



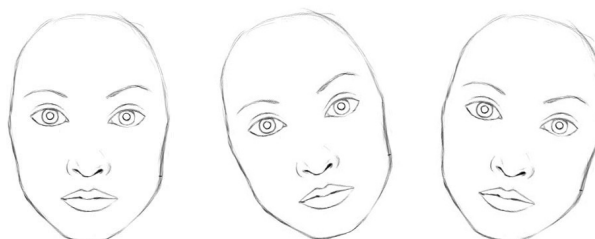
# "Understanding Matrix capsules with EM Routing (Based on Hinton's Capsule Networks)"

Nov 14, 2017

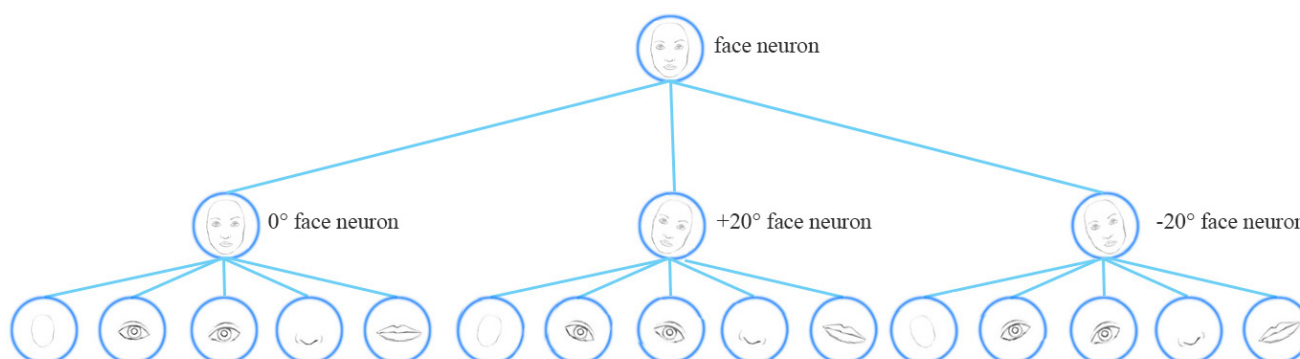
This article covers the second Hinton's capsule network paper [Matrix capsules with EM Routing](#), both authored by Geoffrey E Hinton, Sara Sabour and Nicholas Frosst. We will first cover the matrix capsules and apply EM (Expectation Maximization) routing to classify images with different viewpoints. For those want to understand the detail implementation, the second half of the article covers an implementation on the matrix Capsule and EM routing using TensorFlow.

## CNN challenges

In our [previous capsule article](#), we cover the challenges of CNN in exploring spatial relationship and discuss how capsule networks may address those short-comings. Let's recap some important challenges of CNN in classifying the same class of images but in different viewpoints. For example, classify a face correctly even with different orientations.



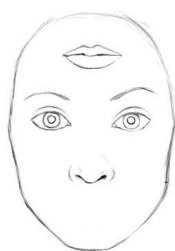
Conceptually, the CNN trains neurons to handle different feature orientations ( $0^\circ$ ,  $20^\circ$ ,  $-20^\circ$ ) with a top level face detection neuron.



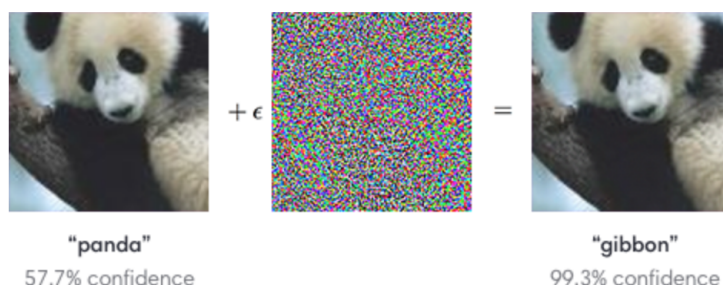
To solve the problem, we add more convolution layers and features maps. Nevertheless this approach tends to memorize the dataset rather than generalize a solution. It requires a large volume of training data to cover different variants and to avoid overfitting. MNist dataset contains 55,000 training data. i.e. 5,500 samples per digits. However, it is unlikely that children need so many samples to learn numbers. Our existing deep learning models including CNN are inefficient in utilizing datapoints.

## Adversaires

CNN is also vulnerable to adversaires by simply move, rotate or resize individual features.



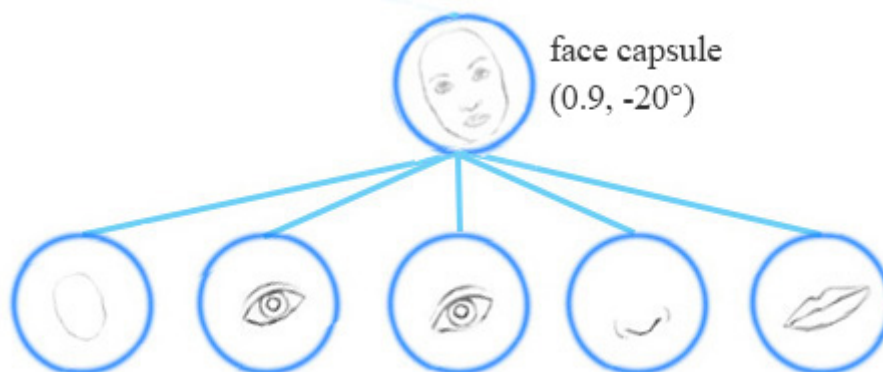
We can add tiny un-noticeable changes to an image to fool a deep network easily. The image on the left below is correctly classified by a CNN network as a panda. By selectively adding small changes from the middle picture to the panda picture, the CNN suddenly misclassifies the resulting image in the right as a gibbon.



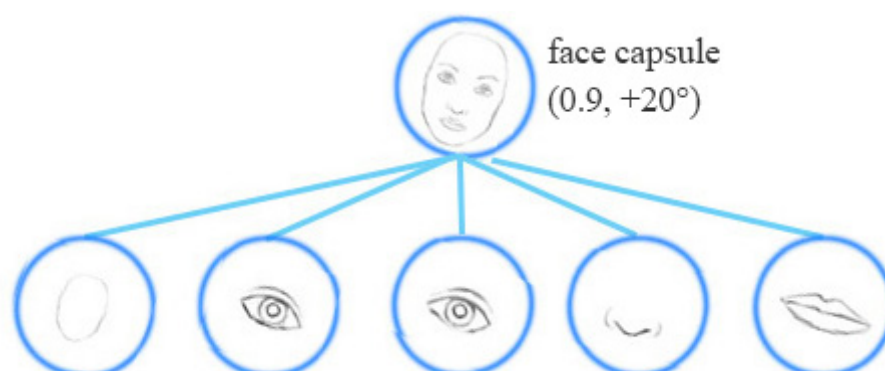
(image source [OpenAi](#))

## Capsule

**A capsule captures the likeliness of a feature and its variant.** So the capsule does not *only* detect a feature but it is trained to learn and detect the variants.



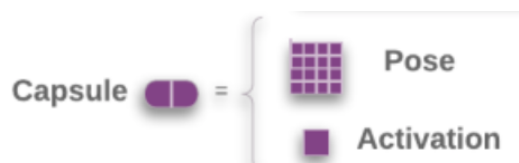
For example, the same network layer can detect a face rotated clockwise.



**Equivariance** is the detection of objects that can transform to each other. Intuitively, the capsule network detects the face is rotated right  $20^\circ$  (an equivariance) rather than realizes the face matched a variant that is rotated  $20^\circ$ . By forcing the model to learn the feature variant in a capsule, we *may* extrapolate possible variants more effectively with less training data. In CNN, the final label is viewpoint invariant. i.e. the top neuron detects a face but losses the information in the angle of rotation. For equivariance, the variant information like the angle of rotation is kept inside the capsule. Maintaining such spatial orientation helps us to avoid adversaires.

## Matrix capsule

A **matrix capsule** captures the activation (likeliness) similar to that of a neuron, but also captures a 4x4 pose matrix. In computer graphics, a pose matrix defines the translation and the rotation of an object which is equivalent to the change of the viewpoint of an object.



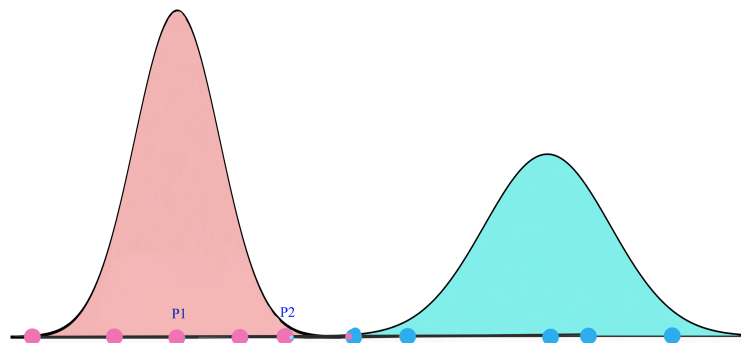
(Source from the Matrix capsules with EM routing paper)

For example, the second row images below represent the same object above them with different viewpoints. In matrix capsule, we train the model to capture the pose information (orientation, azimuths etc...). Of course, just like other deep learning methods, this is our intention and it is never guaranteed.



(Source from the Matrix capsules with EM routing paper)

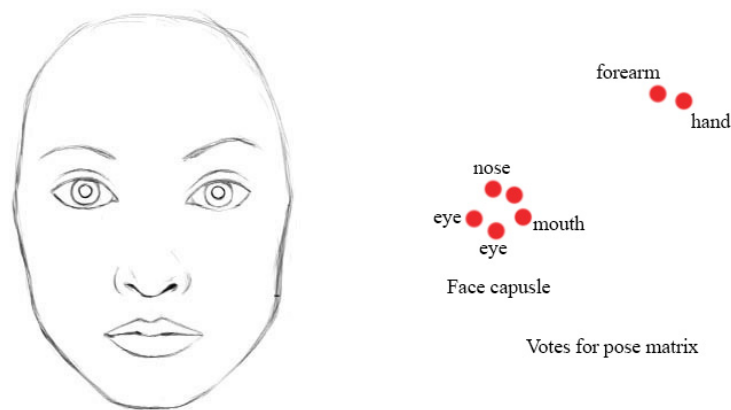
The objective of the EM (Expectation Maximization) routing is to group capsules to form a part-whole relationship using a clustering technique (EM). In machine learning, we use EM clustering to cluster datapoints into Gaussian distributions. For example, we cluster the datapoints below into two clusters modeled by two gaussian distributions  $G_1 = \mathcal{N}(\mu_1, \sigma_1^2)$  and  $G_2 = \mathcal{N}(\mu_2, \sigma_2^2)$ . Then we represent the datapoints by the corresponding Gaussian distribution.



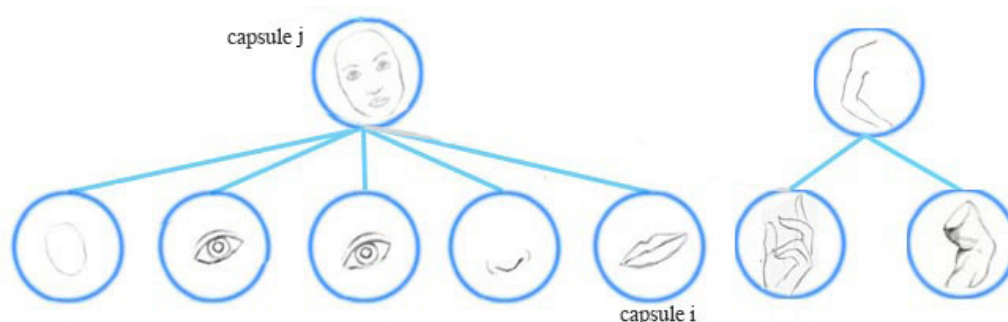
In the face detection example, each of the mouth, eyes and nose detection capsules in the lower layer makes predictions (**votes**) on the pose matrices of its possible parent capsules. Each vote is a predicted value for a parent capsule's pose matrix, and it is computed by multiplying its own pose matrix  $M$  with a **transformation matrix**  $W$  that we learn from the training data.

$$v = MW$$

We apply the EM routing to group capsules into a parent capsule in runtime:



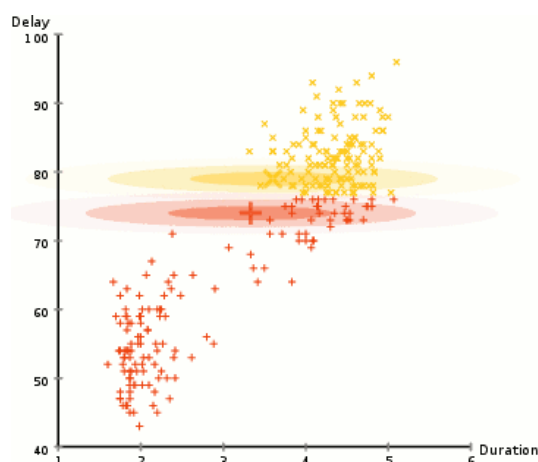
i.e., if the nose, mouth and eyes capsules all vote a similar pose matrix value, we cluster them together to form a parent capsule: the face capsule.



*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.*

## Gaussian mixture model & Expectation Maximization (EM)

We will take a short break to understand EM. A Gaussian mixture model clusters datapoints into a mixture of Gaussian distributions described by a mean  $\mu$  and a standard deviation  $\sigma$ . Below, we cluster datapoints into the yellow and red cluster which each is described by a  $\mu$  and a  $\sigma$ .



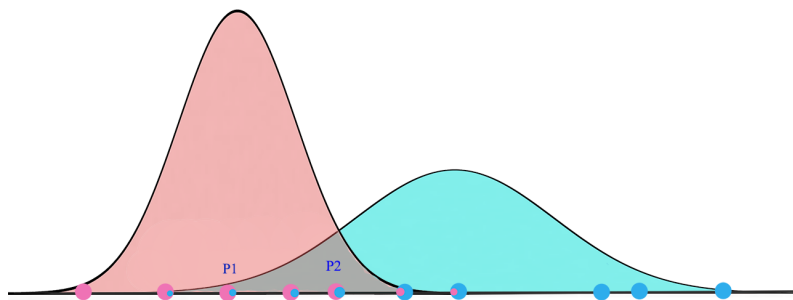
(Image source Wikipedia)

For a Gaussian mixture model with two clusters, we start with a random initialization of clusters  $G_1 = (\mu_1, \sigma_1^2)$  and  $G_2 = (\mu_2, \sigma_2^2)$ . Expectation Maximization (EM) algorithm tries to fit the training datapoints into  $G_1$  and  $G_2$  and then re-compute  $\mu$  and  $\sigma$  for  $G_1$  and  $G_2$  based on Gaussian distribution. The iteration continues until the solution converged such that the probability of seeing all datapoints is maximized with the final  $G_1$  and  $G_2$  distribution.

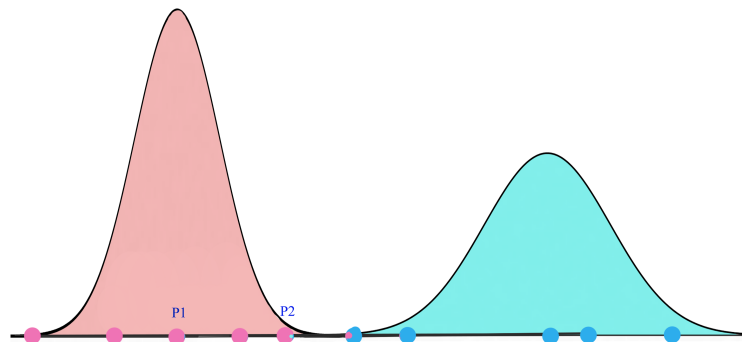
The probability of  $x$  given (belong to) the cluster  $G_1$  is:

$$P(x|G_1) = \frac{1}{\sigma_1 \sqrt{2\pi}} e^{-(x-\mu_1)^2/2\sigma_1^2}$$

At each iteration, we start with 2 Gaussian distributions which we later re-calculate its  $\mu$  and  $\sigma$  based on the datapoints.



Eventually, we will converge to two Gaussian distributions that maximize the likelihood of the observed datapoints.



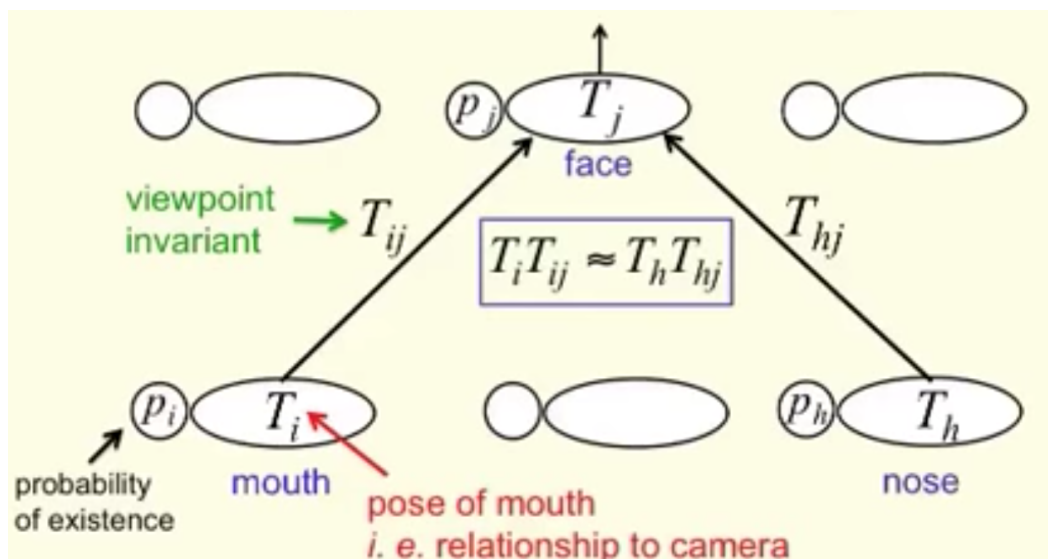
## Using EM for Routing-By-Agreement

Now, let's get into more details. A higher level feature (a face) is detected by looking for agreement between votes from the capsules one layer below. A **vote**  $v_{ij}$  for the parent capsule  $j$  from capsule  $i$  is computed by multiplying the pose matrix  $M_i$  of capsule  $i$  with a **viewpoint invariant transformation**  $W_{ij}$ .

$$v_{ij} = M_i W_{ij}$$

The probability that a capsule  $i$  is grouped into capsule  $j$  as a part-whole relationship is based on the proximity of the vote  $v_{ij}$  to other votes ( $v_{o_1j} \dots v_{o_kj}$ ) from other capsules.  $W_{ij}$  is learned discriminatively through a cost function and the backpropagation. It learns not only what a face is composed of, and it also makes sure the pose information of the parent capsule matched with its sub-components after some transformation.

Here is the visualization of routing-by-agreement with the matrix capsules. We group capsules with similar votes ( $T_i T_{ij} \approx T_h T_{hj}$ ) after transform the pose  $T_i$  and  $T_j$  with a viewpoint invariant transformation. ( $T_{ij}$  aka  $W_{ij}$  and  $T_{hj}$ )



(Source Geoffrey Hinton)

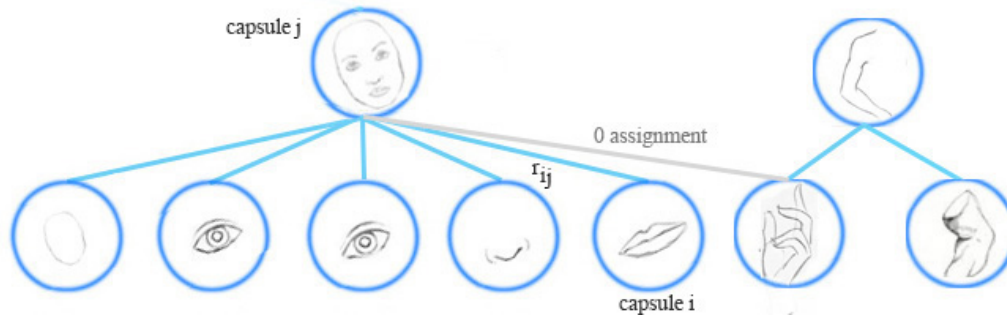
Even the viewpoint may change, the pose matrices and the votes change in a co-ordinate way. In our example, the locations of the votes may change from the red dots to the pink dots below when the face is rotated. Nevertheless, EM routing is based on proximity and therefore EM routing can still cluster the same children capsules together. Hence, the transformation matrices are the same for any viewpoints of the objects: viewpoint invariant. We just need one set of the transformation matrices and one parent capsule for different object orientations.





## Capsule assignment

EM routing clusters capsules to form a higher level capsule in runtime. It also calculates the **assignment probabilities**  $r_{ij}$  to quantify the runtime connection between a capsule and its parents. For example, the hand capsule does not belong to the face capsule, the assignment probability between them is zero.  $r_{ij}$  is also impacted by the activation of a capsule. If the mouth in the image is obstructed, the mouth capsule will have zero activation. The calculated  $r_{ij}$  will also be zero.



## Calculate capsule activation and pose matrix

The output of a capsule is computed differently than a deep network neuron. In EM clustering, we represent datapoints by a Gaussian distribution. In EM routing, we model the pose matrix of the parent capsule with a Gaussian also. The pose matrix is a  $4 \times 4$  matrix, i.e. 16 components. We model the pose matrix with a Gaussian having 16  $\mu$  and 16  $\sigma$  and each  $\mu$  represents a pose matrix's component.

Let  $v_{ij}$  be the vote from capsule  $i$  for the parent capsule  $j$ , and  $v_{ij}^h$  be its  $h$ -th component. We apply the probability density function of a Gaussian

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

to compute the probability of  $v_{ij}^h$  belonging to the capsule  $j$ 's Gaussian model:

$$p_{i|j}^h = \frac{1}{\sqrt{2\pi(\sigma_j^h)^2}} \exp\left(-\frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right)$$

Let's take the natural *log*:



$$\begin{aligned}\ln(p_{i|j}^h) &= \ln \frac{1}{\sqrt{2\pi(\sigma_j^h)^2}} \exp\left(-\frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) \\ &= -\ln(\sigma_j^h) - \frac{\ln(2\pi)}{2} - \frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\end{aligned}$$

Let's estimate the cost to activate a capsule. The lower the cost, the more likely a capsule will be activated. If cost is high, the votes do not match the parent Gaussian distribution and therefore a low chance to be activated.

Let  $cost_{ij}^h$  be the cost to activate the parent capsule  $j$  by the capsule  $i$ . It is the negative of the log likelihood:

$$cost_{ij}^h = -\ln(P_{i|j}^h)$$

Since capsules are not equally linked with capsule  $j$ , we pro-rated the cost with the runtime **assignment probabilities**  $r_{ij}$ . The cost from all lower layer capsules is:

$$\begin{aligned}cost_j^h &= \sum_i r_{ij} cost_{ij}^h \\ &= \sum_i -r_{ij} \ln(p_{i|j}^h) \\ &= \sum_i r_{ij} \left( \frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2} + \ln(\sigma_j^h) + \frac{\ln(2\pi)}{2} \right) \\ &= \frac{\sum_i r_{ij} (\sigma_j^h)^2}{2(\sigma_j^h)^2} + \left( \ln(\sigma_j^h) + \frac{\ln(2\pi)}{2} \right) \sum_i r_{ij} \\ &= (\ln(\sigma_j^h) + k) \sum_i r_{ij} \quad \text{which } k \text{ is a constant}\end{aligned}$$

To determine whether the capsule  $j$  will be activated, we use the following equation:

$$a_j = \text{sigmoid}(\lambda(b_j - \sum_h cost_j^h))$$

In the original paper, “ $-b_j$ ” is explained as the cost of describing the mean and variance of capsule  $j$ . In another word, if the benefit  $b_j$  of representing datapoints by the parent capsule  $j$  outranks the cost caused by the discrepancy in their votes, we activate the output capsules. We do not compute  $b_j$  analytically. Instead, we approximate it through training using the backpropagation and a cost function.

$r_{ij}$ ,  $\mu$ ,  $\sigma$  and  $a_j$  are computed iteratively using EM routing discussed in the next section.  $\lambda$  in the equation above is the inverse temperature parameter  $\frac{1}{\text{temperature}}$ . As  $r_{ij}$  is getting better, we drop the temperature (larger  $\lambda$ ) which increase the steepness in the s-curve in  $\text{sigmoid}(z)$ . It helps us to fine tune  $r_{ij}$  better with higher sensitivity to vote discrepancy in these region. In our implementation,  $\lambda$  is first initialized to 1 at the beginning of the iterations and then increment by 1 after each routing iteration. The paper does not specific the details and we would suggest experimenting different schemes in your implementation.

## EM Routing

The pose matrix and the activation of the output capsules are computed iteratively using the EM routing. The EM method fits datapoints into a mixture of Gaussian models with alternative calls between an E-step and an M-step. The E-step determines the assignment probability  $r_{ij}$  of each datapoint to a parent capsule. The M-step re-calculate the Gaussian models' values based on  $r_{ij}$ . We repeat the iteration 3 times. The last  $a_j$  will be the parent capsule's output. The 16  $\mu$  from the last Gaussian model will be reshaped to form the  $4 \times 4$  pose matrix of the parent capsule.

```

procedure EM ROUTING( $\mathbf{a}, V$ )
   $\forall i \in \Omega_L, j \in \Omega_{L+1}: R_{ij} \leftarrow 1/|\Omega_{L+1}|$ 
  for  $t$  iterations do
     $\forall j \in \Omega_{L+1}: \text{M-STEP}(\mathbf{a}, R, V, j)$ 
     $\forall i \in \Omega_L: \text{E-STEP}(\mu, \sigma, \mathbf{a}, V, i)$ 
  return  $\mathbf{a}, M$ 

```

(Source from the Matrix capsules with EM routing paper)

$\mathbf{a}$  and  $V$  above is the activation and the votes from the children capsules. We initialize the assignment probability  $r_{ij}$  to be uniformly distributed. i.e. we start with the children capsules equally related with any parents. We call M-step to compute an updated Gaussian model ( $\mu$ ,  $\sigma$ ) and the parent activation  $a_j$  from  $\mathbf{a}$ ,  $V$  and current  $r_{ij}$ . Then we call E-step to recompute the assignment probabilities  $r_{ij}$  based on the new Gaussian model and the new  $a_j$ .

The details of M-step:

```

procedure M-STEP( $\mathbf{a}, R, V, j$ )                                      $\triangleright$  for one higher-level capsule
   $\forall i \in \Omega_L: R_{ij} \leftarrow R_{ij} * \mathbf{a}_i$ 
   $\forall h: \mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i R_{ij}}$ 
   $\forall h: (\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i R_{ij}}$ 
   $\text{cost}^h \leftarrow (\beta_v + \log(\sigma_j^h)) \sum_i R_{ij}$ 
   $a_j \leftarrow \text{sigmoid}(\lambda(\beta_a - \sum_h \text{cost}^h))$ 

```

In M-step, we calculate  $\mu$  and  $\sigma$  based on the activation  $a_i$  from the children capsules, the current  $r_{ij}$  and votes  $V$ . M-step also re-calculate the cost and the activation  $a_j$  for the parent

capsules.  $\beta_\nu$  and  $\beta_\alpha$  is trained discriminatively.  $\lambda$  is an inverse temperature parameter increased by 1 after each routing iteration in our code implementation.

The details of E-step:

**procedure** E-STEP( $\mu, \sigma, \mathbf{a}, V, i$ ) ▷ for one lower-level capsule

$$\forall j \in \Omega_{L+1}: \mathbf{p}_j \leftarrow \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} e^{-\sum_h^H \frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}}$$

$$\forall j \in \Omega_{L+1}: \mathbf{R}_{ij} \leftarrow \frac{\mathbf{a}_j \mathbf{p}_j}{\sum_{u \in \Omega_{L+1}} \mathbf{a}_u \mathbf{p}_u}$$

In E-step, we re-calculate the assignment probability  $r_{ij}$  based on the new  $\mu, \sigma$  and  $\mathbf{a}_j$ . The assignment is increased if the vote is closer to the  $\mu$  of the updated Gaussian model.

*We use the  $\mathbf{a}_j$  from the last m-step call in the iterations as the activation of the output capsule  $j$  and we shape the 16  $\mu$  to form the 4x4 pose matrix.*

## The role of backpropagation & EM routing

In CNN, we use the formula below to calculate the activation of a neuron.

$$y_j = ReLU(\sum_i W_{ij} * x_i + b_j)$$

However, the output of a capsule including the activation and the pose matrix is calculated by the EM routing. We use EM-routing to compute the parent capsule's output from the transformation matrix  $W$  and the children capsules' activations and pose matrices. Nevertheless, by no mistake, the matrix capsules still heavily depend on the backpropagation in training the transformation matrix  $W_{ij}$  and the parameters  $\beta_\nu$  and  $\beta_\alpha$ .

In EM routing, we compute the assignment probability  $r_{ij}$  to quantify the connection between the children capsules and the parent capsules. This value is important but short lived. We re-initialize it to be uniformly distributed for every datapoint before the EM routing calculation. In any situation, training or testing, we use EM routing to compute capsules' outputs.

## Loss function (using Spread loss)

Matrix capsules requires a loss function to train  $W, \beta_\nu$  and  $\beta_\alpha$ . We pick spread loss as the main loss function for the backpropagation. The loss from the class  $i$  (other than the true label  $t$ ) is defined as

$$L_i = (\max(0, m - (a_t - a_i)))^2$$

which  $a_t$  is the activation of the target class (true label) and  $a_i$  is the activation for class  $i$ .

The total cost is:

$$L = \sum_{i \neq t} (\max(0, m - (a_t - a_i)))^2$$

If the margin between the true label and the wrong class is smaller than  $m$ , we penalize it by the square of  $m - (a_t - a_i)$ .  $m$  initially start as 0.2 and linearly increased by 0.1 after each epoch training.  $m$  will stop growing after reaching the maximum 0.9. Starting at a lower margin helps the training to avoid too many dead capsules during the early phase.

Other implementations add regularization loss and reconstruction loss to the loss function. Since those are not specific to the matrix capsule, we will simply mention here without further elaboration.

## Capsule Network

The Capsule Network (CapsNet) in the first paper uses a fully connected network. This solution is hard to scale for larger images. In the next few sections, the matrix capsules use some of the convolution techniques in CNN to explore spatial features so it can scale better.

### smallNORB

The research paper uses the smallNORB dataset. It has 5 toy classes: airplanes, cars, trucks, humans and animals. Every individual sample is pictured at 18 different azimuths (0-340), 9 elevations and 6 lighting conditions. This dataset is particular picked by the paper so it can study the classification of images with different viewpoints.

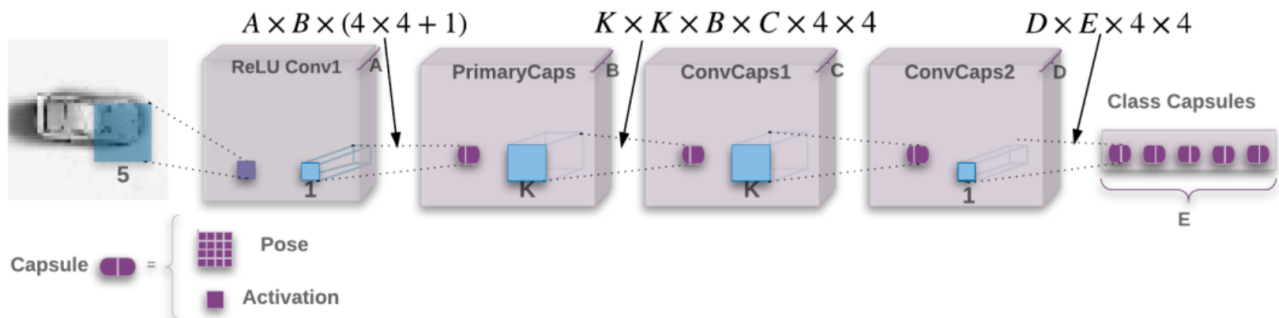


(Picture from the Matrix capsules with EM routing paper)

### Architect

However, many code implementations start with the MNist dataset because the image size is smaller. So for our demonstration, we also pick the MNist dataset. This is the network

design:



(Picture from the Matrix capsules with EM routing paper).

ReLU Conv1 is a regular convolution (CNN) layer using a 5x5 filter with stride 2 outputting 32 ( $A = 32$ ) channels (feature maps) using the ReLU activation.

In PrimaryCaps, we apply a 1x1 convolution filter to transform each of the 32 channels into 32 ( $B = 32$ ) primary capsules. Each capsule contains a 4x4 pose matrix and an activation value. We use the regular convolution layer to implement the PrimaryCaps. We group  $4 \times 4 + 1$  neurons to generate 1 capsule.

PrimaryCaps is followed by a **convolution capsule layer** ConvCaps1 using a 3x3 filters ( $K = 3$ ) with stride 2. ConvCaps1 takes capsules as input and output capsules. ConvCaps1 is similar to a regular convolution layer except it uses EM routing to compute the capsule output.

The capsule output of ConvCaps1 is then feed into ConvCaps2. ConvCaps2 is another convolution capsule layer but with stride 1.

The output capsules of ConvCaps2 are connected to the Class Capsules using a 1x1 filter and it outputs one capsule per class. (In MNist, we have 10 classes  $E = 10$ )

We use EM routing to compute the pose matrices and the output activations for ConvCaps1, ConvCaps2 and Class Capsules. In CNN, we slide the same filter over the spatial dimension in calculating the same feature map. We want to detect the same feature regardless of the location. Similarly, in EM routing, we share the same transformation matrix  $W_{ij}$  across spatial dimension to calculate the votes.

For example, from ConvCaps1 to ConvCaps2, we have

- a 3x3 filter,
- 32 input and output capsules, and
- a 4x4 pose matrix.

Since we share the transformation matrix, we just need  $3 \times 3 \times 32 \times 32 \times 4 \times 4$  parameters for  $W$ .

Here is the summary of each layer and the shape of their outputs:

Layer Name	Apply	Output shape
MNist image		28, 28, 1
ReLU Conv1	Regular Convolution (CNN) layer using 5x5 kernels with 32 output channels, stride 2 and padding	14, 14, 32
PrimaryCaps	Modified convolution layer with 1x1 kernels, strides 1 with padding and outputting 32 capsules. Requiring 32x32x(4x4+1) parameters.	pose (14, 14, 32, 4, 4), activations (14, 14, 32)
ConvCaps1	Capsule convolution with 3x3 kernels, strides 2 and no padding. Requiring 3x3x32x32x4x4 parameters.	poses (6, 6, 32, 4, 4), activations (6, 6, 32)
ConvCaps2	Capsule convolution with 3x3 kernels, strides 1 and no padding	poses (4, 4, 32, 4, 4), activations (4, 4, 32)
Class Capsules	Capsule with 1x1 kernel. Requiring 32x10x4x4 parameters.	poses (10, 4, 4), activations (10)

For the rest of the article, we will cover a detail implementation using Tensor. Here is the code in building our layers:

```
def capsules_net(inputs, num_classes, iterations, batch_size, name='capsules_net'):
    """Define the Capsule Network model"""

    with tf.variable_scope(name) as scope:
        # ReLU Conv1
        # Images shape (24, 28, 28, 1) -> conv 5x5 filters, 32 output channels
        # nets -> (?, 14, 14, 32)
        nets = conv2d(
            inputs,
            kernel=5, out_channels=32, stride=2, padding='SAME',
            activation_fn=tf.nn.relu, name='relu_conv1'
        )

        # PrimaryCaps
        # (?, 14, 14, 32) -> capsule 1x1 filter, 32 output capsule, stride 1
        # nets -> (poses (?, 14, 14, 32, 4, 4), activations (?, 14, 14, 32))
        nets = primary_caps(
```

```

nets,
kernel_size=1, out_capsules=32, stride=1, padding='VALID',
pose_shape=[4, 4], name='primary_caps'
)

# ConvCaps1
# (poses, activations) -> conv capsule, 3x3 kernels, strides 2,
# nets -> (poses (24, 6, 6, 32, 4, 4), activations (24, 6, 6, 32, 4, 4))
nets = conv_capsule(
    nets, shape=[3, 3, 32, 32], strides=[1, 2, 2, 1], iterations=iterations,
    batch_size=batch_size, name='conv_caps1'
)

# ConvCaps2
# (poses, activations) -> conv capsule, 3x3 kernels, strides 1,
# nets -> (poses (24, 4, 4, 32, 4, 4), activations (24, 4, 4, 32, 4, 4))
nets = conv_capsule(
    nets, shape=[3, 3, 32, 32], strides=[1, 1, 1, 1], iterations=iterations,
    batch_size=batch_size, name='conv_caps2'
)

# Class capsules
# (poses, activations) -> 1x1 convolution, 10 output capsules
# nets -> (poses (24, 10, 4, 4), activations (24, 10))
nets = class_capsules(nets, num_classes, iterations=iterations,
                      batch_size=batch_size, name='class_capsules')

# poses (24, 10, 4, 4), activations (24, 10)
poses, activations = nets

return poses, activations

```

## ReLU Conv1

ReLU Conv1 is a simple CNN layer. We use the TensorFlow slim API `slim.conv2d` to create a CNN layer using a 3x3 kernel with stride 2 and ReLU. (We use the slim API so the code is more condense and easier to read.)

```

def conv2d(inputs, kernel, out_channels, stride, padding, name, is_train):
    with slim.arg_scope([slim.conv2d], trainable=is_train):
        with tf.variable_scope(name) as scope:
            output = slim.conv2d(inputs,
                                num_outputs=out_channels,
                                kernel_size=[kernel, kernel], stride=stride,
                                scope=scope, activation_fn=activation_fn)

```



```
tf.logging.info(f"{name} output shape: {output.get_shape()}")
```

```
return output
```

## PrimaryCaps

PrimaryCaps is not much difference from a CNN layer: instead of generating 1 scalar value, we generate 32 capsules with 4 x 4 scalar values for the pose matrices and 1 scalar for the activation:

```
def primary_caps(inputs, kernel_size, out_capsules, stride, padding, pose_shape, name):
    """This constructs a primary capsule layer using regular convolution.

    :param inputs: shape (N, H, W, C) (?, 14, 14, 32)
    :param kernel_size: Apply a filter of [kernel, kernel] [5x5]
    :param out_capsules: # of output capsule (32)
    :param stride: 1, 2, or ... (1)
    :param padding: padding: SAME or VALID.
    :param pose_shape: (4, 4)
    :param name: scope name

    :return: (poses, activations), (poses (?, 14, 14, 32, 4, 4), activations)
    """

    with tf.variable_scope(name) as scope:
        # Generate the poses matrices for the 32 output capsules
        poses = conv2d(
            inputs,
            kernel_size, out_capsules * pose_shape[0] * pose_shape[1],
            name='pose_stacked'
        )

        input_shape = inputs.get_shape()

        # Reshape 16 scalar values into a 4x4 matrix
        poses = tf.reshape(
            poses, shape=[-1, input_shape[-3], input_shape[-2], out_capsules]
        )

        # Generate the activation for the 32 output capsules
        activations = conv2d(
            inputs,
            kernel_size,
            out_capsules,
```

```

        stride,
        padding=padding,
        activation_fn=tf.sigmoid,
        name='activation'
    )

    tf.summary.histogram(
        'activations', activations
    )

    # poses (?, 14, 14, 32, 4, 4), activations (?, 14, 14, 32)
    return poses, activations

```

## ConvCaps1, ConvCaps2

ConvCaps1 and ConvCaps are both convolution capsule with stride 2 and 1 respectively. In the comment of the following code, we trace the tensor shape for the ConvCaps1 layer.

The code involves 3 major steps:

- Use `kernel_tile` to tile (convolute) the pose matrices and the activation to be used later in voting and EM routing.
- Compute votes: call `mat_transform` to generate the votes from the children “tiled” pose matrices and the transformation matrices.
- EM-routing: call `matrix_capsules_em_routing` to compute the output capsules (pose and activation) of the parent capsule.

```

def conv_capsule(inputs, shape, strides, iterations, batch_size, name):
    """This constructs a convolution capsule layer from a primary or convolution capsule layer.

    i: input capsules (32)
    o: output capsules (32)
    batch size: 24
    spatial dimension: 14x14
    kernel: 3x3

    :param inputs: a primary or convolution capsule layer with poses and activations
        pose: (24, 14, 14, 32, 4, 4)
        activation: (24, 14, 14, 32)
    :param shape: the shape of convolution operation kernel, [kh, kw, i, o]
    :param strides: often [1, 2, 2, 1] (stride 2), or [1, 1, 1, 1] (stride 1)
    :param iterations: number of iterations in EM routing. 3
    :param name: name.

    :return: (poses, activations).
    """

```

```
"""
```

```
inputs_poses, inputs_activations = inputs
```

```
with tf.variable_scope(name) as scope:
```

```
    stride = strides[1] # 2
```

```
    i_size = shape[-2] # 32
```

```
    o_size = shape[-1] # 32
```

```
    pose_size = inputs_poses.get_shape()[-1] # 4
```

```
    # Tile the input capsules' pose matrices to the spatial dimension
```

```
    # Such that we can later multiple with the transformation matrices
```

```
    inputs_poses = kernel_tile(inputs_poses, 3, stride) # (?, 14, 14,
```

```
    # Tile the activations needed for the EM routing
```

```
    inputs_activations = kernel_tile(inputs_activations, 3, stride) #
```

```
    spatial_size = int(inputs_activations.get_shape()[1]) # 6
```

```
    # Reshape it for later operations
```

```
    inputs_poses = tf.reshape(inputs_poses, shape=[-1, 3 * 3 * i_size
```

```
    inputs_activations = tf.reshape(inputs_activations, shape=[-1, sp
```

```
with tf.variable_scope('votes') as scope:
```

```
    # Generate the votes by multiply it with the transformation m
```

```
    votes = mat_transform(inputs_poses, o_size, size=batch_size*sp
```

```
    # Reshape the vote for EM routing
```

```
    votes_shape = votes.get_shape()
```

```
    votes = tf.reshape(votes, shape=[batch_size, spatial_size, sp
```

```
    tf.logging.info(f"{name} votes shape: {votes.get_shape()}")
```

```
with tf.variable_scope('routing') as scope:
```

```
    # beta_v and beta_a one for each output capsule: (1, 1, 1, 32,
```

```
    beta_v = tf.get_variable(
        name='beta_v', shape=[1, 1, 1, o_size], dtype=tf.float32,
        initializer=initializers.xavier_initializer()
    )
```

```
    beta_a = tf.get_variable(
        name='beta_a', shape=[1, 1, 1, o_size], dtype=tf.float32,
        initializer=initializers.xavier_initializer()
    )
```

```
    # Use EM routing to compute the pose and activation
```

```
    # votes (24, 6, 6, 3x3x32=288, 32, 16), inputs_activations (?,
```

```
    # poses (24, 6, 6, 32, 16), activation (24, 6, 6, 32)
```

```

poses, activations = matrix_capsules_em_routing(
    votes, inputs_activations, beta_v, beta_a, iterations, n
)

# Reshape it back to 4x4 pose matrix
poses_shape = poses.get_shape()
# (24, 6, 6, 32, 4, 4)
poses = tf.reshape(
    poses, [
        poses_shape[0], poses_shape[1], poses_shape[2], poses
    ]
)

tf.logging.info(f"{name} pose shape: {poses.get_shape()}")
tf.logging.info(f"{name} activations shape: {activations.get_shape()}")

return poses, activations

```

kernel\_tile use tiling and convolution to prepare the input pose matrices and the activations to the correct spatial dimension for voting and EM-routing. (The code is pretty hard to understand so readers may simply treat it as a black box if it is too hard.)

```

def kernel_tile(input, kernel, stride):
    """This constructs a primary capsule layer using regular convolution

    :param inputs: shape (?, 14, 14, 32, 4, 4)
    :param kernel: 3
    :param stride: 2

    :return output: (50, 5, 5, 3x3=9, 136)
    """

    # (?, 14, 14, 32x(16)=512)
    input_shape = input.get_shape()
    size = input_shape[4]*input_shape[5] if len(input_shape)>5 else 1
    input = tf.reshape(input, shape=[-1, input_shape[1], input_shape[2],
                                     size])

    input_shape = input.get_shape()
    tile_filter = np.zeros(shape=[kernel, kernel, input_shape[3],
                                   kernel * kernel], dtype=np.float32)

    for i in range(kernel):
        for j in range(kernel):
            tile_filter[i, j, :, i * kernel + j] = 1.0 # (3, 3, 512, 9)

```

```

# (3, 3, 512, 9)
tile_filter_op = tf.constant(tile_filter, dtype=tf.float32)

# (?, 6, 6, 4608)
output = tf.nn.depthwise_conv2d(input, tile_filter_op, strides=[
    1, stride, stride, 1], padding='VALID')

output_shape = output.get_shape()
output = tf.reshape(output, shape=[-1, output_shape[1], output_shape[2], output_shape[3]])
output = tf.transpose(output, perm=[0, 1, 2, 4, 3])

# (?, 6, 6, 9, 512)
return output

```

`mat_transform` extracts the transformation matrices parameters as a TensorFlow trainable variable  $w$ . It then multiplies with the “tiled” input pose matrices to generate the votes for the parent capsules.

```

def mat_transform(input, output_cap_size, size):
    """Compute the vote.

    :param inputs: shape (size, 288, 16)
    :param output_cap_size: 32

    :return votes: (24, 5, 5, 3x3=9, 136)
    """

    caps_num_i = int(input.get_shape()[1]) # 288
    output = tf.reshape(input, shape=[size, caps_num_i, 1, 4, 4]) # (size, 288, 1, 4, 4)

    w = slim.variable('w', shape=[1, caps_num_i, output_cap_size, 4, 4],
                      initializer=tf.truncated_normal_initializer(mean=0, stddev=0.1))
    w = tf.tile(w, [size, 1, 1, 1, 1]) # (24, 288, 32, 4, 4)

    output = tf.tile(output, [1, 1, output_cap_size, 1, 1]) # (size, 288, 32, 4, 4)

    votes = tf.matmul(output, w) # (24, 288, 32, 4, 4)
    votes = tf.reshape(votes, [size, caps_num_i, output_cap_size, 16]) # (24, 288, 32, 16)

    return votes

```

## EM routing coding

```

procedure EM ROUTING( $\mathbf{a}$ ,  $V$ )
   $\forall i \in \Omega_L, j \in \Omega_{L+1}$ :  $R_{ij} \leftarrow 1/|\Omega_{L+1}|$ 
  for  $t$  iterations do
     $\forall j \in \Omega_{L+1}$ : M-STEP( $\mathbf{a}$ ,  $R$ ,  $V$ ,  $j$ )
     $\forall i \in \Omega_L$ : E-STEP( $\mu$ ,  $\sigma$ ,  $\mathbf{a}$ ,  $V$ ,  $i$ )
  return  $\mathbf{a}$ ,  $\hat{M}$ 

```

Here is the code implementation for the EM routing which calling m\_step and e\_step alternatively. By default, we ran the iterations 3 times. The main purpose of the EM routing is to compute the pose matrices and the activations of the output capsules. In the last iteration, the m\_step already complete the last calculation of those parameters. Therefore we skip the e\_step in the last iteration which mainly responsible for re-calculating the routing assignment  $r_{ij}$ . The comments contain the tracing of the shape of tensors in ConvCaps1.

```

def matrix_capsules_em_routing(votes, i_activations, beta_v, beta_a, iterations):
    """The EM routing between input capsules (i) and output capsules (j).

    :param votes: (N, OH, OW, kh x kw x i, o, 4 x 4) = (24, 6, 6, 3x3*32=288, 1, 1)
    :param i_activation: activation from Level L (24, 6, 6, 288)
    :param beta_v: (1, 1, 1, 32)
    :param beta_a: (1, 1, 1, 32)
    :param iterations: number of iterations in EM routing, often 3.
    :param name: name.

    :return: (pose, activation) of output capsules.
    """

    votes_shape = votes.get_shape().as_list()

    with tf.variable_scope(name) as scope:

        # Match rr (routing assignment) shape, i_activations shape with votes
        # rr: [3x3x32=288, 32, 1]
        # rr: routing matrix from each input capsule (i) to each output capsule (j)
        rr = tf.constant(
            1.0/votes_shape[-2], shape=votes_shape[-3:-1] + [1], dtype=tf.float32, name='rr'
        )

        # i_activations: expand_dims to (24, 6, 6, 288, 1, 1)
        i_activations = i_activations[..., tf.newaxis, tf.newaxis]

        # beta_v and beta_a: expand_dims to (1, 1, 1, 1, 32, 1)
        beta_v = beta_v[..., tf.newaxis, :, tf.newaxis]
        beta_a = beta_a[..., tf.newaxis, :, tf.newaxis]

        # inverse_temperature schedule (min, max)

```

```

it_min = 1.0
it_max = min(iterations, 3.0)
for it in range(iterations):
    inverse_temperature = it_min + (it_max - it_min) * it / max(1.0, iterations)
    o_mean, o_stdv, o_activations = m_step(
        rr, votes, i_activations, beta_v, beta_a, inverse_temperature=inverse_temperature
    )

    # We skip the e_step call in the last iteration because we only
    # need to return the a_j and the mean from the m_step in the last
    # to compute the output capsule activation and pose matrices
    if it < iterations - 1:
        rr = e_step(
            o_mean, o_stdv, o_activations, votes
        )

    # pose: (N, OH, OW, o 4 x 4) via squeeze o_mean (24, 6, 6, 32, 16)
    poses = tf.squeeze(o_mean, axis=-3)

    # activation: (N, OH, OW, o) via squeeze o_activations [24, 6, 6, 1]
    activations = tf.squeeze(o_activations, axis=[-3, -1])

return poses, activations

```

In the equation for the output capsule's activation  $a_j$

$$a_j = \text{sigmoid}(\lambda(b_j - \sum_h \text{cost}_j^h))$$

$\lambda$  is an inverse temperature parameter. In our implementation, we start from 1 and increment it by 1 after each routing iteration. The original paper does not specify how  $\lambda$  is increased and you can experiment different schemes instead. Here is our source code:

```

# inverse_temperature schedule (min, max)
it_min = 1.0
it_max = min(iterations, 3.0)
for it in range(iterations):
    inverse_temperature = it_min + (it_max - it_min) * it / max(1.0, iterations)

    o_mean, o_stdv, o_activations = m_step(
        rr, votes, i_activations, beta_v, beta_a, inverse_temperature=inverse_temperature
    )

```



After the last iteration loop,  $a_j$  is output as the final activation of the output capsule  $j$ . The mean  $\mu_j^h$  is used for the final value of the h-th component of the corresponding pose matrix. We later reshape those 16 components into a 4x4 pose matrix.

```
# pose: (N, OH, OW, o 4 x 4) via squeeze o_mean (24, 6, 6, 32, 16)
poses = tf.squeeze(o_mean, axis=-3)

# activation: (N, OH, OW, o) via squeeze o_activationis [24, 6, 6, 32]
activations = tf.squeeze(o_activations, axis=[-3, -1])
```

## m-steps

The algorithm for the m-steps.

**procedure** M-STEP( $\mathbf{a}$ ,  $R$ ,  $V$ ,  $j$ ) ▷ for one higher-level capsule

$$\forall i \in \Omega_L: R_{ij} \leftarrow R_{ij} * \mathbf{a}_i$$

$$\forall h: \mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i R_{ij}}$$

$$\forall h: (\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i R_{ij}}$$

$$cost^h \leftarrow (\beta_v + \log(\sigma_j^h)) \sum_i R_{ij}$$

$$\mathbf{a}_j \leftarrow \text{sigmoid}(\lambda(\beta_a - \sum_h cost^h))$$

m\_step computes the mean and the variance of the parent capsules. Means and variances have the shape of (24, 6, 6, 1, 32, 16) and (24, 6, 6, 1, 32, 1) respectively in ConvCaps1.

The following is the code listing of the m-step method.

```
def m_step(rr, votes, i_activations, beta_v, beta_a, inverse_temperature):
    """The M-Step in EM Routing from input capsules i to output capsule j
    i: input capsules (32)
    o: output capsules (32)
    h: 4x4 = 16
    output spatial dimension: 6x6
    :param rr: routing assignments. shape = (kh x kw x i, o, 1) =(3x3x32,
    :param votes. shape = (N, OH, OW, kh x kw x i, o, 4x4) = (24, 6, 6, 24,
    :param i_activations: input capsule activation (at Level L). (N, OH, OW,
    with dimensions expanded to match votes for broadcasting.
    :param beta_v: Trainable parameters in computing cost (1, 1, 1, 1, 32)
    :param beta_a: Trainable parameters in computing next level activation
    :param inverse_temperature: lambda, increase over each iteration by 1
    :return: (o_mean, o_stdv, o_activation)
    """
```

```

rr_prime = rr * i_activations

# rr_prime_sum: sum over all input capsule i
rr_prime_sum = tf.reduce_sum(rr_prime, axis=-3, keep_dims=True, name=

# o_mean: (24, 6, 6, 1, 32, 16)
o_mean = tf.reduce_sum(
    rr_prime * votes, axis=-3, keep_dims=True
) / rr_prime_sum

# o_stdv: (24, 6, 6, 1, 32, 16)
o_stdv = tf.sqrt(
    tf.reduce_sum(
        rr_prime * tf.square(votes - o_mean), axis=-3, keep_dims=True
    ) / rr_prime_sum
)

# o_cost_h: (24, 6, 6, 1, 32, 16)
o_cost_h = (beta_v + tf.log(o_stdv + epsilon)) * rr_prime_sum

# o_cost: (24, 6, 6, 1, 32, 1)
# o_activations_cost = (24, 6, 6, 1, 32, 1)
# yg: This is done for numeric stability.
# It is the relative variance between each channel determined which o
o_cost = tf.reduce_sum(o_cost_h, axis=-1, keep_dims=True)
o_cost_mean = tf.reduce_mean(o_cost, axis=-2, keep_dims=True)
o_cost_stdv = tf.sqrt(
    tf.reduce_sum(
        tf.square(o_cost - o_cost_mean), axis=-2, keep_dims=True
    ) / o_cost.get_shape().as_list()[-2]
)
o_activations_cost = beta_a + (o_cost_mean - o_cost) / (o_cost_stdv +

# (24, 6, 6, 1, 32, 1)
o_activations = tf.sigmoid(
    inverse_temperature * o_activations_cost
)

return o_mean, o_stdv, o_activations

```

## E-steps

The algorithm for the e-steps.

**procedure** E-STEP( $\mu, \sigma, \mathbf{a}, V, i$ )

▷ for one lower-level capsule

$$\forall j \in \Omega_{L+1}: \mathbf{p}_j \leftarrow \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} e^{-\sum_h^H \frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}}$$

$$\forall j \in \Omega_{L+1}: \mathbf{R}_{ij} \leftarrow \frac{\mathbf{a}_j \mathbf{p}_j}{\sum_{u \in \Omega_{L+1}} \mathbf{a}_u \mathbf{p}_u}$$

e\_step is mainly responsible for re-calculating the routing assignment (shape: 24, 6, 6, 288, 32, 1) after m\_step updates the output activation  $\mathbf{a}_j$  and the Gaussian models with new  $\mu$  and  $\sigma$ .

```
def e_step(o_mean, o_stdv, o_activations, votes):
    """The E-Step in EM Routing.

    :param o_mean: (24, 6, 6, 1, 32, 16)
    :param o_stdv: (24, 6, 6, 1, 32, 16)
    :param o_activations: (24, 6, 6, 1, 32, 1)
    :param votes: (24, 6, 6, 288, 32, 16)

    :return: rr
    """

    o_p_unit0 = - tf.reduce_sum(
        tf.square(votes - o_mean) / (2 * tf.square(o_stdv)), axis=-1, keep_
    )

    o_p_unit2 = - tf.reduce_sum(
        tf.log(o_stdv + epsilon), axis=-1, keep_dims=True
    )

    # o_p is the probability density of the h-th component of the vote fr
    # (24, 6, 6, 1, 32, 16)
    o_p = o_p_unit0 + o_p_unit2

    # rr: (24, 6, 6, 288, 32, 1)cd

    zz = tf.log(o_activations + epsilon) + o_p
    rr = tf.nn.softmax(
        zz, dim=len(zz.get_shape()).as_list()-2
    )

    return rr
```

## Class capsules

Recall from a couple sections ago, the output of ConvCaps2 is feed into the Class capsules layer. The output poses for ConvCaps2 has a shape of (24, 4, 4, 32, 4, 4).

- batch size of 24
- 4x4 spatial output
- 32 output channels
- 4x4 pose matrix

Instead of using a 3x3 filter in ConvCaps2, Class capsules use a 1x1 filter. Also, instead of outputting a 2D spatial output (6x6 in ConvCaps1 and 4x4 in ConvCaps2), it outputs 10 capsules each representing one of the 10 classes in the MNist. The code structure for the class capsules is very similar to the conv\_capsule. It makes calls to compute the votes and then use EM routing to compute the capsule outputs.

```
def class_capsules(inputs, num_classes, iterations, batch_size, name):
    """
    :param inputs: ((24, 4, 4, 32, 4, 4), (24, 4, 4, 32))
    :param num_classes: 10
    :param iterations: 3
    :param batch_size: 24
    :param name:
    :return poses, activations: poses (24, 10, 4, 4), activation (24, 10)
    """

    inputs_poses, inputs_activations = inputs # (24, 4, 4, 32, 4, 4), (24, 4, 4, 32)

    inputs_shape = inputs_poses.get_shape()
    spatial_size = int(inputs_shape[1]) # 4
    pose_size = int(inputs_shape[-1]) # 4
    i_size = int(inputs_shape[3]) # 32

    # inputs_poses (24*4*4=384, 32, 16)
    inputs_poses = tf.reshape(inputs_poses, shape=[batch_size*spatial_size*pose_size, i_size])

    with tf.variable_scope(name) as scope:
        with tf.variable_scope('votes') as scope:
            # inputs_poses (384, 32, 16)
            # votes: (384, 32, 10, 16)
            votes = mat_transform(inputs_poses, num_classes, size=batch_size*spatial_size*pose_size)
            tf.logging.info(f"{name} votes shape: {votes.get_shape()}")

            # votes (24, 4, 4, 32, 10, 16)
            votes = tf.reshape(votes, shape=[batch_size, spatial_size, pose_size, num_classes, i_size])

            # (24, 4, 4, 32, 10, 16)
            votes = coord_addition(votes, spatial_size, spatial_size)
```

```

tf.logging.info(f"{name} votes shape with coord addition: {votes.get_shape()}")

with tf.variable_scope('routing') as scope:
    # beta_v and beta_a one for each output capsule: (1, 10)
    beta_v = tf.get_variable(
        name='beta_v', shape=[1, num_classes], dtype=tf.float32,
        initializer=initializers.xavier_initializer()
    )
    beta_a = tf.get_variable(
        name='beta_a', shape=[1, num_classes], dtype=tf.float32,
        initializer=initializers.xavier_initializer()
    )

    # votes (24, 4, 4, 32, 10, 16) -> (24, 512, 10, 16)
    votes_shape = votes.get_shape()
    votes = tf.reshape(votes, shape=[batch_size, votes_shape[1], votes_shape[2], votes_shape[3], votes_shape[4], votes_shape[5]])

    # inputs_activations (24, 4, 4, 32) -> (24, 512)
    inputs_activations = tf.reshape(inputs_activations, shape=[batch_size, inputs_activations.get_shape()[1], inputs_activations.get_shape()[2], inputs_activations.get_shape()[3]])

    # votes (24, 512, 10, 16), inputs_activations (24, 512)
    # poses (24, 10, 16), activation (24, 10)
    poses, activations = matrix_capsules_em_routing(
        votes, inputs_activations, beta_v, beta_a, iterations,
    )

    # poses (24, 10, 16) -> (24, 10, 4, 4)
    poses = tf.reshape(poses, shape=[batch_size, num_classes, poses.get_shape()[2], poses.get_shape()[3]])

    # poses (24, 10, 4, 4), activation (24, 10)
    return poses, activations

```

To integrate the spatial location of the predicted class in the class capsules, we add the scaled x, y coordinate of the center of the receptive field of each capsule in ConvCaps2 to the first two elements of the vote. This is called **Coordinate Addition**. According to the paper, this encourages the model to train the transformation matrix to produce values for those two elements to represent the position of the feature relative to the center of the capsule's receptive field. The motivation of the coordinate addition is to have the first 2 elements of the votes ( $v_{jk}^1, v_{jk}^2$ ) to predict the spatial coordinates (x, y) of the predicted class.

By retaining the spatial information in the capsule, we move beyond simply checking the presence of a feature. We encourage the system to verify the spatial relationship of features

to avoid adversaries. i.e. if the spatial orders of features are wrong, their corresponding votes should not match.

The pseudo code below illustrates the key idea. The pose matrices for the ConvCaps2 output has the shape of (24, 4, 4, 32, 4, 4). i.e. a spatial dimension of 4x4. We use the location of the capsule to locate the corresponding element in c1 below (a 4x4 matrix) which contains the scaled x coordinate of the center of the receptive field of the spatial capsule. We add this element value to  $v_{jk}^1$ . For the second element  $v_{jk}^2$  of the vote, we repeat the same process using c2.

```
# Spatial output of ConvCaps2 is 4x4
v1 = [[[8.], [12.], [16.], [20.]],
       [[8.], [12.], [16.], [20.]],
       [[8.], [12.], [16.], [20.]],
       [[8.], [12.], [16.], [20.]],
       ]

v2 = [[[8.], [8.], [8.], [8.]],
       [[12.], [12.], [12.], [12.]],
       [[16.], [16.], [16.], [16.]],
       [[20.], [20.], [20.], [20.]]
       ]

c1 = np.array(v1, dtype=np.float32) / 28.0
c2 = np.array(v2, dtype=np.float32) / 28.0
```

Here is the final code which looks far more complicated in order to handle any image size.

```
def coord_addition(votes, H, W):
    """Coordinate addition.

    :param votes: (24, 4, 4, 32, 10, 16)
    :param H, W: spatial height and width 4

    :return votes: (24, 4, 4, 32, 10, 16)
    """
    coordinate_offset_hh = tf.reshape(
        (tf.range(H, dtype=tf.float32) + 0.50) / H, [1, H, 1, 1, 1]
    )
    coordinate_offset_h0 = tf.constant(
        0.0, shape=[1, H, 1, 1, 1], dtype=tf.float32
    )
    coordinate_offset_h = tf.stack(
        [coordinate_offset_hh, coordinate_offset_h0] + [coordinate_offset_
    ) # (1, 4, 1, 1, 1, 16)
```

```

coordinate_offset_ww = tf.reshape(
    (tf.range(W, dtype=tf.float32) + 0.50) / W, [1, 1, W, 1, 1]
)
coordinate_offset_w0 = tf.constant(
    0.0, shape=[1, 1, W, 1, 1], dtype=tf.float32
)
coordinate_offset_w = tf.stack(
    [coordinate_offset_w0, coordinate_offset_ww] + [coordinate_offset_w
    ] # (1, 1, 4, 1, 1, 16)

# (24, 4, 4, 32, 10, 16)
votes = votes + coordinate_offset_h + coordinate_offset_w

return votes

```

## Spread loss

To compute the spread loss:

$$L = \sum_{i \neq t} (\max(0, m - (a_t - a_i)))^2$$

The value of  $m$  starts from 0.1. After each epoch, we increase the value by 0.1 until it reached the maximum of 0.9. This prevents too many dead capsules in the early phase of the training.

```

def spread_loss(labels, activations, iterations_per_epoch, global_step,
    """Spread loss

    :param labels: (24, 10] in one-hot vector
    :param activations: [24, 10], activation for each class
    :param margin: increment from 0.2 to 0.9 during training

    :return: spread loss
    """

    # Margin schedule
    # Margin increase from 0.2 to 0.9 by an increment of 0.1 for every
    margin = tf.train.pieceswise_constant(
        tf.cast(global_step, dtype=tf.int32),
        boundaries=[
            (iterations_per_epoch * x) for x in range(1, 8)
        ],

```



```

values=[
    x / 10.0 for x in range(2, 10)
]
)

activations_shape = activations.get_shape().as_list()

with tf.variable_scope(name) as scope:
    # mask_t, mask_f Tensor (?, 10)
    mask_t = tf.equal(labels, 1)      # Mask for the true label
    mask_i = tf.equal(labels, 0)      # Mask for the non-true label

    # Activation for the true label
    # activations_t (?, 1)
    activations_t = tf.reshape(
        tf.boolean_mask(activations, mask_t), shape=(tf.shape(activations)
    )

    # Activation for the other classes
    # activations_i (?, 9)
    activations_i = tf.reshape(
        tf.boolean_mask(activations, mask_i), [tf.shape(activations)
    )

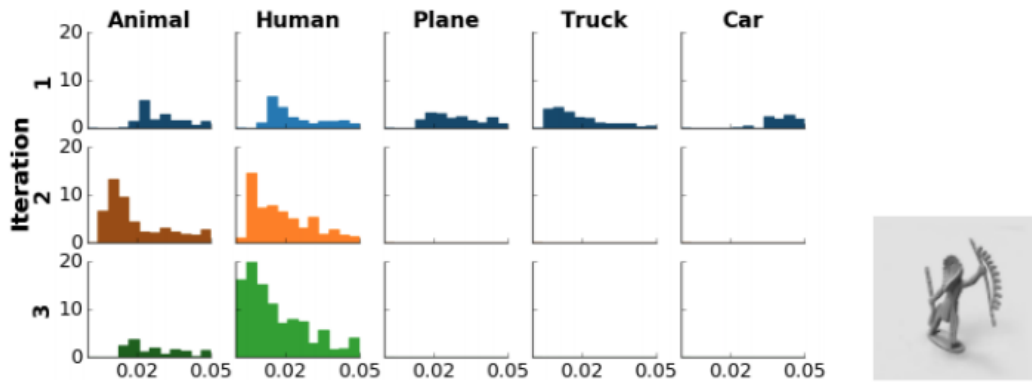
    l = tf.reduce_sum(
        tf.square(
            tf.maximum(
                0.0,
                margin - (activations_t - activations_i)
            )
        )
    )
    tf.losses.add_loss(l)

    return l

```

## Result

The following is the histogram of distances of votes to the mean of each of the 5 final capsules after each routing iteration. Each distance point is weighted by its assignment probability. With a human image as input, we expect, after 3 iterations, the difference is the smallest (closer to 0) for the human column. (any distances greater than 0.05 will not be shown here.)



The error rate for the Capsule network is generally lower than a CNN model with similar number of layers as shown below.

Test set	Azimuth		Elevation	
	CNN	Capsules	CNN	Capsules
Novel viewpoints	20%	13.5%	17.8%	12.3%
Familiar viewpoints	3.7%	3.7%	4.3%	4.3%

(Source from the Matrix capsules with EM routing paper)

The core idea of FGSM (fast gradient sign method) adversary is to add some noise on every step of optimization to drift the classification away from the target class. It slightly modifies the image to maximize the error based on the gradient information. Matrix routing is shown to be less vulnerable to FGSM adversaries comparing to CNN.

## Visualization

The pose matrices in Class Capsules are interpreted as the latent representation of the image. By adjusting the first 2 dimension of the pose and reconstructing it through a decoder (similar to the one in the previous capsule article), we can visualize what the Capsule Network learns for the MNist data.



(Source from the Matrix capsules with EM routing paper)

Some digits are slightly rotated or moved which demonstrate the Class Capsules are learning the pose information of the MNist dataset.

## Credits

Part of the code implementation in computing the vote and the EM-routing is modified from [Guang Yang](#) or [Suofei Zhang](#) implementations. Our source code is located at the [github](#). To run,

- Configure the `mnist_config.py` and `cap_config.py` according to your environment.
- Run the `download_and_convert_mnist.py` before running `train.py`.


Please note that the source code is for illustration purpose with no support at this moment. Readers may refer to other implementations if you have problems to trouble shooting any issues that may come up. Zhang's implementation seems easier to understand but Yang implementation is closer to the original paper.

34 Comments

jhui

 Login ▾

 Recommend 7

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



**William Woof** • 8 days ago

Nice write-up!

You might be interested in a recent memo from MIT's Brain's Minds and Machines Lab: <https://cbmm.mit.edu/public...>

I don't quite fully understand their exact architecture, but conceptually it seems very similar to the capsules idea (although, they claim otherwise, based on their understanding on capsules). While the paper isn't particularly good, I found it fairly inspirational in terms of looking beyond specific architectures.

In essence, the core of the idea both methods rely on sets of objects (a.k.a capsules), which have a number of features, which predict a new set of objects. The exact mechanisms are rather different, but conceptually they both rely on voting for their choices of fixed set of an output objects.

I wonder if this mechanism could be replaced by a more general set2set transformation (e.g. Transformer <https://arxiv.org/abs/1706.....>). The big question is then if the routing process itself is an important part of generalisation, or if it's just a convenient mechanism for predicting an output set of parameterized objects.

Certainly it's possible to distinguish complex compositional structures based on spatial elements without any special routing, as evidenced by approaches to learning on point clouds (e.g. <https://arxiv.org/abs/1703.06114>), so it may be that all that's missing form

conventional CNN architectures is positional information from the different filters.

^ | v • Reply • Share ›



**Jonathan Hui** Mod ➔ William Woof • 7 days ago

Thanks for sharing your thoughts.

The major difference of capsule and objects seems to be (according to MIT paper): "A "capsule" layer would have a dense grid of capsules detecting objects at a grid of locations." That claim is interesting if you look at the attention approach in the 2nd paper. Unfortunately, the MIT paper does not have enough details on how objects are extracted other than sliding a spatial filter in capsule. So MIT paper adopt a different approach in step 1 other than using spatial filters:

1. Extract objects
2. Voting
3. Routing

CNN detects features. With current GPU speed, it solves a lot of problem. Adding all these property detection increases complexity. So I think we may ask how much complexity we can afford. The anchor points in Faster RCNN or YOLO seems like an effort to put back some kind of spatial location back to the CNN. The increase complexity is not that bad. So if you can add back the location information into CNN with the same order of complexity of a CNN, that can be interesting.

p.s.The link for the last 2 papers have an extra ")" that break the link.

^ | v • Reply • Share ›



**Tester\_asker** • 16 days ago

Thanks for the post: Ironically the only thing I am not quite sure about is reshaping using convolution with almost all zeros filter. (tile subfunction).What is the meaning of the non zero coordinates of the filter?

Thanks

^ | v • Reply • Share ›



**Jonathan Hui** Mod ➔ Tester\_asker • 13 days ago

Just FYI:

Thanks Ouyang for pointing out Sara's repository. <https://github.com/Sarasra/...>

She has a convolution capsule implementation. But this gears towards replacing the fully connected capsule with a convolution capsule in the first paper. So it does not have the code in doing the tile function that you ask. Her coding looks pretty easy to understand. Hope she will release code for the second paper soon.

^ | v • Reply • Share ›



**Jonathan Hui** Mod ➔ Tester\_asker • 15 days ago

Let assume the spatial dimension is 14x14 and we are applying a 3x3 filter with padding, strides 1 and 32 input & 32 output capsules. For simplicity, let forget the pose matrix and just output 1 scalar value.

So the input is (14, 14, 32). We want to convert the input to d: (14, 14, 3, 3, 32)

"Understanding Matrix capsules with EM Routing (Based on Hinton's Capsule Networks)"  
 where the extra 3x3 dimension holds all the 9 neighboring points at location (i, j).  
 E.g. `d[i, j, 0, 0, :]` hold the upper left neighbor at spatial location (i, j).

To do that, we apply `tf.nn.depthwise_conv2d` (not the regular convolution). The `tile_filter` is like a one-hot-vector (that is why so many zero) to pick which neighbor to put in `d`. Once we have `d`, we can just do a simple dot product with the weight to create the vote.

It is just tedious.

^ | v • Reply • Share ›



**Tester\_asker** → Jonathan Hui • 15 days ago

Thank you very much for your quick answer: so assuming uneven filter of 1x3

I should do something like:

`kernel=3`

`second_d=1`

```
tile_filter = np.zeros(shape=[kernel, second_d, input_shape[3], # K S
kernel * second_d], dtype=np.float32)
```

```
glob=0
```

```
for i in range(kernel): #kernel
```

```
for j in range(second_d):
```

```
tile_filter[i, j, :, glob] = 1.0 #second d
```

```
glob+=1
```

```
print ("FILTERSHAPE")
```

```
print (tile_filter.shape)
```

or am I messing up the dimensionality yet again ?

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Tester\_asker • 15 days ago

A word of cautions: when I developed the code, some part of the code assume a square filter. However, It should not be hard to extend it to non-square kernels.

Is your code doing a 3x1 kernel instead? Hard for me to check your program. But my 2 cents is to write a simple routine with a 5x5 images. Apply the `tile_filter` and see if the extra dimension created has propagate with your neighboring points in "output".

^ | v • Reply • Share ›



**Tester\_asker** → Jonathan Hui • 15 days ago

I tried to write something based on all of the 3 implementations I found on the net (it's a pity Sarah and co. did not release their own implementation, it would have been interesting to check) : I do not get any dimensionality mismatching (I am not quite sure what you mean by propagate with your neighbouring points), but I generally feel rather uncomfortable when I just modify exiting code without thinking so I wanted some kind of a sanity check (it is 1x3 kernel, so it's kind of a 1d convolution extension of the model.)

Pose matrix in case of 1d doesn't make a lot of geometrical sense, EM routing however is an interesting attention-like mechanism I wanted to try to use on something other than images.

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Tester\_asker • 15 days ago

I was asking are you doing a 3x1 or 1x3 kernel? From your code sample, it looks like a 3x1.

This implementation uses non-square kernel <https://github.com/gyang274...> but will be even harder to understand.

When you convert input in(14, 14, 32, 4, 4) to d(14, 14, 3, 3, 32, 4, 4). Those extra dimensions 3x3 hold the 8 neighboring points + itself at location (i, j)

So my last comment is about running your code and see if say d[ 7, 7, 0, 0, 32, 4, 4 ] really contain the upper left neighbor value. ie. in[ 6, 6, 32, 4, 4 ).

Good luck with your approach. You do make me to think about what is the context of attention with capsule.

^ | v • Reply • Share ›



**Tester\_asker** → Jonathan Hui • 15 days ago

I see, thank you very much. I like when people propose new architectures instead of just stacking layers and writing your own modified version is the best way to make sure your understand new architectures correctly, I am a worse coder than I am applied mathematician so I really appreciate other people posting their implementations on-line.

There are a number of things in this paper that do not quite make sense from my understanding of theory behind it:

For example I honestly wonder why the results tend to drop once you go up to 5 routing operations (for forward pass you are working with a fixed set of data-points so, assuming the low level feature agreement hypothesis is true, it should actually result in better accuracy not worse one). Are we still getting relevant for classification information from "wrong cluster" capsules ?

I am also not super convinced on their system being better when it comes to adversarial attacks: attacks are kind of specific to pure gradient methods, and created specifically to full gradient methods models. As such the fact that a hybrid system does better doesn't seem very telling.

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Tester\_asker • 14 days ago

**Jonathan Hui** Mod → Tester\_asker • 14 days ago

For the iteration, you are right. They are getting information from the "wrong cluster" children capsules. Their implementation is not exactly EM clustering.

^ | v • Reply • Share ›

**Jonathan Hui** Mod → Tester\_asker • 14 days ago

On the adversarial attack, my belief is that we are still detect many low level features in the last few layers. Knowing the gradient, we know the secret sauce on what is the minimum feature change results in largest shift in classification. Since there are many tiny features, we can steal a small amount from each of them without causing major visual difference. Also the latent space for different classes are too close together without gap. If we know the gradient, we can move from one class to another with little visual changes. In capsule, if we can train  $W$  in routing to pull clusters far away from each other, we will need bigger changes in input for the necessary changes on the output. But this is likely not the main objective for the pose matrix though. But at the end, if we are still detecting many low level features, it would not help.

For the iterations, there are people is bother by it too.

^ | v • Reply • Share ›

**D. Lee** • 21 days ago

Hi. Thanks for your kind explanation.

I have a question about coordinate addition.

When i read the paper, I understand it concatenation on vote matrix.

However, in your codes, it is implemented with addition operation.

Could you explain about it more?

^ | v • Reply • Share ›

**Jonathan Hui** Mod → D. Lee • 21 days ago

Somehow I got the same impression as you and one implementation actually changes the dimension of the vote in Class capsules from 16 to 18. But then I re-read the paper again:

... add the scaled coordinate (row, column) of the center of the receptive field of each capsule to the first two elements of the right-hand column of its vote matrix ... This should encourage the shared final transformations to produce values for those two elements that represent the fine position of the entity relative to the center of the capsule's receptive field...

I suspect it should remain at 16 because the right-hand column of a pose matrix usually represent the  $x, y$  transformation. I cannot find the old version of the paper. The paper author Hinton, etc has made some changes to the paper since Nov. I wonder whether it was one of the change. Some other changes include they fix the notation of the equation which is much clear now and an appendix.

^ | v • Reply • Share ›

**Shakin B. Namin** • a month ago



**Shahin R. Namin** • a month ago

Thank you for clarification of the paper Jonathan. There is one equation I do not understand. When computing the  $\text{cost}_h$ , how does  $\sum r_i * (v - \mu)^2$  change to  $\sigma_h^2 * \sum r_i$ ? I would appreciate it if you explain this.

^ | v • Reply • Share ›

**Jonathan Hui** Mod → Shahin R. Namin • a month ago

This should be the definition of standard deviation. That part of the equation may need to multiple it by N but it does not matter eventually because it is simplified to a constant k.

^ | v • Reply • Share ›

**Ali Mirzaei** • 2 months ago

Thanks for your great and helpful post.

"If  $\text{cost}_h$  is low, capsule i is more likely to activate the face capsule."

I don't understand the above sentence because there is a summation over i (capsules of previous layer) in  $\text{cost}_h$ . Could you please explain more?

^ | v • Reply • Share ›

**Jonathan Hui** Mod → Ali Mirzaei • 2 months ago

Thanks for pointing that out. I put some clarification in the article.

^ | v • Reply • Share ›

**Ali Mirzaei** → Jonathan Hui • 2 months ago

Thank you! It makes sense now!

Could you please also explain this sentence?

"-b represents the cost of describing the mean of capsule c"

^ | v • Reply • Share ›

**Jonathan Hui** Mod → Ali Mirzaei • 2 months ago

What! "where '-b' represents the cost of describing the mean of capsule c" is not clear to you! :-) I think for people not engaged in the original research (or terminology) will have tough time to understand it: not just you but me too. I copy the original "statement" from the paper without much of my interpretation because there is not enough description regarding the definition of "describing the mean" to tell whether my intuition is correct. Since you ask, here is my thought in looking at the activation function from a different perspective:

If you look from the perspective of "routing-by-agreement", we start turning capsule c "on" if  $\text{cost}_h$  is below the threshold "b". If the cost are within a threshold, we activate c. i.e. if the sub-components' votes cluster close enough to the Gaussian model of the capsule c, we turn c "on". You may ask how to calculate "b". I treat "b" as if the bias in a neuron and therefore it is trainable with backpropagation.

^ | v • Reply • Share ›



**Zhenhua Chen** • 2 months ago

Thanks for sharing this, Hui! It makes my life easier:)

I am confused about one statement in the paper: "It uses a vector of length  $n$  rather than a matrix with  $n$  elements to represent a pose, so its transformation matrices have  $n^2$  parameters rather than just  $n$ ." My understanding is that there is no difference between "vector capsule" and "matrix capsule". For example, the matrix capsule is just a  $4 \times 4 + 1 = 17$  dimension vector capsule. Could you help me figure this out? Thanks!

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Zhenhua Chen • 2 months ago

To transform a 16-D capsule to a 16-D vector, the transformation matrix is  $16 \times 16$ . ( $1 \times 16 \times 16 \times 16 \rightarrow 1 \times 16$ ) To transform a  $4 \times 4$  capsule to a  $4 \times 4$  pose, we take a  $4 \times 4$  matrix. So the first paper is  $n^2$  and the second paper is  $n$ .

^ | v • Reply • Share ›



**Kun Ouyang** → Jonathan Hui • 13 days ago

Hi Jonathan, I still doubt this statement. For 16-D capsule to 16-D vector, for "transformation" per se, why can't I do a element-wise multiplication with a  $(1 \times 16)$   $W$ ? Actually this is also how Sara has done in their released source code of this paper. <https://github.com/Sarasra/...>

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Kun Ouyang • 13 days ago

The first paper uses element-wise multiplication and implementations for the second paper I see so far use  $4 \times 4$  matrix transformation. The  $4 \times 4$  matrix transformation is more coherent with Hinton's concept of Inverse Graphics.

Chen is asking why the second paper claim it can use less parameters for  $W$ . If it is a  $4 \times 4$  matrix transformation, then instead of a  $16 \times 16$  matrix in element-wise multiplication, we just need a  $4 \times 4$  matrix. So I think we are asking 2 separate questions here. But thanks for pointing out Sara's implementation. But the implementation seems replacing the fully connected capsule layer in paper 1 with the convolution capsule. It makes a lot of sense because it is much scalable.

Will Sara uses a piece wise implementation for the second paper? I think you have to ask Sara instead. :-) But the output\_atoms is 8 in her code. So I suspect her code is a demonstration of the first paper. Anyway, we should give a big thanks for Sara to release the code.

```
capsule1 = layers.conv_slim_capsule(
    input_tensor,
    input_dim=1,
    output_dim=self._hparams.num_prime_capsules,
    layer_name='conv_capsule1',
    num_routing=1,
```

input\_atoms=256,  
output\_atoms=8,  
^ | v • Reply • Share ›



**Kun Ouyang** → Jonathan Hui • 12 days ago

Thank you for your kindly replies:)

^ | v • Reply • Share ›



**Kun Ouyang** → Jonathan Hui • 12 days ago

Thank you very much for your articulations. Actually the thing confusing me is what's the benefit of using the matrix over a vector. I don't see the correlation between entries within the matrix, so I guess each entry encode something separately (e.g., different rotation of a noise), but the same thing can be achieved by a vector capsule. So, apart from the reduced complexity (i.e.,  $N^2$  to  $N$ ), what else there?

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Kun Ouyang • 12 days ago

I suspect there is a motivation for Hinton to demonstrate his idea of Inverse Graphics. Ideally, our view of world can be represented as objects with their pose matrices. Just like the graphics programming. You can drag the object and rotate it in 3-D.

There is a specific reason to pick smallNORB for dataset because the major variants are the viewpoints. The dataset is a good fit to train the pose matrix. We use 4x4 transformation, so we add constraints such that we hope the model will eventually learn the viewpoints. Of course 16x16 is a superset of 4x4. But when you train the system in practice, it can be hard!

But back to the real questions, is a vector or a matrix is better? For smallNORB, it will be pose matrix. But for different datasets, the variants may not just limit to viewpoints. Like handwritten, we also need to take care of stroke width. Color information can be important for some dataset. That will require an vector. Do we need a pose matrix for those situation? Some people ask why after 5 routing iterations, the accuracy drops. That bother me and I hope the authors can give us more insights. But I do suspect the pose matrix and the transformation may pose a role there. However, I am not ready to elaborate that without more findings. But if the capsule really works, we should see significant work in the transformation and the clustering. We want transformation to cluster data well. So your question will likely turn to more research.

^ | v • Reply • Share ›



**Zhenhua Chen** → Jonathan Hui • 2 months ago

I see, thanks! I also have some other confusions, could you please also help me out?

1. In the first paper, why  $b_{ij} = b_{ij} + u_{ji} * b_{ij}$  is updated in an accumulated way? My intuition would be  $b_{ij} = u_{ji} * b_{ij}$ .

2. In both the first paper and the second paper, the routing process is applied for three times. However, in the second paper, when the number of routing iterations is increased to 5, the performance becomes worse. Do you have an intuitive explanation?

3. For me, the intuitive way of implementing the routing algorithm would be through learning which is not adopted by both papers. I am also curious about this:)

^ | v • Reply • Share ›



**Jonathan Hui** Mod → Zhenhua Chen • 2 months ago

1.  $b = b + u \cdot v$ .  $v$  is an estimated value depends on  $b$ . The method uses iterations, and hopefully it can converge to a "better" (accurate) value.

2. I will restrain myself from speculation here since there is not much detail there.

3. Backpropagation in a global scale vs routing-by-agreement in a local scale? Or you can propose a second cost function to train  $r$ . I think the Capsule network can have many direction to explore from now. I believe the first priority is to scale the solution first.

^ | v • Reply • Share ›



**Zhenhua Chen** → Jonathan Hui • 2 months ago

Thank you, Jonathan! For 1, I am still confused why this iterative