Titulo del Documento

Lucchini, Francesca

Universidad Católica de Chile
flucchini@uc.cl

Rojos, Felipe

Universidad Católica de Chile
farojos@uc.cl

Sanabria, Pablo

Universidad Católica de Chile
psanabria@uc.cl

July 9, 2018

Abstract

En este paper presentamos dos modelos que resuelven el problema de responder preguntas a partir de un párrafo de texto: FastQA y R-Net. En este documento, explicamos e implementamos ambos modelos, para luego probarlos en el dataset SQuAD1.1.

El rendimiento de nuestros modelos es menor que los declarados en los documentos oficiales. En el caso de FastQA llegamos a un 4% de precisión y en el caso de R-Net 55% en la métrica F1. Debido a eso, también comparamos nuestros resultados del entrenamiento con los obtenidos en los papers de ambas redes para determinar posibles causas en la diferencia de rendimiento.

Al observar las dificultades de la implementación, discutimos posibles cambios que podrían mejorar el rendimiento ya sea asemejándose más a los modelos descritos en sus papers respectivos o ideas propuestas por nosotros, como la integración de embeddings que incluyan el contexto para palabras OOV y la integración de ideas de las *Memory Networks*.

1 Introducción

En este paper nos enfocamos en implementar un modelo que intenta resolver el problema de responder preguntas que de compresión lectora. Implementamos dos modelos posibles R-Net ([2]) y FastQA[3] que evalúan su desempeño en este problema usando el dataset SQuAD1.1 ([9]) de Stanford.

El dataset SQuAD1.1 consiste en preguntas propuestas por voluntarios sobre un conjunto de artículos de Wikipedia que de ahora en adelante llamaremos pasajes. Este dataset contiene ejemplos de QA extractivo, o sea, no provée respuestas fijas, si no que las posibles respuestas son todos los segmentos del pasaje, por lo que un modelo que quiera lograr alto desempeño debe poder manejar este alto número de respuestas. Entonces, como el objetivo del dataset es encontrar los índices del segmento que contienen la respuesta, el modelo va a requerir lograr decidir si cierta información es relevante para responder una pregunta ([9]).

Dado eso, la motivación de este trabajo es lograr implementar un modelo que logre tener buen desempeño en este dataset. Tomando eso en cuenta es que se escogen este par de modelos para lograr atacar el problema. Por un lado, tenemos el modelo R-Net que hasta la fecha (July 9, 2018) está posicionado en el top 25 del ranking del dataset SQuAD1.1 ([10]) y al mismo tiempo, su versión actualizada (r-net *ensemble*) está en el top 5, lo que indica el potencial que tiene este modelo. En este trabajo, nos enfocaremos en la versión simple del modelo que está en la posición 24 del ranking. El modelo consiste en cuatro etapas: (1) Codificación de preguntas y pasajes, (2) *Matching* de la pregunta y el pasaje, (3) *Self-Matching* del pasaje y (4) Output. El aporte principal de la versión simple de R-Net ([2]) es la incorporación de una etapa de *Self-Matching* del pasaje. El objetivo es aumentar la información que contiene del pasaje para mejorar la inferencia de la respuesta.

Por otro lado, tenemos el modelo FastQA que fue implementado en un trabajo anterior (Pregunta 3, Tarea 2). Este modelo consiste en una heurística simple que ayuda a guiar el desarrollo de futuros modelos neuronales que resuelven el problema. En [3] concluyen que este modelo pone en perspectiva los resultados de modelos anteriores de arquitecturas más complejas. A pesar de que no está en un posición alta en el ranking (está en el top 50, [10]), nos pareció importante incluirlo ya que representa un punto de comparación razonable para los otros estudios ([3]).

Este trabajo se organiza como sigue: la sección 2 describe el marco terórico y trabajos relacionados mencionados en los papers de ambos modelos. Después, en la sección 3 se decriben en mayor detalle ambos modelos. En la sección 4 están los detalles de la implementación. En la sección 5 se muestran los experimentos, lo que incluye además de los resultados, el preprocesamiento de los datos y el método de entrenamiento. Luego, en la sección 6 se entabla una discusión sobre los resultados obtenidos, para terminar en la sección 7 y 8 que son conclusiones y trabajo futuro respectivamente [2]. La sección 9 contiene links para poder acceder a un repositorio con nuestro código y un Google Drive con todos los archivos importantes de gran tamaño.

2 Antecedentes

2.1 Marco Teórico

En el caso de R-Net simple, para poder desarrollar las cuatro etapas descritas en 1, se utilizaron modelos desarrollados por otros trabajos. Estos no se consideran parte de trabajos relacionados porque no resuelven el mismo problema que R-Net.

En la primera etapa, para codificar tanto la pregunta como el pasaje se usan RNN para modelar el lenguaje natural, método propuesto en [14]. Los autores de R-Net usan una red bidireccional para el encoding. El trabajo de [14] usa las RNN para modelar data secuencial y predicción de secuencias de datos.

En relación a la siguientes etapas, utilizan modelos de atención basados en RNN basados en los trabajos de [4], [13] y [12]. En el caso de [4] proponen la base de los modelos de atención basados en RNN. Para eso, generan una red recurrente bidireccional y, a partir de la concatenación de los outputs de los estados de ambas direcciones, generan vectores de atención. O sea, sea X_t el input en el tiempo t, entonces los estados ocultos serían $h_t^{\leftarrow}$ y $h_t^{\rightarrow}$. Luego, se concatenan los estados ocultos y con ellos se calculan los vectores de atención. Lo importante de este enfoque es que considera información de las palabras antecesoras y sucesoras del momento t, con énfasis en las palabras cercanas.

En el trabajo de [13], usan y modifican el modelo de atención para obtener la relación entre preguntas y pasajes. La idea general es que se trabajan hipótesis y la premisa como una misma secuencia, separando ambas partes con un símbolo especial "::". Si X^P es el vector de la premisa y X^H es el vector de la pregunta, la secuencia final sería: $X_1^P X_2^P \dots X_n^P :: X_1^H X_2^H \dots X_m^H$. Cada lado de la secuencia es manejado por una LSTM diferente, pero los estados ocultos de la hipótesis recibe como input toda la secuencia de la premisa además de su input normal. Entonces, usando los pesos de atención, se obtiene el vector final h^* que se calcula a partir de una ponderación de la premisa y el vector de salida que se obtiene al procesar premisa e hipótesis. En sus resultados se ve una mejora en el problema de establecer vinculación entre diferentes hipótesis y premisas.

En el trabajo de [12], proponen una mejora a [13]. En vez de calcular el vector h^*, proponen tanto el vector de la premisa, que llaman h^t, como el vector de ateción a para obtener el vector final h^m. Sea H^s el arreglo de estados finales de la hipótesis, sea h^a el vector de la ponderación de la premisa (como fue descrita en el párrafo anterior), sea h^m el vector que representa el matching entre la premisa y la hipótesis y sea a el vector de atención: en la figura 1, se puede ver una comparación de ambas arquitecturas.

En el caso de FastQA, la capa de embedding que se encarga de hacer el mapeo de los tokens a la representación n-dimensional, el autor usa dos enfoques. El primero es hacer un mapeo de cada palabra haciendo uso de una matriz de embedding. El segundo enfoque que usan para el embedding esta basado en el modelo de char-embedding propuesto por [11]. El autor por último menciona que ambos enfoques son combinados para generar el embedding final.

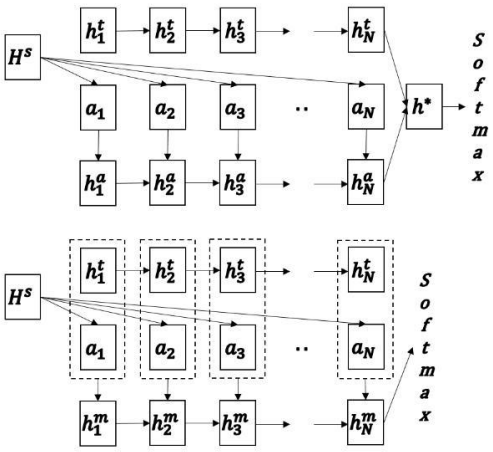

Figure 1: Comparación entre los modelos de [13] y [12]. El modelo de abajo corresponde al de [12]

.

2.2 Trabajos Relacionados

En el ranking de SQuAD1.1 ([10]) hay una lista de los modelos con mejores resultados hasta la fecha en el dataset. De estos modelos se destacan: QANet (*ensemble*) del grupo Google Brain & CMU [1] , Attention-over-Attention Neural Networks (AoANN) de [18] y MARS de [7]; de esta última no hay mucha información disponible.

QANet se diferencia de los modelos antes mencionados en que decide eliminar la naturaleza recurrente de los modelos a la solución en pos de usar convoluciones y mecanismos de auto-atención (self-attentions). O sea, el modelo elimina las RNN de el encoder, resultando en una naturaleza feed-forward completa ([1]). Una de las variantes del modelo está en el primer puesto del ranking.

AoANN son un tipo de red de atención que, como el nombre lo implica, añade un mecanismo de atención sobre un mecanismo previo de atención que actúa a nivel del documento. Usan RNNs (GRU en este caso) para generar los modelos de atención. La atención a nivel de documento obtienen las atenciones individuales de cada palabra de la *query* y luego, para combinarlas, se introduce un segundo mecanismo de atención que decide automáticamente la importancia de cada una de las atenciones individuales [18]. Tanto este modelo como MARS están en la posición 2 del ranking.

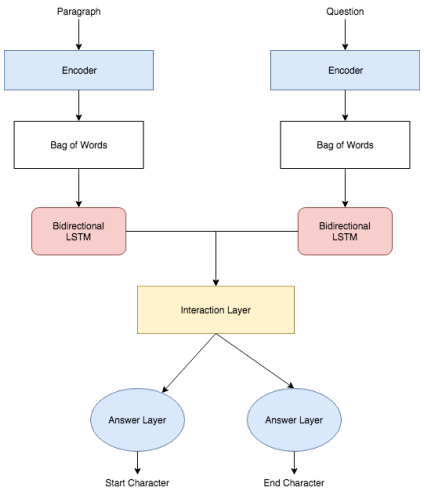

Figure 2: Diagrama del modelo FastQA

3 Modelos

3.1 FastQA

De manera general el modelo propuesto por D. Weissenborn et al. [15] se muestra en la figura 2.

Para modelar los documentos y tener una capa de encoding, el autor propone usar un modelo simple de Bag of Words. Además, el autor usa en todo los casos LSTMs bidireccionales para las capas recurrentes, justificando que las LSTMs bidireccionales mejoran el rendimiento ya que el modelo lograría retener información del presente y del futuro, el cual le permitiría predecir mejor los resultados tal como lo justifica [5].

Después de crear el modelo Inicialmente el autor propone dividir FastQA en dos sub-modelos, uno que permita retener el conocimiento de los párrafos y otro modelo que sea para retener el conocimiento de las preguntas. La estructura de ambos modelos esta dada por:

- Una capa de encoding, que permite obtener un documento en su versión vectorial, donde cada palabra antes de ser ingresada a la red es reemplazada por su respectivo embedding. Además se usa el promedio de embeddings de la vecindad de un span, con largo k, idea similar mencionada en el paper de las Memory Networks ([16]). En este paper, los autores propones ver la vecindad de una palabra no vista(no encontrada en su diccionario) para poder generar un embedding valido. Por otro lado, los autores de fastQA proponen utilizar el embedding de caracteres, que en conjunto con el promedio del embedding de la vecindad, completan la estratégia de encoding en FastQA.

- Una capa recurrente del tipo bidireccional para retener el conocimiento.
- Una capa de interacción que concatena las entradas de ambos modelos, esta capa de concatenación se le pasa a otra capa recurrente del tipo LSTM bidireccional y las salidas se envían a una capa densa que sirve de clasificador, el cual devuelve la posición del primer carácter de la respuesta en el párrafo y la longitud de la respuesta.

Según los resultados obtenidos por el autor, ellos lograron alcanzar un score de 78.9% en F1 y un 70.8% en Exact Match como su mejor resultado obtenido.

3.2 R-Net

Un diagrama que representa el modelo se puede encontrar en la figura 3. En la imágen se muestran las 4 etapas del modelo: (1) Codificación de preguntas y pasajes, (2) *Matching* de la pregunta y el pasaje, (3) *Self-Matching* del pasaje y (4) Output.

La primera etapa es la de codificación. En este paper, usan dos tipos de embeddings: uno a nivel de palabras y otro al nivel de caracteres. Para el primero, se usa GloVe pre-entrenado y sensible a mayúsculas y minúsculas. Para el segundo, se utiliza una RNN bidireccional aplicada a las codificaciones de los caracteres de cada palabra y se obtiene el estado oculto final como resultado. Los autores justifican esta elección mencionando que este método ayuda a manejar palabras OOV (*Out of Vocabulary*). Finalmente, concatenan ambos valores y estos se alimentan a otra RNN bidireccional. Sea e^P y e^Q los vectores de las codificaciones a nivel de palabras y c^P con c^Q los vectores de las codificaciones a nivel de caracteres, se obtienen u^P y u^Q, que son los vectores de las codificaciones del pasaje y la pregunta respectivamente, de la siguiente forma:

$$u_t^Q = BiRNN_Q(u_{t-1}^Q, [e_t^Q, c_t^Q])$$

$$u_t^P = BiRNN_P(u_{t-1}^P, [e_t^P, c_t^P])$$

En las siguientes etapas, proponen soluciones que usan lo que mencionan como *gated attention-based recurrent networks*. Es una variante de las RNN de antención que añade una compuerta al input de la RNN que ayuda a determinar la importancia de acuerdo a la pregunta. Sea W_g una matriz de pesos y sea c el vector de atención sobre toda la pregunta (u^Q), se obtiene el vector de atención de la pregunta g como sigue:

$$g_t = sigmoid(W_g[u_t^P, c_t])$$

Siendo $[u_t^P, c_t]$ la contatenación de estos valores. Luego, se obtiene el input $[u_t^P, c_t]^*$:

$$[u_t^P, c_t]^* = g_t \odot [u_t^P, c_t]$$

Finalmente, $[u_t^P, c_t]^*$ es el input que se le pasa a la RNN junto al estado de la etapa temporal anterior. Esto es lo que se conoce como *gated attention-based recurrent networks* (GABRNN).

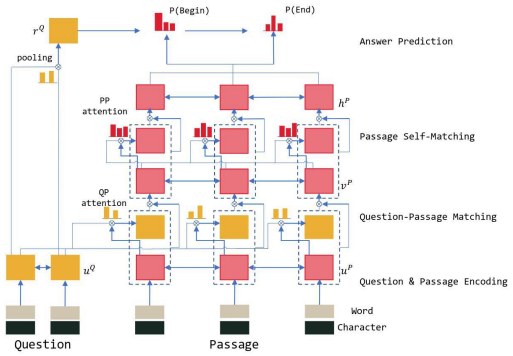

Figure 3: Diagrama del modelo R-Net

En la segunda etapa, usan las GABRNN para incorporar información de la pregunta a la representación del pasaje. Usan las redes propuestas en [12] para aplicar la compuerta de atención g y calculan los vectores de atención usando u^Q, u_t^P y el estado anterior para cada instante t. Sea v^P el vector que representa el output de esta etapa, que representa el pasaje pero con atención en partes relacionadas a la pregunta.

Para la tercera etapa, se necesita extraer evidencia del pasaje que ayude a responder mejor la pregunta. Se propone usar la misma técnica del caso anterior (usar GABRNN), pero los vectores de atención se calculan usando v^P dos veces y v_t^P para cada momento t. El output es el vector h^P que representa la añadidura de la evidencia relevante para cada palabra del pasaje.

Para la cuarta etapa, nuevamente se inspiran en [12] para usar Pointer Networks para predecir los índices de la respuesta. Calculan vectores de atención para los índices que quieren encontrar usando h^P y eligen del vector resultante el de mayor valor como output para la posición de inicio p^1 y de término p^2.

Los resultados mencionados por los autores son un 76% en la métrica Exact Match y 84% en la métrica F1.

4 Implementación

4.1 FastQA

Este modelo fue implementado en Python 3.6 con Keras. Uno de los aspectos principales es que se usaron LTSM para las RNN mencionadas en el paper.

A diferencia de la implementación original, en nuestro trabajo tratamos de reemplazar el embedding con una capa modelada bajo Word2Vec y otra modelada con GloVe[8], esperando que este modelo represente mejor el diccionario de palabras y sea mucho más eficiente con respecto a la memoria . Después de realizar la experimentación y ajustes a los parámetros del modelo, en nuestra implementación solo pudimos llegar a un F1 de 4% aproximadamente, el

cual esta muy lejos incluso del resultado obtenido originalmente por FastQA. Posteriormente probamos el reemplazo de Word2Vec con GloVe pre-entrenado.

Creemos que el problema de nuestra implementación es la capa de embedding y que este podría haberse mejorado o implementado de mejor manera y poder superar al modelo base de FastQA.

4.2 R-Net

Como primer aspecto, nosotros nos basamos en una implementación propuesta en Keras y Python 2.7 ([17]). Nuestro objetivo fue tomar este ejemplo de implementación y hacer dos cosas: pasar todo el código a Python 3.6 y actualizar las funciones de Keras.

Se intentó reproducir el modelo descrito por el paper, pero algunas dificultades se presentaron. En primer lugar, se debió generar la implementación usando la versión 2.0.6 de Keras ya que una de las capas usadas en [17] no pudo ser actualizada a una versión más nueva. Al tratar de hacer el cambio, se mostraba un error de que el tamaño del input no estaba definido. Por esta razón, nuestro código requiere Keras 2.0.6 para correr.

El traspaso de Python 2.7 a Python 3.6 fue esencialmente un cambio de sintaxis asistido por un traductor entre las dos versiones. Pero, hubo un caso que tuvo que ser manejado especialmente. Dado que nuestro código para el preprocesamiento de los datos estaba Python 2.7 y serializaba los resultados con la librería Pickle, nos mostró un error al tratar de deserializar con la versión de la misma librería en Python 3.6. Para resolverlo, fue necesario fijar el encoding a encoding='latin1'.

Para probar el código, contamos con una máquina con 11 Gb de memoria de GPU. Al fijar el tamaño de los vectores de los estados ocultos, en el paper se menciona que usan 75, pero al probarlo de esa forma, el modelo no cupo en la GPU, por lo que lo bajamos a 45, que usaba aproximadamente el 95% de la memoria.

En lo que se refiere al modelo, para todos los casos en que se necesitaba una BiRNN usamos GRU con un wrapper bidireccional de Keras.

4.3 Palabras no codificadas

4.3.1 FastQA

Una de las mayores dificultades que no logramos abordar, fue el embedding de palabras no encontradas. En el paper se utiliza embedding de los caracteres de las palabras, y en adición se ingresa el embedding de su vecindad a una capa convencional.

4.3.2 R-Net

En el paper describen dos tipos de encoding para el pasaje y la pregunta: embedding a nivel de palabras y embedding a nivel de caracteres. Como el embedding

a nivel de palabras se maneja con GloVe, el problema es qué hacer con las palabras OOV (*Out of Vocabulary*). Por esta razón es que se tienen dos tipos de embeddings; si la palabra no está en el diccionario, su embedding a nivel de palabras es un vector de ceros, pero no así el vector a nivel de caracteres. En el paper [2], sección 3.1 mencionan que este método ayuda a manejar este tipo de palabras.

5 Experimentos

5.1 Preprocesamiento

5.1.1 FastQA

Se parsea tanto el contexto como las preguntas, las cuales son utilizadas para entrenar Word2Vec. Posteriormente reemplazamos este entrenamiento con GloVe case-sensitive de 840B de tokens, lo cual no mostró buenas mejoras.

5.1.2 R-Net

Al igual que el paper, usamos Stanford CoreN LP ([6]) para procesar cada pregunta y contexto. Además usamos GloVe case sensitive de 840B de tokens para el embedding a nivel de palabras, los cuales tienen 300 dimensiones, y además usamos char embedding de largo 30 dimensiones. Se divide el dataset de training en 90% para training y 10% para test. Por último, la data es serializada con la librería Pickle de Python.

5.2 Método de Entrenamiento

En ambos casos, entrenamos los modelos usando la versión GPU de TensorFlow.

5.2.1 FastQA

Se entreno por 30 épocas, con 1/4 del dataset de training, usando el valor por defecto para el *learning rate* (0.001), optimizador *Adam* y función de perdida *sparse categorical crossentropy* (a diferencia del paper en el cual usan *softmax-crossentropy*). Por último, cada época en promedio demoró 5 minutos aproximadamente.

5.2.2 R-Net

Se planificó un entrenamiento de 60 épocas, con todo el dataset de training, usando un *learning rate* de 1, optimizador *AdaDelta* y función de pérdida *categorical crossentropy*). El entrenamiento efectivo fue de 42 épocas, ya que en ese momento el modelo comenzó a disminuir muy poco la función de pérdida. Por último, cada época en promedio demoró 5 minutos aproximadamente.

Modelo	Exact Match	F1
QANet (ensemble)	83.877	89.737
Hybrid AoA Reader (ensemble)	82.482	89.281
MARS (single model)	82.587	88.880
r-net (single model)	76.461	84.265
FastQA	68.436	77.070
R-Net (nuestro)	44.947	54.530
FastQA (nuestro)	0.302	2.206

Table 1: Resultados para los modelos mencionados

5.3 Resultados

En la tabla 1, se pueden observar las comparaciones entre los modelos mencionados, incluyendo los resultados originales para FastQA y R-Net; en las últimas dos filas están los resultados de nuestros modelos. Se puede observar, especialmente en el caso de FastQA, que nuestra implementación tiene una gran diferencia con los resultados de los autores.

En la figura 5, se observan los resultados de la función de pérdida en el entrenamiento de FastQA y en la figura 6 para R-Net. Cada gráfico muestra la función de pérdida general, para el índice de inicio de la respuesta y para el índice de término.

En la figura 5, se aprecia que la función de pérdida de FastQA se mantuvo en valores altos (mayores a 8), mientras que en la figura 6, llegó a valores mucho menores. También, se puede ver que el aprendizaje de R-Net se estanca en las 42 épocas.

Por último, en la figura 4, está graficado el desempeño de R-Net versus las épocas de entrenamiento.

6 Discusión

En ambos modelos sucedió que no se logró llegar a los resultados presetados por los autores de los papers. Para ambos casos realizaremos un análisis de las causas posibles. Para el caso de R-Net, hay dos causas principales que pudimos identificar. La primera y más directa, es que no usamos el tamaño del vector de los estados ocultos mencionado en el paper (usamos 45 en vez de 75) por problemas de espacio en la memoria RAM de la GPU. La segunda es que, como se muestra en la figura 6, el aprendizaje disminuye en las 42 épocas. Dado esto, podríamos haber probado modificando el *learning rate* para ver si mejoraba el aprendizaje. Para el caso de FastQA, la razón principal es que no logramos implementar la codificación mencionada en el paper, lo que afecta enormemente los resultados.

Es importante comparar a ambos modelos a nivel de implementación, entrenamiento y desempeño. Si nos fijamos en la implementación, con R-Net

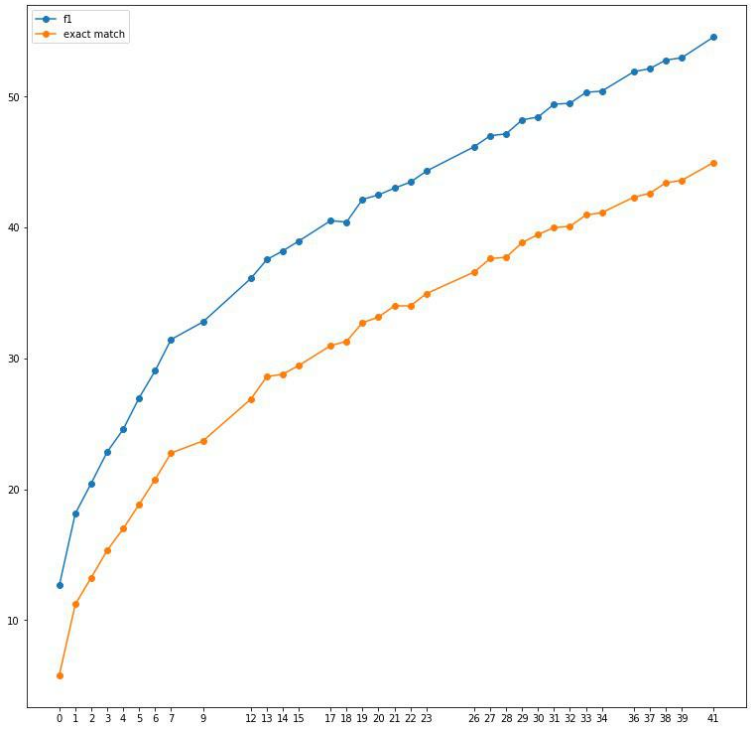

Figure 4: Gráfico del desempeño de R-Net

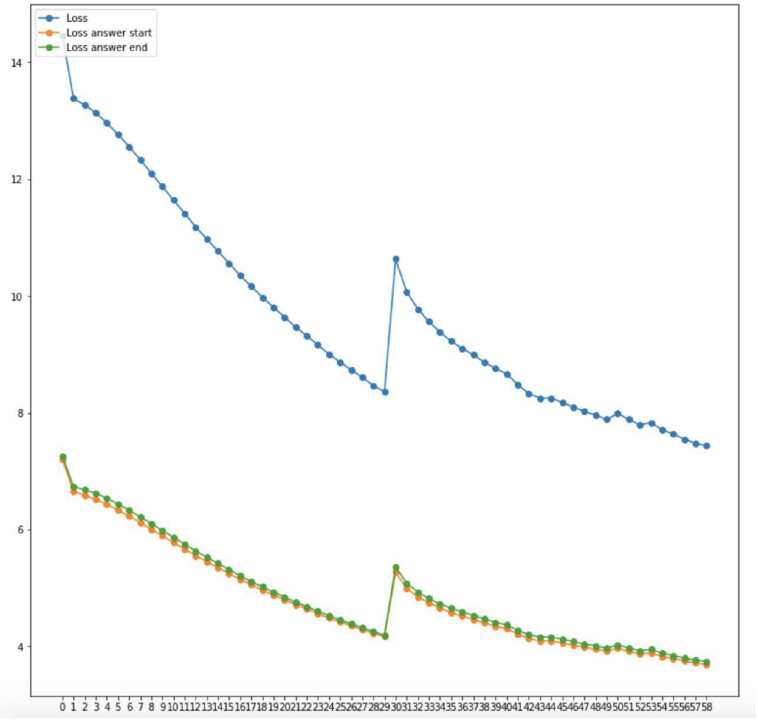

Figure 5: Gráfico de la pérdida de FastQA

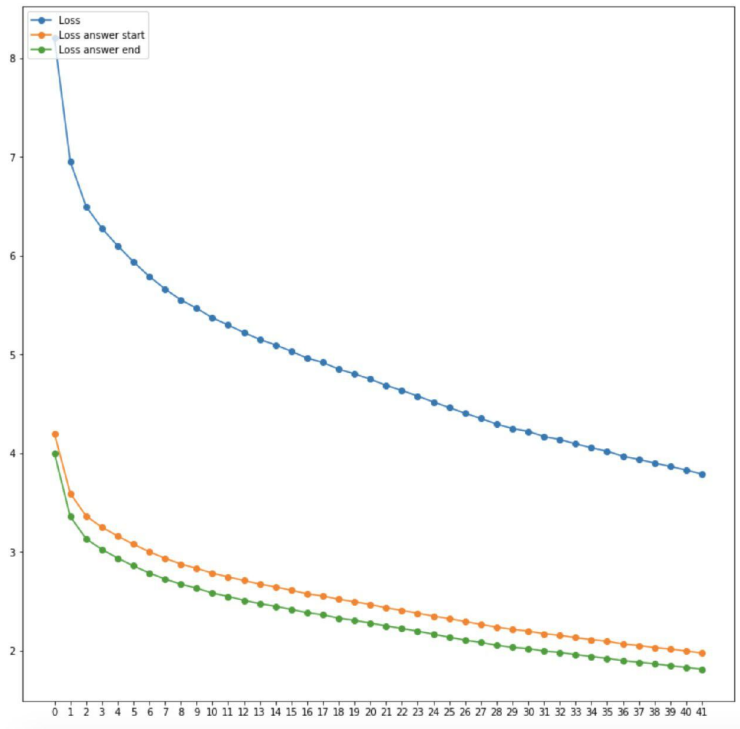

Figure 6: Gráfico de la pérdida de R-Net

obtenemos mejores resultados que con la implementación de FastQA. De hecho, la implementación de R-Net, en comparación con FastQA fue más sencilla de lograr. En FastQA nos encontramos con muchos problemas para comprender cómo lograr el embedding propuesto en el paper. Además, la información disponible en internet sobre FastQA es pobre en comparación con R-Net. Pero, la implementación de R-Net no estuvo desprovista de contratiempos. De hecho, fue necesario usar una versión específica de Keras y, al momento de usar el código de preprocesamiento de los datos provisto en [17], nos encontramos con la dificultad de utilizar esos datos en nuestro modelo en Python 3.

En términos de entrenamiento, FastQA demoraba un 60% del tiempo que toma entrenar una época de R-Net. Esto es curioso especialmente debido al número de parámetros que tiene cada red. Por un lado, nuestra implementación de FastQA tiene 42,763,989 parámetros y la de R-Net, tiene 632,295. Una de las causas posibles para esto es que el modelo de R-Net está compuesto por 4 etapas que son entrenadas end-to-end, lo que aumenta la porción secuencial de la actualización de los pesos. Igualmente, es algo que requiere un análisis profundo para detectar la causa.

Por último, si observamos el desempeño obtenido en ambos modelos, se destacan los resultados obtenidos con R-Net por sobre FastQA.

Durante la ejecución del proyecto, usamos modelos que usaban dos tipos distintos de unidades recurrentes. R-Net usó GRU mientras que FastQA usó LSTM. En [2] mencionan que el rendimiento de GRU es similar a LSTM y que es más eficiente de entrenar. Dado eso, es natural preguntarse si la implementación de FastQA podría ser optimizada usando GRU en vez de LSTM.

Un desafío importante en ambos casos, fue la codificación de los pasajes y preguntas. Por un lado, el embedding propuesto en FastQA es complejo y no pudo lograrse en la implementación, pero tiene buen manejo de palabras OOV que considera el contexto (vecindad) de estas palabras. Por otro lado, el embedding de R-Net tiene una propuesta simple que concatena dos tipos de embedding para que las palabras OOV no sean solamente vectores de ceros. El problema de esta implementación es que no considera el contexto. Por eso, una posible mejora es tomar en cuenta esto para futuras implementaciones.

7 Conclusiones

En este trabajo comparamos dos modelos que resuelven el problema de respuesta a preguntas usando habilidades de compresión lectora. Para ambos casos usamos el dataset SQuAD1.1 para evaluar su desempeño. Uno de los modelos contaba con una codificación compleja que no pudimos replicar, por lo que sus resultados son poco satisfactorios. En ambos casos no pudimos replicar los puntajes presentados en los papers, pero en el caso de R-Net logramos resultados razonables.

Al analizar y discutir los resultados e implementación, notamos varias logramos detectar varias aristas de posibles mejoras para ambos modelos. Dentro de estas, las más importantes son el manejo del embedding (FastQA) y el manejo

de la memoria usada por el modelo (R-Net). También, concluimos que en el caso de R-Net necesitaríamos hacer un refactoring para poder lograr entrenarla más rápido, o sea, probablemente hay espacio para optimizar el modelo. Finalmente, para la implementación de FastQA queremos destacar que en la primera implementación de FastQA reemplazar el embedding con algo ya pre-entrenado o usando un modelo más simple emperoó el rendimiento. Gracias a eso, se pudo comprender la clave de FastQA esá en el embedding, esto es lo que hace que el modelo aprenda. Al haber quitado esa parte en nuestra implementación, el modelo real no aprendió, pero nos dimos cuenta que hacer el embedding puede ser incluso más importante que el propio modelo.

Una opinión que compartimos es que para este tipo de trabajos no suelen compartir las implementaciones de sus modelos o no suelen ser lo suficientemente precisos en su descripción técnica para poder replicarlos. Esta tendencia nos parece negativa porque muestra un interés por tener una buena posición en el ranking, pero no para el avance de la investigación en general.

8 Trabajo Futuro

En esta sección, presentamos ideas propias que podrían ser interesantes para mejorar a ambos modelos fuera de las especificaciones de los papers. Para el caso de R-Net, sería interesante incorporar un embedding que considere el contexto de las palabras, algo que puede ayudar con las palabras OOV. También, sería interesante probar combinar estos modelos con técnicas de Memory Networks.

9 Links

1. Github:
   `https://github.com/FranLucchini/IIC3697_RNet`
2. Google Drive:
   `https://drive.google.com/open?id=15FuLZdRHILJWXvYqd2936SgGvCiJXj1W`

References

[1] Minh-Thang Luong Adams Wei Yu, David Dohan. Qanet: Combining local convolution with global self-attention for reading comprehension. 2018.

[2] Microsoft Research Asia. R-net: Machine reading comprehension with self-matching networks. 2017.

[3] Laura Seiffe Dirk Weissenborn, Georg Wiese. Making neural qa as simple as possible but not simpler. 2017.

[4] Yoshua Bengio Dzmitry Bahdanau, KyungHyun Cho. Neural machine translation by jointly learning to align and translate. 2014.

[5] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[6] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[7] YUANFUDAO Research NLP. Mars: `https://posts.careerengine.us/p/5ad8637755ba6b136fc793ad`. 2018.

[8] Jeffrey Pennington, Richard Socher, and Christopher D Manning. GloVe: Global Vectors for Word Representation.

[9] Konstantin Lopyrev-Percy Liang Pranav Rajpurkar, Ian Zhang. Squad: 100,000+ questions for machine comprehension of text. 2016.

[10] Stanford University Pranav Rajpurkar. The stanford question answering dataset `https://rajpurkar.github.io/SQuAD-explorer`.

[11] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[12] Jing Jiang Shuohang Wang. Learning natural language inference with lstm. 2016.

[13] Karl Moritz Hermann Tomáš Kočiský Phil Blunsom Tim Rocktäschel, Edward Grefenstette. Reasoning about entailment with neural attention. 2015.

[14] Lukáš Burget Jan Černocký Sanjeev Khudanpur Tomáš Mikolov, Martin Karafiát. Recurrent neural network based language model. 2010.

[15] Dirk Weissenborn, Georg Wiese, and Laura Seiffe. Making neural qa as simple as possible but not simpler. *arXiv preprint arXiv:1703.04816*, 2017.

[16] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory Networks. pages 1–15, oct 2014.

[17] YerevaNN. Challenges of reproducing r-net neural network using keras `https://goo.gl/JxHCZp`.

[18] Si Wei Shijin Wang Ting Liu Guoping Hu Yiming Cui, Zhipeng Chen. Attention-over-attention neural networks for reading comprehension. 2017.