adanet_objective

April 24, 2019


```
In [0]: #@title Licensed under the Apache License, Version 2.0 (the "License");
        # you may not use this file except in compliance with the License.
        # You may obtain a copy of the License at
        #
        # https://www.apache.org/licenses/LICENSE-2.0
        #
        # Unless required by applicable law or agreed to in writing, software
        # distributed under the License is distributed on an "AS IS" BASIS,
        # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        # See the License for the specific language governing permissions and
        # limitations under the License.
```

1 The AdaNet objective

```
<a target="_blank" href="https://colab.research.google.com/github/tensorflow/adanet/blob/maste

<a target="_blank" href="https://github.com/tensorflow/adanet/blob/master/adanet/examples/tuto
```

One of key contributions from *AdaNet: Adaptive Structural Learning of Neural Networks* [Cortes et al., ICML 2017] is defining an algorithm that aims to directly minimize the DeepBoost generalization bound from *Deep Boosting* [Cortes et al., ICML 2014] when applied to neural networks. This algorithm, called **AdaNet**, adaptively grows a neural network as an ensemble of subnetworks that minimizes the AdaNet objective (a.k.a. AdaNet loss):

$$F(w) = \frac{1}{m}\sum_{i=1}^{m}\Phi\left(\sum_{j=1}^{N} w_j h_j(x_i), y_i\right) + \sum_{j=1}^{N}\left(\lambda r(h_j) + \beta\right)|w_j|$$

where w is the set of mixture weights, one per subnetwork h, Φ is a surrogate loss function such as logistic loss or MSE, r is a function for measuring a subnetwork's complexity, and λ and β are hyperparameters.

1.1 Mixture weights

So what are mixture weights? When forming an ensemble f of subnetworks h, we need to somehow combine their predictions. This is done by multiplying the outputs of subnetwork h_i with mixture weight w_i, and summing the results:

$$f(x) = \sum_{j=1}^{N} w_j h_j(x)$$

In practice, most commonly used set of mixture weight is **uniform average weighting**:

$$f(x) = \frac{1}{N} \sum_{j=1}^{N} h_j(x)$$

However, we can also solve a convex optimization problem to learn the mixture weights that minimize the loss function Φ:

$$F(w) = \frac{1}{m} \sum_{i=1}^{m} \Phi \left(\sum_{j=1}^{N} w_j h_j(x_i), y_i \right)$$

This is the first term in the AdaNet objective. The second term applies L1 regularization to the mixture weights:

$$\sum_{j=1}^{N} \left(\lambda r(h_j) + \beta \right) |w_j|$$

When $\lambda > 0$ this penalty serves to prevent the optimization from assigning too much weight to more complex subnetworks according to the complexity measure function r.

1.2 How AdaNet uses the objective

This objective function serves two purposes:

1. To **learn to scale/transform the outputs of each subnetwork** h as part of the ensemble.
2. To **select the best candidate subnetwork** h at each AdaNet iteration to include in the ensemble.

Effectively, when learning mixture weights w, AdaNet solves a convex combination of the outputs of the frozen subnetworks h. For $\lambda > 0$, AdaNet penalizes more complex subnetworks with greater L1 regularization on their mixture weight, and will be less likely to select more complex subnetworks to add to the ensemble at each iteration.

In this tutorial, in you will observe the benefits of using AdaNet to learn the ensemble's mixture weights and to perform candidate selection.

```
In [0]: #@test {"skip": true}
        # If you're running this in Colab, first install the adanet package:
        !pip install adanet
```

```
Collecting adanet
  Downloading https://files.pythonhosted.org/packages/e9/20/57c776bf4574c6e666bfb4434ace74755f
    100% || 71kB 4.5MB/s
Requirement already satisfied: numpy>=1.12.0 in /usr/local/lib/python2.7/dist-packages (from a
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python2.7/dist-packages (from ada
Requirement already satisfied: protobuf>=3.6.0 in /usr/local/lib/python2.7/dist-packages (from
Requirement already satisfied: setuptools in /usr/local/lib/python2.7/dist-packages (from prot
```

```
Installing collected packages: adanet
Successfully installed adanet-0.5.0
```

```
In [0]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        import functools
        import os
        import shutil

        import adanet
        import tensorflow as tf

        # The random seed to use.
        RANDOM_SEED = 42

        LOG_DIR = '/tmp/models'
```

1.3 Boston Housing dataset

In this example, we will solve a regression task known as the Boston Housing dataset to predict the price of suburban houses in Boston, MA in the 1970s. There are 13 numerical features, the labels are in thousands of dollars, and there are only 506 examples.

