CONSTRUCTIONISM 2020

The University of Dublin

Trinity College Dublin

IRELAND

"*26-29 May*"

Constructionism 2020

26th to 29th of May 2020 in Dublin, Ireland

Proceedings of the 2020 Constructionism Conference

Edited by Brendan Tangney, Jake Byrne and Carina Girvan

ISBN 978-1-911566-09-0

Deep Learning Programming by All

Ken Kahn, *toontalk@gmail.com,*
Dept of Education, University of Oxford, Oxford, UK

Yu Lu, *luyu@bnu.edu.cn, Advanced Innovation Center for Future Education, Beijing Normal University, Beijing, China*

Jingjing Zhang, *jingjing.zhang@bnu.edu.cn, Big Data Centre for Technology-mediated Education, Beijing Normal University, Beijing, China*

Niall Winters, *niall.winters@education.ox.ac.uk, Dept of Education, University of Oxford, Oxford, UK*

Ming Gao, *mgao519@126.com, Big Data Centre for Technology-mediated Education, Beijing Normal University, Beijing, China*

Abstract

We describe an open-source blocks-based programming library in Snap! [Harvey and Mönig, 2010] that enables non-experts to construct machine learning applications. The library includes blocks for creating models, defining the training and validation datasets, training, and prediction. We present several sample applications: approximating mathematical functions from examples, attempting to predict the number of influenza infections given historical weather data, predicting ratings of generated images, naming random colours, question answering, and learning to win when playing *Tic Tac Toe*.

Keywords (style: Keywords)

Machine learning, Snap!, visual programming, constructionism, neural nets, artificial intelligence

Introduction

This work builds upon the work of Kahn and Winters [Kahn and Winters, 2018] who developed a visual programming library designed to be used by high school students in building AI applications. They first created Snap! blocks that connect to AI cloud services for speech and image recognition as well as speech synthesis [Kahn and Winters, 2017]. They then provided block-based interfaces to several pre-trained deep learning models for services such as transfer learning, pose detection, style transfer, and image labelling. Here we present new additions to this Snap! library that support the definition of the architectures of deep learning models, their loss functions, their optimization methods, training parameters, and obtaining predictions [eCraft2Learn, 2020a]. In addition to the library itself, there are learning resources including interactive guides and sample projects

Motivations

In the last decade machine learning systems have demonstrated very impressive capabilities including image recognition, translation, autonomous driving, speech recognition, interpretation of medical imagery, transcription, recommendation generation, robot control, game playing, and much more. It is likely to continue greatly affecting nearly every aspect of modern life. Students who experience in a hands-on manner the possibilities, strengths, and weaknesses of this technology are likely to obtain a deeper understanding than those who simply study the technology. This viewpoint reflects on Dewey's (1938) 'learning by doing' theory, which emphasises the value of experience and engagement. While we expect a small fraction would go on to become AI researchers or engineers, we expect the rest will be better prepared to contribute,

in an informed manner, to a society rapidly changing due to the impact of AI. Such students are likely to be better equipped to deal with the social, economic, and ethical issues that are arising from the use of machine learning.

We aim to provide high-school and non-computer science undergraduate students with limited programming abilities with tools for acquiring experiences designing, training, testing, and using deep machine learning models. We believe that our reliance upon a blocks-based language has many advantages over simply providing machine learning "wrappers" in JavaScript or Python. Students have fewer distractions such as syntax errors. Their cognitive load is less since they rely upon drag and drop of blocks instead of needing to remember the names of primitives. The blocks can be very readable without the usual cost involved in entering verbose instructions. The Snap! blocks provide an intuitive interface for asynchronous functions, a construct that many learners find difficult in textual languages. And a great number of students have familiarity with Scratch [Resnick et al. 2009] upon which Snap! was based. While these blocks have yet to be used in studies with our intended audience, we expect the kinds of success we have seen with other parts of our Snap! AI library [Kahn and Winters, 2018; Kahn et al, 2018; Loukatos et at, 2019] when we run trials of these machine learning blocks.

The students who master our machine learning library can become empowered to build impressive apps that listen, see, predict, and more. This may motivate them to create innovative applications that match their interests and passions. Furthermore, in the process they may acquire the ability to reflect more deeply upon how they perceive, reason, and act. While deep learning neural nets are very different from brains, and how they perform and are structured is different from minds, they are still useful models of cognition. And perhaps students who acquire concepts for more effectively thinking about thinking may become better learners [Papert, 1980; Minsky, 2019; Kahn and Winters, 2020].

