Bayesian ML

A practical approach

Verena, Filip

Motivation

Key issue of ML: **overfitting**

→ solution at hand: regularisation (dropout,…)

Became concern for us with **Model-based Reinforcement Learning**.

Disclaimer: only in the research phase. No successful application for RL yet.

Filip will show you another application.

Overfitting

Example regression:

NN is a probabilistic model $p(y \mid \mathbf{x}, \mathbf{w})$, p is a Gaussian distribution.

With training data $\mathscr{D} = \{x^{(i)}, y^{(i)}\}$ optimise $\mathbf{w}$ by maximising likelihood function $p(\mathscr{D} \mid \mathbf{w}) = \prod_i p(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w})$… Maximum Likelihood Estimate $\rightarrow$ point estimate of $\mathbf{w}$

Better: regularisation

Would like to maximise $p(\mathbf{w} \mid \mathscr{D})$ …maximum a posteriori estimate of $\mathbf{w}$

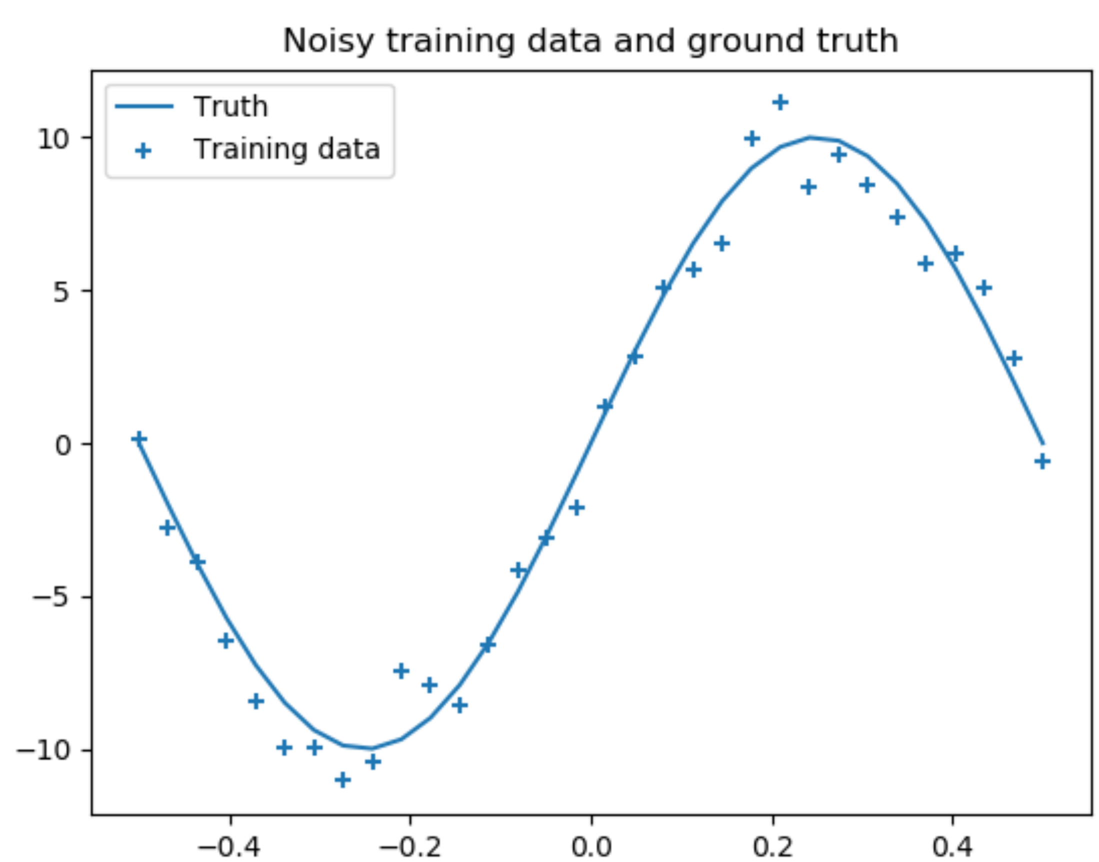
Do this with $p(\mathbf{w} \mid \mathscr{D}) \propto p(\mathscr{D} \mid \mathbf{w}) p(\mathbf{w})$… prior distribution $p(\mathbf{w})$.

$\rightarrow$ still only point estimate of $\mathbf{w}$

Example

Dataset is 32 points of noisy sinusoidal function.

```python
def f(x, sigma):
    epsilon = np.random.randn(*x.shape) * sigma
    return 10 * np.sin(2 * np.pi * (x)) + epsilon

train_size = 32
noise = 1.0

X = np.linspace(-0.5, 0.5, train_size).reshape(-1, 1)
y = f(X, sigma=noise)
y_true = f(X, sigma=0.0)
```



Example

Simple feedforward regression model with regularisation with TensorFlow 2

```
import tensorflow.keras as tfk
from tensorflow.keras import initializers, activations
from tensorflow.keras.layers import Layer
import tensorflow_probability as tfp
import tensorflow as tf
tfd = tfp.distributions

```

Model

```
batch_size = train_size
num_batches = train_size / batch_size



# Build model.
model = tf.keras.Sequential([
    tfk.layers.Input(shape=(1,)),
    tfk.layers.Dense(20, activation="relu"),
    tfk.layers.Dropout(0.1),
    tfk.layers.Dense(20, activation="relu"),
    tfk.layers.Dropout(0.1),
tfk.layers.Dense(1)
,tfk.layers.Dropout(0.1)
])



from tensorflow.keras import callbacks, optimizers

def neg_log_likelihood(y_obs, y_pred, sigma=noise):
    dist = tfp.distributions.Normal(loc=y_pred, scale=sigma)
    return tfk.backend.sum(-dist.log_prob(y_obs))

model.compile(loss=neg_log_likelihood, optimizer=optimizers.Adam(lr=0.08), metrics=['mse'])
model.fit(X, y, batch_size=batch_size, epochs=1500, verbose=1);

