Article

Democratizing Deep Learning Applications in Earth and Climate Sciences on the Web: EarthAIHub

Muhammed Sit [1,*] and Ibrahim Demir [1,2,3]

1 IIHR–Hydroscience & Engineering, University of Iowa, Iowa City, IA 52242, USA
2 Civil and Environmental Engineering, University of Iowa, Iowa City, IA 52242, USA
3 Electrical and Computer Engineering, University of Iowa, Iowa City, IA 52242, USA
* Correspondence: muhammed-sit@uiowa.edu

Abstract: Most deep learning application studies have limited accessibility and reproducibility for researchers and students in many domains, especially in earth and climate sciences. In order to provide a step towards improving the accessibility of deep learning models in such disciplines, this study presents a community-driven framework and repository, EarthAIHub, that is powered by TensorFlow.js, where deep learning models can be tested and run without extensive technical knowledge. In order to achieve this, we present a configuration data specification to form a middleware, an abstraction layer, between the framework and deep learning models. Once an easy-to-create configuration file is generated for a model by the user, EarthAIHub seamlessly makes the model publicly available for testing and access using a web platform. The platform and community-enabled model repository will benefit students and researchers who are new to the deep learning domain by enabling them to access and test existing models in the community with their datasets, and researchers to share their novel deep learning models with the community. The platform will help researchers test models before adapting them to their research and learn about a model's details and performance.

Keywords: deep learning; tensorflow; neural networks; web application; citizen science; scientific communication

check for updates

Citation: Sit, M.; Demir, I. Democratizing Deep Learning Applications in Earth and Climate Sciences on the Web: EarthAIHub. *Appl. Sci.* **2023**, *13*, 3185. https://doi.org/10.3390/app13053185

Academic Editor: Luisa F. Cabeza

Received: 15 January 2023
Revised: 23 February 2023
Accepted: 28 February 2023
Published: 2 March 2023

1. Introduction

One of the main issues regarding applicable deep learning research is the availability of the models, case studies, and research in an easy and accessible way to other researchers [1]. While reproducing results has been possible, albeit both time and resource-intensive, for experts in the field, it remains a problem for non-technical people, researchers, and students who are new to this domain to see how well a deep learning model performs over their own inputs. This is mostly due to either computational limitations or allocating the time needed for setting up and testing many models they might consider for their research. Many studies [2,3] have made additional efforts to support scientific communication and reproducibility by providing a working example of their study in a web-friendly format that, more often than not, accepts inputs from users and returns an output. Such applications typically carry some popular science and outreach value. Another approach used by deep learning researchers to communicate their findings to the public is to provide their code and trained deep learning models via platforms, such as GitHub, and various deep learning model repositories, often called model zoos (i.e., ONNX Model Zoo [4], Nvidia's NGC Catalog [5]).

While most studies adopt the latter method to enable other people to try their research, this tendency has a logistically sound reason. Even though it is vital for the public to be able to see how the findings of some research are realized in real-world settings, it is also difficult to provide an end product that employs the presented research for third parties to use. Providing proof of concept for research is the backbone for any study to be useful and accessible to others. It is also resource-consuming for researchers to transform that proof of concept into a working application for real-world use cases most of the time. The resource-wise needs for an application typically comprise many aspects, from labor to computational power. From a researcher's perspective, it is more sensible to utilize those resources in further research instead of allocating them for real-world applications just for publicity. One could argue that computational power needs could be surpassed by simply implementing a web application that utilizes the client side's computational power. This would be a reasonable approach in order to get past the problem of devoting costly computational capabilities, but it also adds to the need for labor since, typically, a deep learning model is developed in an experimental setting in server-side programming languages such as Python. Consequently, researchers typically prefer to release the code that could be used to reproduce the results, sometimes alongside a pre-trained model. This, in theory, is an attempt to communicate the study's findings, but by its nature, it only communicates with other researchers who are willing to put the necessary time in to advance the science, rather than supporting non-technical people who might be interested in, and beyond that, could benefit from, the conducted research.

Most of the time, these studies share their model codes on platforms such as GitHub, linking their pre-trained models stored in the cloud. Beyond that, there are many "model zoos" from which many pre-trained state-of-the-art models for common tasks can be acquired. These zoos occasionally include the code and sometimes give out instructions to show how to utilize the given model. While there are examples of model zoos that are framework-blind, in other words, model zoos that do not have specific framework requirements, such as modelzoo.co [6], most numeric computing libraries also have their own zoos in their repositories, such as TensorFlow [7] and Open Neural Network Exchange [4]. While these zoos are community-driven efforts, there are some model zoos that list ready-to-use deep learning models in specific commercial AI devices, such as Texas Instruments' TI Edge AI Cloud [8] and Hailo.ai's Model Zoo [9]. Besides all these examples, Nvidia's NGC Catalog [5] also provides many state-of-the-art models with instructions on how to run them on containers.

Besides the aforementioned model zoos, a rare minority of studies in the literature choose to provide a working application, most of the time in the form of a web application. This has been mainly for generative works, as these models typically expect minimal input from the user. Thus, they are easier to conceptualize on the web, though, most of the time, not by the original authors. Such examples are primarily in the "this thing does not exist" format. They typically show a completely new output generated by the proposed network at each reload of the web application. The list of studies employing this approach includes, but is not limited to, face generation [3,10], vase image generation [11], resume generation [12], and artwork generation [13]. Beyond these, some of the ONNX models from the ONNX zoo that are presented on the ONNX.js [14] demo website, such as emotion detection [15] and object detection [16], also apply.

There are also other practical applications of deep learning models available on the web, such as Quick Draw by Google, which predicts what a user is drawing [17], and pix2pix [2], which converts your drawing to a photo [18]. Online deployments of language models such as GPT-3 [19] can also be mentioned as examples of deep learning models running on browsers. It should be noted that, even though some of the studies mentioned here use server-side resources, they typically utilize client-side resources. One commonality among these models is that they depend on individual efforts to communicate their findings. Aside from the ONNX.js showcase, which is only for demonstrative purposes, they do not share a common platform or methodology to include non-technical people in AI research or other domains.

In order to understand how deep learning models could be run on the web, specifically on the client side, it is of utmost importance to discuss how client-side computing is performed. Conventionally, computing on the web has primarily been for citizen science purposes. From extraterrestrial research [20] to hydroscience [21], there are many studies in the literature that take advantage of the client-side resources of volunteers to advance science, some of which are on browsers. With deep learning being a part of many applicable studies, ways to utilize computing on browsers have been explored for deep neural networks as well. The literature on client-side tensor computing libraries, in other words, deep learning frameworks that work on browsers, is abundant. ONNX.js, Keras.js [22], and TensorFlow.js [23] are popular libraries used in many studies.

Open Neural Network Exchange, or ONNX, is a library that aims to be middleware between many tensor computing libraries, such as TensorFlow, PyTorch [24], and MXNet [25]. ONNX provides programmable interfaces to convert models between popular tensor computing libraries. Even though ONNX has its own runtime as a Python package to run converted models, one can also convert an ONNX model to another framework's model format. Beyond Python, the ONNX ecosystem provides a package to run ONNX models on browsers called ONNX.js. ONNX.js, though with limitations in terms of implemented architectures, provides a medium to run ONNX models in browsers. For performance purposes, the ONNX.js runtime supports web technologies such as WebAssembly and WebGL. One drawback that ONNX and inherently ONNX.js carry is that one cannot train a model with them. Considering that they do not aim to provide training functionality in the first place, this is an arguable drawback. It should be noted that ONNX.js was replaced by ONNX Runtime Web [26], which also works in JavaScript. Since it is still in active development, it does not provide a viable framework to build a generalized deep learning platform with.

One popular deep learning package for JavaScript and browsers is Keras.js. Using the boilerplate of a popular deep learning package, Keras [27], Keras.js provides an easy-to-use framework to train and run deep learning models. While it was one of the go-to solutions to deploy some deep learning models on the web when it was first released, it is not actively maintained as of now in favor of TensorFlow.js. TensorFlow.js uses the TensorFlow library, which is backed by Google. Now that Keras models and TensorFlow models are somewhat intertwined, TensorFlow.js supports both Keras and TensorFlow models, which covers a great majority of deep learning models, but not as much of the applicable research. Beyond this, TensorFlow.js has the large functionality of Keras and TensorFlow in terms of available architectures and matrix operations. This makes TensorFlow.js the most popular deep learning framework for online applications. Similar to ONNX.js, TensorFlow.js also supports WebAssembly and WebGL for faster training and execution.

