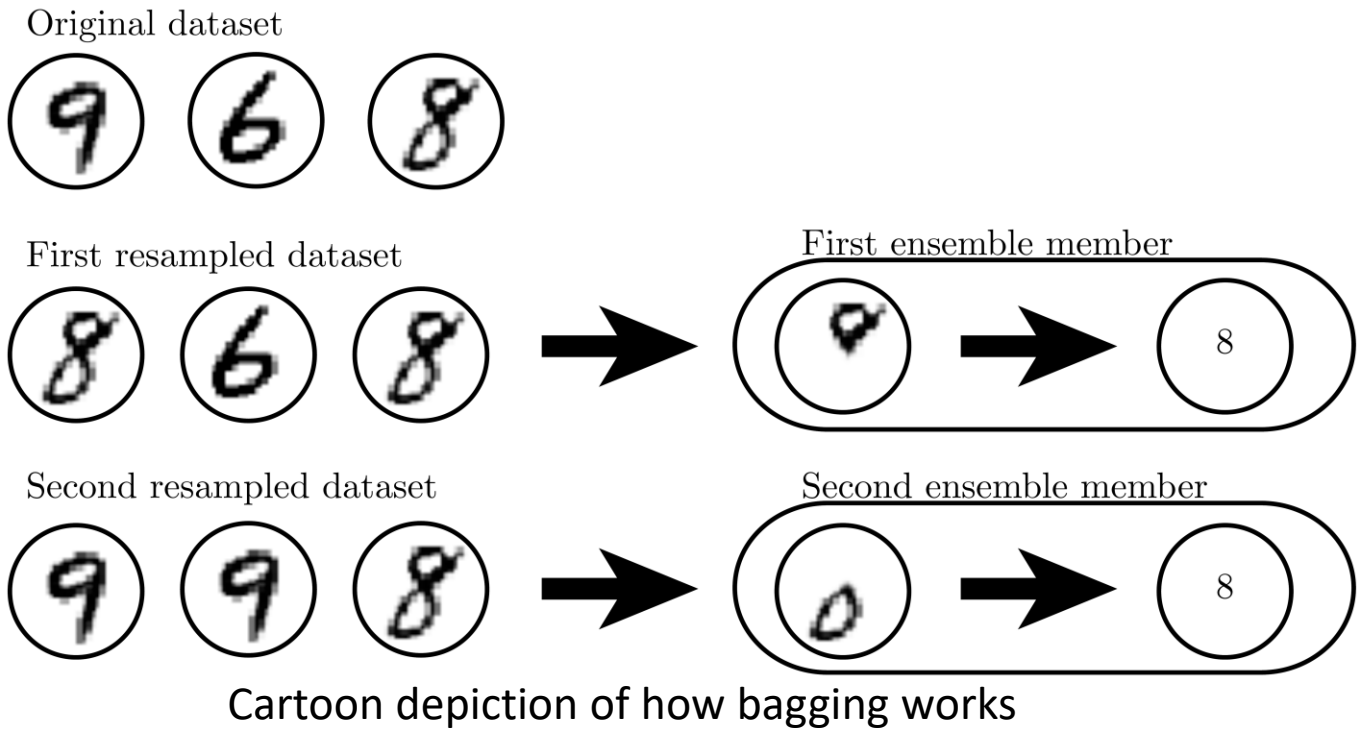
- Idea: To train several models separately and have them vote on the output for test example
  - Model averaging or ensemble methods
- Suppose that each model makes a random error ϵ_i on each example
  - with mean zero, variance $E[\epsilon_i^2] = v$, and covariance $E[\epsilon_i \epsilon_j] = c$
- Then the error made by the average prediction of all k model is

$$E\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2} E\left[\sum_i (\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j)\right] = \frac{1}{k} v + \frac{k-1}{k} c$$

- When error are highly correlated, i.e., $E[\epsilon_i \epsilon_j] = c = v$,
  - The mean square error reduces to v
  - The model averaging does not help
- When they are uncorrelated, i.e., $c = 0$
  - The mean square error is reduced by a factor of $\frac{1}{k}$

Bootstrap Aggregating (Bagging)

- In other words, **on average**, the ensemble will perform at least as well as any of its member, and significant better if the member make independent errors



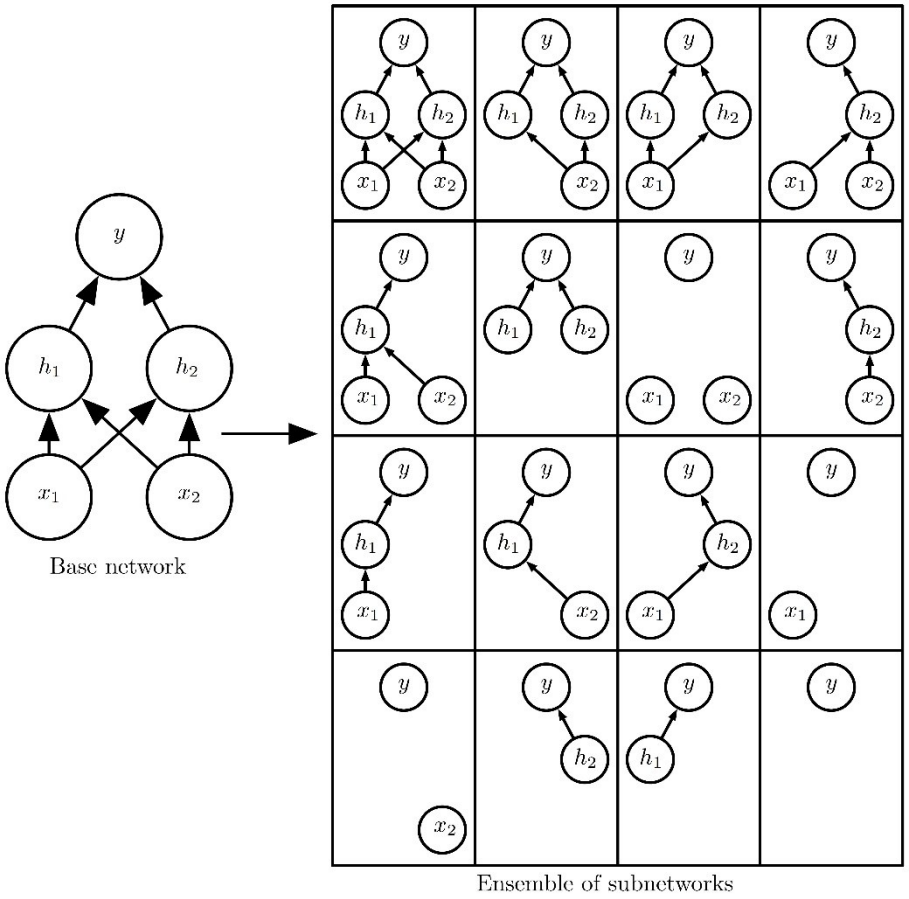
The detector learns that a loop on **top** of the digit corresponds to an 8

The detector learns that a loop on the **bottom** of the digit corresponds to an 8

Cartoon depiction of how bagging works

Dropout

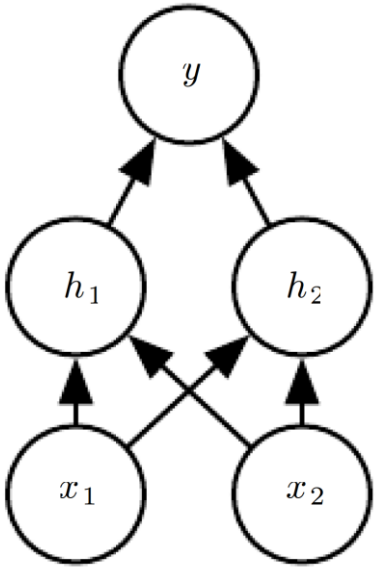
- Idea: To train a bagged ensemble of exponentially many neural networks that consist of subnetworks of a base network



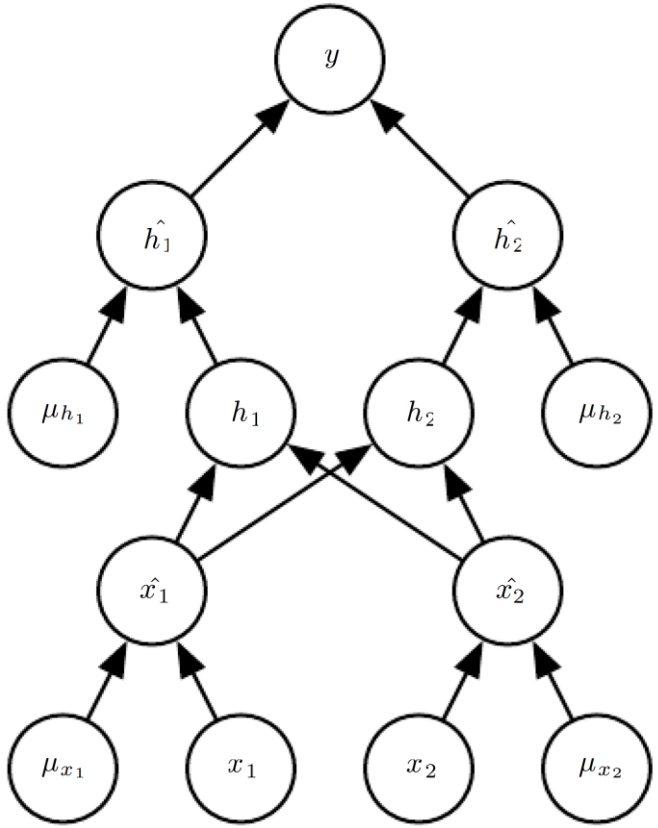
Subnetwork Construction

- To remove nonoutput units through multiplication of their output values by zero, with a mask vector μ indicating which units to keep

a feedforward network with two input units



Forward propagation with dropout



Dropout Training

- Typically, an input unit is included with probability 0.8 and a hidden unit with probability 0.5
- Dropout training
  1. In each mini-batch step, we randomly sample a binary mask μ
  2. Run forward and backward-prop
  3. Update parameters as usual
  - This account to minimizing

$$E_\mu J(\theta, \mu)$$

    - With each subnetwork inheriting a different subset of parameters from the parent neural network
      - Share parameter across all subnetworks

Dropout inference

- Ensemble inference
  - To accumulate votes from all the subnetworks

$$p(y|x) = \sum_{\mu} p(\mu)p(y|x,\mu) = E_{\mu}p(y|x,\mu)$$

  - Where $p(\mu)$ is the distribution used to sample μ at training time
- The summation over μ involves an exponential number of term, and is thus practically intractable
- One workaround is to approximate the inference with sampling

$$p(y|x) = E_{\mu}p(y|x,\mu) \approx \frac{1}{N}\sum_{i=1}^{N} p\left(y\middle|x,\mu^{(i)}\right)$$

Dropout Inference – Empirical Approach

- Another empirical approach – weight scaling inference rule
  - Allow us to approximate $p(y|x)$ in one model
  - The model with all units, but with the weights going out of unit i multiplied by the probability of including unit i
  - Motivation
    - To capture the right expected value of output from that unit
    - To make sure that the expected total input to a unit at test time is roughly the same as that at training time
  - The weight scaling inference rule is exact in some setting
    - I.e., deep networks that have hidden layer **without** non-linearities

Dropout Inference – Empirical Approach

- As an example, it is shown empirically that, in the present context

$$p(y|x) = E_\mu p(y|x,\mu) \approx c \times \sqrt[2^d]{\prod_i p(y|x,\mu^2)}$$

  - c is for normalization
  - 2^d is the number of all possible $u's$
- For a softmax regression classifier with input x and dropout

$$p(y|x;\mu) = softmax(W^T(\mu \odot x) + b)_y$$

  - With each element μ_i having an equal probability of being 0 or 1
- By an application of the geometric mean approximation, we obtain an ensemble softmax classifier with

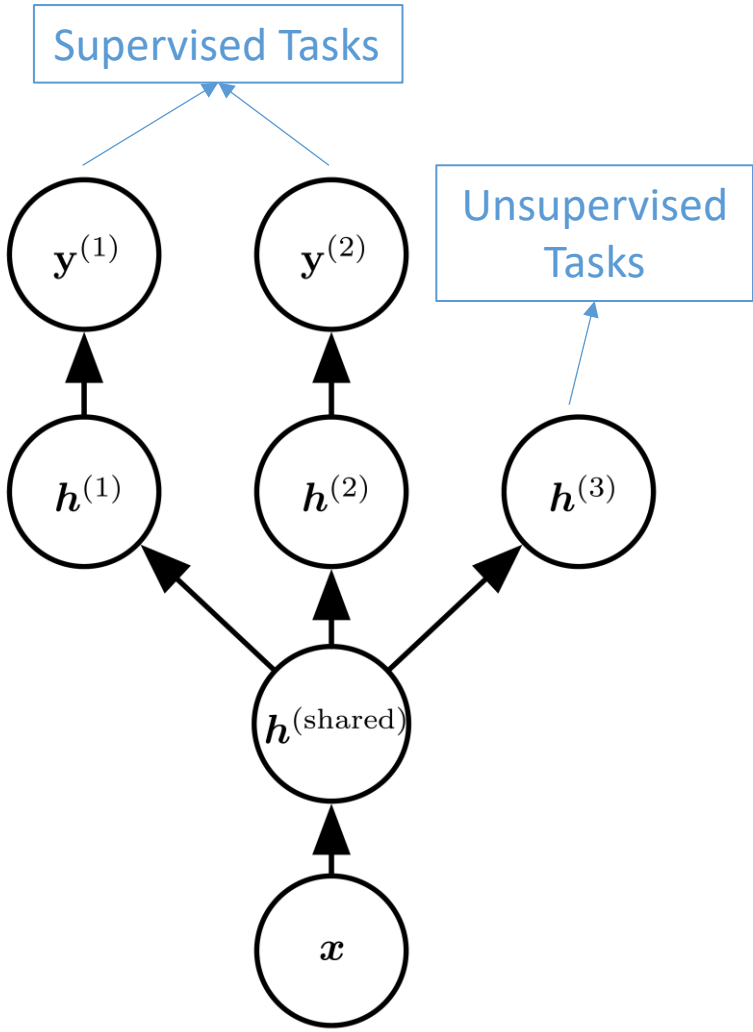
$$p(y|x) \propto \exp\left(\frac{1}{2}W_{y,:}^T w + b_y\right)$$

Dropout - Pros and Cons

- Values of dropout go beyond bagging
  - i.e. removal of hidden units is similar to adaptive deconstruction of high-level contents
- Shared hidden units learn features useful in many contexts/subnetworks
- Applicable to many types of models
- More effective than other standard regularizer
- Computationally cheap $O(n)$ for training and storage
- Increased model size needed for more capable subnetworks
- Less effective with few label training example
- E.t.c.

Multitask Learning

- Idea
  - To improve generalization by pooling examples for several tasks
- Assumption
  - There exists a common pool of factor that explain the data variations
  - Each task is linked to a subnet of these factor
- The cost function may involve both supervised and unsupervised parts



Outline

- Deep Feedforward Networks
- Regularization
- Optimization
- Convolutional Neural Networks
- Practical Methods

How Learning Differs from Pure Optimization

- In pure Optimization
  - Minimizing J is a goal in and of itself
- In most machine learning scenarios
  - We care about some performance measure P
    - Been defined with respect to the test set and may also be intractable
  - We therefore optimize P indirectly
    - Reduce a different cost function $J(\theta)$ in the hope that doing so will improve P
    - **Surrogate Loss Functions**
- Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions

Learning Optimization

- Typically, the cost function can be written as an average over the training set, such as

$$J(\theta) = E_{(x,y)\sim\hat{p}_{data}} L(f(x;\theta), y)$$

  - L: per-example loss function
  - $f(x;\theta)$: predicted output
  - $\hat{p}_{data}$: empirical distribution
  - x: input, y: target output
- To develop the unregularized supervised learning/optimization algorithm
- We would usually prefer to minimize $J^*(\theta) = E_{(x,y)\sim p_{data}} L(f(x;\theta), y)$
  - where the expectation is taken across the data generating distribution p_{data} rather than just over the finite training set

Surrogate Loss and Early Stopping

- Sometimes, the loss function we actually care about (say classification error) is not one that can be optimized efficiently
  - Typically, we optimizes a surrogate loss function instead
- Example
  - The negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss
- Training often halts while the surrogate loss function still has large derivatives(not local minimum)
  - Training halts when a convergence criterion based on early stopping
  - Early stopping based on the true underlying loss function

Batch and MiniBatch

- To minimize the loss function based on empirical distribution

$$\nabla_\theta J(\theta) = E_{x,y\sim\hat{p}_{data}} \nabla_\theta p_{model}(x, y; \theta)$$

- Computing this expectation exactly is very expensive
  - Because it requires evaluating the model on every example in the entire dataset
- In practice, we compute these expectations by randomly sampling a small number of examples from the dataset
  - A stochastic minibatch
- **Deterministic** gradient methods
  - Or **Batch** gradient method
  - Optimization algorithms that use the entire training set
  - Process all the training examples simultaneously in a large batch
- **Stochastic**
  - Using only one single example
- **Minibatch** (stochastic) method
  - Using more than 1 but less than all of the training examples

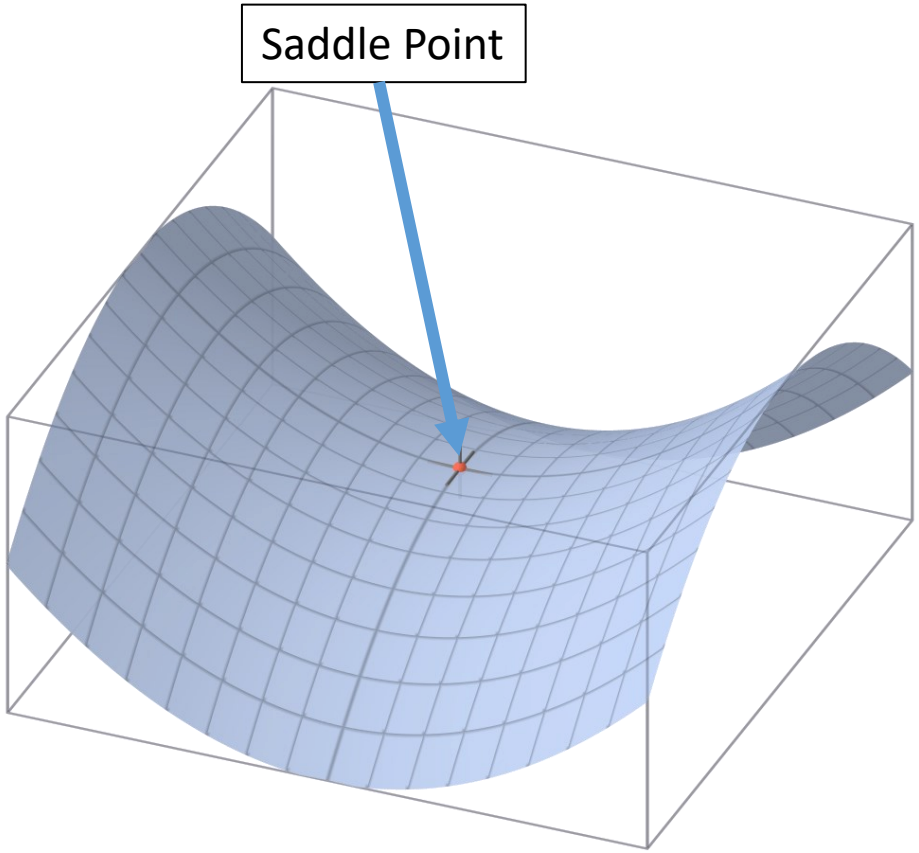
Minibatch

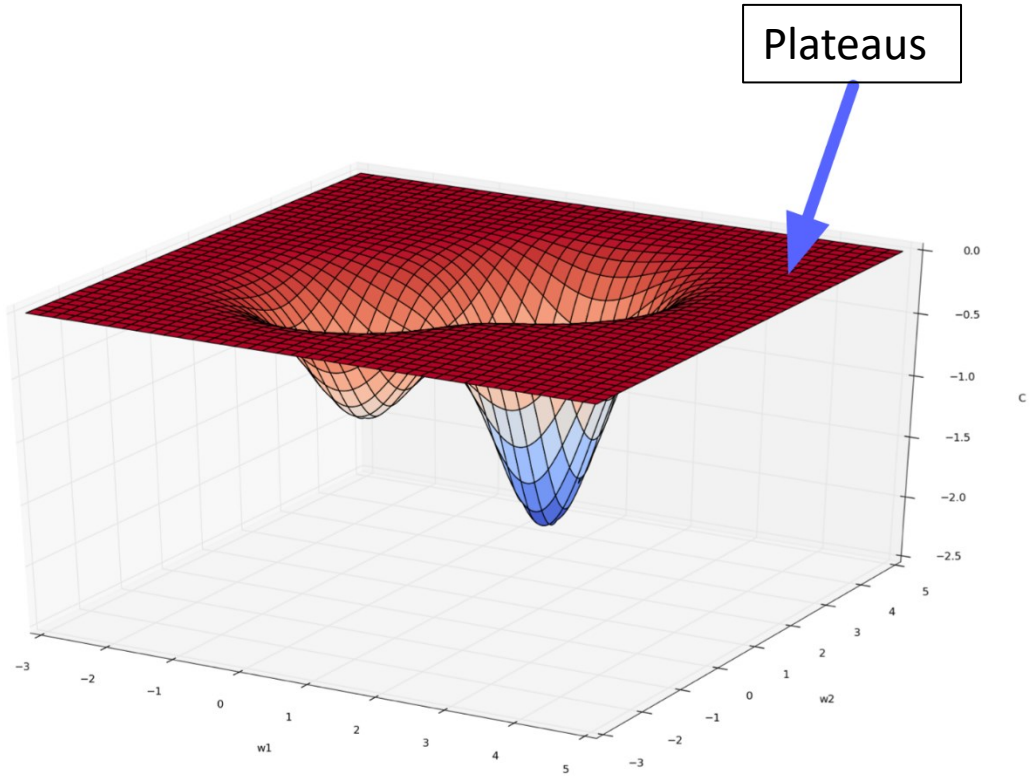
- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns
- Small batches can offer a regularizing effect
  - Perhaps due to the noise they add to the learning process
- It is crucial that the minibatches be selected randomly
  - Computing an unbiased estimate of the expected gradient from a set of samples requires that **those samples be independent**

Challenges in Neural Network Optimization

- When training neural networks, we must confront the general non-convex optimization problems
  - Ill-conditioning of the Hessian matrix H
  - Local Minima
  - Plateaus, Saddle Points and Other Flat Regions
  - Cliff and Exploding Gradient
  - Long-Term Dependencies
  - Inexact Gradients
  - Poor Correspondence between Local and Global Structure

Saddle Point and Plateaus





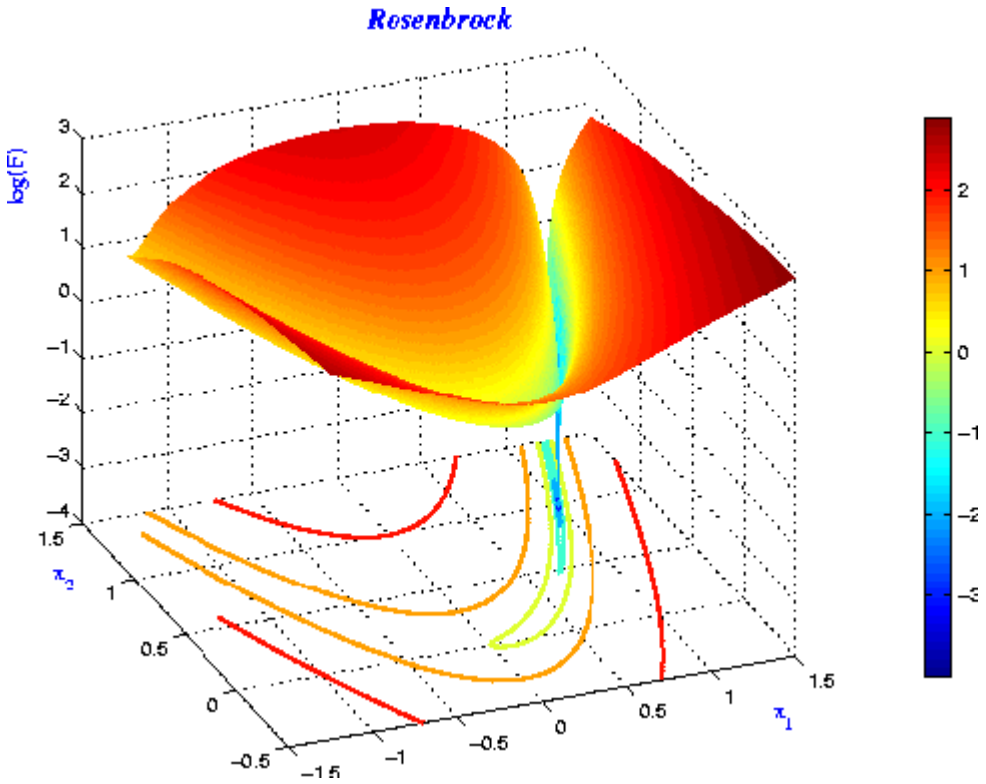
Ill-conditioning of the Hessian matrix H

