Probabilistic programming I
Introduction to Pyro

Thomas Hamelryck
thamelry@bio.ku.dk
PML 2022

Deep probabilistic programming

1. Modelling **scope** of deep learning
2. Principled treatment of **uncertainty** of Bayesian statistics
3. **Probabilistic models = Computer programs**

ChatGPT

What is deep probabilistic programming?

Deep probabilistic programming is a subfield of artificial intelligence and probabilistic programming that focuses on the development of probabilistic models that use deep learning techniques to perform complex probabilistic inference. Probabilistic programming is a framework that allows for the specification and solving of probabilistic models, which are used to represent uncertain or probabilistic knowledge. Deep learning, on the other hand, is a type of machine learning that involves the use of neural networks with many layers to learn complex patterns in data. By combining these two fields, deep probabilistic programming seeks to create powerful models that can perform complex probabilistic inference using deep learning techniques. This approach has many potential applications, including in natural language processing, computer vision, and robotics.

Try again

Reading material

- Pyro article, J. Machine Learning Research, 2019
- Pyro tutorial
  - An introduction to models in Pyro
  - Tensor shapes in Pyro

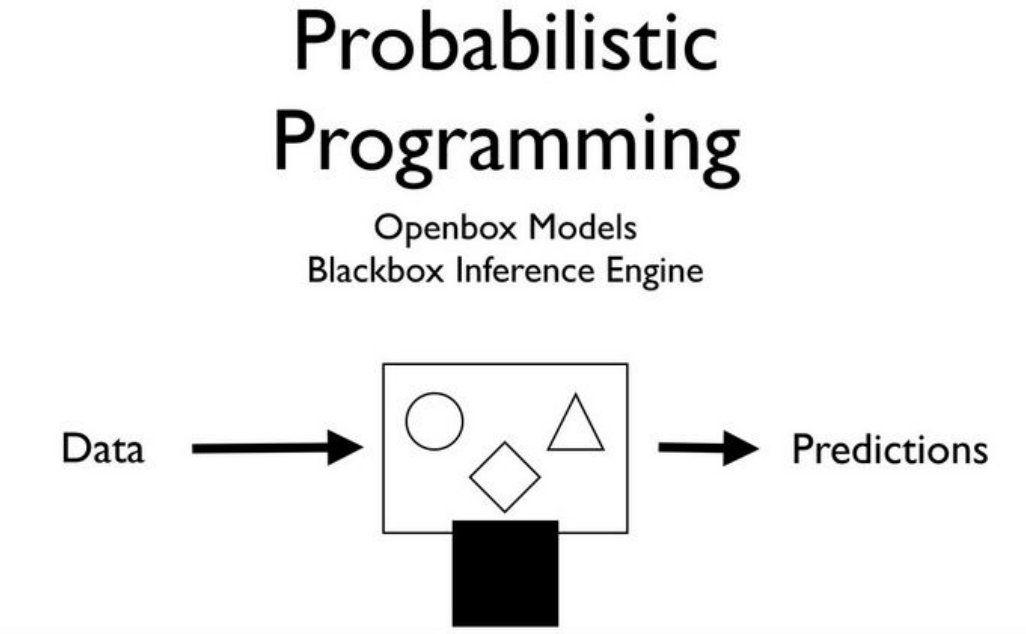
The Bayesian calculus

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

$$p(\theta|D) = \frac{P(D|\theta)\pi(\theta)}{P(D)} \propto P(D|\theta)\pi(\theta)$$

Thomas Bayes
(1701-1761)

