BAB 6:

RNN: Modeling Sequential Data Dengan Menggunakan Recurrent Neural Networks

Pada bab sebelumnya, kita fokus pada Convolutional Neural Networks (CNNs) untuk klasifikasi citra. Dalam bab ini, kita akan menjelajahi Recurrent Neural Networks (RNNs) dan melihat penerapannya dalam memodelkan data sekuensial dan subset tertentu dari data sekuensial—data deret waktu. Sebagai rangkuman, dalam bab ini, kita akan membahas topik-topik berikut:

> 1. Pengenalan data sekuensial
> 2. RNNs untuk pemodelan sekuensial
> 3. Long Short-Term Memory (LSTM)
> 4. Truncated Backpropagation Through Time (T-BPTT)
> 5. Implementasi a multilayer RNN untuk sekuensial—modeling di TensorFlow
> 6. Proyek satu: Analisis sentimen RNN dari kumpulan data ulasan film IMDb
> 7. Proyek dua: pemodelan bahasa tingkat karakter RNN dengan sel LSTM, menggunakan data teks dari The Mysterious Island karya Jules Verne
> 8. Menggunakan pemotongan gradien untuk menghindari ledakan gradien
> 9. Memperkenalkan model Transformer dan memahami mekanisme perhatian diri

A. Pengenalan data sekuensial

Mari kita mulai diskusi tentang RNN dengan melihat sifat data sekuensial, lebih dikenal sebagai data sekuens atau **sequences**. Kita akan melihat sifat unik dari sequences/urutan yang membuatnya berbeda dari jenis data lainnya. Kita kemudian akan melihat bagaimana kita dapat merepresentasikan data sekuensial, dan menjelajahi berbagai kategori model untuk data sekuensial, yang didasarkan pada input dan output model. Ini akan membantu kita menjelajahi hubungan antara RNN dan sekuens nanti di bab ini.

A.1 Modeling data sekuensial – menjadi hal yang penting

Apa yang membuat sequences unik, dibandingkan dengan jenis data lainnya, adalah bahwa elemen dalam sekuensial muncul dalam urutan tertentu dan tidak terpisah satu sama lain. Algoritme pembelajaran mesin tipikal untuk supervised learning mengasumsikan bahwa input adalah data independen dan terdistribusi secara identik (independent and identically distributed (IID)), yang berarti bahwa training example saling independen dan memiliki distribusi dasar yang sama. Dalam hal ini, berdasarkan asumsi independen bersama, sekuensial dari training example yang diberikan ke model tidak relevan. Misalnya, jika kita memiliki sampel data yang terdiri dari n training example, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, urutan penggunaan data untuk melatih algoritme pembelajaran mesin kita tidak masalah . Contoh skenario ini adalah kumpulan data Iris yang sebelumnya kita kerjakan. Dalam dataset Iris, setiap bunga telah diukur secara independen, dan pengukuran satu bunga tidak memengaruhi pengukuran bunga lainnya.

Namun, asumsi ini tidak valid saat kita berurusan dengan data sekuensial—menurut definisi, urutan itu penting. Memprediksi nilai pasar saham tertentu akan menjadi contoh skenario ini. Misalnya, anggaplah kita memiliki sampel n dari training example, di mana setiap dari training example mewakili nilai pasar saham tertentu pada hari tertentu. Jika tugas kita adalah memprediksi nilai pasar saham untuk tiga hari ke depan, akan menjadi benar jika mempertimbangkan harga saham sebelumnya urutan tanggal untuk mendapatkan tren daripada menggunakan training example ini dalam urutan acak.

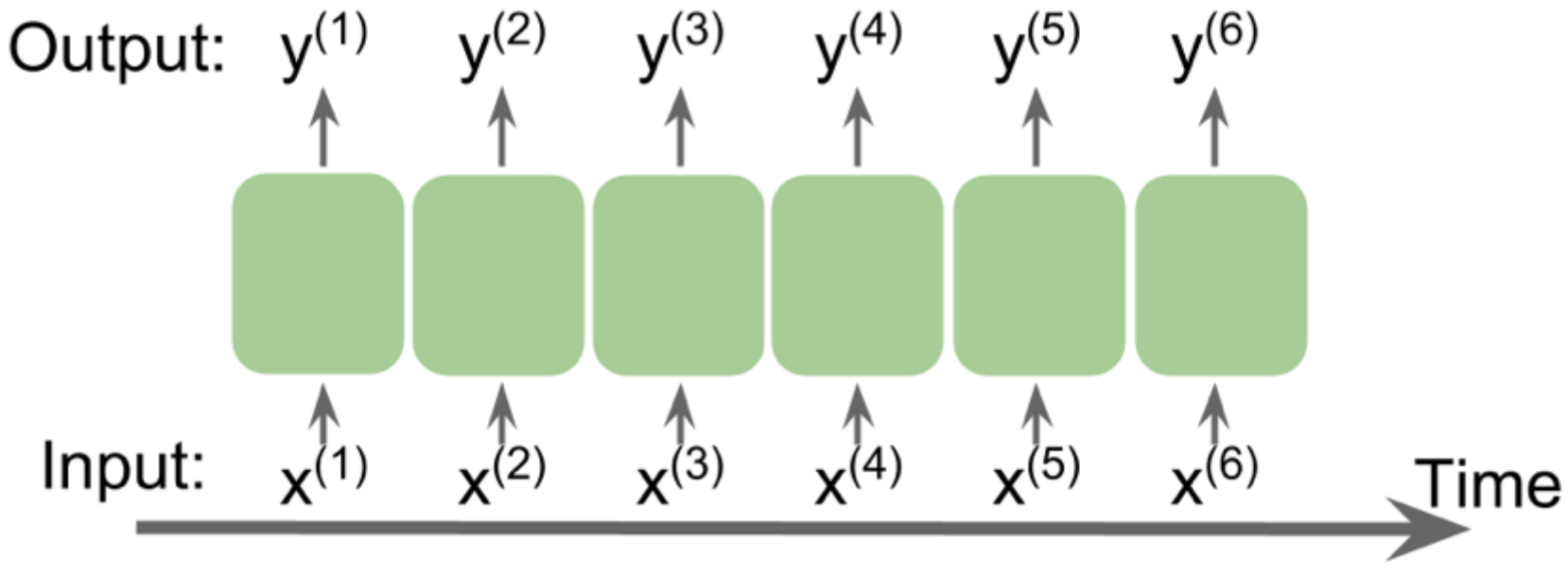
NOTE (data sekuensial versus time-series data)

time-series data adalah tipe khusus dari data sekuensial, di mana setiap example data dikaitkan dengan dimensi waktu. Dalam time-series data, sampel diambil pada stamps waktu yang berurutan, dan oleh karena itu, dimensi waktu menentukan urutan di antara titik data. Misalnya, harga saham dan voice atau speech records adalah time-series data.

Di sisi lain, tidak semua data sekuensial memiliki dimensi waktu, misalnya data teks atau sekuens DNA, di mana data examplenya diurutkan tetapi tidak memenuhi syarat sebagai time-series data. Seperti yang akan Anda lihat, dalam bab ini, kita akan membahas beberapa contoh pemrosesan bahasa alami (NLP) dan pemodelan teks yang bukan merupakan data deret waktu, tetapi perhatikan bahwa RNN juga dapat digunakan untuk time-series data.

A.2 Representing sekuensial (Representasi sequences)

kita telah menetapkan bahwa urutan data di antara titik data penting dalam sekuensial data, jadi selanjutnya kita perlu menemukan cara untuk memanfaatkan informasi urutan ini dalam model pembelajaran mesin. Sepanjang bab ini, kita akan merepresentasikan sekuensial data sebagai $(x^{(1)}, x^{(2)}, \ldots, x^{(T)})$. Indeks superskrip menunjukkan urutan instances, dan panjang sekuensial adalah T. Untuk contoh urutan yang masuk akal, pertimbangkan data deret waktu, di mana setiap titik example data, $x^{(t)}$, milik waktu tertentu, t. Gambar berikut menunjukkan example time-series data. dengan fitur input (**x's**) dan label target (**y's**) secara alami mengikuti urutan sesuai dengan sumbu waktunya; oleh karena itu, baik x dan y adalah sekuensial:

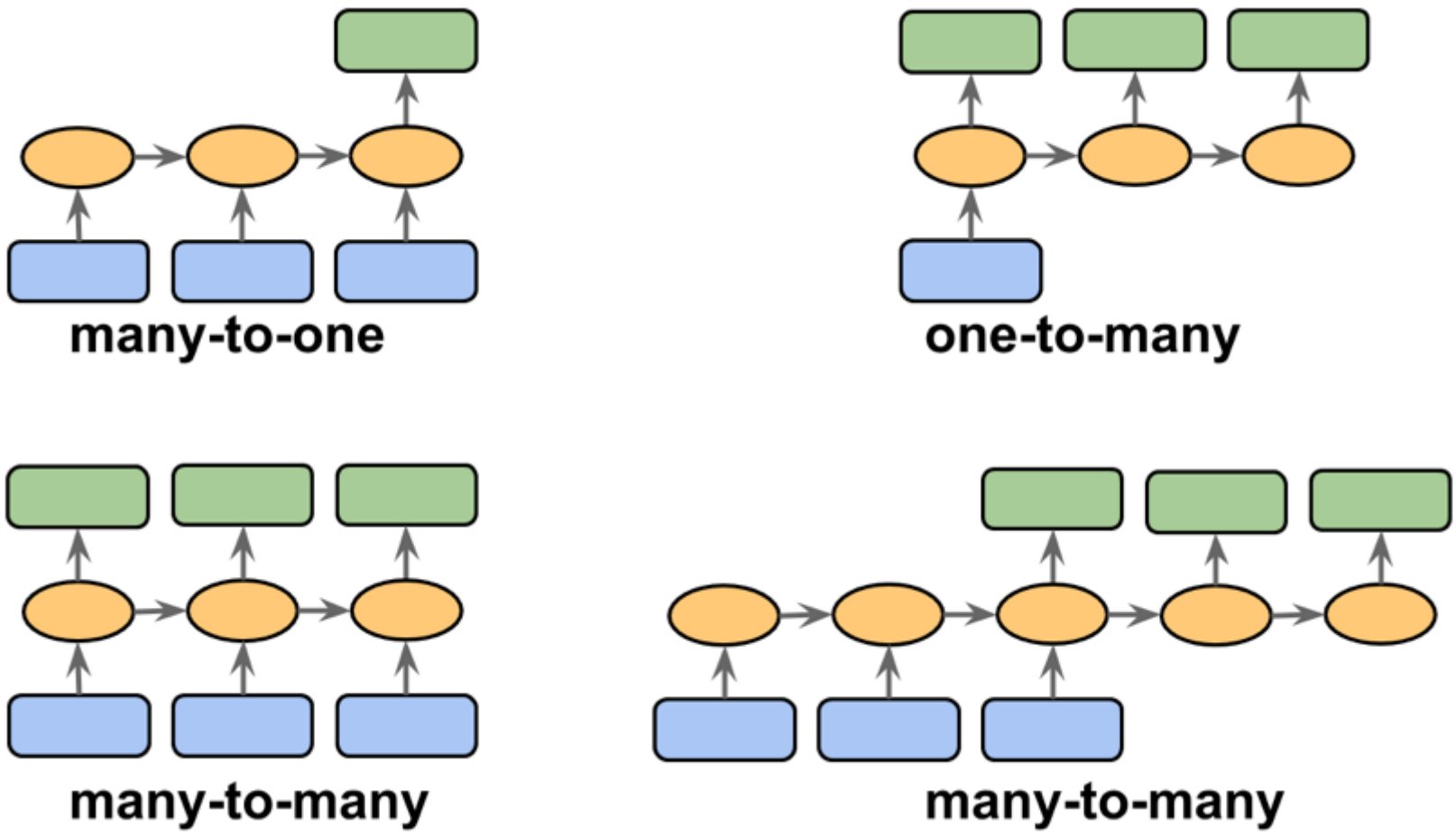


Seperti yang telah kita sebutkan, model neural network standar (NN) yang telah kita bahas sejauh ini, seperti multilayer perceptron (MLP) dan CNN untuk data gambar, mengasumsikan bahwa training example tidak tergantung satu sama lain dan dengan demikian tidak menyertakan pengurutan informasi. Kita dapat mengatakan bahwa model seperti itu tidak memiliki ingatan tentang training example yang terlihat sebelumnya. Misalnya, sampel dilewatkan melalui langkah feedforward dan backpropagation, dan bobot diperbarui secara terpisah dari urutan pemrosesan training example.

