Applications of Neural Networks to strings and sequences

Lecture 8.2

by Marina Barsky

Convolutional Neural Networks

CNNs

Problem with number of layers and number of neurons

- The number of parameters in NN grows very fast as we add more layers
- If a single input vector contains *n* features, and we create another layer of size *m* - we add *n**m* additional weights edges

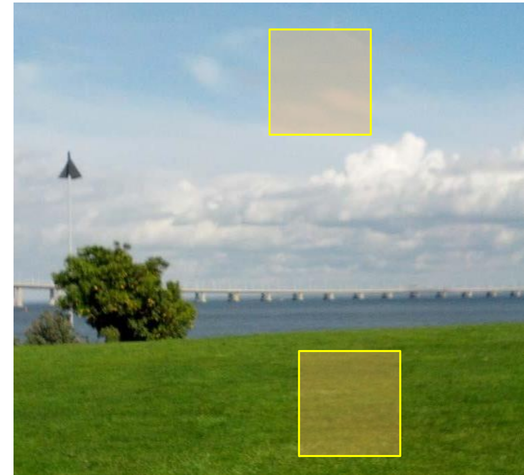
  For n=1000 and m=1000 we need to learn 1,000,000 additional weights

- Optimizing such big models is computationally expensive
- When our training examples are images, the input is already very high-dimensional

- If we use MLP with more than 3 layers - the optimization problem quickly becomes intractable

Recognizing local regions of the image

- Pixels that are close to each other usually represent the same type of information: sky, water, leaves, fur, bricks... This information can likely be combined into a smaller number of features



- The parts of an image where two different types of information “touch” one another - represent a shape

- The idea is to train the neural network to recognize regions of the same information as well as the edges, and use these learned features as new less-dimensional vectors for classifying images

Detecting image patterns

- We can split the image into *square patches* using a sliding frame approach
- We can then train multiple smaller models at once, each small model receiving a square patch as input
- The goal of each small model is to learn to detect a specific kind of pattern in the input patch
- For example, one small model will learn to detect the sky, another one will detect the grass, the third one will detect edges of a building...

The “filter” idea

- If we train a network using a set of labeled images (say, set of cats) the model will learn a set of local patterns which are most common to all cats
- We call these local patterns *filters*
- The CNN learns the filter shapes on its own during training - all you need to decide is the size of each filter, and the network will learn what each filter should look like
- Later, during classification, it will apply each filter to a new image and compute the output - image class

Comparing input patch to a filter

- Once you decide on the size of each filter, the regular training will produce the optimal values for each filter matrix
- Let's assume - for simplicity - that the input image is black and white, with 1 representing black and 0 representing white pixels
- Assume that our patches are 3 by 3 pixels (p = 3). Some patch could then look like matrix P (for "patch")
- Let's say we want to detect a pattern called "cross" in the image data. We initialize the matrix F (filter) with some random values

Patch in a black-and-white image

$$P=\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Filter initialized

$$F=\begin{bmatrix} 0 & 2 & 3 \\ 2 & 4 & 1 \\ 0 & 3 & 0 \end{bmatrix}$$

Learning matrix F

- The small regression model that will detect "cross" patterns (and only them) would need to learn a 3 by 3 parameter matrix F where parameters at positions corresponding to the 1s in the input patch would be positive numbers, while the parameters in positions corresponding to 0s would be close to zero

Patch in a black-and-white image

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Filter: to be learned from data

$$F = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 4 & 1 \\ 0 & 3 & 0 \end{bmatrix}$$

Learning matrix F

- The learning proceeds by calculating the dot-product between matrices P and F and then summing up all values from the resulting vector
- The higher the value, the more similar F is to P:

$P \cdot F = [0 \cdot 0 + 2 \cdot 1 + 0 \cdot 0,\ 2 \cdot 1 + 4 \cdot 1 + 3 \cdot 1,\ 3 \cdot 0 + 1 \cdot 1 + 0 \cdot 1] = [2, 9, 1]$

The sum is $2 + 9 + 1 = 12$

- This operation — **the dot product between a patch and a filter and then summing the values** — is called *convolution*

Patch in a black-and-white image

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Filter: randomly initialized

$$F = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 4 & 1 \\ 0 & 3 & 0 \end{bmatrix}$$

Optimizing (matching) filter to a patch

- If our input patch P had a different pattern - then the convolution would give a lower result: 0 + 9 + 0 = 9
- The more the patch “looks” like the filter, the higher the value of the convolution is
- There’s also a bias parameter b associated with each filter F which is added to the result of a convolution before applying the nonlinearity

