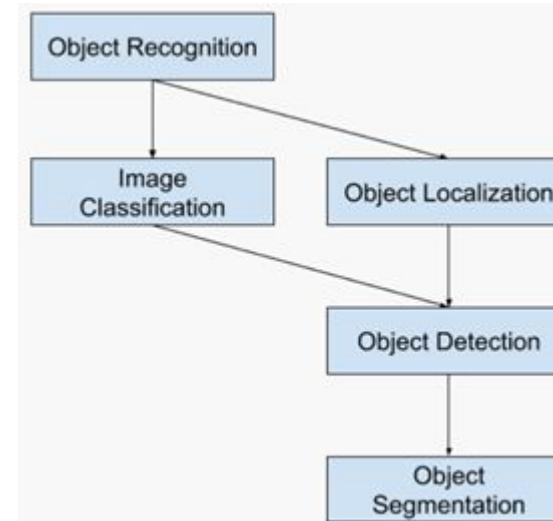
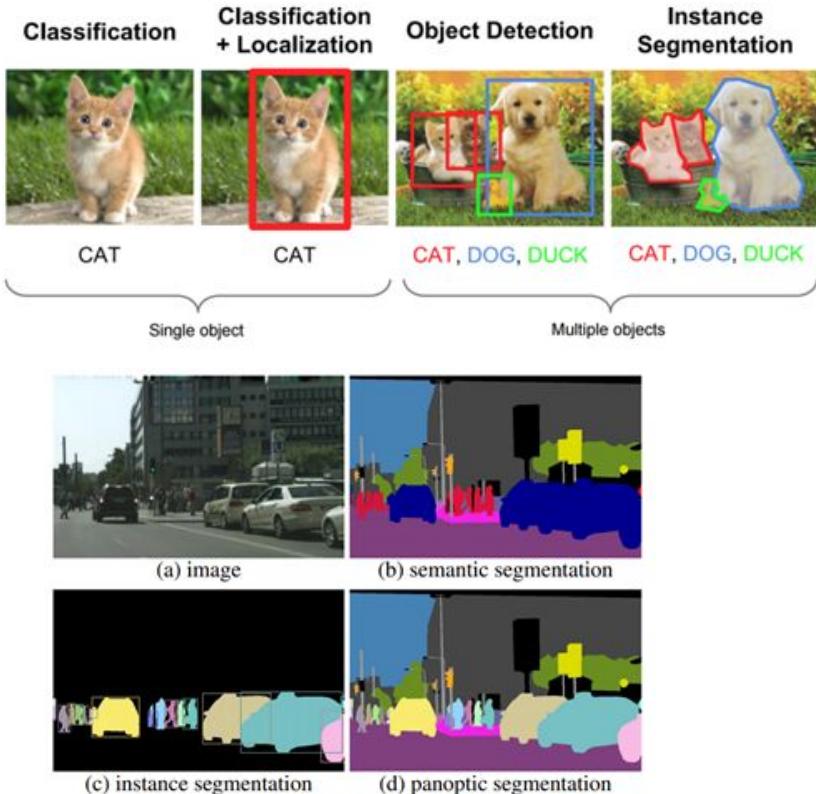


Object Detection/Segmentation

What's the difference between object recognition, classification, localization, object Detection, instance segmentation, semantic segmentation, and panoptic segmentation?



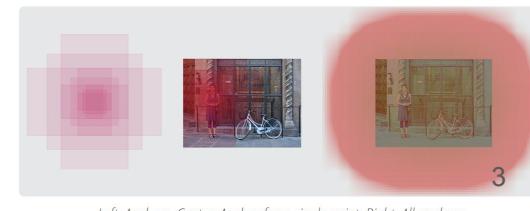
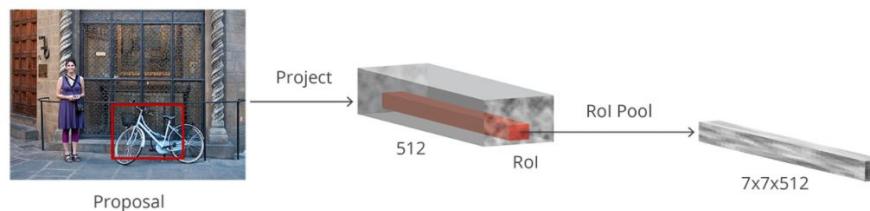
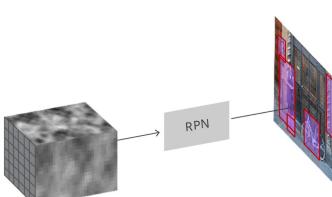
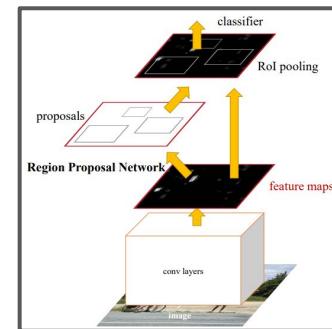
Describe Faster R-CNN. At a high level, how did it improve on Fast R-CNN and R-CNN?

These three methods address object detection. Faster R-CNN works with 4 stages:

- 1) **Region Proposal Network (RPN)** takes a CNN feature map as input. Then for a large, discrete set of predefined anchors and bounding boxes on the feature map, it outputs
 - a) A "objectness" score (representing foreground/background), formulated as binary classification
 - b) Slight transformations on that bounding box, formulated as regression.The regular grid of anchor points naturally arises due to the downscaling of the FEN, it doesn't have to be implemented explicitly.
This network is implemented in a fully convolutional way using 1x1 convolutions on the feature map. Only fg objects have a regression loss.
- 2) **Non-maximum suppression (NMS)** tries to remove duplicates, by iterating over the list of proposals sorted by score, and discarding those which have an IoU larger than some threshold with a proposal that has a higher score.
- 3) A **classification head**, which:
 - a) Applies **ROI pooling** to randomly selected fg and bg bboxed feature proposals from stage 1, by projecting from image to feature map, and standardizing the size using quantization and max pooling (better than avg pooling in practice).
 - b) Performs a final classification, to label it as one of the target classes, or background. This uses FC layers.
 - c) Performs a final bbox regression loss based on class. This uses FC layers.
- 4) Loss is computed as a classification loss (both fg/bg bboxes) and bbox loss based on L1 dist (on only fg)

Some notes:

- RPN needs to have high recall, since if we don't get proposals here there's no chance they will be classified
- For some applications you can just use RPN to save time (eg if you are only recognizing a single class)
- Comparing the 3 methods:
 - R-CNN does not use a CNN to predict region proposals; it uses a selective search algorithm. Then, it uses a SVM to classify.
 - Fast R-CNN, still uses selective search, but uses ROI pooling and classifies with a CNN.
 - Faster R-CNN adds the RPN, and is thus end-to-end.
 - The time it takes to process an image are approx 50 secs for R-CNN, 2 secs for Fast R-CNN, and 0.2 secs for Faster R-CNN.



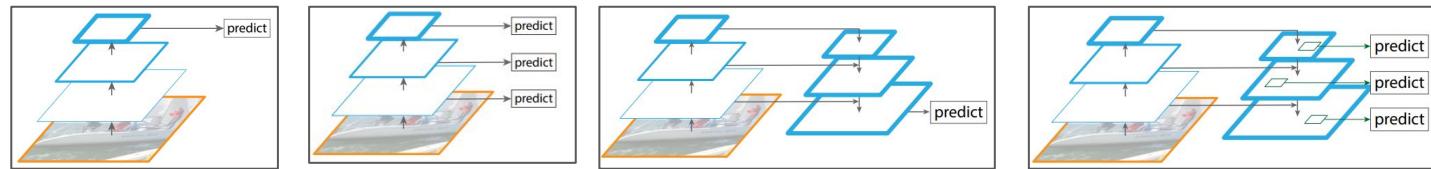
Explain the 1 stage detectors: YOLO, SSD, FPN, and RetinaNet. How do they compare to 2 stage detectors?

YOLO directly regresses bboxes with a CNN + FC layers:

- Input is split to a grid, and the network outputs a fixed number of bboxes per square. For each bbox, there are parameters for (x,y) relative to center of obj, width/height of bbox, confidence, and probabilities for each class.

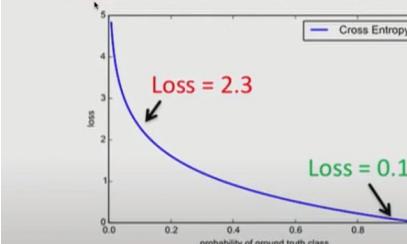
SSD is similar to YOLO, but predicts independent object detections from multiple feature maps of different resolutions. This helps with detecting small objects.

Feature Pyramid Networks (FPN) implements a U-shaped architecture which enables even better detection of small objects. As in U-Net, the intuition is to combine low-resolution, semantically strong features with high-resolution, semantically weak features via a top-down pathway and lateral connections to get rich semantics at all levels. It uses nearest neighbor upsampling (to “hallucinate” high level semantics, yet at high resolutions) and conv layers for upsampling, while U-Net uses transpose convolutions. From left to right, we show YOLO style, SSD style, UNet style, and FPN style.

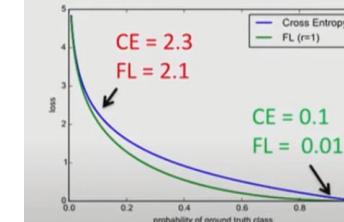


RetinaNet is essentially a combination of a ResNet backbone, FPN neck, and a Focal loss which helps with the fg/bg imbalance issue. Even with Cross Entropy loss's asymptotic behavior, the many easy bg examples can dominate the few hard examples. So, the focal loss adds a simple term to exacerbate the asymptotic behavior.

- 100000 easy : 100 hard examples
- 40x bigger loss from easy examples



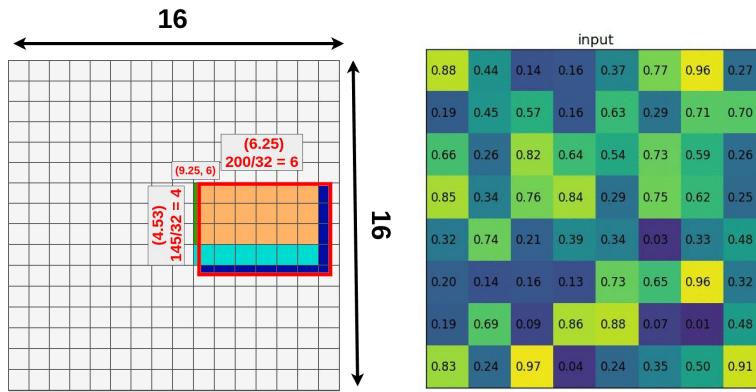
$$\text{CE}(p_t) = -\log(p_t)$$
$$\text{FL}(p_t) = -(1 - p_t)^\gamma \log(p_t)$$



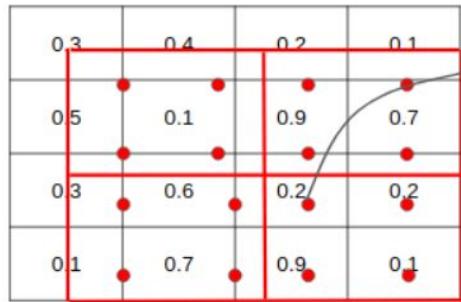
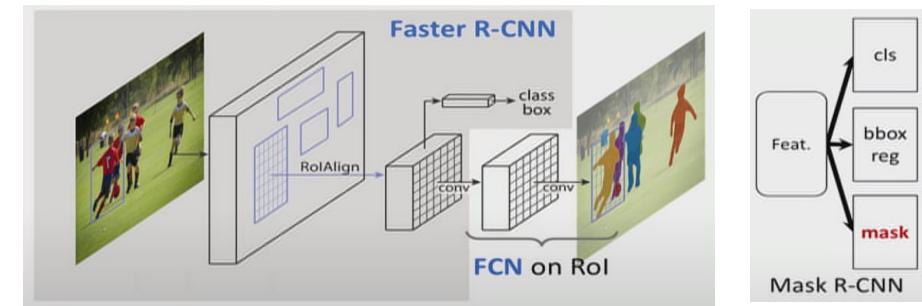
Explain Mask-RCNN.

Essentially a Faster R-CNN (ResNet+FPN backbone), with a Fully Convolutional Network (FCN) on each ROI for segmentation.

Uses **RoI Align** instead of RoI (max) Pooling, which does not use any quantization and does not break pixel-to-pixel alignment, by using bilinear interpolation before the max pooling operation.



RoI (Max) Pooling



0.1	0.5	0.48	0.45
0.6	0.68	0.62	0.15
0.32	0.2	0.284	0.15
0.84	0.34	0.34	0.1

2 x 2

0.68	0.62
0.84	0.34

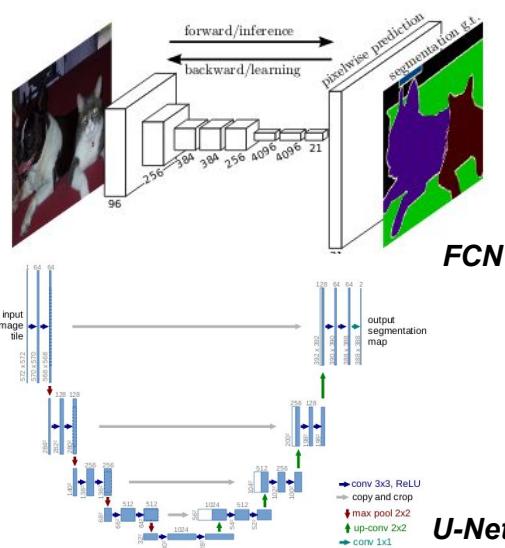
RoI Align

Explain FCN, U-Net, and DeepLab

All three are for semantic segmentation. Here, the key challenge is upsampling, while preserving detail.

Fully Convolutional Network (FCN) stacks a bunch of conv layers in an encoder-decoder fashion.

- No linear layers at the end for classification; only 1x1 convolutions. This allows different size img inputs.
- Upsamples only once in the decoder (using a deconvolution)
- Skip connections are added. The intuition is that this can recover fine-grained spatial information lost in the pooling/downsampling layers for making the final predictions

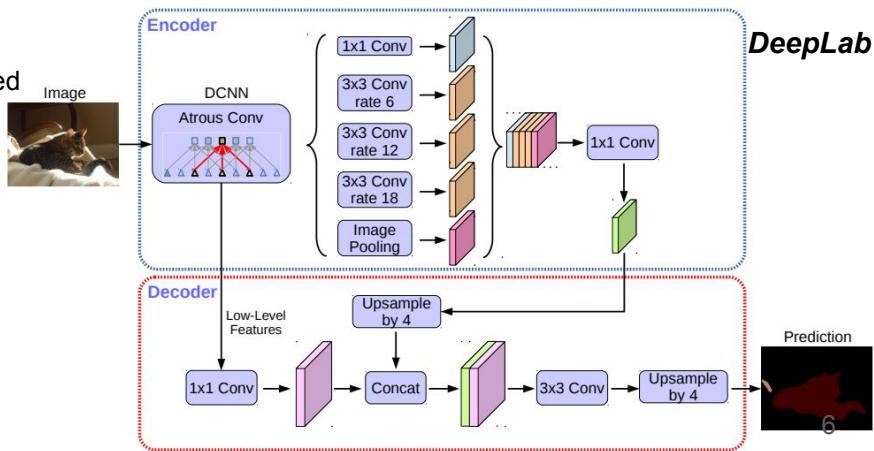
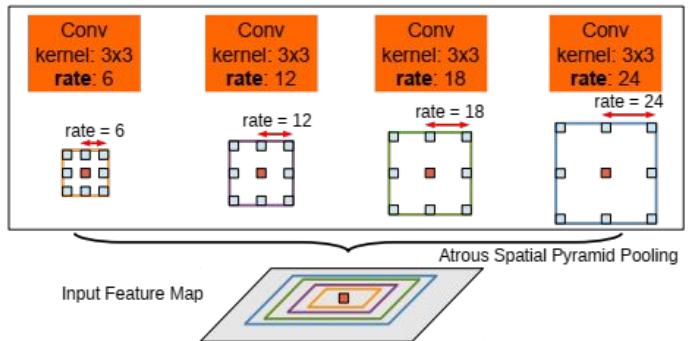


The **U-Net** is similar, but has some modifications:

- Multiple upsampling layers, so that the encoder-decoder is symmetric
- Skip connections are concatenated

The **DeepLabv3+** (from Google) is the best performing of the 3:

- Uses multiple Atrous convolutions, in a depthwise separable manner, at different dilation rates (creating a pyramid) in Encoder
- Upsamples 4x twice using bilinear interpolation, in Decoder
- Original DeepLab used conditional random fields (CRFs), but this was dropped



What losses are used for semantic segmentation?

