The Henry and Marilyn Taub
Faculty of Computer Science

Module 4: Crash Course on Deep Learning

StructML Course

Benny Kimelfeld
Technion – Israel Institute of Technology

Spring 2026

Lecture Goals

- Introduce the general/conceptual *structure & mathematical formulation* of deep neural networks
- Introduce *specific architectures & building blocks* that we encounter later
  - Specifically, understand the attention mechanism
- Explain the learning process through *backpropagation & GPU computation*
  - Mainly to understand the context, requirements, and practical limitations we meet when building deep networks over structured data

The module does not aim to replace a full deep learning course! We will use the basis here to focus on **deep learning over structured data**: tables, graphs, databases.

Outline

Outline

Basic Notation I

- By a *vector* we refer to a *column vector*, unless explicitly stated otherwise
- A n-dimensional vector (n-*vector* for short) has the form

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

- We can also write it by $[v_1, \ldots, v_n]^\top$, that is, the *transpose* of a row vector, or simply by $(v_1, \ldots, v_n)$ when there is no risk of confusion

Basic Notation II

- $\mathbb{R}^n$ refers to column vectors with n elements, and $\mathbb{R}^{n \times m}$ refers to matrices with n rows and m columns
  - Hence, $\mathbb{R}^n$ can be identified with $\mathbb{R}^{n \times 1}$
- Recall: if $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{v} \in \mathbb{R}^n$, then $(\mathbf{A}\mathbf{v})^\top = \mathbf{v}^\top \mathbf{A}^\top$
  - Generally, $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top \mathbf{A}^\top$
- For $\mathbf{v}, \mathbf{u} \in \mathbb{R}^n$ we use $\mathbf{v} \odot \mathbf{u}$ for the *element-wise product* $(v_1 \cdot u_1, \ldots, v_n \cdot u_n)$
  - Hence, $\mathbf{v} \cdot \mathbf{u} \in \mathbb{R}$ while $\mathbf{v} \odot \mathbf{u} \in \mathbb{R}^n$

Outline

Units

- A Deep Neural Network (DNN) is a representation of a parameterized numerical formula defined by a composition of simple formulas, called *DNN units*
  - We call them just *units* when the context is clear
- A *unit* is described by an expression of the form

$$u(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{y}$$

  - $\mathbf{x} = (x_1, \ldots, x_n)$ is an n-vector of input variables
  - $\mathbf{y} = (y_1, \ldots, y_m)$ is an m-vector of output variables
  - $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_k)$ is a k-vector of parameters
  - $u : \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^m$ is a numerical function

Notes on Units

$$u(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{y}$$

- Involved vectors can be organized as *matrices*, so oftentimes you will see abstractions via matrices rather than vectors
- The parameters are learned via gradient-based optimization, so u is *differentiable*
  - Differentiable in $\boldsymbol{\theta}$ — since we take the gradient of the parameters in GD
  - Differentiable in $\mathbf{x}$ — since we wish to compose units (next), so $\mathbf{x}$ may refer to functions with their own parameters
  - "Almost everywhere" — we can allow a finite set of non-differentiable points
    - E.g., ReLU, discussed later

Deep Neural Networks

- A *DNN* is a composition of DNN units; specifically:
- Every DNN unit is a DNN
- If $g_1(\mathbf{x}_1;\boldsymbol{\theta}_1)$, ..., $g_q(\mathbf{x}_q;\boldsymbol{\theta}_q)$ are DNNs and $f'(\mathbf{x}';\boldsymbol{\theta}')$ is a DNN unit, then

$$f(\mathbf{x};\boldsymbol{\theta}) = f'\big(g_1(\mathbf{x}_1;\boldsymbol{\theta}_1),\ldots,g_q(\mathbf{x}_q;\boldsymbol{\theta}_q);\boldsymbol{\theta}'\big)$$

  is a DNN (assuming that all dimensions match)
- Here:
  - $\mathbf{x}$ consists of all variables that occur in the $\mathbf{x}_i$
  - $\boldsymbol{\theta}$ consists of all parameters that occur in $\boldsymbol{\theta}'$ and in the $\boldsymbol{\theta}_i$

Parameter Sharing

$$f(\mathbf{x};\boldsymbol{\theta}) = f'\big(g_1(\mathbf{x}_1;\boldsymbol{\theta}_1),\ldots,g_q(\mathbf{x}_q;\boldsymbol{\theta}_q);\boldsymbol{\theta}'\big)$$

- We usually assume, implicitly, that different parameter sequences are disjoint
  - That is, $\boldsymbol{\theta}' \cap \boldsymbol{\theta}_i = \boldsymbol{\theta}_i \cap \boldsymbol{\theta}_j = \emptyset$ for $j \neq i$
  - Put differently, each unit brings its own, unique set of parameters
- In some cases, we do not make this assumption, since we design the DNN with *parameter sharing*
  - In such cases, it should be stated explicitly which parameters are shared

Outline

Basic Units

- Linear transformation: $f(\mathbf{x}; \mathbf{A}, \mathbf{b}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ where $\mathbf{A}$ is an $(m \times n)$-matrix of parameters, and $\mathbf{b}$ is an m-vector of parameters (called *bias*)
  - Every parameter is *unique*!
  - The most basic ingredient, used in every neural network
- Softmax: Non-parametric function $f : \mathbb{R}^n \to \mathbb{R}^n$

$$f_i(\mathbf{x}) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

  - Used for translating arbitrary n scores x_i to a *probability distribution* over $\{1, \ldots, n\}$; preserves the ordering of the scores, and $\sum_i f_i(\mathbf{x}) = 1$

Activation Units

- Activation units are non-linear functions, typically unary: $\phi : \mathbb{R} \to \mathbb{R}$
- When applied to a vector $\mathbf{z}$, the semantics is *element-wise*:

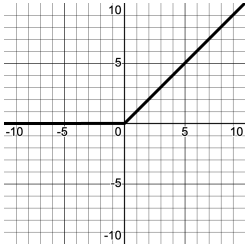
$$\phi\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}\right) = \begin{bmatrix} \phi(z_1) \\ \phi(z_2) \\ \vdots \\ \phi(z_n) \end{bmatrix}$$

- Often the only non-linear components of the network (plus ending softmax)

Note

If all units are linear transformations, then the whole DNN is one linear function, so no gain of expressive power or capacity beyond traditional models.

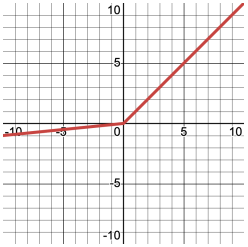
Common Activation Units



ReLU

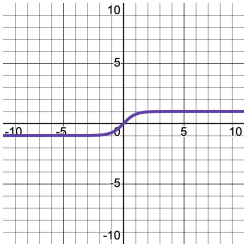
(Rectified Linear Unit)

$$\text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$



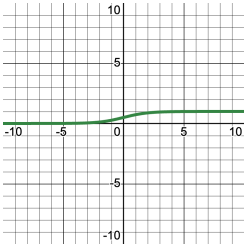
Leaky ReLU

$$a(x;\alpha) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}$$



Hyperbolic tangent

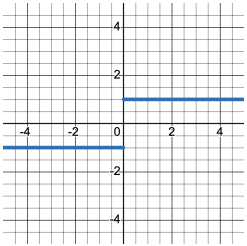
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



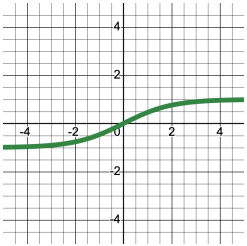
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

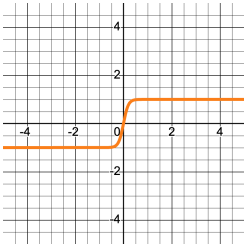
Smooth Alternatives to Sign



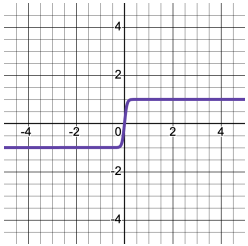
Sign(x)
(Typically intractable to optimize effectively)



$2 \cdot \sigma(x) - 1$



$2 \cdot \sigma(10x) - 1$



$\tanh(10x)$

Universality Theorems

The following universality theorems state that, under reasonable assumptions, each of sigmoid and ReLU (combined with linear units) suffices to approximate every function!

Theorem (Universality of Sigmoid [Cybenko, 1989])

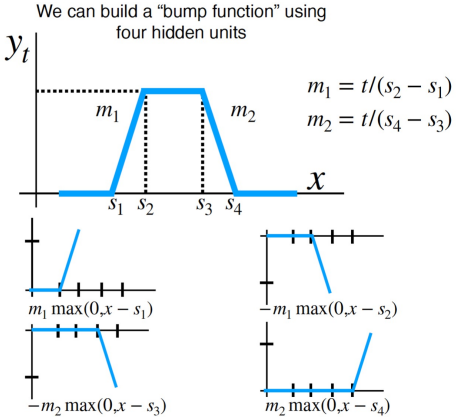
For every continuous function f over a compact subset $\mathcal{K}$ of $\mathbb{R}^n$ and $\epsilon > 0$, there is a function of the form $g(\mathbf{x}) = \mathbf{A} \cdot (\sigma(\mathbf{B}\mathbf{x} + \mathbf{b}))$ such that $|f(x) - g(x)| < \epsilon$ for all $x \in \mathcal{K}$.

Theorem (Universality of ReLU [Lu et al., 2017])

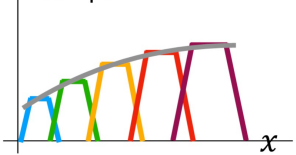
For every integrable function f over $\mathbb{R}^n$ and $\epsilon > 0$ there exists a function g consisting of linear and ReLU activations, such that $\int_{\mathbb{R}^n} |f(x) - g(x)|\, dx < \epsilon$.

These do not mean that we can *learn* every function! Only that ReLU and sigmoid significantly increase the expressive power of linear models

Proof Idea (for ReLU)



With 4K hidden units we can build a sum of K bumps

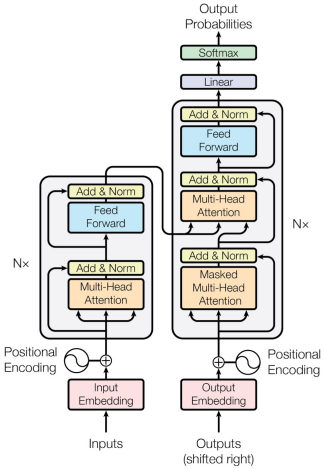


https://deeprob.org/w25/

Outline

Composite Units

- In practice, there are a few common neural networks that are used as *reusable units* that other networks use
- We will discuss a few popular models that we encounter later in the course
- Not an exhaustive survey!
- Example: the *transformer* arch. [Vaswani et al., 2017]:

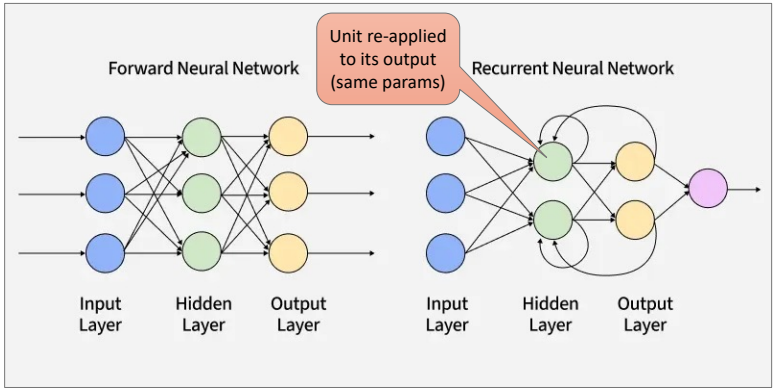


Outline

Feed-Forward Neural Networks

- *FFN* stands for the family of DNNs where information flows strictly forward — from inputs, through intermediate layers (sharing no params), to outputs
- Structure: layers of NN units
- Connections can be *fully connected* ("FC," e.g., MLP) or *selective* (e.g., convolutional)
  - That is, either all inputs contribute to all outputs or some to some
- In contrast, in a *recurrent network* (e.g., RNN, LSTM), neurons use their previous outputs as part of the next computation
  - Enabling sequential processing and memory, suitable for text
- In practice, the term "FFN" usually refers to a simple composition of layers without much sophistication (e.g., linear followed by an activation)

Illustration



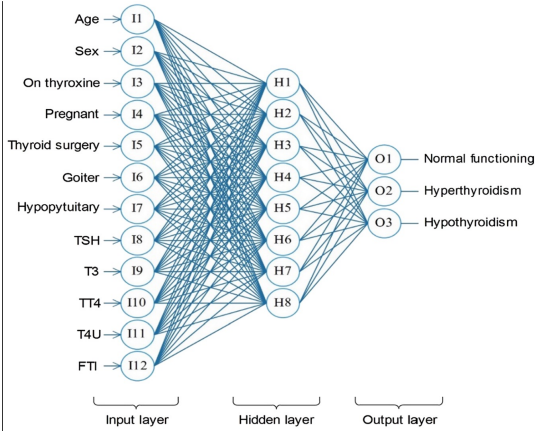
https://www.geeksforgeeks.org/data-analysis/difference-between-feed-forward-neural-networks-and-recurrent-neural-networks/

Example: Multilayer Perceptron (MLP)

- An *MLP* [Rumelhart et al., 1986] is an FFN defined by a sequence of fully connected layers, each consisting of a *linear* and *activation* units, followed by an *output layer* for the final result
- Different MLPs can differ in:
  - The *number of layers*
  - The *activation function*
  - The *layer width* (dimension)
  - The *output layer*
- Following is an MLP for multiclass classification with c classes
- The input is an n-vector $\mathbf{x}$, the output is a c-vector $\mathbf{y}$
  - $\mathbf{y}$ represents a probability distribution over the c classes
- We assume an activation function ϕ

MLP Architecture

$$\mathbf{h}^{(0)} := \mathbf{x}$$
$$\mathbf{h}^{(1)} := \phi(\mathbf{A}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)})$$
$$\vdots$$
$$\mathbf{h}^{(\ell)} := \phi(\mathbf{A}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)})$$
$$\mathbf{y} := \text{Softmax}(\mathbf{h}^{(\ell)})$$



An example of an MLP for disease diagnosis [Hosseinzadeh et al., 2021]

Outline

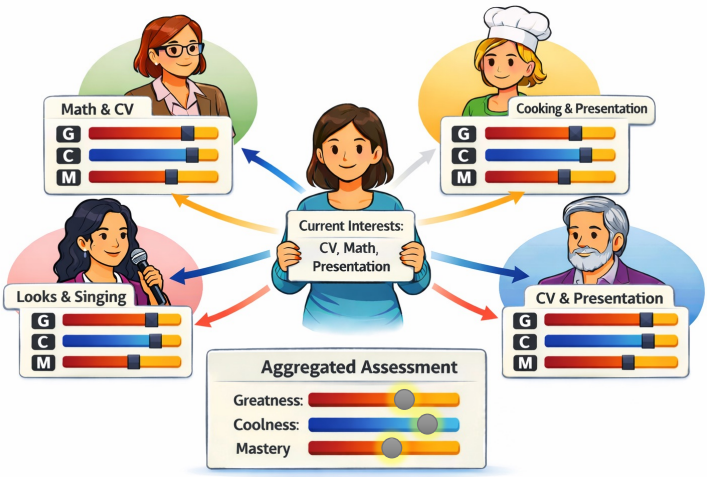
Attention Mechanism: Context

- The *attention mechanism* was proposed originally for language translation [Bahdanau et al., 2015]
- Highlighted by the seminal "Attention Is All You Need" paper [Vaswani et al., 2017], where *transformers* are introduced
  - State-of-the-art models for NLP, where attention is the core unit
- Adopted by other modalities: images, tables, graphs, relational databases, etc.
  - *Graph transformers*, *tabular transformers*, *relational transformers*, etc.
- They provide the richness and flexibility ("capacity") that is essential for *foundation models*, such as LLMs
  - *Capacity* of the model refers to its ability to capture complex patterns
  - Think of information as flowing from the input to the output in *paths* — the more layers you have, the wider layers you have, the more flexibility you have in the neurons — the broader band you have for subtle information

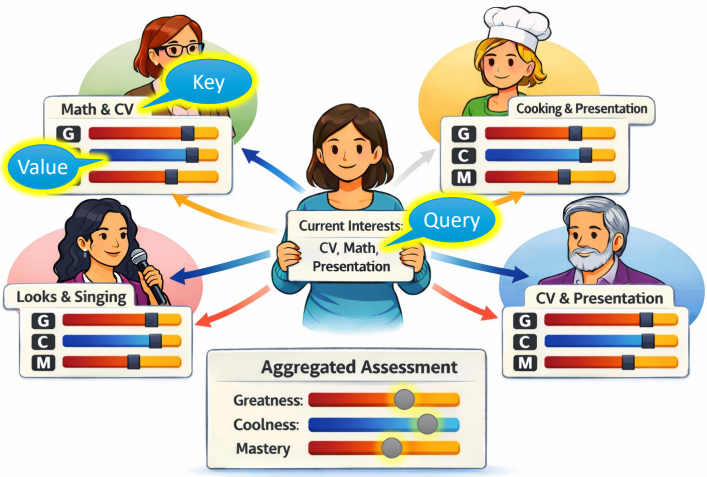
Our Scope

- The following slides explain the idea on a general domain, outside the scope of text, hence a simplified setting
  - Yet, sufficient to capture common usages of attention in ML over structured data
- For example, we do not have a notion of "position" of a token in the sentence

Attention Analogy: Opinion Aggregation



Attention Analogy: Opinion Aggregation



Intuition Is All You Need

- Setting: we have an *observer* x and a context $C = \{c_1, \ldots, c_m\}$ of m *items*, each c_i associated with a value $\mathbf{v}_i$
- We would like to extract from C a single vector $\hat{\mathbf{v}}$ that summarizes the context
- We can take the average of the values, **but**, we would like $\hat{\mathbf{v}}$ to focus on the items that matter to x and ignore what is irrelevant to x (hence, "*attention*")
- So, we want a *weighted average* instead of mere average
- But what are the weights? Specific weights will not generalize — we would like to be able to learn how a *given* x should "attend" to the items of a *given* C
- To this end, we learn a query (*interest*) for x and a key (*offering*) for each c_i, so that the ith weight is obtained by applying the query to the key of c_i
- In effect, the query and key are vectors, and the weight is determined by their dot product (translated into a normalized distribution via softmax)

Attention Unit

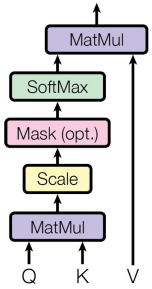
- Input: $\mathbf{q} \in \mathbb{R}^{d_k}$ $\mathbf{K} \in \mathbb{R}^{m \times d_k}$ $\mathbf{V} \in \mathbb{R}^{m \times d_v}$
- Apply the intuition with a conventional *scaling factor* $\sqrt{d_k}$

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) := \text{Softmax}\left(\frac{\mathbf{q} \cdot \mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

- Often, we have a *collection* of observers x instead of one
- In this case, we get $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$ instead of $\mathbf{q} \in \mathbb{R}^{d_k}$
- Hence, we get:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) := \text{Softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Scaled Dot-Product Attention

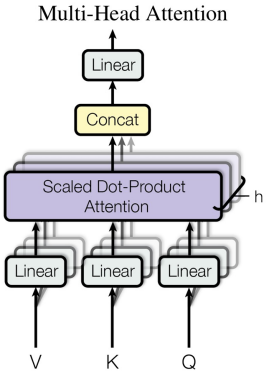


[Vaswani et al., 2017]

Illustration of Multi-Head Attention

To strengthen the attention mechanism, *multi-head attention* applies *multiple attentions* to the input

- Each called an *attention head*
- Combine the attentions via concatenation, followed by a linear transformation to the desired output dimension d_o



[Vaswani et al., 2017]

Multi-Head Attention

- The definition is as follows:

$$\text{MultiHead}(\mathbf{Q},\mathbf{K},\mathbf{V}) := (\text{Attention}(\mathbf{Q}_1,\mathbf{K}_1,\mathbf{V}_1)\odot\cdots\odot\text{Attention}(\mathbf{Q}_h,\mathbf{K}_h,\mathbf{V}_h))\cdot\mathbf{W}^O$$

  where $\mathbf{Q}_i = \mathbf{Q}\cdot\mathbf{W}_i^Q$, and $\mathbf{K}_i = \mathbf{K}\cdot\mathbf{W}_i^K$, and $\mathbf{V}_i = \mathbf{V}\cdot\mathbf{W}_i^V$
- The dimensions of $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ may differ from d_k and d_v, say be d'_q, d'_k, and d'_v
- Hence: $\mathbf{W}_i^Q \in \mathbb{R}^{d'_q\times d_k}$ $\mathbf{W}_i^K \in \mathbb{R}^{d'_k\times d_k}$ $\mathbf{W}_i^V \in \mathbb{R}^{d'_v\times d_v}$ $\mathbf{W}^O \in \mathbb{R}^{h\cdot d_v\times d_o}$
- The matrix concatenation $\mathbf{A}\odot\mathbf{B}$ means that we simply place $\mathbf{B}$ to the right of $\mathbf{A}$ (i.e., row-wise concatenation)

Cross-Attention vs. Self-Attention

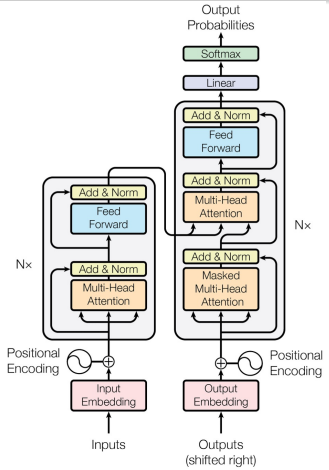
- *Where are* $\mathbf{Q}$, $\mathbf{K}$ *and* $\mathbf{V}$ *coming from?*
- Common scenario:
  - We have embeddings (representations) for the observers, represented by a matrix $\mathbf{X} \in \mathbb{R}^{n \times d_x}$, and embeddings for our context items, represented by $\mathbf{C} \in \mathbb{R}^{m \times d_c}$
    - These are predefined or determined by lower layers
  - Then, we apply weight matrices: $\mathbf{Q} = \mathbf{X} \cdot \mathbf{W}^Q \quad \mathbf{K} = \mathbf{C} \cdot \mathbf{W}^K \quad \mathbf{V} = \mathbf{C} \cdot \mathbf{W}^V$
- Two scenarios of how the attention mechanism is used in a bigger network:
  - Cross-attention: The observers and the context items are different creatures, as we assumed so far; in particular, $\mathbf{X}$ and $\mathbf{C}$ are not necessarily related
    - Example: textual question x about an image c
  - Self-attention: The observers and the context items are the same, and $\mathbf{X} = \mathbf{C}$
    - Example: language modeling, where each token observes the other tokens

Transformer Architectures

By a *transformer* we refer to an architecture with the following layers:

1. Input encoding
2. N layers of: multi-head attention followed by a feed-forward network, each includes a skip connection
3. A *decision layer* that typically consists of a simple NN layer followed by Softmax

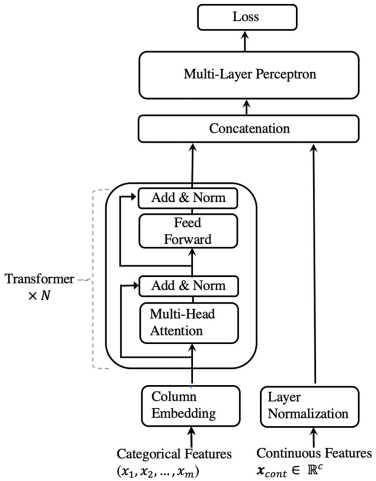
Example: The Original Text Transformer



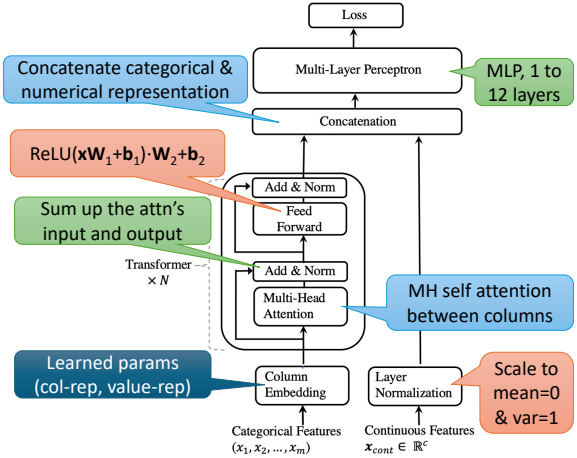
[Vaswani et al., 2017]

Example: TabTransformer

- TabTransformer [Huang et al., 2020] is a neural-network model for *tabular learning*
- Main contribution: transformer architecture for learning embeddings of categorical values
- A simple adaptation of the original language transformer to the categorical columns of a table



TabTransformer Components



Comparable to Boosting Trees

Dataset ds_name	N Datapoints	N Features	Positive Class%	Best Model
albert	425240	79	50.0	GBDT
hcdr_main	307511	120	8.1	GBDT
dota2games	92650	117	52.7	Logistic Regression
bank_marketing	45211	16	11.7	TabTransformer
adult	34190	25	85.4	GBDT
1995_income	32561	14	24.1	TabTransformer
online_shoppers	12330	17	15.5	GBDT
shrutime	10000	11	20.4	GBDT
blastchar	7043	20	26.5	GBDT
philippine	5832	309	50.0	TabTransformer
insurance_co	5822	85	6.0	TabTransformer
spambase	4601	57	39.4	GBDT
jasmine	2984	145	50.0	GBDT
seismicbumps	2583	18	6.6	GBDT
qsar_bio	1055	41	33.7	TabTransformer

XGBoost

Binary classification tasks

[Huang et al., 2020]

Outline

This Part

- We study the idea behind the two key concepts that are *enablers* for the *learning* part of deep learning:
  - Gradient computation via back-propagation
  - GPU-based computation and programming
- We will also learn the idea of *dropout*, where we change the model during training to avoid overfitting

Outline

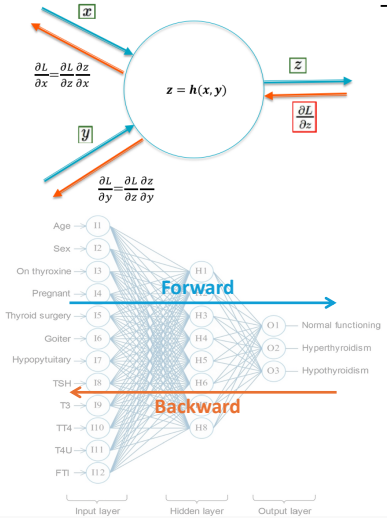
Reminder: Gradient Based Optimization

- Recall that we optimize our loss function using *steps of gradient-based methods*
- Different variations, all based on computing gradients of the coefficient within the loss function
  - Data usage:
    - Plain GD on all examples
    - SGD on one random example
    - Mini-batch SGD on a few random examples
    - Batch GD on a predefined partition (within an epoch)
  - Update method:
    - Plain GD (uniform reaction)
    - Adam optimizer (adaptive)
- This poses a challenge since each of the (many) batches requires the computation of the gradients on a large number of parameters
  - E.g., every entry of every matrix along the network

Backpropagation: Intuition

- *Backward propagation* (*backpropagation* for short) is an efficient technique for computing the gradients in two passes over the network, without back-and-forth communication between layers
- Each layer is responsible for:
  - Computing the derivatives of the parameters that it introduces
  - Deliver needed information to the layer underneath to continue on
- Great benefit as a programming paradigm – whenever a new unit is added to our ML library, we add its BP behavior, and that's it
- Note: **we focus our description to feed-forward networks**; *recurrent networks* require more subtle techniques, e.g., back propagation through time

BP: Forward and Backward Passes



The idea is to apply the *chain rule* for derivatives:

1. A *forward pass* evaluates all NN units in order
2. For the uppermost parameters, all we need is the result of the previous layer (given by the forward pass)
3. For parameters from lower levels, the gradient is a product of a constant computable in the uppermost layer, and derivatives from the lower levels
4. Hence, each layer passes the multiplier to the lower layer and lets it take over
5. Continue *backward* all the way to the input — each layer computes the derivatives of its parameters, updates the multiplier and propagates it back

Illustration 1: Simple Feedforward

- Consider an architecture comprising a chain of ℓ units, each with a single, distinct parameter:

$$\mathbf{h}^{(0)} := \mathbf{x} \qquad \mathbf{h}^{(1)} := f_1(\mathbf{h}^{(0)}; \theta_1) \quad \ldots \quad \mathbf{h}^{(\ell)} := f_\ell(\mathbf{h}^{(\ell-1)}; \theta_\ell) \qquad \hat{\mathbf{y}} := \mathbf{h}^{\ell}$$

- Suppose, for the sake of simplicity, that our loss function has the form $L(\boldsymbol{\theta}) := \text{Risk}(T, \hat{\mathbf{y}})$ for the training set T, or just $\text{Risk}(\hat{\mathbf{y}})$ for short
  - In particular, no regularization is involved
- At an iteration of GD, we need to compute the gradient $\nabla L(\boldsymbol{\theta})$
- **Forward step**: evaluate the units of the network:

$$\mathbf{x} = \mathbf{h}^{(0)} \longrightarrow \mathbf{h}^{(1)} \longrightarrow \cdots \longrightarrow \mathbf{h}^{(\ell)} = \hat{\mathbf{y}}$$

Illustration 1: Simple Feedforward: Backward Step

- For $i = 1, \ldots, \ell$, let $g_i = \nabla_{\theta_i} L(\boldsymbol{\theta}) = \dfrac{\partial L(\boldsymbol{\theta})}{\partial \theta_i}(\boldsymbol{\theta})$, which is a scalar
- Starting with layer ℓ and going backward to layer 1, each layer i needs to:
  1. Compute g_i
  2. Provide layer $i-1$ with the information needed to compute g_j for $j < i$

Derivative at the Last Layer

- For the gradient g_ℓ, we have $g_\ell = \nabla_{\theta_\ell} L(\boldsymbol{\theta}) = \nabla_{\theta_\ell} \text{Risk}(f_\ell(\mathbf{h}^{(\ell-1)}; \theta_\ell))$
- But now, $\mathbf{h}^{(\ell-1)}$ does not involve θ_ℓ, so it is simply *a constant that we computed in the forward step*, so we can compute $\nabla_{\theta_\ell} L(\boldsymbol{\theta})$ by standard differentiation
- Denote $\mathbf{v} = \mathbf{h}^{(\ell-1)}$
- For $j < \ell$, the chain rule gives us $g_j = \nabla_{\theta_j} L(\boldsymbol{\theta}) = \nabla_{\mathbf{v}} \text{Risk}(f_\ell(\mathbf{v}; \theta_\ell)) \cdot \nabla_{\theta_j} \mathbf{v}$
  - Note: $\nabla_{\mathbf{v}} \text{Risk}(f_\ell(\mathbf{v}; \theta_\ell))$ is a *row vector*, while $\nabla_{\theta_j} \mathbf{v}$ is a *column vector*
- We can compute $\mathbf{p} = \nabla_{\mathbf{v}} \text{Risk}(f_\ell(\mathbf{v}; \theta_\ell))$ without looking at the lower layers
- So, provide layer $\ell - 1$ with $\mathbf{p}$ and ask it to compute $g_j = \mathbf{p} \cdot \nabla_{\theta_j} \mathbf{v} = \mathbf{p} \cdot \nabla_{\theta_j} \mathbf{h}^{(\ell-1)}$ for layers $j < \ell$

Derivative at Lower Layers

- We are now at layer $i < \ell$, we got a row vector $\mathbf{p}$ from the higher level, and for $j \leq i$ we need to compute $g_j = \mathbf{p} \cdot \nabla_{\theta_j} \mathbf{h}^{(i)} = \mathbf{p} \cdot \nabla_{\theta_j} (f_i(\mathbf{v}; \theta_i))$ for $\mathbf{v} = \mathbf{h}^{(i-1)}$
- For $j = i$, we can compute $\nabla_{\theta_j} (f_i(\mathbf{v}; \theta_i))$ directly and we are done with g_i
  - Since $\mathbf{v}$ does not involve θ_i
- For $j < i$, the chain rule gives us $\nabla_{\theta_j} (f_i(\mathbf{v}; \theta_i)) = \nabla_{\mathbf{v}} f_i(\mathbf{v}; \theta_i) \cdot \nabla_{\theta_j} \mathbf{v}$
  - Note: $\nabla_{\mathbf{v}} f_i(\mathbf{v}; \theta_i)$ is a matrix (the Jacobian) with a row for each output item of f_i and a column for each element of $\mathbf{v}$
  - $\nabla_{\theta_j} \mathbf{v}$ is a column vector with a row for each entry of $\mathbf{v}$
- Again, we can compute $\nabla_{\mathbf{v}} f_i(\mathbf{v}; \theta_i)$ directly
- So, replace $\mathbf{p}$ with $\mathbf{p} \cdot \nabla_{\mathbf{v}} f_i(\mathbf{v}; \theta_i)$ and go on to layer $i - 1$
  - Note: the new $\mathbf{p}$ is again a row vector

Recalling our MLP Architecture

$$
\begin{aligned}
\mathbf{h}^{(0)} &:= \mathbf{x} \\
\mathbf{h}^{(1)} &:= \phi(\mathbf{A}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}) \\
&\vdots \\
\mathbf{h}^{(\ell)} &:= \phi(\mathbf{A}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}) \\
\mathbf{y} &:= \text{Softmax}(\mathbf{h}^{(\ell)})
\end{aligned}
$$

Illustration 2: Backward on MLP

- Denote $\mathbf{a}^{(i)} = \mathbf{A}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}$
  - So, $\mathbf{h}^{(i)} = \phi(\mathbf{a}^{(i)})$
- Assume that now we have a regularization term: $L(\boldsymbol{\theta}) = \text{Risk}(\hat{\mathbf{y}}) + \lambda\Omega(\boldsymbol{\theta})$
- For each layer i, we need to compute $g_i^{\mathbf{A}} := \nabla_{\mathbf{A}^{(i)}} L(\boldsymbol{\theta})$ and $g_i^{\mathbf{b}} := \nabla_{\mathbf{b}^{(i)}} L(\boldsymbol{\theta})$
- Using the chain rule,

$$g_i^{x} = \nabla_{\mathbf{h}^{(\ell)}} \text{Risk}\big(\text{Softmax}(\mathbf{h}^{(\ell)})\big) \cdot \nabla_{x^{(i)}}(\mathbf{h}^{(\ell)}) + \lambda \nabla_{x^{(i)}}(\Omega(\boldsymbol{\theta}))$$

- Each layer i can compute $\nabla_{x^{(i)}}(\Omega(\boldsymbol{\theta}))$ for its own parameters $\mathbf{A}^{(i)}$ and $\mathbf{b}^{(i)}$
- So the uppermost layer sends down $\mathbf{p} = \nabla_{\mathbf{h}^{(\ell)}} \text{Risk}(\text{Softmax}(\mathbf{h}^{(\ell)}))$
  - Which is now a matrix
- So, it remains for each layer i to compute $\nabla_{x^{(i)}}(\mathbf{h}^{(\ell)})$

Illustration 2: The Lower Layers

- Recall: $\mathbf{a}^{(i)} = \mathbf{A}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}$
- The chain rule gives: $\nabla_{x^{(j)}}(\mathbf{h}^{(i)}) = \nabla_{x^{(j)}}(\phi(\mathbf{a}^{(i)})) = \nabla_{\mathbf{a}^{(i)}}(\phi(\mathbf{a}^{(i)})) \cdot \nabla_{x^{(j)}}(\mathbf{a}^{(i)})$
- $\nabla_{\mathbf{a}^{(i)}}(\phi(\mathbf{a}^{(i)}))$ is computed directly based on the activation ϕ
- For $x^{(i)} = \mathbf{A}^{(i)}$ and $x^{(i)} = \mathbf{b}^{(i)}$, we can compute $\nabla_{x^{(i)}}(\mathbf{a}^{(i)})$ directly
  - For example, for $x^{(i)} = \mathbf{b}^{(i)}$, it is the identity matrix $\mathbf{I}$
- For $j < i$, we apply another chain: $\nabla_{x^{(j)}}(\mathbf{a}^{(i)}) = \nabla_{\mathbf{h}^{(i-1)}}(\mathbf{a}^{(i)}) \cdot \nabla_{x^{(j)}}(\mathbf{h}^{(i-1)})$
  - Note that $\nabla_{\mathbf{h}^{(i-1)}}(\mathbf{a}^{(i)})$ is simply $\mathbf{A}^{(i)}$
- So, we update $\mathbf{p}$ to $\mathbf{p} \cdot \nabla_{\mathbf{a}^{(i)}}(\phi(\mathbf{a}^{(i)})) \cdot \mathbf{A}^{(i)}$

Challenge with BP: Vanishing and Exploding Gradients

- BP multiplies gradients layer by layer through the chain rule, weights repeatedly *shrink* or *grow*, so gradients can:
  - Vanish: Gradients become extremely small $\Rightarrow$ earlier layers learn too slowly
  - Explode: Gradients become extremely large $\Rightarrow$ unstable updates, loss diverges
- These issues are severe in *deep* or *recurrent* networks, where many gradient multiplications occur
- Some mitigation strategies:
  - Careful *weight initialization*
    - E.g., Xavier initialization – weights drawn from a Gaussian distribution, variance determined by the input and output dimensions of the layer
  - *Normalization* scales layer/batch inputs to have zero mean and unit variance
  - *Skip connections* allow the output to accommodate the layer's input directly, bypassing the layer's transformation
  - *Gradient clipping* limits gradients to a maximum threshold

Outline

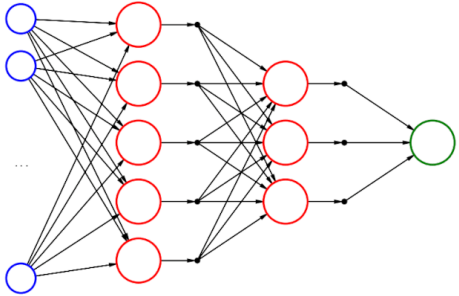
Dropout Regularization

- **Idea:** Randomly deactivate neurons during training
- Rationale: many epochs can lead to overfitting; mitigate by randomly deactivating neurons during training, encouraging robust learning that tolerates errors
- In practice, acts (and modeled) as an ordinary unit $d : \mathbb{R} \to \mathbb{R}$, except:
  1. Activated in each GD step randomly
  2. Ignored in the evaluation (test and validation) phases
- Associated with a hyperparameter p – the probability of dropout:

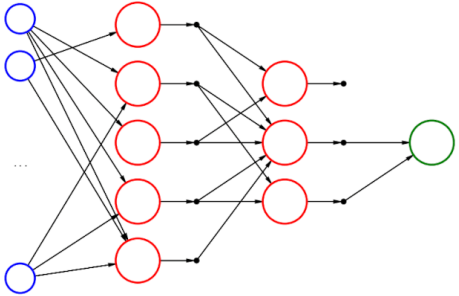
$$d(x) := x \cdot m/(1-p) \quad ; \quad m \sim \text{Bernoulli}(p)$$

  - Hence, x is zeroed out by the mask m with probability p
  - Division by $(1-p)$ retains the original expectation

Illustration (Training Phase)



Regular Connections



Applying Dropout

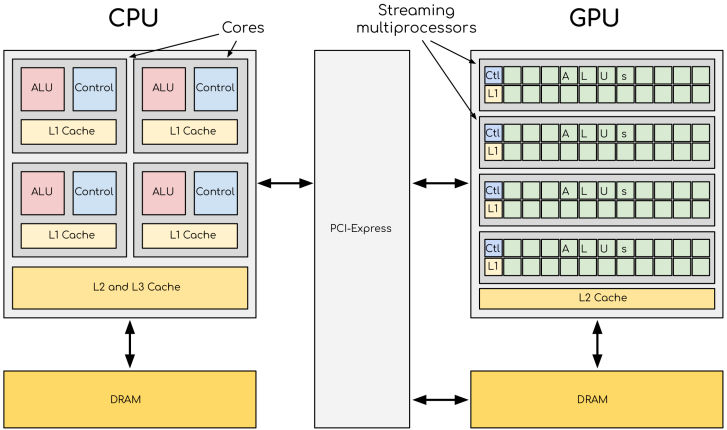
https://en.wikipedia.org/wiki/Dilution_%28neural_networks%29

Outline

GPU (Graphics Processing Unit)

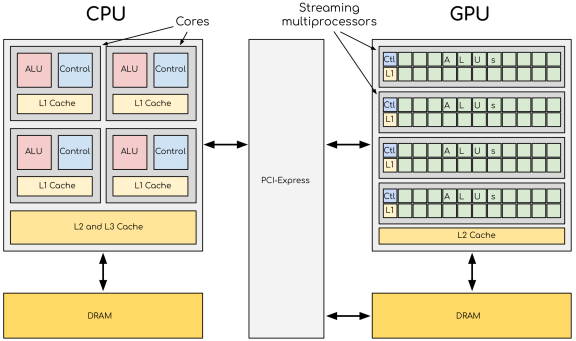
- A hardware processor originally built for rendering images, now used for general high-performance computing
- Key Idea: *parallelism*
  - A GPU has thousands of small cores that execute many simple operations at once — ideal for repetitive data-parallel tasks
- *CPU vs GPU*:
  - CPU: Few powerful cores, optimized for sequential tasks
  - GPU: Many lightweight cores, optimized for massive parallelism
- *Critical to deep learning*: GPUs accelerate both the forward pass and the backward pass (gradient computation) across many parameters and data samples
  - Started with *AlexNet* for image classification [Krizhevsky et al., 2012]
- Developers use frameworks like *CUDA* or *OpenCL* to program the distributed computation on the GPU

CPU-GPU Interaction Mode



https://enccs.github.io/gpu-programming/2-gpu-ecosystem/

GPU Enhancement

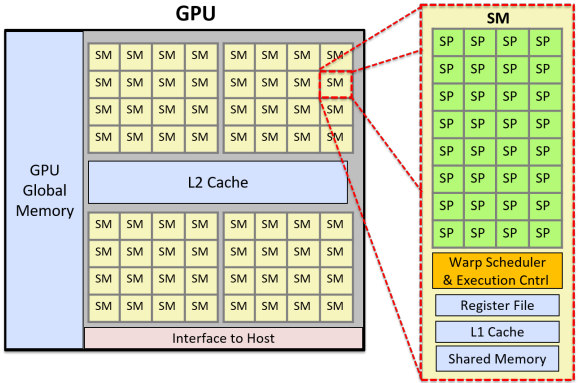


https://enccs.github.io/gpu-programming/2-gpu-ecosystem/

- The GPU lives *alongside* the CPU

- A single CPU can control *multiple* GPUs inside the device
- On the GPU memory, we execute programs ("*kernels*") launched by the CPU
- The CPU ("*host*") manages control flow, data movement, *synchronization* with the GPU ("*device*")
  - *Ensures* GPU computations finish before dependent CPU work continues
- CPU and GPU communicate over a high-speed interconnect (e.g., PCI Express, NVLink)

GPU Architecture

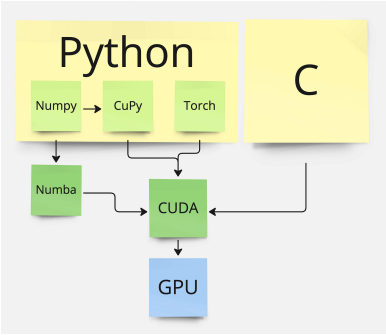


https://diveintosystems.org/book/C15-Parallel/gpu.html

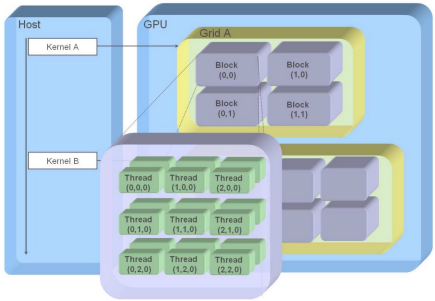
- A GPU has multiple *streaming multiprocessors* (SMs), each with its own *streaming processors* (computing cores, SP) ; for example:
  - **A100**: 108 SMs $\times$ 64 SPs
  - **RTX 4090**: 128 SMs $\times$ 128 SPs
- *Single instruction, multiple threads* (SIMT) model — threads grouped into warps (32 threads) that execute the same instruction on different data on a single SM

CUDA (Compute Unified Device Architecture)

- NVIDIA's computing platform for GPU programming
  - Extending C/C++
- Comes with *libraries*: **cuBLAS** (linear algebra), **cuDNN** (NN ML, e.g., BP), **cuFFT** (FFT), **Thrust** (STL-like)
- DL frameworks like *PyTorch*, *TensorFlow*, and *JAX* use CUDA, including cuBLAS and cuDNN, in their backend



CUDA Control over Threads



[Abdelkhalek et al., 2009]

- *Thread hierarchy*: **threads** $\rightarrow$ **blocks** $\rightarrow$ **grid**
- Each block runs multiple warps on a single SM
- `op<<<`b`,`t`>>>(args)` means: apply the kernel `op(args)` on a grid of b blocks, each running t threads
  - This launches $b \cdot t$ executions of the kernel `op(args)`, one on each thread
- Importantly, when executed, a kernel can *get its thread number* (we'll see how it's useful)

Important CUDA Instructions

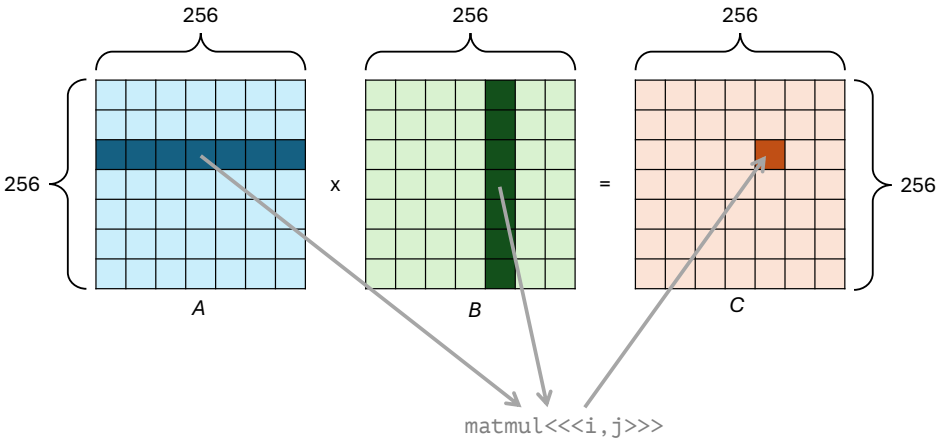
- Memory management
  - `cudaMalloc(ptr,size)` – allocate memory on the GPU
  - `cudaFree(ptr)` – release device memory
  - `cudaMemcpy(dst,src,size,direction)` – copy data between host and device
- Kernel launch
  - `kernel<<<b,t>>>(args)` – run a function on the GPU (we've seen it)
  - `__global__` – marks a function as a kernel callable from the host
- Synchronization
  - `__syncthreads()` – synchronize a block—wait for all current threads to finish
  - `cudaDeviceSynchronize()` – wait for all GPU tasks to finish

Example: Basic CUDA Program

```
__global__ void add(int *a, int *b, int *c) { // code per thread
    int i = threadIdx.x; // thread location
    c[i] = a[i] + b[i];
}
int main() {
    int N = 5, h_a[N] = {1, 2, 3, 4, 5}, h_b[N] = {5, 4, 3, 2, 1}, h_c[N];
    int *a, *b, *c;

    cudaMalloc(&a, N*sizeof(int)); // allocate on GPU
    cudaMalloc(&b, N*sizeof(int));
    cudaMalloc(&c, N*sizeof(int));
    cudaMemcpy(a, h_a, N*sizeof(int), cudaMemcpyHostToDevice); // copy to GPU
    cudaMemcpy(b, h_b, N*sizeof(int), cudaMemcpyHostToDevice);
    add<<<1, N>>>(a, b, c);   // kernel launch
    cudaMemcpy(h_c, c, N*sizeof(int), cudaMemcpyDeviceToHost); // copy to CPU
    cudaFree(a); cudaFree(b); cudaFree(c); // free allocated
}
```

Another Example: (256×256) Matrix Multiplication



C+CUDA Code

```
// A, B, C are 256x256 matrices in row-major order on the device.
__global__ void matmul(const float* A, const float* B, float* C) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;  // 0..255
    int col = blockIdx.x * blockDim.x + threadIdx.x;  // 0..255
    # Compute A.row * B.column
    float sum = 0.0f;
    for (int k = 0; k < 256; ++k)
        sum += A[row * 256 + k] * B[k * 256 + col];
    # Set the result's cell
    C[row * 256 + col] = sum;
}
// Host launcher
void launch(const float* A, const float* B, float* C) {
    dim3 block(16, 16);   // 256 threads per block
    dim3 grid(16, 16);    // 256 blocks = 256*256 threads total
    matmul<<<grid, block>>>(A, B, C);
}
```

Key Takeaways from Module 4

- A deep neural network is defined as a composition of common units (differentiable functions)
- Common composite networks are used as building blocks of bigger networks
- Important example: transformers built on attention
- Attention's idea: observer applies its *query* to the *keys* of context items to obtain a weighted average over a collection of the context *values*
- *Backward propagation* is a practical application for applying the chain rule for derivatives in gradient-based learning (e.g., SGD or Adam)
- Programs are compiled into parallelized hardware (e.g., GPU) using libraries (e.g., CUDA) for distributing the computation to organized processing units

The End

Many thanks for helping with the slides:

- **Shunit Agmon** (Technion)
- **Boaz Berger** (Technion)
- **Ilias Fountalis** (RelationalAI)

References I

Abdelkhalek, R., Calandra, H., Coulaud, O., Roman, J., and Latu, G. (2009).
Fast seismic modeling and reverse time migration on a gpu cluster.
In *2009 International Conference on High Performance Computing & Simulation*, pages 36–43. IEEE.

Bahdanau, D., Cho, K., and Bengio, Y. (2015).
Neural machine translation by jointly learning to align and translate.
In *ICLR*.

Cybenko, G. (1989).
Approximation by superpositions of a sigmoidal function.
Mathematics of control, signals and systems, 2(4):303–314.

Hosseinzadeh, M. et al. (2021).
A multiple multilayer perceptron neural network with an adaptive learning algorithm for thyroid disease diagnosis in the internet of medical things.
The Journal of Supercomputing, 77(4):3616–3637.

References II


Huang, X., Khetan, A., Cvitkovic, M., and Karnin, Z. S. (2020).
Tabtransformer: Tabular data modeling using contextual embeddings.
CoRR, abs/2012.06678.


Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012).
Imagenet classification with deep convolutional neural networks.
In *NIPS*, pages 1106–1114.


Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017).
The expressive power of neural networks: A view from the width.
Advances in neural information processing systems, 30.


Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986).
Learning representations by back-propagating errors.
nature, 323(6088):533–536.

References III

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017).

Attention is all you need.

In *NIPS*, pages 5998–6008.