Inteligencia Artificial

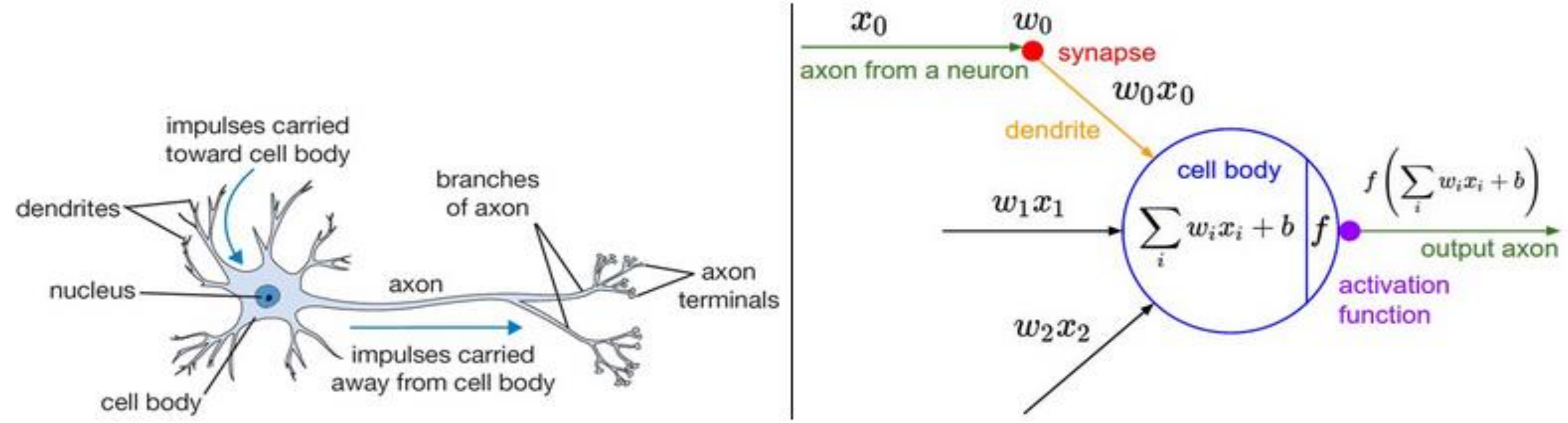
UTN – FRVM

5º Año Ing. en Sistemas de Información

Agenda

- Neural Networks
  - Activation Functions
  - Architecture
  - Gradient Descent
  - ConvNets

Biological motivation and connections



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Each neuron performs a dot product with the input and its weights, adds the bias and applies the non-linearity (or activation function), in this case the sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

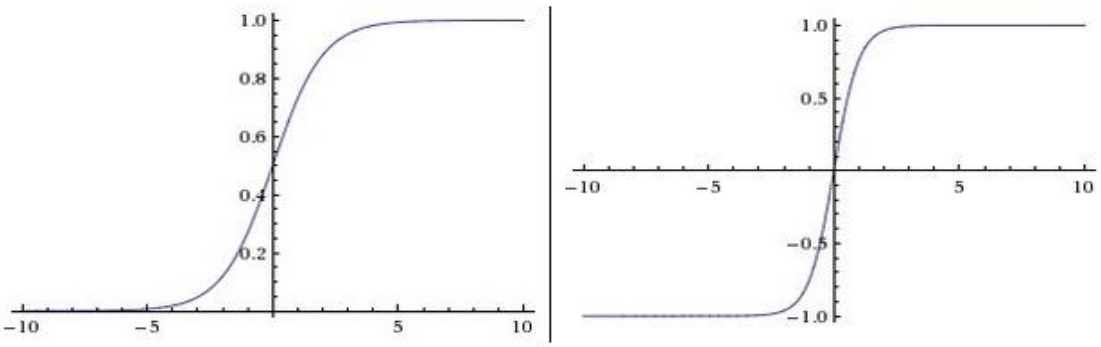
Coarse model?

https://www.sciencedirect.com/science/article/pii/S0959438814000130

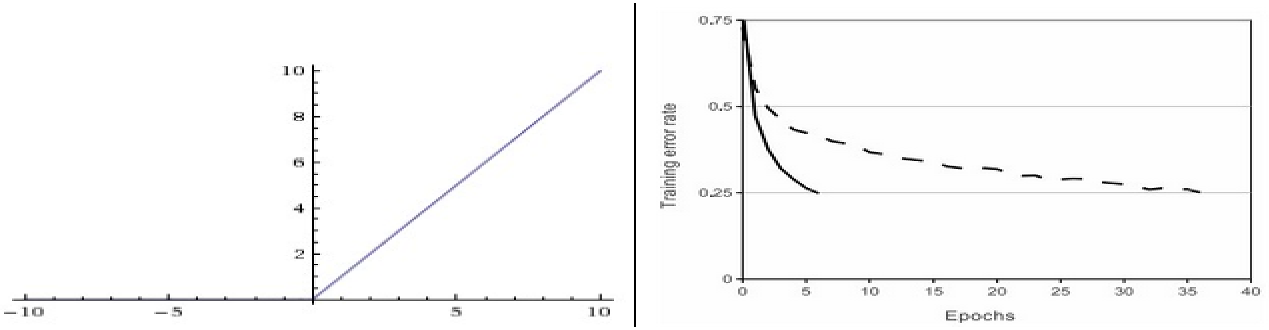
https://neurophysics.ucsd.edu/courses/physics_171/annurev.neuro.28.061604.135703.pdf

Commonly used activation functions

- Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions in practice:



Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right**: The tanh non-linearity squashes real numbers to range between [-1,1].



Left: Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. **Right**: A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

Activation functions

- **Sigmoid.** The sigmoid non-linearity has the mathematical form $\sigma(x) = 1/(1+e^{-x})$
  - *Sigmoids saturate and kill gradients*
  - *Sigmoid outputs are not zero-centered*
- The **tanh** non-linearity squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. The tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2\sigma(2x) - 1$.
- **ReLU**. The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero. There are several pros and cons to using the ReLUs.

(+) It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions.
(+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.),
the ReLU can be implemented by simply thresholding a matrix of activations at zero.
(-) Unfortunately, ReLU units can be fragile during training and can “die”.