Furthermore, there are a few other deep learning frameworks that run on JavaScript and subsequently could be used on web browsers, such as WebDNN [28], ConvNet.js [29], and brain.js [30], but they do not focus on community support and practicality like TensorFlow.js. One exception might be ml5.js [31], which runs on TensorFlow.js and aims to provide an abstraction layer to make running models on the web more accessible, thus differentiating itself from others in terms of audience and value. There are vast quantities of deep learning application papers published every day. Most of the time, the research is not easily reproducible. Even when the code or pre-trained models are provided, since all deep learning frameworks need some level of expertise to run a model, people with no AI modeling experience will not be able to run and see the results of a study for themselves.

Deep learning has been a methodology of interest for many earth science studies. Whether it be about social media streams, such as Twitter data, that is related to natural disasters [32], or digital elevation models [33], there are many studies in earth science that utilize deep learning methods. Modeling floods for prediction [34,35] and rainfall to increase temporal interpolation [36] are among the deep learning application studies in the field, along with benchmark dataset efforts [37] to support the literature [38,39]. Even though deep learning application studies are somewhat in abundance in the literature of earth and climate sciences, there are still challenges in the scientific communication of studies utilizing deep learning. To the best of our knowledge, there are no studies that have shared their developed and trained deep learning models for earth and climate sciences problems on the web for third parties to experience. Beyond already published studies, similar to any data-driven methodology, adopting deep learning in their applications is appealing to domain scientists, i.e., scientists who are experts in earth and climate sciences, but might not have the technical expertise to apply deep learning in their field. In order for them to start exploring how deep learning could be utilized in their fields, understanding the literature beyond what the scientific literature presents is of the utmost importance to capitalize on the deep-learning-based modelling potential in the field.

This study aims to empower scientists studying in the domains of earth and climate sciences to reproduce and test previously trained deep learning models easily with their data and settings without having to train or even know how to use a specific tensor computation library used in selected projects. We present EarthAIHub, an easy-to-use web application utilizing client-side resources to run deep learning applications. EarthAIHub comes with a configuration middleware between individual models and TensorFlow.js to facilitate the easy adoption of the system among deep learning practitioners. The configuration structure detailed in this manuscript enables deep learning researchers to communicate their research while allowing domain scientists to experience deep learning models in Earth sciences easily. The platform is designed to accommodate many different input/output types that a neural network model in Earth sciences could potentially operate over, such as comma-separated values (CSV) files, NumPy [40] files (npy), and various image formats, to facilitate ease of use within the Earth sciences field. Furthermore, in order to demonstrate that EarthAIHub is able to run models using those input and output types, we present several model and use cases. EarthAIHub is designed with a community-centered approach and easy-to-use model configuration templates to deploy models to the platform for others to test and experience for the long-term sustainability of the user base.

The rest of this paper is structured as follows; in the next section, we detail the implementation of the web application to explain how deep learning models specified for different tasks could run on the same platform. Then, in Section 3, we present selected deep learning models in environmental science and show how configurations could be built for them and run on EarthAIHub. In Section 4, we share the findings of using EarthAIHub for the presented use cases along with future directions. Finally, in Section 4, we summarize the study and share final remarks.

2. Materials and Methods

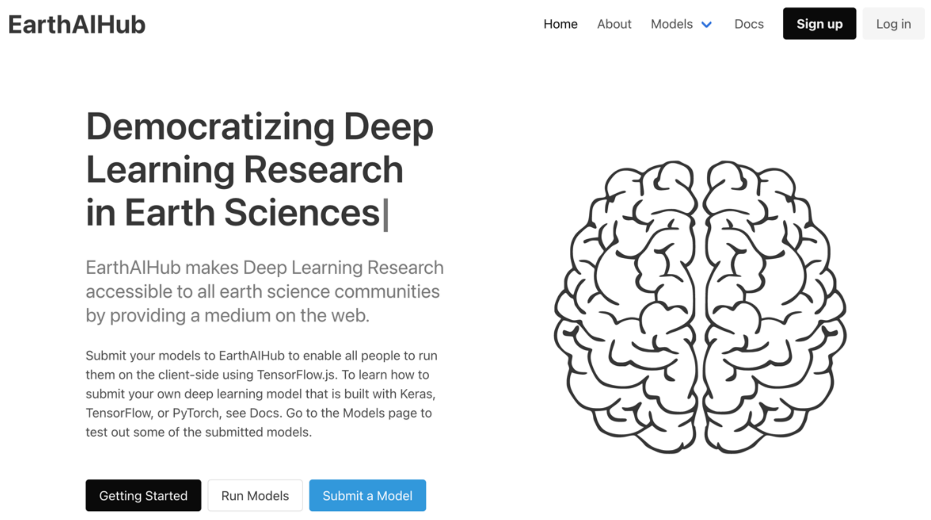
In this section, we describe the architecture and implementation details of the application on the server side and the client side, and present the model specification and process to deploy a model at the platform.

2.1. Architecture and Modules

EarthAIHub (Figure 1) is a web-based deep learning hub (https://earthaihub.org (accessed on 28 September 2022)) for earth and climate researchers for accessing and running pre-trained DL models. While data management and user authentication are carried out on the server side, the most crucial task, running models, is performed on the client side. This allows the computational dependency on the server side to be minimized, reduce the cost, and improve the sustainability of the platform. In order to do so, we implemented a RESTful API and a client-side single-page web application that communicates with the API. Embedded into the client-side web application, a JavaScript Web Worker facilitates the running of models using TensorFlow.js without interrupting the interface and user interactions.

The EarthAIHub runs on a stack based on Flask, which is a web framework in Python. We used TensorFlow.js for running pre-trained models on the client side, and Flask-RESTful, a Flask package that extends Flask's capabilities to implement RESTful services easily on the server side. Vue.js, a client-side JavaScript library, was used to build single-page interfaces, and Bulma, a CSS framework that has modern-looking HTML elements, was used for the design. In order to avoid extra workload, we chose to use SQLite as the database and the Flask-SQLAlchemy package to ensure communication between the database and

Flask. To facilitate the core functionality, the platform was built upon four modules that interact independently with the database and other modules; namely, the User Management Module, Model Management Module, Administration Module, and Compute Module. Aside from these modules, there are also detailed documentation pages that EarthAIHub presents for referencing purposes.


EarthAIHub

Home About Models Docs Sign up Log in

Democratizing Deep Learning Research in Earth Sciences|

EarthAIHub makes Deep Learning Research accessible to all earth science communities by providing a medium on the web.

Submit your models to EarthAIHub to enable all people to run them on the client-side using TensorFlow.js. To learn how to submit your own deep learning model that is built with Keras, TensorFlow, or PyTorch, see Docs. Go to the Models page to test out some of the submitted models.

Getting Started Run Models Submit a Model

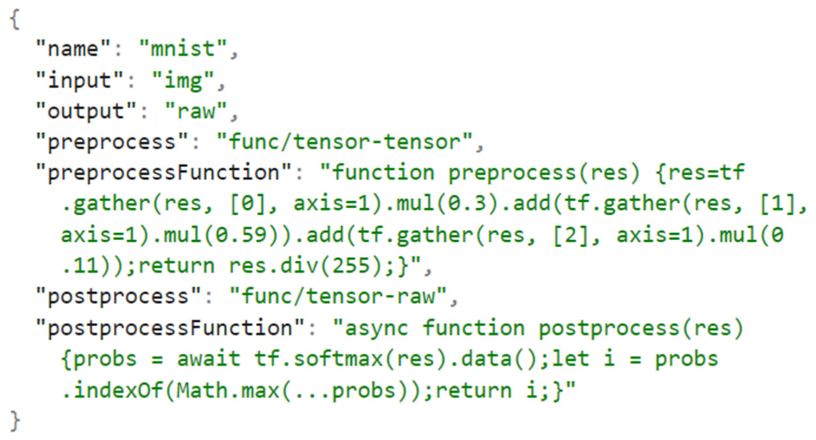
Figure 1. Landing page of EarthAIHub deep learning platform.

2.2. Model Specification

EarthAIHub accepts the TensorFlow.js GraphDef model format, also known as SavedModel, that can be read with the tf.loadGraphModel (...) function, as this function expects the path to the model.json file and the file is typically accompanied by several binary files, all of which are needed by the platform for proper execution. In order to ensure this, the web app expects a zip archive of all binary files and the model.json file in which the model.json is stored at the root of the zip archive at submission time. An example zip archive includes the model configuration file (i.e., model.json) and several binary files (i.e., group1-shard1of3.bin, group1-shard2of3.bin, group1-shard1of2.bin).

EarthAIHub is designed to support two types of users; namely, model developers and model users from the earth and climate science domains. Consequently, the main task for the platform is to provide abstraction layers. In order to do so, we designed a configuration file format and specification that defines everything a user would need to run a TensorFlow.js model. The configuration file includes input format, output format, preprocessing type, preprocessing function, postprocessing type, and postprocessing function. Fields and their definitions for the config file can be seen in Tables A1–A5 in the Appendix. An example model configuration file can be seen in Figure 2.

The example config.json file is built for a classifier model that was trained on 3-channel handwritten number images from the MNIST dataset and returns the predicted number. The model expects an input with the shape of $1 \times 1 \times 28 \times 28$, but ideally, one would expect a user to submit an image to the model. Therefore, in order to facilitate that, we put img as the input type, but this design choice comes with an expense. We now need to preprocess the input that we assume to be an image that is 28×28. As in the config tables in the Appendix, the image will be read from an HTML canvas and will have four channels, so in order to reduce the image to only one channel, we implement cv2.cvtColor with cv2.COLOR_BGR2GRAY from python-OpenCV [41] in TensorFlow.js while avoiding the fourth channel and normalizing the input by dividing it by 255. Since we only dealt with tensors and our end product for the preprocess was a tensor as well, we could choose “func/tensor-tensor” as the preprocess type. Note that other preprocess types might well work without problems here, but our selection was a convenient one for our use case.



```
{
  "name": "mnist",
  "input": "img",
  "output": "raw",
  "preprocess": "func/tensor-tensor",
  "preprocessFunction": "function preprocess(res) {res=tf
    .gather(res, [0], axis=1).mul(0.3).add(tf.gather(res, [1],
    axis=1).mul(0.59)).add(tf.gather(res, [2], axis=1).mul(0
    .11));return res.div(255);}",
  "postprocess": "func/tensor-raw",
  "postprocessFunction": "async function postprocess(res)
    {probs = await tf.softmax(res).data();let i = probs
    .indexOf(Math.max(...probs));return i;}"
}
```

Figure 2. A sample configuration file.

Furthermore, we will need to output a number that is the predicted handwritten digit from the image, so our output type should be raw. Our model outputs a vector with the shape of 1×10 that comprises the probability of each digit before the tf.softmax. Therefore, in postprocess, we run softmax over our output and then convert it to an array. Finally, we find the maximum value's index, which is the value our network has predicted, and return that. Since we chose to use TensorFlow.js's tf.softmax function, it is more convenient for us to expect a tf.tensor into the postprocess function. Finally, since we output a number, we choose "func/tensor-raw" as the postprocessType.

2.3. System Modules

In this subsection, we will provide details for the core system modules (Figure 3) including the User Management, Model Management, Administration, and Compute Modules. The EarthAIHub provides detailed documentation for users to learn how to build submissions for the platform, from creating the config file to converting models in various numerical computing libraries to TensorFlow.js models. In order to refrain from redundancy, we refer the reader to the documents for further details of the platform and usage.

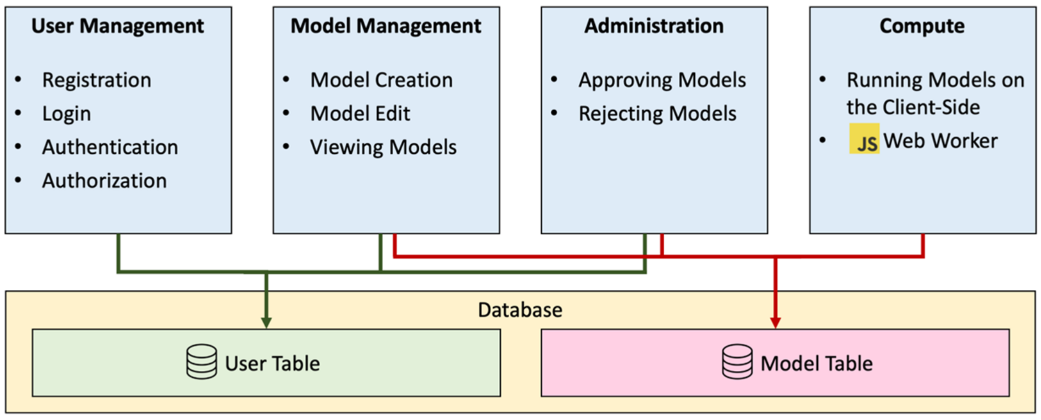


Figure 3. Modules and their connections over different tasks on EarthAIHub.

2.3.1. User Management Module

EarthAIHub is designed as a community-driven platform; thus, authorization and authentication of users is an integral part of the system. To facilitate these, we used JSON Web Tokens (JWTs) [42], which is a proposed internet standard. A JWT is a signature of a user. In authentication, a JWT should be generated by an authority and returned to the user once they provide a set of credentials, such as a username and a password, successfully. Thus, instead of expecting a user to log in every time they make a request or keep session information in the browser, a JWT can be used to authenticate and authorize a user. Based on this widely used methodology, the platform implements an authentication and authorization system that only expects a username and a password from the user. Furthermore, the User table in the database implements an administration functionality where users with an is_admin flag are able to carry out certain tasks such as approving and rejecting submitted models. All the fields that a User table in the database holds can be seen in Table A6 in the appendix.

User Creation: User creation is carried out by filling out a registration form consisting of a username and two password fields. Once a user submits the registration form, the client side validates that both passwords are the same. The server side first checks if a user with the given username already exists. If the user does not exist, it creates a new user entry in the database, saving the password encrypted with SHA256.

User Login: Login functionality on the client side is twofold: POSTing the entered credentials to the server side, and saving returned JWTs on the client side when login is successful. However, the server side handles a more crucial operation once it confirms that the given credentials are correct. The server-side component creates a pair of JWTs for the given user: an access token and a refresh token upon login. An access token is a short-lived JWT, typically for 15 min. It can be used to perform actions throughout the web application that need authorization. On the other hand, a refresh token is a relatively long-lived JWT, typically for 30 days, that can only be used to refresh the access token with a new one. Once these tokens are created, they are sent to the client side. Knowing that the login was successful, the client-side application saves these two tokens as cookies with expiration dates and prompts the user that the login was successful.

From that moment on, whenever the client side needs to make a request that needs authentication, it POSTs the access token as a header along with other data to the server side. However, since the access token is short-lived, a new mechanism is needed to refresh the access token when it is about to expire or has expired already. This is ensured by a function that is run on the client side right before every request that needs to be authorized. To facilitate this, there is a specific route on the server side that checks if the used JWT is an access token and is still valid. If the current access token saved on the client side cannot pass this check, the client-side scripts then proceed to refresh the access token by requesting a token from the refreshing endpoint on the server side with the refresh token as a header in the request. If that endpoint rejects the provided refresh token, then it means either the provided refresh token is corrupt, or it has expired. In either of these cases, the client side receives an error message from the server side, deletes all JWTs from cookies, and redirects to login. How JWTs encrypt user information, validate them, and separate access tokens from refresh tokens will not be discussed here, as we believe these are beyond the scope of this study. We refer the reader to the JWT documentation [41] to better understand these concepts.

User Dashboard: The User Dashboard is an interface on the client side. In communication with the Model Management Module, a logged-in user can see and access their submitted models, see their approval status, and change their password on that interface.

2.3.2. Model Management Module

The Model Management Module in the platform manages model creation and access to those models. While anyone can run any of the models approved by an admin on the app, only registered users can create a model. Full database specifications of a model can be seen in Table A7 in the Appendix.

Model Creation: A user should input the necessary information about the model they are submitting alongside the necessary model files, i.e., the archive file of the TensorFlow.js GraphDef model and the config.json file that we have described earlier. As for the information the user provides regarding the model, EarthAIHub asks for a name, a description of the model, some brief information about the input and output that the model expects and delivers, and finally, an optional extended information text to include any relevant links and publications. Once a user fills out the form and submits it, the client-side Model Management Module makes a POST request to the server side with all the information and files provided and attaches the JWT access token to that request. The server-side aspect is straightforward; both files are saved in the static file directory of the platform with random hexadecimal names, and both unarchived and archived versions of the model are stored. After that, the paths to the saved files and the model information, along with the submitting user's identity, are saved to the database.

Models Page: The Models page is an interface where everyone can see all submitted and approved models, whether they are logged-in users or not. Once a user reaches the Models page, the client-side Model Management Module makes a GET request to the server-side Model Management Module without an authorization header. Those models are returned to the client side without actual model files but with their paths in the app. This way, the burden on both the server-side and client-side steps is minimized because model files should only be downloaded to the client side when needed, which is not until a client decides to try out a model on EarthAIHub.

