PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
IIC-3697 APRENDIZAJE PROFUNDO

TAREA 2: REDES NEURONALES RECURRENTES (RNNS)

Fecha Maxima de Entrega: Lunes 7 de Junio, 23:59 (no habrá aplazamientos)

Objetivo

Estimad@s, en esta tarea tendrán la oportunidad de poner en práctica sus conocimientos sobre redes neuronales recurrentes (RNNs).

1 Redes Neuronales Recurrentes: RNNs (40%)

Como comentamos en clases, las redes recurrentes permiten modelar secuencias, es decir, capturar relaciones de orden en los datos, por ejemplo temporal o espacial. En esta tarea usaremos la librería *PyTorch* para nuestra implementación.

Considere la red recurrente representada en la figura 1.

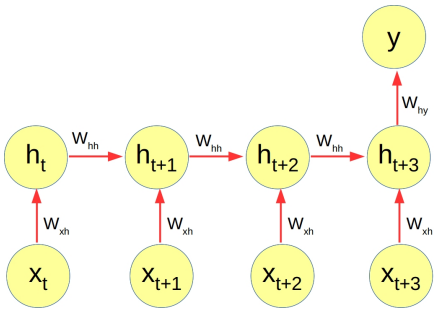

Figura 1: Arquitectura de red recurrente utilizada como ejemplo base.

El siguiente código nos permite implementar la RNN de la figura 1 usando *PyTorch*.

```
import torch
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_dim, hidden_dim, num_layers, batch_first=True)
        self.linear_out = nn.Linear(hidden_dim, 1)

    # This method defines the forward pass of the RNN
    def forward(self, input):
        batch_size, _ = input.size()

        # Initializing hidden state for first input
        h0 = self.init_hidden(batch_size)
        # Passing in the input and hidden state to obtain output
        _, hidden_state = self.rnn(input.unsqueeze(2), h0)
        out = self.linear_out(hidden_state.squeeze())
        return out

    # This method generates the first hidden state of zeros for the forward pass
    # This creates a tensor of zeros in the shape of our hidden states.
```

```
    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.num_layers, batch_size, self.hidden_dim)
        return hidden
```

El primer paso es crear una clase `RNN` que hereda de la clase base `nn.module` de PyTorch. La creación de nuestra clase RNN recibe como parámetros: i) el tamaño del input `(input_size)`, ii) el tamaño del estado oculto de la red recurrente `(hidden_size)`, y iii) la cantidad de capas de la red recurrente `(num_layers)`. En el método constructor de la clase se crea una instancia de `nn.RNN`, que recibe como parámetros: i) el tamaño de los tensores de entrada `(input_size)`, ii) tamaño del estado oculto `(hidden_size)`, y iii) la cantidad de capas recurrentes `(num_layers)`. Adicionalmente, se crea una capa lineal para producir la salida. Esta capa de salida recibe como parámetro: i) la dimensión de entrada `hidden_dim` y ii) la dimensión de salida, en este caso 1 y sin función de activación no-lineal.

Adicionalmente, es necesario definir el método `forward`, que es usado para definir las opciones de la alimentación secuencial de la red. En este caso, indicamos que la red recibe como entrada dos tensores: `input` y `initial_hidden_state`. El primero corresponde a las *features* de entrada y el segundo al estado inicial del estado oculto. Este segundo parámetro es opcional, y si no se entrega se inicializan los estados ocultos con valores iguales a cero. El resultado de invocar a `self.rnn` corresponde a dos tensores: `output` y `hidden_state`. El primero corresponde a todos los estados ocultos obtenidos luego de procesar la secuencia, mientras que el segundo solo contiene los últimos estados ocultos. Finalmente, se define la salida lineal, como el modelo es del tipo *many–to–one*, solo se pasan los últimos estados ocultos de la RNN por la capa lineal para obtener la salida deseada.

Luego, ya podemos usar el modelo anterior, así como definir hiperparámetros, función de pérdida, optimizador y comenzar entrenamiento. Más detalles de la implementación de RNN en PyTorch se darán en la sesión de ayudantía de la tarea.

```
model = RNN(input_size, output_size, hidden_dim, n_layers)

# Define hyperparameters: number of epochs and learning rate
n_epochs = 100
lr=0.01

# Define device, in this case CPU
device = torch.device("cpu")
model.to(device)

# Define Loss, Optimizer, ex: cross entropy and Adam.
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# Run training
for epoch in range(1, n_epochs + 1):
    for input_seq in batches:
        # Reset gradients from previous epoch
        optimizer.zero_grad()

        # run model on inputs
        input_seq.to(device)
        output, hidden = model(here-input-seq)

        # compute loss, run backprop, and update weights
        loss = criterion(output, here-output-seq.view(-1).long())
        loss.backward()
        optimizer.step()
```

En esta parte de la tarea aplicaremos redes recurrentes para clasificar textos. En particular, trabajaremos con el set de datos disponible en: SMS-Dataset. Este set de datos contiene 5000 SMS etiquetados según el rótulo spam o no-spam. Para el problema de clasificación de una secuencia de texto el primer paso será transformar cada oración de entrada a un espacio de característica (feature space o embedding). Para esto deberán crear una capa de *embedding* que transforme las oraciones de entrada a un vector de largo fijo. Esta capa puede operar directamente sobre toda la oración de entrada, o puede operar sobre cada una de las palabras y luego generar un vector integrado mediante un proceso de pooling (ex. sumar los embedding de las palabras de la oración). Averigue sobre posibles codificaciones (embeddings) de oraciones y palabras, y seleccione una de ellas para su tarea.

En su implementación deberán considerar los largos de secuencia de los textos de entrada. Esto pues algunos son más extensos que otros pero su modelo espera *batches* en formato tensorial, lo que implica que las oraciones deben ser llevadas a un largo fijo. Para resolver este problema recomendamos usar la función PadSequence, aunque pueden obtener mejoras en el tiempo de entrenamiento e inferencia si usan la abstracción PackedSequence.

Para la estructura de su modelo use una configuración similar a la figura 1, i.e., sólo una capa oculta para la red recurrente y un clasificador conectado al estado final de la recurrencia. Para el estado oculto de la red recurrente utilice 80 dimensiones. Para el clasificador utilice una capa densa de 60 neuronas y activación soft-max.

Actividad 1

Explique brevemente la técnica seleccionada para el embedding inicial de las oraciones del set de entrada.

Actividad 2

Implemente el modelo indicado usando una RNN simple:

- ¿Depende el número de parámetros del *dataset* utilizado?
- ¿Cómo se podría reducir los parámetros del modelo sin reducir la dimensionalidad del estado interno de la red?

Puede verificar su cálculo ejecutando la siguiente función:

```
def num_trainable_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

Se decide modificar la dimensionalidad del estado oculto de la red recurrente a un valor 120. ¿Cómo se ve afectado el número de parámetros? Pruebe y fundamente su respuesta.

Actividad 3

Entrene su modelo:

- Reporte gráficamente cómo varían la precisión y la función de pérdida conforme avanza el set entrenamiento.
- Reporte la precisión obtenida en el set de *test*.

Actividad 4

La red anterior implementa una red recurrente tradicional. Como comentamos en clase, el uso de una RNN tradicional presenta limitaciones para ajustar los parámetros de la red utilizando métodos de descenso de gradiente (problema de desvanecimiento o explosión de gradientes *vanishing/exploiting gradient problems*). Redes recurrentes tipo LSTM o GRU ofrecen soluciones más estables. PyTorch ofrece implementaciones de ambas alternativas, siendo muy simple su uso.

- Compare el rendimiento de un modelo LSTM con respecto al modelo tradicional de RNN. Considere el rendimiento en términos de calidad de predicción y también el número de parámetros utilizados. Comente brevemente sus resultados.

Actividad 5

Considerando el mejor modelo de las actividades anteriores. Pruebe las siguientes variantes:

- Pruebe agregando una segunda capa de red recurrente. Comente brevemente sobre los resultados obtenidos y la velocidad de convergencia.
- Pruebe utilizando una RNN bidireccional de 1 capa de profundidad, varíe la dimensión de los *embeddings* y de la capa lineal en ambas direcciones. Comente brevemente sobre los resultados obtenidos y la velocidad de convergencia.

2 Red Siamesa para Aprendizaje de Similitud entre Oraciones (60%)

En esta parte de la tarea trabajaremos con redes siamesas. Este tipo de arquitectura se utiliza comúnmente para aprender una métrica de distancia semántica entre las entidades de entrada al modelo. A modo de ejemplo, la figura 2 muestra el caso de una red siamesa que es utilizada para aprender una métrica de similitud entre las 2 imágenes de entrada.

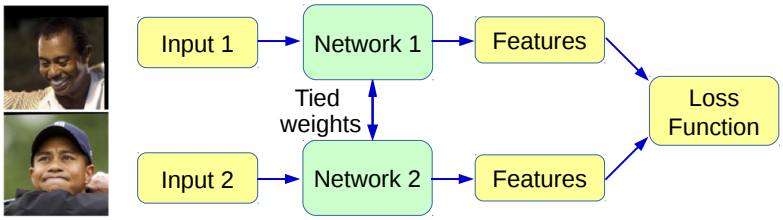

Figura 2: Arquitectura de una red siamesa para aprender similitud entre imágenes.

En la figura 2, las redes 1 y 2 ("Network 1" y "Network 2") utilizan el mismo set de pesos (tied weights). Además, a la salida de estas redes, el bloque "Features" corresponde al vector de característica de la última capa de las redes 1 y 2. Este vector de características es utilizado para calcular la función de pérdida.

Durante el aprendizaje, la función de pérdida tiene como objetivo acercar vectores de features correspondientes a imágenes con similitud semántica y alejar vectores sin este tipo de similitud. Las siguientes ecuaciones muestran dos funciones de pérdida que permiten logran este objetivo: i) Pérdida ranking de pares (pairwise ranking loss) y ii) Pérdida ranking de triples (triplet ranking loss), más detalle sobre esto en clases.

Pairwise Ranking Loss:

$$L(f(I_1), f(I_2), y) = y \left\| f(I_1) - f(I_2) \right\| + (1-y) max \left\{0, m - \left\| f(I_1) - f(I_2) \right\| \right\}$$

$y \in [0,1]$; $y = 1$ si las imágenes son de la misma clase, $y = 0$ en caso contrario; $m > 0$ es una constante.
$\|f(I_1) - f(I_2)\|$: distancia Euclideana.

Triplet Ranking Loss:

$$L(f(I_1), f(I_2)^{I_1}, f(I_3)^{\neg I_1}, y) = max \left\{0, m - \left\{ \left\| f(I_1) - f(I_3)^{\neg I_1} \right\| - \left\| f(I_1) - f(I_2)^{I_1} \right\| \right\} \right\}$$

$f(I_2)^{I_1}$: instancia I_2 es de la misma clase que I_1.
$f(I_3)^{\neg I_1}$: instancia I_3 no es de la misma clase que I_1.

Para el caso de aprender una distancia entre oraciones, las redes 1 y 2 serán del tipo LSTM bidireccionales. Para el entrenamiento usarán el set de datos SICK, disponible en: SICK-Dataset. Como guía puede consultar el modelo presentado en [2].

Para implementar el modelo, el primer paso será llevar las oraciones de entrada a un espacio de características (embedding), en este caso usarán la codificación Word2Vec [4], según lo siguiente:

- Word2Vec consiste de una red neuronal pre-entrenada con una gran cantidad de textos, que permite codificar cada palabra del idioma inglés a un espacio de características, típicamente de 300 dimensiones, de ahí su nombre Word-To-Vector o simplemente word2vec.
- Para codificar una frase, cada palabra en la frase de entrada se transforma a su codificación word2vec de 300D aplicando el modelo correspondiente. Luego, la codificación de la frase completa se calcula como el vector promedio (average pooling) de las codificaciones de sus palabras.
- Para utilizar la codificación word2vec pueden usar librerías como *Gensim* y *Pytorch-NLP*.

2.1 Modelo

Revise el dataset SICK y también su uso para el modelo presentado en [2].

- ¿Qué significan los rótulos de cada oración para este set de datos y como fueron obtenidos?
- ¿Cuál es el tamaño del dataset total?, ¿Cuál es el tamaño de la partición utilizada para entrenamiento y test?. En su tarea utilice también estas particiones.

2.2 Modelo

El siguiente código muestra una posible estructura que puede utilzar como base para construir su modelo. Más detalles de esta implementación se darán en la sesión de ayudantía de la tarea.

```
class Siamese(nn.Module):
    def __init__(self, embeddings_table, embeddings_size_1,
    embeddings_size_2,hidden_size_1,hidden_size_2):
        super().__init__():
        self.embeddings_table = word2vec
        self.embeddings_size_1 = embeddings_size_1
        self.hidden_size_1 = hidden_size1_1
        self.embeddings_size_2 = embeddings_size_2
        self.hidden_size_2 = hidden_size_2
        self.lstm_1 = nn.LSTM(input_size=embeddings_size_1, hidden_size=
            hidden_size_1, bidirectional=True)
        self.lstm_2 = nn.LSTM(input_size=embeddings_size_2, hidden_size=
            hidden_size_2, bidirectional=True)

    def forward(self, src_sentences, src_lengths):
        # ver comentarios siguiente secci\'on para completar
        # TODO: calcule embeddings usando su implementaci\'on de word2vec

        # TODO: ejecute self.lstm sobre los embeddings de las oraciones


        return all_hidden_states, last_hidden_states, last_cell_states
```

Actividad 6

Complete el código anterior en las partes indicadas con el comentario: `[TODO]`. Para la función de pérdida seleccione alguna de las 2 métricas indicadas anteriormente.

- Comente brevemente sus decisiones de diseño del modelo.
- ¿Cuántos parámetros entrenables tiene el modelo resultante?

Actividad 7

Entrene su modelo utilizando el dataset SICK:

- Reporte gráficamente cómo varían la precisión y la función de pérdida conforme avanza el set entrenamiento.
- Reporte la precisión obtenida en el set de test.

2.3 Aumento de datos

Como hemos visto en clases, los modelos de aprendizaje profundo necesitan de grandes cantidades de datos rotulados para alcanzar un buen rendimiento. Como vimos en clases, el uso de técnicas de aumento de datos (data augmentation) sirve para aliviar este problema. Averigue sobre técnicas de aumento de datos usadas comúnmente para el caso de textos. Por ejemplo:

- Reemplazar palabras frecuentes en forma aleatoria por sinónimos.
- Reemplazar palabras poco frecuentes en forma aleatoria por un antónimo.
- Traducir a otro idioma y traducir de vuelta.

Seleccione la técnica que desee y apliquela a los datos de SICK, luego entrene su modelo con el set resultante.

Actividad 8

- Explique brevemente la técnica seleccionada (no tiene por que ser una de las mencionadas) y como la aplicó en su tarea. Indique además el tamaño del set de entrenamiento resultante.
- Reporte gráficamente cómo varían la precisión y la función de pérdida conforme avanza el entrenamiento. Compare el rendimiento en el set de test con respecto al obtenido en la actividad anterior.

2.4 Visualización

Selecciones un subconjunto de sus datos de entrenamiento, ej. 30% de los datos. Utilizando su mejor modelo, calcule el feature vector (salida del modelo) para cada una de las oraciones del subconjunto seleccionado. Aplique a los features resultante la técnica t-SNE para reducir la dimensionalidad de cada vector a un espacio 2D. Existen diversas librerías de Python que incorporan T-SNE.

- Grafique los puntos resultantes y analice cualitativamente los embedding resultantes. Comente brevemente sobre la coherencia semántica del espacio resultante.
- ¿Observa algún(os) cluster(s) significativo(s)?

Consideraciones y formato de entrega

La tarea deberá ser entregada via SIDING en un cuestionario que se habilitará oportunamente. Se deberá desarrollar la tarea en un Jupyter Notebook con todas las celdas ejecutadas, es decir, no se debe borrar el resultado de las celdas antes de entregar. Si las celdas se encuentran vacías, se asumirá que la celda no fue ejecutada. Es importante que todas las actividades tengan respuestas explícitas, es decir, no basta con el *output* de una celda para responder.

Bibliografía

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[2] J. Mueller, A. Thyagarajan. Siamese Recurrent Architectures for Learning Sentence Similarity. In *AAAI-16* 2016

[3] Efficient Estimation of Word Representations in Vector Space. In *arXiv:1301.3781* 2013

[4] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean. Efficient Estimation of Word Representations in Vector Space. In *arXiv:1310.4546* 2013