tfp.layers.DenseFlipout

View source on GitHub (https://github.com/tensorflow/probability/blob/v0.16.0/tensorflow_probability/python/layers/dense_variational.py#L556-L718)

Densely-connected layer class with Flipout estimator.

View aliases

Main aliases

`tfp.layers.dense_variational.DenseFlipout` (https://www.tensorflow.org/probability/api_docs/python/tfp/layers/DenseFlipout)

```
tfp.layers.DenseFlipout(
    units, activation=None, activity_regularizer=None, trainable=True,
    kernel_posterior_fn=tfp_layers_util.default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn=(lambda d: d.sample()),
    kernel_prior_fn=tfp.layers.default_multivariate_normal_fn,
    kernel_divergence_fn=(lambda q, p, ignore: kl_lib.kl_divergence(q, p)), bi
    terior_fn=tfp_layers_util.default_mean_field_normal_fn(is_singular=True),
    bias_posterior_tensor_fn=(lambda d: d.sample()), bias_prior_fn=None,
    bias_divergence_fn=(lambda q, p, ignore: kl_lib.kl_divergence(q, p)), seec
    **kwargs
)
```

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the `kernel` and/or the `bias` are drawn from distributions. By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

```
kernel, bias ~ posterior
outputs = activation(matmul(inputs, kernel) + bias)
```

It uses the Flipout estimator [(Wen et al., 2018)][1], which performs a Monte Carlo approximation of the distribution integrating over the `kernel` and `bias`. Flipout uses roughly twice as many floating point operations as the reparameterization estimator but has the advantage of significantly lower variance.

The arguments permit separate specification of the surrogate posterior (`q(W|x)`), prior (`p(W)`), and divergence for both the `kernel` and `bias` distributions.

Upon being built, this layer adds losses (accessible via the `losses` property) representing the divergences of `kernel` and/or `bias` surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if `kl` is the sum of `losses` for each element of the batch, you should pass `kl / num_examples_per_epoch` to your optimizer).

Examples

We illustrate a Bayesian neural network with variational inference (https://en.wikipedia.org/wiki/Variational_Bayesian_methods), assuming a dataset of `features` and `labels`.

```
import tensorflow_probability as tfp

model = tf.keras.Sequential([
    tfp.layers.DenseFlipout(512, activation=tf.nn.relu),
    tfp.layers.DenseFlipout(10),
])

logits = model(features)
neg_log_likelihood = tf.nn.softmax_cross_entropy_with_logits(
    labels=labels, logits=logits)
kl = sum(model.losses)
loss = neg_log_likelihood + kl
train_op = tf.train.AdamOptimizer().minimize(loss)
```

It uses the Flipout gradient estimator to minimize the Kullback-Leibler divergence up to a constant, also known as the negative Evidence Lower Bound. It consists of the sum of two terms: the expected negative log-likelihood, which we approximate via Monte Carlo; and the KL divergence, which is added via regularizer terms which are arguments to the layer.

References

[1]: Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. Flipout: Efficient Pseudo-Independent Weight Perturbations on Mini-Batches. In *International Conference on Learning Representations*, 2018. https://arxiv.org/abs/1803.04386 (https://arxiv.org/abs/1803.04386)

Args

`units`	Integer or Long, dimensionality of the output space.
`activation`	Activation function (`callable`). Set it to None to maintain a linear activation.
`activity_regularizer`	Regularizer function for the output.
`kernel_posterior_fn`	Python `callable` which creates `tfd.Distribution` instance representing the surrogate posterior of the `kernel` parameter. Default value: `default_mean_field_normal_fn()`.
`kernel_posterior_tensor_fn`	Python `callable` which takes a `tfd.Distribution` instance and returns a representative value. Default value: `lambda d: d.sample()`.
`kernel_prior_fn`	Python `callable` which creates `tfd` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `tfd.Normal(loc=0., scale=1.)`.
`kernel_divergence_fn`	Python `callable` which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd.Distribution`-like instances and the sample is a `Tensor`.
`bias_posterior_fn`	Python `callable` which creates `tfd.Distribution` instance representing the surrogate posterior of the `bias` parameter. Default value: `default_mean_field_normal_fn(is_singular=True)` (which creates an instance of `tfd.Deterministic`).
`bias_posterior_tensor_fn`	Python `callable` which takes a `tfd.Distribution` instance and returns a representative value. Default value: `lambda d: d.sample()`.
`bias_prior_fn`	Python `callable` which creates `tfd` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `None` (no prior, no variational inference)
`bias_divergence_fn`	Python `callable` which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd.Distribution`-like instances and the sample is a `Tensor`.
`seed`	Python scalar `int` which initializes the random number generator. Default value: `None` (i.e., use global seed).

Attributes	
`activity_regularizer`	Optional regularizer function for the output of this layer.

`compute_dtype`	The dtype of the layer's computations. This is equivalent to `Layer.dtype_policy.compute_dtype`. Unless mixed precision is used, this is the same as [`Layer.dtype`](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#dtype) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#dtype), the dtype of the weights. Layers automatically cast their inputs to the compute dtype, which causes computations and the output to be in the compute dtype as well. This is done by the base Layer class in [`Layer.call`](https://www.tensorflow.org/probability/oryx/api_docs/python/oryx/core/state/Module#__call__) (https://www.tensorflow.org/probability/oryx/api_docs/python/oryx/core/state/Module#__call__), so you do not have to insert these casts if implementing your own layer. Layers often perform certain internal computations in higher precision when `compute_dtype` is float16 or bfloat16 for numeric stability. The output will still typically be float16 or bfloat16 in such cases.
`dtype`	The dtype of the layer weights. This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless mixed precision is used, this is the same as [`Layer.compute_dtype`](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#compute_dtype) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#compute_dtype), the dtype of the layer's computations.
`dtype_policy`	The dtype policy associated with this layer. This is an instance of a [`tf.keras.mixed_precision.Policy`](https://www.tensorflow.org/api_docs/python/tf/keras/mixed_precision/Policy) (https://www.tensorflow.org/api_docs/python/tf/keras/mixed_precision/Policy).
`dynamic`	Whether the layer is dynamic (eager-only); set in the constructor.
`input`	Retrieves the input tensor(s) of a layer. Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.
`input_spec`	`InputSpec` instance(s) describing the input format for this layer. When you create a layer subclass, you can set `self.input_spec` to enable the layer to run input compatibility checks when it is called. Consider a `Conv2D` layer: it can only be called on a single input tensor of rank 4. As such, you can set, in `__init__()`:

```
self.input_spec = tf.keras.layers.InputSpec(ndim=4)
```

Now, if you try to call the layer on an input that isn't rank 4 (for instance, an input of shape `(2,)`, it will raise a nicely-formatted error:

```
ValueError: Input 0 of layer conv2d is incompatible
expected ndim=4, found ndim=1. Full shape received:
```

Input checks that can be specified via **`input_spec`** include:

- Structure (e.g. a single input, a list of 2 inputs, etc)
- Shape
- Rank (ndim)
- Dtype

For more information, see [**`tf.keras.layers.InputSpec`**](https://www.tensorflow.org/api_docs/python/tf/keras/layers/InputSpec) (https://www.tensorflow.org/api_docs/python/tf/keras/layers /InputSpec)
.

`losses`

List of losses added using the **`add_loss()`** API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing **`losses`** under a [**`tf.GradientTape`**](https://www.tensorflow.org/api_docs/python/tf/GradientTape) (https://www.tensorflow.org/api_docs/python/tf/GradientTape) will propagate gradients back to the corresponding variables.

```
>>> class MyLayer(tf.keras.layers.Layer):
...   def call(self, inputs):
...     self.add_loss(tf.abs(tf.reduce_mean(inputs))
...     return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
```

```
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

`metrics`	List of metrics added using the **`add_metric()`** API.
	`>>> input = tf.keras.layers.Input(shape=(3,))`
	`>>> d = tf.keras.layers.Dense(2)`
	`>>> output = d(input)`
	`>>> d.add_metric(tf.reduce_max(output), name='max')`
	`>>> d.add_metric(tf.reduce_min(output), name='min')`
	`>>> [m.name for m in d.metrics]`
	`['max', 'min']`

`name`	Name of the layer (string), set in the constructor.
`name_scope`	Returns a **`tf.name_scope`** (https://www.tensorflow.org/api_docs/python/tf/name_scope) instance for this class.
`non_trainable_weights`	List of all non-trainable weights tracked by this layer. Non-trainable weights are *not* updated during training. They are expected to be updated manually in **`call()`**.
`output`	Retrieves the output tensor(s) of a layer. Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.
`submodules`	Sequence of all sub-modules.

	Submodules are modules which are properties of this module, or found as properties of modules which are properties of this module (and so on).
	
	`>>> a = tf.Module()`
	`>>> b = tf.Module()`
	`>>> c = tf.Module()`
	`>>> a.b = b`
	`>>> b.c = c`
	`>>> list(a.submodules) == [b, c]`
	`True`
	`>>> list(b.submodules) == [c]`
	`True`
	`>>> list(c.submodules) == []`
	`True`

`supports_masking`	Whether this layer supports computing a mask using **`compute_mask`**.
`trainable`	
`trainable_weights`	List of all trainable weights tracked by this layer. Trainable weights are updated via gradient descent during training.
`variable_dtype`	Alias of **`Layer.dtype`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer#dtype), the dtype of the weights.
`weights`	Returns the list of all layer variables/weights.

Methods

`add_loss`

```
add_loss(
    losses, **kwargs
)
```

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing the same layer on different inputs `a` and `b`, some entries in `layer.losses` may be dependent on `a` and some on `b`. This method automatically keeps track of dependencies.

This method can be used inside a subclassed layer or model's `call` function, in which case `losses` should be a Tensor or list of Tensors.

Example:

```
class MyLayer(tf.keras.layers.Layer):
  def call(self, inputs):
    self.add_loss(tf.abs(tf.reduce_mean(inputs)))
    return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's `Input`s. These losses become part of the model's topology and are tracked in `get_config`.

Example:

```
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
# Activity regularization.
model.add_loss(tf.abs(tf.reduce_mean(x)))
```

If this is not the case for your loss (if, for example, your loss references a `Variable` of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example:

```
inputs = tf.keras.Input(shape=(10,))
d = tf.keras.layers.Dense(10)
x = d(inputs)
outputs = tf.keras.layers.Dense(1)(x)
```

```
model = tf.keras.Model(inputs, outputs)
# Weight regularization.
model.add_loss(lambda: tf.reduce_mean(d.kernel))
```

Args	
losses	Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-argument callables which create a loss tensor.
****kwargs**	Additional keyword arguments for backward compatibility. Accepted values: inputs - Deprecated, will be automatically inferred.

add_metric

```
add_metric(
    value, name=None, **kwargs
)
```

Adds metric tensor to the layer.

This method can be used inside the `call()` method of a subclassed layer or model.

```
class MyMetricLayer(tf.keras.layers.Layer):
  def __init__(self):
    super(MyMetricLayer, self).__init__(name='my_metric_layer')
    self.mean = tf.keras.metrics.Mean(name='metric_1')

  def call(self, inputs):
    self.add_metric(self.mean(inputs))
    self.add_metric(tf.reduce_sum(inputs), name='metric_2')
    return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's `Input`s. These metrics become part of the model's topology and are tracked when you save the model via `save()`.

```
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
```

```
model = tf.keras.Model(inputs, outputs)
model.add_metric(math_ops.reduce_sum(x), name='metric_1')
```

Note: Calling `add_metric()` with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')
```

Args	
`value`	Metric tensor.
`name`	String metric name.
`**kwargs`	Additional keyword arguments for backward compatibility. Accepted values: `aggregation` - When the `value` tensor provided is not the result of calling a `keras.Metric` instance, it will be aggregated by default using a `keras.Metric.Mean`.

build

View source
 (https://github.com/tensorflow/probability/blob/v0.16.0/tensorflow_probability/python/layers/dense_variational.py#L129-L166)

```
build(
    input_shape
)
```

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of `Layer` or `Model` can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of `call()`.

This is typically used to create the weights of `Layer` subclasses (at the discretion of the subclass implementer).

Args	
`input_shape`	Instance of `TensorShape`, or list of instances of `TensorShape` if the layer expects a list of inputs (one instance per input).

`compute_mask`

```
compute_mask(
    inputs, mask=None
)
```

Computes an output mask tensor.

Args	
`inputs`	Tensor or list of tensors.
`mask`	Tensor or list of tensors.

Returns
None or a tensor (or list of tensors, one per output tensor of the layer).

`compute_output_shape`

View source
 (https://github.com/tensorflow/probability/blob/v0.16.0/tensorflow_probability/python/layers/dense_variational.py#L189-L209)

```
compute_output_shape(
    input_shape
)
```

Computes the output shape of the layer.

Args

Args	
`input_shape`	Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns	
`output_shape`	A tuple representing the output shape.

Raises	
`ValueError`	If innermost dimension of `input_shape` is not defined.

`count_params`

```
count_params()
```

Count the total number of scalars composing the weights.

Returns
An integer count.

Raises	
`ValueError`	if the layer isn't yet built (in which case its weights aren't yet defined).

`from_config`

View source
(https://github.com/tensorflow/probability/blob/v0.16.0/tensorflow_probability/python/layers/dense_variational.py#L252-L283)

```
@classmethod
from_config(
    config
)
```

Creates a layer from its config.

This method is the reverse of `get_config`, capable of instantiating the same layer from the config dictionary.

Args	
`config`	A Python dictionary, typically the output of `get_config`.

Returns	
`layer`	A layer instance.

`get_config`

View source
(https://github.com/tensorflow/probability/blob/v0.16.0/tensorflow_probability/python/layers/dense_variational.py#L703-L718)

```
get_config()
```

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

Returns	
`config`	A Python dictionary of class keyword arguments and their serialized values.

`get_weights`

```
get_weights()
```

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a `Dense` layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another `Dense` layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Returns
Weights values as a list of NumPy arrays.

`set_weights`

```
set_weights(
    weights
)
```

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a `Dense` layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another `Dense` layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Args	
`weights`	a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of `get_weights`).

Raises	
`ValueError`	If the provided weights list does not match the layer's specifications.

`with_name_scope`

```
@classmethod
with_name_scope(
    method
)
```

Decorator to automatically enter the module name scope.

```
>>> class MyModule(tf.Module):
...   @tf.Module.with_name_scope
...   def __call__(self, x):
...     if not hasattr(self, 'w'):
...       self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
...     return tf.matmul(x, self.w)
```

Using the above module would produce **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable)s and **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor)s whose names included the module name:

```
>>> mod = MyModule()
>>> mod(tf.ones([1, 2]))
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32)>
>>> mod.w
<tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
numpy=..., dtype=float32)>
```

Args	
`method`	The method to wrap.

Returns
The original method wrapped such that it enters the module's name scope.

`__call__`

```
__call__(
    *args, **kwargs
)
```

Wraps `call`, applying pre- and post-processing steps.

Args	
`*args`	Positional arguments to be passed to `self.call`.
`**kwargs`	Keyword arguments to be passed to `self.call`.

Returns
Output tensor(s).

Note:

- The following optional keyword arguments are reserved for specific uses:
    - `training`: Boolean scalar tensor of Python boolean indicating whether the `call` is meant for training or inference.
    - `mask`: Boolean input mask.
- If the layer's `call` method takes a `mask` argument (as some Keras layers do), its default value will be set to the mask generated for `inputs` by the previous layer (if `input` did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support.
- If the layer is not built, the method will call `build`.

Raises	
`ValueError`	if the layer's `call` method returns None (an invalid value).
`RuntimeError`	if `super().__init__()` was not called in the constructor.