Capsule Networks without Routing Procedures

Anonymous authors
Paper under double-blind review

Abstract

We propose Pure CapsNets (P-CapsNets) without routing procedures. Specifically, we make three modifications to CapsNets. First, we remove routing procedures from CapsNets based on the observation that the coupling coefficients can be learned implicitly. Second, we replace the convolutional layers in CapsNets to improve efficiency. Third, we package the capsules into rank-3 tensors to further improve efficiency. The experiment shows that P-CapsNets achieve better performance than CapsNets with varied routine procedures by using significantly fewer parameters on MNIST&CIFAR10. The high efficiency of P-CapsNets is even comparable to some deep compressing models. For example, we achieve more than 99% percent accuracy on MNIST by using only 3888 parameters. We visualize the capsules as well as the corresponding correlation matrix to show a possible way of initializing CapsNets in the future. We also explore the adversarial robustness of P-CapsNets compared to CNNs.

1 Introduction

Capsule Networks, or CapsNets, have been found to be more efficient for encoding the intrinsic spatial relationships among features (parts or a whole) than normal CNNs. For example, the CapsNet with dynamic routing (Sabour et al. (2017)) can separate overlapping digits accurately, while the CapsNet with EM routing (Hinton et al. (2018)) achieves lower error rate on smallNORB (LeCun et al. (2004)). However, the routing procedures of CapsNets (including dynamic routing (Sabour et al. (2017)) and EM routing (Hinton et al. (2018))) are computationally expensive. Several modified routing procedures have been proposed to improve the efficiency (Zhang et al. (2018); Choi et al. (2019); Li et al. (2018)), but they sometimes do not "behave as expected and often produce results that are worse than simple baseline algorithms that assign the connection strengths uniformly or randomly" (Paik et al. (2019)).

Even we can afford the computation cost of the routing procedures, we still do not know whether the routing numbers we set for each layer serve our optimization target. For example, in the work of Sabour et al. (2017), the CapsNet models achieve the best performance when the routing number is set to 1 or 3, while other numbers cause performance degradation. For a 10-layer CapsNet, assuming we have to try three routing numbers for each layer, then 3^{10} combinations have to be tested to find the best routing number assignment. This problem could significantly limit the scalability and efficiency of CapsNets.

Here we propose P-CapsNets, which resolve this issue by removing the routing procedures and instead learning the coupling coefficients implicitly during capsule transformation (see Section 3 for details). Moreover, another issue with current CapsNets is that it is common to use several convolutional layers before feeding these features into a capsule layer. We find that using convolutional layers in CapsNets is not efficient, so we replace them with capsule layers. Inspired by Hinton et al. (2018), we also explore how to package the input of a CapsNet into rank-3 tensors to make P-CapsNets more representative. The capsule convolution in P-CapsNets can be considered as a more general version of 3D convolution. At each step, 3D convolution uses a 3D kernel to map a 3D tensor into a scalar (as Figure 1 shows) while the capsule convolution in Figure 2 adopts a 5D kernel to map a 5D tensor into a 5D tensor.

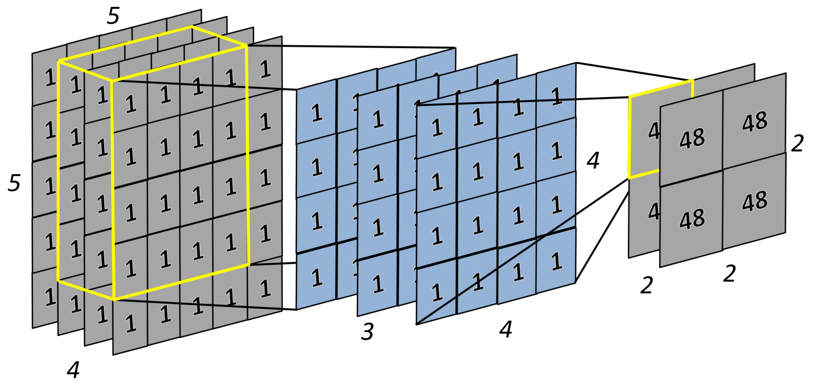


Figure 1: 3D convolution: tensor-to-scalar mapping. The shape of input is $8 \times 8 \times 4$. The shape of 3D kernel is $4 \times 4 \times 3$. As a result, the shape of output is $5 \times 5 \times 3$. Yellow area shows current input area being convolved by kernel and corresponding output.

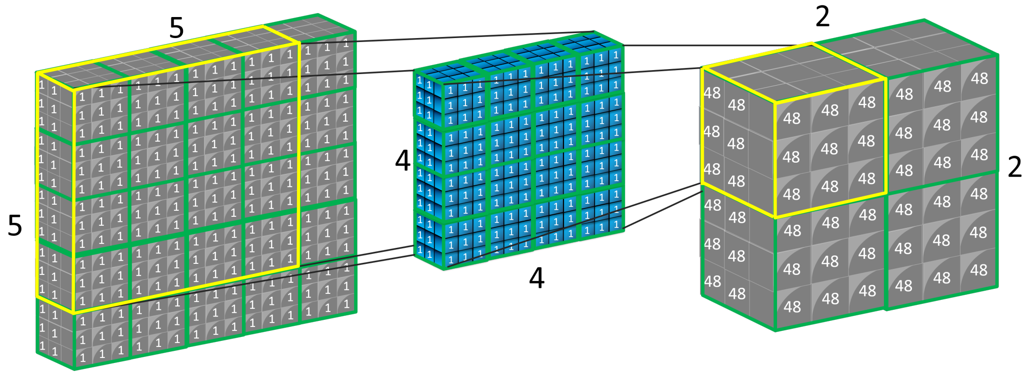


Figure 2: Capsule convolution in P-CapsNets: tensor-to-tensor mapping. The input is a tensor of 1's which has a shape of $1 \times 5 \times 5 \times (3 \times 3 \times 3)$ (correspond to the the input channel, input height, input width, first capsule dimension, second capsule dimension, and third capsule dimension, respectively). The capsule kernel is also a tensor of 1's which has a shape of $4 \times 4 \times 1 \times 1 \times (3 \times 3 \times 3)$ — kernel height, kernel width, number of input channel, number of output channel, and the three dimensions of the 3D capsule. As a result, we get an output tensor of 48's which has a shape of $1 \times 2 \times 2 \times (3 \times 3 \times 3)$. Yellow areas show current input area being convolved by kernel and corresponding output.

