ENTER READING MODE

Convolutional Neural Networks in iOS 10 and macOS (https://www.invasivecode.com/weblog/convolutional-neural-networks-ios-10-macos-sierra/)

Search ...

Latest Posts

- Network Reachability in Swift (https://www.invasivecode.com/weblog/network-reachability-in-swift/)
- Interactive View Animations in iOS 10 (https://www.invasivecode.com/weblog/interactive-animation-view-ios-10/)
- Convolutional Neural Networks in iOS 10 and macOS (https://www.invasivecode.com/weblog/convolutional-neural-networks-ios-10-macos-sierra/)
- Machine Learning for iOS (https://www.invasivecode.com/weblog/machine-learning-swift-ios/)
- Investing in custom development (https://www.invasivecode.com/weblog/custom-development-ios)
- Apple App Distribution (https://www.invasivecode.com/weblog/app-distribution-apple)
- Replicator Layer (https://www.invasivecode.com/weblog/replicator-layer-core-animation/)
- tvOS Focus Engine and Collection View (https://www.invasivecode.com/weblog/tvOS-apple-watch-tv/)
- Capture Video with AVFoundation and Swift (https://www.invasivecode.com/weblog/AVFoundation-Swift-capture-video/)
- Metal: Blazing Fast Image Processing (https://www.invasivecode.com/weblog/metal-image-processing)

Categories

- App Distribution (https://www.invasivecode.com/weblog/development-tools/app-

Posted on Jul 12th, 2016

by **Geppy Parziale (https://www.invasivecode.com/weblog/author/geppy/)**

- cnn (https://www.invasivecode.com/weblog/tag/cnn/)
- computer vision (https://www.invasivecode.com/weblog/tag/computer-vision/)
- convolution (https://www.invasivecode.com/weblog/tag/convolution/)
- convolutional neural network (https://www.invasivecode.com/weblog/tag/convolutional-neural-network/)
- deep learning (https://www.invasivecode.com/weblog/tag/deep-learning/)
- iOS 10 (https://www.invasivecode.com/weblog/tag/ios-10/)
- machine learning (https://www.invasivecode.com/weblog/tag/machine-learning/)
- macOS (https://www.invasivecode.com/weblog/tag/macos/)
- neural network (https://www.invasivecode.com/weblog/tag/neural-network/)
- pooling (https://www.invasivecode.com/weblog/tag/pooling/)
- Sierra (https://www.invasivecode.com/weblog/tag/sierra/)
- Machine Learning (https://www.invasivecode.com/weblog/machine-vision/machine-learning/)
- Machine Vision (https://www.invasivecode.com/weblog/machine-vision/)
- Neural Network (https://www.invasivecode.com/weblog/machine-vision/neural-network/)

(https://www.invasivecode.com/weblog/wp-content/uploads/2016/07/neural-network.png)

Convolutional Neural Networks in iOS 10 and macOS

In iOS 10 and macOS 10.12, Apple introduces new Convolutional Neural Network APIs in the Metal Performance Shaders Framework and the Accelerate Framework.

In a previous post, I already provided you with

distribution/)

- Apple Watch (https://www.invasivecode.com/weblog/apple-watch/)
- Augmented Reality (https://www.invasivecode.com/weblog/augmented-reality/)
- Auto Layout (https://www.invasivecode.com/weblog/development-tools/auto-layout/)
- AVFoundation & Accelerate (https://www.invasivecode.com/weblog/image-video-processing/)
- Biometrics (https://www.invasivecode.com/weblog/biometrics/)
- CloudKit (https://www.invasivecode.com/weblog/cloud-kit/)
- Collection View (https://www.invasivecode.com/weblog/uikit/ui-collection-view/)
- Core Animation (https://www.invasivecode.com/weblog/core-animation-tutorial/)
- Core Bluetooth (https://www.invasivecode.com/weblog/core-bluetooth-tutorial/)
- Core Data (https://www.invasivecode.com/weblog/core-data/)
- Core Graphics (https://www.invasivecode.com/weblog/core-graphics-tutorial/)
- Core Motion (https://www.invasivecode.com/weblog/core-motion/)
- Core Services (https://www.invasivecode.com/weblog/core-services/)
- Core Text (https://www.invasivecode.com/weblog/core-text-tutorial/)
- Development Tools (https://www.invasivecode.com/weblog/development-tools/)
- Foundation (https://www.invasivecode.com/weblog/foundation/)
- Game Kit (https://www.invasivecode.com/weblog/games-game-kit/)
- Instruments (https://www.invasivecode.com/weblog/development-tools/instruments/)

an introduction on Machine Learning (ML) and Artificial Neural Networks (ANN) for iOS (https://www.invasivecode.com/weblog/machine-learning-swift-ios/). If you are not familiar with this topics, I suggest to read that post, first.

