Early Steps in the Creation of a Predictive Autoencoder

Zachary Kirby

*zjk2775@louisiana.edu*

**Abstract**-- Convolutional neural networks have proven to be one of the best methods of image classification, pattern recognition, etc. It has also been proven that adding elements of a long short-term memory (LSTM) model into a convolutional neural network dramatically improves performance by giving the network access to previous states of information. Our research lab set out to create our own convolutional LSTM (convLSTM) model using lower level TensorFlow API implemented in Python. The convLSTM model requires no supervision during training, but has the power to predict movement in procedurally generated animations. While we train the network on basic data, it has inherent scalability into more complicated datasets to be implemented in the future. The model also utilizes TensorBoard, a visualization module of TensorFlow that can create visuals of the training weights, output images, and the computational graph.

—————————— ◆ ——————————

**1 Introduction**

Inside the ever-growing field of machine learning exists a subclass known as deep learning. Deep learning involves the creation of dense neural networks that carry out complex mathematical functions. This makes deep learning very useful in tasks such as computer vision, pattern recognition, natural language processing, etc., as implicit programming becomes difficult. These deep networks exponentially increase their effectiveness through repeated training and validation. This style of networks allows for increasing complexity of problem solving. While this field is rather new, much progress has been made in practical usage as most of society interacts with one form or another of deep learning. Google’s translation algorithm has vastly improved over time with the addition of deep learning. Spotify recommends new music catered to each individual user because of a deep learning network that learns music preference based on user data. Deep learning is becoming ubiquitous in present society.

Among the countless applications of deep learning neural networks, computer vision has undoubted potential. We mainly focused on computer vision’s ability to predict movement in video feed. The human image-processing system in the brain involves a computational system like that of computer vision. Human brains can predict movement with incredible accuracy, and anything that supersedes our processing becomes recognized as motion blur. A mechanism exists inside the brain to take our visual input, process it, and produce real time prediction. A deep learning predictive model could offer insight into how our brain understands the physical world around us. This model would not be a complete vision process as it would not be able to label data and predict at the same time. A much more sophisticated model is needed to achieve such results.

Our research lab set out to create a long short-term memory (LSTM) model that uses convolution instead of basic matrix multiplication to process image data. By using an LSTM, the model will be able to maintain memory of its previous state, improving with each training iteration as a human brain would. This allows for much more accuracy than a traditional convolutional neural network, which is the closest implementation to human-like visual processing we can model on a computer. Our model also allows for several comparisons to be drawn between itself and a natural neural system. Further implementation of this model could be altered to account for more biological architecture in the future.

**2 Training Data**

A robust model is nothing without proper training data. A learning machine needs something to learn from, whether it's training is supervised or unsupervised [4]. In this model, we used unsupervised learning. No specific coding is done for any given input as long as the data can be normalized to fit against the training weights. This allows for a simple convLSTM network to become inherently scalable as more data becomes available. Data can be as simple as a still image, or a complex precipitation map [2]. For our purpose, however, we start with data that does not require complex learning. We train this model using singular images and simple animations.

Animations for training may be outsourced from a private partner or a public dataset, but the most power comes from developing specific training data independently. This gives control over as many variables as possible without the risk of an external bug causing error. Errors in custom created data can be dealt with as soon as the bug is found and potentially save valuable time. The data can be made to any specific constraints needed and any single aspect of creation can be edited to better debug early training models.

**2.1 Blender Automation**

This is where the open-source animation software, Blender, becomes extremely useful. While the professional grade modeling and animation software is great for expressions of creativity, it has very capable Python scripting support. The combination of Python scripting and Blender’s animation suite proves how useful it can be in data science. Scripts can be made to automate the production of as many training animations as needed. These scripts also hold many constraints on the animations created, which can be controlled by a random number generator to ensure each animation has unique qualities for the training model to learn. Every aspect of the training data is within the researchers’ control, allowing for testing data to be made specifically to test different features of the convLSTM’s memory.

