Convolutional Neural Networks

Notes by José A. Espiño P. [1]

Summer Semester 2022–2023

[1]The content in these notes is sourced from what was covered in the MOOG the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Foundations of Convolutional Neural Networks

Computer Vision is one of the areas where deep learning has had the most impact. It is the art of giving machines the ability to see. It is a very active area of research and has many applications. One of the challenges of computer vision is that the input is a 3D array of pixels. This makes it hard to use a standard neural network architecture, especially considering that the input size can get big. For example, a $1000 \times 1000 \times 3$ pixel image would have 3 million input features. This would make the network too big and computationally expensive. Also, it would be hard to train because of the large number of parameters.
To adapt to this issue, we need to improve on the **convolution** operation, which is the core of convolutional neural networks. This operation consists in doing an element-wise multiplication of two matrices and then summing the results. The convolution of two matrices f and g is denoted by $f * g$ and is defined as follows:

$$(f * g)(i, j) = \sum_m \sum_n f(m, n) g(i - m, j - n)$$

In the context of computer vision, we can think of f as the input image and g as the **filter** or **kernel**, which is a small matrix that we apply to the input image. The output of the convolution is called the **feature map**. The filter is usually a small matrix, for example, a 3×3 matrix. Since the filter is usually smaller than the input image, the feature map will be smaller than the input image.
This operation can be visualised as follows:

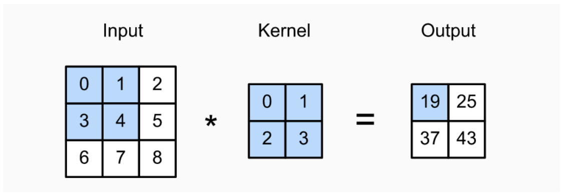


Figure 1: Convolution operation, sourced from this post.

Depending on the kernel values, we will be able to detect different features in the image. The core idea of convolutional neural networks is to learn the values of the kernel—as opposed to manually researching which kernels are most effective for which features. This is done by training the network on a large number of images.
The convolution operation is also used in other areas, such as natural language processing. In this case, the input is a sequence of words, and the filter is a sequence of words that we want to detect. For example, we could have a filter that detects the word “cat” in a sentence.
Padding is a basic modification to the convolution operation that allows us to control the size of the output. It consists in adding zeros around the input image, so that the input gets larger—and thus the output gets larger as well.
Keeping the size of the output the same as the non–padded size of the input allows us to stack multiple convolutional layers on top of each other. Without padding, our image would get progressively smaller as we pass it through more convolutional layers. Two common padding settings are **same** and **valid**. In the same setting, we add enough padding so that the output has the same size as the input. In the valid setting, we add no padding, so the output is forcibly smaller than the input.

Another part of the convolutional neural network puzzle is the **stride**. This is the number of pixels that we shift the filter each time we apply it to the input image. The default stride is 1, but we can increase it to make the output smaller.
Now, given an $n \times n$ input image, an $f \times f$ filter, a stride of s and a padding of p, the output size is given by the following formula:

$$\frac{n+2p-f}{s}+1 \times \frac{n+2p-f}{s}+1$$

Most images are three–dimensional instead of two–dimensional. This means that they have a depth dimension, which is the number of channels. For example, a color image has three channels: red, green and blue. The idea of a convolution can also be extended to this representation of an image. In this case, the filter is also three–dimensional, and it will be applied on each channel of the input image. The result of each convolution will be summed to produce a single value for each pixel in the output.

Usually, when analysing a picture, we will convolute the input with several kernels, and then add a bias term (a real number) to each element in the feature map. Once this is done, we normally pass the result of this addition into an activation function (e.g. ReLU) to get the final output of the layer.

All of this covered, let us see how to implement one layer of a convolutional network. Let l be a convolutional layer, $f^{[l]}$ be the filter size, $p^{[l]}$ the padding, and $s^{[l]}$ the stride. Let $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$ be the dimensions of the input, and $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$ be the dimensions of the output. The way we calculate the dimensions of the output is as follows:

$$
\begin{aligned}
n_H^{[l]} &= \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \\
n_W^{[l]} &= \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \\
n_C^{[l]} &= n_C^{[l-1]}
\end{aligned}
$$

Then, the activations will be given by:

$$A^{[l]} -> m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$$

Furthermore, the weights and biases will be defined as:

$$
\begin{aligned}
&W^{[l]} -> f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]} \\
&b^{[l]} -> n_C^{[l]}
\end{aligned}
$$

Now take a look at this sample implementation of a simple ConvNet:

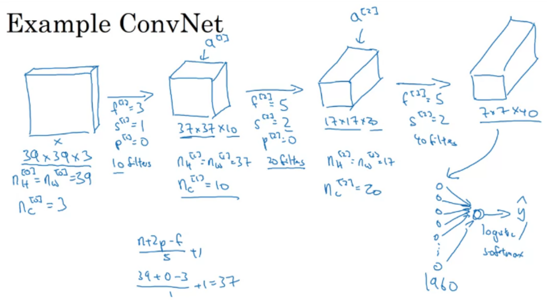


Figure 2: Sample implementation of a simple ConvNet, sourced from this lecture.

