CSCI 1470

Eric Ewing

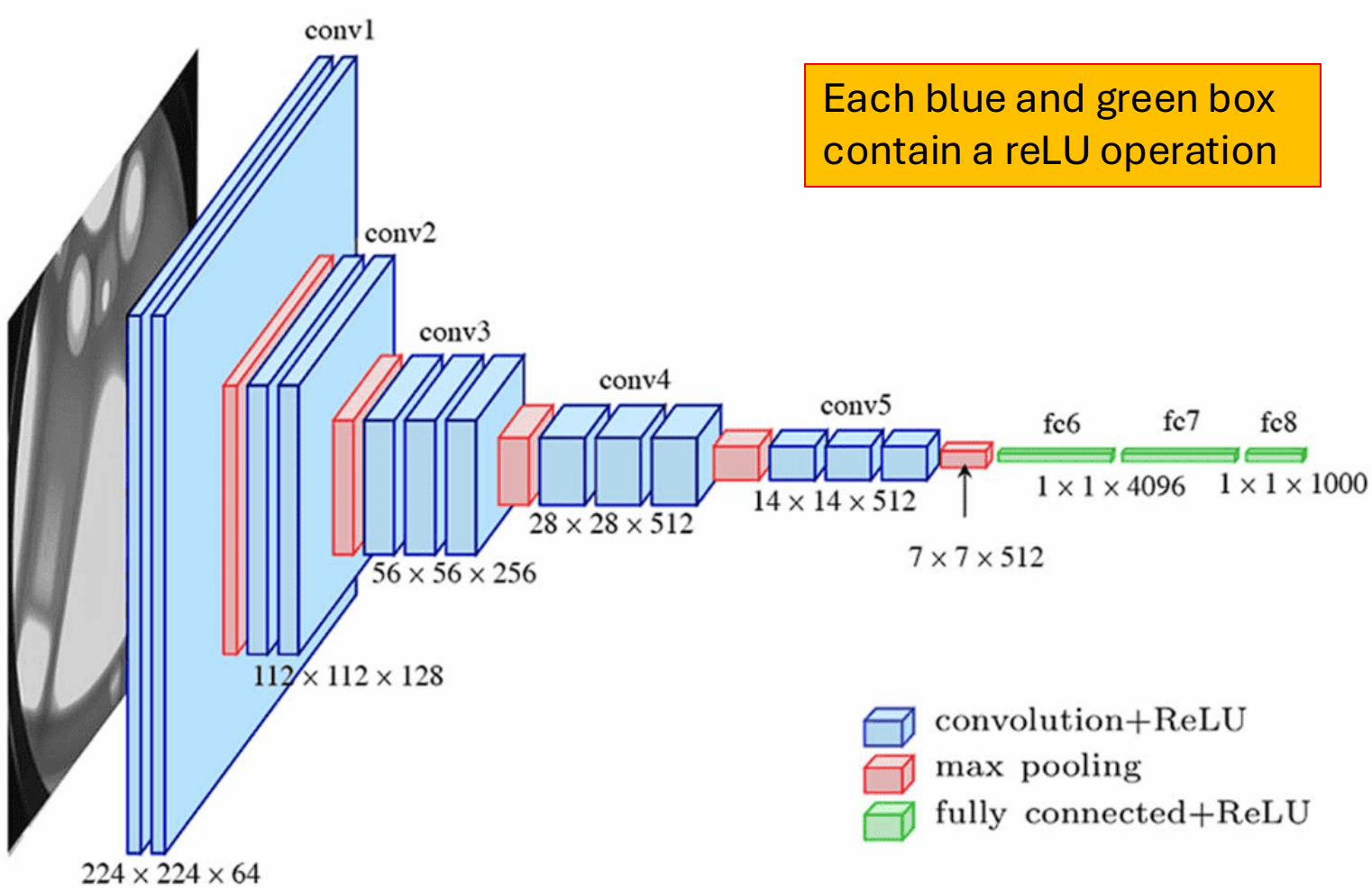
Thursday, 10/2/25

Deep Learning

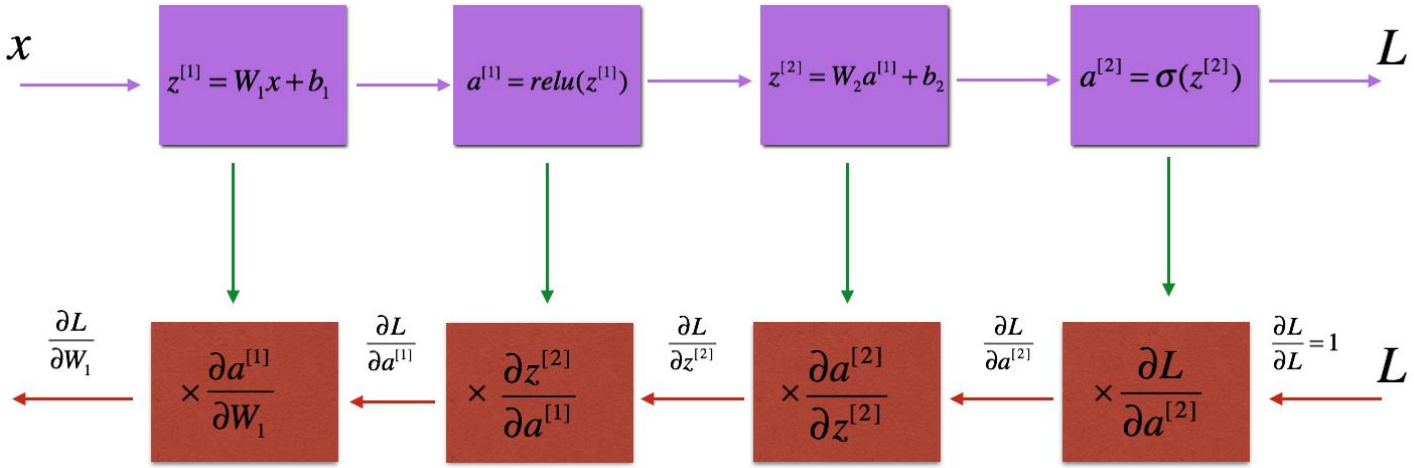
Day 9: ResNet and Adversarial Learning

Depth is increasing ☺

But… other problems start to occur ☹



Each blue and green box contain a reLU operation



$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial W_1}$$

≤ 1

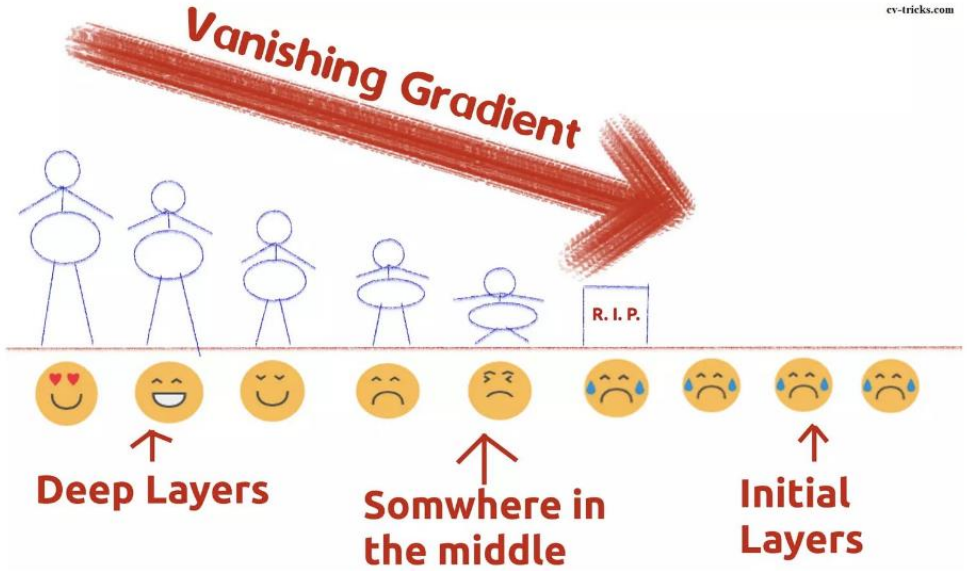
Multiplying by terms ≤ 1 makes things smaller...
Gradients earlier in the network tend to "Vanish"

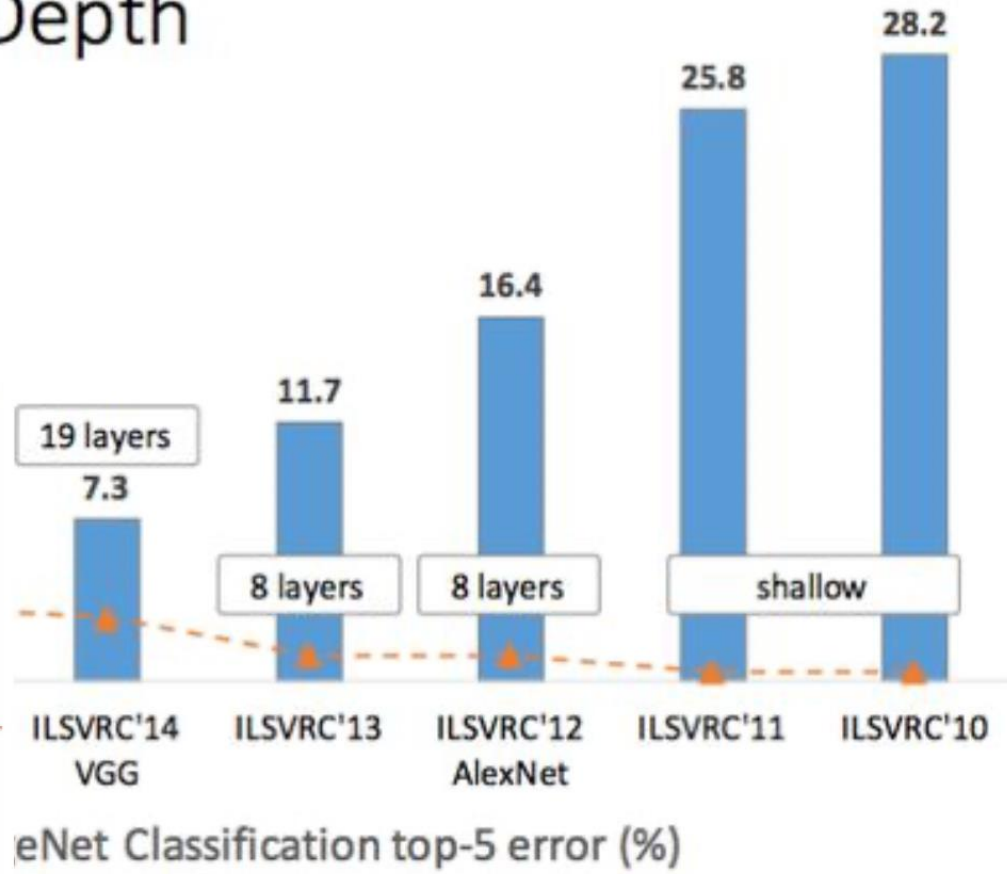
Adding more layers adds more terms with gradient ≤ 1

https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33, https://cv-tricks.com/keras/understand-implement-resnets/

Revolution of Depth

Vanishing gradients!





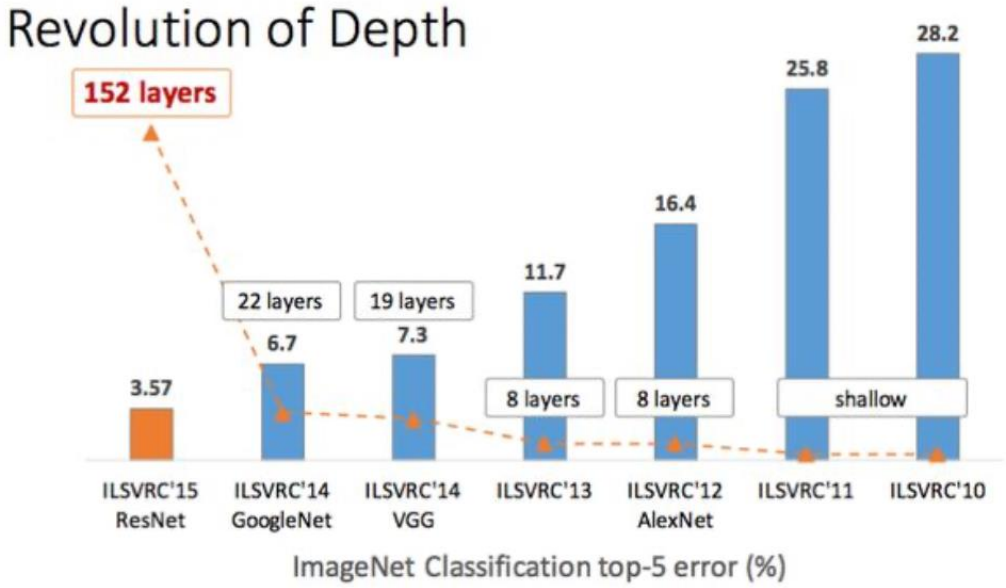
More Complicated Networks

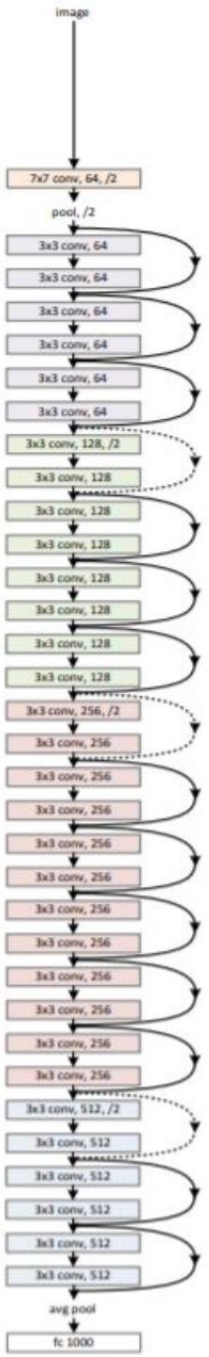
ResNet:

Lots of layers, tons of learnable parameters

Avoids Vanishing Gradient problem

but how?



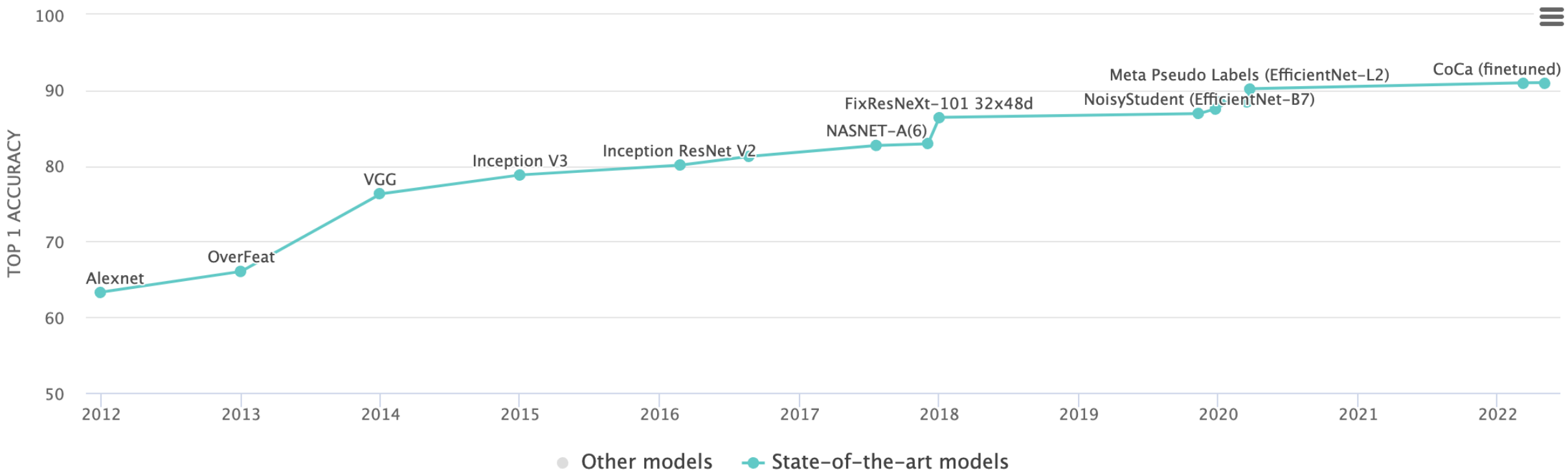


K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.

Image Classification on ImageNet

Leaderboard | Community Models | Dataset

View: Top 1 Accuracy | by: Date | for: All models



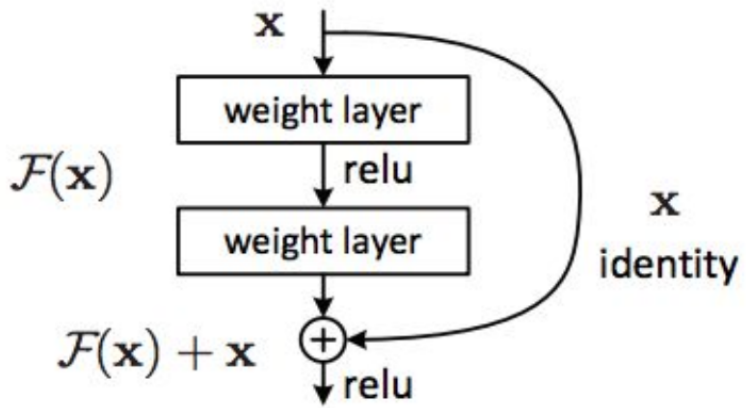
More Complicated Networks

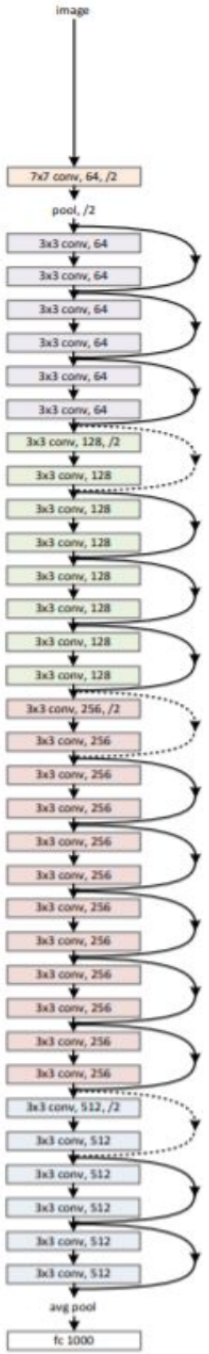
ResNet:

Lots of layers, tons of learnable parameters

Avoids Vanishing Gradient problem

Residual Block

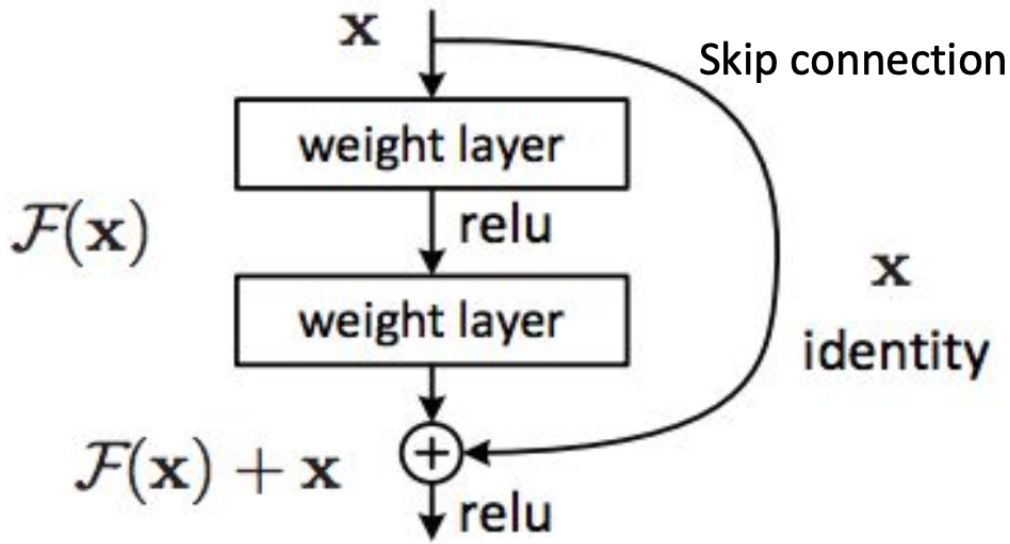




K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.

