Natural Language Processing
Neural Networks

Felipe Bravo-Marquez

April 16, 2024

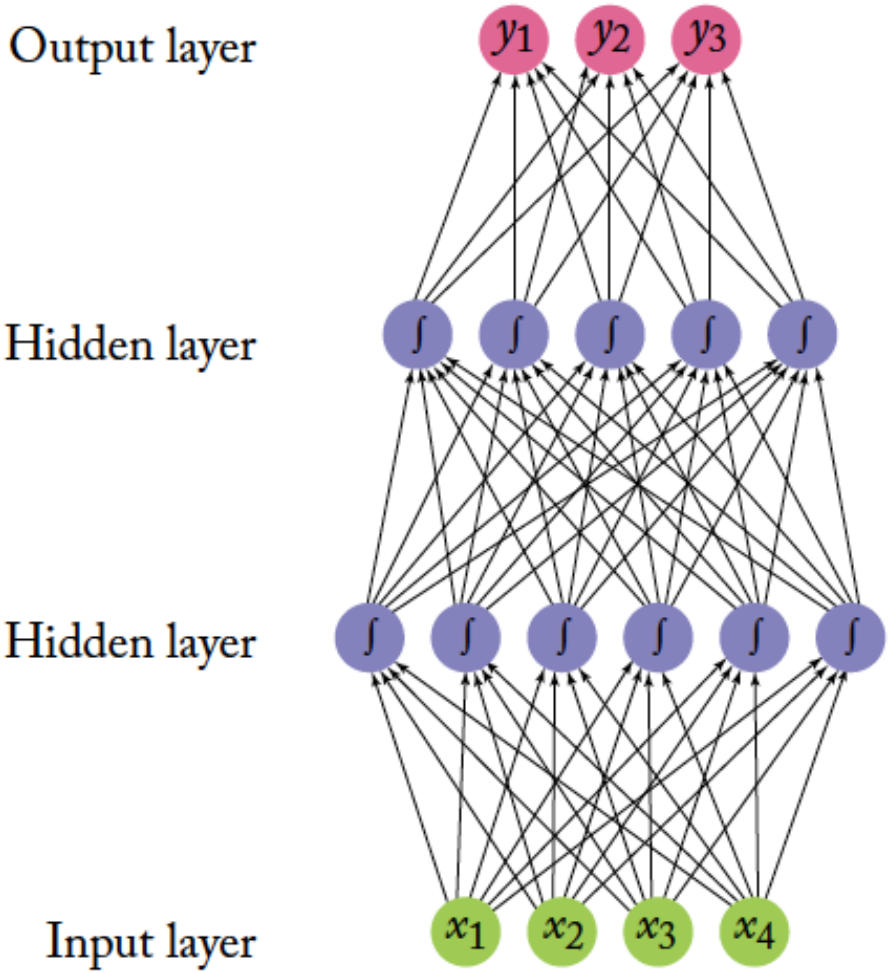
Introduction to Neural Networks

- Very popular machine learning models formed by units called **neurons**.
- A neuron is a computational unit that has scalar inputs and outputs.
- Each input has an associated weight w.
- The neuron multiplies each input by its weight, and then sums them (other functions such as **max** are also possible).
- It applies an activation function g (usually non-linear) to the result, and passes it to its output.
- Multiple layers can be stacked.

Activation Functions

- The nonlinear activation function *g* has a crucial role in the network's ability to represent complex functions.
- Without the nonlinearity in g, the neural network can only represent linear transformations of the input.

Feedforward Network with two Layers



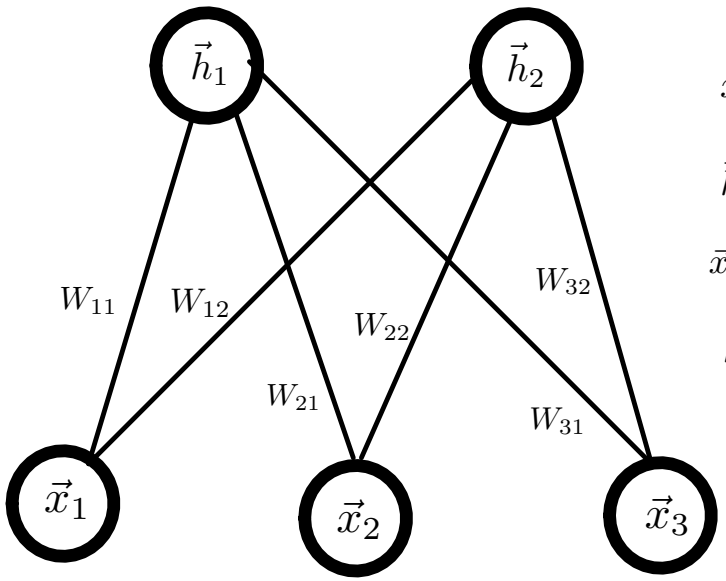
[0]Source:[Goldberg, 2017]

Feedforward Network Neural Networks

- The feedforward network from the picture is a stack of linear models separated by nonlinear functions.
- The values of each row of neurons in the network can be thought of as a vector.
- The input layer is a 4-dimensional vector ($\vec{x}$), and the layer above it is a 6-dimensional vector ($\vec{h}^1$).
- The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions.
- A fully connected layer implements a vector-matrix multiplication, $\vec{h} = \vec{x}W$.
- The weight of the connection from the i-th neuron in the input row to the j-th neuron in the output row is $W_{[i,j]}$.
- The values of $\vec{h}$ are transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer.

[0] Vectors are assumed to be row vectors and superscript indices correspond to network layers.

Fully connected layers as vector-matrix multiplications



$$\vec{x} = [\vec{x}_1, \vec{x}_2, \vec{x}_3] \qquad W = \begin{pmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \\ W_{3,1} & W_{3,2} \end{pmatrix}$$

$$\vec{h} = \vec{x}W$$

$$\vec{x}W = [\vec{x}_1 * W_{11} + \vec{x}_2 * W_{21} + x_3 * W_{31}, \vec{x}_1 * W_{12} + \vec{x}_2 * W_{22} + x_3 * W_{32}]$$

$$\vec{h} = [\vec{h}_1, \vec{h}_2]$$

Neural Networks as Mathematical Functions

- The Multilayer Perceptron (MLP) from the figure is called MLP2 because it has two hidden layers.
- A simpler model would be MLP1, a multilayer perceptron of one hidden layer:

$$\vec{\hat{y}} = NN_{MLP1}(\vec{x}) = g(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2$$
$$\vec{x} \in \mathcal{R}^{in}, W^1 \in \mathcal{R}^{d_{in} \times d_1}, \vec{b}^1 \in \mathcal{R}^{d_1}, W^2 \in \mathcal{R}^{d_1 \times d_{out}}, \vec{b}^2 \in \mathcal{R}^{d_{out}}, \vec{\hat{y}} \in \mathcal{R}^{d_{out}} \quad (1)$$

- Here W^1 and $\vec{b}^1$ are a matrix and a bias term for the first linear transformation of the input.
- The function g is a nonlinear function that is applied element-wise (also called a nonlinearity or an activation function).
- W^2 and $\vec{b}^2$ are the matrix and bias term for a second linear transform.
- When describing a neural network, one should specify the dimensions of the layers (d_1), the input (d_{in}), and the output (d_{out}).