Sebaliknya, RNN dirancang untuk memodelkan urutan dan mampu mengingat informasi masa lalu dan memproses peristiwa baru sesuai dengan urutan data tersebut, yang merupakan keuntungan jelas saat bekerja dengan data sekuens.

A.3 Berbagai kategori pemodelan sekuensial

Pemodelan sekuensial memiliki banyak aplikasi yang menarik, seperti terjemahan bahasa (mungkin dari bahasa Inggris ke bahasa Jerman), keterangan gambar (image captioning), dan pembuatan teks (text generator).Namun, kita perlu memahami berbagai jenis tugas pemodelan sekuensial untuk mengembangkan model yang sesuai. Gambar berikut, berdasarkan penjelasan dalam artikel bagus The Unreasonable Effectiveness of Recurrent Neural Networks oleh Andrej Karpathy (http://karpathy.github.io/2015/05/21/rnn-effectiveness/), menunjukkan beberapa kategori hubungan yang berbeda input dan output data:



Mari kita lihat kategori hubungan yang berbeda antara data input dan output, yang digambarkan pada gambar sebelumnya, secara lebih rinci. Jika baik data input maupun output tidak merepresentasikan urutan, maka kita berurusan dengan data standar, dan kita cukup menggunakan perceptron multilayer (atau model klasifikasi lain yang sebelumnya dibahas dalam buku ini) untuk memodelkan data tersebut. Namun, jika input atau outputnya adalah sekuensial, tugas pemodelan kemungkinan termasuk dalam salah satu kategori berikut:

> **Many-to-one**: Data input adalah sekuensial, tetapi outputnya adalah vektor atau skalar ukuran tetap, bukan sekuensial. Misalnya, dalam analisis sentimen, inputnya berbasis teks (misalnya, ulasan film) dan outputnya adalah label kelas (misalnya, label yang menunjukkan apakah reveiewer menyukai film tersebut).

> **One-to-many**: Data input dalam format standar dan bukan urutan, tetapi outputnya adalah sekuensial. Contoh dari kategori ini adalah keterangan gambar— sebagai input adalah sebuah gambar dan outputnya adalah frase bahasa Inggris yang meringkas konten gambar tersebut.

> **Many-to-many**: Array input dan output adalah urutan. Kategori ini dapat dibagi lebih lanjut berdasarkan apakah input dan output disinkronkan. Contoh tugas pemodelan many-to-many yang disinkronkan adalah klasifikasi video, di mana setiap frame dalam video diberi label. Contoh tugas pemodelan Many-to-many yang terdapat penundaan output adalah menerjemahkan satu bahasa ke bahasa

> lain. Misalnya, seluruh kalimat bahasa Inggris harus dibaca dan diproses oleh mesin sebelum terjemahannya ke dalam bahasa Jerman dihasilkan.

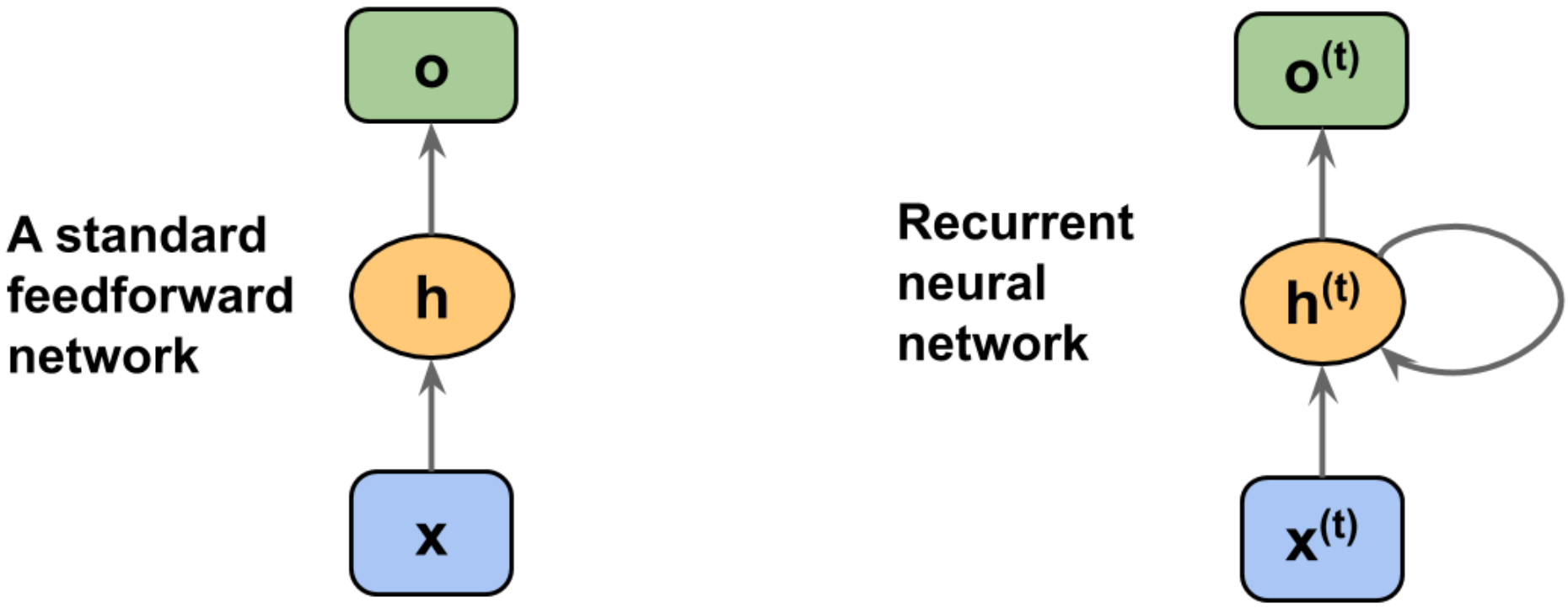
Sekarang, karena kita mengetahui tentang kategori Pemodelan sekuensial, kita dapat melanjutkan untuk membahas struktur RNN.

B. RNNs untuk pemodelan sekuensial

Di bagian ini, sebelum kita mulai mengimplementasikan RNN di TensorFlow, kita akan membahas konsep utama RNN. Kita akan mulai dengan melihat struktur standar dari sebuah RNN, yang menyertakan komponen rekursif untuk memodelkan sekuensial data. Kemudian, kita akan memeriksa bagaimana aktivasi neuron dihitung yang biasanya didalam RNN. Ini akan menciptakan konteks bagi kita untuk membahas tantangan umum dalam melatih RNN, dan kita kemudian akan membahas solusi untuk tantangan metode pengembangan lainnya, seperti LSTM dan gated recurrent units (GRUs).

B.1 Memahami struktur dan flow dari RNN

Mari kita mulai dengan memperkenalkan arsitektur RNN. Gambar berikut menunjukkan feedforward neural network standar dan RNN, diletakan berdampingan sebagai perbandingan:



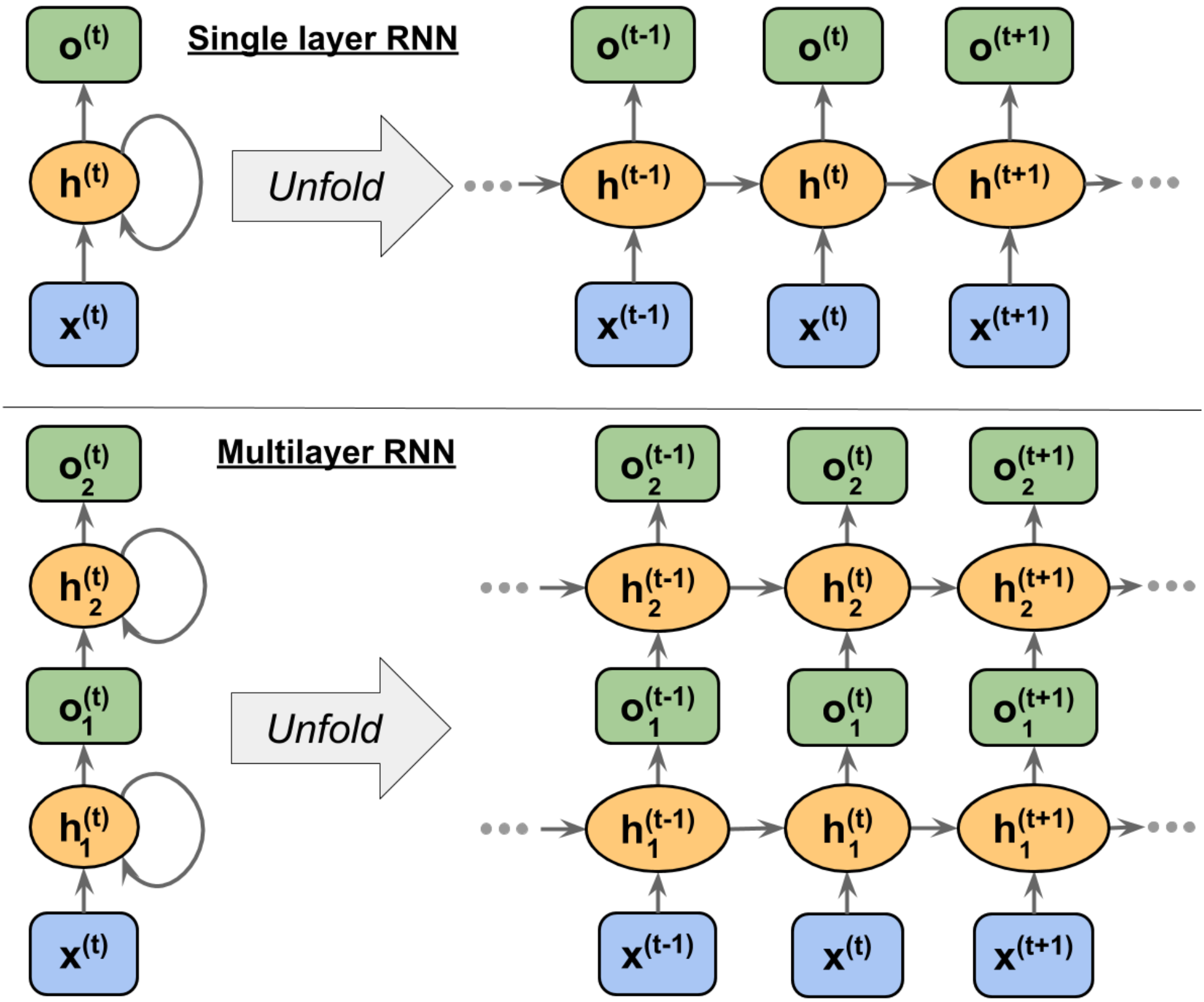
Kedua network ini hanya memiliki satu hidden layer. Dalam representasi ini, unit tidak ditampilkan (dances/jumlah neuron), tetapi diasumsikan bahwa input layer (x), hidden layer (h), dan output layer (o) adalah vektor yang memiliki banyak unit.

NOTE (Menentukan tipe output dari sebuah RNN) Menentukan jenis output dari RNN Arsitektur RNN generik ini dapat sesuai dengan dua kategori pemodelan sekuansial di mana inputnya adalah sekuansial data. Biasanya, recurrent layer dapat mengembalikan sekuensial data sebagai output, $\langle o^{(0)}, o^{(1)}, \ldots, o^{(T)} \rangle$, atau hanya mengembalikan output terakhir (pada t = T, yaitu, $o^{(T)}$). Jadi, bisa jadi many-to-many, atau bisa jadi many-to-one jika, misalnya, kita hanya menggunakan elemen terakhir, $o^{(T)}$, sebagai hasil akhir. Seperti yang akan Anda lihat nanti, di TensorFlow Keras API, perilaku recurrent layer berkorelaso dengan mengembalikan sekuensial sebagai output atau cukup menggunakan output terakhir dapat ditentukan dengan menyetel argumen return_sequences ke True atau False.

Dalam feedforward network standar, informasi mengalir dari input ke hidden layer, dan kemudian dari hidden layer ke output layer. Di sisi lain, dalam RNN, hidden layer menerima masukan dari input layer saat ini dan hidden layer juga menerima masukan dari input layer waktu sebelumnya.

Aliran informasi dalam langkah waktu yang berdekatan di hidden layer memungkinkan network untuk memiliki memori peristiwa masa lalu. Aliran informasi ini biasanya ditampilkan sebagai loop, juga dikenal sebagai tepi berulang (edge recurrent) dalam notasi grafik, yang merupakan asal nama arsitektur RNN umum ini.

Mirip dengan perceptron multilayer, RNN dapat terdiri dari beberapa hidden layer. Perhatikan bahwa sudah menjadi konvensi umum untuk mengacu RNN dengan satu hidden layer sebagai RNN single layer, yang mana agar tidak membingungkan dengan NN single layer tanpa hidden layer, seperti Adaline atau regresi logistik. Gambar berikut mengilustrasikan RNN dengan satu hidden layer (atas) dan RNN dengan dua lapisan tersembunyi (bawah):



Untuk memeriksa arsitektur RNN dan aliran informasi, representasi kompak dengan recurrent edge dapat dijabarkan, yang dapat dilihat pada gambar sebelumnya.

Seperti yang kita ketahui, setiap hidden unit dalam neural network standar hanya menerima satu input — net preaktivasi yang terkait dengan input layer. Sekarang, sebaliknya, setiap hidden layer di RNN menerima dua set input yang berbeda—preaktivasi dari lapisan input dan aktivasi hidden layer yang sama dari langkah waktu proses sebelumnya t-1.

Pada langkah pertama $t = 0$, hidden unit diinisialisasi ke nol atau nilai acak kecil. Kemudian, pada langkah waktu di mana t > 0, hidden unit mendapatkan inputnya dari titik data pada waktu saat ini $x^{(t)}$ dan nilai hidden unit sebelumnya pada $t - 1$, ditunjukkan sebagai $h^{(t-1)}$.

Demikian pula, dalam kasus RNN multilayer, kita dapat meringkas aliran informasi sebagai berikut:

> $layer = 1$: Untuk ini, hidden layer direpresentasikan sebagai $h_1^{(t)}$ dan mendapatkan inputnya dari titik data $x^{(t)}$ dan nilai hasil hidden layer di lapisan yang sama, tetapi dari proses sebelumnya $h_1^{(t-1)}$

> $layer = 2$: hidden layer kedua, menerima inputnya dari hidden unit dari layer di bawahnya pada langkah waktu saat ini ($o_1^{(t)}$) dan nilai hasil process hidden sendiri dari step proses waktu sebelumnya $h_2^{(t-1)}$

Karena, dalam hal ini, setiap recurrent layer harus menerima sekuensia data sebagai input, semua recurrent layer kecuali yang terakhir harus mengembalikan sekuensial sebagai output (yaitu, return_sequences=True). Perilaku lapisan berulang terakhir bergantung pada jenis masalahnya.

B.2 Menghitung aktivasi dalam RNN

Sekarang setelah kita memahami struktur dan flow umum informasi dalam RNN, mari kita lebih spesifik dan menghitung actual aktivasi dari hidden layer serta output layer. Untuk menyederhanakan, kita akan mempertimbangkan hanya satu hidden layer; namun, konsep yang sama berlaku untuk RNN multilayer.

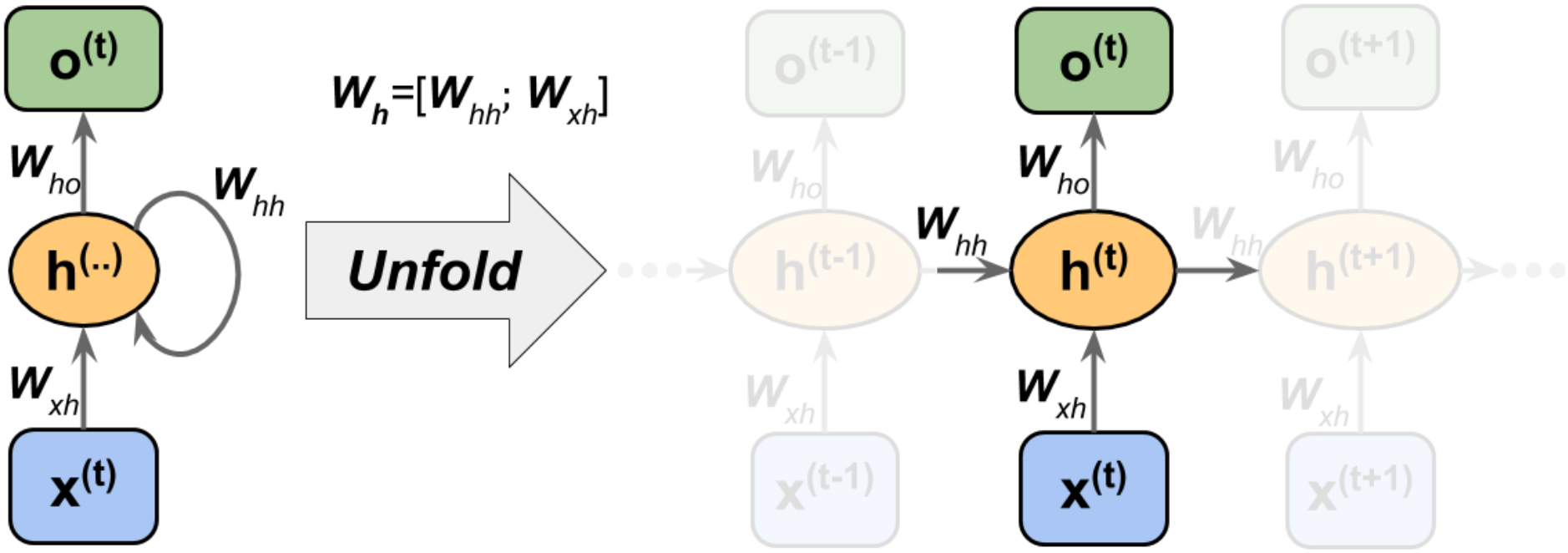
Setiap tepi (edge) secara langsung (koneksi antar kotak) dalam representasi RNN yang baru saja kita lihat dikaitkan dengan matriks bobot (weight matrix). Bobot tersebut tidak bergantung pada waktu t; oleh karena itu, bobot-bobot tersebut dibagi sepanjang interval waktu. Matriks bobot yang berbeda dalam satu lapisan RNN adalah sebagai berikut:

> W_{xh}: Matriks bobot antara input $x^{(t)}$ dan hidden layer **h**

> W_{hh}: Matriks bobot yang terkait dengan tepi berulang /**recurrent edge**

> W_{h0}: Matriks bobot antara hidden layer dan output layer.

Dapat melihat matriks bobot ini pada gambar berikut:



Dalam implementasi tertentu, kita bisa mengamati bahwa matriks bobot (weight matrice) W_{xh} dan W_{hh} digabungkan menjadi matriks gabungan $W_h = [W_{xh}; W_{hh}]$. Nanti, kita akan menggunakan notasi ini juga.

Menghitung aktivasi sangat mirip dengan perceptron multilayer standar dan jenis feedforward neural network lainnya. Untuk hidden layer, net input (preaktivasi) dihitung melalui kombinasi linier. Yaitu, Kita menghitung jumlah perkalian matriks bobot dengan vektor yang sesuai dan menambahkan unit bias

$$z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h$$

Kemudian, aktivasi hidden unit pada prosess saat ini waktu t dihitung sebagai berikut:

$$h^{(t)} = \phi(z_h^{(t)}) = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

Di sini, b_h adalah vektor bias untuk hidden unit dan $\phi_h(\cdot)$ merupakan fungsi aktivasi hidden layer.

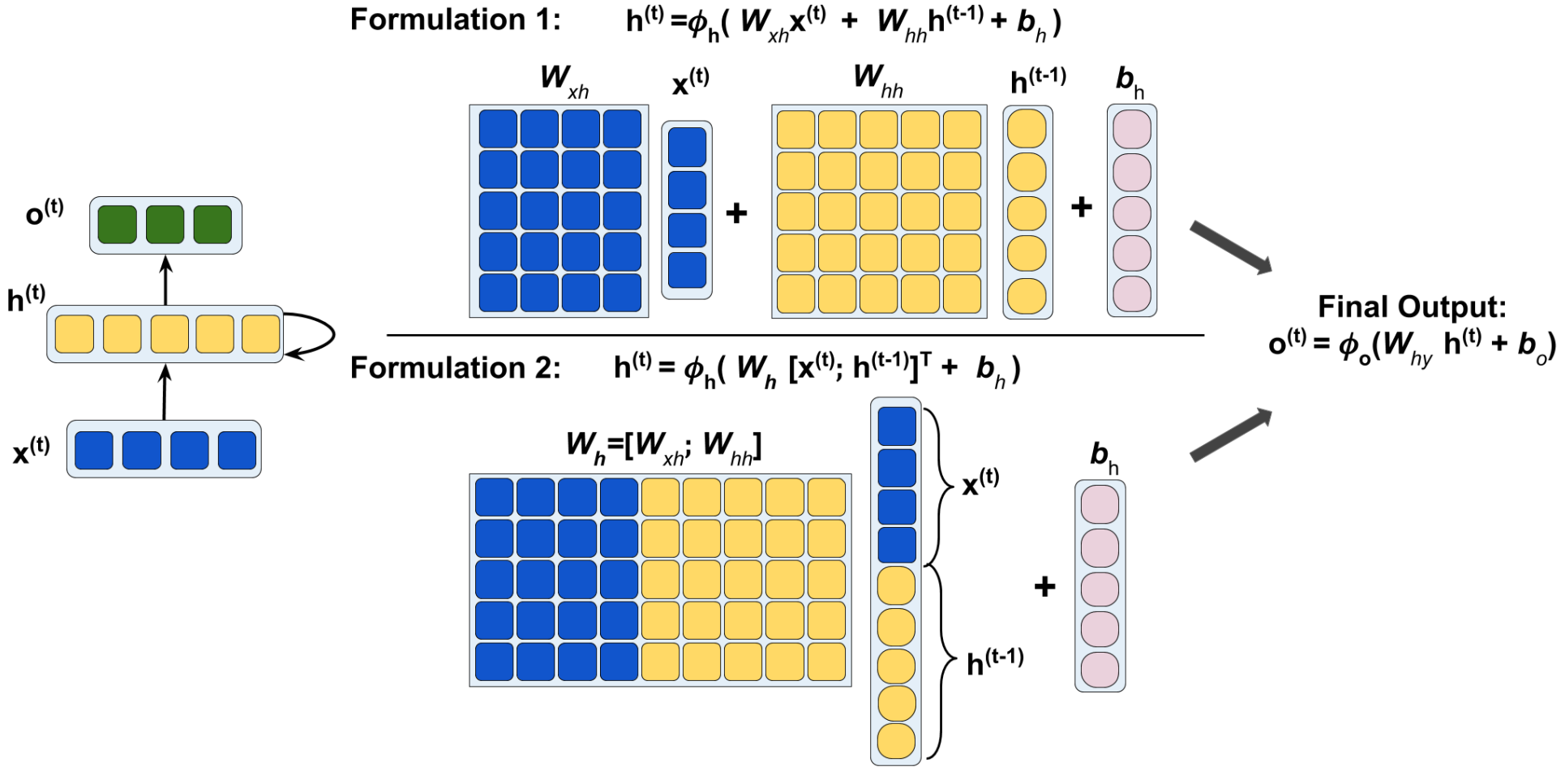
Jika kita ingin menggunakan matriks bobot gabungan (concatenated weight matrix) $W_h = [W_{xh}; W_{hh}]$, rumus untuk menghitung hidden unit akan berubah sebagai berikut:

$$h^{(t)} = \phi_h\left([W_{xh}; W_{hh}] \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h\right)$$

