CS 5824 : Final Project
"Compartment fire predictions using transpose convolutional neural networks" - Jonathan Hodges

Project by: Manuel Barros, Smridh Malhotra, Barret Lattimer, Parisha Joshi

1 Introduction

The objective of the paper was to reproduce and confirm results from a published machine learning research paper. The paper selected was "Compartment fire predictions using transpose convolutional neural networks". This paper was published in the fire safety Journal in 2019.

The chosen research paper aimed to predict temperatures and air velocities for compartment fires much faster and at lower computational cost than CFD models currently available. In order to achieve this objective, the authors used a generative model (transpose convolutional neural network (TCNN)) that was trained and tested with results of a CFD model called Fire Dynamic Simulator (FDS). The idea behind reducing the computational cost of fire prediction is directly related to the fire hazard analysis. Lower cost at fire predictions would represent the possibility of carrying out parametric studies and predictions of large structures that currently can be prohibitive. CFD models solve conservation equation of mass, momentum and energy for each grid cell of the domain, often numbering 10 5 – 107. This high spatial-temporal resolution makes CFD model simulations take hours, days or even weeks to be completed depending on the domain size. An example of spatially resolved temperature prediction by a CFD model is shown in Figure 1 in which the temperature in each cell of the domain is represented by a color. On the other hand, zone fire models or zero-dimensional zone models are also used for fire predictions. These models are more rapid than CFD models since they take advantage of the thermal stratification produced in enclosure fires. The thermal stratification is represented as one or two well mixed regions allowing for the existence of only two cells in each room: upper layer and lower layer. This representation implies that the conservation equations (mass, momentum, and energy) are solved for significant a smaller number of cells of the domain. However, fire models cannot predict local variation of properties such as temperature and flow velocity. The assumption of well mixed regions implies that the result of zone fire models is the mean value of the property in each layer. Zone fire models can provide insightful results; however, their results have lack of resolution that is required in hazard analysis. Generative modeling is a relative new topic in the machine learning community which has been implemented mainly for generation of RGB images. It

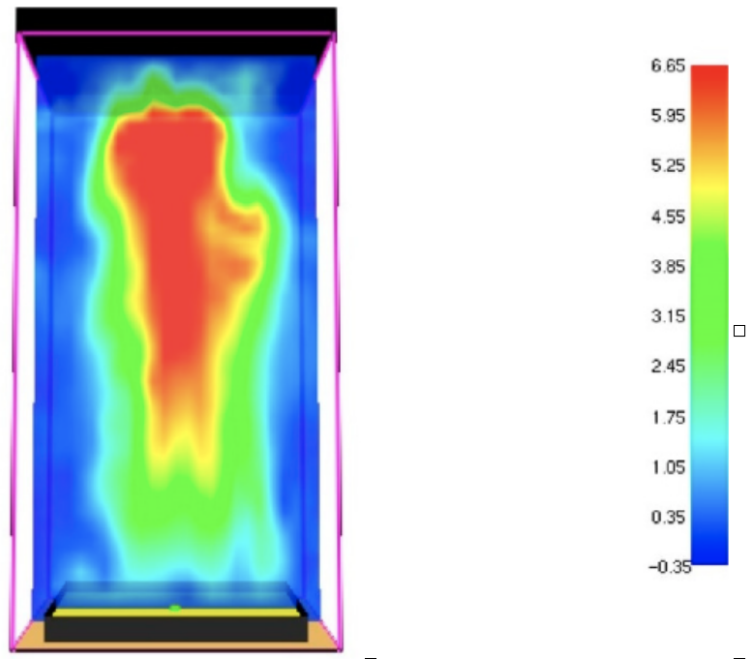


Figure 1: Example of spatially resolved temperature predicted by FDS model

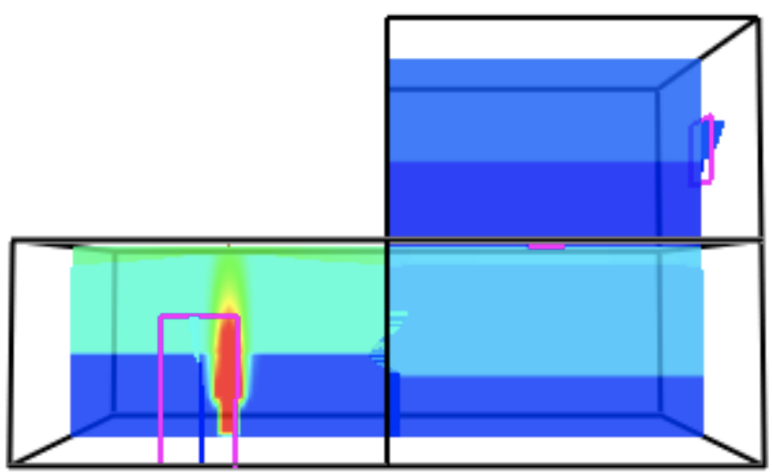


Figure 2: Fire prediction of a zone fire model in which the mean temperature of each layer for each room is shown

is important to mention that generating data, designing, and training model can be time consuming,

but a generative model is able to make rapid prediction and at low computational cost. Due to computational cost associated to CFD models and the lack of resolution of zone fire models, the authors of this research paper decided to apply generative model techniques using machine learning. This approach would contribute to reduce the computational cost of CFD models and at the same time would keep the resolution of CFD model results. The authors proposed a new compartment fire model which uses the results of a zone fire model as inputs to a generative machine learning model (TCNN) allowing for high resolution predictions of intensive properties at low computational costs for these structures. A generative model to provide spatially resolved predictions of temperature and velocity

