Variational Autoencoder

Métodos Bayesianos

Adrián Rubio & Marcos Vázquez

May 17, 2022

1 Tarea 1

1.1 sample_latent_variables_from_posterior()

Utilizando el truco de la reparametrización se genera un $\mathbf{z_i}$ con probabilidad $q_\phi(z|x)$. De esta forma se puede separar la aleatoriedad de $\mathbf{z_i}$ añadiendo el término $\epsilon_i^j \sim \mathcal{N}(0,1)$. Para generar z se utiliza la siguiente línea de código:

```
z=np.exp(log_std)*npr.randn(log_std.shape)+mean
```

Donde las variables son vectores de la misma longitud para evitar que iterar sobre cada elemento. Como la salida de la red neuroanl nos da el logaritmo de la desviación típica, le aplicamos una exponencial para recuperarlo. Mediante npr.randn generamos el ruido aleatorio.

1.2 bernoulli_log_prob()

Tras haber generado los z_i por el truco de la reparametrización,queremos calcular el log de la distribución $p_\theta(x|z)$, la cual es la distribución que genera las imágenes dada la información de alto nivel z. Dicha distribución viene dada por un el producto de D distribuciones de variables aleatorias de Bernouilli. Dado que calculamos el logaritmo, sumamos términos en vez de multiplicarlos.

Se nos proporcionan los logits, que son la salidas de la red neuronal de parámetros θ. Para obtener la probabilidad de que los píxeles valgan 0 o 1, le aplicamos la sigmoide σ. Así con $\sigma(logit)$ y $(1-\sigma(logit))$ obtenemos la probabilidad de que un píxel valga 0 o 1.

Utilizando estimación por máxima verosimilitud de p_θ se puede hacer siguiendo la fórmula (11) del enunciado. En este apartado nos piden calcular el primer término. Tenemos que calcular la esperanza de $log(p_\theta(x|z))$. Al ser multiplicaciones de las dimensiones de la entrada lo que hay dentro del logaritmo se puede descomponer en sumas de logaritmos:

En nuestro código lo hemos implementado tal que así:

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

Siendo targets el vector de datos x.

1.3 compute_KL()

El segundo término de función objetivo(aproximada) que queremos calcular es la divergencia de Kullback-Leibler entre la aproximación de la posteriori dada por $q_\phi(z|x)$ y la priori $p(z)$. Dado que hemos comentado que ambas siguen una distribución Gaussiana(el prior es una Gaussiana estándar), dicha expresión tiene una solución analítica(Fórmula 12 del enunciado). La implemtentamos tal que así:

```
kl=np.sum((np.exp(log_std*2)+mean**2-1-2*log_std)/2,axis=-1)
```

Dado que la red neuronal nos devuelva la log_std, por las propiedades de los logaritmos, multiplicamos por 2 la log desviación y le aplicamos la función exponente para obtener la varianza.

1.4 vae_lower_bound()

Ahora ya podemos calcular la cota inferior de la aproximación ruidosa de nuestra función objetivo usando un solo sample de la simulación de Monte Carlo(fórmula 14 del enunciado). Para encontrar este límite dado un ejemplo solo tenemos que introducirlo en el codificador que devuelve la media y la varianza. Luego se genera las variables del espacio latente a partir de la función creada en la Tarea 1.1. Por último, se calcula la verosimilitud con la función de pérdida de bernoulli (Tarea 1.2.) y la divergencia de Kullback-Leibler (Tarea 1.3.). Se pondera el término de la verosimilitud con el número de datos entre el número de datos por batch ($\frac{N}{|B|}$), dado que estamos haciendo una aproximación ruidosa. La implementación en el código se ha hecho en los 5 pasos:

```
encoder_output=neural_net_predict(rec_params,data)

sample_latent_variables_from_posterior_=sample_latent_variables_
from_posterior(encoder_output)

gen=bernoulli_log_prob(data,neural_net_predict(gen_params,sample_
latent_variables_from_posterior_))

kl=compute_KL(encoder_output)

returnN/batch_size*np.sum(gen-kl,axis=-1)
```

2 Tarea 2

Usando la descripción del algoritmo ADAM proporcionado, primero le damos el siguiente valor inicial a los parámetros que se nos recomienda:

```
alpha=0.001 beta1=0.9 beta2=0.999 epsilon=10**(-8)
```

Con la siguiente actualización de parámetros:

```
m=beta1*m+(1-beta1)*grad

v=beta2*v+(1-beta2)*grad**2

m_est=m/(1-beta1**t)

v_est=v/(1-beta2**t)

lattened_current_params+=(alpha*m_est/(np.sqrt(v_est)+epsilon))
```

Con la actualización de lattened_current_params(θ_t) sumando en vez de restar, para convertir el problema de minimización del algoritmo en uno de maximización.

3 Tarea 3

3.1 Tarea 3.1

En esta prueba se pedía que se generasen aleatoriamente datos utilizando variables del espacio latentes aleatorias como entradas del decodificador. Se puede ver que los resultados parecen razonables y casi todos ellos se puede reconocer como un número. Aunque algunos tienen líneas más flojas que convierten los números generados en otros. Para generar los resutlados se ha utilizado el código:

```
save_images(sigmoid(neural_net_predict(gen_params,npr.randn(25,
                    latent_dim))),"Task_3_1")
```

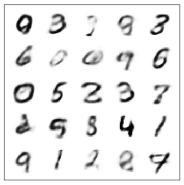

Figure 1: Tarea 3.1: Generar 25 números aleatorios

3.2 Tarea 3.2

En esta prueba se utiliza toda la red decodificando las primeras 10 imágenes de test para volverlas a generar. Se puede ver los números de la parte superior son la entrada y los de la mitad inferior son los generados por el decodificador.El código utilizado:

```
post=neural_net_predict(rec_params,test_images[:10])

sample_latent_variables_from_posterior_=sample_latent_variables_
                    from_posterior(post)

gen=neural_net_predict(gen_params,sample_latent_variables_from_
                        posterior_)

save_images(np.r_[test_images[:10],sigmoid(gen)],"Task_3_2")
```

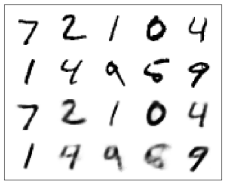

Figure 2: Tarea 3.2: Test de Codificación-Decodificación

3.3 Tarea 3.3

Esta última tarea se basa en generar números a partir de las medias de dos imágenes dando más importancia a la segunda imagen según avanzan los números generados. Se puede ver como parece que no hay una mezcla rara de ambos números, más bien se pasa de uno a otro o a veces a un número intermedio. Esto es debido a que estamos en el espacio latente y al coger 2 puntos(2 imágenes), al trazar una linea entre ellos, vemos la representación latente del modelo, esto son las imágenes intermedias. Así podemos ver que este tipo de redes es muy estable en la generación de datos categóricos. Aunque puede que para datos continuos no sea la mejor solución.

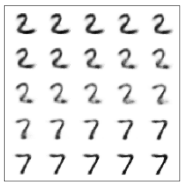

Figure 3: Tarea 3.3: Interpolación 2 a 7

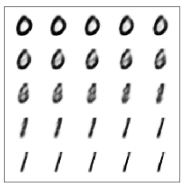

Figure 4: Tarea 3.3: Interpolación 0 a 1

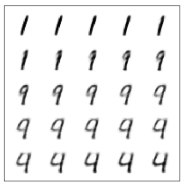

Figure 5: Tarea 3.3: Interpolación 1 a 4

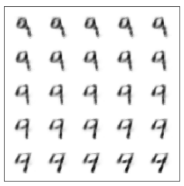

Figure 6: Tarea 3.3: Interpolación 9 a 4

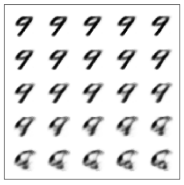

Figure 7: Tarea 3.3: Interpolación 9 a 5