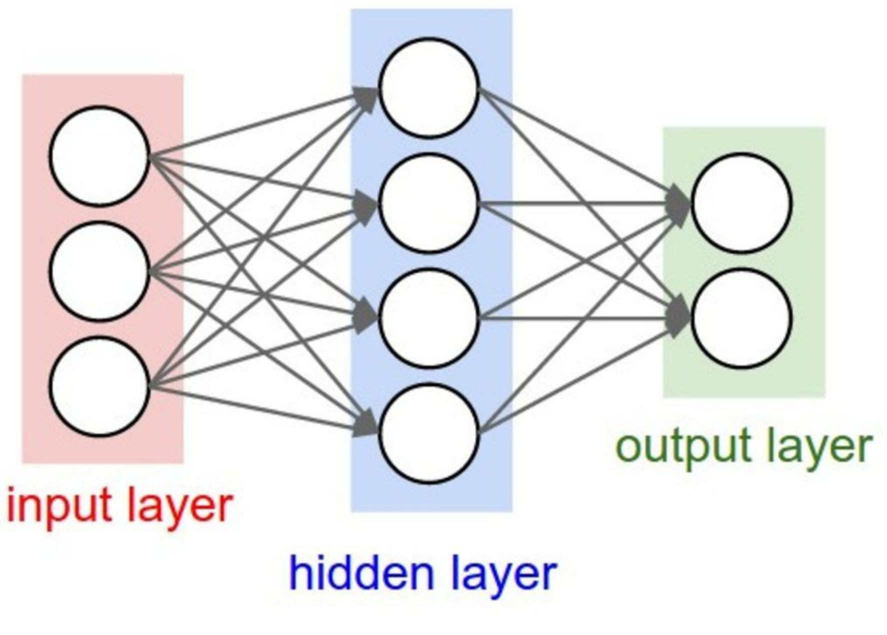
Section 3: Neural Networks

Outline

- Neural Network basics
- Backpropagation
- CNNs
- RNNs
- Transformers

Neural Network (NN) Basics



Dataset: **(x, y)** where **x**: inputs, **y**: labels

Steps to train a 1-hidden layer NN:

- Do a forward pass: **ŷ = f(xW + b)**
- Compute loss: **loss(y, ŷ)**
- Compute gradients using **backprop**
- Update weights using an **optimization** algorithm, like **SGD**
- Do **hyperparameter tuning** on **Dev** set
- **Evaluate** on **Test** set

Activation Functions: Sigmoid

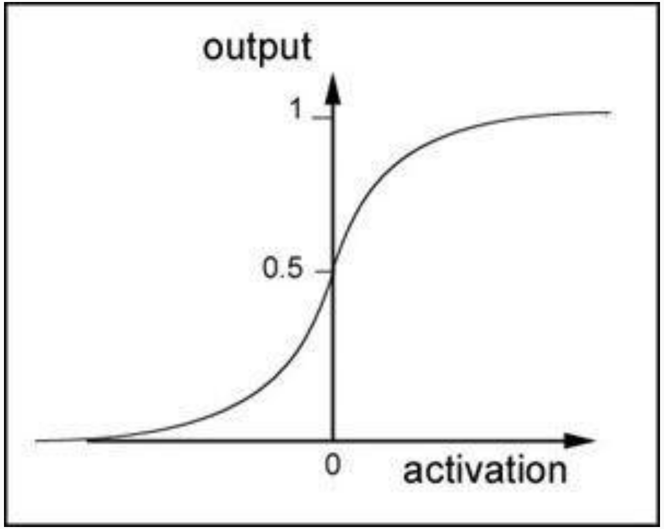
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Properties:

- Squashes input between 0 and 1.

Problems:

- Saturation of neurons kills gradients.
- Output is not centered at 0.



Activation Functions: Tanh

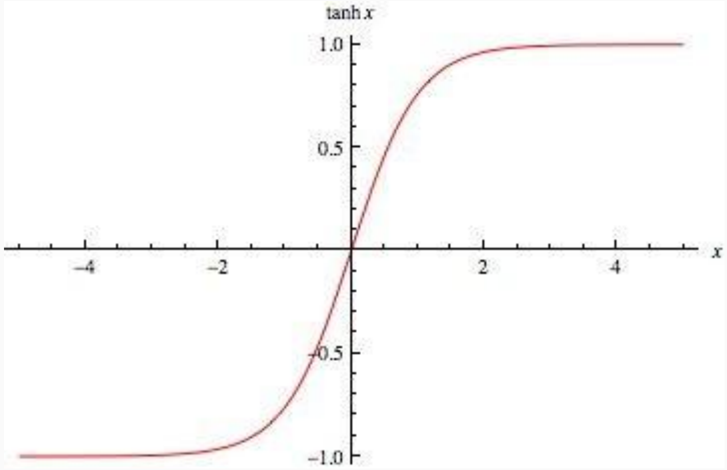
$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Properties:

- Squashes input between -1 and 1.
- Output centered at 0.

Problems:

- Saturation of neurons kills gradients.



Activation Functions: ReLU

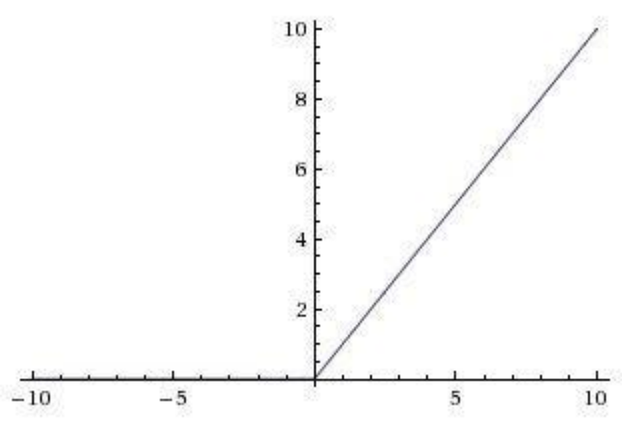
$$relu(x) = max(0, x)$$

Properties:

- No saturation
- Computationally cheap
- Empirically known to converge faster

Problems:

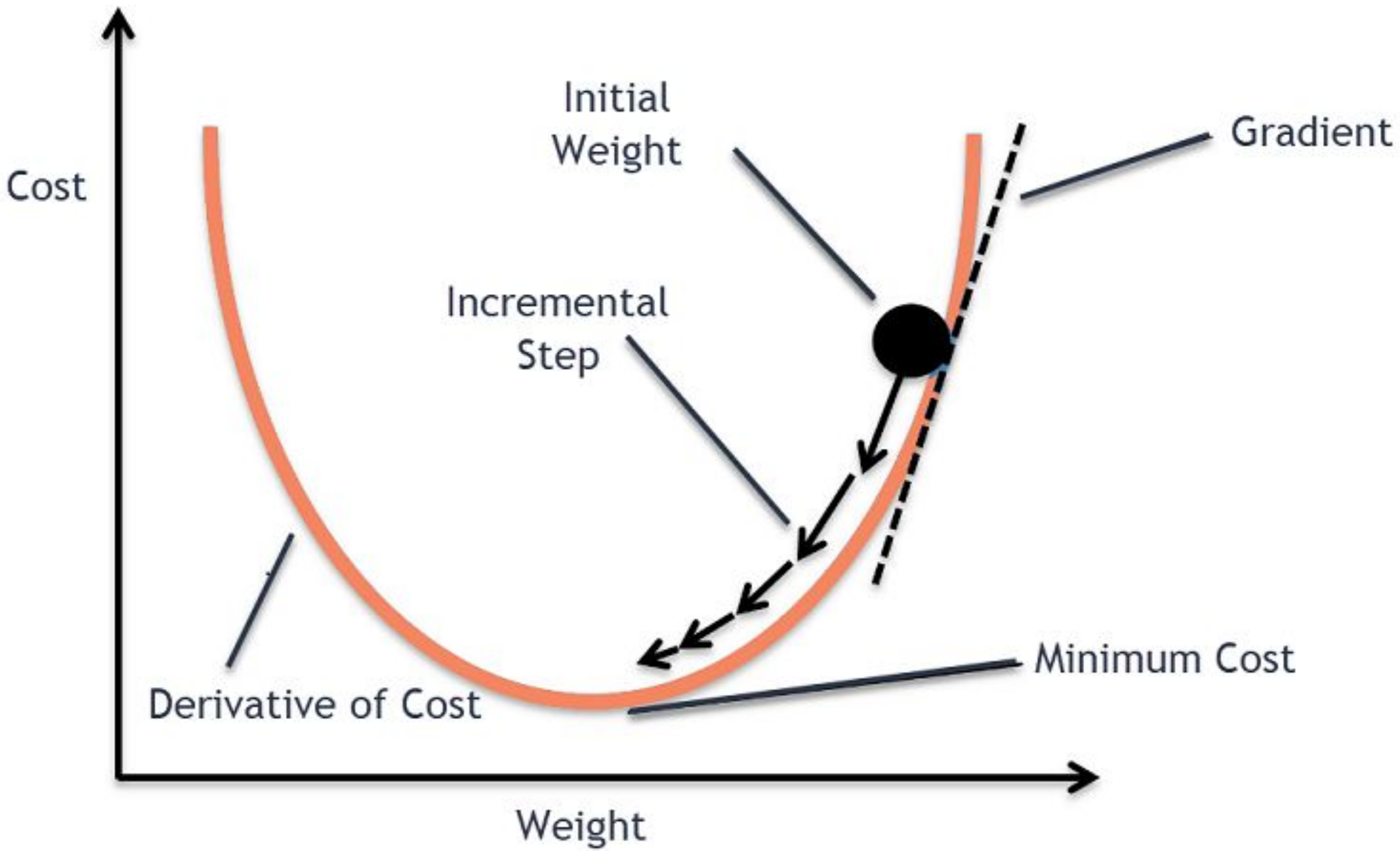
- Output not centered at 0
  When input < 0, ReLU gradient is 0.
  Never changes.



