SK hynix i-TAP 반도체 Data Scientist를 위한 ML/DL 심화 커리큘럼

1강. Convolutional Neural Networks

Ernest K. Ryu (류경석)

2020.11.06

1강. Outline

- Basics of (non-adaptive) SGD
- PyTorch as a GPU-computing numerical library
- Backpropagation
- Multilayer perceptron
- Convolutional neural networks

Optimization

We consider

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^{N} f_i(\theta)$$

where $f_1, \dots, f_N$ are "differentiable" functions.

In DL, the ReLU activation function $\sigma(x) = \max(0, x)$ is said to be "differentiable".

Gradient Descent (GD)

Define $F(\theta) = \frac{1}{N}\sum_{i=1}^{N} f_i(\theta)$. Then

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \quad F(\theta)$$

GD:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla F(\theta^k)$$

where $\alpha_0, \alpha_1, \ldots \in \mathbb{R}$ are stepsizes.

Since $\nabla F = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i$, can parallelize $\nabla F(\theta^k)$ computation.

Why does GD converge?

Taylor expansion of F about θ^k:

$$F(\theta) = F\left(\theta^k\right) + \nabla F\left(\theta^k\right)^{\mathrm{T}}\left(\theta - \theta^k\right) + \mathcal{O}\left(\left(\theta - \theta^k\right)^2\right)$$

Plug in θ^{k+1}:

$$F\left(\theta^{k+1}\right) = F\left(\theta^k\right) - \alpha_k \left\|\nabla F\left(\theta^k\right)\right\|^2 + \mathcal{O}\left(\alpha_k^2\right)$$

$-\nabla F\left(\theta^k\right)$ is steepest descent direction. For small (cautious) α_k, GD step reduces function value.

Stochastic Gradient Descent (SGD)

We consider

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad \frac{1}{N}\sum_{i=1}^{N} f_i(\theta) = \mathbb{E}_{I\sim\text{Uniform}\{1,\ldots,N\}}\left[f_I(\theta)\right]$$

SGD:

$$i(k)\sim\text{Uniform}\{1,\ldots,N\}$$
$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{i(k)}(\theta^k)$$

where $i(k)$ are uniform IID indices.

Why does SGD converge?

$\nabla f_{i(k)}(\theta^k)$ is an unbiased estimate of the gradient $\nabla F(\theta^k)$

$$\mathbb{E}_{i(k)} \nabla f_{i(k)}(\theta^k) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\theta^k) = \nabla F(\theta^k)$$

(So $\nabla f_{i(k)}(\theta^k)$ is a “stochastic gradient” of F at θ^k)

Why does SGD converge?

Plug θ^{k+1} into Taylor expansion of F about θ^k:

$$F\left(\theta^{k+1}\right) = F\left(\theta^k\right) - \alpha_k \nabla F\left(\theta^k\right)^{\mathrm{T}} \nabla f_{i(k)}\left(\theta^k\right) + \mathcal{O}\left(\alpha_k^2\right)$$

Expectation on both sides:

$$\mathbb{E}_k F\left(\theta^{k+1}\right) = F\left(\theta^k\right) - \alpha_k \left\|\nabla F\left(\theta^k\right)\right\|^2 + \mathcal{O}\left(\alpha_k^2\right)$$

($\mathbb{E}_k$ is expectation conditioned on θ^k)

$-\nabla f_{i(k)}\left(\theta^k\right)$ is descent direction <u>in expectation</u>. For small (cautious) α_k, SGD step reduces function value <u>in expactation</u>.

SGD with general expectation

Consider general expectation

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad \mathbb{E}_\omega\left[f_\omega(\theta)\right]$$

where ω is a random variable. Expectation (not finite sum) appears when you have generative model of ω. E.g. GAN.

SGD:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\omega^k}(\theta^k)$$

where $\omega^0, \omega^1, \dots$ IID random samples of ω.

Mini-batch SGD

For each k, let $i(k,1),\dots,i(k,B)$ be IID indices.

$$\frac{1}{B}\sum_{j=1}^{B}\nabla f_{i(k,j)}(\theta^k)$$

is also an unbiased estimate of $\nabla F(\theta^k)$ since

$$\mathbb{E}\frac{1}{B}\sum_{j=1}^{B}\nabla f_{i(k,j)}(\theta^k) = \frac{1}{B}\sum_{j=1}^{B}\mathbb{E}\nabla f_{i(k,j)}(\theta^k) = \frac{1}{B}\sum_{j=1}^{B}\nabla F(\theta^k) = \nabla F(\theta^k)$$

Mini-batch SGD

Mini-batch SGD:

```
$g = 0$
For $j = 1, \ldots, B$
    $i(k,j) \sim \mathrm{Uniform}\{1, \ldots, N\}$
    $g = g + \frac{1}{B} \nabla f_{i(k,j)}$
$\theta^{k+1} = \theta^k - \alpha_k g$
```

is also an instance of SGD.

Mini-batch Size

Mathematically (measuring performance per iteration)

- Use large batch is when noise/randomness is large.
- Use small batch is when noise/randomness is small.

Practically (measuring performance per unit time)

- Large batch allows more efficient communication and computation, up to the GPU memory limit.
- Often best to increase batch size up to the GPU memory limit.

Cyclic (mini-batch) SGD

Cyclic SGD:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{i(k)}(\theta^k)$$

where $i(k)$ is selected in a cyclic order.

Can also write as:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\mathrm{mod}(k,N)+1}(\theta^k)$$

Strictly speaking, is not an instance of SGD as unbiased estimation property lost.

Epoch in Optimization and Training

Epoch: loosely defined as computation time of computing 1 full gradient. One iteration of GD is, by definition, an epoch. N iterations of SGD constitute an epoch.

Epoch is a convenient unit for counting iterations (rather than directly counting iteration numbers).

Cyclic (mini-batch) SGD

Cyclic SGD advantage:

- Simpler than SGD
- Uses all datapoints within single epoch.

Cyclic SGD disadvantage:

- Worse than SGD in some cases, theoretically and empirically.
- In deep learning, neural nets learn to anticipate cyclic order.

Shuffled Cyclic (mini-batch) SGD

Shuffled cyclic SGD:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{i(k)}(\theta^k)$$

where $i(k)$ is selected in a cyclic order shuffled every epoch.

Can also write as:

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\sigma^{\lfloor \frac{k}{N} \rfloor}_{(\mathrm{mod}(k,N)+1)}}(\theta^k)$$

where $\sigma^0, \sigma^1, \ldots$ is a sequence of random permutations.

Shuffled Cyclic (mini-batch) SGD

Shuffled Cyclic SGD:

```
For e = 1, … , $E$      //for each epoch
  $\sigma \sim \mathrm{randomPermuation}(N)$
  For i = 1, … , $N$
    $\theta^{k+1} = \theta^k - \alpha_k \nabla f_{\sigma(k)}(\theta^k)$
    $k = k + 1$
```

Shuffled Cyclic (mini-batch) SGD

Shuffled cyclic SGD advantage:

- Uses all datapoints within single epoch.
- Network cannot learn to anticipate data order.
- Generally best performance.

Shuffled cyclic SGD advantage:

- Not as simple. (But PyTorch makes it simple to use.)
- Theory not as strong as regular SGD.

PyTorch: GPU Numerical Computing Library

PyTorch is a machine learning library of Python, but PyTorch is fundamentally a numerical computation library.

Features of PyTorch that make itsuitable for using neural networks and machine learning:

- PyTorch supports easy GPU computation.
- Automatic differentiation.
- Numerous ML libraries and sample code.

GPU Computing Code With CUDA

GPU computing opeations:

- cudaMemcpy (CPU→GPU or GPU→CPU)
- CPU code
- GPU kernel calls (CPU instructs GPU to execute computation)

GPU computing workflow:

(i) end data CPU→GPU

(ii) Compute on GPU

(iii) Receive result GPU→CPU

CPU vs. GPU variables

Variables either reside in CPU or GPU memory.

- CPU variable computation on CPU
- GPU variable computation on GPU

CPU and GPU variables cannot directly interact. Can interact only after CPU→GPU or GPU→CPU transfer.

PyTorch Demo

Power iteration example on PyTorch

```
send A from host (CPU) to device (GPU)
send x=x0 from host (CPU) to device (GPU)
for _ in range(100):
  tell GPU to compute x=A*x
send x from device (GPU) to host (CPU)
```

Back Propagation $\subseteq$ Automatic Differentiation

Automates gradient computation! Only need to specify how to evaluate function.

Gradient costs roughly $5 \times$ computation cost[*] of evaluating function.

AutoDiff is not

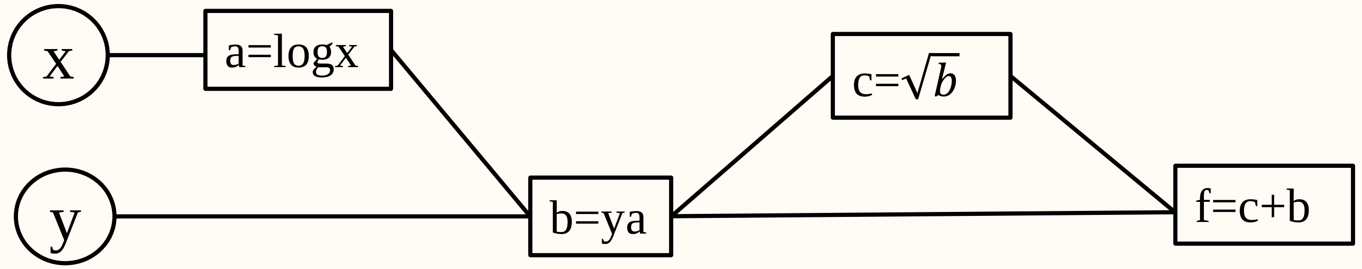
- Finite differencing
- Symbolic differentiation.