For this model, we created a Blender automation script to generate one dimensional movement of a black circle on a white plane. The starting location is randomized in the left half of the plane and the circle moves across at a constant velocity until it reaches the edge. This animation only tests the network in its ability to predict horizontal movement. By using this single movement, we can study specifically how the network learns the features of animation. We created 4000 training animations, 500 testing animations, and 500 validation animations. We have yet to utilize all 4000 animations as the run time becomes exponentially lengthy. Training only 20 animations takes hours of runtime, but this could simply be an optimization issue.

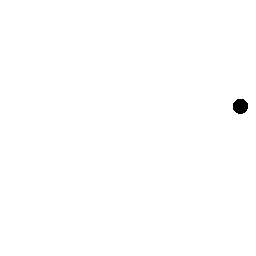
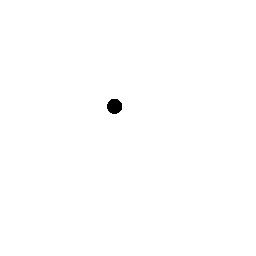


Figure 1: An example of the procedurally generated animation. A random start location was given (left) and the image proceeds at a constant velocity to the right side of the image (right). These animations are generated as frames and then are loaded into three-dimensional arrays to be used as input into the network.

In the case of still images, the only concern is normalization of the image data. We implemented framework to take any input image and do the following: resize the image to fit against the training weights, convert the image to grayscale, and then convert the image data into a usable 4-D tensor using TensorFlow. The model has enough feature maps to accommodate simple grayscale images. Processing images through the network proves that learning occurs as the model’s loss decreases, and simple features are remembered such as edge detection. Training on singular images defeats the purpose of our developed predictive autoencoder, but is a necessary tool in debugging. The framework is powerful enough to process video feed, but must first be able to learn from a still image.

**3 Building the Model**

We chose to avoid higher level implementation of our convLSTM model in TensorFlow. This path of lower level approach ensures a thorough understanding of the core mechanisms running the network. The entire model can be represented by couple of lines of code using *tf.contrib.keras.layers.ConvLSTM2D*. This keeps the network’s “black box” as small as possible. Creating a custom-built model allows for change of the model to fit any design choice. The model can be refactored to support multiple kinds of data without having to remove any pre-constructed methods.

**3.1 The Math**

Before any implementation was made, the supporting math architecture was the first obstacle. The model was first developed based on LSTM architecture as defined in *Deep Learning* (Goodfellow et al., 2016)**.**

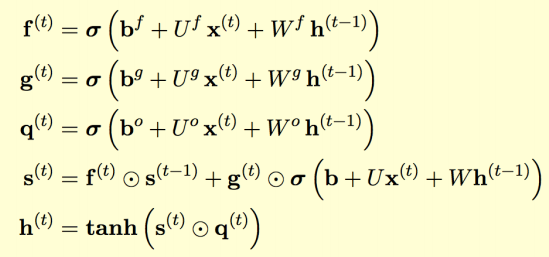
****

Figure 2: Condensed form of the Goodfellow LSTM model, edited by Maida [1]. Shown are the different states of an LSTM, the **forget gate**, f; the **input gate**, g; the **output gate**, q; the current **state**, s; and the **output**, h. These gates work together in the decision-making of what information to commit to memory as well as what information to discard.

The condensed equation block in Figure 2 represents the foundation of building any LSTM model. We take our **input**, x, in the form of four-dimensional tensor that can be processed by TensorFlow’s convolution framework. This input is then multiplied by the **feedforward weights**, U; added to the **bias**, b; and then added to the solution of the **recurrent weights**, W; and the **previous output**, h(t-1).

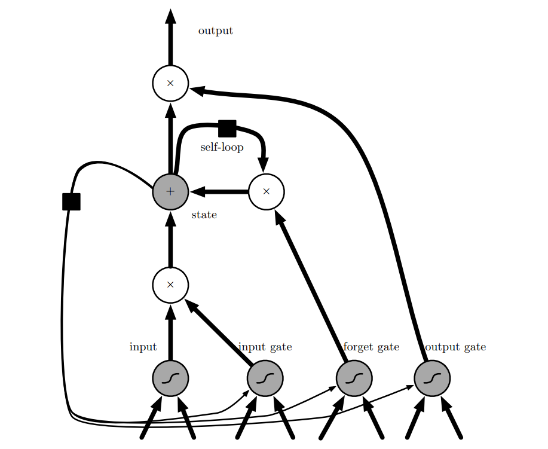
******

Figure 3: Goodfellow’s graph representation of an LSTM module [1]. Our model bares similar with this structure, except for the lack of convolutional operators on the network inputs. This LSTM module is then passed into the predictive autoencoder’s framework to provide prediction based on the given input and error.

While this model is sufficient for many kinds of computation, our model replaces every instance of matrix multiplication with a convolutional operator. With the addition of convolution, the network can process current and previous input, previous states, and error outputs all with convolutional filtering. For the simplicity of the model, we use 5x5 kernel convolution. This allows for enough information to be learned from the training data to produce meaningful results.

**3.2 The Prototype**

Now that the math is in place, we move on to the prototype design of the predictive autoencoder. Adding input from the error module yields much better performance as the network learns from its own error. Figure 4 shows the lowest layer of the network’s design without the autoencoder implementation. It takes input formatted into a normalized grayscale matrix, passes it through a predictive-error module, and returns the result as input to the convLSTM.

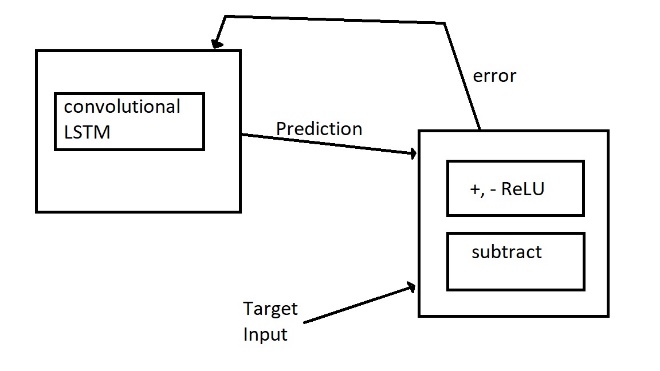


Figure 4: The lowest layer of the predictive coding network without auto-encoding. Shown are two modules to be used in the predictive network, the convLSTM module and the predictive-error module. Higher level implementation of this model can be a series of these chains connected with input being passed into the convLSTM alongside the error.

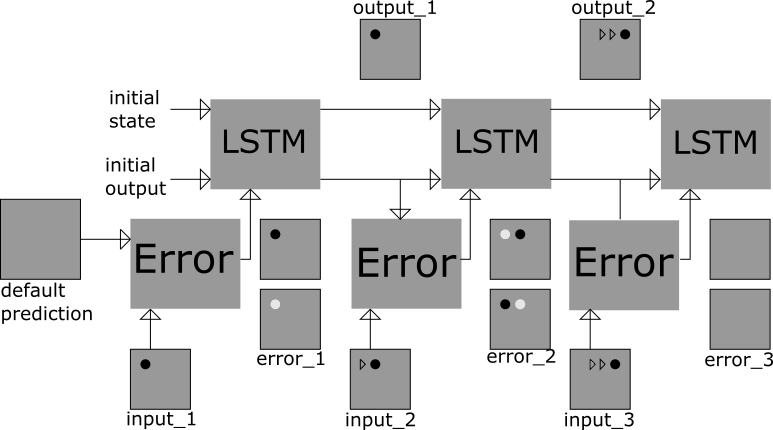


Figure 5: The Predictive Autoencoder model displaying sample input and output. This diagram shows how the network receives animations as input, processes them, and trains itself. Represented here is three LSTM unrollings, the typical configuration for movement prediction.

**4 Experimentation**

After finalizing the architecture of the predictive model, it is ready for experimentation. We test the model by feeding concurrent frames of a single animation, and then scaling into multiple animations trained against the same weights. For a single animation in training, a single frame is passed in through the first LSTM module such as shown in Figure 5. The next two frames are then passed into the second and third modules, respectively. Then, the network is trained against these three inputs for a given amount of training steps. This allows the network to learn the difference in the three separate frames and notice the horizontal movement of the dot. With the next timestep, the model is fed the next three frames and trained again, maintaining the same training weights throughout. With one animation, the model could learn the importance of the positive horizontal movement, as displayed by the training weights (see Figure 9).

One single animation is relatively easy for the network to process. The only feature the network truly learns is that the next frame of input will be slightly more to the right than the previous frame. These results are indicative of the basic network potential; however, we are interested in the larger scale application of movement prediction. With the variability of randomly generated animations introduced into the network, it must learn to locate the black dot, and then also learn to move it horizontally (while not every animation shares the same x-axis of movement). Figure 6 shows that as more animations are passed into the network, the predictions slowly improve.

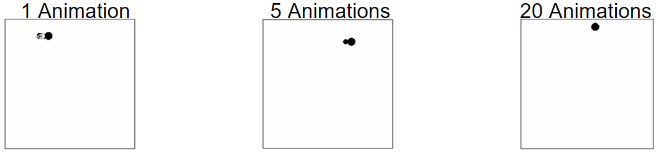
****

Figure 6: Third frame prediction after training on multiple animations. Similar to the ouput\_3 in Figure 5. The more animations the network can learn from, the better the predictions become.

**5 Image Debugging and Improvement**

During the creation of the convLSTM model, the most common debugging method was training the model from a single normalized image. While the network cannot learn anything complex from a still image, it can still learn to recreate the image through training. The closer the network’s loss was to zero, the more identical the convLSTM’s output resembled the input image.While the model was fully functional with its first creation, it was far from optimized. Initially, even training on an image produced a large loss. Several methods were either implemented or removed to improve network performance regarding minimizing loss.

***5.1 Exponentially Decreasing Learning Rate***

The first improvements were seen in the addition of an exponentially decreasing learning rate over time. Differing from a static learning rate, one that exponentially decays allows the function to more precisely adjust the weights with a smaller learning rate than was given in the beginning. This adjustment prevents any singular weight from getting stuck at a local minimum, which may not necessarily be the global minima. This would mean we lose potential for lower loss because of a constant high learning rate. After experimentation, a high learning rate (i.e. 10) decreasing every several hundred training steps proved to be most effective.

**5.2 Gradient Clipping**

Secondary improvements came from reconfiguring our original optimizer block. Our model utilizes a gradient descent optimizer. This optimizer itself could be changed entirely, but for our purpose we sought to keep it unaltered. The only difference inside the model is the addition of a gradient clipping function, known as *tf.clip\_by\_global\_norm()*. This single line of code serves to handle both vanishing and exploding gradients.

**5.3 Activation Functions**

When we previously defined our convLSTM model using the Goodfellow equations, it utilized sigmoid on the four gates, and a hyperbolic tangent on the output. It was explored whether other activation functions could be substituted in the network. This proved to create the best results we had ever received. Our latest model replaces each sigmoid operation with *tanh*, and changes the activation on the output from tanh to a rectified linear unit maxed at six (***ReLU6****).* ReLU6 is used instead of a traditional ReLU function as it gives a better performance. We are unsure of the direct causation of this improvement. The loss of the network dramatically falls from several hundred to less than one, meaning a near perfect prediction on still images.

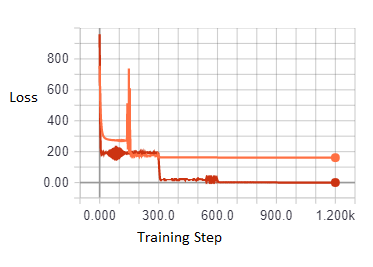


Figure 7: Two graphs taken from the TensorBoard output overlaid. The orange line represents the network using sigmoidal activation functions, and the red line utilizes hyperbolic tangent and ReLU6. Both networks use an exponentially decreasing learning rate every 300 steps, which causes a drop in loss.

**6 Tensorboard Visualization**

TensorFlow has a built-in visualization library known as TensorBoard. It works in a similar fashion to matplotlib, but allows for more complex graph visualization. Since any model designed in TensorFlow is based off a computational graph, we can visualize our model easily and have a physical representation of the inner mechanisms. Debugging a large neural network then becomes more manageable as connections are created between input and weights defined by the network.

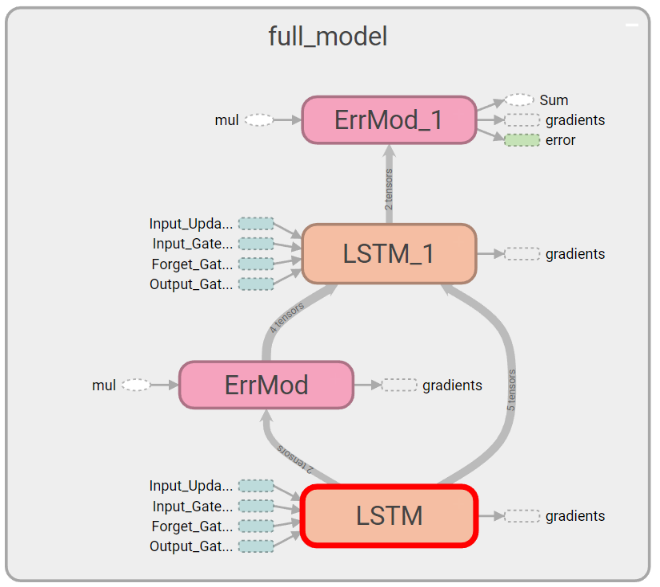


Figure 8: TensorBoard’s visualization of the predictive autoencoder model with two LSTM unrollings. Each module shown can be expanded in Tensorboard to better visualize the formal model architecture and where each weight tensor and input tensor are processed (image omitted due to its size).

Alongside its graph visualization, TensorBoard also contains a multitude of methods to display scalar values, pictures, matrixes, sounds, etc. For this model, we focus on both scalar and image representation as it allows us to see if the model is learning and how the training weights are being affected. More important than the visual output of the model, the weight visualization is an extremely powerful tool. In the case of a moving circle on a plane, we can infer that the weights should show emphasis on the right side. After running several trials with the model, TensorBoard will display the final trained weight matrixes of grayscale values.

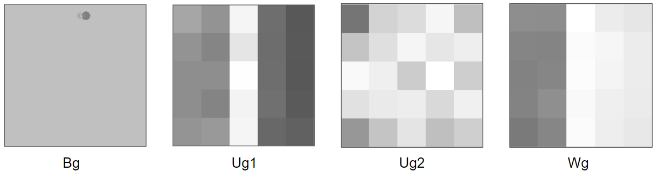


Figure 9: Weight visualization taken from the input gate after training the model against twenty unique animations. The images shown are the **bias**, Bg; **feedforward weights**, Ug1 and Ug2; and the **recurrent weights**, Wg. The feedforward weights are displayed as two separate matrixes as one slice handles error input while the other manages LSTM state input. These weights show signs of bilateral symmetry, emphasizing the right side of the image (which is the direction in which the dot moves).

**8 Discussion**

With our research, we have begun the process of perfecting a predictive autoencoder. Such a complicated model will take more research and trial before we can scale it properly. These first steps are important to any deep learning model creation to avoid the inevitable “black box” that comes from heavily trained and dense networks. Our model in its infant stage still proves to have a powerful enough architecture for unsupervised learning with only a few feature maps. The larger scope of creating an auto-encoding model is the ability to train a neural network using its own error as input, just as a human can learn from error. A predictive autoencoder model can provide more insight into the mystery of the human brain. Such a model is also extremely scalable, and can be implemented in larger, more complex problems.

**Acknowledgements**

The author would like to thank Dr. Anthony S. Maida and his accompanying research lab for the guidance and help necessary to make progress in this model’s conception.

**References**

[1] Goodfellow, Ian, et al. *Deep Learning.* Cambridge, MA, The MIT Press, 2017.

[2] Lotter, William, Gabriel Kreiman, and David Cox. "Deep predictive coding networks for video prediction and unsupervised learning." *arXiv preprint arXiv:1605.08104* (2016).

[3] Xingjian, S. H. I., et al. "Convolutional LSTM network: A machine learning approach for precipitation nowcasting." *Advances in neural information processing systems*. 2015.

[4] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *Nature,* 521.7553 (2015): 436-444.