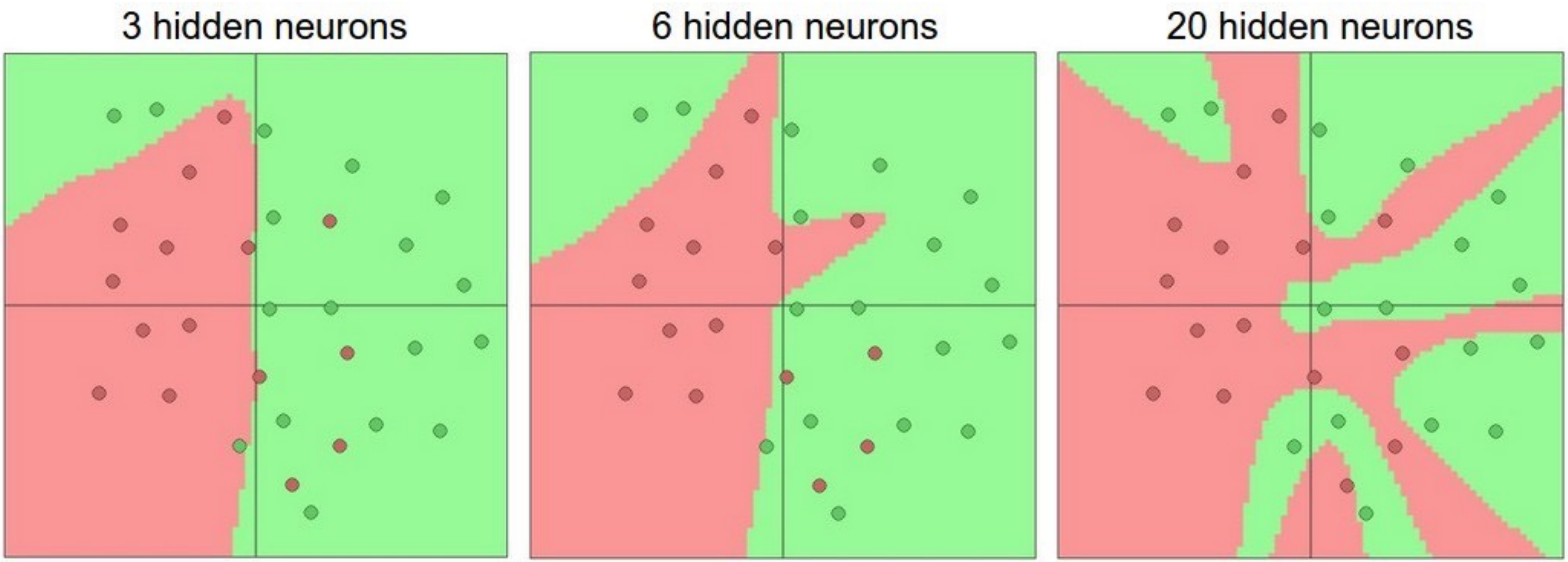
Neural network architectures



Comments

- Reviews for mini-projects due on Friday at midnight
- Clarification about margin bound for boosting, updated in lecture slides

Recall NNs

- Generically: a network of neurons
- Can have many architectures; we have looked at feedforward neural networks
  - directed connections
  - acyclic connections (no connections backwards)
  - neurons composed of dot products with input, and activation function (such as relu, tanh, sigmoid)

Other models

- Convolutional Neural Network
- Recurrent Neural Network
- Long-term Short-term Memory (LSTM)
- Extensions to distributions: integrate over hidden variables

What is the right architecture?

- Fun answer: we don’t know!
- New architectures constantly proposed, hard to track exactly what is performing best
- Many more architectures we could consider
- You get to be in AI during a time where we get to investigate this question

Properties we might want

- What we want: generalize well across many tasks, promote faster learning (either computationally or in terms of samples)

- Properties that might help achieve this
  - **Independence** — do not want correlated features, slows learning
  - **Sparsity** — because it encourages decorrelated features, locality and seems to be better for incremental learning
  - **Invariance** — we may want to be robust to shifting, scaling, rotating or permuting an input
  - **Overcome partial observability** — memory
  - …

Uncorrelated features

- We have seen one way to get uncorrelated features: principal components analysis (PCA)
  - dictionary learning/factorization with small k, least-squares loss
- Start with d features —> generate k new features that are linearly uncorrelated, but capture much of the signal in the data
- Other such strategies: independent component analysis

Sparsity

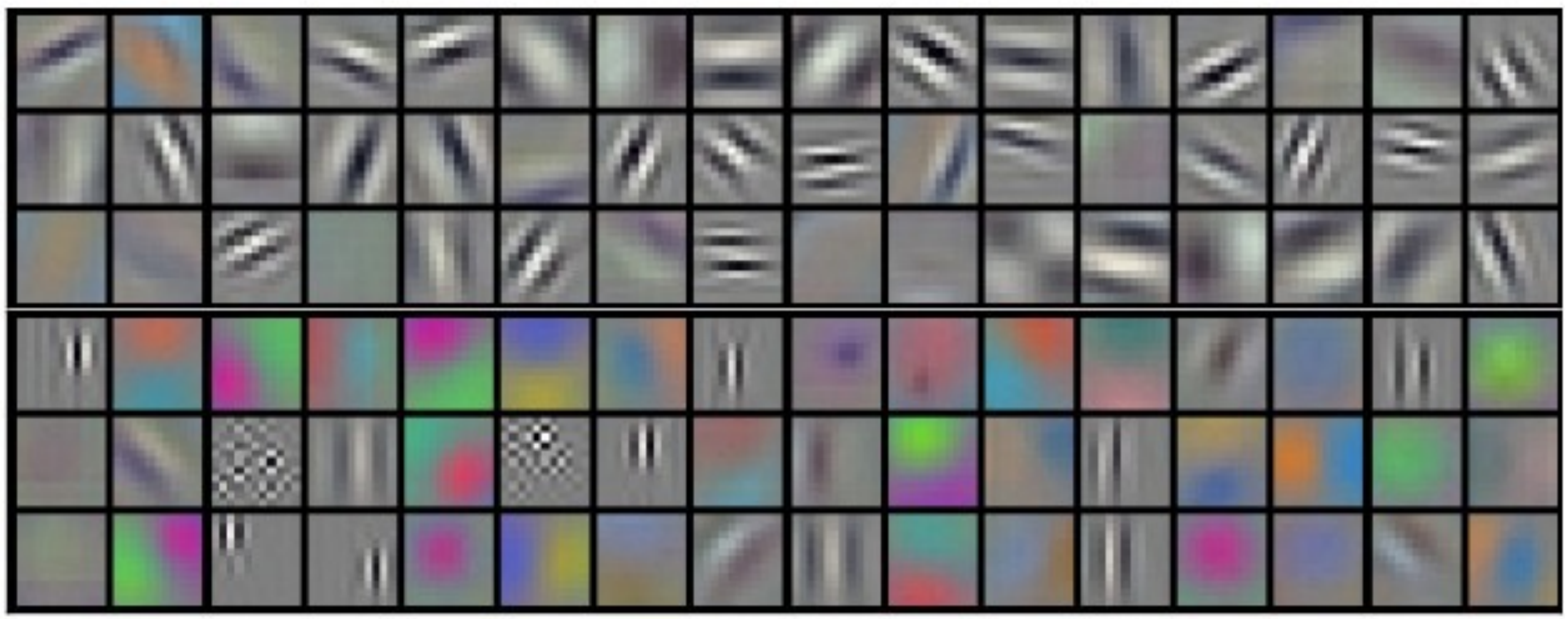
- This is less well-understood, but has some empirical support
- Approaches:
  - sparse coding for dictionary learning
  - sparse layers in neural networks
- Particularly for reinforcement learning sparsity seems key
  - Basis approaches, that capture locality
  - CMAC, i.e., tile coding seems much more stable for incremental learning

Invariance

- Imagine have an image
- Now just want to identify if a sun is in the image or not
  - it is not relevant where it is in the image
- How might we do this?
- Let’s go back to our view of one layer as a filter

One layer can act like a filter

- Dot-product with input x, and a weight vector w, can emphasize or filter parts of x
  - e.g., imagine x is an image, and w is zero everywhere except one small patch in the corner. It will pick out the magnitude of pixels in that small patch



*awesome overview: http://cs231n.github.io/convolutional-networks/

How set w to find an object?

- Consider a fully connected NN
- Each hidden node corresponds to the input image filtered by the weights w —> picks out parts of the image where w is non-zero
- We could define w1, …, wk where w1 has a sun in the top corner (zero elsewhere), w2 has a sun in the top middle (zero elsewhere), …
- If there is a sun in one of these quadrants, the hidden layer will have at least one node significantly activated
- The next layer could simply take the max of these activations (called max pooling)

What if we want to learn this filter?

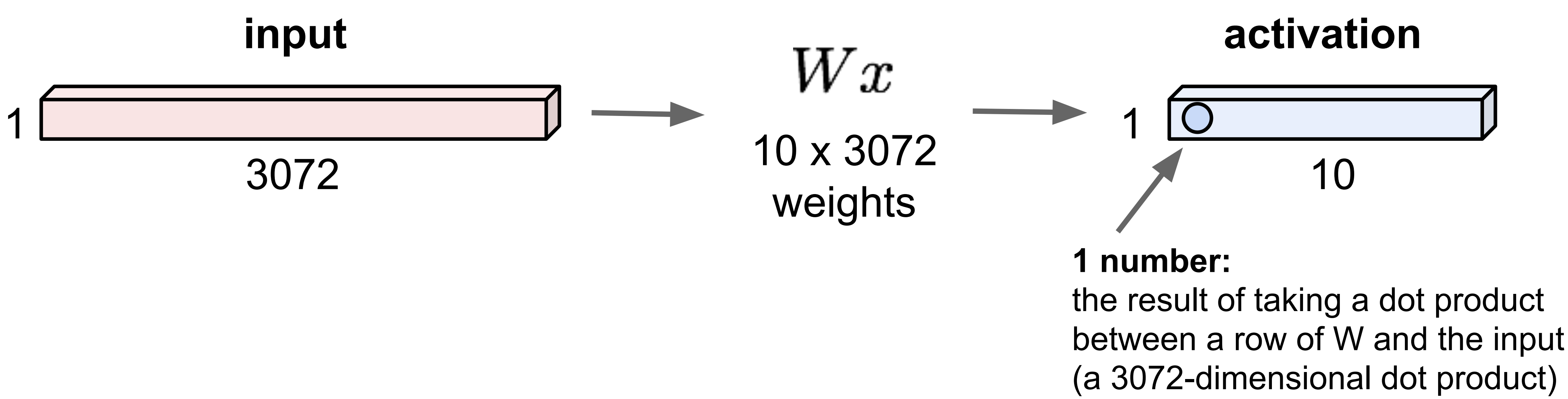
- In the previous example, fixed wi to filter out suns
- What if we want to learn filters, to find objects automatically that are useful for classification (say)?
- We would need to ensure that w1, …, wk only had non-zero values in their respective quadrants AND that they all had the same filter structure to extract the same object
  - That’s a lot of constraints!
- How might we enforce which quadrant each wi filters?

Alternative to constraining equivalent parameters

- Instead, will only learn one filter w, and share it across the quadrants
- This corresponds to the convolution operator
  - can still think of it like a feedforward NN, but with these weights to be constrained to be the same
- Of course, we can have multiple such filters
  - one extracts edges of one sort, another circles, etc.

Example for images: Typical feed-forward NN

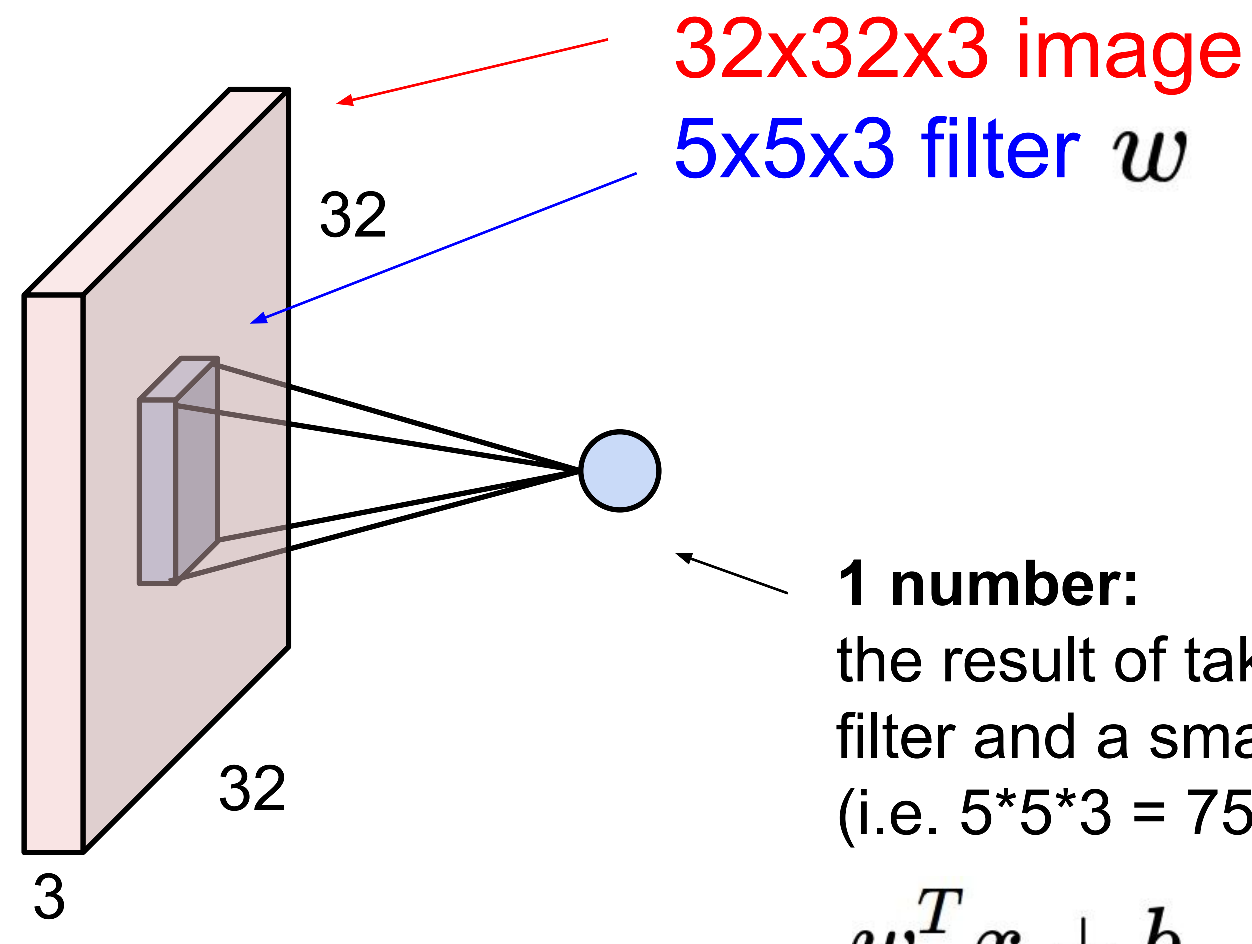
32x32x3 image -> stretch to 3072 x 1



Throws away spatial structure

*from: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf

Example: convolutional layer

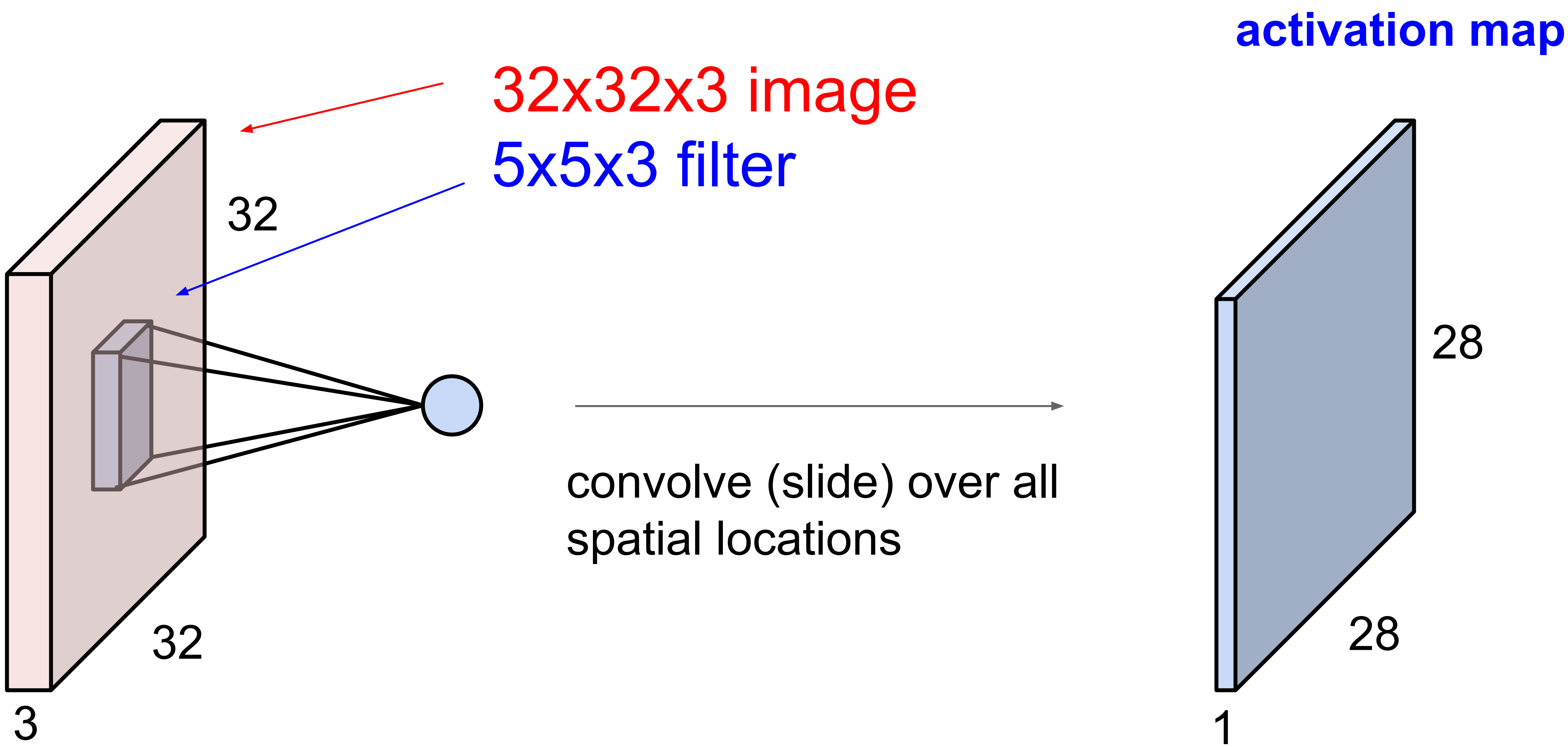


1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Image is 32x32, with 3 RGB colour channels

Output of convolutional layer

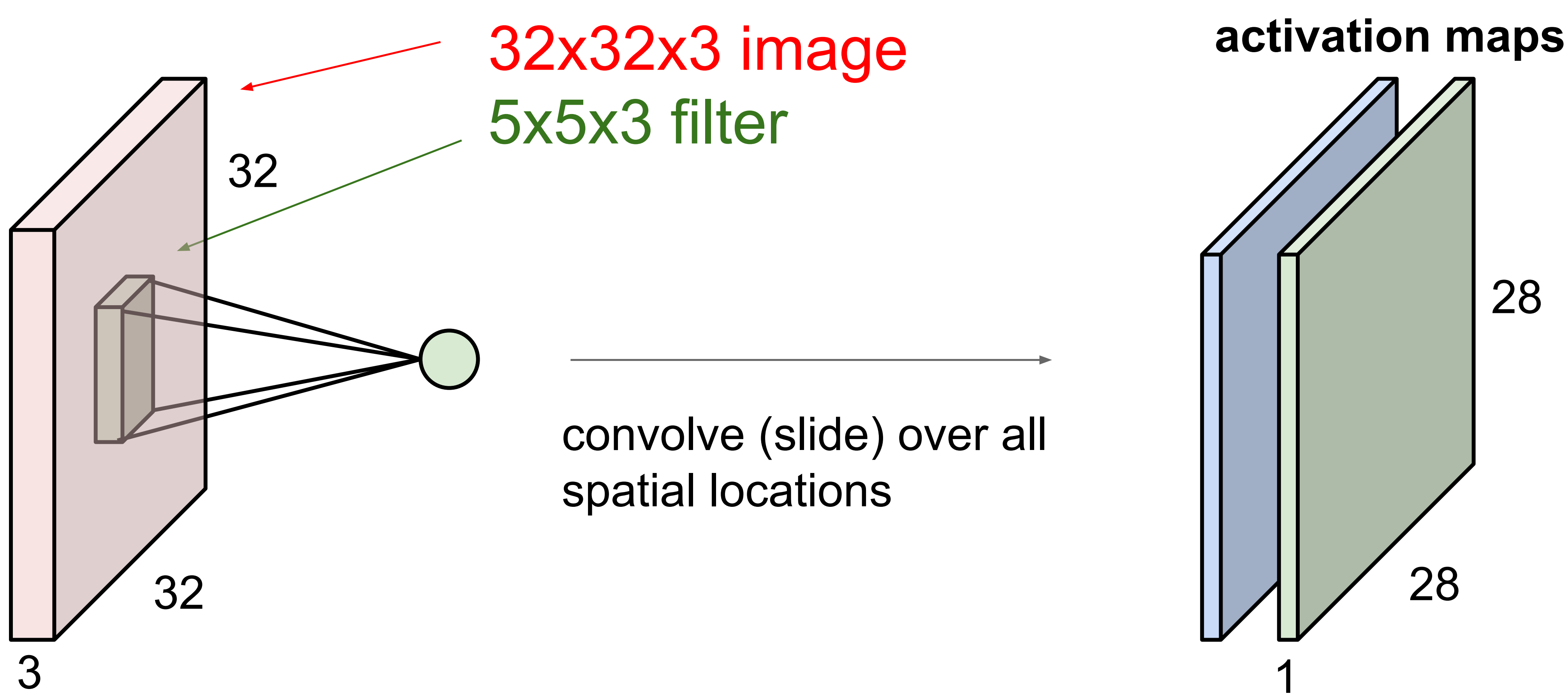


Why 28 x 28 x 1?

Multiple convolutions

Convolution Layer

consider a second, green filter

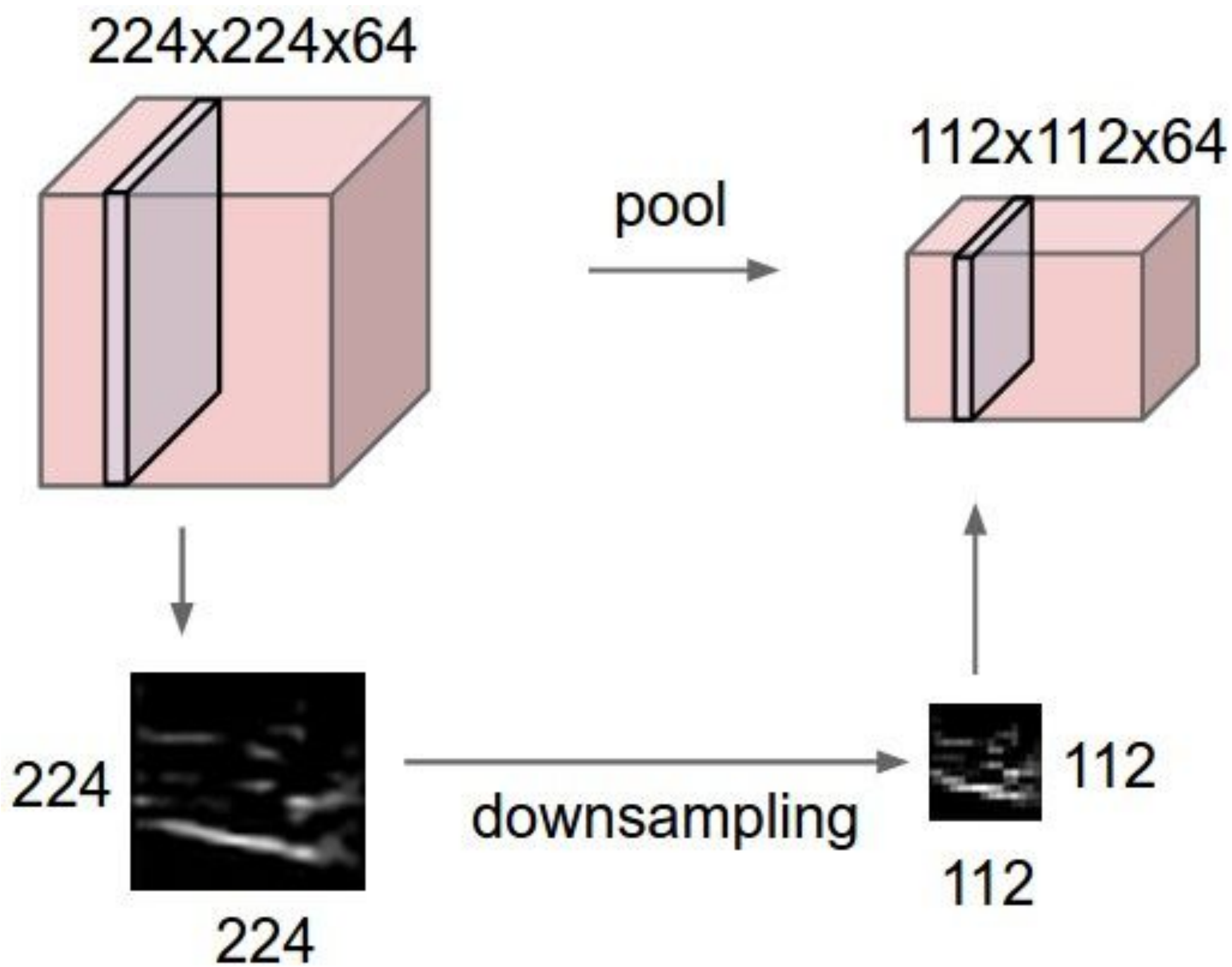


Can also vary **stride**: amount of shift. Here we shift by 1; what if we shift by 3?

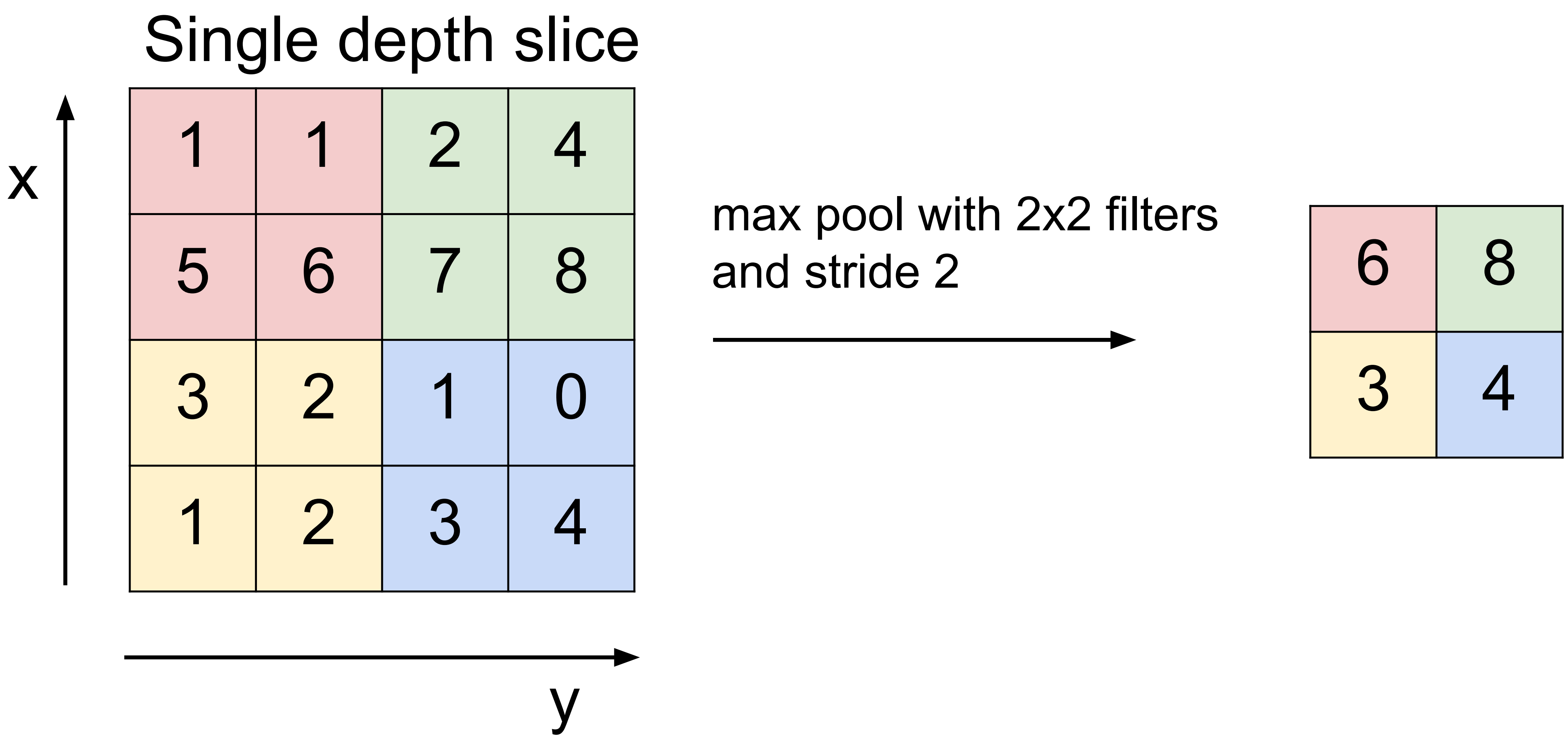
Output size: (32-5)/stride + 1

Pooling layer

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

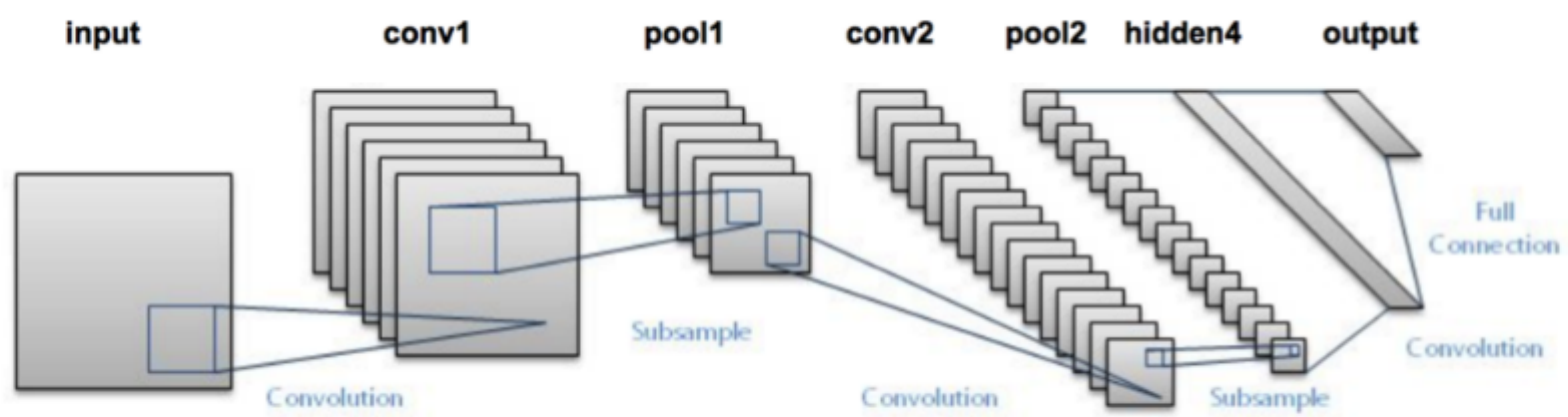


Max pooling

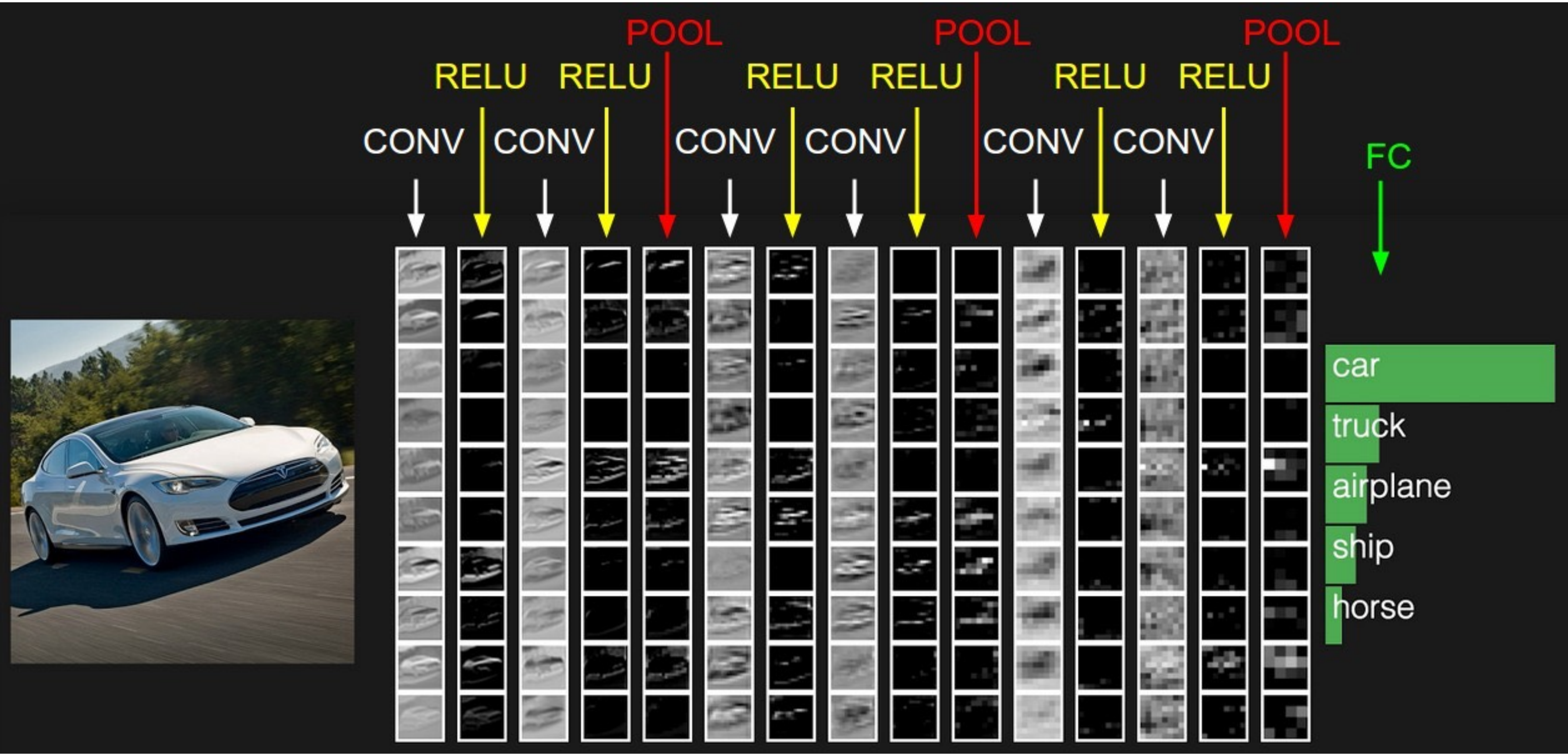


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Example: LeNet for classifying images in ImageNet



Example of hidden layers in a convolutional network



 *awesome overview: http://cs231n.github.io/convolutional-networks/

Other operators

- Average pooling (and other pooling) also used; max pooling more popular right now

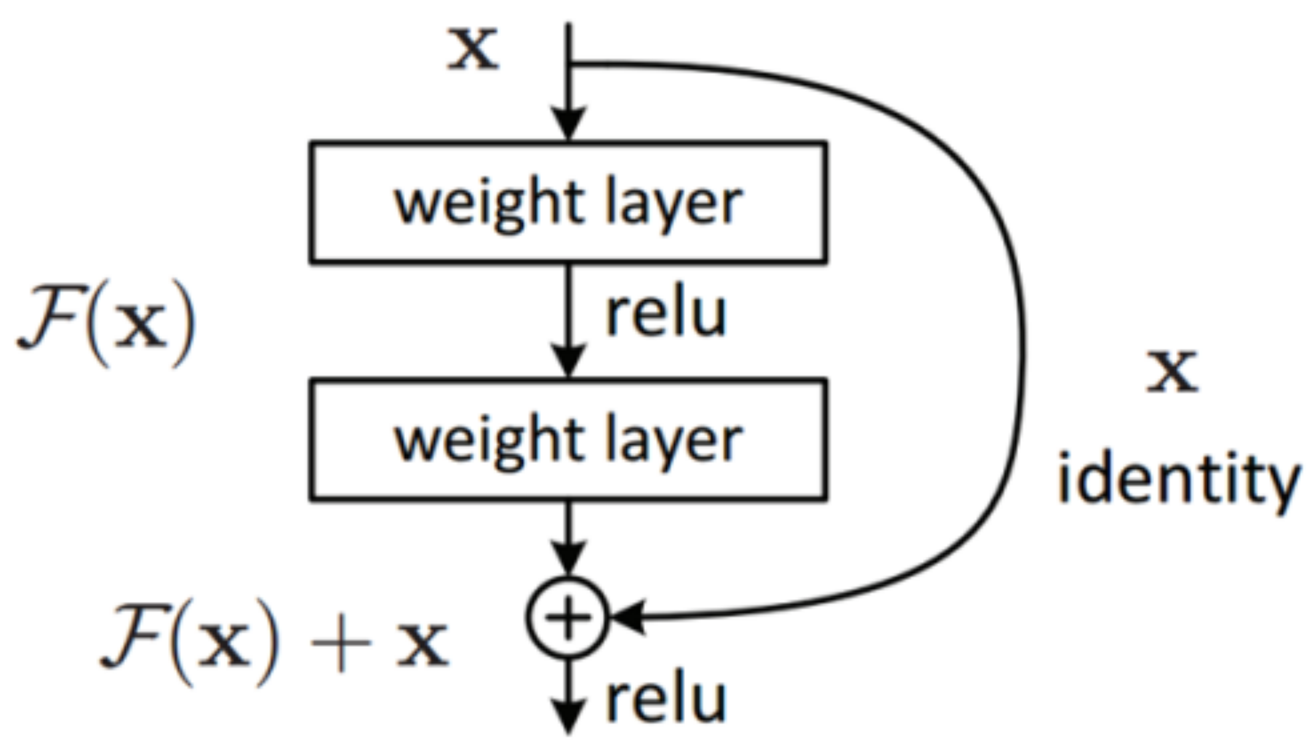
- Different layers components come in and out of favour pretty consistently

Skipping connections

- We have so far assumed that arrows only point into the next layer

- Of course, could have arrows skip layers

- Example: Residual Network



A residual block

How do we optimize through these layers?

- How do we take the derivative for the shared parameters?
- How do we take derivatives when layers are skipped?
- How do we take the derivative through the max pooling operation?
  - Any issues here?
  - How might we select the stepsize?

Exercise: approximate Hessian

- Let’s take a diagonal approximation to the Hessian, to give us a vector of stepsizes
  - this is essentially what adagrad and adadelta are doing
- max is non-differentiable; does this give issues with the Hessian?

Thought exercise: Why GD?

- We use gradient descent alot; aren't there alternatives?
- Our goal: maximize (or minimize) function f
- Depending on properties of f, different optimization strategies
  - Naive (but general) option: guess a bunch of solutions
  - Combinatorial optimization problems (e.g., find best subset of items); could try to formulate as submodular maximization problem
  - If add constraints, could formulate as linear programs, quadratic programs, semi-definite programs
  - Black-box approaches with fewer guarantees, e.g. genetic algorithms
  - If know function is differentiable, gradient-based approaches provide a lot of search information
  - Many, many GD approaches, e.g., BGD, SGD, second-order, conjugate gradient, ADMM, block coordinate descent,

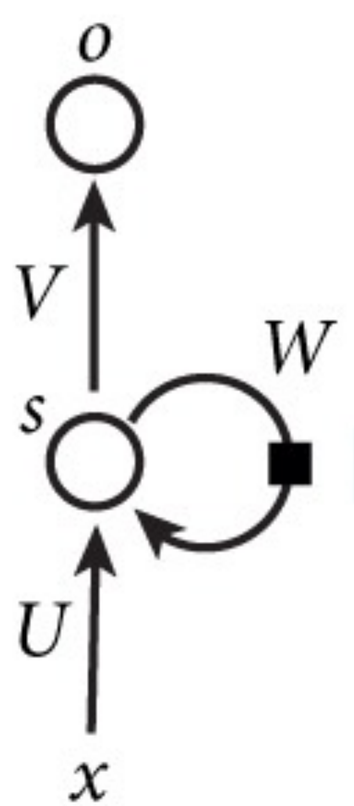
Temporal data

- So far have looked only at i.i.d. data
- Time series (temporal): sequence of temporally connected samples: x1, x2, …., xt
  - e.g., weather
- Common strategy: use a history of p points (lag p)
  - Intuitively, what happened in most recent p steps, should be predictive of what will happen next
  - Cold for the last 5 days suggests it will be cold tomorrow

Simplest strategy

- Create a new dataset, where targets are x_i and features for that target are $[x_{i-1}, \ldots, x_{i-p}]$
- Use your favourite supervised learning algorithm
  - learning conditional distribution $p(x_i \mid x_{i-1}, \ldots, x_{i-p})$
- Two problems:
  - what if chose p too small?
  - now have to learn p weights for each vector x_i
- A more compact approach might be to learn a hidden state

Recurrent Neural Network



- Time steps not obvious
- Recurrent connection means previous state inputted for the next step
- Output (target) is a function of this hidden state

$$s_t = f(s_{t-1}, x_t)$$

*image from Nature

Recurrent Neural Network

How do we learn the weights w?

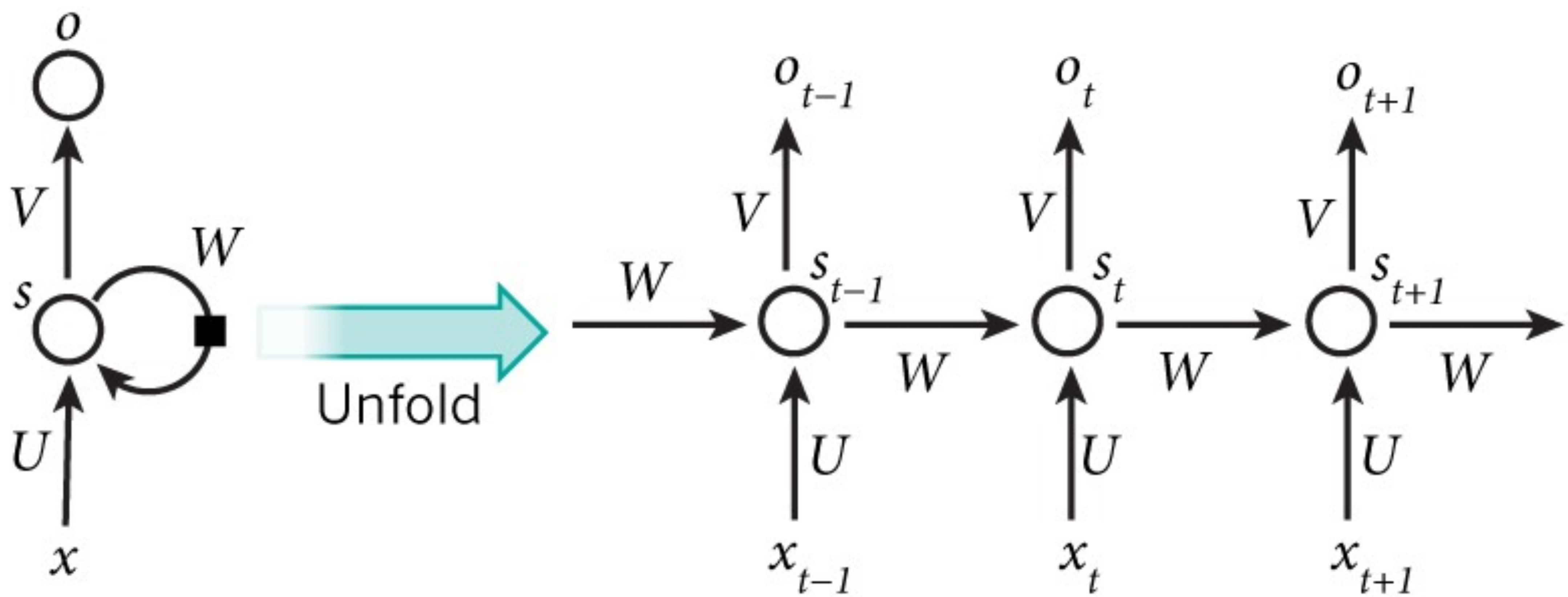


*image from Nature

How long is the memory?

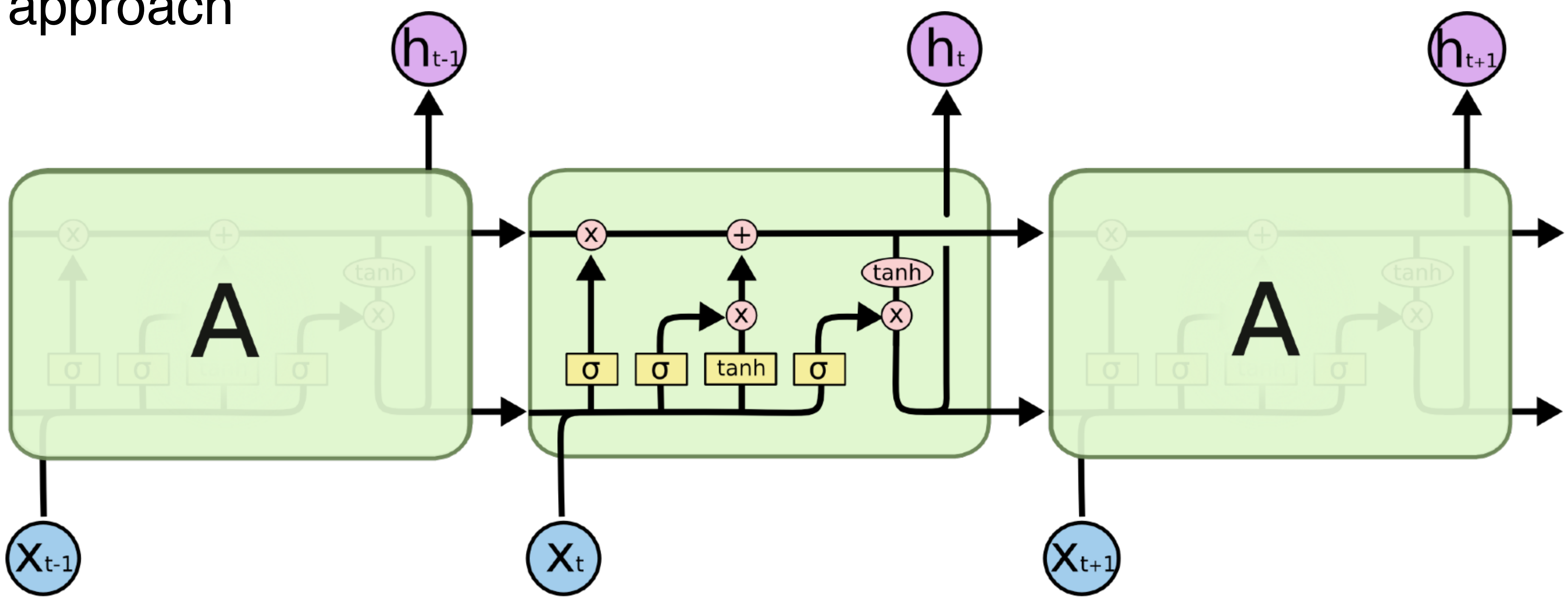
- Can unroll all the way back in time

$$
\begin{aligned}
s_t &= f(s_{t-1}, x_t) \\
&= f(f(s_{t-2}, x_{t-1}), x_t) \\
&= f(f(f(s_{t-3}, x_{t-2}), x_{t-1}), x_t) \\
&= \ldots
\end{aligned}
$$

- Technically a function of infinite lag back in time
- Practically (numerically) dependence drops off
- Gradients back in time stopped after some fixed lag p

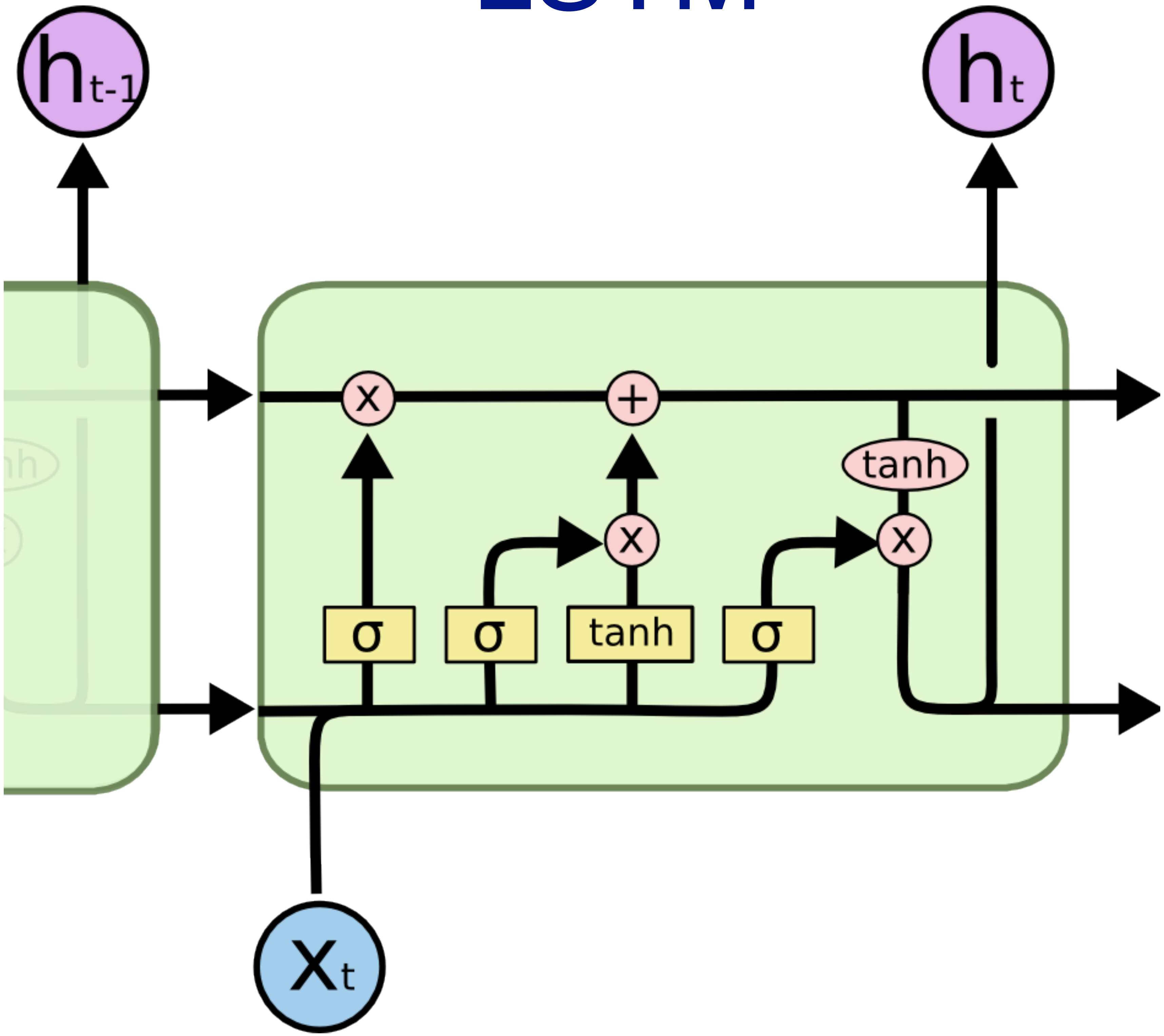
Selective memory

- In addition to keeping (even if implicitly) all observations back in time, could select what to store
- Long-term Short-term Memory (LSTM) architectures one such approach



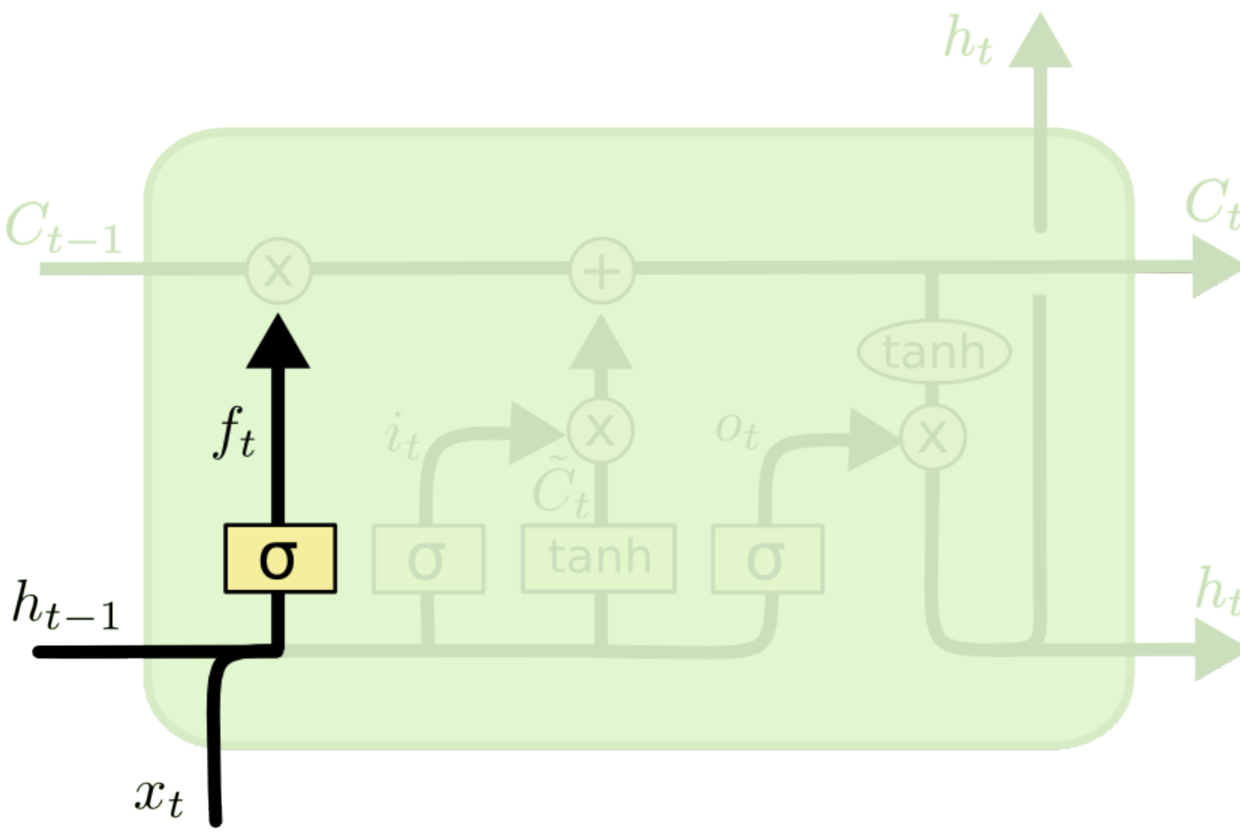
* http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



* http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Example: forgetting



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$