AutoDiff $\approx$ chain rule of vector calculus

[*] Depends on computational structure of function, for 5X difference is mostly true for functions (neural networks) used in machine learning.

Chain Rule

- Consider $f(x, y) = y \log x + \sqrt{y \log x}$
- Evaluate f with the computation graph:

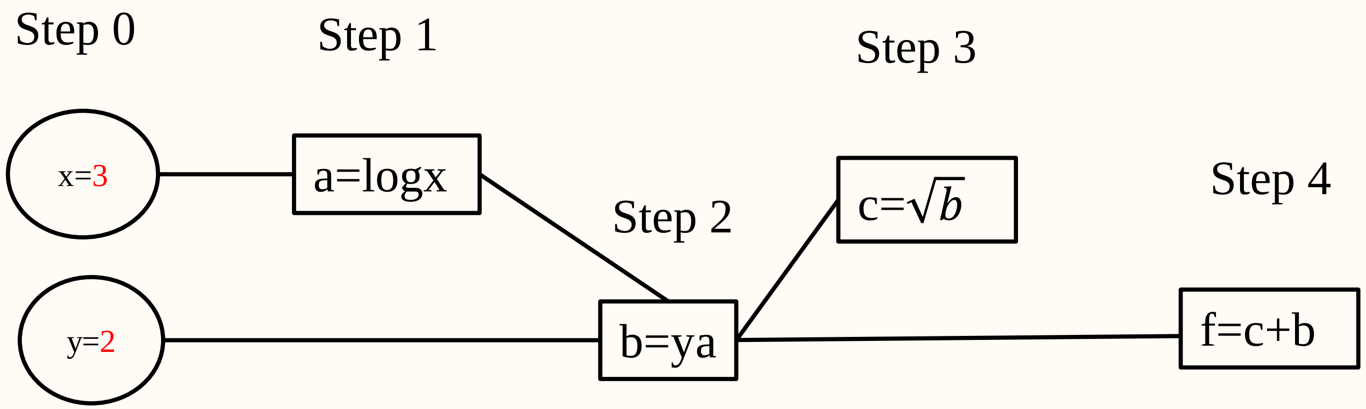


- Chain rule: $$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\frac{\partial x}{\partial x} + \frac{\partial b}{\partial y}\cancelto{0}{\frac{\partial y}{\partial x}}\right) + \frac{\partial f}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\frac{\partial x}{\partial x} + \frac{\partial b}{\partial y}\cancelto{0}{\frac{\partial y}{\partial x}}\right)$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\cancelto{0}{\frac{\partial x}{\partial y}} + \frac{\partial b}{\partial y}\frac{\partial y}{\partial x}\right) + \frac{\partial f}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\cancelto{0}{\frac{\partial x}{\partial y}} + \frac{\partial b}{\partial y}\frac{\partial y}{\partial y}\right)$$

But in what order do you evaluate the chain rule expression?

Forward mode auto-diff



$$0.\ x = 3, y = 2,\ \frac{\partial x}{\partial x} = 1,\ \frac{\partial x}{\partial y} = 0,\ \frac{\partial y}{\partial x} = 0,\ \frac{\partial y}{\partial y} = 1$$

$$1.\ a = logx = log3,\ \frac{\partial a}{\partial x} = \frac{1}{x}\cdot\frac{\partial x}{\partial x},\ \frac{\partial a}{\partial y} = 0$$

$$2.\ b = ya = 2log3,\ \frac{\partial b}{\partial x} = \frac{\partial y}{\partial x}a + y\frac{\partial a}{\partial x} = \frac{2}{3},\ \frac{\partial b}{\partial y} = \frac{\partial y}{\partial y}a + y\frac{\partial a}{\partial y} = a = log3$$

Computation does not involve ‘x’ or derivatives of ‘x’

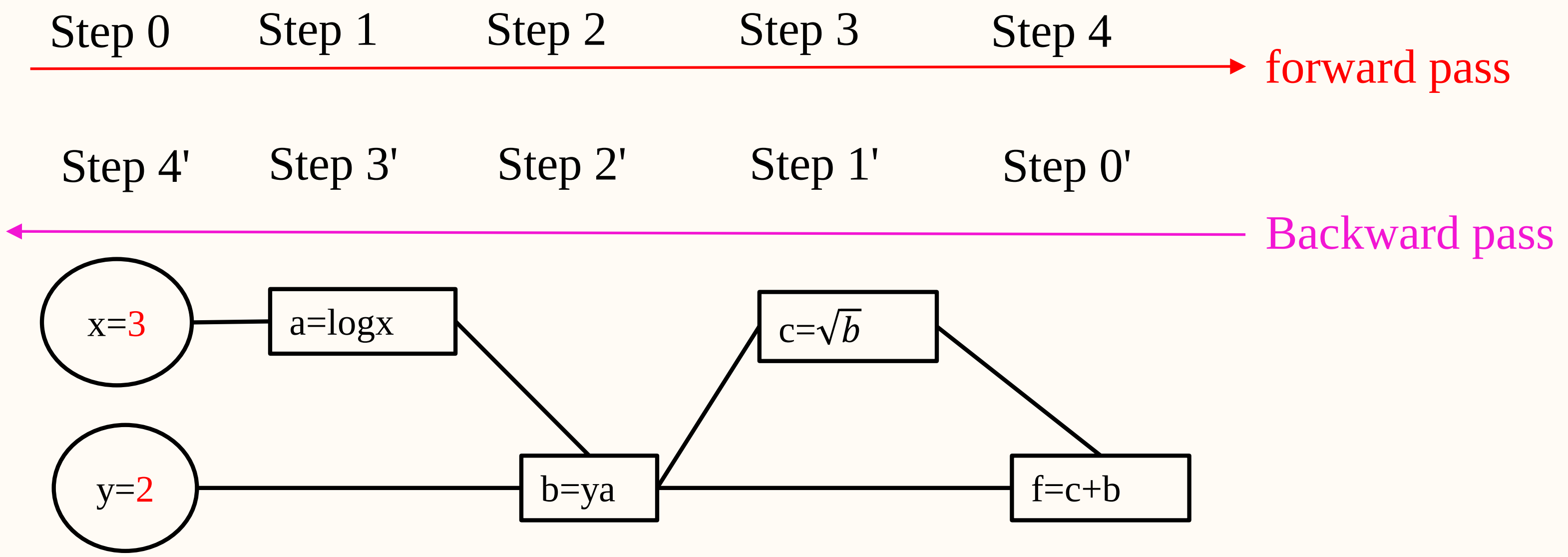
$$3.\ c = \sqrt{b} = \sqrt{2log3},\ \frac{\partial c}{\partial x} = \frac{1}{\sqrt{b}}\frac{\partial b}{\partial x} = \frac{2}{3\sqrt{2log3}},\ \frac{\partial c}{\partial y} = \frac{1}{\sqrt{b}}\frac{\partial b}{\partial y} = \sqrt{\frac{\log 3}{2}}$$

Computation only depends on node ‘b’

$$4.\ f = c + b = \sqrt{2log3} + 2log3, \frac{\partial f}{\partial x} = \frac{\partial c}{\partial x} + \frac{\partial b}{\partial x} = \frac{2}{3}\left(1 + \frac{1}{\sqrt{2\log 3}}\right), \frac{\partial f}{\partial y} = \frac{\partial c}{\partial y} + \frac{\partial b}{\partial y} = \sqrt{\frac{log3}{2}} + log3$$

Computation only depends on node ‘b’ and ‘c’

Reverse mode auto-diff (Backpropagation)



0. $x = 3,\ y = 2$
1. $a = log3$
2. $b = 2log3$
3. $c = \sqrt{2log3}$
4. $f = \sqrt{2log3} + 2log3$

0'. $\frac{\partial f}{\partial f} = 1$

1'. $\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} = 1$

2'. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} = \frac{1}{\sqrt{b}} + 1 = \frac{1}{\sqrt{2log3}} + 1$

3'. $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} = 2(\frac{1}{\sqrt{2log3}} + 1)$

4'. $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = \frac{2}{3}(\frac{1}{\sqrt{2log3}} + 1)$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial y} = \left(\frac{1}{\sqrt{2log3}} + 1\right)a = \sqrt{\frac{\log 3}{2}} + \log 3$$

Backward pass depends on node values computed in forward pass.

Autodiff by Jacobian multiplication

Consider $g = f_1 \circ f_2 \circ \cdots \circ f_N$ where $f_i : R^{n_i} \to R^{n_{i-1}}$ for $i = 1, \cdots, N$.

Chain rule: $\nabla_x g(x) = \underset{n_0 \times n_1}{Df_1} \quad \underset{n_1 \times n_2}{Df_2} \quad \cdots \quad \underset{n_{N-1} \times n_N}{Df_N}$ D denotes Jacobian.