1.4 Download the data

Conveniently, the data is available via Keras:

```
In [0]: (x_train, y_train), (x_test, y_test) = (
            tf.keras.datasets.boston_housing.load_data())

        # Preview the first example from the training data
        print('Model inputs: %s \n' % x_train[0])
        print('Model output (house price): $%s ' % (y_train[0] * 1000))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston_housi
57344/57026 [==============================] - 0s 0us/step
65536/57026 [==============================] - 0s 0us/step
Model inputs: [  1.23247   0.        8.14      0.        0.538     6.142    91.7
   3.9769    4.      307.       21.      396.9      18.72   ]

Model output (house price): $15200.0
```

1.5 Supply the data in TensorFlow

Our first task is to supply the data in TensorFlow. Using the tf.estimator.Estimator convention, we will define a function that returns an input_fn which returns feature and label Tensors.

We will also use the tf.data.Dataset API to feed the data into our models.

Also, as a preprocessing step, we will apply `tf.log1p` to log-scale the features and labels for improved numerical stability during training. To recover the model's predictions in the correct scale, you can apply `tf.math.expm1` to the prediction.

```
In [0]: FEATURES_KEY = "x"
```

```
def input_fn(partition, training, batch_size):
  """Generate an input function for the Estimator."""

  def _input_fn():

    if partition == "train":
      dataset = tf.data.Dataset.from_tensor_slices(({
          FEATURES_KEY: tf.log1p(x_train)
      }, tf.log1p(y_train)))
    else:
      dataset = tf.data.Dataset.from_tensor_slices(({
          FEATURES_KEY: tf.log1p(x_test)
      }, tf.log1p(y_test)))

    # We call repeat after shuffling, rather than before, to prevent separate
    # epochs from blending together.
    if training:
      dataset = dataset.shuffle(10 * batch_size, seed=RANDOM_SEED).repeat()

    dataset = dataset.batch(batch_size)
    iterator = dataset.make_one_shot_iterator()
    features, labels = iterator.get_next()
    return features, labels

  return _input_fn
```

1.6 Define the subnetwork generator

Let's define a subnetwork generator similar to the one in [Cortes et al., ICML 2017] and in `simple_dnn.py` which creates two candidate fully-connected neural networks at each iteration with the same width, but one an additional hidden layer. To make our generator *adaptive*, each subnetwork will have at least the same number of hidden layers as the most recently added subnetwork to the `previous_ensemble`.