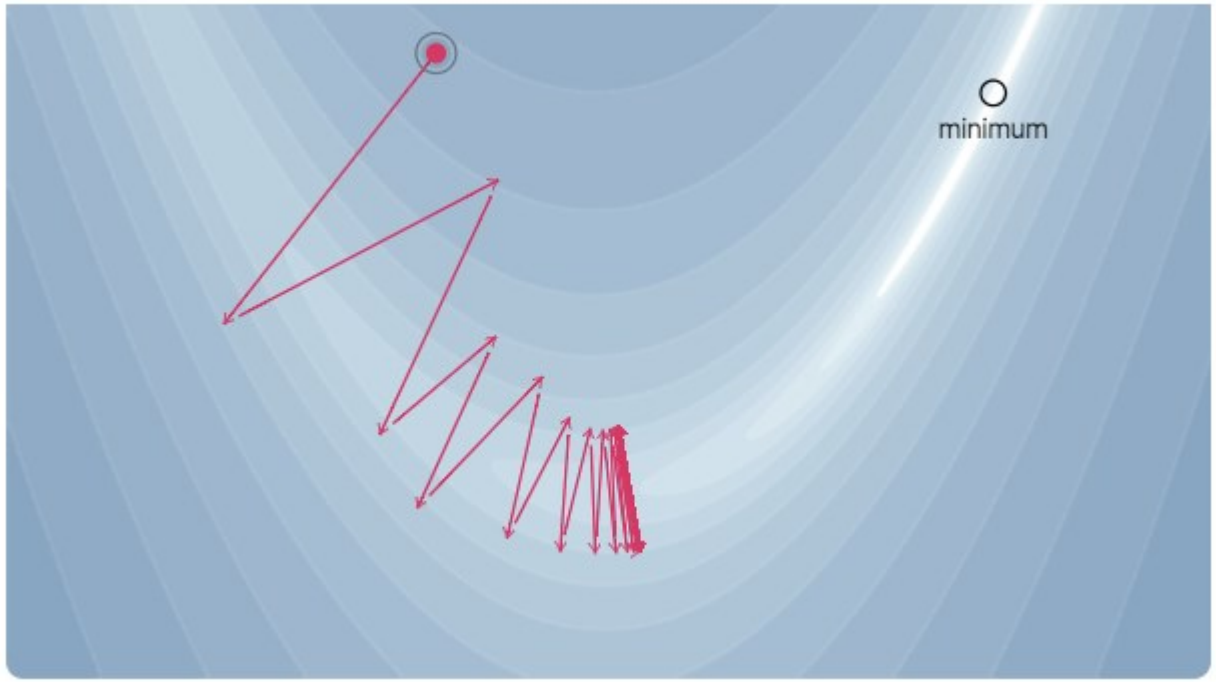
- Causing SGD to get **stuck** in the sense that even very small steps increase the cost function
- Second-order Taylor series expansion of the cost function predicts that a gradient descent step of $-\epsilon g$ will add the following amount to the cost

$$\frac{1}{2}\epsilon^2 g^T H g - \epsilon g^T g$$

- When $\frac{1}{2}\epsilon^2 g^T H g > \epsilon g^T g$
  ⇒ cost cannot further decrease
  ⇒ Ill-conditioning
  ⇒ learning becomes very slow despite the presence of a strong gradient

Ill-conditioned Curvature





Stochastic Gradient Descent (SGD)

- Given
  - Learning rate for iteration k: ϵ_k
  - Initial parameter θ
- SGD Procedure

While stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots x^{(m)}\}$,
  Compute gradient estimate $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(f\left(x^{(i)}; \theta\right), y\right)$
  Apply update: $\theta \leftarrow \theta - \epsilon \hat{g}$
Endwhile

- SGD gradient estimator introduces a source of noise - the random sampling of m training examples

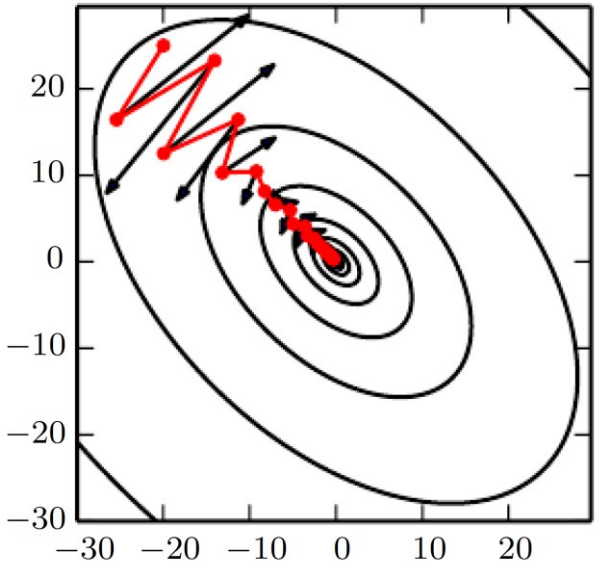
Momentum

- Accumulates an exponentially decaying moving average of past gradients and continues to move in their direction
- Update rules

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f\left(x^{(i)}; \theta \right), y^{(i)} \right) \right), \alpha \in [0,1)$$

$$\theta \leftarrow \theta + v$$

- Solving for
  - Poor conditioning of the Hessian matrix
  - Variance in the stochastic gradient



SGD with Momentum

- Given
  - Learning rate ϵ, Momentum parameter α
  - Initial parameter θ, initial velocity v
- Pseudo Procedure

1: **While** stopping criterion not met **do**
2: Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots x^{(m)}\}$,
3: Compute gradient estimate $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(f\left(x^{(i)}; \theta\right), y\right)$
4: Compute velocity update: $v \leftarrow \alpha v - \epsilon \hat{g}$
5: Apply update: $\theta \leftarrow \theta + v$
6: **Endwhile**

Parameter Initialization Strategies

- Training deep models is a sufficiently difficult task that most algorithms are strongly affected by **the choice of initialization**
- The initial point can determine
  - Whether the algorithm converges at all
  - How quickly learning converges
  - Whether it converges to a point with high or low cost
- Modern initialization strategies are **simple** and **heuristic**
- Our understanding of how the initial point affects generalization is especially primitive
  - **offering little to no guidance** for how to select the initial point
- The only property known with complete certainty is that the initial parameters need to **break symmetry** between different units
  - E.g., two hidden unit with the same activation function are connected to the same inputs
    - These units must have **different initial parameters** ⇒ random initialization of the parameters
    - Otherwise, learning will update these units in the same way⇒ **redundant units**

Parameter Initialization Strategies

- Typical parameter initialization
  - Bias ⇒ constant initialize
  - Weights ⇒ random initialize
  - Extra parameters ⇒ heuristically chosen constant
- Larger initial weights yield a stronger symmetry breaking effect
  - Avoid redundant units
  - Avoid vanishing values in forward/backward prop
  - Too large ⇒ exploding values in forward/backward prop
    - Can be mitigated by gradient clipping
    - However, saturating might lose the gradient propagation
- From the optimization perspective
  - Weights should be large enough to propagate information successfully
- From regularization concerns
  - Encourage making weights smaller

From the Gaussian Prior Viewpoint

- The use of SGD
  - Expresses a prior that the final parameters should be close to the initial parameters
- Initialize the parameter θ to θ_0 is similar to imposing aa Gaussian prior $p(\theta)$ with θ_0
  - If we choose θ_0 to be near 0
    - Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact
  - If we choose θ_0 to be larger
    - Our prior specifies which units should interact with each other, and how they should interact

Initialization of Weights

- Setting the biases to zero is compatible with most weight initialization schemes
- Some situations where we may set some biases to non-zero values
  1. If a bias is for an output unit, initialize the bias to obtain the right marginal statistics of the output
     - We assume that the initial weights are small enough that the output of **the unit is determined only by the bias**
  2. Choose the bias to be a small positive value, e.g., 0.1
     - Avoid saturating the ReLU at initialization
     - Not compatible with weight initialization schemes that do not expect strong input from the biases though.
  3. Advocate setting the bias to 1 for the forget gate of the LSTM model,
     - Sometimes a unit controls whether other units are able to participate in a function

Learning-based Initialization Methods

- Pre-training - Initialize a supervised model with the parameters learned by
  - Unsupervised model trained on the **same inputs**
  - Supervised training on a **related task**
  - Supervised training on an **unrelated task** can sometimes yield an initialization that offers faster convergence than a random initialization
- Encode information about the distribution in the initial parameters of the model
- Set the parameters to have the right scale or set different units to compute different functions from each other

Adaptive Learning Rates

- Learning rate - one of the hyperparameters that is the most difficult to set
  - Because it has a significant impact on model performance
  - Cost is often highly sensitive to some directions in parameter space and insensitive to others
- Momentum – mitigate but introducing another hyperparameter
- If we believe that the directions of sensitivity are somewhat axis-aligned
  - Separate learning rate for each parameter
  - Automatically adapt these learning rates throughout the course of learning
- The **delta-bar-delta** algorithm - an early heuristic approach to adapting individual learning rates for model parameters during training
  - If the **partial derivative of the loss**, with respect to a given model parameter, **remains the same sign**, then the learning rate should **increase**
  - If the partial derivative with respect to that parameter **changes sign**, then the learning rate should **decrease**
  - **Can only be applied to full batch optimization**

Choose the Right Optimization Algo

- **Which algorithm should one choose?**
  - Unfortunately, there is currently no consensus on this point.
  - The family of algorithms with adaptive learning rates performed fairly robustly
  - However, no single best algorithm has significantly outperforming result

- Most popular optimization algorithms actively in use
  - SGD, SGD with momentum
  - RMSProp, RMSProp with momentum
  - AdaDelta
  - Adam

AdaGrad

- Given
  - Global learning rate ϵ
  - Initial parameter θ
  - Small constant δ, e.g., 10^{-7}, for numerical stability
- Pseudo Procedure

Initialize gradient accumulation variable $r = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}, with\ target\ y^{(i)}s$
  compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Accumulate squared gradient: $r \leftarrow r + g \odot g$
  Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$, (division and square root applied element-wise)
  Apply update: $\theta \leftarrow \theta + \Delta\theta$
endwhile

- Note
  - Designed to converge rapidly when applied to a convex function
  - The accumulation of squared gradients from **the beginning of training** can result in a premature and excessive decrease in the effective learning rate
  - AdaGrad performs well for some but not all deep learning models

RMSProp

- Given
  - Global learning rate ϵ, decay rate p
  - Initial parameter θ
  - Small constant δ, e.g., 10^{-6}, used to stabilize division by small values
- Pseudo Procedure

1: Initialize gradient accumulation variable $r = 0$
2: **while** stopping criterion not met **do**
3: Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}, with\ target\ y^{(i)}s$
4: compute gradient: $g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$
5: Accumulate squared gradient: $r \leftarrow pr + (1-p)g \odot g$
6: Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$, ($\frac{1}{\sqrt{\delta+r}}$ applied element-wise)
7: Apply update: $\theta \leftarrow \theta + \Delta\theta$
8: **Endwhile**

- Note
  - Perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average
  - Shown to be an effective and practical optimization algorithm for deep neural networks

ADAM (ADAptive Moments)

- Given
  - Step size ϵ (suggest default: 0.001)
  - Exponential decay rates for moment estimates, p_1 and p_2 in $[0, 1)$ (Suggested defaults: 0.9 and 0.999 respectively)
  - Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
  - Initial parameters θ
- Pseudo Procedure

Initialize 1st and 2nd moment variables $s = 0, r = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$, *with target* $y^{(i)}s$
  Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $s \leftarrow p_1 s + (1 - p_1) g$
  Update biased second moment estimate: $r \leftarrow p_2 r + (1 - p_2) g \odot g$
  Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1 - p_1^t}{}_s$
  Correct bias in second moment: $\hat{r} \leftarrow \frac{}{1 - p_2^t}$
  Compute update: $\Delta\theta = - -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$, (operations applied element-wise)
  Apply update: $\theta \leftarrow \theta + \Delta\theta$
Endwhile

ADAM (ADAptive Moments)

- A variant on the combination of RMSProp and momentum
  - Momentum is incorporated directly as an estimate of the first order moment of the gradient with exponential weighting
    - Does not have a clear theoretical motivation
  - Includes bias corrections to the estimates of both the first-order moments and the second-order moments
    - To account for their initialization at the origin
- Adam is generally regarded as being fairly robust to the choice of hyperparameters
  - The learning rate sometimes needs to be changed from the suggested default

Batch Normalization

- Very deep models involve the composition of several functions or layers
  - If not normalized, makes it very hard to choose an appropriate learning rate
- Batch normalization reparametrizes the model to make some units always be standardized
  - Significantly reduces the problem of coordinating updates across many layers
  - Can be applied to any input or hidden layer in a network
- Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning
  - But allows the relationships between units and the nonlinear statistics of a single unit to change

Batch Normalization

- **Input:** Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
  Parameters to be learned: γ, β

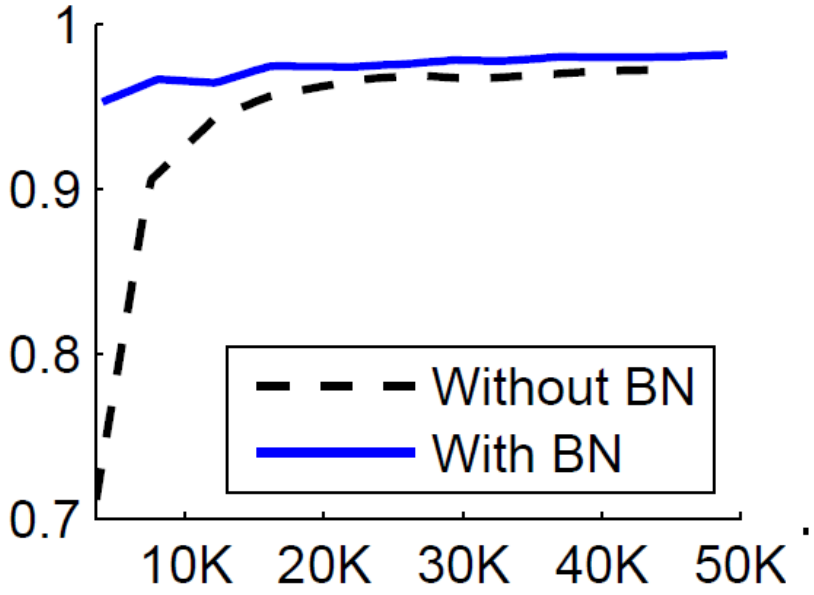
  Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

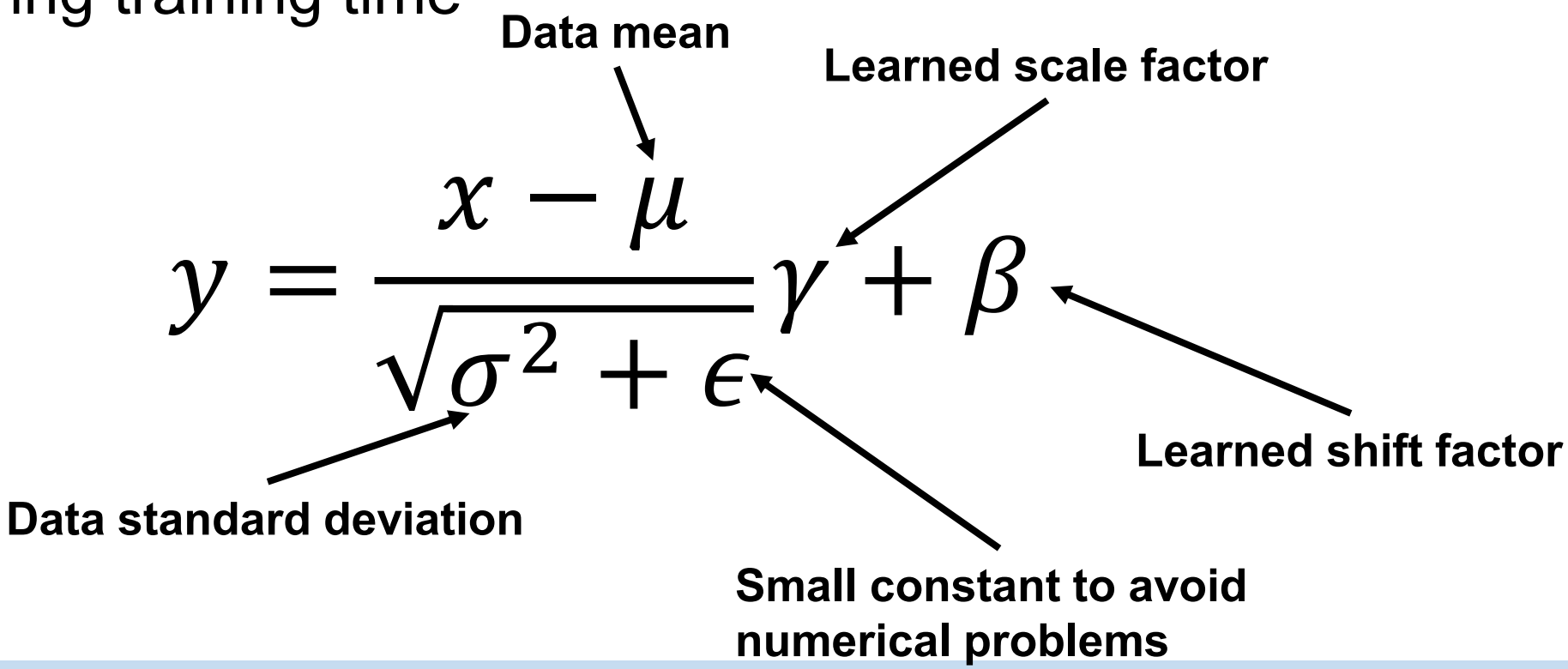
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



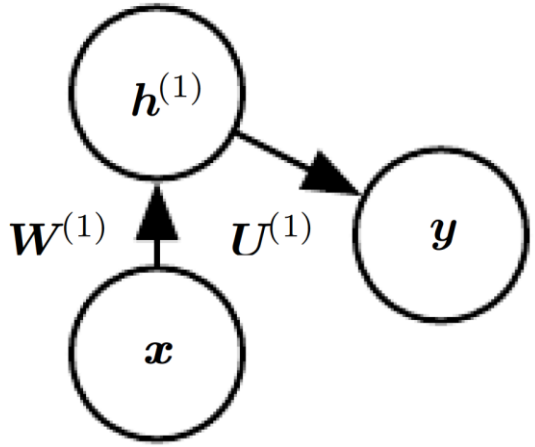
BN Layer Implementation

- The normalized value is further scaled and shifted, the parameters of which are learned from training
- At test time, μ and σ may be replaced by running averages that were collected during training time

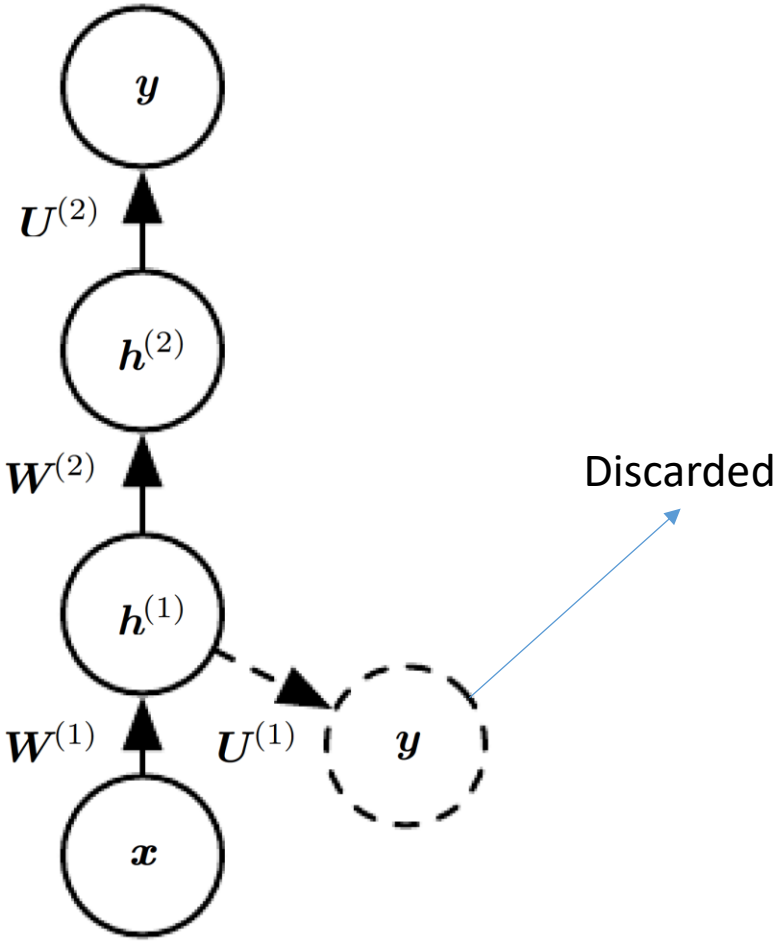


Greedy Supervised Pretraining

training a shallower architecture first



Continue training a deeper network based on $h^{(1)}$



Greedy Supervised Pretraining

- Why would greedy supervised pretraining help?
  - It helps to provide better guidance to the intermediate levels of a deep hierarchy
- In general, pretraining may help both in terms of optimization and in terms of generalization
- Same concept – **Transfer Learning**
  1. Pretrain a deep convolutional net with n layers of weights on a set of tasks
  2. Initialize a same-size network with the first k layers of the first net
  3. All the layers of the second network are then jointly trained to perform a different set of tasks

Designing Models to Aid Optimization

- Many improvements in the optimization of deep models have come from designing the models to be easier to optimize

*In practice, it is more important to **choose a model family that is easy to optimize** than to use a powerful optimization algorithm*

- Most of the advances in neural network learning over the past 30 years have been obtained by changing the model family rather than changing the optimization procedure
  - E.g., SGD was used in training since 1980s, while it still widely in use in modern SOTA network training

Examples

- Activation functions that increase and decrease in jagged non-monotonic patterns
  - Make optimization extremely difficult

- Design choice for the use linear transformations between layers and activation functions
  - Differentiable almost everywhere
  - Significant slope in large portions of their domain

- Linear paths or skip connections between layers
  - Reduce the length of the shortest path from the lower layer's parameters to the output
  - Mitigate the vanishing gradient problem
  - Similar idea - auxiliary heads
    - Adding extra copies of the output
    - Ensure that the lower layers receive a large gradient

Outline

- Deep Feedforward Networks
- Regularization
- Optimization
- Convolutional Neural Networks
- Practical Methods

Convolutional Neural Networks

*Convolutional networks are simply neural networks that use **convolution** in place of general matrix multiplication in at least one of their layers*

- Everything else stays the same
  - Maximum likelihood
  - Back-propagation
- CNNs are used to process data that has grid-like topology
  - 1D – regularly sampled time-series data
  - 2D – Images
  - 3D – volume data, video

