Simulation-based inference

Simulation-based inference

If our data generating process is complex, we might not be able to write down a likelihood density function that computes the probability of the data given the parameters.

We usually still can sample data realisations from this likelihood by building a forward model or simulator that reproduces the data-generating process.

- For a given x and θ, we cannot compute the density $p(x|\theta)$
- But we can sample $x_i|\theta \sim p(\cdot|\theta)$

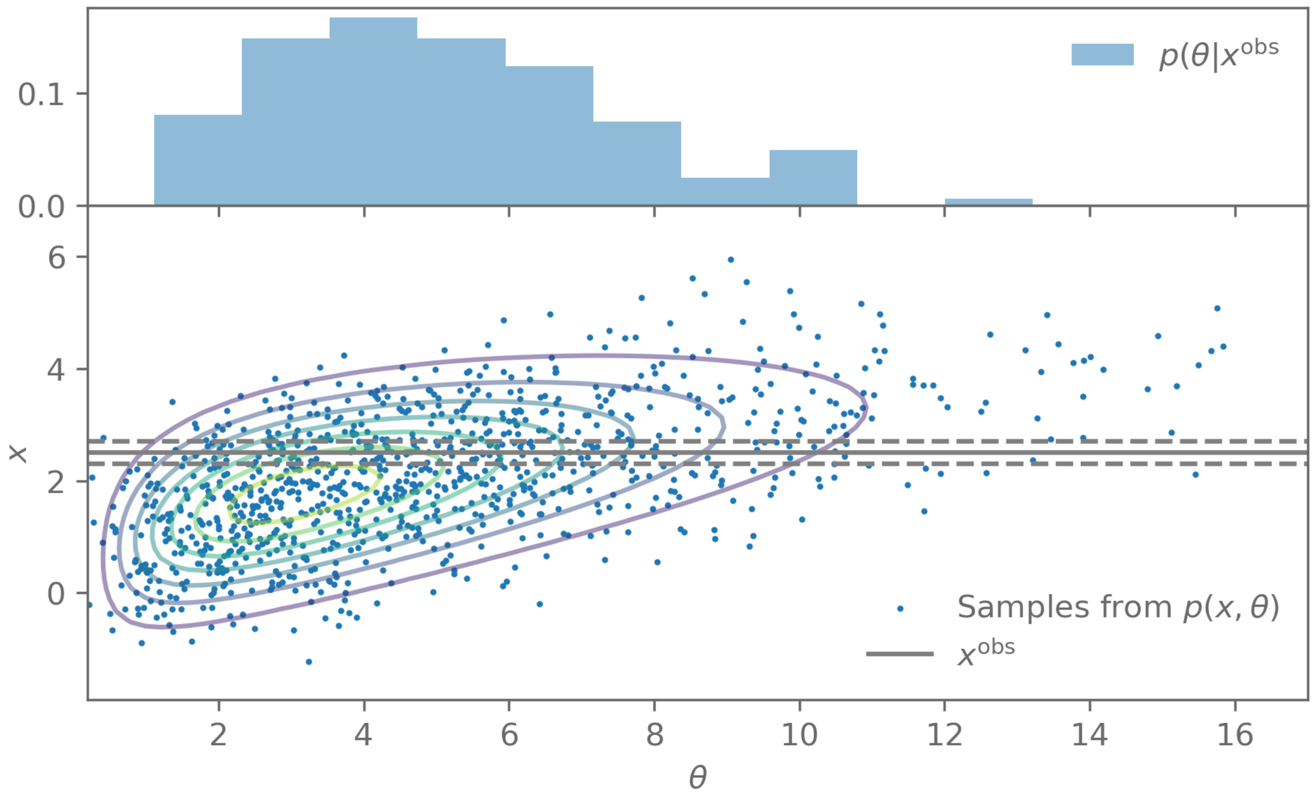
In simulation-based inference we make posterior inference using only the data-generating process, without evaluating likelihood functions.

Approximate Bayesian computation (ABC)

The simplest implementation is rejection ABC.

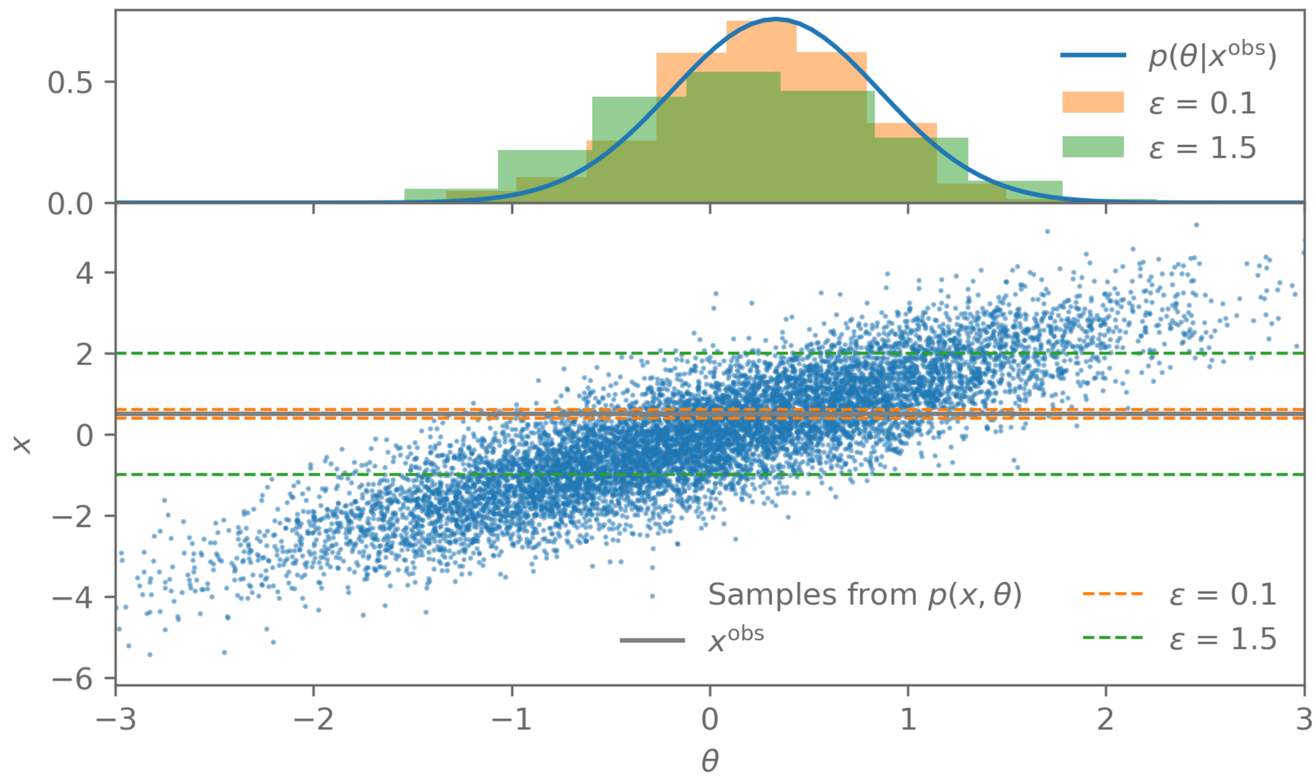
1. Sample θ_i from the prior
2. Generate y_i^{rep} from the forward model, based on θ_i
3. Accept θ_i if some distance metric $d(y, y_i^{\mathrm{rep}})$ between y and y_i^{rep} is smaller than a threshold ϵ

