Deep Learning Primer

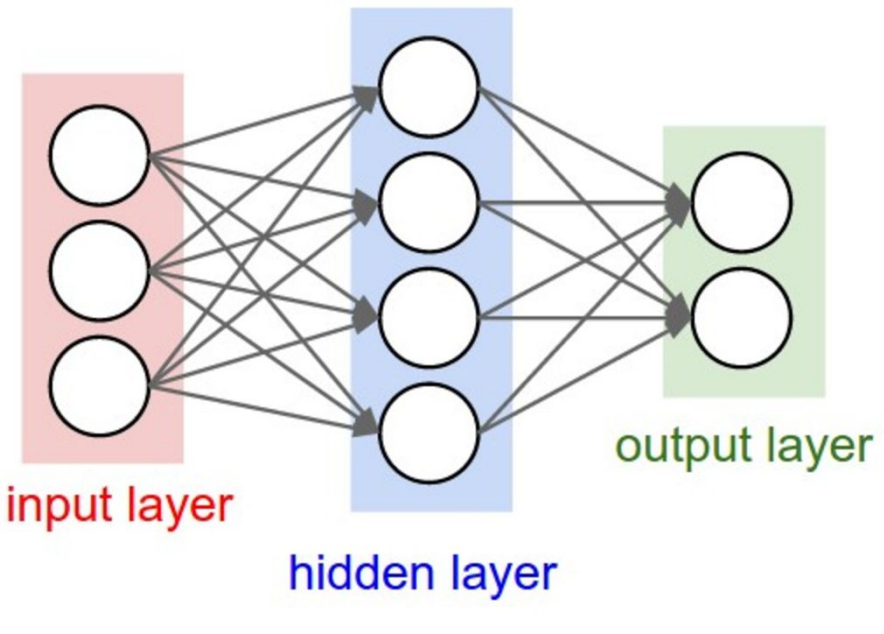
Nishith Khandwala

Neural Networks

Overview

- Neural Network Basics
- Activation Functions
- Stochastic Gradient Descent (SGD)
- Regularization (Dropout)
- Training Tips and Tricks

Neural Network (NN) Basics



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

Steps to train a 1-hidden layer NN:

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

Activation Functions: Sigmoid

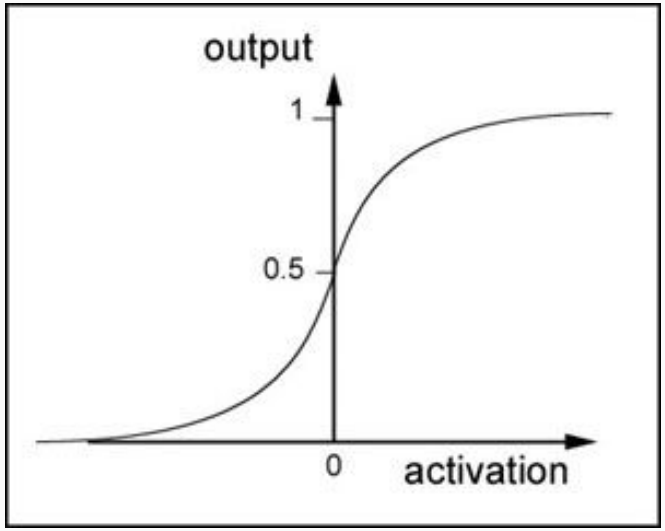
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Properties:

- Squashes input between 0 and 1.

Problems:

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



Activation Functions: Tanh

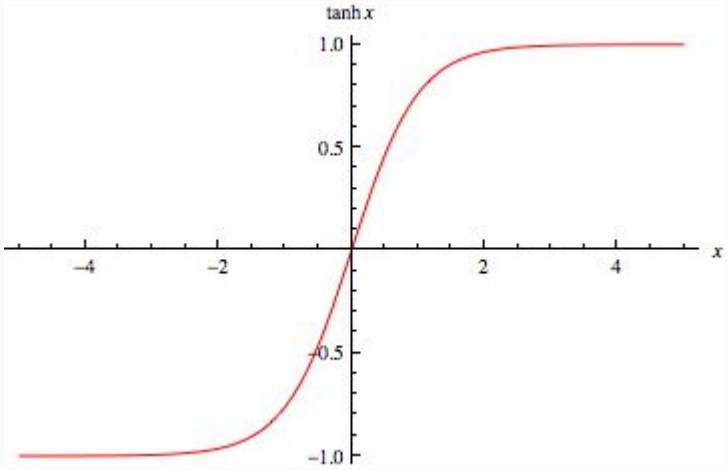
$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Properties:

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

Problems:

- Saturation of neurons kills gradients.