Convolution Operation

- Convolution operation
  - For a signal input $x(t)$ at time t, we apply a weighting function $w(a)$ on every moment, the convolution output $s(t)$ would be

$$s(t) = (x * w)(t) = \int x(a)w(t-a)\,da$$

- Discrete convolution
  - for x and w are defined only on integer t

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Convolution Operation

- 2-D convolution with kernel K

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

- Since convolution is commutative

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

  - More straightforward to implement

- However, many machine learning libraries implement cross-correlation but call it convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

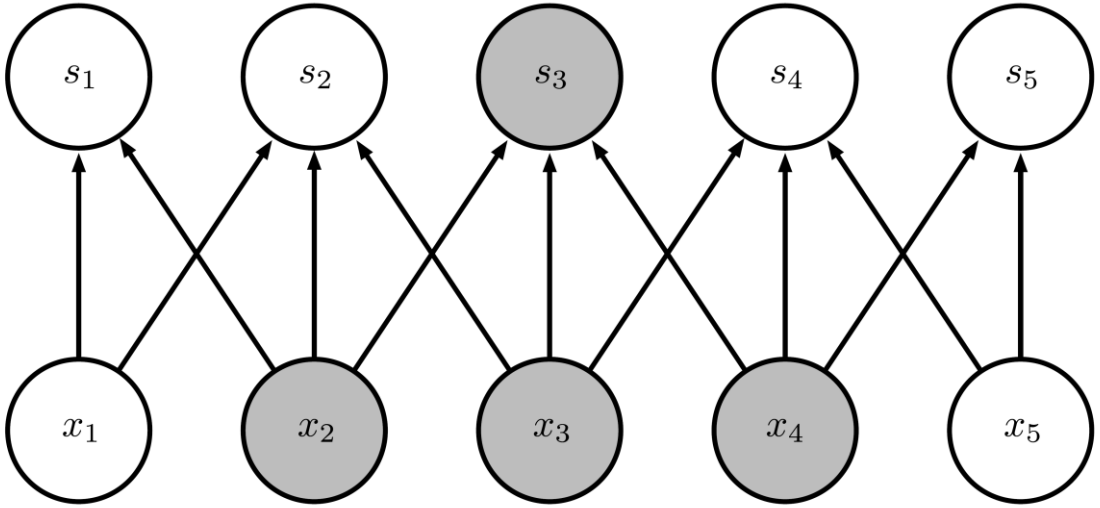
  - The same as convolution but without flipping the kernel
  - With learning algorithm, both can learn correctly

Motivation of CNN

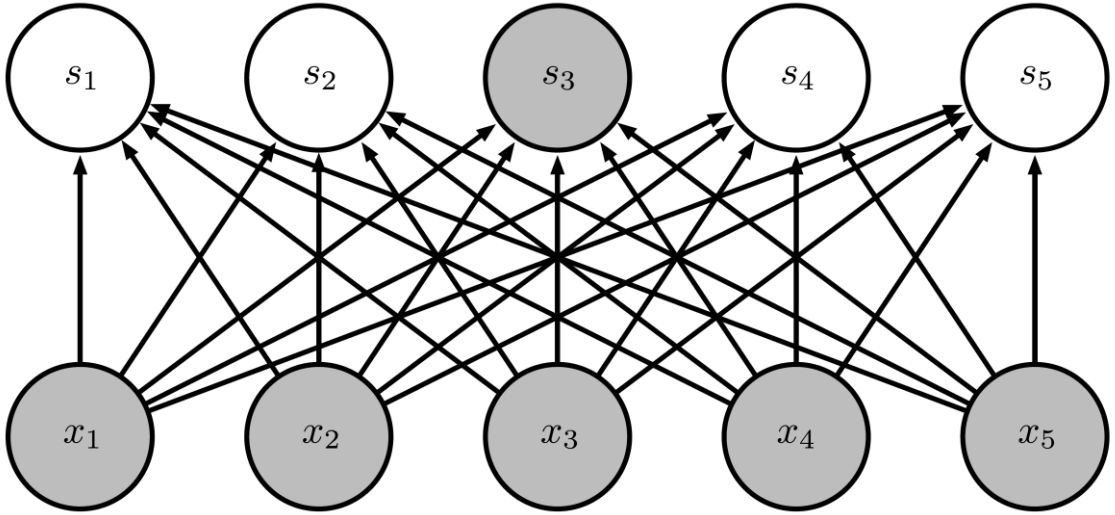
- Scale up neural networks to process very large signals / images / video sequences
- Three essential ideas
  - Sparse and local connectivity
    - Kernel is usually much smaller than input
    - Less memory and computation required
  - Parameter (weight) sharing
    - Use the same set of parameters for more than one function in a model
  - Equivalent representation
    - Automatically generalize across spatial translations of inputs

Local Connectivity

Sparse connections due to small convolution kernel

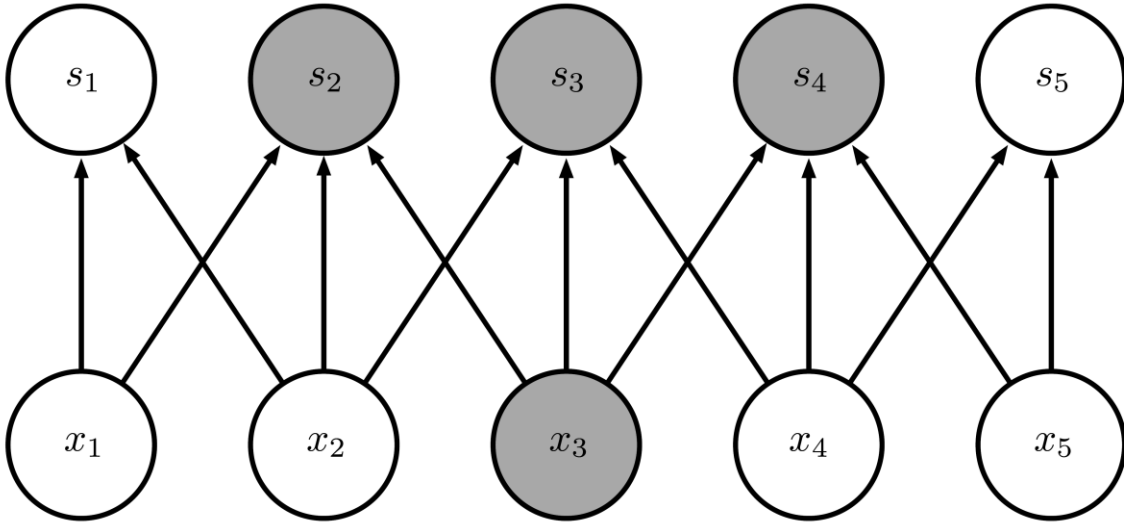


Dense connections

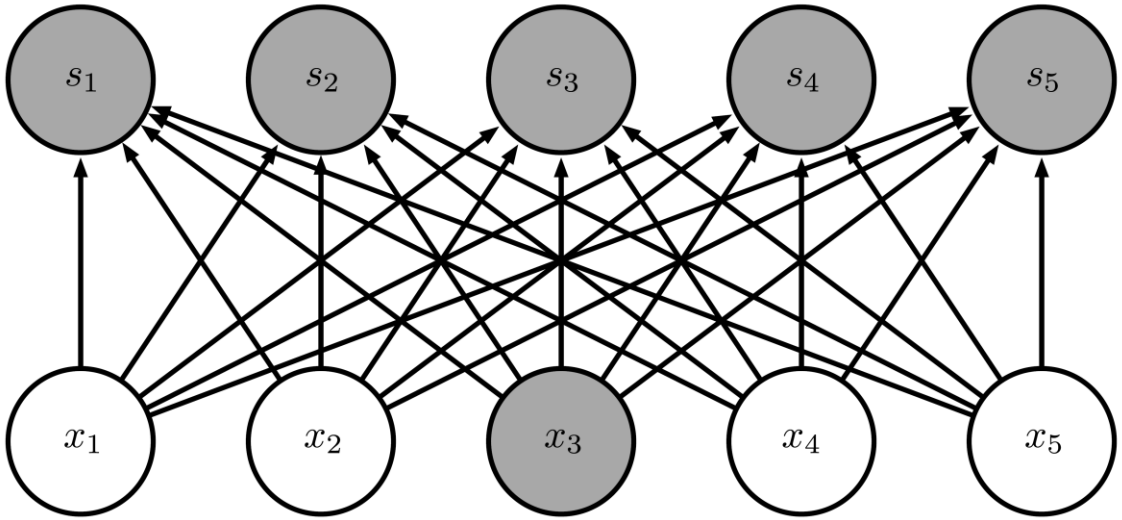


Local Connectivity – Back Propagation

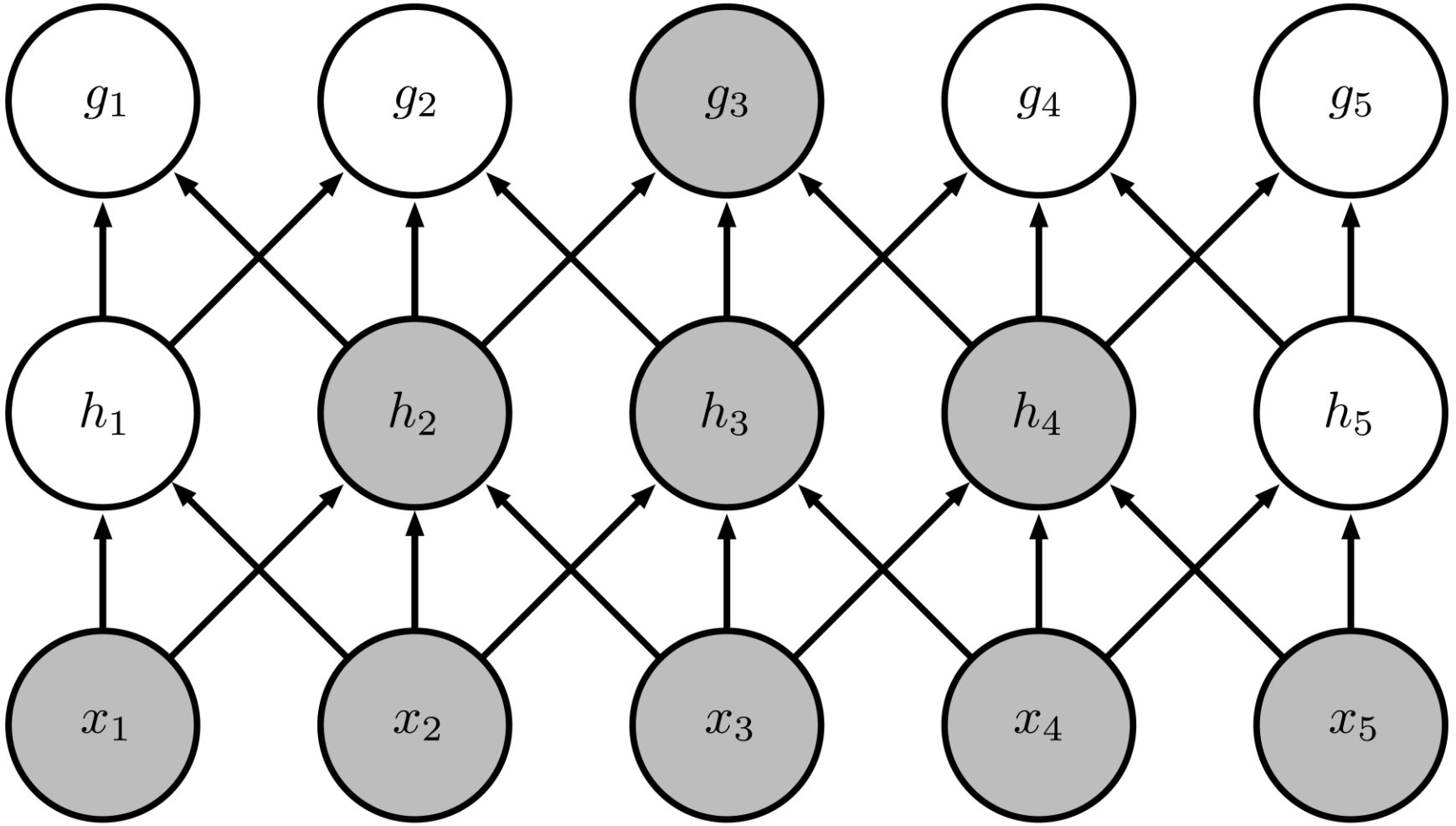
Sparse connections due to small convolution kernel



Dense connections

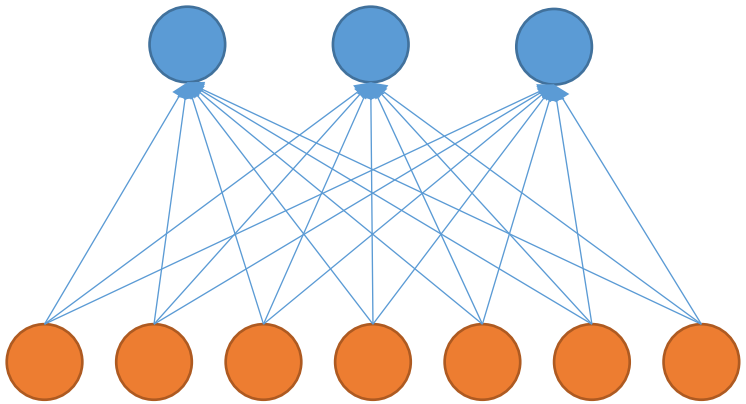


Growing Receptive Fields

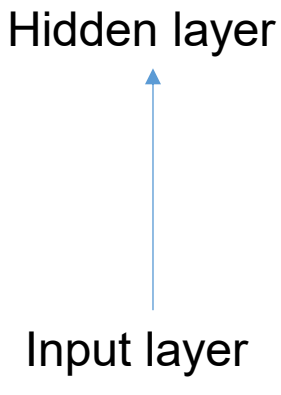


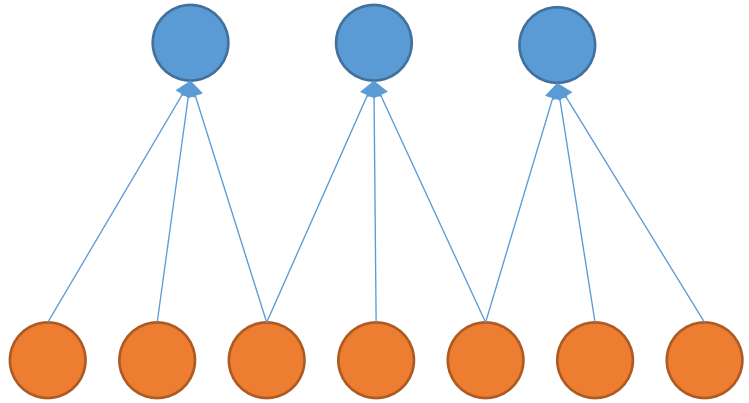
Local Connectivity

- # input units (neurons): 7
- # hidden units: 3
- Number of parameters
  - Global connectivity: 3 x 7 = 21
  - Local connectivity: 3 x 3 = 9



Global connectivity

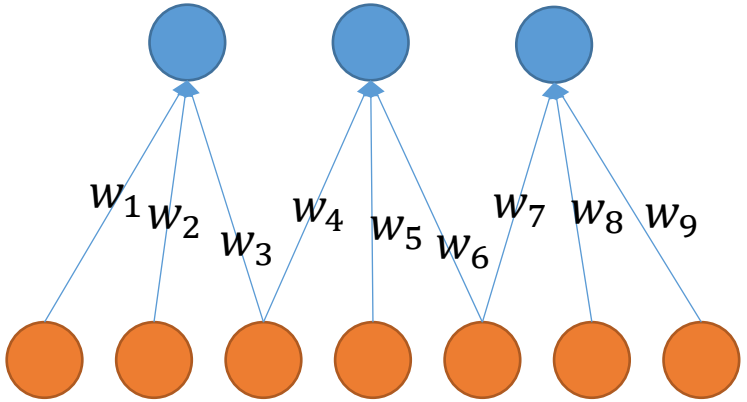




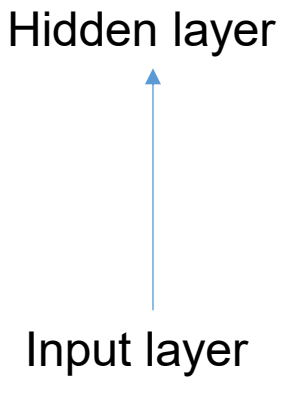
Local connectivity

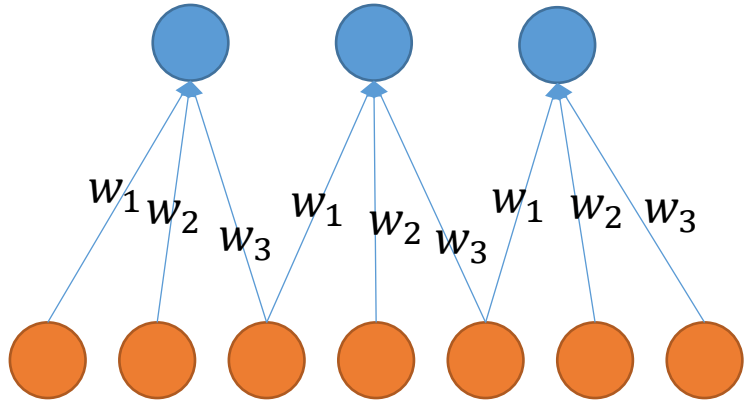
Parameter Sharing

- # input units (neurons): 7
- # hidden units: 3
- Number of parameters
  - Without weight connectivity: 3 x 3 = 9
  - Local connectivity: 3 x 1 = 3



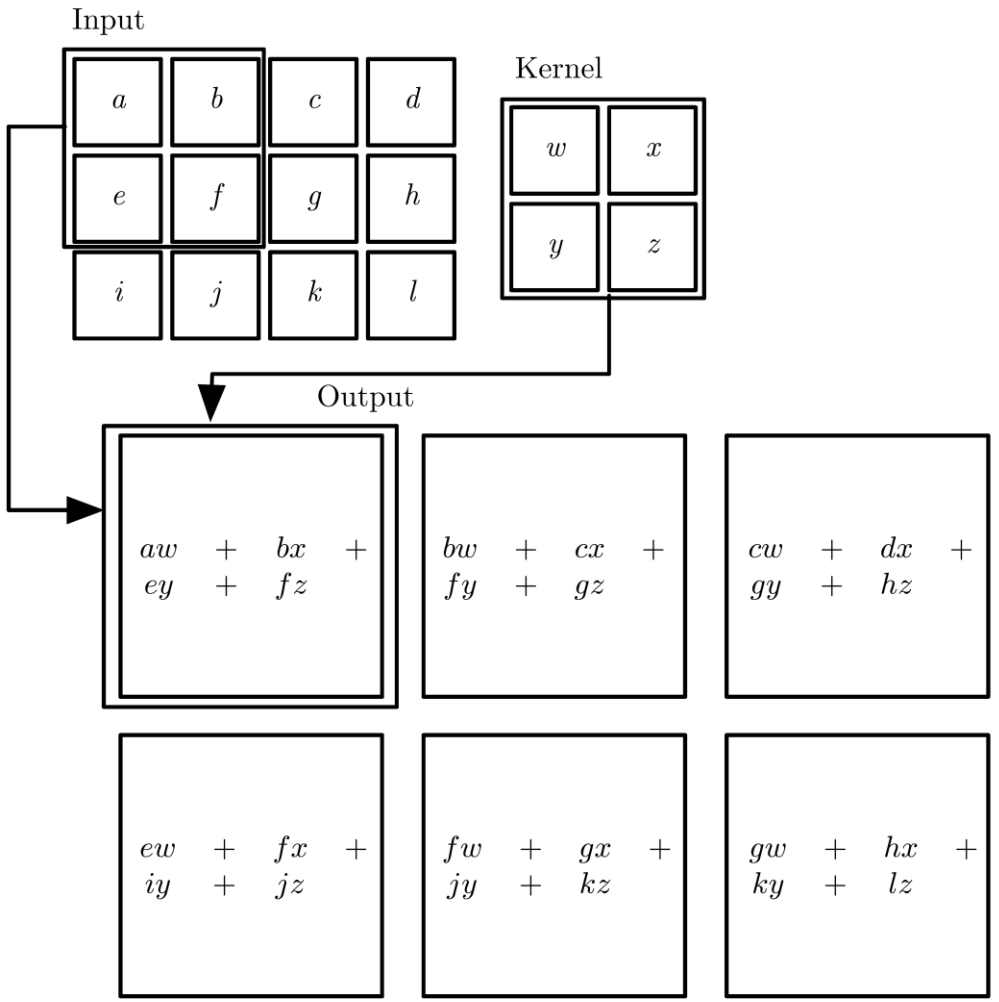
Local connectivity
Without parameter sharing



