Aplicaciones de los Autoencoders

Índice

Referencia principal

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

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

1. Compresión de imágenes

En el tema anterior estudiamos cómo el cuello de botella de los autoencoders representa una reducción de la dimensionalidad de los datos de entrada para obtener una representación eficiente. Esto se consigue con una de codificación, resultado de la red *encoder*, y una de reconstrucción, llevada a cabo por la red *decoder*.

La aplicación más directa y obvia de este tipo de redes es poder enviar información *pesada* (por ejemplo imágenes con muchísima resolución) a través de un ancho de banda pequeño. El emisor implementaría el *encoder* y el receptor el *decoder*, ambos entrenados previamente y ya listos para ser puestos en producción. Evidentemente el canal de comunicación transmite el código resultante.

Un modo de obtener representaciones más eficaces consiste en construir los autoencoders de modo que se reduzcan las neuronas activas del *encoder*. La intuición es: *si restringimos el uso de algo drásticamente, sólo utilizará lo estrictamente necesario*. En este caso, si sólo el 5 % de las neuronas del *encoder* sobreviven, entonces el aprendizaje forzara a representar cada dato de entrada como la combinación de un pequeño número neuronas, y por tanto a que cada neurona capture características más útiles.

Un modo de lograr esto es añadiendo una penalización de dispersión (*sparsity loss*) a la función de pérdida. En primer lugar hay que decir la dispersión que se desea alcanzar como un hiperparámetro más de la red. A efectos prácticos esto significa que inicializamos una variable con el valor buscado. Para saber si estamos cerca o lejos de ese valor objetivo tenemos que medir la dispersión de nuestro *encoder*. Para concretar, supongamos que el *encoder* tiene 1 sola capa. Como la salida de cada una de sus neuronas es precisamente el resultado de su activación, podemos estimar la dispersión como el promedio de estas salidas. Si, por ejemplo, la dispersión objetivo es de 0.1 (10 %) y el promedio es de 0.3, entonces podemos estimar que estamos desviados $(0.1\text{-}0.3)^2$ y añadir este término a la función de pérdida global.

Alternativamente, si las activaciones son tales que la salida está entre 0 y 1, por ejemplo la sigmoide, entonces su promedio también lo estará, y por tanto lo podemos interpretar como una probabilidad. De este modo, la penalización por dispersión puede medirse en términos de la divergencia KL entre el objetivo y el promedio. Sea p el valor objetivo de dispersión, interpretado como la probabilidad de que una neurona del *encoder* se active; y sea q el promedio, interpretado como la misma probabilidad. Como ambos sucesos son variables aleatorias binarias (sí/no se activa) se pueden modelar con una distribución Bernoulli, es decir:

$$P(x) = (x)^p(1-x)^{1-p} \quad , \quad Q(x) = (x)^q(1-x)^{1-q}; \quad \text{con } x = \{0,1\}.$$

La divergencia KL queda entonces:

$$\mathcal{D}(P\|Q) = p\log\frac{p}{q} + (1-p)\log\frac{1-p}{1-q}$$

Y finalmente la función de pérdida para entrenar el autoencoder sería:

$$\ell^{(i)}(\mathbf{x}^{(i)}, \mathbf{z}) = \ell_{\text{Reconstruct}}(\mathbf{x}^{(i)}, \mathbf{z}) + \eta \cdot \mathcal{D}(P\|Q),$$

donde $\ell_{\text{Reconstruct}}$ es la pérdida de la reconstrucción, por ejemplo el MSE entre la imagen original y la obtenida (como ya vimos en el tema anterior), y η es un hiperparámetro con el que decidimos la importancia que le damos a este término.

2. Eliminación de ruido

La aplicación anterior saca partido al código que se genera en los autoencoders.

Otra aplicación interesante, que explota el modo en el que se entrenan los autoencoders es la eliminación de ruido. Como el conjunto de entrenamiento se utiliza tanto a la entrada como a la salida, podemos modificar los datos de entrada añadiendo un ruido gaussiano. En definitiva, el autoencoder recibirá datos con ruido y generará datos que se comparan con los datos limpios (que son los datos originales). Por tanto la actualización de los pesos del autoencoder por la retropropagación del error dará lugar a una red que aprende a limpiar los datos de entrada.

