CSCI 567: Machine Learning

Vatsal Sharan

Spring 2024

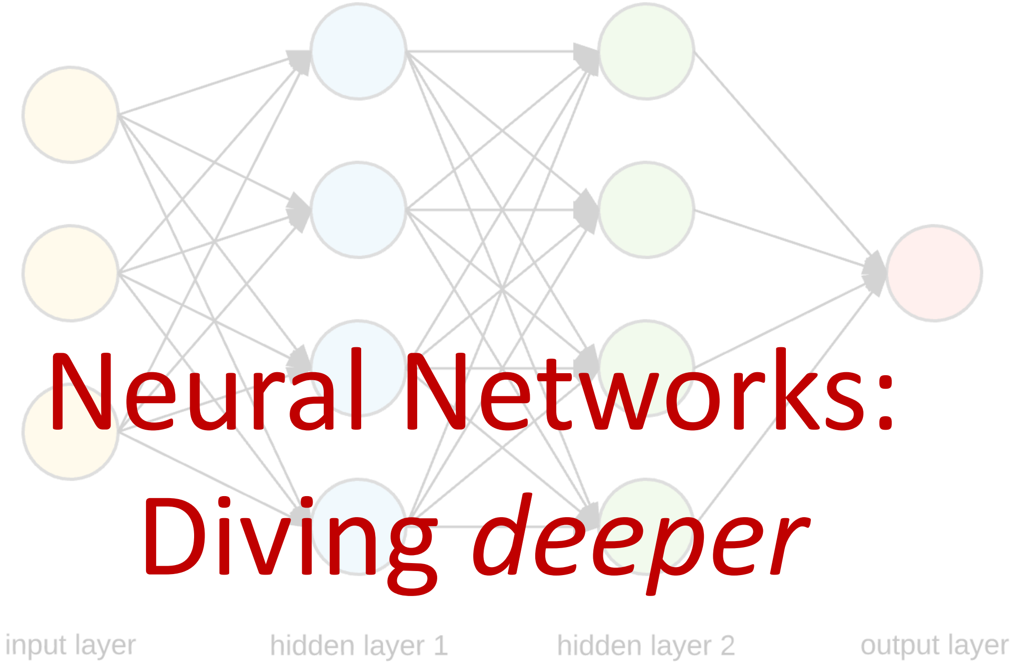
Lecture 7, Feb 23

USC University of Southern California

Administrivia

- Exam 1 is next week (March 1, 2 hr 20 min, approximately starting at 1pm)
- Students will be split into two rooms, instructions later (DEN students will get separate instructions)
- You can bring one cheat sheet (you can write on both sides), though we will generally provide necessary formulae
- No other books, resources etc.

Neural Networks: Diving *deeper*



3.1 Representation: Very powerful function class!

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

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

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

Choosing the network architecture is important.

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

Designing the architecture can be complicated, though various standard choices exist.

3.2 Optimization: Computing gradients efficiently using Backprop

Backpropogation: A very efficient way to compute gradients of neural networks using an application of the chain rule (similar to dynamic programming).

Chain rule:

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

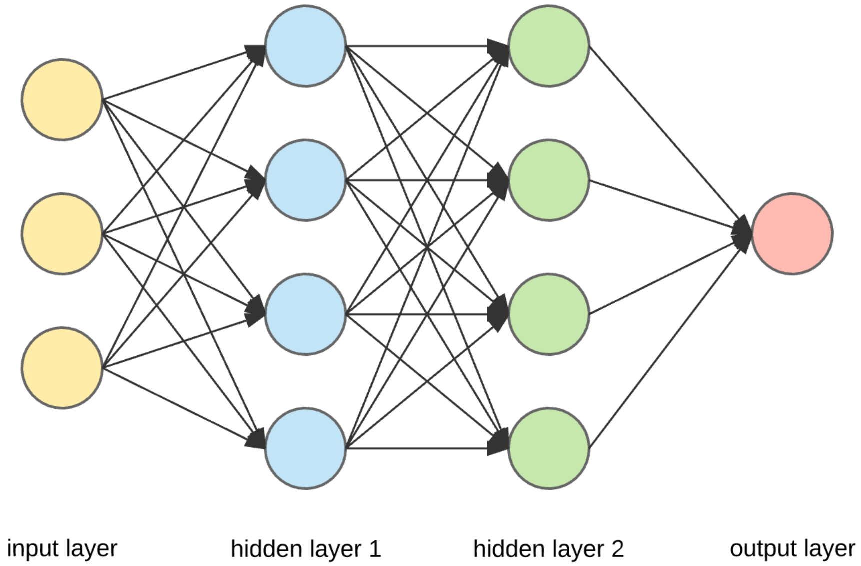
$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial w}$$

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

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

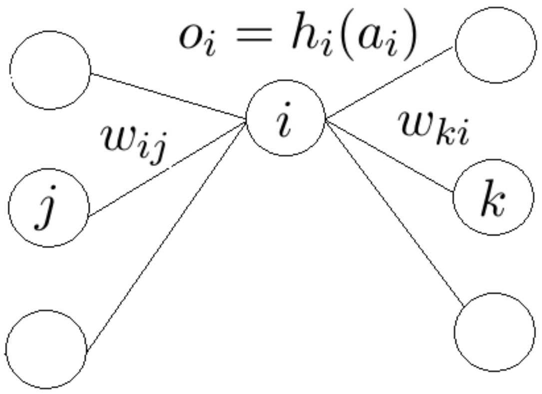
the simplest example $f(g_1(w), g_2(w)) = g_1(w)g_2(w)$

Backprop: Intuition



Backprop: Derivation

Drop the subscript ℓ for layer for simplicity. For this derivation, refer to the loss function as F_m (instead of F_i) for convenience.



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

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

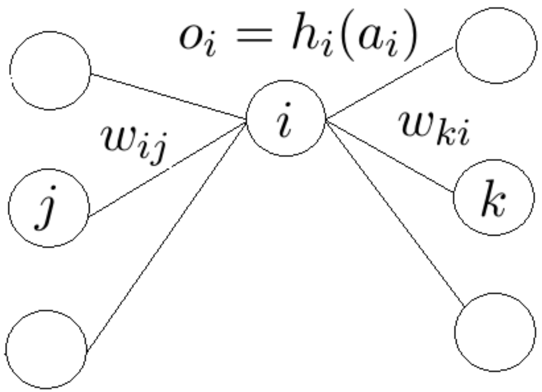
$$\frac{\partial F_m}{\partial a_i} = \frac{\partial F_m}{\partial o_i}\frac{\partial o_i}{\partial a_i} = \left(\sum_k \frac{\partial F_m}{\partial a_k}\frac{\partial a_k}{\partial o_i}\right) h_i'(a_i) = \left(\sum_k \frac{\partial F_m}{\partial a_k} w_{ki}\right) h_i'(a_i)$$

Backprop: Derivation

Adding the subscript for layer:

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

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

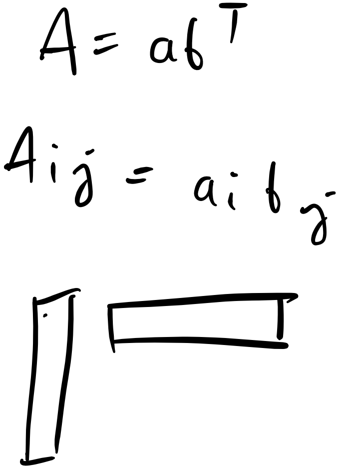


For the last layer, for square loss

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

Exercise: try to do it for logistic loss yourself.

Backprop: Derivation



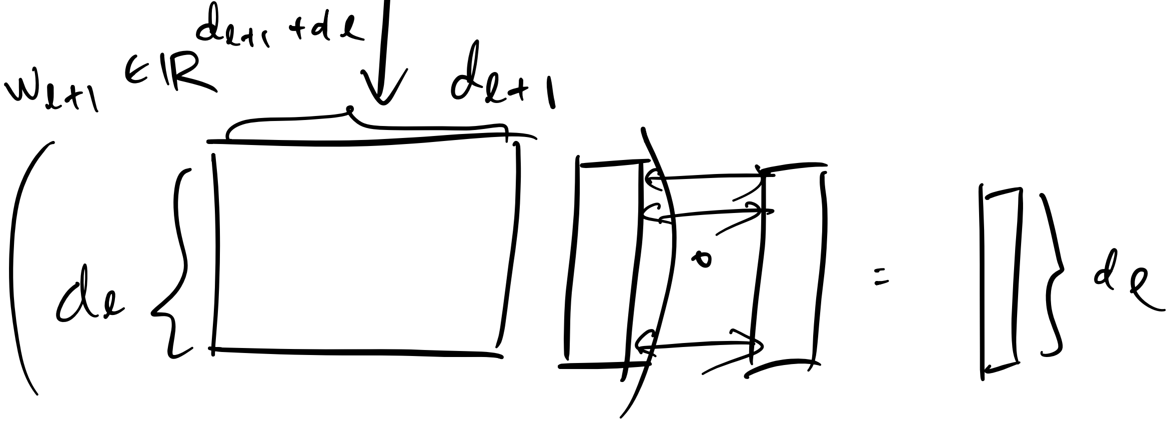
Using **matrix notation** greatly simplifies presentation and implementation:

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

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

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

Verify yourself!



The backpropagation algorithm (Backprop)

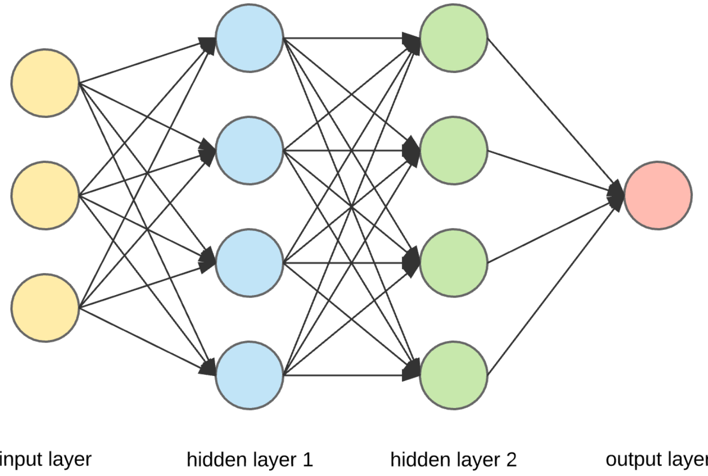
Initialize $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}$ randomly. Repeat:

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

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

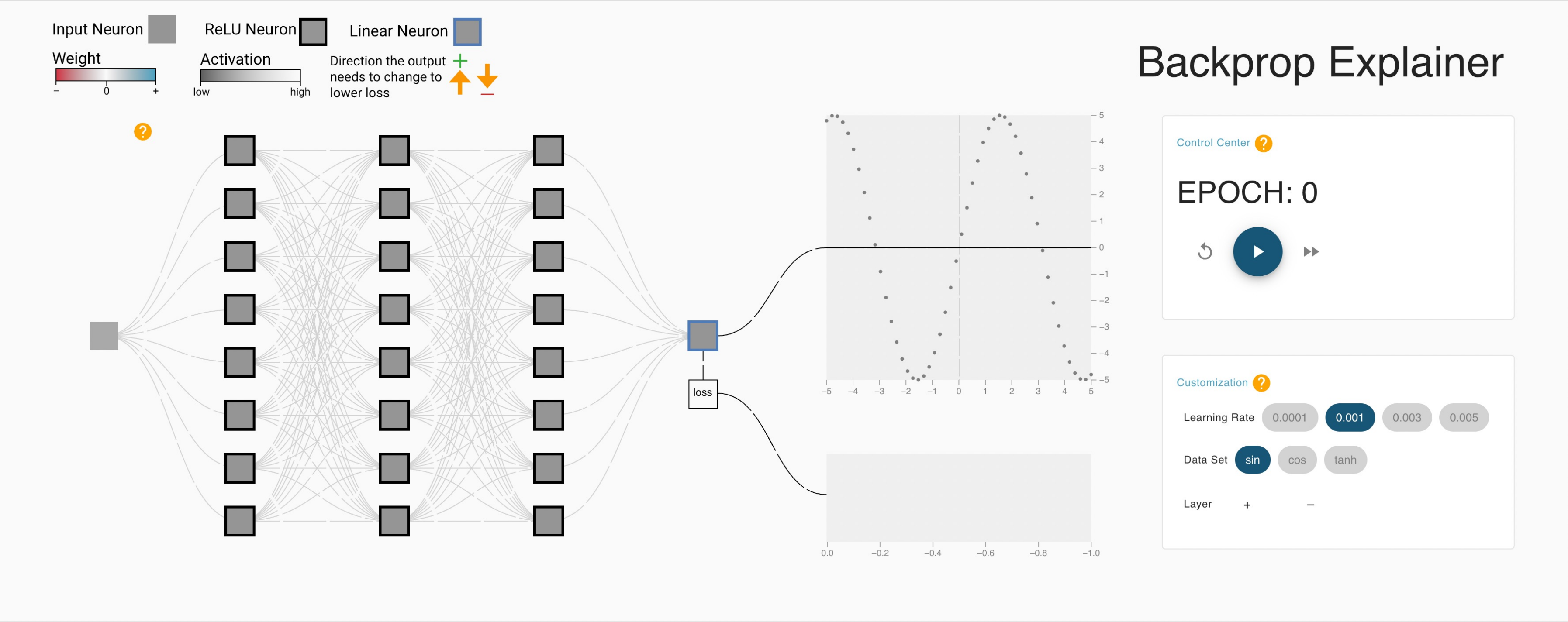
   - update weights

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



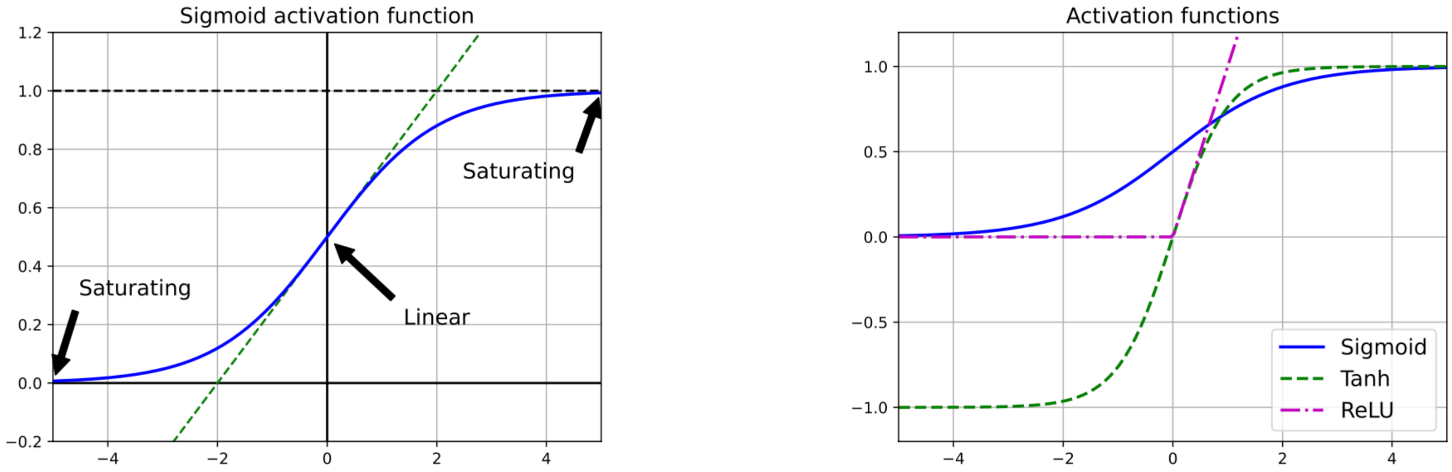
(Important: *should $\boldsymbol{W}_\ell$ be overwritten immediately in the last step?*)

Demo



https://xnought.github.io/backprop-explainer/

Non-saturating activation functions



(a) (b)

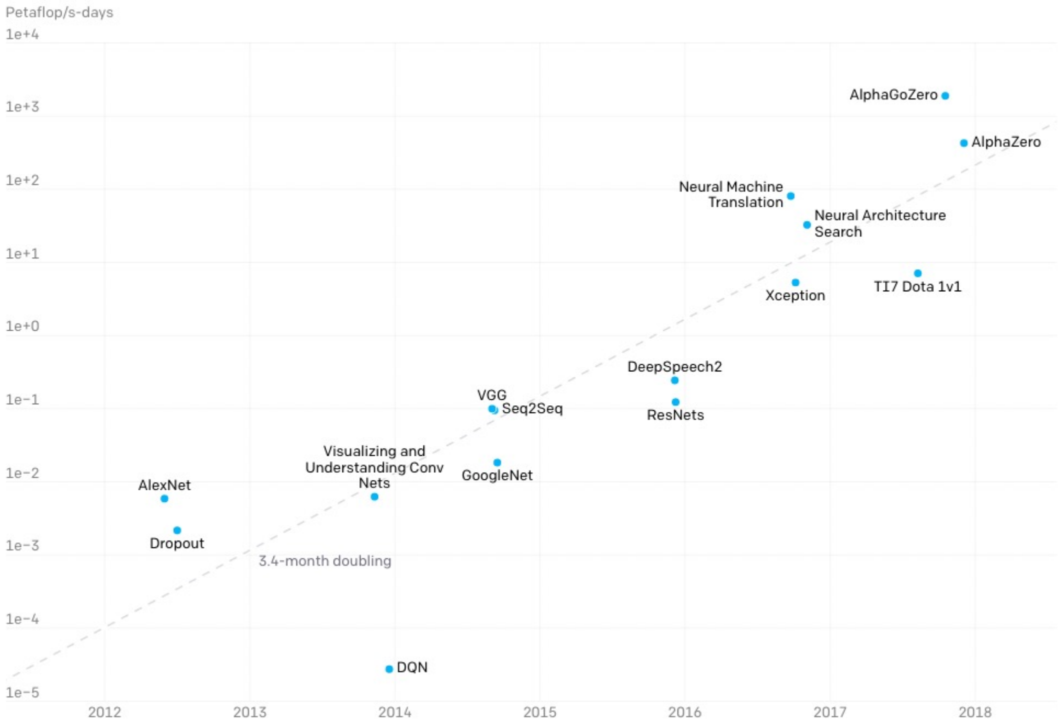
If activation function saturates, gradient is too small

$$\text{ReLU}(a) = \max(a, 0)$$

Figure 13.2 from PML

Modern networks are huge, and training can take time

Y-axis: Petaflop/s-day (pfs-day) consists of performing 10^{15} neural net operations per second for one day, or a total of about 10^{20} operations.