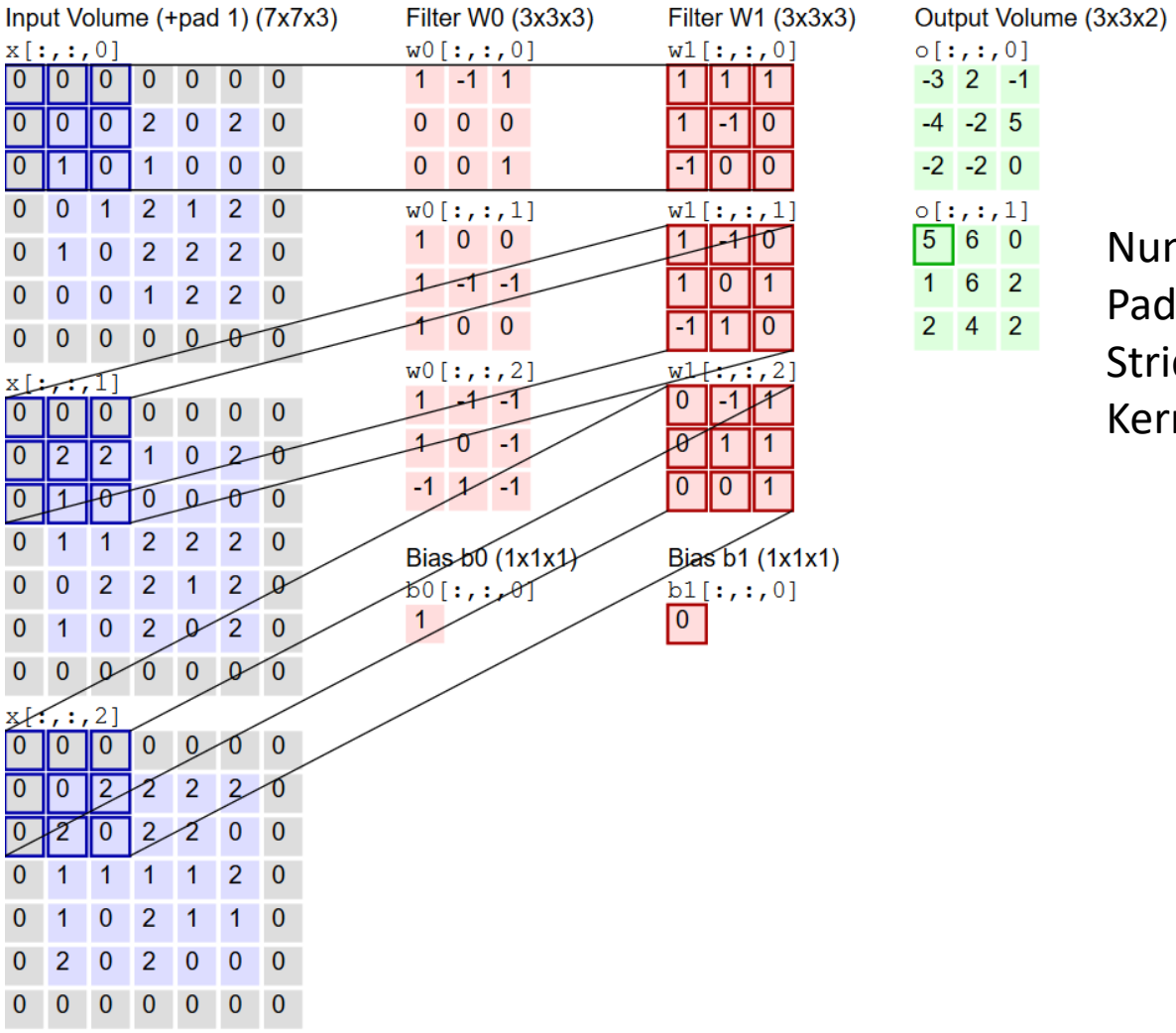


Local connectivity
With Parameter sharing

2D Convolution

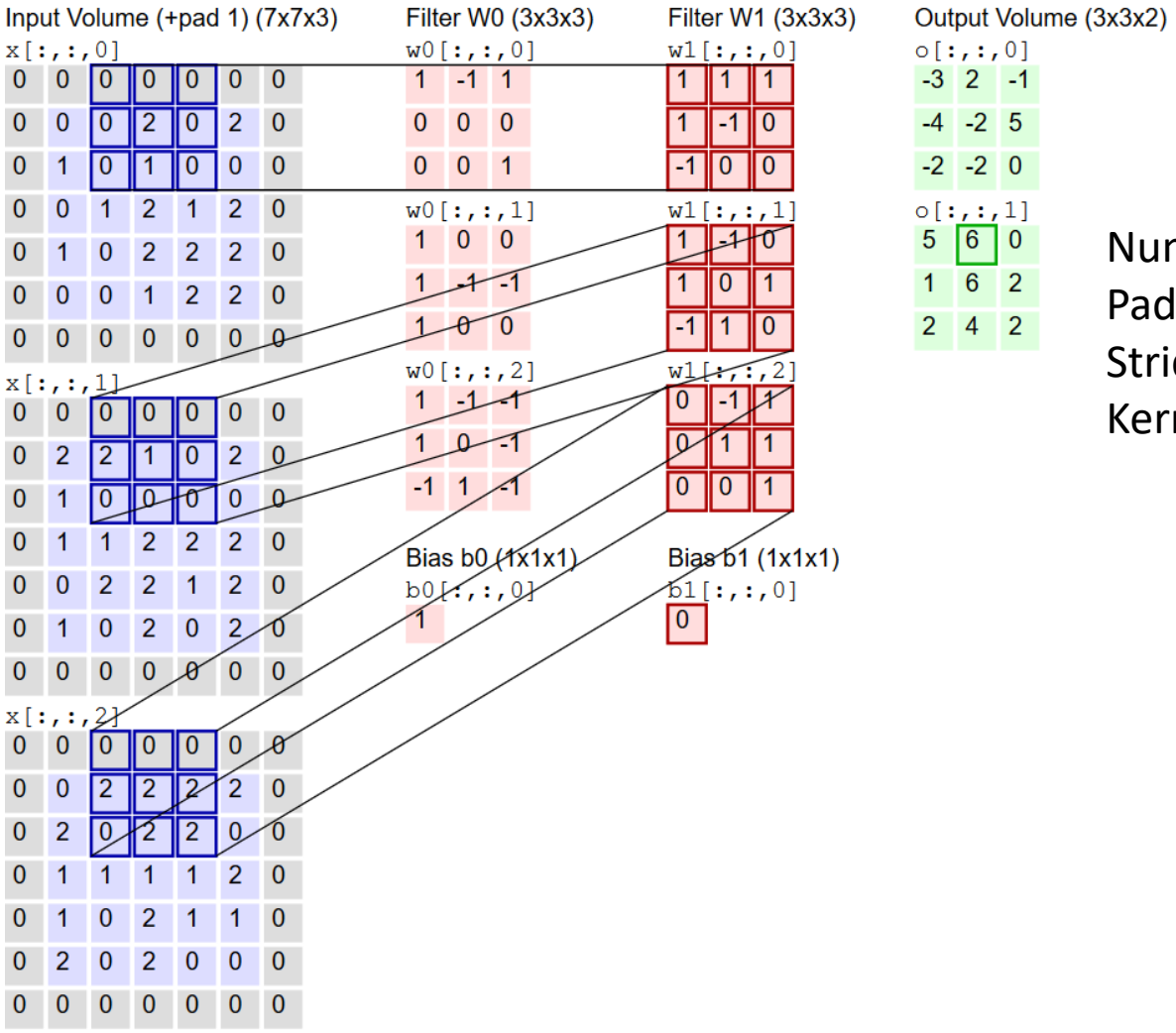


2D Convolution Operation Example



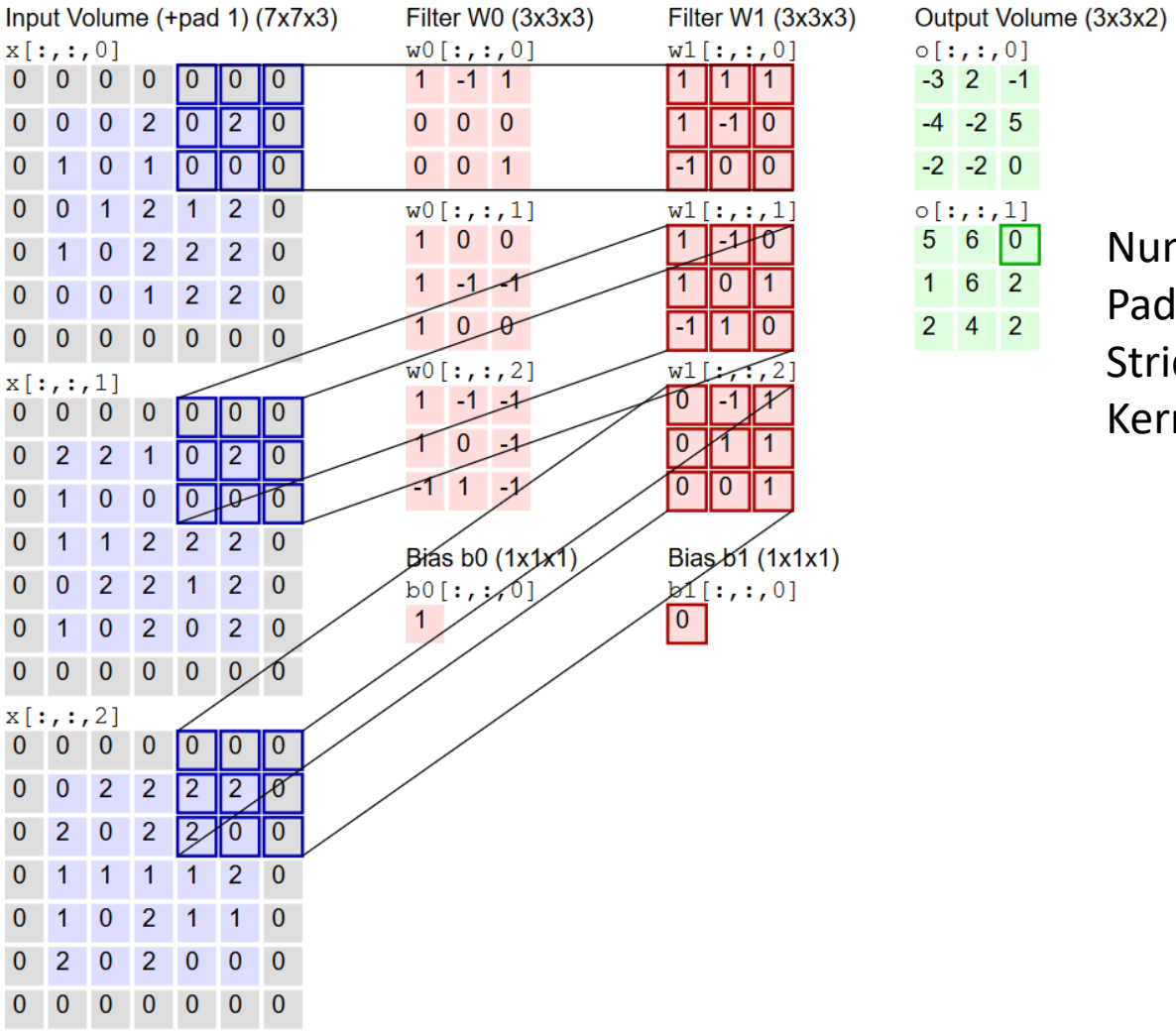
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



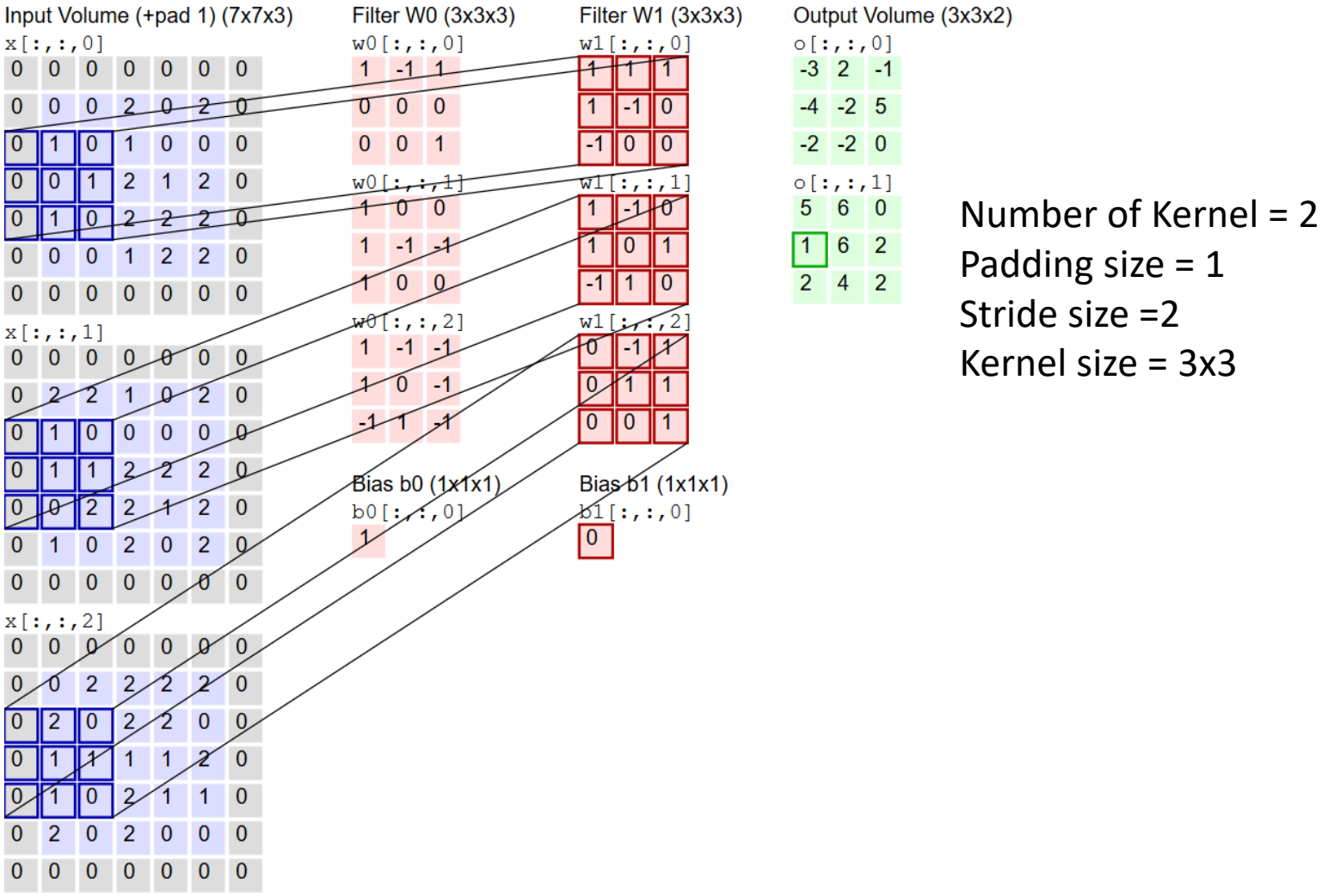
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



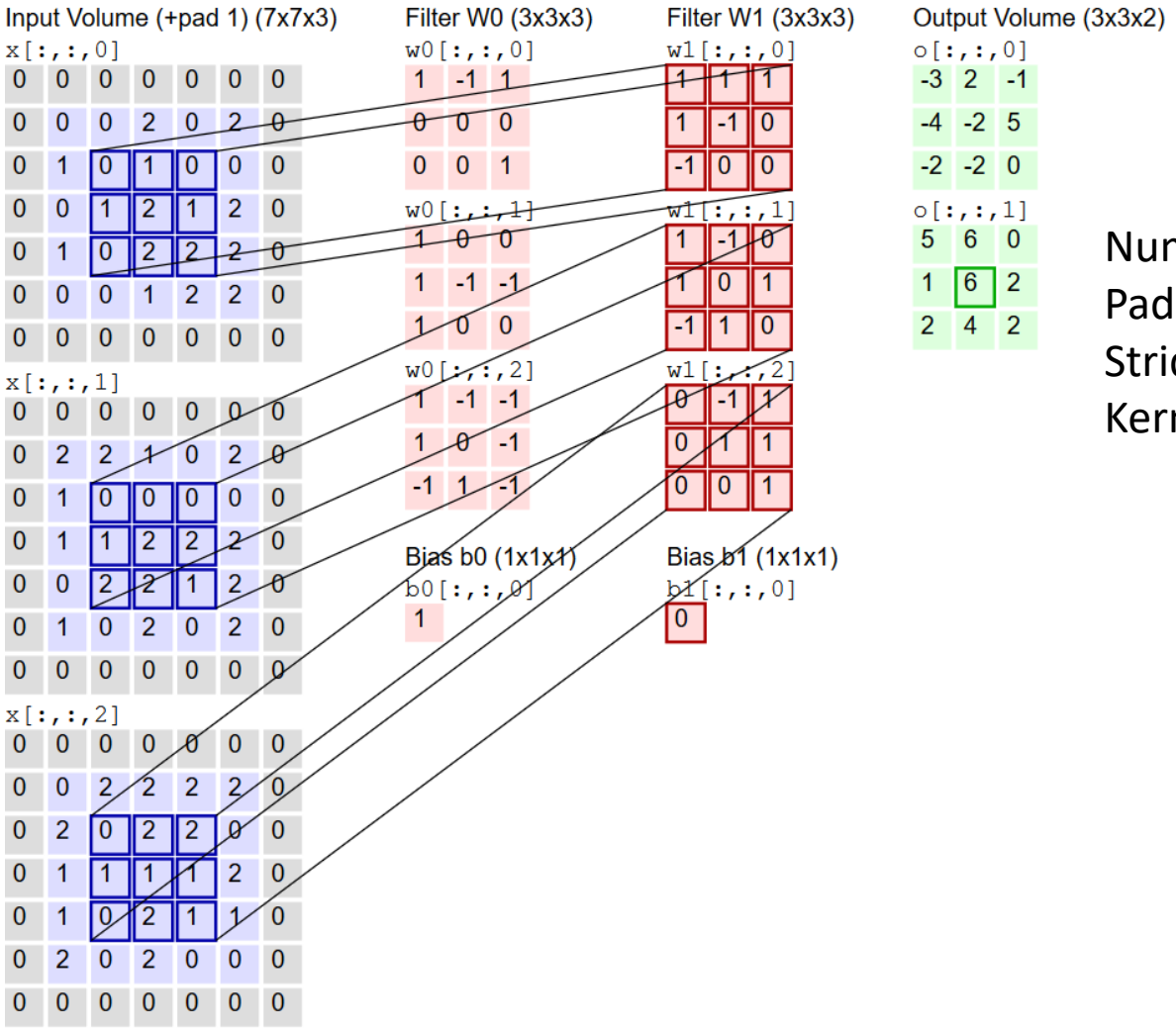
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



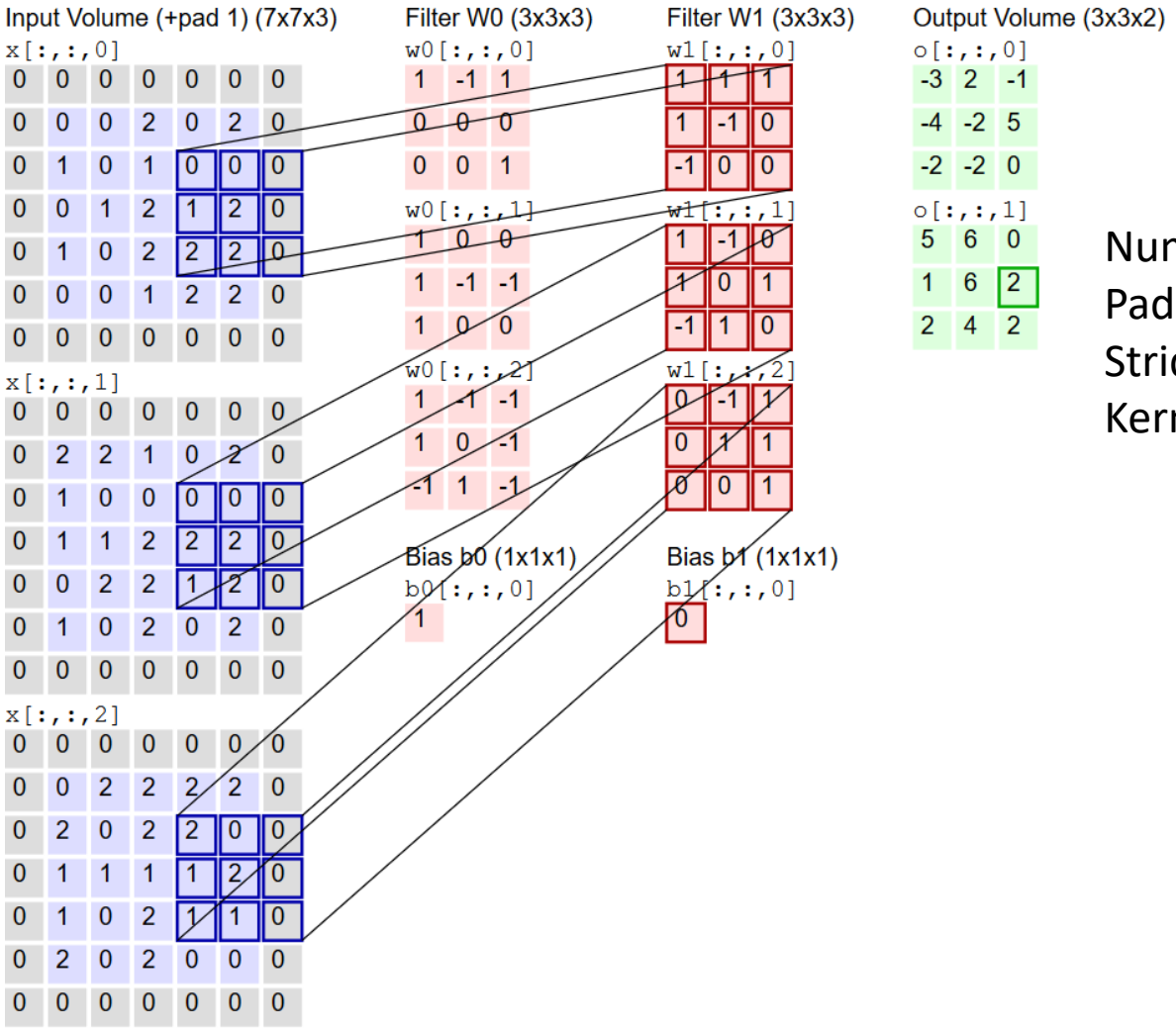
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



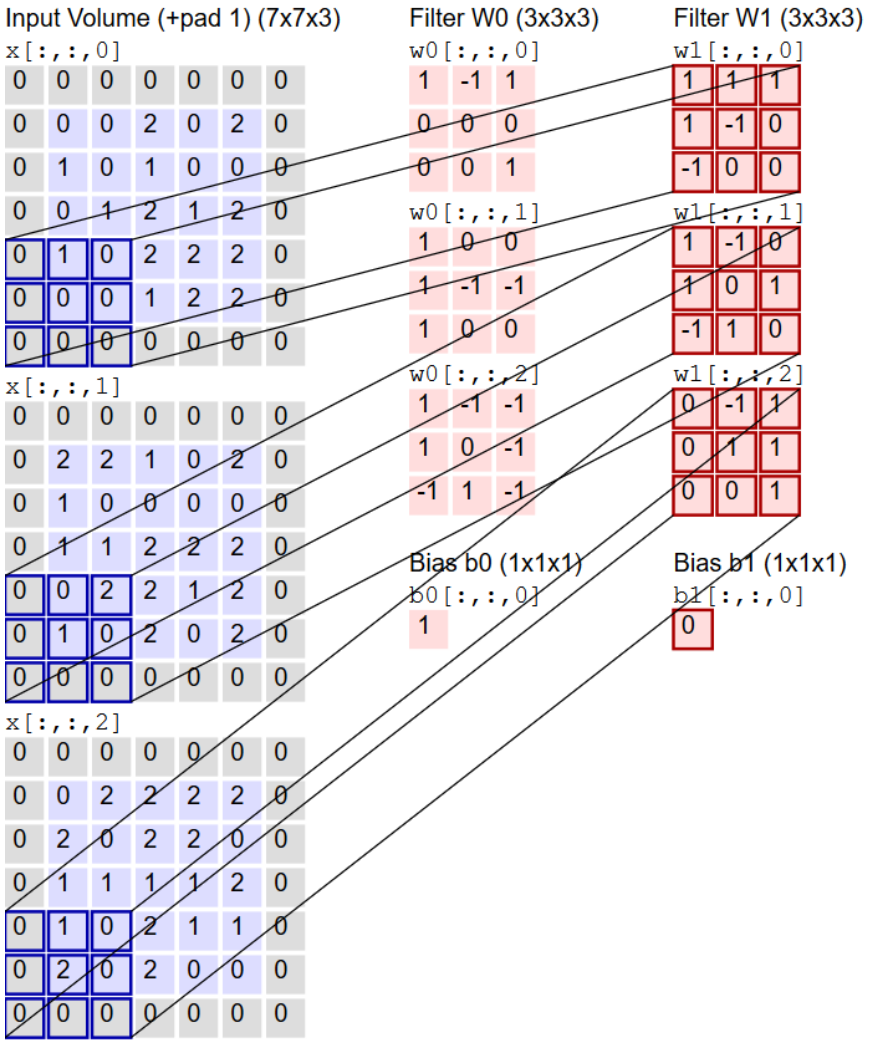
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



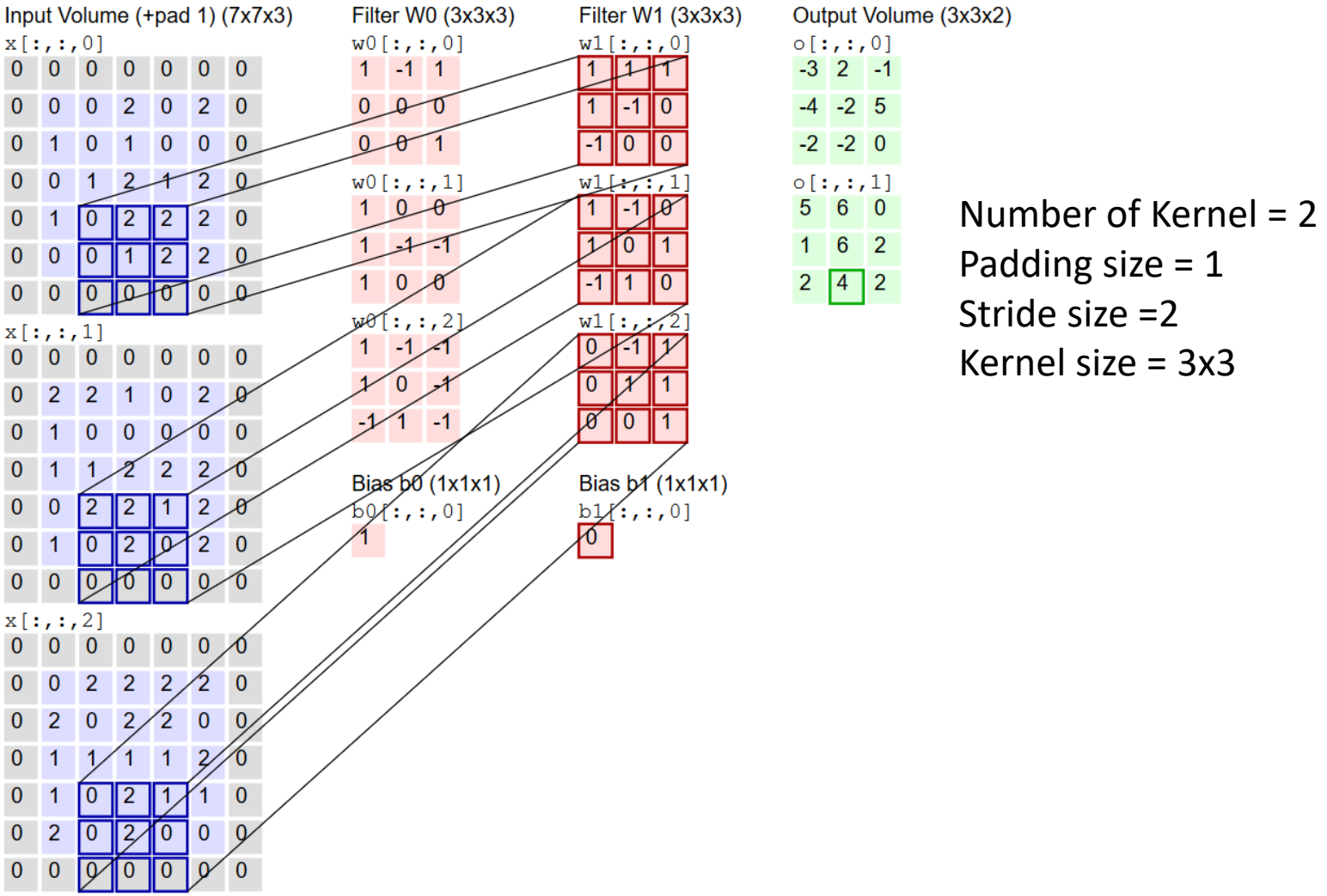
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