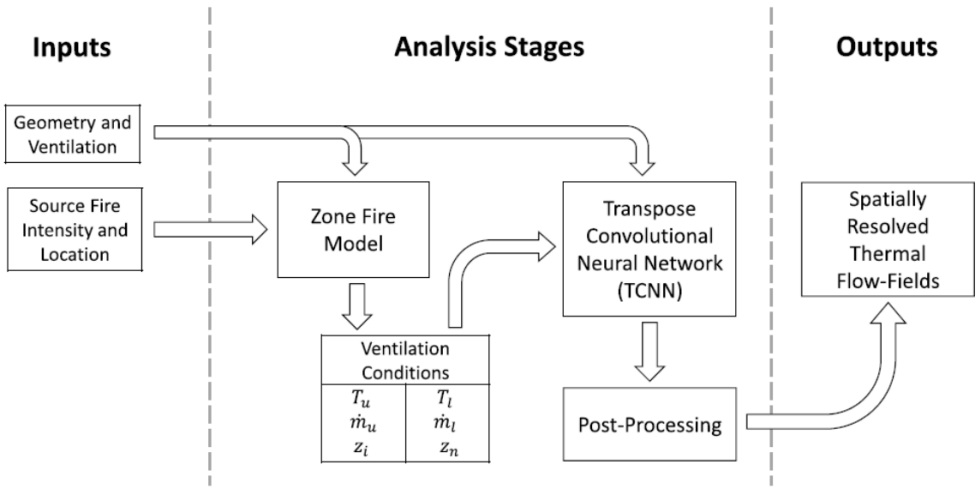


Figure 3: Methodology used in the paper

based on predictions from a zone fire model was developed. The schematic of the methodology used for this study is shown in Figure 3. In this Figure clearly can be seen that the inputs to generative model (TCNN) was composed of parameters and predictions from the zone fire model (such as mass flow rates and gas temperature at doors). The outputs were spatially resolved thermal flow fields.

1.1 Inputs, outputs and network architecture

The network architecture used in this work was a TCNN shown in Figure 4. The architecture consisted of elven hidden layers which nine are trained processing layers. The input vector was composed of 35 elements per each example. Among these 35 elements, three corresponds to the basic geometric compartment (length, width and height). The remaining 32 elements corresponds to each wall surface (describing the junction to the neighboring space for each wall) and results of the zone fire model that represents flow conditions of the ventilation. Some of the elements used in the input vector are shown in Table 1. A total of 12 output channels were predicted which represents two-dimensional slices of temperature and three-component velocity through two vertical centerlines of the compartment, and one horizontal plane 0.1 m below the ceiling (as shown in Figure 5). Each channel corresponds