Setelah aktivasi hidden unit pada step waktu saat ini dihitung, maka aktivasi unit keluaran akan dihitung sebagai berikut:

$$o^{(t)} = \phi_o(W_{ho}h^{(t)} + b_y)$$

Untuk membantu memperjelas hal ini lebih lanjut, gambar berikut menunjukkan proses penghitungan aktivasi ini dengan kedua formulasi tersebut:



NOTE

Training RNNs dengan BPTT (Backpropagation Through Time) Algoritme pembelajaran untuk RNN diperkenalkan pada tahun 1990-an Backpropagation Through Time: Apa yang Dilakukannya dan Bagaimana Melakukannya (Paul Werbos, Prosiding IEEE, 78(10):1550-1560, 1990). Derivasi dari gradien mungkin sedikit rumit, tetapi ide dasarnya adalah bahwa loss keseluruhan L adalah jumlah dari semua fungsi loss pada waktu $t = 1$ sampai $t = T$:

$$L = \sum_{t=1}^{T} L^{(t)}$$

Karena loss pada waktu t bergantung pada hidden unit di semua langkah waktu sebelumnya $1 : t$, gradien akan dihitung sebagai berikut:

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \times \frac{\partial o^{(t)}}{\partial h^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \times \frac{\partial h^{(k)}}{\partial W_{hh}} \right)$$

Di sini $\frac{\partial h^{(t)}}{\partial h^{(k)}}$, dihitung sebagai perkalian langkah waktu yang berdekatan:

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{(i=k+1)}^{t} \frac{\partial h^{(i)}}{\partial h^{(i-1)}}$$

B.3 Hidden-recurrence versus output-recurrence

Sejauh ini, Anda telah melihat recurrent networks di mana hidden layer memiliki properti berulang. Namun, perhatikan bahwa ada model alternatif di mana recurrent connection berasal dari output layer. Dalam hal ini, net aktivasi dari output layer pada langkah waktu sebelumnya, $o^{(t-1)}$, dapat ditambahkan dengan salah satu dari dua cara:

> - Ke lapisan tersembunyi pada langkah waktu saat ini, $h^{(t)}$ (ditunjukkan pada gambar berikut sebagai pengulangan output-to-hidden)
> - Ke lapisan keluaran pada langkah waktu saat ini, $o^{(t)}$(ditunjukkan pada gambar berikut sebagai perulangan output-to-output)

Seperti yang ditunjukkan pada gambar sebelumnya, perbedaan antara arsitektur ini dapat dilihat dengan jelas pada koneksi yang berulang. Mengikuti notasi yang kita gunakan, bobot yang terkait dengan koneksi berulang akan dilambangkan untuk hidden-to-hidden oleh W_{hh}, untuk pengulangan

output-to-hidden oleh W_{oh} , dan untuk pengulangan output-to-output oleh W_{oh}. Dalam beberapa artikel dalam literatur, bobot yang terkait dengan koneksi berulang juga dilambangkan dengan W_{rec}.

Untuk melihat cara kerjanya dalam aplikasi, mari hitung secara manual forward pass untuk salah satu dari tipe berulang ini. Menggunakan TensorFlow Keras API, recurrent layer dapat ditentukan melalui Simple RNN, yang serupa dengan output-ke-output. Dalam kode berikut, kita akan membuat layer berulang dari SimpleRNN dan melakukan forward pass pada input sequence dengan panjang 3 untuk menghitung output. kita juga akan menghitung forward pass secara manual dan membandingkan hasilnya dengan SimpleRNN. Pertama, mari buat layer dan tetapkan bobot untuk perhitungan manual kita:

In [ ]:
```
import tensorflow as tf

tf.random.set_seed(1)

rnn_layer = tf.keras.layers.SimpleRNN( units=2, use_bias=True, return_sequences=True)
rnn_layer.build(input_shape=(None, None, 5))

w_xh, w_oo, b_h = rnn_layer.weights
print('W_xh shape:', w_xh.shape)
print('W_oo shape:', w_oo.shape)
print('b_h shape:', b_h.shape)
```

```
Metal device set to: Apple M1

systemMemory: 8.00 GB
maxCacheSize: 2.67 GB

W_xh shape: (5, 2)
W_oo shape: (2, 2)
b_h shape: (2,)
```

Bentuk input untuk lapisan ini adalah (None, None, 5), di mana dimensi pertama adalah dimensi batch (menggunakan None untuk ukuran batch variabel), dimensi kedua sesuai dengan sekuensial data (menggunakan None untuk panjang urutan variabel), dan dimensi terakhir sesuai dengan fitur. Perhatikan bahwa kita mengatur return_sequences=True, yang, untuk urutan masukan dengan panjang 3, akan menghasilkan urutan keluaran 〈 $o^{(0)}, o^{(1)}, . . , o^{(T)}\rangle$. Jika tidak, itu hanya akan mengembalikan hasil akhir, $o^{(2)}$.

Sekarang, kita akan memanggil forward pass pada rnn_layer dan secara manual menghitung keluaran pada setiap langkah waktu dan membandingkannya:

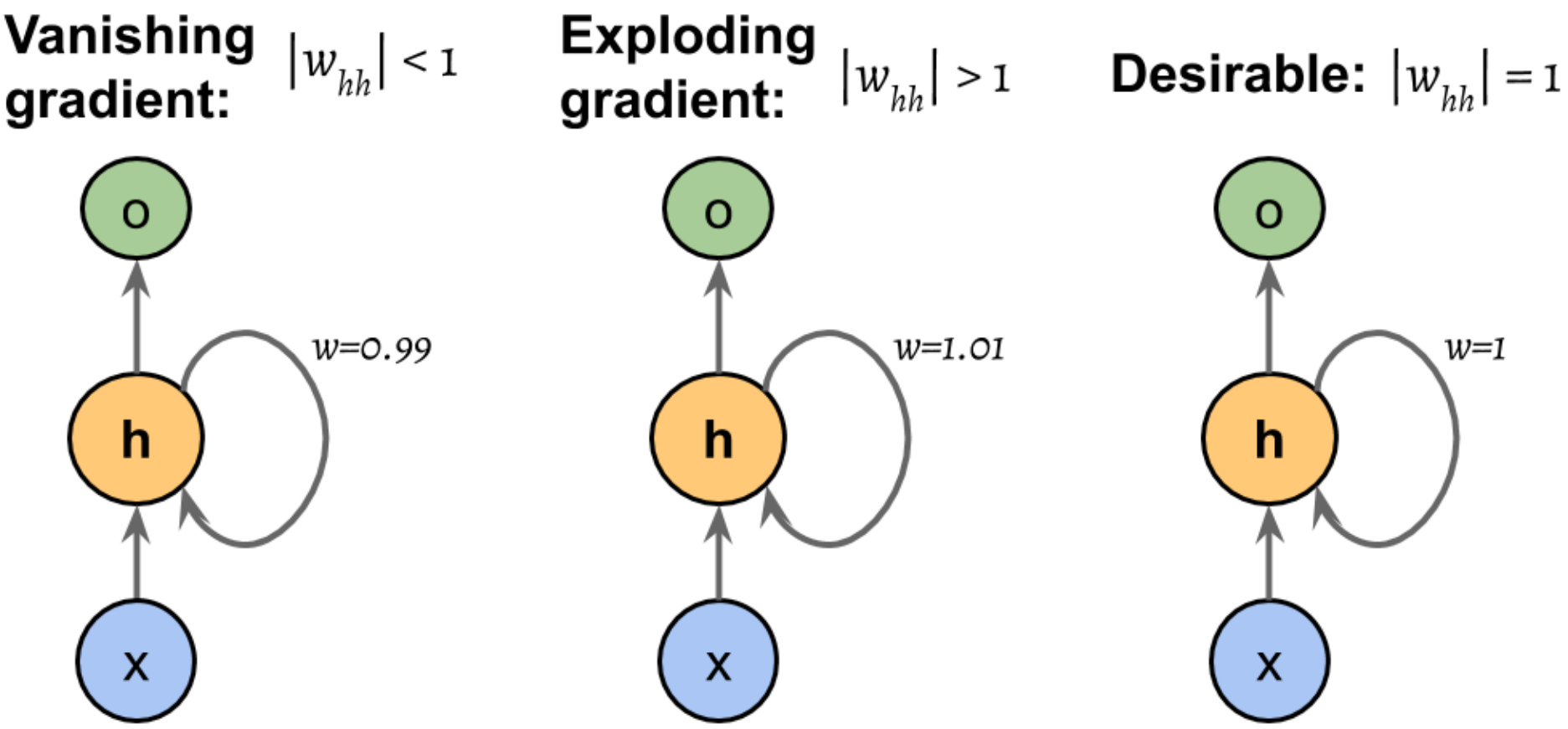
In [ ]:
```
x_seq = tf.convert_to_tensor(
    [[1.0]*5, [2.0]*5, [3.0]*5],
    dtype=tf.float32)


## output of SimepleRNN:
output = rnn_layer(tf.reshape(x_seq, shape=(1, 3, 5)))

## manually computing the output:
out_man = []
for t in range(len(x_seq)):
    xt = tf.reshape(x_seq[t], (1, 5))
    print('Time step {} =>'.format(t))
    print('   Input           :', xt.numpy())

    ht = tf.matmul(xt, w_xh) + b_h
    print('   Hidden          :', ht.numpy())

    if t>0:
        prev_o = out_man[t-1]
    else:
        prev_o = tf.zeros(shape=(ht.shape))

    ot = ht + tf.matmul(prev_o, w_oo)
    ot = tf.math.tanh(ot)
    out_man.append(ot)
    print('   Output (manual) :', ot.numpy())
    print('   SimpleRNN output:'.format(t), output[0][t].numpy())
    print()
```

```
Time step 0 =>
   Input           : [[1. 1. 1. 1. 1.]]
   Hidden          : [[0.03799778 2.1437132 ]]
   Output (manual) : [[0.03797945 0.972892  ]]
   SimpleRNN output: [0.03797945 0.972892  ]

Time step 1 =>
   Input           : [[2. 2. 2. 2. 2.]]
   Hidden          : [[0.07599556 4.2874265 ]]
   Output (manual) : [[0.731829  0.9998504]]
   SimpleRNN output: [0.731829  0.9998504]

Time step 2 =>
   Input           : [[3. 3. 3. 3. 3.]]
   Hidden          : [[0.11399329 6.43114   ]]
   Output (manual) : [[0.5974519  0.99999946]]
   SimpleRNN output: [0.5974519  0.99999946]

   Hidden          : [[0.11399329 6.43114   ]]
   Output (manual) : [[0.5974519  0.99999946]]
   SimpleRNN output: [0.5974519  0.99999946]
```

Dalam perhitungan maju manual kita, kita menggunakan fungsi aktivasi tangen hiperbolik (tanh), karena ini juga digunakan dalam SimpleRNN (aktivasi default). Seperti yang dapat Anda lihat dari hasil yang diprint, output dari forward pass manual sama persis dengan keluaran lapisan SimpleRNN pada setiap langkah waktu. Semoga, hand on ini telah mencerahkan Anda tentang misteri Network yang berulang.

B.4 Tantangan dari learning long-range iterations

BPTT (Backpropagation Through Time) yang telah disebutkan secara singkat sebelumnya, memperkenalkan beberapa tantangan baru. Karena faktor perkalian, $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ dalam menghitung gradien dari fungsi loss, membuat perkalian ini menjadi masalah, dan menyebabkan gradien hilang (Vanishing) dan meledak/tidak terkendali (Exploiding) muncul. Masalah ini dijelaskan dengan contoh pada gambar berikut, yang menunjukkan RNN dengan hanya satu unit tersembunyi untuk penyederhanaan:



Pada dasarnya, $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ memiliki perkalian t – k; oleh karena itu, mengalikan bobot, w, dengan sendirinya t – k kali menghasilkan faktor, w^{t-k} . Sebagai hasilnya, jika $|w| < 1$, faktor ini menjadi sangat kecil ketika t – k besar. Sebaliknya, jika bobot recurrent edge adalah $|w| > 1$, maka w^{t-k} menjadi sangat besar jika t – k besar. Perhatikan bahwa t – k yang besar mengacu pada dependensi long-range. Kita dapat melihat bahwa solusi yang mudah untuk menghindari hilangnya atau meledaknya gradien dapat dicapai dengan memastikan $|w| = 1$. Prakteknya ada 3 solusi untuk mengatasi masalah ini:

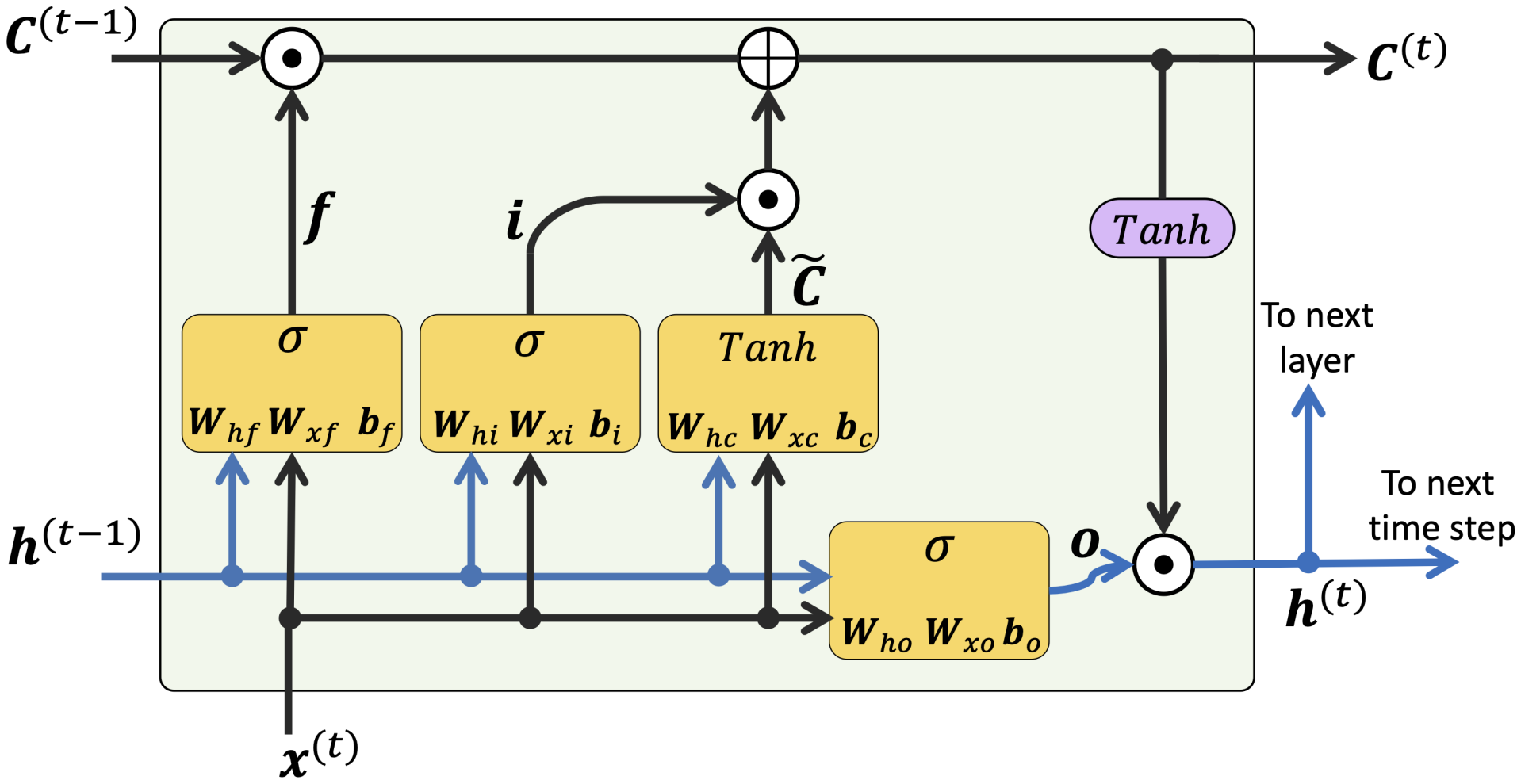
> - Gradient clapping
> - TBPTT
> - LSTM

Menggunakan gradient clappingn, kita menentukan nilai batas atau ambang batas untuk gradien, dan kita menetapkan nilai batas ini ke nilai gradien yang melebihi nilai ini. Sebaliknya, TBPTT hanya membatasi jumlah langkah waktu yang dapat dipropagasi ulang oleh sinyal setelah setiap forward pass. Misalnya, meskipun urutan memiliki 100 elemen atau langkahnya, kita hanya dapat mempropagasi balik 20 langkah waktu terbaru. Sementara gradient clappingn dan TBPTT dapat memecahkan masalah gradien yang tak terkendali, pemotongan membatasi jumlah langkah yang dapat dialirkan kembali secara efektif oleh gradien dan memperbarui bobot dengan benar. Di sisi lain, LSTM, yang dirancang pada tahun 1997 oleh Sepp Hochreiter dan Jürgen Schmidhuber, lebih berhasil dalam mengatasi masalah hilangnya dan tidak terkendalinya gradien sambil memodelkan ketergantungan jarak jauh melalui penggunaan sel memori. Mari kita bahas LSTM lebih detail.

B.5. Long short-term memory cells

Seperti yang dinyatakan sebelumnya, LSTM pertama kali diperkenalkan untuk mengatasi masalah gradien yang hilang (Long Short-Term Memory, S. Hochreiter dan J. Schmidhuber, Neural Computation, 9(8): 1735-1780, 1997). Blok penyusun LSTM adalah sel memori, yang pada dasarnya mewakili atau menggantikan lapisan tersembunyi dari RNN standar.

Di setiap sel memori, ada recurrent edge harus memiliki bobot yang diinginkan, w = 1, seperti yang telah kita diskusikan, untuk mengatasi masalah gradien yang hilang dan yang tidak terkontrol. Nilai-nilai yang terkait dengan recurrent edge ini secara kolektif disebut **cell state**. Struktur sel LSTM modern yang tidak terlipat ditunjukkan pada gambar berikut:



Perhatikan bahwa cell state dari step waktu sebelumnya, $C^{(t-1)}$, dimodifikasi untuk mendapatkan cell state pada langkah waktu saat ini, $C^{(t)}$, tanpa dikalikan langsung dengan faktor bobot apapun. Aliran informasi dalam memory cell ini dikendalikan oleh beberapa unit komputasi (sering disebut gates) yang akan dijelaskan di sini. Pada gambar sebelumnya, $\odot$ mengacu pada **element-wise product** dan $\oplus$ berarti **element-wise summation**. Selanjutnya, $x^{(t)}$ mengacu pada data input pada waktu t, dan $h^{(t-1)}$ menunjukkan hidden unit pada waktu $t-1$. Empat kotak ditunjukkan dengan fungsi aktivasi, baik fungsi sigmoid σ atau tanh , dan satu set bobot; kotak-kotak ini menerapkan kombinasi linier dengan melakukan perkalian matriks-vektor pada inputnya (yaitu $h^{(t-1)}$ dan $x^{(t)}$). Unit perhitungan ini dengan fungsi aktivasi sigmoid, yang unit keluarannya dilewatkan melalui $\odot$, disebut **gates**.

Dalam cell LSTM, ada tiga jenis gete, yang dikenal sebagai forget gate, input gate, dan output get:

> 1. Forget gate (f_t) memungkinkan memory cell untuk mengatur ulang(reset) cell state agar tidak bertambah tanpa batas. Faktanya, Forget gate memutuskan informasi mana yang boleh dilalui dan informasi mana yang harus ditekan. Sekarang, f_t dihitung sebagai berikut:

$$F_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f)$$

Perhatikan bahwa forget gate bukan bagian dari sel LSTM asli; ditambahkan beberapa tahun kemudian untuk menyempurnakan model aslinya (Learning to Forget: Continual Prediction with LSTM, F. Gers, J. Schmidhuber, dan F. Cummins, Neural computation 12, 2451-2471, 2000)

> 2. Input gate (i_t) dan nilai kandidat $(\widetilde{C}_t)$ bertanggung jawab untuk memperbarui cell state. Akan dihitung sebagai berikut:

$$i_t = \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i)$$

$$\widetilde{C}_t = tanh(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c)$$

Cell state pada waktu t dihitung dengan rumus berikut:

$$C^{(t)} = (C^{(t-1)} \odot f_t) \oplus (i_t \odot \widetilde{C}_t)$$

> 3. Output gate (o_t) memutuskan cara memperbarui nilai hidden unit:

$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_0)$$

Dengan persamaan diatas, hidden unit pada step waktu saat ini dihitung sebagai berikut:

$$h^{(t)} = o_t \odot tanh(C_t)$$

Struktur cell LSTM dan perhitungan dasarnya mungkin tampak sangat rumit dan sulit diterapkan. Namun, kabar baiknya adalah TensorFlow telah mengimplementasikan semuanya dalam fungsi pembungkus yang dioptimalkan, yang memungkinkan kita menentukan cell LSTM dengan mudah dan efisien

Note (Model RNN yang lain yang lebih advance)

LSTM menyediakan pendekatan dasar untuk memodelkan dependensi long-rage secara berurutan. Namun, penting untuk dicatat bahwa ada banyak variasi LSTM yang dijelaskan dalam literatur (An Empirical Exploration of Recurrent Network Architectures, Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever, Proceedings of ICML, 2342-2350, 2015). Yang juga perlu diperhatikan adalah pendekatan yang lebih baru, Gated Recurrent Unit (GRU), yang diusulkan pada tahun 2014. GRU memiliki arsitektur yang lebih sederhana daripada LSTM; oleh karena itu, mereka lebih efisien secara komputasi, sementara kinerjanya dalam beberapa tugas, seperti pemodelan musik polifonik, sebanding dengan LSTM. Jika Anda tertarik untuk mempelajari lebih lanjut tentang arsitektur RNN modern ini, lihat makalah, Empiris Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, oleh Junyoung Chung dan lainnya, 2014 (https://arxiv.org/pdf/1412.3555v1.pdf).

C. Implementasi RNNs untuk pemodelan sekuensial dengan TansorFlow

Sekarang setelah kita membahas teori dasar di balik RNN, kita siap untuk beralih ke bagian yang lebih praktis dari bab ini: mengimplementasikan RNN di TensorFlow. Selama sisa bab ini, kita akan menerapkan RNN ke dua kssus masalah umum:

> 1. Sentiment analysis
>
> 2. Language modeling

Kedua proyek ini, yang akan kita bahas bersama di halaman-halaman berikut, sama-sama menarik tetapi juga cukup rumit. Oleh karena itu, daripada memberikan kode sekaligus, kita akan membagi implementasinya menjadi beberapa langkah dan membahas kode tersebut secara mendetail. Jika Anda ingin memiliki gambaran besar dan ingin melihat semua kode sekaligus sebelum terjun ke diskusi, lihat implementasi kode terlebih dahulu, yang dapat Anda lihat di https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch16.

Pada chapter ini, yang pertama akan dibahas analisis sentimen berkaitan dengan menganalisis pendapat yang diungkapkan dari sebuah kalimat atau dokumen teks. Di bagian ini dan subbagian berikutnya, kita akan mengimplementasikan RNN multilayer untuk analisis sentimen menggunakan arsitektur many-to-one.

Untuk yang kedua akan mengimplementasikan RNN Many-to-many untuk aplikasi pemodelan bahasa. Sementara contoh yang dipilih sengaja dibuat sederhana untuk memperkenalkan konsep utama RNN, pemodelan bahasa memiliki berbagai macam aplikasi yang menarik, seperti membuat chatbot—memberi komputer kemampuan untuk langsung berbicara dan berinteraksi dengan manusia.

C.1. Menyiapkan data movie review

Dataset yang telah bersih bernama movie_data.csv, yang akan kita gunakan sekarang. Pertama, kita akan mengimpor modul yang diperlukan dan membaca data ke dalam DataFrame panda, sebagai berikut:

In [ ]:
```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
import pandas as pd
```

```
/Users/istiqomah/anaconda3/envs/mlp/lib/python3.8/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [ ]:
```
import os
import gzip
import shutil


with gzip.open('movie_data.csv.gz', 'rb') as f_in, open('movie_data.csv', 'wb') as f_out:
    shutil.copyfileobj(f_in, f_out)
```

In [ ]:
```
df = pd.read_csv('movie_data.csv', encoding='utf-8')


df.tail()
```

Out[ ]:

	review	sentiment
49995	OK, lets start with the best. the building. al...	0
49996	The British 'heritage film' industry is out of...	0
49997	I don't even know where to begin on this one. ...	0
49998	Richard Tyler is a little boy who is scared of...	0
49999	I waited long to watch this movie. Also becaus...	1

Dapat dilihat bahwa dataframe ini, df, terdiri dari dua kolom, yaitu 'review' dan 'sentiment', dimana 'review' berisi teks review film (fitur input), dan 'sentiment' mewakili label target yang ingin kita prediksi. (0 mengacu pada sentimen negatif dan 1 mengacu pada sentimen positif). Komponen teks ulasan film ini adalah urutan kata, dan model RNN mengklasifikasikan setiap urutan sebagai ulasan positif (1) atau negatif (0). Namun, sebelum kita dapat memasukkan data ke dalam model RNN, kita perlu menerapkan beberapa langkah preprocessing:

> 1. Buat objek dataser TensorFlow dan pisahkan menjadi partisi training, testing, dan validasi set yang terpisah.
> 2. Identifikasi kata-kata unik dalam dataset pelatihan.
> 3. Mapping setiap kata unik ke bilangan integer yang unik dan encode teks ulasan menjadi bilangan integer yang disandikan (indeks dari setiap kata unik).
> 4. Bagilah dataset menjadi beberapa kelompok kecil sebagai masukan untuk model.

Mari lanjutkan dengan langkah pertama: membuat set data TensorFlow dari kerangka data ini:

In [ ]:
```
# Step 1: Create a dataset

target = df.pop('sentiment')

ds_raw = tf.data.Dataset.from_tensor_slices(
    (df.values, target.values))

## inspection:
for ex in ds_raw.take(3):
    tf.print(ex[0].numpy()[0][:50], ex[1])
```

```
b'In 1974, the teenager Martha Moxley (Maggie Grace)' 1
b'OK... so... I really like Kris Kristofferson and h' 0
b'***SPOILER*** Do not read this, if you think about' 0
```

Sekarang, kita dapat membaginya menjadi dataset training, testing, dan validasi set. Seluruh kumpulan data berisi 50.000 example. kita akan menyimpan 25.000 example pertama untuk evaluasi (hold-out testing dataset), lalu 20.000 contoh akan digunakan untuk training dan 5.000 untuk validasi. Kodenya adalah sebagai berikut:

In [ ]:
```
# 1. Buat objek dataser TensorFlow dan pisahkan menjadi partisi training, testing, dan validasi set yang terpisah.
tf.random.set_seed(1)

ds_raw = ds_raw.shuffle(
    50000, reshuffle_each_iteration=False)

ds_raw_test = ds_raw.take(25000)
ds_raw_train_valid = ds_raw.skip(25000)
ds_raw_train = ds_raw_train_valid.take(20000)
ds_raw_valid = ds_raw_train_valid.skip(20000)
```

Untuk menyiapkan data untuk input ke NN, kita perlu mengubah setiap text review menjadi nilai numerik, seperti yang disebutkan di langkah 2 dan 3. Untuk melakukan ini, pertama-tama kita akan menemukan kata unik (token) di dataset training. Sementara menemukan token unik adalah proses di mana kita dapat menggunakan kumpulan data Python, akan lebih efisien jika menggunakan kelas Counter dari paket collections, yang merupakan bagian dari standard library Python.

Dalam kode berikut, kita akan membuat instance objek Counter baru (token_counts) yang akan mengumpulkan frekuensi kata unik. Perhatikan bahwa dalam aplikasi khusus ini (dan berbeda dengan model kantong kata/bag-of-words model), kita hanya tertarik pada kumpulan kata unik dan tidak memerlukan jumlah kata, yang dibuat sebagai side product. Untuk membagi teks menjadi kata (atau token), paket tensorflow_datasets menyediakan Tokenizer class.

In [ ]:
```
## Step 2: Identifikasi kata-kata unik dalam dataset training.

from collections import Counter

try:
    tokenizer = tfds.features.text.Tokenizer()
except AttributeError:
    tokenizer = tfds.deprecated.text.Tokenizer()

token_counts = Counter()

for example in ds_raw_train:
    tokens = tokenizer.tokenize(example[0].numpy()[0])
    token_counts.update(tokens)

print('Vocab-size:', len(token_counts))
```

```
Vocab-size: 87007
```

Jika Anda ingin mempelajari lebih lanjut tentang Penghitung, lihat dokumentasinya di https://docs.python.org/3/library/collections.html#collections.Counter

Selanjutnya, kita akan memetakan setiap kata unik ke bilangan integer unik. Ini dapat dilakukan secara manual menggunakan dictionary Python, di mana kuncinya adalah tokens (kata) unik dan nilai yang terkait dengan setiap kunci adalah bilangan integer unik. Namun, paket tensorflow_datasets sudah menyediakan kelas, TokenTextEncoder, yang bisa kita gunakan untuk membuat mapping semacam itu dan menyandikan seluruh kumpulan data. Pertama, kita akan membuat objek encoder dari kelas TokenTextEncoder dengan meneruskan token unik (token_counts berisi token dan jumlahnya,

meskipun di sini, jumlahnya tidak diperlukan, sehingga akan diabaikan). Memanggil metode encoder.encode() kemudian akan mengonversi teks inputnya menjadi daftar nilai integer:

In [ ]:
```
## Step 3: encoding each unique token into integers

try:
    encoder = tfds.features.text.TokenTextEncoder(token_counts)
except AttributeError:
    encoder = tfds.deprecated.text.TokenTextEncoder(token_counts)

example_str = 'This is an example!'
encoder.encode(example_str)
```

Out[ ]:
```
[232, 9, 270, 1123]
```

Perhatikan bahwa mungkin ada beberapa token dalam data validasi atau pengujian yang tidak ada dalam data training dan karenanya tidak disertakan dalam mapping. Jika kita memiliki q token (yaitu ukuran token_counts diteruskan ke TokenTextEncoder, yang dalam hal ini adalah 87.007), maka semua token yang belum pernah terlihat sebelumnya, dan karenanya tidak termasuk dalam token_counts, akan diberi bilangan integer $q + 1$ (yang akan menjadi 87.008 dalam kasus kita). Dengan kata lain, indeks $q + 1$ dicadangkan untuk kata-kata yang tidak diketahui. Nilai cadangan lainnya adalah bilangan integer 0, yang berfungsi sebagai pengganti/placeholder untuk menyesuaikan panjang urutan. Nanti, saat kita membangun model RNN di TensorFlow, kita akan mempertimbangkan dua placeholder ini, 0 dan $q + 1$, secara lebih mendetail.

Kita dapat menggunakan metode map() dari objek kumpulan data untuk mengubah setiap teks dalam kumpulan data sesuai dengan itu, sama seperti kita akan menerapkan transformasi lainnya ke kumpulan data. Namun, ada masalah kecil: di sini, data teks dilampirkan dalam objek tensor, yang dapat kita akses dengan memanggil metode numpy() pada tensor dalam eager execution mode. Tapi selama transformasi dengan metode map(), eager execution mode akan dinonaktifkan. Untuk mengatasi masalah ini, kita dapat mendefinisikan dua fungsi. Fungsi pertama akan memperlakukan tensor input seolah-olah mode eksekusi diinginkan diaktifkan, dapat dilihat di code 3-A:

Pada fungsi kedua, kita akan membungkus fungsi pertama menggunakan tf.py_function untuk mengubahnya menjadi operator TensorFlow, yang kemudian dapat digunakan melalui metode map(). Proses encoding teks ke dalam daftar bilangan integer ini dapat dilakukan dengan menggunakan code 3-B:

In [ ]:
```
## Step 3-A: define the function for transformation

def encode(text_tensor, label):
    text = text_tensor.numpy()[0]
    encoded_text = encoder.encode(text)
    return encoded_text, label

## Step 3-B: wrap the encode function to a TF Op.
def encode_map_fn(text, label):
    return tf.py_function(encode, inp=[text, label],
                          Tout=(tf.int64, tf.int64))
```

In [ ]:
```
ds_train = ds_raw_train.map(encode_map_fn)
ds_valid = ds_raw_valid.map(encode_map_fn)
ds_test = ds_raw_test.map(encode_map_fn)

tf.random.set_seed(1)
for example in ds_train.shuffle(1000).take(5):
    print('Sequence length:', example[0].shape)

example
```

```
2023-05-29 20:53:54.499578: W tensorflow/tsl/platform/profile_utils/cpu_utils.cc:128] Failed to get CPU frequency: 0 Hz
Sequence length: (24,)
Sequence length: (179,)
Sequence length: (262,)
Sequence length: (535,)
Sequence length: (130,)
```

Out[ ]:
```
(<tf.Tensor: shape=(130,), dtype=int64, numpy=
 array([  579,  1296,    32,   425,    40,   763,  9267,    65,   280,
          308,     6,   481,   155,   473,     2,     3,   684,     9,
          781,   176,   959,   730,  3917,    67,  9905,    13,   277,
           24,    35,   371, 16368,     6,    14, 17231,    29,   187,
         1651,   489,   503,   480,   143,    32,   270,  5851,  2402,
           13,  3592,  3443,   425,  3313,   256,   257,  1577,   117,
            8,   698,   270,   564,    56,     8,    42,  7517,  2629,
          820,    25,    60,    79,   343,    32,   645,    14,   528,
          241,    32,  1980,     8,    56,     8,    42,  1364,   573,
         5183,    43,    12,  3870,    32,   312,   642,   251,  1401,
        17232,     8,   698,   257,   750,     2,     9,    76,   235,
            8,    42,   235,   840,   666,   258, 17233,   419,    32,
        17234,   585,   420,   840,    25,    40,    13,    14,   198,
          266,   623,   173,   179,  4103,   216,    25,   616, 14185,
          186,    35, 16250,   120])>,
 <tf.Tensor: shape=(), dtype=int64, numpy=0>)
```

Sejauh ini, kita telah mengubah urutan kata menjadi urutan bilangan integer. Namun, ada satu masalah yang masih harus kita selesaikan—urutan saat ini memiliki panjang yang berbeda (seperti yang ditunjukkan pada hasil eksekusi kode sebelumnya untuk lima contoh yang dipilih secara acak). Meskipun, secara umum, RNN dapat menangani urutan dengan panjang yang berbeda, kita tetap perlu memastikan bahwa semua urutan dalam mini-batch memiliki panjang yang sama untuk menyimpannya secara efisien di tensor.

Untuk membagi dataset yang memiliki elemen dengan bentuk berbeda ke dalam batch mini, TensorFlow menyediakan metode berbeda, padded_batch() (bukan batch()), yang secara otomatis akan mengisi elemen berurutan yang akan digabungkan ke dalam batch dengan nilai placeholder (0s) sehingga semua urutan dalam satu batch akan memiliki bentuk yang sama. Untuk mengilustrasikan hal ini dengan example praktis, mari ambil subset ukuran 8 dari dataset trainingnya, ds_train, dan terapkan metode padded_batch() ke subset ini dengan batch_size=4. kita juga akan mencetak ukuran masing-masing elemen sebelum menggabungkannya menjadi kumpulan mini, serta dimensi kumpulan mini yang dihasilkan:

In [ ]:
```
## Take a small subset

ds_subset = ds_train.take(8)
for example in ds_subset:
    print('Individual size:', example[0].shape)

