DGTools for Training Custom Models

How I trained a model on my summer vacation

18 FEB 2021

Quick Overview of the Steps to Create a Model

Before describing the details of the training process, we provide here a quick summary of the steps involved to go from a set of data that needs to be labeled through a custom model deployed for inference in Deepgram's inference engine.

1. The first step is **loading audio files into a Hotpepper training set**. This is done by copying the files into a directory ~/dashscript/datasets/<your-data-set-name> and then creating the data set within Hotpepper (see the Hotpepper documentation).
2. **Label the audio files in Hotpepper** -- these must be marked as complete through 4 levels of labeling (L1-4). A level of transcription is considered "Done" when the "Done" button is clicked during transcription. If the transcript at a given level is perfect and doesn't require any more changes, the later levels can simply be marked as "Done" without changes.
3. **Output a training set package from Hotpepper.** See the Section "Training with Data from Hotpepper" for the details on how to do this.
4. **Run training**. See the remainder of this document on step-by-step instructions for training custom models.
5. **Deploy your model** so that it can be used with the API. See the Section "Deploying a model" for the details on how to do this.

Introduction

DGTools provides a suite of tools which enable training custom speech recognition models in a variety of languages. A speech recognition model is used by our on premise inference engine to convert audio into text for a given language. In order to provide the most accurate Automatic Speech Recognition (ASR) possible, we need to train a model using example audio and text. This enables the model to 'learn' important information like what background noises to ignore and what common words and phrases are likely to occur.

Training generally follows three steps:

1) Training data creation - This step takes customer audio and truth text and turns it into 'trainable' data. We generally refer to this as 'chunking' because it takes the audio and text and turns it into short utterances or 'chunks' that are suitable for training.

2) Model training - Model training takes the trainable data and feeds it to the model. At a high level, the model ingests the audio, makes a prediction of what it thinks is in the audio and then the training code uses the truth text provided to correct the model. In order to get the best training, just like a human, we feed the training data to the model several times in what we call ‘epochs’ of training. An important aspect of training is how many epochs you train a model for. DGTools provides sensible defaults to start with and this document will provide recommendations for more advanced cases.
3) Model selection and packaging - The final step of training is to package a model for deployment. During training many secondary files are created to aid in the training process. Packaging, which is a single simple command, removes these extra files and packages the model into a single deployable that can then be used in the production server.

These are the basics, but use-cases often change based on exact requirements. This document will cover how to configure the training environment including how to make data available for training, how to do simple one-time training off of a single dataset and finally it will cover some more advanced training scenarios and considerations of when you may want to use them.

Step one getting dgtools and creating a training directory

Some customers may have custom installations of our onprem tools. In those cases this initial setup step should not be needed.

DGTools uses docker and the same on premise environment outlined in our configuration docs. Familiarity with docker is recommended, but not required past the installation. After following the configuration steps it is easy to then grab the latest dgtools by executing

```
docker pull deepgram/onprem-dgtools:latest
```

This command will retrieve the latest image of dgtools and make it available for use. Re-running this command is all that is required to upgrade to the latest version as updates to dgtools become available.

After retrieving the docker image you need to determine where your training directory should be. This directory will be used for all future DGtools training and will need to be in a location that has sufficient available space to hold all training data as well as additional room for models. A rule of thumb is 3x the expected size of the raw data plus 10GB for models minimum however if your training scenario uses highly compressed audio or you anticipate making many different models it is recommended to have more space available.

Once you have selected a good place for your training directory, create it! This is usually as simple as:

```
mkdir -p /path/to/training/dir
```

A consideration at this stage is what user and group permissions should be assigned to this directory. Setting up permissions is out of scope for this document, but in general it is a good idea to have a training user and group and make sure that your training directory is owned by that user.

