AI INSTITUTE #AIISC
UNIVERSITY OF SOUTH CAROLINA

UNIVERSITY OF SOUTH CAROLINA

CSCE 771: Computer Processing of Natural Language
Lecture: Language Modeling Variants - Mamba

PROF. BIPLAV SRIVASTAVA, AI INSTITUTE

26th SEP 2024

Carolinian Creed: "I will practice personal and academic integrity."

Organization of Lecture 12

- Opening Segment
  - Review of Lecture 111
- Main Lecture

Main Section
- RNNs and Family
  - RNNs
  - Mamba
  - Live Coding - Two CV tasks

- Concluding Segment
  - About Next Lecture – Lecture 13

Recap of Lecture 11

- We covered
  - PyTorch
  - BERT
- We saw
  - how model's are constructed and trained using a convenience library called pytorch
  - how BERT can be used in two ways for two CV-related tasks
- We discussed handling of abstract mathematical objects

Main Lecture

About Me

South Carolina

University of South Carolina

Visit My Webpage

I am Kaushik Roy, a Ph.D. candidate at the Artificial Intelligence Institute, University of South Carolina. My research focuses on developing neurosymbolic methods for declarative and process knowledge-infused learning, reasoning, and sequential decision-making, with a particular emphasis on social good applications. My academic journey has taken me from R.V. College of Engineering in Bangalore for my Bachelor's to Indiana University Bloomington for my Master's, and briefly to the University of Texas at Dallas before settling at the University of South Carolina for my doctoral studies. My research interests span machine learning, artificial intelligence, and their application in social good settings. I'm passionate about pushing the boundaries of AI, particularly in areas where it intersects with human understanding and decision-making.

Topics of Interest to Me: Neurosymbolic AI Knowledge-infused Learning AI for Social Good Healthcare Informatics

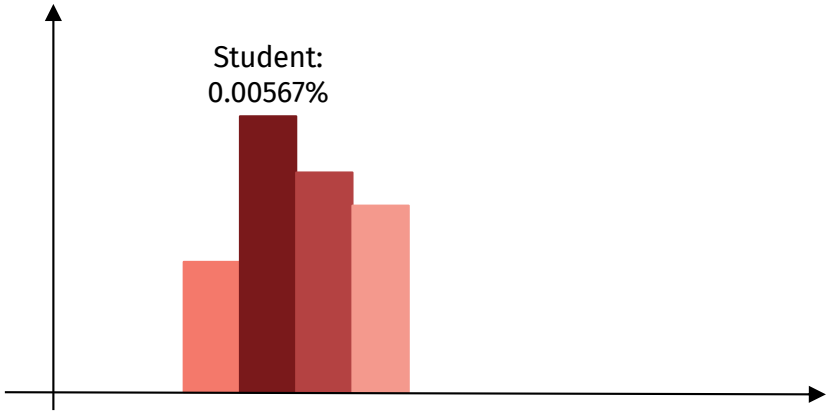
Language Modeling

Input Kaushik Roy is a PhD Student

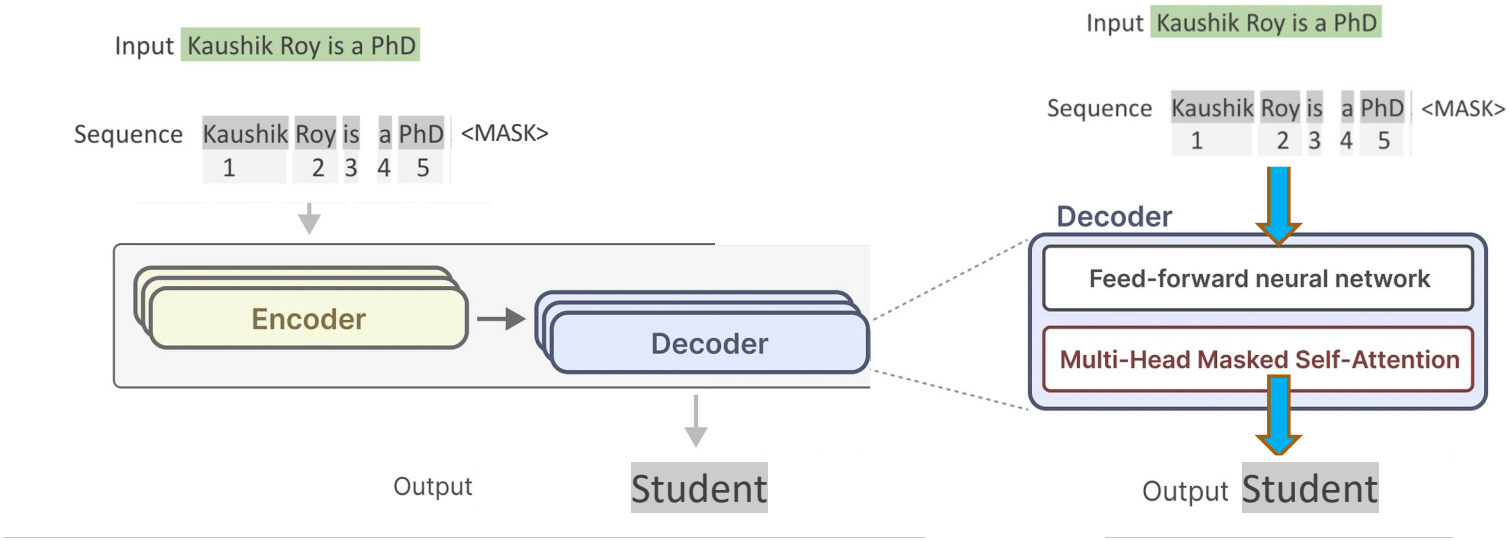
Sequence	Kaushik	Roy	is	a	PhD	Student
	1	2	3	4	5	6

Student

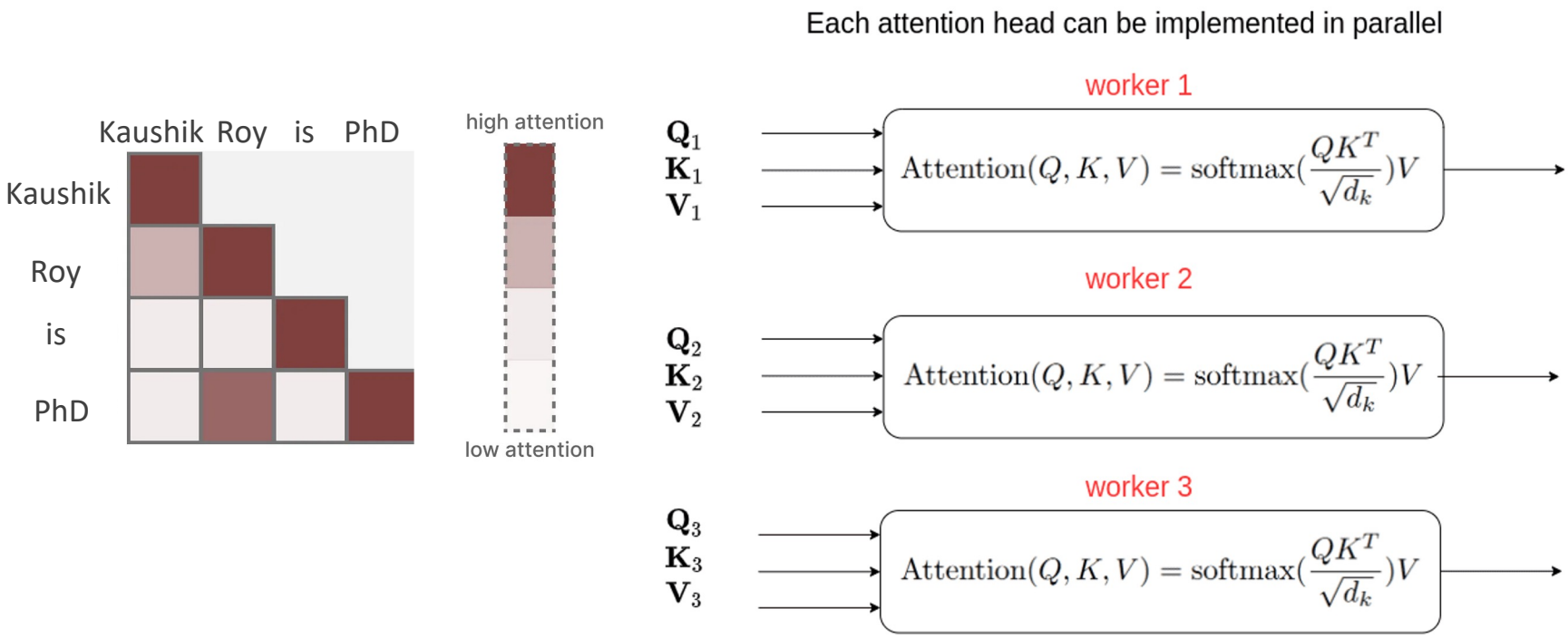
Did you mean: Student
Did you mean: Student Square
Did you mean: Student Success



BERT and family



BERT and family - Multi-headed Self-Attention



Data - CV data as a PDF

Follow Links

Links to my Google Scholar and Socials

South Carolina
University of South Carolina
Email
Google Scholar
Github
LinkedIn

Curriculum Vitae, Updated September 14th 2024.

🎓 Education

- **2020 – Present**: Ph.D. Candidate, University of South Carolina
  - Topic: Process Knowledge-infused Learning and Reasoning
- **2017 – 2020**: Ph.D. Student, University of Texas at Dallas (Transferred in 2020)
  - Topic: Relational Sequential Decision Making [Qualifier Document]
- **2015 – 2017**: M.Sc. Computer Science, Indiana University Bloomington
  - Specialization: Artificial Intelligence and Machine Learning
- **2011 – 2015**: B.E. Computer Science, RV College of Engineering
  - Thesis title: Computer Vision Algorithms for Background Understanding [Thesis Document]

📚 Publications

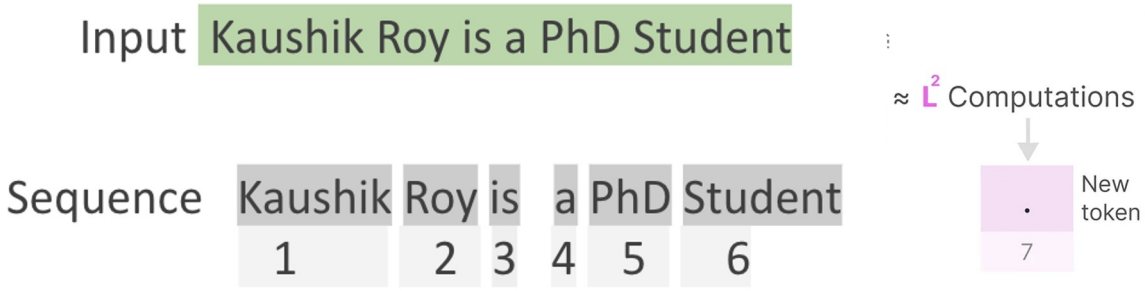
📘 Book Chapters

Tasks

- CV-related tasks
  - Extractive Summarization – Extract Institution, Degree, and Year
    - Institution: UofSC
    - Degree: Ph.D.
    - Year: 2024
  - Abstractive Summarization - Masked Language Filling with Partial Queries
    - Kaushik Roy is a ?__

Image Source

Revisiting (Recurrent Neural Networks) RNNs and Family

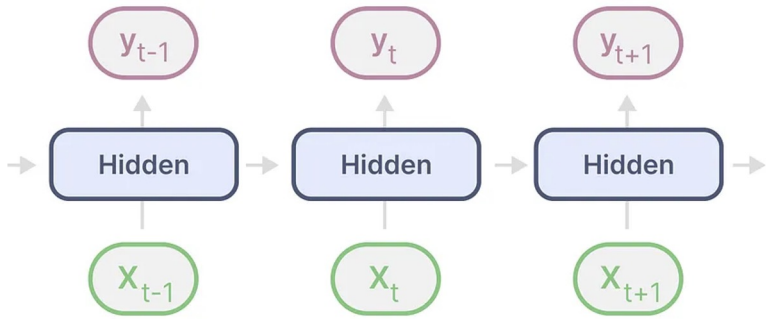


Generating tokens for a sequence of length L needs roughly L^2 computations which can be costly if the sequence length increases.

	Training	Inference
Transformers	**Fast!** (parallelizable)	**Slow...** (scales **quadratically** with sequence length)

Image Source

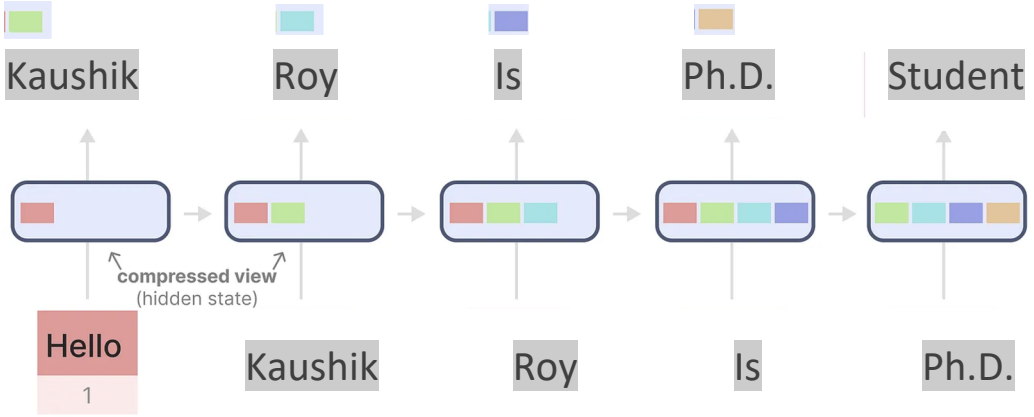
Long Context Issues



- Notice that there is forgetting when maintaining long context!
- Ad-hoc improvements – (Long term Short Term) LSTM, (Gated Recurrent Units) GRUs, etc.

In other words, RNNs can do inference fast as it scales linearly with the sequence length! In theory, it can even have an *infinite context length*.

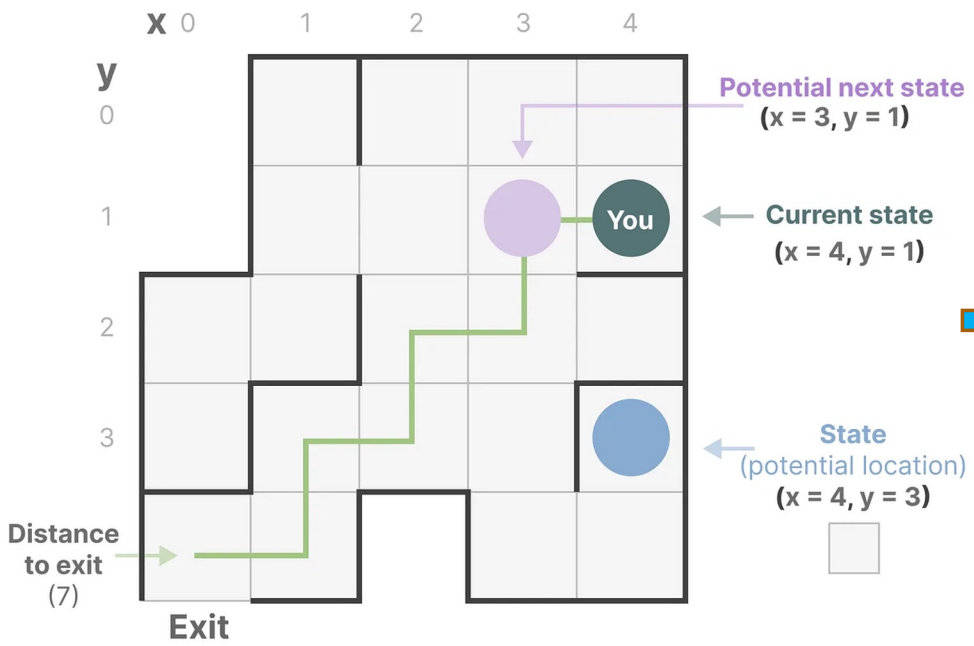
To illustrate, let's apply the RNN to the input text we have used before.

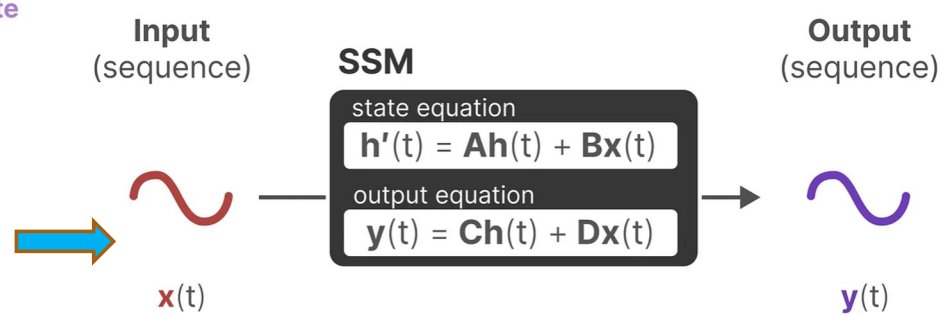


Each hidden state is the aggregation of all previous hidden states and is typically a compressed view.

Image Source

State Space Models

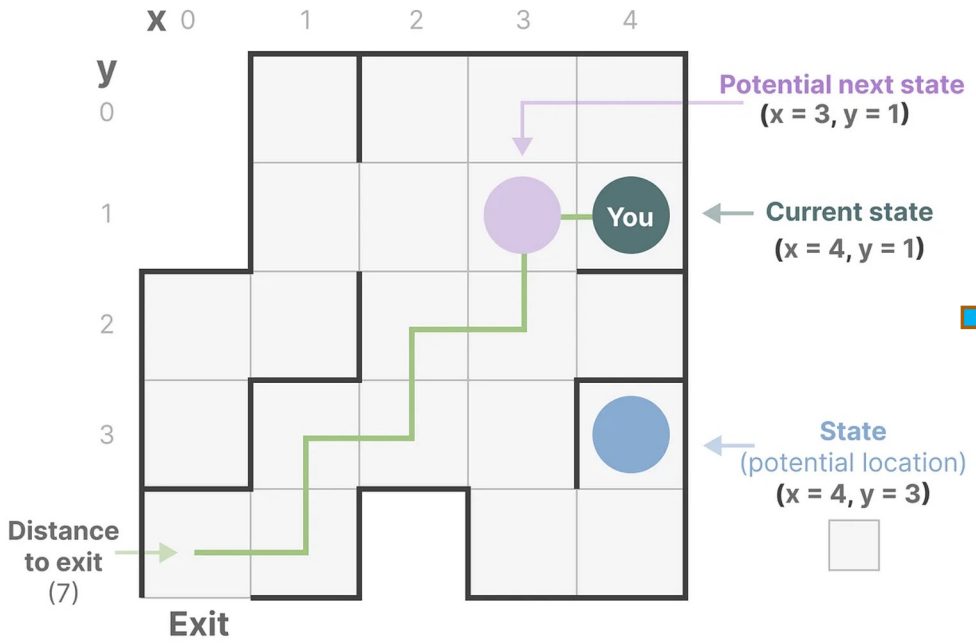


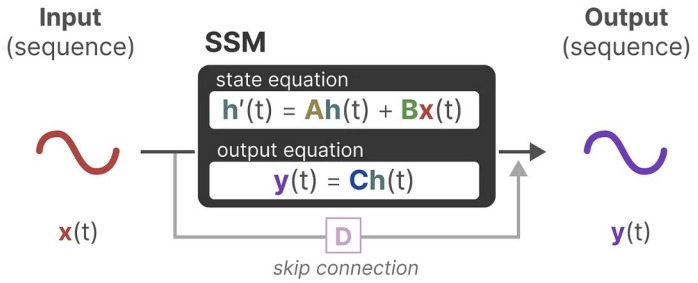


- The theoretical generalization of RNNs, where a system state is maintained to track essential elements, as required for prediction.

Image Source

RNN Family - Mamba

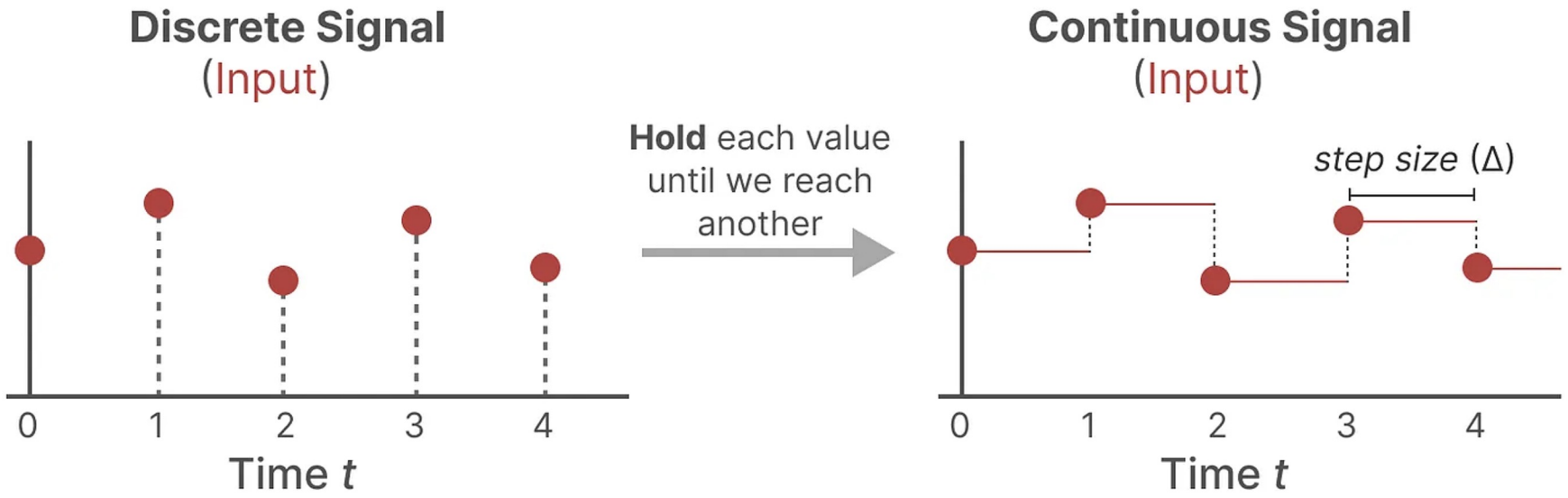




- Mamba is a special type of SSM that makes specific and informed design choices towards (i) Long context tracking (ii) Expressive context representations (iii) Efficient training/inference

Image Source

(i) RNN Family - Mamba (Discretization)



- SSM theory is defined for continuously evolving systems, and in the finite data regime, we require discretization that achieves a kind of "prolong holding of signal", i.e., long context state tracking

Image Source

(i) RNN Family - Discrete Mamba

- The exponentiation-based discretization utilized by Mamba is particularly simple and intuitive to understand, as exponentiation simply amplifies the signal until the next discrete time point

Discretized matrix **A** $\bar{\mathbf{A}} = \exp(\Delta\mathbf{A})$

```
    self.A = nn.Parameter(torch.randn(d_model, state_size))
    nn.init.xavier_uniform_(self.A)

def forward(self, x):
    batch_size, seq_len, _ = x.shape  # Dynamically infer the sequence length from input

    # Initialize dynamic buffers
    B = torch.zeros(batch_size, seq_len, self.state_size, device=x.device)
    C = torch.zeros(batch_size, seq_len, self.state_size, device=x.device)
    delta = torch.zeros(batch_size, seq_len, self.d_model, device=x.device)
    dA = torch.zeros(batch_size, seq_len, self.d_model, self.state_size, device=x.device)
    h = torch.zeros(batch_size, seq_len, self.d_model, self.state_size, device=x.device)
```

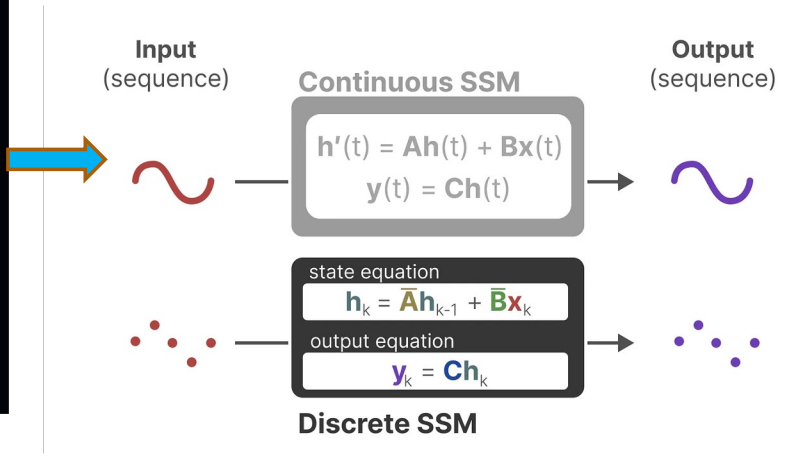


Image Source

(i) Mamba (S6: Structured State Space for Sequence Modeling with Selective Scanning)

- Thus Mamba in it's entirety is a discretized state space model
- as It is called an S6 model because it is first, an S4 model which is an abbreviation for "The Structured State Space for Sequence Modeling"
- The S6 comes from it's selective scanning procedure, where the discretization matrix effective **selects** how much of the context to **hold**

```python
# Step 3: Define the Mamba Model
class S6(nn.Module):
    def __init__(self, d_model, state_size):
        super(S6, self).__init__()
        self.fc1 = nn.Linear(d_model, d_model)
        self.fc2 = nn.Linear(d_model, state_size)
        self.fc3 = nn.Linear(d_model, state_size)
        self.d_model = d_model
        self.state_size = state_size

        self.A = nn.Parameter(torch.randn(d_model, state_size))
        nn.init.xavier_uniform_(self.A)

    def forward(self, x):
        batch_size, seq_len, _ = x.shape  # Dynamically infer the sequence length from input

        # Initialize dynamic buffers
        B = torch.zeros(batch_size, seq_len, self.state_size, device=x.device)
        C = torch.zeros(batch_size, seq_len, self.state_size, device=x.device)
        delta = torch.zeros(batch_size, seq_len, self.d_model, device=x.device)
        dA = torch.zeros(batch_size, seq_len, self.d_model, self.state_size, device=x.device)
        h = torch.zeros(batch_size, seq_len, self.d_model, self.state_size, device=x.device)

        # Apply linear transformations
        B = self.fc2(x)
        C = self.fc3(x)
        delta = F.softplus(self.fc1(x))

        # Discretization operation
        dA = torch.exp(torch.einsum("bld,dn->bldn", delta, self.A))

        # Compute h and y (output)
        h = torch.einsum('bldn,bldn->bldn', dA, h) + torch.unsqueeze(x, dim=-1)
        y = torch.einsum('bln,bldn->bld', C, h)

        return y
```

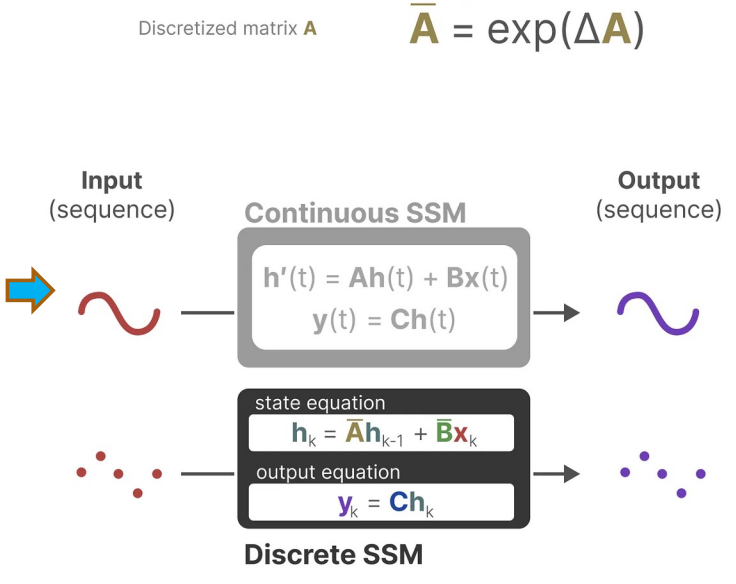


Image Source

(ii) Non-Linear Mamba and HIPPO

- So far we have seen Mamba handle linear matrices
- But of course real world, and especially language data must have non-linear dependencies
- Mamba achieves this using **HIPPO**: High-order Polynomial Projection Operators, specifically by leveraging **Legendre Polynomials?!**

```python
# Legendre Polynomial Function
def legendre_polynomials(x, order=5):
    """
    Compute the first few Legendre polynomials (up to a given order) for each input in x.
    Args:
    x (torch.Tensor): Input tensor of shape (batch_size, seq_len, d_model).
    order (int): The maximum order of the Legendre polynomial to compute.

    Returns:
    torch.Tensor: Transformed input with additional Legendre polynomial features.
    """
    batch_size, seq_len, d_model = x.shape

    # Initialize a list to store polynomials P_0(x), P_1(x), ..., P_order(x)
    polynomials = []

    # P_0(x) = 1 (constant)
    P0 = torch.ones_like(x)
    polynomials.append(P0)

    # P_1(x) = x
    P1 = x
    polynomials.append(P1)

    # Recursively compute P_n(x) for n = 2, 3, ..., order
    for n in range(2, order + 1):
        Pn = ((2 * n - 1) * x * polynomials[n - 1] - (n - 1) * polynomials[n - 2]) / n
        polynomials.append(Pn)

    # Stack all polynomials together along the last dimension (d_model)
    # This will create additional features for each input based on Legendre polynomials
    return torch.cat(polynomials, dim=-1)
```

Image Source

(ii) Recall Polynomial Regression

Matrix form and calculation of estimates [edit]

The polynomial regression model

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m + \varepsilon_i \ (i = 1, 2, \ldots, n)$$

can be expressed in matrix form in terms of a design matrix $\mathbf{X}$, a response vector $\vec{y}$, a parameter vector $\vec{\beta}$, and a vector $\vec{\varepsilon}$ of random errors. The *i*-th row of $\mathbf{X}$ and $\vec{y}$ will contain the *x* and *y* value for the *i*-th data sample. Then the model can be written as a system of linear equations:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_3 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

which when using pure matrix notation is written as

$$\vec{y} = \mathbf{X}\vec{\beta} + \vec{\varepsilon}.$$

The vector of estimated polynomial regression coefficients (using ordinary least squares estimation) is

$$\hat{\vec{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\,\mathbf{X}^\mathsf{T}\vec{y},$$

- Polynomials provide a way to expand a linear model to capture non-linear dependencies
- Polynomials are also **universal approximators** (roughly), i.e., they can approximate any non-linear function upto an arbitrary degree of precision (see Taylor series)
- But successive polynomials are clearly correlated with previous ones, **(e.g., x^2 correlated with x), and in learning, we like uncorrelated features**

Image Source

(ii) Recall Gram Schmidt Orthogonalization

Gram–Schmidt process

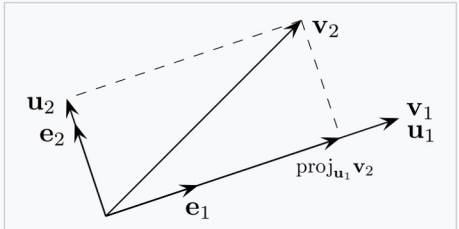
35 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In mathematics, particularly linear algebra and numerical analysis, the **Gram–Schmidt process** or Gram-Schmidt algorithm is a way of finding a set of two or more vectors that are perpendicular to each other.

By technical definition, it is a method of constructing an orthonormal basis from a set of vectors in an inner product space, most commonly the Euclidean space $\mathbb{R}^n$ equipped with the standard inner product. The Gram–Schmidt process takes a finite, linearly independent set of vectors $S = \{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ for $k \leq n$ and generates an orthogonal set $S' = \{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ that spans the same k-dimensional subspace of $\mathbb{R}^n$ as S.



The first two steps of the Gram–Schmidt process

The method is named after Jørgen Pedersen Gram and Erhard Schmidt, but Pierre-Simon Laplace had been familiar with it before Gram and Schmidt.[1] In the theory of Lie group decompositions, it is generalized by the Iwasawa decomposition.

The application of the Gram–Schmidt process to the column vectors of a full column rank matrix yields the QR decomposition (it is decomposed into an orthogonal and a triangular matrix).

- Gram schmidt orthogonalization is a procedure by which a set of correlated vectors can be converted into an orthogonal set of vectors
- When a polynomial basis is **orthogonalized, i.e., made uncorrelated** using the Gram schmidt orthogonalization process, we **get legendre polynomials**

Image Source

(ii) Legendre Polynomials

The first few Legendre polynomials are:

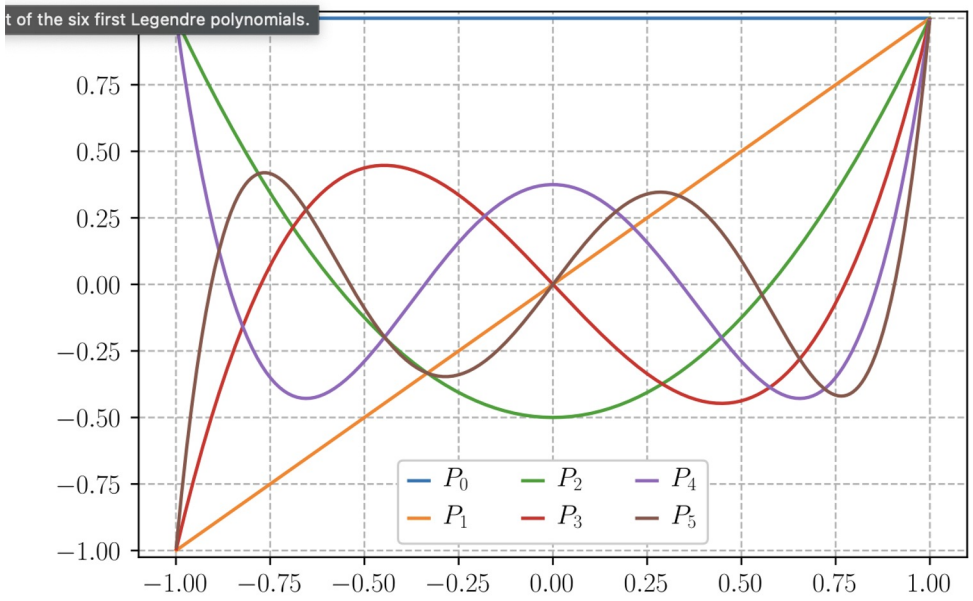
n	$P_n(x)$
0	1
1	x
2	$\frac{1}{2}\left(3x^2-1\right)$
3	$\frac{1}{2}\left(5x^3-3x\right)$
4	$\frac{1}{8}\left(35x^4-30x^2+3\right)$
5	$\frac{1}{8}\left(63x^5-70x^3+15x\right)$
6	$\frac{1}{16}\left(231x^6-315x^4+105x^2-5\right)$
7	$\frac{1}{16}\left(429x^7-693x^5+315x^3-35x\right)$
8	$\frac{1}{128}\left(6435x^8-12012x^6+6930x^4-1260x^2+35\right)$
9	$\frac{1}{128}\left(12155x^9-25740x^7+18018x^5-4620x^3+315x\right)$
10	$\frac{1}{256}\left(46189x^{10}-109395x^8+90090x^6-30030x^4+3465x^2-63\right)$

- Gram schmidt orthogonalization is a procedure by which a set of correlated vectors can be converted into an orthogonal set of vectors
- When a polynomial basis is **orthogonalized, i.e., made uncorrelated** using the Gram schmidt orthogonalization process, we **get legendre polynomials**

Image Source

(ii) Legendre Polynomials

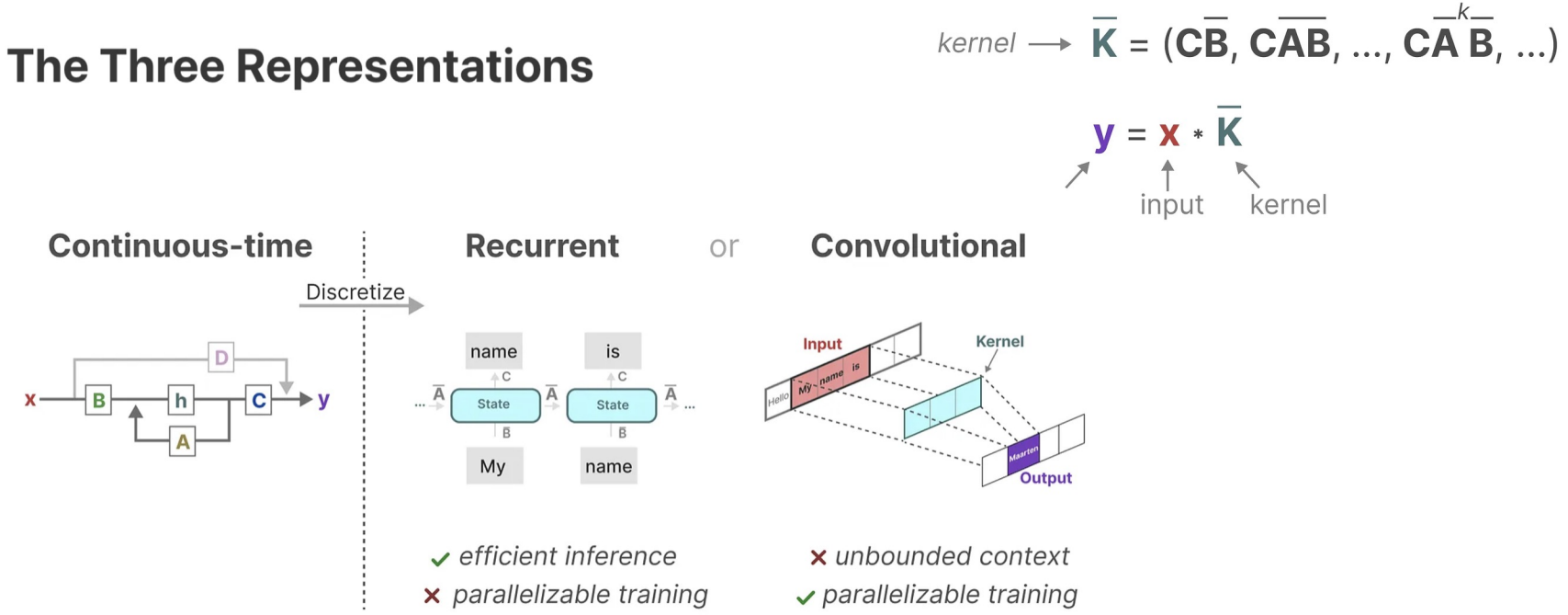
The graphs of these polynomials (up to $n = 5$) are shown below:



- The graph helps visualize intuitively how any non linear function can be captured by some combination of legendre polynomials
- As we can see, even upto just degree five they already create a very expressive basis

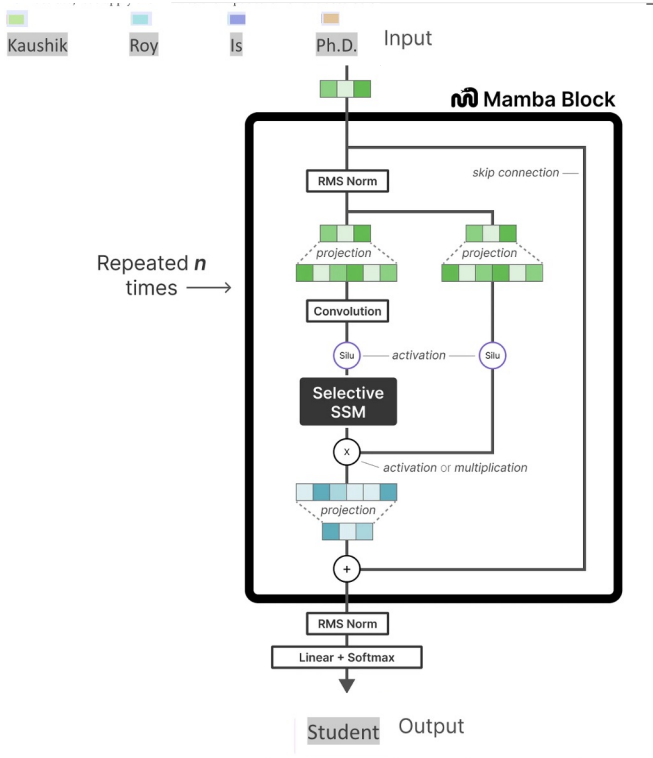
Image Source

(iii) Representation of the Operations for Efficiency



[Image Source]

