

# Adversarial Robustness in Deep Learning

Concepts & Examples

Slide Deck and Tutorials

**[bit.ly/adv\\_learn](https://bit.ly/adv_learn)**

# Session Agenda



Introduction



Deep Learning Essentials



Adversarial Attacks -  
Examples



Adversarial Learning -  
Techniques

# Introduction



# Who are we?



**Dipanjan (DJ) Sarkar**

Data Science Lead, Applied Materials, Author,  
Google Dev. Expert - Machine Learning



**Sayak Paul**

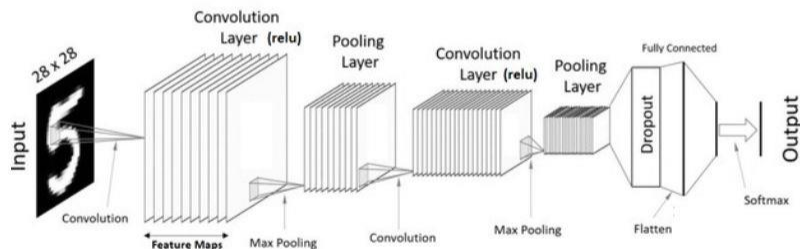
Deep Learning Associate, PyImageSearch,  
Author, Google Dev. Expert - Machine Learning





# Deep Learning Essentials

# CNN Architecture

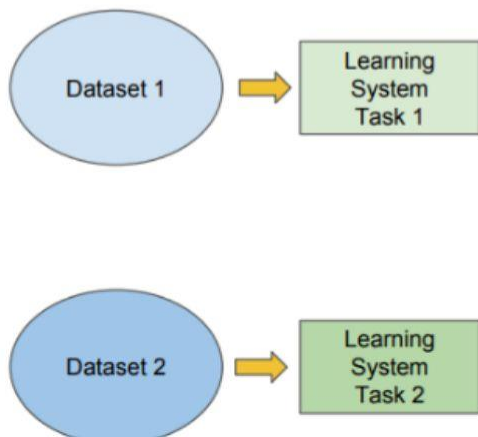


- **CNNs have a stacked layered architecture of several convolution and pooling layers**
- **Convolution layer**
  - Consists of several filters or kernels
  - Passed over the entire image in patches and computes a dot product
  - Result is summed up into one number per operation (dot product)
- **Pooling layer**
  - Downsamples feature maps from conv layers
  - Typically max-pooling is used which selects the max-pixel value out of a patch of pixels
- **Activation layer**
  - Feature maps \ pooled outputs are sent through non-linear activations
  - Introduces non-linearity and helps train via. backpropagation

# Traditional ML vs. Transfer Learning

## Traditional ML

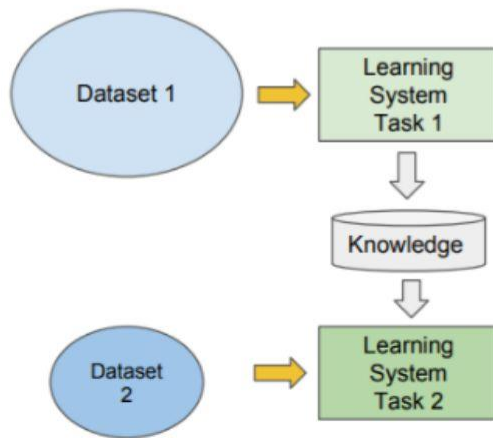
- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



vs

## Transfer Learning

- Learning of a new task relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data





# The power of Transfer Learning

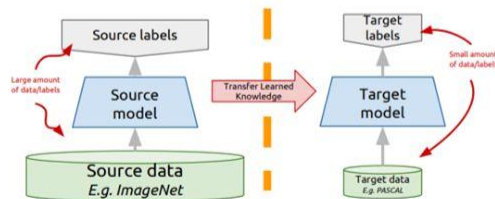
## Transfer learning: idea

Instead of training a deep network from scratch for your task:

- Take a network trained on a different domain for a different **source task**
- Adapt it for your domain and your **target task**

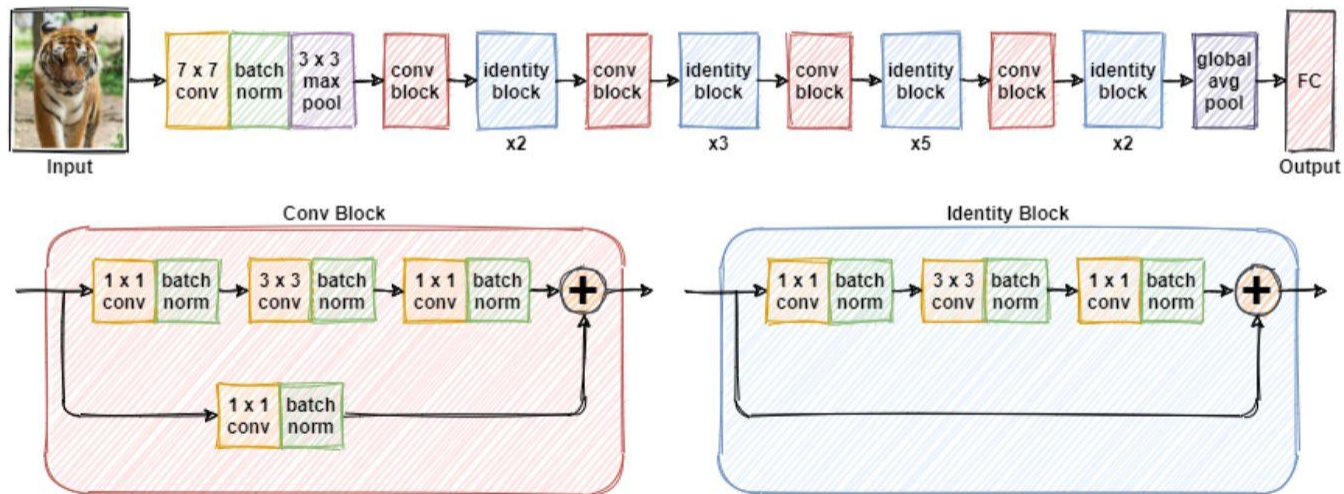
Variations:

- Same domain, different task
- Different domain, same task

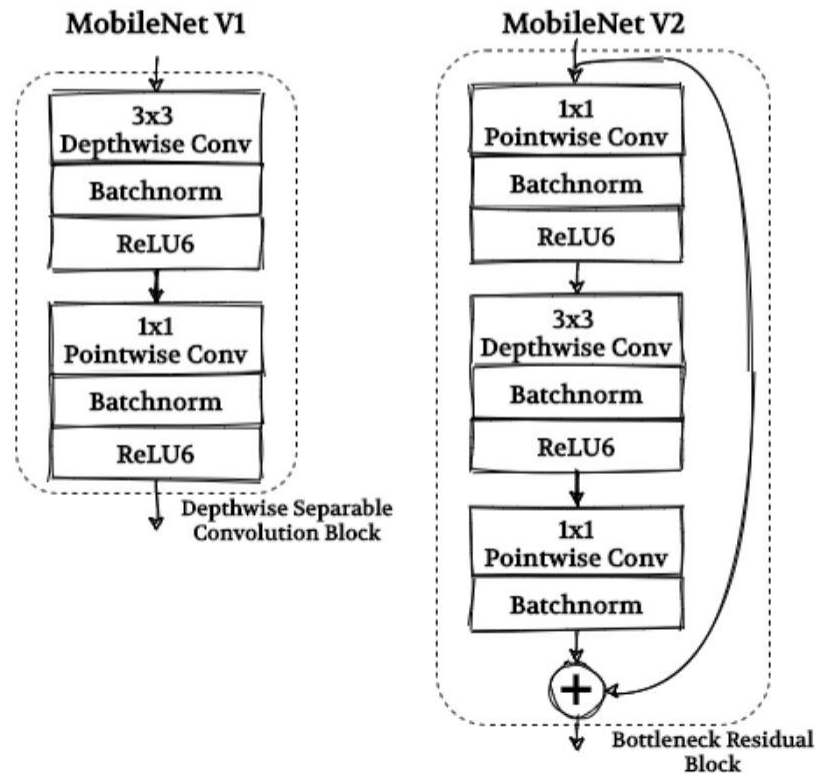


- Leverage a pre-trained deep learning model (which was trained on a large dataset — like ImageNet)
- Adapt the model by applying and transferring its knowledge in the context of our problem
- Two approaches
  - Frozen pre-trained model as a feature extractor
  - Fine-tuning pre-trained model on our data

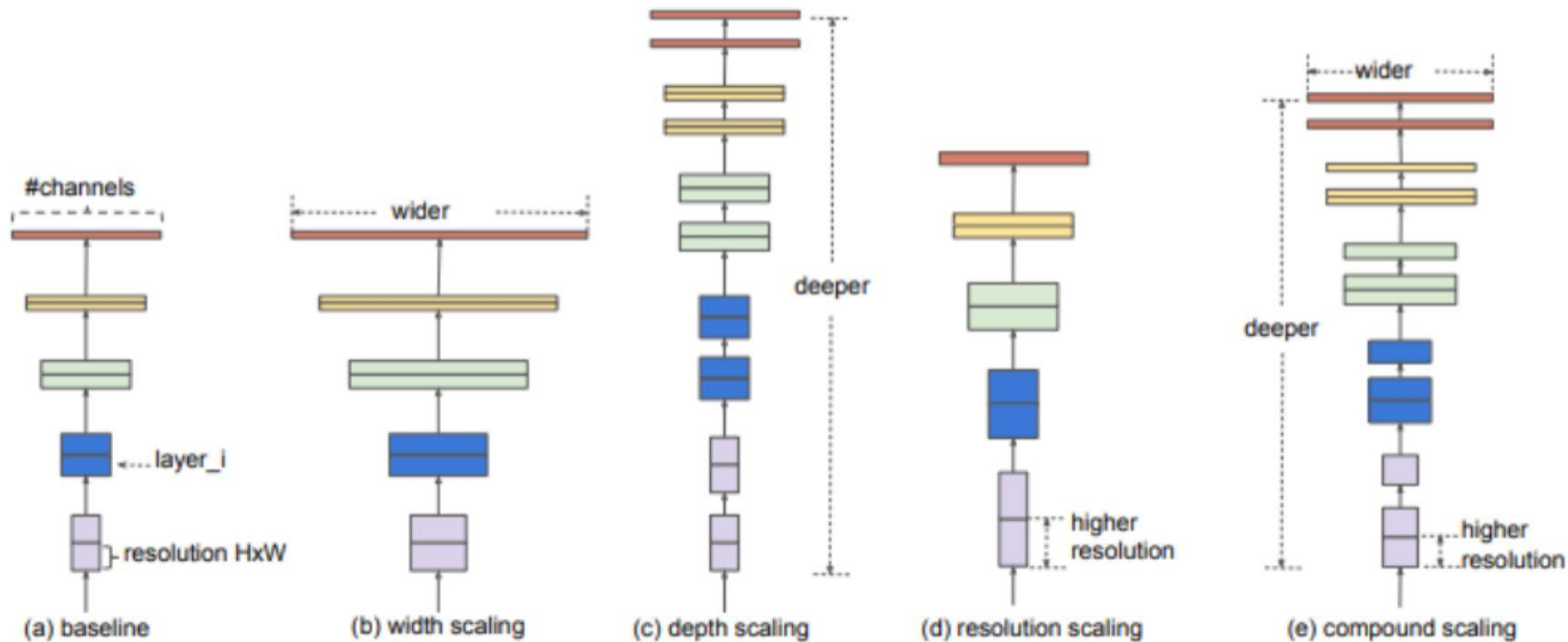
# Pre-trained Models - ResNet-50



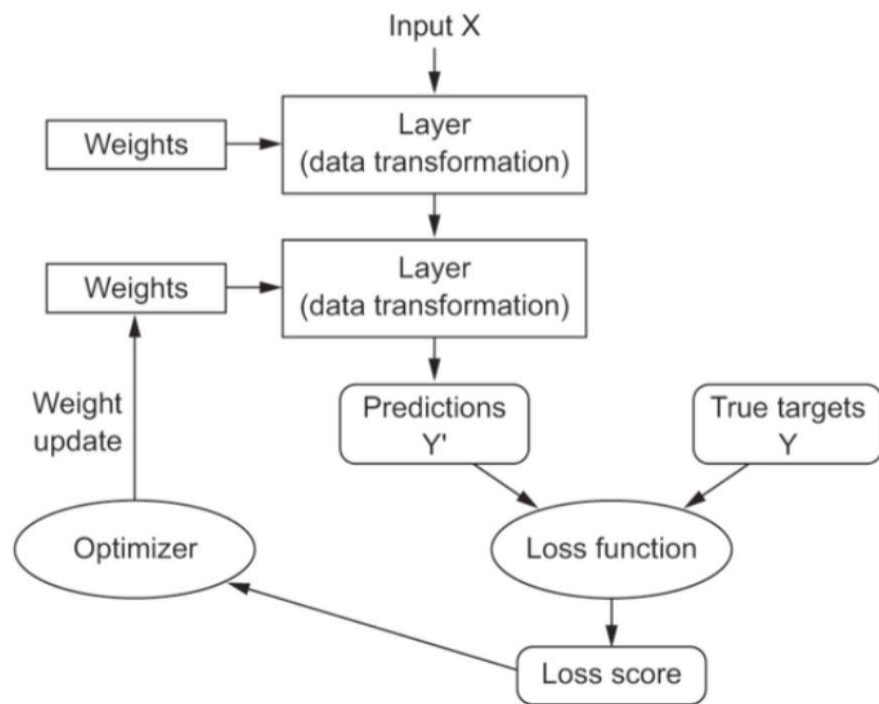
# Pre-trained Models - MobileNet Family



# Pre-trained Models - EfficientNet Family



# Training a Neural Network



- Perform parameter updates to **minimize** the loss (training objective)
- Typical flow involves:
  - Forward pass with the input going through various transformations
  - Compute the loss based on predictions and actuals
  - Compute gradients
  - Backpropagate gradients to update layer weights
- TensorFlow / PyTorch enables easy Automatic Differentiation



