adanet Documentation

Release 0.6.2

AdaNet Authors

Apr 29, 2019

Getting Started

AdaNet is a TensorFlow framework for fast and flexible AutoML with learning guarantees.

AdaNet is a lightweight TensorFlow-based framework for automatically learning high-quality models with minimal expert intervention. AdaNet builds on recent AutoML efforts to be fast and flexible while providing learning guarantees. Importantly, AdaNet provides a general framework for not only learning a neural network architecture, but also for learning to ensemble to obtain even better models.

This project is based on the *AdaNet algorithm*, presented in "AdaNet: Adaptive Structural Learning of Artificial Neural Networks" at ICML 2017, for learning the structure of a neural network as an ensemble of subnetworks.

AdaNet has the following goals:

- **Ease of use**: Provide familiar APIs (e.g. Keras, Estimator) for training, evaluating, and serving models.
- **Speed**: Scale with available compute and quickly produce high quality models.
- **Flexibility**: Allow researchers and practitioners to extend AdaNet to novel subnetwork architectures, search spaces, and tasks.
- **Learning guarantees**: Optimize an objective that offers theoretical learning guarantees.

The following animation shows AdaNet adaptively growing an ensemble of neural networks. At each iteration, it measures the ensemble loss for each candidate, and selects the best one to move onto the next iteration. At subsequent iterations, the blue subnetworks are frozen, and only yellow subnetworks are trained:

AdaNet was first announced on the Google AI research blog: "Introducing AdaNet: Fast and Flexible AutoML with Learning Guarantees".

This is not an official Google product.

CHAPTER 1

Overview

AdaNet is an extended implementation of *AdaNet: Adaptive Structural Learning of Artificial Neural Networks* by [Cortes et al., ICML 2017], an algorithm for iteratively learning both the **structure** and **weights** of a neural network as an **ensemble of subnetworks**.

1.1 Ensembles of subnetworks

In AdaNet, **ensembles** are first-class objects. Every model you train will be one form of an ensemble or another. An ensemble is composed of one or more **subnetworks** whose outputs are combined via an **ensembler**.

Terminology.

Ensembles are model-agnostic, meaning a subnetwork can be as complex as deep neural network, or as simple as an if-statement. All that matters is that for a given input tensor, the subnetworks' outputs can be combined by the ensembler to form a single prediction.

1.2 Adaptive architecture search

In the animation above, the AdaNet algorithm iteratively performs the following architecture search to grow an ensemble of subnetworks:

1. Generates a pool of candidate subnetworks.
2. Trains the subnetworks in whatever manner the user defines.
3. Evaluates the performance of the subnetworks as part of the ensemble, which is an ensemble of one at the first iteration.
4. Adds the subnetwork that most improves the ensemble performance to the ensemble for the next iteration.
5. Prunes the other subnetworks from the graph.
6. Adapts the subnetwork search space according to the information gained from the current iteration.

7. Moves onto the next iteration.
8. Repeats.

1.3 Iteration lifecycle

Each AdaNet **iteration** has the given lifecycle:

AdaNet iteration lifecucle

Each of these concepts has an associated Python object:

- **Subnetwork Generator** and **Subnetwork** are defined in the `adanet.subnetwork` package.
- **Ensemble Strategy**, **Ensembler**, and **Ensemble** are defined in the `adanet.ensemble` package.

1.4 Design

AdaNet is designed to operate primarily inside of TensorFlow's computation graph. This allows it to efficiently utilize available resources like distributed compute, GPU, and TPU, using TensorFlow primitives.

AdaNet provides a unique adaptive computation graph, which can support building models that create and remove ops and variables over time, but still have the optimizations and scalability of TensorFlow's graph-mode. This adaptive graph enables users to develop progressively growing models (e.g. boosting style), develop architecture search algorithms, and perform hyper-parameter tuning without needing to manage an external for-loop.

1.5 Example ensembles

Below are a few more examples of ensembles you can obtain with AdaNet depending on the **search space** you define. First, there is an ensemble composed of increasingly complex neural network subnetworks whose outputs are simply averaged:

Ensemble of subnetworks with different complexities.

Another common example is an ensemble learned on top of a shared embedding. Useful when the majority of the model parameters are an embedding of a feature. The individual subnetworks' predictions are combined using a learned linear combination:

Subnetworks sharing a common embedding.

1.6 Quick start

Now that you are familiar with AdaNet, you can explore our *quick start guide*.

CHAPTER 2

Quick start

If you are already using `tf.estimator.Estimator`, the fastest way to get up and running with AdaNet is to use the `adanet.AutoEnsembleEstimator`. This estimator will automatically convert a list of estimators into subnetworks, and learn to ensemble them for you.

2.1 Import AdaNet

The first step is to import the `adanet` package:

```
import adanet
```

2.2 AutoEnsembleEstimator

Next you will want to define which estimators you want to ensemble. For example, if you don't know if the best model a linear model, or a neural network, or some combination, then you can try using `tf.estimator.LinearEstimator` and `tf.estimator.DNNEstimator` as subnetworks:

```
import adanet
import tensorflow as tf

# Define the model head for computing loss and evaluation metrics.
head = tf.contrib.estimator.multi_class_head(n_classes=10)

# Feature columns define how to process examples.
feature_columns = ...

# Learn to ensemble linear and neural network models.
estimator = adanet.AutoEnsembleEstimator(
    head=head,
    candidate_pool={
```

(continues on next page)

(continued from previous page)

```
        "linear":
            tf.estimator.LinearEstimator(
                head=head,
                feature_columns=feature_columns,
                optimizer=tf.train.FtrlOptimizer(...)),
        "dnn":
            tf.estimator.DNNEstimator(
                head=head,
                feature_columns=feature_columns,
                optimizer=tf.train.ProximalAdagradOptimizer(...),
                hidden_units=[1000, 500, 100])},
    max_iteration_steps=100)

estimator.train(input_fn=train_input_fn, steps=100)
metrics = estimator.evaluate(input_fn=eval_input_fn)
predictions = estimator.predict(input_fn=predict_input_fn)
```

The above code will train both the `linear` and `dnn` subnetworks in parallel, and will average their predictions. After `max_iteration_steps=100` steps, the best subnetwork will compose the ensemble according to its performance on the *training set*.

2.3 Ensemble strategies

The way AdaNet chooses which subnetworks to include in a candidate ensemble is via **ensemble strategies**.

2.3.1 Grow strategy

The default ensemble strategy is `adanet.ensemble.GrowStrategy` which will only select the subnetwork that most improved the ensemble's performance. The remaining subnetworks will be pruned from the graph.

2.3.2 All strategy

Suppose instead of only selecting the *single best* subnetwork, you want to ensemble *all* of the subnetworks, regardless of their individual performance. You can pass an instance of the `adanet.ensemble.AllStrategy` to the `adanet.AutoEnsembleEstimator` constructor:

```
estimator = adanet.AutoEnsembleEstimator(
    [...]
    ensemble_strategies=[adanet.ensemble.AllStrategy()]
    candidate_pool={
        "linear": ...,
        "dnn": ...,
    },
    [...])
```

2.4 Tutorials

To play with AdaNet in Colab notebooks, and learn about more advanced features like customizing AdaNet and training on TPU, see our *tutorials section*.

CHAPTER 3

Tutorials

Play with AdaNet in our interactive Colab notebooks available on GitHub.

To learn more, please visit our *quick start guide*.

For more about the underlying algorithm, see the *algorithm* and *theory* pages.

CHAPTER 4

TensorBoard

TensorBoard is AdaNet's UI.

From TensorBoard, you can vizualize the performance of candidate ensembles and individual subnetworks over time, visualize their architectures, and monitor statics.

CHAPTER 5

Distributed training

AdaNet uses the same distributed training model as `tf.estimator.Estimator`.

For training TensorFlow estimators on Google Cloud ML Engine, please see this guide.

5.1 Placement Strategies

Given a cluster of worker and parameter servers, AdaNet will manage distributed training automatically. When creating an AdaNet `Estimator`, you can specify the `adanet.distributed.PlacementStrategy` to decide which subnetworks each worker will be responsible for training.

5.1.1 Replication Strategy

The default distributed training strategy is the same as the default `tf.estimator.Estimator` model: each worker will create the full training graph, including all subnetworks and ensembles, and optimize all the trainable parameters. Each variable will be randomly allocated to a parameter server to minimize bottlenecks in workers fetching them. Worker's updates will be sent to the parameter servers which apply the updates to their managed variables.

Replication strategy

To learn more, see the implementation at `adanet.distributed.ReplicationStrategy`.

5.1.2 Round Robin Stategy (experimental)

A strategy that scales better than the Replication Strategy is the experimental Round Robin Stategy. Instead of replicating the same graph on each worker, AdaNet will round robin assign workers to train a single subnetwork.

Round robin strategy

To learn more, see the implementation at `adanet.distributed.RoundRobinStrategy`.

CHAPTER 6

TPU

AdaNet officially supports TPU training, evaluation, and prediction via the `adanet.TPUEstimator`.

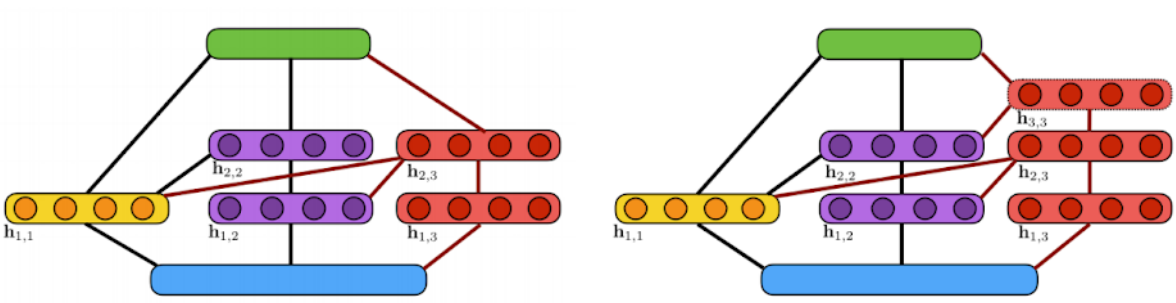
To get started, see our Colab notebook on TPU.

CHAPTER 7

Algorithm

7.1 Neural architecture search

AutoML is a family of techniques and algorithms seeking to automatically solve supervised learning tasks. Recently, researchers in AutoML have investigated whether we can automate learning the structure of a neural network for a given dataset, automating a task that requires significant domain expertise. This subdomain known as neural architecture search has seen advances in the state-of-the-art using reinforcement learning [Zoph et al. '17], evolutionary strategies [Real et al., '17], and gradient-based methods [Liu et al., '18] to learn neural network substructures. However, in these papers, the high-level structure of the network generally remains user defined.



Two candidate ensembles

This illustration shows the algorithm's incremental construction of a fully-connected neural network. The input layer is indicated in blue, the output layer in green. Units in the yellow block are added at the first iteration while units in purple are added at the second iteration. Two candidate extensions of the architecture are considered at the third iteration (shown in red): (a) a two-layer extension; (b) a three-layer extension. Here, a line between two blocks of units indicates that these blocks are fully-connected.

7.2 Neural networks are ensembles

Ensembles of neural networks have shown remarkable performance in domains such as natural language processing, image recognition, and many others. The two composing techniques are interesting in their own rights: ensemble techniques have a rich history and theoretical understanding, while neural networks provide a general framework for solving complex tasks across many domains at scale.

Coincidentally, an ensemble of neural networks whose outputs are linearly combined is also a neural network. With that definition in mind, we seek to answer the question: Can we learn a neural network architecture as an ensemble of subnetworks? And can we adaptively learn such an ensemble with fewer trainable parameters and that performs better than any single neural network trained end-to-end?

7.3 Adaptive architecture search

Our algorithm for performing adaptive neural architecture search is AdaNet [Cortes et al., ICML '17], which iteratively grows an ensemble of neural networks while providing learning guarantees. It is *adaptive* because at each iteration the candidate subnetworks are generated and trained based on the current state of the neural network.

We show this algorithm can in fact learn a neural network (ensemble) that achieves state of the art results across several datasets. We also show how this algorithm is complementary with the neural architecture search research mentioned earlier, as it learns to combine these substructures in a principled manner to achieve these results.

7.4 The AdaNet algorithm

The AdaNet algorithm works as follows: a generator iteratively creates a set of candidate base learners to consider including in the final ensemble. How these base learners are trained is left completely up to the user, but generally they are trained to optimize some common loss function such as cross-entropy loss or mean squared error. At every iteration, the trained base learners then evaluated on their ability to minimize the AdaNet objective F, and the best one is included in the final ensemble.

$$\begin{aligned} &F\left (w \right) = \frac{1}{m} \sum_{i=0}^{N-1} \Phi \left (\sum_{j=0}^{N-1}w_jh_j(x_i), y_i \right) + \sum_{j=0}^{N-1} \left (\lambda r(h_j) + \beta \right)\left | w_j \right |\ &\text{where }w_j \text{ is the weight of model } j \text{'s contribution to the ensemble,}\ &h_j \text{ is model } j,\ &\Phi \text{ is the loss function,}\ &r(h_j) \text{ is model } j\text{'s complexity, and}\ &\lambda \text{ and } \beta \text{ are tunable hyperparameters.} \end{aligned}$$

For every iteration after the first, the generator can generate neural networks based on the current state of the ensemble. This allows AdaNet to create complex structures or use advanced techniques for training candidates so that they will most significantly improve the ensemble. For an optimization example, knowledge distillation [Hinton et al., '15] is a technique that uses a teacher network's logits as the ground-truth when computing the loss of a trainable student network, and is shown to produce students that perform better than a identical network trained without. At every iteration, we can use the current ensemble as a teacher network and the candidates as students, to obtain base learners that perform better, and significantly improve the performance of the final ensemble.

7.5 More information

- A step by step walkthrough of the AdaNet algorithm

CHAPTER 8

Theory

8.1 Focus on generalization

Generalization error is what we really want to minimize when we train a model. Most algorithms minimize generalization error indirectly by minimizing a loss function that consists of a training loss term and additional penalty terms to discourage the models away from acquiring properties that are associated with overfitting (e.g., L1 weight norms, L2 weight norms).

8.2 Rigorous trade-offs between training loss and complexity

How do we know what model properties to avoid? Currently, these usually come from practical experience or industry-accepted best practices. While this has worked well so far, we would like to minimize the generalization error in a more principled way.

AdaNet's approach is to minimize a theoretical upper bound on generalization error, proven in the DeepBoost paper [Cortes et al. '14]:

$$R(f) \leq \widehat{R}_{S, \rho}(f) + \frac{4}{\rho} \sum_{k = 1}^{l} \big | \mathbf{w} _k \big |_1 \mathfrak{R}_m(\widetilde {\cal H}_k) + \widetilde O\Big(\frac{1}{\rho} \sqrt{\frac{\log l}{m}}\Big)$$

This generalization bound allows us to make an apples-to-apples comparison between the complexities of models in an ensemble and the overall training loss – allowing us to design an algorithm that makes this trade-off in a rigorous manner.

8.3 Other key insights

- **Convex combinations can't hurt.** Given a set of already-performant and uncorrelated base learners, one can take a linear combination of them with weights that sum to 1 to obtain an ensemble that outperforms the best among those base learners. But even though this ensemble has more trainable parameters, it does not have a greater tendency to overfit.

- **De-emphasize rather than discourage complex models.** If one combines a few base learners that are each selected from a different function class (e.g., neural networks of different depths and widths), one might expect the tendency to overfit to be similar to that of an ensemble comprised of base learners selected from the union of all the function classes. Remarkably, the DeepBoost bound shows that we can actually do better, as long as the final ensemble is a weighted average of model logits where each base learner's weight is inversely proportional to the Rademacher complexity of its function class, and all the weights in the logits layer sum to 1. Additionally, at training time, we don't have to discourage the trainer from learning complex models – it is only when we consider the how much the model should contribute to the ensemble do we take the complexity of the model into account.
- **Complexity is not just about the weights.** The Rademacher complexity of a neural network does not simply depend on the number of weights or the norm of its weights – it also depends on the number of layers and how they are connected. An upper bound on the Rademacher complexity of neural networks can be expressed recursively [Cortes et al. '17], and applies to both fully-connected and convolutional neural networks, thus allowing us to compute the complexity upper-bounds of almost any neural network that can be expressed as a directed-acyclic graph of layers, including unconventional architectures such as those found by NASNet [Zoph et al. '17]. Rademacher complexity is also data-dependent, which means that the same neural network architecture can have different generalization behavior on different data sets.

8.4 AdaNet loss function

Using these insights, AdaNet seeks to minimize the generalization error more directly using this loss function:

$$\begin{align*} &F\left (w \right) = \frac{1}{m} \sum_{i=1}^{m} \Phi \left (\sum_{j=1}^{N}w_jh_j(x_i), y_i \right) + \sum_{j=1}^{N} \left (\lambda r(h_j) + \beta \right)\left | w_j \right |\ &\text{where }w_j \text{ is the weight of model } j \text{'s contribution to the ensemble,}\ &h_j \text{ is model } j,\ &\Phi \text{ is the loss function,}\ &r(h_j) \text{ is model } j\text{'s complexity, and}\ &\lambda \text{ and } \beta \text{ are tunable hyperparameters.} \end{align*}$$

By minimizing this loss function, AdaNet is able to combine base learners of different complexities in a way that generalizes better than one might expect from the total size of the base learners.

CHAPTER 9

adanet

AdaNet: Fast and flexible AutoML with learning guarantees.

9.1 Estimators

High-level APIs for training, evaluating, predicting, and serving AdaNet model.

9.1.1 AutoEnsembleEstimator

class adanet.**AutoEnsembleEstimator**(*head*, *candidate_pool*, *max_iteration_steps*, *ensemblers=None*, *ensemble_strategies=None*, *logits_fn=None*, *evaluator=None*, *metric_fn=None*, *force_grow=False*, *adanet_loss_decay=0.9*, *worker_wait_timeout_secs=7200*, *model_dir=None*, *config=None*, ***kwargs*)

Bases: adanet.core.estimator.Estimator

A tf.estimator.Estimator that learns to ensemble models.

Specifically, it learns to ensemble models from a candidate pool using the Adanet algorithm.

```
# A simple example of learning to ensemble linear and neural network
# models.

import adanet
import tensorflow as tf

feature_columns = ...

head = tf.contrib.estimator.multi_class_head(n_classes=3)

# Learn to ensemble linear and DNN models.
```

(continues on next page)

(continued from previous page)

```
estimator = adanet.AutoEnsembleEstimator(
    head=head,
    candidate_pool={
        "linear":
            tf.estimator.LinearEstimator(
                head=head,
                feature_columns=feature_columns,
                optimizer=tf.train.FtrlOptimizer(...)),
        "dnn":
            tf.estimator.DNNEstimator(
                head=head,
                feature_columns=feature_columns,
                optimizer=tf.train.ProximalAdagradOptimizer(...),
                hidden_units=[1000, 500, 100])},
    max_iteration_steps=50)

# Input builders
def input_fn_train:
  # Returns tf.data.Dataset of (x, y) tuple where y represents label's
  # class index.
  pass
def input_fn_eval:
  # Returns tf.data.Dataset of (x, y) tuple where y represents label's
  # class index.
  pass
def input_fn_predict:
  # Returns tf.data.Dataset of (x, None) tuple.
  pass
estimator.train(input_fn=input_fn_train, steps=100)
metrics = estimator.evaluate(input_fn=input_fn_eval, steps=10)
predictions = estimator.predict(input_fn=input_fn_predict)
```

Parameters

- **head** – A `tf.contrib.estimator.Head` instance for computing loss and evaluation metrics for every candidate.
- **candidate_pool** – List of `tf.estimator.Estimator` objects or dict of string name to `tf.estimator.Estimator` objects that are candidates to ensemble at each iteration. The order does not directly affect which candidates will be included in the final ensemble, but will affect the name of the candidate. When using a dict, the string key will be used as the name of the candidate.
- **max_iteration_steps** – Total number of steps for which to train candidates per iteration. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before *max_iteration_steps* steps.
- **logits_fn** – A function for fetching the subnetwork logits from a `tf.estimator.EstimatorSpec`, which should obey the following signature:
  - *Args*: Can only have following argument: - estimator_spec: The candidate's `tf.estimator.EstimatorSpec`.
  - *Returns*: Logits `tf.Tensor` or dict of string to logits `tf.Tensor` (for multi-head) for the candidate subnetwork extracted from the given *estimator_spec*. When *None*, it will default to returning *estimator_spec.predictions* when they are a `tf.Tensor` or the `tf.Tensor` for the key 'logits' when they are a dict of string to `tf.Tensor`.

- **ensemblers** – See *adanet.Estimator*.
- **ensemble_strategies** – See *adanet.Estimator*.
- **evaluator** – See *adanet.Estimator*.
- **metric_fn** – See *adanet.Estimator*.
- **force_grow** – See *adanet.Estimator*.
- **adanet_loss_decay** – See *adanet.Estimator*.
- **worker_wait_timeout_secs** – See *adanet.Estimator*.
- **model_dir** – See *adanet.Estimator*.
- **config** – See *adanet.Estimator*.
- **debug** – See *adanet.Estimator*.

Returns An *adanet.AutoEnsembleEstimator* instance.

Raises `ValueError` – If any of the candidates in *candidate_pool* are not `tf.estimator.Estimator` instances.

eval_dir(*name=None*)

Shows the directory name where evaluation metrics are dumped.

Parameters **name** – Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns A string which is the path of directory contains evaluation metrics.

evaluate(*input_fn*, *steps=None*, *hooks=None*, *checkpoint_path=None*, *name=None*)

Evaluates the model given evaluation data *input_fn*.

For each step, calls *input_fn*, which returns one batch of data. Evaluates until: - *steps* batches are processed, or - *input_fn* raises an end-of-input exception (*tf.errors.OutOfRangeError* or *StopIteration*).

Parameters

- **input_fn** – A function that constructs the input data for evaluation. See [Premade Estimators](https://tensorflow.org/guide/premade#create_input_functions) for more information. The function should construct and return one of the following: * A *tf.data.Dataset* object: Outputs of *Dataset* object must be a tuple *(features, labels)* with same constraints as below. * A tuple *(features, labels)*: Where *features* is a *tf.Tensor* or a dictionary of string feature name to *Tensor* and *labels* is a *Tensor* or a dictionary of string label name to *Tensor*. Both *features* and *labels* are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
- **steps** – Number of steps for which to evaluate model. If *None*, evaluates until *input_fn* raises an end-of-input exception.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the evaluation call.
- **checkpoint_path** – Path of a specific checkpoint to evaluate. If *None*, the latest checkpoint in *model_dir* is used. If there are no checkpoints in *model_dir*, evaluation is run with newly initialized *Variables* instead of ones restored from checkpoint.
- **name** – Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns A dict containing the evaluation metrics specified in *model_fn* keyed by name, as well as an entry *global_step* which contains the value of the global step for which this evaluation was performed. For canned estimators, the dict contains the *loss* (mean loss per mini-batch) and the *average_loss* (mean loss per sample). Canned classifiers also return the *accuracy*. Canned regressors also return the *label/mean* and the *prediction/mean*.

Raises

- `ValueError` – If *steps <= 0*.
- `ValueError` – If no model has been trained, namely *model_dir*, or the given *checkpoint_path* is empty.

experimental_export_all_saved_models(*export_dir_base*, *input_receiver_fn_map*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*)

Exports a *SavedModel* with *tf.MetaGraphDefs* for each requested mode.

For each mode passed in via the *input_receiver_fn_map*, this method builds a new graph by calling the *input_receiver_fn* to obtain feature and label *Tensor's*. Next, this method calls the 'Estimator's *model_fn* in the passed mode to generate the model graph based on those features and labels, and restores the given checkpoint (or, lacking that, the most recent checkpoint) into the graph. Only one of the modes is used for saving variables to the *SavedModel* (order of preference: *tf.estimator.ModeKeys.TRAIN*, *tf.estimator.ModeKeys.EVAL*, then *tf.estimator.ModeKeys.PREDICT*), such that up to three *tf.MetaGraphDefs* are saved with a single set of variables in a single *SavedModel* directory.

For the variables and *tf.MetaGraphDefs*, a timestamped export directory below *export_dir_base*, and writes a *SavedModel* into it containing the *tf.MetaGraphDef* for the given mode and its associated signatures.

For prediction, the exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput's, and the inputs are always the input receivers provided by the 'serving_input_receiver_fn*.

For training and evaluation, the *train_op* is stored in an extra collection, and loss, metrics, and predictions are included in a *SignatureDef* for the mode in question.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{'my_asset_file.txt': '/path/to/my_asset_file.txt'}*.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported 'SavedModel's.
- **input_receiver_fn_map** – dict of *tf.estimator.ModeKeys* to *input_receiver_fn* mappings, where the *input_receiver_fn* is a function that takes no arguments and returns the appropriate subclass of *InputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.

Returns The string path to the exported directory.

Raises `ValueError` – if any *input_receiver_fn* is *None*, no *export_outputs* are provided, or no checkpoint can be found.

export_saved_model(*export_dir_base*, *serving_input_receiver_fn*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*, *experimental_mode='infer'*)

Exports inference graph as a *SavedModel* into the given dir.

For a detailed guide, see [Using SavedModel with Estimators](https://tensorflow.org/guide/saved_model#using_savedmodel_with_estimators).

This method builds a new graph by first calling the *serving_input_receiver_fn* to obtain feature *Tensor*‘*s, and then calling this* ‘*Estimator*’s *model_fn* to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given *export_dir_base*, and writes a *SavedModel* into it containing a single *tf.MetaGraphDef* saved from this session.

The exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput*‘*s, and the inputs are always the input receivers provided by the* ‘*serving_input_receiver_fn*.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{*‘*my_asset_file.txt*’*: *‘*/path/to/my_asset_file.txt*’*}*.

The experimental_mode parameter can be used to export a single train/eval/predict graph as a *SavedModel*. See *experimental_export_all_saved_models* for full docs.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported ‘SavedModel‘s.
- **serving_input_receiver_fn** – A function that takes no argument and returns a *tf.estimator.export.ServingInputReceiver* or *tf.estimator.export.TensorServingInputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.
- **experimental_mode** – *tf.estimator.ModeKeys* value indicating with mode will be exported. Note that this feature is experimental.

Returns The string path to the exported directory.

Raises

- `ValueError` – if no *serving_input_receiver_fn* is provided, no
- *export_outputs* are provided, or no checkpoint can be found.

export_savedmodel(*export_dir_base*, *serving_input_receiver_fn*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*, *strip_default_attrs=False*)

Exports inference graph as a *SavedModel* into the given dir.

For a detailed guide, see [Using SavedModel with Estimators](https://tensorflow.org/guide/saved_model#using_savedmodel_with_estimators).

This method builds a new graph by first calling the *serving_input_receiver_fn* to obtain feature *Tensor‘s, and then calling this ‘Estimator*’s *model_fn* to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given *export_dir_base*, and writes a *SavedModel* into it containing a single *tf.MetaGraphDef* saved from this session.

The exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput‘s, and the inputs are always the input receivers provided by the ‘serving_input_receiver_fn*.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{‘my_asset_file.txt’: ‘/path/to/my_asset_file.txt’}*.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported ‘SavedModel‘s.
- **serving_input_receiver_fn** – A function that takes no argument and returns a *tf.estimator.export.ServingInputReceiver* or *tf.estimator.export.TensorServingInputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.
- **strip_default_attrs** – Boolean. If *True*, default-valued attributes will be removed from the ‘NodeDef‘s. For a detailed guide, see [Stripping Default-Valued Attributes](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/README.md#stripping-default-valued-attributes).

Returns The string path to the exported directory.

Raises

- `ValueError` – if no *serving_input_receiver_fn* is provided, no
- *export_outputs* are provided, or no checkpoint can be found.

get_variable_names()

Returns list of all variable names in this model.

Returns List of names.

Raises `ValueError` – If the *Estimator* has not produced a checkpoint yet.

get_variable_value(*name*)

Returns value of the variable given by name.

Parameters **name** – string or a list of string, name of the tensor.

Returns Numpy array - value of the tensor.

Raises `ValueError` – If the *Estimator* has not produced a checkpoint yet.

latest_checkpoint()
Finds the filename of the latest saved checkpoint file in *model_dir*.

Returns The full path to the latest checkpoint or *None* if no checkpoint was found.

model_fn
Returns the *model_fn* which is bound to *self.params*.

Returns *def model_fn(features, labels, mode, config)*

Return type The *model_fn* with following signature

predict(*input_fn*, *predict_keys=None*, *hooks=None*, *checkpoint_path=None*, *yield_single_examples=True*)
Yields predictions for given features.

Please note that interleaving two predict outputs does not work. See: [issue/20506](https://github.com/tensorflow/tensorflow/issues/20506#issuecomment-422208517)

Parameters

- **input_fn** – A function that constructs the features. Prediction continues until *input_fn* raises an end-of-input exception (*tf.errors.OutOfRangeError* or *StopIteration*). See [Premade Estimators](https://tensorflow.org/guide/premade_estimators#create_input_functions) for more information. The function should construct and return one of the following:
  - A *tf.data.Dataset* object: Outputs of *Dataset* object must have same constraints as below.
  - features: A *tf.Tensor* or a dictionary of string feature name to *Tensor*. features are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
  - A tuple, in which case the first item is extracted as features.
- **predict_keys** – list of *str*, name of the keys to predict. It is used if the *tf.estimator.EstimatorSpec.predictions* is a *dict*. If *predict_keys* is used then rest of the predictions will be filtered from the dictionary. If *None*, returns all.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the prediction call.
- **checkpoint_path** – Path of a specific checkpoint to predict. If *None*, the latest checkpoint in *model_dir* is used. If there are no checkpoints in *model_dir*, prediction is run with newly initialized *Variables* instead of ones restored from checkpoint.
- **yield_single_examples** – If *False*, yields the whole batch as returned by the *model_fn* instead of decomposing the batch into individual elements. This is useful if *model_fn* returns some tensors whose first dimension is not equal to the batch size.

Yields Evaluated values of *predictions* tensors.

Raises

- `ValueError` – Could not find a trained model in *model_dir*.
- `ValueError` – If batch length of predictions is not the same and *yield_single_examples* is *True*.
- `ValueError` – If there is a conflict between *predict_keys* and *predictions*. For example if *predict_keys* is not *None* but *tf.estimator.EstimatorSpec.predictions* is not a *dict*.

train(*input_fn*, *hooks=None*, *steps=None*, *max_steps=None*, *saving_listeners=None*)
Trains a model given training data *input_fn*.

Parameters

- **input_fn** – A function that provides input data for training as minibatches. See [Premade Estimators](https://tensorflow.org/guide/premade_estimators#create_input_functions) for more information. The function should construct and return one of the following: * A *tf.data.Dataset* object: Outputs of *Dataset* object must be a tuple *(features, labels)* with same constraints as below. * A tuple *(features, labels)*: Where *features* is a *tf.Tensor* or a dictionary of string feature name to *Tensor* and *labels* is a *Tensor* or a dictionary of string label name to *Tensor*. Both *features* and *labels* are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the training loop.
- **steps** – Number of steps for which to train the model. If *None*, train forever or train until *input_fn* generates the *tf.errors.OutOfRange* error or *StopIteration* exception. *steps* works incrementally. If you call two times *train(steps=10)* then training occurs in total 20 steps. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before 20 steps. If you don't want to have incremental behavior please set *max_steps* instead. If set, *max_steps* must be *None*.
- **max_steps** – Number of total steps for which to train model. If *None*, train forever or train until *input_fn* generates the *tf.errors.OutOfRange* error or *StopIteration* exception. If set, *steps* must be *None*. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before *max_steps* steps. Two calls to *train(steps=100)* means 200 training iterations. On the other hand, two calls to *train(max_steps=100)* means that the second call will not do any iteration since first call did all 100 steps.
- **saving_listeners** – list of *CheckpointSaverListener* objects. Used for callbacks that run immediately before or after checkpoint savings.

Returns *self*, for chaining.

Raises

- `ValueError` – If both *steps* and *max_steps* are not *None*.
- `ValueError` – If either *steps* or *max_steps* <= *0*.

9.1.2 Estimator

class adanet.**Estimator**(*head*, *subnetwork_generator*, *max_iteration_steps*, *ensemblers=None*, *ensemble_strategies=None*, *evaluator=None*, *report_materializer=None*, *metric_fn=None*, *force_grow=False*, *replicate_ensemble_in_training=False*, *adanet_loss_decay=0.9*, *delay_secs_per_worker=5*, *max_worker_delay_secs=60*, *worker_wait_secs=5*, *worker_wait_timeout_secs=7200*, *model_dir=None*, *report_dir=None*, *config=None*, *debug=False*, ***kwargs*)
Bases: `tensorflow_estimator.python.estimator.estimator.Estimator`

A `tf.estimator.Estimator` for training, evaluation, and serving.

This implementation uses an *adanet.subnetwork.Generator* as its weak learning algorithm for generating candidate subnetworks. These are trained in parallel using a single graph per iteration. At the end of each iteration, the estimator saves the sub-graph of the best subnetwork ensemble and its weights as a separate checkpoint. At the beginning of the next iteration, the estimator imports the previous iteration's frozen graph and adds

ops for the next candidates as part of a new graph and session. This allows the estimator have the performance of Tensorflow's static graph constraint (minus the performance hit of reconstructing a graph between iterations), while having the flexibility of having a dynamic graph.

NOTE: Subclassing `tf.estimator.Estimator` is only necessary to work with `tf.estimator.train_and_evaluate()` which asserts that the estimator argument is a `tf.estimator.Estimator` subclass. However, all training is delegated to a separate `tf.estimator.Estimator` instance. It is responsible for supporting both local and distributed training. As such, the *adanet.Estimator* is only responsible for bookkeeping across iterations.

Parameters

- **head** – A `tf.contrib.estimator.Head` instance for computing loss and evaluation metrics for every candidate.
- **subnetwork_generator** – The *adanet.subnetwork.Generator* which defines the candidate subnetworks to train and evaluate at every AdaNet iteration.
- **max_iteration_steps** – Total number of steps for which to train candidates per iteration. If `OutOfRange` or `StopIteration` occurs in the middle, training stops before *max_iteration_steps* steps.
- **ensemblers** – An iterable of *adanet.ensemble.Ensembler* objects that define how to ensemble a group of subnetworks.
- **ensemble_strategies** – An iterable of *adanet.ensemble.Strategy* objects that define the candidate ensembles of subnetworks to explore at each iteration.
- **evaluator** – An *adanet.Evaluator* for candidate selection after all subnetworks are done training. When `None`, candidate selection uses a moving average of their `adanet.Ensemble` AdaNet loss during training instead. In order to use the *AdaNet algorithm* as described in [Cortes et al., '17], the given *adanet.Evaluator* must be created with the same dataset partition used during training. Otherwise, this framework will perform *AdaNet.HoldOut* which uses a holdout set for candidate selection, but does not benefit from learning guarantees.
- **report_materializer** – An *adanet.ReportMaterializer*. Its reports are made available to the *subnetwork_generator* at the next iteration, so that it can adapt its search space. When *None*, the *subnetwork_generator* `generate_candidates()` method will receive empty Lists for their *previous_ensemble_reports* and *all_reports* arguments.
- **metric_fn** – A function for adding custom evaluation metrics, which should obey the following signature:
  - *Args*: Can only have the following three arguments in any order: - `predictions`: Predictions *Tensor* or dict of *Tensor*

    created by given `head`.

    * `features`: Input *dict* of *Tensor* objects created by `input_fn` which is given to `estimator.evaluate()` as an argument.
    * `labels`: Labels *Tensor* or dict of *Tensor* (for multi-head) created by `input_fn` which is given to `estimator.evaluate()` as an argument.

  - *Returns*: Dict of metric results keyed by name. Final metrics are a union of this and `head`'s existing metrics. If there is a name conflict between this and `head`s existing metrics, this will override the existing one.

The values of the dict are the results of calling a metric function, namely a :code:`(metric_tensor, update_op)` tuple.

- **force_grow** – Boolean override that forces the ensemble to grow by one subnetwork at the end of each iteration. Normally at the end of each iteration, AdaNet selects the best candidate ensemble according to its performance on the AdaNet objective. In some cases, the best ensemble is the *previous_ensemble* as opposed to one that includes a newly trained subnetwork. When *True*, the algorithm will not select the *previous_ensemble* as the best candidate, and will ensure that after n iterations the final ensemble is composed of n subnetworks.
- **replicate_ensemble_in_training** – Whether to rebuild the frozen subnetworks of the ensemble in training mode, which can change the outputs of the frozen subnetworks in the ensemble. When *False* and during candidate training, the frozen subnetworks in the ensemble are in prediction mode, so training-only ops like dropout are not applied to them. When *True* and training the candidates, the frozen subnetworks will be in training mode as well, so they will apply training-only ops like dropout. This argument is useful for regularizing learning mixture weights, or for making training-only side inputs available in subsequent iterations. For most use-cases, this should be *False*.
- **adanet_loss_decay** – Float decay for the exponential-moving-average of the AdaNet objective throughout training. This moving average is a data- driven way tracking the best candidate with only the training set.
- **delay_secs_per_worker** – Float number of seconds to delay starting the i-th worker. Staggering worker start-up during distributed asynchronous SGD can improve training stability and speed up convergence. Each worker will wait (i+1) * delay_secs_per_worker seconds before beginning training.
- **max_worker_delay_secs** – Float max number of seconds to delay starting the i-th worker. Staggering worker start-up during distributed asynchronous SGD can improve training stability and speed up convergence. Each worker will wait up to max_worker_delay_secs before beginning training.
- **worker_wait_secs** – Float number of seconds for workers to wait before checking if the chief prepared the next iteration.
- **worker_wait_timeout_secs** – Float number of seconds for workers to wait for chief to prepare the next iteration during distributed training. This is needed to prevent workers waiting indefinitely for a chief that may have crashed or been turned down. When the timeout is exceeded, the worker exits the train loop. In situations where the chief job is much slower than the worker jobs, this timeout should be increased.
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **report_dir** – Directory where the `adanet.subnetwork.MaterializedReport`s materialized by :code:`report_materializer` would be saved. If `report_materializer` is `None`, this will not save anything. If `None` or empty string, defaults to `<model_dir>/report`.
- **config** – `RunConfig` object to configure the runtime settings.
- **debug** – Boolean to enable debug mode which will check features and labels for Infs and NaNs.
- ****kwargs** – Extra keyword args passed to the parent.

Returns An *adanet.Estimator* instance.

Raises

- `ValueError` – If `ensemblers` is size > 1.
- `ValueError` – If `subnetwork_generator` is `None`.
- `ValueError` – If `max_iteration_steps` is <= 0.
- `ValueError` – If `model_dir` is not specified during distributed training.

eval_dir(*name=None*)

Shows the directory name where evaluation metrics are dumped.

Parameters **name** – Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns A string which is the path of directory contains evaluation metrics.

evaluate(*input_fn*, *steps=None*, *hooks=None*, *checkpoint_path=None*, *name=None*)

Evaluates the model given evaluation data *input_fn*.

For each step, calls *input_fn*, which returns one batch of data. Evaluates until: - *steps* batches are processed, or - *input_fn* raises an end-of-input exception (*tf.errors.OutOfRangeError* or *StopIteration*).

Parameters

- **input_fn** – A function that constructs the input data for evaluation. See [Premade Estimators](https://tensorflow.org/guide/premade#create_input_functions) for more information. The function should construct and return one of the following: * A *tf.data.Dataset* object: Outputs of *Dataset* object must be a tuple *(features, labels)* with same constraints as below. * A tuple *(features, labels)*: Where *features* is a *tf.Tensor* or a dictionary of string feature name to *Tensor* and *labels* is a *Tensor* or a dictionary of string label name to *Tensor*. Both *features* and *labels* are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
- **steps** – Number of steps for which to evaluate model. If *None*, evaluates until *input_fn* raises an end-of-input exception.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the evaluation call.
- **checkpoint_path** – Path of a specific checkpoint to evaluate. If *None*, the latest checkpoint in *model_dir* is used. If there are no checkpoints in *model_dir*, evaluation is run with newly initialized *Variables* instead of ones restored from checkpoint.
- **name** – Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns A dict containing the evaluation metrics specified in *model_fn* keyed by name, as well as an entry *global_step* which contains the value of the global step for which this evaluation was performed. For canned estimators, the dict contains the *loss* (mean loss per mini-batch) and the *average_loss* (mean loss per sample). Canned classifiers also return the *accuracy*. Canned regressors also return the *label/mean* and the *prediction/mean*.

Raises

- `ValueError` – If *steps* <= *0*.
- `ValueError` – If no model has been trained, namely *model_dir*, or the given *checkpoint_path* is empty.

experimental_export_all_saved_models(*export_dir_base*, *input_receiver_fn_map*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*)

Exports a *SavedModel* with *tf.MetaGraphDefs* for each requested mode.

For each mode passed in via the *input_receiver_fn_map*, this method builds a new graph by calling the *input_receiver_fn* to obtain feature and label *Tensor's.* Next, this method calls the 'Estimator's *model_fn* in the passed mode to generate the model graph based on those features and labels, and restores the given checkpoint (or, lacking that, the most recent checkpoint) into the graph. Only one of the modes is used for saving variables to the *SavedModel* (order of preference: *tf.estimator.ModeKeys.TRAIN*, *tf.estimator.ModeKeys.EVAL*, then *tf.estimator.ModeKeys.PREDICT*), such that up to three *tf.MetaGraphDefs* are saved with a single set of variables in a single *SavedModel* directory.

For the variables and *tf.MetaGraphDefs*, a timestamped export directory below *export_dir_base*, and writes a *SavedModel* into it containing the *tf.MetaGraphDef* for the given mode and its associated signatures.

For prediction, the exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput's, and the inputs are always the input receivers provided by the 'serving_input_receiver_fn.*

For training and evaluation, the *train_op* is stored in an extra collection, and loss, metrics, and predictions are included in a *SignatureDef* for the mode in question.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{'my_asset_file.txt': '/path/to/my_asset_file.txt'}*.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported 'SavedModel's.
- **input_receiver_fn_map** – dict of *tf.estimator.ModeKeys* to *input_receiver_fn* mappings, where the *input_receiver_fn* is a function that takes no arguments and returns the appropriate subclass of *InputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.

Returns The string path to the exported directory.

Raises `ValueError` – if any *input_receiver_fn* is *None*, no *export_outputs* are provided, or no checkpoint can be found.

export_saved_model(*export_dir_base*, *serving_input_receiver_fn*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*, *experimental_mode='infer'*)

Exports inference graph as a *SavedModel* into the given dir.

For a detailed guide, see [Using SavedModel with Estimators](https://tensorflow.org/guide/saved_model#using_savedmodel_with_estimators).

This method builds a new graph by first calling the *serving_input_receiver_fn* to obtain feature *Tensor's, and then calling this 'Estimator*'s *model_fn* to generate the model graph based on those features. It restores

the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given *export_dir_base*, and writes a *SavedModel* into it containing a single *tf.MetaGraphDef* saved from this session.

The exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput's, and the inputs are always the input receivers provided by the 'serving_input_receiver_fn*.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{'my_asset_file.txt': '/path/to/my_asset_file.txt'}*.

The experimental_mode parameter can be used to export a single train/eval/predict graph as a *SavedModel*. See *experimental_export_all_saved_models* for full docs.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported 'SavedModel's.
- **serving_input_receiver_fn** – A function that takes no argument and returns a *tf.estimator.export.ServingInputReceiver* or *tf.estimator.export.TensorServingInputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.
- **experimental_mode** – *tf.estimator.ModeKeys* value indicating with mode will be exported. Note that this feature is experimental.

Returns The string path to the exported directory.

Raises

- `ValueError` – if no *serving_input_receiver_fn* is provided, no
- *export_outputs* are provided, or no checkpoint can be found.

export_savedmodel(*export_dir_base*, *serving_input_receiver_fn*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*, *strip_default_attrs=False*)

Exports inference graph as a *SavedModel* into the given dir.

For a detailed guide, see [Using SavedModel with Estimators](https://tensorflow.org/guide/saved_model#using_savedmodel_with_estimators).

This method builds a new graph by first calling the *serving_input_receiver_fn* to obtain feature *Tensor's, and then calling this 'Estimator*'s *model_fn* to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given *export_dir_base*, and writes a *SavedModel* into it containing a single *tf.MetaGraphDef* saved from this session.

The exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating

which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput's, and the inputs are always the input receivers provided by the 'serving_input_receiver_fn*.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{'my_asset_file.txt': '/path/to/my_asset_file.txt'}*.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported 'SavedModel's.
- **serving_input_receiver_fn** – A function that takes no argument and returns a *tf.estimator.export.ServingInputReceiver* or *tf.estimator.export.TensorServingInputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.
- **strip_default_attrs** – Boolean. If *True*, default-valued attributes will be removed from the 'NodeDef's. For a detailed guide, see [Stripping Default-Valued Attributes](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/README.md#stripping-default-valued-attributes).

Returns The string path to the exported directory.

Raises

- `ValueError` – if no *serving_input_receiver_fn* is provided, no
- *export_outputs* are provided, or no checkpoint can be found.

get_variable_names()
Returns list of all variable names in this model.

Returns List of names.

Raises `ValueError` – If the *Estimator* has not produced a checkpoint yet.

get_variable_value(*name*)
Returns value of the variable given by name.

Parameters **name** – string or a list of string, name of the tensor.

Returns Numpy array - value of the tensor.

Raises `ValueError` – If the *Estimator* has not produced a checkpoint yet.

latest_checkpoint()
Finds the filename of the latest saved checkpoint file in *model_dir*.

Returns The full path to the latest checkpoint or *None* if no checkpoint was found.

model_fn
Returns the *model_fn* which is bound to *self.params*.

Returns *def model_fn(features, labels, mode, config)*

Return type The *model_fn* with following signature

predict(*input_fn*, *predict_keys=None*, *hooks=None*, *checkpoint_path=None*, *yield_single_examples=True*)
Yields predictions for given features.

Please note that interleaving two predict outputs does not work. See: [issue/20506](https://github.com/tensorflow/tensorflow/issues/20506#issuecomment-422208517)

Parameters

- **input_fn** – A function that constructs the features. Prediction continues until *input_fn* raises an end-of-input exception (*tf.errors.OutOfRangeError* or *StopIteration*). See [Premade Estimators](https://tensorflow.org/guide/premade_estimators#create_input_functions) for more information. The function should construct and return one of the following:
  - A *tf.data.Dataset* object: Outputs of *Dataset* object must have same constraints as below.
  - features: A *tf.Tensor* or a dictionary of string feature name to *Tensor*. features are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
  - A tuple, in which case the first item is extracted as features.
- **predict_keys** – list of *str*, name of the keys to predict. It is used if the *tf.estimator.EstimatorSpec.predictions* is a *dict*. If *predict_keys* is used then rest of the predictions will be filtered from the dictionary. If *None*, returns all.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the prediction call.
- **checkpoint_path** – Path of a specific checkpoint to predict. If *None*, the latest checkpoint in *model_dir* is used. If there are no checkpoints in *model_dir*, prediction is run with newly initialized *Variables* instead of ones restored from checkpoint.
- **yield_single_examples** – If *False*, yields the whole batch as returned by the *model_fn* instead of decomposing the batch into individual elements. This is useful if *model_fn* returns some tensors whose first dimension is not equal to the batch size.

Yields Evaluated values of *predictions* tensors.

Raises

- `ValueError` – Could not find a trained model in *model_dir*.
- `ValueError` – If batch length of predictions is not the same and *yield_single_examples* is *True*.
- `ValueError` – If there is a conflict between *predict_keys* and *predictions*. For example if *predict_keys* is not *None* but *tf.estimator.EstimatorSpec.predictions* is not a *dict*.

train(*input_fn*, *hooks=None*, *steps=None*, *max_steps=None*, *saving_listeners=None*)
Trains a model given training data *input_fn*.

Parameters

- **input_fn** – A function that provides input data for training as minibatches. See [Premade Estimators](https://tensorflow.org/guide/premade_estimators#create_input_functions) for more information. The function should construct and return one of the following: * A *tf.data.Dataset* object: Outputs of *Dataset* object must be a tuple *(features, labels)* with same constraints as below. * A tuple *(features, labels)*: Where *features* is a *tf.Tensor* or a dictionary of string feature name to *Tensor* and *labels* is a *Tensor* or a dictionary of string label name to *Tensor*. Both *features* and *labels* are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.

- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the training loop.
- **steps** – Number of steps for which to train the model. If *None*, train forever or train until *input_fn* generates the *tf.errors.OutOfRange* error or *StopIteration* exception. *steps* works incrementally. If you call two times *train(steps=10)* then training occurs in total 20 steps. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before 20 steps. If you don't want to have incremental behavior please set *max_steps* instead. If set, *max_steps* must be *None*.
- **max_steps** – Number of total steps for which to train model. If *None*, train forever or train until *input_fn* generates the *tf.errors.OutOfRange* error or *StopIteration* exception. If set, *steps* must be *None*. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before *max_steps* steps. Two calls to *train(steps=100)* means 200 training iterations. On the other hand, two calls to *train(max_steps=100)* means that the second call will not do any iteration since first call did all 100 steps.
- **saving_listeners** – list of *CheckpointSaverListener* objects. Used for callbacks that run immediately before or after checkpoint savings.

Returns *self*, for chaining.

Raises

- `ValueError` – If both *steps* and *max_steps* are not *None*.
- `ValueError` – If either *steps* or *max_steps* <= *0*.

9.1.3 TPUEstimator

class adanet.**TPUEstimator**(*head*, *subnetwork_generator*, *max_iteration_steps*, *ensemblers=None*, *ensemble_strategies=None*, *evaluator=None*, *report_materializer=None*, *metric_fn=None*, *force_grow=False*, *replicate_ensemble_in_training=False*, *adanet_loss_decay=0.9*, *model_dir=None*, *report_dir=None*, *config=None*, *use_tpu=True*, *train_batch_size=None*, *eval_batch_size=None*, *debug=False*, ***kwargs*)

Bases: `adanet.core.estimator.Estimator`, `tensorflow.contrib.tpu.python.tpu.tpu_estimator.TPUEstimator`

An *adanet.Estimator* capable of training and evaluating on TPU.

Note: Unless `use_tpu=False`, training will run on TPU. However, certain parts of AdaNet training loop, such as report materialization and best candidate selection still occurr on CPU. Furthermore, inference also occurs on CPU.

Parameters

- **head** – See *adanet.Estimator*.
- **subnetwork_generator** – See *adanet.Estimator*.
- **max_iteration_steps** – See *adanet.Estimator*.
- **ensemblers** – See *adanet.Estimator*.
- **ensemble_strategies** – See *adanet.Estimator*.
- **evaluator** – See *adanet.Estimator*.
- **report_materializer** – See *adanet.Estimator*.

- **metric_fn** – See *adanet.Estimator*.
- **force_grow** – See *adanet.Estimator*.
- **replicate_ensemble_in_training** – See *adanet.Estimator*.
- **adanet_loss_decay** – See *adanet.Estimator*.
- **report_dir** – See *adanet.Estimator*.
- **config** – See *adanet.Estimator*.
- **use_tpu** – Boolean to enable *both* training and evaluating on TPU. Defaults to `True` and is only provided to allow debugging models on CPU/GPU. Use *adanet.Estimator* instead if you do not plan to run on TPU.
- **train_batch_size** – See `tf.contrib.tpu.TPUEstimator`.
- **eval_batch_size** – See `tf.contrib.tpu.TPUEstimator`.
- **debug** – See *adanet.Estimator*.
- ****kwargs** – Extra keyword args passed to the parent.

eval_dir(*name=None*)

Shows the directory name where evaluation metrics are dumped.

Parameters **name** – Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns A string which is the path of directory contains evaluation metrics.

evaluate(*input_fn*, *steps=None*, *hooks=None*, *checkpoint_path=None*, *name=None*)

Evaluates the model given evaluation data *input_fn*.

For each step, calls *input_fn*, which returns one batch of data. Evaluates until: - *steps* batches are processed, or - *input_fn* raises an end-of-input exception (*tf.errors.OutOfRangeError* or *StopIteration*).

Parameters

- **input_fn** – A function that constructs the input data for evaluation. See [Premade Estimators](https://tensorflow.org/guide/premade#create_input_functions) for more information. The function should construct and return one of the following: * A *tf.data.Dataset* object: Outputs of *Dataset* object must be a tuple *(features, labels)* with same constraints as below. * A tuple *(features, labels)*: Where *features* is a *tf.Tensor* or a dictionary of string feature name to *Tensor* and *labels* is a *Tensor* or a dictionary of string label name to *Tensor*. Both *features* and *labels* are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
- **steps** – Number of steps for which to evaluate model. If *None*, evaluates until *input_fn* raises an end-of-input exception.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the evaluation call.
- **checkpoint_path** – Path of a specific checkpoint to evaluate. If *None*, the latest checkpoint in *model_dir* is used. If there are no checkpoints in *model_dir*, evaluation is run with newly initialized *Variables* instead of ones restored from checkpoint.
- **name** – Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

Returns A dict containing the evaluation metrics specified in *model_fn* keyed by name, as well as an entry *global_step* which contains the value of the global step for which this evaluation was performed. For canned estimators, the dict contains the *loss* (mean loss per mini-batch) and the *average_loss* (mean loss per sample). Canned classifiers also return the *accuracy*. Canned regressors also return the *label/mean* and the *prediction/mean*.

Raises

- `ValueError` – If *steps <= 0*.
- `ValueError` – If no model has been trained, namely *model_dir*, or the given *checkpoint_path* is empty.

experimental_export_all_saved_models(*export_dir_base*, *input_receiver_fn_map*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*)

Exports a *SavedModel* with *tf.MetaGraphDefs* for each requested mode.

For each mode passed in via the *input_receiver_fn_map*, this method builds a new graph by calling the *input_receiver_fn* to obtain feature and label *Tensor‘s. Next, this method calls the ‘Estimator*’s *model_fn* in the passed mode to generate the model graph based on those features and labels, and restores the given checkpoint (or, lacking that, the most recent checkpoint) into the graph. Only one of the modes is used for saving variables to the *SavedModel* (order of preference: *tf.estimator.ModeKeys.TRAIN*, *tf.estimator.ModeKeys.EVAL*, then *tf.estimator.ModeKeys.PREDICT*), such that up to three *tf.MetaGraphDefs* are saved with a single set of variables in a single *SavedModel* directory.

For the variables and *tf.MetaGraphDefs*, a timestamped export directory below *export_dir_base*, and writes a *SavedModel* into it containing the *tf.MetaGraphDef* for the given mode and its associated signatures.

For prediction, the exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput‘s, and the inputs are always the input receivers provided by the ‘serving_input_receiver_fn*.

For training and evaluation, the *train_op* is stored in an extra collection, and loss, metrics, and predictions are included in a *SignatureDef* for the mode in question.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{‘my_asset_file.txt’: ‘/path/to/my_asset_file.txt’}*.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported ‘SavedModel‘s.
- **input_receiver_fn_map** – dict of *tf.estimator.ModeKeys* to *input_receiver_fn* mappings, where the *input_receiver_fn* is a function that takes no arguments and returns the appropriate subclass of *InputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.

Returns The string path to the exported directory.

Raises `ValueError` – if any *input_receiver_fn* is *None*, no *export_outputs* are provided, or no checkpoint can be found.

export_saved_model(*export_dir_base*, *serving_input_receiver_fn*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*, *experimental_mode='infer'*)

Exports inference graph as a *SavedModel* into the given dir.

For a detailed guide, see [Using SavedModel with Estimators](https://tensorflow.org/guide/saved_model#using_savedmodel_with_estimators).

This method builds a new graph by first calling the *serving_input_receiver_fn* to obtain feature *Tensor's, and then calling this 'Estimator*'s *model_fn* to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given *export_dir_base*, and writes a *SavedModel* into it containing a single *tf.MetaGraphDef* saved from this session.

The exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput's, and the inputs are always the input receivers provided by the 'serving_input_receiver_fn*.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{'my_asset_file.txt': '/path/to/my_asset_file.txt'}*.

The experimental_mode parameter can be used to export a single train/eval/predict graph as a *SavedModel*. See *experimental_export_all_saved_models* for full docs.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported 'SavedModel's.
- **serving_input_receiver_fn** – A function that takes no argument and returns a *tf.estimator.export.ServingInputReceiver* or *tf.estimator.export.TensorServingInputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.
- **experimental_mode** – *tf.estimator.ModeKeys* value indicating with mode will be exported. Note that this feature is experimental.

Returns The string path to the exported directory.

Raises

- `ValueError` – if no *serving_input_receiver_fn* is provided, no
- *export_outputs* are provided, or no checkpoint can be found.

export_savedmodel(*export_dir_base*, *serving_input_receiver_fn*, *assets_extra=None*, *as_text=False*, *checkpoint_path=None*, *strip_default_attrs=False*)

Exports inference graph as a *SavedModel* into the given dir.

For a detailed guide, see [Using SavedModel with Estimators](https://tensorflow.org/guide/saved_model#using_savedmodel_with_estimators).

This method builds a new graph by first calling the *serving_input_receiver_fn* to obtain feature *Tensor‘s, and then calling this ‘Estimator*’s *model_fn* to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given *export_dir_base*, and writes a *SavedModel* into it containing a single *tf.MetaGraphDef* saved from this session.

The exported *MetaGraphDef* will provide one *SignatureDef* for each element of the *export_outputs* dict returned from the *model_fn*, named using the same keys. One of these keys is always *tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY*, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding *tf.estimator.export.ExportOutput‘s, and the inputs are always the input receivers provided by the ‘serving_input_receiver_fn*.

Extra assets may be written into the *SavedModel* via the *assets_extra* argument. This should be a dict, where each key gives a destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as *{‘my_asset_file.txt’: ‘/path/to/my_asset_file.txt’}*.

Parameters

- **export_dir_base** – A string containing a directory in which to create timestamped subdirectories containing exported ‘SavedModel‘s.
- **serving_input_receiver_fn** – A function that takes no argument and returns a *tf.estimator.export.ServingInputReceiver* or *tf.estimator.export.TensorServingInputReceiver*.
- **assets_extra** – A dict specifying how to populate the assets.extra directory within the exported *SavedModel*, or *None* if no extra assets are needed.
- **as_text** – whether to write the *SavedModel* proto in text format.
- **checkpoint_path** – The checkpoint path to export. If *None* (the default), the most recent checkpoint found within the model directory is chosen.
- **strip_default_attrs** – Boolean. If *True*, default-valued attributes will be removed from the ‘NodeDef‘s. For a detailed guide, see [Stripping Default-Valued Attributes](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/README.md#stripping-default-valued-attributes).

Returns The string path to the exported directory.

Raises

- `ValueError` – if no *serving_input_receiver_fn* is provided, no
- *export_outputs* are provided, or no checkpoint can be found.

get_variable_names()

Returns list of all variable names in this model.

Returns List of names.

Raises `ValueError` – If the *Estimator* has not produced a checkpoint yet.

get_variable_value(*name*)

Returns value of the variable given by name.

Parameters **name** – string or a list of string, name of the tensor.

Returns Numpy array - value of the tensor.

Raises `ValueError` – If the *Estimator* has not produced a checkpoint yet.

latest_checkpoint()
Finds the filename of the latest saved checkpoint file in *model_dir*.

Returns The full path to the latest checkpoint or *None* if no checkpoint was found.

model_fn
Returns the *model_fn* which is bound to *self.params*.

Returns *def model_fn(features, labels, mode, config)*

Return type The *model_fn* with following signature

predict(*input_fn*, *predict_keys=None*, *hooks=None*, *checkpoint_path=None*, *yield_single_examples=True*)
Yields predictions for given features.

Please note that interleaving two predict outputs does not work. See: [issue/20506](https://github.com/tensorflow/tensorflow/issues/20506#issuecomment-422208517)

Parameters

- **input_fn** – A function that constructs the features. Prediction continues until *input_fn* raises an end-of-input exception (*tf.errors.OutOfRangeError* or *StopIteration*). See [Premade Estimators](https://tensorflow.org/guide/premade_estimators#create_input_functions) for more information. The function should construct and return one of the following:
  - A *tf.data.Dataset* object: Outputs of *Dataset* object must have same constraints as below.
  - features: A *tf.Tensor* or a dictionary of string feature name to *Tensor*. features are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
  - A tuple, in which case the first item is extracted as features.
- **predict_keys** – list of *str*, name of the keys to predict. It is used if the *tf.estimator.EstimatorSpec.predictions* is a *dict*. If *predict_keys* is used then rest of the predictions will be filtered from the dictionary. If *None*, returns all.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the prediction call.
- **checkpoint_path** – Path of a specific checkpoint to predict. If *None*, the latest checkpoint in *model_dir* is used. If there are no checkpoints in *model_dir*, prediction is run with newly initialized *Variables* instead of ones restored from checkpoint.
- **yield_single_examples** – If *False*, yields the whole batch as returned by the *model_fn* instead of decomposing the batch into individual elements. This is useful if *model_fn* returns some tensors whose first dimension is not equal to the batch size.

Yields Evaluated values of *predictions* tensors.

Raises

- `ValueError` – Could not find a trained model in *model_dir*.
- `ValueError` – If batch length of predictions is not the same and *yield_single_examples* is *True*.
- `ValueError` – If there is a conflict between *predict_keys* and *predictions*. For example if *predict_keys* is not *None* but *tf.estimator.EstimatorSpec.predictions* is not a *dict*.

train(*input_fn*, *hooks=None*, *steps=None*, *max_steps=None*, *saving_listeners=None*)
Trains a model given training data *input_fn*.

Parameters

- **input_fn** – A function that provides input data for training as minibatches. See [Premade Estimators](https://tensorflow.org/guide/premade_estimators#create_input_functions) for more information. The function should construct and return one of the following: * A *tf.data.Dataset* object: Outputs of *Dataset* object must be a tuple *(features, labels)* with same constraints as below. * A tuple *(features, labels)*: Where *features* is a *tf.Tensor* or a dictionary of string feature name to *Tensor* and *labels* is a *Tensor* or a dictionary of string label name to *Tensor*. Both *features* and *labels* are consumed by *model_fn*. They should satisfy the expectation of *model_fn* from inputs.
- **hooks** – List of *tf.train.SessionRunHook* subclass instances. Used for callbacks inside the training loop.
- **steps** – Number of steps for which to train the model. If *None*, train forever or train until *input_fn* generates the *tf.errors.OutOfRange* error or *StopIteration* exception. *steps* works incrementally. If you call two times *train(steps=10)* then training occurs in total 20 steps. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before 20 steps. If you don't want to have incremental behavior please set *max_steps* instead. If set, *max_steps* must be *None*.
- **max_steps** – Number of total steps for which to train model. If *None*, train forever or train until *input_fn* generates the *tf.errors.OutOfRange* error or *StopIteration* exception. If set, *steps* must be *None*. If *OutOfRange* or *StopIteration* occurs in the middle, training stops before *max_steps* steps. Two calls to *train(steps=100)* means 200 training iterations. On the other hand, two calls to *train(max_steps=100)* means that the second call will not do any iteration since first call did all 100 steps.
- **saving_listeners** – list of *CheckpointSaverListener* objects. Used for callbacks that run immediately before or after checkpoint savings.

Returns *self*, for chaining.

Raises

- `ValueError` – If both *steps* and *max_steps* are not *None*.
- `ValueError` – If either *steps* or *max_steps* <= *0*.

9.2 Evaluator

Measures `adanet.Ensemble` performance on a given dataset.

9.2.1 Evaluator

class adanet.**Evaluator**(*input_fn*, *steps=None*)
Evaluates candidate ensemble performance.

Parameters

- **input_fn** – Input function returning a tuple of: features - Dictionary of string feature name to *Tensor*. labels - *Tensor* of labels.
- **steps** – Number of steps for which to evaluate the ensembles. If an *OutOfRangeError* occurs, evaluation stops. If set to None, will iterate the dataset until all inputs are exhausted.

Returns An *adanet.Evaluator* instance.

evaluate_adanet_losses(*sess*, *adanet_losses*)
Evaluates the given AdaNet objectives on the data from *input_fn*.

The candidates are fed the same batches of features and labels as provided by *input_fn*, and their losses are computed and summed over *steps* batches.

Parameters

- **sess** – *Session* instance with most recent variable values loaded.
- **adanet_losses** – List of AdaNet loss *Tensors*.

Returns List of evaluated AdaNet losses.

input_fn
Return the input_fn.

steps
Return the number of evaluation steps.

9.3 Summary

Extends `tf.summary` to power AdaNet's TensorBoard integration.

9.3.1 Summary

class adanet.**Summary**
Interface for writing summaries to Tensorboard.

audio(*name*, *tensor*, *sample_rate*, *max_outputs=3*, *family=None*)
Outputs a *tf.Summary* protocol buffer with audio.

The summary has up to *max_outputs* summary values containing audio. The audio is built from *tensor* which must be 3-D with shape *[batch_size, frames, channels]* or 2-D with shape *[batch_size, frames]*. The values are assumed to be in the range of *[-1.0, 1.0]* with a sample rate of *sample_rate*.

The *tag* in the outputted tf.Summary.Value protobufs is generated based on the name, with a suffix depending on the max_outputs setting:

- If *max_outputs* is 1, the summary value tag is '*name*/audio'.
- If *max_outputs* is greater than 1, the summary value tags are

  generated sequentially as '*name*/audio/0', '*name*/audio/1', etc

  Parameters

  - **name** – A name for the generated node. Will also serve as a series name in TensorBoard.
  - **tensor** – A 3-D *float32 Tensor* of shape *[batch_size, frames, channels]* or a 2-D *float32 Tensor* of shape *[batch_size, frames]*.
  - **sample_rate** – A Scalar *float32 Tensor* indicating the sample rate of the signal in hertz.
  - **max_outputs** – Max number of batch elements to generate audio for.
  - **family** – Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns A scalar *Tensor* of type *string*. The serialized *tf.Summary* protocol buffer.

histogram(*name*, *values*, *family=None*)
Outputs a *tf.Summary* protocol buffer with a histogram.

Adding a histogram summary makes it possible to visualize your data's distribution in TensorBoard. You can see a detailed explanation of the TensorBoard histogram dashboard [here](https://www.tensorflow.org/get_started/tensorboard_histograms).

The generated [*tf.Summary*](tensorflow/core/framework/summary.proto) has one summary value containing a histogram for *values*.

This op reports an *InvalidArgument* error if any value is not finite.

Parameters

- **name** – A name for the generated node. Will also serve as a series name in TensorBoard.
- **values** – A real numeric *Tensor*. Any shape. Values to use to build the histogram.
- **family** – Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns A scalar *Tensor* of type *string*. The serialized *tf.Summary* protocol buffer.

image(*name*, *tensor*, *max_outputs=3*, *family=None*)
Outputs a *tf.Summary* protocol buffer with images.

The summary has up to *max_outputs* summary values containing images. The images are built from *tensor* which must be 4-D with shape *[batch_size, height, width, channels]* and where *channels* can be:

- 1: *tensor* is interpreted as Grayscale.
- 3: *tensor* is interpreted as RGB.
- 4: *tensor* is interpreted as RGBA.

The images have the same number of channels as the input tensor. For float input, the values are normalized one image at a time to fit in the range *[0, 255]*. *uint8* values are unchanged. The op uses two different normalization algorithms:

- If the input values are all positive, they are rescaled so the largest

one is 255. * If any input value is negative, the values are shifted so input value 0.0

is at 127. They are then rescaled so that either the smallest value is 0, or the largest one is 255.

The *tag* in the outputted tf.Summary.Value protobufs is generated based on the name, with a suffix depending on the max_outputs setting:

- If *max_outputs* is 1, the summary value tag is '*name*/image'.
- If *max_outputs* is greater than 1, the summary value tags are

generated sequentially as '*name*/image/0', '*name*/image/1', etc.

Parameters

- **name** – A name for the generated node. Will also serve as a series name in TensorBoard.
- **tensor** – A 4-D *uint8* or *float32 Tensor* of shape *[batch_size, height, width, channels]* where *channels* is 1, 3, or 4.
- **max_outputs** – Max number of batch elements to generate images for.

- **family** – Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns A scalar *Tensor* of type *string*. The serialized *tf.Summary* protocol buffer.

scalar(*name*, *tensor*, *family=None*)

Outputs a *tf.Summary* protocol buffer containing a single scalar value.

The generated tf.Summary has a Tensor.proto containing the input Tensor.

Parameters

- **name** – A name for the generated node. Will also serve as the series name in TensorBoard.
- **tensor** – A real numeric Tensor containing a single value.
- **family** – Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns A scalar *Tensor* of type *string*. Which contains a *tf.Summary* protobuf.

Raises `ValueError` – If tensor has the wrong shape or type.

9.4 ReportMaterializer

9.4.1 ReportMaterializer

class adanet.**ReportMaterializer**(*input_fn*, *steps=None*)

Materializes reports.

Specifically it materializes a subnetwork's *adanet.subnetwork.Report* instances into *adanet.subnetwork.MaterializedReport* instances.

Requires an input function *input_fn* that returns a tuple of:

- features: Dictionary of string feature name to *Tensor*.
- labels: *Tensor* of labels.

Parameters

- **input_fn** – The input function.
- **steps** – Number of steps for which to materialize the ensembles. If an *OutOfRangeError* occurs, materialization stops. If set to None, will iterate the dataset until all inputs are exhausted.

Returns A *ReportMaterializer* instance.

input_fn

Returns the input_fn that materialize_subnetwork_reports would run on.

Even though this property appears to be unused, it would be used to build the AdaNet model graph inside AdaNet estimator.train(). After the graph is built, the queue_runners are started and the initializers are run, AdaNet estimator.train() passes its tf.Session as an argument to materialize_subnetwork_reports(), thus indirectly making input_fn available to materialize_subnetwork_reports.

materialize_subnetwork_reports(*sess*, *iteration_number*, *subnetwork_reports*, *included_subnetwork_names*)

Materializes the Tensor objects in subnetwork_reports using sess.

This converts the Tensors in subnetwork_reports to ndarrays, logs the progress, converts the ndarrays to python primitives, then packages them into *adanet.subnetwork.MaterializedReports*.

Parameters

- **sess** – *Session* instance with most recent variable values loaded.
- **iteration_number** – Integer iteration number.
- **subnetwork_reports** – Dict mapping string names to *subnetwork.Report* objects to be materialized.
- **included_subnetwork_names** – List of string names of the ‘subnetwork.Report‘s that are included in the final ensemble.

Returns List of *adanet.subnetwork.MaterializedReport* objects.

steps

Return the number of steps.

CHAPTER 10

adanet.ensemble

Defines built-in ensemble methods and interfaces for custom ensembles.

10.1 Ensembles

Interfaces and containers for defining ensembles.

10.1.1 Ensemble

class adanet.ensemble.**Ensemble**

An abstract ensemble of subnetworks.

logits

Ensemble logits `tf.Tensor`.

subnetworks

Returns an ordered `Iterable` of the ensemble's subnetworks.

10.1.2 ComplexityRegularized

class adanet.ensemble.**ComplexityRegularized**

An AdaNet ensemble where subnetworks are regularized by model complexity.

Hence an ensemble is a collection of subnetworks which forms a neural network through the weighted sum of their outputs:

$$F(x) = \sum_{i=1}^{N} w_i h_i(x) + b$$

Parameters

- **weighted_subnetworks** – List of *adanet.ensemble.WeightedSubnetwork* instances that form this ensemble. Ordered from first to most recent.
- **bias** – Bias term `tf.Tensor` or dict of string to bias term `tf.Tensor` (for multi-head) for the ensemble's logits.
- **logits** – Logits `tf.Tensor` or dict of string to logits `tf.Tensor` (for multi-head). The result of the function f as defined in Section 5.1 which is the sum of the logits of all `adanet.WeightedSubnetwork` instances in ensemble.
- **subnetworks** – List of *adanet.subnetwork.Subnetwork* instances that form this ensemble. This is kept together with weighted_subnetworks for legacy reasons.
- **complexity_regularization** – Regularization to be added in the Adanet loss.

Returns An `adanet.ensemble.Weighted` instance.

10.1.3 MixtureWeightType

class adanet.ensemble.**MixtureWeightType**

Mixture weight types available for learning subnetwork contributions.

The following mixture weight types are defined:

- *SCALAR*: Produces a rank 0 *Tensor* mixture weight.
- *VECTOR*: Produces a rank 1 *Tensor* mixture weight.
- *MATRIX*: Produces a rank 2 *Tensor* mixture weight.

10.1.4 WeightedSubnetwork

class adanet.ensemble.**WeightedSubnetwork**

An AdaNet weighted subnetwork.

A weighted subnetwork is a weight applied to a subnetwork's last layer or logits (depending on the mixture weights type).

Parameters

- **name** – String name of `subnetwork` as defined by its *adanet.subnetwork.Builder*.
- **iteration_number** – Integer iteration when the subnetwork was created.
- **weight** – The weight `tf.Tensor` or dict of string to weight `tf.Tensor` (for multi-head) to apply to this subnetwork. The AdaNet paper refers to this weight as w in Equations (4), (5), and (6).
- **logits** – The output `tf.Tensor` or dict of string to weight `tf.Tensor` (for multi-head) after the matrix multiplication of `weight` and the subnetwork's `last_layer`. The output's shape is [batch_size, logits_dimension]. It is equivalent to a linear logits layer in a neural network.
- **subnetwork** – The *adanet.subnetwork.Subnetwork* to weight.

Returns An *adanet.ensemble.WeightedSubnetwork* object.

10.2 Ensemblers

Ensemble learning definitions.

10.2.1 Ensembler

class adanet.ensemble.**Ensembler**

An abstract ensembler.

build_ensemble(*subnetworks*, *previous_ensemble_subnetworks*, *features*, *labels*, *logits_dimension*, *training*, *iteration_step*, *summary*, *previous_ensemble*)

Builds an ensemble of subnetworks.

Accessing the global step via `tf.train.get_or_create_global_step()` or `tf.train.get_global_step()` within this scope will return an incrementable iteration step since the beginning of the iteration.

Parameters

- **subnetworks** – Ordered iterable of *adanet.subnetwork.Subnetwork* instances to ensemble. Must have at least one element.
- **previous_ensemble_subnetworks** – Ordered iterable of *adanet.subnetwork.Subnetwork* instances present in previous ensemble to be used. The subnetworks from previous_ensemble not included in this list should be pruned. Can be set to None or empty.
- **features** – Input `dict` of `tf.Tensor` objects.
- **labels** – Labels `tf.Tensor` or a dictionary of string label name to `tf.Tensor` (for multi-head). Can be `None`.
- **logits_dimension** – Size of the last dimension of the logits `tf.Tensor`. Typically, logits have for shape *[batch_size, logits_dimension]*.
- **training** – A python boolean indicating whether the graph is in training mode or prediction mode.
- **iteration_step** – Integer `tf.Tensor` representing the step since the beginning of the current iteration, as opposed to the global step.
- **summary** – An *adanet.Summary* for scoping summaries to individual ensembles in Tensorboard. Using `tf.summary()` within this scope will use this *adanet.Summary* under the hood.
- **previous_ensemble** – The best `adanet.Ensemble` from iteration *t-1*. The created subnetwork will extend the previous ensemble to form the `adanet.Ensemble` at iteration *t*.

Returns An *adanet.ensemble.Ensemble* subclass instance.

build_train_op(*ensemble*, *loss*, *var_list*, *labels*, *iteration_step*, *summary*, *previous_ensemble*)

Returns an op for training an ensemble.

Accessing the global step via `tf.train.get_or_create_global_step()` or `tf.train.get_global_step()` within this scope will return an incrementable iteration step since the beginning of the iteration.

Parameters

- **ensemble** – The *adanet.ensemble.Ensemble* subclass instance returned by this instance's *build_ensemble()*.
- **loss** – A `tf.Tensor` containing the ensemble's loss to minimize.
- **var_list** – List of ensemble `tf.Variable` parameters to update as part of the training operation.
- **labels** – Labels `tf.Tensor` or a dictionary of string label name to `tf.Tensor` (for multi-head).
- **iteration_step** – Integer `tf.Tensor` representing the step since the beginning of the current iteration, as opposed to the global step.
- **summary** – An *adanet.Summary* for scoping summaries to individual ensembles in Tensorboard. Using `tf.summary` within this scope will use this *adanet.Summary* under the hood.
- **previous_ensemble** – The best *adanet.ensemble.Ensemble* from the previous iteration.

Returns Either a train op or an *adanet.ensemble.TrainOpSpec*.

name
This ensembler's unique string name.

10.2.2 ComplexityRegularizedEnsembler

class adanet.ensemble.**ComplexityRegularizedEnsembler**(*optimizer=None*, *mixture_weight_type='scalar'*, *mixture_weight_initializer=None*, *warm_start_mixture_weights=False*, *model_dir=None*, *adanet_lambda=0.0*, *adanet_beta=0.0*, *use_bias=False*)

The AdaNet algorithm implemented as an *adanet.ensemble.Ensembler*.

The AdaNet algorithm was introduced in the [Cortes et al. ICML 2017] paper: https://arxiv.org/abs/1607.01097.

The AdaNet algorithm uses a weak learning algorithm to iteratively generate a set of candidate subnetworks that attempt to minimize the loss function defined in Equation (4) as part of an ensemble. At the end of each iteration, the best candidate is chosen based on its ensemble's complexity-regularized train loss. New subnetworks are allowed to use any subnetwork weights within the previous iteration's ensemble in order to improve upon them. If the complexity-regularized loss of the new ensemble, as defined in Equation (4), is less than that of the previous iteration's ensemble, the AdaNet algorithm continues onto the next iteration.

AdaNet attempts to minimize the following loss function to learn the mixture weights w of each subnetwork h in the ensemble with differentiable convex non-increasing surrogate loss function Φ:

Equation (4):

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

with $\lambda >= 0$ and $\beta >= 0$.

Parameters

- **optimizer** – A `tf.train.Optimizer` instance to be used for building the train op. If left as None, `tf.no_op()` is returned as train op.
- **mixture_weight_type** – The *adanet.ensemble.MixtureWeightType* defining which mixture weight type to learn on top of the subnetworks' logits.
- **mixture_weight_initializer** – The initializer for mixture_weights. When `None`, the default is different according to `mixture_weight_type`:
  - `SCALAR` initializes to $1/N$ where N is the number of subnetworks in the ensemble giving a uniform average.
  - `VECTOR` initializes each entry to $1/N$ where N is the number of subnetworks in the ensemble giving a uniform average.
  - `MATRIX` uses `tf.zeros_initializer()`.
- **warm_start_mixture_weights** – Whether, at the beginning of an iteration, to initialize the mixture weights of the subnetworks from the previous ensemble to their learned value at the previous iteration, as opposed to retraining them from scratch. Takes precedence over the value for `mixture_weight_initializer` for subnetworks from previous iterations.
- **model_dir** – The model dir to use for warm-starting mixture weights and bias at the logit layer. Ignored if `warm_start_mixture_weights` is `False`.
- **adanet_lambda** – Float multiplier λ for applying $L1$ regularization to subnetworks' mixture weights w in the ensemble proportional to their complexity. See Equation (4) in the AdaNet paper.
- **adanet_beta** – Float $L1$ regularization multiplier β to apply equally to all subnetworks' weights w in the ensemble regardless of their complexity. See Equation (4) in the AdaNet paper.
- **use_bias** – Whether to add a bias term to the ensemble's logits.

Returns An *adanet.ensemble.ComplexityRegularizedEnsembler* instance.

Raises

- `ValueError` – if `warm_start_mixture_weights` is `True` but
- `model_dir` is `None`.

build_ensemble(*subnetworks*, *previous_ensemble_subnetworks*, *features*, *labels*, *logits_dimension*, *training*, *iteration_step*, *summary*, *previous_ensemble*)

Builds an ensemble of subnetworks.

Accessing the global step via `tf.train.get_or_create_global_step()` or `tf.train.get_global_step()` within this scope will return an incrementable iteration step since the beginning of the iteration.

Parameters

- **subnetworks** – Ordered iterable of *adanet.subnetwork.Subnetwork* instances to ensemble. Must have at least one element.
- **previous_ensemble_subnetworks** – Ordered iterable of *adanet.subnetwork.Subnetwork* instances present in previous ensemble to be used. The subnetworks from previous_ensemble not included in this list should be pruned. Can be set to None or empty.
- **features** – Input `dict` of `tf.Tensor` objects.

- **labels** – Labels `tf.Tensor` or a dictionary of string label name to `tf.Tensor` (for multi-head). Can be `None`.
- **logits_dimension** – Size of the last dimension of the logits `tf.Tensor`. Typically, logits have for shape *[batch_size, logits_dimension]*.
- **training** – A python boolean indicating whether the graph is in training mode or prediction mode.
- **iteration_step** – Integer `tf.Tensor` representing the step since the beginning of the current iteration, as opposed to the global step.
- **summary** – An *adanet.Summary* for scoping summaries to individual ensembles in Tensorboard. Using `tf.summary()` within this scope will use this *adanet.Summary* under the hood.
- **previous_ensemble** – The best `adanet.Ensemble` from iteration *t-1*. The created subnetwork will extend the previous ensemble to form the `adanet.Ensemble` at iteration *t*.

Returns An *adanet.ensemble.Ensemble* subclass instance.

build_train_op(*ensemble*, *loss*, *var_list*, *labels*, *iteration_step*, *summary*, *previous_ensemble*)
Returns an op for training an ensemble.

Accessing the global step via `tf.train.get_or_create_global_step()` or `tf.train.get_global_step()` within this scope will return an incrementable iteration step since the beginning of the iteration.

Parameters

- **ensemble** – The *adanet.ensemble.Ensemble* subclass instance returned by this instance's *build_ensemble()*.
- **loss** – A `tf.Tensor` containing the ensemble's loss to minimize.
- **var_list** – List of ensemble `tf.Variable` parameters to update as part of the training operation.
- **labels** – Labels `tf.Tensor` or a dictionary of string label name to `tf.Tensor` (for multi-head).
- **iteration_step** – Integer `tf.Tensor` representing the step since the beginning of the current iteration, as opposed to the global step.
- **summary** – An *adanet.Summary* for scoping summaries to individual ensembles in Tensorboard. Using `tf.summary` within this scope will use this *adanet.Summary* under the hood.
- **previous_ensemble** – The best *adanet.ensemble.Ensemble* from the previous iteration.

Returns Either a train op or an *adanet.ensemble.TrainOpSpec*.

name
This ensembler's unique string name.

10.2.3 TrainOpSpec

class adanet.ensemble.**TrainOpSpec**
A data structure for specifying ensembler training operations.

Parameters

- **train_op** – Op for the training step.
- **chief_hooks** – Iterable of `tf.train.SessionRunHook` objects to run on the chief worker during training.
- **hooks** – Iterable of `tf.train.SessionRunHook` objects to run on all workers during training.

Returns An *adanet.ensemble.TrainOpSpec* object.

10.3 Strategies

Ensemble strategies for grouping subnetworks.

10.3.1 Strategy

class adanet.ensemble.**Strategy**

An abstract ensemble strategy.

generate_ensemble_candidates(*subnetwork_builders*, *previous_ensemble_subnetwork_builders*)

Generates ensemble candidates to search over this iteration.

Parameters

- **subnetwork_builders** – Candidate *adanet.subnetwork.Builder* instances for this iteration.
- **previous_ensemble_subnetwork_builders** – *adanet.subnetwork.Builder* instances from the previous ensemble. Including only a subset of these in a returned *adanet.ensemble.Candidate* is equivalent to pruning the previous ensemble.

Returns An iterable of *adanet.ensemble.Candidate* instances to train and consider this iteration.

10.3.2 SoloStrategy

class adanet.ensemble.**SoloStrategy**

Produces a model composed of a single subnetwork.

An ensemble of one.

This is effectively the same as pruning all previous ensemble subnetworks, and only adding one subnetwork candidate to the ensemble.

generate_ensemble_candidates(*subnetwork_builders*, *previous_ensemble_subnetwork_builders*)

Generates ensemble candidates to search over this iteration.

Parameters

- **subnetwork_builders** – Candidate *adanet.subnetwork.Builder* instances for this iteration.
- **previous_ensemble_subnetwork_builders** – *adanet.subnetwork.Builder* instances from the previous ensemble. Including only a subset of these in

a returned *adanet.ensemble.Candidate* is equivalent to pruning the previous ensemble.

Returns An iterable of *adanet.ensemble.Candidate* instances to train and consider this iteration.

10.3.3 GrowStrategy

class adanet.ensemble.**GrowStrategy**

Greedily grows an ensemble, one subnetwork at a time.

generate_ensemble_candidates(*subnetwork_builders*, *previous_ensemble_subnetwork_builders*)

Generates ensemble candidates to search over this iteration.

Parameters

- **subnetwork_builders** – Candidate *adanet.subnetwork.Builder* instances for this iteration.
- **previous_ensemble_subnetwork_builders** – *adanet.subnetwork.Builder* instances from the previous ensemble. Including only a subset of these in a returned *adanet.ensemble.Candidate* is equivalent to pruning the previous ensemble.

Returns An iterable of *adanet.ensemble.Candidate* instances to train and consider this iteration.

10.3.4 AllStrategy

class adanet.ensemble.**AllStrategy**

Ensembles all subnetworks from the current iteration.

generate_ensemble_candidates(*subnetwork_builders*, *previous_ensemble_subnetwork_builders*)

Generates ensemble candidates to search over this iteration.

Parameters

- **subnetwork_builders** – Candidate *adanet.subnetwork.Builder* instances for this iteration.
- **previous_ensemble_subnetwork_builders** – *adanet.subnetwork.Builder* instances from the previous ensemble. Including only a subset of these in a returned *adanet.ensemble.Candidate* is equivalent to pruning the previous ensemble.

Returns An iterable of *adanet.ensemble.Candidate* instances to train and consider this iteration.

10.3.5 Candidate

class adanet.ensemble.**Candidate**

An ensemble candidate found during the search phase.

Parameters

- **name** – String name of this ensemble candidate.

- **subnetwork_builders** – Candidate *adanet.subnetwork.Builder* instances to include in the ensemble.
- **previous_ensemble_subnetwork_builders** – *adanet.subnetwork.Builder* instances to include from the previous ensemble.

CHAPTER 11

adanet.subnetwork

Low-level APIs for defining custom subnetworks and search spaces.

11.1 Generators

Interfaces and containers for defining subnetworks, search spaces, and search algorithms.

11.1.1 Subnetwork

class adanet.subnetwork.**Subnetwork**

An AdaNet subnetwork.

In the AdaNet paper, an *adanet.subnetwork.Subnetwork* is are called a *subnetwork*, and indicated by *h*. A collection of weighted subnetworks form an AdaNet ensemble.

Parameters

- **last_layer** – tf.Tensor output or dict of string to tf.Tensor outputs (for multi-head) of the last layer of the subnetwork, i.e the layer before the logits layer. When the mixture weight type is MATRIX, the AdaNet algorithm takes care of computing ensemble mixture weights matrices (one per subnetwork) that multiply the various last layers of the ensemble's subnetworks, and regularize them using their subnetwork's complexity. This field is represented by *h* in the AdaNet paper.
- **logits** – tf.Tensor logits or dict of string to tf.Tensor logits (for multi-head) for training the subnetwork. These logits are not used in the ensemble's outputs if the mixture weight type is MATRIX, instead AdaNet learns its own logits (mixture weights) from the subnetwork's *last_layers* with complexity regularization. The logits are used in the ensemble only when the mixture weights type is SCALAR or VECTOR. Even though the logits are not used in the ensemble in some cases, they should always be supplied as adanet uses the logits to train the subnetworks.

- **complexity** – A scalar `tf.Tensor` representing the complexity of the subnetwork's architecture. It is used for choosing the best subnetwork at each iteration, and for regularizing the weighted outputs of more complex subnetworks.
- **persisted_tensors** – DEPRECATED. See *shared*. Optional nested dictionary of string to `tf.Tensor` to persist across iterations. At the end of an iteration, the `tf.Tensor` instances will be available to subnetworks in the next iterations, whereas others that are not part of the *Subnetwork* will be pruned. This allows later *adanet.subnetwork.Subnetwork* instances to dynamically build upon arbitrary `tf.Tensors` from previous *adanet.subnetwork.Subnetwork* instances.
- **shared** – Optional Python object(s), primitive(s), or function(s) to share with subnetworks within the same iteration or in future iterations.

Returns A validated *adanet.subnetwork.Subnetwork* object.

Raises

- `ValueError` – If last_layer is None.
- `ValueError` – If logits is None.
- `ValueError` – If logits is a dict but last_layer is not.
- `ValueError` – If last_layer is a dict but logits is not.
- `ValueError` – If complexity is None.
- `ValueError` – If persisted_tensors is present but not a dictionary.
- `ValueError` – If persisted_tensors contains an empty nested dictionary.

11.1.2 TrainOpSpec

class adanet.subnetwork.**TrainOpSpec**

A data structure for specifying training operations.

Parameters

- **train_op** – Op for the training step.
- **chief_hooks** – Iterable of `tf.train.SessionRunHook` objects to run on the chief worker during training.
- **hooks** – Iterable of `tf.train.SessionRunHook` objects to run on all workers during training.

Returns A *adanet.subnetwork.TrainOpSpec* object.

11.1.3 Builder

class adanet.subnetwork.**Builder**

Bases: `object`

Interface for a subnetwork builder.

Given features, labels, and the best ensemble of subnetworks at iteration t-1, a *Builder* creates a *Subnetwork* to add to a candidate ensemble at iteration t. These candidate ensembles are evaluated against one another at the end of the iteration, and the best one is selected based on its complexity-regularized loss.

build_subnetwork(*features*, *labels*, *logits_dimension*, *training*, *iteration_step*, *summary*, *previous_ensemble=None*)

Returns the candidate *Subnetwork* to add to the ensemble.

This method will be called only once before `build_subnetwork_train_op()`. This method should construct the candidate subnetwork's graph operations and variables.

Accessing the global step via `tf.train.get_or_create_global_step()` or `tf.train.get_global_step()` within this scope will return an incrementable iteration step since the beginning of the iteration.

Parameters

- **features** – Input *dict* of `tf.Tensor` objects.
- **labels** – Labels `tf.Tensor` or a dictionary of string label name to `tf.Tensor` (for multi-head). Can be *None*.
- **logits_dimension** – Size of the last dimension of the logits `tf.Tensor`. Typically, logits have for shape *[batch_size, logits_dimension]*.
- **training** – A python boolean indicating whether the graph is in training mode or prediction mode.
- **iteration_step** – Integer `tf.Tensor` representing the step since the beginning of the current iteration, as opposed to the global step.
- **summary** – An `adanet.Summary` for scoping summaries to individual subnetworks in Tensorboard. Using `tf.summary()` within this scope will use this `adanet.Summary` under the hood.
- **previous_ensemble** – The best `adanet.Ensemble` from iteration t-1. The created subnetwork will extend the previous ensemble to form the `adanet.Ensemble` at iteration t.

Returns An `adanet.subnetwork.Subnetwork` instance.

build_subnetwork_report()

Returns a *subnetwork.Report* to materialize and record.

This method will be called once after `build_subnetwork()`. Do NOT depend on variables created in `build_subnetwork_train_op()`, because they are not called before `build_subnetwork_report()` is called.

If it returns None, AdaNet records the name and standard eval metrics.

build_subnetwork_train_op(*subnetwork*, *loss*, *var_list*, *labels*, *iteration_step*, *summary*, *previous_ensemble*)

Returns an op for training a new subnetwork.

This method will be called once after `build_subnetwork()`.

Accessing the global step via `tf.train.get_or_create_global_step()` or `tf.train.get_global_step()` within this scope will return an incrementable iteration step since the beginning of the iteration.

Parameters

- **subnetwork** – Newest subnetwork, that is not part of the *previous_ensemble*.
- **loss** – A `tf.Tensor` containing the subnetwork's loss to minimize.
- **var_list** – List of subnetwork `tf.Variable` parameters to update as part of the training operation.

- **labels** – Labels `tf.Tensor` or a dictionary of string label name to `tf.Tensor` (for multi-head).
- **iteration_step** – Integer `tf.Tensor` representing the step since the beginning of the current iteration, as opposed to the global step.
- **summary** – An *adanet.Summary* for scoping summaries to individual subnetworks in Tensorboard. Using *tf.summary* within this scope will use this *adanet.Summary* under the hood.
- **previous_ensemble** – The best *Ensemble* from iteration t-1. The created subnetwork will extend the previous ensemble to form the *Ensemble* at iteration t. Is None for iteration 0.

Returns Either a train op or an *adanet.subnetwork.TrainOpSpec*.

name

Returns the unique name of this subnetwork within an iteration.

Returns String name of this subnetwork.

prune_previous_ensemble(*previous_ensemble*)

Specifies which subnetworks from the previous ensemble to keep.

The selected subnetworks from the previous ensemble will be kept in the candidate ensemble that includes this subnetwork.

By default, none of the previous ensemble subnetworks are pruned.

Parameters **previous_ensemble** – `adanet.Ensemble` object.

Returns List of integer indices of *weighted_subnetworks* to keep.

11.1.4 Generator

class adanet.subnetwork.**Generator**

Bases: `object`

Interface for a candidate subnetwork generator.

Given the ensemble of subnetworks at iteration t-1, this object is responsible for generating the set of candidate subnetworks for iteration t that minimize the objective as part of an ensemble.

generate_candidates(*previous_ensemble*, *iteration_number*, *previous_ensemble_reports*, *all_reports*)

Generates *adanet.subnetwork.Builder* instances for an iteration.

NOTE: Every call to *generate_candidates()* must be deterministic for the given arguments.

Parameters

- **previous_ensemble** – The best `adanet.Ensemble` from iteration t-1. DEPRECATED. We are transitioning away from the use of previous_ensemble in generate_candidates. New Generators should *not* use previous_ensemble in their implementation of generate_candidates – please only use iteration_number, previous_ensemble_reports and all_reports.
- **iteration_number** – Python integer AdaNet iteration t, starting from 0.
- **previous_ensemble_reports** – List of *adanet.subnetwork.MaterializedReport* instances corresponding to the Builders composing `adanet.Ensemble` from iteration t-1. The first element in the list corresponds to the Builder

added in the first iteration. If a *adanet.subnetwork.MaterializedReport* is not supplied to the estimator, previous_ensemble_report is *None*.

- **all_reports** – List of *adanet.subnetwork.MaterializedReport* instances. If an `adanet.subnetwork.ReportMaterializer` is not supplied to the estimator, *all_reports* is *None*. If `adanet.subnetwork.ReportMaterializer` is supplied to the estimator and t=0, *all_reports* is an empty List. Otherwise, *all_reports* is a sequence of Lists. Each element of the sequence is a List containing all the *adanet.subnetwork.MaterializedReport* instances in an AdaNet iteration, starting from iteration 0, and ending at iteration t-1.

Returns A list of *adanet.subnetwork.Builder* instances.

11.2 Reports

Containers for metadata about trained subnetworks.

11.2.1 Report

class adanet.subnetwork.**Report**

A container for data to be collected about a *Subnetwork*.

Parameters

- **hparams** – A dict mapping strings to python strings, ints, bools, or floats. It is meant to contain the constants that define the *adanet.subnetwork.Builder*, such as dropout, number of layers, or initial learning rate.
- **attributes** – A dict mapping strings to rank 0 Tensors of dtype string, int32, or float32. It is meant to contain properties that may or may not change over the course of training the *adanet.subnetwork.Subnetwork*, such as the number of parameters, the Lipschitz constant, the $L2$ norm of the weights, or learning rate at materialization time.
- **metrics** – Dict of metric results keyed by name. The values of the dict are the results of calling a metric function, namely a *(metric_tensor, update_op)* tuple. *metric_tensor* should be evaluated without any impact on state (typically is a pure computation results based on variables.). For example, it should not trigger the `update_op` or requires any input fetching. This is meant to contain metrics of interest, such as the training loss, complexity regularized loss, or standard deviation of the last layer outputs.

Returns A validated *adanet.subnetwork.Report* object.

Raises `ValueError` – If validation fails.

11.2.2 MaterializedReport

class adanet.subnetwork.**MaterializedReport**

Data collected about a *adanet.subnetwork.Subnetwork*.

Parameters

- **iteration_number** – A python integer for the AdaNet iteration number, starting from 0.
- **name** – A string, which is either the name of the corresponding Builder, or "previous_ensemble" if it refers to the previous_ensemble.

- **hparams** – A dict mapping strings to python strings, ints, or floats. These are constants passed from the author of the *adanet.subnetwork.Builder* that was used to construct this *adanet.subnetwork.Subnetwork*. It is meant to contain the arguments that defined the *adanet.subnetwork.Builder*, such as dropout, number of layers, or initial learning rate.
- **attributes** – A dict mapping strings to python strings, ints, bools, or floats. These are python primitives that come from materialized Tensors; these Tensors were defined by the author of the *adanet.subnetwork.Builder* that was used to construct this *adanet.subnetwork.Subnetwork*. It is meant to contain properties that may or may not change over the course of training the *adanet.subnetwork.Subnetwork*, such as the number of parameters, the Lipschitz constant, or the $L2$ norm of the weights.
- **metrics** – A dict mapping strings to python strings, ints, or floats. These are python primitives that come from metrics that were evaluated on the trained *adanet.subnetwork.Subnetwork* over some dataset; these metrics were defined by the author of the *adanet.subnetwork.Builder* that was used to construct this *adanet.subnetwork.Subnetwork*. It is meant to contain performance metrics or measures that could predict generalization, such as the training loss, complexity regularized loss, or standard deviation of the last layer outputs.
- **included_in_final_ensemble** – A boolean denoting whether the associated *adanet.subnetwork.Subnetwork* was included in the ensemble at the end of the AdaNet iteration.

Returns An *adanet.subnetwork.MaterializedReport* object.

CHAPTER 12

adanet.distributed

The *adanet.distributed* package.

This package methods for distributing computation using the TensorFlow computation graph.

12.1 PlacementStrategy

class adanet.distributed.**PlacementStrategy**

Abstract placement strategy for distributed training.

Given a cluster of workers, the placement strategy determines which subgraph each worker constructs.

config

Returns this strategy's configuration.

Returns The `tf.estimator.RunConfig` instance that defines the cluster.

should_build_ensemble(*num_subnetworks*)

Whether to build the ensemble on the current worker.

Parameters **num_subnetworks** – Integer number of subnetworks to train in the current iteration.

Returns Boolean whether to build the ensemble on the current worker.

should_build_subnetwork(*num_subnetworks*, *subnetwork_index*)

Whether to build the given subnetwork on the current worker.

Parameters

- **num_subnetworks** – Integer number of subnetworks to train in the current iteration.
- **subnetwork_index** – Integer index of the subnetwork in the list of the current iteration's subnetworks.

Returns Boolean whether to build the given subnetwork on the current worker.

should_train_subnetworks(*num_subnetworks*)
Whether to train subnetworks on the current worker.

Parameters **num_subnetworks** – Integer number of subnetworks to train in the current iteration.

Returns Boolean whether to train subnetworks on the current worker.

12.2 ReplicationStrategy

class adanet.distributed.**ReplicationStrategy**
A simple strategy that replicates the same graph on every worker.

This strategy does not scale well as the number of subnetworks and workers increases. For m workers, n parameter servers, and k subnetworks, this strategy will scale with $O(m)$ training speedup, $O(m*n*k)$ variable fetches from parameter servers, and $O(k)$ memory required per worker. Additionally there will be $O(m)$ stale gradients per subnetwork when training with asynchronous SGD.

Returns A *ReplicationStrategy* instance for the current cluster.

should_build_ensemble(*num_subnetworks*)
Whether to build the ensemble on the current worker.

Parameters **num_subnetworks** – Integer number of subnetworks to train in the current iteration.

Returns Boolean whether to build the ensemble on the current worker.

should_build_subnetwork(*num_subnetworks*, *subnetwork_index*)
Whether to build the given subnetwork on the current worker.

Parameters

- **num_subnetworks** – Integer number of subnetworks to train in the current iteration.
- **subnetwork_index** – Integer index of the subnetwork in the list of the current iteration's subnetworks.

Returns Boolean whether to build the given subnetwork on the current worker.

should_train_subnetworks(*num_subnetworks*)
Whether to train subnetworks on the current worker.

Parameters **num_subnetworks** – Integer number of subnetworks to train in the current iteration.

Returns Boolean whether to train subnetworks on the current worker.

12.3 RoundRobinStrategy

class adanet.distributed.**RoundRobinStrategy**(*drop_remainder=False*)
A strategy that round-robin assigns subgraphs to specific workers.

Specifically, it selects dedicated workers to only train ensemble variables, and round-robin assigns subnetworks to dedicated subnetwork-training workers.

Unlike *ReplicationStrategy*, this strategy scales better with the number of subnetworks, workers, and parameter servers. For m workers, n parameter servers, and k subnetworks, this strategy will scale with $O(m/k)$ training speedup, $O(m * n/k)$ variable fetches from parameter servers, and $O(1)$ memory required per worker.

Additionally, there will only be $O(m/k)$ stale gradients per subnetwork when training with asynchronous SGD, which reduces training instability versus *ReplicationStrategy*.

When there are more workers than subnetworks, this strategy assigns subnetworks to workers modulo the number of subnetworks.

Conversely, when there are more subnetworks than workers, this round robin assigns subnetworks modulo the number of workers. So certain workers may end up training more than one subnetwork.

This strategy gracefully handles scenarios when the number of subnetworks does not perfectly divide the number of workers and vice-versa. It also supports different numbers of subnetworks at different iterations, and reloading training with a resized cluster.

Parameters **drop_remainder** – Bool whether to drop remaining subnetworks that haven't been assigned to a worker in the remainder after perfect division of workers by the current iteration's num_subnetworks + 1. When `True`, each subnetwork worker will only train a single subnetwork, and subnetworks that have not been assigned to assigned to a worker are dropped. NOTE: This can result in subnetworks not being assigned to any worker when num_workers < num_subnetworks + 1. When `False`, remaining subnetworks during the round-robin assignment will be placed on workers that already have a subnetwork.

Returns A *RoundRobinStrategy* instance for the current cluster.

should_build_ensemble(*num_subnetworks*)
Whether to build the ensemble on the current worker.

Parameters **num_subnetworks** – Integer number of subnetworks to train in the current iteration.

Returns Boolean whether to build the ensemble on the current worker.

should_build_subnetwork(*num_subnetworks*, *subnetwork_index*)
Whether to build the given subnetwork on the current worker.

Parameters

- **num_subnetworks** – Integer number of subnetworks to train in the current iteration.
- **subnetwork_index** – Integer index of the subnetwork in the list of the current iteration's subnetworks.

Returns Boolean whether to build the given subnetwork on the current worker.

should_train_subnetworks(*num_subnetworks*)
Whether to train subnetworks on the current worker.

Parameters **num_subnetworks** – Integer number of subnetworks to train in the current iteration.

Returns Boolean whether to train subnetworks on the current worker.

CHAPTER 13

Indices and tables

- genindex
- modindex

Python Module Index

a

Index

H

I

L

M

N

P

R

S

T

W