Patch in a black-and-white image

P=

1	1	1
0	1	0
0	1	0

Filter: randomly initialized

F=

0	2	3
2	4	1
0	3	0

Convolutional Neural Network (CNN)

- A *Convolutional Neural Network* (CNN) is a special kind of Feed Forward NN that significantly reduces the number of parameters in a deep neural network, by converting multi-dimensional data into smaller convolved features
- CNNs are used in image and text recognition where they beat many previously established benchmarks

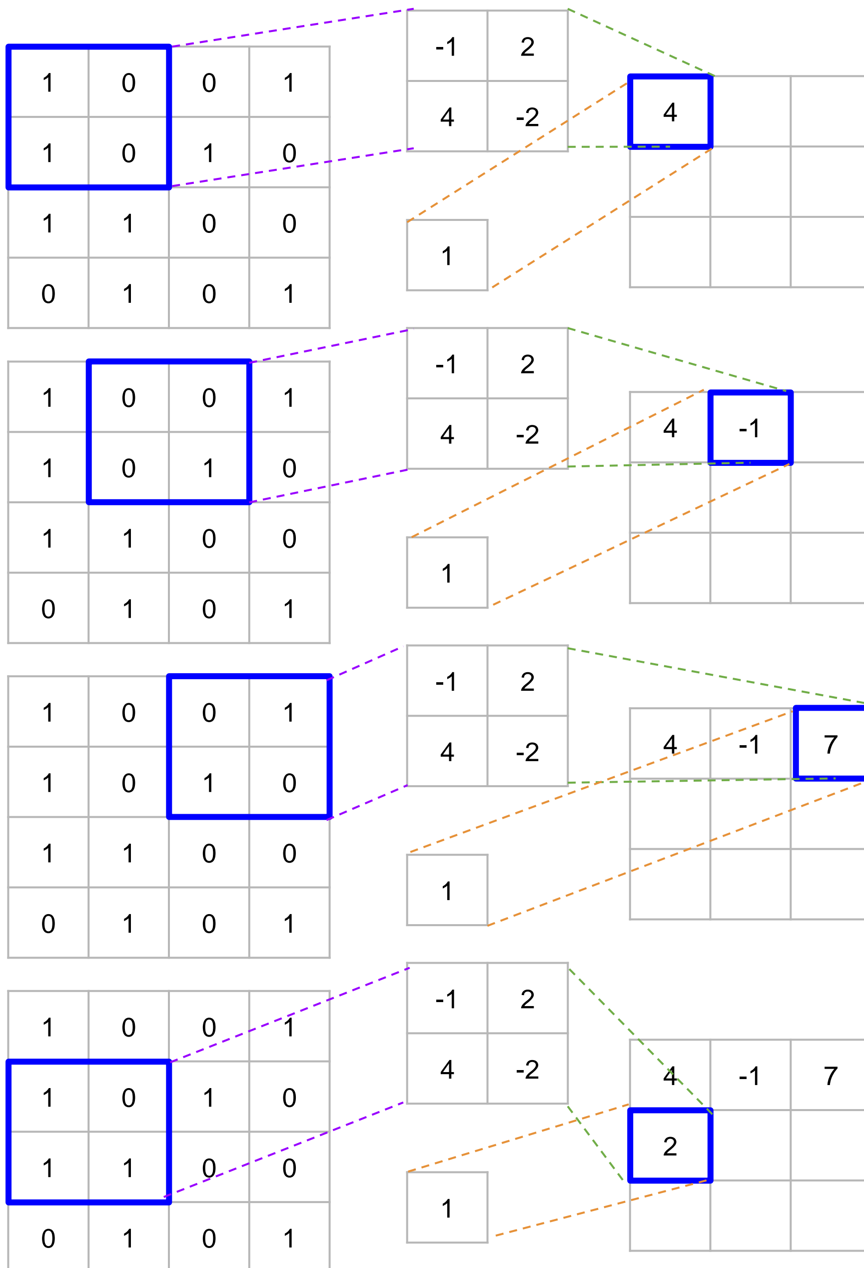
Computing convolution for the entire image

- One hidden layer of a CNN consists of multiple convolution filters (each with its own bias parameter), just like one layer in a vanilla NN consists of multiple units

- Each filter of the first (leftmost) hidden layer slides — or *convolves* — across the input image, left to right, top to bottom, and convolution is computed for each sliding input frame

- So each neuron in a CNN layer is a filter which learns a single pattern. The number of such units generally is much smaller than the number of original pixels