Activation Functions: ReLU

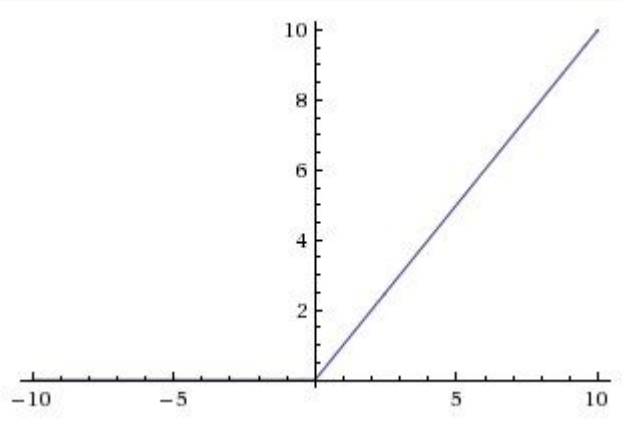
$$relu(x) = max(0, x)$$

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)

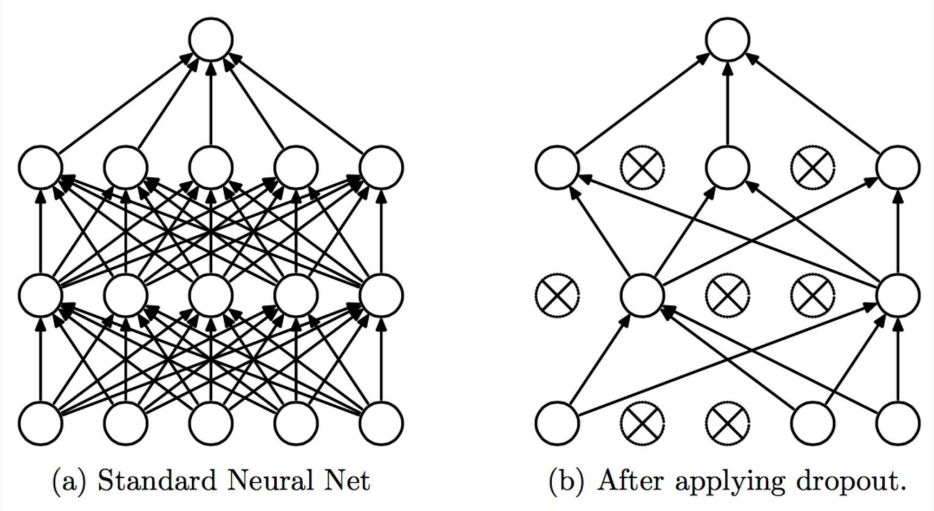
$$\theta \leftarrow \theta - \alpha \nabla_\theta J$$

- 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**.

Regularization: Dropout

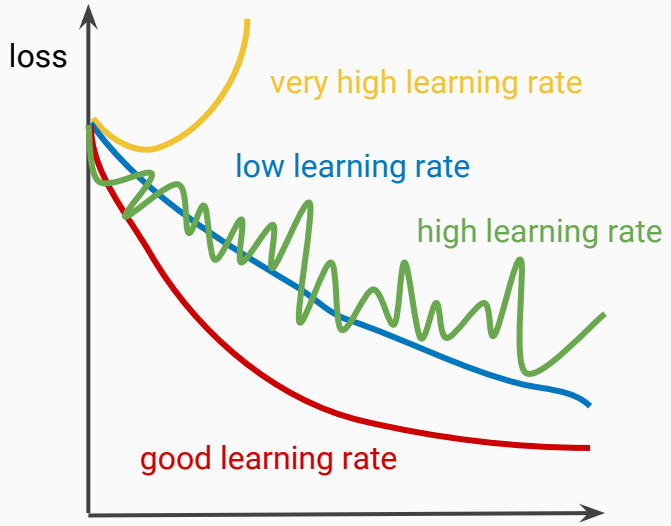
- **Randomly drop neurons** at forward pass during training.
- **At test time, turn dropout off.**
- **Prevents overfitting** by forcing network to learn redundancies.

Think about dropout as **training an ensemble of networks.**



Training Tips and Tricks

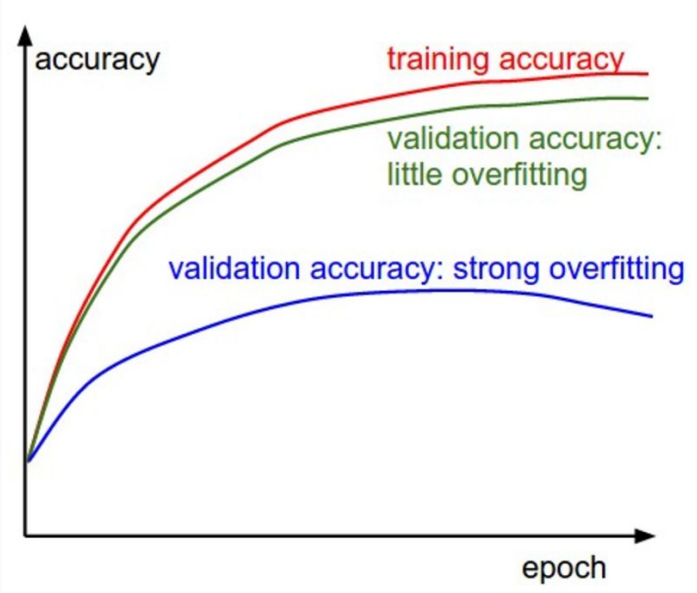
- Learning rate:
  - If loss curve seems to be unstable (jagged line), **decrease** learning rate.
  - If loss curve appears to be “linear”, **increase** learning rate.



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.