Once the training directory is created we need to create the dgtools command in it. This is just a simple script that you will preface all future calls with. It isn't strictly needed, but it helps remove confusing docker commands and lets you focus on just running dgtools. To make this command execute the following:

```
echo "nvidia-docker run -t -i --volume $TRAININGDIR:/big/dgtraining
--user \$(id -u):\$(id -g) --rm deepgram/onprem-dgtools:$TAG \"\$@\""
> $TRAININGDIR/dgtools
```

Replace $TRAININGDIR with the full path to the training directory you created. After that, run:

```
chmod +x $TRAININGDIR/dgtools
```

This will make the new script executable so that you can run it as if it were a program. As mentioned before, this is a convenience script. If you are familiar with docker you will understand that there are options for how to execute commands that could give you more flexibility. This particular command is designed to execute all commands as the current user. As was mentioned before, it is recommended to have a training account. If you decide to create that user then an alternative version of this command would be:

```
echo "nvidia-docker run -t -i --volume $TRAININGDIR:/big/dgtraining
--user \$(id -u $USERNAME):\$(id -g $GROUP) --rm
deepgram/onprem-dgtools:$TAG \"\$@\"" > $TRAININGDIR/dgtools
```

Where $USERNAME and $GROUP should be replaced with the user and group created for training. It is important to set these variables correctly since DGtools will create all new files as this user and will execute with the permissions of this user. It is highly recommended that you do not run dgtools as sudo since, beyond the security implications, dgtools will create all files as root which will make them inaccessible to most accounts.

The final step in the initial configuration is to run dgconfigure. This will create a directory structure that will be used throughout training and it will also test that the dgtools script you just created is working correctly. To do this run:

```
<training_dir>/dgtools dgconfigure
```

Replace <training_dir> with the full path to your training directory. Strictly speaking, you do not need to run this command from the training directory. Absolute and relative path names are not used with dgtools commands, only references to model names and dataset names. Because of how docker works, and specifically how it re-maps paths, this command can now be run from anywhere and could be put in a directory in your path to make it simple to call. However, since all results will appear in the training directory, it is useful to do most of the commands from there so you can see the results easily.

At this point you should have a working installation of DGtools ready to be used!

Creating a training dataset

DGTools uses the concept of a ‘dataset’. ‘Dataset’ here refers to a collection of audio and truth text files that will be used as training data for the speech recognition model. DGTools is flexible in what data it can use as training data and how you organize that training data. Its biggest restriction is where you store it. For DGtools to find data it must be in a sub-directory of <training_dir>/rawdata. Any subdirectory of rawdata will be considered a potential dataset by dgtools. So creating a dataset is easy, just make a directory in rawdata with whatever name you want and add all of your audio and text transcripts into it. Only top level directory names matter so you can organize your training data in subdirectories however you want. DGTools will search all subdirectories to find files with the same name, but different extensions, and match them to create trainable data. These files must be named and stored in one of the following two forms:

```
STRUCTURE 1:
+ <training_dir>                (by default this is ~/training)
+- rawdata
+-- custom-model-data-directory-1
+--- audio
+---- audio-file-name-1.wav
+---- audio-file-name-2.wav
+---- ...
+--- transcripts
+---- audio-file-name-1.txt     (this must match the audio file name)
+---- audio-file-name-2.txt
+---- ...
+-- custom-model-data-directory-2
+--- audio
+---- audio-file-name-a.wav          (names should be unique)
+---- audio-file-name-b.wav
+---- ...
+--- transcripts
+---- audio-file-name-a.txt
+---- audio-file-name-b.txt
+---- ...
```

OR

```
STRUCTURE 2: (this is structure output by Hotpepper)
+ <training_dir>
+- rawdata
+-- custom-model-name
+--- audio-file-name-1
```

```
+---- audio.wav
+---- transcription.txt
+--- audio-file-name-2
+---- audio.wav
+---- transcription.txt
+--- audio-file-name-3 ...
```

Dgtools would find the associated audio and transcript files by either matching their name (STRUCTURE 1) or reading them from the audio file name directory (STRUCTURE 2). In the case of STRUCTURE 1, if two transcript files are named identically (in two different directories) it will cause a warning to appear and dgtools will use the last one it finds. The filename is used as a unique source id and as such should not be re-used by several different training files.

DGTools has no limit on the number of files in a training directory, but there is a practical minimum. In general it is a bad idea to train with fewer than 20 files. This is because DGtools ‘splits’ the files into two groups, train and validate. Validate files are not trained on and are used to evaluate how the model is learning. This split is random and the amount of files used for validation is determined by the total number of files available. As more files are added, a larger percentage is dedicated to training. This means that when there are just a few files, many of them will be dedicated to validation which is why a practical minimum of 20 is recommended. Fewer than that and you run into the possibility that the random split between train and validate will create an empty set on either train or validate and the model will not be able to train.

DGTools can manage any number of datasets. When it trains a model you tell it what dataset(s) to use in training. For the simple training cases the name of the dataset will be the name of the training model (but not the model name for deployment).

Training with Data from Hotpepper

Hotpepper provides a convenient way to label audio data. The instructions for creating datasets and labelling is provided in the Hotpepper documentation. Here we describe how to package Hotpepper data and use for training.

Prerequisite: Before you can train with data from Hotpepper it must go through all 4 stages of transcription (see the Hotpepper documentation). Once a transcript has been reviewed and marked “Done” at Level 4 (i.e. L4) it is available for packaging and training.

To package the data from a data set that has been L4 reviewed:

1. Login to Hotpepper as administrator
2. Select the black “Admin” button
3. Select the “Datasets” button from the Admin dashboard

4. From the Datasets Overview page, find the data set(s) you wish to package and use for training. Click the light blue “Package” button to the right of the data set listing for each dataset you wish to use for training.

The remaining steps require you to open a terminal. First, navigate to the following directory:

```
cd ~/dashscript/packaged-datasets
```

If you list the files (ls) in this directory, you will see files of the form:

```
ls
<model-name>.tar.gz
```

These files contain your training data. The remaining steps are to copy (or move) this file to the desired rawdata directory where you are storing the training data for planned training runs.

At the command line type the following (to copy):

```
cp <model-name>.tar.gz ~/training/rawdata
```

Note: you can create your training directory anywhere you like. Here we’ve used the default location of ~/training.

Now, extract the training files using the following commands:

```
cd ~/training/rawdata
tar xzvf <model-name>.tar.gz
```

After this, you can optionally remove the tar.gz file as it is not required for training. At this point, you can begin the training process described below.

The basics, finally - dgautotrain

For many use cases a customer will want to simply train a single model based off of a single dataset. For this simple use case it is easiest to use the built in dgautotrain feature. After creating a training dataset just run:

```
cd ~/training   (or any location in which training data is stored)
dgtools dgautotrain <dataset_name>
```

Then walk away. For a while. Depending on the amount of data it can take just a few minutes or it can take, well, days. With 100 hours of audio it is not unreasonable to take 3-4 hours to train a model. As the amount of data increases, the time it takes to train will increase. DGAutotrain goes through a set of best practices when training. The rough steps are:

1. **Do an initial ‘chunking’ to the training data** - This makes trainable ‘chunks’ of audio and text called utterances from the supplied training data. These chunks will be used to train the model.
2. **Train a utility model to help improve training data chunking** - Our chunker relies on a utility model to help it make reasonable cuts on the supplied training data and to match those cuts in the audio with what truth text is in them. Just like the custom model we are training, this utility model gets benefit from training and makes better cuts after it has been trained so we use the initial training cuts made to ‘bootstrap’ this utility model.
3. **Re-chunk the data using the utility model** - Now that we have a trained utility model, we can make better training cuts so that the custom model has the best possible training data to work from.
4. **Generate a custom model derived from a base model** - DGTools ships with an english ‘base model’ that is used to derive a custom model from. This base model is training on a wide variety of data so that it is generally good at many different types of audio. We use this base model as a starting point for new custom models. In more advanced training scenarios other base models can be used as a starting point including base models for other languages. That will be covered later in this document.
5. **Train the new custom model** - All the prep work is now used to train the actual custom speech model.
6. **Export the new custom model** - The final step will export a deployable version of the model to:

```
<training_dir>/deployablemodels/<model_name>.tar
```

Each of these steps can be done individually with more advanced commands, but dgautotrain provides a convenient starting point.

A little less basic - dgautotrain a different language

By default DGtools only ships with English and Turkish base models to train against. DGTools however supports training other languages. In order to enable this you will need to copy the files into the <training_dir>/deployablemodels directory. The first, general.<lang_code>.tar, is the 'base model' that custom models will be derived from. The second, phonemes.<lang_code>.tar is a utility model that will be used in training data preparation. Once those files are copied in, the steps for training are nearly identical to the simple case. First, create a dataset next run dgautotrain. The only difference is that the dgautotrain command will need to have the language code specified:

```
<training_dir>/dgtools dgautotrain <dataset_name> --lang-code XX
```

Where XX matches the <lang_code> in the two files you copied to the <training_dir>/deployablemodels directory. The default is 'en' for english and for turkish it is 'tr'. That is it. The exact same steps described in the previous section will take place and in the end a final model will be created and exported to the deployablemodels directory.

Still basic - topping off a model with more data

Many customers train an initial model, but then later more data becomes available so it becomes desirable to train a new model with this additional data included. There are two ways to handle this scenario. First, if the total data is not large, generally less than 100-200 hours, and the added data is small, just a few hours, it is reasonable to just start from scratch. With dgautotrain this is easy. First, make sure that the new data is added to the existing dataset directory within the <training_dir>/rawdata/<dataset_name> directory. As discussed before, where you put it under this directory doesn't matter, but for general organization is it useful to create subdirectories named by date. With the new data added, simply re-run dgautotrain but with the --start-fresh and --keep-existing options:

```
<training_dir>/dgtools dgautotrain <model_name> --start-fresh
```

This will differ slightly from the first time dgautotrain was run in that it will not re-train the utility model and it will only chunk the new data added to the dataset.

The second way to train is to allow dgautotrain to 'top-off' the existing model. This is exactly the same, only you should remove the --start-fresh option:

```
<training_dir>/dgtools dgautotrain <model_name>
```

This option is generally a better option than starting from scratch but it does have a potential drawback, model overfit. Model overfit happens when a model has seen training data too many times. This leads to a model ‘memorizing’ the data it is training on instead of learning generic features that let it do well on data it hasn’t seen before. Every time you ‘top-off’ a model instead of starting fresh the model gets another chance to see the old data and it can start to overfit to that old data. If there is a lot of old data, more than 100-200 hours, or the new data added is significant, 20% increase in total data, it isn;t likely to be an issue and is actually probably best to top-off instead of starting fresh. Unfortunately these are just general guidelines and can change depending on the exact use case and data. Consult Deepgram for specifics on your training use case.

More advanced - updating the utility model to recapture more data

DGAutotrain does a good job of chunking training data for use in training, but it isn’t perfect and sometimes the data can be challenging. Challenging data can lead to bad chunks and low ‘recapture’ rates which in turn leads to less optimal custom models. Good models rest on good data that is well chunked so a key to improving model performance is improving the chinking. DGAutotrain attempts to improve chunking by training a custom utility model to help it chunk the data. It ‘bootstraps’ this model by using an initial general utility model to perform chunking and then using those results to train a follow-on custom utility model that will then creat new, cleaner chunks. Sometimes, especially with hard data or when a lot of new data is added, it is useful to ‘top-off’ the utility model and re-chunk after that model has been trained. In fact, it is often good to do this train/chunk/train/chunk cycle several times to continue to refine the utility model. The commands to do this are simple:

```
<training_dir>/dgtools dgtrainer <dataset_name> --type phonemes --epochs 4
<training_dir>/dgtools dgchunker <dataset_name> --workers 20
```

