Isaac Reyes, Alejandro Moya, José Guzmán

9.5. Implementación de Red Neuronal Recurrente desde Cero

Ahora estamos preparados para implementar una RNR desde cero. En particular, entrenaremos esta RNR para que funcione como un modelo de lenguaje a nivel de caracteres (ver Sección 9.4) y la entrenaremos con un corpus que consiste en el texto completo de La Máquina del Tiempo de H. G. Wells, siguiendo los pasos de procesamiento de datos descritos en Sección 9.2. Comenzamos cargando el conjunto de datos.

imp y conf inicial

In [17]:
```
use strict;
use warnings;
use Data::Dump qw(dump);
use AI::MXNet qw(mx);
use d2l;
IPerl->load_plugin('Chart::Plotly');
use List::Util qw(sum);
use jjap::numperl;
use constant np => 'numperl';
use experimental 'smartmatch';
```

9.5.1. Modelo RNN

Iniciamos definiendo una clase para implementar el modelo RNN (Sección 9.4.2). Es importante notar que el número de unidades ocultas num_hiddens es un hiperparámetro ajustable.

In [18]:

```
package RNNScratch{
    use base qw(d2l::Module); #@save
    #The RNN model implemented from scratch.
    sub new{
        my ($class, %args) = (shift, d2l->get_arguments(num_inputs=>0, num
        my $self = $class->SUPER::new();
        $self->save_hyperparameters(%args);

        $self->{W_xh} = mx->nd->random->randn($args{num_inputs}, $args{num
        $self->{W_hh} = mx->nd->random->randn($args{num_hiddens}, $args{nu
        $self->{b_h} = mx->nd->zeros([$args{num_hiddens}]);
        return bless($self, $class);
    }
    1;
}
```

Out[18]: 1

```
Subroutine new redefined at reply input line 4.
```

El método forward a continuación define cómo calcular la salida y el estado oculto en cualquier momento, dado la entrada actual y el estado del modelo en el paso de tiempo anterior. Note que el modelo RNN recorre la dimensión más externa de inputs, actualizando el estado oculto paso a paso. El modelo aquí utiliza una función de activación $\tanh$ (Sección 5.1.2.3).

In [19]:
```
my $forward = sub{
    my ($self, %args) = (shift, d2l->get_arguments(inputs => undef,
                                                   state => undef,\@_));

    if (!defined $args{state}){
        # Initial state with shape: (batch_size, num_hiddens)
        $args{state} = mx->nd->zeros([$args{inputs}->shape->[1], $self->{n
    }else{
        ($args{state}, undef) = @{$args{state}};
    }
    my $outputs;
    my $first_dot;
    my $second_dot;
    my $total;
    foreach my $X (@{$args{inputs}}){ # Shape of inputs: (num_steps, batch
        $first_dot = mx->nd->dot($X, $self->{W_xh});
        $second_dot = mx->nd->dot($args{state}, $self->{W_hh});
        $total = $first_dot + $second_dot + $self->{b_h};
        $args{state} = mx->nd->tanh($total);
        push @$outputs, $args{state};

    }

    return $outputs, $args{state};

};
d2l->add_to_class('RNNScratch', 'forward', $forward);
```

Out[19]:
```
*RNNScratch::forward

Subroutine RNNScratch::forward redefined at /usr/local/lib/perl5/site_perl/5.
32.1/x86_64-linux/d2l.pm line 4456.
```

Podemos introducir un minibatch de secuencias de entrada en un modelo RNN de la siguiente manera.

In [20]:
```
my ($batch_size, $num_inputs, $num_hiddens, $num_steps) = (2, 16, 32, 100)
my $rnn = RNNScratch->new(num_inputs => $num_inputs,
                          num_hiddens => $num_hiddens);
my $X = mx->nd->ones([$num_steps, $batch_size, $num_inputs]);
my ($outputs, $state) = $rnn->forward($X);

```

Out[20]:
```
ARRAY(0xbf29178)<AI::MXNet::NDArray 2x32 @cpu(0)>
```

Vamos a verificar si el modelo RNN produce resultados con las dimensiones correctas para asegurarnos de que la dimensionalidad del estado oculto permanezca sin cambios.