Backpropagation and Gradients

Slides courtesy of Barak Oshri

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)

$$h_1 = xW_1 + b_1$$

$$z_1 = \sigma(h_1)$$

$$z_2 = z_1W_2 + b_2$$

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

Intermediate **Gradients**
(backward propagation)

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

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

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

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

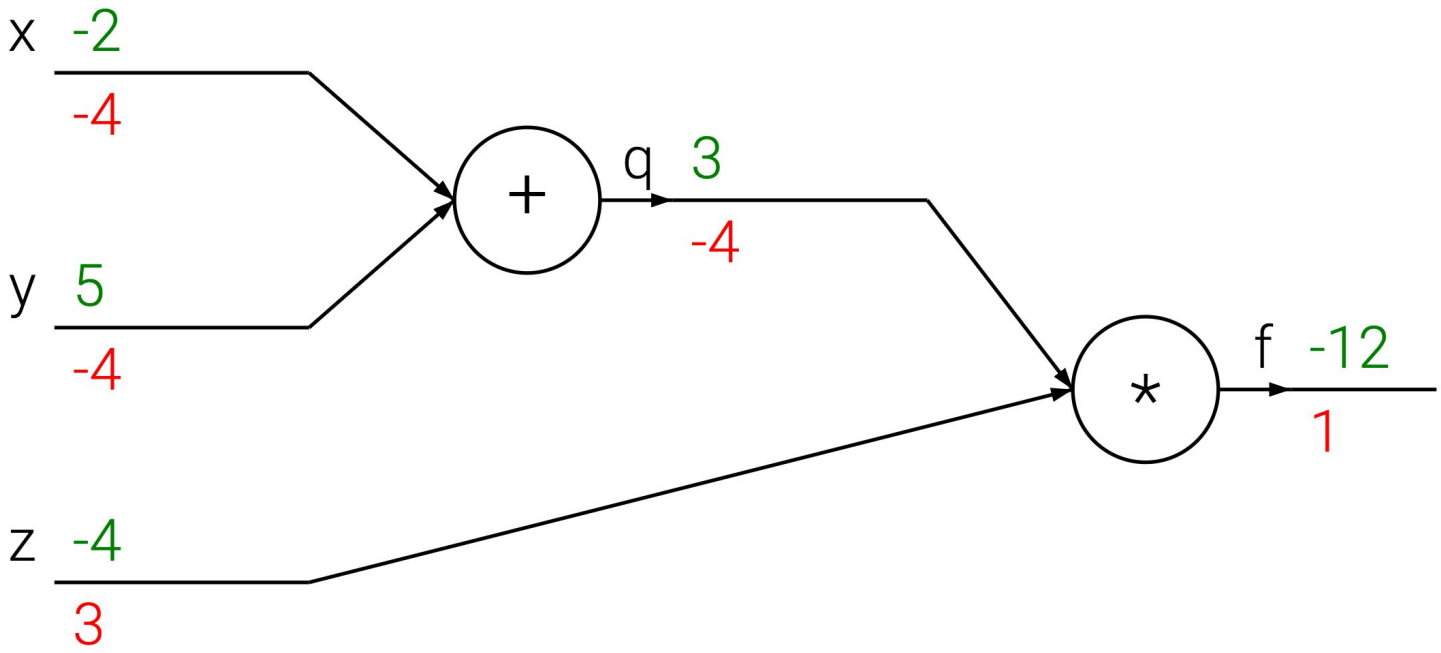
Chain Rule Behavior

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

Key chain rule intuition:

Slopes multiply

Circuit Intuition



Backprop Menu for Success

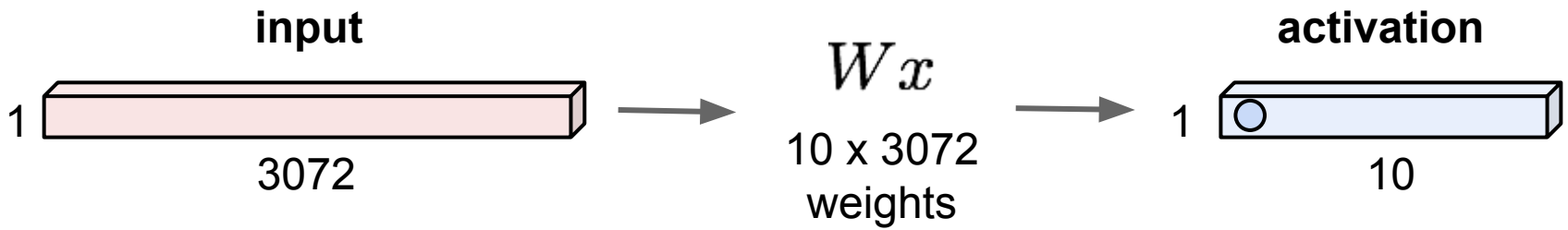
1. Write down variable graph
2. Compute derivative of cost function
3. Keep track of error signals
4. Enforce shape rule on error signals
5. Use matrix balancing when deriving over a linear transformation