Related Research

While there is a great deal of support for building deep neural networks in Python and JavaScript, we are focused on supporting learners lacking the technical skills to effectively use those resources. Mathematica's Wolfram Language has a good deal of support for machine learning including learning resources aimed at middle and high school students [Wolfram, 2017a; Wolfram, 2017b]. Our efforts differ from this in that we are building upon the ease-of-use and familiarity of blocks-based languages such as Scratch [Resnick et al., 2009] and Snap!. Furthermore, Snap! and our library are open-source and run in modern browsers without any installation requirements.

There are other efforts to integrate machine learning with blocks-based languages including the Machine Learning for Kids website [Machine Learning for Kids, 2020] and the Cognimates project [Druga, 2018]. These systems, like the earlier work of Kahn and Winters, offer blocks that provide easy-to-use access to various AI cloud services. Unlike our current efforts, they do not provide a programmatic interface for constructing neural networks – the programmatic interfaces they provide are only for training and using neural nets.

SnAIp is a project that aims to implement machine learning techniques in Snap! [Jatzla et al, 2019]. Unlike the Snap! blocks described in this paper, there are no black-box implementations in JavaScript, and no reliance upon complex APIs or cloud services. Enabling students to see how machine learning works in terms of blocks they are familiar with clearly has advantages. But technically it is very difficult to achieve the speed and scale that our blocks are capable of. Also our blocks provide access to very powerful APIs that would be a tremendous effort to fully replicate in Snap!. Ideally students should have access to both of these Snap! libraries so they can incorporate both transparent functionality and very capable functionality into their projects as appropriate.

Google's Teachable Machine [Google, 2020a] is a web page where users can train the system to classify images. The TensorFlow Playground [Google, 2020b] is a web page where one can interactively define, train, and test a deep learning neural net. While wonderful learning resources, these systems do not provide a programmer interface.

Snap! Blocks for AI Programming

Snap! [Harvey and Mönig, 2010] is a blocks-based programming system that closely resembles the immensely popular Scratch programming system [Resnick et al. 2009]. Snap! is a much more powerful and expressive language than Scratch because of its thorough support for first-class functions and lists. Also, crucial to our efforts, new blocks can be defined either in Snap! or in JavaScript. All the blocks in our library are either defined in JavaScript or in terms of other blocks that ultimately rely upon our JavaScript-defined blocks. Note that this implies that we didn't touch the source code of Snap! in implementing our extensions to Snap!. Student projects relying upon our blocks can be loaded into an ordinary Snap! web page.

Snap! itself is implemented in JavaScript and hence can run in any modern web browser without the need for any extensions or plugins. (Note that the neural network blocks work in Chrome and Firefox but currently not in most other browsers.) Snap! is open source, well-documented, supported by an active community, and under continual further development.

A blocks-based programming system has several advantages over text-based languages. Because blocks only click together if they are syntactically compatible, they eliminate the need to learn the syntax and, perhaps more importantly, eliminate the possibility of syntax errors or misspelled commands. A block can also be any mixture of text, input parameters, and icons, making it more readable. And blocks can easily be displayed in languages other than English.

Blocks are organized into palettes that enable users to browse for appropriate blocks. Consequently, users needn't memorize large numbers of language primitives. Well-designed learning activities could aid students to use blocks to reduce cognitive load during the learning process [Çakiroğlu et al., 2018]. While this relieves memory demands on users and facilitates the discovery of new functionality, it can be significantly slower than typing. Snap! addresses this by providing a keyboard method for searching for blocks.

In Figure 1 the block for obtaining a prediction from a model will either call the "say" block with the prediction for the input (36) or else the "think" block will be called with the error message. Note that while this block expects success and errors continuations (also known as callbacks) it does so in a manner accessible by beginning programmers.

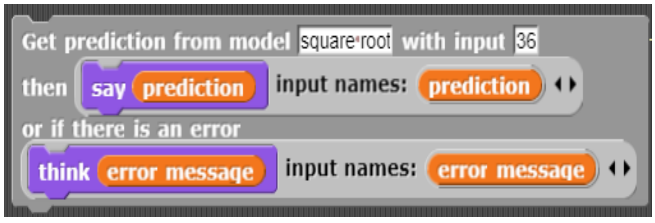

Figure 1 - Prediction block with two embedded blocks

Deep Neural Networks

The neural networks that can be built with our library consist of a large number of connected artificial neurons. These neurons are loosely based upon biological neurons. The connections between neurons have weights that encode how much a neuron influences another neuron. In our work, as is typical for deep learning models, the neurons are organized into layers. The first layer receives the input and the last layer produces the output. The layers in-between, called "hidden layers", typically produce successively higher-level features or interpretations of the data. Our library includes blocks for defining these layers and their connections.

