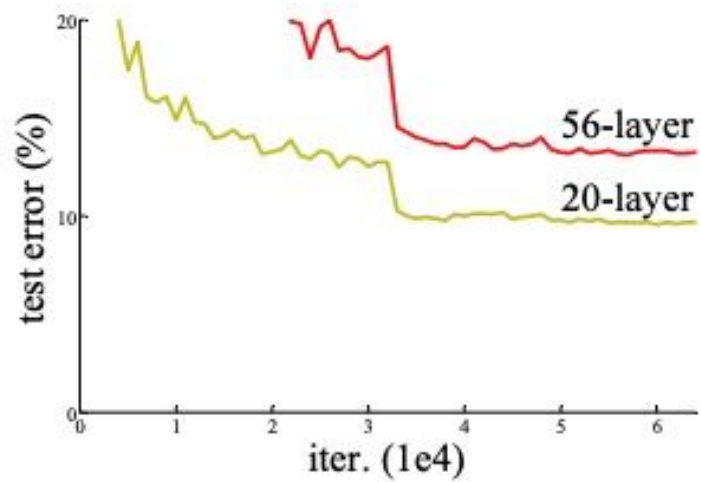


Deep Learning Practice

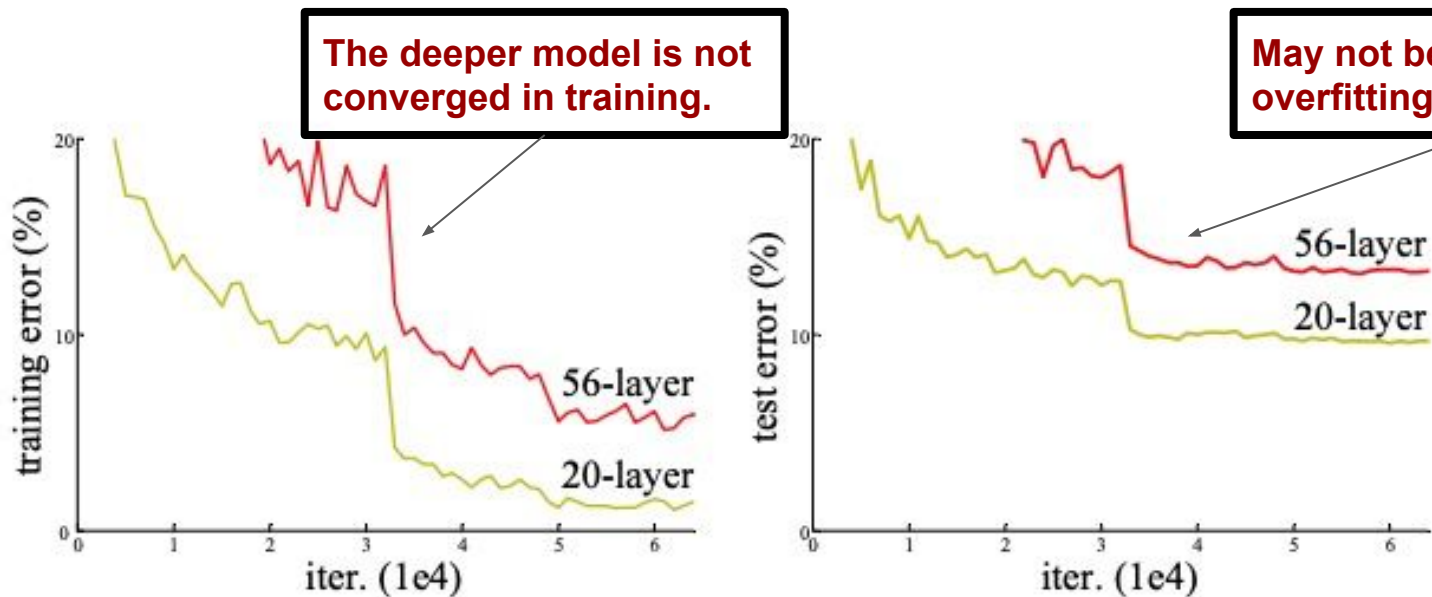
Logistics

1. The following quiz will be designed as survey questionnaire (no technical questions),
2. Assignment II has been released and due next Friday,
3. Our TAs will review the submitted proposal reports and share our comments with you ASAP

Overfitting?



Training a deep model is challenging



Source: <https://arxiv.org/abs/1512.03385>

Agenda

1. Class Imbalance
2. Data Augmentation
 - a. Image
 - b. Text
3. Network Configuration
 - a. Last-Layer Configuration
 - b. Non-linear Activation
4. Parameters Initialization
 - a. Special Initialization
 - b. Transfer Learning
5. Optimizers
6. Regularization Techniques

Make Neural Network Well Trained

Prevent Overfitting

Batch Size

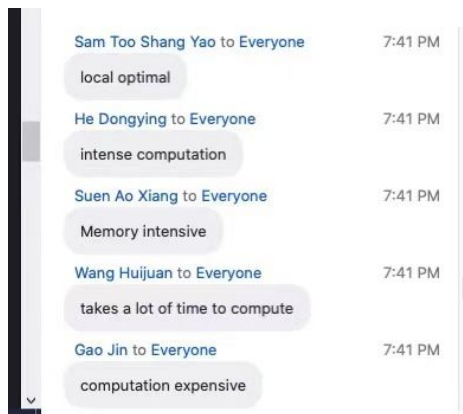
Two approaches to select batch sizes:

1. Batch Gradient Descent

***batch size** = Number of training data*

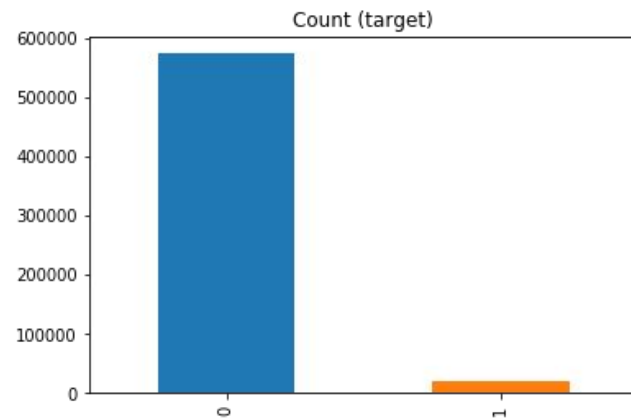
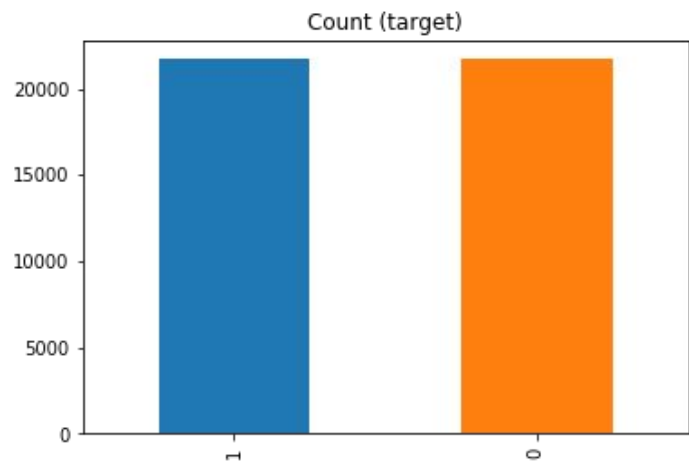
2. Mini-batch Gradient Descent

$1 < \text{batch size} < \text{number of training data}$



Class Imbalance

Small data in some categories



Class imbalance is the norm

1. Bridge Structural Fault Detection
2. Fraud Detection
3. Disease Diagnosis
4. Spam Detection

Why is class imbalance challenging?

1. Not enough knowledge to learn about rare classes
2. Imbalanced problem: the number of fraud cases are much less than the one of normal cases.

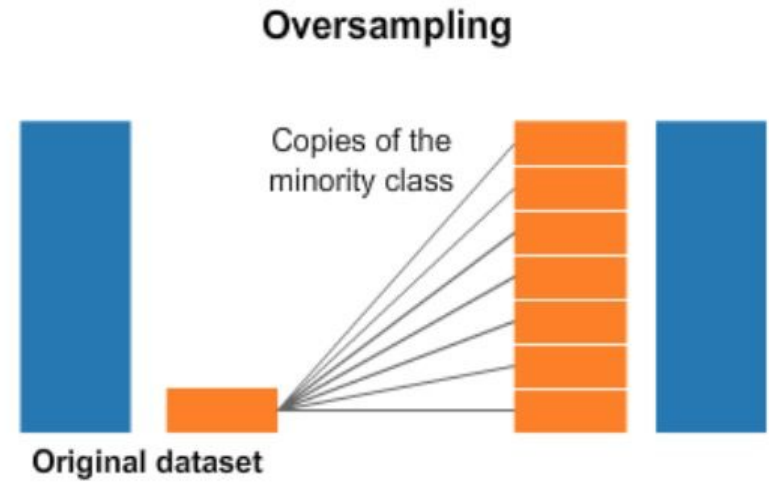
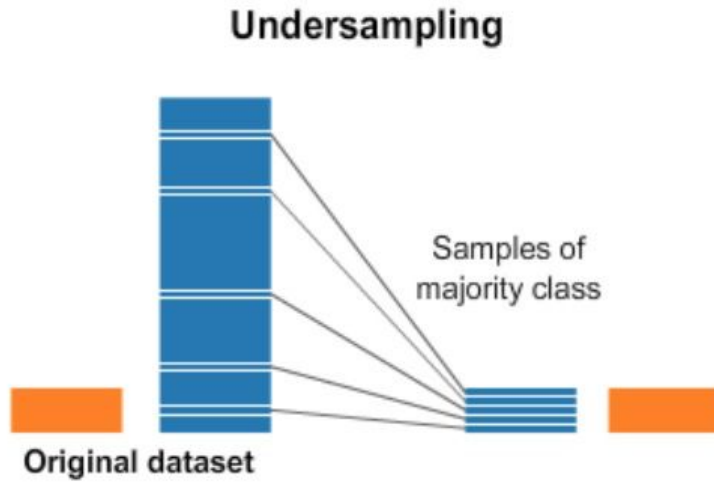