```

Assume $p(y\,|\,\mathbf{x}, \mathbf{w})$ is Gaussian

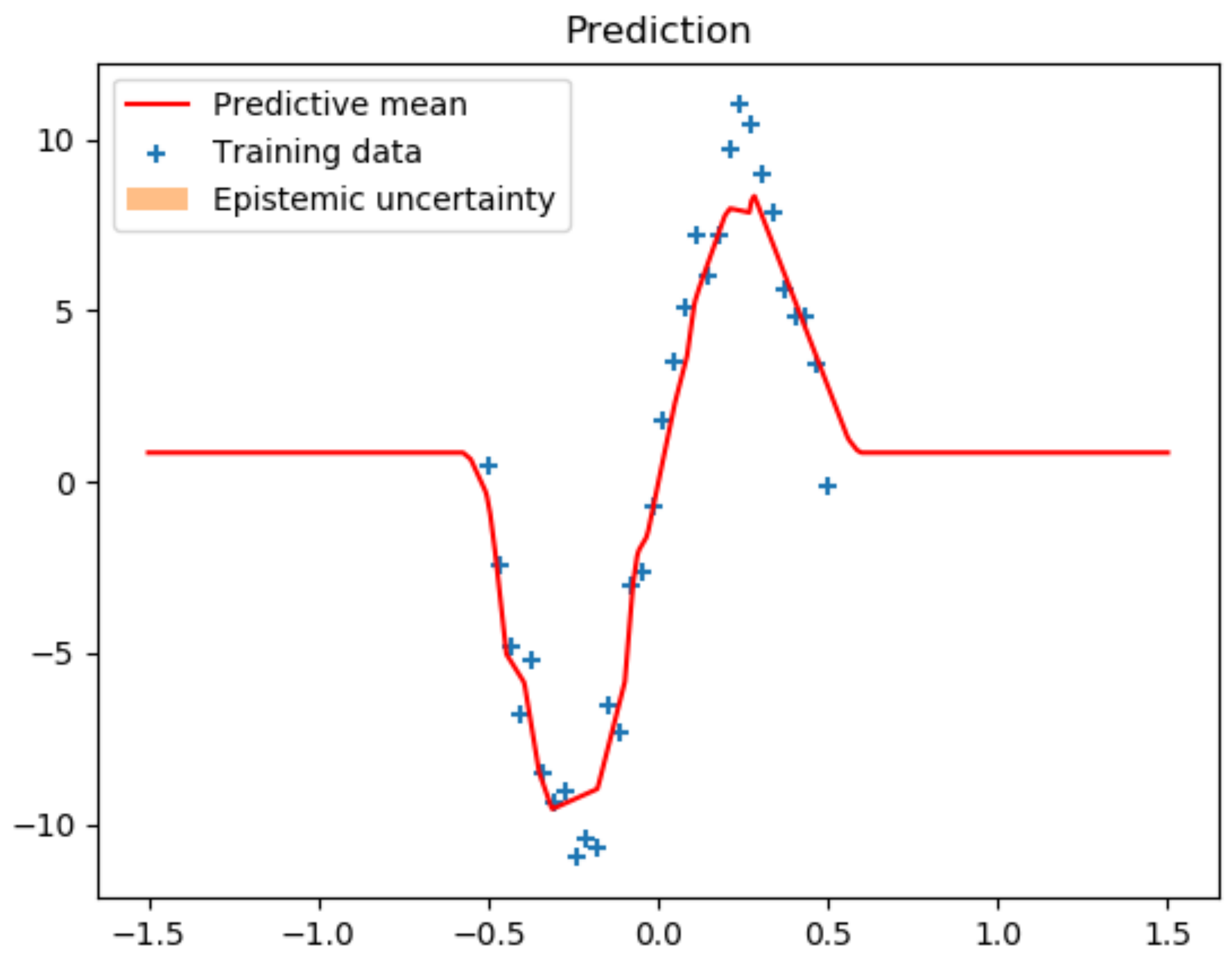
Example

Test of model:

```python

X_test = np.linspace(-1.5, 1.5, 1000).reshape(-1, 1)
y_pred_list = []

import tqdm

for i in tqdm.tqdm(range(500)):
    y_pred = model.predict(X_test)
    y_pred_list.append(y_pred)

y_preds = np.concatenate(y_pred_list, axis=1)

y_mean = np.mean(y_preds, axis=1)
y_sigma = np.std(y_preds, axis=1)

plt.plot(X_test, y_mean, 'r-', label='Predictive mean');
plt.scatter(X, y, marker='+', label='Training data')
plt.fill_between(X_test.ravel(),
                 y_mean + 2 * y_sigma,
                 y_mean - 2 * y_sigma,
                 alpha=0.5, label='Epistemic uncertainty')
plt.title('Prediction')
plt.legend();

plt.show()
```

Example

Result:



Can we trust this? Need **epistemic uncertainty** on predictions.

Towards Variational Layers

With regularisation already assume distributions of $\mathbf{w}$…how to extract uncertainty on predictions of NN?

→ Average predictions over ensemble of NNs weighted by posterior probabilities of their $\mathbf{w}$

$$p(y \mid \mathbf{x}, \mathscr{D}) = \int p(y \mid \mathbf{x}, \mathbf{w}) p(\mathbf{w} \mid \mathscr{D}) d\mathbf{w}$$

But we cannot write $p(\mathbf{w} \mid \mathscr{D})$ down.

→Use variational distribution $q(\mathbf{w} \mid \theta)$ of known functional form. Need to find θ by minimising KL divergence between $p(\mathbf{w} \mid \mathscr{D})$ and $q(\mathbf{w} \mid \theta)$

Towards Variational Layers

Resulting cost function of KL divergence minimisation can be arranged as:

$$\mathscr{F}(\mathscr{D},\theta) = \underbrace{\mathbb{E}_{q(\mathbf{w}|\theta)} \log q(\mathbf{w}\,|\,\theta) - \mathbb{E}_{q(\mathbf{w}|\theta)} \log p(\mathbf{w})}_{\textbf{Data independent}} - \mathbb{E}_{q(\mathbf{w}|\theta)} \log p(\mathscr{D}\,|\,\mathbf{w})$$

See ELBO in *https://medium.com/tensorflow/regression-with-probabilistic-layers-in-tensorflow-probability-e46ff5d37baf*

Constructing a Variational Layer yourself

Assume Gaussian as variational posterior for $\mathbf{w}$

$\rightarrow \theta = (\mu, \sigma)$

Assume non-trainable prior:

$$p(\mathbf{w}) = \pi \mathcal{N}(\mathbf{w} \,|\, 0, \sigma_1) + (1 - \pi)\mathcal{N}(\mathbf{w} \,|\, 0, \sigma_2)$$

Basic steps:

- define your trainable parameters in `build()`

- define in `call()` how to evaluate your layer given `input`

Constructing a Variational Layer yourself

Example:

```python
class MyDenseVariational(Layer):
    def __init__(self,
                 units,
                 kl_weight,
                 activation=None,
                 prior_sigma_1=1.5,
                 prior_sigma_2=0.1,
                 prior_pi=0.5, **kwargs):
        self.units = units
        self.kl_weight = kl_weight
        self.activation = activations.get(activation)
        self.prior_sigma_1 = prior_sigma_1
        self.prior_sigma_2 = prior_sigma_2
        self.prior_pi_1 = prior_pi
        self.prior_pi_2 = 1.0 - prior_pi
        self.init_sigma = np.sqrt(self.prior_pi_1 * self.prior_sigma_1 ** 2 +
                                  self.prior_pi_2 * self.prior_sigma_2 ** 2)

        super().__init__(**kwargs)

    def compute_output_shape(self, input_shape):...

    def build(self, input_shape):...

    def call(self, inputs, **kwargs):...

    def kl_loss(self, w, mu, sigma):...

    def log_prior_prob(self, w):...
```

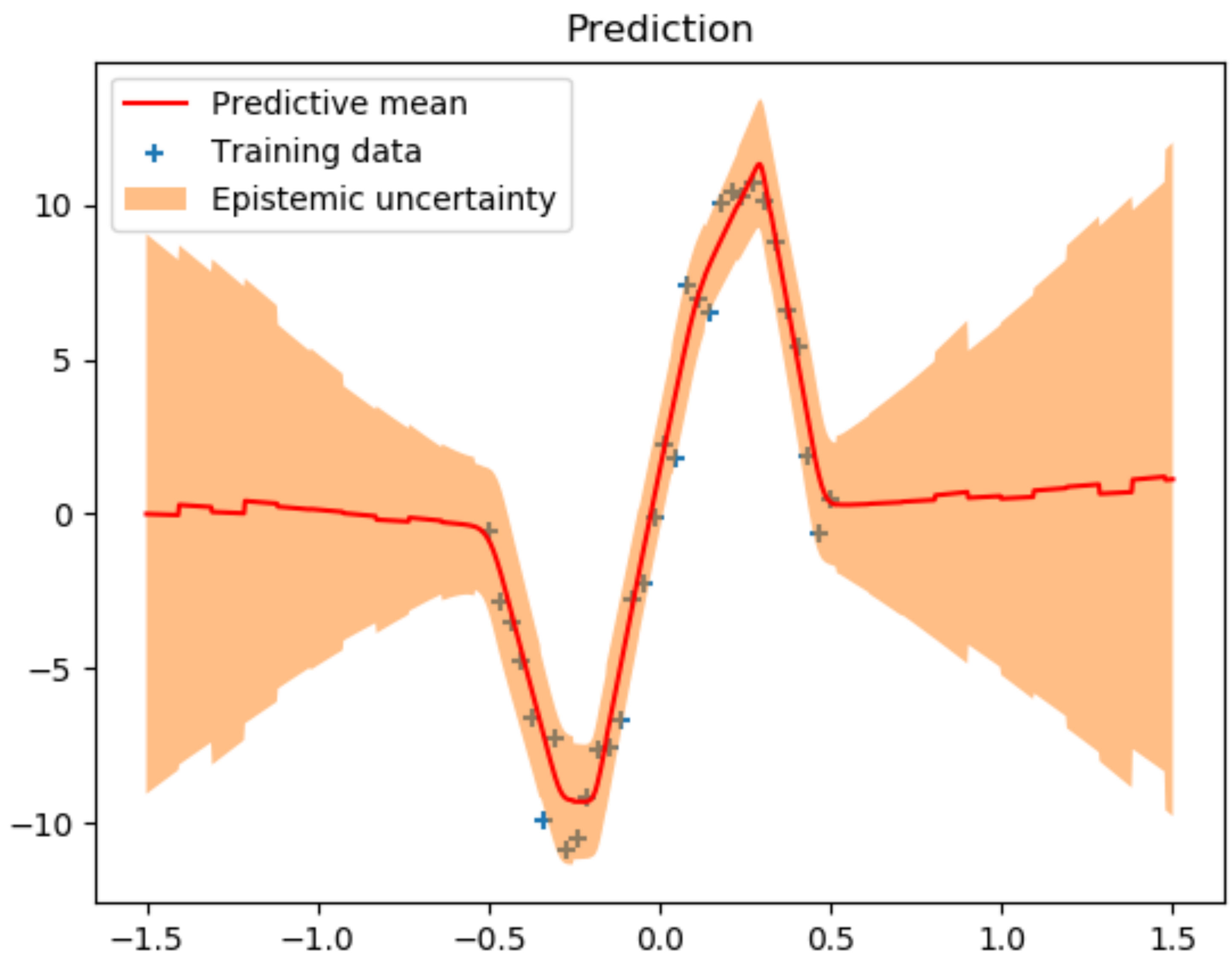
Constructing a Variational Layer yourself

