Chapter 7, Part 3: Modules and Architectures

Advanced Topics in Statistical Machine Learning

Tom Rainforth

Hilary 2024

rainforth@stats.ox.ac.uk

Complicated Architectures from Simple Building Blocks

Ops, Modules, and Factories

To help with exposition, we define three kinds of objects used to construct a computation graph

Op A **node** in a computation graph. Given an input, its output will be a block of (hidden) unit values. We can think of ops as **evaluations** of functions; they have no parameters of their own.

Module A **function** that can applied multiple times within the graph. It may be **parameterized**, in which case the parameters are **shared** across uses. Applying a module creates an op.

Factory A procedure that **generates** modules. This allows us to produce multiple modules that have separate sets of parameters

A Simple Example

Consider a factory Factory(n, σ) that generates functions of the form $\sigma(Wx)$ where $W \in \mathbb{R}^{n \times n}$ and σ is an element–wise non–linearity. We can then construct the computation graph:

$$\begin{aligned}
\text{module}_1 &\sim \text{Factory}(n, \sigma) \\
\text{module}_2 &\sim \text{Factory}(n, \sigma) \\
\text{op}_1 &= \text{module}_1(x) \\
\text{op}_2 &= \text{module}_1(\text{op}_1) \\
\text{op}_3 &= \text{module}_2(\text{op}_2) \\
\text{op}_4 &= \text{module}_2(\text{op}_3)
\end{aligned}$$

module_1 and module_2 have the same form but different parameters W_1 and W_2. The computation graph equates to the computation

$$f(x) = \sigma(W_2\sigma(W_2\sigma(W_1\sigma(W_1x))))$$

Linear Factories

One of the simplest and most important kind of modules are those generated by a **linear factory**

$$\text{module} \sim \text{Linear}(m, n)$$
$$\text{module}(x) = Wx + b$$

which has parameters corresponding to a **weight matrix** $W \in \mathbb{R}^{m \times n}$ and a **bias vector** $b \in \mathbb{R}^m$. Here m and n correspond to the number of output and input units respectively.

In the most common case, W will be a dense matrix—for which the factory is typically known as fully-connected or dense—but it is also possible to have factories that produce sparse weight matrices.

Nonlinearity Modules

Fixed nonlinearities are a class of modules that have no learnable parameters

They are most commonly **element-wise** nonlinearities, such that they form activation functions

$$\begin{aligned}
\text{sigmoid}(x) &= 1/(1+\exp(-x)) \\
\tanh(x) &= (\exp(x)-\exp(-x))/(\exp(x)+\exp(-x)) \\
\text{ReLU}(x) &= \max(0,x) \\
\text{softplus}(x) &= \log(1+\exp(x))
\end{aligned}$$

There are also some non–element–wise nonlinearities, often used for calculating output layers, e.g. a **softmax**

$$\text{softmax}([x_1,\ldots,x_d]) = \left[\frac{\exp(x_1)}{\sum_{i=1}^{d}\exp(x_i)},\ldots,\frac{\exp(x_d)}{\sum_{i=1}^{d}\exp(x_i)}\right]$$

Max Pooling

A **max pooling** module subsamples its input, taking the largest value in some surrounding area

For example, max pooling the first two dimensions of a 3D input:

$$x'_{i'j'k'} = \max_{i=1}^{d_1} \max_{j=1}^{d_2} x_{i+d_1(i'-1),\, j+d_2(j'-1),\, k'}$$

This can be used to reduce the number of parameters in a model as fewer connections are needed on the next layer

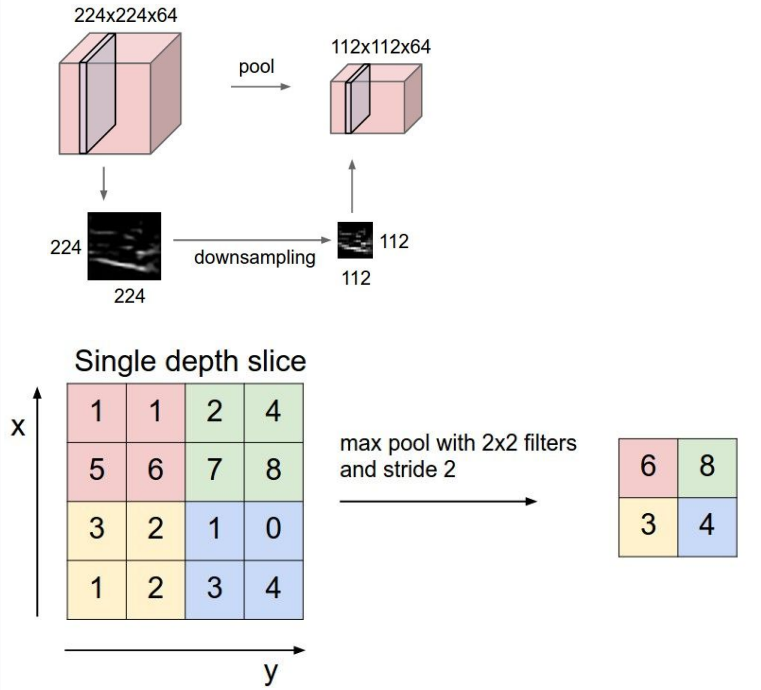


Figure Credit:

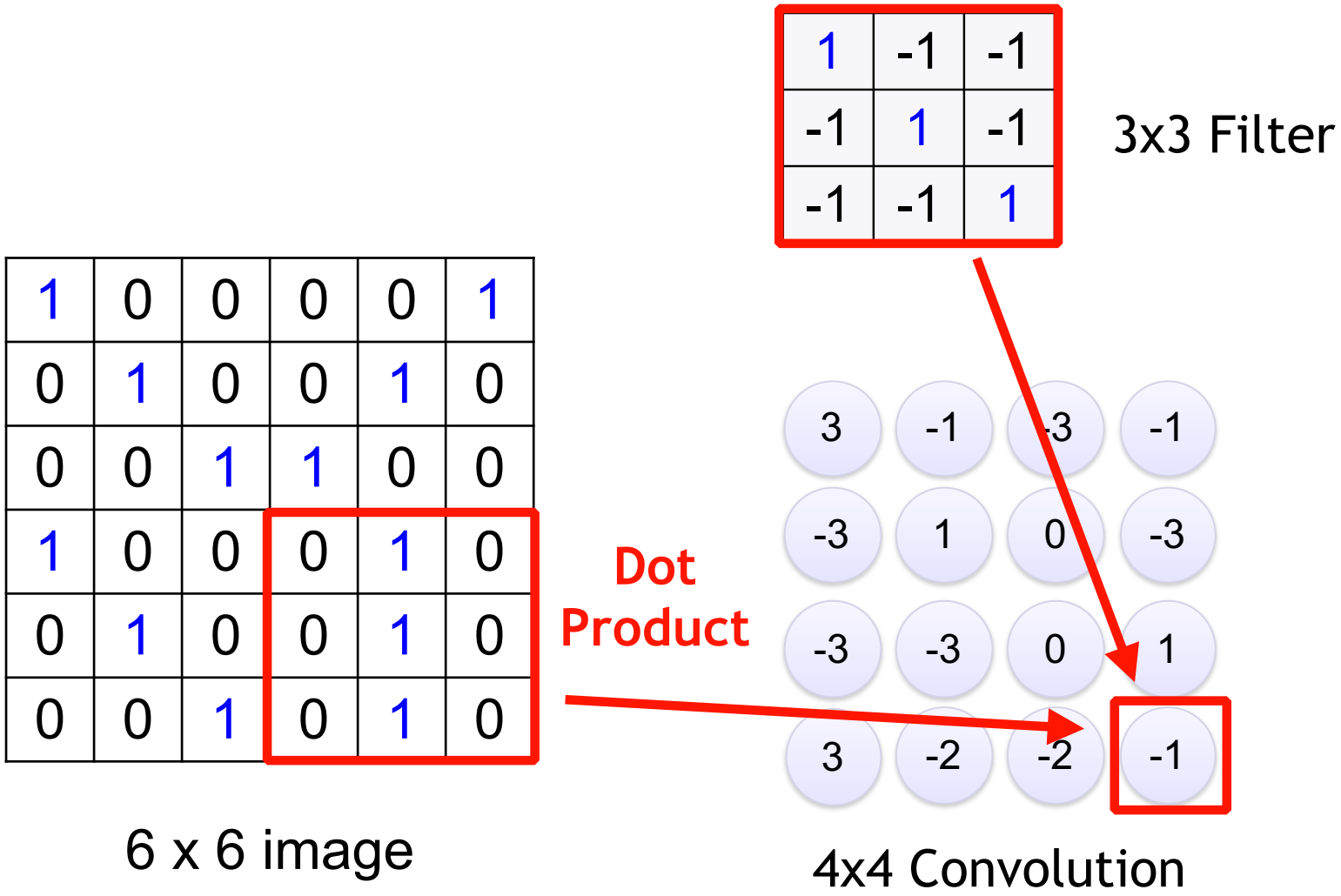
`http://cs231n.github.io/convolutional-networks/`

Convolutional Neural Networks

- **Convolutional Neural Networks** (CNNs or ConvNets) are one of the key workhorses of deep learning, particularly when dealing with high–dimensional inputs like images
- Though many of the core ideas stem back to the 80s and 90s, current state–of–the–art approaches in many application areas are still based on CNNs and their derivatives
- Their key feature, convolutional modules, have **sparse** connections with many **shared** weights; they require far fewer parameters for the same number of hidden units than MLPs
- They form a principled means of designing large networks while retaining a tractable number of parameters[1]

[1] Tractable here is used in quite a loose sense: some modern incarnations have 50M+ parameters. Nonetheless this is dwarfed by the current record for transformer–based networks of 1.6 Trillion parameters.

Convolutions (Technically Cross Correlations)

