Factoring Out Augmentations to Increase Training Efficiency

Sammy Moseley (sjm352) [*1] **Karun Singh (ks939)** [*1]

Abstract

Although extremely helpful in improving the accuracy of deep neural networks, the process of data augmentation comes at a cost during training time. In this paper, we look for ways to separate augmentations from the main training pipeline so as to make the training process more efficient. We explore and compare multiple approaches to achieving this – including augmentation-invariant embeddings and fine-tuning. We found that augmentations are more easily learned in higher-level layers, and we use this observation to propose a general framework for factoring augmentations out of the main training pipeline.

1. Introduction

One of the shortcomings of deep learning is its reliance on high volumes of input data. Extensive training data can be hard to acquire, and furthermore, it might not fully cover all the types of inputs that might be seen at test time which can lead to poor generalization. Data augmentation is a technique that is commonly used to mitigate these issues – it involves artificially extended the training data using label-preserving transforms (for example, an image of a cat might be rotated, translated, or flipped horizontally, yet it still remains an image of a cat).

Having access to augmented data enables models to generalize better, since they can then train using examples that they otherwise would not have seen. This improvement, however, cannot be obtained for free; it comes with the added cost of increased training time, since the network now has to learn from a larger dataset.

In this paper, we try to 'factor out' augmentations from the main training pipeline in the hopes of reducing the overall cost of training. Our approach is driven by the fact that augmentations of the same data point share a lot of semantic information, and their features need not be re-learned repeatedly by the whole network. Instead, we look for ways to learn invariance to augmentations as a component that can be easily added to any conventional training pipeline.

To prototype our ideas and test our hypotheses, we picked the task of rotated image classification. We tried keep every part of our approach as general and transferable as possible, so that it can be applied to augmentations other than rotations, and domains other than images. Our first approach relies on Triplet networks to learn an embedding that is invariant to augmentations. Through several experiments, we observed several interesting patterns of behavior and reshaped our approach accordingly. Eventually, we propose a general technique that can be used to 'teach' a network about augmentations without using as many computational resources as the conventional data augmentation approach.

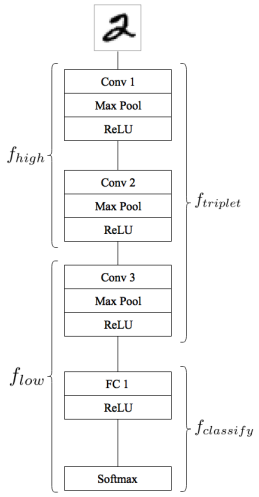
2. Related Work

An approach that resembles ours is transformation-invariant pooling (TI-Pooling) (Laptev et al., 2016). Inputs and their geometric transformations are first passed in parallel through an initial set of layers, followed by a special max-pooling layer that learns transformation-invariant features. The output is a canonical representation of the original example and its transformations, which is then used to train the rest of the network.

The TI-Pooling method itself bears a strong resemblance to the previously proposed Spatial Transformer Network (STN). STNs add a special module to the network architecture that can learn invariance to a specified class of image transformations. (Jaderberg et al., 2015). While such approaches have successfully been able to encode invariances, they necessitate changes in the neural network architecture that may not always be feasible with other less straight forward transformations and augmentations.

There have been additional specialized attempts in the past to make neural networks invariant to certain types of input transformations. For example, to gain rotational invariance, (Zhou et al., 2017) introduce Active Rotating Filters (ARFs) as an alternative to regular convolutional filters. Although ARFs are able to effectively capture rotational invariance, it is a highly specialized technique that cannot

[*]Equal contribution [1]Cornell University. Correspondence to: Sammy Moseley <sjm352@cornell.edu>, Karun Singh <ks939@cornell.edu>.