Una manera alternativa a añadir ruido consiste en hacer dropout tras la capa de entrada, tal y como se muestra en la Figura 1

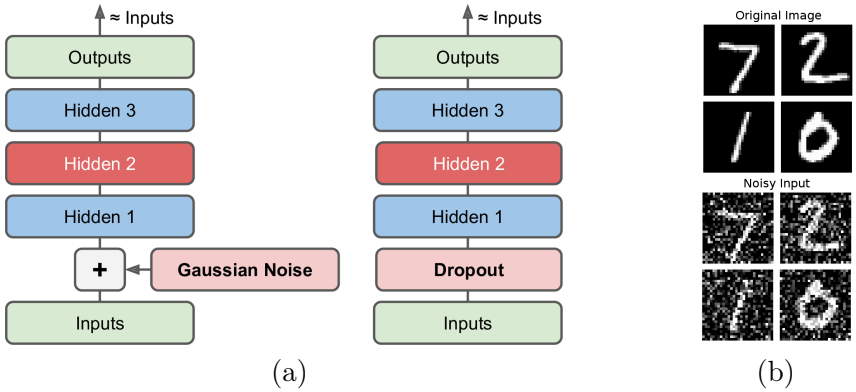


Figura 1: (a) Esquema de un autoencoder para eliminación de ruido. (b) Ejemplos del MNIST con ruido frente al original. (Fuente: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, A.Gerón)

Los autoencoders para eliminación de ruido (*Denoising AutoEncoders* o DAE) fueron publicados por primera vez por P.Vicent et al. en *Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion*, JMLR 2010. En el artículo se redefine el término "buena representación" (en el sentido de un buen vector de características que represente el ejemplo *crudo*) como aquella que puede ser obtenida robustamente a partir de entradas corruptas y que puede resultar útil para recuperar la entrada limpia.

Esta aproximación implica dos ideas fundamentales. En primer lugar es esperable que una representación de más alto nivel sea bastante más estable y robusto frente a corrupción en la entrada. También es lógico pensar que para llevar a cabo con un buen rendimiento la tarea de eliminar ruido es importante capturar y extraer las características importantes de la entrada.

Esta segunda idea es defendida por los autores de dicho artículo explicando que, primero se pueden utilizar diferentes DAEs para extraer diferentes representaciones y, después, evaluar objetivamente cada una de ellas comparando las precisiones alcanzadas con clasificadores que las utilicen como entradas.

Por último, para poder comparar resultados, es decir para saber cuanto ha sido limpiada una imagen, es frecuente utilizar el ratio de Pico de señal frente a Ruido (*Peak Signal to Noise Ratio*, PSNR). Sea I una imagen monocroma, libre de ruido, de resolución $(m \times n)$; y sea K la imagen con ruido, el PSNR, medido en decibelios (dB), se define como

$$PSNR = 20 \log_{10} (MAX_I) - 10 \log_{10} (MSE),$$

donde MSE es el error cuadrático medio entre los píxeles de I y de K; y MAX_I es el valor máximo posible de un píxel en la imagen, por ejemplo usando 8 bits para la intensidad, $MAX_I = 255$.

3. *Neural inpainting*

Eliminar ruido es, en definitiva, estimar el valor de cada uno de los píxeles de la imagen de tal manera que el resultado quede visualmente *limpio*.

El *inpainting* es esencialmente lo mismo, pero con la diferencia de que la imagen de entrada tiene zonas ocultas en vez de píxeles distribuidos por la imagen y modificados aleatoriamente. En la Figura 2 se muestran dos ejemplos. En el de la izquierda se elimina el texto escrito sobre la imagen, mientras que en la derecha se rellena el área morada.

(a) (b)

Figura 2: (a) J. Xie, L. Xu, E. Chen. *Image Denoising and Inpainting with Deep Neural Networks*; NIPS, 2012 (b) C. Yang et al. *High-Resolution Image Inpainting using Multi-Scale Neural Patch Synthesis*; CVPR, 2017

La diferencia esencial entre ambas figuras es la extensión del área donde faltan los píxeles de la imagen. En eliminación de texto, y otras aplicaciones similares como la eliminación de marcas de agua, el área total puede ser relativamente grande, pero los píxeles perdidos o modificados, aunque puedan tener una distribución, están dispersos. En este caso el tratamiento es similar al de eliminación de ruido.

Sin embargo en rellenado de imagen, normalmente los píxeles perdidos ocupan una superficie compacta y extensa. En general, hay dos grupos de soluciones a este problema. Por un lado aquellas que extienden las texturas de las regiones que rodean al hueco, primero a trazo grueso y, en iteraciones sucesivas cada vez más fino (aproximación *coarse-to-fine*). Estas soluciones son buenas para propagar detalles de alta frecuencia en las texturas pero no capturan bien la semántica o la estructura global de la imagen. Es decir que extienden texturas con fidelidad pero el resultado no tiene sentido si el agujero que deben rellenar tiene elementos de aspecto o textura diferente a aquello que le rodea. Por ejemplo si en la foto de un perro recortamos el hocico y rellenamos con texturas de alrededor podría salir un ojo en su lugar.

Por otro lado, están aquellas soluciones dirigidas por datos almacenados en grandes bases de datos. Los huecos se rellenan asumiendo que regiones rodeadas de un contexto similar tendrán un contenido similar. En el ejemplo anterior, si tenemos muchas fotos de perro y somos capaces de detectar que lo que falta en la imagen es el hocico, entonces el objetivo es generar ese hocico.

Uno de los trabajos más recientes que aborda este problema es el árticulo de Yang et al. *High-Resolution Image Inpainting using Multi-Scale Neural Patch Synthesis*; CVPR, 2017 En él se propone un método híbrido, que utiliza redes neuronales para generar parches realistas, con detalles de alta frecuencia por un lado, y la capacidad de predecir estructuras de los autoencoders convolucionales por el otro.

En concreto se utilizan dos redes neuronales, una para contenidos, construida como un autoencoder; y una para texturas, que consiste en una red convolucional. El esquema general se muestra en la Figura 3.

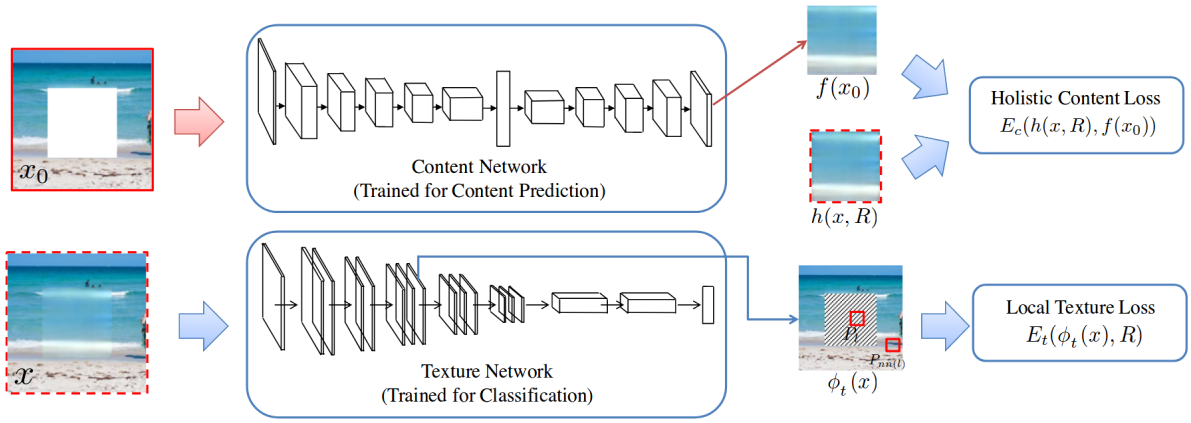


Figura 3: Redes de contenido y textura empleadas en el trabajo de Yang et al.. Los símbolos y expresiones que aparecen en la figura están explicados en el texto. (Fuente: C. Yang et al. *High-Resolution Image Inpainting using Multi-Scale Neural Patch Synthesis*; CVPR, 2017 🔗)

Antes de explicar cada una de estas redes hay que fijar la notación:

- x_0 es la imagen de entrada, que contiene una región R hueca,
- x_g es la imagen completa (*ground truth*),
- $h(x, R)$ es una función que devuelve el contenido de x dentro de R,
- $f(x)$ es la función que implementa la red de contenidos,
  e.g. $f(x_0)$ es lo que devuelve la red de contenidos cuando se introduce una imagen x_0.
- $t(x)$ es la función que implementa la red de texturas
- $\phi_t(x)$ es el mapeado de características que produce la red de texturas.
- R^ϕ es la región R pero en los mapas de características obtenidos por $\phi_t(x)$.

