Chapter 16
Medical Image Recognition: An Explanation and Hands-On Example of Convolutional Networks

Dianwen Ng and Mengling Feng

Abstract This chapter consists of two sections. The first part covers a brief explanation of convolutional neural networks. We discuss the motivation behind using convolution in a neural network and present some of the common operations used in practice alongside with convolution. Then, we list some variations of the convolution layer and we set the guidelines as to when the types of CNN layer are used to manage certain tasks. In the latter section, we will demonstrate the application of a CNN on skin melanoma segmentation with the written approaches and steps to train our model. We provide succinct explanations and hopefully, this will give a better understanding of CNNs in the context of medical imaging. We encourage readers to follow along on their own and try the actual code available from the GitHub repository provided in the second section.

Keywords Convolutional networks · Neural networks · CNNs · Image segmentation · Image processing · Skin melanoma

Learning Objectives

- General overview and motivation of using convolutional neural networks
- Understanding the mechanisms and the computations of a convolutional neural networks model
- Introduction to gradient descent and back-propagation
- Application of convolutional neural networks on real medical images using python programming.

D. Ng · M. Feng (✉)
Saw Swee Hock School of Public Health, National University Health System, National University of Singapore, Singapore, Singapore
e-mail: ephfm@nus.edu.sg

D. Ng
e-mail: ephndw@nus.edu.sg

L. A. Celi et al. (eds.), *Leveraging Data Science for Global Health*,
https://doi.org/10.1007/978-3-030-47994-7_16

16.1 Introduction

The power of artificial intelligence (AI) has thrust it to the forefront as the next transformative technology in conventional business practices. To date, billions of dollar have been invested to drive the accuracy and predictability of AI algorithms in acquiring information, processing, and understanding images like MRI, CT, or PET scans. While much of the data in clinical medicine continues to be incredibly obfuscated and challenging to appropriately leverage, medical imaging is one area of medicine where the pure processing power of today's computers have yielded concrete beneficial results. In 2017, Andrew Ng, the former head of AI research at Baidu and adjunct professor at Stanford University, reported in his research that his group had developed an algorithm that outperformed trained radiologists in identifying pneumonia (Rajpurkar et al. 2017). Because computers have the capacity to process and remember countless petabytes of data more than a human could in their lifetime, machines have the potential to be more accurate and productive than even a trained clinician. Meanwhile, we also see a growing number of AI start-ups who have created algorithms that achieve commercial operating standards in spotting abnormalities in medical images. Be it detecting or diagnosing various diseases ranging from cardiovascular and lung diseases to eye diseases, these AI companies have been rendering services to help health providers to manage the ever increasing workload. Rejoice to the world as we celebrate the triumph of AI in opening endless possibilities in the field of medical imaging.

Medical imaging seeks to visualize the internal structures hidden by the skin and bones, providing clinicians with additional information to diagnose and treat patients. Standard practice establishes a database of normal physiology and anatomy to potentially differentiate the abnormalities in disease. Imaging is often crucial in detecting early stages of disease, where obvious signs and symptoms are sometimes obscured. AI can now process millions of data points, practically instantaneously, to sort through the troves of medical data and discern subtle signs of disease. The machine does this using a class of deep learning networks called "convolutional networks" to simulate the learning of how humans would perceive images. This allows the machine to gain a high level understanding from digital images or videos. In this case, we will focus more on how to build a machine to process medical images.

Convolutional neural networks (CNN) are a specific group of neural networks that perform immensely well in areas such as image recognition and classification. They have proven to be effective in producing favourable results in several medical applications. Such examples include skin melanoma segmentation, where machines use CNNs to detect lesion area from the normal skin. Certainly, we can also apply these to MRI or CT scan for problems like brain tumour segmentation or classification of brain cancer with limitless application to medical disorders. The purpose of this article serves as a guide to readers who are interested in studying medical images and are keen to find solutions that assist with diagnosis through artificial intelligence.

We present a semantic-wise CNN architecture in Fig. 16.1 as a motivation to this chapter. We will learn how to build such a model and put them into practice in segmenting a region of skin lesion. Leading up to that, we will explore the main

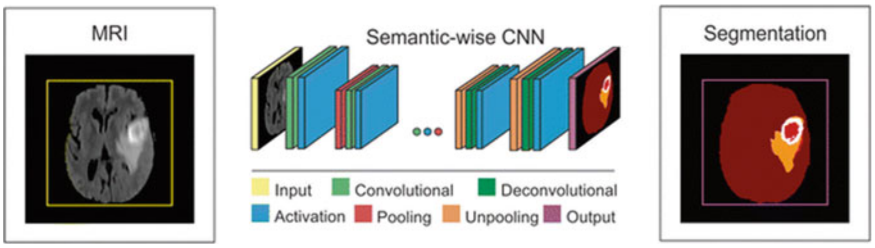


Fig. 16.1 Semantic-wise CNN architecture for brain tumour segmentation task. The left is the input of brain MRI scan and the right is the predicted output by the CNN model. *Source* Akkus et al. (2017)

mechanisms of the networks and show that these are the fundamentals to most state of the art CNN models. However, our goal is to provide the readers with an introduction to the basic structures of the networks. Therefore we will not go beyond the basics nor cover more advanced models that obtain higher performance.

This chapter consists of two sections where the first part covers an intuitive explanation to convolutional networks. We discuss the motivation behind using convolution in a neural network and we talk about some of the common operations used in practice alongside with convolution. Then, we list some variations of the convolution layer and we set the guidelines as to when the types of CNN layer are used to manage certain tasks. In the latter section, we have demonstrated the application of CNN on skin melanoma segmentation with the written approaches and steps to train our model. We provide succinct explanations and hopefully, this will give a better understanding of CNN in the context of medical imaging. We strongly encourage readers to try the code on their own from the GitHub link provided in the second section.

16.2 Introduction to Convolutional Networks

Every image can be represented by a matrix of pixel values. A color image, can be represented in three channels (or 2D matrices) stacked over each other in the RGB color space in which red, green and blue are combined in various ways to yield an extensive array of colours. Conversely, a greyscale image is often represented by a single channel with pixel values ranging from 0 to 255, where 0 indicates black and 255 indicates white.