Filter convolving across the image

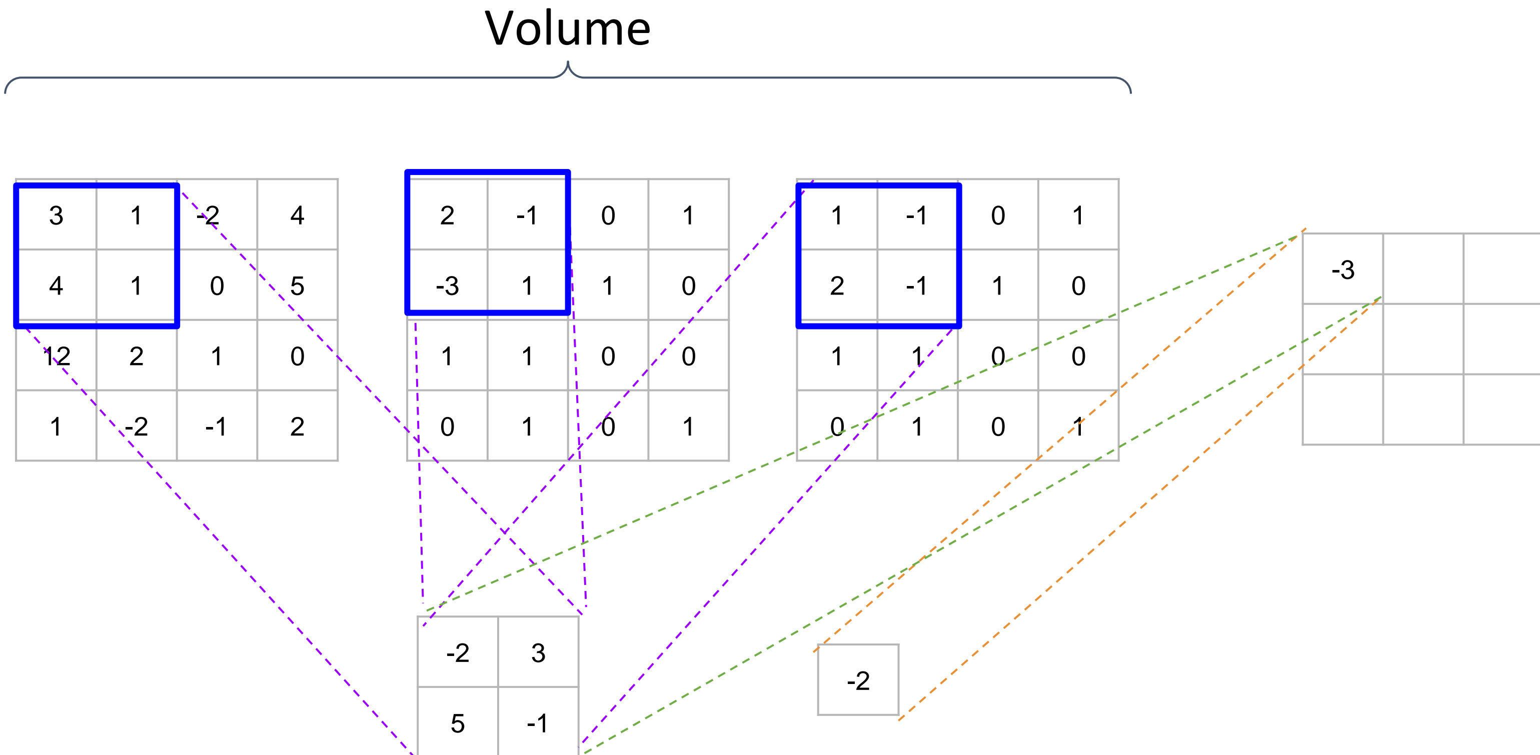


- As a result of this convolution, instead of a 4x4 image matrix we get a 3x3 matrix
- This matrix becomes an input to the next hidden layer
- A nonlinearity (ReLU) is applied to the [sum of the convolution plus the bias]
- The numbers for each filter matrix F and the value of the bias term b, are found by the gradient descent with backpropagation