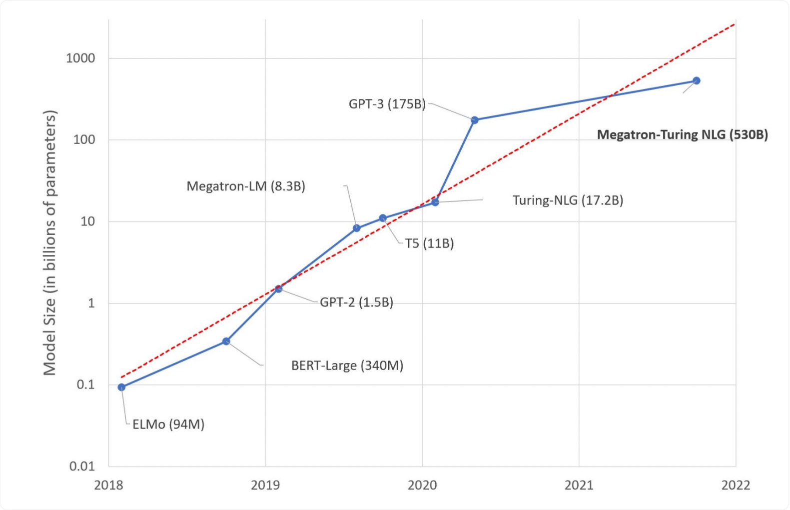


The total amount of compute, in petaflop/s-days,[2] used to train selected results that are relatively well known, used a lot of compute for their time, and gave enough information to estimate the compute used.

..since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.4-month doubling time (by comparison, Moore's Law had a 2-year doubling period). Since 2012, this metric has grown by more than 300,000x (a 2-year doubling period would yield only a 7x increase).

From https://openai.com/blog/ai-and-compute/

Modern networks are huge, and training can take time



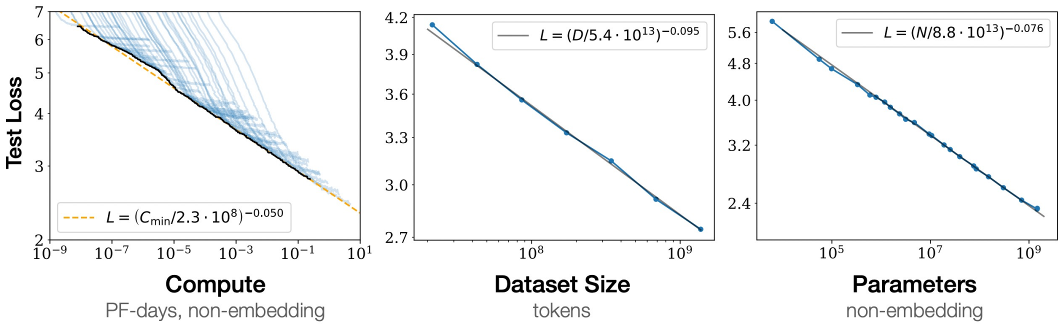


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

https://huggingface.co/blog/large-language-models

Scaling Laws for Neural Language Models [Kaplan et al.'20]

Optimization: Variants on SGD

- **mini-batch**: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)

Mini-batch

Consider $F(\boldsymbol{w}) = \sum_{i=1}^{n} F_i(\boldsymbol{w})$, where $F_i(\boldsymbol{w})$ is the loss function for the i-th datapoint.

Recall that any $\nabla\tilde{F}(\boldsymbol{w})$ is a stochastic gradient of $F(\boldsymbol{w})$ if

$$\mathbb{E}[\nabla\tilde{F}(\boldsymbol{w})] = \nabla F(\boldsymbol{w}).$$

Mini-batch SGD (also known as mini-batch GD): sample $S \subset \{1, \ldots, n\}$ at random, and estimate

the average gradient over these batch of $|S|$ samples:

$$\nabla\tilde{F}(\boldsymbol{w}) = \frac{1}{|S|}\sum_{j \in S} \nabla F_j(\boldsymbol{w}).$$

Common batch size: 32, 64, 128, etc.

Batch size s.t. a batch fits in "GPU memory".

Optimization: Variants on SGD

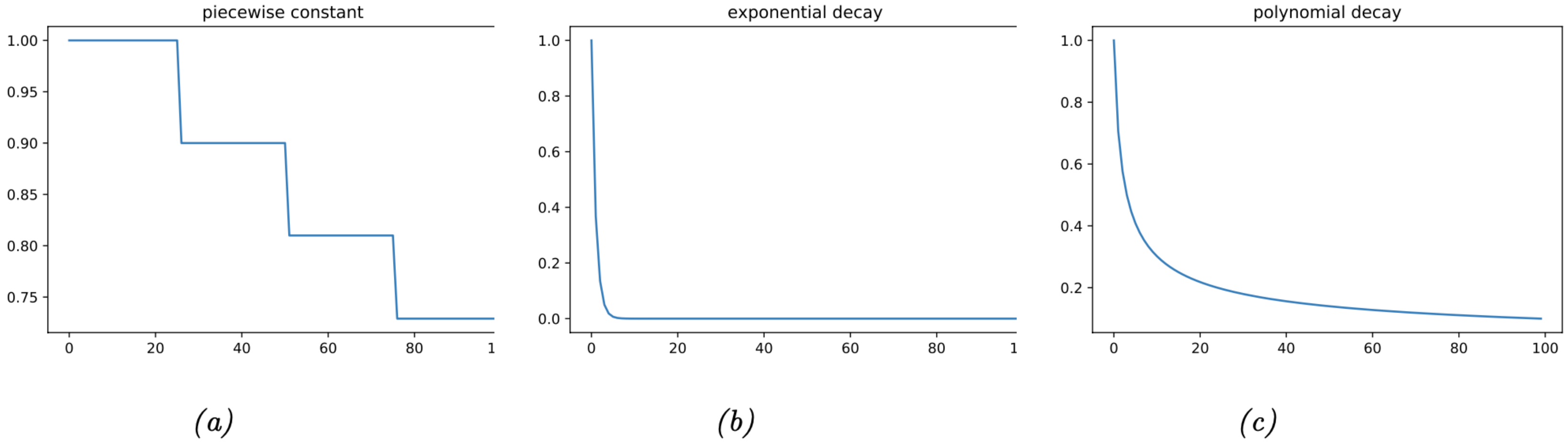
- **mini-batch**: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)

- **adaptive learning rate tuning**: choose a different learning rate for each parameter (and vary this across iterations), based on the magnitude of previous gradients for that parameter (used in Adagrad, RMSProp)

Adaptive learning rate tuning

``The learning rate is perhaps the most important hyperparameter.
If you have time to tune only one hyperparameter, tune the learning rate.”
-Deep learning (Book by Goodfellow, Bengio, Courville)

We often use a **learning rate schedule**.



Some common learning rate schedules (figure from PML)

Adaptive learning rate methods (Adagrad, RMSProp) scale the learning rate of each parameter based on some moving average of the magnitude of the gradients.

Optimization: Variants on SGD

- **mini-batch**: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)

- **adaptive learning rate tuning**: choose a different learning rate for each parameter (and vary this across iterations), based on the magnitude of previous gradients for that parameter (used in Adagrad, RMSProp)

- **momentum**: add a “momentum” term to encourage model to continue along previous gradient direction

Momentum

“move faster along directions that were previously good, and to slow down along directions where the gradient has suddenly changed, just like a ball rolling downhill.” [PML]

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

For $t = 1, 2, \ldots$

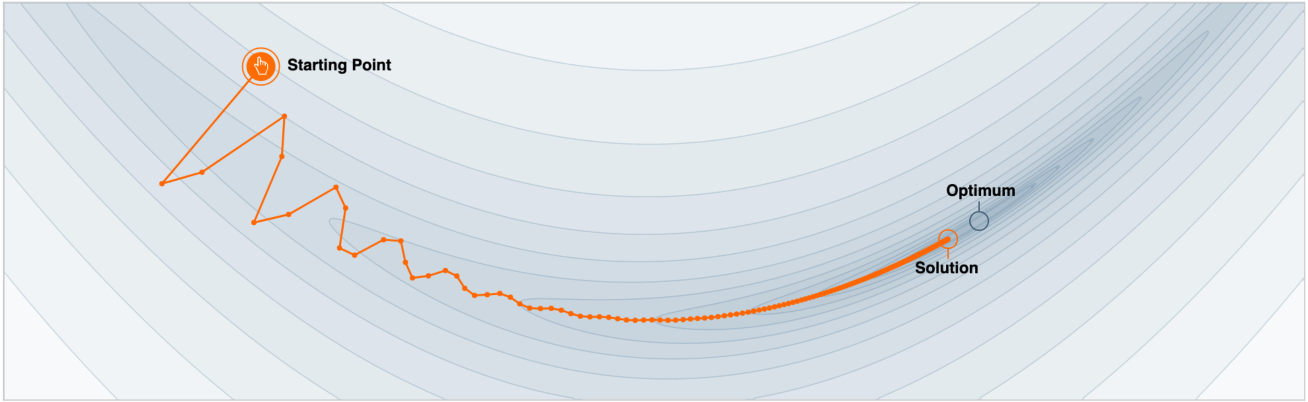
- estimate a stochastic gradient $\boldsymbol{g}_t$
- update $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} + \boldsymbol{g}_t$ for some discount factor $\alpha \in (0, 1)$
- update weight $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta\boldsymbol{v}$

Updates for first few rounds:

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

Momentum

Why Momentum Really Works



Step-size α = 0.02

0 0.003 0.006

Momentum β = 0.99

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

https://distill.pub/2017/momentum/

Optimization: Variants on SGD

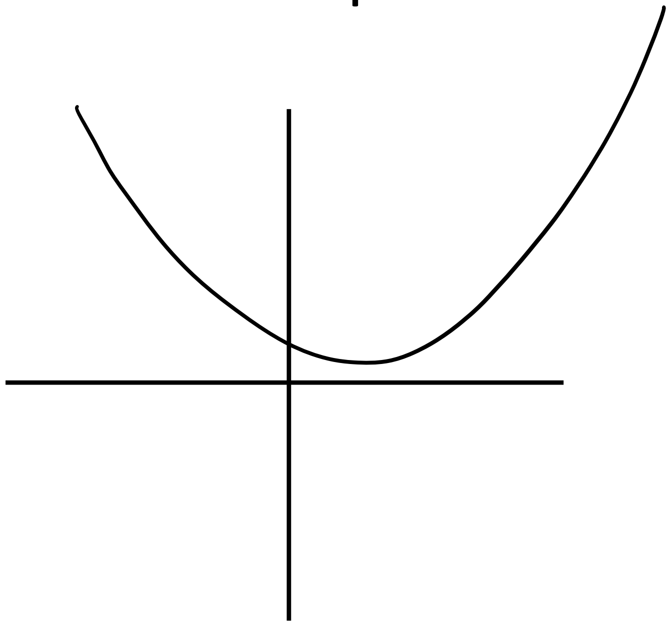
- **mini-batch**: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)

- **adaptive learning rate tuning**: choose a different learning rate for each parameter (and vary this across iterations), based on the magnitude of previous gradients for that parameter (used in Adagrad, RMSProp)

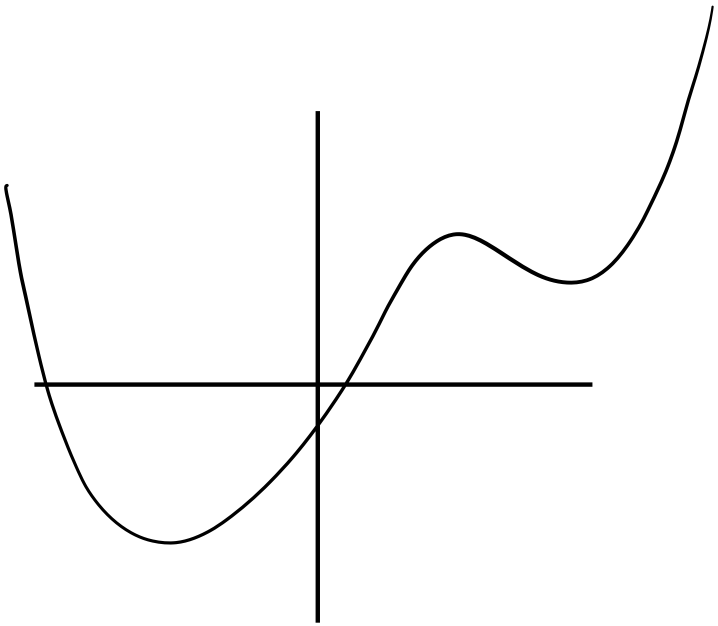
- **momentum**: add a “momentum” term to encourage model to continue along previous gradient direction

- Many other variants and tricks such as **batch normalization**: normalize the inputs of each layer over the mini-batch (to zero-mean and unit-variance; like we did in HW1)

Optimization: Initialization



For convex problems, initialization does not matter



It can make a difference for non-convex problems

For convex problems, you could just initialize at 0
Initializing neural network at all-zeroes is not good, gradients for all weights in a layer will be the same.

To break symmetry, do random initialization (various default schemes)

3.3 Generalization: Preventing Overfitting

Overfitting can be a major concern since neural nets are very powerful.

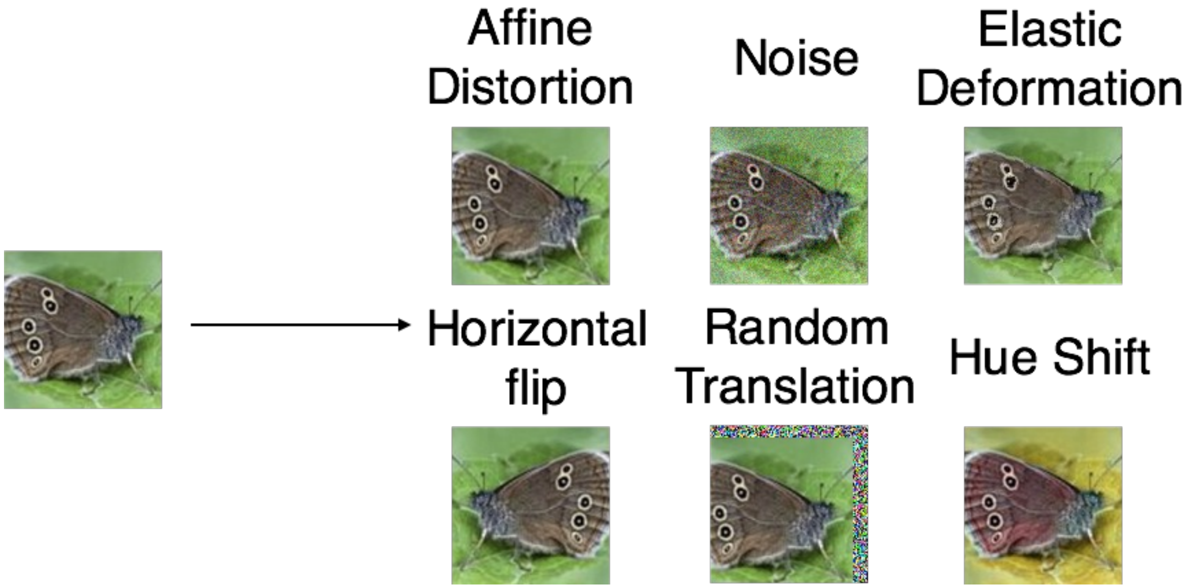
Methods to overcome overfitting:

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

Preventing overfitting: **Data augmentation**

The best way to prevent overfitting? Get more samples.
What if you cannot get access to more samples?

Exploit prior knowledge to add more training data:

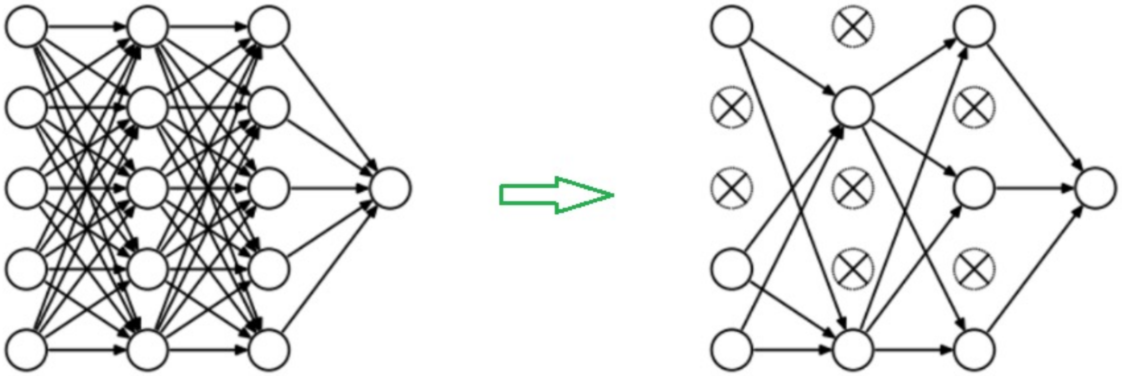


Preventing overfitting: Regularization & Dropout

We can use regularization techniques such as ℓ_2 regularization.
ℓ_2 regularization: minimize

$$G(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}) = F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}) + \lambda \sum_{\substack{\text{all weights } w \\ \text{in network}}} w^2$$

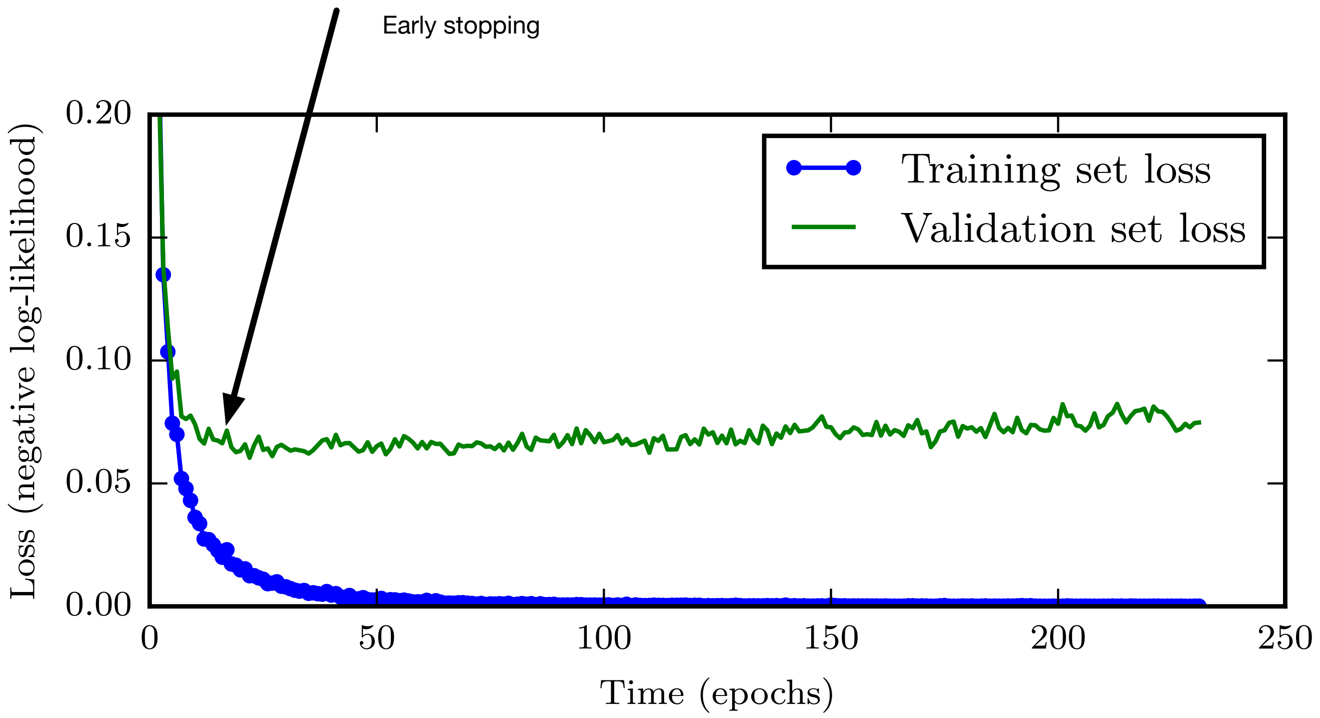
A very popular technique is **Dropout**. Here, we *independently delete each neuron* with a fixed probability (say 0.1), during each iteration of Backprop (only for training, not for testing)



Very effective and popular in practice!

Preventing overfitting: **Early stopping**

Stop training when the performance on validation set stops improving



There are **big mysteries** about how and why deep learning works

- Why are certain architectures better for certain problems? How should we design architectures?
- Why do gradient-based methods work on these highly-nonconvex problems?
- Why can deep networks generalize well despite having the capacity to so easily overfit?
- What implicit regularization effects do gradient-based methods provide?
- …

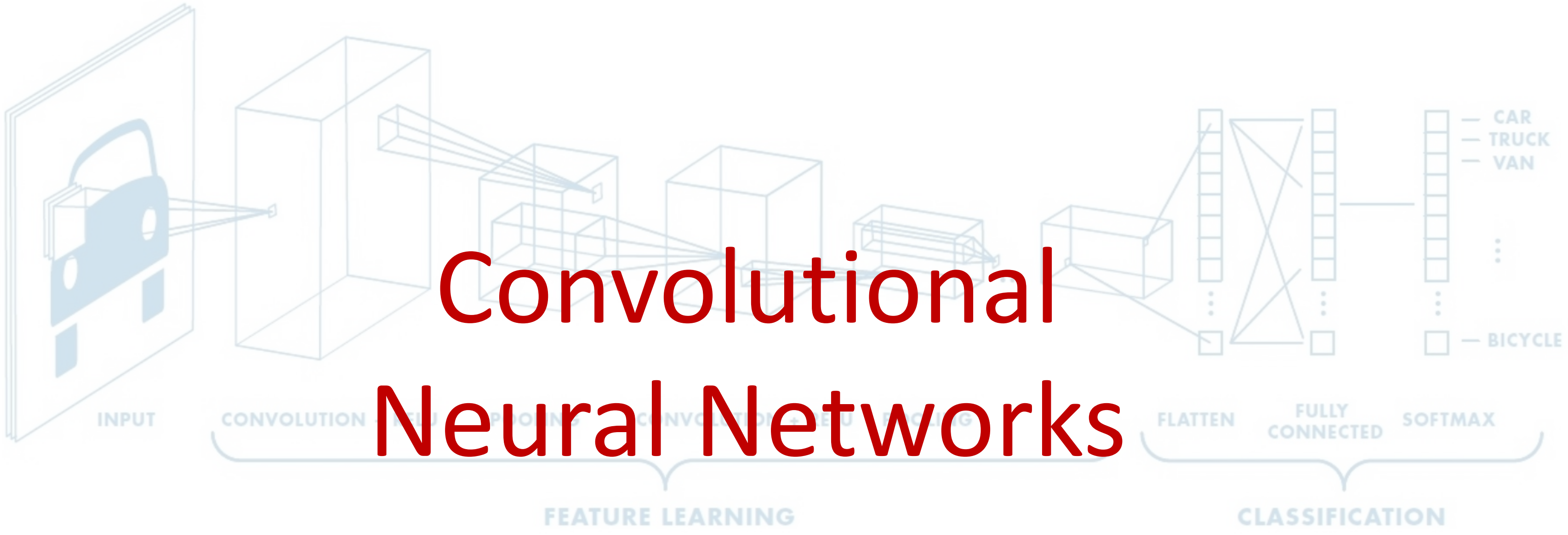
Picture credit: NASA/JPL-Caltech

Neural networks: Summary

Deep neural networks

- are hugely popular, achieving *best performance* on many problems
- do need *a lot of data* to work well
- can 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

Convolutional Neural Networks



Acknowledgements

Not much math in this part, but there'll be empirical intuition (and cat pictures ☺)

The materials in this part borrow heavily from the following sources:

- Stanford's CS231n: http://cs231n.stanford.edu/
- Deep learning book by Goodfellow, Bengio and Courville: http://deeplearningbook.org

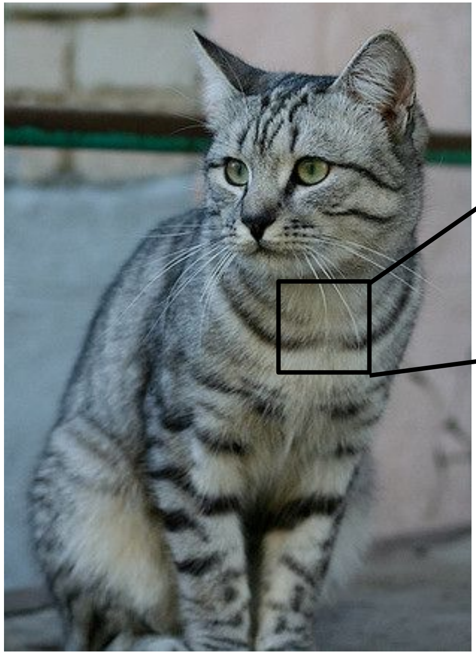
Both website provides a lot of useful resources: notes, demos, videos, etc.

Image Classification: A core task in Computer Vision

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

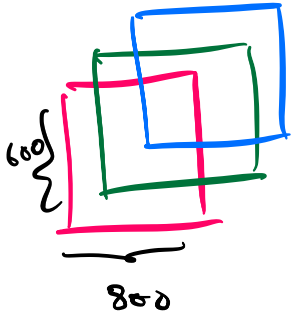
→ cat

The Problem: Semantic Gap




```
[[105 112 108 111 104  99 106  99  96 103 112 119 104  97  93  87]
 [ 91  98 102 106 104  79  98 103  99 105 123 136 110 105  94  85]
 [ 76  85  90 105 128 105  87  96  95  99 115 112 106 103  99  85]
 [ 99  81  81  93 120 131 127 100  95  98 102  99  96  93 101  94]
 [106  91  61  64  69  91  88  85 101 107 109  98  75  84  96  95]
 [114 108  85  55  55  69  64  54  64  87 112 129  98  74  84  91]
 [133 137 147 103  65  81  80  65  52  54  74  84 102  93  85  82]
 [128 137 144 140 109  95  86  70  62  65  63  63  60  73  86 101]
 [125 133 148 137 119 121 117  94  65  79  80  65  54  64  72  98]
 [127 125 131 147 133 127 126 131 111  96  89  75  61  64  72  84]
 [115 114 109 123 150 148 131 118 113 109 100  92  74  65  72  78]
 [ 89  93  90  97 108 147 131 118 113 114 113 109 106  95  77  80]
 [ 63  77  86  81  77  79 102 123 117 115 117 125 125 130 115  87]
 [ 62  65  82  89  78  71  80 101 124 126 119 101 107 114 131 119]
 [ 63  65  75  88  89  71  62  81 120 138 135 105  81  98 110 118]
 [ 87  65  71  87 106  95  69  45  76 130 126 107  92  94 105 112]
 [118  97  82  86 117 123 116  66  41  51  95  93  89  95 102 107]
 [164 146 112  80  82 120 124 104  76  48  45  66  88 101 102 109]
 [157 170 157 120  93  86 114 132 112  97  69  55  70  82  99  94]
 [130 128 134 161 139 100 109 118 121 134 114  87  65  53  69  86]
 [128 112  96 117 150 144 120 115 104 107 102  93  87  81  72  79]
 [123 107  96  86  83 112 153 149 122 109 104  75  80 107 112  99]
 [122 121 102  80  82  86  94 117 145 148 153 102  58  78  92 107]
 [122 164 148 103  71  56  78  83  93 103 119 139 102  61  69  84]]
```

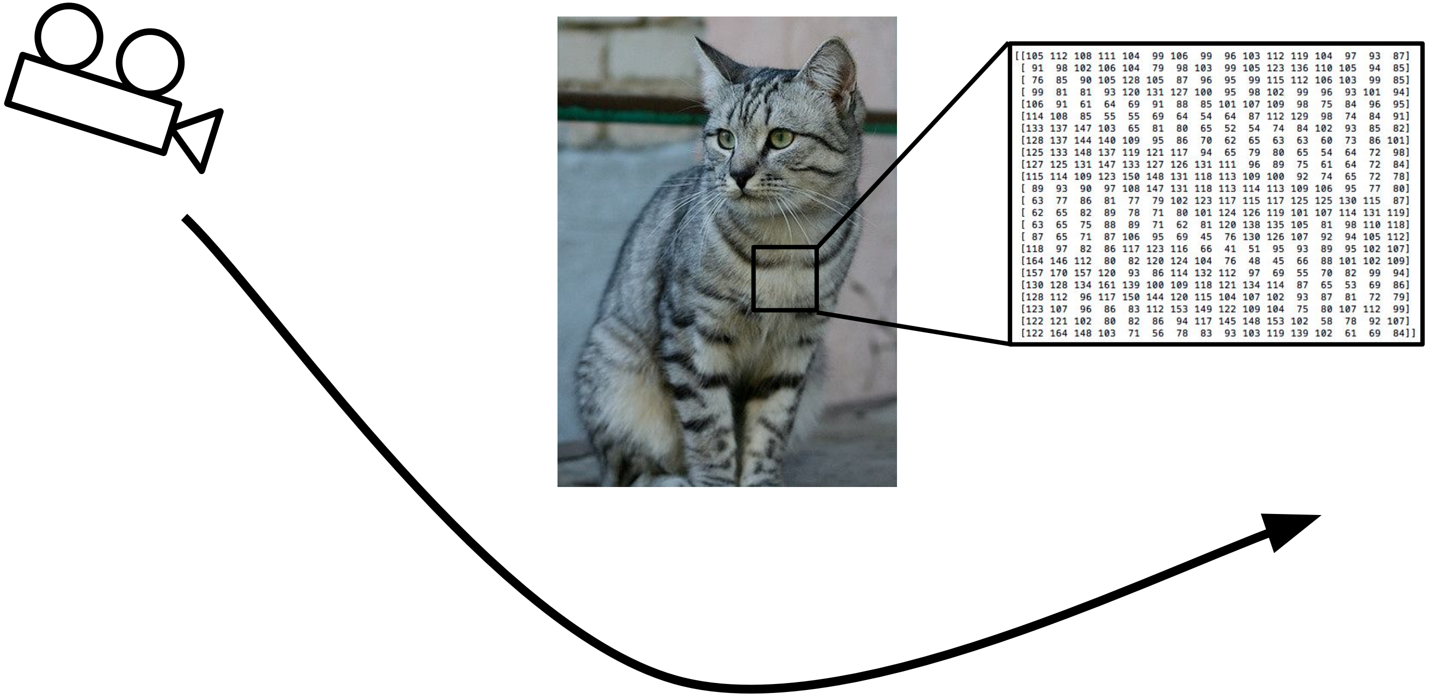
What the computer sees



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



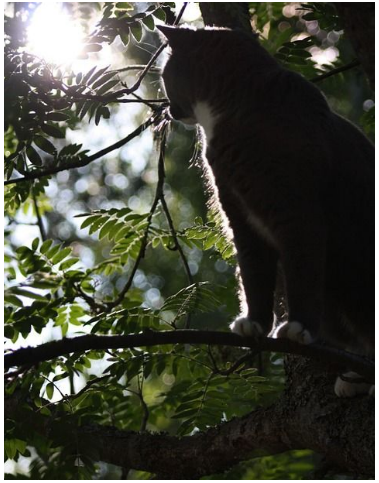
All pixels change when
the camera moves!

