# WEEK 2 – SPATIAL/TEMPORAL ANALYSIS

# TERMS FROM LAST WEEK

Neural Network

CNN

LSTM

# NEW TERMS

Spatial – relating to position/location (i.e. pixels in an image, DNA sequences)

Temporal/sequential – relating to time/sequences (i.e. text, DNA sequences, videos)

# ANALYSIS

- When we analyze these types of data, the spatial/temporal context matters!
- Pixels at certain locations may be good predictors of image type
- One word preceding a second word may be an important in a text classification problem
- We need methods that can model these relationships, rather than just treating each feature independently

# TOPICS

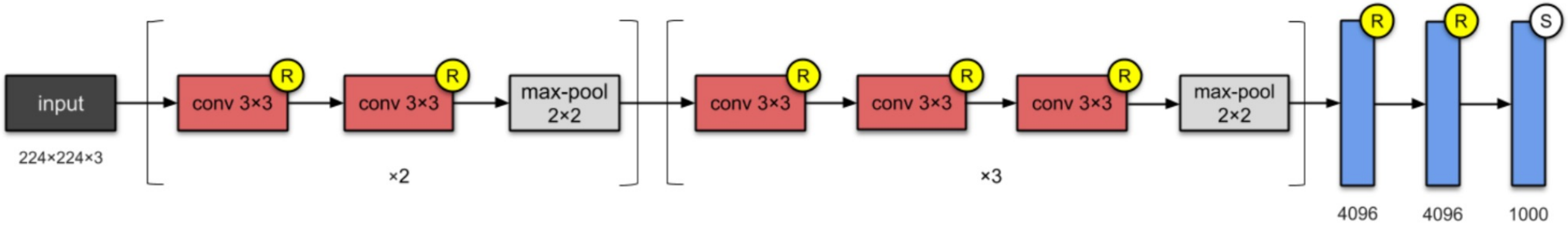
- More on the CNN
- More on the LSTM
- Combining CNNs/LSTMs
- Embeddings
- Transformers
- Self-supervision

# CNN

- Over the years, researchers have designed different ways to structure CNNs
- These different architectures can be valuable depending on the task
- We will discuss 3: VGG-16, Inception, and ResNet

# VGG-16

- This is a pretty straightforward architecture
- Many layers
- Good for when a smaller CNN (i.e. our tutorial) is not performing adequately or when you need extremely highly accuracy values