For the details of `build()` and `call()`:

```python
    def build(self, input_shape):
        self.kernel_mu = self.add_weight(name='kernel_mu',
                                         shape=(input_shape[1], self.units),
                                         initializer=initializers.RandomNormal(stddev=self.init_sigma),
                                         trainable=True)
        self.bias_mu = self.add_weight(name='bias_mu',
                                       shape=(self.units,),
                                       initializer=initializers.RandomNormal(stddev=self.init_sigma),
                                       trainable=True)
        self.kernel_rho = self.add_weight(name='kernel_rho',
                                          shape=(input_shape[1], self.units),
                                          initializer=initializers.constant(0.0),
                                          trainable=True)
        self.bias_rho = self.add_weight(name='bias_rho',
                                        shape=(self.units,),
                                        initializer=initializers.constant(0.0),
                                        trainable=True)
        super().build(input_shape)

    def call(self, inputs, **kwargs):
        kernel_sigma = tf.math.softplus(self.kernel_rho)
        kernel = self.kernel_mu + kernel_sigma * tf.random.normal(self.kernel_mu.shape)

        bias_sigma = tf.math.softplus(self.bias_rho)
        bias = self.bias_mu + bias_sigma * tf.random.normal(self.bias_mu.shape)

        self.add_loss(self.kl_loss(kernel, self.kernel_mu, kernel_sigma) +
                      self.kl_loss(bias, self.bias_mu, bias_sigma))

        return self.activation(tfk.backend.dot(inputs, kernel) + bias)

    def kl_loss(self, w, mu, sigma):
        variational_dist = tfp.distributions.Normal(mu, sigma)
        return self.kl_weight * tfk.backend.sum(variational_dist.log_prob(w) - self.log_prior_prob(w))
```

Using MyDenseVariational

```python
kl_weight = 1.0 / num_batches
prior_params = {
    'prior_sigma_1': 1.5,
    'prior_sigma_2': 0.1,
    'prior_pi': 0.5
}

x_in = Input(shape=(1,))
x = MyDenseVariational(20, kl_weight, **prior_params, activation='relu')(x_in)
x = MyDenseVariational(20, kl_weight, **prior_params, activation='relu')(x)
x = MyDenseVariational(1, kl_weight, **prior_params)(x)

model = Model(x_in, x)
```