Neural nets work exclusively with numbers. To work around this for classification tasks, numbers are used to encode labels. E.g., if the sentiment of some text is either "negative", "neutral", or "positive" this can be encoded as 0, 1, and 2. The training blocks accept text labels as outputs and converts them to "one-hot encodings" (vectors with one 1 and the rest 0). Image input is usually converted to a list of pixel values (either black and white (0 and 1), grayscale (0.0 to 1.0), or values

for the red, green, and blue intensities). There are various schemes for converting words or sentences into a vector of numbers [Mikolov, 2013]. Other parts of our library provide blocks that can be used to convert images and text into vectors of numbers well-suited for machine learning.

The weights associated with neurons are randomly initialized. These weights are updated as the model is trained on data. In supervised learning the data includes examples of outputs associated with inputs. During training the system adjusts the weights in an attempt to reduce the difference between its predictions and the desired outputs given in the datasets. Our Snap! library provides blocks for controlling many aspects of this training phase.

Many of the concepts underlying neural nets are over fifty years old [Minsky & Papert, 1969] but only began leading to many thousands of useful systems in the last decade [Deepindex, 2020]. This recent success is usually attributable to much more powerful computation engines and the availability of large amounts of data. Computation engines typically exploit the graphical processing units (GPUs) found in most computers, tablets, and smartphones. These accelerators often decrease the time it takes to train a model or use it for predictions by a factor of one hundred or more.

In our work we are able to exploit the speedups from using GPUs due to the arrival of TensorFlow.js [Smilkov, 2019]. This is an implementation of TensorFlow, a popular machine learning API, in JavaScript. It can access the GPU of a laptop, desktop computer, or phone using the WebGL interface [WebGL, 2019] that is supported by all modern web browsers.

Training that relies upon big data can be a problem for students using our library. Students are likely to have problems acquiring millions of labelled images and importing them into a browser. And models built by professionals can take weeks to be trained on huge collections of images, even when using a large number of state-of-the-art GPUs or other accelerators. While many tasks are consequently impractical for students to attempt, fortunately, there remain many interesting tasks that don't require huge datasets or computing resources.

Most neural net applications run on servers that accept data from a client and respond with predictions. This enables the service providers to host their models on very powerful servers, sometimes on special hardware for accelerating machine learning. Many business models rely upon providing functionality via servers.

There are drawbacks however. Many people are concerned about privacy concerns when using these services. Voice and video are often transferred to the servers. The cost of using a server is often an obstacle. Another disadvantage of running the models on servers is that applications cannot be as responsive as ones that run locally on the user's devices. Also, applications that rely upon servers work only when the client has a fast and reliable network connection.

By running on a user's device in a web browser these drawbacks are avoided. One can download Snap! and our library and then run everything without an Internet connection.

A Library of Machine Learning Blocks

The Snap! library exists in two forms: (1) a set of blocks that can be imported into any Snap! instance or (2) a Snap! project that includes illustrative instances of the use of the blocks together with informative comments. One can simply click on any of the instances to run them.

Typically, we provide at least two versions of any block: (1) the simplest usable version and (2) a full-featured version.

Model Creation

The simple block for creating a model is illustrated in Figure 2. It creates a model named 'guess relation' with a single input that is connected to 100 neurons. Each of those neurons is connected in turn to 50 neurons that are connected to a single output neuron. When the system has finished creating the model the "say" block is run.

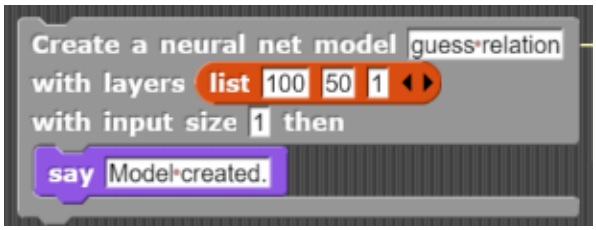

Figure 2- A simple block for creating a model

Figure 3 shows an example of the full-featured version of this block. It differs from the example in Figure 2 by specifying which optimization method and loss function are desired. Note that these are indicated by using pull-down menus for ease of use. Documentation of these methods and functions is provided via the help menu item and in the programming guide for greater detail. Examples of using this block for more difficult tasks are described later.