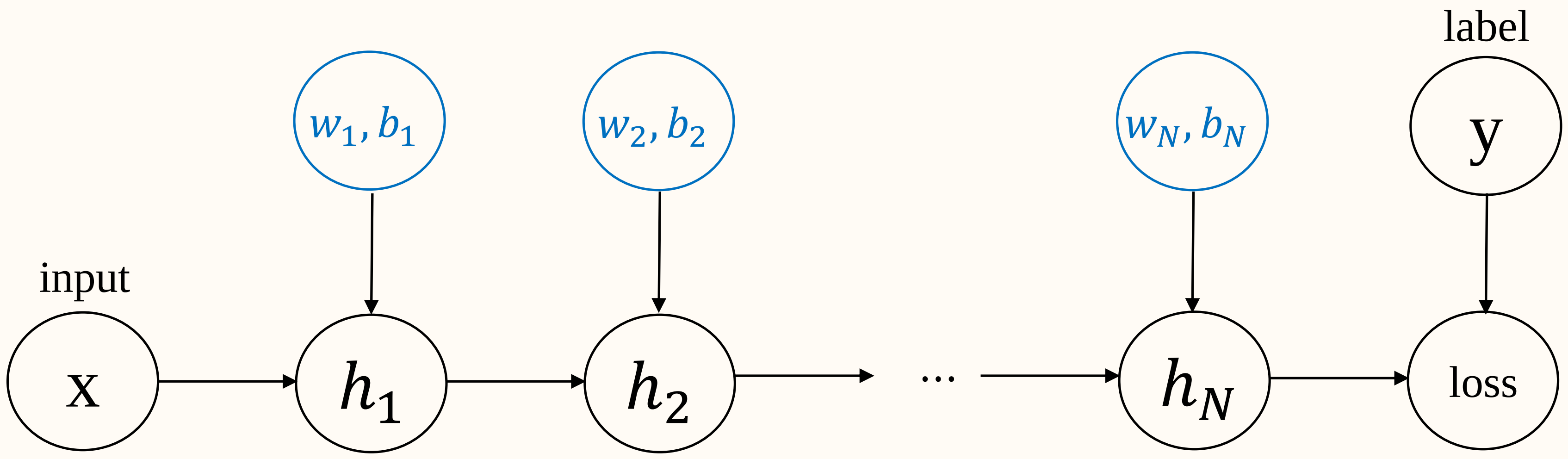
Forward mode: $Df_1(Df_2(\cdots(Df_{N-1}Df_N)\cdots))$

Reverse mode: $(((Df_1\ Df_2)\ Df_3)\ \cdots)\ Df_N$

$\uparrow$

Optimal if $n_0 \le n_1 \le \cdots \le n_N$.
Proof by dynamic programming.

Backprop on multi-layer perceptron



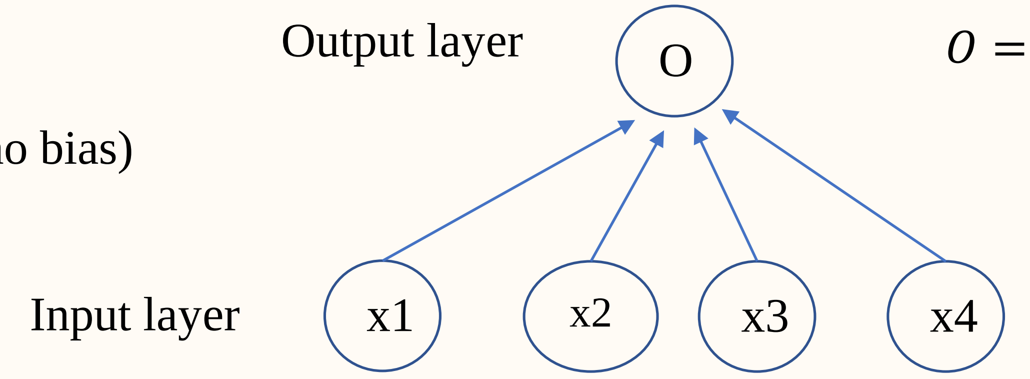
- In NN training, parameters and fixed inputs are distinguished.
- In Pytorch, 1. evaluate the loss function

  2. call ·backward() to perform backward pass and compute gradients.

- When performing the forward pass, intermediate node values are stored so that they can later be used in backward pass.
- In testing loop, we don't compute gradients so this is unnecessary.
- The torch.no_grad() context marger allows intermediate node values to discarded or not be stored. This saves memory and can reduce the time to compute the test loop.

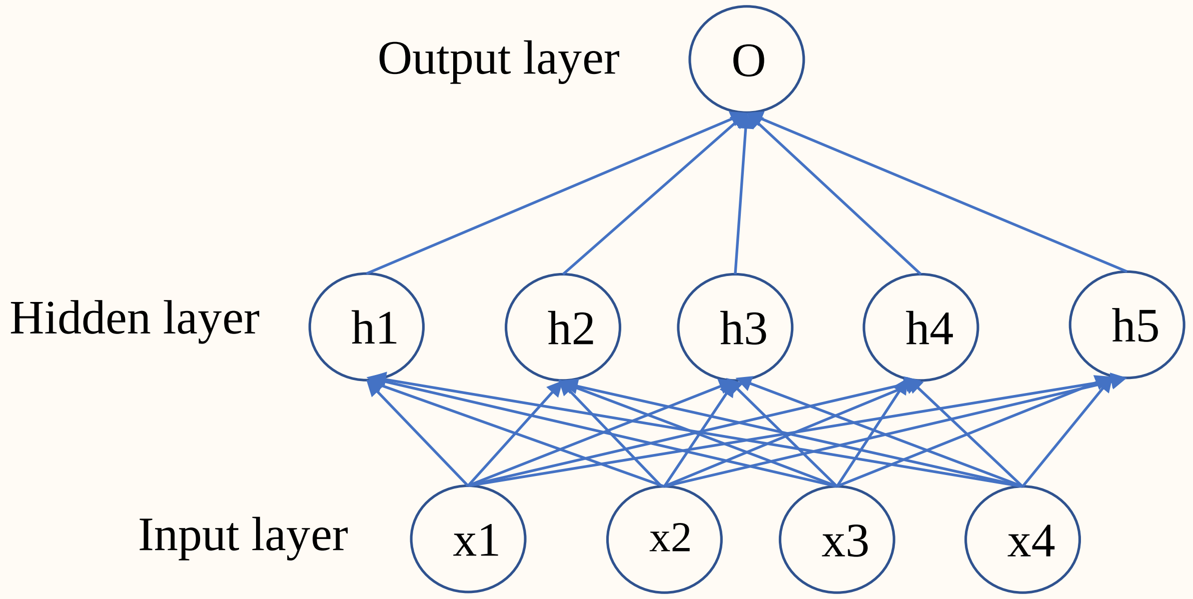
Multilayer Perceptron (fully connected deep neural network)

Logistic Regression (no bias)
1-layer model



$O = w^T x$

2-layer model



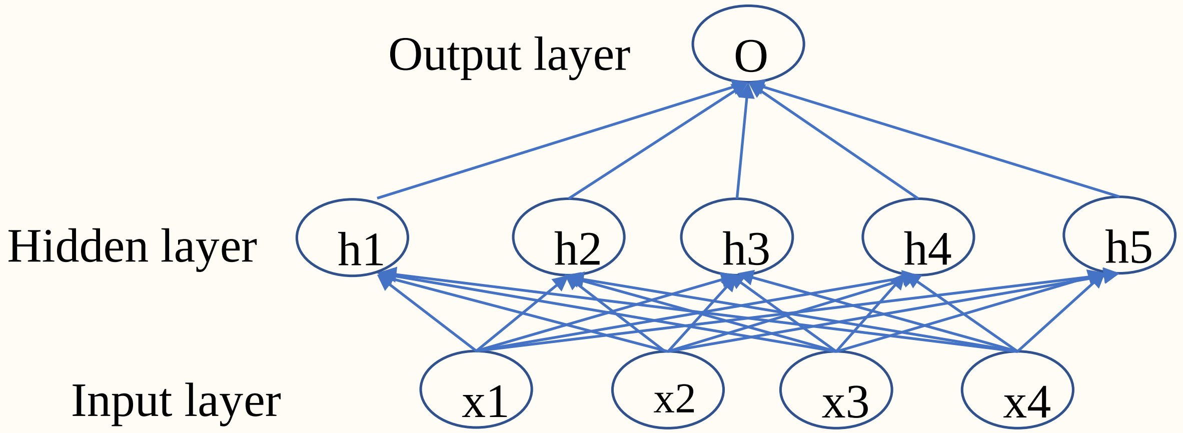
$O = W_2 h$

$h = W_1 x$
(W_1 is 5x4)

$h_i = (W_1)_i x$,
($i = 1,2,3,4,5.$)
W_1 is 1x4

O=$W_2(W_1 x) = (W_2 W_1) x$
(This equation is linear in x !!!)

Deep Neural Network



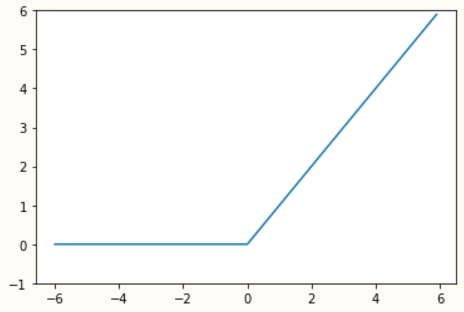
O=$W_2 h$ W_2 is 1×5

$h = \sigma(W_1 x)$ W_1 is 5×4
σ is a non-linear function
Applied elementwise

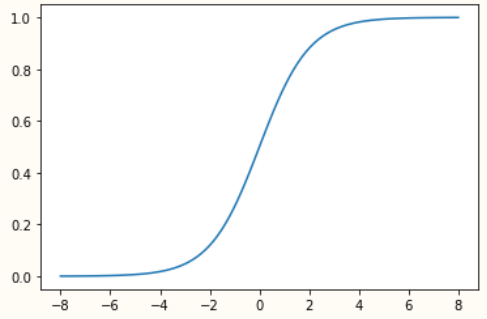
Use non-linear activation functions

Common activation functions

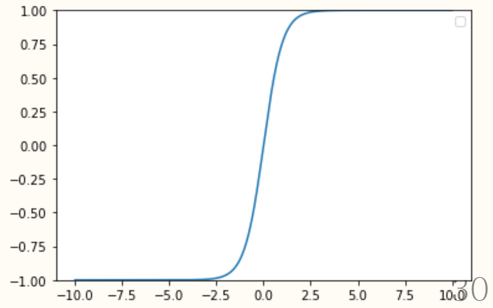
-Rectified Linear Unit (ReLU)
ReLU(z) = max(z,0)



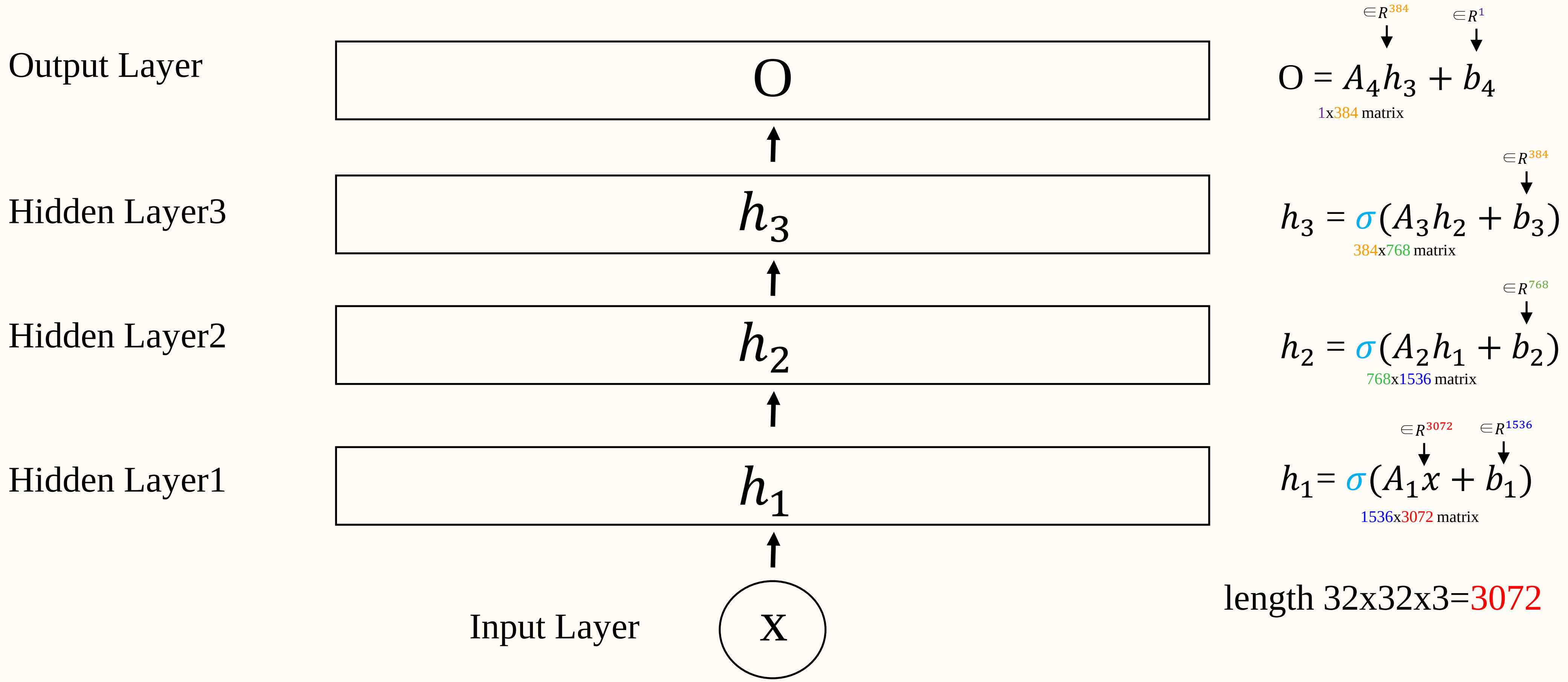
-Sigmoid
Sigmoid(x) = $\frac{1}{1+e^{-x}}$



-Hyperbolic Tangent
tanh(x) = $\frac{1-e^{-2x}}{1+e^{-2x}}$



Architecture for CIFAR10 Binary classification



$\in R^{384}$ $\in R^{1}$

$$O = A_4 h_3 + b_4$$

1x384 matrix

$\in R^{384}$

$$h_3 = \sigma(A_3 h_2 + b_3)$$

384x768 matrix

$\in R^{768}$

$$h_2 = \sigma(A_2 h_1 + b_2)$$

768x1536 matrix

$\in R^{3072}$ $\in R^{1536}$

$$h_1 = \sigma(A_1 x + b_1)$$

1536x3072 matrix

length 32x32x3=3072

Activation function: σ=ReLU

Softmax Regression

Note: $\sum_{i=1}^{k} \mu_i(z) = 1, \mu_i(z) \geq 0$
So we can think of μ as

$$\mu: \mathbb{R}^k \to \mathcal{P}(\{1, \dots, k\})$$

Softmax function $\mu: \mathbb{R}^k \to \mathbb{R}^k$ defined by

$$\mu(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad \text{for } i = 1, \dots, k \quad \text{where } z = (z_1, \dots, z_k) \in \mathbb{R}^k$$

Name softmax is a misnomer:

- ~~$\mu(z) \approx \max(z)$~~
- $\mu(z) \approx \operatorname{argmax}(z)$

$$\mu\left(\begin{bmatrix}1\\2\\3\end{bmatrix}\right) = \begin{bmatrix}0.09\\0.24\\0.67\end{bmatrix}, \quad \mu\left(\begin{bmatrix}999\\0\\-2\end{bmatrix}\right) \approx \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \mu\left(\begin{bmatrix}-2\\-2\\-99\end{bmatrix}\right) \approx \begin{bmatrix}0.5\\0.5\\0\end{bmatrix}$$

Supervised Learning Model for Softmax Regression

Data $x_1, \dots, x_N \in \mathbb{R}^n$. Labels $y_1, \dots, y_N \in \{1, \dots, k\}$. (k classes)

Assume there is an unknown function

$$f : \mathbb{R}^n \to \mathcal{P}(\{1, \dots, k\})$$

Choose model

$$f_{W,b}(x) = \frac{1}{\sum_{i=1}^{k} e^{w_i^T x + b_i}} \begin{bmatrix} e^{w_1^T x + b_1} \\ \vdots \\ e^{w_k^T x + b_k} \end{bmatrix}$$

$$= \mu(Wx + b)$$

$$W = \begin{bmatrix} w_1^T \\ \vdots \\ w_k^T \end{bmatrix} \in \mathbb{R}^{k \times n}$$

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix} \in \mathbb{R}^k$$

Supervised Learning Model for Softmax Regression

Define empirical distribution $\mathcal{P}(y) \in \mathbb{R}^n$ with

$$\left(\mathcal{P}(y)\right)_i = \begin{cases} 1 & \text{if } y = i \\ 0 & \text{otherwise} \end{cases}$$

(also called "one-hot" vector)

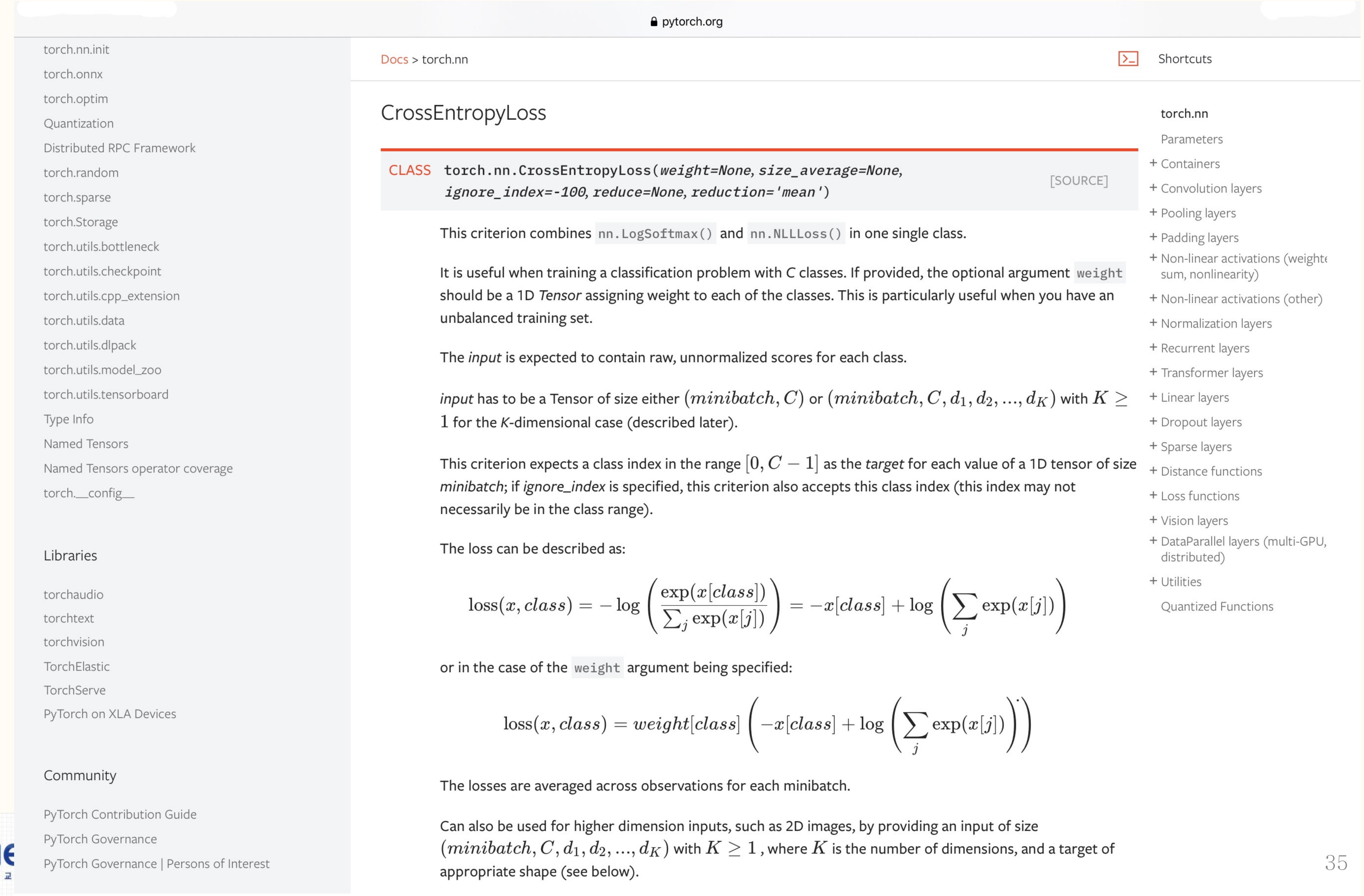
$$\underset{W\in\mathbb{R}^{k\times n}, b\in\mathbb{R}^k}{\text{minimize}} \quad \sum_{i=1}^{N} D_{\text{KL}}(\mathcal{P}(y_i)||f_{W,b}(x_i))$$

