# Recurrent Neural Networks

CS 780/880 Natural Language Processing Lecture 14

Samuel Carton, University of New Hampshire

# Last lecture

Word vector models

- Word2Vec
  - CBOW
  - Skip-gram
- GloVe

Word vectors in classification

- Padding
- Collation
- Centroids

# Word vectors & composition

Word vectors are pretty cool

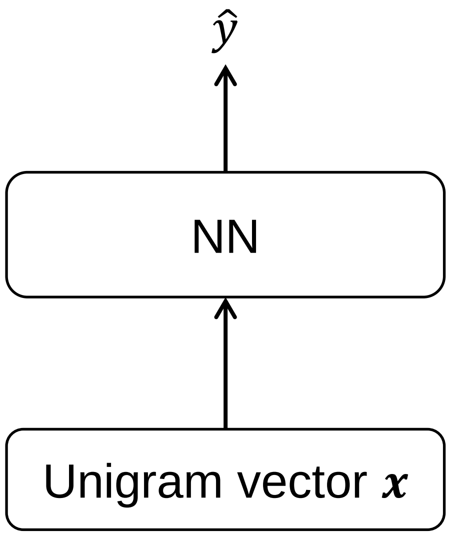
- Semantic similarity
- Analogies

But ultimately, NNs need **fixed-length** input, and it’s not obvious how to **compose** a variable-length sequence of word vectors into a single **document vector**

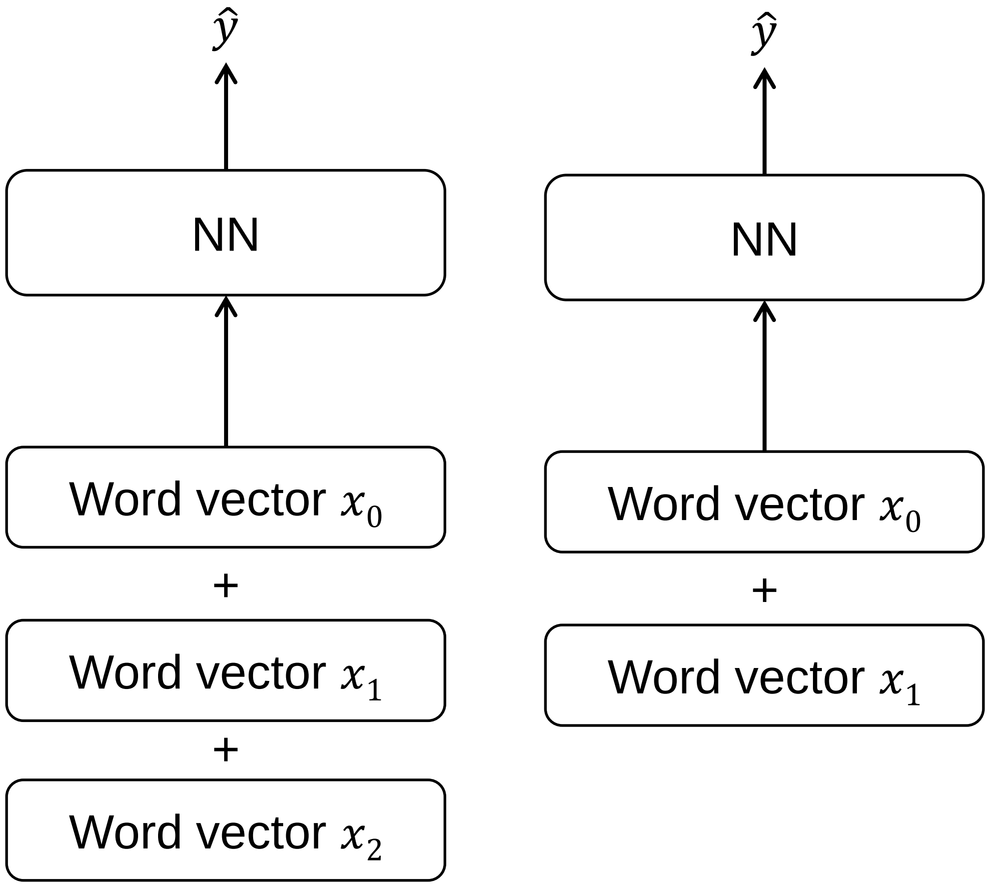
Just taking the centroid netted us some disappointing results

# Neural text classification so far

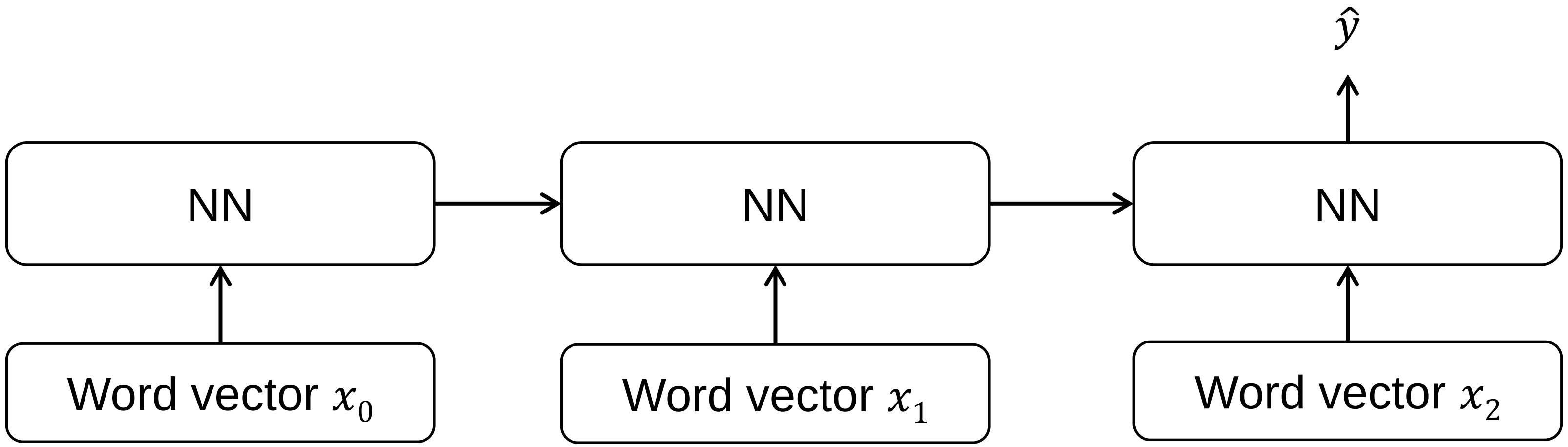
With unigram/TF-IDF vector

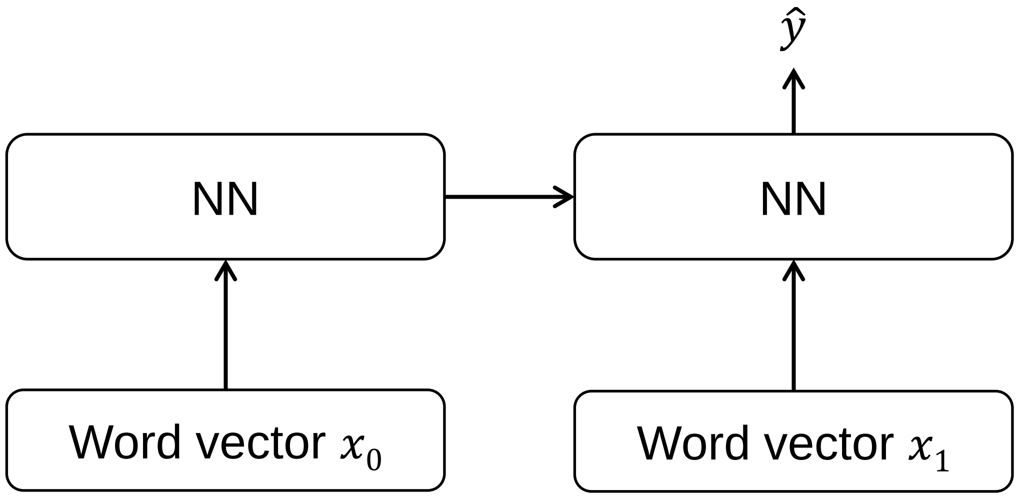



With word vector centroids



# How to properly use word vectors?

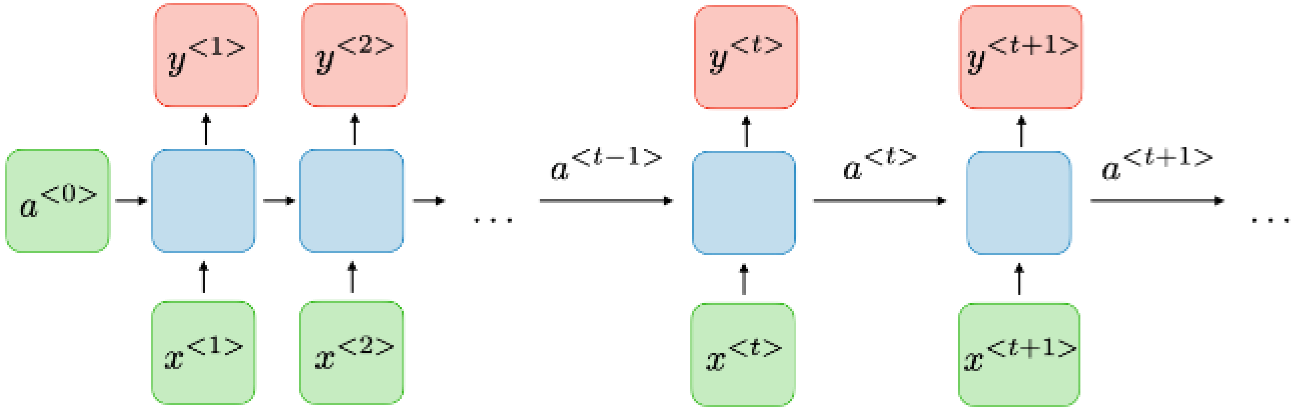


# Recurrent Neural Nets (RNNs)

**Basic idea:** the model runs over one word at a time, producing one or more **hidden state vectors** (aka activation vector) which it passes to itself when it looks at the next word.

**Analogous to humans:** read one word at a time and remember **whatever you need to remember** from word to word, to understand the meaning of the whole text.




Diagrams from https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

- Very nice cheat-sheet for RNNs

# Recurrent Neural Nets (RNNs)

$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

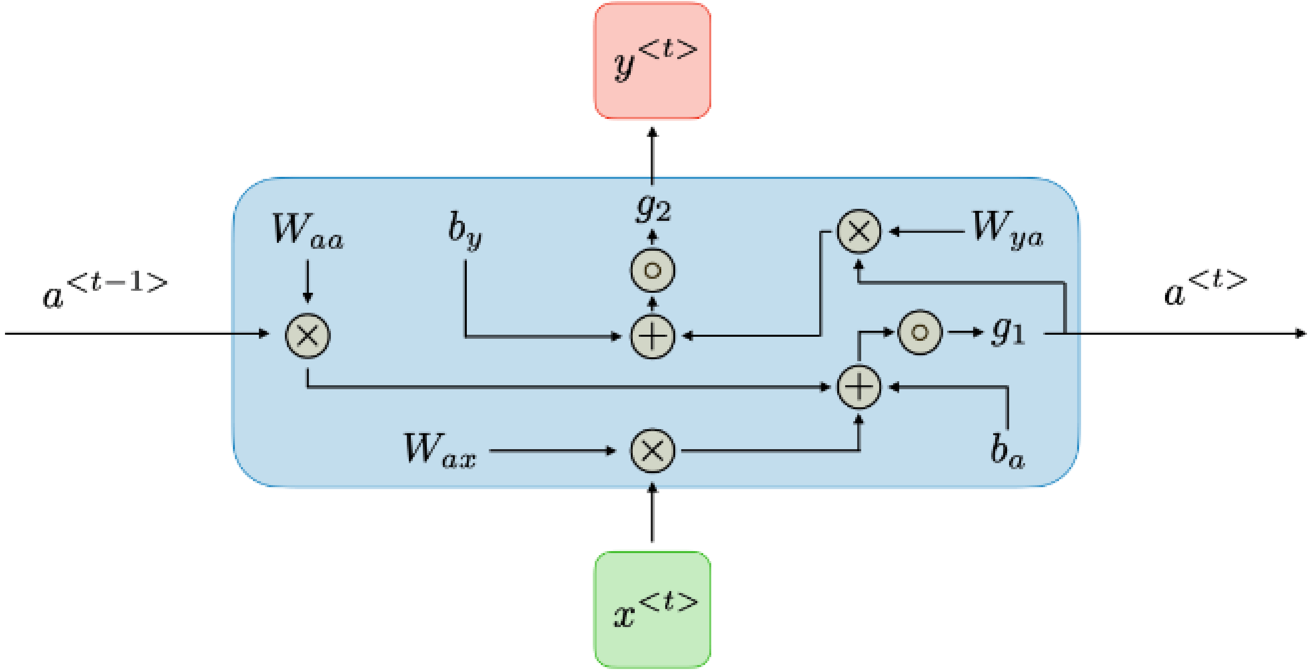
**More generally:**

$$a^t = f(a^{t-1}, x^t)$$

$$\hat{y}^t = g(a^{t-1} \; or \; a^t, x^t)$$

So $a^t$ is what gets **remembered** from word to word, and $\hat{y}^t$ is what gets **outputted** from word to word.

And models learn to remember what they need to remember, via objective functions on $\hat{y}^t$




(a tad over-specific, IMHO)

# Example: "dumb" insult detector

Say you are trying to train an RNN to read a whole text and predict "yes" if the text has the word "dumb" (or a synonym like "moronic") in it, and "no" if not

$$a^t = f(a^{t-1}, x^t)$$

$$\hat{y}^t = g(a^{t-1} \text{ or } a^t, x^t)$$

Then, $a^t$ can just be a 1 or a 0, indicating "has one of these words been found before?"

And $a^t = f(a^{t-1}, x^t)$ can be defined as $(a^{t-1} = 1 \text{ or } x^t = \text{"dumb"})$

And then finally $\hat{y}^t$ could just be equal to $a^t$, and we would put an objective on just the final $\hat{y}^t$ ($\hat{y}^N$), encouraging it to be 1 if there is a "dumb" somewhere in the text.

**Challenge:** how could we detect whether a given $x^t$ = "dumb" or some similar word?

# Example: “dumb” insult detector

Say you are trying to train an RNN to read a whole text and predict “yes” if the text has the word “dumb” (or a synonym like “moronic”) in it, and “no” if not

$$a^t = f(a^{t-1}, x^t)$$

$$\hat{y}^t = g(a^{t-1} \; or \; a^t, x^t)$$

Then, $a^t$ can just be a 1 or a 0, indicating “has one of these words been found before?”

And $a^t = f(a^{t-1}, x^t)$ can be defined as $(a^{t-1} = 1 \; or \; x^t = \text{"dumb"})$

And then finally $\hat{y}^t$ could just be equal to $a^t$, and we would put an objective on just the final $\hat{y}^t$ ($\hat{y}^N$), encouraging it to be 1 if there is a “dumb” somewhere in the text.

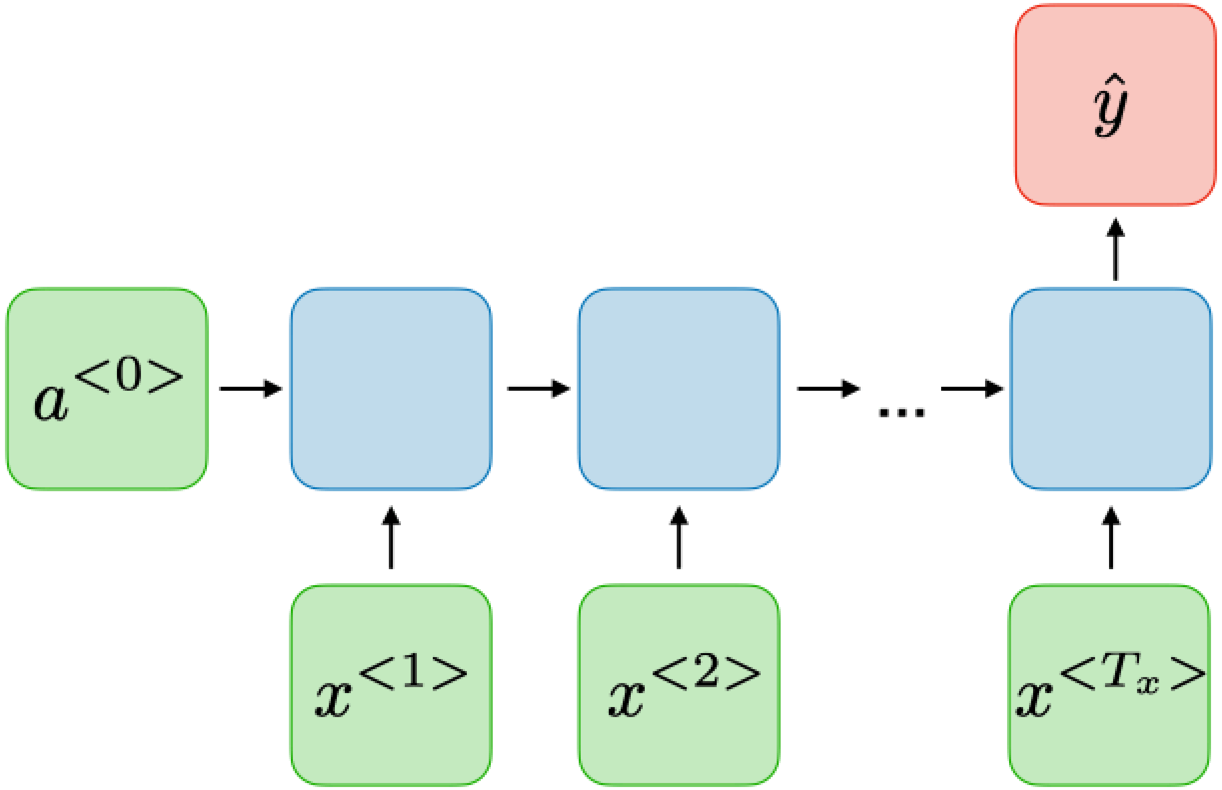
**Challenge:** how could we detect whether a given $x^t$ = “dumb” or some similar word?

- ¡Word vectors!

# Different RNN types

**Many-to-one**

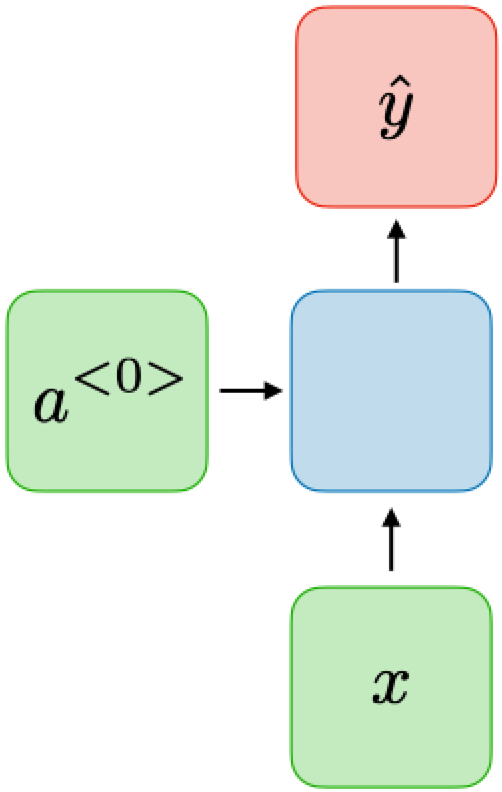
- Most text classification is this



# Different RNN types

**One-to-one**

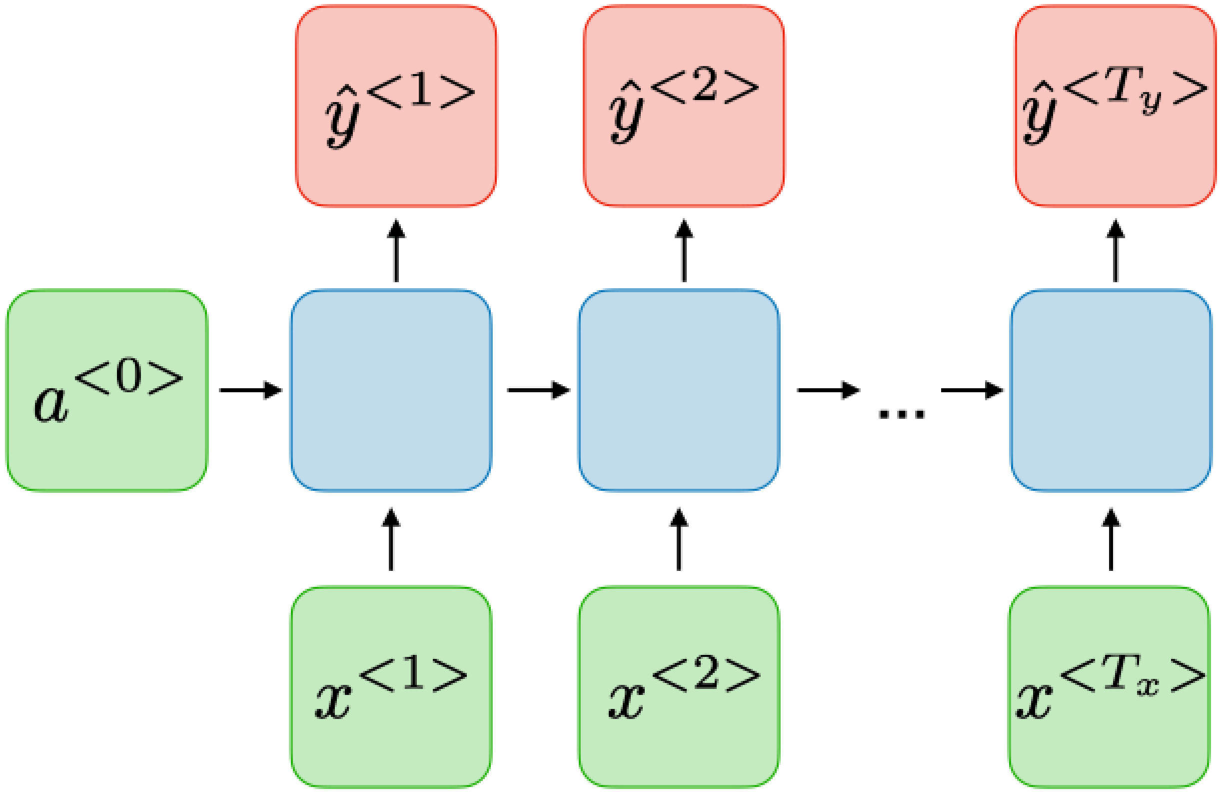
- A conventional (feedworward) neural net could be described as this



# Different RNN types

**Many-to-many**

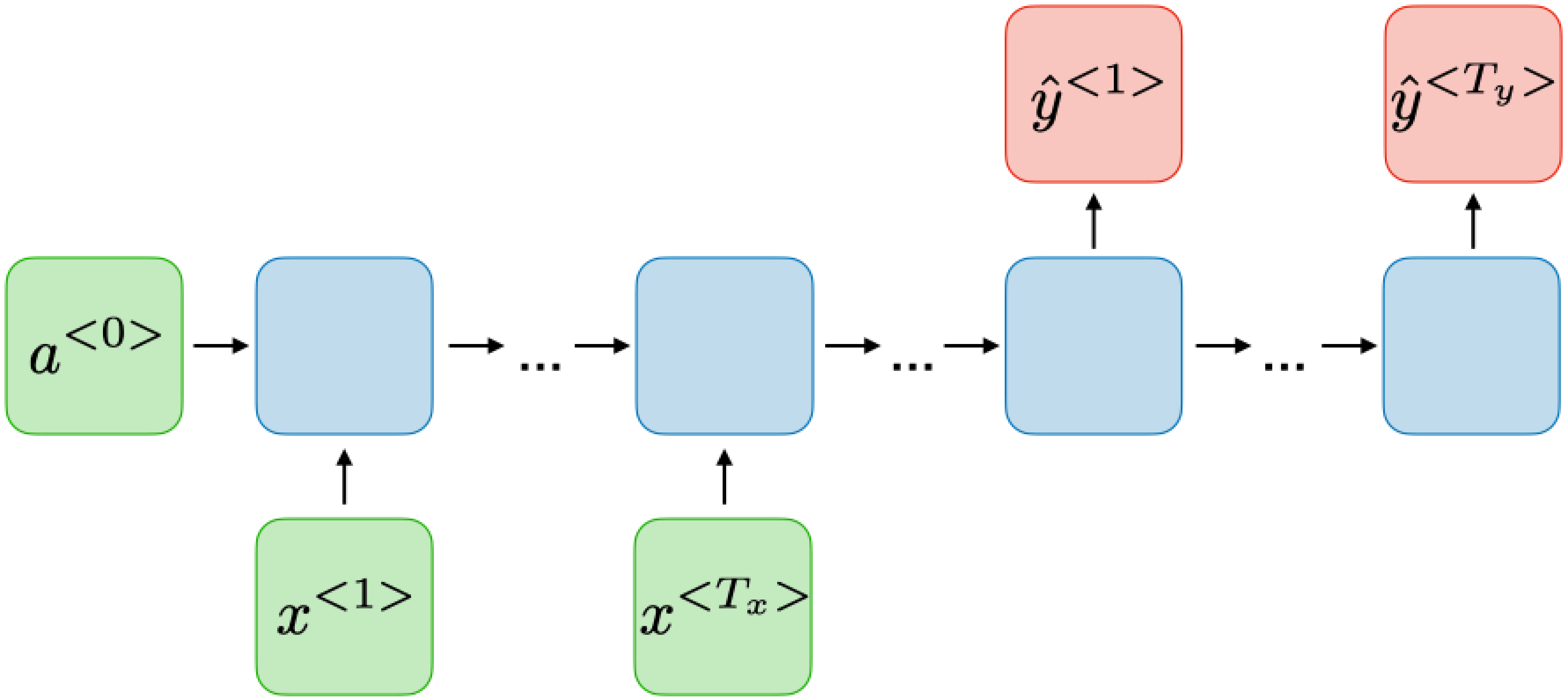
- POS tagging would be an example of this



# Different RNN types

**Many-to-many** $(Tx \neq T_y)$

- Variant of many-to-many where there are inputs and outputs on different cells
- Machine translation is the main example of this



# Vanishing gradients

RNNs are like a feedforward neural net being applied **horizontally** across each word of the text, rather than **vertically** across a flat representation of the text

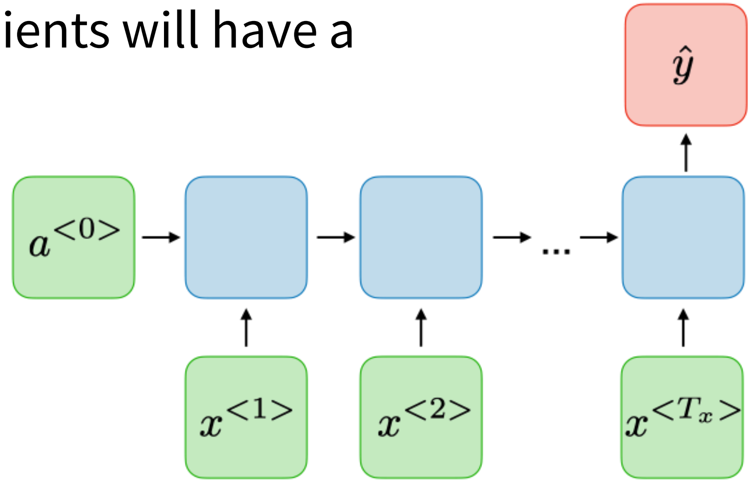
- Such as the centroid of the word vectors in the text, which is what we tried last lecture
- But same parameters at each layer, rather than different weight tensor

Like FFNNs, RNNs have problems with **vanishing gradients**

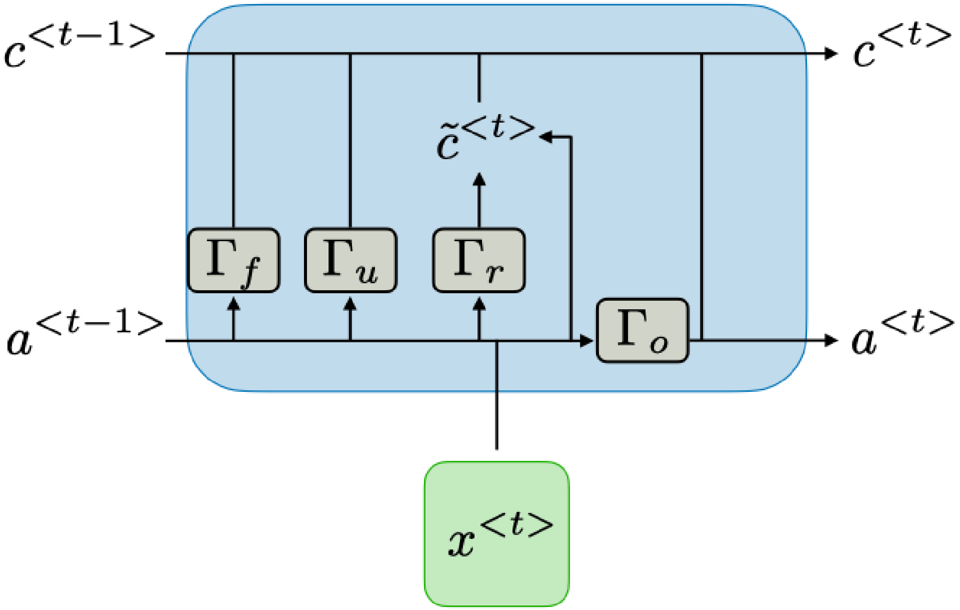
If you apply an objective only to $\hat{y}$ at the end, the gradients will have a tough time training the cells toward the beginning

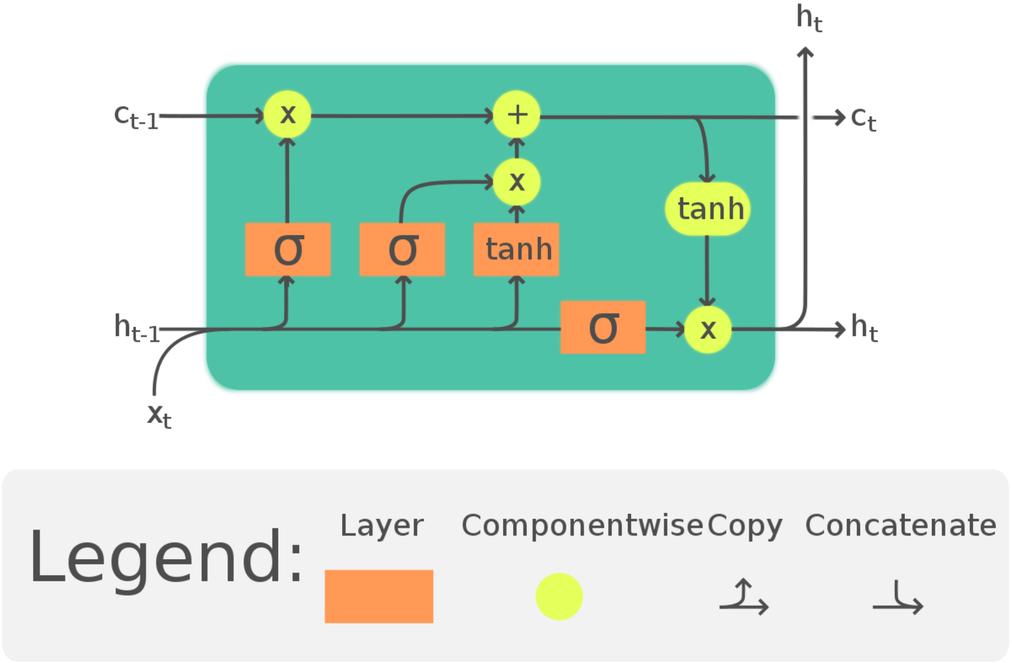
Called **catastrophic forgetting**

- Like losing focus on a sentence before you're done reading it



# Long Short-Term Memory (LSTM)

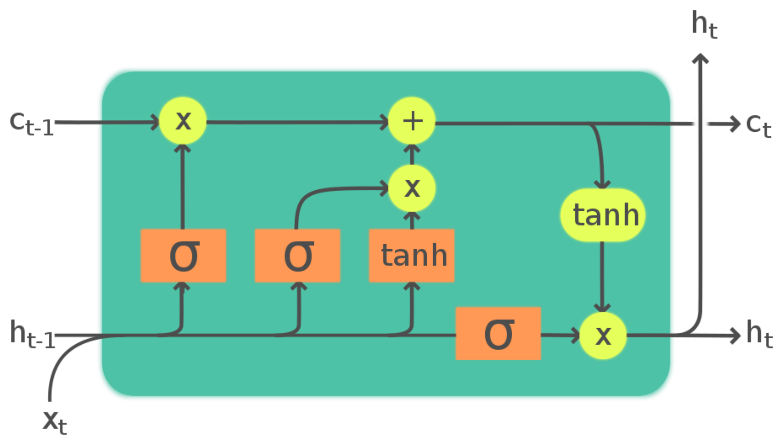


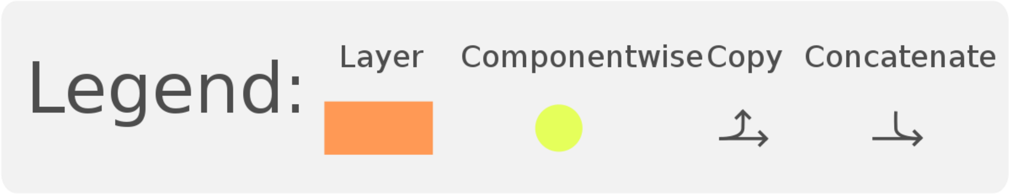




https://en.wikipedia.org/wiki/Long_short-term_memory

# Long Short-Term Memory (LSTM)

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \sigma_h(c_t)
\end{aligned}
$$




- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices and bias vector parameters which need to be learned during training

# Long Short-Term Memory (LSTM)

Seem arbitrary? It kind of is.

Valaee et al. (2017) shows that different kinds of RNNs (GRUs, etc) have similar performance

- https://arxiv.org/pdf/1801.01078.pdf

So the exact internal equations aren't that important, more the idea of a persistent memory vector (or vectors) that can be added to or subtracted from based on new $x^t$'s, in a way that **can be learned** from the objective function.

# Loading GloVe vectors with Gensim

```python
import gensim.downloader as api
from pprint import pprint

# There's a bunch of models available.
pprint(list(api.info()['models'].keys()))
```

```
['fasttext-wiki-news-subwords-300',
 'conceptnet-numberbatch-17-06-300',
 'word2vec-ruscorpora-300',
 'word2vec-google-news-300',
 'glove-wiki-gigaword-50',
 'glove-wiki-gigaword-100',
 'glove-wiki-gigaword-200',
 'glove-wiki-gigaword-300',
 'glove-twitter-25',
 'glove-twitter-50',
 'glove-twitter-100',
 'glove-twitter-200',
 '__testing_word2vec-matrix-synopsis']
```

# Loading GloVe vectors with Gensim

```python
# 'glove-wiki-gigaword-100' is probably what we want
pprint(api.info()['models']['glove-wiki-gigaword-100'])
```

```
{'base_dataset': 'Wikipedia 2014 + Gigaword 5 (6B tokens, uncased)',
 'checksum': '40ec481866001177b8cd4cb0df92924f',
 'description': 'Pre-trained vectors based on Wikipedia 2014 + Gigaword 5.6B '
                'tokens, 400K vocab, uncased '
                '(https://nlp.stanford.edu/projects/glove/).',
 'file_name': 'glove-wiki-gigaword-100.gz',
 'file_size': 134300434,
 'license': 'http://opendatacommons.org/licenses/pddl/',
 'num_records': 400000,
 'parameters': {'dimension': 100},
 'parts': 1,
 'preprocessing': 'Converted to w2v format with `python -m '
                  'gensim.scripts.glove2word2vec -i <fname> -o '
                  'glove-wiki-gigaword-100.txt`.',
 'read_more': ['https://nlp.stanford.edu/projects/glove/',
               'https://nlp.stanford.edu/pubs/glove.pdf'],
 'reader_code': 'https://github.com/RaRe-Technologies/gensim-data/releases/download/glove-wiki-gigaword-100/__init__.py'}
```

# Loading GloVe vectors with Gensim

```
# 'glove-wiki-gigaword-100' is probably what we want
pprint(api.info()['models']['glove-wiki-gigaword-100'])
```

```
{'base_dataset': 'Wikipedia 2014 + Gigaword 5 (6B tokens, uncased)',
 'checksum': '40ec481866001177b8cd4cb0df92924f',
 'description': 'Pre-trained vectors based on Wikipedia 2014 + Gigaword 5.6B '
                'tokens, 400K vocab, uncased '
                '(https://nlp.stanford.edu/projects/glove/).',
 'file_name': 'glove-wiki-gigaword-100.gz',
 'file_size': 134300434,
 'license': 'http://opendatacommons.org/licenses/pddl/',
 'num_records': 400000,
 'parameters': {'dimension': 100},
 'parts': 1,
 'preprocessing': 'Converted to w2v format with `python -m '
                  'gensim.scripts.glove2word2vec -i <fname> -o '
                  'glove-wiki-gigaword-100.txt`.',
 'read_more': ['https://nlp.stanford.edu/projects/glove/',
               'https://nlp.stanford.edu/pubs/glove.pdf'],
 'reader_code': 'https://github.com/RaRe-Technologies/gensim-data/releases/download/glove-wiki-gigaword-100/__init__.py'}
```

```
vector_model = api.load('glove-wiki-gigaword-100')
```

```
[==================================================] 100.0% 128.1/128.1MB downloaded
```

# Loading GloVe vectors with Gensim

```python
print('Vector for "cat"')
print(vector_model['cat'])

print('\n"cat" to ID')
print(vector_model.key_to_index['cat'])

print('\nID to "cat"')
print(vector_model.index_to_key[5450])

# Note that we're using similarity here, not distance
print('\nWords with most similar vectors to "cat"')
vector_model.most_similar('cat')
```

```
Vector for "cat"
[ 0.23088    0.28283    0.6318    -0.59411   -0.58599    0.63255
  0.24402   -0.14108    0.060815  -0.7898    -0.29102    0.14287
  0.72274    0.20428    0.1407     0.98757    0.52533    0.097456
  0.8822     0.51221    0.40204    0.21169   -0.013109  -0.71616
  0.55387    1.1452    -0.88044   -0.50216   -0.22814    0.023885
  0.1072     0.083739   0.55015    0.58479    0.75816    0.45706
 -0.28001    0.25225    0.68965   -0.60972    0.19578    0.044209
 -0.31136   -0.68826   -0.22721    0.46185   -0.77162    0.10208
  0.55636    0.067417  -0.57207    0.23735    0.4717     0.82765
 -0.29263   -1.3422    -0.099277   0.28139    0.41604    0.10583
  0.62203    0.89496   -0.23446    0.51349    0.99379    1.1846
 -0.16364    0.20653    0.73854    0.24059   -0.96473    0.13481
 -0.0072484  0.33016   -0.12365    0.27191   -0.40951    0.021909
 -0.6069     0.40755    0.19566   -0.41802    0.18636   -0.032652
 -0.78571   -0.13847    0.044007  -0.084423   0.04911    0.24104
  0.45273   -0.18682    0.46182    0.089068  -0.18185   -0.01523
 -0.7368    -0.14532    0.15104   -0.71493  ]

"cat" to ID
5450

ID to "cat"
cat

Words with most similar vectors to "cat"
[('dog', 0.8798074722290039),
 ('rabbit', 0.7424427270889282),
 ('cats', 0.732300341129303),
 ('monkey', 0.7288709878921509),
 ('pet', 0.719014048576355),
 ('dogs', 0.7163872718811035),
 ('mouse', 0.6915250420570374),
 ('puppy', 0.6800068020820618),
 ('rat', 0.6641027331352234),
 ('spider', 0.6501135230064392)]
```

# Loading GloVe vectors with Gensim

```
unk_vector = vector_model.vectors.mean(axis=0)
pad_vector = np.zeros_like(unk_vector)

vector_model.add_vectors(['<unk>','<pad>'], [unk_vector,pad_vector])

print(vector_model.key_to_index['<unk>'])
print(vector_model.key_to_index['<pad>'])

print(vector_model.vectors.shape)
```

```
400000
400001
(400002, 100)
```

# Reading and preprocessing SST-2 dataset

```python
import nltk
from nltk import word_tokenize
nltk.download('punkt')

def tokenize(s):
  return word_tokenize(s.lower())

train_df['tokens'] = train_df['sentence'].apply(tokenize)
dev_df['tokens'] = dev_df['sentence'].apply(tokenize)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

```python
# And then we'll do the same token-to-ID lookup as before
def tokens_to_ids(tokens):
  return [vector_model.key_to_index[token] if token in vector_model else vector_model.key_to_index['<unk>'] for token in tokens]

train_df['input_ids'] = train_df['tokens'].apply(tokens_to_ids)
dev_df['input_ids'] = dev_df['tokens'].apply(tokens_to_ids)
display(dev_df)
```

# Reading and preprocessing SST-2 dataset

| | sentence | label | tokens | input_ids |
|---|---|---|---|---|
| **0** | it 's a charming and often affecting journey . | 1 | [it, 's, a, charming, and, often, affecting, j... | [20, 9, 7, 12387, 5, 456, 7237, 3930, 2] |
| **1** | unflinchingly bleak and desperate | 0 | [unflinchingly, bleak, and, desperate] | [101035, 12566, 5, 5317] |
| **2** | allows us to hope that nolan is poised to emba... | 1 | [allows, us, to, hope, that, nolan, is, poised... | [2415, 95, 4, 824, 12, 13528, 14, 7490, 4, 174... |
| **3** | the acting , costumes , music , cinematography... | 1 | [the, acting, ,, costumes, ,, music, ,, cinema... | [0, 2050, 1, 10349, 1, 403, 1, 22181, 5, 1507,... |
| **4** | it 's slow -- very , very slow . | 0 | [it, 's, slow, --, very, ,, very, slow, .] | [20, 9, 2049, 65, 191, 1, 191, 2049, 2] |
| **...** | ... | ... | ... | ... |
| **867** | has all the depth of a wading pool . | 0 | [has, all, the, depth, of, a, wading, pool, .] | [31, 64, 0, 4735, 3, 7, 27989, 3216, 2] |
| **868** | a movie with a real anarchic flair . | 1 | [a, movie, with, a, real, anarchic, flair, .] | [7, 1005, 17, 7, 567, 41588, 17056, 2] |
| **869** | a subject like this should inspire reaction in... | 0 | [a, subject, like, this, should, inspire, reac... | [7, 1698, 117, 37, 189, 11356, 2614, 6, 47, 20... |
| **870** | ... is an arthritic attempt at directing by ca... | 0 | [..., is, an, arthritic, attempt, at, directin... | [434, 14, 29, 57228, 1266, 22, 8044, 21, 63691... |
| **871** | looking aristocratic , luminous yet careworn i... | 1 | [looking, aristocratic, ,, luminous, yet, care... | [862, 21897, 1, 29085, 553, 203745, 6, 4917, 3... |

872 rows × 4 columns

# Dataset and DataLoader

```python
torch.random.manual_seed(1234)
first_train_batch = next(iter(train_dataloader))
print('First training batch:')
print(first_train_batch)

print('First training batch sizes:')
print({key:value.shape for key, value in first_train_batch.items()})
```

```
First training batch:
{'y': tensor([0, 0, 0, 1, 1, 1, 1, 1, 0, 0]), 'input_ids': tensor([[   307,     66,      3,  11114,   2720,      5,   5097,  31351, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [ 42131, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [    29,  51710,  37369,   2692,     12,   1144,   1003,     64,    317,
           2516,      2, 400001, 400001, 400001, 400001, 400001],
        [  2322, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [ 18519, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [    32,   3478,     17,      1,      5,    907,     81,    757,     59,
            403,     81,    107,     36,     33,   1435,    106],
        [    12,  21590,    244,  21609, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [     4,   6636,   1121,   3954,     17,    319,  15215,      5,    608,
          33619,     17,   9693,   3861, 400001, 400001, 400001],
        [    14,     70,    151,      7,   1005, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [    20,    965,   1369,     70,     33,     81,  12681,     25,      0,
           2816,  88552,     20,      9,  16031, 400001, 400001]])}
First training batch sizes:
{'y': torch.Size([10]), 'input_ids': torch.Size([10, 16])}
```

# Basic LSTM classification model

```python
class BasicLSTMClassifier(pl.LightningModule):
  def __init__(self,
               word_vectors:np.ndarray,
               num_classes:int,
               learning_rate:float,
               padding_id:int,
               lstm_hidden_size:int=100, # how big the inner vectors of the LSTM will be
               **kwargs):
    super().__init__( **kwargs)

    # We'll use the same PyTorch Embedding layer as before
    self.word_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(word_vectors),
                                                              freeze=True)

    self.lstm = torch.nn.LSTM(input_size = word_vectors.shape[1], # The LSTM will be taking in word vectors
                              hidden_size = lstm_hidden_size,
                              num_layers=1, # We'll talk about multi-layer and bidirectional LSTMs in a bit,
                              bidirectional=False, # but for now just 1 layer and 1-directional
                              batch_first=True # This is important. Set to False by default for some reason.
                              )

    # The output layer will act on the final hidden output from the LSTM, so its input size should be the LSTM hidden size
    self.output_layer  = torch.nn.Linear(lstm_hidden_size, num_classes)
    self.learning_rate = learning_rate
    self.padding_id = padding_id # we'll need this later
    self.train_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.val_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
```

# Basic LSTM classification model

```python
def forward(self, y:torch.Tensor, input_ids:torch.Tensor, verbose=False):

    inputs_embeds = self.word_embeddings(input_ids) #(batch size x sequence length x embedding size)

    input_lengths = (input_ids != self.padding_id).sum(dim=1).detach().cpu()

    packed_embeddings = pack_padded_sequence(inputs_embeds, input_lengths, batch_first=True, enforce_sorted=False)
    packed_output, (final_hidden, final_state) = self.lstm.forward(packed_embeddings)
    final_hidden = final_hidden.squeeze(0) #(batch size x lstm hidden size)

    py_logits = self.output_layer(final_hidden)
    py = torch.argmax(py_logits, dim=1)
    loss = torch.nn.functional.cross_entropy(py_logits, y, reduction='mean')
    return {'py':py,
            'loss':loss}
```

# Basic LSTM classification model

```
basic_lstm_model = BasicLSTMClassifier(word_vectors=vector_model.vectors,
                                       num_classes = 2,
                                       learning_rate = 0.001, #I'll typically start with something like 1e-3 for LSTMs
                                       padding_id = vector_model.key_to_index['<pad>'],
                                       lstm_hidden_size=100)
print('Model:')
print(basic_lstm_model)
```

```
Model:
BasicLSTMClassifier(
  (word_embeddings): Embedding(400002, 100)
  (lstm): LSTM(100, 100, batch_first=True)
  (output_layer): Linear(in_features=100, out_features=2, bias=True)
  (train_accuracy): MulticlassAccuracy()
  (val_accuracy): MulticlassAccuracy()
)
```

```
basic_lstm_trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=5,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.5,
    )
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
```

```
basic_lstm_trainer.fit(model=basic_lstm_model,
                       train_dataloaders=train_dataloader,
                       val_dataloaders=dev_dataloader)
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name            | Type               | Params
-------------------------------------------------------
0 | word_embeddings | Embedding          | 40.0 M
1 | lstm            | LSTM               | 80.8 K
2 | output_layer    | Linear             | 202
3 | train_accuracy  | MulticlassAccuracy | 0
4 | val_accuracy    | MulticlassAccuracy | 0
-------------------------------------------------------
81.0 K    Trainable params
40.0 M    Non-trainable params
40.1 M    Total params
160.325   Total estimated model params size (MB)
Validation accuracy: tensor(0.5000, device='cuda:0')
```

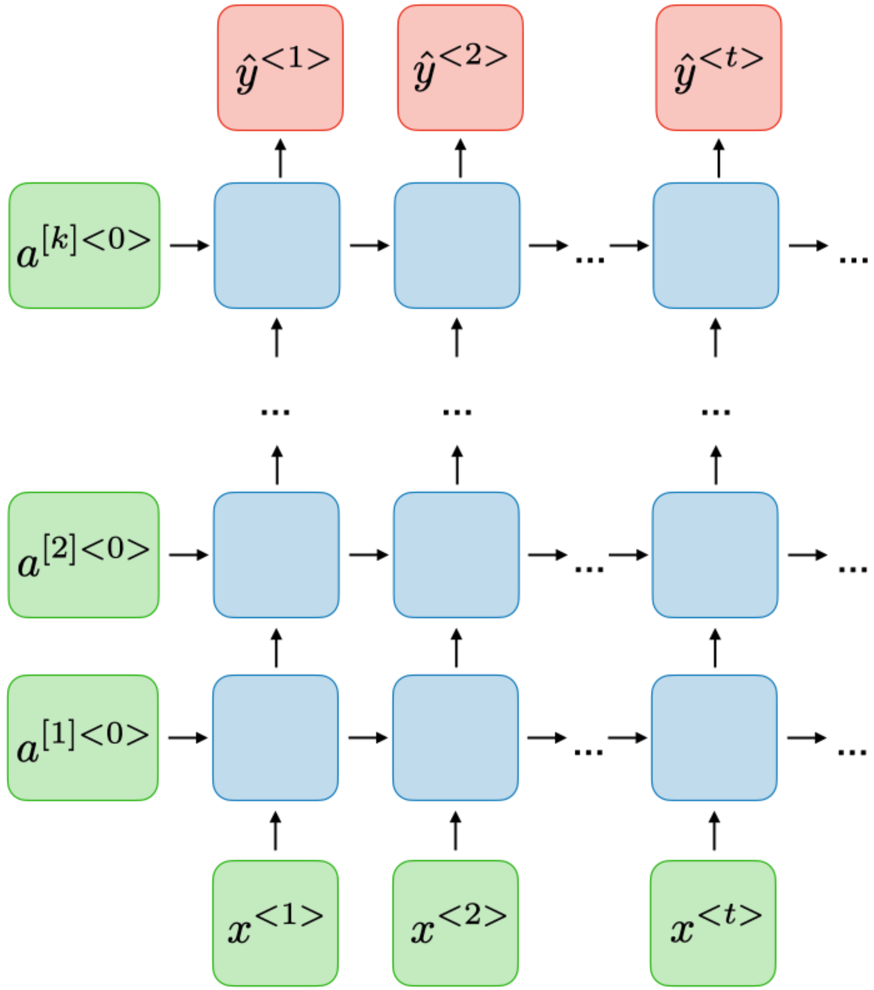
Epoch 4: 100% 6911/6911 [00:42<00:00, 161.78it/s, loss=0.19, v_num=7]

```
Validation accuracy: tensor(0.7926, device='cuda:0')
Validation accuracy: tensor(0.8112, device='cuda:0')
Training accuracy: tensor(0.8298, device='cuda:0')
Validation accuracy: tensor(0.8187, device='cuda:0')
Validation accuracy: tensor(0.8158, device='cuda:0')
Training accuracy: tensor(0.8596, device='cuda:0')
Validation accuracy: tensor(0.8183, device='cuda:0')
Validation accuracy: tensor(0.8201, device='cuda:0')
Training accuracy: tensor(0.8795, device='cuda:0')
Validation accuracy: tensor(0.8220, device='cuda:0')
Validation accuracy: tensor(0.8232, device='cuda:0')
Training accuracy: tensor(0.8938, device='cuda:0')
Validation accuracy: tensor(0.8242, device='cuda:0')
Validation accuracy: tensor(0.8244, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=5` reached.
Training accuracy: tensor(0.9048, device='cuda:0')
```

# Deep RNNs

**Basic idea**: Have multiple RNNs in a “stack”, with the bottom one running over the text, but the upper ones running over the output from the lower ones

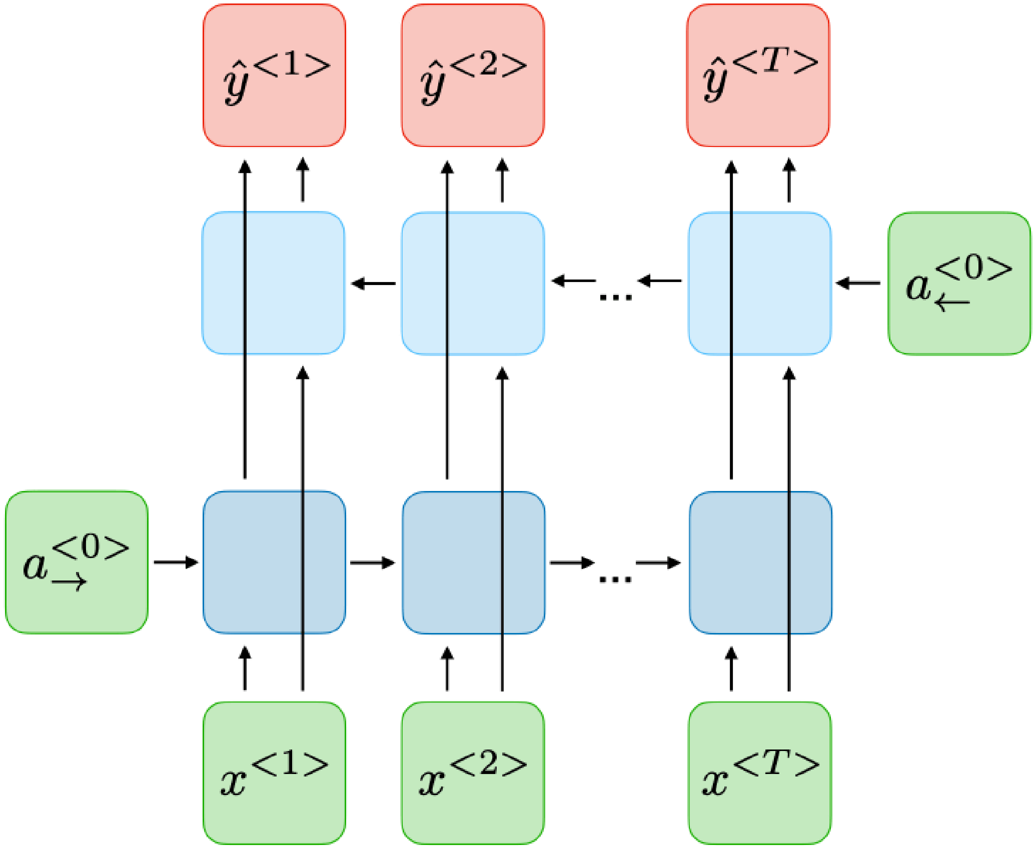
Adds more learning capacity to the model, just like feedforward nets versus logistic regression



# Bidirectional RNNs

**Basic idea**: run the model separately both forward and backward on the text, and then concatenate the final vectors from both passes

Fights catastrophic forgetting by having a gradient that gets applied at both the beginning and end of the text.



# Dropout

**Basic idea**: with some percentage chance, randomly zero intermediate values within the model during training

Another form of regularization, like L1 or L2 regularization

Discourages overfitting by discouraging the model from relying too much on individual parameter values (which may be dropped).

# Multilayer BiLSTM classification model

```python
class BiLSTMClassifier(pl.LightningModule):
  def __init__(self,
               word_vectors:np.ndarray,
               num_classes:int,
               learning_rate:float,
               padding_id:int,
               lstm_hidden_size:int=100, # how big the inner vectors of the LSTM will be,
               lstm_layers:int =2, # how many layers the LSTM will have
               dropout_prob:float=0.1,
               **kwargs):
    super().__init__( **kwargs)

    # We'll use the same PyTorch Embedding layer as before
    self.word_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(word_vectors),
                                                              freeze=True)
    self.lstm = torch.nn.LSTM(input_size = word_vectors.shape[1], # The LSTM will be taking in word vectors
                              hidden_size = lstm_hidden_size,
                              num_layers=lstm_layers,
                              bidirectional=True,
                              dropout=dropout_prob,
                              batch_first=True # This is important. Set to False by default for some reason.
                              )

    # Output layer input size has to be doubled because the LSTM is bidirectional
    self.output_layer  = torch.nn.Linear(2*lstm_hidden_size, num_classes)
    self.lstm_layers = lstm_layers
    self.learning_rate = learning_rate
    self.padding_id = padding_id # we'll need this later
    self.train_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.val_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
```

# Multilayer BiLSTM classification model

```python
def forward(self, y:torch.Tensor, input_ids:torch.Tensor, verbose=False):
    inputs_embeds = self.word_embeddings(input_ids) #(batch size x sequence length x embedding size)

    input_lengths = (input_ids != self.padding_id).sum(dim=1).detach().cpu()

    packed_embeddings = pack_padded_sequence(inputs_embeds, input_lengths, batch_first=True, enforce_sorted=False)
    packed_output, (final_hidden, final_state) = self.lstm.forward(packed_embeddings)

    last_layer_idx = self.lstm_layers-1
    # final_hidden shape is (lstm_layers * 2 x lstm_hidden_size)
    last_layer_final_forward_hiddens = final_hidden[2*last_layer_idx]
    last_layer_final_reverse_hiddens = final_hidden[2*last_layer_idx+1]
    combined_last_layer_hiddens = torch.cat([last_layer_final_forward_hiddens, last_layer_final_reverse_hiddens], dim=1)

    py_logits = self.output_layer(combined_last_layer_hiddens)
    py = torch.argmax(py_logits, dim=1)
    loss = torch.nn.functional.cross_entropy(py_logits, y, reduction='mean')
    return {'py':py,
            'loss':loss}
```

# Multilayer BiLSTM classification model

```python
bilstm_model = BiLSTMClassifier(word_vectors=vector_model.vectors,
                                num_classes = 2,
                                learning_rate = 0.001, #I'll typically start with something like 1e-3 for LSTMs
                                padding_id = vector_model.key_to_index['<pad>'],
                                lstm_hidden_size=100,
                                lstm_layers=2,
                                dropout_prob=0.1)
print('Model:')
print(bilstm_model)
```

```
Model:
BiLSTMClassifier(
  (word_embeddings): Embedding(400002, 100)
  (lstm): LSTM(100, 100, num_layers=2, batch_first=True, dropout=0.1, bidirectional=True)
  (output_layer): Linear(in_features=200, out_features=2, bias=True)
  (train_accuracy): MulticlassAccuracy()
  (val_accuracy): MulticlassAccuracy()
)
```

```python
bilstm_trainer.fit(model=bilstm_model,
                   train_dataloaders=train_dataloader,
                   val_dataloaders=dev_dataloader)
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name            | Type               | Params
-------------------------------------------------------
0 | word_embeddings | Embedding          | 40.0 M
1 | lstm            | LSTM               | 403 K
2 | output_layer    | Linear             | 402
3 | train_accuracy  | MulticlassAccuracy | 0
4 | val_accuracy    | MulticlassAccuracy | 0
-------------------------------------------------------
403 K     Trainable params
40.0 M    Non-trainable params
40.4 M    Total params
161.615   Total estimated model params size (MB)
Validation accuracy: tensor(0.5000, device='cuda:0')
Epoch 4: 100% 6911/6911 [01:12<00:00, 94.68it/s, loss=0.101, v_num=8]
Validation accuracy: tensor(0.8073, device='cuda:0')
Validation accuracy: tensor(0.8131, device='cuda:0')
Training accuracy: tensor(0.8396, device='cuda:0')
Validation accuracy: tensor(0.8177, device='cuda:0')
Validation accuracy: tensor(0.8291, device='cuda:0')
Training accuracy: tensor(0.9023, device='cuda:0')
Validation accuracy: tensor(0.8337, device='cuda:0')
Validation accuracy: tensor(0.8394, device='cuda:0')
Training accuracy: tensor(0.9331, device='cuda:0')
Validation accuracy: tensor(0.8486, device='cuda:0')
Validation accuracy: tensor(0.8417, device='cuda:0')
Training accuracy: tensor(0.9474, device='cuda:0')
Validation accuracy: tensor(0.8440, device='cuda:0')    Yay
Validation accuracy: tensor(0.8612, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=5` reached.
Training accuracy: tensor(0.9569, device='cuda:0')
```

# Concluding thoughts

RNNs

- One-to-one
- **Many-to-one**
- Many-to-many

LSTMS

Increasing RNN capacity

- Depth
- Bidirectionality

Dropout