Stochastic Gradient Descent (SGD)

$$\theta \leftarrow \theta - \alpha \nabla_\theta J$$

- Stochastic Gradient Descent (SGD)
  - θ : weights/parameters
  - α : learning rate
  - J : loss function
- SGD update happens after every training example.
- Minibatch SGD (sometimes also abbreviated as SGD) considers a small batch of training examples at once, averages their loss, and updates θ.


Cost
Initial
Weight
Gradient
Incremental
Step
Minimum Cost
Derivative of Cost
Weight

Backpropagation

- Problem statement
- Simple example

Problem Statement

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

Given a function ***f*** with respect to inputs ***x***, labels ***y***, and parameters θ

compute the gradient of ***Loss*** with respect to θ

Backpropagation

$$Loss = ((\sigma(xW_1 + b_1)W_2 + b_2) - y)^2$$

An algorithm for computing the gradient of a **compound** function as a series of **local, intermediate gradients**

Backpropagation

$$Loss = ((\sigma(xW_1 + b_1)W_2 + b_2) - y)^2$$

1. Identify intermediate functions (forward prop)
2. Compute local gradients
3. Combine with upstream error signal to get full gradient

Modularity - Simple Example

Compound function

$$f(x, y, z) = (x + y)z$$

Intermediate Variables
(forward propagation)

$$q = x + y$$

$$f = qz$$

Modularity - Neural Network Example

Compound function

$$Loss = ((\sigma(xW_1 + b_1)W_2 + b_2) - y)^2$$

Intermediate Variables
(forward propagation)

$$h_1 = xW_1 + b_1$$

$$z_1 = \sigma(h_1)$$

$$z_2 = z_1W_2 + b_2$$

$$Loss = (z_2 - y)^2$$

Intermediate **Variables**

(forward propagation)

$$h_1 = xW_1 + b_1$$

$$z_1 = \sigma(h_1)$$

$$z_2 = z_1 W_2 + b_2$$

$$Loss = (z_2 - y)^2$$

Intermediate **Gradients**

(backward propagation)

$$\frac{\partial h_1}{\partial x} = W_1^T$$

$$\frac{\partial z_1}{\partial h_1} = \sigma'(h_1) = z_1 \circ (1 - z_1)$$

$$\frac{\partial z_2}{\partial z_1} = W_2^\top$$

$$\frac{\partial Loss}{\partial z_2} = 2(z_2 - y)$$

Chain Rule Behavior

$$\frac{d((f \circ g)(x))}{dx} = \frac{d(f(g(x)))}{d(g(x))}\frac{d(g(x))}{dx}$$

Key chain rule intuition:

Slopes multiply

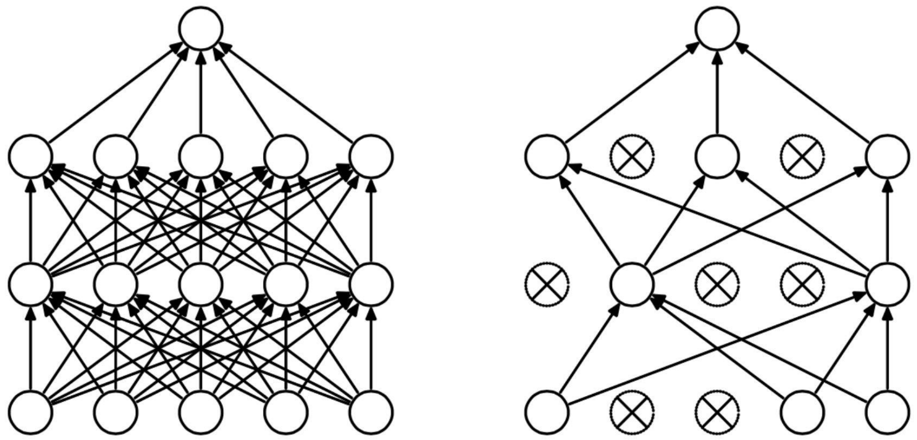
Backprop Menu for Success

1. Write down variable graph
2. Compute derivative of cost function
3. Keep track of error signals
4. Enforce shape rule on error signals
5. Use matrix balancing when deriving over a linear transformation

```
loss.backward()
```

Regularization: Dropout

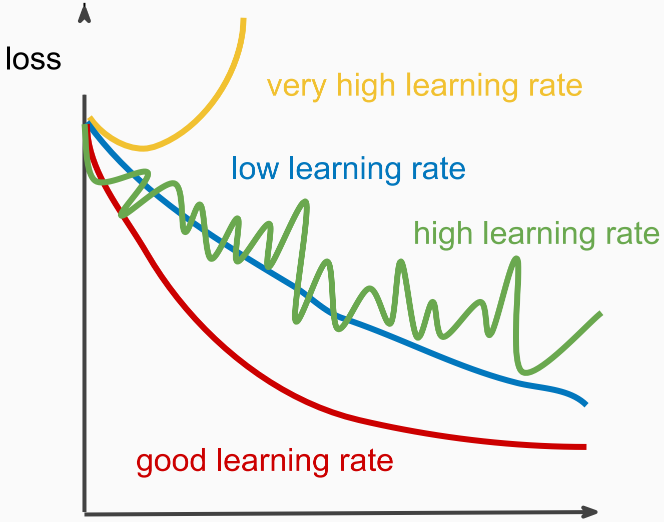
- **Randomly drop neurons** at forward pass during training.
- **At test time, turn dropout off.** **Prevents overfitting** by forcing network to learn redundancies.
- Think about dropout as **training an ensemble of networks.**



(a) Standard Neural Net (b) After applying dropout.

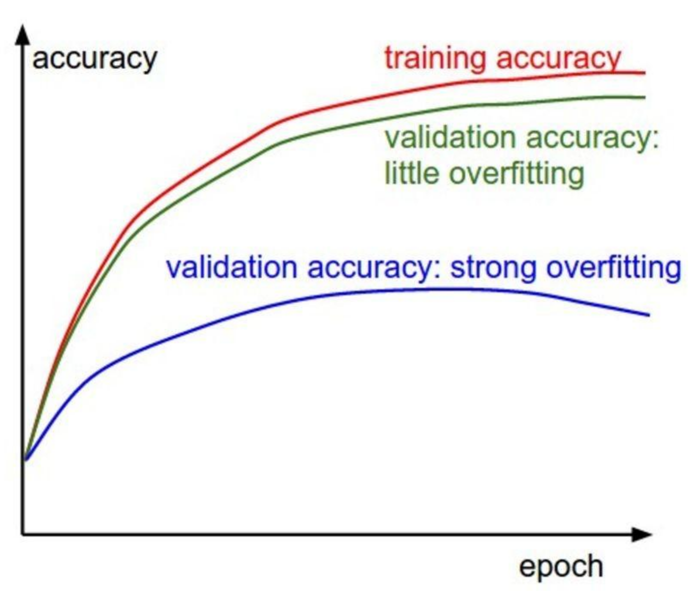
Training Tips and Tricks

- Learning rate:
  - If loss curve seems to be unstable (jagged line), **decrease** learning rate.
  - If loss curve appears to be “linear”, **increase** learning rate.