Convolutional Neural Networks

Slides courtesy of Justin Johnson, Serena Yeung, and Fei-Fei Li

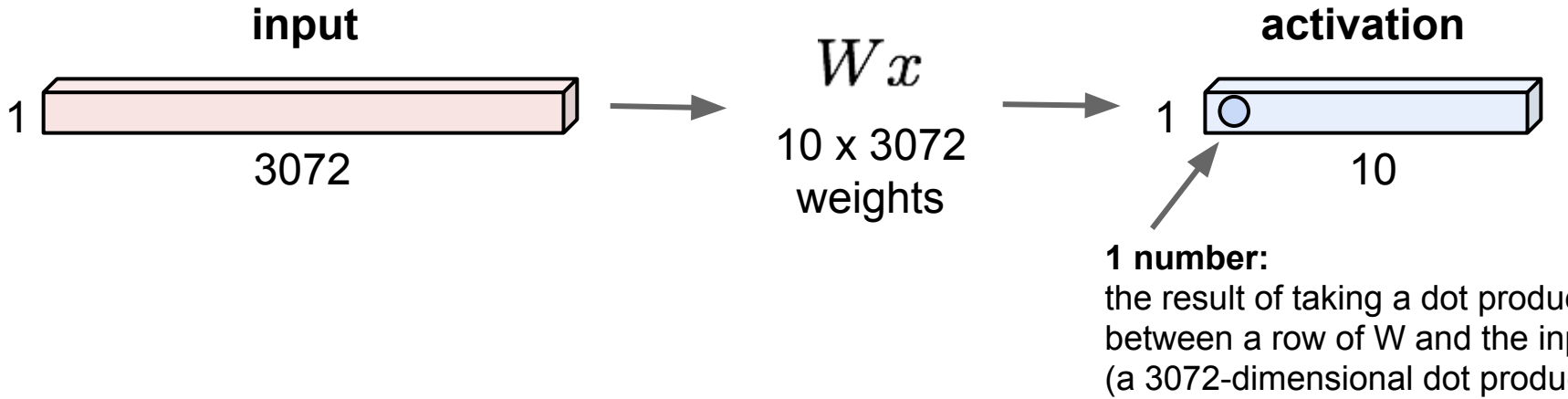
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



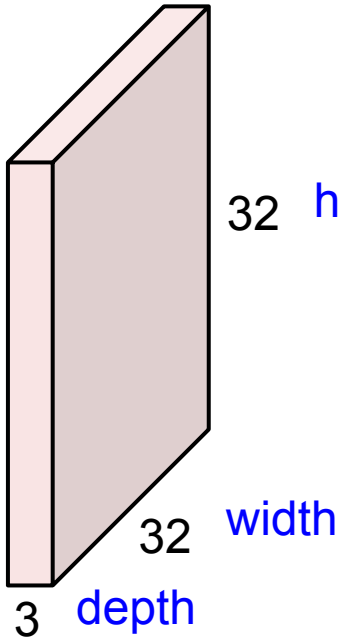
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



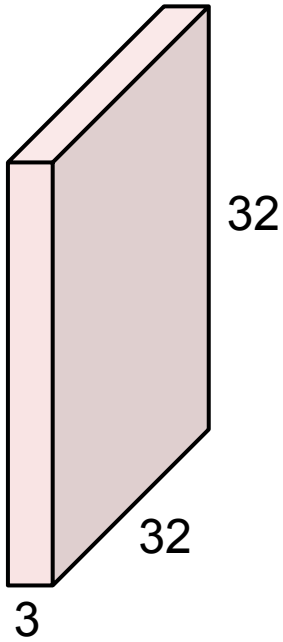
Convolution Layer

32x32x3 image -> preserve spatial structure

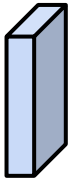


Convolution Layer

32x32x3 image



5x5x3 filter

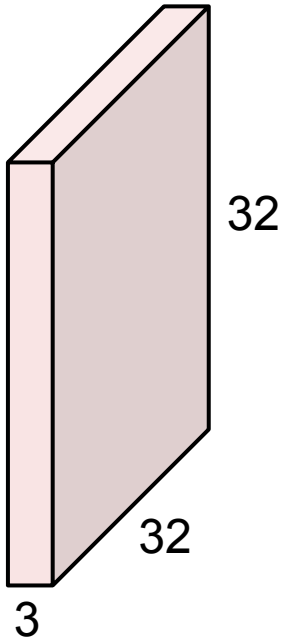


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

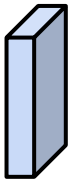
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