I recently attended CVPR 2016 (http://cvpr2016.thecvf.com), a scientific conference on Computer Vision and Pattern Recognition. There, I learnt that Convolutional Neural Networks are used in almost every research work recently done by universities and companies around the world. The popularity of the Convolutional Neural Networks in different fields of computer vision and the availability of fast and powerful GPUs on smartphones made these neural networks a very attractive tool also for mobile development. Convolutional Neural Networks and Deep Learning open a wide range of innovative mobile applications.

I started to work on Convolutional Neural Networks (CNNs) five years ago while I was at Apple. At that time, the available literature and tools were extremely limited compared with what you can find around today. I used CNNs to build an Optical Character Recognition (OCR) running on iOS and OS X. I am talking of an implementation done in iOS 5. The accuracy of the OCR was amazing, even if its implementation on the device was done at that time using the CPU.

After that work, I continued to use CNNs for other kind of applications. Recently, I am using CNNs for face recognition and facial expression. The results we are getting are amazing.

Convolution

A CNN makes a large use of a very common signal processing operation called **convolution**. The convolution performs a weighted sum of a collection of neighboring elements of an array

- Interface Builder (https://www.invasivecode.com/weblog/development-tools/interface-builder/)
- iOS Development (https://www.invasivecode.com/weblog/ios-development/)
- iOS Training (https://www.invasivecode.com/weblog/ios-training/)
- LLDB (https://www.invasivecode.com/weblog/development-tools/lldb/)
- LLVM (https://www.invasivecode.com/weblog/development-tools/llvm/)
- Machine Learning (https://www.invasivecode.com/weblog/machine-vision/machine-learning/)
- Machine Vision (https://www.invasivecode.com/weblog/machine-vision/)
- Map Kit & Core Location (https://www.invasivecode.com/weblog/geo-location-and-maps/)
- Metal (https://www.invasivecode.com/weblog/metal/)
- Networking (https://www.invasivecode.com/weblog/foundation/networking/)
- Neural Network (https://www.invasivecode.com/weblog/machine-vision/neural-network/)
- News (https://www.invasivecode.com/weblog/news/)
- NSAttributedString (https://www.invasivecode.com/weblog/foundation/attributed-string/)
- Objective-C (https://www.invasivecode.com/weblog/objective-c/)
- Research Kit (https://www.invasivecode.com/weblog/research-kit/)
- Swift (https://www.invasivecode.com/weblog/swift/)
- Tech Talks (https://www.invasivecode.com/weblog/tech-talks/)
- tvOS (https://www.invasivecode.com/weblog/tvos/)

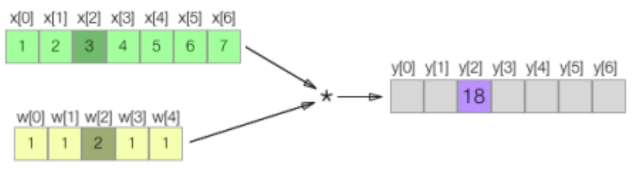
(or matrix). The weights used in the weighted sum calculation are defined by an input array, commonly referred to as the *kernel*, *filter* or *mask* of the convolution.

Convolutions are very important in digital signal (audio, video, image) processing and Graphics Processing Units (**GPUs**) are optimized to executes convolutions in a very efficient way. GPUs become the most important implementation tool when you want to work with CNNs.

As humans, we also use convolutions in our daily activities, specially in those activities where our 5 senses are involved. For example, while listening to music or looking at something, our brain performs millions of convolutions per second on the sound and the light we receive from the external world.