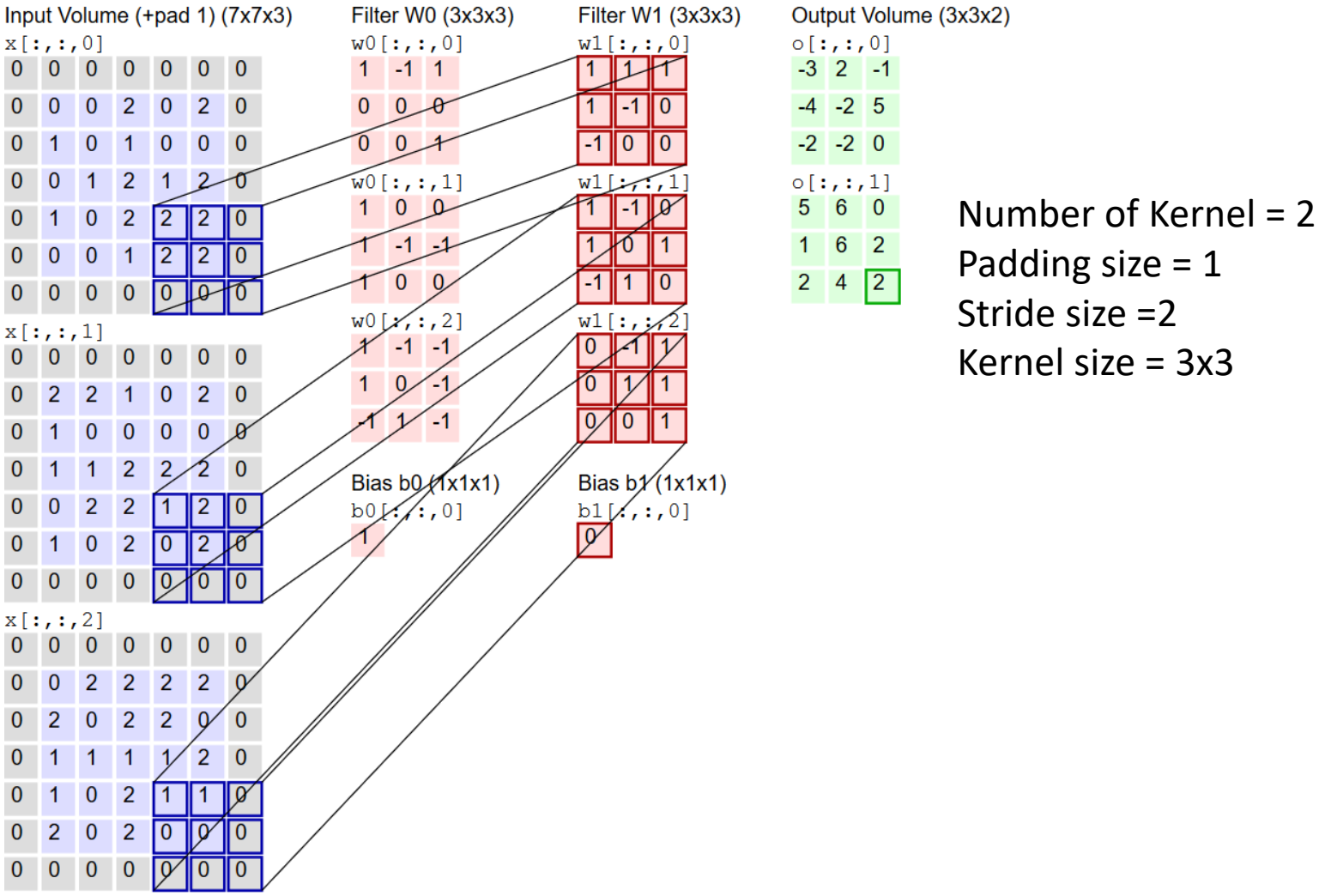
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

2D Convolution Operation Example



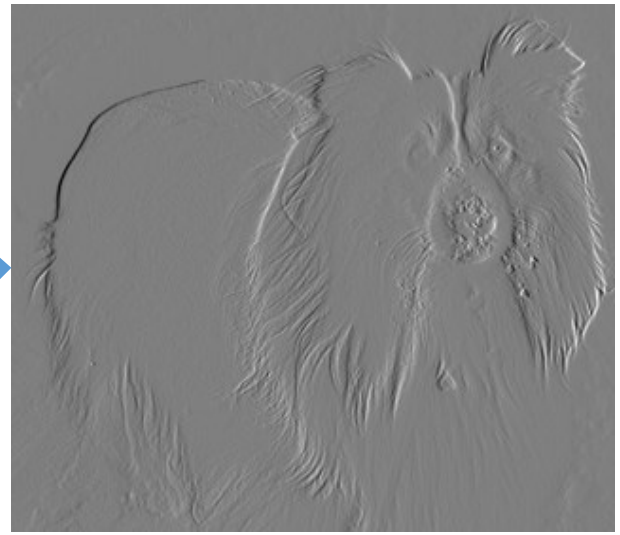
Number of Kernel = 2
Padding size = 1
Stride size =2
Kernel size = 3x3

Edge Detection by Convolution

Input

1	-1

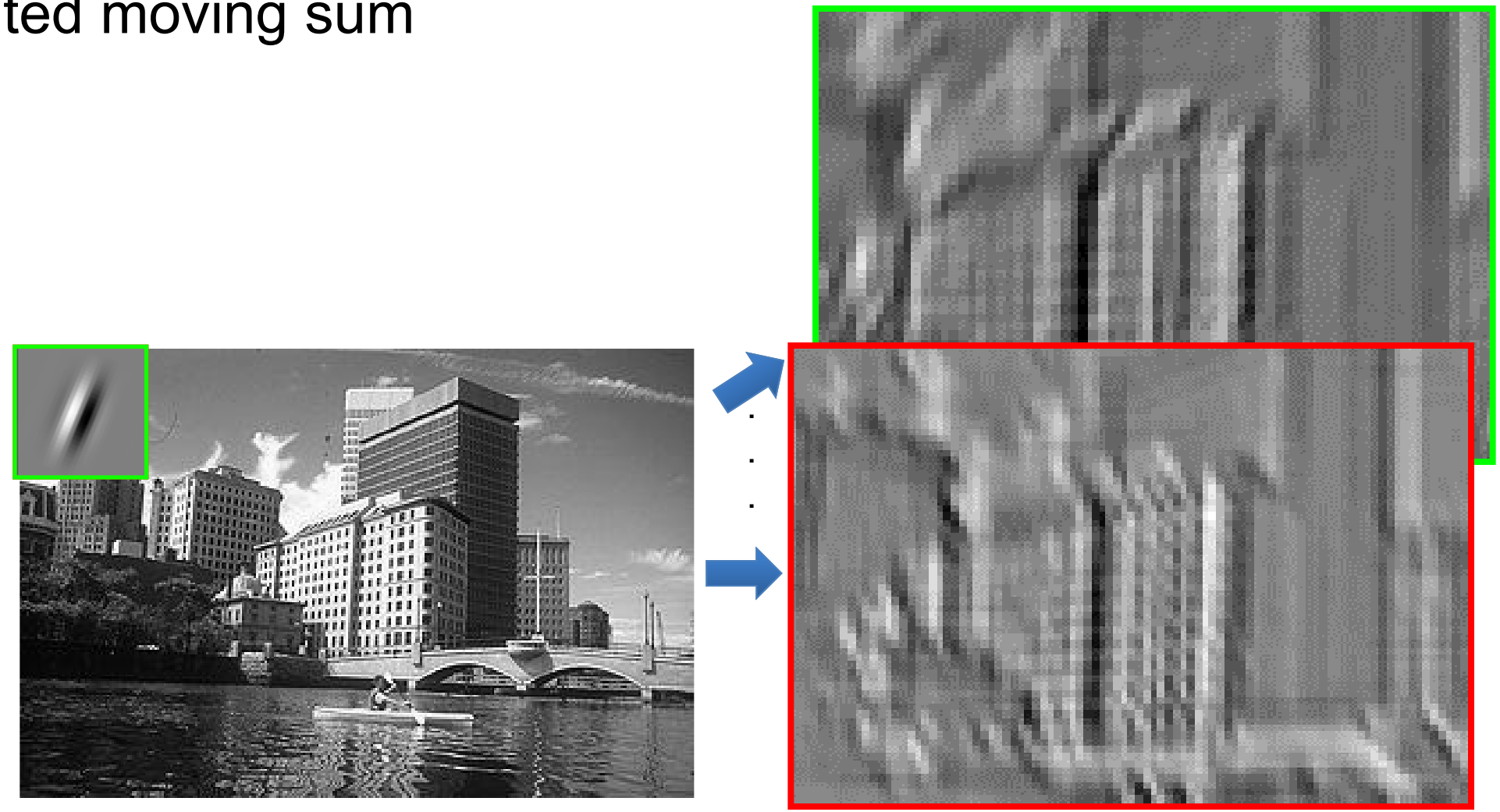
Kernel



Output

Meanings of Convolution

- Weighted moving sum

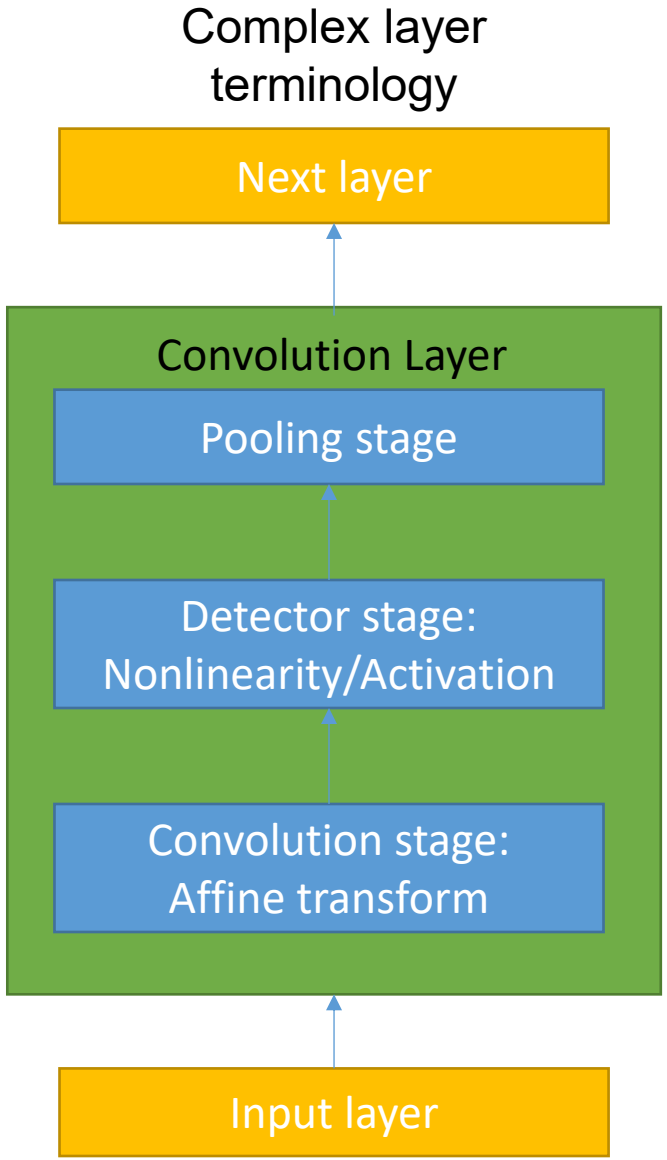


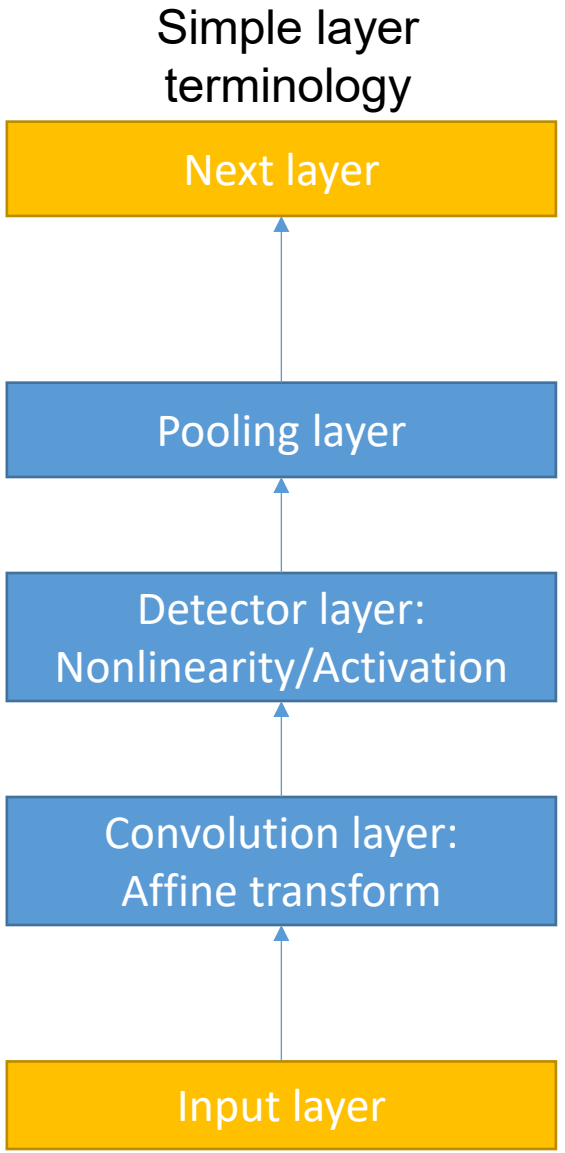
Input

Feature Activation Map

slide credit: S. Lazebnik

CONV Layer Terminology

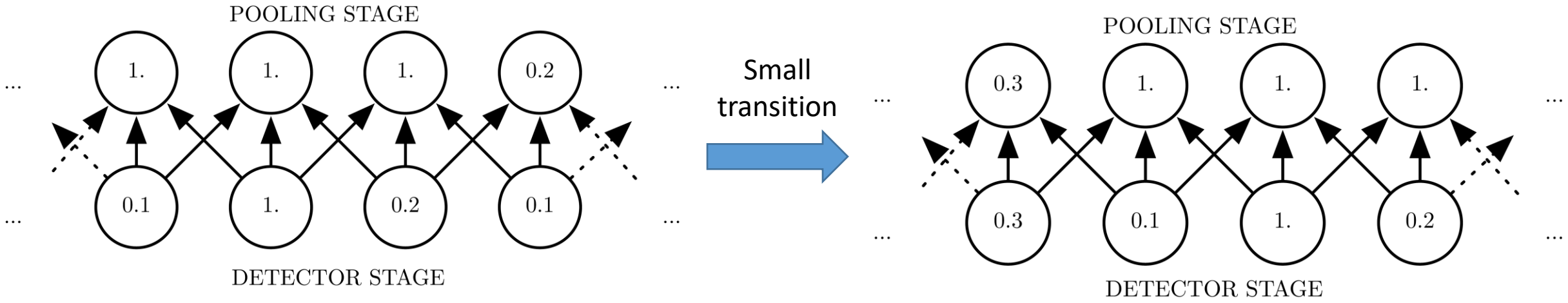




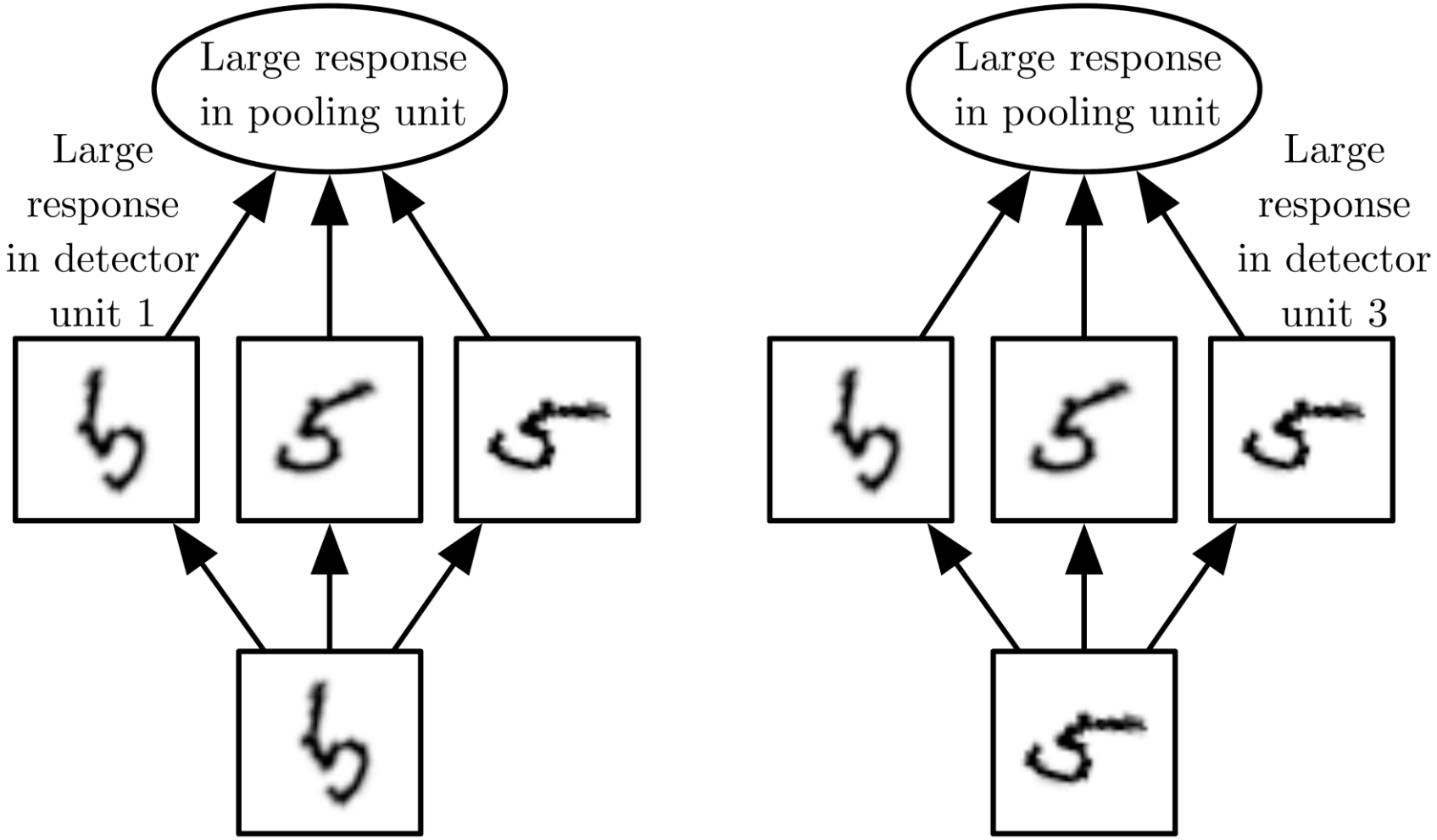
Pooling

- A pooling function replaces the output of the net with a summary statistic of the nearby outputs
- Examples of pooing functions
  - Maximum within a rectangular neighborhood
  - Average of a rectangular neighborhood
  - L2 norm of a rectangular neighborhood
  - Weighted average based on the distance from the central pixel
- Can use fewer pooling units than detector units
- Can help to handle inputs of varying size
- Pooling helps to make the representation become approximately **invariant** to small translation and rotation

Max Pooling and Invariance to Translation

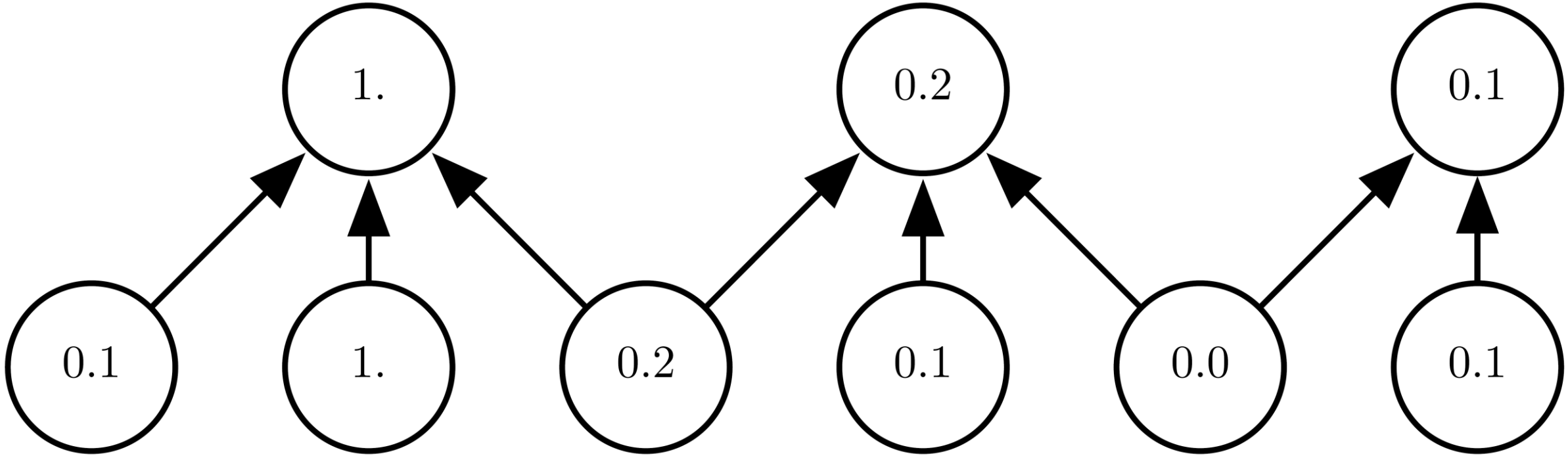


Cross-Channel Pooling and Invariance to Learned Transformations



Pooling with Downsampling

- Max-pooling
  - Pooling width = 3
  - Stride between pools: 2

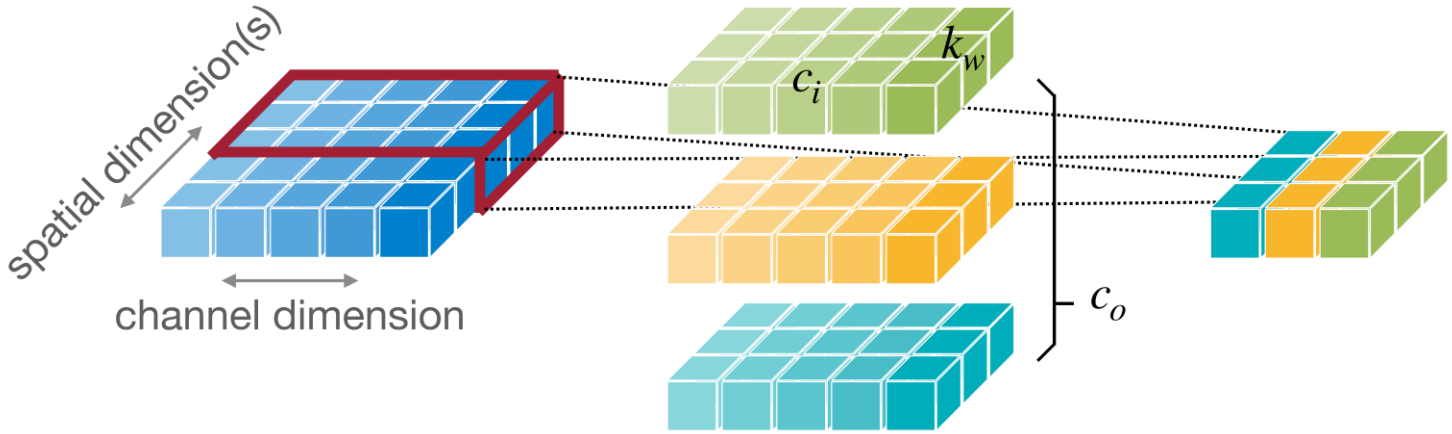


Convolution Layer

- The output neuron is connected to input neurons in the receptive field

Shape of tensors	1D Convolution
Input feature X	(n, c_i, w_i)
Output feature Y	(n, c_o, w_o)
Weight W	(c_o, c_i, k_w)
Bias b	(c_o)

