Autoencoders

Índice

Referencias

Capítulo 15 de *Hands-On Machine Learning with Scikit-Learn and TensorFlow*

Tiempo estimado = 1 sesiones de 2 horas (1 semana)

1. Motivación

Hasta ahora sólo hemos estudiado como aplicar redes neuronales, profundas o no, a problemas de aprendizaje supervisado. La principal dificultad para lograr buenos resultados con redes profundas en tareas supervisadas es encontrar un conjunto de entrenamiento suficientemente grande y diverso, que además debe estar etiquetado. Como la tarea de etiquetado requiere de un humano, habitualmente se recurre a "turcos mecánicos", es decir a la participación de mucha gente, realizando tareas muy sencillas, por las que reciben una muy pequeña cantidad de dinero. Un ejemplo muy conocido es el Amazon Mechanical Turk.

Alternativamente, podríamos intentar aprender a separar los ejemplos en grupos, sin etiquetarlos previamente. Esto elimina el problema de tener el conjunto de datos etiquetado, pero por otro lado no nos asocia un ejemplo a una etiqueta, sino a un grupo.

Si aplicamos esta intuición a redes neuronales, nos encontraríamos ante el escenario mostrado en la Figura 1. Como hemos aprendido, para aprender los pesos de la red neuronal debemos retropropagar el error, o el valor calculado por la función de pérdida. Pero, si no tenemos etiquetas ¿qué es lo que podemos comparar?

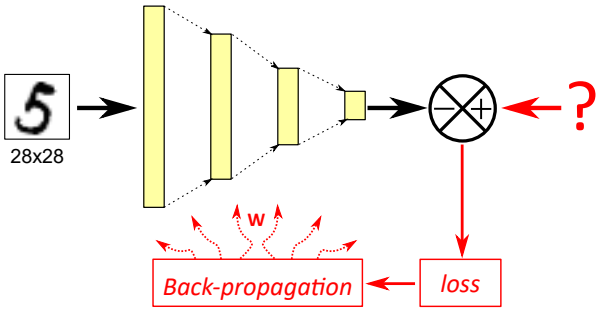


Figura 1: Si no tenemos etiquetas, ¿cómo podemos entrenar la red?

Lo único que tenemos para comparar son los propios ejemplos. Es decir, la idea es comparar el resultado que produce la red neuronal ante un ejemplo y el propio ejemplo, y ajustar los pesos para que cada vez sea más parecido. Esta es la premisa con la que se construyen los Autoencoders.

2. Arquitectura de un autoencoder

Un autoencoder es una arquitectura de red neural que tiene el objetivo de producir una salida lo más parecida posible al ejemplo de entrada.

Esto significa que el vector de salida tendrá las mismas dimensiones que el vector de entrada. Además, el rango de cada elemento del vector de salida debe ser el mismo que el rango de su elemento correspondiente en el vector de entrada. La salida, por tanto, no es una categoría, clase o etiqueta. Sin embargo conocemos el valor objetivo que deseamos alcanzar para cada elemento del vector de salida, por tanto podemos atacar el problema como uno de **regresión**, a pesar de que se trata de un problema no-supervisado visto en global.

Con esta premisa podemos plantear dos tipos de autoenconders: neuronal y convolucional, dependiendo del tipo de neuronas con el que están construidos.

2.1. Autoencoder neuronal

Como hemos visto, en una red neuronal habitualmente las capas ocultas van disminuyendo de tamaño hasta llegar a la capa de salida, donde hay tantas neuronas como clases diferentes. Pero si la red tiene cada vez menos neuronas en cada capa, tras varias capas podría ocurrir que el número de neuronas de salida sea menor que el número de características en un ejemplo.

Cuando los ejemplos son imágenes, y consideramos cada píxel como una característica, es muy probable que partamos de un número muy grande de neuronas de entrada, por lo que, a poco que reduzcamos en cada capa oculta, acabaremos con menos neuronas de salida que de entrada.

Siguiendo esta lógica, necesitaríamos tantas neuronas de salida como píxeles hay en la imagen dada. Si la red neuronal tiene la forma de la Figura 1, necesariamente tenemos que añadir capas de tal manera que cada capa oculta tenga más neuronas que la anterior. La idea esquemática está representada en la Figura 2.

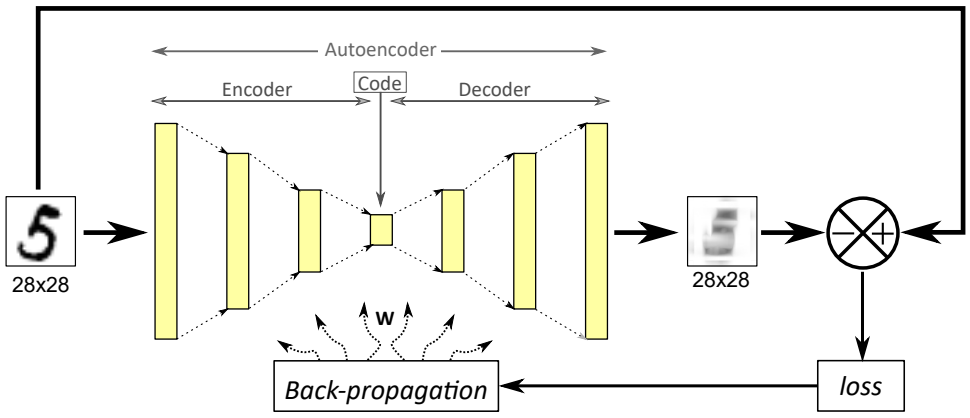


Figura 2: Arquitectura *básica* de un autoencoder y su proceso de entrenamiento.

Normalmente el autoencoder tiene forma de reloj de arena en horizontal, y es simétrico respecto a la capa oculta central, llamada a veces *cuello de botella*.

En su versión más simple habría una única capa oculta con N neuronas, como en la Figura 3, y cada una de ellas con activación lineal. El efecto de este autoencoder es similar al de obtener los N componentes principales; es decir obtenemos una representación de cada ejemplo con menos dimensiones que el espacio en el que viven. La fase *Encoder* es la encargada de reducir la dimensionalidad, mientras que la fase *Decoder* es la encargada de reconstruirla, tal y como hemos explicado.

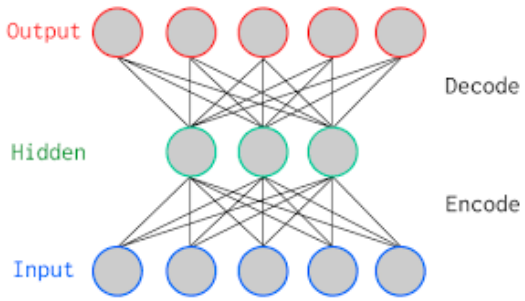


Figura 3: Autoencoder *mínimo*, pero equivalente a los 3 componentes principales. (Fuente: Wikipedia)

Añadir capas al *Encoder* y al *Decoder* modifica el modo de codificar y reconstruir el código, de manera que podrían obtenerse resultados con menos pérdidas para un mismo tamaño de código.

2.2. Autoencoder convolucional

El autoenconder convolucional tienen algunas características que lo hacen considerablemente diferente del autoencoder neuronal.

En el Tema 2 se explicaba cómo se construye una red convolucional en la que, mediante las capas de agrupación (*MaxPool*), se reduce el tamaño de la imagen según esta atraviesa la etapa convolucional. Al terminar la etapa convolucional normalmente tenemos un gran numero de imágenes, de muy baja resolución comparada con la imagen inicial. Estas imágenes se serializan con la función *flatten* para alimentar a una red FCNN donde se realiza la tarea de clasificación. En este proceso, hay dos asuntos clave:

1. Cuando hacemos $MaxPool(n\times n)$ pasamos de tener una vecindad de n^2 píxeles a 1. Por tanto hemos perdido n^2-1 píxeles con una función no invertible, ¿cómo podemos recuperarlos?
2. Es muy probable que el número de neuronas de entrada a la FCNN sea **mayor que** el número de píxeles de la imagen de entrada. Por ejemplo, en el MNIST, las imágenes de entrada son $28\times 28 = 784$ píxeles. Si el resultado de la etapa convolucional fueran 64 imágenes 4×4, tras serializarlas necesitaríamos 1024 neuronas de entrada, un 30 % más. Con esto queremos decir que, muy frecuentemente, la representación de los ejemplos obtenida con una CNN está en un espacio de más dimensiones que el original.

A continuación vamos a tratar cada uno de estos puntos para llegar a la arquitectura de un autoencoder convolucional.

UpSampling. Imaginemos un autoencoder convolucional, en el que colocamos dos etapas convolucionales enfrentadas en el cuello de botella, del mismo modo que el autoencoder neuronal, como se muestra en la Figura 4. Como se puede ver, pasado el cuello de botella, más pronto que tarde nos encontramos con una contradicción: intentamos aumentar el tamaño de la imagen atravesando una capa *MaxPool*, ¡cuya función es reducir!.

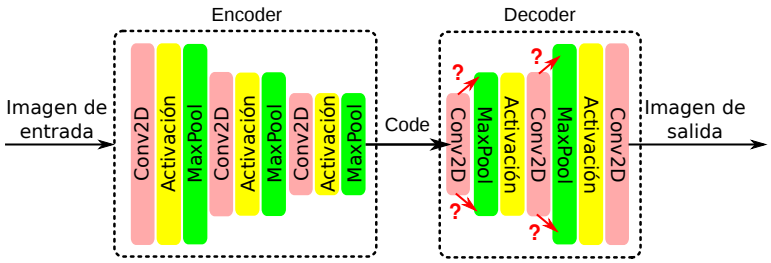


Figura 4: ¿Tiene sentido hacer *MaxPool* cuando queremos aumentar la resolución, no disminuirla?

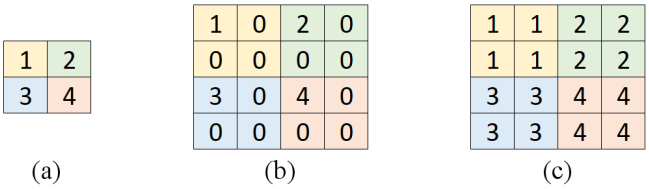


Figura 5: Dos maneras de hacer $UpSampling(2\times 2)$ a la "imagen" 2×2 de (a). En (b) cada pixel se copia 1 vez y el resto se rellena de 0. En (c) cada pixel se copia 2×2 veces.

En realidad, el decodificador no tiene capas *MaxPool*, sino otras en las que, a partir de un píxel, se obtiene una vecindad. Esta operación se denomina *UpSampling*. Por tanto, además del modo

en el que se rellenan los píxeles nuevos, en la función *UpSampling* se debe especificar el tamaño de la vecindad resultante.

A fin de cuentas, estamos interpolando nuevos píxeles a partir de uno, y esta operación se puede hacer de varias maneras: mediante vecinos cercanos, bilineal, bicúbica,...; en la Figura 5 muestran dos posibilidades. Entonces, ¿hay alguna manera óptima/entrenable de realizar el *UpSampling*?

Convolución transpuesta. Como vimos al estudiar la operación de convolución, según sea el *padding*, la imagen resultante puede tener una resolución menor. Esto significa que también podríamos construir codificadores sin utilizar capas *MaxPool*. ¿Sería también posible construir decodificadores sin utilizar *UpSampling*? Para ello deberíamos ser capaces de reconstruir la resolución de una imagen a partir de otra obtenida previamente mediante convolución. Si, además esta operación fuera parametrizable, por ejemplo depende de un filtro como la convolución, entonces además sería posible aprender dichos parámetros, es decir encontrar mediante entrenamiento la mejor manera de reconstruir una resolución a partir de otra menor. Inicialmente, esta operación recibió el nombre de "Deconvolución", pero actualmente se prefiere decir "Convolución transpuesta" porque la deconvolución es algo distinto a lo que realmente se realiza, como vamos a ver a continuación.

La convolución de una imagen con un filtro se puede reescribir como una multiplicación matricial en la que el filtro de convolución se transforma en una matriz de convolución, y tanto la imagen original como la resultante se serializan en un vector. En la Figura 6 se muestra un ejemplo para una imagen; y en este caso la resolución resultante es menor.

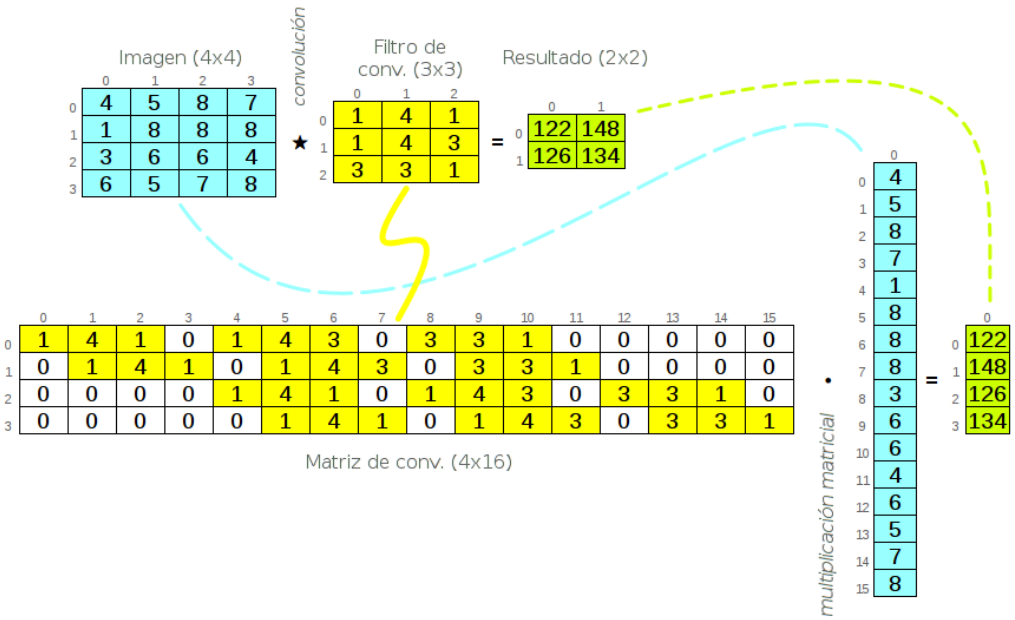
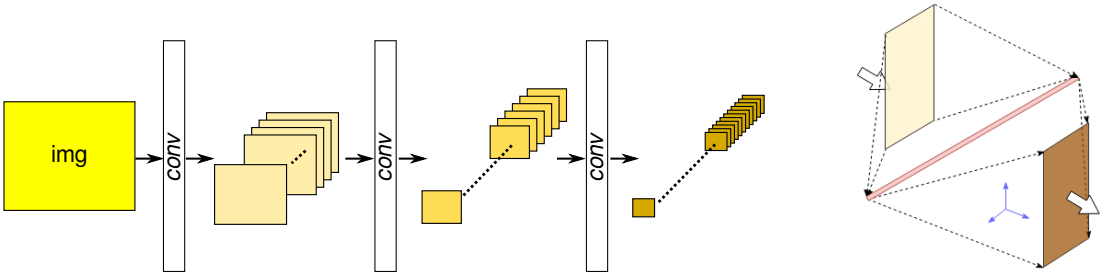


Figura 6: La operación de convolución de una imagen con un filtro, expresada como la multiplicación de una matriz de convolución por un vector.

Si queremos invertir el efecto y aumentar la resolución necesitamos una nueva matriz de convolución cuyas dimensiones son las transpuestas de la anterior. Siguiendo con el ejemplo anterior, si ahora nuestro vector de entrada es (4×1) y nuestro vector de salida es (16×1) entonces la matriz de convolución debe ser (16×4). Además, esta matriz de convolución, de dimensiones transpuestas a la anterior, se puede aprender igual que se aprendía la otra.

En la literatura se han propuesto diferentes maneras de realizar la convolución transpuesta. En V. Dumoulin & F. Visin (2018) (GitHub) (PDF) hay una recopilación bastante completa.

¿Cuello de botella? El segundo asunto importante es que un autoencoder convolucional tiene una forma tridimensional. Con esto queremos decir que, aunque en el codificador la resolución de salida es menor que la de entrada, el código no es una única imagen, sino un banco de ellas. Es decir que para representar todas las imágenes que se producen en el cuello de botella necesitaríamos una tercera dimensión, como se puede ver en la Figura 7.



(a) (b)

Figura 7: (a) Representación del resultado habitual al atravesar varias capas convolucionales. (b) Representación de la forma de una autoencoder convolucional.

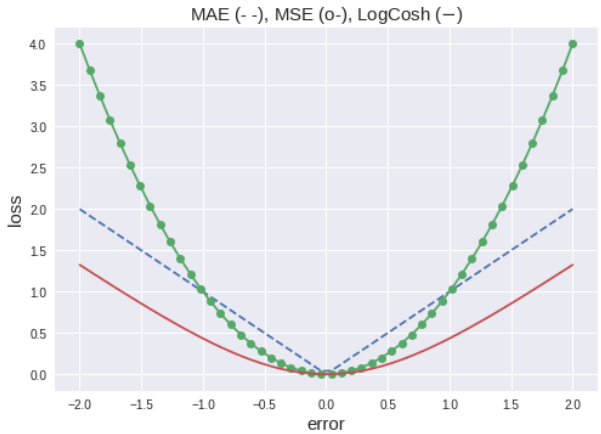
2.3. Funciones de activación y de pérdida.

Cualquiera de las dos arquitecturas anteriores, y también cualquiera de las variantes que se explicarán a continuación, necesitan de una función de pérdida para poder tener un error que retropropagar. Además, hay que elegir una función de activación para las neuronas.

Pérdida. Puesto que el autoencoder trata de reconstruir el ejemplo dado, y la salida se compara con él, debemos tratar el problema como una regresión múltiple, de tantas funciones como atributos tenga cada ejemplo; en el caso de una imagen, tantas como píxeles haya.

Las funciones de pérdida más apropiadas y populares para regresión son:

- Error cuadrático medio, también conocido por pérdida cuadrática, pérdida L_2 o MSE.
- Error absoluto medio, también conocido por pérdida L_1 o MAE.
- Pérdida Log-Cosh, que es una versión suave del Error absoluto medio.



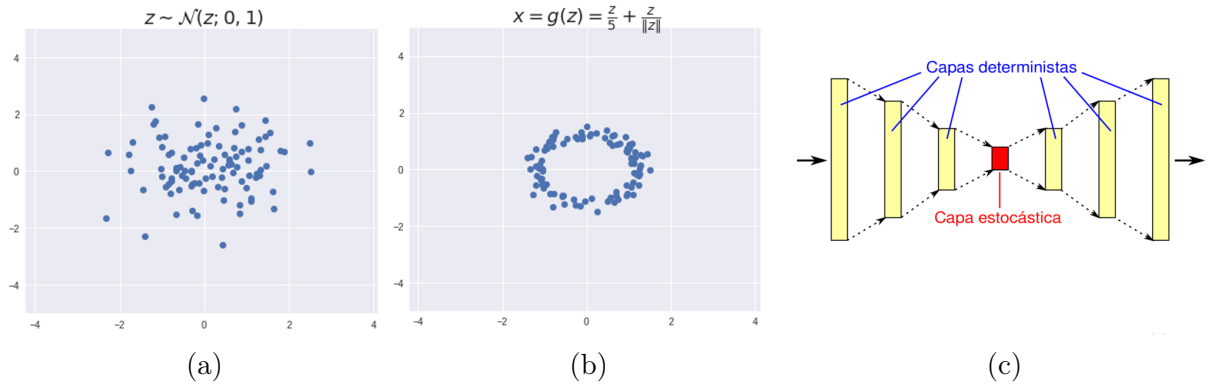
Además, para el caso concreto de visión artificial, se podrían utilizar medidas de comparación de imágenes, de su densidad espectral, etc.

Activación. Puesto que, en definitiva, el problema se aborda como una regresión, la activación de las neuronas de la última capa debería ser lineal. En el resto de neuronas, y ya que la profundidad de un autoencoder es siempre el doble que el de una red neuronal o convolucional, es recomendable utilizar activaciones que eviten el desvanecimiento del gradiente como ReLU.

3. Autoencoder Variacional

Sea una variable aleatoria $\mathbf{z} \sim p(\mathbf{z};\theta)$. Mediante una transformación determinista $\mathbf{x} = g(\mathbf{z})$ es posible crear otra variable aleatoria, con otra distribución distinta, $\mathbf{x} \sim p(\mathbf{x};\phi)$. En la Figura 8(a-b) se muestra un ejemplo donde $\mathbf{z} \sim \mathcal{N}(\mathbf{z};\mu = 0, \sigma = 1)$ y $g(z) = \frac{z}{5} + \frac{z}{\|z\|}$.

Como sabemos, una red neuronal es un aproximador universal. Es decir, se comporta como una función determinista capaz de *imitar* cualquier otra función o mapeo. De manera que podríamos sustituir la función $g(\cdot)$ por una red neuronal y, en conjunto, tendríamos un modo de construir distribuciones arbitrarias, según la red conectada a las muestras $\mathbf{z}$. Por otro lado, podríamos tener una red neuronal que mapeara cada ejemplo a una muestra de la variable aleatoria $\mathbf{z}$. En definitiva, juntando ambas, y haciendo que la salida sea igual a la entrada, tenemos las bases de un autoencoder en el que el cuello de botella es una capa aleatoria o estocástica, representado en la Figura 8(c). A este tipo de autoencoder les llamaremos *variacionales*.



(a) (b) (c)

Figura 8: (a) $\mathbf{z}$ tiene una distribución normal estándar, (b) $\mathbf{x}$ tiene una distribución en anillo construida con una variable aleatoria y una función determinista, (c) capas estocásticas.

En los autoencoders variacionales (*variational autoencoders*, VAE), el codificador no mapea la entrada a un código, sino que lo hace a una distribución de probabilidad; en concreto a la normal multivariada, que viene determinada por dos parámetros: la media y la desviación. De este modo, en el cuello de botella ahora tenemos dos vectores separados, el de la media y el de la desviación, cada uno de ellos con tantos elementos como características queramos que tenga el código.

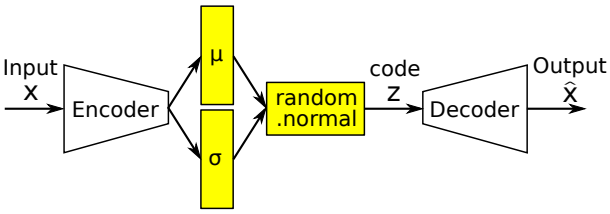


Figura 9: Esquema de un VAE. El código $\mathbf{z}$ es una muestra de la distribución $\mathcal{N}(\mu, \sigma)$

En un autoencoder el código es la entrada al decodificador. Ahora, sin embargo, el código es probabilístico. Por tanto muestreando de la distribución podemos obtener tantos códigos diferentes y aleatorios como queramos; que tras pasar por el decodificador deben dar lugar a la imagen reconstruida. En la Figura 9 se muestra este proceso. Como se puede ver, el código es una muestra de una distribución, por tanto podemos dar la siguiente interpretación probabilística a la red *Encoder* y la red *Decoder* respectivamente:

Encoder: $q(\mathbf{z}|\mathbf{x};\theta)$ es la distribución del código $\mathbf{z}$, dado un ejemplo $\mathbf{x}$, con parámetros θ.

Decoder: $p(\mathbf{x}|\mathbf{z};\phi)$ es la distribución del ejemplo $\mathbf{x}$, dado un código $\mathbf{z}$, con parámetros ϕ.

Este tipo de autoencoders presenta dos novedades.

1. Una función de pérdida diferente.
2. Cómo aplicar retropropagación en el cuello de botella, que es aleatorio.

3.1. Función de pérdida

La función de pérdida del ejemplo i-ésimo tiene dos términos:

$$\ell^{(i)}(\theta, \phi; \mathbf{x}^{(i)}, \mathbf{z}) = -\mathbb{E}_{q(\mathbf{z}|\mathbf{x};\phi)}\left[\log p(\mathbf{x}^{(i)}|\mathbf{z};\theta)\right] + D_{KL}\left(q(\mathbf{z}|\mathbf{x}^{(i)};\phi)\|p_\theta(\mathbf{z})\right)$$

El primer término a la derecha de la igualdad es la pérdida de reconstrucción y se interpreta del siguiente modo:

- El $\log p(\mathbf{x}^{(i)}|\mathbf{z};\theta)$ es una medida de lo bien que reconstruye el decodificador al ejemplo i-ésimo. Cuanto mejor sea la reconstrucción, mayor será su probabilidad en la función de distribución de los ejemplos dados los códigos, y por tanto también mayor será su logaritmo.
- Al utilizar el operador $\mathbb{E}_{q(\mathbf{z}|\mathbf{x};\phi)}[\cdot]$ estamos promediando la medida anterior según la distribución de los códigos. Para hacer este concepto más intuitivo vamos a suponer que sólo hubiera 3 códigos posibles, cada uno de ellos con una probabilidad; como en la medida anterior el código es una variable condicionante tendremos tres posibles medidas. En vez de quedarnos con una de ellas, nos quedamos con el promedio de ellas, ponderando cada una con la probabilidad de su código. Cuanto mayor sea este promedio, mejor es la reconstrucción.
- Añadimos el signo negativo porque estamos construyendo una función de pérdida. Es decir queremos que cuanto mejor sea la reconstrucción peor sea este promedio, ya que después minimizaremos.
- Podemos aproximar este promedio con el MSE entre la imagen reconstruida y la imagen original.

El segundo término es la divergencia Kullback-Leibler (KL) entre la distribución del codificador y una distribución que habitualmente se elige como normal estandar.

- La divergencia KL es una medida de la similitud entre dos distribuciones de probabilidad. Esta medida nunca es negativa y no es simétrica; es decir $KL(f(x)\|g(x)) \neq KL(g(x)\|f(x))$.
- Si hacemos que $q(\mathbf{z}|\mathbf{x}^{(i)})$ sea una distribución normal, cuyos parámetros μ y σ dependan de los datos $\mathbf{x}$, entonces podemos escribir: $q(z|x) = \mathcal{N}(\mathbf{z}; \mu(\mathbf{x}), \sigma(\mathbf{x}))$, donde $\mu(\mathbf{x})$ y $\sigma(\mathbf{x})$ son funciones deterministas, que sólo necesitan los datos.

  Entonces podemos utilizar redes neuronales para construir $\mu(x)$ y $\sigma(x)$. A partir de ellos generaríamos códigos muestreando de la $\mathcal{N}(z; \mu(x), \sigma(x))$.
- Cuanto mayor sea esta divergencia más penaliza la función de pérdida. Por tanto funciona como un término de regularización, *atando* la distribución buscada a la distribución normal estándar.

La justificación de esta función de pérdida se ha añadido como extra en el apéndice.

3.2. El truco de la reparametrización

Cuando pensamos en media y desviación, nos viene a la mente que ambos son el resultado de aplicar una fórmula a un conjunto de datos. Aquí, sin embargo, son valores que genera el codificador.

Estos códigos aleatorios alimentan al decodificador y después la imagen reconstruida se compara con la original. Creando la función de pérdida vista podemos aplicar *back propagation* y descenso del gradiente para actualizar los pesos de las redes (θ, ϕ).

Pero ¿cómo hacemos *back propagation* en la capa estocástica del cuello de botella?

Sabemos que cada código es una muestra $\mathbf{z} \sim \mathcal{N}(\mathbf{z}; \mu(x), \sigma(x))$, por tanto $\mathbf{z} = \mu + \sigma \cdot \epsilon$, con $\epsilon \sim \mathcal{N}(0, 1)$.

Es decir que la capa estocástica es a todos los efectos, una capa determinista.

- *Feed forward*: para muestrear multiplicamos la desviación σ por un *ruido* ϵ y sumamos μ.
- *Back propagation*: para actualizar los pesos nos olvidamos de la existencia de ϵ.

3.3. *Disentangled Variational Autoencoders*

Al hacer en truco de la reparametrización somos capaces de aprender μ como σ , por tanto podemos pensar en unas neuronas especializadas en μ y σ, justo en el cuello de botella

¿Seguro que estas neuronas no están correlacionadas? Si lo están, conocer el comportamiento de una nos daría toda la información de lo que hace otra. Pero lo interesante es que cada una tenga su propio comportamiento.

Podemos ponderar cuanta importancia tiene la D_{KL} en la función de pérdida. De esta manera el autoencoder penalizará la distribución si no aporta beneficios a la función de pérdida. Y eso ocurre si aporta información diferente.

$$\ell^{(i)}(\theta, \phi; \mathbf{x}^{(i)}, \mathbf{z}) = -\mathbb{E}_{q(\mathbf{z}|\mathbf{x};\phi)}\left[\log p(\mathbf{x}^{(i)}|\mathbf{z};\theta)\right] + \beta D_{KL}\left(q(\mathbf{z}|\mathbf{x}^{(i)};\phi)\|p_\theta(\mathbf{z})\right)$$

4. Ejercicios teóricos

P.1: ¿Por qué se suele decir que los autoencoders realizan una tarea no supervisada?

P.2: Razonar que implicaciones tendría un cuello de botella de tamaño unidad, es decir un único elemento, un numero real por ejemplo, en su capacidad de generalizar.

P.3: Razonar cómo utilizar un autoencoder para entrenar un conjunto de datos grande pero con pocas etiquetas.

P.4: Razonar el efecto de tener un cuello de botella con más neuronas que características de entrada.

P.5: Razonar con un autoencoder *de juguete* qué ocurriría si hacemos que los pesos del *encoder* son los pesos del *decoder* traspuestos.

5. Ejercicios prácticos

- Diseñar un autoencoder convolucional tal que su un cuello de botella sea un autoencoder neuronal con 1 capa intermedia.
- Visualizar la representación obtenida en el cuello de botella para un autoencoder variacional utilizando el MNIST.
- Pintar el MNIST en 2 dimensiones utilizando Autoencoder neuronal con diferente número de capas ocultas. En la Figura 10 se muestra un ejemplo.
- Utilizar un autoencoder para hacer clasificación usando sólo una fracción de los datos etiquetados.
- Implementar un autoencoder variacional.

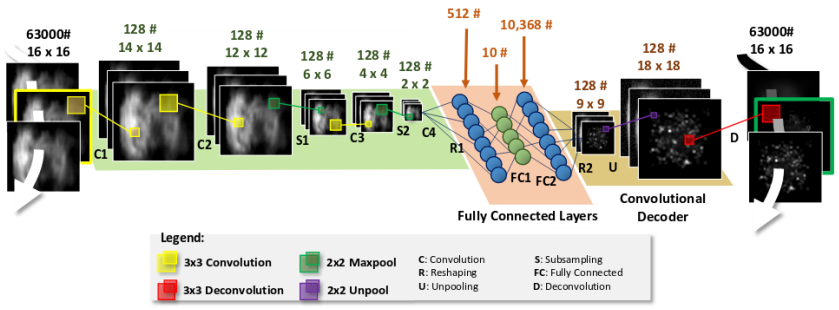


Figura 10: (Fuente: A. Akintano et al. *Prognostics of Combustion Instabilities from Hi-speed Flame Video using A Deep Convolutional Selective Autoencoder*, Int. J. of Prognostics and Health Management (2016))

6. Apéndice

Función de pérdida de los autoencoders variacionales

En primer lugar recordamos la definición de Divergencia KL de una función f respecto de g:

$$\mathcal{D}(f\|g) = \mathbb{E}_f[\log f - \log g] = -\mathbb{E}_f[\log g - \log f],$$

y se cumple que $\mathcal{D}(f\|g) \geq 0$.

Sea $p(x, z)$ la distribución conjunta de una variable observable x y una variable oculta z.

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} = \frac{p(x|z)p(z)}{\int p(x|z)p(z)dz}$$

En muchas ocasiones la integral del denominador es intratable.
↓
Renunciamos a calcular $p(z|x)$.
En su lugar intentamos encontrar otra distribución $q(z|x)$ tal que:

- no nos dé problemas y...
- que sea lo más parecida a $p(z|x)$

Para lograr el segundo punto debemos minimizar la divergencia KL de $q(z|x)$ respecto de $p(z|x)$.

$$q(z|x) = \arg\min \{\mathcal{D}\left(q(z|x)\|p(z|x)\right)\} = \arg\min \left\{\mathbb{E}_{q(z|x)}\left[\log q(z|x) - \log p(z|x)\right]\right\}$$

Sea ℓ el funcional que queremos minimizar.
Teniendo que cuenta que $\log p(z|x) = \log p(x, z) - \log p(x)$,

$$\ell = \mathbb{E}_{q(z|x)}[\log q(z|x) - \log p(x, z) + \log p(x)].$$

En la expresión de arriba, $\log p(x)$ no depende de z y por tanto tampoco de su distribución, así que puede salir del operador $\mathbb{E}_{q(z|x)}$.
Además hay que recordar que el valor esperado es una operación lineal. Reagrupando:

$$\ell = \log p(x) + \mathbb{E}_{q(z|x)}[\log q(z|x) - p(x, z)].$$

Recordar también que ℓ es el funcional que queríamos minimizar, así que

$$\ell = \mathbb{E}_{q(z|x)}\left[\log q(z|x) - \log p(z|x)\right] = \mathcal{D}\left(q(z|x)\|p(z|x)\right)$$

Igualamos una con otra y reordenando términos llegamos a:

$$\log p(x) = \mathcal{D}\left(q(z|x)\|p(z|x)\right) - \mathbb{E}_{q(z|x)}[\log q(z|x) - \log p(x, z)].$$

Podemos cambiar el signo del término más a la derecha para que sólo aparezcan sumas:

$$\log p(x) = \mathcal{D}\left(q(z|x)\|p(z|x)\right) + \mathbb{E}_{q(z|x)}[\log p(x, z) - \log q(z|x)].$$

En esta última expresión:

- Como el dato x es conocido, $\log p(x) \geq 0$ es un valor constante y conocido,
- y como $\mathcal{D}\left(q(z|x)\|p(z|x)\right) \geq 0$, entonces $\log p(x) \geq \mathbb{E}_{q(z|x)}[\log p(x, z) - \log q(z|x)]$.

En definitiva, $\log p(x)$ no puede ser nunca menor que $\mathbb{E}_{q(z|x)}[\log p(x,z) - \log q(z|x)]$.
Es decir que el término

$$\mathcal{L} = \mathbb{E}_{q(z|x)}[\log p(x,z) - \log q(z|x)] = \int q(z|x) \log \frac{p(x,z)}{q(z|x)}$$

es un límite inferior, al que se le da el nombre de *Evidence Lower BOund* o ELBO.

Además es importante darse cuenta de que x es un dato observable, por tanto $\log p(x)$ es un valor conocido y constante. Entonces, si reducimos $\mathcal{D}\left(q(z|x)\|p(z|x)\right)$, tenemos que aumentar ELBO.

↓

Puesto que minimizar $\mathcal{D}\left(q(z|x)\|p(z|x)\right)$ es efectivamente el objetivo que teníamos, al maximizar ELBO logramos lo mismo.

↓

$$\text{máx}\{\mathcal{L}\} = \text{mín}\{-\mathcal{L}\} = \text{mín}\{\mathbb{E}_{q(z|x)}[\log q(z|x) - \log p(x,z)]\} = \text{mín}\{\mathcal{D}_{q(z|x)}(q(z|x)\|p(x,z))\}$$

Reescribiendo $p(x,z) = p(x|z)p(z)$ el funcional de la expresión anterior queda

$$\begin{aligned}\mathcal{D}_{q(z|x)}(q(z|x)\|p(x,z)) &= \mathbb{E}_{q(z|x)}[\log q(z|x) - \log p(x|z) - \log p(z)] \\ &= \mathbb{E}_{q(z|x)}[\log q(z|x) - \log p(z)] - \mathbb{E}_{q(z|x)}[\log p(x|z)] \\ &= \mathcal{D}_{q(z|x)}(q(z|x)\|p(z)) - \mathbb{E}_{q(z|x)}[\log p(x|z)]\end{aligned}$$

En definitiva, reescribiendo $p(x,z) = p(x|z)p(z)$ obtenemos

$$\begin{aligned}\text{mín}\{\mathcal{D}_{q(z|x)}(q(z|x)\|p(x,z))\} &= \text{mín}\{\mathcal{D}_{q(z|x)}(q(z|x)\|p(z)) - \mathbb{E}_{q(z|x)}[\log p(x|z)]\} \\ &= \text{máx}\{\mathbb{E}_{q(z|x)}[\log p(x|z)] - \mathcal{D}_{q(z|x)}(q(z|x)\|p(z))\}\end{aligned}$$

Recopilando todo:

Buscamos una función $q(z|x)$ que sea lo más parecida posible a $p(z|x)$

→ Será aquella que mín $\{\mathcal{D}\left(q(z|x)\|p(z|x)\right)\}$

→ Esto es equivalente a $\text{máx}\{\mathcal{L}\} = \text{mín}\{-\mathcal{L}\} = \text{mín}\{\mathcal{D}_{q(z|x)}(q(z|x)\|p(x,z))\}$

→ Esto es equivalente a $\text{máx}\{\mathbb{E}_{q(z|x)}[\log p(x|z)] - \mathcal{D}_{q(z|x)}(q(z|x)\|p(z))\}$

Aunque desconozcamos $p(z|x)$, utilizando nuestra distribución $q(z|x)$ obtenemos una distribución de la variable oculta z a partir del dato x. La selección de $q(z|x)$ viene dada por la resolución del problema de optimización dado.

Si consideramos que x son imágenes y z son códigos, entonces $q(z|x)$ es la distribución de los códigos dados los ejemplos, es decir el *encoder*, mientras que $p(x|z)$ es la distribución de las imágenes dados los códigos, es decir el *decoder*.

Con esta interpretación:

- el término $\mathbb{E}_{q(z|x)}[\log p(x|z)]$ es el promedio de lo bien que se reconstruyen los códigos creados por $q(z|x)$

  Cuanto más parecidas sean $q(z|x)$ y $p(z|x)$ mayor será este promedio

- el término $-\mathcal{D}_{q(z|x)}(q(z|x)\|p(z))$ me permite seleccionar una distribución a priori de los códigos $p(z)$ y, debido al signo negativo, penalizar las $q(z|x)$ que se alejen de dicha selección. Es decir, cumple funciones de regularización.

  Cuanto más parecidas sean $q(z|x)$ y $p(z|x)$ menor será esta penalización.