Training Tips and Tricks

- Regularization (Dropout, L2 Norm, …):
  If the gap between train and dev accuracies is large (overfitting), **increase** the regularization constant.



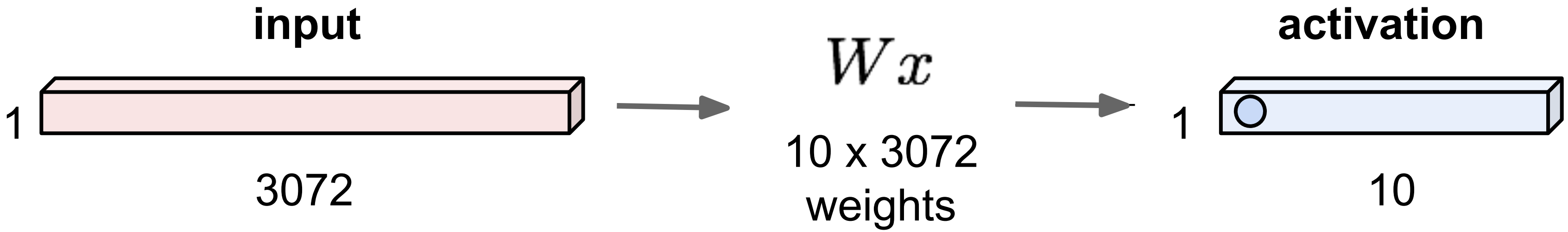
DO NOT test your model on the **test** set until overfitting is no longer an issue.

CNNs

- Fully-Connected Layers
- Convolutional Layers

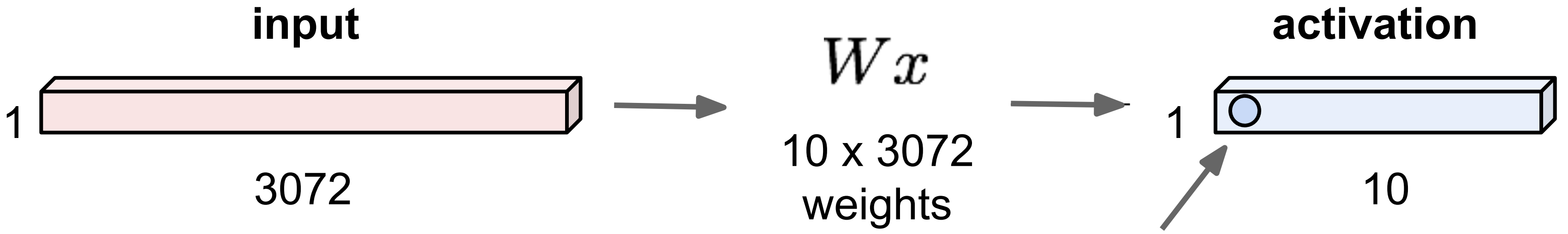
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



Fully Connected Layer

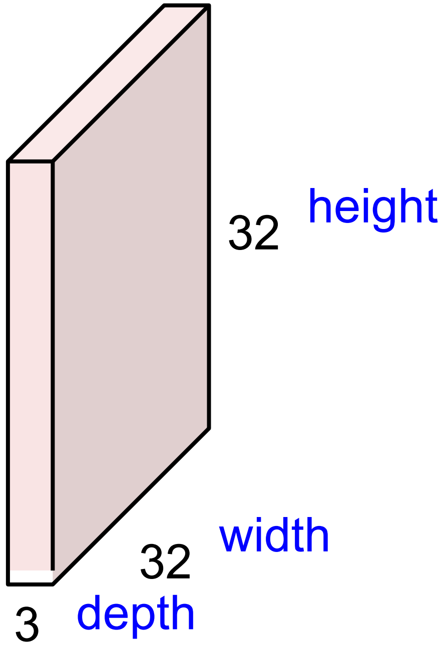
32x32x3 image -> stretch to 3072 x 1



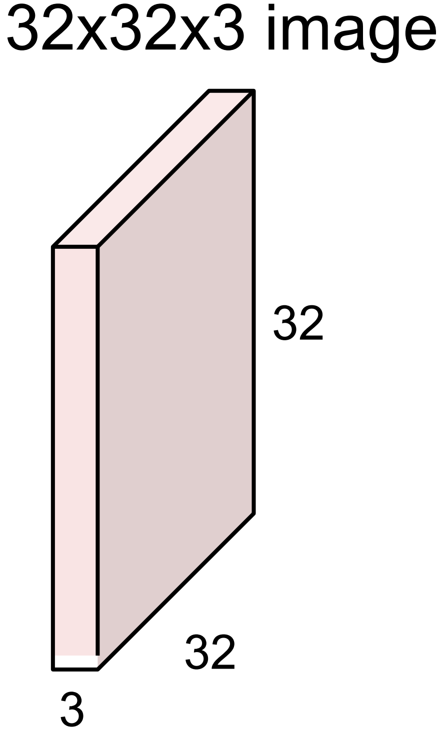
1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

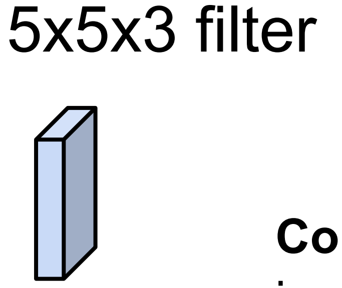
Convolution Layer

32x32x3 image -> preserve spatial structure



Convolution Layer



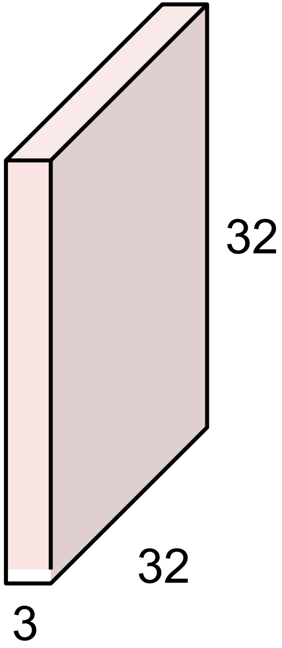


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

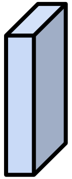
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