Neural Networks as Mathematical Functions

- MLP2 can be written as the following mathematical function:

$$
\begin{aligned}
NN_{MLP2}(\vec{x}) &= \vec{\hat{y}} \\
\vec{h}^1 &= \vec{x}W^1 + \vec{b}^1 \\
\vec{h}^2 &= g^1(\vec{h}^1)W^2 + \vec{b}^2 \\
\vec{y} &= g^2(\vec{h}^2)W^3 \\
\vec{y} &= (g^2(g^1(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2))W^3.
\end{aligned}
\tag{2}
$$

- The matrices and the bias terms that define the linear transformations are the parameters of the network.
- Like in linear models, it is common to refer to the collection of all parameters as Θ.

Representation Power

- [Hornik et al., 1989] and [Cybenko, 1989] showed that a multilayer perceptron of one hidden later (MLP1) is a universal approximator.
- MLP1 can approximate all continuous functions on a closed and bounded subset of $\mathcal{R}^n$.
- This may suggest there is no reason to go beyond MLP1 to more complex architectures.
- The result does not say how easy or hard it is to set the parameters based on training data and a specific learning algorithm.
- It also does not guarantee that a training algorithm will find the correct function generating our training data.
- Finally, it does not state how large the hidden layer should be.

Representation Power

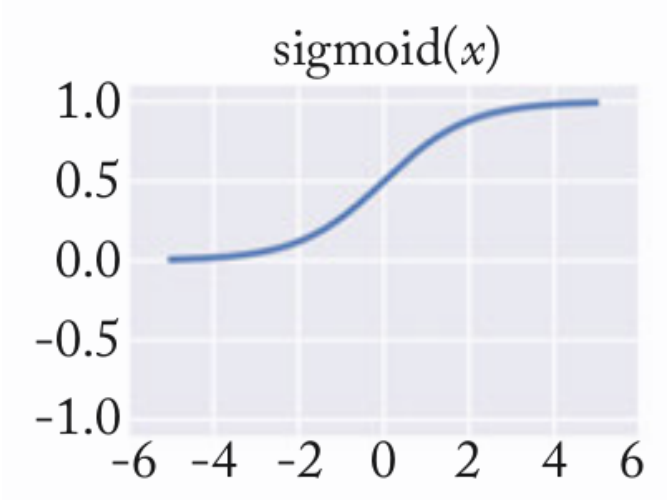
- In practice, we train neural networks on relatively small amounts of data using local search methods.
- We also use hidden layers of relatively modest sizes (up to several thousands).
- The universal approximation theorem does not give any guarantees under these conditions.
- However, there is definitely benefit in trying out more complex architectures than MLP1.
- In many cases, however, MLP1 does indeed provide strong results.

Activation Functions

- The nonlinearity g can take many forms.
- There is currently no good theory as to which nonlinearity to apply in which conditions.
- Choosing the correct nonlinearity for a given task is for the most part an empirical question.

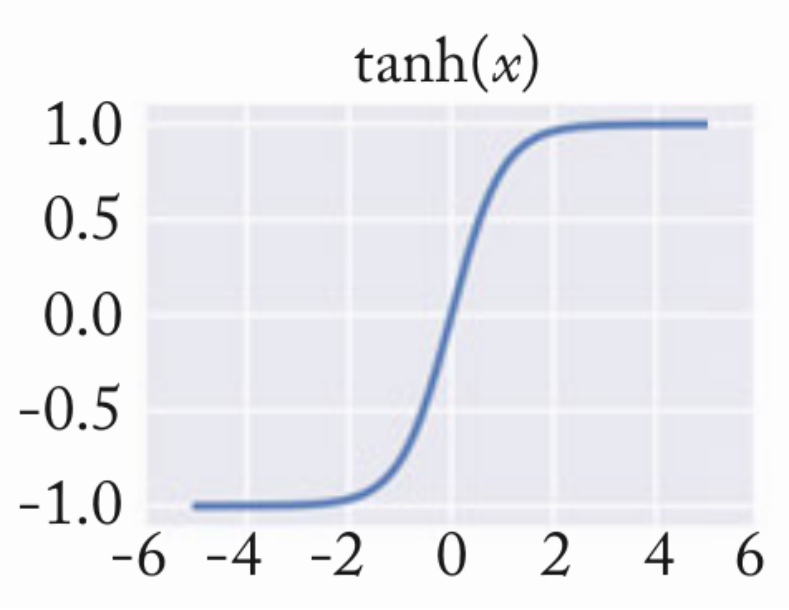
Sigmoid

- The sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ is an S-shaped function, transforming each value x into the range $[0, 1]$.
- The sigmoid was the canonical nonlinearity for neural networks since their inception.
- It is currently considered to be deprecated for use in internal layers of neural networks, as the choices listed next prove to work much better empirically.



Hyperbolic tangent (tanh)

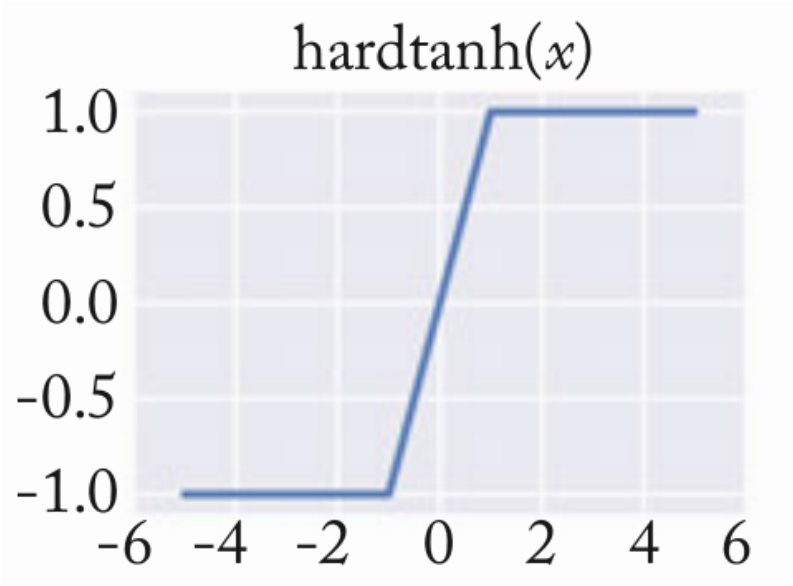
- The hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range$[-1, 1]$.