1D convolution example

Let's build an example to understand better how convolution works. The following figure shows a convolution between an input array or 1D signal x[n] and a 1D kernel array w[n] .


x[0] x[1] x[2] x[3] x[4] x[5] x[6]
1 2 3 4 5 6 7
w[0] w[1] w[2] w[3] w[4]
1 1 2 1 1
*
y[0] y[1] y[2] y[3] y[4] y[5] y[6]
18

$$y[2] = x[0]w[4] + x[1]w[3] + x[2]w[2] + x[3]w[1] + x[4]w[0] =$$

$$1*1 + 2*1 + 3*2 + 4*1 + 5*1 = 18$$

In this example, I am arbitrarily assuming that the values of the input array are 1, 2,...,7 and the values of the kernel array are 1, 1, 2, 1, 1. The previous figure shows how the element (or sample) y[2] of the output sequence y[n] is computed.

- UI Animations (https://www.invasivecode.com/weblog/ui-animations/)
- UI Dynamics (https://www.invasivecode.com/weblog/uikit/uikit-dynamics/)
- UIKit (https://www.invasivecode.com/weblog/uikit/)
- View Controller (https://www.invasivecode.com/weblog/uikit/ui-view-controller/)
- WebKit (https://www.invasivecode.com/weblog/webkit/)
- WWDC (https://www.invasivecode.com/weblog/wwdc/)
- Xcode (https://www.invasivecode.com/weblog/development-tools/xcode/)

Tags

- animation (https://www.invasivecode.com/weblog/tag/animation/)
- apple (https://www.invasivecode.com/weblog/tag/apple/)
- apple watch (https://www.invasivecode.com/weblog/tag/apple-watch/)
- attributed string (https://www.invasivecode.com/weblog/tag/attributed-string/)
- AVFoundation (https://www.invasivecode.com/weblog/tag/avfoundation/)
- barcelona (https://www.invasivecode.com/weblog/tag/barcelona/)
- biometrics (https://www.invasivecode.com/weblog/tag/biometrics/)
- CAReplicatorLayer (https://www.invasivecode.com/weblog/tag/careplicatorlayer/)
- class (https://www.invasivecode.com/weblog/tag/class/)
- classes (https://www.invasivecode.com/weblog/tag/classes/)
- Cocoa (https://www.invasivecode.com/weblog/tag/cocoa/)
- cocoa touch (https://www.invasivecode.com/weblog/tag/cocoa-touch/)
- Core Animation (https://www.invasivecode.com/weblog/tag/core-animation/)

In general, the size of the kernel tends to be an odd number, which makes the weighted sum calculation symmetric around the element being processed. It is also very common to use kernels much smaller than the input sequence x[n] . The central element of the kernel is used as the weight of the element in the input signal we want to process. Then, the remaing elements of the kernel are used as weights of the elements on the right and left handside of the computed element.

To generalize the previous example, if x[n] is an input sequence and w[m] is a kernel sequence, the result y[n] of the convolution operation can be expressed by the following mathematical expression:

$$y[n] = \sum_{m=0}^{m=M} x[m] \cdot w[n-m] (Eq.1)$$

Notice that the sequence w[m] is first reversed and then, translated during the operation.

If we use the previous mathematical formula to compute each element of y[n] of the previous example, then we obtain the following results:

$$y[n] = x[0]w[n-0] + x[1]w[n-1] + \cdots + x[4]w[n-4]$$

Since a convolution is defined in terms of neighboring elements, boundary conditions naturally exist for output elements that are close to the ends of an array. To avoid this issue, it is very common to add enough elements (called *ghost* elements) to the ends of the input sequence x[n] . If you add zeros, the operation is called *zero-padding*. Other approaches are also available. When implementing a convolution, you need to handle the padding.

Convolution in Swift

Let's see how to implement a convolution in Swift. Suppose we have the following input array