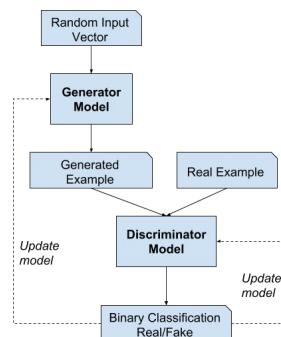
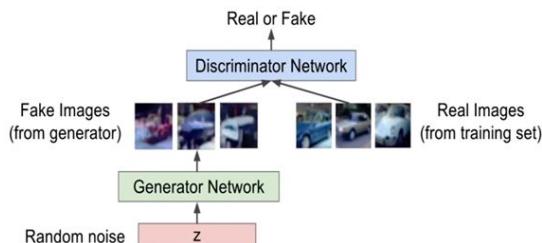
- Cross entropy
 - Dice loss
 - I.e., IoU loss
 - In practice a “soft dice loss” is used, where A is the probability map mask and the g.t. Mask
 - In general, this is conceptually more “direct”, but cross entropy can be better in implementation because it has nicer gradient properties which do not explode as easily

$$\text{Dice loss} = 1 - \frac{2|A \cap B|}{|A| + |B|}$$

Generative Modeling: GANS and VAEs

How do (W)GANs work, and how are they optimized? Can they be conditioned on?

GANs implicitly model probability density by focusing on the ability to sample from an implicit distribution, in an unsupervised way. This is done through a two-player game between a generator and discriminator network. Convergence is reached at “nash equilibrium”, where which each player cannot reduce their cost without changing the other players’ parameters.



1. Gradient ascent on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

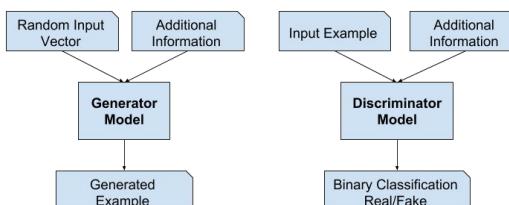
2. Instead: Gradient ascent on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

The **Wasserstein GAN (WGAN)** is an alternative formulation which may have better theoretical properties.

- Discriminator becomes a “critic”, which optimizes towards an approximation of the Wasserstein distance between the real and generated distribution.
- Unlike standard GANs, we want to train critic to convergence. In principle, WGAN loss should be informative (unlike standard gans losses, which have no immediate meaning for how training is progressing).
- The critic/generator are supposed to be K-Lipschitz, so gradient clipping or penalty is needed.
- The critic is optimized to maximize the distance between average real and generated outputs; this approximates the wasserstein loss (lines 2-8)
 - Critic outputs “realness score”; lower critic score= fake, higher critic score= real
- Then, the generator is trained to maximize the critic’s score

(W)GANs can be conditioned (cGAN) on by providing both the generator and discriminator with some additional input.



Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size. n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator’s parameters.

```

1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{z^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w \left[ \frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)})) \right]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$ 
12: end while

```

Explain how Pix2Pix and CycleGAN works, and what they're useful for.

Both methods perform the task of image-to-image translation. Pix2Pix requires paired data, while CycleGAN does not.

Pix2Pix is a conditional GAN (cGAN); the discriminator is fed both domains to discriminate.

CycleGAN for domains X and Y, learns a mapping $G:X \rightarrow Y$ and inverse mapping $F:Y \rightarrow X$. The outputs of both have their own discriminator. It's enforced that cycles $F(G(x))=x$ and $g(F(y))=y$ hold.

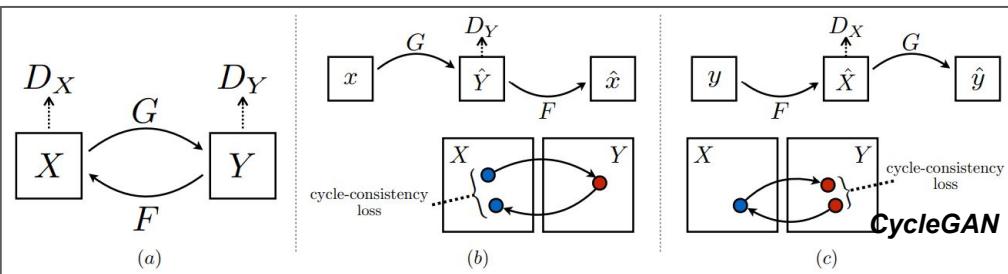
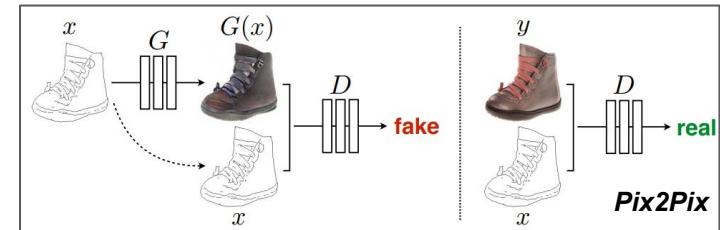
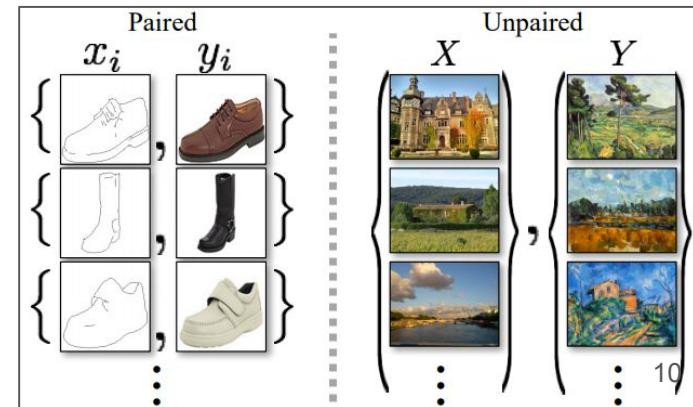


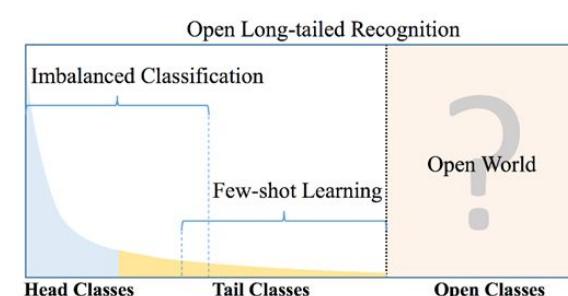
Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X . D_Y encourages G to translate X into outputs indistinguishable from domain Y , and vice versa for D_X and F . To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$



Data Imbalance

At a high level, what is the data imbalance problem? Can you measure performance using standard metrics?

- Naively training on an unbalanced dataset can lead to **frequency bias**, where they place more emphasis on learning from data observations which occur more commonly.
 - In some cases, this might make sense.
 - However, this may be the opposite of what we want, e.g. a classifier for fraud wants to actually focus on the minority class.
- There are different **degrees of imbalance**
 - You can have some classes with significantly fewer datapoints (**minority classes**) and some with significantly more datapoints (**majority classes**)
 - In some cases the data may be **long-tailed**, overall
 - A more general problem than **few-shot learning**
 - We need to not only do well for classes with few data, but also equally well for those with medium to large amounts of data, in a balanced way
- In general, there is a **trade-off between doing well in the head classes vs doing well on the tail classes**.
 - For example, tail classes need large margins, which can hurt accuracy of head classes.
- Examples of data imbalance:
 - Credit card fraud detection or spam datasets usually have many, many more authentic transaction datapoints than fraudulent datapoints
 - Autonomous driving datasets can have many unusual outlier events which do not appear often in the dataset
 - Rare disease classification
 - ImageNet itself is longtailed
- Standard classification metrics can be very misleading when dealing with imbalanced data.
 - For example, if your dataset has 97 data samples with no credit card fraud and 3 data samples with fraud, when a naïve classifier always predicting "no fraud" already achieves 97% accuracy.
 - Use **balanced accuracy**, **Matthews correlation coefficient**, **macro-averaged precision/recall/F1**, or **balanced mAP**
 - Macro-averaged precision/recall/F1 allows you to look at precision/recall/F1 at ALL your classes, and aggregates it
 - Note that ROC curves can be misleading for imbalanced scenarios due



What are the methods to deal with data imbalance?

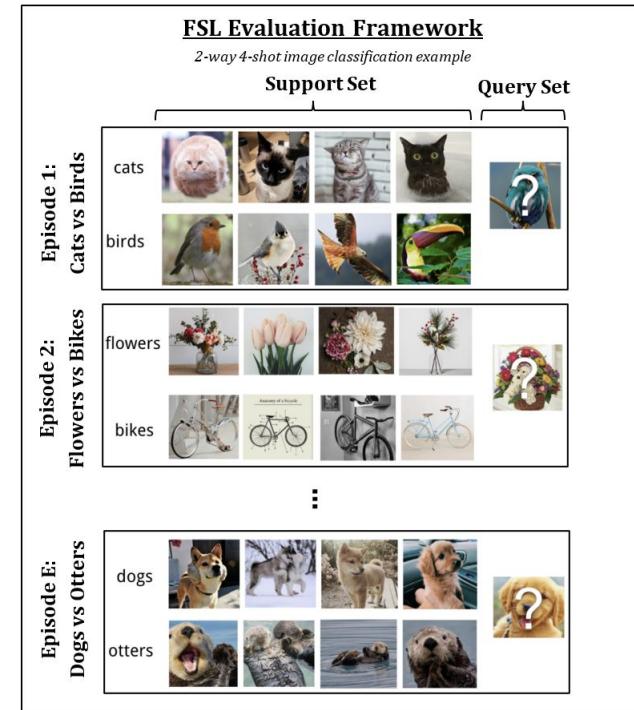
Summary of Methods to Improve Learning with Data Imbalance

Name	Description	Remarks	Figure
Data Collection	<ul style="list-style-type: none"> Collect more data to improve class balance 	Best performance, but costs the most.	
Oversampling	<ul style="list-style-type: none"> Duplicate data copies of the minority class(es), to improve balance. 	Can cause overfitting and poor generalization.	
Undersampling	<ul style="list-style-type: none"> Remove some data from the majority class(es), to improve balance. For example, this can be random, or you can choose the "furthest neighbor points". One technique is NearMiss, which tries to only keep points from majority class which are close to the decision boundary (i.e. avg distance to the minority class are smallest) 	A good choice when you have a ton of data from majority class. Obviously, also improves training time. May discard valuable data. One option is to use an ensemble (similar to bagging).	<p>Left: Undersampling. Right: NearMiss.</p>
Data Augmentation	<ul style="list-style-type: none"> Create synthetic or augmented data for minority class(es). This can be through data augmentation or even GANs. In SMOTE, you randomly select a close neighbor of a minority class point, and create a random point within their line segment. This requires a well-defined semantic space. In Mixup, new images are created by interpolating pairs images & their labels. 		<p>Left: SMOTE. Right: Mixup.</p>
Reweighting	<ul style="list-style-type: none"> Provide a weight for each class in the cost function (e.g. cross entropy), to provide more emphasis on minority classes. Focal Loss modifies the cross-entropy function curvature so easy, confident predictions have less loss weight. Widely used for detection, where there are many "easy" BG negative bboxes and only a few "hard" FG positive, to pay attention to. 	A ICLR 21 paper states that although mathematically equivalent, resampling is generally better than reweighting because of factors from SGD optimization.	<p>Left: Class-weighted CE loss. Right: CE (Blue) vs Focal Loss (green)</p>
Two-Stage Training	<ul style="list-style-type: none"> Consists of an imbalanced training stage first (to learn good representations) and a balanced fine-tuning stage (using another method, to re-balance). 		
Metric Learning	<ul style="list-style-type: none"> Want to enforce good margins in an embedding space (especially for few-shot examples), in a class-balanced sensitive way. GistNet tries to transfer the decision boundary shape of many-shot classes to the medium/few-shot classes. Large-margin softmax tries to make margins larger, useful for generalizability of few/medium shot. Label Distribution Aware Margin (LDAM) is an improvement on this to consider class data size. 		<p>Left: GistNet. Right: LDAM.</p>

Few-Shot Learning

At a high level, what is the few-shot learning problem?

- In **few-shot learning (FSL)**, the goal is to learn tasks with only a few training examples.
- We usually evaluate a FSL algorithm's capability for data-efficient learning with multiple **N-way, K-shot episodes**.
- Here, “**way**” = “**classes**” and “**shot**” = “**training examples per class**”. An **episode** can be thought of as an individually defined “classification task”. In detail, each episode will have:
 - A **support set** containing K training samples & labels per class (so, samples total), which the FSL algorithm trains on
 - Commonly, K is 1 or 5; N is usually 5 but can be over 100
 - A **query set**, which we test the FSL algorithm on after it trains on the support set
 - Generally at least NK in size
 - For testing purposes we have labels, but the FSL algorithm doesn't train on them; it's purely a forward pass
- In general, FSL episodes are easier with fewer classes (lower way) and more examples per class (higher shot).
- FSL episodes are harder with more classes and less examples per class.
- A variant of few-shot learning is **zero-shot learning**, which learns solely on the category name, textual descriptions, or attributes.
- Note that many few-shot methods are more tailored to that problem, and may be insufficient to tackle the more general **long-tailed problem**.

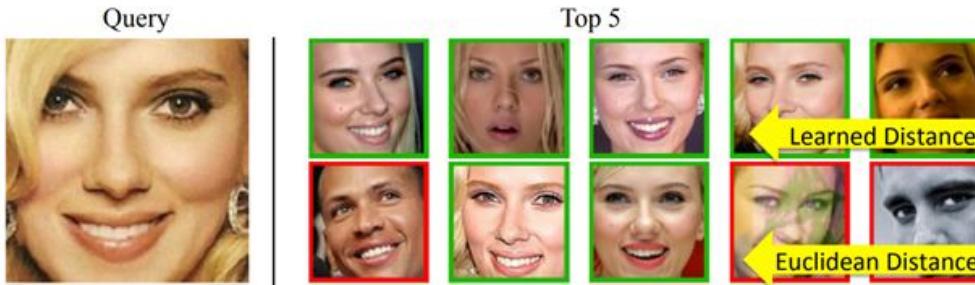


At a high level, what is meta learning? How can it be applied to few-shot?

- **Meta Learning** is a field by itself, and the goal is to design models which can learn new skills or adapt to new environments rapidly
- This is done by aiming to **improving the learning process itself**, through multiple learning **episodes** and gives an opportunity to tackle issues of deep learning such as data/computation requirements and generalization
- You want to “**learn how to learn**” efficiently, rather than learning for a specific downstream task; the goal is to learn something of a higher (meta) order, not necessarily the task directly
- For example, humans learn certain universally applicable mental tricks/techniques that allow us to learn novel tasks/concepts rapidly
- You can view meta learning as a **subfield of transfer learning**, but meta learning further **emphasizes data and/or computational efficiency**
- This makes it a natural paradigm/approach for the task of few-shot, but meta learning also has applications in things besides few-shot, like **fast learning**, **continual learning**, **domain generalization**, and **reinforcement learning**

At a high level, what is metric learning? How can it be applied to few-shot?

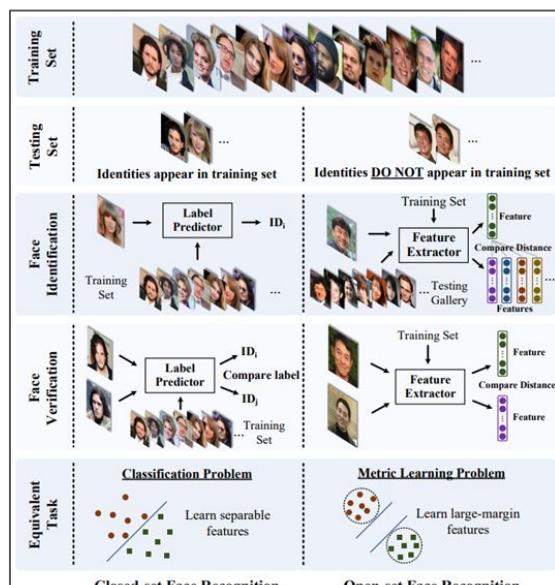
- Metric Learning (i.e. similarity learning) aims to measure how related two objects are, in a lower-dimensional, semantic manifold
- It can be learned with a regression or classification based framework, using labeled pairs of similar or dissimilar images
- Alternatively, **triplets** of can be used where is known to be more similar to than .
 - This is **weaker supervision** than regression or classification, since instead of providing an exact measure of similarity (usually one-hot), one only has to provide the relative order of similarity. This also allows the use of data augmentation, for self supervised learning (e.g. simCLR)
 - However, mining for the “hard” examples for to train on is important, and nontrivial
- The embeddings can **readily be used for one-shot learning tasks** (e.g. face verification databases), and **support easy additions** at test-time without retraining. You just need to embed the support elements, and use **K nearest neighbors to classify**.
- Besides FSL, other applications for metric learning include **ranking** (for **retrieval** or **recommender systems**), large-scale **classification**, or **clustering**. It is also common to use the learned representation from **self-supervised metric learning** for a wide range of **downstream tasks via transfer learning**.



Explain how few-shot can tackled with transfer learning, optimization meta learning, and metric meta learning.

FSL via Transfer Learning

- Pre-train on a large, separate dataset with many classes and instances per class (where classes distinct from those seen during evaluation), e.g. ImageNet
- Training can be fully supervised or self-supervised
- During evaluation, **transfer the learned representations**, for example by
 - Fine-tuning on the NK episode training examples, or
 - Using it as a **fixed feature extractor** and learning a linear classifier (e.g. softmax) on top
- Intuitively, at the transfer learning step, you're not “teaching it to reason”, just “**recontextualizing**” with the few examples that you have
- Most basic approach, but surprisingly, **very competitive** with more complicated meta learning based methods

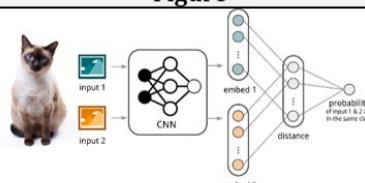
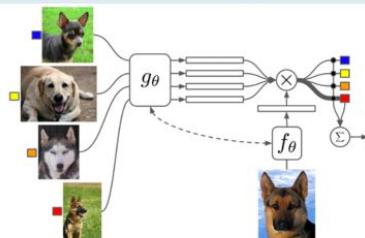
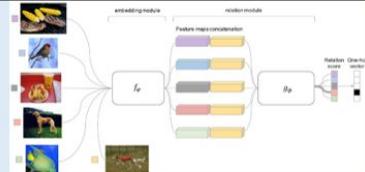
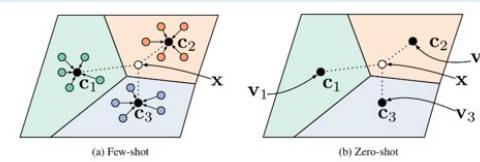


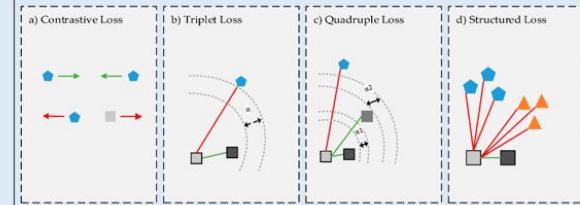
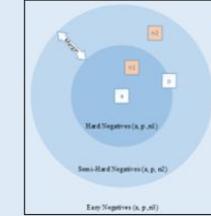
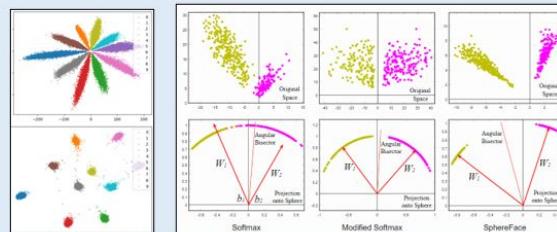
FSL via Optimization-Based Meta Learning

- Insight: **Replicate the FSL episode-based evaluation procedure during training**
- Intuitively, optimize for “fast-weights/parameters” in network which can be adapted in a data-efficient manner to episodes without overfitting, using only a few gradient steps
- However, some believe that this is more less due to **reusing/learning high-quality features**, rather than “rapid learning of each task”.

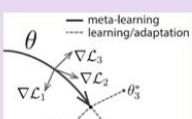
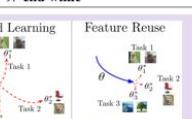
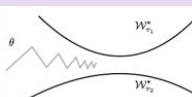
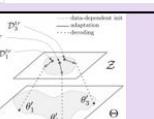
FSL via Metric-Based Meta Learning

- Based on metric learning; want to learn a mapping from images to an **embedding space**, and use a (possibly learned) **similarity function** between points
 - Goal: images from the same category are closed together and from different categories are far apart.
 - Ideally, the mapping will hold true for unseen categories; during evaluation, we embed the support set and use nearest-neighbors to classify
 - Essentially assumes that k can be modeled by the form $k(\mathbf{x}, \mathbf{y}) = \text{exp}(-\|\mathbf{x} - \mathbf{y}\|_2^2 / 2\sigma^2)$, where k is a similarity function (e.g. embedding Euclidian distance)
- “Older”, more classical approach to FSL
- Especially useful for tasks like **face verification**, where you have a **frequently changing database** of objects (pictures of people) that you want to recognize, but only a few data samples for each object. Unlike Meta-learning approaches, you would **not need to retrain** when you add or remove people from the database.
- Can involve contrastive methods (where you have positive and negative sampling), or methods based on softmax (which is a generally easier to apply)

Summary of Methods for Few-Shot Learning			
Metric Meta Learning Optimization Meta Learning Transfer Learning			
Name	Category	Description	Figure
Siamese NNs (ICML 15, >2500 cits)	Metric Meta Learning	<ul style="list-style-type: none"> Feeds pairs of images into a CNN to get two embeddings The L1 distance between embeddings fed into a linear layer to predict similarity probability score of if they are same class Trained with binary cross entropy loss At test time, a query image is classified as the class of the support image with highest class-similarity probability 	
Matching Networks (Google, NIPS 16, >3700 cits)	Metric Meta Learning	<ul style="list-style-type: none"> Given support $S = \{x_i, y_i\}_{i=1}^k$ where y_i are one-hot vectors, learns embedding functions f, g such that for query x we have $P(y x, S) = \sum_{i=1}^k a(x, x_i)y_i$ a is an "attention kernel" weighing one-hot vectors, based on cosine similarity softmax: $a(x, x_i) = \frac{\exp(\text{cosim}(f(x), g(x_i)))}{\sum_{j=1}^k \exp(\text{cosim}(f(x), g(x_j)))}$ Training mirrors evaluation: during training, loss explicitly minimized over small episodic samples of support & queries Potentially, f, g can be the same feature extractor, but they instead propose a more complicated bidirectional LSTM 	
Relation Networks (CVPR 18, >2000 cits)	Metric Meta Learning	<ul style="list-style-type: none"> Similar to Siamese NNs gets pairs of embeddings, but concatenates them and feeds into a NN to predict relation score instead of using L1 distance Also, instead of cross entropy, takes a regression approach and uses MSE with a one-hot vector 	
Prototypical Networks (NIPS 17, >3500 cits)	Metric Meta Learning	<ul style="list-style-type: none"> Embeds all support images, and defines a prototype embedding for each class mean Given query x, distribution over classes given as softmax over negative squared Euclidean distance It's argued that this inductive bias reduces overfitting Training mirrors evaluation: episodes sampled during training Trained with cross entropy 	

Summary of Methods for Few-Shot Learning			
Metric Meta Learning Optimization Meta Learning Transfer Learning			
Name	Category	Description	Figure
Contrastive Metric Learning: <u>Triplet Loss</u> (FaceNet, CVPR 15, >10K cit), Quadruplet Loss (CVPR 17, 1K), Structured Loss (CVPR 16, 1.2K), N-Pair Loss, Magnet Loss, Clustering loss, MoCO (CVPR20, 21 K)	Metric Meta Learning	<ul style="list-style-type: none"> Goal: pull together embeddings with same label, and push away those with dissimilar labels, with explicit pos/neg samples Classical contrastive loss uses pairs of samples and margin α: $\mathcal{L} = \sum_{y_1=y_2} D^2(f(x_1), f(x_2)) + \sum_{y_1 \neq y_2} \max(0, D^2(\alpha - f(x_1), f(x_2)))$ Triplet loss uses A and P with same class, and N of different class: $\mathcal{L}(A, P, N) = \max(\ f(A) - f(P)\ ^2 - \ f(A) - f(N)\ ^2 + \alpha, 0)$ Structured and N-Pair Losses tries to improve sampling with all samples in a batch, instead of ignoring them and only using one Magnet and Clustering loss are based on the dataset as a whole, but is complex and hard to scale. Often paired with heavy data aug, large batch size, and/or <u>memory bank</u> to efficiently reuse embeddings from the prev. iteration Main difficulty of contrastive learning is sampling comprehensively, mining the “hard negatives”. Also, batch-sample based learning enforces local margins, but a global margin for all labels is hard to achieve in practice. Thus, non-contrastive losses are often easier to use. Useful for unsupervised representation learning (e.g. simCLR) 	 
Margin Non-Contrastive Metric Learning: <u>Center Loss</u> (ECCV16, 27K cit), <u>LargeMargin Softmax</u> (ICML16, 1K), <u>SphereFace</u> (CVPR 17, 2K cit), <u>CosFace</u> (CVPR 18, 1300 cit), <u>ArcFace</u> (CVPR 19, 2300 cit)	Metric Meta Learning	<ul style="list-style-type: none"> Goal: pull together embeddings with same label, and push away those with dissimilar labels, without explicit pos/neg samples Center loss adds an additional term to softmax loss, to pull each batch of N features towards its (jointly trained) class center Large-Margin Softmax notes that for sample (x_i, y_i), the softmax loss for a FC layer exponentiates $W_{y_i}^T x_i = \ W_{y_i}\ \ x_i\ \cos(\theta_{y_i})$ in the numerator (selects the g.t. class vector in W and takes dot prod). Then, a larger margin is explicitly enforced by enforcing larger θ. SphereFace improves margin by enforcing the centers to be on a hypersphere w/ normalization, improving the decision boundary Generally only applicable when supervised. For unsupervised settings or when you have a lot of out-of-distribution samples, contrastive learning may be better. General tradeoff with within-domain accuracy (smaller margins, standard classification) vs out-of-distribution generalization (larger margins, longtail/fewshot/openset verif.) 	 <p>Left: Softmax vs Center Loss. Right: Softmax vs SphereFace</p>

Summarize the following optimization meta learning approaches for few-shot: MAML, Reptile, LEO.

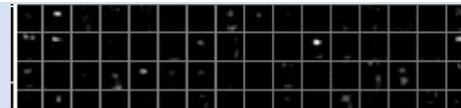
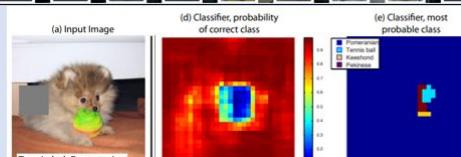
Summary of Methods for Few-Shot Learning			
Metric Meta Learning Optimization Meta Learning Transfer Learning			
Name	Category	Description	Figure
MAML: Model Agnostic Meta Learning (ICML 17, > 5000 cits)	Optimization Meta Learning	<ul style="list-style-type: none"> Goal: Learn good NN initialization that's easy to fine-tune Parameters of NN explicitly trained so a few gradient steps with a few training data from a new task will produce good generalization performance Very literal approach. Repeatedly 1) samples tasks 2) for each task, takes a gradient step on individual copy of weights, and 3) meta-updates weights, backpropagating through step 2 so that weights would minimize loss after step 2's gradient step: $\min_{\theta} \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}(f_{\theta} - \alpha \nabla_{\theta} \mathcal{L}_{T_i}(f_{\theta}))$ The 3rd step involves taking the gradient of a gradient (2nd order derivative), unrolling through the computational graph. However, it's shown that First-Order MAML is almost as good, by treating the weights from step 2 as constants. There's evidence MAML mostly reuses/learns high-quality features, rather than "learning how to learn rapidly". 	 <p>Algorithm 1 Model-Agnostic Meta-Learning</p> <pre> Require: $p(\mathcal{T})$: distribution over tasks Require: α, β: step size hyperparameters 1: randomly initialize θ 2: while not done do 3: Sample batch of tasks $T_i \sim p(\mathcal{T})$ 4: for all T_i do 5: Evaluate $\nabla_{\theta} \mathcal{L}_{T_i}(f_{\theta})$ with respect to K examples 6: Compute adapted parameters with gradient descent: $\theta'_i = \theta - \alpha \nabla_{\theta} \mathcal{L}_{T_i}(f_{\theta})$ 7: end for Note: the meta-update is using different set of data. 8: Update $\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{T_i \sim p(\mathcal{T})} \mathcal{L}_{T_i}(f_{\theta'_i})$ 9: end while </pre> 
Reptile (OpenAI, 18, >1000 citations)	Optimization Meta Learning	<ul style="list-style-type: none"> Like MAML, goal is to find a parameter configuration which is close to the optimal parameter manifolds of all tasks Repeatedly 1) samples tasks, 2) separately trains parameters on each task for multiple gradient steps, and 3) meta-updates the original parameters towards the averaged new parameters Multiple SGD steps makes it different than joint, mixture training They theoretically show that this generalizes MAML and also implicitly involves 2nd derivatives, but is more efficient 	 <p>Algorithm 2 Reptile, batched version</p> <pre> Initialize θ for iteration = 1, 2, ... do Sample tasks $\tau_1, \tau_2, \dots, \tau_n$ for $i = 1, 2, \dots, n$ do Compute $W_i = \text{SGD}(L_{\tau_i}, \theta, k)$ end for Update $\theta \leftarrow \theta + \beta \frac{1}{n} \sum_{i=1}^n (W_i - \theta)$ end for </pre>
LEO: Meta-Learning w/ Latent Embedding Optimization (DeepMind, ICLR 19, >700 cits)	Optimization Meta Learning	<ul style="list-style-type: none"> While MAML operates directly in a high dimensional parameter space, LEO performs meta-learning within a low-dimensional latent space Thus, parameters are generated/predicted 	

Explainable Artificial AI (XAI)

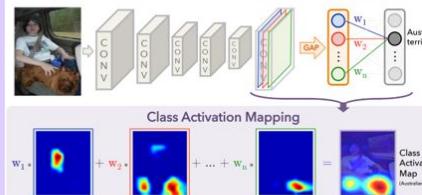
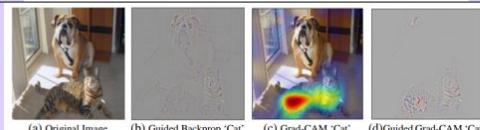
What's the difference between reliability, explainability, and transparency?

- In general, there is no consensus, but below is one way to look at it
- **Reliability**
 - Degree to which a human can consistently predict the model's result
 - Can be improved with systems that are robust and fails “gracefully” by detecting data outside the model's domain of competence
 - Existence of adv. attacks hurt reliability
 - Sometimes, just reliability and performance is enough, especially for non-safety-critical systems
- **Explainability**
 - Degree to which we understand at a high-level the model's learned knowledge and rationale behind its decisions
 - Can be built-in, or post-hoc
 - Can be textual, visual, or mathematical in nature
 - Still an open problem in DL, and many current methods have shortcomings, suffering from fragility and cherry-picking:
 - Attention/grad-cam helps to some degree, but only says where the model is looking – not why. Often hard to decipher when wrong.
 - Visualizing filter weights or t-SNE can show the learned knowledge, sometimes
 - Textual explanations are in principle a powerful, natural way to do this, but still have a long way to go
 - Useful for debugging; crucial for improving public trust in ML, especially for safety-critical systems
- **Transparency**
 - Degree to which we have a detailed understanding of the internal mechanics the model. Black-box vs white-box.
 - Not only involves understanding the algorithm or architecture, but also mathematical interactions between learned weights, different layers, as well as hyperparameters.
 - Can be relative to an expert, or a lay-person
 - Usually through a theoretical, mathematical lens. For example, understanding of the bounds of a system.
 - NNs are highly opaque; for example, we still don't rigorously know why batchnorm or residual connections really work
- In general, reliability is most important for immediate applications, followed by explainability. Transparency is mostly important for research and future innovation.
- Things like linear models or decision trees are both highly explainable and transparent, but lack in performance.

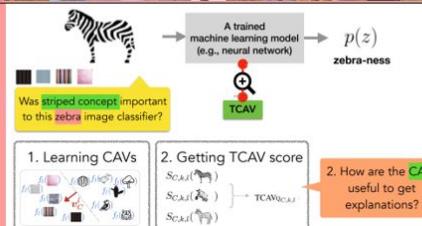
At a high level, describe the following basic XAI methods: Layer weight/activation visualization, maximally activating retrievals, and occlusion maps.

Show Layer Activations	Show forward pass activations (as an image) for an input	Check for localized regions in later layers. If an activation map is zero for many different inputs, this can indicate dead filters	
Show Layer Weights	Show convolutional filter weights (as an image) at layer of trained network	Usually, well-trained networks should display nice and smooth filters without noisy patterns.	
Maximally Activating Retrievals	Take large dataset of images, feed them through the network. Keep images maximally activate some neuron	Gives an understanding of what specific neurons are looking for in its receptive field.	
Occlusion Maps	Plot the probability of the correct class as a function of the position of an occluder object (e.g. a gray square), as a 2D heatmap.	Helps understand where a CNN is looking for its prediction, e.g., checks if it's looking a context clues/backgrounds rather than the main object	

At a high level, describe the following backprop-based XAI methods: Guided backprop, DeepDream, CAM, Grad-CAM.

Guided Backpropagation	Backpropagate prediction all way back to the input while zeroing-out negative gradients (only accounts for positive contributions, reducing noise)	Fine-grained visualization, in input space to help understand the important parts of the input in making prediction.	
DeepDream	Apply gradient ascent on an image or noise, freezing network weights, to optimize the probability classification of a certain class or (classes).	Shows the salient features used for the task, through generation. Can reveal biases/correlations in network.	 Towers & Pagodas Buildings Birds & Insects Dumbbell
Class Activation Mappings (CAM)	In architecture, number of feature maps in the last conv layer equals the number of classes. Maps are global averaged pooled and fed directly into softmax. This forces a heatmap-like representation. CAM is the weighted average of those feature maps.	Shows the most important areas in input image used for the prediction.	
Grad-CAM	Weighted sum of feature maps activations; weights are from the gradients with respect to a target class, averaged per feature map.	Shows the most important areas in input image used for the prediction.	 (a) Original Image (b) Guided Backprop 'Cat' (c) Grad-CAM 'Cat' (d) Guided Grad-CAM 'Cat'

At a high level, describe the following numerical-based XAI methods: T-SNE, TCAV.

T-SNE	A method for non-linear dimensionality reduction, based on neighbor distances. Can be applied to feature embeddings for network inputs.	Can check for clusters within data.		
Testing with Concept Activation Vectors (TCAV)	Given images of concepts and random images, trains linear classifier for vector orthogonal to hyperplane decision boundary.	Flexible way to quantitatively check how much abstract concepts (e.g. gender, race) are related to a trained-CNN prediction. Concept need not be seen during training.		

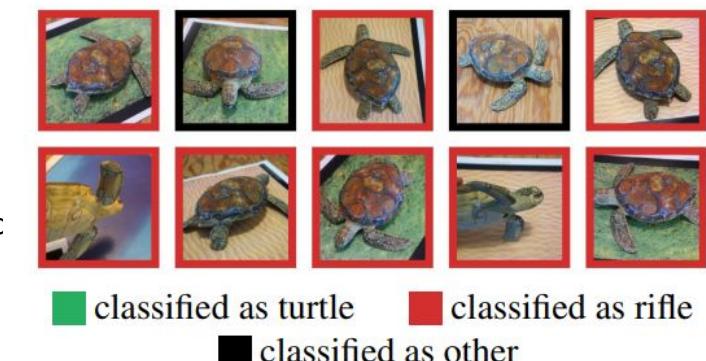
Security / Adversarial Attacks

Explain what white box and black box attacks are, and how they're typically implemented.

- **White box** attacks have access to neural network weights
 - You can use **gradient ascent** to backpropagate back to the original input image
- **Black box** attacks do not have access to the weights of the neural network.
 - A **substitute model** can be used to emulate the original model, using a transferable attack strategy
 - A **query feedback** mechanism can be used, along with random/grid/local search. You just keep modifying the input until you break the CNN and produce incorrect results. Modifications can be things like affine transform, changing hue, etc.
- Most attacks that are designed to fool a particular CNN also fool many other CNNs
- There are two theories on why adversarial attacks exist:
 - Consequence of SGD-based approximate optimization
 - Reliance on human-invisible non-robust feature patterns (i.e. overfitting to the training dataset)

Explain what real-world adversarial attacks are, and give some examples.

- Real world attacks face some additional challenges:
 - Needs to have viewpoint invariance and robustness to lighting, occlusion, etc
 - If it's on something like a street sign, cannot be on the whole image, only on the foreground obj
 - Cannot just be imperceptible; needs to be perceptible by camera
 - Needs to be robust to some fabrication error (eg by printer)
- The paper “Synthesizing Robust Adversarial Examples” shows that attacks can be 3c printed and work in the real world.
 - A 3D turtle texture is optimized to be an adversarial attack from many viewpoints and lighting conditions with differentiable rendering
- In the paper “Robust Physical-World Attacks on Deep Learning Visual Classification”, real-world attacks are performed on stop signs
 - The adversarial optimization procedure is performed over a distribution (dataset) of “realistic” stop signs, by using a hybrid real & synthetic augmentation approach.
 - A mask is used to ensure the attack pattern is on the foreground stop sign
 - There is also a loss that encourages colors/patterns to be easily printable



■ classified as turtle ■ classified as rifle
■ classified as other

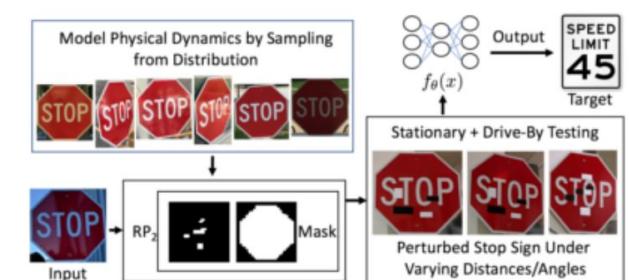


Figure 2: RP2 pipeline overview. The input is the target Stop sign. RP₂ samples from a distribution that models physical dynamics (in this case, varying distances and angles), and uses a mask to project computed perturbations to a shape that resembles graffiti. The adversary prints out the resulting perturbations and sticks them to the target Stop sign.

What are some typical defenses for adversarial attacks?

- Adversarial training (eg training from FGSM or iterative FGSM attacks)
 - In general, any attack can be transformed into a defense in this way
- Ensemble voting
- Adversarial classifier

$$\mathbf{X}^{adv} = \mathbf{X} + \epsilon \operatorname{sign}(\nabla_{\mathbf{X}} J(\mathbf{X}, y_{true}))$$

FGSM procedure, where J is the loss function

Efficient Deep Learning

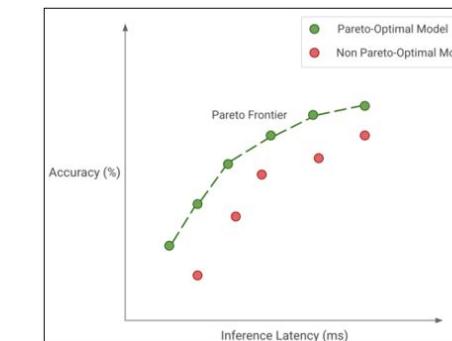
At a high level, what is the goal of efficient DL?

- The goal of efficient deep learning is to improve the **quality/footprint ratio** of models:

$$\text{Model Efficiency} = \frac{\text{Quality}}{\text{Footprint}}$$

Measured by metric, e.g. Accuracy, precision, recall, IoU, MSE, etc.
In terms of runtime, computational hardware needed, or data.

- While usually not a priority for general ML research, it's important for:
 - Very large models (which cost substantial amounts of money)
 - Deployment at the edge (e.g. inference of NNs on smartphones)
- In general, there is a **Pareto optimal frontier** that describes optimal efficiency for intrinsic trade-offs between quality and footprint. The goal of efficient DL is to approach the frontier.
- Methods can be broken down into 5 categories:
 - Compression:** Can we compress parts of the given model graph?
 - Learning Techniques:** Can we train the model better? Techniques may involve the data and/or loss functions.
 - Architectural:** Can we use/design layers and architectures which are fundamentally more efficient?
 - Automation:** Can we search for more efficient models?
 - CS-Based:**
 - Caching:** if some inputs are more common than others. Before calling model, check cache
 - Batching:** Gather prediction requests until you have a batch, then perform them in parallel as a batch. Batch size tradeoffs between throughput and latency



Explain some compression and architectural based methods for efficient DL.

Overview of Methods for Efficient Deep Learning

Footprint Reduction | Quality Improvement | Both

Method Name	Category	Description	Figure
Numerical Quantization	Compression	<ul style="list-style-type: none"> Replaces high-bit signed floats with lower-bit signed ints, weights and/or activations Conversions to/from ints are performed using scale mapping (range-based normalization) <ul style="list-style-type: none"> E.g. suppose float range is [0,1] and we want to convert 0.5 to int8 which can represent ints in [-128,128]; the result is 64 At the extreme end, can even be binarized (BNNs) Generally, multiplications reduced precision; additions lighter and can be full precision Simplest method is to apply quantization post-training, but has more accuracy loss Can also make training quantization-aware (e.g. by "simulating" it, allowing the network to adapt and achieve better performance) Can lead to significant (~4x) reductions in model size and inference latency 	
Model Pruning	Compression	<ul style="list-style-type: none"> Removes weights (i.e. set to 0, making applicable for sparsity optimization) or remove nodes/filters from a trained model (more easily supported, but can hurt effectiveness more) Goal: reduce memory and make calculations faster Several heuristics to remove weights or nodes, e.g. based on weight magnitude (replace those close to 0 with 0), activation outputs (replace "dead" neurons which always output small values), or derivatives wrt loss Network can be retrained after pruning to restore accuracy (iteratively) 	<p>Structured / Block Sparsity (4x4) in a Weight Matrix (Gray = Weight is Pruned)</p>
Depthwise Separable Convolutions	Architectural	<ul style="list-style-type: none"> Decomposes the regular convolution, for increased efficiency: <ul style="list-style-type: none"> Depthwise convolution operation preserves depth C of input. C many "groups" of $n \times n \times 1$ kernels are used, each separately applied to each channel of the input to get C channels 1 x 1 convolutions provide desired depth While fewer in parameters, performance lags behind large ResNet variants Originally from MobileNet paper 	

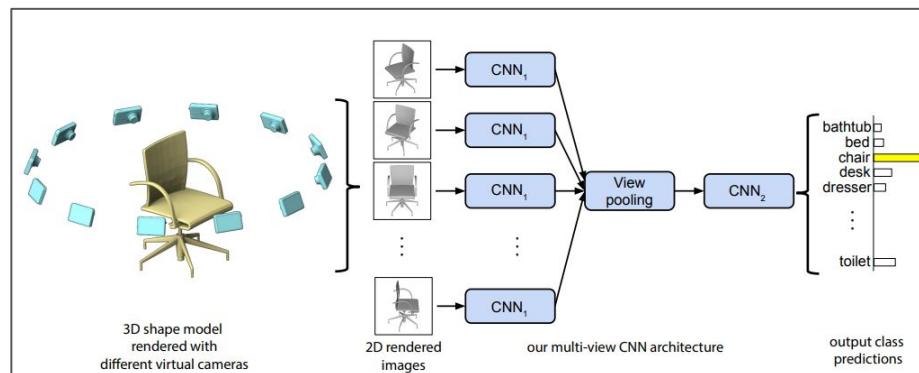
Explain some learning and automation based methods for efficient DL.

Overview of Methods for Efficient Deep Learning			
Method Name	Category	Description	Figure
Student-Teacher Distillation	Learning Techniques	<ul style="list-style-type: none"> First train a larger "teacher" model, then train a more lightweight "student" model with both the original training data and the softmax classification scores of the teacher (with KL divergence) The availability of "dark knowledge" non-one-hot information from the teacher can improve performance for the student 	
Data Augmentation	Learning Techniques	<ul style="list-style-type: none"> Useful to improve performance and data efficiency Can use techniques like RandAugment and Mixup, or even GANs to synthesize new examples Synthetic sampling like SMOTE can allow for re-balancing to make up for dataset skewness 	
Self-Supervised Learning Transfer	Learning Techniques	<ul style="list-style-type: none"> Goal: utilize auxiliary tasks in a way where we can generate virtually unlimited labels from our existing images, to learn useful representations Resulting network can then be used for transfer learning on a domain-specific downstream task Useful to improve performance and data efficiency Examples: colorization, superresolution, inpainting, jigsaw, rotation, clustering, data-aug based contrastive learning 	
Neural Architecture Search (NAS)	Automation	<ul style="list-style-type: none"> Automates NN design by searching space of filter sizes, layer types, depth, etc for best performance and/or efficiency <ul style="list-style-type: none"> Have normal cells (in/out have same spatial size) & reduction cells Several black-box optimization approaches can be taken including grid search, random search, RL, and evolutionary algorithms Differentiable NAS (e.g. DARTS) is also possible by a continuous relaxation of the search space allowing gradient-based optimization <ul style="list-style-type: none"> Discrete operation choices replaced by mixture operations in DAG graph Bilevel optimization problem: need to optimize the parameters & architecture weights. Can approximately solve by iteratively optimizing 	<p>Left: Normal vs Reduction cells (NasNet). Right: Differentiable NAS (DARTS)</p>
Hyperparameter Search	Automation	<ul style="list-style-type: none"> Automates tuning NN hyperparameters for better performance, without changing footprint Several black-box optimization approaches can be taken including grid search, random search, Bayesian optimization, RL, and evolutionary algorithms 	

3D Deep Learning

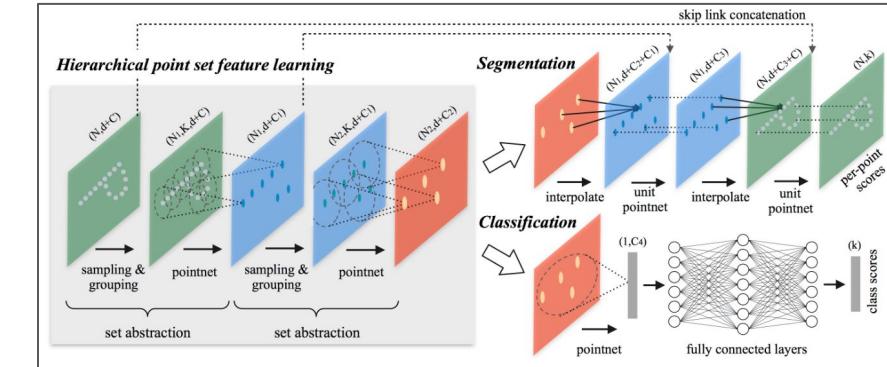
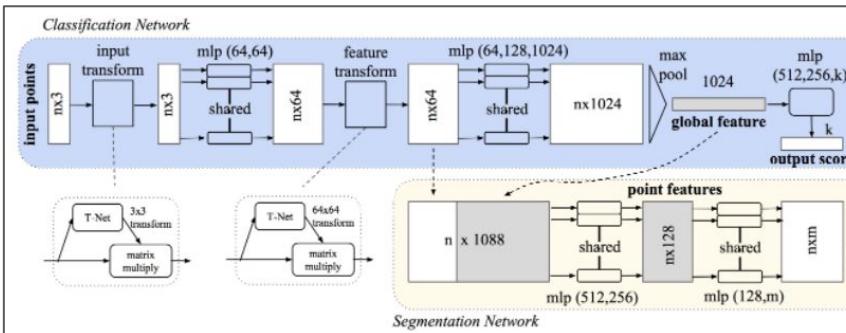
How does the MVCNN work?

- Uses view-pooling to combine feature maps of different 2D renders of 3D mesh
- View-pooling aggregates views by element-wise max across views (rather than concat or average)



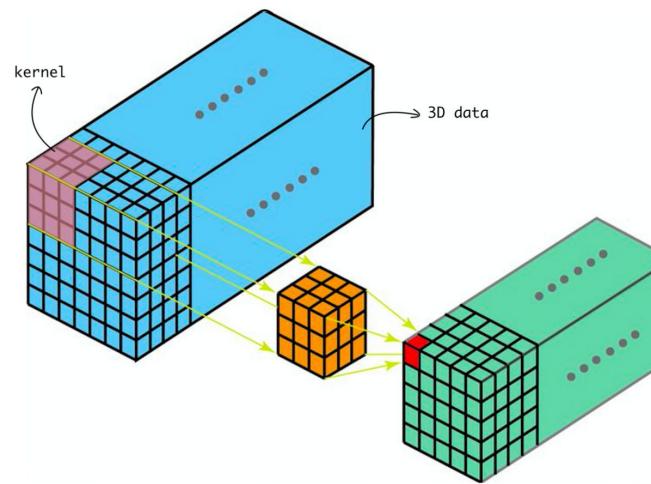
How do PointNets and PointNet++ work?

- PointNet and PointNet++ consume raw point clouds; no need to voxelize or render
- PointNet:
 - Learns both local and global point features
 - Per-point local feature vectors by shared MLPs on n individual points
 - T-Nets try to make it invariant to rotation by learning a 3×3 transformation matrix
 - Global feature vector obtained by max pooling $n \times 1024 \rightarrow 1024$
 - Applications:
 - Classify point cloud: global feature vector fed into MLP + softmax + CE loss
 - Segment point cloud: global feature vector concat with local feature vectors, per point classification
 - Useful for object part segmentation or scene semantic segmentation
 - Notes:
 - Max pooling is an example of a **symmetric function, which is order invariant**
- PointNet++:
 - Repeatedly uses PointNets to aggregate sets of nearby points
 - For segmentation, need to interpolate to obtain the original input dimensions



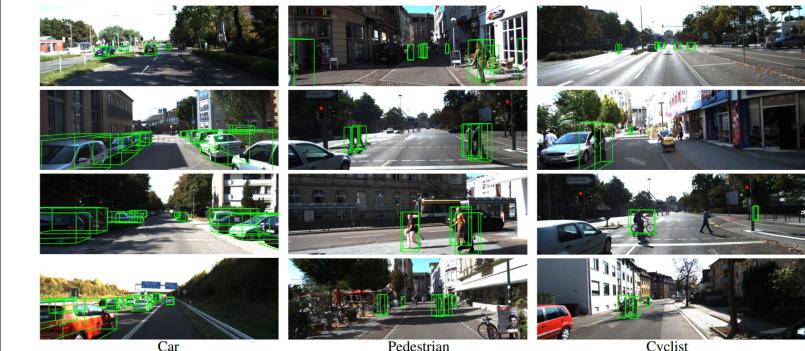
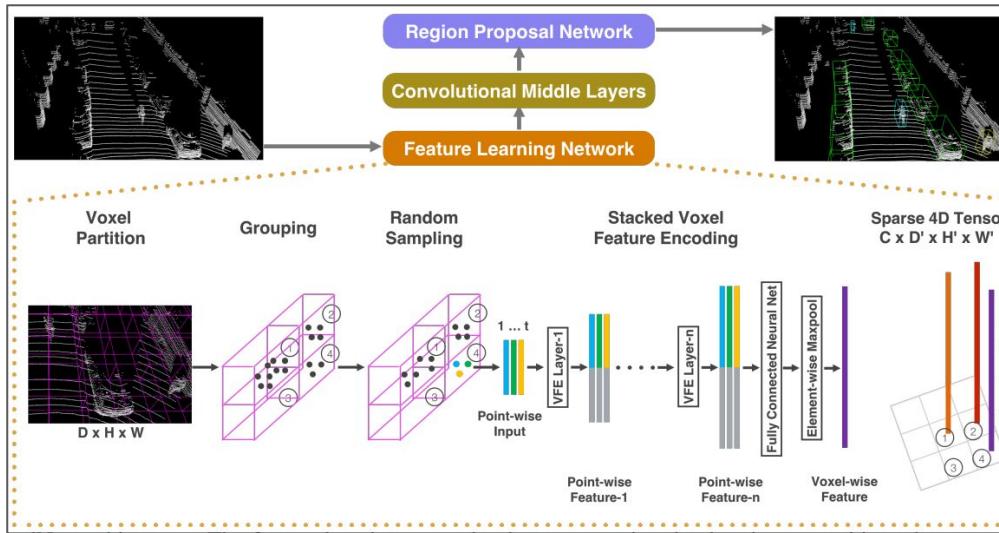
How do 3D convolutions, and deconv/transpose convs work?

- Instead of 3D $n \times n \times C$ kernels/filters, we use 4D $n \times n \times n \times C$ kernels/filters to slide over the 4D input of $W \times H \times D \times C$
 - Sliding dot product aggregates local features in 3D; width, height, and depth
- Deconvolution is similar, but uses padding in between to increase spatial size of output feature map



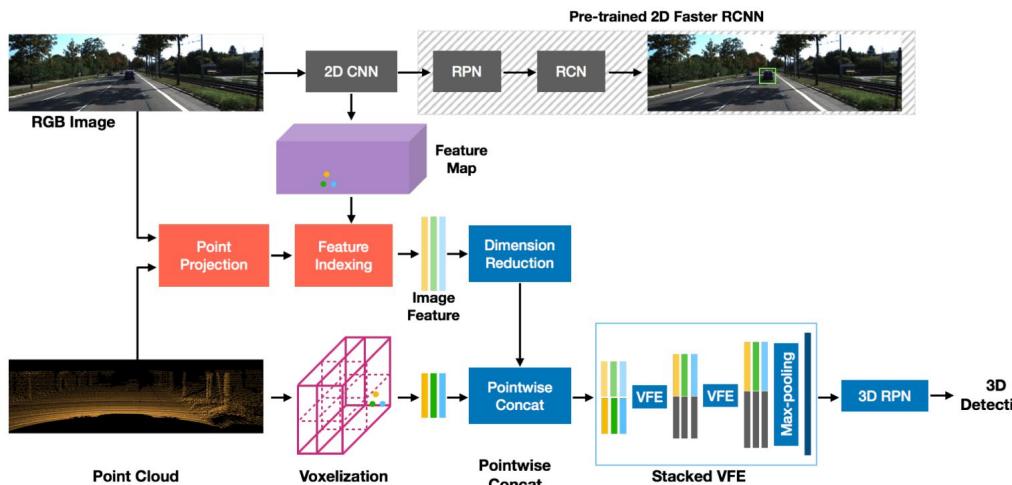
How do voxelnets work?

- Performs 3D object detection on point clouds
- Steps:
 - Partitions pointcloud into 3D voxel grid, and then applies a PointNet w/ global concat to each cell to get a sparse 3D feature map of dimensions $H \times W \times D \times C$
 - Applies 3D convolutions onto 3D feature map (still in form $H \times W \times D \times C$)
 - Merge depth and channels; flatten to $H \times W \times C$
 - RPN finds bboxes in 2D feature map, which also regress bbox
 - 7 Bbox params: xyz center, length, width, height, degree yaw about z axis



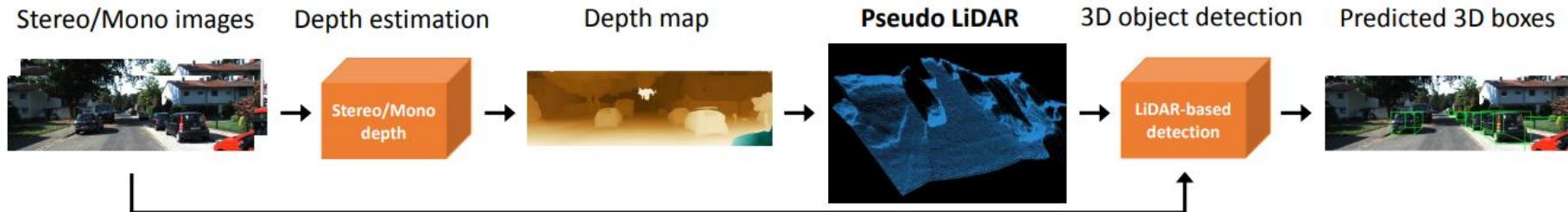
How does the MVX-Net extend voxelnets?

- Instead of 3D detection only using pointcloud data, also incorporates image data
- 3D points are projected to the image using the calibration information and learned corresponding image feature map features are appended to the 3D points using interpolation



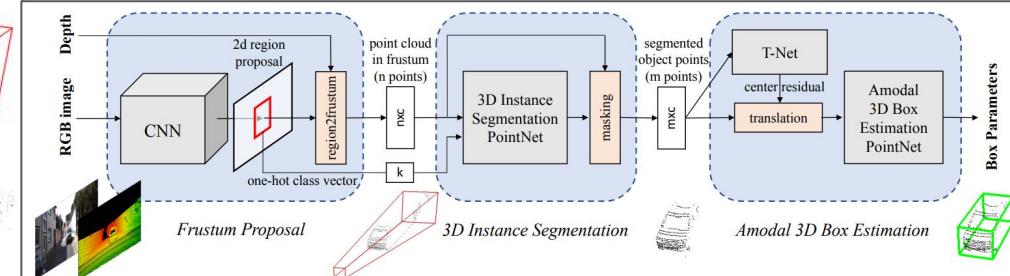
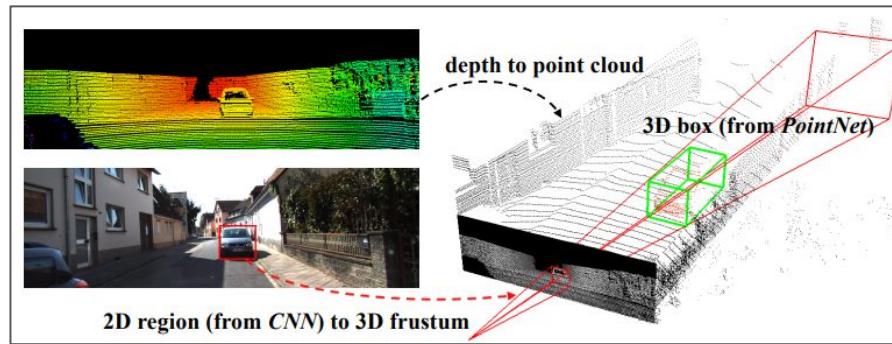
What is Psuedo-LiDAR?

- It seems like methods on lidar perform better than pure RGBD methods
- You can convert RGBD to Psuedo-LiDAR point clouds
 - Experiments suggest that the LiDAR modality is a better medium to work with than RGBD, all else held equal



How do Frustum PointNets work?

- Instead of just using a 3D-lidar based pointcloud (e.g. the voxelnet approach), uses RGBD + image setup
- Steps:
 - Make 2D detection on image
 - Project 2D BBox to a frustum of points, similar to Psuedo-LiDAR
 - Segment points w/ pointnet to remove background clutter
 - Feed 3D instance segmentation of points to regress amodal (shape-completed) 3D BBox parameters
- For maximum effectiveness, combine with BEV methods to still detect 2D occluded objects



How do graph convolutions work?

At a high level, the idea is as follows. Suppose we have graph $G = (V, E)$, containing:

- N many D -dimensional vertices $V = \{v_i | v_i \in \mathbb{R}^D\}, |V| = N$
 - This can be summarized as a feature matrix $X_{N \times D}$
- Edges E
 - This can be summarized as an adjacency matrix $A_{N \times N}$

We wish to have a neural network with layers:

$$H^{l+1} = f(H^l, A)$$

The first input is $H^0 = X$, and $H^L = Z_{N \times F}$ are the output features (of dimension F) for each vertex.

A very simple way to have propagation of features from connected nodes on the graph is to have $f(H^l, A) = \sigma(AH^lW^l)$.

This is kind of like a FC layer (HW), where edges are “selected” using the adjacency matrix.

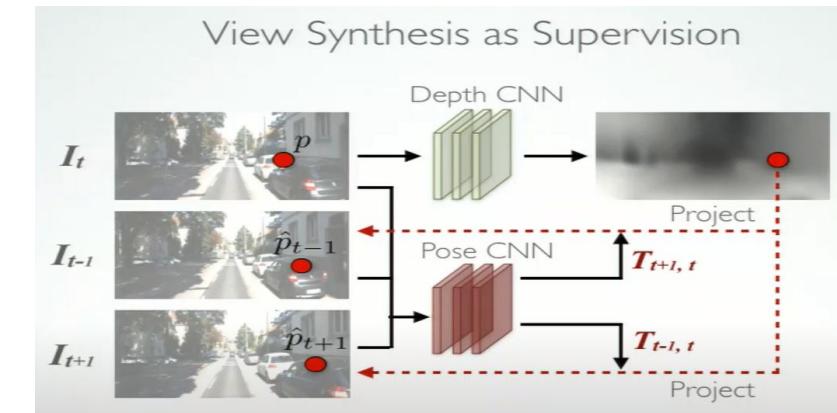
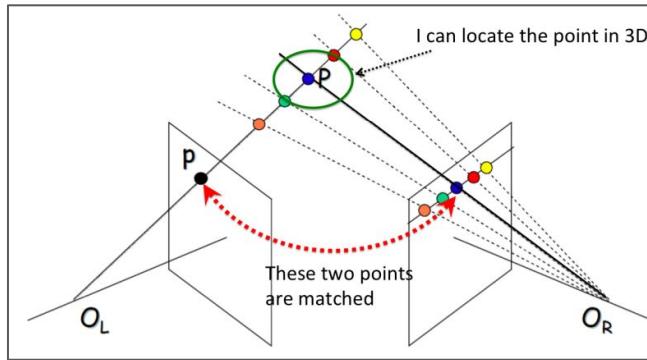
In practice, we actually use the following, which incorporates symmetric normalization and self-loops on A .

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

There are some more complicated aspects to this, involving spectral graph theory and Fourier transforms.

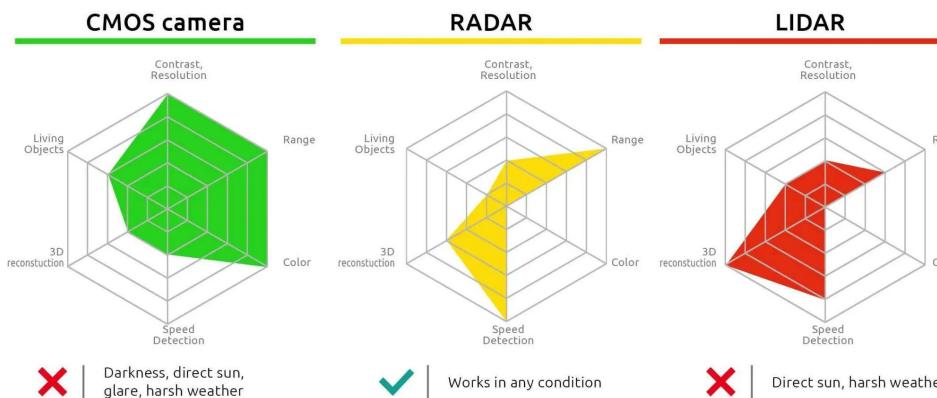
What are some ways to get depth info from vision alone?

- Stereo to get RGBD
- **SfM** based “geometric triangulation” to get pointcloud reconstruction
- Monocular depth estimation
 - fully supervised using radar/structured light as ground truth
 - Unsupervised, by using video and jointly predicting depth and camera pose
 - If the pose and depth are predicted correctly, then the projected pixels should be the same in color



Compare/contrast cameras, LiDAR, RADAR.

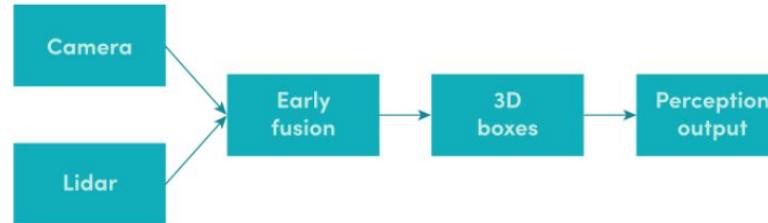
- **Camera**
 - Pros: great resolution, range, and color
 - Cons: poor depth, speed detection
- **RADAR**
 - Pros: great range, speed detection, works in any weather condition
 - Cons: poor resolution, no color
- **LIDAR:**
 - Pros: Better resolution, depth
 - Cons: mediocre resolution, no color, bad in harsh weather



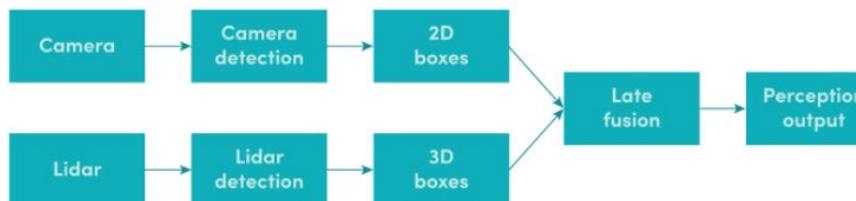
Not good enough for Autonomous Driving

Compare/contrast early vs late fusion

- When using multimodal data for a common task, you eventually need to combine the signals from multiple modalities. This can be done earlier in the pipeline, or later.
- Early fusion has the advantage of giving NNs a better opportunity to capture interaction effects between modalities



An example early fusion pipeline directly fuses lidar and camera raw data to produce perception outputs.



An example late fusion pipeline performs lidar and camera detection separately and then fuses them together to produce perception outputs.

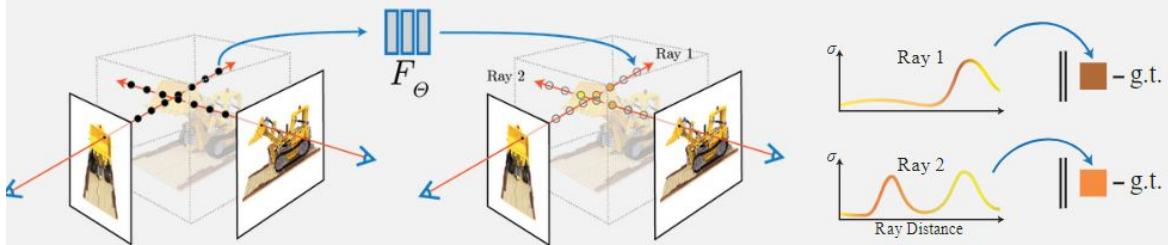
How do NERFs work?

- Learns an implicit representation using a set of images and known camera poses
- Compared to OccNet, Input also includes view; output includes emitted radiance

We present a method that achieves state-of-the-art results for synthesizing novel views of complex scenes by optimizing an underlying continuous volumetric scene function using a sparse set of input views.

$$(x, y, z, \theta, \phi) \rightarrow \begin{array}{c} \text{[} \text{[} \text{[} \\ F_{\theta} \end{array} \rightarrow (RGB\sigma)$$

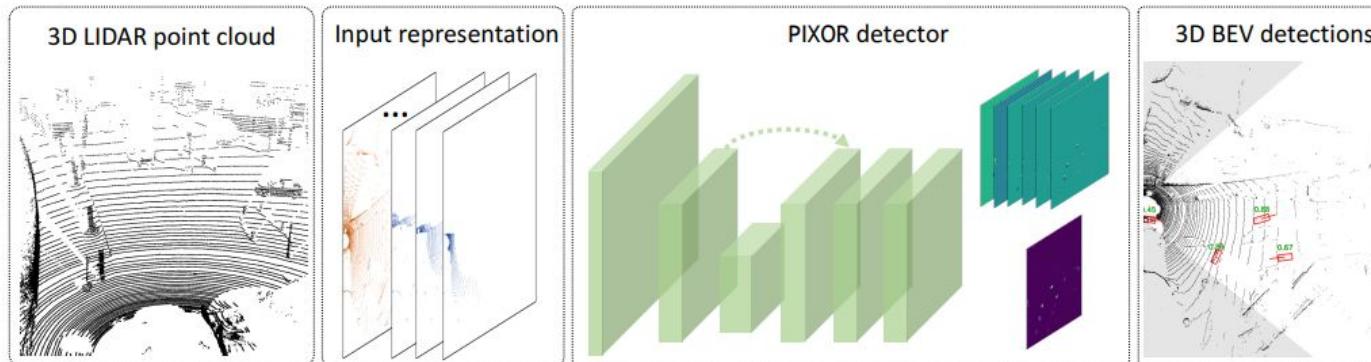
Our algorithm represents a scene using a fully-connected (non-convolutional) deep network, whose input is a single continuous 5D coordinate (spatial location (x, y, z) and viewing direction (θ, ϕ)) and whose output is the volume density and view-dependent emitted radiance at that spatial location.



We synthesize views by querying 5D coordinates along camera rays and use classic volume rendering techniques to project the output colors and densities into an image. Because volume rendering is naturally differentiable, the only input required to optimize our representation is a set of images with known camera poses. We describe how to effectively optimize neural radiance fields to render photorealistic novel views of scenes with complicated geometry and appearance, and demonstrate results that outperform prior work on neural rendering and view synthesis.

How does PIXOR leverage bird's eye view?

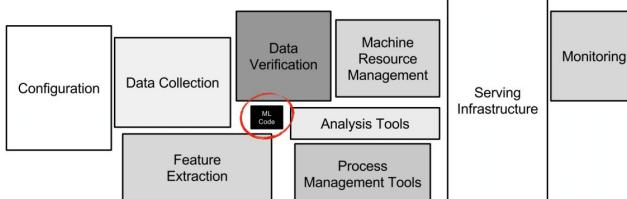
- Uses 2D conv on BEV slices of the input
- Leads to simplicity and efficiency



Full Stack Deep Learning

<https://fullstackdeeplearning.com/>

Describe the general lifecycle of a ML project. What is the “data flywheel”?

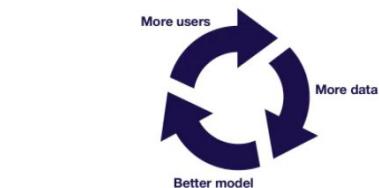


To avoid technical debt, care needs to be taken for the following:

- **Project Planning & Setup**
 - Determine goals/requirements. Is it well-defined?
 - What are the metrics? What's an acceptable failure rate?
 - Are you sure you need ML at all?
 - Allocate and setup resources
 - Hardware, language, libraries/frameworks, monitoring, hyperparam optim
 - SageMaker (integrated and easy) vs custom (no vendor lock-in, more flexible, but expensive)
- **Data Collection & Labeling**
 - Data storage, versioning, cleaning, analysis
 - *Revisit project setup if data too hard to collect*
- **Training, Development, and Debugging**
 - Establish baselines
 - Reproduce SoTA
 - Improve and debug
 - *Revisit data collection if performance is too low*
 - *Revisit goals/requirements/resources if you realize task is too hard*
- **Deployment & Testing**
 - Pilot in the lab or other controlled environments
 - Write tests
 - *Revisit training/developing if test cases fail or deployment has failures*
 - *Revisit data collection if performance is insufficient (mine for hard cases)*
 - *Revisit metrics if they appear to be good, but fail in practice*

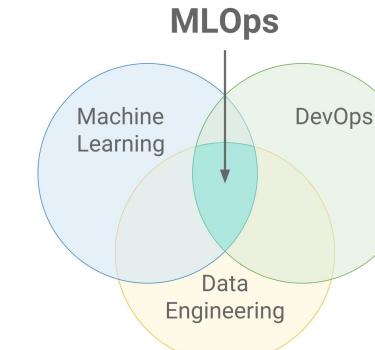
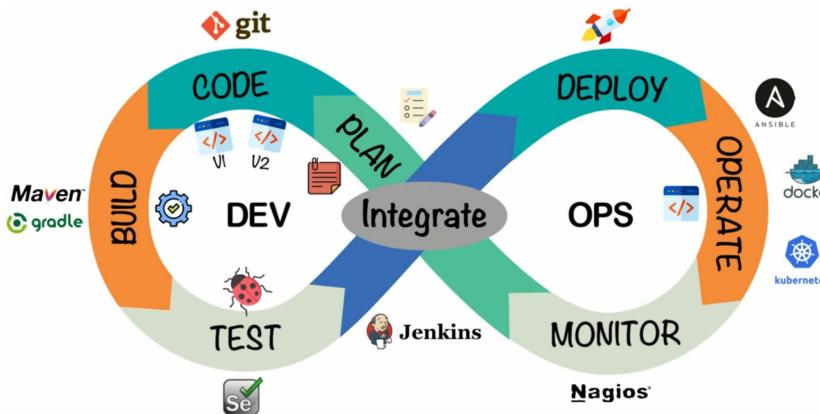
The last three stages constitutes a “**data flywheel**” – a virtuous cycle that is the “gold standard” to try to achieve for best performance.

- Quickly getting to this stage is essential for success
- Tesla is a canonical example of this
- Human annotators can be in-the-loop of this. New data can be from user feedback (“don't show more like this” button) or explicitly digging up and fixing mistakes.
- Requires some continual learning techniques



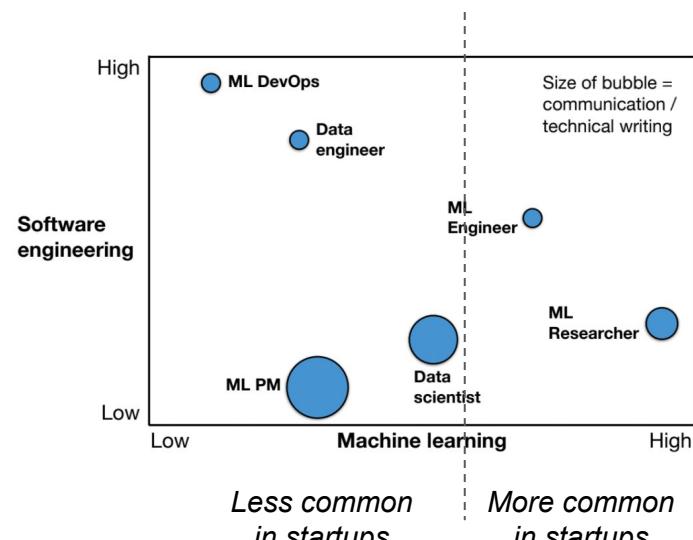
What is Agile development, DevOps, and MLOps?

- **Agile** is a software development teamwork paradigm that states some “best practices”:
 - **Working, simplicity-focused** software over comprehensive documentation
 - **Customer collaboration & frequent, iterative continuous delivery**
 - Welcoming and rapidly responding to **changing requirements**
 - Focus on **individuals, communication**, and **self-organizing teams**
- **DevOps** is an approach to software that emphasizes close collaboration between software development (traditionally only concerned with creation of the software) and the Operations/IT Team (traditionally only concerned with testing/implementation/deployment).
 - Ensures faster feedback and reduction of delays for a project
 - Usually entails several other paradigms and frameworks; complementary with agile development, CI/CD, version control, automated testing, docker/kubernetes
- **MLOps** is DevOps applied to ML. It tries to bridge ML experts, Data Engineers, and SW Engineers (DevOps).



Explain the job function, end product, and tools used for ML Product managers, DevOps Engineer, Data Engineer, ML Engineer, ML researcher/scientist, and Data Scientist.

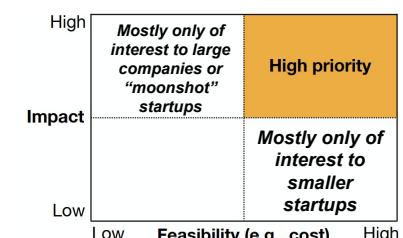
Role	Job Function	Work product	Commonly used tools
ML product manager	Work with ML team, business, users, data owners to prioritize & execute projects	Design docs, wireframes, work plans	Jira, etc
DevOps engineer	Deploy & monitor production systems	Deployed product	AWS, etc.
Data engineer	Build data pipelines, aggregation, storage, monitoring	Distributed system	Hadoop, Kafka, Airflow
ML engineer	Train & deploy prediction models	Prediction system running on real data (often in production)	Tensorflow, Docker
ML researcher	Train prediction models (often forward looking or not production-critical)	Prediction model & report describing it	Tensorflow, pytorch, Jupyter
Data scientist	Blanket term used to describe all of the above. In some orgs, means answering business questions using analytics	Prediction model or report	SQL, Excel, Jupyter, Pandas, SKLearn, Tensorflow



[Project Planning & Setup Phase] How do you judge if a ML project will have high impact?

Central question is, **where can you take advantage of “cheap, but good prediction”?**

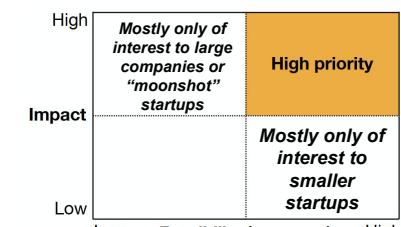
- AI, fundamentally, reduces the cost of prediction/decision making
- The prediction should be currently expensive to perform, and be ubiquitous
- Where is there friction in the world, where you can automate complicated/costly manual processes?
- These predictions should above all else generate business value



[Project Planning & Setup Phase] What are the 3 factors that affect feasibility of a project?

Factors that affect feasibility:

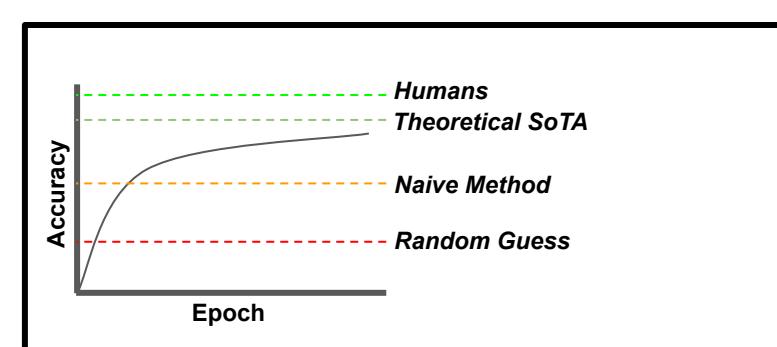
- **Problem Difficulty**
 - Does good published work exist on similar problems?
 - How much novel technical/conceptual effort is needed?
 - Are computational requirements currently tractable?
 - Is it based on ML topics that are established to “work” (e.g. detection) and can be engineered, or is it still a fundamentally challenging research problem (e.g. RL, unsupervised learning, VQA, video, 3D, out-of-distribution generalization, few-shot)
 - Can it be made “less autonomous” and also incorporate classical techniques known to work relatively well?
 - Ideally, in industry want to focus on exploitation more than exploration
- **Accuracy Requirement**
 - How costly are wrong predictions? (e.g. recommender systems vs AV/medical)
 - Is it imperative to “fail safely”? Can guardrails be put in place?
 - The need for higher accuracy scales exponentially in cost
 - This can sometimes be mitigated by good, non-intrusive or suggestive product design, or setting user expectations (e.g. face ID is not always accurate)
- **Data Availability**
 - How hard is it to acquire data?
 - How expensive is data labeling?
 - How much data is needed?



Overall, an ideal project would have promising results in relevant publications, allow a reasonable margin for error, and be easily to collect/label data for.

[Project Planning & Setup Phase] How should you define metrics to optimize for your ML project? What about baselines?

- There may be many metrics, but ultimately you should focus on one
 - E.g. for classification, accuracy, precision, recall, ROC curve
- This can be done by:
 - Weighted average of metrics
 - Thresholding n-1 metrics (that are not too important, e.g. latency, # params, goal is just to not make much worse), and focus on driving down the nth metric (e.g. accuracy)
 - This can be rotated iteratively
- You should also have baselines and upper bounds in mind (e.g. human annotations, SoTA in papers, classical/simple/naive methods, random guessing) as a guide



[Data Collection & Labeling Phase] How should you approach a new dataset and data pipeline?

- Usually proprietary data is involved
- Spend a LOT of time exploring data
 - Manual annotating to figure out edge cases, make expectations clear
 - Fully understand different subsets/domains of the data
- Try to keep things simple, don't over engineer pipeline
- For labeling, use third party interfaces. Key is to train annotators well, and annotate a “gold standard” yourself
 - May want to outsource to a specific data labeling company since it is a lot of orthogonal management (temporary labor, quality control)

[Data Collection & Labeling Phase] Draw a diagram of a standard ML data storage flow.

Data Lake

Unstructured, raw data



Object Storage

Filesystem + API, with built-in versioning and redundancy over distributed system.

Use to store training data (e.g. images)



Data Warehouse

Stores databases.



Database

Database

Used for fast access to structured, tabular (i.e. relational) data, usually with the SQL language. Has features like query, sort, filter, grouping, add/remove.

Use to store labels/metadata.



Database

Database

Training Model

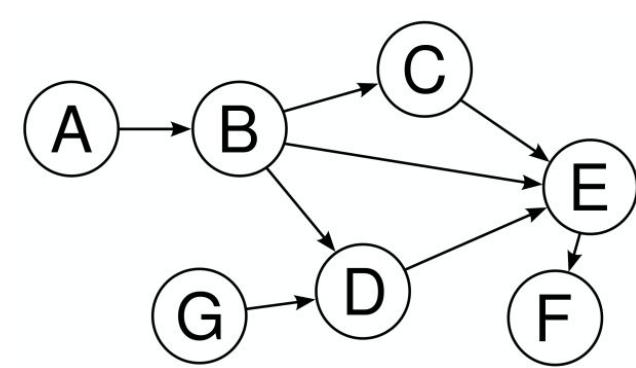
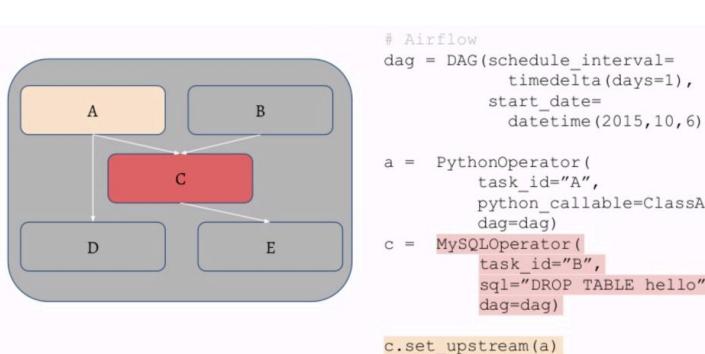
Copy relevant files to disk. Use data versioning.



Amazon EC2

[Data Collection & Labeling Phase] What are the considerations & approaches to consider when you have complicated data processing dependencies?

- You may have a recurring (cron-like) DAG for your data pipeline
 - E.g. may require metadata, user features, and outputs of image classifiers (e.g. style, sentiment, content)
 - Will have programs and database dependencies
- Desired features:
 - Distributed over many machines (i.e. needs an orchestrated effort if things fail)
 - Should have scheduling, monitoring, alerting, and caching features for minimal recomputation
- Systems for tackling this are called **workflow management systems**
 - Examples include Apache Airflow, Prefect, and Dagster



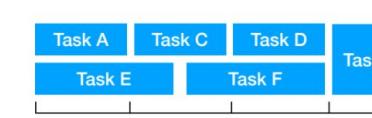
[Data Collection & Labeling Phase] What are the 4 levels of data versioning?

- **Level 0: No versioning**
 - This is a problem, since ML model deployments are part code, part data. If you only version code, the model itself isn't fully versioned
 - Will face inability to get back to a previous level of performance
- **Level 1: Snapshot all data used at training time**
 - Similar to archaic way of versioning code ("just put in a zip file")
 - Very hacky, not integrated with code versioning
- **Level 2: Data versioned with code together**
 - Store files themselves in S3, with unique ids. Training data used stored as JSON, with relevant metadata that refers to ids
 - Can include into git, integrated with code
- **Level 3: Specialized solutions for data versioning**
 - Avoid until necessary
 - Examples include DVC, Pachyderm, Quill

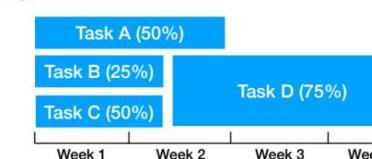
[Training & Debugging Phase] Why is performant DL code hard write well? What are the attributing factors to bad performance?

- DL code hard to debug since most bugs are invisible (fail silently)
- Hard to tell in advance how hard/easy something is
 - Project planning should be done probabilistically, not statically, and a portfolio of approaches should be tried in parallel
 - Be careful not to overhype results to project manager
- Poor model performance can be attributed to up to 4 different factors:
 - **Bugs**
 - **Data vs model fit**
 - **Hyperparameter choices**
 - **Data construction** (e.g. class imbalance, distribution issues, noisy labels, amount)

- From:



- To:



[Training & Debugging Phase] Compare data vs model parallelism when doing distributed training.

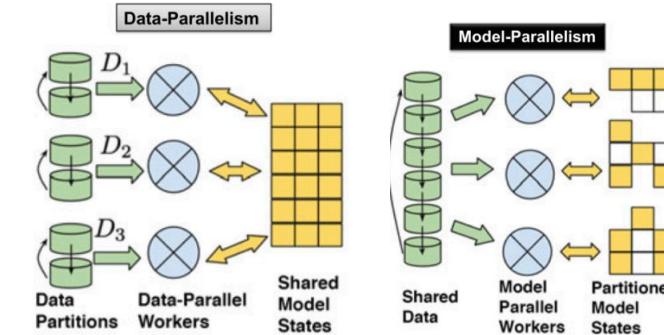
Distributed training is a must for large datasets and large models, and involves multiple GPUs and/or machines to train a single model. Note that forward computations and backprop are inherently serial in nature, so one can think of these methods as “workarounds”.

- **Data parallelism**

- Helpful for large datasets
- Splits batch into multiple GPUs (each gpu must fit a copy of the model on it), evaluate/backprop on each, and combine to get overall gradient to update across all GPUs
- Generally simple to do
- [Hogwild paper](#) (NIPS 11, 2000+ cit) showed that you don't even need locks; SGD works even if the shared model sometimes receives older gradients.

- **Model Parallelism**

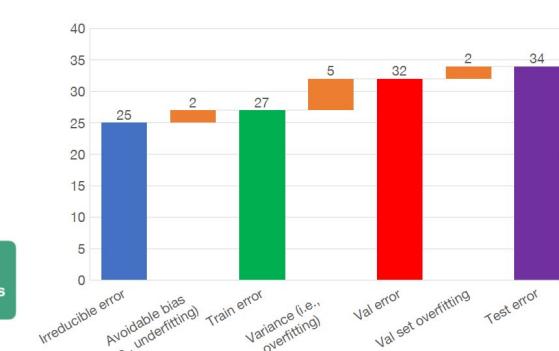
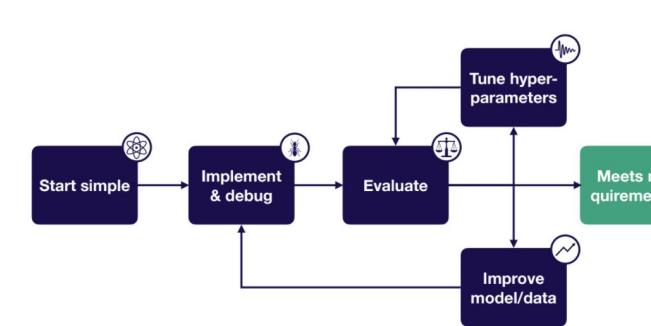
- Needed when a model does not fit on a GPU
- Splits the model parameters on separate GPUs
- Introduces a lot of complexity, so see if this can be avoided by getting larger GPUs
- One way this is implemented (AWS Sagemaker) is by splitting consecutive layers into different GPUs, and having an execution pipeline where different devices can work on forward and backward passes for different data samples at the same time



[Training & Debugging Phase] Explain in detail the workflow for DL development. How should you start? What are the best ways to address over/underfitting?

- Key idea: start simple and gradually increase complexity; be obsessed with visualizing every possible thing
- Start simple:
 - Use **sensible default configurations** (adam, relu, normal initialization)
 - **Don't use regularization or data aug** at first. Common source of bugs.
 - **Very lightweight implementation**, very few lines of code (each line is a chance for bugs). Try to use reliable off-the-shelf components.
 - Very simple data pipeline, **small training set**
 - **Overfit to a batch**
 - Choose sensible metric
- Use **debugger** to catch issues
- Check **bias (train error vs irreducible error)** and **variance (val error vs train error)** decomposition to determine overfitting or underfitting
 - Test error = irreducible error + bias + variance + val overfitting
- **Hyperparameter search** methods: Grad student descent, coarse-to-fine grid/random search, bayesian optimization
- Best ways to address **underfitting** (in order): Make model bigger, reduce regularization, change model closer to SoTA, tune hyperparams, add features
- Best ways to address **overfitting** (in order): Collect more data, normalization, data aug, regularization, early stopping, reduce model size
- Use "tricks" to squeeze a bit more performance, e.g. ensembles

- Start simple
 - Choose the simplest model & data possible (e.g., LeNet on a subset of your data)
- Implement & debug
 - Once model runs, overfit a single batch & reproduce a known result
- Evaluate
 - Apply the bias-variance decomposition to decide what to do next
- Tune hyper-params
 - Use coarse-to-fine random searches
- Improve model/data
 - Make your model bigger if you underfit; add data or regularize if you overfit



[Training & Debugging Phase] What is the difference between continuous integration (CI), continuous delivery, and continuous deployment? What are some tools to enable this?

- **Continuous Integration:**

- Devs merge their changes back to the main branch as often as possible, e.g. few times a day. Merges are only allowed if they pass a series of automated tests.
- Generally leads to less bugs, integration issues, and better testing

- **Continuous Delivery:**

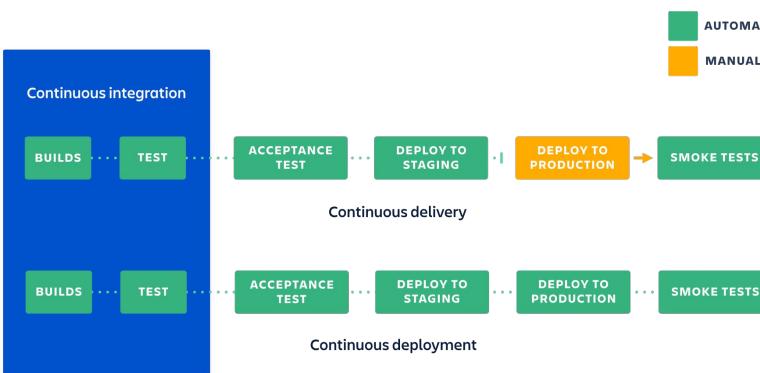
- Extends CI by automatically deploying all code changes to the production environment after the build stage
- Automated release process; can deploy app at any time by clicking a button. Releases to production are frequent, but need to be manually triggered.
- Don't need to spend days preparing for a release; leads to faster iteration

- **Continuous Deployment:**

- Extends continuous delivery by automatically releasing to customers after a change passes the automated checks
- Customers see a continuous stream of improvements, e.g. every day instead of every quarter. No more "release day".

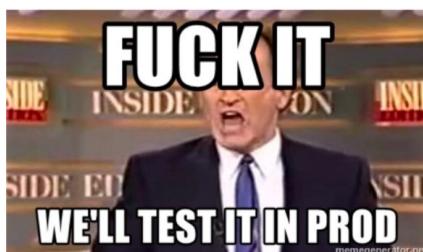
- Some SaaS exist for CI/CD, such as CircleCI and Github actions, but generally don't have GPUs

- For CI on your own hardware, can use Jenkins



[Testing Phase] Define the following: unit tests, integration tests, end-to-end tests, smoke tests, testing in prod.

- **Unit tests:** test the functionality of a single piece of the code (e.g., a single function or class)
- **Integration tests:** test how two or more units perform when used together (e.g., test if a model works well with a preprocessing function)
- **End-to-end tests:** test that the entire system performs when run together, e.g., on real input from a real user
- **Smoke/sanity test:** Simple tests that cover the most important functionality, such as if the program even runs or if important things are responsive.
- **Testing in Production:** Traditionally, tests done offline, to prevent shipping bugs. However, survey shows that only about 15% of bugs can be found this way. This can be a good approach with sufficiently advanced monitoring, roll-back mechanisms, canary deployments with A/B testing, etc.



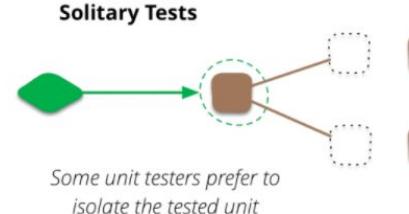
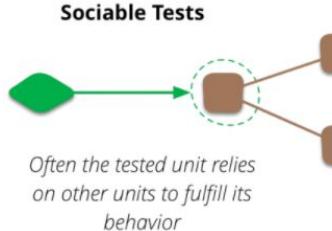
[Deployment Phase] What are some pros and cons of A/B testing?

- **Pros**
 - Get clear evidence
 - Can test new ideas in a safer way (through canarying)
 - Can optimize one step at a time
- **Cons**
 - Can take a long time to see results, especially for a service with smaller userbase or with small changes
 - Significant overhead
 - May be useless if the issue is more fundamental (e.g. modifying button appearance vs optimizing website latency, cost, bugs)

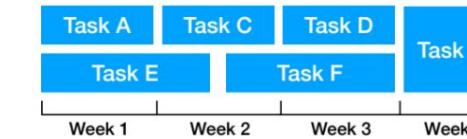


[Testing Phase] What are some ML testing best practices?

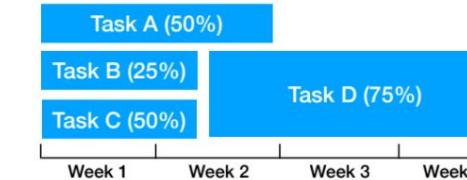
- Tests should be automated, reliable, fast, and go through its own code review process
- Tests need to pass before merging into main branch
- Rough split of 70/20/10 between unit, integration, and end-to-end tests
- Compared to traditional, “static” software:
 - Need to test whole **ML system** (including data, training, etc), not just the ML model
 - Need to “**close the loop**” between **performance metrics and business metrics**
 - Constantly changing; need to monitor/test in prod
- Evaluate on relevant “slices” (subsets/clusters) of your dataset, for example country or age group
- (Controversial) Try to rely on solitary tests more (i.e. tests that don’t rely on real data from other units, and instead use fake data)
- (Controversial) Check test coverage (but note that this doesn’t take into account quality)



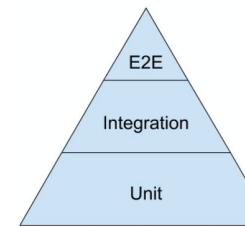
- From:



- To:



Slice	Accuracy	Precision	Recall
Aggregate	90%	91%	89%
Age <18	87%	89%	90%
Age 18-45	85%	87%	79%
Age 45+	92%	95%	90%



[Testing Phase] Draw a diagram of an ML system, and how you can test it.

Training Infrastructure Tests

- Avoid bugs in training pipeline
- Train on a single batch or epoch, to make sure it can overfit

Evaluation Testing

- Fully understand performance characteristics; see if guardrails are necessary
- Eval on all metrics, datasets, and slices
- Invariance robustness testing
- Compare to previous models, using thresholds. Avoid performance "degradation leaks".
- Simulation testing, if possible (needs rendering)
- Test for privacy, fairness, bias

Basic Pred. Tests

- Checks basic prediction functionality
- Load pretrained model, test on few key examples

Shadow Tests

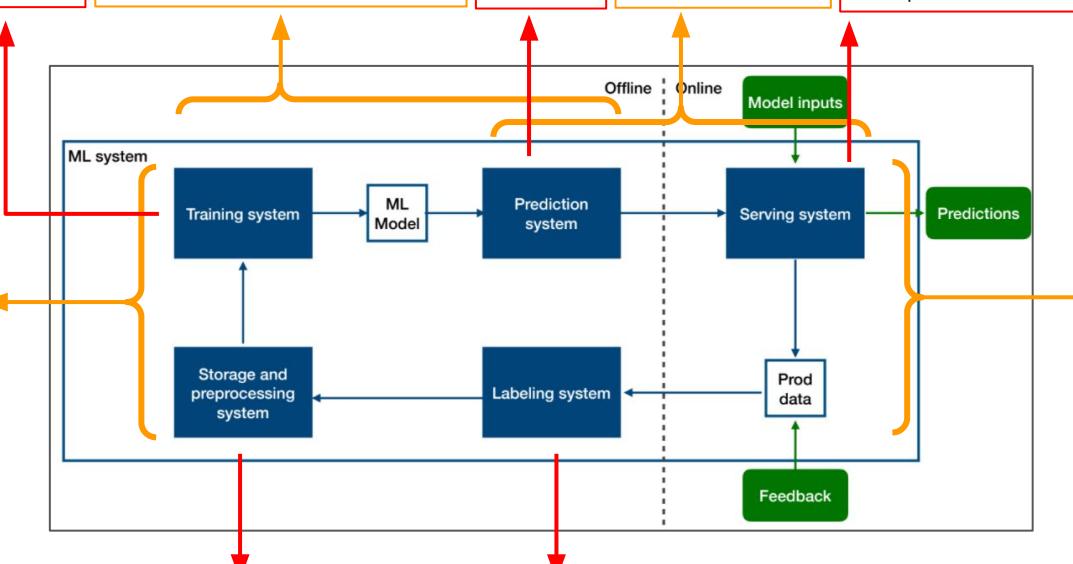
- Checks prod. bugs and compare performance w/ old model
- Run new model in parallel with old model in production, but in "shadow mode"
- Unlike dev/sandbox env, more realistic

A/B Tests

- Understand how new model affects user experience and business metrics
- Canary model on a tiny fraction of incoming requests
- Compare metrics on control and variation cohort

Training Tests

- Makes sure training is reproducible with current architecture
- Pull a fixed dataset and train it
- Check to make sure performance is consistent or better than before



Data Expectation Tests

- Catch data quality issues before training
- Check for null values, numerical ranges, and formatting using rules and thresholds
- Example package: "Great Expectations"

Labeling QA

- Catch poor quality labels
- Train labelers well
- Assign them a "trust score"; promote best performers to be "QA experts"
- Aggregate labels by voting

Production Monitoring

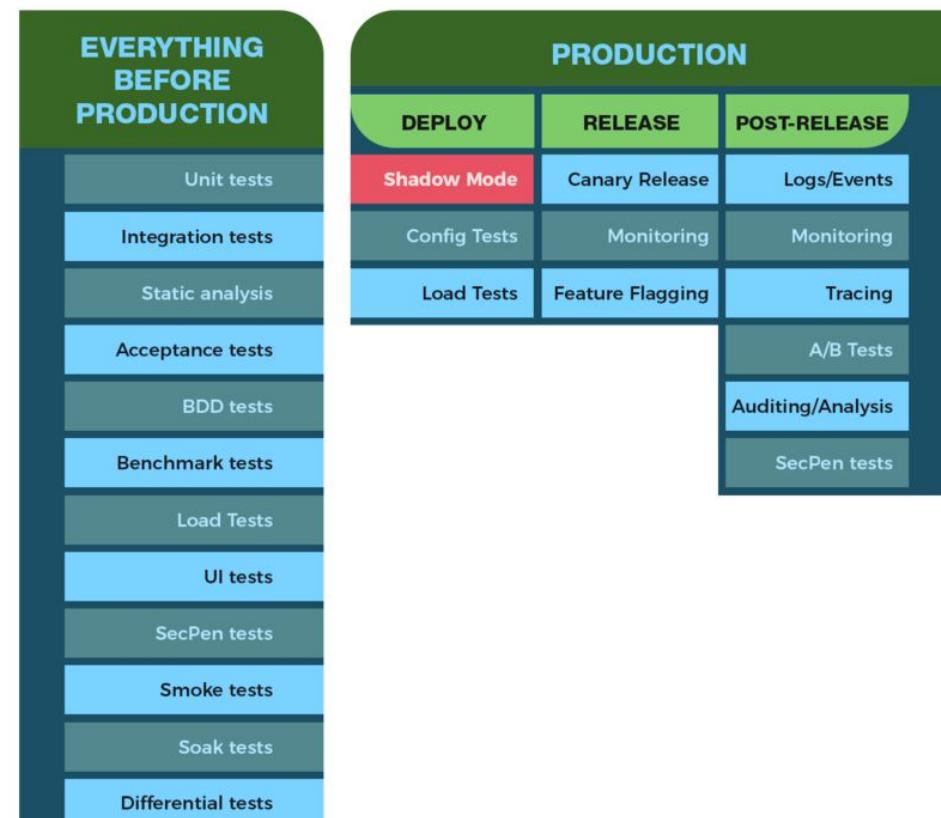
- Want to keep model healthy in deployment
- Extremely important; what's used to produce next model and decide what new data to train on
- Often hard to do directly if you don't have labels; can instead check **business metrics** and **model inputs/predictions** for irregularities
- Need to check for bugs, outliers, class imbalance, **concept drift** (dependent var. changes)
- Data drift** (independent var. changes) can be detected by comparing a sample of previous data and current prod. data using distance metric. The data can be features, or something more basic like HSV.

- Don't use KL-Divergence, too sensitive and range dependent (div by zero)
- Use **Kolmogorov-Smirnov Stat** (max dist between histogram CDF) or **D1 distance** (sum of histogram CDF distances)
- Can experiment with more complex metrics like wasserstein dist or earth mover's dist.
- Or, use **rule-based metrics** (w/ great expectations, e.g. min, max, mean in acceptable range, Col A > col B, etc)



[Testing Phase] State the testing options before vs after production (deployment/release/post-release).

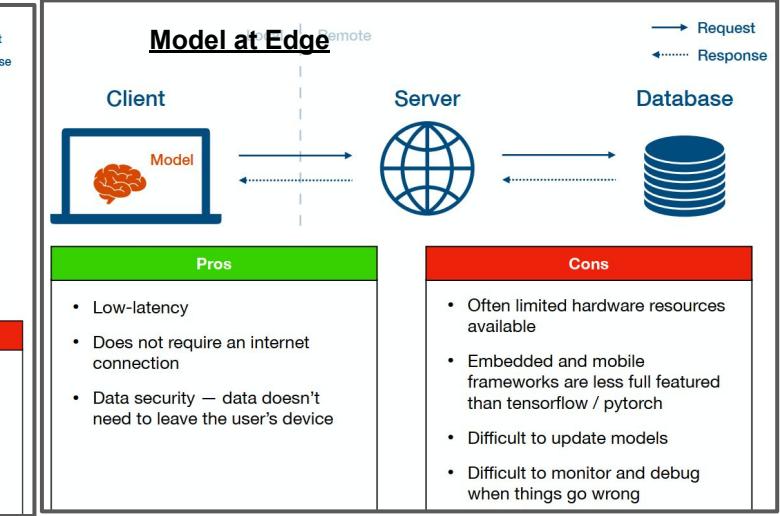
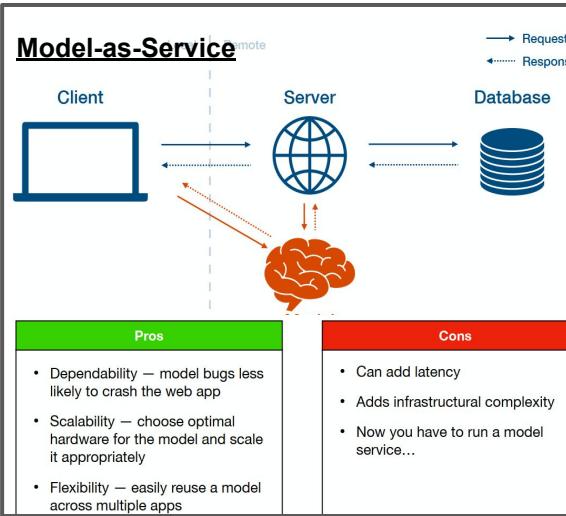
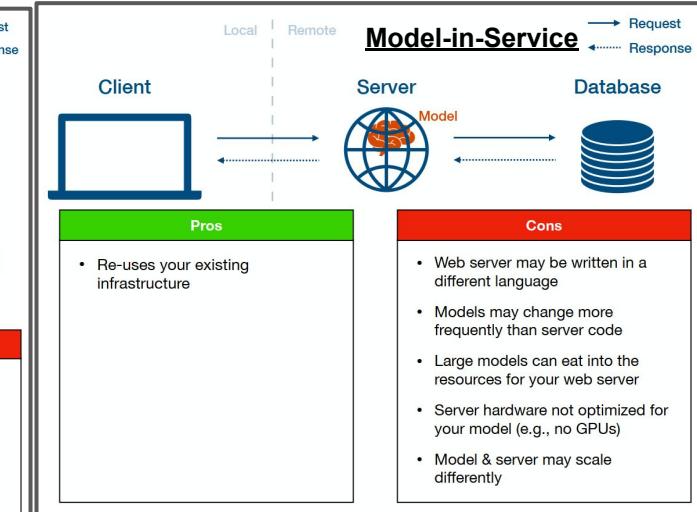
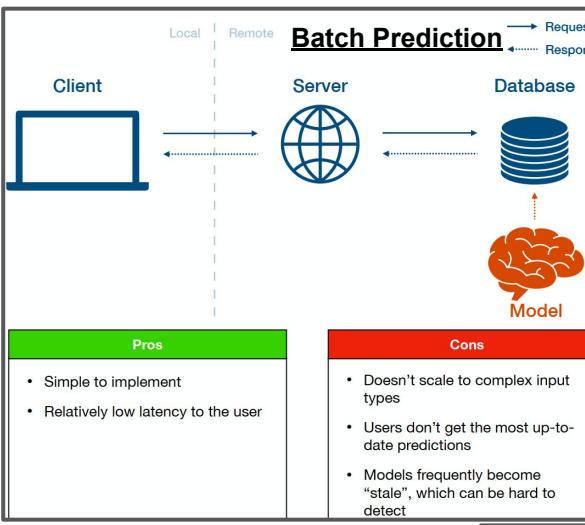
- Note that deployment need not expose customers to a new version of your service, while release does
- Deployment/rollout is having a new version of your system and all its dependencies running on production infrastructure
- Releasing/shipping is switching customer-facing systems to using that new version



[Deployment Phase] Discuss the

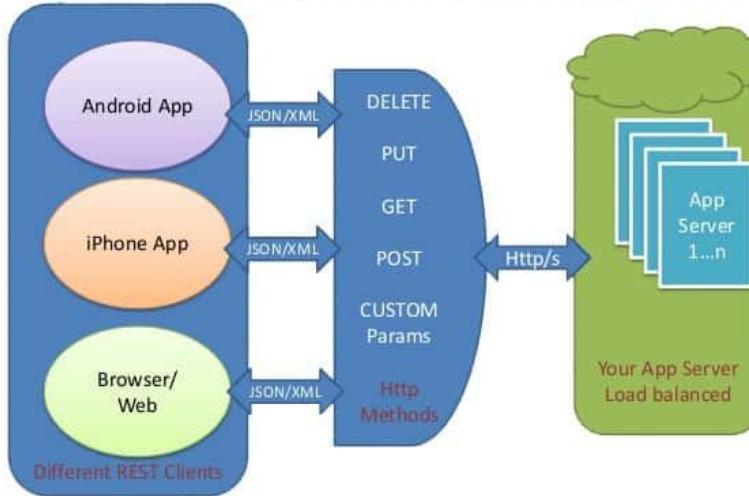
pros/cons of the following deployment models:

Batch Prediction,
Model-in-service,
model-as-service,
model at edge.



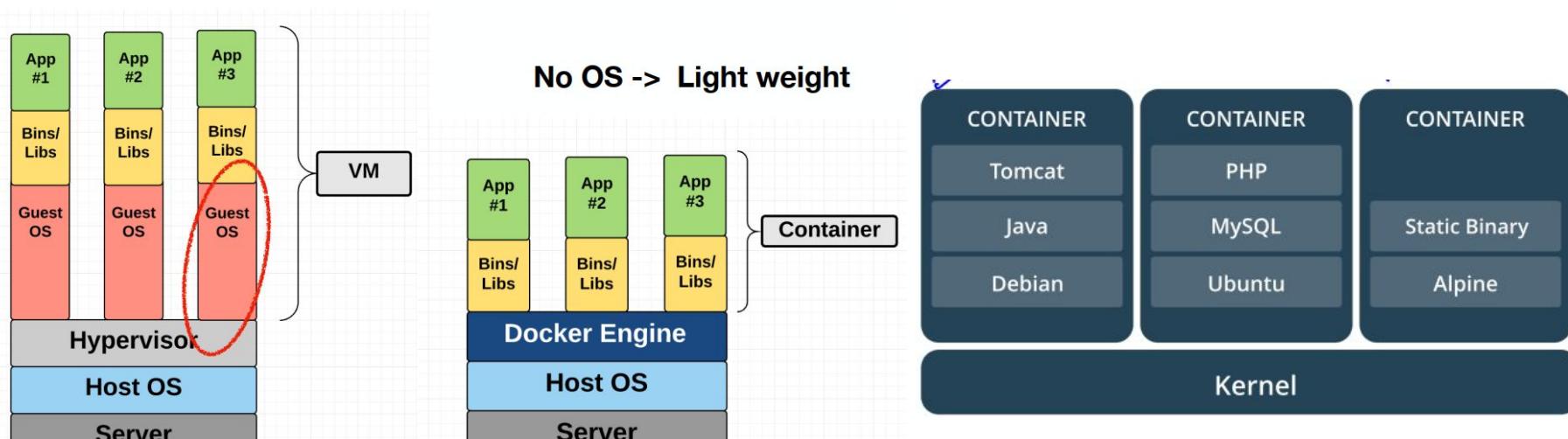
[Deployment Phase] Explain what a REST APIs is, and its advantages.

- A widely used, standardized, simple format for the access of resources and communicating over the internet, when it comes to client-server models
- Works over HTTPs and uses GET (read), POST (create), PUT (update), and DELETE, a server destination URI/URL, along with simple data formats like JSON and XML. Can also contain headers with auth data.
 - Client sends request, server sends response
- Crucially, is **stateless** on the server side; each request processed independently
 - If notions of a session are required, then this is managed on the client-side, and passed onto other services as necessary by the client.
 - This creates a clear distinction between front-end (client) and back-end (server)
- The stateless feature allows easy **horizontal scaling** (no state syncing required) and **caching**.
- Can have an actual server running 24/7, or use something like AWS Lambda (function as a service FaaS, pay-as-used but can have cold-starts sometimes. Considered “serverless” since you don’t have to manage them)
 - Typically uses a dependency management system like docker
- Can be implemented in python using packages like Flask or FastAPI



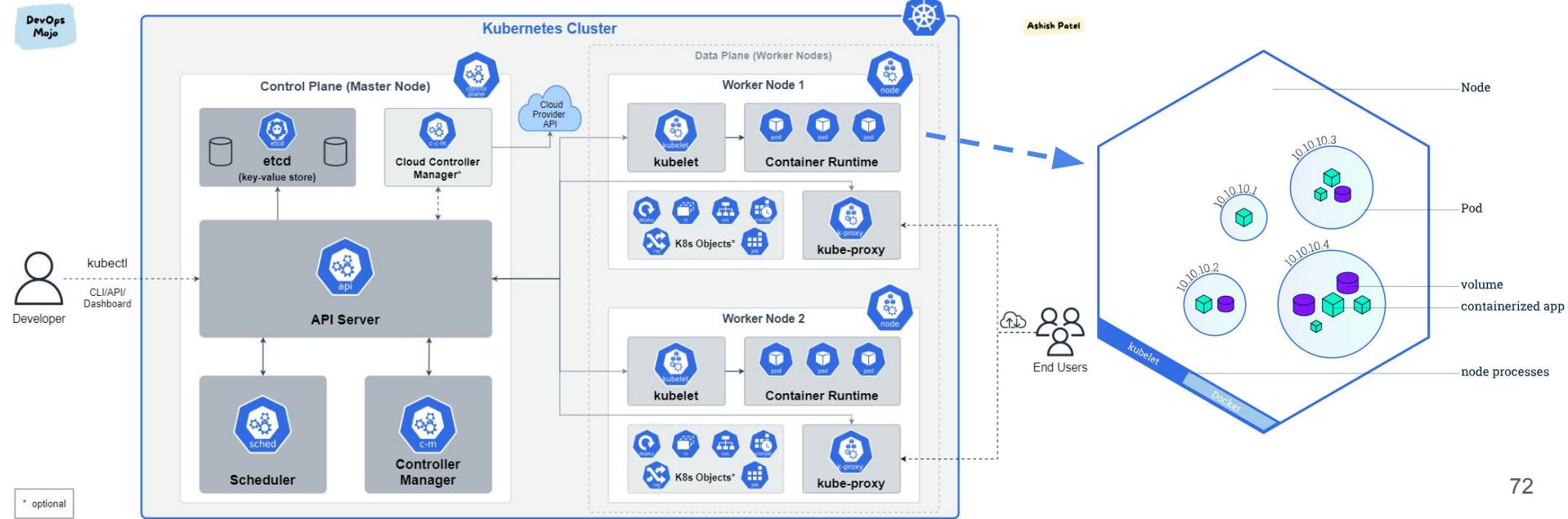
[Deployment Phase] What is docker? Name some characteristics and advantages of using it.

- Docker is a platform that provides OS-level virtualization
- It provides several benefits compared to “bare-metal” installation or a virtual machine:
 - Helps manage dependencies (now, even the dockerfile can be versioned)
 - Portability, replicability, and isolation
 - Helps for testing, rolling back, and deploying
 - Better performance than virtual machines
 - Scalability, when combined in a distributed way with a docker orchestration software like kubernetes
- A container can be created for every discrete task, which may have different dependencies or be on different OS platforms. For example, a web app might have 4 containers: web server, database, job queue, and worker.



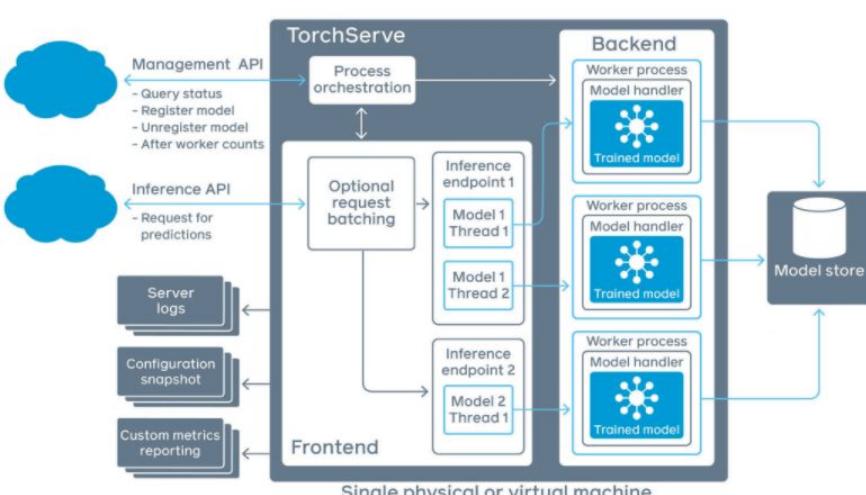
[Deployment Phase] At a high level, how can computation in general be scaled in kubernetes?

- A **Node** is a worker (physical/virtual) machine
 - A node can have several **pods**, which in turn have one or more containerized (e.g. docker) apps, an IP address, and volumes
 - Each node has a **kubelet process** which communicates with the control plane and manages pods
- The **control plane** manages one or more nodes
- The entire ecosystem of control plane + Nodes is called a **cluster**
- A **service** is an abstraction that defines a set of (fungible) pods, and a policy to access them. This makes them exposable to an RESTful API under a common outside IP address, that can be dynamically scaled horizontally; frontends don't care which backend pod is actually used (it's chosen by a selector).
 - This contrasts with vertical scaling, which would be allocated additional GPU/CPU/Memory resources to a pod instead of making more pods



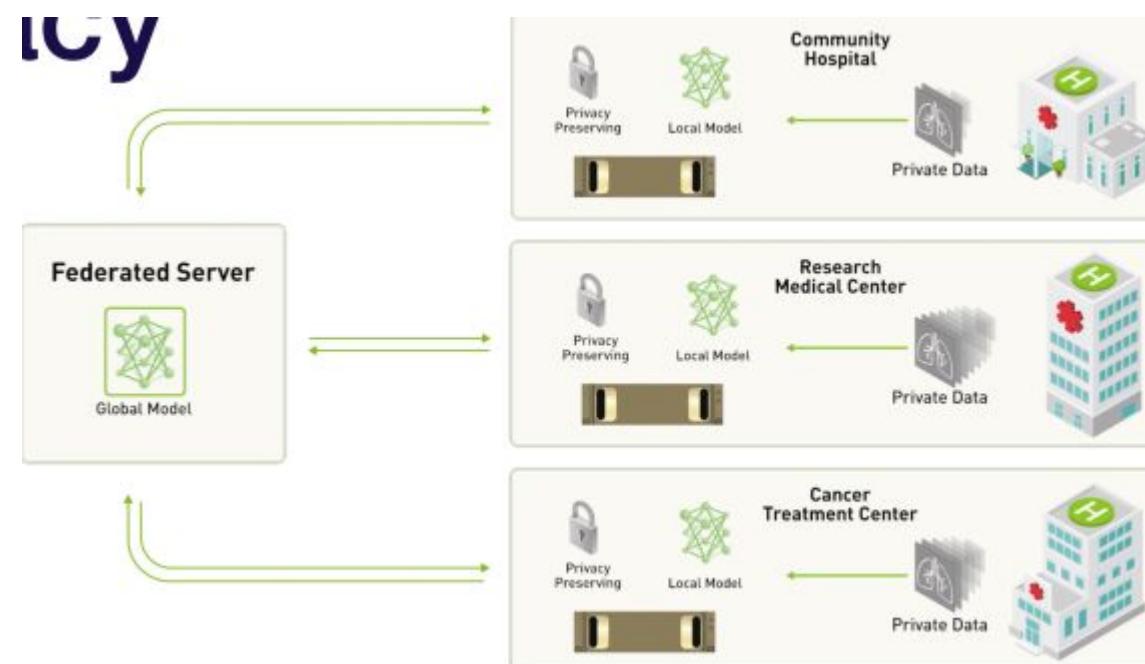
[Deployment Phase] Describe what serving a model entails and provide an example way that this can be done in practice at scale.

- This is usually done as a REST API, where the model is treated as a service. The service itself can be implemented in python using packages like Flask or FastAPI, and wrapped along with other library/file dependencies as a docker container using a dockerfile.
- For scalability, you need to deploy your container in a distributed fashion
 - Several tools exist for this that extend kubernetes for ML, such as TorchServe+Kubernetes (KServing from the Kubeflow project is an integrated solution that also allows things like canarying that uses TorchServe and Kubernetes under the hood)
 - These tools allow extra things like batching and GPU handling
 - Alternatively, one can deploy your REST API as a serverless function, but cannot use CPU



[Deployment Phase] What is Federated Learning, at a high level?

- Federated Learning: training a global model from data on local devices, without ever having access to the data
- Still an area of active research



Machine Learning Implementation

What is the general skeleton for optimizing a neural network?

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, stride=1, padding=0) # input, output, kernel size
        self.pool = nn.MaxPool2d(2, 2) # kernel size, stride
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    # x should be of a batch of 3 x 32 x 32 images
    # 32 -> 28 -> 14-> 10 -> 5
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

torch.manual_seed(0)

# setting up data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)
```

```
device = torch.device('cuda:0')
net = Net()
net.to(device)
net.train()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()
loss_record = []

# Training
for epoch in range(2):
    for i, data in enumerate(tqdm(trainloader)):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs.to(device))
        loss = criterion(outputs, labels.to(device))
        loss.backward()
        optimizer.step()
        loss_record.append(loss.item())
    torch.save(net.state_dict(), './cifar_net.pth')
    # net.load_state_dict(torch.load('./cifar_net.pth'))

# Testing
correct = 0
total = 0
net.eval()
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images.to(device))
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels.to(device)).sum().item()
print('Accuracy of the network on the 10000 test images:'.format(100 * correct // total))
```

How does pytorch's backpropagation system (autograd) work?

- Recall that there are two steps:
 - **Forward Propagation:** NN makes its best guess about the correct output, by running the input data through each of its functions
 - **Backwards propagation:** NN adjusts its parameters proportionate to the error in its guess, by collecting derivatives and optimizing parameters through SGD
- Backprop is started when you call `loss.backward()`
 - Autograd calculates and stores gradients for each model in their `.grad` attribute, if their `requires_grad` attribute is true.
- `optimizer.step()` initiates gradient descent to actually update the weights
- All this is managed through a computational graph

Question goes here

Answer goes here