EGG: a toolkit for research on Emergence of lanGuage in Games

Eugene Kharitonov
Facebook AI
kharitonov@fb.com

Rahma Chaabouni
Facebook AI Research / LSCP
rchaabouni@fb.com

Diane Bouchacourt
Facebook AI
dianeb@fb.com

Marco Baroni
Facebook AI / ICREA
mbaroni@fb.com

Abstract

There is renewed interest in simulating language emergence among deep neural agents that communicate to jointly solve a task, spurred by the practical aim to develop language-enabled interactive AIs, as well as by theoretical questions about the evolution of human language. However, optimizing deep architectures connected by a discrete communication channel is technically challenging. We introduce EGG, a toolkit that simplifies the implementation of emergent-language communication games. EGG's modular design provides a set of building blocks that the user can combine to create new games, easily navigating the optimization and architecture space. We hope that the tool will lower the technical barrier, and encourage researchers from various backgrounds to work in this exciting area.

1 Introduction

Studying the languages that emerge when neural agents interact with each other recently became a vibrant area of research (Havrylov and Titov, 2017; Lazaridou et al., 2016, 2018; Kottur et al., 2017; Bouchacourt and Baroni, 2018; Lowe et al., 2019). Interest in this scenario is fueled by the hypothesis that the ability to interact through a human-like language is a prerequisite for genuine AI (Mikolov et al., 2016; Chevalier-Boisvert et al., 2019). Furthermore, such simulations might lead to a better understanding of both standard NLP models (Chaabouni et al., 2019) and the evolution of human language itself (Kirby, 2002).

For all its promise, research in this domain is technically very challenging, due to the discrete nature of communication. The latter prevents the use of conventional optimization methods, requiring either Reinforcement Learning algorithms (e.g., REINFORCE; Williams, 1992) or the Gumbel-Softmax relaxation (Maddison et al., 2016; Jang et al., 2016). The technical challenge might be particularly daunting for researchers whose expertise is not in machine learning, but in fields such as linguistics and cognitive science, that could contribute to this interdisciplinary research area.

To lower the starting barrier and encourage high-level research in this domain, we introduce the EGG (**E**mergence of lan**G**uage in **G**ames) toolkit. EGG aims at (1) providing reliable building bricks for quick prototyping, (2) serving as a library of pre-implemented games, and (3) providing tools for analyzing the emergent languages.

Notable features of EGG include: (a) Primitives for implementing single-symbol or variable-length communication (with vanilla RNNs (Elman, 1990), GRUs (Cho et al., 2014), LSTMs (Hochreiter and Schmidhuber, 1997), and Transformers (Vaswani et al., 2017)), (b) Training with optimization of the communication channel through REINFORCE or Gumbel-Softmax relaxation via a common

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

interface; (c) Simplified configuration of the general components, such as check-pointing, optimization, Tensorboard support, etc.; (d) A simple CUDA-aware command-line tool for hyperparameter grid-search. EGG is implemented in PyTorch (Paszke et al., 2017) under the MIT license.

2 EGG's architecture

In the first iteration of EGG, we concentrate on a simple class of games, involving a single, unidirectional (Sender $\rightarrow$ Receiver) message. In turn, messages can be either single-symbol or multi-symbol variable-length sequences. Our motivation for starting with this setup is two-fold. First, it corresponds to classic signaling games (Lewis, 1969), it already covers a large portion of the literature (e.g., 5 out of 6 relevant studies mentioned in Introduction) and it allows exploring many interesting research questions. Second, it constitutes a natural first step for further development; in particular, the majority of components should remain useful in multi-directional, multi-step setups.

Design principles As different training methods and architectures are used in the literature, our primary goal is to provide EGG users with the ability to easily navigate the space of common design choices. Building up on this idea, EGG makes switching between Gumbel-Softmax relaxation-based and REINFORCE-based training effortless, through the simple choice of a different wrapper. Similarly, one can switch between one-symbol communication and variable-length messages with little changes in the code.

We aim to maintain EGG minimalist and "hackable" by encapsulating the user-implemented agent architectures, the Reinforce/GS agent wrappers and the game logic into PyTorch modules. The user can easily replace any part.

Finally, since virtually any machine-learning experiment has common pieces, such as setting the random seeds, configuring the optimizer, model check-pointing, etc., EGG pre-implements many of them, reducing the necessary amount of boilerplate code to the minimum.

EGG design EGG, in its first iteration, operates over the following entities. Firstly, there are two distinct agent roles: **Sender** and **Receiver**. Sender and Receiver are connected via a one-directional communication channel from the former to the latter, that has to produce the game-specific output.

The next crucial entity is **Game**. It encapsulates the agents and orchestrates the game scenario: feeding the data to the agents, transmitting the messages, and getting the output of Receiver. Figure 1 illustrates EGG's game flow in a specific example. Game applies a user-provided **loss** function, which might depend on the outputs of Receiver, the message transmitted, and the data. The loss is minimized by **Trainer**. Trainer also controls model checkpointing, early stopping, etc.

The Trainer and Game modules are pre-implemented in EGG. In a typical scenario, the communication method (single or multiple symbol messages) will be implemented by EGG-provided wrappers. As a result, what is left for the user to implement consists of: (a) the data stream, (b) core (non-communication-related) parts of the agents, (c) the loss. The data interface that is expected by Trainer is an instance of the standard PyTorch data loader `utils.data.DataLoader`.

To implement Sender, the user must define a module that consumes the data and outputs a tensor. On Receiver's side, the user has to implement a module that takes an input consisting of a message embedding and possibly further data, and generates Receiver's output.

3 Optimizing the communication channel in EGG

EGG supports two widely adopted strategies for learning with a discrete channel, Gumbel-Softmax relaxation (used, e.g., by Havrylov and Titov (2017)) and REINFORCE (used, e.g., by Lazaridou et al. (2016)). Below, we briefly review both of them.

Gumbel-Softmax relaxation is based on the Gumbel-Softmax (GS) distribution (Maddison et al., 2016; Jang et al., 2016), that allows to approximate one-hot samples from a Categorical distribution. At the same time, GS admits reparametrization, hence allows backpropagation-based training. As a result, if Receiver and the game loss are differentiable w.r.t. their inputs, we can get gradients of all game parameters, including those of Sender, via conventional backpropagation.

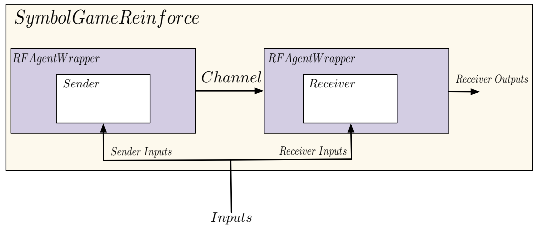

Figure 1: Example of EGG's game flow when using REINFORCE. White boxes (Sender and Receiver) represent the user-implemented agent architectures. The colored boxes are EGG-provided wrappers that implement a REINFORCE-based scenario. *SymbolGameReinforce* is an instance of the Game block, which sets up single-symbol Sender/Receiver game optimized with REINFORCE.

```python
class Sender(nn.Module):
    def __init__(self, vision, output_size):
        super(Sender, self).__init__()
        self.fc = nn.Linear(500, output_size)
        self.vision = vision

    def forward(self, x):
        with torch.no_grad():
            x = self.vision(x)
        x = self.fc(x)
        return x


class Receiver(nn.Module):
    def __init__(self, input_size):
        super(Receiver, self).__init__()
        self.fc = nn.Linear(input_size, 784)

    def forward(self, channel_input, receiver_input=None):
        x = self.fc(channel_input)
        return torch.sigmoid(x)

sender = Sender(vision, output_size)
receiver = Receiver(input_size)
```

Figure 2: MNIST game: Defining and instantiating the user-defined parts of the agents' architecture.

```python
sender = core.GumbelSoftmaxWrapper(sender, temperature=1.0)

receiver = core.SymbolReceiverWrapper(receiver, vocab_size,
                                      agent_input_size=400)

game = core.SymbolGameGS(sender, receiver, loss)

```

(a) Single-symbol communication, Gumbel-Softmax relaxation.

```python
sender = core.ReinforceWrapper(sender)

receiver = core.SymbolReceiverWrapper(receiver, vocab_size,
                                      agent_input_size=400)
receiver = core.ReinforceDeterministicWrapper(receiver)
game = core.SymbolGameReinforce(sender, receiver, loss, sender_entropy_coeff=0.05,
                                receiver_entropy_coeff=0.0)
```

(b) Single-symbol communication, REINFORCE.

```python
sender_rnn = core.RnnSenderGS(sender, vocab_size, emb_size, hidden_size,
                              cell="rnn", max_len=2, temperature=1.0)
receiver_rnn = core.RnnReceiverGS(receiver, vocab_size, emb_size,
                    hidden_size, cell="rnn")
game_rnn = core.SenderReceiverRnnReinforce(sender_rnn, receiver_rnn, loss,
                                           sender_entropy_coeff=0.025,
                                           receiver_entropy_coeff=0.0)
```

(c) Variable-length communication, Gumbel-Softmax relaxation.

```python
sender_rnn = core.RnnSenderReinforce(sender, vocab_size, emb_size, hidden_size,
                                     cell="gru", max_len=2)
receiver_rnn = core.RnnReceiverDeterministic(receiver, vocab_size, emb_size,
                    hidden_size, cell="gru")

game_rnn = core.SenderReceiverRnnGS(sender_rnn, receiver_rnn, loss)

```

(d) Variable-length communication, REINFORCE.

Figure 3: MNIST game: The user can choose different communication wrappers to switch between training regimes (Gumbel-Softmax or REINFORCE) and communication type (single-symbol or variable-length messages).

```python
trainer = core.Trainer(game=game, optimizer=optimizer,
                       train_data=train_loader,
                       validation_data=test_loader,
                       epoch_callback=None)
trainer.train(n_epochs=15)
```

Figure 4: MNIST game: Once the agents and the game are instantiated, the user must pass them to a Trainer, which implements the training/validation loop, check-pointing, etc.

REINFORCE is a standard Reinforcement Learning algorithm (Williams and Peng, 1991). The gradient estimate is found by sampling messages and outputs. A standard trick to reduce the variance of the estimate is to subtract an action-independent baseline from the optimized loss (Williams, 1992). EGG uses the running mean baseline.

Importantly, REINFORCE allows us to optimize agents even if the loss is not differentiable (e.g., 0/1 loss). However, if the loss and Receiver are differentiable, this can be leveraged by a "hybrid" approach: the gradient of Receiver's parameters can be found by backpropagation, while Sender is optimized via REINFORCE. This is a special case of gradient estimation using stochastic computation graphs (Schulman et al., 2015), and it is also supported in EGG.

4 Implementing a game

In this Section we walk through the main steps to build a communication game in EGG. We illustrate them through a MNIST (LeCun et al., 1998) communication-based autoencoding task: Sender observes an image and sends a message to Receiver. In turn, Receiver tries to reconstruct the image. We only cover here the core aspects of the implementation, ignoring standard pre- and post-processing steps, such as data loading. The full implementation can be found in an online tutorial.

We start by implementing the agents' architectures, as shown in Figure 2. Sender gets an input image to be processed by its pre-trained `vision` module, and returns its output after a linear transformation. The way Sender's output will be interpreted depends on the type of the used communication (discussed below). Receiver gets an input from Sender and returns an image-sized output with pixels valued in $[0, 1]$. Again, depending on the type of channel, the Receiver input will have a different semantics.

In the case of one-symbol communication, Sender's output is interpreted as logits of a distribution over the message vocabulary. Hence, the output dimensionality defines the size of the vocabulary. In the case of variable-length messages, Sender's output specifies the initial hidden state of an RNN cell. This cell is then "unrolled" to generate a message, until the end-of-sequence symbol (`eos`) is produced or maximum length is reached. Receiver's input is an embedding of the message: either the embedding of the single-symbol message, or the last hidden state of the RNN cell.

Once Sender and Receiver are defined, the user wraps them into EGG-implemented wrappers which determine the communication and optimization scenarios. Importantly, the actual user-specified Sender and Receiver architectures can be agnostic to whether single-symbol or variable-length communication is used; and to whether Gumbel-Softmax relaxation- or REINFORCE-based training is performed. In Figure 3 we illustrate different communication/training scenarios: (a) single-symbol communication, trained with Gumbel-Softmax relaxation, (b) single-symbol communication, trained with REINFORCE, (c) variable-length communication, trained with Gumbel-Softmax relaxation, (d) variable-length communication, trained with REINFORCE. After defining the Game instance, the user has only to pass it to `core.Trainer`, as shown in Figure 4, and start training.

5 Some pre-implemented games

EGG contains implementations of several games. They (a) illustrate how EGG can be used to explore interesting research questions, (b) provide reference usage patterns and building blocks, (c) serve as means to ensure reproducibility of studies reported in the literature. For example, EGG incorporates an implementation of the signaling game of Lazaridou et al. (2016) and Bouchacourt and Baroni (2018). Finally, EGG provides a pre-implemented game that allows to train agents entirely via the command line and external input/output files. We hope this will lower the learning curve for those who want to experiment with language emergence without previous coding experience.

6 Conclusion and future work

We introduced EGG, a toolkit for research on emergence of language in games. We outlined its main features design principles. We briefly reviewed how training with a discrete communication channel is performed. Finally, we walked through the main steps for implementing a MNIST autoencoding game using EGG.

In future, we want to support multi-direction and multi-step communication scenarios. We also want to add more advanced tooling for analyzing the properties of the emergent languages (such as compositionality Andreas, 2019). Finally, we will continue to extend the set of implemented games.

References

Jacob Andreas. 2019. Measuring compositionality in representation learning. In *ICLR*, New Orleans, LA.

Diane Bouchacourt and Marco Baroni. 2018. How agents see things: On visual representations in an emergent language game. In *EMNLP*.

Rahma Chaabouni, Eugene Kharitonov, Alessandro Lazaric, Emmanuel Dupoux, and Marco Baroni. 2019. Word-order biases in deep-agent emergent communication. In *ACL*.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. 2019. https://openreview.net/forum?id=rJeXCo0cYX BabyAI: First steps towards grounded language learning with a human in the loop. In *International Conference on Learning Representations*.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Jeffrey Elman. 1990. Finding structure in time. *Cognitive Science*, 14:179–211.

Serhii Havrylov and Ivan Titov. 2017. Emergence of language with multi-agent games: Learning to communicate with sequences of symbols. In *NIPS*.

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

Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with Gumbel-Softmax. *arXiv preprint arXiv:1611.01144*.

Simon Kirby. 2002. Natural language from artificial life. *Artificial life*, 8(2):185–215.

Satwik Kottur, José MF Moura, Stefan Lee, and Dhruv Batra. 2017. Natural language does not emerge'naturally'in multi-agent dialog. *arXiv preprint arXiv:1706.08502*.

Angeliki Lazaridou, Karl Moritz Hermann, Karl Tuyls, and Stephen Clark. 2018. Emergence of linguistic communication from referential games with symbolic and pixel input. *arXiv preprint arXiv:1804.03984*.

Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. 2016. Multi-agent cooperation and the emergence of (natural) language. *arXiv preprint arXiv:1612.07182*.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

David Lewis. 1969. *Convention*. Harvard University Press, Cambridge, MA.

Ryan Lowe, Jakob Foerster, Y-Lan Boureau, Joelle Pineau, and Yann Dauphin. 2019. On the pitfalls of measuring emergent communication. *arXiv preprint arXiv:1903.05168*.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. 2016. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*.

Tomas Mikolov, Armand Joulin, and Marco Baroni. 2016. A roadmap towards machine intelligence. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 29–61. Springer.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. 2015. Gradient estimation using stochastic computation graphs. In *NIPS*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Ronald J Williams and Jing Peng. 1991. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268.