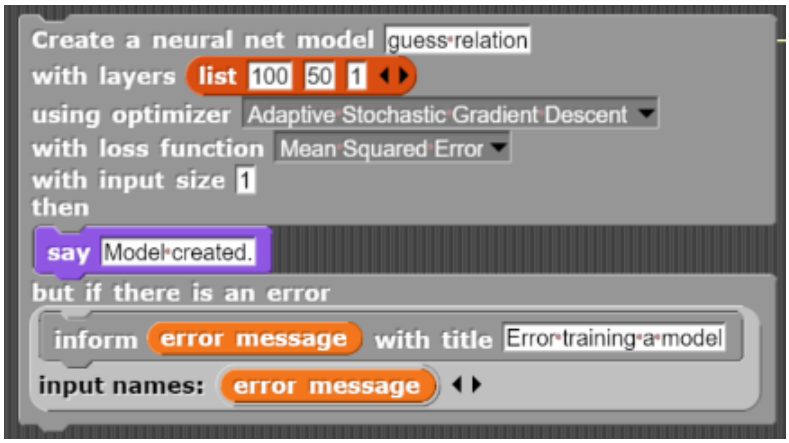

Figure 3 - A full-featured block for creating a model

Training

Once a model is created one can begin training it. First one specifies the training dataset and optionally the validation dataset. A dataset consists of two Snap! lists with the same number of elements: the input and the output. The lists can consist of numbers, or lists of numbers (e.g. coordinates or red-green-blue intensity triples), or any number of levels of lists. The output can also contain text strings that are converted to numbers internally. The validation dataset, if provided, does not influence the weights during training and is used to provide predictions free of the risk of overfitting for evaluating the model. An alternative to providing a validation dataset is to request that a specified fraction of the training data be set aside for validation.

In Figure 4 a dataset containing the first five positive integers is used as input and the output is computed using 2*n+1. (This is the machine learning analogue of the “Hello World” program – an extremely simple example.) This block can be used to either completely define the dataset or to provide data to be added to the current dataset. Datasets can be available for all models to use or be associated only with a specified model.

Figure 4 - Specifying the training dataset

After specifying the dataset one can begin training. Figure 5 illustrates the simplest block for initiating training. It requests 50 iterations of the training step and then displays the statistics from the training as shown in Figure 6.

Figure 5 - A simple block to initiate training

loss	0.0014190125511959
accuracy	0
duration in seconds	3

Figure 6 - The resulting training statistics

Figure 7 is an example of using the full-featured version of the training block. It specifies a learning rate of .001, that the data should be shuffled (to avoid any artefacts resulting from the order), that none of the data should be used for validation. It also responds to any errors that arise.

Figure 7 - A full featured block for initiating training

Prediction and Classification

In Figure 8 we see a block requesting that the model predicts the output given 10 as the input. For this example, the “say” block will be passed a number close to 21 (i.e., 2*10+1). There is a version of this block that accepts a list of inputs and replies with a list of corresponding predictions. If the model has been trained to label the input, then the output is a list of pairings of labels and confidence scores.

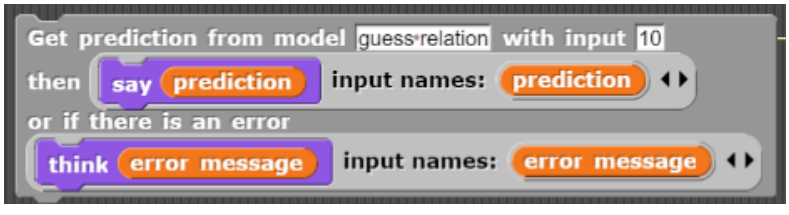

Figure 8 - A block for getting predictions from a model

A Graphical Interface for Creation, Training, and Prediction

One of the challenges in creating good deep learning models is that some architectures (number of layers and number of neurons in each layer) can be quickly trained and produce accurate predictions, while others are difficult to train or produce poor results such as bad predictions or

classifications. Deciding the size of the training data is challenging. Small datasets lead to fast training but only sometimes produce good predictions. Similarly, it is hard to know what are good values for hyper-parameters such as the learning rate, number of training iterations, loss function, and optimization method.

One way we address this is to provide an optional graphical interface for setting all these parameters as illustrated in Figure 9. Students can use it to quickly try different parameters and teachers can use it to accompany a demonstration.

