Humanidades, Tecnología y Ciencia,
del Instituto Politécnico Nacional

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

REDES NEURONALES RECURRENTES PARA EL PROCESAMIENTO DE SECUENCIAS CODIFICADO EN PYTHON

Israel Rivera Zárate
Instituto Politécnico Nacional-CIDETEC
irivera@ipn.mx

Miguel Hernández Bolaños
Instituto Politécnico Nacional-CIDETEC
mbolanos@ipn.mx

Patricia Pérez Romero
Instituto Politécnico Nacional-CIDETEC
promero@ipn.mx

Abstract

The difference between a recurrent neural network (RNN) and other architectures such as neural (NN) or convolutional networks (CNN) lie in the type of data they can analyze. Recurrent networks can analyze data sequences, the other two architectures cannot. RNNs can use their internal state memory to process variable length sequences of inputs; therefore, they are assigned to tasks such as handwriting recognition or speech recognition. This research shows the basics of a RNN's operation.

Palabras clave: Red recurrente, Procesamiento de Secuencias, Lenguaje Python.

Introducción

Se ha expuesto de acuerdo con Goodfellow et al. (2016), que la diferencia entre una red recurrente y otras arquitecturas, como las redes neuronales o convolucionales, radica en el tipo de datos que pueden analizar. Las redes recurrentes están en capacidad de analizar secuencias de datos, las otras dos arquitecturas no. Si a una red neuronal o convolucional se le presenta por ejemplo una imagen o una palabra, con el entrenamiento adecuado estas arquitecturas lograrán un sin número de datos alcanzando a la vez una alta precisión. Pero, si en vez de una palabra se introduce una secuencia de imágenes, es decir un video, o una secuencia de palabras: una conversación; en este caso en ninguna de estas redes será capaz de procesar los datos. En primer lugar, porque estas arquitecturas están diseñadas para que los datos de entrada y de salida siempre tengan el mismo tamaño; sin embargo, un video o una conversación se caracterizan por ser un tipo de dato con un

tamaño variable: una cantidad variable de "*frames*" en el caso del video o una cantidad variable de palabras en el caso de una conversación. En segundo lugar, en un video o en una conversación los datos están correlacionados, esto quiere decir que la siguiente palabra pronunciada o la siguiente imagen en la secuencia de video dependen de la palabra o imagen anterior, e incluso estas palabras e imágenes están relacionadas con aquellas que se presenten más adelante en la secuencia. Una secuencia es por ejemplo un texto escrito, para comprender su contenido no basta con leer cada palabra de manera individual, el cerebro concatena todas las palabras leídas hasta el momento permitiendo comprender la idea central de dicho texto. Así una secuencia es una serie de datos: imágenes, palabras, notas musicales, sonidos que siguen un orden específico y tienen únicamente significado cuando se analizan en conjunto y no de manera individual. Ver figura 1.

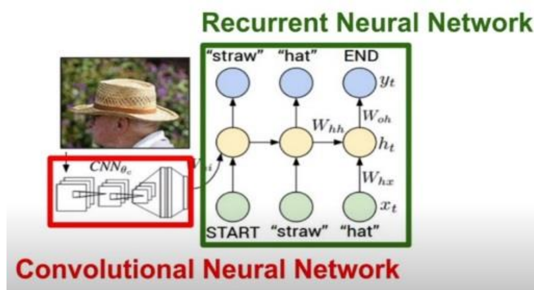


Figura 1. Red recurrente como descriptor de contenido. Jeff Donahue et al. (2014). Long term recurrent convolutional networks for visual recognition and description.

I. Redes Neuronales Recurrentes

Según expresa Graves Alex, (2014). Las redes neuronales recurrentes (RNN) pueden procesar tanto a la entrada como a la salida secuencias sin importar su tamaño, y además teniendo en cuenta la correlación existente entre los diferentes elementos de esa secuencia. Para ello este tipo de redes usan el concepto de "recurrencia", para generar la salida, que también se conoce como activación, la red usa no sólo la entrada actual sino la activación generada en la iteración previa. En pocas palabras, las redes recurrentes utilizan un tipo de memoria para generar la salida deseada, como se describe a continuación:

A. Redes uno a muchos ("*one to many*"): Donde la entrada es un único dato y la salida es una secuencia. Un ejemplo es el "*image captioning"* donde la entrada es una imagen y la salida es una secuencia de caracteres, un texto, que describe el contenido de la imagen.

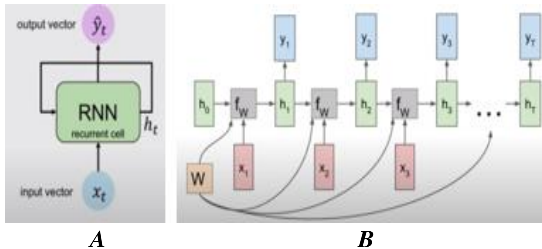
B. Redes muchos a uno ("*many to one*"): Donde la entrada es una secuencia y salida es una categoría. Un ejemplo de esto es la clasificación de sentimientos, donde la entrada es un texto que contiene la crítica a una película y la salida es una categoría indicando si la película le gustó a la persona o no.

C. Redes muchos a muchos ("many to many"): Son aquellas donde tanto a la entrada como a la salida se tienen secuencias. Un ejemplo son los traductores automáticos donde se requiere conocer la totalidad del texto de entrada para producir el texto de salida en otro idioma. En esta misma categoría se encuentran los convertidores de voz a texto o texto a voz que son redes recurrentes cuya entrada y salida son secuencias.

II. Estructura de la RNN

Como indica Fei-Fei et al. (2017). En una RNN el instante de tiempo "*time-step*" es un número entero que define la posición de cada elemento dentro de una secuencia $X_{(t)}: \{X_{(1)}, X_{(2)}, \ldots X_{(T)}\}$ donde $X_{(t)}$: entrada a la RNN en el instante del tiempo t, $Y_{(t)}$: salida de la RNN en el instante de tiempo t. En la figura 2 se pueden observar varios bloques, sin embargo, la RNN es una sola, la idea es mostrar las entradas y las salidas en diferentes instantes de tiempo. Se observa que en cada momento la red tiene dos entradas y dos salidas, las entradas son el dato actual $X_{(t)}$ y la activación anterior $h_{(t-1)}$, mientras que las

salidas son la predicción actual $Y_{(t)}$, y la activación actual $h_{(t)}$. Esta activación recibe el nombre de "*hidden state*" o estado oculto. Son estas activaciones precisamente las que corresponden a la memoria de la red; pues permiten preservar y compartir la información entre un instante de tiempo y otro.



A ***B***

Figura 2. Red recurrente: A. Compacta. B. Desplegada. http://colah.github.io/posts/2015-08-Understanding-LSTMs.

Para calcular la salida y la activación de la RNN a partir de sus dos entradas se usa la misma lógica de una neurona artificial convencional donde dada una entrada X y salida Y donde se aplican dos operaciones: una transformación y una función de activación no lineal. En la RNN la activación se obtiene primero transformando los datos de entrada: la activación anterior y la entrada actual y luego llevarlos a una función de activación no lineal. Los valores más adecuados de los coeficientes W_{hh}, W_{xh}, y W_{hy}, se obtienen a través del entrenamiento como sucede en las redes neuronales feed forward.

De igual forma, para obtener la salida se usa la activación del instante previo y se usan las mismas operaciones: transformación y función de activación, aplicadas anteriormente. Al igual que en el caso anterior, los coeficientes W_{hy} se obtienen durante el proceso de entrenamiento. El concepto de recurrencia, así como el de memoria asociada a las RNN se observa en estas dos ecuaciones ya que la salida $Y_{(t)}$, depende de la activación actual $h_{(t)}$, pero a su vez esta activación depende no solo de la entrada actual $X_{(t)}$ sino también del valor previo de la activación $h_{(t-1)}$, esto es precisamente la memoria de la red y la forma como este concepto permite preservar y compartir la información entre uno y otro instante de tiempo. Ver ecuaciones 1, 2 y 3.

$$h_{(t)} = f_W\left(h_{(t-1)}, X_{(t)}\right) \ldots 1$$

$$h_{(t)} = \tanh\left(W_{hh}^T\, h_{(t-1)} + W_{xh}^T\, X_{(t)}\right) \ldots 2$$

$$Y_{(t)} = W_{hy}\, h_{(t)} \ldots 3$$

Es importante resaltar que los coeficientes a calcular durante el entrenamiento serán los mismos entre uno y otro instante de tiempo. La idea es que, una vez entrenada la red, sea capaz de generar la predicción usando el mismo conjunto de parámetros en cada instante de tiempo. Debido a esto, se usa una representación compacta donde la flecha indica la dependencia entre la activación actual y la generada en un instante de tiempo anterior.

III. Entrenamiento de la RNN

Tal como lo indica Duchi John et al. (2011), el descenso del gradiente es el algoritmo de optimización utilizado para el entrenamiento; donde se busca minimizar la función de error de los valores de salida obtenidos en relación con los esperados respecto a los parámetros de la red (producto de derivadas parciales empleando la regla de la cadena). Ver ecuaciones 4, 5 y 6.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} \ldots 4$$

$$\frac{\partial L_t}{\partial W} = \sum_{K=1}^{t} \frac{\partial L_t}{\partial Y_t}\, \frac{\partial Y_t}{\partial h_t}\, \frac{\partial h_t}{\partial h_K}\, \frac{\partial h_K}{\partial W} \ldots 5$$

$$W_{new} = W - \eta\, \nabla_W L \ldots 6$$

El resultado total del error es la suma de las derivadas parciales calculadas en cada intervalo *time-step*. El proceso termina cuando

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

se logra ajustar los pesos de las neuronas (coeficientes W_{hh}, W_{xh}, y W_{hy}) de ahí que se llama "backpropagation through time" (BTT), como se observa en la figura 3.

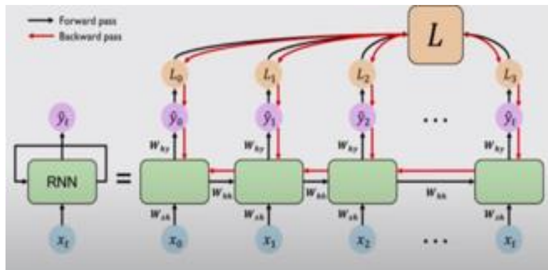


Figura 3. Evaluación del error a través del tiempo. http://colah.github.io/posts/2015-08-Understanding-LSTMs.

El empleo de funciones de activación tanh o bien sigmoide cuyos valores son menores a la unidad en el cero de las abscisas y muy altos fuera del mismo conducen a que en el proceso iterativo del producto de las derivadas parciales se generen valores del gradiente muy cercanos al cero (desvanecimiento: *"vanishing gradients"*) o bien valores muy elevados (explosión: "*exploding gradients*") lo que implica un elevado tiempo de cómputo para converger. La red deja de aprender deteniéndose el entrenamiento. Por esto, en la práctica estas redes no superan más de cien niveles de profundidad y se dice que la red tiene memoria a corto plazo pero que no es apta para aprender relaciones donde la información esté muy distante o de largo plazo.

IV. Caso de estudio

Se ha propuesto un ejemplo en Python basado en el trabajo de Karpathy Andrej, (2018). Donde se busca obtener una red que produzca un carácter en cada instante de tiempo. La red lee una secuencia de caracteres y predice cuál será el siguiente carácter del *stream* de texto. En este ejemplo se tiene un vocabulario muy pequeño compuesto de cuatro letras exclusivamente: h, e, l y o. Se buscará entrenar a la red para producir la secuencia: h, e, l, l, o.

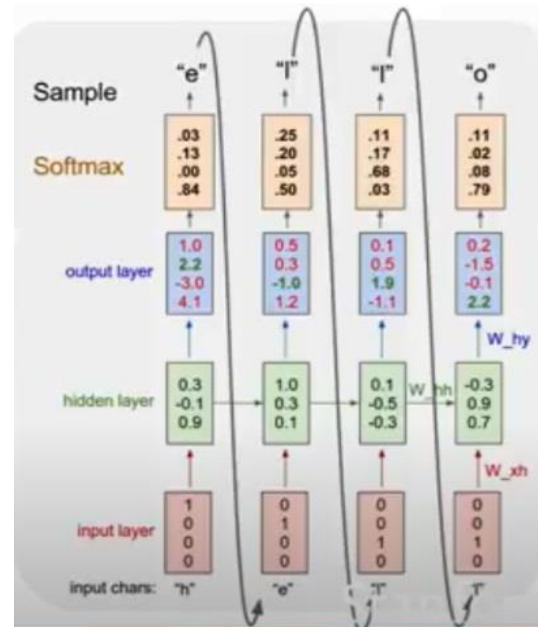


Figura 4. Ejemplo: Arquitectura de la RNN para la generación de la secuencia h, e, l, l, o. http://colah.github.io/posts/2015-08-Understanding-LSTMs.

La RNN recibe el primer carácter y el modelo predice una salida como se muestra en la figura 4. En el siguiente *time-step* el carácter generado es ahora la entrada a la red (recurrencia), este paso se repite hasta obtener la secuencia buscada. Se puede observar que en el siguiente *time-step* se introducen la entrada y el estado oculto o de memoria (*"hidden layer*"). La función *softmax* convierte los valores de la capa de salida en probabilidades relacionadas a los caracteres.

A. Importación de librerías y lectura de los datos. Permite crear una celda recurrente, utilizar un optimizador para entrenar el modelo, la conversión de caracteres a vectores *one-hot* y viceversa. Ver fragmento de código en la figura 5.

```
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }
```

Figura 5. Rivera, Hernández y Pérez (2020). Fragmento de código: Lectura de datos.

B. Implementación. El modelo tendrá dos entradas: el carácter en formato *one-hot* y el estado oculto anterior. Dos salidas: la predicción o carácter generado y el estado oculto actual producido por la función de activación *tanh*. Una capa de salida *softmax* que tomará la activación de la celda recurrente y generará la predicción. Ver figura 6.

```
# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

# forward pass
for t in xrange(len(inputs)):
  xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
  xs[t][inputs[t]] = 1
  hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
  ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
  ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
  loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
```

Figura 6. Rivera et al. (2020). Fragmento de Código: Implementación.

C. Entrenamiento. Se lleva acabo el optimizador gradiente descendente calculando las derivadas parciales y aplicando la regla de la cadena, como se observa en el fragmento de código de la figura 7.

```
# backward pass: compute gradients going backwards
dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
dbh, dby = np.zeros_like(bh), np.zeros_like(by)
dhnext = np.zeros_like(hs[0])
for t in reversed(xrange(len(inputs))):
  dy = np.copy(ps[t])
  dy[targets[t]] -= 1 # backprop
  dWhy += np.dot(dy, hs[t].T)
  dby += dy
  dh = np.dot(Why.T, dy) + dhnext # backprop into h
  dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
  dbh += dhraw
  dWxh += np.dot(dhraw, xs[t].T)
  dWhh += np.dot(dhraw, hs[t-1].T)
  dhnext = np.dot(Whh.T, dhraw)
for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
  np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

Figura 7. Rivera et al. (2020). Fragmento de código: Entrenamiento.

D. Generación de palabras. La activación inicial será una matriz de ceros tanto para la entrada como para el estado oculto anterior. Posteriormente se lleva la activación resultante a la capa *softmax* para así generar la predicción que será un vector que representa la distribución de probabilidad para escoger aleatoriamente un elemento. Finalmente se busca el carácter según el vocabulario y se actualizan las entradas; la predicción generada se convertirá en la entrada a la celda recurrente para el siguiente *time-step*, mientras la activación generada en esta iteración corresponde con el nuevo estado oculto a usar como entrada en la siguiente iteración. El proceso se repite de forma iterativa produciendo uno a uno los caracteres de la secuencia. Ver figura 8.

```
sample a sequence of integers from the model
h is memory state, seed_ix is seed letter for first time step
"""
x = np.zeros((vocab_size, 1))
x[seed_ix] = 1
ixes = []
for t in xrange(n):
  h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
  y = np.dot(Why, h) + by
  p = np.exp(y) / np.sum(np.exp(y))
  ix = np.random.choice(range(vocab_size), p=p.ravel())
  x = np.zeros((vocab_size, 1))
  x[ix] = 1
  ixes.append(ix)
return ixes
```

Figura 8. Rivera et al. (2020). Fragmento de código: Generación de palabras.

V. Conclusiones

Se observó que una red recurrente o RNN es una arquitectura básica parecida a una red *feed-forward* basada en una regresión lineal definida por una entrada multiplicada por un coeficiente de peso sináptico y una ordenada al origen o factor de sesgo; donde además, se cuenta con un estado oculto y una trayectoria de retroalimentación que permite operar bajo el principio de recurrencia de modo que el resultado de la función de activación de salida se incorpore a la entrada para ser considerada en el siguiente paso de cálculo o *time-step* dando origen a un estado de memoria.

Una limitación importante de las redes recurrentes reside en que, debido al problema de desvanecimiento o explosión del gradiente de la función de pérdida en la etapa de entrenamiento, esta arquitectura presenta una especie de memoria de corto plazo, esto es, que funcionan bien mientras las secuencias de entrada y salida son cortas exhibiendo problemas de aprendizaje y entrenamiento ante secuencias largas.

Actualmente existe una variante de las RNN, redes llamada LSTM (“*long-short termo memory”*- memoria de corto y largo plazo) que les permite no solo reconocer las relaciones entre elementos de secuencias cortas sino también de secuencias largas.

VI. Referencias

Duchi John, Hazand Eland & Singer Yoram, (2011). Adaptive subgradient methods for online learning Stochastic Optimization. Journal of Machine Learning Research.

Fei-Fei et al. (2017). Convolutional Neural Networks (CNNs/ConvNets). URL: http://cs231n. github.io/convolutional networks (Consultado 20-02-2020).

Goodfellow Ian, Bengio Yoshua & Courville Aaron, (2016). Deep learning book in preparation for MIT Press. URL http://www.deeplearningbook.org. (Consultado 25-05-2020).

Graves Alex, (2014). Supervised sequence labelling with Recurrent Neural Network, Arxiv Preprint Arxiv: 1308.0850v5.

Karpathy Andrej, (2018). The Unreasonable Effectiveness of Recurrent Neural Networks. 2015.URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (Consultado 08-04-2020).