3. Rare classes are usually with high cost of wrong predictions.

How to deal with class imbalance

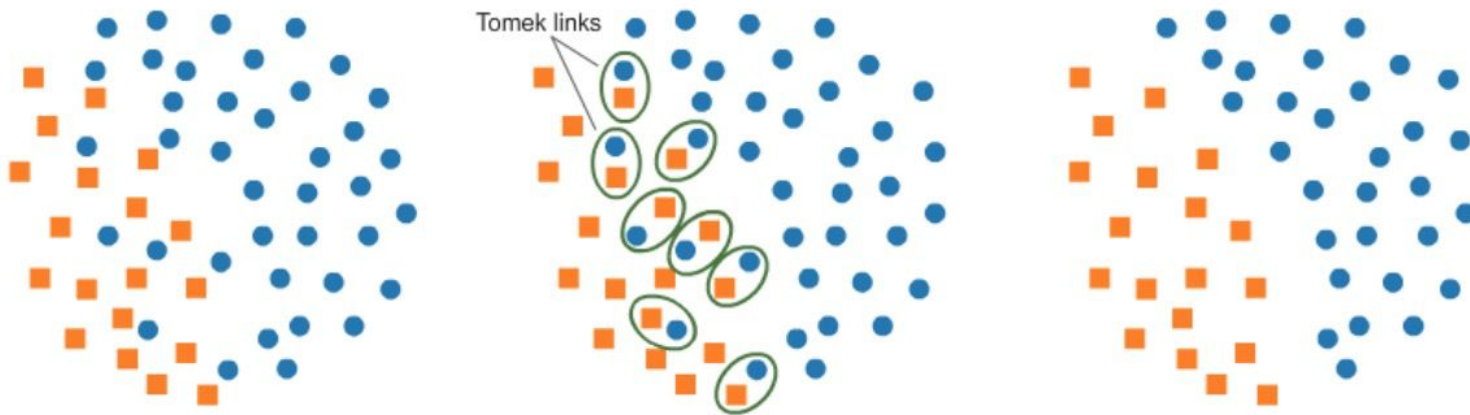
1. Resampling
 - a. Add more minority samples
 - b. Remove majority samples
2. Weights Balancing
 - a. Tweak the loss function
3. Choose robust algorithms to class imbalance

Resampling



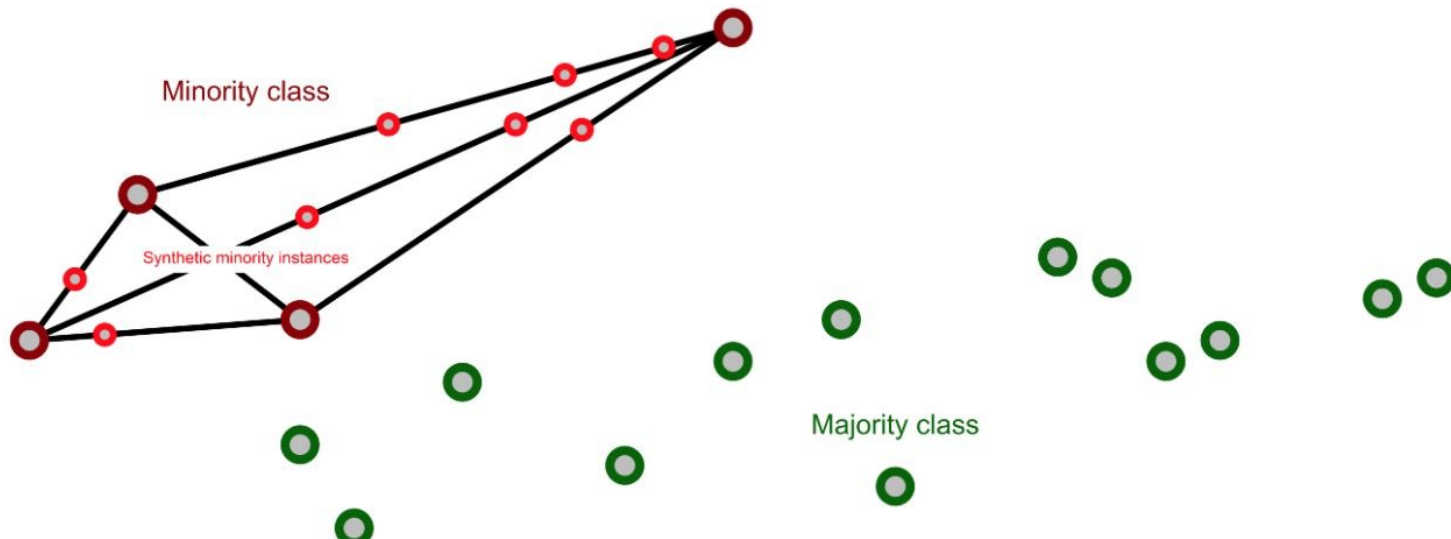
Undersampling: Tomek Links

1. Find pairs of close samples of opposite classes
2. Remove the sample of majority class in each pair



Oversampling: SMOTE

1. Synthesize samples of minority class are convex(~linear) combinations of existing points and their nearest neighbors of same class.



Weight Balancing

1. Normal Loss $L_{\theta} = \sum_i L_{\theta}(x_i)$
2. Weighted Loss

$$L_{\theta} = \sum_i w_{y_i} L_{\theta}(x_i)$$

$$w_c = \frac{N}{N_{y=c}}$$

the number of training samples

the number of training samples in the category: c

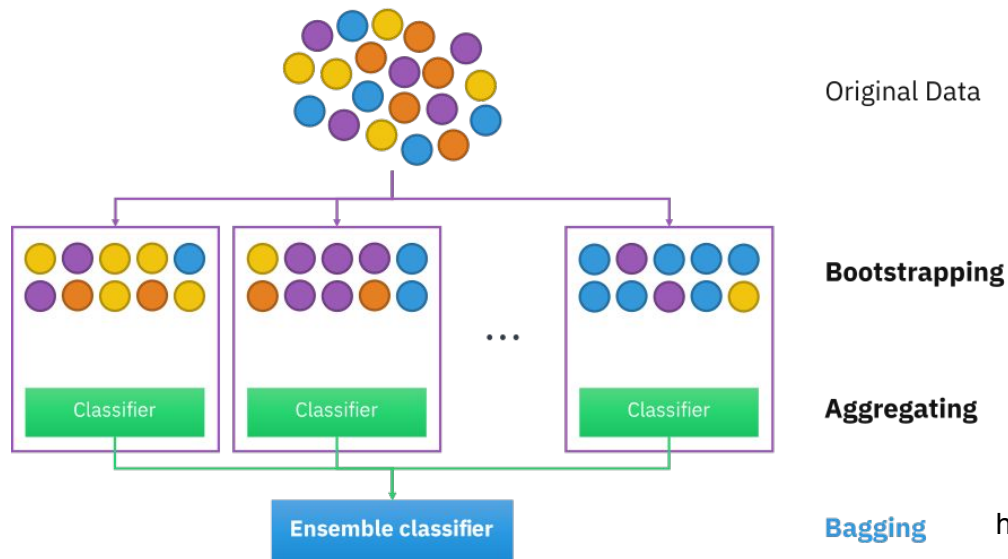
From Keras

`fit` method

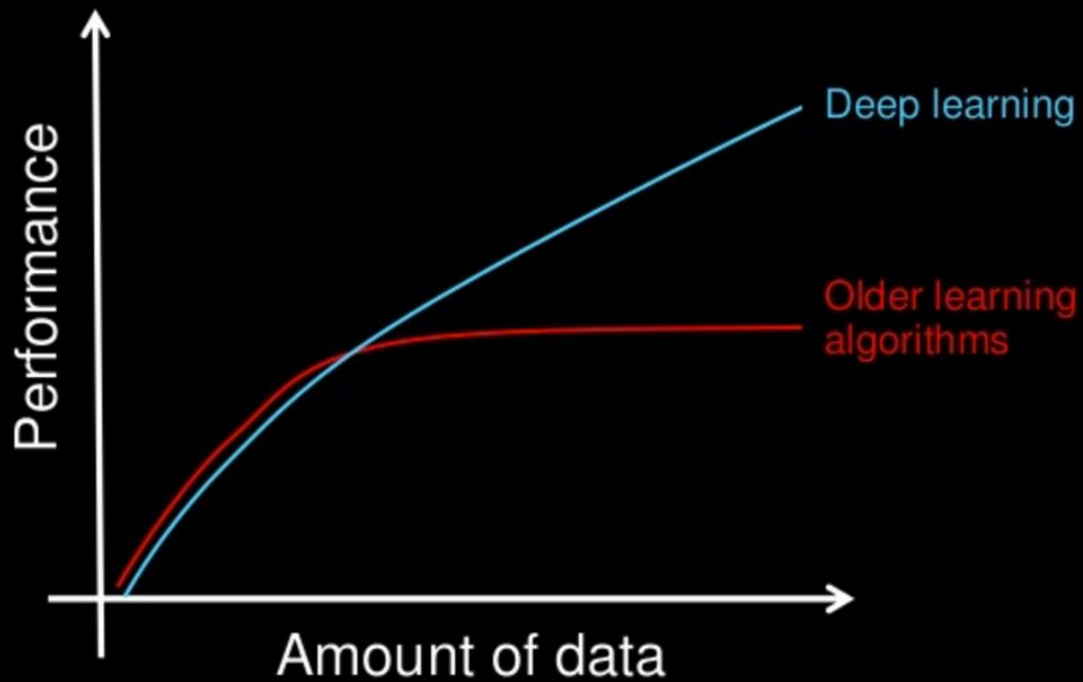
```
Model.fit(  
    x=None,  
    y=None,  
    batch_size=None,  
    epochs=1,  
    verbose=1,  
    callbacks=None,  
    validation_split=0.0,  
    validation_data=None,  
    shuffle=True,  
    class_weight=None,  
    sample_weight=None,  
    initial_epoch=0,  
    steps_per_epoch=None,  
    validation_steps=None,  
    validation_batch_size=None,  
    validation_freq=1,  
    max_queue_size=10,  
    workers=1,  
    use_multiprocessing=False,  
)
```

Robust Algorithm

1. Sample with replacement to create different datasets
2. Train a classifier with each dataset
3. Aggregate predictions from classifiers



Data Augmentation











From Andrew Ng

Data Augmentation

1. Deep learning models usually have billions of parameters and then require massive labeled training data
2. To improve the generalization capability

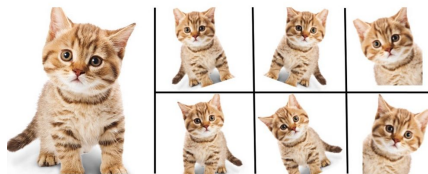
Data Augmentation: create artificially labeled training datasets

Image Augmentation

Original	Flip	Rotation	Random crop
			
<ul style="list-style-type: none">• Image without any modification	<ul style="list-style-type: none">• Flipped with respect to an axis for which the meaning of the image is preserved	<ul style="list-style-type: none">• Rotation with a slight angle• Simulates incorrect horizon calibration	<ul style="list-style-type: none">• Random focus on one part of the image• Several random crops can be done in a row
Color shift	Noise addition	Information loss	Contrast change
			
<ul style="list-style-type: none">• Nuances of RGB is slightly changed• Captures noise that can occur with light exposure	<ul style="list-style-type: none">• Addition of noise• More tolerance to quality variation of inputs	<ul style="list-style-type: none">• Parts of image ignored• Mimics potential loss of parts of image	<ul style="list-style-type: none">• Luminosity changes• Controls difference in exposition due to time of day

How about Text Data

1. In computer vision, data augmentation is quite common.



<https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html>

Rotating an image a few degrees does not change its semantics

2. In NLP or text mining, data augmentation is challenging.

This is simple



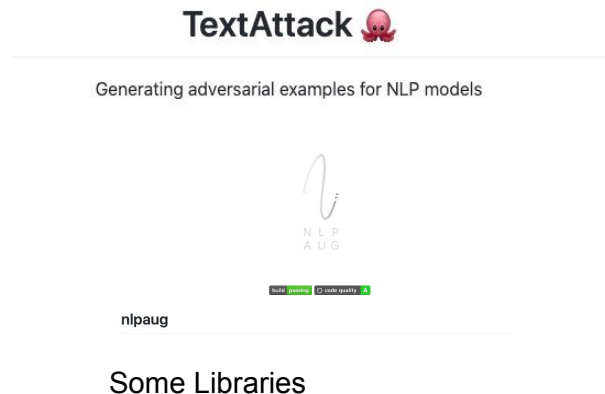
Is this simple

Semantics changed

Text Augmentation

Most of them are very task-specific.

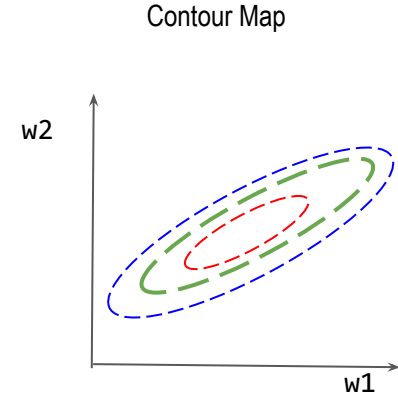
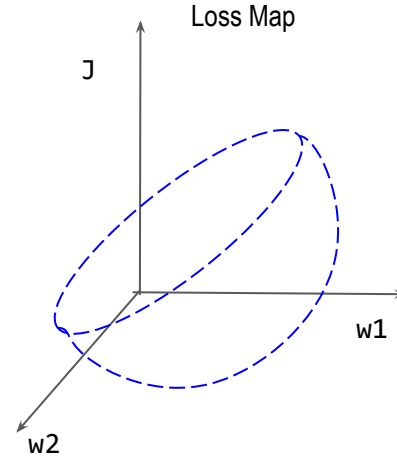
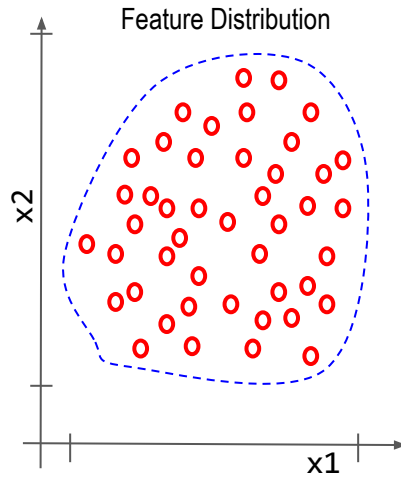
1. Lexical Replacement
2. Back Translation
3. Text Surface Transformation
4. Random Noise Injection
5. Instance Crossover Augmentation
6. Generative Methods



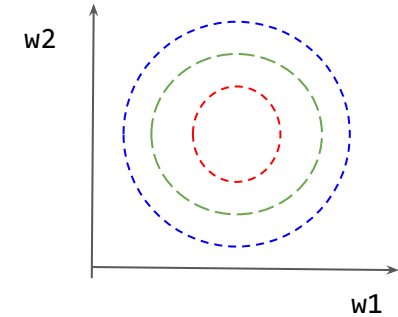
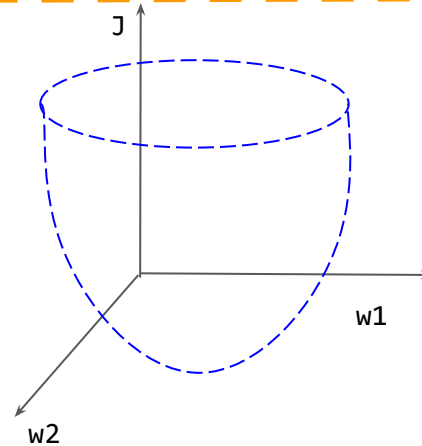
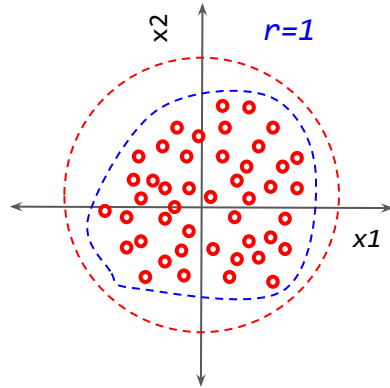
Batch Normalization