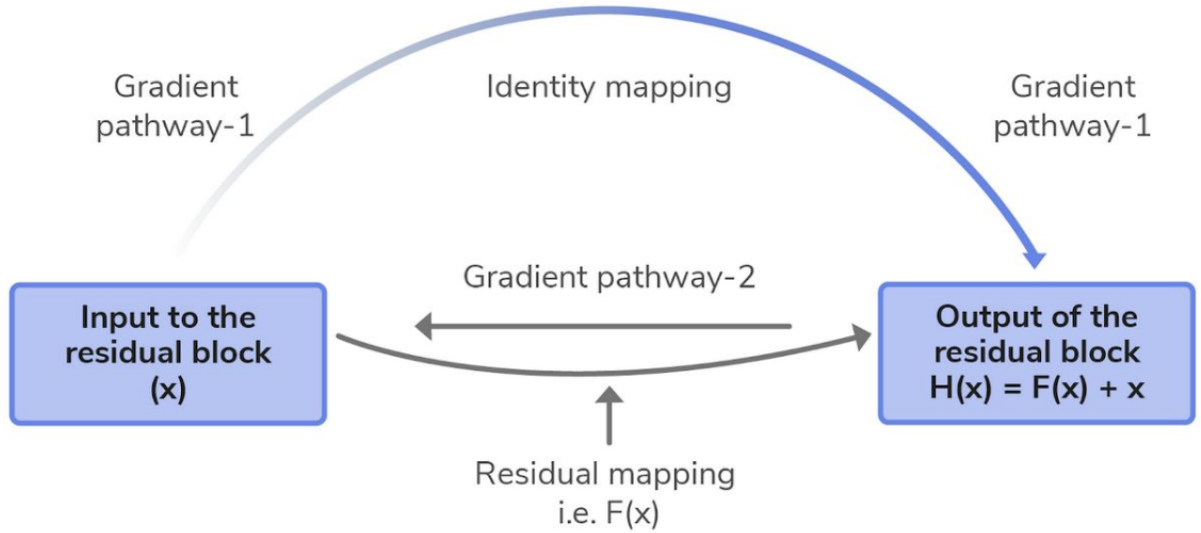
Residual Blocks

- In very deep nets, each layer often needs to learn just a small transformation of the preceding layer (identity + change)
- Idea: explicitly design the network such that the output of each layer is the identity + some deviation from it
  - Deviation is known as a residual



Residual Blocks

- In very deep nets, each layer often needs to learn just a small transformation of the preceding layer (identity + change)
- Idea: explicitly design the network such that the output of each layer is the identity + some deviation from it
  - Deviation is known as a residual
- Allows gradient to flow through two pathways
- **Significantly stabilizes training of very deep networks**



https://blog.perceptilabs.com/using-resnets-to-detect-anomalies-in-industrial-iot-textile-production/

Tensorflow

Option #1: Residual Block

tfm.vision.layers.ResidualBlock(filters, strides)

Option #2:

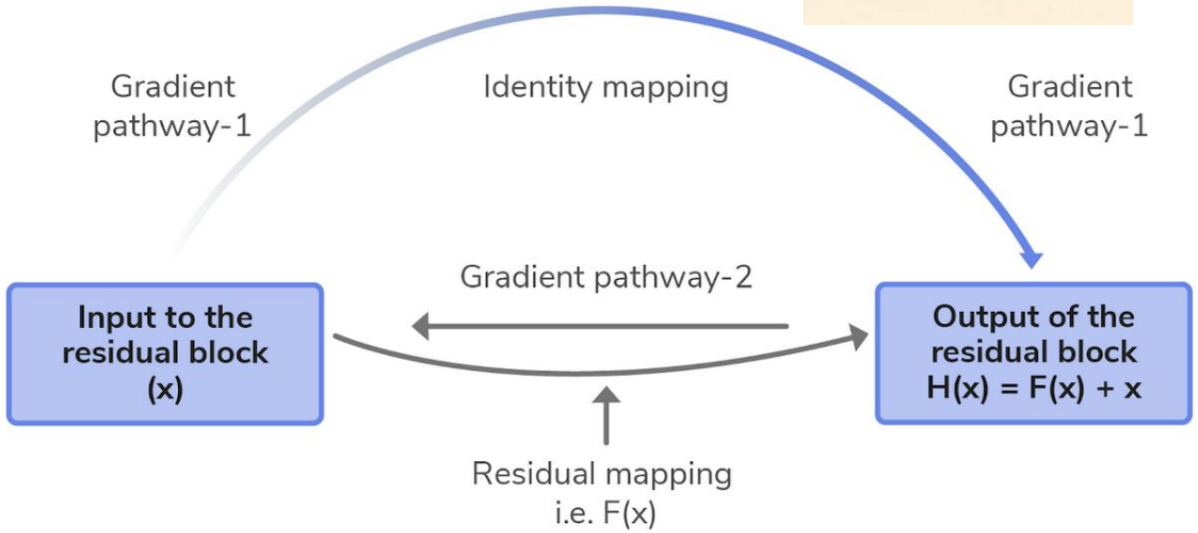
```
# Residual Block
def ResBlock(inputs):
    x = layers.Conv2D(64, 3, padding="same", activation="relu")(inputs)
    x = layers.Conv2D(64, 3, padding="same")(x)
    x = layers.Add()([inputs, x])
    return x
```

Original Input Intermediate Output

https://keras.io/examples/vision/edsr/

Residual Blocks

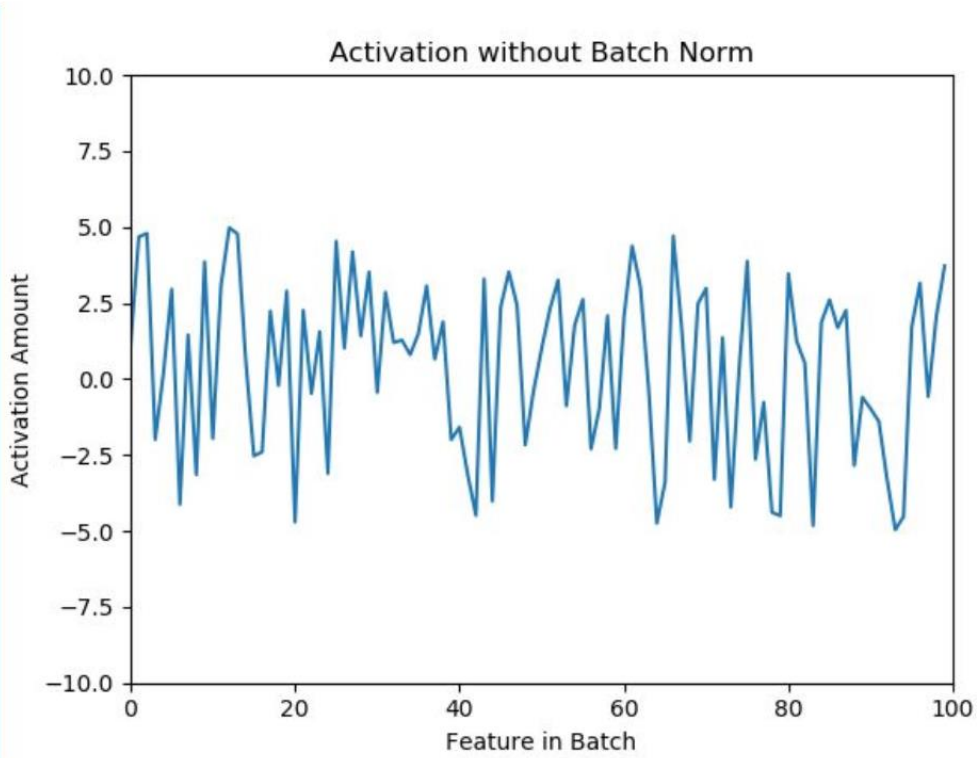
- In very deep nets, each layer often needs to learn just a small transformation of the preceding layer (identity + change)
- Idea: explicitly design the network such that the output of each layer is the identity + some deviation from it
  - Deviation is known as a residual
- Allows gradient to flow through two pathways
- **Significantly stabilizes training of very deep networks**

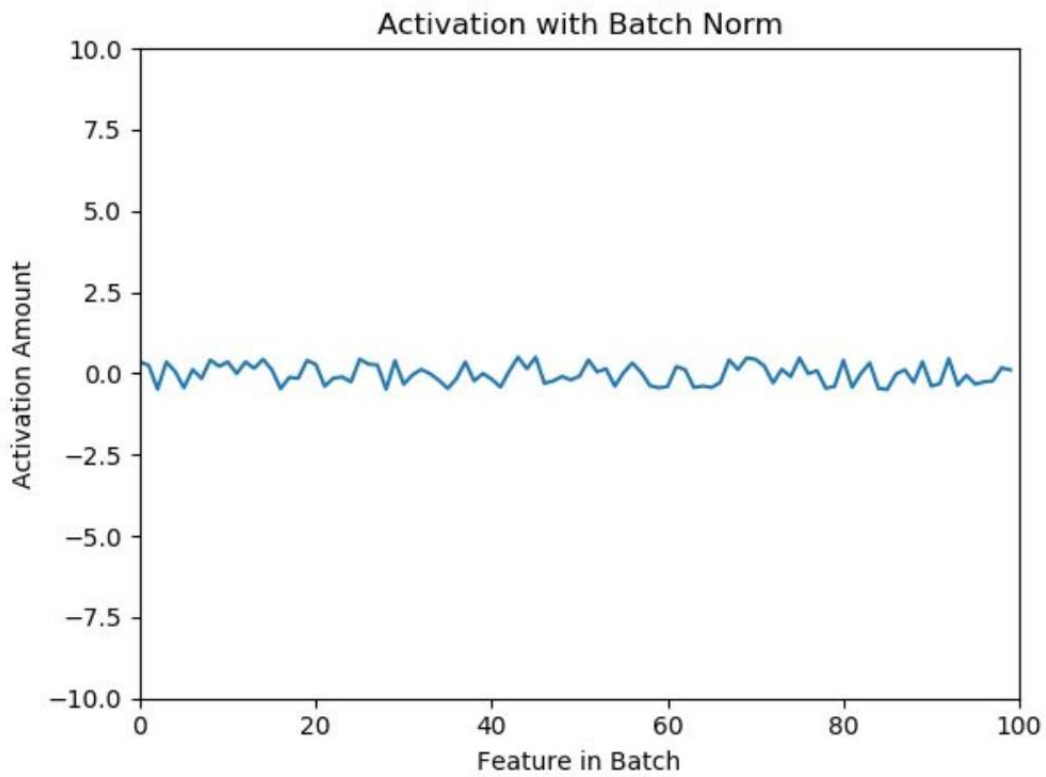


https://blog.perceptilabs.com/using-resnets-to-detect-anomalies-in-industrial-iot-textile-production/

Batch Normalization (stabilizing training)

Idea: normalize the activations for each feature at each layer



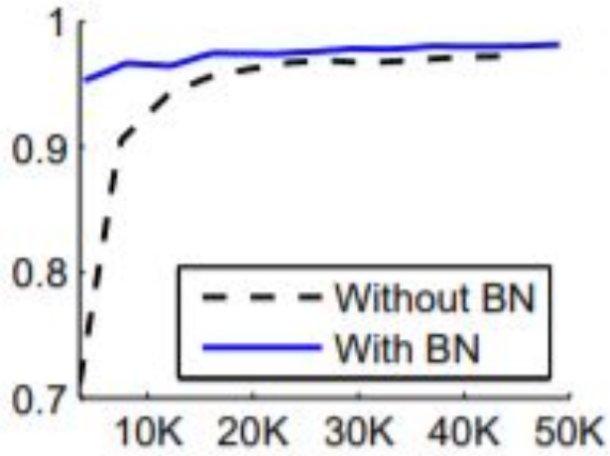


Why might we want to do this?

Batch Normalization: Motivation

More stable inputs = faster training

MNIST test accuracy vs number of training steps



https://arxiv.org/pdf/1502.03167.pdf

Batch Normalization: Implementation

For each feature x, Start by calculating the batch mean and standard deviation for each feature:

$$\mu_{batch} = \frac{\sum_{i=0}^{batch\_size} x_i}{batch\_size}$$

$$\sigma_{batch} = \sqrt{\frac{\sum_{i=0}^{batch\_size} (x_i - \mu_{batch})^2}{batch_{size}}}$$

Batch Normalization: Implementation

Normalize by subtracting feature x's batch mean, then divide by batch standard deviation.

$$x' = \frac{x - \mu_{batch}}{\sigma_{batch}}$$

Feature x now has mean 0 and variance 1 along the batch

Batch Normalization in Tensorflow

```
tf.keras.layers.BatchNormalization(input)
```

Documentation: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/BatchNormalization

Motivation of BatchNorm

- Reduce “internal co-variate shift”
- Neural networks are trained on a certain distribution of data and are expected to be tested on the same distribution
- If we were to scale the colors of an image significantly at test time, we wouldn’t expect a neural network to do well
- The same can be said for our intermediate layers
  - They expect a certain distribution of inputs, if that changes significantly from example to example, it will be hard to learn
- (Most commonly cited reason for using BatchNorm)

The only issue is that controlling internal covariate shift does not matter that much…

How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

BatchNorm makes the loss landscape smoother with fewer local minima, saddle points, and other problematic areas for gradient descent

How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

Theory, intuition, and experimental results can all tell you different things

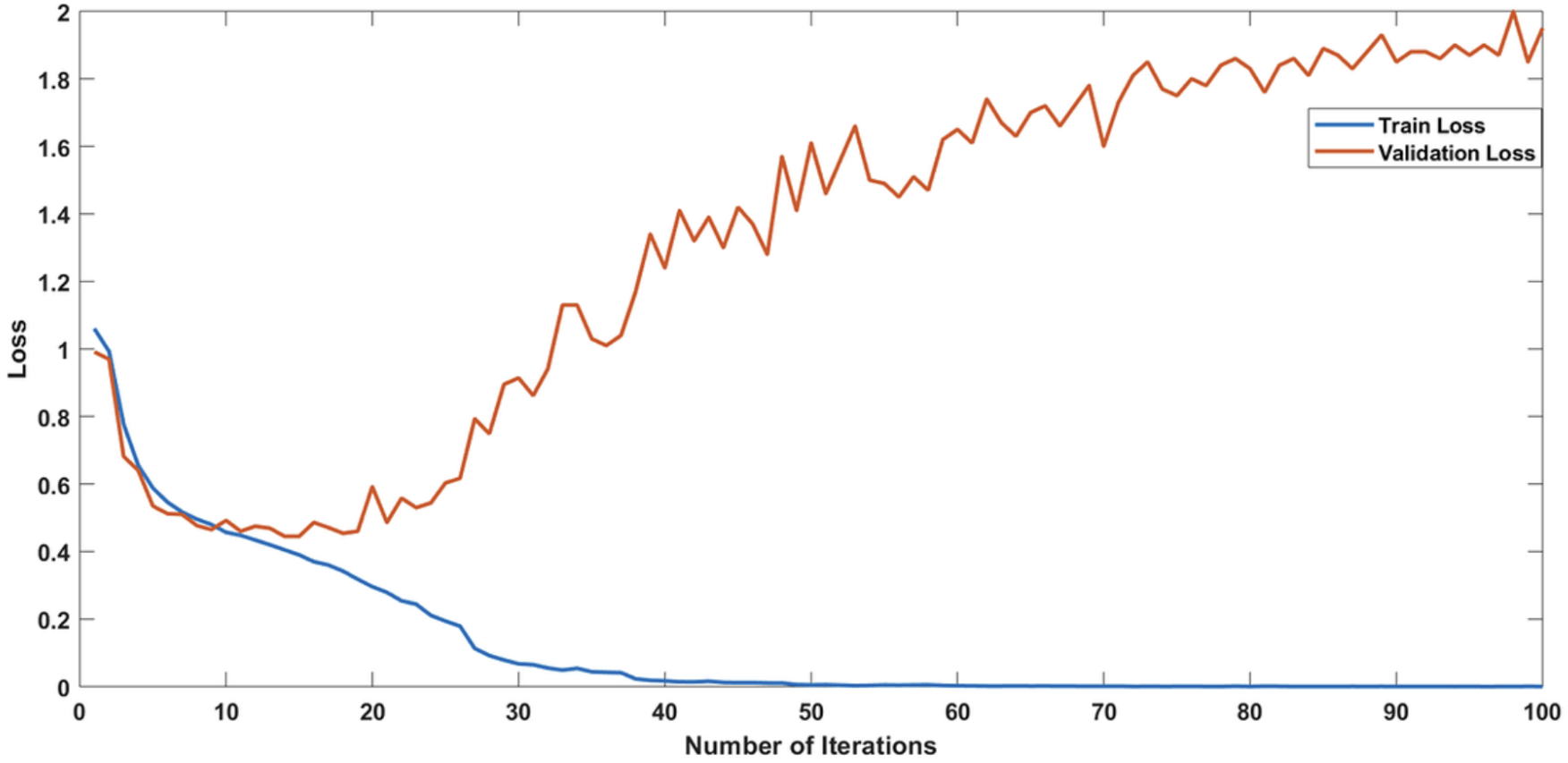
Why does BatchNorm work so well?
Intuition: If normalizing input data works so well for training, why not normalize input features to intermediate layers?

Theory/experiments: Makes gradients of loss function “better”

Why do CNNs work so well?
Intuition: Looking for a way to get “spatial reasoning” or translational invariance

Theory/experiments: Maybe it’s just that using fewer weights lets us go deeper and deep networks learn better (and also they have spatial reasoning)

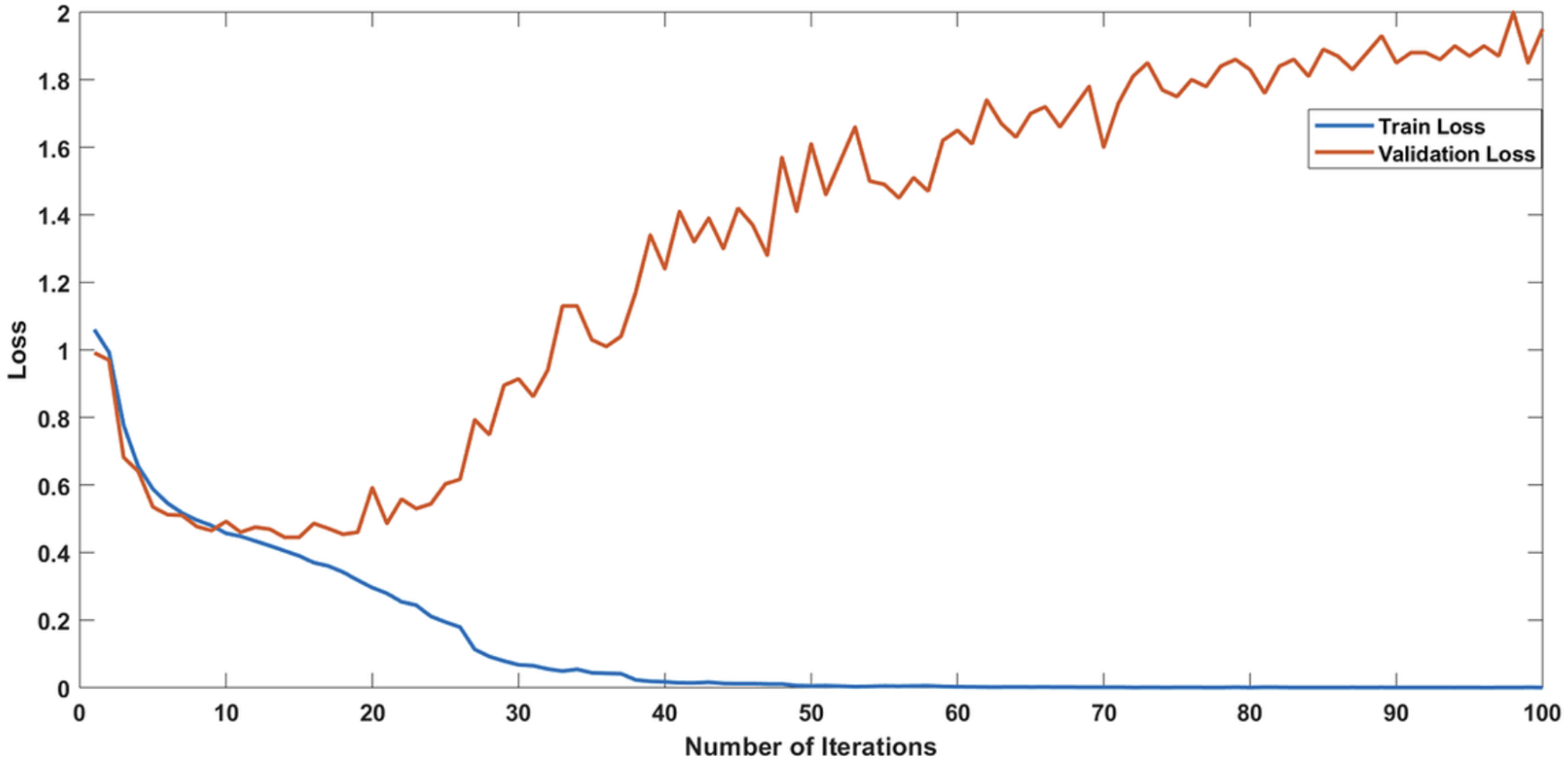
Depth Giveth and Depth Taketh Away



Resnet trained on image classification task

Depth Giveth and Depth Taketh Away

What's the problem?



Resnet trained on image classification task

Dealing with Overfitting (Again)

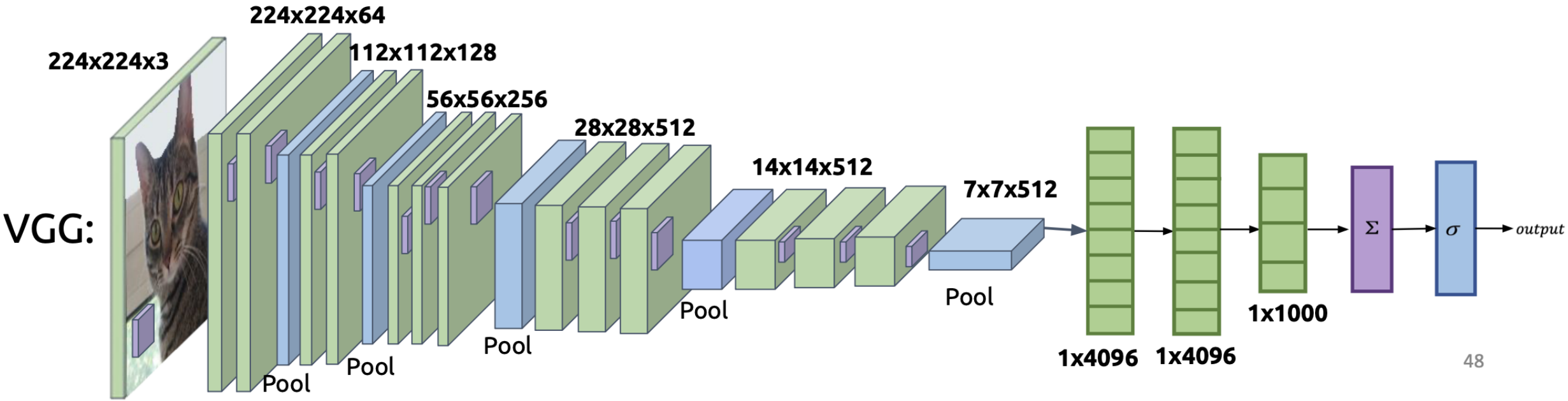
Option #1: Hyperparameter Tuning

- Try a shallower network

Dealing with Overfitting (Again)

Option #1: Hyperparameter Tuning

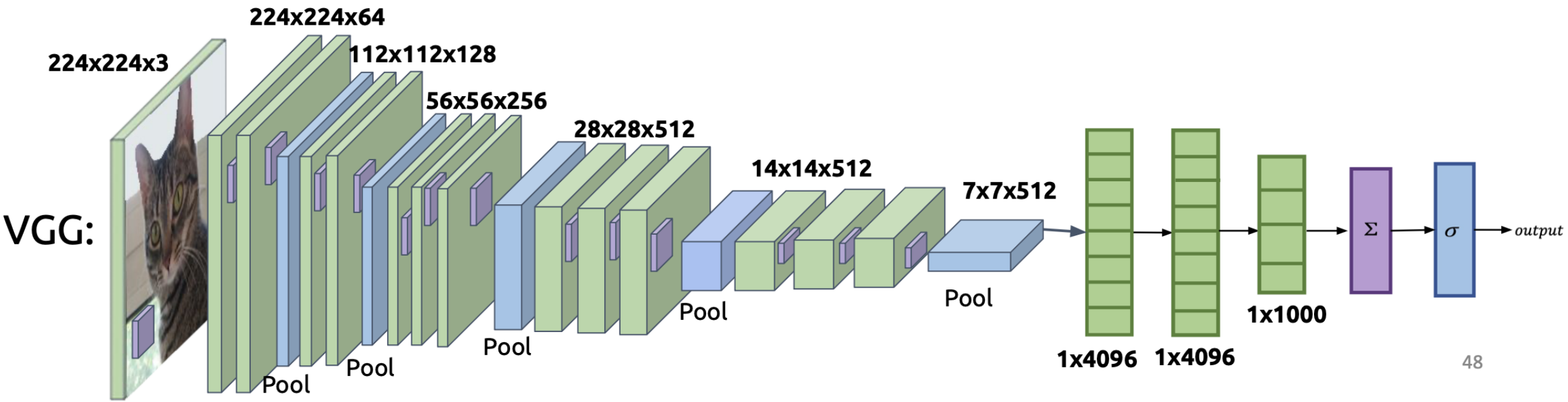
- Try a shallower network



Dealing with Overfitting (Again)

Option #1: Hyperparameter Tuning

- Try a shallower network



The size of the linear layer is controlled by number of max-pools
Fewer convolutions could actually increase weights in the network...

Dealing with Overfitting (Again)

Option #1: Hyperparameter Tuning

- Try a shallower network
- Fewer channels in convolutions

Hyperparameter Tuning

- Manually tuning parameters is seen by DL practitioners as a bit “old fashioned”
  - The goal of deep learning is to automatically find good models in a general way
  - Any human-driven heuristic approach makes the process specific

Hyperparameter Tuning

- Manually tuning parameters is seen by DL practitioners as a bit "old fashioned"
  - The goal of deep learning is to automatically find good models in a general way
  - Any human-driven heuristic approach makes the process specific

  Can we write a method to ___ and then run deep learning on that output?

  (center the image, recognize letters on signs, label parts of a sentence)

The Bitter Lesson of AI

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.

Richard Sutton

The Bitter Lesson of AI

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.

Richard Sutton

1) AI researchers have often tried to build knowledge into their agents
2) This always helps in the short term, and is personally satisfying to the researcher, but
3) In the long run it plateaus and even inhibits further progress
4) Breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning.

Hyperparameter Tuning

- Manually tuning parameters is seen by DL practitioners as a bit "old fashioned"
  - The goal of deep learning is to automatically find good models in a general way
  - Any human-driven heuristic approach makes the process specific

  Can we write a method to ___ and then run deep learning on that output?
  (center the image, recognize letters on signs, label parts of a sentence)

Manual hyperparameter tuning is a flaw that needs to be overcome

Dealing with Overfitting (Again)

Option #1: Hyperparameter Tuning

- Try a shallower network
- Fewer channels in convolutions

Option #2: Regularization

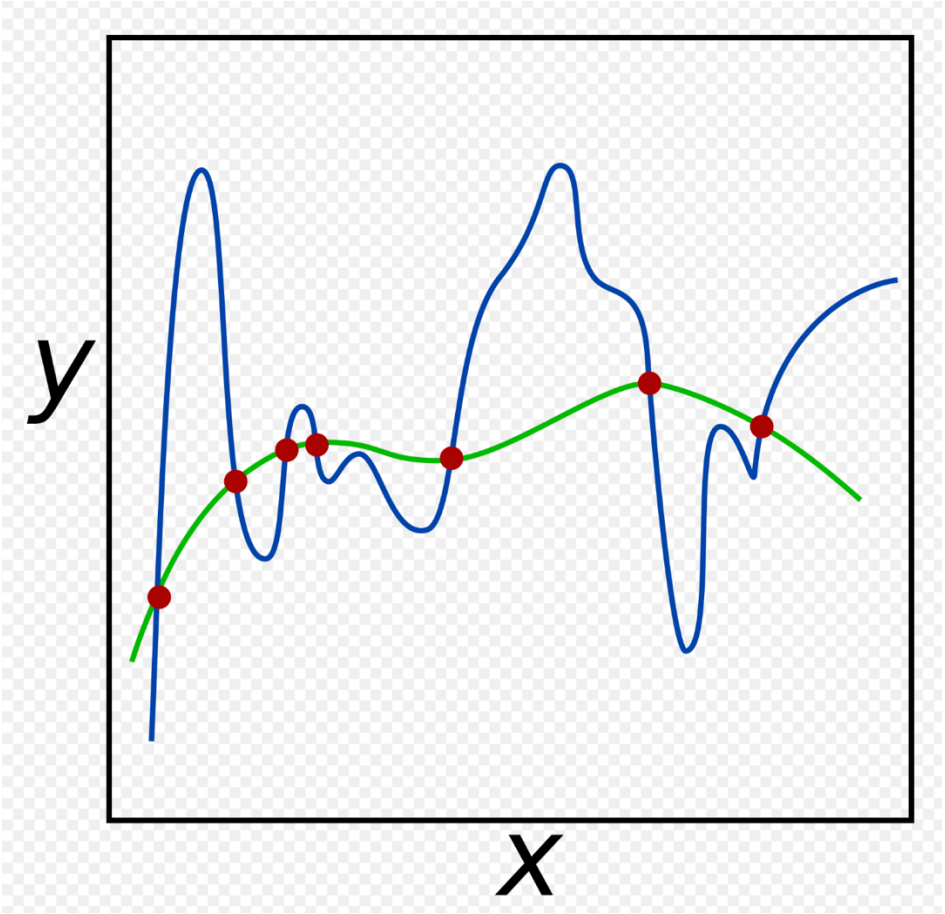
- “Encourage” model to be lower complexity

Regularization: L2 Norm Penalty

Intuition: high degree polynomials typically don't work for regression tasks because they overfit.

When they overfit, the parameters of some terms get very large.

Let's penalize the model for having large parameters.

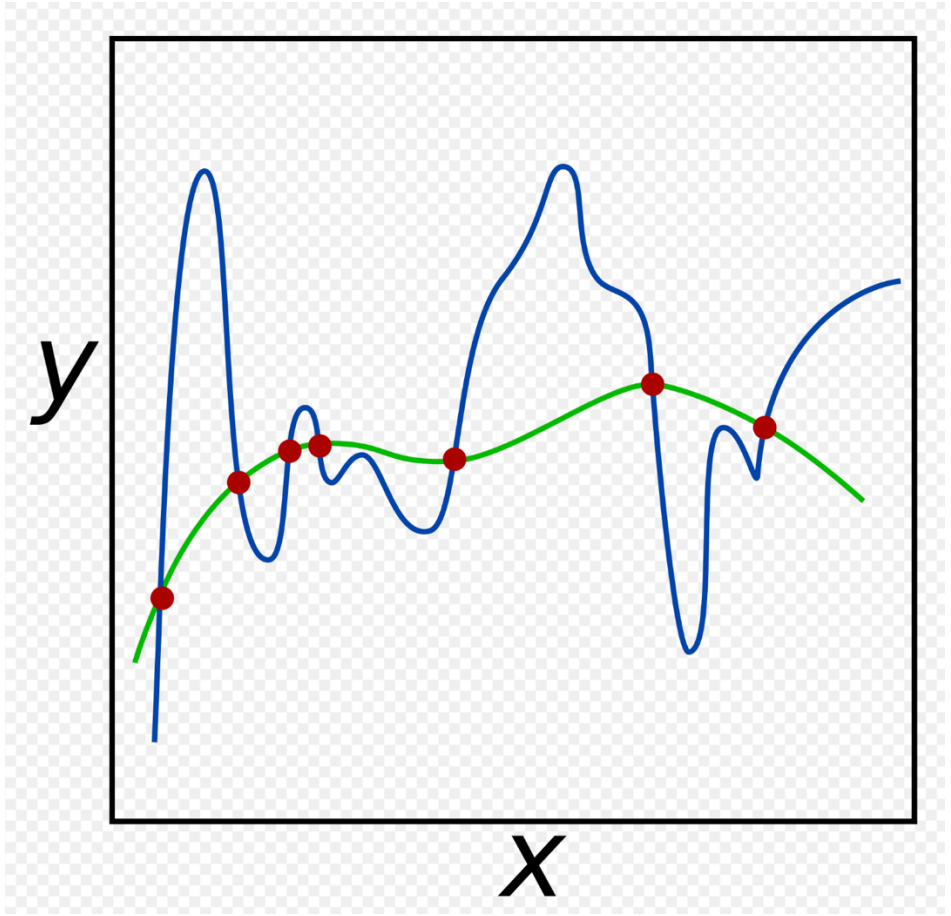


Regularization: L2 Norm Penalty

Intuition: high degree polynomials typically don't work for regression tasks because they overfit.

When they overfit, the parameters of some terms get very large.

Let's penalize the model for having large parameters.



Original Loss=$MSE(y, \hat{y})$

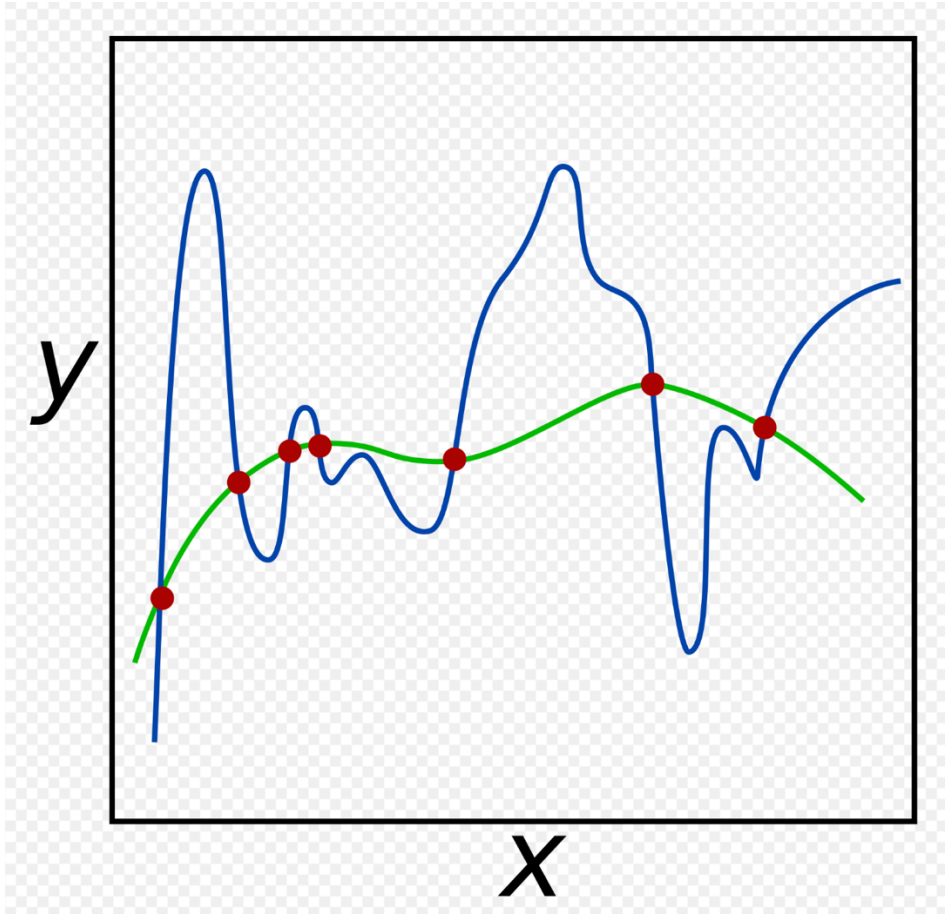
L2 Regularization Loss =$MSE(y, \hat{y}) + \lambda(w_0^2 + w_1^2 + w_2^2 \ldots)^{\frac{1}{2}}$

Regularization: L2 Norm Penalty

Intuition: high degree polynomials typically don't work for regression tasks because they overfit.

When they overfit, the parameters of some terms get very large.

Let's penalize the model for having large parameters.



Original Loss=$MSE(y, \hat{y})$

L2 Regularization Loss =$MSE(y, \hat{y}) + \lambda(w_0^2 + w_1^2 + w_2^2 \ldots)^{\frac{1}{2}}$

L2 Norm (2 refers to power)

Regularization L2 Norm Penalty

- Why do neural networks overfit? Perhaps their weights get large as well.
- Can add a penalty to all weights or individual layers
- Smaller weights → simpler function learned

```
from keras import regularizers

model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)))
```

Dropout – general intuition

- Preventing the network from learning under perfect conditions; that is, make it **harder** for the network to learn

A climbing analogy:

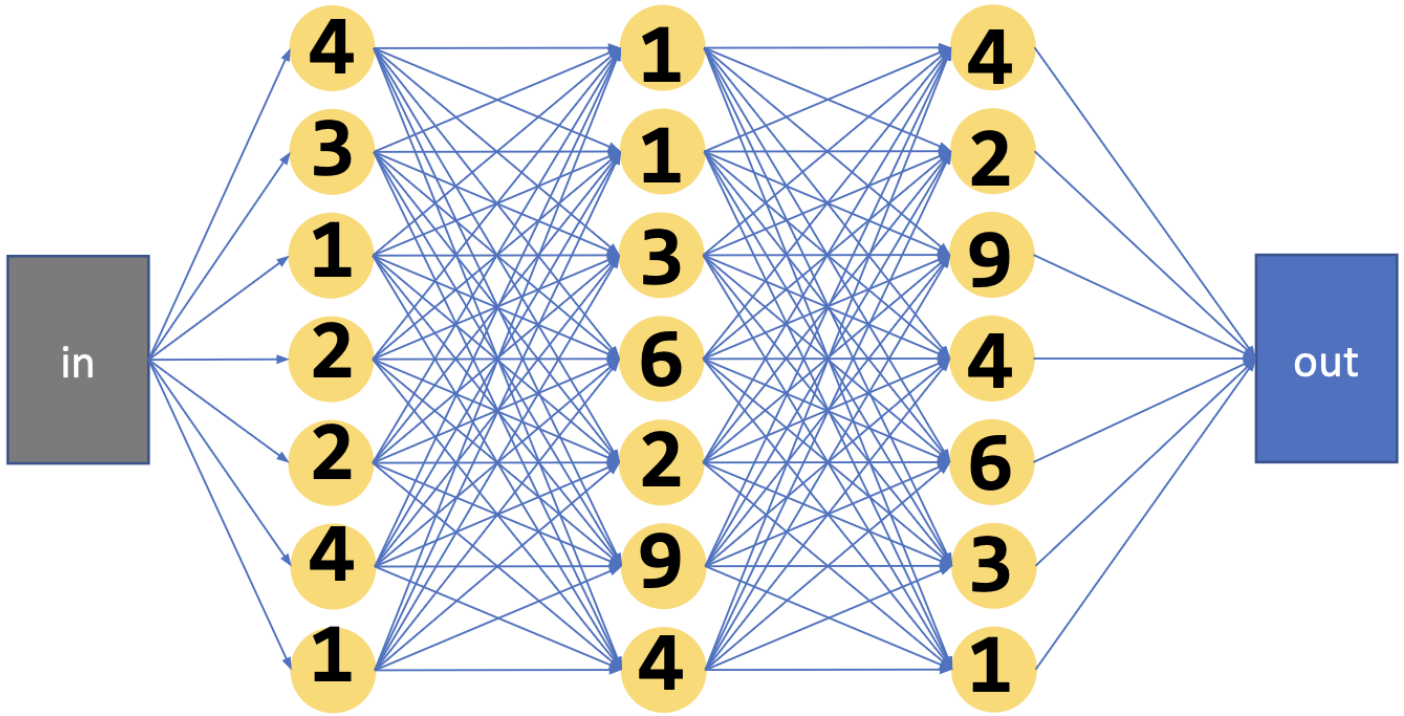
A person is climbing a wall using holds

- What if, I make a rule that she can climb
- … only using certain holds (say just green ones!)
- If she can learn to do this using fewer holds…
- …she'll definitely be able to do it with ALL the holds
- (learn better climbing techniques in the process)

