Project Number: ABZ4

Department of Computer Science
Department of Robotics Engineering

INNDiE: An Integrated Neural Network Development Environment

by

Ryan G. Benasutti and Austin C. Shalit

A Major Qualifying Project Report

Submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science
May 18, 2020

APPROVED:

_____

Brad A. Miller, Advisor

_____

Carlo Pinciroli, Co-Advisor

Abstract

Modern machine learning methods are capable of tackling problems that are traditionally difficult for computers to solve. These methods present a steep learning curve with a wall of information and concepts for novices and often requires expensive computing resources to implement. INNDiE solves these problems by helping the user configure, train, and test neural networks with limited programming and machine learning background knowledge via a graphical user interface. INNDiE also supports training neural networks in the cloud which reduces amount of personal computing resources required for training these networks.

Acknowledgements

We would like to give special thanks to our advisors Brad Miller and Carlo Pinciroli for providing constant, much needed guidance every step of the way through this project. Our project would not have been nearly as successful without them. We would also like to thank the Amazon Educate team for spending many valuable hours working with us over the course of the project.

Contents

Chapter 1

Introduction

1.1 Challenges in Machine Learning

Modern machine learning methods are capable of tackling problems that are traditionally difficult for computers to solve. Challenges such as recognizing people, cars, and other objects in images, detecting the semantic meaning of sentences, captioning images, and generating paintings or music are unreasonably difficult with traditional methods. In the past ten years, advances in machine learning have enabled computers to solve these problems [21, 38, 34, 32, 39]. However, developing and using these methods is not easy. There are a number of barriers to entry that novices face. First, the world of machine learning presents a wall of information, unfamiliar terms and concepts, and a very steep learning curve. Novices must learn both the concepts and algorithms behind machine learning and frameworks that implement machine learning solutions, such as TensorFlow or PyTorch. This learning endeavor is made harder still because of the second obstacle novices face: training neural networks requires considerable computing resources which are typically too expensive for individuals to purchase [35, 25]. Small networks can be trained with small amounts of data on commodity hardware, but developing networks that will perform well in real-world scenarios requires expensive hardware and large datasets. Additionally, running inference using these networks becomes less performant as the size and complexity of the network increases. Robotic applications typically need low-latency inference, so dedicated hardware is often required [29, 37]. However, this hardware can be difficult to configure and presents yet another learning curve for novice users.

1.2 Problem Statement

Existing methods for machine learning suffer from a very steep learning curve. The few methods that have a flatter learning curve require the user to choose from a small set of supported neural networks. These methods also require the user to preprocess their dataset so that it is compatible with the neural network they will use. Many of these methods also require a large amount of personal computing resources needed to train machine learning models. These methods also do not help the user package and deploy these models to edge devices for robotic applications. The goal of this project is to design and build a comprehensive tool that will reduce the barrier to entry, reduce the amount of personal computing resources needed to train a model, and make it easy to package models for edge devices.

1.3 Contributions

In this project, we created an Integrated Neural Network Development Environment (INNDiE) that packages together the necessary tools and workflows for neural network development into one tool that can be used by novices with limited machine learning knowledge. To reduce the barrier to entry, INNDiE helps the user select a neural network model that fits their problem using a graphical

wizard. For maximum flexibility, INNDiE also lets the user bypass this system and choose their own model. To reduce the amount of personal computing resources needed to train a model, INNDiE is capable of training models in the cloud. INNDiE is also capable of compiling a trained model for an edge device.

The remainder of the paper is organized as follows. Chapter 2 provides an overview of relevant background information on machine learning and the cloud services INNDiE uses and discusses existing methods for machine learning. Chapter 3 formalizes INNDiE's target audience, requirements, and approach. Chapter 4 presents the results of the approach. Chapter 5 summarizes this work, draws lessons learned, and presents avenues for further development.

Chapter 2

Background

2.1 Machine Learning

To fully understand INNDiE's design and the motivation behind its creation, a few machine learning concepts should be understood. These concepts will be presented in a somewhat linear fashion, with later concepts building on earlier ones.

The primary building block of the artificial neural network is the *layer*. Layers are logical groups of *neurons*. In turn, *neurons* are simple units of computation that have inputs, outputs, and internal state. Connections are created between neurons to form a neural network. Critically, each incoming connection to a neuron has a *weight* associated with it [22]. This weight forms a linear combination with the value of the connection, allowing the connection to be "strengthened" or "weakened" (think of each weight in the network as a dial that can be tuned to control the overall behavior of the network). However, neurons are seldom used directly because they are so simple (and used in such great quantities) that they are difficult to compose into a meaningful system. Instead, layers are used to build neural networks [22]. Because layers are defined as particular groups of neurons, they can form logical operations and therefore can be reasoned about more easily than their components. There are many different kinds of layers; each can perform very different operations and subsequently require very different configuration data.

The activation function is another important part of the neural network. Every neuron in a neural network has an activation function associated with it. This function is responsible for determining what signal the neuron will output when given input [22]. There are many different types of activation functions; different types are used depending on the type of the neuron they will be associated with.

Once a neural network has been assembled, it is of no practical use until it is trained. The optimizer is responsible for this training process. There are many ways of training a neural network: INNDiE is concerned with one called supervised learning. In supervised learning, a labeled data point (i.e., a question and answer pair, like an image of a cat and the label `cat`) is given to the neural network, which processes the data to produce some output (the output of the last layer of the network) [36]. A mathematical function called a *loss* function is then used to compute how correct the output is (relative to the label for the data point). Each neuron in the network is adjusted by the optimizer based on how correct the entire network was citereed1999, chandramouli2018. In other words, the optimizer is responsible for minimizing the loss function over the entire labeled dataset the neural network is trained on. The optimizer does this by tuning the weight associated with each input to each neuron; the final product of the training process, then, is a set of weights.

With any minimization problem comes the question of how to know when the minimum is found. For some problems, called *convex*, this question can be readily answered; unfortunately, neural networks are rarely, if ever, convex optimization problems. Furthermore, if a large neural network is optimized to the point where the loss function is very close to zero, that neural network may have memorized the dataset it was trained on and therefore will be unable to generalize to other data it has not seen before [19]. In other words, there is a balance when optimizing a neural network. The engineer training the neural network has a few primary ways to control this balance; the simplest of which is the number of *epochs*. To complete one epoch, the optimizer will complete a pass over

the entire dataset it was given. Increasing the number of epochs the optimizer runs for will drive the loss function toward zero. However, to avoid the previously stated memorization problem, the engineer can also stop the optimizer if the value of the loss function has not decreased significantly in a certain number of epochs. This is called “early stopping”. The heuristic this is based on works roughly as follows: when the network is learning the dominant features of the dataset, the loss function will decrease rapidly; after that, when the network is refining its understanding of the dataset, the loss function will decrease less and less rapidly. Eventually, training must be stopped to prevent the network from memorizing the dataset.

The neural network, comprised of layers of neurons; the optimizer, with its own internal state; the loss function; and the weights produced by the optimizer are organized into a container called a *model* [8]. Models can be used to save, share, and deploy neural networks. There are many formats a model can take. Which format is used depends on what tools the engineer used to create and train the neural network. Unfortunately, although these formats largely describe the same data (at a high level), it is very difficult to convert a model from one format to another. Instead, if a neural network from one framework must be replicated in another framework, the structure is reprogrammed in the new framework and the weights are copied over; there is still no standard format for the weights, but this data is typically easier to convert.

2.2 Amazon Web Services

INNDiE primarily makes use of two Amazon Web Services (AWS) services: Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3). EC2 provides configurable virtual machines. A user tells EC2 what type of virtual machine they want: they can specify the computer power and memory, among many other things. EC2 then creates a virtual machine that matches the user’s description and gives them full access to it as if it was their own personal server, enabling the user to shift their workloads into the cloud. INNDiE uses EC2 to run the computationally intensive process of training neural networks.

S3 provides file storage. A user can upload or download files to or from S3 and organize them into folders. S3 partitions its storage into *buckets*: logical divisions of storage that contain files (optionally) grouped into folders. Each bucket has a set of access permissions the user can set to control how different AWS services are allowed to interact with files in the bucket. INNDiE uses S3 to store everything EC2 needs to access when running the training process: the script used to train the model, the model itself, the dataset, and the training results.

2.3 Related Applications

This table summarizes the way other applications compare to INNDiE:

	Weka	MLJAR	Classification Learner	AutoML	INNDiE
Data Preprocessing		✓	✓	✓ *	✓
Bring Your Own Model					✓
Train in the Cloud		✓	✓	✓	✓
Convert Model for an Edge Device				✓	✓
No Programming Needed	✓ **	✓		✓	✓

* AutoML offers limited data preprocessing but still requires the user to generate some complex files to accompany their dataset.

** A significant part of Weka’s functionality requires programming.

Weka is an application for data mining made by the Machine Learning Group at the University of Waikato [18]. It can perform various data mining tasks without requiring any programming, but is primarily built for data mining and therefore does not excel at training neural networks and does not let the user bring their own neural network. Weka also must run on a local computer.

MLJAR is a web application for data mining [13]. It helps the user pre-process their dataset to fill in missing values, convert categorical data, etc. It can compare performance of different machine learning methods (e.g., random forest, logistic regression, k-nearest neighbor, neural network) against a dataset; however, you cannot

bring your own neural network. Models must be trained on MLJAR's servers. Users can download trained models or use a REST API to access the model on MLJAR's servers.

MathWorks' Classification Learner is also targeted at data mining [12]. The user cannot bring their own model: they must select one from a list including decision tree, bayes classifier, support vector machine, nearest neighbor, etc. This list does not include neural networks. Training is done on MathWorks' servers.

Google's AutoML is targeted at more than just data mining; it includes first-class support for neural networks [1]. The user is expected to upload their dataset to Google's servers and select a model from a list of supported models; the user cannot bring their own model. Training is done on Google's servers. The user can choose to host their model on the Google Cloud and use a REST API for inference or download the model for an edge device.

Chapter 3

Methods

3.1 Problem Formulation

3.1.1 Target Audience

INNDiE's ideal target user is familiar with a neural network's basic structure of layers and connections between layers. This user is familiar with the concepts of training and inference. This user also has a machine learning problem to solve and associated dataset fit for that problem.

This target user may use INNDiE to solve a machine learning problem by using INNDiE's job wizard to start with a standard configuration relevant to their problem and tweak it from there. This user may also use INNDiE to become more familiar with the basics of machine learning by varying the number of epochs and seeing how it affects accuracy, playing with different optimizers and loss functions and configurations, and configuring or replacing layers in their model.

3.1.2 Requirements and Approach

1. **Create, train, and test jobs** INNDiE must encapsulate a model, dataset, model configuration (i.e., hyperparameters, callbacks), and training results (i.e., accuracy, loss, trained model), in a structure called a job. A user must be able to view their jobs, create new jobs, delete jobs, edit jobs that have not been started, run jobs to start the training process, cancel jobs that are currently being trained, view training progress on jobs that are in progress, and test jobs that have finished training. Once a job has started training or has finished training, the job is no longer editable; it becomes a record of an experiment the user ran. To meet this requirement, INNDiE features a job editor that helps the user control every part of their job configuration. This editor is structured in such a way that the user can grasp a high-level view of the configuration without diving into the details unless they want to. Jobs are also run and cancelled using this editor. INNDiE also features a results view that collects training progress and all training results into one location so that the user can view their model's accuracy over time and all results. Finally, INNDiE also features a testing view the user can use once their model has finished training. Tests can be started by selecting which test data to infer with and which plugins to use to load the test data and process the model's output. Test results are visualized in this view.
2. **Easy job creation** INNDiE must have a way to help a user with limited machine learning experience create a job that they can use to start solving their machine learning problem. To meet this requirement, INNDiE features a job wizard that guides the user through selecting a problem type (e.x., classification, regression), dataset (e.x., MNIST), and other configuration data. INNDiE infers the model, optimizer, loss function, and other data from the user's high-level selections so that they do not need to understand these concepts to get started.
3. **Persistent jobs** INNDiE's jobs must persist between application runs. To meet this requirement, INNDiE accesses an embedded H2 database via a Java Database Connectivity (JDBC) connection. To make database access typesafe, INNDiE uses `Exposed` [4], a lightweight SQL library written for Kotlin.
4. **Preferences** INNDiE must support configurable user preferences. These preferences must be persistent and must be editable using INNDiE's UI. One example preference is the default EC2 node type used when running a job. To meet this requirement, preference persistence is handled by an interface, `PreferencesManager`. This interface provides the functionality to save/load preferences while keeping the storage method opaque. There are two implementations: `S3PreferencesManager`, which synchronizes a local preferences cache with an object in S3, and `LocalPreferencesManager`, which keeps the

preferences locally in a file inside INNDiE's folder inside WPILib's default folder. The preferences are displayed to the user in INNDiE's user interface in the preferences view.

5. **Load HDF5 models** INNDiE must be able to load a Keras SavedModel after it has been exported by TensorFlow into an `HDF5` file [9]. TensorFlow `v1.15` must be supported. INNDiE must be able to load both `Sequential` and `General` models. To meet this requirement, INNDiE depends on a library called `jHDF` to handle loading the `HDF5` file format [6]. When TensorFlow saves a model to an `HDF5` file, the model's layers are serialized into `JSON` using a format specified by TensorFlow and then stored as a `Dataset` under the attribute `model_config`. Using `jHDF`, INNDiE parses the `HDF5` file and parses the `model_config` data as a byte stream back into its original `JSON` format[1]. From there, INNDiE parses the `JSON` into an instance of INNDiE's `Model` class; this is implemented by hand to provide maximum flexibility to handle unexpected changes in TensorFlow's format. Both `Sequential` and `General` formats are supported; layer loading code is shared between them.
6. **Example datasets** INNDiE must be able to load TensorFlow's example datasets. These datasets will be loaded as part of the training script. The user should only need to select which example dataset to load in the job configuration. To meet this requirement, INNDiE captures TensorFlow's example datasets in a sealed class hierarchy called `ExampleDataset`. Instances of this class can be converted to the equivalent code during code generation. TensorFlow's example dataset API handles loading and caching the actual data in the dataset.
7. **Custom datasets** INNDiE must be able to load a dataset in such a way that it can be fed into the model during training. To meet this requirement, INNDiE requires the user to select a plugin to process the dataset so they can configure it before it is given to the model. During code generation, this plugin is used to account for all of the dataset-specific logic.
8. **Save trained models** INNDiE must be able to save a model after training is finished. To meet this requirement, INNDiE emits code that uses the TensorFlow API to save the entire model (layers, weights, etc.) using the `SaveModelTask`. Using the TensorFlow API save the model (instead of creating our own implementation) ensures that the model will be compatible with the TensorFlow API the user will use to deploy their model and with INNDiE's model loading system.
9. **Save checkpoints** INNDiE must support TensorFlow's training checkpoint mechanism [16]. To meet this requirement, INNDiE emits code that creates, configures, and attaches TensorFlow's `ModelCheckpoint` callback using the `CheckpointCallbackTask`.
10. **Early stopping** INNDiE must support stopping training early if the model stops learning. To meet this requirement, INNDiE emits code that creates, configures, and attaches TensorFlow's `EarlyStopping` callback using the `EarlyStoppingTask`.
11. **Visualize layers** INNDiE must support visualizing a model's layers in a graphical fashion (i.e., a higher-level representation than text; for example, a graph of nodes where each node is a layer). To meet this requirement, INNDiE loads the model's layers into a graphical layout in which each layer is shown as a rectangle, colored according to the type of the layer, with a text label on it stating the type of the layer. When the user click on a layer, an editor opens on the side of the window with layer-specific controls (i.e., a checkbox to freeze or unfreeze the layer).
12. **Modify layers** INNDiE must have a method to freeze and unfreeze a model's layers. Frozen layers may not have their weights adjusted by the optimizer during training. To meet this requirement, INNDiE loads a model into an internal data structure, a `Model`, which itself contains instances of `Layer` in a data structure depending on the type of the model, that it can manipulate more intelligently than `JSON`. After the user has configured their layer operations in the UI, the new set of layers is built and given to the backend along with the original set of layers from the `Model` that was loaded previously. These sets are compared to find the delta between them, which is codified using a set of `LayerOperation`. This `LayerOperation` set is then used to generate the code to create and configure the new model.[2]
13. **Train locally** INNDiE must support running the training script locally. In this mode, the local computer's hardware resources should be used for training the model. To meet this requirement, INNDiE writes the training script into a file in a local cache. This script is then run in a Docker container with various directories in INNDiE's local cache mapped into directories in the container so that the script has access to models, datasets, and other things it may need, without requiring the host to copy these things into the container.
14. **Train with AWS** INNDiE must support running the training script on AWS. In this mode, AWS is used for training the model. To meet this requirement, INNDiE uploads the training script to S3 and starts a new EC2 instance with a script that:

[1] This `JSON` parsing is handled by `Klaxon` [10]

[2] The word "set" is used loosely here because the layers are only stored in an ordered set for `Sequential` models. In the case of a `General` model, the layers are stored in a graph. Still, the uniqueness of each layer is maintained.

(a) Installs various dependencies (e.g., Python, Docker, INNDiE's CLI)

(b) Initializes progress reporting

(c) Downloads the model, dataset, and training script from S3

(d) Executes the training script in a Docker container

(e) Uploads the training results to S3

(f) Completes progress reporting

(g) Terminates the EC2 instance

Also, because EC2 instances use billable storage space while shut down, the instances INNDiE creates are configured to always terminate on shutdown using the AWS API. Terminated instances do not use any storage space, so the user will not be billed for old instances. All of this logic is captured in the `EC2TrainingScriptRunner` class.

15. **Test the model** INNDiE must allow the user to test their trained model with sample data. This sample data should be imported by the user and selected in the test view before running a test. The test results should be presented to the user in INNDiE's UI. INNDiE should be able to visualize at least `png`, `jpeg`, and `csv` test result formats. There are three components that work together to meet this requirement. First, because the user's sample data can be in any format, the user must also select a plugin to load that data into a format that can be given to the model being tested. Second, because the model's output can be in any format, the user must also select a plugin to process that output into files that INNDiE can load and present to the user. Third, when INNDiE loads test results, it looks at the files' types to determine whether it can visualize them. Any files that INNDiE can visualize will be visualized using the appropriate method (i.e., images are loaded into image views, csv files are loaded into charts). Users also have the option to navigate to the path of the test results in their native file browser.
16. **Test the code** All major functionality in INNDiE's backend must be tested. Code which must interact with external APIs (like AWS) can have integration tests that are run manually (because they need some sort of supervision and authentication). These tests can also only be run periodically to save costs; i.e., before every release. All other code must have a mix of unit and integration tests with at least 70% line coverage (this coverage number excludes the aforementioned code that interacts with external APIs because that code cannot be tested in CI). To meet this requirement, INNDiE uses a number of development methodologies and testing tools which are elaborated on in Section 4.1.
17. **Support extensions** INNDiE's implementation must be extensible to support backwards compatible TensorFlow updates. If TensorFlow adds a new layer, optimizer, loss function, activation function, or initializer, support for it should be able to be added to INNDiE purely with code additions (i.e., with no code modifications). If users want support for a new model format, support for that format should be able to be added without changing code outside of the `tf-layer-loader` module. To meet this requirement, INNDiE uses interfaces for all data structures and/or algorithms that interface with or model data from TensorFlow. Model loading (e.g., from an `HDF5` file) is handled by the `ModelLoader` interface. There are sealed class hierarchies for datasets, optimizers, loss functions, activation functions, constraints, data formats, initializers, layers, regularizers, interpolation, padding, and verbosity levels. Modeling these data as sealed class hierarchies makes the implementation more robust to change: for example, if a developer adds an optimizer, the compiler will emit an error stating that not all possible optimizer subclasses have been handled in the relevant areas. This makes it easier for new developers to understand where to modify the implementation when adding support for something new and makes extending the implementation less error-prone.

    Additionally, the entire code generation architecture is structured for maximum extensibility. This starts with the code generator, which operates on a model composed of variables and tasks. Variables are units of storage that tasks can use to share information. Tasks are units of computation; they can represent a function, a block of code, or some combination of both. This architecture provides extensibility because it allows the developer to encapsulate code generation functionality into small, testable units that do not depend on each other at a source level. These units can be combined into larger "programs" through a Kotlin DSL. If the developer needs to add a new code generation feature (e.x., to add support for a new TensorFlow feature), they just need to add a new task implementation and insert it into the "program" much like a programmer might declare, implement, and call a new function in their program.

    The user interface is also extensible. Future developers can easily add additional paths to the wizard by specifying the workflow name, description, and output configuration. If TensorFlow adds add a new optimizer, loss function, activation function, or initializer, a future developer would only need to add an editable data class and associated editor for those features on the job editor pane. If needed, a new step can be added to the job wizard to configure new features.

3.2 Design

3.2.1 Tooling

INNDiE is built on the JVM because of its massive community and adoption, making it a very stable and well-supported platform. The JVM has a proven track record of providing enterprise-class stability while also providing cross-platform support. The JVM also has a rich ecosystem of tooling and libraries, which helps us work efficiently and avoid reinventing the wheel when we need a feature that Kotlin's or Java's standard library lacks.

INNDiE is also built using Kotlin because it provides a number of valuable language-level features. First, Kotlin has first-class support for coroutines. Coroutines make asynchronous programming easy and succinct. INNDiE's UI makes use of coroutines to perform slow operations without blocking the UI thread and without needing to manually manage multiple other threads, which is much more expensive (resource-wise) than using coroutines. Second, Kotlin has a null-safe type system, meaning that nullity is captured at the type level. This helps developers write safer code and avoid unexpected null pointer exceptions. Finally, Kotlin code is succinct; the developer can write less code to accomplish the same task as a Java developer can with more code. In particular, Kotlin has less boilerplate than Java. This reduction in verbosity makes it easy to read Kotlin code because the developer's intent is clear, rather than being hid behind boilerplate code.

3.2.2 Architecture

INNDiE's architecture is separated into three primary layers shown in Figure 3.1.

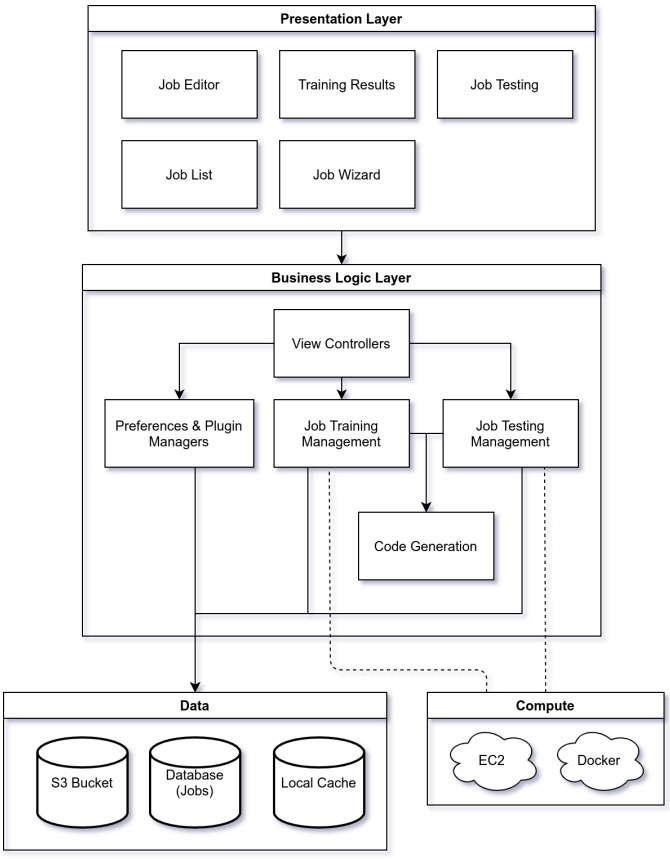

Figure 3.1: INNDiE's architecture.

Presentation Layer

The presentation layer is responsible for the user interface and all user interaction. It contains the job editor view, the training results view, the job testing view, the job list, the job wizard, and the preferences dialog. Users interact with this layer directly to create, delete, edit, train, and test jobs. Users can also modify their preferences and plugins.

Job Editor The job editor is responsible for displaying all the parameters of the currently selected job in an editable fashion. It is also responsible for calling into the business logic layer to initiate training on EC2 or Docker or to cancel training.

Results View The training results view is responsible for displaying the progress data of the currently selected job. This progress data is automatically updated by the business logic layer when the job is training. Primarily, this view updates its progress graph each time new progress data is available.

Testing View The job testing view is responsible for accepting configuration data about what data the currently selected job should be tested on and what plugins should be used to process this data and interpret the trained model's output. This view is also responsible for passing this configuration data to the business logic layer to initiate a test. Once a test finishes, this view must append the latest test data to its list of tests.

Job List The job list is responsible for displaying the name and progress of all jobs in the database; it also contains two buttons to create a new job or to delete the currently selected job. Deleting a job requires sending a delete event to the database. Creating a job requires opening the job wizard; if the wizard returns a job, then the job list will asynchronously tell the database to create that job. The job list also subscribes to the database so that it receives create, update, and delete events; these events are reflected in the displayed job list.

Job Wizard The job wizard is responsible for guiding a novice user through creating a job. Its task and input views are populated by its controller. The data that populates these views contains not only what will be displayed to the user (i.e., the task/input name and a picture) but also what hyperparameters should be inferred given the user's selection. For example, the classification task has three inputs associated with it (`MNIST`, `Fashion MNIST`, and `IMDB`), each of which have an associated optimizer and loss function. This removes the need for the user to understand how to choose an optimizer and loss function.

Preferences Dialog The preferences dialog is responsible for the user's preferences and plugins. Changes to preferences and plugins are given to the business logic layer to be persisted.

Business Layer

The business logic layer is responsible for controlling the presentation layer, generating code, and interacting with EC2 or Docker to train or test jobs. This layer is also responsible for interacting S3, the job database, and the local cache to save training and testing results, preferences, and plugins.

Preferences and Plugin Managers The preferences manager and plugin managers are responsible for saving their respective settings. Because INNDiE needs to support training both locally and in the cloud, there is a local- and an S3-based implementation of the preferences and plugin managers. The local implementation saves the relevant information to the local cache. The S3 implementation is a decorator over the local implementation which also pushes changes to S3.

Job Training Management The job life cycle manager is called from the presentation layer with the id of the job to train and the desired training method (either locally or on EC2). It will immediately update the job status in the database to indicate that the job has started training so the presentation layer can provide immediate feedback to the user that the job is starting. It will then delegate to a job runner to start the job using the relevant implementation for either local- or cloud-based training. If the job finishes training with an exception, the life cycle manager will cancel the job to ensure that any training resources are freed.

When INNDiE starts and the job life cycle manager is initialized, it fetches all running jobs from the database and resumes progress reporting for those jobs. This provides continuity to the user. When they close INNDiE while training a job, training will continue in the background. Once reopened, INNDiE will pick up where it left off.

Job Testing Management Job testing management is similar to job training management. When the user requests to run a test, the testing configuration is captured from the presentation layer and passed to the test runner. The test runner will generate a testing script to run the test with the user's configuration. It will then use Docker to run the test. Tests are expected to be small and complete quickly, so neither progress reporting nor canceling are supported, which makes the test runner's implementation significantly simpler than that of the job runner's. The test results are stored in the local cache. The user can navigate to the results for analysis. The results are loaded when the test view is displayed to the user.