El método propuesto es multi-resolución. Primero se reduce la imagen y, utilizando la red de contenido, se hace una estimación de como rellenar el hueco. Después se refina esta estimación con la restricciones de textura a esta baja resolución. Entonces se aumenta la resolución con *upsampling* y la imagen resultante sirve de inicialización para la nueva resolución.

Función de pérdida

En definitiva, para cada resolución $i = 1, 2, \dots N$, la reconstrucción óptima x^*_{i+1} es aquella que:

$$x^*_{i+1} = \arg\min \left[E_c\big(h(x_i, R), f(x_0)\big) + \alpha E_t(\phi_t(x), R) + \beta \Upsilon(x) \right],$$

donde E_c es la pérdida debida a la red de contenidos, E_t es la debida a la red de texturas y $\Upsilon(x)$ es la pérdida por variación total (*total variation loss*), que ayuda obtener una imagen "suave", y viene dado por la expresión:

$$\Upsilon(x) = \sum_{i,j} \big(x_{i,j+1} - x_{i,j}\big)^2 + \big(x_{i+1,j} - x_{i,j}\big)^2$$

Red de contenidos

La función de pérdida penaliza la diferencia ℓ_2 con la predicción obtenida en la resolución anterior.

$$E_c\big(h(x, R), f(x_0)\big) = \|h(x, R) - h(x_i, R)\|_2^2$$

Red de texturas

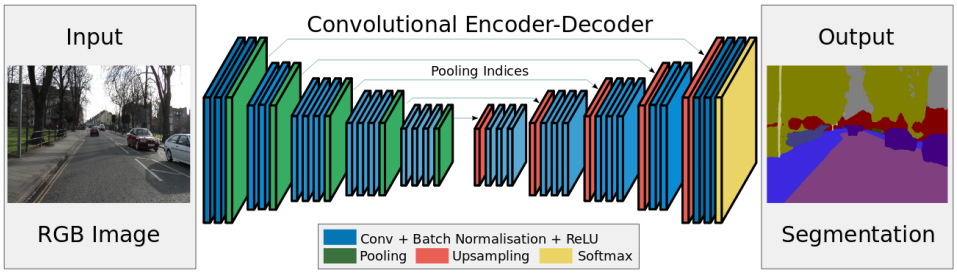
La meta de esta red es asegurar que los detalles dentro de la región R son similares a los detalles fuera de ella. Para ello se penaliza la discrepancia entre la textura dentro y fuera del relleno. Para medir esta similitud se utilizan los "parches neuronales" (*neural patches*). Se elige un mapa de características, o una combinación de ellos y, se toman una serie de parches (vecindades de un cierto tamaño). Por cada parche P_l se calcula su pérdida como la media de su distancia a su vecino más cercano $P_{nn(l)}$.

$$E_t(\phi_t(x), R) = \frac{1}{|R^\phi|} \sum_{i \in R^\phi} \|h\big(\phi_t(x), P_i\big) - h\big(\phi_t(x), P_{nn(i)}\big)\|_2^2,$$

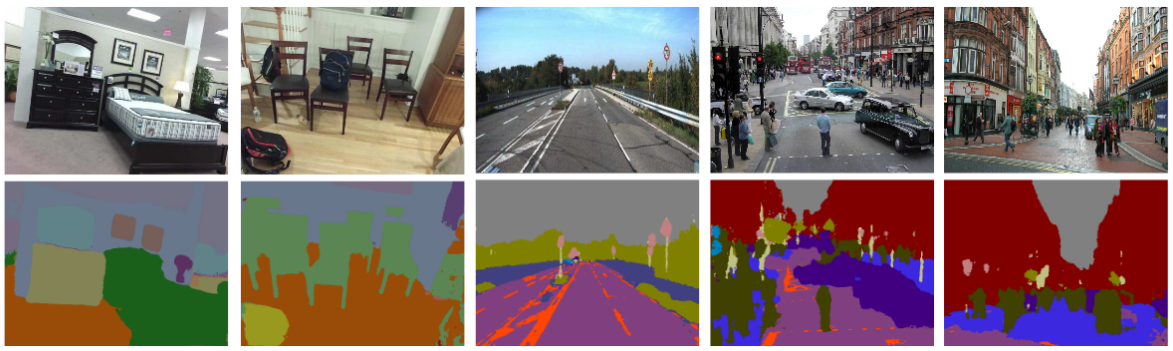
donde $|R^\phi|$ es el número de parches tomados en la región R^ϕ .

4. Segmentación

La segmentación, desde el punto de vista de redes neuronales, consiste en asignar una clase a cada píxel de la imagen. Esa clase indica el objeto al que pertenece. Por tanto el resultado es una imagen de la misma resolución que el original pero con un aspecto diferente. En la Figura 4(b) se puede ver un ejemplo.



(a)



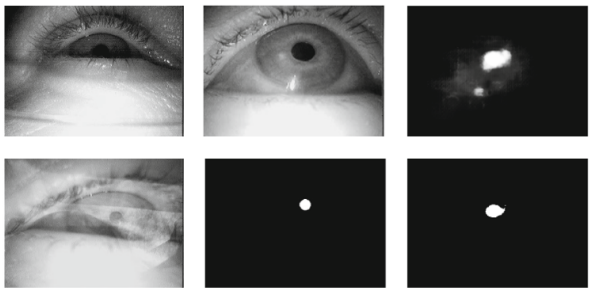
(b)

Figura 4: (a) Arquitectura de la SegNet. (b) Algunos resultados. (Fuente: V. Badrinarayanan, A. Kendall, R. Cipolla; *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*; IEEE TPAMI, 2017)

Como se puede ver en la Figura 4(a), la arquitectura es la de un autoencoder ya que la imagen de entrada y salida son de la misma resolución. Sin embargo no son la misma imagen. Al contrario que los autoencoders, ahora sí necesitamos un conjunto de datos etiquetado ya que la salida

de la red se compara con una salida generada por el humano, que consiste en la imagen de entrenamiento segmentada.

Un ejemplo interesante de aplicación es la detección de la pupila. Esto tiene interés en interfaces para gente con movilidad reducida, para controlar aparatos, y también para detectar y prevenir distracciones al volante.



(a) (b) (c)

Figura 5: (a) Imágenes grabadas del ojo mientras se conduce; con bastante ruido en muchos casos. (b) Imagen junto con el *Ground truth* de la pupila. (c) Resultados de la segmentación, antes y después de la umbralización. (Fuente: F.J. Vera-Olmos, N. Malpica; *Deconvolutional Neural Networks for Pupil Detection in Real-World Environments*; IWINAC, 2017)

En primer lugar es necesario un conjunto etiquetado, e.g. el de la universidad de Tuebingen . Dicho conjunto consiste en varias fotos tomadas con una cámara colocada en el parabrisas enfocando a la cara del conductor. De cada frame se ha extraído un recorte del ojo y se ha creado el *ground truth*, que es una imagen de la misma resolución que el recorte, pero donde todos los píxeles son 0 excepto aquellos que pertenecen al centro de la pupila. En la Figura 5(a) se muestra un ejemplo de recortes de ojo y en la Figura 5(b) un ejemplo de *ground truth*. El trabajo de Vera-Olmos y Malpica (2017) utiliza una red de segmentación doble, una con la imagen a la resolución original y otra con la imagen a menor resolución, para segmentar la pupila. La figura 5(c) muestra un ejemplo de resultado (arriba), al que después aplica una umbralizacion (abajo) para refinar la posición.

Por último, el estado del arte en segmentación lo logra Segnet, creada en la universidad de Cambridge .

5. Ejercicios prácticos

- Construir y probar estas aplicaciones con los diferentes autoencoders estudiados en el tema anterior.
- Otra aplicación interesante para autoencoders es colorear imágenes en escala de grises (*image colorization*). Pues leer más sobre ello en el artículo de Baldassarre et al. *Deep Koalarization: Image Colorization usingCNNs and Inception-Resnet-v2*
- Las redes U-net son un tipo de autoencoders con una arquitectura especial. Aprende más sobre ellas e intenta consutruir una a partir del artículo de Ronneberger et al. *U-Net: Convolutional Networks for BiomedicalImage Segmentation* .