Econ 425 Week 6

Neural networks

Grigory Franguridi

UCLA Econ

USC CESR

franguri@usc.edu

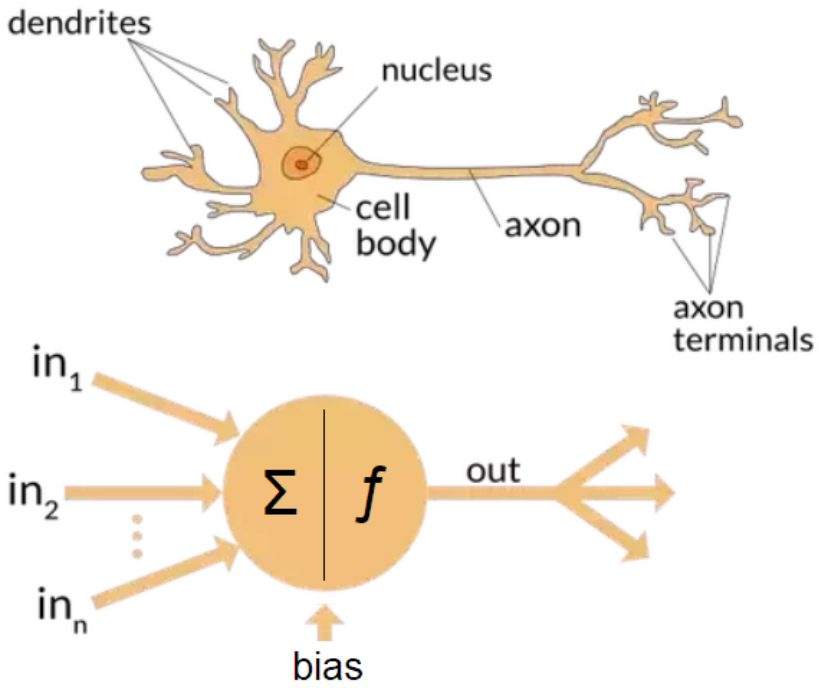
Limitations of linear models

- In earlier settings such as regression, we used a weighted linear combination $x'\beta$
- e.g., standard linear regression:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

- insufficient to capture more elaborate mapping between x and y, e.g.,
  - "any value in the range $[0; 5]$ is equally good"
  - "values over 8 are bad"
  - "values higher than 10 are not worse than values just over 8"
- requires a modeling beyond linear combinations – neural networks (NN)

Neurons



- **Activation:** an artificial neuron fires when its input signals reach a certain threshold, just like a biological one
- **Processing Non-linear Messages:** through activation, neurons can process complex nonlinear patterns and decision boundaries

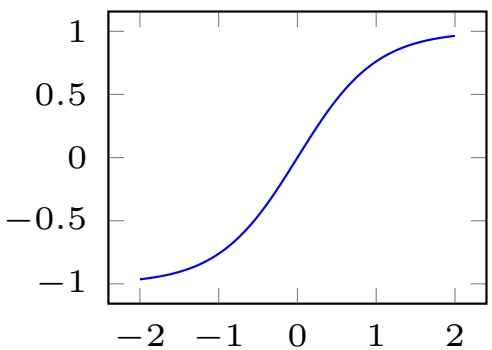
Activation functions

- control the activation pattern of neurons
- on top of the linear combination, a nonlinear function is added:

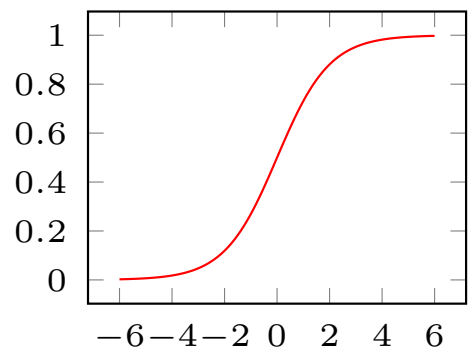
$$\hat{y} = \sigma\left(\beta_0 + \sum_{j=1}^{d} \beta_j x_j\right)$$

- Popular activation functions $\sigma(\cdot)$:

