Neural Networks for Sequence Data

CMPUT 261: Introduction to Artificial Intelligence

GBC §10.0-10.2
P §12.1-12.2, §12.4, §12.6

Lecture Outline

1. Recap & Logistics
2. Unfolding Computations
3. Recurrent Neural Networks
4. Attention & Transformers

After this lecture, you should be able to:

- explain the problems with handling sequence input using dense or convolutional neural networks
- explain the high-level idea behind neural networks and transformers
- describe how self-attention combines inputs to generate its outputs
- describe the architecture of a transformer layer
- explain the high-level idea behind encoder-decoder architectures

Logistics

- **Assignment #3** is available
  - Due **Thursday, Nov 20**
  - Submit via Canvas (deadline is firm)
  - Blank submissions **will not be regraded**, nor granted EA
- Next week is **reading week**
  - No lectures, no labs

Recap: Convolutional Neural Networks

- Convolutional networks: Specialized architecture for **images**
- Number of **parameters** controlled by using **convolutions** and **pooling** operations instead of **dense connections**
- Fewer parameters means more **efficient to train**

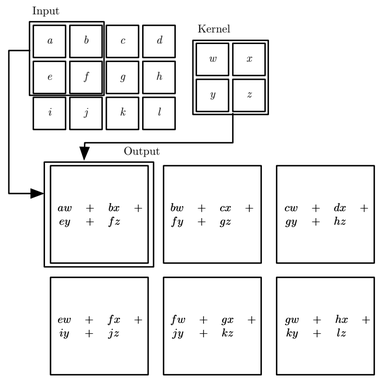


figure depicting convolution

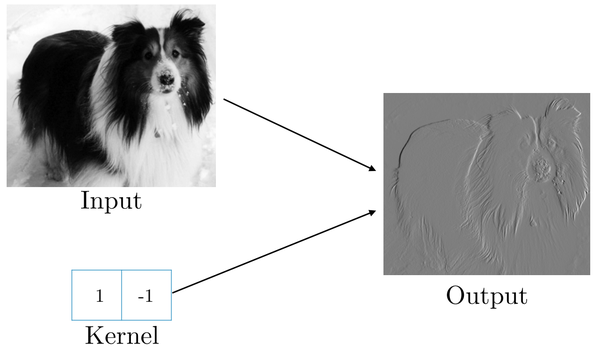


figure depicting example of edge-detection using convolution

(Images: Goodfellow 2016)

Sequence Modelling

- For many tasks, especially involving language, we want to model the behaviour of **sequences**
- **Example:** Translation
  - `The cat is on the carpet` $\Longrightarrow$ `Le chat est sur le tapis`
- **Example:** Sentiment analysis
  - `This pie is great` $\Longrightarrow$ POSITIVE
  - `This pie is okay, not great` $\Longrightarrow$ NEUTRAL
  - `This pie is not okay` $\Longrightarrow$ NEGATIVE
- **Example:** Text generation
  - `CMPUT 261 is the` $\Longrightarrow$ `CMPUT 261 is the best class ever`

Sequential Inputs

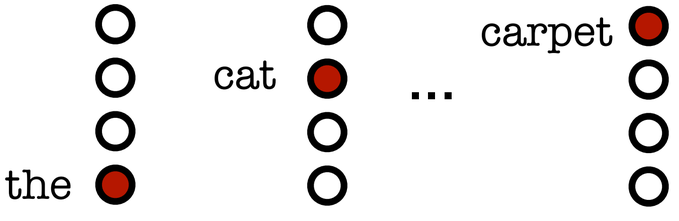
The cat is on the carpet

Question: How should we **represent** sequential input to a neural network?

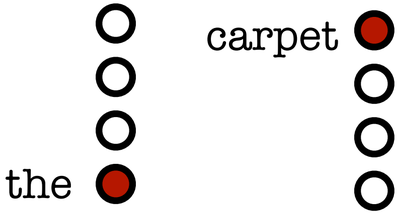
1. 1-hot vector for **each word**
   (Sequence must be a specific length?)

2. 1-hot vector for **last few words**
   (n-gram)

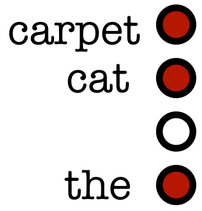
3. **Single vector** indicating each word that is present
   (bag of words)



Three columns of circles, each circle except one is empty in each column. In first column, bottom circle is filled in and labelled "the". In second column, second circle is filled in and labelled "cat". Ellipses between second and third columns. In last column, top circle is filled in and labelled "carpet".



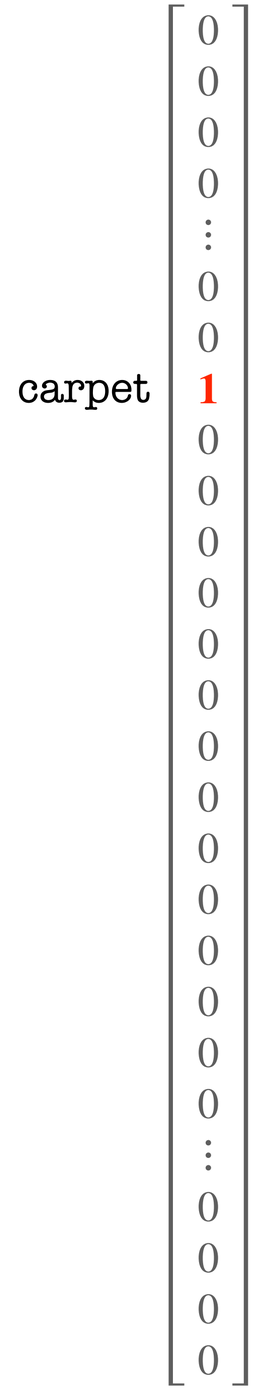
Two columns of circles, each circle except one is empty in each column. In first column, bottom circle is filled in and labelled "the". In second column, top circle is filled in and labelled "carpet".