Bayes in the 21st century



Picture: Olivier Grisel's talk on ML

Bayesian linear model

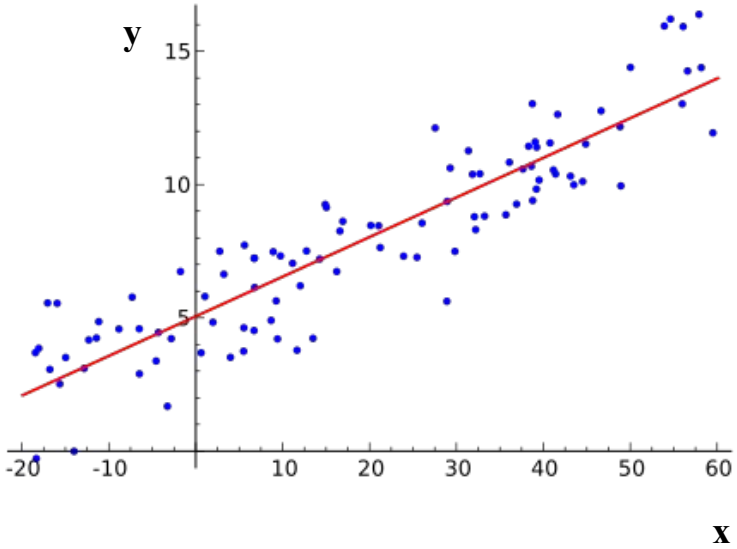
$$a \sim N(a \mid 0, 1)$$

$$b \sim N(b \mid 0, 1)$$

$$\sigma \sim N_+(0, 1)$$

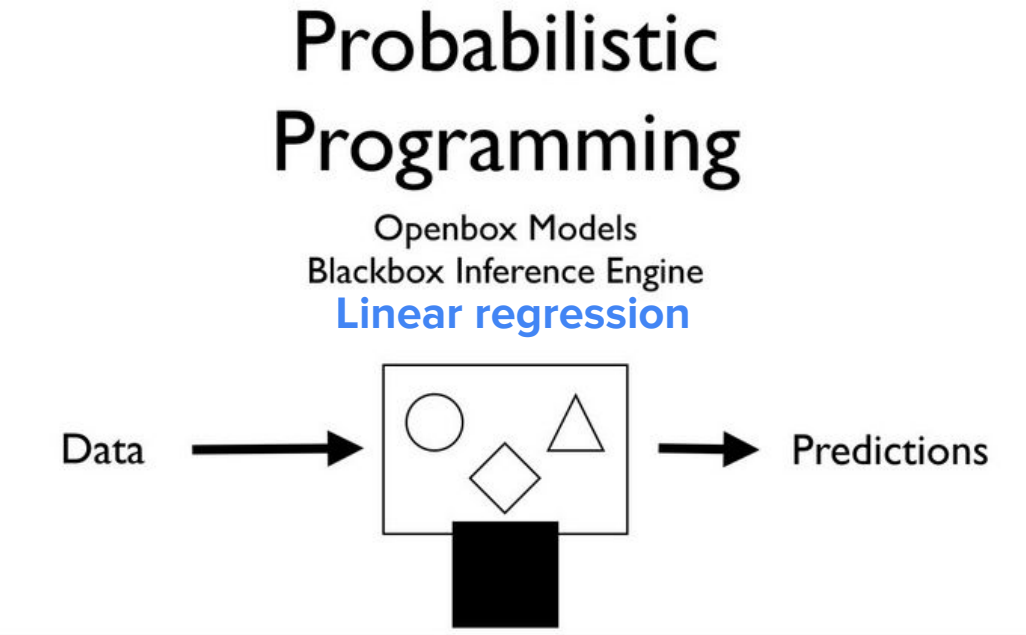
$$y \sim N(y \mid \mu, \sigma)$$

$$\mu = a + bx$$



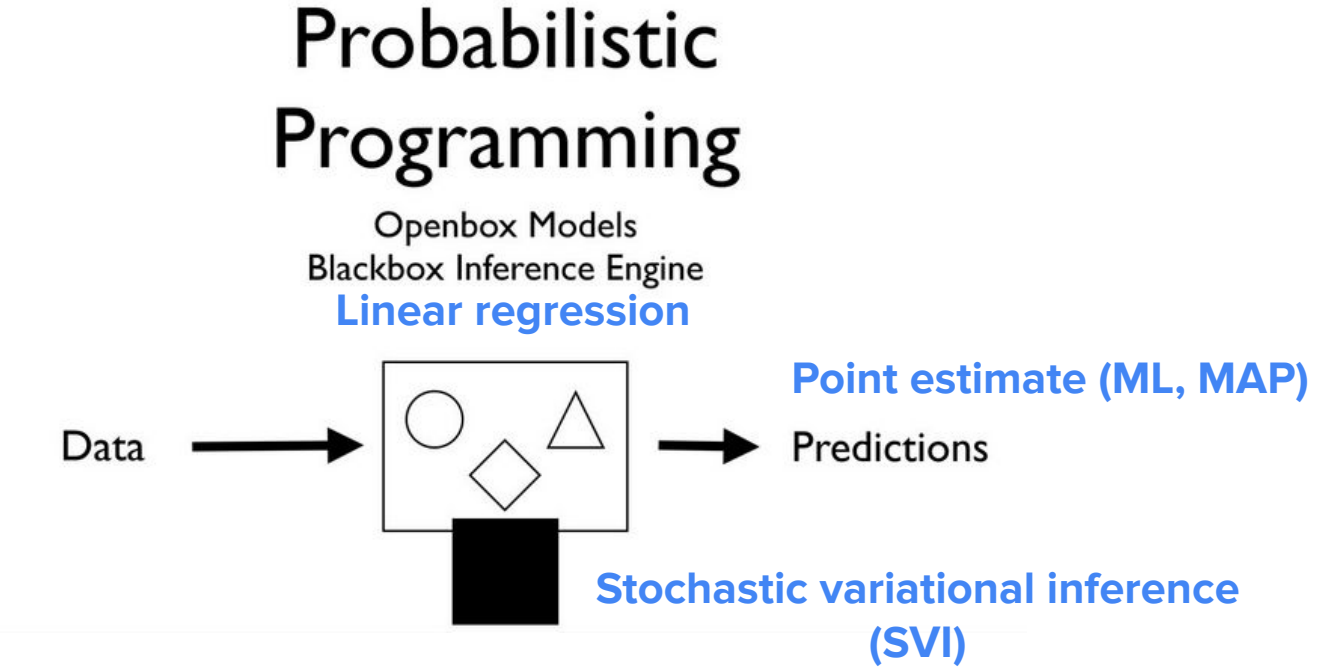
Picture (right): Wikipedia

Bayes in the 21st century



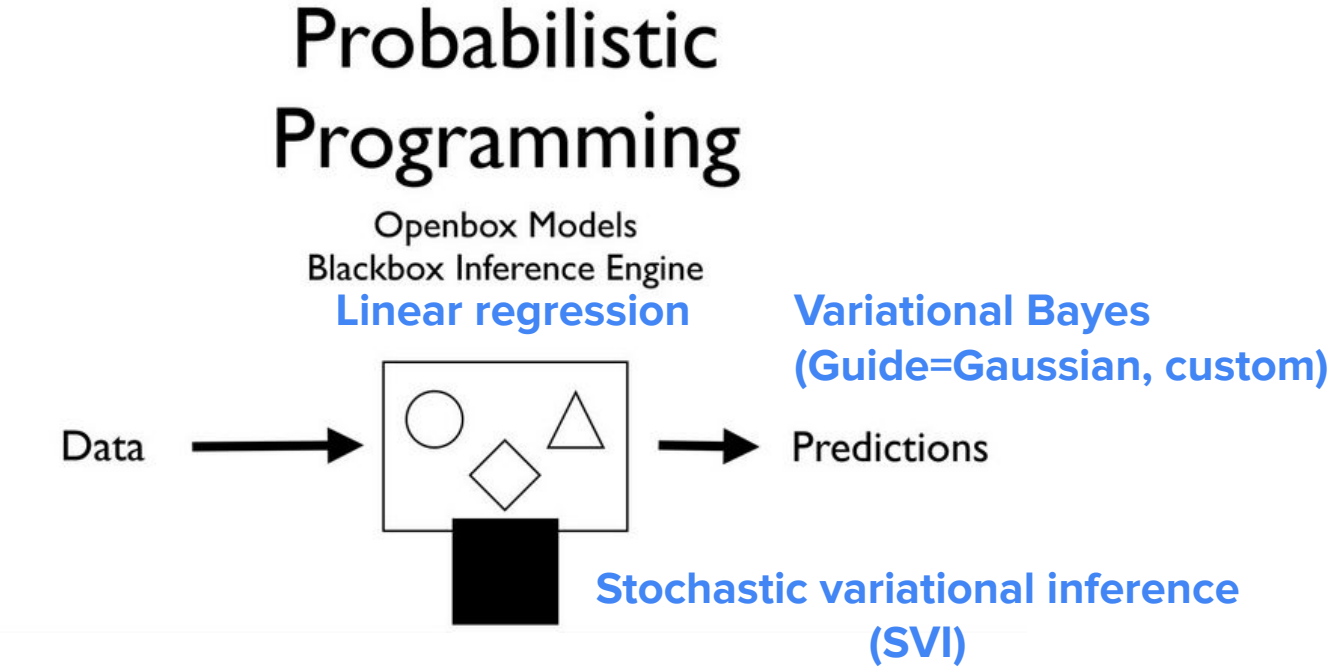
Picture: Olivier Grisel's talk on ML

Bayes in the 21st century



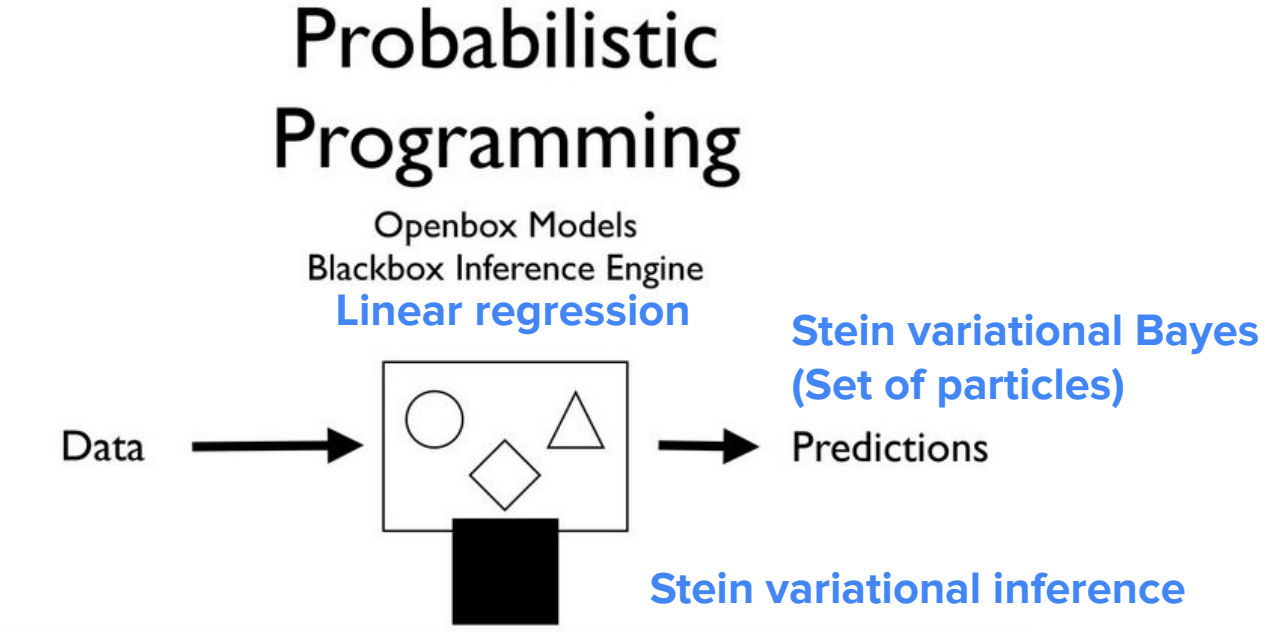
Picture: Olivier Grisel's talk on ML

Bayes in the 21st century



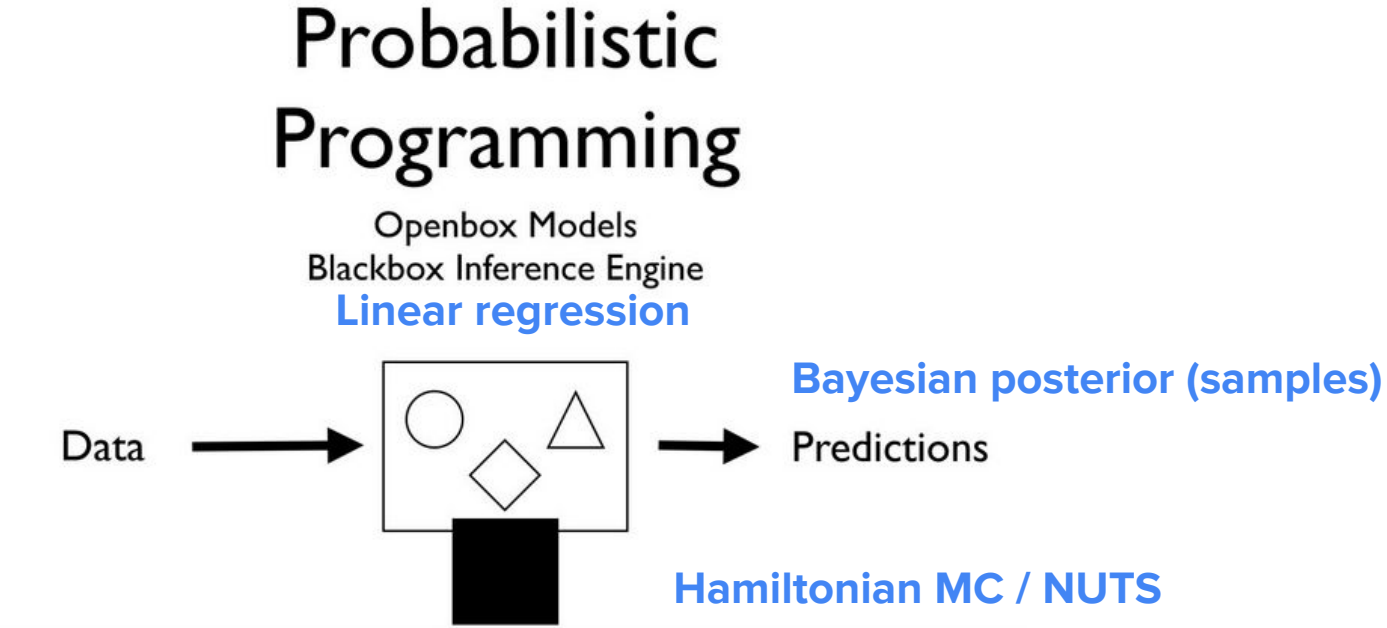
Picture: Olivier Grisel's talk on ML

Bayes in the 21st century



Picture: Olivier Grisel's talk on ML

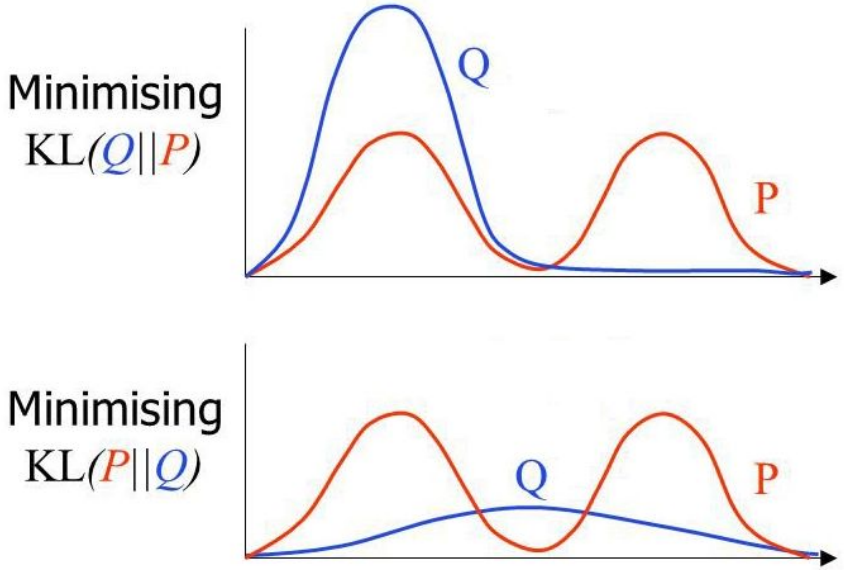
Bayes in the 21st century



Picture: Olivier Grisel's talk on ML

Automatic inference

$$\frac{\partial \theta}{\partial t} = \frac{\partial E_{kin}}{\partial p} = \frac{p}{m}$$
$$\frac{\partial p}{\partial t} = -\frac{\partial E_{pot}}{\partial \theta}$$



Sampling

Hamiltonian Monte Carlo / NUTS (2011)

Optimisation

Stochastic Variational Inference (SVI)

Pictures: Mathieu Lê / John Winn, Microsoft Research Cambridge (adapted)

SVI: KL divergence

- **Kullback-Leibler divergence**
  - Measures “distance” between two probability distributions
  - Not a distance / metric because $D_{KL}(p,q) \neq D_{KL}(q,p)$

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

SVI: Variational autoencoders

- Deep generative models

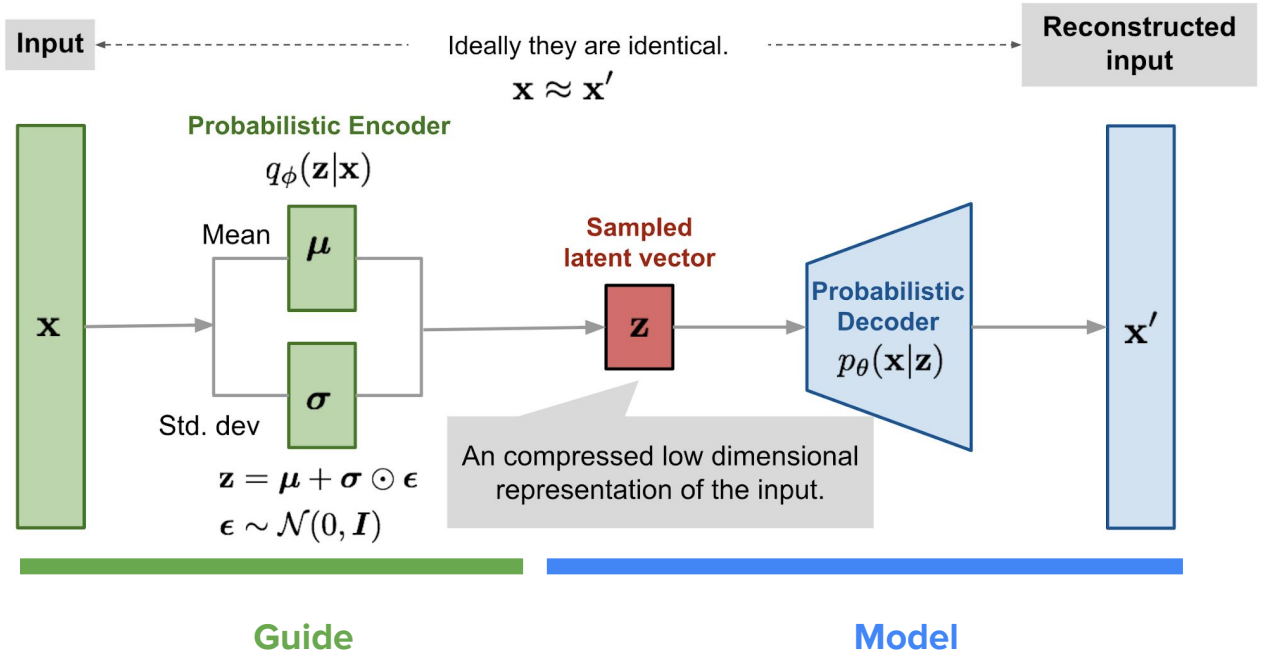


Image: adapted from https://lilianweng.github.io/

Bayesian linear model

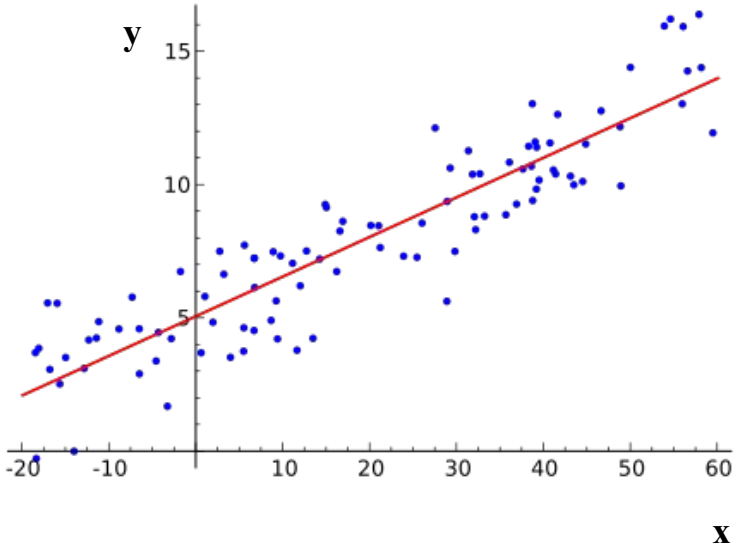
$$a \sim N(a \mid 0, 1)$$

$$b \sim N(b \mid 0, 1)$$

$$\sigma \sim N_+(0, 1)$$

$$y \sim N(y \mid \mu, \sigma)$$

$$\mu = a + bx$$



Picture (right): Wikipedia

Bayesian linear model in PyMC3

$$a \sim N(a \mid 0, 1)$$

$$b \sim N(b \mid 0, 1)$$

$$\sigma \sim N_+(0, 1)$$

$$y \sim N(y \mid \mu, \sigma)$$

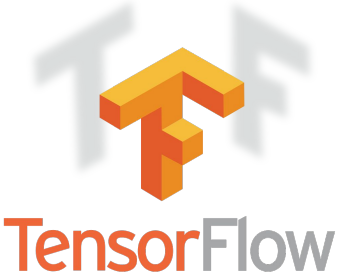
$$\mu = a + bx$$

```
from pymc3 import *

with Model() as model:
    # Define priors
    a = Normal('a', 0, sd=1)
    b = Normal('b', 0, sd=1)
    sigma = HalfNormal('sigma', 1)

    # Define likelihood
    likelihood = Normal('y', mu=a+b*x,
        sd=sigma, observed=y)

    # Inference!
    start = find_MAP() # Find starting value
    step = NUTS() # Sampling
    trace = sample(2000, step, start=start)
```

Stan PPL for Bayesian statistics

- From https://mc-stan.org/:
  - Full Bayesian statistical inference with MCMC sampling (NUTS, HMC)
  - Approximate Bayesian inference with variational inference (SVI)
  - Penalized maximum likelihood estimation with optimization (L-BFGS)

Deep probabilistic programming



TensorFlow Probability

Pyro PPL

Why Pyro?

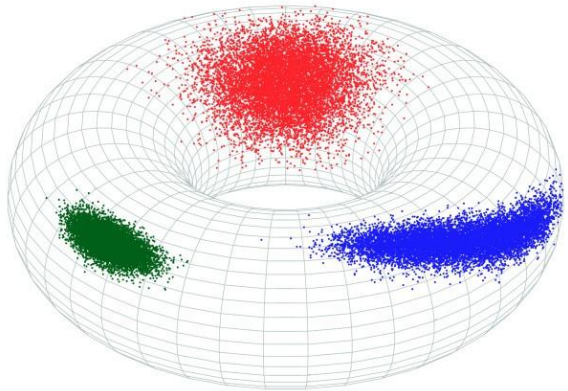
- Website: pyro.ai
- Universal probabilistic programming language (**PPL**)
  - Python based (PyTorch)
  - SVI, NUTS
  - Adds probabilistic layer on deep learning
- Originally developed at Uber AI Labs
- Now at The Broad Institute (MIT, Harvard)
- Freely available, easy installation
  - `pip3 install pyro-pp`

Numpyro

- Webpage: https://github.com/pyro-ppl/numpyro
- Pyro built on **JAX** instead of PyTorch
  - Numpy interface
  - SVI, NUTS, Stein variational inference
- JAX by Google
  - JIT compilation
  - Autograd from Harvard Intelligent Probabilistic Systems Group (HIPS)
  - XLA (Accelerated Linear Algebra) from Tensorflow
- 100x speedup for NUTS (**Iterated NUTS)**

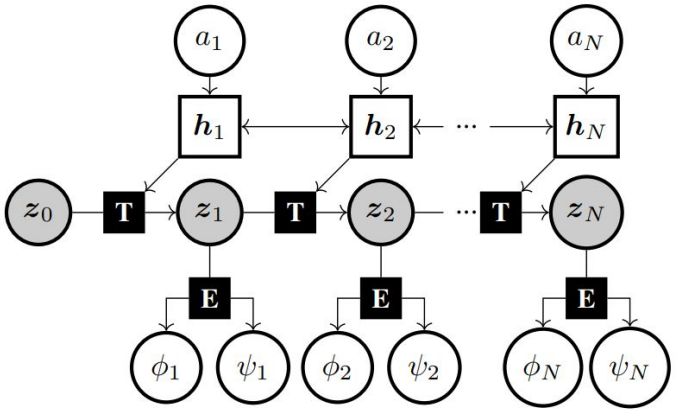
Probabilistic programming group @ DIKU

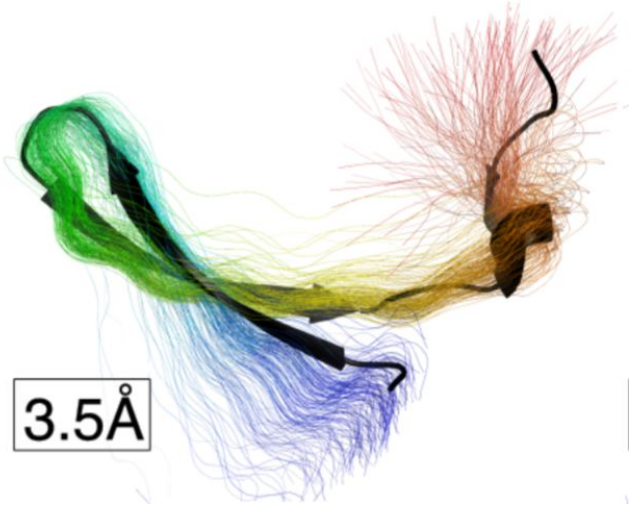
- Probability distributions on non-Euclidean manifolds
  - Bivariate von Mises distribution on the torus
- Mini-batch training for Hamiltonian MCMC
  - Hamiltonian Monte Carlo with energy conserving subsampling
- Stein variational inference
  - Bayesian neural networks
  - EinSteinVI: Stein mixtures



Probabilistic programming group @ DIKU

- Probabilistic model of protein structure (ICML, 2021)
  - Deep Markov model + pdf on the torus
- Vaccine design (Evaxion)





Pyro: Universal deep PPL

- **Universal**
  - Can represent any computable probability distribution.
- **Scalable**
  - Scales to large data sets with little overhead.
- **Minimal**
  - Implemented with a small core of powerful, composable abstractions.
- **Flexible**
  - Aims for automation when you want it, control when you need it.

Running Pyro: Colab

- Website:
  - https://colab.research.google.com/
- **Google's Colaboratory**
  - Jupyter notebook
  - Cloud computing
  - Can be shared
  - Easy to install packages such as Pyro, PyTorch etc.
  - Access to GPUs
  - No configuration needed
  - Commercial grade access (limited to US/Canada/DE)

Running Pyro: Colab

- Install software on the fly
  - Note the **!** at the start

```
!pip3 install pyro-ppl
!pip3 install sklearn
!pip3 install arviz
!pip3 install numpy
!pip3 install matplotlib
!pip3 install scipy
```

Running Pyro: Anaconda

```
$ conda create --name my_pyro python  # conda environment
```

Then:

```
$ conda activate my_pyro

$ pip install pyro-ppl

$ python <do stuff>

$ conda deactivate
```

Importing Pyro

```
import pyro
assert pyro.__version__.startswith('1.8.2')
pyro.set_rng_seed(1)
```

Pyro models

Stochastic functions

- Basic unit of a probabilistic program
  - Python callable
  - Deterministic Python code
  - **Primitive stochastic functions** that call a random number generator
- Stochastic functions=**models** and these can be
  - Composed
  - Reused
  - Imported
  - Serialized

Primitive stochastic functions

- Or in other words, distributions
  - Pyro is built upon PyTorch's distribution library
  - You can also use custom transforms

```
loc = 0.   # mean zero
scale = 1. # unit stddev
# create a normal distribution object
normal = pyro.distributions.Normal(loc, scale)
# draw a sample from N(0,1)
x = normal.sample()
print("sample", x)
# score the sample from N(0,1)
print("log prob", normal.log_prob(x))
```

Models

- Primitive stochastic functions + deterministic computation
  - Conditionals, recursion, transformations of rvs,...

```
def weather():
    cloudy = pyro.distributions.Bernoulli(0.7).sample()
    cloudy = 'cloudy' if cloudy.item() == 1 else 'sunny'
    mean_temp = {'cloudy': 10.0, 'sunny': 11.0}[cloudy]
    scale_temp = {'cloudy': 1.0, 'sunny': 2.0}[cloudy]
    temp = pyro.distributions.Normal(mean_temp, scale_temp).sample()
    return cloudy, temp.item()
```

Named samples: `pyro.sample`

- Used to identify sample statements
- Change their behavior at runtime using effect handlers (Poutine library) and **context managers** (`pyro.sample`, `pyro.markov`).

```
def weather():
    cloudy = pyro.sample('cloudy',
        pyro.distributions.Bernoulli(0.7))
    cloudy = 'cloudy' if cloudy.item() == 1.0 else 'sunny'
    mean_temp = {'cloudy': 10.0, 'sunny': 11.0}[cloudy]
    scale_temp = {'cloudy': 1.0, 'sunny': 2.0}[cloudy]
    temp = pyro.sample('temp',
        pyro.distributions.Normal(mean_temp, scale_temp))
    return cloudy, temp.item()
```

Universality

- Higher-order stochastic functions
  - Models can be called by other models

```
def ice_cream_sales():
    cloudy, temp = weather()
    expected_sales = 200. if cloudy == 'sunny' and temp > 80.0 else 50.
    ice_cream = pyro.sample('ice_cream',
        pyro.distributions.Normal(expected_sales, 10.0))
    return ice_cream
```

Universality

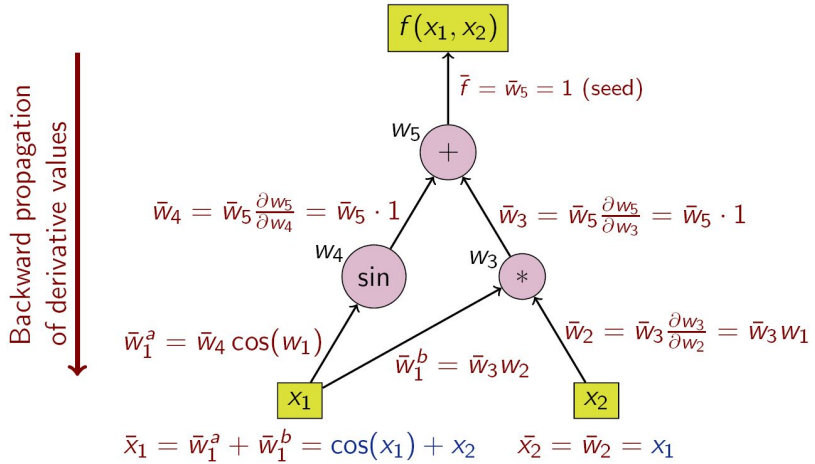
- Stochastic recursion, random control flow
  - The number of rvs per execution may vary

```
def geometric(p, t=None):
    # Number of failures until success (x==1)
    if t is None:
        t = 0
    x = pyro.sample("x_{}".format(t),
            pyro.distributions.Bernoulli(p))
    if x.item() == 1:
        return 0
    else:
        return 1 + geometric(p, t + 1)
```

Universality

- Gradients can be automatically obtained for general functions / algorithms
- Dynamic computational graph or **trace**
  - PyTorch's reverse mode automatic differentiation (autodiff)
  - Typically, dim(output)<<dim(input)

$$
\begin{aligned}
z &= f(x_1, x_2) \\
&= x_1 x_2 + \sin x_1 \\
&= w_1 w_2 + \sin w_1 \\
&= w_3 + w_4 \\
&= w_5
\end{aligned}
$$



Picture: Wikipedia

Pyro tensor shapes

Tensors

- A PyTorch tensor is a multi-dimensional matrix
  - With a specified **shape**
- ...containing elements of a single **data type**

```
>>> t=torch.zeros([2, 4], dtype=torch.int32)
>>> t
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=torch.int32)
>>> t.shape
torch.Size([2, 4])
```

Distribution shapes

- Tensors have a single shape attribute
- Distributions have two shape attributes
  - `batch_shape` and `event_shape`

```
x = d.sample()
assert x.shape == d.batch_shape + d.event_shape
```

Event shape

- Number of **dependent random variables**
- The dimensionality of the random variables
  - Univariate: empty shape
  - Vectors, like multivariate normal: int, length of the vector
  - Matrix, like inverse Wishart: int × int, size of matrix

```
x = d.sample()
assert x.shape == d.batch_shape + d.event_shape
```

Event shape

- Number of **dependent random variables**
- The dimensionality of the random variables
  - Univariate: empty shape
  - Vectors: `torch.Size([m])`
  - Matrix: `torch.Size([m,n])`

```
x = d.sample()
assert x.shape == d.batch_shape + d.event_shape
```

Batch shape

- Number of **independent random variables**
  - Example: `torch.Size([1])`
  - Example: `torch.Size([5,5,3])`

```
d=pyro.distributions.Bernoulli(
    torch.tensor([0.5, 0.5]))
x = d.sample()
assert x.shape == d.batch_shape + d.event_shape
assert d.batch_shape == (2,)
assert d.event_shape == ()
assert d.log_prob(x).shape == d.batch_shape
```

Sample shape

- Number of independent samples of the batched rvs
  - Argument to `d.sample()`

```
sample_shape = torch.Size([2])
x = d.sample(sample_shape)
assert x.shape == sample_shape + d.batch_shape + d.event_shape
```

Exercise

- What is the shape of x?
- Make sense of sample, batch and event shape.

```
parameters = torch.tensor([[0.0,0.0,0.0], [1.0,1.0,1.0]])
d = pyro.distributions.Normal(loc=parameters[0], scale=parameters[1])
x = d.sample()
```

Exercise solution

- What is the shape of x?
- Make sense of sample, batch and event shape.

```
parameters = torch.tensor([[0.0,0.0,0.0], [1.0,1.0,1.0]])
d = pyro.distributions.Normal(loc=parameters[0], scale=parameters[1])
x = d.sample()
assert x.shape == d.batch_shape  # sample and event shape are empty
assert x.shape == torch.Size([3]) # batch shape
```

Exercise

- What is the shape of x?
- Make sense of sample, batch and event shape.

```
s = torch.Size([4,4])
parameters = torch.tensor([[1.0,1.0,1.0], [2.0,2.0,2.0]])
d = pyro.distributions.Dirichlet(parameters)
x = d.sample(s)
```

Exercise solution

- What is the shape of x?
- Make sense of sample, batch and event size.

```
s = torch.Size([4,4])
parameters = torch.tensor([[1.0,1.0,1.0], [2.0,2.0,2.0]])
d = pyro.distributions.Dirichlet(parameters)
x = d.sample(s)
assert x.shape == sample_shape + d.batch_shape + d.event_shape
# sample=(4,4), batch=2, event=3
assert x.shape == torch.Size([4, 4, 2, 3])
```

Another way to batch: expand

- Sizes need to be identical along the rightmost dimensions.
- Rvs are conditionally independent.

```
d = pyro.distributions.Bernoulli(
    torch.tensor([0.1, 0.2, 0.3, 0.4])).expand([3, 4])
assert d.batch_shape == (3, 4)
assert d.event_shape == ()
x = d.sample()
assert x.shape == (3, 4)
assert d.log_prob(x).shape == (3, 4)
```

Another way to batch: expand

- Sizes need to be identical along the rightmost dimensions.
- Rvs are conditionally independent.

```
d = pyro.distributions.Bernoulli(
    torch.tensor([0.1, 0.2, 0.3, 0.4])).expand([3, 4])
assert d.batch_shape == (3, 4)
assert d.event_shape == ()
x = d.sample()
assert x.shape == (3, 4)
assert d.log_prob(x).shape == (3, 4)
```

Conditional dependencies in Pyro

- By default, all variables are **dependent** on all previous variables
  - See graphical models, **d-separation**
- We need to tell Pyro about independence!
  - Will become important for inference
- **Independence within** sample statement
  - sample shape, batch shape, `expand`
- **Dependence within** sample statement
  - event shape, `to_event` (turn batch dimensions into event)
- **Independence between** sample statements via context managers
  - `pyro.plate`
  - `pyro.markov`

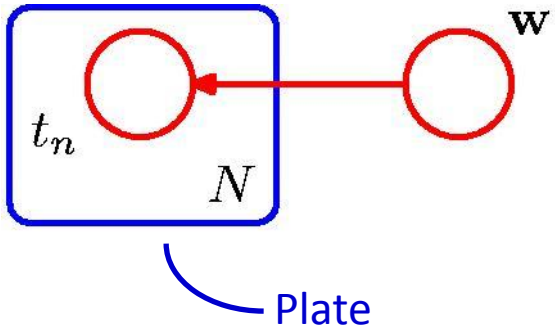
Making dependent: `to_event`

- You can treat a univariate distribution as multivariate using `to_event(n)`
  - n is the number of batch dimensions (from the right) to declare as **dependent**

```
d = pyro.distributions.Bernoulli(
    0.5 * torch.ones(3,4)).to_event(1)
assert d.batch_shape == (3,)
assert d.event_shape == (4,)
x = d.sample()
assert x.shape == (3, 4)
assert d.log_prob(x).shape == (3,)
```

Making independent: `pyro.plate`

- Plates are used in graphical models
  - See lecture on graphical models



Making independent: pyro.plate

- Plate informs Pyro that it can assume independence when estimating gradients and doing inference
  - Typical used on likelihoods (iid data)
  - See d-separation in graphical models

```
with pyro.plate("x_plate", 10):
    # Note that .expand([10]) is automatic
    x = pyro.sample("x", pyro.distributions.Normal(0, 1))
assert x.shape == (10,)
```

Making independent: pyro.plate

- You can also explicitly iterate,
  - Similar to a Python for loop

```
x = torch.zeros(10)
for i in pyro.plate("x_plate", 10):
    x[i] = pyro.sample("x", pyro.distributions.Normal(0, 1))
assert x.shape == (10,)
```

Making independent: Nested plates

- Example: Making pixels in a 2D image independent

```
with pyro.plate("x_axis", 320):
    # within this context,
    # batch dimension -1 is independent
    with pyro.plate("y_axis", 200):
        # within this context,
        # batch dimensions -2 and -1 are independent
```

Making independent: pyro.markov

- Example: Markov chain of Gaussians
  - Note use of pyro.plate and pyro.markov

```
L = 10   # Sequence length
N = 5    # Number of sequences
x = torch.zeros(N,L)
for i in pyro.plate("sequences", N):
 for t in pyro.markov(range(L)):
   if t>0:
     x[i,t] = pyro.sample("x_{}_{}".format(i,t),
       pyro.distributions.Normal(loc=x[i,t-1], scale=1))
```

Example: Gaussian mixture model

Exercise: Sample from a GMM

- Consider a GMM with two components.
- Parameters
  - Probabilities of the two mixture components
  - Means and standard deviations of the two Gaussians

$$p(x) = \sum_{i=1}^{2} p(x \mid \mu_i, \sigma_i) p(i)$$

Exercise: Sample from a GMM

- Get one sample from the GMM with the given parameters.
  - Hint: use Categorical distribution for the mixture component

```
# Mixture component probabilities
p=torch.tensor([0.25, 0.75])
# Means
m=torch.tensor([0.0, 10.0])
# Standard deviations
s=torch.tensor([1.0, 2.0])
```

Exercise: solution

```
# Mixture component probabilities
p=torch.tensor([0.25, 0.75])
# Means
m=torch.tensor([0.0, 10.0])
# Standard deviations
s=torch.tensor([1.0, 2.0])
# Sample
c=pyro.sample("c", pyro.distributions.Categorical(p))
x=pyro.sample("x", pyro.distributions.Normal(m[c], s[c]))
```

Exercise: 100 samples from the GMM

- The previous code only gave us one sample.
- Change the code so it returns 100 samples
  - Use Pyro primitives!

Exercise solution I

```
# Mixture component probabilities
p=torch.tensor([0.25, 0.75])
# Means
m=torch.tensor([0.0, 10.0])
# Standard deviations
s=torch.tensor([1.0, 2.0])
# Number of samples
n=100
with pyro.plate("gmm", n):
    c=pyro.sample("c", pyro.distributions.Categorical(p))
    x=pyro.sample("x", pyro.distributions.Normal(m[c], s[c]))
```

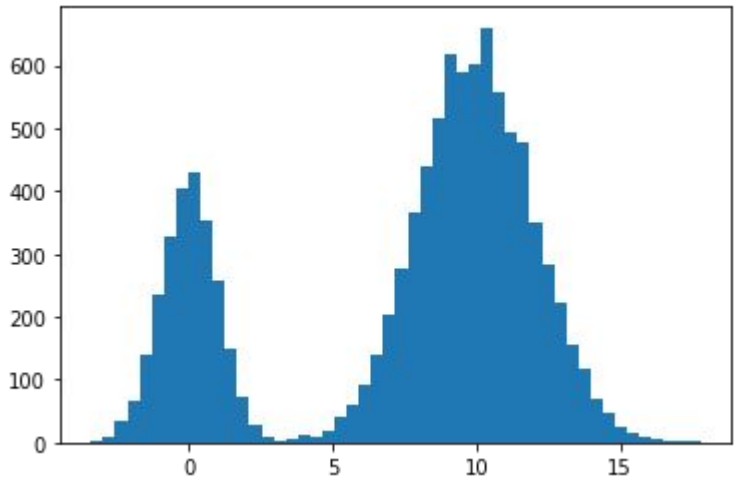
Exercise solution II

```
# Mixture component probabilities
p=torch.tensor([0.25, 0.75])
# Means
m=torch.tensor([0.0, 10.0])
# Standard deviations
s=torch.tensor([1.0, 2.0])
# Number of samples
n=100
c=pyro.sample("c", pyro.distributions.Categorical(p).expand([n]))
x=pyro.sample("x", pyro.distributions.Normal(m[c], s[c]))
```

Exercise solution III

- Histogram for 10.000 samples

```
plt.hist(x, bins=50)
plt.show()
```



Pyro II

- After New Year: Inference with Pyro
  - Stochastic variational inference
  - Hamiltonian Monte Carlo / NUTS
  - [Stein variational inference with NumPyro]