We define the complexity measure function r to be $r(h) = \sqrt{d(h)}$, where d is the number of hidden layers in the neural network h, to approximate the Rademacher bounds from [Golowich et. al, 2017]. So subnetworks with more hidden layers, and therefore more capacity, will have more heavily regularized mixture weights.

```
In [0]: _NUM_LAYERS_KEY = "num_layers"
```

```
class _SimpleDNNBuilder(adanet.subnetwork.Builder):
  """Builds a DNN subnetwork for AdaNet."""

  def __init__(self, optimizer, layer_size, num_layers, learn_mixture_weights,
               seed):
    """Initializes a `_DNNBuilder`.

    Args:
      optimizer: An `Optimizer` instance for training both the subnetwork and
        the mixture weights.
      layer_size: The number of nodes to output at each hidden layer.
      num_layers: The number of hidden layers.
      learn_mixture_weights: Whether to solve a learning problem to find the
        best mixture weights, or use their default value according to the
        mixture weight type. When `False`, the subnetworks will return a no_op
        for the mixture weight train op.
      seed: A random seed.

    Returns:
      An instance of `_SimpleDNNBuilder`.
    """

    self._optimizer = optimizer
    self._layer_size = layer_size
    self._num_layers = num_layers
    self._learn_mixture_weights = learn_mixture_weights
    self._seed = seed

  def build_subnetwork(self,
                       features,
                       logits_dimension,
                       training,
                       iteration_step,
                       summary,
                       previous_ensemble=None):
    """See `adanet.subnetwork.Builder`."""

    input_layer = tf.to_float(features[FEATURES_KEY])
    kernel_initializer = tf.glorot_uniform_initializer(seed=self._seed)
    last_layer = input_layer
    for _ in range(self._num_layers):
      last_layer = tf.layers.dense(
          last_layer,
          units=self._layer_size,
          activation=tf.nn.relu,
          kernel_initializer=kernel_initializer)
    logits = tf.layers.dense(
```

```
        last_layer,
        units=logits_dimension,
        kernel_initializer=kernel_initializer)

    persisted_tensors = {_NUM_LAYERS_KEY: tf.constant(self._num_layers)}
    return adanet.Subnetwork(
        last_layer=last_layer,
        logits=logits,
        complexity=self._measure_complexity(),
        persisted_tensors=persisted_tensors)

  def _measure_complexity(self):
    """Approximates Rademacher complexity as the square-root of the depth."""
    return tf.sqrt(tf.to_float(self._num_layers))

  def build_subnetwork_train_op(self, subnetwork, loss, var_list, labels,
                                iteration_step, summary, previous_ensemble):
    """See `adanet.subnetwork.Builder`."""
    return self._optimizer.minimize(loss=loss, var_list=var_list)

  def build_mixture_weights_train_op(self, loss, var_list, logits, labels,
                                     iteration_step, summary):
    """See `adanet.subnetwork.Builder`."""

    if not self._learn_mixture_weights:
      return tf.no_op()
    return self._optimizer.minimize(loss=loss, var_list=var_list)

  @property
  def name(self):
    """See `adanet.subnetwork.Builder`."""

    if self._num_layers == 0:
      # A DNN with no hidden layers is a linear model.
      return "linear"
    return "{}_layer_dnn".format(self._num_layers)


class SimpleDNNGenerator(adanet.subnetwork.Generator):
  """Generates a two DNN subnetworks at each iteration.

  The first DNN has an identical shape to the most recently added subnetwork
  in `previous_ensemble`. The second has the same shape plus one more dense
  layer on top. This is similar to the adaptive network presented in Figure 2 of
  [Cortes et al. ICML 2017](https://arxiv.org/abs/1607.01097), without the
  connections to hidden layers of networks from previous iterations.
  """
```

```
def __init__(self,
             optimizer,
             layer_size=64,
             learn_mixture_weights=False,
             seed=None):
  """Initializes a DNN `Generator`.

  Args:
    optimizer: An `Optimizer` instance for training both the subnetwork and
      the mixture weights.
    layer_size: Number of nodes in each hidden layer of the subnetwork
      candidates. Note that this parameter is ignored in a DNN with no hidden
      layers.
    learn_mixture_weights: Whether to solve a learning problem to find the
      best mixture weights, or use their default value according to the
      mixture weight type. When `False`, the subnetworks will return a no_op
      for the mixture weight train op.
    seed: A random seed.

  Returns:
    An instance of `Generator`.
  """

  self._seed = seed
  self._dnn_builder_fn = functools.partial(
      _SimpleDNNBuilder,
      optimizer=optimizer,
      layer_size=layer_size,
      learn_mixture_weights=learn_mixture_weights)

def generate_candidates(self, previous_ensemble, iteration_number,
                        previous_ensemble_reports, all_reports):
  """See `adanet.subnetwork.Generator`."""

  num_layers = 0
  seed = self._seed
  if previous_ensemble:
    num_layers = tf.contrib.util.constant_value(
        previous_ensemble.weighted_subnetworks[
            -1].subnetwork.persisted_tensors[_NUM_LAYERS_KEY])
  if seed is not None:
    seed += iteration_number
  return [
      self._dnn_builder_fn(num_layers=num_layers, seed=seed),
      self._dnn_builder_fn(num_layers=num_layers + 1, seed=seed),
  ]
```

1.7 Launch TensorBoard

Let's run TensorBoard to visualize model training over time. We'll use ngrok to tunnel traffic to localhost.

The instructions for setting up Tensorboard were obtained from https://www.dlology.com/blog/quick-guide-to-run-tensorboard-in-google-colab/

Run the next cells and follow the link to see the TensorBoard in a new tab.

```
In [0]: #@test {"skip": true}

        get_ipython().system_raw(
            'tensorboard --logdir {} --host 0.0.0.0 --port 6006 &'
            .format(LOG_DIR)
        )

        # Install ngrok binary.
        ! wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
        ! unzip ngrok-stable-linux-amd64.zip

        # Delete old logs dir.
        shutil.rmtree(LOG_DIR, ignore_errors=True)

        print("Follow this link to open TensorBoard in a new tab.")
        get_ipython().system_raw('./ngrok http 6006 &')
        ! curl -s http://localhost:4040/api/tunnels | python3 -c \
            "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public_url'])"
```

1.8 Train and evaluate

Next we create an `adanet.Estimator` using the `SimpleDNNGenerator` we just defined.

In this section we will show the effects of two hyperparamters: **learning mixture weights** and **complexity regularization**.

On the righthand side you will be able to play with the hyperparameters of this model. Until you reach the end of this section, we ask that you not change them.

At first we will not learn the mixture weights, using their default initial value. Here they will be scalars initialized to $1/N$ where N is the number of subnetworks in the ensemble, effectively creating a **uniform average ensemble**.

```
In [0]: #@title AdaNet parameters
        LEARNING_RATE = 0.001  #@param {type:"number"}
        TRAIN_STEPS = 60000  #@param {type:"integer"}
        BATCH_SIZE = 32  #@param {type:"integer"}

        LEARN_MIXTURE_WEIGHTS = False  #@param {type:"boolean"}
        ADANET_LAMBDA = 0  #@param {type:"number"}
        ADANET_ITERATIONS = 3  #@param {type:"integer"}


        def train_and_evaluate(experiment_name, learn_mixture_weights=LEARN_MIXTURE_WEIGHTS,
```

```
                      adanet_lambda=ADANET_LAMBDA):
  """Trains an `adanet.Estimator` to predict housing prices."""

  model_dir = os.path.join(LOG_DIR, experiment_name)

  estimator = adanet.Estimator(
      # Since we are predicting housing prices, we'll use a regression
      # head that optimizes for MSE.
      head=tf.contrib.estimator.regression_head(
          loss_reduction=tf.losses.Reduction.SUM_OVER_BATCH_SIZE),

      # Define the generator, which defines our search space of subnetworks
      # to train as candidates to add to the final AdaNet model.
      subnetwork_generator=SimpleDNNGenerator(
          optimizer=tf.train.RMSPropOptimizer(learning_rate=LEARNING_RATE),
          learn_mixture_weights=learn_mixture_weights,
          seed=RANDOM_SEED),

      # Lambda is a the strength of complexity regularization. A larger
      # value will penalize more complex subnetworks.
      adanet_lambda=adanet_lambda,

      # The number of train steps per iteration.
      max_iteration_steps=TRAIN_STEPS // ADANET_ITERATIONS,

      # The evaluator will evaluate the model on the full training set to
      # compute the overall AdaNet loss (train loss + complexity
      # regularization) to select the best candidate to include in the
      # final AdaNet model.
      evaluator=adanet.Evaluator(
          input_fn=input_fn("train", training=False, batch_size=BATCH_SIZE)),

      # Configuration for Estimators.
      config=tf.estimator.RunConfig(
          save_summary_steps=5000,
          save_checkpoints_steps=5000,
          tf_random_seed=RANDOM_SEED,
          model_dir=model_dir))

  # Train and evaluate using using the tf.estimator tooling.
  train_spec = tf.estimator.TrainSpec(
      input_fn=input_fn("train", training=True, batch_size=BATCH_SIZE),
      max_steps=TRAIN_STEPS)
  eval_spec = tf.estimator.EvalSpec(
      input_fn=input_fn("test", training=False, batch_size=BATCH_SIZE),
      steps=None,
      start_delay_secs=1,
      throttle_secs=30,
```

```
        )
        return tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)


      def ensemble_architecture(result):
        """Extracts the ensemble architecture from evaluation results."""

        architecture = result["architecture/adanet/ensembles"]
        # The architecture is a serialized Summary proto for TensorBoard.
        summary_proto = tf.summary.Summary.FromString(architecture)
        return summary_proto.value[0].tensor.string_val[0]
```

```
In [0]: results, _ = train_and_evaluate("uniform_average_ensemble_baseline")
        print("Loss:", results["average_loss"])
        print("Architecture:", ensemble_architecture(results))
```

```
INFO:tensorflow:Using config: {'_save_checkpoints_secs': None, '_session_config': allow_soft_pl
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_task_type': 'worker', '_train_distribute': None, '_is_chief': Tr
INFO:tensorflow:Not using Distribute Coordinator.
INFO:tensorflow:Running training and evaluation locally (non-distributed).
INFO:tensorflow:Start train and evaluate loop. The evaluate will happen after every checkpoint
INFO:tensorflow:Beginning training AdaNet iteration 0
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Building iteration 0
INFO:tensorflow:Building subnetwork 'linear'
WARNING:tensorflow:From <ipython-input-6-fb8e373057d9>:60: calling __new__ (from adanet.core.su
Instructions for updating:
`persisted_tensors` is deprecated, please use `shared` instead.
INFO:tensorflow:Building subnetwork '1_layer_dnn'
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 0 into /tmp/models/uniform_average_ensemble_baseline/mod
INFO:tensorflow:loss = 5.1627307, step = 1
INFO:tensorflow:global_step/sec: 281.388
INFO:tensorflow:loss = 0.36111063, step = 101 (0.356 sec)
INFO:tensorflow:global_step/sec: 632.512
INFO:tensorflow:loss = 0.23443298, step = 201 (0.158 sec)
INFO:tensorflow:global_step/sec: 623.624
INFO:tensorflow:loss = 0.05358686, step = 301 (0.160 sec)
INFO:tensorflow:global_step/sec: 634.308
```

```
INFO:tensorflow:loss = 0.056381542, step = 401 (0.158 sec)
INFO:tensorflow:global_step/sec: 619.218
INFO:tensorflow:loss = 0.06720045, step = 501 (0.161 sec)
INFO:tensorflow:global_step/sec: 629.826
INFO:tensorflow:loss = 0.0430552, step = 601 (0.159 sec)
INFO:tensorflow:global_step/sec: 626.814
INFO:tensorflow:loss = 0.02634022, step = 701 (0.160 sec)
INFO:tensorflow:global_step/sec: 633.646
INFO:tensorflow:loss = 0.02167327, step = 801 (0.157 sec)
INFO:tensorflow:global_step/sec: 639.223
INFO:tensorflow:loss = 0.03461258, step = 901 (0.157 sec)
INFO:tensorflow:global_step/sec: 632.959
INFO:tensorflow:loss = 0.10106905, step = 1001 (0.158 sec)
INFO:tensorflow:global_step/sec: 584.874
INFO:tensorflow:loss = 0.035603803, step = 1101 (0.171 sec)
INFO:tensorflow:global_step/sec: 526.681
INFO:tensorflow:loss = 0.05181156, step = 1201 (0.190 sec)
INFO:tensorflow:global_step/sec: 636.704
INFO:tensorflow:loss = 0.103541166, step = 1301 (0.157 sec)
INFO:tensorflow:global_step/sec: 640.824
INFO:tensorflow:loss = 0.025568489, step = 1401 (0.156 sec)
INFO:tensorflow:global_step/sec: 646.132
INFO:tensorflow:loss = 0.017370185, step = 1501 (0.154 sec)
INFO:tensorflow:global_step/sec: 640.73
INFO:tensorflow:loss = 0.046300475, step = 1601 (0.156 sec)
INFO:tensorflow:global_step/sec: 654.467
INFO:tensorflow:loss = 0.04478077, step = 1701 (0.153 sec)
INFO:tensorflow:global_step/sec: 631.445
INFO:tensorflow:loss = 0.037187986, step = 1801 (0.158 sec)
INFO:tensorflow:global_step/sec: 575.729
INFO:tensorflow:loss = 0.041324023, step = 1901 (0.174 sec)
INFO:tensorflow:global_step/sec: 630.163
INFO:tensorflow:loss = 0.03305667, step = 2001 (0.159 sec)
INFO:tensorflow:global_step/sec: 641.272
INFO:tensorflow:loss = 0.047040317, step = 2101 (0.156 sec)
INFO:tensorflow:global_step/sec: 505.523
INFO:tensorflow:loss = 0.02897141, step = 2201 (0.198 sec)
INFO:tensorflow:global_step/sec: 615.858
INFO:tensorflow:loss = 0.034482427, step = 2301 (0.162 sec)
INFO:tensorflow:global_step/sec: 631.804
INFO:tensorflow:loss = 0.02347619, step = 2401 (0.158 sec)
INFO:tensorflow:global_step/sec: 627.955
INFO:tensorflow:loss = 0.07312092, step = 2501 (0.159 sec)
INFO:tensorflow:global_step/sec: 650.335
INFO:tensorflow:loss = 0.037120573, step = 2601 (0.154 sec)
INFO:tensorflow:global_step/sec: 548.438
INFO:tensorflow:loss = 0.05865065, step = 2701 (0.183 sec)
INFO:tensorflow:global_step/sec: 620.456
```

```
INFO:tensorflow:loss = 0.013324069, step = 2801 (0.161 sec)
INFO:tensorflow:global_step/sec: 639.537
INFO:tensorflow:loss = 0.03051297, step = 2901 (0.156 sec)
INFO:tensorflow:global_step/sec: 630.354
INFO:tensorflow:loss = 0.038826123, step = 3001 (0.159 sec)
INFO:tensorflow:global_step/sec: 639.223
INFO:tensorflow:loss = 0.028354896, step = 3101 (0.157 sec)
INFO:tensorflow:global_step/sec: 631.018
INFO:tensorflow:loss = 0.02475904, step = 3201 (0.158 sec)
INFO:tensorflow:global_step/sec: 501.975
INFO:tensorflow:loss = 0.046555758, step = 3301 (0.199 sec)
INFO:tensorflow:global_step/sec: 602.21
INFO:tensorflow:loss = 0.03615782, step = 3401 (0.166 sec)
INFO:tensorflow:global_step/sec: 601.913
INFO:tensorflow:loss = 0.05879674, step = 3501 (0.166 sec)
INFO:tensorflow:global_step/sec: 609.277
INFO:tensorflow:loss = 0.02304113, step = 3601 (0.164 sec)
INFO:tensorflow:global_step/sec: 644.118
INFO:tensorflow:loss = 0.015732538, step = 3701 (0.156 sec)
INFO:tensorflow:global_step/sec: 637.804
INFO:tensorflow:loss = 0.02152918, step = 3801 (0.157 sec)
INFO:tensorflow:global_step/sec: 653.219
INFO:tensorflow:loss = 0.019783746, step = 3901 (0.153 sec)
INFO:tensorflow:global_step/sec: 642.884
INFO:tensorflow:loss = 0.040957503, step = 4001 (0.155 sec)
INFO:tensorflow:global_step/sec: 641.688
INFO:tensorflow:loss = 0.01987027, step = 4101 (0.156 sec)
INFO:tensorflow:global_step/sec: 607.855
INFO:tensorflow:loss = 0.025079198, step = 4201 (0.164 sec)
INFO:tensorflow:global_step/sec: 562.6
INFO:tensorflow:loss = 0.012929243, step = 4301 (0.178 sec)
INFO:tensorflow:global_step/sec: 627.692
INFO:tensorflow:loss = 0.045354083, step = 4401 (0.159 sec)
INFO:tensorflow:global_step/sec: 644.982
INFO:tensorflow:loss = 0.030239534, step = 4501 (0.155 sec)
INFO:tensorflow:global_step/sec: 639.26
INFO:tensorflow:loss = 0.034541786, step = 4601 (0.157 sec)
INFO:tensorflow:global_step/sec: 642.174
INFO:tensorflow:loss = 0.04034666, step = 4701 (0.155 sec)
INFO:tensorflow:global_step/sec: 638.08
INFO:tensorflow:loss = 0.04709795, step = 4801 (0.157 sec)
INFO:tensorflow:global_step/sec: 621.786
INFO:tensorflow:loss = 0.021513231, step = 4901 (0.161 sec)
INFO:tensorflow:Saving checkpoints for 5000 into /tmp/models/uniform_average_ensemble_baseline/
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Building iteration 0
INFO:tensorflow:Building subnetwork 'linear'
INFO:tensorflow:Building subnetwork '1_layer_dnn'
```