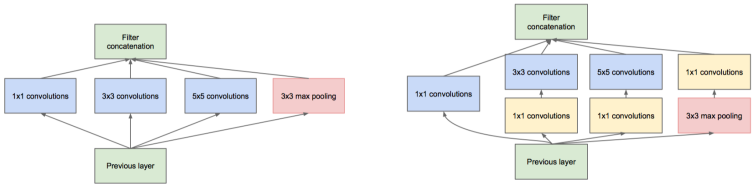
0.1 Inception Network

The Inception network is an important milestone of CNN classifiers. Before Inception network, most popular CNN networks just stack convolution layers deeper and deeper, hoping to get better performance. However, deeper network has more parameters which make gradient descent less effective and lead to overfitting. The Inception network is carefully designed in terms of speed and accuracy while keeping the computational budget constant. The network is organized in the form of «Inception module» which makes it possible for the network to grow both in width and depth. The performance is verified by GoogleLeNet, a 22 layers deep network which won ILSVRC14 competition.

The main difference between Inception module and normal CNN convolution layer is that Inception uses various sizes of filters in each layer while CNN uses only one. The idea of Inception architecture is based on finding out how an optimal local sparse structure in a convolutional vision network can be approximated. One straight forward way is to use more than one filter size and let the training phase to decide the best approximation area. The outputs of each filter are stacked together to form the input of the next stage. As we can see in the figure, there are three shapes of filter: 1x1, 3x3 and 5x5. The 1x1 filter is also used for dimension reduction. In figure a, we can see the naive implementation of this idea. This implementation, however, has one big issue, even a modest number of 5x5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters. This problem becomes even more pronounced when we stack all the outputs together. This leads to the second implementation structure, as we can see in figure b. Whenever the computational requirements increase too much, we can simply apply a 1x1 convolution to reduce the dimension. Another thing to mention is the max pooling layer which exists only due to historical reason. Back in time, good performance CNN architectures have pooling structures and the inventor of Inception module decided to add the max pooling layer into the structure. Nowadays people start to argue about pooling layers, on one hand, pooling layers reduce the dimension of tensors to make deep network possible, but on the other hand, pooling layers miss pixel information which can hurt the performance. In Count-ception architecture the author does not use pooling layers.

Inception module is the fundamental element in inception architectures. By concatenating various sizes of filters in each layer and using 1x1 filters to reduce dimensions, the network can grow wider and deeper. Since Inception module is invented, several improvements are made over the years. However in this thesis, we only use the idea of stacking various sizes of filters and ignore other tricks. So the details of the improvements of Inception networks are not discussed here, only a summary is provided.



(a) Inception module, naive version (b) Inception module with dimension reductions

Figure 1: Inception module

- Inception v1 concatenates Inception modules to make the network wider and deeper.
- Inception v2 uses two 3x3 filters to replace 5x5 filters, decomposes nxn filers into 1xn and nx1 filters in order to increase computation speed.
- Inception v3 introduces RMSprop, factorized 7x7 filters and batch normalization.
- Inception v4 uses the idea of ResNet.

0.2 Fully Convolutional Network

F-CNN is short for fully convolutional neural net which is a convolutional network without fully connected layers. The whole network is built by convolution layers and pooling layers. Normal CNN architectures can be seen as a pipeline structure: first using several convolution and pooling layers to extract features from images, then using fully connected layers to exploit these features for classification. This structure has one disadvantage, once we set up the architecture we can not change the image size anymore, otherwise we can not forward the tensor into fully connected layers.

In fact, passing tensors through fully connected layers can be seen as a convolution operation. We can convert any fully connected layers into convolution layers, with one-to-one map on the weights. Let's see an example, suppose we have a CNN network doing three class classification with one hidden layer of size 128 neurons. After convolution and pooling operations we get a tensor with size 2x2x256. If we want to pass it through fully connected layers, we need to first stretch it into a long vector of size 1024. If we ignore the bias, the weight matrices in fully connected layers are 1024x128 and 128x3. We can convert fully connected layers into convolution layers using the following steps:

1. Do not stretch the 2x2x256 tensor, keep the dimension.

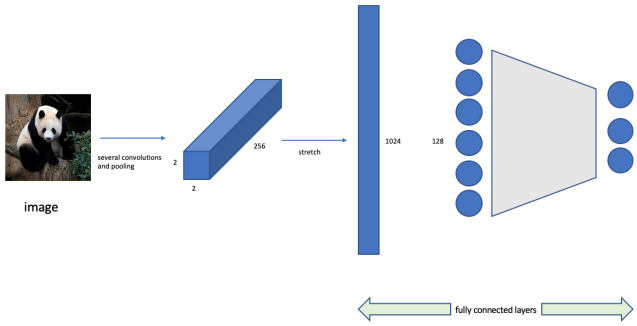


Figure 2: Normal CNN architecure

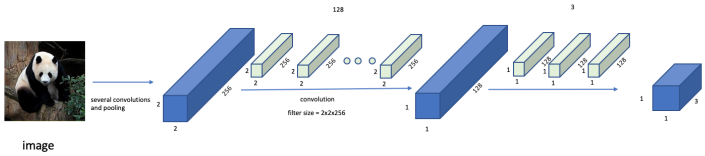


Figure 3: Fully Convolutional Network

2. Build 128 filters, each filter is 2x2x256. Passing 1024 length vector through fully connected layers can be seen as doing convolution with no padding and stride one.

3. Build 3 filters, each filter is 1x1x128.

If we do the math, there is a one-to-one map between weights in fully connected layers and weights in convolution filters, as we can see in the figure. In general, converting any fully connected layers into convolution operations has the following rules:

- Passing vector through fully connected layers is equivalent to doing convolution with no padding and stride one.

- The filter size is equal to the size of input tensor, and the filter number is equal to the number of neurons in the next fully connected layer.

Converting fully connected layers into convolution layers has several benefits. First, we do not need to reshape the image when we have different image size. As long as the input image size is no smaller than the filter size, we can directly forward it through the network. Second, we do not get a single vector at the output, instead we get a tensor. This means if we are doing

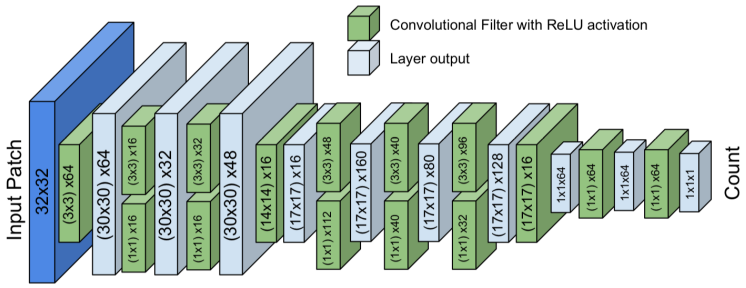


Figure 4: Count-ception Architecture

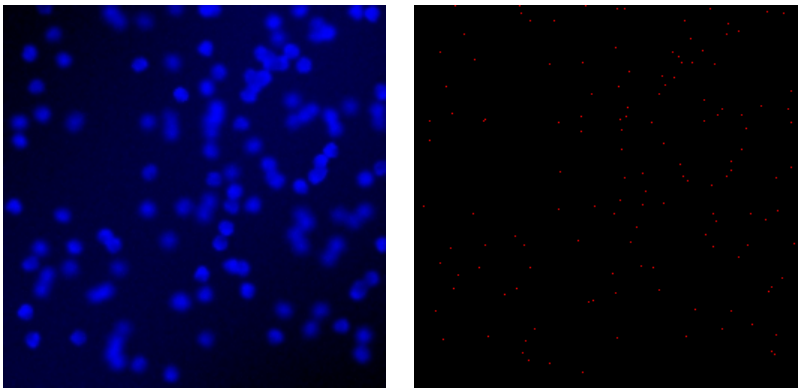
image classification and we feed the network with a large size image, we will not get a single probability vector but a heat map. Thus we can exploit the heat map for further processing.

0.3 Count-ception Architecture

Count-ception network is introduced in paper [**?**]. The author uses inception modules to build a network targeting at counting objects in the image. The whole network is a fully convolutional neural net and no pooling layer is used. The author didn't use pooling layers in order to not lose pixel information and make calculating receptive field easier. The network is shown in figure 3.4, each 32x32 patch produces a 1x1x1 tensor indicating the number of objects in this patch. After each convolution, batch normalization and reLU activation are used in order to speed up convergence.

The network is a fully convolutional neural net and we can input image of different sizes. Each 32x32 patch generates a single output value and if we feed the network with 320x320 image, the output tensor will be 289x289x1. In order to train this network we need to construct our training dataset. The network is targeting at producing object counts in the image, and each object has a dot label at center, as we can see in figure 3.3. Figure a is the original image and we need to count the number of cells in it. Figure b is the labeled image where each dot corresponds to the center point of the cell.

The prediction map is calculated using the original image and Count-ception network. In order to calculate the loss, we need to construct the target manually. The target construction network takes the labeled image and produces a heat map. If an output value in the heat map is not zero, it means the receptive field of it contains an object. The target construction network is a simple one layer CNN network containing only a convolution operation of 0 padding and stride 1. The filter is filled with all ones and its size is 32x32, same as the Count-ception architecture's receptive field.



(a) Original image (b) Labeled image

Figure 5: Count-ception Dataset

The whole procedure for Count-ception is listed below:

1. Pad the image image in order to deal with objects near the boarder.
2. Calculate the prediction map using Count-ception architecture.
3. Calculate the target map using target construction network.
4. Calculate the loss between prediction map and target map. The loss function is L1 loss.
5. Use gradient descent to update the weights. Note that the target construction network's weights are fixed. We only need to update the weights for Count-ception architecture.

After we have trained the Count-ception network, we can calculate the prediction map for each input image. In order to get the number of objects in the image, we can simply sum all the pixel values in the prediction map. Note that due to convolution operation, each pixel in the input image is counted redundantly. We can adjust the object counts using formula (1). F(I) stands for the output of input image, and r is the receptive field size. In the example above, the receptive field of one pixel in the output is 32x32, so r equals 32. Each pixel in the input image is counted 32x32 times.

$$\#counts = \frac{\sum_{x,y} F(I)}{r^2} \tag{1}$$