Homework on Variational Inference

This document provides detailed explanations for the models implemented in

- `logistic-regression/model_logreg_mvn.py` a Bayesian logistic regression / classification model for linearly separable data
- `vae/model_vae.py` an unsupervised model for handwritten digit generation

The implementations use the `pytorch-lightning` framework and the relevant parts of code are

- `logistic-regression/model_logreg_mvn.py`, functions `forward`, `accuracy` and `loss`as well as `logistic-regression/run.py`
- `vae/model_vae.py` functions `forward` and `loss` as well as `vae/run.py` and`vae/config_vae.yaml`

Bayesian logistic regression / classification

The Bayesian logistic regression model is a classification model defined as

$$p(Y, w|X) = p(w) \prod_{i=1}^{n} p(y_i|x_i, w) \tag{1}$$

$$= \mathcal{N}(w; 0, \sigma_{\text{prior}}\ I_d) \prod_{i=1}^{n} \text{Bernoulli}(y_i; x_i w) \tag{2}$$

where

- $x_i \in \mathbb{R}^d$ are features $X = [x_1, x_2, \ldots, x_n]^T$
- $y_i \in \{0, 1\}$ are binary class labels $Y = [y_1, y_2, \ldots, y_n]^T$.
- $\mathcal{N}(w; 0, \sigma_{\text{prior}}\ I_d)$ denotes a multivariate normal distribution

Bayesian inference in this model means

- computing/approximating the posterior

$$q(w; Y, X) \approx p(w|Y, X) = \frac{p(Y, w|X)}{p(Y|X)} \tag{3}$$

- computing/approximating label predictions for new input features

$$
\begin{aligned}
p(y_*|x_*) &= \int dw\, p(w|Y,X)\, p(y_*|x_*,w) && (4)\\
&= \int dw\, q(w)\, p(y_*|x_*,w) && (5)
\end{aligned}
$$

Variational inference

Since the Bayesian posterior $p(w|Y,X)$ is analytically intractable we approximate it either by a multivariate Normal $q_\phi(w) = \mathcal{N}(w;\mu,\Sigma)$ with $\phi = \{\mu,\Sigma\}$ or diagonal/factorised multivariate Normal $q_\phi(w) = \mathcal{N}(w;\mu,\mathrm{diag}(\sigma^2))$ $\phi = \{\mu,\sigma^2\}$ by optimising the negative evidence lower bound (ELBO)

$$
\begin{aligned}
L(\phi;Y,X) &= -\sum_{i=1}^{n} \mathbb{E}_{q_\phi(w)}[\log p(y_i|x_i,w)] + \mathrm{KL}[q_\phi(w)\|p(w)]. && (6)\\
&\geq \log p(Y|X) && (7)
\end{aligned}
$$

Here we need to numerically approximate $\mathbb{E}_{q_\phi(w)}[\log p(y_i|x_i w)]$ and the predictive distribution We can use

$$
\begin{aligned}
\int dw\, f(x^T w)\, \mathcal{N}(u;\mu,\Sigma) &= \int du\, f(u)\, \mathcal{N}(u; x^T w, x^T \Sigma x) && (8)\\
&= \int d\epsilon\, f(x^T w + \sqrt{x^T \Sigma x}\epsilon)\, \mathcal{N}(\epsilon;0,1) && (9)
\end{aligned}
$$

$$
\int du\, \sigma(u)\, \mathcal{N}(u;\mu,\sigma^2) \approx \sigma\left(\frac{1}{\sqrt{1+\pi\sigma^2/8}}\mu\right) \tag{10}
$$

$$
\int du\, \sigma(u)\, \mathcal{N}(u;\mu,\sigma^2) \approx \sum_{k=1}^{K} \sigma(\mu + \sqrt{2}\sigma\tilde{u}_i)\, \frac{1}{\sqrt{\pi}} w_i, \tag{11}
$$

where $\{\tilde{u}_k, w_k\}_{k=1}^K$ are the weights and nodes of the univariate Gauss-Hermite quadrature.

Key variational inference concepts to learn from this model

The approximation of the EBLO objective in implemented in in the function.

```
def loss(self, features, labels):
```

please follow the implementation there.

Batch learning

For large datasets we cannot use all data in each training step due to memory restrictions, therefore, we use the approximation

$$\sum_{i=1}^{n} \mathbb{E}_{q_\phi(w)}[\log p(y_i|x_i, w)] \approx n \frac{1}{|S|} \sum_{s \in S} \mathbb{E}_{q_\phi(w)}[\log p(y_i|x_s, w)], \tag{12}$$

that is, we approximate the objective by using only a random subset $S \subseteq \{1, \ldots, N\}$ to represent the dataset. This makes the objective stochastic w.r.t. sampling S but with the right optimisation procedure (stochastic gradient descent with decaying learning rates) convergence can still be achieved.

Implementation This concept is implemented via the `DataModuleFromNPZ` in `run.py` which uses data batches of size `size_batch`, see

```
dm = DataModuleFromNPZ(
    data_dir="data_logistic_regression_2d",
    feature_labels=["inputs", "targets"],
    batch_size=64,
    num_workers=4,
    shuffle_training=False
)
```

and the code line

```
logp_expct = self.size_data*torch.mean(p_labels.log_prob(labels.repeat((1,self.n_samples_mc))))
```

where we take the batch average of the expected log likelihood and rescale it with the size of the training data set.

Stochastic gradient learning

The expectations $\mathbb{E}_{q_\phi(w)}[\log p(y_i|x_s, w)]$ can rarely be computed exactly or approximated accurately via quadrature methods---quadrature work well only in 1D. For this reason we often use Monte-Carlo estimates

$$\mathbb{E}_{q_\phi(w)}[\log p(y_i|x_s, w)] \approx \frac{1}{R} \sum_{w^{(r)} \sim q(w)} \log p(y_i|x_s, w^{(r)}), \tag{13}$$

that is, we sample R samples $w^{(r)} \sim q(w)$ and average. This again makes the objective stochastic but we hope that with the right number of samples and the right optimisation procedure the optimisation can stlil converge (see above).

Implementation This is implemented in line

```
logp_expct = self.size_data*torch.mean(p_labels.log_prob(labels.repeat((1, self.n_samples_mc))))
```

In our implementation we so something a cleverer detailed in the next two sections.

Reparameterisation of stochastic variables

If a random variable can be represented as a deterministic (preferably differentiable) function of some other/basic random variable with fixed or no parameters, say,

$$z = f_\theta(\epsilon), \quad \epsilon \sim p_0(\epsilon), \qquad p_\theta(z) = \int d\epsilon\, p_0(\epsilon)\, \delta(z - f_\theta(\epsilon)) \tag{14}$$

then we can rewrite expectations w.r.t this base distribution and make the source of stochasticity in p_θ independent of the parameters

$$E_{p(z)}[g(z)] = E_{p_0(\epsilon)}[g(f_\theta(\epsilon))] \approx \frac{1}{R} \sum_{\epsilon^{(r)} \sim p_0(\epsilon)} g(f_\theta(\epsilon^{(r)}). \tag{15}$$

This makes the MC estimates of expectations easily computable and differentiable w.r.t. θ, that is

$$\partial_\theta E_{p(z)}[g(z)] = E_{p_0(\epsilon)}[\partial g(f_\theta(\epsilon)) \partial_\theta f_\theta(\epsilon)] \approx \frac{1}{R} \sum_{\epsilon_r \sim p_0(\epsilon)} \partial g(f_\theta(\epsilon)) \partial_\theta f_\theta(\epsilon). \tag{16}$$

In our model we have $w = \mu + L\epsilon, LL^T = \Sigma, \epsilon \sim \mathcal{N}(0, I)$. Hence we can use Monte-Carlo samples from ϵ to approximate the objective and easily differentiate the approximation.

Implementation This is implemented in lines

```
# reparameterisation of stochastic variables
L      = self.weights_chol()
p_post = MultivariateNormal(loc=self.weights_loc.squeeze(), scale_tril=L)
```

via the helper function

```
def weights_chol(self):
    return torch.tril(self.weights_scale_lower, -1) + torch.diag(torch.exp(self.weights_scale_logdiag))
```

and the parameterisation is defined the `__init__` function in lines

```
self.weights_loc           = nn.Parameter(torch.zeros((self.dim,1)), requires_grad=True)
self.weights_scale_logdiag = nn.Parameter(torch.zeros((self.dim)), requires_grad=True)
self.weights_scale_lower   = nn.Parameter(torch.zeros((self.dim, self.dim)), requires_grad=True)
```

Local reparameterisation

We observe that the the likelihood terms $\mathrm{Bernoulli}(y_i; x_i w)$ depend only on $x_i w$ hence instead to sampling from $w \sim q(w) = \mathcal{N}(\mu, \Sigma)$ we can sample from $z_i = x_i w \sim= \mathcal{N}(x_i \mu, x_i \Sigma x_i^T)$, that is

$$\begin{aligned}
\mathbb{E}_{q_\phi(w)}[\log \mathrm{Bernoulli}(y_i; x_i w)] &= \mathbb{E}_{w \sim \mathcal{N}(\mu,\Sigma)}[\log \mathrm{Bernoulli}(y_i; x_i w)] && (17)\\
&= \mathbb{E}_{z_i \sim \mathcal{N}(x_i\mu, x_i \Sigma x_i^T)}[\log \mathrm{Bernoulli}(y_i; z_i)] && (18)\\
&\approx \frac{1}{R} \sum_{z_i^{(r)} \sim \mathcal{N}(x_i\mu, x_i \Sigma x_i^T)} \log \mathrm{Bernoulli}(y_i; z_i^{(r)}) && (19)
\end{aligned}$$

thus significantly reducing the variance of the stochastic approximation of the objective.

Implementation This is implemented in lines

```
# local reparameterisation and MCsampling
z_loc     = torch.matmul(features, self.weights_loc).squeeze()
z_scale   = torch.sqrt(torch.sum(torch.matmul(features, L)**2, dim=-1, keepdim=True)).squeeze()
z_samples = Normal(loc=z_loc, scale=z_scale).rsample([self.n_samples_mc]).transpose(0,1)

# data distribution via MC samples
p_labels   = Bernoulli(logits=z_samples)
# computing the MC samples based expected log likelihood with batch learning correction
logp_expct = self.size_data*torch.mean(p_labels.log_prob(labels.repeat((1, self.n_samples_mc))))
```

Questions and tasks (at home)

For the logistic regression model detailed above

- run the code with `python run.py`and check metrics with `tensorboard --logdir lightning_logs`
- change `batch_size`, `n_samples_mc`, `max_epochs`, what do you notice?
- try to implement the diagonal version of `class ModelLogisicRegressionMvn(LightningModule)`, what changes do you have to make?
- compare for $q_\phi(w; Y, X) = \mathcal{N}(w; \mu, \Sigma)$ and $q_\phi(w; Y, X) = \mathcal{N}(w; \mu, \mathrm{diag}(\sigma^2))$
    - what are the differences in terms of storage and computational complexity
    - compare the predictive results on a test set,
    - plot data and predictive class probability
    - plot the two distributions as functions of w and compare to $p(w|Y, X)$, what can we learn?

Unsupervised handwritten digit recognition /variational auto-encoder

Variational auto-encoders are unsupervised models that learn to embed and generate new data similar to one in a, i.i.d. dataset $X = [x_1, \ldots, x_n]^T$. They are Bayesian models where the distribution of the data is

$$p_\theta(X) = \prod_{i=1}^{n} p_\theta(x_i|z_i)\, p(z_i). \tag{20}$$

where generally we have

$$p_\theta(x_i|z_i) = \mathcal{N}(x_i; \mathrm{NN}^{\mathrm{dec}}_{\theta_\mu}(z_i), \theta_{\sigma^2} I_d) \quad \text{and} \quad p(z_i) = \mathcal{N}(0, I_d). \tag{21}$$

Training is done via maximum likelihood using variational Bayes with a posterior approximation

$$q_\phi(z_i; x) = \mathcal{N}(z_i; \mathrm{NN}^{\mathrm{enc-mean}}_{\phi_\mu}(x_i), \mathrm{diag}(\mathrm{NN}^{\mathrm{enc-var}}_{\theta_\sigma}(x_i)) \tag{22}$$

.Here NN denotes a neural network.

The function to optimise is the negative evidence lower bound

$$L(\theta, \phi; X) = -\sum_i E_{q_\phi(z_i;x_i)}[\log p_\theta(x_i|x_i)] + \mathrm{KL}[q_\phi(z_i;x_i)\|p(z_i)]. \tag{23}$$

Implementation The parameters of the distributions are implemented via a pair of neural networks. The decoder implements $p_\theta(x_i|x_i)$ while the encoder implements $q_\phi(z_i;x_i)$.

```
class Encoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        ...
        self.model  = nn.Sequential(...)
    def forward(self, data):
        input  = data
        output = self.model(input)
        loc, scale_isp = torch.split(output, [self.d_state, self.d_state], dim=-1)
        return loc, STD_MIN + torch.nn.functional.softplus(scale_isp)

 class Decoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        ...
        self.loc  = nn.Sequential(...)
        self.scale_isp = nn.Parameter(0.5*torch.ones(1), requires_grad=True)

    def forward(self, input:
        loc   = self.loc(input)
        scale = STD_MIN + torch.nn.functional.softplus(self.scale_isp) * torch.ones(loc.shape, device=loc.device)
        return loc, scale
```

Key variational inference concepts to learn from this model

The loss function is implemented in

```
def loss(self, imgs):
```

Maximum likelihood via expectation maximisation

In this model we not only have to approximate $p_\theta(z_i|x_i)$ but we also need to maximise the lower bound with respext to the model parameters θ., that is we jointly maximise the w.r.t. the model parameters and the posterior approximation $q_\phi(z_i;x_i)$.

Implementation joint learning is implemented via a single optimiser

```
def configure_optimizers(self):
    opt = torch.optim.Adam(itertools.chain(self.encoder.parameters(),
                                           self.decoder.parameters()),
                               lr=self.hparams.lr, betas=(self.hparams.b1, self.hparams.b2))
    return opt
```

Amortised variational inference

If we would proceed according to the logistic regression model we would have to approximate each $p_\theta(z_i|x_i)$ in a separate inner loop for each new θ value. Instead we learn $q_\phi(z_i;x_i) \approx p_\theta(z_i|x_i)$ thus replacing the variational inference algorithm with learning the parameter mappings $\mathrm{NN}^{\mathrm{enc-mean}}_{\phi_\mu}(x_i)$ and $\mathrm{NN}^{\mathrm{enc-var}}_{\theta_\mu}(x_i)$.

```
q_zx = torch.distributions.normal.Normal(*self.encoder(imgs.view([size_batch] + self.encoder.shape_input)))
z_samples = q_zx.rsample((self.hparams.n_samples,))
p_xz = torch.distributions.normal.Normal(*self.decoder(z_samples.view([self.hparams.n_samples, size_batch] + self.decoder.shape_input)))
lik_eval = p_xz.log_prob(imgs.view([size_batch] + self.encoder.shape_input))
logp = torch.sum(torch.mean(torch.mean(lik_eval, dim=0, keepdim=True), dim=1, keepdim=True))
```

Optional questions and tasks

- run the code with `python run.py --config config_vae.yaml` and check metrics with `tensorboard --logdir lightning_logs`
- if you have time, read the original paper https://arxiv.org/pdf/1312.6114.pdf
- try to answer the questions

- how is this model different from the logistic regression one
    - in terms of latent variables?
    - in terms of likelihood model?
- what are the differences in term of approximate inference
    - in term of latent variables?
    - in terms of parameterisation, what does amortisation mean?
    - can we use amortisation for the logistic regression model?