Leaky ReLU/Maxout

- ***Leaky ReLU***. Leaky ReLUs are one attempt to fix the “dying ReLU” problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes ***f(x)=1(x<0)(αx)+1(x>=0)(x)*** where α is a small constant.

- ***Maxout***. One relatively popular choice is the *Maxout* neuron that generalizes the ReLU and its leaky version. The Maxout neuron computes the function

$$\max(w_1^T x + b_1, w_2^T x + b_2).$$

- Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have w1,b1=0). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

ReLU vs. Other Functions

- Use the ReLU non-linearity, be careful with the learning rates and possibly monitor the fraction of “dead” units in a network. Then, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

Neural Network architectures: Layer-wise organization

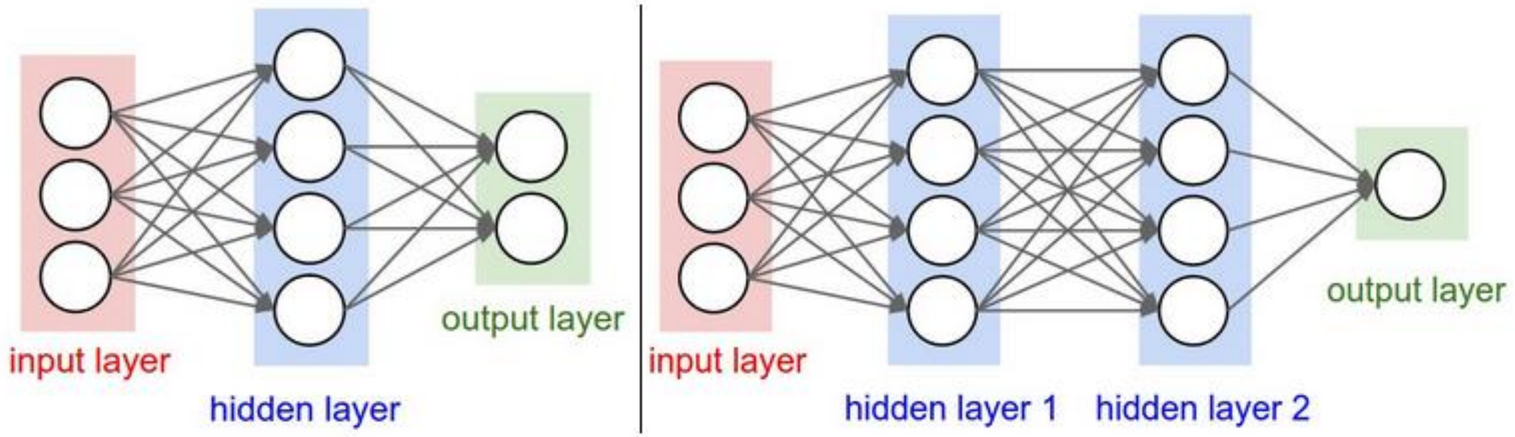
- **Neural Networks as neurons in graphs**. Neural Networks are modeled as collections of neurons that are connected in an acyclic graph.

- In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network.

- Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons.

- For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.

NN Architectures



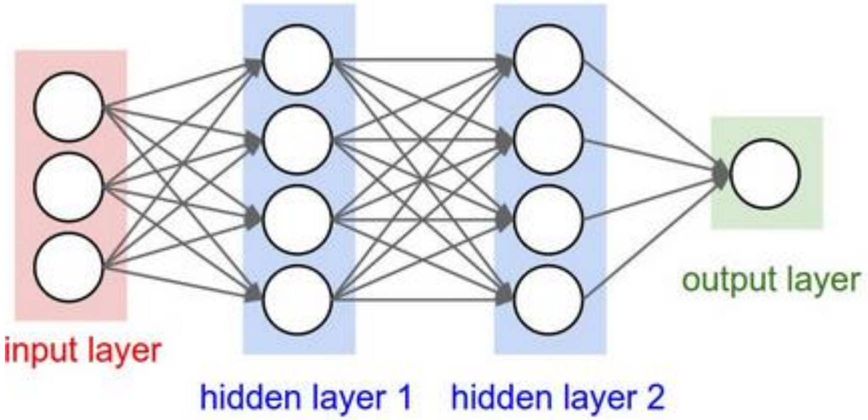
Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right**: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

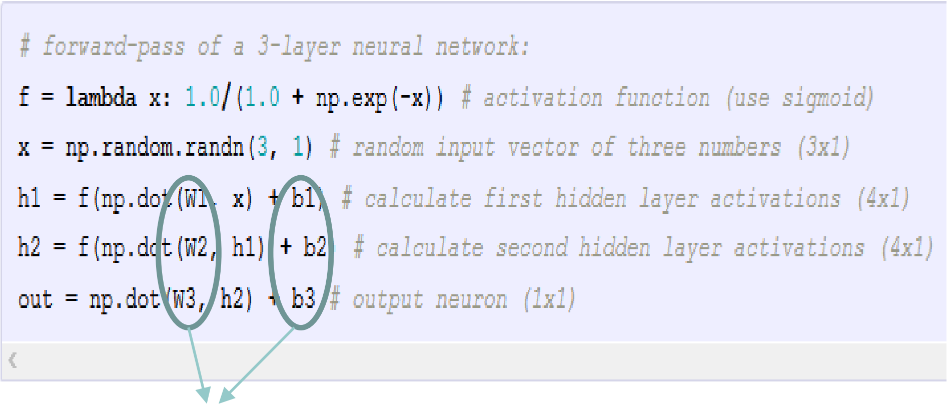
Output layer. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).