core data (https://www.invasivecode.com/weblog/tag/core-data/)
Core Image (https://www.invasivecode.com/weblog/tag/core-image/)
fingerprint (https://www.invasivecode.com/weblog/tag/fingerprint/)
Frameworks (https://www.invasivecode.com/weblog/tag/frameworks/)
image processing (https://www.invasivecode.com/weblog/tag/image-processing/)
Instruments (https://www.invasivecode.com/weblog/tag/instruments/)
invasivecode (https://www.invasivecode.com/weblog/tag/invasivecode/)
ios (https://www.invasivecode.com/weblog/tag/ios/)
iOS 6 (https://www.invasivecode.com/weblog/tag/ios-6/)
iOS 7 (https://www.invasivecode.com/weblog/tag/ios-7/)
ios 8 (https://www.invasivecode.com/weblog/tag/ios-8/)
iOS 9 (https://www.invasivecode.com/weblog/tag/ios-9/)
iOS consulting (https://www.invasivecode.com/weblog/tag/ios-consulting/)
iOS training (https://www.invasivecode.com/weblog/tag/ios-training/)
ipad (https://www.invasivecode.com/weblog/tag/ipad/)
iphone (https://www.invasivecode.com/weblog/tag/iphone/)
mapkit (https://www.invasivecode.com/weblog/tag/mapkit/)
maps (https://www.invasivecode.com/weblog/tag/maps/)
nsattributedstring (https://www.invasivecode.com/weblog/tag/nsattributedstring/)
objective-c (https://www.invasivecode.com/weblog/tag/objective-c/)
os x (https://www.invasivecode.com/weblog/tag/os-x/)
performance (https://www.invasivecode.com/weblog/tag/performance/)
sdk (https://www.invasivecode.com/weblog/tag/sdk/)
Swift (https://www.invasivecode.com/weblog/tag/swift/)
swiftlang (https://www.invasivecode.com/weblog/tag/swiftlang/)

x and kernel array w :

```
let x: [Float] = [1, 2, 3, 4, 5], M = x.count
let w: [Float] = [1, 2, 3], N = kernel.count

let T = N+M-1 // we need this later

```

Before we start, let's add N-1 zeros to the end of the sequence x , and M-1 zeros to the end of the kernel to accomodate the calculations as explained previously. You can use the following function:

```
func pad(sequence x: [Float], other sequence: [Float]) -> [F
   return x + [Float](repeating: 0, count: sequence.count-1
}

```

So, the new padded sequences are:

```
let paddedX: [Float] = pad(sequence: x, other: kernel)
let paddedK: [Float] = pad(sequence: kernel, other:  x)

```

Now, we can build a convolution between paddedX and paddedK :

```
var y = [Float](repeating: 0, count: T)

for i in 0..<T {
   y[i] = 0
   for j in 0..<M {
      if i-j+1 > 0 {
         y[i] += paddedX[j] * paddedK[i-j]
      }
   }
}


```

Finally, the result of the convolution is:

```
// y = [1, 4, 10, 16, 22]

```

Convolution using Accelerate

If you want to speedup the processing of the convolution, you can use the vDSP_conv function provided by the Accelerate framework. Also in this case, I need to handle the boundary

talk (https://www.invasivecode.com/weblog/tag/talk/)

training (https://www.invasivecode.com/weblog/tag/training/)

tvOS (https://www.invasivecode.com/weblog/tag/tvos/)

UIKit (https://www.invasivecode.com/weblog/tag/uikit/)

UIKit Dynamics (https://www.invasivecode.com/weblog/tag/uikit-dynamics/)

wwdc (https://www.invasivecode.com/weblog/tag/wwdc/)

Xcode (https://www.invasivecode.com/weblog/tag/xcode/)

Archives

Archives Select Month

conditions and the kernel reversing. This time, I zero-pad the input and the kernel in a different way. Additionally, I need to reverse the kernel (as explained in the documentation), otherwise I obtain the correlation of the two sequences.

Here the implementation using the Accelerate:

```swift
import Accelerate

let x: [Float] = [1, 2, 3, 4, 5], M = x.count
let kernel: [Float] = [1, 2, 3], N = kernel.count
let T = N+M-1

var res = [Float](repeating: 0, count: T)

let zeros = [Float](repeating: 0, count: N-1)

let newXin = zeros + x + zeros

vDSP_conv(newXin, 1, kernel.reversed(), 1, &res, 1, vDSP_


```

For this very short input sequence, you will not appreciate the speedup provided by the Accelerate framework. Instead, I created an input array with 100,000 elements and convolved it with the same `w` kernel of the previous example. My Swift implementation required 318 ms, while the Accelerate `vDSP_conv` method required just 159 ns on my MacBook Pro.

Convolution using Metal

Let's see how to implement the same example in Metal. Check this post (https://www.invasivecode.com/weblog/metal-image-processing) to learn how to setup a Metal project for computations on the GPU.

In this particular example, we need to create 3 Metal textures (objects conforming to the `MTLTexture` protocol): the first texture will hold the input sequence, the second texture will hold the kernel and the third texture will hold the final result.

Here the source code to create these texture:

```
import Metal

let paddedX: [Float] = input + [Float](repeating: 0, count:
let paddedK: [Float] = kernel + [Float](repeating: 0, count

let inputTextureDescriptor = MTLTextureDescriptor.textu
inputTextureDescriptor.usage = .shaderRead
inTexture = metalContext.device.newTexture(with: input
let region = MTLRegionMake2D(0, 0, paddedX.count, 1)
inTexture?.replace(region, mipmapLevel: 0, withBytes: pa

let kernelTextureDescriptor = MTLTextureDescriptor.text
kernelTexture = metalContext.device.newTexture(with: k
let kernelRegion = MTLRegionMake2D(0, 0, paddedK.cou
kernelTexture?.replace(kernelRegion, mipmapLevel: 0, w

let outputTextureDescriptor = MTLTextureDescriptor.text
outputTextureDescriptor.usage = .shaderWrite
outTexture = metalContext.device.newTexture(with: outp

executeConvolution()

```

In the previous source code, the metalContext is an instance of the following class:

```
final class MetalContext: NSObject {

    let device: MTLDevice
    let commandQueue: MTLCommandQueue
    let library: MTLLibrary

    override init() {

        // Get the device
        self.device = MTLCreateSystemDefaultDevice()!

        // Create a command queue
        self.commandQueue = device.makeCommandQueu

        // Get the default library
        self.library = device.newDefaultLibrary()!

        super.init()
    }
}

```

This is just a helper class I usually use to setup the main objects of a Metal stack.

Last method executeConvolution() is instead used to encode the commands for the GPU:

Finally, we need to the Metal kernel. This is where we perform the convolution. A very simple implementation can be the following:

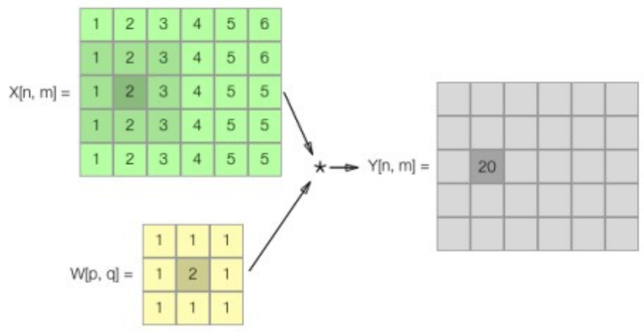
2D Convolution

```
func executeConvolution() {

    guard let outTexture = self.outTexture else { return }

    let commandBuffer = metalContext.commandQueue.c

    let computeCommandEncoder = commandBuffer.com

   computeCommandEncoder.setComputePipelineState(c

    computeCommandEncoder.setTexture(inTexture, at: 0
    computeCommandEncoder.setTexture(kernelTexture,
    computeCommandEncoder.setTexture(outTexture, at:

    computeCommandEncoder.dispatchThreadgroups(MT

    computeCommandEncoder.endEncoding()

    commandBuffer.commit()

    let region = MTLRegionMake1D(0, T)
    var buffer = [Float32](repeating: 0, count: T)
    outTexture.getBytes(&buffer, bytesPerRow: T*sizeof(F
}

```

```
#include <metal_stdlib>
using namespace metal;


constexpr sampler sam(address::clamp_to_zero, filter::n

kernel void convolution( texture2d<float, access::sample
                         texture2d<float, access::read> weights
                         texture2d<float, access::write> outTexture
                         uint2 gid [[thread_position_in_grid]] )
{
    if ( gid.x >= outTexture.get_width() || gid.y >= outTextu

    int i = gid.x;

    float acc = 0;
    for (int j = 0; j < (i+1); ++j) {
        float color = inTexture.sample(sam, static_cast<float
        float weight = weights.read(uint2(i-j, 0)).x;
        acc += (color * weight);
    }

    outTexture.write(acc, gid);
}

```

When processing images, convolutions are performed on to 2D data. In this case, an image is represented by a matrix X[n,m] instead of a single dimensional array.

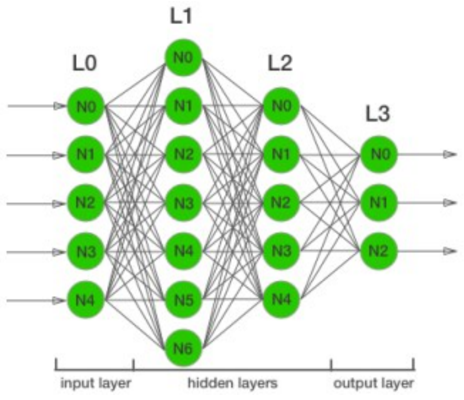
The following figure shows how to compute the result of the convolution for the element Y[1, 2] of the output matrix Y. The highlighted elements are the center of the convolution operation.



As for the 1D convolution case, I can build here the example for the 2D case in Swift, using the Accelerate framework and also Metal. I leave this to you as exercise. Remember to flip the W along rows and along columns. Additionally, rember to pad X with P-1 and Q-1 zeros on both ends of the matrix.

Convolutional Neural Network

The following figure highlightds a fully connected neural network with 2 hidden layers (L1 and L2).



As discussed in a previous post (https://www.invasivecode.com/weblog/machine-learning-swift-ios/), this network is composed by layers and each layer is composed by neurons. Let's consider the neuron N0 in the hidden layer L1 . Its input is the weighted sum of the outputs of each neuron of the previous layer L0 :

$y_0^1 = \sum_{i=0}^4 x_i^0 w_{i0}^1 = x_0^0 w_{00}^1 + x_1^0 w_{10}^1 + x_2^0 w_{20}^1 + x_3^0 w_{30}^1 + x_4^0 w_{40}^1 (Eq.2)$

In this expression, y_0^1 is the input of neuron N0 in layer L1 , x_i^0 is the output of neuron Ni in layer L0 , and w_{i0}^1 is the weight between neuron Ni in layer L0 and neuron N0 in layer L1 .

In the same way, the following equation represents the input of the neuron N1 in layer L1 :

$y_1^1 = \sum_{i=0}^4 x_i^0 w_{i1}^1 = x_0^0 w_{01}^1 + x_1^0 w_{11}^1 + x_2^0 w_{21}^1 + x_3^0 w_{31}^1 + x_4^0 w_{41}^1 (Eq.3)$

Similar equations apply to the remaining the neurons of layer L1 and the neurons in layer L2 and L3 .

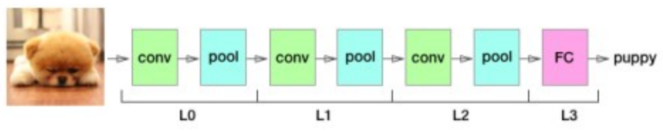
If we organize the inputs of layer L1 in a column array, the weights between layer L0 and L1 in a matrix, and the output of layer L0 in a column array, we obtain the following equation:

$y = W \cdot x (Eq.4)$

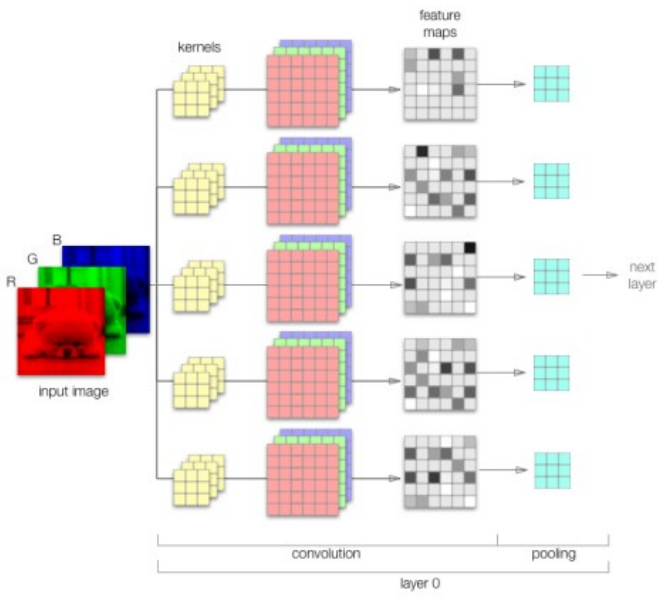
Now, if I want to process an image with this fully connected neural network, the input layer must have a number of neurons equal to the number of pixels in the input image. So, we would need an input layer with 10000 neurons to process an image with only 100x100 pixels. This means the number of columns of the matrix W in the previous equation (4) would be 10000. This is really computational expensive. Additionally, each pixel is processed indipendetly of neighbor pixels.

So, we need to optimize the processing. If we look carefully at the previous equations (2) and (3), we will notice that they look very similar to the convolution equation (1). So, instead of computing different matrix multiplications (one for each layer), we can use fast convolution algorithms. This allows us to replace the fully connected neural network implementation with a CNN implementation.

Similar to the fully connected neural network, a CNN is a sequence of layers. The following figure highlights a typical CNN structure (other configurations have been proposed in the literature):



Each CNN layer is composed by two operations: a convolution followed by a pooling. The following figure highlights the first convolution layer of a CNN:



In iOS 10 and macOS 10.12, the Metal Performance Shaders Framework (and the Accelerate Framework) provides a new class to setup the particular convolution operation for a CNN.

As you can see in the previous figure, the input image is decomposed in its 3 channels (Red, Green and Blue). Each channel is then convolved with different kernels obtained by the training step. The three results are then combined together to obtain **feature maps**. In the previous figure, I am representing just 4 feature maps. In a real CNN, you usually use 16, 32 or more feature maps. So, you need 16x3 or 32x3 kernels.

The following snippet of code shows how to create the convolution operation for a CNN layer.

```
let convDesc = MPSCNNConvolutionDescriptor(kernelWid

var conv0 = MPSCNNConvolution(device: device, convolut

```

Similar to other Metal objects, we need to use a descriptor to create the convolution. In this particular case, we use the `MPSCNNConvolutionDescriptor` class. After that, we can create an instance of the `MPSCNNConvolution` class.

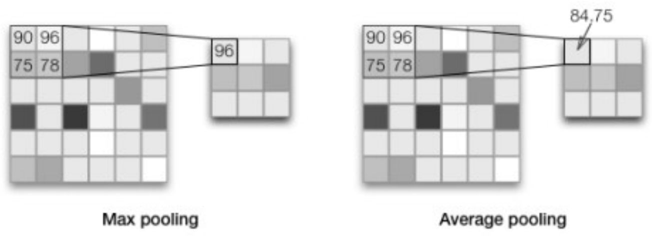
Let's give a look at the pooling operation.

Pooling

An additional operation used in CNN layer is pooling. Basically, it is a downscale operation with the aim to reduce the image size while the image is processed from the input to the output of the CNN.

The pooling has a double functionality. First, it reduces the image resolution passing to the next CNN layer less detailed information. Second, it reduces the number of computations for next layer.

There are different techniques to perform the image size reduction. Apple provides two types of pooling: *max pooling* and *average pooling*. The following figure shows how max or average pooling works.



Max pooling is obtained using the max pixel value in a defined region of the image. Instead, average pooling is obtained computing the mean value of the defined region of the input image. For example, the average pooling for the previous image example would produce a pixel of value (90+96+75+78)/4 = 84.75.

Using the `MPSCNNPoolingMax` class, we can translate the previous figure in source code:

```
var pool = MPSCNNPoolingMax(device: device, kernelWid
```

Similarly, you can use the `MPSCNNPoolingAverage` to obtain the average pooling:

```
var pool = MPSCNNPoolingAverage(device: device, kernel
```

Fully connected layer

After chaining different convolution layers, last layers of a CNN are fully connected layers. Since this can be considered as a special case of a convolution layer, the Metal Performance Shaders Framework provides a very similar API for the fully connected layer:

```
let fcDesc = MPSCNNConvolutionDescriptor(kernelWidth:

var fc = MPSCNNFullyConnected(device: device, convoluti
```

MPSImage and MPSTemporaryImage

How do we handle the data in the Metal CNN? The Metal Performance Shaders Framework provides two new classes: MPSImage and MPSTemporaryImage .

As shown previously, the output of a convolution layer generates multiple feature maps (16 or 32). A MPSImage organizes these feature maps in channels. Since a MTLTexure may have only 4 channles (RGBA), Apple introduced these two new classes to handle more the 4 channels. So, a MPSImage is really a Metal 2D array texture containing multiple slices. The images are organized so that each pixel in the MPSImage contains 4 channels. So, a 32 feature maps is represented by an MPSImage with 8 (=32/4) slices.

This is the API you need to use an MPSImage :

```
let imgDesc = MPSImageDescriptor(channelFormat: .float

var img = MPSImage(device: device, imageDescriptor: img

```

You use the MPSImage for the input and output image of the CNN. For the intermediate results, you should instead use the MPSTemporaryImage class. The advantage of using the temporary image is that it will disappear as soon as the command buffer is submitted. This results in a reduction of the memory allocations and CPU costs.

You create a MPSTemporaryImage in a very similar way than the MPSImage :

```
let img1Desc = MPSImageDescriptor(channelFormat: float

img1 = MPSTemporaryImage(device: device, imageDescrip

```

CNN Training

To use a CNN, first you need to train it to a process known as training. The training generates a set of weight that then you use in the inference phase. The Metal Performance

Shaders APIs allow you to implement only the CNN inference. The training phase is not available through these APIs. Apple suggests to use third-party tools.

A good alternative is to contact us (https://www.invasivecode.com/mobile-computer-vision-app-development-san-francisco.html) and use our tools. We can help you with your specific classification problem. The advantage of working with us is that we take care also of any pre-processing stage that you would need for your data. We can also help you to define the neural network architecture, specifying the optimal number of layers and neurons (and other parameters).

The architecture of the neural network, the number of layers and the number of neurons for each layer requires some experience in this field. Besides knowing the math behind it, you need a lot of practical experience with CNNs. This is where we can help a lot, because of our long experience in this field.

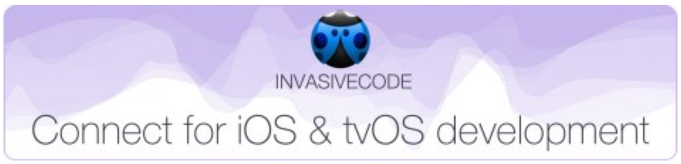
We can also provide you with our own implementation of a CNN in Metal. In this way, you do not need to use third-party APIs and upload your data to a third-party service. Additionally, since we are expert in computer vision and pattern recognition, we can preprocess your images or audio data and prepare them for the neural network. Our convolutional neural network supports iOS 8, iOS 9 and, now iOS 10. We also optimized the algorithms for tvOS and macOS.

Conclusions

In this post, I gave you an overview of the new APIs introduced in the Metal Performance Shaders Framework for iOS 10 and macOS 10.12. In a previous post (https://www.invasivecode.com/weblog/machine-learning-swift-ios/), I also provided you with an introduction on Machine Learning (ML) and Artificial Neural Networks (ANN) for iOS.

Geppy

Geppy Parziale (@geppy (http://twitter.com/geppyp)) is cofounder of InvasiveCode (http://twitter.com/invasivecode). He has developed many iOS applications and taught iOS development (http://ios-training.invasivecode.com) to many engineers around the world since 2008. He worked at Apple as iOS and OS X Engineer in the Core Recognition team. He has developed several iOS and OS X apps and frameworks for Apple, and many of his development projects are top-grossing iOS apps that are featured in the App Store. Geppy is an expert in computer vision and machine learning.



(https://www.invasivecode.com/mobile-app-development-san-francisco.html)

(https://www.invasivecode.com/ios-training-swift.html)

%2Fwww.invasivecode.com%2Fweblog%2Fconvolutional-neural-networks-
t=Convolutional%20Neural%20Networks%20in%20iOS%2010%20and%20macOS)
(http://twitter.com
[illegible]
%2Fwww.invasivecode.com%2Fweblog%2Fconvolutional-neural-networks-
t=Convolutional%20Neural%20Networks%20in%20iOS%2010%20and%20macOS)

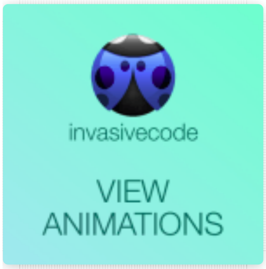
Related Posts

(https://www.invasivecode.com/weblog/machine-learning-swift-ios/)

Machine Learning for iOS (https://www.invasivecode.com/weblog/machine-learning-swift-ios/)

(https://www.invasivecode.com/weblog/network-reachability-in-swift/)

Network Reachability in Swift (https://www.invasivecode.com/weblog/network-reachability-in-swift/)



(https://www.invasivecode.com/weblog/interactive-animation-view-ios-10/)

Interactive View Animations in iOS 10 (https://www.invasivecode.com/weblog/interactive-animation-view-ios-10/)

(https://www.invasivecode.com/weblog/metal-image-processing)

Metal: Blazing Fast Image Processing (https://www.invasivecode.com/weblog/metal-image-processing)

‹
PREVIOUS POST
Machine Learning for iOS (https://www.invasivecode.com/weblog/machine-learning-swift-ios/)

›
NEXT POST
Interactive View Animations in iOS 10 (https://www.invasivecode.com/weblog/interactive-animation-view-ios-10/)