

# Entendiendo las Redes Neuronales

Zhiwei:

¿Podrías explicar el código?

```
self.weights = [w-(eta/len(mini_batch))*nw
                 for w, nw in zip(self.weights, nabla_w)]
self.biases = [b-(eta/len(mini_batch))*nb
               for b, nb in zip(self.biases, nabla_b)]
```

En este fragmento de código, se están actualizando los pesos (`weights`) y los sesgos (`biases`) de una red neuronal utilizando el descenso de gradiente. Aquí está la traducción al español:

```
self.weights = [w-(eta/len(mini_batch))*nw
                 for w, nw in zip(self.weights, nabla_w)]
self.biases = [b-(eta/len(mini_batch))*nb
               for b, nb in zip(self.biases, nabla_b)]
```

En este código, los pesos y sesgos se actualizan restando el producto de la tasa de aprendizaje (`eta`) dividida por el tamaño del mini lote (`mini_batch`) y el gradiente correspondiente (`nabla_w` para los pesos y `nabla_b` para los sesgos). Esto es una implementación típica del descenso de gradiente en el contexto del entrenamiento de redes neuronales.

ChatGPT:

¡Por supuesto! El código que proporcionaste parece ser parte de un algoritmo de aprendizaje automático, específicamente una implementación del descenso de gradiente estocástico (SGD, por sus siglas en inglés) para actualizar los pesos y sesgos en una red neuronal durante el proceso de entrenamiento. Vamos a desglosarlo paso a paso:

1. `self.weights` y `self.biases`: Estas variables probablemente forman parte de una clase de red neuronal, representando los pesos y los sesgos de la red. En una red neuronal, los pesos son parámetros que ajustan la fuerza de las conexiones entre neuronas en diferentes capas, mientras que los sesgos son parámetros que controlan el umbral de activación de la neurona.
2. `nabla_w` y `nabla_b`: Estas variables probablemente almacenan los gradientes de la función de costo con respecto a los pesos y los sesgos, respectivamente. Durante el proceso de entrenamiento, los gradientes se calculan para determinar cómo cambia la función de

costo en relación con los pesos y los sesgos. Esta información se utiliza para actualizar los parámetros con el fin de minimizar el costo.

3. `eta`: Este es la tasa de aprendizaje, que controla el tamaño del paso que se toma en la dirección de los gradientes durante la actualización de los parámetros. Una tasa de aprendizaje más alta puede llevar a una convergencia más rápida durante el entrenamiento, pero también puede causar inestabilidad. Una tasa de aprendizaje más pequeña podría mejorar la estabilidad, pero aumentar el tiempo de entrenamiento.
4. `mini_batch`: Este es un subconjunto de los datos de entrenamiento utilizado en cada iteración del descenso de gradiente estocástico. El descenso de gradiente por mini-lotes es una variante de SGD en la que el conjunto completo de datos de entrenamiento se divide en lotes más pequeños, y las actualizaciones de los parámetros se realizan utilizando cada mini-lote.

Ahora, echemos un vistazo al código en sí:

```
self.weights = [w-(eta/len(mini_batch))*nw
                 for w, nw in zip(self.weights, nabla_w)]
```

*Nota: El código no necesita traducción, ya que es un bloque de código en Python y los nombres de las variables y la sintaxis deben permanecer en inglés.*

Esta línea actualiza los pesos de la red neuronal. Utiliza una comprensión de lista para iterar sobre cada peso `w` y su gradiente correspondiente `nw`. Para cada peso, resta una porción del gradiente (escalado por la tasa de aprendizaje `eta` y el tamaño del mini-lote) del peso original. Así es como se actualizan los pesos para minimizar la función de costo.

```
self.biases = [b-(eta/len(mini_batch))*nb
                for b, nb in zip(self.biases, nabla_b)]
```

En este fragmento de código, se está actualizando el valor de los sesgos (`biases`) en una red neuronal. La operación se realiza restando a cada sesgo actual (`b`) un término que depende de la tasa de aprendizaje (`eta`), el tamaño del mini lote (`mini_batch`), y el gradiente correspondiente (`nabla_b`). La expresión se aplica a cada par de sesgo y gradiente utilizando la función `zip`.

Esta línea actualiza los sesgos de la red neuronal de manera similar. Utiliza una comprensión de lista para iterar sobre cada sesgo `b` y su gradiente correspondiente `nb`. Para cada sesgo, resta una porción del gradiente (escalado por la tasa de aprendizaje `eta` y el tamaño del mini-lote) del sesgo original.

En resumen, estas líneas de código realizan la actualización de los pesos y los sesgos de la red neuronal utilizando el descenso de gradiente estocástico. Este proceso se repite para cada mini-lote durante la fase de entrenamiento para optimizar los parámetros de la red y mejorar su rendimiento en la tarea dada.

---

Zhiwei: ¿Qué es el modelo de lenguaje de SpaCy?

ChatGPT: SpaCy es una biblioteca de procesamiento de lenguaje natural (NLP) de código abierto desarrollada por Explosion AI. Está diseñada para ser eficiente, rápida y lista para producción, lo que la convierte en una opción popular para diversas tareas de NLP. SpaCy proporciona modelos de lenguaje preentrenados que pueden procesar y comprender texto, permitiendo a los desarrolladores realizar diversas tareas de NLP, como etiquetado de partes del discurso, reconocimiento de entidades nombradas, análisis de oraciones y más.

Las características clave del modelo de lenguaje SpaCy incluyen:

1. Tokenización: SpaCy puede dividir un texto en palabras o tokens individuales, teniendo en cuenta diversas reglas y complejidades lingüísticas.
2. Etiquetado de Partes del Habla: La biblioteca puede asignar etiquetas de partes del habla a cada token, indicando si es un sustantivo, verbo, adjetivo, etc.
3. Reconocimiento de Entidades Nombradas (NER): SpaCy puede identificar y categorizar entidades como nombres de personas, organizaciones, ubicaciones, fechas, etc.
4. Análisis de Dependencias: Puede analizar la estructura gramatical de una oración y determinar las relaciones entre las palabras.
5. Lematización: SpaCy puede convertir palabras a su forma base o raíz (lemas).
6. Vectores de palabras: La biblioteca proporciona vectores de palabras, también conocidos como incrustaciones de palabras, que son representaciones numéricas densas de palabras que capturan significado semántico.

SpaCy es compatible con múltiples idiomas y viene con modelos preentrenados para diferentes lenguajes, lo que lo hace ampliamente utilizado en diversas aplicaciones de Procesamiento del Lenguaje Natural (NLP), incluyendo extracción de información, análisis de sentimientos, chatbots, traducción de idiomas y más. Además, permite a los desarrolladores entrenar modelos personalizados en sus propios conjuntos de datos si es necesario.

---

Zhiwei: ¿Qué es el modelo de lenguaje SpaCy?

ChatGPT: SpaCy es una biblioteca de procesamiento de lenguaje natural (NLP) de código abierto desarrollada por Explosion AI. Está diseñada para ser eficiente, rápida y fácil de usar, lo que la convierte en una opción popular para diversas tareas de NLP. SpaCy proporciona modelos de lenguaje preentrenados para varios idiomas, permitiendo a los usuarios realizar tareas como tokenización, etiquetado de partes del discurso, reconocimiento de entidades nombradas, análisis de dependencias y más.

El núcleo de la velocidad y eficiencia de SpaCy radica en sus implementaciones y estructuras de datos optimizadas en Cython, lo que lo hace especialmente adecuado para entornos de producción y aplicaciones del mundo real. La biblioteca está escrita en Python y viene con modelos preentrenados para varios idiomas, incluyendo inglés, alemán, francés, español y más.

Los modelos de lenguaje de SpaCy utilizan técnicas de aprendizaje profundo para aprender los patrones subyacentes en los datos de lenguaje natural. Estos modelos se entrenan en grandes corpus de texto, y las representaciones aprendidas pueden luego utilizarse para diversas tareas de Procesamiento de Lenguaje Natural (NLP, por sus siglas en inglés). Los modelos preentrenados pueden ajustarse aún más con datos específicos para lograr un mejor rendimiento en tareas específicas de un dominio.

La arquitectura y la facilidad de uso han convertido a SpaCy en una opción popular entre desarrolladores e investigadores en la comunidad de Procesamiento de Lenguaje Natural (NLP). A menudo se utiliza para tareas como extracción de información, clasificación de texto, reconocimiento de entidades nombradas, etiquetado de partes del discurso y análisis sintáctico. Además, el diseño modular de SpaCy permite a los usuarios integrarlo fácilmente en sus flujos de trabajo y pipelines de NLP existentes.

---

Zhiwei: ¿Podrías explicar?

```
class Llama:
    @staticmethod
    def build(
        ckpt_dir: str,
        tokenizer_path: str,
```

```

max_seq_len: int,
max_batch_size: int,
model_parallel_size: Optional[int] = None,
) -> "Llama":
    if not torch.distributed.is_initialized():
        torch.distributed.init_process_group("nccl")
    if not model_parallel_is_initialized():
        if model_parallel_size is None:
            model_parallel_size = int(os.environ.get("WORLD_SIZE", 1))
        initialize_model_parallel(model_parallel_size)

local_rank = int(os.environ.get("LOCAL_RANK", 0))
torch.cuda.set_device(local_rank)

# la semilla debe ser la misma en todos los procesos
torch.manual_seed(1)

if local_rank > 0:
    sys.stdout = open(os.devnull, "w")

    start_time = time.time()
    checkpoints = sorted(Path(ckpt_dir).glob("*.pth"))
    assert len(checkpoints) > 0, f"no se encontraron archivos de checkpoint en {ckpt_dir}"
    assert model_parallel_size == len(
        checkpoints
    ), f"Cargando un checkpoint para MP={len(checkpoints)} pero el tamaño del mundo es {model_parallel_size}"
    ckpt_path = checkpoints[get_model_parallel_rank()]
    checkpoint = torch.load(ckpt_path, map_location="cpu")
    with open(Path(ckpt_dir) / "params.json", "r") as f:
        params = json.loads(f.read())

```

Ejecuta el comando y luego obtén el registro de errores.

```

% torchrun --nproc_per_node 1 example_text_completion.py \
    --ckpt_dir llama-2-7b/ \
    --tokenizer_path tokenizer.model \
    --max_seq_len 128 --max_batch_size 4

```

NOTA: Actualmente no se admiten redirecciones en Windows o MacOS.

Traceback (most recent call last):

```
File "/Users/lzwjava/projects/llama/example_text_completion.py", line 55, in <module>
    fire.Fire(main)
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/fire/core.py", line 141, in Fire
    component_trace = _Fire(component, args, parsed_flag_args, context, name)
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/fire/core.py", line 475, in _Fire
    component, remaining_args = _CallAndUpdateTrace(
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/fire/core.py", line 691, in _CallAndUpdateTrace
    component = fn(*varargs, **kwargs)
File "/Users/lzwjava/projects/llama/example_text_completion.py", line 18, in main
    generator = Llama.build(
File "/Users/lzwjava/projects/llama/llama/generation.py", line 62, in build
    torch.distributed.init_process_group("nccl")
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/distributed_c10d.py", line 144, in
    default_pg = _new_process_group_helper(
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/distributed_c10d.py", line 144, in
    raise RuntimeError("Distributed package doesn't have NCCL " "built in")
RuntimeError: Distributed package doesn't have NCCL built in
ERROR:torch.distributed.elastic.multiprocessing.api:falló (código de salida: 1) local_rank: 0 (pid: 23273)
```

Traceback (most recent call last): File "/Users/lzwjava/anaconda3/bin/torchrun", line 33, in  
sys.exit(load\_entry\_point('torch==2.0.1', 'console\_scripts', 'torchrun')()) File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/elastic/multiprocessing/errors/init.py", line 346, in wrapper return  
f(\*args, \*\*kwargs) File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/run.py",  
line 794, in main run(args) File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/run.py",  
line 785, in run elastic\_launch( File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/elastic/multiprocessing/errors/init.py", line 134, in **call** return launch\_agent(self.\_config, self.\_entrypoint, list(args)) File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/launcher/api.py", line 250, in launch\_agent raise ChildFailedError(  
torch.distributed.elastic.multiprocessing.errors.ChildFailedError: =====  
example\_text\_completion.py FALLÓ ----- Fallos: -----  
----- Causa raíz (primer fallo observado): [0]: tiempo  
: 2023-08-03\_15:35:37 host : 1.0.0.127.in-addr.arpa rango : 0 (local\_rank: 0) código  
de salida: 1 (pid: 23273) archivo de error: traza : Para habilitar la traza, consulte:  
<https://pytorch.org/docs/stable/elastic/errors.html> =====

Aprende el rastreo de la pila de errores.

Pregúntale a ChatGPT qué significa Backend.NCCL en PyTorch.

---

Zhiwei:

código:

```
```python
import torch
from transformers import LlamaTokenizer, LlamaForCausalLM

model_path = '.'
# model_path = 'openlm-research/open_llama_7b'

tokenizer = LlamaTokenizer.from_pretrained(model_path)
model = LlamaForCausalLM.from_pretrained(
    model_path, torch_dtype=torch.float16, device_map='auto',
)

prompt = 'Q: ¿Cuál es el animal más grande?\nA:'
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
generation_output = model.generate(
    input_ids=input_ids, max_new_tokens=32
)
print(tokenizer.decode(generation_output[0]))
```

error:

```
/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/generation/utils.py:1445: UserWarning:
  warnings.warn(
Traceback (most recent call last):
  File "/home/lzw/Projects/open_llama_3b/run.py", line 17, in <module>
    generation_output = model.generate(
                        ~~~~~~
  File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/utils/_contextlib.py", line 1
```

```

    return func(*args, **kwargs)
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/generation/utils.py", line 1111
    return self.greedy_search(
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/generation/utils.py", line 1111
    outputs = self(
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/module.py", line 1154
    return forward_call(*args, **kwargs)
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/models/llama/modeling_llama.py", line 1111
    outputs = self.model(
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/module.py", line 1154
    return forward_call(*args, **kwargs)
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/models/llama/modeling_llama.py", line 1111
    inputs_embeds = self.embed_tokens(input_ids)
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/module.py", line 1154
    return forward_call(*args, **kwargs)
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/sparse.py", line 111
    return F.embedding(
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/functional.py", line 2210
    return torch.embedding(weight, input, padding_idx, scale_grad_by_freq, sparse)
    ~~~~~
RuntimeError: Se esperaba que todos los tensores estuvieran en el mismo dispositivo, pero se encontraron en dispositivos diferentes:

```

¿Cómo solucionarlo?

ChatGPT: (No se traduce, ya que es un nombre propio en inglés)

El mensaje de error que estás viendo indica que hay una discrepancia en los tipos de dispositivos entre tu tensor de entrada (`input_ids`) y tu modelo (`LlamaForCausalLM`). El tensor de entrada está en la CPU (`cpu`), mientras que el modelo está en la GPU (`cuda`). Para solucionar



esto, debes asegurarte de que el tensor de entrada esté en el mismo dispositivo que el modelo antes de generar texto utilizando el modelo.

Aquí está el código modificado para solucionar el problema:

```
import torch
from transformers import LlamaTokenizer, LlamaForCausalLM

model_path = '.'
# model_path = 'openlm-research/open_llama_7b'

tokenizer = LlamaTokenizer.from_pretrained(model_path)
model = LlamaForCausalLM.from_pretrained(
    model_path, torch_dtype=torch.float16, device_map='auto',
)

prompt = 'Q: ¿Cuál es el animal más grande?\nA:'
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
```

## Mueve el tensor `input_ids` al mismo dispositivo que el modelo

```
input_ids = input_ids.to(model.device)

generation_output = model.generate(
    input_ids=input_ids, max_new_tokens=32
)
print(tokenizer.decode(generation_output[0]))
```

En este código, he añadido la línea `input_ids = input_ids.to(model.device)` después de obtener el tensor `input_ids` del tokenizador. Esta línea mueve el tensor `input_ids` al mismo dispositivo que el modelo (ya sea CPU o GPU) antes de generar el texto. Esto debería resolver el error “Expected all tensors to be on the same device”.

Recuerda que si estás ejecutando este código en una máquina con GPU y tu modelo está en una GPU, generalmente es más eficiente mantener tanto el modelo como los datos de entrada en la GPU para aprovechar las capacidades de procesamiento en paralelo.

---

Zhiwei: ¿Podrías explicar los siguientes códigos?

```
def generar(
    self,
    tokens_de_prompt: List[List[int]],
    max_long_gen: int,
    temperatura: float = 0.6,
    top_p: float = 0.9,
    logprobs: bool = False,
    eco: bool = False,
) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
    params = self.model.params
    bsz = len(tokens_de_prompt)
    assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)

    min_prompt_len = min(len(t) for t in prompt_tokens)
    max_prompt_len = max(len(t) for t in prompt_tokens)
    assert max_prompt_len <= params.max_seq_len
    total_len = min(params.max_seq_len, max_gen_len + max_prompt_len)

    pad_id = self.tokenizer.pad_id
    tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device="cuda")
    for k, t in enumerate(prompt_tokens):
        tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device="cuda")
    if logprobs:
        token_logprobs = torch.zeros_like(tokens, dtype=torch.float)

    prev_pos = 0
    eos_reached = torch.tensor([False] * bsz, device="cuda")
    input_text_mask = tokens != pad_id
    for cur_pos in range(min_prompt_len, total_len):
        logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
        if logprobs:
            token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(
                input=logits.transpose(1, 2),
```

```

        target=tokens[:, prev_pos + 1 : cur_pos + 1],
        reduction="none",
        ignore_index=pad_id,
    )

    if temperature > 0:
        probs = torch.softmax(logits[:, -1] / temperature, dim=-1)
        next_token = sample_top_p(probs, top_p)
    else:
        next_token = torch.argmax(logits[:, -1], dim=-1)

next_token = next_token.reshape(-1)
# solo reemplazar el token si el prompt ya ha sido generado
next_token = torch.where(
    input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
)
tokens[:, cur_pos] = next_token
eos_reached |= (~input_text_mask[:, cur_pos]) & (
    next_token == self.tokenizer.eos_id
)

prev_pos = cur_pos
if all(eos_reached):
    break

    if logprobs:
        token_logprobs = token_logprobs.tolist()
        out_tokens, out_logprobs = [], []
        for i, toks in enumerate(tokens.tolist()):
            # cortar a la longitud máxima de generación
            start = 0 if echo else len(prompt_tokens[i])
            toks = toks[start : len(prompt_tokens[i]) + max_gen_len]
            probs = None
            if logprobs:
                probs = token_logprobs[i][start : len(prompt_tokens[i]) + max_gen_len]
            # cortar al token eos si está presente
            if self.tokenizer.eos_id in toks:
                eos_idx = toks.index(self.tokenizer.eos_id)
                toks = toks[:eos_idx]

```

```

        probs = probs[:eos_idx] if logprobs else None
    out_tokens.append(toks)
    out_logprobs.append(probs)
    return (out_tokens, out_logprobs if logprobs else None)

```

ChatGPT: ¡Por supuesto! El código proporcionado es un método llamado `generate` que toma un `prompt` (texto de entrada) y genera texto utilizando un modelo de lenguaje. Este método está diseñado para funcionar con un modelo y un tokenizador específicos, y realiza la generación de texto utilizando muestreo codicioso (greedy sampling) o muestreo top-p. Aquí tienes un desglose de cómo funciona el método:

### 1. **Parámetros del Método:**

- `prompt_tokens`: Una lista de prompts tokenizados, donde cada prompt se representa como una lista de enteros (IDs de tokens).
- `max_gen_len`: La longitud máxima de la salida generada (en tokens).
- `temperature`: Un parámetro que controla la aleatoriedad de la generación. Valores más altos (por ejemplo, 1.0) hacen que la salida sea más aleatoria, mientras que valores más bajos (por ejemplo, 0.6) la hacen más determinista.
- `top_p`: Un parámetro que controla la diversidad del texto generado. Valores más altos (por ejemplo, 0.9) permiten un muestreo más diverso.
- `logprobs`: Si es `True`, el método también devuelve las probabilidades logarítmicas de los tokens generados.
- `echo`: Si es `True`, la salida generada incluirá el prompt original.

2. **Inicialización:** El método inicializa variables como `params`, `bsz` (tamaño del lote), `min_prompt_len` y `max_prompt_len`. También calcula `total_len`, que es la longitud total de la salida generada incluyendo el prompt.

3. **Generación de Tokens:** El método genera tokens para el texto de salida. Utiliza un bucle para iterar sobre las posiciones en la secuencia generada, comenzando desde el final del *prompt* y avanzando hacia adelante.

- Los *logits* para cada posición se obtienen pasando la secuencia parcial a la función *forward* del modelo.
- Si `temperature` es mayor que 0, los *logits* se transforman en probabilidades utilizando *softmax* con el valor de temperatura dado. Si `temperature` es 0, se selecciona el token con el *logit* más alto utilizando `torch.argmax`.

- El siguiente token se determina en función de las probabilidades generadas o del *argmax*, y se coloca en la posición correspondiente del tensor `tokens`.
- Se detectan los tokens EOS (fin de secuencia), y si todas las secuencias han alcanzado un token EOS, se detiene el proceso de generación.
- Los tokens generados se mantienen en el tensor `tokens`.

#### 4. **Procesamiento de la Salida Generada:**

- Los tokens generados se procesan para recortar la salida a la longitud máxima deseada (`max_gen_len`).
- Si `echo` es `True`, el prompt se incluye en la salida generada.
- Si se encuentra el token `eos`, el texto generado se corta en ese punto.

5. **Retorno:** El método devuelve una tupla que contiene los tokens generados y, si `logprobs` es `True`, las probabilidades logarítmicas de los tokens generados.

Este método genera texto prediciendo tokens de manera iterativa basándose en el contexto proporcionado por el *prompt* y los tokens generados previamente. Utiliza parámetros de generación específicos para controlar la aleatoriedad y diversidad del texto generado. El método está diseñado para ser utilizado con una arquitectura de modelo y un tokenizador específicos, ya que depende de sus propiedades y métodos para la generación de texto.