A single columns of circles. Top circle is filled in and labelled "carpet". Second circle is filled in and labelled "cat". Third circle is empty. Bottom circle is filled in and labelled "the".

One-Hot Representations

carpet

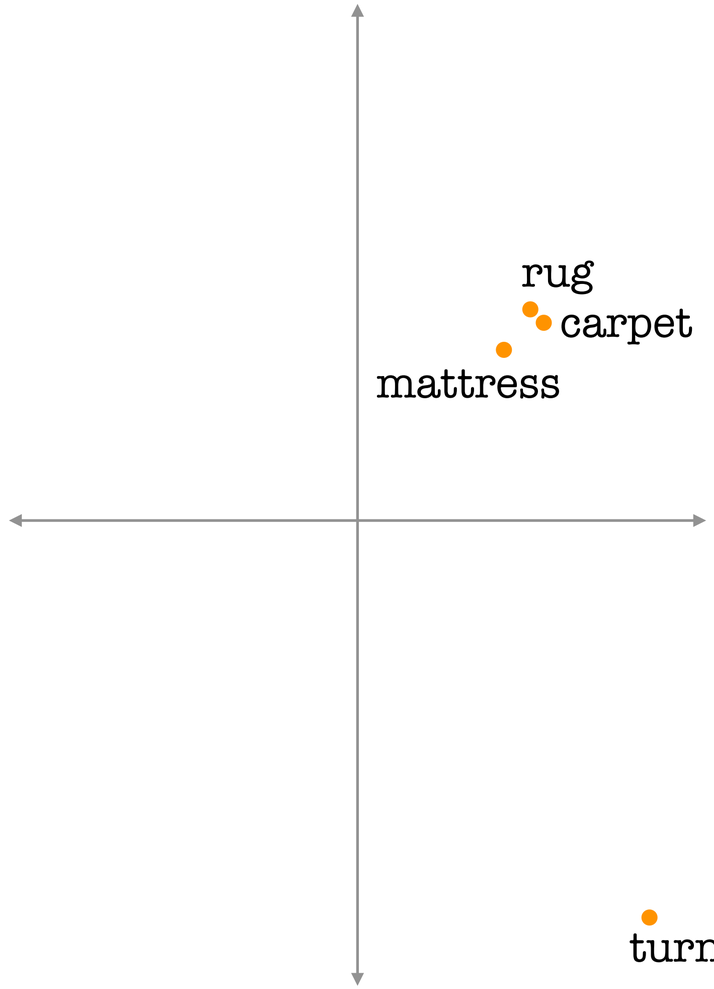


One-hot representations of words have some problems:

1. **Wasteful:** Each input vector must have a dimension equal to the size of the vocabulary (possible words)
    - If vocabulary has 30,000 words, then each vector has 29,999 zeros
2. **Poor generalization:** Ideally, similar words would be treated similarly
    - Exploiting meaningful similarity between images was an important feature of convolutional neural networks

Semantic Embeddings

- The usual approach is to first learn a **semantic embedding** for one-hot vectors
- Every word gets represented as a **dense vector** with smaller dimension than the vocabulary (typical size: 1,024)
- **Goal:** Words with similar **meanings** will have small **distance** between embedded vectors; words with different meanings will have large **distance** between embedded vectors



A graph with four quadrants. Top right quadrant has three dots near to each other labelled "rug", "carpet", and "mattress". Bottom right has a single dot, far from the top three, labelled "turnip".

