Thesis Proposal: Using Student Learning Based on Fluency for the Learning Rate in a Deep Convolutional Neural Network for the Gamification of Information Security Awareness Training

Abien Fred Agarap
Department of Computer Science
Adamson University

March 24, 2017
Abstract

This is a proposal for mathematically determining the learning rate to be used in a deep supervised convolutional neural network (CNN), based on student fluency. The CNN model shall be tasked to imitate how students play the game “Packet Attack”, a form of gamification of information security awareness training, and learn in the same rate as the students did. The student fluency shall be represented by a mathematical function constructed using natural cubic spline interpolation, and its derivative shall serve as the learning rate for the CNN model. If proven right, the results will imply a more human-like rate of learning by machines.
1 Introduction

Chen & Yi (2017) successfully developed a deep convolutional neural network (CNN) that teaches a computer to play Super Smash Bros and Mario Tennis (both ran by using a Nintendo 64 emulator). However, like other neural networks, the learning rate in their model was a pre-programmed numerical constant. This research proposes to use a series of existing methods to mathematically determine what is the learning rate to be used in a neural network.

First, the aim is to measure the learning, based on fluency, of the players who will be subjected to the gamification of information security awareness training. In this instance, the game to be played is about network packets, i.e. determining which is a “bad” packet and a “good” packet, named Packet Attack.

To accomplish the above goal, the data consisting of the time of gameplay of players (x-axis) and the number of correct responses (fluency in this context, y-axis) shall be the basis of constructing a mathematical function which will show the relationship between the data.

The mathematical function shall be constructed using cubic spline interpolation since the graph of learning of individuals are usually not “smooth”, or in mathematical context, not differentiable at some points. The cubic spline interpolation produces a piecewise-polynomial function which gives a mathematical function at a specified interval like \([a,b]\) (see Figure 1). Hence, “smoothening” the graph of learning (see Figure 2).

In the present research, it is proposed to use the rate of change (derivative) of the function from cubic spline interpolation as the learning rate of the CNN. The learning rate in a neural network is crucial for it is the numerical entity which dictates how fast a machine learns a specified task. For the training of the CNN to learn how the human player plays, a close approach to what Chen & Yi (2017) did in their study, the gameplay
frames (760x760 px images) and the coordinates of the packets and aim of the player shall be used as input. This approach shall be done to all players that will be involved in the study, as Lewandowsky & Farrell (2011)\textsuperscript{13} pointed out that \textit{individual fitting} is more advantageous than generalizing.

The implication of using the learning (based on fluency in this study) as the basis for the learning rate in a neural network, specifically, deep convolutional neural network is significant for the point that not only does the machine learns how to play like a human, but also, learn in the same rate like humans do.
2 Literature Review

Constructing a mathematical function that represents the relationship in a data can be accomplished through curve fitting\cite{13, 20}, which may be done using interpolation or regression. The former is a process of approximating an unknown function $f(x)$ which represents the relationship in a data (e.g. $x$ and $y$), such that $f(x) = y$ \cite{4, 5, 14, 16, 18, 21}, and is the process that shall be implemented in this study.

In the studies that came before, aggregating (averaging) data was the convention in cognitive modeling, in order to fit the power function\cite{10, 13, 20} - the law of learning. However, Lewandowsky & Farrell (2011)\cite{13} pointed out that fitting models to individual participants is advantageous for it may reveal some limitations of the applicability of a model, i.e. a model may fit some subjects, but not others. Hence, in the present paper, the aforementioned method, fitting individual participants, shall be implemented. To accomplish this, the curve fitting process shall be done on each individual data.

Since the power function is a continuous function, which in geometrical context, means that the function is “smooth”\cite{10}, one of the suited interpolation algorithms for such a case is the cubic spline interpolation\cite{4, 5, 16, 18, 21}. The said algorithm was proposed by Schoenberg (1971)\cite{22} which has the benefit of constructing a “smooth” interpolating function (called an interpolant), which in turn is continuously differentiable and also has a continuous second derivative. The differentiability of the function is critical due to the conditions (4-5) that shall be stated later. The cubic spline interpolation algorithm was selected for this study for its acceptable computational complexity among others\cite{4, 5}, which is $O(n)$.