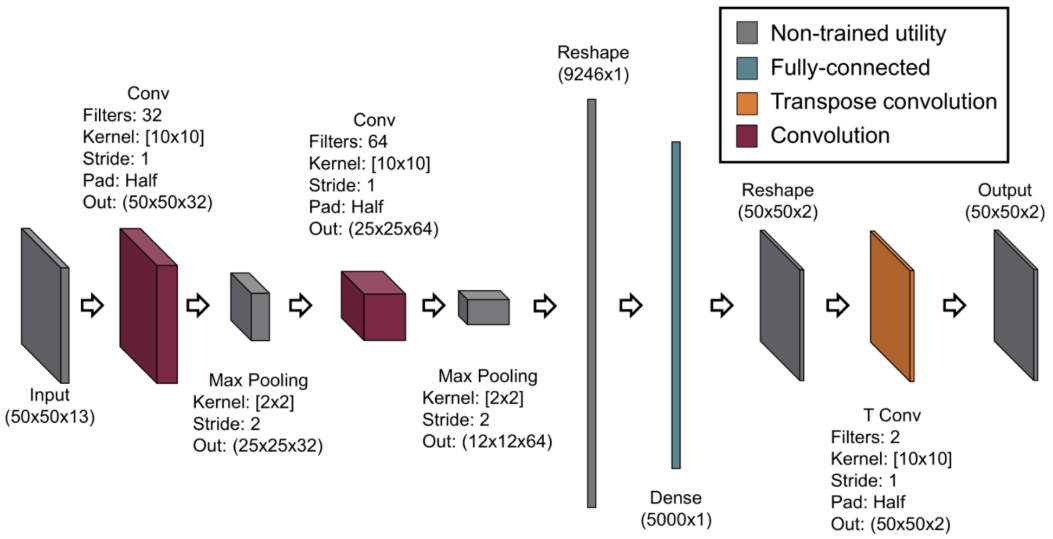


Figure 4: TCNN architecture

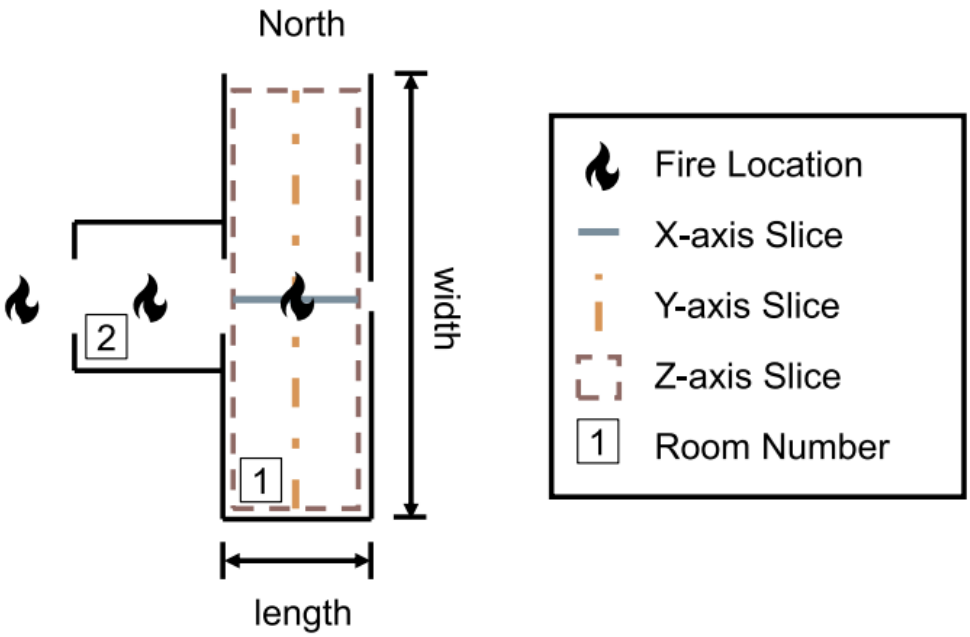


Figure 5: Location of two-dimensional slices of temperature and velocity in Room 1

to the image with the results of spatially resolved temperature, x-axis velocity (U-velocity), y-axis velocity (V-velocity), and z-axis velocity (W-velocity). Each image was composed of 50 x 50 pixels that represents the value of the property in different points of the slice. In the architecture the three dense layers (fully connected) are used to develop a high dimensional representation of the input parameters before processing through the TCNN and CNN layers. A single dropout layer was used

after the fully connected layers to prevent over-fitting during training. The number of filters and step size in the TCNN and neurons in fully connected layers were set to increase the degree of freedom from 35 in the input layer to 30,000 in the output layer. All layers in the TCNN network used leaky rectified linear unit (ReLu) activation functions. The model was trained using stochastic gradient descent with a batch size of 100. Weights and biases were initialized from a uniform distribution. The learning rate for all layers was fixed at 10-4. The cost function used for training was based on square error. The network was trained using 4990 unique scenarios for 200,000 cycles. The total time to train the network was reported around 450 h using a single NVIDIA Quadro K620 graphics card. Also, the total time for prediction of the spatially resolved temperature and velocities for 5000 scenarios was 8 s.

Table 1 Description of some elements of the input vector

Variable	Name	Description
L	Length	Total extent of room along the x-axis
W	Width	Total extent of room along the y-axis
H	Height	Total extent of room along the z-axis
S	State	Whether ventilation is open or closed
D_S	Size	Class of ventilation size. 0: 0.625 m × 1.9 m 1: 1.250 m × 1.9 m 2: Full wall
T_u	Upper layer temperature	Temperature above the interface height
T_l	Lower layer temperature	Temperature below the interface height
z_i	Interface height	Height of the hot gas layer
$\dot{m}_u$	Upper layer mass flow rate	Flow rate above the neutral plane
$\dot{m}_l$	Lower layer mass flow rate	Flow rate below the neutral plane
z_n	Neutral plane height	Height of inflection point in gas velocity

1.2 Results of original paper

333 scenarios were used to test the performance of the TCNN on new scenarios. The error was calculated between the neural network prediction and CFD prediction. For the calculation of the percent error for temperature predictions, Equation 1 was used.

$$E_{T,PD} = \frac{T_{cfd} - T_{CNN}}{T_{cfd}}; \tag{1}$$

Where ET,PD is the percent error, TTCNN is the temperature predicted by the neural network, and TCFD is the temperature predicted by the neural network. For the calculation of velocity predictions, the absolute error was calculated as shown in Equation 2 since some velocities are close to zero and the percent error was not a good metric to examine the robustness of the predictions.

$$E_{V,ABS} = V_{cfd} - V_{CNN}; \tag{2}$$

In the result section of the paper two scenarios included in the testing scenario set were chosen in order to show the proximity between predicted values and truth values. These two scenarios called test case

1 and test case 2 are shown in Figure 6. TCNN predictions of temperature and velocity compared to

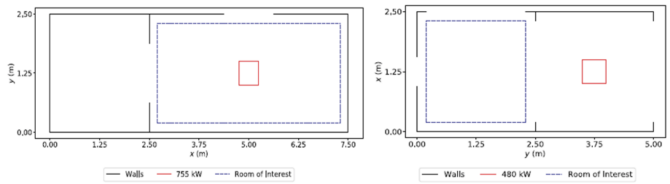


Figure 6: Test case 1 and 2

CFD simulations for testing case 1 and 2 are shown in Figs. 6-11. In each figure the rows correspond to one property where: the left image is the TCNN prediction, the left-middle column is the CFD prediction, the right-middle column compares the centerline profile, and the histograms on the right shows the error in the neural network prediction compared to the CFD prediction. For testing case 1 and 2 results shown in Figs. 9-14 temperatures percent error are typically +- 10.0 percent and velocities error within +- 0.22 m/s The discrete probability density function of temperature percent

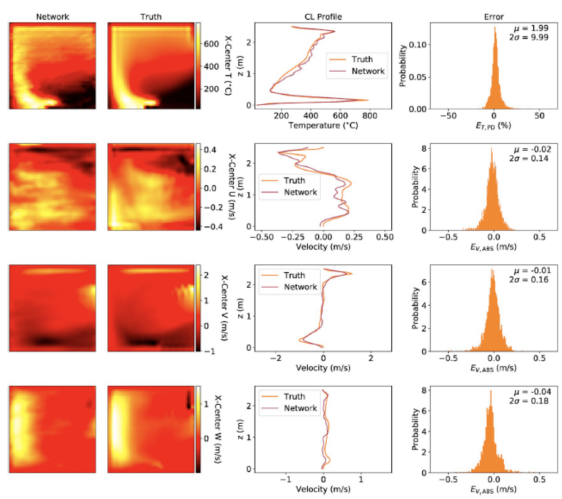


Figure 7: TCNN predictions of x-axis center-line temperature and velocity compared to CFD simulations for Testing Case 1.

error and velocity absolute error for all training and test data can be seen in Figure 13 and Figure 14.

2 Description of the procedure to reproduce the results

Input and output data for training and testing the predictions by the TCNN were provided by the authors of the paper. Two .csv files were sent to us by the authors. The first file corresponded to the input vectors (35 features) of 5000 examples. The second file corresponded to the output of

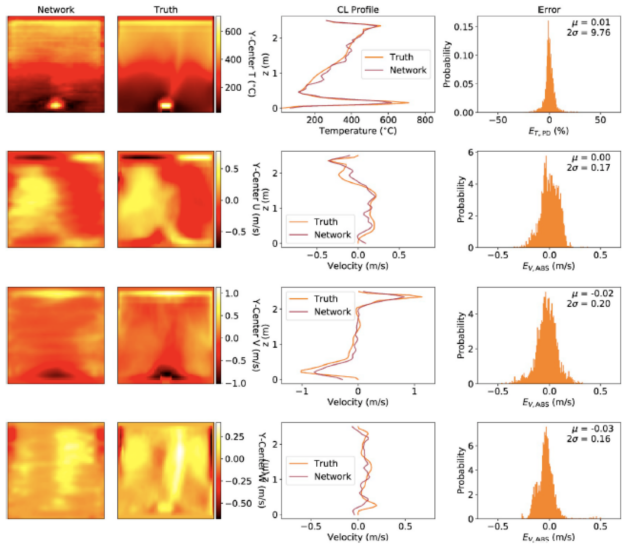


Figure 8: TCNN predictions of y-axis center-line temperature and velocity compared to CFD simulations for Testing Case 1.

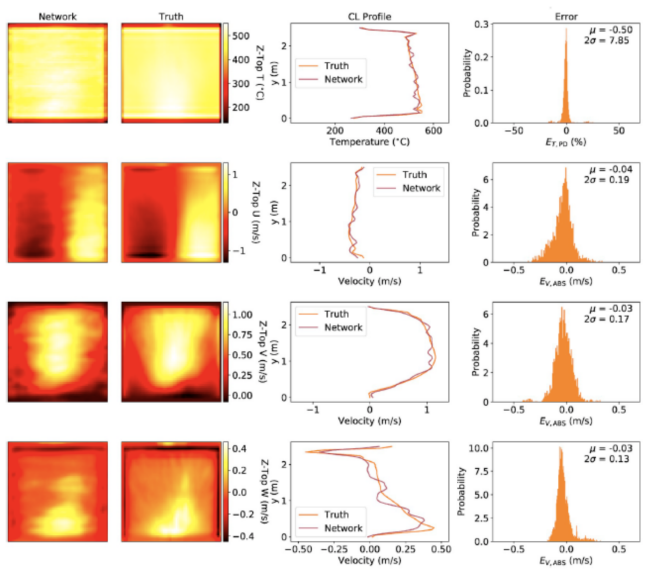


Figure 9: TCNN predictions of y-axis center-line temperature and velocity compared to CFD simulations for Testing Case 1.

each example. The output of each example was composed of a vector of 30000 elements to form 12 channels of 50x50 pixels. As recommended by the authors the TCNN was built in TensorFlow framework version 1.

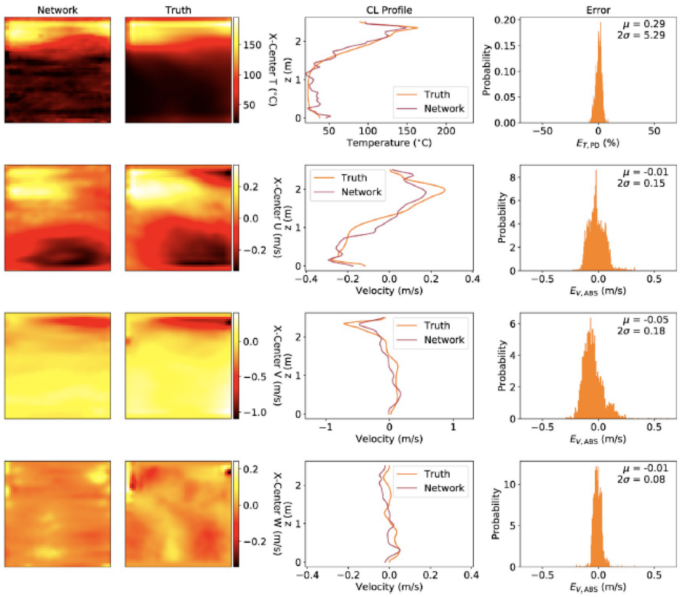


Figure 10: TCNN predictions of x-axis center-line temperature and velocity compared to CFD simulations for Testing Case 2.

