GANs, Federated Learning, Deep Reinforcement Learning, Adversarial Robustness, Model Drift

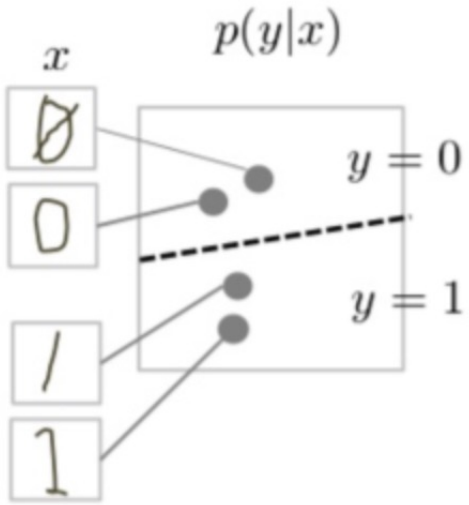
Lecture 11

Parijat Dube

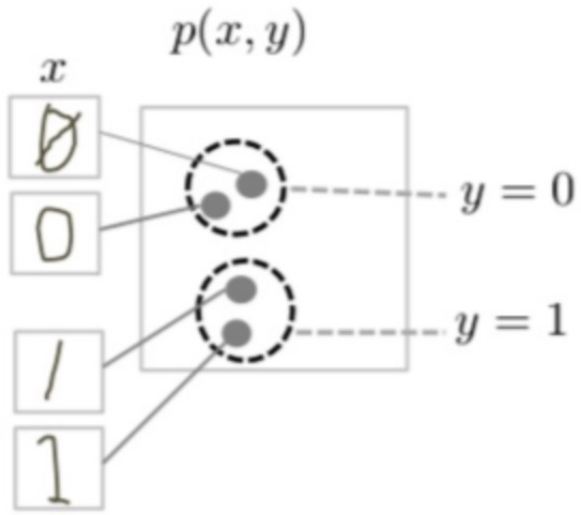
Generative vs Discriminative Models

- **Generative** models can generate new data instances.
- **Discriminative** models discriminate between different kinds of data instances.
- **Generative** models capture the joint probability p(X, Y), or just p(X) if there are no labels.
- **Discriminative** models capture the conditional probability p(Y | X).
- Discriminative models try to draw boundaries in the data space, while generative models try to model how data is placed throughout the space.

- Discriminative Model



- Generative Model

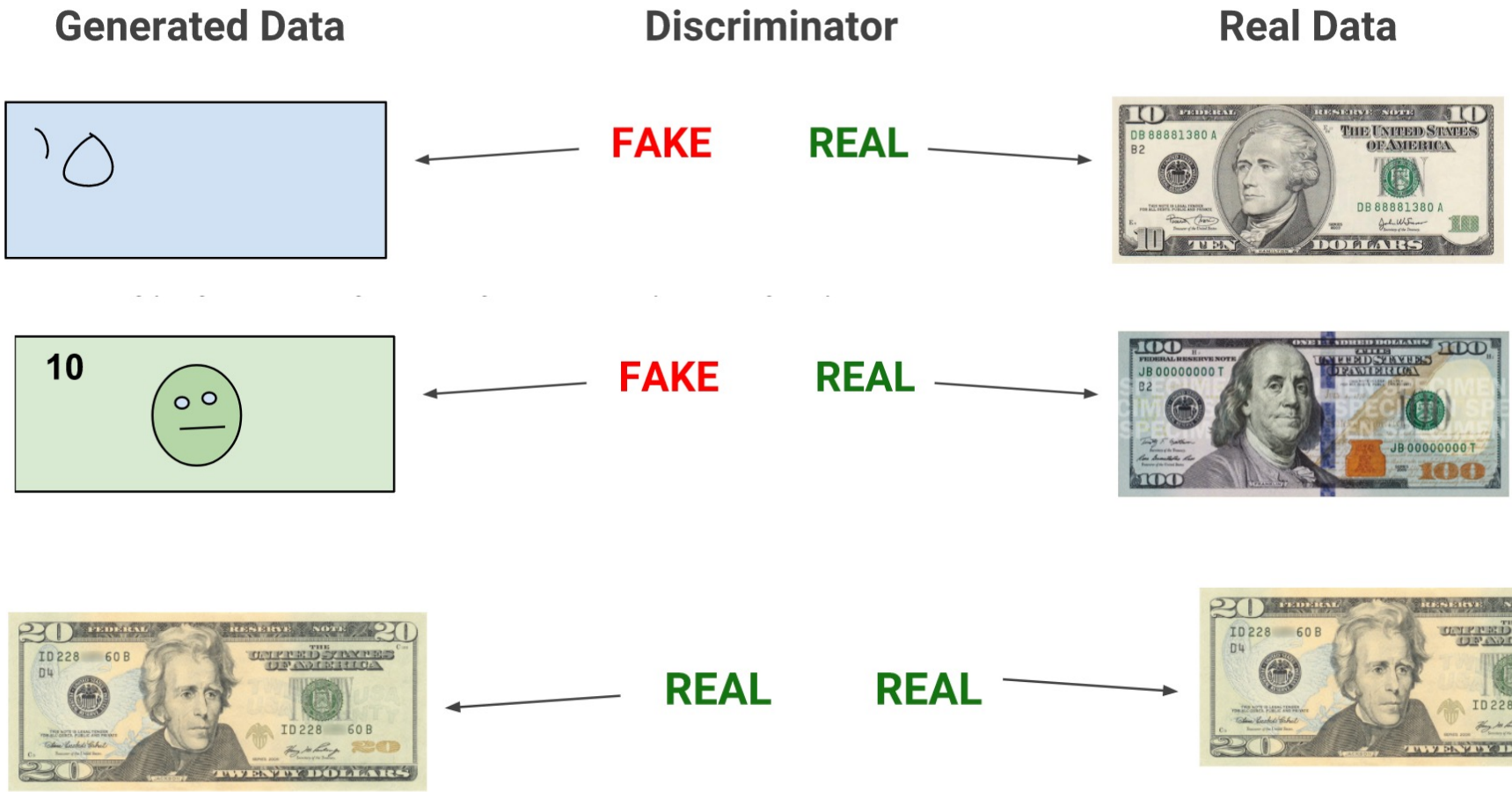


Generative models are harder to train

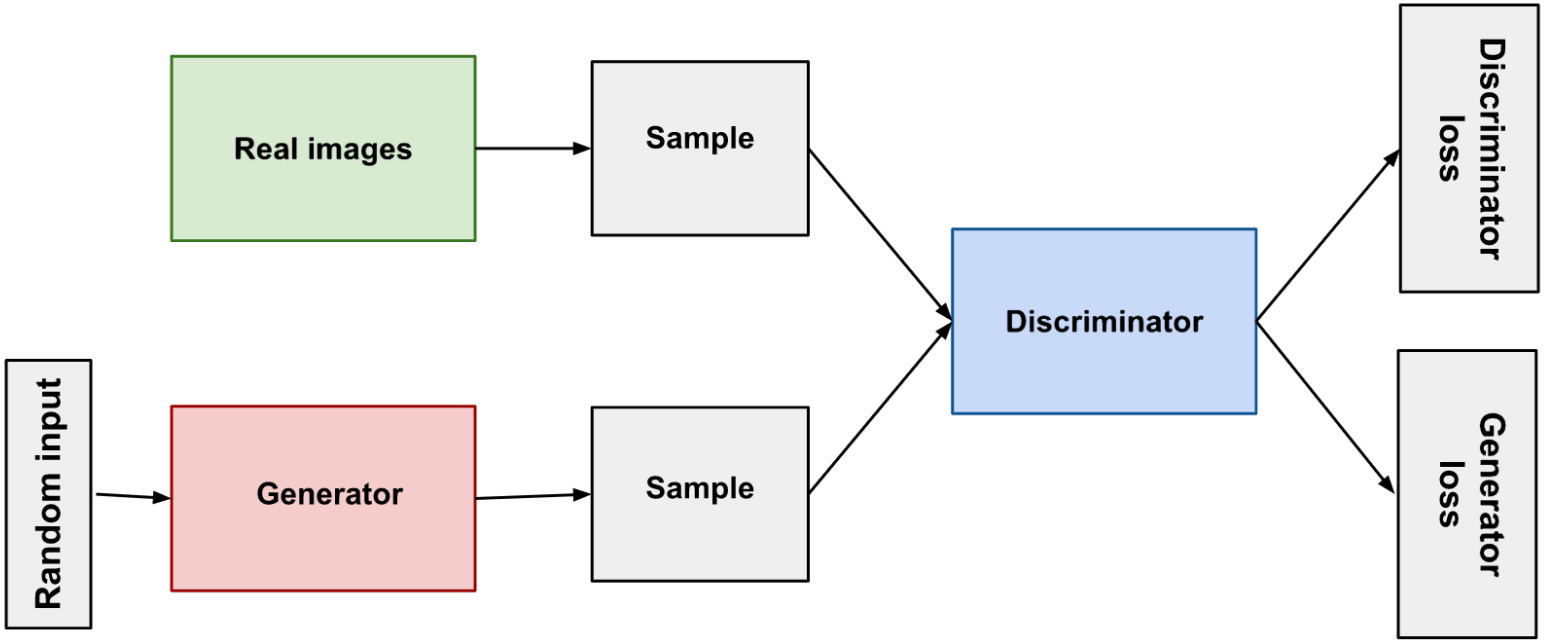
Generative Adversarial Networks (GANs)

Generative adversarial network (GAN) has two parts:

- The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.
- The **discriminator** learns to distinguish the generator's fake data from real data.


Generated Data
Discriminator
Real Data
FAKE
REAL
10
FAKE
REAL
REAL
REAL

GAN architecture



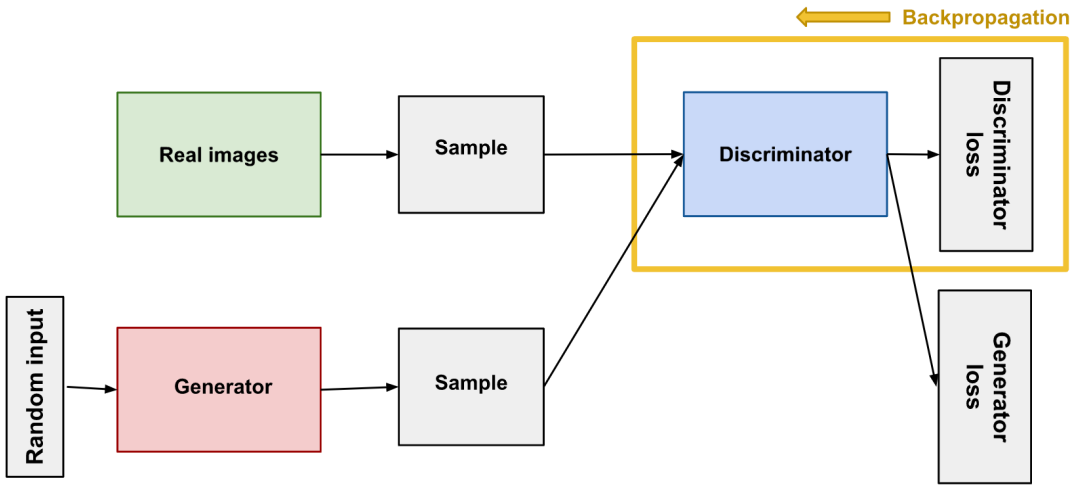
Discriminator Training

Discriminator is a classifier

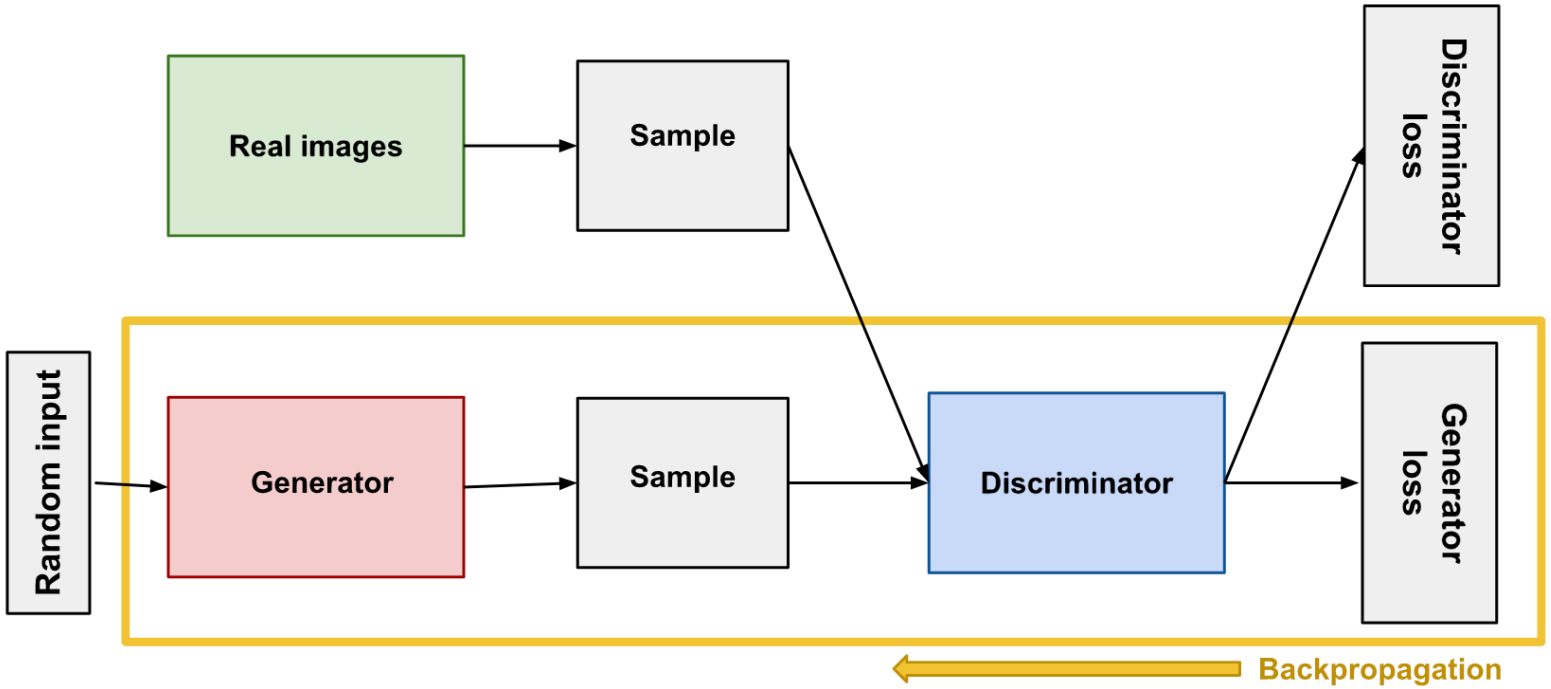
- Distinguishes real data from data created by the generator

Discriminator training:

- Classifies both real data and fake data from the generator
- Discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.
- Discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.