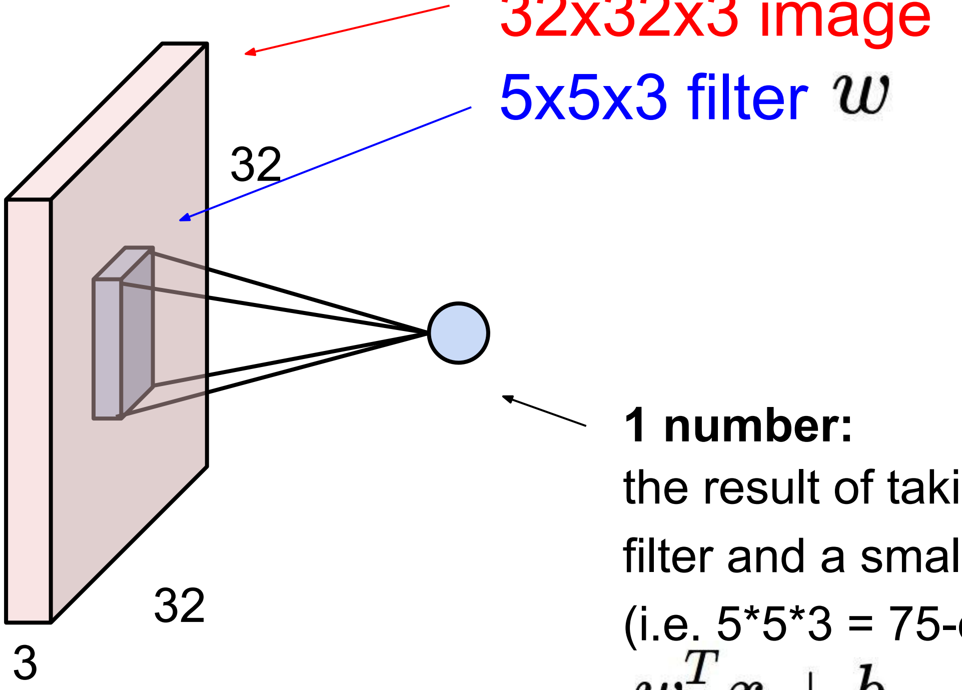


5x5x3 filter



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

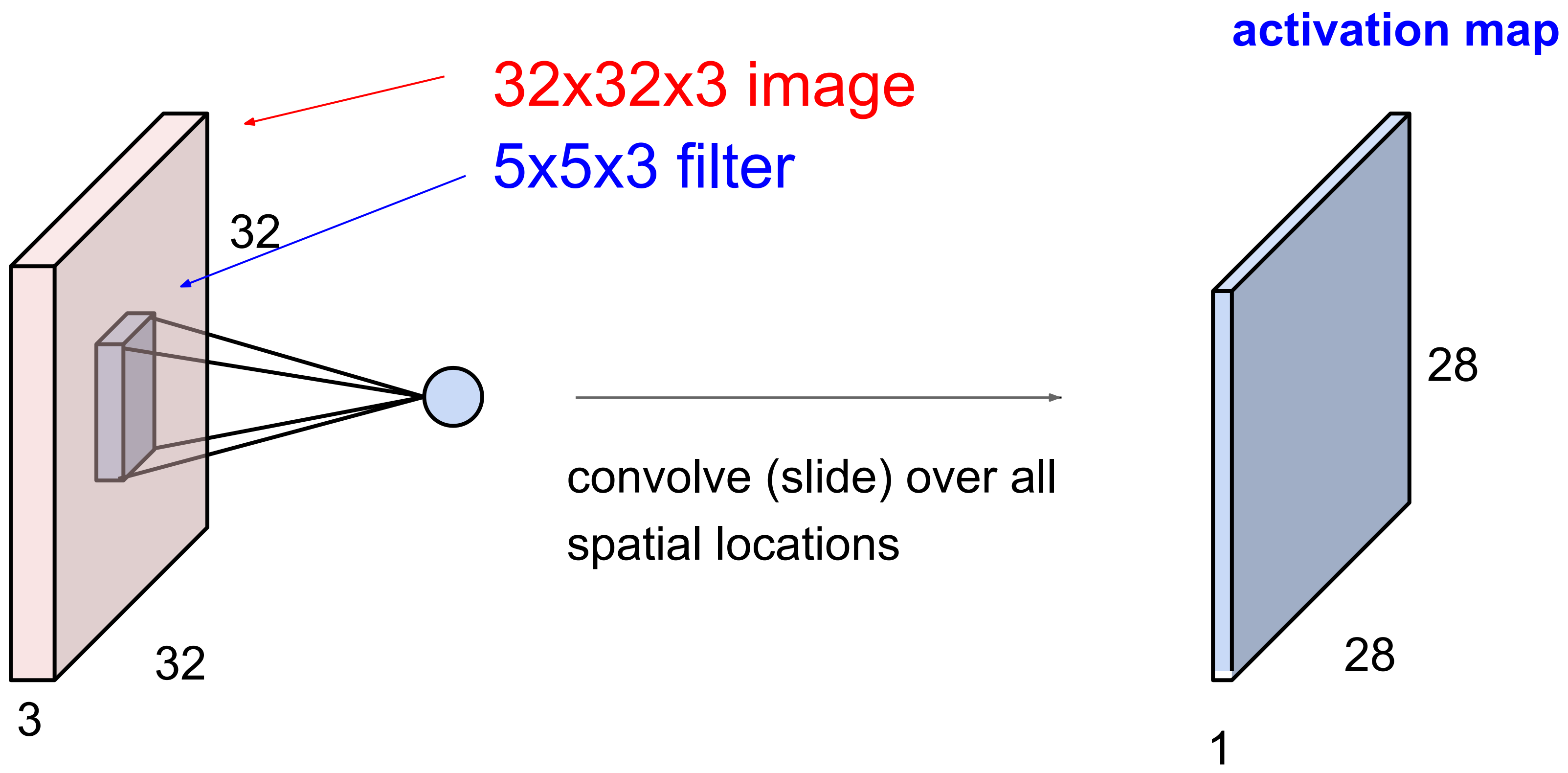
Convolution Layer



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)