Normalization for Neural Network

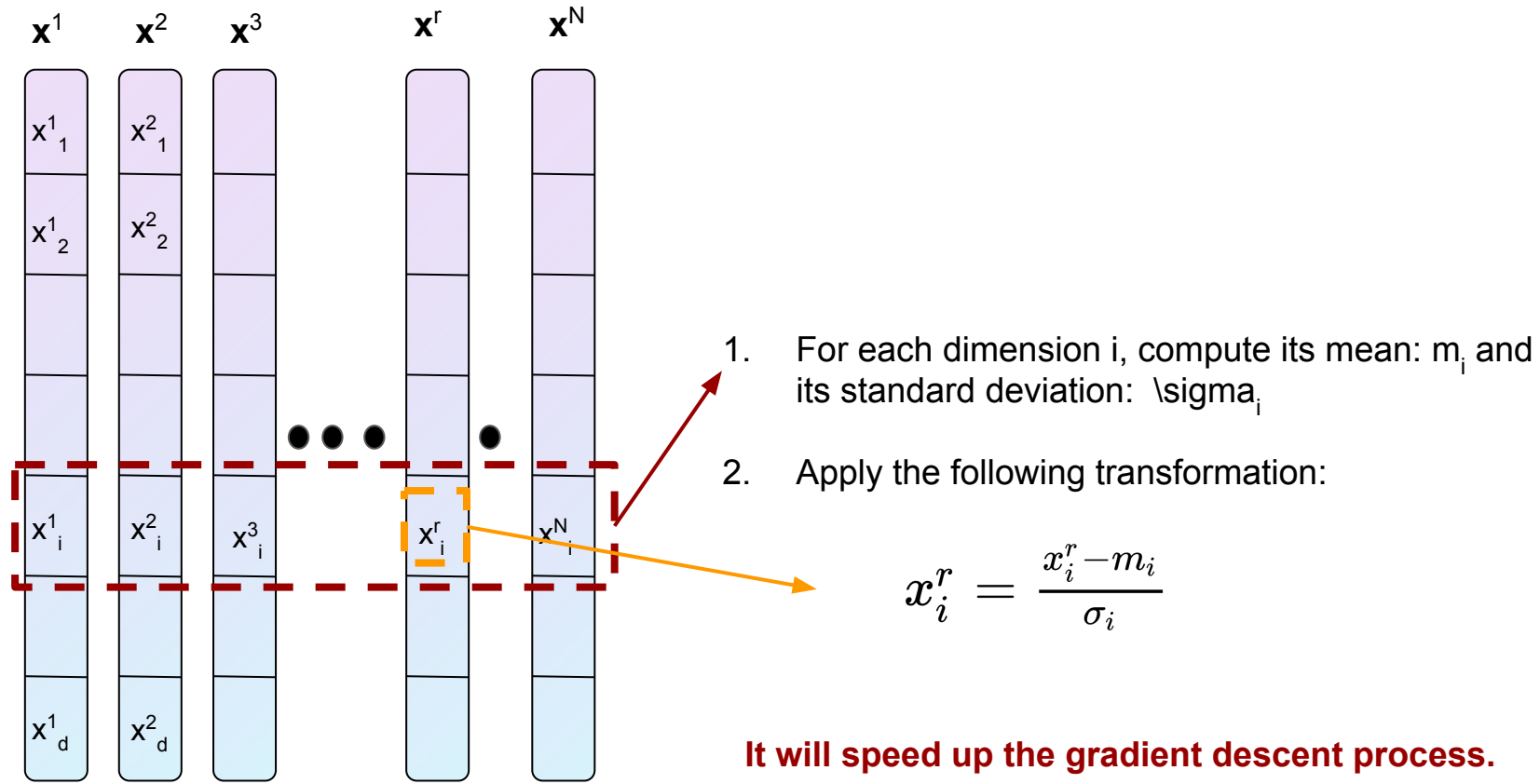
Before
normalization



After
normalization



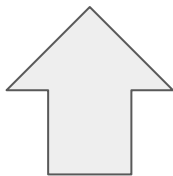
Feature Normalization



How about Hidden Outputs?

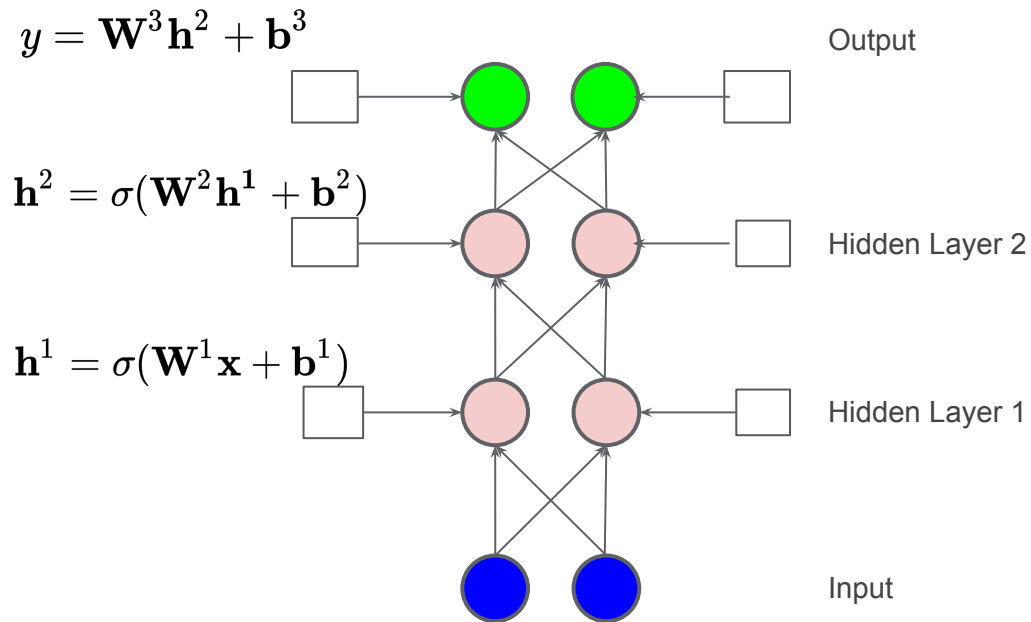
It is challenging to normalize hidden outputs: \mathbf{h}^1 and \mathbf{h}^2

During training, their distributions are changed.

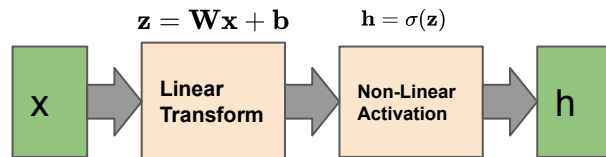


Batch Normalization

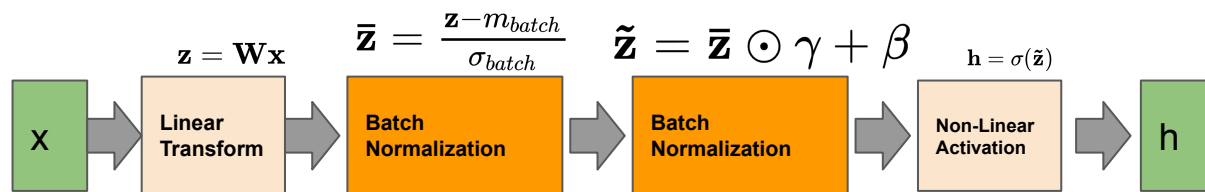
normalization is restrained to each mini-batch in the training process



Batch Normalization



```
model.add(layers.Dense(64, activation='relu'))
```



```
model.add(layers.Dense(64, use_bias=False))  
model.add(layers.BatchNormalization())  
model.add(Activation("relu"))
```

mean and std are computed
from batch samples

Introduce network parameters
to restore the representation
power of the network

Why Batch Normalization?

During testing: how to compute the mean and std

1. Ideal: computing mean and std using the whole training dataset.
2. In practice: compute the moving average of mean and std of the batches during training.

```
model.add(layers.Dense(64, use_bias=False))  
model.add(layers.BatchNormalization())  
model.add(Activation("relu"))
```

What is the number of parameters?

Benefits behind BN

1. Reduce training times, make very deep structure trainable
2. Learning is more stable and less affected by initialization.

Network Configuration

Last-Layer Configuration

Depends on the task type

Last-layer activation

Loss function

Binary Classification

sigmoid

binary_crossentropy

```
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Multi-class Classification

softmax

categorical_crossentropy

```
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Number of unique labels in the task

Last-Layer Configuration

Depends on the task type

Last-layer activation

Loss function

Multi-label Classification

Sigmoid

binary_crossentropy

```
model.add(layers.Dense(10, activation='sigmoid'))  
model.compile(loss="binary_crossentropy", optimizer='rmsprop')
```

Three Type of Classification Tasks

Binary
Classification



- Spam
- Not spam

Multiclass
Classification



- Dog
- Cat
- Horse
- Fish
- Bird
- ...

Multi-label
Classification



- Dog
- Cat
- Horse
- Fish
- Bird
- ...

<https://www.microsoft.com/en-us/research/uploads/prod/2017/12/40250.jpg>

Last-Layer Configuration

Depends on the task type

Last-layer activation

Loss function

Regression to arbitrary values

Linear

mse

```
model.add(layers.Dense(1))  
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

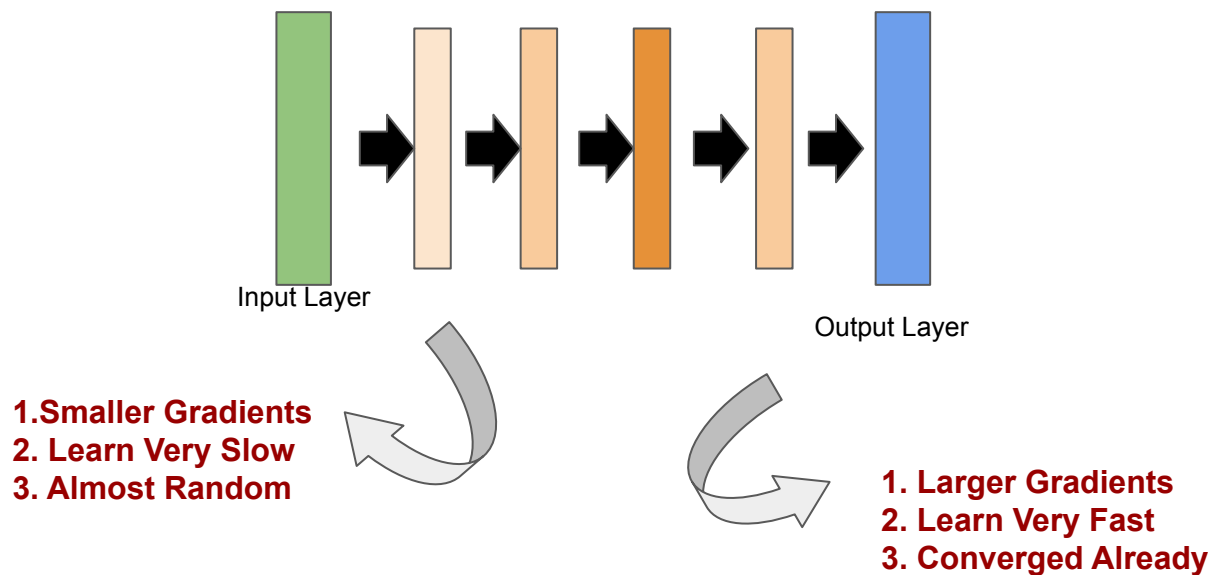
Regression to scaled values
ranging from 0 to 1

sigmoid

mse

```
model.add(layers.Dense(1, activation='sigmoid'))  
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

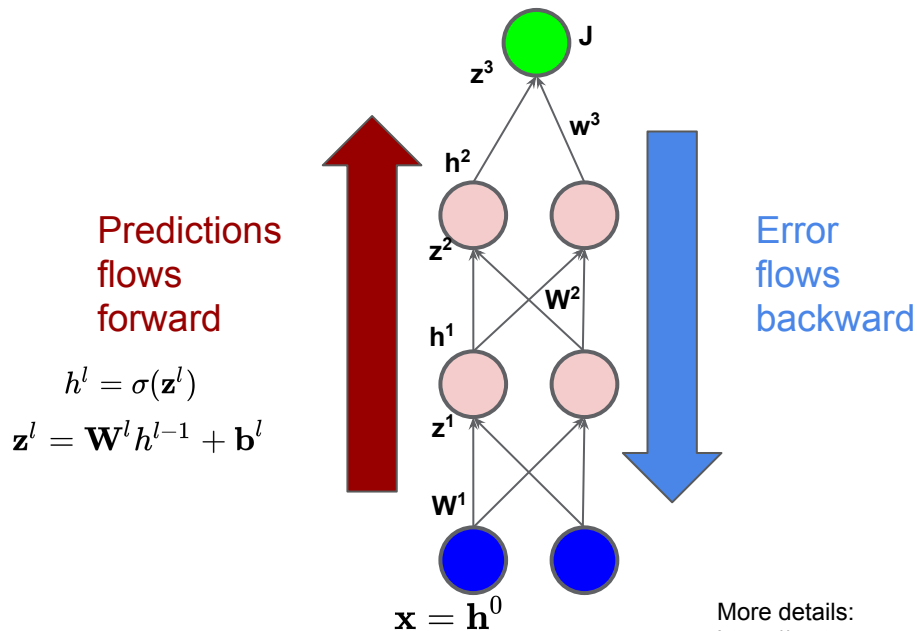

Vanishing Gradient Problem



Backpropagation (from Last Lecture)

Definition (from wiki):

By computing the gradient of the loss function with respect to each weight by the **chain rule**, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule



$$\frac{\partial J}{\partial \mathbf{w}^3} = \frac{\partial J}{\partial \mathbf{z}^3} \frac{\partial \mathbf{z}^3}{\partial \mathbf{w}^3}$$

$$\frac{\partial J}{\partial \mathbf{W}^2} = \frac{\partial J}{\partial \mathbf{z}^3} \frac{\partial \mathbf{z}^3}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{z}^2} \frac{\partial \mathbf{z}^2}{\partial \mathbf{W}^2}$$

From \mathbf{w}^3

$$\frac{\partial J}{\partial \mathbf{W}^1} = \frac{\partial J}{\partial \mathbf{z}^3} \frac{\partial \mathbf{z}^3}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{z}^2} \frac{\partial \mathbf{z}^2}{\partial \mathbf{h}^1} \frac{\partial \mathbf{h}^1}{\partial \mathbf{z}^1} \frac{\partial \mathbf{z}^1}{\partial \mathbf{W}^1}$$

From \mathbf{w}^3 From \mathbf{W}^2

More details:

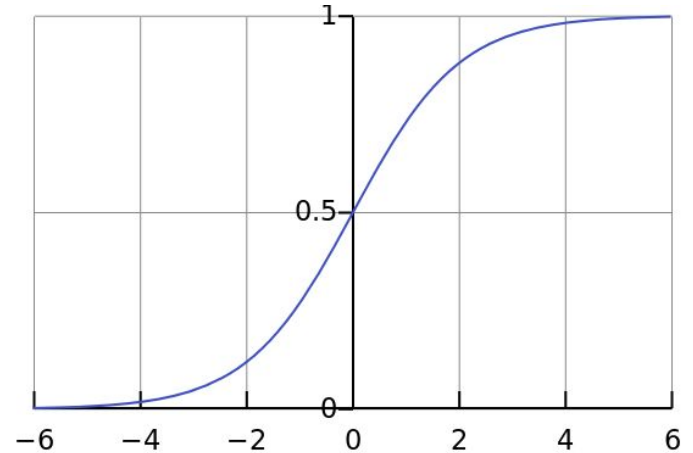
<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

Sigmoid Function

Equation:

$$f(x) = \frac{1}{1+e^{-x}}$$

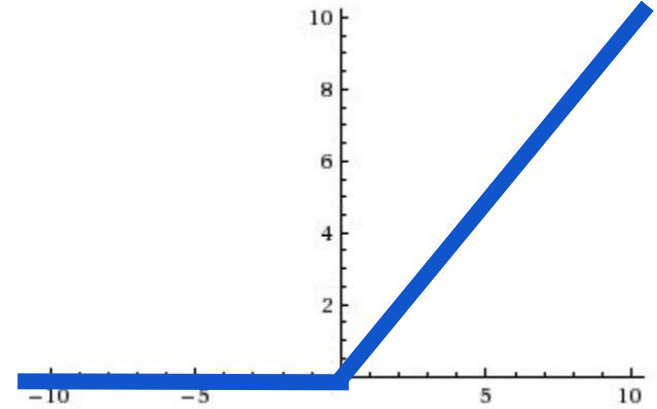
- Vanishing Gradient Problem



How about gradient curve ?

ReLU Function

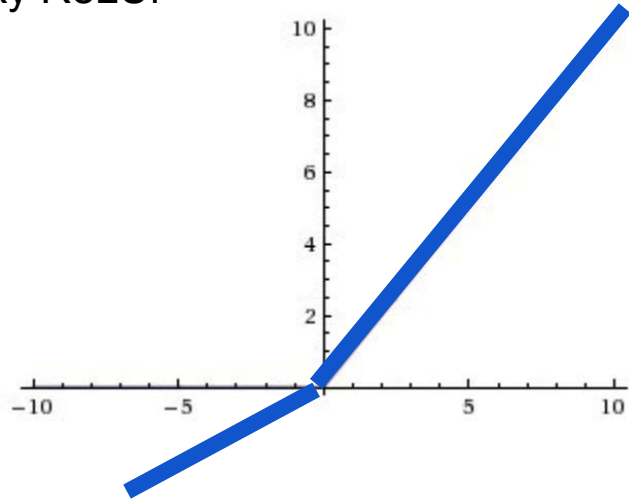
- Fast compute
- Still have vanishing gradient problem



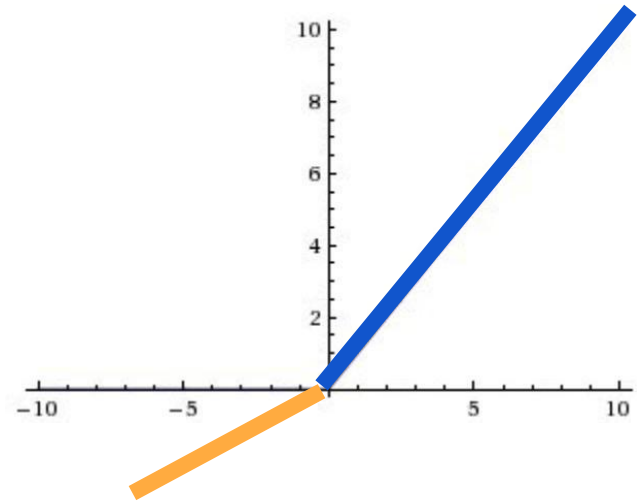
How about gradient curve ?

ReLU Variants

Leaky ReLU:



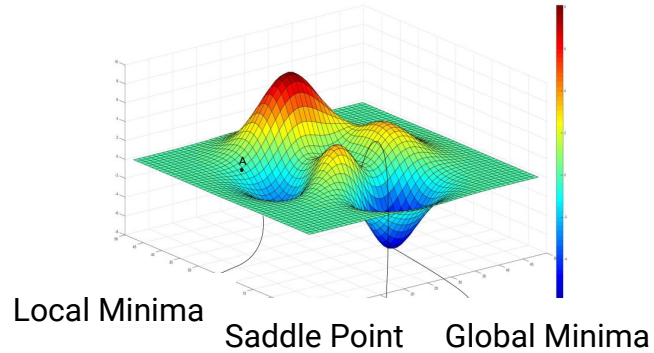
Parametric ReLU::



Parameters Initializations

Initialization

1. Optimization for neural network in nature is an iterative method, which requires initialization.



2. Some general rules for initialization of model parameters:
 - a. Can not initialize all weights to the same value
 - b. **Randomness** should be incorporated

Normal Distribution

1. Initialize weights randomly, following standard normal distribution.
 - The normal distribution should take into account characteristics that are unique to the architecture

For Layers with ReLu

$$\sqrt{\frac{2}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size_l, size_l-1) * np.sqrt(2/size_l-1)$$

<https://keras.io/api/layers/initializers/>

For Layers with Tanh/Sigmoid

$$\sqrt{\frac{1}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size_l, size_l-1) * np.sqrt(1/size_l-1)$$

<https://datascience.stackexchange.com/questions/17987/how-should-the-bias-be-initialized-and-regularized>

Transfer Learning

Task: Build a bear/cat classifier



bear

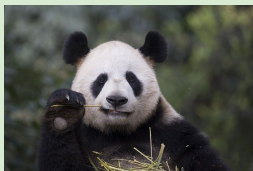


cat

Available Data: not **directly** related



dog



panda

similar domain, different tasks



bear

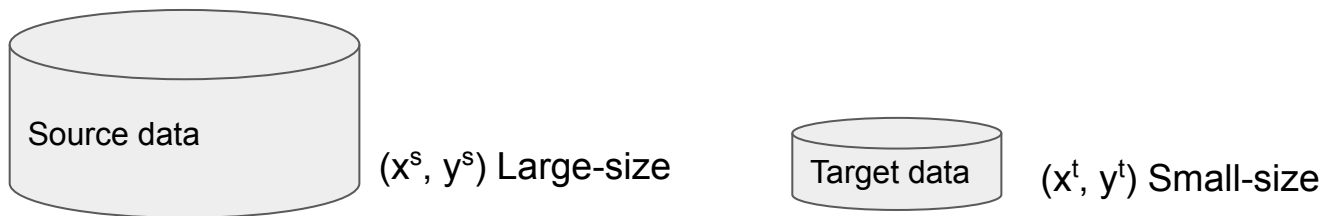


cat

Different domains, similar task

How to transfer knowledge

1. Task Definition:

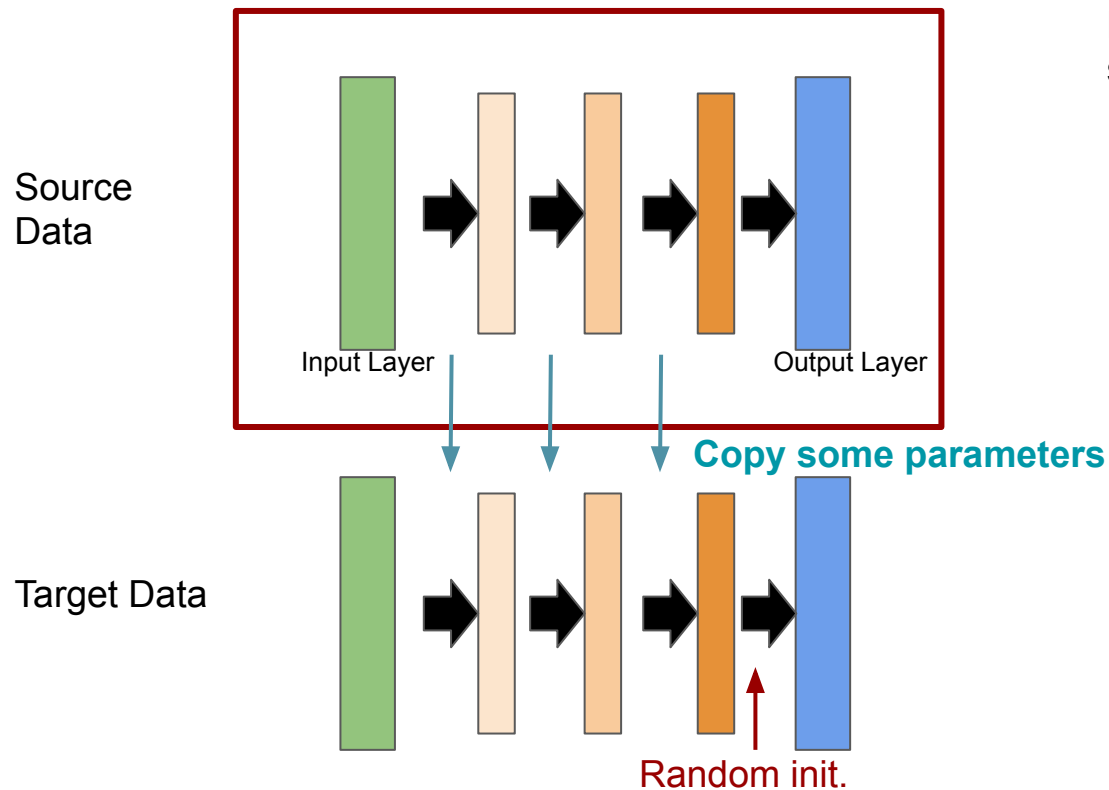


2. Steps:

- a. Train a model using the source data
- b. Transfer Layer from the model trained in source domain to the model in target domain
- c. Fine-tune the model using the target data

Any concerns?

Layer Transfer



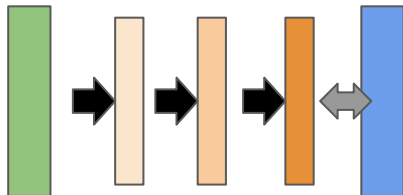
Neural Network: Layer-wise self-contained

1. **Same Task:**
Copy all layers' parameters
2. **Different Tasks:**
Random initialize the softmax/last layer and copy the rest layers' parameters

Fine-tune

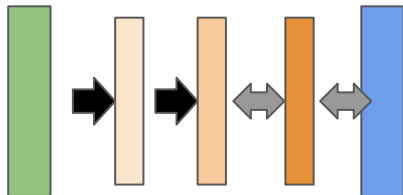
Target Data Size

Small



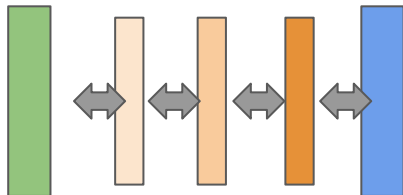
**Freeze all layers, train weights on
softmax/regression layer**

Medium



**Freeze most layers, train weights on last layers
and softmax/regression layer**

Large



Fine-tune all layers

Optimizers for Neural Network

SGD

Gradient for loss function f
over parameters, which
computed by BP algorithm

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

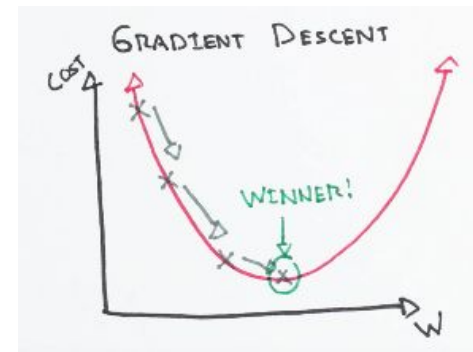
New Parameters Guess

Current Parameters Guess

Learning Rate

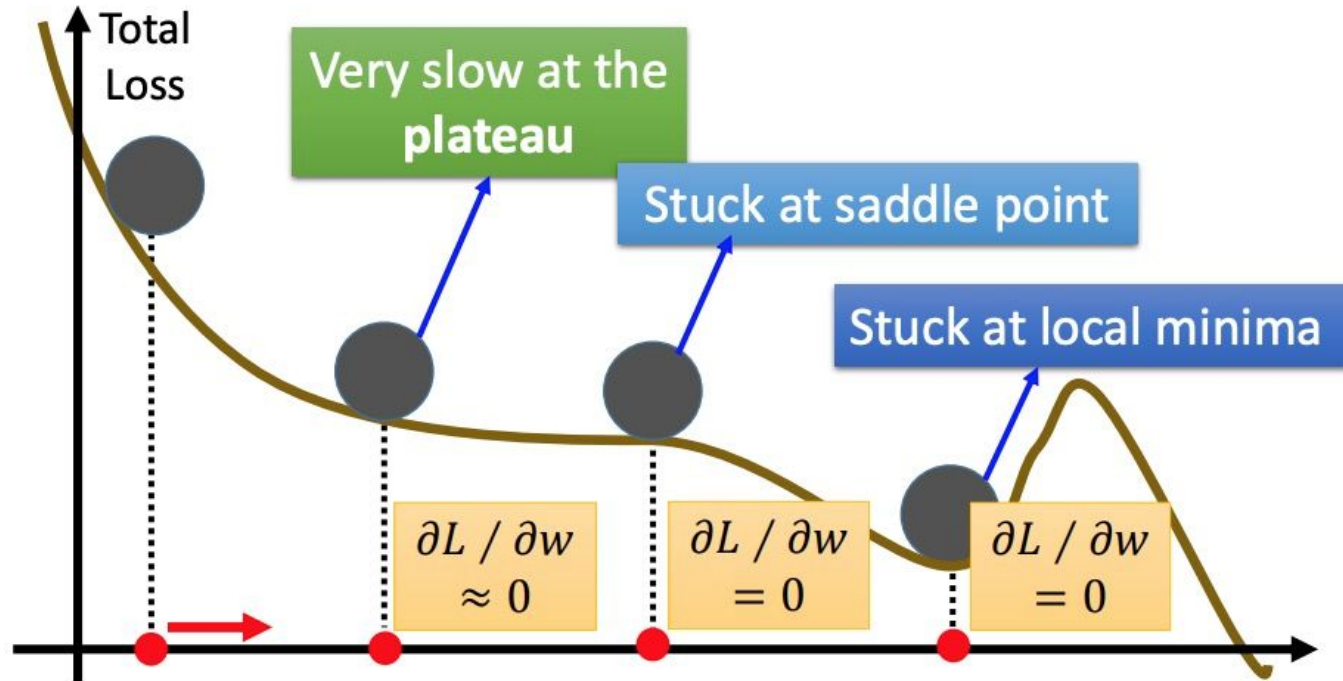


Like hiking down a mountain



Credit: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Hard to find optimal network parameters




Source: <https://speech.ee.ntu.edu.tw/~tlkagk/>

Momentum

Core idea: the current gradient computation will keep the direction as the previous gradient computation

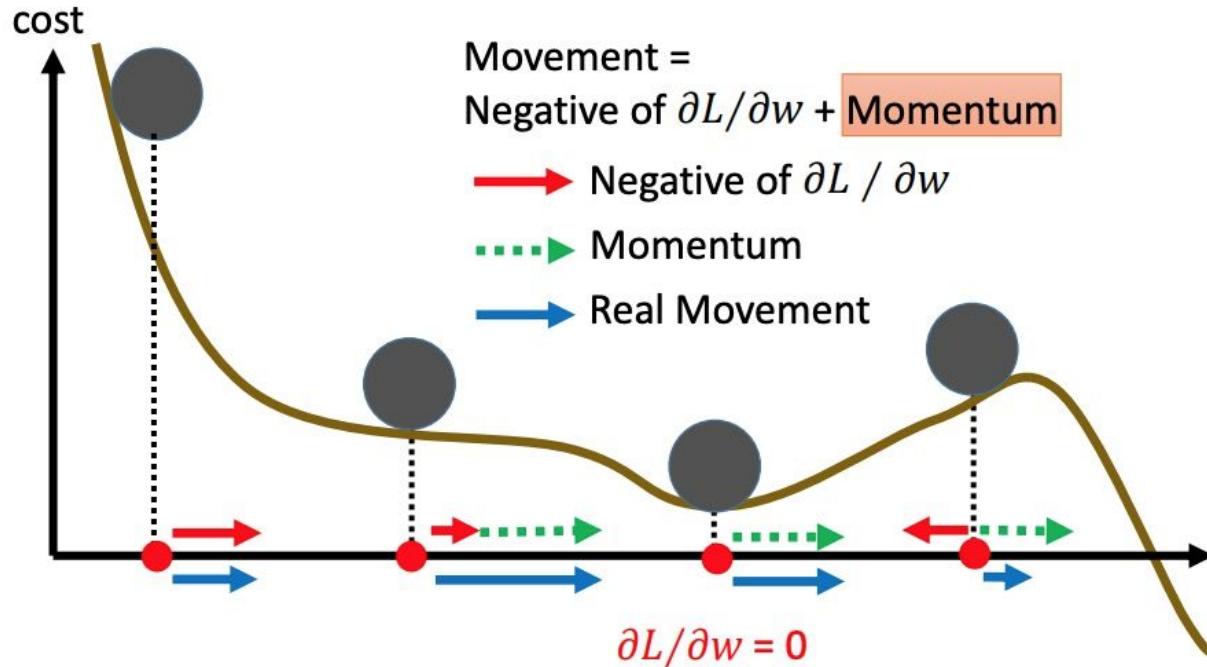
Updated vector of the
previous time step


$$v_t = \beta v_{t-1} + \alpha \nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t$$

- Accelerate SGD
- Dampens Oscillations
- Two Parameters to tune

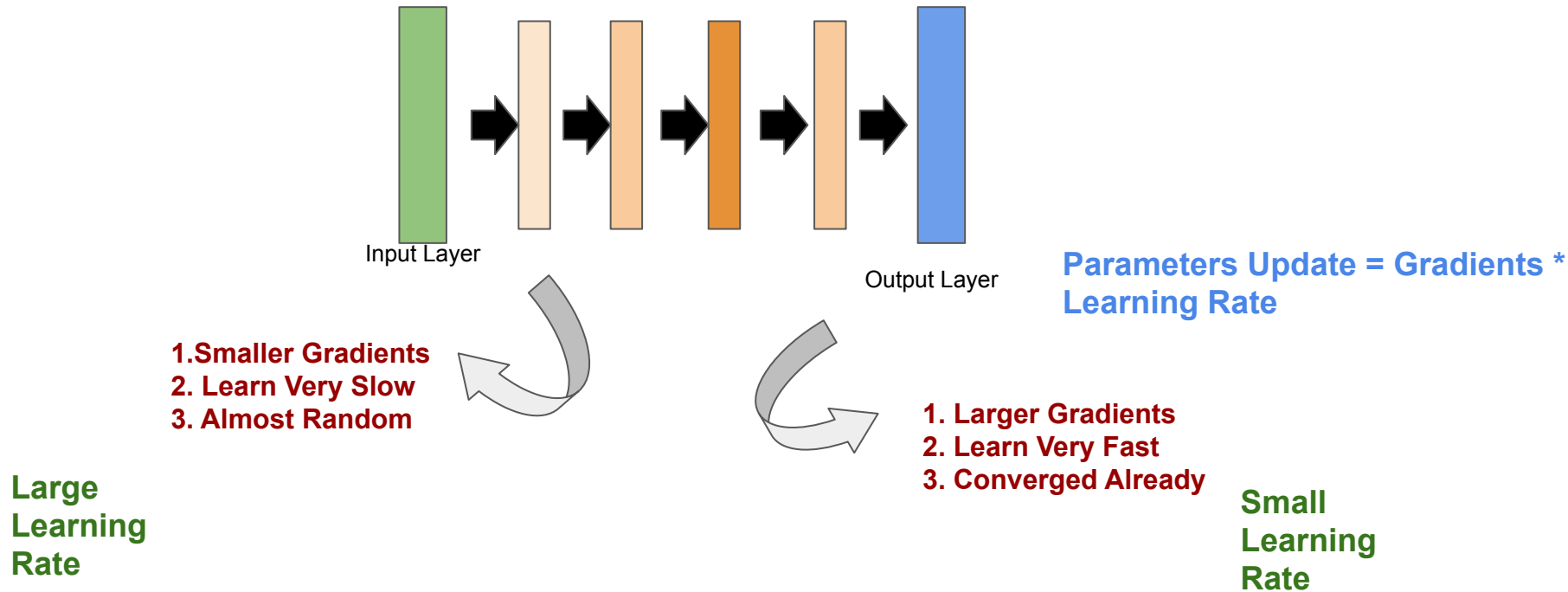
Momentum

Improve the chance to find the
global minima



Source: <https://speech.ee.ntu.edu.tw/~tlkagk/>

Separated Adaptive Learning Rate



Keep a moving average of the squared gradient for each parameter to change the learning rate.

How to select the optimizer

1. Except SGD, Momentum, RMSprop and Adam, other popular methods include Adadelta and Adagrad.
2. It is hard to find a general answer
3. Adam is the most commonly used technique
4. If you want to train a deep or complex neural networks with fast converge, do not just use SGD

Regularization Techniques

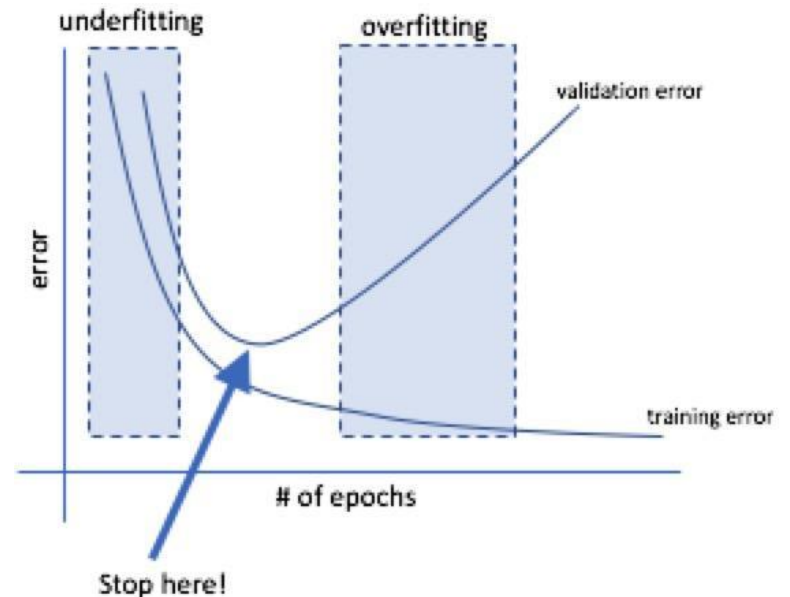
Overfitting for NN

Neural Network with a deep structure easily get overfitted.

1. Early Stopping
2. Parameters Regularization
3. Dropout
4. Most effective: Train with more data

Early Stopping

1. Watch the validation curve
2. Stop updating the weights once validation error starts increasing

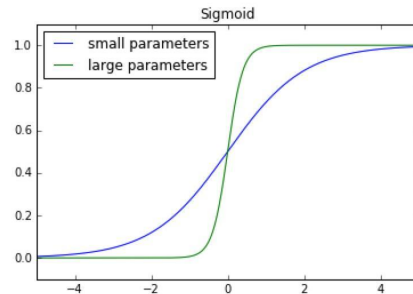


In Keras: https://keras.io/api/callbacks/early_stopping/

Parameter Regularization

Why large model parameters should be penalized:

1. In NN, inputs are linearly combined with parameters. Therefore, large parameters can amplify small changes in the input.
2. Large parameters may **arbitrarily** increase the confidence in our predictions.



To make sure that parameters are not too large and then the model is not overfitting
Add regularization terms to the loss function

$$\dots + \lambda g(\theta)$$

Control the degree to which we select to penalize large parameters

Regularization Terms

1. L1 Regularization:

$$g(\theta) = ||\theta||_1$$

L1-norm is commonly used for feature selection as it tends to produce sparse parameter vectors where only the important features take on non-zero values

2. L2 Regularization:

$$g(\theta) = ||\theta||_2^2$$

L2-Norm does not tend to push less important weights to zero and typically produces better results when training a model.