Challenges: Illumination

This image is CC0 1.0 public domain

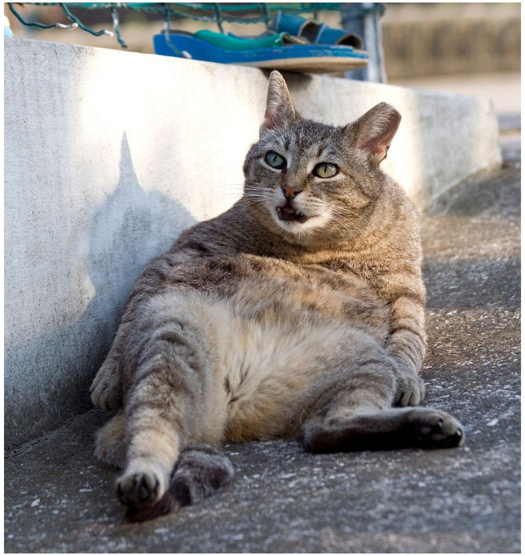
This image is CC0 1.0 public domain

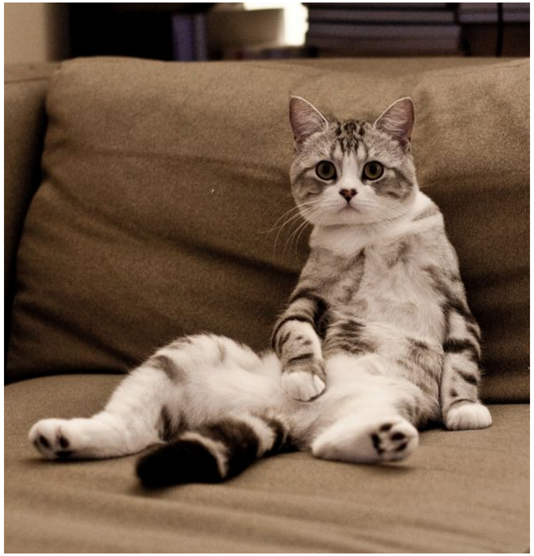
This image is CC0 1.0 public domain



This image is CC0 1.0 public domain

Challenges: Deformation





Challenges: Occlusion

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

An image classifier

```
def classify_image(image):
  # Some magic here?
  return class_label
```

Unlike e.g. sorting a list of numbers,

no obvious way to hard-code the algorithm for recognizing a cat, or other classes.

Attempts have been made

Find edges

Find corners

↓ ← ↑ →

↓

?

John Canny, "A Computational Approach to Edge Detection", IEEE TPAMI 1986

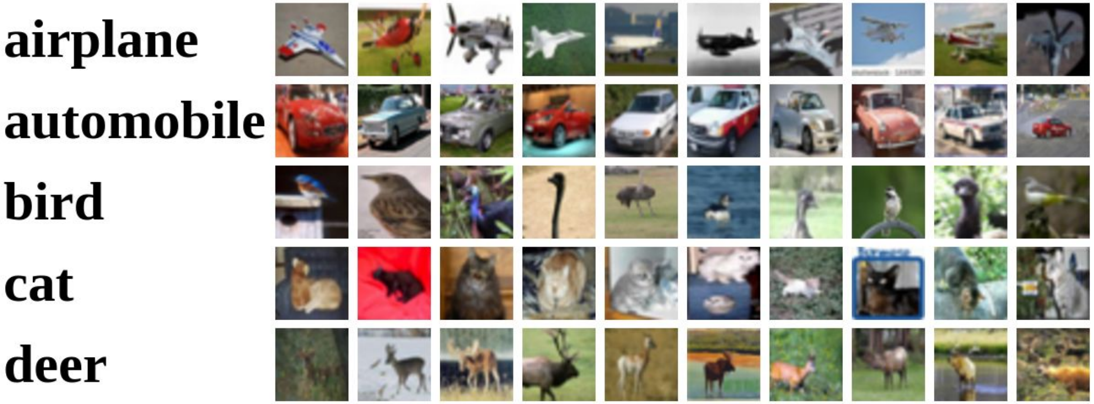
Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning to train a classifier
3. Evaluate the classifier on new images

```python
def train(images, labels):
  # Machine learning!
  return model
```

```python
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

Example training set



The challenge

How do we train a model that can do well despite all these variations?

The ingredients:

- ***A lot of data*** (so that these variations are observed).
- ***Huge models*** with the capacity to consume and learn from all this data (and the ***computational infrastructure*** to enable training)

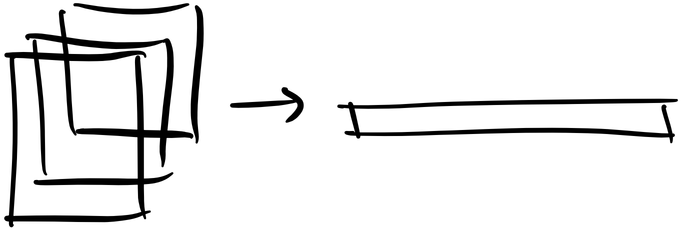
What helps:

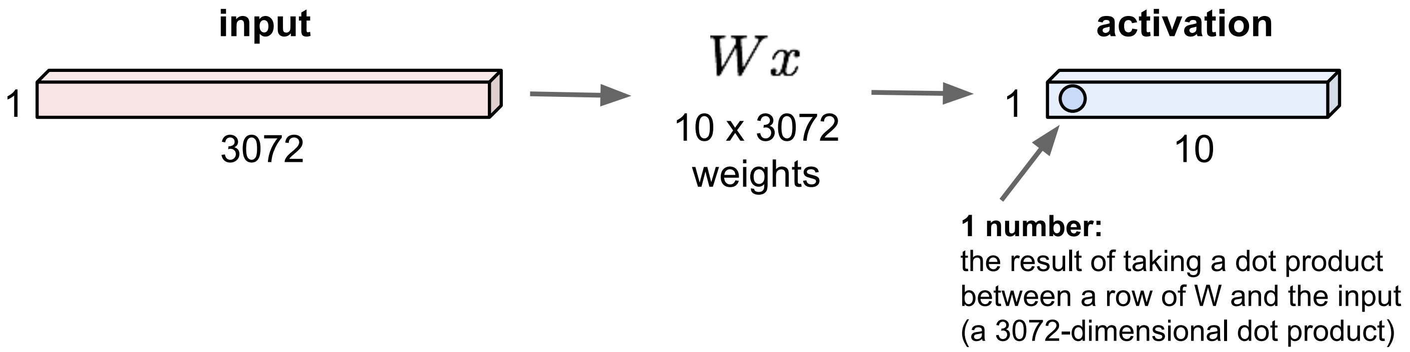
- Models with the right properties which makes the process easier (goes back to our discussion of *choosing the function class*).

The problem with standard NN for image inputs

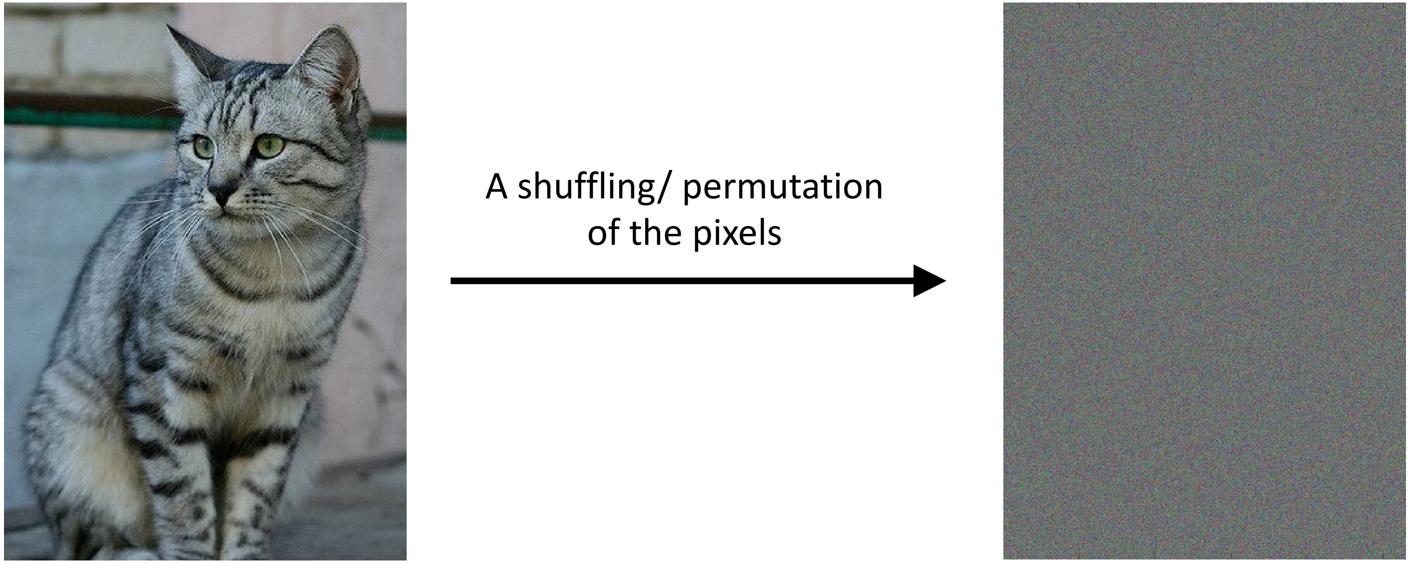
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1





The task is as easy, or rather as difficult, for a fully-connected network even if I shuffle the pixels.
Is this okay?



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

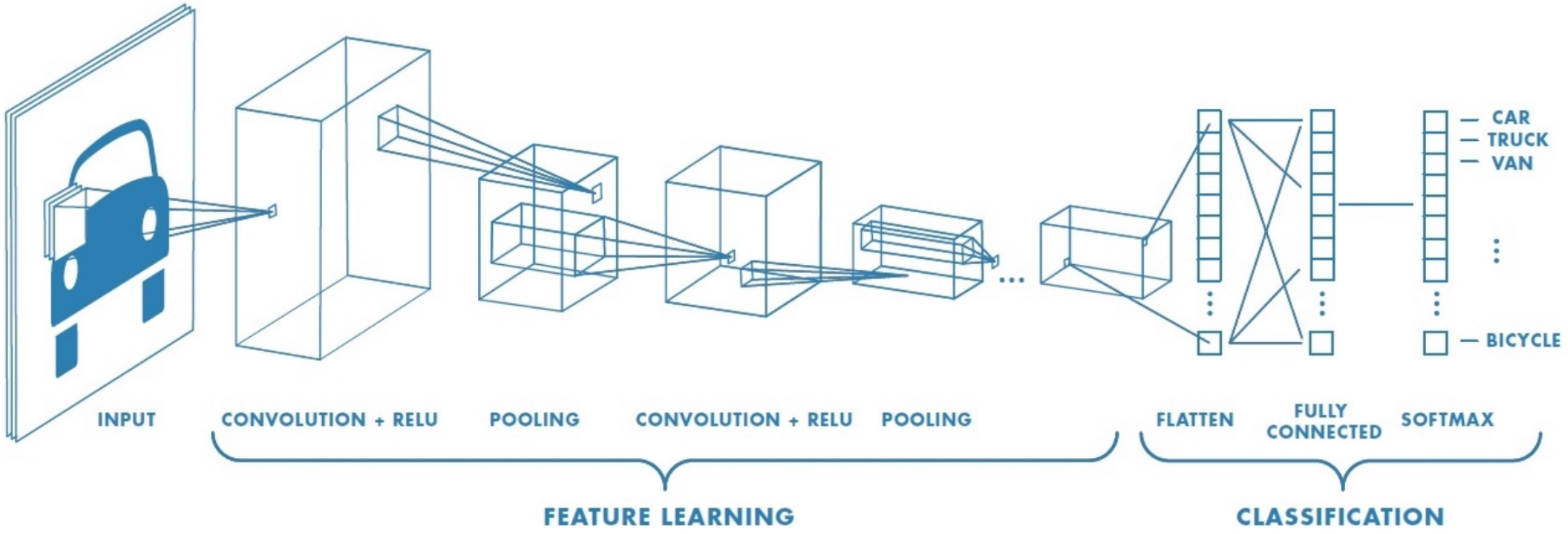


Figure from https://blog.floydhub.com/building-your-first-convnet/

2-D Convolution

$= 0 \cdot 1 + 1 \cdot 1 + 3 \cdot 2 + 4 \cdot 3$

input (3×3)

this is a convolution

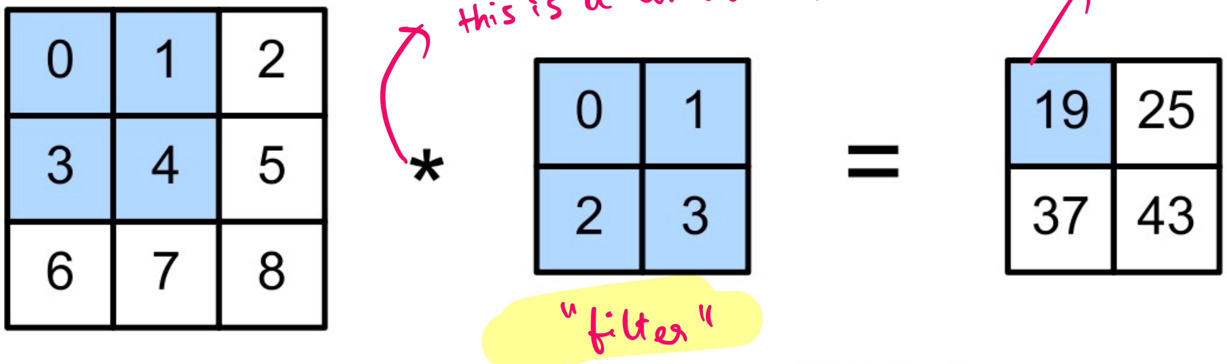


Figure 14.5: Illustration of 2d cross correlation. Generated by conv2d_jax.ipynb. Adapted from Figure 6.2.1 of [Zha+20].

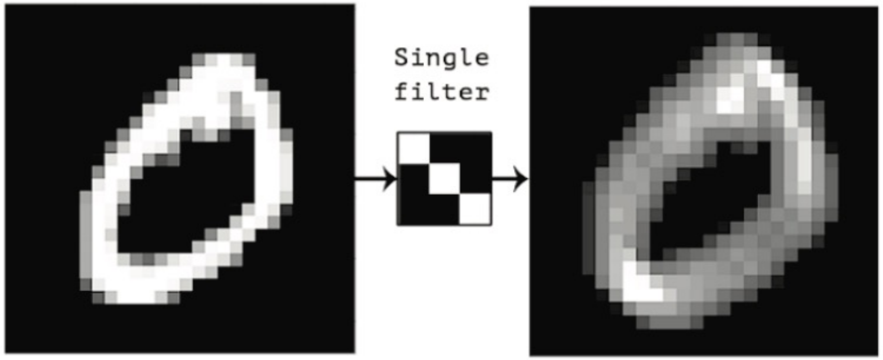


Figure 14.6: Convolving a 2d image (left) with a 3 × 3 filter (middle) produces a 2d response map (right). The bright spots of the response map correspond to locations in the image which contain diagonal lines sloping down and to the right. From Figure 5.3 of [Cho17]. Used with kind permission of Francois Chollet.

Figures from PML

3-D Convolution

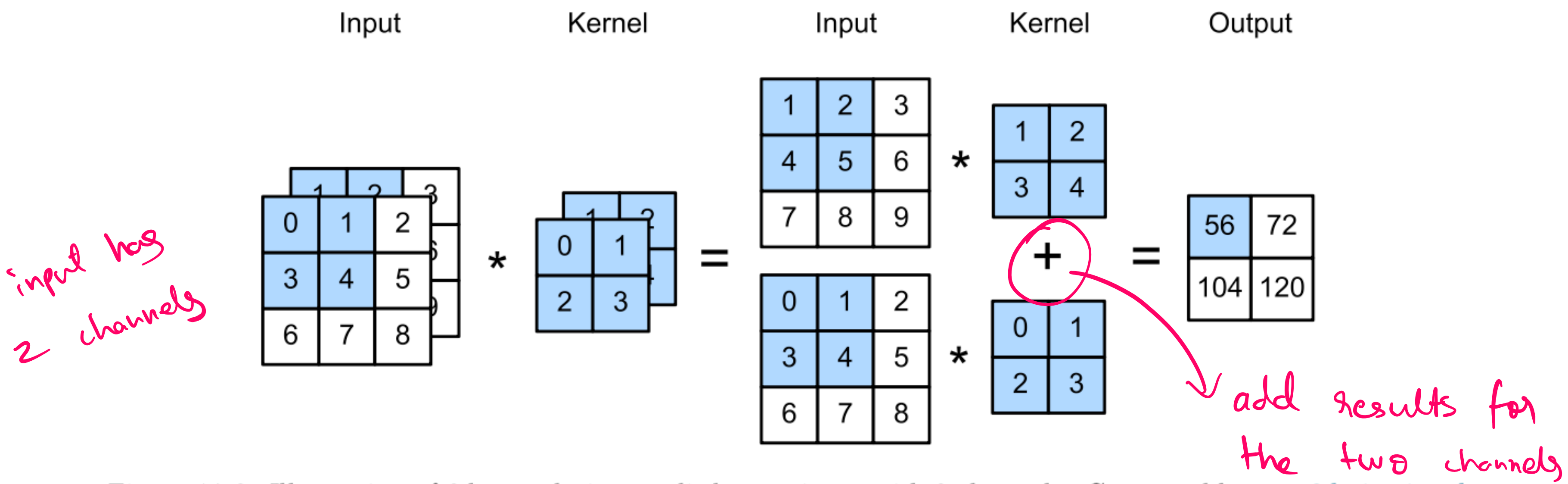
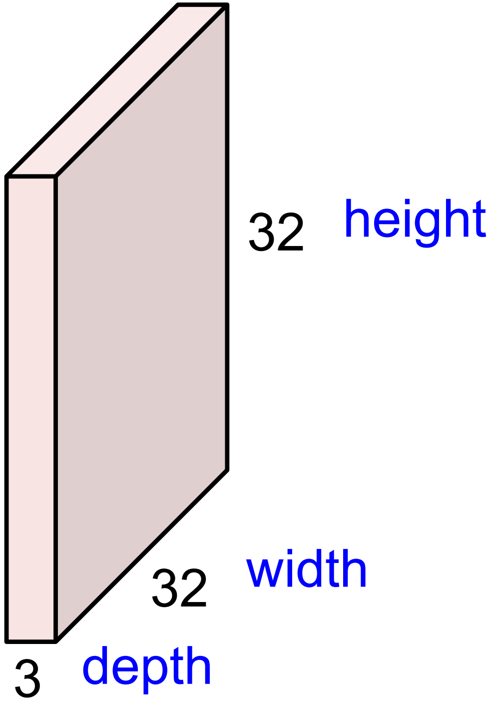


Figure 14.9: Illustration of 2d convolution applied to an input with 2 channels. Generated by conv2d_jax.ipynb. Adapted from Figure 6.4.1 of [Zha+20].

Figure from PML

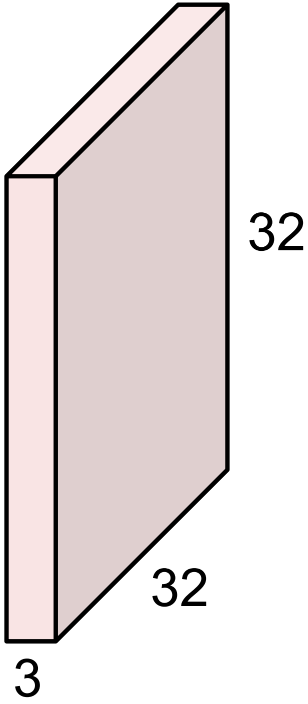
Convolution Layer

32x32x3 image -> preserve spatial structure

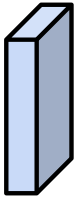


Convolution Layer

32x32x3 image



5x5x3 filter

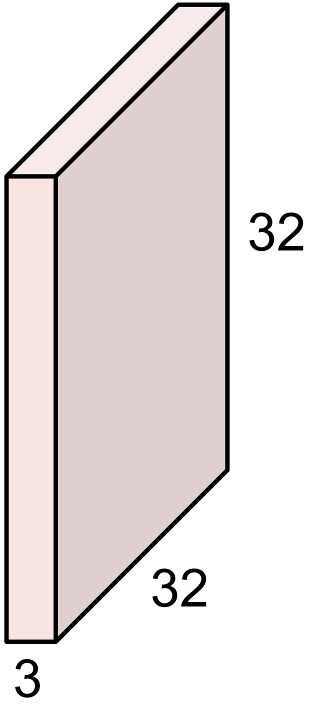


Convolve the filter with the image
i.e. "slide over the image spatially,
computing dot products"

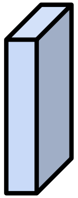
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

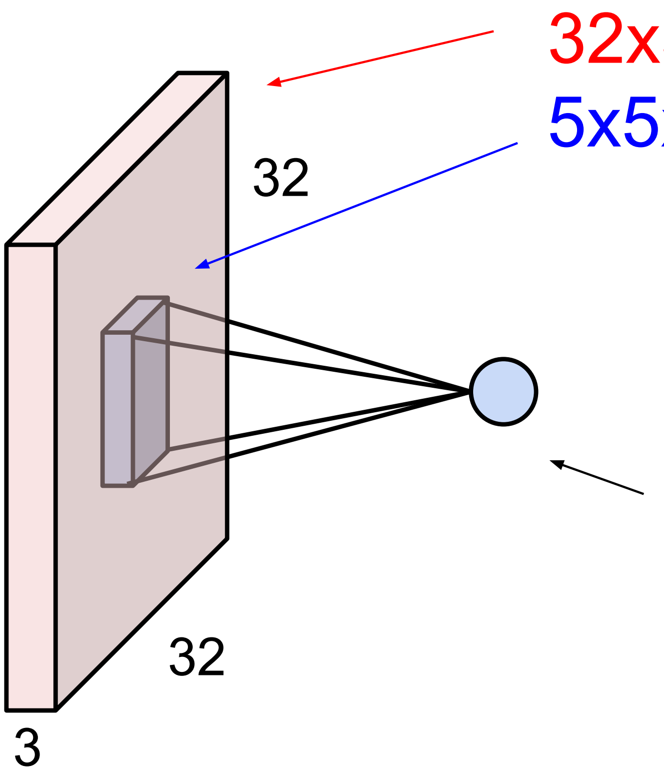


5x5x3 filter



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

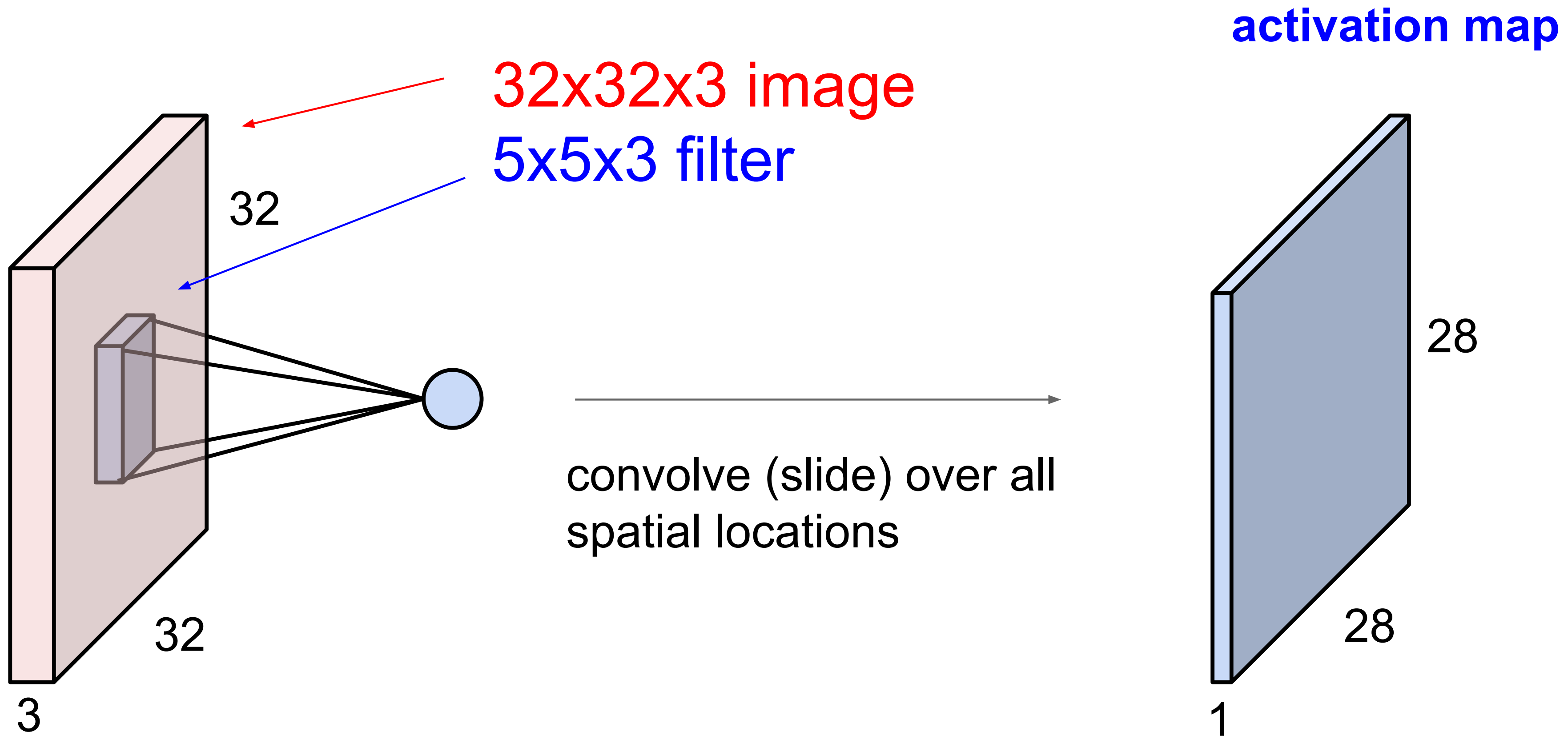
Convolution Layer



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)

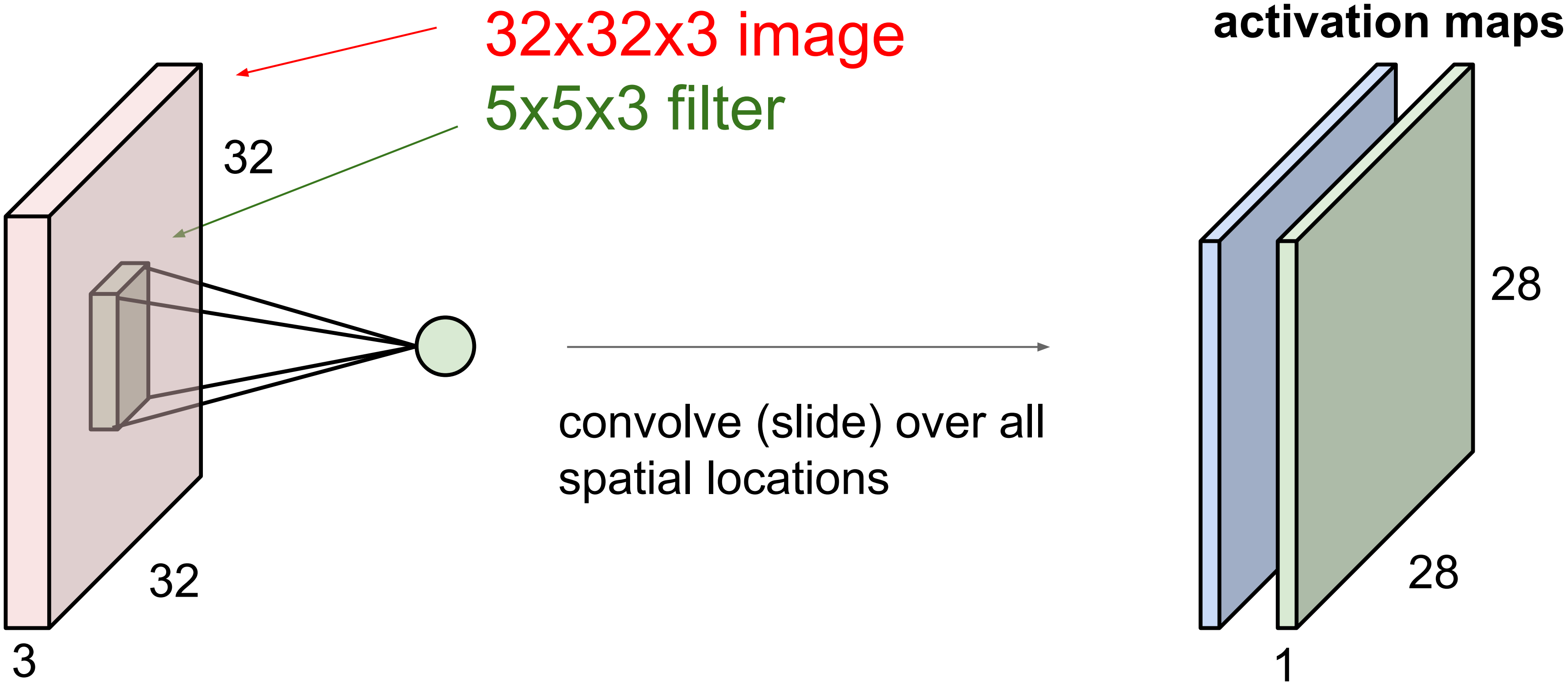
$$w^T x + b$$

Convolution Layer

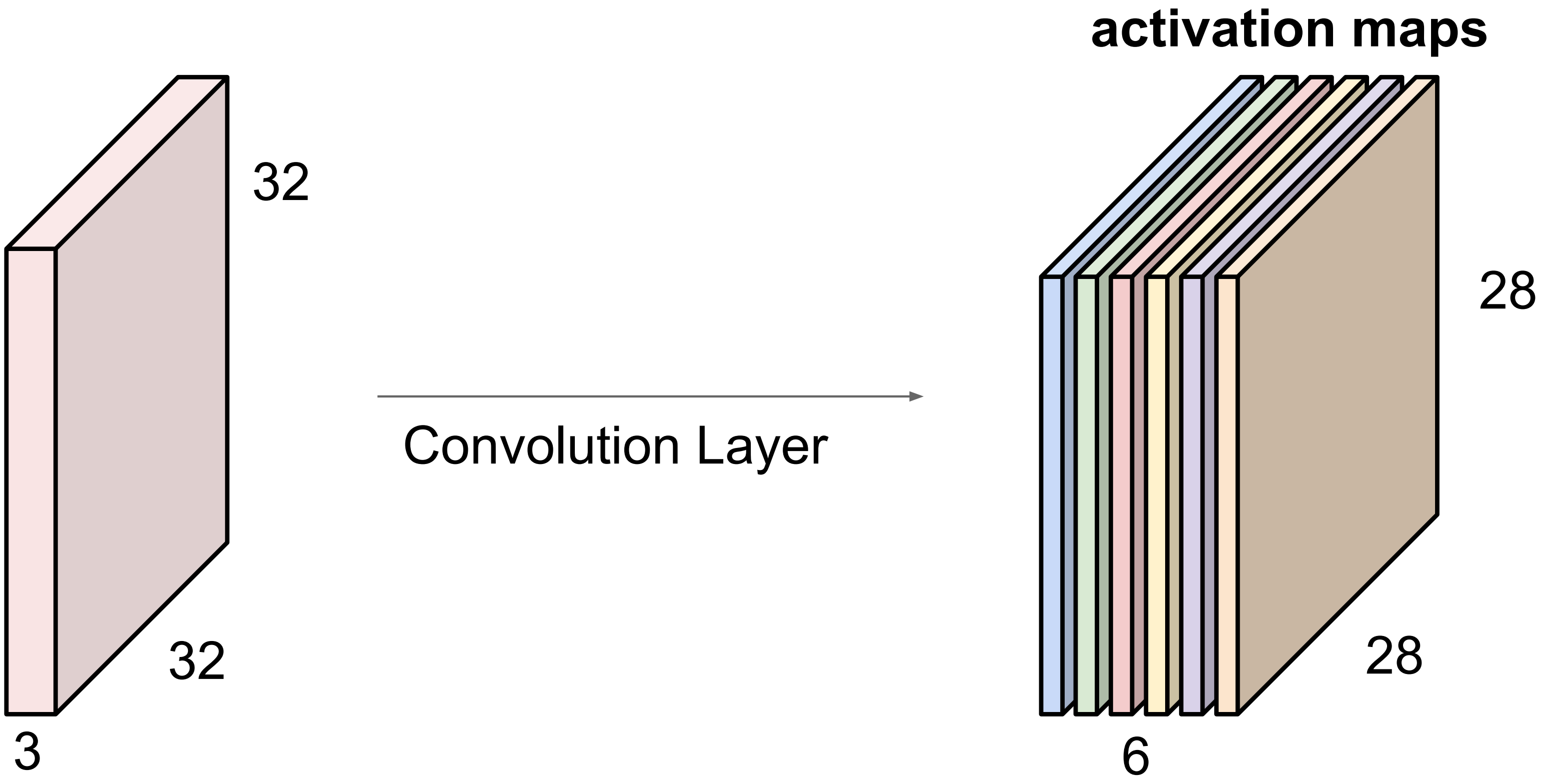


Convolution Layer

consider a second, green filter

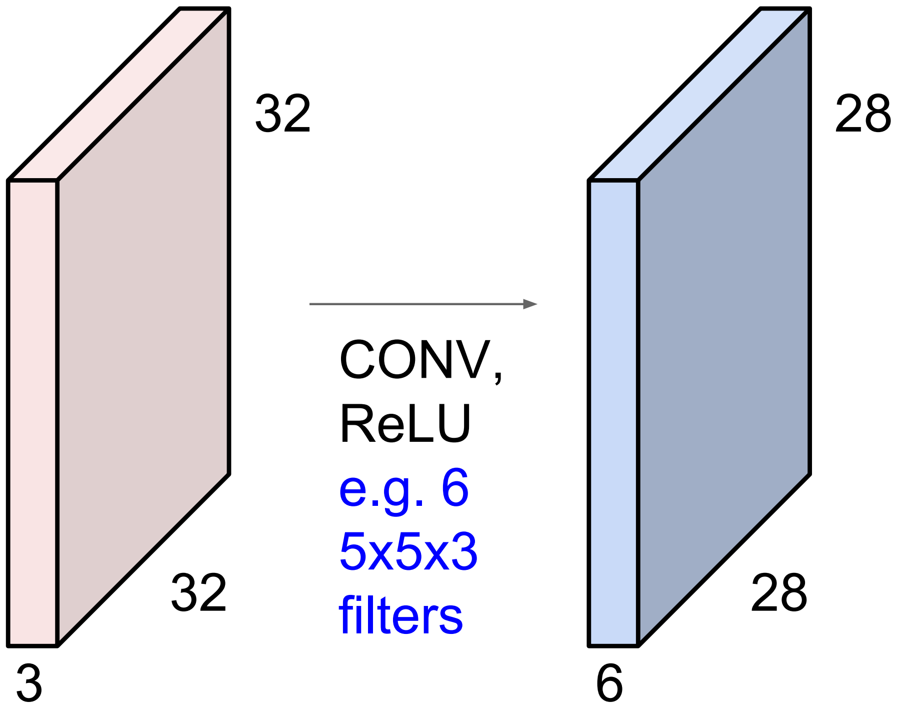


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

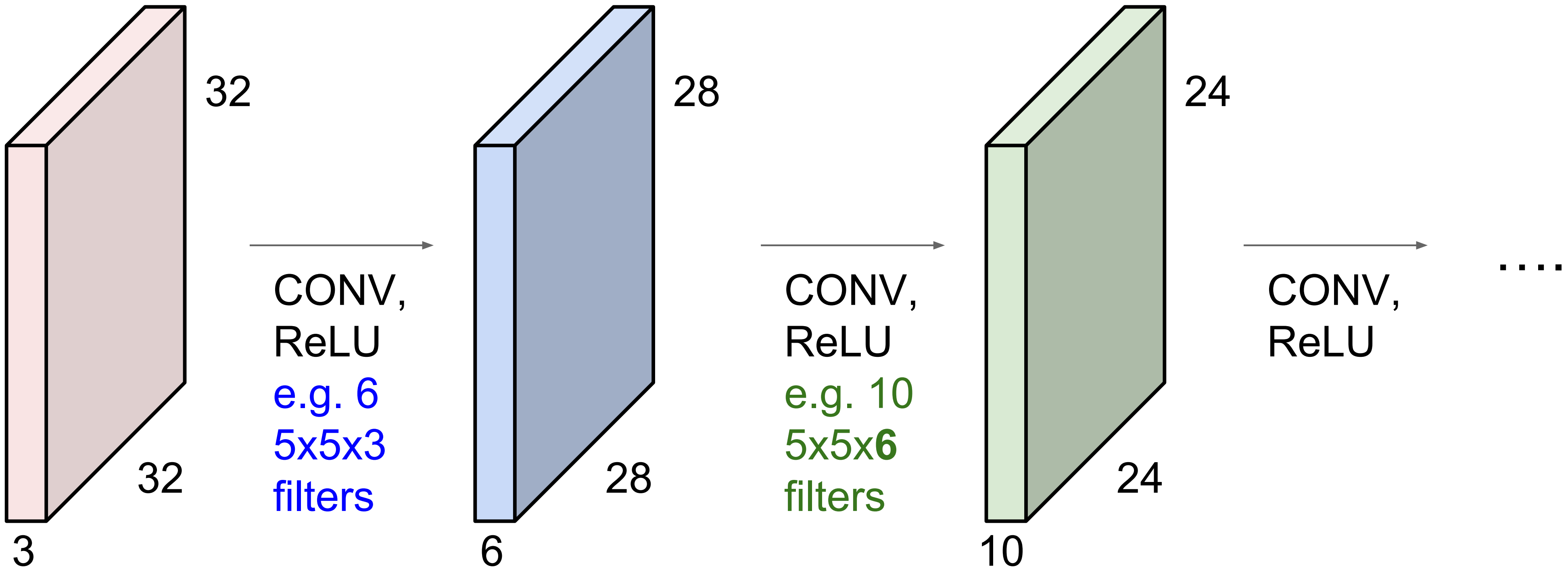


We stack these up to get a "new image" of size 28x28x6!

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



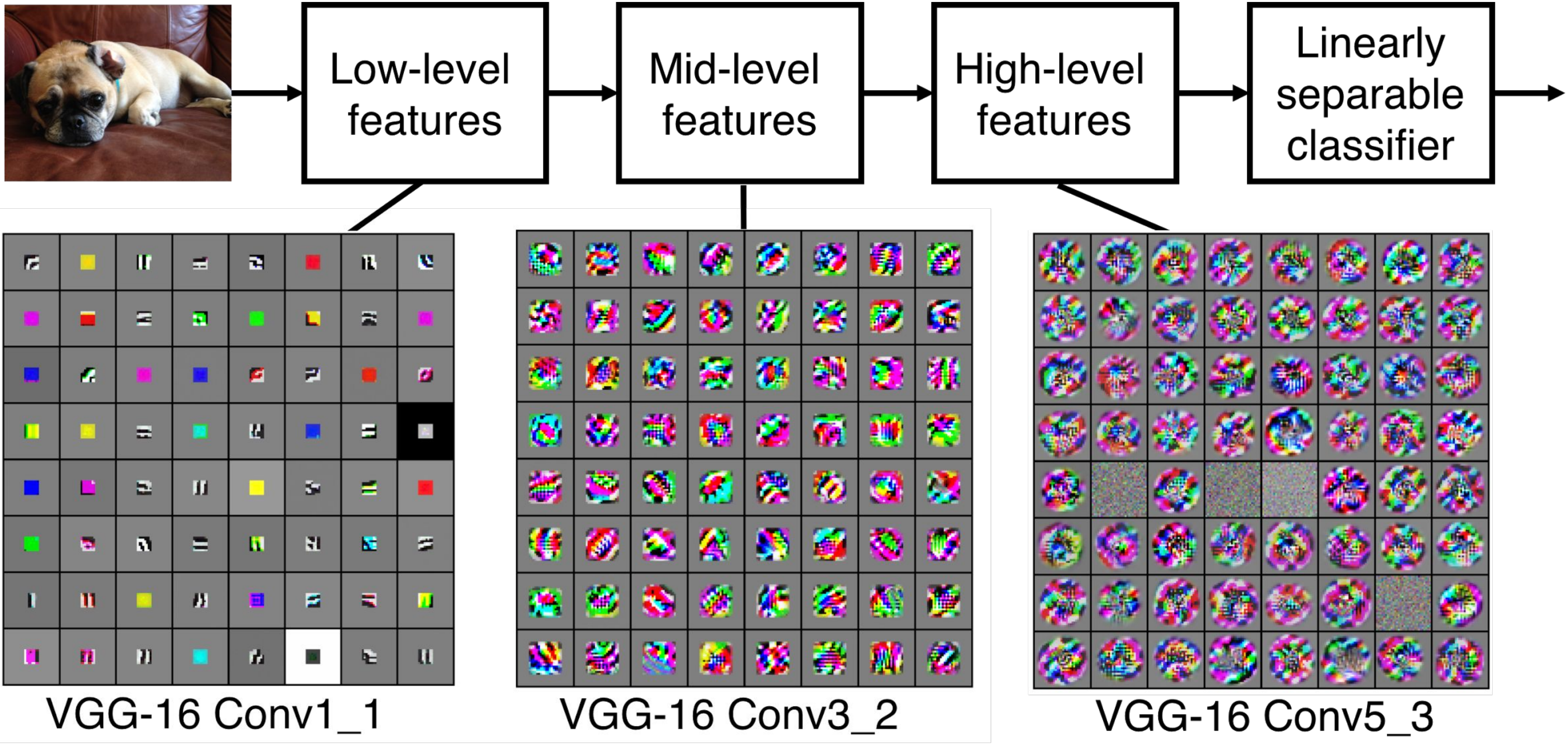
Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



Preview

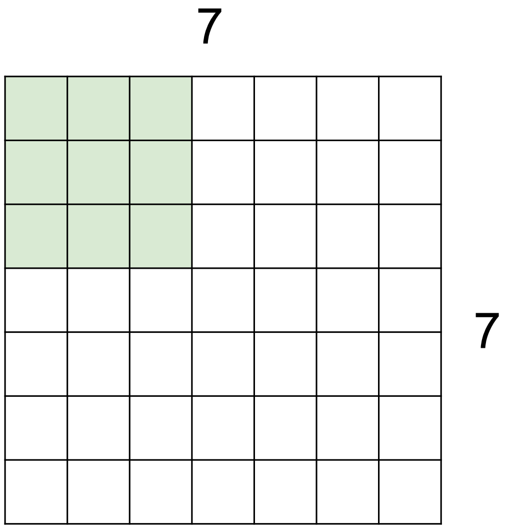
[Zeiler and Fergus 2013]

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



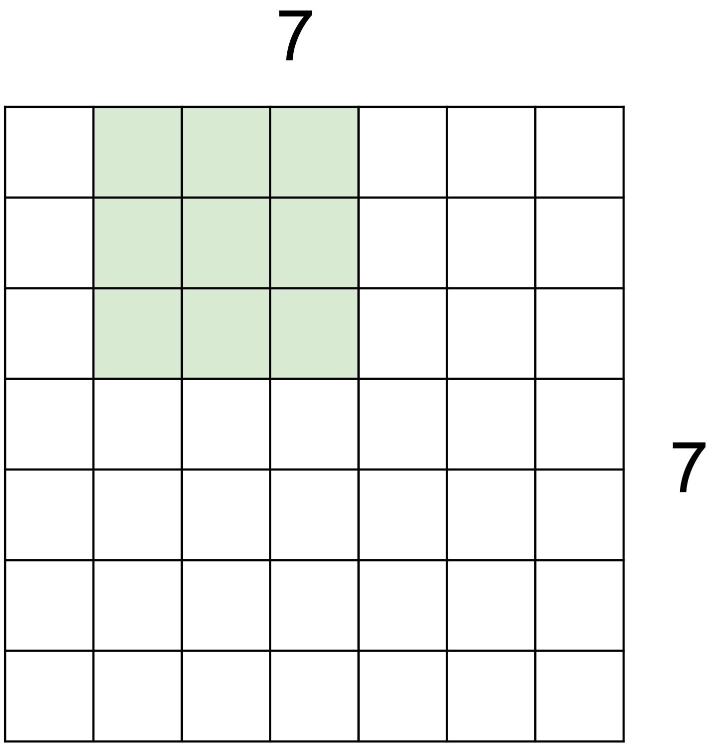
Understanding spatial dimensions of Conv layer

A closer look at spatial dimensions:



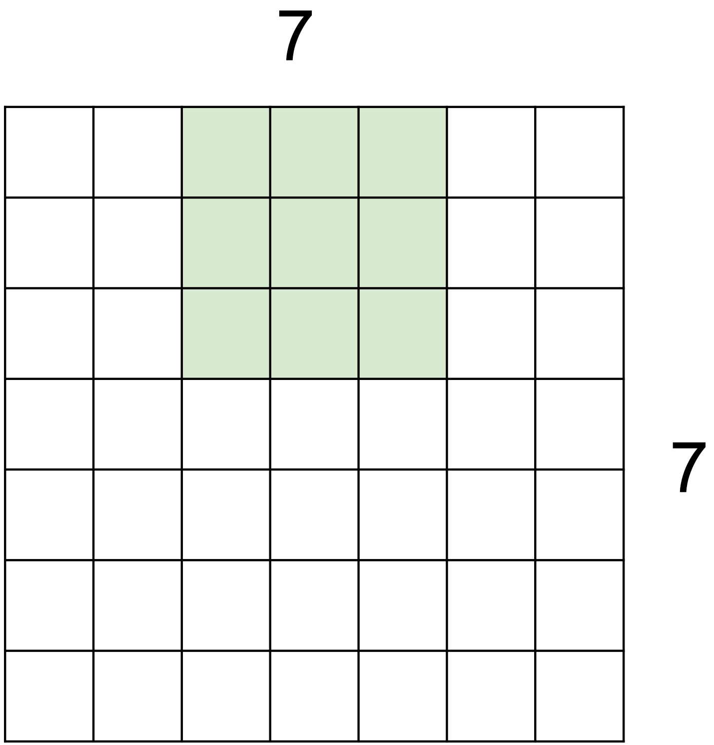
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



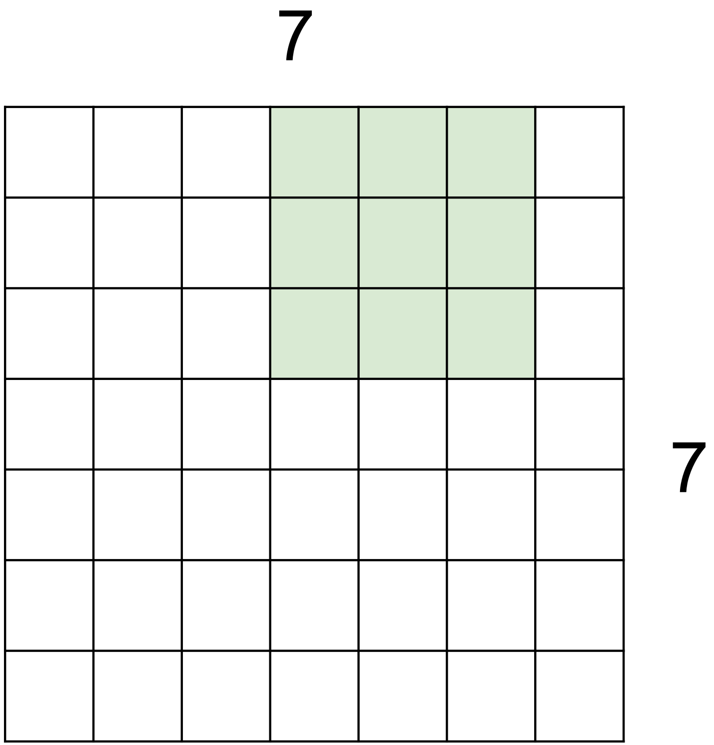
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



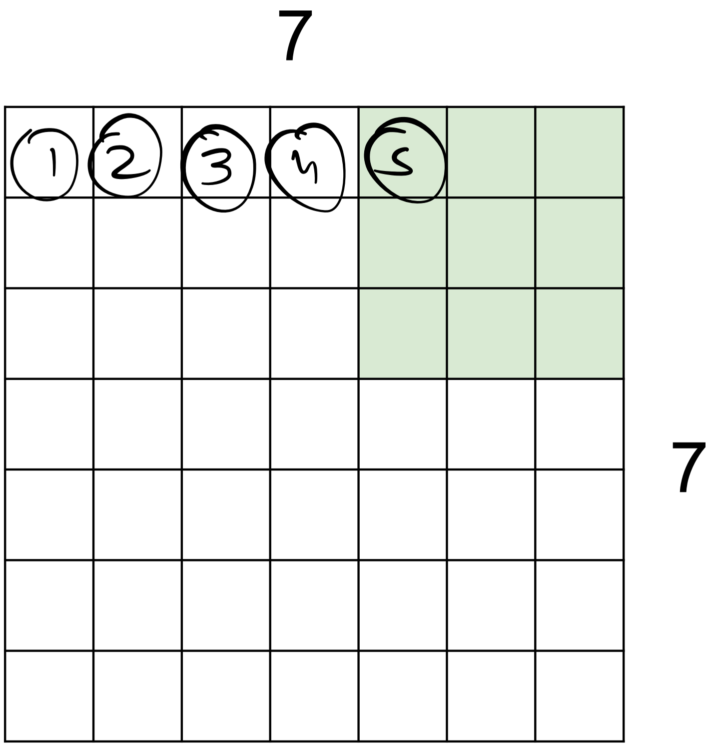
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

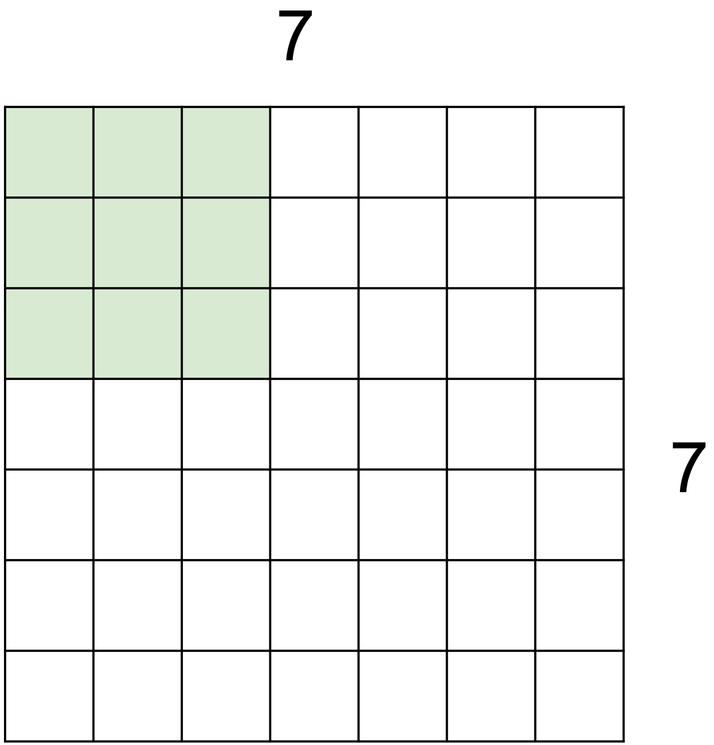
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

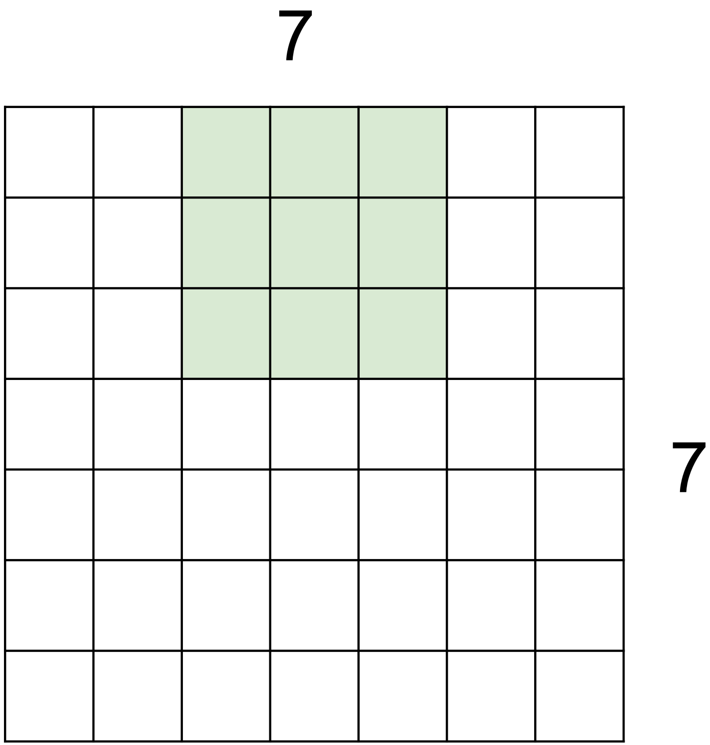
=> 5x5 output

A closer look at spatial dimensions:



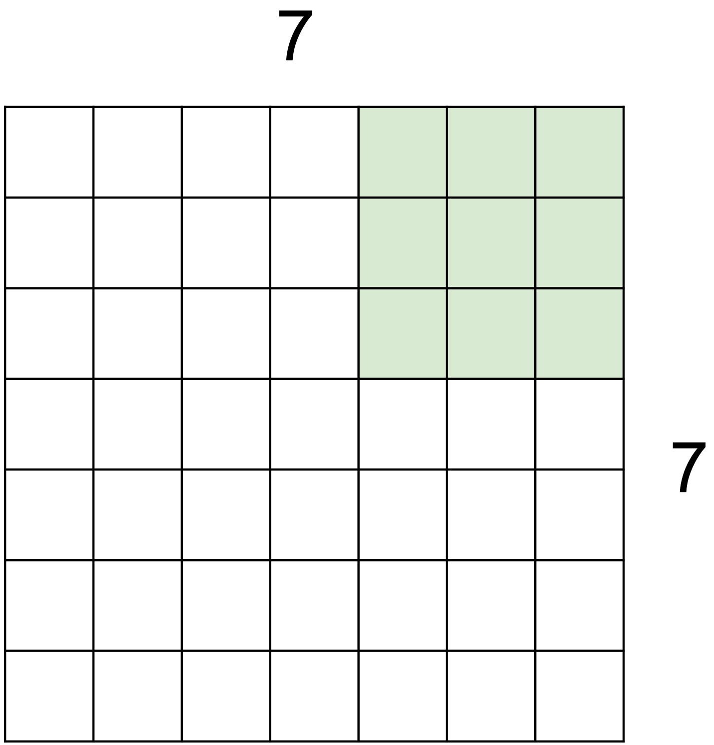
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



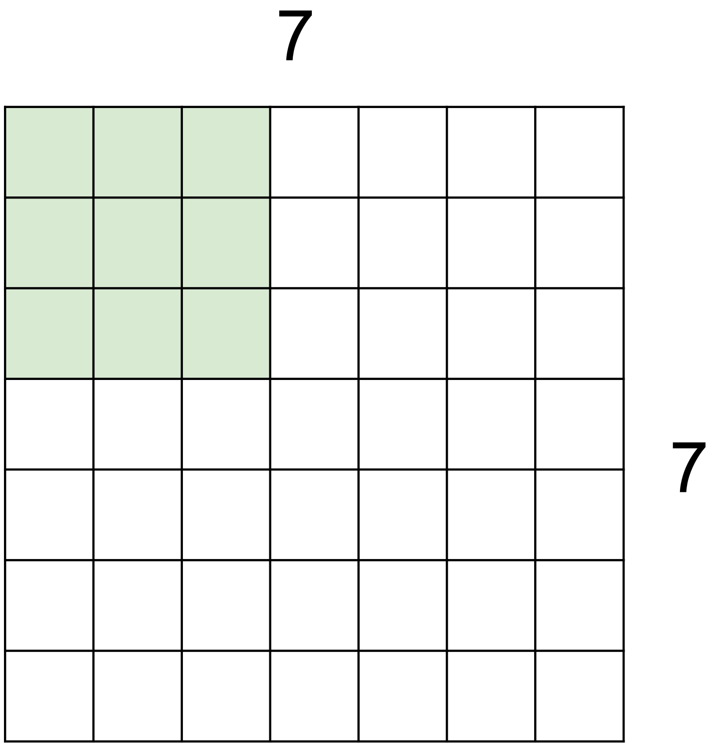
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



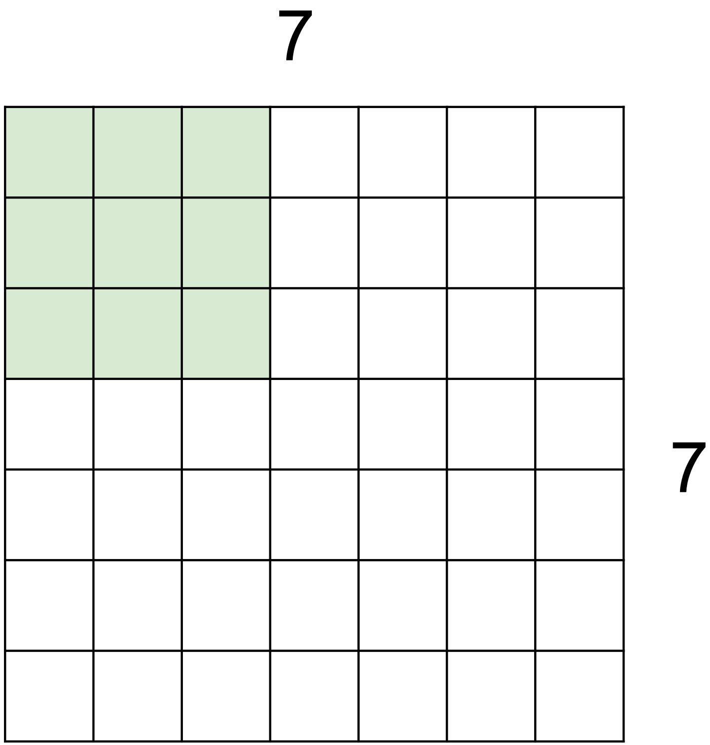
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



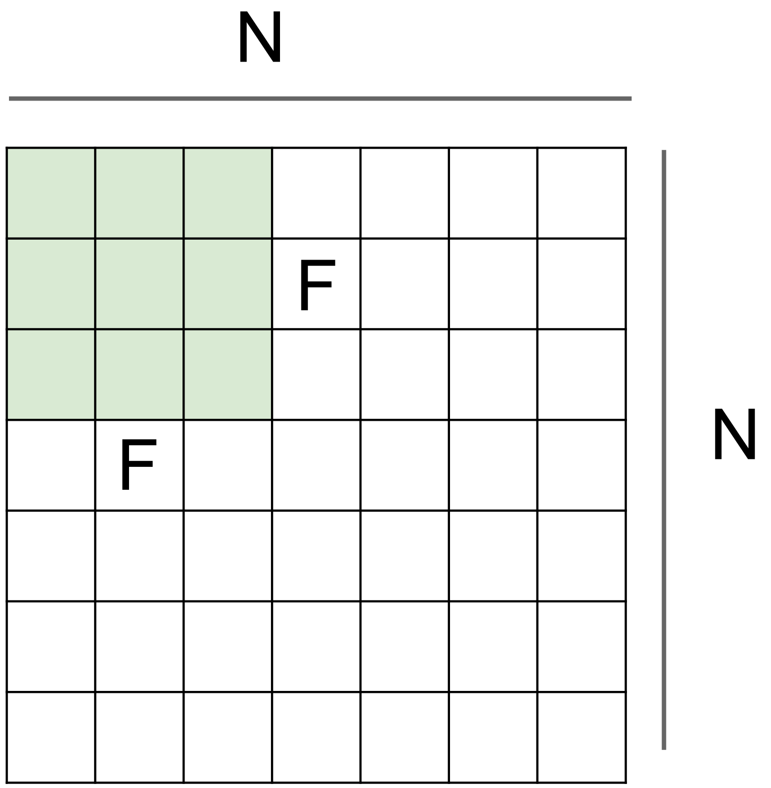
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



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

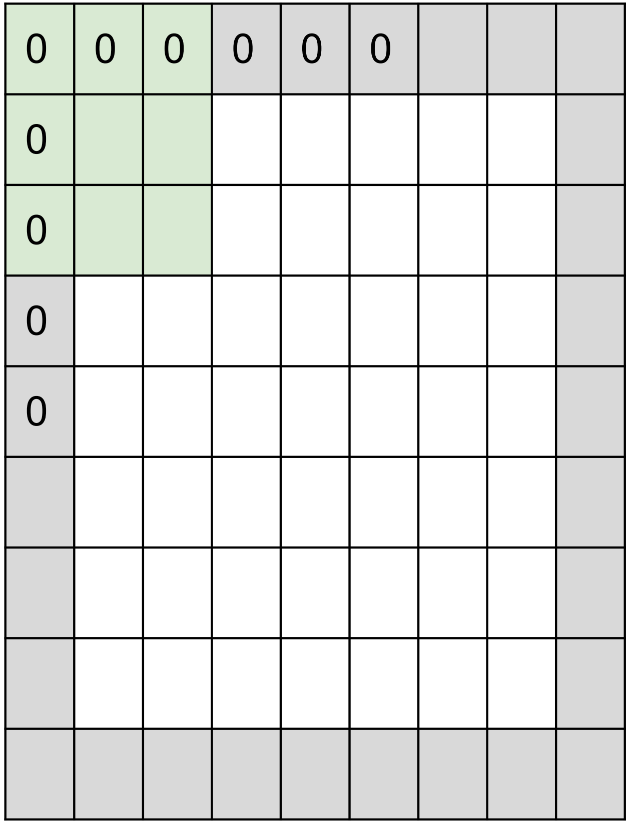
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



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 :\

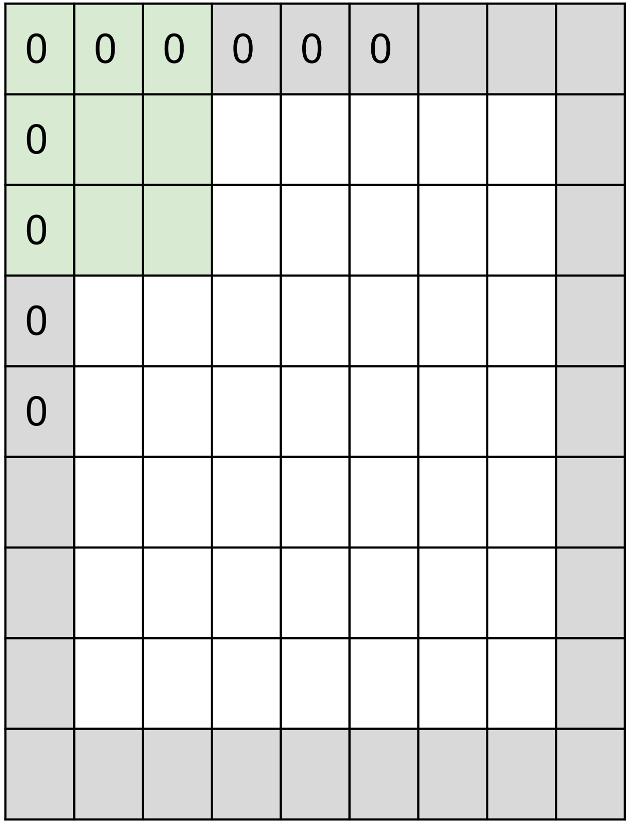
In practice: Common to zero pad the border



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

(recall:)
(N - F) / stride + 1
$(9 - 3)/1 + 1 = 7$

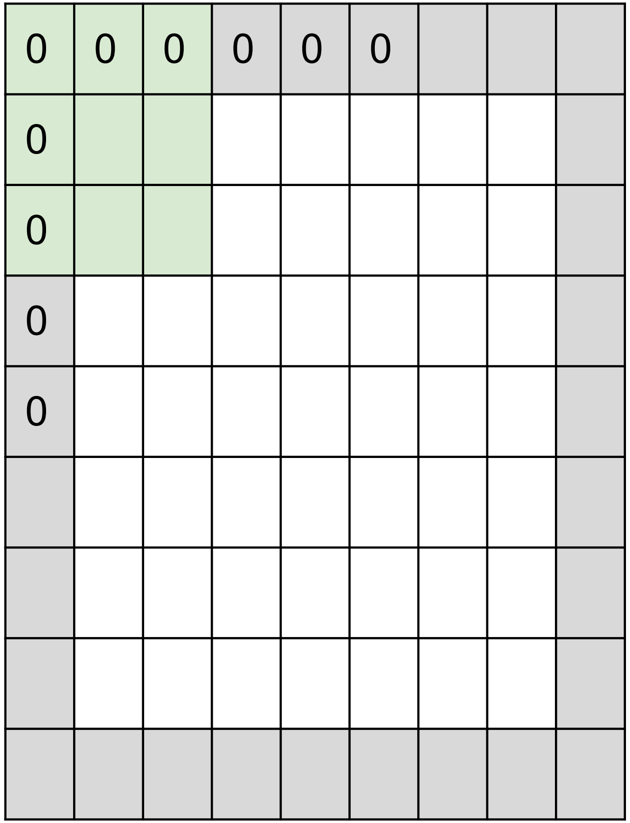
In practice: Common to zero pad the border



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



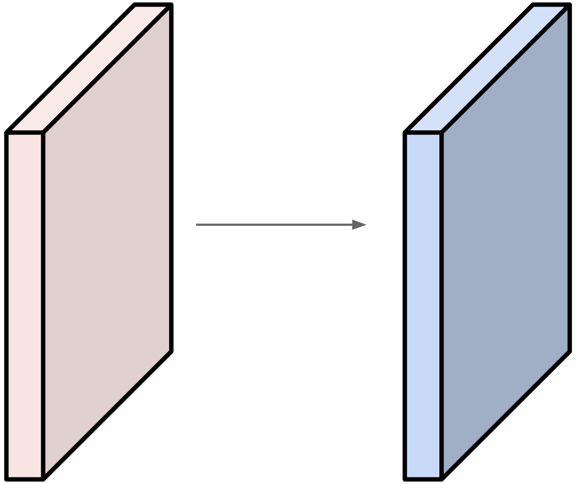
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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

$(N+2P-F)/1 + 1$

$N+F-1-F+1=N$

Examples time:



Input volume: **32x32x3**

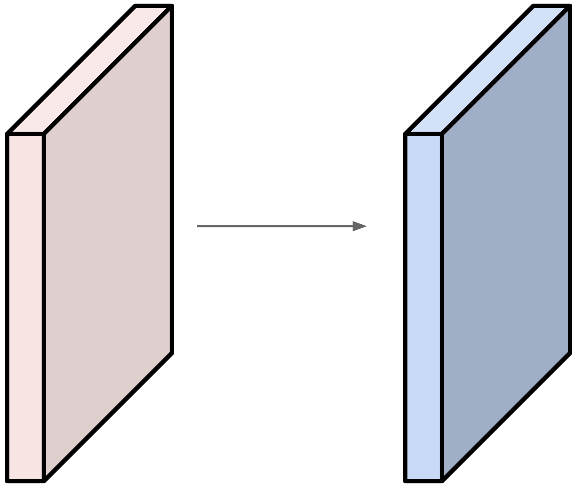
10 5x5 filters with stride 1, pad 2

+3

Output volume size: ?

(N+2P-F)/stride +1

Examples time:



Input volume: **32x32x3**

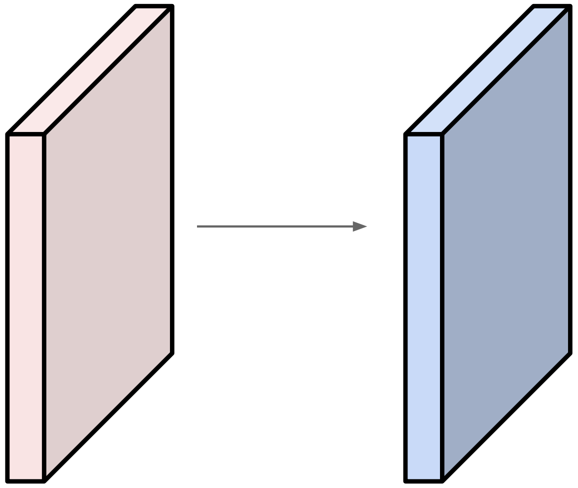
10 5x5 filters with stride 1, pad 2

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

↳ x3

Number of parameters in this layer?

each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

Summary for convolutional 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: $F = 3, S = P = 1$

Demo time

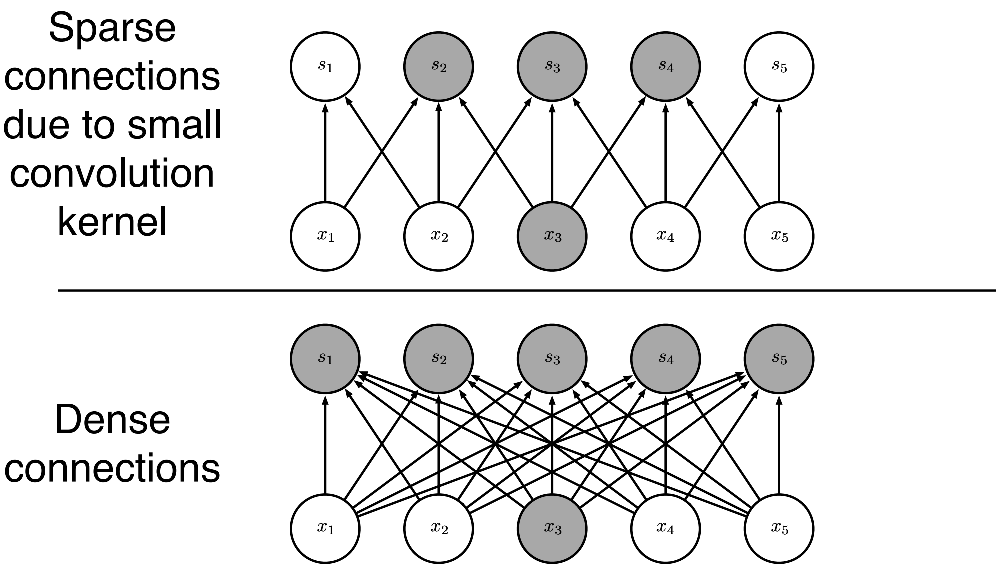
What is a Convolutional Neural Network?

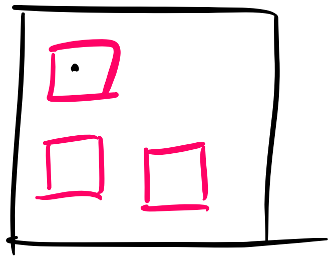
https://poloclub.github.io/cnn-explainer/

Connection to fully connected networks

A convolutional layer is a special case of a fully connected layer:
filter = weights with sparse connection

Local Receptive Field Leads to Sparse Connectivity (affects less)





(Goodfellow 2016)

Figure from Goodfellow'16

Connection to fully connected networks

A convolutional layer is a special case of a fully connected layer:
filter = weights with sparse connection

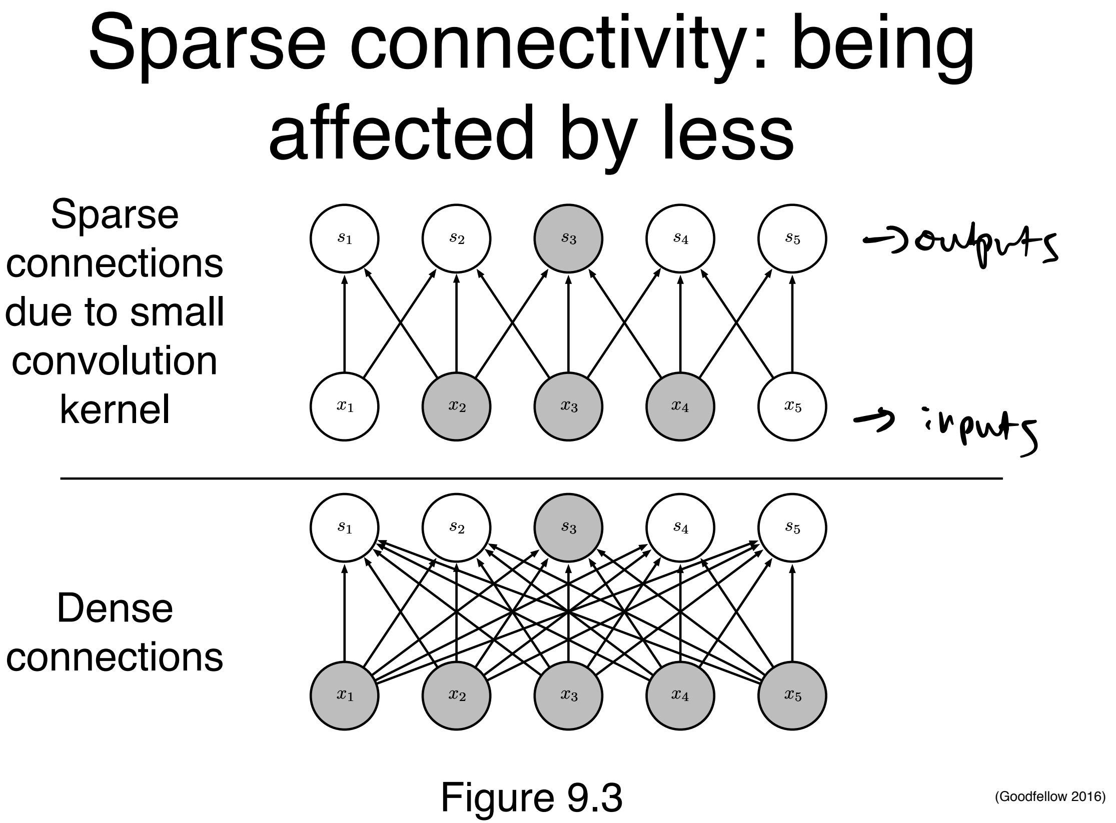


Figure from Goodfellow'16

Connection to fully connected networks

A convolutional layer is a special case of a fully connected layer:
filter = weights with sparse connection and parameter sharing

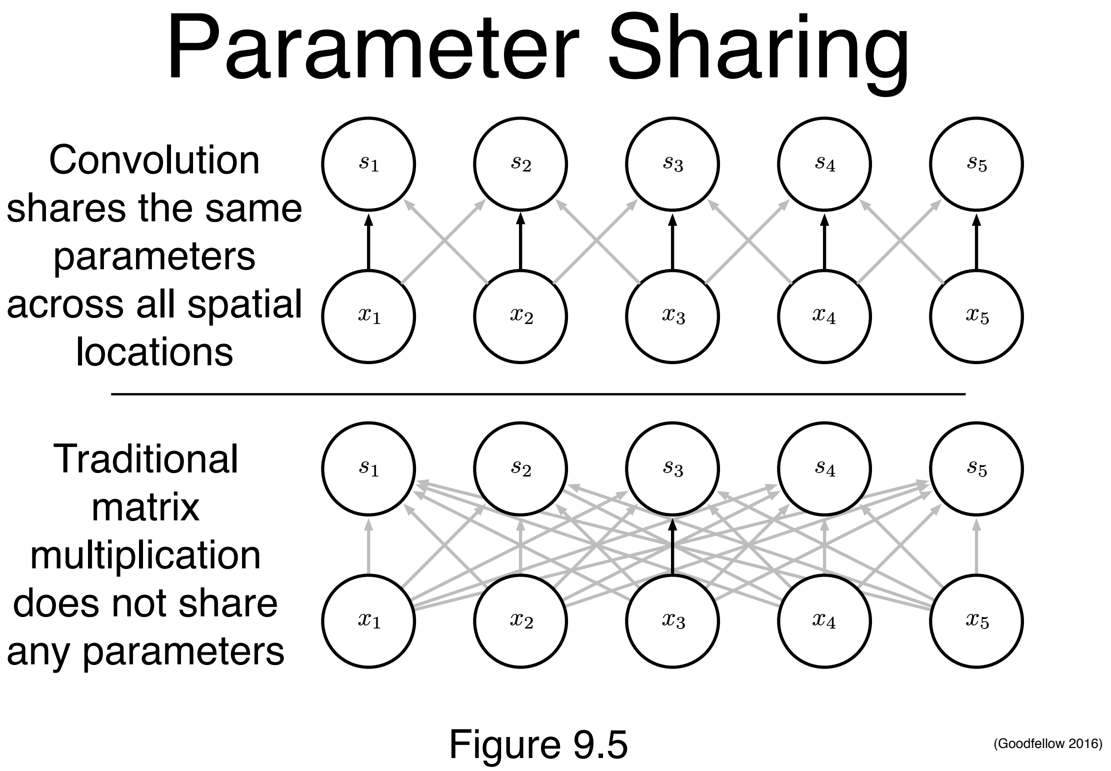


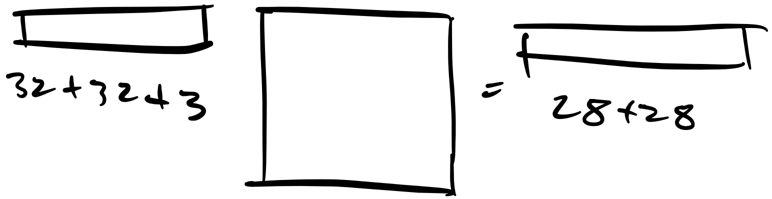
Figure from Goodfellow'16

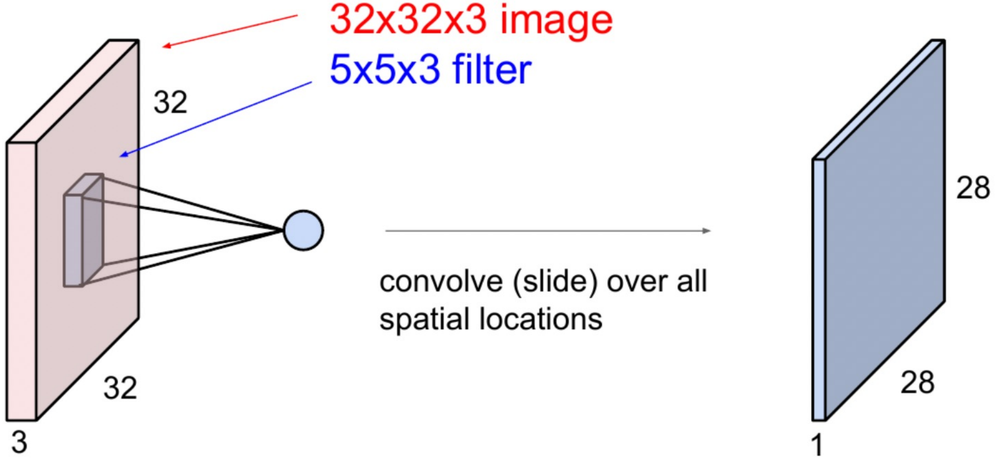
Connection to fully connected networks

A convolutional layer is a special case of a fully connected layer:
filter = weights with sparse connection and parameter sharing

Much fewer parameters! Example (ignoring bias terms):

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

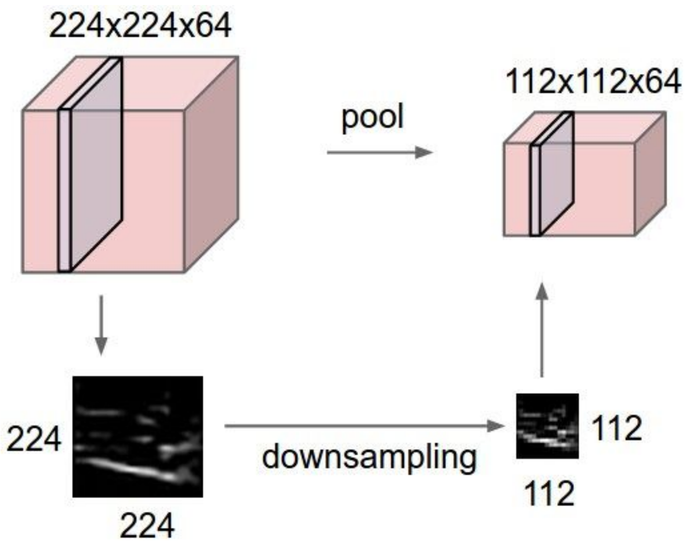




Another element: Pooling

Pooling layer

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



Another element: 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

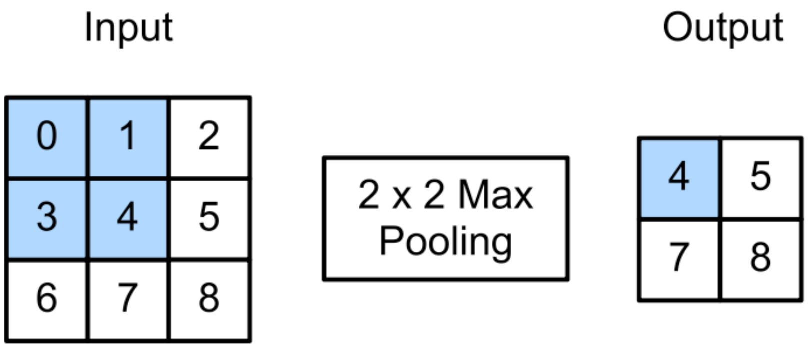


Figure 14.12: Illustration of maxpooling with a 2x2 filter and a stride of 1. Adapted from Figure 6.5.1 of [Zha+20].

Figure from PML

Finishing things up…

Typical architecture for CNNs:

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

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

→ # parameters here is very layer

How do we learn the filters/weights?

Essentially the same as fully connected NNs: apply SGD/backpropagation

Demo time

What is a Convolutional Neural Network?

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

A breakthrough result

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

A breakthrough result

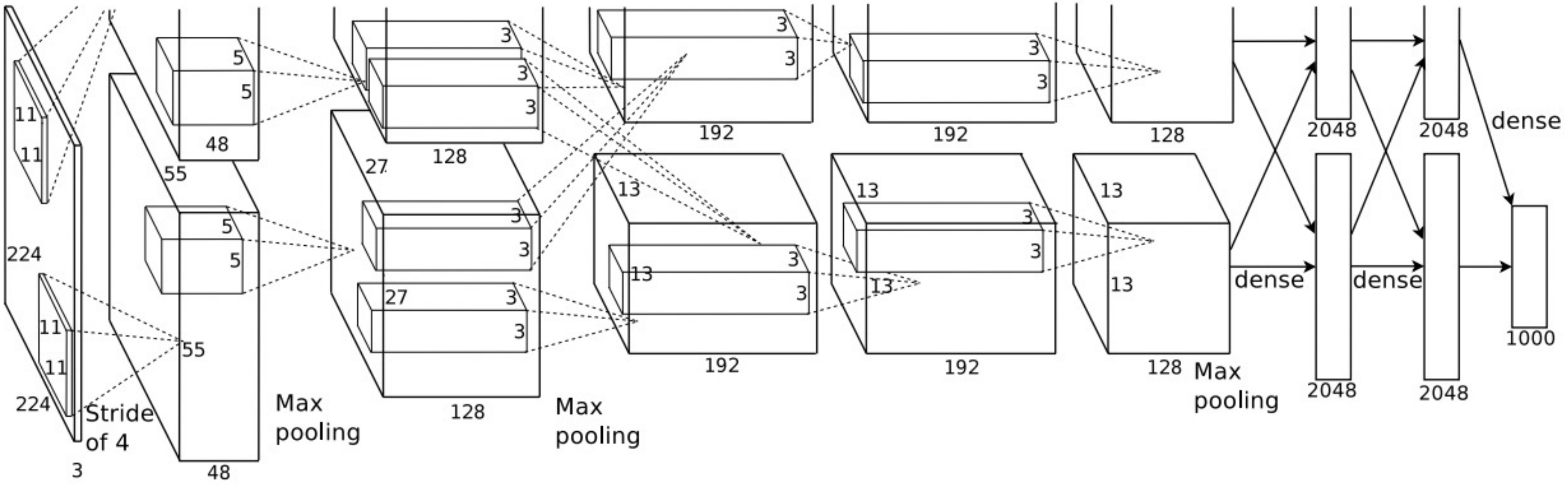


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Figure from Krizhevsky et al.'12