Individual Model Page: On EarthAIHub, each model has its own page accessible through the Models page. To facilitate running models, the platform's Model Management Module renders an input box according to the submitted config on the model page, and, for the same purpose, a button to run the model and a selection box that allows users to select which backend they want to use to run the model (either CPU or WebGL) are visible. Once a user runs the model after choosing their input file or typing in the input, the Compute Module that will be described below takes over, and once it is finished running, the Model Management Module renders the output on the model page according to the config file. On the Individual Model Page, the owner of individual models finds a link to edit their submission. The editing interface is similar to the model submission interface with minor differences; a user must describe the changes in an external text box to ease administration. Once an edit is submitted, the model is again saved as *is_approved = 0* and needs the attention of an admin.

2.3.3. Administration Module

Being a community-driven platform, EarthAIHub relies on submissions and therefore is prone to misuse. In order to form a middleware between submissions and end-users, the platform facilitates an administration layer. An admin is defined as a User in the database with a flag (*is_admin*) in the table. Every model has an *is_approved* field, and every model submitted to the system is saved to the database with "Pending" status that will be approved by admins.

Admin Dashboard: On the admin dashboard, all models that require approval are listed. Admins can go through these models and make a decision about them. For the most part, similar to the Models page, on the admin dashboard, each listed model has a link to an individual model page. The minor differences are for only admins to see and use to make decisions on submitted models. If an admin decides a model is adequately submitted, they can change the status to "Approved", making its *is_approved* field 1. They can also reject a model to alter its database entry to be *is_approved* = 2. However, they also need to provide an explanation for their decision using the text box placed there for further decision-making purposes. Once a model is rejected, the user who submitted that model sees its rejected status in their dashboard and the rejection reason in the editing interface. Thus, they can edit that model accordingly for resubmission, which ultimately creates a new task for admins. Along with models that need attention, admins can also see models they have made decisions on in their Admin Dashboard. This design choice was made to make it easier for admins to see if models they rejected before are resubmitted.

2.3.4. Compute Module

The Compute Module is where client-side computing capabilities are utilized. When a user enters an Individual Model Page, the Compute Module loads the config file simultaneously with the Model Management Module. After the user enters their input (if applicable), the Compute Module is initiated once the user decides to run the model. The module starts by reading the input if the config states the existence of an input. How the input is read depends on the input type, but at the end, the input is stored as a JavaScript variable. Once the input is read, the client-side computing resources are put to use. In order to do so without interfering with how the interface of EarthAIHub works, we implemented a JavaScript Web Worker. A web worker is created right after the input is read. The input, configuration of the model, and the location of the TensorFlow.js model file are communicated to the worker right after the worker is created.

The worker first imports TensorFlow.js and the TensorFlow.js *GraphDef* model and then runs preprocessing on the input variable depending on the specifications defined in the config file. After that, the worker feeds the preprocessed (if applicable) input to the imported model. Once the model's output is obtained, the worker then runs postprocessing over the output, again according to the specifications in the config. After the postprocessing, the output is then converted to a vanilla JavaScript variable and communicated back outside of the worker's scope. Output is specifically transferred in such a format because (a) we do not import TensorFlow.js outside of the worker's scope, and (b) sending messages to and receiving messages from a web worker does not support external object types. Finally, the Compute Module gives the output back to the Model Management Module to render it according to the config. The workflow of the Compute Module can be seen in Figure 4.

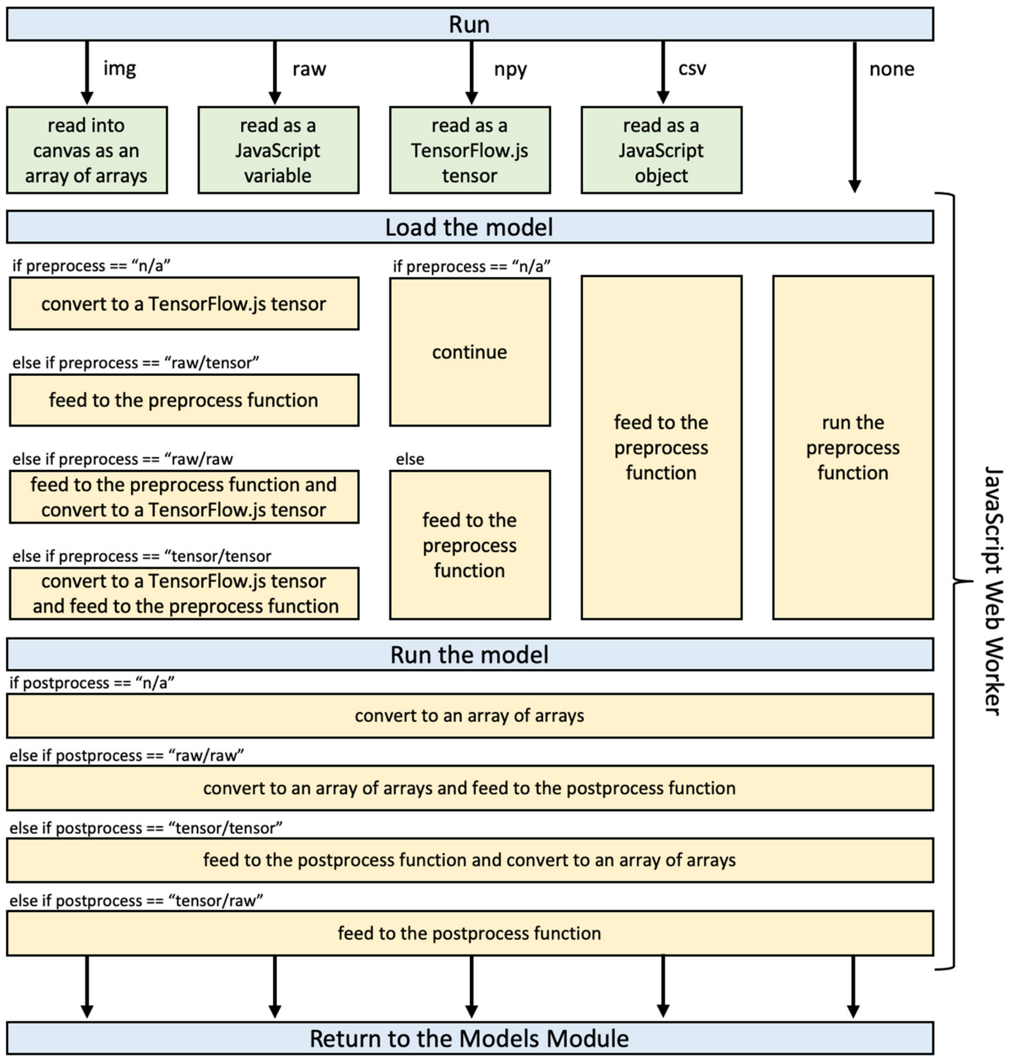

Run
img
raw
npy
csv
none
read into canvas as an array of arrays
read as a JavaScript variable
read as a TensorFlow.js tensor
read as a JavaScript object
Load the model
if preprocess == "n/a"
convert to a TensorFlow.js tensor
else if preprocess == "raw/tensor"
feed to the preprocess function
else if preprocess == "raw/raw
feed to the preprocess function and convert to a TensorFlow.js tensor
else if preprocess == "tensor/tensor
convert to a TensorFlow.js tensor and feed to the preprocess function
if preprocess == "n/a"
continue
else
feed to the preprocess function
feed to the preprocess function
run the preprocess function
Run the model
if postprocess == "n/a"
convert to an array of arrays
else if postprocess == "raw/raw"
convert to an array of arrays and feed to the postprocess function
else if postprocess == "tensor/tensor"
feed to the postprocess function and convert to an array of arrays
else if postprocess == "tensor/raw"
feed to the postprocess function
JavaScript Web Worker
Return to the Models Module

Figure 4. Compute Module schema.

2.4. User Interface and Scenarios

In this subsection, we will outline some common user scenarios. Besides being a registered user of EarthAIHub, submitting a model to the platform requires a clear understanding of the configuration format. Once a researcher has their model in TensorFlow.js *GrapDef* format, also known as SavedModel, they need to make sure their model works flawlessly even without the platform. Then, the next step would be to prepare the config file as described in Section 2.2 and the Appendix. When both the model file and the config are ready, one can quickly fill out the submission form and create a task for admins (Figure 5).