Volume: collection of matrices

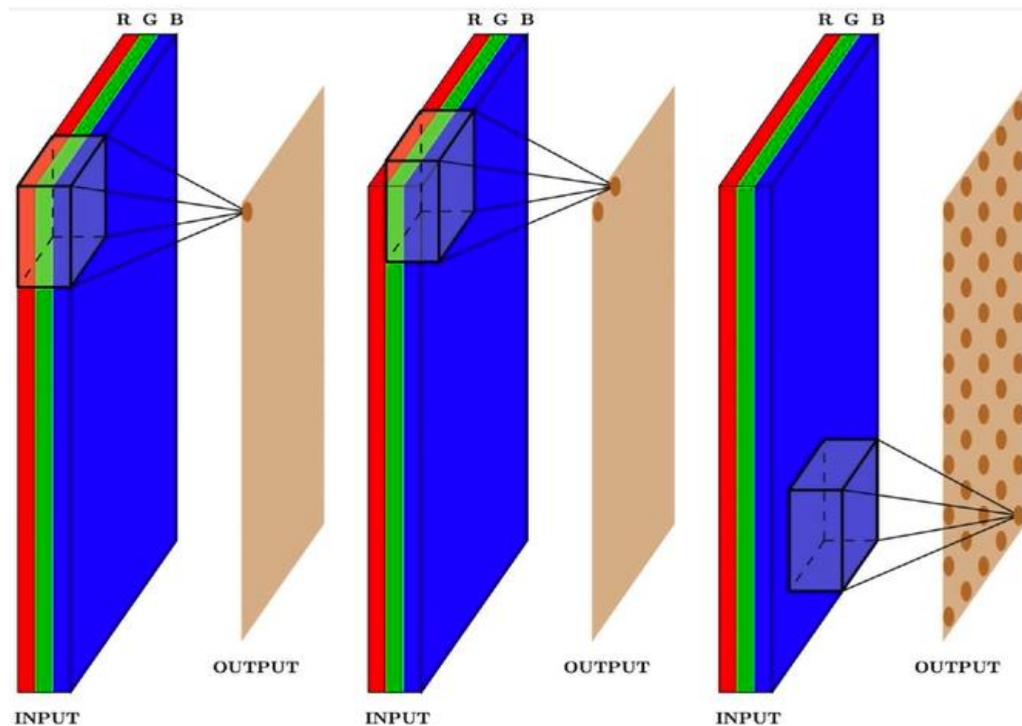
- Each layer with m filters produces m matrices of size pxp which serve as an input for the next hidden layer
- If the next layer is also a convolution layer, then layer $i + 1$ treats the output of the preceding layer i as a collection of m images
- Such a collection is called a *volume*. Each filter of layer $i + 1$ convolves the whole volume produced by filter i
- The convolution of a patch of a volume is simply the sum of convolutions of the corresponding patches of individual matrices in this volume

The volume convolves as a single input



(-2 · 3 + 3 · 1 + 5 · 4 + -1 · 1) +(-2 · 2 + 3 · (-1) + 5 · (-3) + -1 · 1) + (-2 · 1 + 3 · (-1) + 5 · 2 + -1 · (-1)) + (-2)= -3

Input image as a volume of 3 color channels



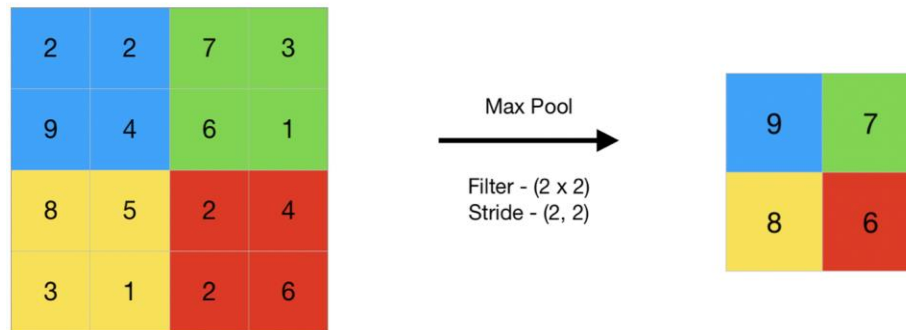
CNNs also often get volumes as input, since an image is usually represented by three channels: R, G, and B, each channel being a monochrome picture

CNN: parameters

This is just a very high-level picture of the CNN architecture

Other essential features include strides, padding, and pooling

- Strides and padding are hyperparameters of the convolution filter and the sliding window
- Pooling example: max pooling