## batching the datasets
ds_batched = ds_subset.padded_batch(
    4, padded_shapes=([-1], []))

for batch in ds_batched:
    print('Batch dimension:', batch[0].shape)
```

```
Individual size: (119,)
Individual size: (688,)
Individual size: (308,)
Individual size: (204,)
Individual size: (326,)
Individual size: (240,)
Individual size: (127,)
Individual size: (453,)
Batch dimension: (4, 688)
Batch dimension: (4, 453)
```

Seperti yang dapat kita amati dari bentuk tensor yang diprint, jumlah kolom (yaitu, .shape[1]) pada batch pertama adalah 688, yang dihasilkan dari penggabungan empat contoh pertama menjadi satu batch dan menggunakan ukuran maksimumnya example. Itu berarti bahwa tiga example lainnya dalam batch ini diisi sebanyak yang diperlukan agar sesuai dengan ukuran ini. Demikian pula, batch kedua mempertahankan ukuran maksimum dari empat contoh individualnya, yaitu 453, dan melapisi contoh lainnya sehingga panjangnya lebih kecil dari panjang maksimum.

In [ ]:
```
## batching the datasets
train_data = ds_train.padded_batch(
    32, padded_shapes=([-1],[]))

valid_data = ds_valid.padded_batch(
    32, padded_shapes=([-1],[]))

test_data = ds_test.padded_batch(
    32, padded_shapes=([-1],[]))
```

Sekarang, data berada dalam format yang sesuai untuk model RNN, yang akan kita terapkan di subbagian berikut. Namun, pada sub-bagian berikutnya, pertama-tama kita akan membahas penyematan fitur, yang merupakan langkah prapemrosesan opsional namun sangat disarankan yang digunakan untuk mengurangi dimensi vektor kata.

C.2. Embedding layers untuk sentence encoding

Selama persiapan data pada langkah sebelumnya, kita menghasilkan urutan dengan panjang yang sama. Elemen dari urutan ini adalah bilangan integer yang sesuai dengan indeks kata-kata unik. Indeks kata ini dapat diubah menjadi fitur input dengan beberapa cara berbeda. Salah satu cara yang memungkinkna adalah menerapkan one-hot encoding untuk mengubah indeks menjadi vektor nol dan satu. Kemudian, setiap kata akan dipetakan ke vektor yang ukurannya adalah jumlah kata unik di seluruh dataset. Mengingat bahwa jumlah kata unik (ukuran kosakata) dapat berada di urutan $10^4 - 10^5$, yang juga akan menjadi jumlah fitur input kita, model yang dilatih pada fitur tersebut mungkin mengalami curse of dimensionality (kutukan dimensi). Selain itu, fitur ini sangat jarang, karena semuanya nol kecuali satu.

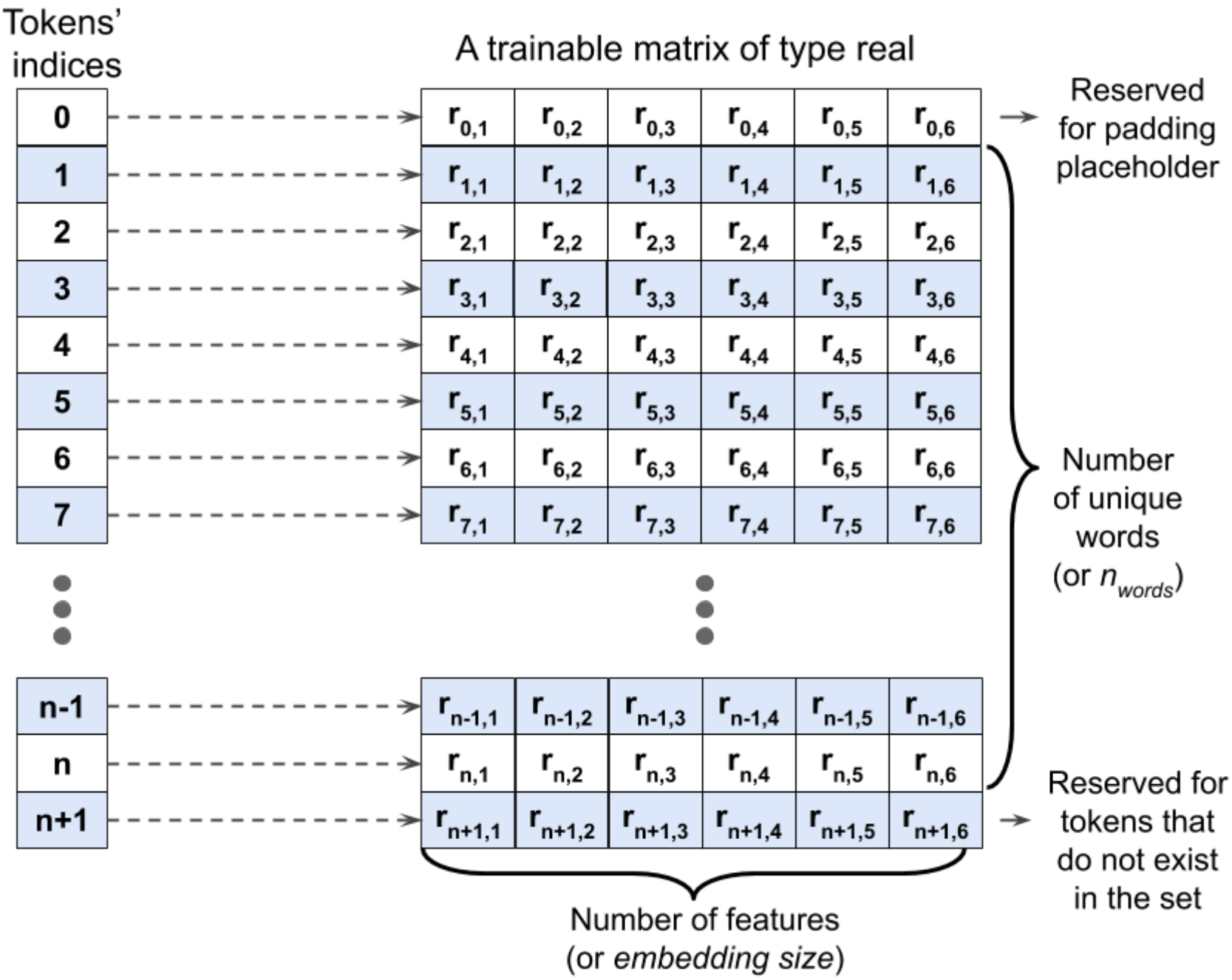
Pendekatan yang lebih elegan adalah memetakan setiap kata ke vektor dengan ukuran tetap dengan elemen bernilai nyata (tidak harus bilangan bulat). Berbeda dengan vektor yang one-hot encoding, kita dapat menggunakan vektor berukuran hingga untuk mewakili jumlah bilangan real yang tak terbatas. (Secara teori, kita dapat mengekstraksi bilangan real tak terhingga dari interval tertentu, misalnya [–1, 1].)

Ini adalah ide di balik embedding, yang merupakan teknik pembelajaran fitur yang dapat kita manfaatkan di sini untuk mempelajari fitur yang menonjol secara otomatis untuk merepresentasikan kata-kata dalam kumpulan data kita. Mengingat jumlah kata unik, n_{words} , kita dapat memilih ukuran vektor embedding (alias embedding dimension) jauh lebih kecil daripada jumlah kata unik ($embedding\_dim << n_{words}$)untuk mewakili seluruh kosakata sebagai fitur masukan.

Keuntungan embedding dibandingkan one-hot encoding adalah sebagai berikut:

> - Pengurangan dimensi ruang fitur untuk mengurangi efek curse of dimensionality
> - Ekstraksi fitur yang menonjol karena lapisan embedding dalam NN dapat dioptimalkan (atau dipelajari)

Representasi skematis berikut menunjukkan cara kerja penyematan dengan memetakan indeks token ke matriks embedding yang dapat dilatih:



Dijelaskan sebelumnya ada satu set token berukuran n + 2 (n adalah ukuran set token, ditambah indeks 0 dicadangkan untuk padding placeholder, dan n + 1 untuk kata-kata yang tidak ada dalam set token), sebuah matriks embedding dari ukuran $(n + 2) \times embedding\_dim$ akan dibuat di mana setiap baris dari matriks ini mewakili fitur numerik yang terkait dengan token. Oleh karena itu, ketika sebuah indeks bilangan integer, i, diberikan sebagai input ke embedding, itu akan mencari baris matriks yang sesuai pada indeks i dan mengembalikan fitur numerik. Matriks embedding berfungsi sebagai lapisan masukan untuk model NN kita. Dalam praktiknya, membuat layer embedding dapat dilakukan dengan mudah menggunakan tf.keras.layer.Embedding. Mari kita lihat contoh dimana kita akan membuat model dan menambahkan layer embedding, sebagai berikut:

In [ ]:
```
from tensorflow.keras.layers import Embedding


model = tf.keras.Sequential()

model.add(Embedding(input_dim=100,
                    output_dim=6,
                    input_length=20,
                    name='embed-layer'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embed-layer (Embedding)     (None, 20, 6)             600

=================================================================
Total params: 600
Trainable params: 600
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embed-layer (Embedding)     (None, 20, 6)             600

=================================================================
Total params: 600
Trainable params: 600
Non-trainable params: 0
_________________________________________________________________
```

input ke model ini (layer embedding) harus memiliki rank 2 dengan dimensi $batchsize \times input_lenght$, di mana input_lenght adalah panjang urutan (di sini, atur ke 20 melalui argumen input_length). Misalnya, sekuensial input dalam mini-match dapat berupa ⟨14, 43, 52, 61, 8,19,67,83,10,7,42,87,56,18,94,17,67,90, 6,39⟩ , di mana setiap elemen dari urutan ini adalah indeks dari kata unik. Outputnya akan memiliki dimensi $batchsize \times input\_lenght \times embedding\_size$, di mana embedding_size adalah ukuran fitur embedding (di sini, disetel ke 6 melalui output_dim).

Argumen lain yang diberikan ke layer embedding, input_dim, sesuai dengan nilai integer unik yang akan diterima model sebagai input (misalnya, n + 2, disetel di sini ke 100). Oleh karena itu, matriks penyisipan dalam hal ini berukuran 100 × 6.

Note (Menyesuaikan variable seuence lengths)

Perhatikan bahwa argumen input_length tidak diperlukan, dan kita dapat menggunakan None untuk kasus di mana panjang urutan input bervariasi. Anda dapat menemukan informasi lebih lanjut tentang fungsi ini dalam dokumentasi resmi di https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding.

C.3. Membangung sebuah RNN model

Sekarang kita siap membuat model RNN. Menggunakan Keras Sequential class,, kita dapat menggabungkan layer embedding, layer recurrent dari RNN, dan layer non-recurrent yang terhubung sepenuhnya. Untuk lapisan berulang, kita dapat menggunakan salah satu implementasi berikut: recurrent

> 1. SimpleRNN: lapisan RNN biasa, yaitu fully connected recurrent layer.
> 2. LSTM: sebuah long short-term memory RNN, yang berguna untuk menangkap long-term dependensi input.
> 3. GRU: lapisan berulang dengan unit gated recurrent, seperti yang diusulkan dalam Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation (https://arxiv.org/abs/1406.1078v3), sebagai alternatif dari LSTM.

Untuk melihat bagaimana model RNN multilayer dapat dibangun menggunakan salah satu dari layer berulang ini, dalam contoh berikut, kita akan membuat model RNN, dimulai dengan layer embedding dengan input_dim=1000 dan output_dim=32. Kemudian, dua lapisan tipe SimpleRNN berulang akan ditambahkan. Terakhir, kita akan menambahkan lapisan non-recurrent fully connected sebagai lapisan output, yang akan mengembalikan nilai keluaran tunggal sebagai prediksi:

In [ ]:
```
## An example of building a RNN model
## with SimpleRNN layer

