EasyChair Preprint

№ 1579

Artificial Intelligence that Learn to Write Code : Memory Guided Programming

Poondru Prithvinath Reddy

October 3, 2019

.

Artificial Intelligence that Learn to Write Code : Memory Guided Programming

Poondru Prithvinath Reddy

ABSTRACT

Writing code requires the brain to understand the meaning of language and to properly organize the thoughts flow using the language. However, current models to write its own working code are primarily limited to searching through a huge code database developed by various experienced programmers. Here, we proposed a Memory Guided Programming(MGP) network to incrementally learn the meaning and usage of diverse functions / parameters, aiming to make best possible arrangement for a human-like machine programming process. MGP contains three subsystems : (1) Code system which consists of a mapping to transfer symbol texts into respective numeric and a RNN to generate the sequence dependencies from the input texts, and a output encoder to convert numeric values into text symbols; (2) Image system that contains an encoder to convert the input into abstract representations, and a DNN to classify image scenarios from real level representations; (3) a LSTM that combines inputs in the forms of both code and image, and predict text symbols and next images accordingly. In this work, the proposed MGP network illustrates the ability to incrementally learn different programming scripts and form a machine programming loop that enables interactions between Code and Image system. The paper presents an architecture that allows the machine to learn, understand and use programming language in a human-like way, which might enable a machine to construct programming scenarios and possibly possess human-like intelligence.

INTRODUCTION

Researchers have developed an Artificial Intelligence that can write code and the AI has the ability to learn. This AI can write working code after searching through a huge

code database. It tries to make the best possible arrangement for the harvested code fragments and improves its efficiency over time. According to Researchers, such system could be of great utility to non-coders. They only have to describe their program idea and wait for the system to create it.

Teaching an AI to write Python code with Python code was published by a French Engineer. He used Long Short Term Memory (LSTM) - one of the most popular architectures of recurrent neural networks. He fed it with lots of Python code (using libraries such as Pandas, Numpy, Scipy, Django, Scikit-Learn, PyBrain, Lasagne, Rasterio). The combined file weighed 27MB. The AI then generated its own code and the code is far from perfect. But the engineer thinks that it's not bad for a network that had to learn everything from reading example code.

There is no doubt that computers will be much better at programming in the near future than they are now and It's just a matter of time until neural networks will produce useful code. But AI isn't going to take away the jobs of developers. Instead, a system based on program synthesis can be used to automate the tedious parts of code development while the developers focus on complex tasks. There are already startups using automation to build 'smart software' that use artificial intelligence to develop custom software, eliminating strenuous processes and drastically reducing manual overhead. The programmers need to be worried about being replaced by AI in the near future, the fact is that nearly a third of software developers fear that artificial intelligence will eventually take their jobs. The thought of obsolescence due to A.I. was also more imminent or by seeing their skills and tools become irrelevant.

METHODOLOGY

In an attempt to reproduce human-like programming patterns in machines, We created an artificial neural network inspired by the human memory processes that take place as people are performing a given task including the maintenance and manipulation of information.

We proposed a memory guided programming network to incrementally learn the meaning and usage of numerous words and syntaxes, aiming to form a human-like machine programming process.

The Memory Guided Programming network has three key components: a coding system, an image system and an artificial neural network implemented by LSTM. The coding system which consists of a mapping to transfer symbol texts into respective numeric and a RNN to extract the sequence dependencies from the input texts, and an output encoder to convert numeric values into text symbols.

The second sub-system, an image system that contains an encoder to convert the input into abstract representations, and a DNN to classify image scenarios from real level representations. The final component of the Memory Guided Programming network mimics the human brain by a LSTM, combining inputs of both image and code representations to predict text symbols and next images accordingly.

We evaluated the Memory Guided Progamming network in a series of experiments and found that it successfully acquired programming tasks in a cumulative way. The technique also formed the 'machine programming loop," showing an interaction between code and image. In the future, the MGP network could aid the development of more advanced AI, which is capable of human-like programming strategies, such as custom Software development and predictive programming.

