Anexos

Anexo I: ADAM Optimizer

Mostramos aquí el funcionamiento del algoritmo ADAM conforme está definido en [11], explicando primero el pseudocódigo y posteriormente demostrando la convergencia del mismo.

Algorithm 1: ADAM, algorithm for stochastic optimization

Require: α : Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0 : Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
  $v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m_t} \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v_t} \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Estudiamos ahora la convergencia del sistema:

Dada una secuencia arbitraria de funciones convexas $f_1(\theta), f_2(\theta), ..., f_T(\theta)$, nuestro objetivo es predecir en el tiempo t el parámetro θ_t, y evaluarla en una función de coste desconocida previamente f_t. Como la secuencia es desconocida, el algoritmo se evalúa usando el "regret", la suma de las anteriores diferencias entre la predicción $f_t(\theta_t)$ y el punto de mejor resultado $f_t(\theta^*)$, es decir:

$$R(T) = \sum_{t=1}^{T} [f_t(\theta_t) - f_t(\theta^*)] \tag{8.1}$$

Donde $\theta^* = argmin_{\theta \in \chi} \sum_{t=1}^{T} f_t(\theta)$. Para $T \geq 1$:

$$R(T) \leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T\widehat{v}_{T,i}} + \frac{\alpha(1+\beta_1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\lambda)^2} \tag{8.2}$$

$$\frac{R(T)}{T} = O(\frac{1}{\sqrt{T}}) \tag{8.3}$$

Y con estas dos ecuaciones, se puede comprobar que

$$\lim_{T\to\infty} \frac{R(T)}{T} = 0 \tag{8.4}$$

Anexo II: Exponentially Decaying Error

Analizamos el error que surge al intentar usar redes que almacenen datos a lo largo de distintas iteraciones del algoritmo, tal y como está explicado en [5].

Definimos como $d_k(t)$ a la salida de la unidad k en un tiempo t. Si calculamos el error cuadrático medio ($MSE(d_k(t)) = \gamma_k(t)$):

$$\gamma_k(t) = f'_k(net_k(t))(d_k(t) - y^k(t)) \tag{8.5}$$

Donde:

$$y^i(t) = f_i(net_i(t)) \tag{8.6}$$

es la activación de una unidad i de no entrada con función de activación f_i,

$$net_i(t) = \sum_j w_{ij} y^j(t-1) \tag{8.7}$$

es la entrada de la red para la unidad i, siendo w_{ij} los pesos de conexión de la unidad j a la i.

El error de propagación para una unidad j de no salida es:

$$\gamma_j(t) = f'_j(net_j(t)) \sum_i w_{ij}\gamma_i(t+1) \tag{8.8}$$

La actualización de los pesos w_{jl} es $\alpha\gamma_j(t)y^l(t-1)$, siendo α el ratio de aprendizaje, y l una unidad cualquiera unida a la j.

Suponemos ahora una red completamente conectada, cuyas unidades de no entrada van numeradas de 1 a n. Si estudiamos el error de una unidad u a una unidad v en un tiempo cualquiera t, este se propaga en el tiempo q pasos, lo que escala el error por un factor:

$$\frac{\partial\gamma_v(t-q)}{\partial\gamma_u(t)} = \begin{cases} f'_v(net_v(t-1))w_{uv} & \text{if } q = 1 \\ f'_v(net_v(t-q)) \sum_{l=1}^{n} \frac{\partial\gamma_l(t-q+1)}{\partial\gamma_u(t)} w_{lv} & \text{if } q > 1 \end{cases} \tag{8.9}$$

Con $l_q = v$ y $l_0 = u$, se puede probar por inducción que:

$$\frac{\partial\gamma_v(t-q)}{\partial\gamma_u(t)} = \sum_{l_1=0}^{n} ... \sum_{l_{q-1}}^{n} \prod_{m=1}^{q} f'_{l_m}(net_{l_m}(t-m))w_{l_m l_{m-1}} \tag{8.10}$$

La suma de los n^{q-1} términos $\prod_{m=1}^{q} f'_{l_m}(net_{l_m}(t-m))w_{l_m l_{m-1}}$ determinan la propagación del error, por lo que podemos ver que, si:

$$|f'_{l_m}(net_{l_m}(t-m))w_{l_m l_{m-1}}| > 1 \tag{8.11}$$

para cualquier m, el producto se incrementa exponencialmente con q, por lo que el error crece también de forma exponencial, lo que puede llevar a pesos oscilantes y errores de aprendizaje.

