Teacher-Student Curriculum Learning

Tambet Matiisen[*,†]
University of Tartu

Avital Oliver[*,‡]
OpenAI

Taco Cohen[*]
University of Amsterdam

John Schulman[*]
OpenAI

Abstract

We propose Teacher-Student Curriculum Learning (TSCL), a framework for automatic curriculum learning, where the Student tries to learn a complex task and the Teacher automatically chooses subtasks from a given set for the Student to train on. We describe a family of Teacher algorithms that rely on the intuition that the Student should practice more those tasks on which it makes the fastest progress, *i.e.* where the slope of the learning curve is highest. In addition, the Teacher algorithms address the problem of forgetting by also choosing tasks where the Student's performance is getting worse. We demonstrate that TSCL matches or surpasses the results of carefully hand-crafted curricula in two tasks: addition of decimal numbers with LSTM and navigation in Minecraft. Using our automatically generated curriculum enabled to solve a Minecraft maze that could not be solved at all when training directly on solving the maze, and the learning was an order of magnitude faster than uniform sampling of subtasks.

1 Introduction

Deep reinforcement learning algorithms have been used to solve difficult tasks in video games (Mnih et al., 2015), locomotion (Schulman et al., 2015; Lillicrap et al., 2015) and robotics (Levine et al., 2015). But tasks with sparse rewards like "Robot, fetch me a beer" remain challenging to solve with direct application of these algorithms. One reason is that the number of samples needed to solve a task with random exploration increases exponentially with the number of steps to get a reward (Langford, 2011). One approach to overcome this problem is to use curriculum learning (Bengio et al., 2009; Zaremba and Sutskever, 2014; Graves et al., 2016; Wu and Tian, 2017), where tasks are ordered by increasing difficulty and training only proceeds to harder tasks once easier ones are mastered. Curriculum learning helps when after mastering a simpler task the policy for a harder task is discoverable through random exploration.

To use curriculum learning, the researcher must:

- Be able to order subtasks by difficulty.
- Decide on a "mastery" threshold. This can be based on achieving certain score (Zaremba and Sutskever, 2014; Wu and Tian, 2017), which requires prior knowledge of acceptable performance of each task. Alternatively this can be based on a plateau of performance, which can be hard to detect given the noise in the learning curve.
- Continuously mix in easier tasks while learning harder ones to avoid forgetting. Designing these mixtures effectively is challenging (Zaremba and Sutskever, 2014).

In this paper, we describe a new approach called Teacher-Student Curriculum Learning (TSCL). The Student is the model being trained. The Teacher monitors the Student's training progress and

---

[*]Work done while interning at OpenAI.

[†]Correspondence to `tambet.matiisen@gmail.com`.

[‡]Author currently at Google Brain

determines the tasks on which the Student should train at each training step, in order to maximize the Student's progression through the curriculum. The Student can be any machine learning model. The Teacher is itself learning about the Student as it's giving tasks, all as part of a single training session.

We describe several Teacher algorithms based on the notion of learning progress (Oudeyer and Kaplan, 2007). The main idea is that the Student should practice more the tasks on which it is making fastest progress i.e. the learning curve slope is highest. To counter forgetting, the Student should also practice tasks where the performance is getting worse i.e. the learning curve slope is negative.

The main contributions of the paper are:

- We formalize TSCL, a Teacher-Student framework for curriculum learning as partially observable Markov decision process (POMDP).
- We propose a family of algorithms based on the notion of learning progress. The algorithms also address the problem of forgetting previous tasks.
- We evaluate the algorithms on two supervised and reinforcement learning tasks: addition of decimal numbers with LSTM and navigation in Minecraft.
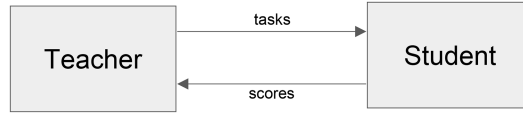
2 Teacher-Student Setup

Figure 1: The Teacher-Student setup

Figure 1 illustrates the Teacher-Student interaction. At each timestep, the Teacher chooses tasks for the Student to practice on. The Student trains on those tasks and returns back a score. The Teacher's goal is for the Student to succeed on a final task with as few training steps as possible. Usually the task is parameterized by a categorical value representing one of N subtasks, but one can imagine also multi-dimensional or continuous task parameterization. The score can be episode total reward in reinforcement learning or validation set accuracy in supervised learning.

We formalize the Teacher's goal of helping the Student to learn a final task as solving a partially observable Markov decision process (POMDP). We present two POMDP formulations: (1) *Simple*, best suited for reinforcement learning; and (2) *Batch*, best suited for supervised learning.

2.1 Simple POMDP Formulation

The simple POMDP formulation exposes the score of the Student on a single task and is well-suited for reinforcement learning problems.

- The state s_t represents the entire state of the Student (*i.e.* neural network parameters and optimizer state) and is not observable to the Teacher.
- The action a_t corresponds to the parameters of the task chosen by Teacher. In following we only consider a discrete task parameterization. Taking an action means training Student on that task for certain number of iterations.
- The observation o_t is the score $x_t^{(i)}$ of the task $i = a_t$ the Student trained on at timestep t, i.e. the episode total reward. While in theory the Teacher could also observe other aspects of the Student state like network weights, for simplicity we choose to expose only the score.
- Reward r_t is the change in score for the task the Student trained on at timestep t: $r_t = x_t^{(i)} - x_{t'_i}^{(i)}$, where t'_i is the previous timestep when the same task was trained on.

2.2 Batch POMDP Formulation

In supervised learning a training batch can include multiple tasks. Therefore action, observation, and reward apply to the whole training set and scores can be measured on a held-out validation set. This motivates the batch formulation of the POMDP:

- The state s_t represents training state of the Student.

- The action a_t represents a probability distribution over N tasks. Each training batch is sampled according to the distribution: $a_t = (p_t^{(1)}, \dots, p_t^{(N)})$, where $p_t^{(i)}$ is the probability of task i at timestep t.

- The observation o_t is the scores of all tasks after the training step: $o_t = (x_t^{(1)}, \dots, x_t^{(N)})$ In the simplest case the scores could be accuracies of the tasks in the training set. But in the case of minibatch training the model evolves during training and therefore additional evaluation pass is needed anyway to produce consistent results. Therefore we use a separate validation set that contains uniform mix of all tasks for this evaluation pass.

- The reward r_t is the sum of changes in evaluation scores from the previous timestep: $r_t = \sum_{i=1}^{N} x_t^{(i)} - x_{t-1}^{(i)}$.

This setup could also be used with reinforcement learning by performing training in batches of episodes. But because scoring one sample (one episode) in reinforcement learning is usually much more computationally expensive than in supervised learning, it makes sense to use simple POMDP formulation and make decision about the next task after each training step.

2.3   Optimization Criteria

For either of the POMDP formulations, maximizing the Teacher episode total reward is equivalent to maximizing the score of all tasks at the end of the episode: $\sum_{t=1}^{T} r_t = \sum_{i=1}^{N} x_{T_i}^{(i)}$, where T_i is the last training step where task i was being trained on[4].

While an obvious choice for optimization criteria would have been the performance in the *final task*, initially the Student might not have any success in the final task and this does not provide any meaningful feedback signal to the Teacher. Therefore we choose to maximize the sum of performances in *all tasks*. The assumption here is that in curriculum learning the final task includes the elements of all previous tasks, therefore good performance in the intermediate tasks usually leads to good performance in the final task.

3   Algorithms

POMDPs are typically solved using reinforcement learning algorithms. But those require many training episodes, while we aim to train the Student in one Teacher episode. Therefore, we resort to simpler heuristics. The basic intuition is that the Student should practice those tasks more for which it is making most progress (Oudeyer and Kaplan, 2007), while also practicing tasks that are at risk of being forgotten.
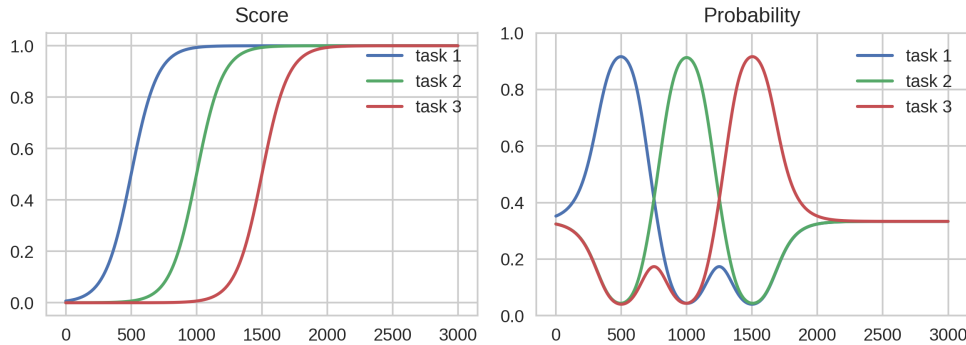

Figure 2: Idealistic curriculum learning. Left: Scores of different tasks improve over time, the next task starts improving once the previous task has been mastered. Right: Probability of sampling a task depends on the slope of the learning curve.

---

[4] Due to telescoping summation cancelling out all $x_t^{(i)}$ terms but the T_i[th].

Figure 2 is a demonstration of the ideal training progress in a curriculum learning setting:

1. At first, the Teacher has no knowledge so it samples from all tasks uniformly.
2. When the Student starts making progress on task 1, the Teacher allocates more probability mass to this task.
3. When the Student masters task 1, its learning curve flattens and the Teacher samples the task less often. At this point Student also starts making progress on task 2, so the Teacher samples more from task 2.
4. This continues until the Student masters all tasks. As all task learning curves flatten in the end, the Teacher returns to uniform sampling of the tasks.

The picture above is idealistic, since in practice some unlearning often occurs, *i.e.* when most of the probability mass is allocated to the task 2, performance on task 1 might get worse. To counter this the Student should also practice all learned tasks, especially those where unlearning occurs. For this reason we sample tasks according to the *absolute value* of the slope of the learning curve instead. If the change in scores is negative, this must mean that unlearning occurred and this task should be practiced more.

This description alone does not prescribe an algorithm. We need to propose a method of estimating learning progress from noisy task scores, and a way to balance exploration and exploitation. We take inspiration from algorithms for the non-stationary multi-armed bandit problem (Sutton and Barto, 1998) and adapt them to TSCL. For brevity we only give intuition for the simple formulation algorithms here, the formal descriptions can be found in appendices A and B.

3.1 Online algorithm

The Online algorithm is inspired by the basic non-stationary bandit algorithm (Sutton and Barto, 1998). It uses exponentially weighted moving average to track the expected return Q from different tasks:

$$Q_{t+1}(a_t) = \alpha r_t + (1 - \alpha)Q_t(a_t),$$

where α is learning rate. The next task can be chosen by ϵ-greedy exploration: sample a random task with probability ϵ, or $\mathrm{argmax}\, Q_t(a)$ otherwise.

Alternatively the next task can be chosen using Boltzmann distribution:

$$p(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{i=1}^{N} e^{Q_t(i)/\tau}},$$

where τ is the temperature of Boltzmann distribution. For details, see Algorithm 1 in Appendix A.

3.2 Naive algorithm

To estimate the learning progress more reliably one should practice the task several times. The Naive algorithm trains each task K times, observes the resulting scores and estimates the slope of the learning curve using linear regression. The regression coefficient is used as the reward in the above non-stationary bandit algorithm. For details, see Algorithm 2 in Appendix A.

3.3 Window algorithm

Repeating the task a fixed number of times is expensive, when clearly no progress is made. The Window algorithm keeps FIFO buffer of last K scores, and timesteps when these scores were recorded. Linear regression is performed to estimate the slope of the learning curve for each task, with the timesteps as the input variables. The regression coefficient is used as the reward in the above non-stationary bandit algorithm. For details, see Algorithm 3 in Appendix A.

3.4 Sampling algorithm

The previous algorithms require tuning of hyperparameters to balance exploration. To get rid of exploration hyperparameters, we take inspiration from Thompson sampling. The Sampling algorithm

keeps a buffer of last K rewards for each task. To choose the next task, a recent reward is sampled from each task's K-last-rewards buffer. Then whichever task yielded the highest sampled reward is chosen. This makes exploration a natural part of the algorithm: tasks that have recently had high rewards are sampled more often. For details, see Algorithm 4 in Appendix A.

4   Experiments

4.1   Decimal Number Addition

Addition of decimal numbers with LSTM is a well known task that requires a curriculum to learn in reasonable time (Zaremba and Sutskever, 2014). It is implemented as sequence-to-sequence model (Sutskever et al., 2014), where the input to the network is two decimal-coded numbers separated by a 'plus' sign, and output of the network is the sum of those numbers, also in decimal coding. The curriculum is based on the number of digits in the input numbers – it is easier to learn addition of short numbers and then move on to longer numbers.

Number addition is a supervised learning problem and therefore can be trained more efficiently by including several curriculum tasks in the mini-batch. Therefore we adopt batch training scheme as outlined in 2.2. The score we use is the accuracy of each task calculated on validation set. The results shown below are means and standard deviations of 3 runs with different random seeds. Full experiment details can be found in appendix C.

4.1.1   Addition with 1-dimensional Curriculum

We started with a similar setup to (Zaremba and Sutskever, 2014), where the curriculum task determines the maximum number of digits in both added numbers. The results are shown on Figure 3. Our algorithms outperformed uniform sampling and the best manual curriculum ("combined") for 9-digit addition from (Zaremba and Sutskever, 2014). An example of the task distribution during training session is given on figure 4.
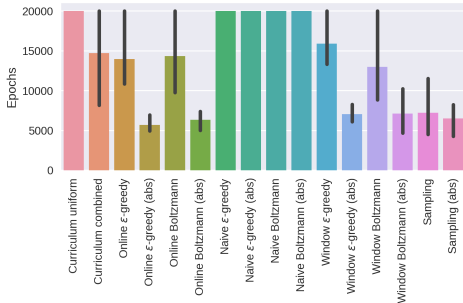

Figure 3: Results for 9-digit 1D addition, lower is better. Variants using the absolute value of the expected reward surpass the best manual curriculum ("combined").
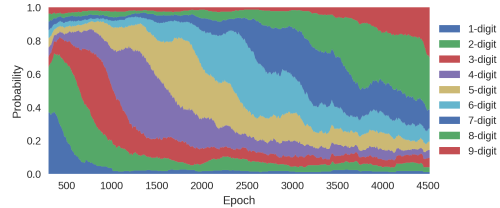
Figure 4: Progression of the task distribution over time for 9-digit 1D addition (Sampling). The algorithm progresses from simpler tasks to more complicated. Harder tasks take longer to learn and the algorithm keeps training on easier tasks to counter unlearning.

4.1.2   Addition with 2-dimensional Curriculum

We also experimented with a curriculum where the ordering of tasks is not obvious. We used the same decimal addition task, but in this case the length of each number is chosen separately, making the task-space 2-dimensional. Each training batch is modelled as a probability distribution over the length of both numbers $P(l_1, l_2)$. We also tried making this distribution independent such that $P(l_1, l_2) = P(l_1)P(l_2)$, but that did not work as well.

There is no equivalent experiment in (Zaremba and Sutskever, 2014), so we created a manual curriculum inspired by their best 1D curriculum. In particular we increase difficulty by increasing the maximum length of both two numbers, which reduces the problem to a 1D curriculum. Figure 5 shows the results for 9-digit 2D addition. Figure 6 illustrates the different approaches taken by manual and automated curriculum.
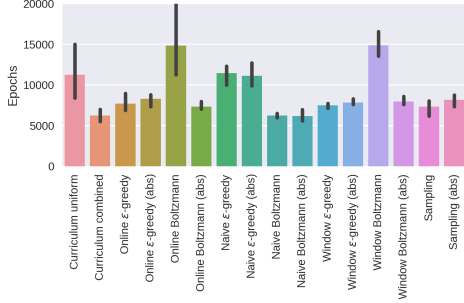
Figure 5: Results for 9-digit 2D addition, lower is better. The task seems easier, manual curriculum is hard to beat and uniform sampling is competitive.
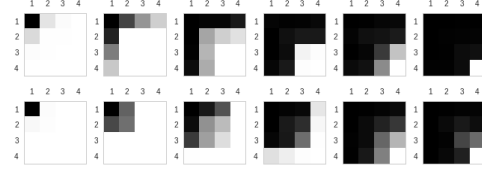

Figure 6: Accuracy progress for 4-digit 2D addition. Top: TSCL. Bottom: the best manual curriculum. Our algorithm takes distinctively different approach by training on shorter numbers first. 9-digit videos can be found `https://youtu.be/y_QIcQ6spWk` and `https://youtu.be/fB2kx-esjgw`.

4.1.3 Observations

- Using absolute value of Q boosts the performance of almost all the algorithms, which means it is efficient in countering forgetting.
- There is no universal best algorithm. For 1D the Window algorithm and for 2D the Naive algorithm performed the best. Sampling is competitive in both and has least hyperparameters.
- Whether ϵ-greedy or Boltzmann exploration works better depends on the algorithm.
- Uniform sampling is surprisingly efficient, especially in 2D case.
- The 2D task is solved faster and the manual curriculum is hard to beat in 2D.

4.2 Minecraft

Minecraft is a popular 3D video game where players can explore, craft tools and build arbitrary structures, making it a potentially rich environment for AI research. We used the Malmo platform (Johnson et al., 2016) with OpenAI Gym wrapper[5] to interact with Minecraft in our reinforcement learning experiments. In particular we used *ClassroomDecorator* from Malmo to generate random mazes for the agent to solve. The mazes contain sequences of rooms separated by the following obstacles:

- **Wall** – the agent has to locate a doorway in the wall.
- **Lava** – the agent has to cross a bridge over lava.

We only implemented the Window algorithm for the Minecraft task, because other algorithms rely on score change, which is not straightforward to calculate for parallel training scheme. As baseline we use uniform sampling, training only on the last task, and a manually tuned curriculum. Full experimental details can be found in appendix D.

4.2.1 5-step Curriculum

We created a simple curriculum with 5 steps:

1. A single room with a target.
2. Two rooms separated by lava.
3. Two rooms separated by wall.
4. Three rooms separated by lava and wall, in random order.
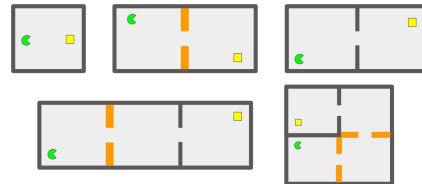5. Four rooms separated by lava and walls, in random order.

Figure 7: 5-step curriculum.

[5]`https://github.com/tambetm/gym-minecraft`

6

Refer to Figure 7 for the room layout. The starting position of the agent and the location of the target were randomized for each episode. Manual curriculum trained first task for $200\,000$ steps, second, third and fourth task for $400\,000$ steps, and fifth task for $600\,000$ steps.

Figure 8 shows learning curves for Minecraft 5-step curriculum. The mean curve and standard deviation are based on 3 runs with different random seeds.
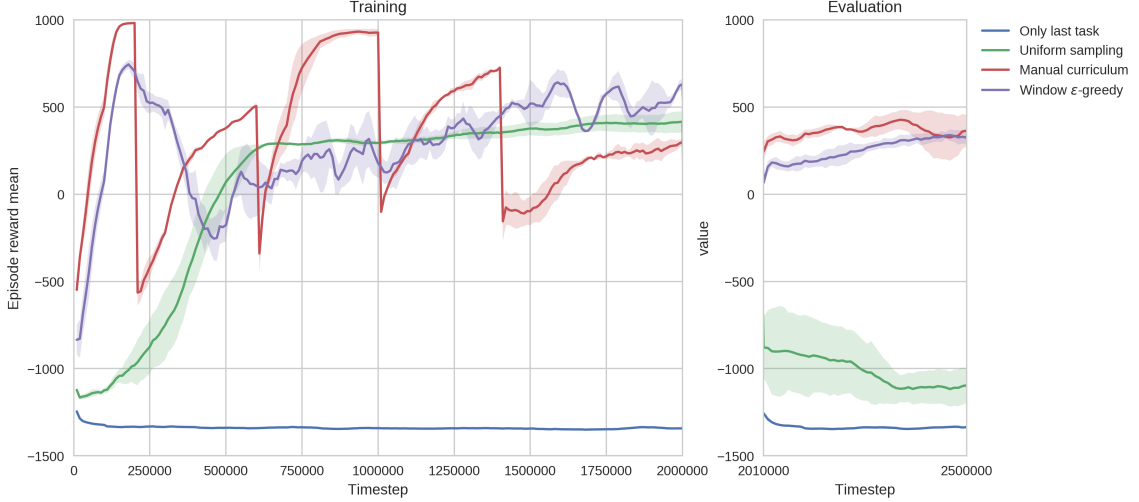

Figure 8: Minecraft 5-step curriculum results, Y-axis shows mean episode reward per $10\,000$ timesteps for the current task. Left: training performance, notice the manual curriculum task switches after $200\,000$, $600\,000$, $1\,000\,000$ and $1\,400\,000$ steps. For automatic curriculum the training score has no clear interpretation. Right: evaluation training on the last task. When training only on the last task the agent did not make any progress at all. When training on a uniform mix of the tasks the progress was slow. Manual curriculum allowed the agent to learn the last task to an acceptable level. TSCL is comparable to the manual curriculum in performance.

Video of the trained agent can be found here: `https://youtu.be/cada0d_aDIc`. The learned policy is robust to the number of rooms, given that obstacles are of the same type. The code is available at `https://github.com/tambetm/TSCL`.

5 Related Work

Work by (Bengio et al., 2009) sparked general interest in curriculum learning. More recent results include learning to execute short programs (Zaremba and Sutskever, 2014), finding shortest paths in graphs (Graves et al., 2016) and learning to play first-person shooter (Wu and Tian, 2017). All those works rely on manually designed curricula and do not attempt to produce it automatically.

The idea of using learning progress as the reward could be traced back to (Schmidhuber, 1991). It has been successfully applied in the context of developmental robotics to learn object manipulation (Oudeyer et al., 2007; Baranes and Oudeyer, 2013) and also in actual classroom settings to teach primary school students (Clement et al., 2015). Using learning progress as the reward can be linked to the concept of intrinsic motivation (Oudeyer and Kaplan, 2007; Schmidhuber, 2010).

Several algorithms for adversarial bandits were analyzed in (Auer et al., 2002). While many of those algorithms have formal worst-case guarantees, in our experiments they did not perform well. The problem is that they come with no assumptions. In curriculum learning we can assume that rewards change smoothly over time.

More recently (Sukhbaatar et al., 2017) proposed a method to generate incremental goals and therefore curricula automatically. The setup consists of two agents, Alice and Bob, where Alice is generating trajectories and Bob is trying to either repeat or reverse them. Similar work by (Held et al., 2017) uses generative adversarial network to generate goal states for an agent. Compared to TSCL, they are

able to generate new subtasks on the go, but this mainly aids in exploration and is not guaranteed to help in learning the final task. (Sharma and Ravindran, 2017) apply similar setup as ours to multi-task learning. In their work they practice more tasks that are underperforming compared to preset baseline, as opposed to our approach of using learning progress. (Jain and Tulabandhula, 2017) estimate transfer between subtasks and target task, and create curriculum based on that.

The most similar work to ours was done concurrently in (Graves et al., 2017). While the problem statement is strikingly similar, our approaches differ. They apply the automatic curriculum learning only to supervised sequence learning tasks, while we consider also reinforcement learning tasks. They use the EXP3.S algorithm for adversarial bandits, while we propose alternative algorithms inspired by non-stationary bandits. They consider other learning progress metrics based on complexity gain while we focus only on prediction gain (which performed overall best in their experiments). Moreover, their work only uses uniform sampling of tasks as a baseline, whereas ours compares the best known manual curriculum for the given tasks. In summary they arrive to very similar conclusions to ours.

Decimal addition has also been explored in (Kalchbrenner et al., 2015; Reed and De Freitas, 2015; Kaiser and Sutskever, 2015), sometimes improving results over original work in (Zaremba and Sutskever, 2014). Our goal was not to improve the addition results, but to evaluate different curriculum approaches, therefore there is no direct comparison.

Minecraft is a relatively recent addition to reinforcement learning environments. Work by (Oh et al., 2016) evaluates memory-based architectures for Minecraft. They use cognition-inspired tasks in visual grid-world. Our tasks differ in that they do not need explicit memory, and the movement is continuous, not grid-world. Another work by (Tessler et al., 2016) uses tasks similar to ours but they take different approach: they learn a Deep Skill Module for each subtask, freeze weights of those modules and train hierarchical deep reinforcement learning network to pick either single actions or subtask policies. In contrast our approach uses simple policy network and relies on the TSCL to learn (and not forget) the subtasks.

While exploration bonuses (Bellemare et al., 2016; Houthooft et al., 2016; Stadie et al., 2015) solve the same problem of sparse rewards, they apply to Student algorithms, while we were considering different Teacher approaches. For this reason we leave the comparison with exploration bonuses to future work.

6 Conclusion

We presented a framework for automatic curriculum learning that can be used for supervised and reinforcement learning tasks. We proposed a family of algorithms within that framework based on the concept of learning progress. While many of the algorithms performed equally well, it was crucial to rely on the absolute value of the slope of the learning curve when choosing the tasks. This guarantees the re-training on tasks which the network is starting to forget. In our LSTM decimal addition experiments, the Sampling algorithm outperformed the best manually designed curriculum as well as the uniform sampling. On the challenging 5-task Minecraft navigation problem, our Window algorithm matched the performance of a carefully designed manual curriculum, and significantly outperformed uniform sampling. For problems where curriculum learning is necessary, TSCL can avoid the tedium of ordering the difficulty of subtasks and hand-designing the curriculum.

7 Future Work

In this work we only considered discrete task parameterizations. In the future it would be interesting to apply the idea to continuous task parameterizations. Another promising idea to explore is the usage of automatic curriculum learning in contexts where the subtasks have not been pre-defined. For example, subtasks can be sampled from a generative model, or taken from different initial states in the same environment.

8 Acknowledgements

References

Peter Auer, Nicolò, Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The non-stochastic multi-armed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. 2016.

Adrien Baranes and Pierre-Yves Oudeyer. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73, 2013.

Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning – ICML '09*, 2009. doi: 10.1145/1553374.1553380.

François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

Benjamin Clement, Didier Roy, Pierre-Yves Oudeyer, and Manuel Lopes. Multi-armed bandits for intelligent tutoring systems. *Journal of Educational Data Mining (JEDM)*, 7(2), 2015.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, and et al. Sergio Gómez Colmenarejo. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):71–76, 2016.

Alex Graves, Marc G. Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks, 2017. http://arxiv.org/abs/1704.03003.

David Held, Xinyang Geng, Carlos Florensa, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. *arXiv preprint arXiv:1705.06366*, 2017.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

Vikas Jain and Theja Tulabandhula. Faster reinforcement learning using active simulators. *arXiv preprint arXiv:1703.07853*, 2017.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *International joint conference on artificial intelligence (IJCAI)*, page 4246, 2016.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

John Langford. Efficient exploration in reinforcement learning. In *Encyclopedia of Machine Learning*, pages 309–311. Springer, 2011.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies, 2015. http://arxiv.org/abs/1504.00702.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andy Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Belle-mare, and et al. Alex Graves. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–33, 2015.

Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016.

Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in Neurorobotics*, 1(November: 6), 2007.

Pierre-Yves Oudeyer, Frdric Kaplan, and Verena V Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE transactions on evolutionary computation*, 11(2):265–286, 2007.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Jürgen Schmidhuber. Curious model-building control systems. In *Neural Networks, 1991. 1991 IEEE International Joint Conference on*, pages 1458–1463. IEEE, 1991.

Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–47, 2010.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Sahil Sharma and Balaraman Ravindran. Online multi-task learning using biased sampling. *arXiv preprint arXiv:1702.06053*, 2017.

Bradly C Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015.

Sainbayar Sukhbaatar, Ilya Kostrikov, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks / a Publication of the IEEE Neural Networks Council*, 9(5):1054–1054, 1998.

Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. *arXiv preprint arXiv:1604.07255*, 2016.

Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *Submitted to Int'l Conference on Learning Representations*, 2017.

Wojciech Zaremba and Ilya Sutskever. Learning to execute., 2014. http://arxiv.org/abs/1410.4615.

Appendices

A   Simple versions of the algorithms

Algorithm 1 Online algorithm

Initialize STUDENT learning algorithm
Initialize expected return $Q(a) = 0$ for all N tasks
for t=1,...,T **do**
    Choose task a_t based on $|Q|$ using ϵ-greedy or Boltzmann policy
    Train STUDENT using task a_t and observe reward $r_t = x_t^{(a_t)} - x_{t'}^{(a_t)}$
    Update expected return $Q(a_t) = \alpha r_t + (1 - \alpha)Q(a_t)$
end for

Algorithm 2 Naive algorithm

Initialize STUDENT learning algorithm
Initialize expected return $Q(a) = 0$ for all N tasks
for t=1,...,T **do**
    Choose task a_t based on $|Q|$ using ϵ-greedy or Boltzmann policy
    Reset $D = \emptyset$
    for k=1,...,K **do**
        Train STUDENT using task a_t and observe score $o_t = x_t^{(a_t)}$
        Store score o_t in list D
    end for
    Apply linear regression to D and extract the coefficient as r_t
    Update expected return $Q(a_t) = \alpha r_t + (1 - \alpha)Q(a_t)$
end for

Algorithm 3 Window algorithm

Initialize STUDENT learning algorithm
Initialize FIFO buffers $D(a)$ and $E(a)$ with length K for all N tasks
Initialize expected return $Q(a) = 0$ for all N tasks
for t=1,...,T **do**
    Choose task a_t based on $|Q|$ using ϵ-greedy or Boltzmann policy
    Train STUDENT using task a_t and observe score $o_t = x_t^{(a_t)}$
    Store score o_t in $D(a_t)$ and timestep t in $E(a_t)$
    Use linear regression to predict $D(a_t)$ from $E(a_t)$ and use the coef. as r_t
    Update expected return $Q(a_t) = \alpha r_t + (1 - \alpha)Q(a_t)$
end for

Algorithm 4 Sampling algorithm

Initialize STUDENT learning algorithm
Initialize FIFO buffers $D(a)$ with length K for all N tasks
for t=1,...,T **do**
    Sample reward $\tilde{r}_a$ from $D(a)$ for each task (if $|D(a)| = 0$ then $\tilde{r}_a = 1$)
    Choose task $a_t = \text{argmax}_a |\tilde{r}_a|$
    Train STUDENT using task a_t and observe reward $r_t = x_t^{(a_t)} - x_{t'}^{(a_t)}$
    Store reward r_t in $D(a_t)$
end for

B   Batch versions of the algorithms

Algorithm 5 Online algorithm

Initialize STUDENT learning algorithm
Initialize expected return $Q(a) = 0$ for all N tasks
for t=1,...,T **do**
    Create prob. dist. $\mathbf{a_t} = (p_t^{(1)}, ..., p_t^{(N)})$ based on $|Q|$ using ϵ-greedy or Boltzmann policy
    Train STUDENT using prob. dist. $\mathbf{a_t}$ and observe scores $\mathbf{o_t} = (x_t^{(1)}, ..., x_t^{(N)})$
    Calculate score changes $\mathbf{r_t} = \mathbf{o_t} - \mathbf{o_{t-1}}$
    Update expected return $\mathbf{Q} = \alpha\mathbf{r_t} + (1 - \alpha)\mathbf{Q}$
end for

Algorithm 6 Naive algorithm

Initialize STUDENT learning algorithm
Initialize expected return $Q(a) = 0$ for all N tasks
for t=1,...,T **do**
    Create prob. dist. $\mathbf{a_t} = (p_t^{(1)}, ..., p_t^{(N)})$ based on $|Q|$ using ϵ-greedy or Boltzmann policy
    Reset $D(a) = \emptyset$ for all tasks
    for k=1,...,K **do**
        Train STUDENT using prob. dist. $\mathbf{a_t}$ and observe scores $\mathbf{o_t} = (x_t^{(1)}, ..., x_t^{(N)})$
        Store score $o_t^{(a)}$ in list $D(a)$ for each task a
    end for
    Apply linear regression to each $D(a)$ and extract the coefficients as vector $\mathbf{r_t}$
    Update expected return $\mathbf{Q} = \alpha\mathbf{r_t} + (1 - \alpha)\mathbf{Q}$
end for

Algorithm 7 Window algorithm

Initialize STUDENT learning algorithm
Initialize FIFO buffers $D(a)$ with length K for all N tasks
Initialize expected return $Q(a) = 0$ for all N tasks
for t=1,...,T **do**
    Create prob. dist. $\mathbf{a_t} = (p_t^{(1)}, ..., p_t^{(N)})$ based on $|Q|$ using ϵ-greedy or Boltzmann policy
    Train STUDENT using prob. dist. $\mathbf{a_t}$ and observe scores $\mathbf{o_t} = (x_t^{(1)}, ..., x_t^{(N)})$
    Store score $o_t^{(a)}$ in $D(a)$ for all tasks a
    Apply linear regression to each $D(a)$ and extract the coefficients as vector $\mathbf{r_t}$
    Update expected return $\mathbf{Q} = \alpha\mathbf{r_t} + (1 - \alpha)\mathbf{Q}$
end for

Algorithm 8 Sampling algorithm

Initialize STUDENT learning algorithm
Initialize FIFO buffers $D(a)$ with length K for all N tasks
for t=1,...,T **do**
    Sample reward $\tilde{r}_a$ from $D(a)$ for each task (if $|D(a)| = 0$ then $\tilde{r}_a = 1$)
    Create one-hot prob. dist. $\tilde{\mathbf{a}}_\mathbf{t} = (p_t^{(1)}, ..., p_t^{(N)})$ based on $\mathrm{argmax}_a |\tilde{r}_a|$
    Mix in uniform dist. : $\mathbf{a_t} = (1 - \epsilon)\tilde{\mathbf{a}}_\mathbf{t} + \epsilon/N$
    Train STUDENT using prob. dist. $\mathbf{a_t}$ and observe scores $\mathbf{o_t} = (x_t^{(1)}, ..., x_t^{(N)})$
    Calculate score changes $\mathbf{r_t} = \mathbf{o_t} - \mathbf{o_{t-1}}$
    Store reward $r_t^{(a)}$ in $D(a)$ for each task a
end for

C   Decimal Number Addition Training Details

Our reimplementation of decimal addition is based on Keras (Chollet et al., 2015). The encoder and decoder are both LSTMs with 128 units. In contrast to the original implementation, the hidden state is not passed from encoder to decoder, instead the last output of the encoder is provided to all inputs of the decoder. One curriculum training step consists of training on 40,960 samples. Validation set consists of 4,096 samples and 4,096 is also the batch size. Adam optimizer (Kingma and Ba, 2014) is used for training with default learning rate of 0.001. Both input and output are padded to a fixed size.

In the experiments we used the number of steps until 99% validation set accuracy is reached as a comparison metric. The exploration coefficient ϵ was fixed to 0.1, the temperature τ was fixed to 0.0004, the learning rate α was 0.1, and the window size K was 10 in all experiments.

D   Minecraft Training Details

The Minecraft task consisted of navigating through randomly generated mazes. The maze ends with a target block and the agent gets 1,000 points by touching it. Each move costs -0.1 and dying in lava or getting a timeout yields -1,000 points. Timeout is 30 seconds (1,500 steps) in the first task and 45 seconds (2,250 steps) in the subsequent tasks.

For learning we used the *proximal policy optimization* (PPO) algorithm (Schulman et al., 2017) implemented using Keras (Chollet et al., 2015) and optimized for real-time environments. The policy network used four convolutional layers and one LSTM layer. Input to the network was 40×30 color image and outputs were two Gaussian actions: move forward/backward and turn left/right. In addition the policy network had state value output, which was used as the baseline. Figure 10 shows the network architecture.
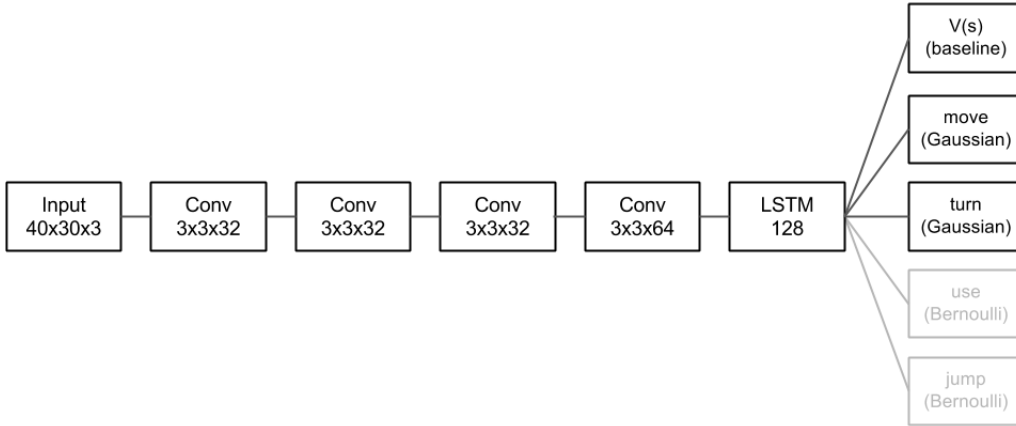

Figure 9: Network architecture used for Minecraft.

For training we used a setup with 10 parallel Minecraft instances. The agent code was separated into runners, that interact with the environment, and a trainer, that performs batch training on GPU, similar to Babaeizadeh et al. (2016). Runners regularly update their snapshot of the current policy weights, but they only perform prediction (forward pass), never training. After a fixed number of steps they use FIFO buffers to send collected states, actions and rewards to the trainer. Trainer collects those experiences from all runners, assembles them into batches and performs training. FIFO buffers shield the runners and the trainer from occasional hiccups. This also means that the trainer is not completely on-policy, but this problem is handled by the importance sampling in PPO.
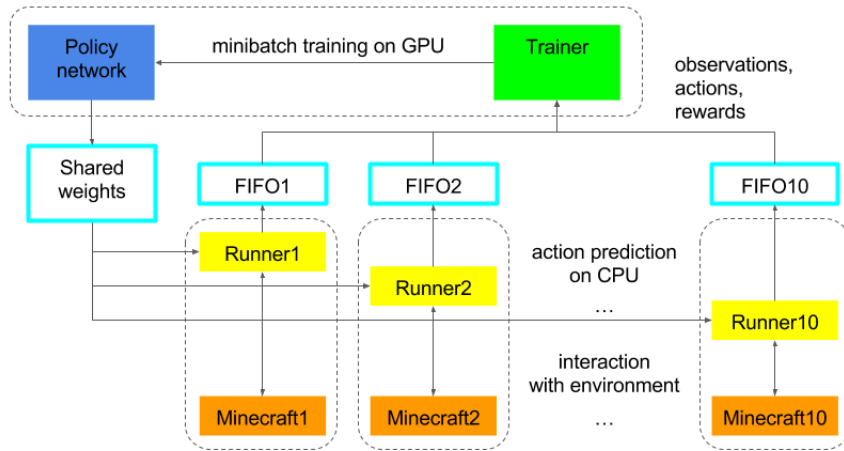
Figure 10: Training scheme used for Minecraft.

During training we also used frame skipping, i.e. processed only every 5th frame. This sped up the learning considerably and the resulting policy also worked without frame skip. Also, we used auxiliary loss for predicting the depth as suggested in (Mirowski et al., 2016). Surprisingly this resulted only in minor improvements.

For automatic curriculum learning we only implemented the Window algorithm for the Minecraft task, because other algorithms rely on score change, which is not straightforward to calculate for parallel training scheme. Window size was defined in timesteps and fixed to 10,000 in the experiments, exploration rate was set to 0.1.

The idea of the first task in the curriculum was to make the agent associate the target with a reward. In practice this task proved to be too simple - the agent could achieve almost the same reward by doing backwards circles in the room. For this reason we added penalty for moving backwards to the policy loss function. This fixed the problem in most cases, but we occasionally still had to discard some unsuccessful runs. Results only reflect the successful runs.

We also had some preliminary success combining continuous (Gaussian) actions with binary (Bernoulli) actions for "jump" and "use" controls, as shown on figure 10. This allowed the agent to learn to cope also with rooms that involve doors, switches or jumping obstacles, see `https://youtu.be/e1oKiPlAv74`.