Sizing neural networks

- The two metrics commonly used to measure the size of neural networks are the number of neurons, or more commonly the number of parameters. Working with the two example networks :
- The first network (left) has 4 + 2 = 6 neurons (not counting the inputs), [3 x 4] + [4 x 2] = 20 weights and 4 + 2 = 6 biases, for a total of 26 learnable parameters.
- The second network (right) has 4 + 4 + 1 = 9 neurons, [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters.
- To give context, modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence *deep learning*). However, as we will see the number of *effective* connections is significantly greater due to parameter sharing

Example feed-forward computation





```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

parameters

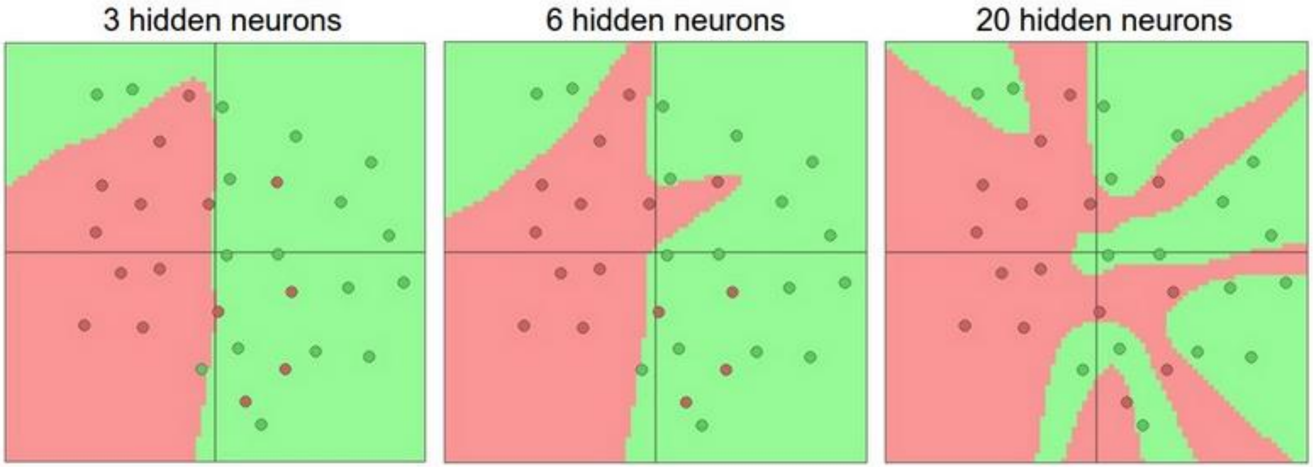
- One of the primary reasons that Neural Networks are organized into layers is that this structure makes it very simple and efficient to evaluate Neural Networks using matrix vector operations.
- Working with the example above, the input would be a **[3x1] vector**. All connection strengths for a layer can be stored in a single matrix.
- The first hidden layer's weights **W1 would be of size [4x3]**, and the biases for all units would be in the vector **b1**, of size **[4x1]**. Here, every single neuron has its weights in a row of **W1**, so the matrix vector multiplication np.dot(W1,x) evaluates the activations of all neurons in that layer. Similarly, **W2** would be a **[4x4]** matrix that stores the connections of the second hidden layer, and **W3** a **[1x4]** matrix for the last (output) layer. The full forward pass of this 3-layer neural network is then simply three matrix multiplications, interwoven with the application of the activation function:

Representational power

- Neural Networks with fully-connected layers define a family of functions that are parameterized by the weights of the network.
- Neural Networks with at least one hidden layer are universal approximators.
- It can be shown (e.g. see Approximation by Superpositions of Sigmoidal Function from 1989), that given any continuous function f(x) and some ϵ>0, there exists a Neural Network g(x) with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that **$\forall x, |f(x)-g(x)|<\epsilon$**. In other words, the neural network can approximate any continuous function.

Setting number of layers and their sizes

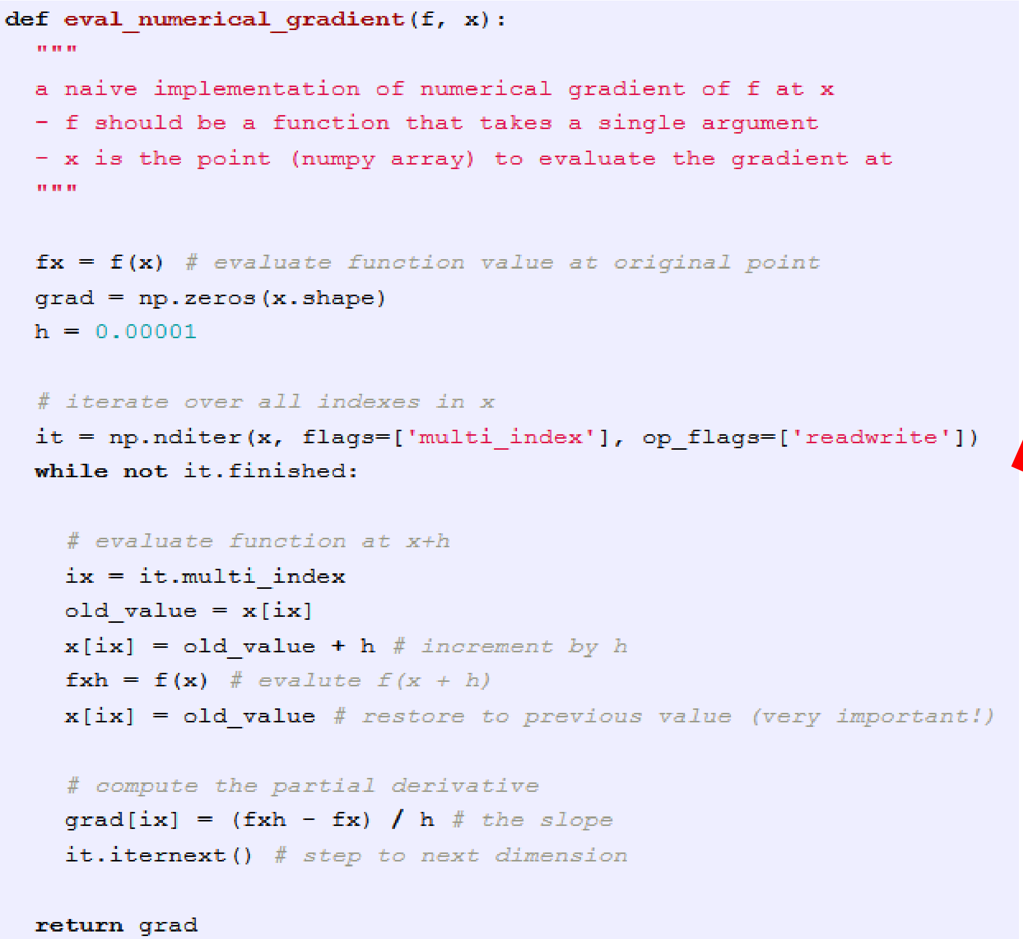
- How do we decide on what architecture to use when faced with a practical problem? Should we use no hidden layers? One hidden layer? Two hidden layers? How large should each layer be?
- First, note that as we increase the size and number of layers in a Neural Network, the **capacity** of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions.
- For example, suppose we had a binary classification problem in two dimensions. We could train three separate neural networks, each with one hidden layer of some size and obtain the following classifiers:



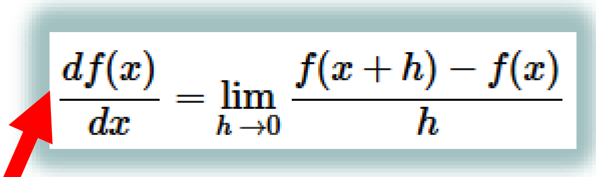
Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath.

Gradient Descent

- Computing the gradient numerically with finite differences



