



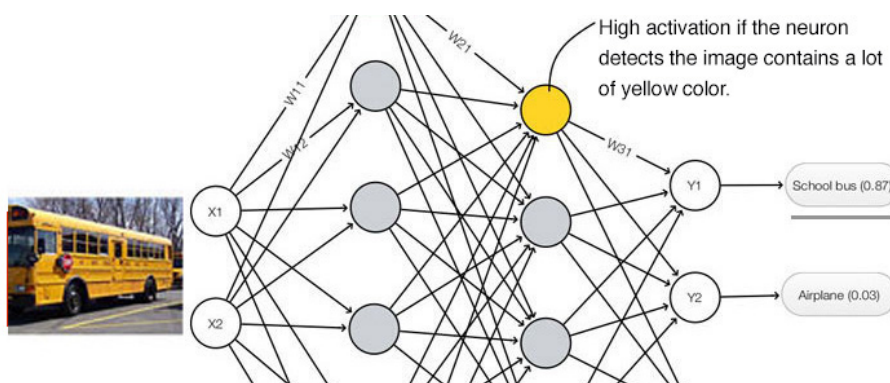
# “Understanding Dynamic Routing between Capsules (Capsule Networks)”

Nov 3, 2017

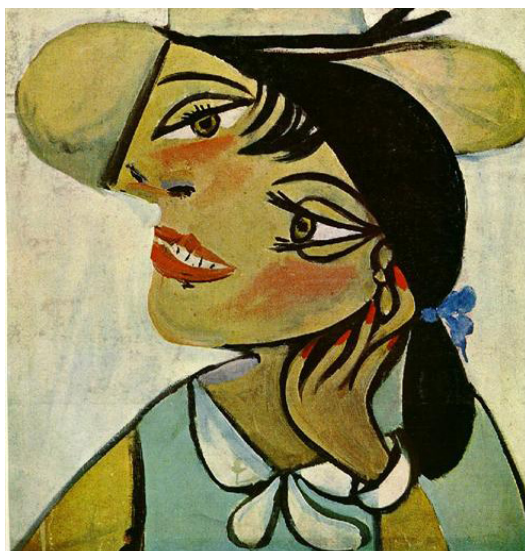
This article covers the technical paper by Sara Sabour, Nicholas Frosst and Geoffrey Hinton on [Dynamic Routing between Capsules](#). The source code implementation is originated from [XifengGuo](#) using Keras with Tensorflow. In this article, we will describe the basic concept first and later apply it with the Capsule network *CapsNet* to detect digits in MNist.

## CNN challenges

In deep learning, the activation level of a neuron is often interpreted as the likelihood of detecting a specific feature.



If we pass the Picasso's "Portrait of woman in d'hermine pass" into a CNN classifier, how likely that the classifier may mistaken it as a real human face?

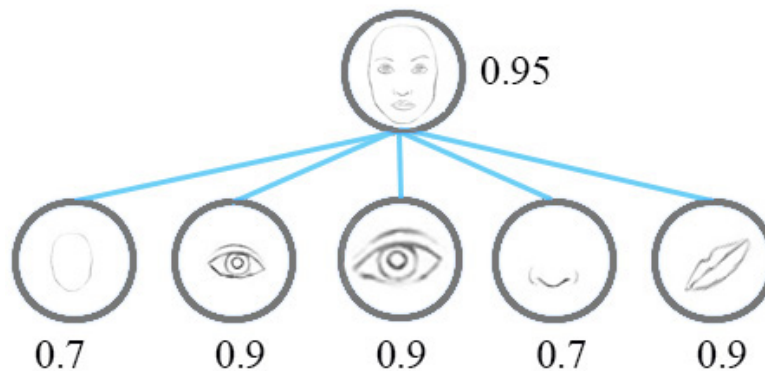


CNN is good at detecting features but less effective at exploring the spatial relationships among features (perspective, size, orientation). For example, the following picture may fool a *simple* CNN model in believing that this a good sketch of a human face.

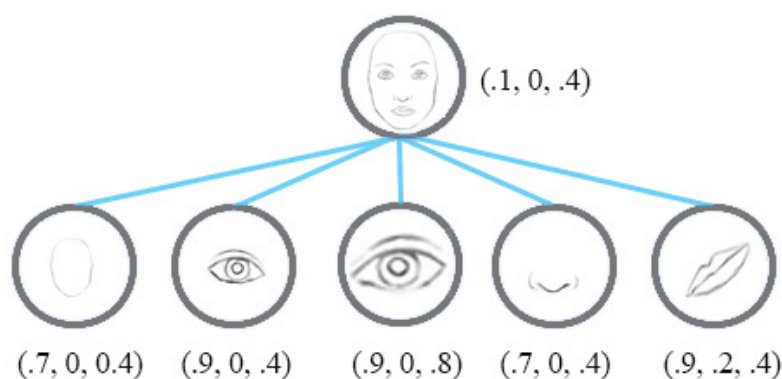


[\(image source\)](#)

A simple CNN model can extract the features for nose, eyes and mouth correctly but will wrongly activate the neuron for the face detection. Without realize the mis-match in spatial orientation and size, the activation for the face detection will be too high.



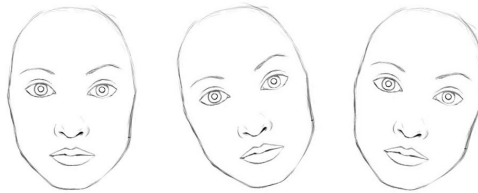
Now, we imagine that each neuron contains the likelihood as well as properties of the features. For example, it outputs a vector containing [likelihood, orientation, size]. With this spatial information, we can detect the in-consistence in the orientation and size among the nose, eyes and ear features and therefore output a much lower activation for the face detection.



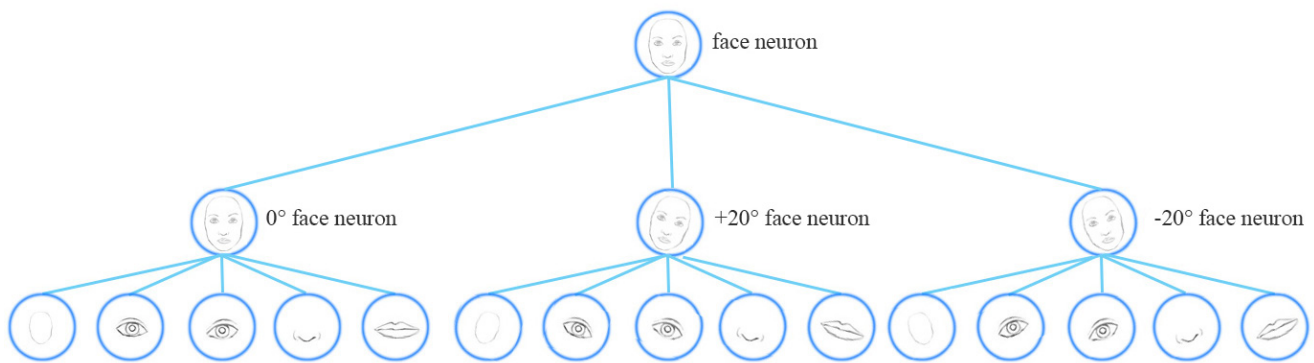
Instead of using the term neurons, the technical paper uses the term **capsules** to indicate that capsules output a vector instead of a single scalar value.

## Viewpoint invariant

Let's examine how a CNN can handle viewpoint variants. For example, how to train a face detection neuron for different orientations.



Conceptually, we can say the CNN trains neurons to handle different orientations with a final top level face detection neuron.

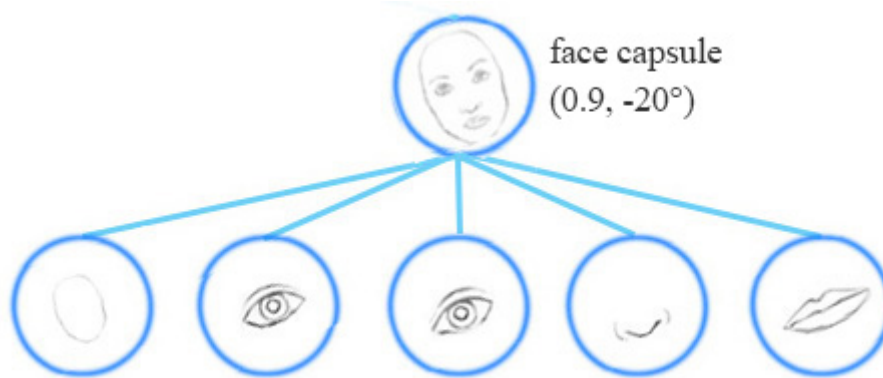


As noted above, for a CNN to handle viewpoint or style variants, we add more convolution layers and features maps. Nevertheless this approach tends to memorize the dataset rather than generalize a solution. It requires a large volume of training data to cover different variants and to avoid overfitting. MNIST dataset contains 55,000 training data. i.e. 5,500 samples per digits. However, it is unlikely that children need to read this large amount of samples to learn digits. Our existing deep learning models including CNN seem inefficient in utilizing datapoints. Here is an ironic quote from Geoffrey Hinton:

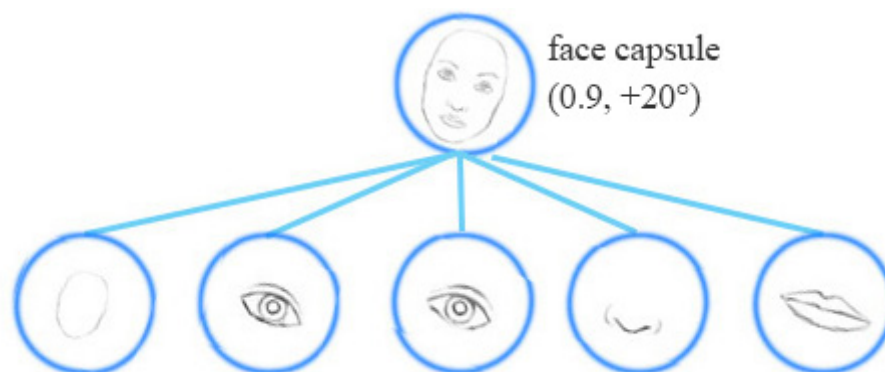
*It (convolutional network) works depressingly well.*

## Equivariance vs invariance

Instead of capture a feature with a specific variant, a capsule is trained to capture the likeliness of a feature and its variant. So the purpose of the capsule is not only to detect a feature but also to train the model to learn the variant.



Such that the same capsule can detect the same object class with different orientations (for example, rotate clockwise):



**Invariance** is the detection of features regardless of the variants. For example, a nose-detection neuron detects a nose regardless of the orientation. However, the loss of spatial orientation in a neuron will eventually hurt the effectiveness of such invariance model.

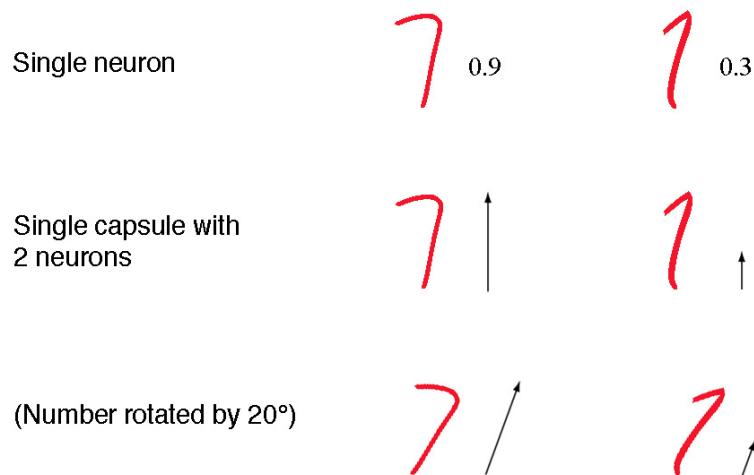
**Equivariance** is the detection of objects that can transform to each other (for example, detecting faces with different orientations). Intuitively, the capsule network detects the face is rotated right 20° (equivariance) rather than realizes the face matched a variant that is rotated 20°. By forcing the model to learn the feature variant in a capsule, we *may* extrapolate possible variants more effectively with less training data. In additional, we may reject adversaries more effectively.

*With feature property as part of the information extracted by capsules, we may generalize the model better without an over extensive amount of labeled data.*

## Capsule

*A capsule is a group of neurons that not only capture the likelihood but also the parameters of the specific feature.*

For example, the first row below indicates the probabilities of detecting the number “7” by a neuron. A 2-D capsule is formed by combining 2 neurons. This capsule outputs a 2-D vector in detecting the number “7”. For the first image in the second row, it outputs a vector  $v = (0, 0.9)$ . The magnitude of the vector  $\|v\| = \sqrt{0^2 + 0.9^2} = 0.9$  corresponds to the probability of detecting “7”. The second image of each row looks more like a “1” than a “7”. Therefore its corresponding likelihood as “7” is smaller (smaller scalar value or smaller vector’s magnitude but with the same orientation) .



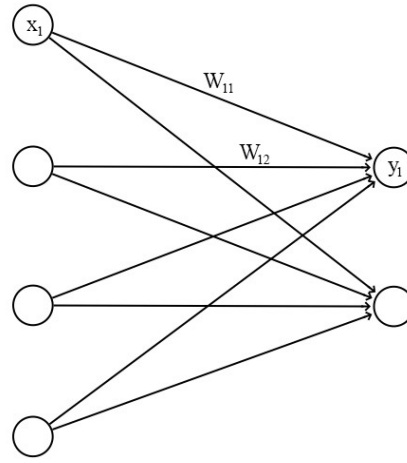
In the third row, we rotate the image by 20°. The capsule will generate vectors with the same magnitude but different orientations. Here, the angle of the vector represents the angle of rotation for the number “7”. As we can image, we can add 2 more neurons to a capsule to capture the size and stroke width.



*We call the output vector of a capsule as the **activity vector** with magnitude represents the probability of detecting a feature and its orientation represents its parameters (properties).*

## Compute the output of a capsule

Recall a fully connected neural network:



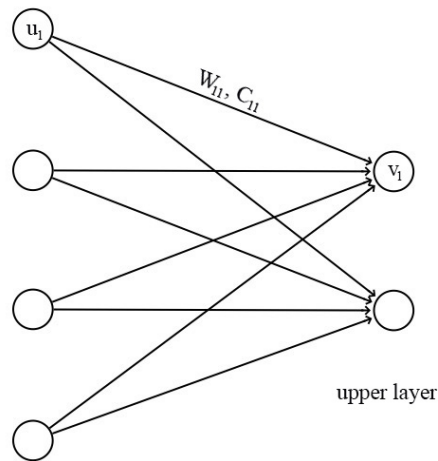
The output of each neuron is computed from the output of the neurons from the previous layer:

$$z_j = \sum_i W_{ij} x_i$$

$$y_j = \text{ReLU}(z_j)$$

which  $W_{ij}$ ,  $z_j$  and  $y_i$  are all scalars.

For a capsule, the input  $u_i$  and the output  $v_j$  of a capsule are vectors.



We apply a **transformation matrix**  $W_{ij}$  to the capsule output  $u_i$  of the pervious layer. For example, with a  $m \times k$  matrix, we transform a k-D  $u_i$  to a m-D  $\hat{u}_{j|i}$ . ( $(m \times k) \times (k \times 1) \implies m \times 1$ ) Then we compute a weighted sum  $s_j$  with weights  $c_{ij}$ .

$$\hat{u}_{j|i} = W_{ij} u_i$$

$$s_j = \sum_i c_{ij} \hat{u}_{j|i}$$

$c_{ij}$  are **coupling coefficients** that are trained by the iterative dynamic routing process (discussed next) and  $\sum_j c_{ij}$  are designed to sum to one.

Instead of applying a ReLU function, we apply a squashing function to scale the vector between 0 and unit length.

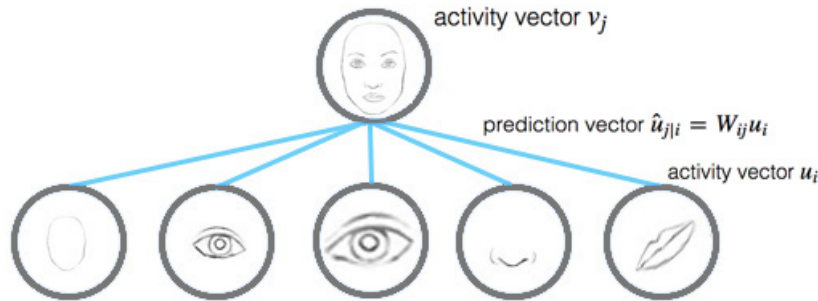
$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|}$$

It shrinks small vectors to zero and long vectors to unit vectors. Therefore the likelihood of each capsule is bounded between zero and one.

$$\begin{aligned} v_j &\approx \|s_j\| s_j && \text{for } s_j \text{ is short} \\ v_j &\approx \frac{s_j}{\|s_j\|} && \text{for } s_j \text{ is long} \end{aligned}$$

## Iterative dynamic Routing

In deep learning, we use backpropagation to train model parameters. The transformation matrix  $W_{ij}$  in capsules are still trained with backpropagation. Nevertheless, the coupling coefficients  $c_{ij}$  are calculated with a new iterative dynamic routing method.



The **prediction vector**  $\hat{u}_{j|i}$  is computed as (with the transformation matrix):

$$\hat{u}_{j|i} = W_{ij}u_i$$

which  $u_i$  is the activity vector for the capsule  $i$  in the layer below.

The **activity vector**  $v_j$  for the capsule  $j$  in the layer above is computed as:

$$\begin{aligned} s_j &= \sum_i c_{ij} \hat{u}_{j|i} \\ v_j &= \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \end{aligned}$$



Intuitively, prediction vector  $\hat{u}_{j|i}$  is the prediction (**vote**) from the capsule  $i$  on the output of the capsule  $j$  above. If the activity vector has close similarity with the prediction vector, we conclude that capsule  $i$  is highly related with the capsule  $j$ . (For example, the eye capsule is highly related to the face capsule.) Such similarity is measured using the scalar product of the prediction and activity vector. Therefore, the similarity takes into account on both likeliness and the feature properties. (instead of just likeliness in neurons) We compute a relevancy score  $b_{ij}$  according to the similarity:

$$b_{ij} \leftarrow \hat{u}_{j|i} \cdot v_j$$

The coupling coefficients  $c_{ij}$  is computed as the softmax of  $b_{ij}$ :

$$c_{ij} = \frac{\exp b_{ij}}{\sum_k \exp b_{ik}}$$

To make  $b_{ij}$  more accurate, it is updated iteratively in multiple iterations (typically in 3 iterations).

$$b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i} \cdot v_j$$

Here is the final pseudo code for the dynamic routing:

---

**Procedure 1** Routing algorithm.

---

```

1: procedure ROUTING( $\hat{u}_{j|i}, r, l$ )
2:   for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow 0$ .
3:   for  $r$  iterations do
4:     for all capsule  $i$  in layer  $l$ :  $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$ 
5:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{u}_{j|i}$ 
6:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$ 
7:     for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i} \cdot \mathbf{v}_j$ 
   return  $\mathbf{v}_j$ 

```

---

Source Sara Sabour, Nicholas Frosst, Geoffrey Hinton

*Routing a capsule to the capsule in the layer above based on relevancy is called **Routing-by-agreement**.*

## Max pooling shortcoming

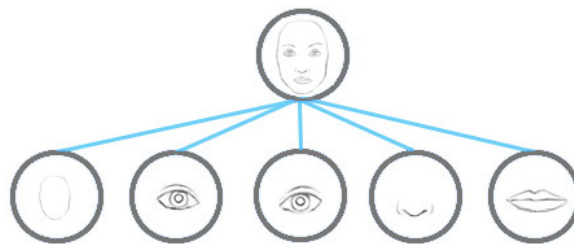
The max pooling in a CNN handles translational variance. Even a feature is slightly moved, if it is still within the pooling window, it can still be detected. Nevertheless, this approach keeps only the max feature (the most dominating) and throws away the others. Capsules maintain a weighted sum of features from the previous layer. Hence, it is more suitable in detecting overlapping features. For example detecting multiple overlapping digits in the handwriting:



## Significant of routing-by-agreement with capsules

In deep learning, we use backpropagation to train the model's parameters based on a cost function. Those parameters (weights) control how signal is routed from one layer to another. If the weight between 2 neurons is zero, the activation of a neuron is not propagated to that neuron.

Iterative dynamic routing provides an alternative of how signal is routed based on feature parameters rather than one size fit all cost function. By utilizing the feature parameters, we can theoretically group capsules better to form a high level structure. For example, the capsule layers may eventually behaves as a **parse tree** that explore the part-whole relationship. (for example, a face is composed of eyes, a nose and a mouth) The iterative dynamic routing controls how much a signal is propagate upward to the capsules above utilizing the transformation matrix, the likeliness and the feature's properties.

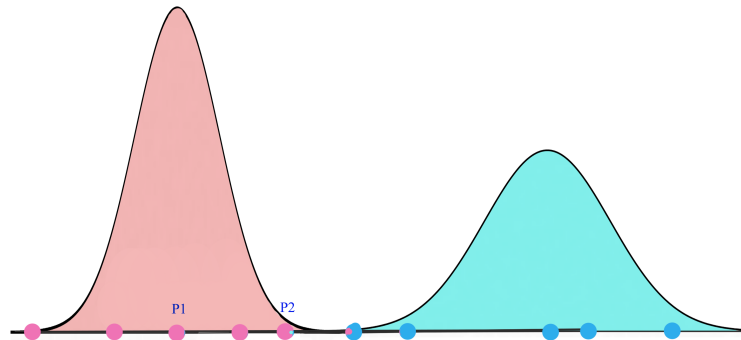


The iterative dynamic routing with capsules is just one showcase in demonstrating the routing-by-agreement. We expect more capsule models with advance routing methods will be introduced in coming years. In a second paper on capsules *Matrix capsules with EM routing*, a [likeliness, 4x4 pose matrix] matrix capsule is proposed (rather than a k-D vector capsule) with a new Expectation-maximization routing (EM routing). The pose matrices are designed to capture the viewpoint of the object. For example, the second row of the picture below represent the same object above with a different pose matrix (viewpoint).



(Source from the paper Matrix capsules with EM routing)

The objective of the EM routing is to group capsules to form a part-whole relationship with a clustering technique (EM). In machine learning, we use EM to cluster datapoints into different Gaussian distributions. For example, we cluster the datapoints below into two clusters modeled by two gaussian distributions.



For the mouth, eyes and nose capsules in the lower layer, each of them makes predictions (votes) on the pose matrices of its possible parent capsule(s) through a transformation matrix. The role of the EM routing is to cluster lower level capsules that produce similar votes. Conceptually the votes from the mouth, eyes and nose capsules may cluster into the pink region above that can be represented by their parent capsule (face).

*A higher level feature (a face) is detected by looking for agreement between votes from the capsules one layer below. We use EM routing to cluster capsules that have close proximity of the corresponding votes.*

The transformation matrix to compute the vote is viewpoint invariant. We do not need different transformation matrices for different viewpoints. Even the viewpoint may change, the pose matrices (or votes) corresponding to the same high level structure (a face) will change in a co-ordinate way such that a cluster with the same capsules can be detected.



Hence, the EM routing groups related capsules regardless of the viewpoint. Unlike CNN which each neuron may detect a different viewpoint, the transformation matrix is viewpoint independent which may require less data to train.

*New capsules and routing algorithm will hopefully build higher level structures much easier and much effectively with less training data.*

(Note: The second paper expands what a Capsule network can do. For those interested, here is my other article on the [Matrix capsule](#)).

## CapsNet architecture

Finally, we apply capsules to build the CapsNet to classify the MNist digits. The following is the architecture using CapsNet.

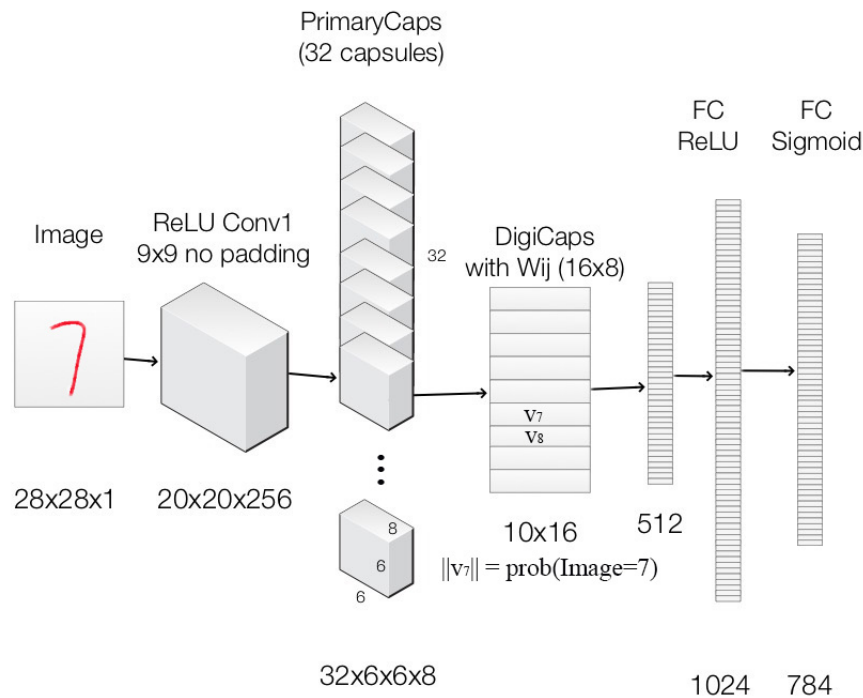


Image is feed into the ReLU Conv1 which is a standard convolution layer. It applies 256 9x9 kernels to generate an output with 256 channels (feature maps). With stride 1 and no padding, the spatial dimension is reduced to 20x20. (  $28-9+1=20$  )

It is then feed into PrimaryCapsules which is a modified convolution layer supporting capsules. It generates a 8-D vector instead of a scalar. PrimaryCapsules used 8x32 kernels to generate 32 8-D capsules. (i.e. 8 output neurons are grouped together to form a capsule) PrimaryCapsules uses 9x9 kernels with stride 2 and no padding to reduce the spatial dimension from 20x20 to 6x6 (  $\lfloor \frac{20-9}{2} \rfloor + 1 = 6$  ). In PrimaryCapsules, we have 32x6x6 capsules.

It is then feed into DigiCaps which apply a transformation matrix  $W_{ij}$  with shape 16x8 to convert the 8-D capsule to a 16-D capsule for each class  $j$  (from 1 to 10).

$$\hat{u}_{j|i} = W_{ij}u_i$$

The final output  $v_j$  for class  $j$  is computed as:

$$s_j = \sum_i c_{ij} \hat{u}_{j|i}$$

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|}$$

Because there are 10 classes, the shape of DigiCaps is 10x16 (10 16-D vector.) Each vector  $v_j$  acts as the capsule for class  $j$ . The probability of the image to be classify as  $j$  is computed by  $\|v_j\|$ . In our example, the true label is 7 and  $v_7$  is the latent representation of our input. With a 2 hidden fully connected layers, we can reconstruct the 28x28 image from  $v_7$ .

Here is the summary of each layers:

Layer Name	Apply	Output shape
Image	Raw image array	28x28x1
ReLU Conv1	Convolution layer with 9x9 kernels output 256 channels, stride 1, no padding with ReLU	20x20x256
PrimaryCapsules	Convolution capsule layer with 9x9 kernel output 32x6x6 8-D capsule, stride 2, no padding	6x6x32x8
DigiCaps	Capsule output computed from a $W_{ij}$ (16x8 matrix) between $u_i$ and $v_j$ ( $i$ from 1 to 32x6x6 and $j$ from 1 to 10).	10x16
FC1	Fully connected with ReLU	512
FC2	Fully connected with ReLU	1024
Output image	Fully connected with sigmoid	784 (28x28)

*Our capsule layers use convolution kernels to explore locality information.*

## Loss function (Margin loss)

In our example, we want to detect multiple digits in a picture. Capsules use a separate margin loss  $L_c$  for each category  $c$  digit present in the picture:

$$L_c = T_c \max(0, m^+ - \|v_c\|)^2 + \lambda(1 - T_c) \max(0, \|v_c\| - m^-)^2$$

which  $T_c = 1$  if an object of class  $c$  is present.  $m^+ = 0.9$  and  $m^- = 0.1$ . The  $\lambda$  down-weighting (default 0.5) stops the initial learning from shrinking the activity vectors of all classes. The total loss is just the sum of the losses of all classes.

Computing the margin loss in Keras

```
def margin_loss(y_true, y_pred):
    """
    :param y_true: [None, n_classes]
    :param y_pred: [None, num_capsule]
    :return: a scalar loss value.
    """
    L = y_true * K.square(K.maximum(0., 0.9 - y_pred)) + \
        0.5 * (1 - y_true) * K.square(K.maximum(0., y_pred - 0.1))

    return K.mean(K.sum(L, 1))
```

## CapsNet model

Here is the Keras code in creating the CapsNet model:

```
def CapsNet(input_shape, n_class, num_routing):
    """
    :param input_shape: (None, width, height, channels)
    :param n_class: number of classes
    :param num_routing: number of routing iterations
    :return: A Keras Model with 2 inputs (image, label) and
             2 outputs (capsule output and reconstruct image)
    """
    # Image
    x = layers.Input(shape=input_shape)

    # ReLU Conv1
    conv1 = layers.Conv2D(filters=256, kernel_size=9, strides=1,
                          padding='valid', activation='relu', name='conv1')(x)

    # PrimaryCapsules: Conv2D layer with `squash` activation,
    # reshape to [None, num_capsule, dim_vector]
    primarycaps = PrimaryCap(conv1, dim_vector=8, n_channels=32,
                             kernel_size=9, strides=2, padding='valid')

    # DigiCap: Capsule layer. Routing algorithm works here.
    digitcaps = DigiCaps(num_capsule=n_class, dim_vector=16,
```

```

num_routing=num_routing, name='digitcaps')(primarycaps)

# The length of the capsule's output vector
out_caps = Length(name='out_caps')(digitcaps)

# Decoder network.
y = layers.Input(shape=(n_class,))

# The true label is used to extract the corresponding v_j
masked = Mask()([digitcaps, y])
x_recon = layers.Dense(512, activation='relu')(masked)
x_recon = layers.Dense(1024, activation='relu')(x_recon)
x_recon = layers.Dense(784, activation='sigmoid')(x_recon)
x_recon = layers.Reshape(target_shape=[28, 28, 1], name='out_recon')(x_recon)

# two-input-two-output keras Model
return models.Model([x, y], [out_caps, x_recon])

```

The length of the capsule's output vector  $\|v_j\|$  corresponds to the probability that it belong to the class  $j$ . For example,  $\|v_7\|$  is the probability of the input image belongs to 7.

```

class Length(layers.Layer):
    def call(self, inputs, **kwargs):
        # L2 length which is the square root
        # of the sum of square of the capsule element
        return K.sqrt(K.sum(K.square(inputs), -1))

```

## PrimaryCapsules

PrimaryCapsules converts 20x20 256 channels into 32x6x6 8-D capsules.

```

def PrimaryCap(inputs, dim_vector, n_channels, kernel_size, strides, padding):
    """
    Apply Conv2D `n_channels` times and concatenate all capsules
    :param inputs: 4D tensor, shape=[None, width, height, channels]
    :param dim_vector: the dim of the output vector of capsule
    :param n_channels: the number of types of capsules
    :return: output tensor, shape=[None, num_capsule, dim_vector]
    """
    output = layers.Conv2D(filters=dim_vector*n_channels, kernel_size=kernel_size, str
    outputs = layers.Reshape(target_shape=[-1, dim_vector])(output)
    return layers.Lambda(squash)(outputs)

```

## Squash function

Squash function behaves like a sigmoid function to squash a vector such that its length falls between 0 and 1.

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|}$$

```
def squash(vectors, axis=-1):  
    """  
    The non-linear activation used in Capsule. It drives the length of a large vector  
    :param vectors: some vectors to be squashed, N-dim tensor  
    :param axis: the axis to squash  
    :return: a Tensor with same shape as input vectors  
    """  
    s_squared_norm = K.sum(K.square(vectors), axis, keepdims=True)  
    scale = s_squared_norm / (1 + s_squared_norm) / K.sqrt(s_squared_norm)  
    return scale * vectors
```

## DigiCaps with dynamic routing

DigiCaps converts the capsules in PrimaryCapsules to 10 capsules each making a prediction for class  $j$ . The following is the code in creating 10 (n\_class) 16-D (dim\_vector) capsules:

```
# num_routing is default to 3  
digitcaps = DigiCap(num_capsule=n_class, dim_vector=16,  
                    num_routing=num_routing, name='digitcaps')(primarycaps)
```

DigiCap is just a simple extension of a dense layer. Instead of taking a scalar and output a scalar, it takes a vector and output a vector:

- input shape = (None, input\_num\_capsule (32), input\_dim\_vector(8) )
- output shape = (None, num\_capsule (10), dim\_vector(16) )

Here is the DigiCaps and we will detail some part of the code for explanation later.

```
class DigiCap(layers.Layer):  
    """  
    The capsule layer.  
  
    :param num_capsule: number of capsules in this layer  
    :param dim_vector: dimension of the output vectors of the capsules in this layer
```



```

:param num_routings: number of iterations for the routing algorithm
"""

def __init__(self, num_capsule, dim_vector, num_routing=3,
              kernel_initializer='glorot_uniform',
              b_initializer='zeros',
              **kwargs):
    super(DigiCap, self).__init__(**kwargs)
    self.num_capsule = num_capsule      # 10
    self.dim_vector = dim_vector        # 16
    self.num_routing = num_routing      # 3
    self.kernel_initializer = initializers.get(kernel_initializer)
    self.b_initializer = initializers.get(b_initializer)

def build(self, input_shape):
    """The input Tensor should have shape=[None, input_num_capsule, input_dim_vector]
    assert len(input_shape) >= 3,
    self.input_num_capsule = input_shape[1]
    self.input_dim_vector = input_shape[2]

    # Transform matrix W
    self.W = self.add_weight(shape=[self.input_num_capsule, self.num_capsule,
                                     self.input_dim_vector, self.dim_vector],
                              initializer=self.kernel_initializer,
                              name='W')

    # Coupling coefficient.
    # The redundant dimensions are just to facilitate subsequent matrix calculation
    self.b = self.add_weight(shape=[1, self.input_num_capsule, self.num_capsule, 1],
                              initializer=self.b_initializer,
                              name='b',
                              trainable=False)

    self.built = True

def call(self, inputs, training=None):
    # inputs.shape = (None, input_num_capsule, input_dim_vector)
    # Expand dims to (None, input_num_capsule, 1, 1, input_dim_vector)
    inputs_expand = K.expand_dims(K.expand_dims(inputs, 2), 2)

    # Replicate num_capsule dimension to prepare being multiplied by W
    # Now shape = [None, input_num_capsule, num_capsule, 1, input_dim_vector]
    inputs_tiled = K.tile(inputs_expand, [1, 1, self.num_capsule, 1, 1])

    # Compute `inputs * W` by scanning inputs_tiled on dimension 0.
    # inputs_hat.shape = [None, input_num_capsule, num_capsule, 1, dim_vector]
    inputs_hat = tf.scan(lambda ac, x: K.batch_dot(x, self.W, [3, 2]),
                          elems=inputs_tiled,
                          initializer=K.zeros([self.input_num_capsule, self.num_capsule, self.dim_vector]))

```

```

# Routing algorithm
assert self.num_routing > 0, 'The num_routing should be > 0.'
for i in range(self.num_routing):
    c = tf.nn.softmax(self.b, dim=2) # dim=2 is the num_capsule dimension
    # outputs.shape=[None, 1, num_capsule, 1, dim_vector]
    outputs = squash(K.sum(c * inputs_hat, 1, keepdims=True))

    # last iteration needs not compute b which will not be passed to the graph
    if i != self.num_routing - 1:
        self.b += K.sum(inputs_hat * outputs, -1, keepdims=True)
return K.reshape(outputs, [-1, self.num_capsule, self.dim_vector])

```

*build* declares the *self.W* parameters representing the transform matrix *W* and *self.b* representing the  $b_{ij}$ .

```

def build(self, input_shape):
    "The input Tensor should have shape=[None, input_num_capsule, input_dim_vector]"
    assert len(input_shape) >= 3,
    self.input_num_capsule = input_shape[1]
    self.input_dim_vector = input_shape[2]

    # Transform matrix W
    self.W = self.add_weight(shape=[self.input_num_capsule, self.num_capsule,
                                    self.input_dim_vector, self.dim_vector],
                              initializer=self.kernel_initializer,
                              name='W')

    # Coupling coefficient.
    # The redundant dimensions are just to facilitate subsequent matrix calculation
    self.b = self.add_weight(shape=[1, self.input_num_capsule, self.num_capsule, 1],
                              initializer=self.b_initializer,
                              name='b',
                              trainable=False)

    self.built = True

```

To compute:

$$\hat{u}_{j|i} = W_{ij}u_i$$

The code first expand the dimension of  $u_i$  and then multiple it with  $w$ . Nevertheless, the simple dot product implementation of  $W_{ij}u_i$  (commet out below) is replaced by *tf.scan* for better speed performance.

```

class DigiCap(layers.Layer):
    ...

    def call(self, inputs, training=None):
        # inputs.shape = (None, input_num_capsule, input_dim_vector)
        # Expand dims to (None, input_num_capsule, 1, 1, input_dim_vector)
        inputs_expand = K.expand_dims(K.expand_dims(inputs, 2), 2)

        # Replicate num_capsule dimension to prepare being multiplied by W
        # Now shape = [None, input_num_capsule, num_capsule, 1, input_dim_vector]
        inputs_tiled = K.tile(inputs_expand, [1, 1, self.num_capsule, 1, 1])

        """
        # Compute `inputs * W`
        # By expanding the first dim of W.
        # W has shape (batch_size, input_num_capsule, num_capsule, input_dim_vector, d
        w_tiled = K.tile(K.expand_dims(self.W, 0), [self.batch_size, 1, 1, 1, 1])

        # Transformed vectors,
        inputs_hat.shape = (None, input_num_capsule, num_capsule, 1, dim_vector)
        inputs_hat = K.batch_dot(inputs_tiled, w_tiled, [4, 3])
        """

        # However, we will implement the same code with a faster implementation using
        # Compute `inputs * W` by scanning inputs_tiled on dimension 0.
        # inputs_hat.shape = [None, input_num_capsule, num_capsule, 1, dim_vector]
        inputs_hat = tf.scan(lambda ac, x: K.batch_dot(x, self.W, [3, 2]),
                              elems=inputs_tiled,
                              initializer=K.zeros([self.input_num_capsule, self.num_cap

```

Here is the code to implement the following Iterative dynamic Routing pseudo code.

---

**Procedure 1** Routing algorithm.

---

```

1: procedure ROUTING( $\hat{\mathbf{u}}_{j|i}, r, l$ )
2:   for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow 0$ .
3:   for  $r$  iterations do
4:     for all capsule  $i$  in layer  $l$ :  $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$ 
5:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$ 
6:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$ 
7:     for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$ 
   return  $\mathbf{v}_j$ 

```

---

```

class DigiCap(layers.Layer):
    ...
    def call(self, inputs, training=None):
        ...

```

```

# Routing algorithm
assert self.num_routing > 0, 'The num_routing should be > 0.'

for i in range(self.num_routing): # Default: loop 3 times
    c = tf.nn.softmax(self.b, dim=2) # dim=2 is the num_capsule dimension

    # outputs.shape=[None, 1, num_capsule, 1, dim_vector]
    outputs = squash(K.sum(c * inputs_hat, 1, keepdims=True))

    # last iteration needs not compute b which will not be passed to the graph
    if i != self.num_routing - 1:
        self.b += K.sum(inputs_hat * outputs, -1, keepdims=True)
    return K.reshape(outputs, [-1, self.num_capsule, self.dim_vector])

```

## Image reconstruction

We use the true label to select  $v_j$  to reconstruct the image during training. Then we feed  $v_j$  through 3 fully connected layers to re-generate the original image.

Select  $v_j$  in training with Mask

```

class Mask(layers.Layer):
    """
    Mask a Tensor with shape=[None, d1, d2] by the max value in axis=1.
    Output shape: [None, d2]
    """
    def call(self, inputs, **kwargs):
        # use true label to select target capsule, shape=[batch_size, num_capsule]
        if type(inputs) is list: # true label is provided with shape = [batch_size, num_capsule]
            assert len(inputs) == 2
            inputs, mask = inputs
        else: # if no true label, mask by the max length of vectors of capsules
            x = inputs
            # Enlarge the range of values in x to make max(new_x)=1 and others < 0
            x = (x - K.max(x, 1, True)) / K.epsilon() + 1
            mask = K.clip(x, 0, 1) # the max value in x clipped to 1 and other to 0

        # masked inputs, shape = [batch_size, dim_vector]
        inputs_masked = K.batch_dot(inputs, mask, [1, 1])
        return inputs_masked


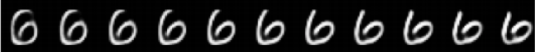

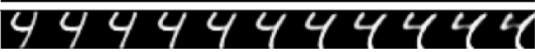

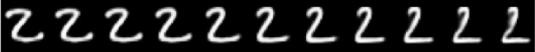
```

## Reconstruction loss

A reconstruction loss  $\| \text{image} - \text{reconstructed image} \|$  is added to the loss function. It trains the network to capture the critical properties into the capsule. However, the reconstruction loss is multiple by a regularization factor (0.0005) so it does not dominate over the marginal loss.

## What capsule is learning?

Each capsule in DigiCaps is a 16-D vector. By slightly varying one dimension by holding other constant, we can learn what property for each dimension is capturing. Each row below is the reconstructed image (using the decoder) of changing only one dimension.

Scale and thickness	
Localized part	
Stroke thickness	
Localized skew	
Width and translation	
Localized part	

Source [Sara Sabour](#), [Nicholas Frosst](#), [Geoffrey Hinton](#)

25 Comments

jhui

1 Login

Recommend

14

Share

Sort by Best

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS


D

f

t

G

Name



Michael Tetelman

• 2 months ago

It looks suspiciously similar to angle - action variables representation in classical mechanics.

1


^

|

v

• Reply

• Share



Jonathan Hui

Mod

→ Michael Tetelman

• a month ago

Thanks for your follow up comments. It makes sense. There are many efforts to learn the latent representation as a compose of invariant property (like class) and variant property (like style).


^

|

v

• Reply

• Share



Jonathan Hui

Mod

→ Michael Tetelman

• 2 months ago

A capsule can capture non-spatial feature properties. In the second paper, the

— A capsule can capture non-spacial feature properties. In the second paper, the capsule is a 4x4 pose matrix. The authors of the paper leaves a lot of rooms on what a capsule can be. And likely many will be proposed later.

^ | v • Reply • Share ›



**Shohrukh Bekmirzaev** • 16 days ago

— | 🚩

Great job and useful explanation ,Thanks

^ | v • Reply • Share ›



**Jonathan Hui** **Mod** → Shohrukh Bekmirzaev • 2 days ago

— | 🚩

Thanks.

^ | v • Reply • Share ›



**Guanghui Ren** • 20 days ago

— | 🚩

Thanks for your great explanantion! There is a typo.  $6 \times 6 \rightarrow (20-9)/2 + 1 = 6$ , not  $(20-9+1)/2 = 6$ .

^ | v • Reply • Share ›



**Jonathan Hui** **Mod** → Guanghui Ren • 20 days ago

— | 🚩

I should give you an award. People usually stop checking the math at the end of the article. Thanks. Fixed.

^ | v • Reply • Share ›



**Guanghui Ren** → Jonathan Hui • 20 days ago

— | 🚩

Can I reprint this blog, and translate it into Chinese?

^ | v • Reply • Share ›



**Jonathan Hui** **Mod** → Guanghui Ren • 20 days ago

— | 🚩

Yes. But please keep a link to the original research paper and give credits to XifengGuo for his code implementation. Let me know where it is posted. Will be interested to see it in Chinese.

^ | v • Reply • Share ›



**Guanghui Ren** → Jonathan Hui • 20 days ago

— | 🚩

Ok, Thanks! ps: I also add something I understand in the blog. here is the link. <http://renguanghui.com/2017...>

^ | v • Reply • Share ›



**Jonathan Hui** **Mod** → Guanghui Ren • 20 days ago

— | 🚩

谢谢! You do a great job. The research in this area is still in infant, and I try not to be too optimistic or pessimistic over some of the claims. You did a good job in translating that message. I am glad that you put your own interpretation also. If you have time, put some thoughts on the routing-by-agreement and its significant. Hinton believes the backpropagation has its own problem. The second paper expands its significant. Bloggers focus a lot in the Capsule part but the routing-by-agreement and the transformation is most likely the critical parts in research.

^ | v • Reply • Share ›



**Guanghui Ren** → Jonathan Hui • 19 days ago



Yeah, the routing-by-agreement is critical. Now it costs so much time and memory. If we train the capsule net on a big dataset, we'll need a vector of many dims. It will be a disaster. I'm expecting more researchers' improvement.

^ | v • Reply • Share ›



**Ayesha Ahmad** • a month ago



Brilliantly explained!

^ | v • Reply • Share ›



**Jonathan Hui** **Mod** → Ayesha Ahmad • a month ago



Thanks for the kind words. Writing it down forces me to understand the intuition behind the technology. But the brilliant part belongs to the paper authors and my thanks to XifengGuo for the code implementation. To make the technology more accessible, I may be over simplify it or making descriptions that are not necessary 100% accurate. So some cautions are needed.

^ | v • Reply • Share ›



**Ayesha Ahmad** → Jonathan Hui • a month ago



I like the simplified version because some of us are just understanding and its hard to picture the solution when you're bombarded with too many technical terms that is dense and hard to imagine. After reading your solution i have a picture in my mind and makes me more confident to explain it to someone else!

2 ^ | v • Reply • Share ›



**Aiman** • a month ago



Thank you very much for the nice explanations.

In the margin loss formula the prediction vector is  $\underline{v}c$  and not  $vc$ . This confused me for two seconds but no big deal

^ | v • Reply • Share ›



**Jonathan Hui** **Mod** → Aiman • a month ago



Thanks. Fixed.

^ | v • Reply • Share ›



**Michael Truong Le** • a month ago



Thanks for the explanantion. You did a really good job. It is still unclear to me why CapsNet solves the Picasso picture problem?

Is it due to the dynamic routing and that max pooling is not used anymore so that the location information is still preserved?

How do we know that CNNs do not encode the pose relation between nose/mouth/exes inherently in the feature maps?

^ | v • Reply • Share ›





**Jonathan Hui** Mod → Michael Truong Le • a month ago



I do believe that CNN can capture "some" spatial relationships BUT less (or far less) efficient. It requires more convolution layers, smaller filters and more feature maps. Also, the CNN approach is more brute force trying to model different spatial combinations to be detected by different feature maps at different layers. The solution is harder to generalize. Because of that, you require a whole lot of training data for different variants. In CapsNet, it captures variant properties (like spatial orientation and size of the feature). The routing algorithm depends on similarity of capsules vectors at 2 adjacent layers after some transformation. This extra information in the vector can decrease the capsule likelihood in the layer above if those properties does not agree in the lower layer. If I mis-place the mouth, can capsules detect it? I can only say capsules are more likely do it better than CNN as CNN goes deeper and deeper, the spatial information is likely loss or too implicitly defined which is hard to manipulate among neurons. I am a little bit conservative here because researchers are still working to scale the capsules to ImageNet.

^ | v • Reply • Share ›



**Alfredo Canziani** • a month ago



Thanks for the write-up. Some notes follow.

1. In section "Capsule" you don't describe the second column of images.
2. In the section "Compute the output of a capsule" the matrix multiplication dimensionality wants a "\times" rather than a "x".
3. In the section "Iterative dynamic Routing" you'd like to wrap "similarity" in a "\text{" field.
4. In the section "Loss function" you forgot to escape the "\max" operator with the backslash.
5. In the subsection "Reconstruction loss" "image" has to be "\text{"-ed.

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Alfredo Canziani • a month ago



Thanks. Fixed.

Since this is a github page, feel free to send a pull request if you know how to do it. Otherwise, feel free to put the suggested changes in the comment section also.

^ | v • Reply • Share ›



**Alfredo Canziani** → Jonathan Hui • a month ago



It's much better now. There are typos also on the paper, so I'll be emailing the author ;)

I usually prefer pointing out issues, and have the author taking care of them, so that there is some learning involved. I know, I'm annoyingly pedagogical.

^ | v • Reply • Share ›



**Benjamin hapunkt** • a month ago



"and ... are designed to sum to one." - I think the summation index should be j here, not i.

^ | v • Reply • Share ›



**Jonathan Hui** Mod

→ Benjamin hapunkt • a month ago



Thanks. Fixed.

Jonathan Hui blog



jhui

“Software architect”