TensorFlow for predicting aptamer affinity

[This documentation was lightly edited for manuscript peer reviewers: links to other project documentation have been removed (or where possible updated to external links, eg Vizier) and references to collaborations outside this manuscript have been changed to “xxx”.]

This document describes the code in aptamers/learning/, written as part of the Aptamer project. This code creates and trains a TensorFlow (TF) model that takes as input a DNA sequence and predicts affinity to a target molecule.

Overview

The overview document describes the general aptamer project including the long-term goals of the project and how all the technical pieces fit together. This document covers the TensorFlow (TF) model portion of the code base, which is the code inside /xxx/learning/ .

If you just want to start running code, there is a guide to the preprocessing pipeline (preprocess/g3doc/preprocess_walkthrough.md) that explains going from raw FASTQ data to tf.Example protos ready for model training and a training playbook (learning/g3doc/playbook.md) that describes the commands for training with some suggestions about analysis after training.

An aptamer is a piece of DNA that binds to a particular target molecule, usually a protein. The purpose of the TF model is to predict how well a DNA sequence bind a target molecule for which it has been trained, proxied as a prediction as to how many times the DNA sequence will appear in a pool of DNA sequences after the pool has been enriched for sequences that bind the target molecule. Note that we are not trying to perform *de novo* prediction -- the TF model has to be trained for each target molecule.

Our training data is next generation sequencing results from aptamer selection experiments. The selection experiments start with a pool of about 1e14 pieces of DNA with random sequences (imagine a random sequence simply as a string 40 characters long where each character has an equal probability of being A, C, G, or T). In successive rounds, the DNA sequences (aptamers) that bind to the target molecule are copied and enriched in the population. Next generation sequencing provides a way to sample the sequences in the pool, so in the sequencing results we expect to see sequences that bind well increase in count value. However, this sampling does not cover the full set of sequences in the pool. In general, we sequence about 1e7 sequences from each round of selection where the initial pool is 1e14. But, over the rounds of selection, the proportion of the pool made up by good binders increases so they are sampled more frequently.

To put it computationally, the input to the TF model is a table where the index column is the DNA sequence (a string 40 characters long) and the data columns of the table each represent how many times the given DNA sequence was found in the sequence sample from a specific selection round. The model tries to predict, based on the characters in the DNA sequence, what the count of the sequence will be in each round of selection. From this count, we can calculate the affinity of the DNA sequence. (Some alternative model architectures invert the roles of affinity and count. These are described below in the Output Layers section.)

The model is a simple feedforward network with some complications around the output layers where different versions of the model handle affinity vs. sequencing counts in different ways. The hidden layers of the network can vary from zero to three fully connected layers and zero to three convolutional layers. The convolutional layers can be standard (use rate to slide the kernel) or a’trous (dilated; use the rate to expand the kernel).

The initial code for this model was written before TF was totally mature, there is some legacy code throughout the codebase that is fairly raw where there might be better versions now.

Configuration

Configuring datasets

Metadata proto (selection.proto)

The metadata for experiments is contained in the experiment_proto which is defined in selection.proto. [Protos are Google internal text files used for data storage & access from code, like JSON or XML.] For example, the Aptitude experiment.proto shows how the experiment was performed and what the results are.

The experiment.proto files are kept alongside the data files because they are metadata for a particular set of data files. We thought about checking them into google3, but it is imperative that the version used for a particular training run be perfectly clear.

The experiment.proto is used during the preprocessing pipeline to determine all the data for the experiment. The preprocessing g3doc covers the pipeline in detail. Briefly, all the FASTQ files containing the raw data to be used for training should be identified in the experiment.proto. The preprocessing pipeline then creates a test and a training experiment.proto for each fold with the FASTQ links removed and read count information added. For example, if you train with fold 0 as the validation (test) fold, which is the default, then the read counts in the _test.pbtxt will indicate the read counts in fold 0 and the read counts in the _train.pbtxt will indicate the read counts in the remaining 80% of the data, i.e. the data you would be training on if you used fold 0 as your validation fold.

The experiment.proto files used during training are those created alongside the SSTable of tf.Example protos. These no longer have FASTQ filenames because the data is now in the SSTable, but they do have the read count values so these don't have to be recalculated. The test and train experient.proto files used in training are recopied into the training output file as further reporting of exactly how training was performed.

