Deep learning book Ch. 6- Deep feedforward networks notes

James Chuang

February 24, 2017

Contents

My notes on chapter 6 of the Deep Learning Book on Deep Feedforward Networks.

Deep feedforward networks, aka ***feedforward neural networks***, aka ***multi-layer perceptrons*** are the quintessential deep learning models. The goal of a feedforward network: approximate some function f^*.

6.2 Gradient-Based Learning

- largest difference between linaer models and neural networks:
  - nonlinearity of a neural network causes most interesting loss functions to become non-convex
    - because of this, neural networks are usually trained by using iterative, gradient-based optimizers (rather than linear equation solvers/convex optimization/SVMs)
    - convex optimization converges starting from any initial parameters (in theory–in practice is robust but can encounter numerical problems)
      - in contrast, stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters
        - for feedforward neural networks, it is important to intialize all weights to small random values
        - biases may be initialized to zero or to small positive values
    - the training algorithm is almost always based on using the gradient to descent the cost function in one way or another
      - the specific algorithms are usually improvements of the stochastic gradient descent algorithm
    - gradient descent can also be used to train simpler models such as linear regression and support vector machines (common when the training set is extremely large)
      - the gradient can be obtained efficiently for a neural network using the back-propagation algorithm and its modern generalizations
    - to apply gradient-based learning to neural networks, need to choose a ***cost function*** and an ***output representation***

6.2.1 Cost Functions

- cost functions for neural networks are more or less then same as those for other parameteric models, e.g. linear models
- in most cases, our parametric model defines a distribution $p(\mathbf{x} \mid \mathbf{x}; \theta)$
  - we then use the principle of maximum likelihood, resulting in the cross-entropy between the training data and the model predictions as the cost function
- sometimes, take a simpler approach: rather than predicting a complete probability distribution over $\mathbf{y}$, merely predict some statistic of $\mathbf{y}$ conditioned on $\mathbf{x}$
  - use a specialized loss function to train a predictor of these estiamtes
- total cost function often combines a primary cost function with a regularization term

6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

- most modern neural networks are trained using maximum likelihood

- the cost function is then the negative-log likelihood, equivalently described as the cross-entropy between the training data and the model distribution:

$$J(\theta) = -\mathsf{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$$

  - the specific form of the cost function depends on the specific form of $\log p_{\text{model}}$
    - the expanded form typically yields some terms that do not depend on the model parameters and may be discarded
  - advantage of deriving the cost function from maximum likelihood: removes the burden of designing cost functions for each model:
    - specifying a model $p(\mathbf{y} \mid \mathbf{x})$ automatically determines a cost function $\log p(\mathbf{y} \mid \mathbf{x})$
- recurring theme throughout neural network design: the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm
  - functions that saturate (become very flat) undermine this objective because they make the gradient become very small
    - in many cases this happens because the activation functions used to produce the output of hidden units or output units saturate
    - negative log-likelihood helps avoid this problem for many models:
      - many output units involve an $\exp$ function that can saturate when its argument is very negative
        - the $\log$ function in negative log-likelihood undoes the $\exp$ of some output units
- an unusual property of the cross-entroyp cost used to perform maximum likelihood estimation: it usually does not have a minimum value when applied to the models commonly used in practice
  - for most discrete output variables, most models are parameterized in such a way that they cannot represent a probability of zero or one, but can come arbitrarily close to doing so (e.g. logistic regression)
  - for real-valued output variables (e.g. learning the variance of a Gaussian output distribution) then it becomes possible to assign extremely high density to the correct training set outputs, resulting in cross-entropy approaching negative infinity
    - regularization is needed to avoid overfitting like this

6.2.1.2 Learning Conditional Statistics

- instead of learning a full probability distribution $p(\mathbf{y} \mid \mathbf{x}; \theta)$, often want to learn just one conditional statistic of $\mathbf{y}$ given $\mathbf{x}$
  - e.g., may have a predictor $f(\mathbf{x}; \theta)$ that we wish to predict the mean of $\mathbf{y}$
- using a sufficiently powerful neural network, can think of the network as being able to represent any function f from a wide class of functions, with this class being limited only by features such as continuity and boundedness (rather than by having a specific parametric form)
  - can view the cost function as being a *functional* rather than just a function
    - ***functional***: a mapping from functions to real numbers
    - thus, can think of learning as choosing a function rather than merely choosing a set of parameters
    - can design the cost functional to have its minimum occur at some specific function we desire
      - e.g., design it to have its minimum lie on the function that maps $\mathbf{x}$ to $\mathsf{E}[\mathbf{y} \mid \mathbf{x}]$
      - solving an optimization problem requires ***calculus of variations***
        - two results derived from calculus of variations:
          1. solving the mean squared error optimization problem

$$f^* = \arg\min_f \mathsf{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|^2$$

             - yields

$$f^*(\mathbf{x}) = \mathsf{E}_{\mathbf{y}\sim p_{\text{data}}(\mathbf{y}\mid\mathbf{x})}[\mathbf{y}]$$

             - , so long as this function lies within the class we optimize over
               - i.e., if we could train on infinitely many samples from the true data-generating distribution, minimizng the mean squared error cost function gives a function that predicts the mean of $\mathbf{y}$ for each value of $\mathbf{x}$
          2. solving the mean absolute error optimization problem

$$f^* = \arg\min_f \mathsf{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|_1$$

- yields a function that predicts the **median** value of $\mathbf{y}$ for each $\mathbf{x}$, so long as such a functino may be described by the family of functions we optimize over
- mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization
  - some outputs units that saturate produce very small gradients when combined with these cost function
  - this is one reason that the cross-entropy cost function is more popular, even when it is not necessary to estimate an entire distribution $p(\mathbf{y} \mid \mathbf{x})$

6.2.2 Output Units

- choice of cost function is tightly couple with the choice of output unit
  - most of the time, we use cross-entropy loss between the data distribution and the model distribution
    - the choice of how to represent the output then determines the form of the cross-entropy function
- any kind of neural network unit that may be used as an output can also be used as a hidden unit
- suppose that the feedforward network provides a set of hidden features defined by $\mathbf{h} = f(\mathbf{x}; \theta)$
  - the role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform

6.2.2.1 Linear Units for Gaussian Output Distributions

- ***linear unit***: output unit based on an affine transformation with no nonlinearity
  - given features $\mathbf{h}$, a layer of linear output units produces a vector $\hat{y} = \mathbf{W}^T\mathbf{h} + \mathbf{b}$
  - often used to produce the mean of a conditional Gaussian distribution:

$$p(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I})$$

  - maximizing the log-likelihood is then equivalent to minimizing the mean squared error
  - max likelihood makes it straightforward to learn the covariance of the Gaussian, or to make the covariance of the Gaussian be a function of the input
    - however, covariace must be constrained to be a positive definite matrix for all inputs
      - difficult to satisfy such constraints with a linear output layer, so typically other output units are used to parameterize the covariance
  - linear units do not saturate, so they pose little difficulty for gradient-based optimization algorithms

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

- many tasks require predicting the value of a binar variable y, esp. two-class classification
  - the max likelihood approach: define a Bernoulli distribution over y conditioned $\mathbf{x}$
    - neural net only needs to predict $P(y = 1 \mid \mathbf{x}) \in [0, 1]$
      - could be satisfied with a thresholded linear unit, but this couldn't be trained effectively with gradient descent (the gradient is zero outside the unit interval)
      - a better approach which ensures that there is always a strong gradient whenever the model has the wrong answer:
        - based on sigmoid output units combined with maximum likelihood
          - ***sigmoid output unit***:

$$\hat{y} = \sigma\left(\mathbf{w}^T\mathbf{h} + b\right)$$

        - sigmoid output unit has two components: first, uses a linear layer to compute $z = \mathbf{w}^T\mathbf{h} + b$; then uses sigmoid activation function to convert z to a probability
        - how to define a probability distribution over y using the value z:
          - the sigmoid can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, then divide by an appropriate constant to obtain a valid probabilty distribution
            - begin with the assumption that the unnormalized log probabilities are linear in y and z:

$$\log \hat{P}(y) = yz$$
$$\hat{P}(y) = \exp(yz)$$
$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)}$$
$$P(y) = \sigma((2y-1)z) \qquad \text{I think this only holds because } y \in \{0,1\}?$$

- z variable defining such a distribution over binary variables is the ***logit***
- this approach to predicting the probabilities in log-space is natural to use with maximum likelihood learning
  - because the cost function used with max likelihood is $-\log P(y \mid \mathbf{x})$, the $\log$ in the cost function undoes the $\exp$ of the sigmoid
    - this keeps the saturation of the sigmoid from preventing gradient-based learning from making progress
  - the loss function for maximum likelihood learning of a Bernoulli parameterized by a sigmoid:

$$\begin{aligned} J(\theta) &= -\log P(y \mid \mathbf{x}) \\ &= -\log \sigma((2y-1)z) \\ &= \zeta((1-2y)z) \end{aligned}$$

6.2.2.3 Softmax Units for Multinoulli Output Distributions

6.2.2.4 Other Output Types

6.3 Hidden Units

6.3.1 Rectified Linear Units and Their Generalizations

6.3.2 Logistic Sigmoid and Hyperbolic Tangent

6.3.3 Other Hidden Units

6.4 Architecture Design

6.4.1 Universal Approximation Properties and Depth

6.4.2 Other Architectural Considerations

6.5 Back-propagation and Other Differentiation Algorithms

- ***forward propagation***:
  - inputs $\mathbf{x}$ provides initial information that propagates through hidden units and finally produces the prediction $\hat{\mathbf{y}}$
  - during training, forward propagation continues until it produces a scalar cost $J(\theta)$.
- ***back-propagation*** (aka ***backprop***):
  - information from the cost flows backwards through the network, in order to compute the gradient
  - refers only to the method for computing the gradient
    - another algorithm, e.g. stochastic gradient descent is used to perform learning using the gradient
  - in principle, can compute derivatives of any function (not just multi-layer neural networks)

- i.e. compute $\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{y})$ for an arbitrary function f
  - $\mathbf{x}$: a set of variables whose derivatives are desired
  - $\mathbf{y}$: an additional set of variables that are inputs to the function (but whose derivatives are not required)
  - most often, we want to calculate the gradient of the cost function w.r.t. the parameters: $\nabla_\theta J(\theta)$

6.5.1 Computational Graphs

- to discuss backprop, it's useful to first develop computational graph language
- let each node in the graph indicate a variable (a scalar, vector, matrix, tensor, or other)
- introduce the idea of an ***operation***- a simple function of one or more variables
  - graph language is accompanied by a set of allowable operations
  - functions more complicated than these operations may be described by composing many operations together
  - wlog, define an operation to *return only a single output variable*
    - the output variable could have multiple entries, e.g. a vector
- if a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y
  - we sometimes annotate the output node with the name of the operation applied, and other times omit the label when the operation is clear from context

6.5.2 Chain Rule of Calculus

- (not to be confused with the chain rule of probability)
- used to compute the derivatives of functions formed by composing other functions whose derivatives are known
- backprop is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient
- let:
  - $x \in \mathbb{R}$
  - $f, g : \mathbb{R} \mapsto \mathbb{R}$
  - $y = g(x)$
  - $z = f(g(x)) = f(y)$
- then,
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- this can be generalized beyond the scalar case:
  - suppose that:
    - $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$
    - $g : \mathbb{R}^m \mapsto \mathbb{R}^n$
    - $f : \mathbb{R}^n \mapsto \mathbb{R}$
    - $\mathbf{y} = g(\mathbf{x})$
    - $z = f(\mathbf{y})$
  - then,
    - $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$
    - in vector notation:
      - $\nabla_{\mathbf{x}}(z) = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}}(z)$
        - here, $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g
        - from this, we see that the gradient of a variable $\mathbf{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by a gradient $\nabla_{\mathbf{y}}(z)$
        - the backprop algorithm consists of performing such a Jacobian-gradient product for each operation in the graph
- usually, backprop is not applied to vectors, but tensors of arbitrary dimensionality
  - conceptually, this is the same as backprop with vectors
    - imagine flattening each tensor into a vector before running backprop, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this view, backprop is still just multiplying Jacobians by gradients
    - denote the gradient of a value z w.r.t. a tensor $\mathcal{X}$ by $\nabla_{\mathcal{X}}(z)$
      - the indices into $\mathcal{X}$ have multiple coordinates

- e.g., a 3-D tensor is indexed by three coordinates
- we abstract this away by using a single variable i to represent the complete tuple of indices
- for all possible tuples i, $(\nabla_{\mathcal{X}}(z))_i = \frac{\partial z}{\partial \mathcal{X}_i}$

- the chain rule as it applies to tensors:

$$\mathcal{Y} = g(\mathcal{X}), z = f(\mathcal{Y}), \text{then}$$

$$\nabla_{\mathcal{X}}(z) = \sum_j (\nabla_{\mathcal{X}} \mathcal{Y}_j) \frac{\partial z}{\partial \mathcal{Y}_j}$$

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- using chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar w.r.t. any node in the computational graph that produced that scalar
- actually evaluating the expression introduces extra considerations
  - many subexpressions may be repeated several times within the overall expression for the gradient
    - for complicated graphs, computing these expressions multiple times can make a naive implementation of the chain rule infeasible
- first, consider a computational graph describing how to compute a single scalar $u^{(n)}$ (e.g., the loss on a training example)
  - want to obtain the gradient w.r.t. the n_i input notes $u^{(1)}$ to $u^{(n_i)}$
    - in the application of back-propagation for computing gradient descent over parameters:
      - $u^{(n)}$ will be the cost associated with an example/minibatch
      - $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model
  - assume the nodes are ordered such that we can compute their output one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$
    - each node $u^{(i)}$ is associated with an operation $f(i)$ and is computed by evaluating the function

$$u^{(i)} = f\left(\mathbb{A}^{(i)}\right)$$

    - here, $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.
  - ***Algorithm 6.1: forward propagation computation***
    - **for** $i = 1, \ldots, n_i$ **do**
      - $u^{(i)} \leftarrow x_i$
    - **end for**
    - **for** $i = n_i + 1, \ldots, n$ **do**
      - $\mathbb{A}^{(i)} \leftarrow \left\{u^{(j)} \mid j \in \text{Pa}\left(u^{(i)}\right)\right\}$
      - $u^{(i)} \leftarrow f^{(i)}\left(\mathbb{A}^{(i)}\right)$
    - **end for**
    - **return** $u^{(n)}$
  - the above algorithm specifies the forward propagation computation, which could be put in a graph $\mathcal{G}$
  - to perform backprop, we can construct a computation graph that depends on $\mathcal{G}$ and adds to it an extra set of nodes
    - these form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$
      - each node of $\mathcal{B}$ computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in \text{Pa}\left(u^{(i)}\right)} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

      - the subgraph $\mathcal{B}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$
        - the edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$
        - for each node, a dot product is performed between:
          - the gradient already computed w.r.t. notes $u^{(i)}$ that are children of $u^{(j)}$
          - the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$

- the amount of computation required for performing backprop scales *linearly* with the number of edge in $\mathcal{G}$
  - computation for each edge corresponds to computing a partial derivative (of one node w.r.t. its parents) as well as performing one multiplication and one addition
- ***Algorithm 6.2: simplified version of backprop*** (for computing the derivatives of $u^{(n)}$ w.r.t the variables in the graph)
  - simplifications: all variables are scalars, and compute derivatives of all nodes in the graph
  - run forward propagation (Algorithm 6.1) to obtain network activations
  - initialize `gradtable`: a data structure that will store the computed derivatives
    - $\texttt{gradtable}\left[u^{(i)}\right] = \frac{\partial u^{(n)}}{\partial u^{(i)}}$
  - $\texttt{gradtable}\left[u^{(n)}\right] \leftarrow 1$
  - **for** $j = n - 1$ down to 1 **do**
    - $\texttt{gradtable}\left[u^{(j)}\right] \leftarrow \sum_{i:j \in \text{Pa}\left(u^{(i)}\right)} \texttt{gradtable}\left[u^{(i)}\right] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
      - this computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in \text{Pa}\left(u^{(i)}\right)} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$
  - **end for**
  - **return** $\{\texttt{gradtable}\left[u^{(i)}\right] \mid i = 1, \ldots, n_i\}$
- backprop is designed to reduce the number of common subexpressions without regard to memory
  - performs on the order of one Jacobian product per node in the graph, thus avoiding exponential explosion in repeated subexpressions
    - this can be seen from that fact that backprop visits each edge from node $u^{(j)}$ to node $u^{(i)}$ of the graph exactly once in order to obtain the associated partial derivative $\frac{\partial u^{(j)}}{\partial u^{(i)}}$
  - other algorithms may be able to avoid more subexpressions by performing simplifications on the computational graph, or may conserve memory by recomputing rather than storing some subexpressions

6.5.4 Back-Propagation Computation in Fully-Connected MLP

- consider the specific graph associated with a fully-connected multi-layer MLP
  - ***Algorithm 6.3: forward propagation*** (compute gradients of cost J w.r.t. parameters $\mathbf{W}$ and $\mathbf{b}$, with single training example $\mathbf{x}$)
    - maps parameters to the supervised loss $L\left(\hat{\mathbf{y}}, \mathbf{y}\right)$ associated with a single training example $(\mathbf{x}, \mathbf{y})$
      - $\hat{\mathbf{y}}$ is the output of the neural network when $\mathbf{x}$ is provided as input
    - **require:** l, the network depth
    - **require:** $\mathbf{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
    - **require:** $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
    - **require:** $\mathbf{x}$, the input to process
    - **require:** $\mathbf{y}$, the target output
      - $\mathbf{h}^{(0)} = \mathbf{x}$
      - **for** $k = 1, \ldots, l$ **do**
        - $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$
        - $\mathbf{h}^{(k)} = f\left(\mathbf{a}^{(k)}\right)$
      - **end for**
      - $\hat{\mathbf{y}} = \mathbf{h}^{(l)}$
      - $J = L\left(\hat{\mathbf{y}}, \mathbf{y}\right) + \lambda \Omega(\theta)$
  - ***Algorithm 6.4: backprop*** on the same network of Algorithm 6.3 (compute the gradients on the activations $\mathbf{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer)
    - after the forward computation, compute the gradient on the output layer:
    - $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$
    - **for** $k = l, l - 1, \ldots, 1$ **do**
      - convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplciation if f is element-wise):
        - $\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'\left(\mathbf{a}^{(k)}\right)$
      - compute gradients on weights and biases (including the regularization term, where needed):
        - $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta)$

        - $\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g}\mathbf{h}^{(k-1)T} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)$
      - propagate the gradients w.r.t. the next lower-level hidden layer's activations:
        - $\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)T}\mathbf{g}$
    - **end for**
- these are simple, specialized algorithms
  - modern software implementations are based on a generalized form of backprop that can accomodate any computational graph by explicitly manipulating a data structure for representing symbolic computation

6.5.5 Symbol-to-Symbol Derivatives

- algebraic expressions and computational graphs both operate on ***symbols***- variables that do not have specific values
  - these algebraic and graph-based representations are called ***symbolic representations***
    - when using or training a neural network, symbolic inputs are replaced with a specific ***numeric*** value
- some approaches to backprop use a ***"symbol-to-number"*** approach to differentiation:
  - take a computational graph and a set of numerical values for the inputs to the graph
  - return a set of numerical values describing the gradient at those input values
  - e.g. Torch, Caffe
- another approach is to use a ***"symbol-to-symbol"*** approach to differentiation:
  - take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives
  - e.g. Theano, Tensorflow
  - primary advantage: derivatives are described in the same language as the original expression
    - because the derivatives are just another computational graph, it is possible to run backprop again, differentiating the derivatives in order to obtain higher derivatives
    - every node can be evaluated as soon as its parents' values are available, allowing us to avoid specifying exactly when each operation should be computed
  - symbol-to-symbol subsumes the symbol-to-number approach
    - symbol-to-number performs the exact same computations as are done in the graph build by the symbol-to-symbol approach
    - the key difference: symbol-to-number does not expose the graph

6.5.6 General Back-Propagation

- to compute the gradient of some scalar z w.r.t. one of its ancestors $\mathbf{x}$ in the graph:
  - begin by observing that the gradient w.r.t. z is given by $\frac{dz}{dz} = 1$
  - we can then compute the gradient w.r.t. each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z
  - continue multipying by Jacobians traveling backwards through the graph in this way until we reach $\mathbf{x}$
    - for any node that may be reached by going backwards from z through two or more paths, simply sum the gradients arriving from different paths at that node
- more formally:
  - each node in the graph $\mathcal{G}$ corresponds to a variable
    - for maximum generality, this variable is a tensor $\mathcal{V}$
      - in general, a tensor can have any number of dimensions (subsuming scalars, vectors, matrices)
  - assume each variable $\mathcal{V}$ is associated with the following subroutines:
    - $\texttt{getoperation}(\mathcal{V})$
      - returns the operation that computes $\mathcal{V}$
      - represented by the edges coming into $\mathcal{V}$ in the computational graph
    - $\texttt{getconsumers}(\mathcal{V}, \mathcal{G})$
      - returns the list of variables that are children of $\mathcal{V}$ in the computational graph $\mathcal{G}$
    - $\texttt{getinputs}(\mathcal{V}, \mathcal{G})$
      - returns the list of variables that are parents of $\mathcal{V}$ in the computational graph $\mathcal{G}$
  - each operation $\texttt{op}$ is also associated with a $\texttt{bprop}$ operation

- bprop computes a Jacobian-vector product, i.e. the chain rule, $\nabla_{\mathcal{X}}(z) = \sum_j (\nabla_{\mathcal{X}} \mathcal{Y}_j) \frac{\partial z}{\partial \mathcal{Y}_j}$
  - e.g. consider a matrix multiplication operation creating a variable $\mathbf{C} = \mathbf{AB}$
    - let the gradient of a scalar z w.r.t. $\mathbf{C}$ is given by $\mathbf{G}$
    - the matrix multiplication operation is responsible for defining two backprop rules, one for each of its input arguments:
      - if we call bprop to request the gradient w.r.t. $\mathbf{A}$ given that the gradient on the output is $\mathbf{G}$, bprop must state that the gradient w.r.t. $\mathbf{A}$ is given by $\mathbf{GB}^T$
      - if we call bprop to request the gradient w.r.t. $\mathbf{B}$, bprop must state that the gradient is $\mathbf{A}^T\mathbf{G}$
      - the backprop algorithm does not need to know any differentiation rules
        - it only needs to call each operation's bprop rules with the right arguments
    - formally, op.bprop $(\texttt{inputs}, \mathcal{X}, \mathcal{G})$ must return

$$\sum_i (\nabla_{\mathcal{X}} \texttt{op.f}(\texttt{inputs})_i) \mathcal{G}_i$$

      - inputs: list of inputs supplied to the operation
      - op.f: the mathematical function that the operation implements
      - $\mathcal{X}$: the input whose gradient we wish to compute
      - $\mathcal{G}$: the gradient on the output of the operation
    - the op.brop method should always treat all of its inputs as distinct from each other, even if they are not
      - e.g. if two copies of x are input to compute x^2, the derivative w.r.t. each input should still be x
    - software implementations of backprop usually provide both the operations and their bprop methods
      - if building a new implementation of backprop or adding a custom operation to an existing library, usually need to derive the op.bprop method for the new operation
- ***Algorithm 6.5: the back-propagation algorithm***
  - this is the outermost skeleton, for simple setup and cleanup
  - most of the important work happens in the buildgrad subroutine of Algorithm 6.6
  - **require:** $\mathbb{T}$, the target set of variables whose gradients must be computed
  - **require:** $\mathcal{G}$, the computational graph
  - **require:** z, the varaiable to be differentiated
    - Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$
    - Initialize gradtable, a data structure associating tensors to their gradients
    - $\texttt{gradtable}[z] \leftarrow 1$
    - **for** $\mathcal{V}$ in $\mathbb{T}$ **do**
      - $\texttt{buildgrad}(\mathcal{V}, \mathcal{G}, \mathcal{G}', \texttt{gradtable})$
    - **end for**
    - return gradtable restricted to $\mathbb{T}$
- ***Algorithm 6.6: the inner loop subroutine*** $\texttt{buildgrad}(\mathcal{V}, \mathcal{G}, \mathcal{G}', \texttt{gradtable})$
  - **require** $\mathcal{V}$, the variable whose gradient should be added to $\mathcal{G}$ and gradtable
  - **require** $\mathcal{G}$, the graph to modify
  - **require** $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient
  - **require** gradtable, a data structure mapping nodes to their gradients
    - **if** $\mathcal{V}$ is in gradtable **then**
      - return $\texttt{gradtable}[\mathcal{V}]$
    - **end if**
    - $i \leftarrow 1$
    - **for** $\mathcal{C}$ in getconsumers$(\mathcal{V}, \mathcal{G})$ **do**
      - $\texttt{op} \leftarrow \texttt{getoperation}(\mathcal{C})$
      - $\mathcal{D} \leftarrow \texttt{buildgrad}(\mathcal{V}, \mathcal{G}, \mathcal{G}', \texttt{gradtable})$
      - $\mathbf{G}^{(i)} \leftarrow \texttt{op.bprop}(\texttt{getinputs}(\mathcal{C}, \mathcal{G}'), \mathcal{V}, \mathcal{D})$
      - $i \leftarrow i + 1$
    - **end for**
    - $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
    - $\texttt{gradtable}[\mathcal{V}] = \mathbf{G}$
    - insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$

- **return** $\mathbf{G}$

- with the algorithm specified, we can examine the computational cost
  - assume that each operation evaluation has roughly the same cost, and then analyze the computational cost in terms of the number of operations executed
    - note: we refer to an operation as the fundamental unit of the computational graph, though each could consist of several arithmetic operations (e.g. a matrix multiplication is one operation but can be many arithmetic operations)
    - then, computing a gradient with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations
      - the backprop algorithm adds one Jacobian-vector product, expressed with $O(1)$ nodes, per edge in the original graph
      - since the computational graph is a directed acyclic graph, it has at most $O(n^2)$ edges
      - for the graphs used in practice, the situation is better:
        - most neural networks are roughly chain-structured, causing backprop to have $O(n)$ cost
      - the potentially exponential cost can be seen by expanding and rewriting the recursive chain rule non-recursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in \text{Pa}\left(u^{(i)}\right)} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path}\left(u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}\right), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}$$

i.e. sum over all paths from $u^{(j)}$ to $u^{(n)}$, multiplying all derivatives along each path

      - the number of paths from node j to node n can grow exponentially in the length of these paths
        - therefore, the number of terms in the sum above (which is the number of paths), can grow exponentially with the depth of the forward propagation graph
          - the large computational cost is incurred when $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is recalculated many times
          - backprop is a ***dynamic programming*** strategy to avoid these recomputations
            - it can be thought of as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u^{(n)}}{\partial u^{(i)}}$
            - each node in the graph has a corresponding slot in a table to store the gradient for that node
              - by filling in these table entries in order, backprop avoids repeating evaluating common subexpressions

6.5.7 Backprop for MLP Training

- consider a simple multilayer perceptron with a single hidden layer
  - train with minibatch stochastic gradient descent
  - backprop is used to compute the gradient of the cost on a single minibatch
    - $\mathbf{X}$: a design matrix representing a minibatch of examples from the training set
    - $\mathbf{y}$: vector of associated class labels
    - $\mathbf{H} = \max\left\{0, \mathbf{X}\mathbf{W}^{(1)}\right\}$
      - simplify by assuming no biases in the model
    - assume the existence of `relu` operation that computes $\max\{0, \mathbf{Z}\}$ elementwise
    - predictions of the unnormalized log probabilities over classes are given by $\mathbf{H}\mathbf{W}^{(2)}$
    - assume the existence of a `crossentropy` operation that computes the cross-entropy between the targets $\mathbf{y}$ and the probability distribution defined by these unnormalized log probabilities
      - the resulting cross-entropy defines the cost J_{MLE}
        - minimizing the cross-entropy = maximum likelihood estimation of the classifier
      - also include a regularization term:

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)}\right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)}\right)^2 \right)$$

6.5.8 Complications

6.5.9 Differentiation outside the Deep Learning Community

6.5.10 Higher-Order Derivatives

6.6 Historical Notes