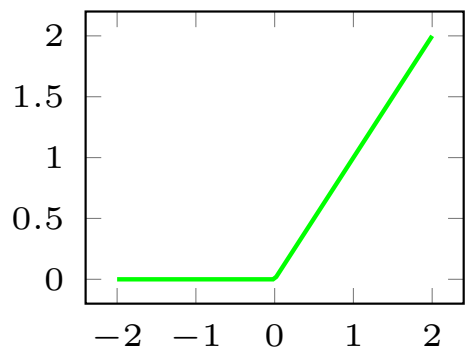
$\tanh(x)$



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



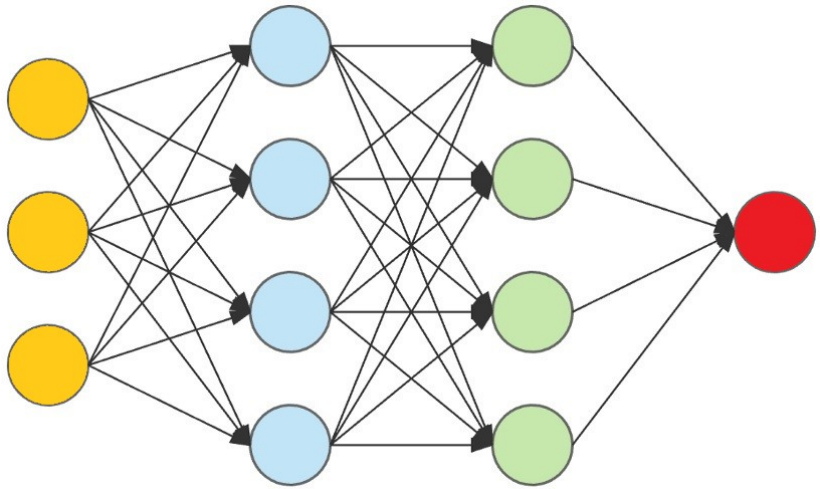
$ReLU(x) = \max(0, x)$



Why activation functions?

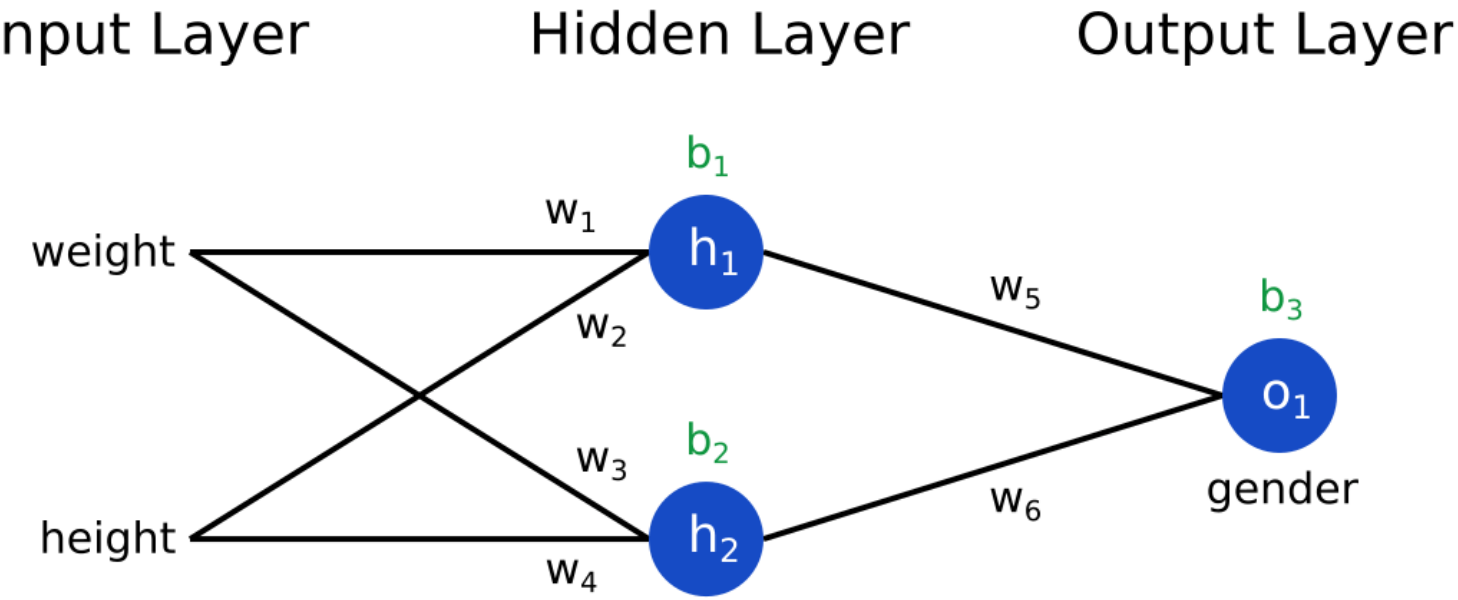
- **Nonlinearity:** enables NN to learn complex patterns beyond what is possible with linear models
- **Deep learning:** each layer in a NN acts as a distinct processing step, and having many such steps (deep NN) allows for the modeling of complex functions
- **Constrained range of values**: through activation, outputs of layers can be constrained, e.g., sigmoid limits outputs to $[0, 1]$, which can represent probability
- **No curse of dimensionality**

From single neurons to neural network



- **Building blocks:** each neuron (i) receives inputs, (ii) applies weights to inputs, and (iii) generates an output using an activation function
- **Layered structure:** neurons are organized into layers, where the outputs of previous layers are weighted and serve as the input to the next

From single neurons to neural network



$$y = \sigma(w_5 \cdot h_1 + w_6 \cdot h_2 + b_3), \text{ where}$$
$$h_1 = \sigma(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1),$$
$$h_2 = \sigma(w_3 \cdot x_1 + w_4 \cdot x_2 + b_2).$$

Here x are features, σ is activation function,
(w, b) are parameters (weights and biases)

From single neurons to neural network

Exercise: inputs $x_1 = 20$ and $x_2 = 3$, weights $w_1 = -0.2, w_2 = 1, w_3 = -0.5$, $w_4 = 0.3, w_5 = -1, w_6 = 0$, and biases $b_1 = 1, b_2 = -0.2, b_3 = 0.5$. Let σ be the ReLU function. Calculate hidden nodes h_1, h_2 and output y.

