MDS7203 - Modelos Generativos Profundos
Profesor: Felipe Tobar
Auxiliares: Camilo Carvajal, Cristobal Alcazar y Roberto Barceló

Proyecto: Pixel RNN y Pixel CNN

Por Diego Dominguez y Daniel Minaya

1. Problema

Dentro del mundo de la inteligencia artificial, los modelos generativos de imágenes desempeñan un rol importante al permitir la creación de contenido visualmente convincente y realista. Entre estos modelos, dos enfoques a destacar son Pixel RNN y Pixel CNN, ambos diseñados para generar imágenes de manera detallada y coherente.
Con estas arquitecturas se busca asignar una probabilidad $p(x)$ a cada imagen x formada por $n \times n$ píxeles, donde las imágenes se representan de manera secuencial como píxeles $x_1, \ldots, x_{n^2}$ de manera iterativa. Se busca estimar la distribución conjunta:

$$p(x) = \prod_{i=1}^{n^2} p(x_i | x_1, \ldots, x_{i-1}).$$

Por lo tanto, la generación de imágenes se lleva a cabo de manera iterativa, donde cada píxel se genera condicionalmente a los píxeles que lo preceden en la secuencia.

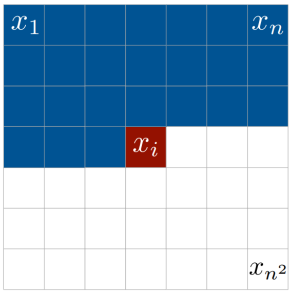


Figura 1: Pixeles de imagen como secuencia.

2. Pixel RNN

Pixel RNN adopta un enfoque secuencial para la generación de imágenes, similar a la generación de texto. La LSTM procesa cada píxel en función de los anteriores, generando así una imagen de manera progresiva donde los estados ocultos se actualizan con cada píxel generado. Esto permite que Pixel RNN sea un modelo potente al lograr capturar las dependencias entre los píxeles, sin embargo, el entrenamiento resulta muy costoso debido a su naturaleza recurrente.

2.1. Row LSTM

La imagen se procesa fila por fila, es decir, cada estado de la LSTM es una fila, y un pixel se genera mediante una convolución 1D.
Para calcular un paso de la LSTM dado el estado oculto $\mathbf{h}_{i-1}$ y el estado de la celda $\mathbf{c}_{i-1}$ se realiza:

$$\begin{aligned} [\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] &= \sigma(\mathbf{K}^{ss} * \mathbf{h}_{i-1} + \mathbf{K}^{is} * \mathbf{x}_i) \\ \mathbf{c}_i &= \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{i}_i \odot \mathbf{g}_i \\ \mathbf{h}_i &= \mathbf{o}_i \odot \tanh(\mathbf{c}_i) \end{aligned} \tag{1}$$

donde $\mathbf{K}^{is}$ y $\mathbf{K}^{ss}$ son las matrices de peso para las transiciones state-to-state y input-to-state, $*$ es la operación de convolución y $\odot$ es el producto de Hadamard.

2.2. Diagonal Bi-LSTM

La imagen se procesa mediante diagonales, es decir, cada estado de la red es una diagonal, y los píxeles se generan mediante una convolución 1D a la imagen torcida. Este proceso se realiza tanto de izquierda a derecha como de derecha a izquierda, y se suman los resultados.

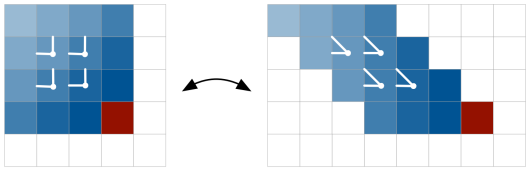


Figura 2: *Skew* y *unskew* de una imagen.

3. Pixel CNN

Pixel CNN emplea capas convolucionales lo que permite procesar todos los píxeles de manera simultánea, mejorando así los tiempos de entrenamiento. Estas capas convolucionales están enmascaradas para evitar que el modelo vea píxeles futuros, de modo que cada pixel depende de sus vecinos que han sido procesados anteriormente.

4. Campo receptivo

Los tres modelos vistos en las secciones anteriores intentan, a su manera, hacer el entrenamiento lo más rápido posible.

Claramente PixelCNN será el ganador debido al uso exclusivo de capas convolucionales, mientras que la diferencia principal entre RowLSTM y DiagLSTM radica en la cantidad de información que se utiliza para generar un píxel dado. En la Figura 3 vemos las transiciones de cada modelo y podemos notar que DiagLSTM hace uso de todo el contexto anterior para predecir el siguiente píxel, mientras que RowLSTM solo captura una región triángular determinada por el tamaño de la convolución 1D que se utilice. Esto último lo aprovecha PixelCNN, dado que al aumentar el tamaño de las convoluciones se logra capturar mayor contexto, sin embargo, al usar exclusivamente capas convolucionales, este proceso es altamente paralelizable por lo que no resulta costoso a diferencia de RowLSTM.

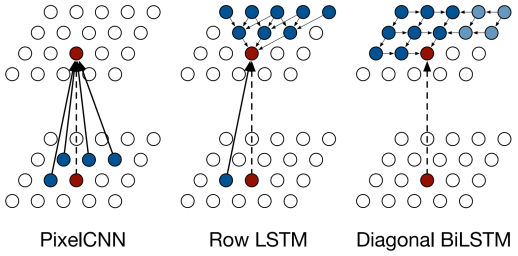


Figura 3: Visualización de las transiciones.

5. Máscaras convolucionales

Se ha visto que todos los modelos usan capas convolucionales de alguna forma, ya sea 1D o 2D. Sin embargo, estas capas no pueden utilizarse directamente, ya que una de las condiciones es que un píxel dado solo dependa de los píxeles anteriores, es decir, no podemos ver los píxeles que vienen después. Para resolver este problema se utilizan máscaras convolucionales.

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

Máscara de tipo A

1	1	1	1	1
1	1	1	1	1
1	1	1	0	0
0	0	0	0	0
0	0	0	0	0

Máscara de tipo B

Estas máscaras, en este caso 2D, anulan los pixeles que se encuentran después del píxel actual. La diferencia entre el tipo de máscara es si consideramos el píxel actual como parte del contexto o no. Al principio se utiliza la máscara de tipo A pues no queremos ver el píxel actual real, mientras que en las etapas posteriores se utilizan máscaras de tipo B, pues ya no es necesario ocultar el píxel pues no es el real.

PixelCNN	**Row LSTM**	**Diagonal BiLSTM**
7×7 conv mask A		
Multiple residual blocks:		
Conv 3×3 mask B	Row LSTM i-s: 3×1 mask B s-s: 3×1 no mask	Diagonal BiLSTM i-s: 1×1 mask B s-s: 1×2 no mask
ReLU followed by 1×1 conv, mask B (2 layers)		
256-way Softmax for each RGB color (Natural images) or Sigmoid (MNIST)		

Figura 4: Arquitectura de los modelos.

6. Experimentos

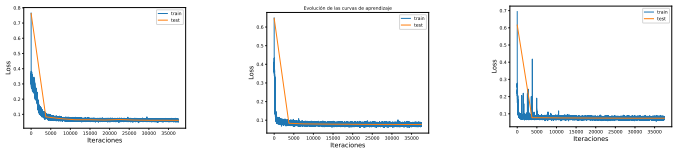
El objetivo de este proyecto es implementar desde cero los modelos de PixelCNN y PixelRNN para luego comparar los costos de inferencia al generar nuevas imágenes. Para esto se usará el datasets de MNIST para entrenar el modelo y generar datos.

Todos los modelos fueron entrenados con una T4 en Google Colab por 10 épocas utilizando el optimizador RMSprop y batches de tamaño 16. Para PixelCNN se utilizó 32 para `hidden_size` y 15 capas convolucionales. Para RowLSTM se utilizó 16 para `hidden_size` y 4 capas LSTM. Para DiagLSTM se utilizó 16 para `hidden_size` y 4 capas LSTM.

Modelo	Tiempo
`PixelCNN`	12 minutos
`RowLSTM`	1 hora y 13 minutos
`DiagLSTM`	2 horas y 50 minutos

Tabla 1: Tiempo de entrenamiento.

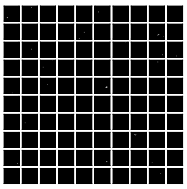
6.1. Función de pérdida



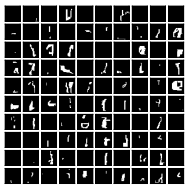
RowLSTM DiagLSTM PixelCNN

La función de pérdida muestra una rápida disminución seguida de un estancamiento en todas las iteraciones de los modelos. Esta tendencia podría atribuirse a la insuficiencia de datos para entrenar eficazmente modelos profundos que incluyen múltiples capas convolucionales y recurrentes, especialmente para los modelos RowLSTM y DiagLSTM.

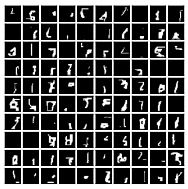
6.2. Datos generados



RowLSTM



DiagLSTM



PixelCNN

Al momento de generar los datos, se inicializa el sampling con una imagen completamente en negro. Por consiguiente, como los modelos no consiguen generar dígitos visibles, se obtienen imágenes con fondo negro y algunos píxeles blancos con un diseño errático. Los números que suelen distinguirse suelen ser el uno, el siete, el ocho y el nueve.

6.3. Tiempos de inferencia

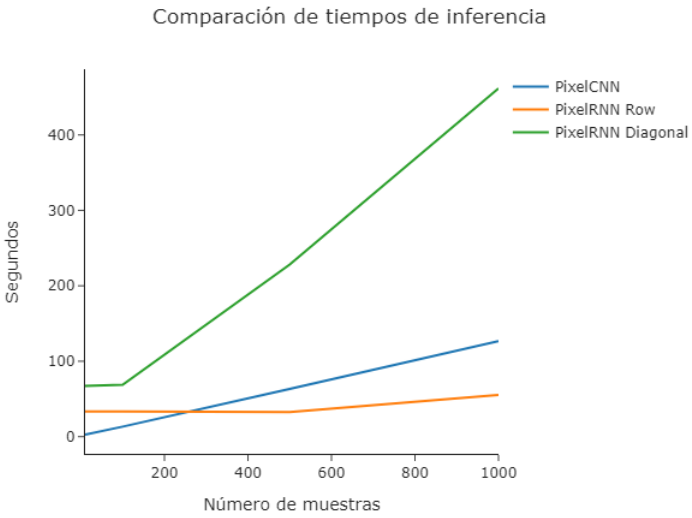


Figura 5: Tiempos de inferencia.

Estos modelos generan imágenes de manera píxel a píxel, independientemente de su arquitectura. El modelo pixelCNN exhibe un comportamiento lineal al incrementar el número de muestras generadas. En contraste, el modelo pixelRNN-Row mantiene un costo similar para cualquier tamaño de N, aunque esto se debe a que prácticamente todas las imágenes resultan mayormente en negro. Por último, el modelo PixelRNN-Diagonal es el más lento en la generación, dada su arquitectura notablemente compleja que involucra diversos procesos de para procesar una imagen.

6.4. Comparación con la literatura

Si bien los resultados no fueron los esperados, podemos hacer la comparación con las referencias. En la Figura 6 y Figura 7 podemos ver los resultados de otros modelos PixelCNN entrenados durante 50 épocas, y en la Figura 8 nuestra implementación de PixelCNN también con 50 épocas. De esto se puede deducir que estos modelos requieren de un número considerable de épocas para obtener resultados decentes.

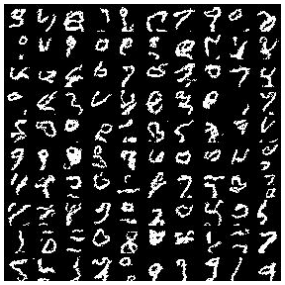


Figura 6: Referencia [5].

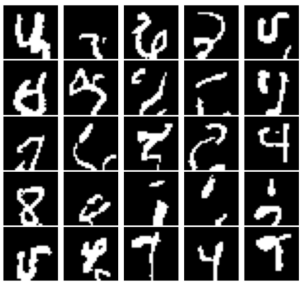


Figura 7: Referencia [6].

Figura 8: Nuestra implementación.

7. Conclusiones

Con este proyecto logramos obtener implementaciones de los modelos PixelRNN y PixelCNN. La observación más obvia es que el entrenamiento de PixelCNN es más rápido que PixelRNN debido a las arquitecturas de los modelos.
Entre los modelos de PixelRNN tenemos que RowLSTM se entrena y hace inferencias más rápido que DiagLSTM, sin embargo, DiagLSTM genera mejores imágenes. Esto puede deberse a que RowLSTM utiliza un contexto menor al de DiagLSTM.
Por otro lado, tenemos que PixelCNN es un punto intermedio entre los modelos anteriores. La generación de imágenes es peor que DiagLSTM pero se generan mucho más rápido. Además, gracias a la arquitectura de PixelCNN el entrenamiento es exponencialmente más rápido por lo que se puede entrenar por más tiempo para obtener mejores resultados como se vió en la sección 6.4. En base a esto, se han desarrollado mejoras al modelo Pixel CNN con el fin de obtener mejores resultados.

Referencias

[1] Aaron van den Oord, Nal Kalchbrenner, Koray Kavukcuoglu, *Pixel Recurrent Neural Networks*, 2016, `https://arxiv.org/abs/1601.06759`, arXiv:1601.06759 [cs.CV].

[2] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu, *Conditional Image Generation with PixelCNN Decoders*, 2016, `https://arxiv.org/abs/1606.05328`, arXiv:1606.05328 [cs.CV].

[3] Tim Salimans, Andrej Karpathy, Xi Chen, Diederik P. Kingma, *PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications*, 2017, `https://arxiv.org/abs/1701.05517`, arXiv:1701.05517 [cs.LG].

[4] axeloh, *PixelCNN*, 2020, `https://github.com/axeloh/pixelcnn-pytorch/`.

[5] carpedm20, *PixelCNN & PixelRNN in TensorFlow*, 2017, `https://github.com/carpedm20/pixel-rnn-tensorflow`.

[6] PacktPublishing, *Hands-On Image Generation with TensorFlow*, 2017, `https://github.com/PacktPublishing/Hands-On-Image-Generation-with-TensorFlow-2.0`.

[7] ardapekis, *pixel-rnn*, 2017, `https://github.com/ardapekis/pixel-rnn`.

[8] heechan95, *PixelRNN*, 2017, `https://github.com/heechan95/PixelRNN-pytorch`.