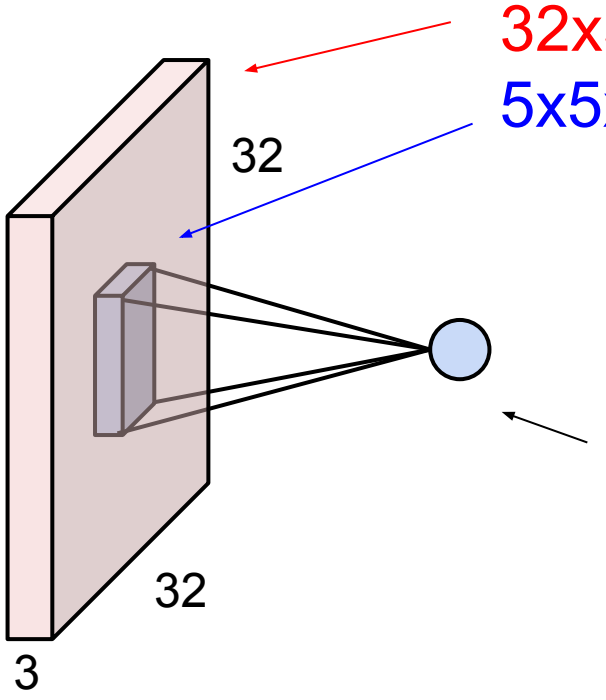


5x5x3 filter



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

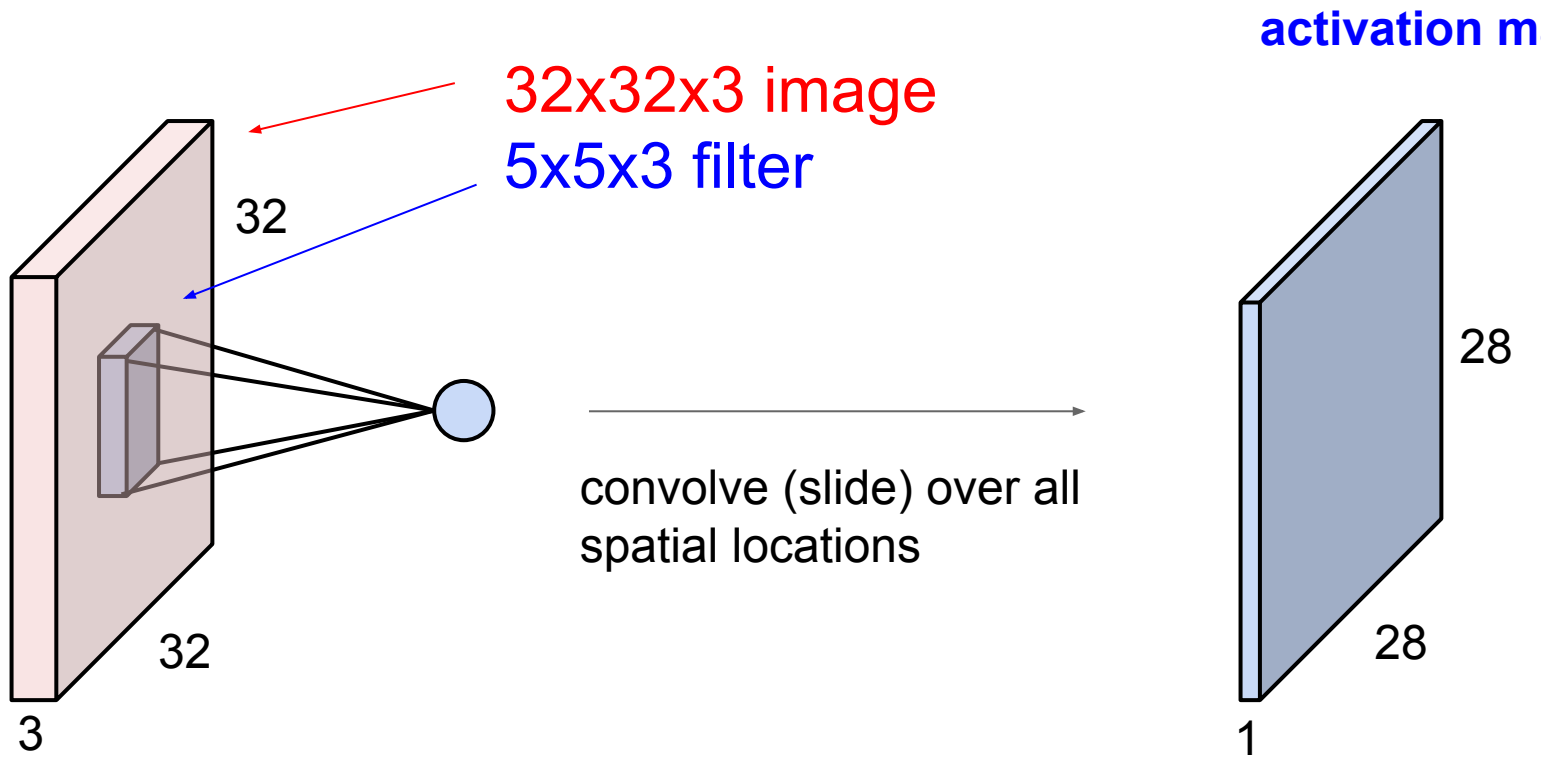
Convolution Layer



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)