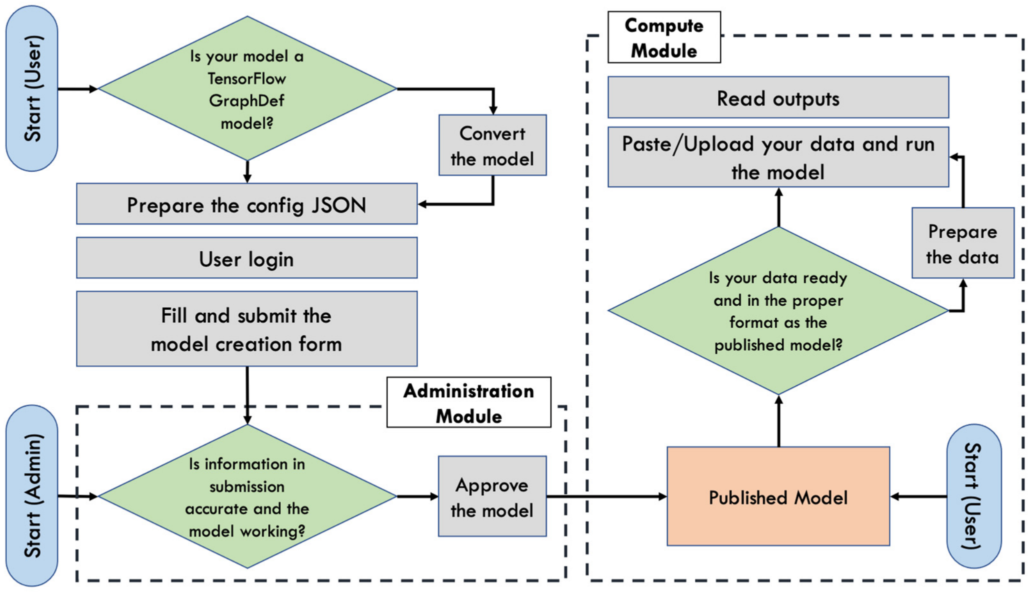

Start (User)
Is your model a TensorFlow GraphDef model?
Convert the model
Prepare the config JSON
User login
Fill and submit the model creation form
Administration Module
Start (Admin)
Is information in submission accurate and the model working?
Approve the model
Compute Module
Read outputs
Paste/Upload your data and run the model
Prepare the data
Is your data ready and in the proper format as the published model?
Published Model
Start (User)

Figure 5. Model creation/submission and computing workflow.

Previously submitted and approved models can be seen by any user of the web application through the navigation bar. Once a user sees the list of already submitted and approved models, they can choose the one they prefer and go to the individual model page to run the model (Figure 5). If the model needs any input, the user can upload their input, select the backend, and run the model. After model computation, the result will appear in the result section of the model page.

3. Results

Many applications from many scientific disciplines can be run on EarthAIHub, as deep learning applications are not limited to any specific field. However, per the goal of this study, we want to draw attention to deep learning applications in the earth and climate sciences, as deep neural networks have been used in various tasks within these disciplines, and we aim to provide a go-to platform on which domain scientists can obtain understanding. This section demonstrates how the platform can be utilized for various deep learning applications. For each model, we first introduce the model and then refer to the necessary EarthAIHub config in the Appendix for that model. Most of the models here will refer to research carried out in hydrology, but also to prove the concept, we put the platform to use for various state-of-the-art deep learning models in common application areas.

3.1. Generic Use Cases

3.1.1. Emotion Detection with FER+

Facial expression recognition has been a task of interest in computer vision even before the comprehensive utilization of deep neural networks for this purpose. FER+ [15] is an annotation dataset where various images that depict human faces with facial expressions are labeled. This model is a deep Convolutional Neural Network (CNN) [43] that was trained on FER+ using the CNTK [44] numeric computing library and was provided for public use in the ONNX GitHub repository. We converted this ONNX model to a TensorFlow.js *GraphDef* model and built a config file that wraps the model in order for it to be run on EarthAIHub. The model on EarthAIHub expects a 64 × 64 image and returns probabilities for eight different emotions. The config file for this model can be seen in Appendix B.1.

3.1.2. Image Classification with EfficientNet-Lite4

EfficientNet [45] is a convolutional neural network architecture that changes how scaling is performed for neural networks. The EfficientNet-Lite4 is a CNN that is trained on ImageNet [46] and achieves state-of-the-art accuracy. One upside of EfficientNet-Lite4 is that it is designed to work on devices with limited computational capabilities, thus making it a great candidate to run on the web. The model on EarthAIHub expects an image of 224 × 224 and outputs five ImageNet categories with top probabilities. The trained model that was trained on the COCO 2017 [47] dataset was obtained from the ONNX GitHub repository. The model was converted to the TensorFlow.js *GraphDef* model. The config file for this model can be seen in Appendix B.2.

3.1.3. Handwritten Digit Recognition from MNIST

MNIST [48] is a handwritten digit image dataset widely used for educational and benchmarking purposes in computer vision. This model is a pretrained CNN over the MNIST dataset that reaches a top-1 error rate of 1.1%. The model was obtained from the ONNX model zoo at the ONNX GitHub repository and then converted to a TensorFlow.js *GraphDef* model. The model takes a matrix of 28 × 28, but the inputting image interface on EarthAIHub reads images with four channels; consequently, the config was designed accordingly. The preprocess function transforms the submitted image into one channel using the *cv2.COLOR_BGR2GRAY* implementation on opencv-python [41], avoiding the fourth channel. The config file for this model can be seen in Appendix B.3.

3.1.4. Image Classification on CIFAR-10

CIFAR-10 [49], like MNIST, is a popular dataset among deep learning practitioners and researchers, making it an excellent go-to dataset for training and demonstrating the promise of deep-learning-related works. We followed the "Deep Learning with PyTorch: A 60 Minute Blitz > Training a Classifier" tutorial for this model and trained a CNN over CIFAR on PyTorch. Then, we converted the PyTorch model to a TensorFlow.js one. The config file for this model can be seen in Appendix B.4.

3.2. Earth and Climate Sciences Use Cases

3.2.1. LiDAR Super-Resolution

Light Detection Additionally, Ranging (LiDAR) is a remote sensing method to map the elevation of terrains or surroundings. In earth sciences, it is widely used in various modeling and geospatial analysis studies, including water delineation [50], flood forecasting, and mapping [51], and higher-resolution LiDAR data is crucial. This model is a pretrained model defined in [52] which takes a 25 × 25 raw JavaScript array of arrays and increases the resolution to 400 × 400. The config file for this model can be seen in Appendix B.5. A set of examples of low-resolution and high-resolution DEMs, and the output of the DSRGAN, can be seen in Figure 6.

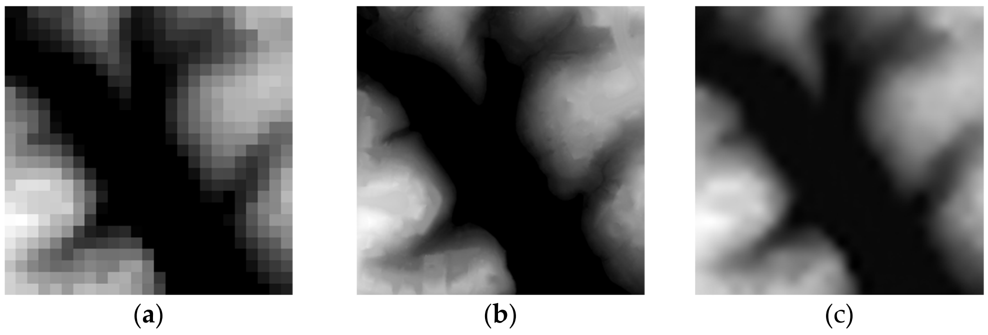


Figure 6. (**a**) Low-resolution DEM, (**b**) high-resolution DEM, and (**c**) generated high-resolution DEM by DSRGAN.

3.2.2. Generating New Satellite River Imagery

Data generation with deep neural networks is one of the approaches that has not been widely explored in water resources and climate research. However, one study explores generative adversarial networks (GANs) [53] in satellite river imagery generation. This model we show as a use case here is a pre-trained version of the Progressively Growing GAN [54] model with the dataset presented in Gautam et al. [55] that outputs a randomly generated bird's eye river image. Since this is a generation task from a random noise vector, the config file was created accordingly. The config file for this model can be seen in Appendix B.6. A generated 1024 × 1024 river image from a random vector on EarthAIHub can be seen in Figure 7.

