Understanding Capsule Network Architecture

By Kshitiz Rimal (https://software.intel.com/en-us/user/1647186), published on June 19, 2018

Capsule networks (CapsNet) are the new architecture in neural networks, an advanced approach to previous neural network designs, particularly for computer vision tasks. To date, convolutional neural networks (CNN) have been used for computer vision tasks. Although CNNs have managed to achieve far greater accuracy, they still have some shortcomings.

Drawback of Pooling Layers

CNNs were built at first to classify images; they do so by using successive layers of convolutions and pooling. The pooling layer in a convolutional block is used to reduce the data dimension and to achieve something called *spatial invariance*, which means regardless of where the object is placed in the image, it identifies the object and classifies it. While this is a powerful concept it has some drawbacks. One of them is that while performing pooling it tends to lose a lot of information, information particularly useful while performing tasks such as image segmentation and object detection. When the pooling layer loses the required spatial information about the rotation, location, scale, and different positional attributes of the object, the process of object detection and segmentation becomes very difficult. While modern CNN architecture has managed to reconstruct the positional information using various advanced techniques, they are not 100 percent accurate, and reconstruction itself is a tedious process. Another drawback of the pooling layer is that if the position of the object is slightly changed the activation doesn't seem to change with its proportion, which leads to good accuracy in terms of image classification but poor in performance, if you want to locate exactly where the object is in the image.

Capsule Networks

To overcome such difficulties, a new approach was proposed by Geoffrey Hinton, called capsule network (https://arxiv.org/pdf/1710.09829.pdf)[1]. A *capsule* is a collection or group of neurons that stores different information about the object it is trying to identify in a given image; information mostly about its position, rotation, scale, and so on in a high dimensional vector space (8 dimension or 16 dimension), with each dimension representing something special about the object than can be understood intuitively (see Figure 4).

In computer graphics there is a concept of rendering, which simply means taking into account various internal representations of an object like its position, rotation, and scale, and converting them to an image on screen. In contrast to this approach our brain works in the opposite way, called *inverse graphics*. When we look at any object, we internally deconstruct it into different hierarchical sub parts, and we tend to develop a relationship between these internal parts of the whole object. This is how we recognize objects, and because of this our recognition does not depend on a particular view or orientation of the objects. This concept is the building block of capsule networks.

To understand how this works in a capsule network let's look at its architectural design. The architecture of a capsule network is divided into three main parts and each part has sub operations in it. They are:

- Primary capsules
  - Convolution
  - Reshape

    - Squash

- Higher layer capsules
    - Routing by agreement

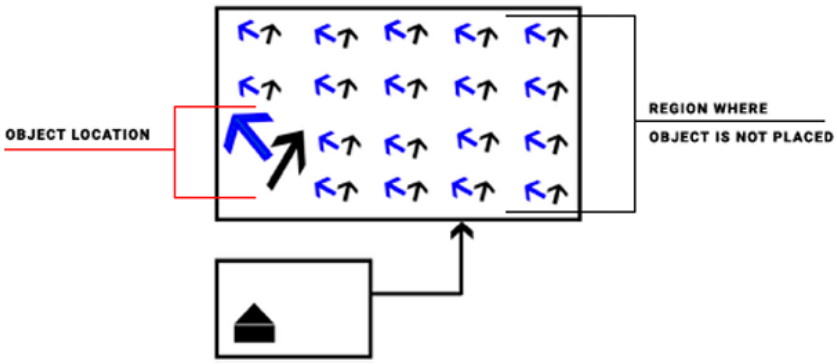
- Loss calculation
    - Margin loss
    - Reconstruction loss

1. Primary capsules

This is the first layer of the capsule network and this is where the process of inverse graphics takes place. Suppose you are feeding the network with the image of a boat or a house, like in the following images:

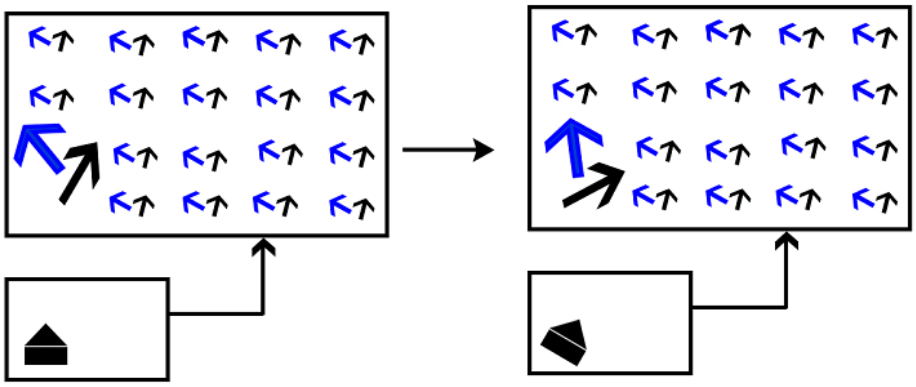
Now, these images are broken down into their sub hierarchical parts in this layer. Let's assume for the sake of simplicity that these images are constructed out of two distinct sub parts; that is, one rectangle and one triangle.

In this layer, capsules representing the triangle and rectangle will be constructed. Let us suppose we initialize this layer with 100 capsules, 50 representing the rectangle and 50 representing the triangle. The output of these capsules is represented with the help of arrows in the image below; the black arrows representing the rectangle's output, and the blue arrows representing the triangle's. These capsules are placed in every location of the image, and the output of these capsules denotes whether or not that object is located in that position. In the picture below you can see that in the location where the object is not placed, the length of the arrow is shorter and where the object is placed, the arrow is longer. The length represents whether the object is present, and the pose of the arrow represents the orientation of that particular object (position, scale, rotation, and so on) in the given image.



An interesting thing about this representation is that, if we slightly rotate the object in our input image, the

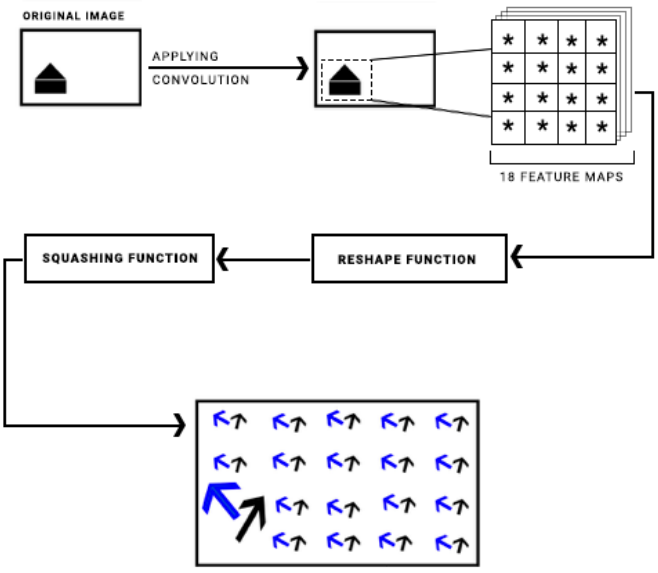
arrows representing these objects will also slightly rotate with proportion to its input counterpart. This slight change in input resulting in a slight change in the corresponding capsule's output is known as *equivariance*. This enables the capsule networks to locate the object in a given image with its precise location, scale, rotation, and other positional attributes associated with it.



This is achieved using three distinct processes:

1. Convolution
2. Reshape function
3. Squash function

In this layer the input image is fed into a couple of convolution layers. This outputs some array of feature maps; let's say this outputs an array of 18 feature maps. Now, we apply the Reshaping function to these feature maps and let's say we reshape it into two vectors of nine dimensions each (18 = 2 x 9) for every location in the image, which is similar to the image above representing the rectangle and triangle capsules. Now, the last step is to make sure that the length of each vector is not greater than 1; this is because the length of each vector is the probability of whether or not that object is located in that given location in the image, so it should be between 1 and 0. To achieve this we apply something called the *Squash function*. This function simply makes sure that the length of each vector is between 1 and 0 and will not destroy the positional information located in higher dimensions of the vector.

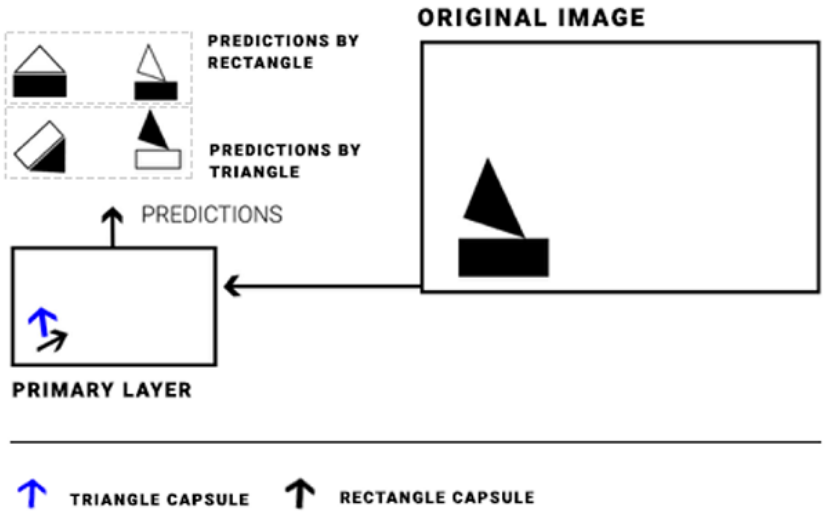


Now we need to figure out what these sub parts or capsules are related to. That is, if we consider our example of boat and house, we need to figure out which triangle and rectangle is part of the house and which is part of the boat. So far, we know where in the image these rectangles and triangles are located, using these convolutions and squashing functions. Now we need to figure out whether a boat or a house

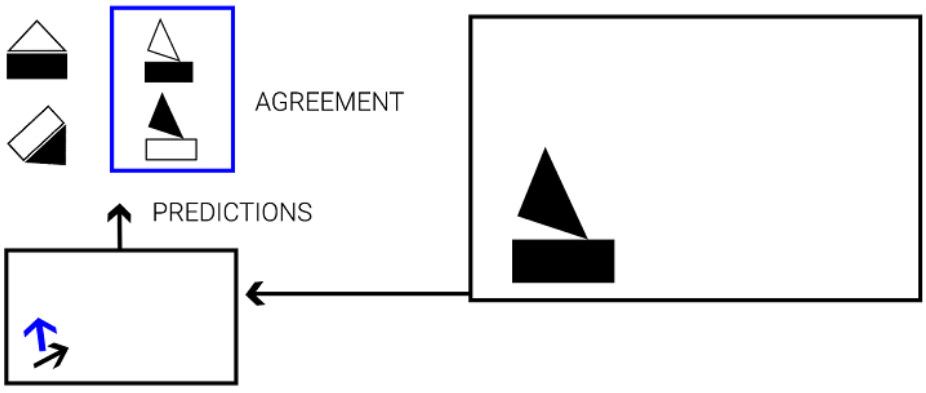
is located there and how these triangles and rectangles are related to the boat and the house.

2. Higher layer capsules

Before we get into higher layer capsules, there is still one major function left by the primary capsule layer. That is, before the higher layer capsule can operate, right after the Squash function in the primary layer, every capsule in the primary layer will try to predict the output of every capsule in the higher layer in the network; for example, we have 100 capsules, 50 rectangles, and 50 triangles. Now, suppose we have two types of capsules in the higher layer, one for house and another for boat. Depending upon the orientation of both triangle and rectangle capsules, these capsules will make the following predictions on higher layer capsules. This will give rise to the following scenario:



And as you can see, with respect to its original orientation, the rectangle capsule and the triangle capsule both predicted the boat present in the picture in one of their predictions. They both agree that it's the boat capsule that should be activated in the higher layer capsule. This means that the rectangle and triangle are more a part of a boat than of a house. This also means that the rectangle and triangle capsules think that selecting the boat capsule will explain their own orientation in the primary capsule. In this, both primary layer capsules agree to select the boat capsule in the next layer as a possible object located in the image. This is called *routing by agreement*.

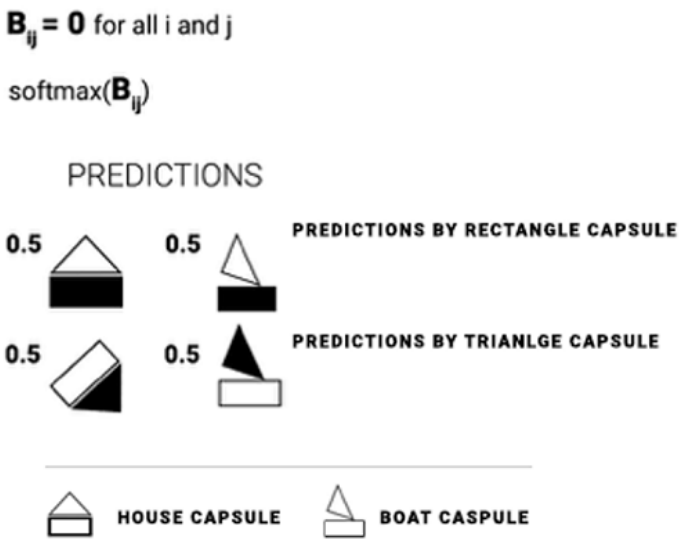


This particular technique has several benefits. Once the primary capsules agree to select a certain higher-level capsule, there is no need to send a signal to another capsule in another higher layer, and the signal in the agreed-on capsule can be made stronger and cleaner and can help in accurately predicting the pose of the object. Another benefit is that if we trace the path of the activation, from the triangle and rectangle to the boat capsule in a higher layer, we can easily sort out the hierarchy of the parts and understand which part belongs to which object; in this particular case, rectangle and triangle belong to the

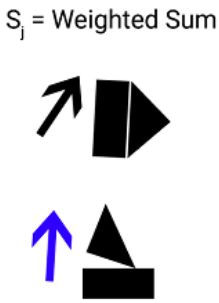
boat object.

So far we have dealt with the primary layer; now the actual work of the higher capsule layer comes in. Even though the primary layer predicted some output for the higher layer, it still needs to calculate its own output and cross-check which prediction matches with its own calculation.

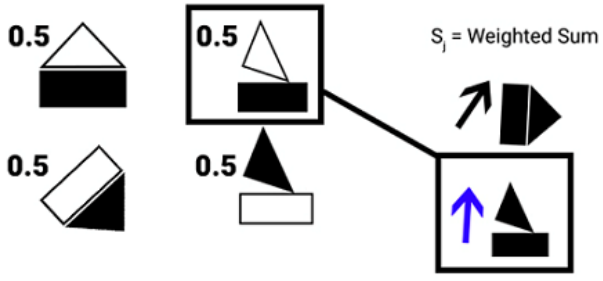
The first step the higher capsule layer takes to calculate its own output is to set up something called *routing weights*. We have some predictions given by our primary layer now for each prediction; at first iteration it declares its routing weights to be zero for all. These initial routing weights are fed into a Softmax function and the output is assigned to each prediction.



Now, after assigning the Softmax output to the predictions, it calculates the weighted sum to each capsule in this higher layer. This gives us two capsules from a bunch of predictions. This is the actual output of the higher layer for the first round or first iteration.



Now we can find which prediction is the most accurate compared to the actual output of the layer.



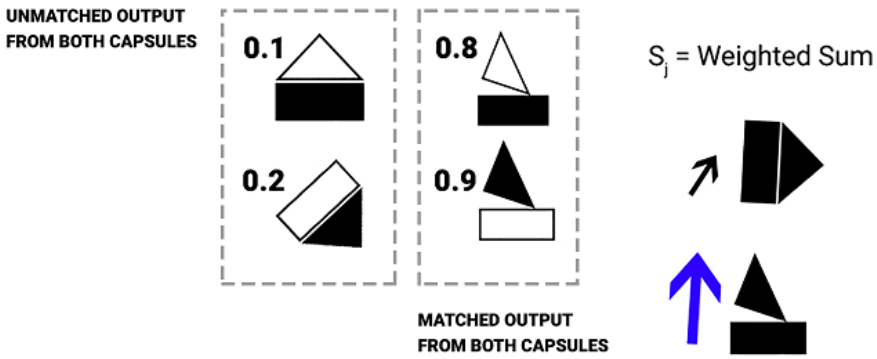
After selecting the accurate prediction, we again calculate another routing weight for the next round by scalar product of the prediction and the actual output of the layer and adding it to the existing routing weight. Given by equation:

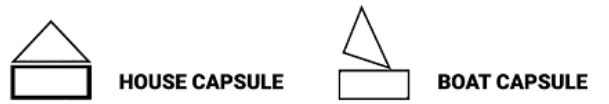
$\hat{U}_{ij}$ (Prediction by primary layer)

V_j (Actual output by the higher layer)

$$B_{ij} \mathrel{+}= \hat{U}_{ij} + V_j$$

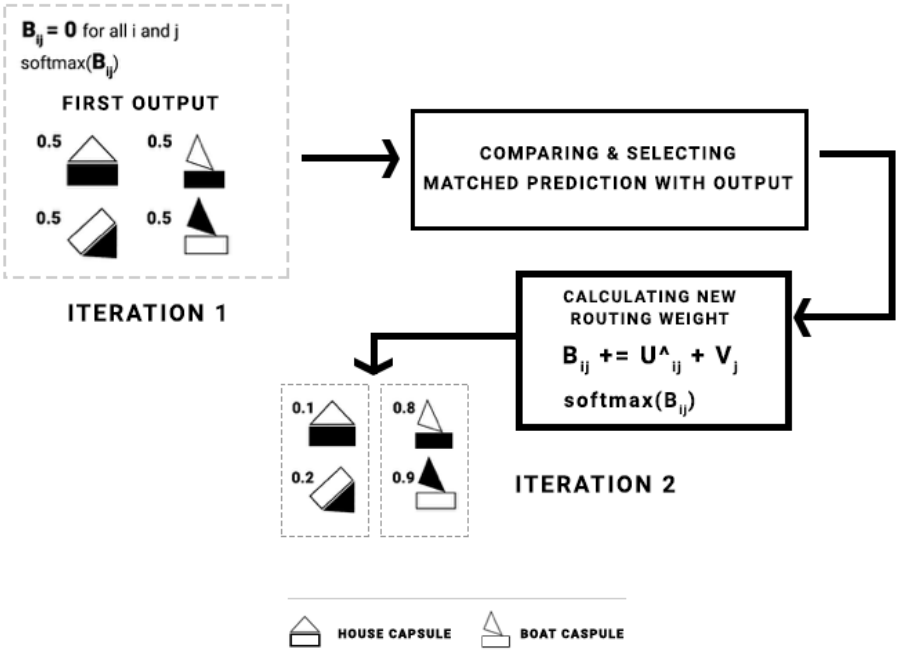
Now, if the prediction and the output match, the new routing weights will be large, and if not, the weight will be low. Again, the routing weights are fed into the Softmax function and the values are assigned to the predictions. You can see that the strong agreed predictions have large weights associated with them, whereas others have low weights.





Again we compute the weighted sum on these predictions with new weights given to it. But now we find that the boat capsule has a longer vector associated with it compared to the house capsule, as the weights were in favor of the boat capsule, so the layer chooses the boat capsule over the house capsule in just two iterations. In this way we can compute the output in this higher layer and choose which capsule to select from this higher layer for the next steps in the capsule network.

I have only described two iterations or rounds for the sake of simplicity, but actually it can take longer, depending upon the task you are performing.



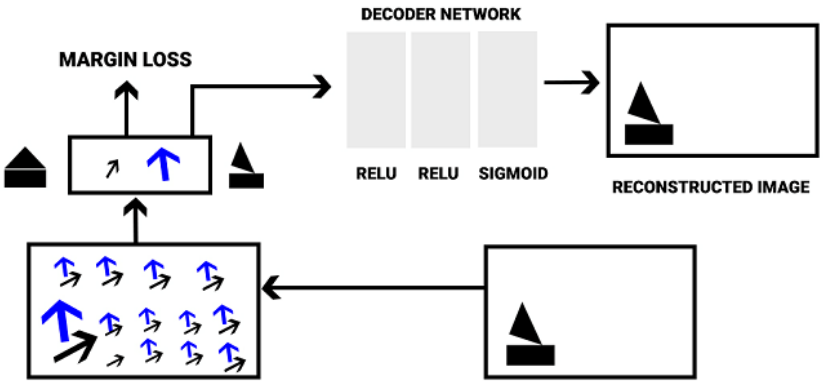
3. Loss calculation

Now that we have made a decision on what object is in the image using the routing by agreement method, you can perform classification. As with our previous higher layer, one capsule per class, that is, one capsule for boat and one for house, we can easily add a layer on top of this higher layer and compute the length of the activation vector and, depending upon the length, we can assign a class probability to make an image classifier.

In the original paper, margin loss was used to calculate the class probability of multiple classes to create such an image classifier. Margin loss simply means that if a certain object of a class is present in the image then the squared length of the corresponding vector of that object's capsule must not be less than 0.9. Similarly, if that object of that class is not present in the image, then the squared length of the corresponding vector of that object should not be more than 0.1.

Suppose $\mathbf{V_k}$ is the length of the output vector of that class **K** object. Now, if that object of class **K** is present then its squared value should not be less than 0.9; that is, $|\mathbf{V_k}|^2 \mathbf{>=0.9}$. Similarly, if the class **K** object is not present then $|\mathbf{V_k}|^2 \mathbf{<=0.1}$.

In addition to margin loss, there is an additional unit called the decoder network connected to the higher capsule layer. This decoder network is three fully connected layers, two of them being rectified linear unit (ReLU) activated units, and the last one the sigmoid activated layer, which is used to reconstruct the input image.



This decoder network learns to reconstruct the input image by minimizing the squared difference between the reconstructed image and the input image:

Reconstruction Loss = (Reconstructed Image – Input Image)^2

Now, we have total loss as:

Total Loss = Margin Loss + alpha * Reconstruction Loss

Here, the value of alpha (a constant to minimize reconstruction loss) in the paper[1] it is 0.0005 (no extra information is given on why this particular value was chosen). Here the reconstruction loss is scaled down considerably so as to give more importance to the margin loss and so it can dominate the training process. The importance of the reconstruction unit and the reconstruction loss is that it forces the network to preserve the information required to reconstruct the image up to the highest capsule layer. This also acts as a regularizer to avoid over-fitting during the training process.

In the paper[1], the capsule network is used to classify between MNIST* digits. As you can see below (in Figure 1), the paper showed different units of CapsNet for MNIST classification. Here, the input after feeding through two convolutional layers is reshaped and squashed to form 32 primary capsules with 6 x 6 x 8 capsules each. These primary capsules are fed into higher layer capsules, a total of 10 capsules with 16 dimensions each, and at last margin loss is calculated on these higher layer capsules to give class probability.

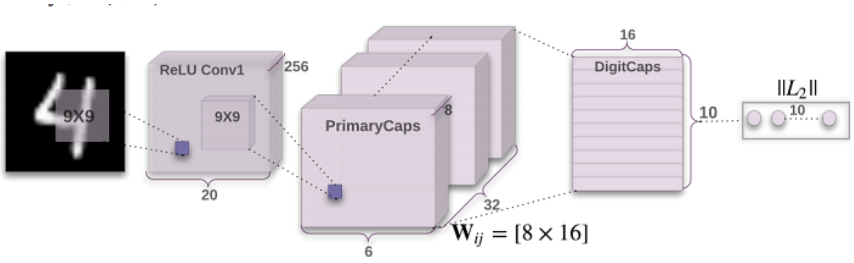


Figure 1: Dynamic routing between capsules[1]

Figure 2 shows the decoder network used to calculate reconstruction loss. A higher layer capsule is connected to three fully connected layers with the last layer being a sigmoid activated layer, which will output 784-pixel intensity values (28 x 28 reconstructed image).

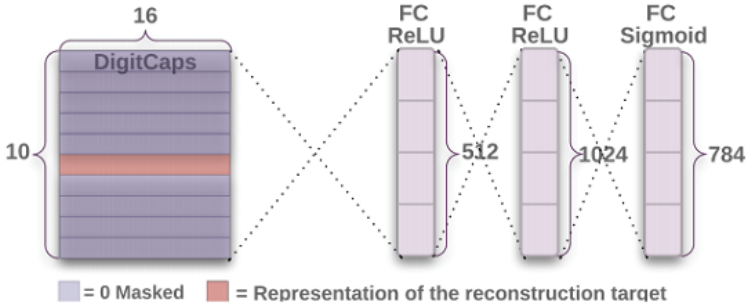


Figure 2: Decoder structure to reconstruct a digit[1]

An interesting thing about this higher capsule layer is that each dimension on this layer is interpretable. That is, if we take the example from the paper on the MNIST dataset, each dimension from the 16-dimension activation vector can be interpreted and signifies certain characteristics of the object. If we modify one of the 16 dimensions we can play with the scale and thickness of the input; similarly, another can represent stroke thickness, another width and translation, and so on.

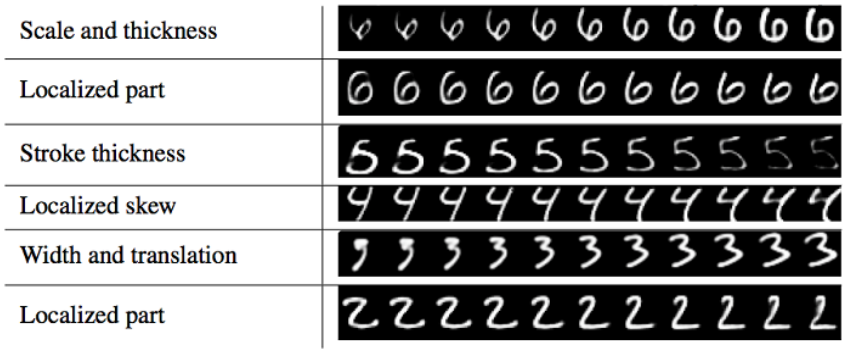


Figure 3: Dimension perturbations[1]

Let's look at how we can implement[3] it using Keras* with TensorFlow* backend. You start by importing all the required libraries:

```
from keras import layers, models, optimizers
from keras.layers import Input, Conv2D, Dense
from keras.layers import Reshape, Layer, Lambda
from keras.models import Model
from keras.utils import to_categorical
from keras import initializers
from keras.optimizers import Adam
from keras.datasets import mnist
from keras import backend as K

import numpy as np
import tensorflow as tf
```

First, let's define the Squash function:

```
def squash(output_vector, axis=-1):
    norm = tf.reduce_sum(tf.square(output_vector), axis,
keep_dims=True)
    return output_vector * norm / ((1 + norm) * tf.sqrt(norm +
1.0e-10))
```

After defining the Squash function, we can define the masking layer:

```
class MaskingLayer(Layer):
    def call(self, inputs, **kwargs):
        input, mask = inputs
        return K.batch_dot(input, mask, 1)

    def compute_output_shape(self, input_shape):
        *_, output_shape = input_shape[0]
        return (None, output_shape)
```

Now, let's define the primary Capsule function:

```
def PrimaryCapsule(n_vector, n_channel, n_kernel_size, n_stride,
padding='valid'):
    def builder(inputs):
        output = Conv2D(filters=n_vector * n_channel,
kernel_size=n_kernel_size, strides=n_stride, padding=padding)(inputs)
        output = Reshape( target_shape=[-1, n_vector],
name='primary_capsule_reshape')(output)
        return Lambda(squash, name='primary_capsule_squash')(output)
    return builder
```

After that, let's write the capsule layer class:

```
class CapsuleLayer(Layer):
    def __init__(self, n_capsule, n_vec, n_routing, **kwargs):
        super(CapsuleLayer, self).__init__(**kwargs)
        self.n_capsule = n_capsule
        self.n_vector = n_vec
        self.n_routing = n_routing
        self.kernel_initializer = initializers.get('he_normal')
        self.bias_initializer = initializers.get('zeros')

    def build(self, input_shape): # input_shape is a 4D tensor
        _, self.input_n_capsule, self.input_n_vector, *_ = input_shape
        self.W = self.add_weight(shape=[self.input_n_capsule,
self.n_capsule, self.input_n_vector, self.n_vector],
initializer=self.kernel_initializer, name='W')
        self.bias = self.add_weight(shape=[1, self.input_n_capsule,
self.n_capsule, 1, 1], initializer=self.bias_initializer, name='bias',
trainable=False)
        self.built = True

    def call(self, inputs, training=None):
        input_expand = tf.expand_dims(tf.expand_dims(inputs, 2), 2)
        input_tiled = tf.tile(input_expand, [1, 1, self.n_capsule, 1,
1])
```

The class below will compute the length of the capsule:

```
class LengthLayer(Layer):
    def call(self, inputs, **kwargs):
        return tf.sqrt(tf.reduce_sum(tf.square(inputs), axis=-1,
keep_dims=False))

    def compute_output_shape(self, input_shape):
        *output_shape, _ = input_shape
        return tuple(output_shape)
```

The function below will compute the margin loss:

```
def margin_loss(y_ground_truth, y_prediction):
    _m_plus = 0.9
    _m_minus = 0.1
    _lambda = 0.5
    L = y_ground_truth * tf.square(tf.maximum(0., _m_plus -
y_prediction)) + _lambda * ( 1 - y_ground_truth) *
tf.square(tf.maximum(0., y_prediction - _m_minus))
    return tf.reduce_mean(tf.reduce_sum(L, axis=1))
```

After defining the different necessary building blocks of the network we can now preprocess the MNIST dataset input for the network:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
y_train = to_categorical(y_train.astype('float32'))
y_test = to_categorical(y_test.astype('float32'))
X = np.concatenate((x_train, x_test), axis=0)
Y = np.concatenate((y_train, y_test), axis=0)
```

Below are some variables that will represent the shape of the input, number of output classes, and number of routings:

```
input_shape = [28, 28, 1]
n_class = 10
n_routing = 3
```

Now, let's create the encoder part of the network:

```
x = Input(shape=input_shape)
conv1 = Conv2D(filters=256, kernel_size=9, strides=1, padding='valid',
```

```
    activation='relu', name='conv1')(x)
3   primary_capsule = PrimaryCapsule( n_vector=8, n_channel=32,
    n_kernel_size=9, n_stride=2)(conv1)
4   digit_capsule = CapsuleLayer( n_capsule=n_class, n_vec=16,
    n_routing=n_routing, name='digit_capsule')(primary_capsule)
5   output_capsule = LengthLayer(name='output_capsule')(digit_capsule)
```

Then let's create the decoder part of the network:

```
1   mask_input = Input(shape=(n_class, ))
2   mask = MaskingLayer()([digit_capsule, mask_input])  # two inputs
3   dec = Dense(512, activation='relu')(mask)
4   dec = Dense(1024, activation='relu')(dec)
5   dec = Dense(784, activation='sigmoid')(dec)
6   dec = Reshape(input_shape)(dec)
```

Now let's create the entire model and compile it:

```
1   model = Model([x, mask_input], [output_capsule, dec])
2   model.compile(optimizer='adam', loss=[ margin_loss, 'mae' ], metrics=[
    margin_loss, 'mae', 'accuracy'])
```

To view the layers and overall architecture of the entire model, we can use this command:

```
model.summary()
```

Finally, we can train the model for three epochs and find out how it will perform:

```
1   model.fit([X, Y], [Y, X], batch_size=128, epochs=3,
    validation_split=0.2)
```

After training the model for only three epochs, the training set output accuracy of the model on the MNIST dataset was 0.9914, and 0.9919 for the validation set, which is 99 percent accurate for both the training and validation sets.

For the above implementation the Intel® AI DevCloud was used to train the network. Intel AI DevCloud is available for academic and personal research purposes for free and the request can be made from here: https://software.intel.com/en-us/ai-academy/tools/devcloud (https://software.intel.com/en-us/ai-academy/tools/devcloud).

In this way, you can implement the capsule network using Keras and TensorFlow backend.

Now let's look at some of the pros and cons of a capsule network.

Pros

1. Requires less training data
2. Equivariance preserves positional information of the input object
3. Routing by agreement is great for overlapping objects
4. Automatically calculates hierarchy of parts in a given object
5. Activation vectors are interpretable
6. Reached high accuracy in MNIST

Cons

1. Results are not state of the art in difficult datasets like CIFAR10*
2. Not tested in larger datasets like ImageNet*
3. Slow training process due to inner loop
4. Problem of crowding—not being able to distinguish between two identical objects of the same type placed close to one another.

References

1. *Dynamic Routing Between Capsules* by Sara Sabour, Nicholas Frosst and Geoffrey E Hinton: https://arxiv.org/pdf/1710.09829.pdf (https://arxiv.org/pdf/1710.09829.pdf)
2. *Capsule Networks (CapsNets) – Tutorial* created by Aurélien Géron: https://www.youtube.com/watch?v=pPN8d0E3900 (https://www.youtube.com/watch?v=pPN8d0E3900)
3. Code used above is adopted from the GitHub* site engwang/minimal-capsule: https://github.com/fengwang/minimal-capsule (https://github.com/fengwang/minimal-capsule)

More implementations of CapsNet in popular frameworks:

- Keras + TensorFlow: https://github.com/XifengGuo/CapsNet-Keras (https://github.com/XifengGuo/CapsNet-Keras)
- TensorFlow: https://github.com/naturomics/CapsNet-Tensorflow (https://github.com/naturomics/CapsNet-Tensorflow)
- PyTorch*: https://github.com/gram-ai/capsule-networks (https://github.com/gram-ai/capsule-networks)
- TensorFlow Implementation in Jupyter Notebook*: https://github.com/ageron/handson-ml/blob/master/extra_capsnets.ipynb (https://github.com/ageron/handson-ml/blob/master/extra_capsnets.ipynb)