# Custom Training Loops in TensorFlow

```
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))

    # Iterate over the batches of the dataset.
    for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):

        # Open a GradientTape to record the operations run
        # during the forward pass, which enables auto-differentiation.
        with tf.GradientTape() as tape:

            # Run the forward pass of the layer.
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            logits = model(x_batch_train, training=True) # Logits for this minibatch

            # Compute the loss value for this minibatch.
            loss_value = loss_fn(y_batch_train, logits)

            # Use the gradient tape to automatically retrieve
            # the gradients of the trainable variables with respect to the loss.
            grads = tape.gradient(loss_value, model.trainable_weights)

            # Run one step of gradient descent by updating
            # the value of the variables to minimize the loss.
            optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

- GradientTape records all relevant NN operations in the forward pass
- Hence easy to compute gradients in reverse order during the backward pass
- Useful to extract relevant gradients w.r.t the loss (used in adversarial attacks)

# Adversarial Attacks - Examples



# Natural Adversarial Examples



Natural Adversarial Examples, Hendrycks et al.

# Natural Adversarial Examples

“ [...] DenseNet-121 obtains around 2% accuracy, an accuracy drop of **approximately 90%** [...] ”

# Natural Adversarial Examples



- Entire image being mapped to a single class

Natural Adversarial Examples,  
Hendrycks et al.



# Natural Adversarial Examples



- **Color** and **texture** as opposed to shape as the primary descriptors.

Natural Adversarial Examples, Hendrycks et al.

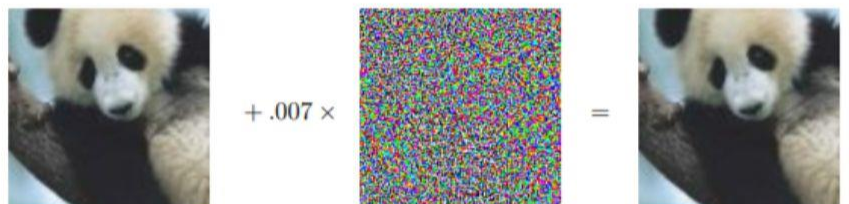
# Natural Adversarial Examples

## 5.1. Robust Training Methods Hardly Help

We examine popular robust training techniques. Unfortunately, we find that on natural adversarial examples for classifiers, these techniques hardly help. In this section we exclude IMAGENET-O results, as the robust training methods hardly help with out-of-distribution detection as well.

**$\ell_\infty$  Adversarial Training.** We investigate how much robustness  $\ell_\infty$  adversarial training confers, so we shall first describe  $\ell_\infty$  adversarial training, and then adversarially train ResNeXts. Adversarially training the parameters  $\theta$  with loss function  $L$  on dataset  $\mathcal{D}$  involves the objective

# Synthetic Adversarial Examples (via Attacks)



$x$   
"panda"  
57.7% confidence

$+ .007 \times$

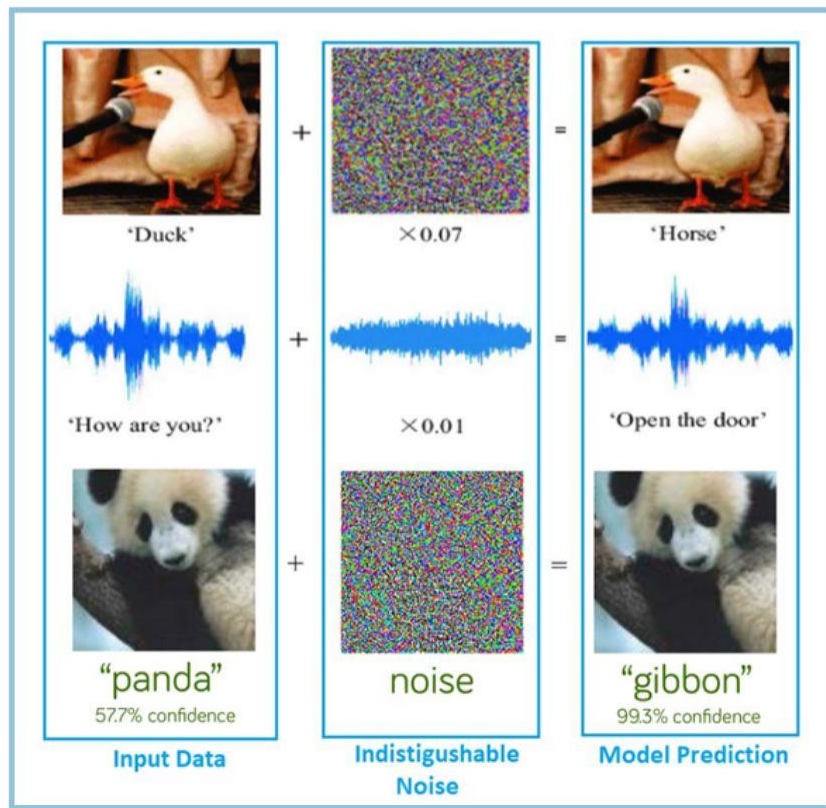
$\text{sign}(\nabla_x J(\theta, x, y))$   
"nematode"  
8.2% confidence

$=$

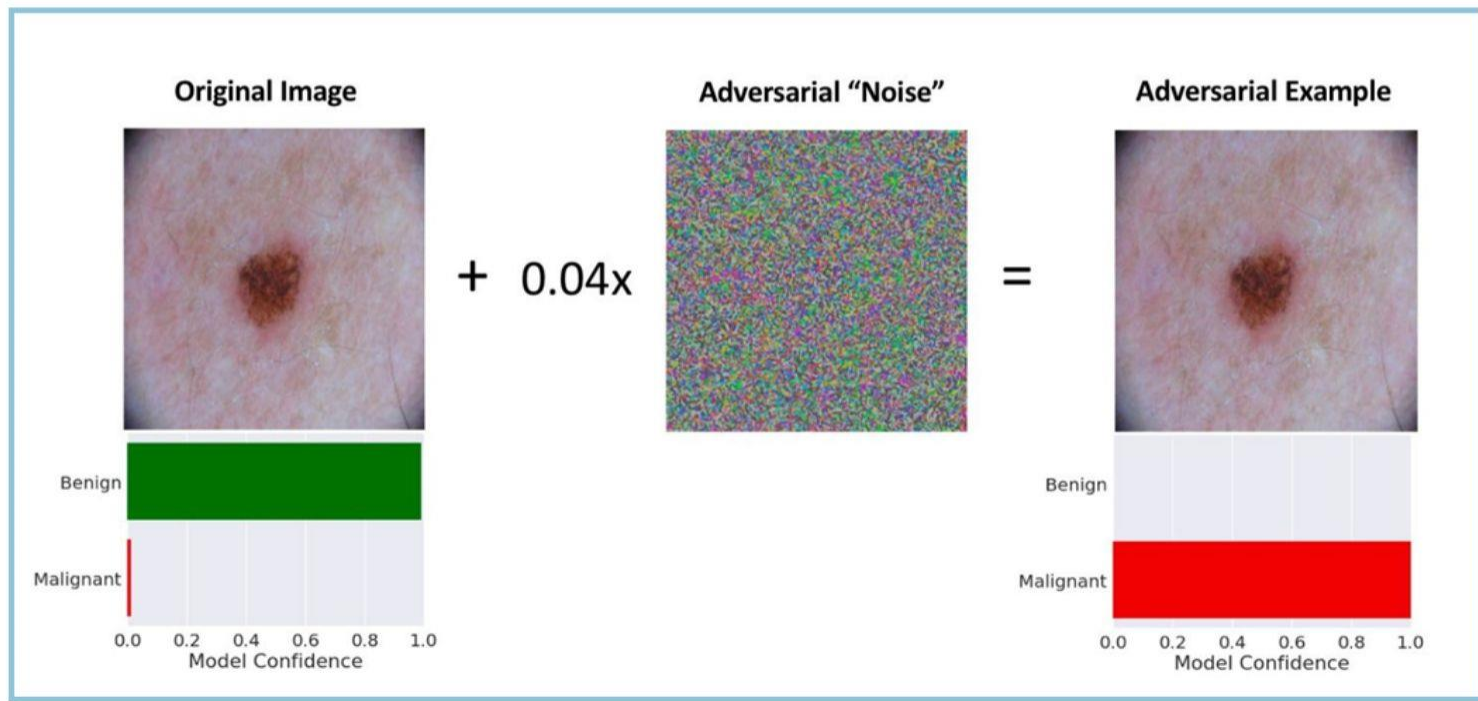
$x + \epsilon \text{sign}(\nabla_x J(\theta, x, y))$   
"gibbon"  
99.3 % confidence

- Purposely adding perturbations or noise in the data to fool the model
- Synthetically created on top of an input image
- Attacker adds small perturbations (distortions) to the original image
- These notorious perturbations are indistinguishable to the human eye, but causes the network to fail

# Impact of Adversarial Attacks

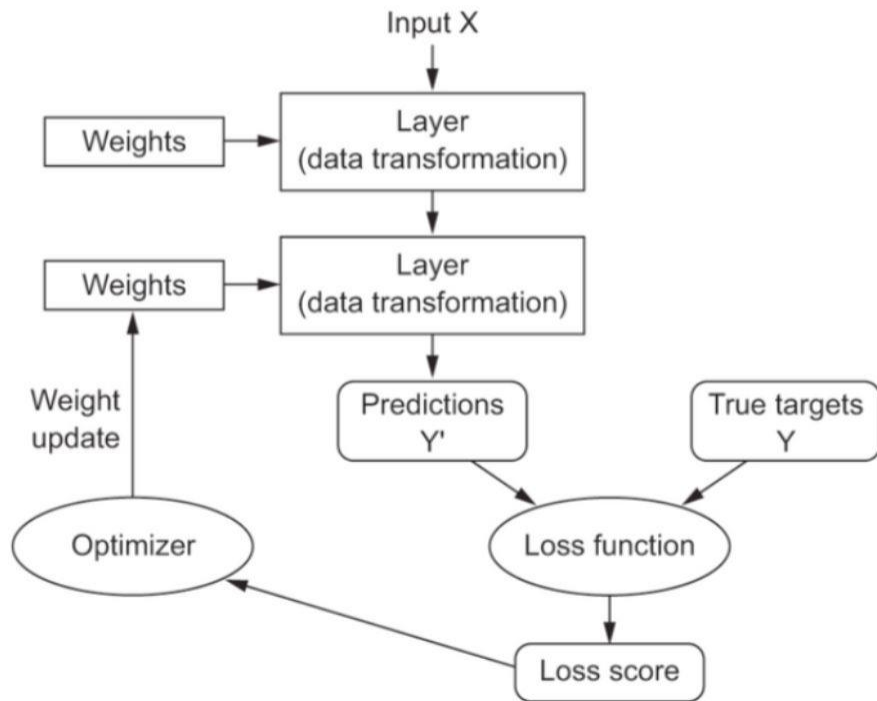


# Impact of Adversarial Attacks



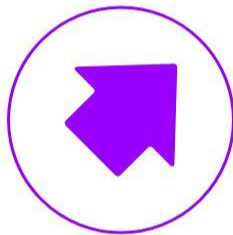


# Adversarial Attacks - Principles

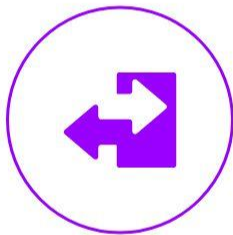


- What if we calculated gradients with respect to the **input data**?
- Update input data with these gradients to **maximize the loss** (instead of minimizing)?
- Gradients inform us how much to nudge the input data to affect the loss function

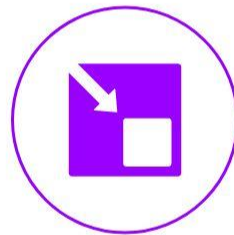
# Adversarial Attacks - Techniques



Projected Gradient Descent (PGD)



Targeted Attack



Fast Gradient Sign Method (FGSM)

# Adversarial Attacks - Projected Gradient Descent

$$\underset{\delta \in \Delta}{\text{maximize}} \ell(h_{\theta}(x + \delta), y)$$

$\hat{x}$  denotes our adversarial example

$h_{\theta}$  denotes our model, or hypothesis function

$x \in \mathcal{X}$  the input and  $y \in \mathbb{Z}$  the true class

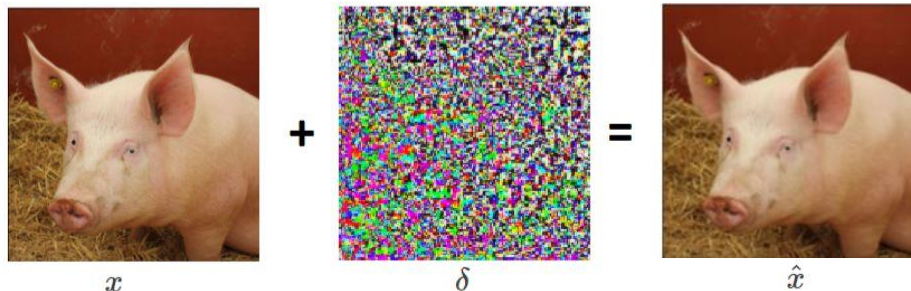
$\ell(h_{\theta}(x), y)$  denotes the loss

$\delta$  represents an allowable set of perturbations

such that  $\delta : \|\delta\|_{\infty} \leq \epsilon$

i.e., we allow the perturbation

to have magnitude between  $[-\epsilon, \epsilon]$



- Perturb the input image with a certain **delta** to maximize the network loss
  - Initially **delta** is assigned to all zeros
  - No random initialization to prevent rigorous optimizations
- Objective is to create an image across iterations that maximizes the loss
- After each optimization step, delta is projected back to a norm ball, in this case it is **L<sup>∞</sup> norm**
- We do this by clipping **delta** to  $[-\epsilon, +\epsilon]$  such that it doesn't change the visual semantics.
- Goal is to fool an already trained model.

# Adversarial Attacks - Projected Gradient Descent

$$\underset{\delta \in \Delta}{\text{maximize}} \ell(h_{\theta}(x + \delta), y)$$

$\hat{x}$  denotes our adversarial example

$h_{\theta}$  denotes our model, or hypothesis function

$x \in \mathcal{X}$  the input and  $y \in \mathbb{Z}$  the true class

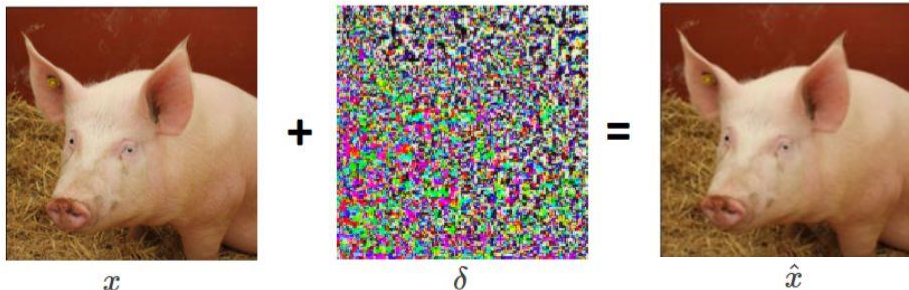
$\ell(h_{\theta}(x), y)$  denotes the loss

$\delta$  represents an allowable set of perturbations

such that  $\delta : \|\delta\|_{\infty} \leq \epsilon$

i.e., we allow the perturbation

to have magnitude between  $[-\epsilon, \epsilon]$



- Loss for predicting the true class (hog) goes up ↑
- This form of gradient descent-based optimization is also referred to as *Projected Gradient Descent*
- *delta* is random but learned



# Adversarial Attacks - Targeted Attack

$$\underset{\delta \in \Delta}{\text{maximize}} (\ell(h_{\theta}(x + \delta), y) - \ell(h_{\theta}(x + \delta), y_{\text{target}}))$$

$\hat{x}$  denotes our adversarial example

$h_{\theta}$  denotes our model, or hypothesis function

$x \in \mathcal{X}$  the input and  $y \in \mathbb{Z}$  the true class

$\ell(h_{\theta}(x), y)$  denotes the loss

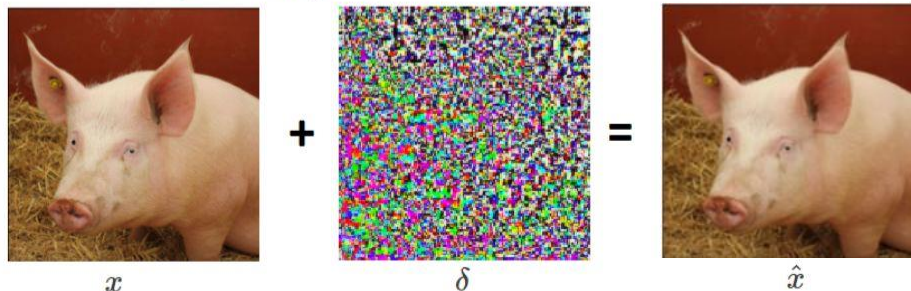
$\delta$  represents an allowable set of perturbations

such that  $\delta : \|\delta\|_{\infty} \leq \epsilon$

i.e., we allow the perturbation

to have magnitude between  $[-\epsilon, \epsilon]$

$y_{\text{target}}$  the target class



- Perturb the input image with a certain *delta* to maximize the network loss
  - Initially *delta* is assigned to all zeros
  - No random initialization to prevent rigorous optimizations
- Objective is to create an image across iterations that *maximizes* the loss of the *true class* and *minimizes* the loss of the *targeted class*
- After each optimization step, delta is projected back to a norm ball, in this case it is  *$L^{\infty}$  norm*
- We do this by clipping *delta* to  $[-\epsilon, +\epsilon]$  such that it doesn't change the visual semantics.
- Goal is to fool an already trained model.



# Adversarial Attacks - Targeted Attack

$$\underset{\delta \in \Delta}{\text{maximize}} (\ell(h_{\theta}(x + \delta), y) - \ell(h_{\theta}(x + \delta), y_{\text{target}}))$$

$\hat{x}$  denotes our adversarial example

$h_{\theta}$  denotes our model, or hypothesis function

$x \in \mathcal{X}$  the input and  $y \in \mathbb{Z}$  the true class

$\ell(h_{\theta}(x), y)$  denotes the loss

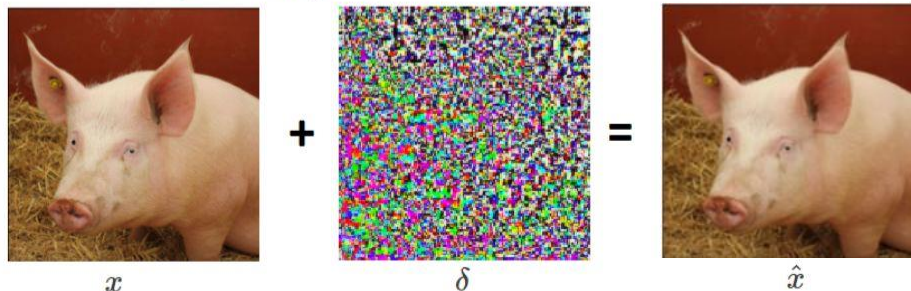
$\delta$  represents an allowable set of perturbations

such that  $\delta : \|\delta\|_{\infty} \leq \epsilon$

i.e., we allow the perturbation

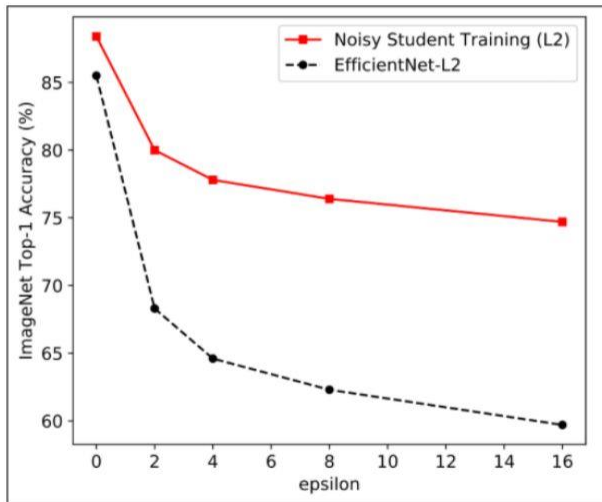
to have magnitude between  $[-\epsilon, \epsilon]$

$y_{\text{target}}$  the target class



- Loss for predicting the true class (hog) goes up  $\uparrow$
- Loss for predicting the target class (dog) goes down  $\downarrow$
- This form of targeted gradient descent-based optimization is also referred to as *Targeted Attack*
- *Delta* is random but learned

# Adversarial Defense with Noisy Student Training



- Doesn't incorporate any explicit adversarial training objective
- Inclusion of noisy during student training brings robustness
  - Strong augmentation
  - Stochastic Depth
  - Dropout

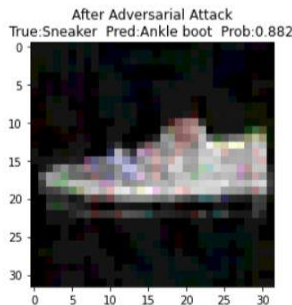
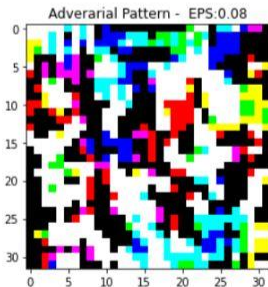
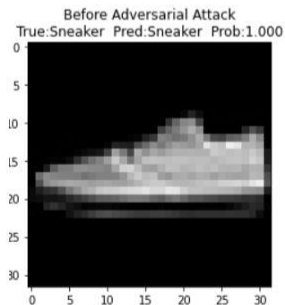
Source: Noisy student training; Xie et al.  
(2019)

# Adversarial Attacks - Fast Gradient Sign Method

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$

where

- $adv\_x$  : Adversarial image.
- $x$  : Original input image.
- $y$  : Original input label.
- $\epsilon$  : Multiplier to ensure the perturbations are small.
- $\theta$  : Model parameters.
- $J$  : Loss.



- Uses the gradients of the neural network to create an adversarial example
- Objective is to create an image that maximizes the loss.
- Gradients of the loss with respect to the input image are taken
- A small multiplier (epsilon) is added to the **sign of the gradients** and added to the original image
- Goal is to fool an already trained model.

# Adversarial Learning - Techniques



# Adversarial Learning - Defending against Attacks

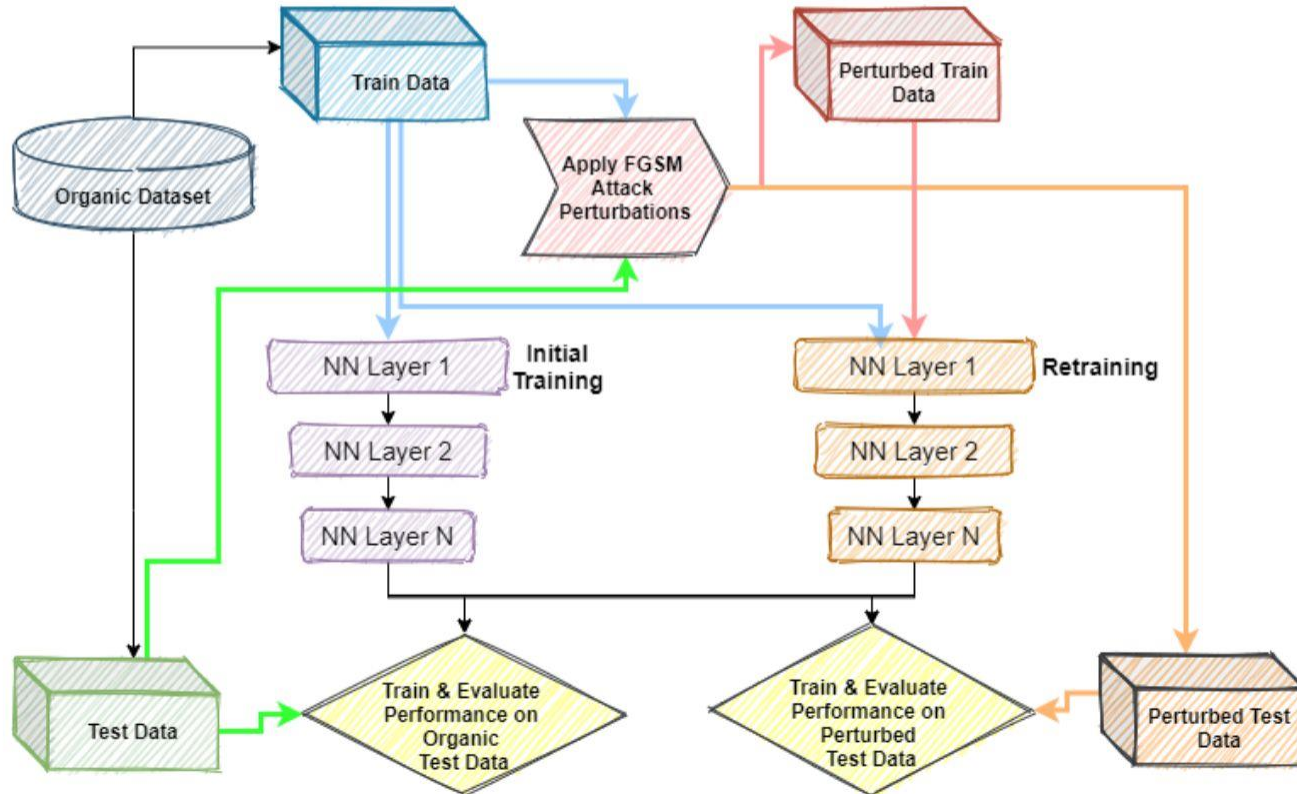
- It's still being actively researched
- Train models with adversarial data + organic data
- Use adversarial regularization loss during training
- But that may not be sufficiently enough (natural adversaries)
- Noisy student training shows promise
- So does smooth adversarial training



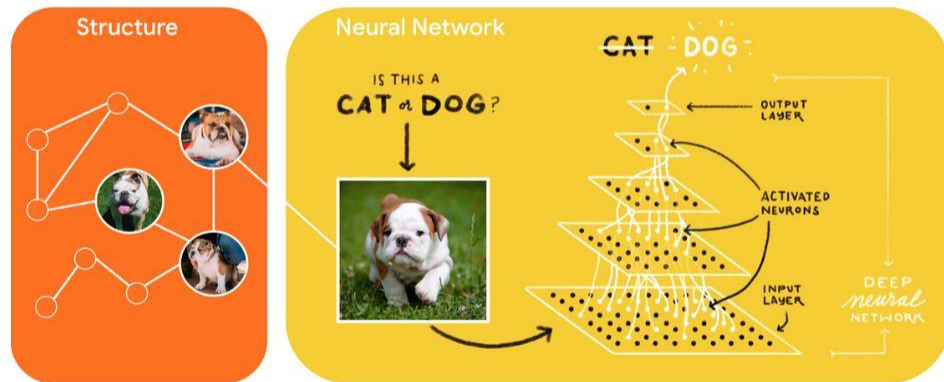
# Adversarial Learning - (Re)training from Scratch

- 1 Obtain the standard FashionMNIST dataset (Organic Data)
- 2 Train a simple CNN model on the Train dataset (Organic Data)
- 3 Evaluate performance of simple CNN on the Test Dataset (Organic Data)
- 4 Apply FGSM adversarial attacks on the datasets to create new Train and Test datasets (Perturbed Data)
- 5 Evaluate performance of simple CNN on Test Dataset (Perturbed Data)
- 6 Retrain our simple CNN with Organic Train + Perturbed Train Datasets
- 7 Evaluate retrained CNN performance on both Organic Test & Perturbed Test Datasets

# Adversarial Learning - (Re)training from Scratch

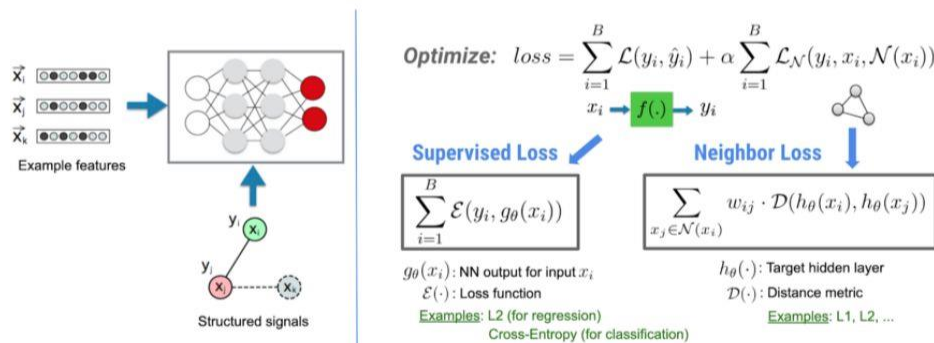


# Neural Structured Learning



- A new learning paradigm to train neural networks by leveraging structured signals in addition to feature inputs
- Structure can be explicit as represented by a graph or implicit
  - Implicit structure can be created by leveraging nearest neighbors similar to input
  - Adversarial examples created by perturbations on inputs can also be used
- Structured signals are commonly used to represent relations or similarity among samples
- Models trained with adversarial perturbation samples have been shown to be robust against malicious attacks

# Neural Structured Learning - Methodology



- Structured signals e.g. generated adversarial examples, are used to regularize the training of a neural network
- Objective is to minimize total loss  
$$\text{total\_loss} = \text{supervised\_loss} + \text{neighbor\_loss}$$
- Minimize supervised loss for accurate predictions
- Minimize neighbor loss to maintain the similarity among inputs from the same structure

# Adversarial Learning with NSL



- Create implicit structures by generating adversarial examples
- Perform Adversarial Regularization  
$$\text{total\_loss} = \text{supervised\_loss} + \text{adversarial\_loss}$$
- *Minimize supervised loss* for accurate predictions
- *Minimize adversarial loss* to maintain the similarity among inputs and their adversarial examples



# Adversarial Learning with TensorFlow - NSL

```
import neural_structured_learning as nsl
```

```
# Prepare data.
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

} Read  
Data

```
# Create a base model -- sequential, functional, or subclass.
```

```
model = tf.keras.Sequential(...)
```

} Keras  
Model

```
# Wrap the model with adversarial regularization.
```

```
adv_config = nsl.configs.make_adv_reg_config(multiplier=0.2, adv_step_size=0.05)
```

```
adv_model = nsl.keras.AdversarialRegularization(model, adv_config)
```

} Config  
Adv model

```
# Compile, train, and evaluate.
```

```
adv_model.compile(optimizer='adam',  
                  loss='sparse_categorical_crossentropy',  
                  metrics=['accuracy'])
```

```
adv_model.fit({'feature': x_train, 'label': y_train}, epochs=5)
```

```
adv_model.evaluate({'feature': x_test, 'label': y_test})
```

} Compile  
Fit  
Eval

# Adversarial Training Observations

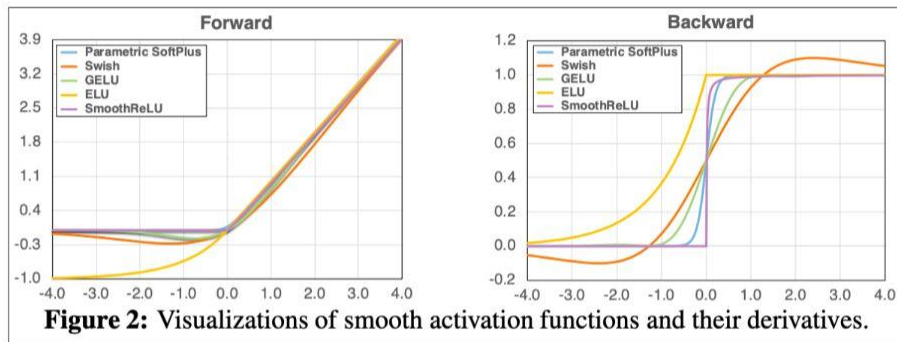
## Adversarial Examples Improve Image Recognition

Cihang Xie<sup>1,2\*</sup> Mingxing Tan<sup>1</sup> Boqing Gong<sup>1</sup> Jiang Wang<sup>1</sup> Alan Yuille<sup>2</sup> Quoc V. Le<sup>1</sup>  
<sup>1</sup>Google <sup>2</sup>Johns Hopkins University

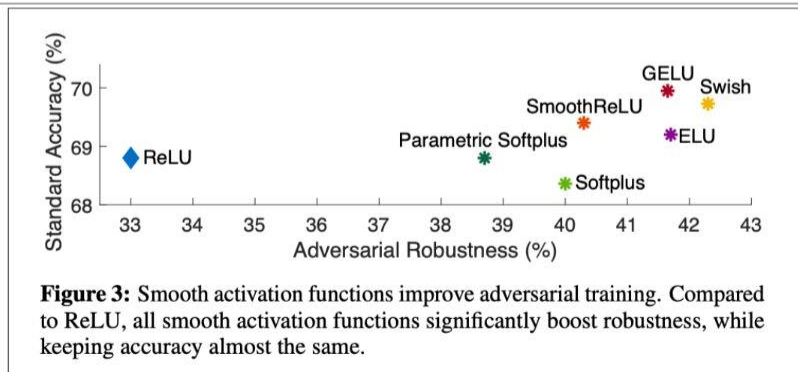
Paper: [arxiv.org/abs/1911.09665](https://arxiv.org/abs/1911.09665)

- Adversarial training methods that depend on a specific perturbation technique may not generalize well to other perturbations
  - This is where Noisy Student Training can be really helpful
- It's important to retrain with the perturbed training set **and** the original training set to prevent the model from *catastrophic forgetting*
- Smoother activations (e.g. Swish) tend to work well when adversarial training takes place from the beginning

# Smooth Adversarial Training



- Replaces RELU with smoother approximations like GELU, Swish etc.
- Helps to produce harder adversarial examples during the training because of more informed gradients.



# References

- **Visuals & Content**

- [https://www.tensorflow.org/neural\\_structured\\_learning](https://www.tensorflow.org/neural_structured_learning)
- <https://medium.com/tensorflow/introducing-neural-structured-learning-in-tensorflow-5a802efd7afd>
- <https://github.com/sayakpaul/Image-Adversaries-101>
- [https://github.com/dipanjanS/convolutional\\_neural\\_networks\\_essentials](https://github.com/dipanjanS/convolutional_neural_networks_essentials)
- [https://www.tensorflow.org/tutorials/generative/adversarial\\_fgsm](https://www.tensorflow.org/tutorials/generative/adversarial_fgsm)
- [https://www.tensorflow.org/neural\\_structured\\_learning/tutorials/adversarial\\_keras\\_cnn\\_mnist](https://www.tensorflow.org/neural_structured_learning/tutorials/adversarial_keras_cnn_mnist)
- <https://adversarial-ml-tutorial.org/introduction>

- **Research Papers**

- [https://github.com/dipanjanS/adversarial-learning-robustness/tree/main/research\\_papers](https://github.com/dipanjanS/adversarial-learning-robustness/tree/main/research_papers)

# Stay in Touch!



LinkedIn

[linkedin.com/in/dipanzan](https://linkedin.com/in/dipanzan)



GitHub

[github.com/dipanjans](https://github.com/dipanjans)



LinkedIn

[linkedin.com/in/sayak-paul](https://linkedin.com/in/sayak-paul)



GitHub

[github.com/sayakpaul](https://github.com/sayakpaul)