(Pre-)Training Semantic Embeddings

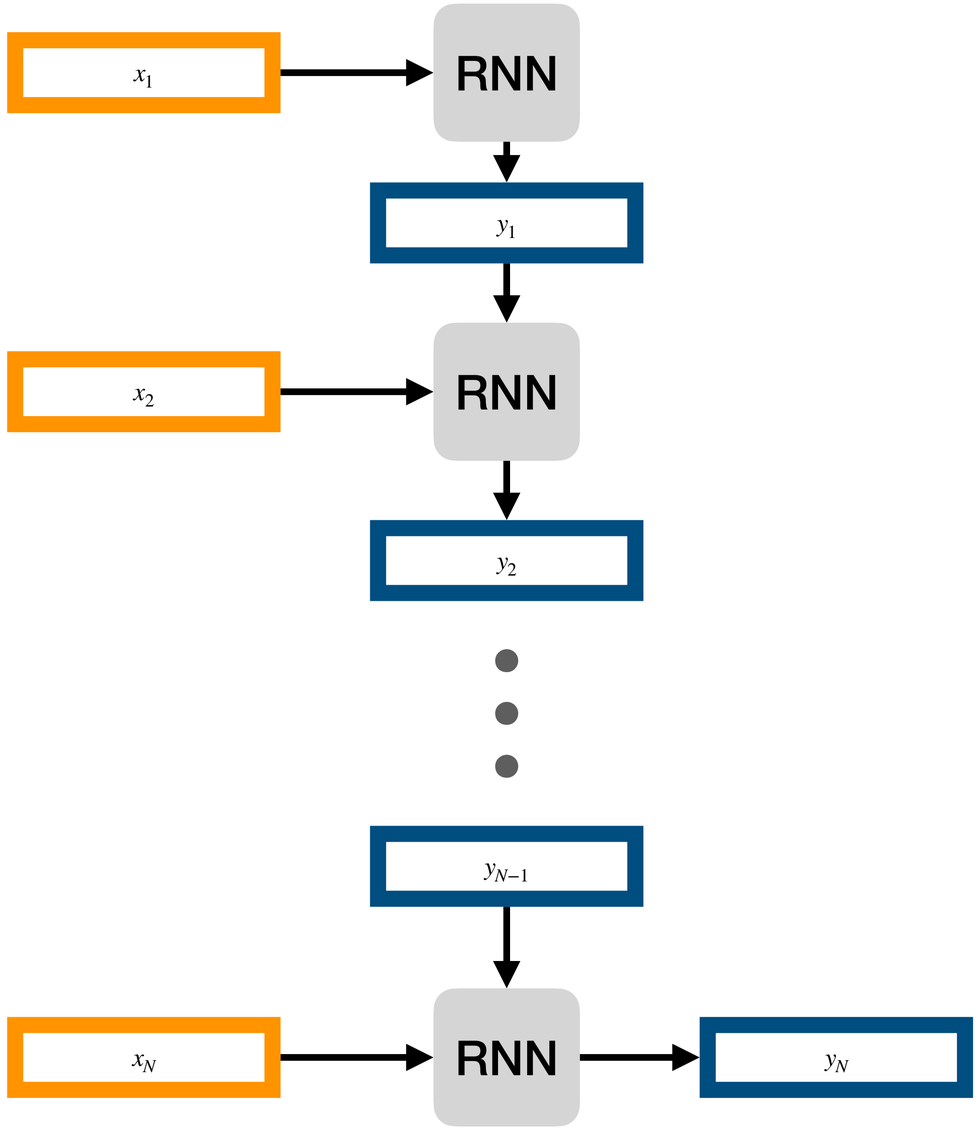
Question: How many **parameters** are required to convert a one-hot encoding for vocabulary of V words into a D-dimensional embedding?

- Embeddings require the training of ***many*** parameters
- Fortunately, this can be done with **unlabeled** data
- **Trick:** "Pre-train" neural network for a task that we don't care about
  - But which can be evaluated using unlabeled data
  - Predicting words from k nearby words
  - Predicting "masked" words
- Keep the weights that convert the one-hot layer into a dense embedding layer
- Throw away the weights that convert the embedding layer into output

Processing Variable-Length Sequences

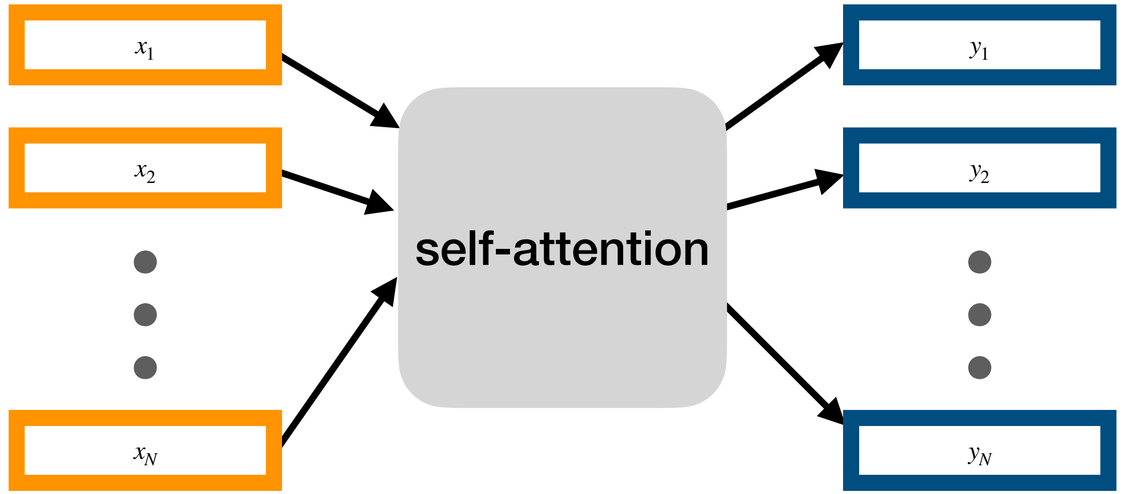
- Image inputs can be restricted to a **standard size** (20x20, 1024x768, etc.)
- Sequence inputs (e.g., text) are **variable-length**
  - And often **very long**
- **Solution:** Apply the **same operations** to each position in the sequence
- Two such approaches:
  1. **Recurrent neural networks:**
     input is current token + fixed-dimension "state" from previous operation
  2. **Transformers / self-attention:** Size of state varies with size of sequence

Self-Attention vs. RNN



A graph depicting the operation of an RNN. A box labelled "x1" has an arrow going into a box labelled "RNN". From RNN, an arrow points to a box labelled "y1". From y1, an arrow points to another box labelled "RNN", which also has an arrow from a box labelled "x2". After ellipses, boxes labelled "y N-1" and "xN" point to a box labelled "RNN", which points to a box labelled "yN".

This lecture



A graph depicting the operation of a self-attention unit. Boxes labelled "x1", "x2", ..., "xN" all point to a box labelled "self-attention", which points to boxes labelled "y1", "y2", ..., "yN".