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import SimpleRNN
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Embedding(1000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32))
model.add(Dense(1))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding (Embedding)        (None, None, 32)          32000

 simple_rnn_1 (SimpleRNN)     (None, None, 32)          2080

 simple_rnn_2 (SimpleRNN)     (None, 32)                2080

 dense (Dense)                (None, 1)                 33

=================================================================
Total params: 36,193
Trainable params: 36,193
Non-trainable params: 0
_________________________________________________________________
```


```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding (Embedding)        (None, None, 32)          32000

 simple_rnn_1 (SimpleRNN)     (None, None, 32)          2080

 simple_rnn_2 (SimpleRNN)     (None, 32)                2080

 dense (Dense)                (None, 1)                 33

=================================================================
Total params: 36,193
Trainable params: 36,193
Non-trainable params: 0
_________________________________________________________________
```


Seperti yang kita lihat, membangun model RNN menggunakan lapisan berulang ini cukup mudah. Pada sub-bagian berikutnya, kita akan kembali ke tugas analisis sentimen dan membuat model RNN untuk menyelesaikannya.

C.4. Membangung sebuah RNN model untuk tugas sentiment analysis

Karena kita memiliki urutan yang sangat panjang, kita akan menggunakan lapisan LSTM untuk memperhitungkan efek long-term. Selain itu, kita akan menempatkan lapisan LSTM di dalam pembungkus Bidirectional, yang akan membuat lapisan berulang melewati urutan input dari kedua arah, awal sampai akhir, serta arah sebaliknya:

In [ ]:
```
## An example of building a RNN model
## with LSTM layer
```

```
from tensorflow.keras.layers import LSTM


model = Sequential()
model.add(Embedding(10000, 32))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(1))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 32)          320000

 lstm (LSTM)                 (None, None, 32)          8320

 lstm_1 (LSTM)               (None, 32)                8320

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 336,673
Trainable params: 336,673
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 32)          320000

 lstm (LSTM)                 (None, None, 32)          8320

 lstm_1 (LSTM)               (None, 32)                8320

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 336,673
Trainable params: 336,673
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
embedding_dim = 20
vocab_size = len(token_counts) + 2

tf.random.set_seed(1)

## build the model
bi_lstm_model = tf.keras.Sequential([
    tf.keras.layers.Embedding(
        input_dim=vocab_size,
        output_dim=embedding_dim,
        name='embed-layer'),

    tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(64, name='lstm-layer'),
        name='bidir-lstm'),

    tf.keras.layers.Dense(64, activation='relu'),

    tf.keras.layers.Dense(1, activation='sigmoid')
])

bi_lstm_model.summary()

## compile and train:
bi_lstm_model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
    metrics=['accuracy'])

history = bi_lstm_model.fit(
    train_data,
    validation_data=valid_data,
    epochs=2)

## evaluate on the test data
test_results= bi_lstm_model.evaluate(test_data)
print('Test Acc.: {:.2f}%'.format(test_results[1]*100))
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embed-layer (Embedding)     (None, None, 20)          1740180

 bidir-lstm (Bidirectional)  (None, 128)               43520

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 1)                 65

=================================================================
Total params: 1,792,021
Trainable params: 1,792,021
Non-trainable params: 0
_________________________________________________________________
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use the legacy Kera
s optimizer instead, located at `tf.keras.optimizers.legacy.Adam`.
WARNING:absl:There is a known slowdown when using v2.11+ Keras optimizers on M1/M2 Macs. Falling back to the legacy Keras optimi
zer, i.e., `tf.keras.optimizers.legacy.Adam`.
Epoch 1/2
625/625 [==============================] - 225s 358ms/step - loss: 0.5235 - accuracy: 0.7282 - val_loss: 0.3693 - val_accuracy:
0.8544
Epoch 2/2
625/625 [==============================] - 313s 500ms/step - loss: 0.3154 - accuracy: 0.8787 - val_loss: 0.3768 - val_accuracy:
0.8496
782/782 [==============================] - 201s 257ms/step - loss: 0.3716 - accuracy: 0.8546
Test Acc.: 85.46%
```

Setelah melatih model ini selama 10 epoch, evaluasi pada data pengujian menunjukkan akurasi 82 persen (Untuk akurasi di buku referensi 85 persen). (Perhatikan bahwa hasil ini bukan yang terbaik jika dibandingkan dengan metode canggih yang digunakan pada kumpulan data IMDb. Tujuannya hanya untuk menunjukkan cara kerja RNN.)

Note (Lebih lanjut tentang the bidirectional RNN)

Bidirectional wrapper class membuat dua lintasan pada setiap urutan input: lintasan maju dan lintasan mundur atau mundur (perhatikan bahwa ini jangan dikacaukan dengan lintasan maju dan mundur dalam konteks backpropagation). Hasil dari operan maju dan mundur ini akan digabungkan secara default. Tetapi jika Anda ingin mengubah perilaku ini, Anda dapat mensetting argumen merge_mode ke 'sum' (untuk penjumlahan), 'mul' (untuk mengalikan hasil dari dua lintasan), 'ave' (untuk mengambil rata-rata keduanya) , 'concat' (yang merupakan default), atau None, yang mengembalikan dua tensor dalam daftar. Untuk informasi lebih lanjut tentang Bidirectional wrapper class, silakan lihat dokumentasi resmi di https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Bidirectional .

In [ ]:
```python
def preprocess_datasets(
    ds_raw_train,
    ds_raw_valid,
    ds_raw_test,
    max_seq_length=None,
    batch_size=32):

    ## Step 1: (already done => creating a dataset)
    ## Step 2: find unique tokens
    try:
        tokenizer = tfds.features.text.Tokenizer()
    except AttributeError:
        tokenizer = tfds.deprecated.text.Tokenizer()
    token_counts = Counter()

    for example in ds_raw_train:
        tokens = tokenizer.tokenize(example[0].numpy()[0])
        if max_seq_length is not None:
            tokens = tokens[-max_seq_length:]
        token_counts.update(tokens)

    print('Vocab-size:', len(token_counts))


    ## Step 3: encoding the texts
    try:
        encoder = tfds.features.text.TokenTextEncoder(token_counts)
    except AttributeError:
        encoder = tfds.deprecated.text.TokenTextEncoder(token_counts)
    def encode(text_tensor, label):
        text = text_tensor.numpy()[0]
        encoded_text = encoder.encode(text)
        if max_seq_length is not None:
            encoded_text = encoded_text[-max_seq_length:]
        return encoded_text, label

    def encode_map_fn(text, label):
        return tf.py_function(encode, inp=[text, label],
                              Tout=(tf.int64, tf.int64))

    ds_train = ds_raw_train.map(encode_map_fn)
    ds_valid = ds_raw_valid.map(encode_map_fn)
    ds_test = ds_raw_test.map(encode_map_fn)
```

```
    ## Step 4: batching the datasets
    train_data = ds_train.padded_batch(
        batch_size, padded_shapes=([-1],[]))

    valid_data = ds_valid.padded_batch(
        batch_size, padded_shapes=([-1],[]))

    test_data = ds_test.padded_batch(
        batch_size, padded_shapes=([-1],[]))

    return (train_data, valid_data,
            test_data, len(token_counts))
```

In [ ]:
```
def build_rnn_model(embedding_dim, vocab_size,
                    recurrent_type='SimpleRNN',
                    n_recurrent_units=64,
                    n_recurrent_layers=1,
                    bidirectional=True):

    tf.random.set_seed(1)

    # build the model
    model = tf.keras.Sequential()

    model.add(
        Embedding(
            input_dim=vocab_size,
            output_dim=embedding_dim,
            name='embed-layer')
    )

    for i in range(n_recurrent_layers):
        return_sequences = (i < n_recurrent_layers-1)

        if recurrent_type == 'SimpleRNN':
            recurrent_layer = SimpleRNN(
                units=n_recurrent_units,
                return_sequences=return_sequences,
                name='simprnn-layer-{}'.format(i))
        elif recurrent_type == 'LSTM':
            recurrent_layer = LSTM(
                units=n_recurrent_units,
                return_sequences=return_sequences,
                name='lstm-layer-{}'.format(i))
        elif recurrent_type == 'GRU':
            recurrent_layer = GRU(
                units=n_recurrent_units,
                return_sequences=return_sequences,
                name='gru-layer-{}'.format(i))

        if bidirectional:
            recurrent_layer = Bidirectional(
                recurrent_layer, name='bidir-'+recurrent_layer.name)

        model.add(recurrent_layer)

    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

    return model
```

In [ ]:
```
from tensorflow.keras.layers import Bidirectional


batch_size = 32
embedding_dim = 20
max_seq_length = 100

train_data, valid_data, test_data, n = preprocess_datasets(
    ds_raw_train, ds_raw_valid, ds_raw_test,
    max_seq_length=max_seq_length,
    batch_size=batch_size
)


vocab_size = n + 2

rnn_model = build_rnn_model(
    embedding_dim, vocab_size,
    recurrent_type='SimpleRNN',
    n_recurrent_units=64,
    n_recurrent_layers=1,
    bidirectional=True)

rnn_model.summary()
```

```
Vocab-size: 58063
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embed-layer (Embedding)     (None, None, 20)          1161300

 bidir-simprnn-layer-0 (Bidi  (None, 128)              10880
 rectional)

 dense_4 (Dense)             (None, 64)                8256

 dense_5 (Dense)             (None, 1)                 65

=================================================================
Total params: 1,180,501
Trainable params: 1,180,501
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
rnn_model.compile(optimizer=tf.keras.optimizers.Adam(1e-3),
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
                  metrics=['accuracy'])


history = rnn_model.fit(
    train_data,
    validation_data=valid_data,
    epochs=2)
```

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.Adam`.
WARNING:absl:There is a known slowdown when using v2.11+ Keras optimizers on M1/M2 Macs. Falling back to the legacy Keras optimizer, i.e., `tf.keras.optimizers.legacy.Adam`.
Epoch 1/2
480/625 [======================>.......] - ETA: 44:50 - loss: 0.6983 - accuracy: 0.4961
```

In [ ]:
```
results = rnn_model.evaluate(test_data)
```

In [ ]:
```
print('Test Acc.: {:.2f}%'.format(results[1]*100))
```

In [ ]:
```
lstm_model = build_rnn_model(
    embedding_dim, vocab_size,
    recurrent_type='LSTM',
    n_recurrent_units=64,
    n_recurrent_layers=1,
    bidirectional=True)

lstm_model.summary()
```

In [ ]:
```
lstm_model.compile(optimizer=tf.keras.optimizers.Adam(1e-3),
                   loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
                   metrics=['accuracy'])


history = lstm_model.fit(
    train_data,
    validation_data=valid_data,
    epochs=10)
```

In [ ]:
```
results_lstm = lstm_model.evaluate(test_data)
```

In [ ]:
```
print('Test Acc.: {:.2f}%'.format(results_lstm[1]*100))
```

In [ ]: