CSCI567 Machine Learning (Fall 2025)

Haipeng Luo

University of Southern California

Sep 26, 2025

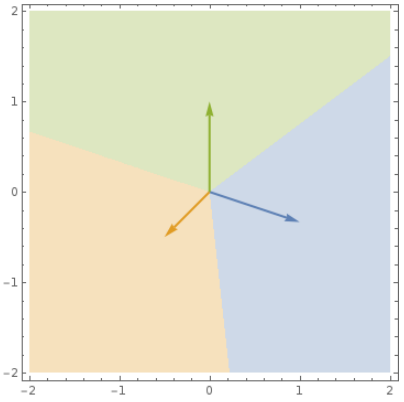
Administration

HW2 available now. Due on Oct 8th.

Outline

Outline

Linear models: from binary to multiclass



$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$
$\boldsymbol{w}_3 = (0, 1)$

- Blue class: $\{\boldsymbol{x} : 1 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$
- Orange class: $\{\boldsymbol{x} : 2 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$
- Green class: $\{\boldsymbol{x} : 3 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\}$$

Softmax + MLE = minimizing cross-entropy loss

Maximize probability of see labels $y_1, \ldots, y_{\mathsf{N}}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathsf{N}}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \operatorname{softmax}(\boldsymbol{W}\boldsymbol{x}_n)_{y_n} = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln \left(\frac{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}} \right) = \sum_{n=1}^{\mathsf{N}} \ln \left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n} \right)$$

This is the *multiclass logistic loss*, a.k.a *cross-entropy loss*.

Kernel functions

Definition: a function $k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \to \mathbb{R}$ is called a *kernel function* if there exists a function $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{M}}$ so that for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^{\mathsf{D}}$,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}')$$

Examples we have seen

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathrm{T}} \boldsymbol{x}')^2$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d}$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathrm{T}} \boldsymbol{x}' + c)^d \qquad \textbf{(polynomial kernel)}$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = e^{-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\sigma^2}} \qquad \textbf{(Gaussian/RBF kernel)}$$

Kernelizing ML algorithms

Key idea: rewrite the algorithm so that its dependence on the transformed dataset $\boldsymbol{\Phi}$ is only through the Gram matrix $\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$.

Perceptron (primal form with $\boldsymbol{\phi}$) *issue: time/space linear in* M

Initialize $\boldsymbol{w} = \boldsymbol{0} \in \mathbb{R}^{\mathsf{M}}$
Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{\phi}(\boldsymbol{x}_n)$

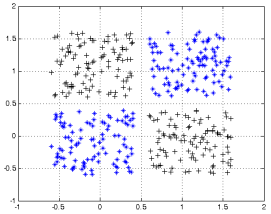
Kernelized Perceptron *independent of* M

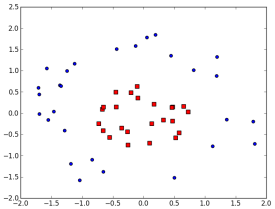
Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m k(\boldsymbol{x}_m, \boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Outline

Linear models are not always adequate





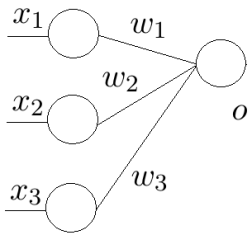
We can use a nonlinear mapping as discussed:

$$\boldsymbol{\phi}(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^{\mathrm{D}} \rightarrow \boldsymbol{z} \in \mathbb{R}^{\mathrm{M}}$$

Instead of manually picking $\boldsymbol{\phi}$ or kernel k, *can we learn this nonlinear mapping from data?*

The most popular nonlinear models nowadays: **neural nets**

Linear model as a one-layer neural net

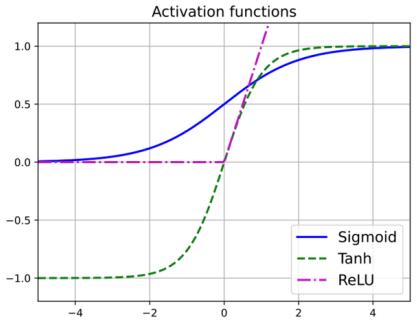


$o = h(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$

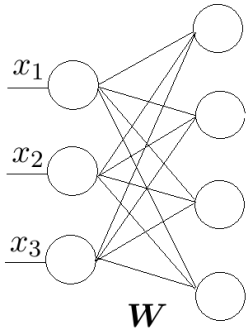
$h(a) = a$ for linear model

To create non-linearity, can use

- Rectified Linear Unit (**ReLU**): $h(a) = \max\{0, a\}$
- sigmoid function: $h(a) = \frac{1}{1+e^{-a}}$
- TanH: $h(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- many more



More output nodes

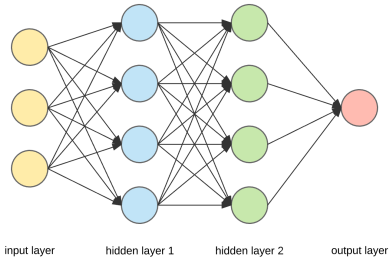


$\boldsymbol{o} = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$

$\boldsymbol{W} \in \mathbb{R}^{4\times 3}$, $\boldsymbol{h} : \mathbb{R}^4 \rightarrow \mathbb{R}^4$ so $\boldsymbol{h}(\boldsymbol{a}) = (h_1(a_1), h_2(a_2), h_3(a_3), h_4(a_4))$

Can think of this as a nonlinear mapping: $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$

More layers



Becomes a network:

- each node is called a **neuron**
- $\boldsymbol{h}$ is called the **activation function**
  - can use $h(a) = 1$ for one neuron in each layer to *incorporate bias term*
  - output neuron can use $h(a) = a$
- #layers refers to #hidden_layers (plus 1 or 2 for input/output layers)
- **deep** neural nets can have many layers and *millions* of parameters
- this is a **feedforward, fully connected** neural net, there are many variants (convolutional nets, recurrent nets, transformers, etc.)

How powerful are neural nets?

Universal approximation theorem (Cybenko, 89; Hornik, 91):

A feedforward neural net with a single hidden layer can approximate any continuous functions.

It might need a huge number of neurons though, and *depth helps!*

Designing network architecture is important and very complicated

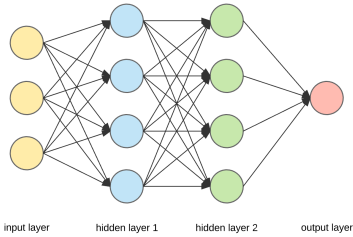
- for feedforward network, need to decide number of hidden layers, number of neurons at each layer, activation functions, etc.

Demo: `http://playground.tensorflow.org`

Math formulation

An L-layer neural net can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_{\mathsf{L}}\left(\boldsymbol{W}_L \boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}_{L-1} \cdots \boldsymbol{h}_1\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right)$$



To ease notation, for a given input $\boldsymbol{x}$, define recursively

$$\boldsymbol{o}_0 = \boldsymbol{x}, \qquad \boldsymbol{a}_\ell = \boldsymbol{W}_\ell \boldsymbol{o}_{\ell-1}, \qquad \boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell) \qquad (\ell = 1, \ldots, \mathsf{L})$$

where

- $\boldsymbol{W}_\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights between layer $\ell - 1$ and ℓ
- $\mathsf{D}_0 = \mathsf{D}, \mathsf{D}_1, \ldots, \mathsf{D}_{\mathsf{L}}$ are numbers of neurons at each layer
- $\boldsymbol{a}_\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}_\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output of layer ℓ
- $\boldsymbol{h}_\ell : \mathbb{R}^{\mathsf{D}_\ell} \to \mathbb{R}^{\mathsf{D}_\ell}$ is activation functions at layer ℓ

Learning the model

No matter how complicated the model is, our goal is the same: minimize

$$F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \frac{1}{N} \sum_{n=1}^{\mathsf{N}} F_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}})$$

where

$$F_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \begin{cases} \|\boldsymbol{f}(\boldsymbol{x}_n) - \boldsymbol{y}_n\|_2^2 & \text{for regression} \\ \ln\left(1 + \sum_{k \neq y_n} e^{f(\boldsymbol{x}_n)_k - f(\boldsymbol{x}_n)_{y_n}}\right) & \text{for classification} \end{cases}$$

How to optimize such a complicated function?

Same thing: apply **SGD**! even if the model is *nonconvex*.

What is the gradient of this complicated function?

Chain rule is the only secret:

- for a composite function $f(g(w))$

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial w}$$

- for a composite function $f(g_1(w), \ldots, g_d(w))$

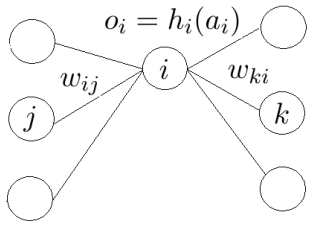
$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial w}$$

the simplest example $f(g_1(w), g_2(w)) = g_1(w)g_2(w)$

Computing the derivative

Drop the subscript ℓ for layer for simplicity.

Find the **derivative of F_n w.r.t. to w_{ij}**



$$\frac{\partial F_n}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial (w_{ij}o_j)}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}o_j$$

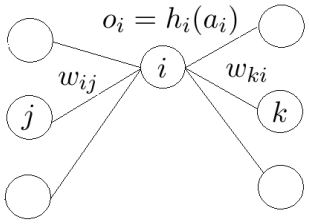
$$\frac{\partial F_n}{\partial a_i} = \frac{\partial F_n}{\partial o_i}\frac{\partial o_i}{\partial a_i} = \left(\sum_k \frac{\partial F_n}{\partial a_k}\frac{\partial a_k}{\partial o_i}\right) h_i'(a_i) = \left(\sum_k \frac{\partial F_n}{\partial a_k} w_{ki}\right) h_i'(a_i)$$

Computing the derivative

Adding the subscript for layer:

$$\frac{\partial F_n}{\partial w_{\ell,ij}} = \frac{\partial F_n}{\partial a_{\ell,i}} o_{\ell-1,j}$$

$$\frac{\partial F_n}{\partial a_{\ell,i}} = \left(\sum_k \frac{\partial F_n}{\partial a_{\ell+1,k}} w_{\ell+1,ki}\right) h'_{\ell,i}(a_{\ell,i})$$



For the last layer, for square loss

$$\frac{\partial F_n}{\partial a_{\mathsf{L},i}} = \frac{\partial (h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i})^2}{\partial a_{\mathsf{L},i}} = 2(h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i}) h'_{\mathsf{L},i}(a_{\mathsf{L},i})$$

Exercise: try to do it for cross-entropy loss yourself.

Computing the derivative

Using **matrix notation** greatly simplifies presentation and implementation:

$$\frac{\partial F_n}{\partial \boldsymbol{W}_\ell} = \frac{\partial F_n}{\partial \boldsymbol{a}_\ell} \boldsymbol{o}_{\ell-1}^{\mathrm{T}} \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$$

$$\frac{\partial F_n}{\partial \boldsymbol{a}_\ell} = \begin{cases} \left(\boldsymbol{W}_{\ell+1}^{\mathrm{T}} \frac{\partial F_n}{\partial \boldsymbol{a}_{\ell+1}}\right) \circ \boldsymbol{h}'_\ell(\boldsymbol{a}_\ell) & \text{if } \ell < \mathsf{L} \\ 2(\boldsymbol{h}_\mathsf{L}(\boldsymbol{a}_\mathsf{L}) - \boldsymbol{y}_n) \circ \boldsymbol{h}'_\mathsf{L}(\boldsymbol{a}_\mathsf{L}) & \text{else} \end{cases}$$

where $\boldsymbol{v}_1 \circ \boldsymbol{v}_2 = (v_{11}v_{21}, \cdots, v_{1\mathsf{D}}v_{2\mathsf{D}})$ is the element-wise product (a.k.a. Hadamard product).

Verify yourself! (But no need to remember this formula.)

Putting everything into SGD

The **backpropagation** algorithm (**Backprop**)

Initialize $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}$ randomly. Repeat:

1. randomly pick one data point $n \in [\mathsf{N}]$
2. **forward propagation**: for each layer $\ell = 1, \ldots, \mathsf{L}$
   - compute $\boldsymbol{a}_\ell = \boldsymbol{W}_\ell \boldsymbol{o}_{\ell-1}$ and $\boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell)$ $\qquad (\boldsymbol{o}_0 = \boldsymbol{x}_n)$
3. **backward propagation**: for each $\ell = L, \ldots, 1$
   - compute

$$\frac{\partial F_n}{\partial \boldsymbol{a}_\ell} = \begin{cases} \left(\boldsymbol{W}_{\ell+1}^{\mathrm{T}} \frac{\partial F_n}{\partial \boldsymbol{a}_{\ell+1}}\right) \circ \boldsymbol{h}_\ell'(\boldsymbol{a}_\ell) & \text{if } \ell < \mathsf{L} \\ 2(\boldsymbol{h}_{\mathsf{L}}(\boldsymbol{a}_{\mathsf{L}}) - \boldsymbol{y}_n) \circ \boldsymbol{h}_{\mathsf{L}}'(\boldsymbol{a}_{\mathsf{L}}) & \text{else} \end{cases}$$

   - update weights

$$\boldsymbol{W}_\ell \leftarrow \boldsymbol{W}_\ell - \eta \frac{\partial F_n}{\partial \boldsymbol{W}_\ell} = \boldsymbol{W}_\ell - \eta \frac{\partial F_n}{\partial \boldsymbol{a}_\ell} \boldsymbol{o}_{\ell-1}^{\mathrm{T}}$$

(Important: *should $\boldsymbol{W}_\ell$ be overwritten immediately in the last step?*)

Important tricks to optimize neural nets

Many important tricks on top on Backprop

- **mini-batch**: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)
- **batch normalization**: normalize the inputs of each neuron over the mini-batch (to zero-mean and one-variance; *c.f.* Lec 1)
- **adaptive learning rate**: scale the learning rate of each parameter based on some moving average of the magnitude of the gradients
- **momentum**: make use of previous gradients (taking inspiration from physics)
- ...

SGD with momentum (a simple version)

Initialize $\boldsymbol{w}_0$ and **velocity** $\boldsymbol{v} = \boldsymbol{0}$

For $t = 1, 2, \ldots$

- form a stochastic gradient $\boldsymbol{g}_t$
- update velocity $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + \boldsymbol{g}_t$ for some discount factor $\alpha \in (0, 1)$
- update weight $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \boldsymbol{v}$

Updates for first few rounds:

- $\boldsymbol{w}_1 = \boldsymbol{w}_0 - \eta \boldsymbol{g}_1$
- $\boldsymbol{w}_2 = \boldsymbol{w}_1 - \alpha\eta \boldsymbol{g}_1 - \eta \boldsymbol{g}_2$
- $\boldsymbol{w}_3 = \boldsymbol{w}_2 - \alpha^2\eta \boldsymbol{g}_1 - \alpha\eta \boldsymbol{g}_2 - \eta \boldsymbol{g}_3$
- ...

Adam (most popular) $\approx$ SGD + adaptive learning rate + momentum

Overfitting

Overfitting is very likely since neural nets are too powerful.

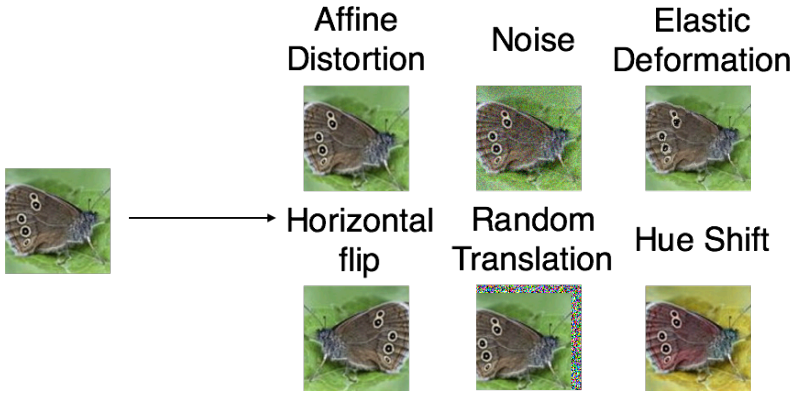
Methods to overcome overfitting:

- data augmentation
- regularization
- dropout
- early stopping
- ...

Data augmentation

Data: the more the better. How do we get more data?

Exploit prior knowledge to add more training data



Regularization

L2 regularization: minimize

$$F'(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) + \lambda \sum_{\ell=1}^{\mathsf{L}} \|\boldsymbol{W}_\ell\|_2^2$$

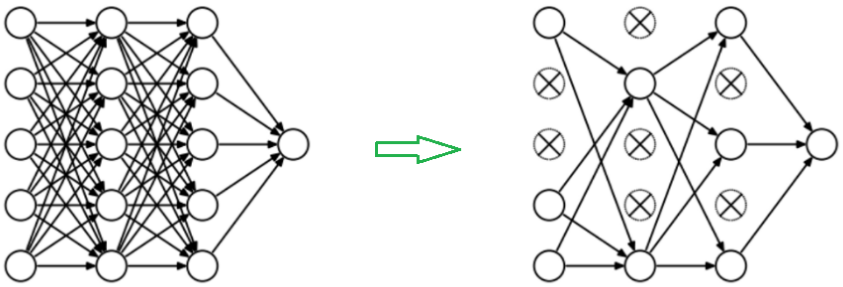
Simple change to the gradient:

$$\frac{\partial F'}{\partial w_{ij}} = \frac{\partial F}{\partial w_{ij}} + 2\lambda w_{ij}$$

Introduce *weight decaying effect*

Dropout

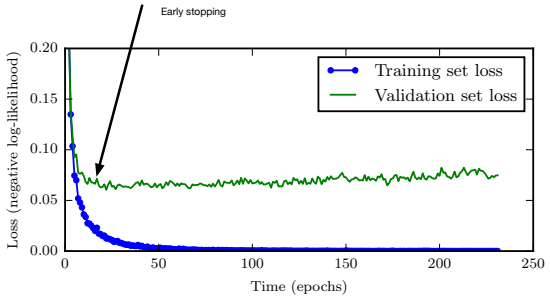
Independently delete each neuron with a fixed probability (say 0.5), during each iteration of Backprop (only for training, not for testing)



Very effective, makes training faster as well

Early stopping

Stop training when the performance on validation set stops improving



Conclusions for neural nets

Deep neural networks

- are hugely popular, achieving *best performance* on many problems
- do need *a lot of data* to work well
- take *a lot of time* to train (need GPUs for massive parallel computing)
- take some work to select architecture and hyperparameters
- are still not well understood in theory

Outline

Acknowledgements

Not much math, a lot of empirical intuitions

The materials borrow heavily from the following sources:

- Stanford Course CS231n: `http://cs231n.stanford.edu/`
- Dr. Ian Goodfellow's lectures on deep learning: `http://deeplearningbook.org`

Both website provides tons of useful resources: notes, demos, videos, etc.

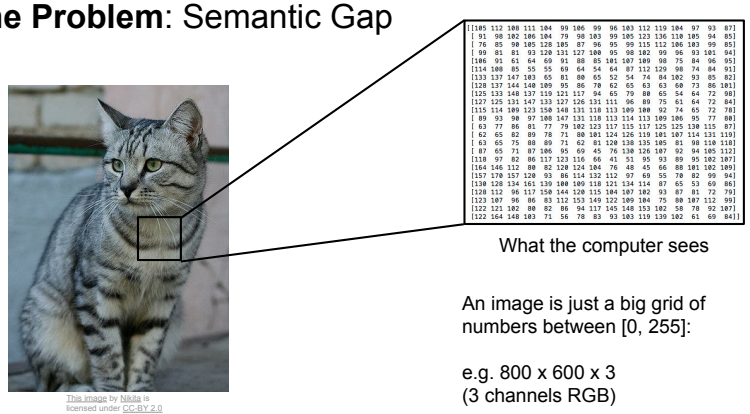
Also, demo from `https://poloclub.github.io/cnn-explainer/`

Image Classification: A core task in Computer Vision

(assume given set of discrete labels)
{dog, cat, truck, plane, ...}

→ cat

The Problem: Semantic Gap

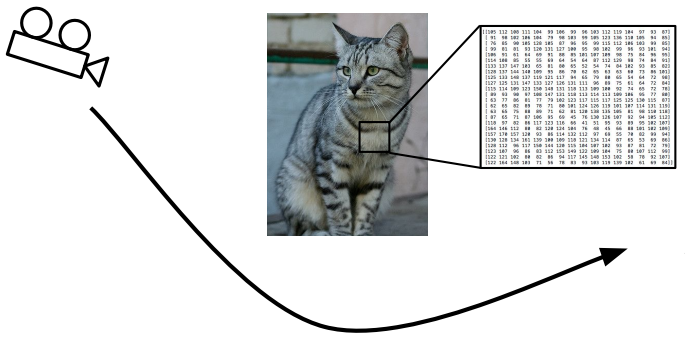


What the computer sees

An image is just a big grid of numbers between [0, 255]:

e.g. 800 x 600 x 3
(3 channels RGB)

Challenges: Viewpoint variation



All pixels change when the camera moves!

Challenges: Illumination

Challenges: Deformation

Challenges: Occlusion

This image is CC0 1.0 public domain | This image is CC0 1.0 public domain | This image by jonsson is licensed under CC-BY 2.0

Challenges: Background Clutter

This image is CC0 1.0 public domain | This image is CC0 1.0 public domain

Challenges: Intraclass variation

This image is CC0 1.0 public domain

Fundamental problems in vision

The key challenge

How to train a model that can tolerate all those variations?

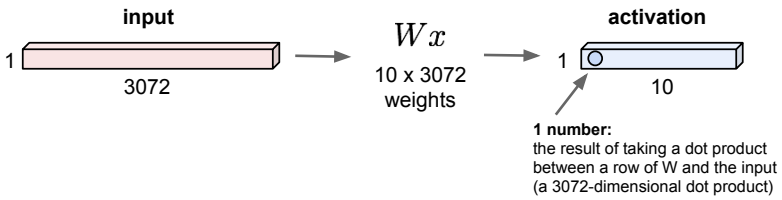
Main ideas

- need a lot of data that exhibits those variations
- need more specialized models to capture the invariance

Issues of standard NN for image inputs

Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1

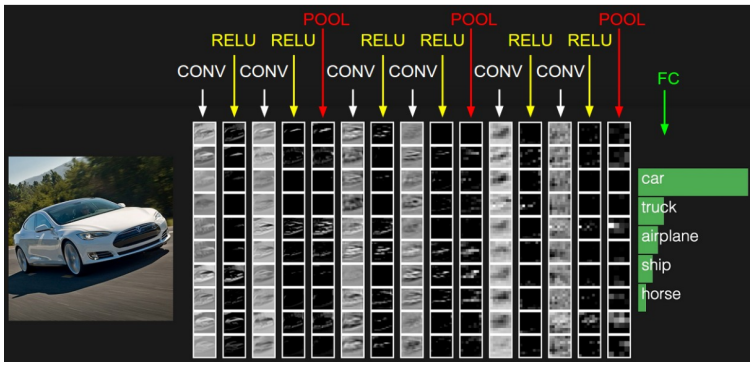


Spatial structure is lost!

Solution: Convolutional Neural Net (ConvNet/CNN)

A special case of fully connected neural nets

- usually consist of **convolution layers**, ReLU layers, **pooling layers**, and regular fully connected layers
- key idea: *learning from low-level to high-level features*

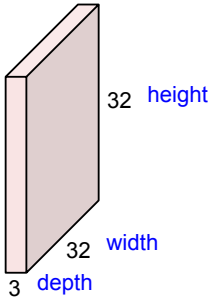


Convolution layer

Arrange neurons as a **3D volume** naturally

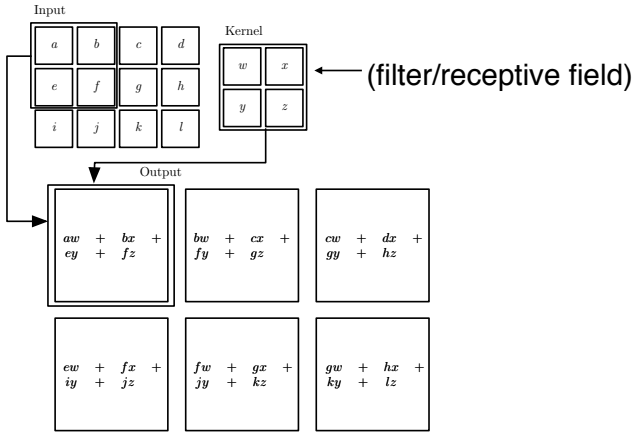
Convolution Layer

32x32x3 image -> preserve spatial structure



Convolution

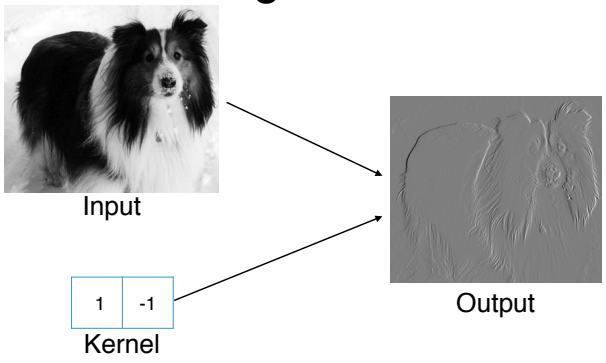
2D Convolution



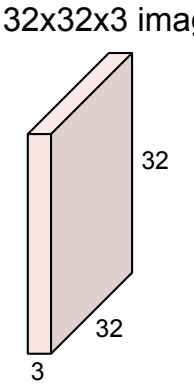
Why convolution makes sense?

Main idea: **if a filter is useful at one location, it should be useful at other locations.**

A simple example why filtering is useful



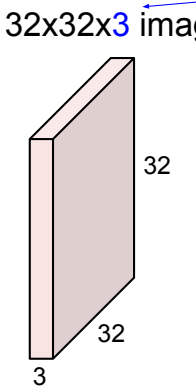
Convolution Layer

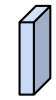


Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

Convolution Layer

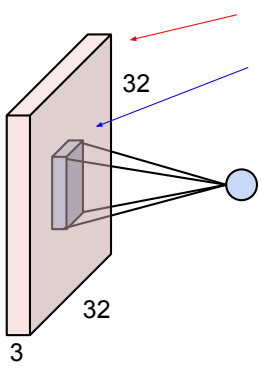
Filters always extend the full depth of the input volume





Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

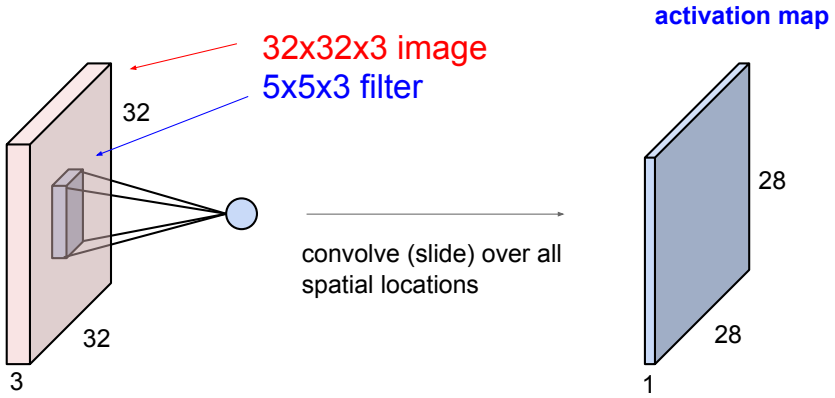
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

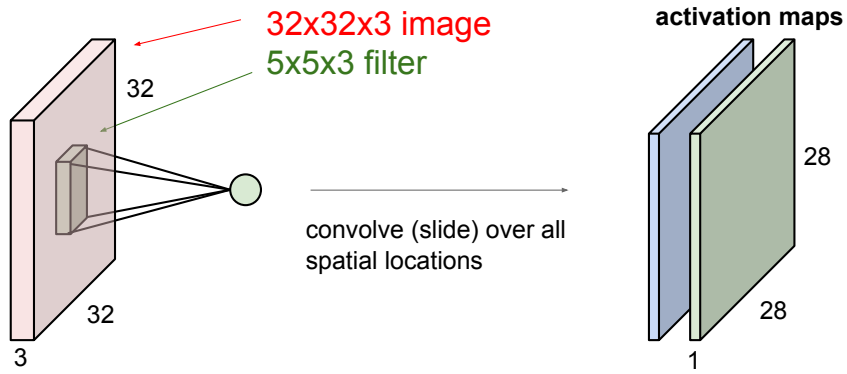
$$w^T x + b$$

Convolution Layer

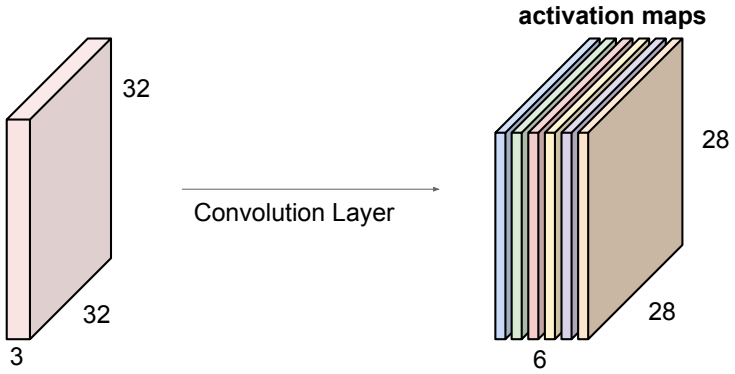


Convolution Layer

consider a second, green filter

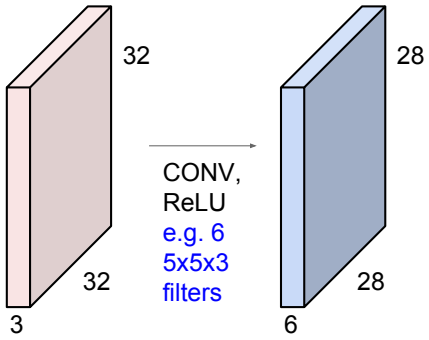


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

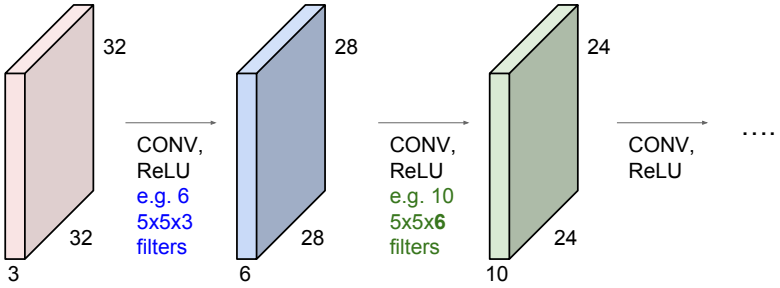


We stack these up to get a "new image" of size 28x28x6!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



Demo: `https://poloclub.github.io/cnn-explainer`

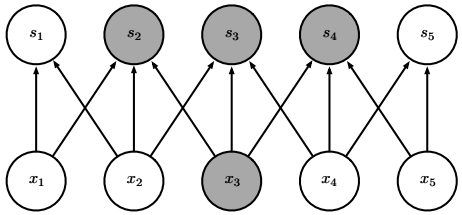
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

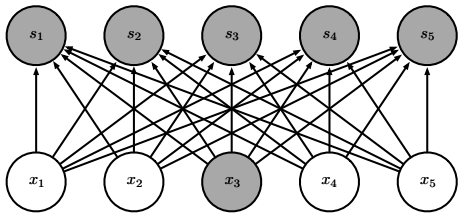
- filter = weights with **sparse connection**

Local Receptive Field Leads to Sparse Connectivity (affects less)

Sparse connections due to small convolution kernel



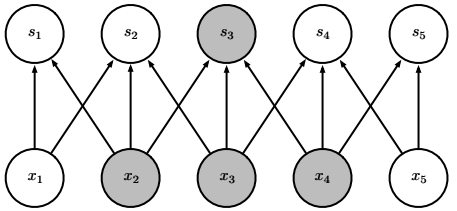
Dense connections



(Goodfellow 2016)

Sparse connectivity: being affected by less

Sparse connections due to small convolution kernel



Dense connections

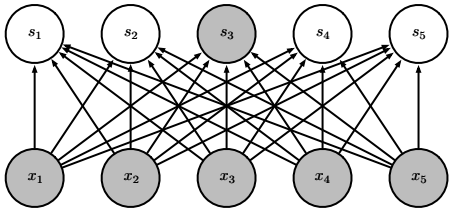


Figure 9.3

(Goodfellow 2016)

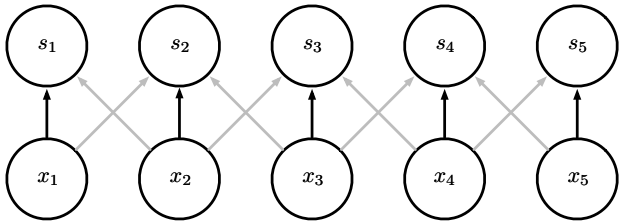
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**
- **parameters sharing**

Parameter Sharing

Convolution shares the same parameters across all spatial locations



Traditional matrix multiplication does not share any parameters

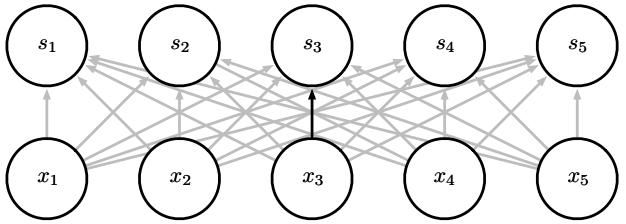


Figure 9.5

(Goodfellow 2016)

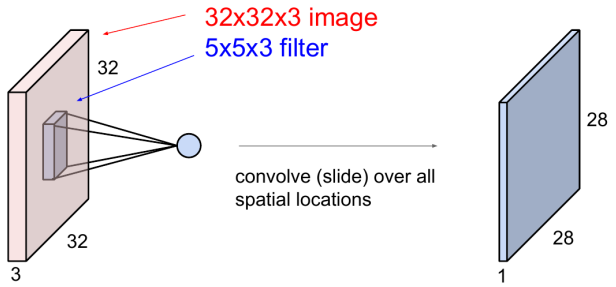
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**
- **parameters sharing**

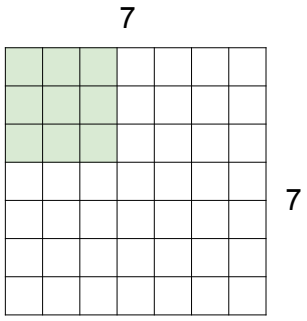
Much fewer parameters! Example (ignore bias terms):

- FC: $(32 \times 32 \times 3) \times (28 \times 28) \approx 2.4M$
- CNN: $5 \times 5 \times 3 = 75$



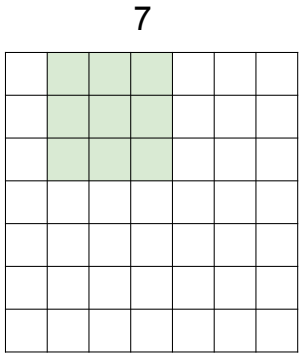
Spatial arrangement: stride and padding

A closer look at spatial dimensions:



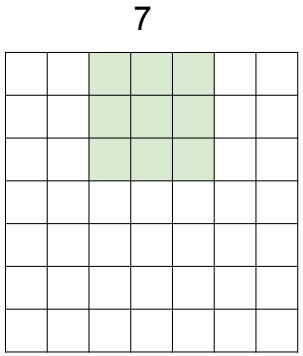
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



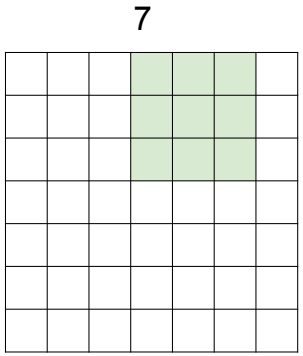
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



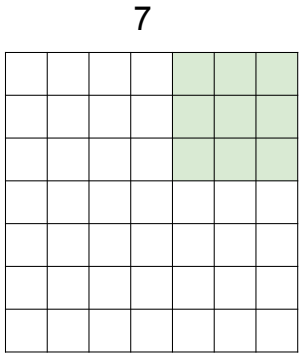
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

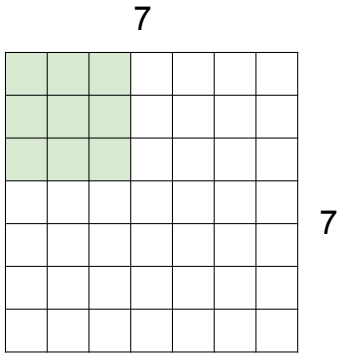
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

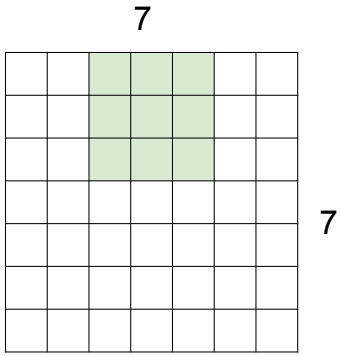
=> 5x5 output

A closer look at spatial dimensions:



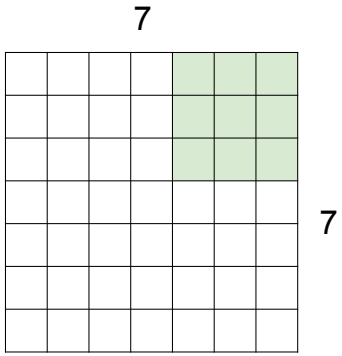
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



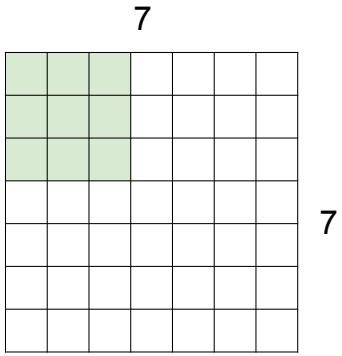
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



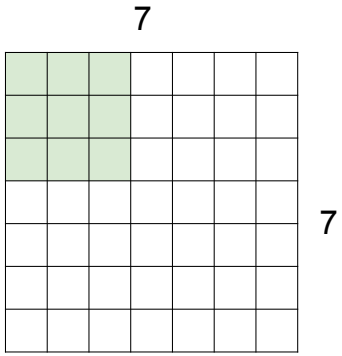
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



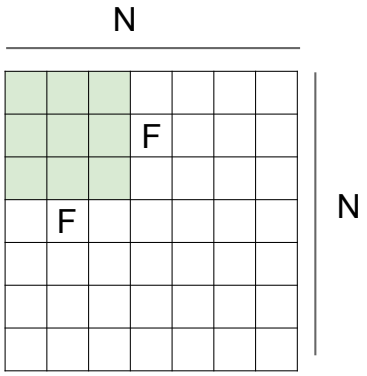
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

