

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 can usually still 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 $x_i|\theta \sim p(\cdot|\theta)$

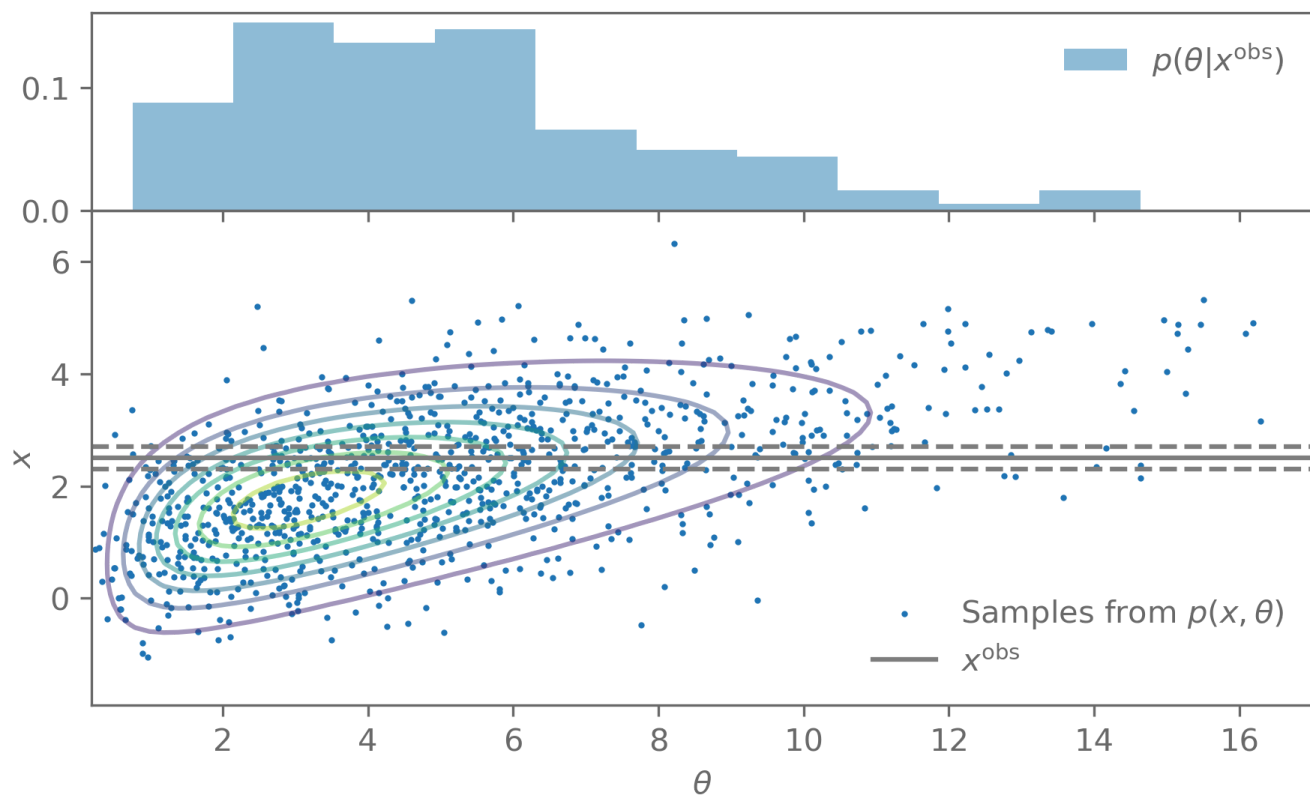
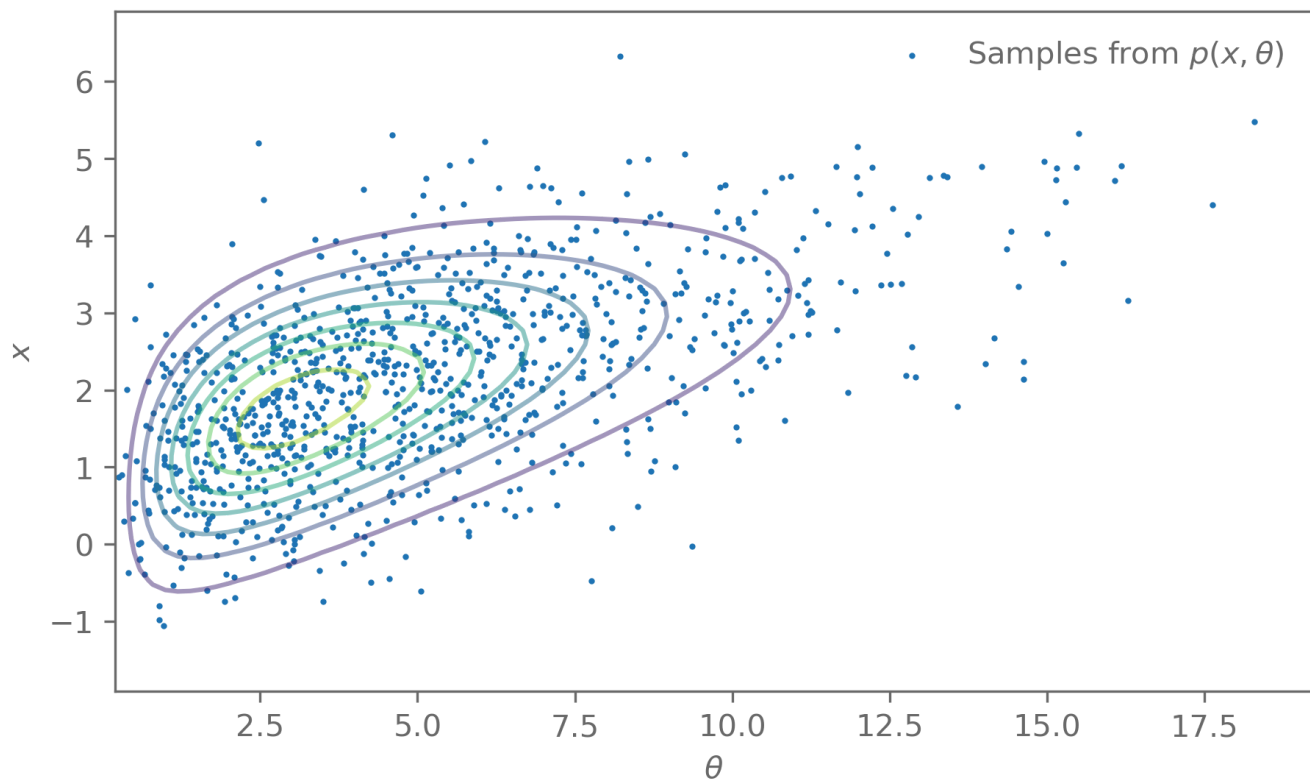
In simulation-based inference we make posterior inference using only the data-generating process, without evaluating likelihood functions.

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 $d(y, y_i^{\text{rep}})$ 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](#).

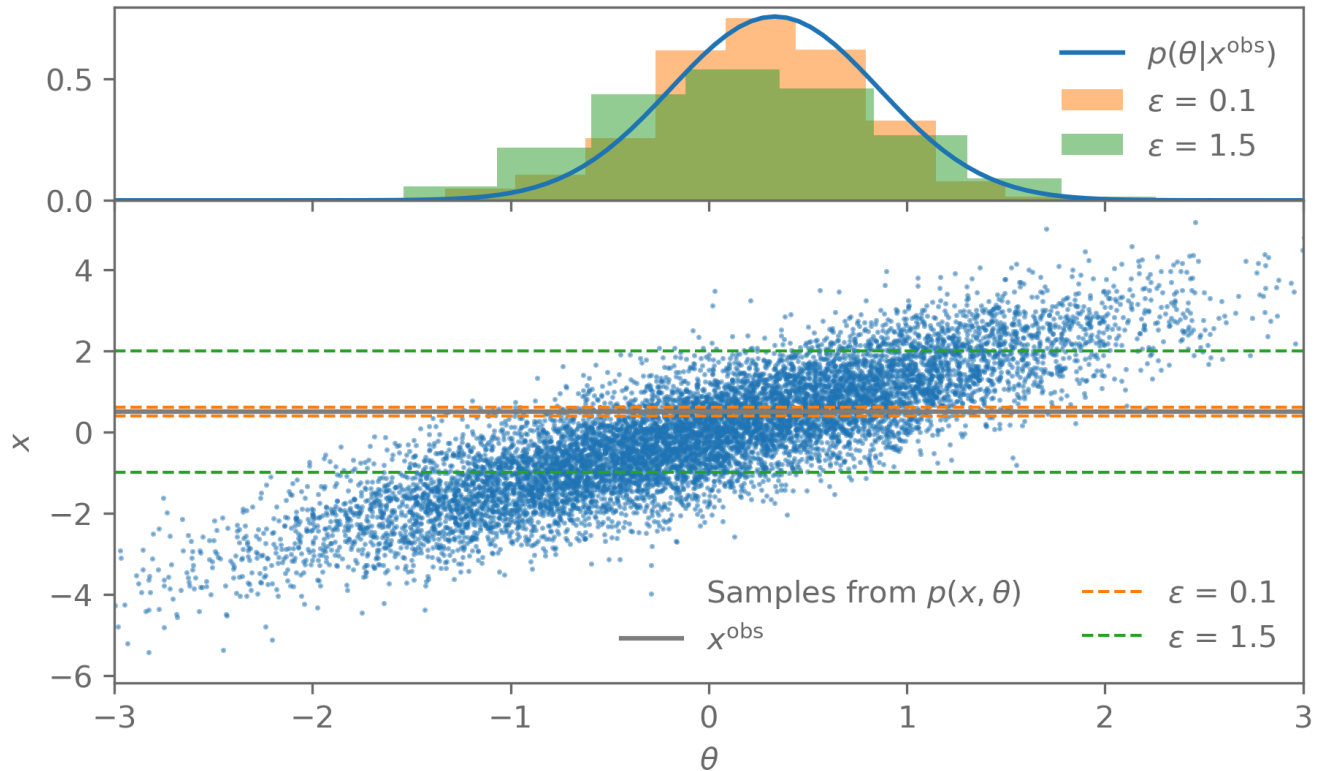


- If ϵ is chosen too small, the acceptance rate will be low
- If ϵ is chosen too large, the approximation to the posterior will be poor

- 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 $\epsilon = 0.1$: 509/10000

Number of accepted samples for $\epsilon = 1.5$: 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

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

between the true distribution p and our approximation q .

The KL divergence measures how well the two distributions agree:

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

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

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

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

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

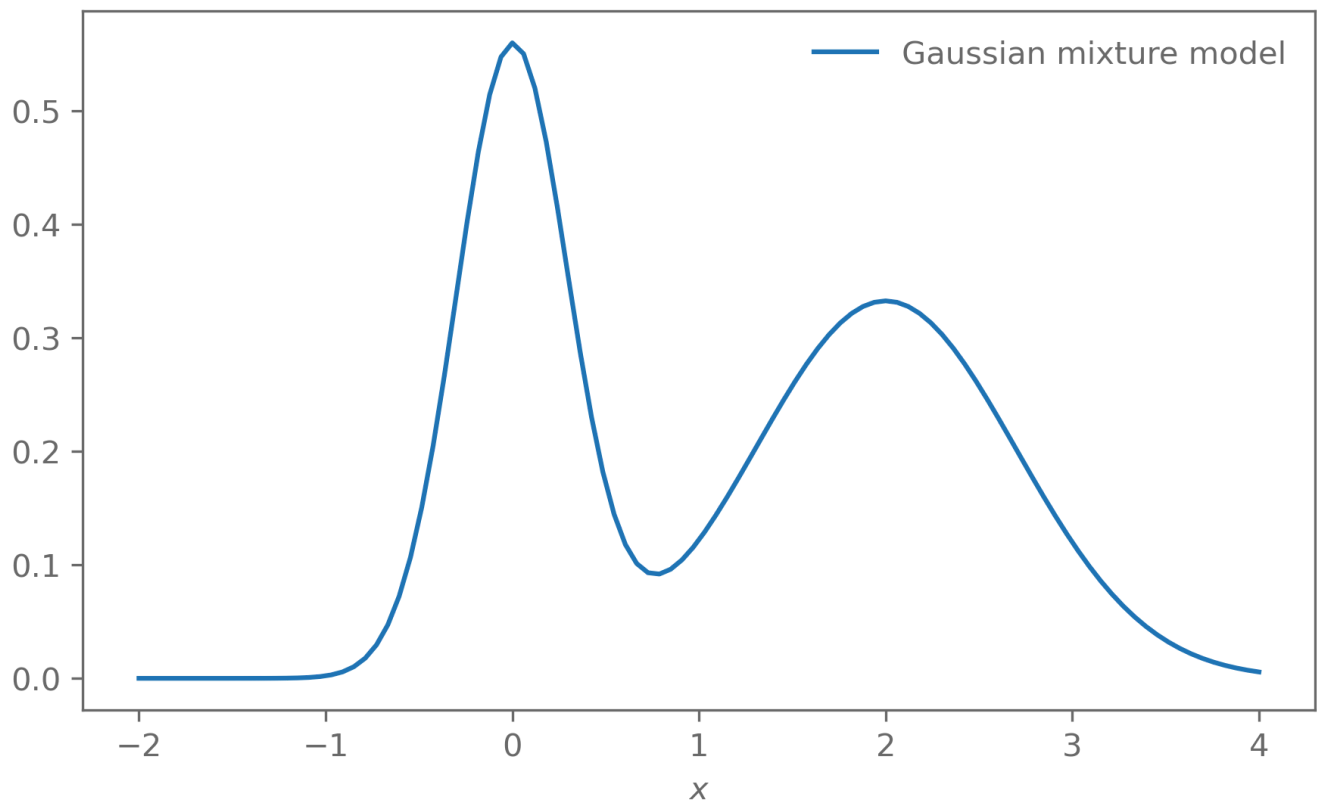
Given samples $x_i \sim p$ from the target distribution, the loss function for the optimisation is then

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

```
# We will use tensorflow probability with 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)
```

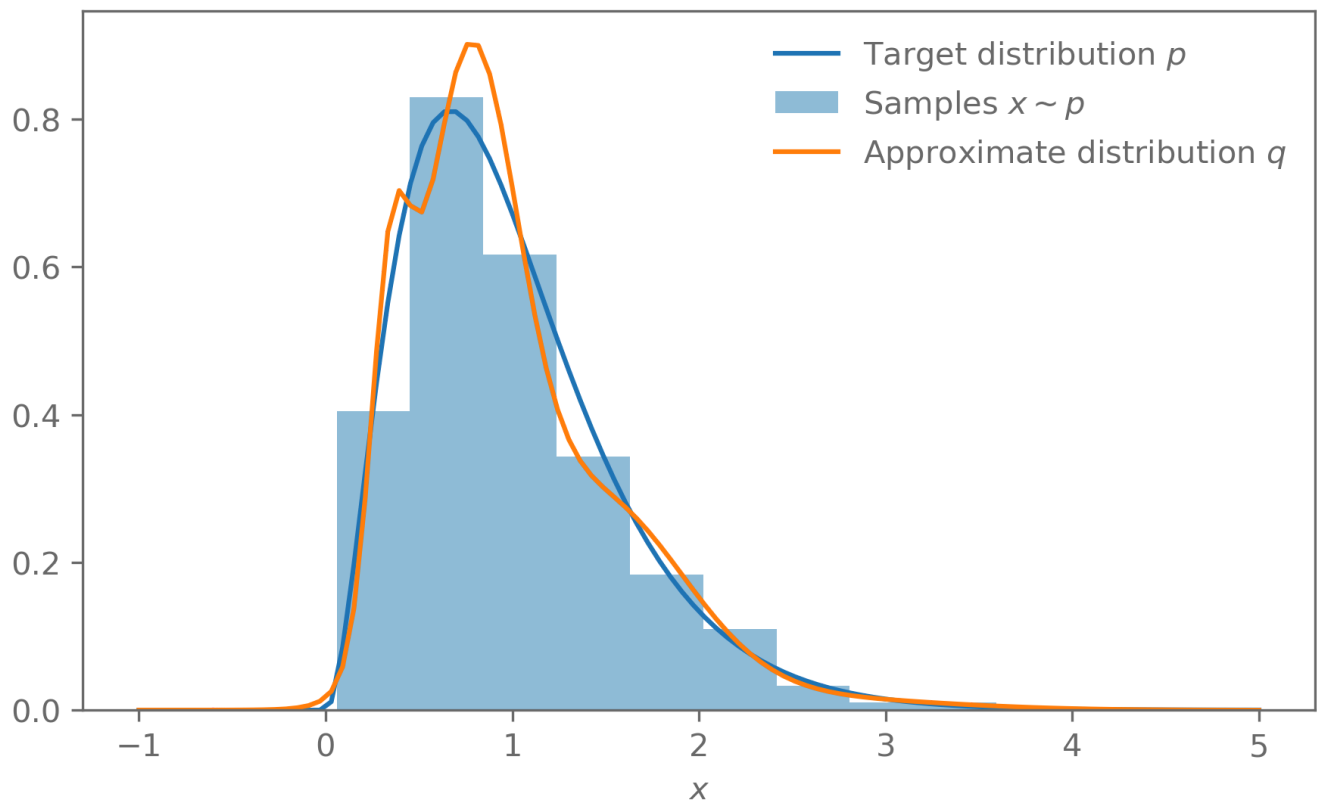
```
import jaxopt

# 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 called 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 run 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

A (very) quick intro to machine learning and neural networks

Machine learning is fundamentally about finding pattern in data.



There are different flavours of machine learning, today we focus on supervised learning.

In supervised learning, we have

- a training set $D = \{x_i, y_i\}$ of pairs of inputs x_i and outputs y_i ,
- a function $f_\theta(x)$ with parameters θ , such that $y = f_\theta(x) + \epsilon$, where ϵ is some noise.

We want to predict the outputs y^* for inputs x^* not in the training set: $p(y^*|x^*, x, y)$.

The input-output pairs can be anything: from 1D points x and y that we model with linear regression to assigning labels to images:

- $x_i =$  $, y_i = \text{"cat"}$
- $x_j =$  $, y_j = \text{"dog"}$
- etc



The distribution over outputs from unseen inputs $p(y^*|x^*, x, y)$ is just the posterior predictive distribution that we have seen and used before:

$$p(y^*|x^*, x, y) = \int p(y^*|x^*, \theta)p(\theta|x, y)d\theta.$$

The direct Bayesian approach we learned is:

- define likelihood $p(y|x, \theta)$ and prior $p(\theta|x)$
- sample posterior $\theta_i \sim p(\cdot|x, y)$
- sample prediction from likelihood $y_i^* \sim p(\cdot|x^*, \theta_i)$.

In machine learning, especially deep learning, $f_\theta(x)$ can be very complex:

- $f_\theta(\text{img}) = \text{"cat"}$

- $f_\theta(\text{img}) = \text{"dog"}$


The function $f_\theta(x)$ can have millions or billions of parameters θ . Not even Hamilton Monte Carlo will help you there.

In machine learning, the problem is therefore usually phrased as an optimisation problem where we want to minimise some loss function $L(x, y, \theta)$ such that the differences between the training outputs y_i and $f_\theta(x_i)$ is minimised.

Common loss functions for regression are

- L_2 loss: $L(x, y, \theta) = \|f_\theta(x) - y\|^2$
- L_1 loss: $L(x, y, \theta) = \|f_\theta(x) - y\|$

To find $f_\theta(x)$, we minimise the expectation of the loss function over the (training) data distribution $p(x, y)$:

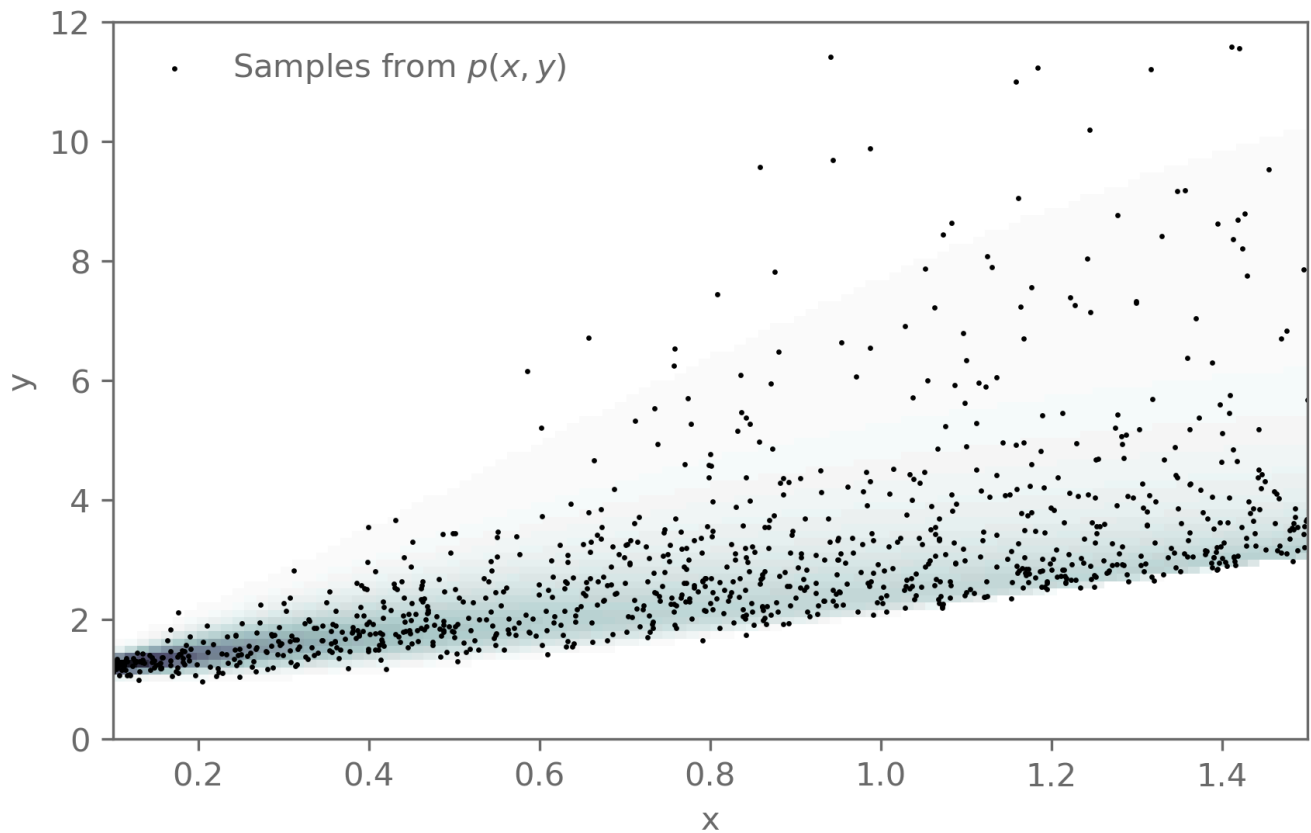
$$\theta^* = \underset{\theta}{\operatorname{argmin}} \mathbb{E}_{p(x,y)}[L(x, y, \theta)].$$

Or for a finite training set $x_i, y_i, i = 1, \dots, n$:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_i^n L(x_i, y_i, \theta).$$

How does the choice of the loss function affect what kind of function $f_\theta(x)$ we get?

Let us look at this (very simple) dataset of points (x_i, y_i) drawn from some (unknown) distribution $p(x, y)$.



We want to find a function $f_{\theta}(x)$ that best describes the relationship between x and y .

If we have no physical model, we can use a general, flexible function. The more complex the data is, the more flexible the function needs to be.

One way to define very flexible functions is using artificial neural networks.

The output \vec{y} is computed from the input \vec{x} as

$$\begin{aligned}\vec{h}^{(1)} &= \sigma(W^{(1)}\vec{x} + \vec{b}^{(1)}) \\ \vec{h}^{(2)} &= \sigma(W^{(2)}\vec{h}^{(1)} + \vec{b}^{(2)}) \\ &\dots \\ \vec{h}^{(L)} &= \sigma(W^{(L)}\vec{h}^{(L-1)} + \vec{b}^{(L)}) \\ \vec{y} &= W^{(L+1)}\vec{h}^{(L)} + \vec{b}^{(L+1)}.\end{aligned}$$

The hidden layers $\vec{h}^{(i)}$ are computed by first linearly transforming the input using the weight matrices $W^{(i)}$ and biases $\vec{b}^{(i)}$ and then applying a *non-linear* activation function $\sigma(\cdot)$.

This is repeated for L layers, where the hidden layer $\vec{h}^{(i)}$ is used as input compute the next layer $\vec{h}^{(i+1)}$.

At the end, there is a final function to map $\vec{h}^{(L)}$ to the output \vec{y} .

This is a very powerful way to define flexible functions.

The universal approximation theorems state that if the activation function $\sigma(\cdot)$ is non-polynomial and the network is sufficiently large, then such a neural network can approximate *any* function.

For our example, we use a fully-connected (or dense) neural network.

Fully-connected means that the weights $W^{(i)}$ are matrices $W^{(i)} \in \mathbb{R}^{n \times m}$ that connect all inputs to a layer to all its outputs.

Other neural network architectures, for example convolutional neural networks, connect subsets of the layer inputs to the outputs.

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#

For our example, we implement a neural network using JAX and the `flax` library.

```
import flax.linen as nn

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

    @nn.compact
    def __call__(self, x):
        x = nn.Dense(features=self.n_hidden)(x)
        x = nn.swish(x) # We use the swish activation function
        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:

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

This corresponds to least-squares if with homoscedastic errors.

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

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

To optimise the parameters θ (which consist of the weights and biases $W^{(l)}$ and biases $\vec{b}^{(l)}$), we use (stochastic) gradient descent.

In gradient descent, we compute the loss on our training set $D = \{x_i, y_i\}$:

$$L(D, \theta) = \frac{1}{n} \sum_i L(x_i, y_i, \theta).$$

We then update the parameters using the gradient of the loss:

$$\theta \rightarrow \theta - \gamma \nabla_{\theta} L(D, \theta),$$

where γ is the *learning rate*. This process is repeated until convergence.

Because the size of training data can be very large in practice, instead of minimising the expected loss over all n training points $\{x_i, y_i\}$, we compute the gradient only on batches B_i of the training data. A batch $B_i \subset D$ is a subset of the training data.

$$\theta \rightarrow \theta - \gamma \nabla_{\theta} L(B_i, \theta).$$

This is repeated for all batches. This is called stochastic gradient descent, because it replaces the loss over the whole data set $L(D, \theta)$ with a noisy approximation $L(B_i, \theta)$.

```
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

```

Define our model for the function $f_{\theta}(x)$ (the neural network) and train it using the L_2 loss:

```

model = MLP()

params, loss = train(model, L2_loss_fn, n_iter=1000, seed=jax.random.PRNGKey(42))
L2_trained_model = model.bind(params)

```

Let us to the same thing for the L_1 loss:

```

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

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

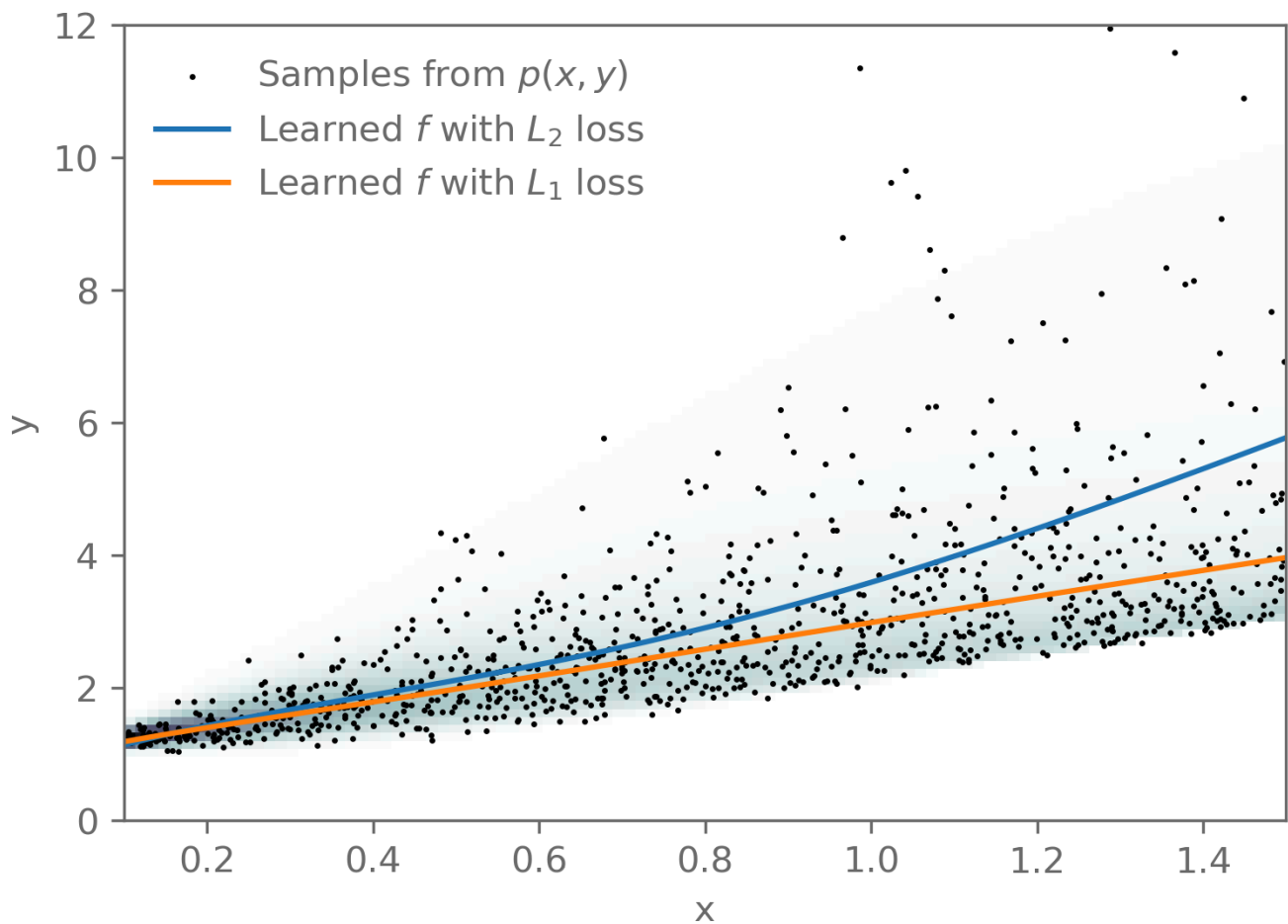
params, loss = train(model, L1_loss_fn, n_iter=1000, seed=jax.random.PRNGKey(42))
L1_trained_model = model.bind(params)

```

```

100%|██████████| 1000/1000 [00:04<00:00, 201.16it/s, loss=1.3839406]

```



Why are the functions learned with different losses different? Or a better question, what functions did we learn?

The expected loss over the population distribution $p(x, y)$ is given by

$$L_2 = \mathbb{E}_{p(x,y)}[\|y - f(x)\|^2] = \int \|y - f\|^2 p(x, y) dx dy.$$

We want to find a function f that minimises the functional

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

where we defined

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

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

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

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

Using

$$\int \|y - f\|^2 p(y|x) dy = \int^f (f - y)^2 p(y|x) dy + \int_f (y - f)^2 p(y|x) dy,$$

we find

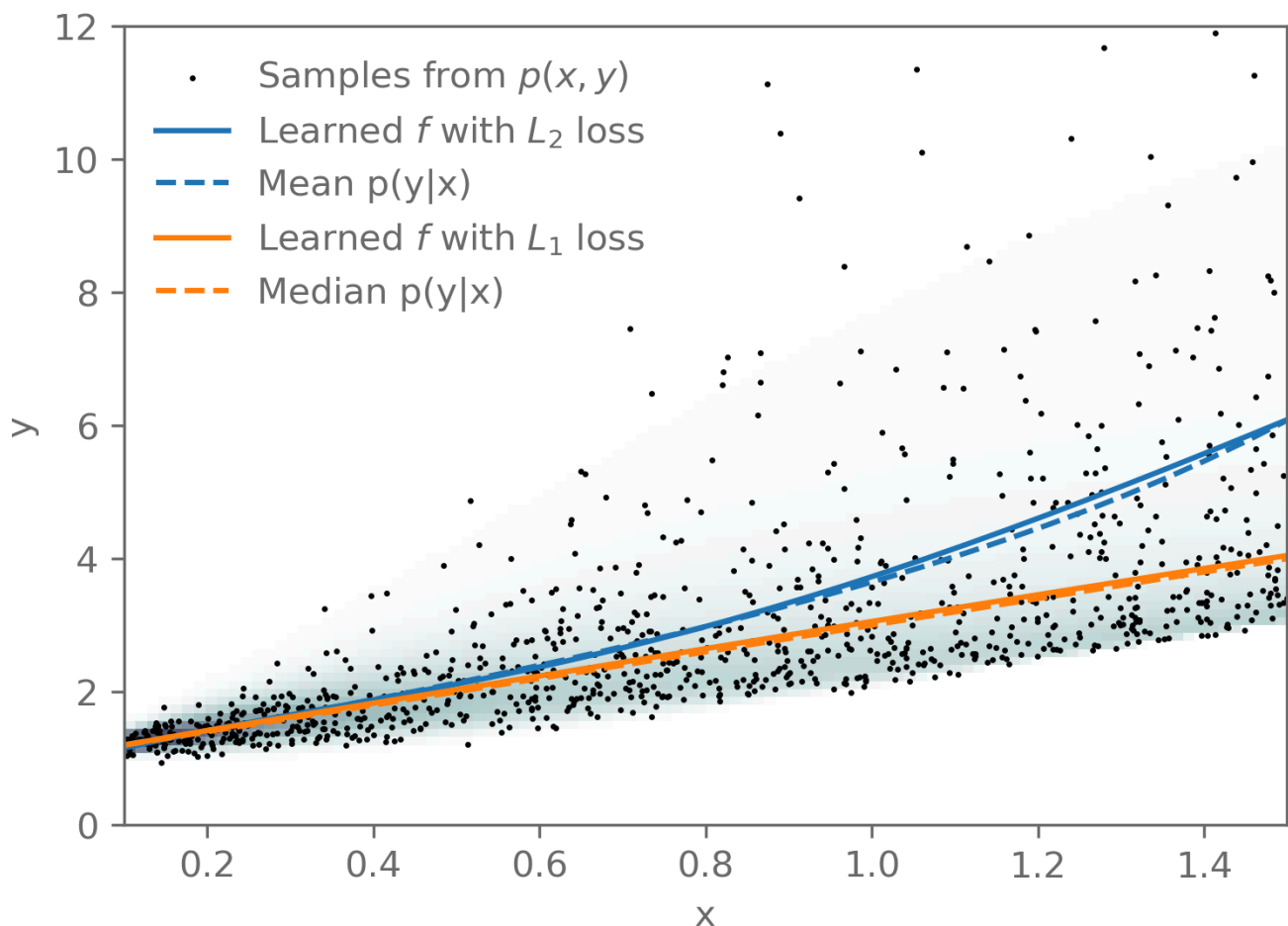
$$\frac{\partial \mathcal{L}}{\partial f} = -2p(x) \int (y - f(x)) p(y|x) dy = 0$$

and we get

$$f(x) = \int y p(y|x) dy = \mathbb{E}_{p(y|x)}[y]$$

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

What if we had used another loss function? For example $L_1(y, f(x)) = |y - f(x)|$?



How does this approach of minimising a loss function compare to what we have been doing in the Bayesian context? For example, finding the maximum of the posterior?

The mode θ^* of the posterior $p(\theta|x, y)$ is

$$\theta^* = \operatorname{argmax}_{\theta} p(\theta|x, y) .$$

Writing this in terms of the logarithm of the likelihood and prior

$$\theta^* = \operatorname{argmax}_{\theta} [\log p(y|x, \theta) + \log p(\theta|x)] .$$

Or equivalently

$$\theta^* = \operatorname{argmin}_{\theta} [-\log p(y|x, \theta) - \log p(\theta|x)] .$$

This can be seen as minimising a loss function (the negative log-likelihood) with a regularisation term (the negative log-prior).

In case of a Gaussian likelihood, this corresponds to a (weighted) L_2 loss.

We have seen that using the L_2 loss yields a function $f_{\theta}(x)$ that approximates the mean of the conditional data distribution $p(y|x)$.

What loss function do we need to use to get an approximation of 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 $f_{\theta}(x)$ 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:

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

Here we use a 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.

```
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)
```

```

# 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

```

```

model = MDN(n_components=16)

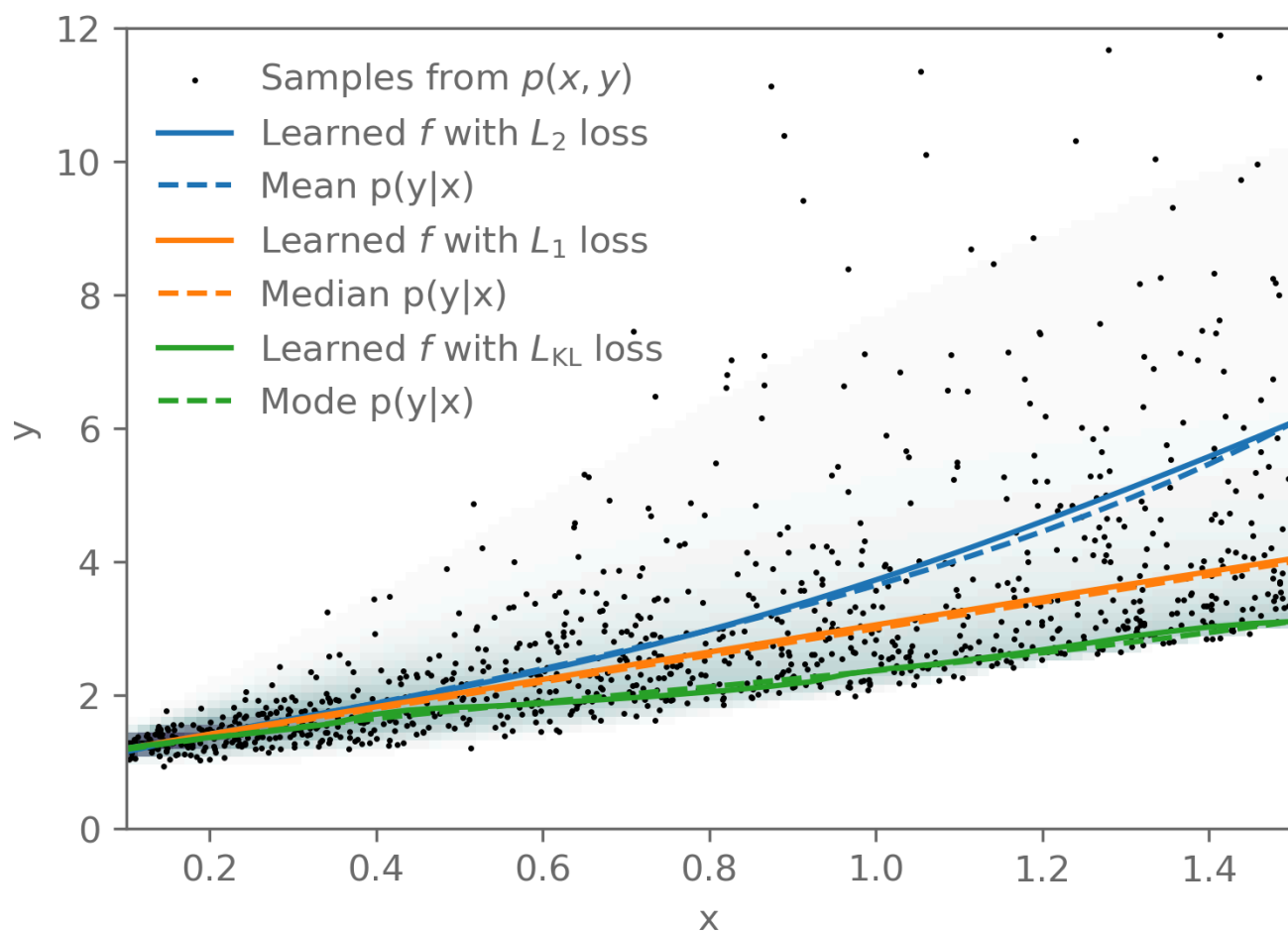
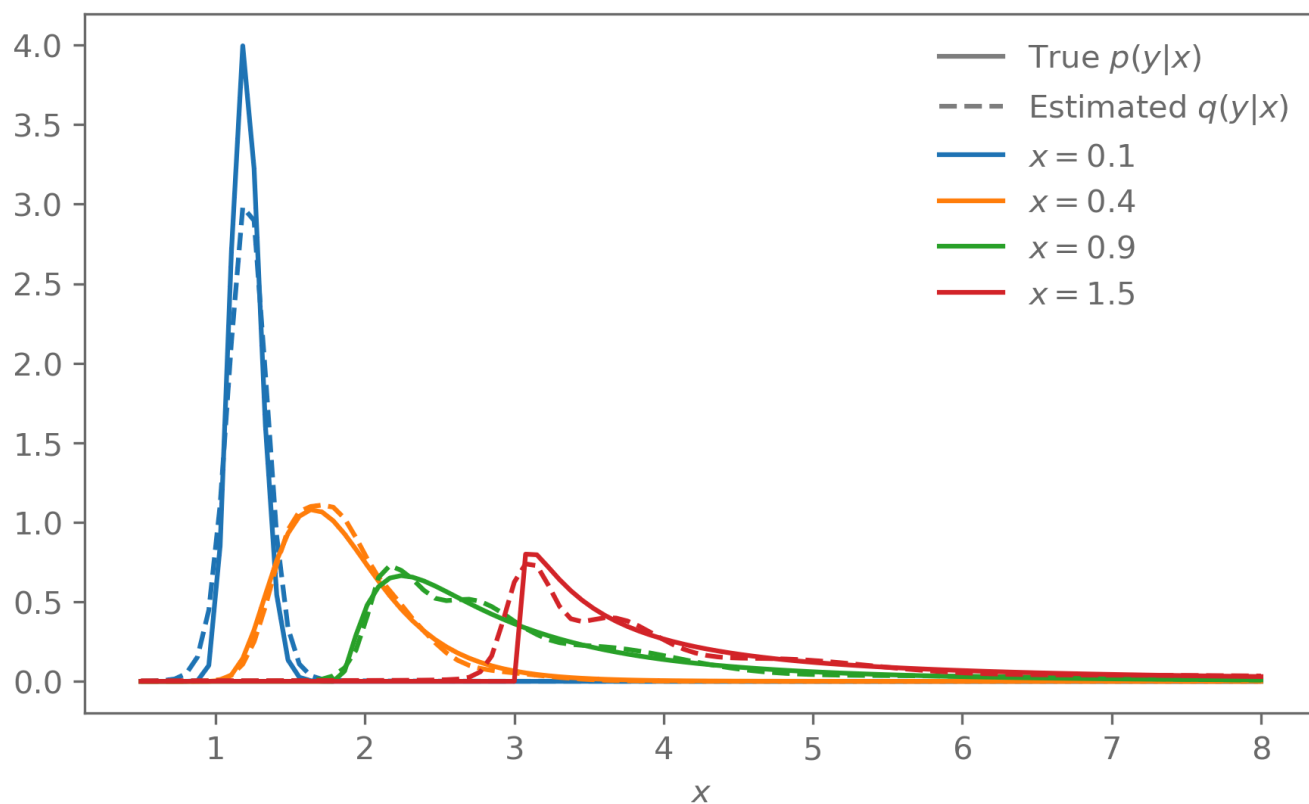
params, loss = train(model, NLL_loss_fn, n_iter=1000, seed=jax.random.PRNGKey(42))
NLL_trained_model = model.bind(params)

```

```

0%|          | 0/2000 [00:00<?, ?it/s]
100%|██████████| 2000/2000 [00:08<00:00, 247.38it/s, loss=0.9764868]

```

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.

Project idea

In this [Google colab notebook](#) is a demonstration on how to use these methods to estimate the mass of galaxy clusters from the number of galaxies and their velocity dispersion. A possible project would be to work through this yourself as well as the associated paper that goes into more of the details.