For an in-depth review of ABC, see for example Sisson et al. 2018.



- If ϵ is chosen too small, the acceptance rate will be low
- If ϵ is chosen too large, the approximation to the posterior will be poor
- If the dimensionality of the data is large, we are hit with the curse of dimensionality and the acceptance rate will be very low.
    - To combat this, usually summary statistics are used. If these summary statistics are not sufficient statistics (capturing all the information of the data), then some information will be lost

```
Number of accepted samples for ϵ = 0.1: 509/10000
Number of accepted samples for ϵ = 1.5: 6484/10000
```



Neural density estimation

Often the simulators are expensive to run and producing samples from the likelihood or posterior is expensive.

The idea of (conditional) density estimation is to estimate a probability density function that approximates the true distribution that the samples came from.

We want to fit an approximate distribution $q_\phi(x)$, which is parametrised by ϕ.

The optimisation objective is often the Kullback-Leibler (KL) divergence

$$D_{\mathrm{KL}}(p\|q) = \int p(x)\log\frac{p(x)}{q(x)}\mathrm{d}x$$

between the true distribution p and our approximation q.

The KL divergence measures how well the two distributions agree:

- $D_{\mathrm{KL}}(p\|q) = 0$ when the distributions agree
- $D_{\mathrm{KL}}(p\|q) \geq 0$ in general
- Not symmetric
- Mutual information $I(X,Y) = D_{\mathrm{KL}}(p(X,Y)\|p(X)p(Y))$

We want to fit an approximate distribution $q_\phi(x)$, which is parametrised by ϕ.

$$
\begin{aligned}
D_{\mathrm{KL}}(p \| q_\phi) &= \mathrm{E}_p\left[\log \frac{p(x)}{q_\phi(x)}\right] \\
&= \mathrm{E}_p\left[-\log q_\phi(x)\right] + \underbrace{\mathrm{E}_p[\log p(x)]}_{\text{does not depend on } q_\phi}
\end{aligned}
$$

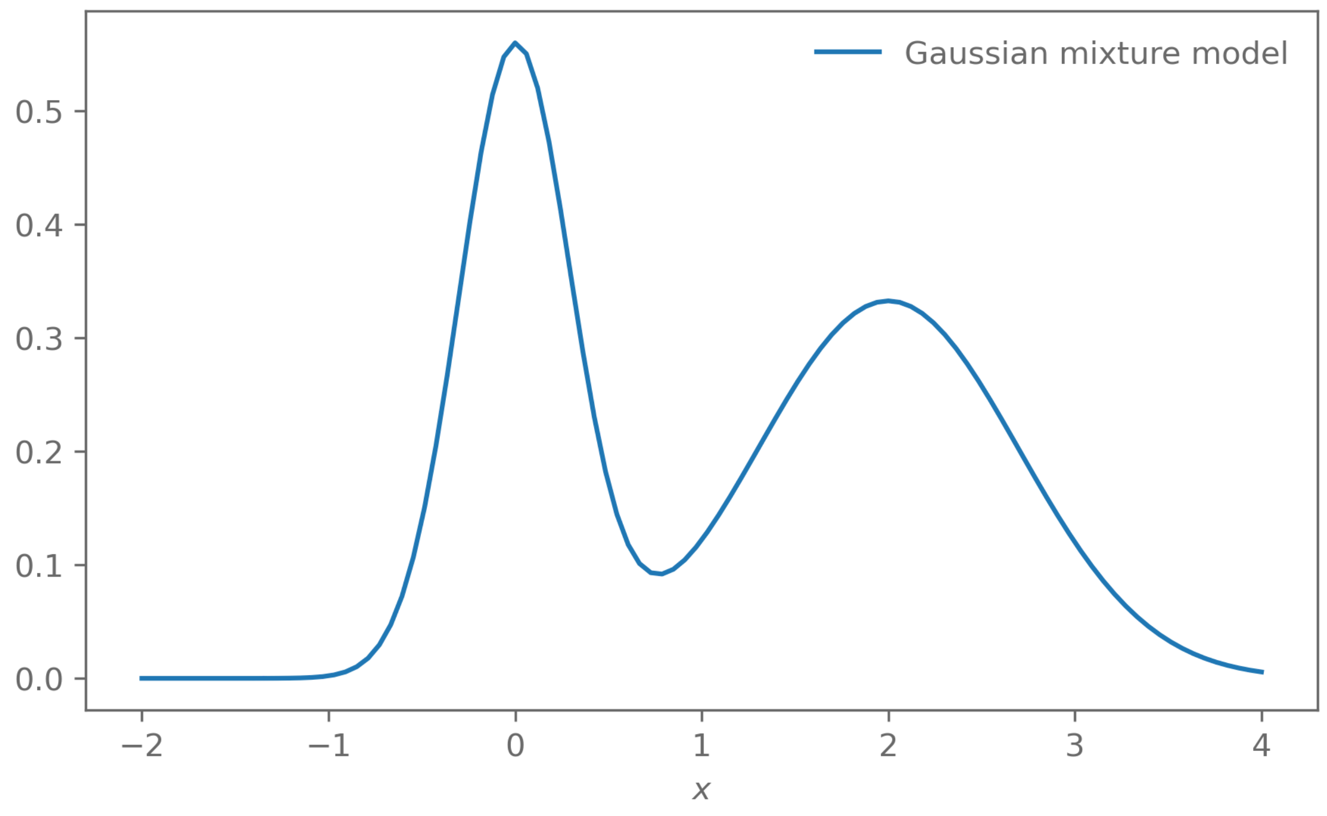
The second term $\mathrm{E}_p[\log p(x)]$ does not depend on ϕ, so it can be ignored when optimising:

$$\phi^* = \underset{\phi}{\operatorname{argmin}}\ D_{\mathrm{KL}}(p \| q_\phi) = \underset{\phi}{\operatorname{argmin}}\ \mathrm{E}_p\left[-\log q_\phi(x)\right]$$

Given samples $x_i \sim p$ from the target distribution, the loss function for the optimisation is then

$$L(\phi) = \mathrm{E}_p\left[-\log q_\phi(x)\right] \approx -\frac{1}{n} \sum_i \log q_\phi(x_i)$$

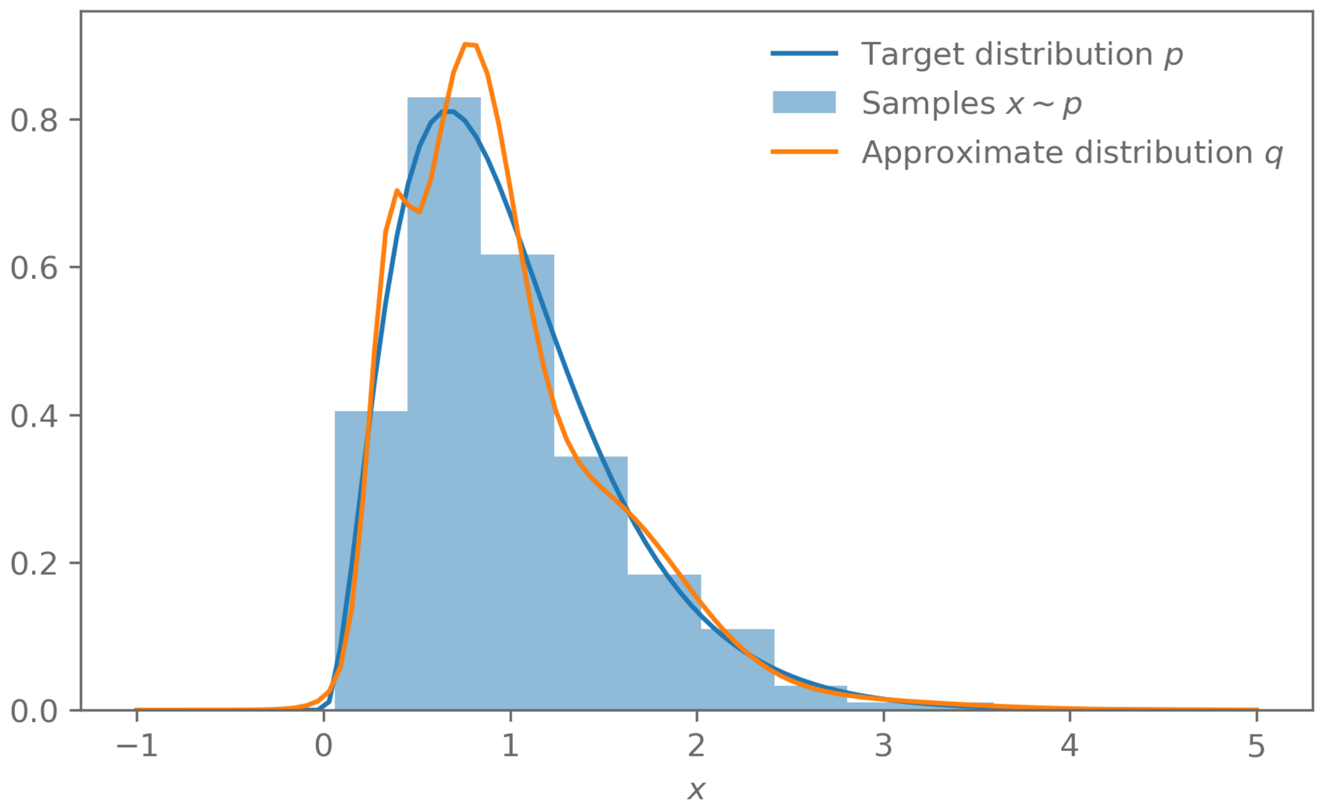
```
# We will use tensorflow probability wit JAX
import tensorflow_probability.substrates.jax as tfp
tfd = tfp.distributions

# Define a Gaussian mixture model
# We use logits (log of probabilities) and log of standard deviations as
# parameters to make optimisation easier
def gmm(logits, means, log_stds):
    distr = tfd.MixtureSameFamily(
        mixture_distribution=tfd.Categorical(logits=logits),
        components_distribution=tfd.Normal(loc=means, scale=jnp.exp(log_stds))
    )
    return distr

# Put in some test parameters
mixture = gmm(
    logits=jnp.log(jnp.array([0.5, 0.7])),
    means=jnp.array([0.0, 2.0]),
    log_stds=jnp.log(jnp.array([0.3, 0.7])),
)
```



Let us now fit a distribution.

```
target_distr = scipy.stats.gamma(a=3, scale=1/3)
target_distr_samples = target_distr.rvs(1000)
```

```
import jaxopt

# Define the loss function
def loss_fn(params):
    variational_distr = gmm(**params)
    loss = jnp.mean(-variational_distr.log_prob(target_distr_samples))
    return loss

key, *subkeys = jax.random.split(jax.random.PRNGKey(42), 4)

# We will use a 5-component Gaussian mixture model
n_components = 5
theta_init = dict(
    logits=jax.random.normal(key=subkeys[0], shape=(n_components,)),
    means=jax.random.normal(key=subkeys[1], shape=(n_components,)),
    log_stds=jax.random.normal(key=subkeys[2], shape=(n_components,)),
)

# Fit the model
solver = jaxopt.ScipyMinimize(fun=jit(loss_fn))
solution = solver.run(theta_init)
```



Once we have fit q_ϕ we can use it to create new samples very quickly.

We can also do conditional density estimation by defining a function that first maps x to the parameters ϕ of our approximate distribution.

If we use a neural network for that mapping from x to ϕ and/or to define flexible distributions, we talk about (conditional) neural density estimation.

When used to approximate the likelihood $p(x|\theta)$, this is called neural likelihood estimation.

- Once we have the approximate likelihood $q_\phi(x|\theta)$, we can continue to our usual Bayesian workflow with sampling the posterior, since now we have a likelihood where we can compute the density.

When used to approximate the posterior $p(\theta|x)$ directly, this is call neural posterior estimation.

- This has the advantage that once we have found $q_\phi(\theta|x)$, we can reuse this for many different observations x^{obs} without having to an MCMC.
- If we change the priors, we need to redo the process, however.

Neural density estimation is often amortised: after creating the training set of pairs of (θ_i, x_i) and fitting the approximate distribution q_ϕ, we can do inference very quickly for different observed data and parameters.

In contrast, ABC is not amortised: the posterior we estimate is specific to a single observation and everything needs to be redone for different observations.

Further resources

A python package that implements a number of simulation-based inference algorithms (using pytorch) is sbi.

A brief overview of recent developments Cranmer et al. 2020 and some discussion on when things can go wrong Hermans et al. 2022.

Exercises

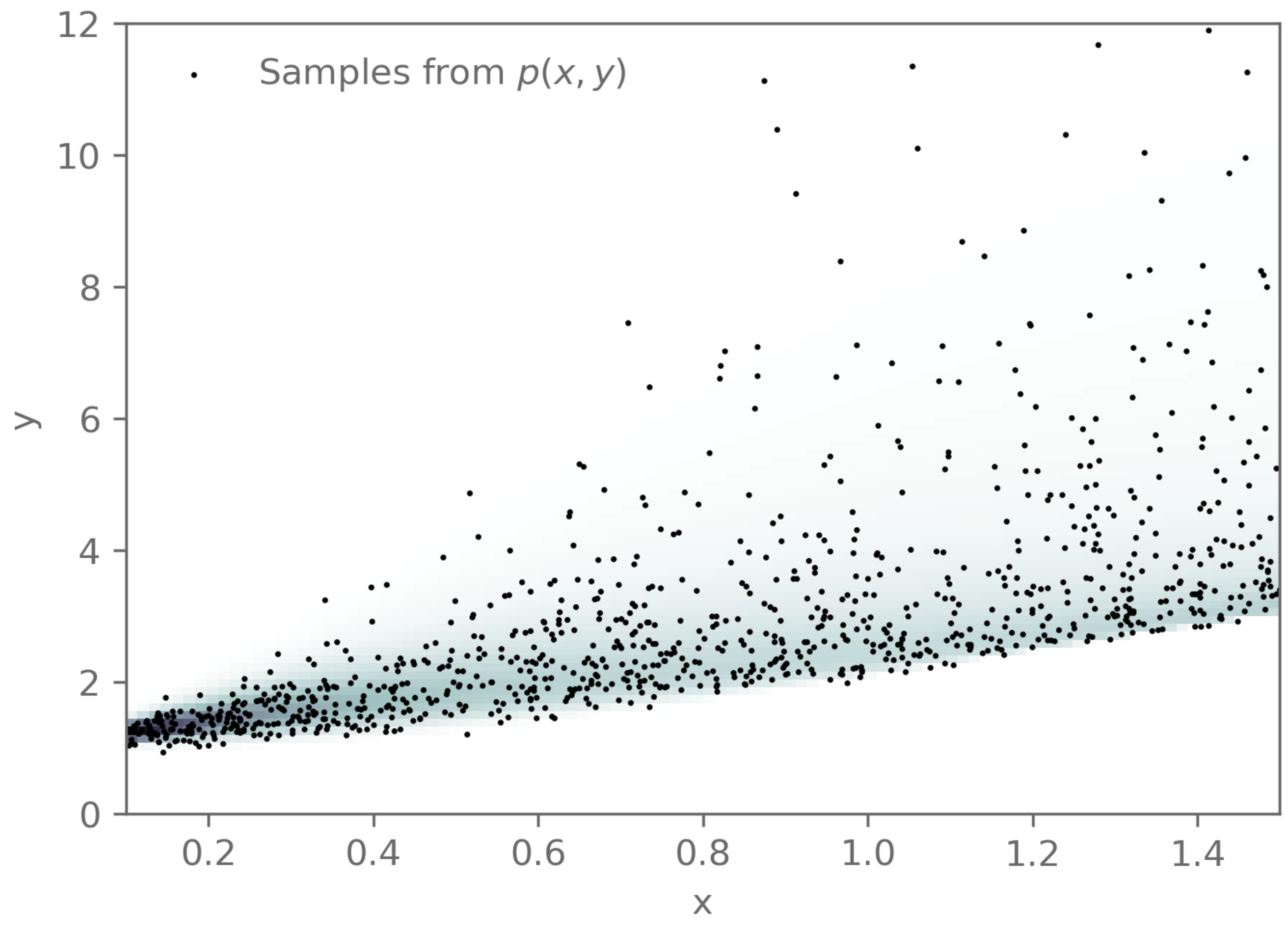
Try rejection ABC on the linear-fit example. Think about what distance metric you want to use, since the data vector is now multidimensional.

Loss functions and posteriors

Let us say we have a dataset (x_i, y_i) drawn from some (unknown) distribution $p(x, y)$.

Ideally, we want the learn the full posterior $p(y|x)$.

But often we start with fitting a function f that maps from x to y (in a loose sense, since there might not be tight relationship between x and y).



Let us use a very general function f, a fully connected neural network.

For an introduction to deep learning in JAX, have a look at the very extensive lectures from the University of Amsterdam: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/JAX/tutorial2/Introduction_to_JAX.html#

```python
import flax.linen as nn

# Define a two-layer multilayer perceptron (MLP)
class MLP(nn.Module):
    n_hidden: int = 128
    n_output: int = 1

    @nn.compact
    def __call__(self, x):
        x = nn.Dense(features=self.n_hidden)(x)
        x = nn.swish(x)
        x = nn.Dense(features=self.n_hidden)(x)
        x = nn.swish(x)
        x = nn.Dense(features=self.n_output)(x)
        return x
```

We train the network using the L_2 loss:

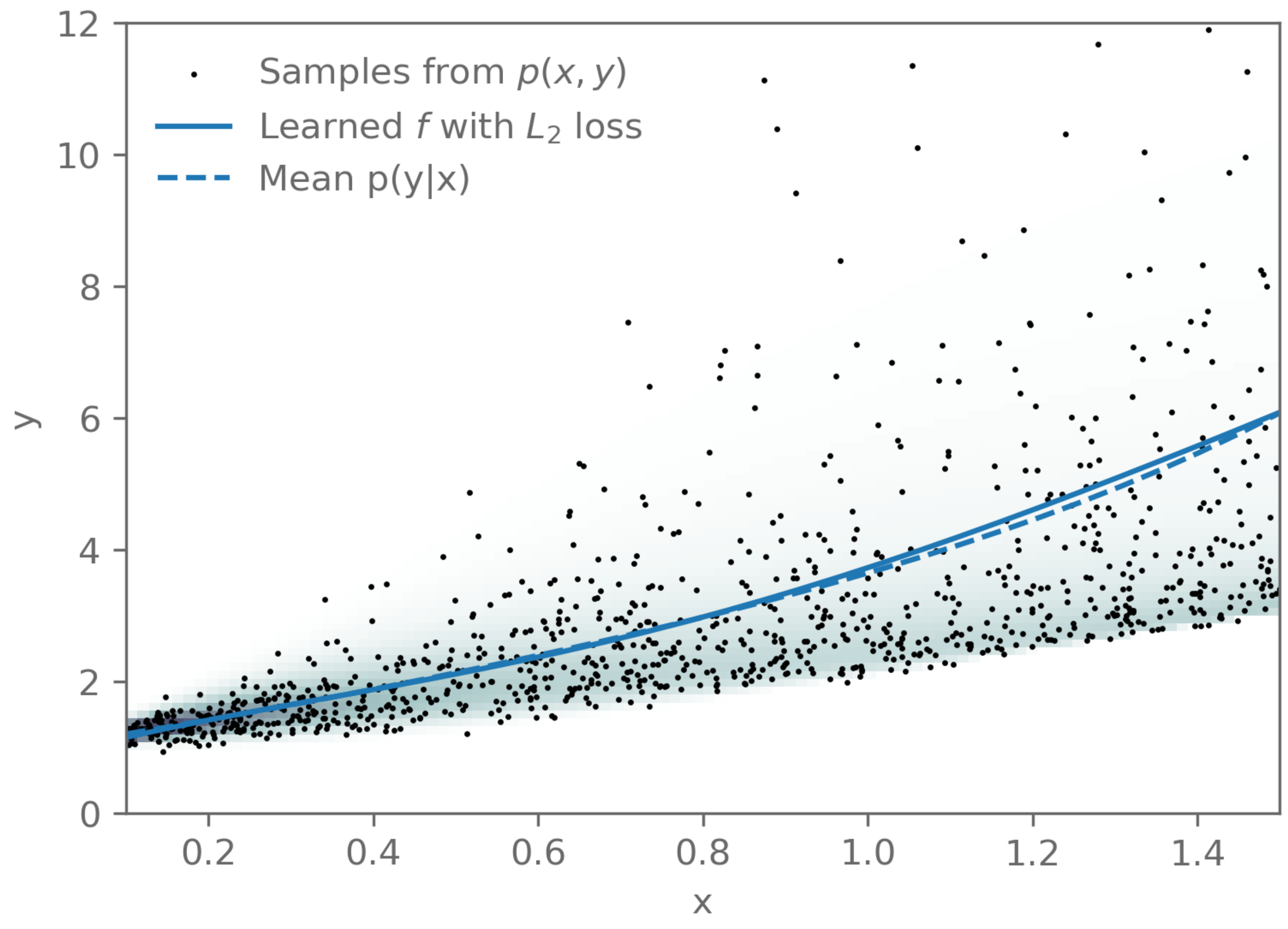
$$L_2(y, f(x)) = \|y - f(x)\|^2 \ .$$

This corresponds to least-squares if with homoscedastic errors.

```python
def L2_loss_fn(params, model, x, y):
    predictions = model.apply(params, x)

    mse = (predictions - y)**2
    return jnp.mean(mse)
```

```python
import optax
import tqdm

# Get n samples from our target distribution
def get_batch(n=64):
    x = np.random.uniform(0.1, 1.5, size=(n, 1))
    y = distr(x).rvs()
    return x, y


def train(model, loss_fn, n_iter, seed):
    # Initialise the model
    x, y = get_batch()
    params = model.init(seed, x)


    # Set up the optimiser. Here we use Adam, which is a variant of
    # stochastic gradient descent
    optimizer = optax.adam(learning_rate=0.001)
    opt_state = optimizer.init(params)

    # This function takes care of applying the parameters with the gradients
    @jax.jit
    def update_model(params, opt_state, x, y):
        # Computes the gradients of the model
        loss, grads = jax.value_and_grad(loss_fn)(params, model, x, y)

        # Computes the weights updates and apply them
        updates, opt_state = optimizer.update(grads, opt_state)
        params = optax.apply_updates(params, updates)

        return params, opt_state, loss

    # Now run the actual optimisation
    progress = tqdm.tqdm(range(n_iter))
    for i in progress:
        # Get a batch of data
        x, y = get_batch(n=1024)

        # Apply the update function
        params, opt_state, loss = update_model(params, opt_state, x, y)

        # Print the current loss
        progress.set_postfix({"loss": loss})
    return params, loss
```

```
model = MLP()

params, loss = train(model, L2_loss_fn, n_iter=2000, seed=jax.random.PRNGKey(42))
L2_trained_model = model.bind(params)
```



Why is that?

The expected loss over the population distribution is given by

$$L_2 = \mathrm{E}_{p(x,y)}[\|y - f(x)\|^2] = \int \|y - f(x)\|^2 p(x, y)\mathrm{d}x\mathrm{d}y$$

We want to find a function f that minimises the functional

$$I[f] = \int \|y - f(x)\|^2 p(x, y)\mathrm{d}x\mathrm{d}y = \int \left(\int \|y - f(x)\|^2 p(y|x)p(x)\mathrm{d}y \right) \mathrm{d}x = \int \mathcal{L}(x, y, f)\mathrm{d}x$$

where we defined

$$\mathcal{L}(x, y, f) = p(x) \int \|y - f(x)\|^2 p(y|x)\mathrm{d}y$$

The Euler-Lagrange equation tells us that the f that minimises I satisfies

$$\frac{\partial \mathcal{L}}{\partial f} = 0$$

since $\mathcal{L}$ does not depend in derivatives of f.

This leaves us with

$$\frac{\partial \mathcal{L}}{\partial f} = -2p(x) \int \|y - f(x)\| p(y|x) \mathrm{d}y = 0$$

and we get

$$\begin{aligned} p(x) f(x) \int f(x) p(y|x) \mathrm{d}y = p(x) f(x) \\ = p(x) \int y p(y|x) \mathrm{d}y = p(x)\,\mathrm{E}_{p(y|x)}[y] \\ f(x) = \mathrm{E}_{p(y|x)}[y] \end{aligned}$$

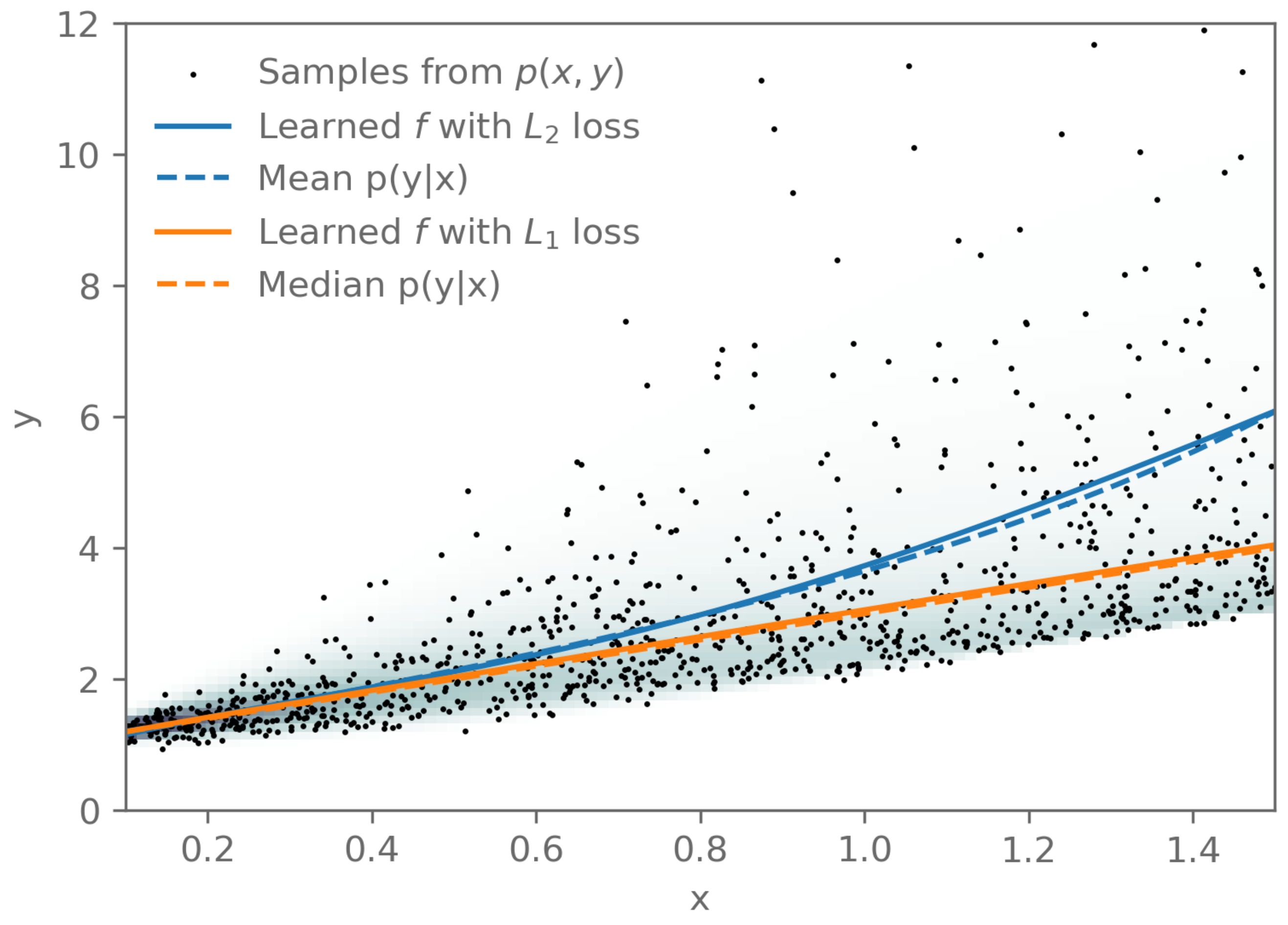
The function f that minimises the L_2 loss is indeed the mean of the conditional distribution $p(y|x)$.

What if we had used another loss function? For example the L_1 loss

$$L_1(y, f(x)) = |y - f(x)|$$

```python
def L1_loss_fn(params, model, x, y):
    predictions = model.apply(params, x)

    return jnp.mean(jnp.abs(predictions - y))


params, loss = train(model, L1_loss_fn, n_iter=2000, seed=jax.random.PRNGKey(42))
L1_trained_model = model.bind(params)
```

```
100%|██████████| 2000/2000 [00:07<00:00, 253.06it/s, loss=0.867985]
```



What about the mode? Or better yet, the full distribution?

Instead of having our neural network predict a point estimate of y given x, we have the network predict a distribution $q(y|x)$ of y given x.

We achieve this by having the network f predict parameters ϕ based on the input x, which parametrise the distribution $q_\phi(y|x)$.

We then optimise the network using the KL loss from the discussion on conditional density estimation:

$$L_{\mathrm{KL}}(x, y) = -\frac{1}{n} \sum_i \log q_{\phi=f(x)}(y_i)$$

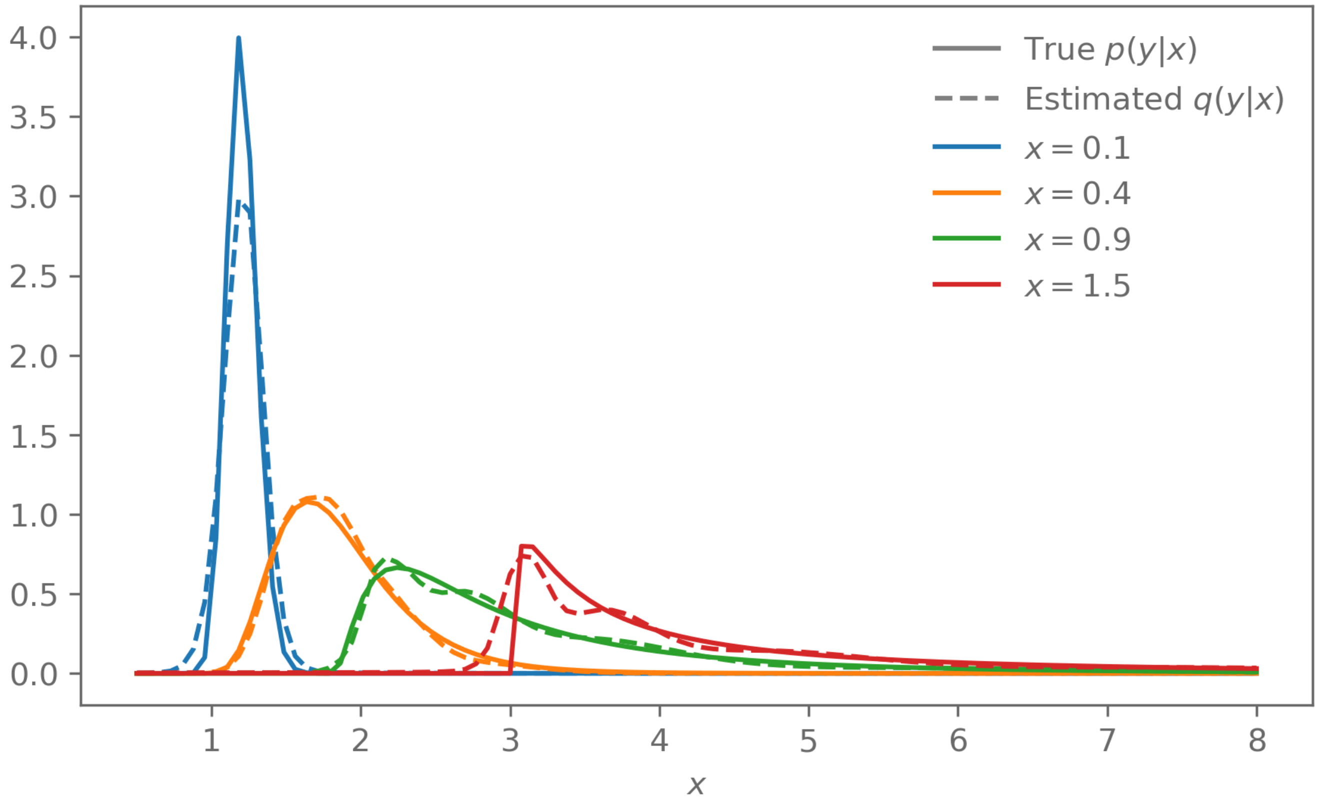
Here we use Gaussian mixture model again for q_ϕ. There are more flexible options, such as normalising flows as well but these are a bit more involved to implement.

```python
def NLL_loss_fn(params, model, x, y):
    q = model.apply(params, x)
    # Compute the negative log likelihood of the outputs
    nll = - q.log_prob(y[:, 0])
    return jnp.mean(nll)
```

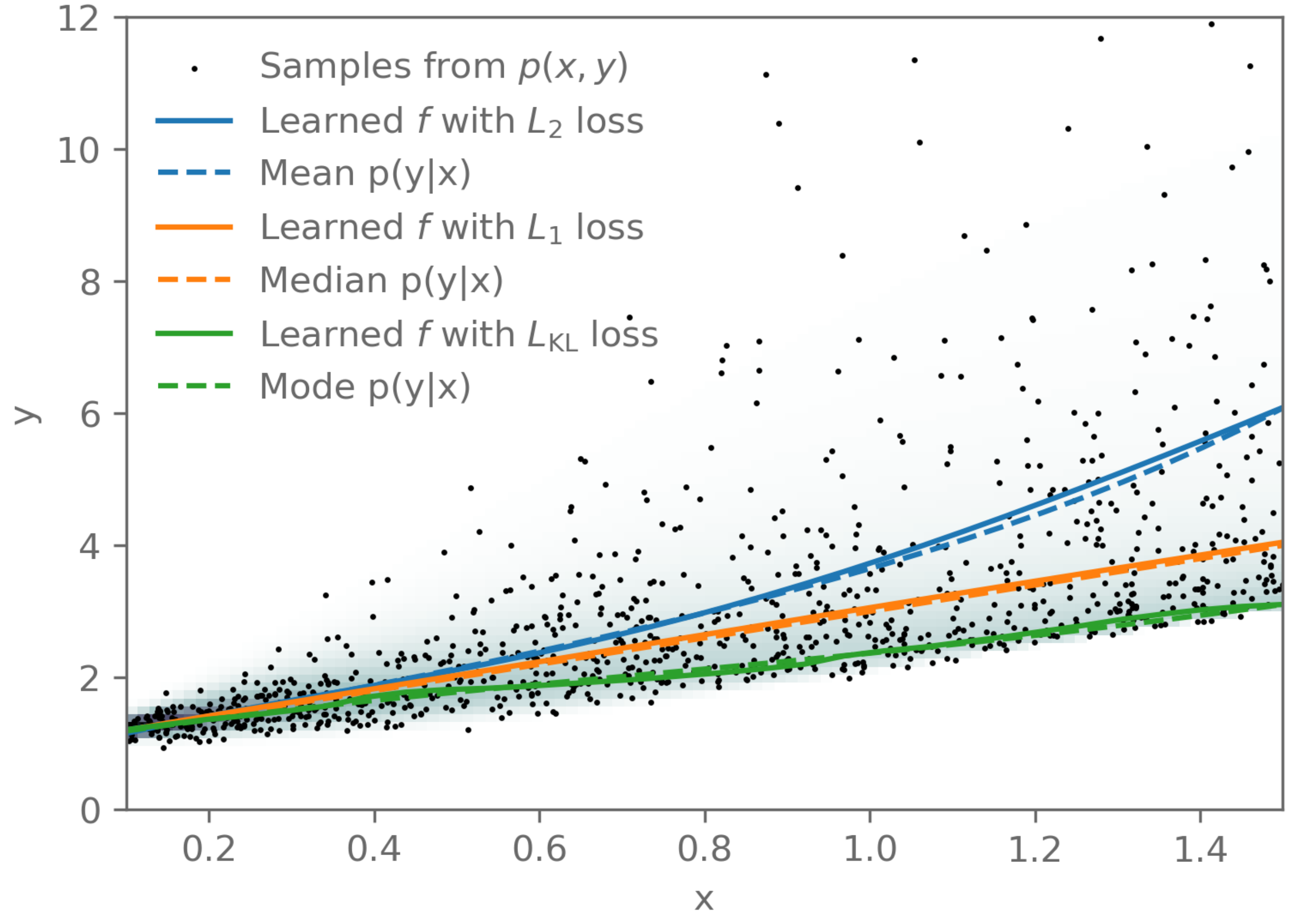
```python
# We need to adapt our model a bit to make the output a distribution.
class MDN(nn.Module):
    n_hidden: int = 128
    n_components: int = 8
    n_output: int = 1

    @nn.compact
    def __call__(self, x):
        x = nn.Dense(features=self.n_hidden)(x)
        x = nn.swish(x)
        x = nn.Dense(features=self.n_hidden)(x)
        x = nn.swish(x)
        x = nn.tanh(nn.Dense(features=self.n_components)(x))

        # Predict the weights, means, and standard deviations of the
        # Gaussian mixture model
        categorical_logits = nn.Dense(self.n_components)(x)
        loc = nn.Dense(self.n_components)(x)
        scale = nn.softplus(nn.Dense(self.n_components)(x))

        # Build the distribution based on these parameters
        dist = tfd.Independent(
            tfd.MixtureSameFamily(
                mixture_distribution=tfd.Categorical(logits=categorical_logits),
                components_distribution=tfd.Normal(loc=loc, scale=scale)
            )
        )
        return dist
```

```python
model = MDN(n_components=16)

params, loss = train(model, NLL_loss_fn, n_iter=2000, seed=jax.random.PRNGKey(42))
NLL_trained_model = model.bind(params)
```

```
  0%|          | 0/2000 [00:00<?, ?it/s]100%|██████████| 2000/2000 [00:08<00:00, 247.
38it/s, loss=0.9764868]
```



True $p(y|x)$
Estimated $q(y|x)$
$x = 0.1$
$x = 0.4$
$x = 0.9$
$x = 1.5$
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
1
2
3
4
5
6
7
8
x



Samples from $p(x, y)$
Learned f with L_2 loss
Mean p(y|x)
Learned f with L_1 loss
Median p(y|x)
Learned f with L_{KL} loss
Mode p(y|x)
0
2
4
6
8
10
12
y
0.2
0.4
0.6
0.8
1.0
1.2
1.4
x

Exercise

Show that the function that minimises the L_1 loss is the median.

Try the neural network predictor on the toy data sets we looked at so far. Does it work with few samples? How many data points do you need for it to work reliably? You can reuse the function from the ABC exercise to create new data points.