Por otro lado, si:

$$|f'_{l_m}(net_{l_m}(t-m))w_{l_m l_{m-1}}| < 1 \tag{8.12}$$

para cualquier m, el error se desvanece y no puede aprender en un tiempo aceptable.

Anexo III: Tipos de Decoders

Los decoders usados en redes neuronales nos dan un conjunto de posibles palabras, junto con sus probabilidades, pero no siempre es más óptimo coger la palabra con la mayor probabilidad, ya que tras una palabra con poca probabilidad se puede esconder una con mucha mayor probabilidad. Dependiendo del algoritmo que se use para elegir, existen diferentes tipos de búsquedas [9] [20].

Greedy Decoder

El algoritmo más sencillo es el Greedy Decoder, traducido como "codicioso", el cual se basa en elegir el token con la mayor probabilidad.

Este sistema cuenta con una ventaja muy importante, y es su velocidad, ya que no es necesario hacer una búsqueda de posibles palabras siguientes, simplemente se elige la que tiene mayor probabilidad. También cuenta con la ventaja de la sencillez, y es que es un algoritmo muy simple de entender e implementar.

Como principal desventaja, nos encontramos con que no nos puede garantizar el devolvernos la frase más óptima, ya que tras una palabra con mucha probabilidad pueden encontrarse muchas con poca probabilidad, y viceversa, haciendo que se pierda parte de la información. Para intentar solventar este problema, se idearon los Beam Decoders.

Beam Decoder

En el caso del Beam Decoder, en vez de elegir un resultado en cada iteración, se dejan abiertas un número K de hipótesis

$$H_t = (y_1^1, ..., y_t^1), ..., (y_1^K, ..., y_t^K) \tag{8.13}$$

A cada hipótesis se le añaden todos los posibles candidatos v para formar posibles hipótesis

$$\tilde{h}_v^i = (y_1^i, ..., y_t^i, v) \tag{8.14}$$

y se calcula la puntuación (score) de cada posible nueva hipótesis como:

$$s(\tilde{h}_v^i) = \sum_{t'=1}^{t} logp(y_{t'}^i | y_{<t'}^i) + logp(v | t_{\leq t}^i) \tag{8.15}$$

A continuación, se seleccionan las K hipótesis con mayor puntuación, y se usan en la siguiente iteración del algoritmo:

$$H_t = arg - top - k_{i,v} s(\tilde{h}_v^i) \tag{8.16}$$

Este paso se repite hasta que todas las hipótesis terminen con un token de fin de frase, momento en el que se devuelve como resultado la hipótesis con la mayor puntuación.

Similitud con Grafos

Todos estos algoritmos son realmente algoritmos de búsqueda de grafos, en el que tenemos un grafo en el que los costes son las probabilidades de cada palabra, y se busca obtener el camino que maximice este valor.

Esto nos podría indicar que se pueden usar otro tipo de algoritmos para la resolución de estos problemas, tales como Dijkstra o A*. Sin embargo, la gran cantidad de posibles caminos y el no conocer de

antemano todos los posibles nodos, hace que estos algoritmos sean desechados como idea y se utilicen algoritmos más simples pero eficientes.

Anexo IV: Código del modelo

En este anexo presentamos el código utilizado en la parte final de este trabajo.

Modelo GPT-2

Empezamos el modelo creando una capa simple de convolución, que será usada más adelante para el algoritmo de attention:

```
class Conv1D(nn.Module):
    def __init__(self, nf, nx):
        super(Conv1D, self).__init__()
        self.nf = nf
        w = torch.empty(nx, nf)
        # Inicializamos con una distribución normal de desviación estandar 0.02
        nn.init.normal_(w, std=0.02)
        self.weight = Parameter(w)
        self.bias = Parameter(torch.zeros(nf))

    def forward(self, x):
        size_out = x.size()[:-1] + (self.nf,)
        x = torch.addmm(self.bias, x.view(-1, x.size(-1)), self.weight)
        x = x.view(*size_out)
        return x
```

Seguimos con una capa de normalización, que se usa varias veces dentro de cada bloque de la red:

```
class LayerNorm(nn.Module):
    def __init__(self, hidden_size, eps=1e-12):
        # Capa de normalización
        super(LayerNorm, self).__init__()
        self.weight = nn.Parameter(torch.ones(hidden_size))
        self.bias = nn.Parameter(torch.zeros(hidden_size))
        self.variance_epsilon = eps

    def forward(self, x):
        u = x.mean(-1, keepdim=True)
        s = (x - u).pow(2).mean(-1, keepdim=True)
        x = (x - u) / torch.sqrt(s + self.variance_epsilon)
        return self.weight * x + self.bias
```

Con estos dos códigos ya podemos programar la capa de attention:

```
class Attention(nn.Module):
    def __init__(self, nx, n_ctx, config, scale=False):
        super(Attention, self).__init__()
        n_state = nx

        assert n_state % config.n_head == 0
```

```
        # Configuramos todas las variables
        self.register_buffer("bias", torch.tril(torch.ones(n_ctx, n_ctx))
            .view(1, 1, n_ctx, n_ctx))
        self.n_head = config.n_head
        self.split_size = n_state
        self.scale = scale
        self.c_attn = Conv1D(n_state * 3, nx)
        self.c_proj = Conv1D(n_state, nx)
        # Configuramos el dopout para la cama de attention,
        # lo cual sólo se utiliza en el pre-entrenamiento
        self.attn_dropout = None
        if config.attn_dropout is not None:
            self.attn_dropout = nn.Dropout(p = config.attn_dropout)

        self.res_dropout = None
        if config.res_dropout is not None:
            self.res_dropout = nn.Dropout(p = config.res_dropout)

    def _attn(self, q, k, v):
        # Una sola capa de attention
        w = torch.matmul(q, k)
        if self.scale:
            w = w / math.sqrt(v.size(-1))

        nd, ns = w.size(-2), w.size(-1)
        b = self.bias[:, :, ns-nd:ns, :ns]
        w = w * b - 1e4 * (1 - b)

        w = nn.Softmax(dim=-1)(w)

        # Introducimos el dropout
        if self.attn_dropout is not None:
            w = self.attn_dropout(w)

        return torch.matmul(w, v)

    # Unir las diferentes capas
    def merge_heads(self, x):
        x = x.permute(0, 2, 1, 3).contiguous()
        new_x_shape = x.size()[:-2] + (x.size(-2) * x.size(-1),)
        return x.view(*new_x_shape)

    # Separar las diferentes capas
    def split_heads(self, x, k=False):
        new_x_shape = x.size()[:-1] + (self.n_head, x.size(-1) // self.n_head)
        x = x.view(*new_x_shape)
        if k:
```

```
            return x.permute(0, 2, 3, 1)
        else:
            return x.permute(0, 2, 1, 3)

    def forward(self, x, layer_past=None):
        x = self.c_attn(x)
        query, key, value = x.split(self.split_size, dim=2)
        query = self.split_heads(query)
        key = self.split_heads(key, k=True)
        value = self.split_heads(value)
        if layer_past is not None:
            past_key, past_value = layer_past[0].transpose(-2, -1), layer_past[1]
            key = torch.cat((past_key, key), dim=-1) # Concatenamos las nuevas keys y values
            value = torch.cat((past_value, value), dim=-2)
        present = torch.stack((key.transpose(-2, -1), value))
        # Pasamos por la capa de attention y unimos los valores
        a = self._attn(query, key, value)
        a = self.merge_heads(a)
        a = self.c_proj(a)

        # Introducimos el dropout
        if self.res_dropout is not None:
            a = self.res_dropout(a)

        return a, present
```

Necesitamos una capa más para poder construid el modelo final. Esta capa es una MLP o Multi Layer Perceptron:

```
class MLP(nn.Module):
    def __init__(self, n_state, config):
        # Capa Multi Layer
        super(MLP, self).__init__()
        nx = config.n_embd
        self.c_fc = Conv1D(n_state, nx)
        self.c_proj = Conv1D(nx, n_state)
        self.act = gelu

        self.dropout = None
        if config.res_dropout is not None:
            self.dropout = nn.Dropout(p = config.res_dropout)

    def forward(self, x):
        h = self.act(self.c_fc(x))
        h2 = self.c_proj(h)

        # Introducimos el dropout
        if self.dropout is not None:
```

```
            h2 = self.dropout(h2)

        return h2
```

Con esto, construimos la clase "block", la cual define cada uno de los bloques que constituyen el modelo final:

```
class Block(nn.Module):
    def __init__(self, n_ctx, config, scale=False):
        # Cada uno de los bloques de los que está constituida la red
        super(Block, self).__init__()
        nx = config.n_embd
        # Capa de normalización
        self.ln_1 = LayerNorm(nx, eps=config.layer_norm_epsilon)
        # Capa de attention
        self.attn = Attention(nx, n_ctx, config, scale)
        # Segunda capa de normalización
        self.ln_2 = LayerNorm(nx, eps=config.layer_norm_epsilon)
        # Capa Multi Layer
        self.mlp = MLP(4 * nx, config)

    def forward(self, x, layer_past=None):
        # Normalizamos y pasamos por la capa de attention
        a, present = self.attn(self.ln_1(x), layer_past=layer_past)
        x = x + a
        # Normalizamos el resultado y lo pasamos por la MLP
        m = self.mlp(self.ln_2(x))
        x = x + m
        return x, present
```

Por último, creamos la función "forward" dentro de nuestro modelo, que será la que se llama en cada paso de la red neuronal:

```
def forward(self, input_ids, position_ids=None, past=None):

    past_length = 0 if past is None else past[0][0].size(-2)
    past = past if past is not None else [None] * len(self.h)
    assert len(past) == len(self.h)

    # Calculamos los valores que tenemos que añadir a cada
    # token por su posición en la frase
    position_ids = torch.arange(past_length, input_ids.size(-1) + past_length,
        dtype=torch.long, device=input_ids.device)
    position_ids = position_ids.unsqueeze(0).expand_as(input_ids)

    # Calculamos los embeddings
    input_shape = input_ids.size()
    inputs_embeds = self.wte(input_ids)
    position_embeds = self.wpe(position_ids)
```

```
    hidden_states = inputs_embeds + position_embeds
    presents = []
    # Pasamos el resultado por todos los bloques
    for block, layer_past in zip(self.h, past):
        hidden_states, present = block(hidden_states, layer_past)
        presents.append(present)

    hidden_states = self.ln_f(hidden_states)
    output_shape = input_shape + (hidden_states.size(-1),)

    #Cálculo del logit
    h_flat = hidden_states.view(input_shape[0] * input_shape[1], self.config.n_embd)
    logits = torch.matmul(h_flat, torch.t(self.wte.weight))
    logits = logits.view(input_shape[0], input_shape[1], self.config.vocab_size)

    return hidden_states.view(*output_shape), presents, logits #hidden_states, presents, logits
```

Entrenamiento

La parte de entrenamiento es más sencilla. A pesar de tener bastantes códigos, la gran mayoría se utilizan para construir el dataset a partir de un fichero de texto. Como esto ya se ha explicado en el capítulo 7, sólo vamos a incluir el código que entrena a la red:

```
def update(engine, batch):
    # Colocamos el modelo en modo entrenamiento
    model.train()
    # Separamos el batch en input/output
    new_batch = [batch[0]]
    solution = batch[1].to(args.device)
    batch = tuple(input_tensor.to(args.device) for input_tensor in new_batch)
    # Lo pasamos al modelo
    hidden_states, presents, logits = model(*batch)

    #Calculamos el error mediante la función cruzada de entropía
    loss = torch.nn.functional.cross_entropy(
        logits.view(args.train_batch_size, -1, args.n_ctx), solution)
    loss = loss / args.gradient_accumulation_steps
    loss.backward()

    torch.nn.utils.clip_grad_norm_(model.parameters(), args.max_norm)
    # En caso necesario, hacemos un paso de optimizador
    if engine.state.iteration % args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()
```

Muestra de resultados

Por último, el código que se encarga de mostrar los resultados. Este código comparte algunas funciones con el algoritmo de entrenamiento, ya que necesitamos convertir el texto a tensores de la misma forma y usar el mismo diccionario, si no daría error.

```
import logging
import random
from argparse import ArgumentParser
from itertools import chain
from pprint import pformat

import torch
import torch.nn.functional as F

from gpt2PyTorch import * # Modelo GPT2 y GPT2 config
from train import SPECIAL_TOKENS, build_input_from_phrase
from tokenization import SPTokenizer # Tokenizer

SPECIAL_TOKENS = ["<bos>", "<eos>", "<pad>"]

def sample_sequence(history, tokenizer, model, args, current_output=None):
    special_tokens_ids = [tokenizer.bosId, tokenizer.eosId]
    if current_output is None:
        current_output = []
    #Construimos la frase uniendo las historias
    phrase = [history[0]]
    for i in range(1, len(history)):
        phrase.append([tokenizer.eosId])
        phrase.append(history[i])
    phrase = [item for sublist in phrase for item in sublist]
    phrase.insert(0, tokenizer.eosId)

    past = None

    for i in range(args.max_length):

        instance, sequence = build_input_from_phrase(phrase, tokenizer, with_eos=False)

        input_ids = torch.tensor(instance["input_ids"], device=args.device).unsqueeze(0)

        hidden_states, presents, logits = model(input_ids, past=past)

        logits = logits[0, -1, :] / args.temperature
        logits = torch.topk(logits, args.top_k)[0]
        probs = F.softmax(logits, dim=-1)

        prev = torch.topk(probs, 1)[1] if args.no_sample else torch.multinomial(probs, 1)
```

```
        if i < args.min_length and prev.item() in special_tokens_ids:
            while prev.item() in special_tokens_ids:
                prev = torch.multinomial(probs, num_samples=1)

        if prev.item() in special_tokens_ids:
            break
        current_output.append(prev.item())

    return current_output


def run():
    parser = ArgumentParser()
    parser.add_argument("--model_checkpoint", type=str, default="run", help="Path, url or short name of
    parser.add_argument("--max_history", type=int, default=0, help="Number of previous utterances to ke
    parser.add_argument("--device", type=str, default="cuda" if torch.cuda.is_available() else "cpu", h
    parser.add_argument("--tokenizer", type=str, default="spm.model", help="Name of dictionary")

    parser.add_argument("--no_sample", action='store_true', help="Set to use greedy decoding instead of
    parser.add_argument("--max_length", type=int, default=1024, help="Maximum length of the output utte
    parser.add_argument("--min_length", type=int, default=1, help="Minimum length of the output utteran
    parser.add_argument("--seed", type=int, default=42, help="Seed")
    parser.add_argument("--temperature", type=int, default=0.7, help="Sampling softmax temperature")
    parser.add_argument("--top_k", type=int, default=0, help="Filter top-k tokens before sampling (<=0:
    parser.add_argument("--top_p", type=float, default=0.9, help="Nucleus filtering (top-p) before samp
    parser.add_argument("--test", action='store_true', help="If true start with a first evaluation befo
    args = parser.parse_args()

    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__file__)
    logger.info(pformat(args))

    random.seed(args.seed)
    torch.random.manual_seed(args.seed)
    torch.cuda.manual_seed(args.seed)

    logger.info("Get pretrained model and tokenizer")
    tokenizer = SPTokenizer(args.tokenizer)
    args.n_vocab = len(tokenizer.sp)


    model = GPT2Model.from_pretrained(args.model_checkpoint, "checkpoint.pth")

    model.to(args.device)
    model.eval()

    while True:
```

```
        raw_text = input(">>> ")
        while not raw_text:
            print('Prompt should not be empty!')
            raw_text = input(">>> ")
        with torch.no_grad():
            out_ids = sample_sequence([tokenizer.encode(raw_text)], tokenizer, model, args)
        out_text = tokenizer.decode(out_ids)
        print(out_text)


if __name__ == "__main__":
    run()
```