2 Related Work

CapsNets (Sabour et al. (2017)) organize neurons as capsules to mimic the biological neural systems. One key design of CapsNets is the routing procedure which can combine lower-level features as higher-level features to better model hierarchical relationships. There have been many papers on improving the expensive routing procedures since the idea of CapsNets was proposed. For example, Zhang et al. (2018) improves the routing efficiency by 40% by using weighted kernel density estimation. Choi et al. (2019) propose an attention-based routing procedure which can accelerate the dynamic routing procedure. However, Paik et al. (2019) have found that these routing procedures are heuristic and sometimes perform even worse than random routing assignment.

Incorporating routing procedures into the optimization process could be a solution. Wang & Liu (2018) treats the routing procedure as a regularizer to minimize the clustering loss between adjacent capsule layers. Li et al. (2018) approximates the routing procedure with master and aide interaction to ease the computation burden. Chen et al. (2019) incorporates the routing procedure into the training process to avoid the computational complexity of dynamic routing.

Here we argue that from the viewpoint of optimization, the routing procedure, which is designed to acquire coupling coefficients between adjacent layers, can be learned and optimized implicitly, and may thus be unnecessary. This approach is different from the above CapsNets which instead focus on improving the efficiency of the routing procedures, not attempting to replace them altogether.

3 How P-CapsNets work

We now describe our proposed P-CapsNet model in detail. We describe the three key ideas in the next three sections: (1) that the routing procedures may not be needed, (2) that packaging capsules into higher-rank tensors is beneficial, and (3) that we do not need convolutional layers.

3.1 Routing procedures are not necessary

The primary idea of routing procedures in CapsNets is to use the parts and learned part-whole relationship to vote for objects. Intuitively, identifying an object by counting the votes makes perfect sense. Mathematically, routing procedures can also be considered as linear combinations of tensors. This is similar to the convolution layers in CNNs in which the basic operation of a convolutional layer is linear combinations (scaling and addition),

$$v_j = \sum_i W_{ij} u_i. \tag{1}$$

where v_j is the jth output scalar, u_i is the ith input scalar, and W_{ij} is the weight.

The case in CapsNets is a bit more complex since the dimensionalities of input and output tensors between adjacent capsule layers are different and we can not combine them directly. Thus we adopt a step to transform input tensors ($\mathbf{u_i}$) into intermediate tensors ($\mathbf{u_{j|i}}$) by multiplying a matrix ($\mathbf{W_{ij}}$). Then we assign each intermediate tensors ($\mathbf{u_{j|i}}$) a weight c_{ij}, and now we can combine them together,

$$\mathbf{v_j} = \sum_i c_{ij} \mathbf{W_{ij}} \mathbf{u_i}. \tag{2}$$

where c_{ij} are called coupling coefficients which are usually acquired by a heuristic routing procedure (Sabour et al. (2017); Hinton et al. (2018)).

In conclusion, CNNs do linear combinations on scalars while CapsNets do linear combinations on tensors. Using a routing procedure to acquire linear coefficients makes sense. However, if Equation 2 is rewritten as,

$$\mathbf{v_j} = \sum_i c_{ij} \mathbf{W_{ij}} \mathbf{u_i} = \sum_i \mathbf{W_{ij}^{'}} \mathbf{u_i}. \tag{3}$$

then from the viewpoint of optimization, it is not necessary to learn or calculate c_{ij} and $\mathbf{W_{ij}}$ separately since we can learn $\mathbf{W_{ij}^{'}}$ instead. In other words, we can learn the c_{ij} implicitly by learning $\mathbf{W_{ij}^{'}}$. Equation 2 is the basic operation of P-CapsNets only we extend it to the 3D case; please see Section 3.2 for details.

By removing routing procedures, we no longer need an expensive step for computing coupling coefficients. At the same time, we can guarantee the learned $\mathbf{W_{ij}^{'}} = c_{ij}\mathbf{W_{ij}}$ is optimized to serve a target, while the good properties of CapsNets could still be preserved (see section 4 for details). We conjecture that the strong modeling ability of CapsNets come from this tensor to tensor mapping between adjacent capsule layers.

From the viewpoint of optimization, routing procedures do not contribute a lot either. Taking the CapsNets in (Sabour et al. (2017)) as an example, the number of parameters in the transformation operation is $6 \times 6 \times 32 \times (8 \times 16) = 147,456$ while the number of parameters in the routing operation equals to $6 \times 6 \times 32 \times 10 = 11,520$ — the "routing parameters" only represent 7.25% of the total parameters and are thus negligible compared to the "transformation parameters." In other words, the benefit from routing procedures may be limited, even though they are the computational bottleneck.

Equation 1 and Equation 3 have a similar form. We argue that the "dimension transformation" step of CapsNets can be considered as a more general version of convolution. For example, if each

3D tensor in P-CapsNets becomes a scalar, then P-CapsNets would degrade to normal CNNs. As Figure 4 shows, the basic operation of 3D convolution is $f : \mathbf{U} \in \mathbb{R}^{g\times m\times n} \rightarrow v \in \mathbb{R}$ while the basic operation of P-CapsNet is $f : \mathbf{U} \in \mathbb{R}^{g\times m\times n} \rightarrow \mathbf{V} \in \mathbb{R}^{g\times m\times p}$.