Dropout ~= using only a certain holds instead of ALL the holds

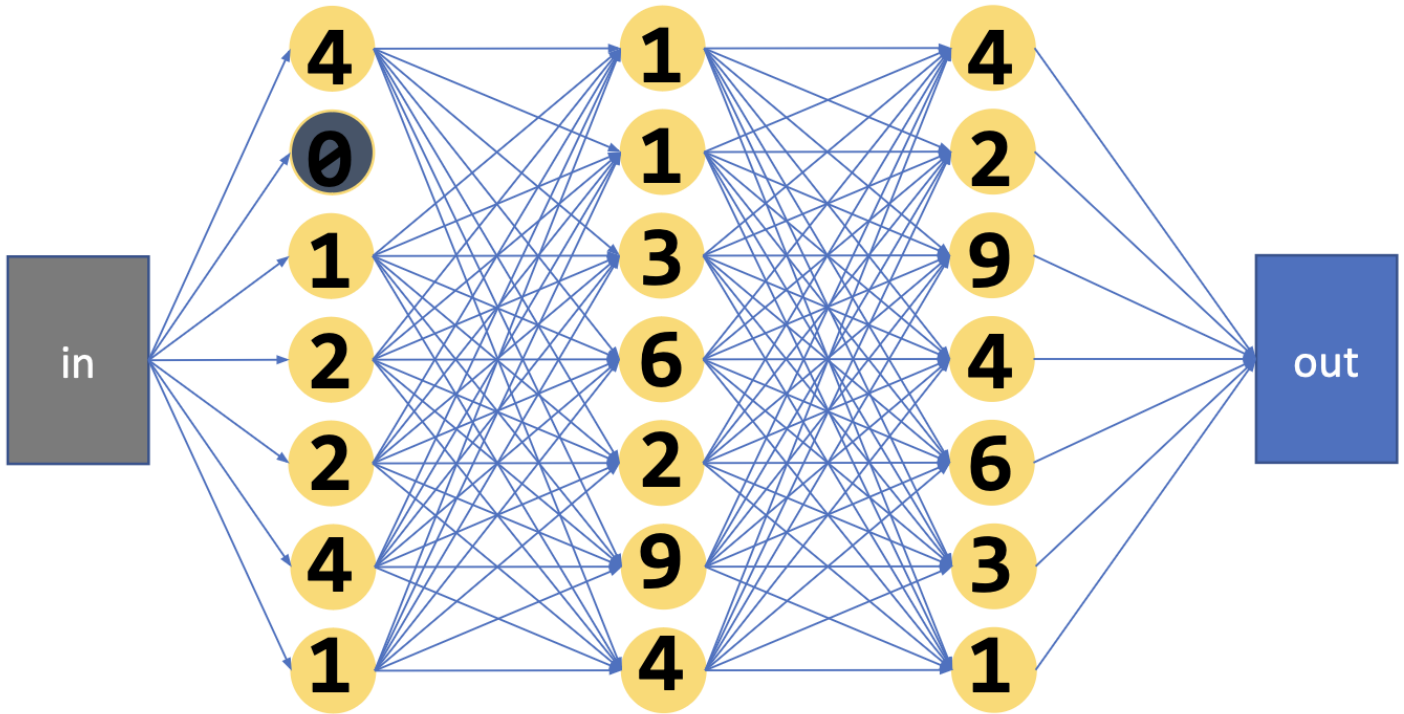
Image source https://www.istockphoto.com/illustrations/indoor-climbing

Dropout - what?



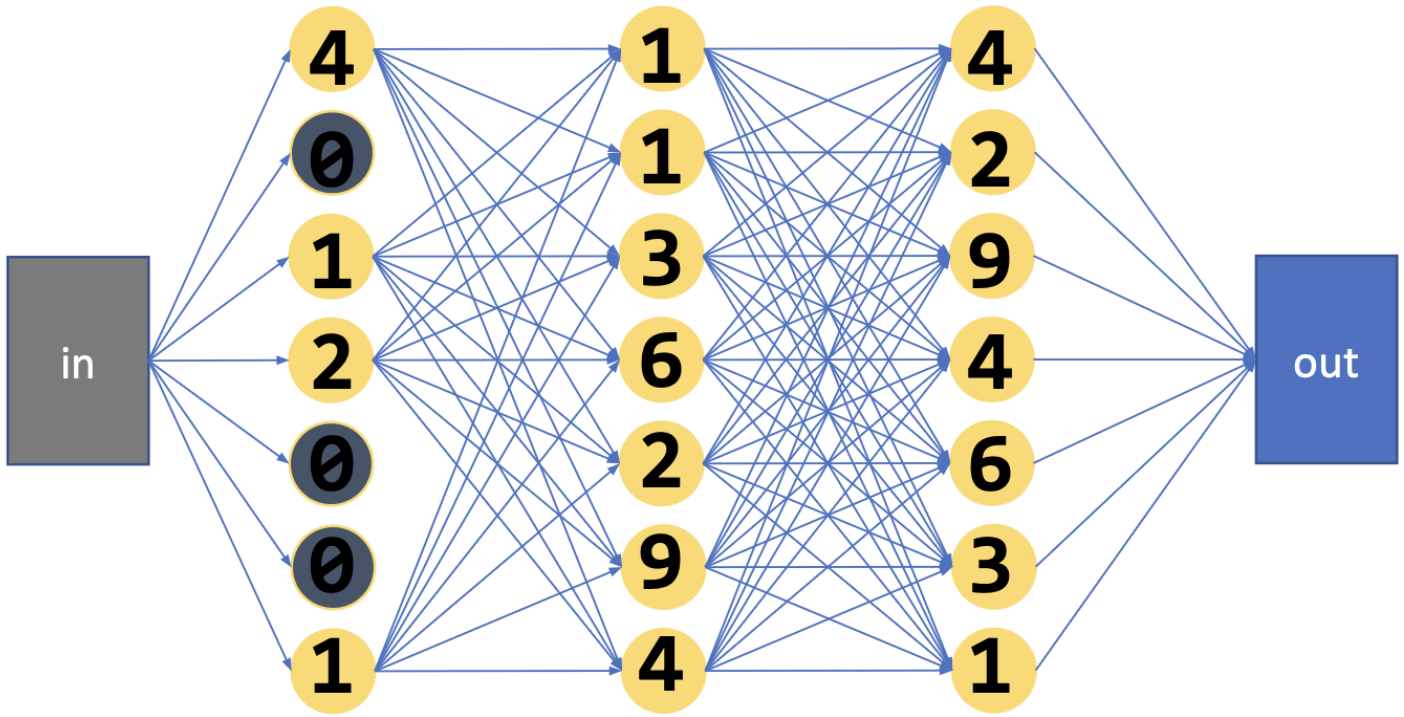
Typical NN: the output of every node in every layer is used in the next layer of the network

Dropout - what?



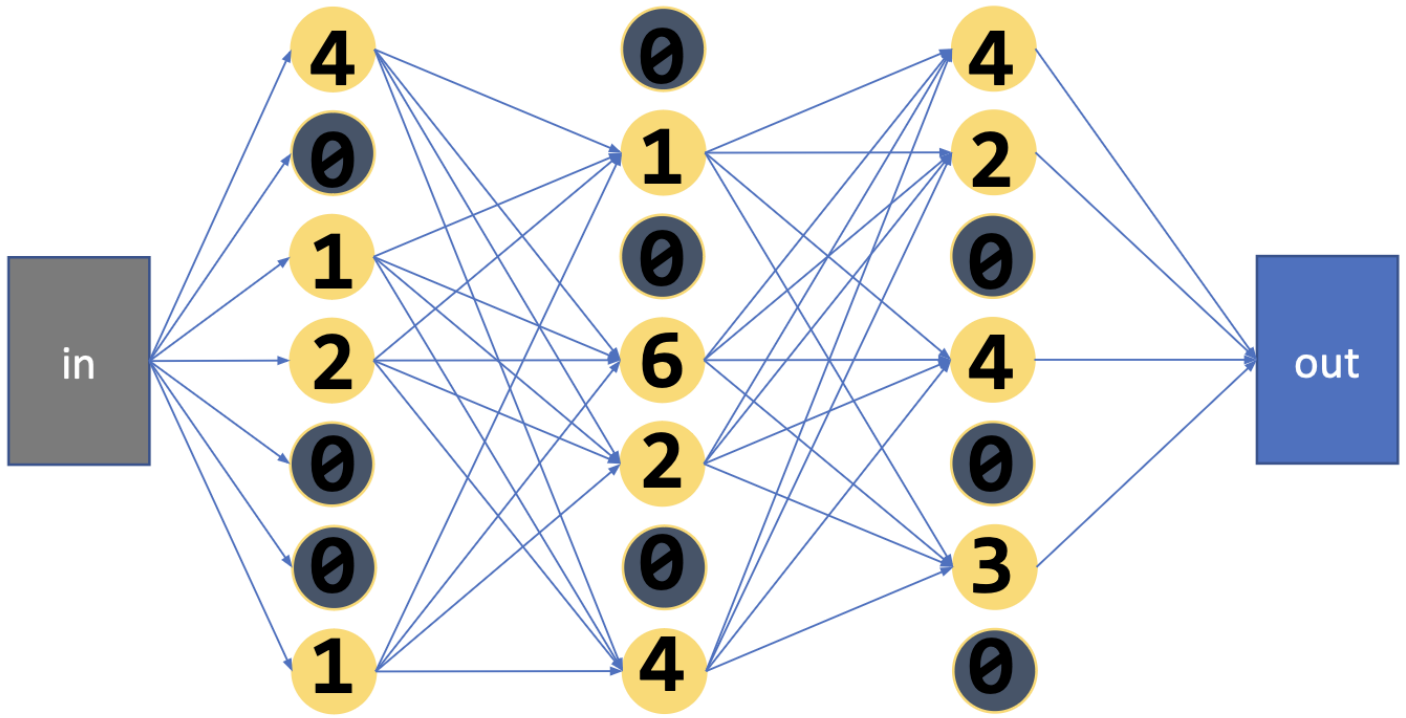
Dropout: *in a single training pass*, the output of randomly selected nodes from each layer will “drop out”, i.e. be set to 0

Dropout - what?



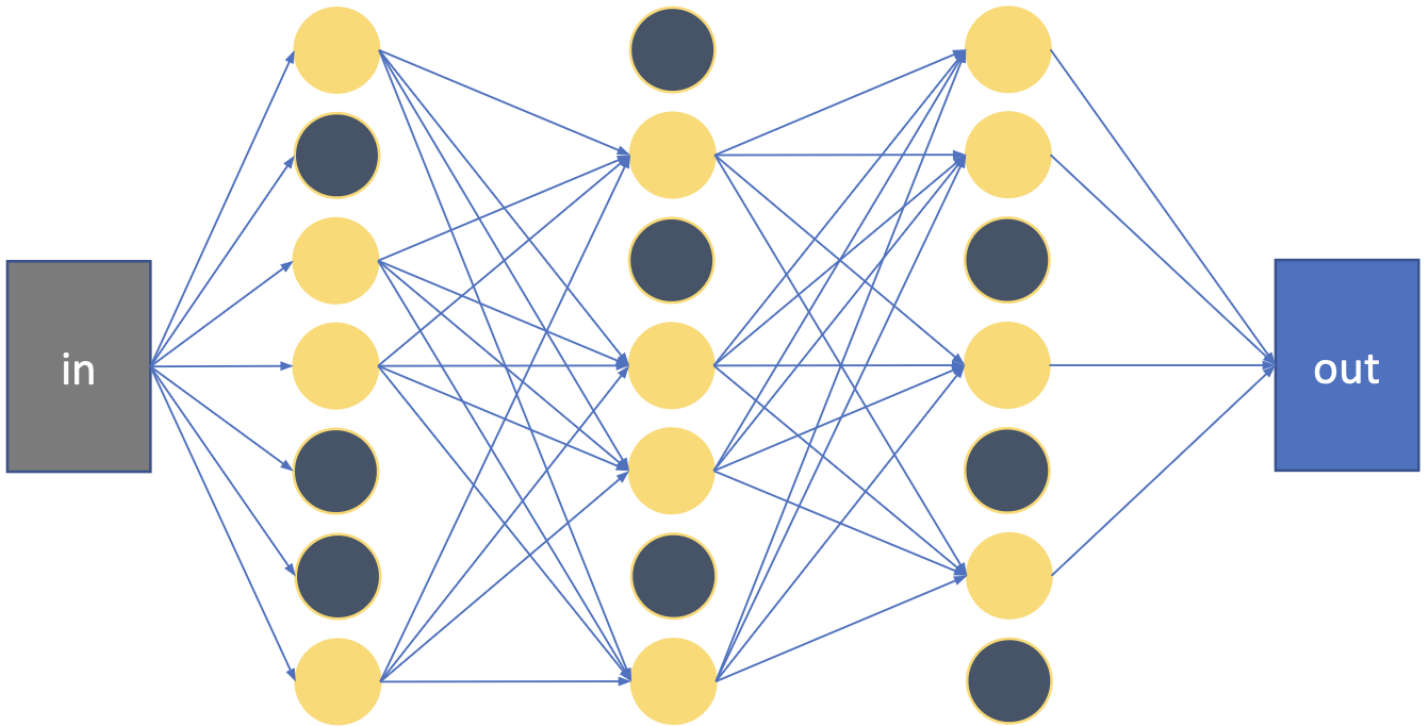
Dropout: *in a single training pass*, the output of randomly selected nodes from each layer will "drop out", i.e. be set to 0

Dropout - what?



Not just limited to the input layer: can do this to *any* layer of the network

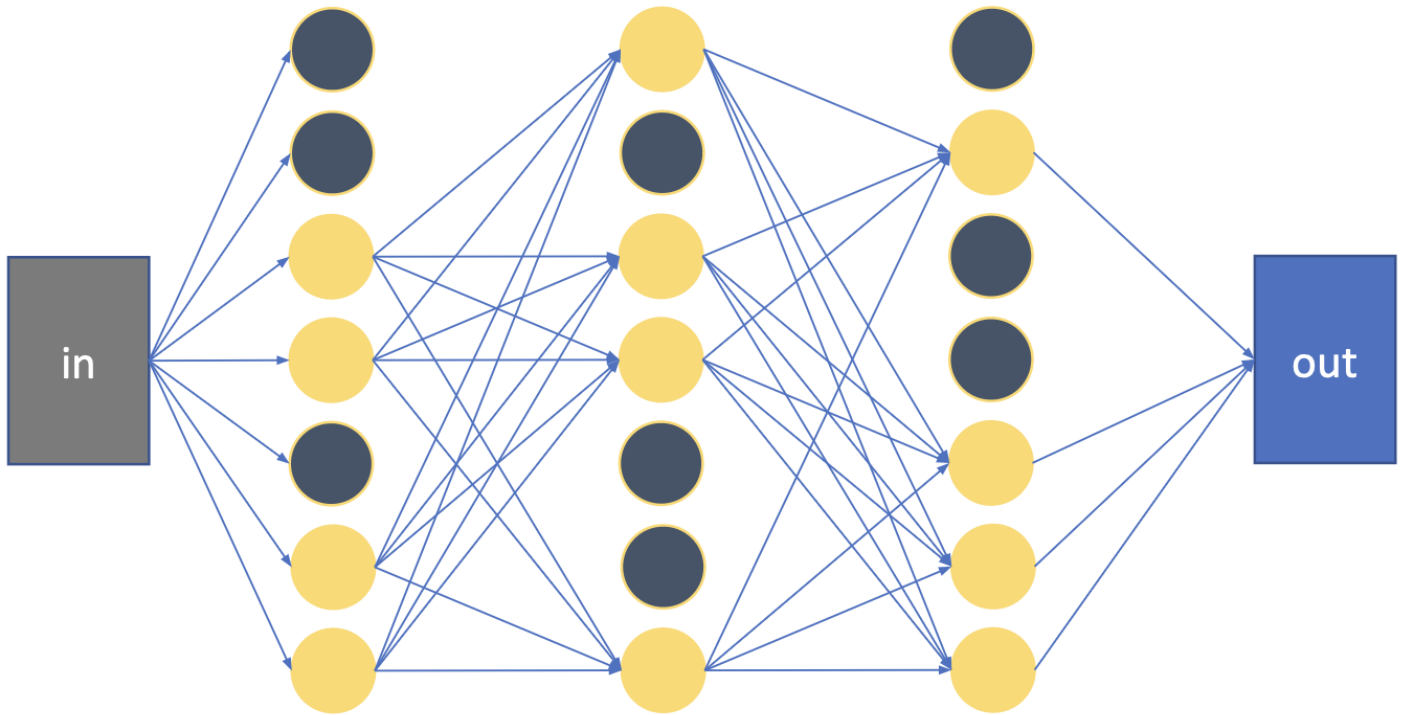
Dropout - what?



The nodes that drop out will be different each pass

(re-randomly selected)

Dropout - what?



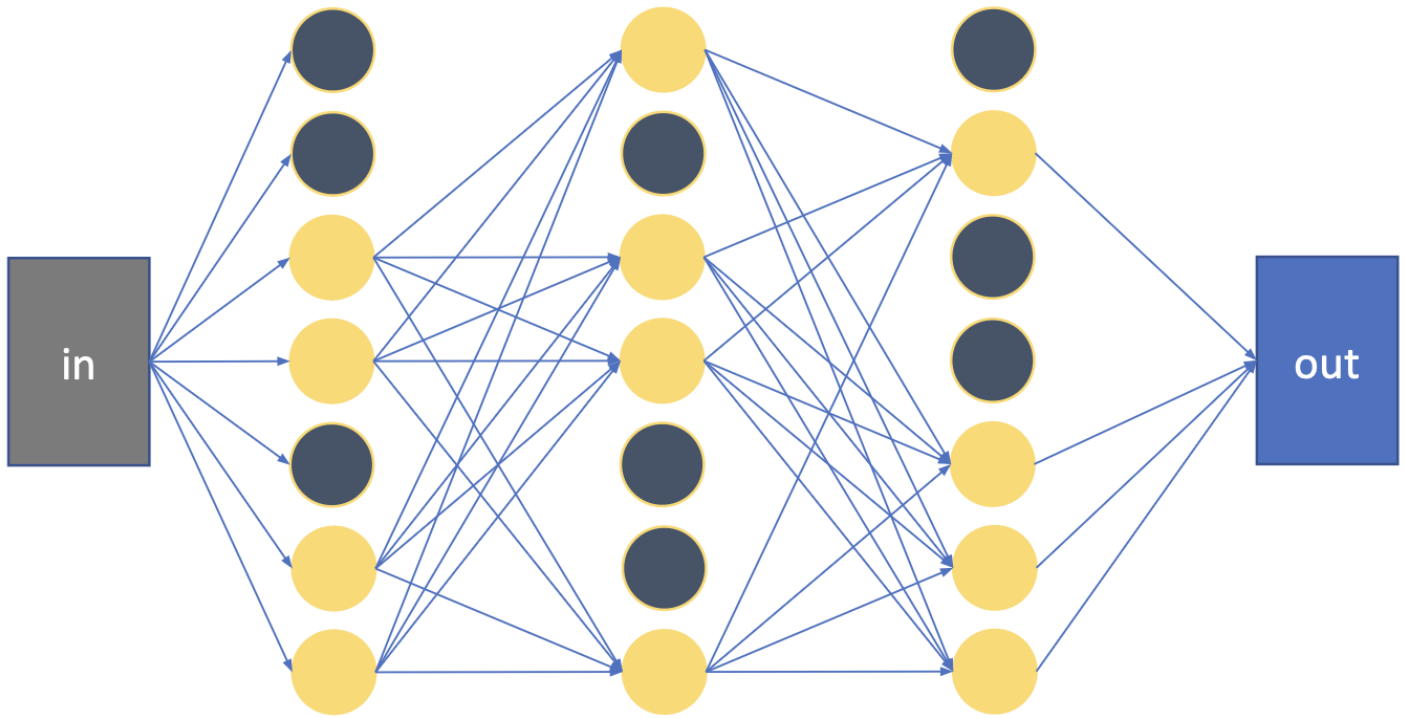
The nodes that drop out will be different each pass

(re-randomly selected)

Dropout - why?

