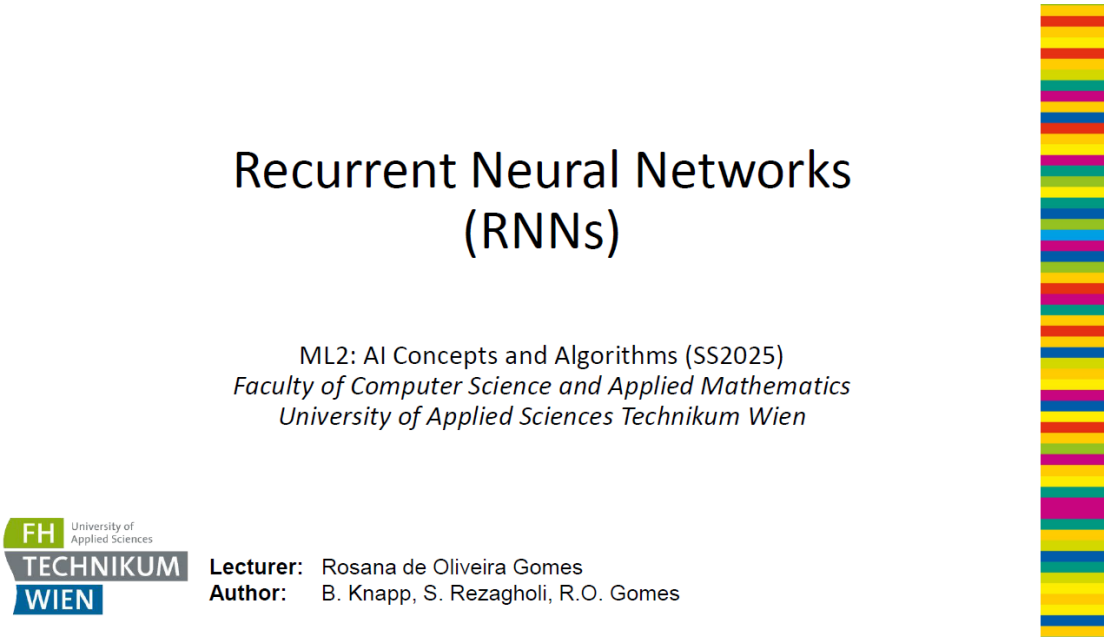
RNNs

June 19, 2025



[1]:
```
import os
!which pandoc
!echo $PATH
```

```
/opt/conda/envs/ml/bin:/opt/conda/bin:/usr/local/nvidia/bin:/usr/local/cuda/bin:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Presentation Script – Slide 1: Title

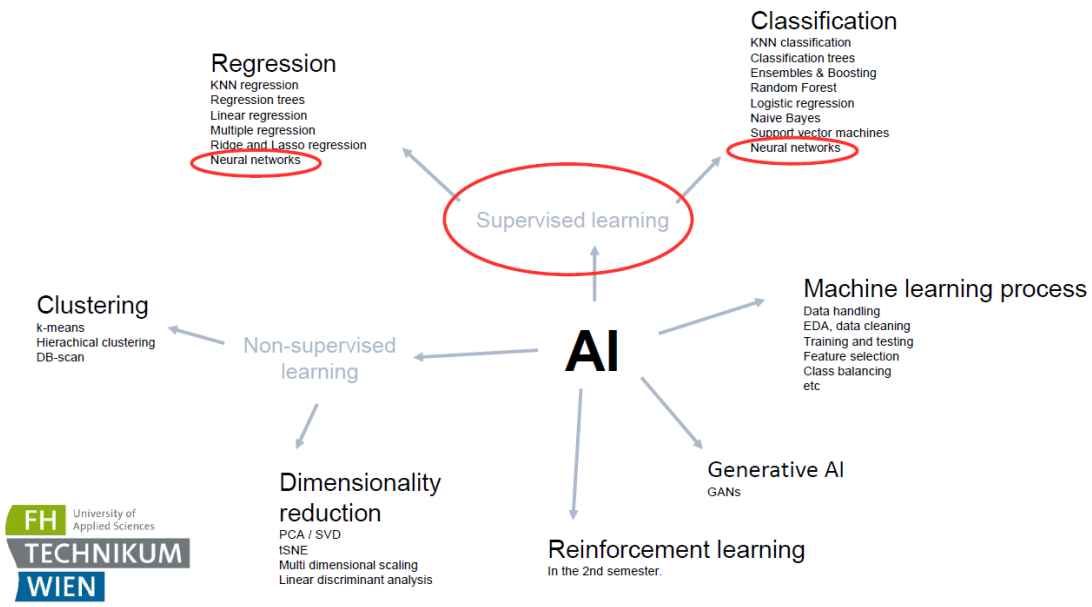
"Good [morning/afternoon], everyone.

Welcome to today's lecture on **Recurrent Neural Networks**, also known as **RNNs**. This session is part of the course **ML2: AI Concepts and Algorithms**, offered in the summer semester of 2025.

We're part of the **Faculty of Computer Science and Applied Mathematics** here at the **University of Applied Sciences Technikum Wien**.

I'm Rosana de Oliveira Gomes, and I'll be guiding you through this exciting topic today. The content for this lecture was co-authored by **Benedikt Knapp**, **Sina Rezagholi**, and myself.

Let's dive into the fascinating world of neural networks designed to handle **sequential data**."



Presentation Script – Slide 2: AI Landscape Overview

"This slide offers a **big picture** view of the various areas within **Artificial Intelligence**.

At the center is AI, and surrounding it are different branches like **Supervised Learning**, **Unsupervised Learning**, **Dimensionality Reduction**, **Generative AI**, and **Reinforcement Learning** — which we'll explore more in the second semester.

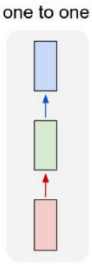
We're currently focused on **Supervised Learning**, which includes two major categories: **Classification** and **Regression**.

Notice that **Neural Networks** appear in both categories. That's because they are extremely flexible and can be used to solve both classification problems — such as spam detection or image labeling — and regression problems — like predicting stock prices or temperature.

Our topic today, **Recurrent Neural Networks**, builds on this powerful foundation, but is specifically tailored for handling **sequential data**, where the order and context of input really matter.

You'll also see that other key areas like clustering, dimensionality reduction, and generative models — including GANs — play vital roles in broader AI applications."

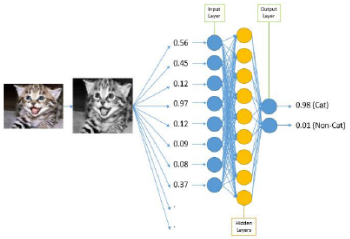
Sequential Problems



Typical ANN: input layer → hidden layer → output layer

ML Problems: Regression, Classification

One to one relationship between input and output



FH University of Applied Sciences TECHNIKUM WIEN

3

Presentation Script – Slide 3: Sequential Problems

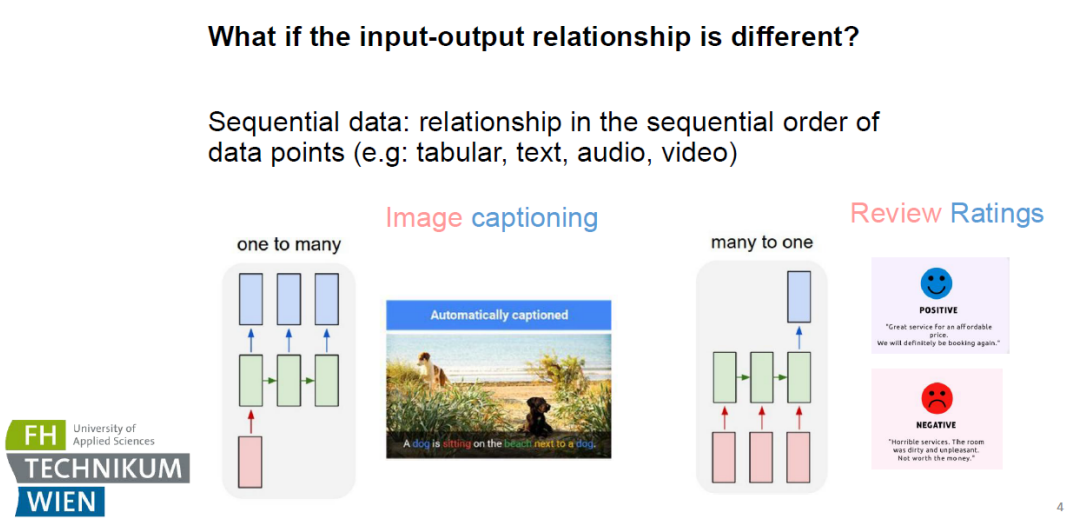
"Let's begin with what we already know — the typical structure of a neural network.

In a **classic artificial neural network** or ANN, we have a fixed sequence: **Input layer → Hidden layer → Output layer.** This is very well suited for **standard machine learning problems** like regression and classification.

And in these cases, we have a **one-to-one relationship**: One input goes in — one prediction comes out.

A good example is image classification. As we see in the diagram, the input is a single image — say, of a cat — and the output is a prediction: 'Cat' with high confidence, or 'Not Cat' with low confidence.

But… what happens if we have a sequence of inputs or need a sequence of outputs? That's where things become more interesting — and where RNNs come into play."



