Contents

Lectures

Contents

Lectures **September 21, 2020**

The Meta-Learning Problem & Black-Box Meta Learning

Table of Contents Local *Written by Brandon McKinzie*

Black-Box Adaptation. Want to learn a "black-box" function $f : \mathcal{D} \mapsto \phi$ that outputs a set of *parameters* ϕ for *another* [known] network g_ϕ, such that g_ϕ performs well on test data from the same distribution as $\mathcal{D}$. Stated another way, f is basically a procedure for setting the weights ϕ of your neural network g, given training data $\mathcal{D}$. Yes, f is performing a similar function as a standard optimizer like SGD, but what's new is that we are now representing f *itself as a neural network* $f = f_\theta$ with its own trainable parameters θ. Now that your head is hopefully in the right place, here is the more concise definition for black-box adaptation of task $\mathcal{T}_i$ from the lecture [1:01:00]

1. Train a NN to represent $\phi_i = f_\theta(\mathcal{D}_i^{tr})$.
2. Predict test points with $\boldsymbol{y}^{ts} = g_{\phi_i}(\boldsymbol{x}^{ts})$.

We can learn the parameters θ of our black-box function with standard supervised learning (*meta*):

$$\theta^* = \arg\max_\theta \sum_{\mathcal{T}_i} \sum_{x,y \sim \mathcal{D}_i^{test}} \log g_{\phi_i}(y \mid x) \tag{1}$$

$$= \max_\theta \sum_{\mathcal{T}_i} \mathcal{L}\left(f_\theta\left(\mathcal{D}_i^{tr}\right), \mathcal{D}_i^{test}\right) \tag{2}$$

So, my first question was: *how does backpropagation work here?*. Basically, for each test pair (x, y), evaluating $g_{\phi_i}(y \mid x)$ will (of course) look like some function of ϕ_i. So we apply the chain rule and end up with a bunch of gradients of the form $\frac{\partial \phi_{i,j}}{\partial \theta}$. Well each of those $\phi_{i,j}$ is really just a function of θ, since it's literally the output of f. So we just keep chain ruling along as usual. Don't overthink it.

Challenge: outputting the full set of neural network parameters ϕ does not seem scalable.
Idea: Don't need to output *all* parameters of NN, only sufficient statistics[1]. [1:14:00]

Black-Box Adaptation

Key idea: Train a NN to represent $\phi_i = f_\theta(\mathcal{D}_i^{tr})$. Repeat:

1. Sample $\mathscr{T}_i$
2. Sample disjoint $\mathcal{D}_i^{tr}, \mathcal{D}_i^{test}$ from $\mathcal{D}_i$
3. Optimize $\phi_i \leftarrow f_\theta(\mathcal{D}_i^{tr})$
4. Update θ using $\nabla_\theta \mathcal{L}(\phi_i, \mathcal{D}_i^{test})$

[1] Santoro et al. MANN, Mishra et al. SNAL

Lectures **September 23, 2020**

Optimization-Based Meta-Learning

Table of Contents Local *Written by Brandon McKinzie*

Multi-Task Learning

Solve multiple tasks $\mathscr{T}_1, \ldots, \mathscr{T}_T$ at once.

$$\min_\theta \sum_{i=1}^{T} \mathscr{L}_i \left(\theta, \mathcal{D}_i\right)$$

Transfer Learning

Solve target $\mathscr{T}_b$ after solving source $\mathscr{T}_a$ by transferring knowledge learned from $\mathscr{T}_a$.

Fine-Tuning

Given a pretrained set of parameters θ, update them on some new training dataset $\mathcal{D}^{tr}$ on a [potetntially] different task to obtained a final set of parameters ϕ[2]

$$\phi \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}\left(\theta, \mathcal{D}^{tr}\right)$$

Meta-Learning

Given data from $\mathscr{T}_1, \ldots, \mathscr{T}_n$, quickly solve new task $\mathscr{T}_{test}$.

Optimization-Based Adaptation. Instead of treating the inner (initial) learning task of obtaining parameters from some training dataset $\mathcal{D}_{tr}$ as a black-box neural network f_θ, we can model it as some kind of optimization process $\nabla_\theta \mathcal{L}$ explicitly . One example is **Model-Agnostic Meta-Learning**[3] which performs fine-tuning at *test-time*. [7:50]

$$\min_\theta \sum_{\text{task } i} \mathcal{L}\left(\theta - \alpha \nabla_\theta \mathcal{L}\left(\theta, \mathcal{D}_i^{tr}\right), \mathcal{D}_i^{ts}\right) \tag{3}$$

Key Idea: Over many tasks, learn θ that transfers [preferably with few examples] via fine-tuning.

- **Q**: Are the two loss functions above the same?
  - **A**: Yes. It appears that is actually a critical assumption/requirement.
- **Q**: Are we assuming that θ is pretrained/reasonably initialized? Or is this done "from scratch" for every test query? Seems extremely expensive
  - **A**: Yes, θ is reinitialized each time **[19:45]**.

[2] Seems like unnecessary cognitive overload to denote the updated θ with a different symbol ϕ but ok.

[3] Finn, Abbeel, Levine. Model-Agnostic Meta-Learning. ICML 2017.

Optimization-Based Adaptation

Key idea: acquire ϕ_i through optimization. Repeat:

1. Sample $\mathscr{T}_i$
2. Sample disjoint $\mathcal{D}_i^{tr}, \mathcal{D}_i^{test}$ from $\mathcal{D}_i$
3. Optimize $\phi_i \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{tr})$ [inner-loop]
4. Update θ using $\nabla_\theta \mathcal{L}(\phi_i, \mathcal{D}_i^{test})$ **[Meta-Objective]** [outer-loop]

Notice how the final step above results in 2nd-order derivatives!

Clarification on how MAML differs from transfer learning (TL) in terms of initializing θ: **[21:10]**

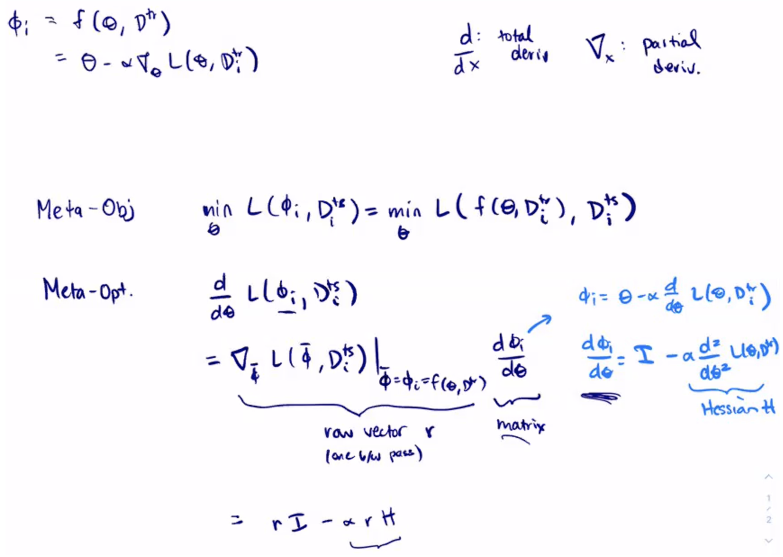
$$\textbf{[TL]} \qquad \theta \leftarrow \arg\min_\theta \sum_i \mathcal{L}(\theta, \mathcal{D}_i^{tr}) \tag{4}$$

$$\textbf{[MAML]} \qquad \theta \leftarrow \arg\min_\theta \sum_i \mathcal{L}\left(\phi_i, \mathcal{D}_i^{ts}\right) \tag{5}$$

$$= \arg\min_\theta \sum_i \mathcal{L}\left(\theta - \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i), \mathcal{D}_i^{ts}\right) \tag{6}$$

In words, TL initializes θ such that it performs well on the initial training [source] task(s), whereas MAML initializes θ such that some *fine-tuned* version of θ [over presumably a small number of steps] will do well on the task **[22:50]**.