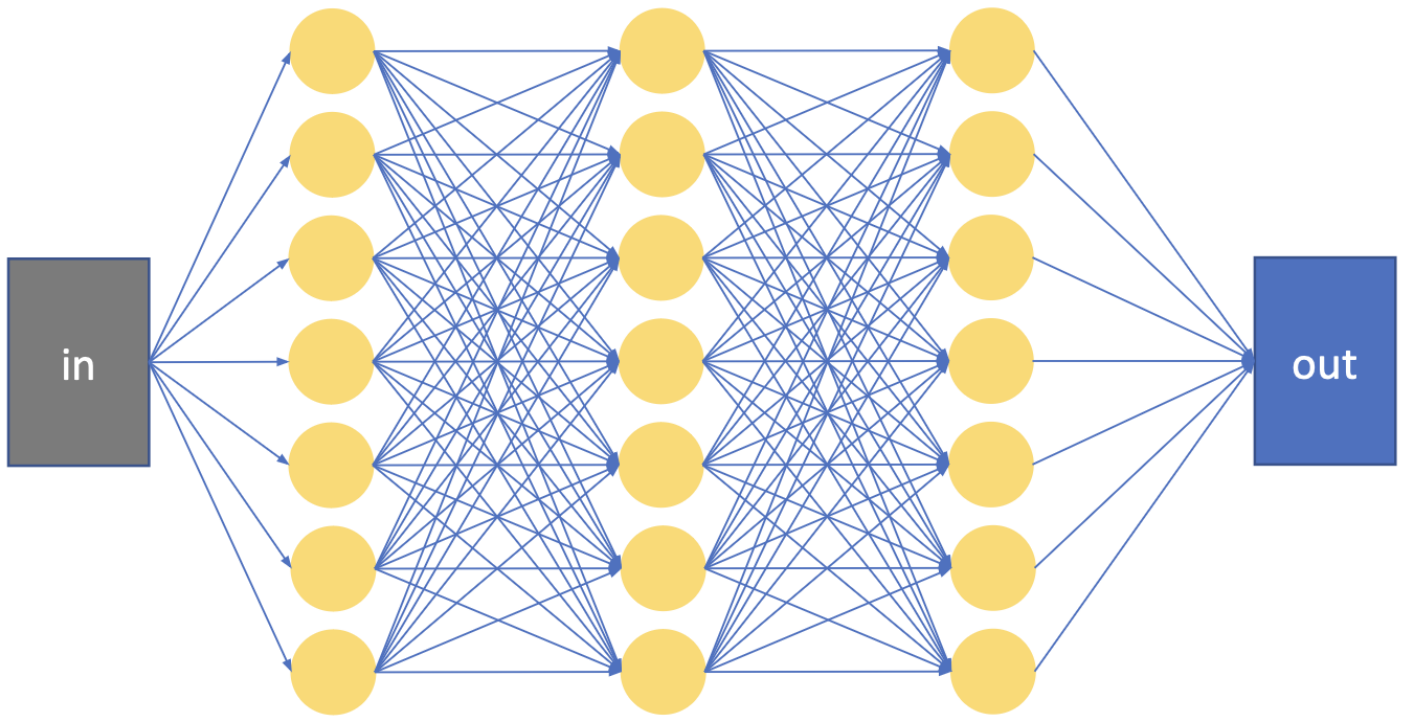
- Sort of looks like data augmentation, if you squint hard enough
  - Augmenting the data by randomly dropping out parts of it
- Over multiple passes through the net (i.e. during training over many epochs):
  - Randomly dropping neurons “forces” each neuron to learn a non-trivial weight
  - The network can’t learn to rely on spurious correlations (i.e. meaningless patterns), because they randomly might not be present

Dropout: Implications for test time



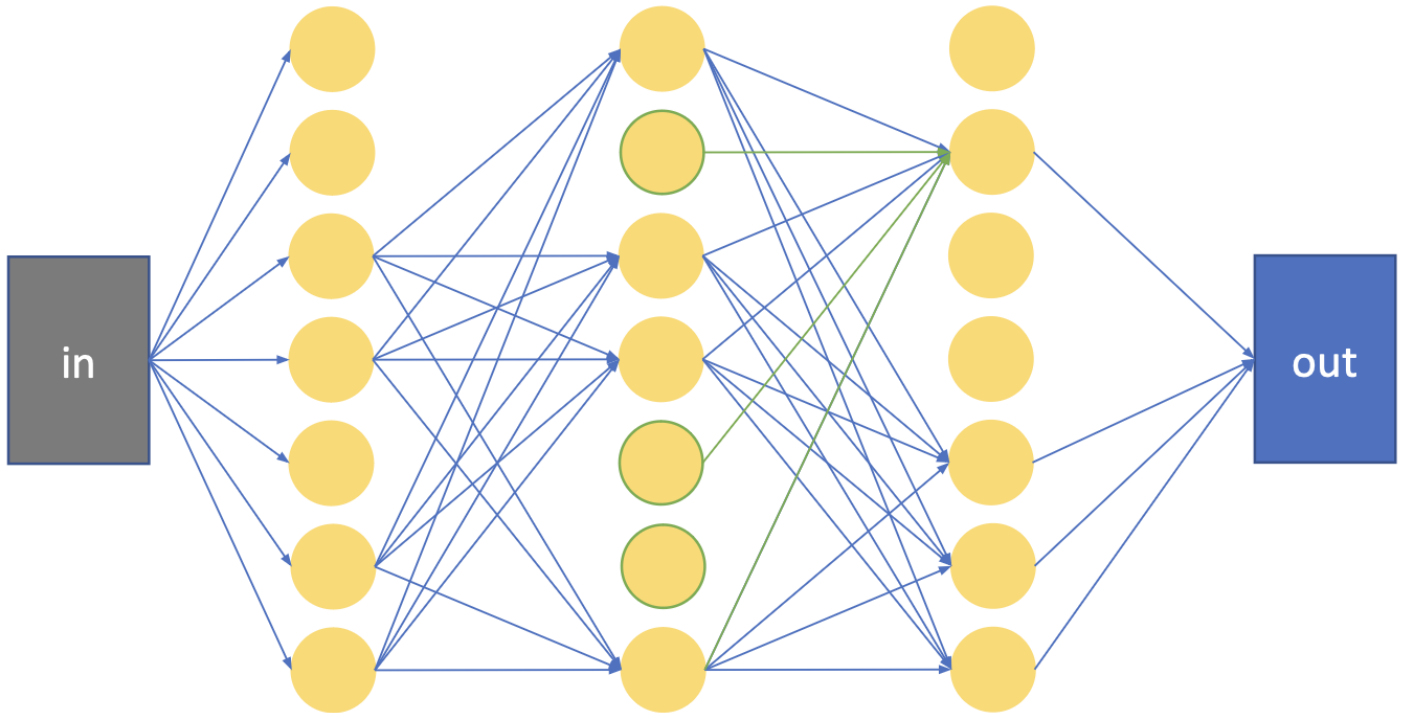
- During testing, we stop dropping out and use all of the neurons again

Dropout: Implications for test time



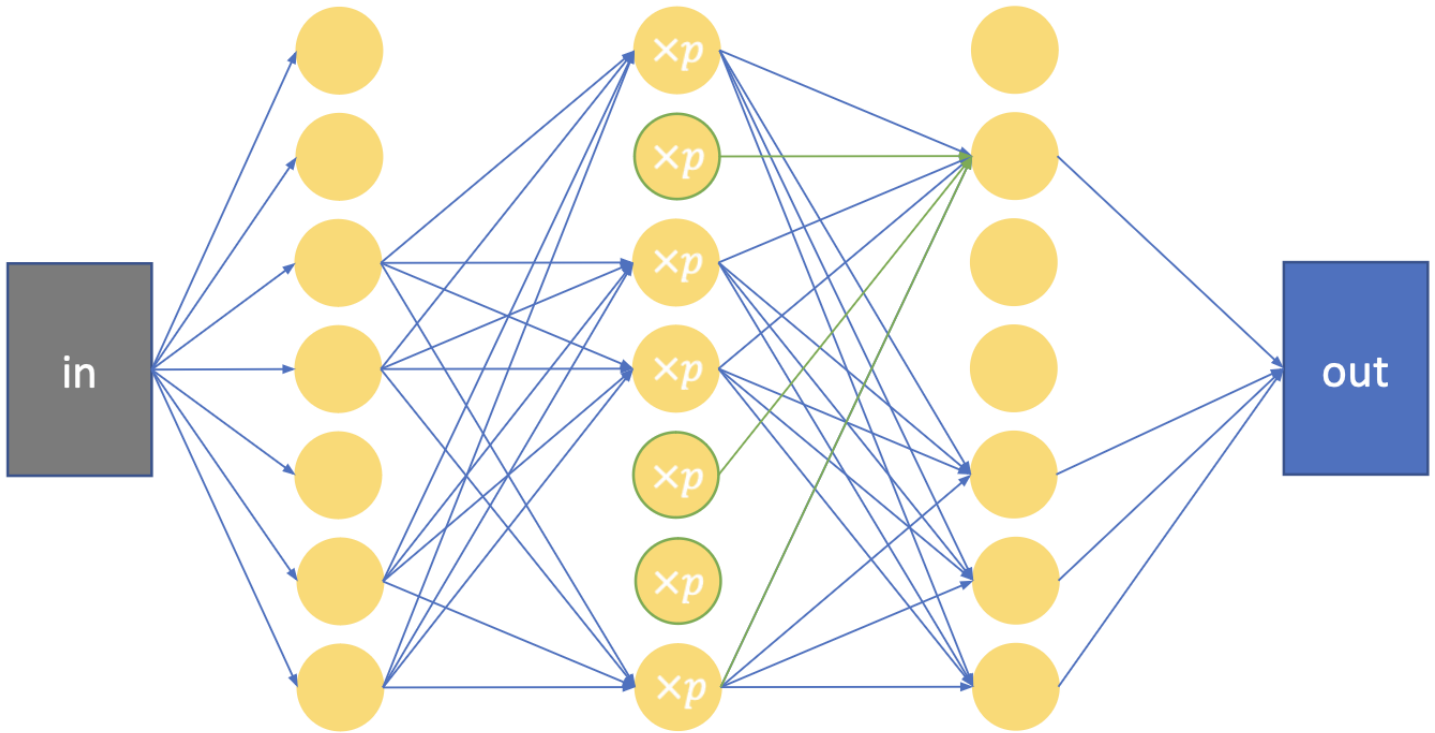
- During testing, we stop dropping out and use all of the neurons again

Dropout: Implications for test time



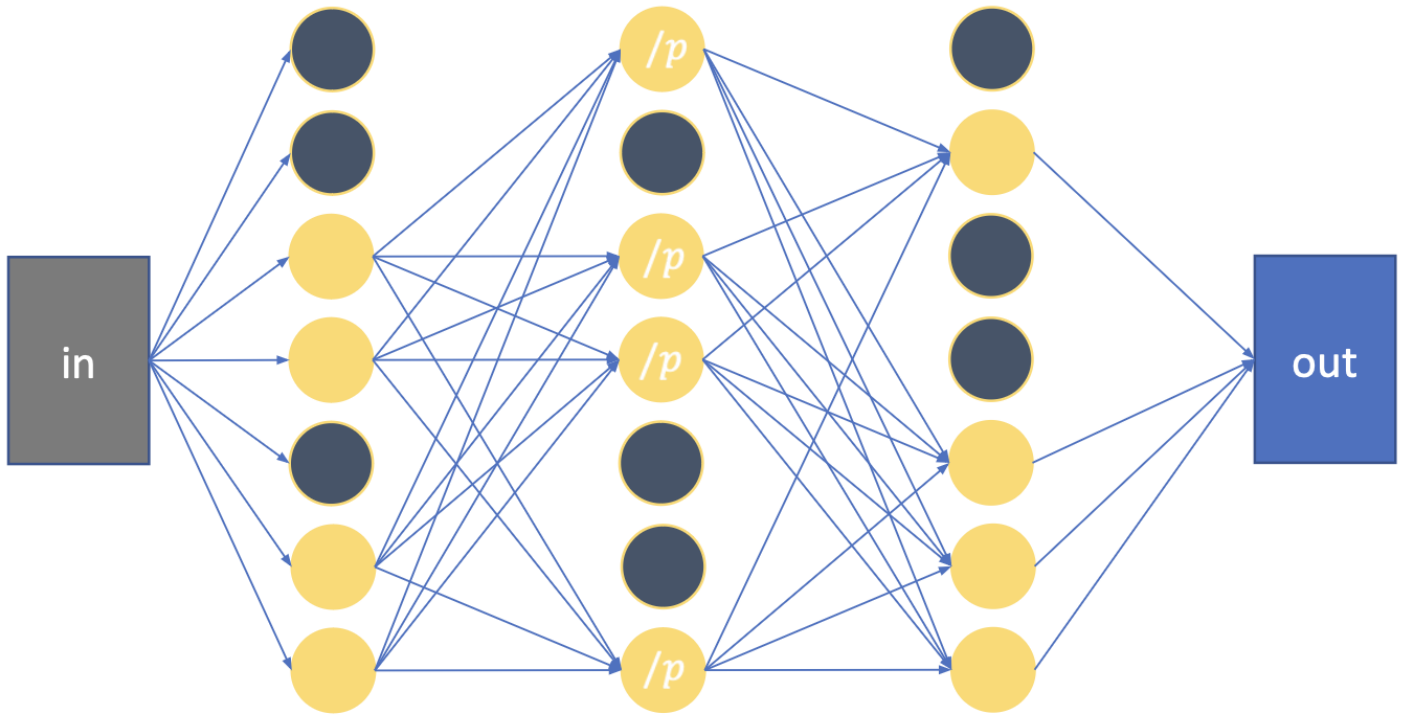
- During testing, we stop dropping out and use all of the neurons again
- If a layer keeps a fraction p of its neurons during training, then when we use all the neurons at test time, the next layer will get a bigger input than expected...
- ***What do we do!?***

Dropout: Implications for test time



- **Solution 1:**
  Multiply the values of all neurons by p, so that the expected magnitude of the sum of neurons is the same

Dropout: Implications for test time



- **Solution 1:**
  Multiply the values of all neurons by p, so that the expected magnitude of the sum of neurons is the same
- **Solution 2:**
  At training time, divide the values of the kept neurons by p

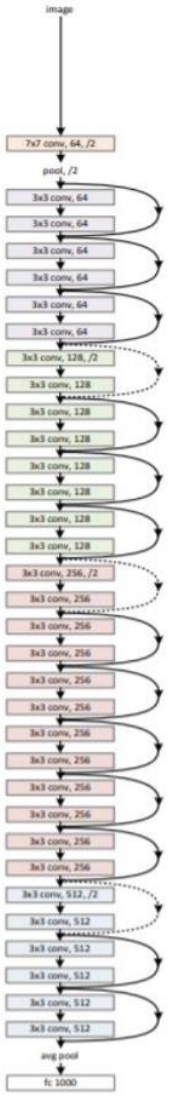
Dropout - implementation

Any questions?

- Handy keras layer!

- **`tf.keras.layers.Dropout(rate)`**
  - Hyperparameter **`rate`** between [0, 1]: the rate at which the outputs of the previous layer are dropped
  - **`Rate = 0.5`**: drop half, keep half
  - **`Rate = 0.25`**: drop ¼, keep ¾

Recap



Residual blocks prevent vanishing gradients

BatchNorm helps to stabilize training as networks get deep

Regularization is a somewhat automated way of preventing overfitting

The Real World

- Hey, your models work great!
- Let’s deploy them to the real world!
- What could go wrong?

Papernot et al. Practical Black-Box Attacks against Machine Learning

The Real World

- Hey, your models work great!
- Let’s deploy them to the real world!
- What could go wrong?

Stop Sign: 99%

Papernot et al. Practical Black-Box Attacks against Machine Learning

The Real World

- Hey, your models work great!
- Let's deploy them to the real world!
- What could go wrong?

Stop Sign: 99%

Papernot et al. Practical Black-Box Attacks against Machine Learning

The Real World

- Hey, your models work great!
- Let's deploy them to the real world!
- What could go wrong?

Stop Sign: 99%

Yield Sign: 99%

Papernot et al. Practical Black-Box Attacks against Machine Learning

The Real World

- Hey, your models work great!
- Let's deploy them to the real world!
- What could go wrong?

Stop Sign: 99%

Yield Sign: 99%

Papernot et al. Practical Black-Box Attacks against Machine Learning

Adversarial Learning

- Can we (or adversaries) break our deep learning models
- Adversarial Attack: Can we add a small amount of noise to an input that results in a misclassification?
- Data Poisoning: Can we insert data in the training dataset that corrupts the model's training?

Objective

- In Deep Learning, our objective is to minimize loss
- What do you think the objective of our adversary is?

Objective

- In Deep Learning, our objective is to minimize loss
- What do you think the objective of our adversary is?

Maximize (Test) Loss

Objective

- In Deep Learning, our objective is to minimize loss
- What do you think the objective of our adversary is?

Maximize (Test) Loss

Want to follow direction of gradient (Gradient Ascent)

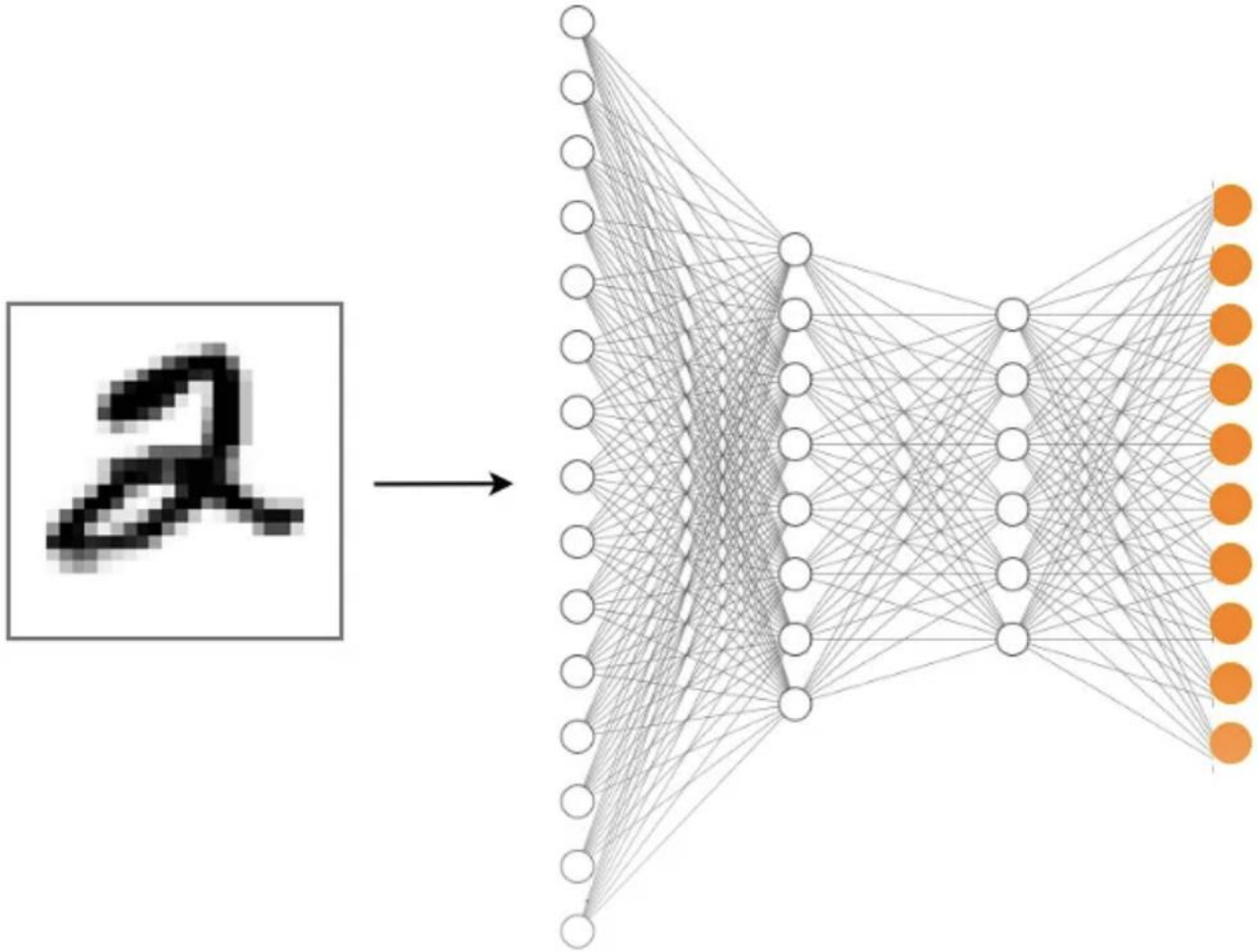
Objective

- In Deep Learning, our objective is to minimize loss
- What do you think the objective of our adversary is?
- What does our adversary have control of?
  - Input data?
  - Training Data?
  - Our model? (Uh oh)