The cubic spline interpolation has two kinds\cite{4, 5, 16, 18, 21}: (i) natural (or free) boundary, and (ii) clamped boundary. In the present paper, the natural spline interpolation algorithm shall be used since the clamped spline interpolation algorithm requires to have more information about the function to be approximated, i.e. either the value of derivative at the endpoints or an accurate approximation of those values. While for this study, only the data on student fluency in the gamification of information security awareness training will be available, i.e. the number of correct responses ($y$-axis) at a given time ($x$-axis), with no more additional information about the function to be approximated.

Defining the cubic spline interpolation, given a function $f$ defined on an interval $[a, b]$, and a set of nodes, $a = x_0 < x_1 < ... < x_n = b$, a cubic spline interpolant $S$ for $f$ is a function that must satisfy the following conditions\cite{4, 5, 16, 18, 21, 22}:

1. $S(x)$ is a cubic polynomial, denoted by $S_j(x)$, on an interval $[x_j, x_{j+1}]$, for each $j = 0, 1, ..., n$. 


2. \( S_j(x_j) = f(x_j) \) and \( S_j(x_{j+1}) = f(x_{j+1}) \), for each \( j = 0, 1, ..., n - 1 \).

3. \( S_{j+1}(x_{j+1}) = S_j(x_{j+1}) \), for each \( j = 0, 1, ..., n - 2 \).

4. \( S'_{j+1}(x_{j+1}) = S_j'(x_{j+1}) \), for each \( j = 0, 1, ..., n - 2 \).

5. \( S''_{j+1}(x_{j+1}) = S_j''(x_{j+1}) \), for each \( j = 0, 1, ..., n - 2 \).

6. \( S''(x_0) = S''(x_n) = 0 \) for natural spline interpolation.

To construct the cubic spline interpolant for function \( f \), the conditions stated above must be applied to the cubic polynomials, 

\[
S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \tag{1}
\]

for each \( j = 0, 1, ..., n - 1 \), where \( a_j, b_j, c_j \), and \( d_j \) are the unknown polynomial coefficients to be solved using crout factorization for tridiagonal linear system algorithm from linear algebra\[4, 5, 16\]. The algorithm is as follows:

(Note: \( S(x) = S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \) for \( x_j \leq x \leq x_{j+1} \).)

**INPUT:** \( n; x_0, x_1, ..., x_n; a_0 = f(x_0), a_1 = f(x_1), ..., a_n = f(x_n) \).

**OUTPUT:** \( a_j, b_j, c_j, d_j \) for \( j = 0, 1, ..., n - 1 \).

1. For \( i = 0, 1, ..., n - 1 \), set \( h_i = x_{i+1} - x_i \)

2. For \( i = 1, 2, ..., n - 1 \), set 

\[
\alpha_i = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})
\]

3. Set \( \ell_0 = 1 \);
   \( \mu_0 = 0 \);
   \( z_0 = 0 \).

4. For \( i = 1, 2, ..., n - 1 \),
   set \( \ell_i = 2(x_{i+1} - x_{i-1}) - h_{i-1}\mu_{i-1} \);
   \( \mu_i = h_i/\ell_i \);
   \( z_i = (\alpha_i - h_{i-1}z_{i-1})/\ell_i \).

5. Set \( \ell_n = 1 \);
   \( z_n = 0 \);
   \( c_n = 0 \).

6. For \( j = n - 1, n - 2, ..., 0 \),
   set \( c_j = z_j - \mu_jc_{j+1} \);
   \( b_j = (a_{j+1} - a_j)/h_j - h_j(c_{j+1} + 2c_j)/3 \);
   \( d_j = (c_{j+1} - c_j)/(3h_j) \).
7. OUTPUT \((a_j, b_j, c_j, d_j\) for \(j = 0, 1, ..., n - 1)\);
TERMINATE.

Once a data set such as \(\{x, y\}_{x, y \in \mathbb{R}}\) has been processed using the algorithm defined above, its interpolant \(S(x)\) can be constructed following the form of equation (1).

Let the following be an example: Using
\[
D = \{(0, 0), (1, 0.7), (2, 0.750241), (3, 0.781286), (4, 0.804089)\}
\]
as the data set, defined by an unknown function \(f(x)\), to be interpolated. Implementing the algorithm above; \(n = 4\), \(h_0 = h_1 = h_2 = h_3 = 1\), \(a_0 = 0\), \(a_1 = 0.7\), \(a_2 = 0.750241\), \(a_3 = 0.781286\), and \(a_4 = 0.804089\). Thus, the following are the solutions to finding the unknown constants:

\[
c_3 = z_3 - \mu_3 c_4 = -0.037318124999999945 - (0.26785714285714285 \cdot 0) = -0.037318124999999945
\]
\[c_3 \approx -0.037318\]
\[c_2 = z_2 - \mu_2 c_3 = 0.11459499999999998 - (0.26666666666666666 \cdot -0.037318124999999945) = 0.12454649999999998\]
\[c_2 \approx 0.124546\]
\[c_1 = z_1 - \mu_1 c_2 = -0.487319249999999987 - (0.25 \cdot 0.124546499999999987) = -0.51845587499999998\]
\[c_1 \approx -0.518456\]
\[b_3 = a_4 - a_3 - \frac{h_3 \cdot (c_4 + 2 \cdot c_3)}{3} = \frac{0.804089 - 0.781286 - 1.0 \cdot (0 + 2 \cdot -0.037318124999999945)}{3} = 0.04768174999999998\]
\[b_3 \approx 0.047682\]
\[b_2 = a_3 - a_2 - \frac{h_2 \cdot (c_3 + 2 \cdot c_2)}{3} = \frac{0.781286 - 0.750241 - 1.0 \cdot (-0.037318124999999945 + 2 \cdot 0.124546499999999987)}{3} = -0.03954662499999993\]
\[b_2 \approx -0.039547\]
\[ b_1 = \frac{a_2 - a_1}{h_1} - \frac{h_1 \cdot (c_2 + 2 \cdot c_1)}{3} = \frac{0.750241 - 0.7}{1.0} - \frac{1.0 \cdot (0.1245464999999987 + 2 \cdot -0.518455874999998)}{3}, \]

\[ b_1 \approx 0.354363 \]

\[ b_0 = \frac{a_1 - a_0}{h_0} - \frac{h_0 \cdot (c_1 + 2 \cdot c_0)}{3} = \frac{0.7 - 0.0}{1.0} - \frac{1.0 \cdot (0.124546499999998 + 2 \cdot 0.0)}{3}, \]

\[ b_0 \approx 0.872819 \]

\[ d_3 = \frac{c_4 - c_3}{3 \cdot h_3} = \frac{0 + 0.03731812499999945}{3 \cdot 1}, \]

\[ d_3 \approx 0.012439 \]

\[ d_2 = \frac{c_3 - c_2}{3 \cdot h_2} = \frac{-0.03731812499999945 - 0.1245464999999987}{3 \cdot 1}, \]

\[ d_2 \approx -0.053955 \]

\[ d_1 = \frac{c_2 - c_1}{3 \cdot h_1} = \frac{0.1245464999999987 + 0.518455874999998}{3 \cdot 1}, \]

\[ d_1 \approx 0.214334 \]

\[ d_0 = \frac{c_1 - c_0}{3 \cdot h_0} = \frac{-0.518455874999998 - 0.0}{3 \cdot 1}, \]

\[ d_0 \approx -0.172819 \]
Figure 3: The graph of the Cubic Spline Interpolant (piecewise-polynomial) \( S(x) \) for the data set \( D \) and the original function \( 0.7x^{0.1} \) (graphed using Desmos Calculator).

Substituting the coefficients \((a, b, c, \text{ and } d)\) to the cubic polynomial \( S_j(x) \) then gives the following interpolant:

\[
S(x) = \begin{cases} 
0 + 0.872819x + 0.172819x^3 & \text{for } x \in [0, 1], \\
0.7 + 0.354363(x - 1) - 0.518456(x - 1)^2 + 0.214334(x - 1)^3 & \text{for } x \in [1, 2], \\
0.750241 - 0.039547(x - 2) + 0.124546(x - 2)^2 - 0.053955(x - 2)^3 & \text{for } x \in [2, 3], \\
0.781286 + 0.047682(x - 3) - 0.037318(x - 3)^2 + 0.012439(x - 3)^3 & \text{for } x \in [3, 4] 
\end{cases}
\]

Figure 3 shows the agreement between the interpolant \( S(x) \) and the actual function \( 0.7x^{0.1} \).

To simulate the “learning” of the individual (test subject), an artificial neural network (ANN) shall be used. An ANN is a computational model of the brain that demonstrates how neurons pass information from one to another \([9, 11, 13, 15, 17]\). The simulation shall involve the use of the derivative of the interpolant, denoted as \( S'(x) \), as the neural network’s learning rate. In ANN, the learning rate is a numerical constant which is conventionally pre-programmed \([2, 9, 11, 13, 17, 19, 23]\). The use of the learning rate in a neural network is to dictate how fast the machine learns, or in machine learning terminology, how fast the machine converges \([2, 9, 11, 13, 17, 19, 23]\). Basically, the learning rate may be described as a rate of change in the learning, hence, the use of the derivative (which is essentially rate of change in a function) \([3, 16]\).
As for the neural network model, Chen & Yi (2017) used an ANN, specifically, a deep supervised convolutional neural network (CNN) that “learns” how to play a video game by “imitating” how a human plays. The said CNN is based on the original AlexNet paper with some modifications. To get the data for the training of their CNN, they ran Mario Bros and Mario Tennis using a Nintendo 64 emulator, and took screen captures of the player’s gameplay. The screen capture contained the game window frames which they used as the input for the CNN, and the keyboard window which contains the keys pressed by the player that they consequently used as the “ground truth” or actual value. The input (game window frames) and the “ground truth” was used to compute for the accuracy of their CNN, that is simply put, the difference between the “ground truth” and the input. Since the researchers have concluded that their CNN is optimal, and they were able to achieve good results on complex games with less training time and less training data, the same model may also prove advantageous for the present study. The following is the design of their primary CNN model with some modifications in their succeeding two CNN models:

**Single Frame CNN**

1. **INPUT:** 128 x 128 x 3
2. **CONV7:** 7 x 7 size, 96 filters, 2 stride
3. **ReLU:** \( \max(x_i, 0) \)
4. **NORM:** \( x_i = \frac{x_i}{(k + (\alpha \sum_j x_j^2))} \)
5. **POOL:** 3 x 3 size, 3 stride
6. **CONV5:** 5 x 5 size, 256 filters, 1 stride
7. **ReLU:** \( \max(x_i, 0) \)
8. **POOL:** 2 x 2 size, 2 stride
9. **CONV3:** 3 x 3 size, 512 filters, 1 stride
10. **ReLU:** \( \max(x_i, 0) \)
11. **CONV3:** 3 x 3 size, 512 filters, 1 stride
12. **ReLU:** \( \max(x_i, 0) \)
13. **CONV3:** 3 x 3 size, 512 filters, 1 stride
14. **ReLU:** \( \max(x_i, 0) \)
15. **POOL:** 3 x 3 size, 3 stride
16. FC: 4096 Hidden Neurons
17. DROPOUT: $p = 0.5$
18. FC: 4096 Hidden Neurons
19. DROPOUT: $p = 0.5$
20. FC: 30 Output Classes

For their second CNN, they implemented the *early integration CNN*, having the same 2-20 layers, but with a different input layer:

1. INPUT: 128 x 128 x 12

Whereas the third CNN, the *late integration CNN*, can be seen in the high-level overview, see Figure [4]. The input for the *late integration CNN* are game frames with $\frac{1}{6}$ seconds interval between them. For processing, instead of concatenating the data (see Figure [3] for comparison) before the first layer, the frames were sent through their own independent CNN layers before merging the activations at the end through a series of layers.

Figure 4: Figure from [7]: **Late Integration.** Using this model, every game frame were separately processed by the AlexNet architecture defined in layers 2-15 of *single frame CNN*, and layers 16-20 of *early integration CNN*. 
Figure 5: Figure from [7]: Model Selection. The researchers chose among their three CNN model candidates, and ultimately selected the last one: (1) single frame, (2) early integration, and (3) late integration.
The defined CNN model above was implemented by the researchers using Theano. Since the researchers have proposed their CNN as a “portable deep learning” model, only minimal modifications to their specifications shall be introduced in this study; namely, the input specifications (128 x 128 x 3 represents the screen resolution and color profile used, which was RGB), and the hyperparameters (which includes the learning rate $\alpha$).

3 Statement of the Problem

Learning rate is a pre-programmed numerical constant in artificial neural networks, the computational model of the human brain[2, 9, 11, 13, 17, 19, 23]. The learning rate dictates how fast the neural network learns, or in machine learning context, “converges”.

The present paper proposes to mathematically determine what will be the learning rate for the neural network by using the learning (based on fluency[6]) of human players which will be subjected to the gamification of information security awareness training. The implication of this hypothesis is that machines will be able to learn in the same rate like humans do.

4 Proposed Methodology

4.1 Data Collection

The data to be collected for this study will be coming from a series of gameplay by students, which pertains to the time of gameplay ($x$-axis) and the number of correct responses (fluency, $y$-axis). The students for this study shall be subjected to the gamification of information security awareness training which at the moment has only one game: packet attack, a game where the player will learn to distinguish which is a “bad” packet and a “good” packet. Like how Chen & Yi (2017)[7] proceeded with their data collection, this study shall use the game frames
as input to the CNN. The difference, however, is the “ground truth”. In
the present study, the game is undergoing a modification to record the co-
ordinates of the player’s aim at the packets (both “bad” and “good”) as the
“ground truth”.

For a dataset needed to train the CNN in this study, it is proposed to ask
forty (40) students/players who are taking any computing-related degrees
at the university, to play the Packet Attack game for five days. Thus, it is
projected to give 2,160,000 frames if the playing time per day would be 30
minutes (with a frame captured every $\frac{1}{6}$s) over 5 days. The following is the
computation for the frames:

\[
30 \text{ mins.} \rightarrow 1800 \text{ sec.} \\
1800 \text{ sec.} \cdot 6 = 10800 \text{ frames} \\
10800 \text{ frames} \cdot 40 \text{ players} = 432000 \text{ frames} \\
432000 \text{ frames} \cdot 5 \text{ days} = 2160000 \text{ frames}
\]

As for the scoring, let the following formula be used:

\[
\text{fluency} = \frac{Y}{N} \cdot 100\%
\]

where $Y$ is the number of bad packets destroyed and good packets which
passed through (an addition of the number of two packets), and $N$ is the
number of packets spawned.

In addition to the dataset to be collected, the sprites representing a good
packet (Figure 7) and a bad packet (Figure 8) in order for the machine to
distinguish between the two shall also be recorded.

4.2 Natural Cubic Spline Interpolation $S(x)$

The algorithm defined in the Literature Review section shall be implemented
using Python, the following is the source program and is also available at
GitHub Gist.
#!/usr/bin/env python3

def main():
    h = []; alpha = []; ell = []; mu = []; z = []
x = []; y = []; b = []; d = []
size = int(input('Enter size (n) of data set(x,y): '))
n = size - 1

    for i in range(0, size):
        x.append(float(input('Enter value for x_{:0}{}{}: '.format(i, '', i))))
        y.append(float(input('Enter value for y_{:0}{}{}: '.format(i, '', i))))
    c = list(x)

    for i in range(0, n):
        h.append(x[i + 1] - x[i])

    for i in range(1, n):
        alpha.append(((3 / h[i]) * (y[i + 1] - y[i])) - ((3 / h[i-1]) * (y[i] - y[i - 1])))
        ell.append(1); mu.append(0); z.append(0);

        for i in range(1, n):
            ell.append((2 * (x[i + 1] - x[i - 1])) - (h[i - 1] * mu[i - 1]))
            mu.append(h[i] / ell[i])
            z.append((alpha[i-1] - (h[i - 1] * z[i - 1])) / ell[i])

        ell.append(1); z.append(0); c[-1] = 0

    for j in range((n - 1), -1, -1):
        c[j] = (z[j] - (mu[j] * c[j + 1]))
        b.append((y[j + 1] - y[j] / h[j]) - ((h[j] * (c[j] + (2 * c[j]))) / 3))
        d.append((c[j + 1] - c[j]) / (3 * h[j]))

    for i in range(0, n):
        print('S_{:0}{}{}(x) = {} + {}x + {}x^2 + {}x^3 for x \in [{}, {}].format(i, 
        {},').format(i, y[i], b[n - i - 1], c[i], d[n - i - 1], x[i], x[i + 1]))

if __name__ == '__main__':
    main()
4.3 Differentiation of the Mathematical Function $S(x)$

To determine the derivative of the cubic spline interpolant from the previous subsection, the algorithm that will be used is based on the rudimentary - the fundamental theorem of Calculus\[^3, 16\]:

$$f'(x) = \lim_{x \to 0} \frac{f(x + \Delta x) - f(x)}{x + \Delta x},$$

where it is axiomatic, by the convention in Mathematics, that $\Delta x$ represents the change in the value of $x$.

In relation with the hypothesis in this study, take for example the following table:

<table>
<thead>
<tr>
<th>$x$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = 0.7x^{0.1}$</td>
<td>0.70</td>
<td>0.750241</td>
<td>0.781286</td>
<td>0.804089</td>
<td></td>
</tr>
<tr>
<td>$f'(x) = \frac{7}{100x^{0.1}}$</td>
<td>error 0.07</td>
<td>0.037512</td>
<td>0.026043</td>
<td>0.020102</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: The graph of $f(x)$ and $f'(x)$: The $f(x)$ bears a striking resemblance to the graph of learning of a neural network\[^2, 9, 11, 13, 17, 19, 23\] (see Figure 10 for comparison), hence, it could be inferred that if it is used as the learning rate $\alpha$ of an ANN, it may result to overfitting\[^19\].

The values of the primary derivative $f'(x)$ of the function $f(x)$ show a much closer resemblance to the learning rates used in neural networks\[^2, 9, 11, 13, 17, 19, 23\] than the values of the function itself, with an exception of
the value of $f'(x)$ being an error at 0. This is because the $f'(x)$ for the $f(x)$ in context is asymptotic, and therefore the values are approaching zero, but not exactly zero (i.e. $(-\infty, 0)$). Nevertheless, a computing (programmatic) solution may be aided to use the closest value to zero in lieu of zero.

According to Riedmiller (1994)[19] and [8], a fairly-larger amount of learning rate in a neural network tends to overfit a data. Thus, together with the verbal description of the derivative as a rate of change in a function, and its evidently smaller values, it may be established as a fit candidate as a learning rate in the neural network.

### 4.4 Deep Convolutional Neural Network (CNN) Model

In contrary with the implementation of Chen & Yi (2017)[7], the deep CNN in this study, which is based on their model, shall be implemented using TensorFlow[1] instead of Theano. As for the input and output, the modification on their specification is as follows:

1. INPUT: 760 x 760 x 3

20. FC: 4 OUTPUT Classes

Then, the rest shall be based on the layers they defined, layers 2-19. In terms of hyperparameters, the following specifications shall be used:

1. Learning Rate: The value of $S'(x)$
2. Learning Rate Decay: 0.95 every 5000 iterations
3. $L_2$ Regularization Penalty: 1e-7
4. Update Rule: Adam
5. Batch Size: 25
6. Training Time: 80 epochs over 5 days.

Note that the above hyperparameters is almost identical to [7], with the learning rate and training time being different, however. The training time will also be different for the fact that there will be forty (40) student-players, and there will be two epochs each for five days (amount of playing time by human players). Figure 11 is the high-level overview of the CNN model to be used in the present study.

Figure 11: The CNN Model to be implemented in this study is based on the Late Integration CNN Model by Chen & Yi (2017)[7].

The above model for the CNN architecture to be implemented has a modified INPUT and OUTPUT layer. Figure 6 is an example screen capture of a gameplay of Packet Attack, which will then be reduced to 760 x 760 resolution as previously mentioned. Regarding the output, it will be of four classes:

1. A good packet has been destroyed
2. A good packet went through
3. A bad packet has been destroyed
4. A bad packet went through.

The classes (2) and (4) stated above could be determined by checking if the coordinates of the sprite of those packets reached the “base” of the user,
i.e. origin point. On the other hand, classes (1) and (3) shall be coming from the Packet Attack game itself.

The following source program is the preliminary draft for the implementation of the CNN model, using TensorFlow:

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf

from tensorflow.contrib import learn
from tensorflow.contrib.learn.python.learn.estimators import model_fn as model_fn_lib

tf.logging.set_verbosity(tf.logging.INFO)

def cnn_model_fn(features, labels, mode):
    '''Model function for CNN'''
    # Input Layer: 760 x 760 x 3
    input_layer = tf.reshape(features, [-1, 760, 760, 3])

    # Conv layer #1: 7x7 size, 96 filters, with ReLU
    conv1 = tf.layers.conv3d(
        inputs=input_layer,
        filters=96,
        kernel_size=5,
        padding='same',
        activation=tf.nn.relu,
        activation=tf.nn.relu)

    # Pooling Layer #1: 3x3 size, 3 strides
    pool1 = tf.layers.max_pooling3d(inputs=conv1, pool_size=3, strides=3)

    # Conv layer #2: 5x5 size, 256 filters, with ReLU
    conv2 = tf.layers.conv3d(
        inputs=pool1,
        filters=256,
        kernel_size=5,
        padding='same',
        activation=tf.nn.relu)

    # Pooling Layer #2: 2x2 size, 2 strides
    pool2 = tf.layers.max_pooling3d(inputs=conv2, pool_size=2, strides=2)

    # Conv layer #3: 3x3 size, 512 filters, with ReLU
    conv3 = tf.layers.conv3d(
```
inputs=pool2,
filters=512,
kernel_size=3,
padding='same',
activation=tf.nn.relu
)

# Conv layer #4: 3x3 size, 512 filters, with ReLU
conv4 = tf.layers.conv3d(
    inputs=conv3,
    filters=512,
    kernel_size=3,
    padding='same',
    activation=tf.nn.relu
)

# Conv layer #5: 3x3 size, 512 filters, with ReLU
conv5 = tf.layers.conv3d(
    inputs=conv4,
    filters=512,
    kernel_size=3,
    padding='same',
    activation=tf.nn.relu
)

# Pooling Layer #3: 3x3 size, 3 strides
pool3 = tf.layers.max_pooling3d(inputs=conv5, pool_size=3,
    strides=3)

# Dense Layer #1: 4096 neurons
pool3_flat = tf.reshape(pool3, [-1, 3 * 3 * 512])
dense = tf.layers.dense(inputs=pool3_flat, units=4096,
    activation=tf.nn.relu)

# Dropout: p = 0.5
dropout = tf.layers.dropout(inputs=dense, rate=0.5,
    training=mode == learn.ModeKeys.TRAIN)

# Dense Layer #2: 4096 neurons
dense2 = tf.layers.dense(inputs=dropout, units=4096,
    activation=tf.nn.relu)

# Dropout: p = 0.5
dropout2 = tf.layers.dropout(inputs=dense2, rate=0.5,
    training=mode == learn.ModeKeys.TRAIN)

# Logits Layer: 4 output classes
logits = tf.layers.dense(inputs=dropout2, units=4)
Afterwards, when the above model has been trained using the data, the measurement of accuracy of the network shall come using the following formula:

\[-\sum_i y'_i \log(y_i),\]

where $y'_i$ is the actual value or in the context of the present paper, “ground truth”, and $y_i$ is the output of the neural network. The following is the implementation of the formula above:

```python
loss = None
train_op = None

# Calculate for Loss (Accuracy)
if mode != learn.ModeKeys.INFER:
    onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32),
                               depth=10)
    loss = tf.losses.softmax_cross_entropy(
        onehot_labels=onehot_labels, logits=logits)
```

References