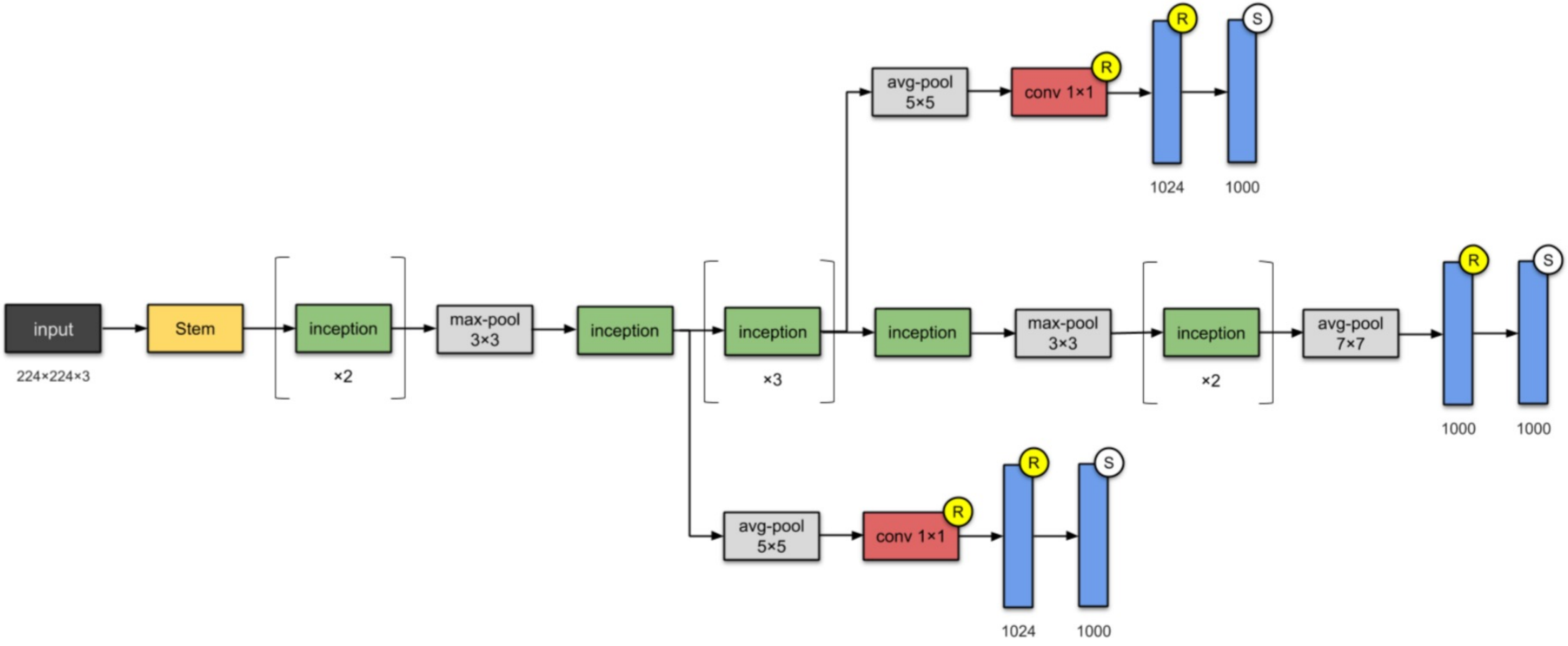

input
224×224×3
conv 3×3
R
conv 3×3
R
max-pool
2×2
×2
conv 3×3
R
conv 3×3
R
conv 3×3
R
max-pool
2×2
×3
R
R
S
4096
4096
1000

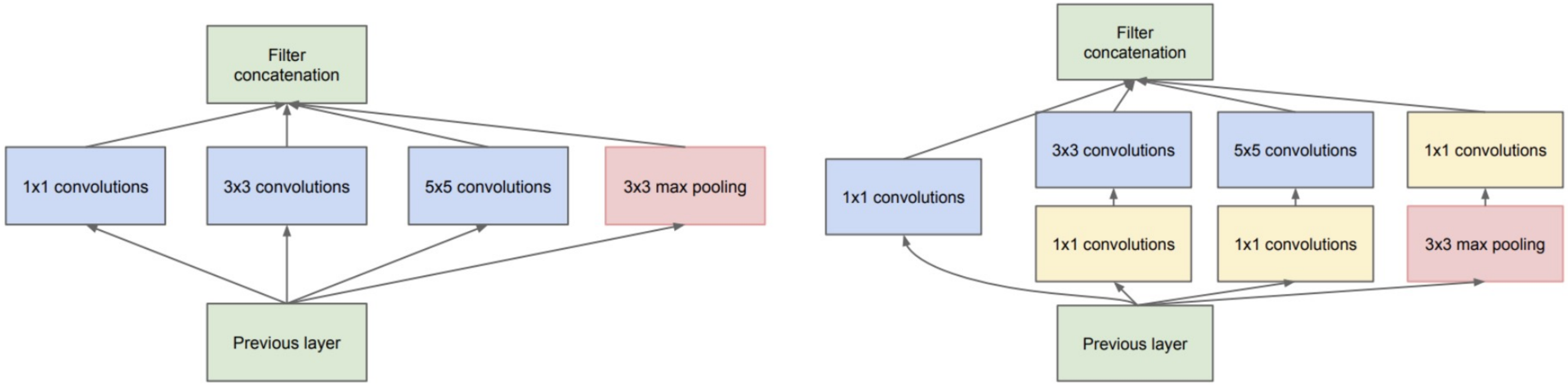
# INCEPTION

Inception is an architecture that merges outputs from kernels with different sizes (1x1, 3x3, 5x5).

This captures information at different resolutions (different perspectives)

Multiple classifiers are used for optimizing the lower part of the network and for regularization


input
224×224×3
Stem
inception
×2
max-pool
3×3
inception
inception
×3
avg-pool
5×5
conv 1×1
R
R
S
1024
1000
inception
max-pool
3×3
inception
×2
avg-pool
7×7
R
S
1000
1000
avg-pool
5×5
conv 1×1
R
R
S
1024
1000




(a) Inception module, naïve version

(b) Inception module with dimension reductions

# RESNETS

- Skips layers early on – then adds them back as we get closer to the ideal solution
- By reducing the # of neurons, we prevent our gradient from becoming too small
- Vanishing gradients prevent effective learning
- ResNets allow for many, many, MANY layers to be used effectively

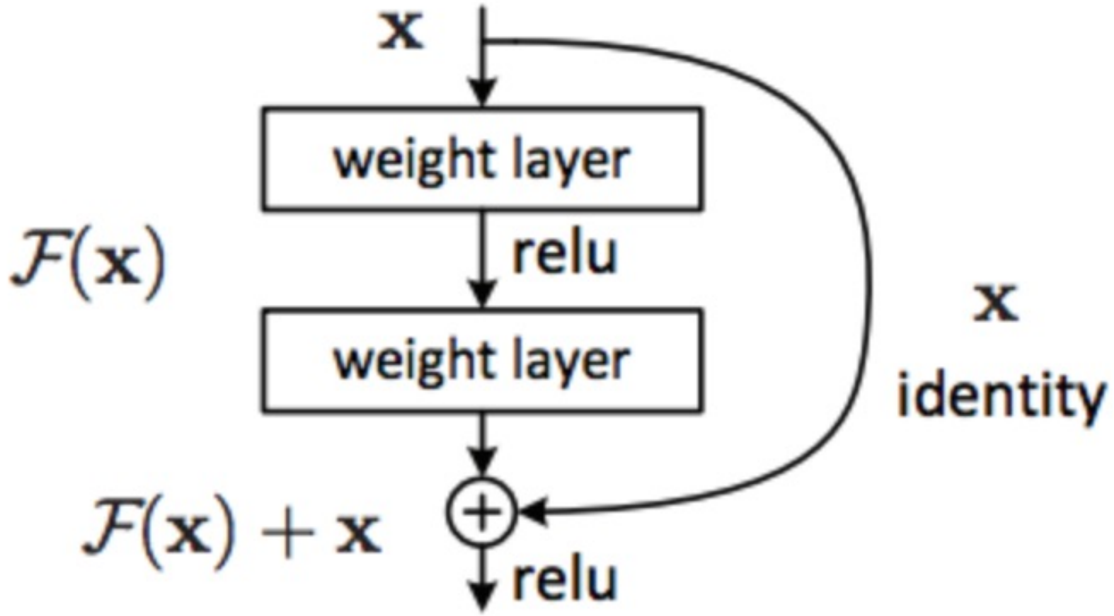
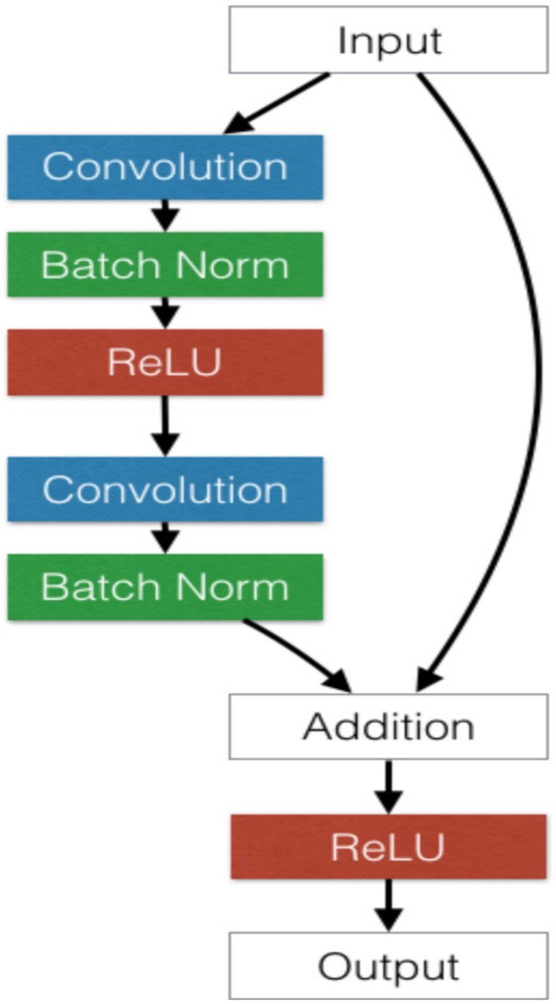



Figure 2. Residual learning: a building block.


Input
Convolution
Batch Norm
ReLU
Convolution
Batch Norm
Addition
ReLU
Output

# RESNETS

ResNets are often used for complex image datasets

ResNets can be used for other complex problems where we need many many layers

Not just images!

# LSTM

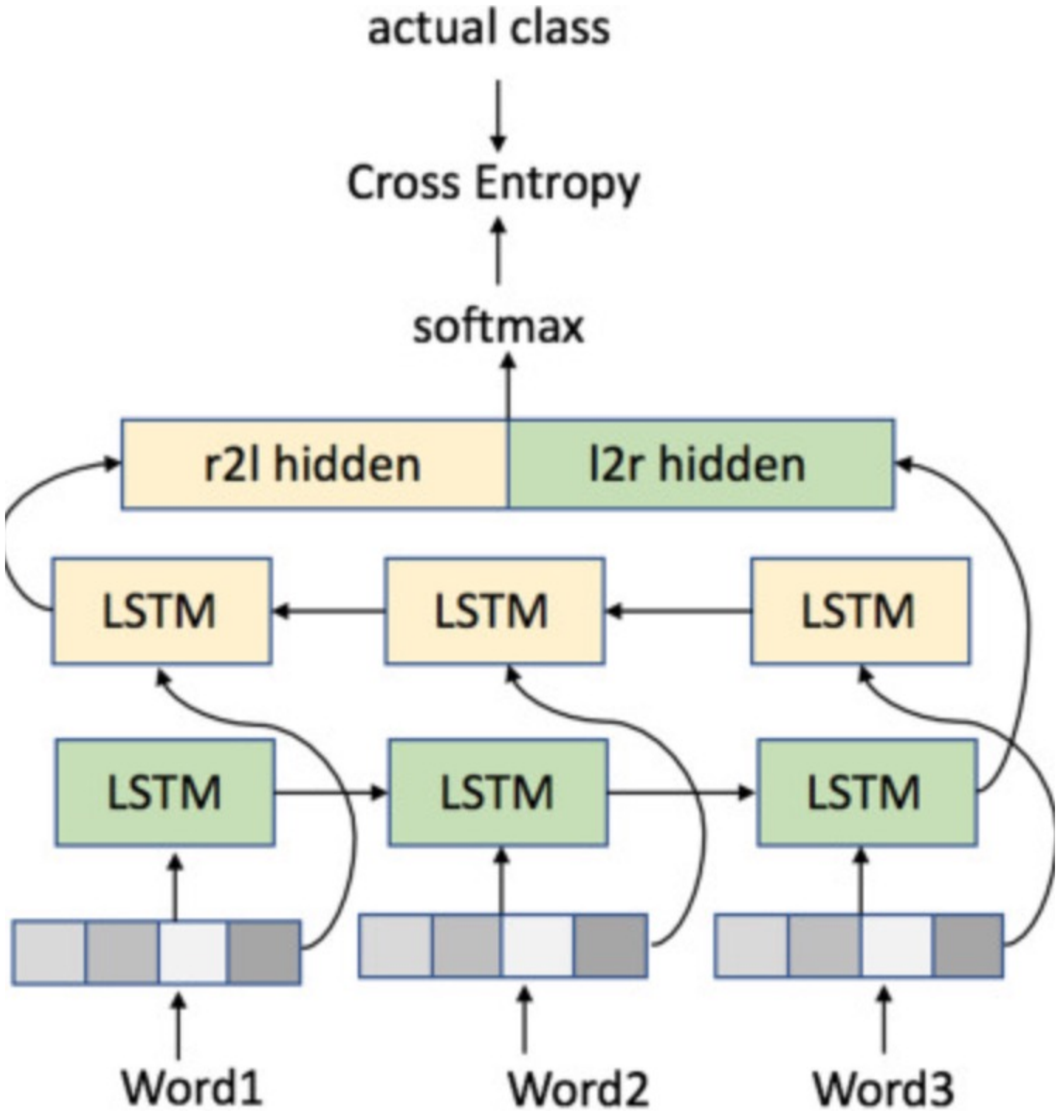
In most cases, you will use the standard LSTM architecture on PyTorch

One modification you can make: bi-directionality

Learning things from what comes before AND what comes after

Pytorch has this has a Boolean variable

# BI-DIRECTIONALITY



- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

Inputs: input, (h_0, c_0)

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape (*num_layers * num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. If the LSTM is bidirectional, num_directions should be 2, else it should be 1. If `proj_size > 0` was specified, the shape has to be (*num_layers * num_directions, batch, proj_size*).
- **c_0** of shape (*num_layers * num_directions, batch, hidden_size*): tensor containing the initial cell state for each element in the batch.

  If (*h_o, c_o*) is not provided, both **h_0** and **c_0** default to zero.

# EMBEDDINGS

An embedding is a vector that represents features about a word/data point

In the language example, these features include context/relationships with other words

We input these embeddings into neural networks to learn things about our data (as we saw with the transformers example)

Pytorch has an embedding module (nn.Embedding) that allows us to learn these embeddings for our datasets

# PYTORCH EMBEDDINGS

- We can take these embeddings and input them into LSTM
- LSTM will generate features using the sequence/sentence
- These features go into an ANN for prediction
- Then, we can optimize the ANN, the LSTM and the embedding model using a loss function and an optimization function

# COMBINATIONS

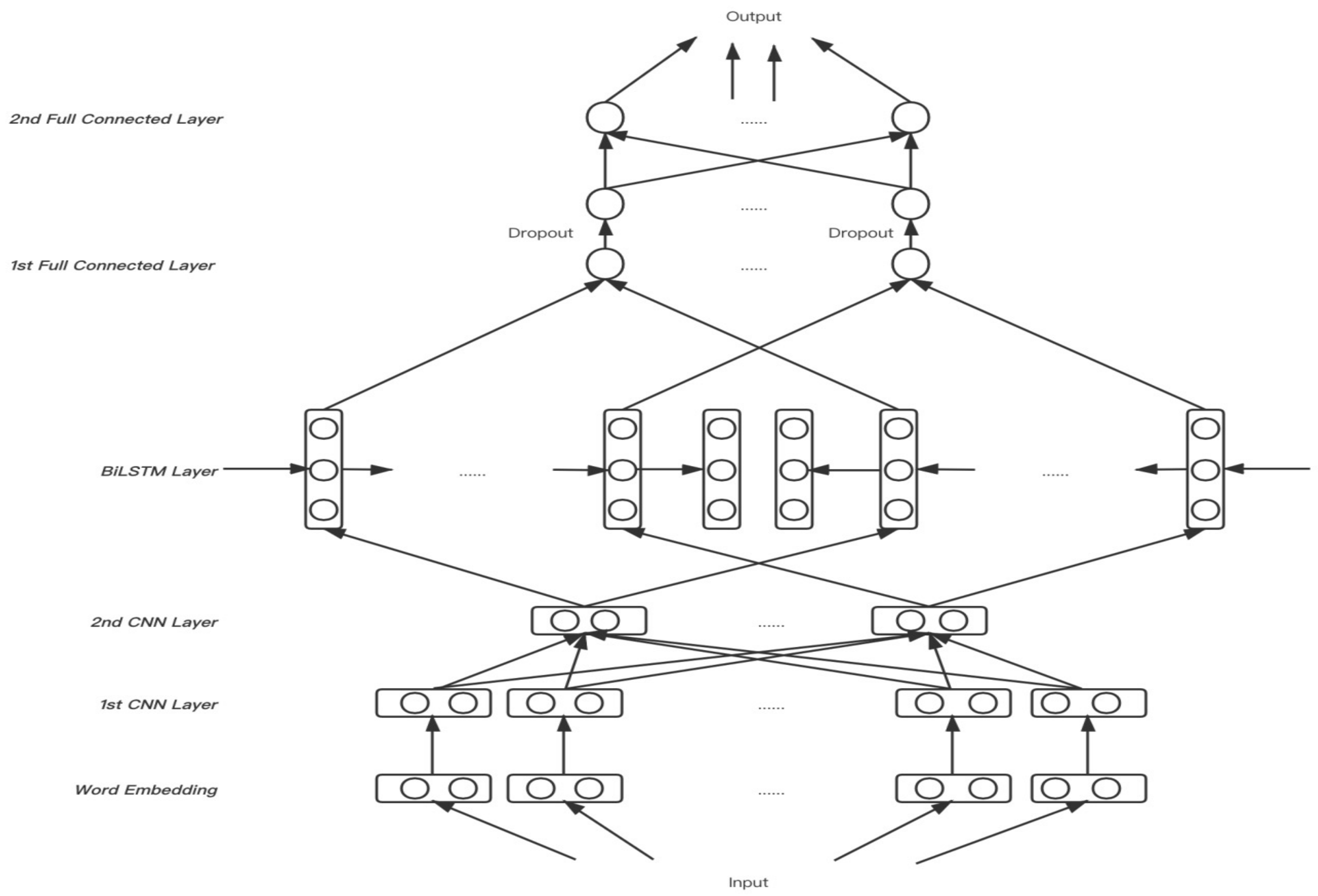
- You can use a combined LSTM-CNN approach when data has spatial/temporal context
- For example, a video is a sequence of still images
- The CNN learns from the position of the features at certain points and time
- The LSTM learns from the sequence
- Better predictions!

# POS TAGGING

Part-of-speech tagging – predict the part of speech for words using information from text/literature

LSTM to learns the relationships between words in a sequence

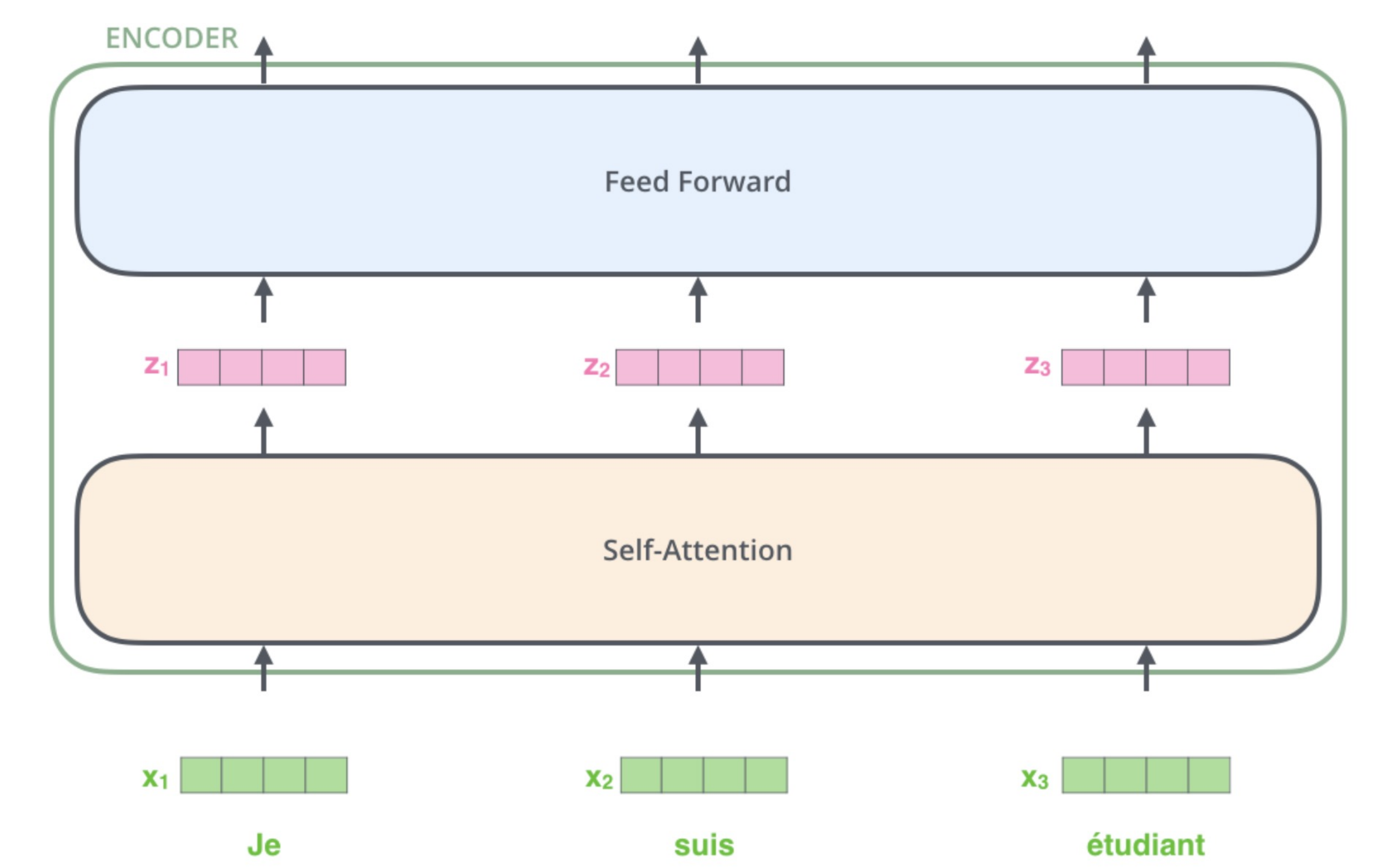
CNN learns info on letters (i.e. -ly suffix == adverb)

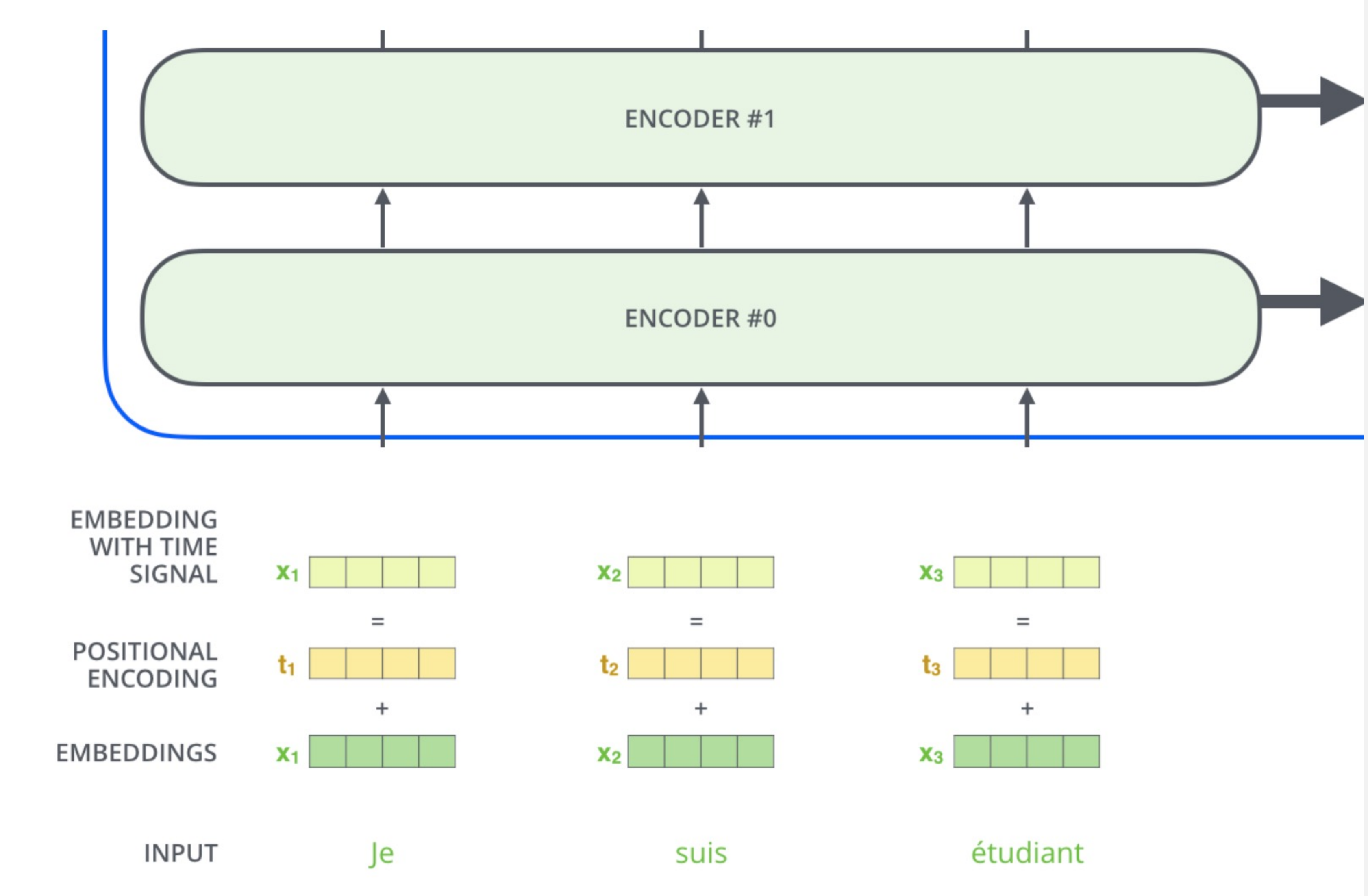


Output
2nd Full Connected Layer
Dropout
Dropout
1st Full Connected Layer
BiLSTM Layer
2nd CNN Layer
1st CNN Layer
Word Embedding
Input

# TRANSFORMERS

- Transformers use a “self-attention” to speed up sequential learning tasks
- Instead of recurrence (i.e. LSTM), transformers use relationships within sequences
- No need for memory
- Sequences (i.e. sentences) do not have to be in order
- This method is used in BERT (NLP!)


ENCODER
Feed Forward
$z_1$
$z_2$
$z_3$
Self-Attention
$x_1$
$x_2$
$x_3$
Je
suis
étudiant



ENCODER #1
ENCODER #0
EMBEDDING WITH TIME SIGNAL
$x_1$
$x_2$
$x_3$
=
POSITIONAL ENCODING
$t_1$
$t_2$
$t_3$
+
EMBEDDINGS
$x_1$
$x_2$
$x_3$
INPUT
Je
suis
étudiant

# TRAINING

So we have learned about these incredible transformers, but what do they actually predict?

These are often used in an unsupervised way – to generate representations of data

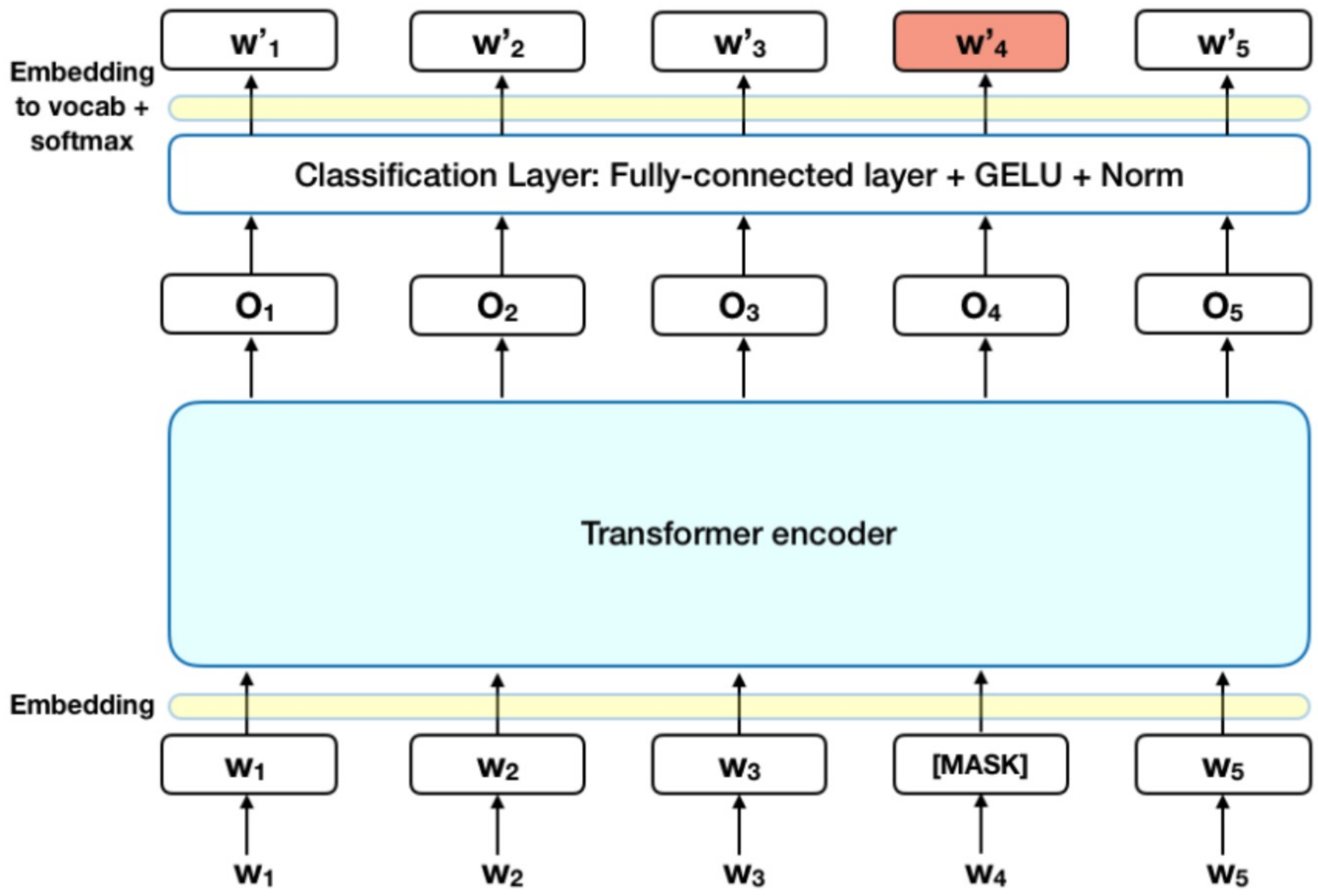
These representations contain key info/relationships/features of the dataset

We can fine-tune these representations for many other tasks

No need to retrain for each one!

# SELF-SUPERVISION

- This is a more general technique that is often used to train transformer models (i.e. BERT)
- This technique involves using parts of the data as the "label"
- If we can use some of the data to predict other part of the data, we learn relationships that "represent" that dataset


w'1
w'2
w'3
w'4
w'5
Embedding to vocab + softmax
Classification Layer: Fully-connected layer + GELU + Norm
O1
O2
O3
O4
O5
Transformer encoder
Embedding
w1
w2
w3
[MASK]
w5
w1
w2
w3
w4
w5

# SELF-SUPERVISION

- Self-supervision is not limited to sequential data
- For example, self-supervision has been used for images
- Use images to reconstruct the same images after a rotation
- Use CD3/CD8 expression to predict CD4/CD45
- Use self-supervision to learn representations of big datasets - useful for many other tasks!

# QUESTIONS + PROJECT DISCUSSION