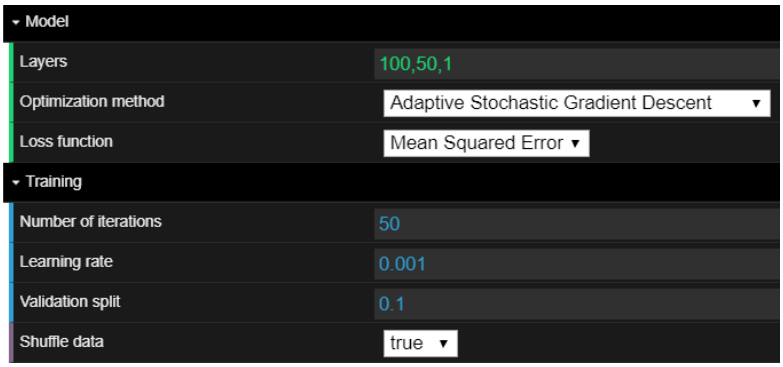

Figure 9 - A graphical interface for exploring architectures and hyper-parameter values

The graphical interface provides buttons for creating, training, and prediction. The training section provides real-time graphs of the training progress as seen in Figure 10. The x-axis is the number of training steps performed and the blue line shows the drop in the difference between predictions and the correct answers from the training data. The red line shows the difference for the validation dataset.

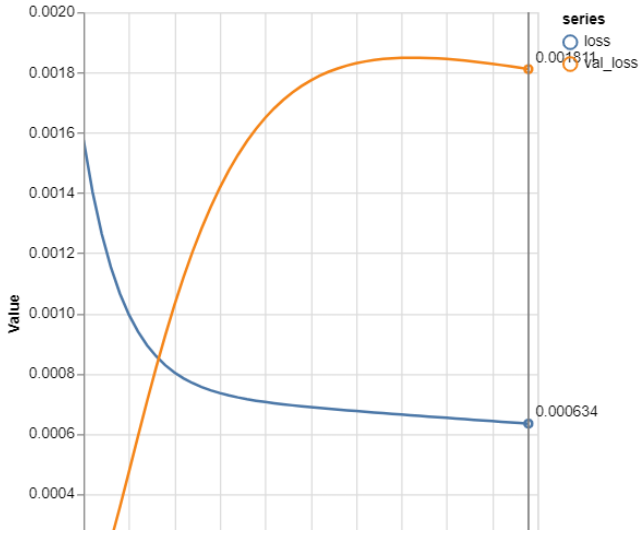

Figure 10 - a graph of the loss function during training

Hyper-parameter optimization

An alternative to tuning the model interactively is to let the computer search for good architectures and hyper-parameter values. While one can implement such a search in Snap*!* we also provide a block that interfaces to a TensorFlow.js hyper-parameter optimization library [Stoyanov, 2018]. Ambitious projects may find good settings much faster using this block than “manual” experimentation. We believe, however, that students can learn a good deal from some manual experimentation but it may become tedious for big projects.

Figure 11 shows a block that starts a search exploring 50 different parameter settings. As each experiment is performed the parameter values are displayed. The best settings are captured when the search completes, and another block can be used to create and train a model using those settings. Boolean switches are used to indicate what hyper-parameters to explore. Finally, weights are provided to guide the search towards the desired trade-off between accuracy, training time, and model size/speed.

The parameter search starts with the current settings and uses various heuristics to try values close to the currently most promising ones. The search can be customized in the graphical interface.

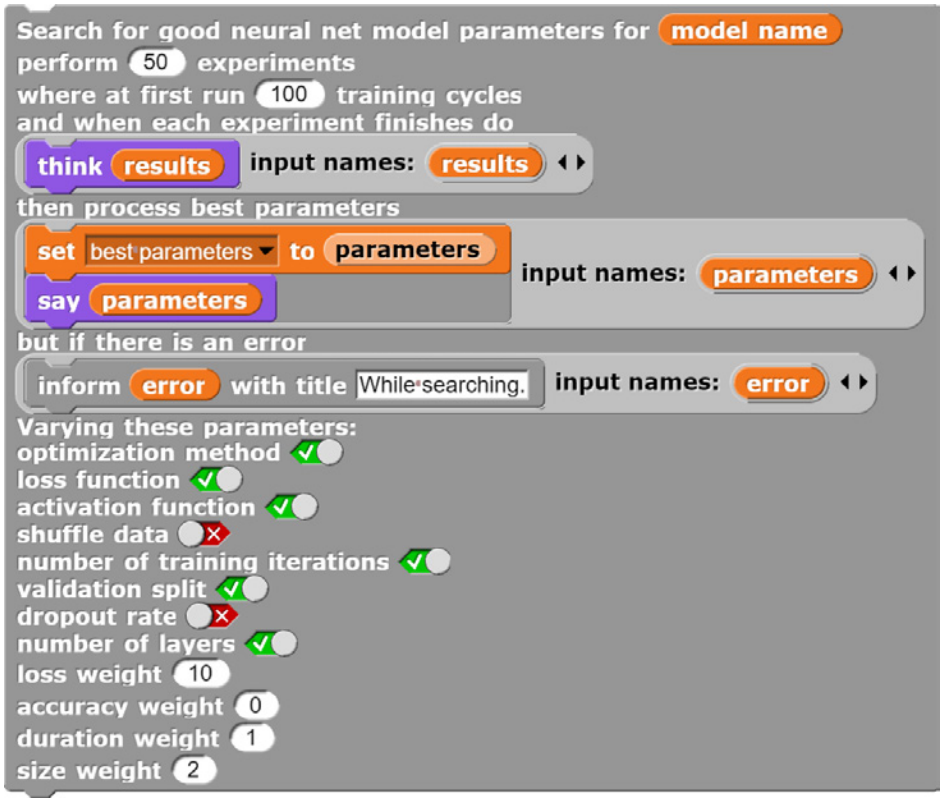

Figure 11 - Launching a search for good architectures and hyper-parameters

Illustrative Projects

Our library does not expose all the functionality of TensorFlow.js (which, while very powerful, supports only a subset of the full TensorFlow API). We decided to construct our library with only the most common and easiest to understand building blocks of deep learning programming. Despite this there is a wide variety of projects that it can support. Here we discuss several classes of projects that we developed, in increasing order of complexity.

Approximating Mathematical Functions

A pedagogically simple exercise using the deep learning blocks is to learn to approximate a mathematical function given sample input and output values. The early example of predicting 2x+1 is so simple that many alternatives to deep learning can work as well. A more interesting example is to provide a list of numbers as output and the square of those numbers as input. Attempting this one can discover how the architecture of the model and number of training examples strongly influences how well it can “predict” the square root of test numbers. Furthermore, if trained on numbers, say, between 1 and 100, one can explore how well it can approximate the square root of 1000 or 1/100. Figure 12 displays the approximations for 2, 1, 49, 900, and 0.01. Applying deep

learning to mathematical functions need not be constrained to functions of one argument nor functions that produce a single output value.

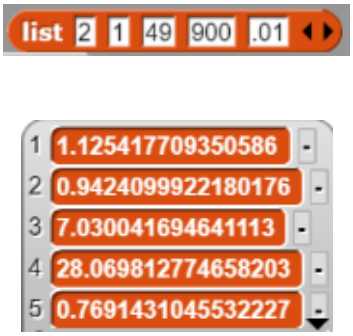

Figure 12 - A model's "predictions" of the square root of five test numbers

(notice how poorly it estimates the square root 0.01)

Some students may enjoy exploring the limits of this kind of technology. Can it reliably identify integers as prime or non-prime? Can it learn the prime factors of numbers? How well can it approximate transcendental functions?

Discovering Real-world Data Relationships

There are many freely available datasets that can be used as training data. Google Dataset Search is a convenient way to find them [Google, 2019c]. The example we explored starts with two datasets: the Global Historical Climatology Network-Daily Database [NOAA, 2019] and the Epi Data Surveillance Information from the World Health Organization [WHO, 2019]. The idea is that perhaps recent temperatures and precipitation can help predict subsequent occurrences of influenza. As is well-known, cases of influenza increase in the winter. Perhaps colder weather leads to more influenza in the following weeks. If there is such a relationship can deep learning discover it?

The first step is to import data into Snap! directly as CSV and JSON files. Alternatively, one can import the contents of a data file as a string and programmatically parse it. Snap! has all the mathematical and list processing primitives needed to support “data wrangling”. We experimented with input data being temperatures from the previous week and the subsequent number of reported flu infections. The temperature data was normalized to be the ratio of the temperature to the average temperature for each location for each time of year.

While this is an illustrative example of the kinds of explorations a typical high school student should be able to perform, we have yet to find any predictive value in knowing the previous week’s weather. Students can explore other relationships such as predictions from the previous several weeks. But negative results can be instructive as well. We remain optimistic that students will follow their interests in applying machine learning to data. There are also plenty of opportunities to tie these explorations to their other studies be it science, history, social studies, athletics, or language. Many machine learning projects can be developed to answer questions about epidemics such as covid-19.