Hard tanh

- The hard-tanh activation function is an approximation of the tanh function which is faster to compute and to find derivatives thereof:

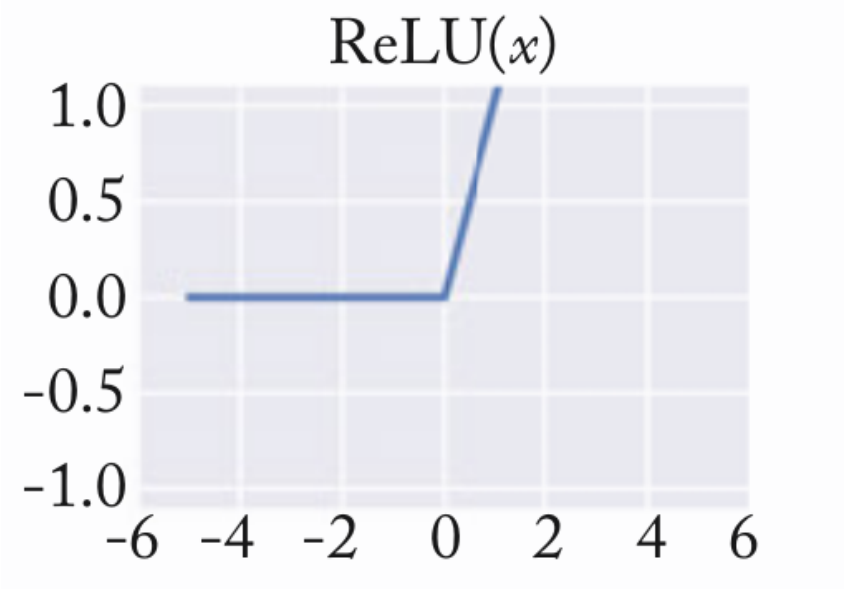
$$\text{hardtanh}(x) = \left\{ \begin{array}{ll} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise.} \end{array} \right\}$$



ReLU

- The rectifier activation function [Glorot et al., 2011], also known as the rectified linear unit is a very simple activation function.
- It is easy to work with and was shown many times to produce excellent results.
- The ReLU unit clips each value $x < 0$ at 0.

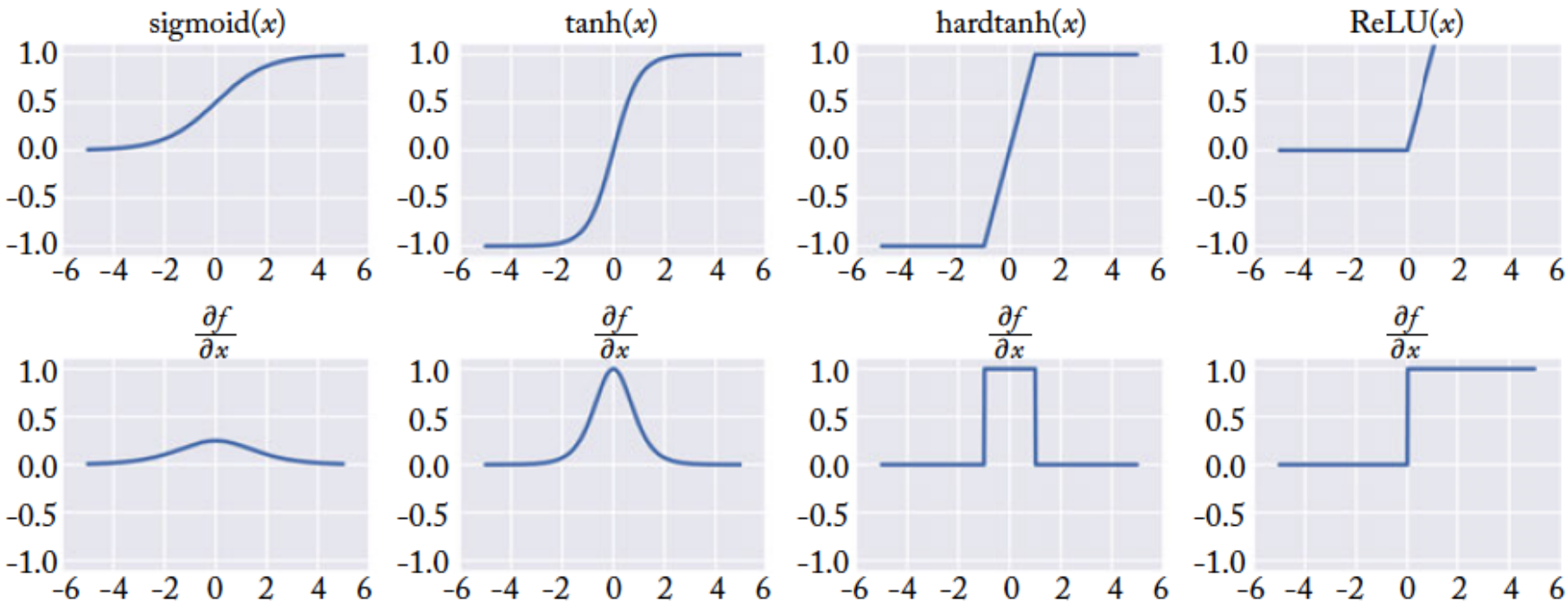
$$\text{ReLU}(x) = \max(0, x)$$



- It performs well for many tasks, especially when combined with the dropout regularization technique (to be explained later).

Activation Functions

- As a rule of thumb, both ReLU and tanh units work well, and significantly outperform the sigmoid.
- You may want to experiment with both tanh and ReLU activations, as each one may perform better in different settings.
- The figure from below shows the shapes of the different activations functions, together with the shapes of their derivatives.



[0]Source:[Goldberg, 2017]

Embedding Layers

- In NLP the input to the neural network contains symbolic categorical features (e.g., words from a closed vocabulary, character n-grams, POS tags).
- In linear models we usually represent the input with sparse vectors e.g., as the sum, average, or the concatenation of one-hot encoded vectors (the sum or the average can produce bag-of words representation).
- In neural networks, it is common to associate each possible feature value (i.e., each word in the vocabulary, each POS tag category) with a $d-$dimensional dense vector for some d.
- These vectors are then considered parameters of the model, and are trained jointly with the other parameters.
- The mapping from a symbolic feature values such as “word number 1249” to $d-$dimensional vectors is performed by an embedding layer (also called a lookup layer).

Embedding Layers

- The parameters in a word embedding layer are simply a matrix $E \in \mathcal{R}^{|vocab| \times d}$ where each row corresponds to a different word in the vocabulary.
- The lookup operation is then simply indexing: $v_{1249} = E_{[1249,:]}$.
- If the symbolic feature is encoded as a one-hot vector $\vec{x}$, the lookup operation can be implemented as a vector-matrix multiplication $\vec{x}E$.
- The embedding vectors are combined before being passed on to the next layer.
- Common combination operations are: concatenation, summation, average.
- A word embeddings matrix E can be initialized with pre-trained word vectors trained from unlabeled documents using specific methods based on the distributional hypothesis such as the ones implemented in Word2Vec (to be discussed later in the course).