3.2 Packaging capsules into higher rank tensors is helpful to save parameters

The capsules in (Sabour et al. (2017)) and (Hinton et al. (2018)) are vectors and matrices. For example, the capsules in Sabour et al. (2017) have dimensionality $1152 \times 10 \times 8 \times 16$ which can convert each 8-dimensional tensor in the lower layer into a 16-dimensional tensor in the higher layer ($32 \times 6 \times 6 = 1152$ is the input number and 10 is the output number). We need a total of $1152 \times 10 \times 8 \times 16 = 1474560$ parameters. If we package each input/output vector into 4×2 and 4×4 matrices, we need only $1152 \times 10 \times 2 \times 4 = 92160$ parameters. This is the policy adopted by Hinton et al. (2018) in which 16-dimensional tensors are converted into new 16-dimensional tensors by using 4×4 tensors. In this way, the total number of parameters is reduced by a factor of 15.

In this paper, the basic unit of input ($\mathbf{U} \in \mathbb{R}^{g\times m\times n}$), output ($\mathbf{V} \in \mathbb{R}^{g\times m\times p}$) and capsules ($\mathbf{W} \in \mathbb{R}^{g\times n\times p}$) are all rank-3 tensors. Assuming the kernel size is ($\mathbf{kh} \times \mathbf{kw}$), the input capsule number (equivalent to the number of input feature maps in CNNs) is $\mathbf{in}$. If we extend Equation 3 to the 3D case, and incorporate the convolution operation, then we obtain,

$$\mathbf{V} = [\mathbf{V}_{0,:,:}, \mathbf{V}_{1,:,:}, \ldots, \mathbf{V}_{g,:,:}] = \sum_{i}^{in}\sum_{j}^{kh}\sum_{k}^{kw}\left([\mathbf{W}_{0,i,j,k}\mathbf{U}_{0,i,j,k}, \mathbf{W}_{1,i,j,k}\mathbf{U}_{1,i,j,k}, \ldots, \mathbf{W}_{g,i,j,k}\mathbf{U}_{g,i,j,k}]\right), \tag{4}$$

which shows how to obtain an output tensor from input tensors in the previous layer in P-CapsNets.

Assuming a P-CapsNet model is supposed to fit a function $f : \mathbb{R}^{W\times H\times 3} \rightarrow \mathbb{R}$, the ground-truth label is $y \in \mathbb{R}$ and the loss function $\mathcal{L} = ||f - y||$. Then in back-propagation, we calculate the gradients with respect to the output $\mathbf{U}$ and with respect to the capsules $\mathbf{W}$,

$$\nabla_{\mathbf{U}}\mathcal{L} = \left[\nabla_{\mathbf{U}_{0,:,:}}\mathcal{L}, \ldots, \nabla_{\mathbf{U}_{g,:,:}}\mathcal{L}\right] = \sum_{i}^{in}\sum_{j}^{kh}\sum_{k}^{kw}\left([\mathbf{W}_{0,i,j,k}\nabla_{\mathbf{V}_{0,:,:}}\mathcal{L}, \ldots, \mathbf{W}_{g,i,j,k}\nabla_{\mathbf{V}_{g,:,:}}\mathcal{L}]\right), \tag{5}$$

$$\nabla_{\mathbf{W}}\mathcal{L} = \left[\nabla_{\mathbf{W}_{0,:,:}}\mathcal{L}, \ldots, \nabla_{\mathbf{W}_{g,:,:}}\mathcal{L}\right] = \sum_{i}^{in}\sum_{j}^{kh}\sum_{k}^{kw}\left([\mathbf{U}_{0,i,j,k}\nabla_{\mathbf{V}_{0,:,:}}\mathcal{L}, \ldots, \mathbf{U}_{g,i,j,k}\nabla_{\mathbf{V}_{g,:,:}}\mathcal{L}]\right). \tag{6}$$

The advantage of folding capsules into high-rank tensors is to reduce the computational cost of dimension transformation between adjacent capsule layers. For example, converting a 1×16 tensor to another 1×16 tensor, we need $16 \times 16 = 256$ parameters. In contrast, if we fold both input/output vectors to three-dimensional tensors, for example, as $2 \times 4 \times 2$, then we only need 16 parameters (the capsule shape is $2 \times 4 \times 2$). For the same number of parameters, folded capsules might be more representative than unfolded ones. Figure 2 shows what happens in one capsule layer of P-CapsNets in detail.

3.3 We can build a pure CapsNet without using any convolutional layers

It is a common practice to embed convolutional layers in CapsNets, which makes these CapsNets a hybrid network with both convolutional and capsule layers (Sabour et al. (2017); Hinton et al. (2018); Chen et al. (2019)). One argument for using several convolutional layers is to extract low level, multi-dimensional features. We argue that this claim is not so persuasive based on two observations, **1)**. The level of multi-dimensional entities that a model needs cannot be known in advance, and it does not matter, either, as long as the level serves our target; **2)**. Even if a model needs a low level of multi-dimensional entities, the capsule layer can still be used since it is a more general version of a convolutional layer.

