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

TAREA 2: REDES NEURONALES RECURRENTES (RNNS)

Fecha de Entrega: 9 de Junio

Objetivo

Estimad@s, en esta tarea tendrán la oportunidad de poner en práctica sus conocimientos sobre redes neuronales recurrentes (RNNs) y modelos de secuencia a secuencia (Seq2Seq).

1 Redes Neuronales Recurrentes: RNNs (50%)

Como comentamos en clases, las redes recurrentes permiten modelar secuencias, es decir, capturar relaciones temporales o espaciales en los datos. En esta tarea usaremos la librería *PyTorch* para nuestra implementación.

Considere la red recurrente representada en la figura 1.

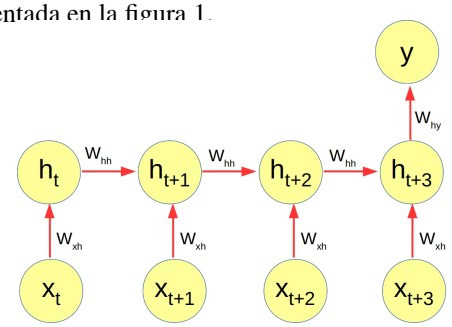

Figura 1: Arquitectura de red recurrente utilizada como ejemplo base.

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()
```

Para esta tarea aplicaremos redes recurrentes a la tarea de detección de sentimiento en texto. Esta tarea consiste en predecir la emoción principal detrás del texto de un comentario. Por ejemplo, predecir si el texto de un comentario en redes sociales es positivo o negativo respecto de alguna situación. En nuestro caso, la tarea consistirá en predecir la emoción detrás del texto de una crítica sobre una película. En particular para la primera parte de esta tarea trabajaremos con el Large Movie Review Dataset que contiene 50.000 comentarios que usuarios han hecho sobre películas, indicando además si cada comentario es positivo o negativo.

Para el problema de clasificación de una secuencia de texto el primer paso será transformar cada palabra a un espacio de característica (feature space o embedding). Para esto deberá crear una capa de *embedding* que realice el mapeo de cada palabra del vocabulario de entrada a un vector de largo fijo. Puede experimentar con embedding como Word2Vec, Glove, o One-Hot, entre otros. Adicionalmente, para aumentar el rendimiento de su modelo, se recomienda realizar un saneo de datos eliminando palabras poco frecuentes y errores ortográficos.

Para su implementación deberá considerar los largos de secuencia de los textos, puesto que hay algunos 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 pad_sequence, aunque pueden obtener mejoras en el tiempo de entrenamiento e inferencia si usan la abstracción PackedSequence.

Para la parte recurrente del modelo utilice dimensión interna de 100 y sola una capa oculta. La salida de la red deberá indicar la probabilidad que el modelo asigna a la clase positiva.

Actividad 1

Considere el modelo resultante y su número de parámetros: ¿depende el número de parámetros del *dataset* que se utilice? ¿cómo se pueden 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 intermedio a un valor 50. ¿Cómo se ve afectado el número de parámetros? Pruebe y fundamente su respuesta.

Actividad 2

Entrene su modelo y calcule la precisión obtenida en el set de *test*. Reporte sus resultados.

Luego pruebe las siguientes variantes de su modelo:

- Prube utilizando una RNN bidireccional, varíe la dimensión de los *embeddings* y de la capa lineal en ambas direcciones.
- Agregue una nueva capa densa de 100 unidades antes de la capa lineal de salida (utilice la dimensionalidad que mejor resulte en el ítem anterior).
- Comente sobre los resultados obtenidos y la velocidad de convergencia.

Actividad 3

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.

2 Modelos de Secuencia a Secuencia (Seq2Seq)

En esta parte de la tarea deberán implementar un modelo recurrente del tipo Seq2Seq, el cual en una segunda etapa será enriquecido por un mecanismo de atención. Para esto nuevamente usarán *PyTorch*.

Como vimos en clase, el modelo Seq2Seq se descompone en dos módulos: un *encoder*, que códifica la secuencia de entrada, para lo cual usaremos una red LSTM bidireccional; y un *decoder*, que genera la secuencia de salida, para lo cual usaremos también una red LSTM bidireccional.

Para las siguientes actividades trabajaremos con el dataset SCAN. Este contiene secuencias de entrada en texto plano que deben ser traducidas a secuencias de acciones atómicas (*primitives*). Por ejemplo, la entrada *jump left* debe ser traducida a la secuencia de dos acciones *LTURN JUMP*, ver [3] para más detalles. Dependiento de las capacidades de generalización que sean evaluadas, existen diversas opciones para el set de test. En esta tarea utilizaremos el set de test: *"simple train-test split"*.

2.1 Codificador

Para codificar las instrucciones de entrada aprovecharemos funciones de la primera parte de la tarea. En particular, la función *embeddings* para codificar la entrada y la función *padding* para dejar las secuencias de igual largo. Para esta parte de la tarea el codificador debe retornar todos los estados intermedios $h_i = LSTM(x_i, h_{i-1}, , c_{i-1})$, es

decir, el estado intermedio correspondiente a cada palabra de la entrada. También deberá retornar el estado oculto y el estado de celda final (h_{last}, c_{last}).

```
class EncoderModule(nn.Module):
    def __init__(self, embeddings_table, embeddings_size, hidden_size):
        super().__init__()

        self.embeddings_table = embeddings_table
        self.embeddings_size = embeddings_size
        self.hidden_size = hidden_size

        self.lstm = nn.LSTM(input_size=embeddings_size, hidden_size=hidden_size,
            bidirectional=True)


    def forward(self, src_sentences, src_lengths):
        # TODO: calcule embeddings usando self.embeddings_table

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

        return all_hidden_states, last_hidden_states, last_cell_states
```

2.2 Modelo Seq2Seq básico

En esta parte usaremos un modelo Seq2Seq básico, donde el decodificador utiliza sólo el estado oculto final del codificador. Específicamente, este estado es utilizado para inicializar el estado oculto del decodificador, tal como el modelo visto en clases (Sutskever et al.,2014).

Para el primer paso usaremos un índice especial para indicar que comienza la decodificación: *"start idx"*. La estructura general del decodificador se muestra a continuación.

```
class DecoderModule(nn.Module):
    def __init__(
            self,
            embeddings_table,
            hidden_size,
            start_idx,
            dst_vocab_size,
        ):
        super().__init__()

        self.embeddings_table = embeddings_table
        self.start_idx = torch.tensor(start_idx).to(DEVICE)
        self.hidden_size = hidden_size

        # capa que mapee estado interno de decodificador a vocabulario de salida
        self.W_vocab = torch.nn.Linear()        # TODO: completar parametros

        # decodificador recurrente
        self.lstm_cell = torch.nn.LSTMCell()        # TODO: completar parametros

    def forward(
            self,
            all_enc_hidden_states,
            final_enc_hidden_states,
            final_enc_cell_states,
            max_sentence_length,
        ):

        out = []

        # estado oculto (h, c) inicial para decodificador
        state = (final_enc_hidden_states, final_enc_cell_states)
```

```
        # loop de decodificacion
        y_t = self.embeddings_table(self.start_idx.repeat(BATCH_SIZE))
        for i in range(max_sentence_length):
            state = self.lstm_cell(y_t, state)

            P_t = # TODO: calule logits de salida usando W_vocab
            out.append(P_t)

            _, max_indices = P_t.max(dim=1)
            y_t = self.embeddings_table(max_indices)

        return torch.stack(out, dim=1)
```

Observación: se deberá asignar un índice al token de *"start of sentence"* para señalar al decodificador que está decodificando la primera palabra de la oración. Esto es importante puesto que el decodificador espera recibir en cada paso el embedding de la respuesta que dio en el paso anterior. Para el primer paso, como aún no ha respondido nada, se le entrega el embedding de *"start of sentence"*.

Actividad 4

Complete el código anterior en las partes indicadas con el comentario: `[TODO]`.

- ¿Cuántos parámetros entrenables tiene el modelo?
- ¿De qué tamaño es el vocabulario del lenguaje de origen?
- ¿De qué tamaño es el vocabulario del lenguaje de destino?

Actividad 5

Entrene su modelo en SCAN y reporte gráficamente cómo varían la precisión y la pérdida confome avanza el entrenamiento. Comente.

2.3 Modelo Seq2Seq con atención

En este paso construiremos sobre el decodificador anterior para agregarle atención. Además de la predicción del paso anterior entregaremos como entrada a la LSTM de decodificación la representación atendida de la secuencia de entrada completa. Se espera para esta sección que sus decodificadores utilicen la salida h_t para calcular los pesos de una suma ponderada sobre todos los estados intermedios del codificador, tal como vimos en clase. El resultado de esta suma ponderada se entrega a la capa lineal (*fully-connected*) final además del estado final de la LSTM.

```
class DecoderModule(nn.Module):
    def __init__(
        self,
        embeddings_table,
        embeddings_size,
        hidden_size,
        start_idx,
        dst_vocab_size,
    ):
    super().__init__()

    self.embeddings_table = embeddings_table
    self.embeddings_size = embeddings_size
    self.hidden_size = hidden_size
    self.start_idx = torch.tensor(start_idx).to(DEVICE)

    # capa que mapee estado interno de decodificador a vocabulario de salida
    self.W_vocab = torch.nn.Linear()            # TODO: completar parametros

    # decodificador recurrente
    self.lstm_cell = torch.nn.LSTMCell()        # TODO: completar parametros

    self.attention = AttentionModule(self.hidden_size)
```

```
    def forward(
        self,
        all_enc_hidden_states,
        final_enc_hidden_states,
        final_enc_cell_states,
        max_sentence_length,
    ):
    out = []

    # estado oculto (h, c) inicial para decodificador
    state = (final_enc_hidden_states, final_enc_cell_states)

    # loop de decodificacion
    y_t = self.embeddings_table(self.start_idx.repeat(BATCH_SIZE))
    for i in range(max_sentence_length):
        state = self.lstm_cell(y_t, state)
        h_t, _ = state

        # obtener representacion batch*hidden usando atencion
        O_t = self.attention(h_t, all_enc_hidden_states)

        # calular logits de salida usando W_vocab
        concat_input = torch.cat((h_t, O_t), -1)
        P_t = self.W_vocab(concat_input)
        out.append(P_t)

        _, max_indices = P_t.max(dim=1)
        y_t = self.embeddings_table(max_indices)

    return torch.stack(out, dim=1)
```

Un módulo de atención compara una *Query* con todos los *Keys*. Luego, utiliza el *alignment score* (métrica de cuan parecidos a Q es cada K) para obtener una distribución sobre cada K, en este caso usaremos las distancias coseno.

$$\text{score}(Q, K_n) = \frac{Q * K}{\sqrt{m}}$$

Esta distribución se obtiene dividiendo las distancias entre los vectores Q y K_n (el nésimo *Key*) por la raíz cuadrada del número de dimensiones de los vectores, y luego usando *Softmax* para normalizar.

$$A = Softmax([\text{score}(Q, K_1), \text{score}(Q, K_2) \ldots \text{score}(Q, K_N)]^T)$$

El resultado de esta operación es un vector de pesos que se usan para calcular el promedio ponderado de todos los *Values*.

$$O = \sum_{n \in [1 \ldots N]} V_n * A_n$$

Expresado usando álgebra matricial:

$$\text{Attention}\,(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{m}}\right) V = AV$$

Para esta tarea usaremos el estado interno de la componente recurrente del decodificador como *Query*, y usaremos todos los h_i del codificador (el estado intermedio correspondiente a cada palabra de la entrada) como *Key* y *Value*. El resultado de la operación completa retornará el promedio ponderado de "las palabras codificadas" donde los pesos ponderadores estarán condicionados por el decodificador.

```
class AttentionModule(nn.Module):
    def __init__(self):
    super().__init__()
```

```
    def forward(
        self,
        h_t,
        all_enc_hidden_states,
    ):
    # TODO: completar
    return O_t
```

Actividad 6

Complete el código indicado en los comentarios marcados con `[TODO]`.

- ¿Cuántos parámetros entrenables tiene el modelo?
- ¿De qué tamaño es el vocabulario del lenguaje de origen?
- ¿De qué tamaño es el vocabulario del lenguaje de destino?

Actividad 7

Entrene su modelo con atención en SCAN y reporte gráficamente cómo varían la precisión y la pérdida confome avanza el entrenamiento. Comente. Compare el rendimiento del modelo con atencón respecto del modelo básico sin atención de la parte anterior. Comente **brevemente** sus principales Observaciones.

Actividad 8

Seleccione 2 instancias del set de test, para cada una obtenga la evolución de los coeficientes de atención durante la decodificación, reporte los coeficientes y comente **brevemente** sus principales observaciones.

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] I. Sutskever, O. Vinyals, and Q. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014.

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

[3] https://github.com/brendenlake/SCAN