Code Generation INNDiE uses code generation for generating scripts for training and testing jobs. The code generator operates on a model composed of variables and tasks. Variables are units of storage that tasks can use to share information. Tasks are units of computation; they can represent a function, a block of code, an expression, or some combination of those. Tasks can be linked together through implicit or explicit dependencies. Implicit dependencies are formed by variables: when one task outputs to a variable which is the input to another task, a dependency is formed between those tasks. Explicit dependencies come from adding a task as a dependency of another task by the developer modifying a task's dependencies. The code generator ensures that the dependency graph formed by the tasks is a singular directed acyclic graph. It then traverses the graph to generate the code for each task, generating the code for a task's dependencies before generating the code for that task.

User Interface Library Change

The user interface for INNDiE was initially developed with `Vaadin` [17]. The team decided to switch to `JavaFX` halfway through the project because:

1. Vaadin does not integrate cleanly with our build tool, Gradle. Vaadin does not release standard Maven packages. The library requires a complex build pipeline that was not easily maintainable with Gradle.
2. Our requirements transitioned from needing a web interface. When we discovered that a way to run INNDiE locally was important to our users, the need to use a web framework was reduced. Vaadin is a web library and while the user could have started a webserver on their computer, it would be a poor user experience.
3. We did not have prior experience with Vaadin before embarking on developing INNDiE. It is important to have a resource of support with any dependency on a project. We did not have any support resources (aside from official documentation) or experts in Vaadin available to us.

Chapter 4

Findings and Analysis

4.1 Experimental Setup

4.1.1 Test-Driven Development

Test-Driven Development (TDD) is the practice of using automated testing to guide development [26]. In this practice, tests are written before implementation code to first define the behavior of a unit before that unit is implemented. Implementation code is only changes to make those tests pass. After all tests pass, a refactoring step is performed to simplify and clean up the implementation.

IBM evaluated their use of TDD in a software development team and found that there was a 50% reduction in defect density versus the previous ad-hoc testing practice (adding tests where necessary after implementing a unit) [30]. IBM also found that the project stayed on schedule, the product design was more extensible and avoided late integration problems, and that developers enjoyed the TDD practice.

Janzen and Saiedian studied the effect of TDD on software quality by comparing two groups of developers from varying backgrounds: one which wrote tests first (before implementation) and one which wrote tests last (after implementation) [27]. They found that test-first programmers wrote statistically significantly smaller classes and methods, according to the cyclomatic complexity and nested blocked depth complexity metrics.

This research shows that TDD is an effective development practice; therefore, we strive to practice it during INNDiE's development. We use a number of tools to assist our practice, the most important of which is JUnit5, a framework for writing automated tests [7]. We use JUnit5 to write all of our unit and integration tests. JUnit5 provides some standard assertions, but because we use Kotlin, we prefer to use the assertions from KotlinTest: a test framework that integrates well with Kotlin [11]. Similarly, we also use MockK, a mocking framework that integrates well with Kotlin [14]. Finally, to collect code coverage metrics so we can roughly estimate how comprehensive our automated tests are, we use JaCoCo [5]. Code coverage is collected and reported every time INNDiE is built.

4.1.2 Static Analysis

Static analysis is defined as "the automatic compile-time determination of run-time properties of programs" [23]. Although the general form of the static analysis problem is undecidable, modern tools use good approximations to perform well despite this fundamental limitation. Microsoft evaluated the effectiveness of using static analysis to detect defects and found that the defect density reported from static analysis correlated statistically significantly with the real defect density measured from testing [31]. Google undertook a large software review using the FindBugs static analysis tool and found that over 77% of the defects reported by the tool were real defects that should be fixed [20].

We use the tool detekt [3] for static analysis of INNDiE's Kotlin code. Static analysis is run every time INNDiE is built; if any module has an issue count over a certain threshold, the build for that module fails. This build failure discourages a developer from merging a pull request until they fix the offending module so that the static analysis passes.

4.1.3 Mutation Testing

Mutation testing is defined as a way to test the effectiveness of a test suite by making small syntactic changes to an implementation's source code and verifying that the corresponding test suite fails [28]. In their work, Jia found evidence that the Coupling Effect hypothesis holds for the purposes of mutation testing. Offutt [33] defined the Coupling Effect hypothesis as

> *Coupling Effect*: Complex faults are *coupled* to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults.

One can then see how mutation testing, which artificially introduces simple faults, can detect complex faults through the same mechanism.

We use the tool Pitest [15] for mutation testing. Pitest is a mutation testing system that is capable of multi-threaded execution, configurable mutation operators, and other features. It integrates with INNDiE's build system and is run on-demand. We do not run Pitest on every build because, although Pitest is fast an efficient, mutation testing is an inherently difficult process that requires significant compute and memory resources. Running Pitest every build would significantly slow down the build process.

4.1.4 Continuous Integration

Continuous Integration (CI) is defined in [24] as

> a software development practice where members of a team integrate their work frequently. . . Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

CI catches integration problems early and therefore helps the team work in an agile manner. This is critical for small teams with limited time like ours. We use Azure Pipelines, a CI solution from Microsoft Azure [2]. Azure Pipelines integrates with GitHub so that it starts a build on every commit, tag, and pull request. Building on every commit provides the developer with constant feedback while they work. Building on every tag ensures that releases are only cut if their build passes, providing a final quality assurance check for the last integration before releasing the software. Finally, building on every pull request ensures that code being merged into the mainline is correct and meets the team's quality standards. Pull requests should only be merged once their build passes.

We have Azure Pipelines configured to run three jobs are part of the build process: building on Windows, macOS, and Linux ensures that INNDiE is compatible with the major desktop operating systems.

4.2 Results

4.2.1 Unit Tests

1. To test 5 (load `HDF5` models), `MobilenetV2` (a `General` model) exported from TensorFlow versions `1.14` and `1.15`, a custom `Sequential` model, a custom `General` model, and an invalid model are used for testing the entire model loading process. Additionally, many small models, each with one layer designed to test a specific part of the model loading logic like loading an `Initializer` or `Regularizer` are used for testing. Each test verifies that the loaded model has the correct name, input and/or output (if applicable), and layers.
2. To test 12 (modify layers), the two `Task` subclasses that are responsible for generating the code to apply a set of `LayerOperation`, `ApplySequentialLayerDeltaTask` and `ApplyGeneralLayerDeltaTask`, are tested using many different combinations of models and layer operations: no operations (i.e., copying all layers), removing, adding, freezing, and swapping layers, and copying unknown layers or layers with unknown components. Additionally, for the `ApplyGeneralLayerDeltaTask`, partially deleting inputs and outputs, reordering inputs, and adding outputs that depend on each other are also tested.
3. To test 14 (train with AWS), interactions with the AWS API are mocked so that starting a script, polling for progress updates, and cancelling a script can be tested. The behavior of the `EC2TrainingScriptRunner` is tested and interactions with the mock AWS API are verified.

4.2.2 Integration Tests

1. Design 1 (create, train, and test jobs) is met using a job editor view seen in Figure 4.1 that aggregates controls for each data point contained in a job. Jobs can be run by selecting a platform to run on using the combo box in the run button and then pressing the run button. Starting a job makes it immutable and moves it into the *creating* progress state, reflected in the progress bar underneath the job name. Once a job finishes, it transitions to either a success or error state which are indicated in the progress bar with either a green or red fill color, respectively. Training progress is also reflected in the progress bar; a more detailed progress overview is available in the results view 4.2. The primary feature of this view is the line chart which shows all the metrics that are reported during training; typically, loss and accuracy per epoch are shown. Once the job has finished training and the user wants to test it on sample

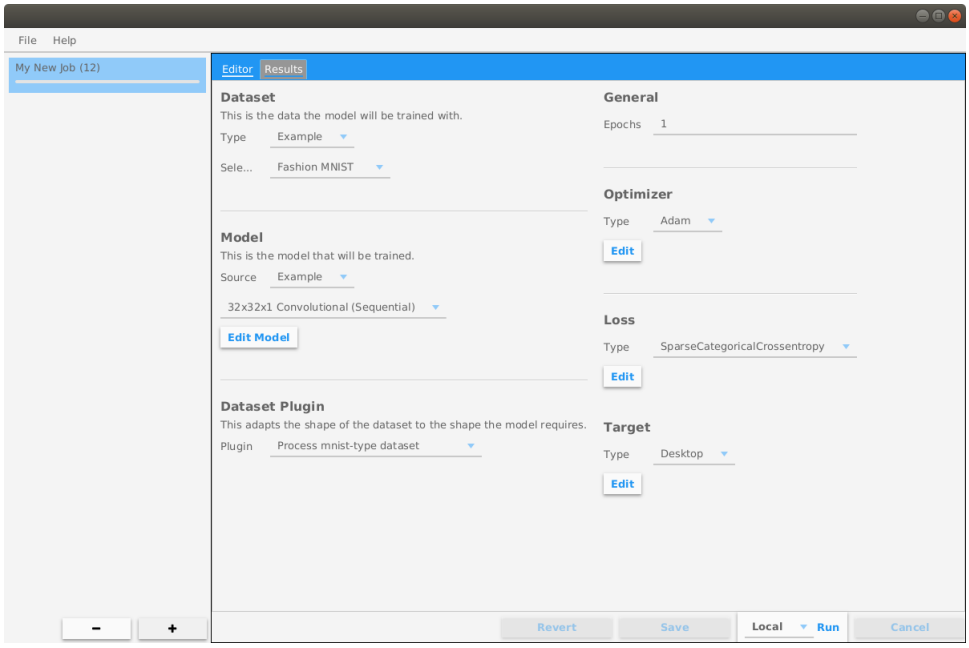

Figure 4.1: The job Editor with a newly created job selected.

data, they can navigate to the test view and see its output visualized on their sample data. This test view also meets design 15. The test view is shown in Figure 4.3.

2. Design 2 (easy job creation) is met using a job wizard that guides the user through configuring a model for their problem. The input selection step of this wizard is shown in Figure 4.4. Once the user finishes the wizard, their job is created and they are taken to the job editor to make finer-detail changes if they need to.
3. Design 3 (persistent jobs) is met by storing jobs in a database saved to a file on the user's machine. When INNDiE starts, it loads jobs from this database into the job list view. As the user modifies jobs, their modifications are committed to the database so that their latest changes are always saved on disk.
4. Design 4 (preferences) is met using a preferences dialog, seen in Figure 4.5. All of INNDiE's preferences are accessible using this dialog.
5. Design 11 (visualize layers) is met by visualizing layers as nodes in a graph, seen in Figure 4.6. Each layer is a colored rectangle; the color corresponds to the type of the layer. The name of the layer's type and other relevant information is also put into the rectangle. When a layer is selected, its editor is opened, allowing the user to edit properties of the layer, like whether it is trainable.
6. Design 14 (train with AWS) is tested using the real AWS API instead of a mock, jobs are started and polled for their progress. After the job finishes, its status should be `Completed` and the trained model file should exist in the job's outputs. These tests require a valid authentication to AWS because they use the real AWS API to interact with S3 and start EC2 instances. These tests are also on timeouts because any bugs in the code could cause the test to hang.

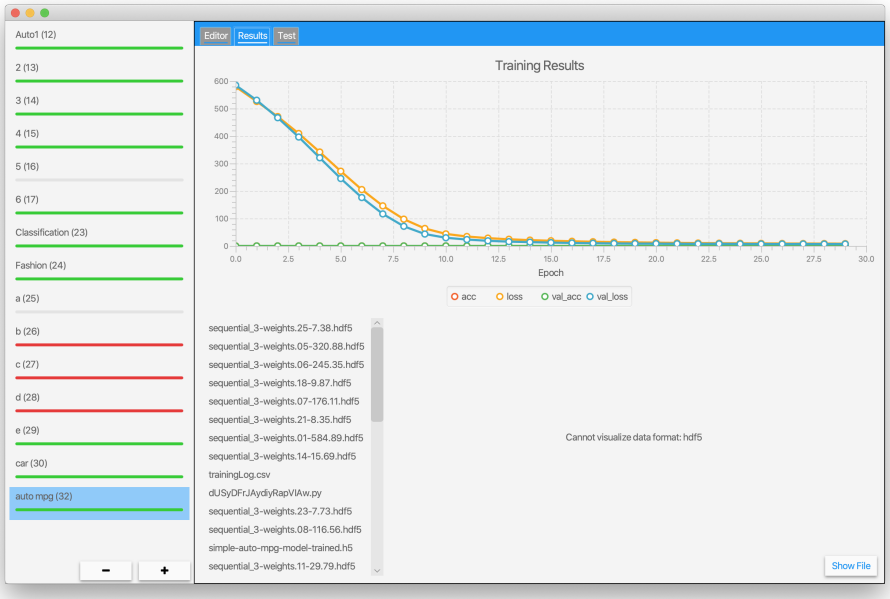

Figure 4.2: The results view.

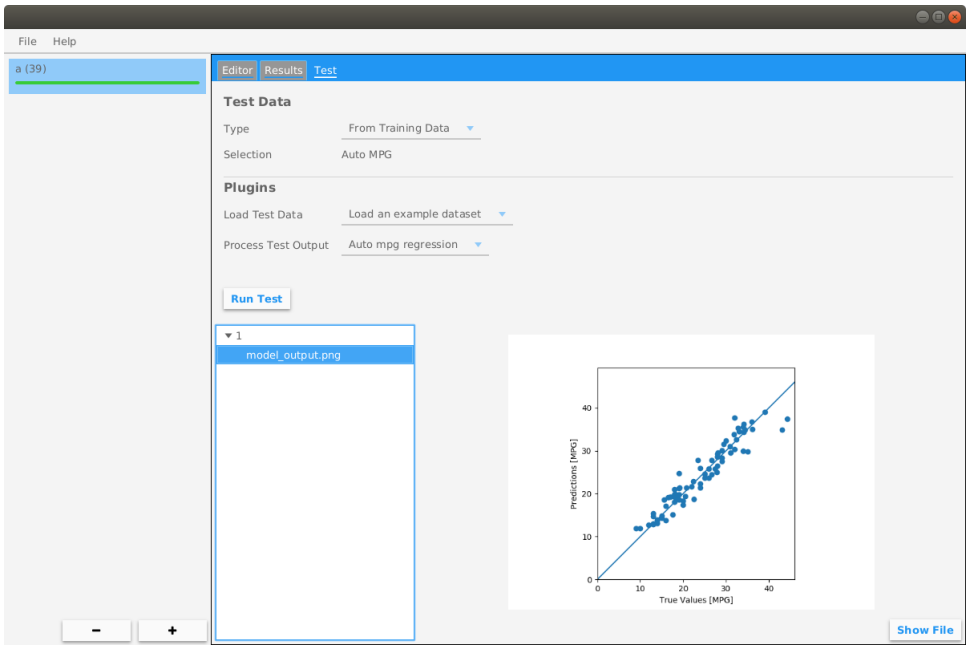

Figure 4.3: The test view.

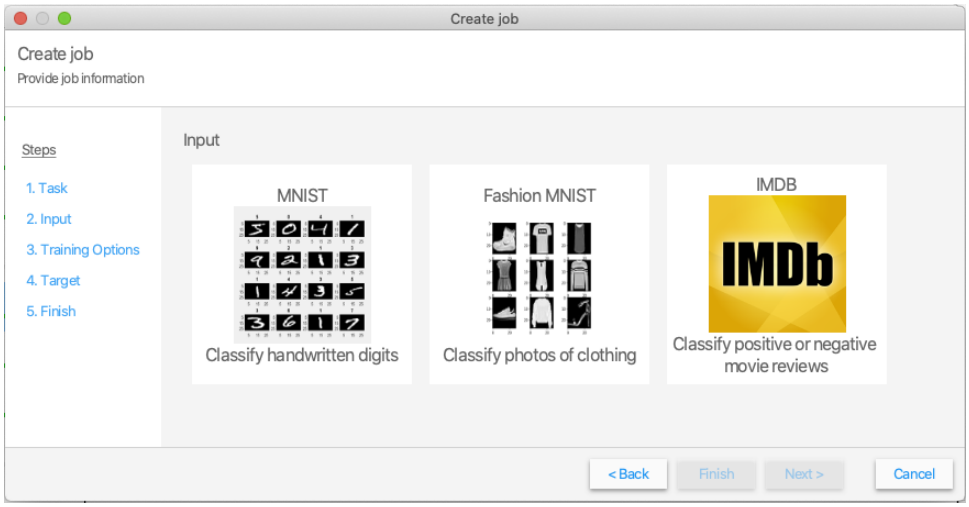

Figure 4.4: The job wizard's input view.

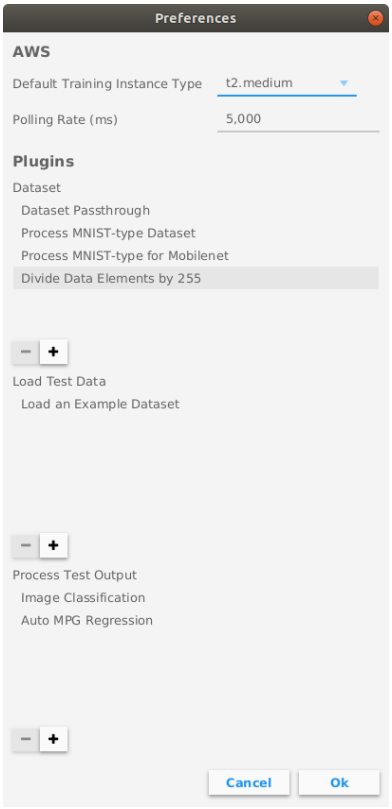

Figure 4.5: The preferences dialog.

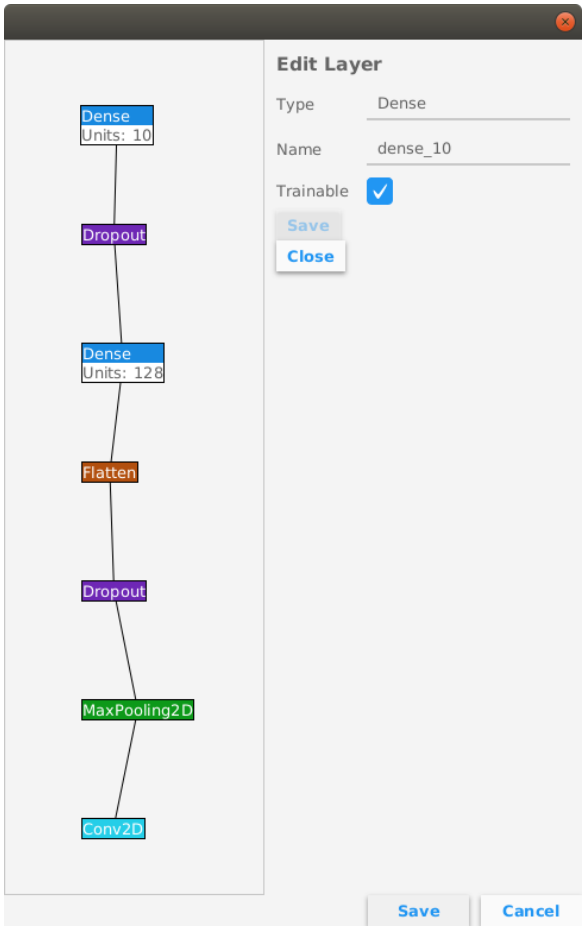

Figure 4.6: The layer editor with a layer selected.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

5.1.1 Summary of the Methods

1. INNDiE sorts data into jobs
   (a) INNDiE encapsulates a model, dataset, model configuration (e.x., hyperparameters, callbacks), and training results (e.g., accuracy, loss, trained model), in a structure called a job.
2. Jobs can be created with a wizard
   (a) Jobs can be created using a wizard that guides the user through selecting a problem type (e.g., classification, regression), dataset (e.g., MNIST), and other configuration data. INNDiE infers the model, optimizer, loss function, and other data from the user's high-level selections so that they do not need to understand these concepts to get started.
   (b) After creation, jobs can also be configured at a very fine level of detail using the job editor. This is targeted at advanced users who wish to have this level of control.
3. Jobs can be trained and tested
   (a) Jobs can be trained either locally on the user's machine or in the cloud on EC2. Training progress is reported dynamically and graphed.
   (b) Jobs can be tested on custom data using the user's machine. Test results are visualized graphically.

5.1.2 Lessons Learned

1. Effective teamwork requires constant communication. Over the course of working on this project, our team deliberated over many of the project's requirements, but nonetheless found that we still did not come to a unanimous understanding. Sometimes this disconnect was obvious; at other times, it was hidden and surfaced later. The unifying problem in all of these misunderstandings was a lack of communication and transparency between all of the team members.
2. Don't commit to a technology without fully understanding its weaknesses. In a project with a strict timeline, there is a relatively little amount of acceptable overhead for learning new technologies and encountering unexpected bugs in those technologies. At the beginning of the project, we were over this limit due to the combination of the multiple new technologies we adopted. It is important to rely on proven technologies in order to iterate quickly and not get bogged down learning new concepts and encountering bugs at every turn.
3. Limiting the scope of a problem makes a solution easier to find. Initially, our goal was to build a completely modular system that works with any machine learning challenge. This proved to be a difficult goal to grasp because the use cases we created were loosely specified. We pivoted during the course of the project to focus on classification and regression in order to reduce the scope of our initial goal while keeping the system modular.

5.2 Future Work

1. INNDiE should be able to train and test an object detector. The current design cannot reasonably support object detectors because models that perform object detection require significantly more support to train

and deploy than models such as image classifiers. INNDiE currently loads a dataset, pre-processes each element using a single operation a the dataset plugin (e.g., a simple operation like scaling images or changing color domains), and trains a model directly from that data with no other data processing steps between the dataset plugin and the model. In order to have INNDiE effectively train an object detector, data processing must allow for multiple steps, INNDiE must have control over generating data batches, and INNDiE must have control over how the output of the model is interpreted because the labels from the dataset do not contain enough information.

2. The layer editor should be extended to support addition, modification, or removal of layers. Layer modification includes adding or removing connections to other layers, modifying activation or loss functions, optimizers, and any others layer-specific hyperparameters. Any hyperparameters that were not modified should be functionally identical to the input model. *Unknown* layers, ones which INNDiE was not programmed to understand[1], may only be removed (or simply not modified in any way, except for (un)freezing them). Duplication or modification of unknown layers cannot be supported. None of the above layer operations may ever modify the input model. Instead, all operations should be put into a new model. INNDiE's backend is already capable of this; only a frontend addition is needed.
3. When the user selects a custom dataset, INNDiE should try to visualize the data. For example, if the dataset contains annotated images, then the user should be able to browse a grid of those images and see the annotations on them. This visualization should also have a plugin system to support other datasets.
4. INNDiE should infer which layers the user should add/remove/modify. For example, the user loads a model and a dataset and wants to perform transfer learning, INNDiE might suggest that the user remove the last layer and add a new `Dense` layer with `10` neurons and a `softmax` activation function (if that modification is appropriate for the dataset). The user should be able to accept this modification with the press of a button (the modification is then performed automatically).
5. INNDiE should be able to detect invalid configurations. For example, using the sparse categorical crossentropy loss function with a classifier that instead needs the categorical crossentropy loss function; a common mistake.
6. There should be support for multiple networks combined in some fashion. For example, two networks that run in parallel, the inputs to which are the inputs to the system and are combined with a third network, the output of which is the output of the system.

[1] There are two main cases in which INNDiE could encounter an unknown layer. The first case is if TensorFlow adds a new layer type and a user loads a model which contains this new layer type before INNDiE is updated with support for it. The second case is if the user loads a model which contains a custom layer that has been written by hand. TensorFlow includes support for implementing custom layers as classes, and will put the implementation code into the layer when it is serialized. In either of these cases, INNDiE can do no better than allow the user to (un)freeze, keep, or remove the layer.

Bibliography

[1] Automl. https://cloud.google.com/automl/.

[2] Azure pipelines. https://azure.microsoft.com/en-us/services/devops/pipelines/.

[3] detekt. https://github.com/arturbosch/detekt.

[4] Exposed. https://github.com/JetBrains/Exposed.

[5] Jacoco. https://www.jacoco.org/.

[6] jhdf. https://github.com/jamesmudd/jhdf.

[7] Junit5. https://junit.org/junit5/.

[8] Keras model. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/keras/Model.

[9] Keras savedmodel. https://www.tensorflow.org/guide/saved_model.

[10] Klaxon. https://github.com/cbeust/klaxon.

[11] Kotlintest. https://github.com/kotlintest/kotlintest.

[12] Mathworks classification learner. https://www.mathworks.com/solutions/machine-learning.html.

[13] Mljar. https://mljar.com/.

[14] Mockk. https://mockk.io/.

[15] Pitest. http://pitest.org/.

[16] Tensorflow checkpoints. https://www.tensorflow.org/guide/checkpoint.

[17] Vaadin. https://vaadin.com/.

[18] Weka. https://www.cs.waikato.ac.nz/ml/weka/.

[19] Devansh Arpit, Stanisław Jastrze, Maxinder S Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and Simon Lacoste-Julien. A closer look at memorization in deep networks. page 10.

[20] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252, 2010.

[21] Antonio Brunetti, Domenico Buongiorno, Gianpaolo Francesco Trotta, and Vitoantonio Bevilacqua. Computer vision and deep learning techniques for pedestrian detection and tracking: A survey. *Neurocomputing*, 300:17–33, 2018.

[22] Subramanian Chandramouli, Saikat Dutt, Amita Dāśa, and an O'Reilly Media Company Safari. *Machine Learning*. 2018.

[23] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, pages 159–179. Springer, 2002.

[24] Martin Fowler. Continuous integration, May 2006.

[25] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[26] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[27] David Janzen and Hossein Saiedian. Does test-driven development really improve software design quality? *Ieee Software*, 25(2):77–84, 2008.

[28] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[29] Daniel Lorencik and Peter Sincak. Cloud robotics: Current trends and possible use as a service. In *2013 IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 85–88. IEEE, 2013.

[30] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569. IEEE, 2003.

[31] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[32] Anh Nguyen, Jeff Clune, Yoshua Bengio, Alexey Dosovitskiy, and Jason Yosinski. Plug & play generative networks: Conditional iterative generation of images in latent space. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4467–4477, 2017.

[33] A Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[34] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[35] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[36] Russell D. Reed. *Neural smithing: supervised learning in feedforward artificial neural networks*. MIT Press, 1999.

[37] J Sieber and B Krauskopf. Extending the permissible control loop latency for the controlled inverted pendulum. *Dynamical Systems*, 20(2):189–199, 2005.

[38] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[39] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

Glossary

Docker A set of services that provide lightweight, containerized virtualization. 7, 8, 9, 10

Acronyms

AWS Amazon Web Services. 4, 7, 8, 13, 14

EC2 Amazon Elastic Compute Cloud. 4, 6, 7, 8, 9, 10, 14, 18

JDBC Java Database Connectivity. 6

S3 Amazon Simple Storage Service. 4, 6, 7, 8, 10, 14