# Normalizing Flows

Abdul Fatir

**Abstract**

Notes on normalizing flows.

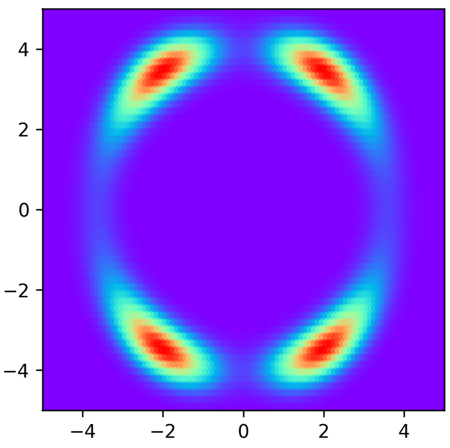
## 0.1 Code for PReLU

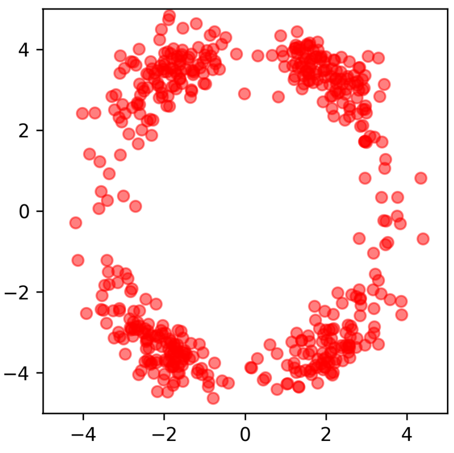
```python
class PReLU(tfb.Bijector):
    def __init__(self, alpha=0.5, validate_args=False, name="p_relu"):
        super(PReLU, self).__init__(
            forward_min_event_ndims=0,
            validate_args=validate_args,
            name=name)
        self.alpha = alpha

    def _forward(self, x):
        return tf.where(tf.greater_equal(x, 0), x, self.alpha * x)

    def _inverse(self, y):
        return tf.where(tf.greater_equal(y, 0), y, 1. / self.alpha * y)

    def _inverse_log_det_jacobian(self, y):
        I = tf.ones_like(y)
        J_inv = tf.where(tf.greater_equal(y, 0), I, 1.0 / self.alpha * I)
        log_abs_det_J_inv = tf.log(tf.abs(J_inv))
        return log_abs_det_J_inv
```

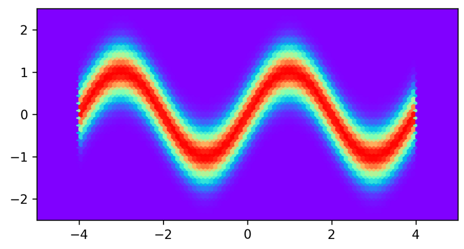
Snippet 1: Parametrized ReLU Bijector

## 0.2 Reproducing Results from Rezende and Mohamed

Reproducibility is an important concern in deep learning, especially because the models are trained stochastically in environments that can be very different from each other. With this is mind, and a lack of official implementation from the authors, I implemented planar flows and conducted a few experiments presented in the paper.


(a) Density 1


(b) Samples from Density 1


(c) Density 2

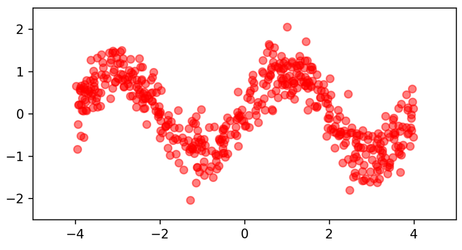

(d) Samples from Density 2

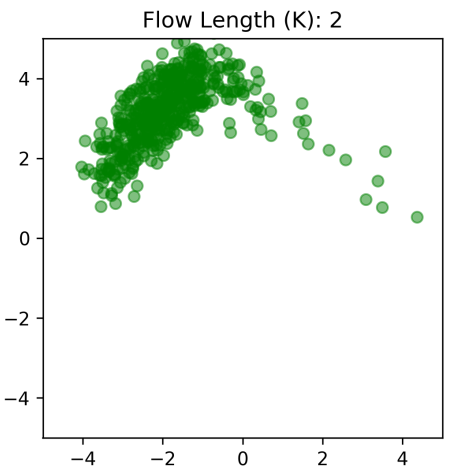
Figure 1: Two probability densities and samples from each density obtained using Metropolis-Hastings

The experiment seeks to approximate two complex 2D densities using planar flows. The unnormalized density functions are given by $p(\mathbf{z}) \propto e^{-U_i(\mathbf{z})}$ where the two energy functions $U_1$ and $U_2$ are

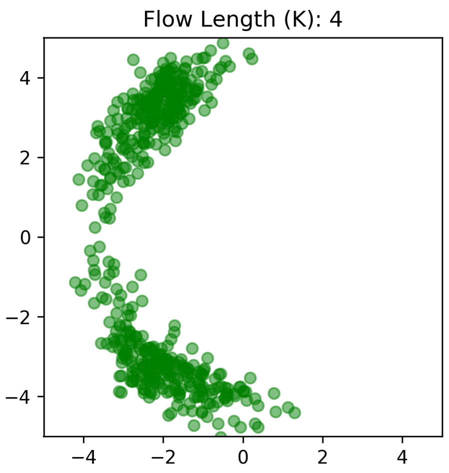
$$U_1(\mathbf{z}) = \frac{1}{2}\left(\frac{\|\mathbf{z}\| - 4}{0.4}\right)^2 - \log\left(e^{-\frac{1}{2}\left[\frac{z_1-2}{0.8}\right]^2} + e^{-\frac{1}{2}\left[\frac{z_1+2}{0.8}\right]^2}\right) \tag{1}$$

$$U_2(\mathbf{z}) = \frac{1}{2}\left[\frac{z_2 - \sin\left(\frac{2\pi z_1}{4}\right)}{0.4}\right]^2 \tag{2}$$

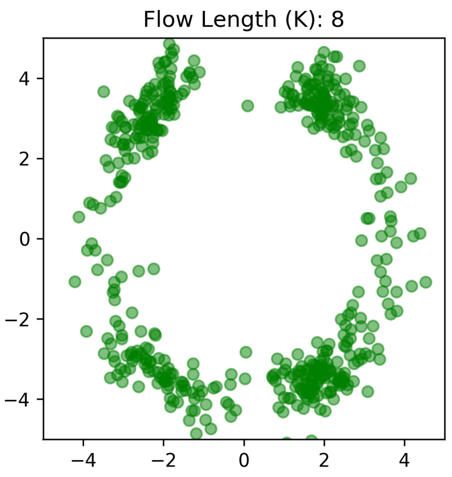
Fig. 1 shows the two densities along with a scatter plot of 500 samples from each density obtained using Metropolis-Hastings (MH) algorithm. The MH algorithm was implemented in `numpy` using the normal distribution with identity covariance matrix as the proposal distribution. A part of the code that implements the MH algorithm is given in Snippet 1. See `Metropolis-Hastings.ipynb` for complete code.



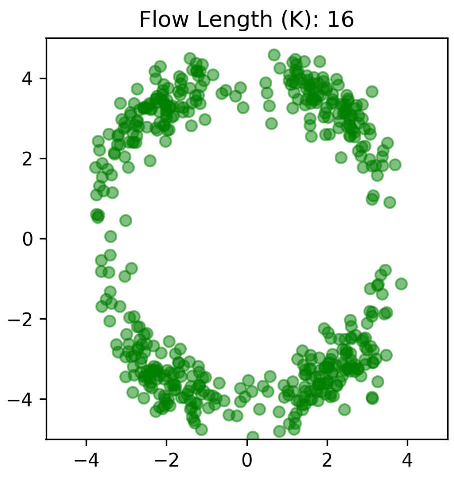
(a) K = 2



(b) K = 4

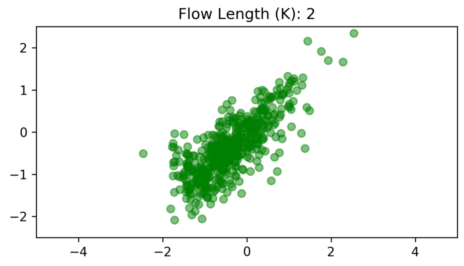


(c) K = 8

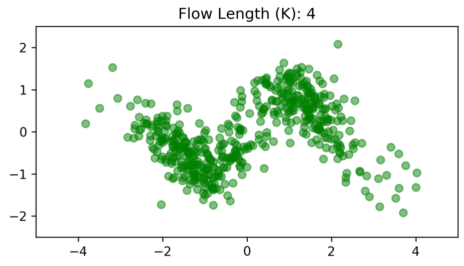


(d) K = 16

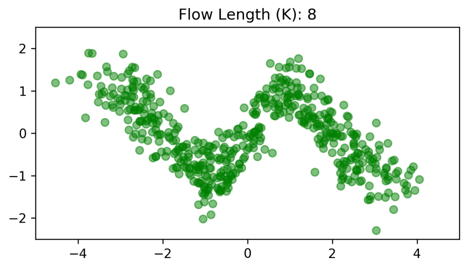
Figure 2: 500 samples from Planar Flows with different flow lengths after 10000 training steps for density 1



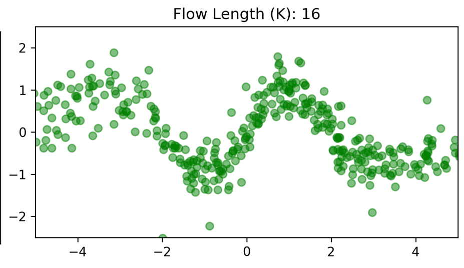
(a) K = 2



(b) K = 4



(c) K = 8



(d) K = 16

Figure 3: 500 samples from Planar Flows with different flow lengths after 5000 training steps for density 2

Planar Flow was implemented in Tensorflow and the code satisfies the invertibility conditions mentioned in the appendix of CITE. Tensorflow Probability (TFP) was not used because it requires implementation of the `_inverse` function which is not available for the function used in planar flows although they are invertible. Flow lengths ($K$) from the set $\{2, 4, 8, 16\}$ were tested and trained using the Adam optimizer. A part of the code that implements planar flow is given in Snippet 2. See `PlanarFlow-Example1.ipynb` and `PlanarFlow-Example2.ipynb` for complete code. Note that constant terms were dropped during the computation of the KL-divergence which may lead to negative KL values.

Figs. 2 and 3 show samples from planar flows of different lengths for density 1 and density 2 respectively at the end of training. It is clear from the results that as flow length increases, the model is able to better approximate the true density. Another interesting aspect encountered during the course of experiments was that the training becomes unstable as length of flow $K$ increases, especially for density 2.

```
def metropolis_hastings(target_density, size=500000):
    burnin_size = 10000
    size += burnin_size
    x0 = np.array([[0, 0]])
    xt = x0
    samples = []
    for i in tqdm(range(size)):
        xt_candidate = np.array(
            [np.random.multivariate_normal(xt[0], np.eye(2))])
        accept_prob = (target_density(xt_candidate)) / (target_density(xt))
        if np.random.uniform(0, 1) < accept_prob:
            xt = xt_candidate
        samples.append(xt)
    samples = np.array(samples[burnin_size:])
    samples = np.reshape(samples, [samples.shape[0], 2])
    return samples
```

Snippet 2: Metropolis-Hastings

```
m = lambda x: -1 + tf.log(1 + tf.exp(x))
h = lambda x: tf.tanh(x)
h_prime = lambda x: 1 - tf.tanh(x) ** 2
base_dist = tfd.MultivariateNormalDiag(loc=[0., 0.], scale_diag=[1., 1.])
z_0 = base_dist.sample(500)
z_prev = z_0
sum_log_det_jacob = 0.
for i in range(K):
    with tf.variable_scope('layer_%d' % i):
        u = tf.get_variable('u', dtype=tf.float32, shape=(1, 2))
        w = tf.get_variable('w', dtype=tf.float32, shape=(1, 2))
        b = tf.get_variable('b', dtype=tf.float32, shape=())
        u_hat = (m(tf.tensordot(w, u, 2)) - tf.tensordot(w, u, 2)) * \
            (w / tf.norm(w)) + u
        affine = h_prime(tf.expand_dims(
            tf.reduce_sum(z_prev * w, -1), -1) + b) * w
        sum_log_det_jacob += tf.log(tf.abs(1 +
                                           tf.reduce_sum(affine * u_hat, -1)))
        z_prev = z_prev + u_hat * \
            h(tf.expand_dims(tf.reduce_sum(z_prev * w, -1), -1) + b)
z_k = z_prev
log_q_k = base_dist.log_prob(z_0) - sum_log_det_jacob
log_p = tf.log(true_density(z_k))

kl = tf.reduce_mean(log_q_k - log_p, -1)
```

Snippet 3: Planar Flow