Using TF2 DenseVariational Layer

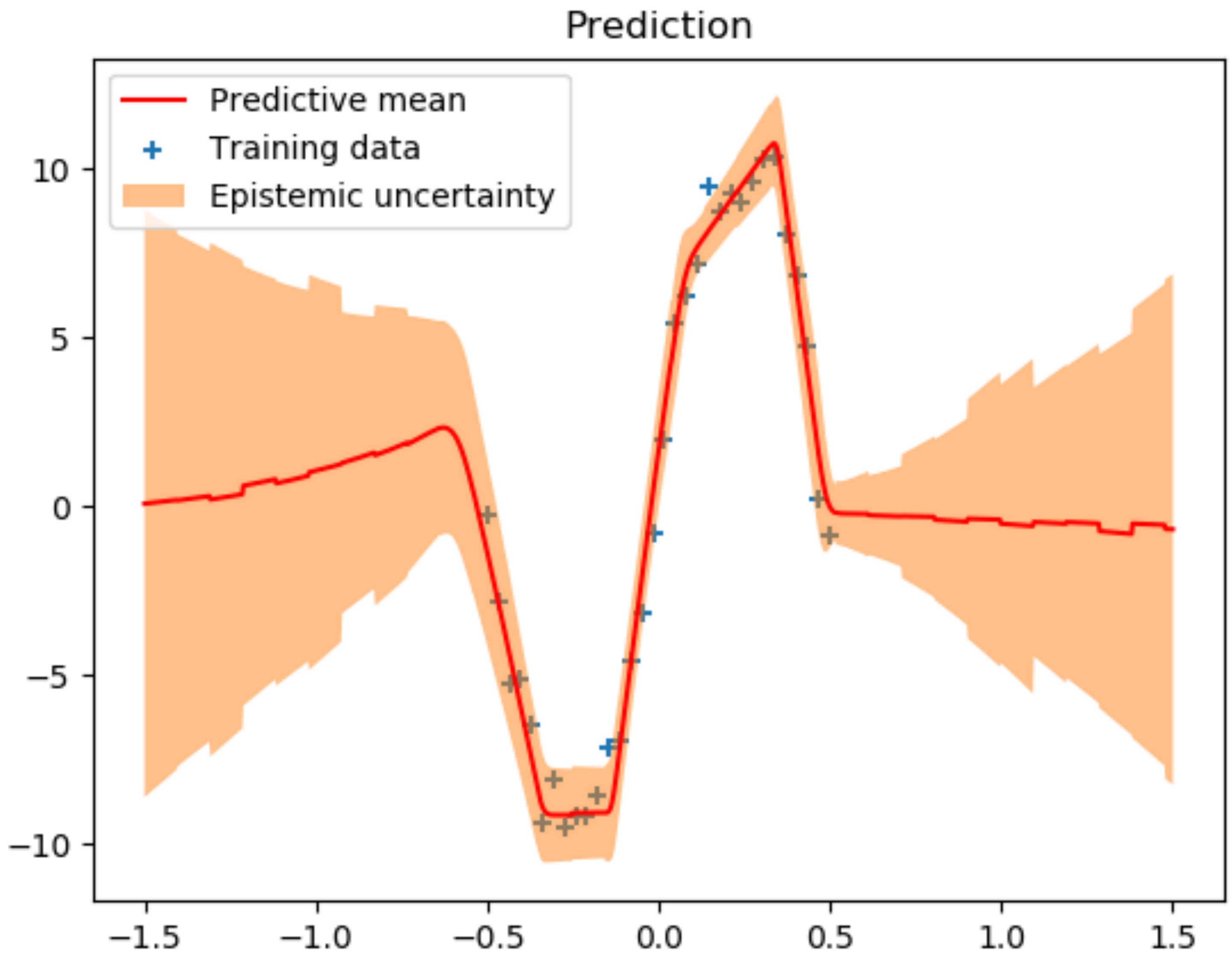
```
# Build model.
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1,)),
    tfp.layers.DenseVariational(units=20,
                                make_posterior_fn=posterior_mean_field,
                                make_prior_fn=prior_trainable,
                                kl_weight=kl_weight,
                                activation='relu'),
    tfp.layers.DenseVariational(units=20,
                                make_posterior_fn=posterior_mean_field,
                                make_prior_fn=prior_trainable,
                                kl_weight=kl_weight,
                                activation='relu'),
    tfp.layers.DenseVariational(units=1,
                                make_posterior_fn=posterior_mean_field,
                                make_prior_fn=prior_trainable,
                                kl_weight=kl_weight)
])

```

```
# Specify the surrogate posterior over `keras.layers.Dense` `kernel` and `bias`.
def posterior_mean_field(kernel_size, bias_size=0, dtype=None):
    n = kernel_size + bias_size
    c = np.log(np.expm1(1.))
    return tf.keras.Sequential([
        tfp.layers.VariableLayer(2 * n, dtype=dtype),
        tfp.layers.DistributionLambda(lambda t: tfd.Independent(
            tfd.Normal(loc=t[..., :n],
                       scale=1e-5 + tf.nn.softplus(c+t[..., n:])),
            reinterpreted_batch_ndims=1)),
    ])
```

```
# Specify the prior over `keras.layers.Dense` `kernel` and `bias`.
def prior_trainable(kernel_size, bias_size=0, dtype=None):
    n = kernel_size + bias_size
    return tf.keras.Sequential([
        tfp.layers.VariableLayer(n, dtype=dtype),
        tfp.layers.DistributionLambda(lambda t: tfd.Independent(
            tfd.Normal(loc=t, scale=1),
            reinterpreted_batch_ndims=1)),
    ])
```

Using TF2 DenseVariational Layer

Result:



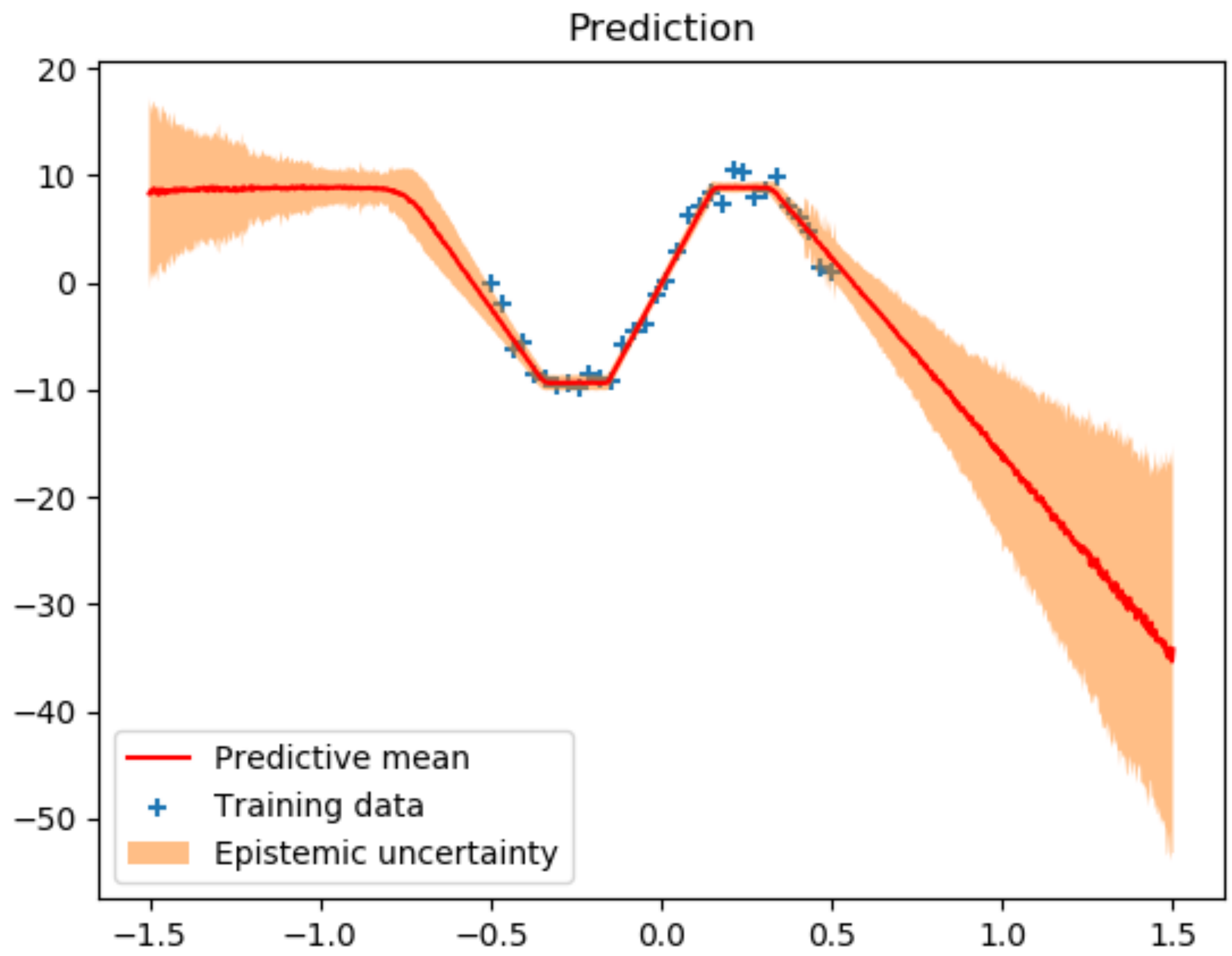
DenseFlipout

Assume distribution of weights and biases.

→ prior and posterior distributions for those to be defined.

Use MonteCarlo approach for sampling weights and biases and integrate over it.

```python
# Build model.
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1,)),
    tfp.layers.DenseFlipout(20, activation="relu"),
    tfp.layers.DenseFlipout(20, activation="relu"),
tfp.layers.DenseFlipout(1)

])
```



Further reading…

https://brendanhasz.github.io/2019/07/23/bayesian-density-net.html

https://medium.com/tensorflow/regression-with-probabilistic-layers-in-tensorflow-probability-e46ff5d37baf

https://github.com/krasserm/bayesian-machine-learning