	Kernel Size	Output
Conv Block 1	$3 \times 3 \times 1 \times 4$	$14 \times 14 \times 4$
Conv Block 2	$3 \times 3 \times 4 \times 8$	$7 \times 7 \times 8$
Conv Block 3	$3 \times 3 \times 8 \times 16$	$4 \times 4 \times 16$
FC 1	256×10	10

Figure 1. The $28 \times 28 \times 1$ input image is fed into a set of three layers, each of which consists of successive 3×3 convolution, 2×2 max-pooling, and ReLU units. The first, second, and third layers output 4, 8, and 16 channels respectively. We call this initial section of the network $f_{triplet}$. The output of $f_{triplet}$ is of shape $4 \times 4 \times 16$, and feeds directly into a fully-connected layer with 10 outputs. These outputs are passed through a Softmax layer to obtain the final classification probabilities. We call this latter section of the network $f_{classify}$. The first two layers of the network are called f_{low}. The last convolutional layer and the fully-connected layer are called f_{high}.

easily generalize to other types of input transformation.

We draw inspiration from all of these approaches, with the added constraint of finding a technique that can be applied to any arbitrary types of augmentation, and can be integrated into any network architecture.

3. Background

3.1. Triplet networks

To learn augmentation invariance, we attempt to learn a function from the data domain to an embedding space in which augmentations of the same data point lie close to each other.

Conventionally, Siamese and Triplet networks have been used to learn embeddings that obey a desired distance metric. Siamese networks learn embeddings by reducing the distance between pairs of examples (Bromley et al., 1994). While Siamese networks have been used in the past for image recognition tasks (Koch et al., 2015), Triplet networks have yielded more success by enforcing a similarity as well as dissimilarity constraint (Hoffer & Ailon, 2014). In a Triplet network, three inputs are fed through the same embedding network: an anchor, a positive, and a negative. The objective of the embedding network is to formulate an embedding space where similar entities are close together, and dissimilar entities are far apart. Triplet networks have recently been successfully applied to tasks such as face recognition (Schroff et al., 2015) and person re-identification (Hermans et al., 2017).

The addition of margins to triplet loss (Schroff et al., 2015) and hard-mining (Hermans et al., 2017) to Triplet networks have resulted in significant improvements. Adding a margin constraint improves the quality of embeddings by enforcing a certain distance between the embeddings of the positive and negative inputs relative to the anchor input. Hard-mining actively looks for hard triplets to feed into the network so that it can learn valuable information in every iteration, rather than seeing easy, trivial triplets.

3.2. t-SNE Visualization

To visualize the embeddings learned by the Triplet network, we use the t-Distributed Stochastic Neighbor Embedding (t-SNE) algorithm (van der Maaten & Hinton, 2008). t-SNE is popularly used to visualize high-dimensional data because of its ability to find a faithful representation of the data in lower-dimensional spaces. We used a SciPy implementation of t-SNE to create our visualizations.

4. Methods

4.1. Dataset Generation

To evaluate our methods, we decided to consider images, since many deep learning applications are fed images as input. Specifically, we construct our own variation of the MNIST dataset using an approach similar to that of (Zhou et al., 2017). We consider rotational augmentations – one of the most common types of augmentations used on images.

To make the effect of data augmentation more pronounced, we create a variation of the MNIST dataset called MNIST-$tilt$. Examples from the original MNIST dataset were randomly rotated by an angle in the range $\left(-\frac{\pi}{6}, \frac{\pi}{6}\right)$. We further created MNIST-$tilt^+$, where every training sample from MNIST-$tilt$ was augmented with 4 rotations, by angles $-\frac{\pi}{6}$, $-\frac{\pi}{12}$, $\frac{\pi}{12}$, and $\frac{\pi}{6}$. Note that MNIST-$tilt^+$ contains 5 times as many examples as MNIST-$tilt$.

4.2. Network Architecture

For all of our experiments, we fix the network architecture to that shown in Figure 1. $f_{triplet}$ is the section of the network that serves as a Triplet network when we learn an embedding space. $f_{classify}$ is the section that is responsible for generating a classification prediction. The early, lower layers of the network are denoted by f_{low}, while the later, higher layers are called f_{high}.