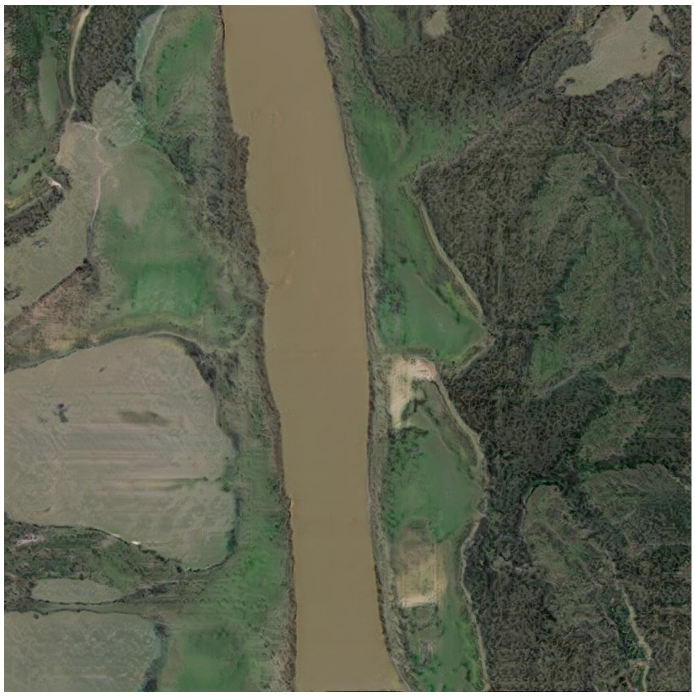


Figure 7. An example output of river image generation GAN [55].

3.2.3. Flood Forecasting with NRM

Flood forecasting is a vital task in hydrology. Beyond physical modeling, various approaches have been used to tackle this task with both conventional machine learning models and deep learning [56]. This model was introduced by [57] and utilizes long short-term memory (LSTM) [58] networks for 24 h of rainfall-runoff forecasting. The model was trained on Keras for a USGS streamflow station in Iowa, United States and then converted to a TensorFlow.js model. The config was created to read a CSV file for all the data that the model expects. The config file for this model can be seen in Appendix B.7. An example forecast for 24 h of input and 24 h of output runoff is visualized on a line chart in Figure 8.

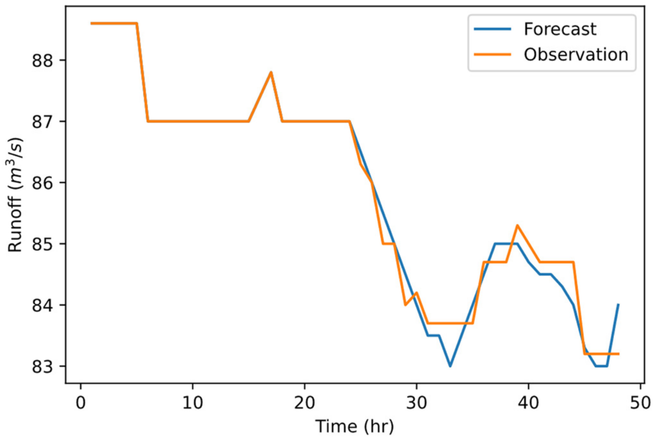


Figure 8. Input/output lines for a streamflow data sequence of 48 h.

4. Discussion

As we briefly discussed in the Introduction section, a common approach to making deep learning accessible is to share trained models through deep learning model repositories, or preparing a typically web-based dedicated platform that allows others to experiment with the model. For the first option, Earth scientists would need technical expertise to train or run specific tensor computing libraries in order to test the shared deep learning model, while the second option brings a burden for deep learning practitioners as they would need to allocate substantially more time and resources to make their research publicly available and accessible to ensure scientific communication. On the contrary, in the previous section, using EarthAIHub, we have shown that with the use of their own data, domain experts from the earth and climate sciences will be able to quickly recreate and test previously trained deep learning models without having to learn how to use a specialized tensor computation library that is utilized in some Earth sciences studies as interface figures suggest that EarthAIHub is a user-friendly web application; thanks to the design choices, it takes advantage of client-side resources to run deep learning models. Furthermore, the configuration middleware between individual models and TensorFlow.js that is included with EarthAIHub makes it hassle-free for deep learning practitioners to utilize the system. The previous section proves that using the configuration for ease of usage in the earth sciences, the platform is made to support a wide range of input/output types that a neural network model may possibly operate over, including comma-separated values (CSV) files, NumPy files (npy), and numerous picture formats.

As of now, EarthAIHub is designed to run only TensorFlow.js models. It is not always possible to convert a model that has been trained in one programming language and framework pair to TensorFlow and later TensorFlow.js. For instance, in order to convert a PyTorch model to a TensorFlow.js model, one needs to convert that model to ONNX, TensorFlow, and finally TensorFlow.js consecutively. Moreover, some implementations of some architectures are not always available in all of these libraries. For instance, some recurrent neural network structures in PyTorch are implemented differently than they are in TensorFlow. Consequently, one could easily encounter problems when converting that model to TensorFlow.js. The coverage of models and architectures could be extended by incorporating other numeric computing and deep learning libraries in JavaScript, such as ONNX Runtime Web. One downside of not implementing ONNX Runtime Web into EarthAIHub during this study was, as mentioned earlier, that ONNX Runtime Web has still not been fully developed for practical web applications.

5. Conclusions

This study presented EarthAIHub, a community-driven web platform where practitioners and researchers make their deep learning applications accessible and end users experience how deep learning research in earth science and climate domains works. In order to achieve a generalized platform that can run many different deep learning models, we created an abstraction layer; a middleware, that is, a configuration file between specific deep learning models and end users based on a JSON file. The config file works to make the EarthAIHub understand individual models. In order to show both the capabilities of the web application and how various models with different input–output types could be formatted to run on the platform, we provided seven use cases. Even though we mostly showed computer vision applications as they are easier to visualize, we also demonstrated that the platform is able to provide a medium for models with various goals, such as time series forecasting and super-resolution applications.

Author Contributions: Conceptualization, M.S. and I.D.; methodology, M.S and I.D.; software, M.S.; validation, I.D.; formal analysis, M.S.; investigation, I.D.; resources, I.D.; data curation, M.S.; writing—original draft preparation, M.S.; writing—review and editing, I.D.; visualization, M.S.; supervision, I.D.; project administration, I.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: There were no new data produced for this study and only data from cited research were used. However, the EarthAIHub web application is available for public use at https://earthaihub.org (accessed on 28 September 2022). Software files are not available as the source code of the platform is not open for safety reasons.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Table A1. Fields and their definitions for the config file.

Field	Description
name	The name of the model, for reference purposes only, is not used in the execution.
input	Defines the type of the input. EarthAIHub enables you to easily run various types of inputs. The model page is rendered to ask the end user to put in or upload the selected input type. For each input type, there is a range of preprocessing function types available. Please refer to Table A2 for the available input types.
preprocess	This field should be filled with one of the input-specific *preprocess* options. Please refer to Table A3 for available *preprocess* options for each input type.
preprocessFunction	A function is defined as a string. The function name must be preprocessed, take an input (even if the input type is none), and return an output.
output	Defines the type of the output. The output could be *img* or *raw* to return the output the model produces in the correct format to the end user. For each output type, there is a range of post-processing function types available. Please refer to Table A4 for the available output types.
postprocess	This field should be filled with one of the output-specific *postprocess* options. Please refer to Table A5 for available postprocess options for each output type.
postprocessFunction	A function is defined as a string. The function name must be postprocess, and it must take an input and return an output.

Table A2. Input types and their definitions.

Input Type	Description
img	Input is a single image read in four channels by an HTML canvas, and the end user uploads an image.
raw	A *textbox* is rendered so the end user can input whatever they want to be read into a JavaScript variable using *eval()*.
npy	An *npy* file that carries exported NumPy (Harris et al., 2020) [40] arrays is needed. The input is always converted to a tf.tensor upon reading.
csv	A *csv* file upload is needed; the uploaded *csv* file will be read as a JavaScript *object*.
none	Nothing is expected as the input, and a preprocessing function must be defined. Whatever the preprocessing function returns is fed to the *GraphDef* model. The preprocess field in the config does not need to be defined.

Table A3. Input types and their definitions.