```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```



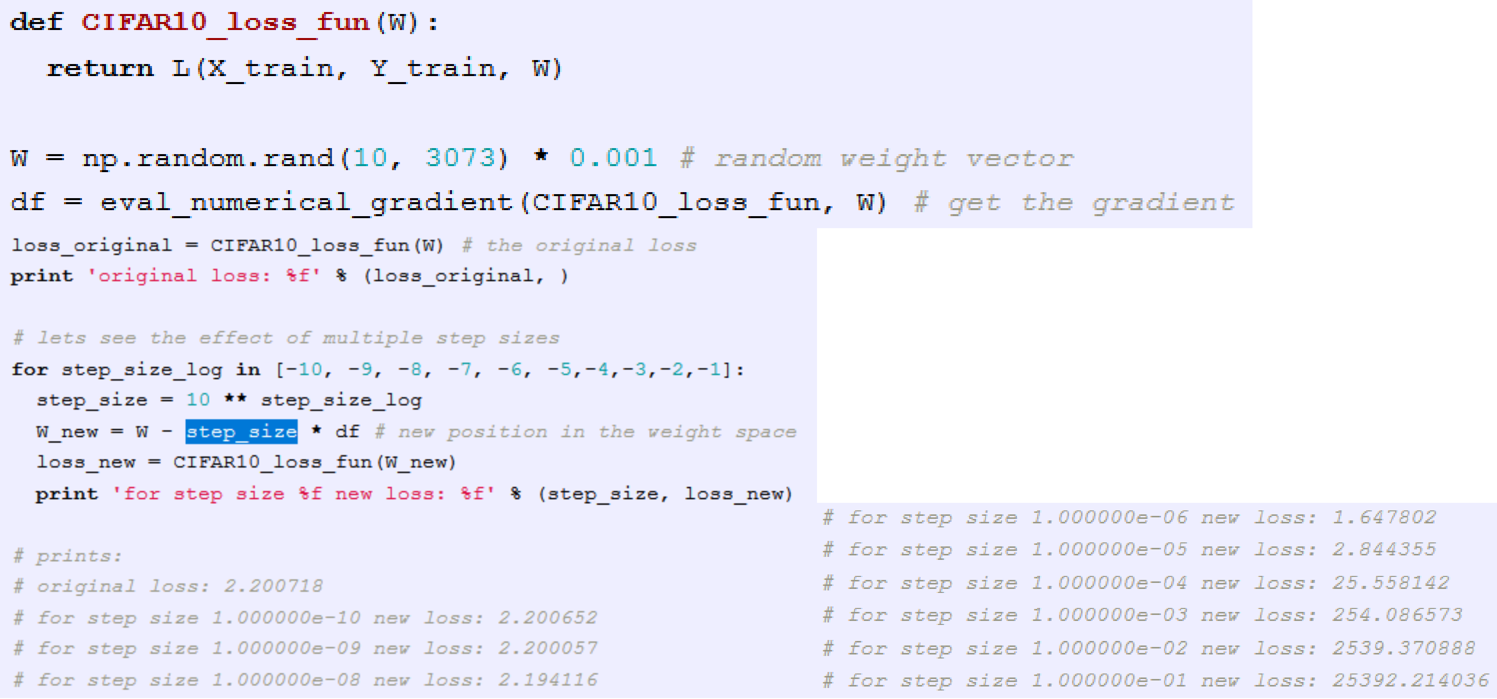
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Gradient Descent

- Following the gradient formula, the code above iterates over all dimensions one by one, makes a small change h along that dimension and calculates the partial derivative of the loss function along that dimension by seeing how much the function changed. The variable grad holds the full gradient in the end.

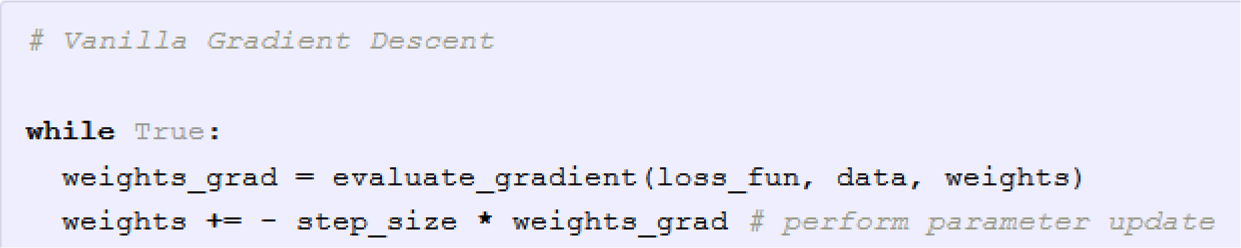
Step Size

- **Effect of step size**. The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step.
- Chosing the step size (also called the ***learning rate***) will become one of the most important (and most headache-inducing) hyperparameter settings in training a neural network.
- **Very small progress** (this corresponds to having a small step size) VS **descend faster**, but this may not pay off.



