Inverse Problems – Final Project

Students: Anna Shchukina, Alon Barkan

Paper: Analyzing Inverse Problems With Invertible Neural Networks

Introduction and overview

In our project we will present a data-driven method - one of the recent deep learning developing for solving ill posed inverse problems. More precisely, this model can give us complete posterior distribution of unknowns conditioned on an observation. In other words, we will use Bayesian approach to inverse problems.

The neural network presented in the paper consists of 'coupling blocks' that let the mapping to be bijective and allow explicit computation of posterior probabilities.

Also, researchers incorporate a latent output variable z (concatenated with output y) that follows a certain distribution (standard normal). We can think of this variable as a tool that captures information otherwise lost in the forward process and information about the parameter x that is not contained in the observation y.

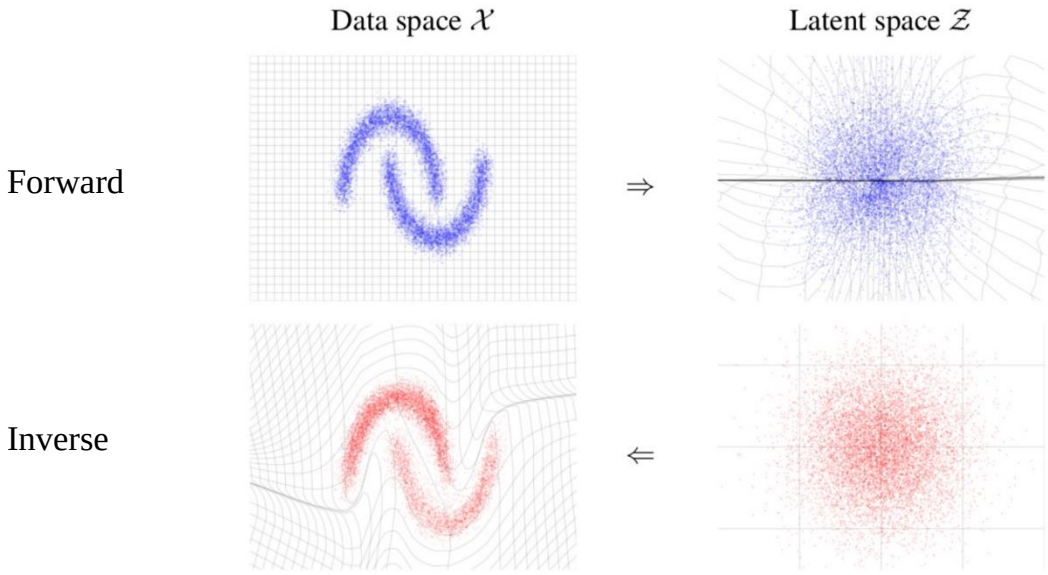
Since our invertible neural network (INN) learns both forward and inverse process, it has three loss functions: forward loss function, x-probability loss function and z-probability loss function. First one is responsible for the forward process to be learned correctly, x-loss learns probability of unknowns and z-loss is responsible for keeping probability of the latent variable as defined and be independent from y. The idea is that INNs are capable of associating hidden parameter values x with unique pairs y, z.

The paper presents four successful use-cases of this network: two artificial problems, one medical and one astrophysical application. We implemented INNs for two toy artificial problems, and obtained results that demonstrate the ability of this model to reveal the complete posterior of unknowns.

Problem formulation

Given a set of observations $y \in \mathbb{R}^M$ and some well-defined forward process $s(x)$, we want to recover a set of variables $x \in \mathbb{R}^D$. The transformation from x to y suffers from an information loss, and so the intrinsic dimension m of y is smaller than the dimension of x. Therefore, it makes sense to express the inverse model as a conditional probability $p(x|y)$.

In order to approximate this full posterior probability, researchers introduce a latent random variable $z \in \mathbb{R}^k$, drawn from multi-variate standard normal distribution $z \sim p(z) = N(z; 0, I_k)$, that does not represent any real-world phenomena, but serves as a model's tool.



The conditional probability $p(x|y)$ is often intractable in real problems, therefore we want to approximate it with a tractable model $q(x|y)$, thus reparametrize $q(x|y)$ in terms of a deterministic function $g(y, z; \theta)$ such that this function g is represented by a neural network with parameters θ:

$$x = g(y, z; \theta).$$

In this paper, the goal is to learn the inverse process jointly with a model $f(x; \theta)$ that approximates the known forward process $s(x)$:

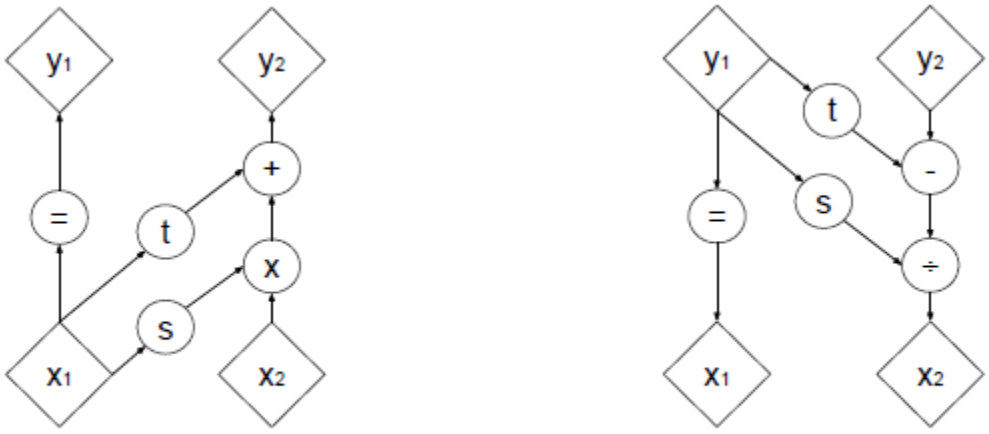
$$\begin{cases} [y, z] = f(x; \theta) = \left[f_y(x; \theta), f_z(x; \theta)\right] = g^{-1}(x; \theta), \\ f_y(x; \theta) \approx s(x) \end{cases}$$

Following these definitions, our network expresses $q(x|y)$ as:

$$q(x = g(y, z; \theta)|y) = p(z)|J_x|^{-1},\ J_x = \det\left(\frac{\partial g(y,z;\theta)}{\partial [y,z]}\Big|_{y,f_z(x)}\right).$$

Model architecture:

The network in this work, consists of "coupling blocks" (which is described in a paper by Dinh et al). The coupling block structure is illustrated in the following computational graphs for forward and inverse propagation:



(a) Forward propagation (b) Inverse propagation

The idea is to split our input vector $x \in \mathbb{R}^D, d < D$ into two halves $x_{1:d}, x_{d+1:D}$ and apply forward propagation in this way:

$$y_{1:d} = x_{1:d}$$

$$y_{d+1:D} = x_{d+1:D} \odot \exp(s(x_{1:d})) + t(x_{1:d})$$

The first half (the first d entries in the input) is passed without any modification. The second part of the output, $y_{d+1:D}$, is the result of the second part of the input vector ($x_{d+1:D}$) multiplied (elementwise) by exp(s($x_{1:d}$)) and then there is an addition of a vector $t(x_{1:d})$. (see illustration above).

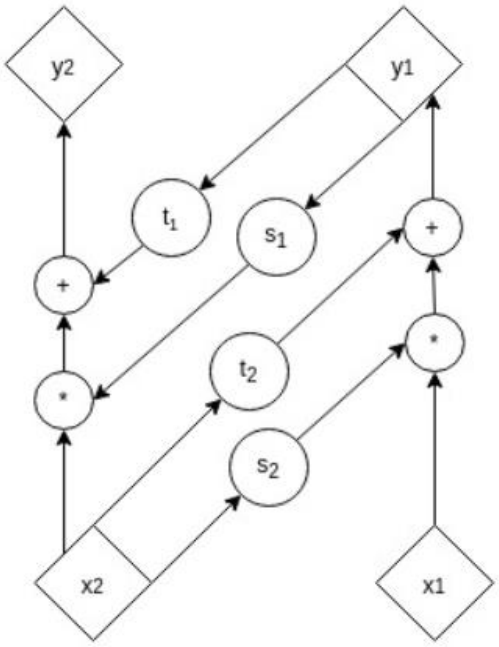
Regarding the inverse propagation:

$$x_{1:d} = y_{1:d}$$

$$x_{d+1:D} = (y_{d+1:D} - t(y_{1:d})) \odot \exp(-s(y_{1:d}))$$

Now we're splitting y into $y_{1:d}$ and $y_{d+1:D}$ and we look for the x's. We know that $y_{1:d}$ is $x_{1:d}$, and so we pass it through. To calculate $x_{d+1:D}$ we subtract $t(y_{1:d})$ from $y_{d+1:D}$ and divide this value elementwise by exp(s($y_{1:d}$)).

It is important to mention that computing the inverse of coupling layer, does not require computing inverse of the functions s and t, meaning that these functions can be arbitrarily complex. In this paper's implementation, these s and t transformations are built of a succession of several dense layers with leaky ReLU activations.



Then, outputs of the coupling blocks will be concatenated and shuffled, in what is called "permutation layer". This layer is inserted between every two coupling blocks.

Note that it is mandatory that input and output dimensions agree. So, sometimes it is necessary or useful to pad either or both input and output vectors with a small noise. Each epoch of training, other values are given to these padding vectors to ensure that the model is not learning on them.

In fact, the authors presented in the paper a different architecture that is described in the figure on the left. But their final model is based on the simpler architecture above.

Training:

Training is done by performing forward and inverse propagation in an alternating manner such that gradients are computed twice – after computing network loss on a batch of examples through the forward pass, and after computing network loss through the inverse pass. This procedure is executed per each batch of training examples and is repeated a certain number of epochs (batch size and number of epochs are hyper-parameters that are tuned according to each application on which the INN is applied on).

Three loss functions are involved in the training process: L_x, L_y, L_z.
L_x, L_z are based on a method called MMD – Maximum Mean Discrepancy.

Maximum Mean Discrepancy (MMD) is a kernel-based estimate of the difference between two distributions, based on samples from each of them. So, it is used to compare two probability distributions when we do not know analytical formula for them.

In our context we use the following formulas for MMD loss and kernel function:

$$MMD(X,Y) = \frac{1}{n}[\sum_{i,j=1}^{n} k(x_i, x_j) - 2\sum_{i,j=1}^{n} k(x_i, x_j) + \sum_{i,j=1}^{n} k(y_i, y_j)\ ]^{\frac{1}{2}}$$

$$k(x, x') = 1/(1 + ||(x - x')/h||_2^2)$$

where h is hyper-parameter.

Let us describe the losses in detail. In the forward pass, the optimization is done with respect to two losses:

- The loss L_y is a supervised loss, for example MSE loss that is used to compare the model's output with the respective label. This loss forces the output to correspond to the ground truth observations.
- L_z is implemented by MMD. It compares joint distribution $q(y, z)$ and $p(y)p(z)$. It ensures that y and z are independent and it keeps z to be distributed as defined.

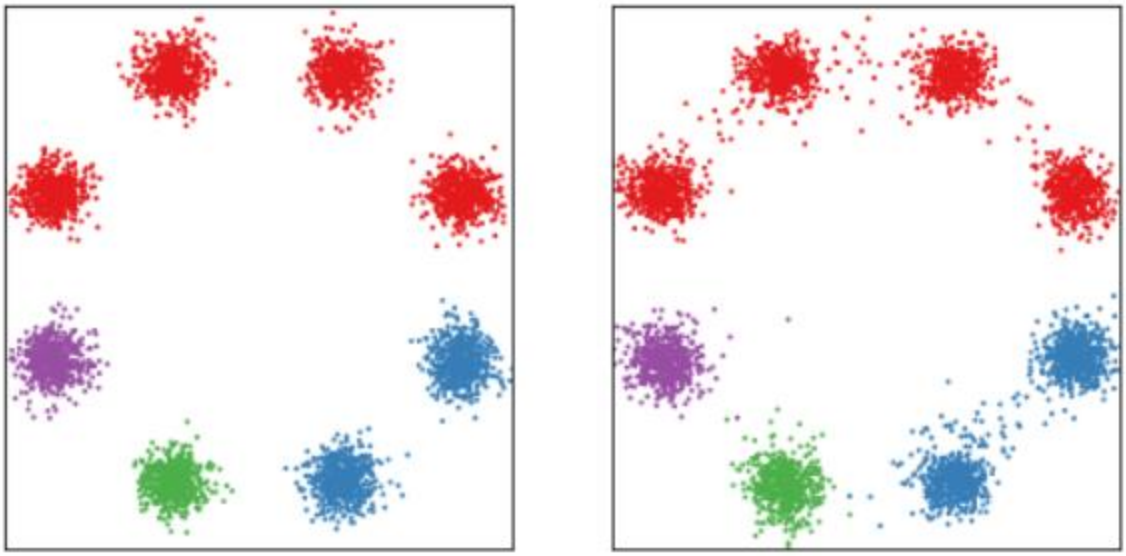
In addition to the last two mentioned losses, we use L_x loss that is calculated during the inverse pass – this loss is also implemented by MMD. Its goal is to force the inverse pass of the model to output x's from the same distribution as the prior $p(x)$. This third loss L_x becomes zero when the two forward losses L_y, L_z have converged to zero. So, we use L_x loss to improve convergence rate. Since the magnitude of the losses is different, we use different weights for them.

Experiments:

In this section we are going to present experimental results that demonstrate the power of the method described in the paper. Specifically, we will show the results obtained after applying the method on two synthetic problems.

Gaussian Mixture Model:

In the forward process, we distribute 2D points in 8 Gaussian distributions with relatively far means. Each of these distributions corresponds to a color. Hence, the forward pass of the INN predicts a color y of a given 2D point x (left). In the inverse pass, the INN finds the posterior probability of positions for a specific color (right). The model is trained with 10^6 generated samples.

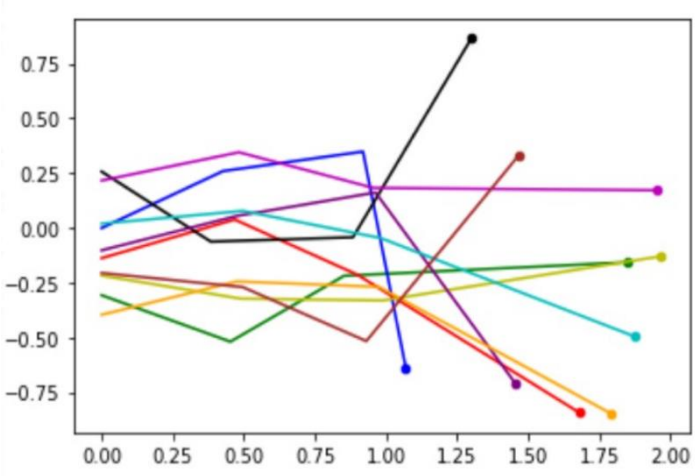


Note: the figure above illustrates the model performance on the test samples

Inverse Kinematics:

We have a robotic arm with three segments that moves in a plane, x_1 is a 'shoulder' that defines vertical starting position of the base. x_2, x_3, x_4 represent rotation of three joints. For each configuration, this arm reaches some position (y_1, y_2) on a plane.

In an inverse process we are given a position y (2D point), and we want to be able to recover the parameters that define a configuration of a simulated robot's arm such that the arm will land at the specified position. In other words, we want to know what values of x_1, x_2, x_3, x_4, will lead the arm to have a pose such that it ends up being located at the given target point.



Each arm pose is given by a connected series of 3 line segments:

P_1P_2, P_2P_3, P_3P_4 that are defined by these 4 points

$P_1 = (0, x_1)$

$$P_2 = (l_1 cos(x_2), x_1 + l_1 sin(x_2))$$

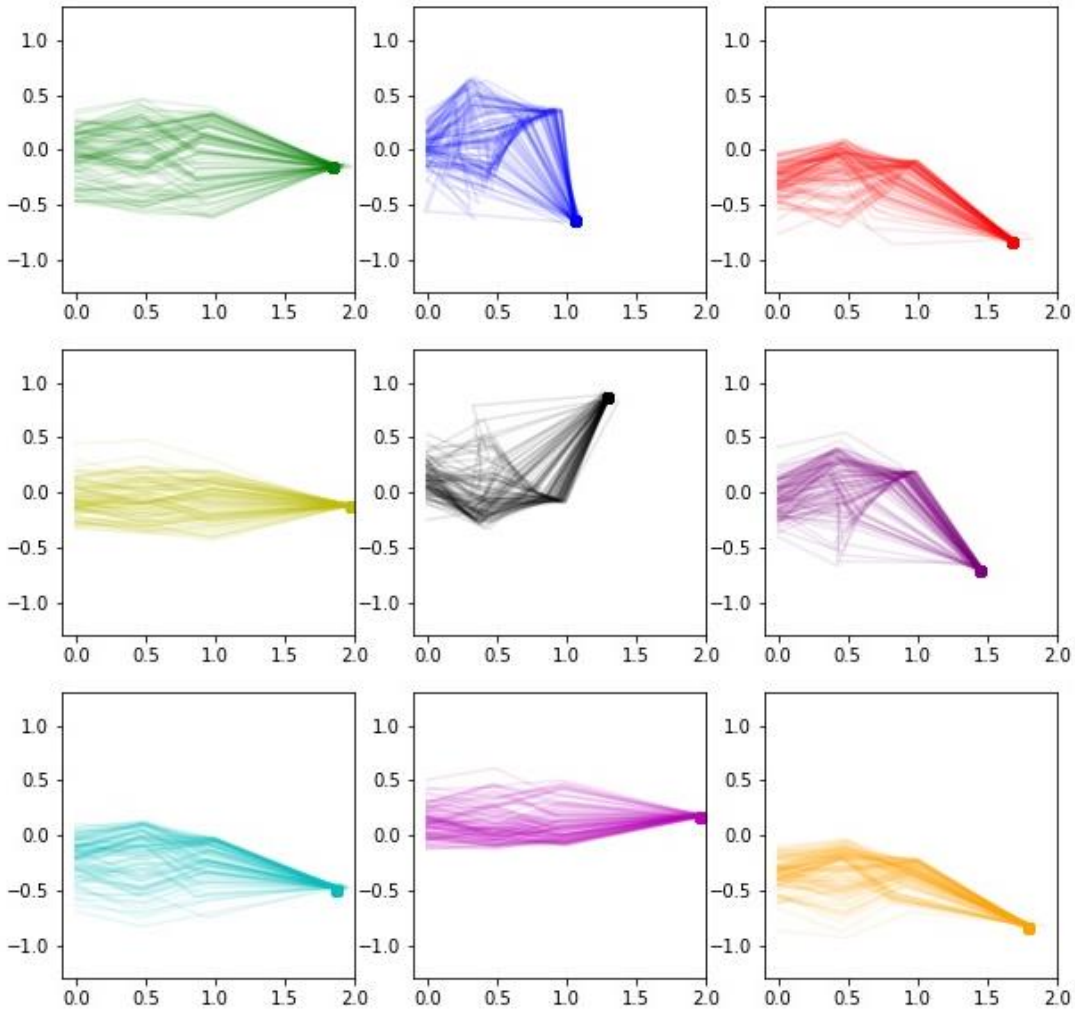
$$P_3 = (l_1 cos(x_2) + l_2 cos(x_3 - x_2), x_1 + l_1 sin(x_2) + l_2 sin(x_3 - x_2))$$

$$P_4 = (l_1 cos(x_2) + l_2 cos(x_3 - x_2) + l_3 cos(x_4 - x_3 - x_2), x_1 + l_1 sin(x_2) + l_2 sin(x_3 - x_2) + l_3 sin(x_4 - x_3 - x_2)) = (\mathrm{y}_1, \mathrm{y}_2)$$

such that x_1 is generated from Gaussian distribution with 0 mean and 0.25 variance. Similarly, x_2, x_3, x_4 are generated with 0 mean and 0.5 variance.

The forward pass finds the target point y, given a configuration x, whereas in the inverse pass we are interested in the distribution of all the parameters x for which the arm is posed such that it lands at the given point y.

Here are some examples of a target point and possible configurations of the arm pose such that it reaches the target input position.



We trained the INN using 10^6 generated samples. In the image above we see 9 different y positions that model was not trained on, each line in these figures represents a configuration of the parameters, that is sampled from the distribution that the INN learns – meaning the distribution of the parameters x given the target position y.

Also, for this problem we performed an experiment using ABC method. For this method, the L_2 distance error is as small as we want but there is a tradeoff between the computation time and accuracy. For instance, we chose error to be 0.005 and it took us 7 minutes to generate some possible x configurations (from 100 to 1300) for the same 9 points above, whereas INN can give unlimited number of configurations for any y immediately.

Summary:

During this project we learned how the innovative field of Deep learning can be exploited to tackle inverse problems. Specifically, we learned a unique architecture of neural networks – Invertible Neural Networks (INNs). We experienced building this architecture and training models based on it.

We demonstrated the performance of this architecture in terms of approximating the full posterior of two inverse problem – the Gaussian Mixture Model and the Inverse Kinematics.

It was interesting to experience how two (seemingly unrelated) fields – Deep Learning and Inverse Problems can be combined in solving both synthetic and real-world applications.

This project enabled us to be exposed to inverse problems from the practical and innovative field of Deep Learning, and in this sense, it provides us an additional practical perspective to Inverse Problems, in addition to what we learned throughout the course.