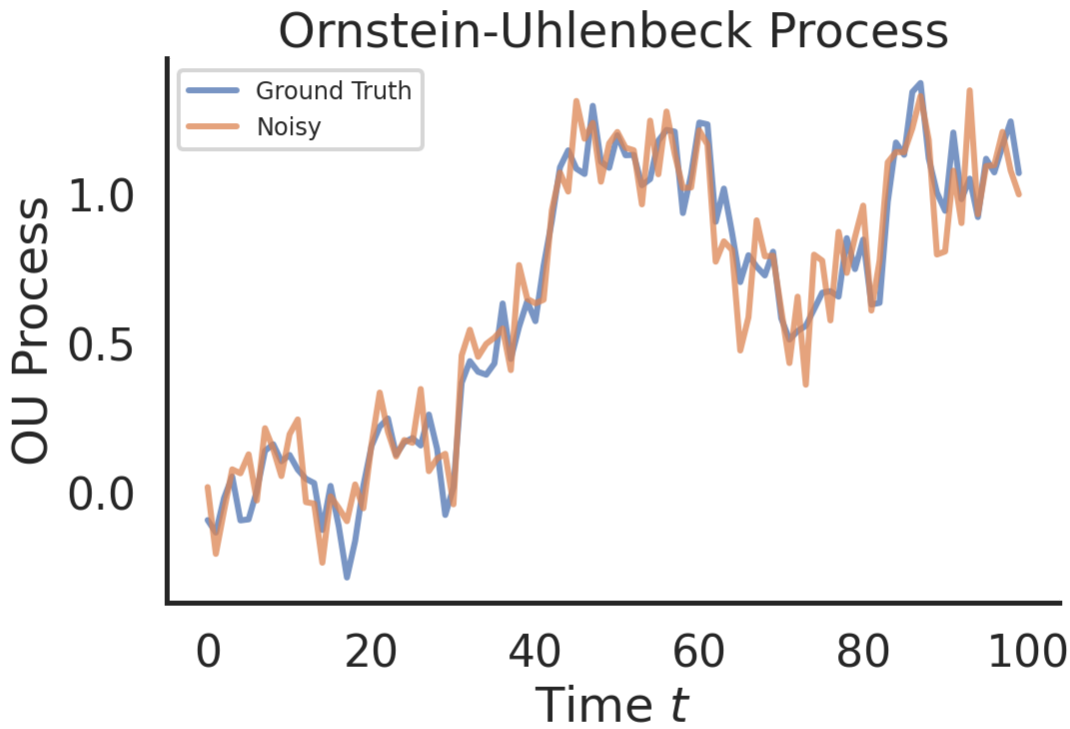
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import math
import time
import numpy as onp
import jax.numpy as np
from jax import grad, jit, vmap, value_and_grad
from jax import random
from jax.experimental import stax
from jax.experimental.stax import (BatchNorm, Conv, Dense, Flatten,
                                   Relu, LogSoftmax)
from jax.experimental import optimizers


# Generate key which is used to generate random numbers
key = random.PRNGKey(1)


from jax.nn import sigmoid
from jax.nn.initializers import glorot_normal, normal

from functools import partial
from jax import lax

def GRU(out_dim, W_init=glorot_normal(), b_init=normal()):
    def init_fun(rng, input_shape):
        """ Initialize the GRU layer for stax """
        hidden = b_init(rng, (input_shape[0], out_dim))

        k1, k2, k3 = random.split(rng, num=3)
        update_W, update_U, update_b = (
            W_init(k1, (input_shape[2], out_dim)),
            W_init(k2, (out_dim, out_dim)),
            b_init(k3, (out_dim,)),)

        k1, k2, k3 = random.split(rng, num=3)
        reset_W, reset_U, reset_b = (
            W_init(k1, (input_shape[2], out_dim)),
            W_init(k2, (out_dim, out_dim)),
            b_init(k3, (out_dim,)),)

        k1, k2, k3 = random.split(rng, num=3)
        out_W, out_U, out_b = (
            W_init(k1, (input_shape[2], out_dim)),
            W_init(k2, (out_dim, out_dim)),
            b_init(k3, (out_dim,)),)
        # Input dim 0 represents the batch dimension
        # Input dim 1 represents the time dimension (before scan moveaxis)
        output_shape = (input_shape[0], input_shape[1], out_dim)
        return (output_shape,
```

```
        return (output_shape,
            (hidden,
             (update_W, update_U, update_b),
             (reset_W, reset_U, reset_b),
             (out_W, out_U, out_b),),)

    def apply_fun(params, inputs, **kwargs):
        """ Loop over the time steps of the input sequence """
        h = params[0]

        def apply_fun_scan(params, hidden, inp):
            """ Perform single step update of the network """
            _, (update_W, update_U, update_b), (reset_W, reset_U, reset_b), (
                out_W, out_U, out_b) = params

            update_gate = sigmoid(np.dot(inp, update_W) +
                                  np.dot(hidden, update_U) + update_b)
            reset_gate = sigmoid(np.dot(inp, reset_W) +
                                 np.dot(hidden, reset_U) + reset_b)
            output_gate = np.multiply(update_gate, hidden) + np.multiply(1-update_gate, np.ta
            return hidden, output_gate

        # Move the time dimension to position 0
        inputs = np.moveaxis(inputs, 1, 0)
        f = partial(apply_fun_scan, params)
        _, h_new = lax.scan(f, h, inputs)
        return h_new

    return init_fun, apply_fun



# Generate & plot a time series generated by the OU process
x_0, mu, tau, sigma, dt = 0, 1, 2, 0.5, 0.1
noise_std = 0.01
num_dims, batch_size = 100, 64  # Number of timesteps in process, 100 time steps, 64 examples

x, x_tilde = generate_ou_process(batch_size, num_dims, mu, tau,
                                 sigma, noise_std, dt)
# x is the ground truth, x_tilde is the noisy version

plot_ou_process(x[0, :], x_tilde[0, :])
```




```
num_dims = 100              # Number of OU timesteps
batch_size = 64             # Batchsize
num_hidden_units = 12       # GRU cells in the RNN layer

# Initialize the network and perform a forward pass
init_fun, gru_rnn = stax.serial(Dense(num_hidden_units), Relu,
                                GRU(num_hidden_units), Dense(1))
_, params = init_fun(key, (batch_size, num_dims, 1))

def mse_loss(params, inputs, targets):
    """ Calculate the Mean Squared Error Prediction Loss. """
    preds = gru_rnn(params, inputs)
    return np.mean((preds-targets)**2)

@jit
def update(params, x, y, opt_state):
    """ Perform a forward pass, calculate the MSE & perform a SGD step. """
    loss, grads = value_and_grad(mse_loss)(params, x, y)
    opt_state = opt_update(0, grads, opt_state)
    return get_params(opt_state), opt_state, loss
```

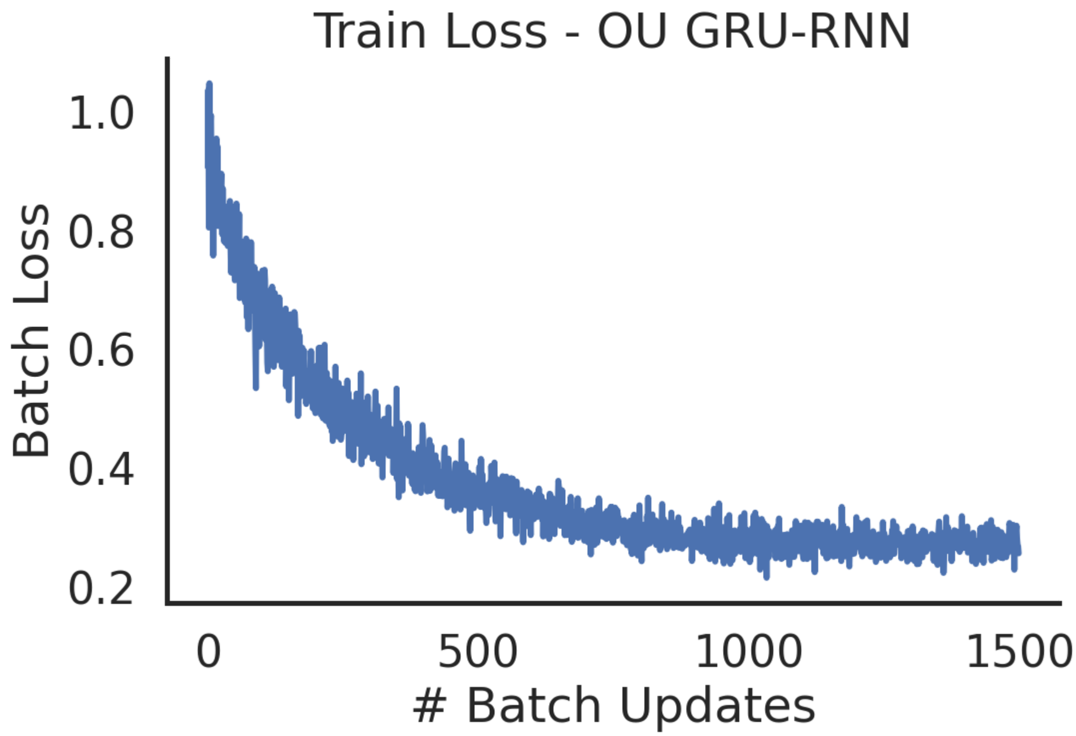
Training the RNN

```
learning_rate = 0.0001
opt_init, opt_update, get_params = optimizers.adam(learning_rate)

opt_state = opt_init(params)
num_batches = 1500
```

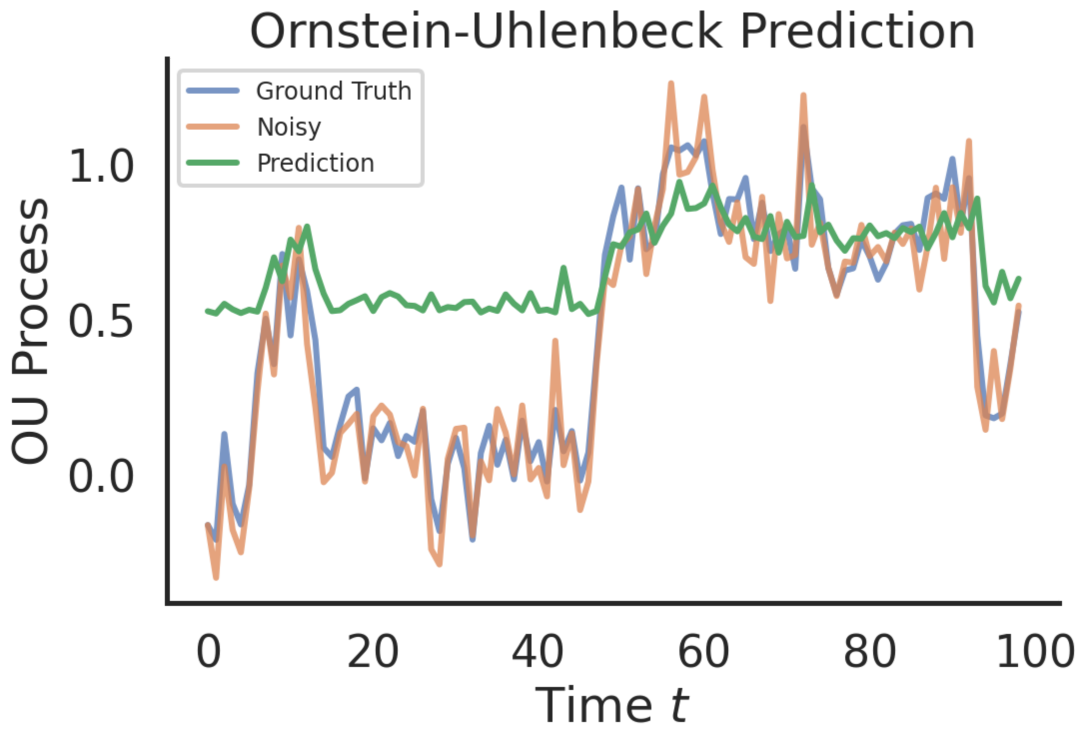
```
train_loss_log = []
start_time = time.time()

for batch_idx in range(num_batches):
    x, x_tilde = generate_ou_process(batch_size, num_dims, mu, tau, sigma, noise_std)
    x_in = x_tilde[:, :(num_dims-1)]
    y = x[:, 1:]
    y = np.array(y)
    x_in = np.expand_dims(x_in, 2)

    #print(x_in.shape)
    #print(y.shape)
    #print(len(params))

    params, opt_state, loss = update(params, x_in, y, opt_state)


    batch_time = time.time() - start_time
    train_loss_log.append(loss)

    if batch_idx % 100 == 0:
        start_time = time.time()
        print("Batch {} | T: {:0.2f} | MSE: {:0.2f} |".format(batch_idx, batch_time, loss))
```

```
Batch 0 | T: 1.41 | MSE: 0.91 |
Batch 100 | T: 3.89 | MSE: 0.70 |
Batch 200 | T: 3.91 | MSE: 0.54 |
Batch 300 | T: 3.85 | MSE: 0.45 |
Batch 400 | T: 3.86 | MSE: 0.43 |
Batch 500 | T: 4.01 | MSE: 0.36 |
Batch 600 | T: 3.92 | MSE: 0.34 |
Batch 700 | T: 3.98 | MSE: 0.31 |
Batch 800 | T: 3.88 | MSE: 0.27 |
Batch 900 | T: 3.90 | MSE: 0.31 |
Batch 1000 | T: 3.92 | MSE: 0.28 |
Batch 1100 | T: 3.87 | MSE: 0.25 |
Batch 1200 | T: 3.94 | MSE: 0.27 |
Batch 1300 | T: 3.94 | MSE: 0.29 |
Batch 1400 | T: 3.96 | MSE: 0.27 |
```

```
# plot the
plot_ou_loss(train_loss_log)
```




```
# Plot a prediction and ground truth OU process
x_true, x_tilde = generate_ou_process(batch_size, num_dims, mu, tau, sigma, noise_std)
x_in = x_tilde[:, :(num_dims-1)]
y = np.array(y)
y = np.array(y)
x_in = np.expand_dims(x_in, 2)
preds = gru_rnn(params, x_in)

y = onp.array(y)
x_true = onp.array(x_true)
x_pred = onp.array(preds)
plot_ou_process(x_true[0, 1:], x_tilde=x_tilde[0, 1:], x_pred=x_pred[:, 0],
               title=r"Ornstein-Uhlenbeck Prediction")
```


Ornstein-Uhlenbeck Prediction
Ground Truth
Noisy
Prediction
OU Process
1.0
0.5
0.0
0
20
40
60
80
100
Time t