An Exploration in Convolutional Neural Networks for Image Segmentation

Nicholas Tapp-Hughes
MATH296 Directed Exploration in Mathematics
under David Adalsteinsson
at the University of North Carolina at Chapel Hill

Contents

1 Background

1.1 Introduction

Over the Spring 2019 semester, I studied Convolutional Neural Networks (CNNs) with direction from Dr. David Adalsteinsson, professor of applied mathematics at the University of North Carolina at Chapel Hill (UNC). The primary purpose of the project was to learn about CNNs and other advanced neural networks, read the current literature regarding Neural Networks and segmentation, and gain experience with the computational tools used to build, train, and test them. Secondarily, in my weekly meetings with Dr. Adalsteinsson, I could present what I had learned, and discuss current research into these topics, so that he could get a better grasp of this burgeoning field in applied mathematics.

This paper will treat the central topics that I studied over the semester. For full comprehension, the reader will need knowledge of lower-level undergraduate mathematics, and a basic familiarity with neural networks and image processing. However, this paper is written so that anyone may find it interesting.

1.2 An account of my project

I spent the first few weeks of the semester studying the online course notes for Stanford's course CS231n Convolutional Neural Networks for Visual Recognition, which can be found here. These notes give a great, comprehensive introduction to CNNs, especially geared toward computer vision tasks. Then, I familiarized myself with Keras by completing the basic TensorFlow tutorials, which can be found here. I began by using Jupyter Notebook to write my Python scripts, but I transitioned to working with .py files in Terminal using ViM (I worked on a Mac). At this point, I began investigating the current literature around segmentation models. In particular, I was interested in Pelt and Sethian's paper introducing the Mixed-Scale Dense Network (MS-D network) and Ronneberger, Fischer, and Brox's paper introducing the U-Net. I also explored other kinds of neural networks such as the Recursive Neural Network (RNN). Simultaneously, I learned about some of the traditional image processing methods such as the Canny edge detector (which I implemented in C++ as an exercise) and the Active Contour edge detector, with instruction from Dr. Adalsteinsson. I also explored TensorFlow itself, and the possibilities of low-level construction of neural network models. In the last segment of the semester, I created synthetic data for an relatively simple image segmentation problem and created and trained a Deep Convolutional Neural Network (DCNN) to segment the images, using Keras.

1.3 Neural Networks and the Image Classification Problem

I will give a brief review of the the usual example of a neural network, known as a **feedforward multilayer perceptron** (FMP). The central problem of machine learning is to develop a program which can take some input and produce some desired output. Usually, this requires **training**, which involves a dataset of possible **inputs** and corresponding desired outputs, or **labels**. During training, the program will learn to associate the inputs with their labels. Once training is completed, the program should be able to take inputs it has never seen before and produce a desirable output. The applications of such a process are immediate: face-recognition, self-driving cars, text translation, pathology diagnosis, etc.

The FMP is an example of such a scheme, where the inputs and outputs are real-valued vectors. The FMP is a mathematical function which takes the input and computes the output. It has parameters, namely the weights and biases, which are factors in the computation.

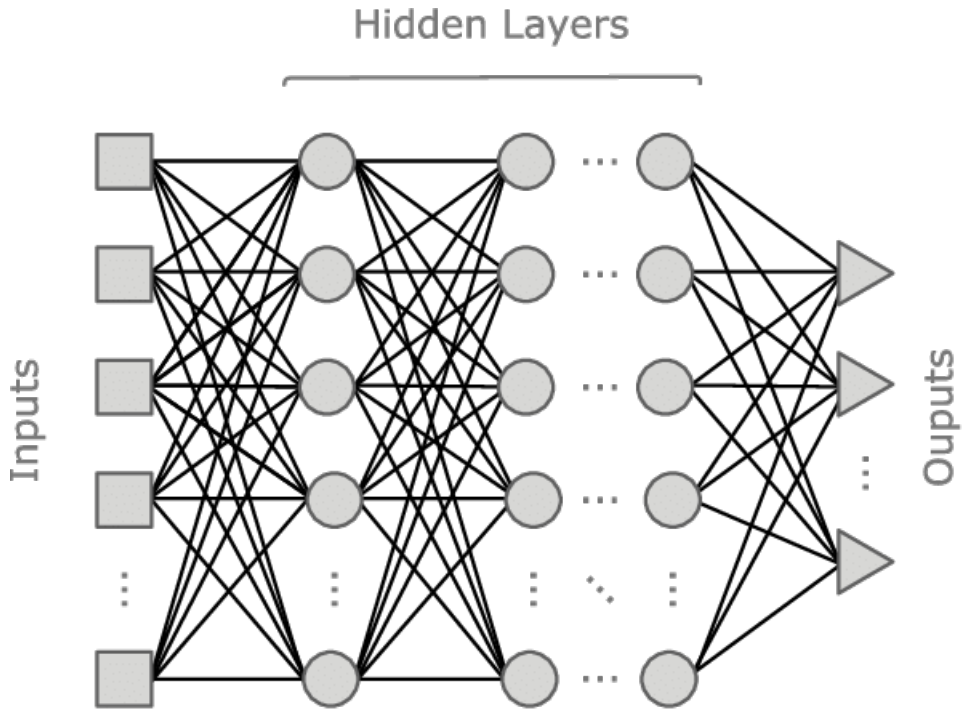

Figure 1: A Schematic Diagram of a FMP
Image Source

The FMP is composed of a sequence of layers. The first layer is precisely the input vector, while the last layer is the output vector. The layers in between are the **hidden layers**, which are composed of **neurons**. These neurons hold a number, called the **activation** of that neuron. Additionally, each neuron owns a **weight vector** and a **bias**. The weight vector has length equal to the number of nodes in the previous layer, while the bias is just a number. These weights and biases are the **parameters** of the FMP. The activation of each neuron is determined by first taking the dot product of the previous layer with the weight vector and then adding the bias. Then, an **activation function** is applied. Common

activation functions are the sigmoid function σ, tanh, softmax,or the rectified linear unit (ReLU) function. All the neurons in a layer use the same activation function. Thus, we may arrange all the weight vectors for the neurons in layer l as rows of the weight matrix w^l, and all the biases as components of the bias vector b^l. We think of the number of layers, number of neurons per layer, and other such structural properties of the FMP as **hyperparameters** of the FMP. In the Figure 1, the sequence of layers is presented left to right, as a graph, with the nodes of the graph as the neurons. The edges signify the presence of a weight. Note that each layer is densely connected to the layer preceding and following it. This model is inspired by the biological neural network with dendrites and axons.

Let a^l denote the activation vector of the lth layer. Suppose we are using the sigmoid activation function. We compute a^l_j:

$$a^l_j = \sigma(\sum_k w^l_{j,k} a^{l-1}_k + b^l_j)$$

Or more succinctly,

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

We see that each layer is a linear transformation, a translation, and a nonlinear transformation composed together. The FMP as a whole is a composition of these layers. We see that for different weights and biases, the properties and effects of this function will vary drastically. It is precisely these parameters that we hope to tweak such that the whole FMP will perform the desired task.

At this point it may be helpful to introduce a motivating example. One application of an FMP is to classify images. For example, we can take images of handwritten digits 0-9 as our inputs, with labels corresponding to the number depicted. The image to the right shows such images (from the MNIST database) with their corresponding labels. Each image is an array of brightness values, with one value for each pixel. That array can be flattened into a vector, and used as the input to the FMP. We can specify the label vector using a **one-hot encoding**: for label value i, we set the label vector to

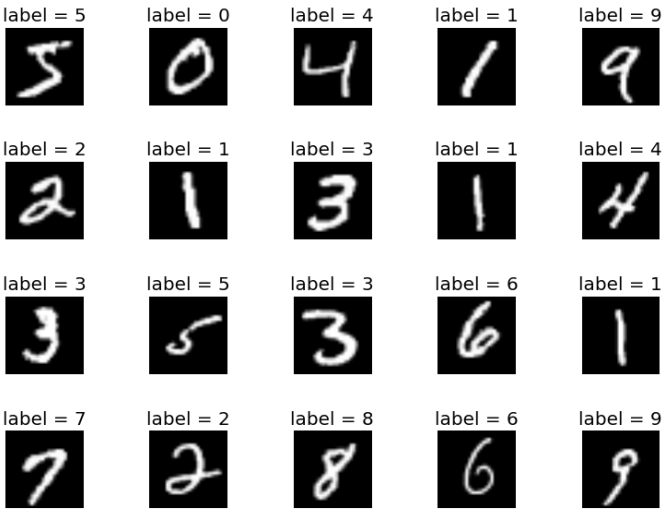

Figure 2: The MNIST Dataset
Image Source

be a vector of length 10 with zeros in each position except for a 1 in the ith position. That is the desired output of the network.

Now imagine you have the MNIST dataset of handwritten digits and you want to use an FMP to classify these images. Before you even get started, you would need to decide on the structure of your model: the number of layers, the number of nodes per layer, etc. The number of layers of a network is the **depth** of the network. You would then initialize your parameters, which is usually done randomly. To begin training, you would partition the dataset into your **training set** and your **testing set**. You will train using the training set, and then test the accuracy of the network using the testing set. That way, when you test your network, it will be operating on images it has never seen before.

Training, in machine learning, amounts to the minimization of a **loss function**. We think of the loss function as the disparity between what our network outputs and the label. Suppose you have a training dataset with n inputs. Let x^n denote the nth input vector and y^n denote its label vector. Let $f(x^i)$ denote the output vector of the FMP, given input vector x^i. Then we can measure the distance between the output and the label:

$$||f(x^i) - y^i||$$

This is not the function we want to minimize. We would like to minimize this function for every input in the dataset, not just one, so that the FMP trains over the whole dataset. So we define the loss function as the average distance between each output and its label. Let $\mathbb{P}$ denote the set of parameters of the FMP.

$$L(\mathbb{P}) = \frac{1}{n} \sum_i ||f(x^i) - y^i||$$

The loss function I have provided here is not the only way to measure loss, there are many other possible loss functions which I will not address here. Note that the loss function is indeed a function of the parameters, as our inputs and labels are fixed. This means that if we have a network with 1000 parameters, L is a function from $\mathbb{R}^{1000}$ to $\mathbb{R}$. In practice, networks may have hundreds of thousands or millions of parameters. We want to minimize $L(\mathbb{P})$ by sufficiently adjusting our parameters. The traditional methods of minimization in calculus will be hard to use here, since the number of parameters is so high. The minimization of $L(\mathbb{P})$ is done by an iterative algorithm called **backpropagation**, which I will not cover here.

Once we have optimized our parameters for the training set, we evaluate our network by measuring the loss over the testing set. Since the training and testing sets are from the same dataset, we expect that a low loss over the training set will correspond to a low loss over the testing set. If our loss over the training set is significantly lower than the than the loss over the testing set, we say that **overfitting** has occurred.

1.4 The Segmentation Problem

In the last section, we introduced neural networks and their applications to image classification. In image classification, our output is a number or small vector, which turns out to be an extremely limited way of gaining information from an image, in general. For example, our model which classifies handwritten digits would struggle to identify, say, images of handwritten phone numbers. Suppose we want a program which takes an image of a handwritten phone number and reads the number. There are 10^{10} possible 10 digit phone numbers. An FMP with output vectors with 10^{10} components, one class for each possible phone number, is not computationally feasible, and a training set which covers all possible labels would be difficult to create. Instead, we want to identify parts of the image corresponding to a handwritten 0, 1, 2, and so on. For each pixel in the image, we would like to determine if it is part of a handwritten digit, or part of the background. We have 11 total choices for what a pixel "belongs to," namely the digits $0-9$ or the background. We are no longer classifying the image, but rather classifying each pixel in the image. This is the problem of **image segmentation**, or semantic segmentation.

There are a few observations to be made about segmentation. If you are dealing with images of real things, i.e. images taken by a camera, and your images are relatively high resolution, then we can say that each pixel in the image is likely to be in the same class as the pixels around it. Also observe that each class will usually correspond to a physical object, which has edges. So we expect that after segmentation, we will see solid regions of pixels all with the same class, and borders between these regions. Also observe that we can easily extend image segmentation to video segmentation, as a video is just a sequence of segmentable images. With a video, we can track the movement of an object over time.

Let's discuss some applications of segmentation. Self-driving cars is an immediate and very interesting application. A self driving car is built with sensors attached on the outside of the vehicle, which could be microphones, cameras, radars, etc. Usually, their will be cameras positioned at the front, sides, and back of the car. These cameras record video which is segmented in real time by an on-board computer system via a trained neural network which identifies objects of interest in the video, such as people, other cars, road signs, buildings, road lines, traffic lights, etc. The system uses this information to control the vehicle (e.g. if the system detects a stop sign, it will cause the vehicle to engage the brakes). Figure 3 shows the segmentation of a road image.

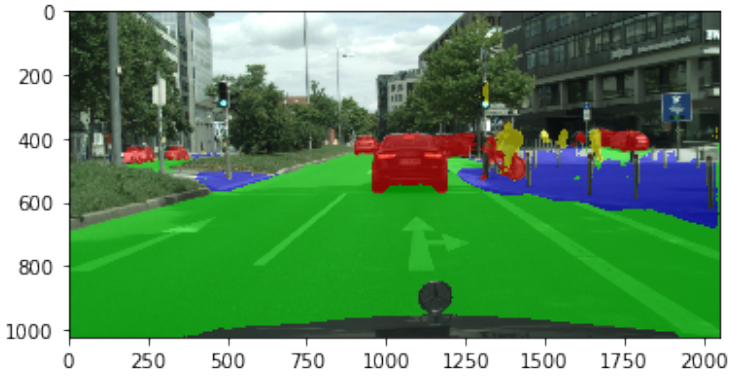

Figure 3: Segmentation of a Dashboard Image
Image Source

Another common application is the segmentation of biomedical images, in research and in practice. For example, we could imagine images of cross-sections of the human brain, and we could train a neural network to identify regions corresponding to tumors, something that, traditionally, only a trained human with years of education could do. In Figure 4, a deep convolutional neural network is being used to segment brain tumors using MICCAI's Multimodal Brain Tumor Image Segmentation Challenge (BraTS) dataset. One could also imagine an experiment in developmental biology in which a group of stem cells is placed on a petri dish, and video is recorded as the stem cells split and specialize into different types of cells. One may be concerned with tracking the cells as they split, so as to reconstruct a "family tree" of the cells, while also keeping track of their relative position, degree of specialization, and what type of cell they are specializing into. This is a segmentation problem, and a neural network could be used here.

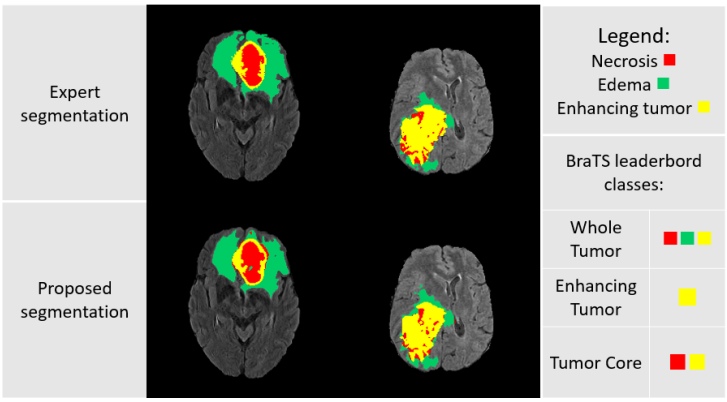

Figure 4: Segmentation of a Tumor in a Brain
Image Source

It turns out that the FMP model is not feasible for the task of image segmentation. The computational time required to train an FMP does not scale well as input images get larger. Furthermore, large FMPs have huge numbers of parameters, which can lead to overfitting. The **Convolutional Neural Network** (CNN) presents a more viable model for image segmentation. Furthermore, a CNN has a way of extracting local-region information, which is more important than per-pixel information in image segmentation, making it better suited for the task of image segmentation. The following section presents the ideas of the CNN.

2 Convolutional Neural Networks

2.1 The Convolution Operation in Image Processing

The convolution operation in image processing is the namesake of the Convolutional Neural Network. Convolution requires an input image and a **filter**, (also known as a kernel, window, stencil, or convolution matrix), which is a square matrix with dimension $(2a+1)$ by $(2a+1)$, for some $a \in \mathbb{N}$. Common values for a are 1 or 2, so the filter is generally much smaller than the image. We convolve the input image with the filter to produce the output image. Suppose we have a greyscale input image I, with dimension m by n. Then I is a matrix, and $\mathrm{I}_{j,k}$ denotes the component of I in the jth row and kth column. Let K denote the filter matrix. Imagine taking the small square matrix K and physically placing it over the big matrix I such that $\mathrm{K}_{a,a}$ (the center component of K) is stacked on top of $\mathrm{I}_{j,k}$, so that K is covering a $(2a+1)$ by $(2a+1)$ region of I. Then we can take the component-wise product of this region of I with K, to obtain a new matrix, and then we can add all the components of that matrix together to compute the **activation** of K at $\mathrm{I}_{j,k}$, which we will call $\mathrm{O}_{j,k}$. We can do this for each $\mathrm{I}_{j,k}$ in I to compute O, which is the output matrix. O is called an **activation map** of I convolved with K. We denote the convolution with $*$ and say $\mathrm{I} * \mathrm{K} = \mathrm{O}$. It is helpful to visualize the convolution as the filter "sliding" across the image, row by row, computing activations as it goes.

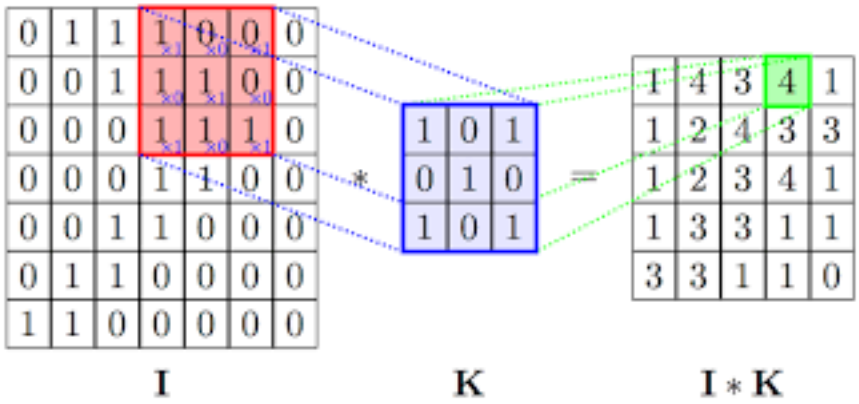

Image Source

There is one caveat to this process. For $a > 0$, we cannot overlay K at the top left corner of I. In fact, the upperleftmost component of I that we can overlay K on is $\mathrm{I}_{a,a}$. Because of this, the output image will be slightly smaller than the input image, with dimension $(m-2a)$ by $(n-2a)$. With this in mind, we can write the formula for convolution:

$$\mathrm{O}_{jk} = \sum_{s=-a}^{a} \sum_{t=-a}^{a} \mathrm{K}_{s,t} \cdot \mathrm{I}_{j+s,k+t}$$

$$a \leq j \leq m-a, \quad a \leq k \leq n-a$$

If we would like the output image to be the same size as the input image, we can apply **padding** to the input image before convolution. For example, we could pad the input image

with a layer of zeros which is a components thick, to ensure that the activation map is the same size as the original input image. Using zeros is called **zero padding**. We could also use **reflective padding**, where the pad is formed by reflecting the image over its border. Reflective padding is commonly used in applications.

We may want to convolve images with multiple channels, such as color images. In this case, I is a m by n by c tensor. We can extend our notion of convolution in the obvious way: K becomes a $(2a+1)$ by $(2a+1)$ by c filter tensor, and do pointwise multiplication, summing the $c(2a+1)^2$ products to get the activation. Note that even when our input image has multiple channels, the activation map is always greyscale.

Lets examine an example of an application convolution. Consider the filter matrices:

$$\mathrm{K}_1 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathrm{K}_2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Here, K_1 and K_2 detect vertical edges and horizontal edges in of bright objects in an image, respectively. Consider an image of a white square on a black background, where white pixels have a brightness value of 1 and black pixels have a brightness of 0. Applying these filters on a region entirely contained in the white square or in a region entirely contained in the background will result in an activation of zero. However, applying K_1 on a region around the left edge of the square will result in a high activation. Similarly, applying K_2 around the top edge of the square will result in a high activation. Also note that K_1 and K_2 will have low activations at the right and bottom edges of the square, respectively. In general, we can say that the absolute value of the activation of K_1 with a region of an image measures how much that region looks like a vertical edge. K_1 and K-2 are the **Sobel kernels**. Define $G(R) = \sqrt{(\mathrm{K}_1 * R)^2 + (\mathrm{K}_2 * R)^2}$, where R is some 3 by 3 local region of an input image I. Then $G(R)$ quantifies how much R looks like an edge. Applying G to every R in I results in an activation map which detects edges in an image. G is the **Sobel operator**. This is the basis of traditional edge detection algorithms such as the Canny edge detector.

Figure 5: An image of a steam engine, with its activation map under the Sobel operator

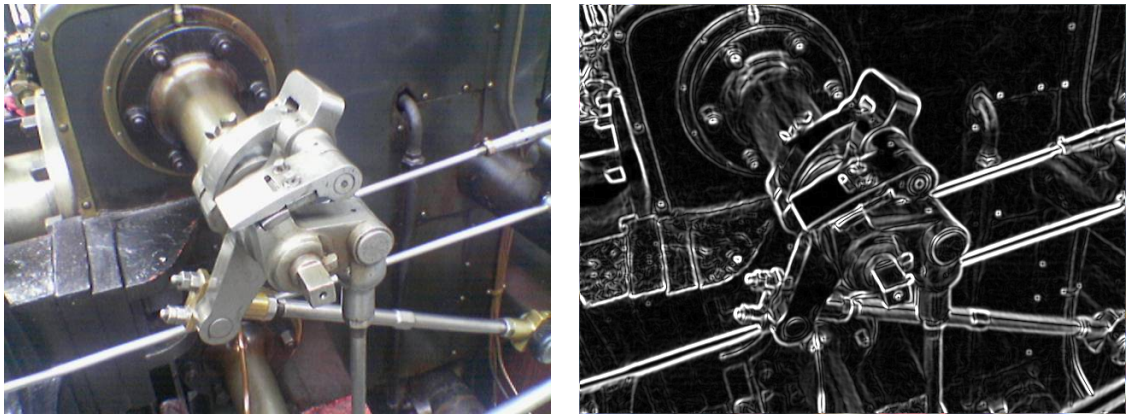

Image Source

We can modify the convolution operation to produce a **dilated convolution**. A dilated convolution uses a filter which is dilated by a **dilation factor** d. A filter dilated by a dilation factor d is only nonzero at components which are a distance d from the center. To illustrate this, let's look at an example of a 3 by 3 filter matrix, the matrix with dilation factor 2, and the matrix with dilation factor 3.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 0 & 9 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 5 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 8 & 0 & 0 & 9 \end{bmatrix}$$

We see that dilating a filter increases its size. In general, dilating a filter with dimension $(2a+1)^2$ by dilation factor d results in a matrix with dimension $(2ad+1)^2$. When calculating an activation, we refer to the region in the input image as the **local region**. We refer to the components in the local region being multiplied by nonzero components in the filter as the **receptive field**. Since the zeros in the filter do not affect the activation, we can equivalently think about dilation as a stretching of the receptive field.

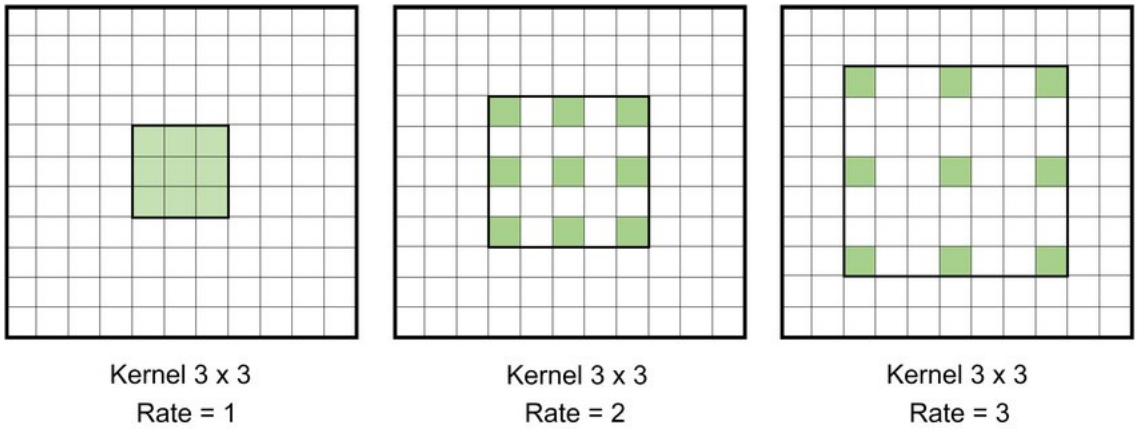

Figure 6: The receptive field of a dilated filter, where "Rate" is the dilation factor
Image Source

We can also compute a **strided convolution** by specifying a **stride length** s. A strided convolution is like a standard convolution, but some of the the local regions are skipped. After computing an activation, the filter moves s pixels across the row before computing the next activation, skipping the $s-1$ pixels in between. After finishing a row, the filter then moves s rows down, skipping the $s-1$ rows in between. The result is an activation map much smaller than the original input. In fact, the activation map has $(m-2a)/s$ rows and $(n-2a)/s$ columns. We now have all the tools to describe convolutional neural networks.

2.2 Convolutional Neural Networks and Image segmentation

A Convolutional Neural Network (CNN), similarly to a Feedforward Multilayer Perceptron (FMP), is a sequence of layers which perform operations on data as it moves through the network. We think of the input to an FMP as a vector, which may correspond to a flattened image. We think of the input of a CNN as an unflattened tensor (a multi-dimensional array). Our input will usually be an image tensor x of dimension m by n by c, where c is usually 3, corresponding to the three color channels. The output of a CNN depends on what it is being used for. CNNs can be used for image classification, in which case, the output will usually be a vector with a one-hot encoding, just like a FMP.

CNNs can also be used for image segmentation. The output will be an m by n by h tensor, where h is the number of classes in the segmentation. Let y^i denote the ith label in our dataset, with dimension m by n by h. y^i has one channel for each class, where, in each channel, a value of 1 at position j, k signifies that the pixel $x^i_{j,k}$ belongs to that class. Then $y^i_{j,k}$ a vector corresponding to classification of the pixel $x^i_{j,k}$, which uses a one-hot encoding. When generating images of the output of image segmentation, such as those displayed in Section 1.4, one assigns an arbitrary color to each channel of the output image. Then we color pixel $x^i_{j,k}$ with the color corresponding to the position of the 1 in the vector $y^i_{j,k}$.

There are several types of layers a CNN can have: convolutional layers, pooling layers, transposed convolutional layers, and dense layers are some of the common layer types. Of primary interest are convolutional layers. A **convolutional layer** takes as input a tensor, and computes convolutions of that tensor with several filters. Convolutions of the input tensor with each filter result in an activation map for each filter. These convolutions can be done with a stride or a dilation. Each filter is also associated with a bias, which is added to each component in the activation map. Then, a nonlinear activation function (e.g. sigmoid, tanh, or ReLU) is applied to each component. The resulting greyscale image is called the **feature map** of the input with the filter, bias, and activation function. The feature maps for each filter are then put together as channels of the output tensor, which is then input to the subsequent layer. The filter components and biases of this layer are the trainable parameters of the layer. In practice, convolution layers may be associated with 10 to 100 filters, with some models using as many as 1000 filters in a single layer. Usually, all filters in the same layer will be the same size and will use the same activation function, and padding may be applied to the input tensor so that the output tensor will be the same size.

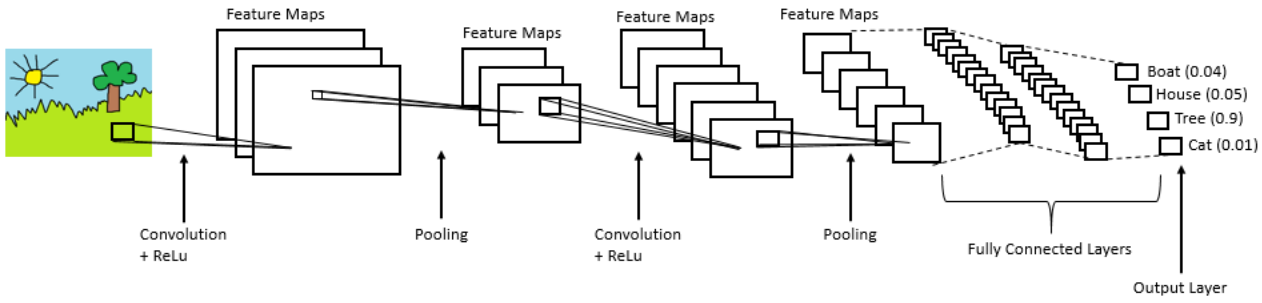

Figure 7: A simple CNN model used for image classification
Image Source

Dense layers, or fully-connected layers, are exactly the type of layer used in the FMP. The input is flattened into a single vector, a weight matrix is applied, a bias vector is added, and an activation function is applied. In a CNN model being used for image classification, you will usually find that the last few layers are dense layers, since they output vectors, and the the output of an image classification network is usually a one-hot vector.

Pooling layers are also commonly used. The purpose of a **pooling layer** is to decrease the size of the input tensor, to reduce redundancy and computation time. Pooling layers perform an operation similar to convolution, using **windows**, which are like filters. Windows do not contain values, and do not compute a component-wise product. Instead, applying a window to a local region of the input tensor computes some function of the local region, such as the mean (**average pooling**), or the max (**max pooling**). Windows are not necessarily square (though they often are) and do not necessarily have odd side lengths. During pooling, the local regions in the input tensor do not overlap, resulting in an output with less columns and rows. If an s by t pooling window is used, the number of rows gets scaled down by s and the number of columns by t. In image classification, pooling layers are often used to reduce feature map size, thereby reducing the number of parameters in the final dense layers. Pooling layers are also used in modern models, as we will see later.

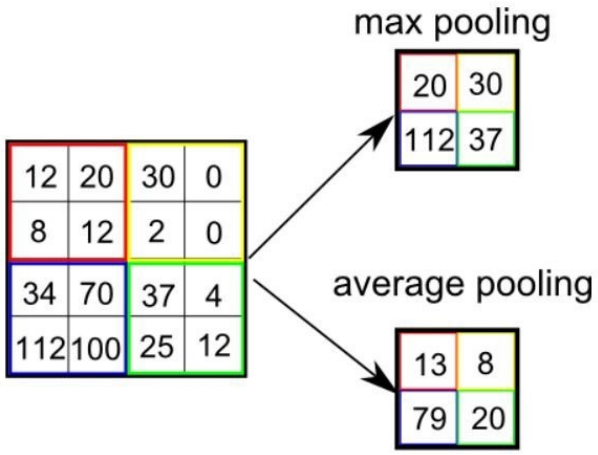

Figure 8: A pooling operation
Image Source

Let's address the **transposed convolutional layer**. This layer performs a transposed convolution, also known as a deconvolution or up-convolution. The purpose of the transposed convolution is to up-sample the data, so that the output is larger than the input. We think of transposed convolution as, in some sense, reversing a convolution, so that we begin with a convolution output, and we produce the original image that was convolved. To compute a transposed convolution, we specify a padding width and a stride length s. Then, we pad the input with zeros, and insert s columns of zeros between our input columns, and s rows of pixels between our input rows.

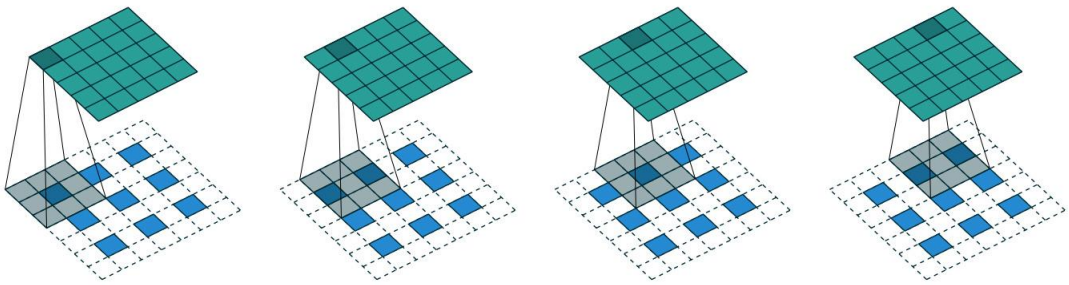

Figure 9: A transposed convolution with stride length 1 and padding width 1
Image Source

Like the convolutional layer, a transposed convolutional layer uses several filters to produce several activation maps. However, no bias or activation function is applied. The filter components are the trainable parameters of this layer. Transposed convolutions are often used in segmentation models to ensure that the output image is the same size as the input image.

As a final note on the convolutional layers, we posit that the computation carried out by a convolutional layer can be written as a matrix multiplication. Recall that for an input image tensor I with dimension $m \times n \times c$ being convolved with a filter K_i with dimension $2a+1 \times 2a+1 \times c$, the output image matrix O will be of dimension $m-2a \times n-2a$. This implies that $(m-2a)(n-2a)$ activation calculations are taking place, one for each $\mathrm{O}_{j,k}$. For each of these calculations, we are taking the pointwise product of K_i with with a local region of in I, which we will denote $R_{j,k}$. We can flatten $R_{j,k}$ and K_i into a vector using im2col, and the pointwise product becomes a vector dot product. The resulting scalar is the activation of component $0_{(j-a),(k-a),i}$. We can arrange all of the local regions of I as columns of a matrix, which we will denote A. Then, O_i can be computed by left-multiplying A with the row vector K_i, and relocating the activations to their appropriate position in the activation map. Furthermore, we can write all the filters K_i as rows of a matrix K. Then the matrix product KA carries out all the necessary dot products between filters and local regions. The resulting matrix can be reshaped as necessary to form the tensor of activation maps. Being able to write the computation of a convolutional layer as a matrix multiplication is significant, as matrix multiplications have been computationally optimized.

2.3 A Review of Selected Models

In this section, we will examine three influential papers, each introducing a **Deep Convolutional Neural Network** (DCNN) model with novel methods or techniques. Before 2015, CNN models for image segmentation were similar to those used for image classification, like the model shown in Figure 7. These models often had huge numbers of parameters, required enormous amounts of training time, were only viable on "nice" data, and did not capture information at different scales well. The novel methods presented in this section will address these problems by modifying the fundamental structure of the classic CNN model.

We will begin by examining "U-Net: Convolutional Networks for Biomedical Image Segmentation" (2015). Ronneberger et al. build on the work of Long et al. with their U-net, which won the ISBI cell tracking challenge in two categories for biomedical segmentation tasks. The U-Net is a refined version of the dominant DCNN segmentation model of 2014/2015, the downscaling/upscaling model.

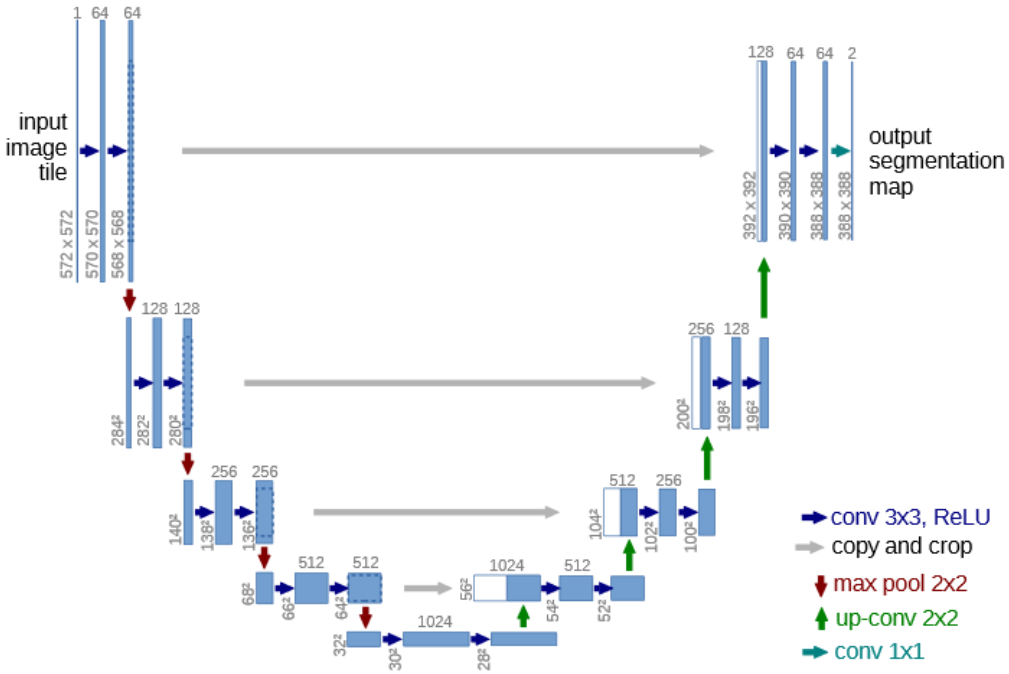

Figure 10: The U-Net
Image Source

As you can see, the figure looks like a "U," hence the name. This network is enormous, with 19 standard convolutional layers, 4 max pooling layers, and 4 transposed convolutional layers. Downscaling is used in the first half of the network, while the second half uses transposed convolutions, also known as up-convolutions, to upscale the image. Before downscaling, the 3×3 convolutional filters can capture information at small scales in the image, such as edges. After downscaling, near the "bottom" of the U, a 3×3 filter captures a huge amount of context in the image, allowing for learning to take place at a higher level. As the activations maps become more coarse with downscaling, more filters are used per

layer, as there are many more useful patterns to look for when presented with more context. During upscaling, activation maps from the previous layer are combined with the activation maps from the corresponding layer from the downscaling stage. This is represented with grey arrows in Figure 10. This catenation allows convolutional layers in the upscaling half of the model to simultaneously learn from local information and contextual information. The network can then segment each pixel with precision while taking in context from multiple scales. The inclusion of multiple convolutional layers with many filters during the upscaling portion of the network is what separates the U-net from the model presented by Long et al.

Ronneberger et al. also claim that their network requires less annotated samples to train on, instead relying heavily on the use of data augmentation. Rotations, shifts, and average brightness variations are applied to the data set to generate more data to train on. Additionally, elastic deformations are applied to the images, allowing the network to learn invariance to these deformations. Elastic deformations are a common natural occurrence in biomedical images, and can be simulated efficiently. Furthermore, pixels corresponding to the border between segmentation classes are determined manually and given a greater weight in the loss function during training, forcing the network to learn to segment the borders.

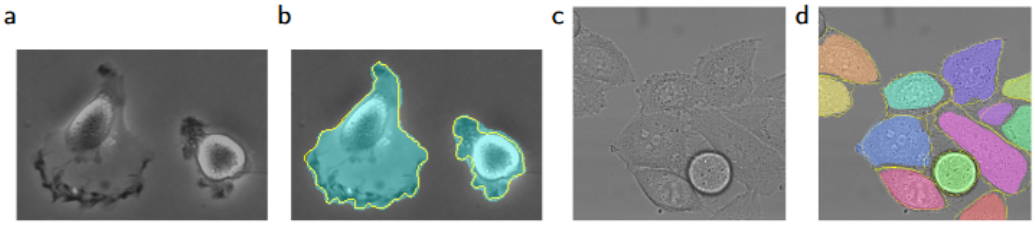

Fig. 4. Result on the ISBI cell tracking challenge. (**a**) part of an input image of the "PhC-U373" data set. (**b**) Segmentation result (cyan mask) with manual ground truth (yellow border) (**c**) input image of the "DIC-HeLa" data set. (**d**) Segmentation result (random colored masks) with manual ground truth (yellow border).

Figure 11: U-Net segmentation of images from ISBI 2015 datasets
Image Source

We will now discuss Yu and Koltun's "Multi-Scale Context Aggregation by Dilated Convolutions" (2016). Yu et al. demonstrate that dilated convolutions allow activations to have a larger receptive field with less layers than standard convolutions, capturing information at different scales. A 3×3 convolution filter with dilation factor d has the same number of parameters associated with it as a 3×3 filter with dilation factor 1, but captures information in a local region of size $(2d + 1)^2$. If we consider multiple layers with dilated convolutions in sequence, the receptive field of each pixel in the activation maps of the lth layer increases exponentially as l increases linearly, whereas without dilations, the receptive field increases linearly.

In Figure 12, a, b and c represent activation maps in sequential layers. The red dots represent the points where a filter is applied to a local region. In b, a dilation factor of 2 is used, and in c, a dilation factor of 4 is used. The blue represents the size of the area in a which affects the pointwise multiplication being carried out by the filter. In a, we see that the output of the filter applied at that local region is only dependent on that local region. In

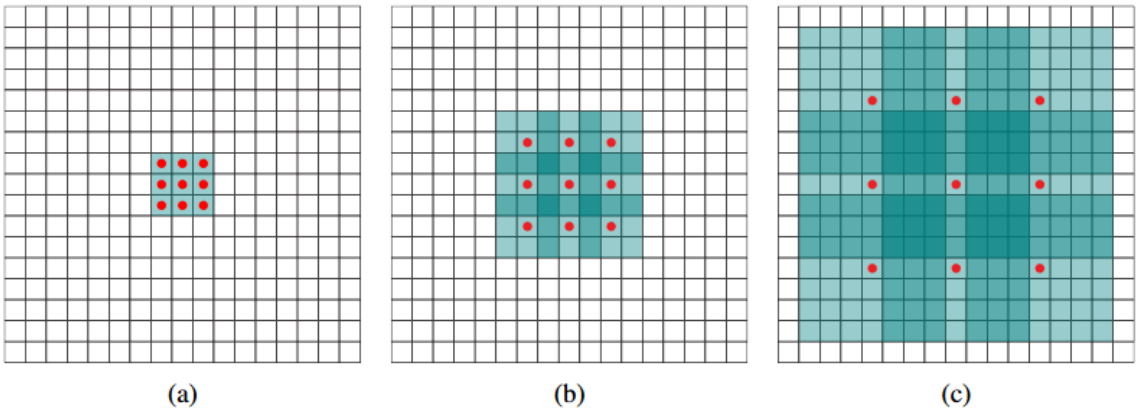

Figure 12: The receptive field increases exponentially with multiple dilations
Image Source

b, each pixel was determined by the application of a 3×3 filter, and so applying a filter with dilation factor 2 means that the resulting activation will be associated with a 7×7 receptive field. Similarly, in c, each pixel is determined from a 7×7 receptive field, so the resulting activation is dependent on a 15×15 receptive field in a. By increasing the dilation factors in this way, the length and width of the receptive field double with each convolutional layer.

Layer	1	2	3	4	5	6	7	8
Convolution	3×3	3×3	3×3	3×3	3×3	3×3	3×3	1×1
Dilation	1	1	2	4	8	16	1	1
Truncation	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Receptive field	3×3	5×5	9×9	17×17	33×33	65×65	67×67	67×67
Output channels								
Basic	C	C	C	C	C	C	C	C
Large	$2C$	$2C$	$4C$	$8C$	$16C$	$32C$	$32C$	C

Figure 13: A summary of Yu et al.'s model
Image Source

The above table summarizes Yu et al.'s architecture for their "Basic" and "Large" models. Here, "truncation" refers to a $\max(\cdot,)$ activation function being applied. We see that the receptive field size increases exponentially, a phenomenon described by Yu et al. as "context aggregation." Yu et al. use a "front-end" model based on the model of Long et al. and Chen et al., applying their model to the resulting activation maps. Yu et al.'s model outperforms the leading models of 2015 and 2014 on various non-biomedical segmentation datasets.

Finally, let's examine "A mixed-scale dense convolutional neural network for image analysis" (2017) by Pelt and Sethian. They introduce the nontraditional Mixed-Scale Dense (MS-D) network, which achieves higher accuracy on biomedical and nonbiomedical segmentation tasks than the U-net as well as other state-of-the-art models. The MS-D uses one dilated filter at a time, computing activation maps with 1 channel. The model utilizes reflective padding to preserve image dimensions, so that the dimensions of the input image, output image, and all feature maps are the same. While computing a new feature map, all

previously computed feature maps are put together as channels of a tensor and convolved on, which is possible because all previous feature maps are the same size. This is how the MS-D maximally (re)uses its feature maps. Although each activation map is 1 channel deep, multiple activation maps are computed per layer, using different filters which may have different dilation factors, and so we think of them as being different activation maps rather than channels of the same activation map. The MS-D has d noninput and nonoutput layers, and computes w activation maps per layer. A set of possible dilation factors is determined before training, and during training the dilation factor that each activation map uses is learned. Pelt et al. do not specify exactly how learning these dilation factors takes place. The output image, like the activation maps, is computed by a convolution on all previous activation maps, but uses a 1×1 filter rather than a 3×3 filter, making it a linear combination of all the activation maps.

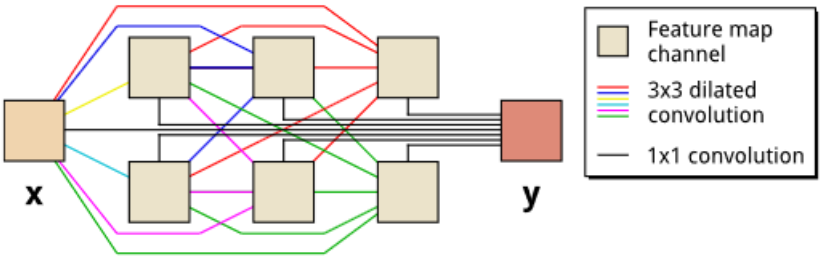

Figure 14: The MS-D network model
Image Source

As you can see, each activation map is densely connected to the other activation maps (except the maps in the same channel) by convolution operations. The dilated convolutions allow for context aggregation and convolution at multiple scales, hence the name. The advantage of reusing activation maps in this way is a massive decrease in the number of parameters in the model, while maintaining quick computation time during segmentation.

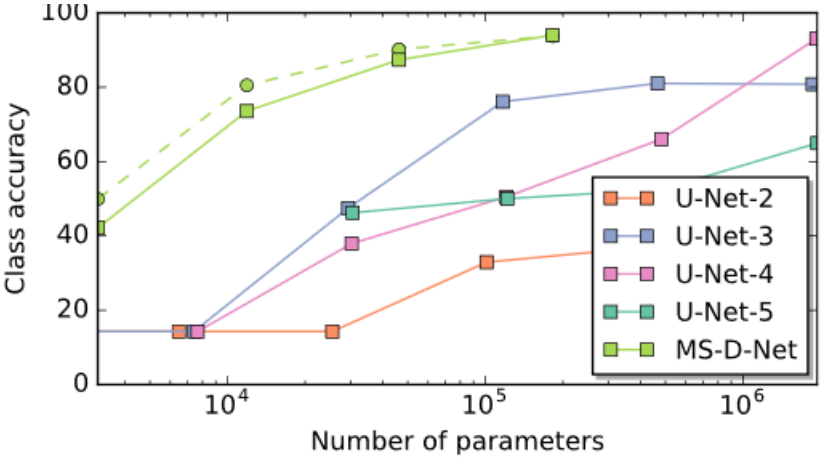

Figure 15: MS-D segmentation efficiency compared to the U-net
Image Source

Figure 15 shows the MS-D model's accuracy on segmentation of a simulated dataset compared to the U-net, as a function of the number of parameters. Here, "U-Net-k" refers to a U-net with k pooling layers and k transposed convolutional layers (Figure 10 shows a U-Net-4). The simulated data is a set of images of overlapping squares and circles of various shapes and sizes with different textures and patterns. Segmentation labels have multiple classes for different types of squares and circles, but some are ignored and treated as background.

The model also performs well on real-life datasets. Here are results from the network trained on the CamVid dataset, a collection of images of road and driving scenes with segmentation labels. The MS-D obtains higher accuracy than the dominant models of 2015 and 2014, while using an order of magnitude less parameters.

Table 1. The number of trainable parameters (Pars) in millions (M), global accuracy (GA), and class accuracy (CA) for the CamVid test set

Method	Pars (M)	GA	CA
MS-D-Net (100 layers)	0.048	85.1	56.8
MS-D-Net (200 layers)	0.187	87.0	63.9
U-Net (3 scaling operations) (5)	1.863	83.2	50.4
U-Net (4 scaling operations) (5)	1.926	85.5	48.4
SegNet-Basic-EncoderAddition (4)	1.425	84.2	56.5
SegNet-Basic (4)	1.425	84.0	54.6
Boosting + Detectors + CRF (31)		83.8	62.5
Super Parsing (32)		83.3	51.2

Figure 16: MS-D segmentation efficiency compared to the U-net
Image Source

3 A Simple Segmentation

As part of my exploration in CNNs, I implemented a basic segmentation model using TensorFlow to perform a relatively simple segmentation on synthetic data. I created synthetic data using DataTank, which consisted of images of a circle with a horizontal gradient ramp and on a background with a vertical gradient ramp. In the figure to the right, a greyscale image (blue to red is black to white) along with its segmentation label is shown. With such a simple dataset, one could easily write a script to detect the direction of the gradient ramp and segment accordingly. Building and training a CNN for this task was an educational exercise in convolutional neural networks and TensorFlow.

My model consists of a sequence of 6 convolutional layers, each using reflective padding, a 3×3 filter, and a tanh activation function, with no strides or dilation. I used ADAM for stochastic gradient descent during training, and used mean euclidean distance for the loss function. Note that I did not use any rounding to produce segmented images with the network's "best guess." I trained the network on an Intel i5 CPU processor for roughly 25 hours. I found that my model's accuracy plateaued at about 85%, after roughly 5 hours of training.

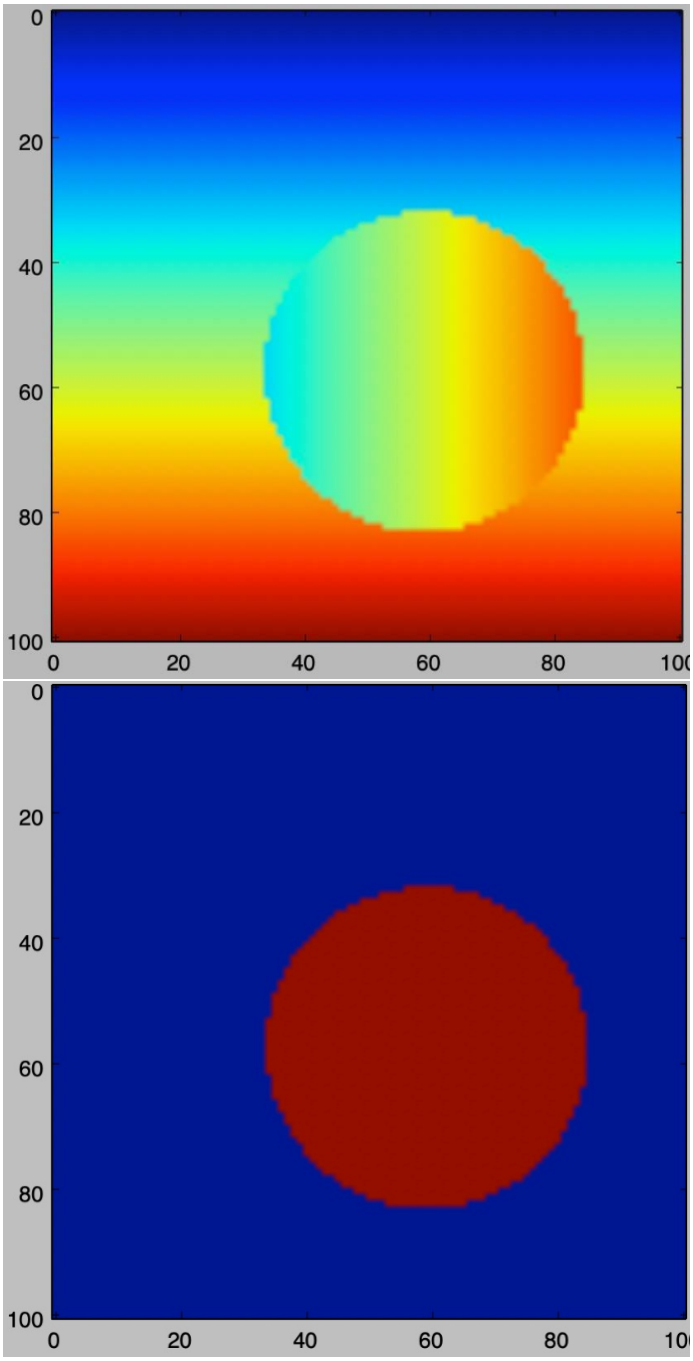

Figure 17: One of the synthetic images with its corresponding label

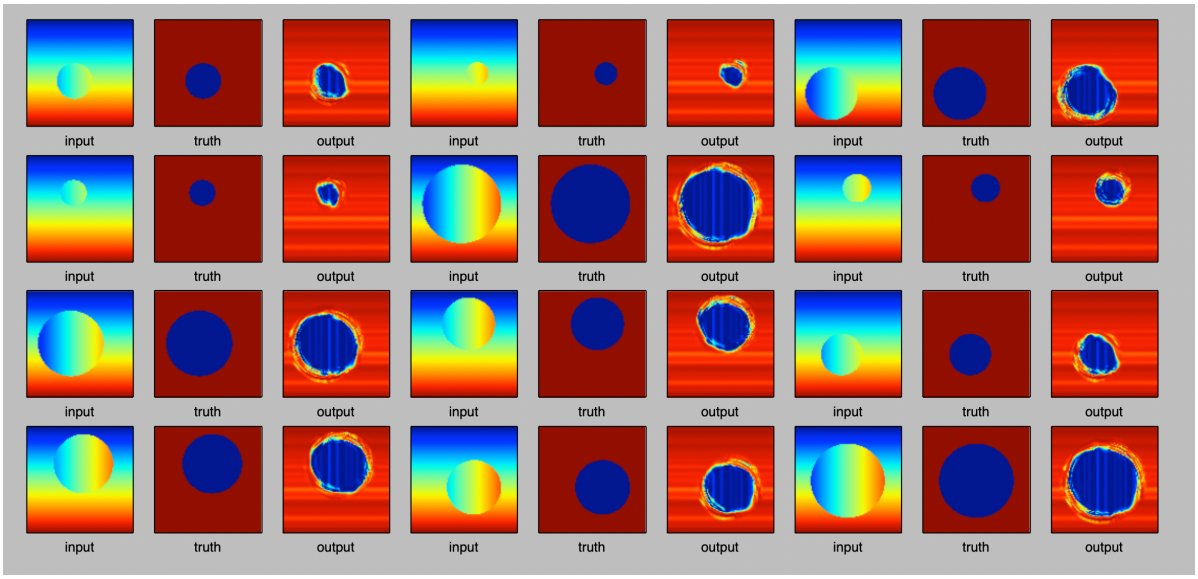

Figure 18: My model's results after 2 hours of training

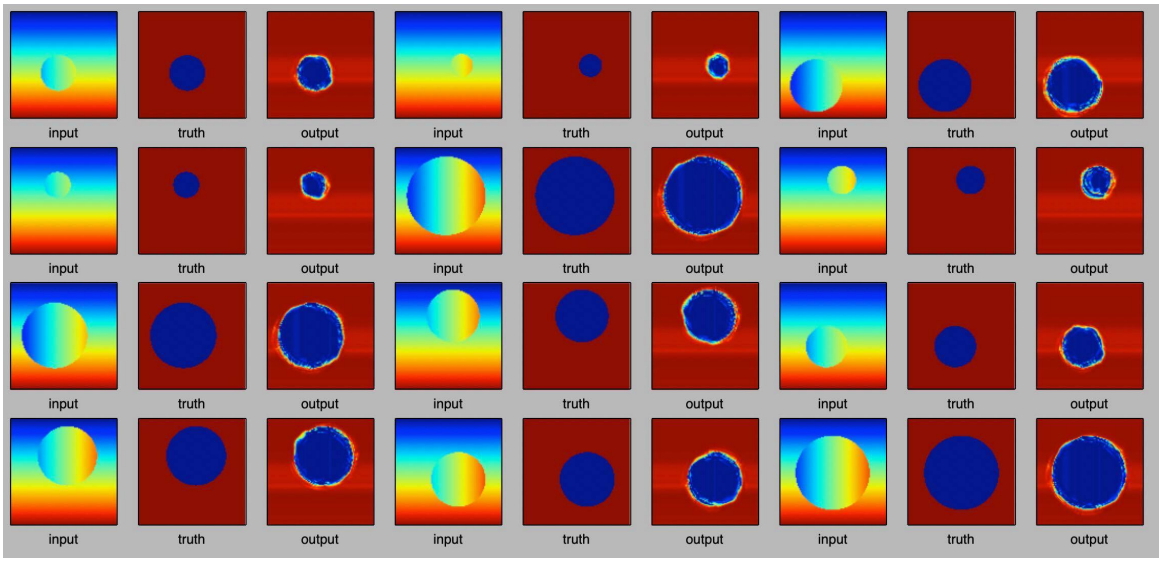

Figure 19: My model's results after 5 hours of training

In further experimentation, I would like to implement models which use multiple previous activation maps in computation of activation maps, such as the U-net or the MS-D, as this seems to be the key to modern segmentation. I would also like to experiment with more complicated synthetic data sets such as those used by Pelt et al.