All this is designed to reduce the number of parameters of a CNN even more

Sequence classification with CNN

CNN for sequence classification

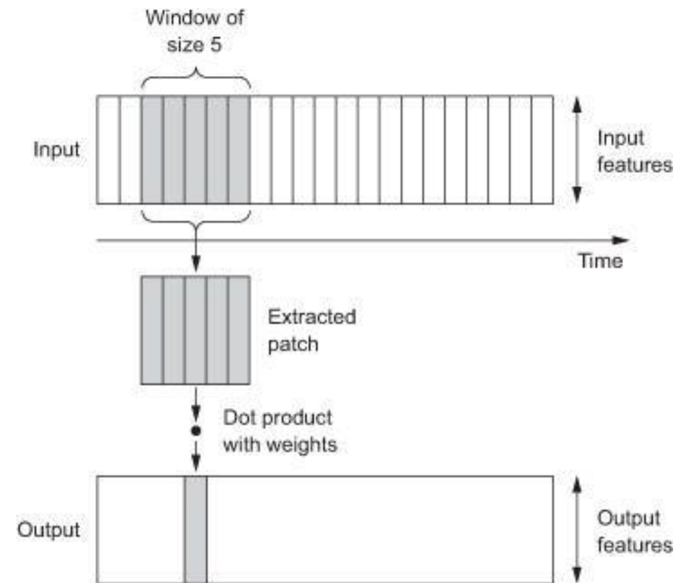
- We can use the same idea with sequence data:

  The sequence can be thought of as a 1D image, where time is treated as a spatial dimension (like the height or width of a 2D image)

- CNNs are often used for:
  - audio generation
  - machine translation
  - text classification
  - timeseries forecasting

1D convolution

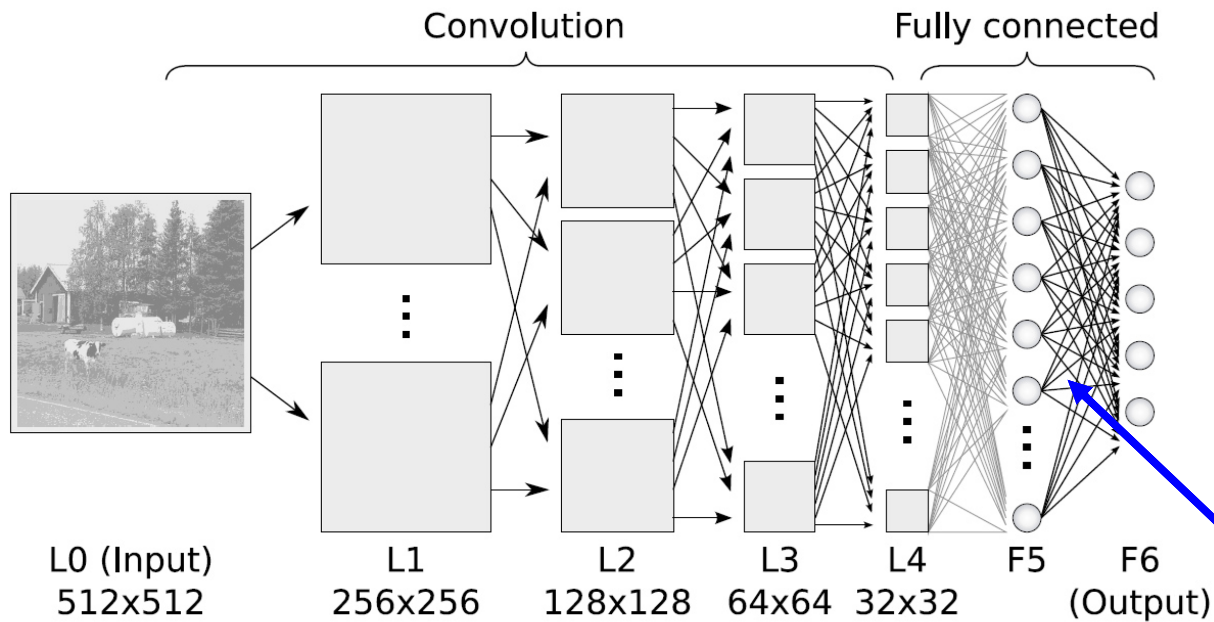
- In 2D convolutions we extracted 2D patches from image tensors
- In the same way, we can use 1D convolutions, extracting local 1D patches (subsequences) from sequences



- Such 1D convolution layers can recognize local patterns in a sequence
- Because the same input transformation is performed on every patch, a pattern learned at a certain position in a sentence can later be recognized at a different position

  For instance, a 1D CNN which is using convolution windows of size 5 is able to learn words or word fragments of length 5 or less, and is able to recognize these words in any context in an input sequence

Demos of using CNN for image and sequence classification: here



What is going on inside?

What does the network learn about images?

How and what does it learn?

The exact way neural networks see and interpret the world remains a black box

We want to better understand of how exactly they recognize specific patterns or objects in order to:

- improve the quality of NN learning
- solve legal problems since in many cases the outputs have to be interpretable by humans

How CNN sees images after learning

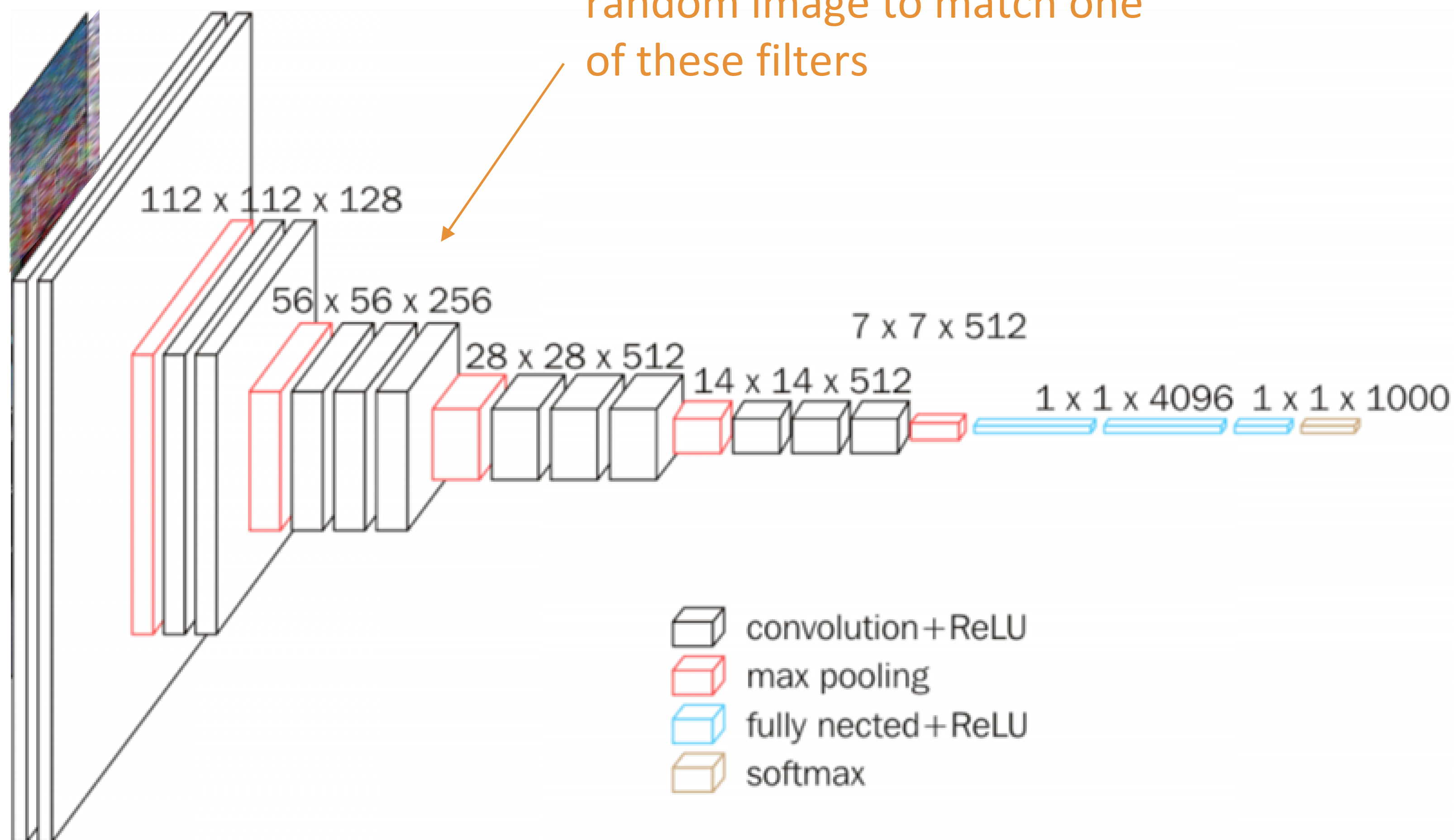
- Neural networks learn to transform images into successive layers of increasingly meaningful and complex representations (filters)
- We can think of a deep network as a "multistage information-distillation operation, where information goes through successive filters and comes out increasingly purified". (François Chollet, "Deep Learning with Python")
- We can generate patterns that maximize the mean activation of a chosen feature map in a certain layer

Experiment: understanding learned patterns (filters)

- Get a network which already learned to recognize thousands of image types
  - Pre-trained model is available from *torchvision.models*
  - Data is from http://www.image-net.org/
  - The network is VGG-16

- The goal is NOT to train the model, but use it in an evaluation mode
- Then show an image with random pixels to the model and optimize the pixel values to best match each filter at each hidden level (Erhan, D. et al. “Visualizing Higher-Layer Features of a Deep Network”, 2009).