Input Type	Preprocess Option	Description
img	n/a	This means preprocessing function is not applicable. Thus, no preprocessing function will be provided; the input will be fed into the model as-is. Typically, not the case.
	func/raw-tensor	The input to the preprocess function is the *raw* input format, which is an array of arrays for *img*. The output of the preprocess function is assumed to be a *tf.tensor*, so if a user chooses to use this option in the config, they need to be sure that their preprocess function returns a tensor.
	func/tensor-tensor	The input to the preprocess function is a *tf.tensor*. The output of the preprocess function is also assumed to be a *tf.tensor*, so if a user chooses to use this option in config, they need to be sure that their preprocess function returns a *tf.tensor*.
	func/raw-raw	The input to the preprocessing function is the *raw* format of the input, which is an array of arrays for *img*. The output is also in raw format, which could be anything but a *tf.tensor*. The system will handle conversion to the tensor before feeding the output of the preprocess function by running *tf.tensor()* on the output.
raw	n/a	No preprocessing function will be provided; the input will be fed into the model as-is. Typically, not the case.
	func/raw-tensor	The input to the preprocess function is the raw format of the input that is the output of the *eval()* when run on the content of the *textarea* for the *raw*. The output of the preprocess function is assumed to be a *tf.tensor*, so if a user chooses to use this option in the config, they need to be sure that their preprocess function returns a tensor.
	func/tensor-tensor	The input to the preprocess function is a *tf.tensor*. The *textarea* is fed to *tf.tensor(eval(...))*, and the result is fed to the model. The output of the preprocess function is also assumed to be a *tf.tensor*, so if a user chooses to use this option in the config, they need to be sure that their preprocess function returns a *tf.tensor*.
	func/raw-raw	The input to the preprocess function is the raw format of the input that is the output of the *eval()* when run on the content of the *textarea* for the raw. The output is also in *raw* format, which could be anything but a *tf.tensor*. The system will handle conversion to the tensor before feeding the output of the preprocess function by running *tf.tensor()* on the output.
npy	n/a	No preprocessing function will be provided; the input will be fed into the model as-is.
	func/raw-tensor	The input to the preprocess function is a *tf.tensor*. The output of the preprocess function is assumed to be a *tf.tensor*, so if a user chooses to use this option in the config, they need to be sure that their preprocess function returns a tensor.
csv	func/raw-tensor	The input to the preprocess function is a JavaScript *object* defined by the *csv* file read. The output of the preprocess function is assumed to be a *tf.tensor*, so if a user chooses to use this option in the config, they need to be sure that their preprocess function returns a *tf.tensor*.
none	n/a	The preprocess type does not have to be defined, but a *preprocessFunction* must be given.

Table A4. Output types and their definitions.

Field	Description
img	The output of the model or postprocess function (if applicable) has to be a *tf.tensor* shaped $1 \times 3 \times w \times h$ so that it can be rendered.
raw	The model output is rendered as-is on the page.

Table A5. Output types and available postprocessing types for each of the output types.

Output Type	Postprocess Option	Description
img	n/a	This means preprocessing function is not applicable. Thus, no postprocess function defined. The model's output is converted to an *array* by *arraySync()* and put on an HTML canvas to render as an image.
	func/raw-raw	The model's output is converted to an array by *arraySync()*, and then the postprocess function is run. The output of the postprocess function should also be something other than a *tf.tensor*, so it would be rendered on an HTML canvas after the postprocess.
	func/tensor-tensor	The output of the model is fed to the postprocess function. The output of the postprocess function should also be a tensor which is then converted to the raw format by *arraySync()* and rendered on an HTML canvas.
	func/tensor-raw	The output of the model is fed to the *postprocess* function. The output of the *postprocess* function should be something other than a *tf.tensor*, so it would be rendered on an HTML *canvas* after the *postprocess*.
raw	n/a	There is no *postprocess* function defined. The model's output is converted to an array by *arraySync()* and rendered as-is.
	func/raw-raw	The model's output is converted to an array by *arraySync()*, and then the *postprocess* function is run. The output of the *postprocess* function could be anything but a tensor so that it would get rendered after the *postprocess*.
	func/tensor-tensor	The output of the model is fed to the *postprocess* function. The output of the *postprocess* function should also be a *tf.tensor*. The output of the *postprocess* function is then converted to the raw format by *arraySync()* and rendered.
	func/tensor-raw	The output of the model is fed to the *postprocess* function. The output of the *postprocess* function should also be something other than a *tf.tensor*, so it would be rendered after the *postprocess*.

Table A6. Fields and their types for the User table in the database.

Field	Type
id	Integer
username	String
password	String
is_admin	Boolean
timestamp	DateTime

Table A7. Fields and their types for the Model table in the database.

Field	Type
id	Integer
owner	String
name	String
description	String
input_output	String
more_info	String
model_file	String
config_file	String
is_approved	Integer
approving_mod	String
notes	String
timestamp	DateTime
updated	DateTime

Appendix B

Appendix B.1. Emotion Detection with FER+

```
{
“name”: “FER+”,
“input”: “img”,
“output”: “raw”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res) {res=tf.gather(res, [0], axis=1).mul(0.3).add
(tf.gather(res, [1], axis=1).mul(0.59)).add(tf.gather(res, [2], axis=1).mul(0.11));return res;}”,
“postprocess”: “func/tensor-raw”,
“postprocessFunction”: “async function postprocess(res) {probs = await tf.softmax(res).data();
return {‘neutral’:probs [0], ‘happiness’:probs [1], ‘surprise’:probs [2], ‘sadness’:probs [3], ‘anger’
:probs [4], ‘disgust’:probs [5], ‘fear’:probs [6], ‘contempt’:probs [7]};}”
}
```

Appendix B.2. Image Classification with EfficientNet-Lite4

```
{
“name”: “EfficientNet-Lite4”,
“input”: “img”,
“output”: “raw”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res) {res=tf.gather(res, [0, 1, 2], axis=1);res=tf.einsum
(‘bcij->bijc’, res);return res.sub(127).div(128);}”,
“postprocess”: “func/tensor-raw”,
“postprocessFunction”: “async function postprocess(res) {labels={...};probs = tf.softmax(res);
topkIndices = await probs.topk(5).indices.data();var categories=[];for (var i = 0; i < topkIndices.length;
i++) {categories.push(labels[topkIndices[i]]);};return categories;}”
}
```

Appendix B.3. Handwritten Digit Recognition from MNIST

```
{
“name”: “MNIST”,
“input”: “img”,
“output”: “raw”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res) {res=tf.gather(res, [0], axis=1).mul(0.3).add
(tf.gather(res, [1], axis=1).mul(0.59)).add(tf.gather(res, [2], axis=1).mul(0.11));return res.div(255);}”,
“postprocess”: “func/tensor-raw”,
“postprocessFunction”: “async function postprocess(res) {probs = await tf.softmax(res).data();
let i = probs.indexOf(Math.max(...probs));return i;}”
}
```

Appendix B.4. Image Classification on CIFAR-10

```
{
“name”: “CIFAR”,
“input”: “img”,
“output”: “raw”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res){res=tf.gather(res, [0, 1, 2], axis=1);return
res.div(255).sub(0.5).div(2);}”,
“postprocess”: “func/raw-raw”,
“postprocessFunction”: “function postprocess(res){classes = [‘plane’, ‘car’, ‘bird’, ‘cat’, ‘deer’,
‘dog’, ‘frog’, ‘horse’, ‘ship’, ‘truck’];let i = res [0].indexOf(Math.max(...res [0]));return classes[i];}”
}
```

Appendix B.5. LiDAR Super-Resolution

```
{
“name”: “LiDAR”,
“input”: “raw”,
“output”: “raw”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res) {return tf.expandDims(tf.expandDims(res, axis=0), axis=0).div(1000);}”,
“postprocess”: “func/tensor-tensor”,
“postprocessFunction”: “function postprocess(res) {return res.squeeze().mul(1000);}”
}
```

Appendix B.6. Generating New Satellite River Imagery

```
{
“name”: “RiverGAN”,
“input”: “none”,
“output”: “img”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res) {return tf.randomNormal([1, 512]);}”,
“postprocess”: “func/tensor-tensor”,
“postprocessFunction”: “function postprocess(res) {return res.add(1).div(2).mul(255);}”
}
```

Appendix B.7. Flood Forecasting with NRM

```
{
“name”: “NRM”,
“input”: “csv”,
“output”: “raw”,
“preprocess”: “func/tensor-tensor”,
“preprocessFunction”: “function preprocess(res){enc_q = tf.tensor(res [0]).div(10000);enc_pcp = tf.tensor(res [1]).div(200);enc_et = tf.tensor(res [2]);dec_pcp = tf.tensor(res [3]).div(200);dec_et = tf.tensor(res [4]);enc = tf.concat([enc_q, enc_pcp, enc_et], 0);dec = tf.concat([dec_pcp, dec_et], 0);return [dec, enc];}”,
“postprocess”: “func/tensor-tensor”,
“postprocessFunction”: “function postprocess(res){return res.mul(10000)}”
}
```

References

1. Sit, M.; Demiray, B.Z.; Xiang, Z.; Ewing, G.J.; Sermet, Y.; Demir, I. A comprehensive review of deep learning applications in hydrology and water resources. *Water Sci. Technol.* **2020**, *82*, 2635–2670. [CrossRef] [PubMed]
2. Isola, P.; Zhu, J.Y.; Zhou, T.; Efros, A.A. Image-to-image translation with conditional adversarial networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017. [CrossRef]
3. Karras, T.; Laine, S.; Aila, T. A style-based generator architecture for generative adversarial networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019. [CrossRef]
4. Open Neural Network Exchange. Available online: https://github.com/onnx/onnx (accessed on 28 September 2022).
5. Nvidia GPU Cloud Catalog. Available online: https://catalog.ngc.nvidia.com (accessed on 28 September 2022).
6. Model Zoo. Available online: https://modelzoo.co (accessed on 28 September 2022).
7. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv* **2015**, arXiv:1603.04467.
8. Texas Instruments Edge AI Cloud. Available online: https://dev.ti.com/edgeai/ (accessed on 28 September 2022).
9. Hailo AI Model Zoo. Available online: https://hailo.ai/developer-zone/model-zoo/ (accessed on 28 September 2022).
10. This Person Does Not Exist. Available online: http://thispersondoesnotexist.com (accessed on 28 September 2022).
11. This Vessel Does Not Exist. Available online: https://thisvesseldoesnotexist.com/ (accessed on 28 September 2022).

