Optimization as a Model for Few-Shot Learning

Sachin Ravi
Princeton University & Twitter

In collaboration with

Hugo Larochelle
A RESEARCH AGENDA

• Deep learning successes have required a lot of labeled training data
  ‣ collecting and labeling such data requires significant human labor
  ‣ is that really how we’ll solve AI?

• Alternative solution: exploit other sources of data that are imperfect but plentiful
  ‣ unlabeled data (unsupervised learning)
  ‣ multi-modal data (multimodal learning)
  ‣ multi-domain data (transfer learning)
A RESEARCH AGENDA

• One example of this problem: few-shot learning
  ‣ Defined as k-shot, N-class classification: k examples for each of N classes
  ‣ Model needs to generalize after seeing few examples from each class
META-LEARNING

• How to do well at few-shot training task?
  ‣ Training algorithms such as SGD or ADAM prone to overfitting with random initialization
    - hard to know what good initialization is
  ‣ We want to design a training algorithm for each small dataset
    - given training set with few examples
    - should output parameters $\theta$ for model $M$ that generalize well to test set

• Idea: let's learn such a training algorithm, end-to-end
  ‣ this is known as meta-learning or learning-to-learn
META-LEARNING

• Consider a training algorithm
  ‣ input: training set $D_{\text{train}} = \{(X_i, Y_i)\}_{i=1}^T$
  ‣ output: parameters $\theta$ of model $M$
  ‣ objective: good performance on test set $D_{\text{test}} = (X, Y)$

• Desire a meta-learning algorithm
  ‣ input: meta-training set $D_{\text{meta-train}} = \{(D_{\text{train}}^{(n)}, D_{\text{test}}^{(n)})\}_{n=1}^N$
  ‣ output: parameters $\Theta$ representing a training algorithm
  ‣ objective: good performance on meta-test set $D_{\text{meta-test}} = \{(D_{\text{train}}^{(n')}, D_{\text{test}}^{(n')})\}_{n'=1}^{N'}$
META-LEARNING

Figure 1: Example of meta-learning setup. The top represents the meta-training set \(D_{\text{meta\_train}}\), where inside each gray box is a separate dataset that consists of the training set \(D_{\text{train}}\) (left side of dashed line) and the test set \(D_{\text{test}}\) (right side of dashed line). In this illustration, we are considering the 1-shot, 5-class classification task where for each dataset, we have one example from each of 5 classes (each given a label 1-5) in the training set and 2 examples for evaluation in the test set.

The meta-test set \(D_{\text{meta\_test}}\) is defined in the same way, but with a different set of datasets that cover classes not present in any of the datasets in \(D_{\text{meta\_train}}\) (similarly, we additionally have a meta-validation set that is used to determine hyper-parameters).

Our key observation that we leverage here is that this update resembles the update for the cell state in an LSTM (Hochreiter & Schmidhuber, 1997)

\[
c_t = f_t c_{t-1} + i_t \tilde{c}_t,
\]

if \(f_t = 1\),

\[
c_t = \phi_t c_{t-1},
i_t = \rho_t,
\]

and \(\tilde{c}_t = r \phi_t 1_L t\).

Thus, we propose training a meta-learner LSTM to learn an update rule for training a neural network. We set the cell state of the LSTM to be the parameters of the learner, or \(c_t = \phi_t\), and the candidate cell state \(\tilde{c}_t = r \phi_t 1_L t\), given how valuable information about the gradient is for optimization. We define parametric forms for \(i_t\) and \(f_t\) so that the meta-learner can determine optimal values through the course of the updates.

Let us start with \(i_t\), which corresponds to the learning rate for the updates. We let

\[
i_t = W_I \cdot \phi_t 1_L t + b_I,
\]

meaning that the learning rate is a function of the current parameter value \(\phi_t 1_L t\), the current gradient \(r \phi_t 1_L t\), the current loss \(L_t\), and the previous learning rate \(i_t\). With this information, the meta-learner should be able to finely control the learning rate so as to train the learner quickly while avoiding divergence.

As for \(f_t\), it seems possible that the optimal choice isn't the constant 1. Intuitively, what would justify shrinking the parameters of the learner and forgetting part of its previous value would be if the learner is currently in a bad local optima and needs a large change to escape. This would correspond to a situation where the loss is high but the gradient is close to zero. Thus, one proposal for the forget gate is to have it be a function of that information, as well as the previous value of the forget gate:

\[
f_t = W_F \cdot \phi_t 1_L t + b_F \Delta
\]

Additionally, notice that we can also learn the initial value of the cell state \(c_0\) for the LSTM, treating it as a parameter of the meta-learner. This corresponds to the initial weights of the classifier (that the meta-learner is training). Learning this initial value lets the meta-learner determine the optimal initial weights of the learner so that training begins from a beneficial starting point that allows...
Let us start with meta-learning. We define parametric forms for the candidate cell state of the LSTM (Hochreiter & Schmidhuber, 1997) if the learner is currently in a bad local optima and needs a large change to escape. This would justify shrinking the parameters of the learner and forgetting part of its previous value. Additionally, notice that we can also learn the initial value of the cell state for the forget gate: its value is to have it be a function of that information, as well as the previous value of the learner.

We set the cell state of the LSTM to be the parameters of the learner, or it as a parameter of the meta-learner. This corresponds to the initial weights of the classifier (that cover classes not present in any of the datasets in the meta-validation set that is used to determine hyper-parameters). Intuitively, what would it mean to avoid divergence?

Our key observation that we leverage here is that this update resembles the update for the cell state in an LSTM (Hochreiter & Schmidhuber, 1997). Thus, we propose training a meta-learner that is responsible for determining the optimal initial value of the cell state, which corresponds to the learning rate for the updates. We let the meta-learner determine the optimal initial value lets the meta-learner determine the optimal initial value of the cell state, given how valuable information about the gradient is for optimization. We define parametric forms for the candidate cell state.
A META-LEARNING MODEL

• How to parametrize training algorithms?
  ‣ we take inspiration from the gradient descent algorithm:
    \[
    \theta_t = \theta_{t-1} - \alpha_t \nabla \theta_{t-1} \mathcal{L}_t
    \]
  ‣ we parametrize this update similarly to LSTM state updates:
    \[
    c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t
    \]
    - state \( c_t \) is model \( M \)'s parameters
    - state candidate \( \tilde{c}_t \) is the negative gradient
    - \( f_t \) and \( i_t \) are LSTM gates:
      \[
      i_t = \sigma \left( W_I \cdot \left[ \nabla_{\theta_{t-1}} \mathcal{L}_t, \mathcal{L}_t, \theta_{t-1}, i_{t-1} \right] + b_I \right)
      \]
      \[
      f_t = \sigma \left( W_F \cdot \left[ \nabla_{\theta_{t-1}} \mathcal{L}_t, \mathcal{L}_t, \theta_{t-1}, f_{t-1} \right] + b_F \right)
      \]
META-LEARNING UPDATES

Under review as a conference paper at ICLR 2017

Figure 1: Computational graph for the forward pass of the meta-learner. The dashed line divides examples from the training set $D_{\text{train}}$ and test set $D_{\text{test}}$. Each $(X_i, Y_i)$ is the $i$th batch from the training set whereas $(X, Y)$ is all the elements from the test set. The dashed arrows indicate that we do not back-propagate through that step when training the meta-learner. We refer to the learner as $M$, where $M(X; ✓)$ is the output of learner $M$ using parameters $✓$ for inputs $X$. We also use $r_t$ as a shorthand for $r✓_t$.

Thus to have training conditions match those of test time. During evaluation of the meta-learning, for each dataset $D = (D_{\text{train}}, D_{\text{test}}) \in D_{\text{meta}}$, a good meta-learner model will, given a series of learner gradients and losses on the training set $D_{\text{train}}$, suggest a series of updates for the learner model that trains it towards good performance on the test set $D_{\text{test}}$.

The training objective we use is the loss $L_{\text{test}}$ of the final learner model on $D_{\text{test}}$. While iterating over the examples in $D_{\text{train}}$, at each time step $t$ the LSTM meta-learner receives $(r✓_t, L_t)$ from the learner and proposes the new set of parameters $✓_t$. The process repeats for $T$ steps, after which the learner and its final parameters are evaluated on the test set to produce the loss that is then used to train the meta-learner. The training algorithm is described in Algorithm 1 and the corresponding computational graph is shown in Figure 1.

3.3.1 Gradient Independence Assumption

Notice that our formulation would imply that the losses $L_t$ and gradients $r✓_t$ of the learner are dependent on the parameters of the meta-learner. Gradients on the meta-learner’s parameters should normally take this dependency into account. However, as discussed by Andrychowicz et al. (2016), this complicates the computation of the meta-learner’s gradients. Thus, following Andrychowicz et al. (2016), we make the simplifying assumption that these contributions to the gradients aren’t important and can be ignored, which allows us to avoid taking second derivatives, a considerably expensive operation. We were still able to train the meta-learner effectively in spite of this simplifying assumption.

3.3.2 Initialization of Meta-Learner LSTM

When training LSTMs, it is advised to initialize the LSTM with small random weights and to set the forget gate bias to a large value so that the forget gate is initialized to be close to 1, thus enabling gradient flow (Zaremba, 2015). In addition to the forget gate bias setting, we found that we needed to initialize the input gate bias to be small so that the input gate value (and thus the learning rate) used by the meta-learner LSTM starts out being small. With this combined initialization, the meta-learner starts close to normal gradient descent with a small learning rate, which helps initial stability of training.
META-LEARNING UPDATES

Under review as a conference paper at ICLR 2017

Figure 1: Computational graph for the forward pass of the meta-learner. The dashed line divides examples from the training set $D_{\text{train}}$ and test set $D_{\text{test}}$. Each $((X_i, Y_i))$ is the $i$th batch from the training set whereas $((X, Y))$ is all the elements from the test set. The dashed arrows indicate that we do not back-propagate through that step when training the meta-learner. We refer to the learner as $M$, where $M((X); \theta)$ is the output of learner $M$ using parameters $\theta$ for inputs $X$. We also use $r_t$ as a shorthand for $r_{\theta_t}$.1

To have training conditions match those of test time. During evaluation of the meta-learning, for each dataset $D = (D_{\text{train}}, D_{\text{test}})$, a good meta-learner model will, given a series of learner gradients and losses on the training set $D_{\text{train}}$, suggest a series of updates for the learner model that trains it towards good performance on the test set $D_{\text{test}}$. Thus to match test time, when considering each dataset $D \in D_{\text{meta}}$, the training objective we use is the loss $L_{\text{test}}$ of the final learner model on $D$'s test set $D_{\text{test}}$. While iterating over the examples in $D$'s training set $D_{\text{train}}$, at each time step $t$, the LSTM meta-learner receives $(r_{\theta_t}, L_t)$ from the learner and proposes the new set of parameters $\theta_{t+1}$. The process repeats for $T$ steps, after which the learner and its final parameters are evaluated on the test set to produce the loss that is then used to train the meta-learner. The training algorithm is described in Algorithm 1 and the corresponding computational graph is shown in Figure 1.

3.3.1 Gradient Independence Assumption

Notice that our formulation would imply that the losses $L_t$ and gradients $r_{\theta_t}$ of the learner are dependent on the parameters of the meta-learner. Gradients on the meta-learner's parameters should normally take this dependency into account. However, as discussed by Andrychowicz et al. (2016), this complicates the computation of the meta-learner's gradients. Thus, following Andrychowicz et al. (2016), we make the simplifying assumption that these contributions to the gradients aren't important and can be ignored, which allows us to avoid taking second derivatives, a considerably expensive operation. We were still able to train the meta-learner effectively in spite of this simplifying assumption.

3.3.2 Initialization of Meta-Learner LSTM

When training LSTMs, it is advised to initialize the LSTM with small random weights and to set the forget gate bias to a large value so that the forget gate is initialized to be close to 1, thus enabling gradient flow (Zaremba, 2015). In addition to the forget gate bias setting, we found that we needed to initialize the input gate bias to be small so that the input gate value (and thus the learning rate) used by the meta-learner LSTM starts out being small. With this combined initialization, the meta-learner starts close to normal gradient descent with a small learning rate, which helps initial stability of training.
Algorithm 1 Train Meta-Learner

**Input:** Meta-training set \( \mathcal{D}_{meta-train} \), Learner \( M \) with parameters \( \theta \), Meta-Learner \( R \) with parameters \( \Theta \).

1: \( \Theta_0 \leftarrow \) random initialization
2: 
3: for \( d = 1, n \) do
4: \( D_{train}, D_{test} \leftarrow \) random dataset from \( \mathcal{D}_{meta-train} \) \( \triangleright \) Initialize learner parameters
5: \( \theta_0 \leftarrow c_0 \)
6: 
7: for \( t = 1, T \) do
8: \( X_t, Y_t \leftarrow \) random batch from \( D_{train} \) \( \triangleright \) Get loss of learner on train batch
9: \( \mathcal{L}_t \leftarrow \mathcal{L}(M(X_t; \theta_{t-1}), Y_t) \)
10: \( c_t \leftarrow R((\nabla_{\theta_{t-1}} \mathcal{L}_t, \mathcal{L}_t); \Theta_{d-1}) \) \( \triangleright \) Get output of meta-learner using Equation 2
11: \( \theta_t \leftarrow c_t \) \( \triangleright \) Update learner parameters
12: end for
13: 
14: \( X, Y \leftarrow D_{test} \) \( \triangleright \) Get loss of learner on test batch
15: \( \mathcal{L}_{test} \leftarrow \mathcal{L}(M(X; \theta_T), Y) \)
16: Update \( \Theta_d \) using \( \nabla_{\Theta_{d-1}} \mathcal{L}_{test} \) \( \triangleright \) Update meta-learner parameters
17: 
18: end for

---

### 3.4 Batch Normalization

Batch Normalization (Ioffe & Szegedy, 2015) is a recently proposed method to stabilize and thus speed up learning of deep neural networks by reducing internal covariate shift within the learner’s hidden layers. This reduction is achieved by normalizing each layer’s pre-activation, by subtracting by the mean and dividing by the standard deviation. During training, the mean and standard deviation are estimated using the current batch being trained on, whereas during evaluation a running average of both statistics calculated on the training set is used. We need to be careful with batch normalization for the learner network in the meta-learning setting, because we do not want to collect mean and standard deviation statistics during meta-testing in a way that allows information to leak between different datasets being considered. One easy way to prevent this issue is to not collect statistics at all during the meta-testing phase, but just use our running averages from meta-training. This, however, has a bad impact on performance, because we have changed meta-training and meta-testing conditions, causing the meta-learner to learn a method of optimization that relies on batch statistics which it now does not have at meta-testing time. In order to keep the two phases as similar as possible, we found that a better strategy was to collect statistics for each dataset \( D \) during \( D_{meta-test} \), but then erase the running statistics when we consider the next dataset. Thus, during meta-training, we use batch statistics for both the training and testing set whereas during meta-testing, we use batch statistics for the training set (and to compute our running averages) but then use the running averages during testing. This does not cause any information to leak between different datasets, but also allows the meta-learner to be trained on conditions that are matched between training and testing. Lastly, because we are doing very few training steps, we computed the running averages so that higher preference is given to the later values.

### 4 Related Work

While this work falls within the broad literature of transfer learning in general, we focus here on positioning it relative to previous work on meta-learning and few-shot learning.
TO SUM UP

• We use our meta-learning LSTM to model parameter dynamics during training
  ‣ LSTM parameters are shared across $M$’s parameters (i.e. treated like a large minibatch)
  ‣ learns $c_0$, which is like learning $M$’s initialization

• Inputs to meta-learning LSTM are the loss and gradient of learner
  ‣ we use the preprocessing proposed by Andrychowicz et al. (2016)

• It is trained to produce parameters that have low loss on the corresponding test set
  ‣ possible thanks to backprop (though we ignore gradients through the inputs of the LSTM)

• Model $M$ uses batch normalization
  ‣ we are careful to avoid “leakage” between and within meta-sets
RELATED WORK

• Learning to learn using gradient descent (2001)
  Sepp Hochreiter, A. Steven Younger, and Peter R. Conwell
  ‣ LSTM-based meta-learner that isn’t using $M$’s gradients and was applied to synthetic learning problems

• Gradient-based hyperparameter optimization through reversible learning (2015)
  Dougal Maclaurin, David Duvenaud, and Ryan P Adams
  ‣ learns the learning rates of each time-step of minibatch SGD

• Learning to learn by gradient descent by gradient descent (2016)
  Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas
  ‣ LSTM outputs the update, instead of using its cell state explicitly for that

• Matching networks for one shot learning (2016)
  Oriol Vinyals, Charles Blundell, Timothy P. Lillicrap, Koray Kavukcuoglu, and Daan Wierstra
  ‣ learns a metric that generalizes well to new dataset with meta-learning
EXPERIMENT

- Mini-ImageNet
  - random subset of 100 classes (64 meta-training, 16 meta-validation, 20 meta-testing)
  - random sets $D_{train}$ are generated by randomly picking 5 classes from class subset
  - model $M$ is a small 4-layer CNN; meta-learner LSTM has 2 layers

<table>
<thead>
<tr>
<th>Model</th>
<th>5-class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-shot</td>
</tr>
<tr>
<td>Baseline-finetune</td>
<td>28.86 ± 0.54%</td>
</tr>
<tr>
<td>Baseline-nearest-neighbor</td>
<td>41.08 ± 0.70%</td>
</tr>
<tr>
<td>Matching Network</td>
<td>43.40 ± 0.78%</td>
</tr>
<tr>
<td>Matching Network FCE</td>
<td>43.56 ± 0.84%</td>
</tr>
<tr>
<td>Meta-Learner LSTM (OURS)</td>
<td>43.44 ± 0.77%</td>
</tr>
</tbody>
</table>
EXPERIMENT

- Learned input gates

![Learned input gates](image)
IN CONCLUSION

• We consider learning on multi-domain data, in the form of few-shot learning problem
  ‣ rather than usual train/test dataset split, each dataset consists of a set of datasets

• We parameterize a training algorithm in the form of a LSTM
  ‣ we train the meta-learner LSTM end-to-end on few-shot learning task
  ‣ parameters of LSTM represent both the training algorithm and initialization of model \( M \)

• We evaluate our meta-learner model on mini-ImageNet dataset
  ‣ the meta-learner model is competitive with state-of-the-art metric-learning methods
THANKS!