VGG-16



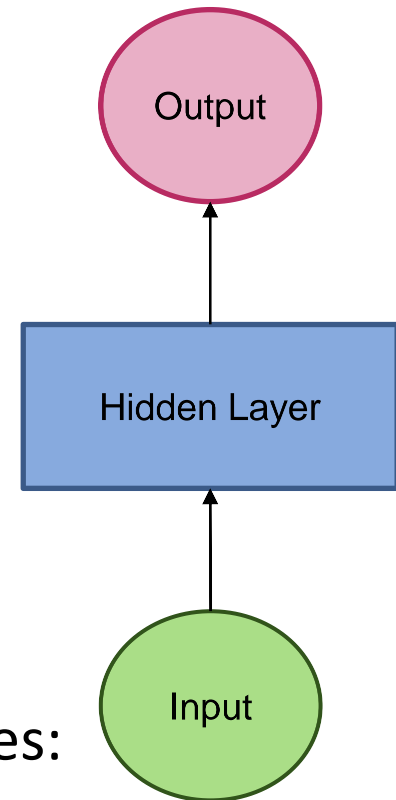
Here is a blog and the code

Recurrent neural Networks (RNN)

Basics

Networks that model sequences

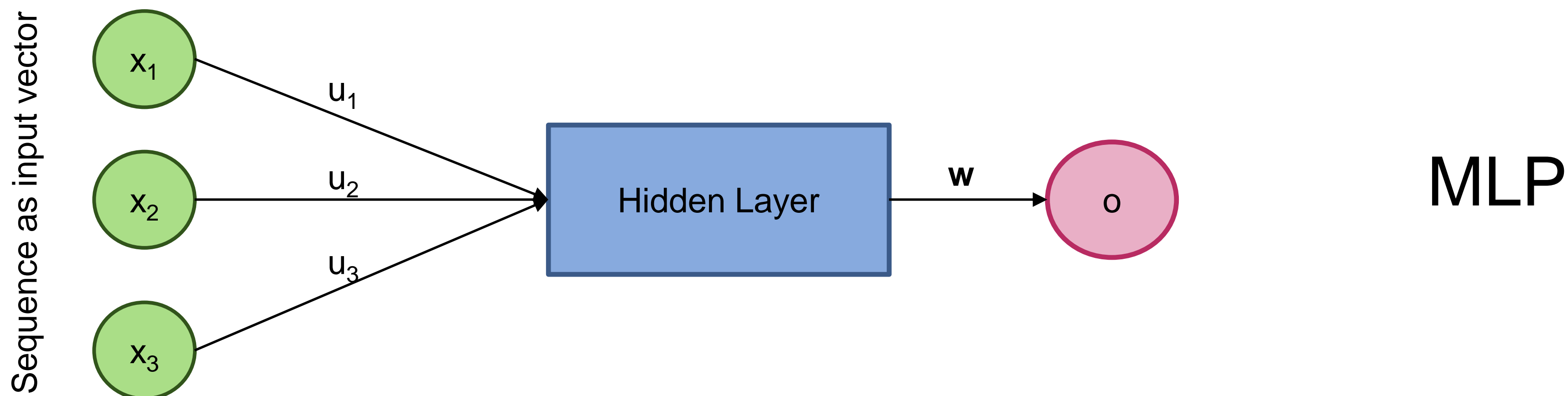
- Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step
- The main feature of RNN is that its hidden layer remembers the information about a sequence seen so far: it accumulates the information

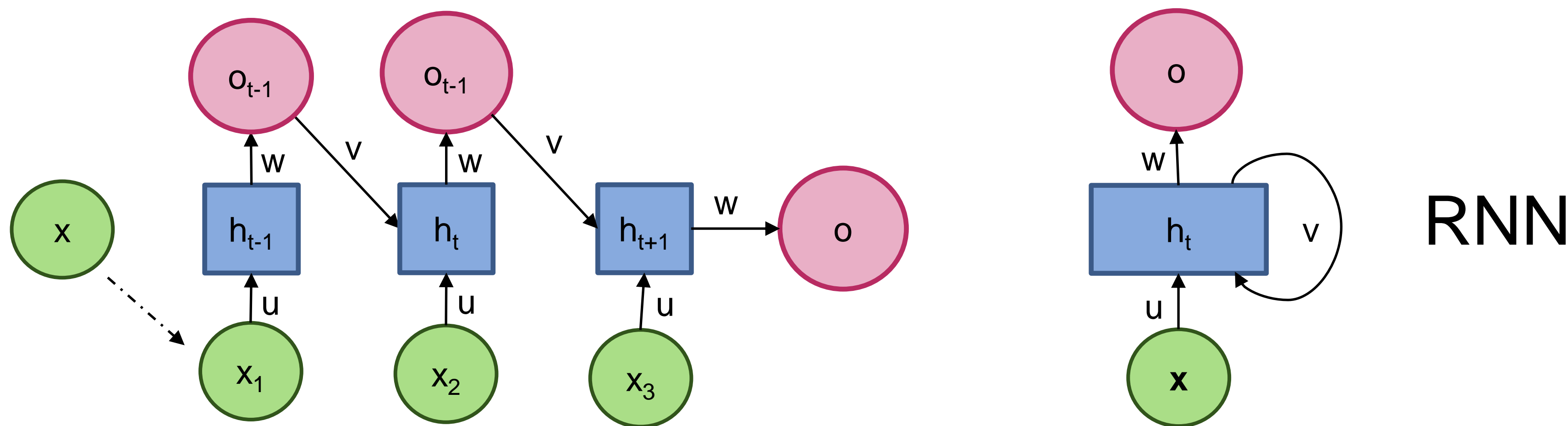


Recurrent Neural Network has three major states:

- Input state
- Output state
- **Recurrent state**, which is a chain of hidden states, which accumulates all the knowledge about the sequence seen so far

RNN vs. Multi-Layer Perceptron (MLP)





How RNN works

- RNN has a “memory” which remembers all information about what has been seen so far
- Unlike MLP where each symbol of the input vector has its own weight, RNN uses the same weights for each symbol as it performs the same task on all the symbols
- RNN converts independent activations into dependent activations by providing the same weights and biases to all the layers, thus reducing the complexity of increasing parameters and memorizing each previous output by giving each output as input to the next hidden layer

Training RNN

- A single-time step of the input is provided to the network.
- It calculates its current state using a current input and the previous state.
- The current h_t becomes h_{t-1} for the next time step
- We can go as many time steps as there are symbols in the input and join the information from all the previous states.
- Once all the time steps are completed, the final current state is used to calculate the output.
- The output is then compared to the actual output i.e the target output and the error is generated.
- The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

Current state depends on the current symbol and the previous state

- The formula for calculating current state:

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

where:
h_t -> current state
h_{t-1} -> previous state
x_t -> input state

Same activation for all neurons in the recurrent layer

- Formula for applying Activation function(tanh):

$$h_t = \tanh (W_{hh} h_{t-1} + W_{xh} x_t)$$

where:
w_{hh} -> weight at recurrent neuron
w_{xh} -> weight at input neuron

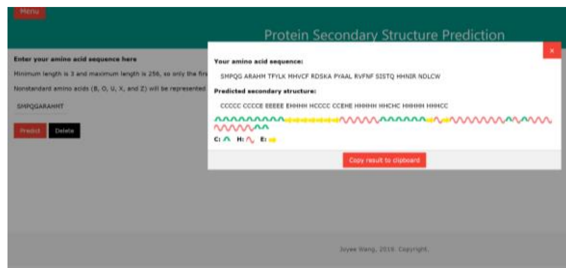
Output: as before

- The formula for calculating output:

$$y_t = W_{hy}h_t$$

y_t -> output
W_{hy} -> weight at output layer

Sample student project with RNN
(Bidirectional Long Short-Term Memory Network)



[Protein structure predictor]

by Joyee Wang

Sample aminoacid sequence for input:

SMPQGARAHMTFYLKMHVCFRDSKAPYAALRVFNFSISTQHHNIRNDLCW

RNN: real-life example plus code: link

Readings

- The 100 Page ML book by Andriy Burkiv: link
  Chapter 6 (pp. 74 – 78) – deep learning
  Chapter 6.2.1. (p. 78) – CNN
  Chapter 6.2.2. (p. 82) – RNN

- Grokking Deep Learning by Andrew Trask: link
  Chapters 1 – 6 (at least)

Comparison

- Interesting discussion about HMM vs. RNN: link
- Paper on the same topic: link