



Chapter 14

Deep Computer Vision Using Convolutional Neural Network



GitHub Page: [LINK](#)

Google Drive: [LINK](#)

- Computers are able to beat the best chess player in 1996 but it's not until recently computer vision can perform reliably perform tasks
- Convolutional neural networks emerged from study of the brain's visual cortex
- CNN has been used in image classification since 1980s
- thanks to increase in computational power amount of available training data, computer vision has exceeded human performance
- CNN can:
 - power image search
 - be used in self-driving cars
 - perform automatic video classification
 - Not only in images:
 - voice recognition
 - natural language processing

The Architecture of the Visual Cortex

- Visual vortex has small local receptive field meaning they react to only visual stimuli located in a limited region
- Some neurons in local receptive field react to only horizontal lines while others only react to lines with different orientation

- Some neuron has larger reception field to receive more complex information
- This powerful architecture can detect any image
- These studies inspired the creation that eventually led to convolutional neural network
- An important milestone was a 1998 paper by Yann LeCun et al
- A famous architecture called LeNet-5
- It introduces 2 new building blocks:
 - Convolutional layers
 - pooling layers

Convolutional Layers

- Most important building block for CNN is convolutional layer
- First layer is not connected to every pixel in input image but only pixels in their receptive fields
- Second layer is only connected to the neurons from the first layer
- This allows network to focus on small low-level features in the first hidden layer
- This is common in real world images therefore it performs well



Diagram on page 448

- CNN scans through each row in a square of the value your choice (3*3 in example) and it converts it to a abstract feature map
- To make the image the same size it's common to have zero padding around the image
- Also possible to connect a large input layer to smaller layer by spacing out receptive fields

➡ Diagrams on page 449-450

Filters

- Weights in CNNs are called filters
- The vertical filters enhances vertical lines
- Horizontal filters enhances horizontal lines as shown in Figure 14-5

➡ See page 451 for diagram

Stacking Multiple Feature Maps

- For simplicity all examples before are in 2D but realistically it should be in 3D
- But amount of dimension could be set to any number
- IT has one neuron per pixel in each feature map
- all neurons within given feature map share the same parameters
- Neurons in different feature maps use different parameters
- Input images are also composed of multiple layers
 - Typically there are three layers(RGB):
 - Red
 - Green
 - Blue
 - Grayscale only has one layer

➡ Diagram on page 452

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k}$$

$$with = \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ is the output of the neuron located in row i , column j in feature map k of the convolutional layer l
- s_h and s_w are vertical and horizontal
- f_h and f_w are the height and width of the receptive field
- $f_{n'}$ is the number of feature maps in the previous layer
- $w_{i',j',k'}$ is the output of the neuron
- b_k is the bias term for feature map k
- $w_{u,v,k',k}$ is the connection weight between any neuron in the feature map k


TensorFlow Implementation

- In TF, each input image is typically represented as a 3D tensor:
 - height
 - width
 - channels
- A mini-batch is represented as 4D tensor shape:
 - mini-batch size
 - height
 - width
 - channels
- The code below is an example using 2 sample images

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7d532654-b388-4824-a169-6cde27aaf520/skl_sample_image.py
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0dcbd2ab-4626-4a87-a27b-3ecb934c665b/skl_sample_image.ipynb

- Going through the code, first normalize the layers by dividing 255 to get the range to from 0-1

 Padding diagram on page 455

- Manually define filters in the example below

```
conv = keras.layers.Conv2D(filters=32, kernel=3, strides=1, padding="same",  
                           activation="relu")
```


- then create 2 a 2×2 filters
- apply both images using `tf.nn.conv2d()`
- Using zero padding and stride one in the `conv2d()`
- plot one of the resulting feature maps
- `Conv2d()`
 - `images` is the input mini-batch (4D tensors)
 - `filters` is the set of filters to apply (Also 4D)
 - `Strides` is equal to 1 but it could also be 1D array with 4 elements but not usually used
 - `Padding` can either be "SAME" or "VALID"
 - "SAME" layers use zero padding if necessary
 - "VALID" layer does not use zero padding and may ignore some rows and columns at the bottom and right of the input image
-
- Creates a Conv2D layer with 32 filters each 3×3 using stride 1
- Using "same" padding

Memory Requirements

- Another issue with CNNs is that convolutional layers require a huge amount of RAM
- For example, consider a convolutional layer with:
 - 5×5 filters
 - outputting 200 feature maps of size 150×100
 - with 1 stride and "same" padding
 - If the input is 150×100 RGB
 - then there are $15200+1$ (bias) parameters
 - 200 feature maps contains 150×100 neurons
 - each neuron needs to compute weighted sum of 75 inputs
 - The total 225 MILLION float operation
 - which would occupy 12MB of RAM
 - that's just one instance
 - if training a batch containing 100 instances then it would take up 1.2GB of RAM

Pooling Layers

- The goal for pooling layers is to subsample the input image in order to reduce the computational load, memory usage and number of parameters
- Each neuron in pooling layer is connected to the outputs of limited number of neuron provided by the layer
- It's required to define its size stride and padding type
- Pooling has no weights
- Figure 14-8 is an example of max pooling:
 - There is no padding in the example
 - It takes the largest value in each box and scan across the image

 Diagram on page 457


- max pooling layer also introduces some level of invariance to small translation
- As mentioned previously, Max pooling may have invariance issue
- On the example fig 14-9, A and B is correct but on C, we are expecting it to be the same as A and B but it was completely shifted to the right side
- This could be useful when prediction does not depend on details like classification
- The downside to pooling is the fact that it's very destructive, it removes 75 percent of the pixels and in some cases, invariance is not desired at all

TensorFlow Implementation

- Implementing max pooling layer in TensorFlow is easy, the code below creates a 2×2 Max pooling layer

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

- To create average pooling just use AvgPool2D instead of MaxPool2D
- AvgPool2D is the same as MaxPool2D but it computes the average instead of the max
- Max pooling only reserves the strongest feature but average pooling might create translation variance

 An diagram showing CNN is on page 459

- Keras does not include a depthwise max pooling layer but TF's low level API does
- To do that just use `tf.nn.max_pool()` function and specify the kernel size and strides as 4-tuples
 - First 3 values should be 1

- Indicates kernel size and stride along the batch, height and width
- the last value is whatever kernel size and stride you want along the depth dimension
- The last value must be a divisor of the input depth

```
output = tf.nn.max_pool(images, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                        padding="valid")
```

- To include this in Keras layer

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                             padding="valid")
```

- The last type of layer is seen very often in the modern world, it's called global average pooling layer
- It works very differently, it compute the mean of each entire feature map, this means that it just outputs a single number per feature map per instance
- although it's very destructive but it could be useful in as the output layer
- To implement this:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

- It's also equivalent to a Lambda layer

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

CNN Architectures

- Typical CNN stacks a few convolutional layers then a pooling layer and then repeat
- Typically the image gets smaller and smaller and the network gets deeper and deeper



An graph showing this is on page 461 Figure 14-11

- A common CNN for Fashion MNIST

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu"),
    keras.layers.Conv2D(128, 3, activation="relu"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu"),
    keras.layers.Conv2D(256, 3, activation="relu"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

- Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/dc7c0b02-d9ad-4b29-b8e6-8d9548e8fe87/CNN_Fasion_MNIST.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2f68e320-5919-4463-bc58-aa9eb482ce08/cnn_fasion_mnist.py

- Create a 7×7 filter with no strides with input shape of [28, 28, 1]
- Pool size of 2 so it divides each spatial dimension by a factor of 2
- Repeat the structure twice
- The number of filters grows as CNN goes towards output
 - $64 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow \dots$
- Next create a fully connected layer with a dropout of 50 percent each layer to reduce overfitting

- This CNN has the accuracy of 92%
- Image classification challenge ILSVRC ImageNet challenge increased from error rate 26% to 2.3% in the past 5 years

LeNet-5

- The most widely known CNN architecture



Table 14-1 Le-Net on page 463

- MNIST have 28×28 pixels and with zero padding it becomes 32×32 pixels
- The average pooling computes each neuron computes the mean of its input then multiplies the result by learnable coefficient and adds a learnable bias term
- each neuron outputs the square of the Euclidian distance between its input vector and its weight vector

AlexNet

- Won 2012 ImageNet ILSVRC by large margin, achieved error rate of 17% while second place achieved 26%



Table of AlexNet architecture on Page 464

- To reduce overfitting, the authors used two regularization techniques:
 - first applied dropouts with 50 percent dropout rate
 - Second they performed data augmentation by randomly shifting the training images by various offsets
- AlexNet also uses a competitive normalization step immediately after the ReLU step of layer C1 and C3 called local response normalization (LRN)
- Equation for LRN:

$$b_i = a_i(k + \alpha \sum_{j=j_{low}}^{j_{high}} a_j^2)^{-\beta} \text{ with } \begin{cases} j_{high} = \min(i + \frac{r}{2}, f_n - 1) \\ j_{low} = \max(0, i - \frac{r}{2}) \end{cases}$$

- b_i is the normalized output of the neuron located in feature map i , at some row u and column v
- a_i is the activation of the neuron after ReLU step and before normalization
- k, α, β and r are hyperparameters, k is bias and r is depth radius
- f_n is the number of feature map

Data Augmentation

- Data augmentation artificially increases the size of the training set by offsetting the images in different directions
- This reduces the chances of overfitting which makes this a regularization technique
- Slightly shifting the image helps when there isn't enough image so this simulates an image taken in different directions



Image examples on page 465

Google Net

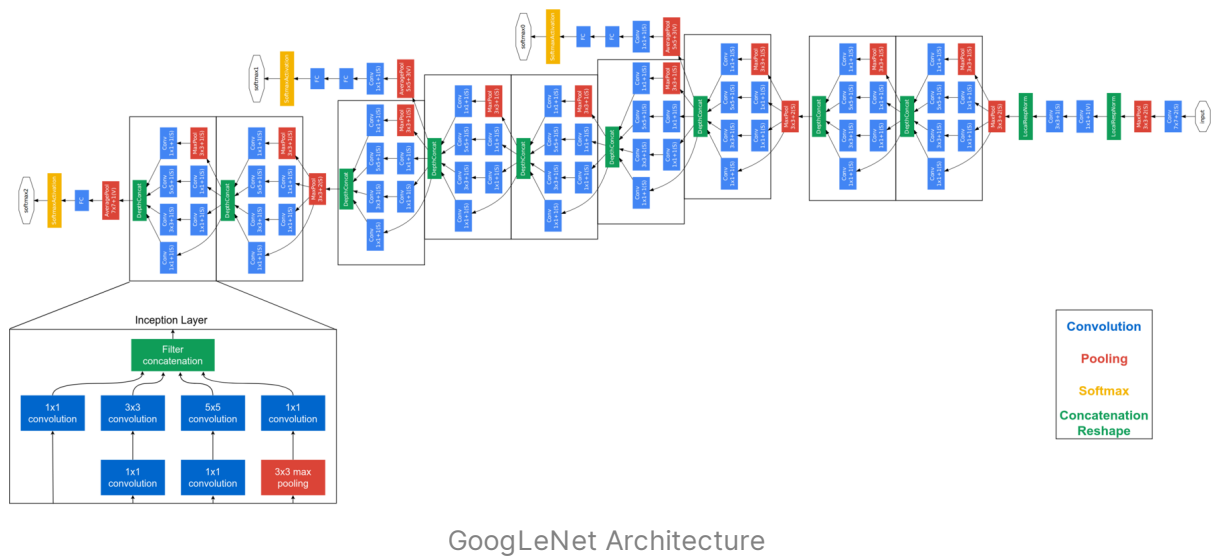
- won ILSVRC 2014 by pushing error rate below 7%
- A reason is because it's much deeper than other CNN
- This is possible by subnetworks called inception modules
- Figure 14-13 shows the architecture of an inception module



Figure 14-13 on page 467

- Inception module only have CNN of 1×1 kernels, these serves several purposes:
 - Although cannot capture pattern but can capture depth

- configured to output fewer feature maps than their inputs so they serve as bottleneck layers which means that they reduce dimensionality
 - This speeds up computation and improves generalization
- Each pair of convolutional layers acts like a single powerful layer capable of capturing complex patterns
- The network is so deep it has to be represented in a few columns
- All layers are using ReLU activation function



GoogLeNet Architecture



A Better Graph on page 469 Figure 14-14

- Network architecture:
 - First two layers divide the image's height and width by 4 to reduce computational load
 - First layer uses a large kernel size to preserve much information
 - Local response normalization layer ensures that the previous layers learn a wide variety of features
 - Two convolutional layers after that acts as a bottleneck layer
 - Next max pooling reduces the image height and width by 2

- Then a stack of 9 inception module interleaved with couple max pooling layers reduce dimension and speed up the net
- Next global average pooling layer outputs the mean of each feature map
- Input image is typically 224×224 and after 5 max pooling it get reduced to 7×7
- last layer has dropouts for regularization and fully connected layer with 1000 units and a softmax activation for output
- Several variants of GoogLeNet were later proposed by google including:
 - Inception-v3 and Inception-v4

VGGNet

- Runner-up for ILSVRC 2014 challenge
- Very classic architecture:
 - 2 or 3 convolutional layers and pooling layers
 - 2 or 3 convolutional layer and a single pooling layer
 - then repeat up to 16 or 19 convolutional layers
 - Final dense network with 2 hidden layers and a output layer
- The network uses 3×3 filters but using lots of them

ResNet

- Won ILSVRC 2015 using a Residual Network
- error rate under 3.6%
- extremely deep CNN composed of 152 layers
- The models are getting deeper and deeper with fewer and fewer parameters
- The reason its possible to train a network so deep is because of skip connections:
 - The signal deeding into a layer is added to a layer located higher up the stack

- The goal for training neural network is to make it model a target function $h(x)$
- If add the input x to the output of the network
- The network will be forced to model $f(x) = h(x) - x$ rather than $h(x)$
- The technique above is called residual learning



Residual learning diagram on page 471 Figure 14-15

- When initialize a regular network the weights are close to zero, network just computes values close to zero
- If add skip connection the resulting network just outputs a copy of its input
- This speeds up training considerably
- If add multiple skip connections the network can start making progress even if several layers have not started learning yet
- Deep residual network can be seen as a stack of residual units where each residual unit is a small neural network with a skip connection



Figure 14-16 regular deep neural network and deep residual network page 472

- Starts and ends exactly like GoogLeNet without dropout layers
- In between is just a deep stack of simple residual units
- Each residual unit is composed of two convolutional layers with Batch normalization and ReLU and no Pooling
- It uses 3×3 and preserving spatial dimensions



Figure 14-17 ResNet architecture on page 473

- Number of number map is doubled every few residual units at the same time as their height and width are halved

- When this happens it cannot be directly added to the output residual since they have different shapes
- A solution to this problem is to create 1×1 convolutional layer with stride 2 and right number of output feature maps



Diagram for skip connection when changing feature map size on page 473 Figure 14-18

- ResNet-34 is ResNet with 34 layers
 - Containing 3 residual units that output 64 feature maps
 - 4 RUs with 128 maps
 - 6 RUs with 256 maps
 - 3 RUs with 512 maps
- ResNets deeper than that like ResNet-152 use slightly different residual units
 - Instead of two 3×3 convolutional layers:
 - First 1×1 convolutional layer with 64 feature maps
 - then 3×3 convolutional layer with 64 feature maps
 - and then 1×1 convolutional layer with 256 feature maps
- ResNet-152 contains:
 - 3 such RUs that output 256 maps
 - 8RUs with 512 maps
 - 36 RUs with 1024 maps
 - 3 RUs with 2048 maps

Xception

- Proposed in 2016 and significantly outperformed Inception-v3
- Merged GoogLeNet and ResNet
- Replace inception modules with special type of layer called depth-wise separable convolution layer

- Normal CNN tries to capture spatial patterns and cross channel pattern at the same time
- Xception separates spatial and cross-channel patterns:
 - First part applies a single spatial filter for each input feature map
 - Second part looks exclusively for cross channel patterns



A diagram on page 475 Figure 14-19

- Avoid using them after layers that have too few channels such as input layer
- Xception starts with 2 regular convolutional layers
- The rest of the network uses only separable convolutions and a few max pooling layers
- The reason Xception is a variation of GoogLeNet is because of the intermediate between traditional convolutional layers and other layers
- In practice separable convolutional layers generally perform better

SENet

- Winning architecture in ILSVRC 2017
- stands for Squeeze-and-Excitation Network
- astonishing 2.25% error rate
- The boost comes from the network adding a small neural network called SE block to every unit in the architecture



Diagram showing SE-Inception module and SE-ResNet Unit On page 476 Figure 14-20

- An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension and it learns which features are usually most active together
- It uses these information to calibrate the feature map

- Take a face for example:
 - When you see a mouth you typically see nose and eyes too
 - If see mild activation on eyes then the network will boost feature map of the eyes
- feature map recalibration will help resolve the ambiguity



SE Block Diagram on page 477 Figure 14-21

- An SE block is composed of 3 layers:
 - global average pooling layer
 - hidden dense layer using ReLU
 - dense output layer using sigmoid activation function



A diagram of SE block on page 477 Figure 14-22

- average pooling layers computes the mean activation function for each feature map
 - if its input contains 256 feature maps, it will output 256 numbers representing the overall level of response for each filter
- Next layer where "squeeze" happens:
 - This layer has significantly fewer than 256 neurons and typically 16 times fewer than the number of feature maps
- 16 times fewer than the number of feature maps
- 256 numbers get compressed into a small vector
- This bottleneck step forces the SE block to learn a general representation of the feature combinations
- The output layer takes the embedding and outputs a recalibration vector containing one number per feature map each between 0 and 1

Implementing a ResNet-34 CNN Using Keras

- To implement a ResNet-34, first create a ResidualUnit layer

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1a52515d-d5be-4466-8a00-3b00e2ecfaa0/resnet_34.py
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c98e4978-508f-40b8-9695-5c1449f42a24/resnet_34.ipynb
```

- This is under 40 lines of code and this won the 2015 ILSVRC challenge

Using Pretrained Models from Keras

- Generally it's not required to implement GoogLeNet or ResNet manually
- There's lots of pretrained networks in Keras

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d88042c1-a1bd-4e51-900e-cde262e4e0cd/pretrained_network_keras.ipynb
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/084ab2c9-5c8e-4433-88da-448219e7a5e8/pretrained_network_keras.py
```

Pretrained Models for Transfer Learning

- When building a network without enough data it's good to reuse lower layers of a pre-trained model

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9e730b06-a07c-4406-8526-9127a0a9cd1e/pretrained_transfer_learning.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/abc00beb-3521-4758-887e-5551b23e79c1/Pretrained_transfer_learning.ipynb

Classification and Localization

- Localizing object in a picture is an regression task
- to predict a bounding box around the object is the common approach
- Just required to add a second dense output layer with 4 units
- Can be trained using MSE loss

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2bc8ef64-7bda-4aac-9157-ecf7f6060acd/classification_and_localization.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/978fde0e-ef50-490b-8e72-955190eeafa5/classification_and_localization.ipynb

Object Detection

- A common approach was to train a CNN that classifies only one object and slide it though the image
- An example is given in figure 14-24, it scans though a region of 3×3



Diagram showing and example on detecting multiple objects on page 486 Figure 14-24

- This technique is straight forward but here are some issues:
 - post-processing might be needed to get rid of all the unnecessary bounding boxes
 - might detect the same object multiple times
- A common approach is called **non-max suppression**:
 - First add extra objectness output to the CNN to estimate the probability that the flower is indeed present
 - Must be sigmoid activation function
 - can train using binary cross-entropy loss
 - Then get rid of all the bounding boxes
 - Find bounding box with highest objectness score and get rid of all the other bounding boxes that overlaps
 - Repeat step two until there are no more bounding boxes to get rid of
- This approach is simple but requires to run CNN multiple times therefore it's not efficient
- A better approach is using a fully convolutional network (FCN)

Fully Convolutional Network (FCN)

- Idea was introduced in 2015
- It was proposed to replace the dense layers at the top of CNN by convolutional layer
- When it's replaced by layers with 1×1 per feature map, it's identical to using dense layers:
 - Convolutional layer: [batch-size, 1, 1, 200]
 - Dense layer: [batch-size, 200]
- This is important because dense layer requires a specific input size while convolution layer doesn't, it takes any image size
- Suppose a network is trained on 224×224 images
 - If you want to use an 448×448 image instead, it's possible since the bottleneck layer would change the size of the filter

- Figure 14-25 shows this
- You Only Look Once is also a popular object detection architecture



Diagram showing FCN processing different size images on page 489
Figure 14-25

You Only Look Once (YOLO)

- Proposed in 2015
- Improved in 2016 (YOLOv2) and 2018 (YOLOv3)
- YOLOv3's architecture is similar to FCN with a few differences:
 - Outputs have bounding boxes for each grid cell instead of 1
 - outputs 20 class probabilities per grid cell
 - 45 numbers per grid cell:
 - 4 coordinates
 - 5 objectness scores
 - 20 class probabilities
 - Predicts offset relative to coordinates instead of absolute coordinate of bounding box
 - Trained to predict only bounding boxes whose center lies in that cell
 - applies logistic activation function to the bounding box coordinates to ensure in range 0-1
 - Before training YOLOv3 finds 5 representative bounding box dimensions called **anchor boxes**
 - It does this by applying k-mean algorithm to height and width of training set bounding box
 - The rescaling happens before training therefore it speeds up training
 - The network is trained using images of different scales:
 - This allows it to detect objects in different scales

- The smaller the scale the faster but less accurate
- The larger the scale the slower and more accurate
- There are a few YOLO implementations built using TensorFlow available on GitHub
- some have been ported to TF hub like:
 - Single Shot Multibox Detector (SSD)
 - Faster R-CNN
- The choice depends on many factors:
 - speed
 - Accuracy
 - Training time
 - Complexity

Semantic Segmentation

- each pixel is classified according the class of the object it belongs to
- An example on figure 14-26
- Although it separates it into classes but each different object isn't classified
- The challenge is that spatial resolution would be lost since strides are greater than 1
 - A solution may be up-sampling
- A fairly simple implementation have been discussed (FCN)



Example of Semantic segmentation on page 493 Figure 14-26

- Are several solutions available for up-sampling
- Bilinear interpolation only works reasonably well up to $\times 4$ or $\times 8$
- Instead, an better option would be using transposed convolutional layer

- Equivalent to first stretching the image by inserting empty rows and columns
- Then perform a regular convolution
- Alternatively some people prefer to think of it as a regular convolutional layer that uses fractional strides
- just add Conv2DTranspose layer to use it in Keras



Diagram for upscaling using transposed convolutional layer on page 493 Figure 14-27

TensorFlow Convolution Operations

```
keras.layers.Conv1D
```

- Creates a convolutional layer for 1D inputs such as:
 - Time series
 - Text

```
keras.layers.Conv3D
```

- Creates a convolutional layer for 3D inputs such as:
 - 3D PET scans

```
dilation_rate
```

- Setting this hyperparameter of any convolution layer to value 2 or more creates an a-trous convolutional layer
- Equivalent to using regular convolutional layer with a filter dilated by inserting rows and columns of zeros
- For example `[[1,2,3]]` could be have a dilation 4 and change to:
 - `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`

```
tf.nn.depthwise_conv2d
```

- Can be used to create depthwise convolutional layer
 - It applies every filter to every individual input channel independently
 - there are f_n filter and $f_{n'}$ input channels then this would output $f_n \times f_{n'}$ feature maps
-
- This solution is fine but still too imprecise
 - To do better the authors added skip connections from lower layers:
 - up-sample the output image by a factor of 2 instead of 32
 - added a lower layer that had this double resolution
 - Then up-sample the result by a factor of 16 leading to total up-sampling factor of 32
 - This recovered some spatial resolution lost earlier
 - Output of original CNN goes through the following extra steps:
 - upscale $\times 2$
 - add the output of an even lower layer
 - and then up-scale $\times 8$
 - This increase to resolution is called **super-resolution**



Diagram Showing skip layer on page 495 Figure 14-28

- GitHub provides TensorFlow implementations of semantic segmentation and there are also pretrained instance segmentation models in TensorFlow
- Instance segmentation is similar to semantic segmentation but instead of merging all objects of the same class, each object is distinguished from the others
- Available projects on TensorFlow are based on Mask R-CNN architecture

- Not only there is a bounding box around each object, but there's also a pixel mask that locates pixels in the bounding box that belongs to the object

Exercises

1. One advantage of CNN is to reduce the number of neurons that need to be connected in each layer so this saves computational power which translates to saving time. This is also a lot more efficient compared to traditional DNN.
2. There are 19300 parameters. Which is 12.8 MB if using 32-bit float.
3. Here are some things you could do if GPU runs out of memory when training CNN:
 - Reduce Mini Batch
 - Increase strides in each layer
 - Remove one or more layers
 - Distribute on multiple machines
 - use 16-bit float instead of 32-bit float
4. Max Pooling layer has no parameters while traditional convolutional layer has multiple parameters
5. A local response normalization layer makes neurons that are active inhibit the neurons neighboring which pushes the feature map to have a broader range of features which reduces. Typically used in the lower-level layer but could be used in higher-level layers
6. AlexNet applied:
 - Data augmentation to shift the data around to avoid overfitting
 - 50 percent dropout to avoid overfitting
 - A new normalization technique called local responsive normalization
 GoogLeNet's main innovations are:
 - Bottleneck layer (Using 1×1 convolutional layer)
 - Deeper network
 ResNet:
 - Skip connection
 - Deeper network
 SENet:
 - Has Mini SE block, it's like a mini neural network in each block

Xception:

- Separate spatial and cross-channel patterns

7. Converting from a dense layer to a convolutional layer makes the input image more dynamic, so the layer could adapt to whatever resolution or aspect ratio that has been sent to the neural network. There are multiple implementations on GitHub and some in TF Hub. To convert a dense layer to FCN, just replace the dense layer with a convolutional layer that has a filter of 1×1
8. Difficulty in semantic segmentation is the loss of spatial resolution. A solution may be to use super-resolution with a combination of skip connection to make up for some or even most loss in spatial resolution
9. Code:

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3f2bc5dd-73cd-42b2-b1b0-dcabf69432e4/excercise_9_mnist_cnn.py
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/65d39dc1-3b0b-4bff-85fb-70b54479ff46/excercise-9-mnist-cnn.ipynb
```

10. Code:

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/84a892fa-a8b2-488f-97c1-23940bc5b4b7/excercise_10_transfer_learning.ipynb
```

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9cd2b250-11f5-4fd8-8908-895544efb870/excercise_10_transfer_learning.py
```

