Homework 2

Deadline: Monday, Feb. 11, at 11:59pm.

Submission: You must submit two files through MarkUs[1]:

- A PDF of your solutions. You can produce the file however you like (e.g. LaTeX, Microsoft Word, scanner), as long as it is readable.
- Your completed `maml.py`

Late Submission: MarkUs will remain open until 3 days after the deadline, after which no late submissions will be accepted. The late penalty is 10% per day, rounded up.

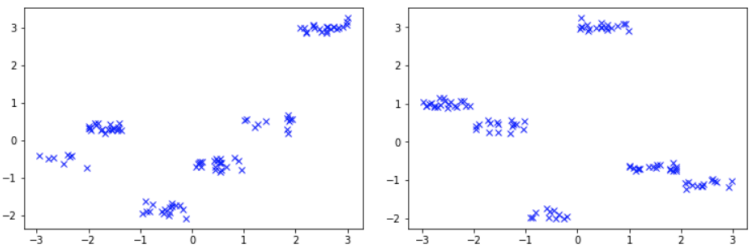
Weekly homeworks are individual work. See the Course Information handout[2] for detailed policies.

1. **MAML. [5pts]** This question is meant to introduce a cool application of automatic differentiation. Much like gradient-based hyperparameter optimization (last two slides of Lecture 6), it involves treating the gradient descent learning procedure itself as a computation graph, and differentiating through it. While the algorithm would be a major pain to implement by hand, it only requires a few extra lines of Autograd code compared to ordinary neural net training.

   Suppose you want to train an agent that learns to perform many different but related tasks, such as having a robot arm pick up a variety of objects. The agent, through gaining experience with many such tasks, ought to be able to improve the rate at which it can learn similar tasks. This kind of learning is known as **learning to learn**, or **meta-learning**.

   This question concerns a meta-learning algorithm called **Model-Agnostic Meta-Learning (MAML, pronounced "mammal")**.[3] The idea is that if you choose a good enough set of initial weights for the network, it should be possible to learn a new task in only a few steps of gradient descent. Hence, MAML trains a single, task-generic set of weights, with the meta-objective defined as the loss on any particular task after K steps of gradient descent. (K is a small number, such as 5.) The term "model-agnostic" is because MAML assumes pretty much nothing about the model, other than that it's trainable by gradient escent.

   MAML was originally formulated in the more complex setting of reinforcement learning, but we will consider the setting of simple univariate regression problems. We will sample random univariate regression problems, where the inputs are sampled uniformly from the interval $[-3, 3]$, and the functions are sampled as random piecewise constant functions with breaks at the integers. For instance,



[1] `https://markus.teach.cs.toronto.edu/csc421-2019-01`

[2] `http://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/syllabus.pdf`

[3] You're welcome to read the original paper, though this isn't necessary to do this question: `https://arxiv.org/abs/1703.03400`

It's worth thinking a bit about how the meta-learner might solve this problem. Clearly, 5 iterations would not be enough to train a generic MLP from scratch. But suppose you initialized the network such that each hidden unit computed a basis function which takes the value 1 on the interval $[k, k+1]$ for some integer k, and 0 everywhere else. Then you could fit the function simply by adjusting the weights in the output layer, which is just a linear regression problem, and can therefore be solved pretty quickly. Hence, such a network would be a great according to MAML's objective. Of course, there are probably other good ways for MAML to solve this problem, and we don't know what it will actually do when we run it.

The code you actually have to write is mostly straightforward once you know how to use Autograd. The challenging (and hopefully valuable) part is wrapping your head around how the starter code works. This is defined in `maml.py`. Here are the functions and classes it defines:

- `net_predict`: this implements the forward pass for an MLP. The parameters are stored in a Python dict `params`.
- `random_init`: initializes the weights to a Gaussian with a small standard deviation.
- `ToyDataGen`: This class generates the random piecewise linear functions.
- `gd_step`: Performs one step of gradient descent. Note that this function returns a new set of parameters, rather than modifying the arrays that were passed in.
- `InnerObjective`: The cost function for *one* regression dataset. This is just mean squared error.
- `MetaObjective`: The MAML objective, i.e. the inner cost after `num_steps` steps of gradient descent.
- `train`: Runs the actual training, i.e. repeatedly samples random regression datasets and does gradient descent on the meta-objective.

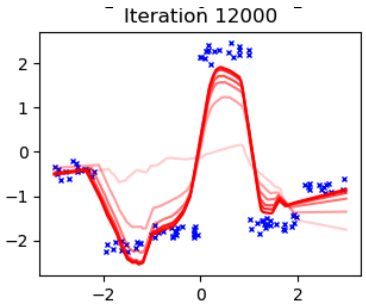
Here is what you need to do. Each of these parts requires only a few lines of code, and you should not need to do any messy derivations.

(a) **[2pts]** Implement `gd_step`. You should do this by calling `ag.grad`.

(b) **[2pts]** Implement `MetaObjective.__call__`. (This is Python syntax for the method that gets called when you call the class instance as if it were a function, i.e. `meta_obj(params)`.) Your implementation should call `gd_step`.

(c) **[1pt]** Finish the implementation of `train`. I.e., sample a random regression dataset, and do a gradient descent step on the meta-objective.

Once you finish the code, calling `train` will produce a visualization such as the following, where the thinnest line corresponds to the initial parameters learned by MAML, and thicker lines correspond to more steps of SGD:



Observe that your solution will involve calling `gd_step` on a function which itself calls `gd_step`. Since `gd_step` calls `ag.grad`, this means you are calling `ag.grad` on a computation graph which was itself generated by `ag.grad`. Understanding why this happens is an important part of understanding the code.

Submit your code solution as `maml.py`. You don't need to submit anything else for this question.

2. **Adam. [5pts]** Adam[4] is a widely used optimization algorithm which essentially combines the benefits of RMSprop and momentum. Here is a slightly simplified version of the original algorithm. All arithmetic operations, such as squaring or division, are applied elementwise.

   $\mathbf{m}_0 \leftarrow 0$
   $\mathbf{v}_0 \leftarrow 0$
   $t \leftarrow 0$
   While $\boldsymbol{\theta}_t$ not converged:
   $\mathbf{g}_t \leftarrow \nabla \mathcal{J}(\boldsymbol{\theta}_{t-1})$
   $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$
   $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$
   $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha_{\mathrm{A}} \mathbf{m}_t / (\sqrt{\mathbf{v}_t} + \epsilon_{\mathrm{A}})$

   The hyperparameters of the algorithm are the learning rate α_{A}, the moments timescales β_1 and β_2, and the damping term ϵ_{A}. The A subscript stands for "Adam," to distinguish these hyperparameters from the other algorithms discussed in this question.

   Here is what you need to analyze:

   (a) **[1pt]** Recall the RMSprop algorithm, rewritten here to match the notation of this question:

   $\mathbf{v}_0 \leftarrow 0$
   $t \leftarrow 0$
   While $\boldsymbol{\theta}_t$ not converged:
   $\mathbf{g}_t \leftarrow \nabla \mathcal{J}(\boldsymbol{\theta}_{t-1})$
   $\mathbf{v}_t \leftarrow \gamma \mathbf{v}_{t-1} + (1-\gamma)\mathbf{g}_t^2$
   $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha_{\mathrm{R}} \mathbf{g}_t / (\sqrt{\mathbf{v}_t} + \epsilon_{\mathrm{R}})$

   The hyperparameters are α_{R}, γ, and ϵ_{R}. Specify Adam hyperparameters $(\alpha_{\mathrm{A}}, \beta_1, \beta_2, \epsilon_{\mathrm{A}})$ which make Adam equivalent to RMSprop with hyperparameters $(\alpha_{\mathrm{R}}, \gamma, \epsilon_{\mathrm{R}})$. You should explain your answer, though a full derivation isn't required.

   (b) **[2pts]** Now consider SGD with momentum:

   $\mathbf{p}_0 \leftarrow 0$
   $t \leftarrow 0$
   While $\boldsymbol{\theta}_t$ not converged:
   $\mathbf{p}_t \leftarrow \mu \mathbf{p}_{t-1} - (1-\mu)\nabla \mathcal{J}(\boldsymbol{\theta}_{t-1})$
   $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \alpha_{\mathrm{S}} \mathbf{p}_t$

[4] Here is the original paper, but you don't need to read it to solve this question: `https://arxiv.org/abs/1412.6980`

Specify Adam hyperparameters $(\alpha_{\mathrm{A}}, \beta_1, \beta_2, \epsilon_{\mathrm{A}})$ which make Adam approximately equivalent to momentum SGD with parameters $(\alpha_{\mathrm{S}}, \mu)$. Explain your answer.

(c) **[2pts]** An important fact about Adam is that it is invariant to rescaling of the loss function. I.e., suppose we have a loss function $\mathcal{L}(y, t)$, and we define a modified loss function as $\tilde{\mathcal{L}}(y, t) = C \cdot \mathcal{L}(y, t)$ for some positive constant C. Show that for $\epsilon_{\mathrm{A}} = 0$, Adam is invariant to this rescaling, i.e. it passes through the same sequence of iterates $\boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_T$.

Hint: Denote the quantities computed by Adam on the modified loss function as $\tilde{\mathbf{g}}_t$, $\tilde{\mathbf{m}}_t$, etc. Use induction to find relationships between these and the original $\mathbf{g}_t$, $\mathbf{m}_t$, etc.)