Notation	
n	Batch size
c_i, c_o	Input/output channel
w_i, w_o	Input/output width
h_i, h_o	Input/output height
k_w, k_h	Kernel width/height

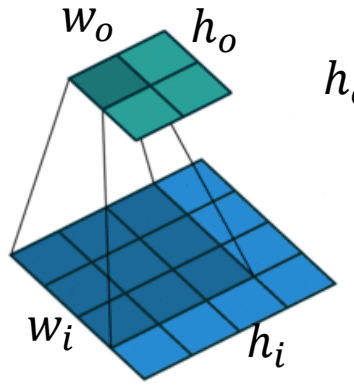


Convolution Layer

- The output neuron is connected to input neurons in the receptive field

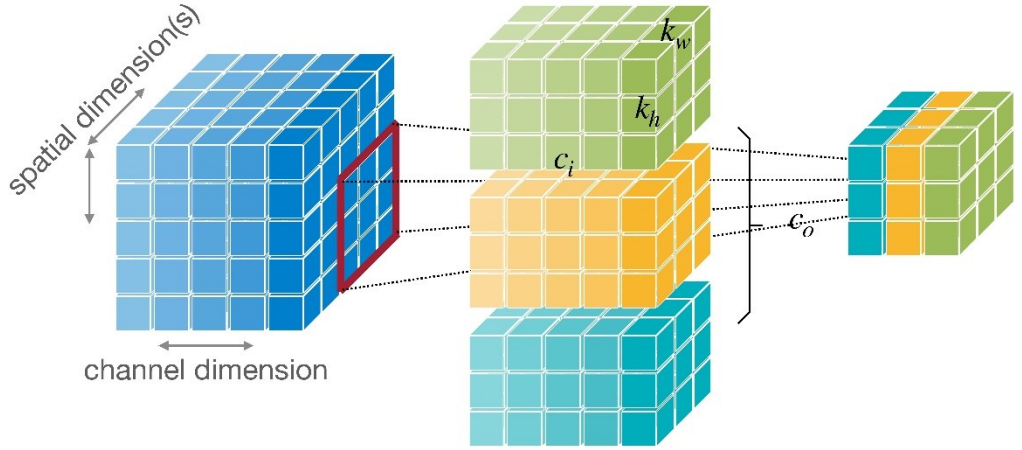
Shape of tensors	1D Convolution	2D Convolution
Input feature X	(n, c_i, w_i)	(n, c_i, h_i, w_i)
Output feature Y	(n, c_o, w_o)	(n, c_o, h_o, w_o)
Weight W	(c_o, c_i, k_w)	(n, c_o, k_h, k_w)
Bias b	(c_o)	(c_o)

Notation	
n	Batch size
c_i, c_o	Input/output channel
w_i, w_o	Input/output width
h_i, h_o	Input/output height
k_w, k_h	Kernel width/height



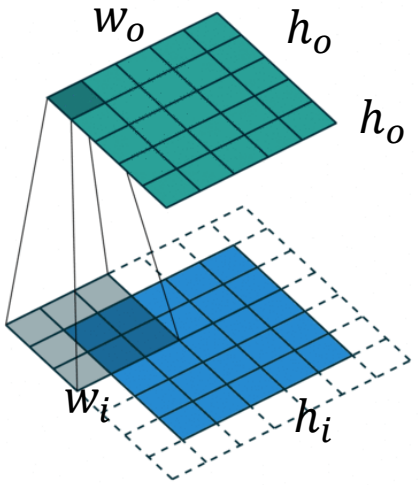
$$h_o = h_i - k_h + 1$$

Feature map size becomes smaller



Convolution Layer - Padding

- Padding can be used to control the size of output feature map
  - Zero Padding
    - Pads the input boundaries with zero
  - Other Paddings: Reflection Padding, Replication Padding, Constant Padding



$$h_o = h_i - k_h + 1 + 2p$$

Same input/output feature map size

Zero Padding

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	2	3	0	0
0	0	4	5	6	0	0
0	0	7	8	9	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Reflection Padding

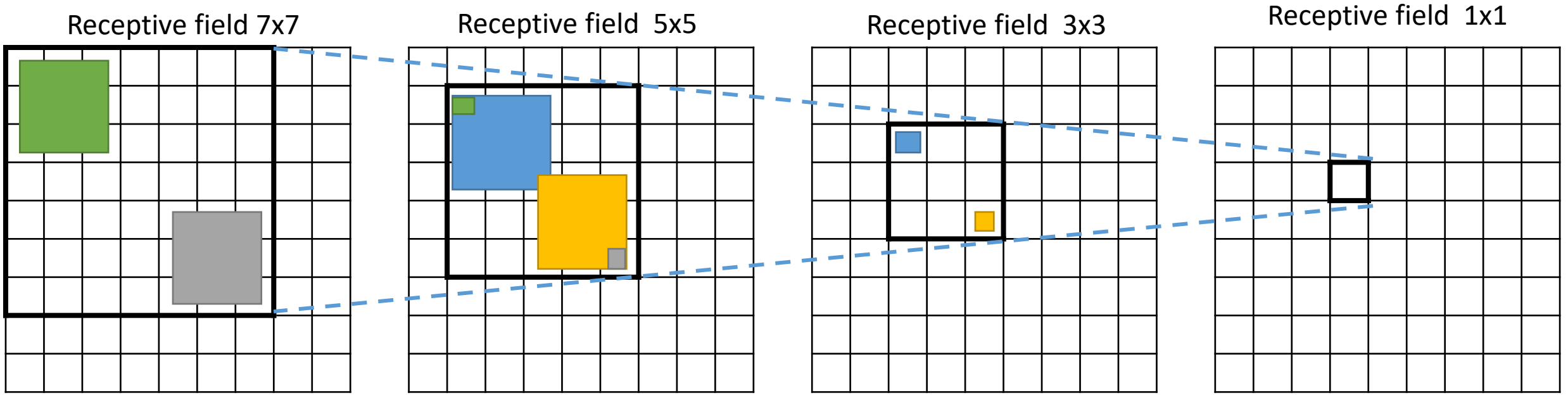
9	8	7	8	9	8	7
6	5	4	5	6	4	5
3	2	1	2	3	2	1
6	5	4	5	6	5	4
9	8	7	8	9	8	7
6	5	4	5	6	5	4
3	2	1	2	3	2	1

Replication Padding

1	1	1	2	3	3	3
1	1	1	2	3	3	3
1	1	1	2	3	3	3
4	4	4	5	6	6	6
7	7	7	8	9	9	9
7	7	7	8	9	9	9
7	7	7	8	9	9	9

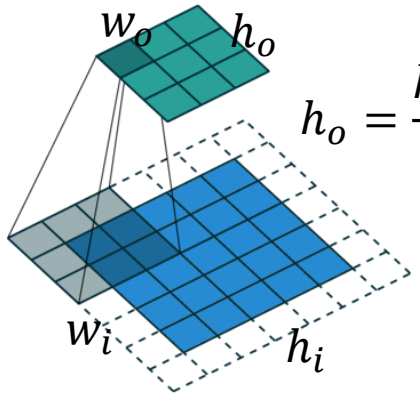
Convolution Layer: Receptive Field

- In convolution, each output element depends on $k_h \times k_w$ receptive field in the input
- Each successive convolution adds $k - 1$ to the receptive field size
- With L layers, the receptive field size is $L \times (k - 1) + 1$

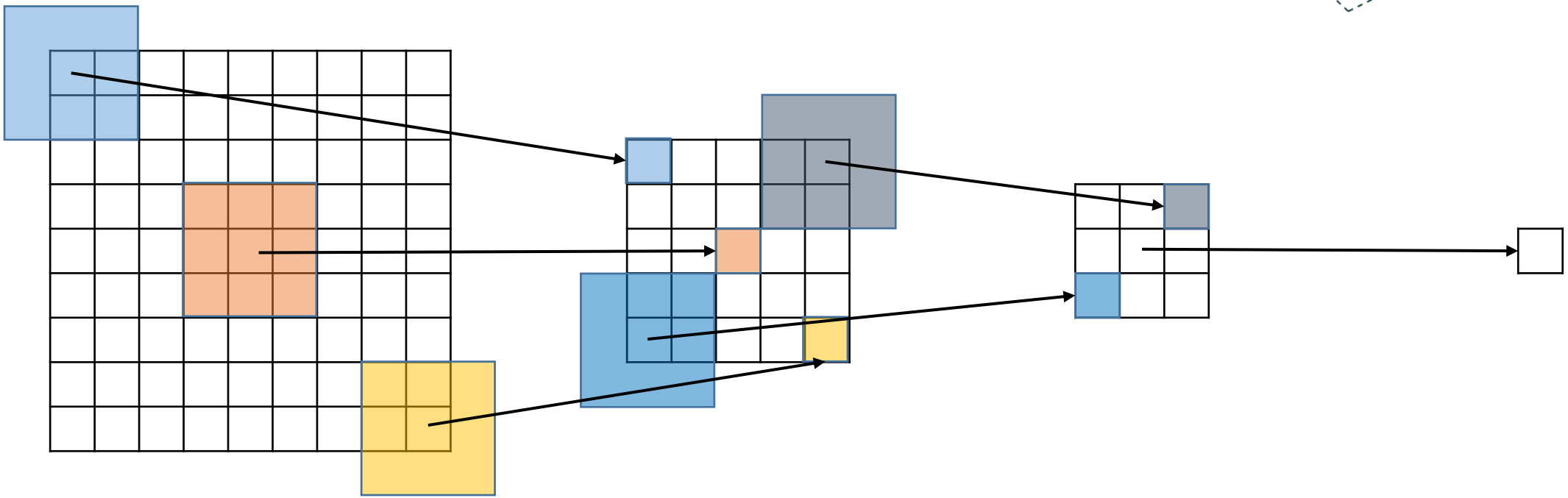


$$L = 4, k = 3$$

Strided Convolution Layer



$$h_o = \frac{h_i - k_h + 2p}{s} + 1$$

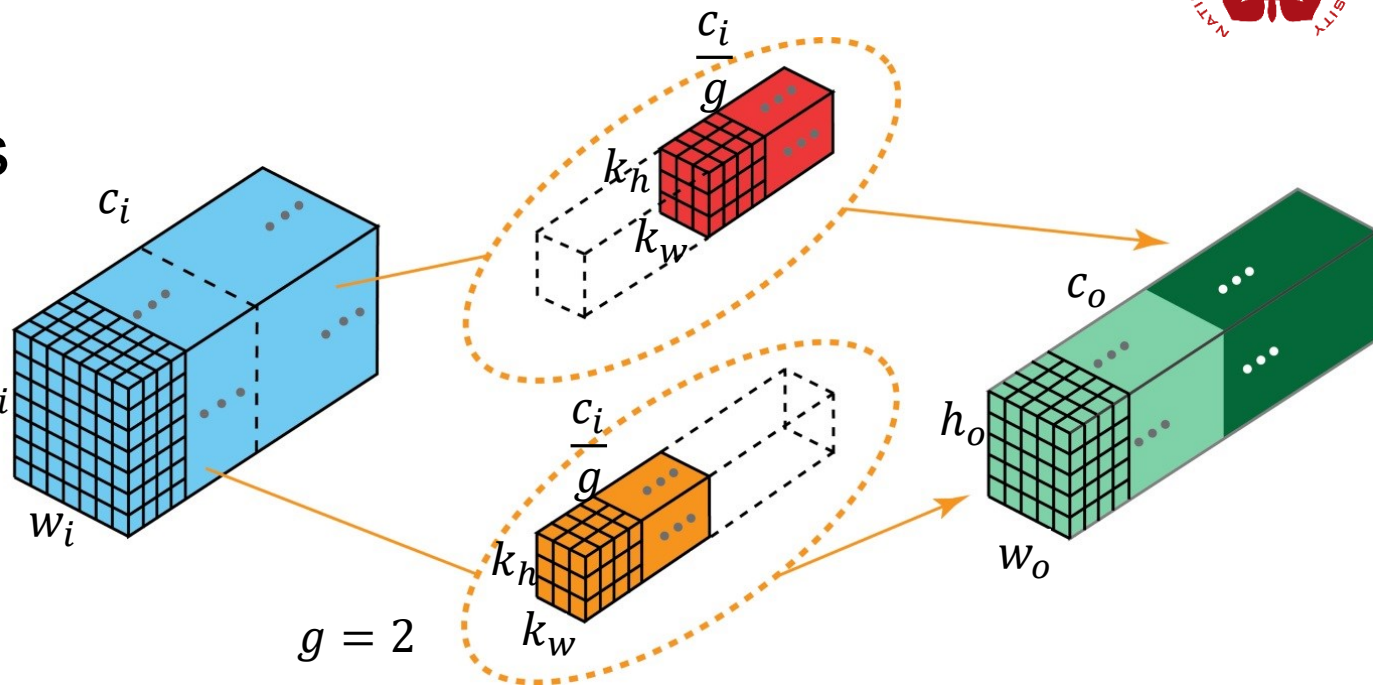


$L = 4, k = 3,$ Stride $s = 2,$

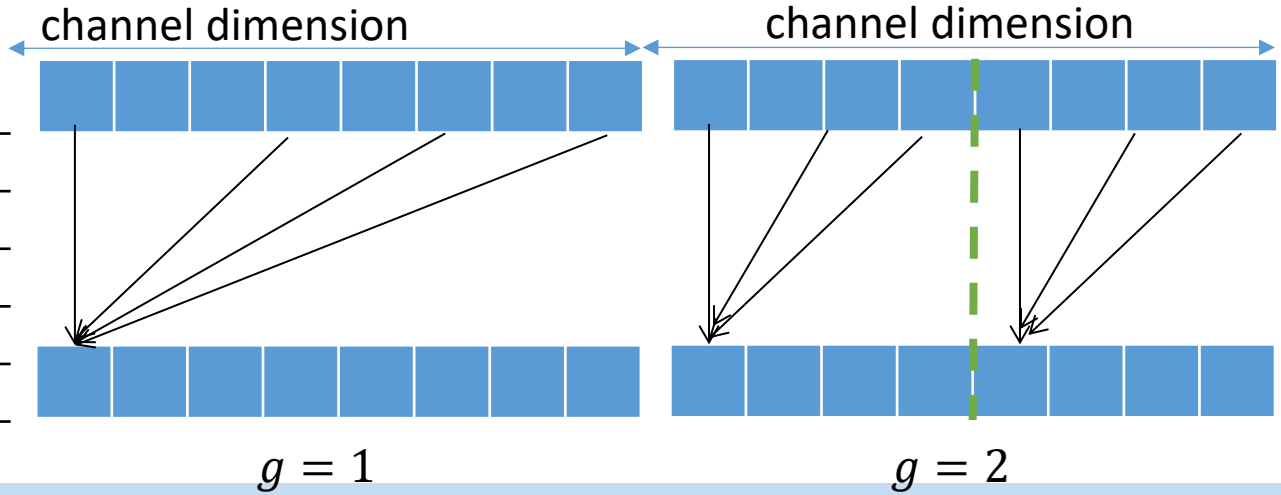
Grouped Convolution Layer

- A group of narrower convolutions

Shape of tensors	2D CONV	Grouped CONV
Input feature X	(n, c_i, h_i, w_i)	(n, c_i, h_i, w_i)
Output feature Y	(n, c_o, h_o, w_o)	(n, c_o, h_o, w_o)
Weight W	(n, c_o, h_h, w_w)	$(g \times \frac{c_o}{g}, \frac{c_i}{g}, k_h, k_w)$
Bias b	(c_o)	(c_o)



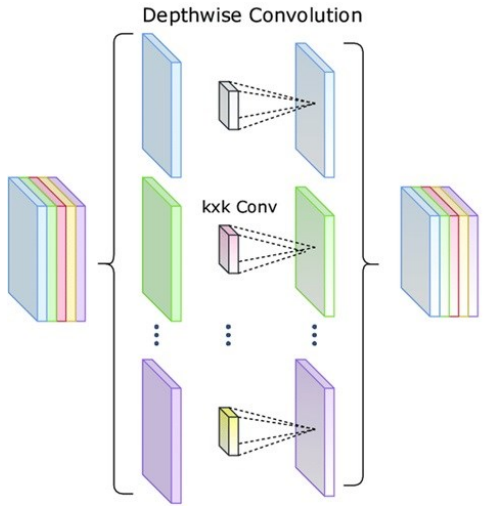
Notation	
n	Batch size
c_i, c_o	Input/output channel
w_i, w_o	Input/output width
h_i, h_o	Input/output height
k_w, k_h	Kernel width/height
g	Groups

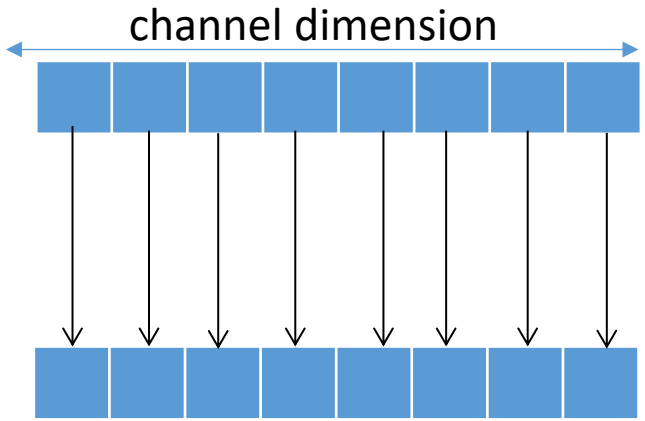


Depthwise Convolution Layer

- Independent filter for each channel
  - A special case of grouped convolution: $g = c_i = c_o$

Shape of tensors	Grouped CONV	Depthwise CONV
Input feature X	(n, c_i, h_i, w_i)	(n, c_i, h_i, w_i)
Output feature Y	(n, c_o, h_o, w_o)	(n, c_o, h_o, w_o)
Weight W	$(g \times \frac{c_o}{g}, \frac{c_i}{g}, k_h, k_w)$	(c, k_h, k_w)
Bias b	(c_o)	(c_o)



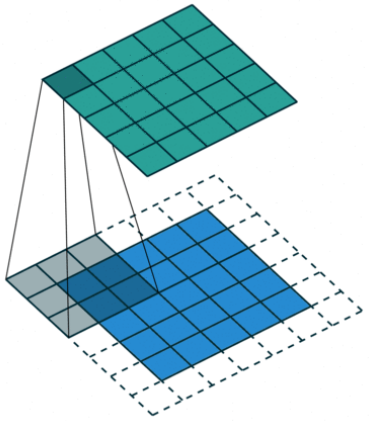


$g = c_i = c_o$

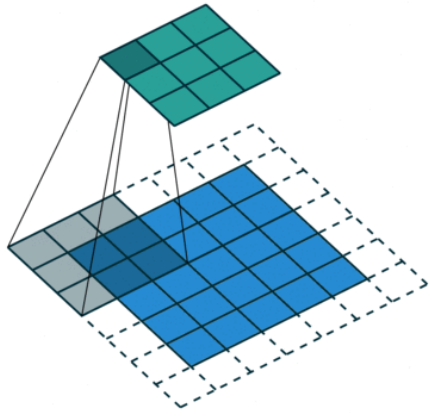
Notation	
n	Batch size
c_i, c_o	Input/output channel
w_i, w_o	Input/output width
h_i, h_o	Input/output height
k_w, k_h	Kernel width/height
g	Groups

Other Convolution Layers

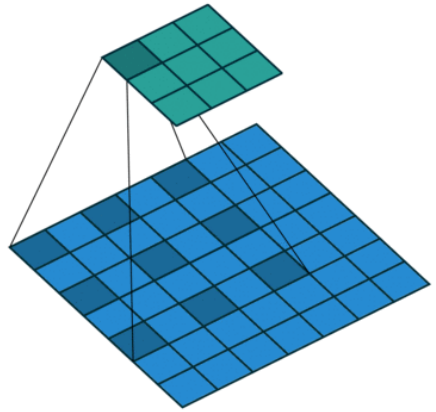
Convolution



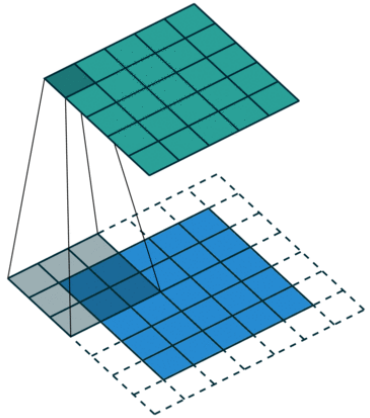
Convolution
Stride = 2



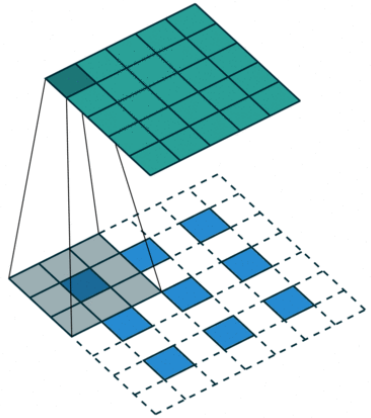
Dilated Convolution



Transposed Convolution

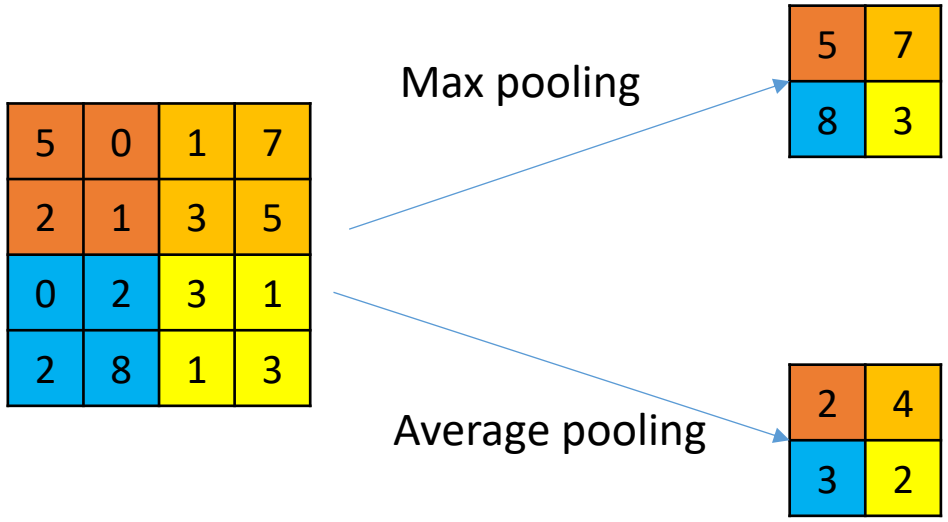


Transposed Convolution
Stride = 2



2D Pooling Layer

- Downsample the feature map to a smaller size
  - The output neuron pools the features in the receptive field, similar to convolution
  - Usually, the stride is the same as the kernel size: $s = k$
- Pooling operates over each channel independently
  - No learnable parameters