ARCHITECTURE

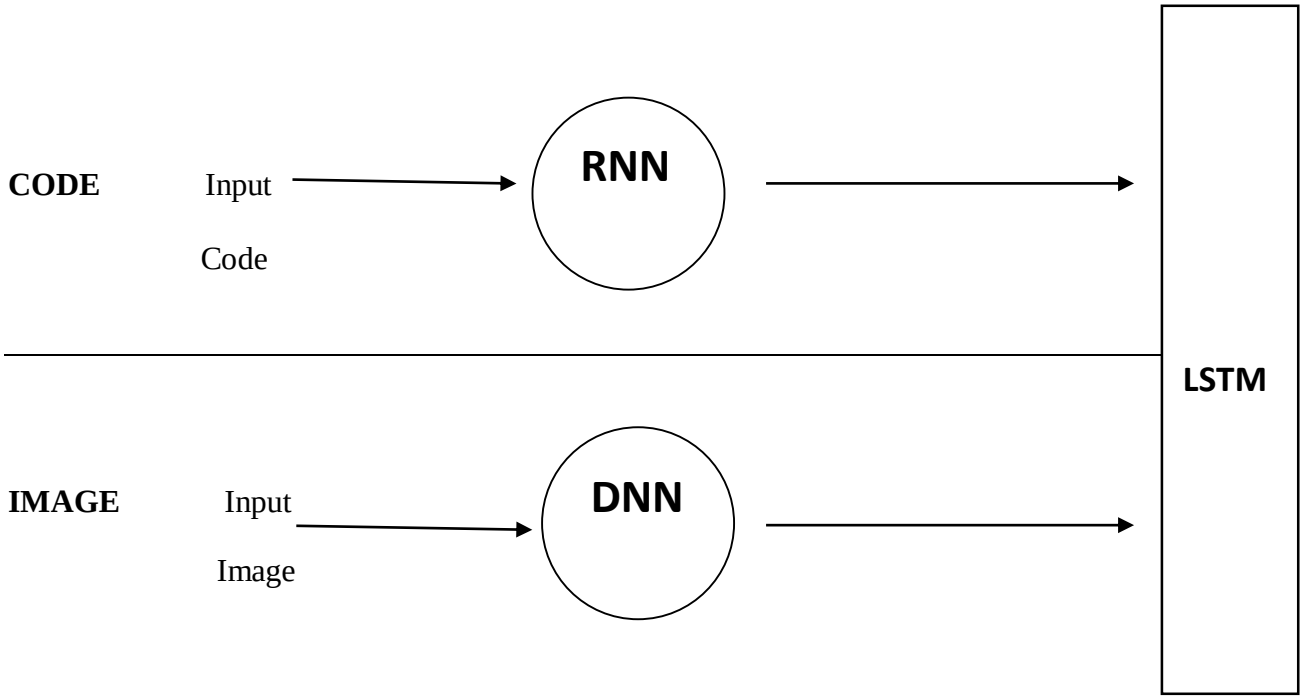


Figure 1: The MGP(Memory Guided Programming) architecture. It contains three subsystems that are trained separately.. In the image subsystem, the encoder can transfer an input (or predicted) image into a population representation vector I at the DNN layer (mimicking the Deep Neural Network for high-level image representation), and the decoder can reconstruct a vector output from LSTM to a predicted image, which can be fed into the encoder to form the guided loop. In the code subsystem, The coding system which consists of a mapping to transfer symbol texts into respective numeric and a RNN to extract the sequence dependencies from the input texts, and an output encoder to convert numeric values into text symbols. There is a memory layer implemented by a RNN to extract sequence information from the vector C. The LSTM layer serves as working memory, that takes the concatenated input [C, I] from both code and image subsystems, and output the predicted next element representation that could be fed back into both subsystems to form a guided loop.

As is shown in Figure 1, the MGP network contains three main subsystems including the code, image and LSTM subsystems. The image encoder network was trained separately. After training, the encoder is separated into two parts: the encoder (or recognition) part ranges from the image entry point to the final encoding layer, to provide the high-level abstract representation of the input image; the decoder part ranges up to image prediction point. The activity vector of the encoding layer are concatenated with code activity vector as input signals to the LSTM. Finally, the predicted image is fed back to the encoder network for the next iteration. The code processing component first converts the input text symbol into a sequence of binary vectors [C(t = 0), . . . ,C(T)], where T is the text length. To improve the code recognition, we added one RNN layer to generate the sequence dependencies of the text. The LSTM training based on the next component prediction (NCP). The LSTM is trained by the NCP principle, where the goal of the LSTM is to output the representation vectors (including both code and image) of the next component which required the understanding of the previous text code and observed images. The LSTM subsystem contains a LSTM and a full connected layer. It receives inputs from both code and image subsystems in a concatenated form of c(t) = [C(t),I(t)] at time t, and gives a prediction output a a'(t) = [C'(t), I'(t)], which is expected to be identical to a(t + 1) = [C(t + 1), I(t + 1)] at time t+1. This has been achieved with a next component prediction (NCP). So given an input image, the LSTM can predict the corresponding code description. The strategy of learning by predicting its own next component is essentially an unsupervised learning. Our LSTM subsystem was trained separately after code and image components had completed their functionalities. Finally, we demonstrate how the network forms a thinking loop with text code and predicted images.

DATASET

User Interface Elements

When designing the user interface, the following Interface elements are considered but are not limited to:

- **Input Controls**: pointer, checkboxes, radio buttons, dropdown lists, list boxes, buttons, toggles, text fields, date field, frames, combo boxes, timer, hscrollbar, vscrollbar, drivelistboxes, dirlistboxes, filelistboxes, shape, line, pictureboxes, data, ole, labels, charts
- **Navigational Components**: breadcrumb, slider, search field, pagination, slider, tags, icons
- **Informational Components**: tooltips, icons, progress bar, notifications, message boxes, modal windows, links

- **Containers**: accordion

A total of 40 user interface components / elements along with C programming language scripts / code associated with each visual component has been selected as dataset.

CODE SUBSYSTEM

The First problem is to represent our data.

A neural network treats only numbers. Everything else is unknown to the network. Thus, each character of our dataset should be represented in this form (a number / characters).

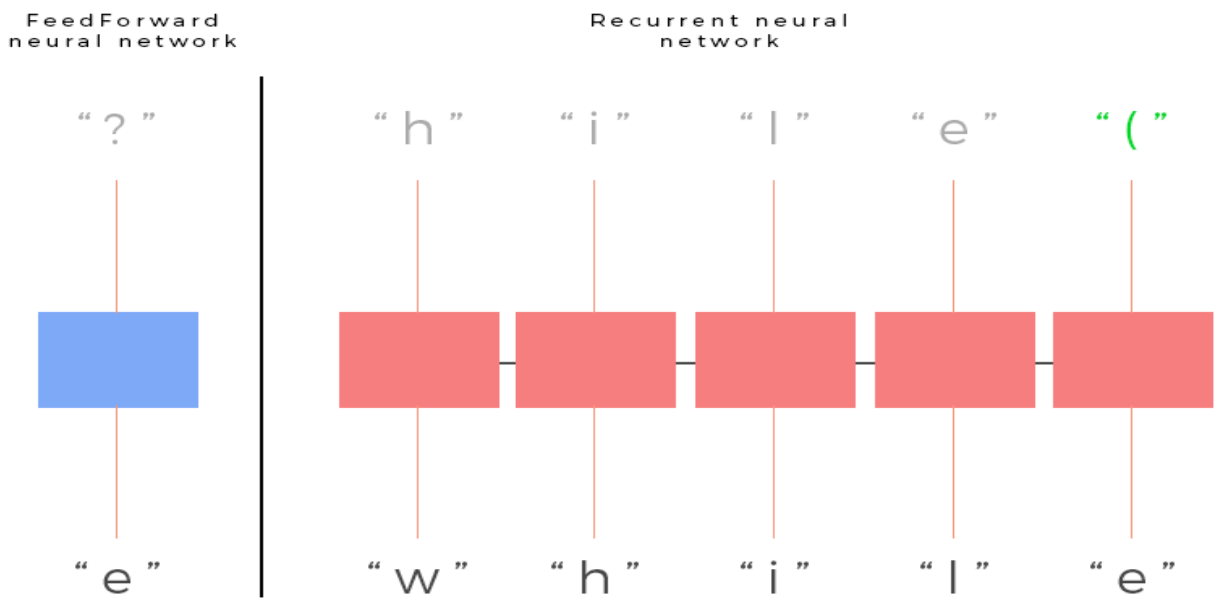
First we need to Load the text file and create character to integer mappings. The entire text file is read, we would be mapping each character to a respective number and all characters are converted to numbers. This is done to make the computation part of the RNN easier.

For example, if the character "=" is assigned to the number 7, we will then represent each number in one hot encoding in order to better converge during the backpropagation.

The three important variables to remember here are **vocab_to_int**, **int_to_vocab** and **encoded**. The first two allow us to easily switch between a character and an int and vice versa. The last is the representation of all our dataset in an encoder format. (Only int instead of characters)

We therefore create a neural network taking into account the temporal space of the characters type. To do this, we need to use a reccurent neural network.

Recurrent neural network



In order to illustrate, a classic classifier (on the left of the diagram) takes the preceding letter; it’s passed by the hidden layer represented in blue in order to deduce an output. A recurring neural network is architecturally different. Each cell (represented in red) is not only connected to the inputs, but also to the cell of the instant t-1. In order to represent our code subsystem, we will use RNN (Recurrent Neural Network) cells.

Building the model

We will describe this with 5 main parts. Placeholder serving as an entry to our model. The initialization of our cells used to create the RNN. The output layer connected to each cell. The operation used to measure the model error. Finally, we will define the training operation.

Graph inputs

We define a placeholder for the input, and the shape expected for our input is therefore of size [number, size]. Each entry of the input batch being associated with a single output, we can define the same shape for our target. Finally we define a placeholder for the value of the probability used for the future dropout.

RNN

- **create_cell()** is used to create an RNN cell composed of neurons. This function also adds a dropout to the cell output.
- **tf.contrib.rnn.MultiRNNCell** is used to easily instantiate our rnn. We give as a parameter an array of **create_cell()** because we want an RNN consisting of several layers.
- **initial_state**: Knowing that each cell of an RNN depends on the previous state, we must instantiate an initial state filled with zero that will serve as input to the first entries.
- **cell_outputs** gives us the output of each cell of our RNN.
- **final_state** returns the state of our last cell which can be used during training as a new initial state for a next batch.

Graph outputs

The values at the output of our cells are stored in a three-dimensional table [number of sequences, sequence size, number of neurons] or [2, 10, 4]. We no longer need to separate the outputs by sequences. We then resize the output to get an array of dimension [20, 4] stored in the **seq_out_reshape** variable.

Finally, we apply a simple linear operation: **tf.matmul (..) + b.** This followed by a softmax in order to represent our outputs in the form of probability.

Loss

In order to apply our error operation, the targets of our batch must be represented in the same way and in the same dimension as the output values of the model. We use **tf.one_hot** to represent our outputs under the same encoding as our inputs. Then we resize the array (**tf.reshape ()**) to the same dimensions of the linear output: **tf.matmul (..) + b**. We can now use this function to calculate the error of the model.

Training

We simply apply an AdamOptimizer to minimize our errors.

Results

It‘s finally the results of the training. We have for this one used the following parameters:

- Size of a sequence: 50
- Size of a batch: 40
- Number of neurons : 256
- Depth of RNN: 2
- Learning rate: 0.0005
- Dropout: 0.5

The results presented below were obtained after training the model on CPU and the model is fit over 100 epochs.

Finally, let’s look at what type of code our model is capable of generating . It’s interesting to see that this model has clearly understood the general structure of a program related to visual components; A function, parameters, variables, conditions, etc.

IMAGE SUBSYSTEM

This is an implementation of building a deep neural network with TensorFlow.for Image Classification in user interface component dataset.

We used 40 images of different visual components / elements from User Interface elements dataset.

We start with a pretty simple analysis with the help of the ndim and size attributes of the images array: Note that the images and labels variables are lists, so we might need to use np.array() to convert the variables to an array.

. As guessed the 40 labels that are included in this dataset, the components are different from each other. Also These images are not of the same size.

Let’s start first with extracting some features - we’ll rescale the images, and we’ll convert the images that are held in the images array to grayscale. We’ll do this color conversion mainly because the color matters less in classification.

To tackle the differing image sizes, we’re going to rescale the images; We can do this with the help of the skimage or Scikit-Image library, which is a collection of algorithms for image processing.

In this case, the transform module will come in handy, as it offers a resize() function; We'll see that we make use of list comprehension to resize each image to 28 by 28 pixels. Once again, for every image that we find in the images array, we'll perform the transformation operation that is borrowed from the skimage library. Finally, we store the result in the images28 variable:

. Next we'll also go through the trouble of converting the images to grayscale. Just like with the rescaling, we again count on the Scikit-Image library to help out; In this case, it's the color module with its rgb2gray() function that we need to use to get where we need to be.

However, we need to convert the images28 variable back to an array, as the rgb2gray() function does expect an array as an argument.

We checked the result of grayscale conversion by plotting some of the images;

Now that we have explored and manipulated the data, it's time to construct neural network architecture, layer by layer with the help of the TensorFlow package.