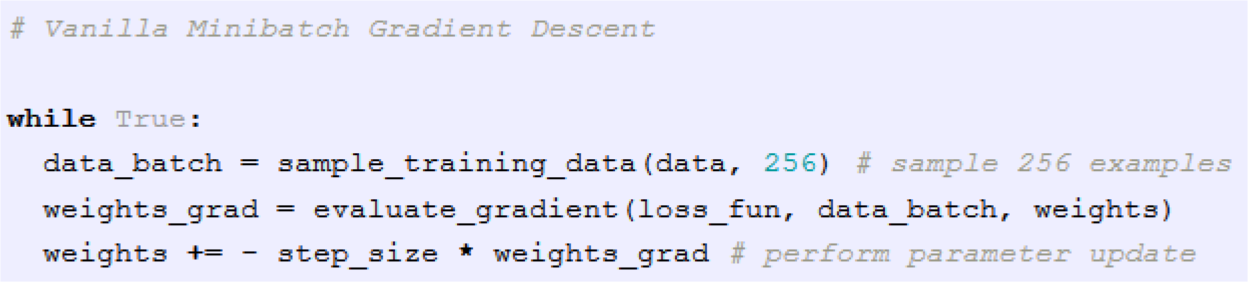
```
def CIFAR10_loss_fun(W):
  return L(X_train, Y_train, W)

W = np.random.rand(10, 3073) * 0.001 # random weight vector
df = eval_numerical_gradient(CIFAR10_loss_fun, W) # get the gradient
loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

# lets see the effect of multiple step sizes
for step_size_log in [-10, -9, -8, -7, -6, -5,-4,-3,-2,-1]:
  step_size = 10 ** step_size_log
  W_new = W - step_size * df # new position in the weight space
  loss_new = CIFAR10_loss_fun(W_new)
  print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
# for step size 1.000000e-10 new loss: 2.200652
# for step size 1.000000e-09 new loss: 2.200057
# for step size 1.000000e-08 new loss: 2.194116
# for step size 1.000000e-06 new loss: 1.647802
# for step size 1.000000e-05 new loss: 2.844355
# for step size 1.000000e-04 new loss: 25.558142
# for step size 1.000000e-03 new loss: 254.086573
# for step size 1.000000e-02 new loss: 2539.370888
# for step size 1.000000e-01 new loss: 25392.214036
```

Vanilla GD



```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- **Mini-batch gradient descent.** In large-scale applications, the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over **batches** of the training data. For example, in current state of the art ConvNets, a typical batch contains 256 examples from the entire training set of 1.2 million. This batch is then used to perform a parameter update:



```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic GD

- The extreme case of this is a setting where the mini-batch contains only a single example.
- This process is called **Stochastic Gradient Descent (SGD)** (or also sometimes **on-line** gradient descent).
- This is relatively less common to see because in practice due to **vectorized code** optimizations it can be computationally much more efficient to evaluate the gradient for 100 examples, than the gradient for one example 100 times.
- Even though SGD technically refers to using a single example at a time to evaluate the gradient, the term SGD is used when referring to **mini-batch gradient descent** (i.e. mentions of MGD for “Minibatch Gradient Descent”, or BGD for “Batch gradient descent” are rare to see), where it is usually assumed that mini-batches are used.
- The size of the mini-batch is a **hyperparameter** but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or set to some value, e.g. 32, 64 or 128. We use powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2.

Convolutional Neural Networks (CNNs / ConvNets)

- Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.
- The whole network still expresses a **single differentiable score function**: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.
- ConvNet architectures make the explicit assumption that **the inputs are images**, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

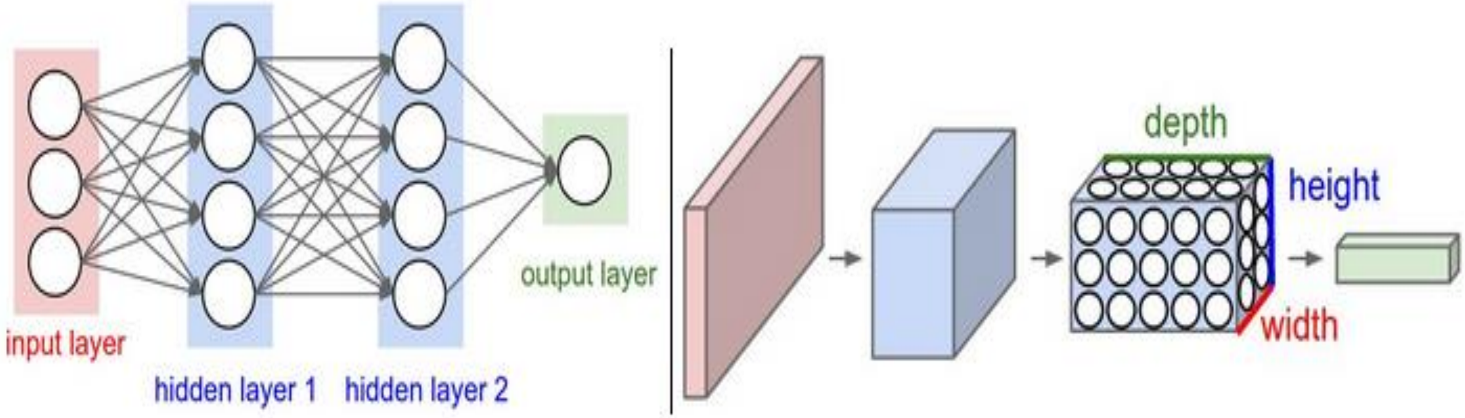
Architecture Overview

- *3D volumes of neurons*. Convolutional Neural Networks take advantage of the fact that the input consists of images

- The layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**.

- For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it.
- Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension.

Architecture Overview



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Layers used to build ConvNets

- A simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function.
- Types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

  - INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
  - CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
  - RELU layer will apply an elementwise activation function, such as the max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
  - POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
  - FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10.

