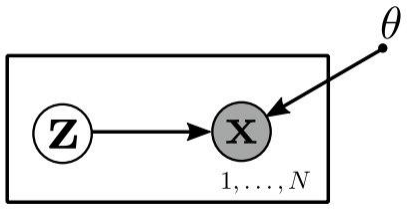
Auto-encoding Variational Bayes

Applied Bayesian Methods

Luis Antonio Ortega Andrés
Juan Ignacio Álvarez Trejos

May 12, 2021

The main objective of this assignment is to develop a variational autoencoder using the working principles of variational inference, and, use this method for doing approximate inference in the MNIST dataset. The probabilistic model considered is a generative model of the form:



Task 1. *Complete the missing parts of the code to allow for training a VAE on the MNIST dataset.*

We are briefly reviewing the implemented steps for each of the four functions designed to this task. Let us begin with `sample_latent_variables_posterior`, the main aim of this function is to generate an unique sample from $Q(z \mid x)$ for each x in the given batch. To this end, the mean and variance of each factor Q is given as input for the function. These values can be retieved form the first and last half over the last axis. In order to return the desired samples we are using the decomposition property of the Gaussian distribution, that is

$$\mathcal{N}(\mu, \sigma^2) = \mu + \sigma\mathcal{N}(0, 1).$$

The resulting code is

```
mean, log_std = encoder_output[:, :D], encoder_output[:, D:]
z = npr.randn(*mean.shape)
return mean + z * np.exp(log_std)
```

The `bernoulli_log_prob` function aims to return the log-likelihood of the actual data given the computed logits using the output of the neural netork, that is, to compute

$$\begin{aligned}\log P(\boldsymbol{x} \mid \boldsymbol{z}) &= \sum_{x \in \boldsymbol{x}} \log\Big(p(x)^x + (1 - p(x))^{(1-x)}\Big) \\ &= \sum_{x \in \boldsymbol{x}} \log\Big(x\, p(x) + (1 - x)\, (1 - p(x))\Big).\end{aligned}$$

To this end, it is needed to compute the probabilities fro the given logits using the sigmoid function:

```
    probs = sigmoid(logits)
    log_prob = np.log(targets * probs + (1 - targets) * (1 - probs))
    return np.sum(log_prob, axis=-1)
```

On the other hand, function `compute_KL` is used to compute the Kullback-Leibler divergence between the variational distribution $Q(\boldsymbol{z} \mid \boldsymbol{x})$ and the prior distirbution $\mathcal{N}(0, I)$. Given that both distributions are Gaussian, their KL divergence has a closed form which is in turn simplified given the form of the prior. More precisely,

$$\mathcal{KL}\Big(\mathcal{N}\big((\mu_1, \ldots, \mu_k)^T, \text{diag}(\sigma_1^2, \ldots, \sigma_k^2)\big) \mid \mathcal{N}(\mathbf{0}, \boldsymbol{I})\Big) = \frac{1}{2} \sum_{i=1}^{k} \left(\sigma_i^2 + \mu_i^2 - 1 - \log \sigma_i^2\right).$$

This value can be easily computed given `mean` and `log_stds` from the neural network output as

```
KL = 0.5 * (np.exp(2 * log_std) + mean ** 2 - 1 - 2 * log_std)
return np.sum(KL, axis = -1)
```

Lastly, `vae_lower_bound` is used to compute a noisy estimate of the lower bound by using a single Monte Carlo sample. That is, to approximate the expectation in the objective function with a single observation of the variable. The steps are the following:

1. Compute the encoder output using `neural_net_predict`:

   ```
   output = neural_net_predict(params=rec_params, inputs=data)
   ```

2. Sample the latent variables associated to the batch in data:

   ```
   latents = sample_latent_variables_from_posterior(output)
   ```

3. Reconstruct the image and to compute the log likelihood of the actual data:

   ```
   x_samples = neural_net_predict(gen_params, latents)
   log_prob = bernoulli_log_prob(data, x_samples)
   ```

4. Compute the KL divergence between $Q(\boldsymbol{z} \mid \boldsymbol{x})$ and the prior:

   ```
   KL = compute_KL(output)
   ```

5. Estimate the lower bound by substracting the KL to the data dependent term

   ```
   return np.mean(log_prob - KL, axis=-1)
   ```

Task 2. *Complete the initialization of the ADAM parameters and write the ADAM updates in the main training loop of the code provided in* `vae.py`.

Using the provided description of the algorithm, the parameter updates can be computed as the following:

```
m = beta1 * m + (1 - beta1) * grad
v = beta2 * v + (1 - beta2) * grad ** 2
m_unbiased = m / (1 - beta1 ** t)
v_unbiased = v / (1 - beta2 ** t)

flattened_current_params += (
      alpha * m_unbiased / (np.sqrt(v_unbiased) + epsilon)
)
```

With initial values given by

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10**-8
m = np.zeros_like(flattened_current_params)
v = np.zeros_like(flattened_current_params)
```