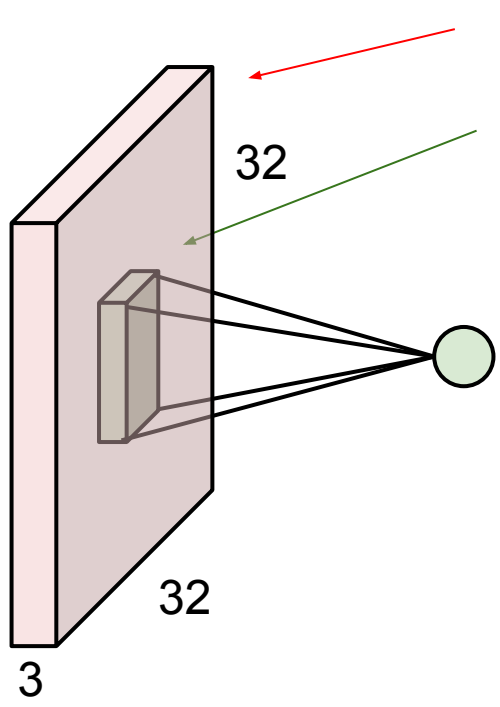
$$w^T x + b$$

Convolution Layer

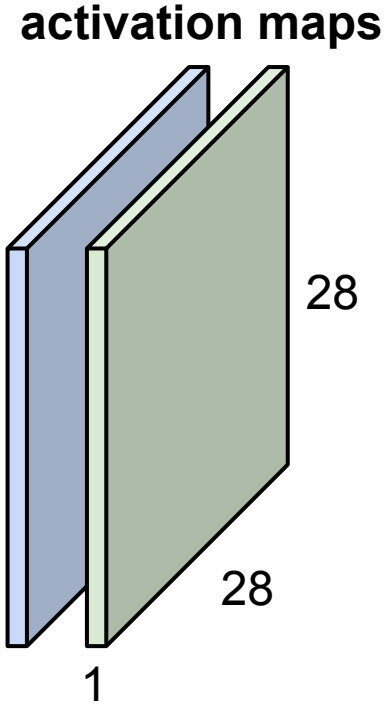


Convolution Layer

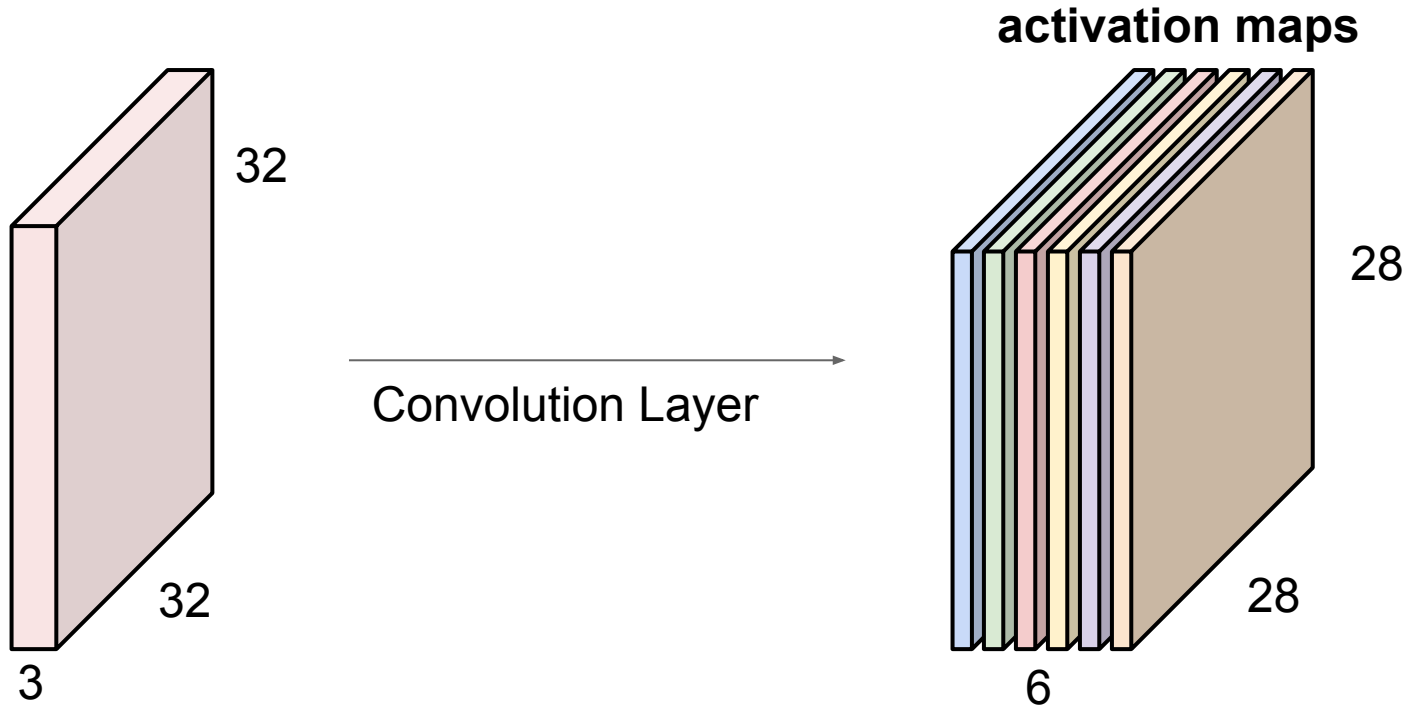
consider a second, green filter



convolve (slide) over all spatial locations

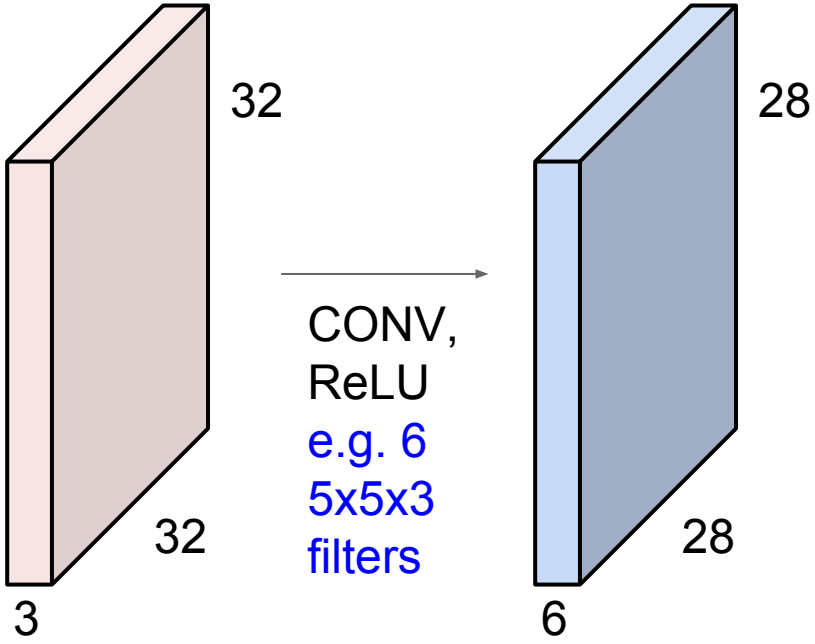


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

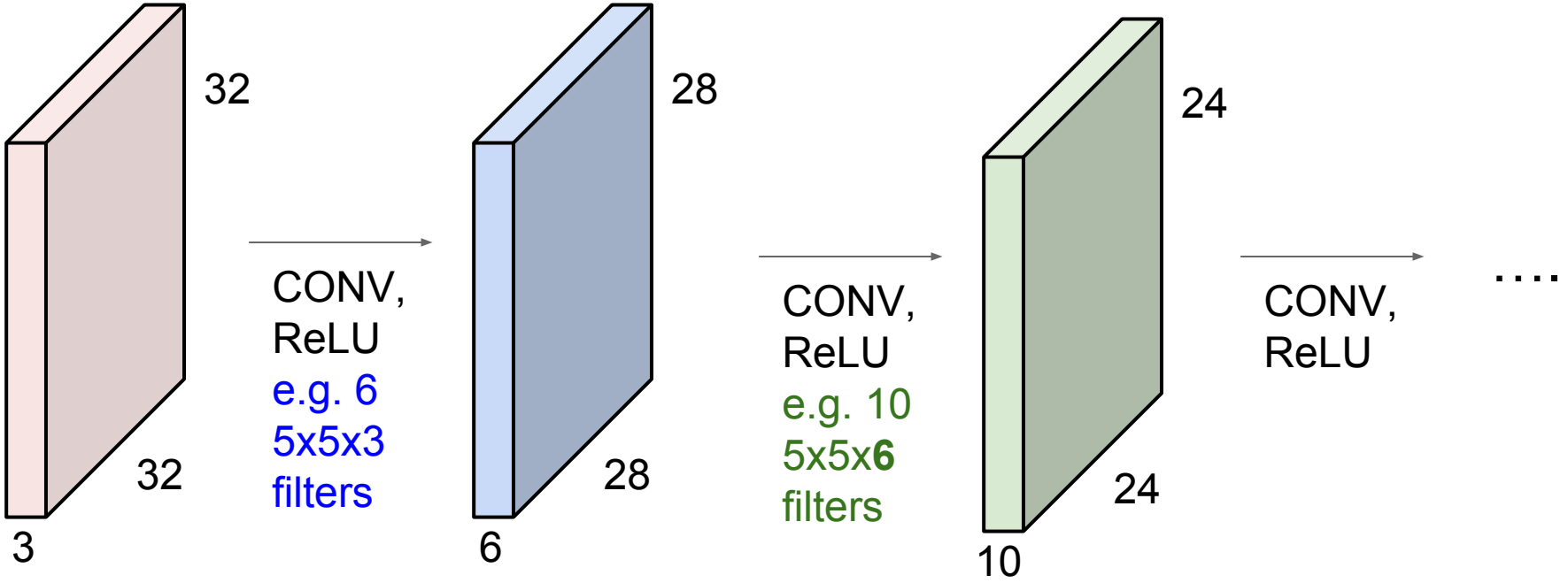


We stack these up to get a "new image" of size 28x28x6!

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



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



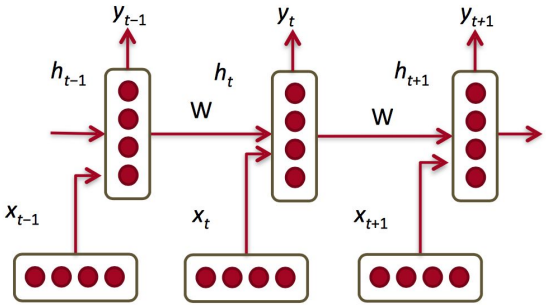
RNNs, Language Models, LSTMs, GRUs

Slides courtesy of Lisa Wang and Juhi Naik

RNNs

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

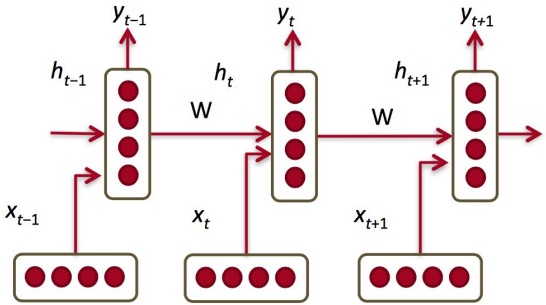
RNN Review



Key points:

- Weights are shared (tied) across timesteps (W_{xh}, W_{hh}, W_{hy})
- Hidden state at time *t* depends on previous hidden state and new input
- Backpropagation across timesteps (use unrolled network)

RNN Review



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

RNN Language Model

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

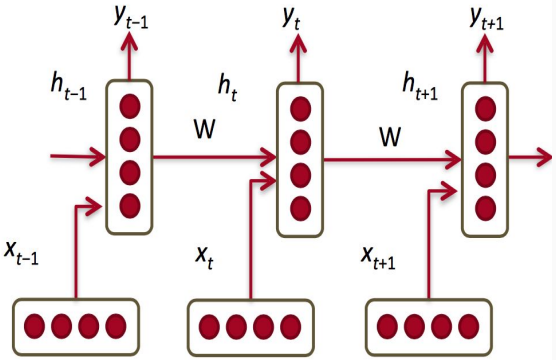
Given list of word vectors: $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$

At a single time step:

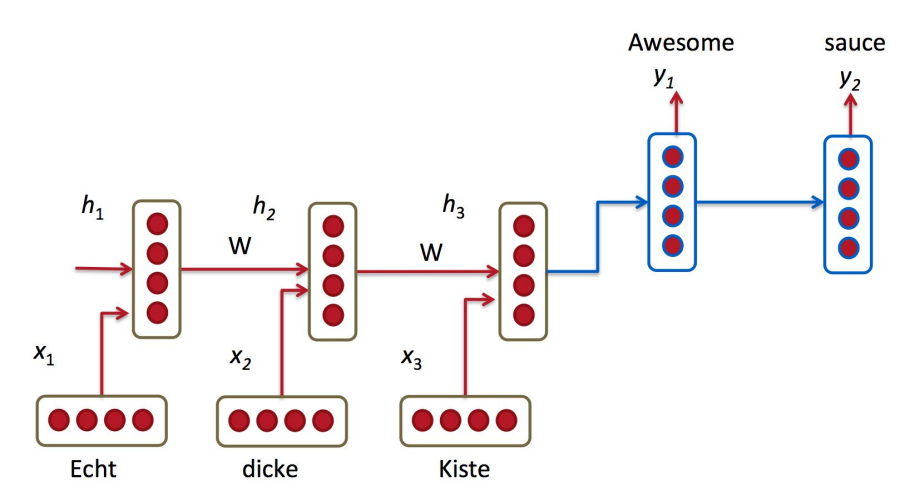
$$h_t = \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right)$$

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

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



RNN Language Model for Machine Translation



- Encoder for source language
- Decoder for target language
- Different weights in encoder and decoder sections of the RNN (Could see them as two chained RNNs)

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!

Gated Recurrent Units (GRUs)

- Reset gate, r_t
- Update gate, z_t
- r_t and z_t control long-term and short-term dependencies (mitigates vanishing gradients problem)

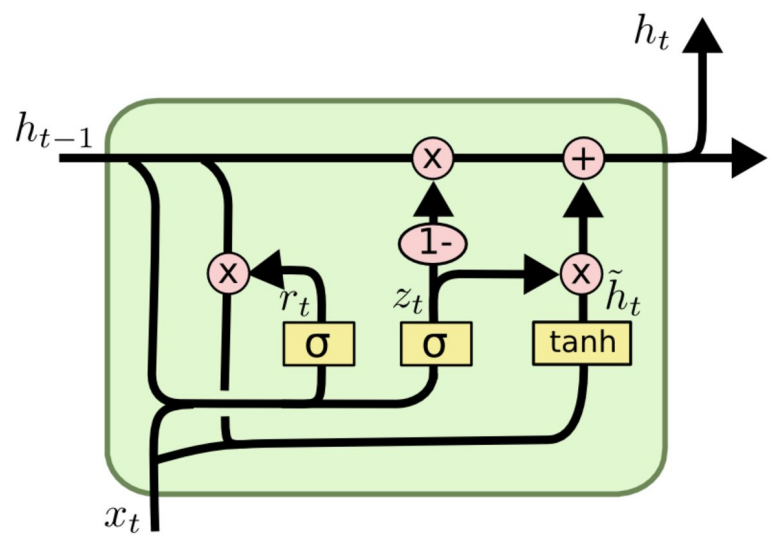
$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

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

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

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

Gated Recurrent Units (GRUs)



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

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

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

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

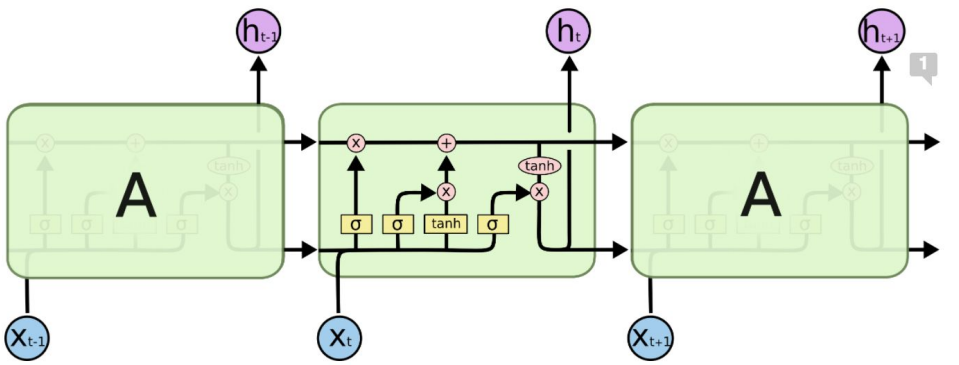
Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTMs

- i_t: Input gate - How much does current input matter
- f_t: Input gate - How much does past matter
- o_t: Output gate - How much should current cell be exposed
- c_t: New memory - Memory from current cell

$$
\begin{aligned}
i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right) \\
f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right) \\
o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right) \\
\widetilde{c}_t &= \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t \\
h_t &= o_t \circ \tanh\left(c_t\right)
\end{aligned}
$$

LSTMs



The repeating module in an LSTM contains four interacting layers.

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

$$
\begin{aligned}
i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right) \\
f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right) \\
o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right) \\
\widetilde{c}_t &= \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t \\
h_t &= o_t \circ \tanh\left(c_t\right)
\end{aligned}
$$

Acknowledgements

- Slides adapted from:
  - Barak Oshri, Lisa Wang, and Juhi Naik (CS224N, Winter 2017)
  - Justin Johnson, Serena Yeung, and Fei-Fei Li (CS231N, Spring 2018)
- Andrej Karpathy, Research Scientist, OpenAI
- Christopher Olah, Research Scientist, Google Brain