- **RNN:** accept "previous" state and current input; output "next" state
  - Final output is last state
- **Self-attention:** Accept ALL inputs
  - Final output is ALL states

Self-Attention

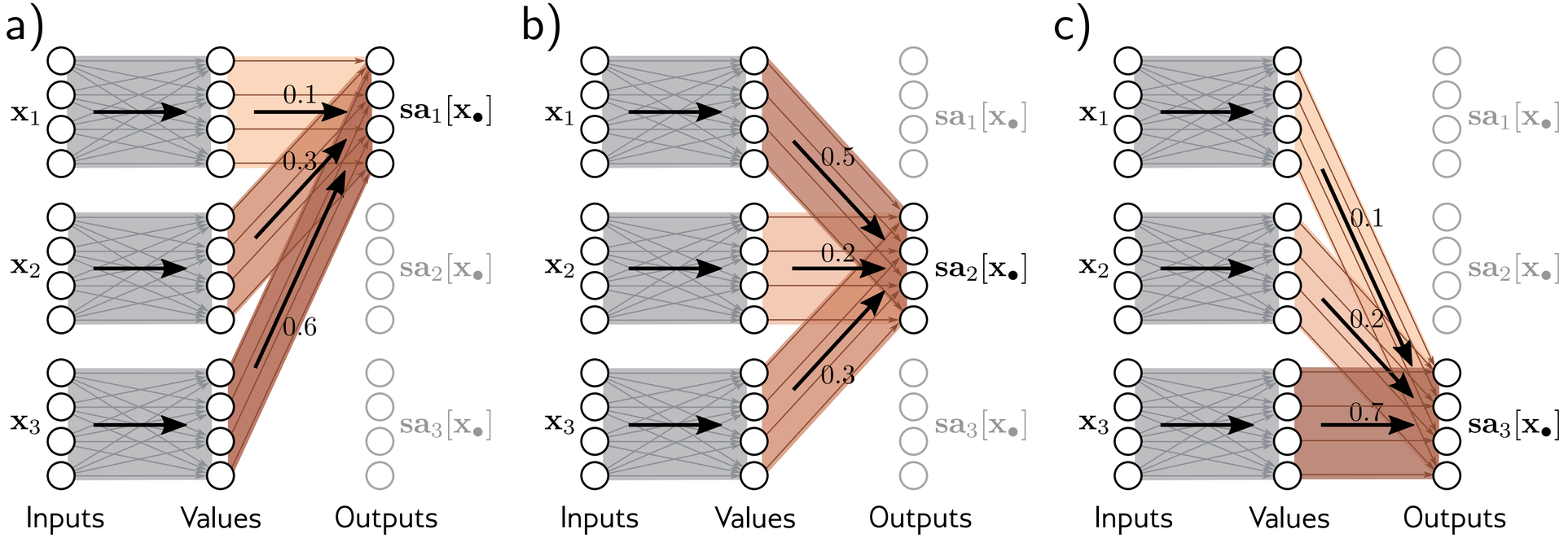
- Each input is transformed into a single output
  - N inputs means N outputs
- An output is computed by:
  1. Each **input** x_i transformed into **value** v_i by a **linear** operation

$$v_i = \beta_v + \Omega x_i$$

  2. Each **output** y_j is a **weighted combination** of the **values**:

$$y_j = \sum_i a[x_i, x_j] v_i$$

Self-Attention:
Outputs are Weighted Sums of Values



Three figures depicting attention weighting. In each figure, there is a column of three groups of four dots labelled "Inputs". The groups are labelled "x1", "x2", and "x3. Next to that is a column of three groups of four dots labelled "Values". Next to that is a column of three groups of four dots labelled "Outputs". The output groups are labelled "sa1[x]", "sa2[x]", and "sa3[x]". In each figure, there is a large arrow connecting one input group to one value group, with faint edges connecting each input dot in the group to all of the dots in the value group. In figure (a), there is an arrow labelled "0.1" connecting the value group for x1 to sa1[x], an arrow labelled "0.3" connecting the value group for x2 to sa1[x], and an arrow labelled "0.6" connecting the value group for x3 to sa1[x]. The top dot of the each value group has a faint edge connecting it to the top dot of the sa1[x] group; the second dot of each value group has a faint edge connecting it to the second dot of each value group; and so forth. In figure (b), the arrows got from each value group to sa2[x], and are labelled "0.5", "0.2", and "0.3", with faint arrows from top dot of each value dot to top dot of sa2[x] and so forth. In figure (b), the arrows got from each value group to sa3[x], and are labelled "0.1", "0.2", and "0.7", with faint arrows from top dot of each value dot to top dot of sa3[x] and so forth.

1. x_1 transformed into v_1 by linear operation $(\beta_v + \Omega_v x_1)$
   - x_2 transformed into v_2 by the **same** linear operation $(\beta_v + \Omega_v x_2)$, ...
2. y_1 is a **weighted average** of v_1, v_2, v_3
   - The weights are the attention values $a[x_1, x_1], a[x_2, x_1], a[x_3, x_1]$

(Image: Prince 2022)

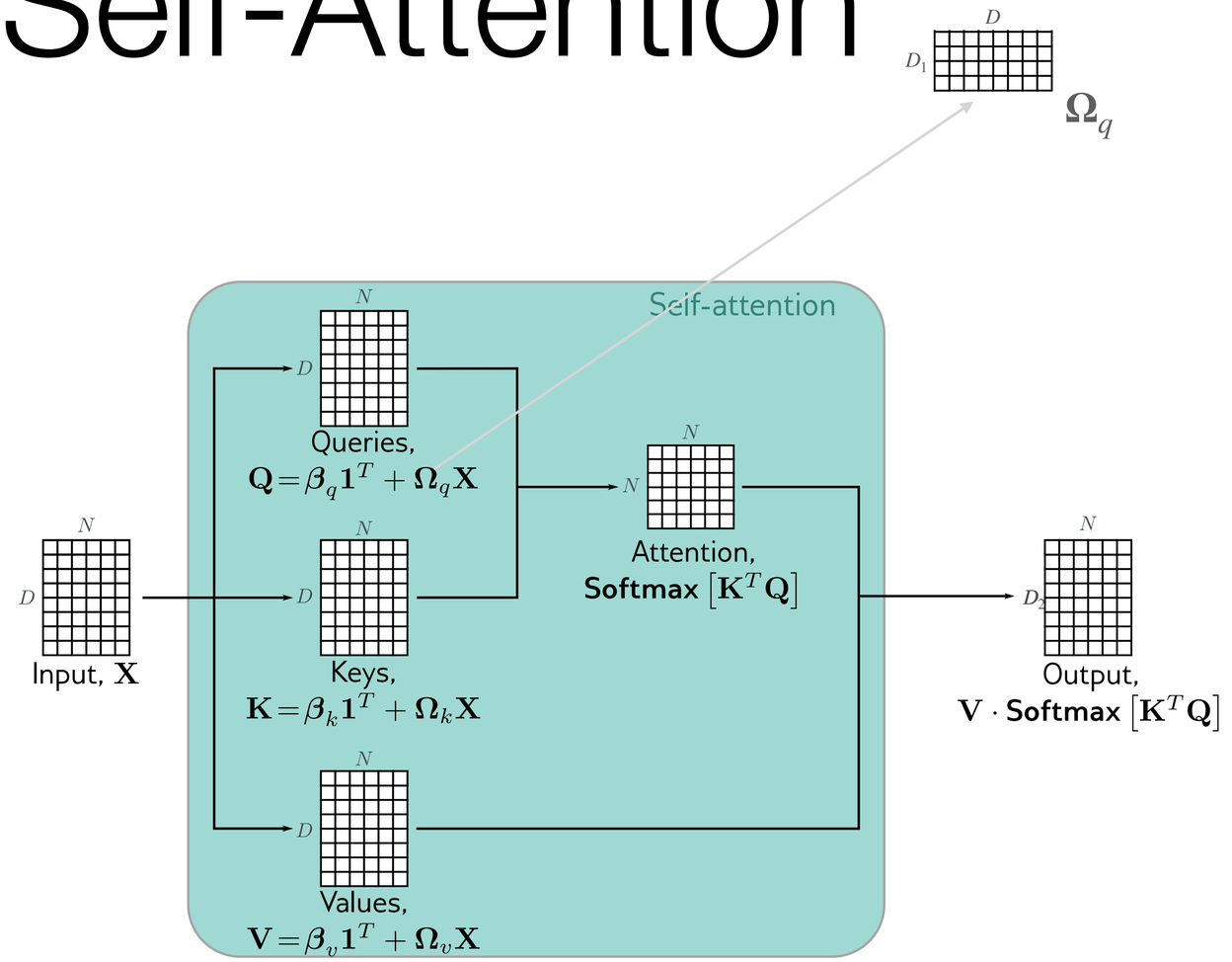
Dot-Product Self-Attention

- A **self-attention unit** computes **three values** for each input x_i:
  - **Query** q_i, **Key** k_i, and **Value** v_i
  - These values are computed in the **same way** for each input
- Each output is a **weighted** combination of the **values** of **all** inputs:

$$y_j = \sum_i a[x_i, x_j] v_i = \sum_i w_{ij} v_i$$

- Weight for output y_j of **value** x_i is proportional to the dot-product of j's **query** and i's **key**

$$w_{ij} = \frac{\exp(q_j^\top k_i)}{\sum_\ell \exp(q_j^\top k_\ell)}$$



A DxN matrix labelled "Input X" has arrows going to three additional matrices: "Queries", a D1xN matrix labelled with the equation $Q = \beta_q 1^\top + \Omega_q X$; "Keys", a D1xN matrix labelled with the equation $K = \beta_k 1^\top + \Omega_k X$; and "Values", a D2xN matrix labelled with the equation $V = \beta_v 1^\top + \Omega_v X$. Queries and Keys have arrows leading to an NxN matrix "Attention", labelled with the equation $\mathrm{softmax}[K^\top \circ_p Q]$. Attention and Values have arrows pointing to a D2xN matrix "Output" labelled with the equation $V \cdot \mathrm{softmax}[K^\top \circ_p Q]$. Ω_q is a D1xD matrix. There is a shaded box labelled "Self-attention" that contains all elements except for Input on the left and Output on the right.

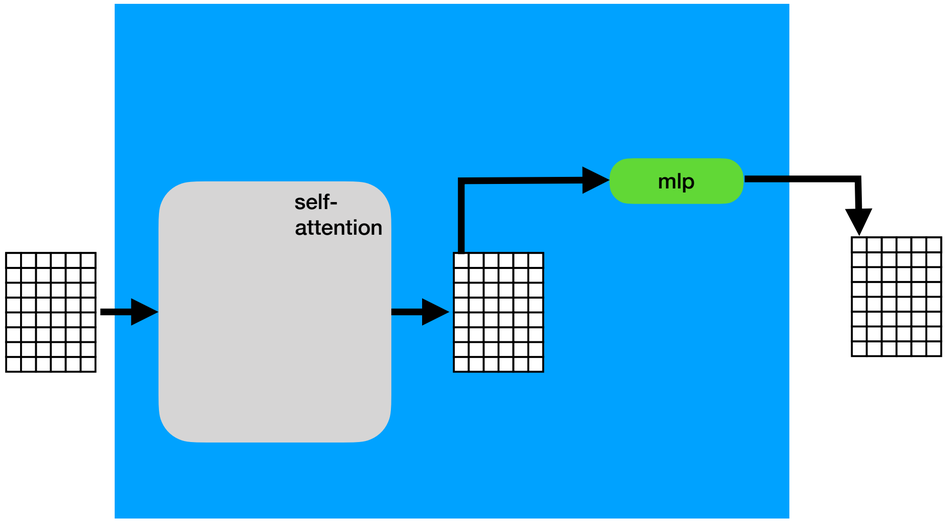
(Image: Prince 2022)

Transformer Blocks

- A **transformer layer** is a **self-attention** unit followed by a **dense feedforward network**
- The **same** feedforward network gets applied to each output of the self-attention unit:

$$y_j = \text{mlp}(x_j; \Omega) \quad \text{for } j = 1, \ldots, N$$

- In a typical transformer architecture, several transformer blocks will be strung together in parallel ("multiple heads")



A matrix of squares has an arrow pointing to a block labelled "self-attention", which has an arrow pointing to another matrix of squares. An arrow goes from the leftmost vector of the second matrix to the leftmost vector of a third vector via an oval labelled "mlp". A shaded box encloses self-attention, the second matrix, and mlp; the first and third matrices are outside the shaded box.

Training a Transformer Network (for encoding tasks)

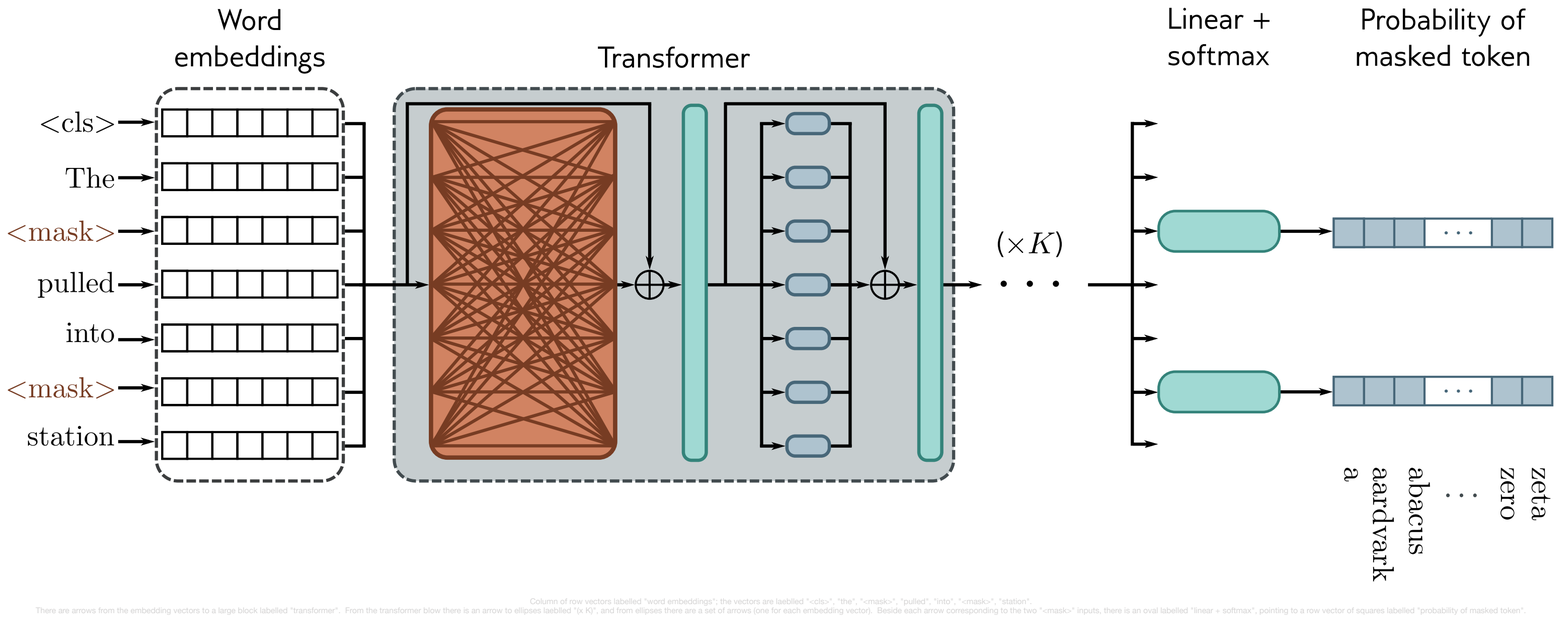
Transformers are trained in two phases:

1. **Semi-supervised pre-training:** Using a very large dataset, train the network to perform task for which dataset implies the answer
   - we **need not label** the examples manually
   - e.g., predicting masked words

Question: Is this important? Why?

2. **Fully supervised fine-tuning:**
   Add another layer or two at the end, and train for the real task using manually labelled examples
   - e.g., sentiment analysis, word classification, text span prediction

Pre-training

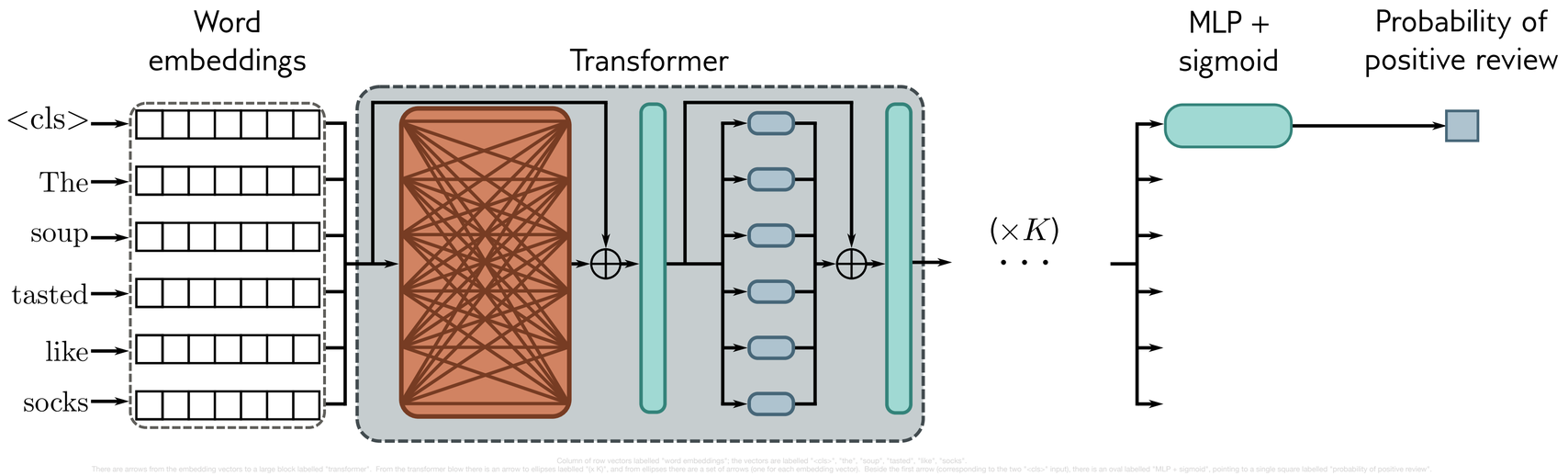


Column of row vectors labelled "word embeddings"; the vectors are labelled "<cls>", "the", "<mask>", "pulled", "into", "<mask>", "station".
There are arrows from the embedding vectors to a large block labelled "transformer". From the transformer blow there is an arrow to ellipses labelled "(x K)", and from ellipses there are a set of arrows (one for each embedding vector). Beside each arrow corresponding to the two "<mask>" inputs, there is an oval labelled "linear + softmax", pointing to a row vector of squares labelled "probability of masked token".

- Subset of tokens in an example sequence are **masked** (replaced with a special token)
- Neural network applied to each masked output predicts probabilities for missing token
  - Loss is back-propagated through entire network
- At end of training, that neural network is **thrown away**

(Image: Prince 2022)

Fine-tuning: BERT for Sentiment classification

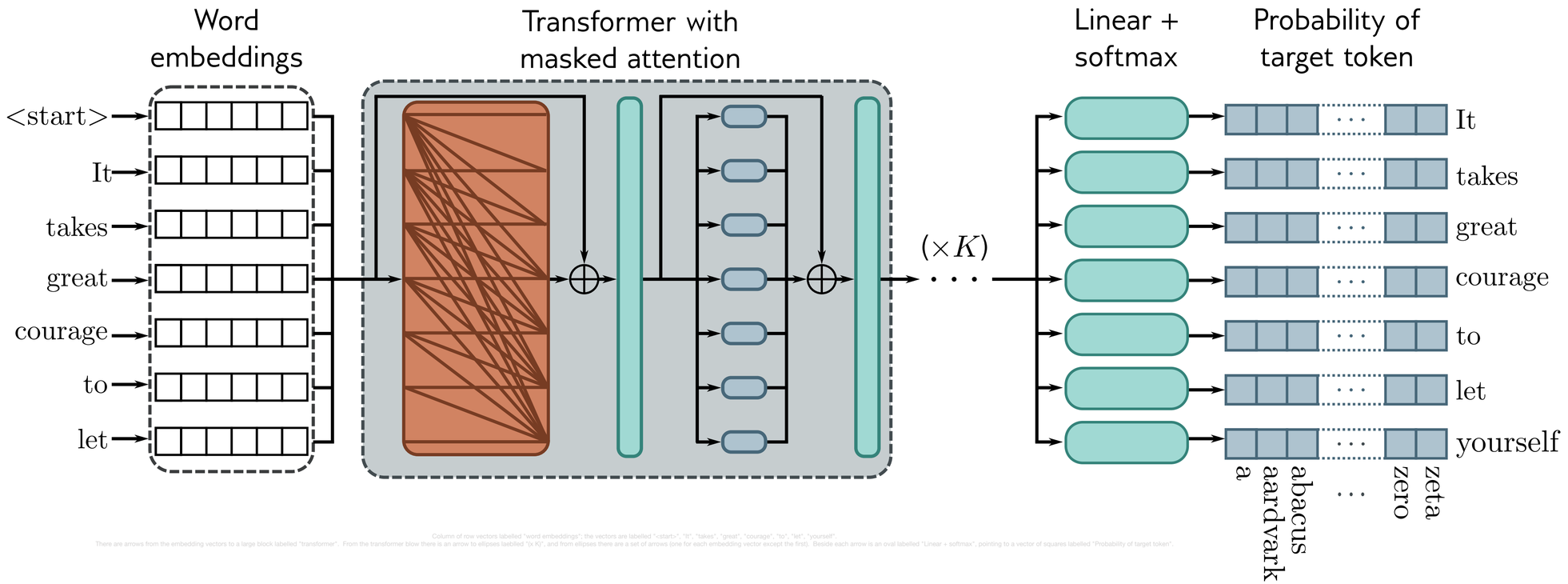


Column of row vectors labelled "word embeddings"; the vectors are labelled "<cls>", "the", "soup", "tasted", "like", "socks".
There are arrows from the embedding vectors to a large block labelled "transformer". From the transformer blow there is an arrow to ellipses laebled "(x K)", and from ellipses there are a set of arrows (one for each embedding vector). Beside the first arrow (corresponding to the two "<cls>" input), there is an oval labelled "MLP + sigmoid", pointing to a single square labelled "probability of positive review".

- **Sentiment classification:** Predict if a sequence is positive or negative
- First token of sequence is always a special "classify" token
- Neural network trained on corresponding output token using labelled dataset

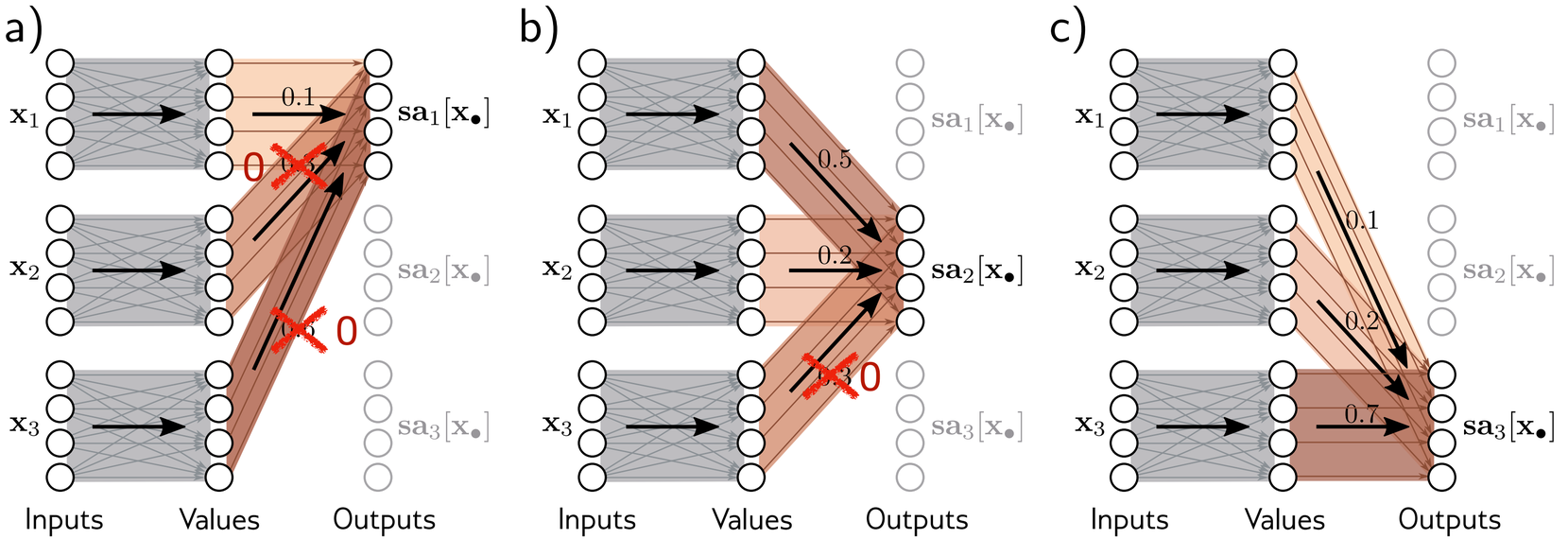
(Image: Prince 2022)

GPT3: Autoregressive Text



- Produces probability distribution for each output
- **Goal:** Maximize probability of each token given only the previous tokens
- Accomplished with **masked self-attention**: all attention values to subsequent tokens are forced to zero

GPT3: Masked Self-Attention



Three figures depicting attention weighting. In each figure, there is a column of three groups of four dots labelled "Inputs". The groups are labelled "x1", "x2", and "x3. Next to that is a column of three groups of four dots labelled "Values". Next to that is a column of three groups of four dots labelled "Outputs". The output groups are labelled "sa1[x]", "sa2[x]", and "sa3[x]". In each figure, there is a large arrow connecting one input group to one value group, with faint edges connecting each input dot in the group to all of the dots in the value group. In figure (a), there is an arrow labelled "0.1" connecting the value group for x1 to sa1[x], an arrow labelled with a bold red "X" connecting the value group for x2 to sa1[x], and an arrow labelled with a bold red "X" connecting the value group for x3 to sa1[x]. The top dot of the each value group has a faint edge connecting it to the top dot of the sa1[x] group; the second dot of each value group has a faint edge connecting it to the second dot of each value group; and so forth. In figure (b), the arrows got from each value group to sa2[x], and are labelled "0.5", "0.2", and bold red "X", with faint arrows from top dot of each value dot to top dot of sa2[x] and so forth. In figure (b), the arrows got from each value group to sa3[x], and are labelled "0.1", "0.2", and "0.7", with faint arrows from top dot of each value dot to top dot of sa3[x] and so forth.

- When computing attention values, use 0 for all weights that point to later tokens
- This prevents the model from "cheating" by guessing next token given the later tokens

Summary

- Naïvely representing **sequential inputs** for a neural network requires infeasibly many input nodes (and hence **parameters**)
- One-hot encodings are wastefully large and have no semantic structure
  - **Embeddings** solve these problems and can be trained without explicit labels
- **Recurrent neural networks** are a **specialized architecture** for sequential inputs
  - **State** accumulates across input elements
  - Each stage computed from **previous stage** using **same parameters**
- **Transformers** are another specialized architecture!
  - **Self-attention** to combine inputs instead of accumulating state
  - All states output (not just last in a sequence)
  - Improved ability to attend to long-range dependence
  - Pre-training followed by fine-tuning
  - Classification: attend to the output corresponding to a special "classify" input token
  - Text generation: probabilities of each word given preceding words; required **masked attention**