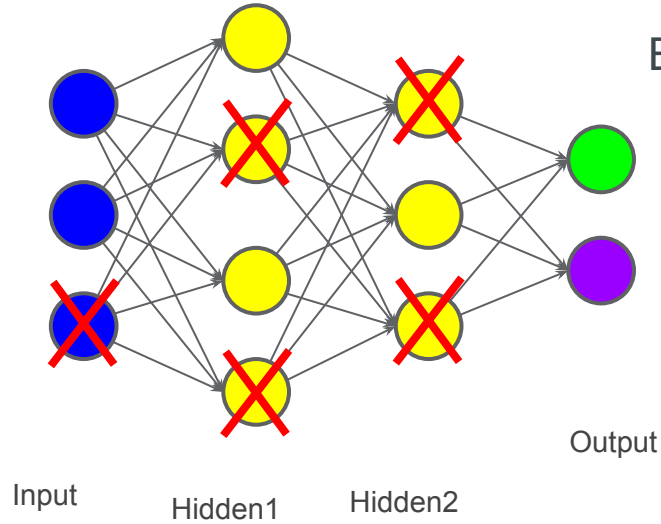
3. Elastic Net:

$$g(\theta) = \alpha ||\theta||_1 + (1 - \alpha) ||\theta||_2^2$$

Trade-off between L1 and L2 Regularization techniques

Dropout

Training:

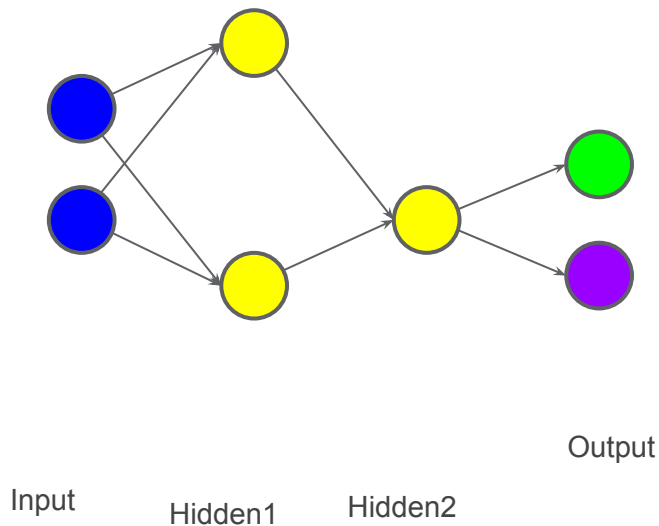


Each mini-batch before updating the parameters

1. Each neuron has **%p** to dropout(mask)

Dropout

Training:



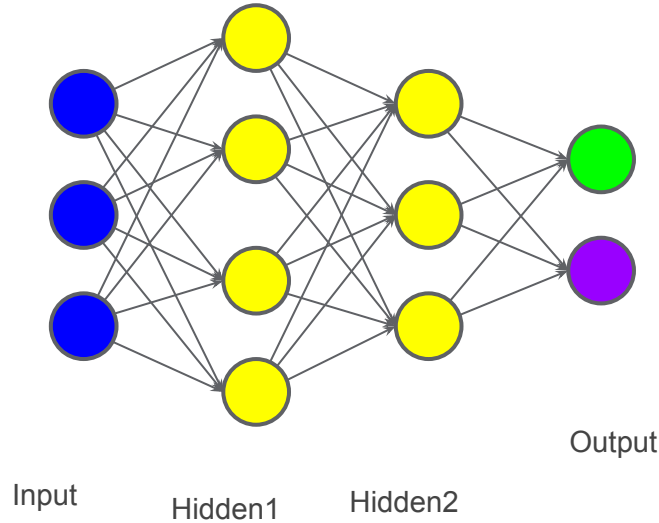
Each mini-batch before updating the parameters

1. Each neuron has **%p** to dropout(mask)
2. The network structure is changed (More Thinner!)
3. Using the updated network structure for training

For each mini-batch, we resample the dropout neurons.

Dropout

Testing:



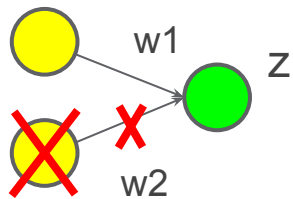
No dropout, but shrink weights following the rule:

If the dropout rate during training is $p\%$, all the weights will time $1-p\%$.

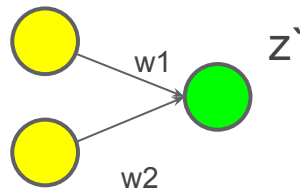
When many people work together, they usually rely on others to do more of the work and share the same results.

Dropout in testing

Training: Assume dropout rate is 50%



Testing: No dropout



Directly Copy:

$$z' = 2z$$

Weight multiply $1-p\%$:

$$z' \sim z$$

Dropout Effects

Experimental Studies on MNIST dataset:

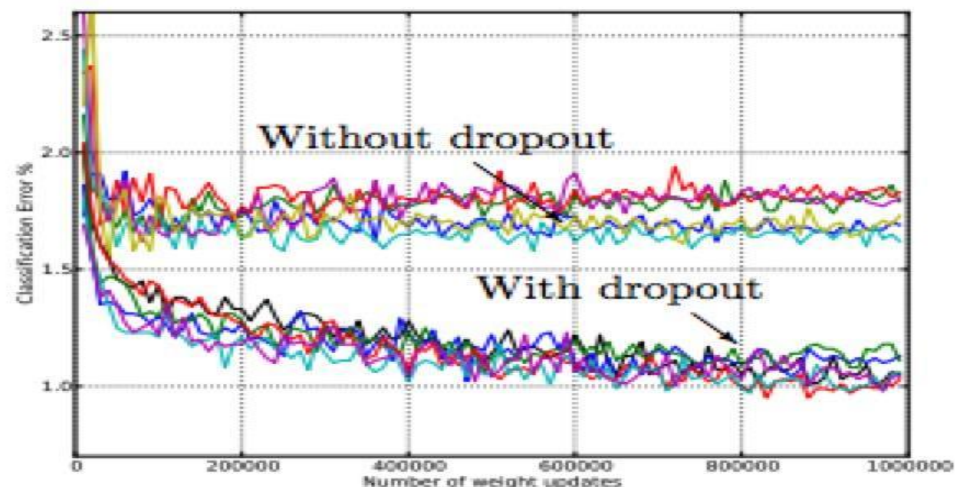


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

When DL may not Work

Limitations

- DL always requires a large amount of annotated data



14 million

Pre-training, Transfer Learning, Data Augmentation

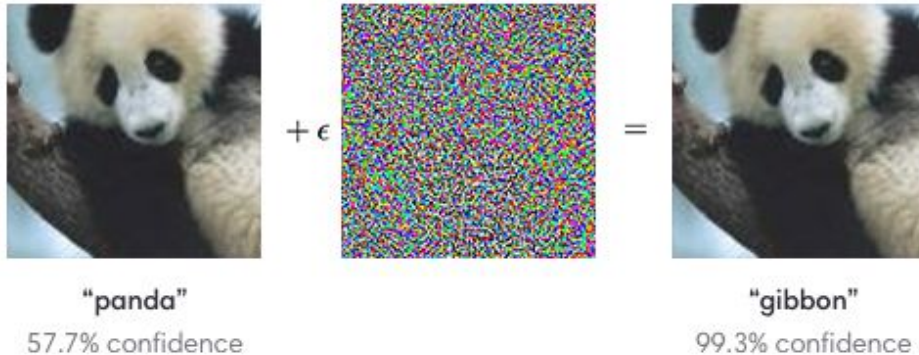
- Generalization capability is low, e.g. the model that perform well on benchmarked datasets fail badly on real world images



- Easily got attacked by random, tiny noise
- How to explain such huge black box

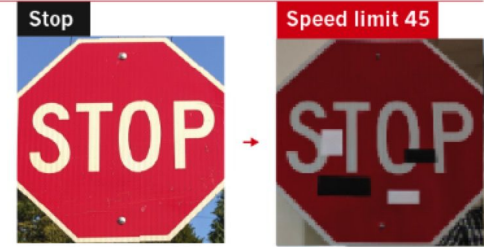
Attack Machine Learning

Adversarial Examples

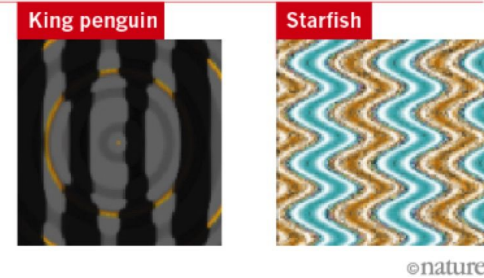


Open AI

These stickers made an artificial-intelligence system read this stop sign as 'speed limit 45'.



Scientists have evolved images that look like abstract patterns — but which DNNs see as familiar objects.



Why deep-learning AIs are so easy to fool

Three challenges for Deep Learning

- ▶ **Deep Supervised Learning** works well for perception
 - ▶ When labeled data is abundant.
- ▶ **Deep Reinforcement Learning** works well for action generation
 - ▶ When trials are cheap, e.g. in simulation.
- ▶ **Three problems the community is working on:**
- ▶ **1. Learning with fewer labeled samples and/or fewer trials**
 - ▶ Self-supervised learning / unsup learning / learning to fill in the blanks
 - ▶ learning to represent the world before learning tasks
- ▶ **2. Learning to reason**, beyond “system 1” feed-forward computation.
 - ▶ Making reasoning compatible with gradient-based learning.
- ▶ **3. Learning to plan complex action sequences**
 - ▶ Learning hierarchical representations of action plans