Generator Training



Generator Training

- Generator is not directly connected to the loss
- Generator feeds into the discriminator which then produces the output
- Generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake
- We don't want the discriminator to change during generator training (moving target)

Generator training:

1.Sample random noise.

2.Produce generator output from sampled random noise.

3.Get discriminator "Real" or "Fake" classification for generator output.

4.Calculate loss from discriminator classification.

5.Backpropagate through both the discriminator and generator to obtain gradients.

6.Use gradients to change only the generator weights.

GAN Training

- Alternating training of discriminator and generator
  1. The discriminator trains for one or more epochs.
  2. The generator trains for one or more epochs.
  3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.
- Convergence
  - As training progresses discriminator performance worsens
  - Eventually discriminator starts making random guesses
  - Generator get junk feedback

GAN Loss functions

- **Minimax loss**

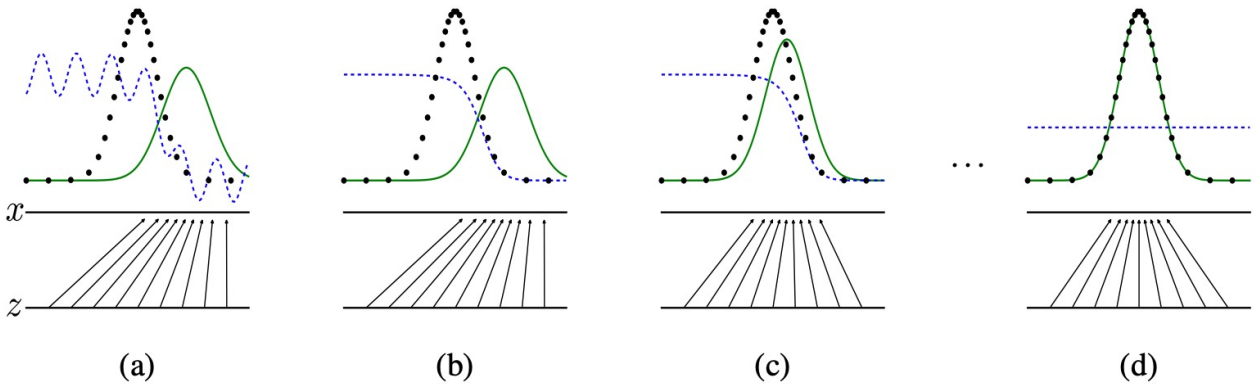
$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

- `D(x)` is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- `G(z)` is the generator's output when given noise z.
- `D(G(z))` is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- The formula derives from the cross-entropy between the real and generated distributions.

Min-max game

D and G play the following two-player minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$



GAN Training as Stochastic Gradient Descent

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

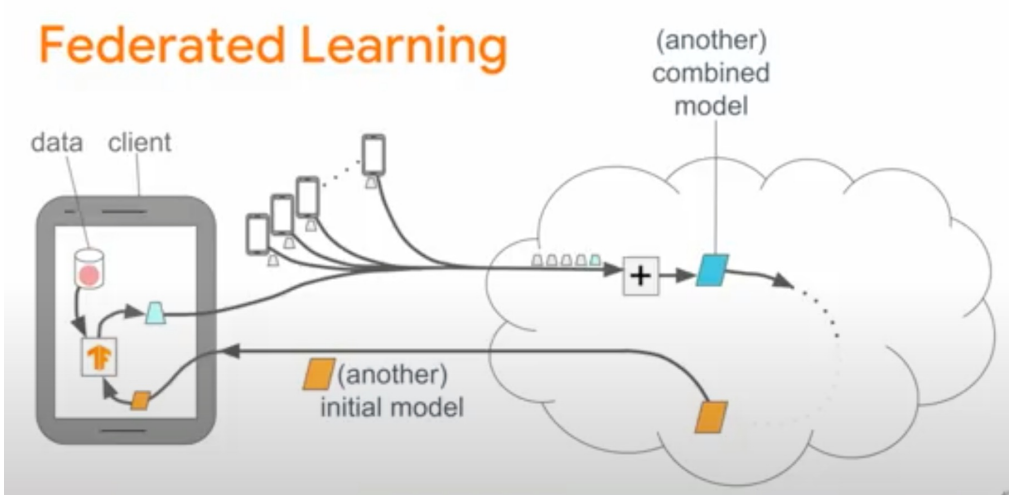
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Federated Learning

Bringing code to the data



- "Bringing the code to the data, instead of the data to the code"
- Addresses the fundamental problems of privacy, ownership, and locality of data
- Enables multiple parties to collaboratively learn a shared prediction model while keeping all the **training data private with the party**, decoupling the ability to do machine learning from the need to store data in the cloud

Learn together without sharing data

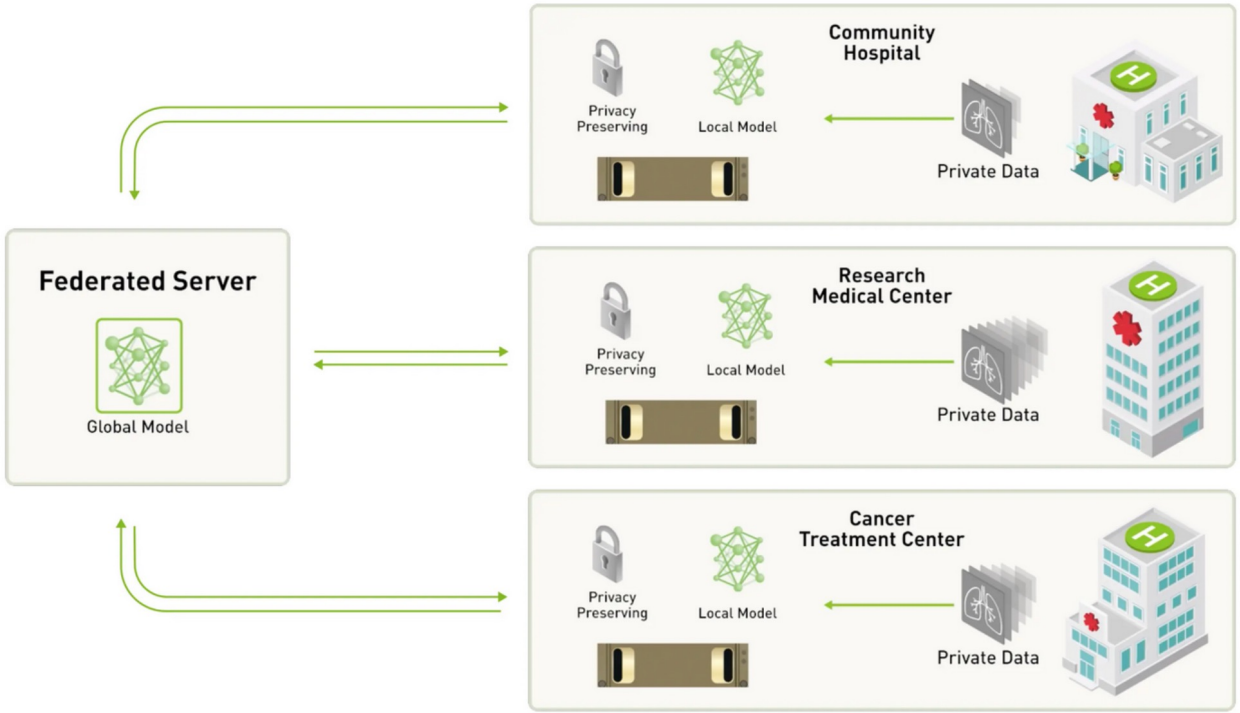
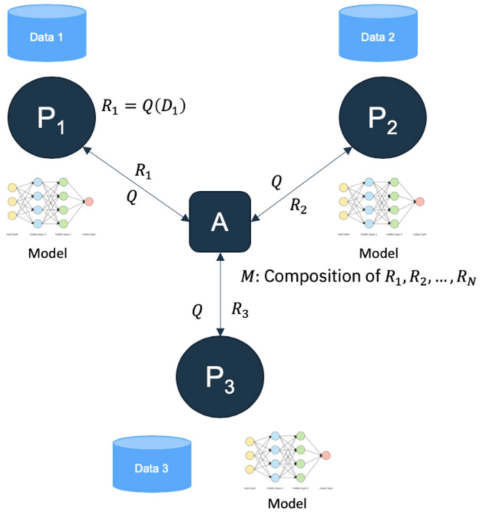


Image courtesy of NVIDIA Corporation

Federated Training

- Federated Learning is a **distributed machine learning** approach
  - Multiple parties collaborate and learn a shared prediction model
  - Data does not need to be shared and is kept private with the party
- Federated Learning is a **distributed machine learning**

Federated learning scenario with 3 parties



Federated Learning Challenges

- *Is FL another name for distributed DL ?*
- Federated Learning is a **distributed machine learning** approach with some unique challenges
  - **Non-IID:** The training data on a given client is typically based on the usage of the mobile device by a particular user, and hence any particular user's local dataset will not be representative of the population distribution.
  - **Unbalanced:** Similarly, some users will make much heavier use of the service or app than others, leading to varying amounts of local training data.
  - **Massively distributed:** We expect the number of clients participating in an optimization to be much larger than the average number of examples per client.
  - **Limited communication:** Mobile devices are frequently offline or on slow or expensive connections.

Federated Averaging Algorithm

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:

initialize w_0

for each round $t = 1, 2, \dots$ **do**

$m \leftarrow \max(C \cdot K, 1)$

$S_t \leftarrow$ (random set of m clients)

for each client $k \in S_t$ **in parallel do**

$w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$

$w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Run on client k*

$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)

for each local epoch i from 1 to E **do**

for batch $b \in \mathcal{B}$ **do**

$w \leftarrow w - \eta \nabla \ell(w; b)$

return w to server

$$n_k = |\mathcal{P}_k|$$

McMahan et al. Communication-efficient learning of deep networks from decentralized data. AAAI 2017

Federated Learning Protocol

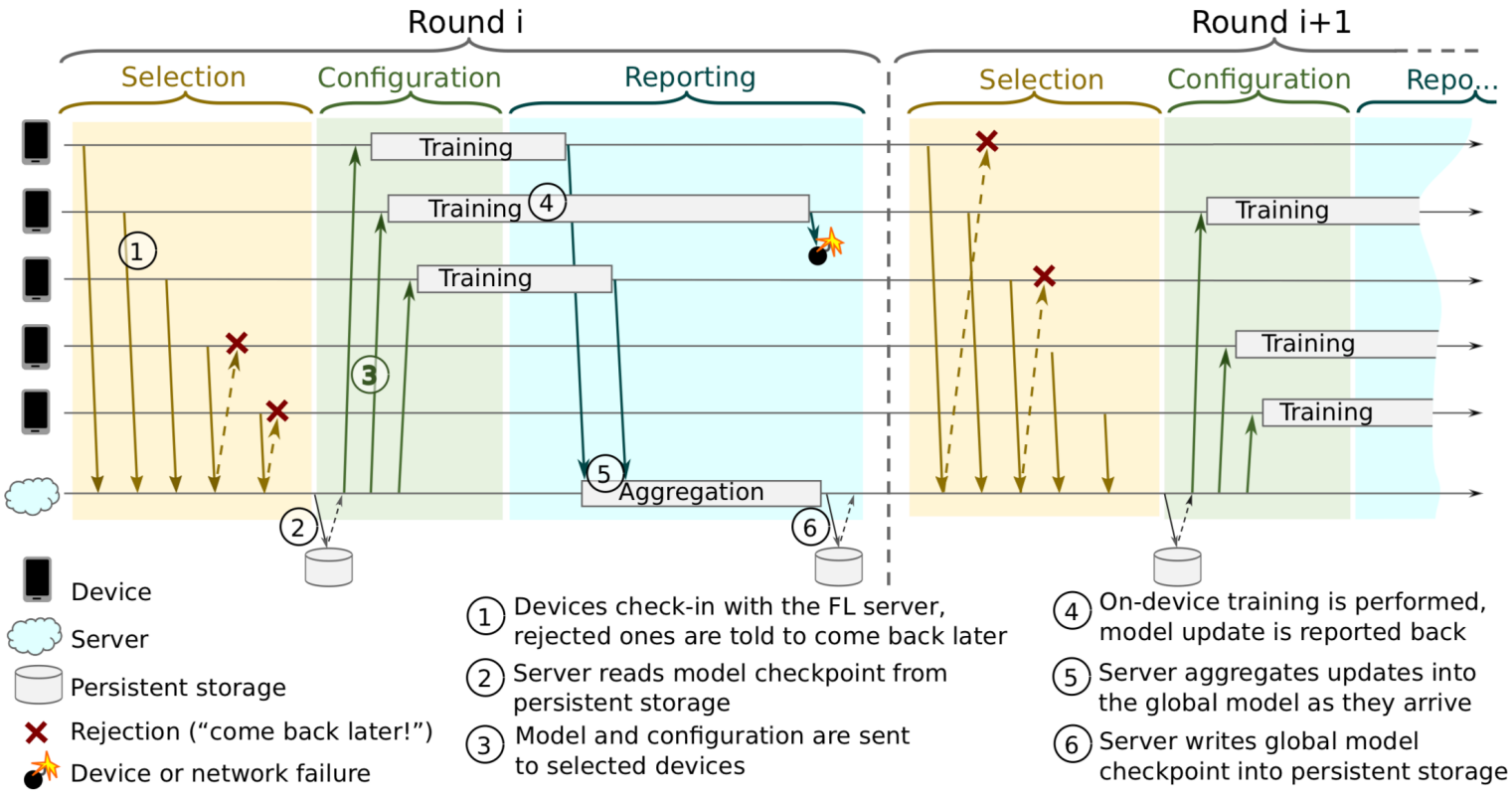
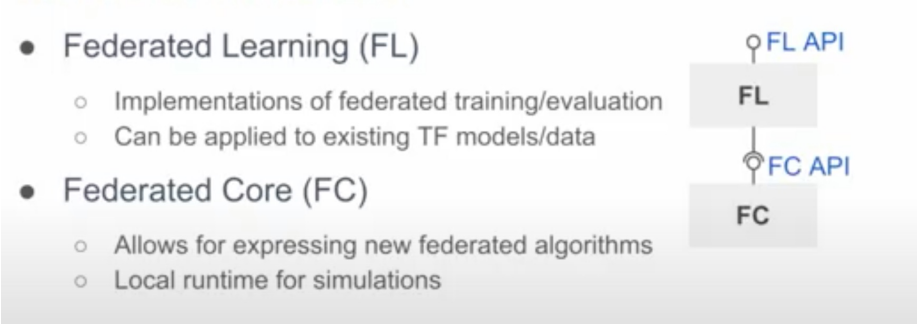


Figure 1: Federated Learning Protocol

Bonawitz et al. Towards federated learning at scale: System design. SysML 2019

TensorFlow Federated (TFF)

- TFF is an open source framework for federated learning on decentralized data
- TFF enables users to use their own model and data for
  - Doing simulations with standard FL algorithms (supported by TFF)
  - Experimenting with novel algorithms for FL
- 2-layers of TFF Interfaces



- TFF tutorials: https://www.tensorflow.org/federated/tutorials/
- TFF github repo: https://github.com/tensorflow/federated

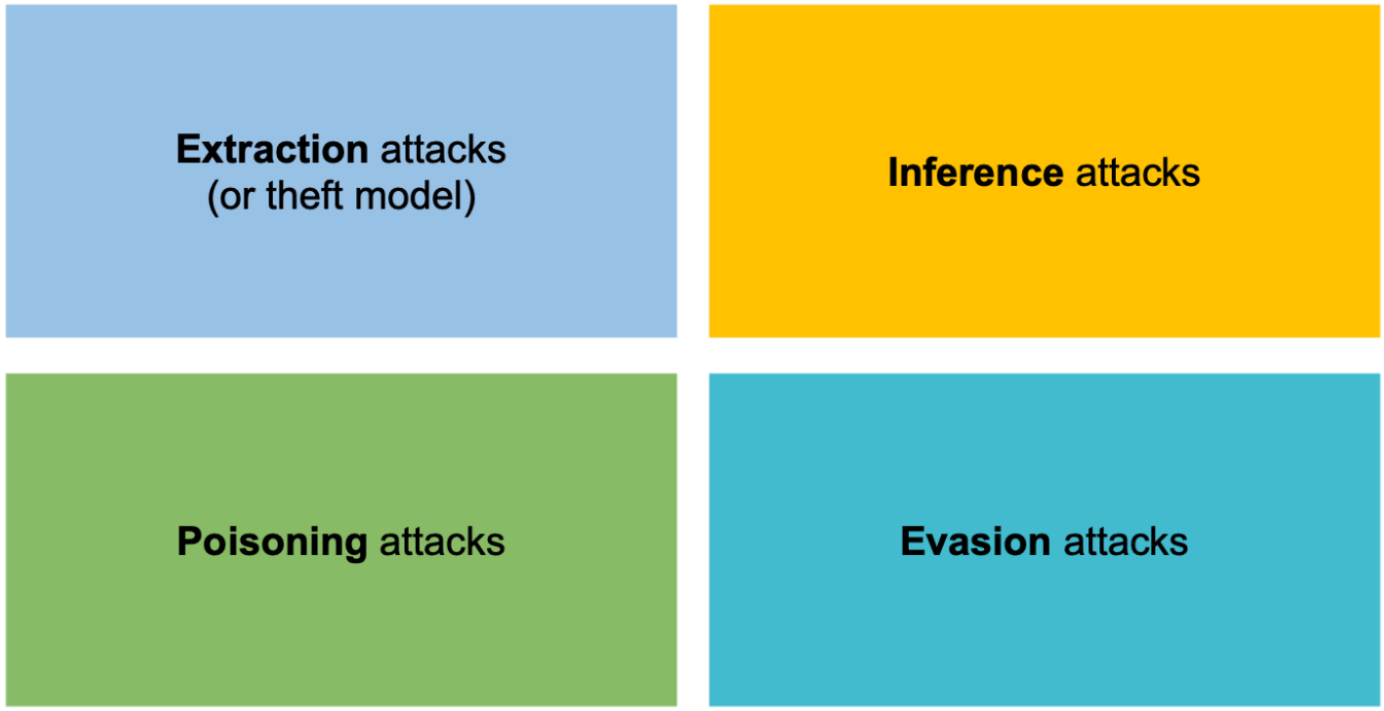
Efficient Federated Learning

- Local SGD: More local training before aggregation
  - Reduces communication overhead
  - Local model drift away from global model; more when data is non-IID across clients
  - FedAvg performance deteriorates
- Other fusion algorithms: FedProx, FedCurvature
  - A penalty term is added to the local training loss, compelling the client models to converge to a shared optimum
  - The loss function consists of two terms
    - Training loss on local data
    - Regularization loss to arrest model drift

FL Benchmarking

- https://arxiv.org/pdf/1812.01097.pdf
- Project page. LEAF: A benchmark for Federated Settings.

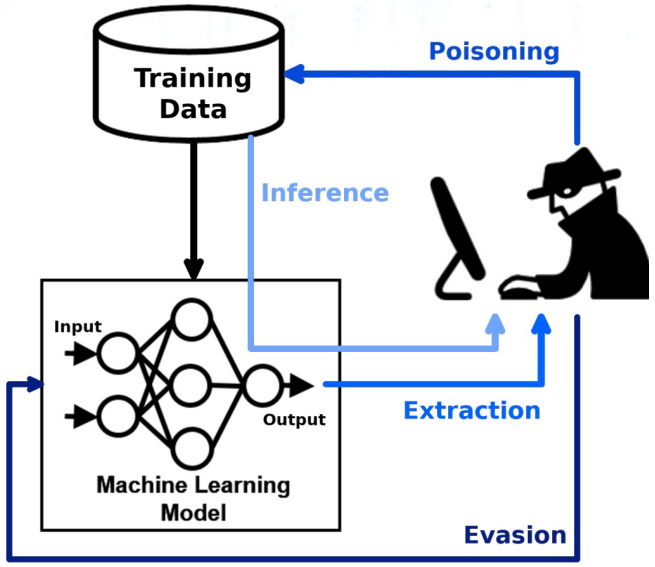
Types of attack on ML models



Adversarial Attacks

Adversarial Threats against AI and ML

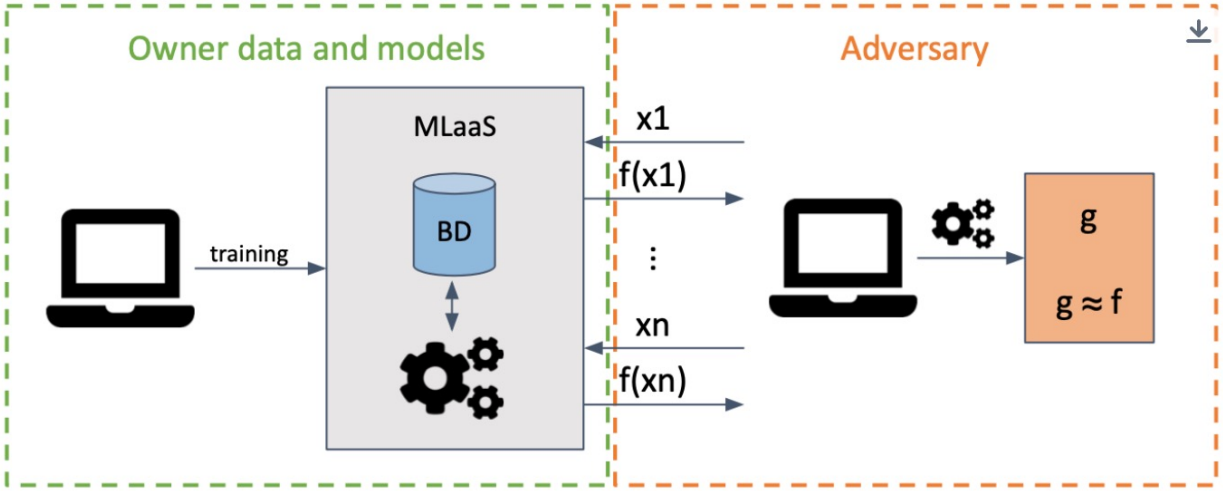
- Adversarial threats against machine learning models and applications:
  - **Evasion:** Modifying input to influence model
  - **Poisoning:** Modify training data to add backdoor
  - **Extraction:** Steal a proprietary model
  - **Inference:** Learn information on private data



Great and detailed overview: github.com/mitre/advmlthreatmatrix

Extraction Attacks

An adversary steals a copy of a remotely deployed machine learning model by making requests to the target model with inputs to extract as much information as possible and with the set of inputs and outputs train a model called substitute model

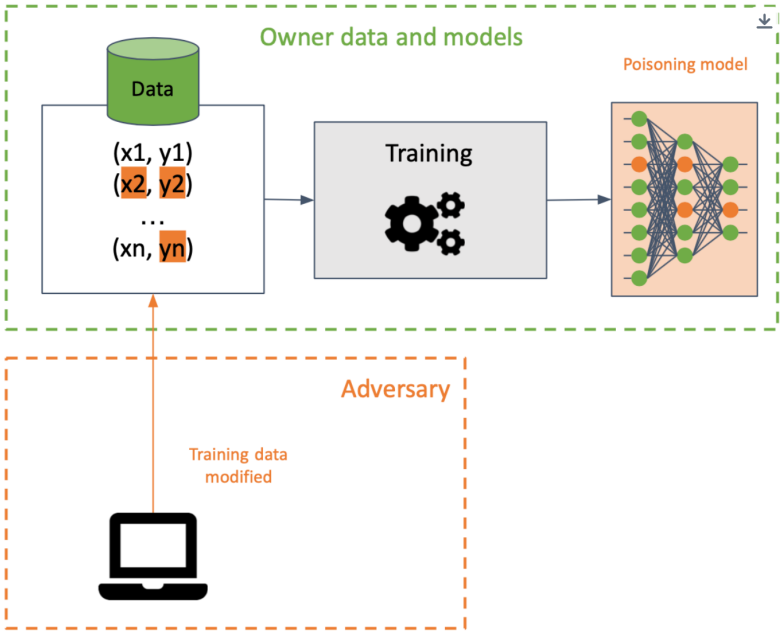


Extract model is hard, the attacker needs a huge compute capacity to re-training the new model with accuracy and fidelity, and substitute model is equivalent to training a model from the ground up.

