Pushing the Limits of Capsule Networks

Prem Nair, Rohan Doshi, Stefan Keselj
Princeton University
{pnair, rkdoshi, skeselj}@princeton.edu

1 Motivation and Goal

Convolutional neural networks use pooling and other downscaling operations to maintain translational invariance for detection of features, but in their architecture they do not explicitly maintain a representation of the locations of the features relative to each other. This means they do not represent two instances of the same object in different orientations the same way, like humans do, and so training them often requires extensive data augmentation and exceedingly deep networks.

A team at Google Brain recently made news with an attempt to fix this problem: Capsule Networks (Sabour et al. [2017]), hereon referred to as CapsNets. While a normal CNN works with scalar outputs representing feature presence, a CapsNet works with vector outputs representing entity presence. There is much discussion about whether these new models could actually work as intended because Sabour et al. [2017] mainly applied them to the MNIST dataset under favorable hyperparameter conditions.

We want to stress test CapsNet in various incremental ways to better understand their performance and expressiveness. In broad terms, the goals of our investigation are to:

1. Test CapsNets on datasets that are like MNIST but harder in a specific way.
2. Explore the internal embedding space and sources of error for CapsNets.

2 Background and Related Work

The original idea driving this work is the capsule, first discussed in Hinton et al. [2011]. The capsule was pitched as a convenient way to represent an entity: it is a vector whose norm indicates the probability that entity is present and whose direction indicates the configuration that entity is in. Capsules could theoretically be combined to form hierarchical tree structures representing entities. It was a nice idea, but it did not get much traction until a few months ago when Sabour, Frosst, and Hinton finally figured out how to train a network to recognize and work in this space (Sabour et al. [2017]). In this section we will attempt to convey our understanding of their model, but the best resource is their actual paper.

2.1 Fundamentals of Capsules and the Dynamic Routing Algorithm

CapsNets are comprised of layers of capsules, each of which are composed of neurons. But unlike neurons, capsules deal with inputs and an output that are vectors. Each vector is meant to encode a rich representation of some entity: the vector's direction indicates what form the entity seems to be in (e.g. the pose of an object) and the vector's norm indicates the confidence of the representation. These vectors can be combined by affinely transforming and then adding them (analogous to multiplying scalars by weights and then adding them). The method for the forward pass of information from capsule layer to capsule layer is called dynamic routing.

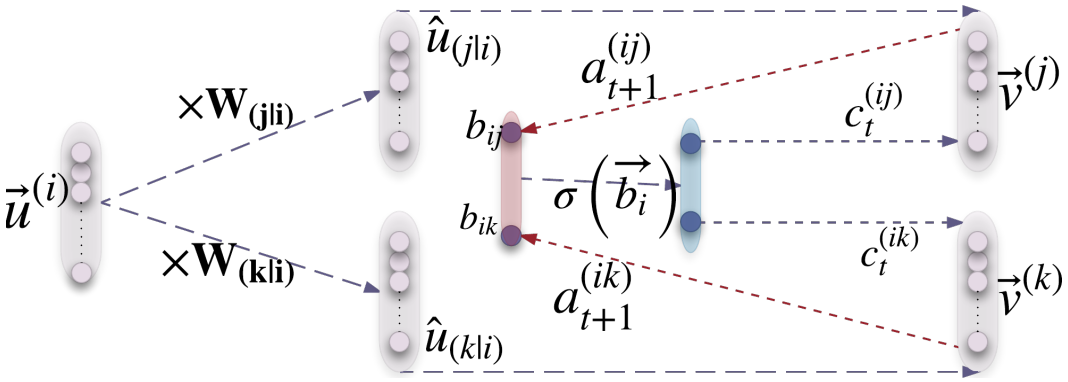


Figure 1: Capsule Routing (Sabour et al. [2017])

A given capsule j processes total input vector $\mathbf{s}_j$ and outputs an activation vector $\mathbf{v}_j$, whose direction is preserved, but whose magnitude is "squashed" between 0 and 1 by the non-linearity in Equation 1. The length of $\mathbf{v}_j$ indicates the probability of existence for the entity represented by the capsule.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||} \tag{1}$$

The vector $\mathbf{s}_j$ is generated by processing the vectors from each of the connected capsules i from the previous layers. For a given capsule activation from the previous layer u_i, we apply weight matrix W_{ij} to yield $\hat{\mathbf{u}}_{\mathbf{j}|\mathbf{i}}$, a "prediction" vector to approximate $\mathbf{v}_j$. The $\hat{\mathbf{u}}_{\mathbf{j}|\mathbf{i}}$ from each capsule i is weighted by a coupling coefficient c_{ij}, which is calculated via the dynamic routing algorithm.

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i \ , \qquad \mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i} \tag{2}$$

The sum of the coupling coefficients between capsule i and all possible capsules j in the next layer equals 1, forcing capsules to weigh the importance of more up-stream capsules. This is enforced by calculating c_{ij} through a "routing softmax", whose inputs b_{ij} are the log prior probabilities that capsule i and j are coupled.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})} \tag{3}$$

Now, to update c_{ij} via dynamic routing, we rely on a simple heuristic: we want to reward coupling coefficients that maximize the agreement a_{ij} between the prediction and activation vector. In other words, we want to maximize $a_{ij} = u_{i|j} \cdot \mathbf{v}_j$ In order to update the coupling coefficients, b_{ij} is iteratively incremented by a_{ij} through iterations of the dynamic routing algorithm (Procedure 1) between training iterations for the model, and also visualized in Figure 1.

Procedure 1 Routing algorithm (from Sabour et al. [2017])

procedure ROUTING($\hat{\boldsymbol{u}}_{j|i}$, r, l)
  for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
  for r iterations **do**
    for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax` computes Eq. 3
    for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
    for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash` computes Eq. 1
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
  return $\mathbf{v}_j$

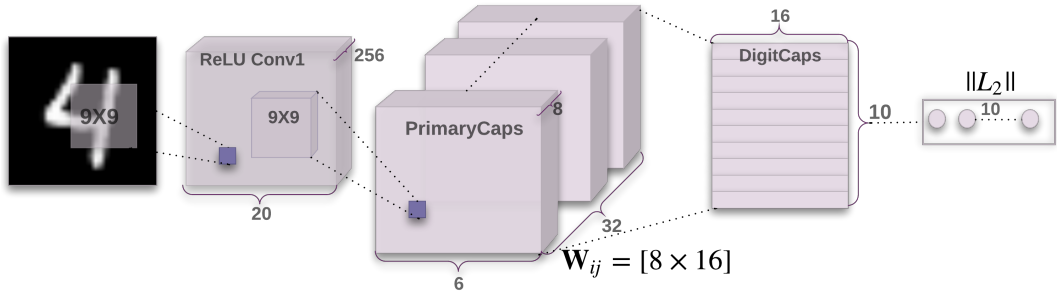


Figure 2: CapsNet Architecture (Sabour et al. [2017])

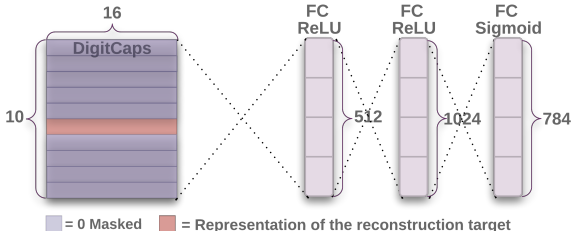


Figure 3: Reconstruction Architecture (Sabour et al. [2017])

2.2 CapsNet Architecture

We work with a simple CapsNet architecture proposed in the original paper. It contains two halves. The first half (Figure 2) outputs a class prediction after three layers of processing (two convolution layers and one fully connected layer). The first layer is a convolutional layer that maps the 28×28 image to a $6 \times 6 \times 256$ volume using a 9×9 kernel, 256 feature maps, a stride of 1, and ReLU non-linearity. Next, the second layer (PrimaryCaps), is a convolution capsule layer that produces a $6 \times 6 \times 256$ volume using a 9×9 kernel and stride of 2. The volume is now sliced along its depth into 32 different layers of 8-dimension capsules, for a total of $6 \times 6 \times 32$ capsules. The third layer (DigitCaps) is a fully connected layer of 10 different 16-dimension capsules, with each capsule corresponding to an output class (a.k.a. 10 digit). Note that we have dynamic routing only between the PrimaryCaps and DigitCaps layers. Finally, we take the magnitude of these 16-dimension embeddings and output a final predicted corresponding to the embedding with the largest magnitude.

The purpose of the second half of the model (Figure 3), also referred to as the decoder, is to implement reconstruction as a regularization method. The 16-dimension embeddings from each class are concatenated, with all but the winning class's vector components masked to 0. Ten digit classes would mean a final embedding of length 160, which is then fed through 3 fully connected layers with 512, 1024, and 764 neurons respectively. The final 784-dimension output is reshaped into a 28×28 reconstruction of the ground truth.

2.3 Loss Function

The two halves of the model are combined to train the network with a two part loss function. The first term penalizes false predictions, with predictions corresponding to the DigitCaps embeddings with the largest magnitude. The second term penalizes reconstruction error, or the difference in the ground truth image and its reconstruction after passing its DigitCaps embedding through the fully connected layers. We can use backpropagation to minimize the loss since dynamic routing is differentiable when unrolled.

2.4 Our Contributions

Based on the aforementioned sections, one cannot help but question the design decisions made in making CapsNets work on MNIST. This is the root of our previously mentioned project goals, and now that we have properly explained CapsNets, we can talk about them in more detail.

First, we will explore whether this architecture will perform well on more difficult datasets. Unlike MNIST, other datasets may include images with noise, color, affine transformations, intra-class variation, natural scenes, and a variety of other factors that may or may not work well with the capsule architecture. We will also explore the effect of the number of routing iterations on performance, balanced against complexity constraints.

Second, to better understand the source of error and the expressiveness of the underlying model, we will try to understand and visualize what sorts of representations of the underlying data the model is able to capture. Our work adds to the existing literature by extending CapsNet to new datasets and providing unique visualizations that bring to light novel insights on CapsNets.

3 Approach and Implementation

3.1 Datasets

It has been demonstrated that CapsNets can achieve state of the art on MNIST. We want to try them on datasets that are marginally harder in specific ways, so that even if they fail we can gain insight into the nature of the limitations of CapsNets.

1. **MNIST (LeCun and Cortes [2010], Figure 4a)** The standard set of normalized and centered 28×28 black and white images of handwritten digits (0-9). 60,000 training samples and 10,000 testing samples.
2. **Fashion-MNIST (Xiao et al. [2017], Figure 4b)** Exactly like MNIST except the image classes are items of clothing (from t-shirts to ankle boots).
3. **SVHN (Netzer et al. [2011], Figure 4c)** Contains cropped RBG 32×32 pixel images of house number digits taken from Google Street View. Like MNIST in that it is digits, but more complex because it has varying colors and styles, and multiple digits could be in a single sample. Is also larger, with 73257 training samples and 26032 testing samples.
4. **CIFAR-10 (Krizhevsky and Hinton [2009], Figure 4d)** Real-world objects database of RBG 32×32 pixel images across 10 classes, including vehicles (airplane, automobile, ship, and truck) and animals (bird, cat, deer, dog, frog, and horse). 50,000 training samples and 10,000 testing samples.

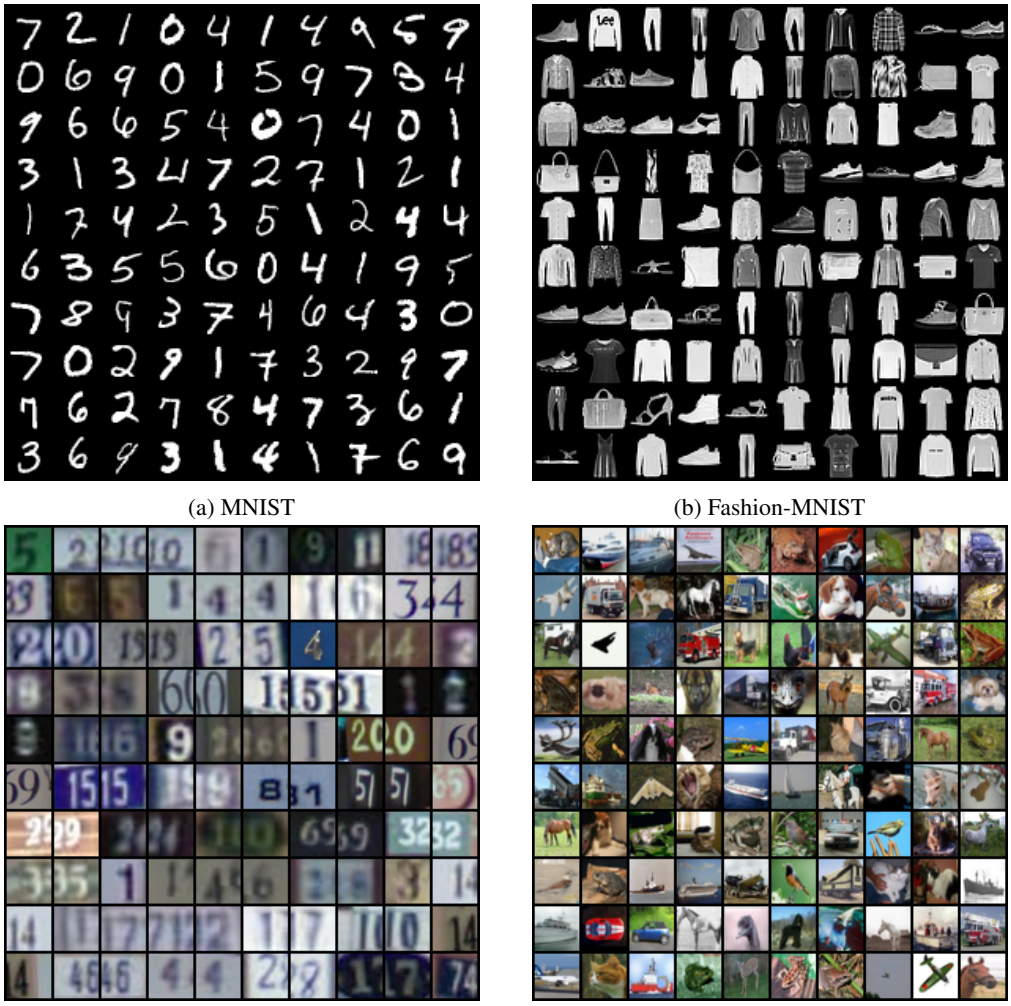
Testing on MNIST will provide us an opportunity to verify the results of Sabour et al. [2017]. Testing on Fashion-MNIST and CIFAR10 will allow us to see whether CapsNets generalize to different types of data. Testing SVHN and CIFAR10 again will allow us to see whether CapsNets generalize to color and to more complexity and intraclass variation.

3.2 Deformations

On top of these four datasets, we want to introduce some synthetic deformations to see how robust CapsNets are to new data. Specifically, we want to see how well they can classify data which is unlike that which they have seen before, but which still belongs to the same broader distribution. In other words, we want to see the extent to which CapsNets are deformation invariant for classification.

For each of the four datasets described above (MNIST, Fashion-MNIST, SVHN, CIFAR10) we generate an alternate, deformed dataset by applying a random affine deformation consisting of:

1. **Rotation**: Rotated image by a uniformly sampled angle within $[-20°, 20°]$.
2. **Shear**: Sheared along x and y axes by uniformly sampling shear parameters within $[-0.2, 0.2]$. (Shear parameters are numbers added to the cross-terms in the 2×3 matrix describing an affine transformation.)
3. **Translation**: Translated along x and y axis by a uniformly sampled displacement parameters within $[-1, +1]$. (Displacement parameters are numbers added to the constant terms in the 2×3 matrix.)
4. **Scale**: Always scaled image 150%.



(a) MNIST (b) Fashion-MNIST

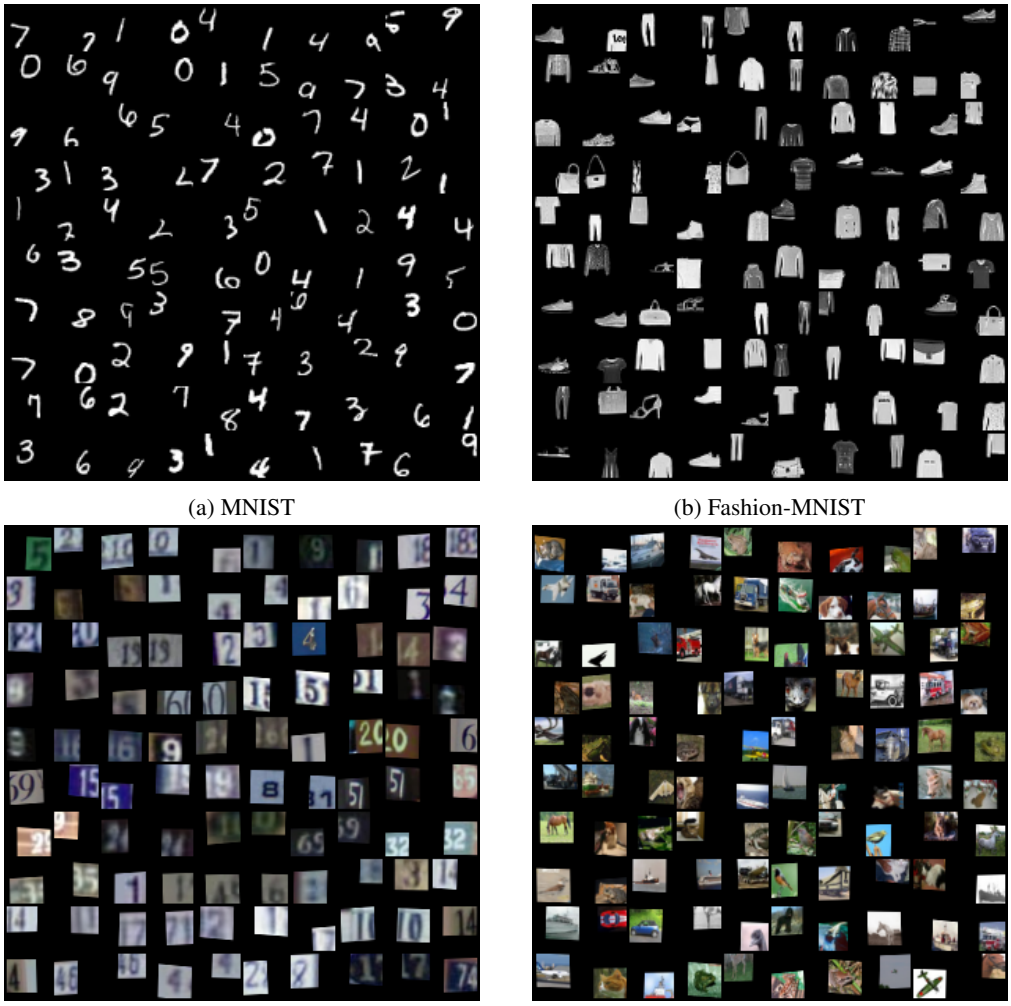
(c) SVHN (d) CIFAR10

Figure 4: Dataset visualizations.

In our experiments on robustness to transformation, we trained the CapsNet on the original versions with the datasets and tested on their affine-deformed versions. 100 examples of affinely transformed testing data are shown in Figure 5.

3.3 Baseline

We need to baseline the performance of our ConvNet against a neural network architecture with a similar complexity (not only in number of parameters, but also, more importantly, in runtime). We chose to baseline against AlexNet, specifically a modification of the implementation put forth in the "One Weird Trick..." paper which proposes the parallelization of the training of CNNs across GPUs (Krizhevsky [2014]). We retain the 5 traditional AlexNet layers of repeated convolution, ReLU non-linearities, and max-pooling, but we replace the final three fully connected hidden layers with a single fully connected layer of only 256 neurons. This produces performance that reaches close to state-of-the-art.



(a) MNIST (b) Fashion-MNIST

(c) SVHN (d) CIFAR10

Figure 5: Affine dataset visualizations.

3.4 Implementation

3.4.1 Code

We started by forking a GitHub repository (Iwasaki [2018]) of a PyTorch CapsNet implementation with an architecture matching the one used in the dynamic routing paper. This repository was the most "starred" implementation on Github and had clean, albeit modest code. We made significant changes to enable all the different ways we wanted to stretch CapsNets. Our code can be found at our open-sourced repository, Keselj et al. [2018].

1. **Datasets** We enabled the use of additional datasets beyond MNIST, specifically Fashion-MNIST, SVHN, and CIFAR10, by adding new data loading and processing methods.
2. **Reconstructions** At the end of every epoch, we extract the DigitCapsule embeddings, reconstruct every class by running it through the decoder, and store all the reconstructions.
3. **Perturbations** Also at the end of every epoch, we perturb each element of the winning capsule embedding and store the reconstructions which result from them.
4. **Affine Transformations** We made a class which takes in a normal dataset and outputs the randomly affine transformed version of that dataset.

5. **AlexNet** We added basic support for training an AlexNet model under the same conditions as our CapsNets.

6. **Visualizations** The original code had basic error and accuracy logging on Visdom, but we moved it to TensorBoard and added displays of all our reconstructions, perturbations. The move to TensorBoard was primarily motivated by its useful dimensionality reduction suite, which allows us to visualize the internal embeddings of the model.

3.4.2 Computing

Apart from these infrastructural changes, we had to run many trials under different parameters dozens of trials to train all of the CapsNets models using various hyperparameters and datasets. To quicken and parallelize the training process, we implemented our models with CUDA so that we could leverage GPUs to decrease training time. All three of the authors had access to GPUs associated with the labs in which they are conducting their theses. Prem and Rohan used 8 GPUs as part of the Visual AI Lab and Stefan has access to 6 GPUs associated with the Seung lab (all GPUs were GEFORCE GTX 1080s). We ran at least 30 CapsNet runs for our results in this paper, each taking about 4 hours, so a lower bound of our GPU-hours used is 120.

4 Results and Discussion

4.1 Performance

4.1.1 Different Datasets

We were able to apply CapsNets and AlexNet to our four dataset to achieve reasonable, but far from state of the art results. The training and testing curves for CapsNets and AlexNet are shown in below in Figure 6 and the end accuracies after 50 epochs are shown in the first half of Table 1.

We already know from the original paper that CapsNets can achieve almost state of the art MNIST, and here we verified that result by getting 99.5% accuracy. CapsNets did relatively well on Fashion-MNIST and SVHN, with 89.8% and 91.06% accuracies, respectively. This makes sense because they are very similar to MNIST; Fashion-MNIST is in the same style and SVHN is of similar content. It is safe to say that CapsNets performed poorly on CIFAR10, achieving only 68.53% accuracy before train and test error began diverging. This was expected, since the intra-class variation and background noise is more complex than that of MNIST .

CapsNets outperformed AlexNet in every case. For MNIST and Fashion-MNIST, the difference was marginal (1.03% and 2.24%, respectively), but in SVHN and CIFAR10 it was more substantial (9.01% and 7.92%, respectively). When making this comparison, it is important to note that neither model is state-of-the-art, so it is not really an apples-to-apples comparison of architectures. Furthermore, we must emphasize that both models' hyperparameters are not optimal; for CapsNets we used the hyperparameters used in Sabour et al. [2017] (which was optimized for MNIST), and for AlexNet, we used PyTorch's default parameters (however, we decreased the learning rate for some datasets if the model could not converge). Nonetheless, this first-pass sanity check confirms that there is something interesting about CapsNets, since they can outperform CNNs that have many more parameters.

4.1.2 Affine Deformations

Table 1: CapsNet v. AlexNet: Classification test accuracy across 4 datasets after 50 epochs. With and without affine deformations.

Dataset	Normal		Affine	
Network	CapsNet	AlexNet	CapsNet	AlexNet
MNIST	**99.50**	98.47	42.75	**55.21**
Fashion-MNIST	**89.80**	83.00	**30.01**	25.80
CIFAR10	**68.53**	49.97	**22.89**	20.21
SVHN	**91.06**	87.43	**24.24**	22.86

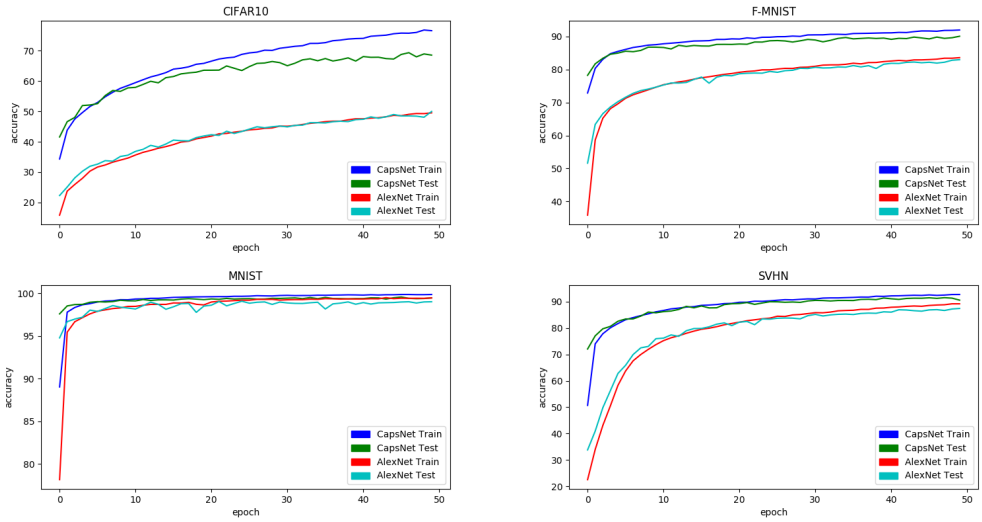


Figure 6: CapsNet vs. AlexNet: Train and Test Curves for 50 epochs on 4 datasets (MNIST, Fashion-MNIST, SVHN, and CIFAR-10) without deformation.

A motivating reason for the study of CapsNets is the claim that they can recognize entities regardless of their configuration, unlike CNNs. As a test for this hypothesis, we trained CapsNet and AlexNet on the training sets from the four datasets above, but then tested on an affinely transformed version of their test sets, as described in our approach.

The final accuracies achieved by each model are shown in Table 1. Surprisingly, AlexNet outperforms CapsNet on deformed test data for MNIST, even though it had a lower accuracy on the normal test data. Still, CapsNets performed better on the affine-deformed versions of Fashion-MNIST, CIFAR-10, and SVHN datasets with more complex samples and greater intraclass variation. One explanation is that AlexNet has enough parameters to memorize the ten simple classes of MNIST, resulting in over-fitting. This is why its performance was not able to generalize to more complex classes. In that case, CapsNets may be learning a more generalizable representation of the data.

However, it could be the case that our AlexNet is simply a poor representative of CNNs, so it performing poorly on deformed data does not mean all CNNs would. To get a better idea about how capsule networks verus CNNs perform on this task in general, it might be interesting to consult a reasonable proxy for how poorly a model generalizes to deformed test data: the drop in accuracy when testing on the deformed data versus the normal data. In every dataset, CapsNet had a larger drop than AlexNet (56.75 % vs 43.26% for MNIST, 59.79 % vs 57.20% on Fashion-MNIST, 66.82% vs 64.57 on SVHN, and 45.64% vs 29.76% on CIFAR10). This is raises doubts about whether CapsNets are actually better at capturing spacial relationships, because a model which builds spatially dependent representations should be better at generalizing to deformed data. It is perhaps the case that our AlexNet "cheated" by starting out with a poor performance and then not dropping that much. This is at least in part true (AlexNet had 49.97% accuracy on normal CIFAR10), but it is not true in every case (AlexNet had 98.47% accuracy on normal MNIST).

4.1.3 Number of Routing Iterations

It is evident that CapsNets need to improve their performance on datasets other than vanilla MNIST if they are to be as useful as CNNs. As mentioned above, we did not spend much time optimizing hyperparameters in our initial CapsNet vs. AlexNet tests, but now is a good time to dive into that. We found that varying the standard hyperparameters describing SGD (batch size, learning rate, learning rate decay, and momentum) did not change final performance much. This is because our CapsNets were already converging well so these parameters could only decrease the training time.

The hyperparameter which most interests us is the number of routing iterations (from the dynamic routing algorithm). This parameter is unique to CapsNets and has important implications on their performance and runtime: every time inference is run, this many routing operations are run to

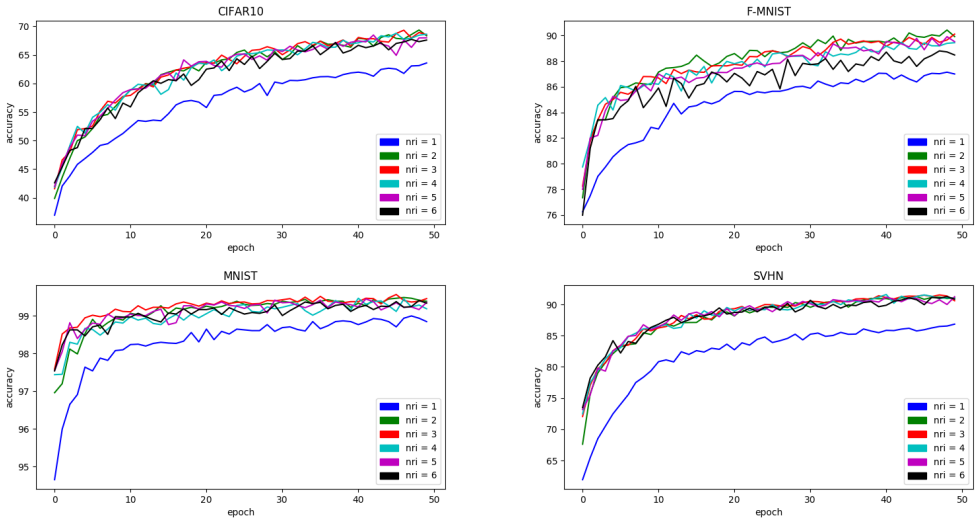


Figure 7

determine which DigitCapsules the PrimaryCapsules send their information to. Sabour et al. [2017] recommends 3 routing iterations, but we are curious about what the best number would be with respect to all four of our datasets.

We ran a set of experiments where we set the number of routing operations to one of (1, 2, 3, 4, 5) for each dataset. Although 3 iterations yielded good test accuracies, overall 2 performed about as well, even performing better in some instances. This is interesting for two reasons. First, it seems to go against the original paper's claim. Perhaps it could be the case that 3 iterations converges more stably than 2, but we reran our experiments multiple times and got similar graphs each time. Second, it could be indicative that the proposed dynamic routing algorithm is too "abrupt". The process of deciding which PrimaryCapsules feed into which DigitCapsules is a complex one, and would be surprising if an algorithm would only ever need two iterations to do this well.

4.2 Embedding Spaces

4.2.1 Reconstructions

Now that we have an idea of the performance of CapsNet on various tasks, we want to understand why they are behaving the way they are so we can improve them. We can get a direct look into one component of the network's loss function by visualizing the reconstructions it generates. In Figures 8, 9, 10, and 11 we have shown the network's reconstruction after epoch 2, its reconstruction after epoch 50, and the ground truth images for each dataset.

MNIST's reconstruction improved substantially from epoch 2 to 50, mostly matching ground truth. However, the classification accuracy began to plateau earlier than the reconstruction accuracy, which suggests that the reconstruction does not necessarily help much with classification. Fashion-MNIST also had good reconstruction improvement, but the finer clothing details were not captured, which could be an effect of an embedding that is too small, or a decoder that is too shallow. SVHN went from failing to reconstruct anything substantial to a mostly gray rendering of the numbers. The network still failed to replicate colors. The reconstructions of CIFAR10, were always unrecognizable, apart from horses, which we assume is due to the consistency of point of view for most horse photos.

4.2.2 Understanding Errors

For the datasets with the best reconstructions, MNIST and Fashion-MNIST, we can make use of their reconstructions when understanding error sources, since we can decode the embeddings and recover visualizations of the misclassification the network made, as seen in Figure 12. Normally, the decoder accepts the 160-length embedding vector masked to 0 in all but the section that corresponded to the

Figure 8: Epoch 2 reconstruction, epoch 50 reconstruction, and ground truth for MNIST.

Figure 9: Epoch 2 reconstruction, epoch 50 reconstruction, and ground truth for Fashion-MNIST.

Figure 10: Epoch 2 reconstruction, epoch 50 reconstruction, and ground truth for SVHN.

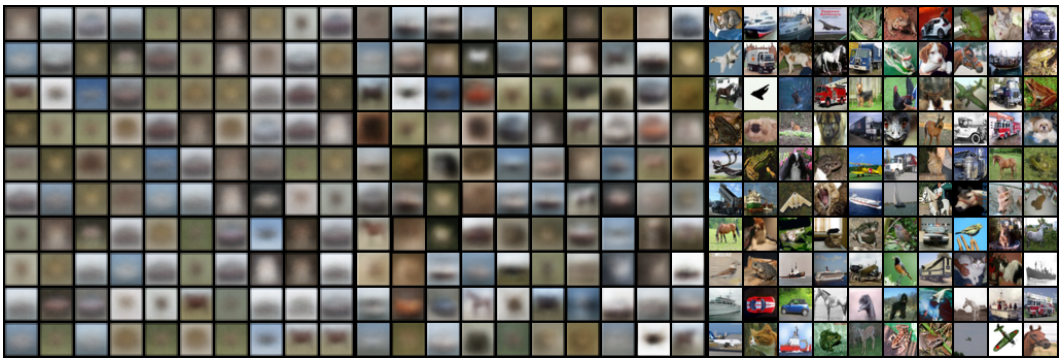
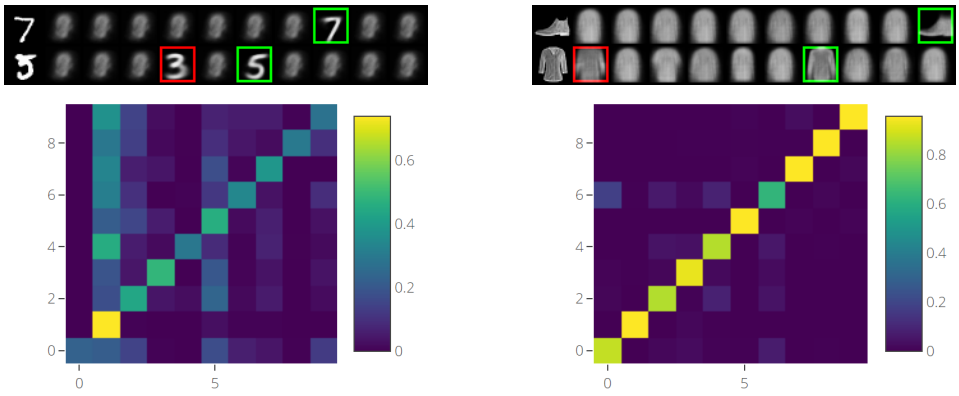


Figure 11: Epoch 2 reconstruction, epoch 50 reconstruction, and ground truth for CIFAR10.

most confident class, and uses it to reconstruct the image. By simply unmasking each class's section in turn, we can generate visualizations for each class.

When the network is very confident and gets something right, as seen in the top row of reconstructions in Figure 12, the other class's components are small enough such that even without masking we don't see any meaningful reconstructions. However, as seen in the bottom row of reconstructions, when you reconstruct the correct class of most errors, a meaningful result is observed. The errors are genuinely ambiguous from a human perspective as well. In the MNIST case, the digit is somewhere between 5 and 3, and in the Fashion-MNIST case, the erroneous class is "T-shirt/top" while the correct class is "Shirt".



(a) MNIST (deformed)

(b) Fashion-MNIST

Figure 12: Confusion matrices and example correct and incorrect classifications for MNIST (affine deformation) and Fashion-MNIST.

4.2.3 Perturbations

Similar to Figure 4 of Sabour et al. [2017], we want to qualitatively observe and understand the learned embedding space and see what is parameterized by each component for each class. The embedding vector for our model is 160 components long, 16 for each output class. All of it is masked away except for the section for the predicted output class. By replacing each of the 16 non-zero components of a sample input image's embedding vector with a scalar between -0.25 and 0.25, we can observe what the high level meaning of these components is by running this perturbed vector through the network decoder, as seen in MNIST (Figure 13), Fashion-MNIST (Figure 14), SVHN (Figure 15), and CIFAR10 (Figure 16). Each row is a different component, and each column is a substituted value of a multiple of 0.05 from -0.25 to 0.25.

In MNIST, we can clearly see general components that correspond to stroke thickness, height, width, and skew, across all output digit classes. Then there are digit-specific ones that capture unique properties, like the whirl near the bottom left of some 2s, or the squashing of the enclosed part of the 6. In Fashion-MNIST, the embedding captures some general properties like height, width, grayscale color, and texture, and some specific properties like bag handle arc size and shape, the prominence of the tongue and topline for the shoe, or the arm length and waist size of the dress. SVHN components appear to mostly capture the color variation of the digits. Perturbations of CIFAR10 embeddings result in strong color changes as well. We suspect due to the shallowness of the decoder part of the network, both color and high class variance datasets have much worse reconstructions, and require a deeper network with larger embeddings.

4.2.4 PCA Visualization of Dynamic Routing

We used TensorBoard to visualize the drift of the digit output embeddings over the 3 dynamic routing iterations during the forward pass (Figures 17, 18, 19). In each figure, we can consistently see a large shift in the distribution of vectors as dynamic routing iteration progresses. The large spread of vectors generally in the second iteration gets reduced by the third, as they converge to one of the final classes.

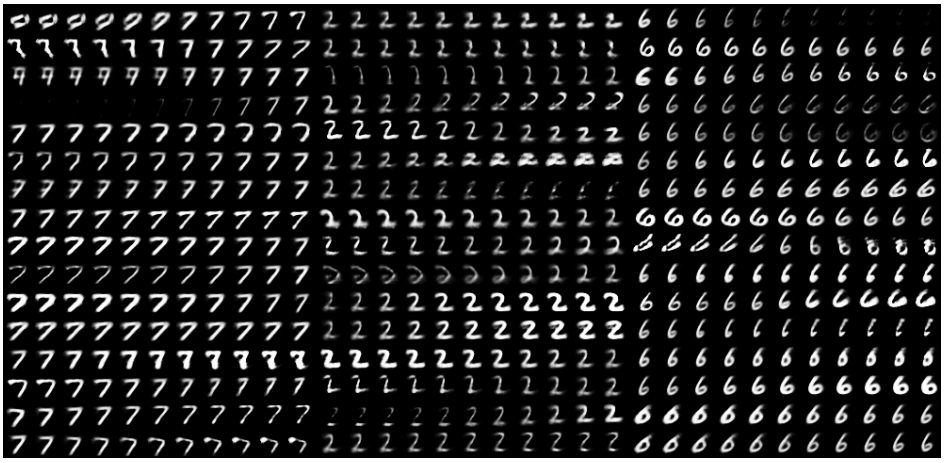


Figure 13: Perturbation of an MNIST sample embedding vector at epoch 34, 38, and 50 respectively.

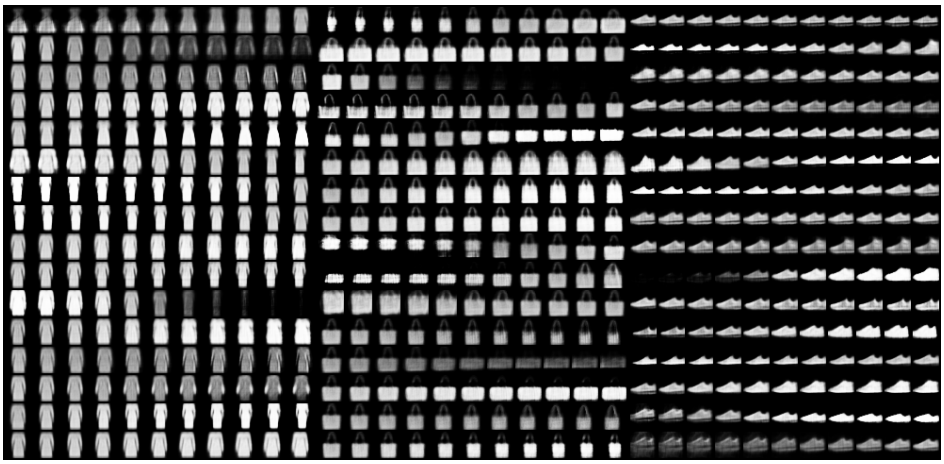


Figure 14: Perturbation of a Fashion-MNIST sample embedding vector at epoch 33, 34, and 38 respectively.

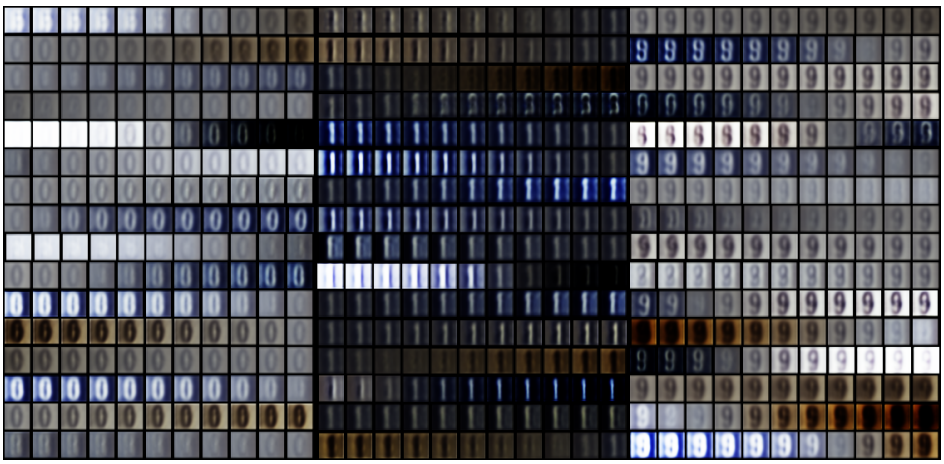


Figure 15: Perturbation of an SVHN sample embedding vector at epoch 34, 38, and 50 respectively.

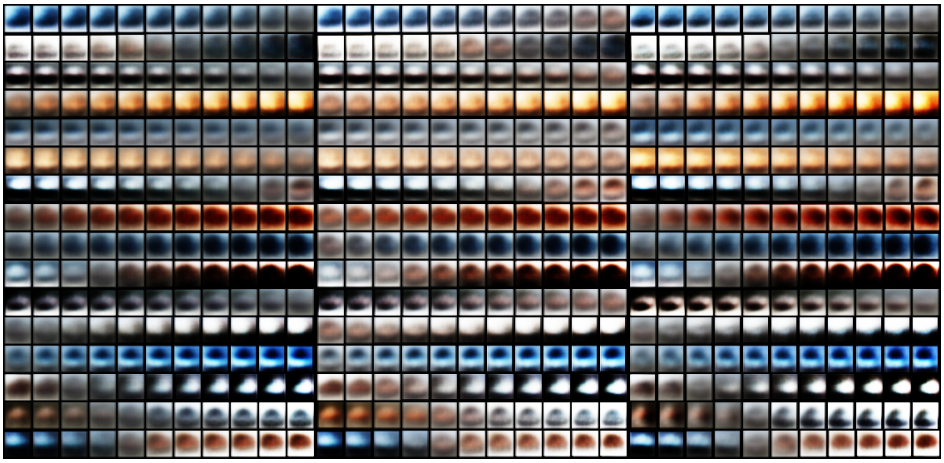


Figure 16: Perturbation of a CIFAR10 sample embedding vector at epoch 34, 38, and 50 respectively.

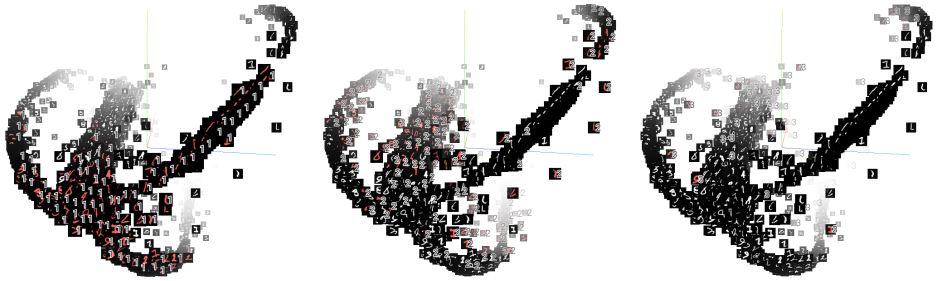


Figure 17: 3D sphereized PCA of drifting embedding vector outputs over 3 dynamic routing iterations (epoch 50 of MNIST).

Despite the spread from the second iteration, we showed above that 2 was sufficient for achieving the accuracy from the recommendation of 3 in Sabour et al. [2017].

4.2.5 Applying t-SNE to Embeddings

t-SNE visualizations project embeddings into a 2D or 3D space, putting similar vectors closer to each other in Euclidean distance. When coupled with the ground truth class labels, they visualize how close or far embeddings of the same class are from each other. We visualized all datasets in Figure 20.

Figure 20a shows near perfect clustering into 10 groups for MNIST. However, each of the other datasets is not nearly so neat. Fashion-MNIST is next in complexity, and the worst is CIFAR10. The

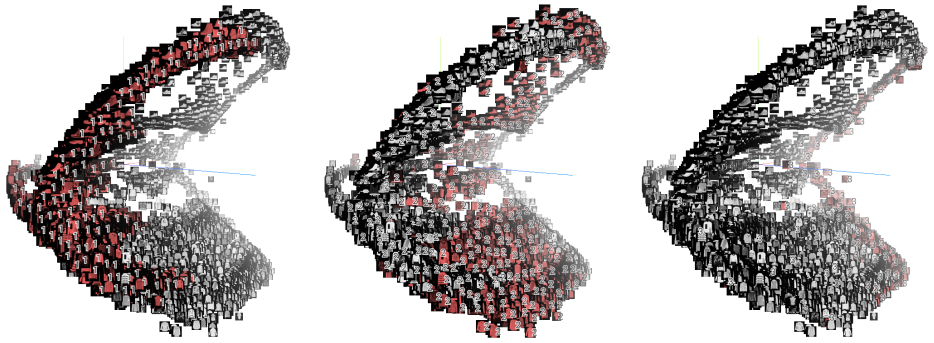
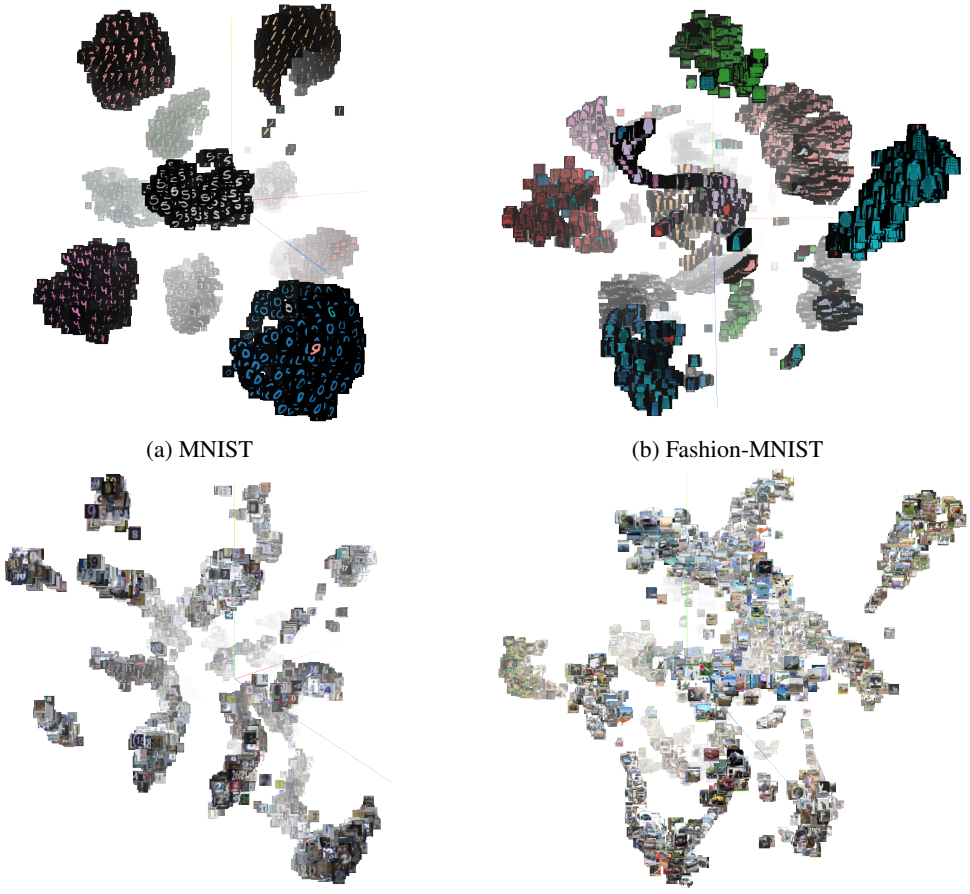


Figure 18: 3D sphereized PCA of drifting embedding vector outputs over 3 dynamic routing iterations (epoch 50 of Fashion-MNIST).

Figure 19: 3D sphereized PCA of drifting embedding vector outputs over 3 dynamic routing iterations (epoch 50 of SVHN).



(a) MNIST

(b) Fashion-MNIST

(c) SVHN

(d) CIFAR10

Figure 20: t-SNE visualizations of 10k embeddings (50 epochs, 3rd dynamic routing iteration).

disorder of embedding appears correlated to the test accuracy and quality of reconstruction. This means that the t-SNE visualization is a potentially informative complement to accuracy for assessing strategies for improving network performance on more complex datasets.

5 Conclusion and Future Work

In this work, we pushed the limits of CapsNets by exploring their performance and expressiveness when controlled, incremental changes were made to them. Our experiments yielded findings that are pertinent to the goals we outlined at the beginning:

1. CapsNets were able perform better than AlexNet on datasets marginally harder than MNIST, but they were not as good at coping with deformations as one might expect. Increasing the number of routing iterations from the recommended 3 did not help, and in fact, decreasing it to 2 did not harm performance. This suggests that although CapsNets are probably learning some useful information about spatial relationships between entities, they are not making full use of routing to encode them robustly.
2. CapsNets could convincingly reconstruct MNIST and Fashion-MNIST samples, whose embeddings seemed to measure meaningful qualities like thickness or skew. However, they could not do so for SVHN or CIFAR10, whose embeddings mainly encoded intensity and color. Across all datasets, most of the errors made by CapsNets corresponded to noisy or ambiguous reconstructions. The t-SNE visualizations of the embeddings further corroborated this, by showing that only MNIST had a cleanly separable label space.

Together, our findings indicate that the current CapsNet design is unlikely to work on other classification tasks, let alone machine learning tasks in general. This being said, we are not recommending that the idea be discarded; the concept of a capsule is intuitively appealing and they have demonstrated reasonable performance in our experiments. We believe that there is significant potential for CapsNets to be improved and made useful. The three paths we are most excited about are:

1. **Routing** The current routing algorithm is perhaps the most simple and intuitive way of deciding which high level capsules get assigned to which low level capsules. Fleshing it out to be more informed and less brittle could enable CapsNets to learn more complicated structure in the data. For example, a neural network could act as the mechanism that reweighs coefficients.
2. **Architecture** The current architecture is also relatively austere. There is only one layer of convolution for feature generation and only one layer of capsules before the end object capsules are built. Although this shallow structure might work on MNIST, it seems unreasonable to expect that all the nuances of a vehicle or animal in CIFAR could be represented with one level of entities. A deeper network, perhaps with some domain specific structure, might overcome this. Sabour et al. [2017] has some other experiments in this direction.
3. **Task** Finally, it could be the case that CapsNets would be good at other more complicated tasks, even though they are not the best at the "simple" task of classification. The task we are most curious about is segmentation, because perhaps the cosine similarities between embeddings computed by the CapsNet could be used as indicators of how likely it is that two pixels are in the same object.

6 Acknowledgements

We would like to thank Dr. Russakovsky and Dr. Ferencz for their tireless patience in teaching us the fundamentals of computer vision. This work would not be possible without the computational resources of the Visual AI Lab and Seung Lab and our starter code from Kenta Iwasaki of Gram.AI.

References

Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pages 44–51. Springer, 2011.

Kenta Iwasaki. capsule-networks: A PyTorch implementation of the NIPS 2017 paper "Dynamic Routing Between Capsules", January 2018. URL https://github.com/gram-ai/capsule-networks. original-date: 2017-11-02T14:47:54Z.

Stefan Keselj, Rohan Doshi, and Prem Nair. Capsule Network Experiments, January 2018. URL https://github.com/skeselj/capsule-network-experiments. original-date: 2018-01-12T00:34:45Z.

Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011.

Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic Routing Between Capsules. *arXiv:1710.09829 [cs]*, October 2017. URL http://arxiv.org/abs/1710.09829. arXiv: 1710.09829.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.