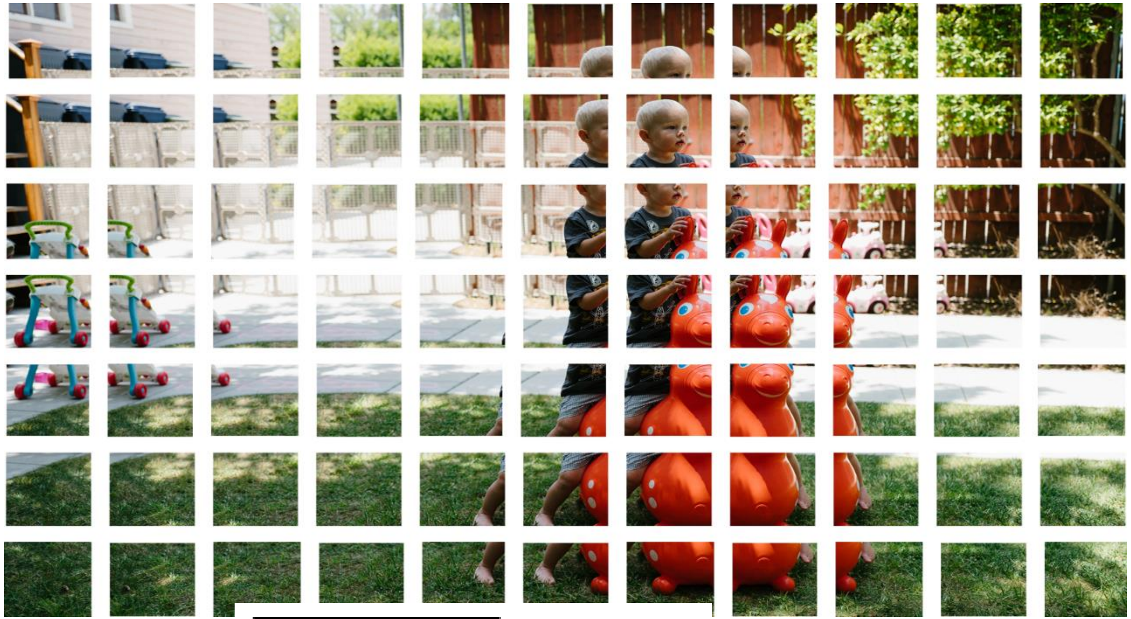
ConvNets

- Instead of feeding entire images into our neural network as one grid of numbers, we’re going to do something a lot smarter that takes advantage of the idea that an object is the same no matter where it appears in a picture.

- **Step 1: Break the image into overlapping image tiles**

ConvNets

By doing this, we turned our original image into 77 equally-sized tiny image tiles.



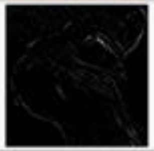
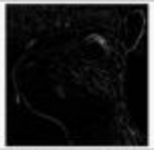
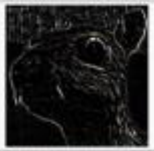
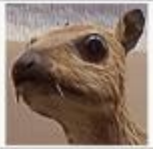
1	0	1
0	1	0
1	0	1

1	1	1	0	0
0	1	1	1	0
0	0	$1_{\times1}$	$1_{\times0}$	$1_{\times1}$
0	0	$1_{\times0}$	$1_{\times1}$	$0_{\times0}$
0	1	$1_{\times1}$	$0_{\times0}$	$0_{\times1}$

Image

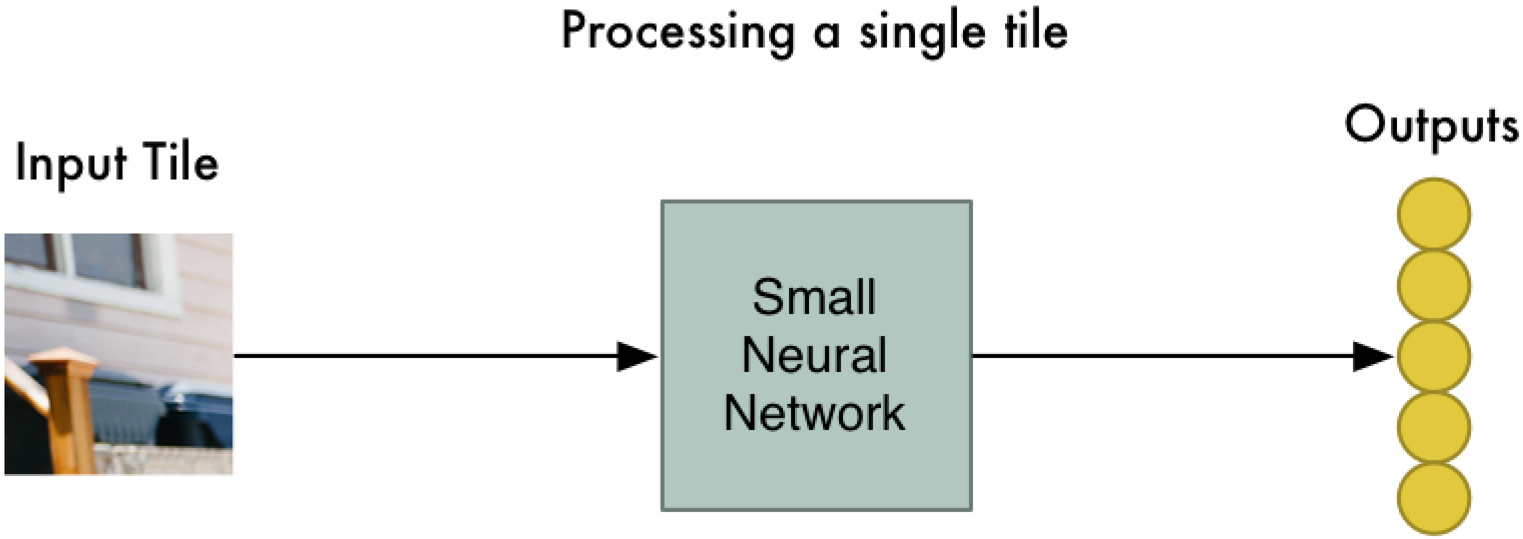
4	3	4
2	4	3
2	3	4

Convolved Feature

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$	
Edge detection	$\begin{bmatrix}1&0&-1\\0&0&0\\-1&0&1\end{bmatrix}$	
	$\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}$	
	$\begin{bmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{bmatrix}$	
Sharpen	$\begin{bmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	

ConvNets

- **Step 2: Feed each image tile into a small neural network**

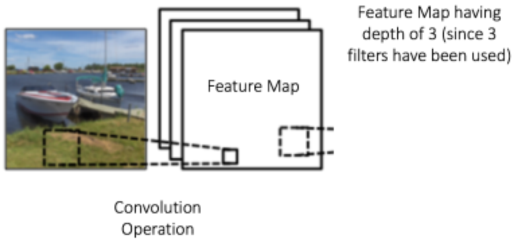


However, **there's one big twist**: We'll keep the **same neural network weights** for every single tile in the same original image. In other words, we are treating every image tile equally. If something interesting appears in any given tile, we'll mark that tile as interesting.