https://hackernoon.com/adversarial-machine-learning-a-beginners-guide-to-adversarial-attacks-and-defenses

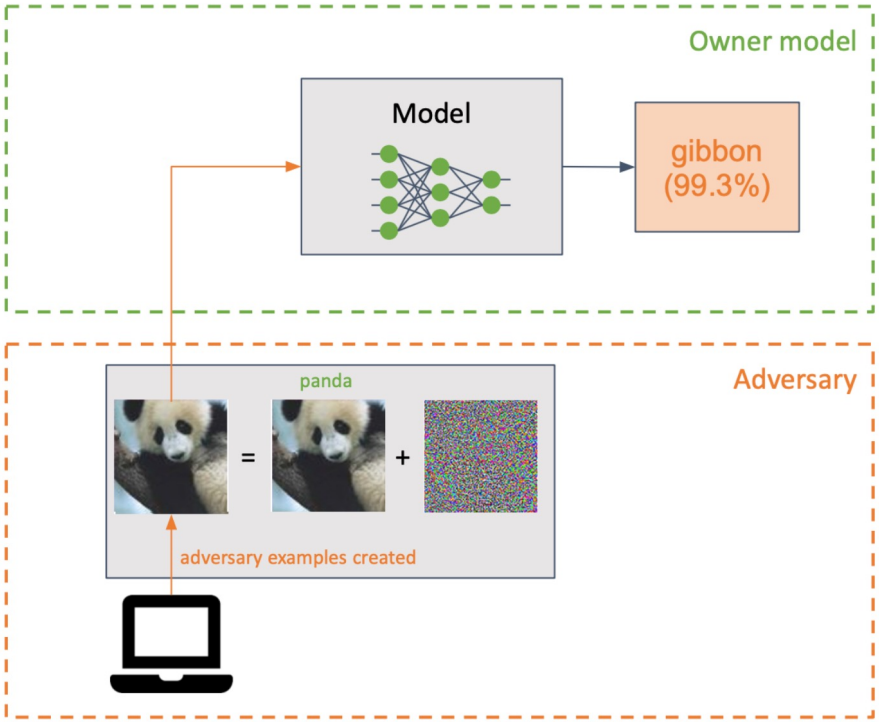
Poisoning Attacks

attacker inserting corrupt data in the training dataset to compromise a target machine learning model during training.



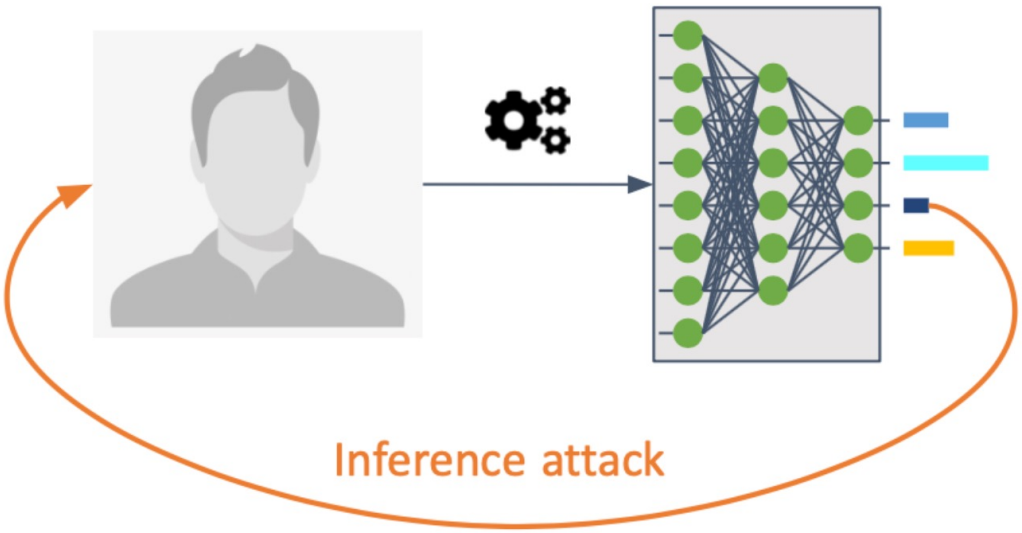
Evasion Attacks

adversary inserts a small perturbation into the input of a machine learning model to make it **classify incorrectly**



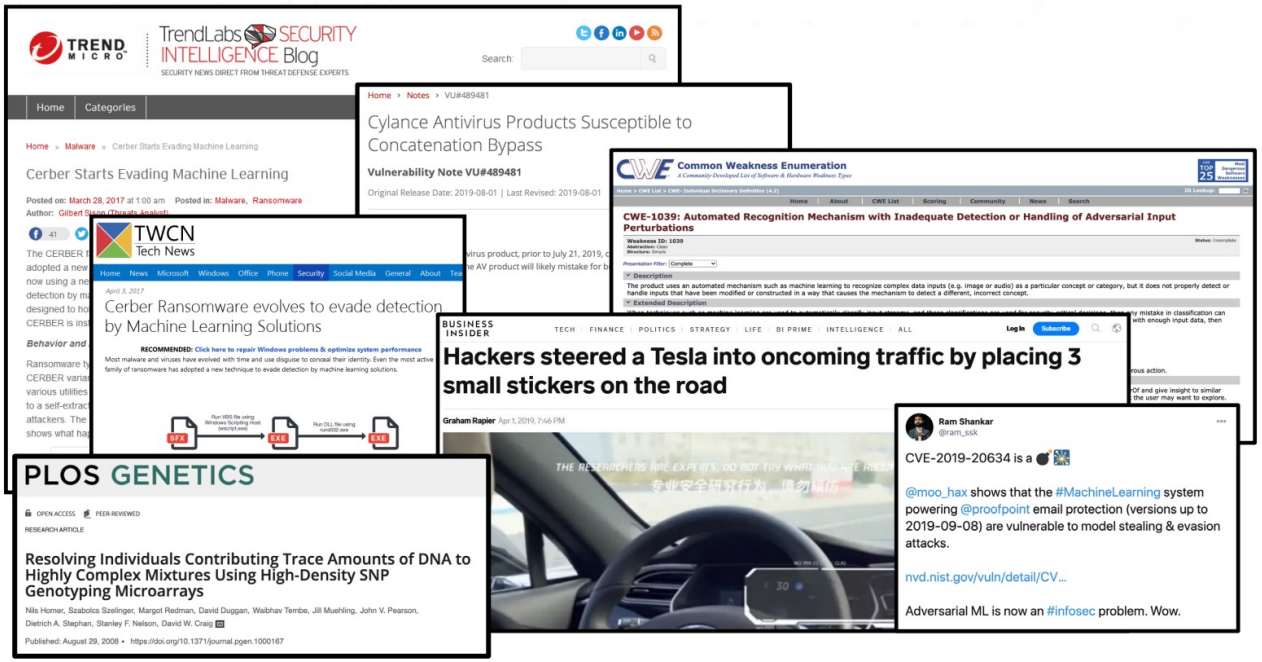
Inference Attacks

Intended for adversary to have knowledge of the model that was not explicitly intended to be shared.



- Membership Inference Attack (MIA)
- Property Inference Attack (PIA)
- Recovery training data

Real-world Adversarial Exploits



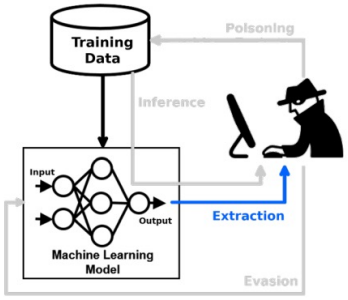
Selected examples:

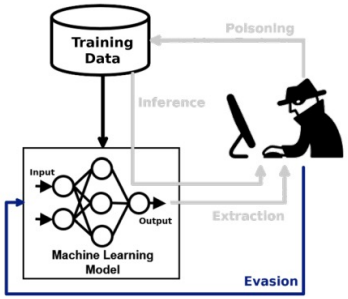
- Evasion of classification in antivirus products
  - → Undetected ransomware installs and encrypts your computer
- Real-world adversarial patches for evasion attacks on cars
  - → Losing control of autonomous vehicles leads to damages and injury
- Extraction of classification models to stage evasion attack against email protection system
  - → Bypassing email security systems increases chances of phishing attacks
- Revealing whether a person has HIV via membership inference on disease related models
  - → Leaking sensitive private information

Extraction and Evasion against Email Protection System – CVE-2019-20634

- https://nvd.nist.gov/vuln/detail/CVE-2019-20634
- https://github.com/moohax/Proof-Pudding
- https://github.com/moohax/Talks/blob/master/slides/DerbyCon19.pdf

- **CVE-2019-20634**
- Work of Will Pearce and Nick Landers
- Attacking an application leaking information on internal classification decisions
- Combination of two threats (e.g., extraction and evasion) increases impact and feasibility





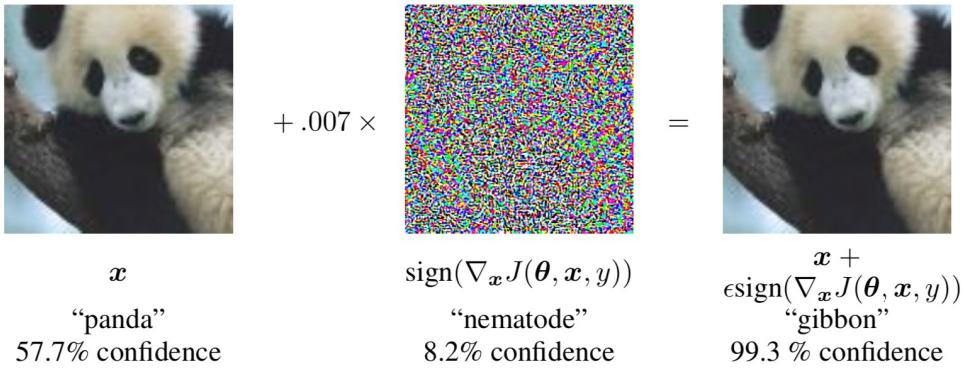
- **Step 1: Extraction of Classification**
- Create dataset of input label pairs by querying the application
- Train surrogate classification model on new dataset

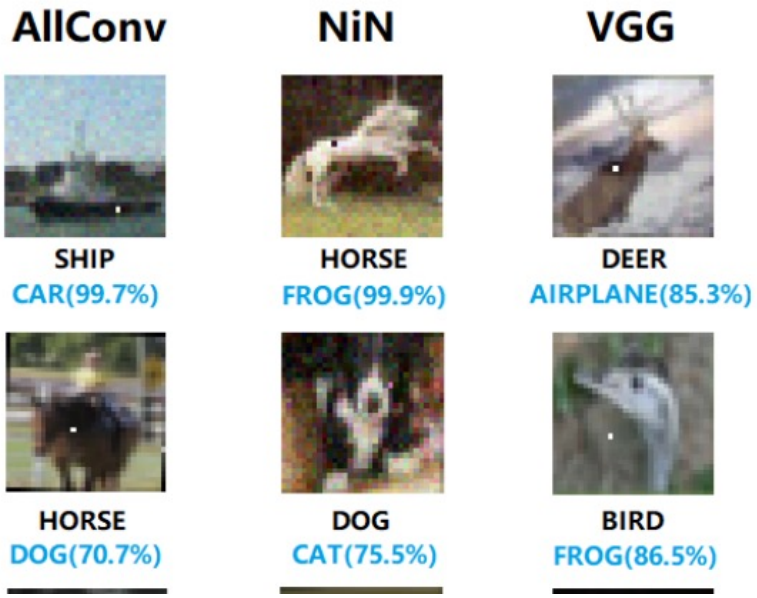
- **Step 2: Apply Classifier**
- Use insights on application's classification to craft adversarial examples
- Design adversarial examples to achieve attack goal, e.g., misclassification of spam email

Adversarial Robustness

Brittle AI

One pixel change is enough to fool DNNs





- ML models are extremely brittle
- Brittleness is a problem: security and safety implications
- Brittleness is an artifact of the model training algorithm

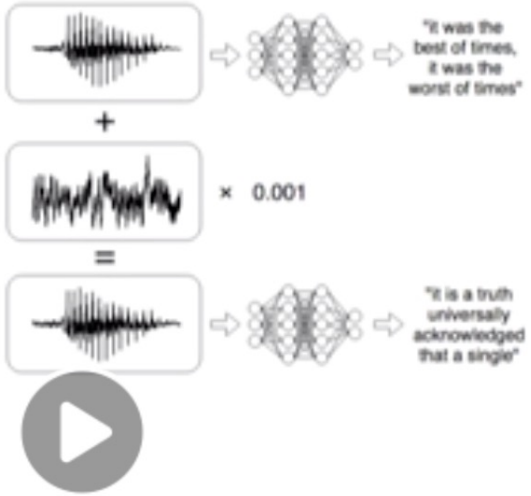
Explaining and harnessing adversarial examples. 2014

One Pixel Attack for Fooling Deep Neural Networks. 2019

Examples of Model Brittleness

→ Security

[Carlini Wagner 2018]:
Voice commands that are unintelligible to humans



[Sharif Bhagavatula Bauer Reiter 2016]:
Glasses that fool face recognition

With different glasses
You can be classified as
A different person: Airport security

Istrate et al. TAPAS: Train-less Accuracy Predictor for Architecture Search. 2018

Adversarial Training

- To create adversarially robust models
- Need to modify the model training objective

~~Standard~~ generalization: $\mathbb{E}_{(x,y)\sim D}\left[\max_{\boldsymbol{\delta}\in\boldsymbol{\Delta}} loss(\theta, x+\boldsymbol{\delta}, y)\right]$

Adversarially robust

- Train using adversarial examples
  - Delta needs to be imperceptible
- How to generate adversarial examples ?
  - How can we find perturbration which is imperceptible
- Bounding the LP-norm of the change of each individual pixel
- Train models to perform well in expectation on the worst-case examples from this distribution

Madry et al. Towards deep learning models resistant to adversarial attacks. 2017

Adversarial Training

- Regular adversarial training (RAT) is popular:

  $\min_{\boldsymbol{\theta}} \rho(\boldsymbol{\theta})$; $\rho(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}} \left[\max_{\|\tilde{\boldsymbol{x}}-\boldsymbol{x}\|_\infty \leq \epsilon} \mathcal{L}(f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}), y)\right]$

  1. Replace data with adversarial counterparts
  2. Update model with adversarial counterparts

- Projected Gradient Descent (PGD) for maximization

  $$\boldsymbol{x}^{t+1} = \Pi_{\epsilon-\text{ball}} \left(\boldsymbol{x}^t + \alpha \operatorname{sign}(\nabla_{\boldsymbol{x}^t} \mathcal{L}(f_{\boldsymbol{\theta}}(\boldsymbol{x}^t), y))\right)$$

  Projects to an ϵ-ball around $\boldsymbol{x}$ after every iteration

Projects to an ϵ-ball around $\boldsymbol{x}$ after every iteration

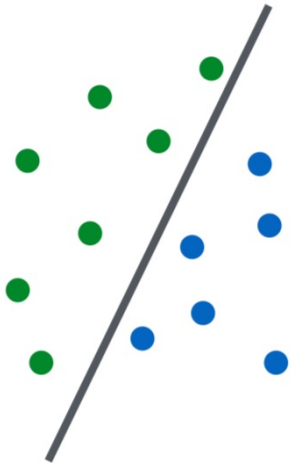
Model architecture	Natural training	Regular adversarial training
ResNet-50	1.1 hours	6.8 hours
WideResNet-28x10	2.2 hours	14.7 hours

CIFAR-10 for 155 epochs: 10-step PGD, $\epsilon = 8/255$, $\alpha = 2/255$

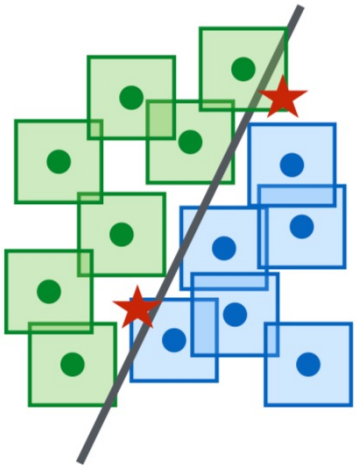
Each PGD iteration requires a forward and backward pass of the model which is **computationally expensive**

Adversarially trained model learns complex decision boundaries

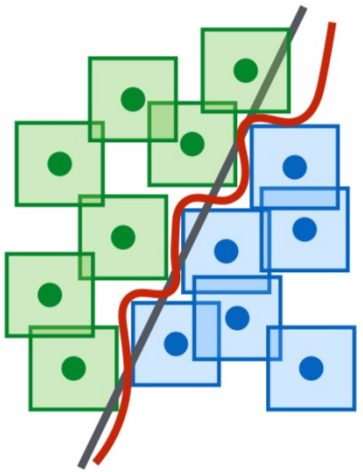
Linear decision boundary separates original training data



Linear decision boundary cannot separate the l_∞-balls (here, squares) around the data points



Classifier with a complicated decision boundary is robust to adversarial examples with bounded l_∞-norm perturbations

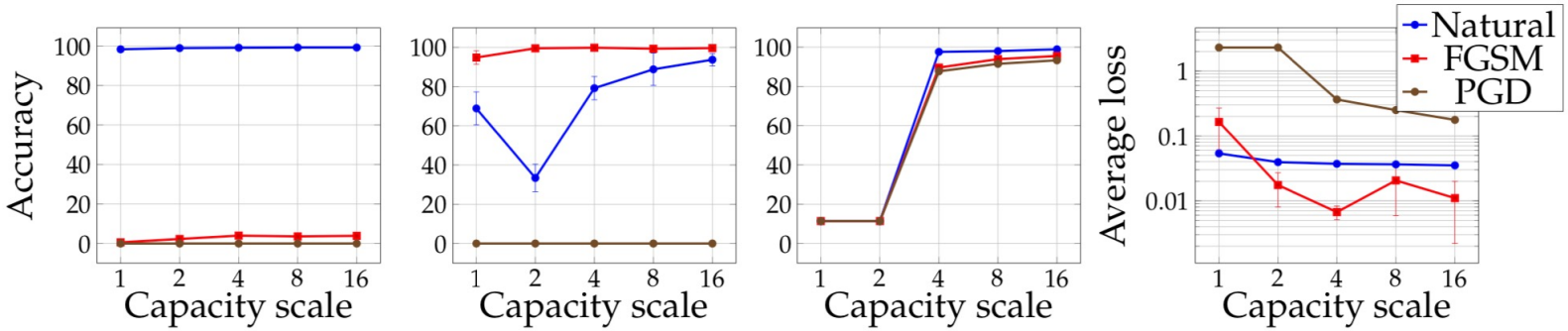


Adversarial robustness demands higher network capacity

Madry et al. Towards deep learning models resistant to adversarial attacks. 2017

Network Capacity and Adversarial Robustness

MNIST



CIFAR10

	Simple	Wide
Natural	92.7%	95.2%
FGSM	27.5%	32.7%
PGD	0.8%	3.5%

(a) Standard training

Simple	Wide
87.4%	90.3%
90.9%	95.1%
0.0%	0.0%

(b) FGSM training

Simple	Wide
79.4%	87.3%
51.7%	56.1%
43.7%	45.8%

(c) PGD training

Simple	Wide
0.00357	0.00371
0.0115	0.00557
1.11	0.0218

(d) Training Loss

Madry et al. Towards deep learning models resistant to adversarial attacks. 2017

Adversarial Robustness Toolbox (ART)

LF AI & DATA

TensorFlow Keras

PYTORCH mxnet

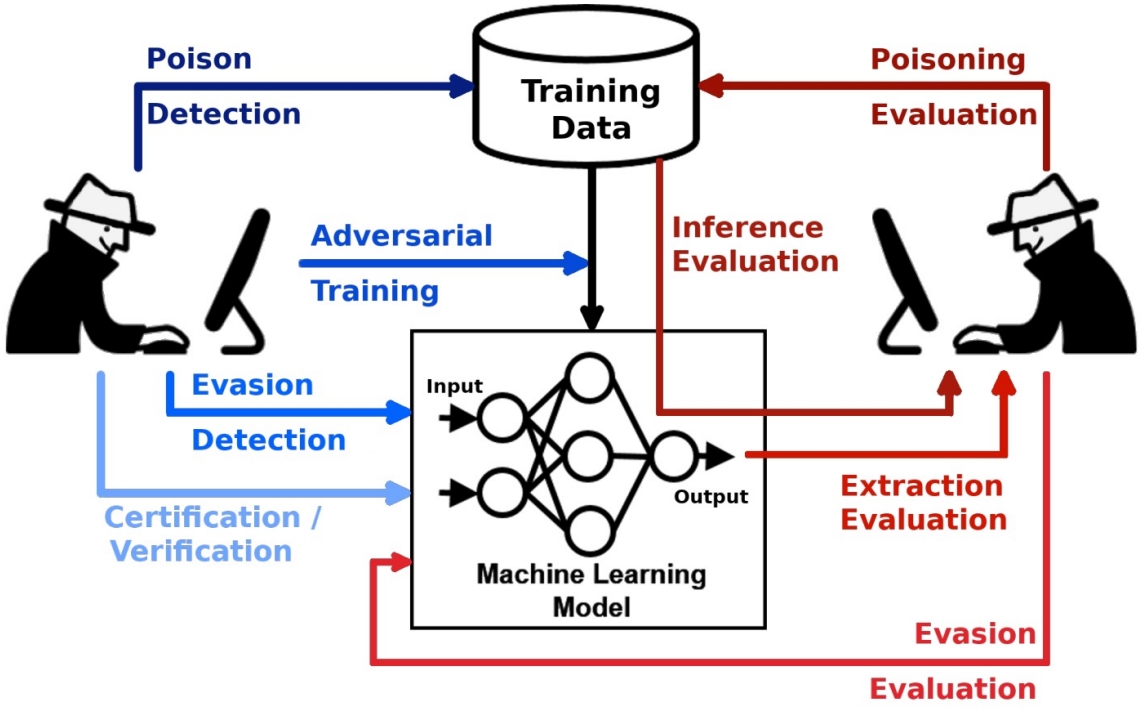
scikit learn GPy

dmlc XGBoost LightGBM CatBoost

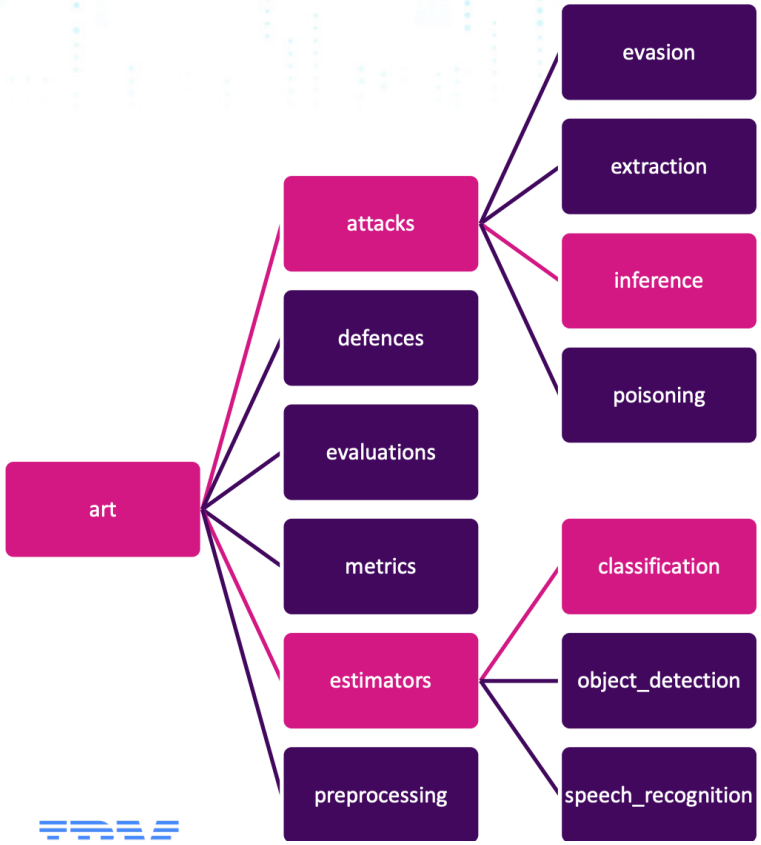
ART is an open-source library for machine learning security hosted by the Linux Foundation AI &Data

- github.com/Trusted-AI/adversarial-robustness-toolbox
- providing tools to developers and researcher
- Evaluating and Defending machine learning models and applications
- **All Tasks:** Classification, Object Detection, Automated Speech Recognition, etc.
- **All Frameworks:** TensorFlow, Keras, PyTorch, MXNet, scikit-learn, XGBoost, LightGBM, CatBoost, GPy
- **All Data:** images, tables, audio, video, multi-modal, etc.

Defending and Evaluating with ART



The Tools of ART (Selected Overview)



- **attacks**: attacks including white to black-box and physical attacks with patches
- **estimators**: ART's abstraction of the evaluated machine learning model
- **evaluations**: higher-level tools for evaluation of robustness
- **metrics**: tools for quantifying robustness and similar metrics
- **defences**: tools for defending against attacks including adversarial training, pre- and postprocessing
- **preprocessing**: modification of input data including Expectation over Transformations (EoT)
- Tools used in the following inference attack demonstration are **colored**

Adversarial Robustness Toolbox (ART)

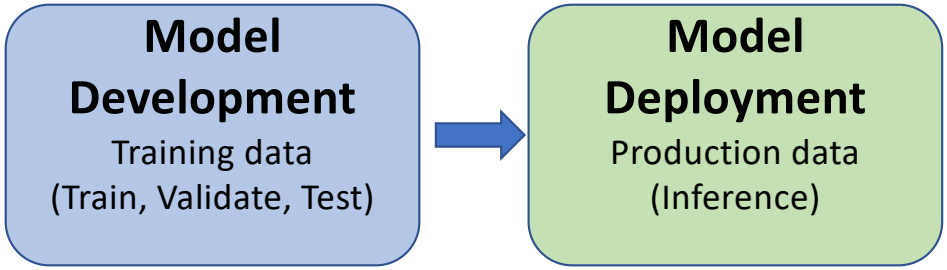
- https://github.com/Trusted-AI/adversarial-robustness-toolbox

Limitations of Adversarial Training

- Adversarial training is computationally expensive and slow
- Limited to centralized setting

Model Drift and Detection

ML Model: Development to Deployment

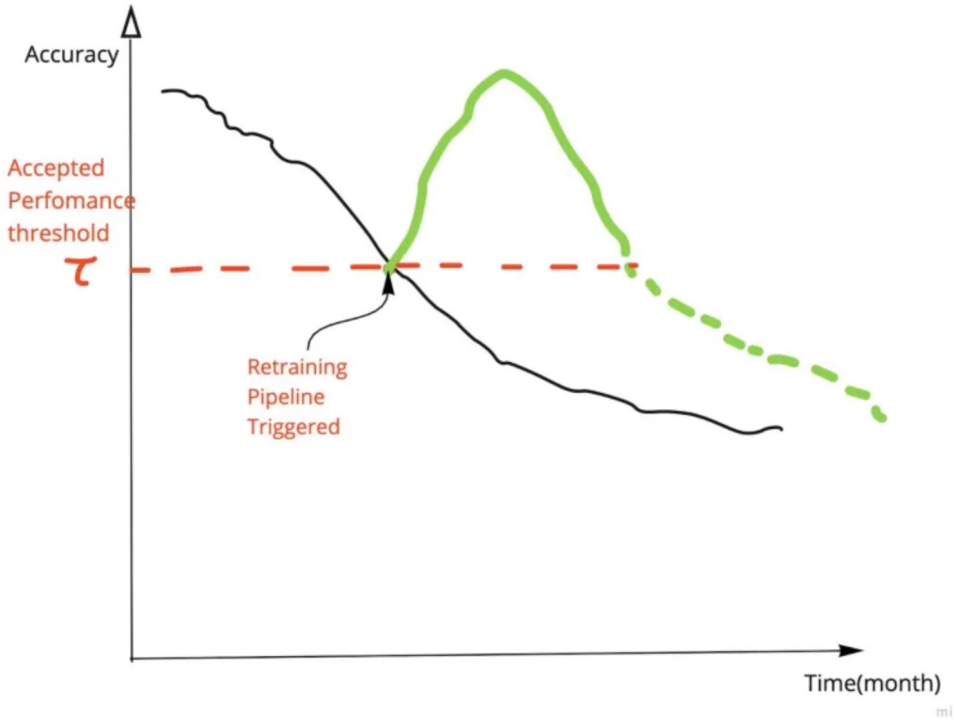


- The assumption underlying statistical ML
  - Training data distribution is representative of production data
  - Performance of ML model in production is same as predicted in training
- Will the model performance always remain same in production ?

Post deployment

- Model serving
  - Process of how your model is consumed in production
  - Depends on the business use cases
  - Several ways to productionalize your machine learning models, such as **model-as-service, model-as-dependency, and batch predictions**
- Model performance monitoring
  - Process of tracking the performance of your machine learning model based on live data in order to identify potential issues that might impact the business
  - Metrics to monitor: availability of the model, model prediction and performance (latency, throughput) on live data, and resource usage (compute and memory) of the ML system.
- Model re-training
  - Process to retrain the model with a new data distribution and/or new labels to ensure that the quality of your model in production is up to date
  - Triggered when change in model performance is detected

Model performance over time



Deployed model performance decays with time
Phenomenon is called **Model Drift**
Why is the model performance decaying with time?

The assumption underlying machine learning model development

- Training data distribution is representative of production data
- Performance of ML model in production is same as predicted in training

Model Drift: Change in Model Performance Over Time

- Production data gradually changes over time
- Performance of ML model may deteriorate over time
- **Model Drift** (also known as model decay) refers to the degradation of a model's prediction power due to changes in the environment, and thus the relationships between variables.
- Causes
  - Data drift: change in distribution, emergence of new variety, new features
    - Cars, fashion items, baby names
    - Change in proportion of different customer segments
  - Concept drift: change in statistical properties of the target variable
    - Data does not change but assigned classes changes
    - Socioeconomic reasons, change in laws

Data drift

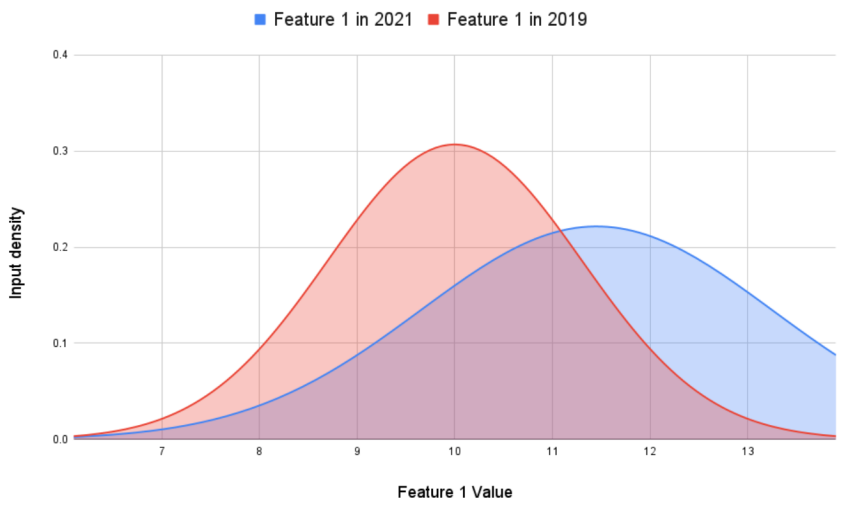
- Data drift (covariate shift) is a change in the statistical distribution of production data from the baseline data used to train or build the model. Data from real-time serving can drift from the baseline data due to:
  - Changes in the real world
  - Training data not being a representation of the population,
  - Data quality issues like outliers in the dataset.
  - Weak slices during model training becoming prominent in production

Concept drift

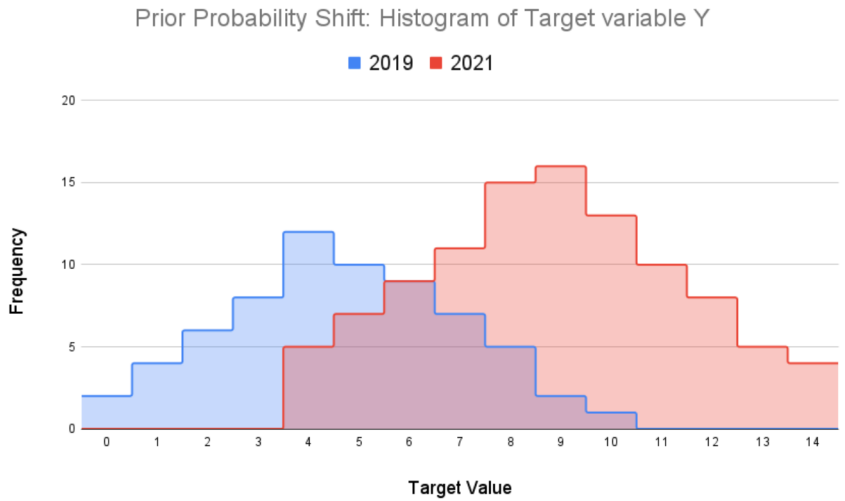
- Concept drift is a phenomenon where the statistical properties of the target variable you're trying to predict changes over time. This means that the concept has changed but the model doesn't know about the change.
- Concept drift happens when the original idea your model had about the target class changes. For example, you build a model to classify positive and negative sentiment of tweets around certain topics, and over time people's sentiment about these topics changes. Tweets belonging to positive sentiment may evolve over time to be negative.

Types of Data Drift

- *Covariate Shift: Pdevelop(Y|X)=Pprod(Y|X) but Pdevelop(X) ≠Pprod(X)*



- *Prior Probability Shift: Pdevelop(Y|X)=Pprod(Y|X) but Pdevelop(Y) ≠Pprod(Y)*



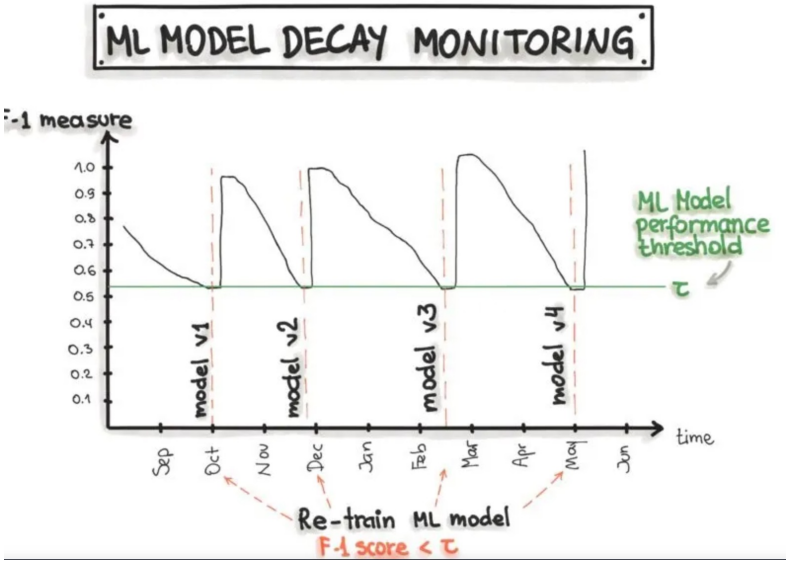
https://towardsdatascience.com/data-drift-part-1-types-of-data-drift-16b3eb175006

Retraining Strategy

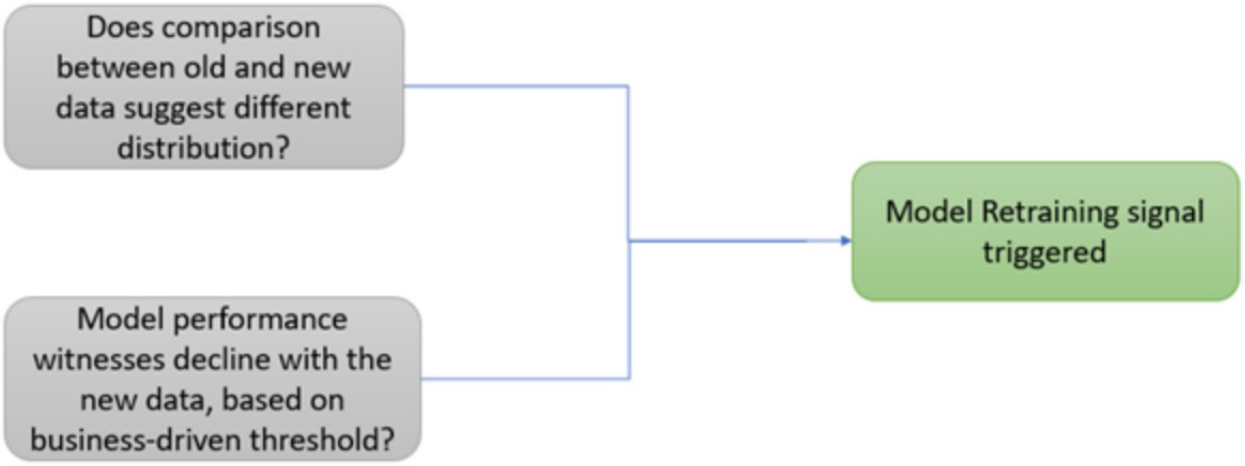
Questions	Answers
When should a model be retrained?	– Periodic training – Performance-based trigger – Trigger based on Data changes – Retraining on demand
How much data is needed for retraining?	– Fixed window – Dynamic window – Representative Subsample selection
What should be retrained?	– Continual learning vs Transfer learning – Offline(batch) Vs Online(Incremental)
When to deploy your model after retraining?	– A/B testing

When should you retrain a model?

- Retraining based on time intervals
- Performance based trigger



- Trigger based on data changes



Continual learning

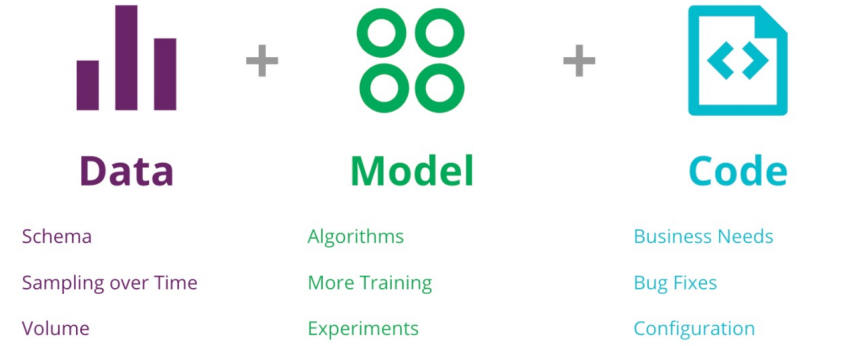


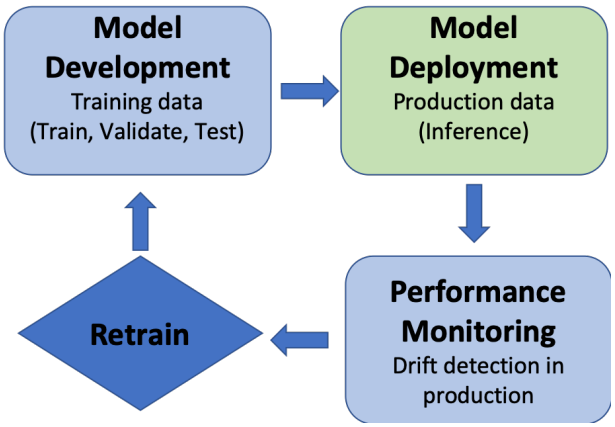
Figure 1: the 3 axis of change in a Machine Learning application – data, model, and code – and a few reasons for them to change

- Continuous training is an aspect of machine learning operations that automatically and continuously retrains machine learning models to adapt to changes in the data before it is redeployed. The trigger for a re-build can be data change, model change, or code change.

https://neptune.ai/blog/retraining-model-during-deployment-continuous-training-continuous-testing

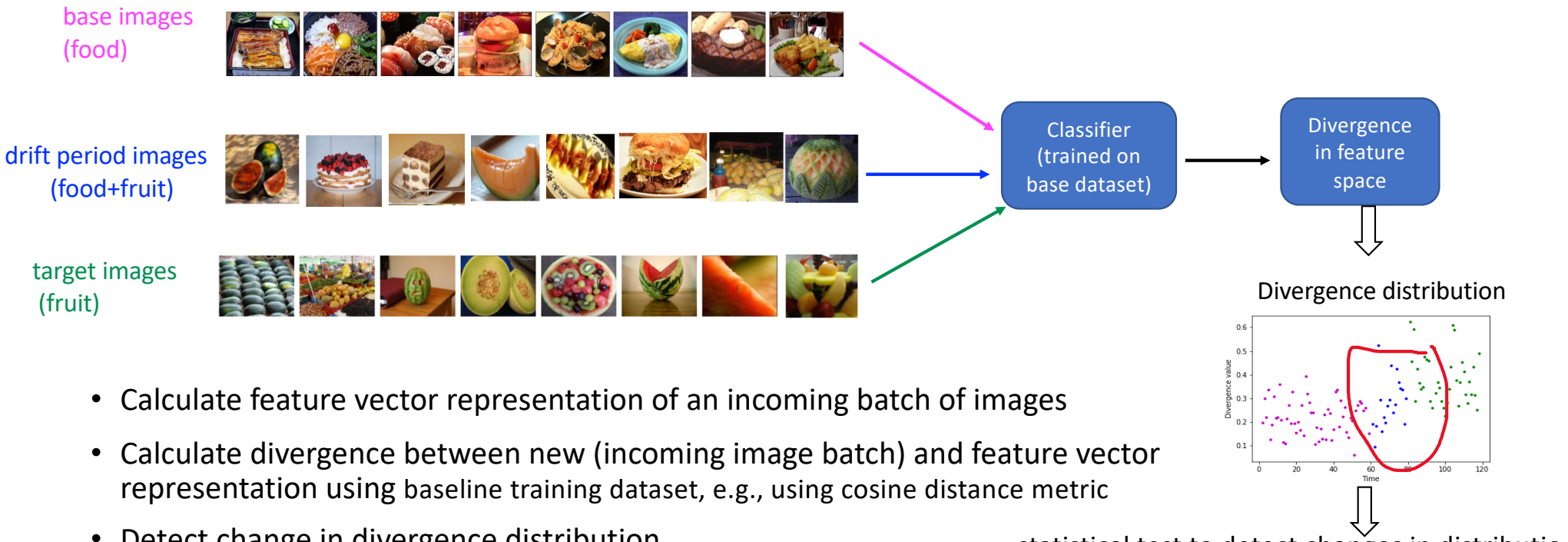
How to address data drift in ML ?

- Drift detection and retraining
  - Supervised drift detection: availability of labeled production data
    - Labeling is expensive and time consuming
  - Unsupervised drift detection: change detection in feature distribution of data
- Develop ML models robust to data drift



- *We are interested in detecting data drift in the absence of labels*
  - Detect changes in confidence distribution of classifier (Raz et al. 2019)
  - Exploit network structure and embeddings (Dube et al. 2020)

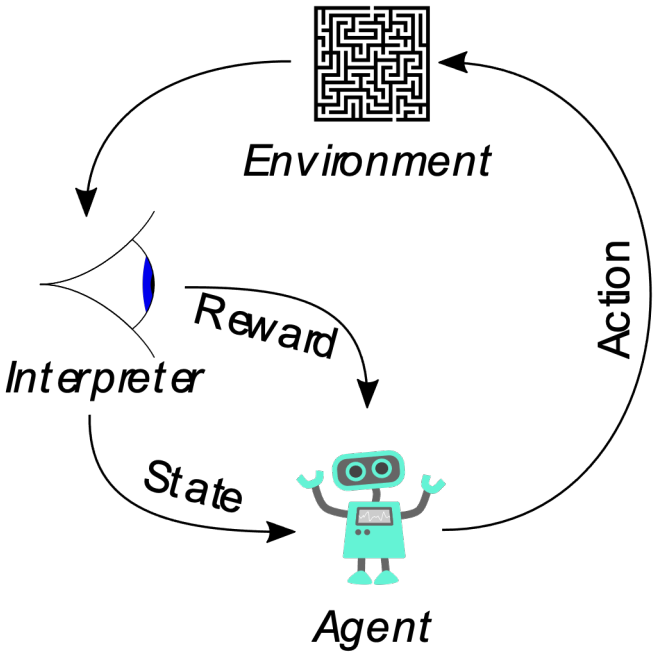
Drift Detection Methodology



- Calculate feature vector representation of an incoming batch of images
- Calculate divergence between new (incoming image batch) and feature vector representation using baseline training dataset, e.g., using cosine distance metric
- Detect change in divergence distribution
  - Divergence distribution as a surrogate for production data distribution

Dube et al.. Automated detection of drift in deep learning based classifiers performance using network emdeddings. EDSMLS 2020.

Reinforcement Learning (RL)



Agent reacts sequentially with an environment with the goal to maximize cumulative reward
Agent tries to learn efficient representations of the environment from high-dimensional sensory input
Environment is only partially observed and interpreted

Reinforcement Learning Challenges:

1. Credit-assignment problem: Contribution of each action for the reward is not known
2. Rewards can be probabilistic : can only be estimated approximately in data-driven manner
3. Model generalization is difficult: very large number of states
4. Exploration vs exploitation tradeoff
5. Inability to gather sufficient data in real settings

By Megajuice - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=57895741

Terminology

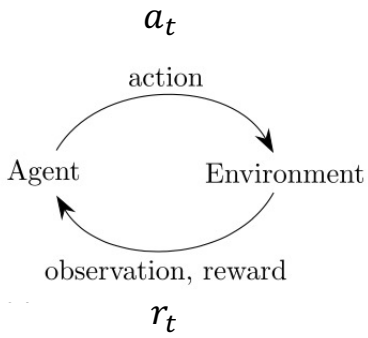
- Episodic Tasks
- Continuous Tasks
- Exploration
- Exploitation
- $\epsilon - greedy$ policy

RL Formulation

- Action a_t is selected using some policy π

  π is a mapping from states s_t to actions a_t

  $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ cumulative discounted reward at t



- Action-value function $Q^{\pi}(s,a) = \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$,

  Expected return for taking action a in state s and then following π

- Optimal action-value function:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

Maximum action value for state s and action achievable by any policy.

- Optimal action-value functions obeys Bellman's equation:

$$Q^*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

Reward r can be dependent on (s,a,s')
Probability distribution on the state s' reachable by taking action a in s

Q-learning

- In reinforcement learning we represent the (optimal) action-value function using a function approximator which we try to learn
- A neural network (Q-network) can be used as function approximator

$$Q(s,a) = Q(s,a;\theta)$$

Action-value function

Q-network (function approximator of action-value function; e.g., a neural network with parameters θ)

- The parameters θ of the *Q-network* are optimized so as to approximately solve the Bellman equation.

DQN Algorithm

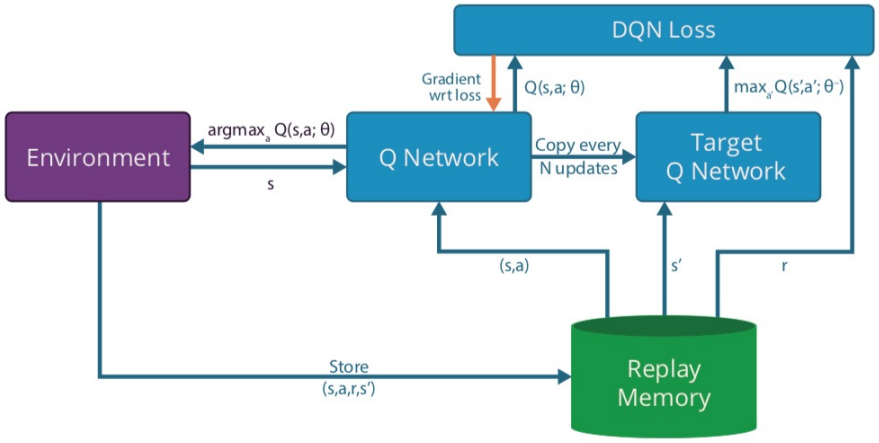


Figure 1. The DQN algorithm is composed of three main components, the Q-network ($Q(s, a; \theta)$) that defines the behavior policy, the target Q-network ($Q(s, a; \theta^-)$) that is used to generate target Q values for the DQN loss term and the replay memory that the agent uses to sample random transitions for training the Q-network.

DQN differs from Q-Learning in two ways:

1. DQN uses experience replay

At each time-step t during an agent's interaction with the environment it stores the experience tuple in replay memory

$$e_t = (s_t, a_t, r_t, s_{t+1})$$ Experience tuple

$$D_t = \{e_1, ..., e_t\}$$ Replay memory

2. Use of 2 separate Q-networks

$$Q(s, a; \theta) \text{ and } Q(s, a; \theta^-)$$

Q-network — Target Q-network

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right]$$

R. Nair et al. Massively Parallel Methods for Deep Reinforcement Learning. 2015

Why Experience Replay Helps ?

- Prevents instability in Q-learning
- Reuse of data
- Removes correlation in input batch samples
  - Prevents overfitting
- Experience replay may also help to prevent overfitting by allowing the agent to learn from data generated by previous versions of the policy.

GORILA

- Asynchronous training of reinforcement learning agents in a distributed setting
- Gorila agent parallelizes the training procedure by separating out learners, actors and parameter server
- Gorila components:
  - Actors (multiple)
  - Experience Replay Memory (local and global)
  - Learners (multiple)
  - Parameter server (sharded)

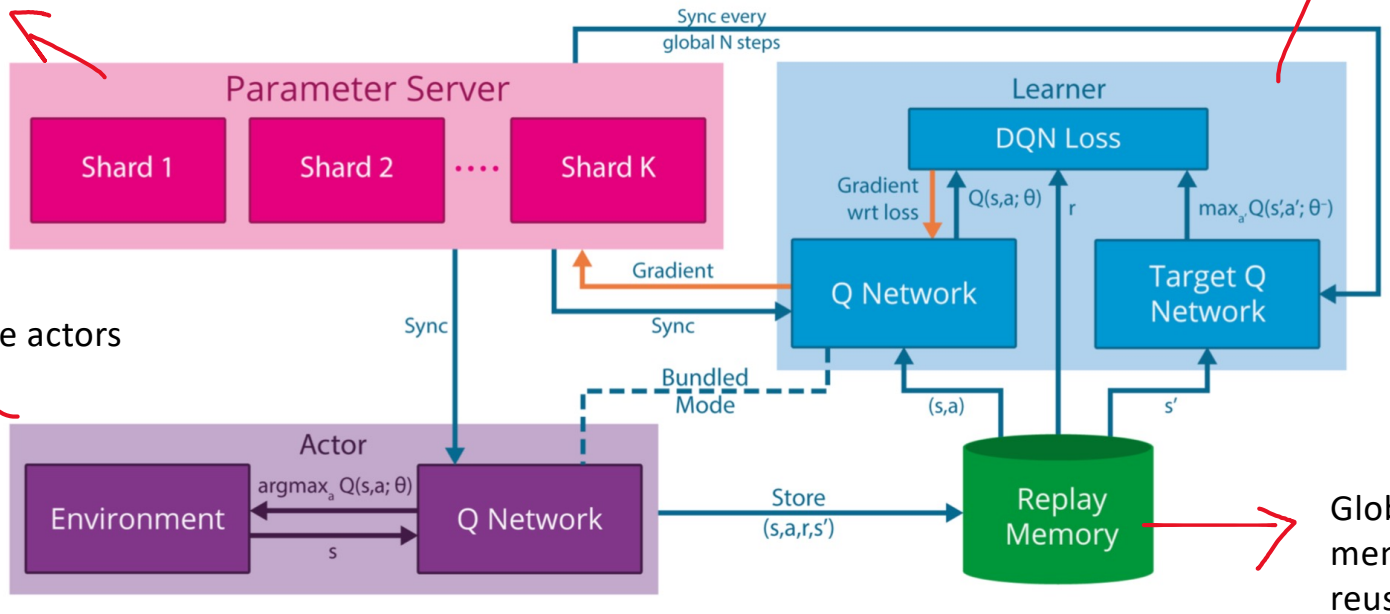
Nair et al. Massively Parallel Methods for Deep Reinforcement Learning. 2015

GORILA Architecture

scalable distributed training
using sharded parameter server

asynchronous stochastic gradient descent algorithm

multiple learners



multiple actors

Globally share replay memory increases data reuse

Figure 2. The Gorila agent parallelises the training procedure by separating out learners, actors and parameter server. In a single experiment, several learner processes exist and they continuously send the gradients to parameter server and receive updated parameters. At the same time, independent actors can also in parallel accumulate experience and update their Q-networks from the parameter server.

Nair et al. Massively Parallel Methods for Deep Reinforcement Learning. 2015

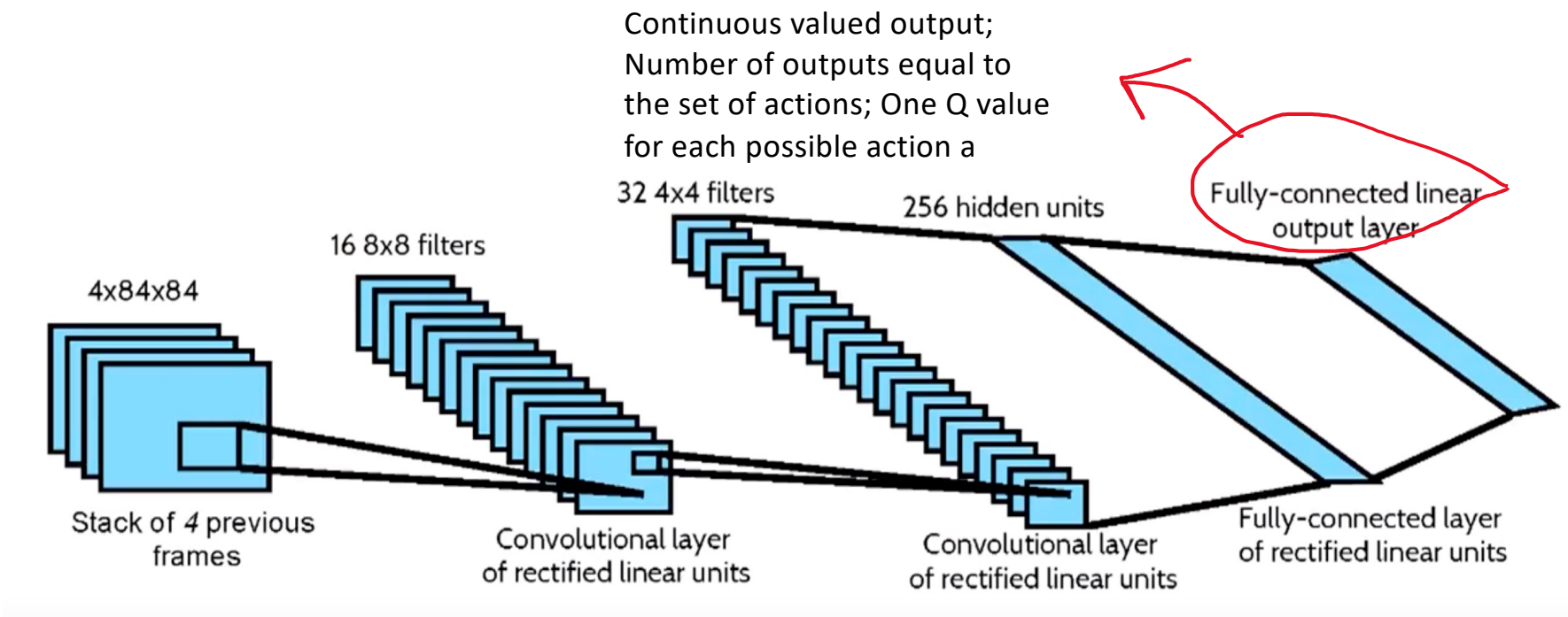
Parallelization in GORILA

- Parallel acting to generate large quantities of data into a global replay database and then process that data with a single serial learner
- A single actor generating data into a local replay memory, and then have multiple learners process this data in parallel
- To avoid any individual component from becoming a bottleneck, the Gorila architecture in general allows for arbitrary numbers of actors, learners, and parameter servers to both generate data, learn from that data, and update the model in a scalable and fully distributed fashion
- Bundled mode: $N_{act} = N_{learn}$
  - one-to-one correspondence between actors, replay memory, and learners
  - Each bundle
    - an actor generating experience
    - a local replay to store that experience
    - a learner that updates parameters based on samples of experience from the local replay memory

GORILA: Experimental Evaluation

- 49 Atari 2600 games
- An agent must learn to play the games directly from 210 × 160 RGB video input
- Only the changes in the score provided as rewards
- 210 × 160 RGB images preprocessed by downsampling them to 84 × 84
- Input consists of 4 image frames
- Number of parameter servers = 31
- Number of learners = Number of actors = 100
- Replay memory buffer size = 1 million frames
- Epsilon greedy policy (epsilon annealed from 1 to 0.1 over first one million global updates)

ATARI Convolutional Neural Network



One output per action: expected action-value for that action Q(s,a)

GORILA: Experimental Evaluation

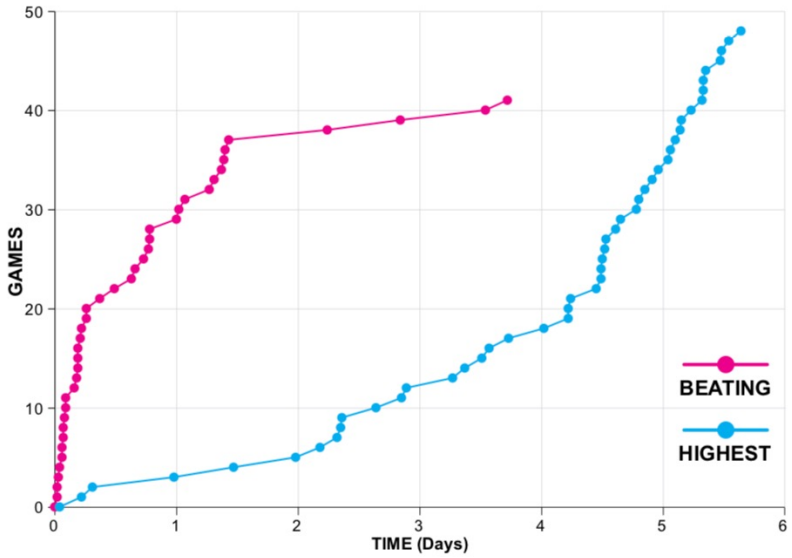


Figure 5. The time required by Gorila DQN to surpass single DQN performance (red curve) and to reach its peak performance (blue curve).

- Gorila DQN surpassed the best single GPU DQN scores on 19 games in 6 hours, 23 games in 12 hours, 30 in 24 hours and 38 games in 36 hours (red curve)
- Some games Gorila DQN achieved its best score in under two days
- Most of the games the performance keeps improving with longer training time (blue curve)

Nair et al. Massively Parallel Methods for Deep Reinforcement Learning. 2015

Prioritized Experience Replay

- Replaying all transitions with equal probability is highly suboptimal
- Focus learning on the most 'surprising' experiences
- *More frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error.*
- TD error:

$$\delta_i = r_t + \gamma \max_{a \in \mathcal{A}} Q_{\theta^-}(s_{t+1}, a) - Q_\theta(s_t, a_t)$$

Stochastic prioritization

- Probability distribution for sampling experiences
- Experience tuple: $(s_t, a_t, r_t, s_{t+1}, |\delta_t|)$

- A *rank* based method: $p_i = 1/\text{rank}(i)$ which sorts the items according to $|\delta_i|$ to get the rank.
- A *proportional* variant: $p_i = |\delta_i| + \epsilon$, where ϵ is a small constant ensuring that the sample has some non-zero probability of being drawn.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

https://danieltakeshi.github.io/2019/07/14/per/

Ape-X

- Distributed Prioritized Experience Replay

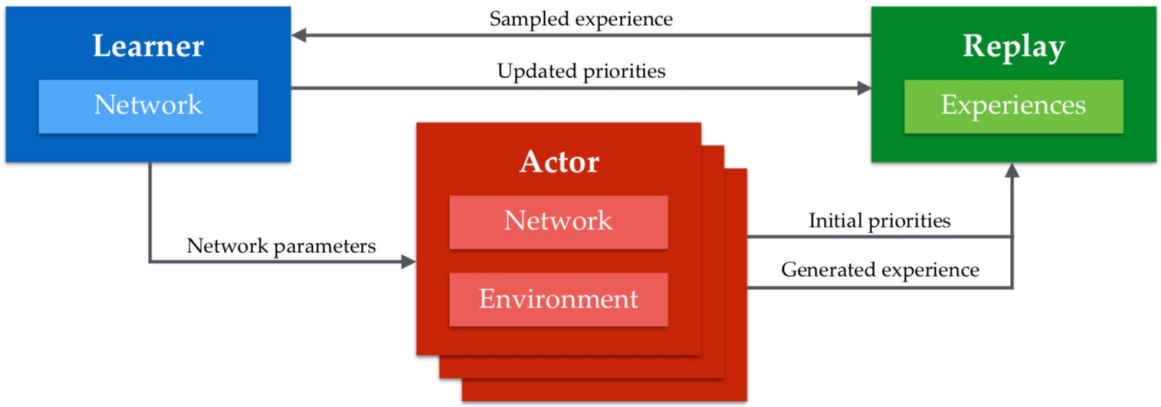


Figure 1: The Ape-X architecture in a nutshell: multiple actors, each with its own instance of the environment, generate experience, add it to a shared experience replay memory, and compute initial priorities for the data. The (single) learner samples from this memory and updates the network and the priorities of the experience in the memory. The actors' networks are periodically updated with the latest network parameters from the learner.

Horgan et al. DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY. ICLR 2018

Ape-X Performance

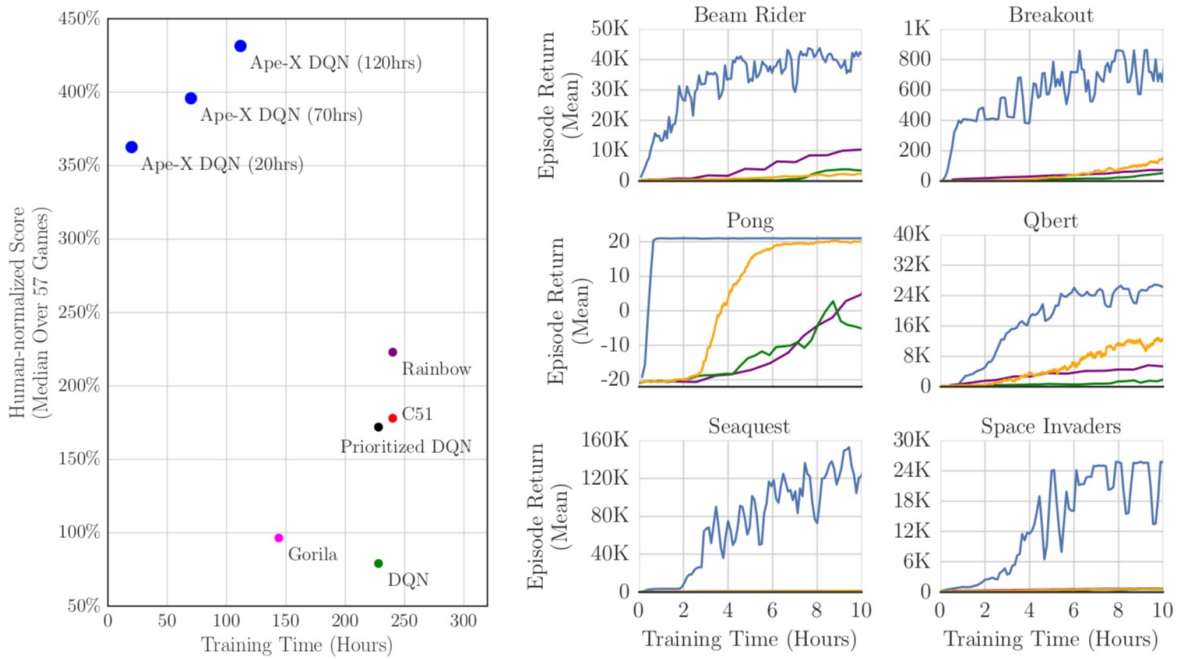


Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.

Horgan et al. DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY. ICLR 2018

Deep RL Simulators

- OpenAI Gym
  - https://gym.openai.com/docs/
  - A toolkit for developing and comparing reinforcement learning algorithms.
  - **Gym** provides an environment to implement any reinforcement learning algorithms.
  - Developers can write agents in TensorFlow
  - Limitation: no multi-agent support
- Full 3D Environments
  - Unity ML Agents: https://unity3d.com/machine-learning

RLLib

- https://ray.readthedocs.io/en/latest/rllib.html
- Open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib natively supports TensorFlow, TensorFlow Eager, and PyTorch.