# Before We Get Started…

Today, we'll be doing some live-coding!

- Go to this link: https://bit.ly/SeqLearningPracticum
- File > Save a copy in Drive [SUPER IMPORTANT STEP!!]

Instead of making you retype what's on my screen, I've provided the solutions

- Don't look ahead if you want to challenge yourself!
- The "solutions" are purposefully buggy anyway!

# CSE 493G1: Deep Learning

Friday Lecture 6: Sequence Learning Practicum

# Motivation

Large language models (LLMs) are the rage in AI right now—both within industry and in the public sphere

Many of you may want to work with LLMs or other sequence models as part of your project

We want to give you a taste of how to structure a sequence learning project, along w/ common pitfalls/bugs you'll encounter

# Plan for Today

- Overview of sentiment classification
- Data loading and training/testing utils
- Writing sequence model code
- Leveraging pre-trained LLMs

# Sentiment Classification: Task and Data

# Sentiment Classification Task

Sentiment classification is the task of deciding if an input text is “positive” or “negative” in overall sentiment

- “That was really cool!” -> “positive”
- “I hated that!” -> “negative”

Can be useful for things like recommendations and community monitoring

# IMDB Sentiment Dataset

The IMDB dataset contains examples of movie review, along with a simple binary label (positive or negative)

To work with this dataset, we must first make a vocabulary mapping common words in the training set to a unique index (and some special characters)

- Common words are lower: `vocab['the'] = 4; vocab['flute'] >> 4`
- Special padding character (for batching): `vocab['<pad>'] = 0`
- Special “uncommon word” character: `vocab['<unk>'] = 1`

# Code Sample 1: Dataset and Vocabulary

# PyTorch `Dataset` and `DataLoader`

In PyTorch, we create a subclass of the `Dataset` class which must implement `__init__`, `__len__`, and `__getitem__` functions

- The last function is the logic to load a single item from the dataset

The `Dataset` object can then be used to initialize a `DataLoader` object, which implements threaded loading, batching, and randomization for you

# Live Coding 1: `Dataset` and `DataLoader`

# Training/Testing Utilities

# Training and Evaluating the Model in PyTorch

For some number of epochs, we will:

- Go through the full training dataset (in batches) and make gradient updates
- Go through the full testing dataset and compute metrics

We want to compute loss, accumulate gradients, and make update steps during training, but not during testing—however, everything else is the same

- Let's write just one `run_epoch` function which does either training or testing depending on input parameter

# Live Coding 2: Training and Testing Utilities

# Model 1: Simple RNN

# Simple RNN Model

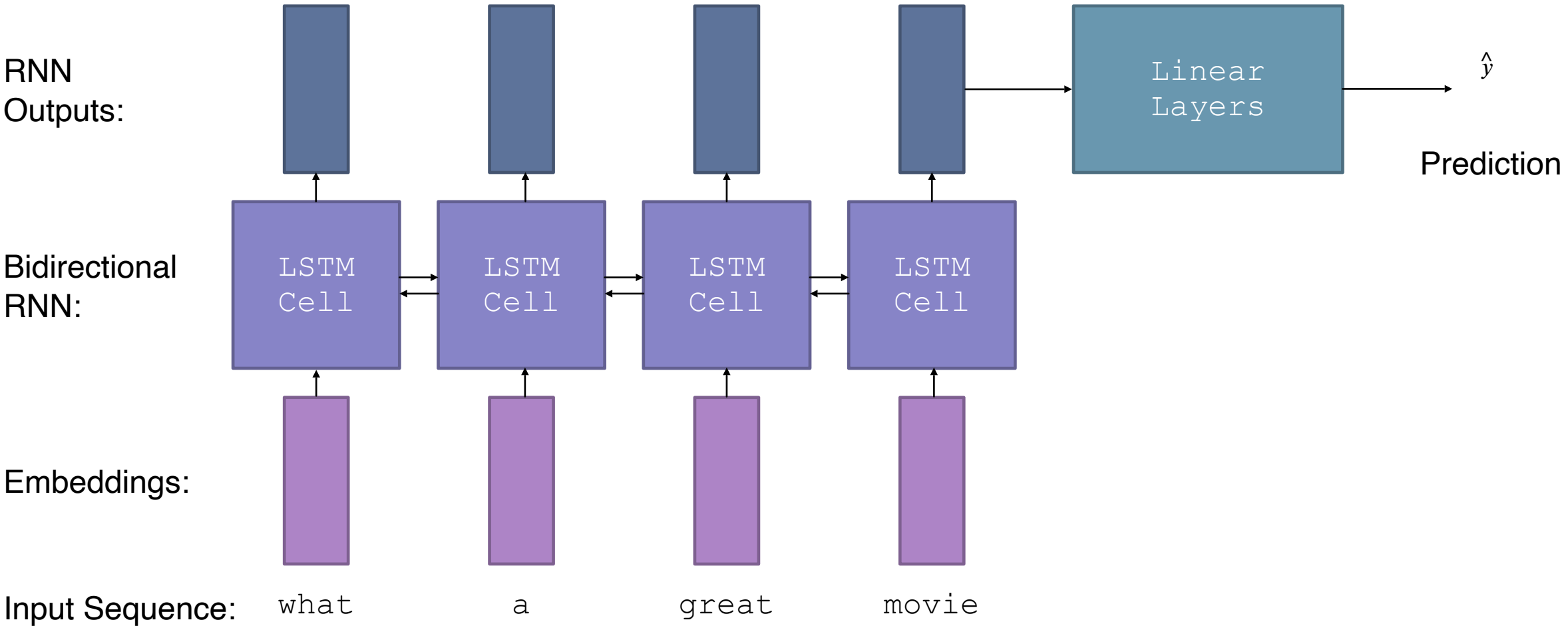
We will first use a very simple LSTM-RNN for sentiment classification task

The RNN forward pass will be as such:

- Lookup embeddings for each token in the sequence
- Pass embeddings through RNN
- Take the last* output vector from RNN and pass it through a couple linear layers

* Since seq length varies within batch and padding is used to achieve same-length examples, the “last vector” will be a different one for each example!

# Visual Intuition of RNN



# Live Coding 3: Simple RNN Model

# Bugs Summary 1

1. Bad Linear layer input dim causing runtime error
    1. This time: bidirectional RNN causes dim to be 2x
    2. In general: print shapes to make sure they are what you expect
2. Bad input to BCE loss function causing negative loss
    1. This time: 1 not subtracted from target, causing there to be 2 in vector
    2. In general: make sure loss function inputs are valid
3. Forgetting `zero_grad` call resulting in loss not going down
    1. This time: don't forget this
    2. In general: don't forget this lol

# Model 2: RNN w/ Attention

# Why Are We Taking the Last Vector?

Let's say the movie review was: "This was a very bad movie to me."

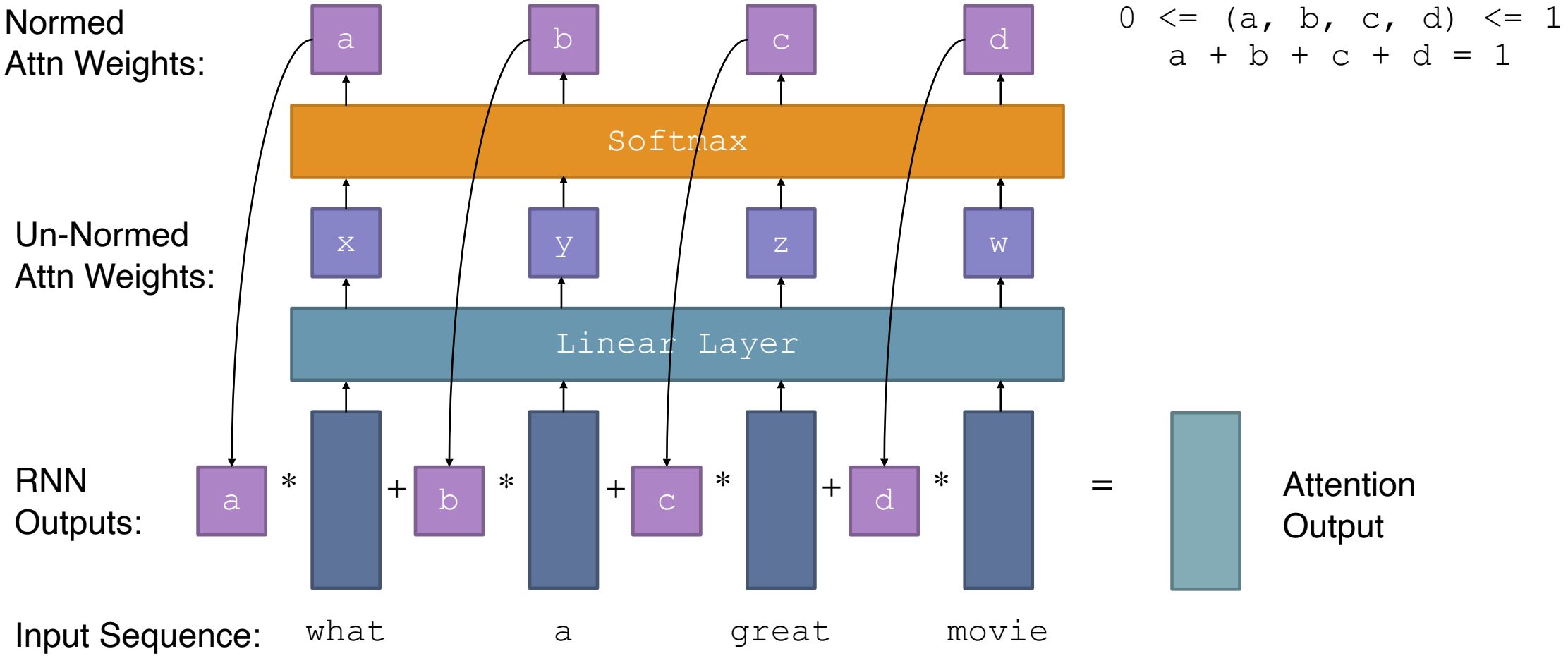
- The last vector represents the word "me"—the important word, though, is "bad"

We should learn a function which dynamically "decides" which vector to use

- Instead of picking one vector we will weight and add all the vectors together

Attention layer learns a parameters which uses the vectors to generate weights, then uses the weights to compute convex combination of the vectors to be passed to linear layers

# Visual Intuition of Attention Weighting



# Pad Masking

Remember that most of the examples in the batch have padding tokens

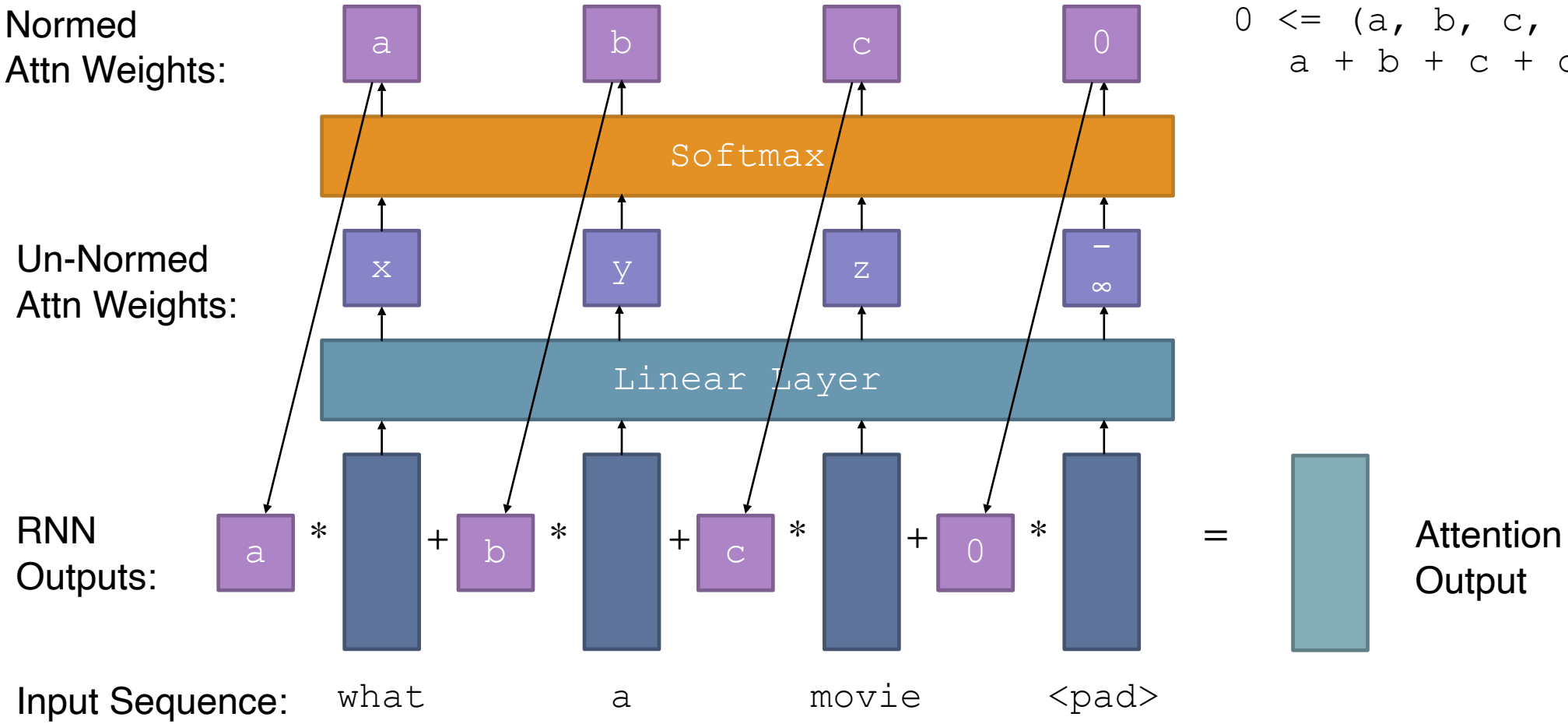
We want to ignore these vectors when computing the attention weights

- In other words, we want the attention weights for those to be 0

We can do so by setting the unnormalized attention weights to $-\infty$ on padding tokens

- Then, when we apply softmax, those weights will compute to 0

# Visual Intuition of Pad Masking



Before Coding Up Attention RNN,
Let's Revisit Simple RNN Results…

# Live Coding 4: Attention RNN Model

# Bugs Summary 2

1. Unnecessary sigmoid resulting in loss not going down
    1. This time: `binary_cross_entropy_with_logits_loss` expects (surprise) logits
    2. In general: make sure loss function inputs are valid

# Model 3: Transformer

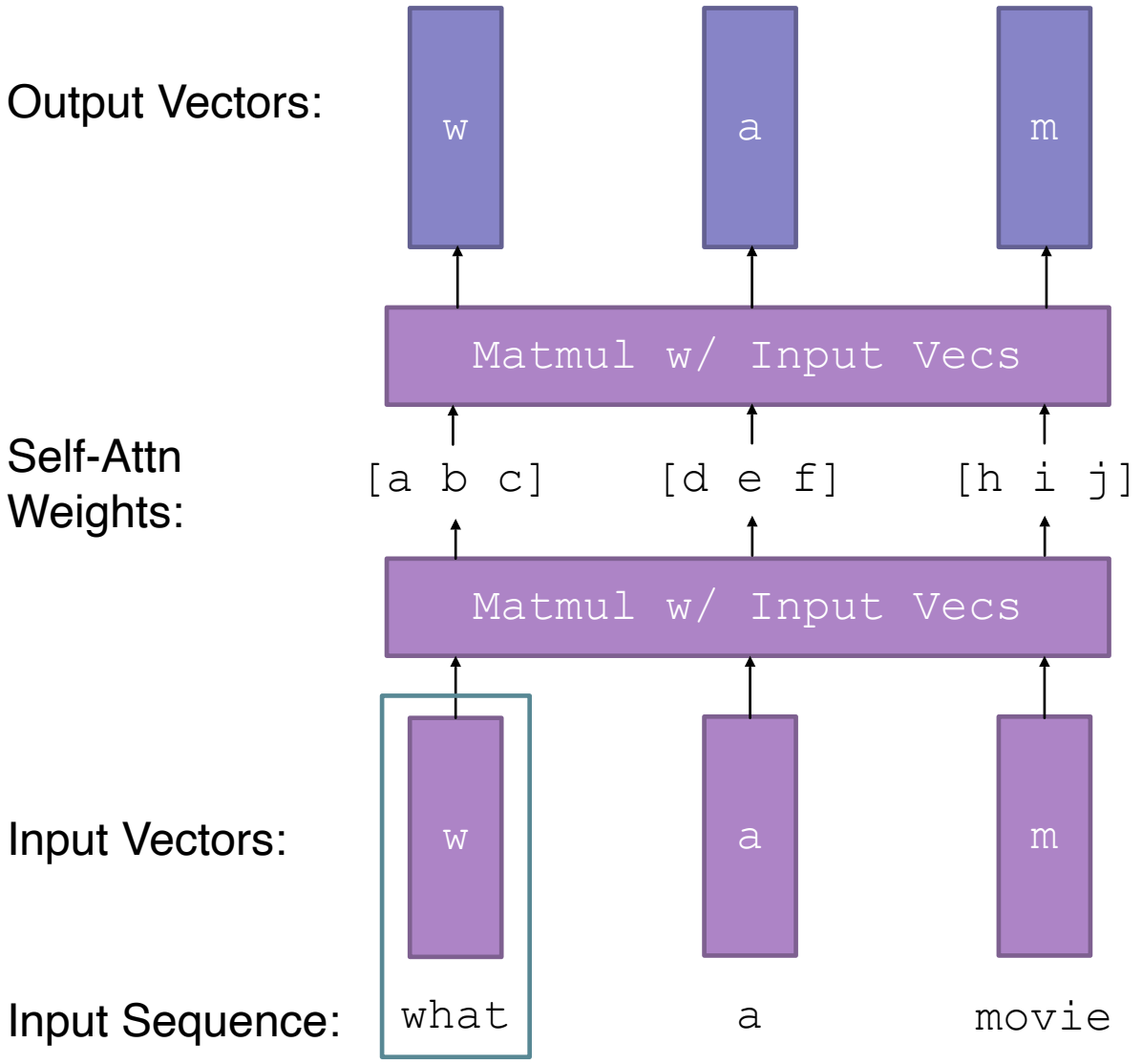
# Transformer Model

Right now, we are encoding the sequence using RNN cells; instead, we can use a Transformer encoder

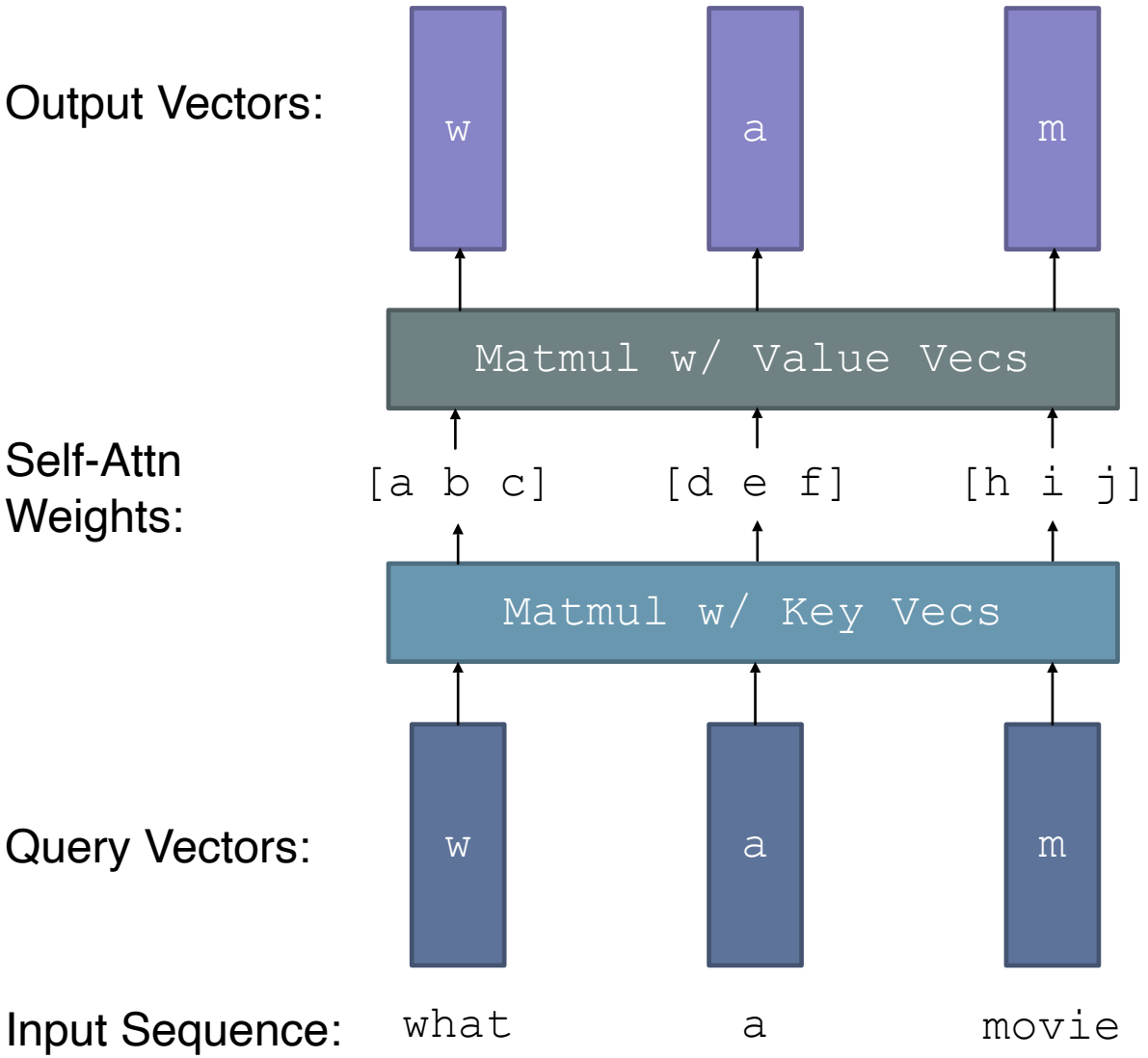
- Uses self-attention to encode deeper and deeper representations of input sequence
- Unique consideration 1: we must add positional encodings as Transformers are permutation invariant
- Unique consideration 2: we must pass in padding mask to ignore vectors corresponding to pad tokens during self-attention calculation

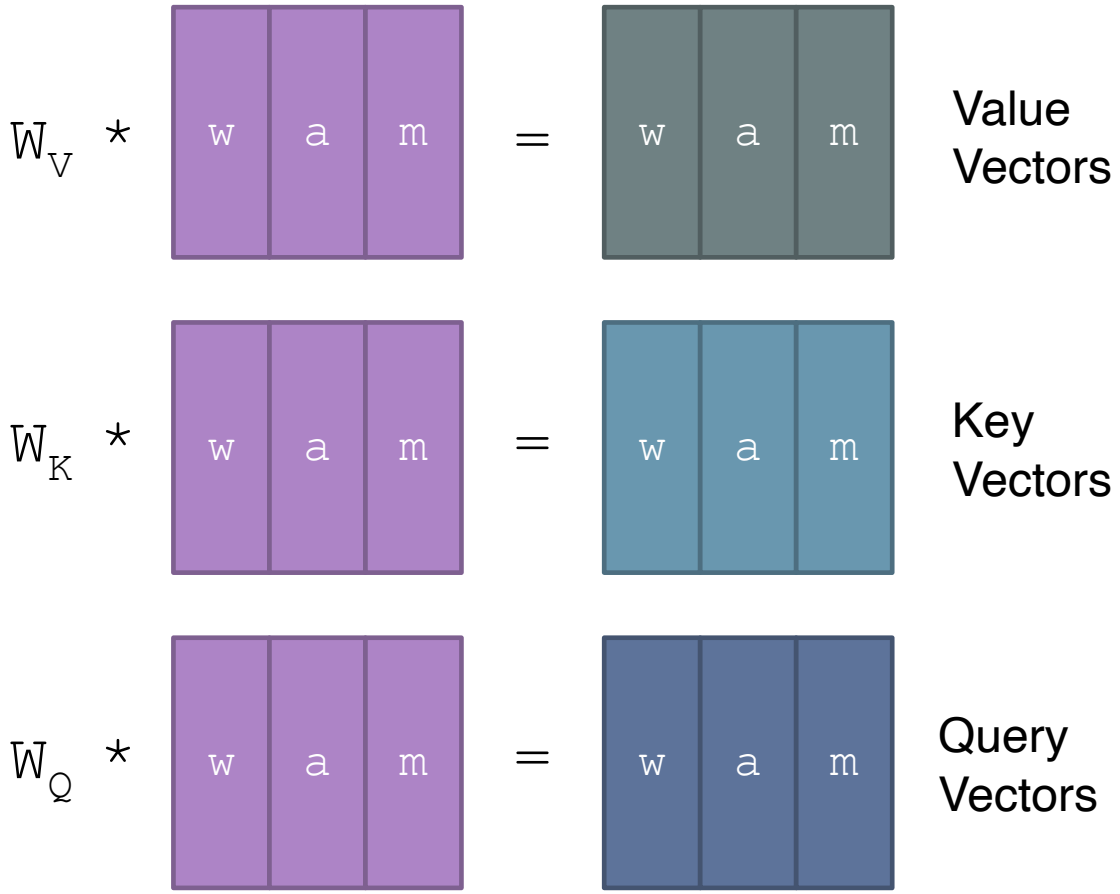
We will replace RNN w/ Transformer encoder, keeping all else the same

# Visual Intuition of Self-Attention



# Visual Intuition of Self-Attention (Reality)



Before Coding Up Transformer,
Let's Revisit Attention RNN Results…

# Understanding Model Decision Using Attention Weights

We can consider the attention weights as a way of understanding the model's "thinking"

More "important" words w.r.t. the prediction are given higher attention weight!

# Code Sample 2: Attention Visualization

# Live Coding 5: Transformer Model

# Bugs Summary 3

1. Low learning rate resulting in loss not going down (or so it seemed)
    1. This time: 1e-5 far too low—loss actually *was* going down, but too slowly
    2. In general: plot loss for a couple epochs—if it's going down very slowly, increase by an order of magnitude

Note: we have encountered three unique bugs causing loss to not go down—"loss not going down" is the most notoriously difficult bug to solve!

- Logical error in model architecture
- Logical error in loss or gradient computation
- Simply: need more hyperparameter search

# Model 4: T5 LLM

# T5 LLM and Prompt Engineering

LLMs are capable of performing well on a wide array of NLP tasks

T5 is an LLM which formulates many NLP problems as text-to-text

- To do translation, literally pass to the model: “Translate English to Spanish: [English sentence]”
- To do sentiment classification, prepend “sst2 sentence:” (i.e. Stanford Sentiment Treebank)

This can be thought of as a form of prompt engineering

- Prompt engineering is modifying the prompt/question to an LLM to get better or more desirable results
- E.g., adding “Let’s think through it step by step” increases LLM accuracy!

Before Running T5,
Let’s Revisit Transformer Results…

# Code Sample 3: T5 LLM