Objective

- In Deep Learning, our objective is to minimize loss
- What do you think the objective of our adversary is?
- What does our adversary have control of?
  - Input data?
  - Training Data?
  - Our model? (Uh oh)

Most Commonly Studied



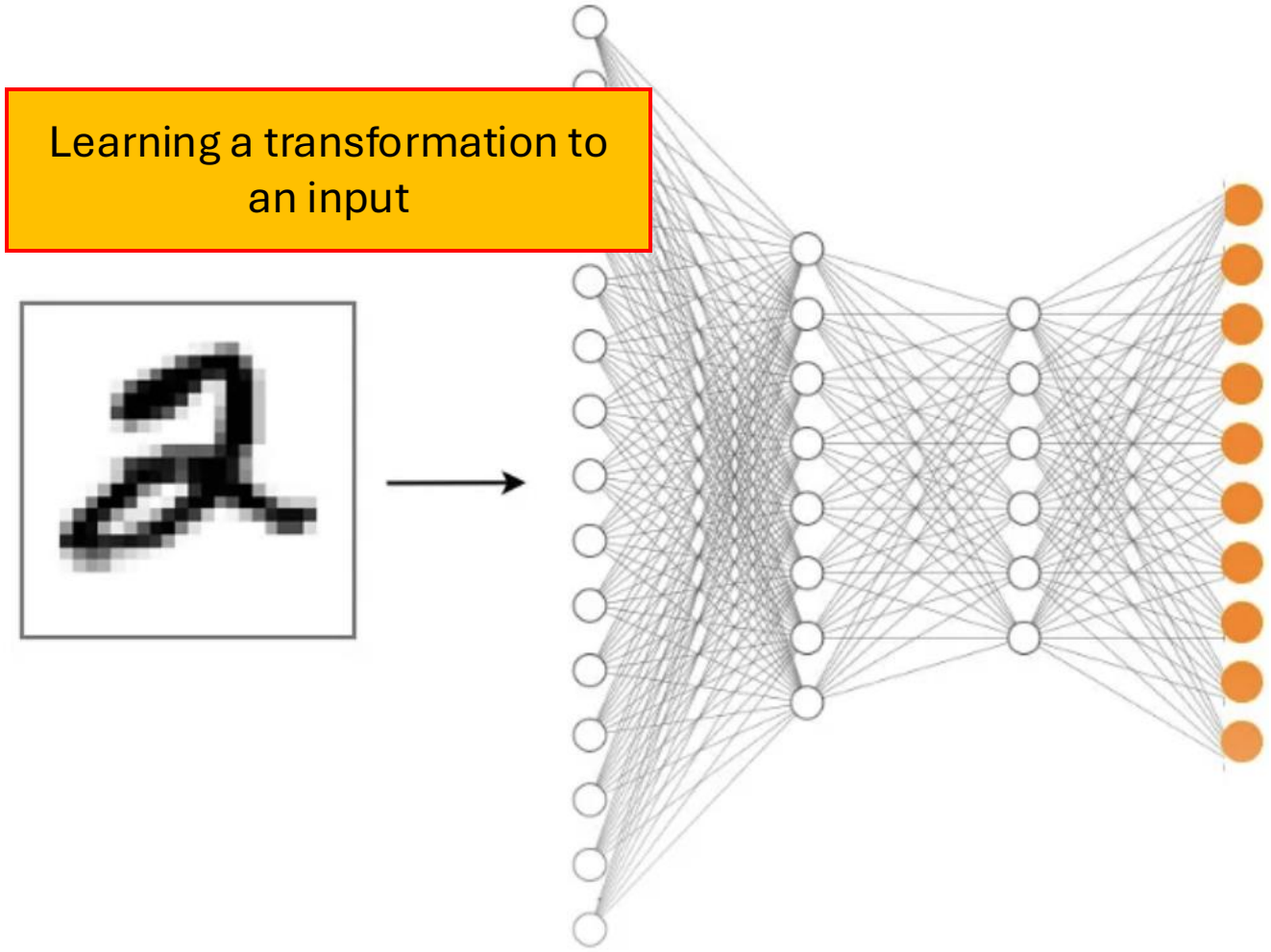
Normal Training:

- Compute gradients wrt weights and biases
- Update via gradient descent

Adversarial Example:

- Compute gradients wrt input
- Update **input** via gradient **ascent**

Learning a transformation to an input



Normal Training:

- Compute gradients wrt weights and biases
- Update via gradient descent

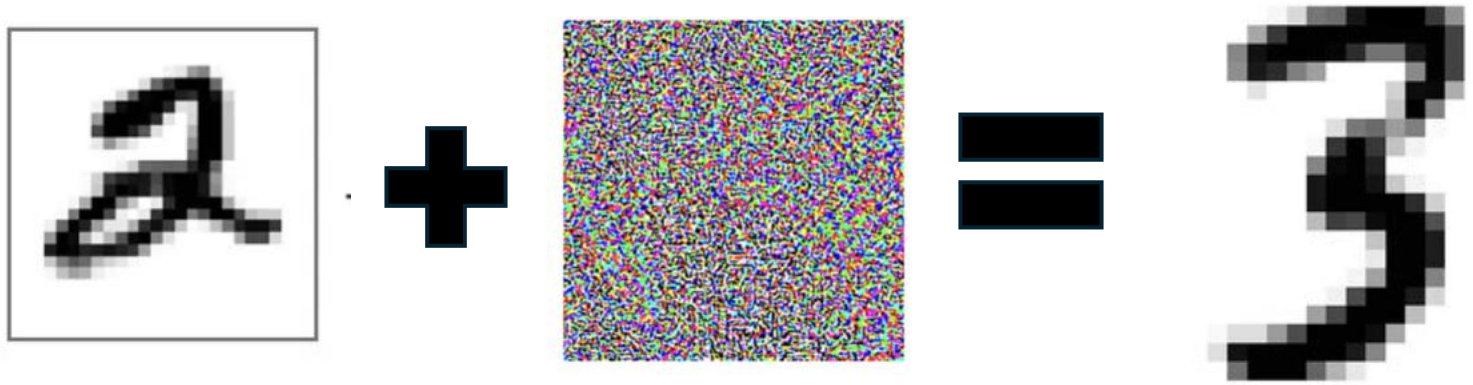
Adversarial Example:

- Compute gradients wrt input
- Update input via gradient ascent

Attack Model

We do not expect to be able to withstand an attacker with unlimited power.

If attackers can add unlimited noise, they can just change the image entirely.

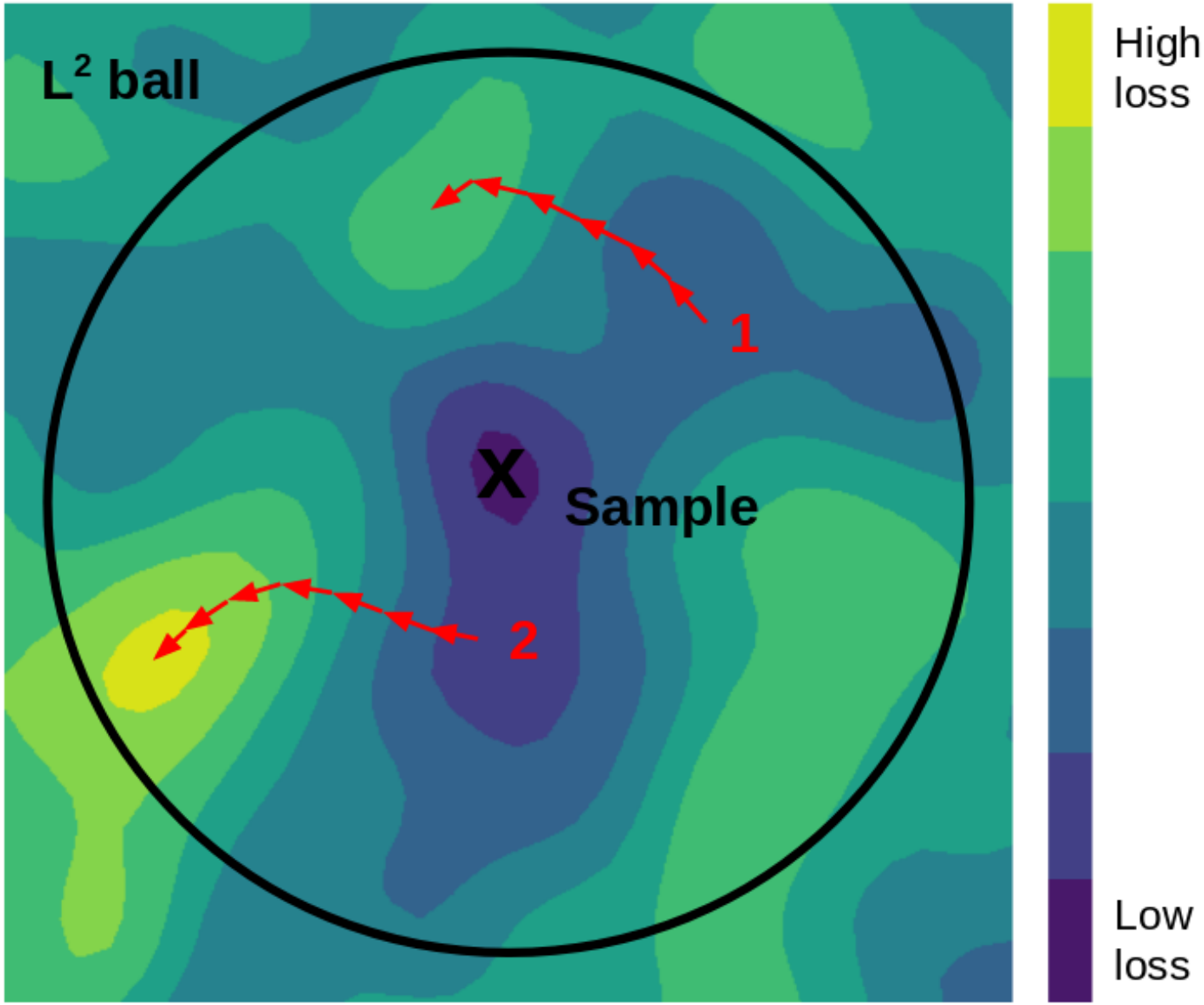


Threat Model

- We limit the power of the attacker
- Attacks must fall within some L^p-Ball of radius r
  - L^1-Ball: Sum of noise must be below r
  - L^2-Ball: Square root (sum of squared noise for each pixel) must be below r
  - L^∞-Ball: Largest individual value of attack noise must be below r

Gradient Ascent around an input sample

What happens if we hit the constraint and can't keep following the gradient?

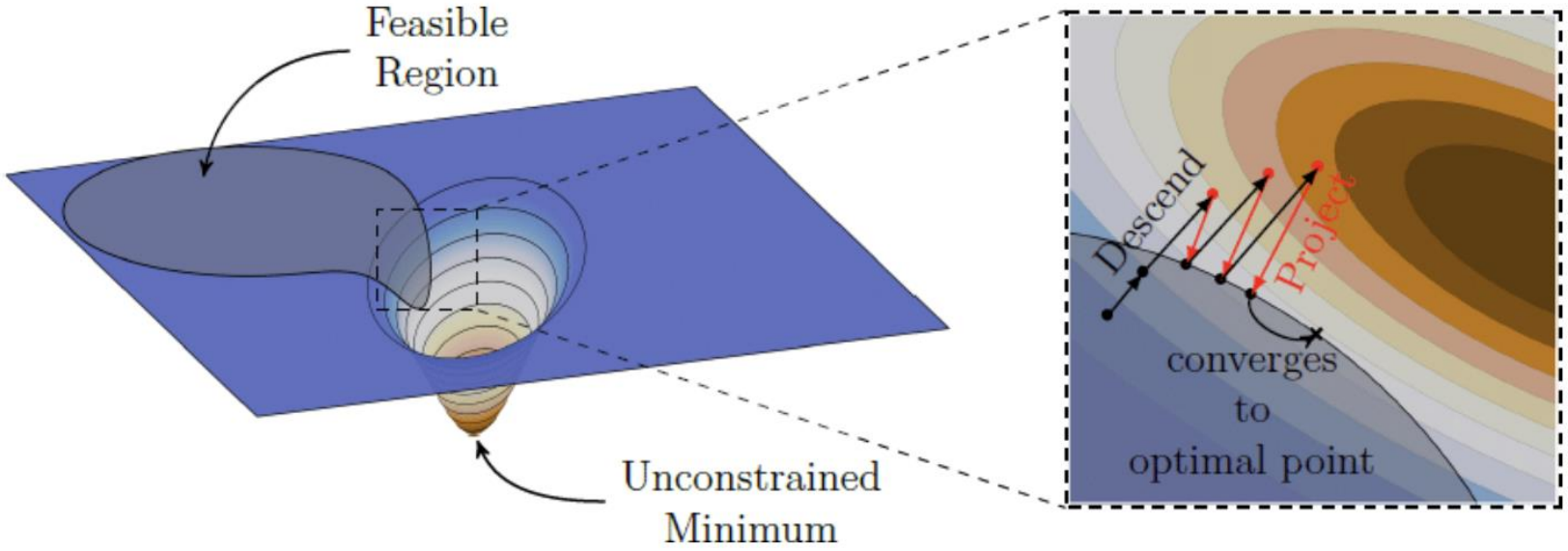


https://medium.com/towards-data-science/know-your-enemy-7f7c5038bdf3

Constrained Optimization

- Projected Gradient Ascent (PGA):
  - Run Gradient Ascent
  - If noise goes outside of constraint set, project back into constraint set

(Picture is for minimization)



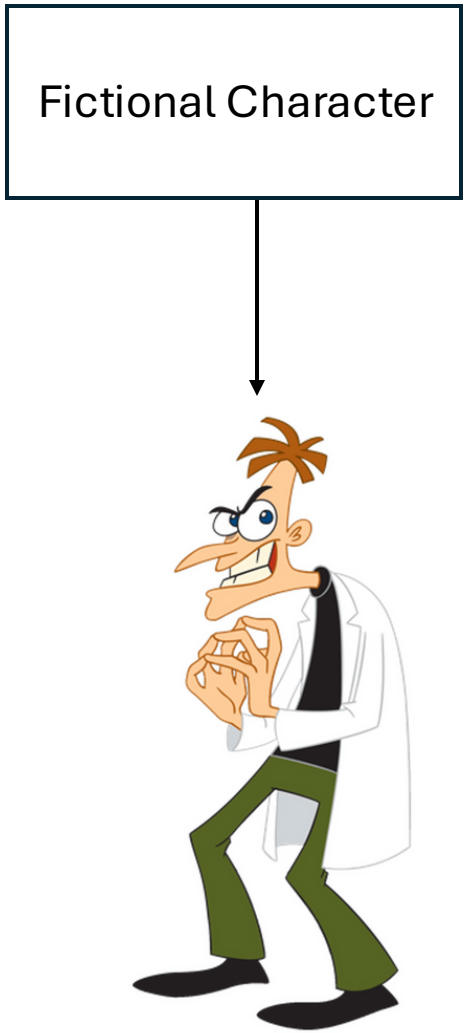
https://www.researchgate.net/publication/358122337_Adaptive_Model_for_Magnetic_Particle_Mapping_Using_Magnetoelectric_Sensors

How big of a problem is this?

- Most models will never be under threat from adversarial attacks
- But doesn’t this tell us something new about our models?

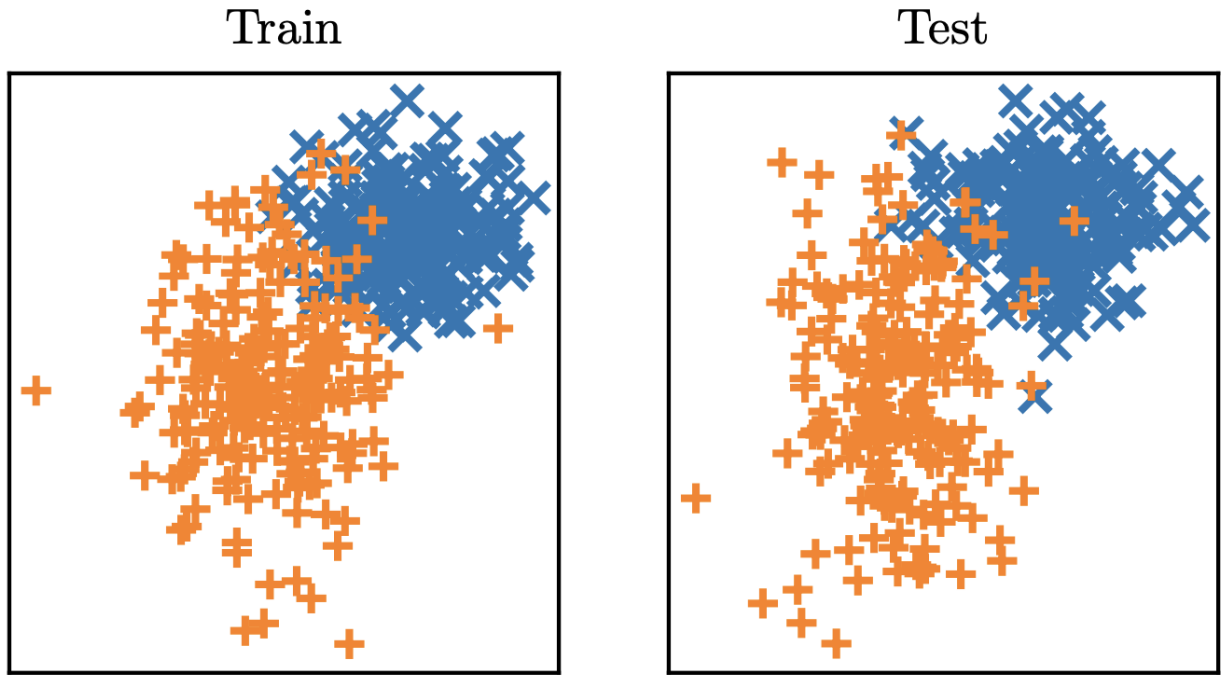
How big of a problem is this?

- Most models will never be under threat from adversarial attacks
- But doesn't this tell us something new about our models?



Why Adversarial Attacks Work

I.I.D. Machine Learning



I: Independent

I: Identically

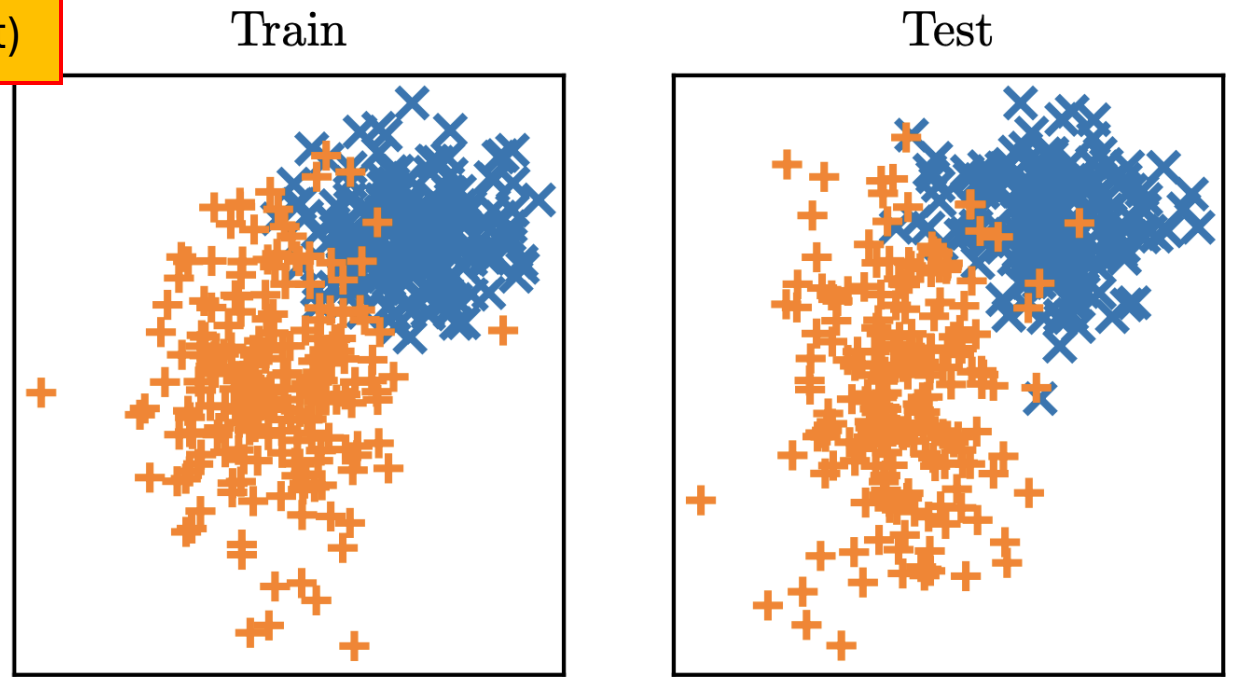
D: Distributed

All train and test examples drawn independently from same distribution

Why Adversarial Attacks Work

We assume our datasets are IID (Train set looks like validation set looks like test set)

I.I.D. Machine Learning



I: Independent

I: Identically

D: Distributed

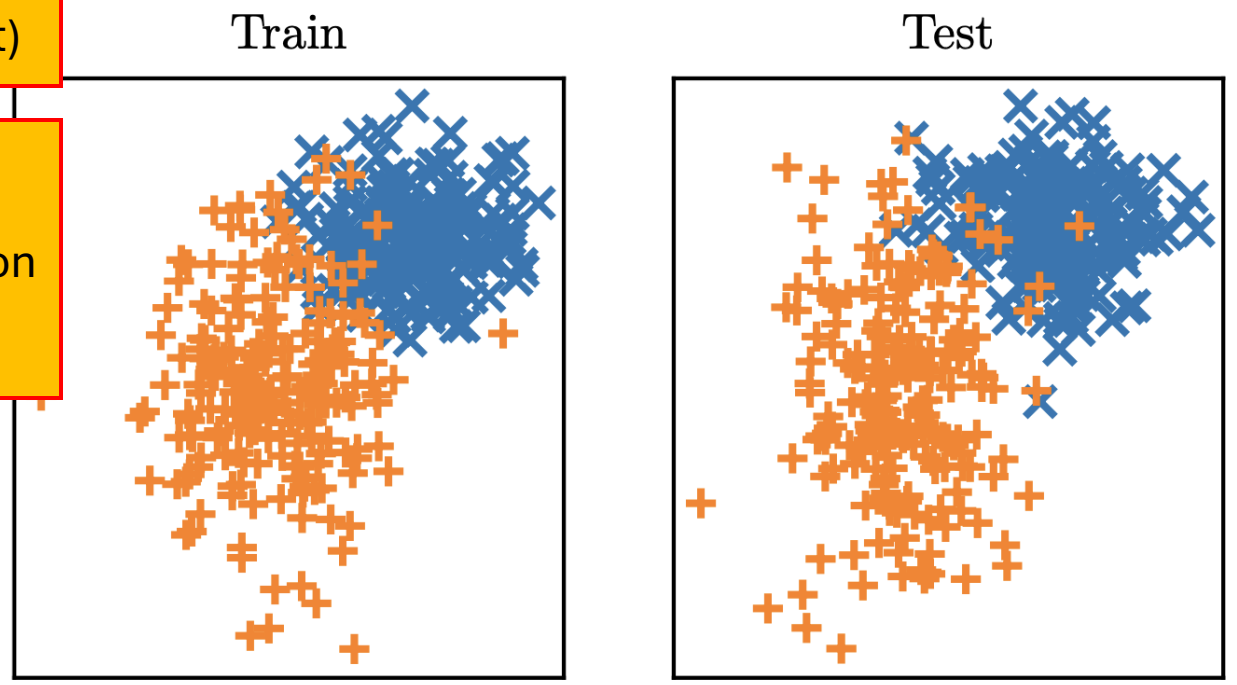
All train and test examples drawn independently from same distribution

Why Adversarial Attacks Work

We assume our datasets are IID (Train set looks like validation set looks like test set)

Adversarial attacks change the distribution of the test set

I.I.D. Machine Learning



I: Independent

I: Identically

D: Distributed

All train and test examples drawn independently from same distribution

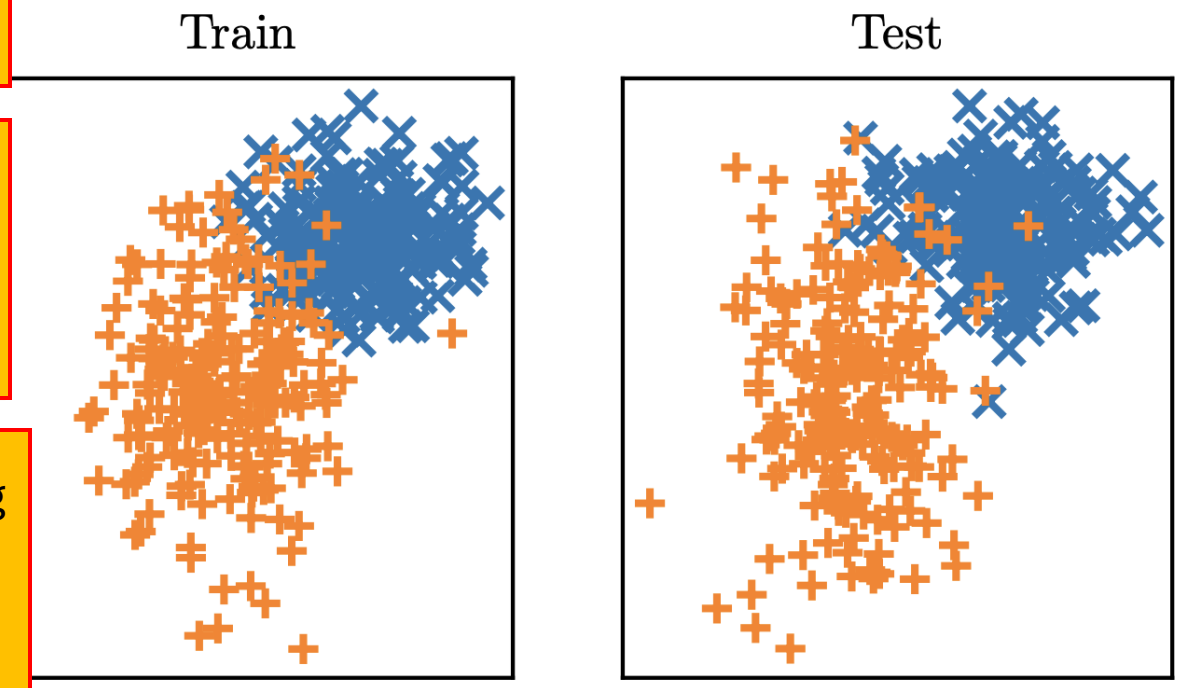
Why Adversarial Attacks Work

We assume our datasets are IID (Train set looks like validation set looks like test set)

Adversarial attacks change the distribution of the test set

Performance on training set/validation set is no longer indicative of test performance

I.I.D. Machine Learning



I: Independent

I: Identically

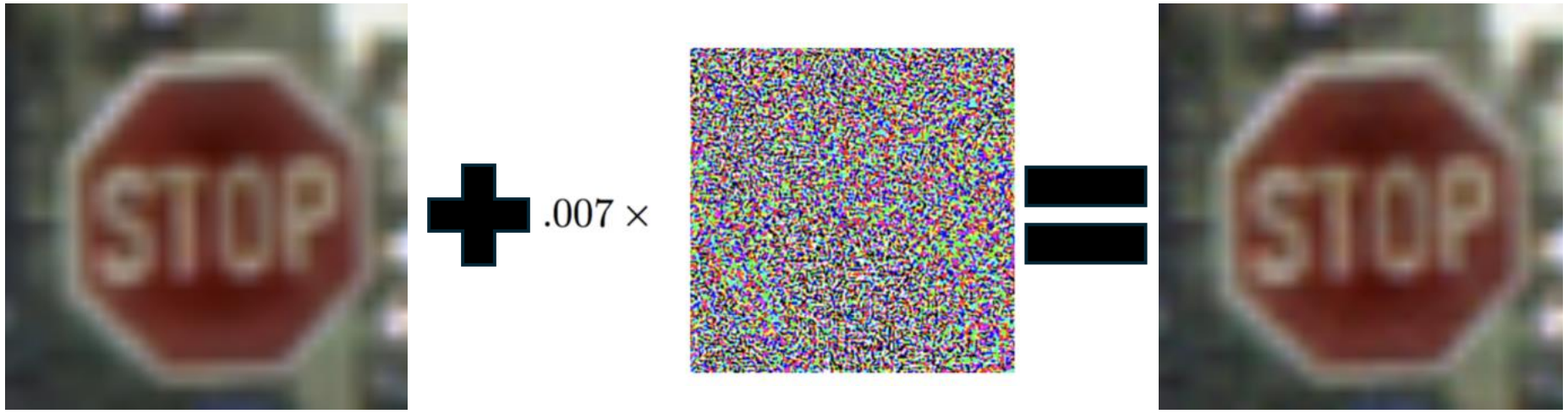
D: Distributed

All train and test examples drawn independently from same distribution

What did we learn in the first place?

If such small noise can change the outputs of our network, it clearly is not making decisions in the way that humans do.

It isn’t always making decisions about stop signs based on color, shape, or text…

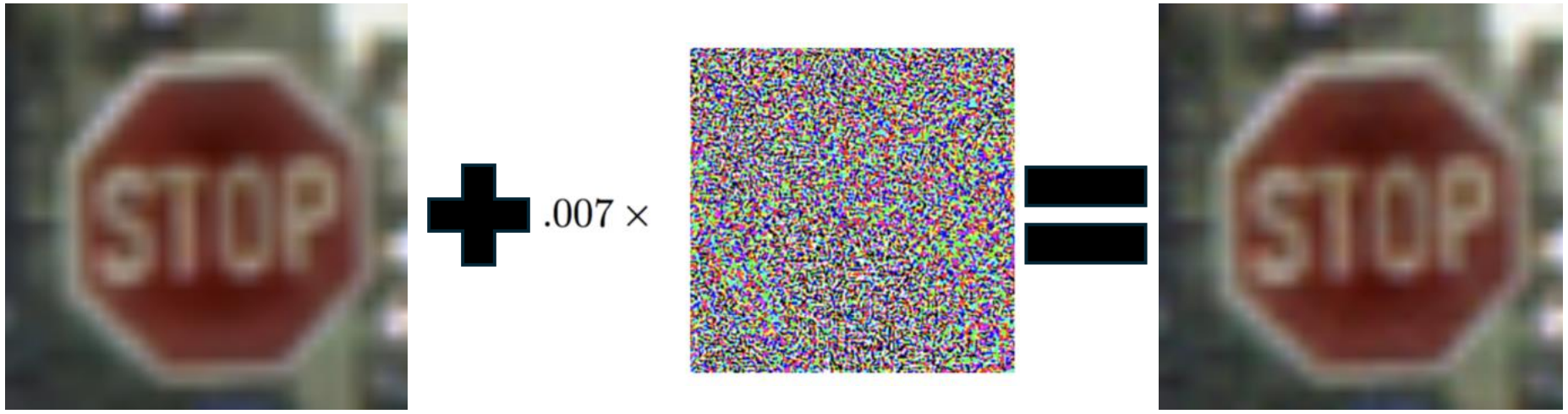


Stop Sign

Yield

What did we learn in the first place?

Deep learning learns the “easiest” good representation, which can be very brittle and break under small perturbations



Stop Sign

Yield

Defenses

Defenses

How can we make more robust models?

Defenses

How can we make more robust models?

- Ensembles
  - Train multiple different models average results
  - (Can make models more robust, but not resistant to adversarial attacks)

Defenses

How can we make more robust models?

- Ensembles
  - Train multiple different models average results
  - (Can make models more robust, but not resistant to adversarial attacks)
- Data Augmentation?
  - Just add lots of random noise to inputs while training?
  - Add in Adversarial Examples while training?

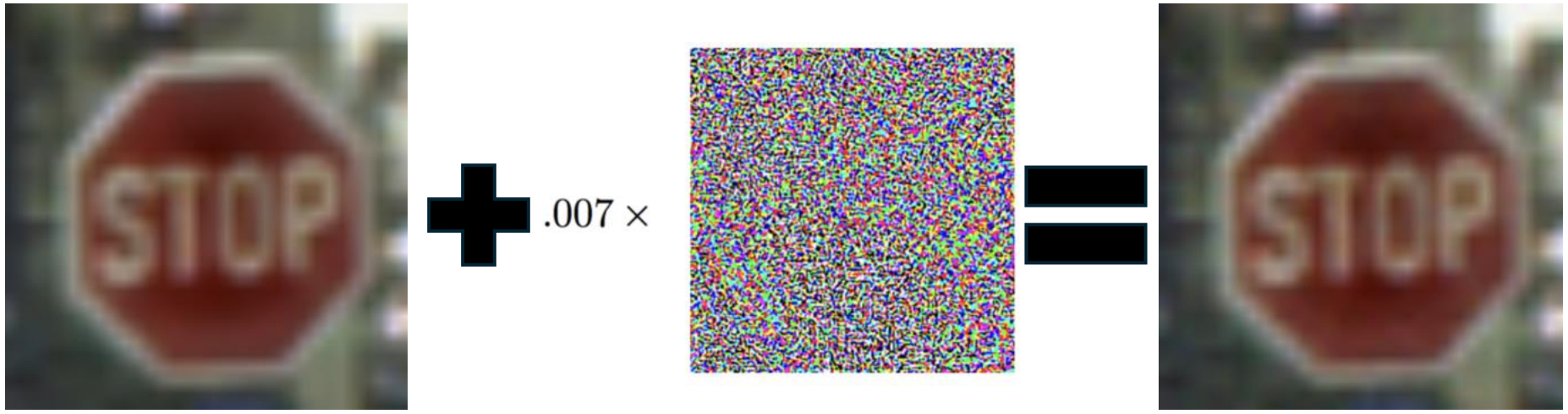
Defenses

How can we make more robust models?

- Ensembles
  - Train multiple different models average results
  - (Can make models more robust, but not resistant to adversarial attacks)
- Data Augmentation?
  - Just add lots of random noise to inputs while training?
  - Add in Adversarial Examples while training?
- Provably Robust Networks
  - Lipschitz Continuity!

Attack Transfer

Adversarial Examples *tend to fool* other networks as well



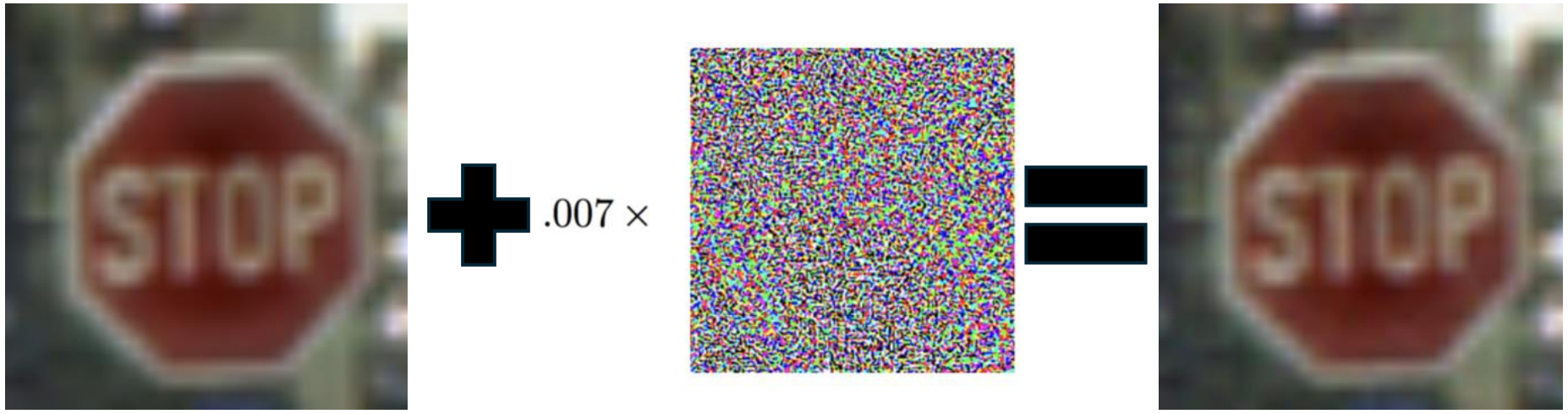
Stop Sign

Yield

Attack Transfer

Adversarial Examples *tend to fool* other networks as well

If this attack was made using ResNet, it would likely work against VGG



Stop Sign

Yield

Attack Transfer

- This also gives us another tool for adversarial attacks
- Suppose the model we are trying to break is not public (i.e., you can’t find the gradients)
- Black-box attack:
  - Train a “surrogate” model on the same dataset
  - Construct an adversarial example that works against your surrogate model
  - Send attack to original model

Data Augmentation

If breaking the IID assumption caused our issues, can we just change the distribution of the training set?

Data Augmentation

What if we just add lots of images with small amounts of random noise to our training data?

Data Augmentation

What if we just add lots of images with small amounts of random noise to our training data?

Cannot have enough new data to densely sample a high dimensional ball around each original input (number of points required grows exponentially with dimension)

Data Augmentation

What if we just add lots of images with small amounts of random noise to our training data?

Cannot have enough new data to densely sample a high dimensional ball around each original input (number of points required grows exponentially with dimension)

Holes will still exist where your network can be exploited

Adversarial Training

Adversarial Training

New Training Objective: Train a network that has lowest loss **when attacked**

Adversarial Training

New Training Objective: Train a network that has lowest loss **when attacked**

$$\min_{\theta} \max_{\epsilon} L(x + \epsilon)$$

Adversarial Training

New Training Objective: Train a network that has lowest loss **when attacked**

$$\min_{\theta} \max_{\epsilon} L(x + \epsilon)$$

Min-Max optimization problem can utilize sets of techniques from adversarial game theory

Adversarial Training

For each batch:

Network produces output y_{pred}

Attacker finds attack noise ϵ

$y_{adv} = y_{pred} + \epsilon$

Compute loss $L(y_{adv}, y)$

Run SGD to update weights

Adversarial Training

For each batch:

Network produces output y_{pred}

Attacker finds attack noise ϵ

$y_{adv} = y_{pred} + \epsilon$

Compute loss $L(y_{adv}, y)$

Run SGD to update weights

Another whole gradient descent process

Adversarial Training

For each batch:

Network produces output y_{pred}

Attacker finds attack noise ϵ

$y_{adv} = y_{pred} + \epsilon$

Compute loss $L(y_{adv}, y)$

Run SGD to update weights

Another whole gradient descent process

Adversary makes move (generates noise)
Defender responds (updates weights)

Adversarial Training

For each batch:

Network produces output y_{pred}

Attacker finds attack noise ϵ

$y_{adv} = y_{pred} + \epsilon$

Compute loss $L(y_{adv}, y)$

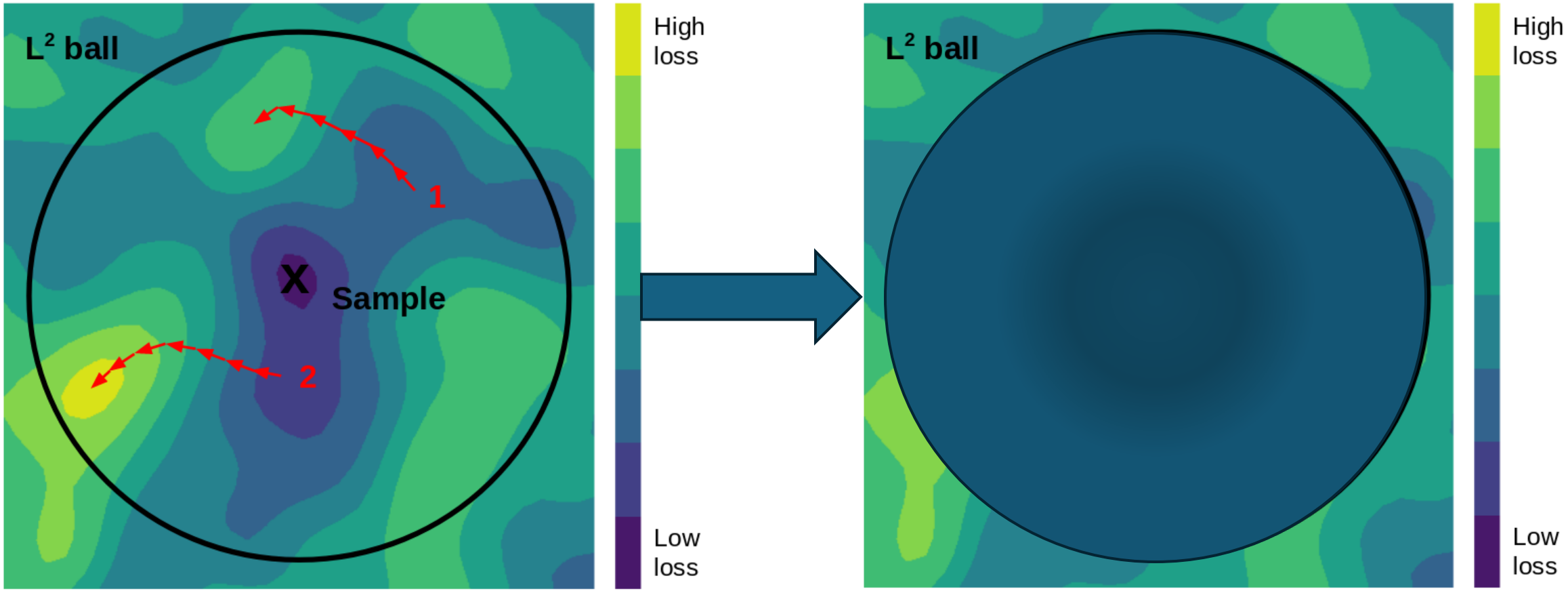
Run SGD to update weights

Another whole gradient descent process

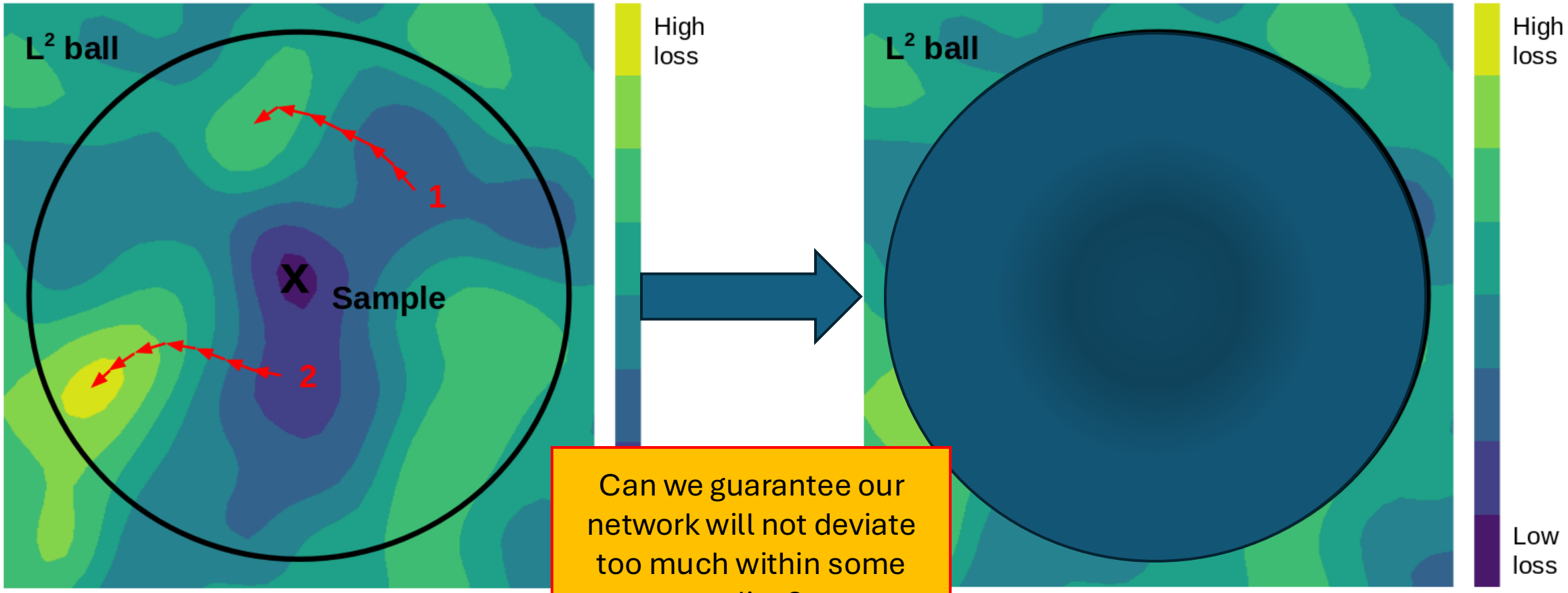
Adversary makes move (generates noise)
Defender responds (updates weights)

What are the tradeoffs of using adversarial training?

Provably Robust Networks

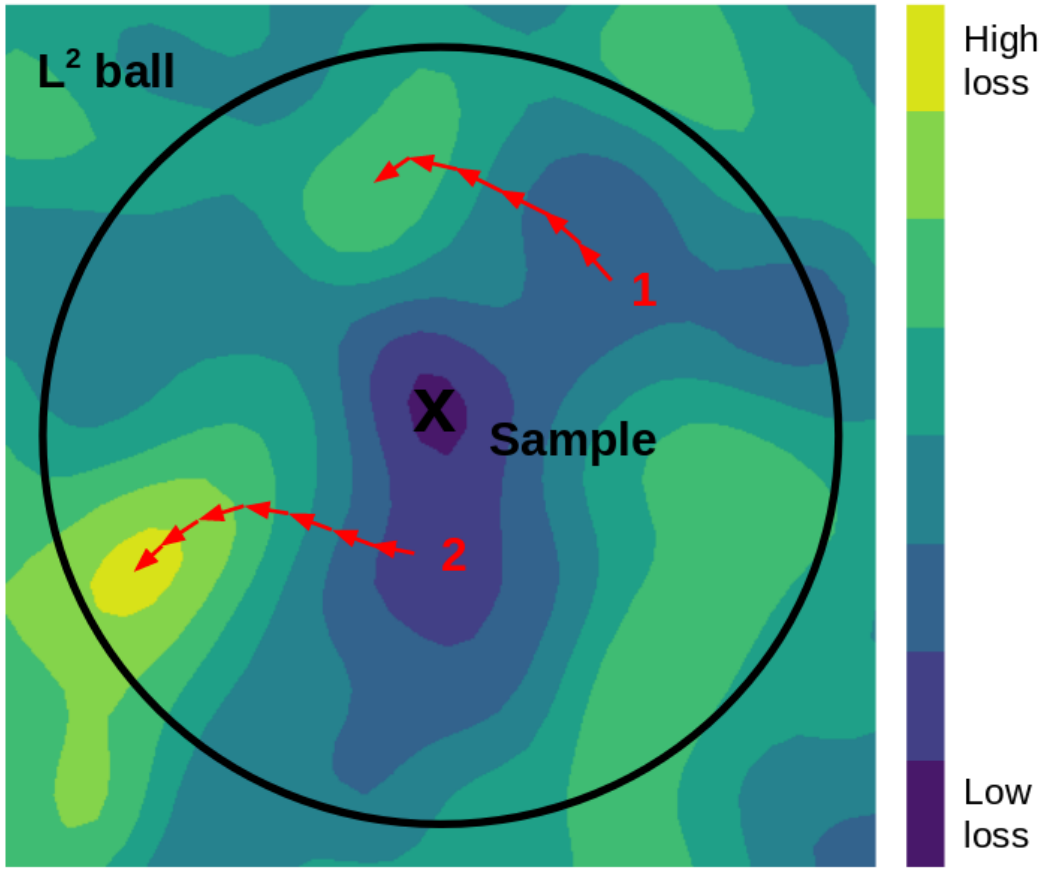


Provably Robust Networks



Maximum Gradient

If we knew the maximum gradient $c = \nabla_\epsilon L$, then we know that our loss function can change up to $c \cdot r$



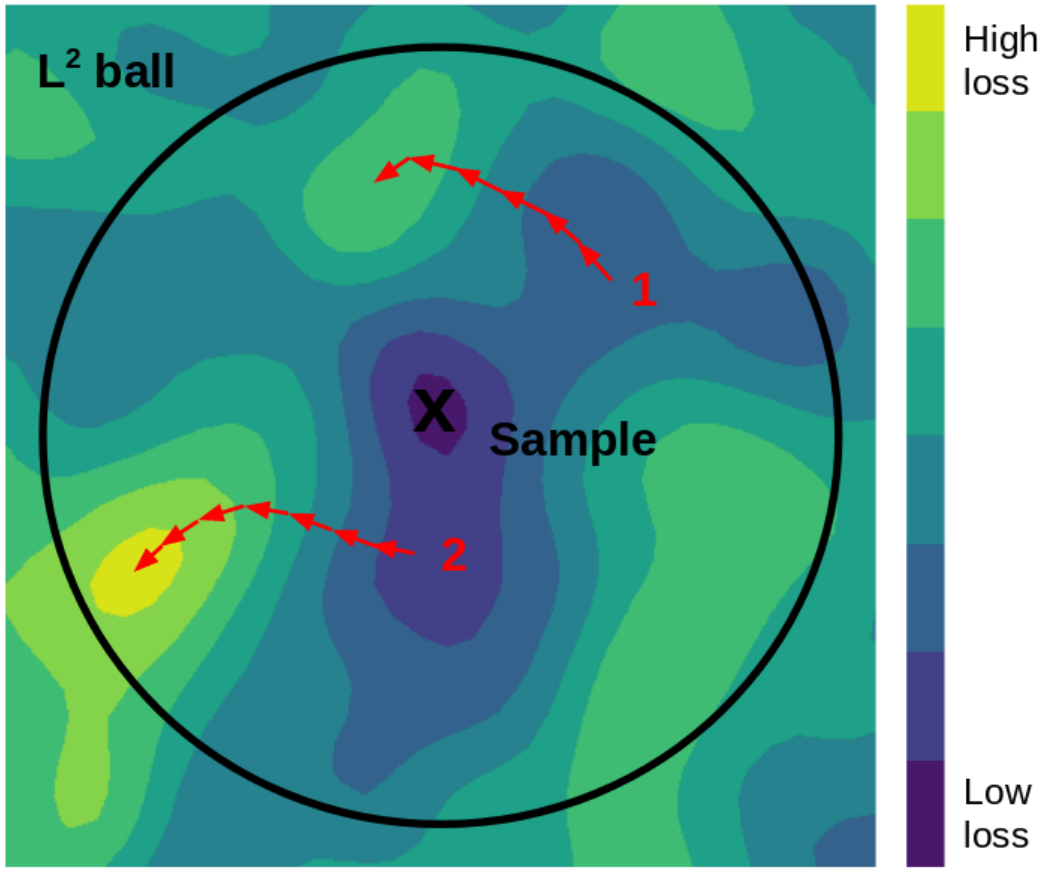
If we can bound the gradient of a function to some constant c, that function is *Lipschitz Continuous*.

Maximum Gradient

If we knew the maximum gradient $c = \nabla_\epsilon L$, then we know that our loss function can change up to $c \cdot r$

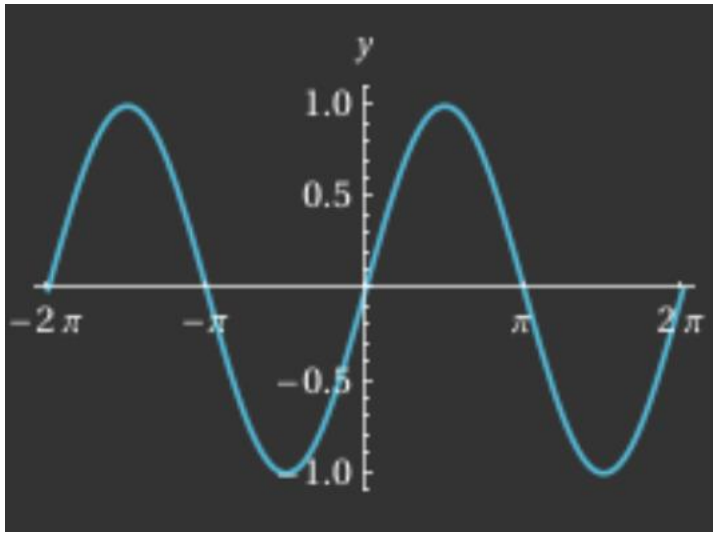
Why is this true?

If we can bound the gradient of a function to some constant c, that function is *Lipschitz Continuous*.

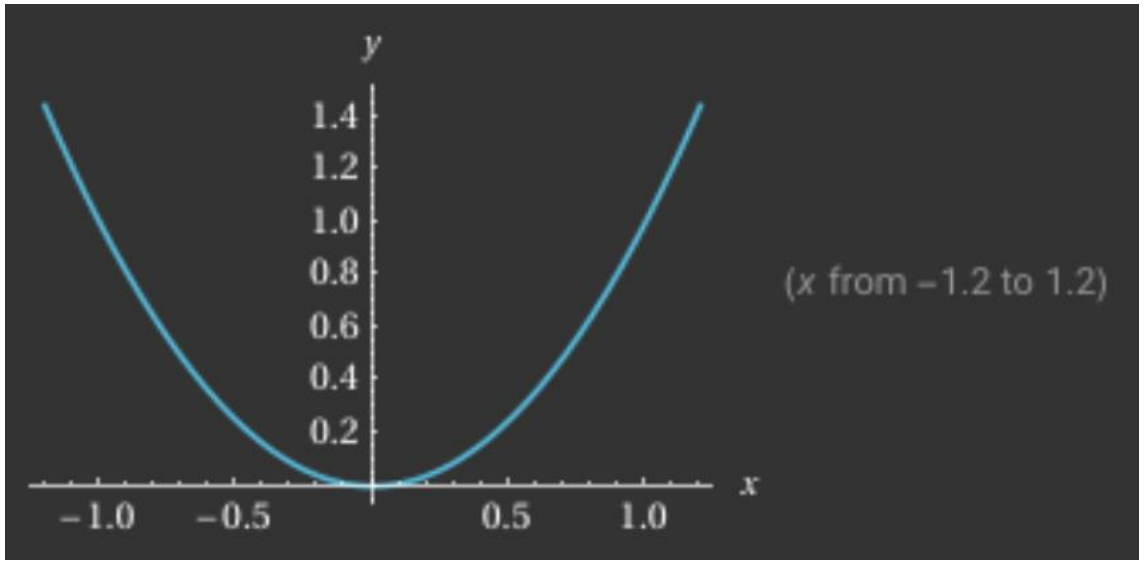


Lipschitz Continuity

$\sin(x)$ is Lipschitz Continuous,
it has a maximum derivative of 1



x^2 is not Lipschitz Continuous, it
does not have a maximum derivative



Are Neural Networks Lipschitz Continuous?

Are Neural Networks Lipschitz Continuous?

- If f and g are both Lipschitz continuous functions, then h=f(g(x)) is also Lipschitz continuous.

Are Neural Networks Lipschitz Continuous?

- If f and g are both Lipschitz continuous functions, then h=f(g(x)) is also Lipschitz continuous.
- Gradients are determined by weight layers and activation functions

Are Neural Networks Lipschitz Continuous?

- If f and g are both Lipschitz continuous functions, then h=f(g(x)) is also Lipschitz continuous.
- Gradients are determined by weight layers and activation functions
- Assume ReLU activation for simplicity (maximum derivative of 1)

Are Neural Networks Lipschitz Continuous?

- If f and g are both Lipschitz continuous functions, then h=f(g(x)) is also Lipschitz continuous.
- Gradients are determined by weight layers and activation functions
- Assume ReLU activation for simplicity (maximum derivative of 1)
- Maximum gradient possible is determined by weights of network (which are finite)

Are Neural Networks Lipschitz Continuous?

- If f and g are both Lipschitz continuous functions, then h=f(g(x)) is also Lipschitz continuous.
- Gradients are determined by weight layers and activation functions
- Assume ReLU activation for simplicity (maximum derivative of 1)
- Maximum gradient possible is determined by weights of network (which are finite)
- Lipschitz constant *c* may be very large, but it exists

Limiting Lipschitz Constant

Limiting Lipschitz Constant

- It can be shown that the Lipschitz Constant for a single weight matrix W is the largest singular value of that matrix
  - The largest Singular Value is the square root of the largest eigenvalue of the matrix $W^T W$

Limiting Lipschitz Constant

- It can be shown that the Lipschitz Constant for a single weight matrix W is the largest singular value of that matrix
  - The largest Singular Value is the square root of the largest eigenvalue of the matrix $W^T W$
- If we want to limit the Lipschitz Constant for a single layer, we just have to divide by that Singular Value…
  - Can divide by 2 * Singular value to limit Lipschitz constant to 1/2

Limiting Lipschitz Constant

- It can be shown that the Lipschitz Constant for a single weight matrix W is the largest singular value of that matrix
  - The largest Singular Value is the square root of the largest eigenvalue of the matrix $W^T W$
- If we want to limit the Lipschitz Constant for a single layer, we just have to divide by that Singular Value…
  - Can divide by 2 * Singular value to limit Lipschitz constant to 1/2
- This is called Spectral Normalization

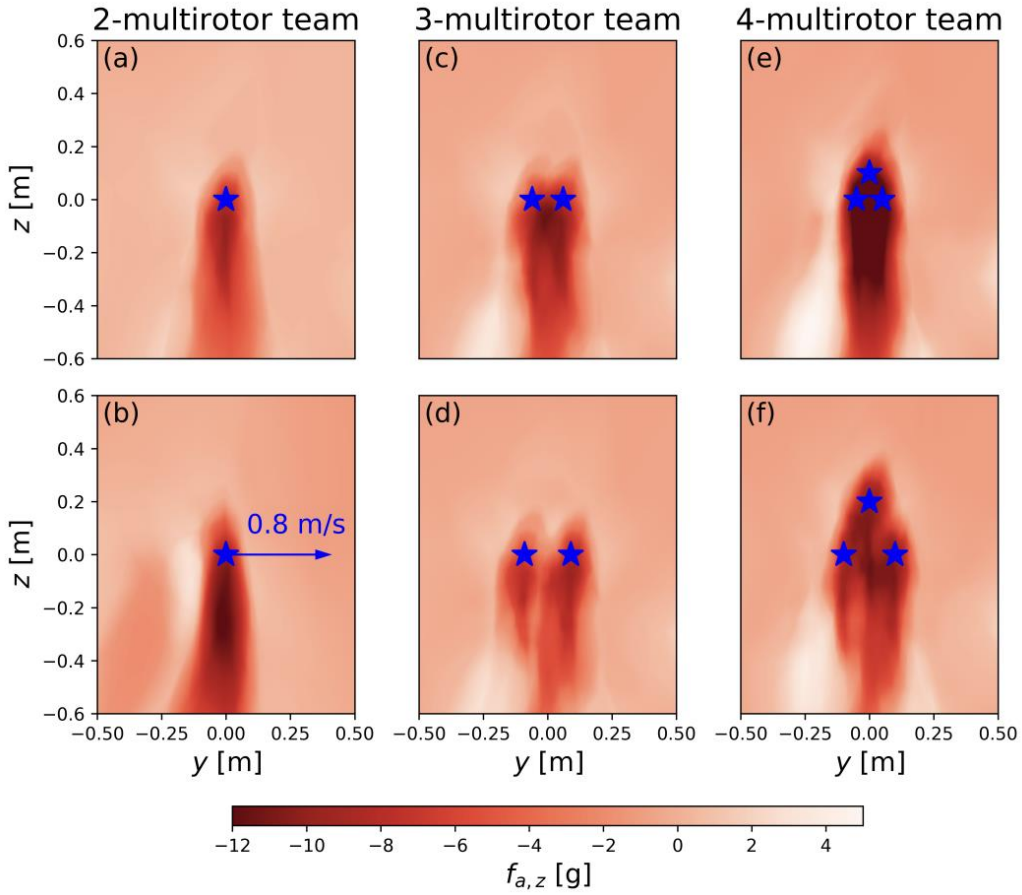
Lipschitz Continuity

- Adding SpectralNormalization to layers, like BatchNorm, can help networks learn smoother loss functions
- Can make models (slightly more) robust to adversarial attacks
- The downside is that it is a much more restrictive condition on the network and the network may no longer learn good policies

Also for other applications…

Many physical phenomena are also Lipschitz Continuous

If you are trying to predict a physical phenomena, it may make sense to use Lipschitz continuity regardless of adversarial attacks.



Shi et al. Neural Swarm. 2022

Takeaways

Adversarial Attacks show how brittle models can be

Studying them gives us insights into what our networks learn

Defenses that make models robust against attacks probably also make them robust against other disturbances as well