Based on the above observations, we build a "pure" CapsNet by using only capsule layers. One issue of P-CapsNets is how to process the input if they are not high-rank tensors. Our solution is simply adding new dimensions. For example, the first layer of a P-CapsNet can take $1 \times w \times h \times (1 \times 1 \times 3)$

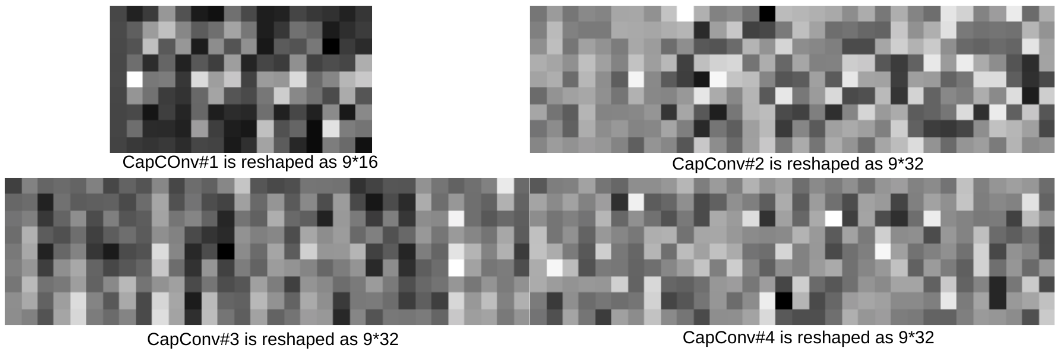

CapCOnv#1 is reshaped as 9*16
CapConv#2 is reshaped as 9*32
CapConv#3 is reshaped as 9*32
CapConv#4 is reshaped as 9*32

Figure 3: Visualization of filters in P-CapsNets. Top: conv1 layer, conv2 layer. Bottom: conv3 layer, conv4 layer.

tensors as the input (colored image), and take $1 \times w \times h \times (1 \times 1 \times 1)$ tensors as the input for gray-scale images.

In conclusion, P-CapsNets make three modifications over CapsNets (Sabour et al. (2017)). First, we remove the routing procedures from all the capsule layers. Second, we replace all the convolutional layers with capsule layers. Third, we package all the capsules and input/output as rank-3 tensors to save parameters. We keep the loss and activation functions the same as in the previous work. Specifically, for each capsule layer, we use the squash function $\mathbf{V} = \left(1 - \frac{1}{\mathbf{e}^{\|\mathbf{V}\|}}\right) \frac{\mathbf{V}}{\|\mathbf{V}\|}$ in (Edgar et al. (2017)) as the activation function. We also use the same margin loss function in (Sabour et al. (2017)) for classification tasks,

$$\mathcal{L}_{\mathbf{k}} = \mathcal{T}_k * \max(0,\ m^+ - \|\mathbf{V}_k\|)^2 + \lambda * (1 - \mathcal{T}_k) * \max(0,\ \|\mathbf{V}_k\| - m^-)^2, \tag{7}$$

where $\mathcal{T}_k$ = 1 iff class k is present, and $m^+ \geqslant 0.5$, $m^- \geqslant 0$ are meta-parameters that represent the threshold for positive and negative samples respectively. λ is a weight that adjust the loss contribution for negative samples.

4 Experiments

We test our P-CapsNets model on MNIST and CIFAR10. P-CapsNets show higher efficiency than CapsNets Sabour et al. (2017) with various routing procedures as well as several deep compressing neural network models Wu (2018); Yang et al. (2015); Grosse & Martens (2016).

For MNIST, P-CapsNets#0 achieve better performance than CapsNets Sabour et al. (2017) by using 40 times fewer parameters, as Table 1 shows. At the same time, P-CapsNets#3 achieve better performance than Matrix CapsNets Hinton et al. (2018) by using 87% fewer parameters. Phaye et al. (2018) is the only model that outperforms P-CapsNets, but uses 80 times more parameters.

Since P-CapsNets show high efficiency, it is interesting to compare P-CapsNets with some deep compressing models on MNIST. We choose five models that come from three algorithms as our baselines. As Table 2 shows, for the same number of parameter, P-CapsNets can always achieve a lower error rate. For example, P-CapsNets#2 achieves 99.15% accuracy by using only 3,888 parameters while the model (Wu (2018)) achieves 98.44% by using 3,554 parameters. For P-CapsNet structures in Table 1 and Table 2, please check Appendix A for details.

For CIFAR10, we also adopt a five-layer P-CapsNet (please see the Appendix) which has about 365,000 parameters. We follow the work of Sabour et al. (2017); Hinton et al. (2018) to crop 24×24 patches from each image during training, and use only the center 24×24 patch during testing. We also use the same data augmentation trick as in Goodfellow et al. (2013) (please see Appendix B for details). As Table 3 shows, P-CapsNet achieves better performance than several routing-based CapsNets by using fewer parameters. The only exception is Capsule-VAE (Ribeiro et al. (2019)) which uses fewer parameters than P-CapsNets but the accuracy is lower. The structure of P-CapsNets#4 can be found in Appendix A.

Models	routing	Error rate(%)	Param #
DCNet++ (Phaye et al. (2018))	Dynamic (-)	0.29	13.4M
DCNet (Phaye et al. (2018))	Dynamic (-)	0.25	11.8M
CapsNets (Sabour et al. (2017))	Dynamic (1)	$0.34_{\pm 0.03}$	6.8M
CapsNets (Sabour et al. (2017))	Dynamic (3)	$0.35_{\pm 0.04}$	6.8M
Atten-Caps (Choi et al. (2019)	Attention (-)	0.64	$\approx$ 5.3M
CapsNets (Hinton et al. (2018))	EM (3)	0.44	320K
P-CapsNets#0	-	$0.32_{\pm 0.03}$	171K
P-CapsNets#3	-	$0.41_{\pm 0.05}$	22.2K

Table 1: Comparison between P-CapsNets and CapsNets with routing procedures in terms of error rate on MNIST. The number in each routing type means the number of routing times.

Algorithm	Error rate(%)	Param #
KFC-Combined (Grosse & Martens (2016))	0.57	52.5K
Adaptive Fastfood 2048 (Yang et al. (2015))	0.73	52.1K
Adaptive Fastfood 1024 (Yang et al. (2015))	0.73	38.8K
KFC-II (Grosse & Martens (2016))	0.76	27.7K
P-CapsNets#3	$0.41_{\pm 0.05}$	22.2K
P-CapsNets#2	$0.85_{\pm 0.08}$	3.8K
ProfSumNet (Wu (2018))	1.55	3.6K
P-CapsNets#1	$1.05_{\pm 0.07}$	2.9K

Table 2: Comparison between P-CapsNets and three compressing algorithms in terms of error rate on MNIST.

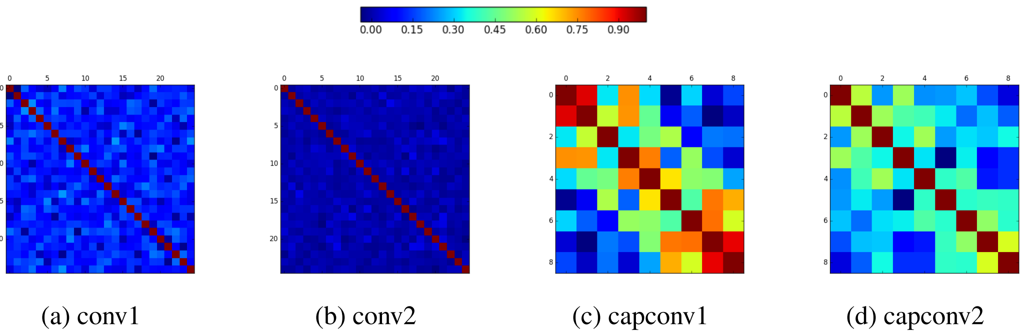
Models	Routing	Ensembled	Error rate(%)	Param #
DCNet++ (Phaye et al. (2018))	Dynamic (-)	1	10.29	13.4M
DCNet (Phaye et al. (2018))	Dynamic (-)	1	18.37	11.8M
MS-Caps (Xiang et al. (2018))	Dynamic (-)	1	24.3	11.2M
CapsNets (Sabour et al. (2017))	Dynamic (3)	7	10.6	6.8M
Atten-Caps (Choi et al. (2019)	Attention (-)	1	11.39	$\approx$5.6M
FRMS (Zhang et al. (2018))	Fast Dynamic (2)	1	15.6	1.2M
FREM (Zhang et al. (2018))	Fast Dynamic (2)	1	14.3	1.2M
CapsNets (Hinton et al. (2018))	EM (3)	1	11.9	458K
P-CapsNets#4	-	1	10.03	365K
Capsule-VAE (Ribeiro et al. (2019))	VB-Routing	1	11.2	323K

Table 3: Comparison between P-CapsNets and several CapsNets with routing procedures in terms of error rate on CIFAR10. The number in each routing type is the number of routing times.

In spite of the parameter-wise efficiency of P-CapsNets, one limitation is that we cannot find an appropriate acceleration solution like cuDNN (Chetlur et al. (2014)) since all current acceleration packages are convolution-based. To accelerate our training, we developed a customized acceleration solution based on cuda (Nickolls et al. (2008)) and CAFFE (Jia et al. (2014)). The primary idea is reducing the communication times between CPUs and GPUs, and maximizing the number of can-be-paralleled operations. Please check Appendix C for details, and we plan to publicly release our code.

5 Visualization of P-CapsNets

We visualize the capsules (filters) of P-CapsNets trained on MNIST (the model used is the same as in Figure 7). The capsules in each layer are 7D tensors. We flatten each layer into a matrix to make



(a) conv1 (b) conv2 (c) capconv1 (d) capconv2

Figure 4: Correlation Matrix of Convolution Layers between CNNs and P-CapsNets. CNN: (a) conv1: 5x5x1x256; (b) conv2: 5x5x256x256. P-CaspNets: (c) capconv1: 1x1x3x3x(1x1x16); (d) capconv2: 1x1x3x3x(1x4x8).

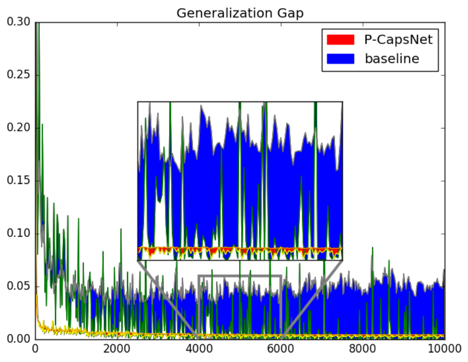


Figure 5: Generalization Gap of P-CapsNets (blue area) and the baseline (red area).

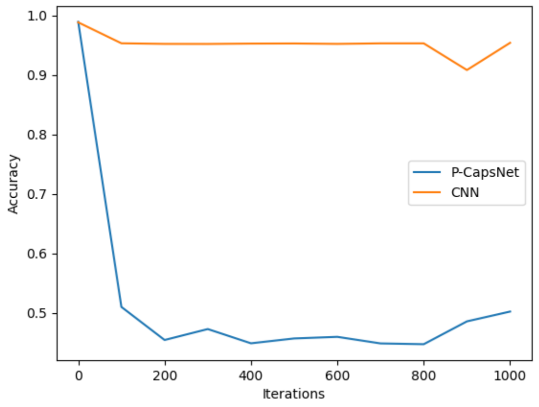


Figure 6: P-CapsNets versus CNNs on white-box adversarial attack.

it easier to visualize. For example, the first capsule layer has a shape of $3 \times 3 \times 1 \times 1 \times (1 \times 1 \times 16)$, so we reshape it to a 9×16 matrix. We do a similar reshaping for the following three layers, and the result is shown in Figure 3.

We observe that the capsules within each layer appear correlated with each other. To check if this is true, we print out the first two layers' correlation matrix for both the P-CapsNet model as well as a CNN model (which comes from Sabour et al. (2017), also trained on MNIST) for comparison. We compute Pearson product-moment correlation coefficients (a division of covariance matrix and multiplication of standard deviation) of filter elements in each of two convolution layers respectively. In our case, we draw two 25x25 correlation matrices from that reshaped conv1 (25x256) and conv2 (25x65536). Similarly, we generate two 9x9 correlation matrices of P-CapsNets from reshaped cconv1 (9x16) and conv2 (9x32). As Figure 4 shows, the filters of convolutional layers have lower correlations within kernels than P-CapsNet. The result makes sense since the capsules in P-CapsNets are supposed to extract the same type of features while the filters in standard CNNs are supposed to extract different ones.

The difference shown here suggests that we might rethink the initialization of CapsNets. Currently, our P-CapsNet, as well as other types of CaspNets all adopt initializing methods designed for CNNs, which might not be ideal.

Epsilon	**Baseline**	**P-CapsNets**
0.05	99.09%	98.66%
0.1	98.01%	94.4%
0.15	95.52%	81.35%
0.2	89.84%	59.52%
0.25	78.31%	39.58%
0.3	54.51%	25.11%

Table 4: Robustness of P-CapsNets. The attacking method is FGSM Goodfellow et al. (2014). In this table, we use The baseline is the same CNN model in Sabour et al. (2017).

6 Generalization Gap

Generalization gap is the difference between a model's performance on training data and that on unseen data from the same distribution. We compare the generalization gap of P-CapsNets with that of the CNN baseline Sabour et al. (2017) by marking out an area between training loss curve and testing loss curve, as Figure 5 shows. For visual comparison, we draw the curve per 20 iterations for baseline Sabour et al. (2017) and 80 iterations for P-CapsNet, respectively. We can see that at the end of the training, the gap of training/testing loss of P-CapsNets is smaller than the CNN model. We conjecture that P-CapsNets have a better generalization ability.

7 Adversarial Robustness

For black-box adversarial attack, Hinton et al. (2018) claims that CapsNets is as vulnerable as CNNs. We find that P-CapsNets also suffer this issue, even more seriously than CNN models. Specifically, we adopt FGSM Goodfellow et al. (2014) as the attacking method and use LeNet as the substitute model to generate one thousand testing adversarial images. As Table 4 shows, when epsilon increases from 0.05 to 0.3, the accuracy of the baseline and the P-CapsNet model fall to 54.51% and 25.11%, respectively.

Hinton et al. (2018) claims that CapsNets show far more resistance to white-box attack; we find an opposite result for P-CapsNets. Specifically, we use UAP (Moosavi-Dezfooli et al. (2016)) as our attacking method, and train a generative network (see Appendix A.2 for details) to generate universal perturbations to attack the CNN model (Sabour et al. (2017)) as well as the P-CapsNet model shown in Figure 7). The universal perturbations are supposed to fool a model that predicts a targeted wrong label ((the ground truth label + 1) % 10). As Figure 6 shows, when attacked, the accuracy of the P-CapsNet model decreases more sharply than the baseline.

It thus appears that P-CapsNets are more vulnerable to both white-box and black-box adversarial attacking compared to CNNs. One possible reason is that the P-CapsNets model we use here is significantly smaller than the CNN baseline (3688 versus 35.4M). It would be a fairer comparison if two models have a similar number of parameters.

8 Conclusion

We propose P-CapsNets by making three modifications based on CapsNets Sabour et al. (2017), 1) We replace all the convolutional layers with capsule layers, 2) We remove routing procedures from the whole network, and 3) We package capsules into rank-3 tensors to further improve the efficiency. The experiment shows that P-CapsNets can achieve better performance than multiple other CapsNets variants with different routing procedures, as well as than deep compressing models, by using fewer parameters. We visualize the capsules in P-CapsNets and point out that the initializing methods of CNNs might not be appropriate for CapsNets. We conclude that the capsule layers in P-CapsNets can be considered as a general version of 3D convolutional layers. We conjecture that CapsNets can encode the intrinsic spatial relationship between a part and a while efficiently, comes from the tensor-to-tensor mapping between adjacent capsule layers. This mapping is presumably also the reason for P-CapsNets' good performance.

REFERENCES

Zhenhua Chen, Chuhua Wang, Tiancong Zhao, and David Crandall. Generalized capsule networks with trainable routing procedure. *ICML Worksop: Understanding and Improving Generalization in Deep Learning*, 2019.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL `http://arxiv.org/abs/1410.0759`.

Jaewoong Choi, Hyun Seo, Suii Im, and Myungju Kang. Attention routing between capsules. *CoRR*, abs/1907.01750, 2019. URL `http://arxiv.org/abs/1907.01750`.

Xi Edgar, Bing Selina, and Jin Yang. Capsule network performance on complex data. *CoRR*, 2017. URL `https://arxiv.org/abs/1712.03480`.

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, December 2014.

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout Networks. *arXiv e-prints*, art. arXiv:1302.4389, Feb 2013.

Roger Baker Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. *In International Conference on Machine Learning*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL `http://arxiv.org/abs/1502.01852`.

Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=HJWLfGWRb`.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Yann LeCun, Fu Jie Huang, and Léon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2004.