Config.py

General configuration values can be found in learning/config.py.

Some of the general configuration set up here includes that we use fold 0 as the test (validation) fold and standard filenames.

Each dataset has two entries here. The first, defined in INPUT_DATA_DIRS indicates where the data for this dataset is located. This is the output directory for the preprocessing pipeline. The

second, defined in DEFAULT_AFFINITY_TARGET_MAPS, sets up how to calculate affinity given the selection experiment and sequencing results. For example, for the xxx datasets, affinity is calculated as the sum of the last two rounds of selection ('round3_count' and 'round4_count'). The names included in this map must be valid SequencingReads names in the experiment.proto. A more complicated example is the Aptitude dataset where there are two targets: 'target' (aka NGAL) calculates affinity to NGAL without serum whereas 'serum' calculates the affinity with serum present. Details on this map and its usage are covered below in the Output Layer section.

Configuring Training (train_feedforward.*)

The learning playbook describes how to run the model both locally and on many machines. This document strives not to repeat the actual commands but to talk about the code behind the training. Specifics about the construction of the feedforward network are covered in the Construction section below and specifics about reading the tf.Example protos into file queues are covered below in the Input Data section.

The code for training the model is in train_feedfoward.py and the code to run this training as a job is in xxx [not provided, google internal code].

Hyperparameters

The training code sets up the hyperparameters for the model. These are the configuration of exactly how the model should be constructed -- for example, will the input DNA be sent to the model as only one-hots or should the kmer counts be used also? Exactly how many fully connected layers should the model have?

All these configuration parameters are contained in an tf.HParams object which is, fundamentally, just a dictionary of configuration parameters that gets sent everywhere (it's the parameter 'hps' you'll see in many functions across the codebase).

In the simple example where you are running without a tuner, this HParams object is defined directly in the train_feedforward module (running with a tuner is described below), see _train_without_autotune for this. This function constructs an hps object with the default parameters and then overrides values sent into train_feedforward in the flags. Many of the flags into train_feedfoward are copied into the hps object explicitly, these are specified in the HPARAM_FLAGS list. In addition, if there are more parameters you want to override, you can use the hpconfig flag to set them. This flag allows a comma-separated string of any parameters you want copied into the hps object. Note that this copy occurs after the creation of the default object but before the explicit flags in HPARAM_FLAGS happens, so any parameters specified in hpconfig will override default values but will get overridden themselves if they are set in explicit flags.

Running with a tuner (Vizier)

Vizier is a tuning service for optimizing hyperparameter values. It can be used in one of two ways. The simple use for Vizier is to kick off many training jobs each with different hyperparameter values within the provided valid ranges. In this mode, the analysis to determine the best hyperparameters is on the user. The more advanced way to use Vizier is to iteratively train / evaluate to search for the best hyperparameters. (The supported algorithms are described in more detail on the Vizier website.)

The default for our model is Random Search (the first mode), though this can be changed using different flags when running the the training script.

When using Vizier, we set up a 'study' which provides a range of valid values for each hyperparameter. Similarly to without a tuner, we first create the default ranges for each hyperparameter and then copy in anything from FLAGS.hpconfig and then copy in the explicit HPARAM_FLAGS options.

In this case, we let the tuner determine when the study has completed. While the tuner says to keep training, we ask the tuner for a specific hps to use (i.e. pick specific values within all the valid ranges and provide a single configuration) then kick off a new training run with that hps and report results.

Training Loop

Within train_feedforward.py, the actual training is set up in the run_training function.

The training consists of the following steps:

1. Determine if the hps is feasible (Vizier can pick a hyperparater in each valid range where the final result is not feasible -- for example a valid convolutional stride and a valid number of convolutional layers may, in combination, be invalid for the given sequence length). Vizier has accounted for this by providing a way to end a training early and letting the tuner know that the particular parameters are infeasible.
2. Set up the paths to the training and validation SSTables, set up an output directory for this training run, etc.
3. Calculate statistics (e.g., mean and variance for the each count) for normalizing inputs.
4. Actually create the model
5. Create a trainer class that controls the actual training. Most of this is boilerplate / standard TF, but this is where the optimizer and loss are set up. The Optimizer is currently a MomentumOptimizer. If you wanted to change to a different algorithm, for example AdamOptimizer, this would be the area of the code to change.
6. Start the training loop: for each epoch (set number of training batches), the model will train then evaluate the model and write out the model and the evaluation report.