$$w^T x + b$$

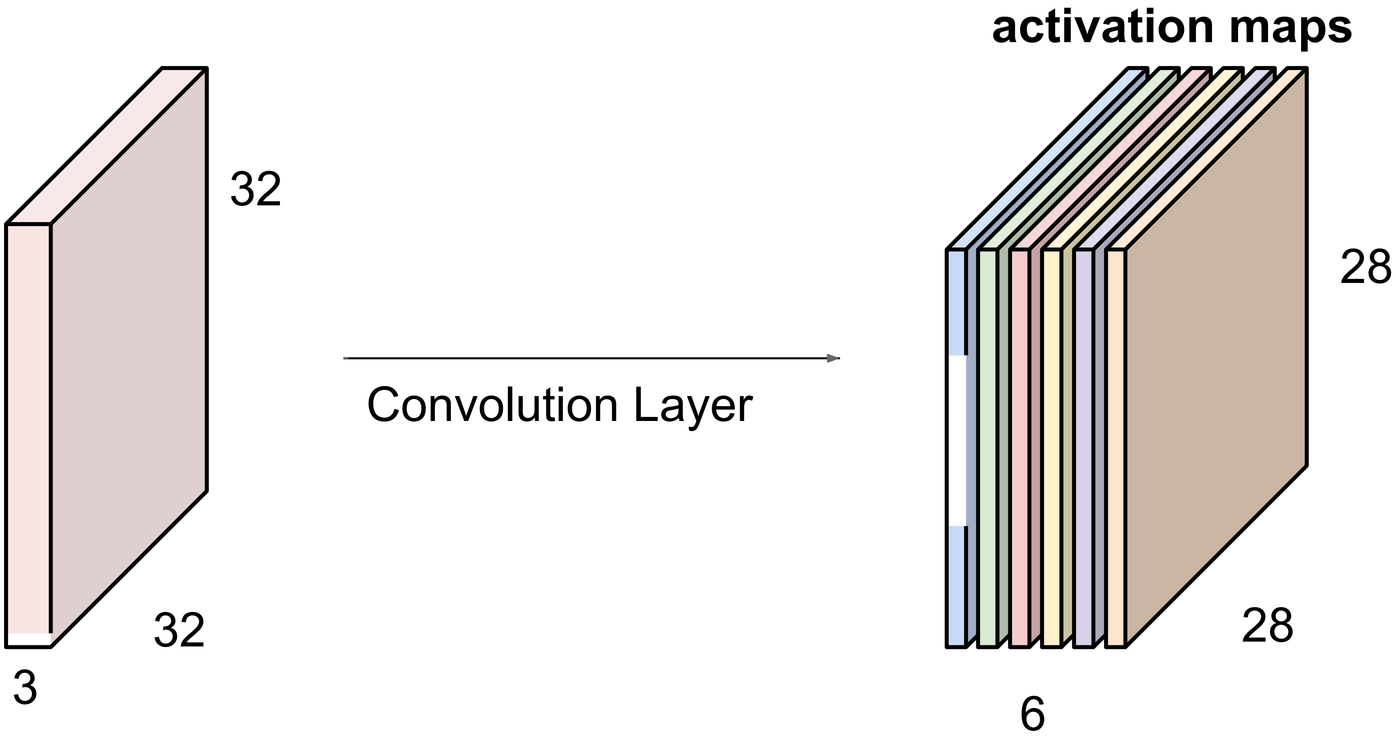
Convolution Layer



Convolution Layer

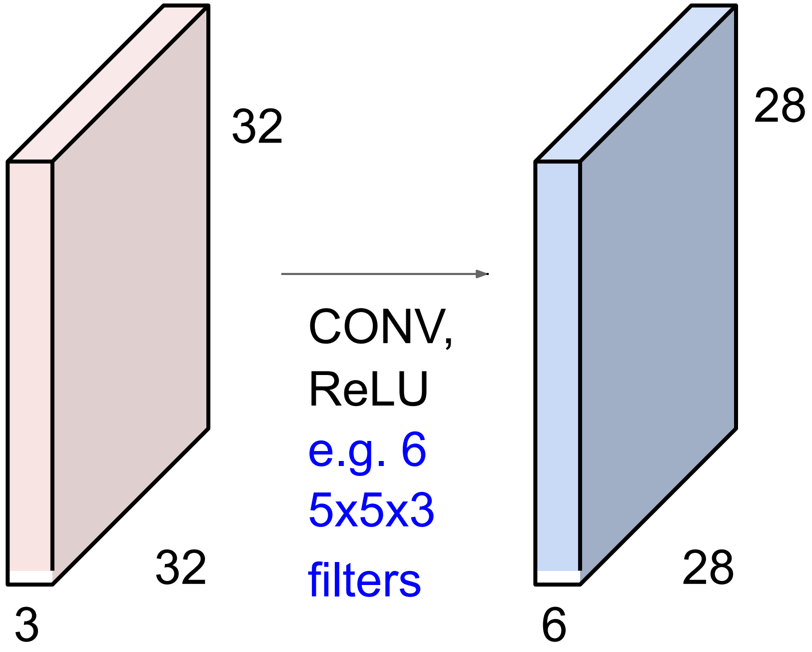
consider a second, green filter

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

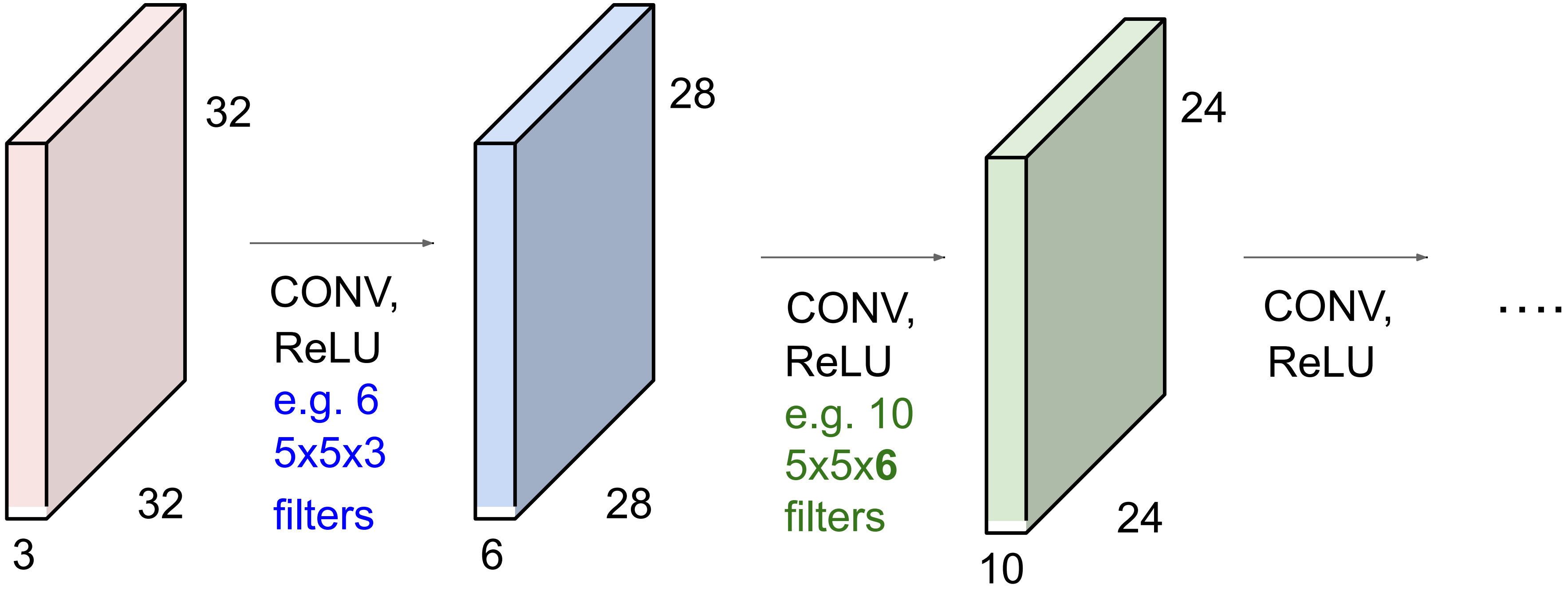


We stack these up to get a "new image" of size 28x28x6!

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



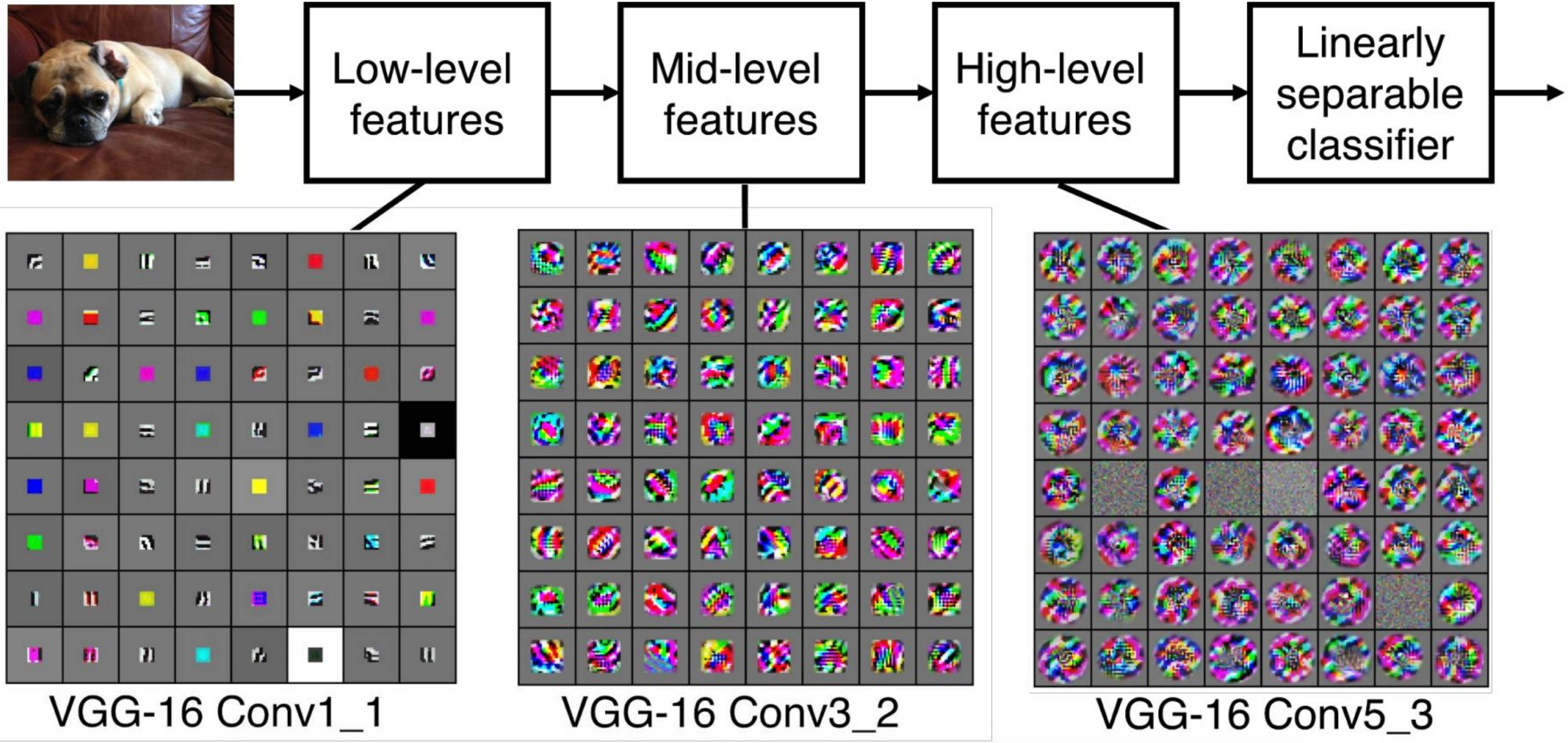
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview

[Zeiler and Fergus 2013]

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

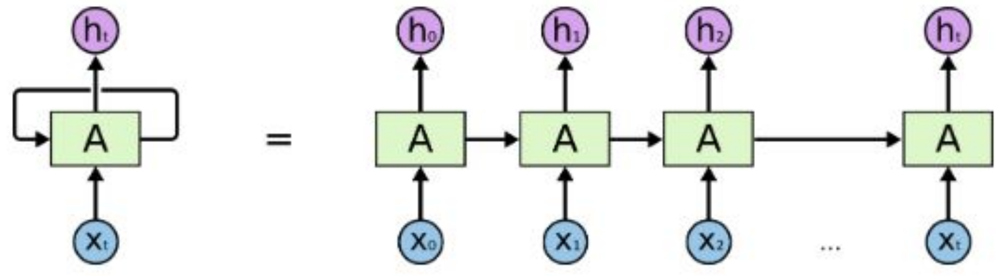


RNNs

- Review of RNNs
- RNN Language Models
- Vanishing Gradient Problem
- GRUs
- LSTMs

RNNs are good for:

- Learning representations for sequential data with temporal relationships
- Predictions can be made at every timestep, or at the end of a sequence
- **Q:** How do we incorporate past information to make predictions about the future?