The Embedding Matrix

One-hot-encoded word vector

$$\vec{x} = \underbrace{[0, 0, \ldots, 1, \cdots 0]}_{1 \times |V|}$$

abduct dog zumba

Embedding Matrix

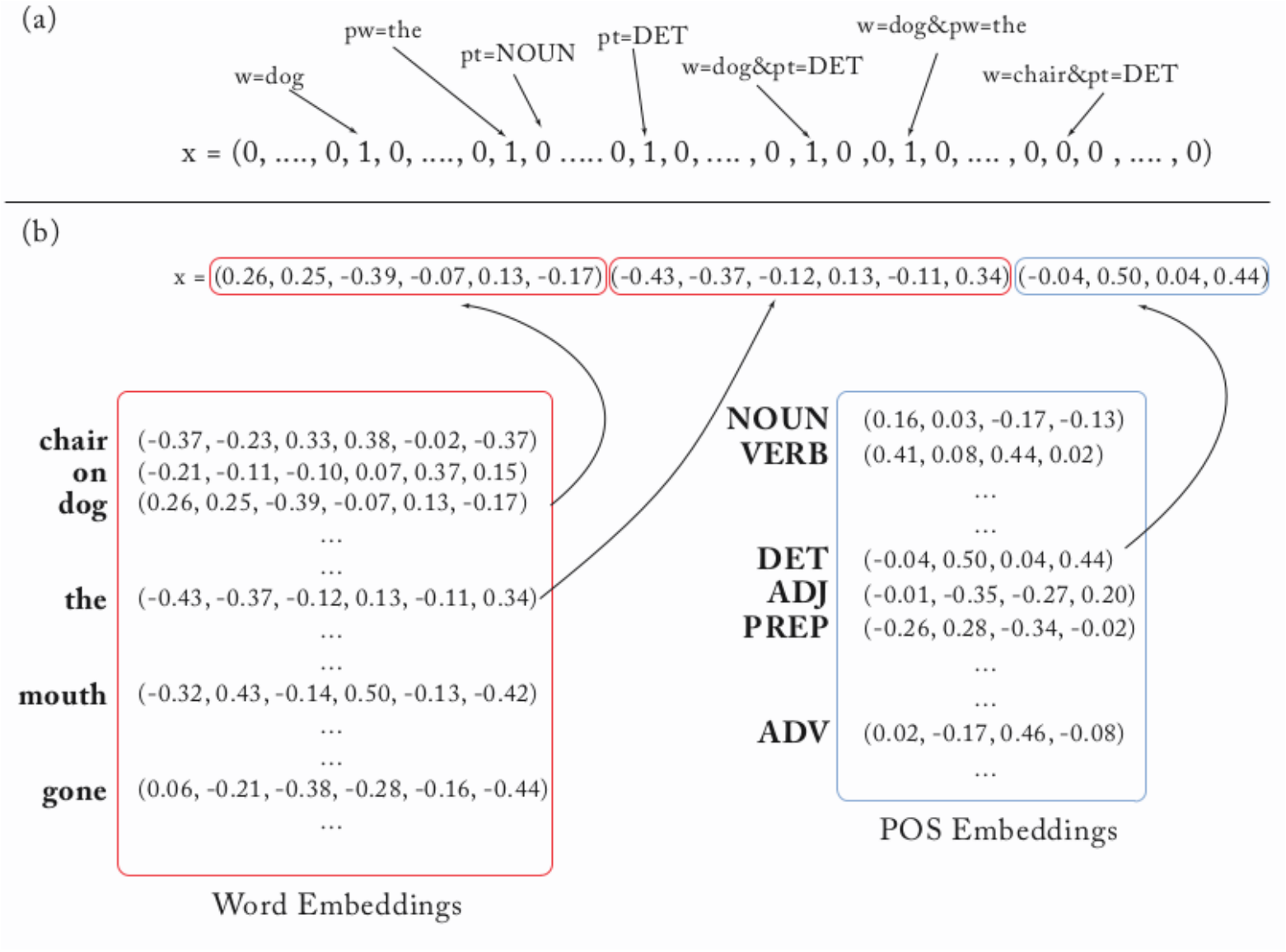
$$E = \underbrace{\begin{bmatrix} -1.8 & 2.3 & \ldots & 3.1 \\ \vdots & \vdots & \ddots & \vdots \\ 3.3 & -2.1 & \cdots & 4.6 \\ \vdots & \vdots & \ddots & \vdots \\ 4.2 & 1.9 & \cdots & -3.3 \end{bmatrix}}_{|V| \times d} \begin{matrix} \\ \\ \longleftarrow \text{abduct} \\ \\ \longleftarrow \text{dog} \\ \\ \longleftarrow \text{zumba} \end{matrix}$$

$$\vec{x}E = [3.3, -2.1, \ldots, 4.6]$$

Dense Vectors vs. One-hot representations

- What are the benefits of representing our features as vectors instead of as unique IDs?
- Should we always represent features as dense vectors?
- Let's consider the two kinds of representations.
- 1) **One Hot**: each feature is its own dimension.
  - Dimensionality of one-hot vector is same as number of distinct features.
  - Features are completely independent from one another. The feature "word is 'dog' " is as dissimilar to "word is 'thinking' " than it is to "word is 'cat' ".
- 2) **Dense**: each feature is a d -dimensional vector.
  - Dimensionality of vector is d.
  - Model training will cause similar features to have similar vectors: information is shared between similar features.

Example: Dense Vectors vs. One-hot representations



Example: Dense Vectors vs. One-hot representations

- Previous figure shows two encodings of the information: current word is “dog;” previous word is “the;” previous pos-tag is “DET.”
- (a) Sparse feature vector.
  - Each dimension represents a feature.
  - Feature combinations receive their own dimensions.
  - Feature values are binary.
  - Dimensionality is very high.
- (b) Dense, embeddings-based feature vector.
  - Each core feature is represented as a vector.
  - Each feature corresponds to several input vector entries.
  - No explicit encoding of feature combinations.
  - Dimensionality is low.
  - The feature-to-vector mappings come from an embedding table.

Dense Vectors vs. One-hot representations

- One benefit of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors.
- However, this is just a technical obstacle, which can be resolved with some engineering effort.
- The main benefit of the dense representations is in generalization power.
- If we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities.

Dense Vectors vs. One-hot representations

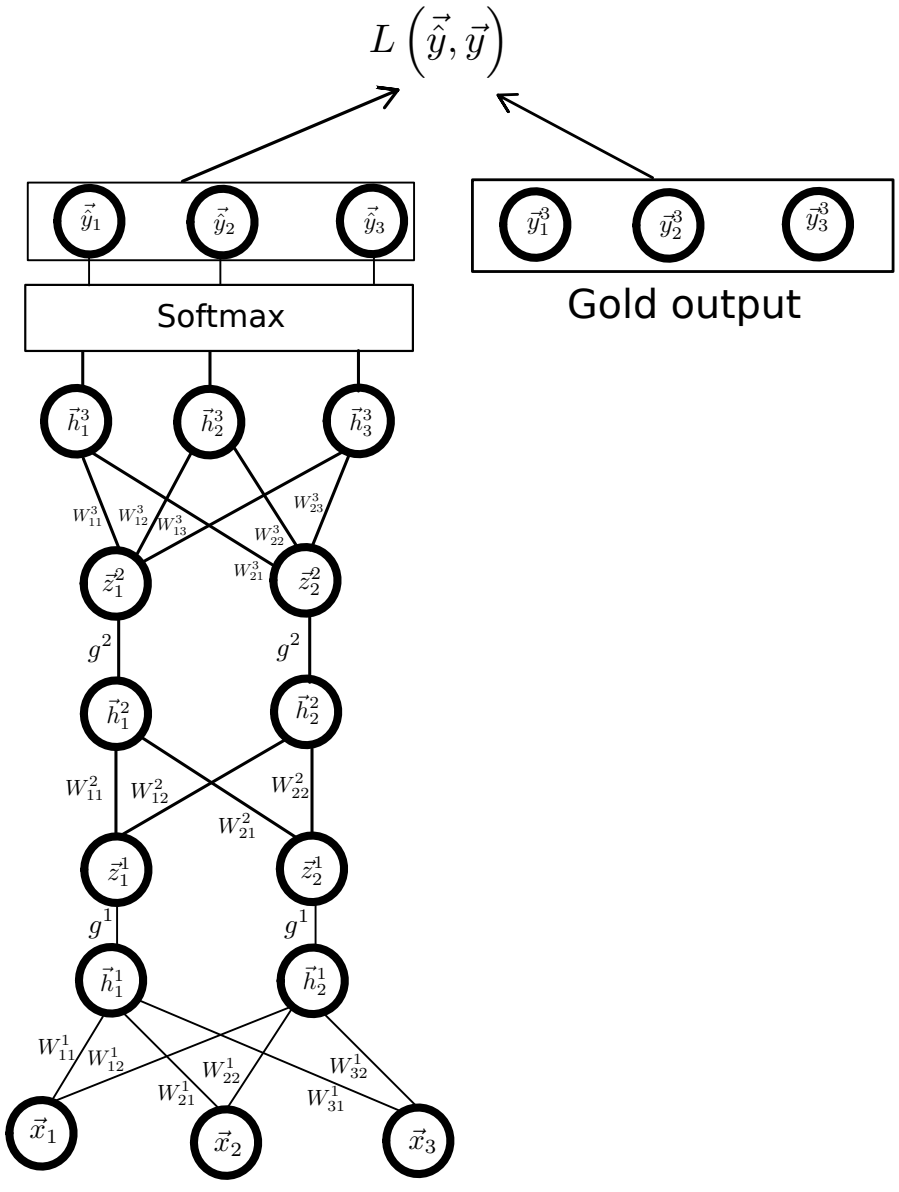
- Let's assume we have observed the word dog many times during training, but only observed the word cat a handful of times.
- If each of the words is associated with its own dimension (one-hot), occurrences of dog will not tell us anything about the occurrences of cat.
- However, in the dense vectors representation the learned vector for dog may be similar to the learned vector for cat.
- This will allow the model to share statistical strength between the two events.
- This argument assumes that we saw enough occurrences of the word cat such that its vector will be similar to that of dog.
- Pre-trained word embeddings (e.g., Word2Vec, Glove) to be discussed later in the course can be used to obtain dense vectors from unannotated text.

Neural Network Training

- Neural networks are trained in the same way as linear models.
- The network's output is used to compute a loss function $L(\hat{y}, y)$ that is minimized across the training examples using gradient descent.
- Backpropagation is an efficient technique for evaluating the gradient of a loss function L for a feed-forward neural network with respect to all its parameters [Bishop, 2006].[1]
- Those parameters are: $W^1, \vec{b}^1, \ldots, W^m, \vec{b}^m$, for a network of m layers.
- Recall that superscripts are used to denote layer indexes (not exponentiations).
- For simplicity, we will assume that L is calculated over a single example.
- Challenge: in neural networks the number of parameters can be huge and we need an efficient way to calculate the gradients.
- Idea: apply the derivative chain rule wisely.

[1] The following slides on backpropagation are based on [Bishop, 2006], we adapted the notation to be consistent with [Goldberg, 2017].

Neural Network Training



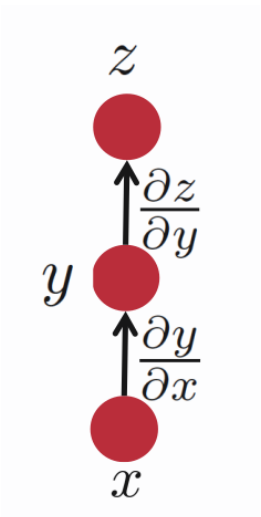
Derivative Chain Rule Recap

- Simple chain rule: let $z = f(y)$, $y = g(x)$,

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$$

- Example: $z = e^y$, $y = 2x$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x} = e^y \times 2 = 2e^{2x}$$



[1] Figure taken from:
http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf

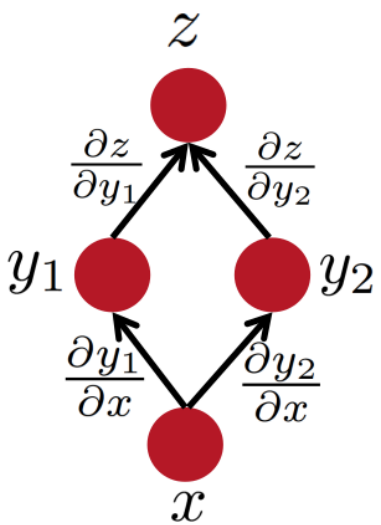
Derivative Chain Rule Recap

- Multiple path chain rule: let $z = f(y_1, y_2)$, $y_1 = g_1(x)$, $y_2 = g_2(x)$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \times \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \times \frac{\partial y_2}{\partial x}$$

- Example: $z = e^{y_1 \times y_2}$, $y_1 = 2x$, $y_2 = x^2$

$$\frac{\partial z}{\partial x} = (e^{y_1 \times y_2} \times y_2) \times 2 + (e^{y_1 \times y_2} \times y_1) \times 2x = e^{2x^3} \times 6x^2$$

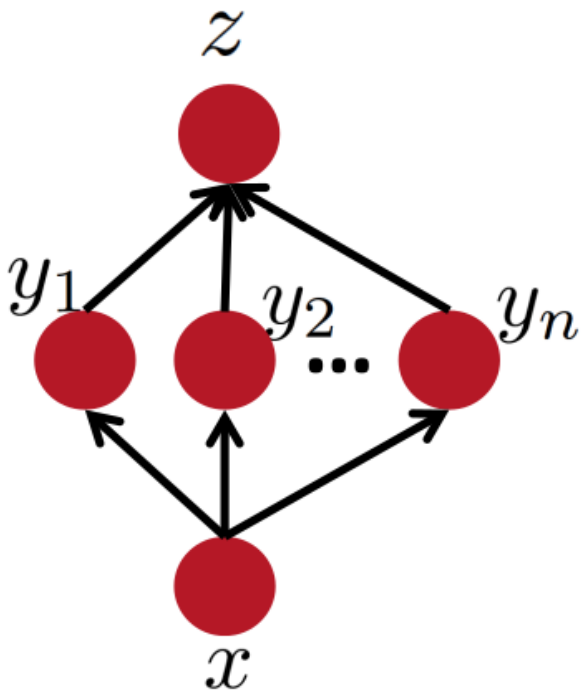


[1] Figure taken from:
http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf

Derivative Chain Rule Recap

The general version of the multiple path chain rule would be:

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \times \frac{\partial y_i}{\partial x}$$



[1] Figure taken from:
http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf

Backpropagation

- In a general feed-forward network, each unit computes a weighted sum of its inputs in the form:

$$\vec{h}^{l}_{[j]} = \left(\sum_{i} W^{l}_{[i,j]} \times \vec{z}^{(l-1)}_{[i]} \right) + \vec{b}^{l}_{[j]} \tag{3}$$

- The variable $\vec{z}^{(l-1)}_{[i]}$ is an input that sends a connection to unit $\vec{h}^{l}_{[j]}$, $W^{l}_{[i,j]}$ is the weight associated with that connection, and l is the layer index.
- The biases vectors $\vec{b}_{[j]}$ can be excluded from (eq.3) and included to the weight matrix $W^{l}_{[i,j]}$ by introducing an extra unit, or input, with activation fixed at +1.