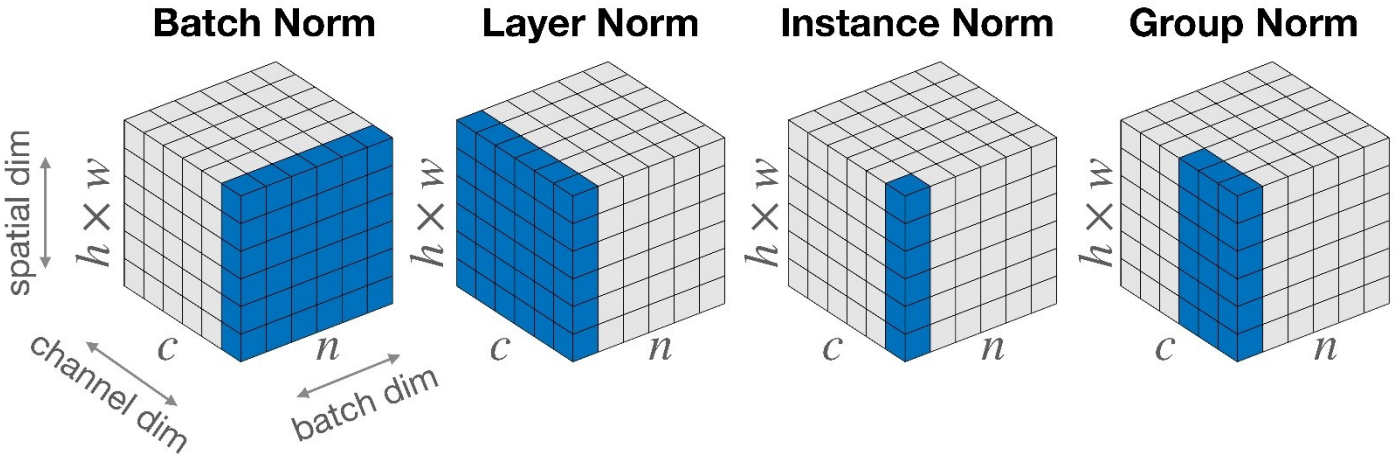
Normalization Layer

- Normalizing the features makes optimization faster
  - Normalization layer normalizes the features

$$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i), \qquad \mu_i := \frac{1}{m}\sum_{k \in \mathcal{S}_i} x_k \,, \sigma_i = \sqrt{\frac{1}{m}\sum_{k \in \mathcal{S}_i}(x_k - \mu_i)^2 + \epsilon}$$

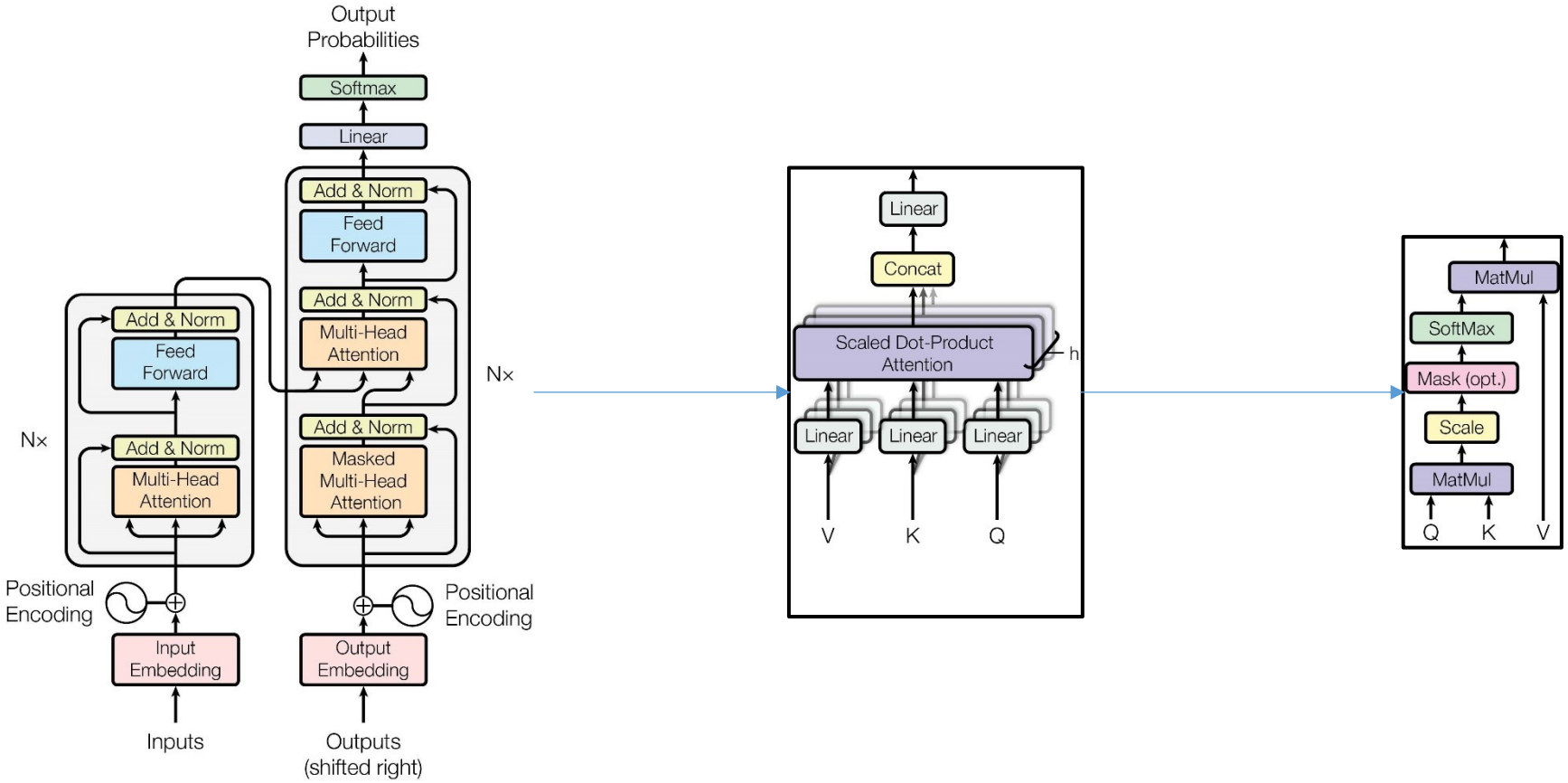
  - Then learns a per-channel linear transform (trainable scale and shift) to compensate for the possible lost of representational ability

$$y = \gamma_{i_c}\hat{x}_i + \beta_{i_c}$$

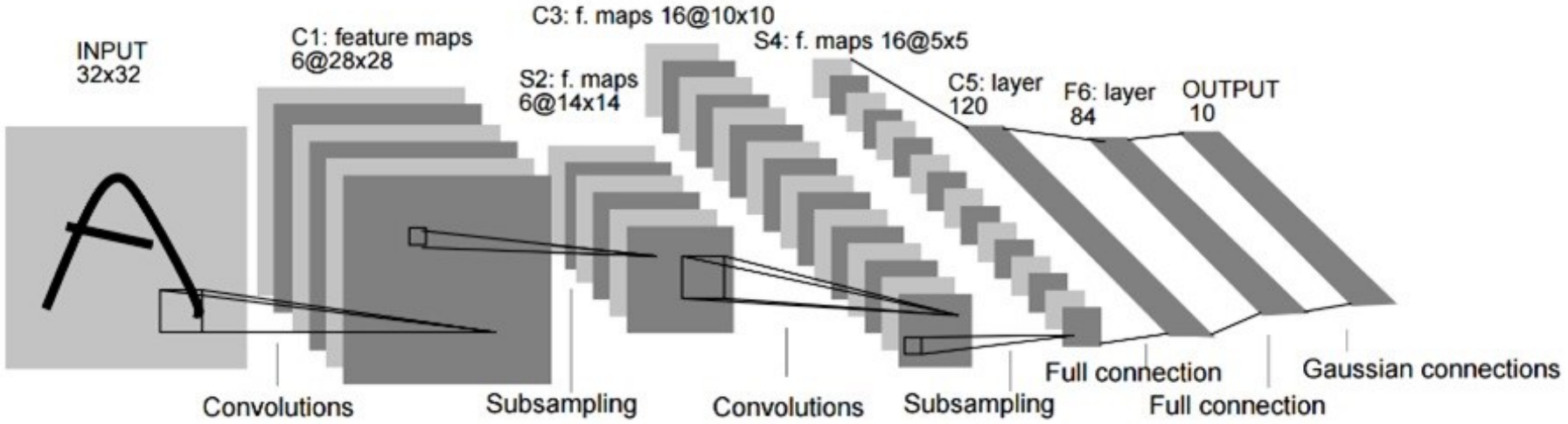


Different normalizations use different definitions of the set $\mathcal{S}_i$(Colored in blue)

Transformer



LeNet [LeCun et al. 1998]



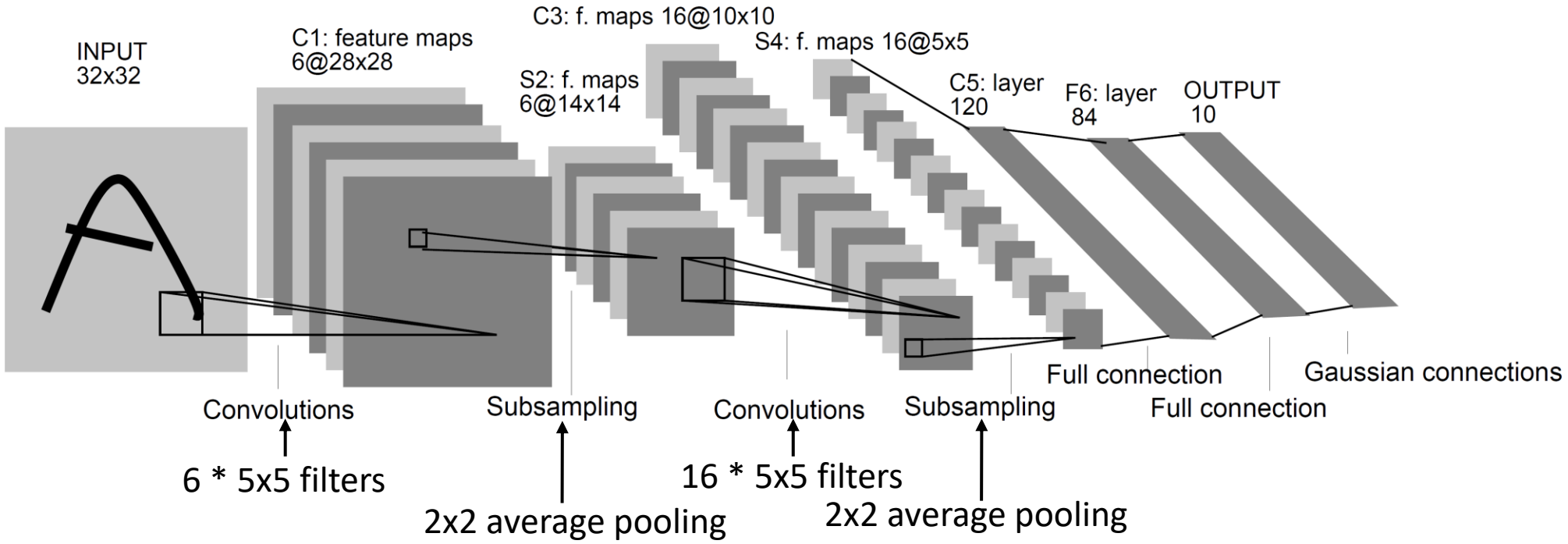
LeNet-1 from 1993

Gradient-based learning applied to document recognition [LeCun, Bottou, Bengio, Haffner 1998]

LeNet-5 Architecture

- CONV Layers: 2
- Fully Connected Layers: 2
- Sigmoid used for non-linearity

Weights: 60k
MACs: 341k

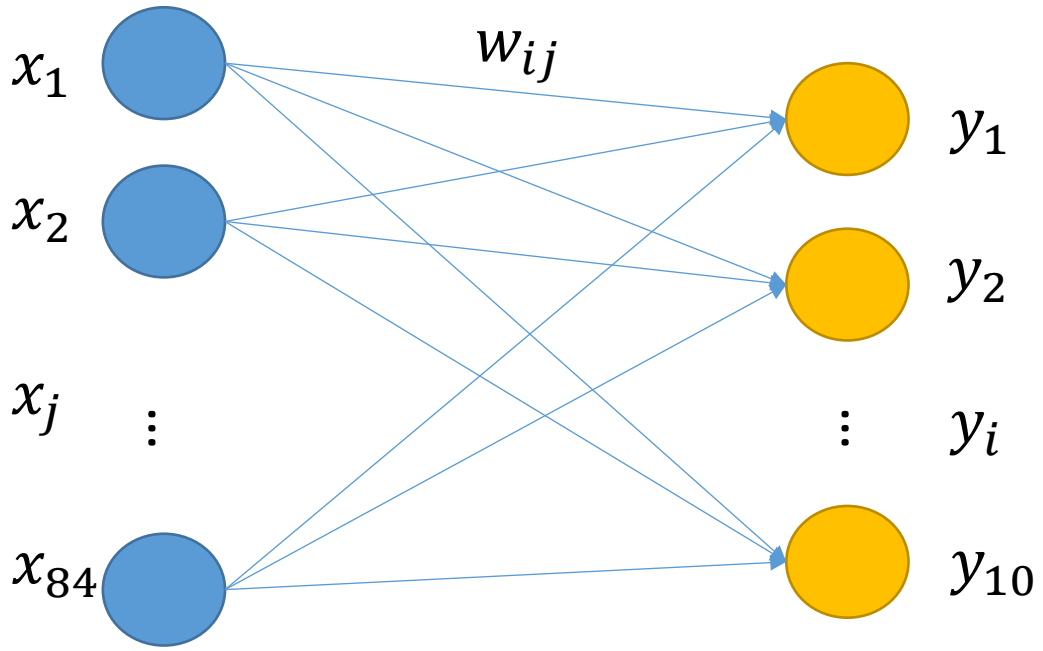


Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

LeNet-5 Output Layer

- Gaussian connection with Euclidean Radial Basis Function units(RBF)

$$y_i = \sum_j (x_i - w_{ij})^2$$



LeNet-5 Loss Function

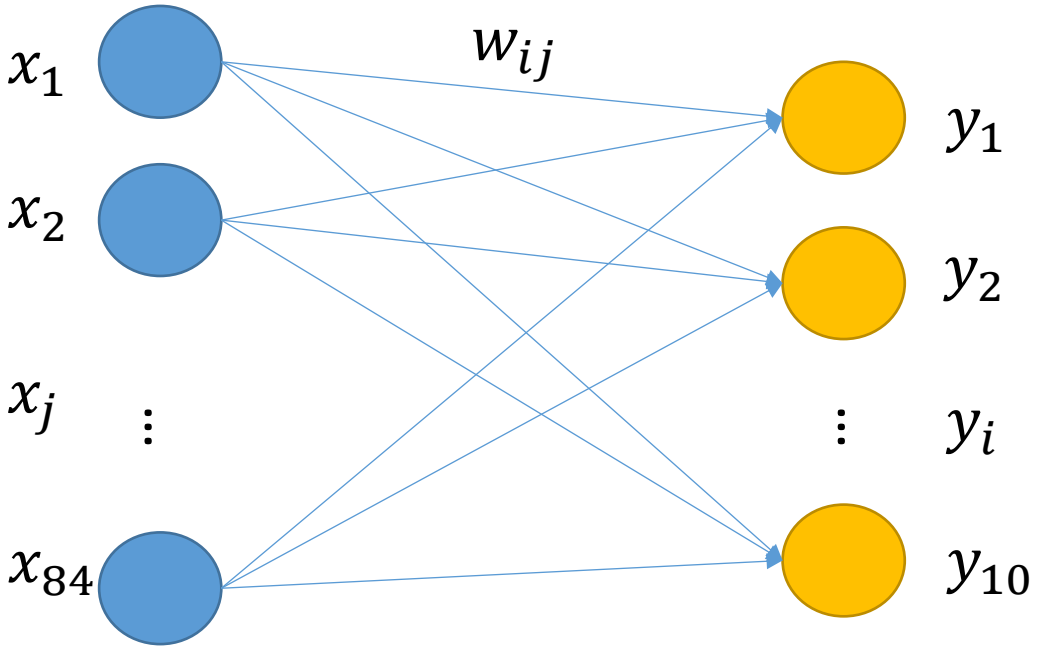
- Loss function
  - Gaussian connection with Euclidean Radial Basis Function units(RBF)

$$\text{minimize } E(W) = \frac{1}{P}\sum_{p=1}^{P}\left(y_{D_p}(Z^p, W) + \log\left(e^{-j} + \sum_i e^{-y_i(Z^p, W)}\right)\right)$$

Pull-down correct class

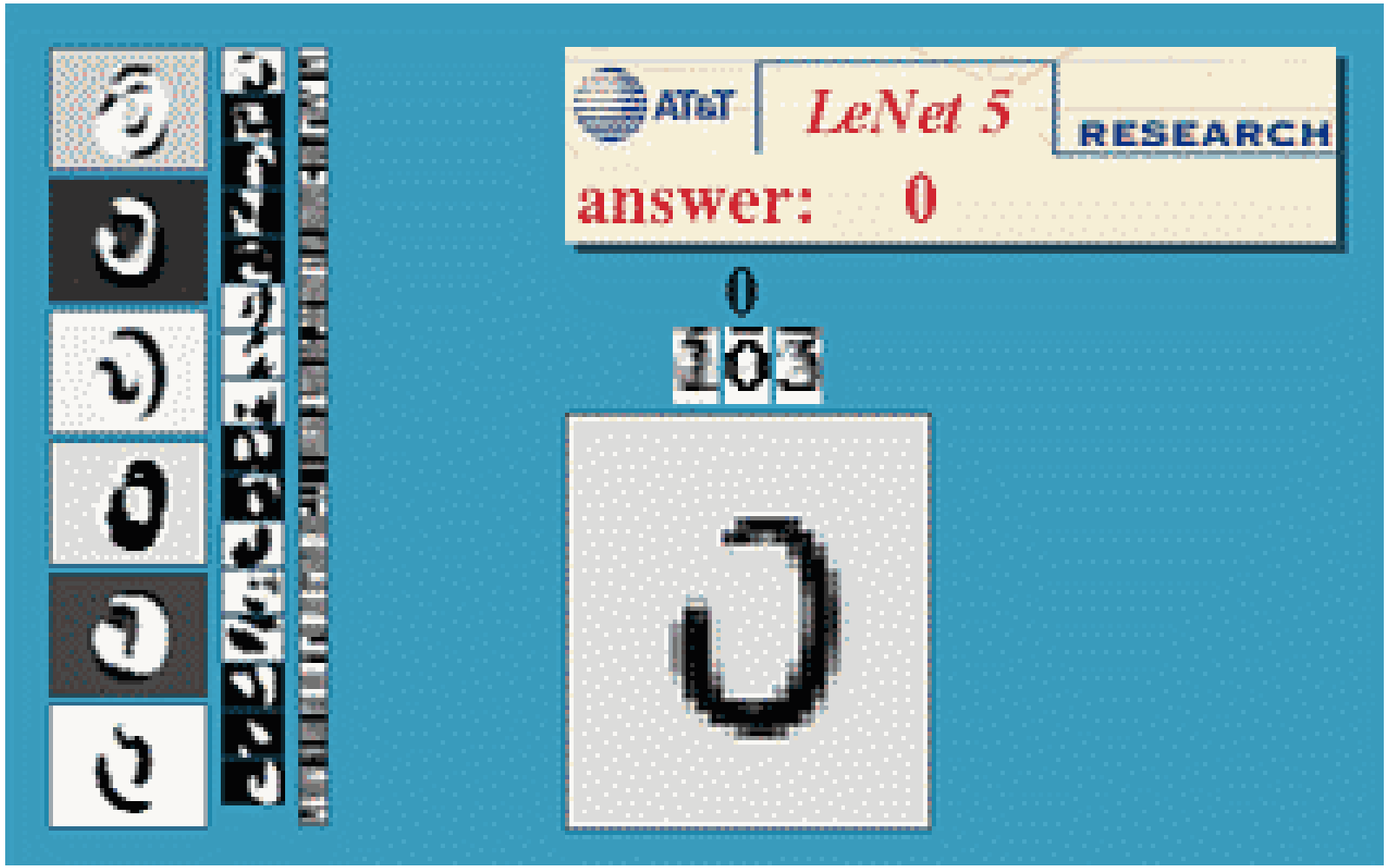
prevents class score too large

Pull-up background class



D_p: correct class of input pattern
P: $samples$
Z^p: input pattern
j: constant

LeNet-5 Example

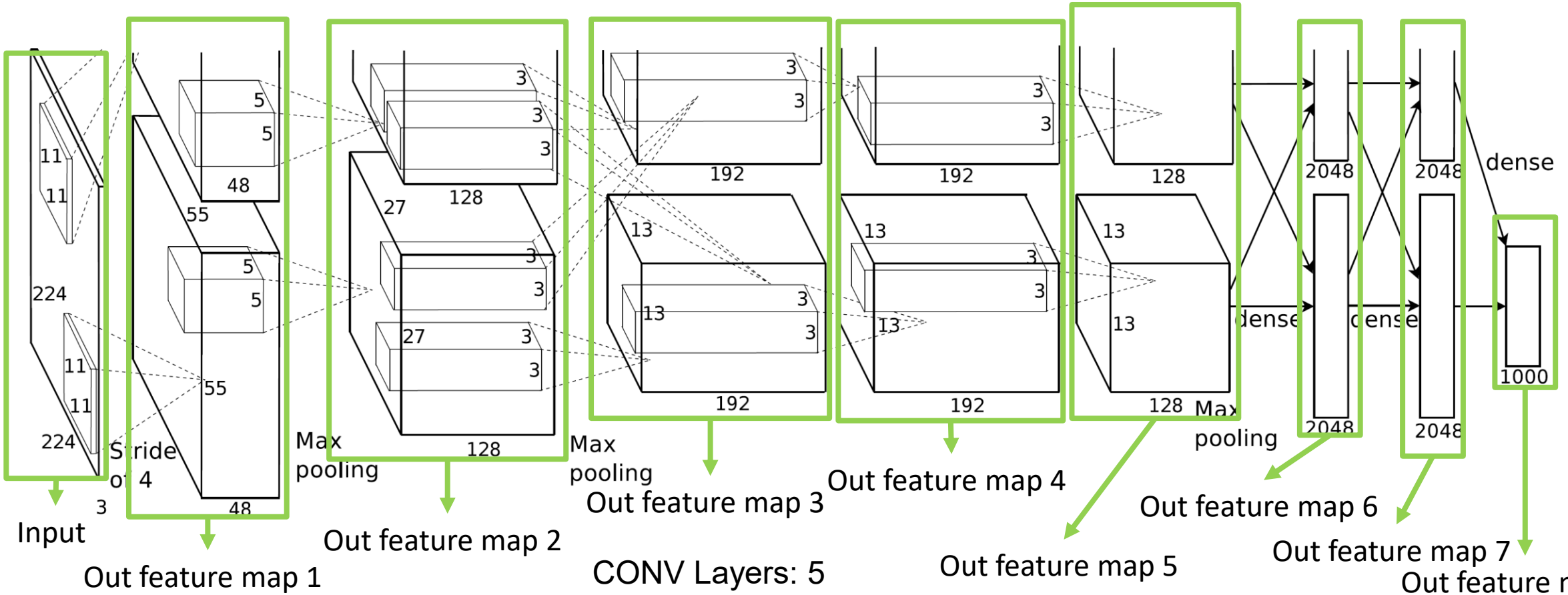


http://yann.lecun.com/exdb/lenet/

AlexNet Architecture

ILSCVR12 Winner !

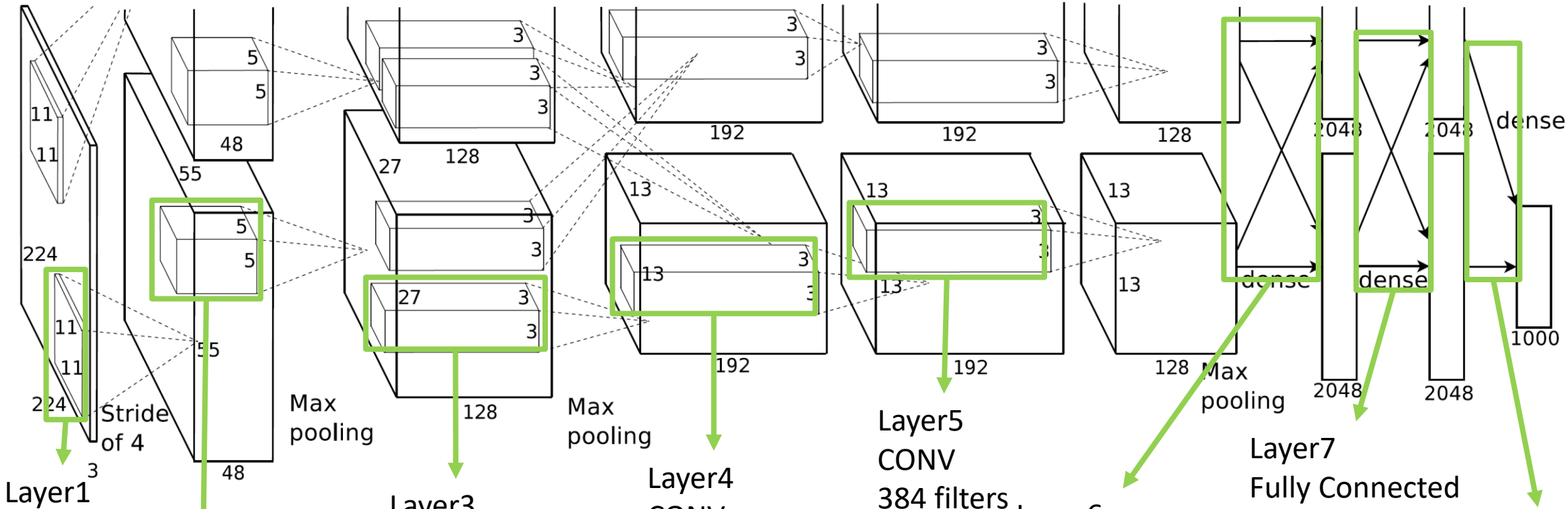
Weights: 61M
MACs: 724M



CONV Layers: 5
Fully Connected Layers: 3
ReLU used for non-linearity

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

AlexNet Architecture



Layer1
CONV
96 filters
11x11x3

Layer2
CONV
256filters
5x5x48

Layer3
CONV
384 filters
3x3x256

Layer4
CONV
384 filters
3x3x192

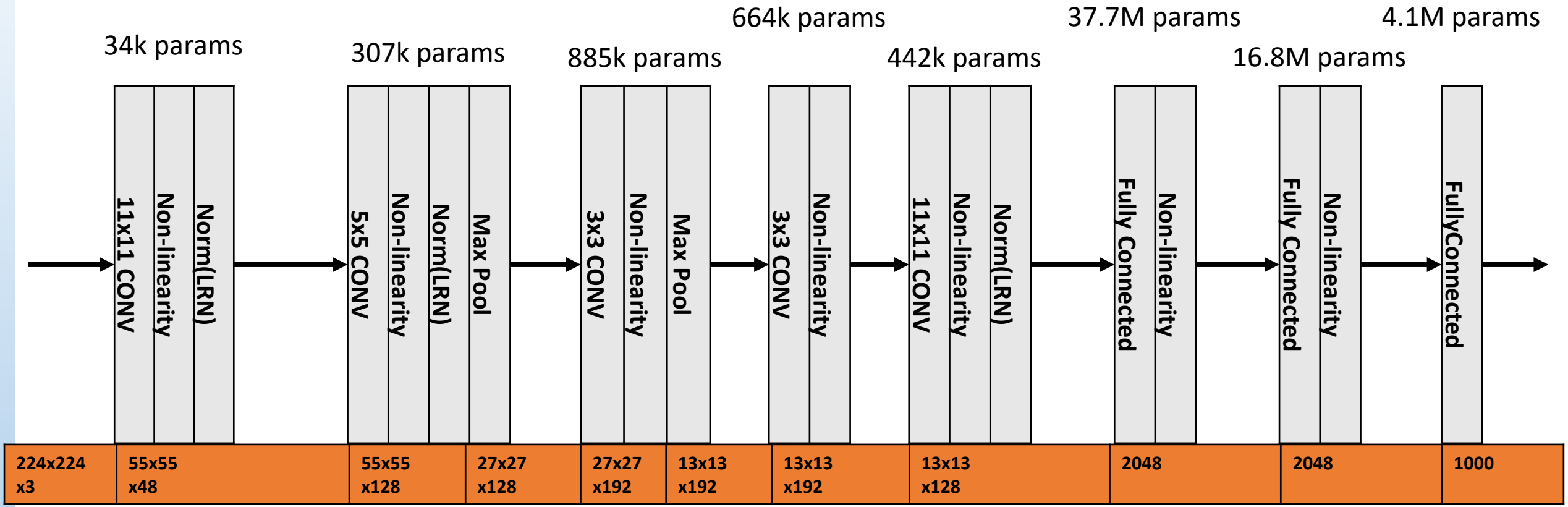
Layer5
CONV
384 filters
3x3x192

Layer6
Fully Connected
4096 filters
13x13x256

Layer7
Fully Connected
4096 filters
4096x1

Layer8
Fully Connected
1000 filters
4096x1

AlexNet Dataflow (Simplified)



Local Response Normalization (LRN)

- Help generalization

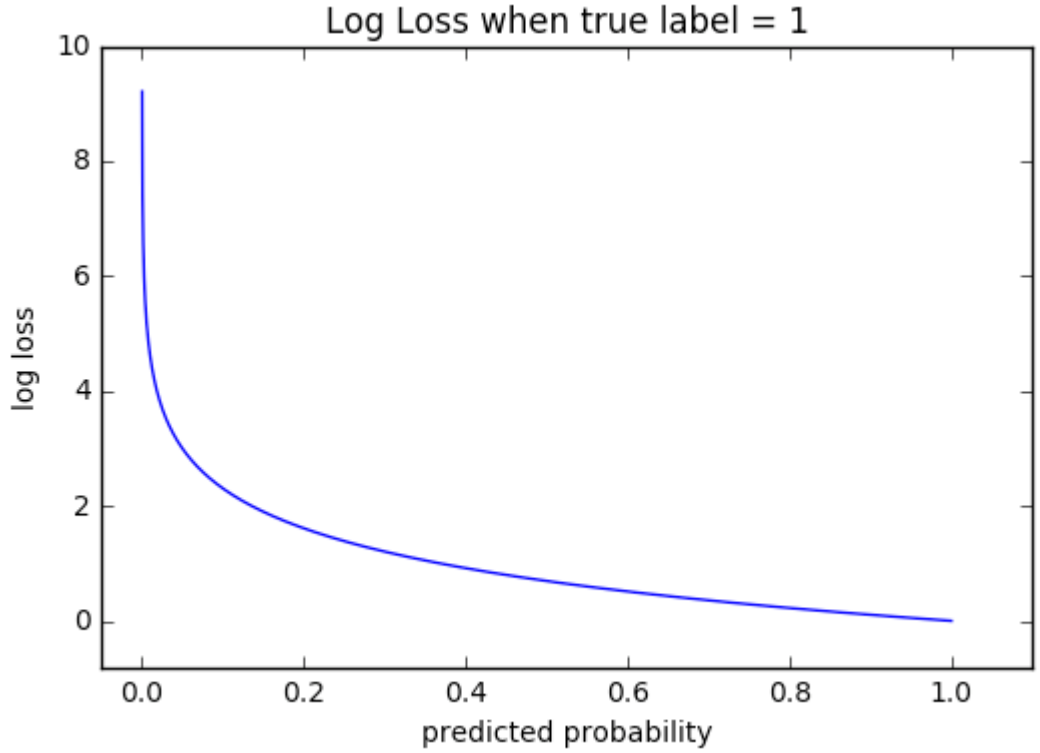
$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^{\beta}$$

  - $\mathrm{n} = 5, K = 2, \alpha = 10^{-4}, \beta = 0.75$ => heuristic
  - N: # of channels, a: output of CONV at location x,y

- Limited improvement in modern DNNs
  - Seldom used recently

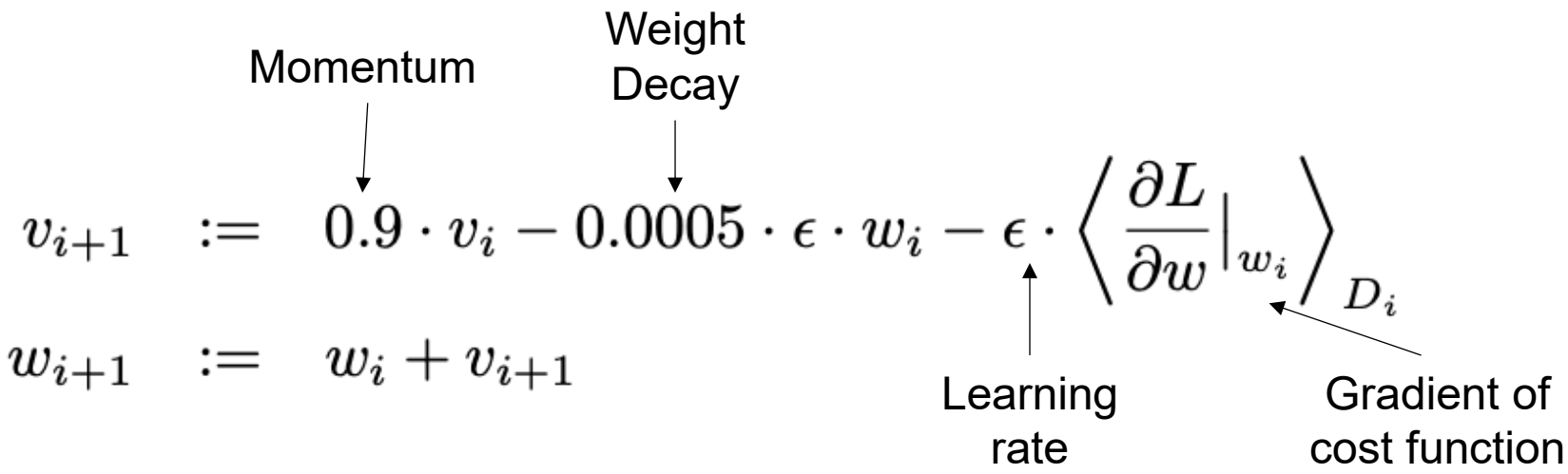
AlexNet Cost(Loss) Function

- Minimize the cross-entropy loss function
- Measures the performance of a classification model whose output is a probability value between 0 and 1
- $CF = -\frac{1}{N}(\sum_{i-1}^{N} y_i \cdot \log(\hat{y}_i))$



AlexNet Optimization Method

- Stochastic Gradient Descent
  - Batch size = 128
  - Update rule:



Outline

- Deep Feedforward Networks
- Regularization
- Optimization
- Convolutional Neural Networks
- Practical Methods

Three Step Process

- Use needs to define metric-based goals
  - High accuracy or low accuracy?
  - Surgery robot: high accuracy
  - Celebrity look-a-like app: low accuracy
- Build an end-to-end system
  - Get up and running ASAP
  - Build the simplest viable system first
  - What baseline to start with though?
  - Copy state-of-the-art from related publication
- Data-driven refinement
  - Choose what to do based on data
  - Don’t believe hype
  - Measure train and test error
  - Overfitting versus underfitting

Choose Metircs

- Accuracy
  - % of examples correct
- Coverage
  - % of examples processed
- Precision
  - % of detections that are right
- Recall
  - % of objects detected
- Amount of error
  - For regression problems

Deep or Not

- Lots of noise, little structure
  - =>Not deep
- Little noise, complex structure
  - =>Deep
- Good shallow baseline:
  - Use what you know
  - Logistic regression, SVM, boosted tree are all good

Choosing Architecture Family

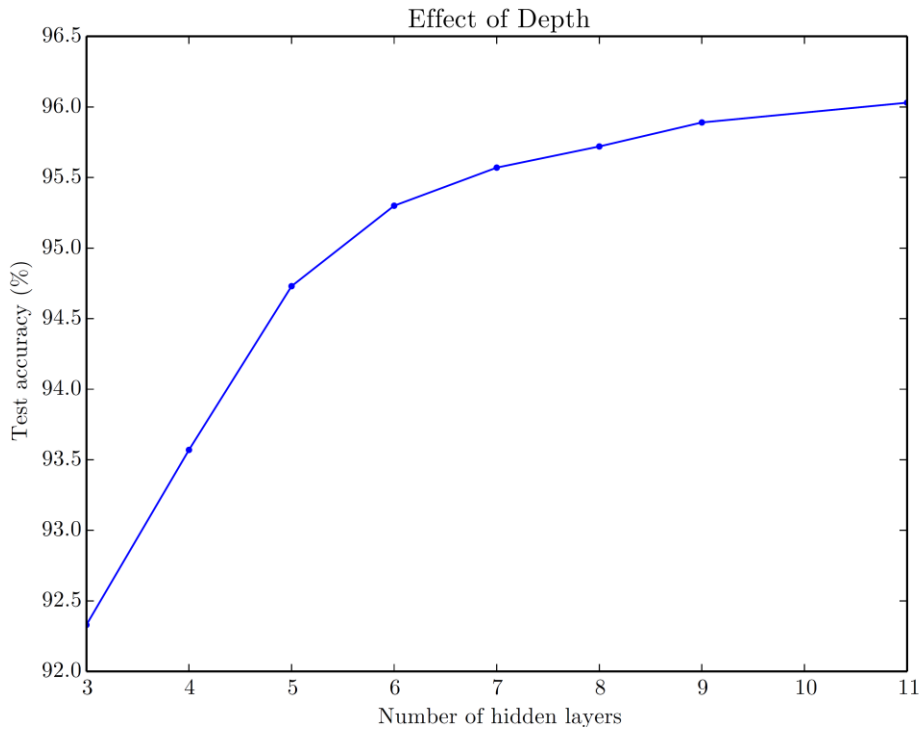
- No structure
  - =>Fully connected
- Spatial structure
  - =>Convolutional
- Sequential structure
  - =>Recurrent

Baselines

- Fully Connected
  - 2-3 hidden layer feed-forward neural network
    - AKA “multilayer perceptron”
  - Rectified linear units
  - Batch normalization
  - Adam
  - Maybe dropout
- Convolutional Network
  - Download a pretrained network
  - Or copy-paste an architecture from a related task
  - Or
    - Deep residual network
    - Batch normalization
    - Adam

High Train Error

- Inspect data for defects
- Inspect software for bugs
  - Don’t roll your own unless you know what you’re doing
- Tune learning rate
  - And other optimization settings
- Make model bigger
- Increasing Depth

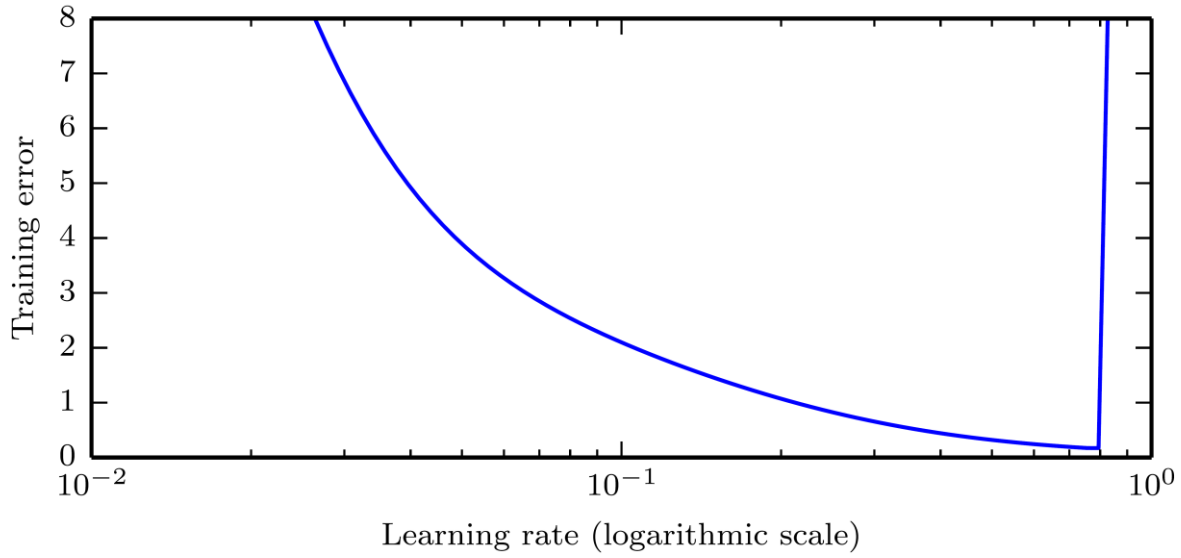


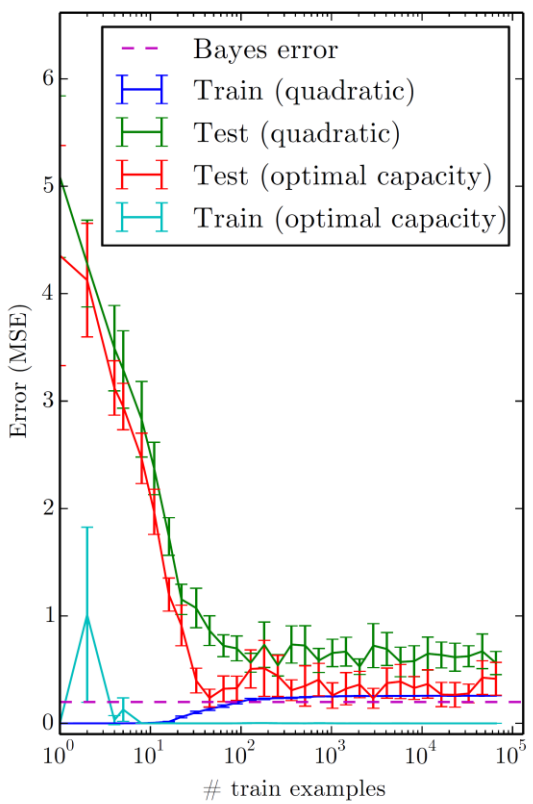
Check Data for Defects

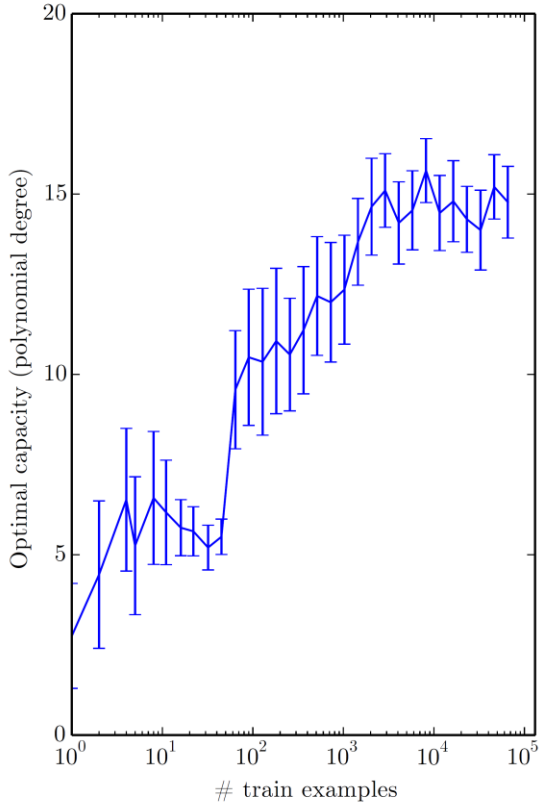
- Can a human process it?

High Testing Error

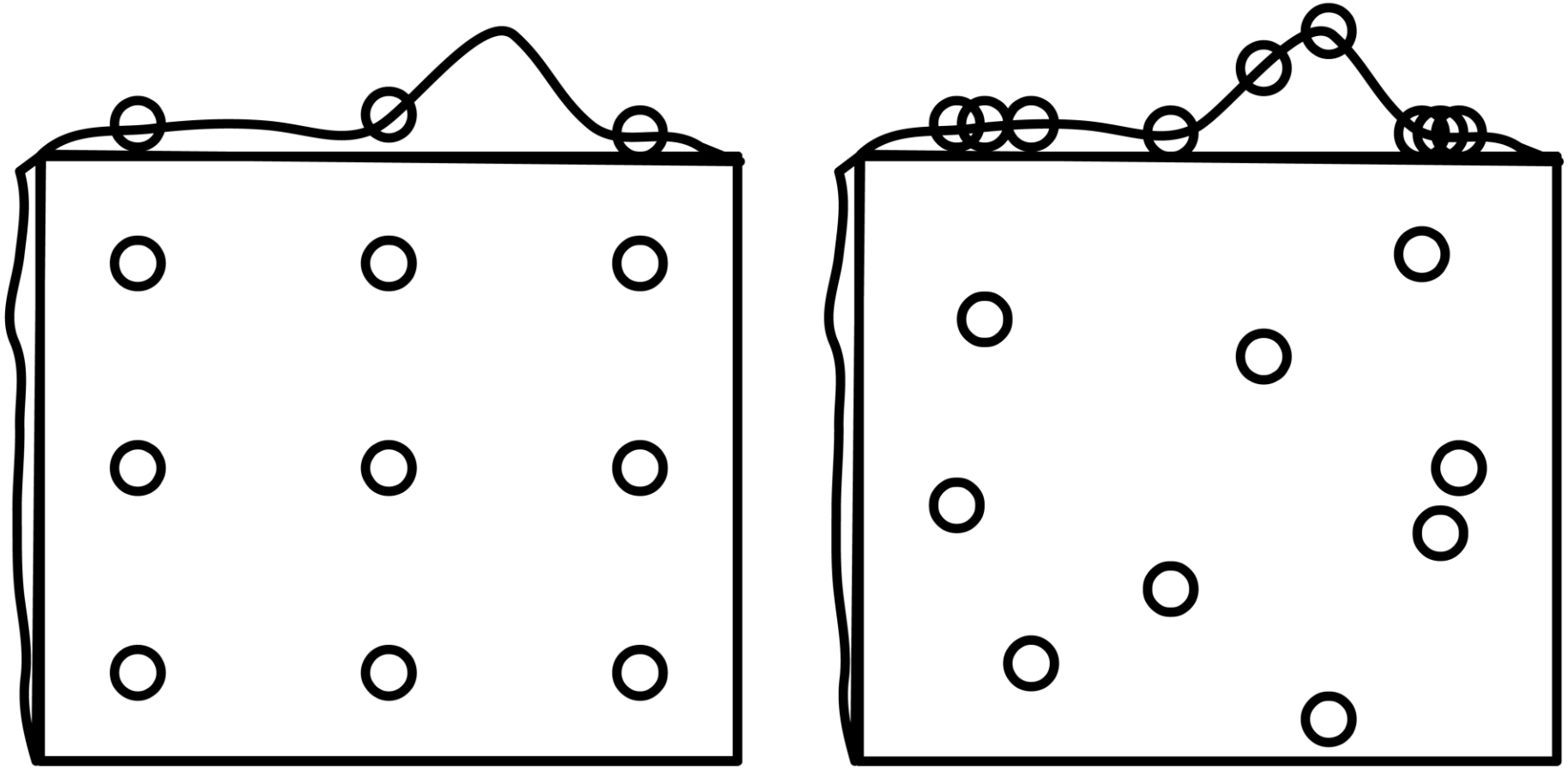
- Add dataset augmentation
- Add dropout
- Increase training set size
  - Collect more data
- Tunning the learning rate







Hyperparameter Search



Grid

Random