Learning to Win (at *Tic Tac Toe*)

Deep learning has had several impressive accomplishments in game playing. DeepMind built a system that learned to play dozens of Atari video games [Mnih et al., 2013]. They later created AlphaZero that learned to play *Go* and *Chess* at world-class levels [Silver et al., 2017]. We have explored how students can do this themselves for simple games such as *Tic Tac Toe*.

We provide a Snap! implementation of *Tic Tac Toe* since our focus is on using machine learning to discover winning ways of playing and we are ignoring the challenge of learning the rules of play. We frame the problem as one of predicting the probability of winning given a specific board. A

board consists of 9 squares that can be X, O, or empty. One possible input to a neural net can be a vector of 9 instances of 0, 1, or 2. However, these numbers don't work as well as using a vector of 27 instances of either 0 or 1. Each board square is either <0, 0, 1>, <0, 1, 0> or <1, 0, 0>.

As a game is played a list of successive board positions is recorded. When the game ends then the outcome is encoded for each board that occurred during the game: 0 for a tie, 1 for a win, and -1 for a loss. After training, a model can be used to make moves by considering the boards resulting from all possible next moves. It can then choose the move that is most likely to lead to a win or instead choose a random move based upon the probability of that move winning.

It is not difficult to seed this process by collecting moves from games where a purely random player plays against another random player. After using this game play data as training data, one can begin to have a model play itself, a random player, another trained model (perhaps by a different student), or a human player. The moves from these games can then be used for further training.

We also provide a web page where one can experiment interactively with deep learning for *Tic Tac Toe*. Unlike the Snap! *Tic Tac Toe* program, the webpage does not yet support human players. It does, however, facilitate large scale experimentation since it can play hundreds of games per minute.

This approach to learning to win at a game relies upon the fact that the state of the game is known to all players and can be concisely and simply represented and that the number of possible moves on each step is small. Consequently, many popular games are too complex to learn to play them well in a manner similar to *Tic Tac Toe*. There are, however, several games that could use this approach. *Connect the Dots* and *Nim* are good examples.

More sample projects

We provide other examples of machine learning using the Snap! blocks. One generates random images and learns to rate them. Another asks the user to name randomly generated colours and then learns to predict the name of additional colours. Another project is a question answering system that can answer questions about the Snap! AI blocks library.

Learning Resources

For many students a library of deep learning programming blocks is not enough. They need tutorials, guides, sample programs, and clear documentation. In addition to the examples and documentation provided by the library presented as a Snap! project, we provide an interactive web-based guide [eCraft2Learn, 2020b]. Within the guide are instances of Snap! that enable readers to explore the blocks and examples on the same page as they are reading the guide. The guide by default also contains sections describing the underlying ideas, history, project ideas, links to videos and further information, and societal impacts. For students who are focused on programming these can easily be hidden.

Current Status and Future Developments

This paper reports on the design, implementation, and motivations behind our deep learning programming library and its associated learning resources. At the time of this writing we have tested this with only two high school students and two university students. We learned that unless the students understand the larger context and challenges of machine learning they are mystified why the computer can't more easily and accurately figure out, for example, square roots. Or even why one would need to train it since "it already knows how to compute square roots". Toy examples can be pedagogically valuable but only if presented as learning exercises and not as example of serious machine learning.

We have plans to test our library and learning resources with non-computer science university students in at least two classes. Given how well students do using Snap! libraries for speech

synthesis and recognition, using pre-trained neural nets, and word embeddings, we are optimistic students will master our deep learning library.

The source code, libraries, guides, sample projects, and additional documentation are all freely available online [eCraft2Learn, 2020a].

Currently our library has only been thoroughly tested in Chrome on personal computers. We have yet to succeed in getting it to run reliably on tablets or smartphones.

Ideally high-level building blocks should support not only construction but “deconstruction”, i.e. transparency in how they function. Users ideally should be able to open up our blocks to see how they work and modify them. While we see that is as very valuable it is very challenging in this case and beyond the scope of this project. But [Jatzlau et al 2019] demonstrate this is possible in some cases.

There are many technical enhancements we are considering. All of the models that can be created now are a sequence of fully connected layers. Convolutional layers are not yet supported; nor are recurrent networks and reinforcement learning. The ultimate challenge is to support all the functionality in TensorFlow.js in a manner that is accessible to people who are not expert users of some textual programming language such as Python or JavaScript.

Acknowledgments

Initially the work reported here was supported by the eCraft2Learn project funded by the European Union’s Horizon 2020 Coordination & Research and Innovation Action under Grant Agreement No 731345. Subsequently it is partially supported by the National Natural Science Foundation of China (No. 61702039) and the Fundamental Research Funds for the Central Universities.

References

Çakiroğlu, Ü., Suiçmez, S. S., Kurtoğlu, Y. B., Sari, A., Yildiz, S., & Öztürk, M. (2018). Exploring perceived cognitive load in learning programming via Scratch. *Research in Learning Technology*, 26.

Deepindex (2020) https://deepindex.org

Dewey J. (1938) Experience and Education. Macmillan, New York, NY.

Druga, S. (2018) Growing up with AI: Cognimates: from coding to teaching machines., PhD diss., Massachusetts Institute of Technology.

eCraft2Learn (2020a) https://ecraft2learn.github.io/ai/

eCraft2Learn (2020b) https:// ecraft2learn.github.io/ai/AI-Teacher-Guide/chapter-6.html

Google (2020a) https://experiments.withgoogle.com/teachable-machine

Google (2020b) https://playground.tensorflow.org

Google (2019c) https://toolbox.google.com/datasetsearch

Harvey, B. & Mönig, J. (2010) Bringing “No Ceiling” to Scratch: Can One Language Serve Kids and Computer Scientists? *Constructionism 2010 Proceedings*, Paris, France.

Jatzlau, S., Michaeli, T., & Seegerer, S. (2019). It’s not Magic After All - Machine Learning in Snap! using Reinforcement Learning. https://www.semanticscholar.org/paper/It-%E2%80%99-s-not-Magic-After-All-Machine-Learning-in-Snap-Jatzlau-Michaeli/6ffb305d63d33d9f1530db6883083e6f1a8e1ca3

Kahn, K. & Winters, N. (2017) Child-friendly programming interfaces to AI cloud services, *Proceedings of EC-TEL 2017: Data Driven Approaches in Digital Education*, 10474, 566-570.

Kahn, K. & Winters, N. (2018) AI Programming by Children, *Proceedings of Constructionism 2018*, Vilnius, Lithuania, 2018.

Kahn, K, Megasari R., Piantari, E. & Junaeti, E. (2018) AI Programming by Children using Snap! Block Programming in a Developing Country, Proceedings of the *EC-TEL Conference*, Leeds, UK, September 2018.

Kahn, K. & Winters, N. (2020) Constructionism and AI: A history and possible futures, *Proceedings of Constructionism 2020*, Dublin, Ireland, 2020.

Loukatos, D., Kahn, K. & Alimisis, D. (2019) Flexible Techniques for Fast Developing and Remotely Controlling DIY Robots, with AI flavor. Proceedings of the *Edurobotics 2018 Conference*, Rome, Italy, October 2018. In Moro, M., Alimisis, D. and Iocchi, L., 2019. *Educational robotics in the context of the maker movement*, Springer.

Machine Learning for Kids (2020) machinelearningforkids.co.uk.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). "Distributed representations of words and phrases and their compositionality." In *Advances in neural information processing systems*, pp. 3111-3119.

Minsky, M. (2019). *Inventive Minds: Marvin Minsky on Education*. MIT Press.

Minsky, M. & Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. MIT press.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). "Playing Atari with deep reinforcement learning." arXiv preprint arXiv:1312.560.

National Oceanic and Atmospheric Administration (2019). https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/readme.txt

Papert, S. (1980) *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y. (2009), Scratch: programming for all. *Communications of the ACM*, 52(11), 60-67 DOI=http://dx.doi.org/10.1145/1592761.1592779.

Silver, D., Schrittwieser, J., Simonyan, K. Antonoglou, I., Huang, A, Guez, A., Hubert, T. (2017). "Mastering the game of go without human knowledge." *Nature,* 550, no. 7676: 354.

Smilkov, D., Thorat, N. Assogba, Y., Yuan, A., Kreeger, N., Yu, P., Zhang, K. (2019) "TensorFlow.js: Machine Learning for the Web and Beyond." *arXiv preprint arXiv:1901.05350*.

Stoyanov, M., (2018).HPJS: Hyper-parameter Optimization for JavaScript, *Medium*, https://medium.com/@martin_stoyanov/hpjs-hyperparameter-optimization-for-javascript-8f78aa7a3368.

WebGL (2019) https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

World Health Organization (2019) http://apps.who.int/flumart/Default?ReportNo=16

Wolfram, S. (2017a) *An Elementary Introduction to the Wolfram Language, Second Edition*. Wolfram Media.

Wolfram, S. (2017b) Machine Learning for Middle Schoolers. Stephen Wolfram blog. http://blog.stephenwolfram.com/2017/05/machine-learning-for-middle-schoolers/.