Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales

Fine Tuning con Lora de Modelos Grandes de Lenguaje

Tesis de Licenciatura en Ciencias de Datos

Juan Tollo

Director: Luciana Ferrer
Codirector: Lautaro Estienne
Buenos Aires, 2024

LA GUERRA DE LAS GALAXIAS: REBELIÓN E IMPERIO

La princesa Leia, líder del movimiento rebelde que desea reinstaurar la República en la galaxia en los tiempos ominosos del Imperio, es capturada por las malévolas Fuerzas Imperiales, capitaneadas por el implacable Darth Vader. El intrépido Luke Skywalker, ayudado por Han Solo, capitán de la nave espacial "El Halcón Milenario", y los androides, R2D2 y C3PO, serán los encargados de luchar contra el enemigo y rescatar a la princesa para volver a instaurar la justicia en el seno de la Galaxia (aprox. 200 palabras).

Palabras claves: Guerra, Rebelión, Wookie, Jedi, Fuerza, Imperio (no menos de 5).

STAR WARS: REBELLION AND EMPIRE

In a galaxy far, far away, a psychopathic emperor and his most trusted servant – a former Jedi Knight known as Darth Vader – are ruling a universe with fear. They have built a horrifying weapon known as the Death Star, a giant battle station capable of annihilating a world in less than a second. When the Death Star's master plans are captured by the fledgling Rebel Alliance, Vader starts a pursuit of the ship carrying them. A young dissident Senator, Leia Organa, is aboard the ship & puts the plans into a maintenance robot named R2-D2. Although she is captured, the Death Star plans cannot be found, as R2 & his companion, a tall robot named C-3PO, have escaped to the desert world of Tatooine below. Through a series of mishaps, the robots end up in the hands of a farm boy named Luke Skywalker, who lives with his Uncle Owen & Aunt Beru. Owen & Beru are viciously murdered by the Empire's stormtroopers who are trying to recover the plans, and Luke & the robots meet with former Jedi Knight Obi-Wan Kenobi to try to return the plans to Leia Organa's home, Alderaan. After contracting a pilot named Han Solo & his Wookiee companion Chewbacca, they escape an Imperial blockade. But when they reach Alderaan's coordinates, they find it destroyed - by the Death Star. They soon find themselves caught in a tractor beam & pulled into the Death Star. Although they rescue Leia Organa from the Death Star after a series of narrow escapes, Kenobi becomes one with the Force after being killed by his former pupil - Darth Vader. They reach the Alliance's base on Yavin's fourth moon, but the Imperials are in hot pursuit with the Death Star, and plan to annihilate the Rebel base. The Rebels must quickly find a way to eliminate the Death Star before it destroys them as it did Alderaan (aprox. 200 palabras).

Keywords: War, Rebellion, Wookie, Jedi, The Force, Empire (no menos de 5).

AGRADECIMIENTOS

A mi pareja Elisa, a mis padres, compañeros y profesores; por la motivación y el apoyo constante.

A mi persona favorita.

Índice general

1. MODELADO DE LENGUAJE

1.1. Fundamentos de modelos de lenguaje

Para introducir los modelos de lenguaje seguimos el capitulo 3 de *Speech and Language Processing*, de Jurafsky [**?**]. Según Jurafsky, un modelo de lenguaje (en adelante **LM** por sus siglas en inglés) es un modelo que calcula las probabilidades de la siguiente palabra o de una frase. Dada una frase previa, que llamamos historia, h, el objetivo es calcular las probabilidades de la siguiente palabra o frase y, esto es $P(y|h)$. El calculo de la probabilidad de la siguiente palabra le permite a la los grandes modelos de lenguaje actuales (**LLMs**) construir un texto coherente basado en una frase inicial. Iterativamente, se calcula la siguiente palabra más probable y se agrega esa palabra a la historia en el siguiente paso.
La probabilidad de una secuencia de palabras es la probabilidad conjunta de cada una de las palabras en ese orden. Una aplicación concreta de este uso podía ser un corrector. Si tenemos una secuencia de palabras donde la probabilidad de la secuencia sea muy baja tal vez expresaría que hay una palabra incorrecta o algún error ortográfico. Por ejemplo, tenemos la oración .El niño come manzna todos los días.Çon bigramas podríamos encontrar que las probabilidades de çome manznaz "manzna todos"son muy bajas pudiendo haber algún error ortográfico y a partir de ahí ofrecer alguna alternativa más probable. Entonces podríamos encontrar la palabra más probable y sugerirla. Para expresar la probabilidad conjunta formalizamos la notación siguiendo a Jurafsky. Sea X_i la variable aleatoria que toma el valor *el*. Luego podemos calcular la probabilidad de que la variable aleatoria tome ese valor: $P(X_i = "el")$ que simplificamos como $P("el")$. Para calcular la probabilidad conjunta de una frase, esto es cada palabra asumiendo un valor particular escribimos, $P(X_1 = w_1, X_2 = w_2, ..., X_n = w_n)$, y que simplificamos como $P(w_1, w_2, ..., w_n)$.
Luego Jurafsky muestra que por la regla de la cadena tenemos que ,

$$P(X_1, X_2, ..., X_n) = P(X_2|X_1)P(X_3|X_{1:2})P(X_n|X_{1:n-1}) = \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \tag{1.1}$$

Entonces la probabilidad conjunta de una secuencia por la regla de la cadena es:

$$P(w_1, w_2, ..., w_n) = P(w_2|w_1)P(w_3|w_{1:2})P(w_n|w_{1:n-1}) = \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \tag{1.2}$$

Luego, usando n-gramas podemos aproximar cada una de estas probabilidades considerando sólo las últimas palabras si suponemos que la probabilidad de una palabra depende principalmente las últimas palabras. Esto es un modelo de **Markov** donde el peso de los últimos estados es decisivo en el calculo de las probabilidades futuras y no es necesario remontarse a estados muy lejanos. Luego podríamos aproximar con n-gramas. Tomemos N el valor de N-grama, entonces decimos que

$$P(w_n|w_1 : w_{n-1}) \approx P(w_n|w_{n-N+1:n-1})$$

Asumiendo esto luego podemos aproximarnos a la probabilidad de una oración reemplazando esta aproximación en la ecuación $regla_c adena P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{n-N+1:n-1})$

Por ejemplo podríamos aproximar con bigramas. Dada la secuencia .El niño comió una manzanaçaclulamos

$$P(\text{"}el\text{"},\text{"}niño\text{"},\text{"}comió\text{"},\text{"}una\text{"},\text{"}manzana\text{"}) \approx P(\text{"}niño\text{"}|\text{"}el\text{"})\times P(\text{"}comió\text{"}|\text{"}niño\text{"})\times\ldots\times P(\text{"}manzana\text{"}|\text{"} \tag{1.3}$$

Como el calculo de estas probabilidades involucra la multiplicación de números muy pequeños por lo general se usa una suma de logaritmos en lugar de productos de probabilidades.

Los modelos n-gramas actualmente no se utilizan para construir oraciones consistentes pero son parte de la historia de los modelos de lenguaje y nos permitieron introducir nociones de probabilidad y notación con modelos sencillos al tiempo que nos dan una intuición sobre el problema que estamos trabajando.

Los **n-gramas** son un modelo sencillo a partir del cual nos podemos familiarizar con el problema y avanzar en su formalización, Un caso particular de n-grams es un bi-grama que consiste en una secuencia de dos palabras donde queremos predecir la probabilidad de la segunda palabra. Tomemos por ejemplo el bi-grama *Peter Pan* . Queremos calcular la probabilidad de observar Pan luego de la palabra Peter entre todas las posibles palabras que pueden venir después de Peter. Basados en un corpus de texto (por ejemplo la Wikipedia, libros, etc.) contamos la frecuencia de ver *Peter Pan* y contamos las apariciones de bi-gramas que comiencen con Peter. El cociente entre estas dos es la probabilidad de ver Pan luego de ver Peter. Podemos extender esta idea para distintos tamaños de secuencias que es lo que llamamos n-gramas.[**?**]

Comenzar con los n-gramas nos permite comprender el propósito del modelo de lenguaje sin entrar en los detalles de las arquitecturas de redes neuronales utilizadas actualmente. Un problema que se busca superar es que como las expresioens de lenguaje son creativas no siempre vamos a tener representatividad en el corpus de todas las palabras posibles. Para dar una idea, con 10 conjuntos de sustantivos, verbos y adjetivos tenemos 10^3 combinaciones posibles y puede que algunas tengan menos representatividad en el corpus de las esperadas. Además, calcular la probabilidad ya para una secuencia pequeña resulta muy complicado. Por ejemplo, si quisiéramos calcular la probabilidad conjunta de una secuencia especifica de tamaño 5. Necesitamos contar las apariciones de esa secuencia especifica en el corpus y contar la cantidad de secuencias de tamaño 5 presentes en el corpus que comienzan con las primeras 4 palabras. [**?**]

1.1.1. Entrenamiento de un modelo de lenguaje

Tarea de modelo de lenguaje. Calcular la probabilidad de una secuencia. Jurafsky. Fundamentar bien la tareas que se está realizando. Calculo de proba de secuencia. Entrenar con texto no supervisado. Y queremos resolver tareas especificas. Plantear transfer learning que se retoma luego. Explicar la representacion de las palabras en en embedings de tokens.

1.2. Arquitecturas actuales de Modelos de lenguaje en el contexto de aprendizaje automático

https://web.stanford.edu/ jurafsky/slp3/10.pdf

1.2.1. LSTM como contexto de apararicion.

1.2.2. Arquitectura transformers.

1.2.3. - Modelo decoder only: familia gpt.Loss autoregresiva.

1.2.4. Tendencia actual: modelo de lenguaje grande

Performance de los modelos y propiedades emergentes.

2. ALINEACIÓN DEL MODELO DE LENGUAJE

PEFT. Modelo zero shots.

Queremos maximizar las probabilidades condicionales dado un prompt relacionada a una tarea especifica. Siguiendo a Hue *et. al*, dado un modelo de lenguaje autoregresivo $P_\Phi()y|x)$ parametrizado por Φ. $P_\Phi()y|x)$ puede ser cualquier modelo multiproposito sin que esté entrenado para una tarea particular como GPT basado en la arquitectura de Transformer. Entonces consideremos el problema de adaptar ese modelo para una tarea de razonamiento. Por ejemplo, dada una oración responder cuál de las siguientes oraciones es su continuación más plausible. Cada tarea posterior (al pre-entrenamiento) es representada por un conjunto de pares contexto-objetivo (usualmente *target* en inglés): $Z = (x_i), y_{ii=1,...N}$ donde x_i e y_i son secuencias de *tokens* [**?**]. En nuestro caso una secuencia que representa a la oración y las continuaciones plausibles, y otro *token* que representa la respuesta correcta. Durante el ajuste completo del modelo queremos maximizar la probabilidad de obtener nuestro valor objetivo y_i. Para eso inicializamos los pesos del modelo Phi_0 y actualizamos $\Phi_0 + \delta\Phi$y sucesivamente siguiendo el gradiente para maximizar la probabilidad condicional de ver la secuencia de tokens objetivo:

$$\max_{\Phi} \sum_{(x,y)\in Z} \sum_{t=1}^{|y|} log(P_{\Phi(y_t|x,y)}) < t$$

Uno de los principales problemas del ajuste de todos los pesos es que en cada tarea *posterior* es necesario actualizar un conjunto de pesos que tiene igual dimensión que los pesos del modelo. Entonces si tenemos un modelo con miles de millones de parámetros, guardar y poner a funcionar distintas instancias independientes para cada tarea particular puede ser computacional mente costoso. [**?**]

2.1. Fine Tuning con LoRA

En esta sección sintetizamos las secciones relevantes para nuestro propósito del trabajo de Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, & Weizhu Chen. (2021). *LoRA: Low-rank adaptation of large language models.*

Nos interesa ajustar los pesos de un modelo pre-entrenado para realizar tareas especificas, en nuestro caso, tareas de razonamiento de sentido común. Esta adaptación es realizada ajustando los pesos de todo el modelo. Entonces el principal problema del ajuste necesitamos ajustar *tantos tantos* pesos como el modelo original. La magnitud de los recursos involucrados hace que resulte un desafío enorme entonces para la escala de modelo con la que se esté trabajando realizar un ajuste de los pesos. Para Hu *et al.* las alternativas de adaptación que intentan aprender algunos parámetros o añadir un módulo adicional tienen el atractivo que resultan agiles de poner en producción, pero suelen introducir problemas de latencia y no logran tener un desempeño como el de ajustar todos los pesos del modelo.

Hu *et al.* para proponer *Low Rank Adaptation* (LoRa) se inspiran en la idea de que la adaptación de un modelo sobre parametrizado es en realidad un problema de baja dimensión. Un modelo puede tener muchisimos parámetros libres pero las variaciones de los mismos que realmente afectan el desempeño del modelo pueden reducirse a un subespacio mucho menor. Es decir, una porción menor de parámetros relevantes definen un subespacio definido por un conjunto de direcciones significativas que son combinaciones lineales de los parámetros. Basado en esta idea, Lora adapta de forma indirecta los pesos del modelo optimizando sobre un descomposición formada por capas densas de bajo rango mientraslos pesos del modelo original preentrenado permancen sin cambios. Hu *et al.* señalan algunas ventajas del método propuesto. Por un lado, dado un modelo base si tenemos las matrices de adaptación A y B calculadas, podemos adaptar el modelo para distintas tareas sin introducir demoras en la puesta en producción ni latencia en la inferencia. Por otro lado, como no se tienen que actualizar todos los parámetros los costos de computacionales se reducen 3 veces respecto a un ajuste completo de todos los pesos.

Comentario sobre notación

Lora trabaja sobre los módulos de autoatención de la arquitectura de *transformers* y para facilitar la explicación los autores siguen la notación convencional para referirse a sus partes. Las dimensiones de entrada y salida del transformes se nombran d_{model}. Para nombrar a las matrices de proyección *query/key//value/output* (por sus nombres en inglés), en el módulo de auto-atención, los autores siguen la convención de usar W_q, W_k, W_v y W_o. Y se denota con la letra r al rango del módulo LoRa. Finalmente, LoRa usa Adam para optimizar el modelo y usan un red neuronal *transformer MLP feedforward* donde MLP es el inglés perceptrón multi capa con un red de avance directo sin ciclos.

2.1.1. Propuesta Lora

Retomando el problema de cómo maximizar la probabilidad de ver nuestra secuencia de tokens objetivos, la propuesta de LoRa consiste en aprender un conjunto de parámetros θ cuya dimensión es más baja que Φ de manera que podemos aproximar la optimización de $\Delta\Phi$ optimizando sobre θ:

$$\max_{\Phi} \sum_{(x,y)\in Z} \sum_{t=1}^{|y|} log(P_{\Phi_o+\Delta\Phi(\theta)(y_t|x,y)}) < t$$

Una red neuronal contiene muchas capas densas en su interior que hacen multiplicaciones de matricas. En general estas matrices son de rango completo. Sin embargo, Aghajanyan *et al.* demuestran que en una adaptación posterior las matrices son de bajo rango. Inspirados en esta idea los autores se proponen descomponer $\Delta\Phi$ de dimensiones $d \times k$ por dos matrices A y B de $d \times r$ y $r \times k$ respectivamente con $r \ll min(r, k)$ y AB de dimensiones $d \times k$. Los presos de W_0 no son modificados. A es inicializada al azar con una distribución gausiana y B es puesta en cero al comienzo. Entonces el modelo completo es $W_0 + AB$. Como puede verse al comienzo del ajuste la matriz de pesos es la misma que la original. Por último, el único ajuste adicional del modelo es un factor de esca-

lamiento $\frac{\alpha}{r}$ donde α tiene un rol parecido a realizar ajustes sobre la tasa de aprendizaje. [**?**]

Este enfoque presenta varias ventajas. Por un lado, como no se modifican los pesos del modelo pre-entrenado es muy fácil de poner en funcionamiento y no implica costos adicionales sobre inferencia. Por otro lado, cómo redujimos la dimensión de las matrices de pesos a optimizar el costo computacional es mucho menor. Para Hue et al. LoRa puede aplicarse a cualquiera de las matrices de pesos de las redes neuronales para reducir la cantidad de parámetros a entrenar. En particular, podemos aplicar LoRa en la arquitectura Transformers hay 4 matrices en donde podemos aplicarlas (W_v, W_q, W_k y W_0) y 2 matrices en los módulos MLP (Capas Perceptrón Multicapa) . Los autores tratan a cada una de las matrices del módulo de auto atención como si fueran de tamaño d_{model} por d_{model} aunque luego esta dimensión sea descompuesta en múltiples cabezas de atención. En el trabajo de LoRa se centran sólo en modificar las matrices de los módulos de autoatención y dejan abierto para un estudio posterior el análisis de trabajar sobre las matrices de los módulos MLP.

Para Hue et. al es interesante preguntarse dado un presupuesto de una cantidad finita de parámetros a optimizar qué matrices de pesos convendría modificar. Indirectamente también estaríamos aprendiendo al responder esta pregunta dónde los modelos aprenden las relaciones que les permiten resolver las tareas. Para Hue *et al.* la optimización de parte de los pesos del modelo original conviene hacerla en la cabezas de atención, los Transformers, porque es más eficiente que modificar las capas de perceptrones simples que están fuera de la capa de Transformers. En cada capa de auto atención teníamos las matrices de pesos W_v, W_q, W_k y W_0 que representa los pesos del modelo pre-entrenado . Para evaluar empíricamente dónde se consiguen mejores resultados, Hue *et al.* definen un presupuesto de una matriz de rango 8 que impone una limitación de cantidad de parámetros que podemos modificar. Ahora bien, podemos usar esta matriz para ajustar los pesos de una de las matrices o bien utilizar 4 dimensiones para ajustar los pesos de una de las matrices y 4 dimensiones para ajustar los pesos de otra. En el resultado empírico observan que la mejora en la performance en tareas posteriores se obtiene destinando 4 dimensiones a W_v y 4 dimensiones a W_q en lugar de destinar 8 dimensiones a cualquiera de las matrices W_v, W_q, W_k y W_0 .

Poner la que mas varia Otra pregunta interesante es saber cuál es rango óptimo para el rango r de las matrices A y B. Hue et al. empíricamente observan que una dimensión de 1 es suficiente para las matrices W_k y W_v mientras que la matriz W_q si es entrenase sola, sí precisa un rango mayor.

3. EXPERIMENTOS

3.1. Modelos

3.1.1. Arquitectura Microsoft Phi 1.5

3.1.2. Arquitectura Mistral

3.2. Datos

Trabajamos con datos para entrenar y evaluar al modelo en tareas de razonamiento que son comunes a las métricas publicadas por los principales modelos de lenguaje natural: *Socialiqa* y *Hellaswag* ([**?**], [**?**]). En ambos casos están disponibles los conjunto de entrenamiento, validación y testeo. El conjunto de testeo no tiene etiquetas entonces usamos el de validación en ambos casos para testeo y construimos un conjunto de validación a partir del de entrenamiento.

Socialiqa entrena en comprender implicancias sociales de ciertas situaciones. Veamos un ejemplo de una instancia:

Contexto: Cameron decided to have a barbecue and gathered her friends together.
Pregunta: How would others feel as a result?

- **Respuesta A:** like attending
- **Respuesta B:** like staying home
- **Respuesta C:** a good friend to have

La instancia tiene un *contexto, pregunta y posibles respuestas correctas*. La respuesta correcta depende de las implicancias sociales de esa consigna. Como hay tres posibles respuestas correctas y el los conjuntos de datos tienen etiquetas balanceadas, elegir una al azar nos daría un *accuracy* alrededor de 0.33. *Hellaswag* propone una oración inicial y 4 posibles continuaciones, con una respuesta correcta. El *accuracy* de elegir una respuesta al azar debería ser más bajo.

Then, the man writes over the snow covering the window of a car, and a woman wearing winter clothes smiles. Then,

- the man adds wax to the windshield and cuts it.
- the man puts on a Christmas coat, knitted with netting.
- the man continues removing the snow on his car.
- a person boards a ski lift, while two men support the head of the person wearing winter clothes as the women sled.

La extensión promedio de las instancias de *Hellaswag* es mayor comparada a la de *Socialiqa*. Una entrada de *hellswag* puede tener más de 300 palabras mientras que es normal encontrar instancias de alrededor de 50 palabras de *socialiqa*. Cuando expliquemos el entrenamiento veremos que tendremos dos opciones. Una sería entrenar en predecir la etiqueta de una instancia y otra entrenar en predecir la siguiente palabra para cada palabra de la

instancia y tomando como *prompt* las palabras anteriores de una instancia. Esta segunda opción que es la que elegiremos y que justificaremos más adelante implica mayor costo computacional cuanto mayor es el tamaño de cada instancia. Al mismo tiempo, la elección de hiperámetros de entrenamiento como el tamaño del batch y la cantidad de capas LoRa aumentan el costo computacional. Como nos interesa tener libertad para modificar estos hiperparámetros, elegimos restringir la extensión de las instancias de *hellaswag* y quedarnos con un subconjunto de instancias de *Hellaswag* que tienen menos de 100 palabras. En cuanto a la distribución de los conjuntos de entrenamiento incluyen en ambos casos más de 30 mil instancias. Para una versión preliminar a la entrega final de la tesis elegimos correr con conjuntos elegidos al azar de 5000 instancias de entrenamiento, 300 de validación y 2000 de testeo.

Por último tanto para entrenamiento como para evaluación usamos 0 *shots*. Es decir no le pasamos ninún ejemplo al modelo de la tarea que debía resolver sino que le pasamos la consigna. De esta forma se mide la capacidad de un modelo para realizar una tarea sin haber recibido ningún ejemplo específico de esa tarea en el momento del entrenamiento directo para esa tarea. Es una prueba de la capacidad generalizadora del modelo basada en el conocimiento previo adquirido durante el entrenamiento pre-fine-tuning o durante el propio fine-tuning en contextos más generales.

3.3. Métricas

3.3.1. Teoría de la decisión de Bayes

Supongamos que elegimos una función de costo $C(h, d)$ definida tal como lo hicimos en la sección 2.

Hay dos casos: d categorica. Estamos en la matriz que vimos antes. Pero si d no es categorica, la función de costo es una generalización de la matriz que vimos antes.

Usando proba condicional $p(h, x) = p(x)p(h|x) \implies$

$[V(z)]_{p(z)}$ denota el valor de la esperanza de v(z), donde z es una o más variables aleatorias con respecto a la distribución $p(z)$. En este paper Luciana reserva Costo Esperado solo para el caso categórico a pesar de que este también ser un costo esperado. La esperanza definida en en la ecuación anterior puede ser minimizada, eligiendo, para cada x, la decisión que minimiza la suma interior.

$$d_B = argmin_d \sum_{i=1}^{K} C(H_i.d)P(H_i|x) \tag{3.1}$$

d_B usualmente es llamada la decisión de Bayes., Ahora bien, qué distribución de p usamos para obtener la esperanza y la decisión de bayes. Nos gustaría poder usar la verdadera distribución de nuestra data. Pero a menos que corramos simulaciones no tenemos acceso a esta data. Solo podemos tener modelos para ella. De hecho nuestro clasificador produce un modelo de la pinta $P(h|x)$. Y podemos usar estas posteriors en la ecuación anterior. Para la decisión óptima de Bayes no precisamos de la distribución, sólo la usamos para el calculo de la esperanza. En adelante entonces para el calculo de la esperanza se van a usar los datos del modelo de distribución de los datos de evaluación.

3.3.2. Métricas para un sistema de puntaje

En esta sección vamos a describir un conjunto de métricas que corresponden al valor de la esperanza de la PSR. Las PSRs miden la bondad de los puntajes asumiendo que la información disponible son las probabilidades a poteriori, *posteriors* por su uso común en inglés, que nos da la salida de un modelo. Estas *posteriors* son usadas para realizar desiciones Bayesianas.

Dada un vector de puntajes s que es la salida al sistema para una muestra x con una clase h y la función de costo $C(h, d)$ la cual representa el costo incurrido por el sistema cuando toma la decisión d con para una muestra de clase h, podemos construir la PSR, C^* como,

$$C^*(h, s) = C(h, f_B(s)) \tag{3.2}$$

donde $d_B(s)$ es la decisión bayesiana correspondiente a la función de costo C. Notemos que expresamos la decisión bayesiana en función de los puntajes y no en función de x, que depende sólo del vector de *posteriors* $P(.|x)$ que acá tomamos igual a $s(x)$ que es la salida del clasificador. Luego de PSR. C, mide la bondad de las *posteriors* puestas a trabajar, usandolas para tomar una decisión bayesiana para una función de costo, que es la mejor decision posible si todo lo que conocemos son las *posteriors* de la salida del sistema. El PSR mide el costo para una muestra particular. Para obtener una métrica que podamos usar en evaluación tenemos que calcular la esperanza de la PSR sobre la data. Tal como lo hicimos con el costo esperado, normalmente lo que hacemos es calcular la estimación empirica de la esperanza que denotamos EPSR.

$$EPSR = \frac{1}{N} \sum_{t=1}^{N} C^*(h^{(t)}, s^t) \tag{3.3}$$

Como vemos a continuación, diferentes EPSR pueden ser calculadas usando distintas funciones de costo C. La ecuación3.2presenta una definición constructuva de PSR dado que las desiciones bayesianas para s son las que minimizan el costo esperado con respecto de la distribución q(h) sobre las clases es minimizada si el vector de puntajes s coincide con la distribución. Esto es , C es un PSR si satisface que

$$q \in argmin[C^*(h, s)]_{q(h} \tag{3.4}$$

Esta propiedad es satisfecha para cualquier C^* construída con la ecuación 3.2, dado que las desiciones bayesianas respectoa a s son las que minimizan el costo esperado con respecto a s.

3.3.3. Entropía Cruzada

Definimos a d como el vector de *posteriors* a partir del cual se toman las decisiones y llamamos k al tamaño del vector que coincide con la posibles clases que puede tomar nuestra variable aleatoria. Decimos que d vive en R^K, y definimos la función de costo $C(h, d) = -\log(d_h)$, llamada también de pérdida logarítmica, donde en un abuso pequeño de notación, d_h es el vector de decisión d en el índice correspondiente a la clase h.

Puede demostrarse que para esta función de costo la decision de Bayes óptoma es $d_B(s) = s$. Esto es , la desicion que minimiza el esperanza de la función de pérdida lograrimia es el score mismo. Aquí también la
La esperanza de esta función de pérdida respecto a la distribución empírica nos da la

entropía cruzada que es ampliamente utilizada en *machine learning* como función objetivo para el entrenamiento del sistema:

$$XE = -\frac{1}{N}\sum_{t=1}^{N}\log(s_{h^(t)}(x^t)) \tag{3.5}$$

donde $s_{h^(t)}(x^t)$ representa la salida del sistema para las variables de la muestra t para la clase $h^{(t)}$, que sería la *verdadera* clase de la muestra t. Esta expresión puede ser reescrita de la misma forma que hicimos con la ecuación considerando las priors de las clases.

$$XE = -\sum_{t=1}^{N}\frac{P_{h^{(t)}}}{N_{h^t}}\log(s_{h^(t)}(x^t)) \tag{3.6}$$

$$= -\sum_{i=1}^{K}\frac{P_i}{N_i}\sum_{t=1|H^{(t}=H_i}^{N}\log(s_i(x^t)) \tag{3.7}$$

Donde la *prior* P_i puede ser distinta en entrenamiento y evaluacióny esta formula nos permite ajustarlo. Si $s_{h^(t)}(x^{(t)})$ es cercano a 1 el costo es cercano a cero y cuando es cercano a 0 el sistema tiende a penalizar mucho más la certeza sobre desiciones incorrectas. Esta es una propiedad que queremos tener en muchas aplicaciones de machine learning por lo que XE es una excelente EPSR que se puede usar de base para evaluar la bondad de las probabilidades posteriores de la salida de un sistema.

Puede verse que la pérdida logarítmica es una PSR estricta por la discrepancia de Kullback-Liebler entre dos distribuciones es cero únicamente cuando las distribuciones son iguales. A diferencia de $\widetilde{CE}$, XE considera no uno sino todos los puntos posibles de corte para la decisión de clasificación.

Para tener un resultado más interpretable podemos normalizar XE con un score *naive*. El mejor estimador naive es el que devuelve la distribución a *priori*: $-\sum_{i=1}^{K} P_i \log(P_i)$ que es la entropía de la distribución a priori. Dividiendo a XE por esta entropía tenemos un estimador normalizado. **Normalizacion para cualquier metrica**

3.4. Experimentos y análisis

3.4.1. Explicación experimento

Los experimentos involucraron en la adaptación de los modelos Phi 1.5 y Phi 2 para un subconjunto de 5000 instancias elegidos al azar de los conjuntos de entrenamiento de Social IQA y Hellaswag, los cuales contenían 33 mil y 39 mil instancias respectivamente. A pesar de la reducción significativa en el número de instancias de entrenamiento,la adaptación resultó en mejoras relevantes comparado con los modelos base originales de Phi.

Realizamos una sola modificación al código de entrenamiento de Lora. en lugar de solo guardar el último modelo en intervalos fijos de iteraciones, también conservamos el modelo que demostró el mejor desempeño en el conjunto de validación. Este enfoque nos permite evitar la retención del último modelo ajustado, que podría estar sobreajustado a los datos de entrenamiento

Los resultados obtenidos para los modelos base de Phi 1.5 y y Phi 2 fueron consistentes con los publicados por Microsoft en sus informes. [**?**] [**?**] No obstante, notamos una diferencia significativa en SoiciaIQA base para Phi 1.5. Microsoft reporta un *accuracy* de 0.53 mientras que nosotros obtuvimos un 0.59 de *accuracy*. 3.1. Ambos modelos deberían tener resultados similares porque no se ajustaron los pesos. Puede ser que haya habido algún cambio del modelo Phi 1.5 o bien que la diferencia sea por el subconjunto de entrenamiento que seleccionamos al azar.

En dicha tabl, presentamos los resultados de los modelos ajustados los conjuntos de testeo de ambos data sets. Seleccionamos los modelos ajustados que reportaron un *accuracy* más alto en el conjunto de validación. Observamos que tanto en el conjunto Hellaswag como en SocialIQA el ajuste de los modelos base con LoRa mejora el *accuracy* . En Phi 1.5 al ajustarse pasa de 0.59 a 0.61 de accuracy en SocialIQA y de 0.38 a 0.4 de accuracy en Hellaswag. En el caso de Phi 2 también se ve una mejora, incluso más considerable en SocialIQA que pasa de 0.67 a 0.72 el accuracy

En el caso del ajuste del modelo Phi 1.5 en las tareas SocialIQA la mejora parece acotada. Todavía tiene mucha distancia con un modelo Phi 2 base. En cambio en para las tareas de Hellaswag sí observamos que Phi 1.5 ajustado se desempeña igual o mejor que Phi 2.

Phi 2 ajustado con LoRa logra buen desempeño superando a modelos mucho más grandes

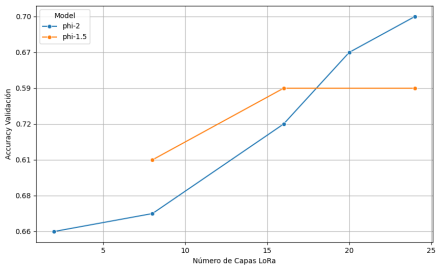
Modelo	SocialIQA	Hellaswag
phi-1.5 - Base	0.59	0.38
phi-1.5 - Fine-tuned	0.61	0.46
phi-2 - Base	0.67	0.44
phi-2 - Fine-tuned	0.72	0.44

Tab. 3.1: Comparación modelos Base vs. con fine tuning

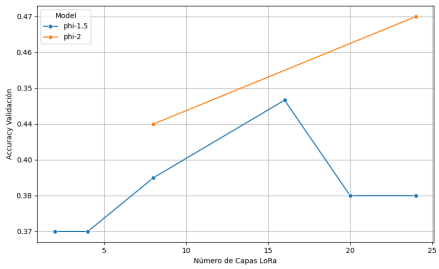
como Chat GPT 3.5 con 350 billones de parámetros. Por ejemplo, en la verisón instruct GPT 3.5 tiene un accuracy de 0.71, un poco menor al 0.72 de Phi 2.

Aún así entre estos modelos hay una distancia respecto a la performance de los humanos que es de 88,5[**?**]. El desempeño regular puede estar relacionado con que en el campo de tareas de sentido común se requiere un profundo entendimiento de las formas de comportarse humanas y las interacciones sociales puede deberse a que aparecen más infrequentemente en los corpus de texto de pre-entrenamiento. Es decir, los modelos de lenguaje deberían aprender más de las formas de actuar de los humanos que del significado semántico de las palabras. [**?**].

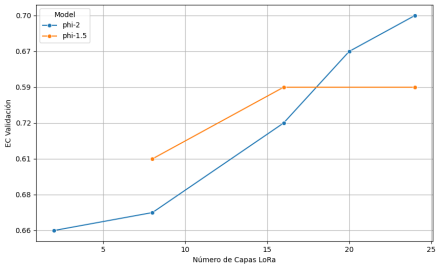
3.4.2. Capas de Lora



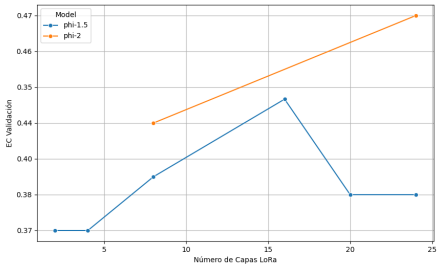
(a) Accuracy Validación for SocialiQA



(b) Accuracy Validación for HellaSwag



(c) EC Validación for SocialiQA



(d) EC Validación for HellaSwag

Fig. 3.1: Compaciónn de las métricas Accuracy y Entropía Cruzada en el conjunto de validación de los datasets SocialIQA y HellaSwag. La primera columna muestran los resultados para SocialiQA, mientras que la segunda los resultados para HellaSwag.

Por columna data set Primera columna SQA Segunda columna HS Tasa de error CENormalizada

3.4.3. Variación del rango de Lora

3.4.4. Otros de hiperparametros

Curvas de perdida Learning rate Iteraciones

4. CONCLUSIONES Y POSIBLES TRABAJOS A FUTURO

Bibliografía