Presentation Script – Slide 4: Sequential Problems (More Complex Cases)

"Now here's the key question: **What if the relationship between input and output is not one-to-one?**

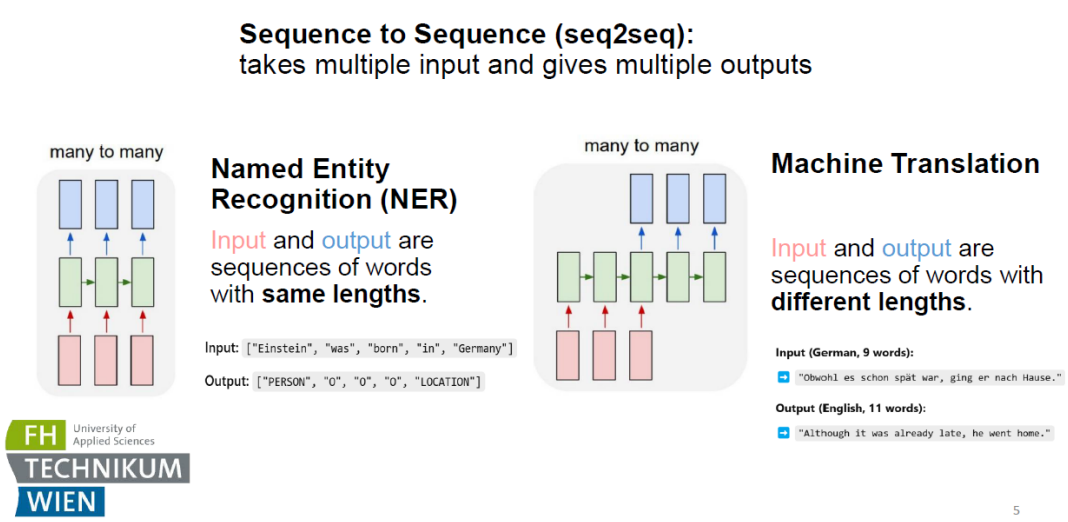
That's exactly what happens with **sequential data** — like **text, audio, video**, or even structured time series. In these cases, the order and context of the data matter. Each element in the sequence depends on what came before — and sometimes even on what comes after.

Let's look at two examples:

On the left, we have a **One-to-Many** case — for example, **image captioning**. One input image produces a **sequence of words** as output.

On the right, we see a **Many-to-One** case — like **review classification**. A sequence of words is analyzed to produce **one single output**, such as a positive or negative sentiment label.

These cases highlight why we need special architectures like **RNNs** — they allow us to model temporal or ordered dependencies in a flexible way."



Presentation Script – Slide 5: Sequence to Sequence (Seq2Seq)

"In the previous slides, we saw cases of one-to-many and many-to-one mappings. Now we take it a step further — to **Sequence-to-Sequence models**, or **Seq2Seq** for short.

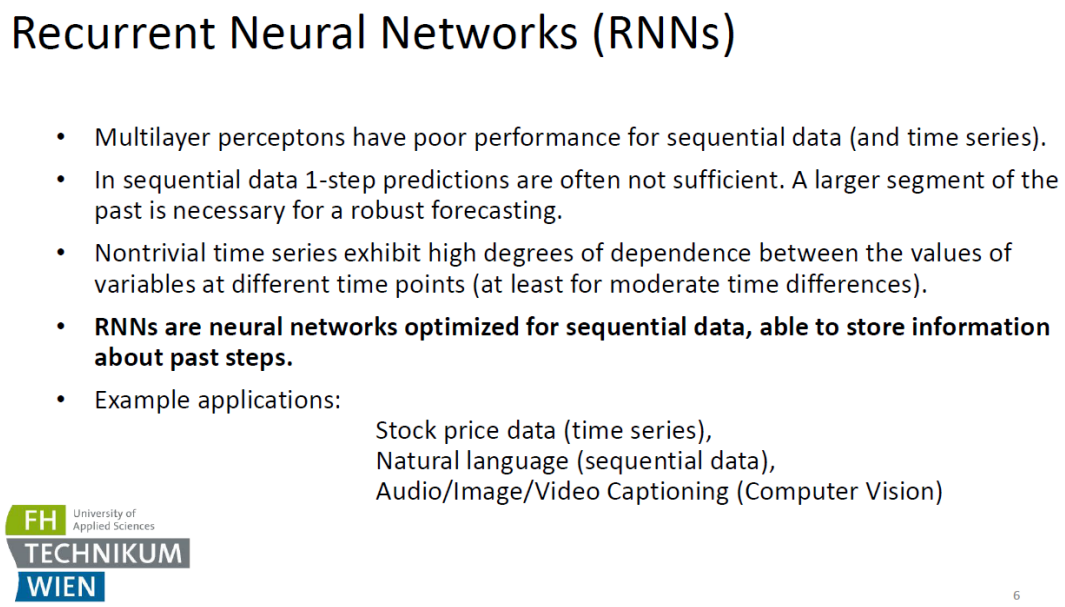
These models are designed to handle **many-to-many** relationships — that means both the input and output are sequences, not single values.

Let's look at two examples:

On the left, we have **Named Entity Recognition (NER)**. Here, both input and output sequences have the **same length**. We input a sentence like: 'Einstein was born in Germany', and the model outputs a sequence of **entity tags**, such as 'PERSON' or 'LOCATION'.

On the right is **Machine Translation**. This is another many-to-many task — but with a twist: The input and output **can have different lengths**. For instance, a German sentence might contain 9 words, and its English translation could have 11.

These kinds of tasks — with long-range dependencies and flexible sequence lengths — are exactly where **RNNs** and their variants like LSTMs shine."



Presentation Script – Slide 6: Recurrent Neural Networks (RNNs)

"So now that we've looked at different types of sequential problems, let's talk about how we can actually solve them — with **Recurrent Neural Networks**, or **RNNs**.

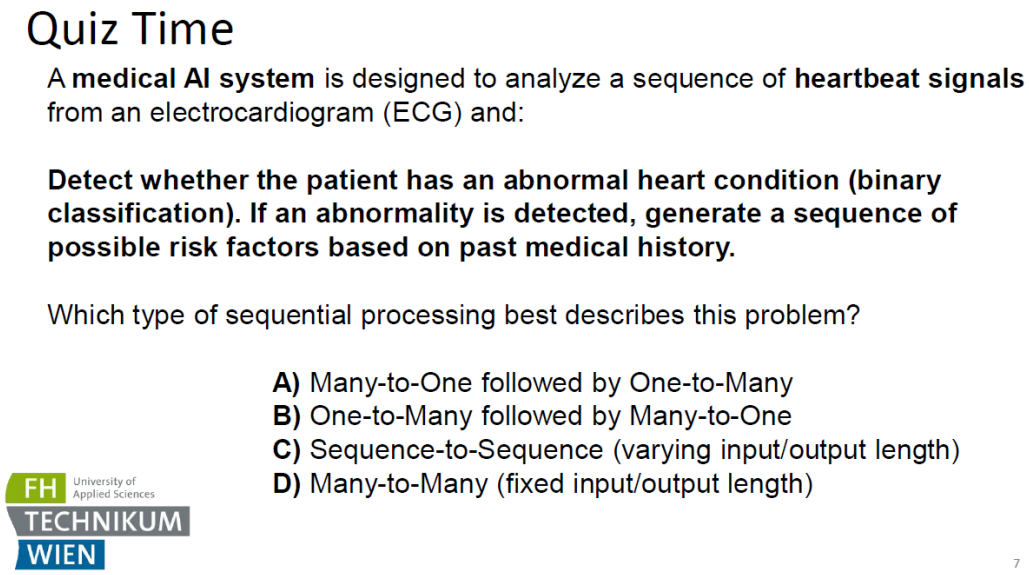
Here's the issue: Standard multilayer perceptrons — the classic feedforward networks — struggle with **sequential data**, especially in **time series forecasting** or **language modeling**. They can only make predictions based on the current input, without remembering what came before.

But in the real world, that's rarely enough. **One-step predictions don't cut it.** For robust decision-making, we often need a memory of what happened in the recent past — for example, in stock market analysis, where past values heavily influence future ones.

That's where RNNs come in. They're **specially designed to process sequences**, and can **retain information across time steps**.

Some real-world examples include: Stock price predictions, Natural language tasks like translation or chatbots, And even captioning for audio, images, or videos.

Think of RNNs as a kind of **neural memory system** — they don't just look at the current input, they also remember what just happened."



Presentation Script – Slide 7: Quiz Time!

"Alright — let's take a short break for a **quick quiz**!

Imagine this scenario:

We have a **medical AI system** analyzing a sequence of **heartbeat signals** from an ECG. Its job is two-fold:

1 First, it **detects whether the patient has an abnormal heart condition** — a classic binary classification problem. 2 Then — if an abnormality is found — it **generates a sequence of risk factors** based on past medical history.

So we have two tasks:

- A **many-to-one** decision (classification based on a sequence),
- Followed by a **one-to-many** output (producing a sequence of possible causes).

So… which type of sequential processing best describes this scenario?

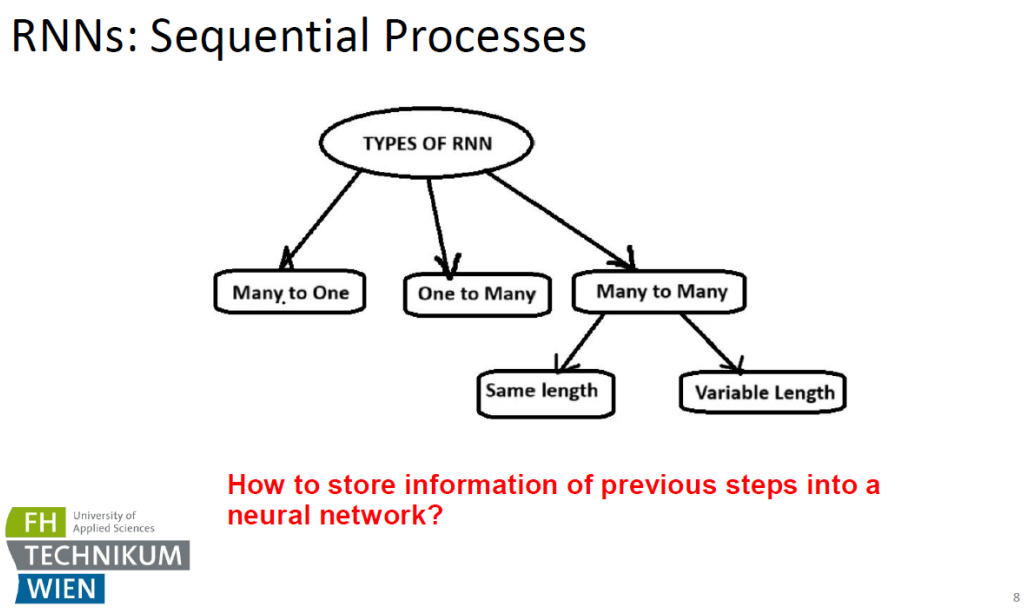
Let me read out the options:

A) Many-to-One followed by One-to-Many
B) One-to-Many followed by Many-to-One
C) Sequence-to-Sequence (varying input/output length)
D) Many-to-Many (fixed input/output length)

[Pause, engage audience]

The correct answer is **A**: We start with a **Many-to-One** — the heartbeat signals are mapped to a single classification. Then, if needed, we continue with **One-to-Many** — generating a sequence of risk factors.

This type of logic is very typical in real-world AI systems — especially in healthcare, where the **sequence of data matters deeply**."



Presentation Script – Slide 8: RNNs – Sequential Processes

"So far, we've seen that RNNs can handle different **types of input-output sequences** — from many-to-one, to one-to-many, and many-to-many — with either fixed or variable sequence lengths.

These correspond to real-world applications like sentiment analysis, text generation, or translation.

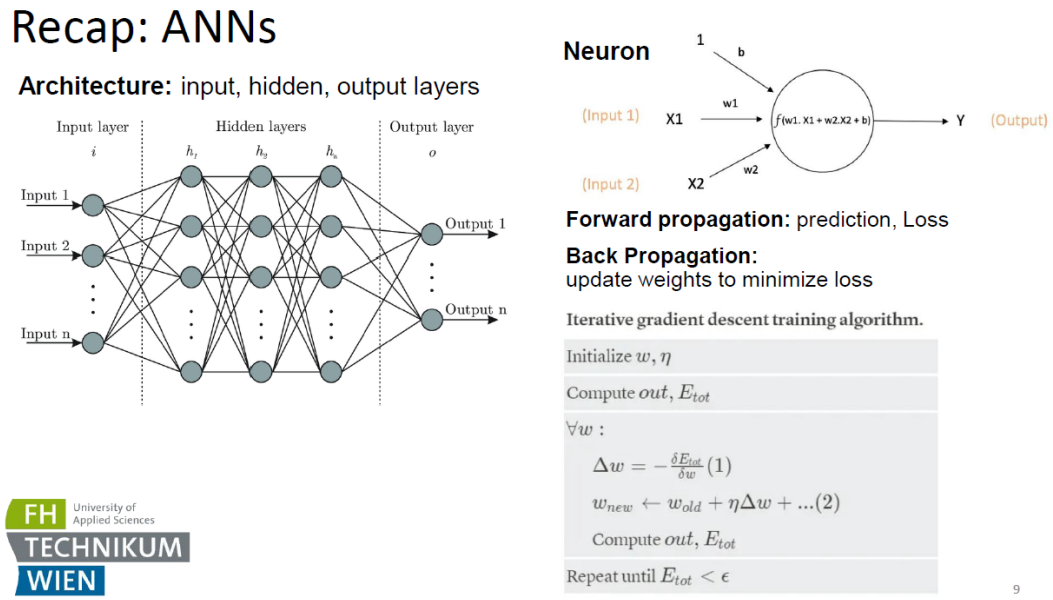
But now comes the big question — highlighted in red at the bottom:

How do we store information from previous steps in a neural network?

In other words, how can a model 'remember' what just happened?

This is where the **recurrent architecture** of RNNs comes in. Unlike feedforward networks, RNNs pass information forward in **time**, using a hidden state that gets updated step-by-step.

That's exactly what we'll look at next — how RNNs use recurrence to **carry memory across time steps**."



Presentation Script – Slide 9: Recap – Artificial Neural Networks (ANNs)

"Before we dive deeper into RNNs, let's quickly recap how a typical **artificial neural network** works.

We have a basic architecture: An **input layer**, one or more **hidden layers**, and finally an **output layer**.

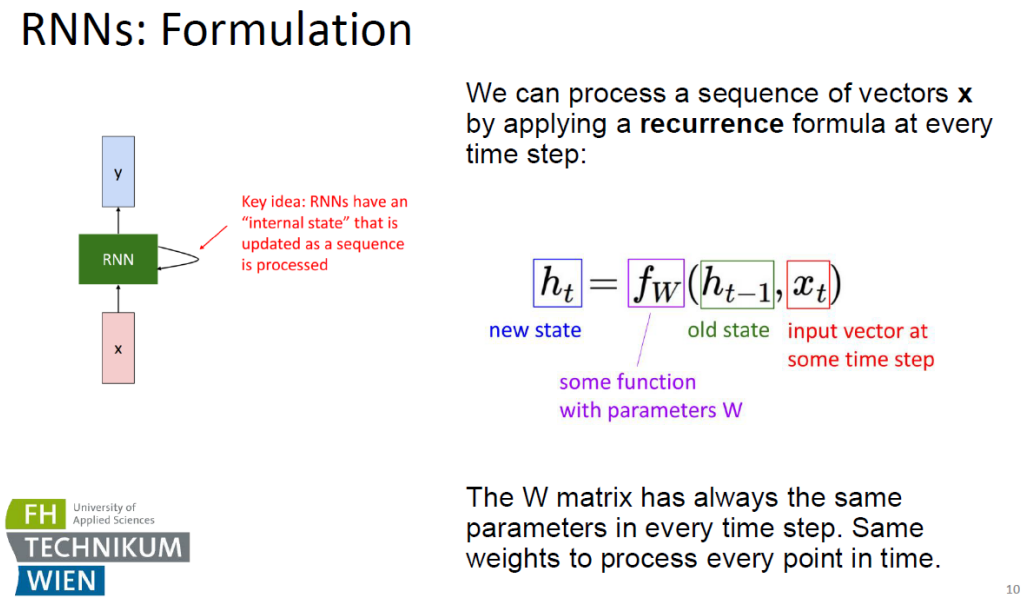
Each node in a layer is called a **neuron**, and every connection carries a **weight**.

During **forward propagation**, inputs are processed through the layers to generate predictions — and those predictions are compared against actual values to calculate the **loss**.

Then comes **backpropagation**, where we compute the gradients of the loss function and update the weights using **gradient descent** to minimize that loss over time.

This is the core training loop we use in feedforward networks.

But here's the catch: In ANNs, all inputs are processed independently — they have **no memory** of what happened before. That's why we now need a new kind of architecture… the **Recurrent Neural Network**."



Presentation Script – Slide 10: RNNs – Formulation

"So now, let's get to the **core idea** behind Recurrent Neural Networks.

Unlike traditional networks, an RNN processes inputs **sequentially**, and — most importantly — it **maintains an internal memory** of past inputs by reusing its **hidden state**.

You can see this in the diagram on the left: The RNN block loops back into itself — this loop represents the **recurrent connection**. This is how the network keeps track of past information as it processes a sequence.

Mathematically, this is expressed using a **recurrence formula**:

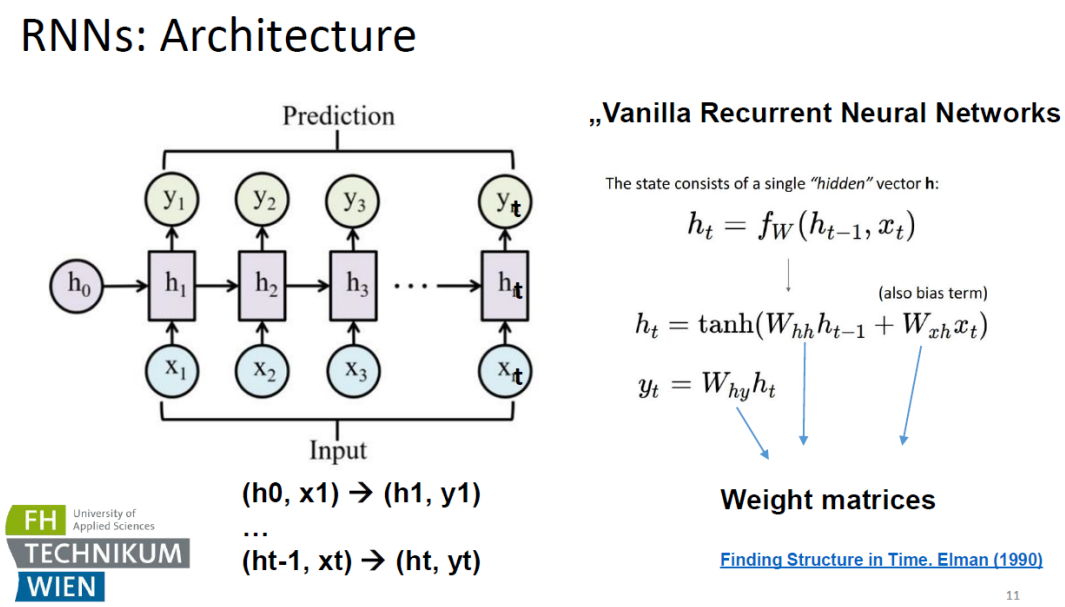
$$h_t = f_W(h_{t-1}, x_t)$$

Let me break that down:

- x_t is the input at time step t,
- h_{t-1} is the **hidden state** from the previous step — the memory,
- f_W is a function with **parameters W**, which are **shared across all time steps**,
- And h_t is the updated hidden state — our **new memory**.

That means we're using **the same weight matrix** W at each step to process a new input and update the internal state.

This weight sharing gives RNNs the ability to **generalize across time**, and makes them very efficient at learning temporal patterns."



Presentation Script – Slide 11: RNNs – Architecture

"Now that we've seen the recurrence formula, let's visualize what's happening inside an **RNN architecture**.

What you see here is a typical **Vanilla RNN**, unrolled through time.

At each time step t, we feed in an input x_t along with the hidden state from the previous step h_{t-1}. The RNN processes this and updates its hidden state to h_t, which is also used to compute the output y_t.

This happens **repeatedly** for each time step — and the same weight matrices are reused across all steps.

The diagram on the right shows this mathematically:

- First, the hidden state is computed as:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

- Then the output is:

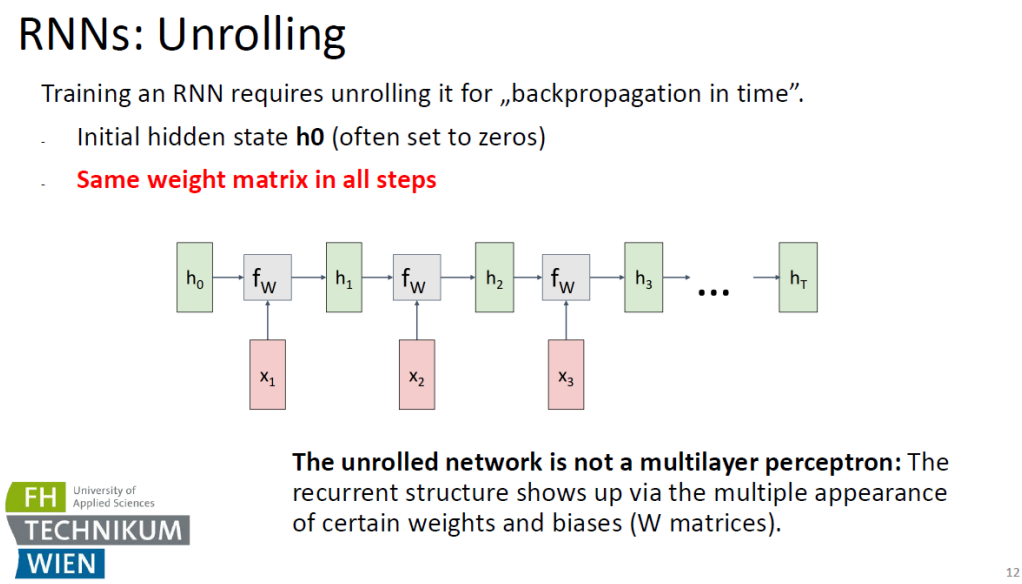
$$y_t = W_{hy}h_t$$

The three weight matrices are:

- W_{xh}: input-to-hidden
- W_{hh}: hidden-to-hidden (recurrent)
- W_{hy}: hidden-to-output

This model was formalized by Jeffrey Elman in 1990 in the foundational paper: **"Finding Structure in Time"** — and it's still the base for many modern sequence models.

In essence, RNNs are all about **propagating hidden state forward**, allowing the network to learn from previous steps."



Presentation Script – Slide 12: RNNs – Unrolling

"To train an RNN, we need to **unroll it in time** — this is known as **Backpropagation Through Time**, or **BPTT**.

Let me explain.

RNNs are recurrent — but to apply gradient descent, we need to visualize what happens across time steps. That's why we **unroll** the network — laying it out step by step.

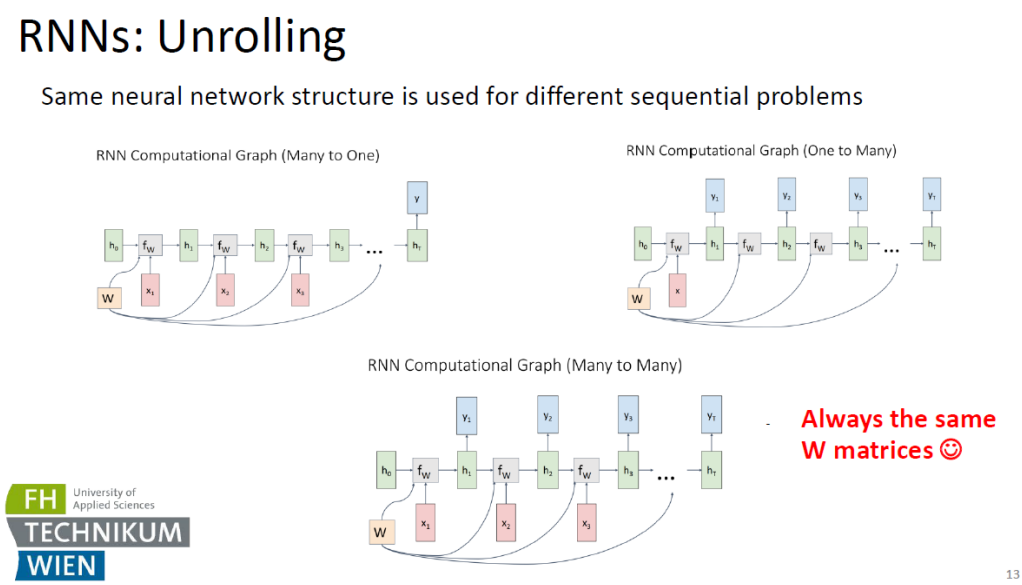
We start with an **initial hidden state** h_0, often initialized to zero.

At each step t, we feed in a new input x_t, and apply the same function f_W, which updates the hidden state to h_t.

And here's a key detail, shown in red: **The weight matrix W is the same at every time step.** We're not learning different parameters for each moment — instead, we learn a single set of weights that generalizes across the whole sequence.

At the bottom, notice this important clarification: Even though the unrolled RNN looks like a deep network, it's **not a traditional multilayer perceptron.** The repeated structure is due to **time**, not **depth**.

This recursive reuse of weights is what allows RNNs to **model dependencies across time** efficiently."



Presentation Script – Slide 13: RNNs – Unrolling for Different Problems

"This slide shows how the **same RNN structure** can be adapted to handle **different types of sequential problems**.

Let's walk through the three diagrams:

Top left: This is the **Many-to-One** case — for example, sentiment analysis. We process an input sequence x_1 to x_T, and at the very end, we produce a **single output** y, using the final hidden state h_T.

Top right: This shows the **One-to-Many** case — like image captioning. We start with a single input x, and generate a **sequence of outputs** $y_1, y_2, \ldots, y_T$.

Bottom diagram: This is the **Many-to-Many** setup — for example, machine translation. We process a full sequence of inputs, and produce a corresponding output at **each time step**.

The key takeaway here is marked in red:

Always the same W matrices!

This reuse of weights across time steps is what makes RNNs **efficient** and **general-purpose**. It means we're not learning a different network for every time step — instead, the same logic is applied recursively as we move through time."

Backpropagation through time

Forward run outputs results:

$$h_t = \sigma_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$\hat{y}_t = \sigma_o(W_{oh}h_t + b_o)$$

Total Loss function: $L = \sum_{t=1}^{T} L_t(\hat{y}_t, y_t)$

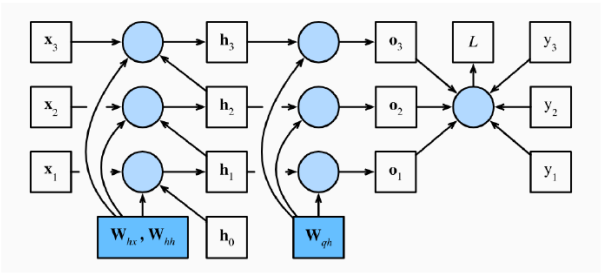
Backpropagation Process:

Update weights to minimize the loss

$$W_{xh} \leftarrow W_{xh} - \eta \cdot \frac{\partial L}{\partial W_{xh}}$$

$$W_{hh} \leftarrow W_{hh} - \eta \cdot \frac{\partial L}{\partial W_{hh}}$$

$$W_{oh} \leftarrow W_{oh} - \eta \cdot \frac{\partial L}{\partial W_{oh}}$$



https://d2l.ai/chapter_recurrent-neural-networks/bptt.html

Backpropagation through time (BPTT):

Unfold all the way to the initial step

(see last slide for an example)

$$h_t = f_W(h_{t-1}, x_t)$$

14

Presentation Script – Slide 14: Backpropagation Through Time (BPTT)

"Now let's talk about **how RNNs learn** — and that involves a method called:

Backpropagation Through Time, or **BPTT**.

Just like with regular neural networks, RNNs are trained using **gradient descent**, but since RNNs deal with sequences, we need to apply **backpropagation across time steps**.

On the **left**, you can see how forward propagation works:

- At each step, we compute the hidden state h_t using inputs and the previous hidden state.
- The prediction $\hat{y}_t$ is made from h_t.

Then we compute the **total loss** across the full sequence:

$$L = \sum_{t=1}^{T} L_t(\hat{y}_t, y_t)$$

In the **backpropagation phase**, we compute the gradients with respect to each weight:

- Input-to-hidden weights W_{xh}
- Hidden-to-hidden weights W_{hh}
- Hidden-to-output weights W_{oh}

We apply the usual update rule:

$$W \leftarrow W - \eta \cdot \frac{\partial L}{\partial W}$$

Where η is the learning rate.

But here's the twist: In BPTT, we **unfold** the RNN through time — and backpropagate the loss all the way to the **initial time step**.

That's why RNN training can be slow and sometimes unstable — because we're computing gradients **through a deep chain of time steps**.

The diagram on the right gives a great overview of this process — and there's also a helpful resource linked below from d2l.ai if you want to study this in more depth."

RNNs: Pseudo-Code

Repeat till the stopping criterion is met:

1. Set all h to zero.
2. Repeat for t = 0 to n-k
    1. Forward propagate the network over the unfolded network for k time steps to compute all h and y
    2. Compute the error as: $e = y_{t+k} - p_{t+k}$
    3. Backpropagate the error across the unfolded network and update the weights

FH University of Applied Sciences TECHNIKUM WIEN

15

Presentation Script – Slide 15: RNNs – Pseudo-Code

"This slide breaks down the entire RNN training process into a simple **pseudo-code loop**.

Let's go through it step by step.

First: We repeat the training loop **until a stopping criterion is met** — usually based on convergence or number of epochs.

1 **Step 1:** We start by **initializing all hidden states** h **to zero**. This gives the network a clean memory at the start of each sequence.

2 **Step 2:** We iterate over time — from $t = 0$ up to $n - k$, where k is the length of the sequence window.

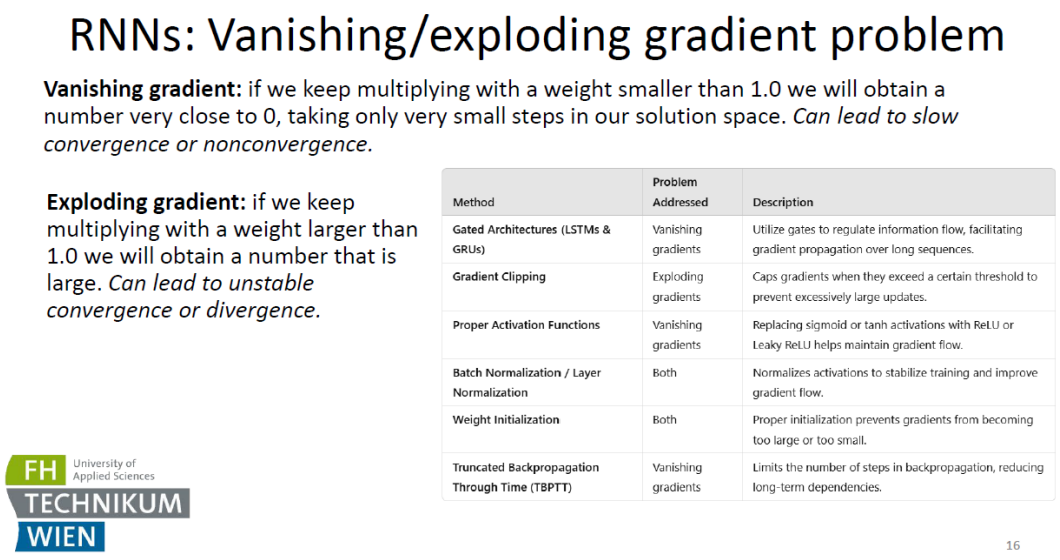
2.1. For each t, we **forward propagate** the network through the next k time steps, computing all the hidden states h and predicted outputs y.

2.2. Then we compute the **error** using the difference between the actual output y_{t+k} and the predicted value p_{t+k}.

2.3. Finally, we apply **Backpropagation Through Time (BPTT)** across those k steps to update the weights.

In practice, this structure is used in **sliding window training**, and k is chosen depending on how far back the RNN needs to look in the sequence.

This pseudo-code captures the **essence of RNN learning** — memory, temporal propagation, and weight adjustment."



Presentation Script – Slide 16: RNNs – Vanishing and Exploding Gradient Problem

"Now let's talk about a **major challenge** in training RNNs: the **vanishing and exploding gradient problem**.

Vanishing gradients occur when we multiply small numbers repeatedly — for example, weights less than 1.0 — across time steps. Eventually, the values get so small that updates become almost zero. This causes the model to **stop learning**, especially for long-term dependencies.

Exploding gradients happen when the weights are greater than 1.0 — and repeated multiplication causes the gradients to blow up, which can make the training **unstable or even diverge** completely.

To the right, we have a table with strategies to mitigate these problems:

Gated architectures like LSTMs and GRUs are designed to address vanishing gradients by using special gates that **control what to forget and what to retain**.

Gradient clipping prevents exploding gradients by **capping** their values during training.

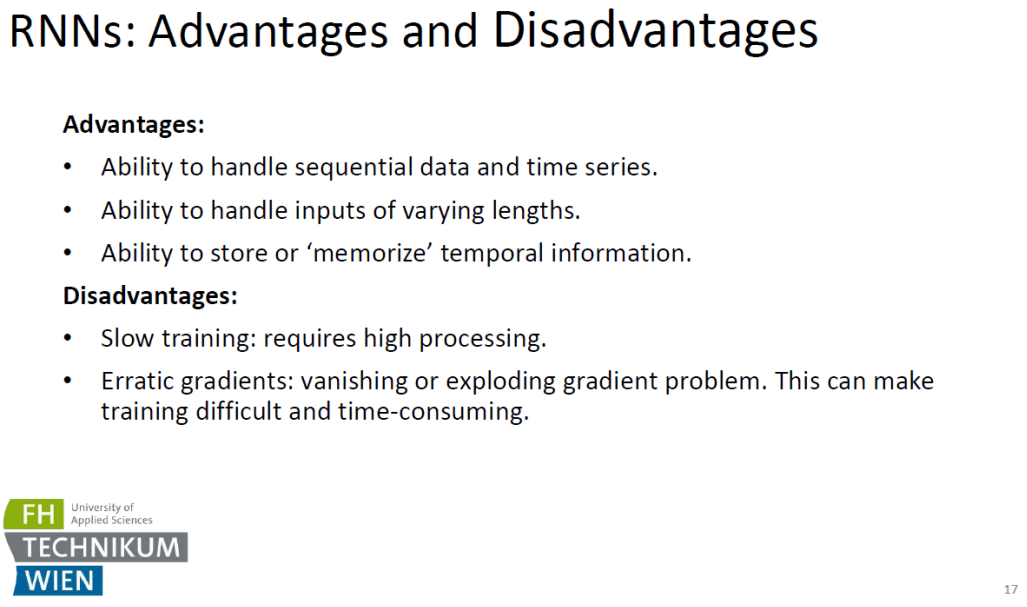
Proper activation functions like ReLU or Leaky ReLU help avoid saturation — which is a major cause of vanishing gradients in sigmoid or tanh.

Normalization techniques like batch or layer norm stabilize the gradient flow.

Weight initialization strategies are also key — too large or too small can break training.

Finally, **Truncated Backpropagation Through Time (TBPTT)** limits the number of time steps in BPTT to prevent gradients from becoming too weak over long sequences.

These issues were one of the driving forces behind the development of LSTMs, GRUs, and — later — Transformers."



Presentation Script – Slide 17: RNNs – Advantages and Disadvantages

"Before we move into advanced variants of RNNs, let's take a moment to **evaluate their strengths and weaknesses**.

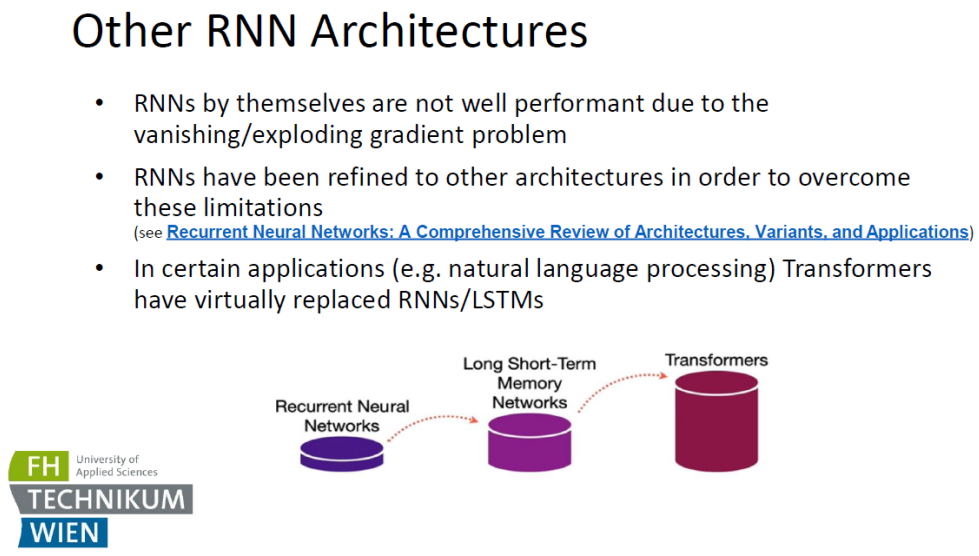
Advantages:

- RNNs are designed to **handle sequential data**, making them ideal for time series, natural language, and audio/video applications.
- They can **accept input sequences of varying lengths** — very useful in real-world data like sentences or sensor readings.
- Most importantly, they have the ability to **memorize temporal dependencies**, capturing patterns across time — which feedforward networks simply can't do.

Disadvantages:

- Training RNNs can be **computationally expensive** and **slow**, especially for long sequences or large datasets.
- They're also **notoriously sensitive to gradient issues** — either vanishing (where learning stalls) or exploding (where training becomes unstable).

These challenges motivated the development of better architectures — like **LSTMs**, **GRUs**, and eventually **Transformers**, which we'll explore next."



Presentation Script – Slide 18: Other RNN Architectures

"So far, we've seen the strengths of RNNs — and also their limitations, especially around the **vanishing and exploding gradient** problem.

Because of these limitations, RNNs by themselves are **not always ideal** — especially for **long sequences** where long-term memory is critical.

To address this, researchers have developed more advanced architectures, like:

LSTMs – Long Short-Term Memory networks, **GRUs** – Gated Recurrent Units, And most recently, **Transformers** — which are now dominating NLP.

Each of these refinements improves on basic RNNs by offering **better control over memory and gradients**.

There's a great reference listed here for further reading: *"Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications."*

And finally: In domains like **natural language processing**, **Transformers** have now virtually **replaced** RNNs and LSTMs due to their **parallelization, scalability, and long-range memory**.

You can see that visually at the bottom — the evolution from RNN → LSTM → Transformer.

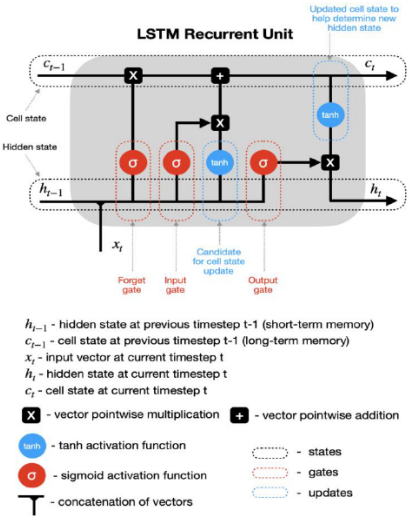
Next, we'll explore some of these models more closely."

Other RNN Architectures

Long Short Term Memory (LSTM): networks designed to address the vanishing gradient problem in RNNs. LSTMs use three gates (called **input, output, and forget gate**) to pass information between long term and short term memory cells.
See this video for a detailed explanation.

Gated Recurrent Units (GRU): also designed to handle the vanishing gradient problem through gates. They use **reset and update gates** to determine which information is to be retained for future predictions, similarly to LSTMs.

FH University of Applied Sciences TECHNIKUM WIEN



Presentation Script – Slide 19: Other RNN Architectures – LSTM & GRU

"Let's now look more closely at two powerful alternatives to the basic RNN: **LSTMs** and **GRUs**.

Long Short-Term Memory (LSTM) networks were developed specifically to solve the **vanishing gradient** problem. The key innovation is the use of **three gates**:

- An **input gate** to decide what new information to store,
- A **forget gate** to determine what to discard,
- And an **output gate** to control what is passed to the next step.

These gates regulate the flow of information through **two states**:

- The **hidden state** h (short-term memory), and
- The **cell state** c (long-term memory).

You can see a full LSTM unit on the right — with all its gates and flow logic clearly illustrated.

For more detail, you can check out the linked video in the slide.

Gated Recurrent Units (GRUs) simplify this even further. They combine the forget and input gates into a **single update gate**, and also include a **reset gate**.

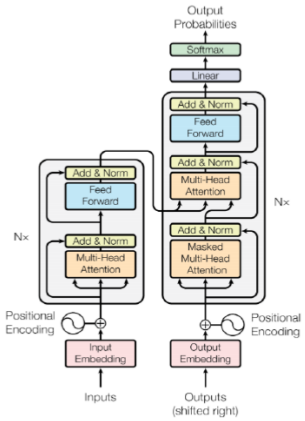
GRUs are faster and more efficient to train than LSTMs, but still handle long-term dependencies very well.

Both LSTMs and GRUs remain popular in practice — particularly for **language modeling**, **speech recognition**, and **time series prediction**."

Other RNN Architectures

Bidirectional Recurrent Neural Networks (BRNN): inputs from future time steps are used to improve the accuracy of the network. It is like knowing the first and last words of a sentence to predict the middle words.

Transformers: use self attention mechanisms to provide context. Determining the significance of each part of the input sequence relative to others allows transformer to capture long term dependencies. See paper walkthrough in this video.



FH University of Applied Sciences TECHNIKUM WIEN

Presentation Script – Slide 20: Other RNN Architectures – BRNN & Transformers

"Now let's wrap up this section by looking at two more powerful neural architectures:

Bidirectional RNNs (BRNNs)

Traditional RNNs only look **forward** — from past to present.

But what if we could look at both **past and future** when making a prediction?

That's what **Bidirectional RNNs** do. They process the sequence in both directions — which means the model can **use future context** to make a better prediction at each time step.

A good analogy: It's like knowing the **first and last words** of a sentence before trying to guess the **middle**.

Transformers

Transformers are currently the **state-of-the-art** in sequential modeling — especially in **natural language processing**.

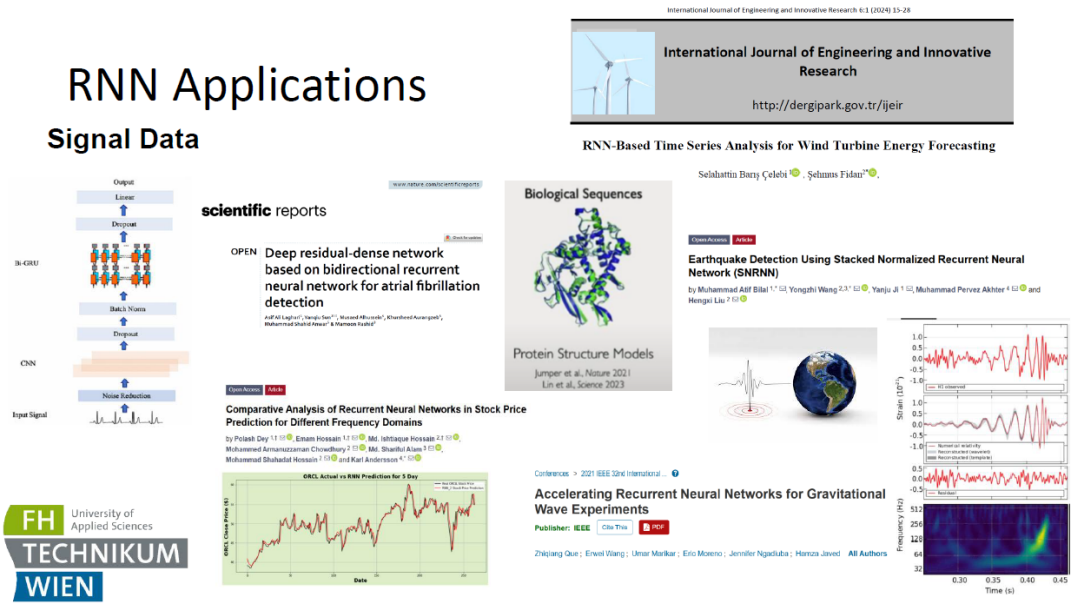
Instead of relying on recurrence or memory cells, Transformers use **self-attention mechanisms** to determine the **importance of each token** in the input relative to all others.

This allows them to **capture long-range dependencies efficiently** and to **process all input positions in parallel**, which is a massive speed advantage.

On the right, you can see the classic Transformer architecture from the original paper — with **multi-head attention**, **feedforward layers**, **residual connections**, and **positional encoding**.

There's also a link to a video walkthrough if you want to dive deeper into how Transformers work.

So to summarize: **BRNNs** give RNNs lookahead ability. **Transformers** reimagine sequence modeling entirely — without recurrence — and are now the industry standard for large-scale NLP models."



Presentation Script – Slide 21: RNN Applications – Signal Data

"To wrap up, let's take a look at **how RNNs are used in real-world research and engineering** — particularly for **signal data**.

RNNs are widely applied to **time series and signal processing tasks**, where understanding sequential patterns is critical.

Here are some fascinating examples featured on this slide:

Atrial fibrillation detection using bidirectional recurrent networks — These models can learn to recognize subtle irregularities in heart signals.

Earthquake detection — Here, RNNs are trained on seismic time series data to distinguish between noise and actual tremors.

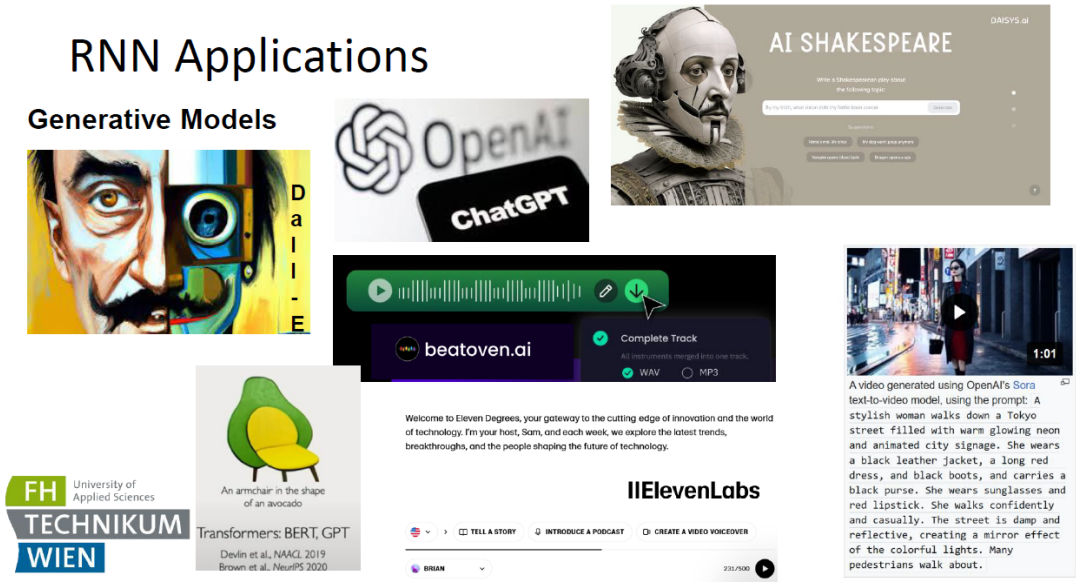
Stock price prediction — Financial models leverage RNNs to predict market trends across different frequency bands.

Gravitational wave experiments — Yes, even in astrophysics — RNNs are being used to detect gravitational signals buried in noisy time-series data.

Protein structure modeling, **wind turbine energy forecasting**, and many more.

As you can see, RNNs are incredibly powerful for working with **structured sequences over time**, making them indispensable in domains like healthcare, geophysics, finance, and physics.

This slide makes it clear: RNNs are not just theory — they're **actively transforming science and industry.**"



Presentation Script – Slide 22: RNN Applications – Generative Models

"We've seen how RNNs can be used in **signal processing**, but they also play a key role in one of the most exciting areas of AI today: **Generative Models**.

At the top left, you see a nod to **DALL · E**, a model that can generate stunning artwork from text prompts. While it's built on transformers, the early foundations of generative models — including text generation — were pioneered by **RNNs** and **LSTMs**.

ChatGPT — and other large language models — also evolved from earlier RNN-based language models, before transformers took over with architectures like **GPT** and **BERT**.

We also see **AI Shakespeare** — tools that can generate entire plays or poems, creatively mimicking human style.

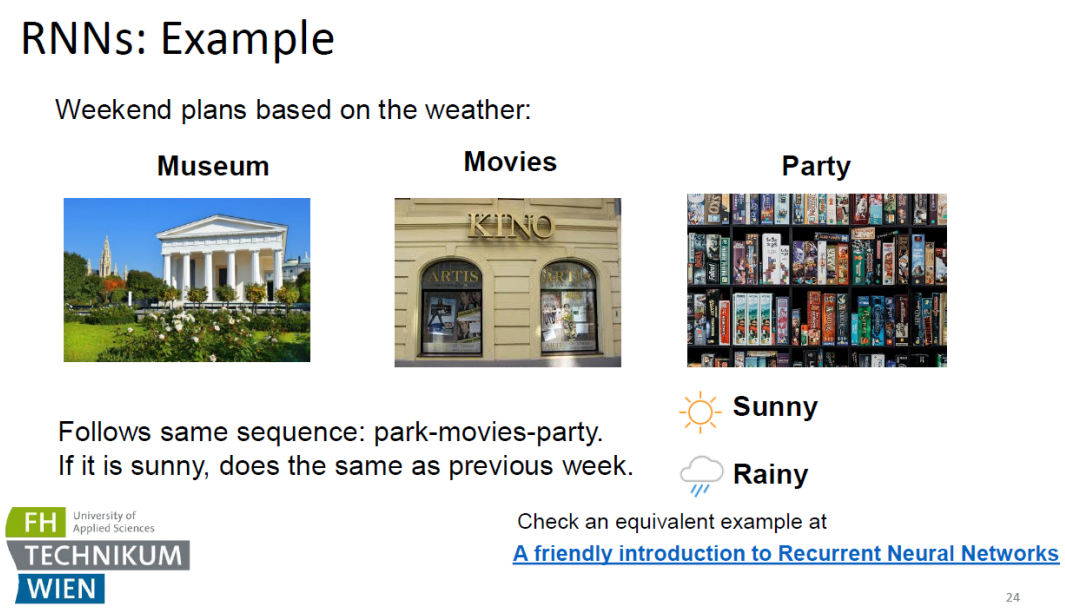
Beatoven.ai uses AI to compose original music, adapting to mood, genre, or scene.

ElevenLabs allows for realistic voice synthesis — often using RNN-based components for prosody and timing, now enhanced by deep transformer models.

And finally — **text-to-video generation**, like **OpenAI's Sora**, is pushing generative AI even further — creating videos from just a description, with storytelling coherence that used to be impossible.

So while transformers now dominate the scene, remember: **RNNs laid the groundwork** for these advances — from **sequence prediction** to **creativity in machines**.

This concludes our walkthrough of **Recurrent Neural Networks** — from fundamentals to cutting-edge applications. Thank you!"



Presentation Script – Slide 24: RNNs – Intuitive Example

"To make things a bit more tangible, here's a **light-hearted example** of how an RNN might work — based on something we all understand: **weekend plans**.

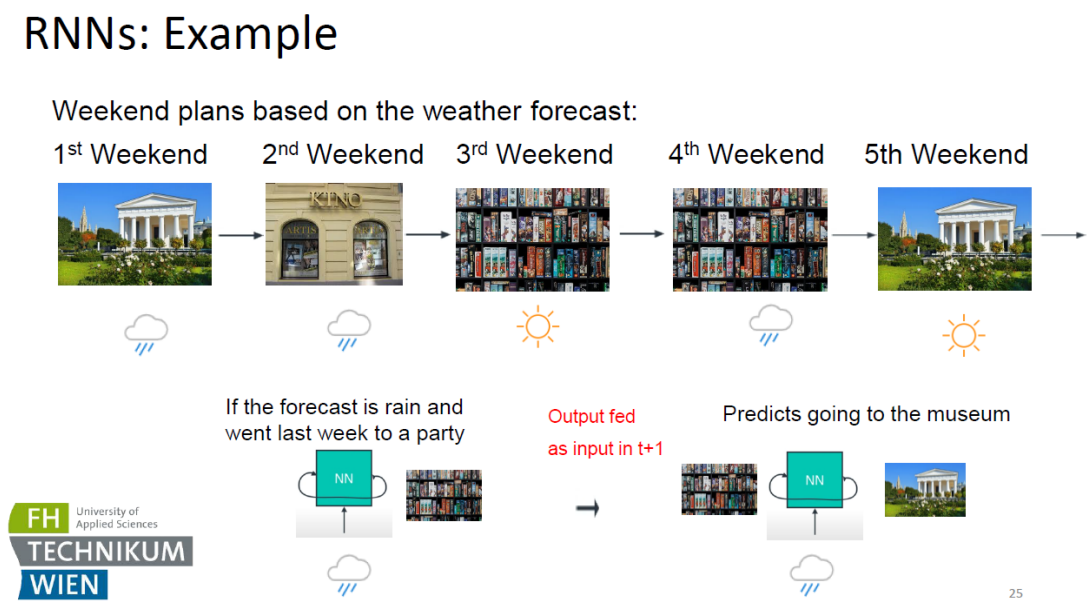
Imagine you have a typical pattern: you go from the **park**, to the **movies**, to a **party** — and you repeat this every week.

But there's a twist: your behavior depends on the **weather** — if it's **sunny**, you repeat what you did the previous week. If it's **rainy**, you change your plan.

This is exactly the kind of logic RNNs try to learn: Sequential patterns over time, How external factors influence those patterns, And how to **remember and update** what happened in the past.

This playful but insightful analogy captures the essence of how RNNs **store state and make predictions** based on past events and current input.

If you'd like to explore this example further, there's a great walkthrough linked at the bottom of the slide."



Presentation Script – Slide 25: RNNs – Example (continued)

"Let's now extend our weekend example — this time with a **forecast-driven decision process**.

We see a sequence of weekend activities — starting with the **museum**, followed by **movies**, then a **party** — with the decisions influenced by the **weather** forecast.

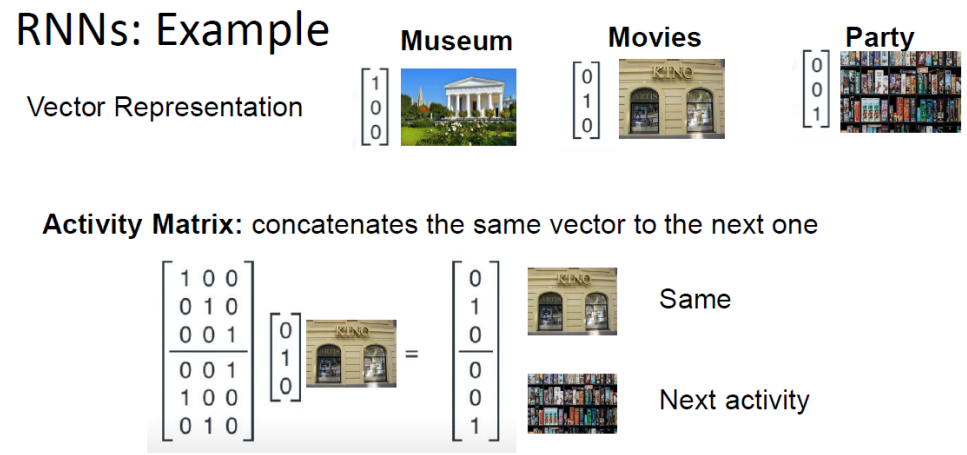
Here's what's happening:

- On the **first and second weekends**, it rains, so the pattern progresses.
- By the **third weekend**, it's sunny — and the RNN repeats the **previous week's activity** (party).
- On the **fourth weekend**, it rains again — but the model remembers: *"Last time it rained after a party, the next action was the museum."*
- So it **predicts going to the museum** on the **fifth weekend**.

This demonstrates **temporal dependency**: RNNs remember what happened in previous time steps and use that memory, along with current inputs, to predict future outcomes.

As you can see in the bottom flow: The **output from one step is used as input for the next** — this is the very nature of sequence modeling in RNNs.

This is a **fun but powerful metaphor** for how RNNs can model behavior and predict sequences — not based on just one step, but based on **patterns across time.**"



Presentation Script – Slide 26: RNNs – Vector Representation Example

"Let's now translate our weekend activity example into the language RNNs actually 'understand' — **vectors and matrices.**

At the top, we assign a **one-hot vector** to each activity:

- **Museum** → [1 0 0]
- **Movies** → [0 1 0]
- **Party** → [0 0 1]

This is how we numerically encode categorical inputs before feeding them into an RNN.

The bottom part shows what happens inside the model — we create an **Activity Matrix**, where these vectors are **concatenated** step-by-step to form a temporal input sequence.

For example:

- The matrix on the left stacks previous activity vectors,
- Then it gets **concatenated** with the current input (middle column)
- To form a **temporal feature vector** that captures both history and present.

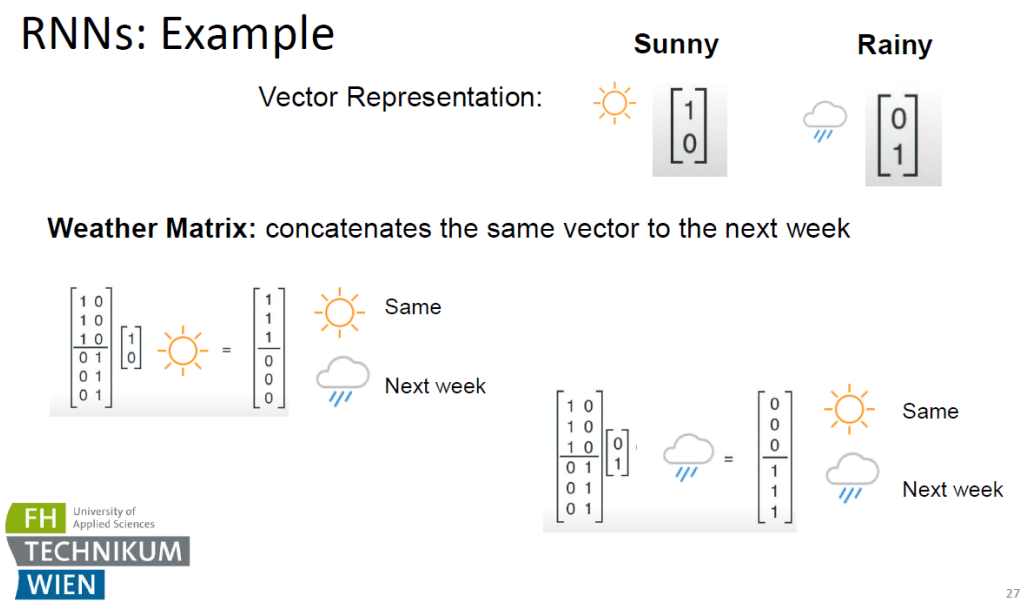
The goal is to allow the network to **learn transitions** — so that, given a sequence of past vectors, it can **predict the next one**.

As shown here:

- The same input vector leads to a **repeated prediction**,

- Or, with additional context, it might lead to the **next expected activity**.

This nicely illustrates how RNNs use **vector operations** to learn and replicate **patterns in sequences.**"



Presentation Script – Slide 27: RNNs – Weather Vector Representation

"Continuing our RNN example, let's now look at how **external factors** — like the weather — can also be **encoded as vectors** and combined with the sequence.

At the top, we see a **one-hot encoding** for weather:

- **Sunny** → [1, 0]
- **Rainy** → [0, 1]

We then build a **Weather Matrix**, which is used to represent a week-by-week pattern of weather.

As shown at the bottom:

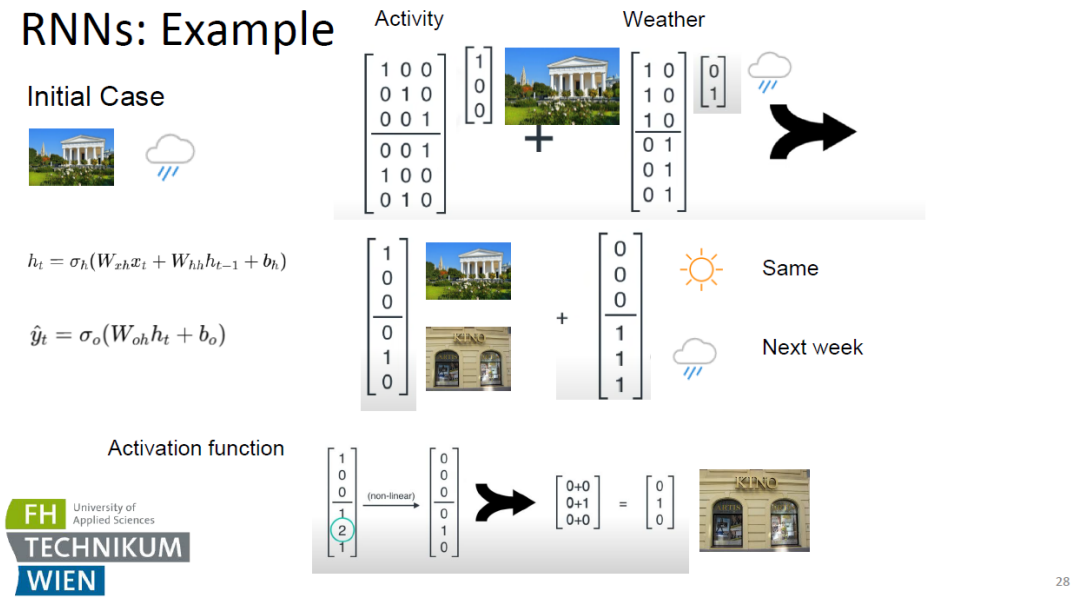
- The weather vector is **concatenated** to the existing matrix for the previous activities, forming a new input matrix that captures both **past actions** and **current conditions**.

This is a core idea in deep learning: We can **combine multiple inputs** — for example, user history + weather + day of the week — to make context-aware predictions.

In our playful weekend-planning example:

- When it's sunny, the system tends to **repeat the same activity**.
- When it's rainy, it may **predict a different activity** for the following week — depending on the prior sequence.

This is how RNNs learn to model **temporal patterns with contextual inputs**."



Presentation Script – Slide 28: RNNs – Final Example: Full Forward Pass

"Let's now look at the **complete RNN prediction process** using the weekend planner scenario.

At the top, we have the **initial case**:

- The activity is **museum**, and
- The weather is **rainy**.

Each is represented as a **one-hot vector**, and those vectors are **concatenated** — so the RNN receives both the **past activity and weather** as its input.

In the middle section, we apply the RNN's forward formula:

$$h_t = \sigma_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \quad \text{and} \quad \hat{y}_t = \sigma_o(W_{oh}h_t + b_o)$$

This combines:

- The **input vector** (activity + weather),
- The **hidden state** from the previous step,

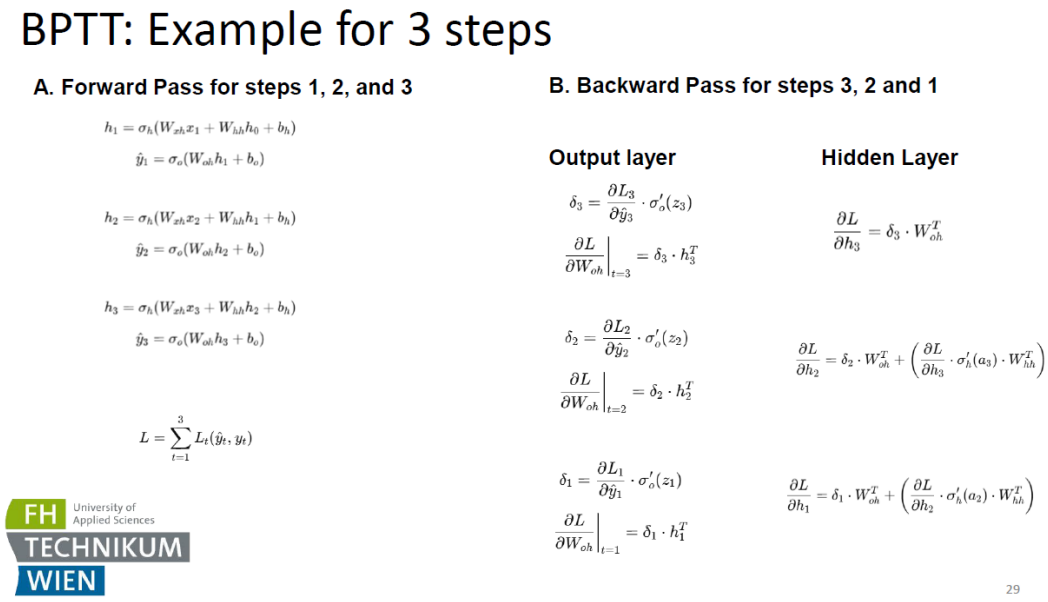
- And applies **weights** and an **activation function** (= nonlinearity like tanh or ReLU).

At the bottom, we see the **activation step**:

- The network computes weighted sums,
- Applies a non-linear function to squash the result,
- And finally **outputs a new activity prediction** — in this case, **movies**.

This slide illustrates the **entire RNN loop** in action — starting from inputs, processing with matrix math, and generating a prediction — just like in any real sequence model.

This is how even complex applications — like language translation or time-series forecasting — are built on simple building blocks like these."



Presentation Script – Slide 29: BPTT – Example for 3 Steps

"Let's now dive into an example of **Backpropagation Through Time (BPTT)** applied to a sequence of **three time steps**.

We'll break it into two parts: **A: Forward Pass** and **B: Backward Pass**

0.0.1 Forward Pass – Left Side

Here, we process the inputs step-by-step:

- At each time t, we compute the **hidden state** h_t from the current input x_t and previous hidden state h_{t-1}
- Then, we compute the **output** $\hat{y}_t$ from h_t
- This is done for steps 1, 2, and 3

At the bottom, we calculate the **total loss** across all time steps:

$$L = \sum_{t=1}^{3} L_t(\hat{y}_t, y_t)$$

0.0.2 Backward Pass – Right Side

Now we work **backwards** in time — this is where the RNN's recurrence makes things more complex.

Output Layer Gradients For each time step t, we:

- Compute the error term δ_t for the output layer
- Multiply by the derivative of the activation function
- And compute the partial derivative of the loss with respect to the output weights W_{oh}

This happens separately for $t = 3$, $t = 2$, and $t = 1$

Hidden Layer Gradients Here's where it gets recursive.

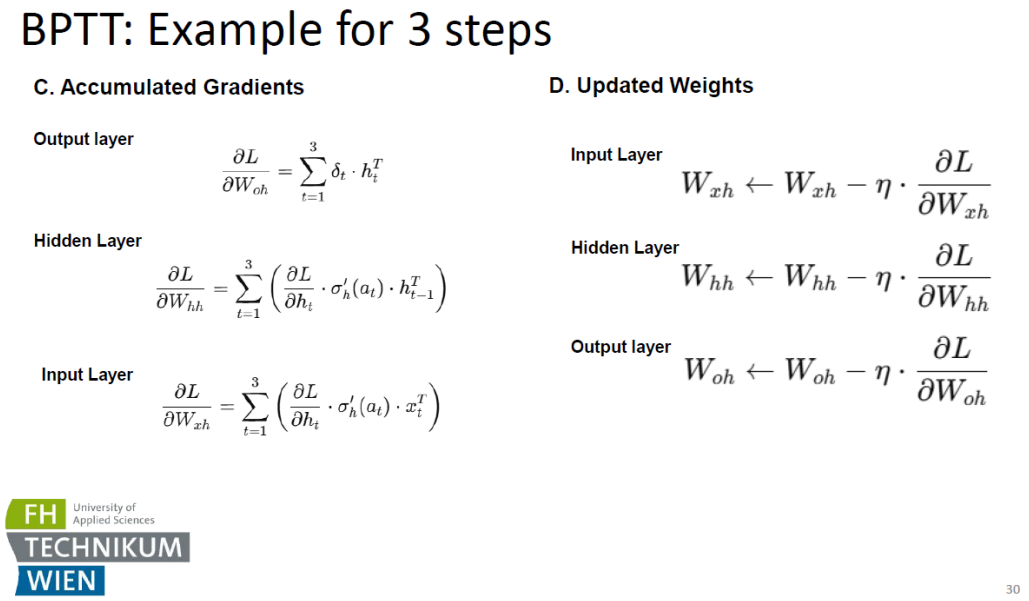
- To compute the gradient of the loss with respect to h_t, we not only consider its immediate contribution but also the **effect it has on all future time steps** — through the recurrence

For example:

$$\frac{\partial L}{\partial h_2} = \delta_2 W_{oh}^T + \left(\frac{\partial L}{\partial h_3} \cdot \sigma_h' \cdot W_{hh}^T \right)$$

This is what makes BPTT **costly and sensitive to gradient issues**, especially for longer sequences.

This slide helps you see the **entire BPTT loop mathematically**, making clear how gradients propagate **not just layer to layer**, but **step to step through time**.



Presentation Script – Slide 30: BPTT – Accumulated Gradients and Weight Updates

"This slide concludes our step-by-step breakdown of **Backpropagation Through Time** by showing two final stages:

0.0.3 Accumulated Gradients (Left Side)

To update weights, we first need to **accumulate the total gradient** across time steps.

Output Layer:

$$\frac{\partial L}{\partial W_{oh}} = \sum_{t=1}^{3} \delta_t \cdot h_t^T$$

We take the contribution from each time step and sum them — this gives us the **total error influence** for the output weight matrix.

Hidden and Input Layers:

Here, things are a bit more complex. We take into account the derivative of the loss with respect to each hidden state and then multiply by the derivative of the activation function.

We then multiply by:

- h_{t-1} for hidden-to-hidden weights
- x_t for input-to-hidden weights

This step ensures that the **entire time sequence contributes** to the final weight gradients.

0.0.4 Weight Updates (Right Side)

Once we have the gradients, we use standard **gradient descent**:

$$W \leftarrow W - \eta \cdot \frac{\partial L}{\partial W}$$

This is applied to:

- Input weights W_{xh}
- Recurrent weights W_{hh}
- Output weights W_{oh}

This completes the BPTT cycle:

1. Forward pass over time
2. Backward pass through time
3. Accumulate gradients
4. Update weights

All of this allows the RNN to learn from **sequences**, one step at a time — across time and layers.