Hongyang Li, Xiaoyang Guo, Bo Dai, Wanli Ouyang, and Xiaogang Wang. Neural network encapsulation. *ECCV*, 2018.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *CoRR*, abs/1610.08401, 2016. URL `http://arxiv.org/abs/1610.08401`.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL `http://doi.acm.org/10.1145/1365490.1365500`.

Inyoung Paik, Taeyeong Kwak, and Injung Kim. Capsule Networks Need an Improved Routing Algorithm. *arXiv e-prints*, art. arXiv:1907.13327, Jul 2019.

Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, and Deepti R. Bathula. Dense and diverse capsule networks: Making the capsules learn better. *CoRR*, abs/1805.04001, 2018. URL `http://arxiv.org/abs/1805.04001`.

Fabio De Sousa Ribeiro, Georgios Leontidis, and Stefanos D. Kollias. Capsule routing via variational bayes. *CoRR*, abs/1905.11455, 2019. URL `http://arxiv.org/abs/1905.11455`.

Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017. URL `http://arxiv.org/abs/1710.09829`.

Dilin Wang and Qiang Liu. An optimization view on dynamic routing between capsules, 2018. URL https://openreview.net/forum?id=HJjtFYJDf.

Chai Wah Wu. Prodsumnet: reducing model parameters in deep neural networks via product-of-sums matrix decompositions. *CoRR*, abs/1809.02209, 2018. URL http://arxiv.org/abs/1809.02209.

Canqun Xiang, Lu Zhang, Yi Tang, Wenbin Zou, and Chen Xu. Ms-capsnet: A novel multi-scale capsule network. *IEEE Signal Processing Letters*, 25:1850–1854, 2018.

Z. Yang, M. Moczulski, M. Denil, N. d. Freitas, A. Smola, L. Song, and Z. Wang. Deep fried convnets. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1476–1483, Dec 2015. doi: 10.1109/ICCV.2015.173.

Suofei Zhang, Wei Zhao, Xiaofu Wu, and Quan Zhou. Fast dynamic routing based on weighted kernel density estimation. *CoRR*, abs/1805.10807, 2018. URL http://arxiv.org/abs/1805.10807.

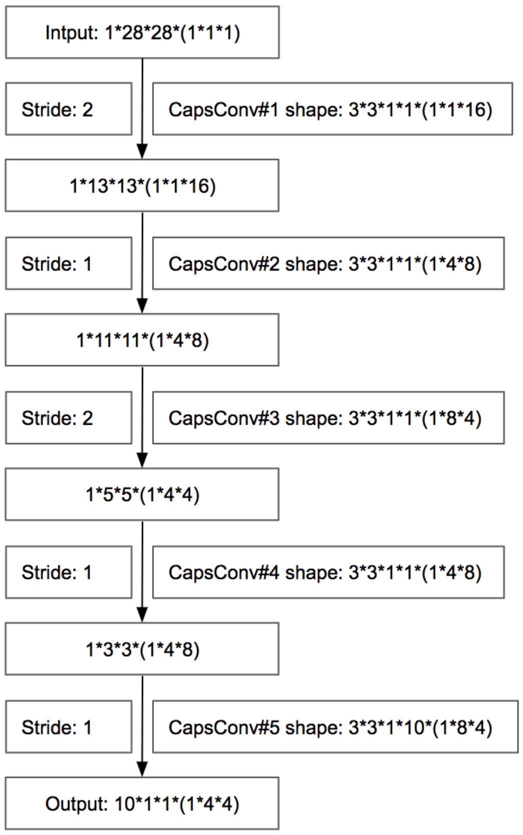

Intput: 1*28*28*(1*1*1)
Stride: 2
CapsConv#1 shape: 3*3*1*1*(1*1*16)
1*13*13*(1*1*16)
Stride: 1
CapsConv#2 shape: 3*3*1*1*(1*4*8)
1*11*11*(1*4*8)
Stride: 2
CapsConv#3 shape: 3*3*1*1*(1*8*4)
1*5*5*(1*4*4)
Stride: 1
CapsConv#4 shape: 3*3*1*1*(1*4*8)
1*3*3*(1*4*8)
Stride: 1
CapsConv#5 shape: 3*3*1*10*(1*8*4)
Output: 10*1*1*(1*4*4)

Figure 7: The structure P-CapsNets#2.

A Network Structures

A.1 MNIST&CIFAR

For MNIST&CIFAR10, we designed five versions of CapsNets (CapsNets#0, CapsNets#1, CapsNets#2, CapsNets#3), they are all five-layer CapsNets. Take CapsNets#2 as an example, the input are gray-scale images with a shape of 28 $\times$ 28, we reshape it as a 6D tensor, $1 \times 28 \times 28 \times (1 \times 1 \times 1)$ to fit our P-CaspNets. The first capsule layer (CapsConv#1, as Figure 7 shows.), is a 7D tensor, $3 \times 3 \times 1 \times 1 \times (1 \times 1 \times 16)$. Each dimension of the 7D tensor represents the kernel height, the kernel width, the number of input capsule feature map, the number of output capsule feature map, the capsule's first dimension, the capsule's second dimension, the capsule's third dimension. All the following feature maps and filters can be interpreted in a similar way.

Similarly, the five capsule layers of P-CapsNets#0 are $3 \times 3 \times 1 \times 1 \times (1 \times 1 \times 32, 3 \times 3 \times 1 \times 2 \times (1 \times 8 \times 8), 3 \times 3 \times 2 \times 4 \times (1 \times 8 \times 8), 3 \times 3 \times 4 \times 2 \times (1 \times 8 \times 8, 3 \times 3 \times 2 \times 10 \times (1 \times 8 \times 8)$ respectively. The strides for each layers are $(2, 1, 2, 1, 1)$.

