

SLIDE 1

RNN pertama kali diusulkan pada tahun 1985. Pada saat itu, unit dasar RNN **masih sama seperti unit jaringan neural biasa**.

Seiring waktu, RNN dikembangkan menjadi berbagai versi yang lebih canggih, yaitu:

- **LSTM (Long Short-Term Memory)**: setiap unitnya memiliki **3 gate**.
- **GRU (Gated Recurrent Unit)**: versi yang lebih ringan, hanya menggunakan **2 gate**.
- Dan yang lebih baru, dikembangkan lagi menjadi **Recurrent Highway Network** atau **Recurrent Unit dengan gating yang lebih kompleks**, kadang juga disebut **Gated Linear Unit (GLU)**.

RNN dan turunannya dirancang khusus untuk **menangani data sekuensial (sequential data)**.

SLIDE 2

Feedforward adalah proses dalam jaringan saraf di mana data mengalir satu arah dari input ke output tanpa ada aliran balik atau ingatan dari masa lalu. Bayangkan seperti seseorang yang membaca buku dari halaman pertama sampai terakhir tanpa pernah kembali ke halaman sebelumnya—informasi hanya mengalir maju. Sebaliknya, **feedback** (seperti pada Recurrent Neural Network) melibatkan aliran balik, di mana output sebelumnya digunakan kembali sebagai input berikutnya. Ini mirip dengan seseorang yang menulis jurnal harian dan selalu membaca catatan kemarin sebelum menulis hari ini, sehingga apa yang telah terjadi memengaruhi keputusan selanjutnya. Dengan cara ini, jaringan memiliki semacam "ingatan" untuk memahami data berurutan seperti teks atau suara.

Pada RNN, kita memproses input **berurutan dari waktu ke waktu** (disebut **time steps**), dan output di tiap time step **dipengaruhi oleh hasil sebelumnya**.

Ini disebut **backward link**.

Ketika kita menggunakan RNN dengan N time steps, sebenarnya model terdiri dari **N jaringan yang identik (parameter sharing)**.

Artinya, setiap time step menggunakan **parameter yang sama**:

SLIDE 3

Tiga jenis data yang ditampilkan di slide ini adalah:

1. **Data suara (speech/audio waveform)** – ditampilkan dalam bentuk gelombang suara. Data ini bersifat waktu-nyata dan memiliki dependensi antara satu frame dengan frame berikutnya, sehingga cocok diproses oleh RNN untuk tugas seperti speech recognition.
2. **Video atau rangkaian gambar (frame-by-frame video)** – ditampilkan dalam bentuk cuplikan gambar berurutan. Video merupakan kumpulan frame yang membentuk aliran waktu,

sehingga konteks dari frame sebelumnya sangat membantu dalam memahami frame berikutnya, misalnya untuk action recognition.

3. **Sinyal medis seperti EKG (Electrocardiogram)** – ditampilkan sebagai grafik sinyal jantung. Sinyal ini memiliki pola berulang dan membutuhkan model yang bisa mengingat pola-pola sebelumnya untuk mendeteksi anomali atau perubahan kondisi.

Ketiganya menggambarkan karakteristik utama dari data yang cocok untuk RNN, yaitu data yang memiliki **urutan, waktu, dan ketergantungan antar elemen**.

SLIDE 4

Slide ini menjelaskan bagaimana arsitektur Recurrent Neural Network (RNN) bekerja dengan memanfaatkan konsep **time step**, yaitu urutan waktu yang sangat penting dalam pemrosesan data sekuensial. Di bagian atas, kita melihat contoh kalimat “Aku suka belajar Machine Learning”, yang setiap katanya diberi nomor sesuai urutannya, dari 1 hingga 5. Ini menggambarkan bahwa RNN memproses input **satu per satu secara berurutan**, dan setiap kata dalam kalimat akan diproses pada satu time step.

“Aku suka belajar RNN.”

Kalimat ini terdiri dari 4 kata.

Maka, dalam RNN akan ada **4 time step**:

- Time step 1 → "Aku"
 - Time step 2 → "suka"
 - Time step 3 → "belajar"
 - Time step 4 → "RNN"

👉 Pada **setiap time step**, RNN memproses satu elemen dari urutan data — bisa kata, angka, frame video, dll.

Di setiap **time step** ttt dalam RNN, model:

- Menerima input xtx_txt
 - Menggunakan hidden state sebelumnya $ht-1h_{\{t-1\}}ht-1$
 - Menghasilkan **hidden state saat ini** hth_{tht}

→ Jadi, satu time step menghasilkan satu hidden state

SLIDE 5

Dalam Recurrent Neural Network (RNN), proses pemrosesan data berlangsung secara bertahap pada setiap **time step**. Pada satu time step, model menerima input x_t (misalnya kata ke- t dalam sebuah kalimat) dan hidden state dari time step sebelumnya, yaitu h_{t-1} , yang berperan sebagai "memori" dari informasi sebelumnya. Untuk menghasilkan hidden state saat ini (h_t), input dan hidden state sebelumnya diproses menggunakan persamaan:

$$h_t = f(W_{xh} \cdot x_t + W_h \cdot h_{t-1} + b_h)$$

Di sini, W_{xh} adalah bobot dari input ke hidden layer, W_h adalah bobot antar hidden state, b_h adalah bias, dan f adalah fungsi aktivasi (biasanya tanh atau ReLU) yang membantu menormalkan hasilnya agar stabil dalam jaringan.

Setelah mendapatkan hidden state h_t , RNN akan menghitung **output** pada time step tersebut, yaitu y_t , menggunakan rumus:

$$y_t = g(W_{hy} \cdot h_t + b_y)$$

Bobot W_{hy} menghubungkan hidden state ke output layer, b_y adalah bias untuk output, dan g merupakan fungsi aktivasi (seperti softmax untuk klasifikasi atau linear untuk regresi).

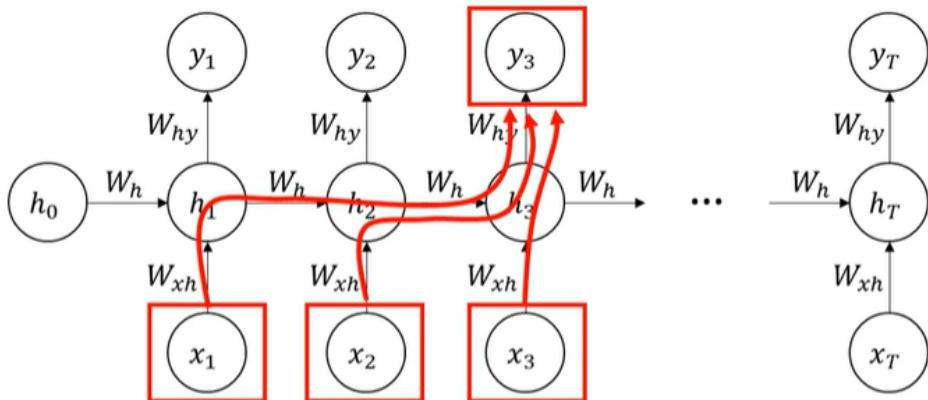
Output y_t ini berfungsi sebagai hasil akhir prediksi RNN pada time step tersebut. Dalam banyak aplikasi, y_t bisa digunakan langsung (misalnya dalam prediksi kata selanjutnya dalam teks), atau dikumpulkan dari semua time step untuk diproses lebih lanjut, seperti pada machine translation, speech recognition, atau

Ask anything

Tujuan RNN adalah **belajar pola berulang dalam urutan data**.

Dengan menggunakan parameter yang sama (weight dan bias sama), model bisa **generalize** pola di seluruh urutan, bukan hanya menghafal posisi.

Pada RNN setiap output dipengaruhi time step sebelum sebelumnya, misalnya y_3 disini dipengaruhi oleh input sekarang dan juga input sebelumnya yaitu x_1, x_2, x_3



dari input sekarang dan juga sebelumnya yaitu x_1, x_2 , dan x_3 .

SLIDE 6

Nah, teman-teman pasti bertanya-tanya, dalam arsitektur RNN ini sebenarnya **output yang mana sih yang digunakan?** Apakah semuanya (y_1, y_2, \dots, y_n) akan dipakai, atau hanya sebagian? Jawabannya tergantung pada jenis arsitektur RNN yang kita gunakan, karena setiap bentuk arsitektur punya tujuan dan karakteristik yang berbeda-beda.

Pada model **One to Many**, kita memberikan satu input tetap, seperti gambar atau noise acak, lalu RNN menghasilkan banyak output secara bertahap. Contohnya bisa kita temukan dalam **image captioning** atau **music generation**, di mana seluruh output y_1 sampai y_n penting karena merupakan bagian dari hasil akhir yang ingin kita bangun secara berurutan.

Sementara itu, pada arsitektur **Many to One**, kita memberikan banyak input — misalnya sebuah kalimat atau sekuens data — lalu hanya menggunakan satu output di akhir, yaitu y_n , yang biasanya berasal dari *last hidden state*. Ini cocok untuk tugas seperti **sentiment analysis**, di mana keseluruhan kalimat dibaca, dan model menyimpulkan satu label seperti "positif" atau "negatif".

Kemudian ada **Many to Many** versi pertama, di mana jumlah input dan output **sama panjang**. Dalam skenario ini, setiap input memiliki output yang sejajar, seperti pada tugas **Named Entity Recognition**

(NER). Misalnya, kata "Budi" diikuti oleh label "B-PER", dan begitu seterusnya. Maka seluruh output y_1 hingga y_n digunakan dan masing-masing terhubung langsung dengan input-nya.

Terakhir, ada juga bentuk **Many to Many** dengan panjang input dan output yang berbeda, seperti pada **machine translation**. Di sini, seluruh kalimat input (misalnya dalam bahasa Inggris) dibaca terlebih dahulu oleh encoder, dan kemudian decoder menghasilkan output (misalnya dalam bahasa Indonesia). Output y_1 sampai y_m semua digunakan, tapi tidak lagi sejajar satu per satu dengan input-nya karena urutan dan panjang kalimat bisa berbeda.

Jadi, intinya, apakah kita menggunakan satu output, semua output, atau output yang panjangnya berbeda dari input, semuanya bergantung pada **tugas spesifik yang ingin kita selesaikan dengan RNN**.

Penjelasan yang Bisa Kamu Sampaikan:

Pernah lihat meme Spider-Man saling nunjuk begini?

Nah, ini sebenarnya *mirip banget* dengan ide dasar dari **Siamese Networks** — yaitu **dua (atau lebih) jaringan yang identik**, memiliki struktur dan bobot yang sama, dan digunakan untuk **membandingkan dua input yang berbeda**.

- ② Input A dan Input B masuk ke **encoder yang sama**.
- ② Encoder menghasilkan **dua vektor representasi** (misal: vec_A dan vec_B).
- ② Dua vektor ini dibandingkan → dihitung **jaraknya** (misalnya pakai cosine similarity atau Euclidean).
- ② Hasil perbandingan ini digunakan untuk menghitung **loss** (kerugian).
- ② Loss digunakan untuk **update bobot encoder** — satu encoder saja yang di-update, karena memang cuma satu encoder yang shared.

Cosine Similarity mengukur **kemiripan antara dua vektor** berdasarkan **sudut (angle)** di antara mereka, **bukan panjangnya**.

Nilainya	berkisar	antara	-1	sampai	1		
Semakin	dekat	ke	1	→	vektor	makin	mirip

Semakin dekat ke **-1** → vektor makin **berlawanan arah**

Cosine similarity digunakan untuk mengukur seberapa mirip dua vektor.

Dalam konteks ini, vektor yang dimaksud adalah **representasi dari dua pertanyaan**.

Jadi, **cosine similarity** memberitahu kita seberapa mirip dua pertanyaan tersebut.

Nilai hasil perhitungannya disebut **\hat{y} (dibaca: y-hat)**, yaitu **prediksi dari Siamese Network**.

Nilai \hat{y} ini berada dalam **rentang -1 hingga 1**:

- Jika \hat{y} mendekati **1**, berarti **dua pertanyaan sangat mirip**.
- Jika \hat{y} mendekati **-1**, berarti **dua pertanyaan sangat berbeda**.

💡 **Ambang Batas (Threshold):**

Kita akan menetapkan sebuah ambang batas (misalnya disebut **Z**):

- Jika $\hat{y} \leq Z$, kita anggap **dua pertanyaan tidak sama**.
- Jika $\hat{y} > Z$, kita anggap **dua pertanyaan sama**.

➡ Nilai Z ini bisa kamu sesuaikan tergantung **seberapa ketat kamu ingin mendeteksi kemiripan**.

Misalnya:

- **Z = 0.8** artinya hanya pertanyaan yang *sangat mirip* yang akan dianggap sama.
- **Z = 0.5** lebih longgar, sehingga pertanyaan yang cukup mirip juga dianggap sama.

Proyek ini bertujuan untuk membangun model **Siamese Network dengan arsitektur Bidirectional LSTM (BiLSTM)** untuk mendeteksi **kemiripan antara dua pertanyaan** dalam dataset Quora Question Pairs. Dataset ini berisi pasangan pertanyaan yang telah dilabeli apakah mereka termasuk **duplicat** (1) atau **bukan** (0). Langkah awal dalam proyek ini adalah memuat dataset berformat .tsv yang diunggah secara manual, lalu dilakukan eksplorasi data seperti melihat distribusi label dan memastikan tidak ada data kosong. Untuk menangani **ketidakseimbangan kelas (class imbalance)** — di mana label 0 jauh lebih dominan — dilakukan **undersampling** agar distribusi label lebih seimbang.

Setelah itu, dilakukan proses **tokenisasi dan vektorisasi teks** menggunakan VocabularyProcessor dari TensorFlow. Teks dikonversi menjadi urutan indeks kata, dengan panjang maksimum (max_len) yang ditentukan. Untuk representasi kata, digunakan **pre-trained word embeddings GloVe** berukuran 300 dimensi yang diunduh dari internet dan dimuat ke dalam **embedding matrix**. Matrix ini kemudian digunakan dalam embedding layer yang bersifat non-trainable, sehingga bobot dari GloVe tidak diubah selama pelatihan.

Model SiameseGloveLSTM dibangun menggunakan tf.keras.Model subclassing, di mana arsitekturnya terdiri dari **tiga lapis Bidirectional LSTM dengan jumlah unit berbeda di tiap layer** (misalnya 64, 32, dan 16 unit). Output dari ketiga layer ini digunakan untuk merepresentasikan masing-masing pertanyaan, dan dihitung **jarak Euclidean** antara keduanya sebagai ukuran kemiripan. Fungsi loss yang digunakan adalah **Contrastive Loss**, yang dirancang untuk membuat jarak antar pasangan duplikat sekecil mungkin dan jarak antar pasangan tidak duplikat sebesar mungkin (hingga batas margin).

Model kemudian dikompilasi dan dilatih selama beberapa epoch, dan hasil training seperti **nilai loss** dicatat untuk dianalisis lebih lanjut. Setelah pelatihan, model diuji dengan **input manual dari dua pertanyaan** untuk menghitung skor kemiripan (similarity score). Jika nilai skor di atas ambang batas (misalnya ≥ 0.5), maka pasangan dianggap duplikat; sebaliknya, jika skor di bawah ambang batas, maka pasangan dianggap berbeda. Pendekatan ini berguna untuk berbagai aplikasi seperti **chatbot QA retrieval, pencarian cerdas, dan deteksi konten ganda**.