One thing to note, both the dgtrainer and dgchunker commands require a --lang-code XX option if training on data other than english. The first command, dgtrainer, will update the utility model and the second command, dgchunker, will re-chunk the data into cleaner training utterances using the newly trained utility model. At the end of chunking dgchunker will display a recapture percentage. The higher that number the better, but in general a well chunked dataset will recapture around 90% of the audio for training. 100% is rare mainly due to long pauses and other untrainable gaps that appear in most audio. If you repeat the dgtrainer/dgchunker commands several times you will likely see the recapture go up and then plateau at which point further training is likely to hurt the data and not help it. Just like the main model, the utility model is subject to overfit. The --workers option controls the number of CPU workers used for chunking. This should generally match the number of available CPU cores on the training machine to reduce the amount of time it takes to chunk the data.

After updating the utility model you can re-train with dgautotrain on the now cleaner data to hopefully better results.

Testing a model

After training is complete it is useful to see how well it is doing on the data. As was discussed before, some of the supplied data is withheld for validation. We can use these files to test the accuracy of the model and get sample transcripts to look at and compare against the truth transcripts.

Testing a model is easy. Just run:

```
<training_dir>/dgtools dgtestwer <model_name>
```

This will run the model against the validation files and output results into <training_dir>/testresults/<model_name>_<dataset-name>. For simple use cases the model name and the dataset name will be the same. Inside this directory you will find a results.txt and a results.csv file. Both contain basically the same information, total word counts, error counts and word error rates for every file tested. These minimum outputs are useful for finding 'hard' files that may need more training data and are the minimum outputs from dgtestwer. To get more outputs use the --full flag:

```
<training_dir>/dgtools dgtestwer <model_name> --full
```

This option will put results in the same directory but add two new files, wordresults.txt and wordresults.csv. These files will show detailed results by word and can help show what words are the model is having problems with and may need more data on. Additionally, the results.txt file will now contain full text outputs from the model with word by word comparisons against the truth text showing what errors were made and where. This is again useful for finding what the model needs to improve on as well as if there are any truth data errors. It is sometimes the case that special characters or tags appear in the transcript that text cleaning is unable to find and remove. It is also common to find systemic truth transcription issues. For example, inconsistently transcribing 'okay' as 'k'. These issues generally become obvious after running the dgtestwer with the --full option. Finally, the results.csv file will now also report word recall rates (WRR) as well as WER. The reason this isn't the default is because these outputs now contain some of your original training data (the truth transcripts) and for sensitive training data, such as PHI data, it is important to minimize when training data is copied.

After testing a custom model it is nice to know how well it compares to the base model. Testing a base model is just as easy:

```
<training_dir>/dgtools dgtestwer general --dataset <dataset_name>
```

This will output results into <training_dir>/testresults/general_<dataset_name> with all the same information as the custom model. One thing to note, when testing a language other than english --lang-code XX must be supplied or testing will fail.

Deploying a model

In order to get your custom model to work within the API, you must do the following.

Copy the model to the models/ folder:

```
cp <training_dir>/deployablemodels/<model_name>.tar ~/models/
```

Add the following lines to ~/cfg/engine.toml :

```
[[products]]
  # The name of the model (if no model is specified, the API will try
  # to find the "general" model).
  name = "<model_name>"
  # Version string.
  generation = "alpha"
  # Path to the weights on disk.
  path = "/models/<model_name>.tar"
```

where <model_name> should be replaced with the name of the model you are deploying. If the model is for a language other than English, the entry in ~/cfg/engine.toml should look like this:

```
[[products]]
  # The name of the model (if no model is specified, the API will try
  # to find the "general" model).
  name = "<model_name>"
  # Language codes
  languages = ["<lang_code>"]
  # Version string.
  generation = "alpha"
  # Path to the weights on disk.
  path = "/models/<model_name>.tar"
```

where <lang_code> should be replaced with the two-letter language code for the model.

Finally, restart the API by executing the following command from any directory:

```
docker-compose restart engine
```

Once it finishes restarting, your new model should be available for use in the API.