Backpropagation

- The inputs at layer l, $\vec{z}_{[i]}^{(l-1)}$ are the result of applying the activation function g to units from the previous layer:

$$\vec{z}_{[j]}^{l} = g(\vec{h}_{[j]}^{l}) \tag{4}$$

- For the input layer ($l = 0$), $\vec{z}$ corresponds to the input vector $\vec{z} = \vec{x}$

$$\vec{z}_{[j]}^{0} = \vec{x}_{[j]} \tag{5}$$

- For each instance in the training set, we supply the corresponding input vector $\vec{x}$ to the network.
- Next we calculate the activations of all of the hidden and output units in the network by successive application of (eq.3) and (eq.4).
- This process is often called forward propagation because it can be regarded as a forward flow of information through the network.

Backpropagation

- Now consider the evaluation of the derivative of L with respect to a weight $W^l_{[i,j]}$.
- Assuming that the loss L is calculated over a single example, we can note that L depends on the weight $W^l_{[i,j]}$ only via the summed input $\vec{h}^l_{[j]}$.
- We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial L}{\partial W^l_{[i,j]}} = \frac{\partial L}{\partial \vec{h}^l_{[j]}} \times \frac{\partial \vec{h}^l_{[j]}}{\partial W^l_{[i,j]}} \tag{6}$$

Backpropagation

- We now introduce a useful notation:

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} \tag{7}$$

- Using (3), we can write

$$\frac{\partial \vec{h}^{l}_{[j]}}{\partial W^{l}_{[i,j]}} = \vec{z}^{(l-1)}_{[i]} \tag{8}$$

- Substituting (7) and (8) into (6), we then obtain

$$\frac{\partial L}{\partial W^{l}_{[i,j]}} = \vec{\delta}^{l}_{[j]} \times \vec{z}^{(l-1)}_{[i]} \tag{9}$$

Backpropagation

- Equation (9) tells us that the required derivative is obtained simply by multiplying the value of $\vec{\delta}_{[j]}^{l}$ by the value of $\vec{z}_{[i]}^{(l-1)}$.
- Thus, in order to evaluate the derivatives, we need only to calculate the value of $\vec{\delta}_{[j]}^{l}$ for each hidden and output unit in the network, and then apply (9).
- Calculating $\vec{\delta}_{[j]}^{m}$ for output units ($l = m$), is usually straightforward, since activation units $\vec{h}_{[j]}^{m}$ are directly observed in the loss expression.
- The same applies for shallow linear models.

Backpropagation

- To evaluate the $\vec{\delta}^{l}_{[j]}$ for hidden units, we again make use of the chain rule for partial derivatives:

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left(\frac{\partial L}{\partial \vec{h}^{l+1}_{[k]}} \times \frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} \right) \tag{10}$$

- The sum runs over all units $\vec{h}^{l+1}_{[k]}$ to which unit $\vec{h}^{l}_{[j]}$ sends connections.
- We assume that connections go only to consecutive layers in the network (from layer l to layer $(l+1)$).
- The units $\vec{h}^{l+1}_{[k]}$ could include other hidden units and/or output units.
- If we now substitute the definition of $\vec{\delta}^{l}_{[j]}$ given by (eq.7) into (eq.10), we get

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left(\vec{\delta}^{(l+1)}_{[k]} \times \frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} \right) \tag{11}$$

Backpropagation

- Now, for expression $\vec{h}^{l+1}_{[k]}$ we can go to its definition (eq.3):

$$\vec{h}^{(l+1)}_{[k]} = \left(\sum_i W^{l+1}_{[i,k]} \times \vec{z}^{l}_{[i]} \right) + \vec{b}^{(l+1)}_{[k]}$$

- Now, we replace (eq.4) ($\vec{z}^{l}_{[i]} = g(\vec{h}^{l}_{[i]})$) into previous equation and we obtain:

$$\vec{h}^{(l+1)}_{[k]} = \left(\sum_i W^{l+1}_{[i,k]} \times g(\vec{h}^{l}_{[i]}) \right) + \vec{b}^{(l+1)}_{[k]}$$

- Now when calculating $\frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}}$ all the terms in the summation where $i \neq j$ get canceled out.
- Hence:

$$\frac{\partial \vec{h}^{l+1}_{[k]}}{\partial \vec{h}^{l}_{[j]}} = W^{l+1}_{[j,k]} \times g'(\vec{h}^{l}_{[j]}) \tag{12}$$

Backpropagation

- Now, if we substitute (eq.12) into (eq.11)

$$\vec{\delta}^{l}_{[j]} \equiv \frac{\partial L}{\partial \vec{h}^{l}_{[j]}} = \sum_{k} \left(\vec{\delta}^{(l+1)}_{[k]} \times W^{l+1}_{[j,k]} \times g'(\vec{h}^{l}_{[j]}) \right) \tag{13}$$

- Since $g'(\vec{h}^{l}_{[j]})$ doesn't depend on k we can obtain the following backpropagation formula:

$$\vec{\delta}^{l}_{[j]} = g'(\vec{h}^{l}_{[j]}) \times \sum_{k} \left(\vec{\delta}^{(l+1)}_{[k]} \times W^{l+1}_{[j,k]} \right) \tag{14}$$

- Which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network.
[Bishop, 2006].

Backpropagation

The backpropagation procedure can be summarized as follows.

1. Apply an input vector $\vec{x}$ to the network and forward propagate through the network using (eq.3) and (eq.4) to find the activations of all the hidden and output units.
2. Evaluate the $\vec{\delta}^{m}_{[j]}$ for all the output units (recall that the derivatives involved here are easy to calculate).
3. Backpropagate the $\vec{\delta}^{(l+1)}_{[k]}$ using (eq.14) to obtain $\vec{\delta}^{l}_{[j]}$ for each hidden unit in the network. We go from higher to lower layers in the network.
4. Use (eq.9) ($\frac{\partial L}{\partial w^{l}_{[i,j]}} = \vec{\delta}^{l}_{[j]} \times \vec{z}^{(l-1)}_{[i]}$) to evaluate the required derivatives.

The Computation Graph Abstraction

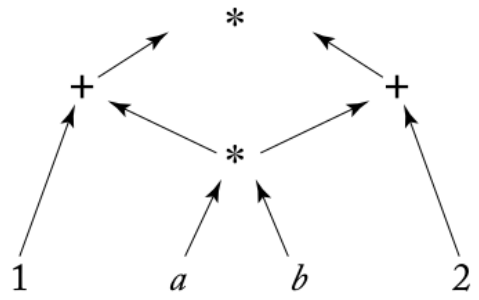
- One can compute the gradients of the various parameters of a network by hand and implement them in code.
- This procedure is cumbersome and error prone.
- For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012].
- A computation graph is a representation of an arbitrary mathematical computation (e.g., a neural network) as a graph.
- This abstraction will allow us computing the gradients from any kind of neural network architecture using the backpropagation algorithm.
- Previous formulation was restricted to feedforward networks.

The Computation Graph Abstraction

- A computation graph is a directed acyclic graph (DAG).
- Nodes correspond to mathematical operations or (bound) variables.
- Edges correspond to the flow of intermediary values between the nodes.
- The graph structure defines the order of the computation in terms of the dependencies between the different components.
- The graph is a DAG and not a tree, as the result of one operation can be the input of several continuations.

The Computation Graph Abstraction

- Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



- The computation of $a * b$ is shared.
- Since a neural network is essentially a mathematical expression, it can be represented as a computation graph.

The Computation Graph Abstraction

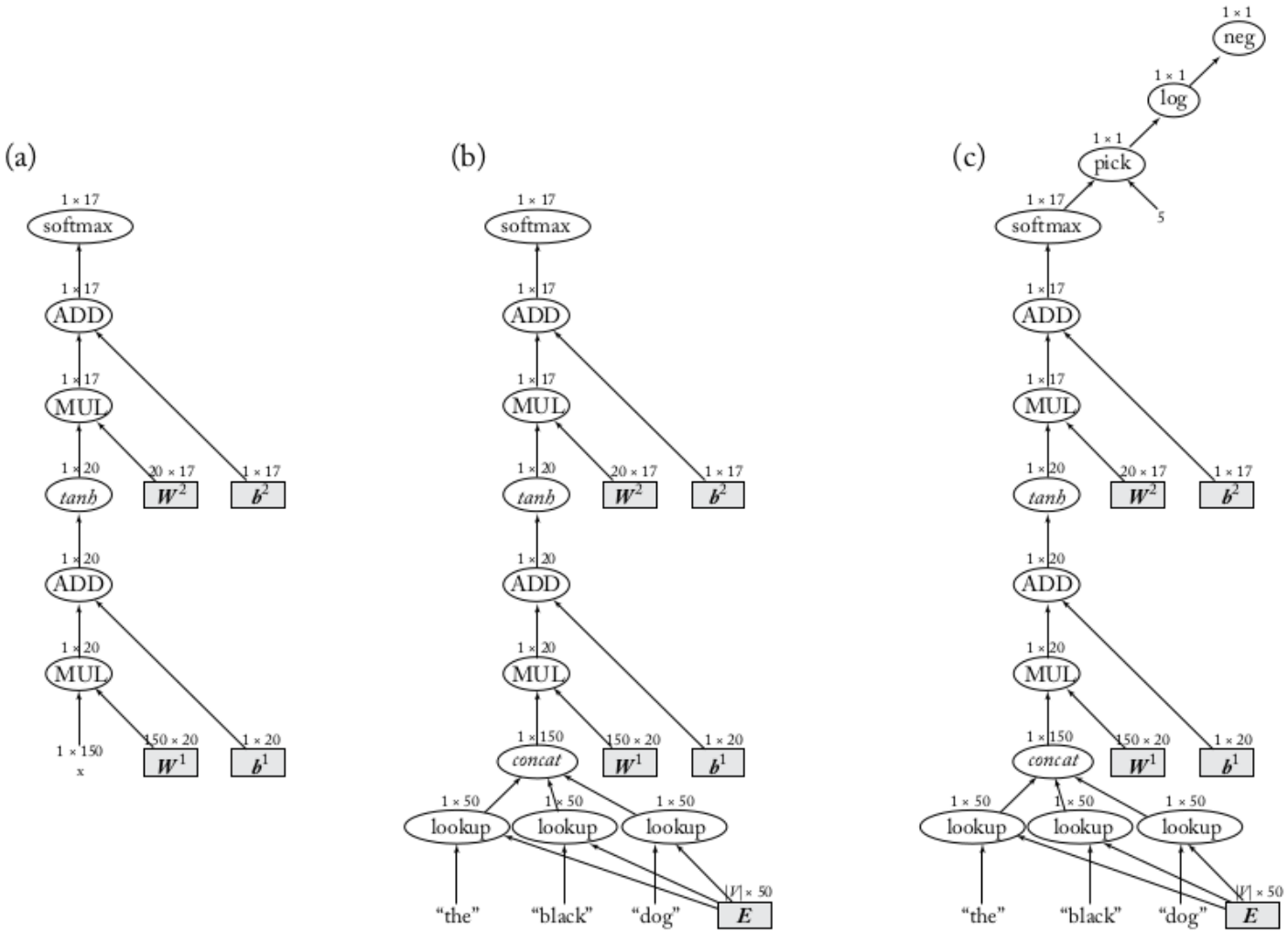


Figure 5.1: (a) Graph with unbound input. (b) Graph with concrete input. (c) Graph with concrete input, expected output, and a final loss node.

[1] Figure taken from: [Goldberg, 2017]

The Computation Graph Abstraction

- The figure above shows the computation graph for an MLP with one hidden-layer and a softmax output transformation.
- Oval nodes represent mathematical operations or functions, and shaded rectangle nodes represent parameters (bound variables).
- Network inputs are treated as constants, and drawn without a surrounding node.
- Input and parameter nodes have no incoming arcs, and output nodes have no outgoing arcs.
- The output of each node is a matrix, the dimensionality of which is indicated above the node.

The Computation Graph Abstraction

- This graph is incomplete: without specifying the inputs, we cannot compute an output.
- Figure 5.1b shows a complete graph for an MLP that takes three words as inputs, and predicts the distribution over part-of-speech tags for the third word.
- This graph can be used for prediction, but not for training, as the output is a vector (not a scalar) and the graph does not take into account the correct answer or the loss term.
- Finally, the graph in Figure 5.1c shows the computation graph for a specific training example, in which the inputs are the (embeddings of) the words “the,” “black,” “dog,” and the expected output is “NOUN” (whose index is 5).
- The pick node implements an indexing operation, receiving a vector and an index (in this case, 5) and returning the corresponding entry in the vector.

Forward Computation

- The forward pass computes the outputs of the nodes in the graph.
- Since each node's output depends only on itself and on its incoming edges, it is trivial to compute the outputs of all nodes.
- This is done by traversing the nodes in a topological order and computing the output of each node given the already computed outputs of its predecessors.
- More formally, in a graph of N nodes, we associate each node with an index i according to their topological ordering.
- Let f_i be the function computed by node i (e.g., multiplication, addition , etc.).

Forward Computation

- Let $\pi(i)$ be the parent nodes of node i , and $\pi^{-1}(i) = \{j | i \in \pi(j)\}$ the children nodes of node i (these are the arguments of f_i).
- Denote by $v(i)$ the output of node i , that is, the application of f_i to the output values of its arguments $\pi^{-1}(i)$.
- For variable and input nodes, f_i is a constant function and $\pi^{-1}(i)$ is empty.
- The computation-graph forward pass computes the values $v(i)$ for all $i \in [1, N]$.

Algorithm 1: Computation graph forward pass

for $i = 1$ **to** N **do**
  Let $a_1, \ldots, a_m = \pi^{-1}(i)$
  $v(i) \leftarrow f_i\left(v(a_1), \ldots, v(a_m)\right)$
end

Backward Computation (Backprop)

- The backward pass begins by designating a node N with scalar (1×1) output as a loss-node, and running forward computation up to that node.
- The backward computation computes the gradients of the parameters with respect to that node's value.
- Denote by $d(i)$ the quantity $\frac{\partial N}{\partial i}$.
- The backpropagation algorithm is used to compute the values $d(i)$ for all nodes i.
- The backward pass fills a table of values $d(1), \ldots, d(N)$ as shown in the following algorithm.

Algorithm 2: Computation graph backward pass (backpropagation)

$d(N) \leftarrow 1$; // $\frac{\partial N}{\partial N} = 1$

;

for *i = N-1* **to** *1* **do**

  $d(i) = \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$; // $\frac{\partial N}{\partial i} = \sum_{j \in \pi(i)} \frac{\partial N}{\partial j} \cdot \frac{\partial j}{\partial i}$

  ;

end

Backward Computation (Backprop)

- The backpropagation algorithm is essentially following the chain-rule of differentiation.
- The quantity $\frac{\partial f_j}{\partial i}$ is the partial derivative of $f_j(\pi^{-1}(j))$ w.r.t the argument $i \in \pi^{-1}(j)$.
- This value depends on the function f_j and the values $v(a_1), \ldots, v(a_m)$ (where $a_1, \ldots, a_m = \pi^{-1}(j)$) of its arguments, which were computed in the forward pass.
- Thus, in order to define a new kind of node, one needs to define two methods: one for calculating the forward value $v(i)$ based on the node's inputs, and the another for calculating $\frac{\partial f_j}{\partial i}$ for each $x \in \pi^{-1}(i)$.

Summary of the Computation Graph Abstraction

- Notice that the above formulation of backpropagation is equivalent to one given earlier in the class.
- Te computation graph abstraction allows us to:
  1. Easily construct arbitrary networks.
  2. Evaluate their predictions for given inputs (forward pass)
  3. Compute gradients for their parameters with respect to arbitrary scalar losses (backward pass or backpropagation).
- A nice property of the computation graph abstraction is that it allows computing the gradients for arbitrary networks (e.g., networks with skip-connections, shared weights, special loss functions, etc.)

[1] A comprehensive tutorial on the backpropagation algorithm over the computational graph abstraction can be found here: `https://colah.github.io/posts/2015-08-Backprop/`.

Derivatives of "non-mathematical" functions

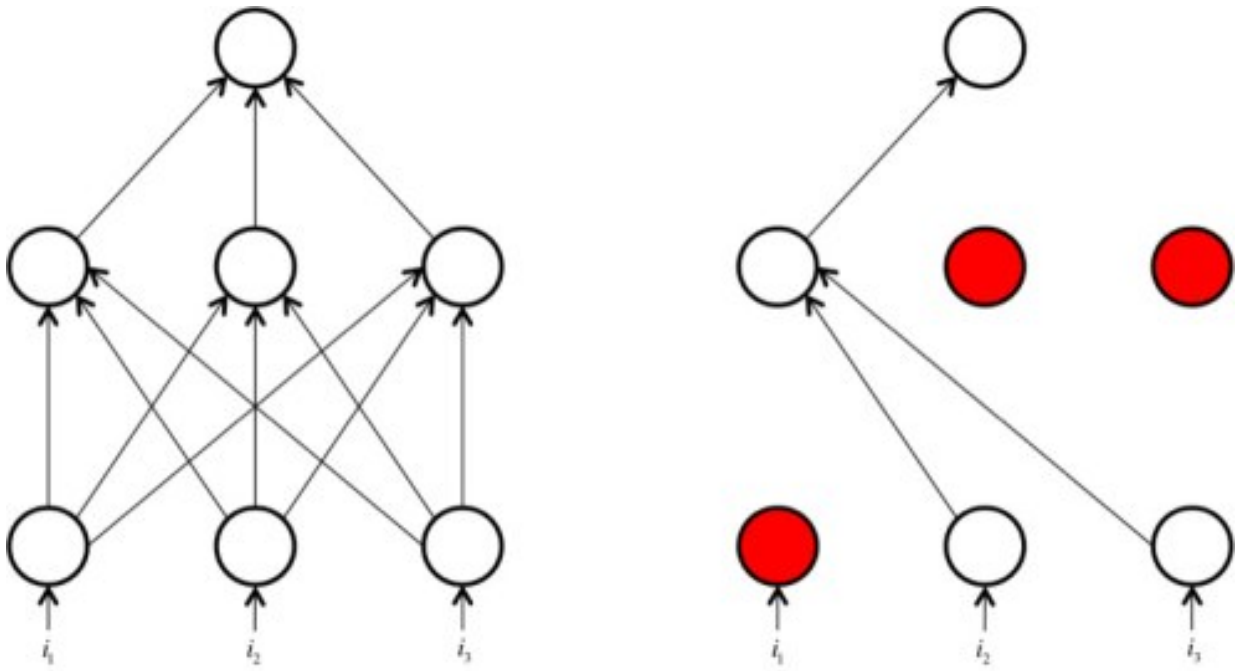
- Defining $\frac{\partial f_j}{\partial i}$ for mathematical functions such is as *log* or $+$ is straightforward.
- It may be challenging to think about the derivative of operations as as pick$(\vec{x}, 5)$ that selects the fifth element of a vector.
- The answer is to think in terms of the contribution to the computation.
- After picking the i-th element of a vector, only that element participates in the remainder of the computation.
- Thus, the gradient of pick$(\vec{x}, 5)$ is a vector $\vec{v}$ with the dimensionality of $\vec{x}$ where $\vec{v}_{[5]} = 1$ and $\vec{v}_{[i \neq 5]} = 0$.
- Similarly, for the function $\max(0, x)$ the value of the gradient is 1 for $x > 0$ and 0 otherwise.

Regularization and Dropout

- Multi-layer networks can be large and have many parameters, making them especially prone to overfitting.
- Model regularization is just as important in deep neural networks as it is in linear models, and perhaps even more so.
- The regularizers discussed for linear models, namely L_2 , L_1, and the elastic-net, are also relevant for neural networks.
- Another effective technique for preventing neural networks from overfitting the training data is **dropout training** [Srivastava et al., 2014].

Regularization and Dropout

- The dropout method is designed to prevent the network from learning to rely on specific weights.
- It works by randomly dropping (setting to 0) half of the neurons in the network (or in a specific layer) in each training example in the stochastic-gradient training.



[1] Figure taken from: https://www.kdnuggets.com/wp-content/uploads/drop-out-in-neural-networks.jpg

Deep Learning Frameworks

Several software packages implement the computation-graph model. All these packages support all the essential components (node types) for defining a wide range of neural network architectures.

- TensorFlow (`https://www.tensorflow.org/`): an open source software library for numerical computation using data-flow graphs originally developed by the Google Brain Team.
- Keras: High-level neural network API that runs on top of Tensorflow as well as other backends (`https://keras.io/`).
- PyTorch: open source machine learning library for Python, based on Torch, developed by Facebook's artificial-intelligence research group. It supports dynamic graph construction, a different computation graph is created from scratch for each training sample. (`https://pytorch.org/`)

Questions?

Thanks for your Attention!

References I


Bengio, Y. (2012).
Practical recommendations for gradient-based training of deep architectures.
In *Neural networks: Tricks of the trade*, pages 437–478. Springer.


Bishop, C. M. (2006).
Pattern recognition and machine learning.
springer.


Cybenko, G. (1989).
Approximation by superpositions of a sigmoidal function.
Mathematics of control, signals and systems, 2(4):303–314.


Glorot, X., Bordes, A., and Bengio, Y. (2011).
Deep sparse rectifier neural networks.
In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.


Goldberg, Y. (2017).
Neural network methods for natural language processing.
Synthesis Lectures on Human Language Technologies, 10(1):1–309.


Hornik, K., Stinchcombe, M., and White, H. (1989).
Multilayer feedforward networks are universal approximators.
Neural networks, 2(5):359–366.

References II

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014).
Dropout: a simple way to prevent neural networks from overfitting.
The journal of machine learning research, 15(1):1929–1958.