Konsep Recurrent Neural Network dan Aplikasinya

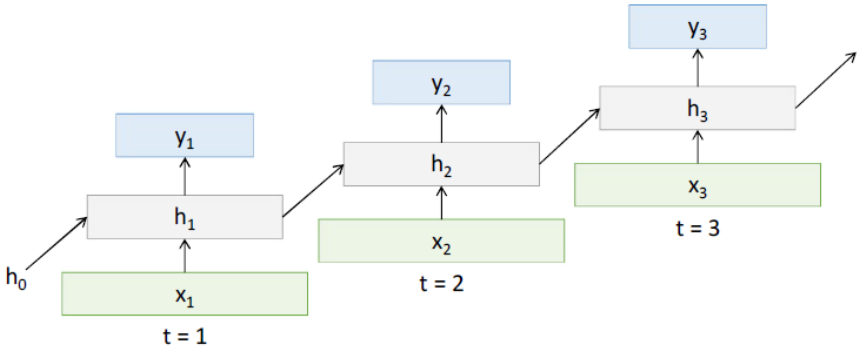
Muhammad Sidik Asyaky[1]

Institut Teknologi Bandung
23519021@std.stei.itb.ac.id

Abstract. Model pembelajaran *neural network* memiliki kesulitan dalam menghasilkan model yang bagus dari data yang sekuensial. Namun dengan Recurrent Neural Network (RNN), masalah tersebut hilang dengan menggunakan konsep hidden state yang juga bertindak sebaagai "memory" dalam setiap time steps sehingga model pembelajaran RNN memiliki performa yang baik terhadap kasus dengan data sekuensial. Meskipun begitu, terdapat masalah yang menyebabkan RNN memiliki performa yang buruk terhadap dataset yang besar, yaitu *vanishing gradient*. Tetapi dijelaskan juga bagaimana Long Short-Term Memory dapat menyempurnakan RNN dan menghindari masalah *vanishing gradient* dengan mekanisme gates. Terdapat penjelasan mengenai 4 projek yang mengaplikasikan konsep RNN sehingga pembaca dapat tahu bagaimana implementasi konsep RNN ke bentuk kode dan juga bagaimana langkah-langkah yang dilakukan dalam tahap pembuatan model pembelajaran yang menggunakan RNN dan LSTM pada RNN. Dapat dibuktikan dengan projek text generator dan stock prediction, bahwa RNN dapat bekerja dengan baik terhadapat data yang sekuensial

1 Introduction to Recurrent Neural Network

Recurrent Neural Network (RNN) adalah tipe *Neural Network* yang menggunakan *output* atau *hidden state* di step sebelumnya menjadi input sehingga RNN bagus digunakan untuk memproses data sequential. Dengan kata lain, metode RNN digunakan sebagai solusi ketika menghadapi permasalahan dimana kita membutuhkan algoritma pembelajaran yang memperhitungkan informasi yang didapat dari input sebelumya ke input setelahnya. Contohnya dalam kasus *text recognition*, *stock prediction*, *image captioning*, *language translation*, dan sebagainya.



Gambar 1. How RNN works

Diagram diatas menunjukkan bagaimana RNN bekerja. Berikut penjelasan dari setiap simbol didalamnya :

- t = time step atau tahapan.
- x = input pada setiap time step (t).
- h = hidden state pada setiap time step (t). Hidden state bisa kita sebut sebagai memory pada sebuah network yang berfungsi menyimpan hasil kalkulasi dan rekaman yang telah dilakukan. h memperhitungkan hidden state (h) pada step sebelumnya.
- y = output pada setiap time step (t).

RNN memiliki hidden state yang bisa menyimpan informasi input sebelumnya untuk waktu yang tidak tetap secara apriori. Dengan itu, informasi input sebelumnya dapat diproses di step setelahnya. Tetapi RNN memiliki masalah dalam mempertahan informasi dari step sebelumnya. Dalam hidden state, setiap informasi yang didapat dari step sebelumnya akan memiliki distribusi yang berbeda, tergantung kepada seberapa jauh step sebelumnya ke step yang sekarang. Disebut juga *short-term memory*. Hal tersebut disebabkan oleh *Gradient Vanishing* -- masalah yang umum dalam arsitektur neural network lainnya. Gradient Vanishing terjadi karena ada proses backpropagation, yaitu algoritma yang digunakan untuk melatih dan mengoptimisasi neural network. Berikut penjelasannya :

Proses melatih recurrent neural network terdiri dari 3 langkah utama, yaitu :

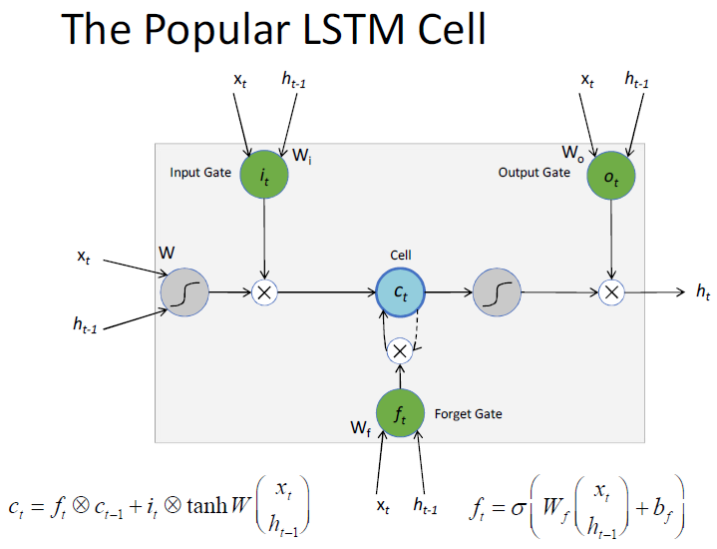
1. Melakukan forward pass dan membuat prediksi
2. Membandingkan hasil prediksi dengan ground truth menggunakan *Loss Function* (C). Hasil perhitungan fungsi tersebut adalah nilai error yang digunakan untuk menghitung performa network.
3. Nilai error digunakan untuk melakukan *backpropagation through time* yang menghitung gradient untuk setiap node dalam network. Gradient tersebut adalah nilai yang digunakan untuk menyesuaikan weight dalam network, sehingga network dapat ‘belajar’. Semakin besar gradient, maka

semakin besar penyesuaian yang dilakukan.

Ketika melakukan backpropagation, setiap node dalam time step menghitung nilai gradient-nya berdasarkan efek gradient dari time step sebelumnya. Sehingga nilai penyesuaian untuk time step akan semakin kecil pada time step berikutnya. Dengan kata lain, nilai gradient akan menyusut secara exponensial ketika proses *back propagates down*, menyebabkan pengaruh time steps sebelumnya terhadap weight semakin kecil dan semakin non-existent atau disebut juga *Vanishing Gradient*.

Solusi dari masalah *short-term memory* yang dimiliki RNN 'vanilla', salah satunya adalah Long Short-Term Memory (LSTM), yaitu metode RNN advance yang menggunakan Constant Error Flow untuk membuat Constant Error Carousel yang memastikan nilai gradient tidak menyusut. LSTM memiliki mekanisme yang disebut gates untuk meregulasi *value flows*. Gates tersebut dapat 'belajar' untuk memilih data dalam sequens yang harus disimpan atau dibuang. Dengan itu, hanya informasi yang relevan yang akan digunakan untuk membuat prediksi.

Secara garis besar, komponen kunci dalam LSTM adalah memory cell yang menjadi *accumulator* (menyimpan hubungan identitas) *over time*. Dengan kata lain, memory cell menyimpan informasi relevan selama pemprosesan sequensial sehingga informasi dari time step sebelumnya dapat dipertahankan sampai time step setelahnya selama informasi tersebut relevan. Oleh karena itu, masalah short-term memory berkurang karena menghindari vanishing gradient. Sepanjang pemrosesan sequensial, informasi dalam memory cell bertambah dan berkurang melalui gates. Gates adalah mekanisme yang menilai apakah informasi relevan atau tidak untuk disimpan di memory cell.



Gambar 2. Popular LSTM Cell

Terdapat tiga gates yang mengatur informasi dalam LSTM, yaitu :

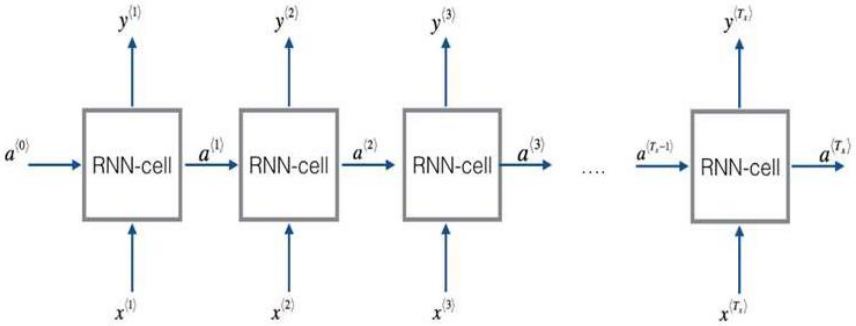
1. Input gate, mengontrol nilai baru mana yang disimpan ke cell. Dilakukan dengan menghitung fungsi sigmoid dari hidden state sebelumnya dan input saat ini. Fungsi sigmoid tersebut memutuskan nilai mana yang akan relevan dengan mengubah nilainya ke bentuk 0 atau 1. 0 berarti tidak penting, dan 1 berarti informasi tersebut relevan. Selain itu, melakukan perhitungan tanh function dari hidden state dan input saat ini untuk meregulasi network. Lalu mengalikan output tanh dengan output sigmoid. Ouput sigmoid akan memutuskan informasi mana yang relevan dari output tanh.
2. Forget gate, mengontrol apakah nilai dipertahankan atau dikerluarkan dari cell.
3. Output gate, mengontrol nilai dalam cell yang mana yang akan digunakan untuk menghitung output activation dari LSTM unit.

2 RNN Implementation

2.1 Recurrent Neural Network Hands On

Dalam projek ini, saya membuat program pengimplementasian RNN dari konsep di bab Introduction ke dalam bentuk kode. Program yang dibuat akan mengimplementasikan RNN dengan forward propagation, LSTM network dan backward propagation.

2.1.1 Forward Propagation

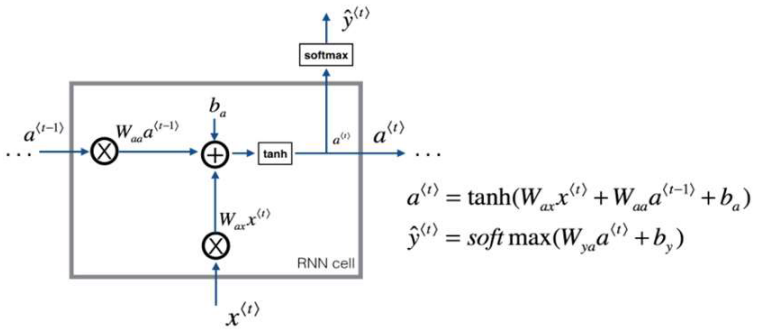


Gambar 3. Struktur Forward Propagation

Tahapan :

1. Mengimplementasikan kalkulasi yang dibutuhkan untuk satu time step dari RNN.
2. Membuat loop sebanyak Tx time-steps untuk memproses semua input satu per satu.

Komputasi untuk satu time step dapat digambarkan sebagai berikut :



Gambar 4. Struktur dan rumus single forward step pada RNN

RNN Cell Forward : Mengimplementasikan single forward step pada RNN cell.

Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan gambaran komputasi diatas :

- xt = Input data pada time step (t).
- a_prev = hidden state pada time step sebelumnya (t-1).
- Wax = Matriks weight yang mengalikan input.
- Waa = Matriks weight yang mengalikan hidden state.
- Wya = Matriks weight antara hidden state dan output.
- ba = Bias.
- by = Bias antara hidden state dan output.
- a_next = hidden state selanjutnya (t+1).
- yt_pred = prediksi pada time step (t)
- cache = nilai tuple yang digunakan untuk backward pass.

```
def rnn_cell_forward(xt, a_prev, parameters) :
    # Memanggil parameter
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
```

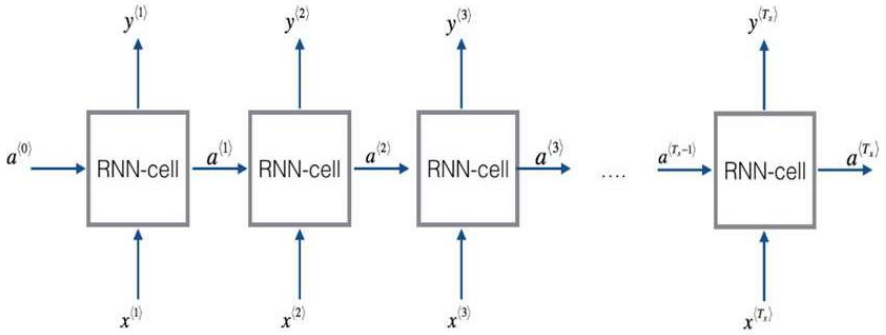
```
    by = parameters["by"]

    # Menghitung activation state berikutnya
    a_next = np.tanh(np.dot(Wax,xt)+np.dot(Waa,a_prev)+ba)
    #compute output of the current cell using the formula given above
    yt_pred = softmax(np.dot(Wya,a_next)+by)

    # Menyimpan nilai untuk backward propagation dalam cache
    cache = (a_next, a_prev, xt, parameters)

    return a_next, yt_pred, cache
```

Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung a_next, yt_pred dengan rumus yang digunakan dalam RNN Cell Forward.

```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa" : Waa, "Wax" : Wax, "Wya" : Wya, "ba" : ba, "by" : by}

a_next, yt_pred, cache = rnn_cell_forward(xt, a_prev, parameters)
print("a_next[4] = ", a_next[4])
print("a_next.shape = ", a_next.shape)
print("yt_pred[1] = ", yt_pred[1])
print("yt_pred.shape = ", yt_pred.shape)
```

```
a_next[4] =  [ 0.59584544  0.18141802  0.61311866  0.99808218  0.85016201  0.99980978
 -0.18887155  0.99815551  0.6531151   0.82872037]
a_next.shape =  (5, 10)
yt_pred[1] =  [0.9888161  0.01682021 0.21140899 0.36817467 0.98988387 0.88945212
 0.36920224 0.9966312  0.9982559  0.17746526]
yt_pred.shape =  (2, 10)
```

RNN Forward Pass : Mengimplementasikan forward propagation pada RNN



Gambar 6. Struktur RNN Forward Pass

Setiap cell mengambil hidden state dari cell sebelumnya (t-1) dan input time step saat ini (t), dan menjadikannya hidden state dan prediksi untuk time step saat ini.

Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan gambaran komputasi diatas :

- x = Input data untuk setiap time step (t).
- A0 = hidden state pertama.
- Wax = Matriks weight yang mengalikan input.
- Waa = Matriks weight yang mengalikan hidden state.
- Wya = Matriks weight antara hidden state dan output.
- ba = Bias.
- by = Bias antara hidden state dan output.
- a = hidden state untuk setiap time step (t+1).
- y_pred = prediksi untuk setiap time step (t)
- caches = nilai tuple yang digunakan untuk backward pass.

```
def rnn_forward(x, a0, parameters) :
    # Menginisialisasi “caches” yang akan menyimpan daftar semua cache
    caches = []

    # Mengambil ukuran bentuk x dan parameters["Wya"]
    n_x, m, T_x = x.shape
    n_y, n_a = parameters["Wya"].shape

    # Meninisialisasi a dan y dengan “zeros”
    a = np.zeros((n_a, m, T_x))
    y_pred = np.zeros((n_y, m, T_x))

    # Menginisialiasi a_next
```

```
    a_next = a0

    # Melakukan loop untuk setiap time steps
    for t in range(0, T_x):

        # Memperbarui hidden state selanjutnya dan menghitung prediksi
        a_next, yt_pred, cache = rnn_cell_forward(x[:,:,t], a_next, parameters)

        # Menyimpan nilai hidden selanjutnya
        a[:,:,t] = a_next

        # Menyimpan nilai prediksi
        y_pred[:,:,t] = yt_pred

        # Menambahkan cache ke daftar cache
        caches.append(cache)

    # Menyimpan nilai yang dibutuhkan untuk backward propagation dalam cache
    caches = (caches, x)

    return a, y_pred, caches
```

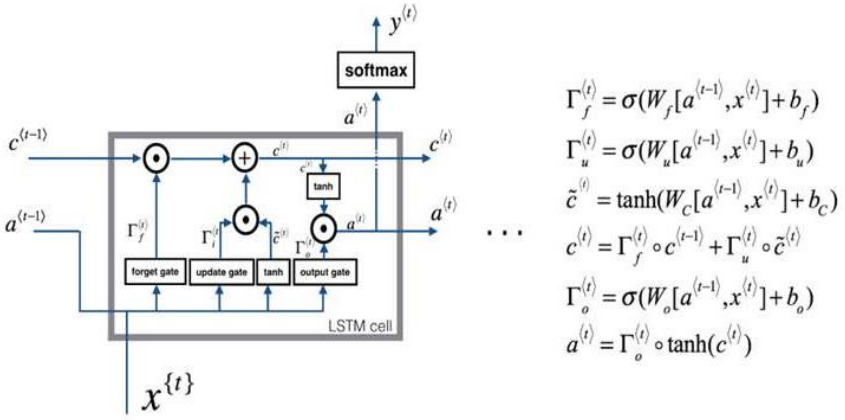
Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung a, y_pred dengan rumus yang digunakan dalam RNN Forward Pass.

```
np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa":Waa, "Wax":Wax, "Wya":Wya, "ba":ba, "by":by}

a, y_pred, caches = rnn_forward(x, a0, parameters)
print("a[4][1] = ", a[4][1])
print("a.shape = ", a.shape)
print("y_pred[1][3] = ", y_pred[1][3])
print("y_pred.shape = ", y_pred.shape)
print("caches[1][1][3] = ", caches[1][1][3])
print("len(caches) = ", len(caches))
```

```
a[4][1] =  [-0.99999375  0.77911235 -0.99861469 -0.99833267]
a.shape =  (5, 10, 4)
y_pred[1][3] =  [0.79560373 0.86224861 0.11118257 0.81515947]
y_pred.shape =  (2, 10, 4)
caches[1][1][3] =  [-1.1425182  -0.34934272 -0.20889423  0.58662319]
len(caches) =  2
```

2.1.2 Long Short Term Memory (LSTM) Network

Tidak seperti RNN biasa, LSTM-cell melacak dan memperbarui "cell state" atau memory cell pada setiap time step.



$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\langle t \rangle} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C)$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

Gambar 7. Strutur dan rumus LSTM Cell Forward

LSTM Cell Forward : Mengimplementasikan single forward step pada LSTM cell

Algoritmanya hampir sama dengan algoritma RNN Cell Forward, bedanya adalah dalam algoritma ini mengimplementasikan memory state dan tiga gates seperti yang telah dijelaskan dalam bab Introduction. Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan gambaran komputasi diatas :

- xt = input data pada time step (t)
- a_prev = hidden state pada time step sebelumnya (t-1)
- c_prev = memory state pada time step sebelumnya (t-1)
- Wf = matriks weight pada forget gate
- bf = bias pada forget gate
- Wi = matriks weight pada update gate
- bi = bias pada update gate
- Wc = matriks weight pada "tanh" pertama

- bc = bias pada “tanh” pertama
- Wo = matriks weight pada output gate
- bo = bias pada output gate
- Wy = matriks weight antara hidden state dan output
- by = bias antara hidden state dan output
- a_next = hidden state selanjutnya
- c_next = memory state selanjutnya
- yt_pred = prediksi pada time step (t)
- cache = nilai tuple yang diperlukan untuk backward pass

```
def lstm_cell_forward(xt, a_prev, c_prev, parameters) :
    # Mengambil parameter
    Wf = parameters["Wf"]
    bf = parameters["bf"]
    Wi = parameters["Wi"]
    bi = parameters["bi"]
    Wc = parameters["Wc"]
    bc = parameters["bc"]
    Wo = parameters["Wo"]
    bo = parameters["bo"]
    Wy = parameters["Wy"]
    by = parameters["by"]

    # Mengambil ukuran bentuk xt and Wy
    n_x, m = xt.shape
    n_y, n_a = Wy.shape

    # Concat a_prev dan xt
    concat = np.zeros((n_x + n_a, m))
    concat[: n_a, :] = a_prev
    concat[n_a :, :] = xt

    # Menghitung nilai untuk ft, it, cct, c_next, ot, a_next
    ft = sigmoid(np.dot(Wf, concat)+bf)
    it = sigmoid(np.dot(Wi, concat)+bi)
    cct = np.tanh(np.dot(Wc, concat)+bc)
    c_next = c_prev*ft + it*cct
    ot = sigmoid(np.dot(Wo,concat)+bo)
    a_next = ot*np.tanh(c_next)

    # Menghitung prediksi LSTM cell
    yt_pred = softmax(np.dot(Wy,a_next)+by)

    # Menyimpan nilai untuk backward propagation pada cache
    cache = (a_next, c_next, a_prev, c_prev, ft, it, cct, ot, xt, parameters)
```

```
    return a_next, c_next, yt_pred, cache
```

Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung a_next, c_next dan yt_pred dengan rumus yang digunakan dalam LSTM Cell Forward.

```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
c_prev = np.random.randn(5,10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5,1)
Wi = np.random.randn(5,5+3)
bi = np.random.randn(5,1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5,1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5,1)
Wy = np.random.randn(2,5)
by = np.random.randn(2,1)

parameters = {"Wf": Wf, "Wi" : Wi, "Wo" : Wo, "Wc" : Wc, "Wy" : Wy, "bf" : bf, "bi" : bi, "bo" : bo, "bc" : bc, "by" : by}

a_next, c_next, yt, cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)
print("a_next[4] = ", a_next[4])
print("a_next.shape = ", a_next.shape)
print("c_next[2] = ", c_next[2])
print("c_next.shape = ", c_next.shape)
print("yt[1] = ", yt[1])
print("yt.shape = ", yt.shape)
print("cache[1][3] = ", cache[1][3])
print("len(cache) = ", len(cache))
```

```
a_next[4] =  [-0.66408471  0.0036921   0.02088357  0.22834167 -0.85575339  0.00138482
  0.76566531  0.34631421 -0.00215674  0.43827275]
a_next.shape =  (5, 10)
c_next[2] =  [ 0.63267805  1.00570849  0.35504474  0.20690913 -1.64566718  0.11832942
  0.76449811 -0.0981561  -0.74348425 -0.26810932]
c_next.shape =  (5, 10)
yt[1] =  [0.79913913 0.15986619 0.22412122 0.15606108 0.97057211 0.31146381
 0.00943007 0.12666353 0.39380172 0.07828381]
yt.shape =  (2, 10)
cache[1][3] =  [-0.16263996  1.03729328  0.72938082 -0.54101719  0.02752074 -0.30821874
  0.07651101 -1.03752894  1.41219977 -0.37647422]
len(cache) =  10
```

Forward Pass for LSTM : Mengimplementasikan forward propagation pada RNN yang menggunakan LSTM cell

Mengiterasikan algoritma sebelumnya untuk memproses Tx input secara sekuensial. Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan gambaran komputasi diatas :

- x = input data untuk setiap time step (t)
- a0 = hidden state awal (t-1)
- Wf = matriks weight pada forget gate
- bf = bias pada forget gate
- Wi = matriks weight pada update gate
- bi = bias pada update gate
- Wc = matriks weight pada “tanh” pertama
- bc = bias pada “tanh” pertama
- Wo = matriks weight pada output gate
- bo = bias pada output gate
- Wy = matriks weight antara hidden state dan output
- by = bias antara hidden state dan output
- a = hidden state untuk setiap time step
- y = prediksi untuk setiap time step
- caches = nilai tuple yang diperlukan untuk backward pass

```
def lstm_forward(x, a0, parameters):
    # Menginisialisasi "caches" yang akan menyimpan semua cache
    caches = []

    # Mengambil ukuran bentuk x dan paramters['Wy']
    n_x, m, T_x = x.shape
    n_y, n_a = parameters["Wy"].shape

    # Menginisialisasi "a", "c" dan "y" dengan zeros

    a = np.zeros((n_a, m, T_x))
    c = np.zeros((n_a, m, T_x))
    y = np.zeros((n_y, m, T_x))

    # Menginisialisasi a_next dan c_next
    a_next = a0
    c_next = np.zeros((n_a, m))

    #loop sebanyak time steps
    for t in range(0, T_x):
        # Memperbarui hidden state selanjutnya dan memory state selanjutnya
        a_next, c_next, yt, cache = lstm_cell_forward(x[:, :, t], a_next, c_next,
```

```
parameters)
        # Menyimpan nilai hidden state selanjutnya
        a[:,:,t] = a_next
        # Menyimpan nilai prediksi
        y[:,:,t] = yt
        # Menyimpan nilai cell state selanjutnya
        c[:,:,t] = c_next
        # Menambahkan cache ke daftar cache
        caches.append(cache)

    # Menyimpan nilai yang dibutuhkan untuk backward propagation pada cache
    caches = (caches, x)

    return a, y, c, caches
```

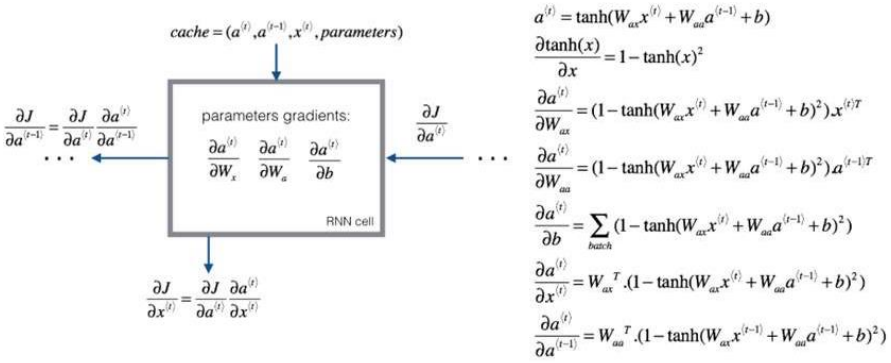
Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung a, c dan y dengan rumus yang digunakan dalam LSTM Forward.

```
np.random.seed(1)
x = np.random.randn(3,10,7)
a0 = np.random.randn(5,10)
Wf = np.random.randn(5,5+3)
bf = np.random.randn(5,1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5,1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5,1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5,1)
Wy = np.random.randn(2,5)
by = np.random.randn(2,1)

parameters = {"Wf": Wf, "Wi":Wi, "Wo":Wo, "Wc":Wc, "Wy":Wy, "bf":bf,
"bi":bi, "bo":bo, "bc":bc, "by":by}

a, y, c, caches = lstm_forward(x, a0, parameters)
print("a[4][3][6] = ", a[4][3][6])
print("a.shape = ", a.shape)
print("y[1][4][3] = ", y[1][4][3])
print("y.shape = ", y.shape)
print("caches[1][1][1] = ", caches[1][1][1])
print("c[1][2][1]", c[1][2][1])
print("len(caches) = ", len(caches))
```

```
a[4][3][6] =  0.17211776753291672
a.shape =  (5, 10, 7)
y[1][4][3] =  0.9508734618501101
y.shape =  (2, 10, 7)
caches[1][1][1] =  [ 0.82797464  0.23009474  0.76201118 -0.22232814 -0.20075807  0.18656139
  0.41005165]
c[1][2][1] -0.8555449167181981
len(caches) =  2
```

2.1.3 Backpropagation pada RNN

Basic RNN Backward Pass



$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)$$
$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2$$
$$\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} = (1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2).x^{\langle t \rangle T}$$
$$\frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} = (1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2).a^{\langle t-1 \rangle T}$$
$$\frac{\partial a^{\langle t \rangle}}{\partial b} = \sum_{batch} (1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)$$
$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}{}^{T}.(1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)$$
$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}{}^{T}.(1 - \tanh(W_{ax}x^{\langle t-1 \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)$$

Gambar 9. Struktur dan rumus RNN Cell Backward

RNN Cell Backward : Mengimplementasikan backward pass untuk RNN Cell (single time step)

Dalam tahap ini, dimulai dengan mengambil nilai yang dihitung pada tahap RNN Cell Forward yang disimpan dalam cache. Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan gambaran komputasi diatas :

- da_next = Gradient yang hilang dari hidden state selanjutnya
- cache = output dari rnn_cell_forward()
- dx = gradients dari data input
- da_prev = gradients dari hidden state sebelumnya
- dWax = gradient pada input ke hidden weights
- dWaa = gradient pada hidden to hidden weights
- dba = gradient pada vektor bias

```
def rnn_cell_backward(da_next, cache):
    # Mengambil nilai dari cache
    (a_next, a_prev, xt, parameters) = cache

    # Mengambil nilai dari parameter
    Wax = parameters["Wax"]
```

```
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    # Menghitung gradient tanh dengan memperhitungkan a_next
    dtanh = 1 - np.power(a_next,2)

    # Menghitung gradient hilang dengan memperhitungakn Wax
    dxt = np.dot(Wax.T, da_next*dtanh)
    dWax = np.dot(da_next*dtanh, xt.T)

    # Menghitung gradient dengan memperhitungkan Waa
    da_prev = np.dot(Waa.T, da_next*dtanh)
    dWaa = np.dot(da_next*dtanh, a_prev.T)

    # Menghitung gradient dengan memperhitungkan bias
    dba = np.sum(da_next*dtanh, axis=1, keepdims=True)

    # Menyimpan gradient ke kamus "gradients"
    gradients = {"dxt":dxt, "da_prev":da_prev, "dWax":dWax, "dWaa":dWaa,
"dba":dba}

    return gradients
```

Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung gradients dengan rumus yang digunakan dalam RNN Backward Pass.

```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Wax = np.random.randn(5,3)
Waa = np.random.randn(5,5)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Wax":Wax, "Waa":Waa, "Wya":Wya, "ba":ba, "by":by}

a_next, yt, cache = rnn_cell_forward(xt, a_prev, parameters)

da_next = np.random.randn(5,10)
gradients = rnn_cell_backward(da_next, cache)
print("gradients[\"dxt\"][1][2] =", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape =", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3] =", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape =", gradients["da_prev"].shape)
```

```
print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape =", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape =", gradients["dWaa"].shape)
print("gradients[\"dba\"][4] =", gradients["dba"][4])
print("gradients[\"dba\"].shape =", gradients["dba"].shape)
```

```
gradients["dxt"][1][2] = -1.3872130506020928
gradients["dxt"].shape = (3, 10)
gradients["da_prev"][2][3] = -0.15239949377395473
gradients["da_prev"].shape = (5, 10)
gradients["dWax"][3][1] = 0.41077282493545836
gradients["dWax"].shape = (5, 3)
gradients["dWaa"][1][2] = 1.1503450668497135
gradients["dWaa"].shape = (5, 5)
gradients["dba"][4] = [0.20023491]
gradients["dba"].shape = (5, 1)
```

Backward Pass melalui RNN

Dalam tahap ini, menghitung gradients dengan memperhitungkan (a(t)) pada setiap time steps (t) akan berguna untuk membantu gradient back propagate ke RNN cell sebelumnya. Untuk melakukannya, harus iterasi melalui setiap time steps dimulai dari akhir dan pada setiap step-nya lakukan increment untuk nilai dba, dWaa, dWax, dan simpan dx.

RNN Backward : Mengimplementasikan backward pass untuk RNN ke seluruh sekuen input data

Algoritmanya hampir sama dengan algoritma sebelumnya, bedanya hanya algoritma ini menggunakan nilai hasil pemrosesan di RNN Forward yang disimpan dalam cache. Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan gambaran komputasi diatas untuk seluruh time steps :

- da = Upstream gradient dari semua hidden states
- caches = output dari rnn_forward()
- dx = gradients dari data input
- da0 = gradients dari hidden state awal
- dWax = gradient pada input ke hidden weights
- dWaa = gradient pada hidden to hidden weights
- dba = gradient pada vektor bias

```
def rnn_backward(da, caches) :
  # Mengambil nilai dari cache pertama
```

```
    (caches, x) = caches
    (a1, a0, x1, parameters) = caches[1]

    # Mengambil ukuran bentuk da dan x1
    n_a, m, T_x = da.shape
    n_x, m = x1.shape

    # Menginisialisasi gradient
    dx = np.zeros((n_x, m, T_x))
    dWax = np.zeros((n_a, n_x))
    dWaa = np.zeros((n_a, n_a))
    dba = np.zeros((n_a, 1))
    da0 = np.zeros((n_a, m))
    da_prevt = np.zeros((n_a, m))

    for t in reversed(range(0, T_x)):
        # Menghitung gradient pada time step
        gradients = rnn_cell_backward(da[:,:,t] + da_prevt, caches[t])
        # Mengambil derivatif dari gradient
        dxt, da_prevt, dWaxt, dWaat, dbat = gradients["dxt"], gradients["da_prev"],
gradients["dWax"], gradients["dWaa"], gradients["dba"]
        # Meng-increment setiap nilai global derivatif
        dx[:,:,t] = dxt
        dWax += dWaxt
        dWaa += dWaat
        dba += dbat

    # Set da0 untuk gradient yang telah back propagated melalui semua time steps
    da0 = da_prevt

    # Menyimpan gradient dalam python dictionary
    gradients = {"dx":dx, "da0":da0, "dWax":dWax, "dWaa":dWaa, "dba":dba}

    return gradients
```

Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung gradients dengan rumus yang digunakan dalam RNN Backward.

```
np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Wax = np.random.randn(5,3)
Waa = np.random.randn(5,5)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
```

```
parameters = {"Wax":Wax, "Waa":Waa, "Wya":Wya, "ba":ba, "by":by}
a, y, caches = rnn_forward(x, a0, parameters)
da = np.random.randn(5,10,4)
gradients = rnn_backward(da, caches)

print("gradients[\"dx\"][1][2] =", gradients["dx"][1][2])
print("gradients[\"dx\"].shape =", gradients["dx"].shape)
print("gradients[\"da0\"][2][3] =", gradients["da0"][2][3])
print("gradients[\"da0\"].shape =", gradients["da0"].shape)
print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape =", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape =", gradients["dWaa"].shape)
print("gradients[\"dba\"][4] =", gradients["dba"][4])
print("gradients[\"dba\"].shape =", gradients["dba"].shape)
```

```
gradients["dx"][1][2] = [-2.07101689 -0.59255627  0.02466855  0.01483317]
gradients["dx"].shape = (3, 10, 4)
gradients["da0"][2][3] = -0.31494237512664996
gradients["da0"].shape = (5, 10)
gradients["dWax"][3][1] = 11.264104496527777
gradients["dWax"].shape = (5, 3)
gradients["dWaa"][1][2] = 2.3033331265798935
gradients["dWaa"].shape = (5, 5)
gradients["dba"][4] = [-0.74747722]
gradients["dba"].shape = (5, 1)
```

LSTM Cell Backward : Mengimplementasikan backward pass untuk LSTM Cell (Single time step)

Berikut nama variabel dan artinya yang digunakan untuk mengimplementasikan komputasi LSTM Backward Pass :

- da_next = Gradient hidden state selanjutnya
- dc_next = Gradient cell state selanjutnya
- cache = output dari rnn_forward()
- dbo = gradient bias pada output gate
- dxt = gradients dari data input pada time step (t)
- da_prev = Gradient dari hidden state sebelumnya
- dc_prev = gradient dari cell state sebelumnya
- dWf = matriks weight dari forget gate
- dWi = matriks weight dari update gate
- dWc = matriks weight dari memory gate
- dWo = matriks weight dari output gate
- dbf = gradient bias dari forget gate
- dbi = gradient bias dari update gate

- dbc = gradient bias dari memory gate
- dbo = gradient bias dari output gate

```
def lstm_cell_backward(da_next, dc_next, cache) :
    # Mengambil informasi dari cache hasil LSTM Forward Pass
    (a_next, c_next, a_prev, c_prev, ft, it, cct, ot, xt, parameters) = cache

    # Mengambil ukuran bentuk xt dan a_next
    n_x, m = xt.shape
    n_a, m = a_next.shape

    # Menghitung gates dari derivatif terkait
    dot = da_next*np.tanh(c_next)
    dcct = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*it
    dit = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*cct
    dft = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*c_prev

    dit = dit*it*(1-it)
    dft = dft*ft*(1-ft)
    dot = dot*ot*(1-ot)
    dcct = dcct*(1-np.power(cct, 2))

    # Menghitung parameter dari derivatif terkait
    concat = np.zeros((n_x + n_a, m))
    concat[: n_a, :] = a_prev
    concat[n_a :, :] = xt
    dWf = np.dot(dft, concat.T)
    dWi = np.dot(dit, concat.T)
    dWc = np.dot(dcct, concat.T)
    dWo = np.dot(dot, concat.T)
    dbf = np.sum(dft, axis=1, keepdims=True)
    dbi = np.sum(dit, axis=1, keepdims=True)
    dbc = np.sum(dcct, axis=1, keepdims=True)
    dbo = np.sum(dot, axis=1, keepdims=True)

    # Menghitung derivatif hidden state sebelumnya, memory state sebelumya,
dan input
    da_prevx = np.dot(parameters['Wf'].T, dft) + np.dot(parameters['Wo'].T, dot)
+ np.dot(parameters['Wi'].T, dit) + np.dot(parameters['Wc'].T, dcct)
    da_prev = da_prevx[: n_a, :]
    dc_prev = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*ft
    dxt = da_prevx[n_a :, :]

    # Menyimpan gradient ke kamus "gradients"
    gradients = {"dxt":dxt, "da_prev":da_prev, "dc_prev":dc_prev, "dWf":dWf,
"dbf":dbf, "dWi":dWi, "dbi":dbi, "dWc":dWc, "dbc":dbc, "dWo":dWo,
```

```
"dbo":dbo}

    return gradients
```

Langkah selanjutnya adalah mengalokasikan nilai random ke parameter dan menghitung gradients dengan rumus yang digunakan dalam RNN LSTM Cell Backward.

```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
c_prev = np.random.randn(5,10)
Wf = np.random.randn(5,5+3)
bf = np.random.randn(5,1)
Wi = np.random.randn(5,5+3)
bi = np.random.randn(5,1)
Wo = np.random.randn(5,5+3)
bo = np.random.randn(5,1)
Wc = np.random.randn(5,5+3)
bc = np.random.randn(5,1)
Wy = np.random.randn(2,5)
by = np.random.randn(2,1)

parameters = {"Wf":Wf, "Wi":Wi, "Wo":Wo, "Wc":Wc, "Wy":Wy, "bf":bf, "bi":bi, "bo":bo, "bc":bc, "by":by}

a_next, c_next, yt, cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)

da_next = np.random.randn(5,10)
dc_next = np.random.randn(5,10)
gradients = lstm_cell_backward(da_next, dc_next, cache)

print("gradients[\"dxt\"][1][2] =", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape =", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3] =", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape =", gradients["da_prev"].shape)
print("gradients[\"dc_prev\"][2][3] =", gradients["dc_prev"][2][3])
print("gradients[\"dc_prev\"].shape =", gradients["dc_prev"].shape)
print("gradients[\"dWf\"][3][1] =", gradients["dWf"][3][1])
print("gradients[\"dWf\"].shape =", gradients["dWf"].shape)
print("gradients[\"dWi\"][1][2] =", gradients["dWi"][1][2])
print("gradients[\"dWi\"].shape =", gradients["dWi"].shape)
print("gradients[\"dWc\"][3][1] =", gradients["dWc"][3][1])
print("gradients[\"dWc\"].shape =", gradients["dWc"].shape)
print("gradients[\"dWo\"][1][2] =", gradients["dWo"][1][2])
print("gradients[\"dWo\"].shape =", gradients["dWo"].shape)
print("gradients[\"dbf\"][4] =", gradients["dbf"][4])
```

```
print("gradients[\"dbf\"].shape =", gradients["dbf"].shape)
print("gradients[\"dbi\"][4] =", gradients["dbi"][4])
print("gradients[\"dbi\"].shape =", gradients["dbi"].shape)
print("gradients[\"dbc\"][4] =", gradients["dbc"][4])
print("gradients[\"dbc\"].shape =", gradients["dbc"].shape)
print("gradients[\"dbo\"][4] =", gradients["dbo"][4])
print("gradients[\"dbo\"].shape =", gradients["dbo"].shape)
```

```
gradients["dxt"][1][2] = 3.230559115109188
gradients["dxt"].shape = (3, 10)
gradients["da_prev"][2][3] = -0.06396214197109235
gradients["da_prev"].shape = (5, 10)
gradients["dc_prev"][2][3] = 0.7975220387970015
gradients["dc_prev"].shape = (5, 10)
gradients["dWf"][3][1] = -0.1479548381644968
gradients["dWf"].shape = (5, 8)
gradients["dWi"][1][2] = 1.0574980552259903
gradients["dWi"].shape = (5, 8)
gradients["dWc"][3][1] = 2.304562163687667
gradients["dWc"].shape = (5, 8)
gradients["dWo"][1][2] = 0.3313115952892109
gradients["dWo"].shape = (5, 8)
gradients["dbf"][4] = [0.18864637]
gradients["dbf"].shape = (5, 1)
gradients["dbi"][4] = [-0.40142491]
gradients["dbi"].shape = (5, 1)
gradients["dbc"][4] = [0.25587763]
gradients["dbc"].shape = (5, 1)
gradients["dbo"][4] = [0.13893342]
gradients["dbo"].shape = (5, 1)
```

2.2 Character Level Language Model – Dinosaurus Land

Dalam projek ini, saya akan mengimplementasikan RNN untuk melakukan teks recognition dan generate nama dinosarurus berdasarkan pola dalam teks tersebut. Bahan yang dipakai dalam projek ini adalah :

- dinos.txt : Sumber teks yang akan diproses
- utils.py : Terdapat fungsi rnn_forward dan rnn_backward yang akan dipakai dalam program
- rnn_utils.py

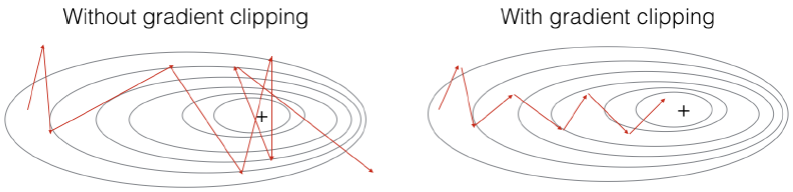
Pada dinos.txt terdapat 27 karakter unik yaitu a-z dan \n (karakter newline) untuk menandakan akhir dari nama dinosaurus yang ada pada teks.

Model yang dibuat akan memiliki struktur sebagai berikut :

- Menginisialisasi parameter
- Menjalankan pengulangan untuk optimisasi
  - Menghitung loss function dengan forward propagation
  - Menghitung gradient dengan memperhitungkan loss function dengan backward propagation
  - Clip gradient untuk menghidari exploding gradients
  - Memperbarui parameters dengan rule gradient descent
- Return parameter yang dipelajari

Terdapat dua building block yang penting dalam model, yaitu Gradient Clipping dan Sampling. Gradient Clipping digunakan untuk menghindari masalah exploding gradients, dan Sampling digunakan untuk generate karakter.

Proses Gradient Clipping dipanggil dalam setiap loop optimisasi, tepatnya sebelum memperbarui nilai parameter untuk memastikan nilai gradient tidak “explode” atau mengambil nilai yang terlalu besar. Dalam program ini, fungsi *clip* dibuat dengan menetapkan nilai maxValue menjadi 10 dan -10 sehingga jika ada nilai yang melebihi 10 atau -10 maka nilai tersebut akan diubah ke 10 dan -10.



Gambar 10. Pengaruh gradient clipping

Building block yang kedua adalah sampling. Sampling digunakan untuk generate

karakter baru setelah model dilatih. Berikut tahapan sampling :

- Tahap 1 : Memberikan input “dummy” pertama dengan nilai vector of zeros ke network.
- Tahap 2 : Menjalakan one step forward propagation
- Tahap 3 : Melakukan sampling dengan memilih index karakter selanjutnya sesuai dengan distribusi probabilitas yang dihasilkan dari tahap dua.
- Tahap 4 : Mengimplementasikan sample() dengan cara overwrite variabel x (t) dengan x (t+1) dengan membuat one-hot vector sesuai dengan karakter yang dipilih sebagai prediksi. Lalu lakukan forward propagate dalam tahap 1 dan terus ulang sampai mendapatkan token newline.

Setelah membuat building block, langkah yang dilakukan adalah membuat model bahasa yang akan digunakan untuk generate teks (nama baru untuk dinosaurus). Langkah pertama adalah membuat fungsi one step of Stochastic Gradient Descent sebagai algoritma optimisasi. Di fungsi ini juga metode Clipped Gradient digunakan. Nilai gradient yang dihasilkan digunakan untuk tahap selanjutnya yang juga tahap terakhir yaitu melatih model. Pada tahap tersebut, setiap baris (satu nama dinosaurus) dalam dataset yang diberikan akan dijadikan satu training example. Dan setiap 50 steps stochastic gradient descent, program akan sample 7 nama random. Saya menggunakan kode berikut untuk membuat satu example (X, Y) ketika ada examples[index] yang terdapat satu nama dinosaurus.

```
index = j % len(examples)
    X = [None] + [char_to_ix[ch] for ch in examples[index]]
    Y = X[1:] + [char_to_ix["\n"]]
```

Kode tersebut diimplementasikan di fungsi bernama model. İndex = j mod len(examples) digunakan untuk memastikan examples[index] selalu valid. Entri pertama dari X bernilai none akan diinterpretasikan oleh fungsi rnn_forward(). Sehinga memastikan Y sama dengan X yang digeser satu step ke kiri dengan tambahan token newline.

Setelah program dijalankan, dapat dilihat bahwa iterasi pertama model hanya generate teks random yang tidak menandakan nama dinosaurus, tetapi lama kelamaan iterasi selanjutnya model berhasil generate nama yang menandakan nama dinosaurus seperti nama dengan akhiran saurus, don, aura, tor, dsb.

2.3 RNN Using LSTM For $TATA Stock Prediction

Projek ini mengimplementasikan Long Short-Term Memory RNN untuk memprediksi harga saham. Dataset yang digunakan adalah data saham $TATA. Library yang digunakan dalam program ini adalah NumPy untuk komputasi ilmiah, MatPlotLib untuk membuat diagram, dan Pandas untuk memanipulasi data, MinMaxScaler untuk feature scaling, dan modules yang disediakan Keras. Module

tersebut adalaah Sequential untuk menginisialisasi neural network, Dense untuk menambahkan neural network layer yang *densely connected*, LSTM untuk manambahkan Long Short-Term Memory layer, dan Dropout untuk menghindari *overfitting.*

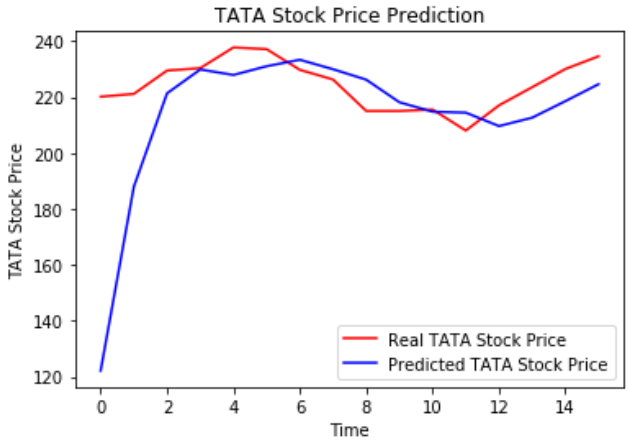
Tahap pertama dalam program ini adalah load data training dan test, namun atribut yang digunakan dalam model pembelajaran adalah Open dan High saja. Sebelum lanjut ke tahap berikutnya, data training set discale dengan fungsi MinMaxScaler. Proses ini dilakukan dalam tahap preproses untuk normalisasi data jika nilai atribut-atribut berbeda-beda magnitude-nya. Feature scaling berfungsi untuk menghindari masalah tersebut dan juga dapat membantu kecepatan kalkulasi dalam algoritma pembelajaran. Hasil scaling dialokasikan ke array X_train dan y_train dalam bentuk 3D array agar dapat diterima model pembelajaran LSTM.

Dalam proses pembuatan model LSTM. LSTM layer ditambahkan dengan nilai parameter sebagai berikut :

- Dimensionality of the output space = 50 units
- return_sequences = true, sehingga akan return last output dalam output sequence.
- input_shape = menyesuaikan dengan training set yang telah dibuat.

Selanjutnya, nilai Dropout layer diset ke 0.2, yang artinya 20% dari layers akan dibuang. Sehingga dapat ditambahkan Dense layer yang menetapkan 1 unit output. Setelah itu, model di-compile menggunakan metode Adam optimizer dan set mean_squared_error yang menghitung rata-rata dari squared error. Terakhir, model akan di-fit untuk berjalan untuk 100 epochs dengan setiap batch size nya berjumlah 32.

Setelah proses training selesai, model yang dihasilkan digunakan untuk memprediksi harga saham TATA pada dataset *test*. Hasil prediksi dibandingkan dengan data real_stock_price untuk mengetahui performa model yang dibuat. Sebelum melakukan visualisasi, hasil prediksi diubah ke format data yang lebih mudah dibaca dengan fungsi inverse_transform. Data hasil prediksi dibandingkan dengan data nyata dalam periode waktu yang sama dengan diagram yang dibuat menggunakan library MatPlotLib. Berikut hasilnya :

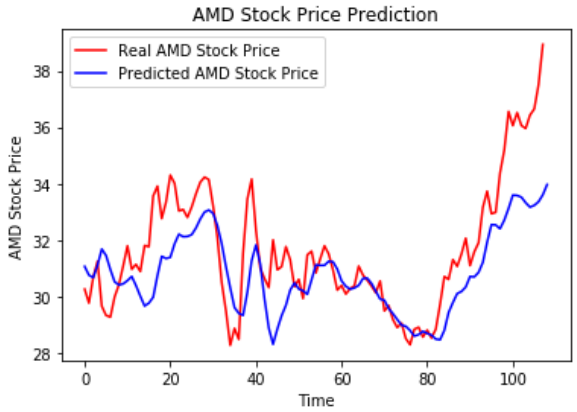


Gambar 11. Perbandingan harga saham TATA yang diprediksi dengan yang nyata

Pada graph tersebut, dapat dilihat dari naik turunnya garis bahwa hasil prediksi sangat mendekati harga saham nyatanya. Menunjukan bahwa LSTM merupakan model yang powerful dalam menghadapi time-series dan data sekuensial.

2.4 RNN Using LSTM For $AMD Stock Prediction

Project ini mengulangi tahapan yang dikerjakan dalam projek 2.3 namun dilakukan terhadap saham yang berbeda. Dalam project ini saya menggunakan saham AMD. Dataset train dimulai dari tanggal 17 Oktober tahun 2011 sampai 14 Juni 2019, sedangkan dataset test dimulai dari tanggal 17 Juni 2019 sampai 15 November 2019.



Gambar 12. Perbandingan harga saham AMD yang diprediksi dengan yang nyata

Diagram tersebut menunjukkan bahwa hasil prediksi tidak jauh dari harga saham nyata dalam periode yang sama. Namun terdapat waktu dimana harga saham prediksi berlawanan dengan harga saham nyatanya. Setelah saya cek, ternyata pada waktu tersebut terdapat berita bagus untuk perusahaan AMD sehingga saham naik daripada turun seperti yang hasil prediksi.

3 Summary

Dengan membangun 4 projek diatas yaitu projek membangun fungsi RNN, mengimplementasikan RNN pada text recognition untuk generate nama, dan mengimplementasikan RNN untuk memprediksi harga saham dari saham $AMD dan $TATA, dapat dilihat bahwa RNN adalah model pembelajaran yang sangat bagus untuk digunakan pada task sekuensial atau dengan kata lain, task dimana tuple sebelumnya dapat mempengaruhi tuple setelahnya. Meskipun begitu, RNN 'vanilla' tidak cocok digunakan terhadap dataset yang besar karena masalah vanishing gradients. Masalah tersebut dapat dihindari dengan menggunakan LSTM RNN yang menyempurnakan RNN dengan menggunakan mekanisme gates untuk menghindari masalah *vanishing gradients.* Selain masalah tersebut, dalam RNN terdapat masalah yang disebut sebagai exploding gradients. Tetapi seperti yang dijelaskan dalam projek kedua, masalah tersebut dapat dengan mudah dihindari dengan menggunakan metode gradient clipping, sehingga nilai gradient lebih stabil. Kita juga belajar bagaimana data preprocessing dalam pembuatan model deep learning (RNN).