Game Engine Induction with Deep Networks

Dave Gottlieb
Stanford University
Department of Philosophy
dmg1@stanford.edu

Abstract

A game engine is a probabilistic generative process, which produces a stream of outputs based on inputs and some hidden state. In this paper, I consider learning the transition and output functions of such a process using only input and output streams – with no access to hidden states.

I adapt two different network architectures from video classification to learn the outputs and transitions of the Pong game engine. The second architecture combines recurrency with spatial convolutions in the same layers, and I analyze the spatio-temporal receptive fields of those layers with the concept of "speed of light."

1. Introduction

I present two models to learn the output and transition functions of the Pong game engine. A game engine can be treated as a probabilistic generative process, which, at each time step, given a hidden state, q_t, and a control input, p_t, produces a screen output, y_t. Output and transition functions, f and g, define the process's behavior over time:

$$\begin{aligned} f(q_t, p_t) &= P(y_t | q_t, p_t) \\ g(q_{t-1}, q_{t-1}) &= P(q_t | q_{t-1}, p_{t-1}). \end{aligned}$$

The task is to learn this behavior while treating the game engine as a black box – without ever having access to the hidden states, Q. In practical terms, the models take as input the sequence of previous screen outputs, $Y_{<t}$, and control inputs, $P_{\leq t}$. They then use one of two convolutional network architectures to output a prediction of the next output frame in the sequence. Both models are able to solve this task well enough to continuously predict a stream of outputs, using sampled model outputs as prediction inputs.[1]

[1]In my submission, I include source code for both models as well as for the Pong game engine, and code used to generate training and test data. Trained weights for the model are available on request.

I evaluate two main models for this task. **Model 1** is a convolutional network architecture similar to early fusion video approaches, where a finite window of previous screen outputs are "stacked" in the channels of the input to a 2D convolution. **Model 1** is accurate but has the shortcoming of being in-principle incapable of learning long-distance dependencies beyond its finite window size. To account for this, I devised **Model 2**, which combines 2D convolutions with a recurrent architecture. The first layers of **Model 2** are LSTM-RCNs, recurrent units which use spatially local convolutions instead of fully-connected transformations, similar to the proposal of [1]. This allows the early stages of **Model 2** to preserve spatial and temporal locality, while also learning long-distance dependencies. **Model 2** is also accurate, although its computational demands are greater than **Model 1**'s.

In addition to reporting these results, I investigate the receptive fields of LSTM-RCN output elements, whose spatial extent increases at a fixed rate the further back in time you go. I show that this imposes a limit on the spatial velocity of motion patterns that can be learned by such a unit. By analogy to the value c in cellular automata, I call this the *speed of light*[2]. I believe this is the first time this property of recurrent convolutional units has been investigated in depth.

Although Pong in particular is a toy problem, and the narrow task of game engine induction has little practical application, there are deep similarities to important problems. For example, just as deep Q learning results for video games have practical applications in model-free reinforcement learning generally[3], game engine induction is relevant to model learning for model-*based* reinforcement learning. Another possible application area is video generation. Generating video streams with control inputs could be used to procedurally generate videos with character movements, like dancing or sports.

2. Related work

The methods I use in **Model 1** and **Model 2** both reflect approaches that have been used in video classification. The convolutional layers of **Model 1** are similar to the "early fusion" architectures for video classification described in [4]. The LSTM-RCN layers of **Model 2** are based on the GRU-RCN architecture described in [1], also designed for video classification. These layers extend the basic idea of recurrent networks using backpropagation through time (BPTT)[5], and in particular on the LSTM design presented by [6].

Little work has directly addressed the problem of predicting future behavior of a game engine. [7] takes the approach of splitting screen output into patches and using patchwise Bayesian methods to estimate output distributions over each patch. This has the advantage of reducing the complexity of the problem compared to the full joint distribution. However, their model used knowledge of the game engine and its states, rather than treating it as a black box. Furthermore, it had no capability to learn non-local dependencies among different screen patches – not relevant in Pong but relevant to game engine induction in general.

Finally, a similar approach to roughly the same problem is described by [8][2]. They are able to predict output for several Atari games in the Arcade Learning Environment[9]. They demonstrate two different encoder-decoder architectures. The first begins with 2D convolutions across stacked sequential frames, similar to my **Model 1**. The second takes in only one frame at a time, but introduces a fully-connected LSTM layer after the initial convolutions. There are two major architectural differences between their approaches and mine (to be described in full detail below):

1. Their models use a decoding network of "deconvolutions" (aka "upconvolutions") before producing output.
2. Their recurrent model uses a fully-connected recurrent layer, whereas my **Model 2** uses spatially local recurrent convolutions.

Although my models are able to produce good results without a deconvolutional decoder network, this may be partly owing to the simplicity of Pong relative to Atari games. However, I speculate that my spatially local recurrent layers are able to compensate to some extent for not having a decoder network, because they enforce spatial locality on learned temporal dependencies. In prototypes for this project, I trained fully connected LSTM models on top of spatial convolutions, just like [8], and they were not really able to reconstruct the spatial structure of the Pong frames. Adding a deconvolutional decoder network perhaps could have solved this problem, but in my experiments it was sufficient to use spatially local recurrent layers instead.

[2]Unfortunately, I only learned about this paper during the poster session this week, limiting my ability to incorporate its work into my project.

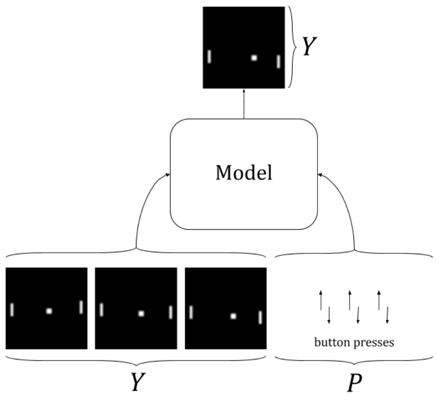

Figure 1. Overall pipeline for both models.

3. Methods

Per above, the Pong game engine behavior is characterized by output and transition functions, f and g respectively.

$$\begin{aligned} f(q_t, p_t) &= P(y_t|q_t, p_t) \\ g(q_{t-1}, q_{t-1}) &= P(q_t|q_{t-1}, p_{t-1}). \end{aligned}$$

Since our models never have access to the internal state q, they estimate f and g conjointly:

$$(\hat{f} \circ \hat{g})(Y_{<t}, P_{\leq t}) \approx P(y_t|Y_{<t}, P_{\leq t})$$

Both models are characterized by training on sequences of output frames and control inputs, with prediction targets of subsequent output frames (Figure 1).

Frames produced by the Pong game engine are $32 \times 32 \times 1$ bit, so reconstructing them can be characterized as finding "on" or "off" values for each of 32×32 pixels. Accordingly, predicted output frames are $32 \times 32 \times 1$ pixel arrays. Both models use a softmax output function to produce a probability estimate $\in (0, 1)$ for whether each pixel is active. Performance is then evaluated using a weighted average pixel-wise crossentropy loss,

$$L = -\frac{1}{32 \times 32} \sum_i [\gamma y_{i_j} (\log y_i) + (1 - y_{i_j} (\log(1 - y_i))],$$

where i indexes the pixels of the output array, y_i is the predicted value for pixel i, y_{i_j} is the true value for pixel i, and γ is a weight factor used to weight the relative importance of "on" *versus* "off" pixels ($\gamma = 14$ in the results below). The models are trained by backpropagating this loss to the parameters.

Both models are trained with minibatch gradient descent using the Adam algorithm[10].

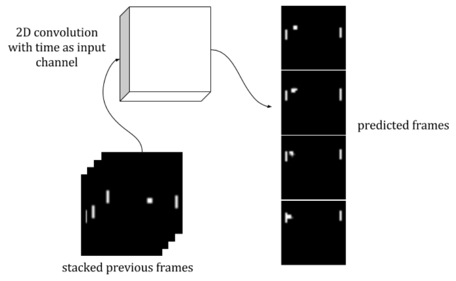

Figure 2. Core **Model 1** architecture with some predicted frame outputs.

Both models are implemented in Python using the Numpy[11] and Theano[12] frameworks and trained on NVIDIA GRID K520 GPUs.

3.1. Model 1

Model 1 is an early fusion video model. Input is 4 consecutive frames of Pong screen output, $\{y_i\}_{i=-4,\ldots,-1}$, and one frame of control inputs, p_{-1}. These are then used to predict the next frame of Pong screen output, y_0. The input frames are combined into a $32 \times 32 \times 4$ stack that then forms the input to a 2D convolution layer (the core architecture is depicted in Figure 2). (All convolutional filters are 3×3.) ReLU activation is then applied.

The convolved inputs are then passed through a second convolutional layer, followed by another ReLU activation, and then 2×2 max pooling, for robustness to degraded inputs. The output of the pooling layer is a $16 \times 16 \times 16$ filters spatially local activation volume. This volume is then *unrolled* into a $(16 \times 16 \times 16) = 4096$-dimensional feature vector, which is concatenated with the 2-dimensional vector of control inputs, p_{-1}. This concatenated feature vector forms the input to a 512-unit fully connected hidden layer, using the ReLU activation function. Per-pixel raw scores for the output frame are then calculated by a final fully connected transformation with $(32 \times 32) = 1024$ output values. Finally, an element-wise sigmoid (logistic) transformation is applied to give the "on"-"off" probabilities for each pixel.

3.2. Model 2

Model 2 is a recurrent convolutional network featuring "long short-term memory recurrent convolutional network" (LSTM-RCN) units in the early stages of processing, similar to the "gated recurrent unit recurrent convolutional network" (GRU-RCN) units described in [1]. After the LSTM-RCN layers process screen input, 2×2 max pooling is applied. Activations are then unrolled and concatenated with control inputs. The resulting vector is then fed into a fully-connected LSTM (not LSTM-RCN) unit. Per-pixel raw scores for the output frame are then calculated by a final fully connected transformation with $(32 \times 32) = 1024$ output values, and "on"-"off" probabilities are obtained by applying an elementwise sigmoid transform.

Table 1. Architectural details for **Model 1**.

Layer	Activation volume
Stacked screen input	$32 \times 32 \times 4$
CONV1	$32 \times 32 \times 8$ filters
RELU	
CONV2	$32 \times 32 \times 16$ filters
RELU	
POOL (2×2)	$16 \times 16 \times 16$ filters
UNROLL and CONCATENATE	$4096 + 2$ control inputs
FC1	512 hidden units
FC2	1024 raw scores
SOFTMAX	1024 "on-off" probabilities

During training, **Model 2** processes an input window of 8 consecutive frames at a time, maintaining hidden and cell states at each LSTM or LSTM-RCN layer. At each time step, the model emits a prediction for the next frame. All 8 such predicted frames are then compared to the true output frames (which are the same as the input sequence, but shifted one time step ahead). Using the BPTT algorithm, the loss at each time step is used to update the parameters at all previous time steps. Using this training schedule, the model is able to learn any temporal dependencies so long as they occur within an 8-frame window in the training data. In online prediction, the model can handle long-term dependencies of indefinite length, but the dependencies must be seen in the training data within the 8-frame window[3].

Below, I describe the new LSTM-RCN units in some detail, with particular attention to the extent in space and time of their receptive fields.

3.2.1 LCTM-RCNs

LSTM-RCNs are recurrent units, similar to simple recurrent neural network (RNN) units, that maintain an internal state across time-steps, while processing time series inputs. They

[3]By analogy, a text-prediction LSTM can learn the rule that "(" must be followed by ")" no matter how many characters there are in between – a dependency of indefinite length. But in order to learn that rule, its training data must include "(" followed by ")" within its finite window for BPTT.

Table 2. Architectural details for **Model 2**.

Layer	Activation volume
Screen input	$32 \times 32 \times 1 \times 8$ time-steps
LSTM-RCN1	$32 \times 32 \times 4$ filters$\times 8$ time-steps, hidden state $32 \times 32 \times 4$ filters$\times 8$ time-steps, cell state
LSTM-RCN2	$32 \times 32 \times 8$ filters$\times 8$ time-steps, hidden state $32 \times 32 \times 8$ filters$\times 8$ time-steps, cell state
POOL (2×2)	$16 \times 16 \times 8$ filters$\times 8$ time-steps
UNROLL and CONCATENATE	$(2048 + 2$ control inputs$) \times 8$ time-steps
LSTM	512 hidden units$\times 8$ time-steps
FC	1024 raw scores$\times 8$ time-steps
SOFTMAX	1024 "on-off" probabilities$\times 8$ time-steps

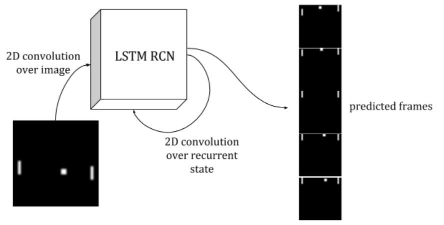
2D convolution over image
LSTM RCN
2D convolution over recurrent state
predicted frames

Figure 3. Core **Model 2** architecture with some predicted frame outputs.

do this by, at each time step, incorporating a "hidden state" based on previous time steps of processing. In this way, at each time step, the unit is not only emitting outputs, but transmitting hidden state information to future time steps. Thus, at each time step, the unit potentially incorporates information from all previous time steps. This architecture has proven effective at learning complex relationships over long temporal distances in time series data, for example syntax and semantics of text and source code[13].

LSTMs are a variation on the general recurrent architecture which introduce a "cell state" vector in addition to the hidden state. This innovation allows the model to specialize, training one set of parameters to pass information to future time steps and another set to pass information forward in the network (at a given time step). LSTMs are known to perform better than plain RNNs.

LSTM-RCNs introduce an additional variation to the basic LSTM architecture. An LSTM-RCN transforms inputs at each time step, and combines them with previous time-step hidden and cell states to produce new hidden and cell states. The difference from basic LSTMs is that each of these transformations is a spatially local 2D convolution rather than a general ("fully connected") linear transformation. Concretely, the computations performed by the LSTM-RCN are as follows:

$$
\begin{aligned}
a_i &= X_t * W_{xi} + b_{xi} \\
a_f &= X_t * W_{xf} + b_{xf} \\
a_o &= X_t * W_{xo} + b_{xo} \\
a_g &= X_t * W_{xg} + b_{xg} \\
C_t &= \sigma(a_f) \cdot C_{t-1} + \sigma(a_i) \cdot \tanh(a_g) \\
H_t &= \sigma(a_o) \cdot \tanh(C_t)
\end{aligned}
$$

where $*$ stands for the 2D convolution operator, $\cdot$ stands for elementwise multiplication, X_t is the input volume at time t, C_t is the cell state, H_t is the hidden state, and $W_{xi}, W_{xf}, W_{xo}, W_{xg}, b_{xi}, b_{xf}, b_{xo}$ and b_{xg} are learned weight and bias parameters. Since the input X_t is an image, and C_t and H_t are computed by transformations that preserve spatial locality, C_t and H_t are activation maps over the spatial structure of the input image X_t, along with the inputs at previous time steps.

Accordingly, the LSTM-RCN architecture should allow units to pass information through time in a way that is also spatially local[4]. In **Model 2**, we exploit this by feeding one frame of Pong screen output in at each time step. As each frame is fed in, the internal states of the LSTM-RCN units update to reflect information from that frame, as well as all previous frames. Over time, the model should learn transformations to preserve all previous-frame information that is relevant to future outputs, and incorporate that into its predictions.

3.2.2 LSTM-RCN receptive fields and the speed of light

Just as the output elements of 2D convolutions have receptive fields that can be characterized in terms of space, LSTM-RCN output elements have receptive fields that can be characterized in terms of space *and time*. As we will see, this has the effect of imposing a *speed of light* or speed limit on the spatial motions that can be learned by the LSTM-RCN.

In an ordinary 2D convolution, any cell of its output volume is computed from a spatially contiguous region of its input volume, called its *receptive field*. In the 3×3 convolutions used in both **Models 1 & 2**, a single cell of output volume has a receptive field of 3×3 input volume cells. In other words, a given output volume cell depends *only* on the 3×3 input cells in its receptive field – it can't incorporate any information from elsewhere in the input volume.

[4]A prototype architecture using basic LSTMs instead of LSTM-RCNs succeeded in learning some dependencies across time, but failed to reconstruct screen outputs that respected spatial locality. This may be attributable to the lack of spatial locality in the fully connected LSTM architecture.

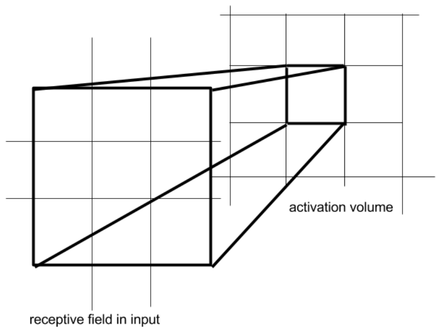

Figure 4. Receptive field for an ordinary 2D convolution. After a 3×3 convolution, one cell in the activation volume is computed from a 3×3 region called its *receptive field*.

Stacking 2D convolutions has the effect of giving later convolution output cells larger receptive fields in the original input volume. For example, if (as in **Model 1**) there are two stacked 3×3 convolutions, then:

1. the first output volume cells have 3×3 receptive fields in the input volume,
2. the second output volume cells have 3×3 receptive fields in the first output volume, and
3. accordingly, the second output volume cells have 5×5 receptive fields in the original input volume,

because they draw from the receptive fields in the input volume of all of the first output volume cells in *their own* receptive fields.

Now extend this thought to the case of LSTM-RCNs, which can be thought of as convolutional layers *stacked in time*. An LSTM-RCN hidden state, H_0 is created by adding together a convolution over an input image, X_0 with convolutions over the previous hidden state and cell state, H_{-1} and C_{-1}. Accordingly, a cell in H_0 has a receptive field of 3×3 in the input image X_0, and a receptive field of 3×3 in the previous hidden state and cell state, H_{-1} and C_{-1}. By a similar argument, a cell in H_{-1} or C_{-1} has a receptive field of 3×3 in the previous input image, X_{-1}, and receptive fields of 3×3 in the further previous hidden state and cell state, H_{-2} or C_{-2}.

In other words, a cell in the hidden unit has larger spatial receptive fields in input images further back in time, with the receptive field expanding by one cell in each spatial direction for every time step into the past.

$$\text{receptive field of } H_0 \text{ cells in } X_{-t} = (3 + 2t) \times (3 + 2t)$$

The factor 2 in the $2t$ term can be thought of as a measure of how quickly information propagates between spatially separated image regions – in fact, a maximum. In particular, it stands for the fact that the spatial receptive fields

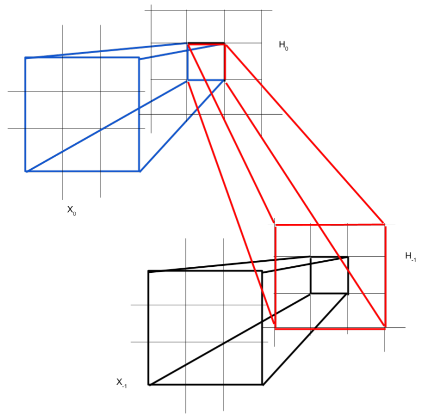

Figure 5. Receptive field for an RCN convolution. Note that a cell in the hidden state at time 0 has a 3×3 receptive field in the input image *and* a 3×3 receptive field in the hidden state at time -1, giving it a receptive field of 5×5 in the input image at time -1.

expand by 1 in each direction in each time step. An event that occurs in input image cell $(0, 0)$ at time 0 will not be able to influence hidden unit cell $(8, 8)$ until time 7.

The speed limit on the propagation of information can be thought of as a *speed of light*. The receptive fields of a given H_0 cell, extending back in time to each X_{-t}, can then be thought of as that cell's *past light cone*.

I will say that a recurrent convolutional unit has speed of light equal to the *rate at which its receptive fields expand in each direction for each time step in the past*. In general, a LSTM-RCN (or other recurrent convolutional layer) will have a speed of light equal to $1/2$ its filter size (round down) times its stride in the input image. As I've shown, the first LSTM-RCN layer has a speed of light equal to 1. The second LSTM-RCN layer is stacked on top of the first. This has the effect of increasing both its spatial and temporal extent. A cell in the second LSTM-RCN layer at time 0 has a receptive field of 5×5 in the input image at time 0, as in the case of stacked 2D convolutions. It has a receptive field of 9×9 in the input image at time -1. Accordingly, at the second LSTM-RCN layer, the speed of light is 2.

A low speed of light corresponds to a tighter constraint on the model, and a higher degree of parameter sharing over time within a given spatial region. It should be easier to train models with a low speed of light, and easier to avoid overfitting. On the other hand, the speed of light puts a limit on the model's expressive power to learn short temporal dependencies over long distances. If the Pong ball could move 10 pixels in a single frame, that would be above the speed of light for a single-layer LSTM-RCN, and it would be impossible for such a model to extrapolate its motion. An LSTM-RCN can only learn patterns that are spatio-temporally local, and the speed of light summarizes what counts as local in the model.

In Pong, the ball and paddles are strictly limited in their motion, to a maximum of about 2 pixels per frame, which is the speed of light for the second LSTM-RCN layer of **Model 2**. If our target data was instead something in which there could be much larger spatial gaps between frames, then we might need to use a model with a higher speed of light (although the fully connected layers after the LSTM-RCN layers are able to learn relationships that are not spatio-temporally local). However, motion of 2 pixels per frame is pretty high for most sources of video, so a speed of light of 2 might be sufficient for many applications.

4. Dataset

For training, I created about 1MM frames of random Pong data for each model, broken up into sequences of length appropriate to the model. Each training sequence is created as follows: initialize the ball and paddles to a random position, and initialize the ball to a random velocity. Then, run the game engine for the desired sequence length, storing each frame produced. Control input streams are generated using a simple AI algorithm, whereby each paddle tries to align its height with the height of the ball.

For **Model 1**, I generated sequences of length 5. The first 4 frames are used as input to the model, and frame 5 is the prediction target. The control input at frame 4 is also saved for model input.

For **Model 2**, I generated sequences of length 9. The first 8 frames are used as input to the model, and the offset time series of frames 2-9 is used as prediction targets. Control inputs at frames 1-8 are also saved for model input.

Training on a large number of short, randomly initialized sequences ensures that a wide variety of ball and paddle positions are sampled. In addition, it compensates for a weakness in the paddle AI. Since both paddles are trying to align their height with that of the ball, after running for a dozen frames or so the paddles will be at the same height and moving in unison. This makes the data ambiguous as to which control inputs are associated with the movement of which paddle – both input streams and both paddle movements are identical. Randomly initializing paddle position, and not giving them time to get in sync, breaks this symmetry.

5. Results and discussion

After training for 10 epochs on about 1MM frames of training data, the models show training loss of 7×10^{-4} (**Model 1**) and 5.0×10^{-3} (**Model 2**). Recall that this is a weighted average of pixelwise crossentropy loss in the output frames, and is bounded below by the plain average pixelwise crossentropy. So these figures represent a high degree of accuracy. Using an additional 10,000 frames of set-aside data generated for testing, the models show loss of 4.1×10^{-3} (**Model 1**) and 6.5×10^{-3} (**Model 2**).

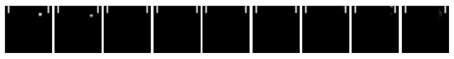

Figure 6. Pathological **Model 2** outputs. The ball flickers, disappears, and then comes back as a cloud of points.

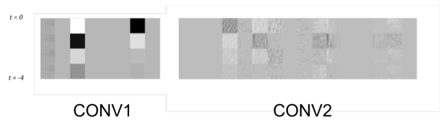

Figure 7. Activation maps of filters from **Model 1**, first two layers. The filters are applied across a temporal window of 4 frames, from the present at the top to $t-4$ at the bottom. The temporal gradation acts as a *motion detector*.

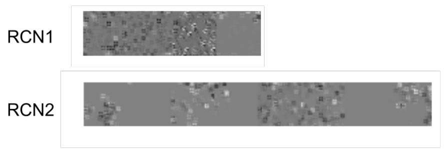

Figure 8. Activation maps of filters from **Model 2**, first two layers. These filters are spatially local as well as temporally recurrent. Most of the filters appear to be devoted to ball motion.

The accuracy promised by those loss numbers can be verified by visually inspecting some predicted outputs: the overwhelming majority of frames in the training *and* test sets are predicted almost completely accurately. Some examples can be seen in Figures 2 and 3, above. In general, all of the following are predicted with complete fidelity:

1. Straight-line motion of the ball.
2. Bounces of the ball on the top or bottom of the screen.
3. Motion of the paddles in response to control inputs.

However, running the models online with predicted outputs fed back into input, errors are quickly encountered. See Figure 6 for some examples. I discuss diagnoses and potential fixes for these problems in the next section.

To understand how the models *see* the Pong frames, I have visualized the activation maps of the earlier convolutional layers of both models. I use the technique of [**???**] to find the most-preferred input image for each convolutional filter. The results show us something about how time and motion is represented in the network – see Figures 7 and 8.

6. Potential improvements

Failures of the model can mostly be attributed to:

1. Difficult-to-predict sequences are rare in training data.
2. Errors accumulate when the models are run online, feeding their output frames back into input.
3. Overfitting.

All three problems can be readily addressed.

Difficult-to-predict sequences. The models learn straight line motion very well. More difficult for them are (1) bounces against the paddles, and (2) resetting the ball to the center of the screen after it goes off the edge. Both of these scenarios are rare in the training data, because the ball usually starts in the middle of the field and only travels a few frames (most randomly chosen locations are in the middle of the field).

This could be easily fixed by biasing the ball's starting positions towards the edges a little, and sampling training data from longer sequences.

Accumulating errors. Since the model is trained on completely clean true Pong outputs, when it is run on its own somewhat noisy outputs, it's not sure what to do, and the outputs get noisier with time until the signal is degraded.

This could be fixed by gradually introducing predicted frames into sequences during training[14].

Overfitting. Model 1 in particular might benefit from dropout.

More generally, a direction for further research might be applying the models to a larger class of games. More complex game engines might motivate introducing more complex models, for example by adding decoder networks, or a variational autoencoder architecture[15].

7. Conclusion

I have demonstrated two architectures for game engine induction, the problem of learning an underlying game engine process using only streams of inputs and outputs. Although the current models are not as accurate as they could be, they should be a convincing proof of concept and, I have argued, improved performance should now be easy to achieve.

Furthermore, I have contributed to the theory of recurrent convolutional units like the LSTM-RCNs of **Model 2**, devising the concept of a model's speed of light. Speed of light quantifies a model's spatio-temporal locality. Speed of light is also a measure of the expressive power of a spatio-temporally local model – models with higher speed of light are able to learn a wider class of relationships, which brings with it both strengths and weaknesses.

References

1. Ballas, N., L. Yao, C. Pal, and A. Courville. 2016. Delving Deeper into Convolutional Networks for Learning Video Representations. *ICLR*.

2. Gardner, M. 1970. Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American* 223: 120–123.

3. Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing atari with deep reinforcement learning. *CoRR* abs/1312.5602.

4. Karpathy, Andrej, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the iEEE conference on computer vision and pattern recognition*, 1725–1732.

5. Mozer, Michael C. 1989. A focused back-propagation algorithm for temporal pattern recognition. *Complex systems* 3: 349–381.

6. Hochreiter, Sepp, and Jrgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9. MIT Press: 1735–1780.

7. Bellemare, Marc, Joel Veness, and Michael Bowling. 2013. Bayesian learning of recursively factored environments. In *Proceedings of the 30th international conference on machine learning (iCML-13)*, ed. Sanjoy Dasgupta and David Mcallester, 28:1211–1219. 3. JMLR Workshop; Conference Proceedings.

8. Oh, Junhyuk, Xiaoxiao Guo, Honglak Lee, Richard L. Lewis, and Satinder P. Singh. 2015. Action-conditional video prediction using deep networks in atari games. *CoRR* abs/1507.08750.

9. Bellemare, Marc G., Yavar Naddaf, Joel Veness, and Michael Bowling. 2012. The arcade learning environment: An evaluation platform for general agents. *CoRR* abs/1207.4708.

10. Kingma, Diederik P., and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980.

11. Walt, S. van der, S. C. Colbert, and G. Varoquaux. 2011. The numPy array: A structure for efficient numerical computation. *Computing in Science Engineering* 13: 22–30. doi:10.1109/MCSE.2011.37.

12. Bergstra, James, Olivier Breuleux, Frdric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math expression compiler. In *Proceedings of the python for scientific computing conference (SciPy)*. Austin, TX.

13. Karpathy, Andrej, Justin Johnson, and Fei-Fei Li. 2015. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*.

14. Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for se-

quence prediction with recurrent neural networks. *CoRR* abs/1506.03099.

15. Kingma, D. P, and M. Welling. 2013. Auto-Encoding Variational Bayes. *ArXiv e-prints*.