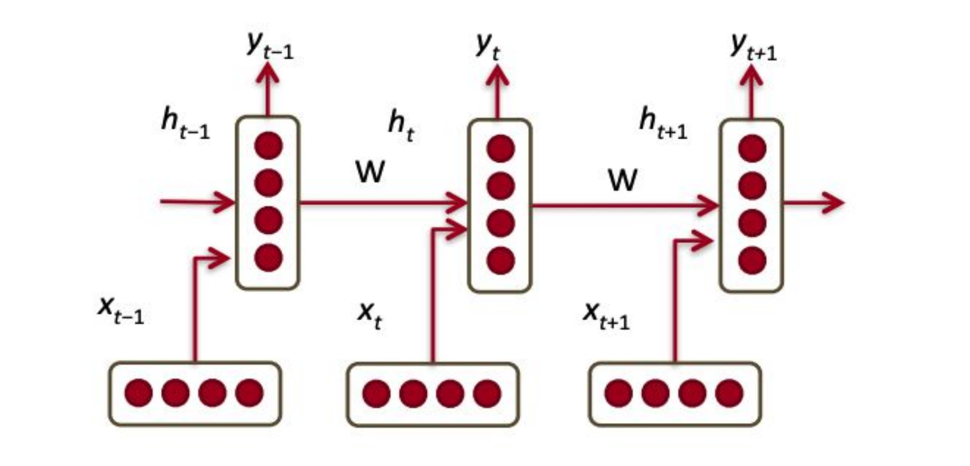


An unrolled recurrent neural network.

RNNs

Key points:

- Weights are shared (tied) across timesteps
- Hidden state at time *t* depends on previous hidden state and new input
- Backpropagation across timesteps (use unrolled network)



RNN Language Model

- Language Modeling (LM): task of computing probability distributions over sequence of words P(w_1, …, w_T)
- Important role in speech recognition, text summarization, etc.

Given list of word vectors: $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$

At a single time step:

$$h_t = \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right)$$

$$\hat{y}_t = \mathrm{softmax}\left(W^{(S)} h_t\right)$$

$$\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) = \hat{y}_{t,j}$$

Vanishing Gradient Problem

- Backprop in RNNs: recursive gradient call for hidden layer
- Magnitude of gradients of typical activation functions between 0 and 1.

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\|$$

- When terms less than 1, product can get small very quickly
- Vanishing gradients → RNNs fail to learn, since parameters barely update.
- GRUs and LSTMs to the rescue!

High-Level Idea

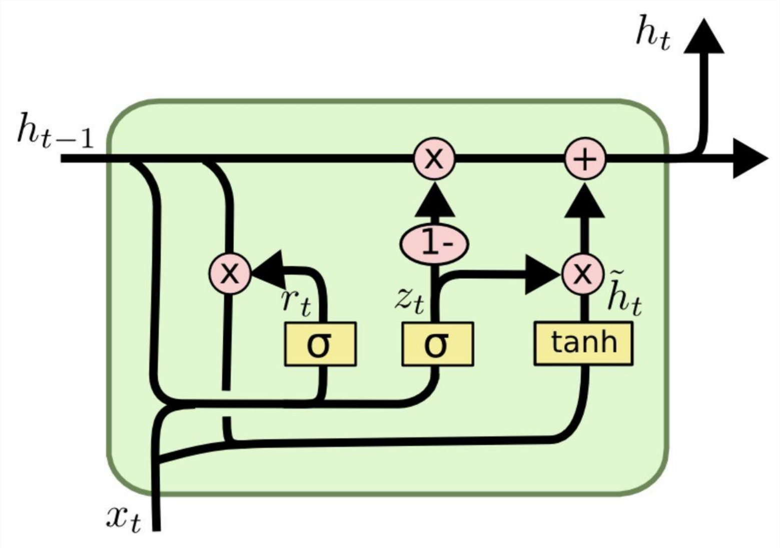
Gating mechanisms control information flow:

- How much do I care about the past?
- How much do I care about the present?
- How much do I want to output at the current timestep?

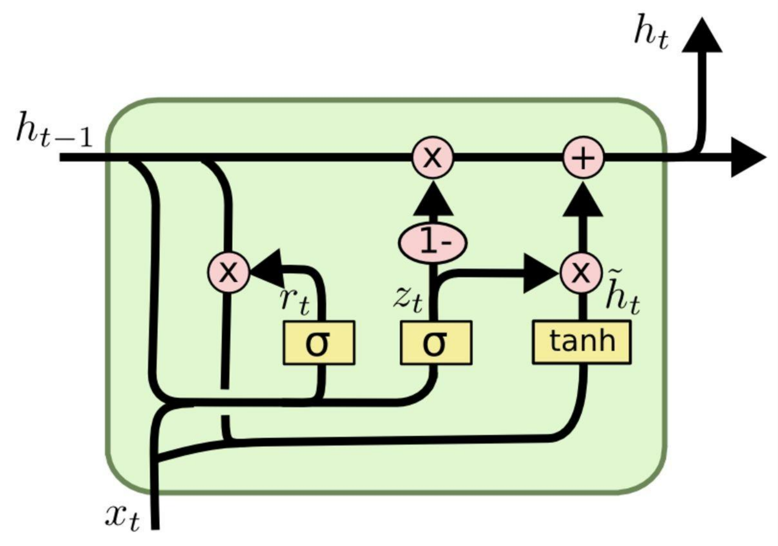
These questions are the underlying mechanisms behind GRUs/LSTMs.

Gated Recurrent Units (GRUs)

- z_t: Update gate
- r_t: Reset gate
- h_t: Cell memory content
    - Mixture of past memory and current memory content
    - Also functions as cell output
- The reset and update gates control long- and short-term dependencies (mitigate vanishing gradients problem!)



Gated Recurrent Units (GRUs)



$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

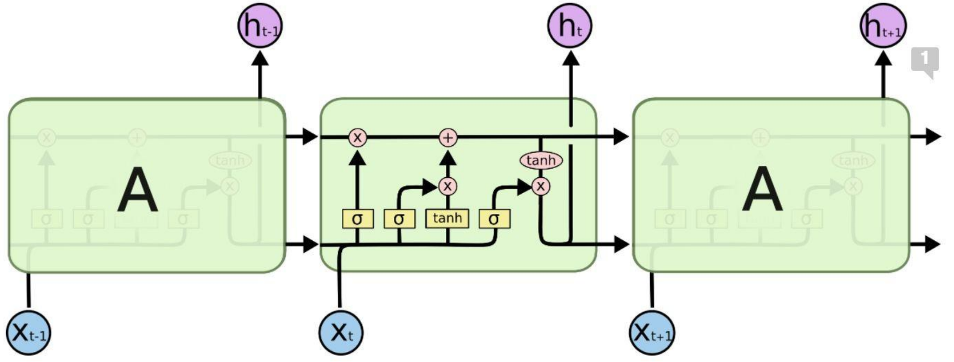
$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

$$\tilde{h}_t = \tanh(W x_t + r_t \circ U h_{t-1})$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t$$

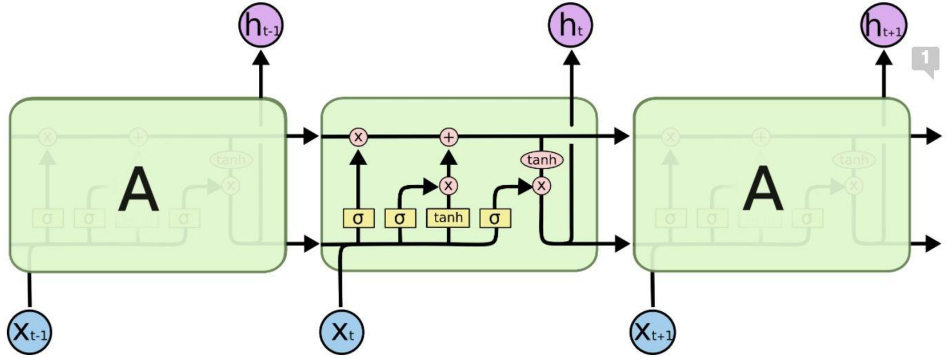
LSTMs

- f_t: forget gate
    - How much do I care about the past?
- i_t: input gate
    - How much do I care about the present?
- o_t: output gate
    - How much information do I output?
- C_t: current cell state
- h_t: cell output
- Cell state + output are separate!



The repeating module in an LSTM contains four interacting layers.