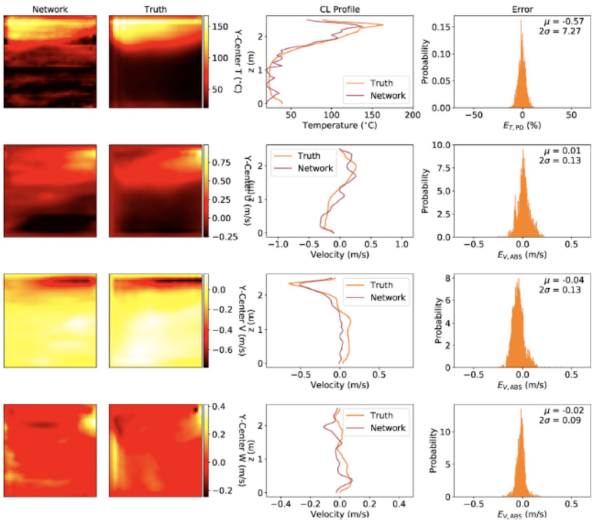


Figure 11: TCNN predictions of y-axis center-line temperature and velocity compared to CFD simulations for Testing Case 2

2.1 TensorFlow

TensorFlow is free and opensource library developed by Google, which is used to for data flow and differential programming. It is often used for machine learning applications such as neural networks.

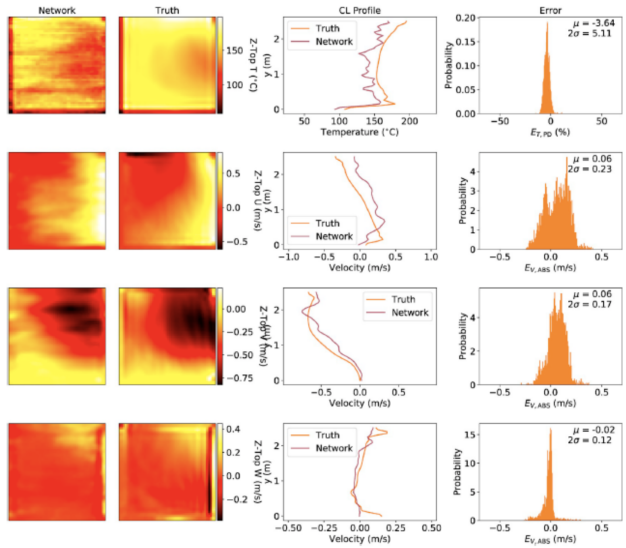
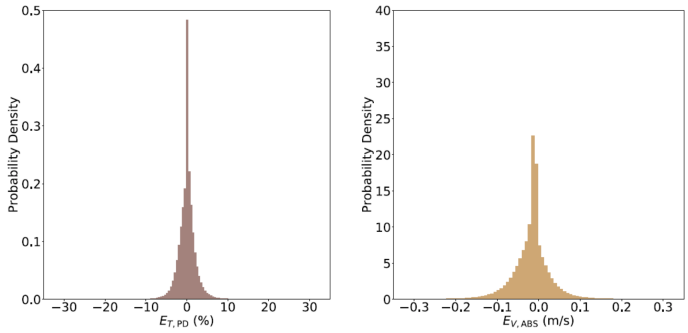


Figure 12: predictions of z-axis ceiling temperature and velocity 0.1m below the ceiling compared to CFD simulations for Testing Case 1

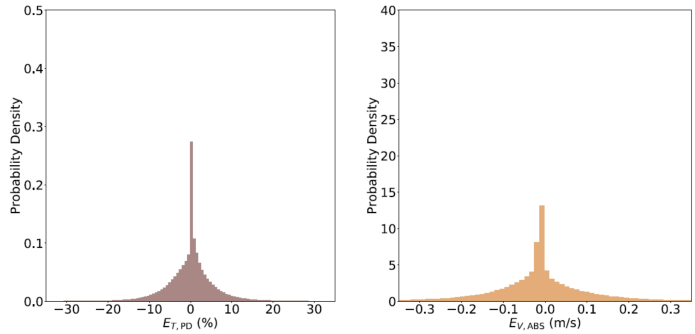


(a) Temperature percent error.

(b) Velocity absolute error.

Figure 13: Discrete probability density functions of TCNN error from training data set

TensorFlow basically provides a Frontend API for various machine learning and neural network architecture and algorithms. TensorFlow allows users to make dataflow graphs, which govern how the data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each edge is a multidimensional array known as tensor. The front-end abstractions are provided to users in Python, while the backend code is written in C++. The single biggest benefit TensorFlow provides for machine learning development is abstraction. Instead of dealing with the nitty-gritty details of implementing algorithms or figuring out proper ways to hitch the output of one function to the input of another, the developer can focus on the overall logic of the application. TensorFlow takes care of the details behind the scenes.



(a) Temperature percent error. (b) Velocity absolute error.

Figure 14: Discrete probability density functions of TCNN error from testing data set

2.2 Procedure

The Architecture of the neural network was designed as per the specifications of the paper. The Input as discussed previously are examples with 35 features, which are fed to a 11 layers neural network, with each layer performing specific function. The following are various units and layers used in the architecture.

2.2.1 Initializers and Activations:

Initializers are often used to set up the values required for various elements in the network, that are required to be generated. For this project we use two different initializer such as the Glorot uniform and random uniform initializers. The glorot uniform initializer also called Xavier uniform initializer draws samples from a uniform distribution within [-limit, limit] where limit is sqrt(6 / (fan-in + fan-out)) where fan-in is the number of input units in the weight tensor and fan-out is the number of output units in the weight tensor. We use this to initialize the kernel and the bias values. In the following lines can be seen how this initializers were used in the code for kernel and bias initialization

- kernel_initializer=tf.random_uniform_initializer(minval=-1,maxval=1),
- bias_initializer=tf.random_uniform_initializer(minval=-1,maxval=1))
- kernel_initializer=tf.compat.v1.glorot_uniform_initializer(),
- bias_initializer=tf.compat.v1.glorot_uniform_initializer()

In the context of artificial neural networks, the rectifier is an activation function defined as the positive part of its argument:

$$f(x) = x^{+} = max(0, x); \tag{3}$$

where x is the input to a neuron. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering. A unit employing the rectifier is also called a rectified linear unit

(ReLU). For this project we used leaky ReLU as an activation function which allow a small positive gradient when the unit is not active. The alpha value for the Leaky ReLU Activation function was set to 0.2. Therefore, In the code an auxiliary function of leaky ReLU was used as shown in the

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise.} \end{cases}$$

following line: def my_leaky_relu(x):
return tf.nn.leaky_relu(x, alpha=0.2)

2.2.2 Dense Layer:

A dense layer represents a matrix vector multiplication. (assuming your batch size is 1) The values in the matrix are the trainable parameters which get updated during backpropagation.($u^T.W, W$) So, we get a m dimensional vector as output. A dense layer thus is used to change the dimensions of your vector. Mathematically speaking, it applies a rotation, scaling, translation transform to your vector. In the following lines some examples of dense layers used in the code are shown:

- layer_1 = tf.compat.v1.layers.dense(inputs=features["x"],
  activation=my_leaky_relu, units=625,
  use_bias=True,
  kernel_initializer=tf.compat.v1.glorot_uniform_initializer,
  bias_initializer=tf.compat.v1.glorot_uniform_initializer())

- layer_3 = tf.compat.v1.layers.dense(inputs=layer_2, units=1250,
  activation=my_leaky_relu,
  use_bias=True,kernel_initializer=tf.compat.v1.glorot_uniform_initializer()
  ,bias_initializer=tf.compat.v1.glorot_uniform_initializer())

2.2.3 Dropout Layer:

A dropout layer is used for regularization where you randomly set some of the dimensions of your input vector to be zero with probability p. A dropout layer does not have any trainable parameters i.e. nothing gets updated during backward pass of backpropagation. To ensure that expected sum of vectors fed to this layer remains the same if no dropout was applied, the remaining dimensions which are not set to zero are scaled by (1/p). We used a dropout value of p = 0.75 and is only to be done at the time of the training as shown in the following:

- layer_2 = tf.compat.v1.layers.dropout(layer_1, rate=0.75,
  training=(mode == tf.estimator.ModeKeys.TRAIN))

2.2.4 Convolution Layer:

The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input. Kernel Size for this project is 5X5. Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map. Stride controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and to large output volumes. When the stride is 2 then the filters jump 2 pixels at a time as they slide around. Similarly, for any integer S¿0, a stride of S causes the filter to be translated by S units at a time per output. In practice, stride lengths of S ¿= 3 are rare. The receptive fields overlap less, and the resulting output volume has smaller spatial dimensions when stride length is increased. Sometimes, it is convenient to pad the input with zeros on the border of the input volume. Padding provides control of the output volume spatial size. Sometimes it is desirable to exactly preserve the spatial size of the input volume. 'Same' padding means a pad of half the value is added, while valid means no pad. Some convolutional layers used in the TCNN architecture code are shown in the following lines:

- layer_7 = tf.compat.v1.layers.conv2d(inputs=layer_6, filters=48,
  kernel_size=[5,5],
  strides=1, padding="same",activation=my_leaky_relu,use_bias=True,
  kernel_initializer=tf.compat.v1.glorot_uniform_initializer(),
  bias_initializer=tf.compat.v1.glorot_uniform_initializer())

- layer_9 = tf.compat.v1.layers.conv2d(inputs=layer_8, filters=24,
  kernel_size=[5,5], strides=1,
  padding="same",activation=my_leaky_relu,
  use_bias=True,kernel_initializer=tf.compat.v1.glorot_uniform_initializer(),
  bias_initializer=tf.compat.v1.glorot_uniform_initializer())

2.2.5 Transposed Convolution Layer:

It is used to up-sample optimally while maintaining the relationships between the data. The transposed convolution operation forms the same connectivity as the normal convolution but in the backward direction. Also, the weights in the transposed convolution are learnable. So we do not need a predefined interpolation method. The main point is that the association between the input and the

output is handled in the backward fashion compared with a standard convolution matrix (one-to-many rather than many-to-one association). As such, the transposed convolution is not a convolution. But we can emulate the transposed convolution using a convolution. We up-sample the input by adding zeros between the values in the input matrix in a way that the direct convolution produces the same effect as the transposed convolution. You may find some article explains the transposed convolution in this way. However, it is less efficient due to the need to add zeros to up-sample the input before the convolution. Some transpose convolutional layers used in the TCNN architecture code are shown in the following lines

- layer_6 = tf.compat.v1.layers.conv2d_transpose(inputs=layer_5, filters=48, kernel_size=[5,5], strides=2,padding="valid",activation=my_leaky_relu,use_bias=True, kernel_initializer=tf.compat.v1.glorot_uniform_initializer(), bias_initializer=tf.compat.v1.glorot_uniform_initializer()
- layer_8 = tf.compat.v1.layers.conv2d_transpose(inputs=layer_7,filters=24, kernel_size=[5,5], strides=2,padding="valid",activation=my_leaky_relu,use_bias=True, kernel_initializer=tf.compat.v1.glorot_uniform_initializer(), bias_initializer=tf.compat.v1.glorot_uniform_initializer()

2.3 Training

We train this neural network designed above with certain parameters as follows:

1. Learning Rate: different learning rates were used such as 10-4, 10-10 and 10-15
2. Batch Size: The batch size is kept as 100 samples
3. Cycles: We trained it for 200,000 cycles, with each cycle going over the 5000 samples.
4. Eps: In order to avoid data to be turning to NaN we add a small epsilon of 10-12 to the data after each cycle.
5. Loss Function: We use root mean square difference as loss function, using which the model is trained in order to reduce this value over the length of 200,000 cycles.

3 Results

3.1 Performance Metrics

The paper talks in details about three main parameters to measure the performance of the model.Where variables t_p, f_p and f_n represent 'correctly identified fire pixels', 'falsely identified fire pixels' and 'failed to identify fire pixels' respectively. The precision of over prediction errors is represented by P, Sensitivity or measure of underprediction error is represented by S and F-Measure or overall performace of the model is represented by F.Equations for which are:

- $P = \frac{t_p}{t_p+f_p}$
- $S = \frac{t_p}{t_p+f_n}$
- $F = \frac{2.P.S}{P+S}$

The attempts to achieve the best scores were made by trial and test method.It takes around 49 hours to complete the training process for 200000 cycles for any trial and three such trials were made in order to achieve similar results to the original paper.

Trial 1 : Changing Initializer

The first trial was with naive model with uniform initializers in keras with values ranging from -1

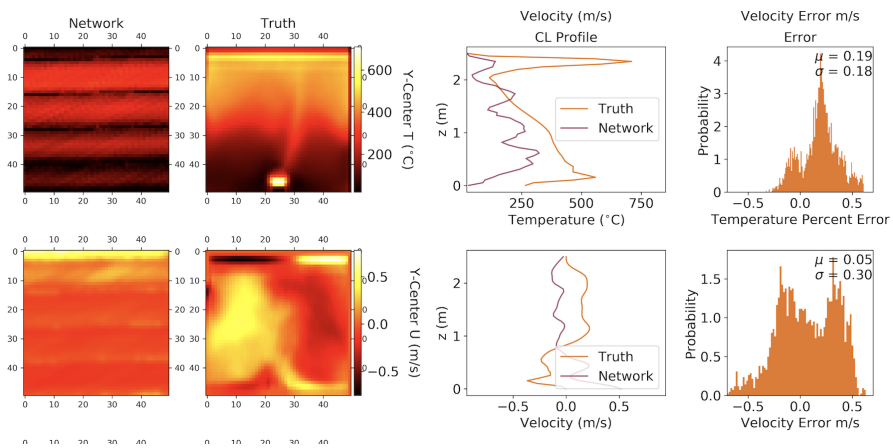


Figure 15: Trial 1

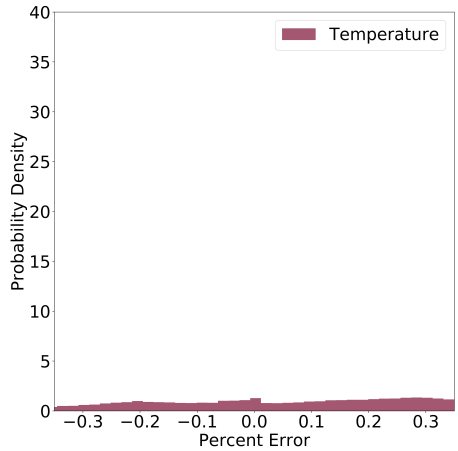


Figure 16: Trial 1 Probability Density function

to 1 and learning rate of 10^{-4}.We followed the model given in Figure 4 to make model layers.The results shown in Figure 15 shows the first results that we got with the most basic architecture. The

Velocity and temperature curvs do not seem to converge, There was a lot of distortion, and the Error distribution, both absolute and percentage, was asymmetric(Right most). The blocks shown in the left most lanes do not seem to be same(first column is network result and second column is the target).The third column shows the calculated and true values of temperature and velocity with respect to z(m).The probability density function shown in Figure 16 has almost null area of density with restpect to the percentage error, Which should look like a bell curve for normal distribution functions. We discovered that initializing the weights in a model plays a very important part in convergence for the optimum results.As explained in section 2.2.1. the glorot initialzer performed best with the performance metrics.

Trial 2 : Loss function

Results after changing the initializers looks much better with error distribution function(Rightmost

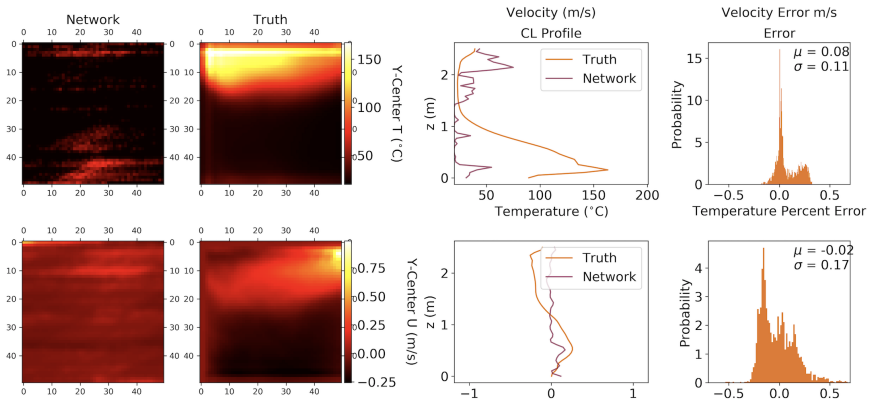


Figure 17: Trial 2

column) as the first two columns have very similar values.The initializing values ranged from -0.1 to 0.1 and learning rate was unchanged.Although the curves in the third column do not seem to converge yet, for which We were advised to change the loss function. The original paper makes use of root mean squared error for calculating loss. Instead of using the sqrt(keras.losses.mean_squared_error(y_true, y_pred), keras.losses.mean_squared_error(y_true, y_pred)**(1/2) ended up solving the problem.

Trial 3 : Final Result

The final run gave spectacular results with exact convergence of the velocity and temperature curves and a bell shaped normal distribution curve for the probability density function.Figure 19 shows a loss curve which decreases exponentially with training epochs. Conclusively, The training model used for compartment fire prediction was ran and tested on Windows based system with GTX 2080, each run takes around 48 hours for completing the training. We were able to replicate and verify the results with little discrepancies.

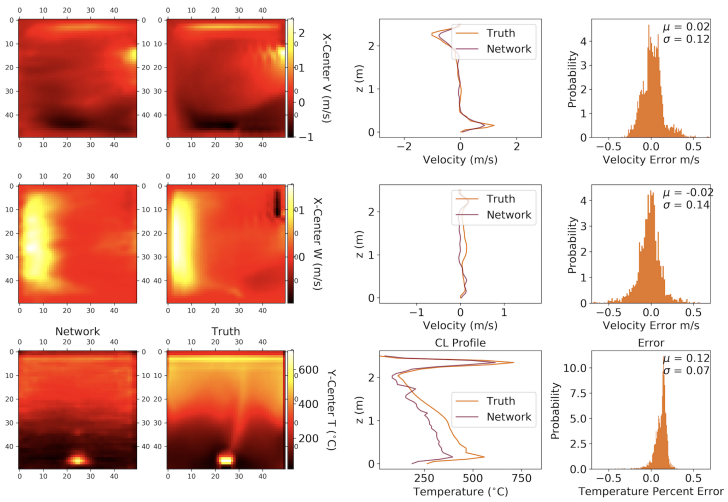


Figure 18: Final Results

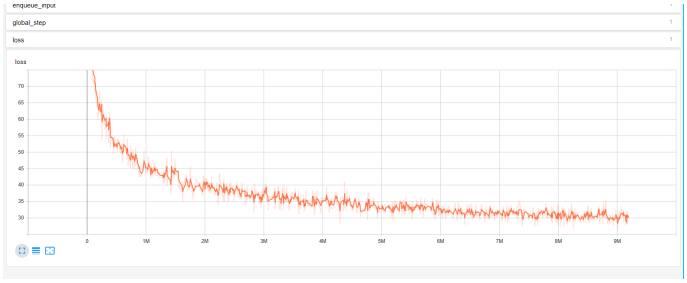
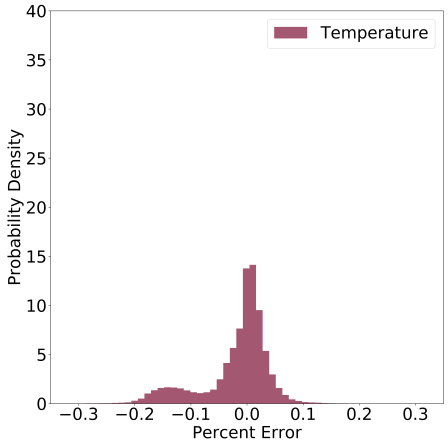
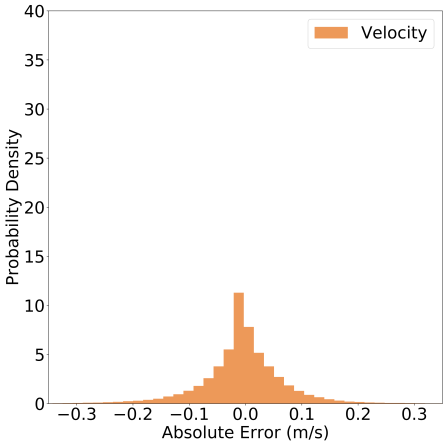


Figure 19: Tensorboard curve for loss function



(a) Final Results: Temperature Percentage error



(b) Final Results: Velocity absolute error

References

[1] Compartment fire predictions using transpose convolutional neural networks
`Jonathan L. Hodges, Brian Y. Lattimer, Kray D. Luxbacher`

[2] Learning to Generate Chairs, Tables and Cars with Convolutional Networks
`Alexey Dosovitskiy, Jost Tobias Springenberg, Maxim Tatarchenko, Thomas Brox`

[3] Wildland Fire Spread Modeling Using Convolutional Neural Networks
`Jonathan L. Hodges , and Brian Y. Lattimer, Jensen Hughes, 2020 Kraft Drive, Suite 3020, Blacksburg, VA 24060, USA`

[4] Keras Documentation
`https://keras.io`

[5] Neural Network and Deep learning
`By Michael Nielsen / Jun 2019`