KL-divergence

Equivalent to:

$$\underset{W\in\mathbb{R}^{k\times n}, b\in\mathbb{R}^k}{\text{minimize}} \quad \sum_{i=1}^{N} H(\mathcal{P}(y_i), f_{W,b}(x_i))$$

Cross entropy



pytorch.org

torch.nn.init
torch.onnx
torch.optim
Quantization
Distributed RPC Framework
torch.random
torch.sparse
torch.Storage
torch.utils.bottleneck
torch.utils.checkpoint
torch.utils.cpp_extension
torch.utils.data
torch.utils.dlpack
torch.utils.model_zoo
torch.utils.tensorboard
Type Info
Named Tensors
Named Tensors operator coverage
torch.__config__

Libraries

torchaudio
torchtext
torchvision
TorchElastic
TorchServe
PyTorch on XLA Devices

Community

PyTorch Contribution Guide
PyTorch Governance
PyTorch Governance | Persons of Interest

Docs > torch.nn

Shortcuts

CrossEntropyLoss

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

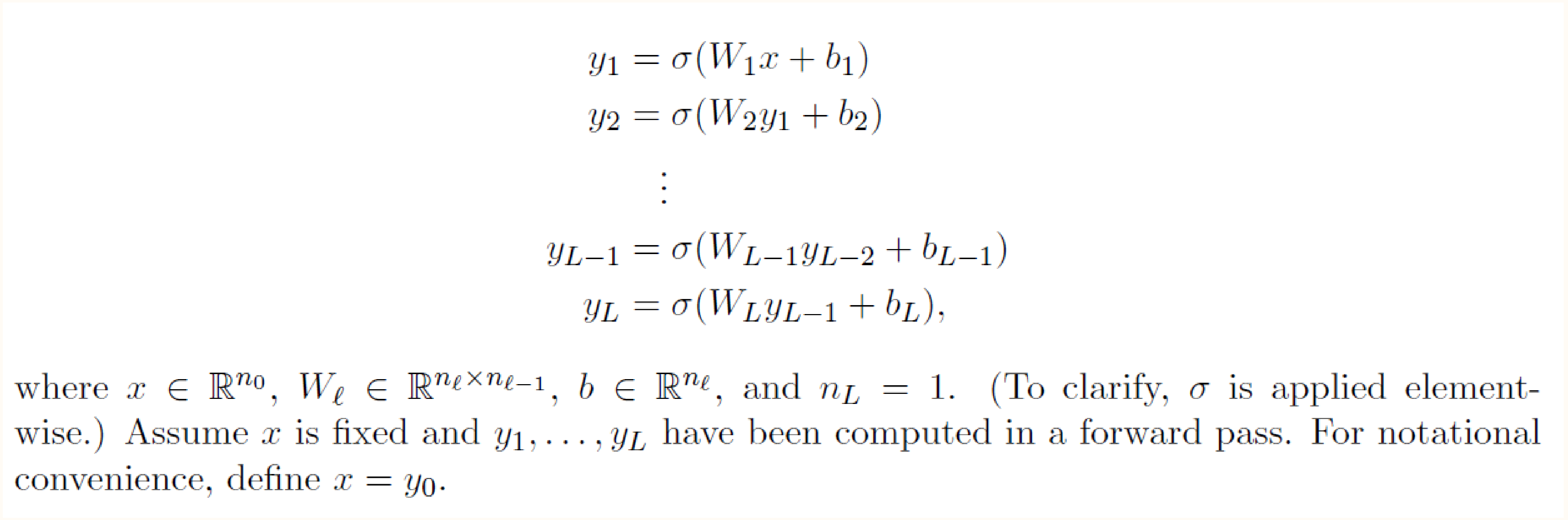
The losses are averaged across observations for each minibatch.

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$, where K is the number of dimensions, and a target of appropriate shape (see below).

torch.nn
Parameters
+ Containers
+ Convolution layers
+ Pooling layers
+ Padding layers
+ Non-linear activations (weighte sum, nonlinearity)
+ Non-linear activations (other)
+ Normalization layers
+ Recurrent layers
+ Transformer layers
+ Linear layers
+ Dropout layers
+ Sparse layers
+ Distance functions
+ Loss functions
+ Vision layers
+ DataParallel layers (multi-GPU, distributed)
+ Utilities
Quantized Functions

Backprop for multi-layer perceptron

Let $\sigma: \mathbb{R} \to \mathbb{R}$ be a differentiable activation function. Consider the multi-layer perceptron



$$\begin{aligned} y_1 &= \sigma(W_1 x + b_1) \\ y_2 &= \sigma(W_2 y_1 + b_2) \\ &\vdots \\ y_{L-1} &= \sigma(W_{L-1} y_{L-2} + b_{L-1}) \\ y_L &= \sigma(W_L y_{L-1} + b_L), \end{aligned}$$

where $x \in \mathbb{R}^{n_0}$, $W_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $b \in \mathbb{R}^{n_\ell}$, and $n_L = 1$. (To clarify, σ is applied element-wise.) Assume x is fixed and $y_1, \ldots, y_L$ have been computed in a forward pass. For notational convenience, define $x = y_0$.

(i) Show

$$\frac{\partial y_\ell}{\partial b_\ell} = \operatorname{diag}\left(\sigma'(W_\ell y_{\ell-1} + b_\ell)\right), \qquad \text{for } \ell = 1, \dots, L$$

and

$$\frac{\partial y_\ell}{\partial y_{\ell-1}} = \operatorname{diag}\left(\sigma'(W_\ell y_{\ell-1} + b_\ell)\right) W_\ell, \qquad \text{for } \ell = 2, \dots, L,$$

where $\frac{\partial y_\ell}{\partial b_\ell} \in \mathbb{R}^{n_\ell \times n_\ell}$ and $\frac{\partial y_\ell}{\partial y_{\ell-1}} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ are Jacobian matrices. (For any $v \in \mathbb{R}^k$, we define $\operatorname{diag}(v)$ to be the $k \times k$ diagonal matrix with $v_1, \dots, v_k$ as its diagonal entries.)

(ii) Using this results of (i), we have

$$\frac{\partial y_L}{\partial b_\ell} = \frac{\partial y_L}{\partial y_{L-1}} \frac{\partial y_{L-1}}{\partial y_{L-2}} \cdots \frac{\partial y_{\ell+1}}{\partial y_\ell} \frac{\partial y_\ell}{\partial b_\ell}, \qquad \text{for } \ell = 1 \dots, L.$$

Which matrix multiplication order corresponds to backpropagation?

(iii) Since y_ℓ is a vector and W_ℓ is a matrix, writing $\frac{\partial y_\ell}{\partial W_\ell}$ would not make sense. However, $y_L \in \mathbb{R}$ is a scalar, so if we endow $\mathbb{R}^{n_\ell \times n_{\ell-1}}$ with the inner product

$$\langle X, Y \rangle = \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_{\ell-1}} X_{ij} Y_{ij}, \qquad \forall X, Y \in \mathbb{R}^{n_\ell \times n_{\ell-1}},$$

then we can write $\frac{\partial y_L}{\partial W_\ell} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$. Show

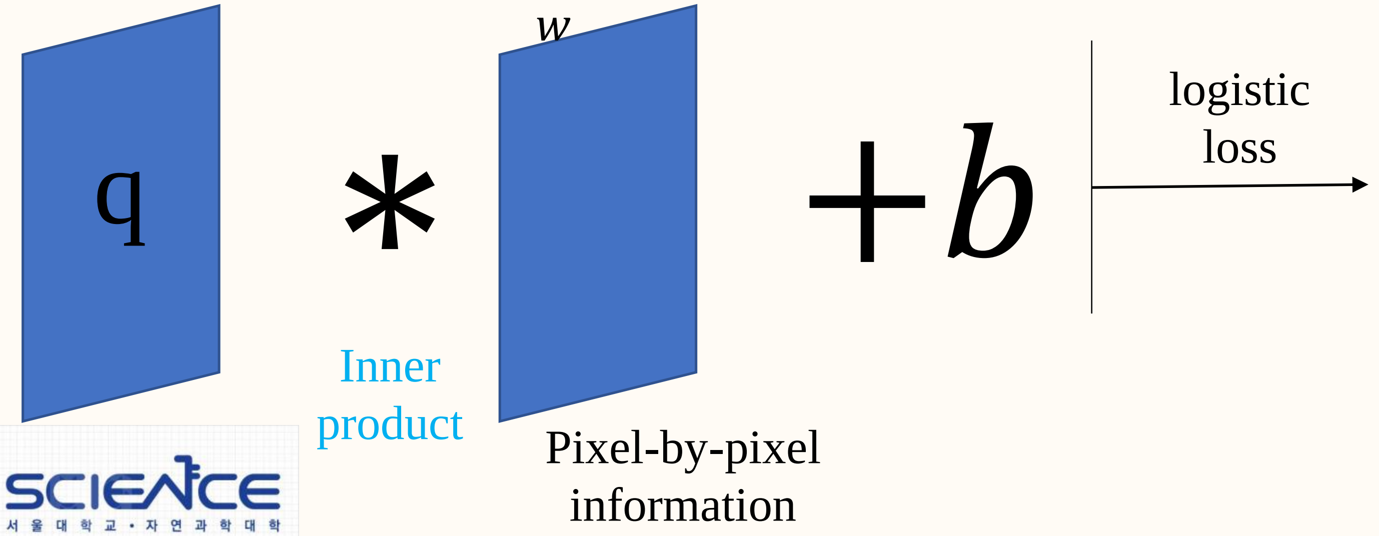
$$\frac{\partial y_L}{\partial W_\ell} = \left(\frac{\partial y_L}{\partial y_\ell}\right)^\intercal \operatorname{diag}\left(\sigma'(W_\ell y_{\ell-1} + b_\ell)\right) y_\ell^\intercal, \qquad \text{for } \ell = 1 \dots, L.$$