LSTMs



The repeating module in an LSTM contains four interacting layers.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$
$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \cdot \tanh(C_t)$$

So What’s Missing?

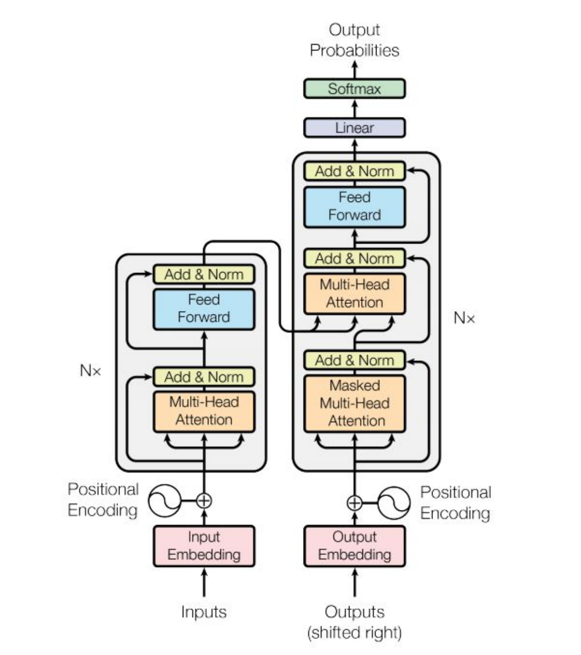
- RNNs + variants very successful for variable-length representations/seqs
  - Gating (LSTMs) for long-range error propagation
- But what if we want context from a ***really*** long time back? (many thousands of steps)
- Sequentiability prohibits parallelization within instances
- Long-range dependencies are still tricky

Transformer

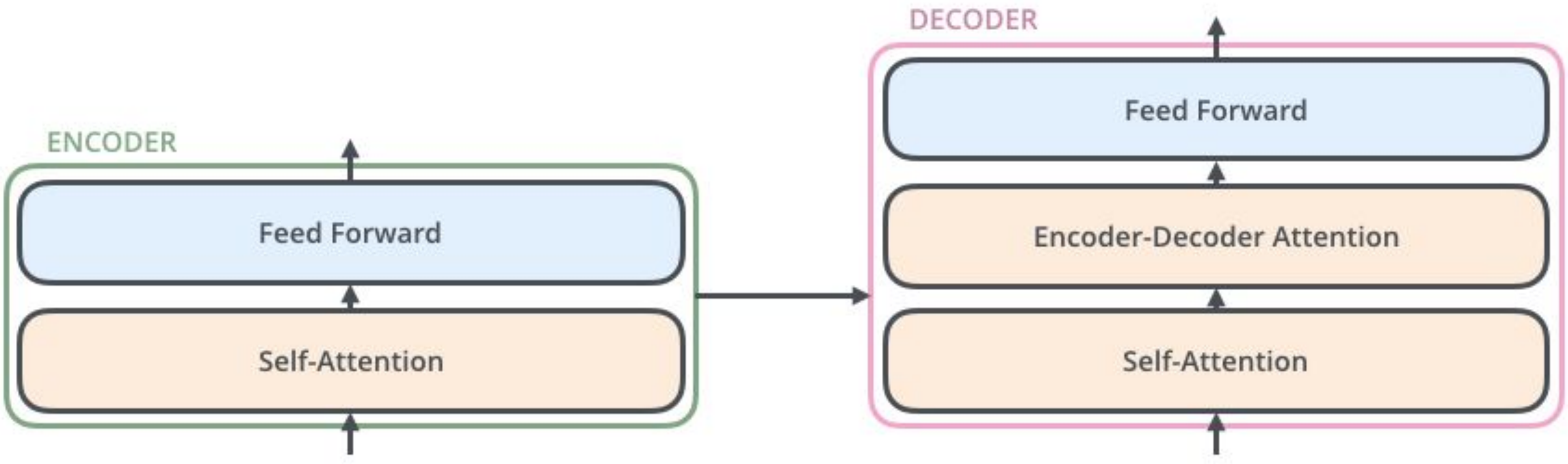
- Architecture
- Self-Attention
- Multi-Attention Heads
- Prediction vs. Sampling

Transformer Architecture

- Encoder stack
  - 6 layers, each with 2 sublayers: Multi-head Attention + FFN
- Decoder stack
  - Same as encoder, but with encoder-decoder self-attention
- Positional encodings
  - Added to input embedding



Transformer (Simplified)



Self-Attention

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

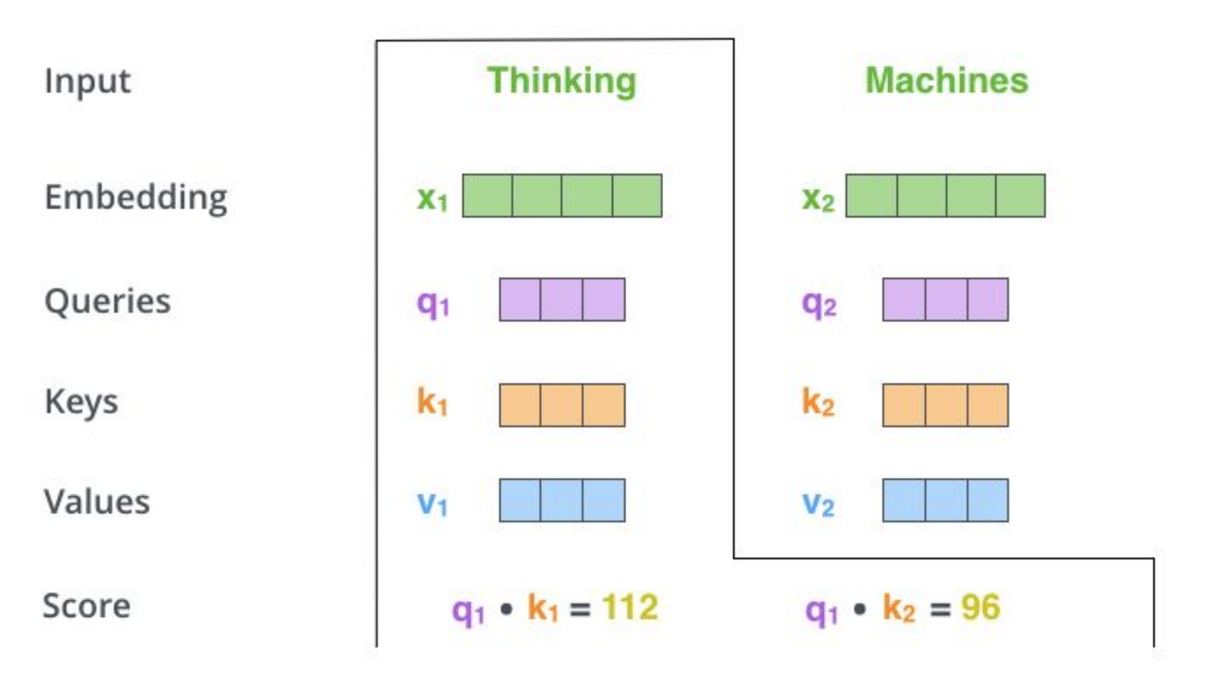
Self-Attention

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

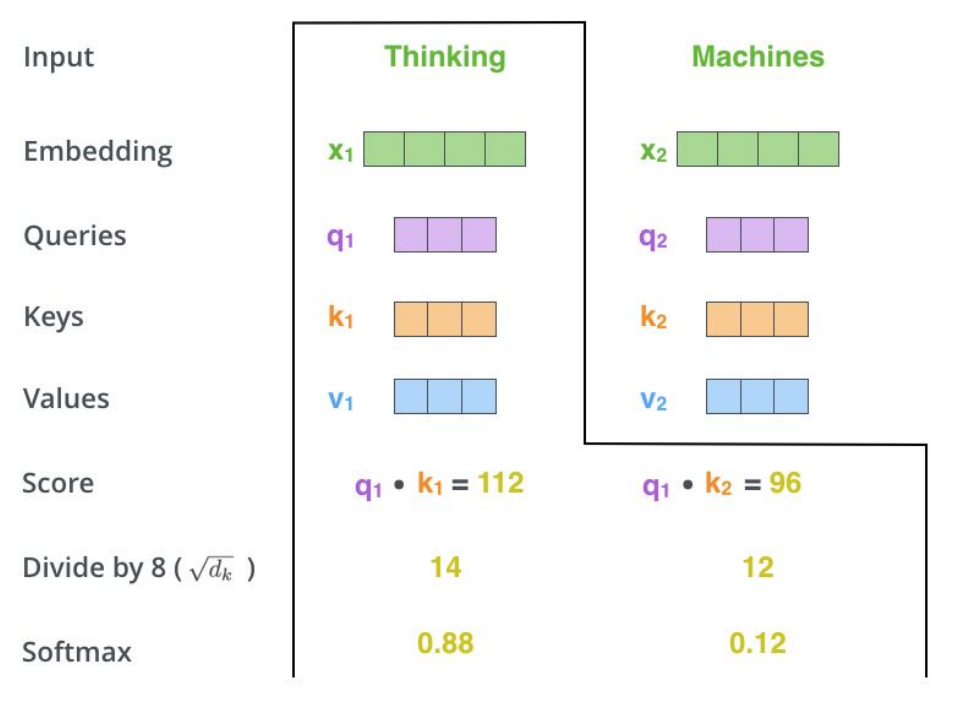
Key Idea:

- Reference by context
- Attention: query with vector, and look at similar things in your past
    - Find most similar key and get values that correspond to these similar things
- Softmax gives you probability distribution over keys
- Normalize by sqrt(d_k) for numerical stability

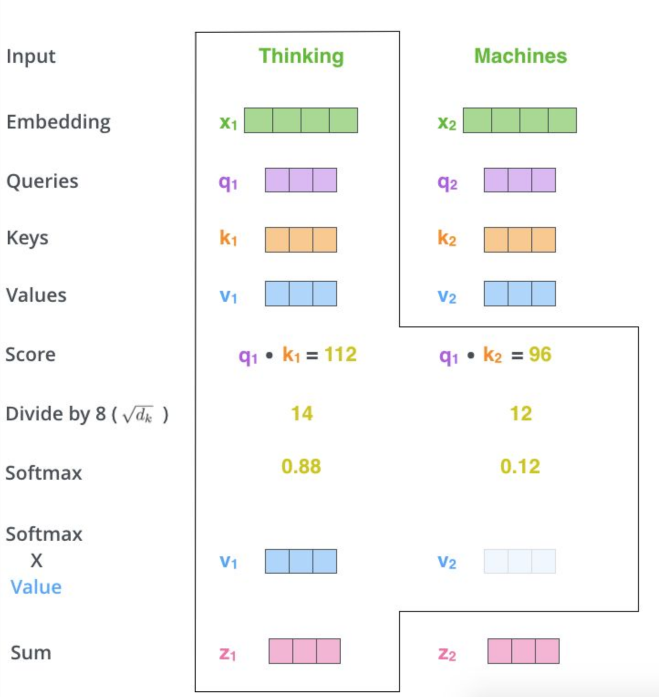
Self-Attention Example



Self-Attention Example



Self-Attention Example

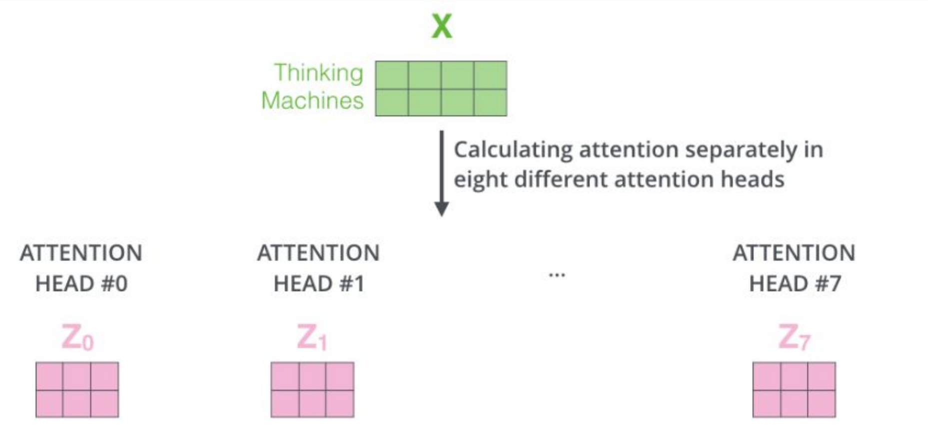


3 Types of Self-Attention

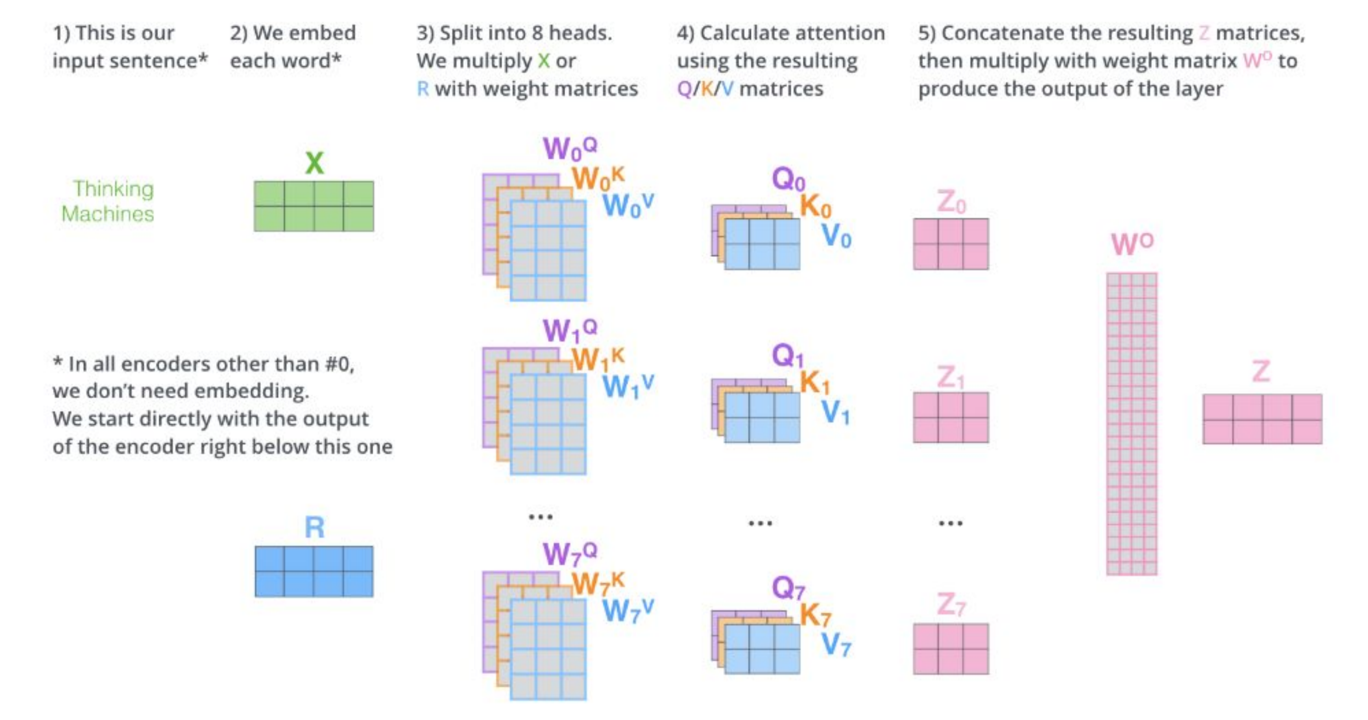
- Encoder self-attention: attends everywhere in the input
- Encoder-decoder attention (from output, attends to input)
- Masked decoder attention: only attends to things before

Multi-Attention Heads

Multiple attention heads to get more “representational power”



Multi-Attention Heads



Transformer

- Final linear layer
    - Projection of decoder output into a logits vector → each cell corresponds to the score of a unique word in the possible vocabulary
- Softmax layer
    - Turns logits into probabilities, which we use for prediction (training) or sampling (testing/inference)
- Cross Entropy Loss
    - Compare two probability distributions

Transformer: Prediction vs. Sampling

- During training, our examples are “labeled” in the sense that we know the true word that we are supposed to decode
- During sampling, we don’t know our target
    - Generate autoregressively, but using as input the previously generated token

Acknowledgements

- Slides adapted from:
  - Justin Johnson, Serena Yeung, and Fei-Fei Li (CS231N, Spring 2018) [slides]
  - Barak Oshri, Lisa Wang, and Juhi Naik (CS224N, Winter 2017) [slides]
  - Nish Chintala (CS236, Fall 2018) [slides]
- Chris Olah, OpenAI [blog]
- Lukasz Kaiser, Google Brain [talk]
- Anna Huang and Ashish Vaswani, Google Brain [slides] [paper]
- Jay Alammar, [blog]