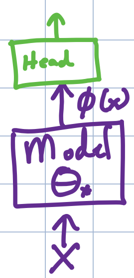
Today: Meta-learning
Forgetting
Generative Models: VAE

Announce: Fill out survey
Extra Credit for everyone (3%)
IF 75% of class does the survey

Recall Approaches To Adapt A Model to a new task:

0) IF something promptable, simply prompt it (Potentially using a prompt optimizer)

1) Use the pretrained model as an embedder / Feature extractor

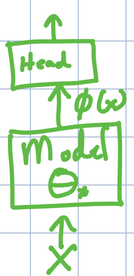


Called Linear Probing:
Train a new task-specific head for this task.
e.g. linear classifier, regression

Advantage: Easy to do. No need for data beyond task.

Disadvantage: Linear Probing might not work as well.

2) Full fine-tune: Train everything — use pretrained model as initial condition for a part of the model.



Advantage: Typically higher performance on new task.

Disadvantage: Can be far bigger of a training job.
Risk of overfitting.

Practical Tip: Don't just initialize a random head & then fine-tune.

Better: Initialize Head to 0.
Even Better: Use some data to train just the head First.

3) LoRAs or Soft-prompting combined with a new head...

Meta-learning: Making a model better at being fine-tuned for tasks
(Think categories 2&3 above)

What's a good baseline approach?

0) Do Nothing: Random Initialized
1) General Foundation Model
2) MAML: Model Agnostic Meta-Learning

What do we need? A) A collection of tasks from the family.
i.e. Training Data for these different tasks
+ Loss Function.

B) Approach to fine-tuning.
e.g. Use a LoRA and the SGD optimizer on training data.

c) Approach to evaluation
e.g. Eval performance on held-out set

Key Insight: In ML, default is train like you'll be tested.

Second Insight: Learning Process of SGD is like an RNN.

Let's be precise.—

A task i has Training Data D_i, Training Loss L_i, Test Loss $\tilde{L}_i$

Want to train to do well on all tasks from this family, including held-out tasks.

Note: tasks may or may not share output cardinalities.
If they do, or have nested/overlapping outputs, we can consider learning a good initialization for the output heads too.
If they don't, we can use zero for the initialization of task-specific heads.

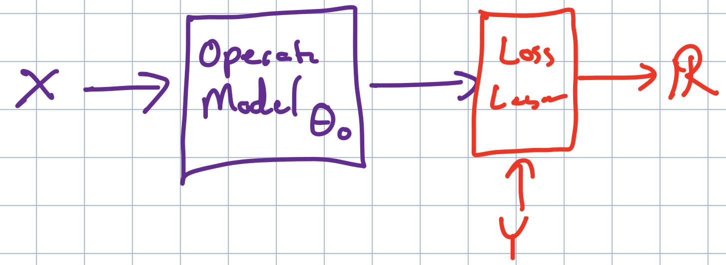
Assume we have a good initialization Θ_0. How would we use it?

1) Start Model at Θ_0 ← All learnable parameters in the model

2) Do gradient descent steps using D, L to get to Θ_{final}

3) Evaluate Θ_{final} using held-out $\tilde{D}$ and Test Loss $\tilde{L}$ to get loss

Fit this (1,2,3) into our standard form:

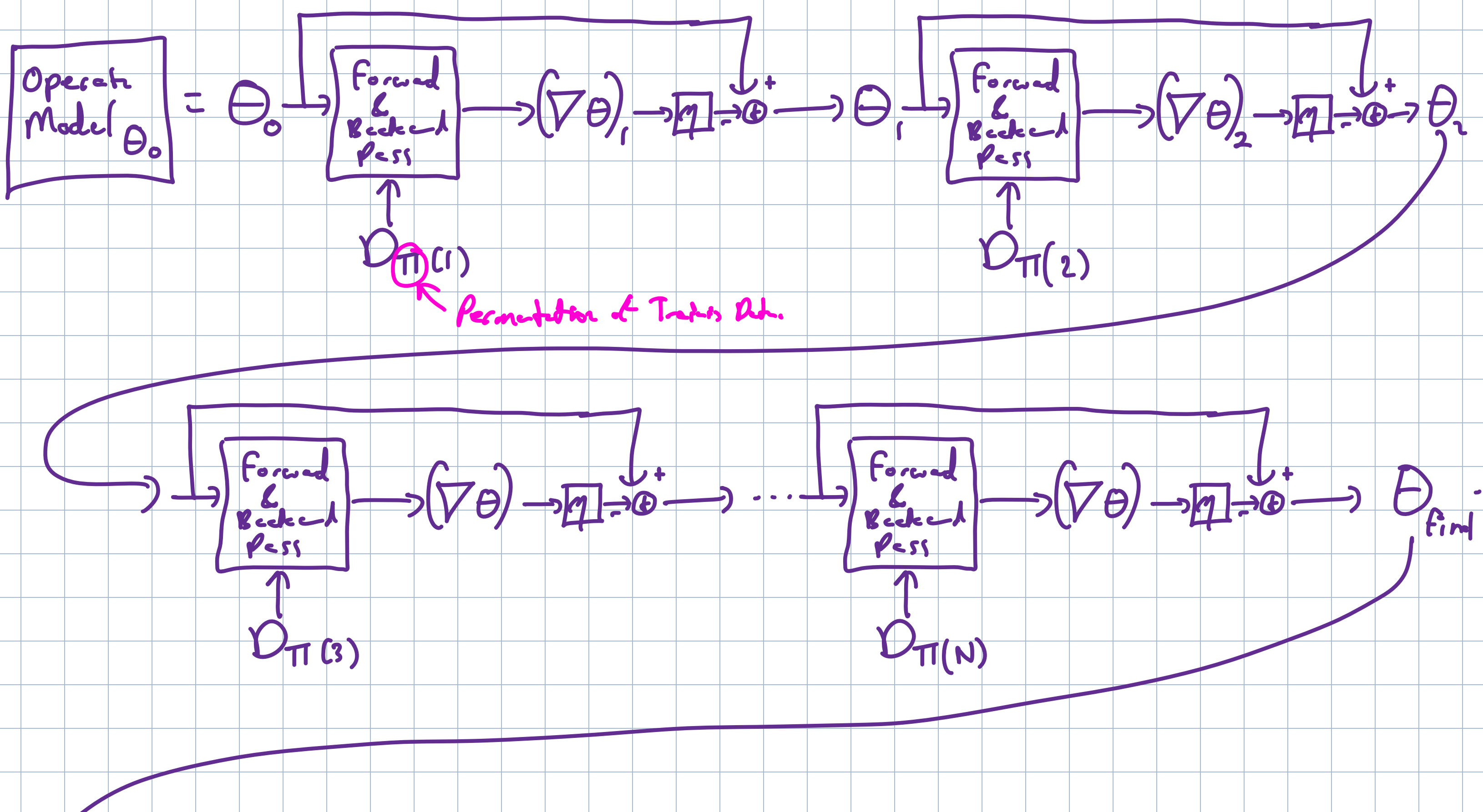


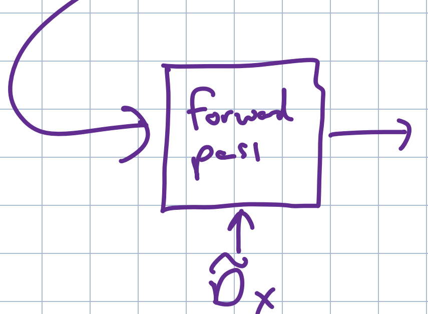
What is X?
Y?
operate model ?
Loss Layer?

Just Pattern Match: $X \longleftrightarrow (D, \tilde{D}_x)$

$Y \longleftrightarrow$ Label info in $\tilde{D}_Y$

Loss Layer $\longleftrightarrow \tilde{L}$





Note: To run backprop through this, need to store θ_i history, but can gradient checkpoint to redo forward/backward pass again during backprop.

MAML insight: That's just a deep model. Can backprop through it.

Gives a gradient $\nabla\Theta_0$ on initial condition.

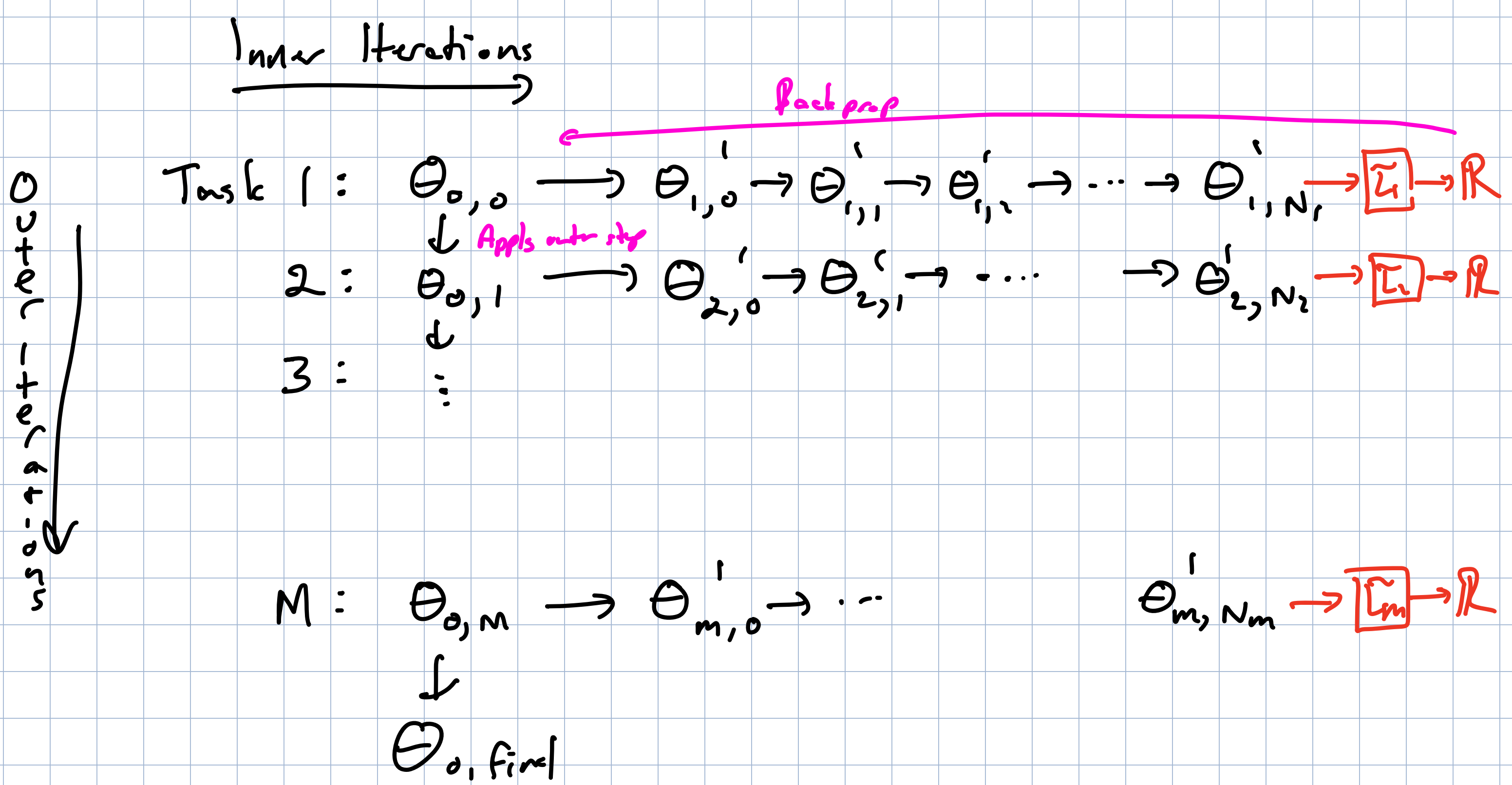
Can take a step in that direction for a better initial condition

Repeat for a new task (randomly drawn).....

Hope: This gives us a much better initial condition.

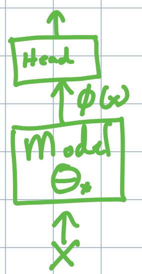
Schematic Picture: Outer-loop on tasks, learning-rate η_{outer}

Inner-loop on batches/examples, learning-rate η_{inner}



Two Variants:

1) Reptile : Avoid backprop through backprop
   Approximate $\nabla\theta \approx \theta_{final} - \theta.$
2) ANIL/ Meta Opt Net / R2D2 : Optimize for Linear Probes

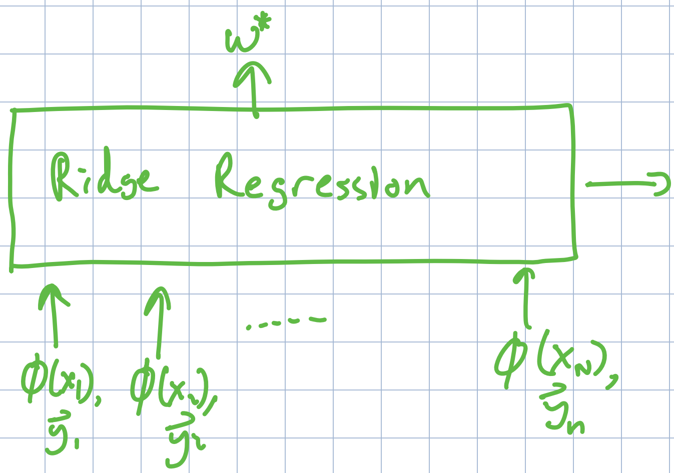


e.g. consider regression problems.

Have closed-form formulas for Head.
(Instead of Gradient updates...)
Take a gradient step through that to the parameters of feature extractor

Explicitly:

(Added during Office Hours)



Let $\Phi = \begin{bmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_N)^T \end{bmatrix}$

$Y = \begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_n^T \end{bmatrix}$

$$W^* = \underbrace{\left[\Phi^T \Phi + \lambda I\right]^{-1} \Phi^T Y}_{\text{Differentiable}}$$

$\Rightarrow$ Sends gradients to Φ and hence to $\phi(x_1), \dots \phi(x_N)$ during backprop.

Note: Convex Problem (even not in closed form) can send gradients via natural iterative solutions. e.g. A few Newton Steps

$W^{*} \phi(x_{test}) = \hat{y}_{test}^{T}$

output on test point

So loss $\tilde{L}$ on $\hat{y}_{test}$ relative to y_{test} sends gradients to both W^* and $\phi(x_{test})$

We end up with parallel gradients flowing back through weight-shared $\phi(x_{test}), \phi(x_1), ----, \phi(x_n)$

These can be used to get gradients on $\theta \rightarrow$ params defining ϕ

$\Rightarrow$ Updates on θ for meta-learning

Note: Can use gradient checkpointing ideas to avoid having to store all intermediate activations. Do forward pass twice and backward once.

Catastrophic Forgetting: When fine-tuning, model forgets how to do what it knew how to do.

Key Practical Solution: During fine-tuning, mix in some pretraining-style data.

Like 10%

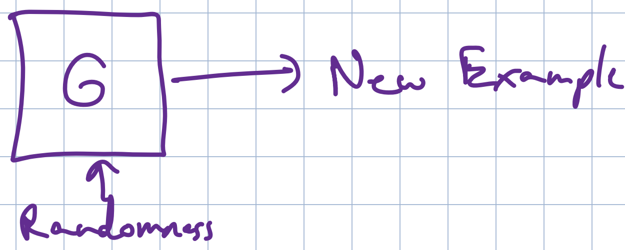
Added in Office Hours:

A) If the pretraining-task(s) had distinct heads, keep them during fine-tuning — they will (along with their losses) send gradients into the shared part of the model to prevent/reduce forgetting. These heads should update — they are not frozen.

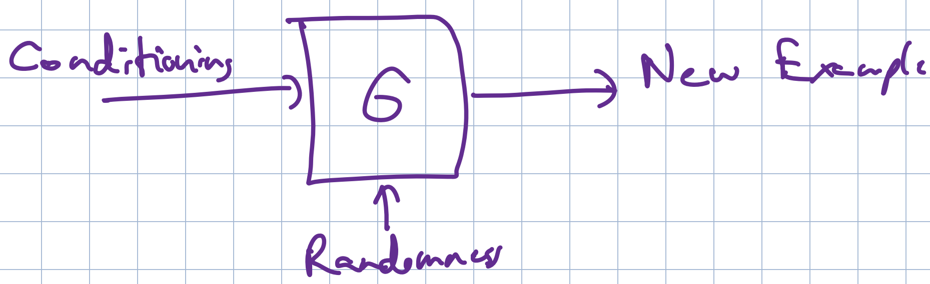
B) Forgetting can be thought of as a type of overfitting.

Generative Models

Unconditional

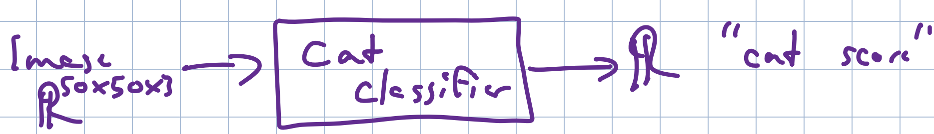


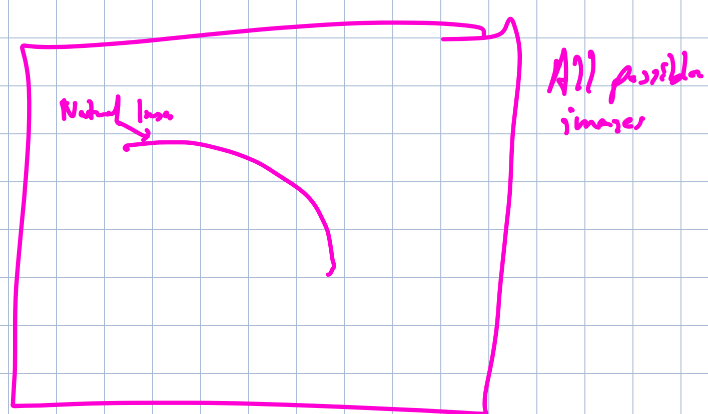
Conditional



Ideas that don't work

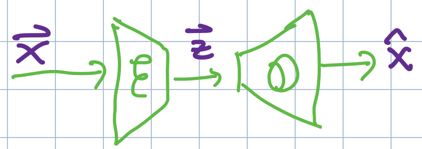
A) Use a classifier





B) Use an autoencoder

Core Ingredients: Labels are $\vec{x}_i$ itself.
Architecture has an encoder followed by a decoder
Bottleneck in the middle.



Traditional Perspective: Decoder is scaffolding.

Try using $\mathcal{D}$ to generate samples. — If $\vec{z}$ too small, get blurry junk
If $\vec{z}$ big, "hiss"

VAE Approach

3 key ingredients: 1) Make $\vec{z}$ random during training

2) Add a loss on distribution of $\vec{z}$

3) Make this work with SGD