Learning to Learn Neural Networks

Tom Bosc
INRIA, France
tom.bosc@inria.fr

Abstract

Meta-learning consists in learning learning algorithms. We use a Long Short Term Memory (LSTM) based network to learn to compute on-line updates of the parameters of another neural network. These parameters are stored in the cell state of the LSTM. Our framework allows to compare learned algorithms to hand-made algorithms within the traditional train and test methodology. In an experiment, we learn a learning algorithm for a one-hidden layer Multi-Layer Perceptron (MLP) on non-linearly separable datasets. The learned algorithm is able to update parameters of both layers and generalise well on similar datasets.

1 Introduction

Meta-learning is the process of learning to fit parameters, that is, learning to learn. In this article, we focus on learning on-line learning algorithms : the data arrives sequentially and is shown only once. Thus, the meta-learning system is forced to update the parameters of the model after each observation.

Hochreiter et al. [1] show that it is possible to train a Recurrent Neural Network (RNN) to learn the coefficients of quadratic functions under supervision using gradient descent. At each timestep, the RNN is given the inputs of the current timestep and the previous target. With a simple objective – correctly predicting the targets at each timestep – the RNN learns to improve its predictions with appropriate updates of its internal state. They use a LSTM [2] with an additional hidden layer between the LSTM block and the output.

In this work, we introduce a separator flag that split each sequence into a train and a test set. This enables us to compare learned algorithms to hand-made algorithms. In an experiment, a LSTM-based network is trained to learn a small one-hidden-layer MLP on a binary classification task. The weights of the MLP are stored in the LSTM memory state. We show that potentially complex and deep networks could be learned using this technique.

2 Framework

Consider N datasets $\{(x_1^{(i)}, y_1^{(i)}), ..., (x_{n_i}^{(i)}, y_{n_i}^{(i)})\} \in D_{train}$ where $(x_t^{(i)}, y_t^{(i)})$ are n_i pairs of inputs and targets that are i.i.d. according to the i-th distribution. Similarly, we have N' other datasets in D_{test}. We drop the index (i) notation when it is not necessary.

We use a deterministic RNN that breaks down in two parts. The first part, the model, computes an estimate o_t of the targets y_t given the inputs x_t. It is parametrised by θ_t, the parameter estimate at time t, which is the hidden state of the RNN at each timestep.

$$o_t = f_{\theta_t}(x_t) \tag{1}$$

θ_t is updated at every timestep by the second part of the network, the learner. The learner is an on-line updating mechanism parametrised by α and θ_1, the initial state. It uses the inputs x_t, the

targets y_t, the prediction of the model o_t to compute the updates θ_{t+1}. In addition, it uses a special input $1_{t<\tau}$ (indicator function) which role is explained below.

$$\theta_{t+1} = g_\alpha(x_t, y_t * 1_{t<\tau}, 1_{t<\tau}, o_t) \tag{2}$$

We call *learning* the process of learning using the learner, i.e. learning θ_t. *Meta-learning* is the process of learning the learner, i.e. learning α and θ_1, the initial parameter value. Note that the initial state of the RNN θ_1, which corresponds to some sort of prior is optimised during the process of meta-learning.

The above update scheme does not impose that we show a target at each timestep, thanks to $1_{t<\tau}$. It is useful at test time, when we want to use the model for prediction without having the targets. Our datasets, from both D_{train} and D_{test}, are themselves divided into training sets and test sets. This division is artificial when we generate datasets (for meta-learning or evaluating the learner), but is not when we are actually using real data. Let's call τ_i the indices for each dataset that separate the training set from the test set. $(x_t^{(i)}, y_t^{(i)})$ belongs to the i-th training set for each $t < \tau_i$ and to the i-th test set for each $t \geq \tau_i$. The RNN takes inputs sequences that are created using an arbitrary ordering of the samples, where all the training samples are shown first. [1] As we do not show any targets for samples in the test sets, we pass this information as a binary flag so that the learner does not misinterpret zeros as targets. This flag is $1_{t<\tau}$.