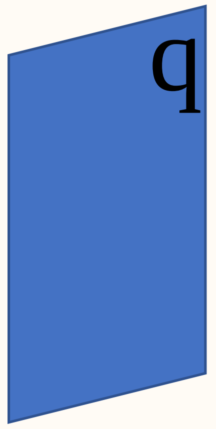
Shift Invariance in Vision to Convolution

Cat

Still a Cat

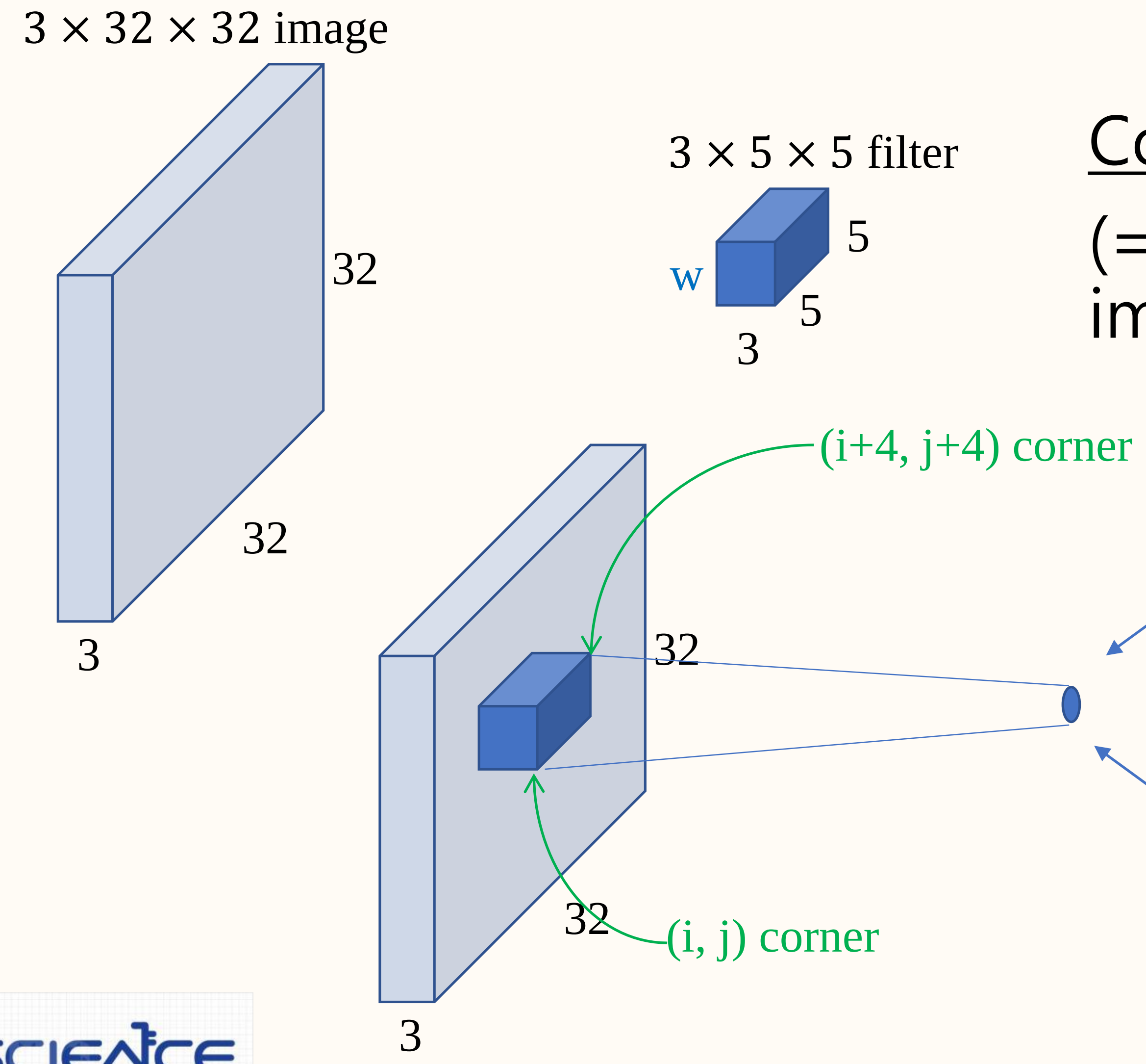
Logistic regression (with a single fully connected layer) does not encode shift invariance





Translating digit changes output significantly

Convolutional Layer

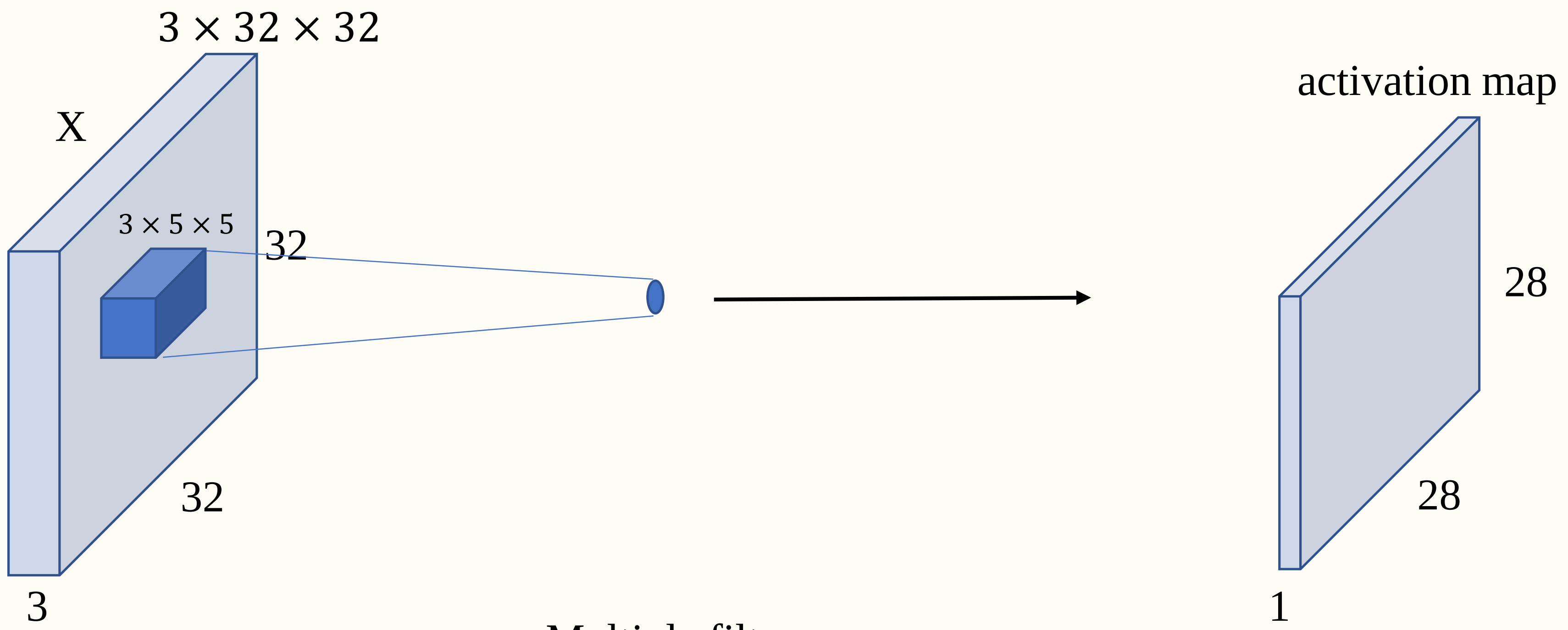


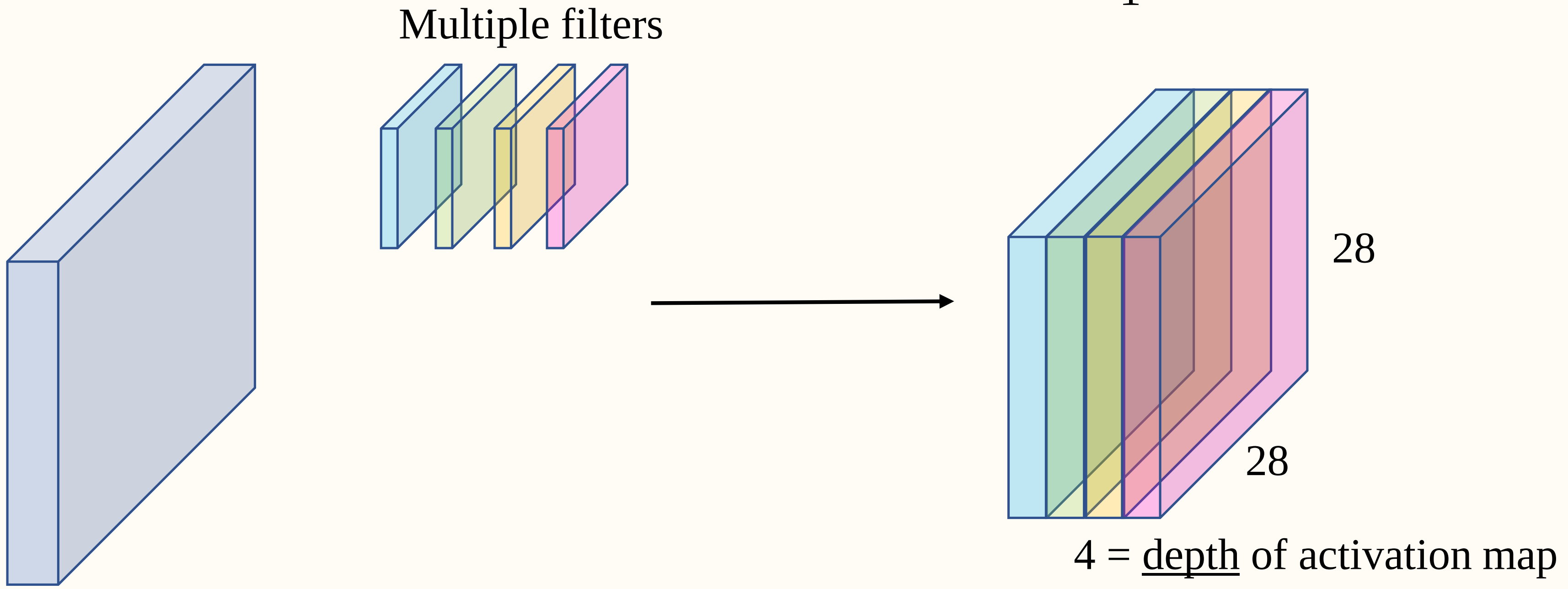
<u>Convolve</u> the filter with the image