Other than convolutional layers, ConvNets also use **pooling layers**. These layers reduce the size of the input by applying a function to a small window of the input. The most common pooling function is the **max pooling** function, which takes the maximum value in the window. Check out this example of max pooling:

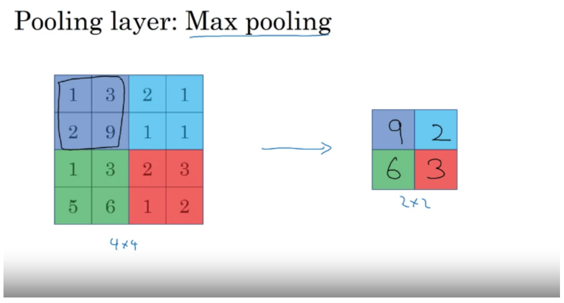


Figure 3: Max pooling, sourced from this lecture.

Max pooling has two main parameters: the stride (s) and the filter size (f). These define how much the image will be reduced. The idea of max pooling is to reduce the size of the input, and thus the number of parameters and computations in the network. This helps us to avoid overfitting. In fact, a benefit of max pooling is that, in spite of it having two parameters, these are set—there is nothing for it to learn. Similar to convolutional layers, pooling layers can be three–dimensional.
Another type of pooling is **average pooling**, which takes the average of the values in the window instead of the maximum. This is not used as often as max pooling, but it can be useful in some cases, such as when we want to get a more smooth output or deep inside the network where we want to collapse the representation of the input.
Fully connected layers are also used in ConvNets. These are the same as the ones we have seen in deep neural networks. They are used to connect the convolutional layers to the output layer.
Implementing pooling and fully connected layers, a typical neural network looks like this:

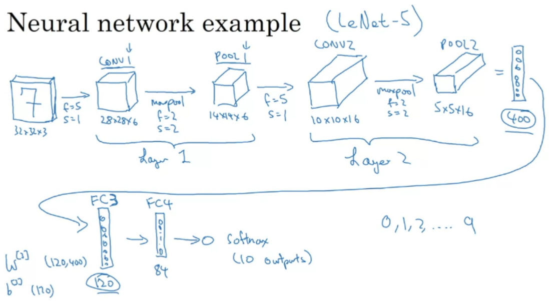
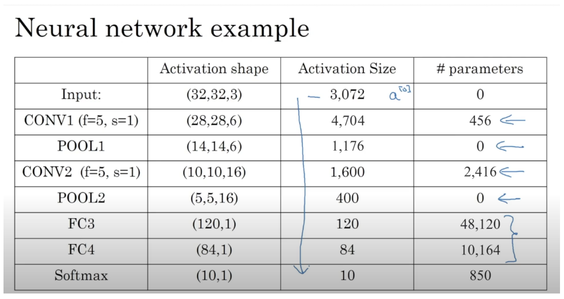


Figure 4: Typical ConvNet, sourced from this lecture.

This table presents the shape and size of the activation at each layer:



Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,6)	4,704	456
POOL1	(14,14,6)	1,176	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	2,416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,120
FC4	(84,1)	84	10,164
Softmax	(10,1)	10	850

Figure 5: Shape and size of the activation at each layer, sourced from this lecture.

The two main reasons why convolutions are desirable is the fact parameters are shared and the sparsity of their connections. This means that the number of parameters is reduced, and that the network is more robust to changes in the input.
When it comes to training, it is similar to what we have seen for other types of neural networks: we calculate the cost and try to optimise the values of the weights so as to minimise

the cost function. Further in this document, we will cover some ways to achieve this in practise.

2 Deep Convolutional Models: Case Studies

First, we will study some of the classic neural network architectures:

1. LeNet-5:
   This is one of the first ConvNets, and it was used to read zip codes, digits, etc. It has two convolutional layers, two average pooling layers, and three fully connected layers. It is not used as much today, but it is still a good example of a ConvNet. The following image illustrates this:

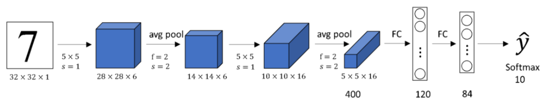
   

   Figure 6: LeNet-5, sourced from this website.

   The important thing to notice in this architecture is that as you go deeper in the network, the number of channels increases while the size of the image decreases.
   Another feature of this architecture is the arrangement of convolutional layer(s), followed by pooling layers, followed by convolutional layer(s), and so on until we reach the fully connected layers. This is a common pattern in ConvNets.

2. AlexNet:
   This is one of the most influential ConvNets. It was used to win the ImageNet competition in 2012. It has eight layers: five convolutional layers and three fully connected layers. It also uses ReLU as the activation function. The following image illustrates this:

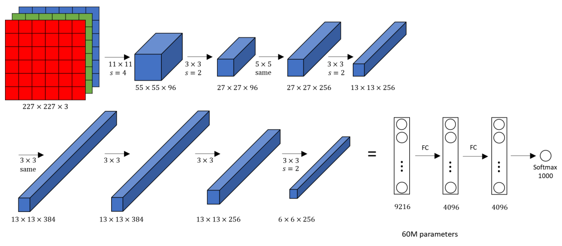
   

   Figure 7: AlexNet, sourced from this website.

   This network had a lot of similarities to LeNet-5, but it was much larger and deeper. It also used ReLU instead of sigmoid as the activation function. You are encouraged to read the original paper, which is available here.

3. VGG-16:
   This is another influential ConvNet. It was used to win the ImageNet competition in 2014. It has 16 layers: 13 convolutional layers and three fully connected layers. It also uses ReLU as the activation function. The following image illustrates this:

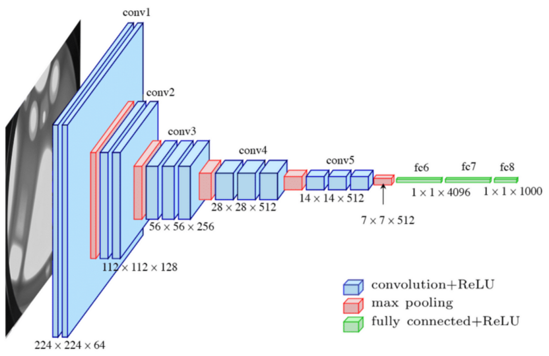


Figure 8: VGG-16, sourced from this website.

   The 16 in the name of the network refers to the number of layers. As you can see in the diagram, it is a pretty uniform network, where the number of filters keeps doubling as we go deeper in the network.

Looking beyond these classic networks, let us take a look at two of the most influential modern network architectures: **ResNets**:
ResNets are built out of **residual blocks**, which are blocks that have a skip connection. This means that, instead of passing the activation of a layer through the entire path leading to a layer far away, we pass it directly to that layer, linearly adding it to the activation of the layer that is far away before applying the activation function. In a ResNet, it is common to see several residual blocks stacked together, so that information travels deeper into the network.
The reason why ResNets are useful is because they allow us to train very deep networks. In fact, the authors of the original paper trained a 152-layer network. This is possible because the skip connections allow the network to learn identity functions, which are easier to learn than the functions that a network without skip connections would have to learn.
In the cases where the activation of a layer is not the same size as the activation of the layer that is far away, we can use a convolutional layer to change the size of the activation. This is called a **projection shortcut**.

An important idea when designing ConvNet architectures is **one–by–one convolutions**. These are convolutions with a kernel size of 1×1. They are useful because they allow us to change the number of channels, and thus control the number of parameters in the net — the number of channels in the output is equal to the number of filters in the convolution.

Inception Network:
Inception networks are built out of **inception blocks**, which are blocks that have several par-

allel paths. Each path has a different type of convolutional layer. The output of each path is concatenated together to form the output of the inception block. In a typical inception network, it is common to see several inception blocks stacked together, so that information travels deeper into the network. The advantage of this is that, instead of having to choose the kernel size for each layer, you apply several in parallel and let the network learn which ones it needs and in what proportion.
When it comes to kernels with a big size, and thus a big amount of multiplications performed, the computational cost can be very high. To solve this problem, we can use a 1×1 convolution to reduce the number of channels before applying the bigger convolution. This is called a **bottleneck layer**. Inception networks are organised in modules, where each module has several inception blocks. The following image illustrates this:

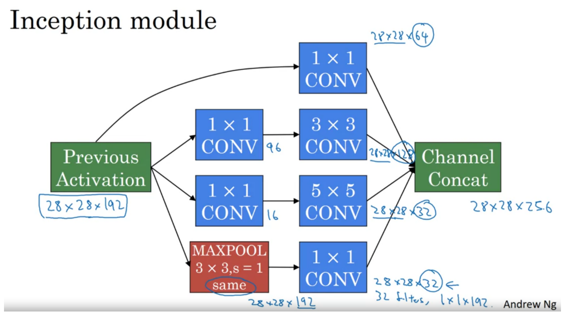


Figure 9: Inception network, sourced from this lecture.

An entire network can look like this:

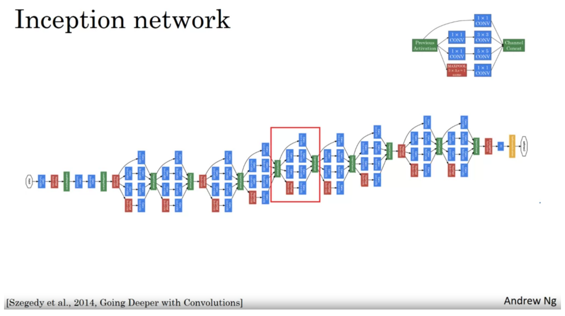


Figure 10: Inception network, sourced from this lecture.

Another foundational convnet architecture used in computer vision is **MobileNet** — a less computationally–expensive alternative to the architectures presented above. In regular convolutions, the computational cost can be defined as number of filer parameters × number of filter positions × number of filters. In contrast to normal convolutions, **depthwise separable convolutions** have two steps:

1. A depthwise convolution, which applies a single filter to each input channel. The computational cost of this is number of filter parameters × number of filter positions × number of filters

2. A pointwise convolution, which applies a 1×1 convolution to combine the output of the previous step. The computational cost of this is $\text{number of filter parameters} \times \text{number of filter positions} \times \text{number of filters}$

This is more economic than regular convolutions, since the number of filter positions is much smaller than the number of input channels. The idea of MobileNet is to use depthwise separable convolutions instead of regular convolutions. This reduces the computational cost of the network, at the expense of a slight decrease in accuracy.
Another advancemenet in MobileNets is **MobileNetv2**. This is a variation of MobileNet that uses a different type of bottleneck, containing a residual connection and an expansion step that increases the number of channels before applying the depthwise convolution. This can be seen in the following image:

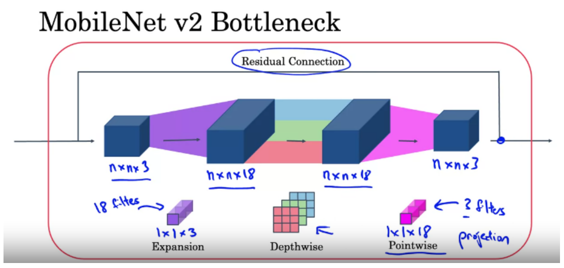


Figure 11: MobileNetv2, sourced from this lecture.

The expansion generates a larger image within the bottleneck, which in turn allows for the learning of a richer function. Since on mobile devices we have a limited amount of memory, the pointwise convolution is used to reduce the number of channels back to the original number—thus freeing up the memory for use.

2.1 Practical Advice

Sometimes, it is hard to replicate the results of research papers. This is because the authors of the papers do not always provide all the details of the implementation. Luckily, open–source implementations of the most popular architectures are available online. Platforms such as *Github* are a great place to find these implementations.
Usually, we download pre–trained models and use them as a starting point (*pre–training*)for our own models. This is called **transfer learning**. We achieve this by downloading the weights of a model that already exist, eliminate the last layer of the model, *freeze* the model so we do not change the weights, and then add our own layers on top of the model. This technique is especially helpful when we do not have a lot of data.
When we have a lot of training data available, we can opt to just freeze portions of the model, and train the rest. This is called **fine–tuning**.
If you have a lot of data, it is also possible to train a model from scratch. This is optly called **training from scratch**.

A technique that is often used to obtain more data is called **data augmentation**. There are several techniques to achieve this:

- Mirroring
- Random cropping
- Rotation
- Shearing
- Local warping
- Color shifting
- PCA color augmentation

3 Object Detection

The first important concept we need to understand is **object localization**, the task of finding the location of an object in an image. This is usually done by drawing a bounding box around the object. The difference between localization and detection is that the former only finds the location of one object, while the latter finds the location of multiple objects.
A way localization is achieved is by passing the image through a CNN and then using a fully–connected layer to predict the bounding box. This layer will output four numbers: b_x, b_y, b_h, b_w, which are used to define the bounding box, and a class label. In short, the output of the fully–connected layer is a vector of the form:

$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

Where p_c is the probability that there is an object in the image, b_x, b_y, b_h, b_w are the coordinates of the bounding box, and $c_1, c_2, \ldots, c_n$ are the probabilities that the object belongs to each of the n classes.
The loss function will be determined by the value of p_c:

- If $p_c = 1$, then the loss function will be the sum of the squared differences between the predicted and actual values of b_x, b_y, b_h, b_w and the class label.
- If $p_c = 0$, then the loss function will be the squared difference between the predicted and actual value of p_c.

The loss function is then summed over all the training examples. In this explanation, we use the squared difference as the loss function, but in practice, we use the **log–loss** function for classification (the c terms) and the **mean–squared error** for regression (the b terms).
We can also have a neural network output the x and y coordinates of important points in the image, **landmarks**. This is useful for tasks such as face recognition, since the landmarks could be facial features such as the eyes, nose, and mouth. This can be achieved by using a fully–connected layer with $2n$ outputs, where n is the number of landmarks.

3.1 Convolutional Implementation of Sliding Windows

The **Sliding Windows Technique** consists in training a neural network to recognise objects in small windows of an image, and then sliding the window across an image to see if the object is present. The technique can be repeated iteratively, where on each run the window size is increased. This is a computationally expensive technique, since we need to run the neural network multiple times.
Through a convolutional implementation, this method can be rendered feasible. The idea is to use a convolutional layer to replace the fully–connected layer. This convolutional layer will have a filter of size $n \times n \times n_c$, where n is the size of the window, and n_c is the number of channels. The output of this layer will be a $1 \times 1 \times 1$ tensor, which can be passed to a sigmoid function to obtain the probability that the object is present in the window. This is clearly presented in the following image:

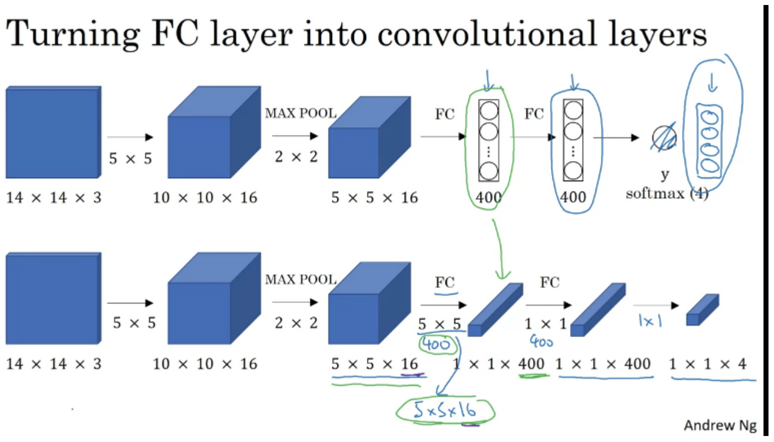


Figure 12: Convolutional implementation of sliding windows, sourced from this lecture.

An issue with the convolutional implementation of sliding window is that it does not output the most accurate bounding box coordinates. This is because the convolutional layer is not fully–connected. To solve this, we can use a fully–connected layer to refine the coordinates, which is done in the **YOLO** algorithm.
The basic idea behind this algorithm is to divide the image into a grid of $S \times S$ cells, and then for each cell, predict the probability that an object is present in the cell, the bounding box coordinates, and the class of the object. The output of the algorithm is a $S \times S \times (5B + C)$

tensor, where B is the number of bounding boxes predicted per cell, and C is the number of classes. The algorithm will assign the object to the grid cell where the centrepoint of the object is found, which prevents duplicate counts of objects.
The function **Intersection over Union** (IoU) is used to determine how accurate the predicted bounding box is. It is defined as:

$$\text{IoU} = \frac{\text{Area of intersection}}{\text{Area of union}}$$

By convention, a lot of CV algorithms use a threshold of 0.5 for the IoU.
One of the problems of object detection is that the algorithm may find multiple detections of the same object. This is solved by using a technique called **non–max suppression**. The algorithm is as follows:

1. Discard all boxes with $p_c \leq 0.6$.
2. While there are any remaining boxes:
   a) Pick the box with the largest p_c.
   b) Discard any remaining box with IoU > 0.5 with the box selected in the previous step.

If there are multiple classes, then the algorithm is run for each class separately.
Object detection, as described earlier, can only detect one object per grid. To solve this, we can use a technique called **anchor boxes**. The idea is to modify the vector output of the algorithm to be of the form:

$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_n \\ p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_n \\ \vdots \end{bmatrix}$$

Where the first set of values corresponds to the first anchor box, and the second set of values corresponds to the second anchor box. If there are multiple objects in the same grid, then the algorithm will assign the object to the anchor box with the highest IoU, and the other object to the other anchor box.
Let us put all of these concepts together to explain the **YOLO** algorithm more in detail. Every training sample will be of dimensions $3 \times 3 \times a \times 5 + c$, where a is the number of anchor boxes, and c is the number of classes. The output of the algorithm will be of dimensions $3 \times 3 \times a \times (5 + c)$, where the first 5 values correspond to the bounding box coordinates and the probability that an object is present, and the last c values correspond to the probability that the object belongs to each class.
The algorithm will make predictions by using a 3×3 grid, and for each cell, it will predict the bounding box coordinates and the probability that an object is present. The algorithm will assign the object to the anchor box with the highest IoU (non–max supression).

Another very influential idea in Computer Vision is **Region Proposals** (R–CNN): the idea is to use a neural network to propose regions of an image where an object may be present, and then use a second neural network to classify the object. This is a very computationally efficient technique, since the second neural network only needs to run on the proposed regions, and not on the entire image.
The original R–CNN algorithm was very computationally expensive, since it used a sliding window technique to propose regions. This was solved by using a **Convolutional Implementation of Sliding Windows**, which is a convolutional layer that outputs the probability that an object is present in a window.
The first neural network is called the **Region Proposal Network** (RPN), and it is a convolutional neural network that takes as input an image, and outputs a set of bounding boxes and the probability that an object is present in each bounding box, an algorithm known as **segmentation algorithm**. The second neural network is called the **Fast R–CNN**, and it takes as input the proposed regions, and outputs the class of the object.
There is even a **Faster R–CNN** algorithm, which uses a convolutional network to propose regions.

Semantic Segmentation is another very important idea in Computer Vision. The idea is to assign a class to each pixel in an image. This is different from **Image Classification**, where the algorithm assigns a class to the entire image. The reason why we do this is because it allows us to detect multiple objects in an image, and it allows us to detect objects that are partially occluded. Semantic Segmentation will output a complete matrix full of labels, where each label corresponds to a class — one per pixel. This is different from **Object Detection**, where the algorithm will output the coordinates of bounding boxes and the class of the object in each bounding box.
The architecture of a semantic segmentation algorithm is similar to that of a convolutional neural network, except that, instead of eventually passing the output through a fully–connected layer, we will make it become bigger and bigger, until it is the same size as the input image. We achieve this through implementing **transpose convolutions**. Transpose convolutions are the opposite of convolutions: instead of shrinking the image, they will expand it. The way

they work is multiplying each element of the image by a filter (which is a matrix), and then projecting the results onto the output. Similar to regular convolutions, we can add padding and strides to transpose convolutions. When the values of the multiplication of two different entries in the image overlap, we sum them.
This is the architecture of a semantic segmentation algorithm:

$$\begin{bmatrix} 3\times 3 \text{ conv, } 64 \\ 3\times 3 \text{ conv, } 64 \\ 2\times 2 \text{ max pool} \\ 3\times 3 \text{ conv, } 128 \\ 3\times 3 \text{ conv, } 128 \\ 2\times 2 \text{ max pool} \\ 3\times 3 \text{ conv, } 256 \\ 3\times 3 \text{ conv, } 256 \\ 3\times 3 \text{ conv, } 256 \\ 2\times 2 \text{ max pool} \\ 3\times 3 \text{ conv, } 512 \\ 3\times 3 \text{ conv, } 512 \\ 3\times 3 \text{ conv, } 512 \\ 2\times 2 \text{ max pool} \\ 3\times 3 \text{ conv, } 512 \\ 3\times 3 \text{ conv, } 512 \\ 3\times 3 \text{ conv, } 512 \\ 2\times 2 \text{ max pool} \\ \text{Fully connected} \\ \text{Transpose conv, } 512 \\ \text{Transpose conv, } 512 \\ \text{Transpose conv, } 512 \\ 2\times 2 \text{ upsample} \\ \text{Transpose conv, } 512 \\ \text{Transpose conv, } 512 \\ \text{Transpose conv, } 512 \\ 2\times 2 \text{ upsample} \\ \text{Transpose conv, } 256 \\ \text{Transpose conv, } 256 \\ \text{Transpose conv, } 256 \\ 2\times 2 \text{ upsample} \\ \text{Transpose conv, } 128 \\ \text{Transpose conv, } 128 \\ 2\times 2 \text{ upsample} \\ \text{Transpose conv, } 64 \\ \text{Transpose conv, } 64 \\ \text{Transpose conv, } c \end{bmatrix}$$

Where c is the number of classes.

This diagram illustrates the architecture of a semantic segmentation algorithm (**U–Net**):

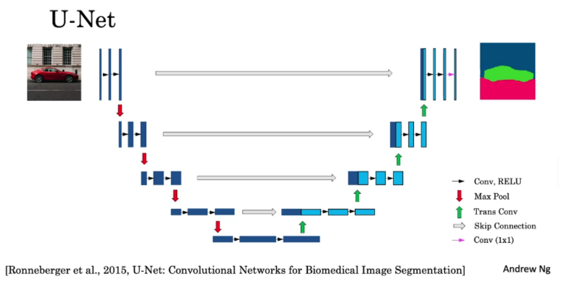

U-Net
Conv, RELU
Max Pool
Trans Conv
Skip Connection
Conv (1x1)
[Ronneberger et al., 2015, U-Net: Convolutional Networks for Biomedical Image Segmentation]
Andrew Ng

Figure 13: U–Net, sourced from this lecture.

Notice that the dimension of the output is $h \times w \times c$, where h and w are the height and width of the input image and c is the number of classes.

4 Special Applications: Face Recognition & Neural Style Transfer

There are two big types of facial algorithms: **Face Verification** and **Face Recognition**. The former refers to the task of verifying whether a person is who they say they are, and the latter refers to the task of recognizing a person in an image.
The building blocks of a face recognition algorithm are high–accuracy face verification algorithms. Thus, we will cover these first and then go on to proper face recognition algorithms.
One of the challenges of face recognition is that the algorithm must be able to recognise the person given only one image of them. This is different from other applications, where the algorithm is given many images of the object it must recognise. This is known as **One–shot Learning**. To make this work, we need to train a **similarity function** that takes as input two images and outputs a number between 0 and 1 that represents how similar the two images are. If the output is smaller than a certain threshold τ, then the algorithm will output that the two images are of the same people. Otherwise, it will output that they are of different people.
The similarity function d is trained using a **Siamese Network**. This is a neural network that takes as input two images, and outputs a number between 0 and 1 that represents how similar the two images are. The architecture of a Siamese Network is as follows:

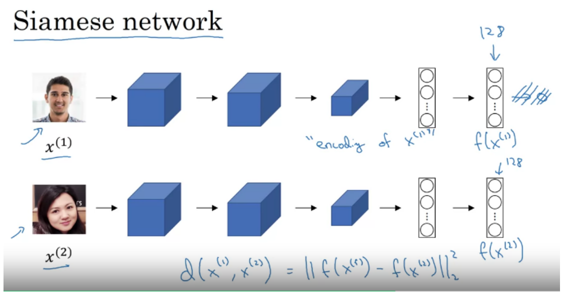


Figure 14: Siamese Network, sourced from this lecture.

As you can see, you will pass two different pictures through two convolution networka so that they are *encoded* and then you will pass the two encodings through a fully–connected layer to get the similarity score.
The objective in training a Siamese Network is to minimise the value of d given two pictures of the same person, and maximise the value of d given two pictures of different people.
One way to learn the parameters of the CNN is to use the **triplet loss function**:

$$\mathscr{L}(A,P,N) = \max\left(\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 + \alpha, 0\right)$$

It is called the triplet loss because your algorithm will be looking at three different pictures each time: an *anchor* picture A, a *positive* picture P (of the same person as the anchor) and a *negative* picture N (of a different person). The objective is to make the distance between the anchor and the positive as small as possible, and the distance between the anchor and the negative as big as possible, namely:

$$\|f(A) - f(P)\|^2 + \alpha \leq \|f(A) - f(N)\|^2$$
$$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leq 0$$

The α is a hyperparameter that determines how big the difference between the two distances must be.
Thus, the loss function of the Siamese Network is:

$$mathcalL(A^{(i)}, P^{(i)}, N^{(i)}) = \max\left(\left\|f(A^{(i)}) - f(P^{(i)})\right\|^2 - \left\|f(A^{(i)}) - f(N^{(i)})\right\|^2 + \alpha, 0\right)$$

And the overall cost function is:

$$\mathscr{J} = \sum_{i=1}^{m} \mathscr{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

Note that this is the **Euclidean distance** between the two encodings. It is also possible to use the **Cosine similarity**:

$$\mathscr{L}(A,P,N) = \max\left(\frac{\langle f(A), f(P)\rangle}{\|f(A)\|\,\|f(P)\|} - \frac{\langle f(A), f(N)\rangle}{\|f(A)\|\,\|f(N)\|} + \alpha, 0\right)$$

Choosing A, P, and N during training is non–trivial: you want to choose triplets that are *hard* to train on, i.e. triplets that are close to each other. If you choose triplets at random, then the algorithm will not learn much, because it will not be too difficult to get the right answer.
In *PyTorch*, you can implement the triplet loss function as follows:

```
    '''
    Inputs:
        tensors a, p, n of shape (N,D)
        alpha: margin (scalar)
    '''
    triplet_loss = nn.TripletMarginLoss(margin=1.0, p=2)
    anchor = torch.randn(100, 128, requires_grad=True)
    positive = torch.randn(100, 128, requires_grad=True)
    negative = torch.randn(100, 128, requires_grad=True)
    output = triplet_loss(anchor, positive, negative)
    output.backward()
```

Another way to learn the parameters of a face recognition system is to treat it as a **classification problem**. You can train a neural network to output a 128 dimensional vector encoding a face, and then train a softmax classifier on top of that.

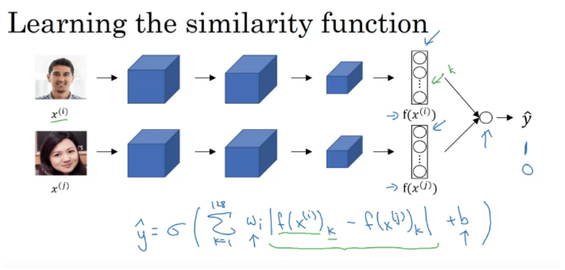

Learning the similarity function
$x^{(i)}$
$x^{(j)}$
$f(x^{(i)})$
$f(x^{(j)})$
$\hat{y}$
1
0
$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_i \left| f(x^{(i)})_k - f(x^{(j)})_k \right| + b\right)$

Figure 15: Face recognition as a classification problem, sourced from this lecture.

Other variations of the loss function, such as the **Chi–square loss** and the **Contrastive loss** are also possible. This algorithm was presented in the **FaceNet** paper.

4.1 Neural Style Transfer

Neural Style Transfer is a technique that allows you to take the style of one image s and apply it to another image c. In order to implement this, you need to look at the features that the ConvNet is extracting from the images.
In the earlier layers, neurons tend to detect edges and simple shapes, while in the deeper layers, neurons tend to detect more complex shapes. This is because each hidden unit is looking at a larger patch of the image.
The first component of our neural style transfer algorithm is the **cost function**:

$$\mathcal{J}(G) = \alpha \mathcal{J}_{\text{content}}(C, G) + \beta \mathcal{J}_{\text{style}}(S, G)$$

The first element of this function is the *content cost function*, which will make sure that the generated image G has the same content as the image C. The second element, the *style cost*

function, will make sure that the generated image G has the same style as the image S. We will weight these two costs with the hyperparameters α and β.
In order to generate a new image, what we do is the following:

1. Initialise the image G with random values.
2. Use gradient descent to minimise the cost function $\mathscr{J}(G)$. This will update the pixel values of the image G.

This being said, we need to see how to define both the content and style cost functions. The former is defined as follows:

$$\mathscr{J}_{\text{content}}(C,G) = \frac{1}{2 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{[l](C)} - a^{[l](G)})^2$$

Where l stands for the layer in the network and a stands for activation. Notice that if you use a hidden layer that is deeper in the network, the output will keep more of the content of the original picture. Conversely, if you use a layer that is higher up in the network, the output will keep less of the content of the original picture and more of the style.
The style cost function is defined as follows:

$$\mathscr{J}_{\text{style}}(S,G) = \frac{1}{(2 \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]})^2} \sum_k \sum_{k'} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)})^2$$

Where $G^{[l](S)}$ and $G^{[l](G)}$ are the *Gram matrices* of the images S and G respectively. The Gram matrix is a matrix of dot products between the different filter activations. Thus, the definition of the Gram matrix is:

$$G_{kk'}^{[l]} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l]} a_{ijk'}^{[l]}$$

Where $a_{ijk}^{[l]}$ is the activation of the kth filter at position (i,j).
The Gram matrix is a way to measure the correlation between different filter activations. Correlation refers to which high level features appear or do not appear together in an image.
Thus, we will compute the Gram matrix for both the style and generated images, and then compute the style cost function.
The final step is to combine the content and style cost functions into a single cost function, and then use gradient descent to minimise it.