ANNA: Artificial Neural Network Abundances
User Guide

Donald Lee-Brown
donald[at]ku.edu
Department of Physics & Astronomy
The University of Kansas

August 18, 2017

Contents

1 Overview

ANNA (Lee-Brown et al. in prep) is a Python program capable of automatically parameterizing 1D, continuum-fit, stellar spectra, calculating atmospheric parameters such as surface temperature and gravity, [Fe/H], and microturbulent and rotational velocities. It determines these parameters through use of a convolutional neural network (CNN), a machine learning technique where a model with many free parameters is algorithmically trained to accurately translate between input data and desired outputs. Notably, an early goal during the writing of ANNA was to produce a tool flexible enough for a wide range of use cases. Thus, despite its name, ANNA is capable of being trained to infer an arbitrary number of parameters from stellar spectra, provided those parameters are encoded in the spectra. ANNA is also computationally efficient and highly scalable, and can be used to analyze a single spectrum or many thousands of spectra. Finally, efforts have been made to make it as user-friendly as possible, and there are several easy-to-use diagnostic utilities included with the program. ANNA is open-source, and I hope that through the feedback and help of others in the astronomical community, it will continue to evolve and improve over its useful lifetime.

This is the extraneously comprehensive user manual for ANNA. Some brief background information and an outline of the various capabilities and design choices associated with the program are given in Section 2. Installation instructions and dependencies are given in Section 3. Section 4 contains descriptions of the various parts of ANNA and outlines the use of the program. Several appendices are also included in this manual. Appendix A contains a walkthrough of the program using the included example - users interested in quickly testing the program without worrying about all the bells and whistles may want to skip to this section after installation. Appendix B makes an attempt to address potential issues, most of which were learned through personal experience. Finally, the manual concludes with a list of subroutines and files in Appendix C.

Disclaimer: This manual is a work-in-progress, and will likely undergo significant revision as ANNA moves towards a stable release version.

2 Algorithm Description

2.1 Neural Networks

An artificial neural network (ANN) is a non-linear computational model capable of mapping inputs to a set of outputs whose values are encoded in the inputs. This mapping is accomplished via a series of nested, interconnected mathematical operations whose organization was originally inspired by early models of biological nervous systems (see e.g., Haykin, 1998). It has been shown that ANNs are "universal approximators" capable of approximating any well-defined function (see e.g., Bishop, 1995). As such, ANNs have proven to be exceptionally versatile data analysis tools, and have been successfully used in

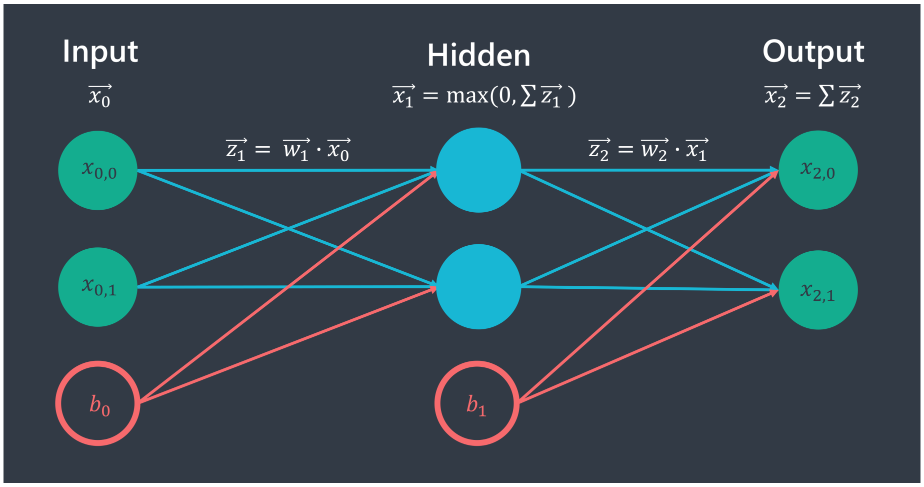

Input
$\overrightarrow{x_0}$
Hidden
$\overrightarrow{x_1} = \max(0, \sum \overrightarrow{z_1})$
Output
$\overrightarrow{x_2} = \sum \overrightarrow{z_2}$
$\overrightarrow{z_1} = \overrightarrow{w_1} \cdot \overrightarrow{x_0}$
$\overrightarrow{z_2} = \overrightarrow{w_2} \cdot \overrightarrow{x_1}$
$x_{0,0}$
$x_{0,1}$
b_0
b_1
$x_{2,0}$
$x_{2,1}$

Figure 1: example architecture

many data-driven fields, including astronomy (Bailer-Jones et al., 1997; Allende Prieto et al., 2000; Snider et al., 2001; Bailer-Jones et al., 2002; Manteiga et al., 2010; Dafonte et al., 2016; Li et al., 2017).

A simple ANN architecture is shown in Fig 1. The basic element of an ANN is a *node*, which represents a mathematical operation that transforms input data into an output, or activation. Nodes are organized into layers, such that the nodes in one layer receive inputs from the preceding layer and pass their activations to the next layer. This structure of sequential layers is known as a feedforward neural network. If each node activation in one layer is dependent on every node activation in the preceding layer, then the network is *fully-connected*. Thus, the network shown in Fig 1 is a fully-connected, feedfoward ANN.

The *input layer* of a feedforward network is simply the input data (e.g., pixel values) to be analyzed. Each node in the input layer simply passes an input datum to the next layer. Following the input layer are one or more *hidden layers*. Each node in a hidden layer receives activations from the previous layer, each multiplied by some associated weight. These weighted inputs are then summed, and some non-linear function (the *activation function*) is applied to this sum and returned as the node activation. Thus, the activations of hidden layer nodes are non-linear sums of activations of the previous layer's nodes. In the network shown in Fig 1, there is a single hidden layer. Because the network in Fig 1 is fully-connected, the nodes in the hidden layer each receive weighted values from every node in the input layer.

The exception to the rule of every node in a layer being connected to one or more activations of the previous layer's nodes is the *bias node*. Unlike the other nodes in a layer, the bias node's activation does not depend on previous layer activations, and serves a role analogous to that of the constant b in the simple linear function $h(x) = ax + b$. However, the bias node's activation is weighted and propogated to the next layer as with the other node activations.

A common choice for the activation function is the sigmoid, defined as $f(\vec{w}, \vec{x}) = 1 + e^{-a \times \Sigma \vec{w} \cdot \vec{x}}$, where $\vec{w}$ and $\vec{x}$ are the weights and associated activations from the previous layer and a is some constant. Another, less computationally demanding choice is the *rectified linear unit*, or ReLU, defined as $g(\vec{w}, \vec{x}) = \max(0, \Sigma \vec{w} \cdot \vec{x})$. There are many other viable options as well (e.g., hyperbolic tangent); the primary purpose of the activation function is to inject non-linearity into the ANN (Bishop, 1995). In the network shown in Fig 1, a ReLU activation function is used.

The final layer of the network is the *output layer.* Nodes in the output layer receive weighted activations from the preceding hidden layer, and depending on the application a non-linear activation function may also be applied to the sums of these inputs. However, the output layer activations are the outputs of the ANN; they correspond to whatever parameters the network was designed to infer from input data (e.g., probability of an image being of a particular object, stellar surface temperature, etc.). In other words, the values of the activations of the nodes in the output layer are of principal interest to anyone using a neural network to analyze data.

For an ANN accurately determine appropriate output values given a set of inputs, the weight values assigned to each activation must be carefully set. Since ANNs often contain many thousands (or even millions or billions) of individual weights, correctly choosing weight values is a computational task. This process is called *training.*

During training, the ANN hyperparameters (number of nodes and layers) are set, and the network is given a number of example inputs whose outputs are known via some other method. The weights in the network are set to their initial values (usually small, random numbers), and the training examples are fed through the network, producing outputs. These inferred outputs $\vec{x}$ can then be quantitatively compared with the the known output values $\vec{y}$ via a *cost function*, $J(\vec{x}, \vec{y})$, where $\vec{x}$ is itself a function of the network weights and inputs. The specific form of the cost function depends on what the network is being used to do (e.g., classify, produce continuous-valued outputs, etc.).

Once the cost has been calculated, training the network is reduced to solving a minimization problem, where the function to be minimized is $J(\vec{x}, \vec{y})$. The most straightforward method of doing this is to calculate the gradient of the cost function with respect to each weight in the network. This gradient can be used to update each weight with the magnitude of the update controlled by a parameter known as the network *learning rate.* Once the weights have been updated, training data can again be fed through the network and another weight update performed. Over many update steps, the network weights will converge to their optimal values and the ANN will be capable of correctly translating between inputs and outputs. This processing of training the network is known as *backpropagation.* While straightforward to understand, it is computationally expensive due to the large number of gradient calculations required and static learning rate; newer training algorithms operate using the same general principle but employ various schemes to speed up the weight update process (see examples in, e.g., Simonyan & Zisserman, 2014).

The method of training a neural network using examples with known outputs falls into the category of *supervised learning.* It is critically important that the training examples (and their known outputs) be representative of the data the network will be used to analyze. It is also crucial to have sufficient training data; the network weights are effectively free parameters that must be carefully tuned, so the training data must be sufficiently comprehensive to allow the network to map out (using the weights) the relations between inputs and outputs.

This is but a basic overview of neural networks and their use, for more comprehensive discussions, interested readers are referred to: Bishop (1995), Haykin (1998), or for a discussion of neural networks in the context of astronomy, see Bailer-Jones et al. (2002).

2.2 Description of ANNA

ANNA was largely inspired by the examples set by several (successful) previous explorations of neural networks as tools for spectroscopic analysis (e.g., Bailer-Jones et al., 1997). It has been shown that ANNs are capable of characterizing spectra with accuracies rivaling contemporary techniques (template fitting, EW analysis) (Recio-Blanco et al., 2016). However, studies have tended to be exploratory in nature (e.g., Bailer-Jones et al., 1997; Snider et al., 2001; Li et al., 2017) or specifically focused on analysis of a large survey dataset (e.g., Manteiga et al., 2010; Dafonte et al., 2016; Recio-Blanco et al., 2016). In contrast, ANNA was explicitly built to be a general framework for neural network-based spectroscopic analysis, enabling the analysis of a wide range of spectroscopic data.

As such, ANNA was designed to be as flexible and user-friendly as possible, while still providing a well-developed tool for a wide range of use cases. Some key features of ANNA are:

1. ANNA can be trained to extract an arbitrary number of parameters from input 1D spectra. For example, a user wishing only to use ANNA to determine surface temperatures may do so, while a user wishing to determine surface temperatures, gravities, microturbulent velocities, etc. may also do so.

2. The program was designed to be trained using either synthetic spectra or real spectra. Thus, it provides some capability to modify supplied synthetic spectra to better mimic real spectra.

3. The user retains control over many architectural aspects of ANNA, and can tune many of the neural network parameters to offer the best blend of performance and accuracy for a given dataset.

4. Once trained, ANNA models are automatically saved, can be moved between computers, and are generally small enough to be sent via email. Once moved to a different computer, the models can straightforwardly be used.

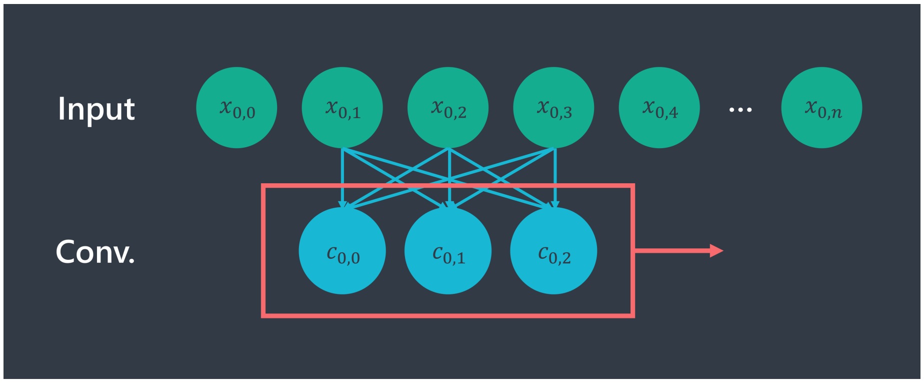


Figure 2: example conv

5. The neural network architecture used by ANNA incorporates many modern developments in machine learning, which together improve ANNA's accuracy and ability to generalize while reducing the time needed to train.

6. ANNA is entirely built using common Python packages, while using Google's TensorFlow deep learning library for the neural network routines. This makes ANNA compatible with both Linux and macOS systems, and relatively easy to install. It is capable of carrying out computations on a supported Nvidia graphics processing unit (GPU) if one is available, leading to orders of magnitude faster operation.

7. Care was taken during the design process to ensure that ANNA is capable of running on modest hardware. Virtually any modern PC hardware is capable of running ANNA.

8. ANNA includes several utilities to visualize and test neural network performance, allowing users to quickly tune ANNA and verify determined parameter accuracy.

The neural network at the core of ANNA is a *convolutional* neural network (CNN), an ANN variant that has enjoyed significant success in solving computer vision problems (see Simonyan & Zisserman, 2014). In contrast to the simple fully-connected networks described in the previous section, a CNN contains one or more hidden layers that apply one or more convolutional kernels to the layer inputs. These convolutional kernels, or filters, can be thought of as a small collection of nodes that only receive inputs from a small, localized subsample of inputs, as shown in Fig 2. The filters are "slid" across the entire range of inputs to the layer, such that every input is connected to the convolutional filter at some point, but only its neighboring inputs are connected to the filter at the same time.

The convolutional layer ensures that within such layers, a CNN only "sees" correlations between neighboring inputs. This property is ideal for a program

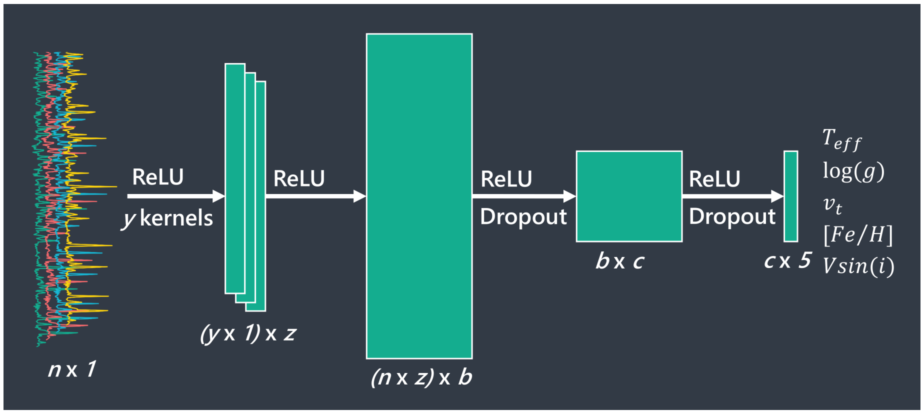


Figure 3: architecture

such as ANNA, since widely spaced pixel values likely have little to do with one another, but spectral lines or line complexes form localized regions of highly correlated pixel values whose variance contains important atmospheric information. While a fully-connected network can eventually be trained to recognize these localized regions as well, the much larger number of free parameters in a fully-connected layer vs. a convolutional layer results in longer training times to achieve the same accuracy as a CNN.

ANNA contains a convolutional layer as the first hidden layer. The outputs of the convolutional layer are then used as inputs into two sequential fully-connected layers, as shown in Fig 3. Thus, the network is still free to learn correlations between widely spaced spectral regions, but the convolutional layer provides a computationally inexpensive way to capture correlations between localized regions. The user is free to determine the number of nodes and filters in all three hidden layers of ANNA.

Between the hidden layers, ANNA employs dropout, a mechanism to prevent overfitting (Srivastava et al., 2014). Dropout involves randomly selecting nodes (according to a user-specified probability) to ignore during a training step. If a node is ignored, it is subsequently re-initialized during the next training step. This process of randomly resetting nodes during training helps avoid the problem of the network becoming overly-dependent on the activations of a few nodes, and improves the network's resiliance to overfitting. Importantly, dropout is computationally cheap and requires minimal tuning to be effective.

The activation function used throughout ANNA is the ReLU, defined in the previous section. This choice of activation function affords a computational advantage relative to more complex activation functions such as the sigmoid, with no appreciable loss in accuracy. The only exception is the output layer of the network, which consists of weighted sums of the layer inputs with no ReLU activation (i.e. the layer uses the identity function as its activation).

During training, ANNA first initializes the network weights according to the scheme outlined in (He et al., 2015). The adopted weight initialization improves the ability of the weights to converge to the optimal solution during training, and was specifically developed for the case of ReLU activation functions. The bias units in the network are initialized to 0.

The cost function adopted for training the network is the squared error, $\Sigma(\vec{y} - \vec{y}')^2$, where $\vec{y}$ and $\vec{y}'$ are the true and inferred parameters, respectively. This cost function was chosen as it is similar to the commonly adopted function to be minimized during regression problems. It is also guaranteed to be convex with respect to each parameter, preventing the network from converging to a local, rather than global, minimum in the cost function.

ANNA uses mini-batching during training. In a traditional training process, the entire training set is used during each training step. This is inefficient, as it requires that every training example be examined by the network before the weights are updated. A much faster training method is to randomly sample the entire training set and only feed a small number of training examples into the network during each weight update step (see discussion and references in Ruder, 2016). Over many training steps, the network will be exposed to the entire training sample, but will be allowed to update its weights during that process. This results in some stochasticity in the cost function with respect to training step, as one mini-batch may be better fit by the current network weights than a different mini-batch, but results in significantly faster training.

The weight optimizer used during training is the Adam Optimizer, a stochastic gradient-based algorithm (Kingma & Ba, 2014). Adam incorporates several features that have been shown to significantly increase training speed, including momentum (Nesterov, 1983), and was designed to be computationally efficient. Empirical tests with ANNA have shown that Adam leads to significantly faster weight convergence when compared with other gradient-based optimization methods.

The aspects of ANNA covered above are largely hard-coded and cannot easily be modified, and they were empirically found to offer the best blend of speed, accuracy, and stability. However, ANNA also includes several user-controlled features designed to improve performance. The practical usage of all these features is detailed in Sec 4.2, but the justifications for the major performance-impacting ones are briefly covered below.

First, ANNA supports the use of a *cross-validation* dataset and early stopping during training. A cross-validation dataset is kept separate from the main pool of training data. At user-defined intervals, the network pauses training to run inference on this data, computing the cross-validation cost. The training the continues until the cross-validation cost has not improved for a user-defined number of iterations. At this point, the network stops training (early stopping), as the cross-validation error is minimized and further decreases in the cost function during training are likely due to overfitting of the training data.

Second, ANNA supports an optional, second training stage. This means that the user can run training with an initial set of network learning parameters, then when the cost is minimized during this stage of training, training can continue with a second set of learning parameters. This allows the user, e.g., to adopt a high initial learning rate or small batch size to quickly and approximately converge the network weights, then switch to a smaller learning rate or larger batch size to “fine-tune” the weights.

ANNA also implements basic input preprocessing, along with TensorFlow’s queue method for importing data. Preprocessing includes optional addition of random noise to input examples, according to a user-supplied, wavelength-dependent template. This is particularly useful when training ANNA using synthetic input data, as it ensures a more or less unique noise solution can be added to each example when it is used during training. This prevents ANNA from “memorizing” the noise associated with each example, as it is recomputed every time the example is used, and can also prevent the network from assigning to much value to low SN spectral regions. Preprocessing incurs a computational cost, but due to the queue method, preprocessing is assigned its own hardware resources that can execute independently of the weight updating process.

Finally, ANNA provides the capability to save trained models in a hardware-agnostic form. The upshot of this is that ANNA can be trained on one computer (perhaps a powerful machine with a GPU) and then the trained model can simply be transferred to another computer (e.g., a laptop), and used for inference, provided the second machine has ANNA installed. Naturally, the saved model is still only applicable to data similar to what was used during training. The save feature also allows for models to be archived and used at a later date.

3 Setup

3.1 Required Software and Hardware

ANNA can be run on either Linux or OSX systems. ANNA was not written to be run on Windows, and it is virtually guaranteed that it will not do so without large modifications to several subroutines.

ANNA is written in Python 3 (3.5+ tested), and it is backwards compatible with Python 2 version 2.7+. The list of packages and their minimum versions required by ANNA is given below:

- numpy version 1.1.0
- tensorflow 1.11.0
- time
- threading
- math
- random
- array
- scipy

Users of Python will almost certainly have these packages installed, possibly with the exception of TensorFlow. If you don't have Python installed, I recommend installing the Anaconda distribution (`www.continuum.io/downloads`) which includes the required packages, except for TensorFlow.

TensorFlow installation instructions can be found at `www.tensorflow.org/install/`, and which version to install is hardware-dependent. Users wishing to try ANNA out or who just need to run inference using a pre-trained model will be best served by installing the CPU-only version of TensorFlow using `pip` or `conda`. This method gets a basic version of TensorFlow installed quickly and without hassle, but forgoes many of the hardware-specific optimizations TensorFlow supports.

If you are installing ANNA on a Linux system, plan to train models, and have a supported Nvidia GPU, it is recommended that you install the GPU enabled version of TensorFlow and the required Nvidia CUDA deep learning libraries. ANNA can utilize a GPU during training, and will generally complete training an order of magnitude faster when doing so. Newer versions of TensorFlow do not support GPU compute on OSX.

Particularly for CPU-only users wishing to train ANNA on their machines, installing TensorFlow from source may be useful. This allows hardware-specific optimizations to be installed, which may speed up the training process. Instructions for doing this can be found on the TensorFlow website.

A note on hardware: ANNA was written to run on modest hardware, but training a neural network is inherently computationally intensive. In particular, users will benefit from additional CPU cores and/or an Nvidia GPU, and in general the more RAM available the better. For example, most of ANNA's development and testing was carried out on a Linux (Scientific Linux) machine with a quad-core Intel CPU and 16 GB RAM for CPU-only testing, and an Linux (Ubuntu) machine with a quad-core Intel CPU, 8 GB of RAM, and an Nvidia GTX 1060 6 GB GPU for GPU testing. OSX testing was conducted using a MacBook Pro with a quad-core Intel CPU, 16 GB of ram, and macOS Sierra.

3.2 Downloading ANNA

Once the required Python packages have been installed, you are ready to download ANNA. The current version of the program can be downloaded from `www.github.com/dleebrown/ANNA`. The minimum files you will need to download are `ANNA.param`, `ANNA_train.py`, `ANNA_test.py`, `ANNA_infer.py`, `convnet2.py`, and `readmultispec.py` [1]. If you plan on following the step-by-step tutorial given in Appendix A, you will also need to download the additional data files listed there.

These files should all be downloaded and placed in the same directory on your computer and run from there using either the terminal or a Python developer

[1] The multispec reader `readmultispec.py` comes from Kevin Gullikson, based originally on a script by Rick White (2012). The script can be found at: Kevin Gullikson. (2014, May 20). General-Scripts_v1.0. Zenodo. `http://doi.org/10.5281/zenodo.10013`.

environment. The parameter file `ANNA.param` includes fields to point to your training/testing/inference/template data. Note that the `convnet2.py` file is required by all three modes of ANNA.

4 Running ANNA

End-to-end use of ANNA occurs in three distinct stages. First, the neural network hyperparameters and preprocessing options must be set, and the network initialized and trained. This is done through use of the `ANNA_train.py` file. Next, it is generally a good idea to make use of the testing capability contained in `ANNA_test.py` to verify that the trained network accurately parameterizes a small but representative sample of the data inference will be run on. Testing is particularly important if the network was trained using synthetic spectra, and ideally it should be conducted using an already-parameterized sample of real spectra, if such a sample exists. This process verifies that the synthetic training spectra mimic the characteristics of the "real-world" data sufficiently enough to avoid any unfortunate systematics when running inference. Finally, the network can be used to infer the parameters of input spectra whose outputs are unknown. This last step is generally the end goal of any analysis, but the training stage has the most impact on the final result.

4.1 The Parameter File

The parameter file `ANNA.param` contains all the various buttons and knobs used to control the behavior of ANNA. The various parameters and their format are listed and described here, grouped by which stage of the program they belong to. More details regarding the specific use of each parameter can be found in the stage-specific subsections for training, testing, and inference.

Training Parameters

- `NUM_PX: integer`
  The number of pixels per training spectrum. Also used during testing and inference.

- `CONV1_SHAPE: 1, integer, 1, integer`
  Shape of the convolutional layer. The first integer corresponds to the width in pixels of each convolutional kernel, the second integer controls the number of convolutional kernels. In general, the optimal width and number of kernels depend on the resolution and wavelength coverage. Kernels should be wide enough to encompass features in the spectrum but not so wide as to also include many line-free pixels. There should be enough kernels pass interesting features to the next layers of the network, but not so many as to introduce overfitting or greatly increase computation time. 10 kernels with widths of 20Å is a good place to start.

- `FC1_OUTDIM: integer`
  The output dimension of the first fully-connected layer (i.e. the number of intputs into the second fully-connected layer). As with the convolutional layer, this should be a number large enough to avoid throwing away spectral information but small enough to avoid overfitting. Increasing the value of this parameter has a strong impact on computation speed, particularly if training is done on the CPU. A good starting value is 256.

- `FC2_OUTDIM: integer`
  Similar to `FC1_OUTDIM` but for the second fully connected layer. Again, 256 is a good starting value.

- `NUM_OUTS: integer`
  The number of outputs from the neural network. The number of outputs here needs to be the same as the number specified in `OUTPUT_NAMES`, `MIN_PARAMS`, and `MAX_PARAMS`.

- `LEARN_RATE1: float`
  The learning rate to use during the first stage of training. This parameter strongly influences the performance of the network and must be carefully chosen, and improper values are responsible for most failures to learn. Too small, and the network will take forever to learn. Too large, and the network will be incapable of learning. There is a lengthy discussion on choosing an appropriate learning rate in Section 4.2. Generally values range from 1e-4 to 1e-8.

- `NUM_TRAIN_ITERS1: integer`
  The maximum number of training iterations for the first stage of training. Note that each iteration corresponds to a single batch of input spectra. In conjunction with early stopping (see below), this value can be set to an arbitrarily high number and ANNA will automatically stop training when the training error stops improving. If early stopping is not used, then this value should be large enough to maximally train the network but not so large as to waste computation time or begin overfitting. Good values for this parameter are generally in the tens to hundreds of thousands.

- `BATCH_SIZE1: integer`
  Number of example spectra to use during each training iteration. Should be large enough to contain a somewhat representative sample of spectra but not so large as to make training take forever. Good default value is 100.

- `KEEP_PROB1: float (0.0, 1.0]`
  Controls dropout, a technique for reducing overfitting. Value indicates the probability that any particular response in the fully connected layers is used during each training iteration. A good value is 0.8.

- `DO_TRAIN2:  YES/NO`
  Specify whether or not to continue to a second round of training with a new set of learning parameters.

- `LEARN_RATE2:  float`
  Learning rate for the second round of training. Generally less than `LEARN_RATE1`.

- `NUM_TRAIN_ITERS2:  integer`
  Number of training iterations for second round of training.

- `BATCH_SIZE2:  integer`
  Batch size for second round of training. Generally larger than `BATCH_SIZE1`.

- `KEEP_PROB2:  float (0.0, 1.0]`
  Probability to keep a given node during each training iteration. Generally larger than `KEEP_PROB1`.

- `PREPROCESS_TRAIN: YES/NO`
  Specifies whether or not to do on-the-fly preprocessing to the training examples as they are batched.

- `REL_CONT_E_TRAIN: -float, float`
  Range to randomly offset continuum level (constant offset per training example) if preprocessing selected. The range should encompass the likely continuum placement error level. For example, -0.02, 0.02 would encompass 2% deviations in continuum placement.

- `SN_RANGE_TRAIN: float, float`
  Signal-to-noise to add to training spectra if preprocessing selected. Noise is assumed to be normally distributed, and a wavelength-dependent relative SN profile can be specified using the `SN_TEMPLATE` option below. The range specified here should mirror the range in SN of likely input data, e.g., 175.0, 250.0.

- `SN_TEMPLATE: string`
  Path to a file containing a wavelength-dependent relative SN profile. The file should be a text file with two whitespace-delimited columns, the first with wavelength values (in Angstroms) and the second with normalized SN for each wavelength value. The way this works is the preprocessing loop draws a SN value for each training example, draws a noise value from a normal distribution for each pixel, then scales each pixel's noise value by the template-specified amount. For example, if the template value is 1.0 at some wavelength, then the noise is unscaled, if the template value is 0.90, the S/N value is multiplied by 0.90, etc.

- `NUM_FETCH: integer`
  Number of training examples to preprocess at once. This should be a large number, like 10,000.

- `TRAINING_DATA: string`
  Path to a 64-bit floating point binary file containing training data in 1-D array form (1 row, n columns). The first entry in the file should be the number of pixels per spectra, the second should be the number of output parameters to train for (the same as `NUM_OUTS`), and third should be the wavelengths, in ascending order, corresponding to each pixel. Following the wavelengths should be the training data, where each example consists of - in this order - star ID, corresponding atmospheric parameters (number equal to `NUM_OUTS`), a dummy entry (for storing, for example, radial velocities), and flux values for each pixel in ascending wavelength order. For example, for spectra consisting of 500 pixels, each example would contain of 507 entries: ID, parameters, dummy, and pixels.

- `SAVE_LOC: string`
  Path to a folder to save and load TensorFlow models.

- `OUTPUT_NAMES: string, string, ...`
  Names of the output parameters, in order supplied in the input training data. For example, one could specify `NUM_OUTS` to be 5, and set `OUTPUT_NAMES` to be temperature, gravity, metallicity, microturb, rotvel.

- `MIN_PARAMS: float, float, ...`
  Minimum values of output parameters in order listed above. These should correspond to the minimum values present in the training data.

- `MAX_PARAMS: float, float, ...`
  Maximum parameter values in the same order as `OUTPUT_PARAMS`.

- `SAMPLE_INTERVAL: integer`
  Interval (in mini-batches) to calculate training progress and log results, if desired. Sampling frequently results in longer training times, sampling infrequently may result in overfitting. A good happy medium is 100 mini-batches.

- `PP_THREADS: integer`
  Number of threads to spawn to preprocess examples. More is not always better. In fact, for spectra of just a few hundred to thousand pixels or any time ANNA is run on a system without a GPU, a single thread will likely give the best performance. This is due to the way Python handles multiple threads. For very large batch sizes, high-resolution spectra, or systems with slower GPUs, increasing this number may be beneficial.

- `MAX_PP_DEPTH: integer`
  Maximum number of preprocessed examples to hold in the training queue. This is essentially a buffer, and should be as large as memory permits - 10,000+ is ideal.

- `LOG_LOC: string`
  Path to a folder to log TensorFlow results.

- TIMELINE_OUTPUT: YES/NO
  Control whether or not to profile the runtime of the ultimate SAMPLE_INTERVAL training iteration and store the results in a .json file in LOG_LOC. This file can then be opened in a web browser and used to examine how long each portion of a training iteration takes, and which device it is run on (CPU/GPU). Useful for identifying performance bottlenecks.

- TBOARD_OUTPUT: YES/NO
  Control whether or not to log results every SAMPLE_INTERVAL to a TensorFlow Timeline file (see Section 4.5) located in LOG_LOC.

- TRAINING_XVAL: YES/NO
  Control whether or not to perform cross-validation on a dataset distinct from the training set. If YES, then ANNA will additionally calculate the cost associated with this cross-validation dataset every SAMPLE_INTERVAL iterations. It is recommended to use this option, as it provides an independent means to evaluate training performance.

- XVAL_DATA: string
  Path to a binary file containing the cross-validation data. The file should be the same format as TRAINING_DATA.

- XVAL_SIZE: integer
  Number of examples to read in from the cross-validation binary file to use as a cross-validation set. This option is useful for controlling memory used by the cross-validation set and reducing computational overhead. The first XVAL_SIZE examples are read in. Several thousand examples is usually sufficient.

- XV_THREADS: integer
  Number of threads to spawn for cross-validation dataset construction. Usually one is sufficient (see discussion under PP_THREADS above).

- PREPROCESS_XVAL: YES/NO
  Perform preprocessing identical to that used for training on the cross-validation dataset. This option should be set to NO if real spectra are used to cross-validate.

- EARLY_STOP: YES/NO
  Control whether or not to use early stopping during training. If set to YES, then training is automatically stopped when ES_SAMPLE number of sampling intervals is reached with no improvement in network accuracy. The cross-validation set is used to evaluate network accuracy, if one is specified, otherwise the current training batch is used. Highly recommended to use this option, as it prevents overfitting and can reduce training time.

- ES_SAMPLE: integer
  Maximum number of sampling intervals that may go by without improvement in network accuracy before training is ended, if EARLY_STOP is set

to YES. A good rule of thumb is to set this such that `ES_SAMPLE` × `SAMPLE_INTERVAL` is ~ 10% of the total number of training iterations.

Testing Parameters

- `TEST_MODEL_LOC: string`
  Location to load a saved TensorFlow model from - note that it should point to the directory holding the saved model, not the saved model itself. If training, testing, and inference are being done on the same computer, then this is generally the same as `SAVE_LOC`.

- `TEST_SAVE_LOC: string`
  Location to save a star-by-star summary of testing results as a .csv file.

- `TESTING_DATA: string`
  Path to a binary file of the same form as `TRAINING_DATA` to use during testing.

- `PREPROCESS_TEST: YES/NO`
  Control whether or not to preprocess test examples using the parameters given below. Should be set to NO if real spectra are used to test network accuracy.

- `REL_CONT_E_TEST: -float, float`
  Same as `REL_CONT_E_TRAIN`, except for the testing data.

- `SN_RANGE_TEST: float, float`
  Same as `SN_RANGE_TRAIN`, except for the testing data.

- `SN_TEMPLATE_TEST: string`
  Same as `SN_TEMPLATE_TRAIN`, except for the testing data.

Inference Parameters

- `DEBUG_MODE: YES/NO`
  If set to YES, then `DEBUG_DATA` must be specified, and inference will be run on the spectra contained within that file. This option offers a way to verify everything is going as it should by running inference on data with known outputs.

- `DEBUG_DATA: string`
  Path to a binary file of the same format as `TRAINING_DATA` to use if `DEBUG_MODE` is set to YES.

- `SINGLE_MS_MODE: YES/NO`
  If set to YES, then a single multispec fits file, specified below, will be read in for inference.

- `MS_FITS_IMAGE: string`
  If `SINGLE_MS_MODE` is set to YES, then this points to the multispec file to use for inference.

- `WAVE_TEMPLATE: string`
  Path to a two-column, whitespace-delimited text file containing the wavelengths used during training. The wavelengths should be in the first column; the second column is not used but is specified here for convenience, so that any training example consisting of wavelengths and fluxes may be used as a template. Generally the wavelengths corresponding to each pixel in spectra used for inference are not the same as the wavelengths used during training - this file provides a wavelength grid to interpolate the inference data fluxe values onto.

- `INFER_MODEL_LOC: string`
  Path to a file containing the TensorFlow model to load and use for inference.

- `INFER_SAVE_LOC: string`
  Path to a folder to output inference results.

4.2 Training

Before ANNA can be used to infer atmospheric parameters from spectra, it must be "trained" on a representative set of spectra with known outputs. This process results in a model capable of accurately mapping input pixel values to desired outputs. Training is done using `ANNA_train.py` and is controlled using `ANNA.param`.

4.2.1 The Training Data

The first step to train a model is to assemble the data to train the network with. This data should all be stored in a 64-bit floating point binary file and pointed to with the `TRAINING_DATA` parameter. This binary file must be of a certain form for ANNA to correctly work, and will be read in as a 1-D array. The first entry in the binary file should be the number of pixels in each spectrum. Each spectrum stored in the binary file must have this many flux values. The second entry should be the number of output parameters per spectrum. The next series of entries should be the wavelength values associated with each pixel in the training spectra, in angstroms. Note that these values do not have to correspond to the exact wavelength values of the data you wish to run inference on, as inference data is interpolated onto the wavelength grid used during training. However, the wavelength resolution of the training data should be approximately the same as the inference data.

After this "preamble" to the binary file, the rest of the file should consist of training spectra and parameters. Each training example should consist of n+x+2 entries, where n is the number of pixels per spectrum and x is the number of output parameters. The entries are to be written in this order: first, a float identifying the training example, followed by the output atmospheric parameters (unnormalized) in the same order for each star and in the same order given in `OUTPUT_NAMES`, `MIN_PARAMS`, and `MAX_PARAMS`. After the atmospheric parameters, there is a dummy entry that can be used to store a number that won't be considered during training (e.g., radial velocity, S/N, or a quality flag). Finally, after these 7 entries, the continuum-normalized flux values should be listed in ascending wavelength order. Immediately following the flux values, the next training example entry should start.

The number of training values should be relatively large (10,000+, ideally), and should cover the entire range of atmospheric parameters your inference data is likely to span. During training, ANNA will randomly select a subset (with repetition) of the training set for each training iteration. If you have too few training examples, ANNA will be unable to learn the relationships between flux values and atmospheric parameters and will likely just "memorize" your training set. If your training examples do not span a wide enough range in atmospheric parameters, then ANNA will not be able parameterize spectra with atmospheric parameters outside the training domain.

The training data do not have to be real spectra. In fact, it's unlikely that you have tens of thousands of pre-parameterized spectra lying around to train ANNA with. However, particularly for optical spectra of main-sequence stars, modern stellar spectroscopic synthesis codes do a pretty good job of replicating actual spectra. Thus, ANNA can be trained on synthetic stellar spectra and then used to parameterize real spectra. All you need to do is use your favorite synthesis program and atmospheric models to generate a grid of synthetic spectra, repack them into the binary file format, and you can use them to train ANNA. For example, SPECTRUM, by Richard Gray of the Department of Physics and Astronomy, Appalachian State University (`http://www.appstate.edu/~grayro/spectrum/spectrum.html`) provides the ability to batch synthesize spectra of specified resolution using standard Kurucz models.

There are several important things to keep in mind when using synthetic spectra. First, to accurately train ANNA, the training data must be representative of the data used for inference. This means that your synthetic training spectra *must* capture the various idiosyncrasies of your inference data - wavelength-dependent S/N, continuum placement errors, resolution, dispersion profile, radial velocity shifts, etc. They must do this in a relatively accurate way - even small deviations can result in poor end results.

Fortunately, these issues are largely independent of one another and can be handled individually. SPECTRUM, for example, allows one to specify a resolution and line-spread function with which to postprocess high-resolution synthetic spectra. Radial velocity shifts are relatively easy to compute, and the resulting shifted spectra can be reinterpolated onto the training wavelength grid. There are tentative plans to make available the "in-house" tools I've developed to do these things automatically, and this manual will be updated when that time occurs.

ANNA includes several functions to further (pre)process training data as it is read in. If you are training on synthetic spectra, it is likely that your input data is noiseless and has perfect continuum placement - nice things that are sadly absent in real data. To preprocess training examples, set the parameter

PREPROCESS_TRAIN to YES. Then set the REL_CONT_E_TRAIN parameter to introduce random continuum offsets to the training data, and SN_RANGE_TRAIN to the desired S/N range to randomly downgrade each training example within. Finally, point the SN_TEMPLATE field to a text file containing two columns - wavelength (in angstroms) and associated relative S/N. This file is extremely important to get right, as it captures the wavelength-dependent response of the spectrograph the inference data come from. The associated relative S/N column should contain S/N values normalized to 1.0, which are then used to construct a wavelength-dependent noise profile. A relative S/N value of 1.0 results in S/N at that wavelength equal to the value drawn from the range specified by SN_RANGE_TRAIN, a relative value of 0.9 results in S/N equal to the drawn value times 0.9, etc.

The easiest way to generate the S/N template file is to obtain a high-resolution spectrum of a hot star with few lines, then compute the S/N at each pixel location according to c/σ, where c is the continuum level and σ is the standard devation computed using a few neighboring pixels. You might consider using a clipped sigma value to reject any errant cosmic rays or strong lines. On that subject, ANNA has no current capability to handle addition of cosmic rays to training examples, so some care must be taken to clean your inference data of cosmic rays.

If you do elect to train ANNA using real spectra, then you should set PREPROCESS_TRAIN to NO and you do not need to supply a noise template. However, note that because the noise and continuum are not recomputed every time a training example is read in, there is the danger of ANNA memorizing an example's specific noise and continuum values, resulting in poor generalization. The solution to this is to train on a sufficiently large number of training examples. Alternately, you can use very high S/N spectra and preprocess them as if they were synthetic spectra using PREPROCESS_TRAIN.

The final parameters to specify regarding the training data are MIN_PARAMS and MAX_PARAMS, which contain the approximate min/max parameter values present in your training data (in the same order as OUTPUT_NAMES). Neural networks learn much more effectively when output parameters are normalized to the same dynamic range; these parameters take care of that. Output parameters are all normalized to the range [0, 1] before training, using the ranges provided by MIN_PARAMS and MAX_PARAMS. This normalization must be identical for training and inference data.

4.2.2 Running ANNA_train.py

Once your data is all ready to go, you can begin the process of training ANNA. There are many knobs to tweak during this stage; this section will only cover ones related to algorithm performance; users should reference Section 4.1 for parameters related to computational aspects.

The first step of training is to define the architecture of the neural network to be trained. This means you need to set the sizes of the weight matrices used to translate between fluxes and outputs. The first is easy; NUM_PIX should be set to exactly what it sounds like - the number of pixels in each training example. CONV1_SHAPE refers to the shape of the convolutional layer of the network and should include each convolutional kernel's width (in pixels), as well as the total number of convolutional kernels (for a description of what the layers do, refer to Section 1). FC1_OUTDIM and FC2_OUTDIM control the sizes of the fully-connected layers in the neural network. Finally, NUM_OUTS should be the number of output parameters.

Sensible defaults for the layer dimensions are included in ANNA.param. Note that increasing the sizes of the layers in general does not always improve network accuracy, and comes with a large computational cost. If layers are too large, you are liable to just overfit your training data and spend a lot of time waiting for the network to train. On the other hand, if your layers are too small, you will be unable to capture the variance present in your training data. So try the defaults first, and change them if things don't go well.

The next step is to specify the other network hyperparameters. LEARN_RATE1 controls the magnitude of weight updates every training iteration; it should be a small value around 1e-6. Too large and the network will fail to train, too small and the training will take a very long time. You also need to specify the total number of training iterations using NUM_TRAIN_ITERS1. This should be set to some large number that will result in every training example being used multiple times, and is used as an upper limit on the number of iterations when EARLY_STOP is used (see below).

Next is BATCH_SIZE1; this controls the number of examples to be randomly selected from the total training set and used to train the network during a given iteration. Appropriate choice of this number follows the same rules governing the sizes of the layers in the network; too large and training will take a very long time with no increase in accuracy, but too small and training may be ineffective. Ideally, the batch size should be large enough to capture a somewhat representative sample of the training data. As with the layer sizes, sensible default values are included in ANNA.param for each of these parameters.

The final hyperparameter to specify is KEEP_PROB1. This parameter controls dropout, a technique to minimize overfitting. This is done by randomly ignoring some subset of the network's responses during each training iteration, scaling the remaining responses appropriately. This effectively reduces the dependency of the network on any single response's value, increasing the overall robustness and reducing overfitting. Thus, KEEP_PROB1, should be set to some value less than 1.0, though go too low and the network will not learn well unless it has very large layers, since most of the responses are being dropped out every iteration. Values to keep a response are usually around 0.8.

ANNA supports two-stage training, controlled by the DO_TRAIN2 field. If set to YES, then the network will be additionally trained using a second set of hyperparameters given in the parameter file, while holding the overall architecture constant. This is useful for speeding up training - a large learning rate allows the network weights to quickly converge on their optimal values but then causes them to "bounce around" with each batch. A smaller learning rate reduces this random variance by making the weights less volatile, but slows convergence.

Two-stage training combines the best of both worlds - weights quickly converge due to a large first-stage learning rate, and then converge more precisely due to a small second-stage learning rate. You might try increasing the batch size or decreasing dropout for the second stage of training, as well. The number of iterations for the second stage of training should be somewhat smaller than the number for the first stage, since the weights are already roughly correct.

During training, ANNA self-evaluates and prints its accuracy in intervals specified by SAMPLE_INTERVAL (given in number of batches). This allows you to verify that the network is training successfully, and gives an idea of how much time is required until training is finished. By default, ANNA evaluates its accuracy using the current batch. However, it is better to provide a data set independent of the training data with which to evaluate training progress. Such a dataset is alternately referred to as a "watchlist" or "cross-validation set"; the latter term is adopted here. Monitoring training progress using a cross-validation dataset helps ensure that overfitting is not occurring - if the accuracy on the current batch keeps increasing but the cross validation accuracy is constant, than the network is merely overfitting the training data.

If you wish to use a cross-validation set, flip the TRAINING_XVAL parameter to YES and point XVAL_DATA to a binary file of the same form as used for storing training data. You can opt to use synthetic spectra for cross-validation, in which case you can preprocess them in the same was as the training data through use of the PREPROCESS_XVAL field. Alternately, you can use a cross-validation dataset consisting of real spectra, in which case PREPROCESS_XVAL should be set to NO.

Finally, ANNA supports early stopping. This is yet another technique for avoiding overfitting, this time by stopping training when network accuracy hasn't improved after a specified number of training iterations. This is particularly useful when you have a cross-validation set as it allows one to stop training once cross-validation accuracy stops improving, even if the current batch accuracy continues to improve due to overfitting. Early stopping is controlled via the EARLY_STOP parameter, and the number of sampling intervals (controlled by SAMPLE_INTERVAL) the cost must improve within is specified by ES_SAMPLE. If a cross-validation set is specified, then ANNA will use it to control early stopping. Otherwise training batch accuracy is used. A good starting point when using early stopping is to set ES_SAMPLE such that at most 10% of the maximum training iterations go by without accuracy improvement. Thus, under this rule, ES_SAMPLE × SAMPLE_INTERVAL should equal 10% of NUM_TRAIN_ITERS1.

ANNA includes several optional (but recommended) diagnostic and visual tools to help you monitor training and ensure everything is running smoothly. The parameter fields LOG_LOC, TIMELINE_OUTPUT, and TBOARD_OUTPUT refer to these functions and are described in Section 4.5.

After ANNA has completed training, it will automatically save the trained model. Trained models are output to a file called frozen.model in the location specified by the SAVE_LOC. These models can then be transferred (along with the ANNA.param file) to other computers and used for inference.

At last! Once you have worked your work way through setting all these parameter values, you are ready to train ANNA. All that remains is to double-check that ANNA_train.py, convnet2.py, and ANNA.param are in the same directory, and that the parameter file points to the appropriate input data and templates. Then you can run ANNA_train.py from the terminal or a Python development environment, and ANNA will read in the input data and begin training. It is a good idea to stick around for the first few iterations and:

- Ensure there are no "looks like input array contains extra entries!" errors (see Appendix B).
- Verify that the cost is large initially, but rapidly decreases over the first few iterations.
- If you are using TensorBoard to monitor training progress, that it is running correctly (see Section 4.5).

During training, ANNA will print the current cost of the network at the intervals you specified in the parameter file. This cost is actually the RMS error for the current batch, and for the cross-validation set, if you specified it. Don't be too worried if the cost jumps around a bit - the important thing is that on average, it should be decreasing over time.

4.3 Testing

In principle, once training is completed once, the output model will be applicable to any data with similar SN, wavelength coverage, resolution, and output parameters as the training data. For example, once a model is trained on data representative of FGK dwarfs measured by spectrograph X operating in mode X, then any future FGK measurements by the same insrument in the same mode can be accurately parameterized. In practice, however, the training process may have to be carried out several times in order to arrive at the optimal set of network hyperparameters. While cross-validation during training provides a useful windows into how well the trained model will generalize, ANNA includes more detailed testing functionality in ANNA_test.py.

To use ANNA_test.py, you need two things: a trained model, and some data to test. ANNA_test.py is controlled through the same ANNA.param file as used for training. You may use a model you have trained locally, or you can transport models between computers (see Section 4.6). The data should be of the same binary form as used for training - if you used a cross-validation set during training, you may consider using the same data for testing.

The relevant parameters for testing using ANNA are contained under TESTING PARAMETERS, with the exception of the parameter names, minimum values, and maximum values used during normalization. These are still the OUTPUT_NAMES, MIN_PARAMS, and MAX_PARAMS fields from the TRAINING PARAMETERS section. It is important that you use the same minimum and maximum parameter values when testing as were used during training, or you will get some very surprising (and unwelcome) results during testing.

You can opt to preprocess your test data using random continuum and S/N solutions using the REL_CONT_E_TEST and SN_RANGE_TEST parameters, respectively. Of course, if you are testing the network with real data, these can be turned off using PREPROCESS_TEST. As during training, if you are applying synthetic noise to your test spectra, you must also supply a S/N profile using SN_TEMPLATE_TEST.

Once you have set the test parameters, you can run ANNA_test.py. This will be much, much faster than training, since there are no gradients to calculate at this point. Once ANNA has passed all the spectra through the trained neural network, it will compare the output parameters it has inferred from these spectra to the real values supplied in the input binary file. It then prints a summary of these results to the terminal (or console) and writes a file, test_stats.out, to TEST_SAVE_LOC. This comma-separated file contains on each line the supplied star ID, followed by the supplied "true" parameters, followed by several columns containing the results of $(P_{true} - P_{inferred})$. These data can then be used to judge ANNA's performance, and look for any systematics.

It is very important to make sure your test data looks like the training data. Once you reach the testing stage, ANNA has already been "taught" what spectra look like - so if testing (or inference) spectra look different, then the network will likely return very innaccurate parameters. If you have trained ANNA using synthetic spectra, it is a good idea to use ANNA_test.py to verify that the program is capable of accurately handling a small sample of real spectra you have parameterized (using an independent method). Ideally, ANNA will be able to parameterize these test spectra with accuracies similar to those achieved during training. If ANNA's accuracy is severely degraded when you test it on real spectra, then that's a clear indicator that your training data is not representative of your real data, either because of bad models, incorrect S/N profile or resolution, etc.

4.3.1 Residual Trends

In a perfect world, the neural network at the core of ANNA could be made such that it perfectly encodes the relations between input pixel values and correct output parameters. Sometimes this can be done; often it cannot. The latter case could be because of computational limitations (a network complex enough to translate between inputs and outputs is too complex to fully train in a reasonable amount of time) or insufficient input signal (not enough pixels/features to be able to accurately parameterize spectra across the entire training parameter space). In these situations it is likely there will be trends visible in the residuals output by ANNA_test.py.

If there are trends in these residuals, they can be removed by fitting functions (usually low-order polynomials) of the form $\mathrm{TrueValue} = f(\mathrm{InferredValue})$ to each parameter ANNA is being trained to recognize. For example, if there is a visible trend in true surface temperature as a function of inferred surface temperature (as given by ANNA_train.py, a relation of the form $\mathrm{Temp}_{\mathrm{True}} = f(\mathrm{Temp}_{\mathrm{Infer}})$ can be fit to the test results. This relation can then be applied to the temperatures inferred by `ANNA_infer.py` in order to correct for systematics introduced during training.

4.4 Inference

Once a model has been trained with `ANNA_train.py` and its generalizability has been verified with `ANNA_test.py`, it is ready to be used for inference using `ANNA_infer.py`. There are three parameters that must always be set when running inference. First, you must specify where the model to use is saved using `INFER_MODEL_LOC`. Next, you must specify where to save the outputs from inference using `INFER_SAVE_LOC`. Finally, you must provide a sample training spectrum using `WAVE_TEMPLATE`.

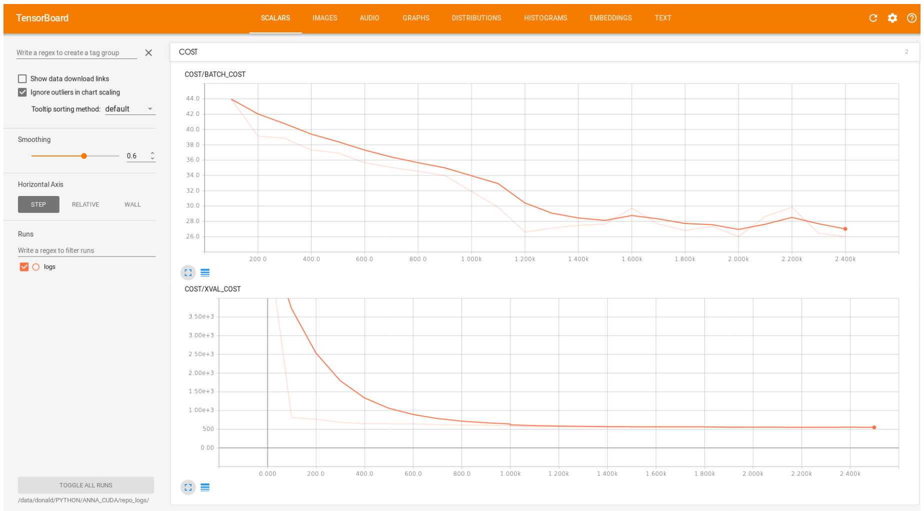
The sample spectrum is used as a template to define the training wavelength grid, onto which each inference spectrum will be interpolated before being fed through the neural network. The reason this is necessary is because the model was trained using a certain wavelength grid; your inference data will likely not be on the same grid. The wavelength template should be supplied as a text file with two whitespace-delimited columns - one for wavelength (in angstroms) and one for flux values (unused in this case).

Once you have specified these parameters, all that remains is to indicate what kind of data format you will be providing ANNA for inference. There are three main read-in modes. First is a debugging mode. If `DEBUG_MODE` is set to YES, then you should point `DEBUG_DATA` to a binary file of the same form as used for training and testing. ANNA will read this file in, discarding the known parameter values. In this mode, it is not necessary to supply a wavelength template, as ANNA will just fetch the wavelengths from the binary file while making the assumption that they are the same as the training wavelengths. This debugging mode is provided in case you are transferring models between computers and want to verify that everything went okay.

The second mode is for standard multispec fits file read-in. Currently, this mode only allows for a single multispec file to be read in at once, and every line in the file will be read in. If you wish to use this mode, set `SINGLE_MS_MODE` to YES and point `MS_FITS_IMAGE` to your multispec fits file. When you run `ANNA_infer.py`, the program will use the `readmultispec.py` script to read in your fits file.

standard fits image read-in, etc etc.

After feeding the input data through the trained model, ANNA will print the inferred parameters to a comma-separated file, `infer_stats.out`, located in `INFER_SAVE_LOC`. This file contains the results for one star in each row, listing the ID first, then listing the inferred parameters. If running in debug mode, the star IDs will be whatever was specified in the binary file. If using multispec fits read-in, then star IDs will be the corresponding row of the multispec file.



□ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default

Smoothing 0.6

☑ ○ logs

/data/donald/PYTHON/ANNA_CUDA/repo_logs/

Figure 4: Cost graphs in TensorBoard

4.5 Monitoring and Visualization Tools

TensorFlow includes a useful built-in utility called TensorBoard to visualize a neural network's architecture and graphically monitor performance during training. TensorBoard is useful for quickly making sure that training is going smoothly, and can also help in the process of parameter tuning. ANNA provides the option to use TensorBoard, allowing the user to visually track the network cost during training.

To use TensorBoard during training, you need to set the `TBOARD_OUTPUT` parameter to YES. You must also specify a location to save the TensorBoard logs using `LOG_LOC`. You can then proceed with training as detailed in Section 4.2. Once training is started, you can start a TensorBoard session, which will automatically read the most recent log being written to, using the terminal command:

```
> tensorboard --logdir=/path/to/logdir/
```

This will start a TensorBoard session. To actually see the session, you need to open a browser window and go to the local address:

```
http://0.0.0.0:6006
```

This will open the TensorBoard screen in your browser. There should be a series of tabs running along the top bar of the screen; if not already selected, you can click on “Scalars” to bring up the cost graphs. You may have to click on “COST” in order to actually display the graphs. If you are using a

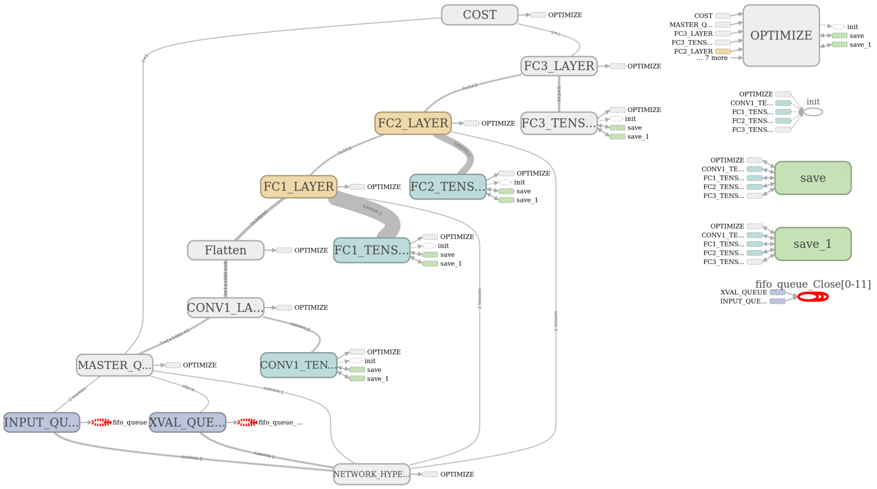


Figure 5: ANNA structure as visualized in TensorBoard

cross-validation set during training, then both the current batch cost and cross-validation cost will be displayed as a function of the total number of iterations (sampled every SAMPLE_INTERVAL) and refreshed every minute or so. Thus, you can track the accuracy of the network as it is trained, as shown in Figure 4.

You can also access other visualizations in the Tensorboard session. Clicking on the "GRAPHS" tab will bring up a visualization of ANNA's layout (Figure 5). Selecting "DISTRIBUTIONS" or "HISTOGRAMS" will allow you to see how the network's weights are changing during the training process (Figure 6).

All of the accessible graphs can be customized and are interactive. For example, hovering your mouse cursor over the cost graph will provide a popup giving more information about the sampling interval being hovered over. You can expand the various windows in the "GRAPHS" tab to see the subfunctions in ANNA and various algorithms used. For full details on how to use TensorBoard, you can read the documentation at `www.tensorflow.org/get_started/summaries_and_tensorboard`.

There is one additional graphical utility included with ANNA - the timeline output. This allows you to profile code performance, determining how the various tasks in ANNA use up compute time. This output is controlled using the TIMELINE_OUTPUT parameter. If set to YES, then for the most recent completed, sampled iteration, ANNA will write a `timeline_01.json` file to the LOG_LOC. This file can then be opened in the Google Chrome browser by typing:

```
> chrome://tracing
```

into the search bar. This will open the tracing screen in Chrome, where you can

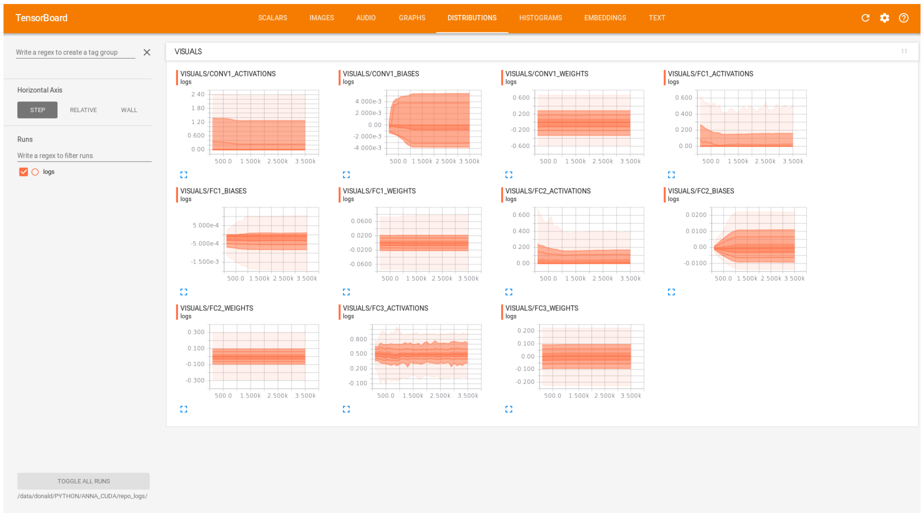


Figure 6: Distribution of weights as shown in Tensorboard

load in the `timeline_01.json` file and see where hardware resources are being devoted. This functionality is useful f you are using a GPU and wish to identify any bottlenecks in ANNA's performance. *note: as of 7/6/17, timeline output is broken in ANNA. there's an open issue on github for it.*

4.6 Transferring Models

A Step-by-Step Example

This section covers an example of how to use ANNA, starting with training and ending with inference. To follow this tutorial, you will need to set up ANNA as detailed in Section 3. You will also need to download the sample binary data file `allspec_test_200_600` from the GitHub repository, the S/N template `sn.robo.template`, the `jun2011_3133.fits` file we'll use for inference, and the wavelength template file, `wavelength.template`.

The data file contains 5000 synthetic spectra and is approximately 40 MB in size. The spectra have temperatures, gravities, [Fe/H], microturbulent velocities, and rotational velocities specified, with values between 4000.0 K, 2.7 dex, -0.5 dex, 0.0 km/s, and 0.0 km/s, and 7000.0 K, 4.8 dex, +0.2 dex, 3.0 km/s, and 40.0 km/s, respectively. They have random radial velocities between -100.0 km/s and 100.0 km/s, stored in the dummy entry for each star. In general with such a wide range of parameters it would be advisable to have a much larger training set - maybe hundreds of thousands of spectra - but to keep the file size small there are only a few thousand.

The training spectra cover the wavelength range $6625.0 < \lambda < 6825.0$ angstroms, and the pixel resolution is 0.2 angstroms/px. The actual spectral resolution is 0.6 angstroms. Thus, there are 1001 pixels in each spectrum.

With our training data in hand, we can define the neural network architecture and prepare to train. NUM_PX is set to 1001. We set the kernel width (in CONV1_SHAPE) to 45 pixels, which gives enough wavelength coverage for each kernel to encompass typical line complexes in the wavelength range we're working with. Since we only have 5000 training spectra and there are relatively few pixels per spectrum, we'll set the number of kernels to 10, and the dimensions of the fully-connected layers (FC1_OUTDIM and FC2_OUTDIM) to be 128. This is on the small side for a neural network, but will work for the purposes of this example. The number of outputs is set to 5, as there are five parameters supplied with each spectrum (not counting radial velocity, which is stored in the dummy entry).

Next, we need to determine the learning hyperparameters. We'll elect to take advantage of two-stage training here, using dropout in the first stage to reduce overfitting. Thus, we first flip DO_TRAIN2 to YES. We set LEARN_RATE1 to 5e-5, which is relatively fast, and use a smaller learning rate of 1e-6 for LEARN_RATE2 to better converge the network weights. For dropout, we want it during the first stage of training, and set KEEP_PROB1 to 0.8, but for the second stage we'll turn it off, so we set KEEP_PROB2 to 1.0.

The first stage of training is the more important one, since it is where the weights go from being random to more or less correct. Thus, we set NUM_TRAIN_ITERS1 to 13000, and set NUM_TRAIN_ITERS2 to 2000, for 15000 total iterations. When training outside of this example, the number of iterations should be much higher - a hundred thousand or so. Batch size is kept constant during both stages of training, at 100 examples/batch.

The training spectra have infinite S/N and perfect continuum placement, so using them as-is is not a great idea, since the network will just overfit and generalize poorly. We take advantage of ANNA's ability to preprocess examples as they are read in to a batch, setting PREPROCESS_TRAIN to YES. Small random continuum placement errors at the few percent level are added by setting REL_CONT_E_TRAIN to -0.010, 0.010. S/N values are set to be between 175.0 and 250.0, with the SN_TEMPLATE parameter pointed towards the sn.robo.template file. NUM_FETCH is set to 1000. The TRAINING_DATA parameter is pointed to the allspec_test_200_600 file, and SAVE_LOC is set to an empty directory.

We still need to set the OUTPUT_NAMES, MIN_PARAMS, and MAX_PARAMS fields. OUTPUT_NAMES is set to temperature, gravity, metallicity, microturb, and rotvel, corresponding to the names and order of the output parameters for each spectrum. The max/min parameters are set to 4000.0, 2.7, -0.5, 0.0, 0.0 and 7000.0, 4.8, 0.2, 3.0, 40.0, respectively, following the minimum and maximum parameter values and their order present in the training data. Technically there are no stars with temperatures exactly 4000.0 K in the training set, but the ranges we set are pretty close to the max/min parameters.

We have relatively few training iterations and we eventually want to visualize the training process in TensorBoard, so SAMPLE_INTERVAL is set to the relatively dense sampling of 50. PP_THREADS is left at 1 and MAX_PP_DEPTH is set to 200000. We also specify a LOG_LOC in an empty directory different than the save directory.

As noted, we set TBOARD_OUTPUT to YES, but since we aren't interested in profiling the code we set TIMELINE_OUTPUT to NO.

It would be nice to take advantage of the cross-validation feature during training, but we don't have separate cross-validation dataset. However, we set TRAINING_XVAL to YES anyway and point XVAL_DATA to the training data file. During normal use of ANNA, this is discouraged, but it will have to do for this example. There's no need to use the entire training data set for cross-validation, so we set XVAL_SIZE to 500. This means that only the first 500 stars in the training set are used for cross-validation. PREPROCESS_XVAL is set to YES since the "cross-validation" data is just the perfect training data. XV_THREADS is left at 1.

Since we aren't training for many iterations, early stopping likely won't be useful, but as an exercise, we set EARLY_STOP to YES, and set ES_SAMPLE to 100. This means if the cost hasn't decreased within 2000 iterations (SAMPLE_INTERVAL × ES_SAMPLE) then ANNA will stop training and save the model in its current state. Note that there's no way the second stage of training will stop early here, since we only specified 500 iterations.

At this point we're ready to train. We fire up a terminal, and run ANNA_train.py. Once ANNA is training (prints the first batch's cost to the terminal screen), we can start running TensorBoard using the log file that is being written to as the network trains. After launching a TensorBoard session in a separate terminal window using the `> tensorboard --logdir=/path/to/logdir/` command, we open a new web browser window and navigate to http://0.0.0.0:6006. From there, we can monitor the cost as it is updated.

Now all that remains to do while ANNA is training is to grab a cup of coffee and watch the cost graphs in TensorBoard. Once training is done we can close the Tensoboard browser and terminal windows, since we won't be needing them anymore. The training process takes around 15-30 minutes, depending hardware.

At the end of training, the current batch cost and cross-validation cost should be around 65.0 and 315.0, respectively. Note that the cross-validation set is five times larger than a training batch so the cost is correspondingly higher. These cost values will change somewhat run-to-run, since the weights are randomly initialized every time training is started. It should also be pretty apparent from the cost graphs in TensorBoard that the model isn't fully trained; the cross-validation cost hasn't bottomed out yet. So we could re-run training and increase the total number of iterations - this is recommended if another coffee break is needed.

There should now be a frozen.model file lurking in the directory specified as the save location. This is the trained model, and contains both the network architecture and weight values. If we were going to, say, send the model to a collaborator to run inference, all we would need to do is email the frozen model, which could then be used with ANNA by the collaborator.

Instead, we'll test and run inference ourselves. As with the cross-validation set, we don't have a separate test set, so we're forced to set TESTING_DATA to the training data. In real usage, this is highly discouraged, since the testing isn't independent at all if the training data is reused, since ANNA has already seen and learned from it. We point `TEST_MODEL_LOC` at the directory containing the frozen model, and preprocess the testing data in the same way as we did the training data, providing the same S/N template. We also set `TEST_SAVE_LOC` to a convenient directory, since we'll be navigating there in a few seconds.

We now run `ANNA_test.py` from the terminal, which takes very little time. When testing is done, ANNA prints some summary statistics to the terminal window. We see that the parameter accuracy isn't great - but this is to be expected. Even though we used the same data to test as we did to train, we didn't train for very many iterations, and we didn't show the network very many spectra during training, so the weights aren't fully trained. Thus, the network isn't able to parameterize spectra very well right now. If we had more training examples and ran for more iterations, the parameter errors would decrease.

For a more detailed look at the test results, we can navigate to `TEST_MODEL_LOC` and examine the comma-separated value file that was written by ANNA. Diving into it, we can see that especially for hotter stars, the parameter values are poorly inferred by ANNA. The spectral region we trained the network on has very few lines as atmospheric temperature increases, so there are fewer features for ANNA to use when classifying these stars. It's a good idea to do these sorts of checks during real usage of ANNA.

We know that our model isn't great, since we didn't train it very well, and we see that there are some systematics in the results from `ANNA_test.py`, but in the interest of science, we'll continue on anyway to inference. Our inference data, `jun2011_3133.fits`, is in multispec format, so we'll use ANNA's multispec reading ability. This particular test data consists of 90 daytime solar spectra collected with the Hydra spectrograph on the WIYN 3.5m telescope at Kitt Peak National Observatory in June 2011 [2]. By complete coincidence, the training data and S/N template were chosen to mimic these spectra.

We don't need debug mode, so we set `DEBUG_MODE` to NO and ignore the `DEBUG_DATA` parameter. Instead, we set `SINGLE_MS_MODE` to YES and supply the `jun2011_3133.fits` file to the `MS_FITS_IMAGE` parameter. `WAVE_TEMPLATE` should point to the `wavelength.template` file. `INFER_MODEL_LOC` should point to our saved model, and `INFER_SAVE_LOC` should be a convenient directory.

Once we set those parameters, we can run `ANNA_infer.py`. It should complete the parameterization very quickly and write the results to `infer_stats.out`, located in the specified save location.

The results should look relatively promising. These are all spectra of the same object, so we would expect relatively little variance in parameters when considering different lines of input multispec file. Indeed, the standard deviations in temperature, gravity, [Fe/H], microturbulent velocity, and rotational velocity should be around 60 K, 0.025 dex, 0.015 dex, 0.1 km/s, and 1 km/s,

[2] The WIYN Observatory is a joint facility of the University of Wisconsin-Madison, Indiana University, the National Optical Astronomy Observatory and the University of Missouri. Kitt Peak National Observatory is a facility of the National Optical Astronomy Observatory, which is operated by the Association of Universities for Research in Astronomy (AURA) under a cooperative agreement with the National Science Foundation

respectively. Furthermore, the parameters are relatively accurate - respective values should be around 5700 K, 3.8 dex, -0.1 dex, 1.8 km/s, and 8 km/s. Compare these with standard values of 5770 K, 4.40 dex, 0.0 dex, 1.14 km/s, and a few km/s. Considering that we only trained ANNA using several thousand spectra and that the training likely hadn't completed, these results aren't too bad! It looks like gravity and microturbulent velocity are anomalously bad, but the wavelength region we are using doesn't contain much information that would constrain these parameters, so it is reasonable to expect that ANNA would have the hardest time converging on accurate solutions for these.

This concludes the end-to-end tutorial of how to use ANNA. If you find that your results don't look anything like those listed here, you might try training the network again - because the network weights are randomly initialized, it can take longer to converge for some initializations than others. During general usage of ANNA, this problem can be avoided by training (using early stopping) for many more iterations than we have used here. If aspects of this tutorial are unclear, you can find more complete documentation of ANNA in the main body of this manual.

B Troubleshooting

Some common issues are listed below. Most of the time, if ANNA doesn't seem to work, there's a problem with one of the neural network hyperparameters. However, the software undoubtedly contains several undiscovered bugs, so if you believe you have found one, please contact me through GitHub or by email.

1. **The network doesn't learn!**
   This is usually a hyperparameter issue. Fortunately, exactly *which* hyperparameter is responsible for the failure of the network to learn can be diagnosed by looking at how the cost during training behaves and/or looking at the inferred outputs from `ANNA_test.py`.

   (a) **The cost never decreases or decreases very little, and outputs all seem random**
   The network weights aren't changing appropriately from their initial values.

   (b) **The cost decreases at first, but then stops decreasing, and every spectra gives the same output parameters when I run `ANNA_test.py`.**
   Generally this means that large numbers of the nodes in the network are "dying" - moving to such large negative numbers that they will always give a response of 0 when data is fed through.

   The other alternative is that the output parameters you are trying to get ANNA to learn do not map to features in the input spectra. For example, if you want to train ANNA to determine carbon abundances but your spectra have no carbon lines, then ANNA will never learn how to map your spectra to accurate carbon abundances.

2. **The network learns, but it is not very good at parameterizing spectra during inference.**
   If ANNA generalizes poorly, make sure that your training spectra are representative of your real spectra. This means your SN profile, average/peak SN, resolution, and parameter values should approximately match between your training and inference data. If you trained using synthetic spectra, your models must accurately reproduce your real spectra (an easy way to diagnose a model issue is to generate a synthetic spectrum and compare it with a well-studied spectrum taken under the same instrumental conditions as your inference data, e.g., the solar spectrum). If you are sure everything matches up, take note of the number of absorption lines in your inference spectra. Hot stars or certain wavelength ranges have very few spectral features; ANNA is unlikely to be able to accurately parameterize such spectra since it doesn't have much to work with.

3. **Training takes forever!**
   If the network cost decreases but training takes a very long time, there are several areas to look to for speedups.

   (a) **Try a larger learning rate.**
   Ideally, the learning rate should be as large as possible while still maintaining network stability (see issue above).

   (b) **Decrease network complexity**
   Note that increasing the dimensions of a layer of the network (e.g., `FC1_OUTDIM`) actually increases the size of that layer by $O(n^2)$ since you are really increasing the length of one side of a 2D matrix. So check these values and see if turning them down speeds up training without decreasing network accuracy.

   (c) **Decrease batch size**
   Very large batch sizes can slow down training significantly, since the entire batch worth of spectra must be fed through the network, and then many, many gradients must be calculated and weights adjusted. The batch size should be small enough that each iteration (i.e. each update of the network weights) takes relatively little time, but large enough that large fluctuations in the network cost due to batches being extremely non-representative of the overall training data are avoided.

   (d) **Decrease the sampling frequency.**
   The `SAMPLE_INTERVAL` parameter controls how frequently ANNA runs diagnostics while training. If this is a very small number, diagnostics will be run very frequently, which can slow down training significantly.

(e) **Compile TensorFlow from source.**
TensorFlow supports many vectorized instructions enabled on modern CPUs (e.g. SSE4), but in the interest of widespread compatiblity, these instructions are generally not enabled on the default TensorFlow version and are only available if TensorFlow is compiled from source. However, enabling these instructions can provide sizeable speedups, particularly when ANNA is trained using a CPU. Instructions for compiling TensorFlow from source can be found online.

(f) **If you have GPU, ensure CUDA is installed and the GPU-enabled version of TensorFlow is installed**
Newer Nvidia GPUs, even modest ones, can speed up training by orders of magnitude. However, the gpu-enabled version of TensorFlow must be correctly installed and CUDA must also be installed on the computer used for training. Instructions for doing this can be found online.

(g) **Be patient.** ANNA was explicitly designed to run on modest computer hardware. However, training a neural network is computationally intensive. CPU-only training can take several days to complete. If none of the above work for you, you may just need to be patient!

4. **ANNA gives me the warning "looks like input array contains extra entries!" and won't train.**
   This happens when the specified number of pixels in the input binary file results in a non-integer number of stars being read in from the binary file. First, ensure the binary file you are reading in was written in 64-bit floating point (double precision); numpy by default stores 64-bit floating point. Second, ensure that each star consists of n+7 entries, where n is the number of pixels in each spectrum.

5. **I always run out of RAM and the program crashes when I try to train the network.**
   ANNA in its current state reads in the entire binary file used for training to RAM when initializing. This choice was made because having everything in RAM considerably speeds up training. That being said, large training sets or training sets consisting of high-resolution spectra covering large wavelength ranges may result in out-of-memory errors. For now, the only workaround is to reduce the size of the training set. Planned updates to ANNA should eliminate this issue by making the program more memory-frugal.

C Glossary of Important Files

1. `ANNA_train.py`. The main ANNA training function. Also referenced in both `ANNA_test.py` and `ANNA_infer.py`, so even when the network is not being trained, this file must be present.

2. `ANNA_test.py`. Tests a trained ANNA model. The user supplies data to be tested, and ANNA returns a comparison of the provided “true” parameters and the ones it has inferred.

3. `ANNA_infer.py`. Allows inference to be run on data where the output parameters are unknown. Includes several functionalities to allow the read-in of FITS images.

4. `convnet2.py`. Wrapper library used by the main ANNA training, testing, and inference files.

5. `ANNA.param`. The ANNA parameter file. Contains fields that control all aspects of ANNA operation.

6. `readmultispec.py`. A slightly modified version of Kevin Gullikson’s `readmultispec.py` script, which was itself originally based on an IDL script written by Rick White. Full citation in Sec 3.

References

Allende Prieto, C., Rebolo, R., García López, R. J., et al. 2000, The Astronomical Journal, 120, 1516

Bailer-Jones, C. A. L., Gupta, R., & Singh, H. P. 2002, in Automated Data Analysis in Astronomy, ed. R. Gupta, H. P. Singh, & C. A. L. Bailer-Jones, 51

Bailer-Jones, C. A. L., Irwin, M., Gilmore, G., & von Hippel, T. 1997, Monthly Notices of the Royal Astronomical Society, 292, 157

Bishop, C. M. 1995, Neural Networks for Pattern Recognition (New York, NY, USA: Oxford University Press, Inc.)

Dafonte, C., Fustes, D., Manteiga, M., et al. 2016, Astronomy & Astrophysics, 594, A68

Haykin, S. 1998, Neural Networks: A Comprehensive Foundation, 2nd edn. (Upper Saddle River, NJ, USA: Prentice Hall PTR)

He, K., Zhang, X., Ren, S., & Sun, J. 2015, in The IEEE International Conference on Computer Vision (ICCV)

Kingma, D. P., & Ba, J. 2014, in Proceedings of the 3rd International Conference on Learning Representations (ICLR)

Li, X.-R., Pan, R.-Y., & Duan, F.-Q. 2017, Research in Astronomy and Astrophysics, 17, 036. http://stacks.iop.org/1674-4527/17/i=4/a=036

Manteiga, M., Ordóñez, D., Dafonte, C., & Arcay, B. 2010, Publications of the Astronomical Society of the Pacific, 122, 608

Nesterov, Y. 1983in , 372–376

Recio-Blanco, A., de Laverny, P., Allende Prieto, C., et al. 2016, Astronomy & Astrophysics, 585, A93

Ruder, S. 2016, ArXiv e-prints, arXiv:1609.04747

Simonyan, K., & Zisserman, A. 2014, arXiv preprint arXiv:1409.1556

Snider, S., Allende Prieto, C., von Hippel, T., et al. 2001, The Astrophysical Journal, 562, 528

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. 2014, Journal of machine learning research, 15, 1929