Training Script

The training script kicks off training a model and also provides all the settings for running Vizier and exporting the results to MLDash. These parameters are described in the learning playbook.

TF Model Code

Predicting Affinity vs. Counts (output_layers.py)

Normally this document would first cover the basics of the model and then go into detailed description, but in this case it makes sense to cover the output layers first because the difference between affinity and sequencing counts is vital to an understanding of the whole TF model.

As mentioned in the beginning, the training data we have is sequencing data from Next Generation Sequencing of aptamer selection experiments. The information we want is the affinity of a DNA sequence for a particular target molecule. This discrepancy between what we can measure & have in training data (sequencing counts) and what we truly want to know (affinity) is vital to understanding our model architecture.

We have a couple ways of handling this discrepancy resulting in different model architectures. One model architecture, Fully Observed, uses a basic neural network architecture approach: the data we have is sequencing counts, so design a network to predict sequencing counts. The calculation of affinity is then done based on the predicted counts. Another set of model architectures, Latent Affinity, explicitly model the affinity of the DNA sequences to the experimental selection target molecule using the sequencing counts to guide the value determined for the affinity.

In the code base, this complexity is managed in the output_layers.py module. Every other piece of the model (constructing the model, running training, running inference, evaluating performance, etc) is agnostic to whether the model is predicting sequencing counts or affinity.

First, some vital definitions:

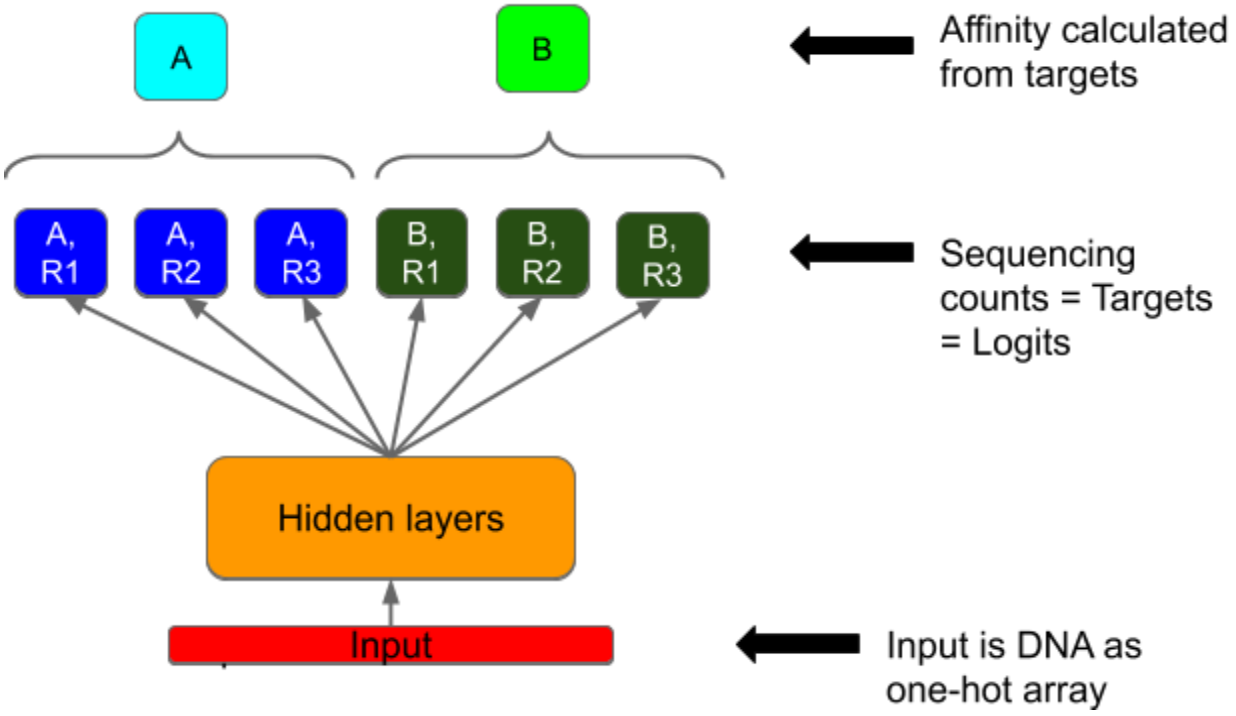
- **Target Molecule (aka Selection Target Molecule or just molecule)** : This is the actual molecule used in aptamer selection. In other words, this is the thing we want our aptamer DNA to bind. For Aptitude, this target molecule is NGAL, a protein commonly used as a biomarker for kidney distress. For xxx, the target molecules include xx, xxx, and xxx (all proteins) along with blank beads. Rounds of SELEX performed in the lab can actually have multiple target molecules due to experimental setup. For example, the xxx experiment was performed using magnetic beads, which were present in all

experimental conditions. Therefore, while experimentally we want to design aptamers to xxx, the target molecules for the xxx were actually both xxx and blank beads. Note that the 'target' in the experiment.proto means the target molecule, not the target as defined below.

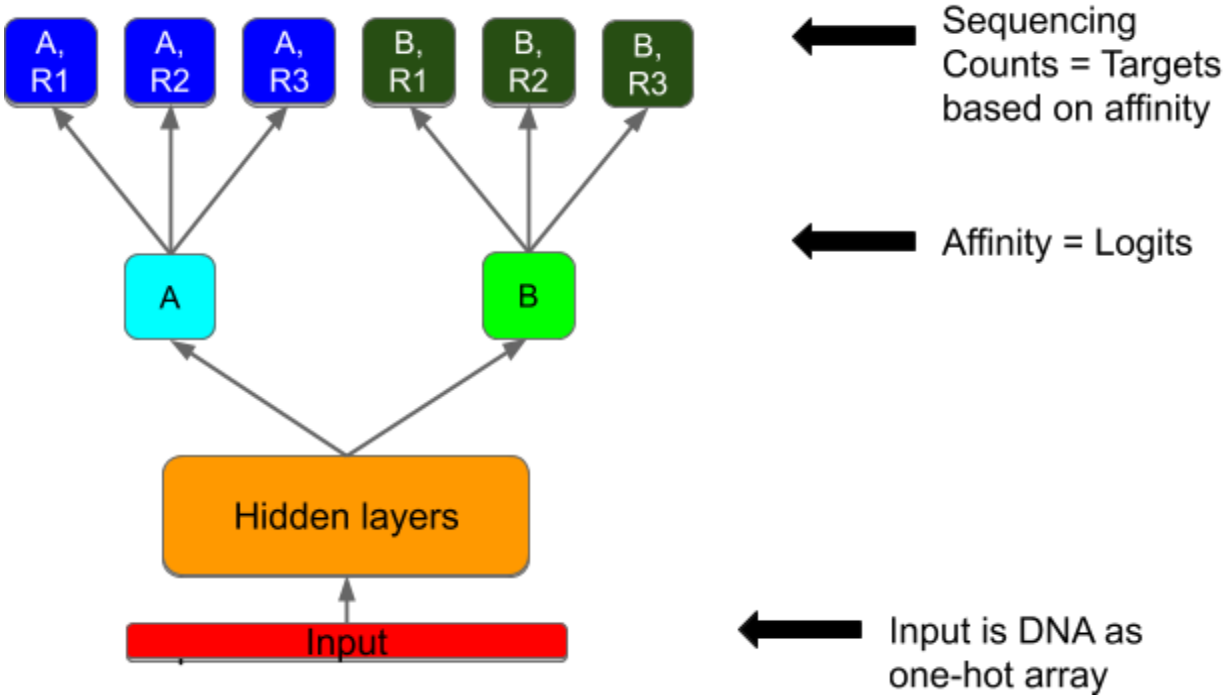
- **Target** : The target from the model's perspective, in other words, the count value of a sequencing pool. (Yes, this is super confusing. 'Target' as the target of SELEX is used everywhere and 'target' as the target of machine learning is used everywhere. In both cases, this is consistent with the field documentation so switching would have been confusing. Ultimately, we try very hard to use 'target' as the machine learning definition of the word and say the full 'target molecule' when talking about the selection experiment.) In machine learning terms, the target is the value you are trying to predict, either the prediction or the ground truth version thereof. If you drew the standard box & arrow version of a machine learning model, this would be the box into which you would write the final answer of your model so you could compare it to ground truth and determine loss.
- **Output** : The actual training data value, the ground truth value, also called 'labels' in TF. (Yes, yet another generic word used in a very specific manner here. We keep them because this way we are consistent with tensorflow documentation. In that documentation, 'input' is the name of the data that you feed into the model while 'output' is the name of the ground truth data you already have on what the model should say for your given input.) In this case, this is the actual sequencing pool value that we take from the tf.Example proto. So while the target describes the axis along which loss is calculated (which row of boxes holds the numbers to be used for loss), output always refers to the ground truth value in those boxes.
- **Logit** : In our model, logits are the output of the feedforward network (the orange box of machine learning magic in the diagram below). Whereas the target axis always lines up with the output axis, the logit axis may or may not. In the simple Fully Observed model, the logit axis and the target axis are the same so this distinction isn't required. For the Latent Affinity model, the target axis is still the sequencing pools (it must be -- that's where our training data is) but the logit axis is per affinity calculation.
- **Axis** : In our models, we use LabeledTensor instead of standard Tensors. Unlike the usual tensors in TF, LabeledTensors allow you to access them using string labels instead of numbers. This means when we calculate loss, for example, we can expect the parameter 'logits' to have batch and logit_axis. In another example, the complicated explanation below about Fully Observed vs. Latent Affinity model can be simply described as "In the Fully Observed model, the target_axis is the same as the logit_axis while the affinity axis requires calculation, while in the Latent Affinity models, the logit_axis is the same as the affinity_axis and then loss function must convert the logit_axis into the target_axis."