12. This Resume Does Not Exist. Available online: https://thisresumedoesnotexist.com/ (accessed on 28 September 2022).
13. This Artwork Does Not Exist. Available online: https://thisartworkdoesnotexist.com/ (accessed on 28 September 2022).
14. ONNX.js. Available online: https://github.com/microsoft/onnxjs (accessed on 28 September 2022).
15. Barsoum, E.; Zhang, C.; Ferrer, C.C.; Zhang, Z. Training deep networks for facial expression recognition with crowd-sourced label distribution. In Proceedings of the 18th ACM International Conference on Multimodal Interaction, Tokyo, Japan, 12 November 2016. [CrossRef]
16. Redmon, J.; Farhadi, A. YOLO9000: Better, faster, stronger. In Proceedings of the IEEE conference on computer vision and pattern recognition, Honolulu, HI, USA, 21–26 July 2017. [CrossRef]
17. Quick, Draw. Available online: https://quickdraw.withgoogle.com/ (accessed on 28 September 2022).
18. Image-to-Image Demo. Available online: https://affinelayer.com/pixsrv/ (accessed on 28 September 2022).
19. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *arXiv* **2020**, arXiv:2005.14165.
20. Anderson, D.P.; Cobb, J.; Korpela, E.; Lebofsky, M.; Werthimer, D. SETI@ home: An experiment in public-resource computing. *Commun. ACM* **2002**, *45*, 56–61. [CrossRef]
21. Agliamzanov, R.; Sit, M.; Demir, I. Hydrology@ Home: A distributed volunteer computing framework for hydrological research and applications. *J. Hydroinform.* **2020**, *22*, 235–248. [CrossRef]
22. Keras.js. Available online: https://github.com/transcranial/keras-js (accessed on 28 September 2022).
23. Smilkov, D.; Thorat, N.; Assogba, Y.; Yuan, A.; Kreeger, N.; Yu, P.; Zhang, K.; Cai, S.; Nielsen, E.; Soergel, D.; et al. Tensorflow. js: Machine learning for the web and beyond. *arXiv* **2019**, arXiv:1901.05350.
24. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 8026–8037.
25. Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv* **2015**, arXiv:1512.01274.
26. ONNX Runtime Web. Available online: https://github.com/microsoft/onnxruntime/tree/master/js/web (accessed on 28 September 2022).
27. Keras. Available online: https://github.com/keras-team/keras (accessed on 28 September 2022).
28. WebDNN. Available online: https://github.com/mil-tokyo/webdnn (accessed on 28 September 2022).
29. ConvNet.js. Available online: https://github.com/karpathy/convnetjs (accessed on 28 September 2022).
30. brain.js. Available online: https://github.com/BrainJS/brain.js (accessed on 28 September 2022).
31. ml5. Available online: https://github.com/ml5js/ml5-library (accessed on 28 September 2022).
32. Sit, M.A.; Koylu, C.; Demir, I. Identifying disaster-related tweets and their semantic, spatial and temporal context using deep learning, natural language processing and spatial analysis: A case study of Hurricane Irma. *Int. J. Digit. Earth* **2019**, *12*, 1205–1229. [CrossRef]
33. Demiray, B.Z.; Sit, M.; Demir, I. DEM Super-Resolution with EfficientNetV2. *arXiv* **2021**, arXiv:2109.09661.
34. Sit, M.; Demiray, B.; Demir, I. Short-term hourly streamflow prediction with graph convolutional gru networks. *arXiv* **2021**, arXiv:2107.07039.
35. Xiang, Z.; Demir, I.; Mantilla, R.; Krajewski, W.F. A regional semi-distributed streamflow model using deep learning. *EarthArxiv* **2021**. [CrossRef]
36. Sit, M.; Seo, B.C.; Demir, I. CNN-based Temporal Super Resolution of Radar Rainfall Products. *arXiv* **2021**, arXiv:2109.09289.
37. Ebert-Uphoff, I.; Thompson, D.R.; Demir, I.; Gel, Y.R.; Karpatne, A.; Guereque, M.; Kumar, V.; Cabral-Cano, E.; Smyth, P. A vision for the development of benchmarks to bridge geoscience and data science. In Proceedings of the 7th International Workshop on Climate Informatics, Boulder, CO, USA, 19–22 September 2017.
38. Sit, M.; Seo, B.C.; Demir, I. Iowarain: A statewide rain event dataset based on weather radars and quantitative precipitation estimation. *arXiv* **2021**, arXiv:2107.03432.
39. Demir, I.; Xiang, Z.; Demiray, B.Z.; Sit, M. WaterBench: A Large-scale Benchmark Dataset for Data-Driven Streamflow Forecasting. *Earth Syst. Sci. Data* **2023**, *14*, 5605–5616. [CrossRef]
40. Harris, C.R.; Millman, K.J.; van der Walt, S.J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N.J.; et al. Array programming with NumPy. *Nature* **2020**, *585*, 357–362. [CrossRef]
41. Bradski, G.; Kaehler, A. OpenCV. Dr. *Dobb's J. Softw. Tools* **2000**, *25*, 120–123.
42. JSON Web Tokens. Available online: https://jwt.io (accessed on 28 September 2022).
43. LeCun, Y.; Bengio, Y. Convolutional networks for images, speech, and time series. In *The Handbook of Brain Theory and Neural Networks*, 1st ed.; Arbib, M.A., Ed.; The MIT Press Location: Cambridge, MA, USA, 1995; Volume 3361, p. 1995.
44. Seide, F.; Agarwal, A. CNTK: Microsoft's open-source deep-learning toolkit. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016. [CrossRef]
45. Tan, M.; Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In Proceedings of the International Conference on Machine Learning 2019, Long Beach, CA, USA, 9–15 June 2019.
46. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009. [CrossRef]

47. Lin, T.Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft coco: Common objects in context. In Proceedings of the 13th European Conference on Computer Vision, Zurich, Switzerland, 6 September 2014. [CrossRef]
48. Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Process. Mag.* **2012**, *29*, 141–142. [CrossRef]
49. Krizhevsky, A.; Hinton, G. Convolutional deep belief networks on cifar-10. (Unpublished work).
50. Sit, M.; Sermet, Y.; Demir, I. Optimized watershed delineation library for server-side and client-side web applications. *Open Geospat. Data Softw. Stand.* **2019**, *4*, 8. [CrossRef]
51. Hu, A.; Demir, I. Real-time flood mapping on client-side web systems using hand model. *Hydrology* **2021**, *8*, 65. [CrossRef]
52. Demiray, B.Z.; Sit, M.; Demir, I. D-SRGAN: DEM super-resolution with generative adversarial networks. *SN Comput. Sci.* **2021**, *2*, 48. [CrossRef]
53. Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. Generative adversarial networks. *Commun. ACM* **2020**, *63*, 139–144. [CrossRef]
54. Karras, T.; Aila, T.; Laine, S.; Lehtinen, J. Progressive growing of gans for improved quality, stability, and variation. *arXiv* **2017**, arXiv:1710.10196.
55. Gautam, A.; Sit, M.; Demir, I. Realistic river image synthesis using deep generative adversarial networks. *Front. Water* **2022**, *4*, 10. [CrossRef]
56. Sit, M.; Demir, I. Decentralized flood forecasting using deep neural networks. *arXiv* **2019**, arXiv:1902.02308.
57. Xiang, Z.; Yan, J.; Demir, I. A rainfall-runoff model with LSTM-based sequence-to-sequence learning. *Water Resour. Res.* **2020**, *56*, e2019WR025326. [CrossRef]
58. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef]