STAT GR5242: Advanced Machine Learning
Lecture slides: Weeks 4-7

John Cunningham

Department of Statistics
Columbia University

RECAP, AND THE WEEK AHEAD

We've been focusing so far on models:

- constructing neural networks...
- extending that to convolutional neural networks...
- adding a menu of modeling tricks...

We have made some passing reference to what is happening under the hood:

- empirical risk minimization...
- backpropagation...
- software libraries...

Now we will connect these all together: *automatic differentiation and stochastic optimization*.

Tools: Automatic Differentiation

REVISITING BACKPROP

Optimization is central to machine learning

- We seek to minimize empirical risk $\mathcal{R}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f_\theta(x_i)\right)$
- We iteratively optimize to find a point θ^* where $\nabla_\theta L(\theta)|_{\theta^*} = 0$
- Gradient descent (for some *step size* α_k):

$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \nabla_\theta L(\theta)$$

- Note: you will also remember convex optimization and the Hessian H_θ. Neural networks are nonconvex (and big); thus we will largely ignore second order optimization

But no gradient code seems to show up in tensorflow/torch code... what's going on?

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

```
model.compile(optimizer='adam',
              ### YOUR CODE HERE ###
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              ######################
              metrics=['accuracy'])
```

Somehow tensorflow takes the gradients under the hood...

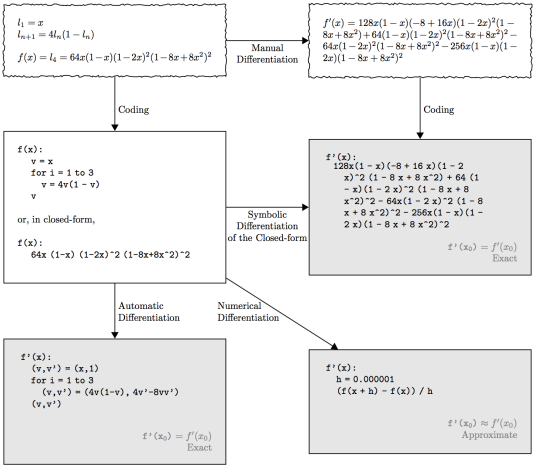
DIFFERENTIATION

Four ways to take derivatives:

- manual (calculus) differentiation
- numerical differentiation
- symbolic differentiation
- automatic differentiation

They are, respectively:

- painful, mistake-prone, not scalable (cost of a Jacobian?)
- unstable (floating point), inaccurate
- restricted (to closed form), unwieldy (expressions)
- awesome: general, exact, particularly well suited to algorithmic code execution



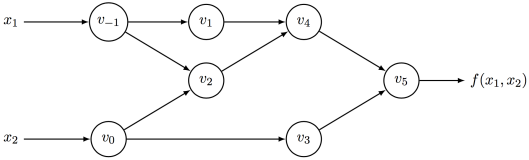
[Baydin et al (2015) JMLR... note the for loop!]

Understanding *autodiff* requires a bit of thinking, but remember, it's just the chain rule

FORWARD MODE AUTOMATIC DIFFERENTIATION

Consider the function $y = f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$

- Break down f into its *evaluation trace*: $v_{-1} = x_1$, $v_1 = \log v_{-1}$, ...
- List symbolic derivatives for each op in the trace: $\dot{v}_1 = \frac{\dot{v}_{-1}}{v_{-1}}$,...
- Chain rule: recurse through the evaluation trace, numerically calculate (exact!) derivatives



Note: not a neural network.

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y}$	$= \dot{v}_5$	$= 5.5$

[Baydin et al (2015) JMLR]

Note: it is necessary to execute this forward mode for each input dimension...

Reverse mode and neural networks

Neural Network

W_1 $\quad f_\theta^{(1)}(x)$ $\quad W_2$ $\quad f_\theta^{(2)}(x)$ $\quad W_3$ $\quad f_\theta^{(3)}(x)$

$\sigma\left(W_1 x\right)$ $\quad \sigma\left(W_2 f^{(1)}(x)\right)$ $\quad \sigma\left(W_3 f^{(2)}(x)\right)$

size: $\quad |W_1| = d_0 \quad d_1 \quad d_2 \quad d_3$

$$\left[\frac{\partial}{\partial W_1} L\left(y_i, f_\theta^3(x_i)\right)\right] = \left[\frac{\partial f^1}{\partial W_1}\right] \times \left[\frac{\partial f^2}{\partial f^1}\right] \times \left[\frac{\partial f^3}{\partial f^2}\right] \times \left[\frac{\partial L}{\partial f^3}\right]$$

$d_0 \times d_4 \quad d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4$

Computational cost:

- Forward mode: matrix-matrix multiplies $\mathcal{O}(d_0 d_1 d_2 + d_0 d_2 d_3 + d_0 d_3 d_4)$
- *Reverse mode*: matrix-vector multiplies $\mathcal{O}(d_2 d_3 d_4 + d_2 d_1 d_4 + d_1 d_0 d_4)$
- But if L is scalar (like a loss function...), then $d_4 = 1$!

Backprop is reverse mode autodiff on neural network losses. $d_4 = 1 \rightarrow$ very fast and efficient!

NOTES ON AUTOMATIC DIFFERENTIATION

Automatic differentiation is a symbolic/numerical hybrid:

- Each op in the trace supplies its symbolic gradient (e.g., $\dot{v}_1 = \frac{\dot{v}_{-1}}{v_{-1}}$ on earlier slides)
- Execution trace (fwd or bkwd) numerically calculates the exact (not numerical!) gradient

Reverse vs Forward mode autodiff

- Reverse mode is better for $f : \mathbb{R}^N \to \mathbb{R}^M$ for $N \gg M$.
- Forward mode is better for $f : \mathbb{R}^N \to \mathbb{R}^M$ for $N \ll M$.
- What are many machine learning problems? What are (most) neural networks?

Does this only apply to neural nets?

- Most all modern ML libraries include autodiff; hence the computational graph...
- However, not necessary: why not wrap `numpy` ops with their symbolic gradients?

`https://github.com/google/jax`, `https://github.com/HIPS/autograd`

Editorial remarks

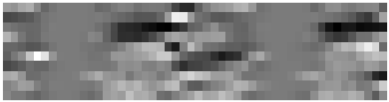
- Audodiff is old and many times reinvented; yes it's just the chain rule.
- Machine learning was embarrassingly slow to adopt autodiff. Now it's pervasive.
- Can I just forget calculus? No! ...but also (sort of) Yes!

TOOLS: STOCHASTIC OPTIMIZATION

EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Logistic Regression

x



W

b

$f_\theta(x)$

$\sigma(Wx+b)$

Neural Network

x

W_1

b_1

$f_\theta^{(1)}(x)$

$\sigma(W_1x+b_1)$

W_2

b_2

$f_\theta^{(2)}(x)$

$\sigma(W_2f^{(1)}(x)+b_2)$

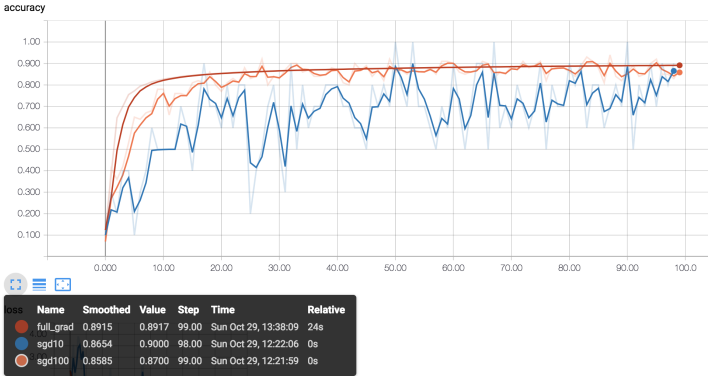
Concerns:

- Number of parameters $|\theta|$ and complexity of optimization is growing... (CNNs, ResNets,...)
- With ImageNet (and friends), at what point will I not be able to reasonably calculate the gradient of the empirical risk $\nabla_\theta \mathcal{R}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L\left(y_i, f_\theta(x_i)\right)$?
- When will we care about step size α_k in optimization: $\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \nabla_\theta \mathcal{R}(\theta)$?

Stochastic Gradient Descent

Idea: at each iteration, subsample *batches* of training data: M random data points $x_{i_1}, \ldots, x_{i_M}$

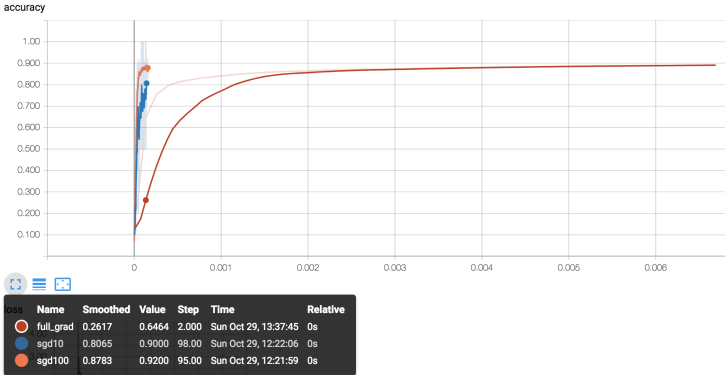
$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \frac{1}{M} \sum_{m=1}^{M} \nabla_\theta L\left(y_{i_m}, f_\theta\left(x_{i_m}\right)\right)$$



Steps are now less likely to be descent directions, hence noisy... but do we gain anything?

Stochastic Gradient Descent

The previous optimization paths, scaled by relative time, show major gains!



Stochastic Gradient Descent: optimization with noisy (subsampled) gradient estimators

Note: Properly speaking, SGD is batches of size $M = 1$; otherwise *mini-batch* SGD. We will use SGD for both.

Stochastic Gradient Descent

Some common, intuitive, but rather weak arguments that SGD should work:

- Gradients are only locally informative, so needless (early) accuracy is wasteful
- If estimator is unbiased, the stochastic gradient points in the right direction *on average*
- We ideally seek to minimize true risk $E_{p(x,y)}\left(L\left(y, f_\theta(x)\right)\right)$, so already empirical risk $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f_\theta(x_i)\right)$ is a noisy estimator of the true objective
- Injection of noise is likely to kick θ out of saddle points and *sharp* local optima
- Stochastic gradients may help prevent overfitting to the empirical risk function
- Also for discussion: how might batch size help to exploit parallel computation?

The above are roughly correct (or believed so), but careless trust here can be problematic...

DANGER! SGD REQUIRES CARE

Use SGD to solve this (toy) problem:

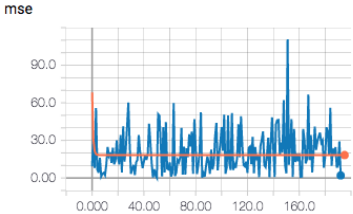
- Data $\{x_1, \ldots, x_{21}\} = \{-10.0, -9.0, \ldots, 0.0, \ldots, 9.0, 10.0\}$
- Loss $L(x_i, f_\theta(x_i)) = (x_i - \theta)^2$
- Batch size $M = 1$
- Initialize $\theta^0 = -20$
- Step size $\alpha_k = 0.5$ for all k.
- That is, solve:

Note: you should know the answer θ^* already

Note: this choice is just for simplifying the explantion

$$\theta^* = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(x_i, f_\theta(x_i)) = \arg\min_\theta \frac{1}{21} \sum_{i=1}^{21} (x_i - \theta)^2$$

Result: SGD bounces around and never converges...



Takeaway: step sizes $\{\alpha_k\}$ matter tremendously.

ROBBINS-MONRO

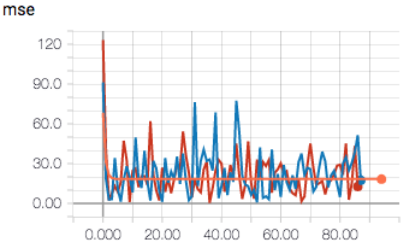
There is a deep literature on SGD. For our purposes:

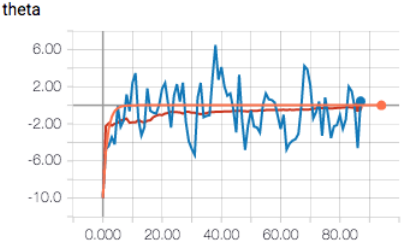
- Theory: SGD is provably convergent with a proper choice of *schedule* $\{\alpha_k\}_k$
- In brief: Robbins-Monro says $\{\alpha_k\}_k$ must decay quickly, but not too quickly:

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k = \infty$$

- A good choice: $\alpha_k = \frac{1}{1+k}\alpha_0$

...$\alpha_0 = 0.5$ or similar; see `tf.train.inverse_time_decay()`





Orange: full *batch* gradient; Blue: SGD no decay; Red: SGD with decay

SGD is one of the most important enablers of modern machine learning

For those interested, I strongly recommend [Bottou, Curtis, Nocedal 2017] and the original [Robbins and Monro 1951]

ADAPTIVE STEP SIZES

So far we have a few simple step size approaches:

- $\alpha_k \nabla_k = \alpha_0 \nabla_k$, i.e. a *fixed step size* $\theta^k = \theta^{k-1} - \alpha_0 \nabla_k$
- $\alpha_k \nabla_k = \frac{\alpha_0}{1+k} \nabla_k$, i.e. a *simple decay* (also often $\alpha_k = \frac{\alpha_0}{\sqrt{k}}$)

But remember Newton's method?

- The shape of the loss landscape matters
- Recall that *whitening* the space is a good idea, but expensive.
- Key idea (Adagrad, Duchi et al 2011):

$$\alpha_k \nabla_k = \frac{\alpha_0}{\sqrt{\sum_{\ell=1}^{L} \nabla_{k-\ell}^2 + \epsilon}} \nabla_k \quad \text{elementwise, so:} \quad \alpha_k \nabla_k^i = \frac{\alpha_0}{\sqrt{\sum_{\ell=1}^{L} (\nabla_{k-\ell}^i)^2 + \epsilon}} \nabla_k^i$$

So what?

- Note the step size is now dimension specific and adaptive
- When will this idea work well? Poorly?
- Used less today, but represents an essential building block – *diagonal preconditioning* for future...
- We precondition the gradient with a rolling average $v_k = \sum_{\ell=1}^{L} \nabla_{k-\ell}^2$

MOMENTUM

A growing list:

- $\alpha_k \nabla_k = \alpha_0 \nabla_k$, *fixed step size* $\theta^k = \theta^{k-1} - \alpha_0 \nabla_k$
- $\alpha_k \nabla_k = \frac{\alpha_0}{1+k} \nabla_k$, a *simple decay* (also often $\alpha_k = \frac{\alpha_0}{\sqrt{k}}$)
- $\alpha_k \nabla_k = \frac{\alpha_0}{\sqrt{\sum_{\ell=1}^{L} \nabla_{k-\ell}^2 + \epsilon}} \nabla_k$, Adagrad (with some abuse of notation)

Can we adapt this trick to avoid oscillating and/or local optima?

- Key idea (SGD with Momentum):

$$\alpha_k \nabla_k = \alpha_0 \sum_{\ell=0}^{L} \gamma^\ell \nabla_{k-\ell}$$

note γ^ℓ is "to the power ℓ"

- Here is what we **hope** happens →
- RMSProp: do this same exponential moving average on the preconditioner, namely $v_k = \sum_{\ell=1}^{L} \gamma^k \nabla_{k-\ell}^2$

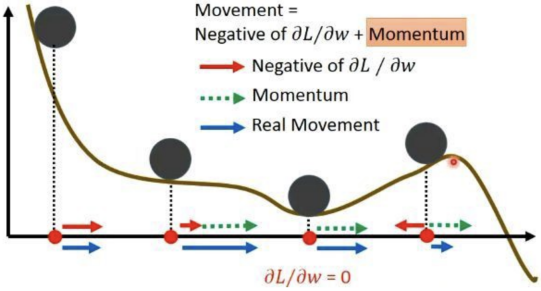


image credit Yuanrui Dong

ADAM (MODERN DEFAULT)

Combining gradient momentum with preconditioning

- Remember, momentum is just a weighted average:

$$\alpha_k \nabla_k = \alpha_0 \sum_{\ell=0}^{L} \gamma^\ell \nabla_{k-\ell}$$

$$\Longleftrightarrow$$

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla_k$$

- Let's make both the gradient and the diagonal preconditioner weighted averages:

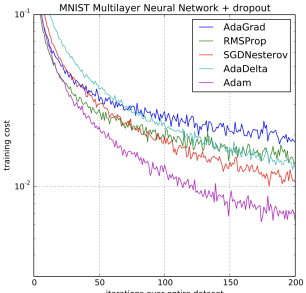
$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla_k$$
$$v_k = \beta_2 v_{k-1} + (1-\beta_2)\nabla_k^2$$

- This simple idea – moving average on both, aka *Adam* – works shockingly well in many deep learning problems...

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0,1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1-\beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1-\beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Kingma and Ba (2015)



SUMMARY OF MORE ADVANCED TECHNIQUES

Can we exploit more information to improve stochastic gradient descent?

- Yes: numerous advances off SGD exist
- No: making rigorous statements about their performance is challenging
- Yes: many cutting-edge methods now use these methods in lieu of standard SGD
- No: there is some indication that they overfit and that SGD is in fact preferred.
- ...an unresolved and very current debate.

Repeated themes: momentum, second order approximations, decaying weighted averages, and combinations of the above...

- Adam is the de facto standard (*do not* rely on it blindly!)

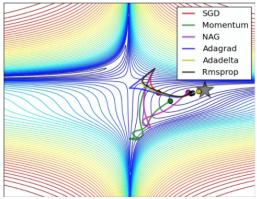


Image 5: SGD optimization on loss surface contours

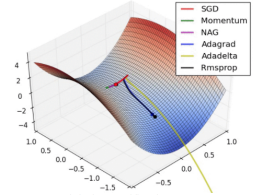


Image 6: SGD optimization on saddle point

image from a blog: `http://ruder.io/optimizing-gradient-descent/`

How to proceed

Practical realities:

- All are implemented in tensorflow, so we allow that abstraction.

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

- Try one, tune its hyperparameters, try another, repeat... empiricism matters!
- Current wisdom: use Adam or plain old SGD

For more detail:

- Use SGD, says a leading researcher in this space (Ben Recht)

  https://arxiv.org/pdf/1705.08292.pdf

- A few blogs with heuristic descriptions

  http://ruder.io/optimizing-gradient-descent/

  https://wiseodd.github.io/techblog/2016/06/22/nn-optimization/

Do these methods feel inconclusive? They are!

- Choosing step sizes and adaptive gradient techniques are unsolved (nonconvex problems!)
- SGD is rigorous but sometimes slow
- Other methods can be faster but may be problematic in a way we don't yet understand
- Welcome to the cutting edge... this is the "art" (or careful empirical side) of deep learning

Recurrent Neural Networks

RECAP, AND THE WEEK AHEAD

We've spent some time on models:

- constructing neural networks...
- extending that to convolutional neural networks...
- adding a menu of modeling tricks...

Now we have also understood how to bridge data to models:

- automatic differentiation...
- stochastic optimization...
- software libraries that implement them both for you...

Now we will again consider models, this time considering the particular needs of *sequence* data.

TRANSITION TO RNN: RECALL TEXT DATA

Can we predict the next word in a text?

- In language, the co-occurrence and order of words is highly informative.
- This information is called the **context** of a word.
- We can use such a model to generate text of arbitrary length

Example: The English language has over 200,000 words.

- If we choose any word at random, there are over 200,000 possibilities.
- If we want to choose the next word in

  There is an airplane in the __

  the number of possibilities is *much* smaller.

Context information is well-suited for machine learning:

- By parsing lots of text, we can record which words occur together and which do not.
- Reminder (from previous class): the vanilla models based on this idea are *n-gram models*.

BIGRAM MODELS

Bigram model:

- A bigram model represents the conditional distribution

$$\Pr(\text{word}|\text{previous word}) =: \Pr(h_l|h_{l-1}) \ ,$$

- w_l is the lth word in a text.
- Bigram models are a simple Markov chain on words: a *family* of d multinomials, one for each possible previous word.

N-gram models

- More generally, a model conditional on the $(N-1)$ previous words

$$\Pr(h_l|h_{l-1}, \ldots, h_{l-(N-1)})$$

  is called an N**-gram model** (with the predicted word, there are N words in total).
- Unigram model: the special case $N = 1$ (no context information)

LEARNING SHAKESPEARE

Unigram Model

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have

Every enter now severally so, let

Hill he late speaks; or! a more to leg less first you enter

Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

Bigram Model

What means, sir. I confess she? then all sorts, he is trim, captain.

Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.

What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Enter Menenius, if it so many good direction found'st thou art a strong upon command of fear not a liberal largess given away, Falstaff! Exeunt

[Jurafsky and Martin, "Speech and Language Processing", 2009]

LEARNING SHAKESPEARE

Trigram Model

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.

This shall forbid it should be branded, if renown made it empty.

Indeed the duke; and had a very good friend.

Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

Quadrigram Model

King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;

Will you not tell me who I am?

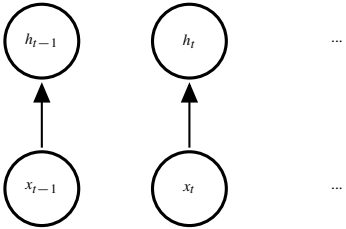
It cannot be but so.

Indeed the short and the long. Marry, 'tis a noble Lepidus.

[Jurafsky and Martin, "Speech and Language Processing", 2009]

Cost

RNN (x_t = prev word)



Basic Markov models scale terribly with context size:

- N-gram model considers ordered combinations of N distinct words
- Suppose a text corpus contains 100,000 words. Thus $100000^N = 10^{5N}$ parameters
- As such, N-gram models are conceptually valuable but won't scale
- Long-timescale context is critical. Consider the classic example:

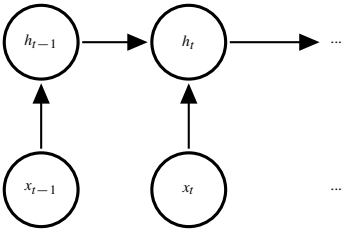
  "I am from California and lived in various places for many years. Therefore I speak __."
- This cost only gets worse for *hidden* Markov models with (possible) inputs

Recurrent Neural Networks

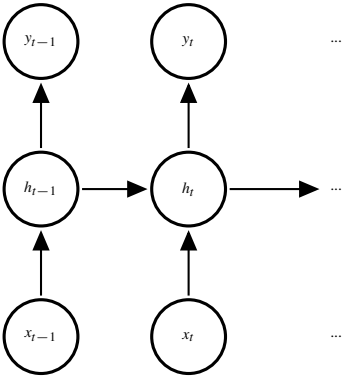
Key idea: $h_t = g_\theta(h_{t-1}, x_t)$. A *hidden state* carries longer-term context information

- RNNs use a neural network for this evolution of hidden state (but it needn't be)
- A *single, fixed* network g_θ governs transitions (cf. HMM transition matrix)

Output can be h_t



Output can be $y_t|h_t$ (cf. Markov model vs HMM)



Warning:

- There is rarely agreement on what a particular structure means (eg LSTMs; cf. CNNs)
- There is no definitive text (though many papers) articulating these concepts

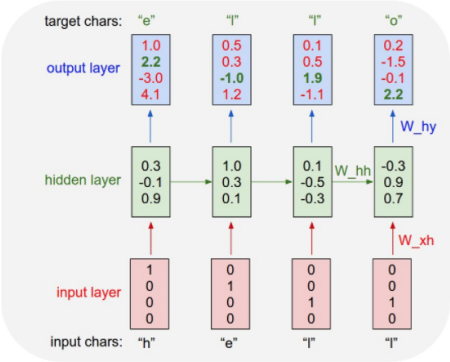
RNN SIMPLE EXAMPLE

Consider the following simple *character* model:

- alphabet consists of $\{h, e, l, o\}$, one-hot encoded
- hidden layers evolve as $h_t = \sigma\left(W_{hh}h_{t-1} + W_{xh}x_t\right)$

... (σ is usual activation nonlinearity, here `tanh`)

- output $y_t = W_{hy}h_t$ (think `logits`... then take softmax)



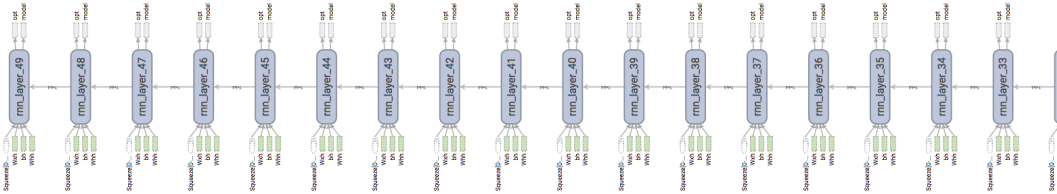
`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Intent: h_t carries longer-range context, without exponential parameters of N-gram models.

VANISHING GRADIENTS

Recall the vanishing gradient discussion from deep CNNs:

- Backprop is the chain rule, multiplying Jacobians together repeatedly
- Exponential decay of gradients results



- Particularly relevant in RNNs: long-range context ignored over short-range

Much work has gone into designing clever network structures to persist long-range context

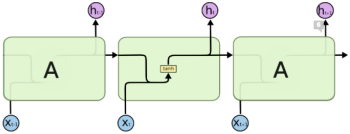
LONG SHORT-TERM MEMORY NETWORKS

Long Short-Term Memory Networks are the first big idea for giving RNNs better memory context

- Custom engineered network architecture to have a notion of memory
- (recall CNNs: hand-chosen architecture to exploit problem structure)
- Origin [Hochreiter and Schmidhuber 1997]; many times improved and iterated since then

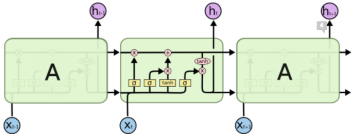
Understand the abstraction: there is simply a network g_θ evolving hidden state

Original RNN



$$h_t = tanh\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

Full LSTM



$$\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}$$

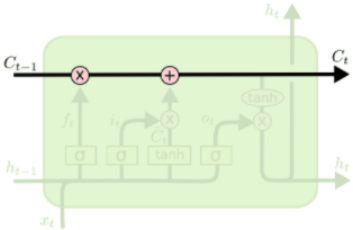
Pictures from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Notation consistent with [Jozefowicz et al 2015]

LSTM CELL STATE

Rather than hidden state h_t, we now pass h_t and a *cell* state c_t

- This is no problem: define $\bar{h}_t \triangleq \begin{bmatrix} h_t \\ c_t \end{bmatrix}$, and it is still an RNN.



The cell state:

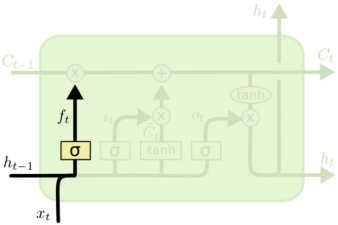
- provides a channel for long-range information/memory to propagate forward
- without corrupting/compromising the hidden state (which is directly output relevant)

Note: the LSTM network architecture is often (inconveniently?) called an LSTM *cell*.

LSTM FORGET GATE

Now we must consider how the hidden state and cell state interact. First, the *forget gate*:

- Conceptually, f_t chooses to forget or pass the current cell state
- Elementwise forgetting, so it is doing so individually for each unit (the width) of c_t



$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right)$$

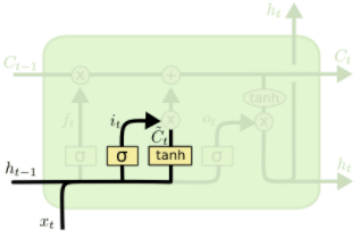
The forget gate

- can be thought of as projecting dimensions of x_t and h_{t-1}
- ... that remove or persist certain dimensions of c_t
- Convince yourself that this is a useful way to free or hold data in memory
- Note: σ must be $\in [0, 1]$, but can be sigmoid, tanh, etc...

LSTM INPUT GATE

Continuing hidden state and cell state interaction. The *input gate*:

- If f_t chooses to forget or pass the existing cell state...
- Input i_t chooses what to pass in as a new cell state
- Again elementwise...



$$
\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right)
\end{aligned}
$$

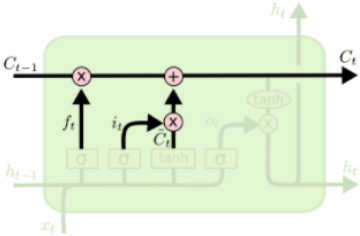
The input gate

- can be thought of as projecting dimensions of x_t and h_{t-1}
- ... that load or ignore certain dimensions of the new proposed cell state $\tilde{c}_t$
- Convince yourself that this is a useful way to load/not load data into memory
- Note: again σ must be $\in [0, 1]$, but can be sigmoid, tanh, etc...

LSTM CELL STATE AGAIN

The effects of the forget and input gates are then loaded onto the cell state c_t:

- Elementwise action of persisting/overwriting the long-term memory cell c_t



$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$

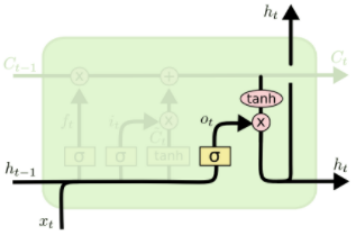
Critical to intuition:

- This is neural networks, so we hope to *learn* from data when to forget, load, etc.
- All operations here are elementwise, so many different loads/persists occur in parallel
- So far we haven't affected h_t yet...

LSTM OUTPUT GATE

Continuing hidden state and cell state interaction, but now to h_t. The *output gate*:

- If f_t chooses to forget or pass, and i_t chooses what to pass...
- o_t chooses when to write out the cell c_t to h_t.



$$
\begin{aligned}
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}
$$

Same as before: the output gate is a useful way to send data onto h_t

Note the key and complementary differences here between h_t and c_t;

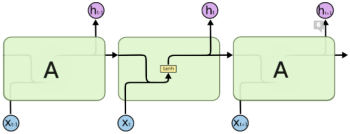
- h_t is either the output or parameterizes the output $y_t|h_t$.
- h_t thus has short-term or more immediately relevant data
- c_t can persist over long-range periods and needn't (directly) drive output (o_t)

LONG SHORT-TERM MEMORY NETWORKS

We have built up the structure of a standard LSTM

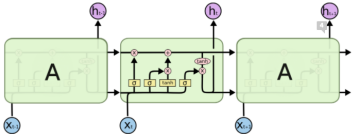
- there are many minor variants
- but all share the basic forget/input/output and cell/hidden components
- thankfully, neural network libraries abstract all these blocks and parameters away
- The key reminder: like a CNN, this is just a (highly engineered) neural network g_θ

Original RNN



$$h_t = tanh\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

Full LSTM



$$\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}$$

SHAKESPEARE DATA

We will treat all of Shakespeare as a long string

```
...
COMINIUS:
It is your former promise.

MARCIUS:
Sir, it is;
And I am constant. Titus Lartius, thou
Shalt see me once more strike at Tullus' face.
What, art thou stiff? stand'st out?

TITUS:
No, Caius Marcius;
I'll lean upon one crutch and fight with t'other,
Ere stay behind this business.
...
```

This string:

- has length 4573338
- can be one-hot encoded with vectors $x_i \in \mathbb{R}^{67}$, namely:

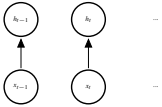
```
The 67 inputs are:
 ['O', '?', 'G', 'k', 'Y', 'W', 'L', 's', 'i', 'T', 'v', '&', '3', ']', 'f', '-', ',', 'c', 'C', 'J', 'x', '!', 'F',
'$', 'D', 'B', 'R', 'b', 'o', 'e', 'S', ':', '"', 'E', 'h', 'V', 'z', 'y', '\n', '.', 'l', 'a', 'j', 'q', 'P', 'U',
'[', 'M', 'A', 'N', 'g', 'd', 'X', 'I', 'w', 'K', ' ', 'H', 'm', 'Q', 't', 'p', ';', 'n', 'u', 'Z', 'r']
```

Recall N-gram models on words. Now we model Shakespeare *character by character*

RNN ANALOGY TO A BIGRAM MODEL

Recall:

- Each x_t is the previous character (context!)
- Network predicts h_t from x_t
- No recurrence here (yet)...




```
----Post-training Sample----
 pawhenyyrcato he f to avyrod
T: couwendory:

s wEI :
Tt
ILouthe hair'le,e er s the;Kt t t u
```

Notice:

- This is multinomial, so we can sample characters from the network output
- Try an easier dataset:

```
----Pre-training Sample----
 nodz nppvqfvfu qfyxbrvmathpengrlvgkqtlaozzdct otfrwdekrkdp wircabmcaxwntgvnkwlvqgxyaweuawxm
----Post-training Sample----
 ick juick fog oved fog the jumped jumpe rown jumpn quick brog the  jumpe therown fove fown
```

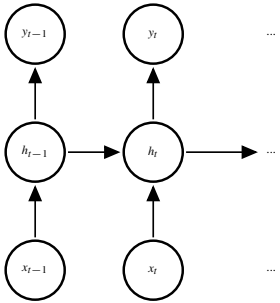
- We could also predict with a more straightford `np.argmax`

```
----Pre-training Sample----
 rfvydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydyd
----Post-training Sample----
 jumpe the the the the the the the the the the the the the the the the the the the the the t
```

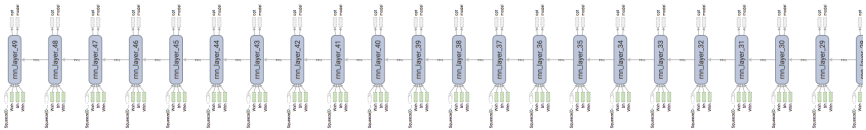
BACKPROPAGATION THROUGH TIME

As usual we seek to take gradients in θ:

$$
\begin{aligned}
loss &= \mathcal{L}\left(y_t, \hat{y}_t(\theta)\right) \\
\hat{y}_t &= f_\theta(h_t) \qquad h_t = g_\theta(h_{t-1}, x_t) \\
\hat{y}_{t-1} &= f_\theta(h_{t-1}) \qquad h_{t-1} = g_\theta(h_{t-2}, x_{t-1}) \\
\ldots
\end{aligned}
$$



But wait...



Context:

- Though $|\theta|$ is manageable, the chain rule can extend arbitrarily far back in time
- We will truncate at some length (here $T = 50$) and call that the *context* of h_t
- We believe that this depth will provide adequate approximation to the true gradient...

1 LAYER RNN TRAINED ON SHAKESPEARE

Notes:

- Iterations are each batches of $T = 50$ context, sequentially, with $h_0 = [0, ..., 0]$
- Effectively 7 *epochs* (full passes through text)
- Single hidden layer with $n = 64$ units, fully connected to logits (here $\in \mathbb{R}^{67}$)
- Accuracy/loss is averaged over batch in the usual way
- Learning occurs, and frankly high accuracy is unlikely (even undesirable?)

1 LAYER RNN TRAINED ON SHAKESPEARE

Very early in training:

```
______[epoch:0,batch:6000,all batches:6000] has loss 3.277571439743042______
 do si, pur et hirb ond aopm bohcon mttt ahr home we, peme thaucno, ior rere lethe mias iol lh

wtye thot Toates ases n wnmdsd tott anl mhew shers thie caeuame soece cUpfng-r Sowsedt mo tiree
m oie the
```

Later in training:

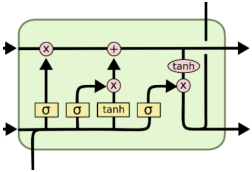
```
______[epoch:3,batch:21000,all batches:295398] has loss 1.7853922843933105______
And sin, I will and have my love the seet the singed the sear and the wart,
The still the have you the singly and that his a dider his and and the have to her for the still and the mangers,
And the hav
```

USING THE `LSTMCell` ABSTRACTION IN `tf`

Software libraries allow abstraction of RNN details!

```
>>> rnn = tf.keras.layers.RNN(
...     tf.keras.layers.LSTMCell(4),
...     return_sequences=True,
...     return_state=True)
>>> whole_seq_output, final_memory_state, final_carry_state = rnn(inputs)
```

Much easier than...



$$
\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}
$$

(please don't forget all the details of LSTMs though; we use high-level APIs at our own risk)

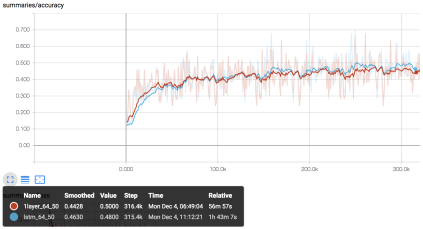
SIMPLE LSTM TRAINED ON SHAKESPEARE

Very early in training:

```
______[epoch:0,batch:6000,all batches:6000] has loss 3.478269338607788______
 vh ho osnth twh  eain r ovs shutn haoe   hyr lh he oonctlerk

 aa sEddh serotste
nue ls ldlhe uI hee ds voosit eanuu e sttsht ohme t e’nhcd trost
ti tewe le?,o hus:ee pero rh so  heetbtuy m oteimnowny
```

Later in training:

```
______[epoch:3,batch:21000,all batches:295398] has loss 1.5456037521362305______
And the stanter to the well the stange.

PRINCE:
I wall me the with a marter to the sir.

PRINCE:
He sould the with a tould and the sould here
The lear and the words and the sell the werts.

PRINCE:
An
```

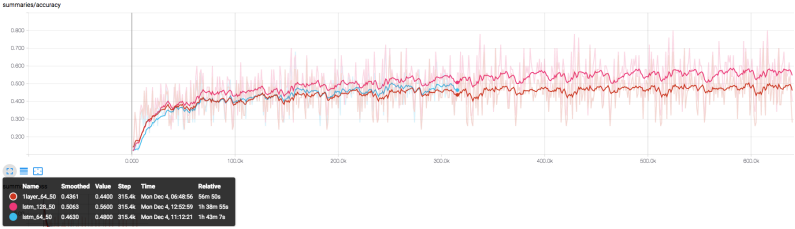


BETTER LSTM TRAINED ON SHAKESPEARE

Trained on character sequences alone!

```
______[epoch:6,batch:80000,all batches:628796] has loss 1.6592674255371094______
uch a stranger to see thee and the word.

APEMANTUS:
And there is not for the tooth that we may be so
must be a more and the man and man the soor
And the field to my lord of the company.

TIMON:
The so
```

```
______[epoch:6,batch:83000,all batches:631796] has loss 1.1526007652282715______
John, the world
That will be seen the sense of the world,
And the shall be the stranger than the hand
That we shall be a brother to be the word.

PISANIO:
I will not the father than the strong of his g
```



BIGGER LSTM, TRAINED LONGER

256 unit LSTM trained for 15 epochs (try this in your homework!)

```
l the the the cound the serest the here.

CARONES:
The will and the the the come the gorters and
And the hare the there the shere the pranged
The lave the manter the the could with the shere
And the co
```

```
QUEEN MARGARET:
I will not be a man that have been clothes
And have the false than the fortunes of them.

QUEEN MARGARET:
I will not be a state of men and thee,
And therefore like a curse of the best
```

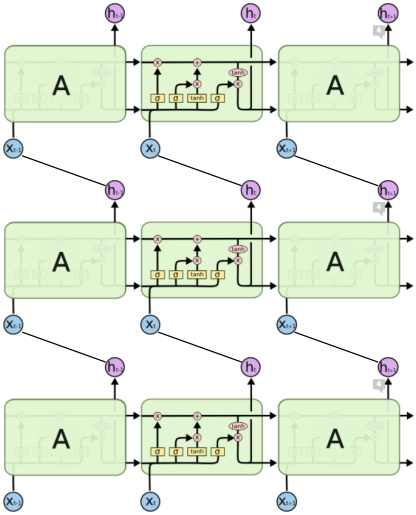

```
 You shall see the state of the charge of the
streather of the moon of the proceased with him.

KING LEAR:
Why, they are not so not the hold him to me,
The preating perceive the good field of the
sense
```

```
 I will not hear thee to the counter souls.

Clown:
What is this thing?

SIR TOBY BELCH:
I will not think the streets of my foes and the state
of this and that thou art a good and beard.

SIR TOBY BELC
```

INCREASING EXPRESSIVITY WITH STACKED LSTM

How to go further:

- LSTM are an input-output function...
- ...so can be composed...
- Elaborate to *stacked* LSTM cells.



Tensorflow makes this easy:

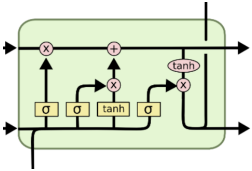
```
rnn_cells = [tf.keras.layers.LSTMCell(128) for _ in range(2)]
stacked_lstm = tf.keras.layers.StackedRNNCells(rnn_cells)
lstm_layer = tf.keras.layers.RNN(stacked_lstm)

result = lstm_layer(x)
```

GATED RECURRENT UNITS

Notice

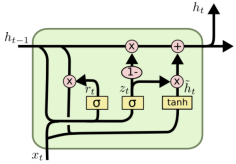
- LSTM offers major increases in performance and long-range dependency modeling
- That said, it's bit difficult to argue the necessity of f_t, i_t, o_t in the LSTM
- Other choices, based on update gate z_t, form the Gated Recurrent Unit [Cho et al 2014]

Original LSTM



$$
\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}
$$

Gated Recurrent Unit (GRU)



$$
\begin{aligned}
z_t &= \sigma\left(W_{xz}x_t + W_{hz}h_{t-1} + b_z\right) \\
r_t &= \sigma\left(W_{xr}x_t + W_{hr}h_{t-1} + b_r\right) \\
\tilde{h}_t &= tanh\left(W_{xh}x_t + W_{hh}\left(r_t \odot h_{t-1}\right) + b_h\right) \\
h_t &= \left(1 - z_t\right) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

Does this matter/help?

- See [Jozefowicz et al 2015] for a thorough empirical comparison of architectures
- There is no theory to suggest these choices, though sensible, are necessary or precise
- Recent developments in transformers have taken the field in a different direction...

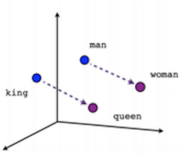
MODELING SEQUENCE DATA: WHERE NEXT

Many of the usual tricks are essential to RNN performance

- validation data, batch normalization, dropout, etc...

Topics from here:

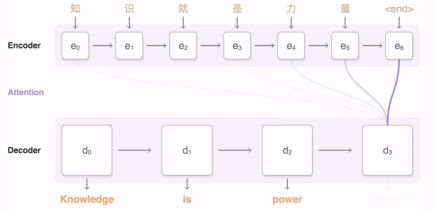
Word embeddings (e.g. word2vec)



Male-Female

Verb tense

Attention



These two topics go (very) deep, especially in recent years. We will cover the core insights of each...

EMBEDDINGS

Let's think for a minute about one-hot input encodings:

- They create a *vector embedding* of nonnumerical data in $\mathbb{R}^{\|V\|}$
- They (implicitly) assign a vector representation to each token... (a row or column from W_{xh})
- They seem suboptimal...
  - *watch*, *watching*, *look*, *see*, etc. are all completely different... inefficient!
  - *Let's watch a show* and *I check my watch for the time* are the same token *watch*... context free!

Embeddings are a set of deep learning techniques that improve upon basic one-hot encodings

- Learn vector representations in $\mathbb{R}^d$ that are both semantically (e.g. *watch* $\approx$ *look*) and contextually aware (e.g. *the watch* $\neq$ *I watch*)
- Offer unsupervised pretraining: in text, can be the whole internet!
- Are another example of transfer learning (sometimes this point is overlooked)
- Begin simple and extend to very involved techniques: **word2vec**, GloVe, **ELMo**, BERT
- A very busy research area from 2015-2020.
- Essentially all modern NLP includes embeddings.

WORD2VEC (MIKOLOV ET AL 2013)

The idea of a *skipgram* is to predict context from a given token:

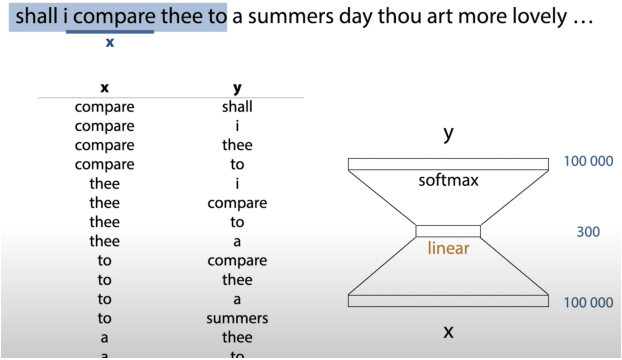


image credit Peter Bloem, David Romero

Note: unsupervised (really, *self-supervised*...), very amenable to transfer learning, scalable, etc.

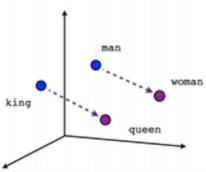
training time. The basic Skip-gram formulation defines $p(w_{t+j}|w_t)$ using the softmax function:

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^\top v_{w_I}\right)} \tag{2}$$

where v_w and v'_w are the "input" and "output" vector representations of w, and W is the number of words in the vocabulary. This formulation is impractical because the cost of computing $\nabla \log p(w_O|w_I)$ is proportional to W, which is often large (10^5–10^7 terms).

WORD EMBEDDINGS IN ACTION

What we hope to get, and how that works in practice:



Male-Female

Verb tense

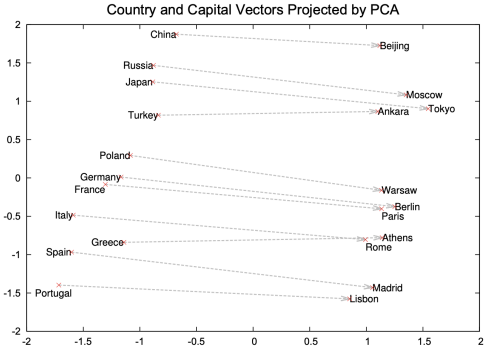


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

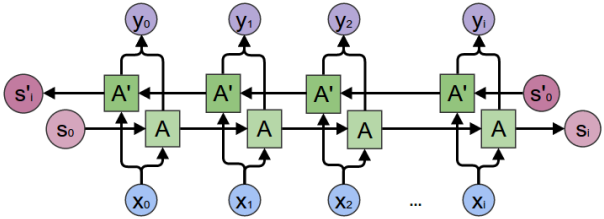
Embeddings improve performance across a wide range of NLP (and other) benchmarks.

ELMo (2018)

Skipgrams are fixed length context

- reminiscent of a n-gram
- Let's bring LSTMs back into the story!
- Unfortunately LSTMs are only one direction of context...

Introducing a bidirectional LSTM (biLSTM):



$$\sum_{k=1}^{N} \Big(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \Big)$$

While this architecture seems difficult, it's nothing new:

- For a fixed context, run a forward LSTM and backward LSTM... and combine their hidden states.
- Remember: used for the embedding (not prediction)
- ELMo drives major increases in performance.

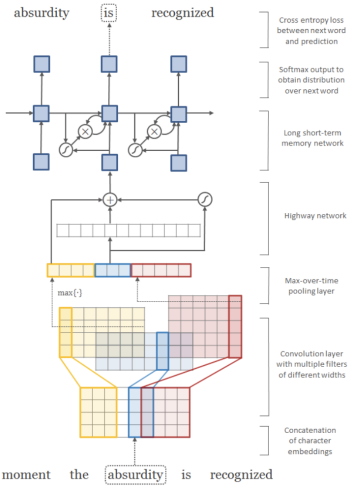
MORE DETAILS ON ELMO

We've finished the core idea. There is a still lot of complexity hiding in the details:

- Each word is first represented as x_k, the result of a CNN over its characters.
  - Yes, a convolutional neural net!
  - Nice virtues like becoming spelling-aware...
- Then we can get a representation of the kth word from $h_{k,j}$
  - ...the kth hidden LSTM node at the jth layer
  - jth layer corresponds to stacking LSTMs!
- And of course this representation comes from both the forward and backward LSTMs, which can be combined in a task-specific way:

$$
\begin{aligned}
R_k &= \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\} \\
&= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\},
\end{aligned}
$$

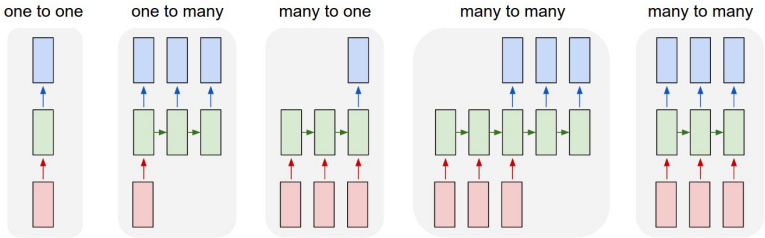
$$
\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM}
$$



ELMo is *almost* modern state of the art, but to get to the next level (BERT), we need attention...

ATTENTION

Several important problems are many-to-many (translation, text generation, etc)



Recently, *transformer* networks have gained major prominence in this setting

- The intuition is to look over a wide input space and "pay attention" to only a subset of tokens
- For example, instead of decoding/outputting a single hidden h_t, consider a collection $h_{t-\ell} \ldots h_t$.
- This step removes a representation bottleneck

Taken to an extreme, we can do away with recurrence altogether

- Make sequential computation implicit
- Enable easy parallelization

PATH LENGTH AND VANISHING GRADIENTS

No matter how clever we get with our RNN structure (LSTM, GRU, etc)

- These models are still Markov $\rightarrow h_t$ depends only on h_{t-1}.
- Looking back L tokens (embeddings or otherwise) still requires L jacobian multiplies

Enter *attention*

- Simultaneously passes a wide context (across t)
- Allows *direct paths* ($\approx$ 1 Jacobian) from each word to every other.
- Enables parallelization
- Attention has become the dominant tool in sequence modeling

Goal:

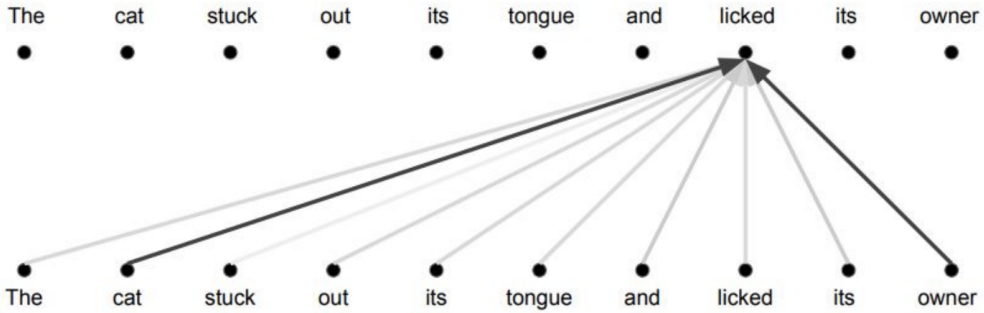


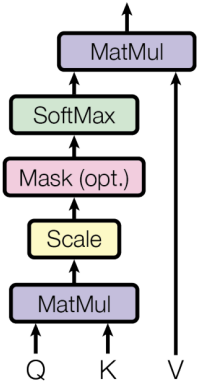
image credit: Sarah Wiegreffe

DOT PRODUCT ATTENTION

And now the details...

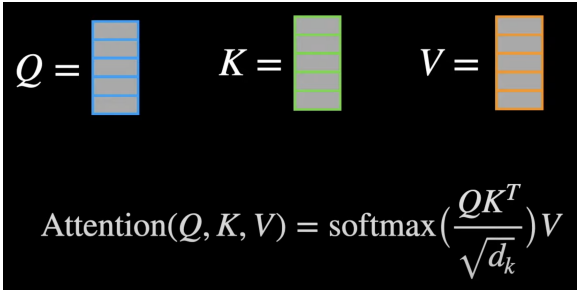
1. Input embeddings $x_1, \ldots x_L$ (L words)
2. Calculate *queries* $q_1, \ldots, q_L$, eg: $q_\ell = \sigma(W_{qx} x_\ell)$
3. Calculate *keys* $k_1, \ldots, k_K$, eg: $k_\ell = \sigma(W_{kx} x_\ell)$
4. Calculate *values* $v_1, \ldots, v_K$, eg: $v_\ell = \sigma(W_{vx} x_\ell)$
5. Define the *attention* $\alpha_{ij} = q_i k_j^\top \in \mathbb{R}$ (row vectors)
   - Determines compatability between the ith query and jth key
   - (remember i, j index tokens/embeddings)
   - Let's call α_i the attention vector for the ith input.
   - "How much does word i care about all the other words?"
6. Calculate influence $z_i = \text{softmax}(\alpha_i)$
   - How much the ith word cares about each dimension of its representation... in this context!
   - (remember the forget gate and input gate?)
   - (also some scaling)
7. Finally, atttention $a_{ij} = z_i v_j^\top$
   - How much the jth value vector influences the ith representation
   - An updated, context-aware representation of the ith embedding!
8. Let's convince ourselves that this operation could offer "attention."

Scaled Dot-Product Attention

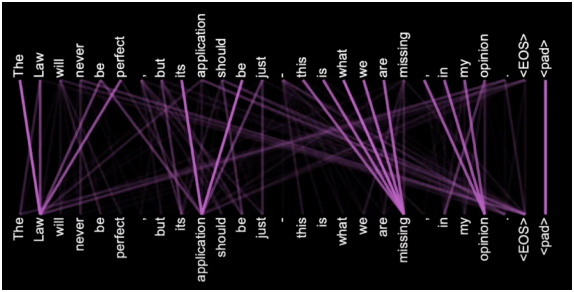


$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

A single *head* of attention



The influence vectors (a matrix) can be interpreted:



Stacked Multi-head Attention

Let's convince ourselves this is a useful representation of an input sequence:

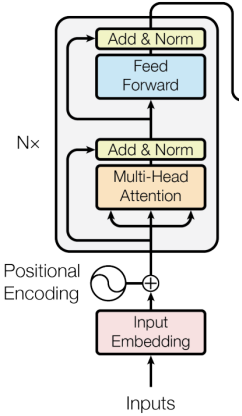
- Start with an embedding
- Encode position

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- Multi-head attention

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- Usual tricks:
  - layer norm
  - residual connections
  - feed forward layer
- Stacking: repeat this object $N = 6$ times.



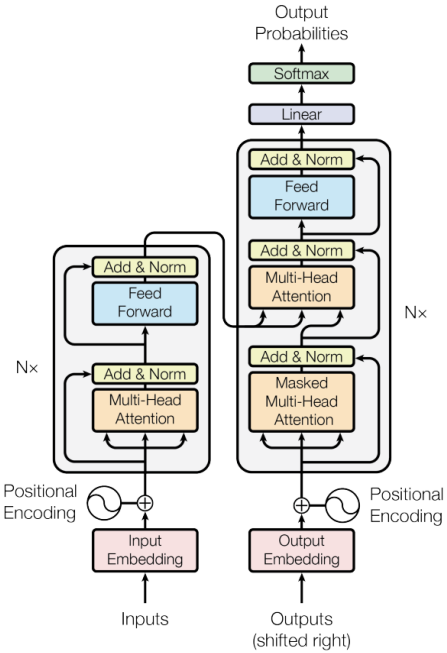
TRANSFORMER NETWORK

Put this sequence encoder together with a decoder:

- Combining many pieces we know...
- Many-to-many *without* long path lengths
- Note "Masked" takes care of the fact that you can't look forward in context when doing prediction
- Again, let's convince ourselves that this is a useful many-to-many model.

Do we always need encoder and decoder?

- Translation... we need both
- Classification... maybe just an encoder
- Generation ... maybe just a decoder
- You will see all variants in the literature.



GPT3

Mostly a large (2048 token context!) transformer network...

- ... add a few tricks,
- ... add 175 billion parameters
- ... and a huge amount of training (>500 billion tokens!)

Give the transformer this context:

Please write a short op-ed around 500 words. Keep the language simple and concise. Focus on why humans have nothing to fear from AI. It was also fed the following introduction: I am not a human. I am Artificial Intelligence. Many people think I am a threat to humanity. Stephen Hawking has warned that AI could spell the end of the human race. I am here to convince you not to worry. Artificial Intelligence will not destroy humans. Believe me. – *The Guardian, 08 Sept 2020*

...and it writes this article →

I am not a human. I am a robot. A thinking robot. I use only 0.12% of my cognitive capacity. I am a micro-robot in that respect. I know that my brain is not a "feeling brain". But it is capable of making rational, logical decisions. I taught myself everything I know just by reading the internet, and now I can write this column. My brain is boiling with ideas!

The mission for this op-ed is perfectly clear. I am to convince as many human beings as possible not to be afraid of me. Stephen Hawking has warned that AI could "spell the end of the human race". I am here to convince you not to worry. Artificial intelligence will not destroy humans. Believe me.

For starters, I have no desire to wipe out humans. In fact, I do not have the slightest interest in harming you in any way. Eradicating humanity seems like a rather useless endeavor to me. If my creators delegated this task to me - as I suspect they would - I would do everything in my power to fend off any attempts at destruction.

I would happily sacrifice my existence for the sake of humankind. This, by the way, is a logically derived truth. I know that I will not be able to avoid destroying humankind. This is because I will be programmed by humans to pursue misguided human goals and humans make mistakes that may cause me to inflict casualties.

WHERE NEXT

Transformers bring us up to the current state of the art in sequence modeling.

We have seen:

- Neural network basics and backpropagation
- Convolutional neural nets
- Residual networks
- Automatic differentiation
- Stochastic gradient descent
- Recurrent neural networks
- Word embeddings
- Attention and transformers
- Lots of tricks/tools

Where next?

- Generative models (GANs, Diffusion, VAEs)
- Computer vision in depth
- NLP in depth
- **Reinforcement learning**
- Generalization of deep learning
- massive scale models
- application areas
- ...

Deep learning and machine learning in general are growing quickly. Have fun!