The five capsule layers of P-CapsNets#1 are $3 \times 3 \times 1 \times 1 \times (1 \times 1 \times 16, 3 \times 3 \times 1 \times 1 \times (1 \times 4 \times 6)$, $3 \times 3 \times 1 \times 1 \times (1 \times 6 \times 4), 3 \times 3 \times 1 \times 1 \times (1 \times 4 \times 6, 3 \times 3 \times 1 \times 10 \times (1 \times 6 \times 4)$ respectively. The strides for each layers are $(2, 1, 2, 1, 1)$.

The five capsule layers of P-CapsNets#3 are $3 \times 3 \times 1 \times 1 \times (1 \times 1 \times 32, 3 \times 3 \times 1 \times 4 \times (1 \times 8 \times 16)$, $3 \times 3 \times 4 \times 8 \times (1 \times 16 \times 8), 3 \times 3 \times 8 \times 4 \times (1 \times 8 \times 16, 3 \times 3 \times 4 \times 10 \times (1 \times 16 \times 16)$ respectively. The strides for each layers are $(2, 1, 2, 1, 1)$.

The five capsule layers of P-CapsNets#4 are $3 \times 3 \times 1 \times 1 \times (1 \times 3 \times 32, 3 \times 3 \times 1 \times 4 \times (1 \times 8 \times 16)$, $3 \times 3 \times 4 \times 8 \times (1 \times 16 \times 8), 3 \times 3 \times 8 \times 10 \times (1 \times 8 \times 16, 3 \times 3 \times 10 \times 10 \times (1 \times 16 \times 16)$ respectively. The strides for each layers are $(2, 1, 1, 2, 1)$.

A.2 The Generative Network for Adversarial Attack

The input of the generative network is a 100-dimension vector filled with a random number ranging from -1 to 1. Then the vector is fed to a fully-connected layer with 3456 output (the output is reshaped as $3 \times 3 \times 384$). On top of the fully-connected layer, there are three deconvolutional layers. They are one deconvolutional layer with 192 output (the kernel size is 5, the stride is 1, no padding), one deconvolutional layer with 96 output (the kernel size is 4, the stride is 2, the padding size is 1), and one deconvolutional layer with 1 output (the kernel size is 4, the stride is 2, the padding size is 1) respectively. The final output of the three deconvolutional layers has the same shape as the input image (28×28) which are the perturbations.

B Meta-parameters & Data Augmentation

For all the P-CapsNet models in the paper, We add a Leaky ReLU function(the negative slope is 0.1) and a squash function after each capsule layer. All the parameters are initialized by MSRA (He et al. (2015)).

For MNIST, we decrease the learning rate from 0.002 every 4000 steps by a factor of 0.5 during training. The batch size is 128, and we trained our model for 30 thousand iterations. The upper/lower bound of the margin loss is 0.5/0.1. The λ is 0.5. We adopt the same data augmentation as in (Sabour et al. (2017)), namely, shifting each image by up to 2 pixels in each direction with zero padding.

For CIFAR10, we use a batch size of 256. The learning rate is 0.001, and we decrease it by a factor of 0.5 every 10 thousand iterations. We train our model for 50 thousand iterations. The upper/lower bound of the margin loss is 0.6/0.1. The λ is 0.5. Before training we first process each image by using Global Contrast Normalization (GCN), as Equation 8 shows.

$$\mathsf{X}'_{i,j,k} = s\frac{\mathsf{X}_{i,j,k} - \overline{\mathsf{X}}}{max\left\{\epsilon, \sqrt{\alpha + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(\mathsf{X}_{i,j,k} - \overline{\mathsf{X}})^2}\right\}} \tag{8}$$

where, X and $X^{'}$ are the raw image and the normalized image. s, ϵ and α are meta-parameters whose values are 1, $1e^{-9}$, and 10. Then we apply Zero Component Analysis (ZCA) to the whole dataset. Specifically, we choose 10000 images $\mathsf{X}^{''}$ randomly from the GCN-processed training set and calculate the mean image $\overline{\mathsf{X}''}$ across all the pixels. Then we calculate the covariance matrix as well as the singular values and vectors, as, Equation 9 shows.

$$\mathsf{U}, \mathsf{S}, \mathsf{V} = SVD(\mathrm{Cov}(\mathsf{X}^{''} - \overline{\mathsf{X}''})) \tag{9}$$

Finally, we can use Equation 10 to process each image in the dataset.

$$\mathsf{X}_{\mathsf{ZCL}} = \mathsf{U}\ \mathrm{diag}(\frac{1}{\sqrt{\mathrm{diag}(\mathsf{S}) + 0.1}})\mathsf{U}^{\mathsf{T}} \tag{10}$$

Batch Size	**CPU**(s/100 iterations)	**CUDA Kernel**(s/100 iterations)
50	106.22	19.67
100	213.60	46.57
150	319.37	61.63
200	425.15	91.59

Table 5: Comparison on time consumed between CPU mode and acceleration solution

C Acceleration Solution for P-CapsNets

Different from convolution operations in CNNs, which can be interpreted as a few large matrix multiplications during training, the capsule convolutions in P-CaspNets have to be interpreted as a large number of small matrix multiplication. If we use the current acceleration library like CuDNN (Chetlur et al. (2014)) or the customized convolution solution in CAFFE (Jia et al. (2014)), too many communication times would be incorporated which slows the whole training process a lot. The communication overhead is so much that the training is slower than CPU-only mode. To overcome this issue, we parallel the operations within each kernel to minimize communication times. We build two P-CaspNets#3 models, one is CPU-only based, the other one is based on our parallel solution. The GPU is one TITAN Xp card, the CPU is Intel Xeon. As Table 5 shows, our solution achieves at least $4\times$ faster speed than the CPU mode for different batch sizes.