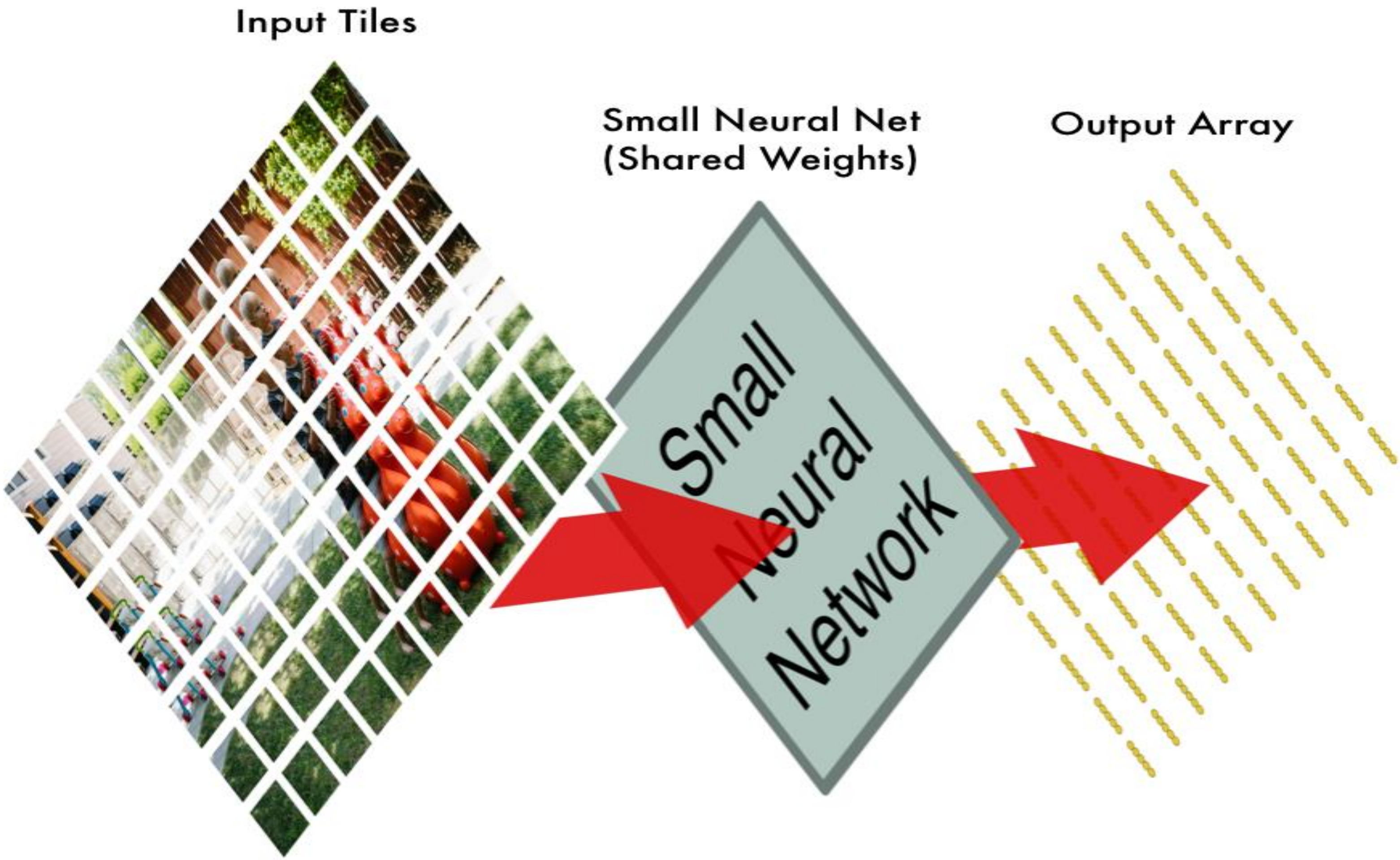
ConvNets

- **Step 3: Save the results from each tile into a new array**

- We don’t want to lose track of the arrangement of the original tiles. So we save the result from processing each tile into a grid in the same arrangement as the original image.
- In other words, we’ve started with a large image and we ended with a slightly smaller array that records which sections of our original image were the most interesting.


Feature Map having depth of 3 (since 3 filters have been used)
Feature Map
Convolution Operation

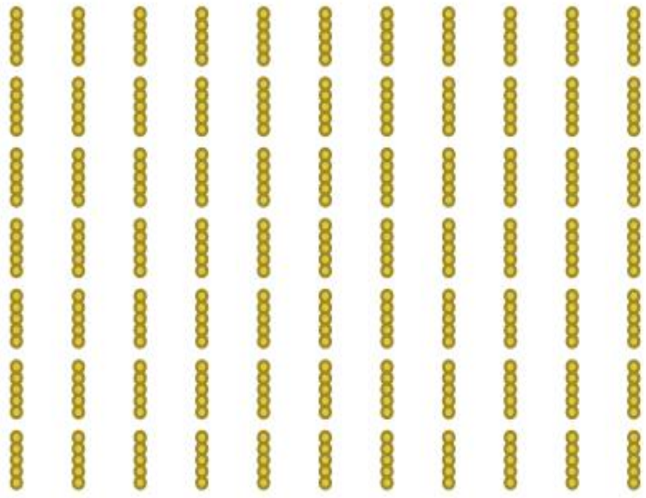
ConvNets


Input Tiles
Small Neural Net
(Shared Weights)
Output Array
Small
Neural
Network

ConvNets

Step 4: Downsampling

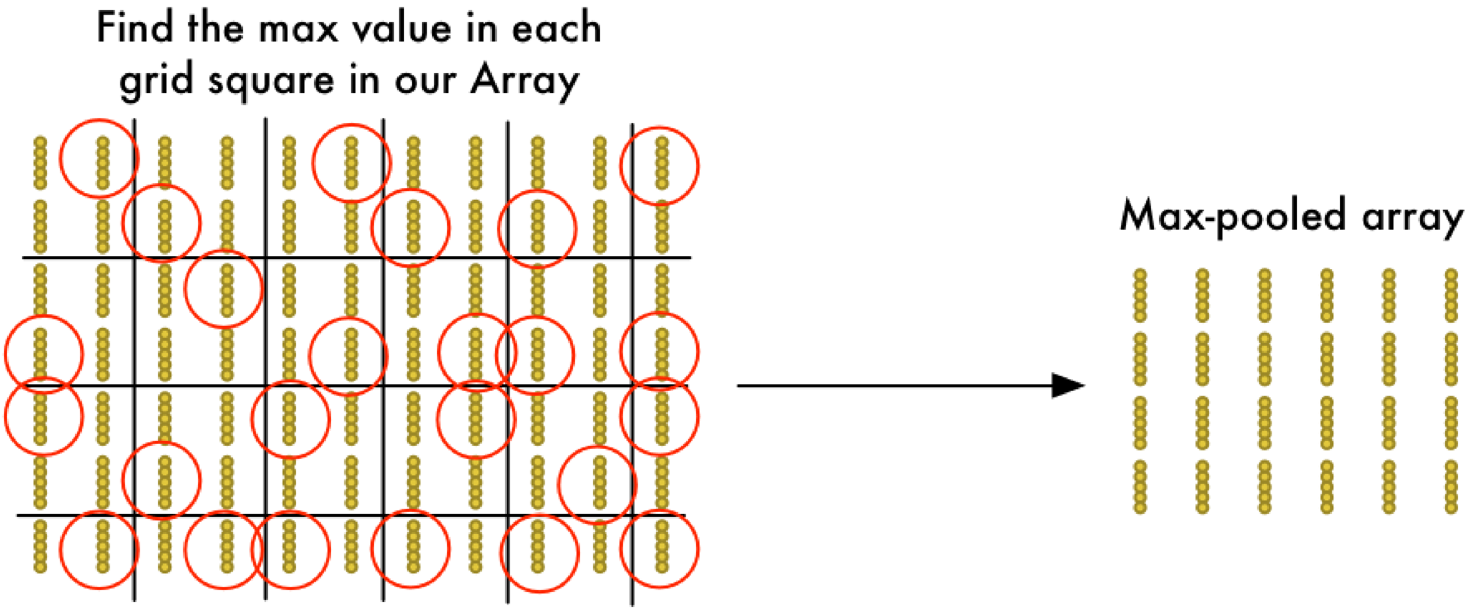
Original Input Image



Array resulting from convolution in Step 3

To reduce the size of the array, we *downsample* it using an algorithm called max pooling.

ConvNets

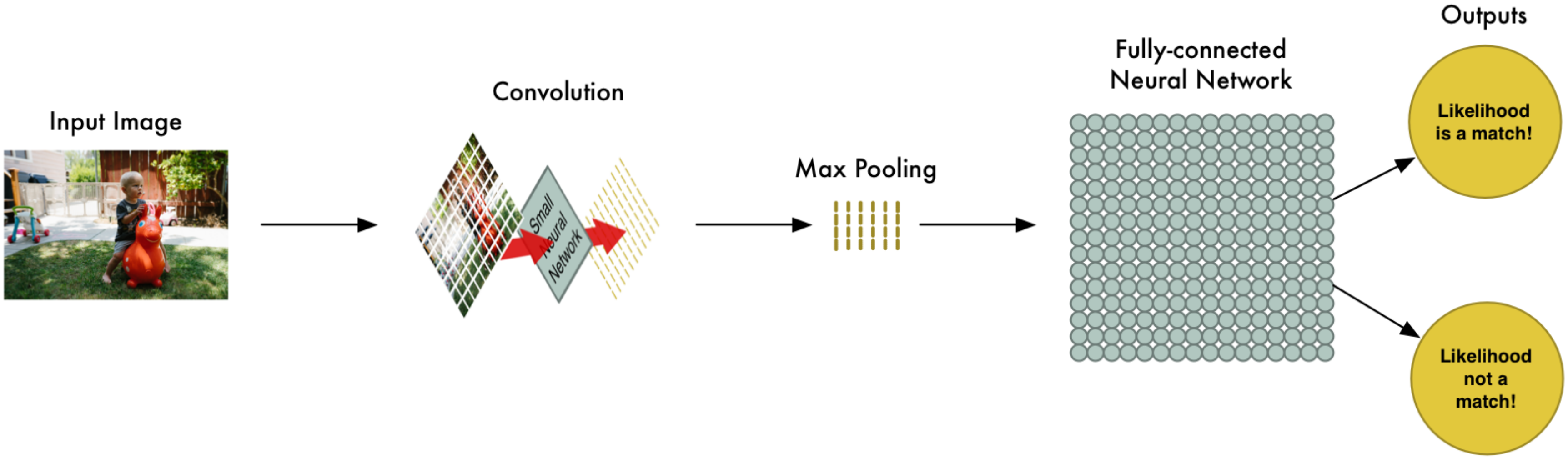


The idea here is that if we found something interesting in any of the four input tiles that makes up each 2x2 grid square, we'll just keep the most interesting bit. This reduces the size of our array while keeping the most important bits.

ConvNets

Final step: Make a prediction

So far, we've reduced a giant image down into a fairly small array. That array is just a bunch of numbers, so we can use that small array as input into *another neural network*. This final neural network will decide if the image is or isn't a match. To differentiate it from the convolution step, we call it a "fully connected" network. So from start to finish, our whole five-step pipeline looks like this:


Outputs
Fully-connected
Neural Network
Convolution
Input Image
Max Pooling
Small
Neural
Network
Likelihood
is a match!
Likelihood
not a
match!