Here's a schematic of the Fully Observed model, where the training data we have (sequencing counts) is directly the values we are trying to predict. In the example below, we have performed 3 rounds each of selection against 2 targets. We have sequencing counts R1 (round1), R2, and

R3 for each of the targets A and B. The model suggests that affinity to A is calculated from all of R1, R2 and R3, though in practice we've found that using the sum of the last 2 rounds is the most correlated with affinity. The calculation of which rounds of selection should be used to calculate the affinity are defined for a specific dataset in the config.py file in DEFAULT_AFFINITY_TARGET_MAPS. However, this value does not need to be set unless you want to calculate the affinity for a target molecule. The training of the model never calculates this value -- training is based solely on sequencing counts -- so you don't need this map until you have a trained model and you want to predict affinity values from it.



Below is the schematic for the most simple version of the Latent Affinity model. In this model, the logits are the affinities. The trained model (and thus inference) no longer needs to proxy through the sequencing counts to predict how well a DNA sequence will bind a target molecule. In this model, calculating the loss during training is the part that requires information about the design of the experiment. The model loads this information from the experiment.proto and uses it to set up the calculation of loss. The details of this for different latent affinity models are explained in more detail below.



Within the output_layers.py module, there are definitions for all the different types of output layers, all extending from AbstractOutputLayers. Each output layer must override:

- loss_per_example_and_target : Method to calculate loss given the logits and the outputs. Regardless of the output layer class, returns loss for each target (sequencing pool).
- predict_counts : Method to predict outputs (aka sequencing counts) given the logits
- predict_affinity : Method to predict affinity per target given the logits

The FullyObserved layer has the logits equal to the targets (counts) so predict_counts is straightforward while predict_affinity requires logic and metadata to calculate affinity from sequencing counts.

The LatentAffinity layer (and its subclasses) have the logits equal to the affinity so predict_affinity is easy but predict_counts requires the metadata and calculation. In addition to the LatentAffinity as described, there are three subclasses that build in even more knowledge of the experiment into the model architecture:

- LatentAffinityWithDeps : In this model, the predicted count values for a sequence depend on both the predicted affinity (as base LatentAffinity) and the count for this sequence in the previous round. This mimics the actual selection experiment -- the prevalence of a sequence in a sequencing pool depends on whether it was in the previous sequencing pool and how well it bound to the target molecule. Because this

model uses counts from previous rounds as input, its predictions should *not* be directly compared to models that do not have access to this information.

- LatentAffinityWithPredDeps : This model is similar to the LatentAffinityWithDeps but instead of using the actual counts in the previous round, this model uses the predicted counts in the previous round. Usually for inference (running the trained model to score DNA, not training) only the affinity is required so LatentAffinityWithDeps can calculate affinity for any DNA sequence. However, LatentAffinityWithDeps cannot calculate predicted counts for sequences outside the initial experiment so if your model must be able to predict counts, you have to cannot use LatentAffinityWithDeps and must use LatentAffinityWithPredDeps instead. This class also has the important advantage of allowing direct comparisons to other output layers that do not make use of counts in previous rounds.
- LatentAffinityWithCrossDeps : Similar to LatentAffinityWithDeps but allows the predicted counts to also depend on the product of the count in the previous round and the affinity.

In addition to the different model architectures, there are two different loss functions which both derive from an AbstractLoss object. By creating them as objects deriving from a common class, it is possible to easily mix and match which loss function is applied to which model architecture. SquaredError loss does the standard square of the difference between predicted and actual value. CrossEntropy calls the TF function sigmoid_cross_entropy_with_logits which computes sigmoid cross entropy (loss is based more on is a count present and I called it present than whether the exact score is right).

Optionally, the model can also predict 1D auxiliary information such as the partition function or boltzman probability of the minimal-free-energy structure. Specify the set of 1D data to predict in the FLAG “additional_output” by concatenating the feature names with comma and encapsulating it with double quotes, for example “partition_function,boltzmann_probability”. Every feature name should be a valid key of “feature_tensors” in data.py. Regardless of the type of output layer, one output node that is directly connected to the hidden layers will be created for each of the 1D features to predict. The real labels of these predictions will be normalized in the same way as counts, and compared against the corresponding predictions with the same type of loss used for count predictions.

Reading Input Data (train_feedforward.py and data.py)

In the training section above, we glossed over exactly how the tf.Example protos are read and turned into input tensors for the network. This section dives into those details, in specific the “input_pipeline” function in train_feedfoward and the functionality in data.py.

The input is a DNA sequence which is encoded as one-hots. In other words, the input is a set of characters, generally 40 characters long, where each character can be A, C, G, or T and this

choice is represented to the model as 4 one-hots for each base. So a 40 base DNA sequence becomes a vector of 160 values which can be 0 or 1.

The input_pipeline function in train_feedforward is responsible for reading the SSTables of tf.Example protos and turning them into a queue of input and output tensors divided into batches for training. The code in input_pipeline is fairly boilerplate to run through the SSTables. Note that it invokes `string_input_producer` to create the queue and this function, by default, will shuffle the order of the strings returned each epoch.

The actual code to turn proto strs into input and output tensors is in data.py starting at the function preprocess. First, the code will parse the proto strs into a dictionary of feature tensors. For example, at this point the feature tensor named ‘sequence’ will contain the string sequence. This is also the place where random DNA with no sequencing counts can be added to the input data if requested on the command line (preprocess_mode flag includes ‘PREPROCESS_INJECT_RANDOM_SEQUENCES). After this, the feature tensors are converted into tensors suitable for input into the training model in the function create_inputs_and_outputs. For example, this function is where the feature tensor ‘sequence’ becomes an input tensor called ‘SEQUENCE_ONE_HOTS’ that is actually a tensor of zeros and ones with the axes position and channel. If kmers are requested on the command line, this is where the string sequence will be turned into kmer counts.

The downstream code will automatically reshape these tensors to feed into the model. These tensors are expected to be shaped in one of two ways:

- 2D tensors should have axes [‘batch’, X] where X can be anything. As an example, ‘kmer’ is like this. The secondary structure features that cover the whole sequence (like partition function) should be shaped like this.
- 3D tensors should have axes [‘batch’, ‘position’, X] where X can be anything. For example, the sequence one hots are like this. By default, convolutions are only run along the tensors with 3 dimensions and they are run along the position axes. The secondary structure features that occur per base (like probability that each base is paired) should be shaped like this.

Model Construction (feedforward.py)

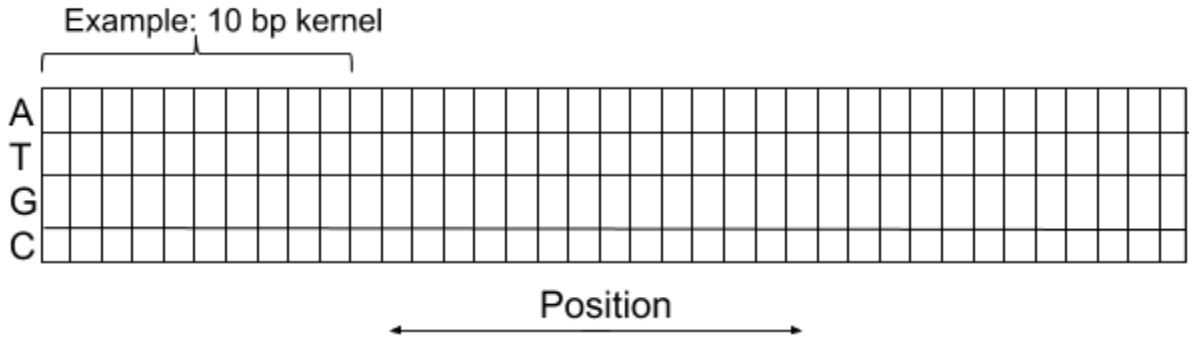
The code for constructing the model is in feedforward.py. This doesn’t include any information about training, just the actual instructions to hook all the inputs and layers together. This is where the convolutional and hidden layers are constructed and wired together, and where dropout are added.

The initialization takes in dummy_inputs, which are labeled tensors of the same shape as the inputs during training. These are constructed based on the experiment_proto using the same preprocessing code in data.py that reads the queue of tf.Example protos during training (see

description above), but in the case of the dummy_inputs, the labeled tensors are placeholders without any queue or SSTable behind them.

Initializing the FeedForward class wires the dummy_inputs that are passed in, connecting them through the requested convolutional and hidden layers based on the config, and then connecting them to the logits (which are passed in as a parameter because the output layer is set up as part of the training code).

The convolutional layers apply to all 3D input tensors (which have axes of ['batch', 'position', X] as described above) but not to the 2D input tensors. The convolution is across the position axis, in this case X is 'channel' representing the bases:



The actual convolutions are set up in conv1d. The name is confusing -- our labeled tensor is 3 dimensions -- but we are only doing the convolution in 1 dimension, the position dimension, so this function name aligns with the rest of the tensorflow documentation. The standard convolution defines a filter_width, which is the width of the kernel (the number of bases), and a stride (how many bases to shift each step, generally 1 or 2). Dilated (aka a'trous) convolutions can also be done by specifying a rate, which is how much to spread out the kernel each step.

Inference and Evaluation

Inference is running examples through a network to get their output results. Comparing these results to ground truth in the examples is called evaluation.

After a model is trained, inference can be used to score examples whether or not they have associated ground truth (in our case, whether or not they have sequencing counts from an experiment). This inference is how we use the trained models to search for never-seen-before sequences that are better aptamers.

Evaluation is run on both training data and validation data during training to determine how well training is progressing. In our models, the evaluation is set up in train_feedforward and is

automatically run after each epoch of training. (Other models set up a separate process to run evaluation, depending on preferences and how they coordinate with a tuner service like Vizier.)

Running inference on a trained model (eval_feedforward.py)

The eval_feedforward module defines an Inferer object that knows how to load a model from a saved checkpoint and run a set of examples through the model to return their output scores.

The examples can be provided as paths to SSTables of TF example protos ('run_on_files'), as TF example protos themselves ('run_on_examples') or as sequences ('run_on_sequences').

Evaluation during training (eval_feedforward.py)

During training, you want to know how training is progressing. In our case, after each epoch of training, we run evaluation on some number of examples from our training and test folds, and we write out the reports into the checkpoint directory and report the results to a tuner service if we are using a tuner service.

The code to use this evaluation framework is found in train_feedforward. This code creates an evaluator for the training and test datasets. Each epoch, the training loop calls 'run' on the training set evaluator and 'run_and_report' on the test set evaluator (to report values back to the tuner service).

The Evaluator object is defined in eval_feedforward. This object is a wrapper around an Inferer Object. Each time the evaluator is run it constructs an Inferer, runs the desired number of examples through inference, then reports on the difference between the ground truth values and the inference results.