Putting it All together - Mamba



Simplified for Pedagogical Purpose

```python
class MambaBlock(nn.Module):
    def __init__(self, d_model, state_size, legendre_order=5):
        super(MambaBlock, self).__init__()
        self.legendre_order = legendre_order  # Order of Legendre polynomials
        self.inp_proj = nn.Linear((legendre_order + 1) * d_model, 2 * d_model)
        self.out_proj = nn.Linear(2 * d_model, d_model)
        self.s6 = S6(2 * d_model, state_size)
        self.norm = nn.LayerNorm(2 * d_model)  # LayerNorm matches the d_model a

    def forward(self, x):
        # Apply Legendre polynomials to the input
        x_legendre = legendre_polynomials(x, order=self.legendre_order)  # Non-

        # Project input to 2*d_model after Legendre expansion
        x_proj = self.inp_proj(x_legendre)
        x_proj = self.norm(x_proj)  # Apply normalization
        x_ssm = self.s6(x_proj)  # Pass through S6 module
        x_out = self.out_proj(x_ssm)  # Project back to d_model dimension
        return x_out
```

Image Source

Live Coding

Mamba-based CV processing

Do in parallel live coding with your resume.

```python
# Mamba Block with Legendre Polynomial Transformation
class MambaBlock(nn.Module):
    def __init__(self, d_model, state_size, legendre_order=5):
        super(MambaBlock, self).__init__()
        self.legendre_order = legendre_order  # Order of Legendre polynomials
        self.inp_proj = nn.Linear((legendre_order + 1) * d_model, 2 * d_model)  # Adjust input size after Legendre expansion
        self.out_proj = nn.Linear(2 * d_model, d_model)
        self.s6 = S6(2 * d_model, state_size)
        self.norm = nn.LayerNorm(2 * d_model)  # LayerNorm matches the d_model after projection

    def forward(self, x):
        # Apply Legendre polynomials to the input
        x_legendre = legendre_polynomials(x, order=self.legendre_order)  # Non-linear transform

        # Project input to 2*d_model after Legendre expansion
        x_proj = self.inp_proj(x_legendre)
        x_proj = self.norm(x_proj)  # Apply normalization
        x_ssm = self.s6(x_proj)  # Pass through S6 module
        x_out = self.out_proj(x_ssm)  # Project back to d_model dimension
        return x_out

# Full Mamba Model
class Mamba(nn.Module):
    def __init__(self, d_model, state_size, vocab_size, legendre_order=5):
        super(Mamba, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)  # Embedding layer
        self.mamba_block1 = MambaBlock(d_model, state_size, legendre_order)
        self.mamba_block2 = MambaBlock(d_model, state_size, legendre_order)
        self.mamba_block3 = MambaBlock(d_model, state_size, legendre_order)
        self.fc_out = nn.Linear(d_model, vocab_size)  # Final output layer for MLM

    def forward(self, x):
        x = self.embedding(x)  # Embed the input tokens to shape (batch_size, seq_len, d_model)
        x = self.mamba_block1(x)
        x = self.mamba_block2(x)
        x = self.mamba_block3(x)
        return self.fc_out(x)  # Return logits for each token

# Step 4: Training the Mamba Model
# Hyperparameters
d_model = 128  # Dimensionality of the model
state_size = 256  # Size of the hidden state
batch_size = 32
num_epochs = 5
vocab_size = tokenizer.vocab_size  # Vocabulary size from the tokenizer
```

Concluding Segment

Image Source

Concluding Comments

- We looked at the Mamba model
  - learned about the design choices that went into the Mamba model that make it superior for long context sequential modeling compared to transformer models
  - Did CV processing using it
  - Discussed tradeoffs

Image Source

About Next Lecture – Language Modeling

- Playground
- Comparisons on a suite of tasks

Course Project

Discussion: Course Project

Theme: Analyze quality of official information available for elections in 2024 [in a state]

- Take information available from
  - Official site: State Election Commissions
  - Respected non-profits: League of Women Voters
- Analyze information
  - State-level: Analyze quality of questions, answers, answers-to-questions
  - Comparatively: above along all states (being done by students)
- Benchmark and report
  - Compare analysis with LLM
  - Prepare report

- Process and analyze using NLP
  - Extract entities
  - Assess quality – metrics
    - Content – *Englishness*
    - Content – *Domain* -- election
  - … other NLP tasks
  - Analyze and communicate overall

Major dates for project check

- Sep 10: written – project outline
- Oct 8: in class
- Oct 31: in class // LLM
- Dec 5: in class // Comparative

About Next Lecture – Lecture 13

Lecture 13 Outline

- Comparing architectures
- Comparing finetuning
- Applying a new domain/ task – elections?

7	Sep 10 (Tu)	Statistical parsing, QUIZ
8	Sep 12 (Th)	Evaluation, Semantics
9	Sep 17 (Tu)	Semantics, Machine Learning for NLP, Evaluation - Metrics
10	Sep 19 (Th)	Towards Language Model: Vector embeddings, Embeddings, CNN/ RNN
11	Sep 24 (Tu)	Language Model – PyTorch, BERT, {Resume data, two tasks} **– Guest Lecture**
12	Sep 26 (Th)	Language Model – Finetuning, Mamba - **Guest Lecture**
13	Oct 1 (Tu)	Language model – comparing arch, finetuning - **Guest Lecture**
14	Oct 3 (Th)	Language model – comparison of results, discussion, ongoing trends– **Guest Lecture**
15	Oct 8 (Tu)	PROJ REVIEW