# Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Spring 2024

# Contents

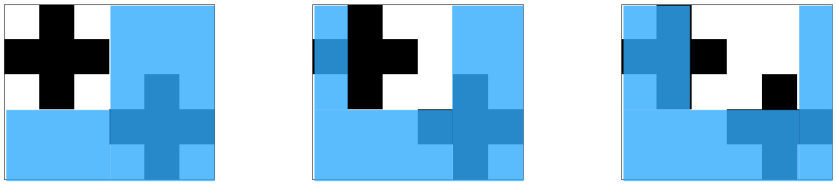
# CHAPTER 11

# Transformers

### 11.0.1 Review: CNNs, RNNs

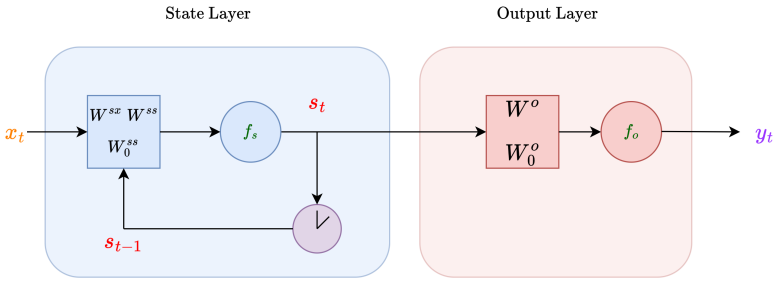
In the last two chapters, we built up two types of "neural networks":

- **Convolutional Neural Networks (CNNs)** understand data in **space**.



CNNs look at small region of data, searching for patterns in space.

- **Reccurent Neural Networks (RNNs)** understand data over **time**.




RNNs store data in a "state", allowing us to save it over time.

Both of these models use a kind of "locality": data which is nearby, is **related**.

**Concept 1**

RNNs and CNNs both use **locality**:

- In a CNN, **nearby** data is used to search for patterns.
- In an RNN, **recent** data has more effect on the state.

This allows us to use smaller, **simpler** models:

- Rather than thinking about every possible connection between data, we only connect "nearby" data. Thus, we need **fewer** parameters.

### 11.0.2 The problem with locality

This presents one major weakness, that we've ignored so far:

- If we focus on information that is **nearby**, we're missing out on information that's **far away**.
- We need a way to encode "distance" of information, that doesn't ignore/forget the "distant" info.

**Concept 2**

If information is spread over **long distances**, our RNN/CNN models won't capture it.

- If a pattern is **too big** for our CNN filter, we'll have more trouble finding it.
- The **longer** we run our RNN, the less our state usually remembers about the **distant** past.

**Example:** Consider the following sentence:

- **The sweater** that I found in the back of my old closet, which I hadn't opened since we moved into the house several years ago, **still fits me perfectly.**

Note that the beginning and the end of this sentence are linked as a single idea: "The sweater still fits me perfectly".

- But there's a **huge gap** between these phrases: it might be difficult for an RNN to

remember "**the sweater**", by the end of sentence.

- This also comes up in longer passages: in a paragraph, the first sentence might create **context** for the last sentence.

Imagine trying to feed a book into an RNN... it would eventually completely forget the beginning of the book!

### 11.0.3 Transformers

To solve this problem, and many others, we'll introduce a new model, called a **transformer**.

- Transformers allow us to create long-distance connections between data, using the mechanism of **attention**.

**Clarification 3**

In this chapter, we'll use **transformers** to **process language**.

- But the same tools can be applied to **many other problems**: image and audio processing, robotics, etc.

We'll develop this model in several steps:

- First (11.1), we'll convert words into vectors. One-hot encoding is too simple, so we'll use a different approach: **vector embeddings**.
- Next (11.2), we'll figure out which words in a passage are **relevant**(or connected) to each other, using a clever system called **attention**.
- Finally (11.3), we'll put together these ideas to create a complete model, known as a **transformer**.

## 11.1 Vector embeddings and tokens

### 11.1.1 One-hot encoding isn't enough

First, we want to turn words into something computable, like a **vector**.

It's difficult to try to do math on the word "cheddar". It's not numerical.

The simplest approach would be **one-hot encoding**.

- **Example:** Suppose that we want to classify **furniture** as table, bed, couch, or chair.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \tag{11.1}$$

- For each class:

$$v_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad v_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad v_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad v_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{11.2}$$

This approach is simple, but often, it's *too* simple.

**Concept 4**

**One-hot encoding** loses a lot of information about the objects it's representing.

- It's hard to say which words are "**similar**" to each other, for example.

**Example:** You probably associate the word "sugar" with "sweet", and "salt" with "savory".

- But, if you use one-hot encoding, all of these words are "equally different".

$$v_{\text{salt}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad v_{\text{savory}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad v_{\text{sugar}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad v_{\text{sweet}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{11.3}$$

You could **shuffle** the rows of one-hot vectors, and represent the same information.

So, we can't use the order of 1's and 0's to determine "closeness": the order can be **freely changed**.

In order to incorporate this information, we'll need a better way to represent words as vectors.

### 11.1.2 Word Embeddings: Similarity between words

Our new approach will convert each word $w$ into a **vector** $v_w$ of **length** $d$.

Unlike one-hot encoding, we don't require that d equals the size of our vocabulary.

$$w \longrightarrow v_w \qquad v_w \in \mathbb{R}^d \tag{11.4}$$

*How* do we want to convert words into vectors? Above, we mentioned that one-hot doesn't tell us how **similar** two words are.

**Clarification 5**

There are many ways for words to be **similar**: similar word length, similar choice of letters, etc.

But in our case, we're interested in **semantics**: the **meanings** of the words. We want to know which words have similar meanings.

- **Example:** We don't consider "sugar" and "sweet" to be similar because they both start with "s".
  - They're similar because of **meaning**: sugar tastes sweet. Sweet strawberries contain sugar.

**Concept 6**

We often want our **word embeddings** $v_w$ to tell us which words are **semantically similar** to each other: which words have similar **meanings**.

$v_a$ and $v_b$ are **similar vectors** $\iff$ a and b are **semantically similar words**

Our goal is to make this statement true. But we have a problem: these are *concepts*, rather than computable *numbers*.

- So, we'll have to turn each side into something computable.

### 11.1.3 Vector Similarity: Dot Products

First, we'll handle the left side: how do we know if vectors are **similar**?

- We've come across this problem multiple times, and we'll solve it the same way as always: using the **dot product**.

**Concept 7**

*Review from the Classification chapter*

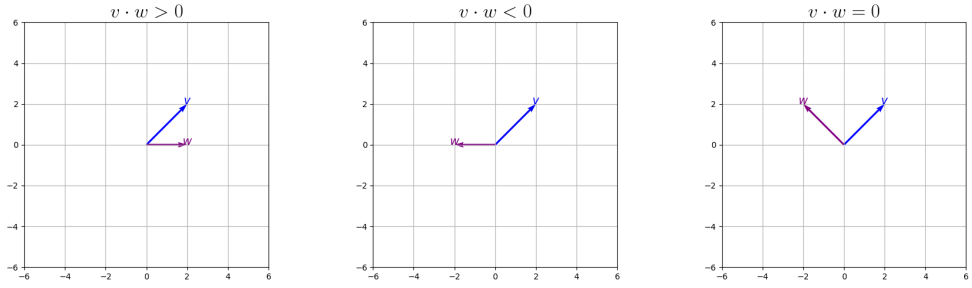
You can use the **dot product** between vectors $u$ and $v$, **normalized by their magnitudes**, to measure their "**cosine similarity**".

$$S_C(u, v) = \frac{u \cdot v}{|u| \cdot |v|}$$

If two vectors are more **similar**, they have a **larger** normalized dot product.

- This function ranges from -1 (opposite vectors) to +1 (identical vectors). Perpendicular vectors receive a 0.

We call it "cosine similarity", because this is equal to the cosine of the angle $\alpha$ between $a$ and $b$.




We can see here what we mean by "similar" or "dissimilar".

**Clarification 8**

You can use $S_C(u, v)$ to measure the **similarity** between two vectors, ignoring magnitude.

But for simplicity, we'll skip the **normalizing** step, and just take the **dot product**:

$$S_D(u, v) = u \cdot v$$

We're getting closer to a computable form:

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \iff a \text{ and } b \text{ are semantically similar words} \tag{11.5}$$

### 11.1.4 Semantic Similarity and Word Frequency

The "right side" of our expression is a bit trickier: how do you compute which words have **similar meanings**?

We can't directly turn "meaning" into a number. But instead, we'll focus on a different

concept, that might help us predict similarity:

- **Example:** Earlier, we showed that "sweet and "sugar" were related, by referencing the fact that "**sugar tastes sweet**".
- While our machine might not understand the concept, it can see that "sugar" and "sweet" showed up **together** in a sentence.

How much do/can large language models "understand" what they're saying? Lots of very smart people continue to argue exactly how much they know.

Often, words that are related, show up in the same sentences, or paragraphs. So, we'll try to use this to our advantage:

a and b are **semantically similar words** $\overset{\text{maybe?}}{\Longleftrightarrow}$ a and b **frequently** show up together

These two aren't *actually* equivalent, but we hope that we can use one to predict the other.

**Concept 9**

We can predict which words might be **more similar** by observing **how often** they show up **together** in a body ("corpora") of text.

- When two words occur **together** in a context, we call this **co-occurrence**.
- Thus, we're measuring **frequency of co-occurrence**.

If two words show up near each other more frequently, we predict that they might be **more similar**.

This kind of word embedding is often called "**word2vec**", named after a particular set of algorithms that use this approach.

- **Example:** The words "quantum" and "physics" go together often. So do the words "rain" and "weather".

Sometimes, "word2vec" is used to reference any technology that creates word embeddings. But this isn't always technically accurate.

**Clarification 10**

We **don't actually know** for certain that, if two words often show up together, they have **related meanings**.

But, in practice, we find that "**frequency of co-occurrence**" is a **surprisingly good** measure of similarity.

Because we're talking about frequency, we'll consider the **probability** of seeing both words together.

a and b are **semantically similar words** $\overset{\text{maybe?}}{\Longleftrightarrow}$ P ( a and b **occur together** ) is **high**

Finally, we have something closer to math:

$$\overbrace{v_a \cdot v_b \text{ is large}}^{\text{Similar vectors}} \iff \overbrace{P\left(a \text{ and } b \text{ occur together}\right) \text{ is high}}^{\text{Similar words}}$$

Now, we have some "mathematical" concepts: we can start using these to create mathematical **objects**.

### 11.1.5 Clarifying our probability

In order to proceed, we need to be a little more specific.

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \iff \overbrace{P\left(a \text{ and } b \text{ occur together}\right) \text{ is high}}^{\text{Similar words}}$$

The dot product is already an equation, so the left side is fine.

The right side is all we need to clear up: "P ( a and b **occur together** )" is a bit **vague**.

- We want to know if a and b tend to show up **together**, rather than **separately**.

Here's a concrete way to say this: "**if** we find one word, **how often** do we find the other nearby?"

One interpretation would be: "if we look at a random phrase, how often do we have words a and b?"

But we only care whether a and b are together/separate: we don't care about sentences containing neither.

**Concept 11**

To predict how **similar** words a and b are, we want to compute how often they **co-occur**.

- One way to phrase this: "**given** that we find a, what are the **chances** we find b nearby?"

$$\mathbf{P}\left\{b \text{ nearby} \;\middle|\; a \text{ found}\right\}$$

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \iff \overbrace{\mathbf{P}\left\{b \text{ nearby} \;\middle|\; a \text{ found}\right\} \text{ is large}}^{\text{Occur together frequently}}$$

We're getting warmer!

- We "**find**" a at index t:

$w_t$ is the $t^{th}$ word in our passage.

$$w_t = a \tag{11.6}$$

- Now, let's define what it means for b to be "**nearby**".

**Definition 12**

In a text, we may want to find the "context" for **center word** $w_t$: we want all of the words **nearby**.

- We'll use the c nearest words on either side: these are our **context words**. c is our **maximum skip distance**.

This collection of $2c+1$ words is called our **context window**.

$$\cdots \quad w_{t-4} \quad \overbrace{w_{t-3} \quad w_{t-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2} \quad w_{t+3}}^{\text{Context window for } c=3} \quad w_{t+4} \quad \cdots$$

We call c our "maximum skip distance", because it's the largest number of words we can "skip" over, starting from $w_t$.

We're allow to move over by c words, in either direction.

So, we want to look for b in our context window. There are two ways we can turn this into a probability:

- You check all of the context words **at the same time**:

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2}}^{\text{All words within } c=2 \text{ context window}} \quad w_{t+3} \quad \cdots \tag{11.7}$$

- You check **one word at a time**: j units to the right/left.

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2}}^{\text{Only } w_{t-2}\text{. Thus, } j=-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2} \quad w_{t+3} \quad \cdots \tag{11.8}$$

For now, it's easier to use the latter approach: **each index** has a **separate probability**.

**Concept 13**

We measure the **co-occurrence** of $a$ and $b$ by asking:

- "**Given** that we find $a$ at index $t$...

$$w_t = a$$

- what are the **chances** that we find $b$ at index $t + j$?"

$$w_{t+j} = b$$

With this, we find our result:

$$\mathbf{P}\Big\{ w_{t+j} = b \;\Big|\; w_t = a \Big\}$$

We did it! This is a clear, explicit probability.

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \iff \overbrace{\mathbf{P}\Big\{ w_{t+j} = b \;\Big|\; w_t = a \Big\} \text{ is large}}^{\text{Occur together frequently}}$$

**Notation 14**

We can make this notation a little denser:

$$\mathbf{P}\Big\{ w_{t+j} = b \;\Big|\; w_t = a \Big\} \quad = \quad \mathbf{P}\Big\{ b \;\Big|\; a \Big\}_j$$

This assumes that $t$ **doesn't** affect our probability: it doesn't matter **where** we found $a$, just **how far away** $b$ is (and on which side).

- This is a reasonable assumption for our purposes.

### 11.1.6 Computing predicted probabilities

How do we turn a **real number** $v_a \cdot v_b$ into a **probability** $P(b \mid a)_j$?

- $P(b \mid a)_j$ is the chance of finding $b$ at index $t + j$, if $a$ is at index $t$.

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2} \quad w_{t-1} \quad w_t}^{\text{What word is at } w_{t-2}\text{? Is it b?}} \quad w_{t+1} \quad w_{t+2} \quad w_{t+3} \quad \cdots \tag{11.9}$$

- So, we need to compare $b$ to every other word that we could find at $t + j$: this is a

**multi-class problem**, using the **softmax function**.

We have one class for each possible word we could find at $t+j$.

$$\text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_i e^{z_i}} \tag{11.10}$$

Let's review the concept behind "softmax":

**Concept 15**

Suppose that we have $n$ **possible words** ($n$ "classes"), and we want to figure out which one is **correct**.

The $k^{\text{th}}$ **class** has a score, $z_k$, used to compute probability.

- The bigger $z_k$ is, the **more likely** $k$ is to be the **correct class**.

To keep it **positive**, $z_k$ is converted to $e^{z_k}$: each $e^{z_i}$ competes to see which class is more likely.

- To create a probability, we **compare** the score of class $k$ to all of our other classes, using **softmax**.

$$\overbrace{e^{z_k}}^{\text{Class k}} \text{ vs } \overbrace{\sum_i e^{z_i}}^{\text{All classes}} \implies \text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_i e^{z_i}}$$

- We repeat this process for every possible word $i$, to get all of our predictions.

Now, the big question: what is $z_k$?

$$\overbrace{(v_a \cdot v_b) \text{ is large}}^{\text{Similar vectors}} \iff \overbrace{P\left\{w_{t+j} = b \mid w_t = a\right\} \text{ is large}}^{\text{Occur together frequently}}$$

$z_k$ and $(v_a \cdot v_b)$ serve the **same purpose**:

- Large **dot product** predicts high probability.
- Large $z_k$ predicts high probability.

So, we can use our dot product as a "score" $z_k$:

$$z_b = v_a \cdot v_b \tag{11.11}$$

Now, we can plug this into our probability equation!

**Key Equation 16**

The **more similar** (bigger dot product) $a$ and $b$ are, the **more likely** we predict to find them together.

- We use a **softmax** to compute this probability for each possible word $b$.

$$\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\} = \frac{e^{v_a \cdot v_b}}{\sum_i e^{v_a \cdot v_i}}$$

Or, in alternate notation:

$$\mathbf{P}\left\{b \;\middle|\; a\right\} = \frac{\exp\left(v_a \cdot v_b\right)}{\sum_i \exp\left(v_a \cdot v_i\right)}$$

Ta-da! We've combined two separate concepts into a single equation.

Note that, in both top and bottom, we keep $v_a$: we're considering every possible word for $w_{t+j}$, while we *know* $w_t = a$.

### 11.1.7 Skip-gram approach: Training our word2vec model

One remaining issue: this equation doesn't tell us what the "true" probabilities are: they tell us the probability that our model **predicts**.

- Now, we have to choose a **good model** (word embedding).

**Clarification 17**

Our equation is $\mathbf{P}\left\{w_{t+j} = b \;\middle|\; w_t = a\right\}$ is our **estimation** for the probability.

- The real probabilities could be **different**: we'll design our word embedding to give us the most **accurate probabilities**.

First: what does our model look like? How do we even **generate** word embeddings?

- Often, we rely on a neural network.

**Definition 18**

We have two common **models for word embedding** ($\theta$):

- Separately assigning a **vector** to each word.
- Using a shared **neural network** to embed every word as a vector.

Our neural network uses **parameters** $\theta$. We'll use $\theta$ to represent our embedding, that we want to **train**.

$$w \xrightarrow{\theta} v_w$$

How do we pick a good model?

- We'll **train** our embedding $\theta$, so that our **probabilities** are as accurate as possible.

As we established, our problem is multi-class classification:

**Concept 19**

*Review from Classification chapter*

For **multi-class classification**, we use the **negative log-likelihood multiclass** (NLLM) equation to compute **loss**:

$$\mathcal{L}_{\text{NLLM}}(g, y) = -\sum_{i=1}^{n} y_i \log(g_i)$$

$y$ is a one-hot vector, so all terms of the sum except the "correct" term $i = k$ cancel out to 0:

$$-y_k \log(g_k) \qquad \xrightarrow{y_k = 1} \qquad -\log(g_k)$$

$g_k$ is the probability we assigned to the correct answer.

Next, we need training data: a body of **text**.

- For an example, let's visit index $t$ in the text: this is the center of our **context window**.
- $a$ is replaced by whatever word we find at that index: $w_t$. We still want to predict $w_{t+j}$.

$$\cdots \quad w_{t-3} \quad w_{t-2} \quad w_{t-1} \quad w_t \quad \cdots \quad w_{t+j} \quad w_{t+j+1} \quad \cdots \tag{11.12}$$

How good is our word embedding? According to NLLM: "how likely were we to correctly

predict $w_{t+j}$?"

$$\mathcal{L}_{\text{NLLM}}(g, y) = -\log\left(\mathbf{P}\left\{\text{Correct word for index } t+j \;\middle|\; w_t\right\}\right)$$

The correct word for index $t+j$ would be... $w_{t+j}$. We can read the outcome from the text, and use our model to check how **likely** we thought that outcome was.

$$\mathcal{L}_{\text{NLLM}}(g, y) = -\log\left(\overbrace{\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}}^{\text{How likely we thought } w_{t+j} \text{ was, based on model}}\right)$$

Note that, the higher this probability is (the more sure we are of the correct answer), the closer the loss gets to 0.

**Key Equation 20**

We train our **word embedding** $\theta$ by **maximizing** the probability $\mathbf{P}(w_{t+j} \mid w_t)$ of predicting the **correct word** in each spot.

In our case, we want to **minimize**

$$\mathcal{L}_{\text{NLLM}}(\theta, j) = -\log\left(\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}\right)$$

where

$$\mathbf{P}\left\{b \;\middle|\; a\right\} = \frac{\exp\left(v_a \cdot v_b\right)}{\sum_i \exp\left(v_a \cdot v_i\right)}$$

Now, we know how to compute these odds for a **single index**, $t+j$. We want to repeat this process for the rest of our context window:

$$\cdots \quad w_{t-3} \quad \overbrace{w_{t-2} \quad w_{t-1} \quad w_t \quad w_{t+1} \quad w_{t+2}}^{\text{All words within } c=2 \text{ context window}} \quad w_{t+3} \quad \cdots \tag{11.13}$$

**Key Equation 21**

We can find the **total loss** of our embedding $\theta$, over our entire **context window**, by adding up the loss from each **context word**.

This includes all of the indices $(t+j)$, going from $j=-c$ to $j=+c$. Meaning, we want

- $|j| \leqslant c$ (within window)
- $j \neq 0$ (don't want to compare $w_t$ with itself)

$$\mathcal{L}_t(\theta) = -\sum_{|j|\leqslant c}^{j\neq 0} \log\left(\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}\right)$$

One more modification: the loss function above only computes loss for a **single context window**.

But, for a passage of text, there are many possible context windows: all we have to do is shift our target word, $w_t$.

- **Example:** Below, with $c=2$, we'll show all of our possible context windows:

Target word is red, context words are blue.

This is a sample sentence
This is a sample sentence
This is a sample sentence (11.14)
This is a sample sentence
This is a sample sentence

We'll average the loss over **all context windows**.

We'll ignore negative indices, so we don't cause problems when $t=0$ or $t=T$.

**Key Equation 22**

Take a body of text with $T$ words, and a **context window** with a **max skip distance** of $c$. We use word embedding $\theta$.

Our **objective function** $J(\theta)$ for the **skip-gram word2vec** algorithm, over the entire passage, is:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}_t(\theta)$$

or,

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \left( \sum_{|j|\leqslant c}^{j\neq 0} \log\left(\mathbf{P}\left\{w_{t+j} \;\middle|\; w_t\right\}\right) \right)$$

This is the completed loss function that we can use to train our embedding.

**Concept 23**

We can train our **embedding** $\theta$ using our **loss function** $J$, via **gradient descent**.

$$\theta' = \theta - \eta \nabla_\theta J$$

- If we're using a **neural network** to create embeddings, we'll need **back-propagation** to train.

### 11.1.8 One-hot encoding

One quick side-detail: typically, in multi-class, $y$ is a one-hot vector.

$$y = \begin{bmatrix} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{bmatrix}^T$$

We'd like to introduce notation to freely move between

- A word,
- Its one-hot vector,
- Its vector embedding

First, let's turn words into one-hot vectors:

**Notation 24**

Suppose we have our **vocabulary** of words given a particular **order**. The $n^{th}$ word in our vocabulary is notated as $w^n$.

- $w^n$ will be converted into a **one-hot vector** with a "1" in dimension $n$.
- $I(w)$ is the **function** that does this: takes in a **word**, and returns its **one-hot vector**.

$$I : \left\{\text{All N words}\right\} \rightarrow \left\{\text{All length-N one-hot vectors}\right\}$$

**Example:** Consider a vocabulary of three words: $w^1$, $w^2$, $w^3$.

$$I(w^1) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad I(w^2) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad I(w^3) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Next: how do we convert between one-hot vector and vector embedding?

**Notation 25**

We'll store all of our vector embeddings in matrix $W^T$:

$$W^T = \begin{bmatrix} | & | & & | \\ v_{w^1} & v_{w^2} & \dots & v_{w^N} \\ | & | & & | \end{bmatrix}$$

We can use this to convert a **one-hot vector** into its **vector embedding**, using matrix multiplication:

$$v_{w^n} = W^T \cdot I(w^n) = \begin{bmatrix} | & | & & | \\ v_{w^1} & v_{w^2} & \dots & v_{w^N} \\ | & | & & | \end{bmatrix} \cdot \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

- Note that, if we don't use a **neural network**, $W^T$ fully describes our encoding $\theta$.

Now, we can convert our word between our three forms.

### 11.1.9 Issues with skip-gram

There are a few problems worth addressing. First, look again at our equation:

$$P\left\{w_{t+j} = b \;\middle|\; w_t = a\right\} = \frac{e^{v_a \cdot v_b}}{\sum_i e^{v_a \cdot v_i}}$$

Something might strike you: our probability is **totally independent** of which index $t + j$ we want to predict.

- That means, our model would make the exact same prediction for every nearby word.
- This is more easily resolved in our transformer model, so we won't worry about it for now.

**Concept 26**

Our **probability** calculation in skip-gram is **independent** of the **skip distance** $j$ between words $w_t$ and $w_{t+j}$.

- All words within the **context window** have the **same probability distribution**.

Another problem: if "more **similar**" means "more likely to **co-occur**", doesn't that suggest that we would expect a word to appear with itself, really often?

- This would be true for every word: nothing can be more similar to a vector than itself, after all.
- Our solution is to just **exclude** $w_t$ from predictions about nearby words.

**Concept 27**

The most similar word to $w_t$, is **itself**!

- So, we often **exclude** $w_t$ from predictions.

Another problem: our objective function $J(\theta)$ includes a logarithm. To optimize $\theta$, we'd need to compute its **derivative**.

- This becomes really expensive, especially when our vocabulary can have millions of words.
- Our solution is to "**prune**"/remove a lot of words from our probability calculation.

We can predict in advance, that some words don't need to be included.

**Concept 28**

**Skip-gram** can become expensive to train, when the **vocabulary** becomes too large.

- So, we **prune** some unlikely words in our vocabulary, to speed up our **predictions**.

One more concern: so far, we've talked about predicting whole words, because it's easy to work with.

- But often, for language analysis, we break up words into parts, called **tokens**.
- These are the objects we study/predict, rather than whole words.

**Clarification 29**

Rather than using/predicting entire **words**, we use **small parts of words**, called **tokens**.

- A "token" is the **smallest unit** in our language model.

- **Example:** You might break up the word "eating" into "eat" and "ing": both are meaningful, by themselves.

This is another reason our vocabulary becomes so big: we're not just considering every word, we're including smaller pieces of words.

- While "tokens" are more appropriate than "words", words often make for better examples, so we'll keep using them through the rest of this chapter.

**Clarification 30**

We'll continue using words (instead of tokens) for examples, when it's convenient.

### 11.1.10 "Adding" words together

Our word2vec system works under the hope that these vector embeddings can accurately represent the **meanings** of words.

- In practice, this assumption works **surprisingly well**, for being so simple.

One example is the idea of "**adding**" words together. Normally, it's hard to say how to "add words" together, but we *do* know how to add **vectors**.

Consider the following example:

$$v_{king} - v_{man} + v_{woman} \approx v_{queen} \tag{11.15}$$

This sort of reasoning makes sense to most english speakers:

$$\overbrace{v_{king} - v_{man}}^{\text{monarch}} + v_{woman} \approx \overbrace{v_{queen}}^{\text{female monarch}} \tag{11.16}$$

We can repeat this process for other words:

Paris is the capital of France, and Rome is the capital of Italy.

$$v_{paris} - v_{france} + v_{italy} \approx v_{rome} \tag{11.17}$$

**Concept 31**

Transforming a word into a **vector** allows you to use vector operations, like **addition** and **subtraction**.

- The result can be surprisingly **meaningful**, for some word combinations.

This approach doesn't always work, but the fact that it works *sometimes* suggests that our vectors might capture real information about the "**meanings**" of words.

That said, this approach is often an over-simplification:

**Concept 32**

Reducing a word to a **single vector** can cause problems, because the same word might change its meaning, based on **context**.

- **Example:** For example, the word "bank" has a very different meaning when you compare "bank account" to "river bank".

## 11.2 Attention

Our **word embedding** technique has given us a basic way to talk about which words are "**related**".

- We can even use this to learn some about the "**meanings**" of words.

But there's some work to be done:

**Concept 33**

Our **word embedding** technique has two major problems, for representing the **meanings** of words:

- There's a lot of information we're **missing**: **similarity** to other words isn't enough. We'll need a vector to represent that information.
- The meaning of a word is **contextual**: the sentence you put a word in, will affect its meaning.

It may not look like it, but our **word embedding** technique has already given us the basic tools we need to solve these problems.

Here's the basic idea, for how we handle each problem:

**Concept 34**

We'll create a system that solves both of these problems, using **3 word embeddings**: $v$, $k$, and $q$.

- We'll **embed information** about each word in a **value vector** $v$.
- When finding the **meaning** of a word, we'll calculate **context** from nearby words.
  - We'll use **word similarity** to figure out which parts of the context are **most important**.
  - For this purpose, word will need **two embeddings**: a **key vector** $k$, and a **query vector** $q$.

The result is a powerful model called the **attention mechanism**.

This description is over-simplified, which is why we'll need to go into detail below.

### 11.2.1 The Attention Mechanism: queries, keys

Let's consider an **example**, to get used to the idea of k, v, and q.

Suppose we want a general idea of what "mexican" food is like. We'll need to consider lots of foods, and take an **average** of those we consider to be "mexican".

Admittedly, we're turning "mexican-ness" into a number, which can be a bit strange.

It may help to think this way: "if someone is talking about mexican food, how often are they talking about this food?"

This problem comes in three parts: let's consider the first two, "query" and "key".

- **Query** q: we're searching for "mexican" food. The word "mexican" is represented by a **query vector** q.
  - This is like our previous **word2vec** embedding: if two vectors are similar, then we expect them to have similar/related **meanings**.
  - So, we'll **compare** q to each food, to see which foods are 'close' to mexican.

**Definition 35**

The **query vector** q represents a word, that we're **comparing** to **several other words** ("keys").

- It answers the question, "what kinds of words are we **searching** for?"

Because q is a word embedding, it encodes "meaning": similar words, should have similar vectors.

- And we expect **similar words** to be **more relevant** to our query.

- **Key** k: Each food (apple, burrito, sushi...) has a **key vector** k, representing it.
  - A **word2vec**-style embedding, just like the **query**.
  - Combining k and q will tell us which foods are **'more' mexican**.

**Definition 36**

The **key vector** k represents a word, that we want to **compare** to **the query** q.

- It answers the question, "what kinds of searches does this word **match**"?

Because it's a word embedding, which encodes meaning, we expect that, if k **and** q **are similar**, then our key word is **more relevant** to our query.

Each embedding has a role: a **query** is used to search for relevant words, and a **key** is responding to that search, on behalf of one word.

Reminder that when we say "word", we're simplifying: we could talk about any kind of token.

**Concept 37**

Another way we could view keys vs. queries:

- **Query vector** $q$: asks, "how relevant are these words/tokens **to me**?"
- **Key vector** $k$: asks, "how relevant is my word **to the query**?"

Notice that we've made a perspective shift, in how we view word embeddings:

**Concept 38**

When we were developing word2vec, we wanted **similar vectors** to represent **semantically similar** words.

- But, in this case, we're less focused on "similarity", than **relevance**.

We look for **keys** that are the **most relevant** to our **query**.

These two ideas don't necessarily conflict, but they have somewhat different goals.

### 11.2.2 The parts of Attention: attention weights

How do we compute how similar $k$ and $q$ are? The same way as we did for word2vec: we use a **dot product**.

**Key Equation 39**

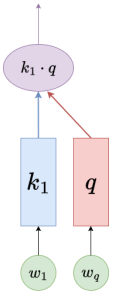
We can get a score for how **relevant** the word $b$ is to word $a$, by taking the **dot product** between $b$**'s key**, and $a$**'s query**.

$$k_b \cdot q_a$$

We can also write this as matrix multiplication:

$$k \cdot q = k^T q$$

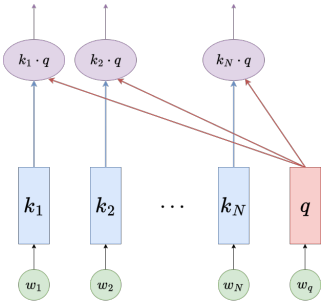
This gives us a "**score**": the higher $k \cdot q$ is, the more similar they are.




We convert $w_1$ and $w_q$ into a key and query, respectively, before taking the dot product.

But we're not just considering one key word: we're considering **all of of them**.

- In our "mexican food" example, we need to check every food, to see which ones best fit the category.




We re-use our query q for every single dot product.

How do we compare each of these keys?

- Once again, we'll reuse a tool from word2vec: **softmax**.

**Key Equation 40**

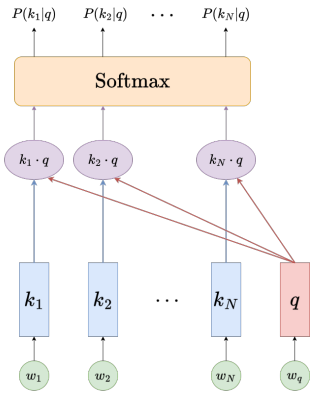
We can compute the relative **relevance** of a key $k_n$, by:

- **Comparing** each key $k_i$ to $q$ ($k_i \cdot q$)
- Use **softmax** to compute $p(k_n|q)$: given query $q$, how important is $k_n$?

$$\mathbf{P}\left\{k_n \;\middle|\; q\right\} = \frac{e^{k_n \cdot q}}{\sum_i e^{k_i \cdot q}}$$

$\mathbf{P}\left\{k_n \;\middle|\; q\right\}$ tells you, "how much **attention** should $q$ pay to $k_b$"?

- Thus, we call $\mathbf{P}\left\{k_n \;\middle|\; q\right\}$ an **attention weight**.




Finally, we've converted each word into their "probability" of being relevant.

One notational thing: we can write this a bit more densely.

- So far, we've been computing $k^T q$ for each $k_i$ term **separately**.
- But, one benefit of matrix multiplication, is that we can **combine multiple operations** into one.

First, we'll **combine** all of our key vectors into a matrix $K$:

$$K = \begin{bmatrix} | & | & & | \\ k_1 & k_2 & \dots & k_N \\ | & | & & | \end{bmatrix} \tag{11.18}$$

With that, we can compute all of our dot products **at the same time**:

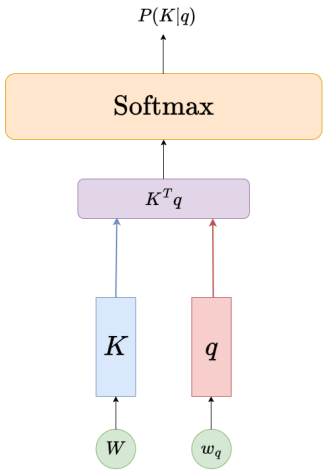
$$K^T q = \begin{bmatrix} k_1 \cdot q \\ k_2 \cdot q \\ \vdots \\ k_N \cdot q \end{bmatrix} \tag{11.19}$$

And we can combine all of these together into a softmax.

**Key Equation 41**

By combining all of our keys into a matrix K, we can compute all of our **attention weights at the same time**.

$$\mathbf{P}\left\{K \;\middle|\; q\right\} = \begin{bmatrix} \mathbf{P}(k_1 \mid q) \\ \mathbf{P}(k_2 \mid q) \\ \vdots \\ \mathbf{P}(k_N \mid q) \end{bmatrix} = \text{softmax}\left(K^T q\right)$$




Now, our diagram is visually simpler, though it reflects the same information.

### 11.2.3 The Attention Mechanism: values, attention

Now, we have a collection of **attention weights**: each one tells us relevant each word is to q.

- Now, we want to make them useful. Our original goal was to get an **average** sense of what "mexican" food is like.

To make this concrete, we'll introduce our third embedding: the **value vector**.

- **Value $v$**: Each food has a **value vector**, directly storing information about a word.
  - Unlike the key/query vectors, this embedding isn't based on **similarity** to other words.
  - Instead, it usually contains more direct **information** about our word: in this example, maybe it contains the price, calories, ingredients, etc.

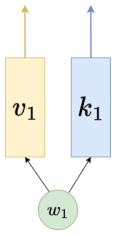
Note that, in a real model, value vectors are often "learned" during training. So, they won't always contain such simple, easily explained data.

**Definition 42**

The **value vector** $v$ represents a word, and stores useful **information** that it can contribute to the **query**.

- It answers the question, "what useful data could this word contribute to the query?"

By **adding together** the value vectors from each word **relevant** to the query, we can get an overall "**averaged value**" for q.




Each word has both a value and a key attached to it.

For our example, let's suppose that the value vector contains price, calories, and salt.

$$v_i = \begin{bmatrix} \text{price}_i \\ \text{cal}_i \\ \text{salt}_i \end{bmatrix} \tag{11.20}$$

We want to get an "average" calorie count for mexican food.

- Some foods are **common** for mexican food, and some are more rare.
- So, to get an average, we'll need to **emphasize** more "common" mexican food.

How do we do that? Using our **attention weights**: the larger the attention weight, the more "**relevant**" a food is to our mexican food calculation.

If we use q to represent mexican food, and $k_i$ is the **key** for the $i^{th}$ food, we get:

$$\text{cal}_q = \overbrace{\sum_i P(k_i|q)\ \text{cal}_i}^{\text{Weighted average}} \tag{11.21}$$

Rather than repeating this process for each row of $v$, we can just do a **weighted average** of the whole vector, at the same time:

$$v_q = \sum_i P(k_i|q)\ v_i \tag{11.22}$$

**Key Equation 43**

Each word $i$ has a **value vector** $v_i$, which represents all of the useful **information** it can provide to the **query**.

- We can use a **weighted average** to combine all of these value vectors together: this provides the "**overall context**" for the query.
- Each value is weighted based on its **attention weight** $P(k_i|q)$: how likely it is to be relevant.

$$v_q = \sum_i P(k_i|q)\ v_i$$

This is the calculation for **attention**.

Just like we did for the $k_i \cdot q$ operation, we can re-write this in terms of matrix multiplication.

- We'll change from $P(k_i|q)$ to $P(K|q)$.

$$\mathbf{P}\left\{K \;\middle|\; q\right\} = \begin{bmatrix} \mathbf{P}(k_1 \mid q) \\ \mathbf{P}(k_2 \mid q) \\ \vdots \\ \mathbf{P}(k_N \mid q) \end{bmatrix} = \text{softmax}(K^T q)$$

- We'll stack all of our value functions $v_i$ into a matrix $V$.

$$V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \dots & v_N \\ | & | & & | \end{bmatrix} \tag{11.23}$$

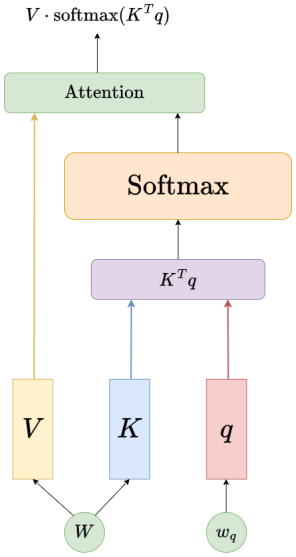
Now, we can compute with every value vector at once:

**Key Equation 44**

We can compute **attention** using matrix multiplication:

$$\text{Attention}\left(q, K, V\right) \quad = \quad V \cdot \text{softmax}\left(K^T q\right)$$

Where $\text{softmax}\left(K^T q\right)$ computes our **attention weights**.




We now have a completed representation of attention. The only thing missing: sometimes we normalize each dot product $k^T q$, before softmax.

With this, we can summarize the basic idea of attention:

**Concept 45**

**Attention** is a mechanism that allows you to **combine** information from multiple tokens, weighting each token by how **relevant** it is.

This mechanism is broken into three parts:

- **Value vector** $v$: **what information** are we trying to combine?
- **Query vector** $q$: what kinds of words are **relevant** to this search?
- **Key vector** $k$: what kinds of searches is this word **relevant** for?

Each token has a **value vector** (information from that token), and a **key vector** (used to compare this token to the query).

Note that this isn't the **only** way to do attention:

**Clarification 46**

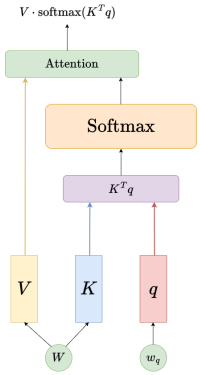
There are multiple ways we can implement attention.

- For example, we use $\mathtt{k} \cdot \mathtt{q}$ to measure similarity, but we could **replace** it with a different metric.

Reminder: a "metric" is just "a way of measuring something. The dot product is a **similarity metric** for vectors.

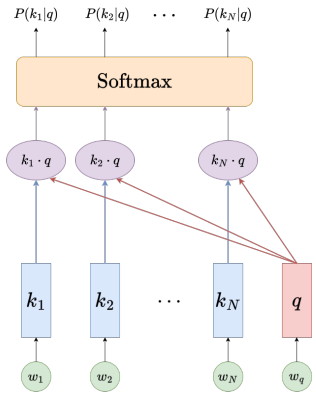
### 11.2.4 Diagramming attention (Optional)

Above, we opted to use the "condensed" matrix notation, when introducing V to the attention equation:



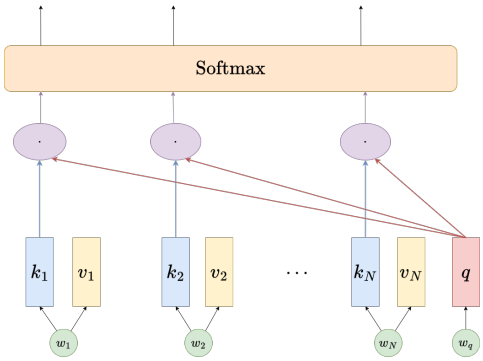

We now have a completed representation of attention. The only thing missing: sometimes we normalize each dot product $k^T q$.

But sometimes, it's helpful to be able to see each individual key and value, $k_n$ and $v_n$. We'll create an alternate diagram following this format.



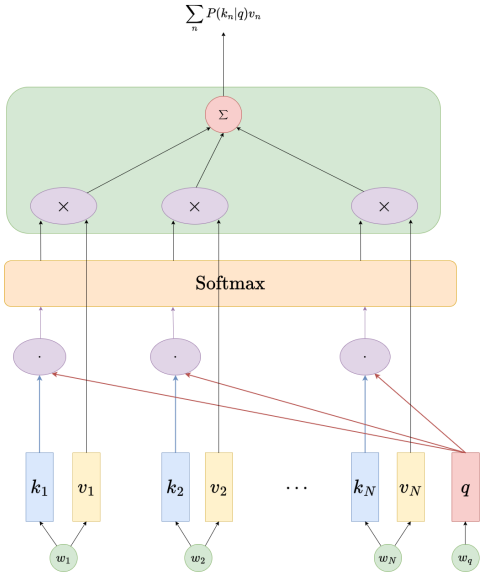

We'll start from this diagram.

Each word $w_n$ not only has its own **key**, but also its own **value**.




Each word produces a key and a value.

The notation here gets a bit... messy. We need to pair each weight $P(k_n|q)$ with its matching value, $v_n$.




The end result is messy, which is why we tend to go for the matrix-based notation.

### 11.2.5 RNNs: a review

Our new **attention mechanism**, is a powerful tool for language processing. We'll develop this in three stages:

- Reviewing our old language model: the **recurrent neural network (RNN)**.
- Improving on the RNN model with **attention**.
- Moving past RNNs, to create an **attention-only** model.

Attention is All You Need, one might say.

First: what exactly does a "language model" do? Language stuff, I presume.

- But we need to be more specific, if we want to solve a real problem.

**Definition 47**

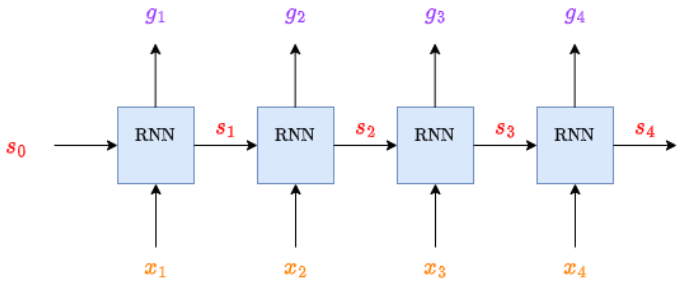
A basic **language model** follows the **predictive text** problem:

- You predict the **next word** in a sentence, given **previous words**.

- **Example:** This is similar to a phone's "autocomplete", which suggests a completed sentence, based on a partial text.

Previously, we used RNNs to handle this problem. Let's review the basic structure of an RNN:

If you want to review a little more in-depth, revisit chapter 10. For language stuff in particular, review 10.4.




At each timestep t, our model takes in an **input** $x_t$, saves information in **state** $s_t$, and **outputs** a value $g_t$.

**Concept 48**

In a **language model RNN**, each input $x_t$ is a **token**, representing one part of the text.

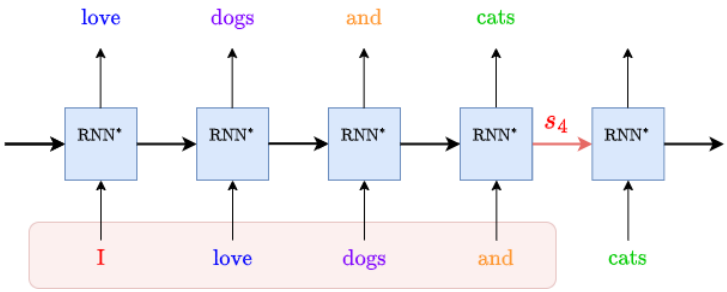
- Tokens which come later in a **sentence**, are presented to the RNN later in **time**.

These inputs are "remembered" using our state, $s_t$.

**Concept 49**

In a **language model RNN**, our **state** $s_t$ stores information about the **inputs** we've received before: all of the **context** in our sentence.

- Our RNN uses this context to **predict** our next token.




In a good RNN, we hope that state $s_4$ would contain info about the entire sentence portion that comes before: "I like dogs and".

Technically, we should turn our words into vector embeddings first. We'll skip that step, for visual simplicity.

All we would need is to turn each word into a **value vector**.

### 11.2.6 RNNs: Encoding

This property of RNNs is interesting: $s_t$ is designed to **encode** information about the prior inputs.

- This might remind of our **autoencoders** from an earlier chapter.

**Concept 50**

Our RNN can be thought of as an **encoder**: it compresses our input x into a more **compressed** representation.

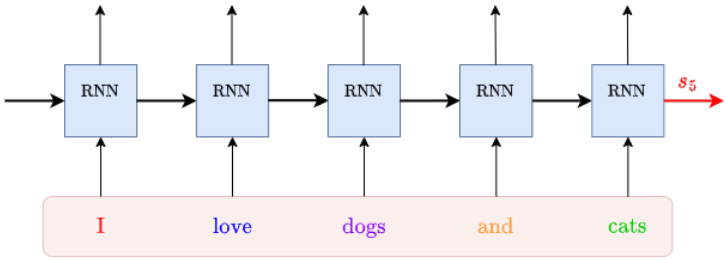
- This information is represented in the **state** $s_T$.

In particular, because RNNs process **sequential** data (like text), it can be used to **encode sequential information**.

We've created a new type of encoder!

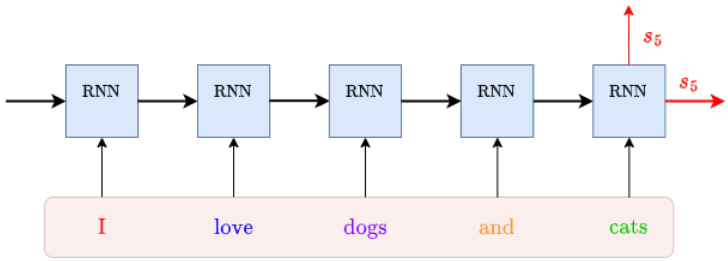
- Unlike a fully-connected NN (our previous style of encoder), this handles information **over time**.

We could take an entire sentence, and **encode** its useful information with an RNN.




After we've fed in all of our inputs, we hope that $s_5$ encodes the meaning of the sentence: "I love dogs and cats".

The **encoding** $s_5$ is what we're interested in. We'll omit the other outputs, for simplicity.




We don't know the full meaning of the sentence until we finish. So, we'll just focus on the state after seeing **every input**.

Notice that we **changed our output function**: $s_5$ is the output for the $5^{th}$ timestep, $g_5$!

**Definition 51**

Our **encoder RNN** returns its **state** $s_t$ as its **output** $g_t$.

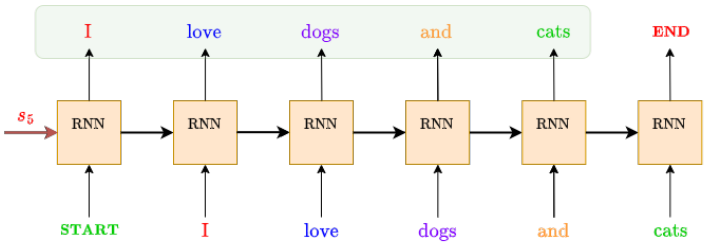
- The state is the **compressed representation** of the input $x$.
- For an encoder, that's the **most important information**: hence, we provide it as output.

$$g_t = s_t$$

### 11.2.7 RNNs: Decoding

Following this train of logic, we can create an **autoencoder**: we just have to **decode** our state with a second RNN.

- We'll start from our compressed state, $s_5$.




Now, this looks similar to our text prediction problem from before.

Similar to the text prediction problem, we'll feed in the correct answers as an input, *after* our model predicts it.

If we don't have a "correct" answer, we can just use our model's previous output as the input.

With any luck, we'll get the above, and the decoder can reproduce the original input, based on the state.

**Definition 52**

Our **decoder RNN** starts with **state** $s_T$, representing the sentence received by the **encoder RNN**.

- The goal is to sequentially **reproduce** the sentence that created $s_T$.

This model works similar to the text prediction problem: it receives the **correct input** one timestep after it predicts it.

And with this, we have a complete autoencoder.

### 11.2.8 RNNs: Translation

This autoencoder system is cool, but maybe a little basic: we've already seen autoencoders before.

- Instead, we'll try a different task: **translation!**

Here's our general idea:

- We can **encode** text in one language, into a compressed format.
- Then, we **decode** that text, in a different language.

The meaning is preserved, with the grammar and vocab shifted to suit the language.

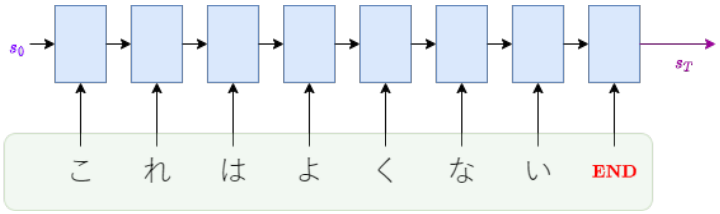
**Definition 53**

In the **RNN translation task**, we use an encoder-decoder system to convert text **between languages**.

- **Encoder**: We input our target text, and it outputs as $s_T$: this should represent the **meaning** of the sentence.
- **Decoder**: We input the embedding $s_T$, and it outputs a **translation** of our text into **another language**.

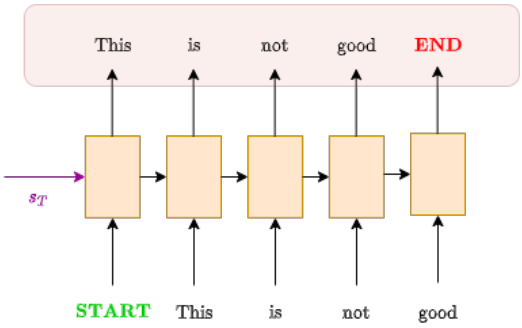
**Example:** We'll convert text from japanese to english. Our example text: "kore wa yokunai".

- First, we encode our **japanese** into a **state**.



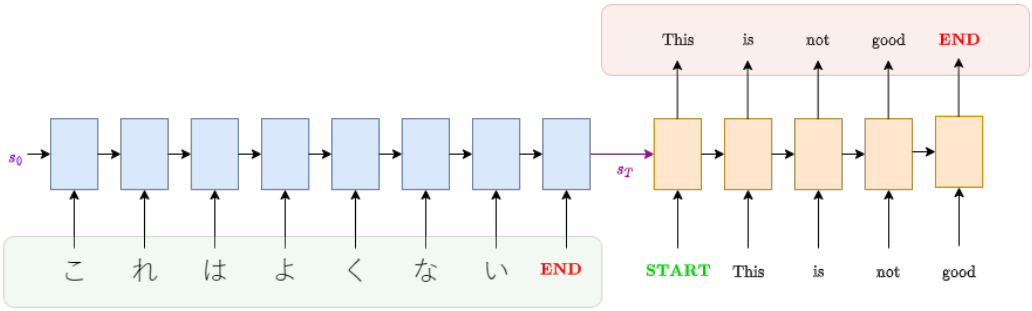

Our RNN is trained to understand japanese grammar and vocabulary.

- Then, we reverse this: turning our **state** into an **english** sentence.




Our RNN can "decode" the information in the state $s_T$, and turn it into english.

All together, we have a system we can use to translate whatever sentence we'd like.




Our japanese prompt is shown in green, and our english response is shown in red.

**CHAPTER IS INCOMPLETE**