In this work, meta-learning is a byproduct of learning to correctly predict targets. There are no constraints on the learner other than producing a model that does good prediction, avoiding overfitting and underfitting. Evaluation of a learning algorithm should be done on a held-out test set. That is why the meta-learning objective is an average loss on the test sets of datasets in D_{train}:

$$\underset{\alpha,\theta_1}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \frac{1}{n_i} \sum_{t=\tau_i}^{n_i} L(f_{\theta_t}(x_t^{(i)}), y_t^{(i)}) \tag{3}$$

Although, we introduce an alternative cost function used only for training purposes. It is the mean of the loss of all samples from all datasets in D_{train}:

$$\underset{\alpha,\theta_1}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \frac{1}{n_i} \sum_{t=1}^{n_i} L(f_{\theta_t}(x_t^{(i)}), y_t^{(i)}) \tag{4}$$

On the one hand, the cost (3) prevents any kind of hard-wiring between the outputs and the targets that could happen if we decide to repeat inputs and targets during several timesteps. Indeed, learners that cheat by copying the target and pasting it at subsequent timesteps will fail to predict correctly on the test set. On the other hand, meta-learning is much harder with (3) because there isn't any supervision during the learning phase. In our experiments, we use cost (3) to evaluate learning algorithms against each other, but use (4) in order to carry out meta-learning.

3 Implementation

The learner parameters α and θ_1 are optimised by a learning algorithm and the model parameters θ_t are optimised by the learner. In this work, f and g should be differentiable because the errors at each timestep are backpropagated not only through the learner but also through the model. We use a LSTM-based network to store the parameters θ_t of the model. Thus, the learner only learn to update the weights, that is, to compute useful LSTM gate parameters. Our framework slightly differ from [1] in that we first compute the prediction and then update the weights. [2]

g_α is an LSTM block which gates are computed after the concatenation of the inputs went through several regular layers. It has n fully connected layers of the form : $g_i(x) = h_i(W_i x + b_i)$ with h_i an activation function such as tanh, sigmoid, Rectified Linear Unit (ReLU), etc. The output of the last

[1] It should be possible to interlace training samples with test samples and alternate training and predicting phases, but as we want to be able to evaluate the learned algorithm against other algorithms, we will not do this.

[2] That is why the learner can receive x_t and y_t at the same timestep and does not hard-wire the target to the output o_t. This does not make any difference in performance and is not an improvement over [1].

layer x^* is used to compute the gate parameters of the LSTM block. There isn't any LSTM output gate because the output at each timestep is provided by the model.

$$\begin{aligned} x^* &= g_1 \circ ... \circ g_n([x_t, y_t * 1_{t<\tau}, 1_{t<\tau}, o_t]) \\ z_t &= W_z x^* + b_z \\ i_t &= \sigma(W_i x^* + b_i) \\ f_t &= \sigma(W_f x^* + b_f) \end{aligned} \tag{5}$$

The parameters at each timestep are given by :

$$\theta_{t+1} = g_\alpha(x_t, y_t * 1_{t<\tau}, 1_{t<\tau}, o_t) = i_t * z_t + f_t * \theta_t \tag{6}$$

It should be noted that the learner and the model are inseparable. Each component of each gate parameter in the learner is bound to a specific parameter in the model. That is also why the learner can learn without seeing the parameters of the model as an input. But it could be argued that the behavior of certain parameters of the model (for example, units of the same layer in the case of a neural network) could and should be generalised. This criticism could be addressed in a future work.

4 Experiment

We train the learner described above to fit parameters of a neural network on binary classification tasks. The model is a one-hidden layer MLP of $n_{in} = 5$ inputs, 32 hidden units and 1 output. The activation function of the first layer is a ReLU and the second is the sigmoid function. Counting the biases, there are 225 parameters to learn. The learner contains two layers of size 128 and 256 before the LSTM gates that are activated by ReLUs. There are 207651 meta-learnable parameters. The loss function is cross-entropy.

Our training and test datasets are generated randomly in a hierarchical way. The process described here is arbitrarily designed and creates noisy non-linearly separable data. We generate one random covariance matrix per dataset by computing the product of a $n_{in} \times n_{in}$ matrix of random coefficients (uniformly distributed from -1 to 1) with its transpose. It is the covariance of a multivariate Gaussian that produces samples X for this dataset. In order to model noise, we draw from an exponential distribution (of parameter $\beta = 2$) a parameter ϵ_i for each dataset. It is then used to draw a vector v_i of size n_{in} from the exponential distribution of parameter $\beta = \epsilon_i$. Noise is then generated using a multivariate Gaussian of mean 0 and covariance defined by the diagonal matrix of diagonal v_i, and added to the samples. Then, we partition our inputs twice by applying a randomly chosen linear transformation and thresholding it to create binary classes. Then, a logical XOR is applied to obtain the targets. We throw away datasets which classes balances are below a minimal threshold (0.15). Postprocessing of the inputs include centering and scaling to unit variance.

The type and quantity of noise can be seen as a prior on the data. But it may also ease the optimisation process by introducing noise in the gradients, hypothesis which remains to be investigated.

We use a gradient descent algorithm called SMORMS3[3] with a learning rate of 10^{-3} on one randomly drawn sequence from D_{train} per iteration (Stochastic Gradient Descent).

In order to test whether the non-linearity of the model is effectively used by the learner, we present in the following array the mean and standard deviation of mean-cross entropy losses of hand-made algorithms against a learned algorithm. D_{test} consists of 500 randomly generated datasets of 200 samples each, with variable train over test ratios. The noise parameter β is lowered to 1 so that there is less noise than during meta-learning. Logistic regression is regularised with either L1 or L2, with $\lambda \in \{0.1, 1.0, 10\}$ and the hyperparameters are chosen with k-fold cross-validation.

	LSTM	Logistic Regression	SVM (linear)	SVM (RBF)
μ MCE	0.540	0.574	0.573	0.507
σ MCE	0.139	0.208	0.159	0.164

The learned algorithm achieves a significantly lower cost on average than the linear models. It generalises on non-separable data, which we know is the limit of these models. Although, it is worse than SVM with a RBF kernel. A proper evaluation that compare performance of the same MLP model trained with different learning algorithms is needed and is left for future work.

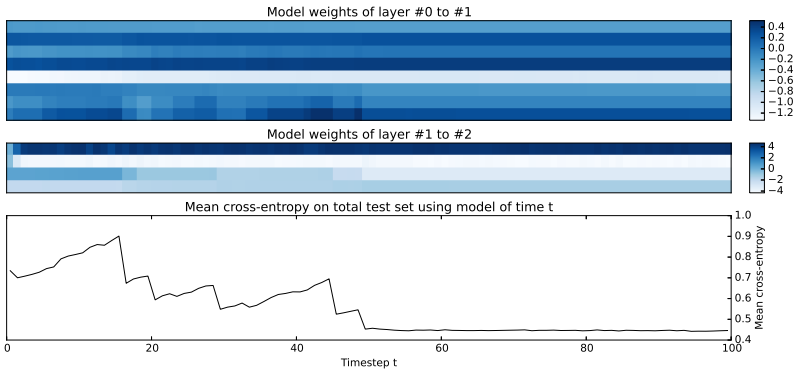

Figure 1: Illustration of the learning process on an example dataset split in training and test sets of the same size (50 samples each). The input space is $\mathbb{R}^2$ and the model is smaller than the one described above (2 inputs, 4 hidden units, 1 output). The matrices show the weights of the model across time. The biases are not shown. The plot at the bottom shows the mean cross-entropy on the entire test set using the model θ_t.

Figure 1 shows details of the learning process. The weight matrices indicates that some parameters are modified by the learner while others are left aside and practically do not change over time. This illustrates a trade-off between learning and meta-learning. One explanation is that the expressivity of the learner does not allow it to compute good updates of all the weights. It is better to let the meta-learning process select a subset of parameters which will will not be updated, optimise their initial value and focus on learning to update other parameters which provide relatively easy performance gains. Another possible explanation is that the optimisation process was stopped too early or that D_{train} contains too few datasets.

The parameters that are updated by the learner and the parameters updated during meta-learning are scattered across the two weight matrix of the model. It indicates that the learner can compute updates for neural networks that are more complicated than shallow networks.

The first half of the last plot can be seen as a learning curve. The loss is aligned with the updates made by the learner so the updates effectively improves the loss on the test set. Although individually, not all updates improve the performance on the test set. After having seen all the train samples, at iteration 50, the performance stops changing. The second halves of the weight matrices indicate that the learner stops to update the weights of the model on the test sets. With our architecture, nothing forbids the learner to keep modifying the parameters. A possible direction of research is to encourage updates during the test phase so that the learner simulate sampling from a distribution of parameters instead of using the same point-estimate over the whole test set.

5 Perspectives and conclusion

It is possible to train a neural network to train another neural network using backpropagation. More work is needed to characterise both the learning and the meta-learning process, in terms of generalisation and regularisation abilities as well as in speed of convergence.

References

[1] Sepp Hochreiter, A Younger, and Peter Conwell. Learning to learn using gradient descent. *Artificial Neural Networks - ICANN 2001*, pages 87–94, 2001.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[3] Simon Funk. Rmsprop loses to smorms3 - beware the epsilon! `http://sifter.org/~simon/journal/20150420.html`, 2015.