In [21]:
```
sub check_len{ #@save
    my ($a, $n) = @_;
    #Check the length of a list.
    if (ref ($a) eq 'AI::MXNet::NDArray'){
        if($a->len != $n){
        print STDERR " list's length ", $a->len, " != expected length ", $
        }
    }
    if (ref($a) eq 'ARRAY'){
        if(scalar @$a != $n){
            print STDERR " list's length ", scalar @$a, " != expected leng
        }
    }

}

sub check_shape{ #@save
    my ($a, $shape) = @_;
    #Check the shape of a tensor.
    print STDERR "tensor's shape", dump ($a->shape), " != expected shape "


}

check_len($outputs, $num_steps);
check_shape($outputs->[0], [$batch_size, $num_hiddens]);
check_shape($state, [$batch_size, $num_hiddens]);
```

Out[21]: 1

```
Subroutine check_len redefined at reply input line 1.

Subroutine check_shape redefined at reply input line 17.
```

9.5.2. Modelo de Lenguaje Basado en RNN

La clase RNNLMScratch que se presenta a continuación define un modelo de lenguaje basado en RNN, donde pasamos nuestro RNN a través del argumento rnn del método **init**. Al entrenar modelos de lenguaje, las entradas y salidas provienen del mismo vocabulario. Por lo tanto, tienen la misma dimensión, que es igual al tamaño del vocabulario. Note que usamos la perplejidad para evaluar el modelo. Como se discutió en la Sección 9.3.2, esto asegura que secuencias de diferentes longitudes sean comparables.

In [22]:

```
package RNNLMScratch{
    use base qw(d2l::Classifier); #@save
    use Data::Dump qw(dump);
    use jjap::numperl;
    use constant np => 'numperl';
    #The RNN-based language model implemented from scratch.
sub new{
        my ($class, %args) = (shift, d2l->get_arguments(rnn=>undef, vocab_


        my $self = $class->SUPER::new();
        $self->save_hyperparameters(%args);
        $self->init_params();
        return bless($self, $class);
    }
    sub init_params{
        my $self = shift;

        $self->{W_hq} = mx->nd->random->randn(
        $self->{rnn}->{num_hiddens}, $self->{vocab_size}) * $self->{rnn}->
        $self->{b_q} = mx->nd->zeros([$self->{vocab_size}]);

        for my $param (@{$self->get_scratch_params()}){
            $param->attach_grad();
        }
    }
    sub training_step{
        my ($self, $batch) = @_; #batch es un array
        my $l = $self->loss($self->forward(@$batch[0 .. scalar(@$batch) -
        $self->plot('ppl', mx->nd->array(np->exp($l->asarray)->asarray), t
        return $l;
    }

    sub validation_step{
        my ($self, $batch) = @_;
        my $l = $self->loss($self->forward(@$batch[0 .. scalar(@$batch) -
        $self->plot('ppl', mx->nd->array(np->exp($l->asarray)->asarray), t
    }


    1;
}
```

Out[22]: 1

```
Subroutine new redefined at reply input line 7.

Subroutine init_params redefined at reply input line 16.

Subroutine training_step redefined at reply input line 27.

Subroutine validation_step redefined at reply input line 34.
```

9.5.2.1. Codificación One-Hot

Recuerda que cada token se representa por un índice numérico que indica la posición en el vocabulario de la palabra/carácter/pieza de palabra correspondiente. Podrías pensar en construir una red neuronal con un solo nodo de entrada (en cada paso de tiempo), donde el índice se podría introducir como un valor escalar. Esto funciona cuando tratamos con entradas numéricas como el precio o la temperatura, donde dos valores suficientemente cercanos deben ser tratados de manera similar. Pero esto no tiene mucho sentido. Las palabras 45^{a} y 46^{a} en nuestro vocabulario resultan ser "their" y "said", cuyos significados no son para nada similares.

Cuando tratamos con datos categóricos, la estrategia más común es representar cada elemento mediante una *codificación one-hot* (recuerda la Sección 4.1.1). Una codificación one-hot es un vector cuya longitud es dada por el tamaño del vocabulario N , donde todas las entradas se establecen en 0 , excepto la entrada correspondiente a nuestro token, que se establece en 1 . Por ejemplo, si el vocabulario tuviera cinco elementos, entonces los vectores one-hot correspondientes a los índices 0 y 2 serían los siguientes.

In [23]:
```
mx->nd->one_hot(mx->nd->array([0,2]),5)->aspdl;
```

Out[23]:
```
[
 [1 0 0 0 0]
 [0 0 1 0 0]
]
```

Los minibatches que muestreamos en cada iteración tomarán la forma (tamaño del lote, número de pasos de tiempo). Una vez que representamos cada entrada como un vector one-hot, podemos pensar en cada minibatch como un tensor tridimensional, donde la longitud a lo largo del tercer eje está dada por el tamaño del vocabulario (`len(vocab)`). A menudo transponemos la entrada para que obtengamos una salida de forma (número de pasos de tiempo, tamaño del lote, tamaño del vocabulario). Esto nos permitirá recorrer más convenientemente la dimensión más externa para actualizar los estados ocultos de un minibatch, paso a paso en el tiempo (por ejemplo, en el método `forward` mencionado anteriormente).

In [24]:
```
my $one_hot = sub {
my ($self, $X) = @_;
    # Output shape: (num_steps, batch_size, vocab_size)
    return mx->nd->one_hot($X->transpose, $self->{vocab_size});
};
d2l->add_to_class('RNNLMScratch', 'one_hot', $one_hot);

```

Out[24]:
```
*RNNLMScratch::one_hot

Subroutine RNNLMScratch::one_hot redefined at /usr/local/lib/perl5/site_perl/
5.32.1/x86_64-linux/d2l.pm line 4456.
```

9.5.2.2. Transformando las Salidas de RNN

El modelo de lenguaje utiliza una capa de salida completamente conectada para transformar las salidas de RNN en predicciones de tokens en cada paso de tiempo.

In [25]:
```
my $output_layer = sub{
    my ($self, $rnn_outputs) = @_;
    my @outputs;
    foreach my $H (@$rnn_outputs){
        push (@outputs, (mx->nd->dot($H, $self->{W_hq}) + $self->{b_q}));
    }
    return mx->nd->stack(@outputs, axis => 1);
};
d2l->add_to_class('RNNLMScratch', 'output_layer', $output_layer); #@save
```

Out[25]:
```
*RNNLMScratch::output_layer

Subroutine RNNLMScratch::output_layer redefined at /usr/local/lib/perl5/site_
perl/5.32.1/x86_64-linux/d2l.pm line 4456.
```

In [26]:
```
my $forward = sub{
    my ($self, $X, $state) = @_;
    my $embs = $self->one_hot($X);
    (my $rnn_outputs, $state) = $self->{rnn}->forward($embs, $state);
    return $self->output_layer($rnn_outputs);

};
d2l->add_to_class('RNNLMScratch', 'forward', $forward); #@save

```

Out[26]:
```
*RNNLMScratch::forward

Subroutine RNNLMScratch::forward redefined at /usr/local/lib/perl5/site_perl/
5.32.1/x86_64-linux/d2l.pm line 4456.
```

Verifiquemos si el cálculo hacia adelante produce salidas con la forma correcta.

In [27]:
```
my $model = RNNLMScratch->new(rnn => $rnn, vocab_size => $num_inputs);
my $outputs = $model->forward(mx->nd->ones([$batch_size, $num_steps], dtyp
check_shape($outputs, [$batch_size, $num_steps, $num_inputs]);
```

Out[27]:
```
1
```

9.5.3 Recorte de Gradientes

Aunque ya estás acostumbrado a pensar en las redes neuronales como "profundas" en el sentido de que muchas capas separan la entrada y salida incluso dentro de un solo paso de tiempo, la longitud de la secuencia introduce una nueva noción de profundidad. Además de

pasar por la red en la dirección de entrada a salida, las entradas en el primer paso de tiempo deben pasar por una cadena de (T) capas a lo largo de los pasos de tiempo para influir en la salida del modelo en el último paso de tiempo. Tomando la perspectiva hacia atrás, en cada iteración, retropropagamos gradientes a través del tiempo, resultando en una cadena de productos matriciales de longitud (\mathcal{O}(T)). Como se mencionó en la Sección 5.4, esto puede resultar en inestabilidad numérica, causando que los gradientes exploten o desaparezcan, dependiendo de las propiedades de las matrices de pesos.

Lidiar con gradientes que desaparecen y explotan es un problema fundamental al diseñar RNNs y ha inspirado algunos de los mayores avances en arquitecturas modernas de redes neuronales. En el próximo capítulo, hablaremos sobre arquitecturas especializadas que fueron diseñadas con la esperanza de mitigar el problema del gradiente que desaparece. Sin embargo, incluso los RNNs modernos a menudo sufren de gradientes que explotan. Una solución común pero poco elegante es simplemente recortar los gradientes, forzando a los "recortados" a tomar valores más pequeños.

Generalmente, al optimizar algún objetivo por descenso de gradiente, actualizamos iterativamente el parámetro de interés, digamos un vector (\mathbf{x}), empujándolo en la dirección del gradiente negativo (\mathbf{g}) (en descenso de gradiente estocástico, calculamos este gradiente en un minibatch muestreado aleatoriamente). Por ejemplo, con una tasa de aprendizaje (\eta > 0), cada actualización toma la forma (\mathbf{x} \gets \mathbf{x} - \eta \mathbf{g}). Supongamos además que la función objetivo (f) es suficientemente suave. Formalmente, decimos que el objetivo es *continuo de Lipschitz* con constante (L), lo que significa que para cualquier (\mathbf{x}) y (\mathbf{y}), tenemos

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|.$$

Como puedes ver, cuando actualizamos el vector de parámetros restando (\eta \mathbf{g}), el cambio en el valor del objetivo depende de la tasa de aprendizaje, la norma del gradiente y (L) de la siguiente manera:

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|.$$

En otras palabras, el objetivo no puede cambiar más de (L \eta |\mathbf{g}|). Tener un valor pequeño para este límite superior podría ser visto como bueno o malo. Por un lado, estamos limitando la velocidad a la que podemos reducir el valor del objetivo. Por otro lado, esto limita cuánto podemos equivocarnos en un solo paso de gradiente.

Cuando decimos que los gradientes explotan, nos referimos a que (|\mathbf{g}|) se vuelve excesivamente grande. En el peor de los casos, podríamos hacer tanto daño en un solo paso de gradiente que podríamos deshacer todo el progreso realizado durante miles de iteraciones de entrenamiento. Cuando los gradientes pueden ser tan grandes, el entrenamiento de redes neuronales a menudo diverge, sin lograr reducir el valor del objetivo. Otras veces, el entrenamiento eventualmente converge pero es inestable debido a picos masivos en la pérdida.

Una forma de limitar el tamaño de (L \eta |\mathbf{g}|) es reducir la tasa de aprendizaje (\eta) a valores muy pequeños. Esto tiene la ventaja de que no sesgamos las actualizaciones. Pero ¿y si solo *raramente* obtenemos grandes gradientes? Este drástico movimiento ralentiza nuestro progreso en todos los pasos, solo para lidiar con los raros eventos de explosión de gradientes. Una alternativa popular es adoptar una heurística de *recorte de gradientes* proyectando los gradientes (\mathbf{g}) sobre una bola de algún radio dado (\theta) de la siguiente manera:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}.$$

Esto garantiza que la norma del gradiente nunca exceda (\theta) y que el gradiente actualizado esté completamente alineado con la dirección original de (\mathbf{g}). También tiene el efecto secundario deseable de limitar la influencia que cualquier minibatch dado (y dentro de él cualquier muestra dada) puede ejercer sobre el vector de parámetros. Esto otorga cierto grado de robustez al modelo. Para ser claros, es un truco. El recorte de gradientes significa que no siempre estamos siguiendo el verdadero gradiente y es difícil razonar analíticamente sobre los posibles efectos secundarios. Sin embargo, es un truco muy útil, y es ampliamente adoptado en implementaciones de RNN en la mayoría de los frameworks de aprendizaje profundo.

A continuación, definimos un método para recortar gradientes, que es invocado por el método `fit_epoch` de la clase `d2l.Trainer` (ver Sección 3.4). Nota que al calcular la norma del gradiente, estamos concatenando todos los parámetros del modelo, tratándolos como un único vector de parámetros gigante.

In [28]:
```
my $clip_gradients = sub{
my ($self, $grad_clip_val, $model) = @_;
    my $params = $model->parameters();
    if (ref $params ne 'ARRAY') {
        $params = map { $_->data() } values %{$params};
    }

    my $norm = 0;
    foreach my $param (@$params) {
        $norm += ($param->grad() ** 2)->sum();
    }
    $norm = sqrt($norm);

    if ($norm > $grad_clip_val) {
        foreach my $param (@$params) {
            $param->grad()->[':'] *= $grad_clip_val / $norm;
        }
    }
};
d2l->add_to_class('d2l::Trainer', 'clip_gradients', $clip_gradients);

```

Out[28]:
```
*d2l::Trainer::clip_gradients

Subroutine d2l::Trainer::clip_gradients redefined at /usr/local/lib/perl5/sit
e_perl/5.32.1/x86_64-linux/d2l.pm line 4456.
```

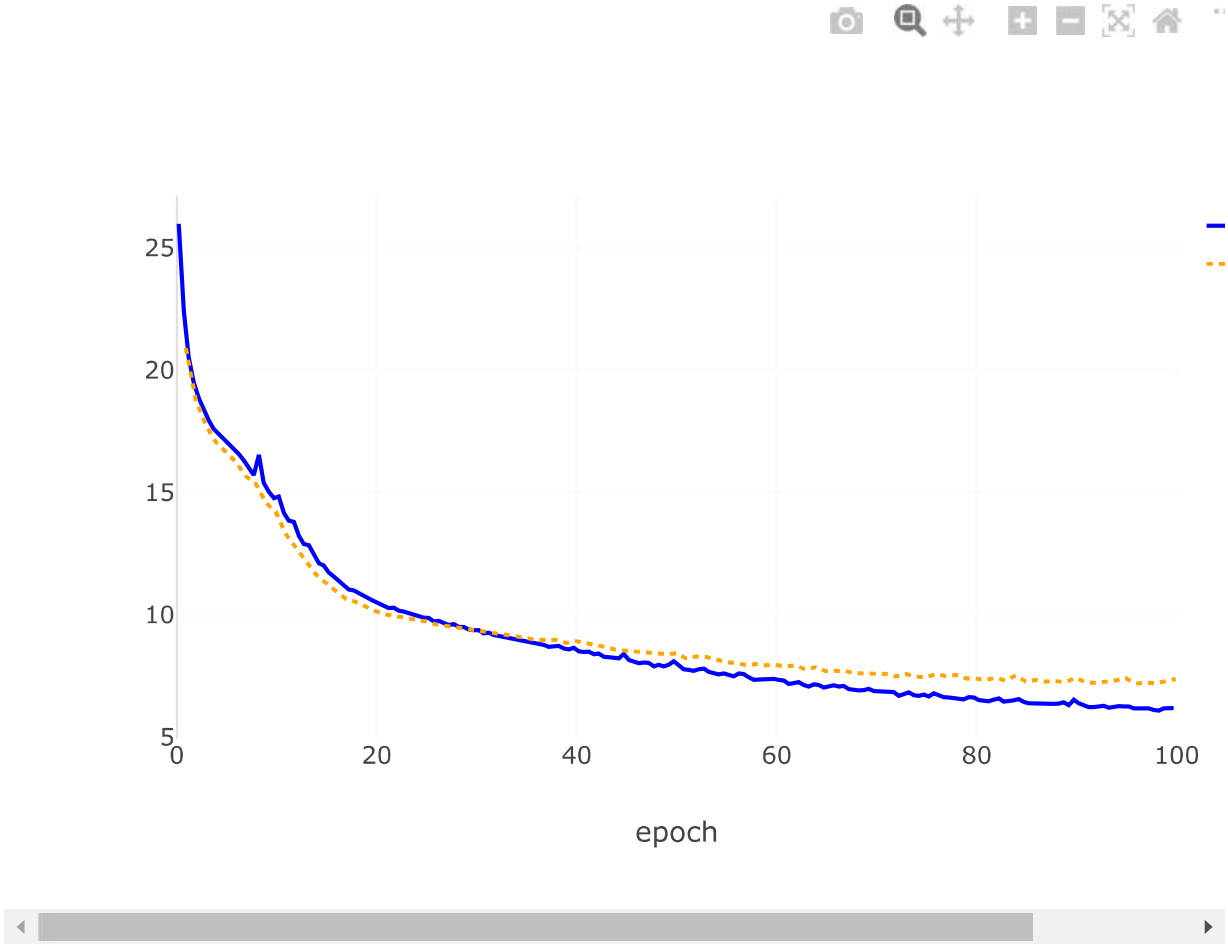
9.5.4. Entrenamiento

Usando el conjunto de datos de *La Máquina del Tiempo* (`data`), entrenamos un modelo de lenguaje a nivel de caracteres (`model`) basado en el RNN (`rnn`) implementado desde cero. Ten en cuenta que primero calculamos los gradientes, luego los recortamos y finalmente actualizamos los parámetros del modelo usando los gradientes recortados.

In [29]:
```
my $data = d2l::TimeMachine->new(batch_size => 1024, num_steps => 32);
my $rnn = RNNScratch->new(num_inputs => $data->{vocab}->len, num_hiddens =
my $model = RNNLMScratch->new(rnn => $rnn, vocab_size => $data->{vocab}->l
my $trainer = d2l::Trainer->new(max_epochs => 100, gradient_clip_val => 1,
$trainer->fit($model, $data);
```

```
Training time: 21:14

No GPU support.
```




```
Argument ":" isn't numeric in array element at reply input line 16.
```

9.5.5. Descodificación

Una vez que se ha aprendido un modelo de lenguaje, podemos usarlo no solo para predecir el siguiente token sino para continuar prediciendo cada uno de los siguientes, tratando el token previamente predicho como si fuera el siguiente en la entrada. A veces solo querremos generar texto como si estuviéramos comenzando al principio de un documento. Sin embargo, a menudo es útil condicionar el modelo de lenguaje en un prefijo proporcionado por el usuario. Por ejemplo, si estuviéramos desarrollando una función de autocompletar para un motor de búsqueda o para ayudar a los usuarios a escribir correos electrónicos, querríamos introducir lo que han escrito hasta ahora (el prefijo), y luego generar una continuación probable.

El siguiente método `predict` genera una continuación, un carácter a la vez, después de procesar un `prefijo` proporcionado por el usuario. Al recorrer los caracteres en `prefijo`, seguimos pasando el estado oculto al siguiente paso de tiempo pero no generamos ninguna salida. Esto se llama el período de *calentamiento*. Después de procesar el prefijo, ahora estamos listos para comenzar a emitir los caracteres siguientes, cada uno de los cuales será

In [30]:
```
my $predict = sub {
    my ($self, $prefix, $num_preds, $vocab, $device) = @_;

    my ($state, @outputs) = (undef, $vocab->{$prefix->[0]});

    for my $i (0..(length($prefix) + $num_preds - 1)) {

        my $X = mx->nd->array([[$outputs[-1]]], ctx => $device);

        my $embs = $self->one_hot($X);

        my ($rnn_outputs, $state) = $self->{rnn}($embs, $state);

        if ($i < length($prefix) - 1) {

            #no funciona
            #prefix solo debe ser una letra
            print "Value of prefix: ", ($prefix->[$i + 1]);
            #push @outputs, $vocab->getitem($prefix->[$i + 1]);
            last;



        } else {

            my $Y = $self->output_layer($rnn_outputs);
            print "Value of \$Y: ", ($Y);
            my $index = int($Y->argmax(axis => 2)->reshape(1));

            push @outputs, $index;
}

    }
    #no funciona

    my @tokens = map { $vocab->{idx_to_token}->[$_] // '<unk>' } @outputs;


    return join('', @tokens);
};

d2l->add_to_class('RNNLMScratch', 'predict', $predict);
```

Out[30]:
```
*RNNLMScratch::predict

Subroutine RNNLMScratch::predict redefined at /usr/local/lib/perl5/site_perl/
5.32.1/x86_64-linux/d2l.pm line 4456.
```

In [31]:
```
$model->predict(['it', 'has'], 20, $data->{vocab}, d2l::try_gpu());
```

```
Value of prefix: has
```

Out[31]:

```
Use of uninitialized value $_ in array element at reply input line 36.
```

Aunque implementar el modelo RNN anterior desde cero es instructivo, no es conveniente. En la siguiente sección, veremos cómo aprovechar los frameworks de aprendizaje profundo para crear RNNs usando arquitecturas estándar y obtener mejoras en el rendimiento al depender de funciones de biblioteca altamente optimizadas.

Resumen

Podemos entrenar modelos de lenguaje basados en RNN para generar texto siguiendo el prefijo de texto proporcionado por el usuario. Un modelo de lenguaje RNN simple consiste en codificación de entrada, modelado RNN y generación de salida. Durante el entrenamiento, el recorte de gradientes puede mitigar el problema de gradientes que explotan, pero no aborda el problema de gradientes que desaparecen. En el experimento, implementamos un modelo de lenguaje RNN simple y lo entrenamos con recorte de gradientes en secuencias de texto, tokenizadas a nivel de carácter. Condicionando en un prefijo, podemos usar un modelo de lenguaje para generar continuaciones probables, lo cual resulta útil en muchas aplicaciones, por ejemplo, funciones de autocompletar.

Ejercicios

1. ¿El modelo de lenguaje implementado predice el siguiente token basado en todos los tokens pasados hasta el primer token en *La Máquina del Tiempo*?
2. ¿Qué hiperparámetro controla la longitud de la historia utilizada para la predicción?
3. Demuestra que la codificación one-hot es equivalente a elegir un embedding diferente para cada objeto.
4. Ajusta los hiperparámetros (por ejemplo, número de épocas, número de unidades ocultas, número de pasos de tiempo en un minibatch y tasa de aprendizaje) para mejorar la perplejidad. ¿Hasta dónde puedes llegar manteniéndote con esta arquitectura simple?
5. Reemplaza la codificación one-hot con embeddings aprendibles. ¿Esto conduce a un mejor rendimiento?
6. Realiza un experimento para determinar qué tan bien este modelo de lenguaje entrenado en *La Máquina del Tiempo* funciona en otros libros de H. G. Wells, por ejemplo, *La Guerra de los Mundos*.
7. Realiza otro experimento para evaluar la perplejidad de este modelo en libros escritos por otros autores.
8. Modifica el método de predicción para usar muestreo en lugar de elegir el siguiente carácter más probable.
   - ¿Qué sucede?
   - Sesga el modelo hacia salidas más probables, por ejemplo, muestreando de (q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha) para (\alpha >

1).

9. Ejecuta el código de esta sección sin recortar el gradiente. ¿Qué pasa?
10. Reemplaza la función de activación utilizada en esta sección con ReLU y repite los experimentos de esta sección. ¿Todavía necesitamos recortar el gradiente? ¿Por qué?