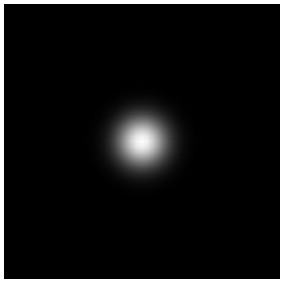


Credit: https://cs.uwaterloo.ca/~mli/Deep-Learning-2017-Lecture5CNN.ppt

Convolutions for Image Processing

*

Gaussian
Convolution



=

Credit: Frank Wood

Convolutions for Image Processing

Emboss Filter

$$* \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} =$$

Credit: Frank Wood

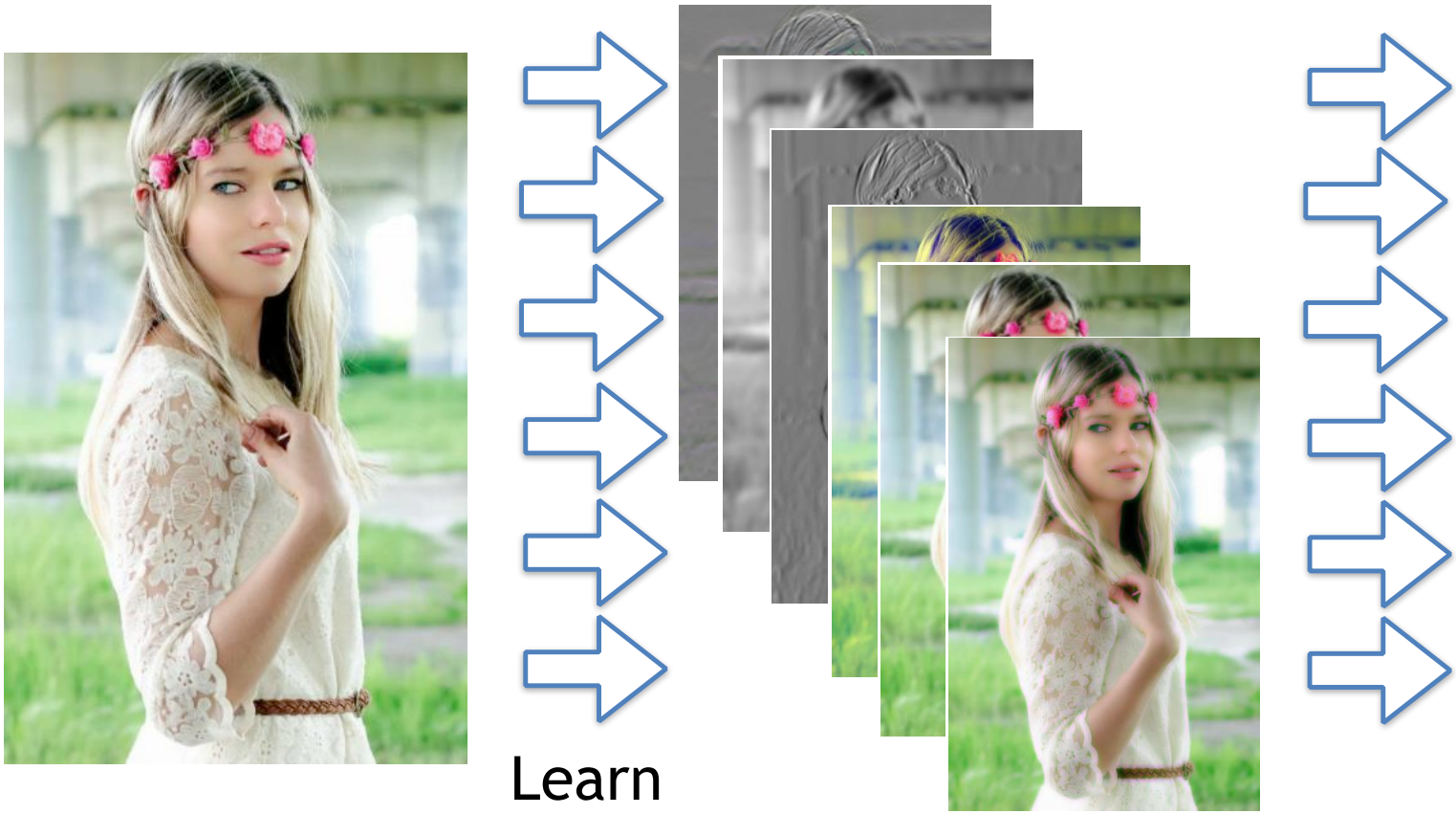
Convolutions for Image Processing

Sharpen Blue Channel

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Credit: Frank Wood

Convolutional Layers

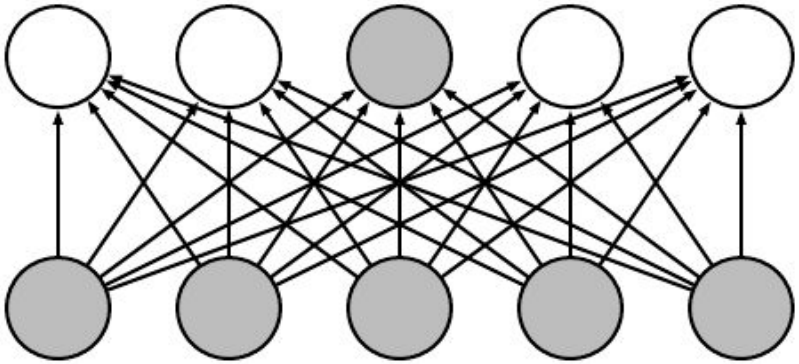


Input Layer

Hidden Layer 1 (before applying activations)

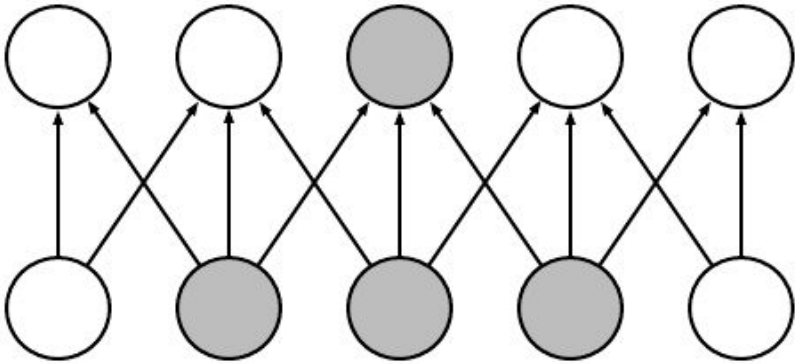
Convolutions as Sparse Connections

We can think of a convolution as a **sparse matrix multiplication** with **shared parameters**



Fully connected layer

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix}$$



Convolutional layer

$$\begin{bmatrix} w_2 & w_3 & 0 & 0 & 0 \\ w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \\ 0 & 0 & 0 & w_1 & w_2 \end{bmatrix}$$

