INSTITUT FÜR INFORMATIK
Machine Learning

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Adversarial Attacks on Capsule Networks

Felix Michels

Bachelorarbeit

Beginn der Arbeit:	08. April 2019
Abgabe der Arbeit:	21. Juli 2019
Gutachter:	Prof. Dr. Stefan Harmeling
	Prof. Dr. Stefan Conrad

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 21. Juli 2019

Felix Michels

Abstract

This bachelor thesis extensively evaluates the vulnerability of capsule network to various adversarial attacks. Capsule networks are a class of artificial neural network in which neurons are grouped in capsules whose activity vectors describe different properties of the same entity. Recent work attributes them more robustness to adversarial attacks than other types of neural networks.

To investigate those conjectures we train four pairs of capsule networks and convolutional neural networks to classify different image datasets. Using multiple strong and diverse adversarial attack we compute adversarial examples for each combination of dataset and architecture.

By comparing the Euclidean norms of the generated perturbations we find that capsule networks are vulnerable to adversarial attacks in a similar way as convolutional networks. We discover in particular that capsule networks show no robustness to sufficiently strong white-box attacks. Additionally, we demonstrate limited transferability of the adversarial examples between the two types of architectures.

Furthermore, we study the structure of the adversarial perturbations and observe that they seem to lie in a relatively low dimensional linear subspace for both types of networks. Despite those similarities we detect inherent differences between the perturbations constructed for the different network types using dimensionality reduction.

Finally, we aim to identify the cause of adversarial vulnerability in the networks we trained. Since capsule networks and convolutional networks partially use the same type of convolutional layer, we focus on the disturbance on the activation of individual layers caused by adversarial attacks. We find that susceptibility to attacks is not just ascribable to the first layers in a network but a result of deeper layers as well. Specifically, we find that the dynamic routing algorithm used in capsule networks does not establish capsule activation invariance to counteract adversarial attacks as previously conjectured.

Contents

1 Introduction

Capsule networks (CapsNets) are a novel form of artificial neural networks introduced by Sabour et al. (2017) and Hinton et al. (2018). CapsNets partition the neurons of a layer into groups called *capsules* and route their output dynamically. This aims to preserve information about hierarchical spatial relationships between objects and thereby to generalize better to new viewpoints. In some computer vision tasks CapsNets have reached the state-of-the-art performance of convolutional neural networks (ConvNets), and have surpassed them in some particular functions like segmenting highly overlapping objects. As a possible alternative to ConvNets, they also face some of the same challenges.

With the recent emergence of deep learning in many real-world applications, from face recognition to brain tumor detection and satellite imagery analysis, security in machine learning has become a growing concern. Adversarial attacks refer to techniques designed to cause machine learning models to perform poorly. An attack can start before the model is even trained, e.g. by maliciously manipulating the training data (poisoning), but test-time attacks are more widely applicable and pose a bigger threat.

In computer vision tasks specifically *adversarial examples* (Szegedy et al., 2013) refer to images that are designed in such a way that a machine learning system make mistakes, but still appear to be an ordinary image to a human viewer. Goodfellow et al. (2014) hypothesize that linear behavior in high-dimensional spaces is the cause for the existence of adversarial examples and develop an attack which finds adversarial examples for deep neural networks with particular ease. This is not just problematic for security-sensitive applications like self-driving cars, but also casts doubt on their ability to generalize in a meaningful way.

The effects of adversarial attacks on CapsNets have not yet been studied in a satisfactory manner. We will investigate the robustness of CapsNets in comparison to ConvNets against multiple common adversarial attacks and attack scenarios across four different popular image datasets. We also look at the structural difference of the generated adversarial examples, the transferability of perturbations between architectures and unit activations between layers. We show that CapsNets are in general no less susceptible to adversarial examples than ConvNets.

This thesis is structured as follows: in Section 3 we reiterate the concept of capsule networks and the routing-by-agreement algorithm. In Section 4 we describe the attacks we use to test the CapsNets. We present the results and interpretations of our experiments in Section 5.

2 Related Work

Hinton et al. (2018) show that CapsNets with EM routing are less vulnerable to the simple white-box attacks FGSM (Goodfellow et al., 2014) and the Basic Iterative Method (Kurakin et al., 2016) than standard ConvNets. Marchisio et al. (2019) study adversarial robustness of CapsNets using the usual routing-by-agreement, although they focus on the performance of their own proposed black-box algorithm and their own definition of

imperceptibility. Frosst et al. (2018) claim the robustness of CapsNets against white-box attacks and approach the problem further by detecting adversarial examples through use of the reconstruction network, which is trained alongside the CapsNet in most applications anyway. Peer et al. (2018) introduce a further routing algorithm and demonstrate its robustness against FGSM.

3 Capsule Networks

3.1 History and Concept

Hinton et al. (2000) introduced *credibility networks* to combine image segmentation and recognition by representing images using parse trees. Later Hinton et al. (2011) coined the term *capsule* for substructures of neural network that learn to recognize implicitly defined entities by performing some—possibly quite complicated—internal computation and outputting not just the probability of the entity's presence but also a set of *instantiation parameters* encoding the entity's attributes. The idea was that invariant unit activations with respect to viewpoint changes are impossible to achieve in conventional neural networks without losing spatial information, e.g. through the use of max pooling. Properly working capsules should instead establish equivariance in their instantiation parameters and keep only the probability that the entity is present invariant.

While conventional neural networks form a entangled representation of features in each layer, capsules aim to produce dedicated representations. CapsNets may utilize these separated representations to improve the information flow to the next layer. Ordinary neural network with scalar valued units can control gradient routing in some limited way with max pooling or similar operations, but since the output of each capsule is a vector (or a matrix) some advanced *routing algorithm* may be employed.

Sabour et al. (2017) first developed such a dynamic routing algorithm, routing-by-agreement, constructed a concrete CapsNet architecture and applied it for the segmentation of highly overlapping digits and other tasks with remarkable results. Most actual CapsNets use a linear transformation as the internal action for each capsule, but there have been multiple other routing algorithm introduced, most notable EM routing (Hinton et al., 2018) but also scaled-distance-agreement (Peer et al., 2018), cognitive consistency routing (Li, 2018) and more.

Given that routing-by-agreement is still the most commonly used method, we will exclusively consider this approach in this thesis.

3.2 Routing-By-Agreement

A capsule in the routing-by-agreement setting consists of a vector in the unit ball. Its direction encodes the instantiation parameters and its Euclidean length the probability that the attached entity is present.

For a capsule i of dimension d_1 let us designate the output with $u_i \in \left\{ \mathbb{R}^{d_1} \;\middle|\; \|u\|_2 \leq 1 \right\}$. If i is connected to a capsule j in the next layer with dimension d_2, the network includes

a learned transformation matrix $W_{ij} \in \mathbb{R}^{d_1 \times d_2}$. Thus we can compute $\hat{u}_{j|i} = W_{ij} u_i$. The vector $\hat{u}_{j|i}$ can be seen as a prediction of the feature's parameters modeled by capsule j given the information of the feature in capsule i. The output for the capsule j then is a linear combination with dynamically computed coupling coefficients of these predictions over all connected capsules in the previous layer as described in Algorithm 1. The routing-by-agreement algorithm aims to find coefficients such that the instantiation parameters of predictions $\hat{u}_{j|i}$ with high coupling coefficients c_{ij} match those of the capsule j in the next layer, i.e. the angle between $\hat{u}_{j|i}$ and v_j is low. Thereby CapsNets are meant to achieve the "explaining away" necessary for segmenting overlapping objects.

The transformation matrices W_{ij} can be trained by backpropagation through the unrolled iterations of the routing algorithm. The b_{ij} used in Algorithm 1 are logits for the coupling coefficients, therefore their initial value can be seen as a bias for the routing algorithm and can also be trained. This seems to offer little benefit, so we initialize them as zero (as done by Sabour et al., 2017).

As in conventional neural networks there are multiple types of layers depending on the possible connections to the units in the lower layer. If each capsule in a layer is connected to every capsule in the layer before it, we call it a *dense capsule layer*. Like in a convolutional layer, capsule can also be arranged in a spatial grid (in the shape of $height \times width \times types \times dimension$) where each capsule is only connected to those in it's receptive field and capsules of the same type share weights. Such a configuration is called a *convolutional capsule layer*. Of course such layers, and the routing used between them, are only applicable if two capsule layers are placed consecutively. To convert the output of scalar layers to capsules we use *primary capsule layers*. These layers either apply a fully connected or a convolutional layer to their input, reshape the result into capsules and squash them so that their vector length is between zero and one.

A typical CapsNet for image classification begins with one or more convolutional layers, followed by a primary convolutional layer, optionally some number of convolutional capsule layers and finally a dense capsule layer with one capsule per class. In a fully trained CapsNet those class capsules should contain all instantiation parameters necessary to recreate the input image. Thus a *reconstruction network* that takes the output of the class capsule with largest norm as input and tries to reconstruct the input image can be trained simultaneously. The Euclidean distance between the original image and reconstruction, the *reconstruction loss*, may be minimized as a regularization technique. Since images also often contain some unneeded information that cannot be sensibly linked to a capsule in the next layer, an additional capsule may be used to serve as a dead end. This is referred to as the *none-of-the-above* category (Sabour et al., 2017).

Algorithm 1 Routing-by-agreement as proposed by Sabour et al. (2017) with r routing iterations and predictions $\hat{u}$ of the lower layer capsules.

procedure ROUTING($\hat{u}, r$)
  $\forall i, j : b_{ij} \leftarrow 0$
  for r iterations **do**
    $c_{ij} \leftarrow \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$ ▷ Softmax
    $s_j \leftarrow \sum_i c_{ij} \hat{u}_{j|i}$
    $v_j \leftarrow \frac{\|s_j\|_2}{1+\|s_j\|_2^2} s_j$ ▷ Squash capsules
    $b_{ij} \leftarrow b_{ij} + \langle v_j, \hat{u}_{j|i} \rangle$
  return v

4 Adversarial Attacks

In general, adversarial attacks describe any techniques with the purpose to disturb the performance of machine learning models. While adversarial attacks exist for a wide variety of applications like speech recognition or natural language processing, this work will concentrate on adversarial attacks on image classification tasks. *Adversarial examples* refer to modified images which are very close to some original image, but are classified differently by the model. In our context they are in particular only mentioned in relationship to the original image. The *adversarial perturbation* is the difference between the adversarial example and the original.

There are various attack scenarios differing in the knowledge, goals and abilities of the attacker. In *white-box* attacks all information about the model architecture, training images and learned parameters is available to the attacker, while in the *black-box* scenario only the networks output can be observed. In particular, gradients with respect to the input can be computed in the white-box setting. These terms are in general not strictly defined and many intermediate stages exist, including limited access to the models output, information about the architecture but not the learned parameters or knowledge about the training images but not the architecture etc.

Targeted and *untargeted* describes the objective of the attacker. With an untargeted attack an adversarial example is just classified as any other label than the true label, while an adversarial example from a targeted attack has to be classified as a given target label chosen by the attacker beforehand. Targeted attacks are strictly more difficult for the attacker, but also more useful. Furthermore, in some instances untargeted adversarial examples are easy to generate, or rather are near impossible to defend against, just because some classes may be conceptually very close to each other. For example, differentiating between a Golden Retriever and a Labrador Retriever in ImageNet (Deng et al., 2009) can even be quite challenging for humans.

The last type of adversarial attacks that we will test are *universal perturbations*. Most attacks compute a different adversarial perturbation for each input image, but a universal perturbation aims to fool many images while using the same perturbation. Non-universal perturbations are easier to find, but universal perturbations are simpler for the attacker to apply in a realistic attack scenario. Additionally, the mere existence of universal perturbations reveals significant flaws in the used architecture, more so than that of normal

adversarial examples. The presence of these particular perturbation hints at correlations of abstract features learned by the classifier. This is related to the notion that deep layers in neural networks do not disentangle variation factors across their units (Szegedy et al., 2013).

An important problem to consider is the definition of *closeness* to the original image. Possibilities for such an image similarity metric include simple L^p norms (including the quasinorms for $0 \leq p < 1$), or more sophisticated metrics, like entropy based metrics (Göpfert et al., 2019). While no metric exists which reliable simulates human perception of image similarity, and L^p norms may be insufficient in this regard (Sharif et al., 2018), they still give us information about the robustness of the neural networks. Here we will examine adversarial perturbations optimized towards a small L^2-norm (i.e. Euclidean norm), since it is often used in discussion about adversarial attacks and is a good compromise between the other two commonly used, but more specific scenario targeting L^0 and L^∞ norms.

Throughout this work $x \in [0,1]^n$ will denote the (flattened) input image and $\delta \in \mathbb{R}^n$ the perturbation. The label assigned to x by the neural network will be referred to as $C(x)$, while the correct label is $C^*(x)$.
The networks output as logits is $Z(X)$ and the output interpretable as probabilities is $F(x)$. This means, in the case of the ConvNet we have $F(x) = \mathrm{softmax}(Z(x))$ and in the case of the CapsNet $Z(x) = \mathrm{arctanh}(2F(x) - 1)$.
Furthermore, any gradient in the following section refers to the gradient with respect to the input image x and not the network's parameters.

4.1 Carlini-Wagner Attack

Carlini and Wagner (2017) extensively evaluated a wide range of adversarial attacks. In particular they developed the following targeted white-box attack, which will be referred to as the Carlini-Wagner (CW) attack.

For a given label t, the Carlini-Wagner attack aims to solve the constrained optimization problem

$$\begin{aligned} \underset{\delta}{\text{minimize}} \quad & \|\delta\|_2 \\ \text{subject to} \quad & C(x+\delta) = t \\ & x + \delta \in [0,1]^n. \end{aligned} \tag{1}$$

This is the general problem for targeted attacks. Due to the highly non-convex nature of deep neural networks, it can be very difficult to solve directly. Using Lagrangian relaxation, Equation (1) is transformed to the box-constrained problem

$$\begin{aligned} \underset{\delta}{\text{minimize}} \quad & \|\delta\|_2 + c \cdot f(x+\delta) \\ \text{subject to} \quad & x + \delta \in [0,1]^n, \end{aligned} \tag{2}$$

where $c > 0$ is a suitable chosen constant and f is an *objective function*, i.e. a function with the property that $f(x+\delta) \leq 0$ if and only if $C(x+\delta) = t$. There are many possible objective functions, but a particularly good choice is

$$f(x') = \max\left\{ \max\left\{ Z(x')_i \;\middle|\; i \neq t \right\} - Z(x')_t, -\kappa \right\}. \tag{3}$$

The *confidence parameter* κ determines with how much confidence the adversarial example is classified as the target class, i.e. how close to the decision boundary it is. The optimal value for c in Equation (2) is the smallest value that results in an adversarial example and can be found using a binary search.

The box-constrained problem can further be simplified by introducing the variable $w \in \mathbb{R}^n$ and setting $\delta_i = \frac{1}{2}(\tanh(w_i)+1)$. This substitution yields an unconstrained problem, which can be solved with various popular methods. We use the Adam optimizer (Kingma and Ba, 2014).

The Carlini-Wagner attack is generally a quite strong attack and leads to almost undetectable adversarial examples. In particular, it can find adversarial examples even when defensive techniques like distillation are utilized (Carlini and Wagner, 2017). However, due to the binary search it can often be rather slow.

4.2 Boundary Attack

The boundary attack is a black-box attack proposed by Brendel et al. (2017). It is furthermore a *decision based* attack. This means, not only is no knowledge of the model architecture or the learned weights required, but also the output scores of the network are hidden from the attack. Only the final decision $C(x)$ is be used to construct an adversarial example. Unlike other attacks, the boundary attack does not start with the original image and modifies it, but instead starts with a (possible random) misclassified image and changes it to resemble the original image.

The boundary attack iteratively generates adversarial examples $x^{(k)}$ (and therefore perturbations $\delta^{(k)} = x^{(k)} - x$) using a random walk. At the beginning it samples $x^{(0)} \sim \mathcal{U}(0,1)^n$ until $C(x^{(0)}) \neq C^*(x)$, i.e. $x^{(0)}$ is an adversarial example for x. The next iteration should then fulfill the following criteria to ensure that the sequence $(x^{(k)})_{k\in\mathbb{N}}$ has a limit point on the decision boundary:

1. The new image $x^{(k+1)}$ is in the range of valid images:
$$x^{(k+1)} \in [0,1]^n.$$
2. The size of the random step is proportional to the size of the perturbation $\delta^{(k)}$ for a parameter $\gamma > 0$:
$$\left\|\delta^{(k+1)} - \delta^{(k)}\right\|_2 = \gamma \cdot \left\|\delta^{(k)}\right\|_2 .$$
3. The distance to the original is reduced with a factor $0 < \nu < 1$:
$$\left\|\delta^{(k+1)}\right\|_2 = \nu \cdot \left\|\delta^{(k)}\right\|_2 .$$

While these conditions are difficult to meet exactly, they are approximated by sampling from a normal distribution orthogonal to $\delta^{(k)}$ and making a step with size γ in this direction. If this point is still adversarial, a step towards x is made with size ν. The parameters γ and ν are adapted dynamically, similarly to trust region methods (Sorensen, 1982). A

moving average of the success rate of the orthogonal of the direct step is kept and the parameters are increased or decreased if this average deviates too much from beforehand chosen optimal value.

The success rate for the orthogonal step should be close to 0.5, the idea being that if $x^{(k)}$ is close to the decision boundary, which is locally linear almost everywhere for most types of neural networks, then approximately 50% of random orthogonal steps should pass the decision boundary. The target success rate for the direct step is set to a value around 0.25, which empirically leads to good adversarial examples. The algorithm terminates once ν is close to zero.

4.3 DeepFool Attack

Moosavi-Dezfooli et al. (2016) developed the untargeted white-box attack *DeepFool*. The authors calculate minimal adversarial perturbations for a linear classifier by projecting the original image to the nearest decision boundary. By approximating the network with its first order Taylor polynomial, these calculations can be applied iteratively to the nonlinear classifier until an adversarial example is found. In detail, the perturbation is initialized as $\delta^{(0)} = 0$. For each class label $i \neq C^*(x)$ the distance to the decision boundary is estimated with

$$l_i = \frac{\left|Z_i(x + \delta^{(k)}) - Z_{C^*(x)}(x + \delta^{(k)})\right|}{\left\|\nabla Z_i(x + \delta^{(k)}) - \nabla Z_{C^*(x)}(x + \delta^{(k)})\right\|_2}, \tag{4}$$

and $\delta^{(k+1)}$ is the projection to the nearest boundary under the distance approximation from above. Our version of the DeepFool attack slightly differs from the original version in insofar that we clip $\delta^{(k)}$ to obtain $x + \delta^{(k)} \in [0, 1]^n$ and that we restrict $\left\|\delta^{(k)}\right\|_2$ at each step, which yields better adversarial examples in our experience.

4.4 Universal Adversarial Perturbations

The concept of universal adversarial perturbations was proposed by Moosavi-Dezfooli et al. (2017) and refers to a single perturbation vector $\delta \in \mathbb{R}^n$ such that $C(x + \delta) \neq C^*(x)$ for many different x sampled from the input image distribution. To do this, we use following variation of their algorithm.

As long as the accuracy on the test set is above a previously chosen threshold, repeat these steps:

1. Initialize $\delta^{(0)} \leftarrow 0$.
2. Sample a batch $X^{(k)} = \{x_1^{(k)}, ..., x_N^{(k)}\}$ of images such that $\forall x \in X^{(k)} : C(x + \delta^{(k)}) = C^*(x)$.
3. For each $x_i^{(k)}$ compute a perturbation $\delta_i^{(k+1)}$ using FGSM (Goodfellow et al., 2014).
4. Update the perturbation: $\delta^{(k+1)} \leftarrow \delta^{(k)} + \frac{1}{N} \sum_{i=0}^{N} \delta_i^{(k+1)}$.

This differs insofar from the original algorithm described by Moosavi-Dezfooli et al. (2017) that in step 3 approximate perturbations for a whole batch is computed, while the computed an optimal perturbation for each x_i using their previous work DeepFool.

In principle many other attacks instead of FGSM can be used, however we found that for our applications FGSM reached adequate results compared to much slower methods. Since FGSM uses the gradient of the network, FGSM and therefore this algorithm for computing universal adversarial perturbations are white-box attacks.

5 Experiments

5.1 Datasets and Network Architectures

We trained models on each of the following benchmark datasets:

- MNIST: 28×28 grayscale images of handwritten digits (60000 images for training, 10000 images for testing) LeCun et al., 1998
- Fashion-MNIST: 28×28 grayscale images of fashion items (60000 images for training, images 10000 for testing) Xiao et al., 2017
- SVHN: 32×32 color images of house numbers (73257 images for training, 26032 images for testing) Netzer et al., 2011
- CIFAR-10: 32×32 color images of animals and vehicles (50000 images for training, 10000 images for testing) Krizhevsky and Hinton, 2009

Each of the four data sets consists of ten classes.

As baseline architectures we trained convolutional networks utilizing max pooling, batch normalization (Ioffe and Szegedy, 2015) and dropout (Srivastava et al., 2014). Training capsule networks can be comparatively more difficult in practice, therefore we had to carefully construct well-suited architectures:

For the MNIST data set we used a three layer CapsNet like that of Sabour et al. (2017), but with only 64 convolutional kernels in the first layer. For the Fashion-MNIST and the SVHN data sets we used CapsNets with two convolutional layers in the beginning, followed by the primary capsule layer and the class capsules.

Simple CapsNets however do not perform very well on more complex data like CIFAR-10 (Xi et al., 2017), therefore we use a modified DCNet (Phaye et al., 2018) with three convolutional capsule layers and a non-of-the-above category (see Section 3) for the dynamic routing.

We train all CapsNets using margin loss and the reconstruction loss for regularization (Sabour et al., 2017), and we use three iterations in the routing algorithm. All CapsNets as well as ConvNets are trained with the Adam optimizer (Kingma and Ba, 2014). For more details on the model architectures, please refer to the tables in Appendix A.

Network	MNIST	Fashion-MNIST	SVHN	CIFAR-10
ConvNet	99.39%	92.90%	92.57%	88.22%
CapsNet	99.40%	92.65%	92.35%	88.21%

Table 1: Test accuracies achieved by our networks.

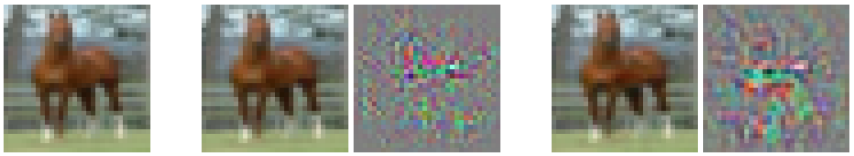
The test accuracies of our networks on the respective datasets are displayed in Table 1. While these accuracies do not reach the state of the art, the similarity between the performances of the CapsNets and ConvNets indicates their suitability for the task of comparing adversarial robustness.

For each of the data sets we compute adversarial examples using our four chosen attacks in the following manner:
We randomly select 1000 images each from the test set and attack them with DeepFool and the boundary attack. For Carlini-Wagner (with hyperparameter $\kappa = 1$ in Equation (3)) we calculate 500 adversarial examples for randomly chosen images from the test set and randomly chosen target labels that are different from the true labels. We make five steps with the binary search for the c value in Equation (2), but we determine a favorable initial value for c by performing some runs with more iterations for each architecture and dataset combination. In the case of the universal perturbation we divide the test set into ten parts and compute ten adversarial perturbations on each part.

We do not restrict the maximal perturbation norm for any of the attacks and hence the Carlini-Wagner, the boundary and the DeepFool attack only generate valid adversarial examples. Regarding the universal perturbations, we terminate the algorithm once accuracy falls below 50%. We found that the universal perturbations generalize well: while only a tenth of the test set is used to generate each universal perturbation, they can reduce the accuracy on the whole test set to $50 \pm 2.5\%$.

In Figure 1 some examples of the product of the attacks on CIFAR-10 can be seen. The Carlini-Wagner, the boundary and the DeepFool attack result in humanly imperceptible adversarial examples. Only the universal perturbations are clearly visible. Furthermore, we can observe some differences in the structure of the adversarial perturbations. While the black-box attack (boundary attack) produces perturbations resembling random noise, some noticeable patterns emerge in the perturbations of the white-box attacks. For more examples of adversarial images for this and the other datasets, see Appendix B.

CW



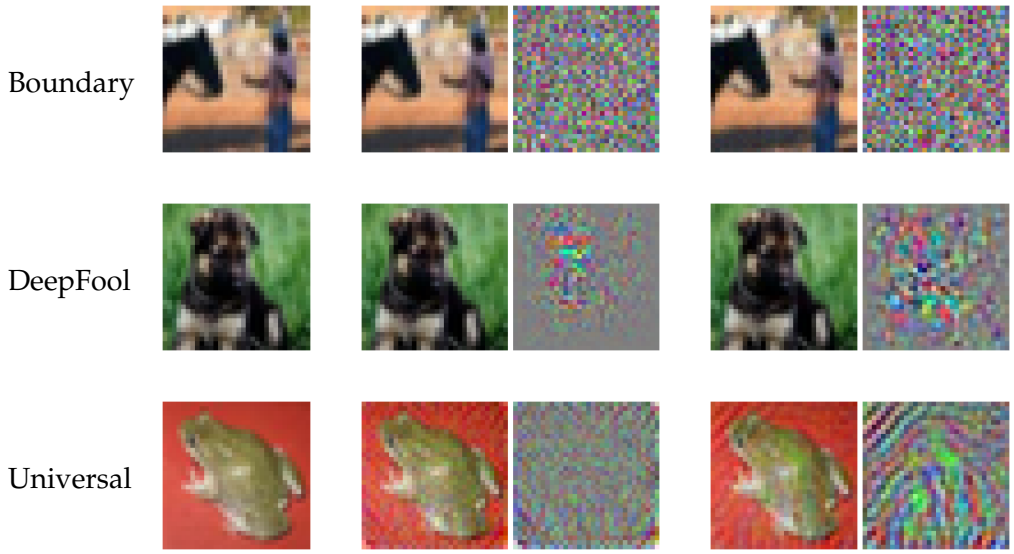


Figure 1: Original images from the CIFAR-10 dataset (left), adversarial examples and perturbations for CapsNet (middle) and adversarial examples and perturbations for ConvNet (right). Pixel values of perturbation images are scaled for visibility.

5.2 Robustness to Adversarial Attacks

The most important aspect with respect to the dangers of adversarial attacks is how much an image has to be modified for an successful attack, i.e. the distance to the original image in some metric. In our case this is the Euclidean norm (see Section 4). These results are displayed in Table 2. On all datasets except MNIST the CapsNets is more susceptible to the Carlini-Wagner attack than the ConvNets.

We observe the same trend for the adversarial examples calculated using the boundary

Attack	Network	MNIST	Fashion	SVHN	CIFAR-10
CW	ConvNet	1.40	0.51	0.67	0.37
	CapsNet	1.82	0.50	0.60	0.23
Boundary	ConvNet	3.07	1.24	2.42	1.38
	CapsNet	3.26	0.93	1.88	0.72
DeepFool	ConvNet	1.07	0.31	0.41	0.23
	CapsNet	2.02	0.55	0.80	0.16
Universal	ConvNet	6.71	2.61	2.46	2.45
	CapsNet	11.45	5.31	8.59	2.70

Table 2: Average perturbation norms for each attack and architecture.

attack. This means that CapsNets are in general neither more robust against white-box attacks nor against black-box attacks than ConvNets, as long as the attacks are sufficiently strong. Against DeepFool however the CapsNets perform slightly better and have significantly higher adversarial perturbation norms for all datasets except CIFAR-10.

The biggest difference between the architectures is the behavior against universal perturbations, where the norms of the perturbations for the CapsNets are considerably higher for the MNIST, Fashion-MNIST and SVHN datasets. Especially the perturbations for SVHN calculated for the CapsNet are not just clearly visible, the original image is in some cases not recognizable anymore (see Figure 10). Only the universal perturbations on CIFAR-10 have similar norms for the CapsNet and the ConvNet.

However, we have to mention that while the existence of adversarial perturbations with small norms shows the lack of robustness, proving resistance to adversarial attacks is vastly more difficult. After all, adversarial perturbations with large norms can just be the result of an inadequate attack.

5.3 Transferability of Adversarial Examples

Papernot et al. (2016) demonstrated the existence of *cross-technique transferability* for a wide variety of machine learning techniques. This means that adversarial examples constructed for a specific classifier may also be able to fool another classifier using a vastly different machine learning model. Such transfer attacks can be seen as black-box attacks where the first model functions as a oracle (or substitute) for the second one.

To measure the transferability we evaluate the fooling rate of the adversarial examples calculated for CapsNets when applied to the ConvNets and vice versa (see Figure 2). We define an image to *fool* a network in the case of the untargeted attacks (boundary, DeepFool, universal) if it is misclassified and in the case of targeted attacks (Carlini-Wagner) if it is classified as the target label for that the adversarial example was computed.

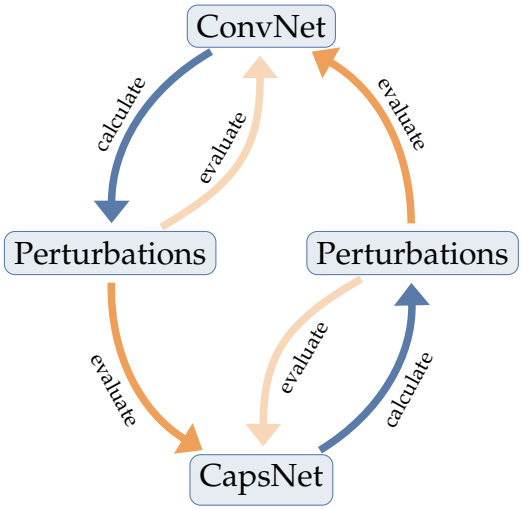


Figure 2: Our evaluation procedure. The light orange arrows show the usual application of adversarial perturbations used in Table 2 while the dark orange arrows show the application used in Table 3.

The transfer fooling rates (Table 3) for the Carlini-Wagner attack are quite small, even though we used a non-zero confidence parameter of $\kappa = 1$. The untargeted boundary and DeepFool attacks have higher fooling rates in general and match the result of Table 2 for most data sets, as in adversarial examples with high norms lead to a higher transfer fooling rate. This is not the case for the boundary attack with MNIST and the DeepFool attack with Fashion-MNIST.

Attack	Network	MNIST	Fashion	SVHN	CIFAR-10
CW	ConvNet	0.8%	1.2%	2.8%	2.4%
	CapsNet	2.0%	2.0%	3.8%	2.0%
Boundary	ConvNet	8.8%	9.5%	10.5%	13.4%
	CapsNet	14.2%	14.6%	12.9%	26.1%
DeepFool	ConvNet	4.3%	8.5%	13.5%	11.8%
	CapsNet	0.9%	10.9%	10.8%	14.1%
Universal	ConvNet	4.9%	20.4%	35.0%	25.9%
	CapsNet	38.2%	25.7%	53.4%	47.2%

Table 3: Fooling rates of adversarial examples calculated for a CapsNet and evaluated on a ConvNet and vice versa. For the universal attack we report the accuracy on the whole test set.

However the difference in fooling rates for the latter case is quite small and it is not surprising that MNIST would produce an outlier, since the adversarial perturbations for this dataset are so large.

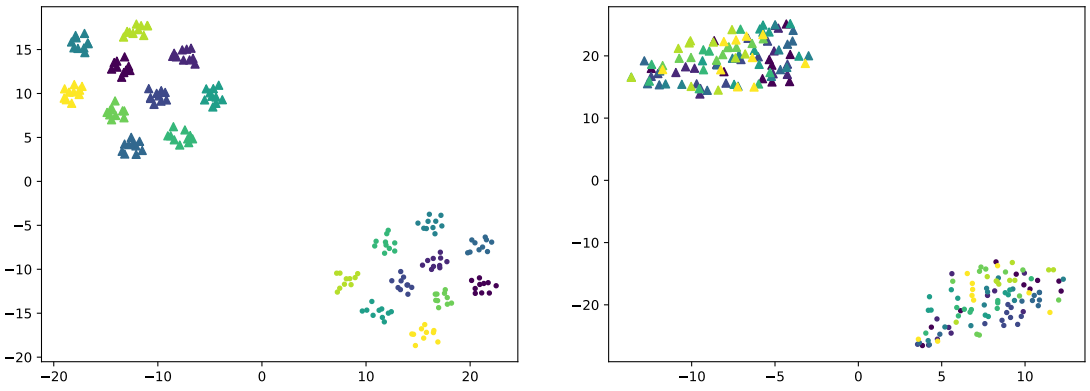
The most noteworthy finding here are the fooling rates for the universal perturbations. Although the norms of the CapsNet universal perturbations were much larger (see Table 2), they perform poorly on the ConvNet, while the small perturbations computed for the ConvNet achieve relatively high fooling rates on the CapsNet. Especially for SVHN the universal perturbations of the ConvNet are much more effective on the CapsNet than the perturbations computed on the CapsNet itself. This shows that CapsNets are even vulnerable to black-box universal perturbations.

5.4 Structural Analysis of Adversarial Perturbations

We have found that CapsNets exhibit comparable behavior to ConvNets for adversarial perturbations with regard to their norms. The low success rate of the transfer attacks and the radically different construction of the networks themselves however suggest that there may be some structural differences between adversarial examples computed for ConvNets and those computed for CapsNets. Understanding these difference could help to find specialized attacks and defenses for the respective architecture.

5.4.1 Visualizing Universal Perturbations

We use t-SNE (Maaten and Hinton, 2008) to visualize the normalized universal perturbations created for the CapsNet and the ConvNet (see Figure 4). We discover that for all datasets there is a clear distinction between the perturbations calculated for the different architectures. Although t-SNE can often distinguish the perturbations of models that are trained slightly differently, this is not just an outcome of this phenomenon. By including universal perturbations calculated for another ConvNet with a different architecture in



(a) CIFAR-10 (b) SVHN

Figure 4: Two dimensional embedding of the universal perturbations calculated using t-SNE (Maaten and Hinton, 2008). The upper left cluster represents perturbations computed on a ConvNet, whereas the cluster in the lower right represents those calculated on a CapsNet. Perturbations with the same color were created using the same subset of test data.

the embedding (see Figure 3), we find the clusters of the two ConvNets still significantly separated from the cluster of the CapsNet. This means, that the universal perturbation of CapsNets and ConvNets are intrinsically different.

In Figure 4 we can furthermore see subclusters based on the dataset splits used to calculate the perturbations, even though all universal perturbations perform very similarly on the whole test set. These distinctions are however less pronounced with the other datasets besides CIFAR-10.

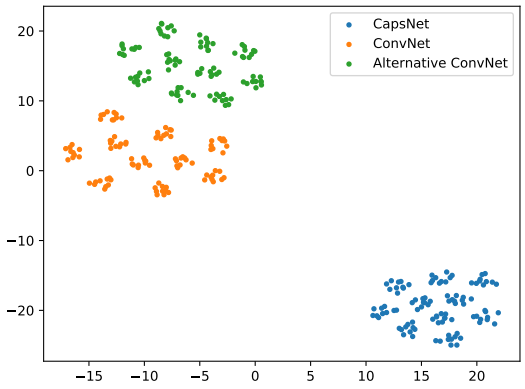
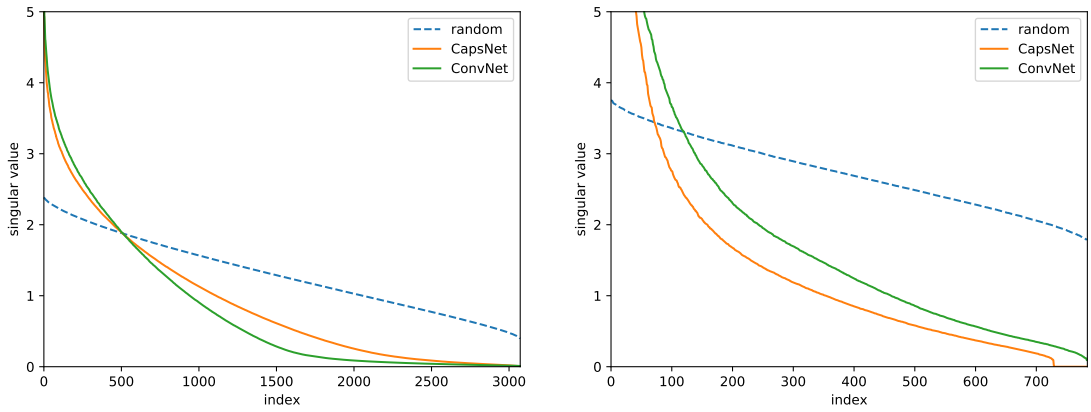


Figure 3: Two dimensional embedding of universal perturbations for CIFAR-10 with additional ConvNet architecture calculated using t-SNE (Maaten and Hinton, 2008).

5.4.2 Singular Values of Adversarial Perturbations

Moosavi-Dezfooli et al. (2017) considered singular values of the matrix containing normalized adversarial examples to determine if adversarial examples lie in a low dimensional subspace.
For this purpose, let us denote with $\delta(x)$ the minimal adversarial perturbation for the input $x \in [0,1]^n$, and define the Matrix N as



(a) CIFAR-10 (b) MNIST

Figure 5: Singular values of the matrix containing normal vectors to the decision boundary.

$$N = \left[\frac{\delta(x_1)}{\|\delta(x_1)\|_2}, ..., \frac{\delta(x_k)}{\|\delta(x_k)\|_2} \right] \in \mathbb{R}^{n \times k}$$

for some $x_1, ..., x_k$ in the test set with $k > n$.
The perturbation vector $\delta(x)$ is orthogonal to the decision boundary, assuming it is reasonably smooth, therefore the singular values of N give us information about the decision boundary. For example, for an binary linear classifier N would have a rank of one, i.e. only one non-zero singular value.

Accurately computing $\delta(x)$ is very challenging, so we use the result of the DeepFool attack as an approximation. We computed 6000 adversarial examples and plot the size of the singular values of the resulting Matrix N together with a matrix containing columns sampled uniformly from the unit sphere S^{n-1} in Figure 5.

For both the CapsNets and the ConvNets the singular values decay much more quickly than those of the random matrix, confirming the findings of Moosavi-Dezfooli et al. (2017). In this regard the difference between CapsNets and ConvNets is inconclusive. Whilst the curve of the CapsNet decreases more slowly than that of the ConvNet in the case of the CIFAR-10 dataset, the opposite occurs for the other cases.

Goodfellow et al. (2014) argue that the linear nature of neural networks is the cause for both their ability to generalize and their vulnerability to adversarial examples. Indeed, many ConvNets model piecewise linear functions: max pooling and the ReLU activation are piecewise linear, while convolutional operations are linear, as are batch normalization and dropout at test time. Due to the dynamic routing, CapsNets are generally not piecewise linear. The linearity of the adversarial perturbations implied by their fast decaying singular values is however evidence that CapsNets also exhibit linear behavior similarly to ConvNets.

5.5 Effects on Activations

To further the efforts in creating architectures robust to adversarial attack, we need to inspect which parts of the networks are responsible for their failure. This is particularly important for the difference between CapsNets and ConvNets. The primary capsule layer seems to need sufficiently complex features as input in order for the network to achieve adequate test performance. For this reason, CapsNets usually start with a series of convolutional layers. If an attack can perturb the leading convolutional layers in a considerable way, the following capsule layers will not be able to rectify the error.

To test this, we examine the difference of the layer's activations in out network when applied to unaltered and adversarial images. As a dimension and scale invariant metric we define

$$d(z, w) = \frac{\|z - w\|_\infty}{\|z\|_\infty} \qquad \text{for original activation } z \text{ and perturbed activation } w. \tag{5}$$

This is related to the layer's Lipschitz constant. Let ϕ be the function modeled by some layer l and z and w the original and perturbed activations of the previous layer $l-1$. We then have a lower bound for the Lipschitz constant K of the layer l with respect to the L^∞-norm:

$$K \geq \frac{d(\phi(z), \phi(w))}{d(z, w)} \cdot \frac{\|\phi(z)\|_\infty}{\|z\|_\infty} \,. \tag{6}$$

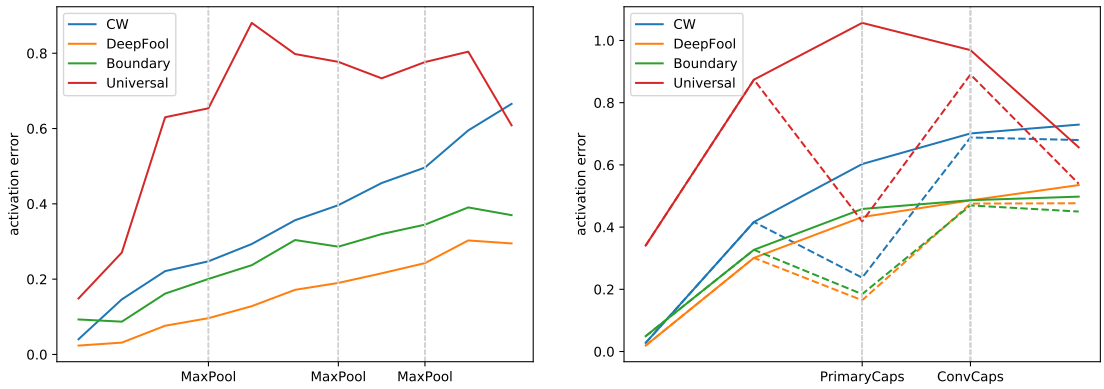
Low Lipschitz constants for every layer imply robustness to L^p based attacks (in case $p \geq 1$). Upper bounds for Lipschitz constants of convolutional and other common layers are known (Szegedy et al., 2013), but have yet to be found for the routing-by-agreement algorithm.

Regarding CapsNets, we can differentiate between the activation of the instantiation parameters and that of the probabilities of presence. Since we are using vector capsules, in our case the instantiation parameters is simply the complete output of a capsule layer and the probability of presence is given by the Euclidean norm of the capsules.

We compute the average activation error $d(z, w)$ for select layers over 128 adversarial examples for our CIFAR-10 networks. Our CapsNet trained on CIFAR-10 begins with eight denseley connected convolutional layers and is therefore an apt choice for this comparison. The results appear in Figure 6.

For the ConvNet the activation errors of the DeepFool, Boundary and Carlini-Wagner attacks are almost monotonically increasing. This means, that not just a single layer is culpable for their deficiency to inhibit adversarial attacks, but instead the error accumulates over all the layers. Curiously, the universal perturbations create substantially higher errors in the internal layers, but not so much in the final layer. This is most likely the case because a universal adversarial example not only has to disturb the activations of few features that are used for a specific image, but of all possible features. We want to mention that a high activation error in the last layer not necessarily corresponds to misclassifications of high confidence. It can merely signify a change in *many* class scores, not just those of the true class and the actually assigned false class.

The activation errors of the instantiation parameters match those of the ConvNet, meaning that even after convolutional layers in the beginning the error accumulates further. While



(a) ConvNet (b) CapsNet. Dashed lines are capsule norms.

Figure 6: Activation errors of select layers. For the CapsNet we show the difference in the whole activation of each layer (instantiation parameters) and the difference of the capsule norms (probability of entity's presence).

the error for the convolutional layers at the start seems large, we want to point out that the first error depicted is the result of eight convolutional layers and that the primary capsule layer has a direct skip connection to the input, but the activation error rises considerably in just the last two capsule layers. We can notice that the error in the capsule norms starts with a dip in the primary capsule layer. This is not automatically meaningful, since the two types of activations measure different aspects. However, in all cases the error of the capsule norms escalates remarkably in the next capsule layer. Together those results show not only the vulnerability of capsule layers themselves in general but also specifically that adversarial attacks are not treated as a capsule activation invariance by the routing-by-agreement algorithm.

6 Conclusion

By applying sufficiently strong attacks to the trained networks we showed that CapsNets can be fooled by imperceptible adversarial examples just as easily as ConvNets. In general, CapsNets and ConvNets have a similar level of vulnerability to adversarial attacks and it is difficult to determine which model is more robust in a specific situation. CapsNets also do not show more resistance against white-box attacks compared to black-box attacks as previously suspected. We could also demonstrate that adversarial examples can be transferred between the two architectures. Specifically, using universal perturbations computed for fooling the ConvNet to attack the CapsNet can lead to a higher fooling rate than the original universal perturbation, even though the transfer attack is in this context a black-box attack.

Moreover we could ascertain that adversarial perturbations of CapsNets exist in a low dimensional subspace, like it was previously known to be the case for ConvNets. Despite the comparable effectiveness of adversarial examples on CapsNets and CapsNets we

discovered underlying structural differences. By embedding the universal perturbation in a two dimensional space a clear distinction became noticeable.

Through analysis of activations between layers we could recognize the roles of capsule layers in the networks with respect to adversarial attacks. The routing-by-agreement algorithm not only fails to amend missteps of the earlier convolutional layer by establishing equivariance in the capsule vector direction and invariance in their norm, but also offers another layer for an adversarial example to intensify their deceiving effect when propagating through the network.

Further work should include the study of additional CapsNet architectures, especially ones utilizing other routing algorithms as well as attempts to help CapsNets detect and defend against adversarial attacks.

References

Wieland Brendel, Jonas Rauber, and Matthias Bethge (2017). "Decision-based Adversarial Attacks: Reliable Attacks against Black-box Machine Learning Models". In: *arXiv preprint arXiv:1712.04248*.

Nicholas Carlini and David Wagner (2017). "Towards Evaluating the Robustness of Neural Networks". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 39–57.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*.

Nicholas Frosst, Sara Sabour, and Geoffrey Hinton (2018). "DARCCC: Detecting Adversaries by Reconstruction from Class Conditional Capsules". In: *arXiv preprint arXiv:1811.06969*.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy (2014). "Explaining and Harnessing Adversarial Examples". In: *arXiv preprint arXiv:1412.6572*.

Jan Philip Göpfert, Heiko Wersing, and Barbara Hammer (2019). "Adversarial attacks hidden in plain sight". In: *arXiv preprint arXiv:1902.09286*.

Geoffrey E Hinton, Zoubin Ghahramani, and Yee Whye Teh (2000). "Learning to parse images". In: *Advances in neural information processing systems*, pp. 463–469.

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

Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst (2018). "Matrix Capsules with EM Routing". In: *International Conference on Learning Representations*.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger (2017). "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708.

Sergey Ioffe and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv preprint arXiv:1502.03167*.

Diederik P Kingma and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization". In: *arXiv preprint arXiv:1412.6980*.

Alex Krizhevsky and Geoffrey Hinton (2009). *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. Citeseer.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio (2016). *Adversarial Examples in the Physical World*. arXiv: 1607.02533 [cs.CV].

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. (1998). "Gradient-based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Huayu Li (2018). *Cognitive Consistency Routing Algorithm of Capsule-network*. arXiv: 1808.09062 [cs.AI].

Laurens van der Maaten and Geoffrey Hinton (2008). "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.Nov, pp. 2579–2605.

Alberto Marchisio, Giorgio Nanfa, Faiq Khalid, Muhammad Abdullah Hanif, Maurizio Martina, and Muhammad Shafique (2019). "CapsAttacks: Robust and Imperceptible Adversarial Attacks on Capsule Networks". In: *CoRR* abs/1901.09878. arXiv: 1901.09878.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard (2016). "Deepfool: A Simple and Accurate Method to Fool Deep Neural Networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard (2017). "Universal Adversarial Perturbations". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1765–1773.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng (2011). "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow (2016). *Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples*. arXiv: 1605.07277 [cs.CR].

David Peer, Sebastian Stabinger, and Antonio Rodriguez-Sanchez (2018). *Increasing the adversarial robustness of capsule networks through scaled distance agreements*. arXiv: 1812.09707 [cs.LG].

Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, and Deepti R. Bathula (2018). "Dense and Diverse Capsule Networks: Making the Capsules Learn Better". In: *CoRR* abs/1805.04001. arXiv: 1805.04001.

Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton (2017). "Dynamic Routing Between Capsules". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., pp. 3856–3866.

Mahmood Sharif, Lujo Bauer, and Michael K. Reiter (2018). "On the Suitability of Lp-norms for Creating and Preventing Adversarial Examples". In: *CoRR* abs/1802.09653. arXiv: 1802.09653.

Danny C Sorensen (1982). "Newton's method with a model trust region modification". In: *SIAM Journal on Numerical Analysis* 19.2, pp. 409–426.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: a Simple Way to Prevent Neural Networks from Overfitting". In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus (2013). "Intriguing properties of neural networks". In: *arXiv preprint arXiv:1312.6199*.

Edgar Xi, Selina Bing, and Yang Jin (2017). "Capsule Network Performance on Complex Data". In: *arXiv preprint arXiv:1712.03480*.

Han Xiao, Kashif Rasul, and Roland Vollgraf (2017). "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: *CoRR* abs/1708.07747. arXiv: 1708.07747.

Appendices

A Detailed Description of Neural Network Architectures

The following tables describe the structure of the neural networks used for the experiments in detail. Each row represents a layer (for brevity's sake some types of layers are merged into a single row). The tables for the CapsNets are separated into two parts: The encoder network (top) and the decoder network (bottom). Some expressions may need further explanation:

- BN: spatial batch normalization (Ioffe and Szegedy, 2015)
- Conv K $F \times F$, {valid,same}: convolutional layer with K filters of shape $F \times F$ and either valid padding (i.e. no padding) or same padding (appropriate padding, such that output height and width is equal to that of the input)
- FC n: fully connected layer with n units
- Densely Connected: a densely connected block (Huang et al., 2017) contains multiple layers in such a way that the input for each layer is the concatenated output of all previous layers in this block.
- (Primary) Conv Caps, dim=d: a convolutional (primary) capsule layer with capsules of dimension d as explained in Section 3. Takes same further arguments as an ordinary convolutional layer.
- Dense Caps, dim=d: a dense capsule layer with capsules of dimension d as explained in Section 3.

Layer	Output Shape
Conv 64 9×9, valid, ReLU, BN	$24 \times 24 \times 64$
Primary Conv Caps, 32 9×9, dim=8, stride=2	$8 \times 8 \times 32 \times 8$
Dense Caps, dim=16	10×16
FC 512, ReLU	512
FC 1024, ReLU	1024
FC 784, sigmoid	784

Table 4: CapsNet architecture for MNIST

Layer	Output Shape
Conv 32 5×5, same, ReLU	$28 \times 28 \times 32$
Max-Pooling 2×2, Dropout 0.15, BN	$14 \times 14 \times 32$
Conv 64 3×3, same, ReLU	$14 \times 14 \times 64$
Max-Pooling 2×2, Dropout 0.15, BN	$7 \times 7 \times 64$
Conv 128 3×3, same, ReLU	$7 \times 7 \times 128$
Dropout 0.15, BN	$7 \times 7 \times 128$
FC 1024, ReLU, Dropout 0.5	1024
FC 10	10

Table 5: ConvNet architecture for MNIST

Layer	Output Shape
Conv 32 3×3, valid, ReLU, BN	$26 \times 26 \times 32$
Conv 32 3×3, valid, Leaky ReLU, BN	$24 \times 24 \times 32$
Primary Conv Caps 16 9×9, dim=8, stride=2	$8 \times 8 \times 16 \times 8$
Dense Caps, dim=16	10×16
FC 512, ReLU	512
FC 1024, ReLU	1024
FC 784, sigmoid	784

Table 6: CapsNet architecture for Fashion-MNIST

Layer	Output Shape
Conv 32 5×5, same, ReLU	$28 \times 28 \times 32$
Max-Pooling 2×2, Dropout 0.15, BN	$14 \times 14 \times 32$
Conv 64 3×3, same, ReLU	$14 \times 14 \times 64$
Max-Pooling 2×2, Dropout 0.15, BN	$7 \times 7 \times 64$
Conv 128 3×3, same, ReLU	$7 \times 7 \times 128$
Dropout 0.15, BN	$7 \times 7 \times 128$
FC 1024, ReLU, Dropout 0.5	1024
FC 10	10

Table 7: ConvNet architecture for Fashion-MNIST

Layer	Output Shape
Conv 64 5×5, valid, ReLU, BN	$28 \times 28 \times 64$
Conv 256 5×5, valid, Leaky ReLU, BN	$24 \times 24 \times 256$
Primary Conv Caps 64 9×9, dim=8 stride=2	$8 \times 8 \times 64 \times 8$
Dense Caps, dim=16	10×16
FC 2048, ReLU	2048
FC 4096, ReLU	4096
FC 3072, sigmoid	3072

Table 8: CapsNet architecture for SVHN

Layer	Output Shape
Conv 32 5×5, same, ReLU	$32 \times 32 \times 32$
Max-Pooling 2×2, Dropout 0.15, BN	$16 \times 16 \times 32$
Conv 64 3×3, same, ReLU	$16 \times 16 \times 64$
Max-Pooling 2×2, Dropout 0.15, BN	$8 \times 8 \times 64$
Conv 128 3×3, same, ReLU	$8 \times 8 \times 128$
Dropout 0.15, BN	$8 \times 8 \times 128$
FC 1024, ReLU, Dropout 0.5	1024
FC 10	10

Table 9: ConvNet architecture for SVHN

Layer	Output Shape
Densely Connected with one Conv 29 3×3, ReLU and seven BN, Conv 32 3×3, ReLU	$32 \times 32 \times 256$
Dropout 0.2, BN	$32 \times 32 \times 256$
Primary Conv Caps 32 5×5, dim=12, stride=2	$14 \times 14 \times 32 \times 12$
Conv Caps, 64 3×3, dim=24, stride=2	$6 \times 6 \times 64 \times 24$
Dense Caps, dim=48	10×48

Densely Connected with two FC 1024, ReLU	2048
FC 2048, ReLU	2048
FC 3072, sigmoid	3072

Table 10: CapsNet architecture for CIFAR-10 (uses none-of-the-above category in dynamic routing between all capsule layers)

Layer	Output Shape
Conv 32 5×5, same, ReLU	$32 \times 32 \times 32$
Conv 32 5×5, same, ReLU	$32 \times 32 \times 32$
Max-Pooling 2×2, Dropout 0.1, BN	$16 \times 16 \times 32$
Conv 64 3×3, same, ReLU	$16 \times 16 \times 64$
Conv 64 3×3, same, ReLU	$16 \times 16 \times 64$
Max-Pooling 2×2, Dropout 0.1, BN	$8 \times 8 \times 64$
Conv 128 3×3, same, ReLU	$8 \times 8 \times 128$
Max-Pooling 2×2, Dropout 0.1, BN	$4 \times 4 \times 128$
FC 1024, ReLU, Dropout 0.5	1024
FC 10	10

Table 11: ConvNet architecture for CIFAR-10

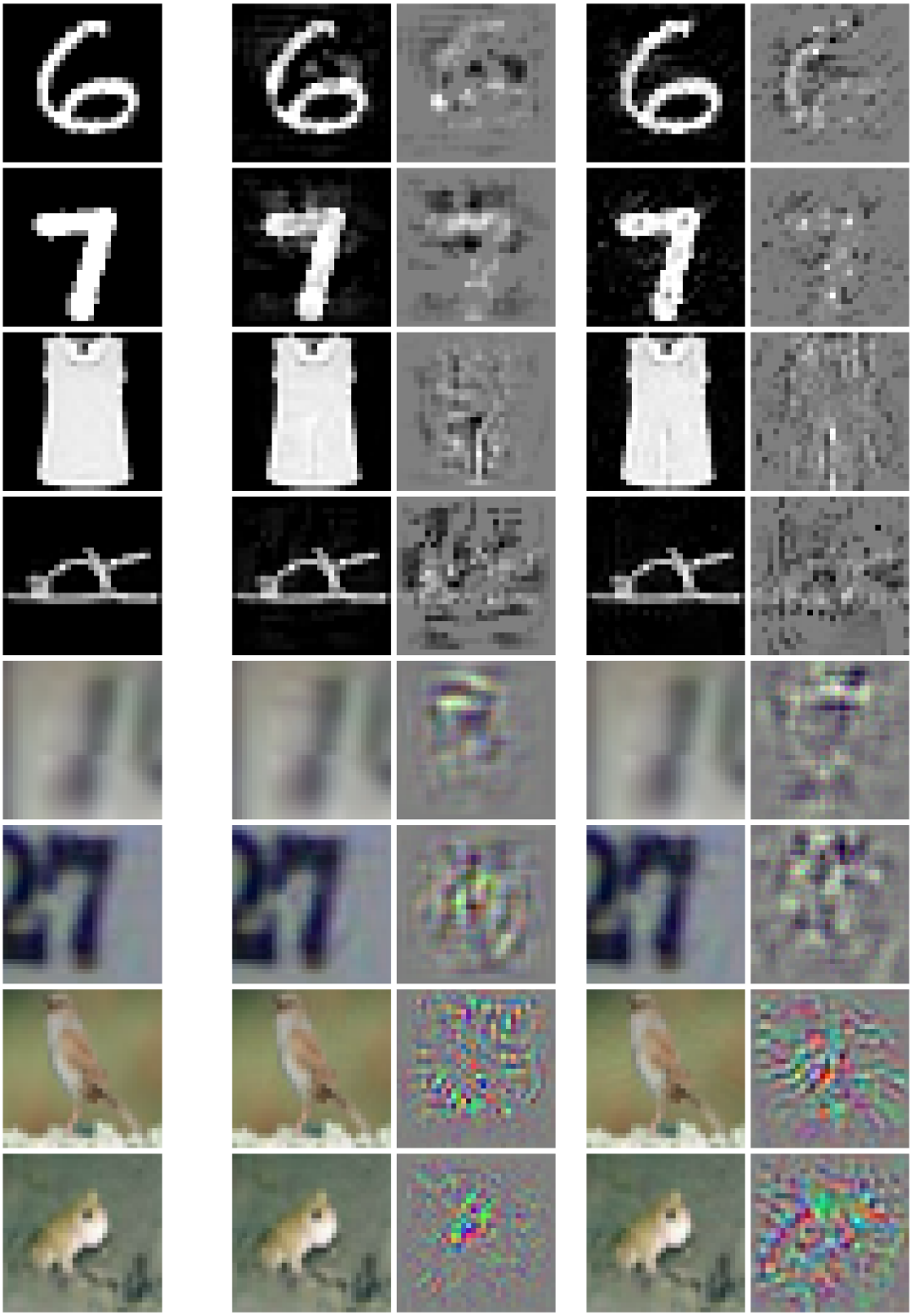
B Adversarial Examples

The following pages display adversarial examples categorized by attack. The pixel values for the perturbations are scaled for visibility in the following way:

$$\delta_{visible} \leftarrow \frac{1}{2}\frac{\delta + 1}{\|\delta\|_\infty} .$$

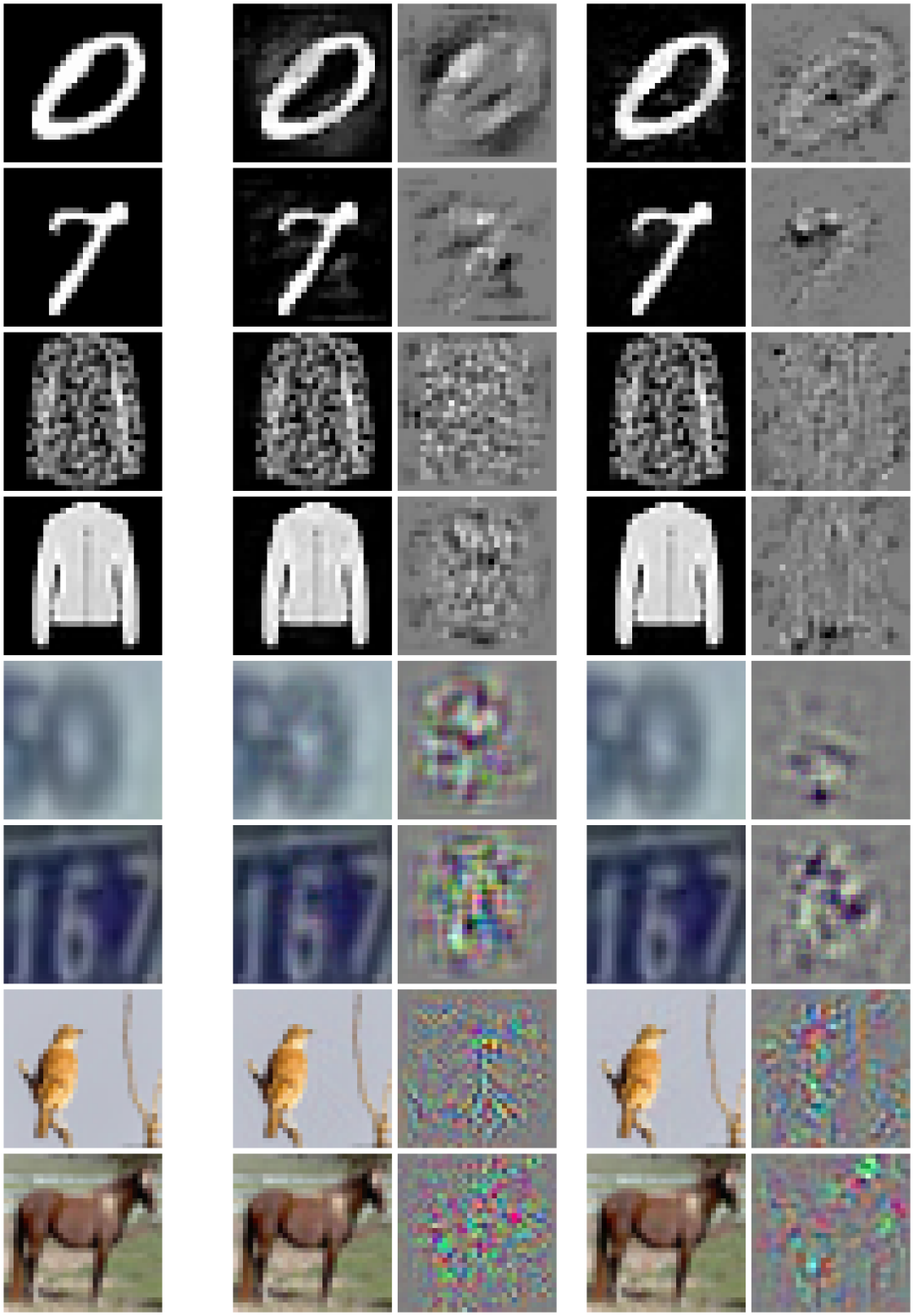
Because of the restriction $x + \delta \in [0, 1]^n$ we have $\delta \in [-1, 1]^n$ and $\delta_{visible} \in [0, 1]^n$.

All images are chosen at random.



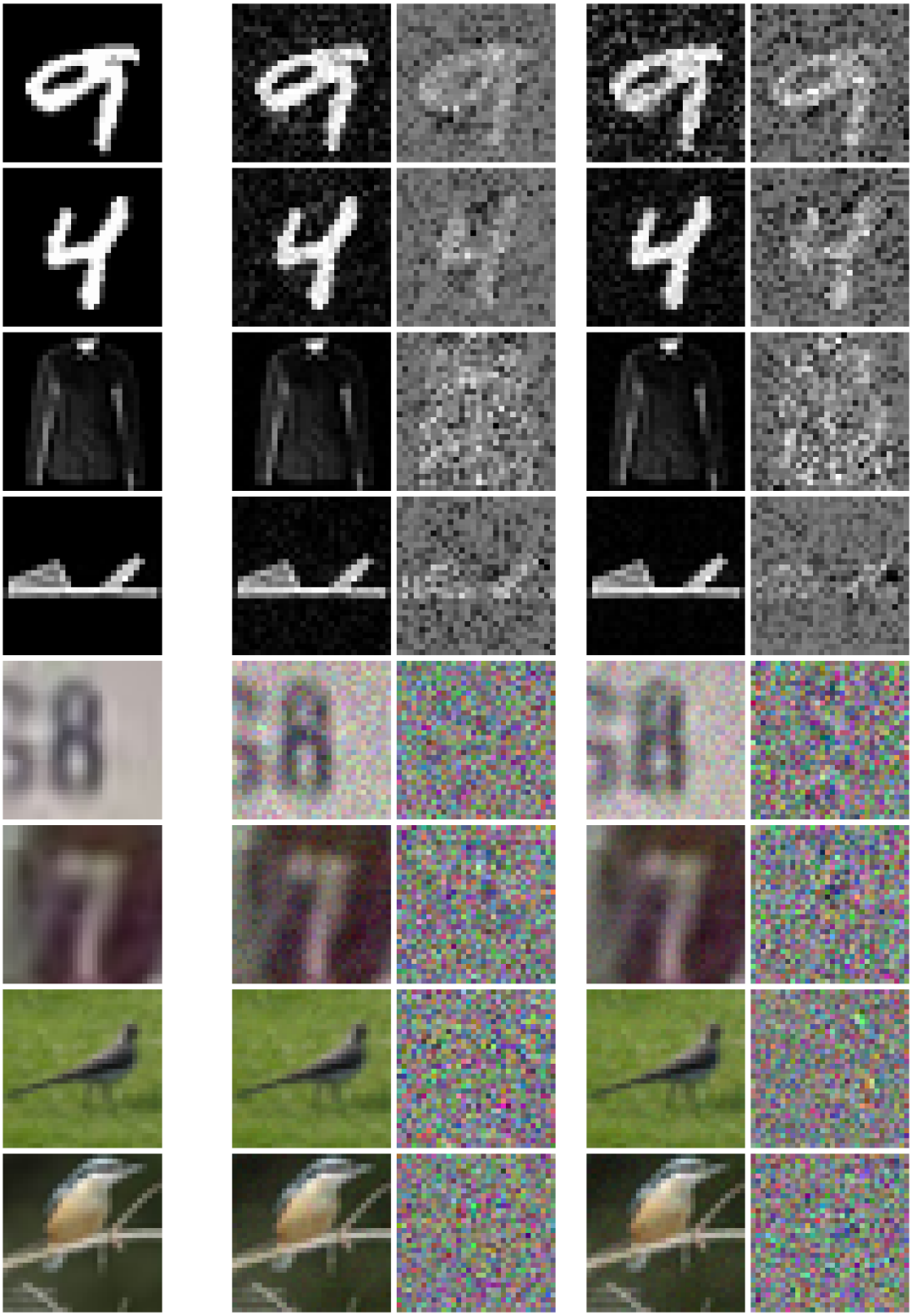
(a) Original (b) CapsNet (c) ConvNet

Figure 7: Carlini-Wagner adversarial examples and perturbations. Pixel values of perturbation images are scaled for visibility.



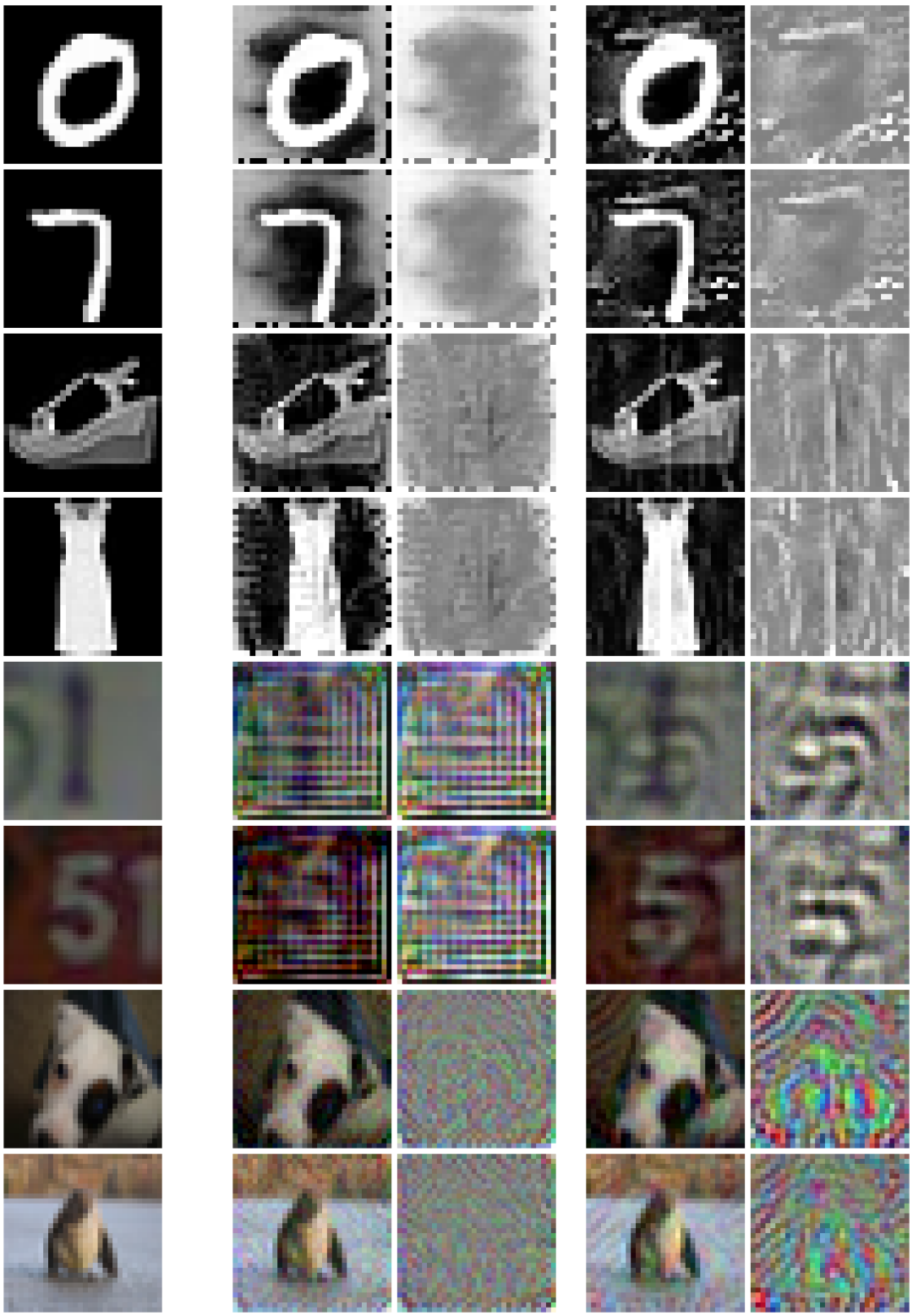
(a) Original (b) CapsNet (c) ConvNet

Figure 8: DeepFool adversarial examples and perturbations. Pixel values of perturbation images are scaled for visibility.



(a) Original (b) CapsNet (c) ConvNet

Figure 9: Boundary attack adversarial examples and perturbations. Pixel values of perturbation images are scaled for visibility.



(a) Original (b) CapsNet (c) ConvNet

Figure 10: Universal adversarial examples and perturbations. Pixel values of perturbation images are scaled for visibility.

C ICML Paper

Our work resulted in a paper that was accepted as a poster presentation at the ICML 2019 Workshop on the Security and Privacy of Machine Learning.

It is included below and accessible at https://arxiv.org/abs/1906.03612.

On the Vulnerability of Capsule Networks to Adversarial Attacks

Felix Michels [*1] **Tobias Uelwer** [*1] **Eric Upschulte** [*1] **Stefan Harmeling** [1]

Abstract

This paper extensively evaluates the vulnerability of capsule networks to different adversarial attacks. Recent work suggests that these architectures are more robust towards adversarial attacks than other neural networks. However, our experiments show that capsule networks can be fooled as easily as convolutional neural networks.

1. Introduction

Adversarial attacks change the input of machine learning models in a way that the model outputs a wrong result. For neural networks these attacks were first introduced by Goodfellow et al. (2014) with alarming results. Recently capsule networks (CapsNets) (Sabour et al., 2017) have been shown to be a reasonable alternative to convolutional neural networks (ConvNets). Frosst et al. (2018) state that CapsNets are more robust against white-box adversarial attacks than other architectures. Adversarial robustness of CapsNets has been previously studied by Marchisio et al. (2019), but with focus on the evaluation of their proposed attack. Detecting adversarial examples using the reconstruction quality of the CapsNets has been investigated by Frosst et al. (2018). Also Peer et al. (2018) have briefly discussed the application of the fast gradient sign method (FGSM) (Goodfellow et al., 2014) on CapsNets. Hinton et al. (2018) report results of the FGSM on CapsNets using EM routing. Another established approach for CapsNets is the dynamic routing algorithm (Sabour et al., 2017).

In this paper, we will focus on these variants of CapsNets and investigate their robustness against common adversarial attacks. In particular, we compare the results of four different attacks on CapsNets trained on different datasets and examine the transferability of adversarial perturbations. We will show that CapsNets are in general not more robust to adversarial attacks than ConvNets.

Our paper is structured as followed: in Sec. 2 we recapitulate the idea of CapsNets and the dynamic routing algorithm. In Sec. 3 we describe the attacks we apply to CapsNets. We summarize our experiments and results in Sec. 4.

2. Capsule Networks and Dynamic Routing

The concept of vector capsules and the dynamic routing algorithm was proposed by Sabour et al. (2017). In an essence, neurons are grouped into vectors, so-called capsules. Each capsule vector is dedicated to a distinct abstract entity, i.e. a single object class in a classification setting. The norm of a capsule vector encodes the probability of the represented object being present in the input, while the vector orientation encodes the object's characteristics. Thus, CapsNets aim to develop dedicated representations that are distributed into multiple vectors in contrast to convolutional networks that utilize an entangled representation in a single vector at a given location. This allows the application of linear transformations directly to the representations of respective entities. Spatial relations, which can be implemented as a matrix product, can thus be modeled more efficiently.

CapsNets are organized in layers. Initially, the original CapsNet applies a convolutional layer. The resulting feature maps are then processed by the primary capsule layer. Internally, it applies a series of convolutional layers on its own, each yielding a spatial grid of capsules. Within all capsule layers the *squashing* function serves as a vector-to-vector non-linearity that squashes each capsule vector length between 0 and 1 while leaving the orientation unaltered. Subsequently, convolutional or densely connected capsule layers can be applied. While the latter does not utilize weight sharing, convolutional capsule layers share the kernels over the spatial grid, as well as capsules from the previous layer. These layers effectively estimate output capsules based on respective input capsules. The dynamic routing algorithm determines each agreement between estimate and iteratively calculated output capsule. This is done by introducing a scalar factor, the so-called routing coefficient, for each connection between an estimate and respective output. Such an output is defined as the sum over all respective estimates, weighted by their routing coeffi-

[*]Equal contribution [1]Department of Computer Science, Heinrich-Heine-Universität Düsseldorf, Germany. Correspondence to: Felix Michels <felix.michels@hhu.de>, Tobias Uelwer <tobias.uelwer@hhu.de>, Eric Upschulte <eric.upschulte@hhu.de>, Stefan Harmeling <harmeling@hhu.de>.

ICML 2019 Workshop on Security and Privacy of Machine Learning, Long Beach, California.

cients. Theoretically, that means information flows where it is needed, both during forward and backpropagation. This non-parametric procedure supports the goal of capsules with clean dedicated representations. To improve results, an additional capsule may be used within the routing algorithm to serve as a dead end for information that may not be linked to known abstract capsule categories. This is also referred to as the *none-of-the-above* category.

3. Adversarial Attacks

Adversarial attacks can be performed in different settings: *white-box* attacks compute the gradient of the networks output with respect to the input, whereas in the *black-box* setting such calculations are not possible. Furthermore, adversarial attacks can be classified into *targeted* attacks, where the goal of the attack is that the network assigns a chosen label to the manipulated image, and *untargeted* attacks, where the attacker's goal is to fool the network in the sense that it missclassifies a given image.

Throughout this paper we denote the input image as $x \in [0,1]^{n\times n}$, the neural network's output logits as $Z(x)$ and the perturbation as δ. If $F(x)$ is the output of the network interpretable as probability, then $F(x) = \mathrm{softmax}(Z(x))$ in the case of the ConvNet and $Z(x) = \mathrm{arctanh}(2F(x)-1)$ in the case of the CapsNet. We refer to the label assigned to x by the networks as $C(x)$ and to the correct label of x by $C^*(x)$. Furthermore, we denote the i-th entry of $Z(x)$ as $Z(x)_i$.

3.1. Carlini-Wagner Attack

The Carlini-Wagner (CW) attack (2017) is a targeted white-box attack and performed by solving the following constrained optimization problem

$$\begin{aligned} \underset{\delta}{\text{minimize}} \quad & ||\delta||_2 + c \cdot \max(G(x,\delta,t) - Z(x)_t, -\kappa) \\ \text{subject to} \quad & x+\delta \in [0,1]^{n\times n}, \end{aligned} \tag{1}$$

where $G(x,\delta,t) := \max_{i\neq t}(Z(x+\delta)_i)$ and $c > 0$. The parameter $\kappa > 0$ controls the confidence. The optimal value for c, i.e. the smallest value, that results in an adversarial example, is found using a binary search. To ensure the box-constraint on $x+\delta$ the authors suggested the following transform of variables

$$\delta = \frac{1}{2}(\tanh(w)+1) - x, \tag{2}$$

where the $\tanh$-function is applied componentwise. After this transformation the optimization problem is treated as unconstrained and can be solved in terms of w using the Adam optimizer (Kingma & Ba, 2014). Carlini and Wagner (2017) also proposed two different approaches to handle the box-constraint: projected gradient descent and clipped gradient descent. For details we refer the reader to the original work (Carlini & Wagner, 2017).

3.2. Boundary Attack

The idea of the boundary attack as introduced by Brendel et al. (2017) is to sample a perturbation which leads to a missclassification of the original image $x^{(0)} := x$. Additionally, the desired perturbation should have the smallest possible norm. The initial perturbation $\delta^{(0)}$ is sampled componentwise from a uniform distribution $\delta_{ij}^{(0)} \sim \mathcal{U}(0,1)$. Initial perturbations, which are not missclassified, are rejected. During the attack adversarial images are constructed iteratively $x^{(k+1)} := x^{(k)} + \delta^{(k)}$ by a random walk close to the decision boundary. During this random walk the following three conditions are enforced by appropriate scaling and clipping of the image and the perturbation:

1. The new image $x^{(k+1)}$ is in the range of a valid image, i.e. in $x^{(k+1)} \in [0,1]^{n\times n}$.
2. The proportion of the size of the perturbation $\delta^{(k)}$ and the distance to the given image is equal to a given parameter γ.
3. The reduction of the distance from the adversarial image to the original image $d(x,x^{(k)}) - d(x,x^{(k+1)})$ is proportional to $d(x,x^{(k)})$ with $\nu > 0$.

The parameters γ and ν are adjusted dynamically, similarly to Trust Region methods.

3.3. DeepFool Attack

DeepFool is an untargeted white-box attack developed by Moosavi-Dezfooli et al. (2016). The authors found that minimal adversarial perturbations for affine multiclass classifiers can be computed exactly and quickly, by calculating the distance to the (linear) decision boundaries and making an orthogonal projection to the nearest boundary. DeepFool initializes $\delta^{(0)} \leftarrow 0$ and then iteratively approximates F with its first degree Taylor polynomial at $x+\delta^{(k)}$, computes a perturbation $\Delta\delta^{(k)}$ for this approximation as described above and updates $\delta^{(k+1)} \leftarrow \delta^{(k)} + \Delta\delta^{(k)}$. For better results, we restrict the norm of $\Delta\delta^{(k)}$ each step k.

3.4. Universal Adversarial Perturbations

A universal perturbation is a single vector $\delta \in \mathbb{R}^{n\times n}$, such that $C(x+\delta) \neq C^*(x)$ for multiple x sampled from the input image distribution. This concept was proposed by Moosavi-Dezfooli et al. (2017) and we use a variation of their algorithm, which we briefly describe in the following. As long as the accuracy on the test set is above a previously chosen threshold, repeat these steps:

Table 1. Average perturbation norms for each attack and architecture.

Attack	Network	MNIST	Fashion	SVHN	CIFAR10
CW	ConvNet	1.40	0.51	0.67	0.37
	CapsNet	1.82	0.50	0.60	0.23
Boundary	ConvNet	3.07	1.24	2.42	1.38
	CapsNet	3.26	0.93	1.88	0.72
DeepFool	ConvNet	1.07	0.31	0.41	0.23
	CapsNet	2.02	0.55	0.80	0.16
Universal	ConvNet	6.71	2.61	2.46	2.45
	CapsNet	11.45	5.31	8.59	2.70

Table 2. Fooling rates of adversarial examples calculated for a CapsNet and evaluated on a ConvNet and vice versa. For the universal attack we report the accuracy on the whole test set.

Attack	Network	MNIST	Fashion	SVHN	CIFAR10
CW	ConvNet	0.8%	1.2%	2.8%	2.4%
	CapsNet	2.0%	2.0%	3.8%	2.0%
Boundary	ConvNet	8.8%	9.5%	10.5%	13.4%
	CapsNet	14.2%	14.6%	12.9%	26.1%
DeepFool	ConvNet	4.3%	8.5%	13.5%	11.8%
	CapsNet	0.9%	10.9%	10.8%	14.1%
Universal	ConvNet	4.9%	20.4%	35.0%	25.9%
	CapsNet	38.2%	25.7%	53.4%	47.2%

1. Initialize $\delta^{(0)} \leftarrow 0$.
2. Sample a batch $X^{(k)} = \{x_1^{(k)}, ..., x_N^{(k)}\}$ of images with $\forall x \in X^{(k)} : C(x + \delta^{(k)}) = C^*(x)$.
3. For each $x_i^{(k)}$ compute a perturbation $\delta_i^{(k+1)}$ using FGSM (Goodfellow et al., 2014).
4. Update the perturbation:

$$\delta^{(k+1)} \leftarrow \delta^{(k)} + \frac{1}{N} \sum_{i=0}^{N} \delta_i^{(k+1)}$$

Since this method depends on the FGSM it is a white-box attack.

4. Experiments

4.1. Datasets and Network Architectures

We train models on each of the following benchmark datasets: MNIST (LeCun et al., 1998), Fashion-MNIST (Xiao et al., 2017), SVHN (Netzer et al., 2011) and CIFAR-10 (Krizhevsky & Hinton, 2009). Each dataset consists of ten different classes. As a baseline architecture we use a ConvNet which we trained on each of the datasets using batch-normalization (Ioffe & Szegedy, 2015) and dropout (Srivastava et al., 2014). Since training CapsNets can be rather difficult in practice, we had to carefully select appropriate architectures:

Like Sabour et al. (2017) we use a three layer CapsNet for the MNIST dataset, where we only used 64 convolutional kernels in the first layer. For the Fashion-MNIST and the SVHN dataset we use two convolutional layers at the beginning (32 and 32 channels with 3×3 filter for Fashion-MNIST, 64 and 256 channels with 5×5 filter for SVHN), followed by a convolutional capsule layer with 8D capsules, 32 filter with size 9×9 and a stride of 2, and finally a capsule layer with one 16D capsule per class. Since CapsNets have problems with more complex data like CIFAR10 (Xi et al., 2017), we use a modified DCNet (Phaye et al., 2018) with three convolutional capsule layers and so-called none-of-the-above category for the dynamic routing for this dataset. We train all CapsNet architectures using the margin loss and the reconstruction loss for regularization (Sabour et al., 2017).

For each dataset we calculate 1000 adversarial examples on images randomly chosen from the test set using the DeepFool attack and the boundary attack. For the Carlini-Wagner attack we calculate 500 adversarial examples again on random samples from the test set (with hyperparameter $\kappa = 1$). The target labels too are chosen at random, but different from the true labels. To evaluate the performance of universal perturbation we split the test set in ten parts and compute ten adversarial perturbations according to the procedure described in Sec. 3.4 on each part.

None of the attacks restrict the norm of the perturbation. This means, the Carlini-Wagner, the boundary and the DeepFool attack generate only valid adversarial examples. In the case of the universal perturbation, we stop once accuracy falls below 50%.

4.2. Results

We are aware of the fact that the test accuracies shown in Tab. 3 of our models are not state-of-the-art. However, we found our models to be suitable for the given task, since the similar performances of ConvNets and CapsNets ensure comparability.

We also compare the average Euclidean norm of the perturbation for each attack, dataset and network. The results are displayed in Tab. 1. Our main result is that applying the Carlini-Wagner attack on the CapsNets yields smaller adversarial perturbations than on the ConvNet. Nevertheless,

Table 3. Test accuracies achieved by our networks.

Network	MNIST	Fashion-MNIST	SVHN	CIFAR10
ConvNet	99.39%	92.90%	92.57%	88.22%
CapsNet	99.40%	92.65%	92.35%	88.21%

for most of the dataset we found that the DeepFool attack performs worse on the CapsNets.

To compare the transferability of adversarial examples we calculate perturbations on the ConvNet and apply those to the CapsNet and vice versa (see Tab. 2). In case of the (targeted) Carlini-Wagner (CW) attack we define a network *fooled* if the perturbed image is classified with the target label. For the Carlini-Wagner attack, the boundary attack and the DeepFool attack our results fit to those displayed in Tab. 1. Especially the perturbations calculated using the universal attack seem to generalize well on the other architecture. For this attack we also found out that the smaller perturbations calculated on the ConvNet can be successfully transferred to CapsNets, while the other way around this approach was less effective, although the norms of the perturbations for the CapsNets are very large.

The adversarial examples for the CapsNets calculated with the Carlini-Wagner, the boundary and the DeepFool attack are not visible for the human eye. Only the universal perturbations are observable (see Fig. 1).

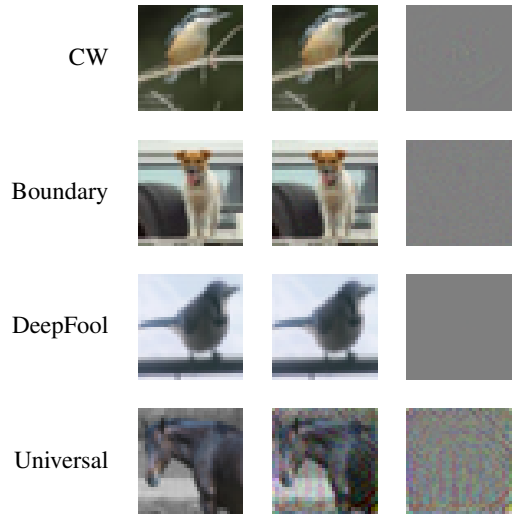


Figure 1. Original images from the CIFAR10 dataset (left), adversarial images (middle) and the corresponding perturbation (right) calculated for a CapsNet.

4.3. Visualizing Universal Perturbations

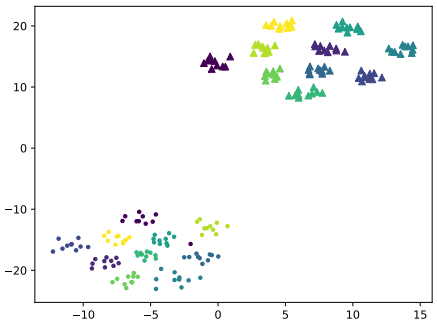


Figure 2. Two dimensional embedding of the universal perturbations calculated using t-SNE (Maaten & Hinton, 2008). The upper right cluster represents perturbations calculated on a ConvNet, whereas the lower left cluster represents those calculated on a CapsNet. Perturbations with the same color were created using the same subset of test data.

We also visualized the universal perturbations calculated for the CapsNet and for the ConvNet using t-SNE (Maaten & Hinton, 2008) and we observe that the perturbations for the CapsNet seem to be inherently different than the perturbations for the ConvNets (see Fig. 2).

5. Conclusion

Our experiments show that CapsNets are not in general more robust to white-box attacks. With sufficiently sophisticated attacks CapsNets can be fooled as easily as ConvNets. Our experiments also show that the vulnerability of CapsNets and ConvNets is similar and it is hard to decide which model is more prone to adversarial attacks than the other. Moreover, we showed that adversarial examples can be transferred between the two architectures.

To fully understand the possibly distinguishable roles of the convolutional and capsule layers with respect to adversarial attacks, we are currently examining the effects of attacks on the activation level of single neurons. However, this analysis is not finished yet and beyond the scope of this paper.

References

Brendel, W., Rauber, J., and Bethge, M. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

Carlini, N. and Wagner, D. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 39–57. IEEE, 2017.

Frosst, N., Sabour, S., and Hinton, G. DARCCC: Detecting adversaries by reconstruction from class conditional capsules. *arXiv preprint arXiv:1811.06969*, 2018.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Hinton, G. E., Sabour, S., and Frosst, N. Matrix capsules with em routing. *International Conference on Learning Representations*, 2018.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Maaten, L. v. d. and Hinton, G. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov): 2579–2605, 2008.

Marchisio, A., Nanfa, G., Khalid, F., Hanif, M. A., Martina, M., and Shafique, M. CapsAttacks: Robust and imperceptible adversarial attacks on capsule networks. *CoRR*, abs/1901.09878, 2019. URL http://arxiv.org/abs/1901.09878.

Moosavi-Dezfooli, S.-M., Fawzi, A., and Frossard, P. Deepfool: A simple and accurate method to fool deep neural Networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582, 2016.

Moosavi-Dezfooli, S.-M., Fawzi, A., Fawzi, O., and Frossard, P. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1765–1773, 2017.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature Learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

Peer, D., Stabinger, S., and Rodríguez-Sánchez, A. J. Training deep capsule networks. *CoRR*, abs/1812.09707, 2018. URL http://arxiv.org/abs/1812.09707.

Phaye, S. S. R., Sikka, A., Dhall, A., and Bathula, D. R. Dense and diverse capsule networks: Making the capsules learn better. *CoRR*, abs/1805.04001, 2018. URL http://arxiv.org/abs/1805.04001.

Sabour, S., Frosst, N., and Hinton, G. E. Dynamic routing between capsules. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 3856–3866. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Xi, E., Bing, S., and Jin, Y. Capsule network performance on complex data. *arXiv preprint arXiv:1712.03480*, 2017.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

D Source Code

All of our source code used for this thesis is available at https://github.com/felixmichels/Adversarial-Attacks-on-CapsNets

We use Python 3.6 and Tensorflow 1.8. Below are excerpts from our implementation of the adversarial attacks.

D.1 Carlini-Wagner Attack

```
import tensorflow as tf


class CWAttackOp():
    def __init__(self,
                 model_class,
                 model_params,
                 shape,
                 num_classes,
                 kappa,
                 rand_start_std=0.0):
        """
        Builds a optimize op for the carlini wagner attack.

        Args:
            model_class: The class of the model to construct.
                         Expects subclass of BasicModel
            model_params: Additional parameter for the model init
            shape: Shape of the images
            num_classes: Number of classes
            kappa: Confidence parameter
            rand_start_std: std if we want to start from a
                            randomly disturbed image
        """

        self.original = tf.placeholder(dtype=tf.float32, shape=shape)
        self.target = tf.placeholder(dtype=tf.int64, shape=())
        self.lagrangian = tf.placeholder(dtype=tf.float32, shape=())

        initial_im = tf.clip_by_value(
            self.original + tf.random_normal(self.original.get_shape(),
                                             mean=0.0, stddev=rand_start_std),
            clip_value_min=0.001,
            clip_value_max=0.999)
        w = tf.get_variable('w', trainable=True,
                            initializer=tf.atanh(2*initial_im - 1))

        image = (0.5 + 1e-8) * (tf.tanh(w) + 1)

        self.model = model_class(
            tf.expand_dims(image, axis=0),
            tf.expand_dims(self.target, axis=0),
            num_classes=num_classes,
            trainable=False,
            **model_params)
```

```
46
47          logits = tf.squeeze(self.model.logits)
48          mask = tf.logical_not(
49                  tf.cast(tf.one_hot(self.target, num_classes), tf.bool))
50
51          target_logits = logits[self.target]
52          others_logits = tf.reduce_max(tf.boolean_mask(logits, mask))
53
54          # add kappa, so that adv_loss >= 0
55          adv_loss = kappa + tf.maximum(others_logits - target_logits, -kappa)
56          pert_norm = tf.norm(image-self.original)
57          loss = pert_norm + self.lagrangian*adv_loss
58          opt = tf.train.AdamOptimizer()
59          opt_op = opt.minimize(
60              loss,
61              var_list=[w],
62              global_step=tf.train.get_or_create_global_step())
63
64          init = tf.variables_initializer(
65                  opt.variables() + [w, tf.train.get_global_step()])
66          target_reached = self.model.accuracy > 0.5
67
68          self.loss = loss
69          self.target_reached = target_reached
70          self.optimizer = opt_op
71          self.init = init
72          self.image = image
73
74
75  def cw_attack(sess, orig, target, attack, max_opt_iter, max_bin_iter, c_prop):
76      """
77      Carlini wagner attack for a single image
78
79      Args:
80          sess: The tensorflow session to use
81          orig: The original image (as an numpy array) to attack
82          target: The target label
83          attack: An instance of CWAttackOp
84          max_opt_iter: Maximal number of steps for the adam optimizer
85          max_bin_iter: Maximal number of steps for the binary search for c
86          c_prop: A property for the initial c value.
87                  Is set to the value returned by binary search
88
89      Returns:
90          An numpy array containing the adversarial example,
91          or None if the attack was unsuccessful
92      """
93
94      def test_func(c):
95          tf.logging.info('Testing adv img with c=%f', c)
96          sess.run(attack.init, feed_dict={attack.original: orig})
97          loss_prev = 1e6
98          for it in range(max_opt_iter):
99              _, loss = sess.run(
100                 [attack.optimizer, attack.loss],
101                 feed_dict = {
102                     attack.original: orig,
```

```
                    attack.target: target,
                    attack.lagrangian: c,
                })
            if it % (max_opt_iter // 10) == 0:
                tf.logging.debug('Loss: %f', loss)
                # Check if improvement is made
                if loss > 0.999 * loss_prev:
                    tf.logging.debug('Stopped early')
                    break
                loss_prev = loss

        return sess.run(attack.target_reached,
                        feed_dict = {attack.target: target})

    tf.logging.debug('Starting binary search')
    c = _binary_search_min(test_func, init_c=c_prop.fget(),
                           max_iter=max_bin_iter)
    if c is None:
        return None

    c_prop.fset(c)

    sess.run(attack.init, feed_dict={attack.original: orig})
    good_adv = None
    tf.logging.info('Final run with c=%f', c)
    for it in range(max_opt_iter):
        _ = sess.run(
            attack.optimizer,
            feed_dict={
                attack.original: orig,
                attack.target: target,
                attack.lagrangian: c,
            })
        if it % (max_opt_iter // 10) == 0 or it == max_opt_iter-1:
            success, adv = sess.run([attack.target_reached, attack.image],
                                    feed_dict={attack.target: target})
            if success:
                good_adv = adv
            elif good_adv is not None:
                tf.logging.debug('Early stopping in final run')
                break
    return good_adv


def _binary_search_min(func, init_c, max_iter=10):
    """Finds minimal c, s.t. func(c) is True
    Assumes, that func is monotone and func(0) is False
    Returns None, if no valid c could be found"""
    c_min = 0.0
    c_max = None
    c = init_c

    for _ in range(max_iter):
        if func(c):
            c_max = c
        else:
            c_min = c
```

```
160
161         if c_max is None:
162             c *= 2
163         else:
164             c = (c_max + c_min) / 2
165
166     # Final test
167     if not func(c):
168         c = c_max
169
170     # Return slightly larger c for safety
171     if c_max is not None:
172         c += 1e-3*(c_max - c_min)
173         return c + 1e-3*(c_max - c_min)
```

D.2 Boundary Attack

```
1  import numpy as np
2  import numpy.linalg as la
3
4
5  def boundary_attack(original, is_adv, eps_min, max_steps, dtype=np.float32):
6      """
7      Computes an adversarial example using the boundary attack
8      Args:
9          original: original image as numpy array
10         is_adv: objective function. is_adv(x) == True iff x is adversarial
11         eps_min: tolerance for termination of algorithm
12         max_steps: maximum number of iterations
13         dtype: dtype of adversarial example
14
15     Returns: valid adversarial example as numpy array
16
17     """
18     epsilon = 0.5
19     gamma = 1
20     min_orth = 0.4
21     max_orth = 0.6
22     min_perp = 0.17
23     max_perp = 0.32
24     mov_avg_rate = 1/10
25
26     orth_rate = (max_orth - min_orth) / 2
27     perp_rate = (max_perp - min_perp) / 2
28
29     shape = original.shape
30     perp_success = False
31     orth_success = False
32     pert_norm_old = 1
33
34     adv = np.random.uniform(size=shape)
35     while not is_adv(adv):
36         adv = np.random.uniform(size=shape)
37
38     steps = 0
39     while epsilon > eps_min and steps < max_steps and pert_norm_old > eps_min:
40         # Orthogonal step
```

```
41          pert_norm_old = la.norm(original-adv)
42          eta = np.random.normal(size=shape)
43          eta *= gamma * pert_norm_old / la.norm(eta)
44
45          pert = adv + eta - original
46          pert *= pert_norm_old / la.norm(pert)
47          adv_orth = np.clip(original+pert, 0, 1)
48
49          orth_success = is_adv(adv_orth)
50          steps += 1
51          if orth_success:
52              # Perpendicular
53              adv_perp = np.clip((1-epsilon)*adv_orth + epsilon*original, 0, 1)
54              perp_success = is_adv(adv_perp)
55              steps += 1
56              if perp_success:
57                  adv = adv_perp
58              else:
59                  adv = adv_orth
60
61          orth_rate = mov_avg_rate*orth_success + (1-mov_avg_rate)*orth_rate
62          perp_rate = mov_avg_rate*perp_success + (1-mov_avg_rate)*perp_rate
63
64          # Update epsilon, gamma
65          if orth_rate < min_orth:
66              gamma *= 0.7
67          if orth_rate > max_orth:
68              gamma *= 1.3
69          if perp_rate < min_perp:
70              epsilon *= 0.7
71          if perp_rate > max_perp:
72              epsilon = 0.3 + 0.7*epsilon
73
74      return adv.astype(dtype)
```

D.3 DeepFool Attack

```
1   import tensorflow as tf
2   import numpy as np
3   import numpy.linalg as la
4
5   try:
6       from tensorflow.python.ops.parallel_for.gradients import jacobian
7       tf.app.flags.FLAGS.op_conversion_fallback_to_while_loop = True
8   except ImportError:
9       tf.logging.info('Using own jacobian function (slow)')
10      def jacobian(y,x):
11          grads = []
12          for k in range(y.shape.as_list()[0]):
13              grads.append(tf.gradients(y[k], x)[0])
14          return tf.stack(grads, axis=0)
15
16
17  class DeepfoolOp():
18      def __init__(self,
19                   model_class,
20                   dataset,
```

```
21                  params):
22          """
23          Creates necessary ops for deepfool in the tensorflow graph
24          Args:
25              model_class: The class of the model to construct.
26                           Expects subclass of BasicModel
27              dataset: The dataset to use.
28                       Only necessary for shape and number of classes
29              params: Additional parameters to pass to the model init
30          """
31          self.image = tf.placeholder(dtype=tf.float32, shape=dataset.shape)
32
33          self.model = model_class(
34              tf.expand_dims(self.image, axis=0),
35              trainable=False,
36              num_classes=dataset.num_classes,
37              **params)
38
39          self.logits = self.model.logits[0]
40          self.num_classes = self.logits.shape.as_list()[0]
41          self.logits_grad = jacobian(self.logits, self.image)
42
43      def attack(self,
44                 sess,
45                 original,
46                 label,
47                 max_iter,
48                 step_size):
49          """
50          Deepfool attack
51          Args:
52              sess: Tensorflow session
53              original: original image to attack
54              label: true label of original
55              max_iter: maximum number of iterations
56              step_size: Limit each step to this value
57
58
59          Returns: Returns a valid adversarial example,
60                   or None if max_iter was reached
61          """
62          adv = original.copy()
63          dtype = np.float32
64          # For numerical stability
65          eps = 1e-5
66
67          ks = [k for k in range(self.num_classes) if k != label]
68          w = np.empty(shape=(self.num_classes,)+original.shape, dtype=dtype)
69          f = np.empty(shape=self.num_classes, dtype=dtype)
70          l = np.empty(shape=self.num_classes, dtype=dtype)
71          l[label] = np.inf
72
73          for it in range(max_iter):
74
75              logits, logit_grads = sess.run(
76                      [self.logits, self.logits_grad],
77                      feed_dict={self.image: adv})
```

```
78
79              for k in ks:
80                  w[k] = logit_grads[k] - logit_grads[label]
81                  f[k] = logits[k] - logits[label]
82                  l[k] = np.abs(f[k]) / la.norm(w[k])
83
84              if (f[ks] > eps).any():
85                  return adv
86
87              l_hat = np.argmin(l)
88
89              pert = ((np.abs(f[l_hat]) + eps) / la.norm(w[l_hat])**2) * w[l_hat]
90              pert *= min(1.0, step_size/(la.norm(pert)+1e-8))
91              adv = np.clip(adv+pert, 0, 1)
```

D.4 Universal Adversarial Perturbation

```
1  import numpy as np
2  import tensorflow as tf
3  import numpy.linalg as la
4
5
6  def _clip(img, norm):
7      np.clip(img, -1, 1, out=img)
8      if norm is not None and la.norm(img) >= norm:
9          img *= norm / la.norm(img)
10
11
12 class UniversalPerturbation():
13     """
14     Class for computing a universal perturbation for a specific
15     image data subset. The resulting perturbation after running the
16     fit method will be in the perturbation attribute
17     """
18
19     def __init__(self,
20                  attack,
21                  img, label,
22                  batch_size,
23                  max_it,
24                  max_norm=None,
25                  target_rate=None):
26         """
27
28         Args:
29             attack: FastAttack instance,
30                     i.e. a class than can compute adversarial examples
31                     for a whole batch of images on a specific model.
32                     Usually FGSM is used
33             img: numpy array of originals
34             label: original labels
35             batch_size: used in FastAttack
36             max_it: Maximal number of iterations
37             max_norm: Maximal l2-norm of perturbation,
38                       or None for no restriction
39             target_rate: Stop, if this fooling rate is reached
40         """
```

```
41
42          self.target_rate = target_rate
43          self.attack = attack
44          self.img = img
45          self.label = label
46          self.batch_size = batch_size
47          self.max_it = max_it
48          self.max_norm = max_norm
49
50          self.perturbation = np.zeros_like(img[0])
51          self._best_fool_rate = 1.0
52          self._work_pert = None
53
54      def _feed(self, idx):
55          img = np.clip(self.img[idx] + self._work_pert, 0, 1)
56          return {self.attack.model.img: img,
57                  self.attack.model.label: self.label[idx]}
58
59      def _not_fooled(self, sess):
60          preds = []
61          N = len(self.label)
62          for idx in np.array_split(np.arange(N), N//self.batch_size):
63              preds.append(sess.run(
64                      self.attack.model.prediction,
65                      feed_dict=self._feed(idx)))
66
67          preds = np.concatenate(preds)
68          return preds == self.label
69
70      def fit(self, sess):
71          """
72          Compute a universal perturbation for the given images/labels
73          Args:
74              sess: Tensorflow session
75          """
76          self._work_pert = np.random.normal(size=self.perturbation.shape)\
77                            .astype(np.float32)
78          if self.max_norm is None:
79              init_norm = np.mean([la.norm(img) for img in self.img]) / 100
80          else:
81              init_norm = self.max_norm / 10
82
83          _clip(self._work_pert, init_norm)
84
85          correct = self._not_fooled(sess)
86          for it in range(self.max_it):
87
88              idx = np.random.choice(np.where(correct)[0], self.batch_size)
89              pert = sess.run(self.attack.perturbation,
90                              feed_dict=self._feed(idx))
91              pert = pert.mean(axis=0)
92              self._work_pert += pert
93
94              _clip(self._work_pert, self.max_norm)
95
96              correct = self._not_fooled(sess)
97              acc = np.mean(correct)
```

```
98              if acc < self._best_fool_rate:
99                  tf.logging.debug('Found good perturbation')
100                 self._best_fool_rate = acc
101                 self.perturbation = self._work_pert
102
103             if self.target_rate is not None and \
104             self._best_fool_rate < self.target_rate:
105                 break
106
107             tf.logging.debug('it: %d, acc: %1.3f, norm: %1.3f',
108                              it, acc, la.norm(self._work_pert))
109
110         tf.logging.info('Reached acc %1.3f with norm %2.3f',
111                         self._best_fool_rate, la.norm(self.perturbation))
```

```
1  class FGSM(FastAttack):
2
3      def __init__(self, model, epsilon):
4          self.epsilon = epsilon
5          super(self.__class__, self).__init__(model)
6
7      def _unclipped_pert(self):
8          grad = tf.gradients(self.model.loss, self.model.img)[0]
9          return self.epsilon*tf.sign(grad)
```

List of Figures

List of Tables