4.2.1. Triplet network architecture

A Triplet network is called so because it contains three copies of the same set of underlying network weights. In our case, the $f_{triplet}$ layers provide these network weights. To train the network, three inputs – together called a triplet – are fed into the three network replicas. Triplets consist of an anchor input x_a, a positive input x_p, and a negative input x_n. The network outputs a result for each of the three inputs, and these results are used to formulate the loss function, which is described below.

4.3. Training

4.3.1. Triplet network loss

To train the Triplet network, we experimented with two types of loss functions: Softmax, and triplet loss with soft margin. We define $D_{i,j} = D(f_{triplet}(x_i), f_{triplet}(x_j))$, where D measures the L2 distance between two given inputs. Let $d_+ = D_{a,p}$ and $d_- = D_{a,n}$.

1. **Softmax loss**, as described in (Hoffer & Ailon, 2014), applies the Softmax function to d_+ and d_- to obtain the values p_+ and p_-. The final loss is defined as follows:

   $$L_{softmax}(p_+, p_-) = p_+^2$$

   This formulation effectively looks at each triplet as a 2-class classification problem, while trying to answer whether the anchor is part of the positive or negative class.

2. **Triplet loss with soft margin**, as described in (Hermans et al., 2017):

   $$L_{soft\_triplet}(d_+, d_-) = \ln(1 + e^{d_+ - d_-})$$

   This loss function encourages the network to push the negative embeddings far away from the anchor, while pulling the positive embeddings closer to the anchor.

On trying both of these loss functions, we consistently achieved better results with the soft margin Triplet loss, and thus fixed it as our Triplet network's loss function for the evaluations. For the rest of the paper, we refer to the Triplet network's loss function as $L_{embed} = L_{soft\_triplet}$, since it is responsible for learning an embedding.

4.3.2. Triplet network batches

The triplet generation process is crucial to learning the desired embedding space. In our case, we want the embedding to be invariant to image augmentations. Thus, we want augmentations of the same image to lie close together. As a first attempt, we tried random triplet generation. We selected an image from MNIST-$tilt$ as x_a and an augmented version of x_a from MNIST-$tilt^+$ as x_p. For x_n, we selected a random image from MNIST-$tilt^+$ that was *not* an augmentation of x_a.

Using this method, we found that the network does not learn a useful embedding since a majority of the triplets it sees are easy. To combat this, we incorporated batch hard-mining (Hermans et al., 2017) into our approach.

Batch hard-mining involves first generating a batch of images of size PK, where K examples are sampled from each of the P chosen classes. In our adaptation of this technique, two images are considered to be part of the same 'class' if they are augmentations of each other. Then, each of the PK examples is considered as the anchor of a triplet. For each one, the hardest positive and hardest negative examples within the batch are found to form a triplet. In our case, we achieved the best results using a relaxed version of this approach. Instead of picking the hardest negative example, we randomly pick a negative example from the hardest 20% of candidate images. Without this, our network – perhaps due to its limited capacity – tended to diverge in the early stages of training.

4.3.3. Classification loss

We use cross-entropy loss as our classification loss. If p represents the Softmaxed output and y represents the one-hot ground truth vector, the loss function can be described as:

$$L_{class} = -\sum_i y_i log(p_i)$$

4.3.4. Overall process

We experimented with two different overall training procedures, as described below:

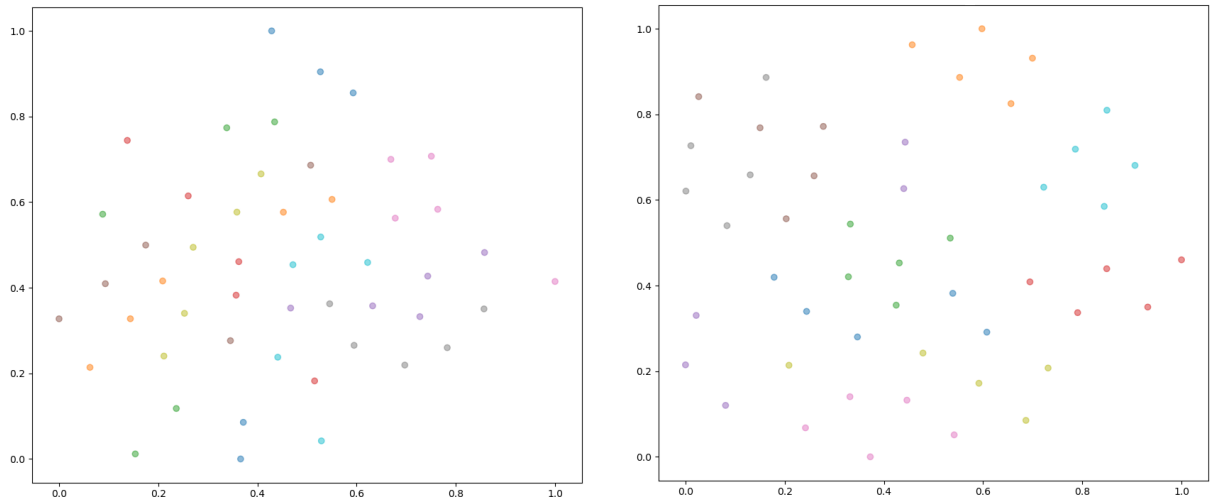


Figure 2. t-SNE visualization of pixel space (left) and the learned embedding space after 10000 iterations (right). 10 examples – one from each digit – are chosen, along with their 4 augmentations each within the dataset. Each of the 10 examples is represented by a single color, so points of the same color are augmentations of each other. In pixel space, augmentations are often very scattered. The embedding space, although not perfect, does seem to help in bringing augmentations closer to each other.

1. Init(e, c): Train $f_{triplet}$ for e iterations to minimize L_{embed}. Use this as an initialization for c end-to-end iterations to minimize L_{class}.
2. Fine-Tune(c, e), shortened as FT(c, e): Train the network end-to-end for c iterations to minimize L_{class}. Freeze f_{low}, and train f_{high} for e iterations using the following hybrid loss function: $L_{ft} = \alpha L_{class} + (1 - \alpha) L_{embed}$ For our evaluations, we found it best to set $\alpha = 0.9$.

We used the Adam optimizer (Kingma & Ba, 2014) with default parameters while training. Dropout with $p = 0.5$ was used in all evaluations.

5. Results

5.1. Can triplet networks learn such an embedding?

As the first step, we checked whether the Triplet network was even capable of learning an embedding where augmentations lie close to each other. To do so, we trained $f_{triplet}$ for 10000 iterations starting from a random initialization, and monitored the embedding loss to ensure convergence. A visualization of the learned embedding space compared to pixel space is shown in Figure 2.

5.2. Is this embedding useful for classification?

The next question we looked to answer was whether this learned embedding was of any use or not for the task of classification. To do so, we used the training procedure Init$(5000, 5000)$. The first 5000 iterations were dedicated to learning an augmentation-invariant embedding. The next 5000 iterations started with the learned embedding network as an initialization, and looked to minimize the network's classification loss. We tracked the embedding loss (shown in Figure 3) throughout this run to assess whether the classifier found the initialization useful or not.

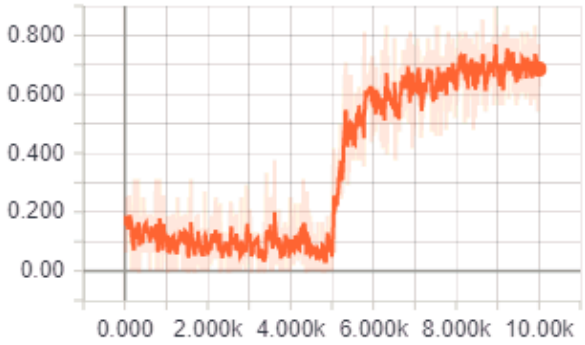


Figure 3. Embedding loss (y-axis) plotted against training iterations (x-axis) for the Init$(5000, 5000)$ run. At iteration 5000, the network starts minimizing L_{class} instead of L_{embed}

As the figure shows, when the network starts learning to classify, there is an abrupt increase in embedding loss. This is an indication that the classification objective does not find the embedding initialization useful; the embedding is unlearned once the classification training starts. This result

might be explained by a few different hypotheses:

1. Learning augmentation invariance before learning anything about how certain classes look might not be a good idea. Understanding augmentations is dependent on a good understanding of the input's semantics, which can only be gained through prior classification training.

2. Imposing the augmentation invariance constraint on the early layers of the network might not be the way to go. The earlier layers might be better off extracting lower-level features. The concept of augmentations is one that is easier to encode in the higher layers that have access to more semantic information.

3. The optimal arrangement of the embedding space for the classification task may not involve putting all augmentations of the same image close together. With that in mind, the embedding loss might be better used as a regularizer rather than the dominant objective.

5.3. An alternative approach

To fix the potential issues underlying the previous approach, we decided to come up with an alternate training process. The FT(c, e) training procedure stems directly from the three hypotheses presented above. Firstly, it only attempts to learn about augmentations *after* it has learned sufficient features for basic classification on the original data. Secondly, the embedding constraint is only imposed on the higher layers of the network. Thirdly, we use L_{embed} simply as a regularizer for L_{class} rather than as its own objective function.

We run FT$(5000, 5000)$ and compare its validation accuracy to two baselines approaches. The first baseline, B, is the same network architecture trained end-to-end for 10000 iterations to minimize L_{class}, with the constraint that its training data comes exclusively from MNIST-$tilt$. The second baseline, B^+ is the same as B except it additionally sees augmented examples from MNIST-$tilt^+$. Figure 4 summarizes the results of this comparison.

As the figure shows, FT$(5000, 5000)$ benefits from an increase in accuracy once it begins to minimize L_{ft} instead of purely minimizing L_{class}. In fact, the 5000 fine-tuning iterations help it achieve an accuracy close to that of B^+, which has been trained on augmentations from the very first iteration. It is worth noting that since the 5000 fine-tuning iterations of FT$(5000, 5000)$ only train a subset of the network's layers, it carries out far fewer weight updates compared to the two baseline methods.

This result is a sign that learning about augmentations need not involve training all the layers of the network. Instead, it might be sufficient to only show augmentations to the higher layers.

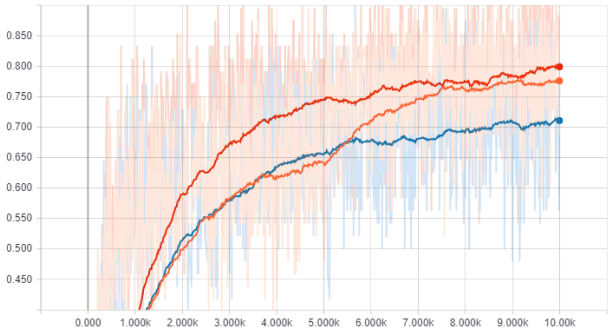


Figure 4. Validation accuracy (y-axis) of FT$(5000, 5000)$ (orange), baseline B (blue) and baseline B^+ (red) plotted against 10000 training iterations (x-axis). The fine-tuning phase starts at iteration 5000, and this corresponds to a marked jump in accuracy of the FT network.

5.4. Is the embedding constraint important?

With this result, we aimed to isolate the cause of this accuracy jump in the fine-tuning phase. In this phase, the network begins to see augmentations for the first time via the Triplet network's embedding constraint. An increase in accuracy is noted in this phase; however, what role does the embedding constraint play in achieving this improvement?

To assess how the embedding constraint is impacting model performance, we trained the network using Augmentation-FT$(5000, 5000)$ – a slightly tweaked version of $FT(5000, 5000)$. In this version, we impose no embedding constraint at all. All 10000 iterations aim to minimize L_{class}, but the first 5000 iterations draw examples from MNIST-$tilt$ only, while the subsequent iterations additionally see augmentations from MNIST-$tilt^+$. Augmentation-FT achieves a higher test accuracy compared to not just FT, but also B^+ (Table 1). This points towards the fact that the embedding constraint that we formulated may not be very helpful during classification.

The eventual test accuracies of the training variations discussed are presented in Table 1.

Table 1. Test accuracy on MNIST-$tilt$ of different training procedures after 10000 iterations each.

	B	B+	FT	Aug-FT
Test accuracy	0.700	0.796	0.766	0.813

5.5. Proposed framework

Our proposed framework for factoring out image augmentations from the main training pipeline follows directly from the Augmentation-FT result from above. We believe that there are huge potential training cost savings if one follows the procedure below:

1. Train the network without showing it any augmented data
2. Train only the highest layers of the network using augmented data

6. Discussion

6.1. What we learned

Our initial idea was to learn embeddings in an augmentation-invariant space, and to then use this learned embedding to train a classifier. As we repeatedly reshaped our proposed technique through multiple experiments, we learned a few important things:

1. Imposing such an embedding constraint does not necessarily align with the internal representation learned organically by convolutional neural networks.
2. Invariance to augmentations is better learned *after* basic features of the data have already been learned.
3. One can get away by showing augmentations only to higher layers of the network that have access to semantic information. This can be used to gain potentially huge training cost savings.

6.2. Future directions

6.2.1. Generalization

Our proposed framework does not contain any components specific to certain types of augmentations, input domains or tasks. Therefore, the natural next step is to evaluate it in different settings to see if it generalizes.

6.2.2. Quantifying cost savings

Being able to quantify the actual training cost savings of such an approach in terms of floating point operations per second or training time would also be an important next step.

6.2.3. Impact on model robustness

As (Papernot et al., 2015) showed, learning a decision boundary in a smoother space can help increase a model's robustness to input perturbations. Motivated by the t-SNE projection of augmentations in pixel space (Figure 2, left) where augmentations of the same image often lie far apart, we believe that training a network end-to-end on augmentations might hurt its robustness. If the network has to learn a very jagged decision boundary in pixel space to capture augmentations, it might end up being more susceptible to perturbations. An interesting evaluation in the future could be assessing the impact of data augmentation on model robustness, and further comparing our proposed approach to the conventional augmentation approach based on that parameter.

7. Conclusion

Through this work, we were able to arrive at a technique that can potentially help neural networks gain invariance to augmentations while making the training process more efficient. While the method still needs to be extensively tested across different types of augmentations, different input domains, and different tasks, the process of coming up with it has taught us a lot about the nature of augmentations and convolutional neural networks.

If validated, such a technique is exciting to us because it would allow us to think of augmentation invariance as a module that can be 'applied' to neural networks, rather than being deeply integrated into the training pipeline. Simultaneously, the cost savings of such a method could possibly help in accelerating the training process, which is often a roadblock for applications in industry.

References

Bromley, Jane, Guyon, Isabelle, LeCun, Yann, Sackinger, Eduard, and Roopak, Shah. Signature Verification using a "Siamese" Time Delay Neural Network. 1994.

Hermans, Alexander, Beyer, Lucas, and Leibe, Bastian. In Defense of the Triplet Loss for Person Re-Identification. *arXiv:1703.07737 [cs]*, March 2017. URL http://arxiv.org/abs/1703.07737. arXiv: 1703.07737.

Hinton, Geoffrey, Vinyals, Oriol, and Dean, Jeff. Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [cs, stat]*, March 2015. URL http://arxiv.org/abs/1503.02531. arXiv: 1503.02531.

Hoffer, Elad and Ailon, Nir. Deep metric learning using Triplet network. *arXiv:1412.6622 [cs, stat]*, December 2014. URL http://arxiv.org/abs/1412.6622. arXiv: 1412.6622.

Jaderberg, Max, Simonyan, Karen, Zisserman, Andrew, and Kavukcuoglu, Koray. Spatial Transformer Networks. *arXiv:1506.02025 [cs]*, June 2015. URL http://arxiv.org/abs/1506.02025. arXiv: 1506.02025.

Kingma, Diederik P. and Ba, Jimmy. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. URL http://arxiv.org/abs/1412.6980. arXiv: 1412.6980.

Koch, Gregory, Zemel, Richard, and Salakhutdinov, Ruslan. Siamese Neural Networks for One-shot Image Recognition. 2015.

Laptev, Dmitry, Savinov, Nikolay, Buhmann, Joachim M., and Pollefeys, Marc. TI-POOLING: transformation-invariant pooling for feature learning in Convolutional Neural Networks. *arXiv:1604.06318 [cs]*, April 2016. URL http://arxiv.org/abs/1604.06318. arXiv: 1604.06318.

Papernot, Nicolas, McDaniel, Patrick, Wu, Xi, Jha, Somesh, and Swami, Ananthram. Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks. *arXiv:1511.04508 [cs, stat]*, November 2015. URL http://arxiv.org/abs/1511.04508. arXiv: 1511.04508.

Schroff, Florian, Kalenichenko, Dmitry, and Philbin, James. FaceNet: A Unified Embedding for Face Recognition and Clustering. *arXiv:1503.03832 [cs]*, pp. 815–823, June 2015. doi: 10.1109/CVPR.2015.7298682. URL http://arxiv.org/abs/1503.03832. arXiv: 1503.03832.

van der Maaten, Laurens and Hinton, Geoffrey. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 2008.

Zhou, Yanzhao, Ye, Qixiang, Qiu, Qiang, and Jiao, Jianbin. Oriented Response Networks. *arXiv:1701.01833 [cs]*, January 2017. URL http://arxiv.org/abs/1701.01833. arXiv: 1701.01833.

8. Appendix

8.1. Source Code

This research was done using TensorFlow. Code has been made publicly available on GitHub (github.com/sammyjmoseley/embedding-research)

8.2. Distillation

In addition to the above approaches, we attempted to distill an embedding for augmentations from a trained network. Distillation has been shown to be effective in transferring knowledge from trained models to smaller models (Hinton et al., 2015). In the distillation approach, we try to distill the implicit embedding created by a convolutional network trained on MNIST, to a similar architecture trained on MNIST-$tilt$ such that the rotated images (which can be thought of as augmentations) are mapped close to their original MNIST images.

We first train one model to classify MNIST images. Then, we create a network with two different child networks. One child network is the embedding part of the pre-trained MNIST classifier, and the other is a new network that we wish to use to embed augmentations. We freeze the pre-trained network and feed into it a reference image. We then feed into the new embedding network an augmented form of the reference image, and try to minimize the L2 distance between the two network outputs.

We tried several different embedding networks. We found that using the same architecture as the frozen network was not adequate in capturing the complexity of the augmentations. To sufficiently embed the augmentations, we used a network consisting of three convolution-max-pool blocks with 10 times the number of filters as the original network, followed by two fully-connected layers.

After we trained the new embedding network, we trained it along with the final readout layers on non-augmented data once more, to help account for differences between the original embedding space and the newly trained embedding space.

The results were not as positive as we were hoping. In line with our other results, we think that this approach was unsuccessful largely because lower layers are unable to capture the complexity of augmentations. For future work, it could be interesting to try this approach without changing lower layers, and instead trying to get higher layers to embed augmented images in the same embedding space as the original images.