(=Slide the filter spatially over the image and compute dot products)

1 number: take a $3 \times 5 \times 5$ chunk of the image and take the inner product with w and add bias b

= w.reshape(-1).T@ X [:,i:i+5, j:j+5].reshape(-1)+b

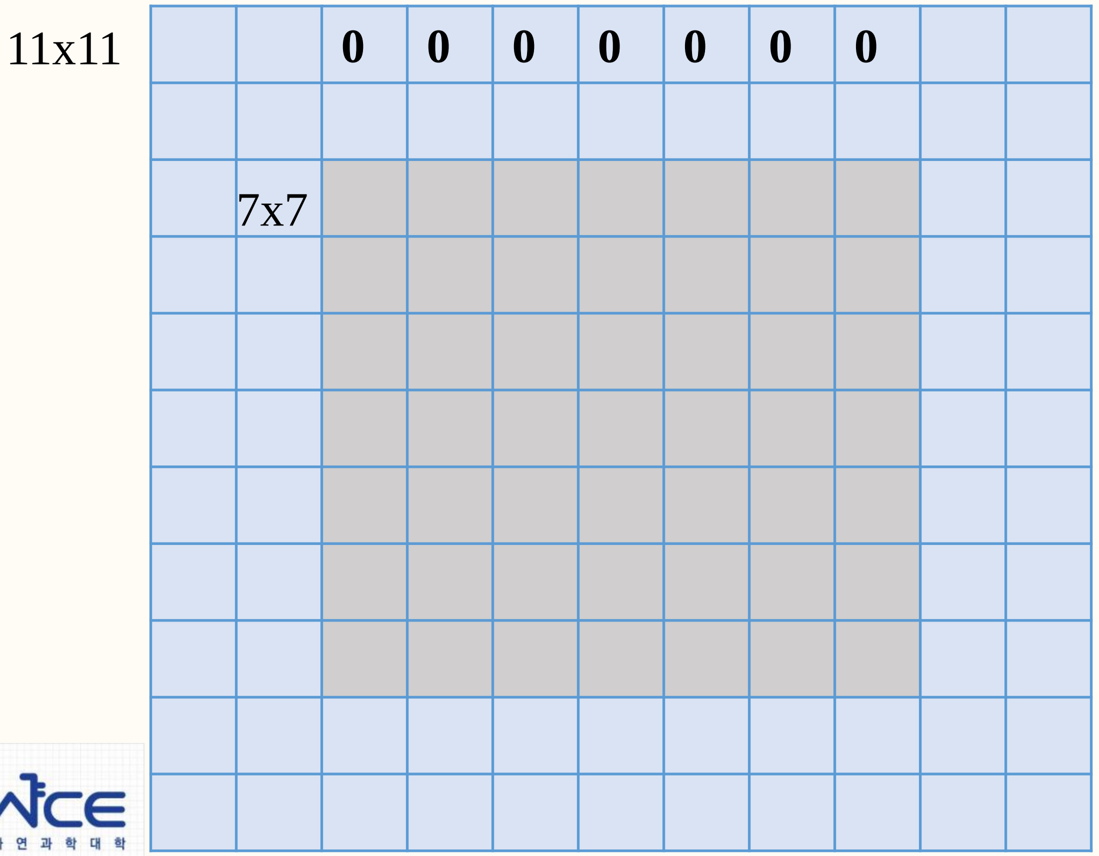

$3 \times 32 \times 32$
X
$3 \times 5 \times 5$
32
32
3
activation map
28
28
1


Multiple filters
28
28
4 = depth of activation map

Convolution options : Zero Padding

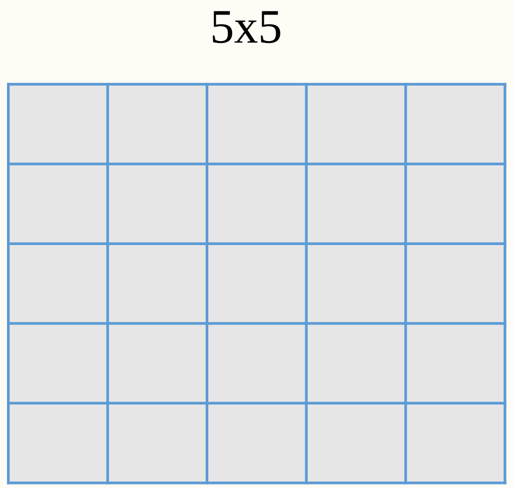
$3 \times 32 \times 32$ convolved with $3 \times 5 \times 5$ filter $\Rightarrow 1 \times 28 \times 28$ activation map

Spatial dimension 32 reduced to 28



Padding=2

convolve

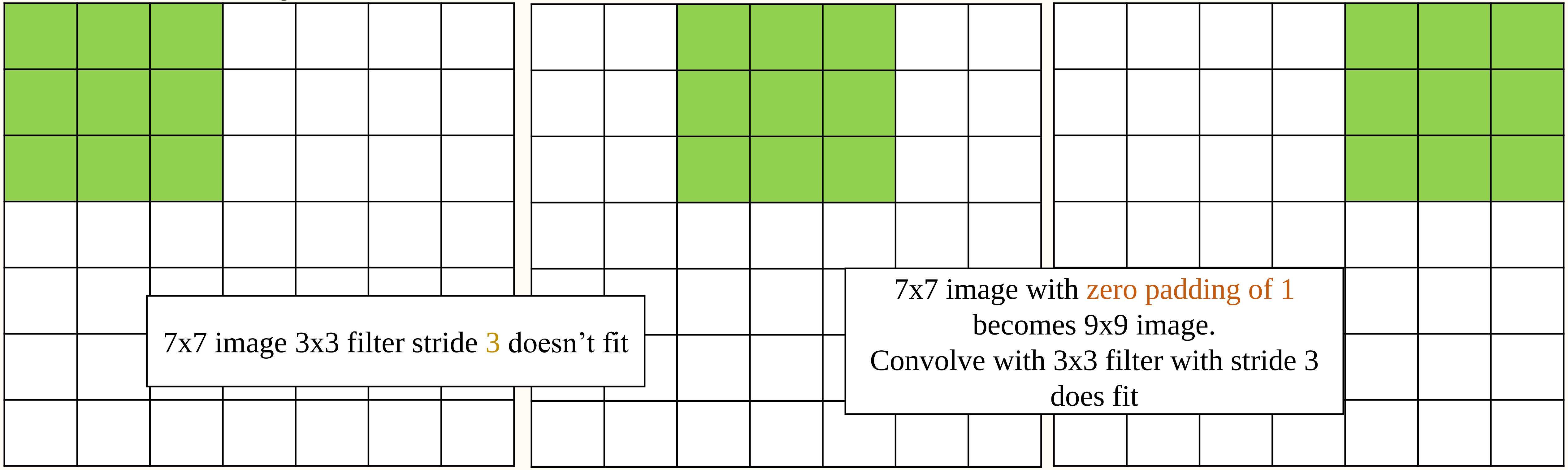


7x7 with Zero padding
(spatial dimension preserved)

3x3 without Zero padding

Convolution options : Stride

7x7 image convolved with 3x3 filter with stride 2



7x7 image 3x3 filter stride 3 doesn't fit

7x7 image with zero padding of 1 becomes 9x9 image.
Convolve with 3x3 filter with stride 3 does fit

Output 3x3
(with stride 1, output is 5x5)

Summary

Input $D_1 \times W_1 \times H_1$

Conv layer parameters

╠ K filters
╠ F spatial extent ($F \times F \times D_1$ filters)
╠ S stride
╚ P padding

Output $D_1 \times W_2 \times H_2$

The number of parameters

$$\underbrace{F^2 D_1 K}_{\text{filters}} + \underbrace{K}_{\text{biases}}$$

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

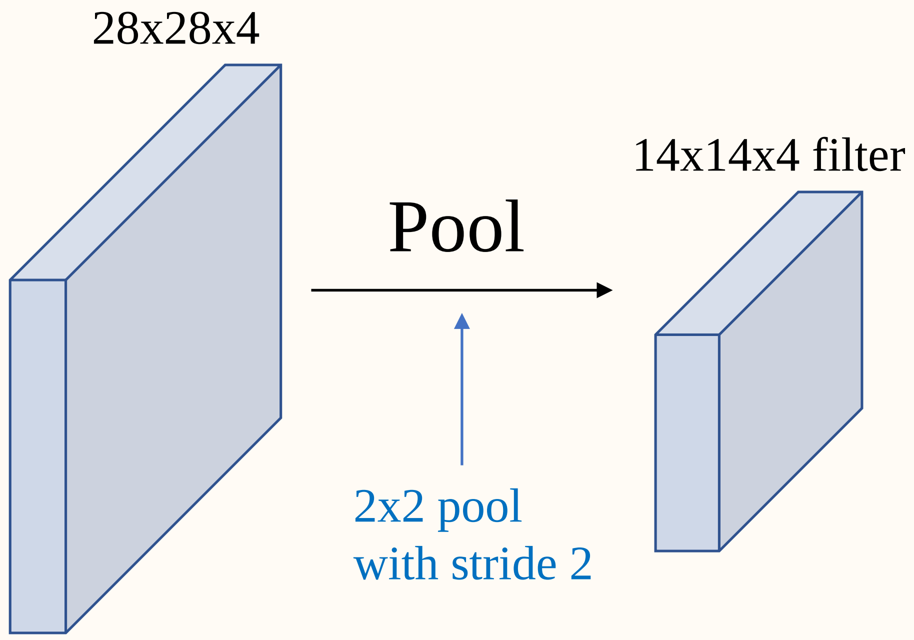
Should be integers

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

$$D_2 = K$$

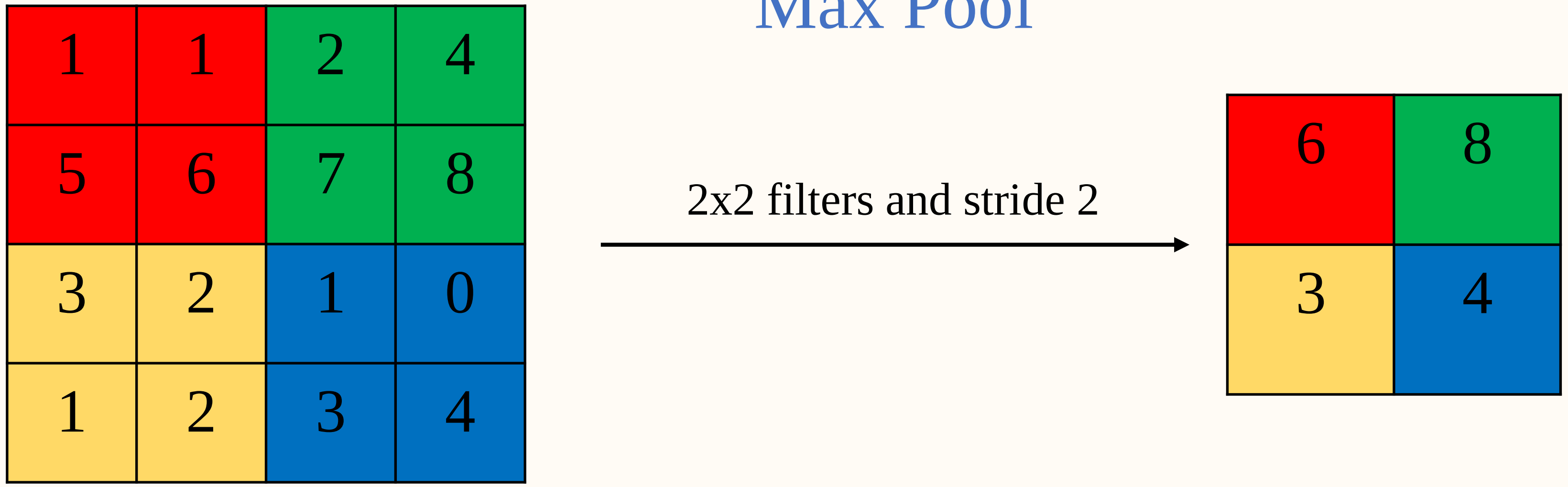
Pooling

- Similar to convolutions
- Used to reduce the size of the output
- Operates over each activation map independently



Single depth size

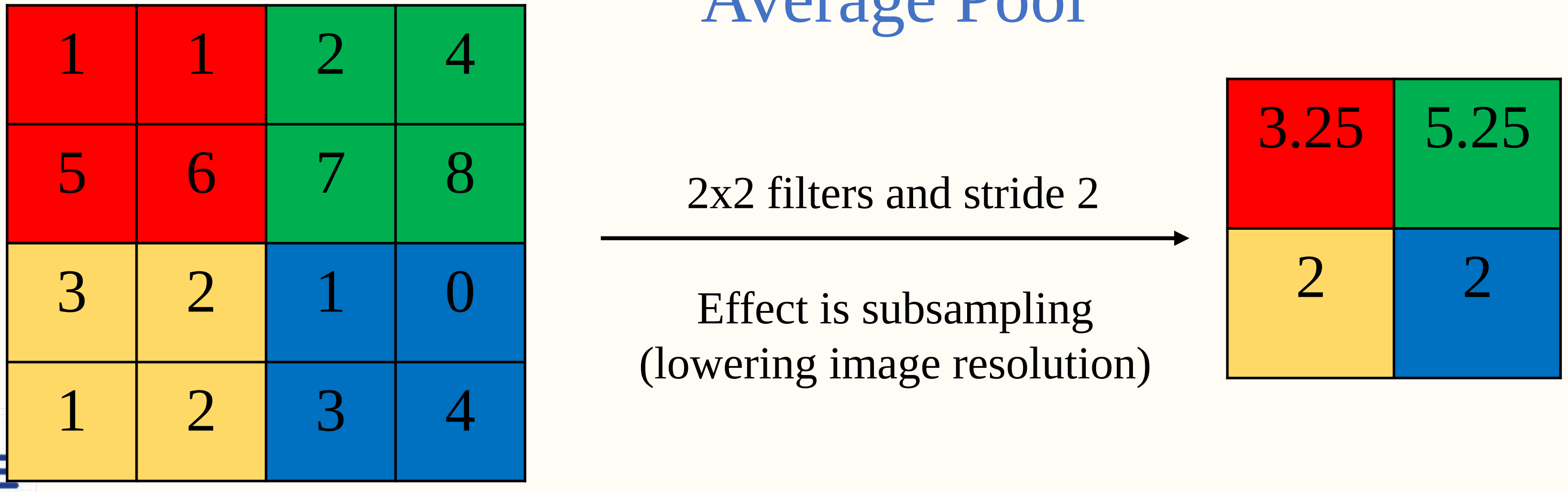
Max Pool



Not an instance of convolution

Precise definitions in
torch.nn.MaxPool2D
torch.nn.AvgPool2D

Average Pool



Effect is subsampling
(lowering image resolution)

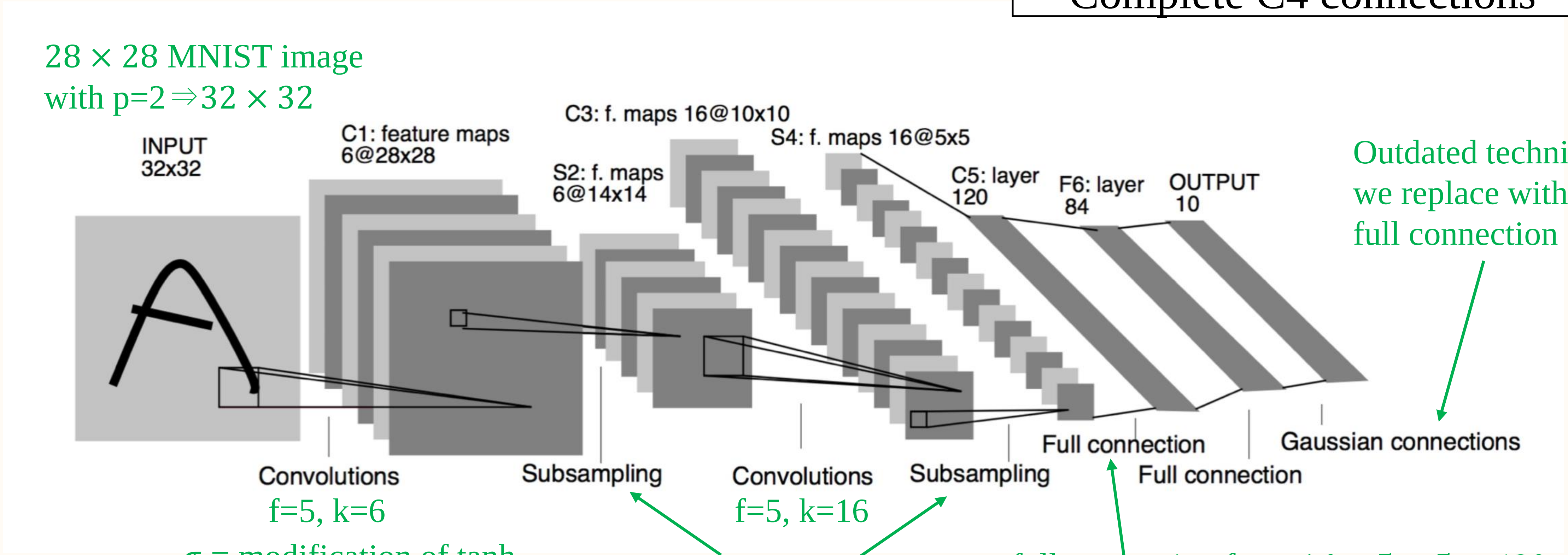
Instance of convolution with fixed (untrainable) weights, including the independent operation over each activation map.
(Why?)

LeNet5

(LeCun, Bottou, Bengio, Haffner 1998)

Modern instances of LeNet5 use

- σ =ReLu
- MaxPool instead of avg. pool
- No σ after S2, S4 (Why?)
- No Gaussian connections
- Complete C4 connections



28 × 28 MNIST image with p=2 ⇒ 32 × 32

Outdated technique we replace with full connection

f=5, k=6

f=5, k=16

σ = modification of tanh
Applied after
C1, S2, C3, S4, C5, F6

Something like average pool
f=2, S=2

full connection from 16 × 5 × 5 to 120
⇔conv. with f=5, k=120