- **Q**: How do we determine [in practice] the number of gradients steps & over what training examples to perform the inner gradient step(s) [step 3 of alg above] compared to the outer gradient step(s) [step 4 of alg above]?
  - **A**: **INCOMPLETE ANSWER** Prof Finn says the outer gradient step is a single step, whereas the inner gradient can be one or more steps. That still doesn't answer the question of which example(s) to use for both of them though.
- **Q**: Do we need to compute the full Hessian?
  - **A**: No, just the vector-hessian product $\boldsymbol{rH}$ in image below **[35:40]**.



$\phi_i = f(\theta, D^{tr})$
$= \theta - \alpha \nabla_\theta L(\theta, D_i^{tr})$

$\frac{d}{dx}$: total deriv $\quad \nabla_x$: partial deriv.

Meta-Obj $\quad \min_\theta L(\phi_i, D_i^{ts}) = \min_\theta L(f(\theta, D_i^{tr}), D_i^{ts})$

Meta-Opt. $\quad \frac{d}{d\theta} L(\underline{\phi_i}, D_i^{ts})$

$= \underbrace{\nabla_{\bar\phi} L(\bar\phi, D_i^{ts}) \Big|_{\bar\phi = \phi_i = f(\theta, D^{tr})}}_{\text{row vector } r \text{ (one b/w pass)}} \underbrace{\frac{d\phi_i}{d\theta}}_{\text{matrix}}$

$\phi_i = \theta - \alpha \frac{d}{d\theta} L(\theta, D_i^{tr})$

$\underline{\frac{d\phi_i}{d\theta}} = I - \alpha \underbrace{\frac{d^2}{d\theta^2} L(\theta, D^{tr})}_{\text{Hessian } H}$

$= rI - \alpha \underbrace{rH}$

- **Q**: Do we get higher-order derivatives with more inner gradient steps?
  - **A**: No, basically because the gradients are taken in succession – take one step on θ to get some θ', take the next on θ' to get some θ'' etc., as opposed to taking two gradients on θ [40:00].

Optimization vs Black-Box. [43:30]

$$
\begin{aligned}
\textbf{[BB]} \qquad y^{ts} &= f_{BB}(\mathcal{D}_i^{tr}, x^{ts}) && (7)\\
\textbf{[MAML]} \qquad y^{ts} &= f_{MAML}(\mathcal{D}_i^{tr}, x^{ts}) && (8)\\
&= f_{\phi_i}(x^{ts}) \quad \text{where } \phi_i = \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{tr}) && (9)
\end{aligned}
$$

*MAML can be viewed as a **computation graph**, with an embedded gradient operator.*

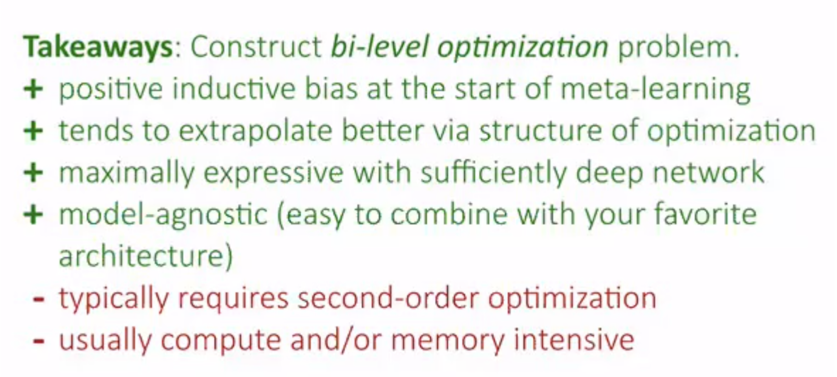
So, does embedding this optimization structure into our meta-learner come at a cost?. Answer: for a sufficiently deep network, MAML can approximate any function of $\mathcal{D}_i^{tr}, x^{ts}$, given the following **assumptions**[4]: [53:00]

- nonzero α
- loss function gradient does not lose information about the label (**what?**)
- Datapoints in $\mathcal{D}_i^{tr}$ are unique.

[4] Finn & Levine, ICLR 2018

Challenges [of optimization-based adaptation] and potential solutions for each: [55:20]

- Bi-level optimization can exhibit instabilities.
  - Automatically learn inner vector learning rate α.
  - Only optimize a subset of the params in the inner loop.
  - Decouple inner learning rate and the batch-norm statistics per-step.
  - Introduce context vars for increased expressive power.
- Backprop through many inner gradient steps is compute & memory intensive.
  - Crudely approx jacobian $\frac{d\phi_i}{d\theta}$ as identity matrix. "Things that look like ϕ_i look like θ."
  - Only optimize last layer of weights.
  - Derive meta-gradient using the implicit function theorem.
- How to choose architecture that's effective for inner gradient step?
  - Progressive NAS + MAML. Finds deep & narrow architectures that are quite different from standard supervised learning ones.



Takeaways: Construct ***bi-level optimization*** problem.

+ positive inductive bias at the start of meta-learning

+ tends to extrapolate better via structure of optimization

+ maximally expressive with sufficiently deep network

+ model-agnostic (easy to combine with your favorite architecture)

- typically requires second-order optimization

- usually compute and/or memory intensive

Lectures **September 28, 2020**

Automatic Differentiation

Table of Contents Local *Written by Brandon McKinzie*

Guest lecture from Matt Johnson (Google). Notation[5][6]:

$$
\begin{aligned}
f &: \mathbb{R}^n \mapsto \mathbb{R}^m && (13)\\
\partial f &: \mathbb{R}^n \mapsto (\mathbb{R}^n \mapsto \mathbb{R}^m) && (14)\\
\partial f(x) &: \mathbb{R}^n \mapsto \mathbb{R}^m && (15)\\
\partial f(x) &\in \mathbb{R}^{m \times n} && (16)\\
f(x+v) &= f(x) + \partial f(x)[v] + \mathcal{O}(||v||^2) && (17)\\
\langle \nabla f(x), v \rangle &= \partial f(x)[v] && (18)
\end{aligned}
$$

Review: Taylor Series

A **power series** is an infinite series of the form $\sum_{k=0}^{\infty} c_k (x-a)^k$ where the coefficients c_k and the **center** of the series a are constants. Recall that if a function f is differentiable at a point a, it can be approximated near a by its tangent line:

$$f(x) \approx f(a) + f'(a)(x-a) \tag{19}$$

Let's denote this by $p_1(x)$ – this is a first-order approximation of f at a, since $p_1(a) = f(a)$ and $p_1'(a) = f'(a)$. We can find successively higher-order approximations by adding the next term in the power series, which here would be $c_2(x-a)^2$, and solving for c based on the constraints that all derivatives (from order 0 to n – where n is 2 here) must be the same as the original function's. For the 2nd order approximation, we end up finding that $c_2 = \frac{1}{2} f''(a)$.
So, how good are these approximations? **Taylor's Theorem** states:

$$
\begin{aligned}
f(x) &= p_n(x) + R_n(x) && (20)\\
R_n(x) &= \frac{f^{(n+1)}(c)}{(n+1)!}(x-a)^{n+1} && (21)
\end{aligned}
$$

for some point c between x and a.

The function $\partial f(x)$ is a linearized version of f at the point x [28:43].

[5] Note that he uses brackets to mean "times." They could just as well be parentheses. He also prefers to think of the quantity $\partial f(x)$ as a linear map (i.e. a matrix) being applied (i.e. left-multiplied) to v **[30:00]**. You can also remember the bracket notation as the same interpretation as our linear algebra textbook for operators/maps.

[6] Reminder: If we define the taylor expansion to be centered at some fixed point x, and want to evaluate that expansion at $x + v$:

$$
\begin{aligned}
f(x+v) &\approx f(x) + \partial f(x)(x - (x+v)) && (10)\\
&= f(x) + \partial f(x) v && (11)\\
& && (12)
\end{aligned}
$$

Two Linear Maps of AD

[37:20]

- **Jacobian Vector Product (JVP)**/push-forward/forward-mode. The map $v \mapsto \partial f(x)[v]$. Note that if $v = \hat{e}_n$ (one-hot vector with $v_n = 1$ and 0 elsewhere), then this maps to the nth *column* of the Jacobian, $\partial f(x)_{:,n}$.

  Example: we're given $v = \frac{\partial x}{\partial \theta}$ for input scalar θ. You want to compute $\frac{\partial f(x)}{\partial \theta}$ for some function $f(x)$. We can use $\partial f(x)$ to *push forward* the perturbation information we were given to obtain it: $\frac{\partial f(x)}{\partial \theta} = \partial f(x)[v]$ [40:00].
- **Vector Jacobian Product (VJP)**/pull-back/reverse-mode. The map $w^T \mapsto \partial f(x)^T[w^T]$ where $w^T \in \mathbb{R}^m$. Just the transpose of JVP basically. Can use to obtain the Jacobian one *row* at a time.

  Example: we're given a row vector, $w^T = \frac{\partial \ell}{\partial y}$, representing how a scalar-valued loss ℓ would change given a perturbation in some intermediate vector y. We want to know how the loss will change given a perturbation in some earlier vector x, where $y = f(x)$. We can use $\partial f(x)^T$ to *pull-back* sensitivity information to do this: $u^T = \frac{\partial \ell}{\partial x} = \partial f(x)^T[w^T]$.

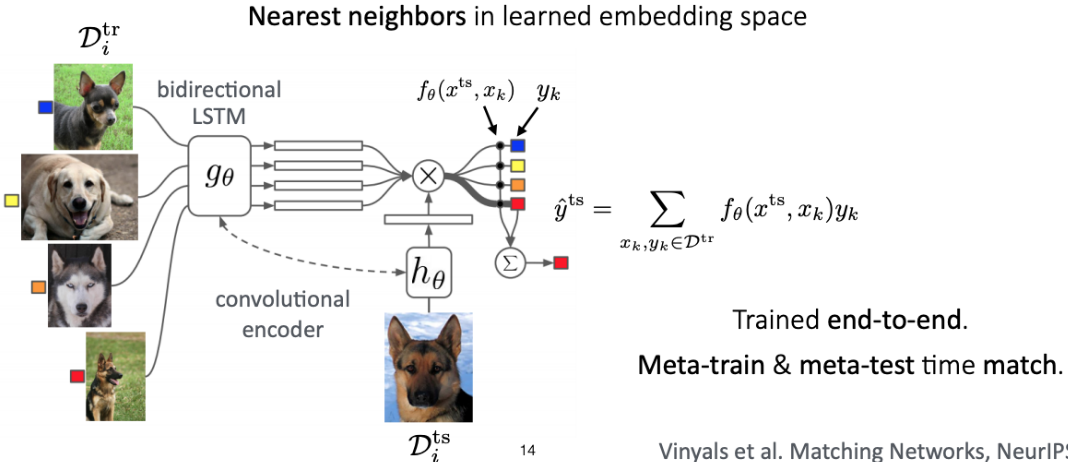
Lectures **September 30, 2020**

Non-Parametric Few-Shot Learning

Table of Contents Local *Written by Brandon McKinzie*

Non-Parametric Methods. Motivation: can we embed a learning procedure *without* a second-order optimization (like we saw in optimization-based meta-learning)? In low data regimes, **non-parametric** methods are simple and work well. Well, the inner loop where we do few-shot learning is a perfect low-data regime! We still want parametric methods for the outer loop (meta-training). **Can we use parametric meta-learners that produce effective non-parametric learners?** [6:40]

Say we want to perform a non-parametric technique like nearest neighbors in the inner loop to classify the meta-test example images. Well, how do we do the comparison between the test images and meta-training data? L2 distance in pixel space? **Idea**: train a [parametric] siamese network to act as a comparator function between two images, that returns a binary yes/no if the two input images are the same class[7]. We can use that in the inner loop as our comparator for KNN. However, it would be nice if we could match the meta-training and meta-test tasks [14:00] (instead of doing binary classification for former and N-way classification for latter). This is where **matching networks** come in, which perform NN in a learned embedding space.



Nearest neighbors in learned embedding space

$\mathcal{D}_i^{\mathrm{tr}}$
bidirectional LSTM
g_θ
$f_\theta(x^{\mathrm{ts}}, x_k)$
y_k
Σ
h_θ
convolutional encoder
$\mathcal{D}_i^{\mathrm{ts}}$

$$\hat{y}^{\mathrm{ts}} = \sum_{x_k, y_k \in \mathcal{D}^{\mathrm{tr}}} f_\theta(x^{\mathrm{ts}}, x_k) y_k$$

Trained **end-to-end.**

Meta-train & meta-test time **match.**

14

Vinyals et al. Matching Networks, NeurIP

[7] At meta-test time, we compare the test input $\boldsymbol{x}_{test}$ with each meta-train example, and predict the label of the meta-train image that had the highest probability of being the same class as the test image.

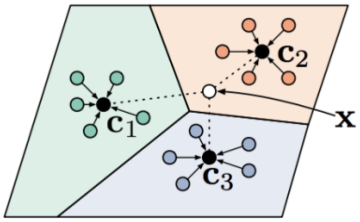
The operations from input to output in the figure above are as follows:

1. Feed the meta-training images through g_θ.
2. Feed the meta-test image through h_θ.
3. Compute something akin to a negative distance between the two aforementioned embeddings. For example, the paper defines the big $\otimes$ in the figure as softmax(cos(f, g)). Interpretation here of elem i would be "probability of being the same class as meta-training example $\boldsymbol{x}_i$".
4. Multiply each of the probabilities (we have num-meta-train-examples probabilities here) by their associated one-hot label vector.
5. Compute the sum over these to obtain the final output vector of size num-meta-train-examples, whose 'i'th output is equal to $f_\theta(\boldsymbol{x}^{ts}, \boldsymbol{x}_i)$.

Non-Parametric Meta-Learning with Matching Networks

1. Sample $\mathscr{T}_i$
2. Sample disjoint $\mathcal{D}_i^{tr}, \mathcal{D}_i^{test}$ from $\mathcal{D}_i$
3. Compute $\hat{y}^{ts} = \sum_{x_k, y_k \in \mathcal{D}^{tr}} f_\theta(x^{ts}, x_k) y_k$ [inner-loop]
4. Update θ using $\nabla_\theta \mathcal{L}(\hat{y}^{ts}, y^{ts})$ **[Meta-Objective]** [outer-loop]

Everything we've seen so far has been single-shot. What if we have more than 1 shot? Authors of matching network just treated each shot independently, and don't aggregate any information across shots. Can we aggregate class information in some way to create a *prototypical embedding*? This is what **prototypical networks** (Snell et al. 2017) do. [31:00]

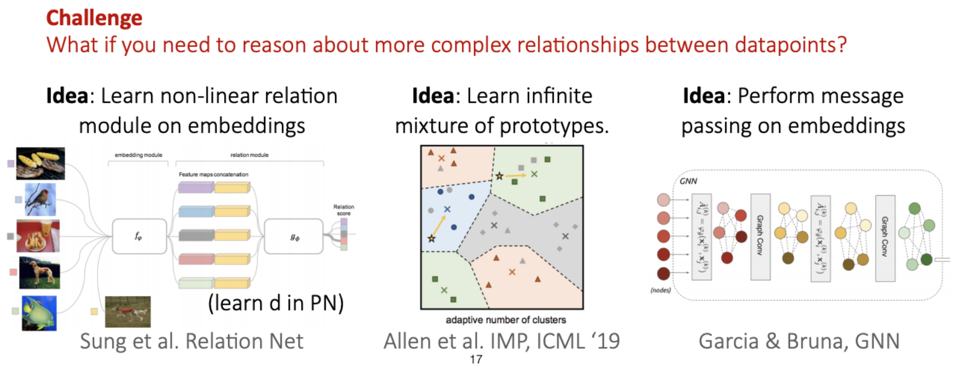


$$\mathbf{c}_n = \frac{1}{K} \sum_{(x,y) \in \mathcal{D}_i^{\text{tr}}} \mathbb{1}(y = n) f_\theta(x)$$

$$p_\theta(y = n | x) = \frac{\exp(-d\left(f_\theta(x), \mathbf{c}_n\right))}{\sum_{n'} \exp(d(f_\theta(x), \mathbf{c}_{n'}))}$$

d: Euclidean, or cosine distance

In other words, for each class, take the average embedding of the associated meta-train examples, and use that the same way we did with matching networks.



Challenge

What if you need to reason about more complex relationships between datapoints?

Idea: Learn non-linear relation module on embeddings

Idea: Learn infinite mixture of prototypes.

Idea: Perform message passing on embeddings

Sung et al. Relation Net

Allen et al. IMP, ICML ‘19

Garcia & Bruna, GNN

Quick review of the three ML approaches we’ve seen so far: [52:30]

[BB]
$$y^{ts} = f_\theta(\mathcal{D}_i^{tr}, x^{ts}) \tag{22}$$

[Opt]
$$y^{ts} = f_{MAML}(\mathcal{D}_i^{tr}, x^{ts}) = f_{\phi_i}(x^{ts}) \tag{23}$$
$$\text{where} \quad \phi_i = \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{tr}) \tag{24}$$

[Non-Param]
$$y^{ts} = f_{PN}(\mathcal{D}_i^{tr}, x^{ts}) = \text{softmax}\left(-d\left(f_\theta\left(x^{ts}, \boldsymbol{c}_n\right)\right)\right) \tag{25}$$
$$\text{where} \quad \boldsymbol{c}_n = \frac{1}{K} \sum_{x,y \in \mathcal{D}_i^{tr}} \mathbb{1}(y{=}n) f_\theta(x) \tag{26}$$

Algorithmic properties:

- **Expressive power**: ability for f to represent a range of learning procedures.
- **Consistency**: learned learning procedure will monotonically improve w/more data.
- **Uncertainty Awareness**: ability to reason about ambiguity during learning.

Black-Box	Optimization-Based	Non-Parametric
Complete expressive power	Only expressive for very deep models	expressive for most architectures
Not consistent	Consistent, reduces to GD	Consistent under certain condiations

Table 1: [1:00:00] TODO: revisit lecture here – lots of good points!

TODO: try proving everything above.

Bunch of example applications from [1:09:00] until the end of lecture:

- One-Shot Imitation Learning
- Low-Resource Molecular Property Prediction [1:11:00].
- Few-Shot Human Motion Prediction [1:12:00]
- Language Modeling [1:14:30].

Lectures **October 05, 2020**

Advanced Meta-Learning: Task Construction

Table of Contents Local *Written by Brandon McKinzie*

How should tasks be defined for good meta-learning performance?

Beginning of lecture explores what happens if the label assignments is consistent across shots in $\mathcal{D}_{tr}$. Model will basically learn to ignore the meta-training examples and just learn to classify the test inputs like a supervised learning problem. Note that this is fine if we don't plan on showing the model instances of unseen classes in the future. However, if we give it some new meta training set and ask it to predict test examples from it (on classes it wasn't trained on), it won't be able to adapt well at all.

Stated another way, if the tasks are **non-mutually exclusively**, meaning that a single function can solve all tasks, then the model can just learn that function and basically ignore the set of meta-training examples $\mathcal{D}_t r$ provided to it[8]. What we'd prefer, of course, is that it actually learn nothing about how to classify some canonical example, butt rather that it learns (i.e. encodes in its parameters θ) how to quickly acquire such knowledge from the meta-training examples in $\mathcal{D}_t r$. **[36:00]**

Meta-Learning without Memorization. Introduce **meta-regularization**. One option is to maximize the mutual information between the prediction $\hat{y}_{ts}$ and the meta-training examples $\mathcal{D}_{tr}$, given a test input x_{ts}:

$$\max I(\hat{y}_{ts}, \mathcal{D}_{tr} \mid x_{ts}) \tag{27}$$

$$\text{where } I(X, Y) \triangleq D_{KL}\left(P(X, Y) || P(X)P(Y)\right) \tag{28}$$

but defining this term in practice can be challenging. Instead, we can minimize a meta-training loss and the information in θ:

$$\mathcal{L}(\theta, \mathcal{D}_{meta-train}) + \beta D_{KL}\left(q(\theta; \theta_\mu, \theta_\sigma) || p(\theta)\right) \tag{29}$$

where q is a distribution over our weights and p is some prior distribution[9]. Amounts to injecting noise on our weights with variance θ_σ. Places precedence on using info from $\mathcal{D}_{tr}$ over storing info in θ. Quickly shows proof about whether meta-regularization leads to better generalization at **[53:30]**.

[8] It will just classify the test inputs based on the test inputs (aka regular supervised learning tbh).

[9] Another name for the KL term is **Bayes by Backprop.**

- **Q**: How do you "minimize information in θ" without basically saying "don't learn anything"? I think she meant to say more like "minimize information *about the tasks themselves* in θ."
  - **A**:
- **Q**: How are q and p actually defined? Does it matter?
  - **A**:

Meta-Learning without Tasks. What if we only have unlabeled data? Can we have an algorithm *propose* tasks such that it performs well in a few-shot setting at meta-test time? Want the proposed tasks to have some properties: [55:00]

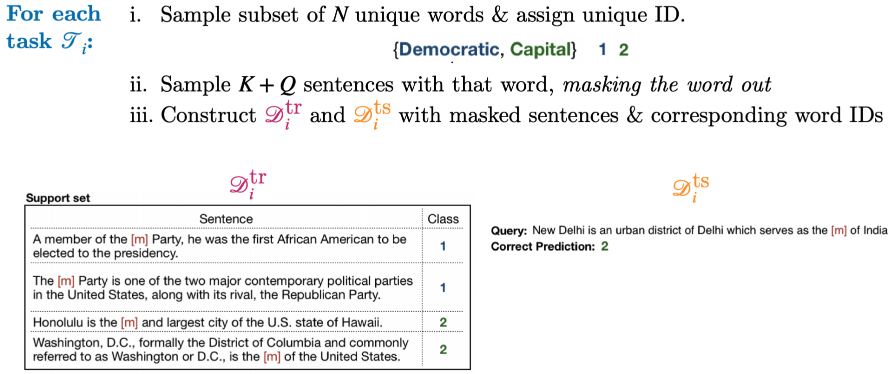
- **Diverse**. More likely to cover future test tasks.
- **Structured**. If tasks are *too* diverse, meta-learning may not be able to pick up on the underlying structure(s) useful for good performance.

Unlabeled Images[10]. [1:02:00]

1. Get image embeddings from some unsupervised algorithm.
2. Run K-means clustering in latent space.
3. Propose cluster discrimination tasks.
4. Run meta-learning (e.g. MAML) on the proposed labeled dataset(s).

We can also exploit **domain knowledge** when constructing tasks (e.g. translation invariance in images). So basically data augmentation, where the augmented data is stored in $\mathcal{D}^{ts}$.

Unlabeled Text. One option is formulating it as a **language modeling** problem (e.g. GPT-3). Finn mentions that this is both harder to (a) combine with optimization-based meta-learning[11], and (b) apply to classification tasks (since model outputs are raw text, not class labels). Another option is to construct tasks by *masking out words*, and defining the tasks as classifying the masked out word[12]. [1:15:30]



For each task $\mathscr{T}_i$:

i. Sample subset of N unique words & assign unique ID.

{Democratic, Capital} 1 2

ii. Sample $K+Q$ sentences with that word, *masking the word out*

iii. Construct $\mathscr{D}_i^{\mathrm{tr}}$ and $\mathscr{D}_i^{\mathrm{ts}}$ with masked sentences & corresponding word IDs

$\mathscr{D}_i^{\mathrm{tr}}$

Support set

Sentence	Class
A member of the [m] Party, he was the first African American to be elected to the presidency.	1
The [m] Party is one of the two major contemporary political parties in the United States, along with its rival, the Republican Party.	1
Honolulu is the [m] and largest city of the U.S. state of Hawaii.	2
Washington, D.C., formally the District of Columbia and commonly referred to as Washington or D.C., is the [m] of the United States.	2

$\mathscr{D}_i^{\mathrm{ts}}$

Query: New Delhi is an urban district of Delhi which serves as the [m] of India
Correct Prediction: 2

[10] Hsu, Levine, Finn. Unsupervised Learning via Meta-Learning. ICLR'19

[11] Why? Perhaps it's hard to extract the labels from the shots (context)? Might be worth reflecting on this more...

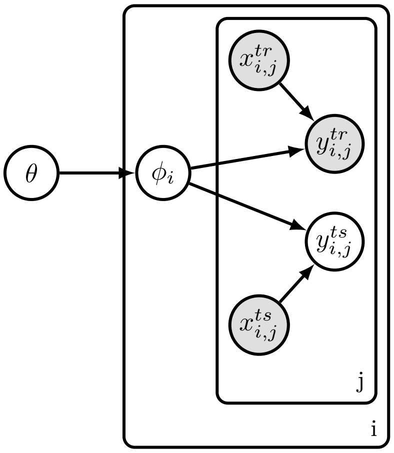
[12] Basically the text analog of what we did in HW1 for images

Lectures **October 07, 2020**

Bayesian Meta-Learning

Table of Contents Local *Written by Brandon McKinzie*

Begins with graphical model perspective of meta-learning.



$$\phi_a \perp \phi_b \mid \theta \quad \Longrightarrow \quad \mathcal{H}\left(p(\phi_i \mid \theta)\right) < \mathcal{H}\left(p(\phi_i)\right) \tag{30}$$

- **Q**: If we can identify θ, when should learning [the task-specific params] ϕ_i be faster than learning from scratch (i.e. learning from random ϕ_i and no notion of θ, instead of having θ inform the starting point of ϕ_i)
  - **A**: When θ contains information that is *not* present in $\mathcal{D}_i^{tr}$ or can't be learned directly from $\mathcal{D}_i^{tr}$ – yet is somehow useful prior knowledge for adapting to the task.
- **Q**: What if $\mathcal{H}\left(p(\phi_i \mid \theta)\right) = 0 \forall i$?
  - **A**: Then the true ϕ_i responsible for the data-generating distribution is a deterministic function of θ *for all tasks* $\mathcal{T}_i$. As a consequence, it means we can completely ignore $\mathcal{D}_i^{tr}$ $\forall i$.
- **Q**: What if you meta-learn without a lot of tasks?
  - **A**: Meta-overfitting to the family of training functions.

Bayesian Deep Learning Toolbox. Goal: represent distributions with neural networks (aka CS236): [38:00]

- Latent variable models + variational inference.
- Bayesian ensembles.
- Bayesian neural networks.
- Normalizing flows.
- Energy-based models & GANs.

Lectures **October 12, 2020**

Reinforcement Learning Primer

Table of Contents Local *Written by Brandon McKinzie*

RL Terminology Review

- **Utility** U is discounted $\sum R(s,a,s')$ on the path (this is a random quantity).
- **Value** $V_\pi(s)$ of policy π from state s is the $\mathbb{E}\left[U\right]$ received by following policy π.
- **Q-value** $Q_\pi(s,a)$ is $\mathbb{E}\left[U\right]$ of taking action a from state s, and then following policy π.

We can compute these quantities via:

$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd(s)} \\ Q_\pi(s, \pi(s)) & \text{otherwise} \end{cases} \tag{31}$$

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_\pi(s') \right] \tag{32}$$

Multi-Task RL. We can cast definition of an RL task into a multi-task setting by appending a **task identifier** z_i to the state vector $\boldsymbol{s} \to (\bar{\boldsymbol{s}}, z_i)$ (where $\bar{\boldsymbol{s}}$ is how we refer to the original state vector/formulation henceforth). Formally, the original definition of an RL task $\mathscr{T}_i$, followed by the reformulation to the multi-task setting, is shown below. [30:30]

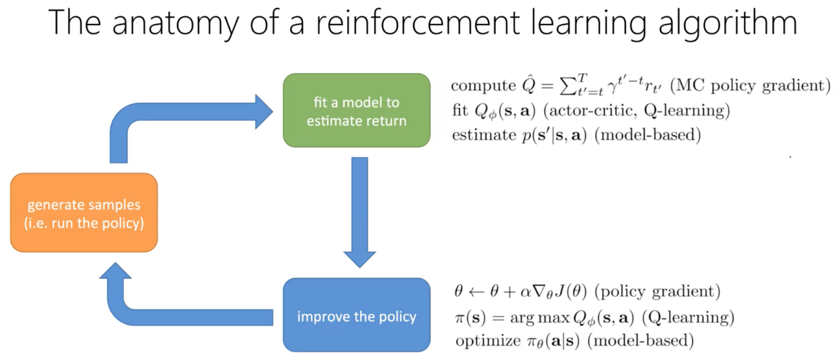
$$\text{[Single-Task RL]} \quad \mathscr{T}_i \triangleq \{\mathcal{S}_i, \mathcal{A}_i, p_i(\boldsymbol{s}_1), p(\boldsymbol{s'} \mid \boldsymbol{s}, \boldsymbol{a}), r(\boldsymbol{s}, \boldsymbol{a})\} \tag{33}$$

$$\text{[Multi-Task RL]} \quad \{\mathscr{T}_i\} = \left\{ \cup \mathcal{S}_i, \cup \mathcal{A}_i \frac{1}{N} \sum_i p_i(\boldsymbol{s}_1), p(\boldsymbol{s'} \mid \boldsymbol{s}, \boldsymbol{a}), r(\boldsymbol{s}, \boldsymbol{a}) \right\} \tag{34}$$

goal-conditioned RL is when there is some desired goal state that we want our model to reach.

- **Q**: Why is the multi-task initial state distribution an average over the initial-state distributions of each task? Shouldn't it be more like a renormalized sum (with new denominator over the union of states)?
  - **A**:

Policy Gradients & Multi-Task Counterparts. [35:20]



The anatomy of a reinforcement learning algorithm

generate samples (i.e. run the policy)

fit a model to estimate return

compute $\hat{Q} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ (MC policy gradient)
fit $Q_\phi(\mathbf{s}, \mathbf{a})$ (actor-critic, Q-learning)
estimate $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ (model-based)

improve the policy

$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$ (policy gradient)
$\pi(\mathbf{s}) = \arg\max Q_\phi(\mathbf{s}, \mathbf{a})$ (Q-learning)
optimize $\pi_\theta(\mathbf{a}|\mathbf{s})$ (model-based)

- **On-Policy**:
  - Data comes from the current policy.
  - Compatible with all RL algorithms.
  - Can't reuse data from previous policies.
- **Off-Policy**:
  - Data comes from any policy.
  - Works with specific RL algorithms.
  - Much more sample efficient, can reuse old data.

Let $\pi_\theta(\tau)$ denote the joint probability of some full episode (trajectory) τ, factorized as

$$\pi_\theta(\tau) \triangleq \pi_\theta(\boldsymbol{s}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{s}_T, \boldsymbol{a}_T) \tag{35}$$

$$= p(\boldsymbol{s}_1) \prod_{t=1}^{T} \pi_\theta(\boldsymbol{a}_t \mid \boldsymbol{s}_t) p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t) \tag{36}$$

[40:00]

where each $\pi_\theta(\boldsymbol{a}_t \mid \boldsymbol{s}_t)$ is typically parameterized as a neural network from observations to actions (predictions), and $p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$ is the "world's" distribution, sampled from the environment. **Policy gradient** can be formulated as follows:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \triangleq \arg\max_\theta J(\theta) \tag{37}$$

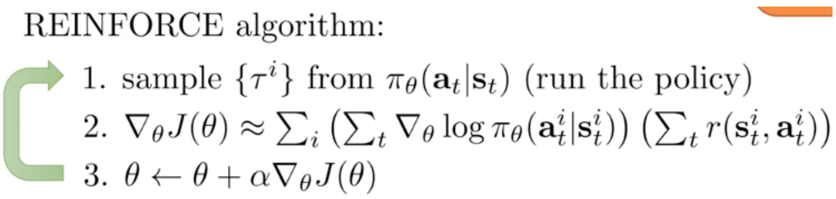
$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) \mathrm{d}\tau \tag{38}$$

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right] \tag{39}$$

[49:30]

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t \mid \boldsymbol{a}_t) \right) \left(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right) \right] \tag{40}$$

As usual, in practice we typically take MC samples from π_θ to approximate this expectation over trajectories τ.



REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Pros/cons of policy gradients: [56:00]

- ✓ Simple
- ✓ Easy to combine w/existing multi-task & meta-learning algs
- ✗ High-variance gradient
- ✗ Requires on-policy data

Q-Learning. Begins w/review of value-based RL. [58:00]

Fitted Q-iteration Algorithm [1:04:00]

Algorithm for learning a parameterized Q function $Q_\phi : (\boldsymbol{s}, \boldsymbol{a}) \mapsto \mathbb{R}$ (output is Q-value which I assume will always be a real-valued scalar).

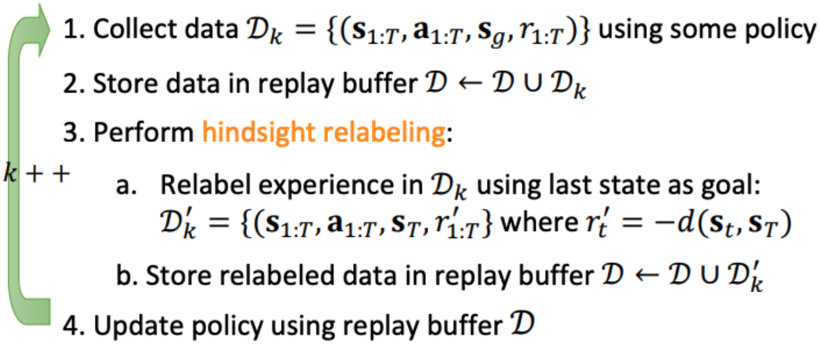
1. Collect dataset $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$ using some (not necessarily the current) policy.
2. Set targets $\leftarrow_i r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \gamma \max_{\boldsymbol{a}'_i} Q_\phi(\boldsymbol{s}'_i, \boldsymbol{a}'_i)$. These represent labels/ground-truth for the expected [cumulative] reward of taking action $\boldsymbol{a}_i$ from state $\boldsymbol{s}_i$.
3. Set $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sigma_i ||Q_\phi(\boldsymbol{s}_i, \boldsymbol{a}_i) - \boldsymbol{y}_i||^2$.

Result: get a policy $\pi(\boldsymbol{a} \mid \boldsymbol{s})$ via $\arg\max_a Q_\phi(\boldsymbol{s}, \boldsymbol{a})$.

Pros/cons of Q-learning:

- ✓ More sample efficient than on-policy methods.
- ✓ Can incorporate off-policy data.
- ✓ Can update policy even w/o seeing the reward.
- ✓ Relatively easy to parallelize.
- ✗ Harder to apply standard meta-learning algs (DP alg).
- ✗ Lots of “tricks” to make it work.
- ✗ Potentially could be harder to learn than just a policy.

Multi-Task Q-Learning. One benefit of the multi-task setting is that we can perform **hindsight relabeling**: reusing our experiences from previous tasks, and just replacing the rewards with the current task rewards. For example, if we are trying to learn how to pass well in hockey, but we just happen to shoot a really good goal, we can reuse that experience when explicitly learning the "goal shooting" task and relabel with the associated reward for shooting that goal. Furthermore, if our multi-task setup includes tasks that are essentially opposites of one another (e.g. closing a drawer vs opening a drawer), we can use the *successes* of one task as example *failures* for the other task (and vice-versa). [1:12:30]



1. Collect data $\mathcal{D}_k = \{(\mathbf{s}_{1:T}, \mathbf{a}_{1:T}, \mathbf{s}_g, r_{1:T})\}$ using some policy

2. Store data in replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_k$

3. Perform hindsight relabeling:

    a. Relabel experience in $\mathcal{D}_k$ using last state as goal: $\mathcal{D}'_k = \{(\mathbf{s}_{1:T}, \mathbf{a}_{1:T}, \mathbf{s}_T, r'_{1:T}\}$ where $r'_t = -d(\mathbf{s}_t, \mathbf{s}_T)$

    b. Store relabeled data in replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}'_k$

4. Update policy using replay buffer $\mathcal{D}$

Lectures **October 19, 2020**

Model-Based Reinforcement Learning

Table of Contents Local *Written by Brandon McKinzie*

Model-Based RL. Learn model of the environment dynamics $p(s' \mid s, a)$. Today we'll look at the case where the only difference between all of our RL tasks is the reward function. For example, a personal robot may do laundry, cook, etc. (same environment dynamics for all). [16:20]

Approach 1: Planning

Optimize over actions using model

1. Run some policy (e.g. random policy) to collect data $\mathcal{D} = \{(s, a, s')_i\}$.
2. Learn model $f_\phi(s, a)$ to minimize $\sum_i ||f_\phi(s_i, a_i) - s'_i||^2$.
3. Choose actions via one of the common sub-approaches:
   - **Gradient-based**: Backprop through $f_\phi(s, a)$ (or pass gradients to $\pi_\theta(a \mid s)$.)
   - **Gradient-free**: iteratively sample action sequences, run through $f_\phi(s, a)$.

Possible failures of planning. Action optimization will exploit imprecisions in model (compounding errors). Can correct for this by refitting model using new data. Specifically, can add a 4th step the algorithm above where append new tuples (s, a, s') to $\mathcal{D}$ that we didn't perform well on. [29:18]

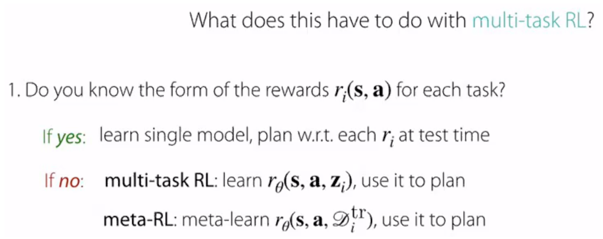
- **Q**: The model above, $f_\phi(s, a)$ seemingly is supposed to approximate the density $p(s' \mid s, a)$, so why do some diagrams here look like they imply it predicts the reward $r(s, a)$?
  - **A**:
- **Q**: How is the policy π_θ defined here? Is it an entirely different model?
  - **A**:
- **Q**: What's the distinction bw steps 2 and 3? It seems to imply that "learning the model" is not the same as backpropagating through it.
  - **A**: Yeah, so the second step is learning the global model of environment dynamics $p(s' \mid s, a)$ that is *task independent*. Then, although, ~~she doesn't explicitly clarify this~~ (she does around [35:00]), it seems assumed that step 3 is for learning to predict actions that maximize the task-specific reward function $r(s, a)$.

Approach 2: Model-Predictive Control (MPC) [31:30]

"Plan & replan using model". Literally the same as approach 1 but we basically just take one action at a time (instead of predicting & executing a full action sequence).

1. Run base policy $\pi_0(a_t \mid s_t)$ (e.g. random policy) to collect data $\mathcal{D} = \{(s, a, s')_i\}$.
2. Learn model $f_\phi(s, a)$ to minimize $\sum_i ||f_\phi(s_i, a_i) - s'_i||^2$.
3. Use model $f_\phi(s, a)$ to optimize action sequence a_t, a_{t+1}, a_{t+N}.
   (a) Execute the first planned action a_t, observing resulting state s'.
   (b) Append (s, a, s') to $\mathcal{D}$.

- compute intensive



What does this have to do with multi-task RL?

1. Do you know the form of the rewards $r_i(\mathbf{s}, \mathbf{a})$ for each task?

If *yes*: learn single model, plan w.r.t. each r_i at test time

If *no*: **multi-task RL**: learn $r_\theta(\mathbf{s}, \mathbf{a}, \mathbf{z}_i)$, use it to plan

meta-RL: meta-learn $r_\theta(\mathbf{s}, \mathbf{a}, \mathscr{D}_i^{\mathrm{tr}})$, use it to plan

Rest of lecture, starting from around **[45:00]** is about RL with high-dimensional observations (e.g images).

Lectures **October 21, 2020**

Meta-RL: Adaptable models and policies

Table of Contents Local *Written by Brandon McKinzie*

Met-RL Learning Problem. Given small amount of experience from task, can we learn a policy that does well on this task. For example, the meta-RL *maze navigation* problem. Each task is navigating a different maze. Give some small amount of experience in a new maze, can we quickly learn to solve it?

In meta-RL, common to use RNNs since there is a natural temporal structure to RL episodes. Input is (current state, prev reward)[13], and output is predicting the next action. Also the hidden state is maintained across episodes for a given task (bc we want to be able perform well across multiple episodes). An episode here is some full sequence of T (s, a, r) triplets.

Black-box meta-RL [26:40]

Repeat:

1. Sample task $\mathscr{T}_i$
2. Roll-out (execute) policy $\pi(a \mid s\mathcal{D}_{tr})$ for N episodes (under dynamics $p_i(s' \mid s, a)$ and reward $r_i(s, a)$.)
3. Store sequence in replace buffer for task $\mathscr{T}_i$
4. Update policy to maximize discounted return for all tasks.

Brief mention of RL^2 architecture (and others) at [33:00].

[13] This is a notable difference bw what you did with a recurrent policy.

Lectures **October 26, 2020**

Meta-RL: Learning to Explore

Table of Contents Local *Written by Brandon McKinzie*

Learning to Explore. Can we learn exploration strategies based on experience from other tasks in that domain?

Posterior sampling slide/explanation around **[18:30]** (PEARL).

Decoupled Exploration & Exploitation (Focus of HW4). Prev strategy tried to reconstruct states/rewards via training model $f(s', r \mid s, a, \mathcal{D}_{tr})$ & collecting $\mathcal{D}_{tr}$ so that model is accurate. Rather trying to reconstruct states/rewards, how might we explore *only* information relevant for solving the task? We can pass in a feature vector μ describing various features of the task to a model learning an *information bottleneck* representation z containing the "core task representation" (and outputs an execution policy). We learn to explore by recovering that information. At meta-test time, run exploration episode to learn the execution policy [**DREAM**] **[50:00]**. **[41:40]**

End-to-End Opt of Exploration Strategies

Homework

Contents

Homework **October 25, 2020**

Goal Conditioned RL and HER (HW3)

Table of Contents Local *Written by Brandon McKinzie*

Environment 1: Bit Flipping Environment

- ***State:*** *binary vector w/length n.*
- ***Reward:*** *$r(s, a)$ is -1 when performing a from s doesn't result in the goal vector/state, and 0 if it does match.*

In the bit-flipping environment, the state is a binary vector with length n. The goal is to reach a known goal vector, which is also a binary vector with length n. At each step, we can flip a single value in the vector (changing a 0 to 1 or a 1 to 0). This environment can very easily be solved without reinforcement learning, but we will use a DQN to understand how adding HER can improve performance.

The bit flipping environment is an example of an environment with sparse rewards. At each step, we receive a reward of -1 when the goal and state vector do not match and a reward of 0 when they do. With a larger vector size, we receive fewer non-negative rewards. Adding HER helps us train in an environment with sparse rewards (more details later).

Implementing Goal-Conditioned RL on Bit Flipping (Problem 1).

- Model: DQN which maps an input bit vector to `num_bits` logits. The ith output element is the predicted Q value for taking action i from the input state. We can take the argmax to obtain the next action.
- Buffer: container of (state, action, reward, next state) tuples. Supports sampling batches from the buffer with batch size `sampleSize`.

flipBits

For each iteration of each epoch:

1. Re-init state and goal state randomly from environment.
2. (solveEnvironmentNoGoal) Have the model flip the state numBits times sequentially.
3. (updateReplayBuffer): add all the experiences from the previous step to the replay buffer.
4. Repeat for `optSteps`:
    (a) Sample a batch of experiences from the replay buffer.
    (b) Run the targetModel on the next state s' to obtain logits over a'.
    (c) Compute target reward $r + \gamma \max_{a'} Q_{targ}(s', a')$.
    (d) Compute $Q_{val} = \sum_b Q_b^{model}(s, a^{taken})$ (sum of model-predicted Q values for the actions it took).
    (e) Compute MSE loss on Q_{val} and targetReward.

- updateTarget(model, targetModel, tau)
    1. Compute $w_i^{targ} \leftarrow \tau w_i^{targ} + (1 - \tau) w_i^{mod}$

- **Q**: Why are there two models? What do they each represent (from lecture)?
  - **A**: The target model is used for setting the targets from the second step of the Fitted Q-iteration algorithm from lecture.
- **Q**: Why does the target model have different weights than the main model? The Q-iteration algorithm from lecture made no such distinction.
  - **A**:
- **Q**: Why does the solve environment function iterate for numBits steps? Coincidence?
  - **A**:
- **Q**: Why is the clip min $-1/(1-\gamma)$?
  - **A**:

Homework **November 08, 2020**

Exploration in Meta-RL (HW4)

Table of Contents Local *Written by Brandon McKinzie*

Jargon:

- **Episode**: sequence of **experiences** (s, a, r, s')
- **Trial**: one or more episodes (default is 2 for this homework).
- **Agent**: synonymous with **policy** (in the code at least).
  - ~~Confusingly, however, the agent object is an instance of `DQNAgent`, which is somehow distinct from a class below it called `DQNPolicy`~~. A DQNAgent is composed of (re: has as an attrib) a DQNPolicy.
- Environments:
  - Exploration:
  - Instruction:
  - Multi-episode:

CityGridEnv

- ***State:***
  - *agent's (x, y) position*
  - *one-hot indicator of the object at the agent's position (none, bus)*
  - *one-hot goal **instruction** i that corresponds to one of four possible goal locations*
- ***Actions**: move up/down/right/left, and ride bus.*
- ***Episode**: 20 time steps. Agent receives -1 each time step until it reaches the goal, at which point it receives reward +1.*
- ***Task distribution** $p(\mu)$ corresponds to different bus arrangements.*

Problem 1: Evaluating End-to-End Meta-RL.

- **Environment**: CityGridEnv ("vanilla")
- **agent**: DQNAgent
  - **policy** (aka agent._dqn): RecurrentDQNPolicy
  - Underlying network/guts (self._Q) is DuelingNetwork : $(s, ?) \mapsto$ (Qval, hidden state).
    - hidden state is unused by act(...) (**??**)
    - state embedder is ultimately instantiated by the embedder factory in DQNPolicy.from_config. Determined by config.embedder.type. For problem 1, this is "recurrent", which makes the state embedder an instance of *RecurrentStateEmbedder.*

For each **episode**, do:

1. Generate the exploration and multi-episode environment.

   (a) Create the **exploration environment** (create_env). In the end, for the vanilla CityGridEnv, this just returns an instance of CityGridEnv using `index` as the random seed.
   (b) Create the **instruction environment**.
   (c) Create the **multi-episode environment**.

2. Run the episode using multi-episode env and agent. Loop until `done`:
   (a) Run `policy.act` to get action and next state.
   (b) Execute the action via `env.step`.

Problem 2: DREAM

DREAM consists of three main components:

1. Learned task encoder $F_\psi(z \mid \mu)$, where μ is task ID.
2. Execution policy π_θ^{exec}.
3. Exploration policy.

F_ψ and π_θ^{exec} are trained jointly via:

$$\max_{\psi,\phi} \mathbb{E}_{\mu\sim p(\mu), z\sim F_\psi(z|\mu), i\sim p_\mu(i)} \left[V^{\pi_\theta^{exec}}(i, z; \mu) \right] - \lambda I(z; \mu) \tag{41}$$