- Next, we build up the network. We first start by flattening the input with the help of the flatten() function, which will give an array of shape [None, 784] instead of the [None, 28, 28], which is the shape of our grayscale images.
- Activation function :The activation function of a node defines the output given a set of inputs. A common activation function is a Relu, Rectified linear unit.
- After we have flattened the input, we construct a fully connected layer that generates logits of size [None, 40]. Logits is the function operates on the unscaled output of previous layers, and that uses the relative scale to understand the units is linear.
- With the multi-layer perceptron built out we can define the loss function. Loss function - after we have defined the hidden layers and the activation function, we need to specify the loss function and the optimizer. The loss function is a measure of the model's performance. We make use of

  sparse_softmax_cross_entropy_with_logits()

- This computes sparse softmax cross entropy between logits and labels. In other words, it measures the probability error in discrete classification tasks in which the classes are mutually exclusive. This means that each entry is in exactly one class. Here, a user element can only have one single label.
- The optimizer will help improve the weights of the network in order to decrease the loss. In this case, we pick the ADAM optimizer, for which we define the learning rate at 0.001.

The above has been implemented with Python and TensorFlow as a backend.

Now that we have built up our model layer by layer, it's time to actually run it! To do this, we first need to initialize a session with the help of Session(). Next, we can use this initialized session to start epochs or training loops. In this case, we pick 201 because we want to be able to register the last loss_value; In the loop, we run the session with the training optimizer and the loss (or accuracy) metric that we defined. We also pass a feed_dict argument, with which we feed data to the model. After every 10 epochs, we'll get a log that gives us more insights into the loss or cost of the model.

We have now successfully trained our model with all the visual components.

We still need to evaluate our neural network. In this case, we try to get a glimpse of how well our model performs by picking 10 random images and by comparing the predicted labels with the real labels.

We can first print them out, by using matplotlib to plot the components themselves and to make a visual comparison.

However, by looking at random images give us many insights into how well our model actually performs. Then we loaded in the test component data and run predictions , and found that images were classified with good accuracy.

LSTM SUBSYSTEM

The LSTM subsystem contains a LSTM and a fully connected layer. It receives inputs from both code and image subsystems in a concatenated form of $\mathbf{c}(t) = [\mathbf{C}(t),\mathbf{I}(t)]$ at time t, and gives a prediction output $\mathbf{a'}(t) = [\mathbf{C'}(t),\mathbf{I'}(t)]$, which is expected to be identical to $\mathbf{a}(t + 1) = [\mathbf{C}(t + 1),\mathbf{I}(t + 1)]$ at time t+1. This has been achieved with a next component prediction (NCP) . So given an input image, the LSTM can predict the corresponding code description. The strategy of learning by predicting its own next element is essentially an unsupervised learning.

The Training is based on the next component prediction (NCP). The LSTM-FC is trained by the NCP principle, where the goal of the LSTM-FC is to output the representation vectors (including both code and image) of the next component / element. At time T, the LSTM of MGP generated the guided digit instance, which required the understanding of the previous code language and observed images.

The LSTM subsystem was trained separately after vision and code components had completed their functionalities. We have trained the network to accumulatively learn different components, and the related code results. Finally, it is demonstrated how the network forms a thinking loop with code language and observed images.

The LSTM layer serves as working memory, that takes the concatenated input [**C**,**I**] from both code and image subsystems, and output the predicted next component representation that could be fed back into both subsystems to form a guided loop.

RESULTS

After 200 steps training, MGP could not only reconstruct the input image but also predict the element / component with associated program script / code, correct parameters and variables just after the image classification. MGP has the capacity to correctly predict the next component and the code (with correct syntax, parameters, variables) at the proper time point. After training of 200 steps, MGP could classify various visual components with correct code (accuracy = 16%). Note that, the classification process is not performed by large dataset, but by small number of training steps or iterations which is resulting in less accuracy.

CONCLUSION

In this paper, we introduced a LSTM layer to involve representations from both Code and Image Subsystems to form a human-like programming system(the MGP system). The MGP contains three subsystems; the Code, Image and LSTM subsystem , which are trained separately. In the Code subsystem, we use a RNN layer to extract the sequence dependencies from language text and proposed a output encoder to produce text symbols output. We propose to train the MGP with the NCP. MGP shows its ability to learn different structures or tasks in a cumulative learning way, and form a machine programming loop with the interaction between images and language text.

REFERENCES

1. **V Murali, L Qi, S Chaudhuri, C Jermaine** . Neural sketch learning for conditional program generation **- arXiv preprint arXiv:1703.05698, 2017 - arxiv.org**