Convolutional Modules

The most common convolutional factory produces 2D convolutional modules that have 3D inputs and outputs where the third dimension is a number of **channels** that are summed over:

$$\text{module} \sim \text{Conv2D}(c_{\text{in}}, c_{\text{out}}, d_1, d_2)$$

$$x' = \text{module}(x)$$

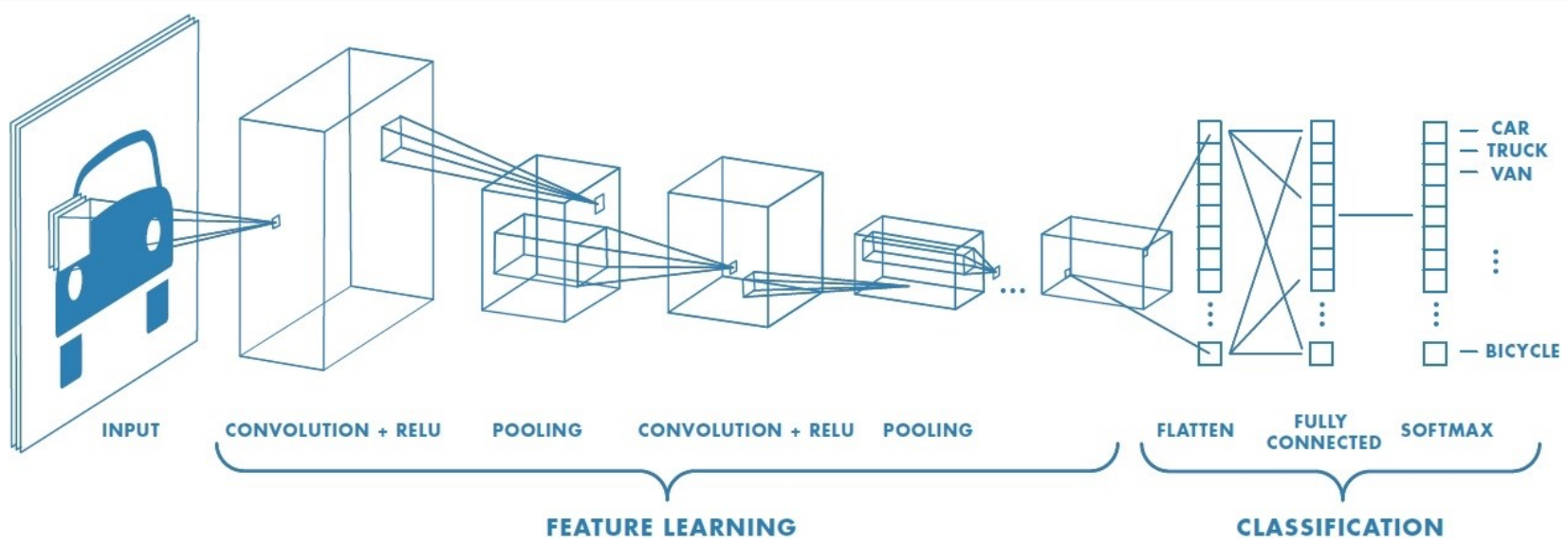
$$x'_{i'j'k'} = \sum_{i=1}^{d_1} \sum_{j=1}^{d_2} \sum_{k=1}^{c_{in}} w_{ijkk'} \, x_{i'+i-1,j'+j-1,k} \quad \forall i', j', k'$$

Their are a number of variants on this such as including a bias for each output channel, padding the edges of the input (e.g. with zeros) so that the output is the same size, and introducing a **stride**, wherein the filter is moved by multiple indices at a time

A **depthwise spatial convolution** instead applies a separate convolution to each channel (i.e. $w_{ijkk'} = 0$ if $k \neq k'$)

CNNs

The traditional CNN setup has a mixture of convolutional and max pooling layers to learn features, before finishing with one or more fully connected layers to do the final prediction[2]



Compared with MLPs, this reduces the number of parameters, thereby reducing both memory and computational costs; ultimately this allows us to train deeper networks

[2]Some modern large CNNs forgo the fully connected layers

Spatial Invariances

The other main motivation for using a mix of convolutional and max pooling layers is that it can naturally induce spatial invariances to where objects are in an image

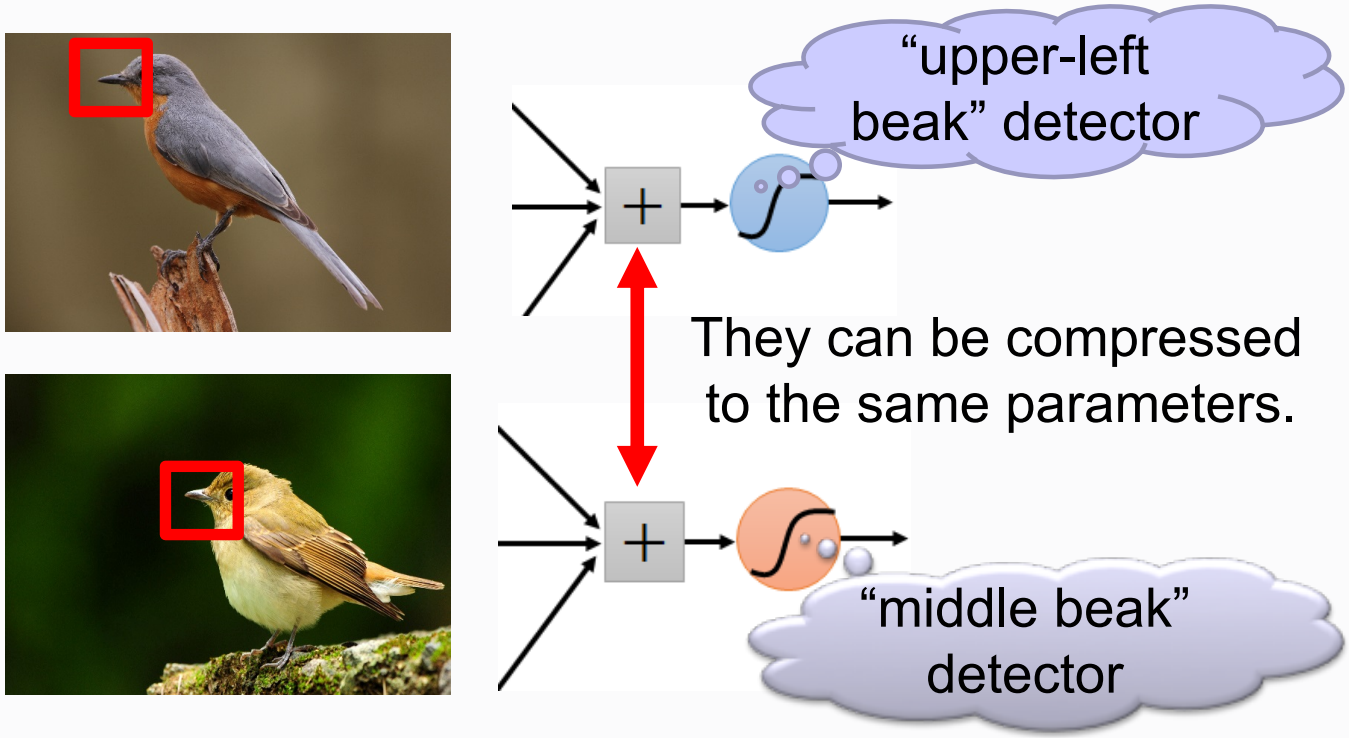
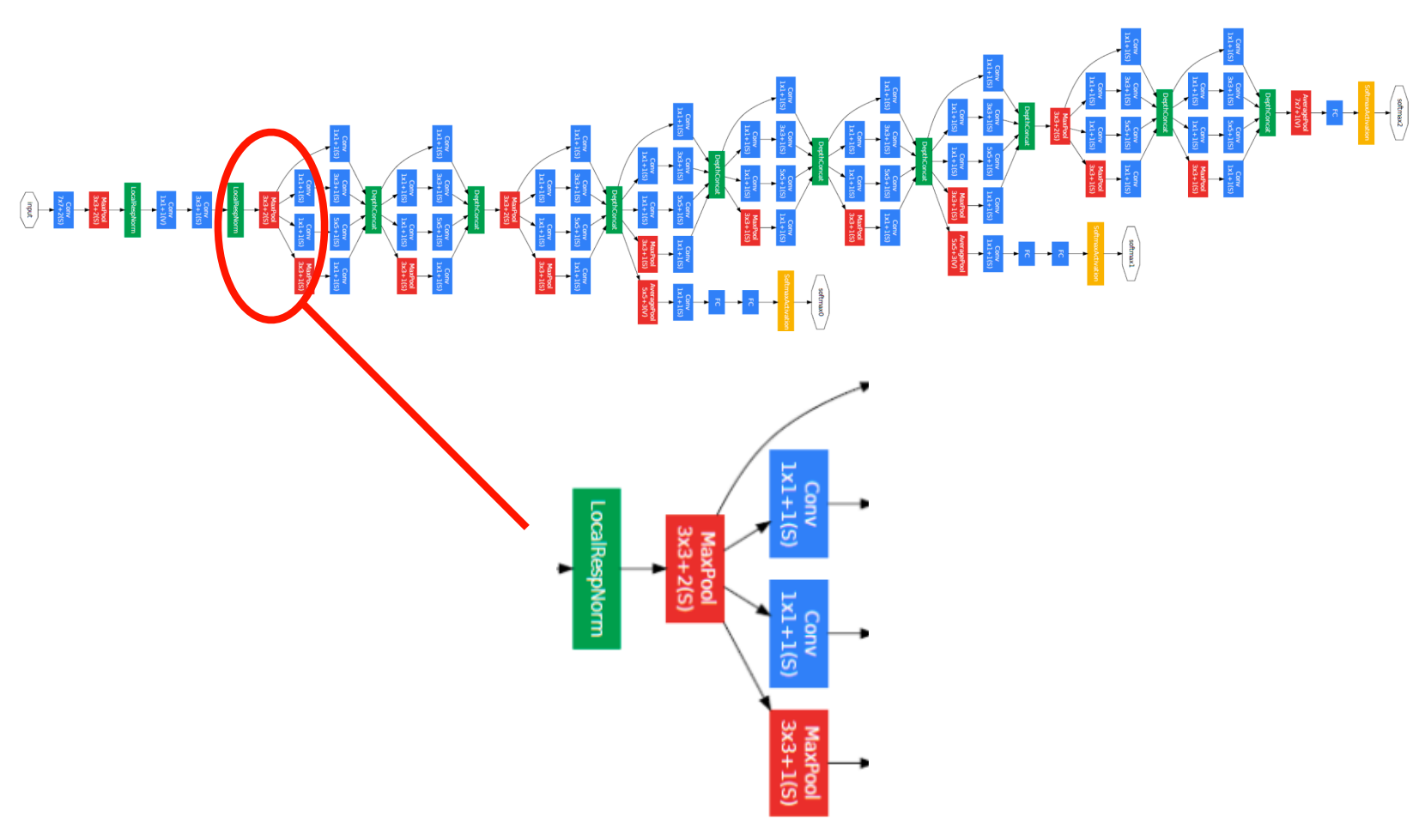


Figure Credit: https://cs.uwaterloo.ca/~mli/Deep-Learning-2017-Lecture5CNN.ppt

Improving CNNs

Example Architecture: GoogleNet

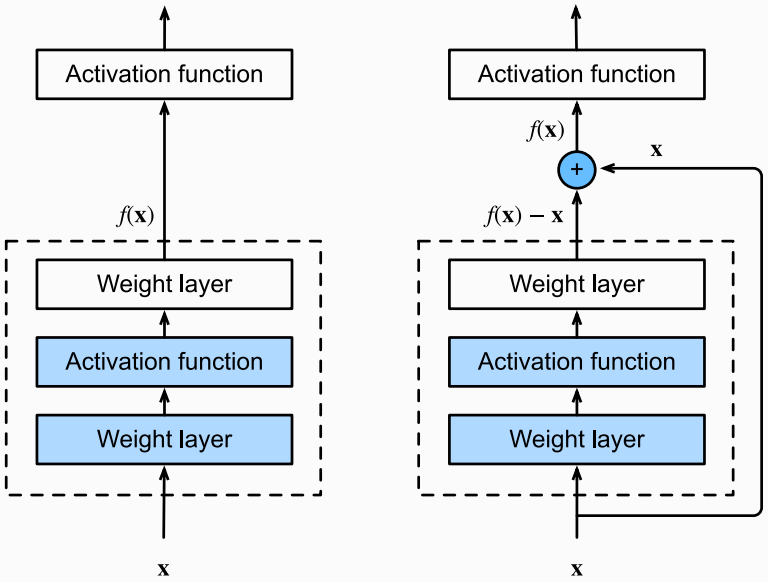


Going Deeper With Convolutions. Szegedy et al. CVPR 2015

Getting Too Deep: ResNets and DenseNets

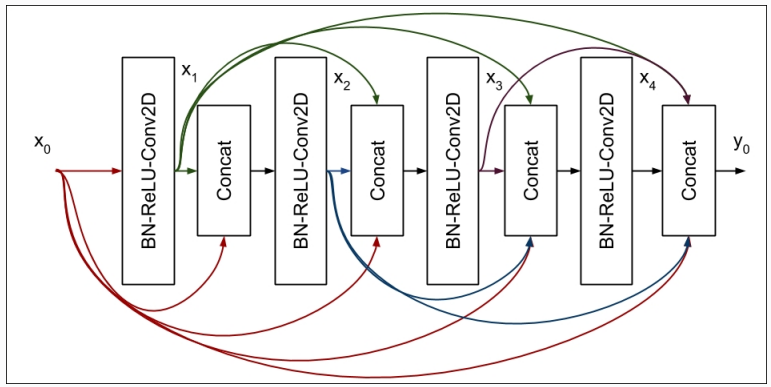
Making networks too deep can cause difficulties with training; we can end up with worse empirical risks at train (and test) time

ResNets and **DenseNets** use **skip connections** to help alleviate this issue and allow deeper networks to be trained effectively



Left: regular block. Right: residual block.

Source: `https://www.d2l.ai/chapter_convolutional-modern/resnet.html`



DenseNet blocks replace the addition with a concatenation. Image Credit: Rowel Atienza

Recurrent Neural Networks (RNNs)

- Data is often sequential and exploiting this sequential structure can be essential for accurate prediction
  - Time series, text, audio, video
- We may also need to make predictions "online," such that we must predict each output given only the sequence so far
- Even for non–sequential, fixed–dimensional data, we may want to use sequential reasoning processes
  - "Attention" for image data
- Recurrent neural networks (RNNs) are a class of architectures that allow us to deal with such sequential settings by processing inputs and outputs in a sequence
- They have a very wide range of applications and are particularly prominent in natural language processing

Basic RNN Framework

Imagine we have some input sequence $x_1, x_2, \ldots, x_\tau$ and want to predict a corresponding sequence of outputs $y_1, y_2, \ldots, y_\tau$. Further assume that we will only have access to $x_{1:t}$ when predicting y_t.

RNNs introduce a sequence of **hidden states** h_t that represent the sequence history $x_{1:t}$ and which can be calculated recursively as

$$h_t = f_e(h_{t-1}, x_t)$$

Here f_e is known as an **encoder** module and its parameters are generally shared between points in the sequence, i.e. the same f_e is used for each t

Prediction is then performed from the hidden state at each step in sequences using a **decoder** f_d, such that $\hat{y}_t = f_d(h_t)$

A Simple Example RNN Architecture

$$f_e \sim \text{MLP}$$

$$f_d \sim \text{MLP}$$

$$h_1 = f_e(0, x_1)$$

$$h_t = f_e(h_{t-1}, x_t) \qquad \forall t \in \{2, \ldots, \tau\}$$

$$\hat{y}_t = f_d(h_t) \qquad \forall t \in \{1, \ldots, \tau\}$$

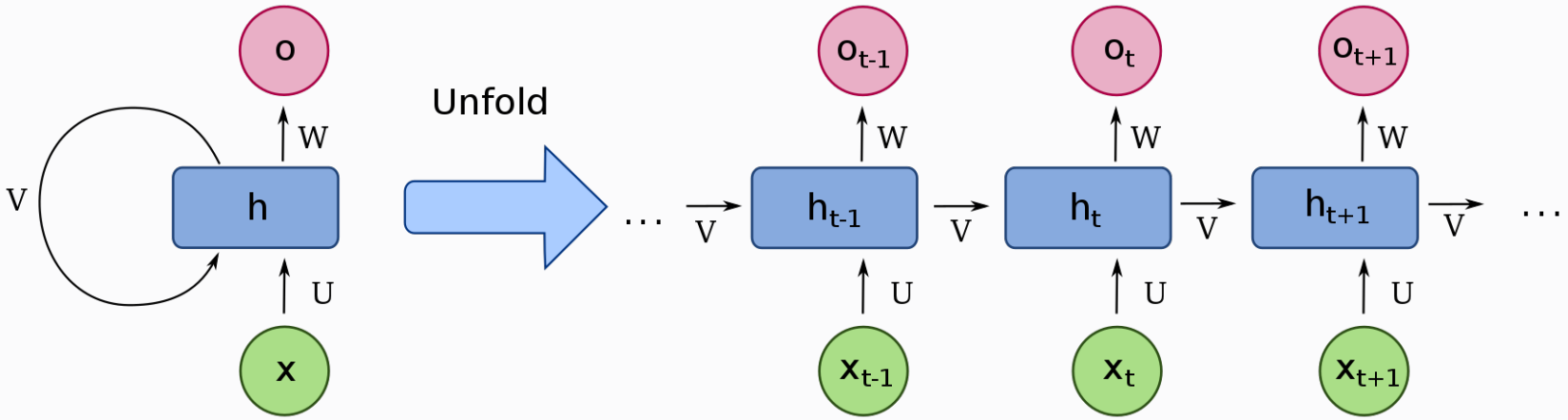


Image Credit: fdeloche, https://commons.wikimedia.org/w/index.php?curid=60109157

Bidirectional RNN

Sometimes we want predictions to depend on all the inputs rather than just the inputs so far

We can deal with this by using a **bidirectional** RNN that has a two sets of hidden states, one in each direction

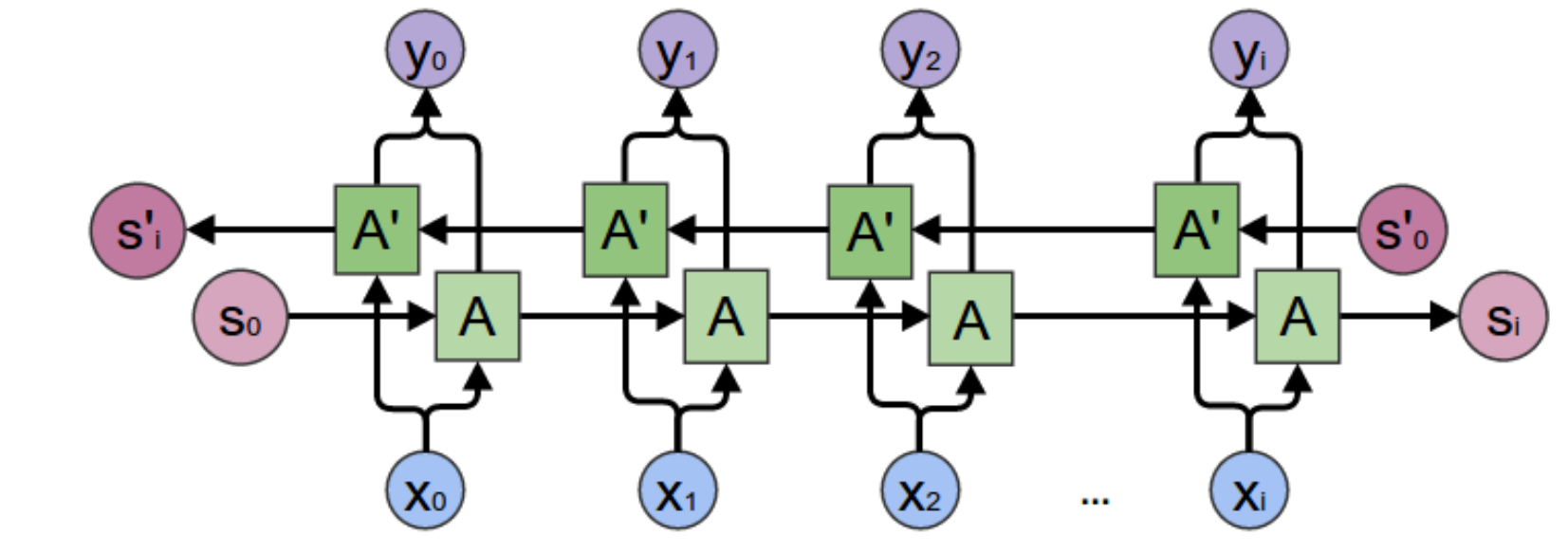


Image Credit: `http://colah.github.io/posts/2015-09-NN-Types-FP/`

Dealing with Varying Input and Output Lengths

Input data and predictions can also be of varying size

- Text translation/generation, incomplete data, videos/audio of varying length, identifying multiple objects in an image

Careful setups of RNNs allow us to deal with this, e.g. using "end of sentence" as a possible input and output for text data which triggers changepoint behavior

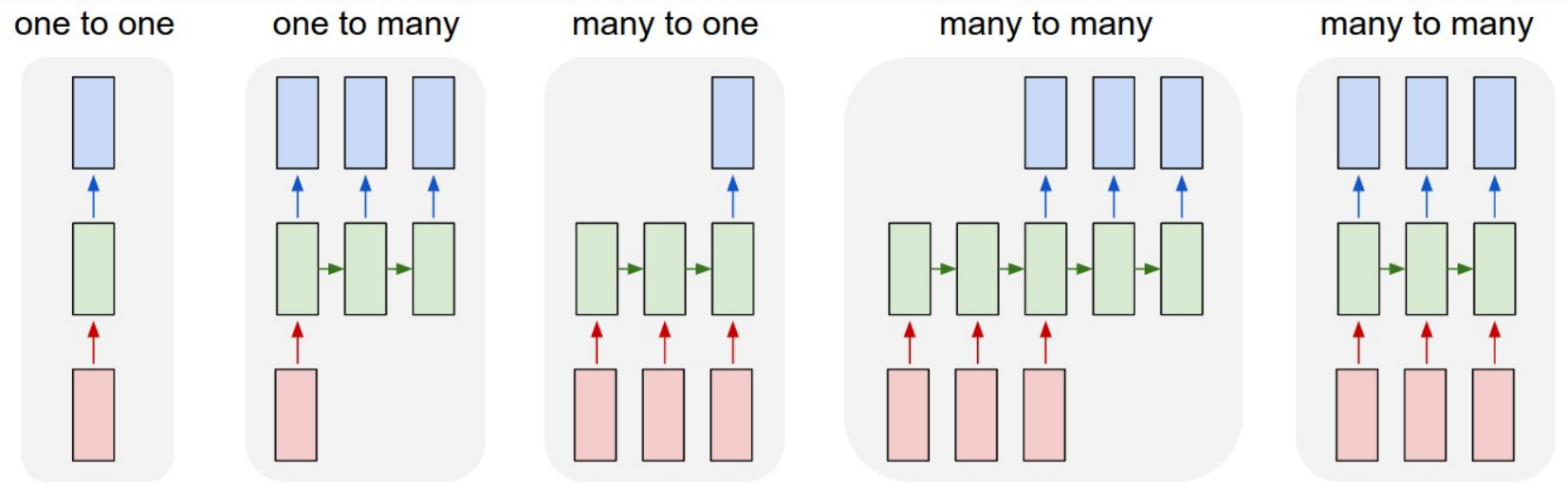
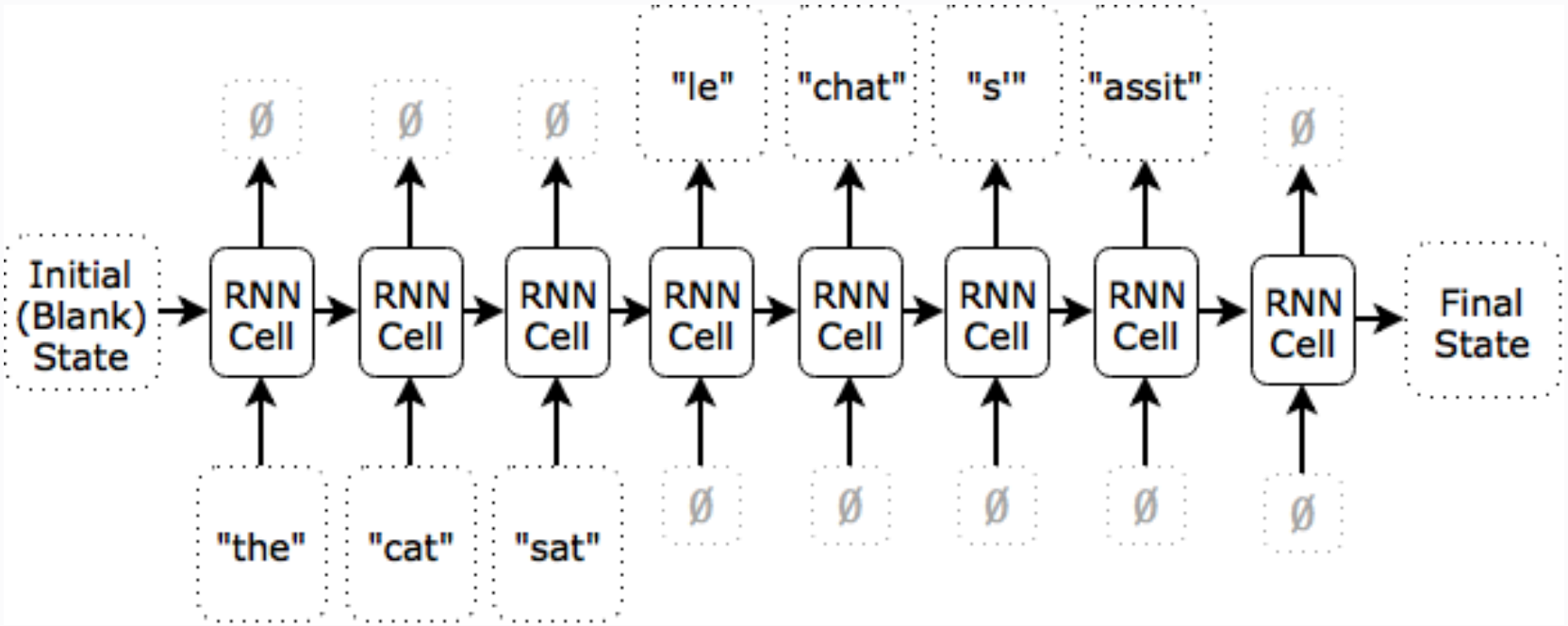


Image Credit: Andrej Karpathy

Example Application: Text Translation



Credit: `https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html`

Further Reading

- Interactive and up–to–date online book on deep learning with code examples etc: `https://www.d2l.ai/index.html`
- Chapters 9 and 10 of Ian Goodfellow, Yoshua Bengio, and Aaron Courville. **Deep Learning**. `http://www.deeplearningbook.org`. MIT Press, 2016
- Deep learning software tutorials (note that you can play around with these directly in your browser without installing anything)
  - PyTorch: `https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`
  - Tensorflow: `https://www.tensorflow.org/tutorials`
- Stanford course on deep learning: `https://youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv` (Lectures 5, 9, and 10 of particular relevance for this lecture)