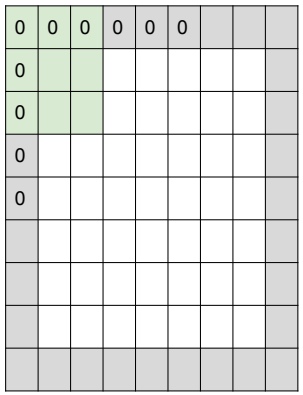
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

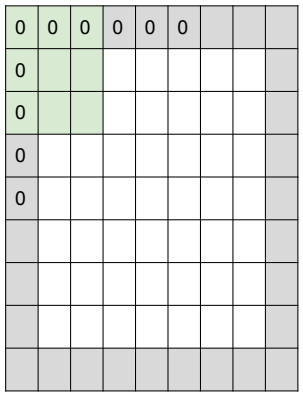
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

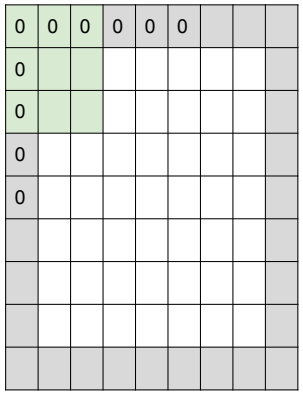
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

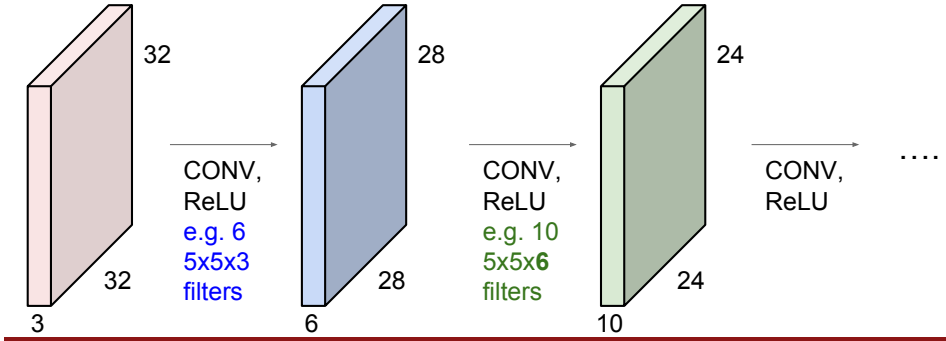
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 = (H_1 + 2P - F)/S + 1$
- $D_2 = K$

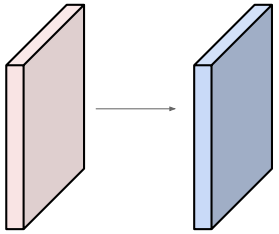
#parameters: $(F \times F \times D_1 + 1) \times K$ weights

Common setting: $F = 3, S = P = 1$

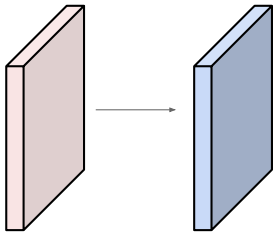
Examples time:

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?



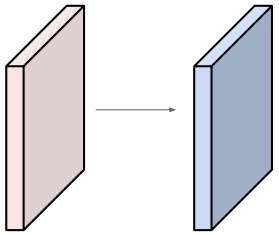
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

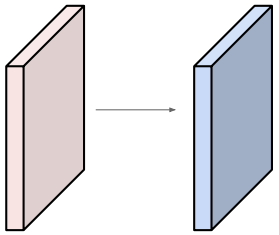
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Examples time:

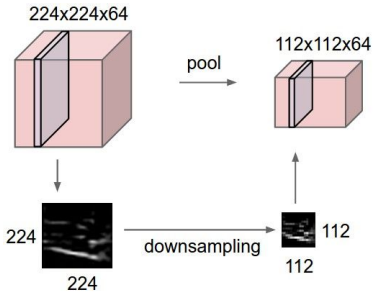


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Another element: pooling

Pooling layer
- makes the representations smaller and more manageable
- operates over each activation map independently:

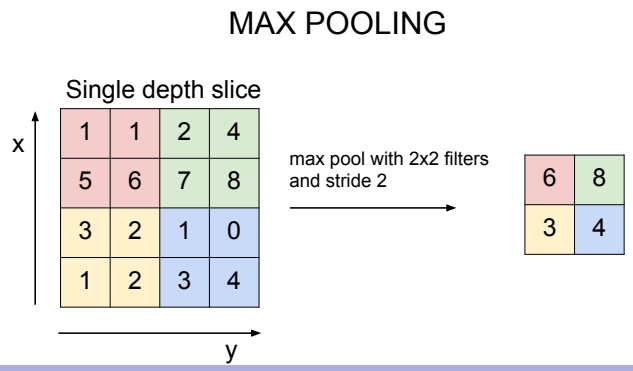


Pooling

Similar to a filter, except

- depth is always 1
- different operations: average, L2-norm, max
- no parameters to be learned

Max pooling with 2×2 filter and stride 2 is very common



Putting everything together

Typical architecture for CNNs:

Input → [[Conv → ReLU]*N → Pool?]*M → [FC → ReLU]*Q → FC

Common choices: $N \leq 5, Q \leq 2$, M is large

Well-known CNNs: LeNet, AlexNet, ZF Net, GoogLeNet, VGGNet, etc.

All achieve excellent performance on image classification tasks.

Demo: `https://poloclub.github.io/cnn-explainer`

How to train a CNN?

How do we learn the filters/weights?

Essentially the same as FC NNs: apply **SGD/backpropagation**