16.2.1 Convolution Operation

Suppose that we are trying to classify the object in Fig. 16.2. Convolutional networks allow the machine to extract features like paws, small hooded ears, two eyes and so on from the original image. Then, the network makes connections with all the extracted information to generate a likelihood probability of its class category. This feature extraction is unique to CNN and is achieved by introducing a convolution filter, or

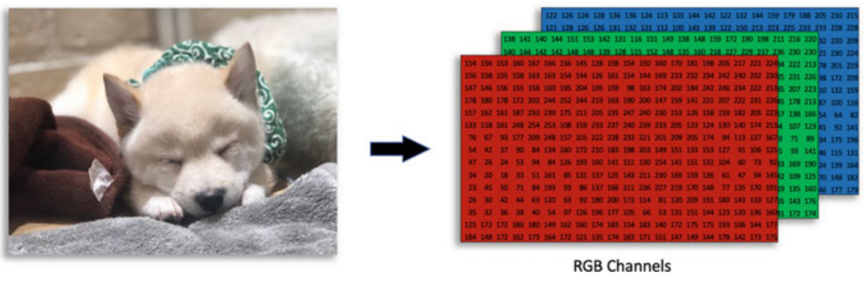


Fig. 16.2 Representation of the RGB channels (Red, Green and Blue) of a dog. Each pixel has a value from 0 to 255

the kernel, which is defined by a small two dimensional matrix. The kernel acts as feature detector by sliding the window over the high-dimensional input matrices of the image. At each point, it performs a point-wise matrix multiplication and the output is summed up to get the elements to the new array. The resulting array of this operation is known as the convolved feature or the feature map. A feature map conveys a distinct feature drawn from the image activated by the kernel. In order for our networks to perform, we often assign sufficiently large number of kernels in the convolution function to allow our model to be good at recognizing patterns in the unseen images.

Besides, after every convolution, the resolution of the output becomes smaller as compared to the input matrix. This is due to the arithmetic computation using a sliding window of size greater than 1×1. As a result, information are summarized at the cost of losing some potentially important data. To control this, we can utilize zero padding which appends zero values around the input matrix.

To illustrate, we refer to the example as shown below. The dimension of the original output after convolution is a 3×3 array. For us to preserve the original resolution of the 5×5 matrix, we can add zeros around the input matrix to make it 7×7. Then it can be shown that the final output is also a 5×5 matrix. This does not affect the quality of the dataset as adding zeros around the borders does not transform nor change the information of the image.

A formula to calculate the dimension of the output from a square input matrix is given as follows (Fig. 16.3),

$$Width_{\text{feature map}} = \frac{Width_{\text{input}} - Width_{\text{kernel}} + 2(padding)}{stride} + 1$$

From Figure 16.4, we show the typical learned filters of a convolutional network. As mentioned previously, the filters in convolutional networks extract features by activating them from the matrices. We would like to highlight that the first few layers of the network are usually very nice and smooth. They often pick-up lines, curves and edges of the image as those would fundamentally define the important elements that are crucial for processing images. In the subsequent layers, the model will start to learn more refined filters to identify presence of the unique features.

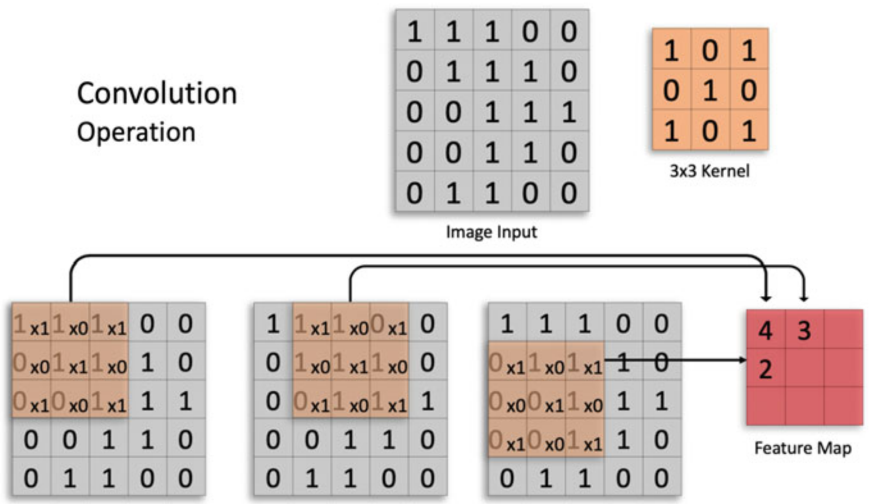


Fig. 16.3 A 3 × 3 kernel is introduced in this example. We adopt stride of 1, i.e. sliding the kernel by one pixel at a time to perform the convolution operation. Note that the overlay region of the kernel and the input matrix over the matrix multiplication is called the receptive field

In comparison to the traditional neural network, convolution achieves better image learning system by exploiting three main attributes of a convolutional neural network: *1. sparse interactions*, 2. *parameter sharing* and 3. *equivariant representation*.

Sparse interactions refer to the interactions between the input and the kernel. It is the matrix multiplication as described earlier, and sparse refers to the small kernel since we construct our kernel to be smaller than the input image. The motivation behind choosing a small filter is because machines are able to find small, meaningful features with kernels that occupy only tens or hundreds of pixels. This reduces the parameters used, which cuts down the memory required by the model and improves its statistical computation.

Furthermore, we applied the same kernel with the same parameters over all positions during the convolution operation. This means that instead of learning a distinctive set of parameters over every location, machines only require to learn one set of filter. As a result, it makes the computation even more efficient. Here, the idea is also known as parameter sharing. Subsequently, combining the two effects of sparse interaction and parameter sharing, we have shown in Fig. 16.4 that it can drastically enhance the efficiency of a linear function for detecting edges in an image.

In addition, the specific form of parameter sharing enables the model to be equivariant to translation. We say that a function is equivariant if the input changes and the output changes in the same way. In this way, it allows the network to generalise texture, edge and shape detection in different locations. However, convolution fails to be equivariant to some transformations, such as rotation and changes in the scale of an image. Other mechanisms are needed to handle such transformations, i.e. batch normalisation and pooling.

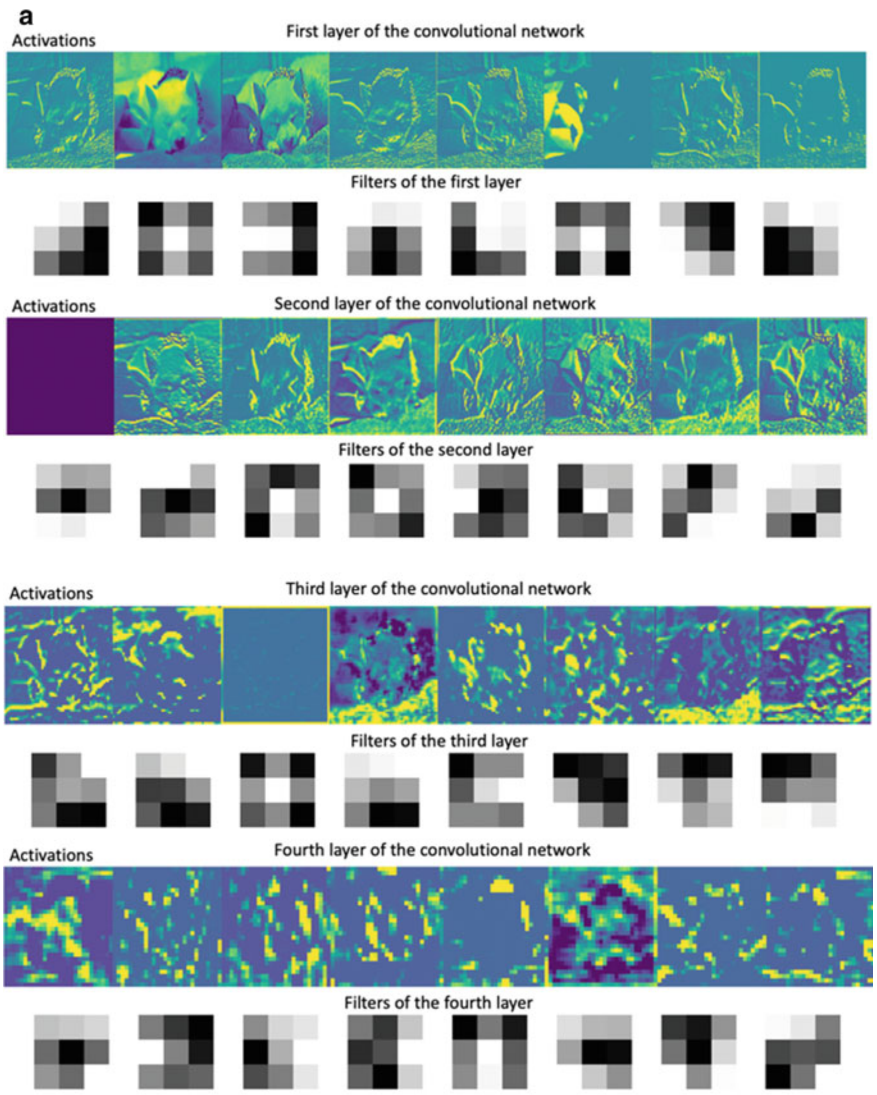

a
Activations
First layer of the convolutional network
Filters of the first layer
Activations
Second layer of the convolutional network
Filters of the second layer
Activations
Third layer of the convolutional network
Filters of the third layer
Activations
Fourth layer of the convolutional network
Filters of the fourth layer

Fig. 16.4 **a** Visualization of the first eight activations, filters of layers 1, 2, 3 and 4 in the VGG16 network trained with ImageNet. **b** Visualization of the filters in AlexNet (Krizhevsky et al. 2012)

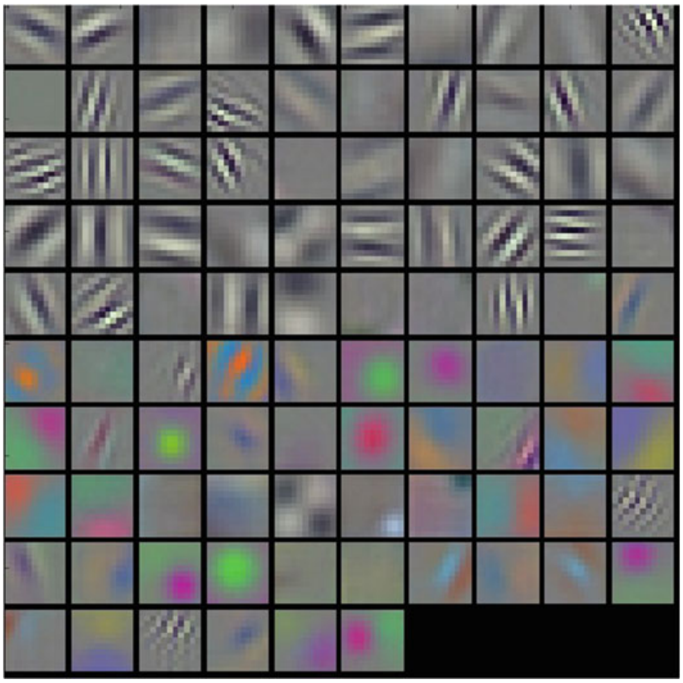
b


Fig. 16.4 (continued)

16.2.2 *Non-linear Rectifier Unit*

After performing convolution operation, an activation function is used to select and map information from the current layer. This is sometimes called the detector stage. Very often, we use a non-linear rectifier unit to induce non-linearity in the computation. This is driven by the effort to simulate the activity of neurons in human brain as we usually process information in a non-linear manner. Furthermore, it is also motivated by the belief that the data in the real world are mostly non-linear. Hence, it enables better training and fitting of deeper networks to achieve better results. We have listed a few commonly used activation functions as shown below.

16.2.2.1 Sigmoid or Logistic Function

A sigmoid function is a real continuous function that maps the input to the value between the range of zero and one. This property gives an ideal ground in predicting a probabilistic output since it satisfies the axiom of probability. Moreover, considering the output value between zero and one, it is sometimes used to access the weighted importance of each feature, by assigning a value to each component. To elaborate, a

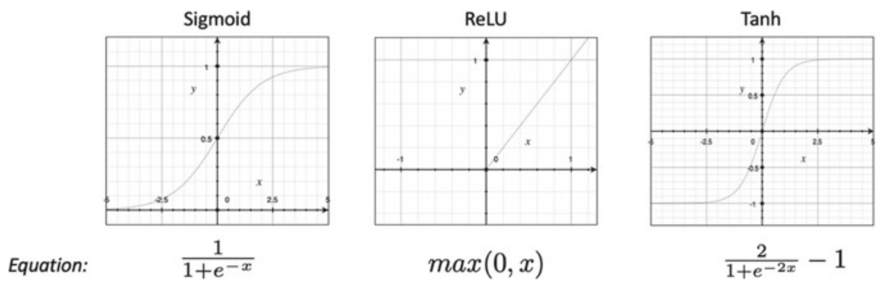


Fig. 16.5 Some commonly used activation functions in deep learning models

value of zero removes the feature component in the layer while a value of one keeps every information in the layer. The preserved information will be used for computing prediction in the subsequent event. This attribute is helpful when we work with data that are sequential in event i.e. RNN, LSTM model.

16.2.2.2 ReLU (Rectified Linear Unit)

ReLU is the most frequently used activation function in deep learning. It is reported to be the most robust in terms of model performance. As we can see in Fig. 16.5, ReLU function sets the output to zero for every input value that is negative or else, it returns the input value. However, a shortcoming with ReLU is that all negative values become zero immediately which may affect the capacity of the model to train the data properly.

16.2.2.3 Hyperbolic Tangent (TanH)

The last activation function that we have on the list is tanh. It is very similar to a sigmoid function with the range from negative one to positive one. Hence, we would usually use it for classification. This maps the input with strong prior in which a negative input will be strongly negative and zero inputs will be close to zero in the tanh graph.

Here, the execution of the activation function takes place element wise, where the individual element of each row and column from the feature map is passed into the function. The derived output has the same dimensionality as the input feature map.

16.2.3 Spatial Pooling

Typical block of a classifying CNN model that achieves state of the art would consist of three stages. First, a convolution operation finds acute patterns in the image.

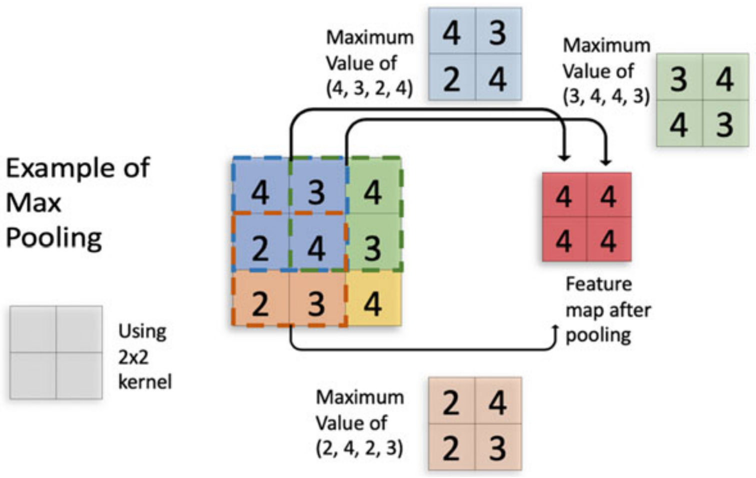

Example of Max Pooling
Using 2x2 kernel
Maximum Value of (4, 3, 2, 4)
4 3
2 4
Maximum Value of (3, 4, 4, 3)
3 4
4 3
4 3 4
2 4 3
2 3 4
4 4
4 4
Feature map after pooling
Maximum Value of (2, 4, 2, 3)
2 4
2 3

Fig. 16.6 Featuring steps to max pooling. Here, we use kernel size of 2 × 2 and stride of 1. i.e. we slide the kernel window by one pixel length for every pooling step. The output of this max pooling has a dimension of 2 × 2

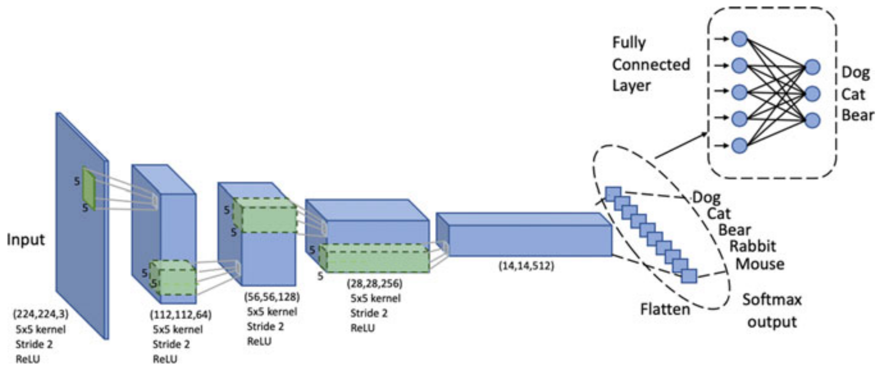
Then, the output features are handed over to an activation function in the second stage. At the last stage, we would implement a pooling function that trimmed the dimensionality (down sampling) of each feature map while keeping the most critical information. This would in turn reduce the number of parameters in the network and prevent overfitting of our model.

Spatial pooling comes in various forms and the most frequently used pooling operation is max pooling. To illustrate the process of max pooling, we use a kernel of a definite shape (i.e. size = 2 × 2) and then carry out pointwise operation to pull the maximum value of the location. A diagram is drawn in Fig. 16.6 to visualize the process.

One of the most important reasons of using pooling is to make the input feature invariant to small translations. This means that if we apply local translation to Fig. 16.2, max pooling helps to maintain most of the output value. Essentially, we are able to acquire asymptotically the same output for convoluting a cat that sits on top of a tree versus the same cat that sleeps under the tree. Hence, we conclude that pooling ignores the location of subjects and places more emphasis on the presence of the features, which will be the cat in this example.

16.2.4 Putting Things Together

Until now, we have covered the main operating structures found in most typical CNN model. The CNN block is usually constructed in the checklist as listed below:



Fully Connected Layer
Dog
Cat
Bear
Input
(224,224,3) 5x5 kernel Stride 2 ReLU
(112,112,64) 5x5 kernel Stride 2 ReLU
(56,56,128) 5x5 kernel Stride 2 ReLU
(28,28,256) 5x5 kernel Stride 2 ReLU
(14,14,512)
Dog Cat Bear Rabbit Mouse
Flatten
Softmax output

Fig. 16.7 Sample network architecture of a functional CNN model

1. *Convolution*
2. *Activation Function (ReLU)*
3. *Pooling (Sub-sampling)*
4. *Fully connected layer*

The last component of CNN is usually the fully connected layer (Fig. 16.7). This layer connects all the sophisticated features extracted at the end layer of the convolution with a vector of individual parameters specifying the interactions between each pixels of the feature maps. The weights for the parameter are learned to reduce inaccuracy in the prediction. This is similar to the concepts of a regression model as we fit the parameter weights with the least square solution to explain the target outcome. However, our predictors in this case are the flatten vector of the convolved map. Finally, we use a sigmoid function to generate the likelihood of the classes for the input image of a two classes problem or else, we will use a Softmax function in the case of multiclass.

16.2.5 Back-Propagation and Training with Gradient Descent

During backpropagation, we conduct supervised learning as we train our model with gradient descent algorithm to find the best fitted parameters that gives optimal prediction. Gradient descent is a first order iterative optimization method where we can find a local minimum that minimizes the loss function. Here, the loss function defines an evaluating metric that measures how far off the current model performs against the target in our dataset. This is also sometimes referred to as the error or the cost function. If we know the local minimum, our job is almost done and we conclude that the model is optimized at that region.

To understand the motivation behind gradient descent, suppose we are learning the parameters of a multiple linear regression, i.e. $y = X\beta + \epsilon$. The least square estimate of β is the minimizer of the square error $\mathbb{L}(B) = (Y - X\beta)'(Y - X\beta)$. The first and second order derivatives of $\mathbb{L}(\beta)$ with respect to β is given by

$$\frac{\partial \mathbb{L}}{\partial \beta} = -2X'(Y - XB),\ \frac{\partial^2 \mathbb{L}}{\partial \beta \partial \beta'} = 2X'X$$

Since $X'X$ is positive semi-definite and if we assume $X'X$ is of full rank, the least square solution of $\mathbb{L}(\beta)$ is given by

$$\widehat{\beta_{\text{Loss}}} = \left(X'X\right)^{-1}\left(X'Y\right)$$

Suppose now that $X'X$ is non full rank, i.e. $p \gg n$. This is often the case for an image dataset where the number of features is usually very large. We can’t simply inverse the matrix and it turns out that there is no unique solution in this case. However, we do know that $\mathbb{L}(\beta)$ is a strictly convex function and the local minimum is the point where error minimizes, i.e. least square solution. As such, we take another approach to solve this problem with the ‘descending stairs’ approach to find our solution.

This approach is an iterative process that begins with a random location, x_0, on the convex curve that is not the minimum. Our aim is to find the optimum $x*$ that gives the minimum loss, $\text{argmin}_x \mapsto F(x)$ by updating x_i in every ith iteration. We choose a descent direction such that the dot product of the gradient is negative, $\langle \nabla F(x); d \rangle < 0$, where $\nabla F(x) = \frac{1}{N}\sum_{i=1}^{N} \nabla_x L(x, y_i)$. This ensures that we are moving towards the minimum point where the gradient is less negative.

To show this, we refer to the identity of the dot product given

$$\cos(\theta) = \frac{a \cdot b}{|a||b|}$$

Suppose vector a, b is a unit vector and the identity is reduced to $\cos\theta = a.b$. We know that taking cosine of any angle larger than 90° is negative. Since gradient is pointing towards the ascent direction as shown in Fig. 16.8, we can find any descent directions of more than 90° and the dot product computed to be negative, i.e. $\cos\theta$ = negative.

$$\langle \nabla F(x); -\nabla F(x) \rangle = -|\nabla F(x)|^2 < 0$$

Hence a naive descent direction,

$$d = -\nabla F(x)$$

This guarantees a negative value which indicates a descent direction.

Then, the steps to compute the new x is given by

$$x_{n+1} = x_n + \eta_n d_n$$

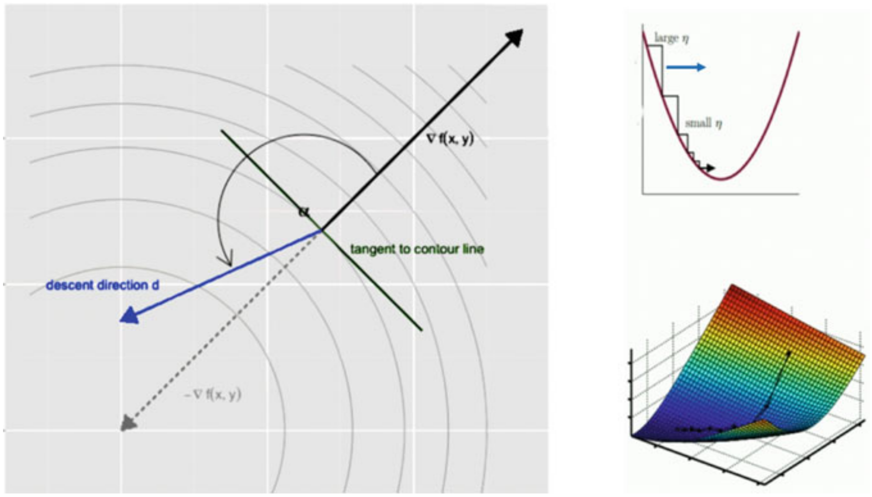


Fig. 16.8 Contour plot of a convex function on the left, where the gradient is less negative towards the origin of the axes. Cross sectional visualisation of a convex curve on the right

OR

$$x_{n+1} = x_n + \eta_n \nabla f(x_n)$$

where η_n is the learning rate.

The learning rate (or step-size) is a hyper-parameter that controls how much we are adjusting x_n position with respect to the descent direction. It can be thought of as how far should we move in the descending direction of the current loss gradient. Taking too small of a step would result in very slow convergence to the local minimum and too big of a step would overshoot the minimum or even cause divergence. Thus, we have to be careful in choosing a suitable learning rate for our model. Then after, we iterate through the algorithm as we let it computationally alter towards the optimum point. The solution is asymptotically close to the estimated βloss.

However, this is computationally expensive as we are aggregating losses for every observed data point. The complexity increases as the volume of the dataset increases. Hence, a more practical algorithm would sample a smaller subset from the original dataset and we would estimate the current gradient loss based on the smaller subset. The randomness in sampling smaller sample is known as stochastic gradient descent (SGD) and we can also prove that $\mathbb{E}[\nabla \hat{F}(x)] = \nabla F(x)$. In practice, the estimated loss converge to the actual loss if we sample this large enough of times by the law of large numbers.

Hence, we prefer that n ≪ N.

$$\nabla \widehat{F(x)} = \frac{1}{n} \sum_{k=1}^{n} \nabla_x L(x, y_{i\,k})$$

This results in updating x with

$$x_{n+1} = x_n - \eta_n \nabla f\left(\hat{x}_n\right)$$

Lastly, there are a few commonly used loss functions namely, cross entropy, Kullback Leibler Divergence, Mean Square Error (MSE), etc. The first two functions are used to train a generative model while MSE is used for a discriminative model. Since the performance of the prediction model improves with every updated parameters from the SGD, we expect the loss to decrease in all iterations. When the loss converges to a significantly small value, this indicates that we are ready to do some prediction.

16.2.6 Other Useful Convolution Layers

In this section, we discuss some innovation to the convolution layer to manage certain tasks more effectively.

16.2.6.1 Transposed Convolution

Transposed convolution works as an up sampling method. In some cases where we want to generate an image from lower resolution to higher resolution, we need a function that maps the input without any distortion to the information. This can be processed by some interpolation methods like nearest neighbour interpolation or bilinear interpolation. However, they are very much like a manual feature engineering and there is no learning taking place in the network. Hence, if we hope to design a network to optimize the up sampling, we can refer to a transposed convolution as it augments the dimension of our original matrix using learnable parameters.

As shown in Fig. 16.9, suppose we have a 3 $\times$ 3 matrix and we are interested to obtain a matrix with 5 $\times$ 5 resolution. We choose a transposed convolution with 3 $\times$ 3 kernel and stride of 2. Here, the stride is defined slightly different from the convolution operation. When stride of 2 is called upon, each pixel is bordered with a row and a column of zeros. Then, we slide a kernel of 3 $\times$ 3 down every pixels and carry out the usual pointwise multiplication. This will eventually result in a 5 $\times$ 5 matrix.

16.2.6.2 Dilated Convolution

Dilated convolution is an alternative to the conventional pooling method. It is usually used for down sampling tasks and we can generally see an improvement in performance like for an image segmentation problem. To illustrate this operation, the input is presented with the bottom matrix in Fig. 16.10 and the top shows the output of a

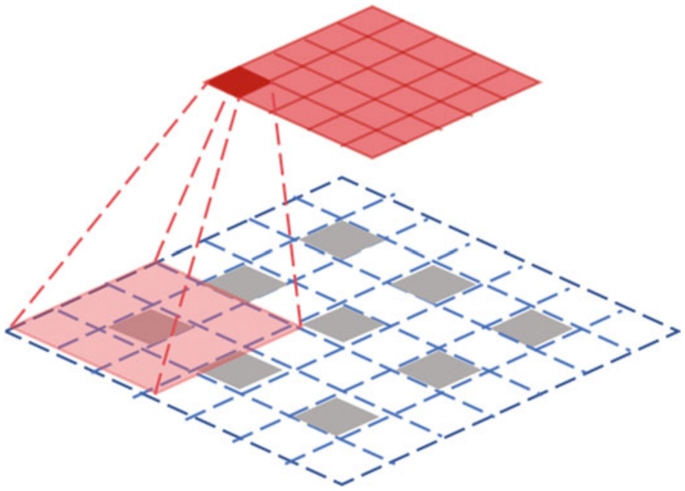


Fig. 16.9 Structure of a transposed convolution with 3 × 3 kernel and stride of 2. Input is a 3 × 3 matrix and output is a 5 × 5 matrix

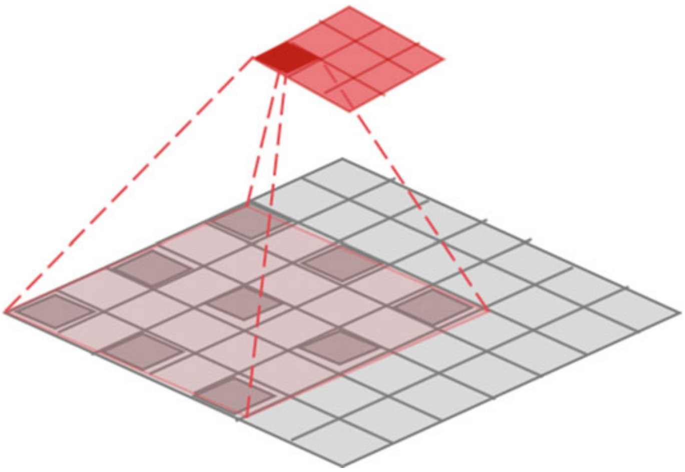


Fig. 16.10 Structure of a dilated convolution with 3 × 3 kernel and stride of 2. Input is a 7 × 7 matrix and output is a 3 × 3 matrix

dilated convolution. Similarly, when we set a 3 × 3 kernel and the stride to be two, it does not slide the kernel two pixels down for every matrix multiplication. Instead, a stride of two slots zeros around every pixel row wise and column wise of the kernel and the multiplication involves a 5 × 5 kernel matrix (larger receptive field with same computation and memory costs while preserving resolution). Then, pointwise matrix multiplication is done in every pixel interval and we can show that our final output is a 3 × 3 matrix. The main benefit of this is that dilated convolutions support exponential growth of the receptive field without loss of resolution or coverage.

16.3 Application of CNN on Skin Melanoma Segmentation

In this section, our aim is to build a semantic segmentation model to predict the primary lesion region of the melanoma skin. The model that we will be constructing is based on the 2018 ISIC challenge dataset and we mainly focus on task 1 of the image segmentation problem.

In this task, all lesion images comprise of exactly one primary lesion. We do not consider any of the other smaller secondary lesions or other pigmented regions as it lies beyond our interest for this tutorial. The image datasets are created with several techniques. However, all data are reviewed and curated by practicing dermatologists with expertise in dermoscopy. The distribution of the dataset follows closely to the real world setting where we get to observe more benign lesion as opposed to malignant cases. Furthermore, the response data is a binary mask image containing a single skin lesion class (primary) indicated by 255 and the background indicated by zero. Take note that the mask image must possess the exact same resolution as its corresponding lesion image. More details can be found from the challenge webpage.

The evaluating metric (loss function) used for this training is the threshold Jaccard index metric. The score is a piecewise function,

$$Score(index) = \begin{cases} 0, & index \leq 0.65 \\ index, & index > 0.65 \end{cases}$$

To kick start, you can first download the code to the tutorial from the textbook repository at: https://github.com/crticaldata/globalhealthdatabook.

Next, download the data from the official challenge page provided (https://challenge2018.isic-archive.com/task1/) and save it in a folder called data. Ensure that the name of the downloaded skin dataset is unchanged and correctly labelled or you may face run error in reproducing the code. It should be titled as
`'ISIC2018_Task1-2_Training_Input'`,
`'ISIC2018_Task1-2_Validation_Input'`,
`'ISIC2018_Task1-2_Test_Input'`
and `'ISIC2018_Task1_Training_GroundTruth'`.

Thereafter, place the `data` folder in `/U-net/Datasets/ISIC_2018/` and we are done with the setup. To try running this code on your own, we suggest that the readers open `segmentation.ipynb` and run the cells in jupyter notebook or alternatively, run `segmentation.py` in the terminal with

```
$python segmentation.py.
```

In this tutorial, we build our model with the following environment.

1. `python version 3.6.5`
2. `keras version 2.2.4`

3. `Tensorflow version 1.11.0`

The dependencies for this tutorial include

1. `tqdm version 4.29.1`
2. `skimage version 0.14.1`
3. `pandas version 0.23.4`
4. `numpy version 1.15.4`

Before we begin to build our CNN model, we import modules to be used in our code.

```
In [1]:
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
from models import *
from Datasets.ISIC2018 import *
import numpy as np
import os as os
import matplotlib.pyplot as plt
%matplotlib inline
```

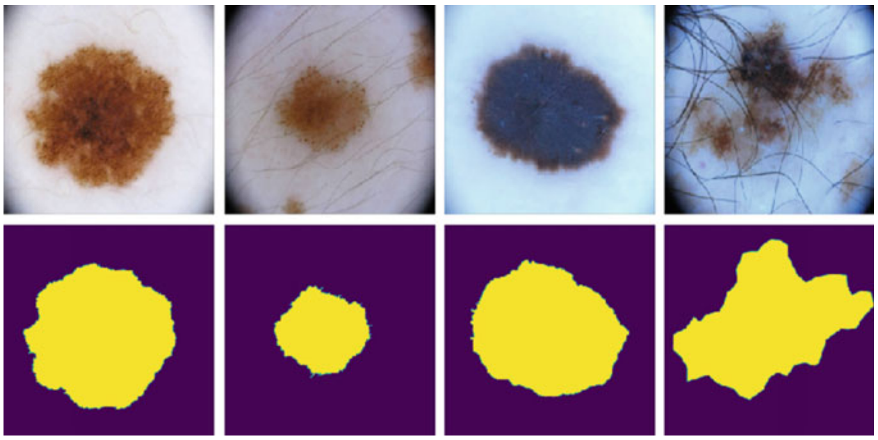
16.3.1 Loading Data

To load data in the environment, we run `load_training_data()` from the `models` module. This function reads the skin image from the `data` folder and performs image pre-processing to adopt the resolution of our input model. We set our model's input shape as 224 × 224 × 3 and this will resize all images to the same dimension. Next, the function will do a data split to form our training and validating set by choosing the ith partition from the k number of folds we defined.

```
In [2]:
(x_train, y_train), (x_valid, y_valid), _ = load_training_data(
                                                  output_size=224,
                                                  num_partitions=num_folds,
                                                  idx_partition=k_fold)
```

Here are some samples of the skin image as shown below. The bottom row shows our targets which are checked by the specialists of the segmented boundary of the skin lesion. The goal of this exercise is to come up with a model that learns the segmentation such that our model can come up with its own segmentation that performs close to the target.

```
In [124]:
fig, axs = plt.subplots(2,4, figsize=(12,7))
fig.subplots_adjust(hspace=0.1, wspace=0.05)
for i in range(4):
    axs[0,i].imshow(x_train[i + 50])
    axs[1,i].imshow(y_train[i+50])
    axs[0,i].axis('off')
    axs[1,i].axis('off')
```



In practice, we often carry out data augmentation as a pre-processing stage before we fit our model. This is because deep learning algorithms achieve better results with large datasets. Since deep learning networks have parameters in the order of millions, it would be ideal to have a proportional amount of examples. The bottom line is to have at least a few thousands of images before our model attains good performance. However, we are sometimes limited by the natural constraint that certain diseases are not commonly found in patients or we just simply do not have that many observations. Hence, we can try to augment our data artificially by flipping images, rotation or putting small translation to the image. The machine would treat it as if they were new distinct data points so that it would get enough realisations to tune its parameters during training. Here, we used keras function to do this.

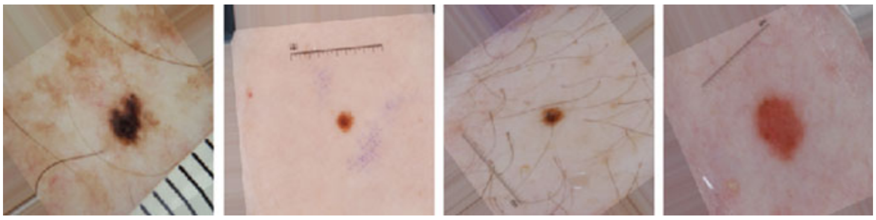
First, we define the type of alterations we planned to do on the existing image. Some suggestions would be listed as follows:

1. `horizontal flip = True`,
   random activating horizontal flip of image
2. `vertical flip = True`,
   random activating vertical flip of image
3. `rotation angle = 180`,
   random image rotation that covers up to 180°

4. `width_shift_range = 0.1,`
   random horizontal translation of image up to 0.1 unit
5. `height_shift_range = 0.1`
   random vertical translation of image up to 0.1 unit

We show some augmentations processed by the function as seen below

```
In [149]:
image_datagen = ImageDataGenerator(**data_gen_args)
image_generator = image_datagen.flow(x=x_train, seed= 609)
fig, axs = plt.subplots(1,4, figsize=(12,7))
fig.subplots_adjust(hspace=0.1, wspace=0.05)
for i in range(4):
    axs[i].imshow(np.array(image_generator[1][i+5]).astype(np.uint8))
    axs[i].axis('off')
```



16.3.2 *Our segmentation Model*

We introduce a semantic segmentation model called U-net in this tutorial. The model owes its name to the symmetric shape of its architecture. It can be largely divided into two parts, the encoder and decoder part. The encoder part is the typical CNN structure that we often see in most classification models which extract more abstract features from an input image by passing through a serious of convolutions, nonlinearities and poolings. The output of the encoder is a feature map which is smaller in spatial dimension but richer in abstract features. You can see from the illustration in Fig. 16.11 below that after passing through an encounter input image which was $572 \times 572 \times 1$ in size, it has been encoded to a feature map of a size $30 \times 30 \times$ channel size. The next task is to decode this encoded feature back to the segmentation image which we want to predict. Decoder is similar to encoder in a sense that they both have a series of convolutions and nonlinearities. However, the interpolation layer is used instead of the pooling layer to up-sample the encoded feature back to the dimension that is identical to the outcome segmentation label. There are many possible candidates for the interpolation layer. One possible way is to simply project features from each pixel to 2×2 with bilinear interpolation. Another way is to use

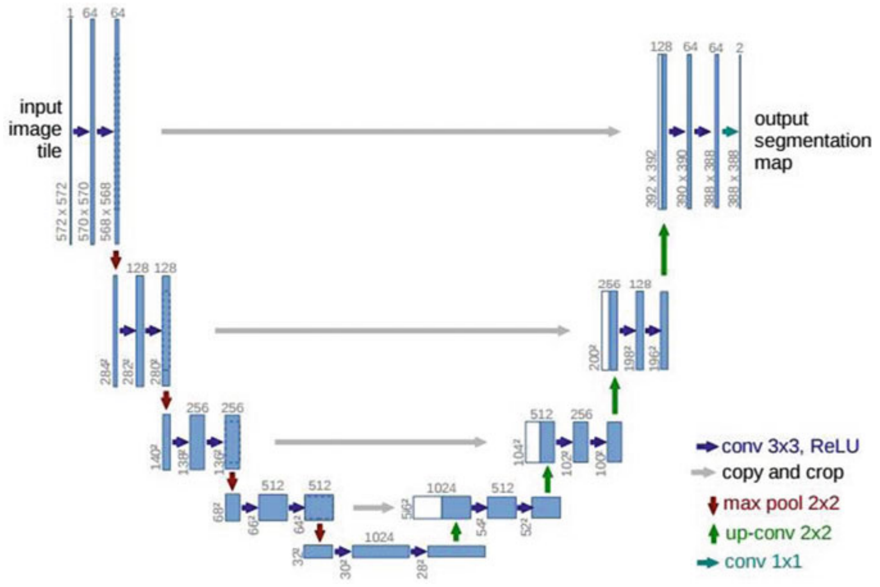


Fig. 16.11 Architecture of U-Net (Example for 32 × 32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of that box. The x-y size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations

transposed convolution with learnable parameters as we have discussed previously so that the networks learn what is the best way to up-sample and make a better prediction. Then, we implement skip connections to provide local information to the global information while up sampling. This combines the location information from the down sampling path with the contextual information in the up sampling path to finally obtain a general information combining localisation and context, which is necessary to predict a good segmentation map.

To build this model, we have written the framework in the model script that takes in the parameter of the loss function, learning rate, evaluating metrics and the number of classes. We train our model for 10 epochs and the results are shown as follows.

```
In [6]:
model = unet(loss='crossentropy', lr=1e-4 ,metrics= metrics, num_classes=1)
model.fit_generator(generator= train_generator,
                    steps_per_epoch= steps_per_epoch,
                    epochs = 10,
                    initial_epoch = initial_epoch,
                    verbose= 1,
                    validation_data= (x_valid, y_valid),
                    workers = 16,
                    use_multiprocessing= False)
```

```
Epoch 1/10
129/129 [==============================] - 73s 565ms/step - loss: 0.4866 -
binary_jaccard_index: 0.5171 - binary_pixelwise_sensitivity: 0.7696 - binar
y_pixelwise_specificity: 0.6413 - val_loss: 0.4977 - val_binary_jaccard_ind
ex: 0.5825 - val_binary_pixelwise_sensitivity: 0.8912 - val_binary_pixelwis
e_specificity: 0.6560
Epoch 2/10
129/129 [==============================] - 66s 512ms/step - loss: 0.3857 -
binary_jaccard_index: 0.6362 - binary_pixelwise_sensitivity: 0.8501 - binar
y_pixelwise_specificity: 0.6859 - val_loss: 0.3636 - val_binary_jaccard_ind
ex: 0.6869 - val_binary_pixelwise_sensitivity: 0.8978 - val_binary_pixelwis
e_specificity: 0.6951
Epoch 3/10
129/129 [==============================] - 65s 501ms/step - loss: 0.3645 -
binary_jaccard_index: 0.6623 - binary_pixelwise_sensitivity: 0.8565 - binar
y_pixelwise_specificity: 0.7006 - val_loss: 0.3304 - val_binary_jaccard_ind
ex: 0.7106 - val_binary_pixelwise_sensitivity: 0.8908 - val_binary_pixelwis
e_specificity: 0.7167
Epoch 4/10
129/129 [==============================] - 65s 505ms/step - loss: 0.3456 -
binary_jaccard_index: 0.6886 - binary_pixelwise_sensitivity: 0.8675 - binar
y_pixelwise_specificity: 0.7134 - val_loss: 0.3944 - val_binary_jaccard_ind
ex: 0.6222 - val_binary_pixelwise_sensitivity: 0.8472 - val_binary_pixelwis
e_specificity: 0.6898
Epoch 5/10
129/129 [==============================] - 65s 501ms/step - loss: 0.3271 -
binary_jaccard_index: 0.7047 - binary_pixelwise_sensitivity: 0.8727 - binar
y_pixelwise_specificity: 0.7258 - val_loss: 0.2942 - val_binary_jaccard_ind
ex: 0.7452 - val_binary_pixelwise_sensitivity: 0.8581 - val_binary_pixelwis
e_specificity: 0.7541
Epoch 6/10
129/129 [==============================] - 65s 504ms/step - loss: 0.3135 -
binary_jaccard_index: 0.7199 - binary_pixelwise_sensitivity: 0.8737 - binar
y_pixelwise_specificity: 0.7384 - val_loss: 0.2806 - val_binary_jaccard_ind
ex: 0.7554 - val_binary_pixelwise_sensitivity: 0.8523 - val_binary_pixelwis
e_specificity: 0.7663
Epoch 7/10
129/129 [==============================] - 65s 503ms/step - loss: 0.2994 -
binary_jaccard_index: 0.7319 - binary_pixelwise_sensitivity: 0.8793 - binar
y_pixelwise_specificity: 0.7486 - val_loss: 0.2848 - val_binary_jaccard_ind
ex: 0.7338 - val_binary_pixelwise_sensitivity: 0.8638 - val_binary_pixelwis
e_specificity: 0.7614
Epoch 8/10
129/129 [==============================] - 65s 505ms/step - loss: 0.2923 -
binary_jaccard_index: 0.7347 - binary_pixelwise_sensitivity: 0.8808 - binar
y_pixelwise_specificity: 0.7573 - val_loss: 0.2783 - val_binary_jaccard_ind
ex: 0.7337 - val_binary_pixelwise_sensitivity: 0.8381 - val_binary_pixelwis
e_specificity: 0.7733
```

```
Epoch 9/10
129/129 [==============================] - 65s 503ms/step - loss: 0.2814 -
binary_jaccard_index: 0.7415 - binary_pixelwise_sensitivity: 0.8823 - binar
y_pixelwise_specificity: 0.7673 - val_loss: 0.2508 - val_binary_jaccard_ind
ex: 0.7796 - val_binary_pixelwise_sensitivity: 0.8502 - val_binary_pixelwis
e_specificity: 0.7934
Epoch 10/10
129/129 [==============================] - 65s 505ms/step - loss: 0.2651 -
binary_jaccard_index: 0.7569 - binary_pixelwise_sensitivity: 0.8805 - binar
y_pixelwise_specificity: 0.7781 - val_loss: 0.2525 - val_binary_jaccard_ind
ex: 0.7280 - val_binary_pixelwise_sensitivity: 0.8506 - val_binary_pixelwis
e_specificity: 0.7886

Out[6]:
<keras.callbacks.History at 0x7f55bc5d64e0>
```

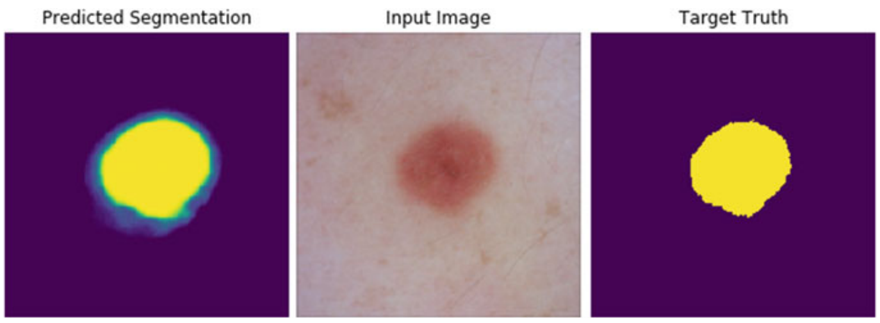
16.3.3 Making Prediction

To make the prediction of a new image, we call the predict function and send the original image to the function. We have printed an example of segmenting image below.

```
In [7]:
predict_img = model.predict(np.expand_dims(x_valid[20],axis=0))
predict_img.shape
Out[7]:
(1, 224, 224, 1)
```

```
In [17]:
fig, axs = plt.subplots(1,3, figsize=(12,7))
axs[0].imshow(np.squeeze(predict_img))
axs[0].axis('off')
axs[0].set_title('Predicted Segmentation')
axs[1].imshow(np.squeeze(x_valid[20]))
axs[1].axis('off')
axs[1].set_title('Input Image')
axs[2].imshow(np.squeeze(y_valid[20]))
axs[2].axis('off')
axs[2].set_title('Target Truth')

Out[17]:
Text(0.5, 1.0, 'Target Truth')
```



On the left, we see that our model has performed well as compared to the target truth on the right. It has achieved Jaccard index of more than 0.7 in the validating set and attained a score of above 0.75 for both pixel-wise sensitivity and specificity. We conclude that the model has learned well in this segmenting task.

References

Akkus, Z., Galimzianova, A., Hoogi, A., et al. (2017). *Journal of Digital Imaging, 30,* 449. https://doi.org/10.1007/s10278-017-9983-4.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In: *Advances in Neural Information Processing Systems* (pp. 1097–1105).

Rajpurkar, P., Irvin, J., Zhu, K., Yang, B., Mehta, H., Duan, T., et al. (2017). Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. http://arxiv.org/abs/1711.05225.