Exercise Sheet 00

Welcome to the first hands-on session. We want you to gain first experiences in using `jax` and `jax.numpy` in the context of statistical modeling. In this exercise sheet, we will help help you become able to writing code that aligns closely with the code demonstrated in the workshop slides.

Objective

Our primary objective is to code a simple location-scale model for normally distributed observations. These observations are characterized by sharing the same mean μ and scale σ. The model is formulated as follows:

- Observations: $y_i|\mu, \sigma \sim \mathcal{N}(\mu, \sigma^2)$
- Mean prior: $\mu \sim \mathcal{N}(0, 10)$
- Scale prior: $\sigma \sim \mathcal{HC}(0.1)$

Here, $\mathcal{HC}(0.1)$ denotes the half cauchy distribution with scale parameter 0.1, that is a Cauchy distribution with expectation zero and scale 0.1 truncated to support only positive values.

In the first exercise, we focus on coding the model with a known variance. Your tasks involve optimizing the log posterior to determine the posterior mode of μ. For those seeking a more challenging task, the second exercise involves creating the model with an unknown variance. Here, you will use optimization techniques to find the posterior modes of both μ and σ, requiring creativity as optimizing a scalar-valued function of a vector was not covered in the slides.

Prelimaries

Before you can use `jax`, we need to import the modules. In Python this is done using import statements. One can give these different names to make refering to them easier and more readable. We use this to refer to `jax.numpy` with `jnp` and to `tensorflow_probability.substrates.jax` with `tfp.`

```
import jax
import jax.numpy as jnp
import tensorflow_probability.substrates.jax as tfp
```

Next, we create some data. We just draw a few times ($n = 10$) from a normal distribution with mean -1.4 and variance 0.8^2. For that, we specify the parameters first then create the distribution and finally sample from the distribution. Please not that you have to supply a seed to the method `dist.sample`, here the variable `key`. The reason is, that `jax` employs

an explicit pseudo-random number generator (PRNG) state to ensure reproducibility and control in stochastic computations. You are probably used to statefull PRNGs from R and numpy where handling the state happens hidden behind the scenes. With `jax`, the programmer is responsible to advance the state. Please refer to the `jax` manual for more information on this topic.

```
key = jax.random.PRNGKey(0)
mu = -1.4
sigma = 0.8
n = 10

dist = tfp.distributions.Normal(mu, 0.8**2)
obs = dist.sample(n, seed=key)
```

The values, we used in this example are:

```
print(obs)
```

```
[-1.6381509 -1.2308921 -1.5168177 -1.8715646 -1.6817944 -1.4973723
 -1.8296661 -1.778153  -0.9317191 -1.0369263]
```

Exercise 1: Model with Known Variance

In the first exercise, we will focus on coding the model with a known variance. Your task will involve optimizing the log unnormalized posterior to determine the posterior mode of μ. The subtasks of this exercise are:

1. Implement the `log_prior` of the model.

```
def log_prior(params):
    ...
```

- You can use normal distribution implemented in Tensorflow Probability TFP under the name `tfp.distributions.Normal`. For the interface description, refer to the documentation.

2. Implement the `log_likelihood` of the model.

```
def log_likelihood(params, y):
    ...
```

- Again, you can use the normal distribution implemented in TFP.

- `jax`, allows a vectorized evaluation of the `log_prob` method. It will return a vector with the same shape as the argument. Therefore, you do can avoid using a for loop. Remeber that the log likelihood is a scalar and you need to sum the contributions over all observations.

3. Implement the log unnormalized posterior of the model:

```
def log_uposterior(params, y):
    ...
```

4. Create functions to compute the gradient and Hessian of the log unnormalized posterior with respect to the parameter using `jax`' automatic differentiation capabilies.
    - If you want, you can refer to the API documentation of those functions `jax.grad` and `jax.hessian`.
    - Hint: use the argument `argnums` in `jax.grad` and `jax.hessian` to derive functions that return the gradient/Hessian with respect to the model parameters.
    - In the next subtask, we call the functions calculating gradient and Hessian `dlp` and `dlpp`, respectively.
5. Finally, you can choose a starting value and run Newton's method for a few iterations to find the posterior mode of μ. If your functions have the same signature as we suggested above, the optimization loop should look like:

```
params = jnp.array(0.0)

for i in range(10):
    # params - gradient / hessian
    params = params - dlp(params, obs) / ddlp(params, obs)
    print(f"iteration {i + 1}. Parameters: {params}")
```

Exercise 2: Model with Unknown Variance

For those seeking a more challenging task, the second exercise involves creating the model with an unknown variance. Here, you will use optimization techniques to find the posterior modes of both μ and σ.

When having more complicated models, it often makes sense to think about how you want to represent your parameters. Of course, one can use a vector but than one is required to memorize at which position which parameter is stored. `jax` allows us to use other ways to represent a parameter and provides tools to convert it to a vector when needed, e.g., for calculating the gradient or Hessian. In fact, one can use things like pyhon dictionaries, tuple, lists, and custom classes. In general, you can use everything that is a so-called pytree. Check the the solution to this sheet, to learn more.

In this exercise, however, we recommend that you use a vector to represent $(\mu, \log(\sigma))^T$. In the solution, we use a custom class to showcase an alternative way.

The subtasks in this exercise are very similar to Exercise 1, with a few exceptions. First, remember that the optimization requires that all parameter are defined on $\mathbb{R}$. Therefore, we have to work with $\log(\sigma)$ and apply the change of variable theorem.

1. Implement the `log_prior` of the model.

- Remember, this means we have to add the log prior of μ and $\log(\sigma)$.
- Tensorflow probability provides an implementation of the HalfCauchy distribution, you can use `tfp.distributions.HalfCaucht`. For the API, please refer to the documentation.
- The density of the log transformed variable is determined by adding $\log(\sigma)$ to the prior density of σ.

2. Implement the `log_likelihood` of the model.

- This is almost the same likelihood as above but this time the variance depends on the parameters.

3. Implement the unnormalized `log_posterior` of the model:

4. Create functions to compute the gradient and Hessian of the log unnormalized posterior with respect to the parameter using `jax`' automatic differentiation capabilities.

5. Finally, you can choose a starting value and run the optimization for a few iterations to find the posterior mode of μ and σ. In this exercise the Hessian is a matrix and we need to adjust the algorithm slightly. You can use, for example, the gradient ascent method (see Wikipedia) or incorporate the second order derivative information to automatically tune the step-size (see Gradient descent using Newton's method). We opt for the second option and the optimization loop is slightly more involeved than in Exercise 1:

```
params = jnp.array([0.0, 0.0])

for i in range(10):
    grad = dlp(params, obs)
    hess = ddlp(params, obs)
    alpha = (grad ** 2).sum() / (grad.T @ hess @ grad)
    params = params - alpha * grad
    print(f"iteration {i + 1}. Parameters: {params}")
```