$h_1 = ReLU(20 * (-0.2) + 3 * 1 + 1) = 0$
$h_2 = ReLU(-0.5 * 20 + 0.3 * 3 - 0.2) = 0$
$y = ReLU(0 + 0 + 0.5) = 0.5.$

NN in matrix form

$$y = \sigma(\mathbf{W}_2\mathbf{h} + b_3),$$

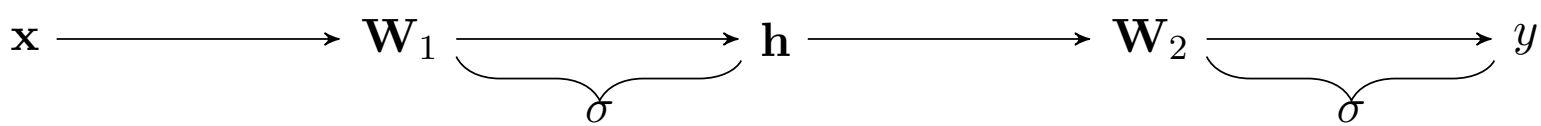
where

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b})$$

and

$$\mathbf{W}_1 = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}, \quad \mathbf{W}_2 = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Here $\mathbf{x}$ are input features, σ is activation function, $\mathbf{h}$ are hidden features and $\mathbf{b}$ and b_3 are biases



Loss function

Given data $\{x_i, y_i\}_{i=1}^n$, we can learn the weights w_j's by minimizing a loss function

Popular loss functions:

- **Mean Squared Error (MSE)**
  - $\frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - typically used for regression tasks, where the goal is to predict continuous values closest to the target value
- **Cross-Entropy**
  - $-\frac{1}{n}\sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$
  - used for classification tasks to measure the difference between probability distributions of output and target

(brief intro to double descent)

How to optimize a loss function?

- how to optimize the loss function in simple models?
- use gradient descent
- how to optimize the loss function in NN (composite function)?
- use gradient descent with backpropagation

Why backpropagation?

- **efficient learning**: efficiently computes the gradient of the loss function w.r.t. each weight in NN by using the chain rule, making it possible to update all weights in a direction that minimizes the loss
- review the chain rule
- **deep learning**: enables NNs to learn from complex data and perform tasks like image recognition, language translation (large language models), and playing games (reinforcement learning)
- **scalability**: scales well with large NNs and datasets, which is crucial for modern deep learning applications that involve vast amounts of data and complex models

Example: calculating gradient in NN

- Data:
  - Input: $x = 2$
  - Target: $y = 1$
- Network architecture:
  - Bias $b = 0$
  - Activation function: $\sigma(z) = \frac{1}{1+e^{-z}}$ (sigmoid)
  - Prediction: $\widehat{y} = \sigma(w \cdot x + b)$ (one hidden layer)
- Gradient descent:
  - Initial value for the weight $w = 0.5$
  - Learning rate $\alpha = 0.1$
- Loss Function: $L = \frac{1}{2}(y - \widehat{y})^2$ (MSE)

Example: calculating gradient in NN

Forward pass (calculating the loss):

1. Compute the weighted input:

$$z = x'w + b = 0.5 \cdot 2 + 0 = 1$$

2. Compute the output (prediction):

$$\widehat{y} = \sigma(x'w + b) = \frac{1}{1 + e^{-1}} = 0.731$$

3. Calculate the loss:

$$L = \frac{1}{2}(y - \widehat{y})^2 = 0.036$$

Example: calculating gradient in NN

Backward pass (calculating the gradient)

1. apply the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

2. calculate the components & the gradient:

$$\frac{\partial L}{\partial \widehat{y}} = -(y - \widehat{y}) = -0.269, \quad \frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y}) = 0.197,$$
$$\frac{\partial z}{\partial w} = x = 2, \qquad \frac{\partial L}{\partial w} = -0.269 \cdot 0.197 \cdot 2 = -0.106$$

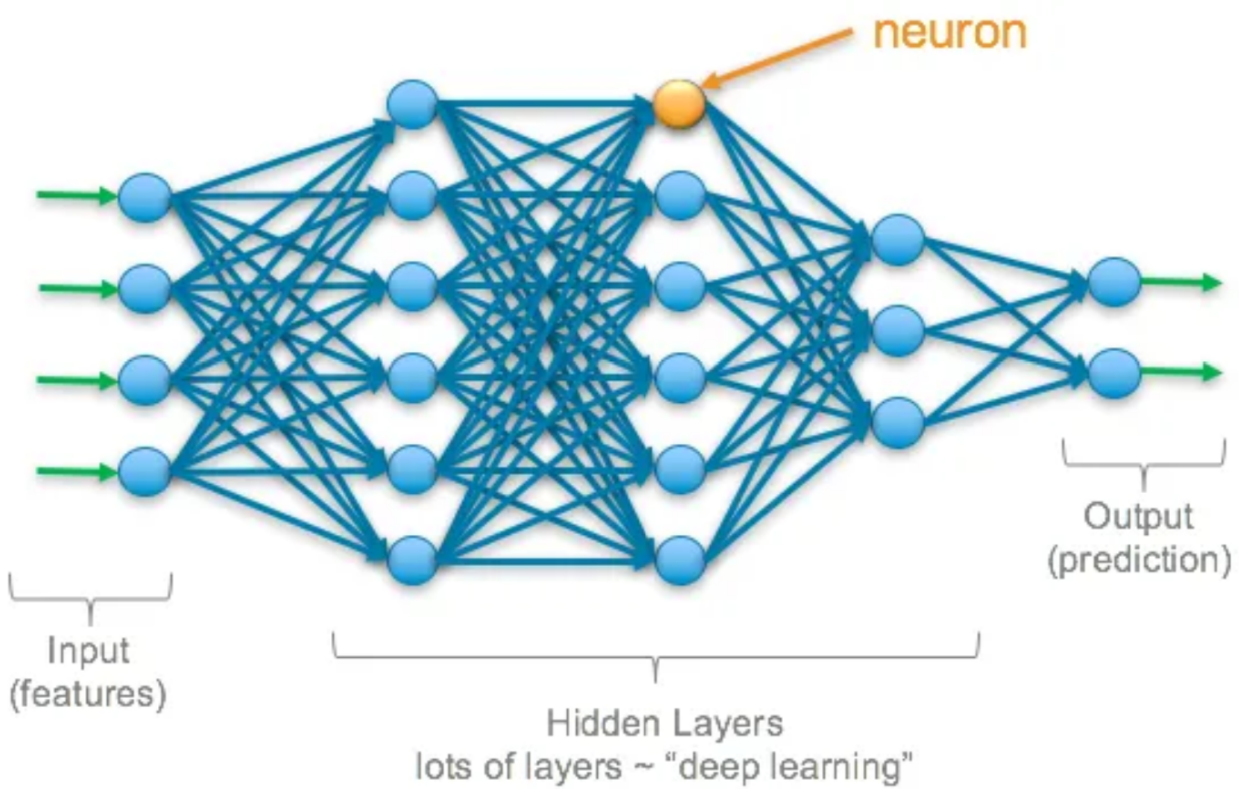
Weight update:

$$w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w} = 0.5 - 0.1 \cdot (-0.106) = 0.51$$

Deep neural networks

- contain many hidden layers (often hundreds), enabling them to learn features at multiple levels of abstraction
- able to handle very complex tasks
  - **natural language processing**: Google Translate
  - **text generation**: ChatGPT, Gemini
  - **playing strategic games**: AlphaGo, AlphaZero
  - **image recognition and generation**: Midjourney
  - economics, finance, computer vision, drug discovery etc
- our previous examples are 1-hidden layer (fully connected) NNs, i.e., *shallow* NNs

Layers in deep NNs



Layers in deep NNs

Input layer

- receives the raw data; each neuron in the input layer represents a feature
  - raw data: pixel in image data; token of text data; embedding of tabular data

Output layer

- the final layer that produces the output of the model. Design depends on specific task (regression/classification). For classification, the output layer often uses a softmax function to generate probabilities for each class

Layers in deep NNs

Hidden layers

- **dense (fully connected) layers**: every neuron in one layer is connected to every neuron in the next layer
- **convolutional layers**: apply the *convolution* operation to the input, capturing the spatial and temporal dependencies
  - mainly used in processing grid-like data such as images
- **pooling layers**: downsample the data by summarizing features in patches of the input data
  - used to reduce the dimensionality of the data, which helps decrease computational load and minimize overfitting
- **recurrent layers**: used for processing sequential data, where the output from previous steps is fed back into the model to predict the outcome of a sequence
  - e.g., in language modeling and time series analysis

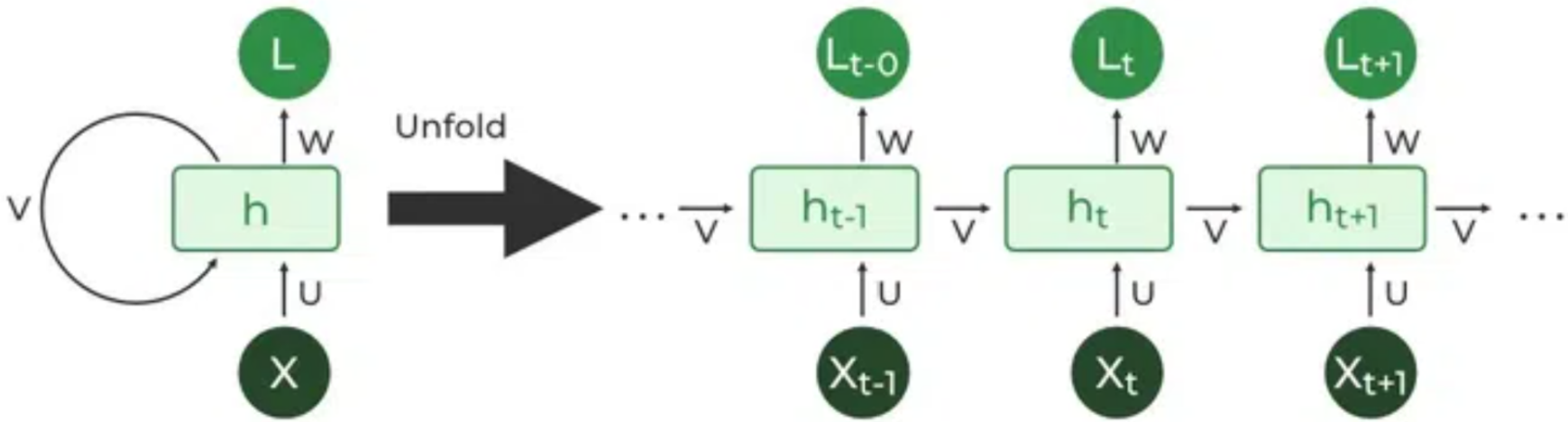
Drawbacks of feedforward NNs

- **sequence data processing**: feedforward NNs assume that inputs are independent of each other; however, in many applications like language translation or stock market prediction, the order of data points is crucial
- **variable-length input and output**: traditional NNs require fixed-size i/o; variable-length i/o is useful in applications like speech recognition, where the duration of input audio clips can vary
- one solution is recurrent NNs

Recurrent neural nets (RNN)

- has a **hidden state** (memory state), which remembers some information about a data sequence
- uses the same parameters (wights, biases) for each input
- $\Rightarrow$ performs the same task on all the inputs or hidden layers; this reduces the complexity of parameters, compared to other NNs

Schematic diagram of RNN



How do RNNs work?

- multiple fixed activation function units, one for each time step. Each unit has an internal state which is called its **hidden state**
- the hidden state reflects the past knowledge that RNN holds at a given time step; updated at every time step
- the hidden state is updated using the recurrence relation
  - $h_t = f(h_{t-1}, x_t)$
  - where h_t is the current state, h_{t-1} is the previous state, and x_t is the input state

Example: RNN with 1 hidden layer

- Hidden layer:
  - $h_t = \sigma(W_{hh} h_{t-1} + W_{xh} x_t)$,
    where W_{hh} is weight at recurrent neuron,
    W_{xh} is weight at input neuron
- Output:
  - $y_t = W_{hy} h_t$,
    where W_{hy} is weight at output layer
- parameters are updated using backpropagation

RNNs

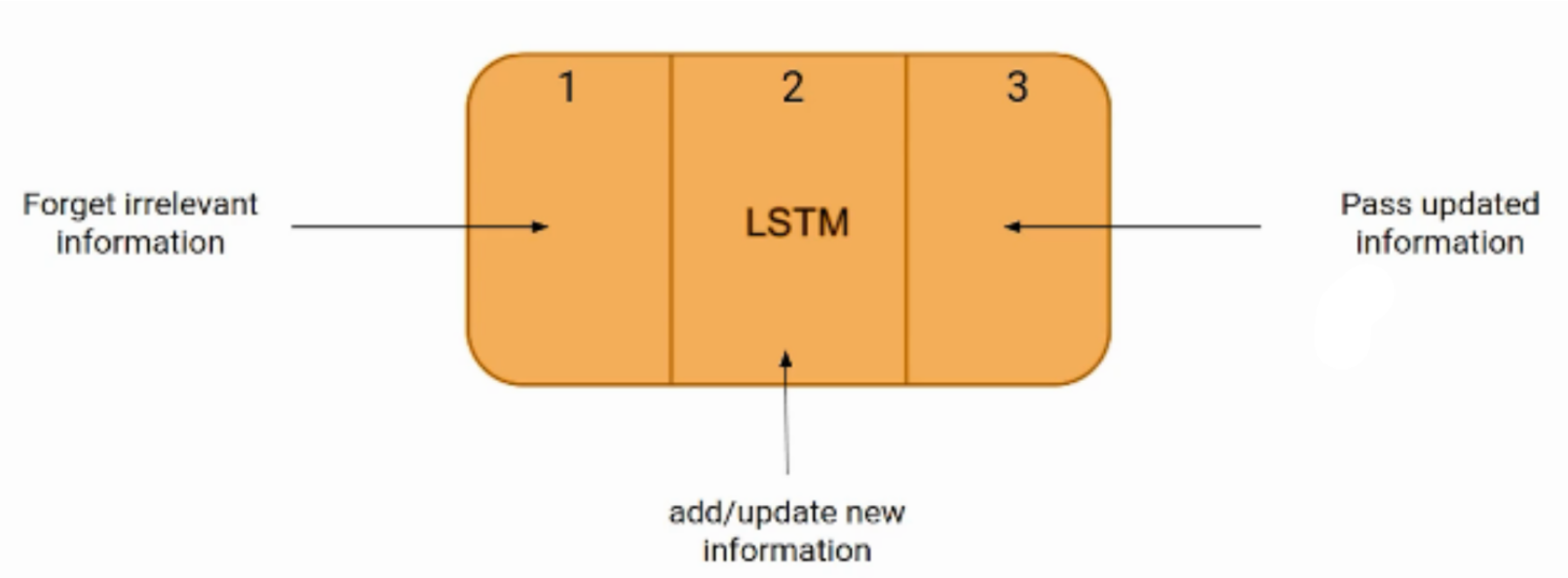
- **Advantages**:
  - remember all the information through time; useful in time-series prediction
  - can be used with convolutional layers (say, to extend the effective pixel neighborhood)
- **Disadvantages**
  - gradient can vanish/explode
  - training very difficult
  - cannot process very long sequences with $\tanh$ or ReLU activation functions
- **Solution: LSTM** (Long Short-Term Memory)

Long Short-Term Memory (LSTM) architecture

- LSTM is an RNN widely used in deep learning; excels at capturing long-term dependencies, making it ideal for sequence prediction tasks
- unlike traditional NNs, incorporates feedback connections, allowing it to process sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.

Long Short-Term Memory (LSTM) architecture

- similar to RNN cell
- three **gates**: forget, input, output (see below)

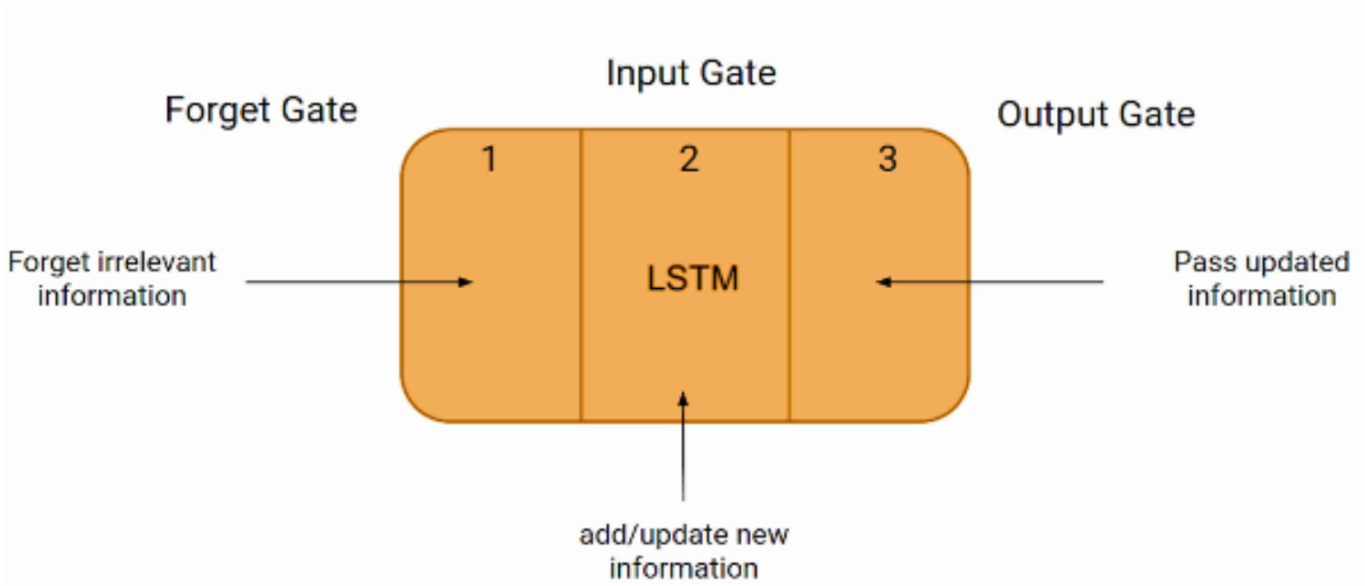


Logic of LSTM

LSTM cycle (single time step):

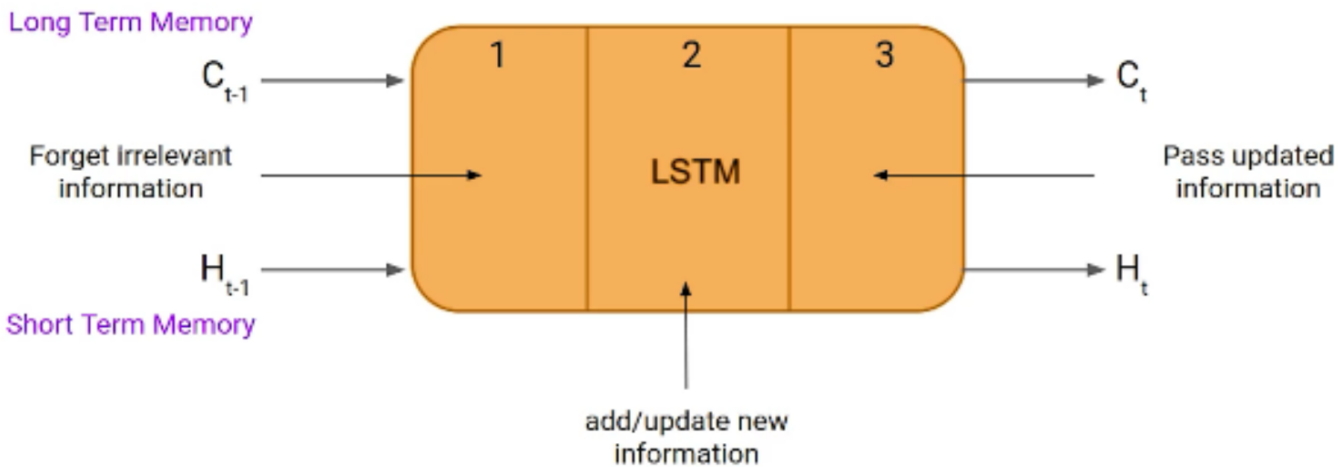
1. **(forget gate)** decides if the information from the previous timestamp is remembered or can be forgotten
2. **(input gate)** learns new information from the input
3. **(output gate)** passes the updated information from the current to the next timestamp

Logic of LSTM



Logic of LSTM

- **hidden state** H_t at time t (short-term memory)
- **cell state** C_t at time t (long-term memory)



LSTM: forget gate

- decides which information should be kept
- looks at the current input x_t and the previous hidden state h_{t-1}, and outputs a number in $[0, 1]$ for each number in the cell state C_{t-1}
  - 0 means "forget everything"
  - 1 means "retain everything"
- decision according to

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where W_f are weights, b_f is bias, and σ is sigmoid

LSTM: input gate

- decides what new information to store in the cell state
- steps:
  1. σ layer decides which values to update
  2. $\tanh$ layer creates a vector of new candidate values $\widetilde{C}_t$ that could be added to the state
- cell state update:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right),$$
$$\widetilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C),$$

where i_t is output of input gate, determining which values to update, and $\widetilde{C}_t$ is vector of candidate values

Transformer neural networks

- extremely popular for sequence data (much more than RNN/LSTM)
- used for machine translation, speech recognition, text generation (ChatGPT), image/pattern recognition, protein folding (Google AlphaFold 2), etc

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

"ImageNet Moment for Natural Language Processing"

Pretraining:
Download a lot of text from the internet

Train a giant Transformer model for language modeling

Finetuning:
Fine-tune the Transformer on your own NLP task

Image captioning with transformer NNs

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, \ldots, y_T$

Encoder: $\mathbf{c} = T_W(\mathbf{z})$
where **z** is spatial CNN features
$T_W(.)$ is the transformer encoder

Decoder: $y_t = T_D(\mathbf{y}_{0:t-1}, \mathbf{c})$
where $T_D(.)$ is the transformer decoder

CNN

Extract spatial features from a pretrained CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Features: H x W x D

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$ … $z_{2,2}$

Transformer encoder

$c_{0,0}$ $c_{0,1}$ $c_{0,2}$ … $c_{2,2}$

Transformer decoder

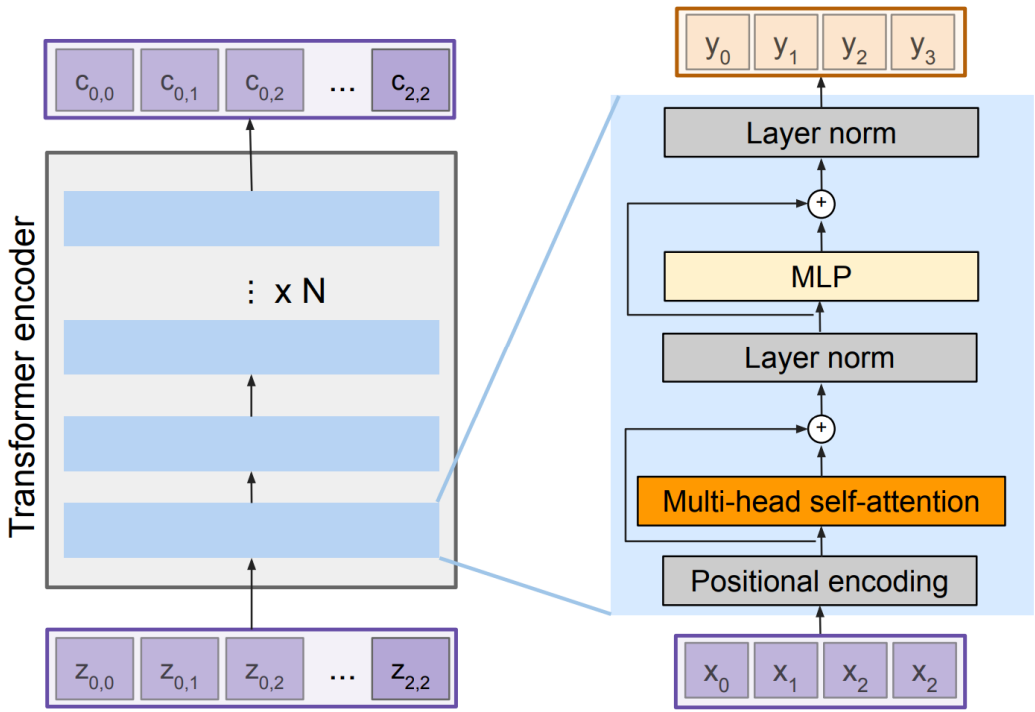
[START] person wearing hat

y_0 y_1 y_2 y_3

person wearing hat [END]

y_1 y_2 y_3 y_4

Transformer NNs: encoder block



Transformer Encoder Block:

Inputs: Set of vectors **x**
Outputs: Set of vectors **y**

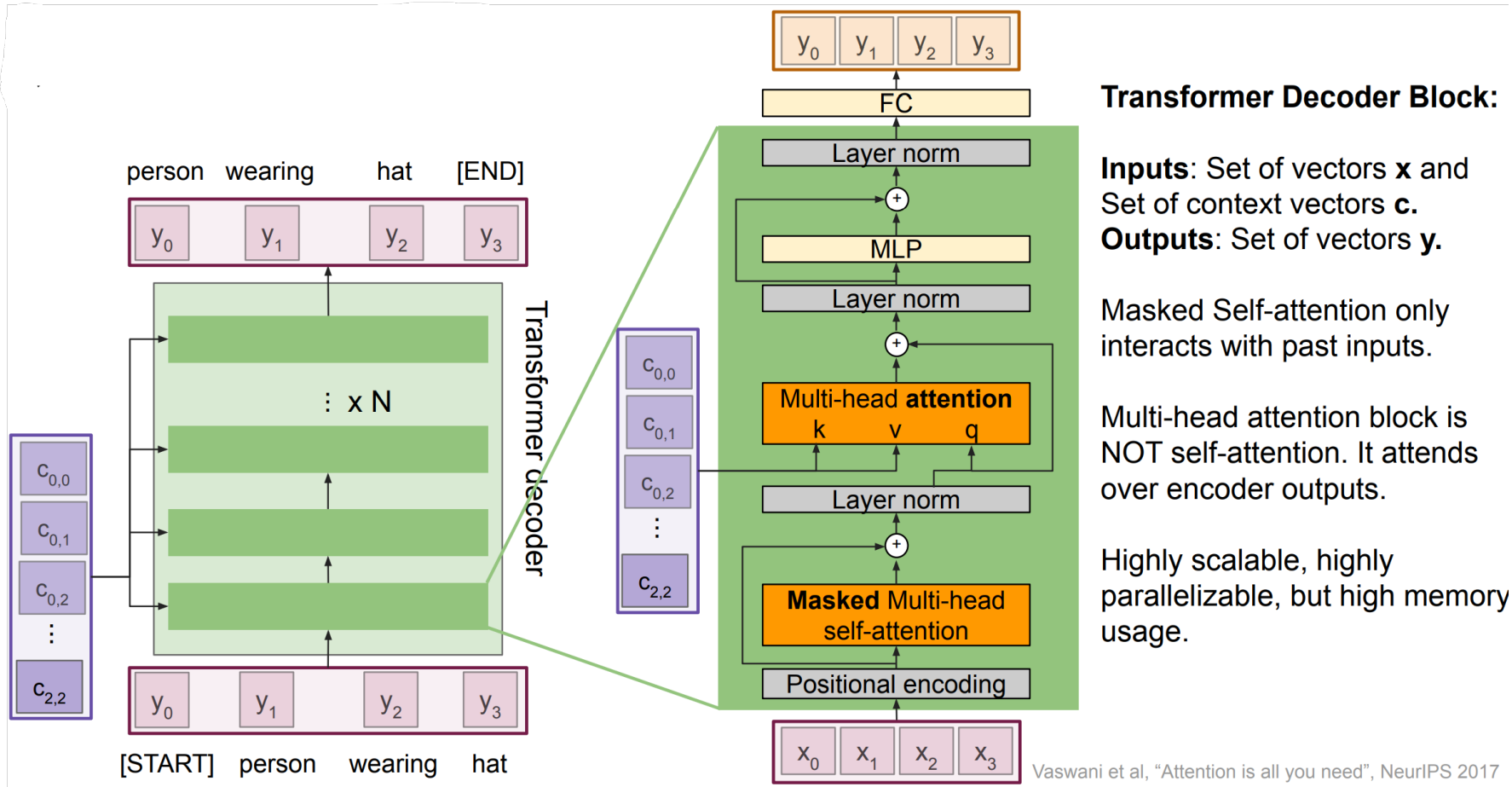
Self-attention is the only interaction between vectors.

Layer norm and MLP operate independently per vector.

Highly scalable, highly parallelizable, but high memory usage.

Vaswani et al, "Attention is all you need", NeurIPS 2017

Transformer NNs: decoder block



Transformer Decoder Block:

Inputs: Set of vectors **x** and Set of context vectors **c.**
Outputs: Set of vectors **y.**

Masked Self-attention only interacts with past inputs.

Multi-head attention block is NOT self-attention. It attends over encoder outputs.

Highly scalable, highly parallelizable, but high memory usage.

Vaswani et al, "Attention is all you need", NeurIPS 2017

Image captioning with transformers ONLY

pixels to language: