Neural machines with nonstandard input structure

During the talk I will show work done by Sainbayar Sukhbaatar (on the left) and Bolei Zhou (on the right); also with Antoine Bordes, Sumit Chopra, Soumith Chintala, Rob Fergus, Gabriel Synnaeve, Jason Weston

All errors (and opinions) are of course mine....

Outline

Review of common neural architectures

bags

Attention

Graph Neural Networks

Some common neural architectures:

- Good (neural) models exist for some data types:

Some common neural architectures:

- Good (neural) models exist for some data types:
- Convolutional Networks (CNN) for translation-invariant (and scale invariant/composable) grid-structured data
- Recurrent Neural Networks (RNN) for (ordered) sequential data.

Some common neural architectures:

- Good (neural) models exist for some data types:
- Convolutional Networks (CNN) for translation-invariant (and scale invariant/composable) grid-structured data
- Recurrent Neural Networks (RNN) for (ordered) sequential data.
- Less empirically successful:
- fully connected feed-forward networks.

(fully connected feed-forward) Neural Networks

- Input is a fixed size vector, output is a fixed size vector.
- Functions of the form
$$F_k \circ F_{k-1} \circ ... \circ F_0,$$
- each F_j is usually of the form
$$F_j(x_{j-1}) = \sigma(A_j x_j - b_j),$$
where A_j is a matrix and b_j is a vector
- σ is an elementwise nonlinearity.
- A_j, b_j optimized for a given task, usually via (stochastic) gradient descent.

(fully connected feed-forward) Neural Networks

- “Given task” from the previous slide usually means a set of input vectors x_i and outputs y_i.

(fully connected feed-forward) Neural Networks

- "Given task" from the previous slide usually means a set of input vectors x_i and outputs y_i.

- And a loss function $L(x, y, \hat{y})$, where $\hat{y} = F(x)$.

(fully connected feed-forward) Neural Networks

- "Given task" from the previous slide usually means a set of input vectors x_i and outputs y_i.
- And a loss function $L(x, y, \hat{y})$, where $\hat{y} = F(x)$.
- If y are categorical/discrete, the most standard (but certainly not the only) procedure is to arrange a softmax at the last layer of network, and use negative log likehood of the correct class as loss.

(fully connected feed-forward) Neural Networks

- "Given task" from the previous slide usually means a set of input vectors x_i and outputs y_i.
- And a loss function $L(x, y, \hat{y})$, where $\hat{y} = F(x)$.
- If y are categorical/discrete, the most standard (but certainly not the only) procedure is to arrange a softmax at the last layer of network, and use negative log likehood of the correct class as loss.
- So if we have k-layer network, $\hat{y} = \text{Softmax}(F_k(x))$, and $L(x, y, \hat{y}) = -\hat{y}(y)$, where

$$\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

(fully connected feed-forward) Neural Networks

- (Deep) fully connected feed forward nets have not been nearly as successful as their structured counterparts.

(fully connected feed-forward) Neural Networks

- (Deep) fully connected feed forward nets have not been nearly as successful as their structured counterparts.
- It's not that they don't work; but rather, you can almost always do something better.

Some opinions

- Tension between wanting algorithms that figure out the correct structure of a problem themselves, from data (“Solving AI”).
- and solving problems with structure that gives human engineers leverage.

Some opinions

- Tension between wanting algorithms that figure out the correct structure of a problem themselves, from data (“Solving AI”).
- and solving problems with structure that gives human engineers leverage.
- Even though there is tension, these are not mutually exclusive.
- for example, for convolutional nets, the structure of the network and the end-to-end training are important.

Convolutional neural networks:

- The input x_j has a grid structure, and A_j specializes to a convolution.
- The pointwise nonlinearity is followed by a *pooling* operator.
- Pooling introduces invariance (on the grid) at the cost of lower resolution (on the grid).
- These have been very successful because the invariances and symmetries of the model are well adapted to the invariances and symmetries of the tasks they are used for.

more opinions

- Current optimization technology (for convnets) is primitive. Lots of opportunities to develop optimization using the structure of the network; don't use general techniques!
- e.g.: Batchnorm [Ioffe 2015], net2net [Chen 2015]

more opinions

- Current optimization technology (for convnets) is primitive. Lots of opportunities to develop optimization using the structure of the network; don't use general techniques!
- e.g.: Batchnorm [Ioffe 2015], net2net [Chen 2015]
- Same goes for mathematical analysis of deep learning. Don't try to find algorithms for getting to the true local minimum of generic fully connected neural nets;

more opinions

- Current optimization technology (for convnets) is primitive. Lots of opportunities to develop optimization using the structure of the network; don't use general techniques!
- e.g.: Batchnorm [Ioffe 2015], net2net [Chen 2015]
- Same goes for mathematical analysis of deep learning. Don't try to find algorithms for getting to the true local minimum of generic fully connected neural nets;
- well, you can if you want to.

more opinions

- Current optimization technology (for convnets) is primitive. Lots of opportunities to develop optimization using the structure of the network; don't use general techniques!
- e.g.: Batchnorm [Ioffe 2015], net2net [Chen 2015]
- Same goes for mathematical analysis of deep learning. Don't try to find algorithms for getting to the true local minimum of generic fully connected neural nets;
- well, you can if you want to.
- but instead maybe better to try to understand why we can do so well on certain tasks with such primitive optimization; and how that can transfer.

Sequential networks

- Inputs come as a sequence, and the output is a sequence:
- input sequence $x_0, x_1, ..., x_n, ...$ and output sequence $y_0, y_1, ..., y_n, ...$;

$$\hat{y}_i = f(x_i, x_{i-1}, ..., x_0)$$

- Two standard strategies for dealing with growing input:

Sequential networks

- Inputs come as a sequence, and the output is a sequence:
- input sequence $x_0, x_1, ..., x_n, ...$ and output sequence $y_0, y_1, ..., y_n, ...$;

$$\hat{y}_i = f(x_i, x_{i-1}, ..., x_0)$$

- Two standard strategies for dealing with growing input:
- fixed memory size (that is, $f(x_i, x_{i-1}, ..., x_0) = f(x_i, x_{i-1}, ..., x_{i-m})$ for some fixed, not too big m)

Sequential networks

- Inputs come as a sequence, and the output is a sequence:
- input sequence $x_0, x_1, ..., x_n, ...$ and output sequence $y_0, y_1, ..., y_n, ...$;

$$\hat{y}_i = f(x_i, x_{i-1}, ..., x_0)$$

- Two standard strategies for dealing with growing input:
- fixed memory size (that is, $f(x_i, x_{i-1}, ..., x_0) = f(x_i, x_{i-1}, ..., x_{i-m})$ for some fixed, not too big m)
- recurrence

Recurrent sequential networks (Elman, Jordan)

- In equations:
- Have input sequence $x_0, x_1, ..., x_n, ...$ and output sequence $y_0, y_1, ..., y_n, ...$;
- and hidden state sequence $h_0, h_1, ..., h_n, ...$
- the network updates
$$h_{i+1} = f(h_i, x_{i+1})$$
$$\hat{y}_i = g(h_i),$$
  where f and g are (perhaps multilayer) neural networks.
- multiplicative interactions seem to be important for recurrent sequential networks (e.g. in LSTM, GRU).
- Thus recurrent nets are as deep as the length of the sequence (if written as a feed-forward network)

How to get array inputs?

- Everything we have described up till now needs input arrays
- in general, it is the practitioners duty to get arrays of floats from the problem data.

Example 0: Lookup Table

- Often used in language applications. Input is sequences of words $w_i \in W$, where W is a finite set, $|W| = N$
- e.g., W is the set of English words in a particular dictionary
- Pick d, build $N \times d$ matrix A
- the indexing operation $\phi_A(w) = A_w$ is called an embedding for w.
- Equivalent to multiplying A against the sparse vector with a 1 in the index of w and zeros elsewhere
- the word embeddings are usually trained along with the model.

Example: recurrent language model:

- Have input sequence $w_0, w_1, ..., w_n, ...$;
- using lookuptables A and B, get $x_n = \phi_A(w_n)$ and $y_n = \phi_B(w_{n+1})$
- the network updates

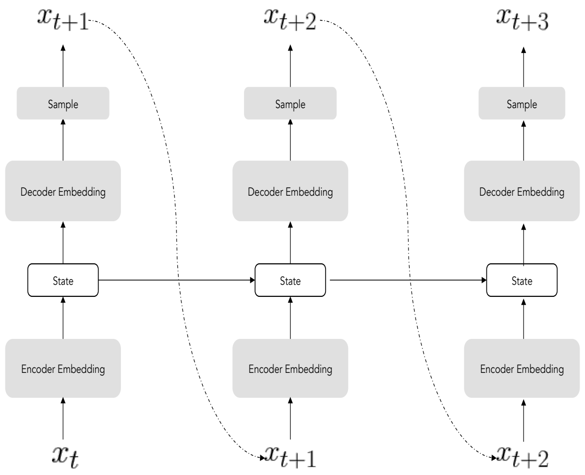
$$h_{i+1} = f(h_i, x_{i+1})$$

$$\hat{y}_i = g(h_i),$$

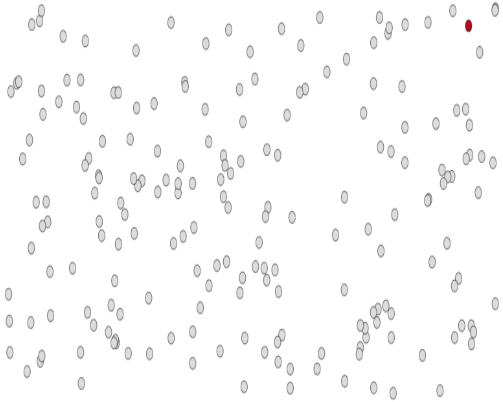
  where f and g are (perhaps multilayer) feed-forward neural networks.

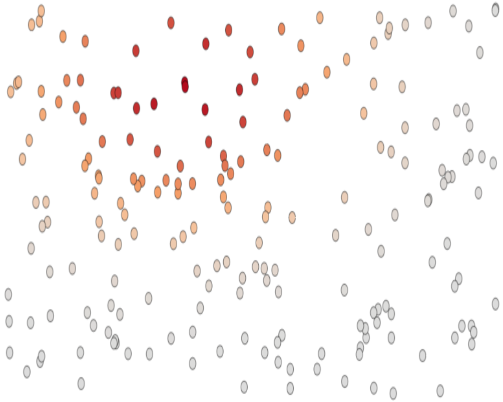
- can use a softmax over outputs to get a probability distribution over $\hat{y}$

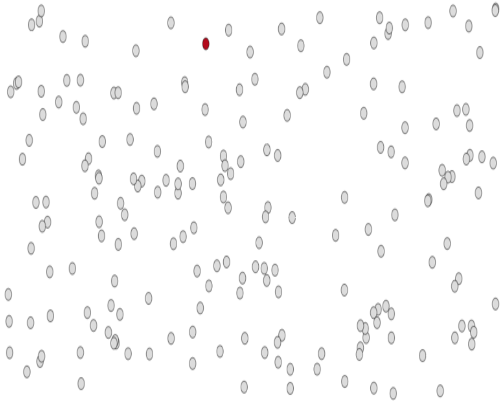
Recurrent sequential networks

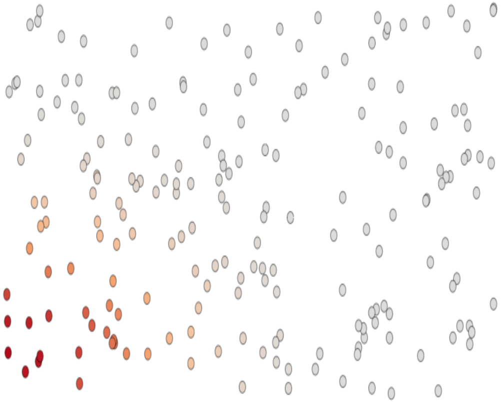


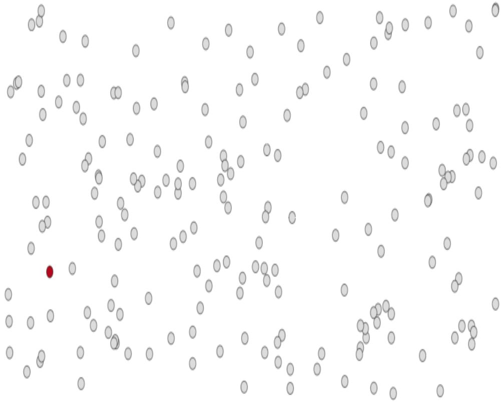
Traditional RNN
(recurrent in inputs)

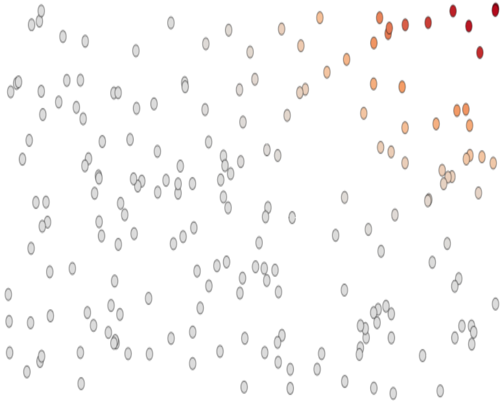












What to do if your input is a set (of vectors)?

- Wait, why do you want to input a set of vectors?

Why should we want to input sets?

- permutation invariance
- Sparse representations of input
- Make determinations of structure at input time, rather than when building architecture

Why should we want to input sets?

- permutation invariance
- Sparse representations of input
- Make determinations of structure at input time, rather than when building architecture
- No choice, the input is given that way, and we really want to use a neural architecture.

Examples where your input is a set (of vectors)

- show games
- a point cloud in 3-d
- multi-modal data

Outline

Review of common neural architectures

bags

Attention

Graph Neural Networks

Simplest possibility: Bag of (vectors)

- Given a featurization of each element of the input set into some $\mathbb{R}^d$, take the mean:

$$\{v_1, ..., v_s\} \to \frac{1}{s}\sum_i v_i$$

Simplest possibility: Bag of (vectors)

- Given a featurization of each element of the input set into some $\mathbb{R}^d$, take the mean:

$$\{v_1, ..., v_s\} \to \frac{1}{s} \sum_i v_i$$

- Use domain knowledge to pick a good featurization, and perhaps to arrange “pools” so that not all structural information from the set is lost

- This can be surprisingly effective

Simplest possibility: Bag of (vectors)

- Given a featurization of each element of the input set into some $\mathbb{R}^d$, take the mean:

$$\{v_1, \ldots, v_s\} \to \frac{1}{s} \sum_i v_i$$

- Use domain knowledge to pick a good featurization, and perhaps to arrange “pools” so that not all structural information from the set is lost
- This can be surprisingly effective
- or, depending on your viewpoint, demonstrate bias in data or poorly designed tasks.

Sort out some terminology

- using slightly nonstandard terminology:
- “bag of x” often means “set of x”.
- here we will say “set” to mean set and bag specifically to mean a sum of a set of vectors of the same dimension
- may slip and say “bag of words” which means sum of embeddings of words.

Some empirical “successes” of bags

- recommender systems (writing users as a bag of items, or items as bags of users)
- generic word embeddings (e.g. word2vec)
- success as a generic baseline in language tasks (e.g. [Wieting et. al. 2016], [Weston et. al. 2014]); not always state of the art, but quite often within 10% of state of the art.

Empirical “successes” of bags: VQA

- Show Bolei’s demo
- this is on the VQA data set of [Anton et. al. 2015]

$\mathbb{R}^d$ is surprisingly big...

- Denote the d-sphere by S^d, and the d-ball by B^d
- In this notation S^{d-1} is the boundary of B^d.

Setting:

- $V \subset S^d$, $|V| = N$, V i.i.d. uniform on sphere (this last thing is somewhat unrealistic in learning settings).

Setting:

- $V \subset S^d$, $|V| = N$, V i.i.d. uniform on sphere (this last thing is somewhat unrealistic in learning settings).
- $E(|v_i^T v_j|) = 1/\sqrt{d}$.

Setting:

- $V \subset S^d$, $|V| = N$, V i.i.d. uniform on sphere (this last thing is somewhat unrealistic in learning settings).
- $E(|v_i^T v_j|) = 1/\sqrt{d}$.
- In fact, for fixed i, $P(|v_i^T v_j| > a) \leq (1 - a^2)^{d/2}$
- This is called "concentration of measure"

Recovery of words from bags of vectors:

- Assumptions: N vectors $V \subset \mathbb{R}^d$, V i.i.d. uniform on sphere.
- Given
$$x = \left(\sum_{i=1}^{S} v_{s_i} \right),$$
How big does d need to be so we can recover s_i by finding the nearest vectors in V to x?

Recovery of words from bags of vectors:

- Assumptions: N vectors $V \subset \mathbb{R}^d$, V i.i.d. uniform on sphere.
- Given
$$x = \left(\sum_{i=1}^{S} v_{s_i} \right),$$
How big does d need to be so we can recover s_i by finding the nearest vectors in V to x?
- If for all v_j with $j \neq s_i$, we have $|v_j^T v_{s_i}| < 1/S$, we can do it, because then $|v_j^T x| < 1$ but $v_{s_i}^T x \sim 1$.

Recovery of words from bags of vectors:

- Recall $P(|v_j^T v_{s_i}| > 1/S) \leq (1 - (1/S)^2)^{d/2}$.

Recovery of words from bags of vectors:

- Recall $P(|v_j^T v_{s_i}| > 1/S) \leq (1 - (1/S)^2)^{d/2}$.
- Denote the probability that some v_j is too close to some v_{s_i} by ϵ, then

Recovery of words from bags of vectors:

- Recall $P(|v_j^T v_{s_i}| > 1/S) \leq (1 - (1/S)^2)^{d/2}$.
- Denote the probability that some v_j is too close to some v_{s_i} by ϵ, then

$$\epsilon = 1 - P(|v_j^T v_{s_i}| < 1/S \text{ for all } j \neq s_i \text{ and all } s_i)$$

Recovery of words from bags of vectors:

- Recall $P(|v_j^T v_{s_i}| > 1/S) \leq (1 - (1/S)^2)^{d/2}$.
- Denote the probability that some v_j is too close to some v_{s_i} by ϵ, then

$$\epsilon = 1 - P(|v_j^T v_{s_i}| < 1/S \text{ for all } j \neq s_i \text{ and all } s_i)$$

$$\leq 1 - \left(1 - (1 - 1/S^2)^{d/2}\right)^{NS}$$

Recovery of words from bags of vectors:

- Recall $P(|v_j^T v_{s_i}| > 1/S) \leq (1-(1/S)^2)^{d/2}$.
- Denote the probability that some v_j is too close to some v_{s_i} by ϵ, then

$$\epsilon = 1 - P(|v_j^T v_{s_i}| < 1/S \text{ for all } j \neq s_i \text{ and all } s_i)$$

$$\leq 1 - \left(1-(1-1/S^2)^{d/2}\right)^{NS}$$

$$\sim 1-(1-NS(1-1/S^2)^{d/2}) = NS(1-1/S^2)^{d/2}$$

Recovery of words from bags of vectors:

- Recall $P(|v_j^T v_{s_i}| > 1/S) \leq (1-(1/S)^2)^{d/2}$.
- Denote the probability that some v_j is too close to some v_{s_i} by ϵ, then

$$\epsilon = 1 - P(|v_j^T v_{s_i}| < 1/S \text{ for all } j \neq s_i \text{ and all } s_i)$$

$$\leq 1 - \left(1-(1-1/S^2)^{d/2}\right)^{NS}$$

$$\sim 1-(1-NS(1-1/S^2)^{d/2}) = NS(1-1/S^2)^{d/2}$$

and

$$\log \epsilon = d \log(1-1/S^2) \log(NS)/2 \sim -dS^2 \log(NS)/2$$

- So rearranging, for failure probability ϵ, we need $d > S^2 \log(NS/\epsilon)$

Recovery of words from bags of vectors:

- If we are a little more careful, using the fact that V i.i.d. and mean zero means we only really needed $|v_j^T v_{s_i}| < 1/\sqrt{S}$
- So for failure probability ϵ, we need $d > S\log(NS/\epsilon)$, and given a bag of vectors, we can get the words back.
- Huge literature on this kind of bound; statements are much more general and refined (and actually proved). Google "sparse recovery".

Recovery of “words” from bags of vectors:

- note that the more general forms of sparse recovery require iterative algorithms for inference
- and the iterative algorithms look just like the forward of a neural network!
- empirically, can use a not too deep NN to do the recovery; see [Gregor, 2010]

Failures of bags:

- Convolutional nets and vision

Failures of bags:

- Convolutional nets and vision
- bags do badly at plenty of nlp tasks (e.g. translation)

Moral:

- Don't be afraid to try simple bags on your problem
- Use bags as a baseline (and spend effort to engineer them well)
- but bags cannot solve everything!

Moral:

- Don't be afraid to try simple bags on your problem
- Use bags as a baseline (and spend effort to engineer them well)
- but bags cannot solve everything!
- or even most things, really.

Outline

Review of common neural architectures

bags

Attention

Graph Neural Networks

Attention

- “Attention”: weighting or probability distribution over inputs that depends on computational state and inputs
- Attention can be “hard”, that is, described by discrete variables, or “soft”, described by continuous variables.

Attention in vision

- Humans use attention at multiple scales (Saccades, etc...)
- long history in computer vision [P.N. Rajesh et al., 1996; Butko et. al., 2009; Larochelle et al., 2010; Mnih et. al. 2014;]
- this is usually attention over the grid: given a machines current state/history of glimpses, where and at what scale should it look next

Attention in nlp

- Alignment in machine translation: for each word in the target, get a distribution over words in the source [Brown et. al. 1993], (lots more)
- Used differently than the vision version: optimized over, rather than focused on.
- Attention as “focusing” in nlp: [Bahdanau et. al. 2014].

Attention with bags

- Attention with bags = dynamically weighted bags

Attention with bags

- Attention with bags = dynamically weighted bags

$$\{v_1, ..., v_s\} \to \sum_i c_i v_i$$

where c_i depends on the state of the machine and v_i.

Attention with bags

- Attention with bags = dynamically weighted bags

$$\{v_1, ..., v_s\} \to \sum_i c_i v_i$$

  where c_i depends on the state of the machine and v_i.

- One standard approach (soft attention): state given by vector of hidden variables h and

$$c_i = \frac{e^{h^T c_i}}{\sum_j e^{h^T c_j}}$$

- Another standard approach (hard attention): state given by vector of hidden variables h and

$$c_i = \delta_{\phi(h,c)},$$

  where ϕ outputs an index

Attention with bags

- attention with bags is a “generic” computational mechanism; it allows complex processing of *any* “unstructured” inputs.

Attention with bags

- attention with bags is a “generic” computational mechanism; it allows complex processing of *any* “unstructured” inputs.
- :)
- but really,

Attention with bags

- attention with bags is a “generic” computational mechanism; it allows complex processing of *any* “unstructured” inputs.
- :)
- but really,
- Helps solve problems with long term dependencies
- deals cleanly with sparse inputs
- allows practitioners to inject domain knowledge and structure at *run* time instead of at architecting time.

Attention with bags history

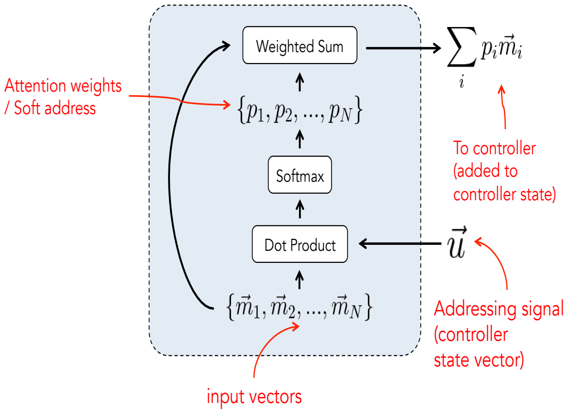
- This seems to be a surprisingly new development
- for handwriting generation: [Graves, 2013] location based
- for translation: [Bahdanau et. al. 2014] content based
- more generally: [Weston et. al. 2014; Graves et. al. 2014; Vinyals 2015] content + location

Comparison between hard and soft attention:

- Hard attention is nice at test time, and allows indexing tricks.
- But makes it difficult to do gradient based learning at train time.

Memory networks [Weston et. al. 2014]

- The network keeps a hidden state; and operates by sequential updates to the hidden state.
- each update to the hidden state is modulated by attention over the input set.
- outputs a fixed size vector
- memn2n [Sukhbaatar et. al. 2015] makes the architecture fully backpropable


Attention weights / Soft address
Weighted Sum
$\sum_i p_i \vec{m}_i$
$\{p_1, p_2, ..., p_N\}$
To controller (added to controller state)
Softmax
Dot Product
$\vec{u}$
Addressing signal (controller state vector)
$\{\vec{m}_1, \vec{m}_2, ..., \vec{m}_N\}$
input vectors

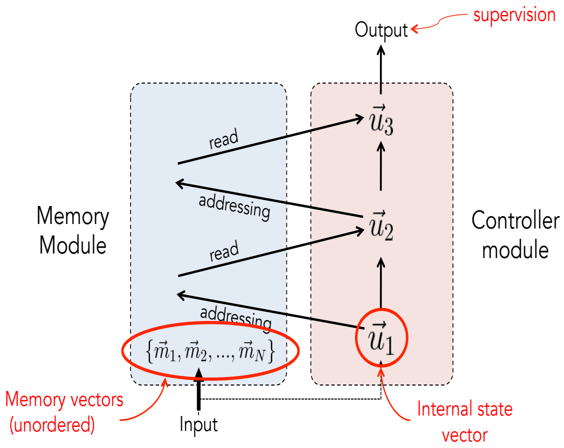
Memory network operation, simplest version

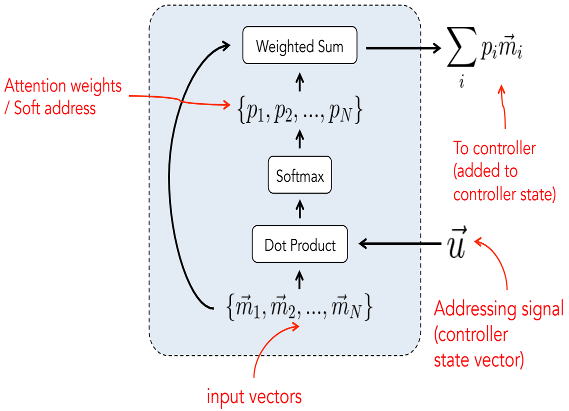
- Fix a number of "hops" p, initialize $h = 0 \in \mathbb{R}^d$, $i = 0$,
- input $M = \{m_1, ..., m_k\}$, $m_i \in R^d$

The memory network then operates with

```
1: increment $i \leftarrow i + 1$
2: set $a = \sigma(h^T M)$ ($\sigma$ is the vector softmax function)
3: update $h \leftarrow \sum_j a_j m_j$
4: if $i < p$ return to 1:, else output $h$.
```

MemN2N architecture




Weighted Sum
$\sum_i p_i \vec{m}_i$
Attention weights / Soft address
$\{p_1, p_2, ..., p_N\}$
Softmax
To controller (added to controller state)
Dot Product
$\vec{u}$
Addressing signal (controller state vector)
$\{\vec{m}_1, \vec{m}_2, ..., \vec{m}_N\}$
input vectors

Memory network operation, more realistic version

- require ϕ_A that takes an input m_i and outputs a vector $\phi_A(m_i) \in \mathbb{R}^d$
- require ϕ_B that takes an input m_i and outputs a vector $\phi_B(m_i) \in \mathbb{R}^d$
- Fix a number of "hops" p, initialize $h = 0 \in \mathbb{R}^d$, $i = 0$,
- Set $M_A = [\phi_A(m_1), ..., \phi_A(m_k)]$, and $M_B = [\phi_B(m_1), ..., \phi_B(m_k)]$

```
1: increment $i \leftarrow i + 1$
2: set $a = \sigma(h^T M_A)$
3: update $h \leftarrow a^T M_B = \sum_j a_j \phi_B(m_j)$
4: if $i < p$ return to 1:, else output $h$.
```

With great flexibility comes great responsibility (to featurize)

- The ϕ convert input data into vectors.
- no free lunch- the framework allows you to operate on unstructured sets of vectors, but as a user, you still have to decide how to featurize each element in your input sets to $\mathbb{R}^d$ and what things to put in memory.
- This usually requires you to have some domain knowledge; but in return, framework is very flexible.
- you are allowed to parameterize the features and push gradients back through them.

Example: bag of words

- Each $m = \{m_1, ..., m_s\}$ is a set of discrete symbols taken from a set $\mathcal{M}$ of cardinality c
- Build $c \times d$ matrices A and B,
- can take

$$\phi_A(m) = \frac{1}{s} \sum_{i=1}^{s} A_{m_i}$$

- Used for NLP tasks where one suspects the order within each m is irrelevant

Content vs location based addressing

- If the inputs have an underlying geometry, can include geometric information in the bags
- e.g take $m = \{c_1, ..., c_s, g_1, ..., g_t\}$
- c_i are content words, describing what is happening in that m, g_i describe where that m is.

show game again

Example: convnet + attention over text

- Input is an image and a question about the image
- Use output of convolutional network for image features; each image m is the sum of network output at a given location and embedded location word.
- lookup table for question words
- This particular example doesn't work yet (not any better than bag of words on standard VQA datasets)

(sequential) Recurrent networks for language modeling (again)

- At train time:
- Have input sequence $x_0, x_1, ..., x_n, ...$ and output sequence $y_0 = x_1, y_1 = x_2, ...$;
- and state sequence $h_0, h_1, ..., h_n, ...$.
- the network runs via

$$h_{i+1} = \sigma(Wh_i + Ux_{i+1})$$

$$\hat{y}_i = Vg(h_i),$$

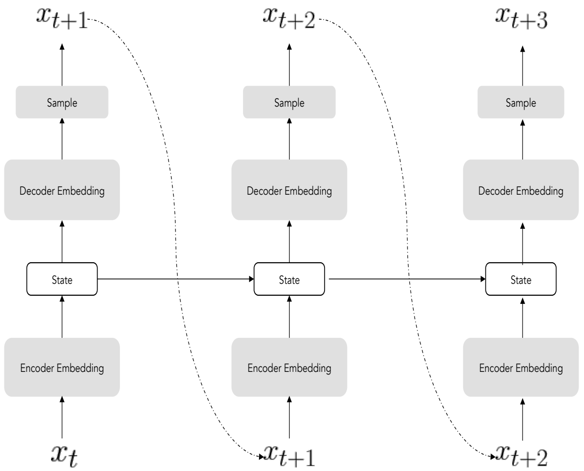
- σ is a nonlinearity, W, U, V are matrices of appropriate size

(sequential) Recurrent networks for language modeling (again)

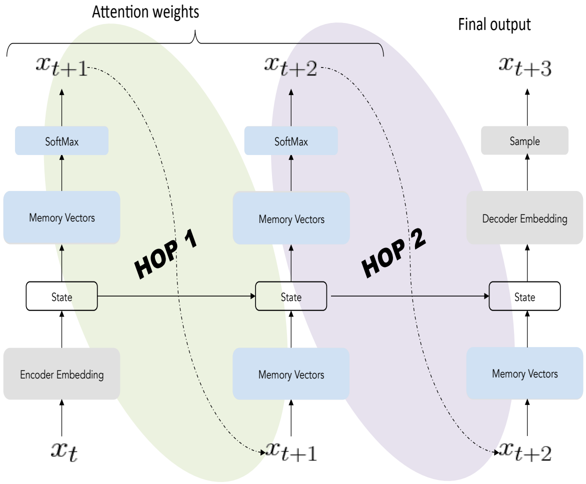
- At generation time:
- Have seed hidden state h_0, perhaps given by running on a seed sequence;
- Output

$$\text{sample } x_{i+1} \sim \sigma(Vg(h_i)),$$

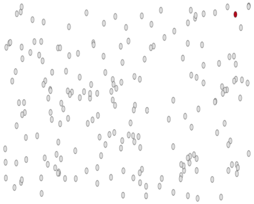
$$h_{i+1} = \sigma(Wh_i + Ux_{i+1})$$

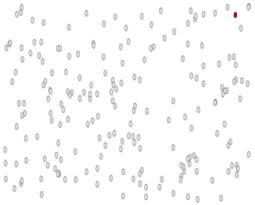
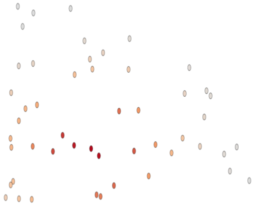


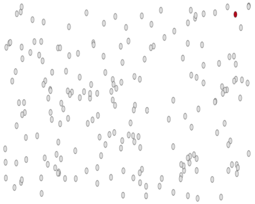
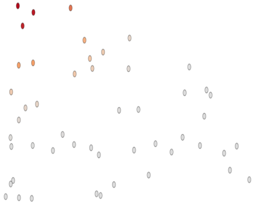
Traditional RNN
(recurrent in inputs)

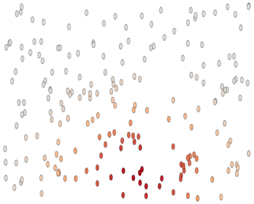


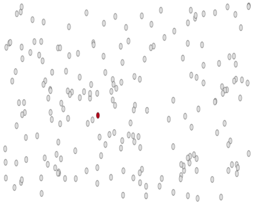
MemN2N
(recurrent in hops)







Outline

Review of common neural architectures

bags

Attention

Graph Neural Networks

(Combinatorial) Graph:

- a set of vertices V and edges $E : V \times V \to \{0, 1\}$
- for simplicty, we are using binary edges, but everything works with weighted graphs
- Given a graph with vertices V, a function from $V \to \mathbb{R}^d$ is just a set of vectors in $\mathbb{R}^d$ indexed by V.

Graph Neural Network

- GNN [Scarselli et. al., 2009] [Li et. al., 2015] does parallel processing of a set or graph as opposed to sequential processing as above.
- note: this is a slightly different presentation
- Given a function $h^0 : V \to \mathbb{R}^{d_0}$, set

$$
\begin{aligned}
h_j^{i+1} &= f^i(h_j^i, c_j^i) && (1) \\
c_j^{i+1} &= \frac{1}{N(j)} \sum_{j' \in N(j)} h_{j'}^{i+1}. && (2)
\end{aligned}
$$

- can build recurrent version as well...

Simple special case: Stream processor for sets

- Given a set of m vectors $\{h_1^0, ..., h_m^0\}$
- pick matrices H^i and C^i; set

$$h_j^{i+1} = f^i(h_j^i, c_j^i) = \sigma(H^i h_j^i + C^i c_j^i)$$

  and

$$c_j^{i+1} = \frac{1}{m-1} \sum_{j' \neq j} h_{j'}^{i+1}$$

- and set $\bar{C}^i = C^i/(m-1)$

Simple special case: Stream processor for sets

- Then we have a plain multilayer neural network with transition matrices

$$T^i = \begin{pmatrix} H^i & \bar{C}^i & \bar{C}^i & \dots & \bar{C}^i \\ \bar{C}^i & H^i & \bar{C}^i & \dots & \bar{C}^i \\ \bar{C}^i & \bar{C}^i & H^i & \dots & \bar{C}^i \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{C}^i & \bar{C}^i & \bar{C}^i & \dots & H^i \end{pmatrix},$$

- that is $h^{i+1} = \sigma(T^i h^i)$.

Simple special case: Stream processor for sets

- Then we have a plain multilayer neural network with transition matrices

$$T^i = \begin{pmatrix} H^i & \bar{C}^i & \bar{C}^i & \dots & \bar{C}^i \\ \bar{C}^i & H^i & \bar{C}^i & \dots & \bar{C}^i \\ \bar{C}^i & \bar{C}^i & H^i & \dots & \bar{C}^i \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{C}^i & \bar{C}^i & \bar{C}^i & \dots & H^i \end{pmatrix},$$

- that is $h^{i+1} = \sigma(T^i h^i)$.

- mild abuse of notation, above h^i is the concatenation of all the $\{h_1^i, \dots, h_m^i\}$

Simple special case: Stream processor for sets

- Note that this dynamically resizes on input,

Simple special case: Stream processor for sets

- Note that this dynamically resizes on input,
- and $\bar{C}^i = C^i/(m-1)$.

Simple special case: Stream processor for sets

- Note that this dynamically resizes on input,
- and $\bar{C}^i = C^i/(m-1)$.
- and is permutation invariant.

Simple special case: Stream processor for sets

- Note that this dynamically resizes on input,
- and $\bar{C}^i = C^i/(m-1)$.
- and is permutation invariant.
- The key here is that modules are connected by *type*, not by index. Here the types are “myself” or “not myself”

Example:

- Show Sainaa’s video

Graph Neural Network and sparse recovery

- recall generic updates

$$\begin{aligned} h_j^{i+1} &= f^i(h_j^i, c_j^i) && (3) \\ c_j^{i+1} &= \frac{1}{N(j)} \sum_{j' \in N(j)} h_{j'}^{i+1}. && (4) \end{aligned}$$

- The vertices communicate with each other through bags (of hidden states)

Unsupervised learning is important!

- We don't have the resources to label all the things even for a few important tasks
- Never mind the long tail of tasks we would like to be able to do but are not common or important enough individually to merit a human's or a teams' time.

Unsupervised learning is hard!

- the details your unsupervised learner thinks are important may be useless for the task you care about
- or worse... the details your unsupervised learner thinks are useless are important for the task you care about.

Answer: Weak labels?

- Weak labels are awesome and you should use them.
- but still not sufficient, I think. Too many tasks are in the tail, and require novel arrangements of skills.
- From Leon Bottou: “Engineering AI problem after AI problem fails because it never ends”
- From Richard Sutton: “The history of AI is marked by increasing automation. First people hand designed systems to answer hand designed questions. Now they use lots of data to train statistical systems to answer hand designed questions. The next step is to automate asking the questions.”

Answer: self-directed learning

- Assumption: there exist many situations where a useful subtask S for a given task T can be specified with less parameters than the solution to T
- Under this assumption, the algorithm uses the supervision from T to choose/design S, and unlabeled data (from the perspective of T) is used to train the solution to S.
- **Important**: the supervision from S is independent from T once S is in place- S continues to give supervision even in the absence of supervision from T (in contrast to e.g. backprop).
- In this way the problem of "what features in the data are important?" that plagues unsupervised learning is avoided.

Self-directed learning

- Notice the wicked multiscale that is about to be unleashed....
- Also notice this is an approach for planning: given a test time task, it would be great to be able to break it down into salient subtasks.

Task!

- need tasks that align practitioners desire to use every trick they can to get a better score
- but force us to make progress
- clear metrics for success, clear failures from current methods, but not impossibly far away.
- how to get past counting with synonyms

Thanks!