Deep Feedforward Networks

Pankaj Kumar, http://datacabinet.info

Introduction

Feedforward neural networks are called networks because they are represented by composing together many functions as an acyclic graph. The have no feedback connections.

f(x) = f4(f3(f2(f1(x)))) means f1 is the first layer, f2 is the second and f3 the third.

By training a neural network with training data, we are finding an f(x) that is approximating f*(x) which is the real distribution.

The output layer is the last layer (f4) that produces a y for every x that is the closest approximation to f*(x)

The behavior of the other layers f1, f2 and f3 is not directly determined by the training data. The neural net has to decide how to use these to approximate f*(x).

Introduction

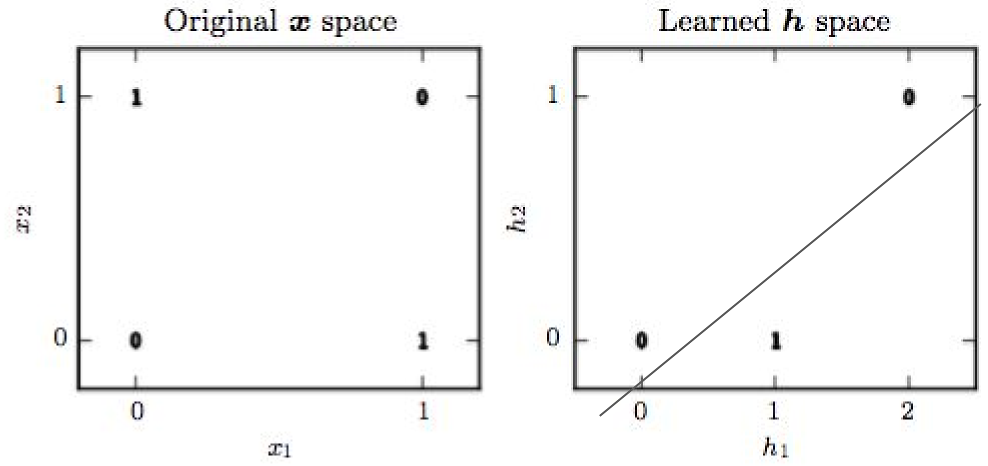
Width of the network - The number of neurons in each layer.

Each layer can be thought of as multiple vector to scalar functions that act in parallel and produce activations after typically applying a non-linearity.

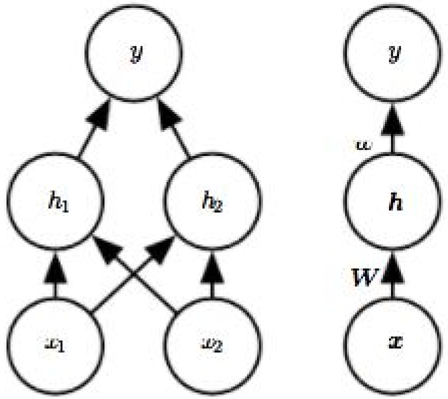
Linear Regression to Neural networks

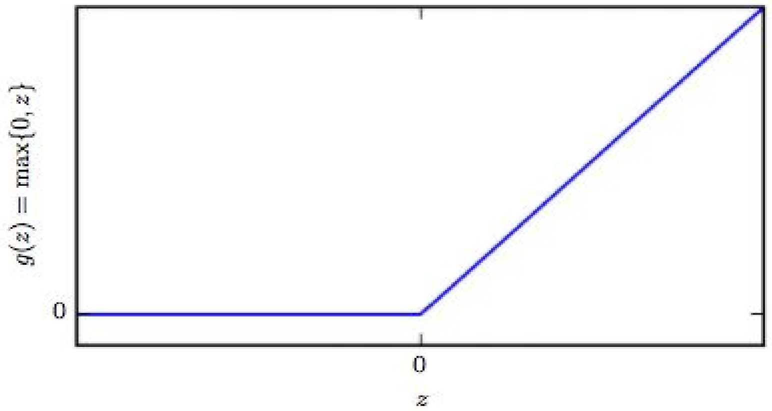
1. Problem with Linear regression is that if the f*(x) is inherently non-linear(for eg x ^ 2), it cannot be approximated by a linear function.
2. We can transform the input to a non-linear space or use the kernel trick ϕ(x) (RBF function). This can give enough capacity to the classifier but test case generalization is bad. Generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.
3. Other option is to manually engineer ϕ(x) which has been a dominant method so far but does not scale to a variety of problems.
4. The strategy of deep learning is to find ϕ itself. First we learn θ which are the parameters of the network(depth, width etc). This chooses a function within a broad class of functions. Then we find the weights that train that function. This lets go of the convexity requirement in other class of functions

Example (XOR function)



Example (Network and RELU)





f(x; W , c, w, b) = w max{0, Wx + c} + b.

Example(continued)

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \qquad \boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \qquad \text{XW + c} \quad \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \qquad \text{RELU} \quad \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}.$$

Gradient Based Learning

Because neural networks use non linearities, most interesting cost functions become non-convex. This means that neural networks use iterative methods to run down the value of the cost function. It does not have a closed form solution.

Gradient Based Learning: Cost: Max Likelihood

Simply negative log likelihood if we are predicting a distribution over the training set.

$J(\theta) = -E(x,y \sim p\{data\})\ [\log p\{model\}(y \mid x)].$

A lot of times, the expansion of the above term has terms that do not depend on θ and need not be optimized.

If a model has an exp function at the output layer, it can get saturated (low gradient) at very low values. This can be fixed by the Max likelihood cost function

Also, the max likelihood cost function, there is no minimum value for common models. For eg, in the logistic regression case if one class has a high probability and other has low probability, cross entropy can keep going down to negative

Gradient Based Learning: Cost: Max Likelihood

Sometimes instead of learning the full probability distribution of y, we need to learn some other statistics of y. For eg, for each x we might be interested in possible mean or median value of y.

From calculus of variations, which optimizes over functions, these results are useful:

if we could train on infinitely many samples from the true data generating distribution,
minimizing the mean squared error cost function gives a function that predicts the
mean of y for each value of x. Similarly, for the mean absolute value error predicts the median or y for each value of x.

But mean square error and mean absolute error saturate quickly when used with gradient based learning and thus maximum likelihood is used.

Gradient Based Learning: Output(or hidden) Units

Linear Units : If the output is gaussian, linear units can be used to learn the mean and the covariance matrix. Maximum likelihood and least mean square are both equivalent here. Also, gradient descent can be easily used because linear unit gradients do not saturate.

Sigmoid Units: If the output is binary, then it is hard to use a linear unit because it may product any real value. The neural net needs to output 0 or 1. An output of 15 will not mean anything.

Another construction could be $P(y = 1 \mid x) = \max(0, \min(1, wh + b))$ but this this hard to train using gradient descent as the gradients are 0 almost everywhere

Gradient based learning: Output Units : Sigmoid

Sigmoid units(cont) : Sigmoid can be used to model a binary problem. After a linear transformation $z = ax + b$. z could be treated as an unnormalized probability for getting 1. c can treated as an unnormalized probability of getting 0. Now, we can normalize it using $e ^ c / \{e ^ c + e ^ (ax + b)\}$, $e ^ (ax + b) / \{e ^ c + e ^ (ax + b)\}$. $e ^(ax + (b - c)) / \{1 + e ^ (ax + b - c)$. b -c is just another constant (d) . This can thus become $1 / \{ 1 + e ^ -(ax + d)\}$. This is the sigmoid function. Because of the exponentiation, the max likelihood becomes easy to learn and the gradients do not saturate.

Gradient based Learning: Output Units: Softmax

When there are inputs predicting multiple classes, we can use a softmax. The name comes from a soft version(differentiable version) of the argmax function that is not differentiable.

A softmax is similar to sigmoid and can be [W X + b] scores for c classes: z1, z2 .. zc

Similar to sigmoid, the probabilities are : exp (zi) / (exp (z1) + exp(z2) … exp(zc)) (One of the parameters can be fixed)

Similar to sigmoid, it can be trained with max log likelihood.

Gradient based learning: Other Output Types

Not very common.

In general, we can think of the neural network as representing a function $f(x;\theta)$. The outputs of this function are not direct predictions of the value
y. Instead, $f(x;\theta) = \omega$ provides the parameters for a distribution over y
. Our loss function can then be interpreted as $-\log p(y; \omega(x))$.

Hidden Units

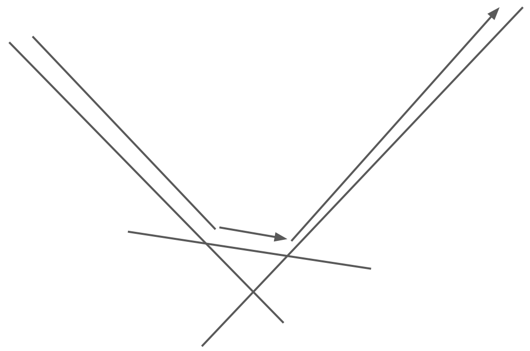
RELU - Problems when x < 0

Absolute RELU

Leaky RELU alpha fixed at like .5

Parametric RELU where alpha gets learnt

Maxout takes k lines and uses the max of it for any x. The parameters are learnt



Hidden Units : Sigmoid and tanh

$g(z) = \sigma(z)$

$g(z) = \tanh(z)$

These units are not widely used as hidden units because they saturate for half of the domain.

When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid. It resembles the identity function more closely, in the sense that
$\tanh(0) = 0$ while $\sigma(0) = .5$. Because tanh is similar to the identity function near 0, training a deep neural network resembles training a linear modelso long as the activations of the network can be kept small. This makes training the tanh network easier.

Hidden Units:

There are many other activation functions which work as well. For eg cos function.

Sometimes no activation(or an identity activation) can be used as well. This just leads to a linear classifier but it can save a number of parameters.

Radial basis function, Hard Tanh, Softplus are also used but they are not very popular.

Architecture: How many units and how they connect

Most neural networks are organized into groups called layers. Each layer is a function of the one that preceded it.

h(1)= g(1)(W(1) x + b(1))
h(2)= g(2)(W(2) h(1) + b(2))

h2(h1(x))

Height and width of the network is important.
Deeper networks have more capacity but are harder to train.

Architecture: Single hidden layer universal Approximator

Linear functions are convex but can only approximate a linear function. A non-linear function you would assume is specific to the problem at hand. Neural networks as a universal approximator come to rescue:

Specifically, the universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

Architecture: Single hidden layer universal Approximator

The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.

For our purposes it suffices to say that any continuous function on a closed and bounded subset of Rn is Borel measurable and therefore may be approximated by a neural network. Also, the result holds for most activation functions including RELU.

Architecture: Limitations of Universal Approximator

Given a function, there exists a neural net but given a set of training points, it cannot generalize to test points arbitrarily.

Learning may fail because the learning algorithm may not be able to find the correct parameters or training algorithm might find the wrong function because of overfitting

It does not say how many hidden units will be required and some studies site an exponential number which is not very useful. There exist families of functions that can be approxiamted with networks with depth d but shallower networks need exponential number of nodes.

Theoretical results for deep architectures

Some families of functions that take exponential number of hidden units for single layer of hidden units can be represented well when depth is increased.

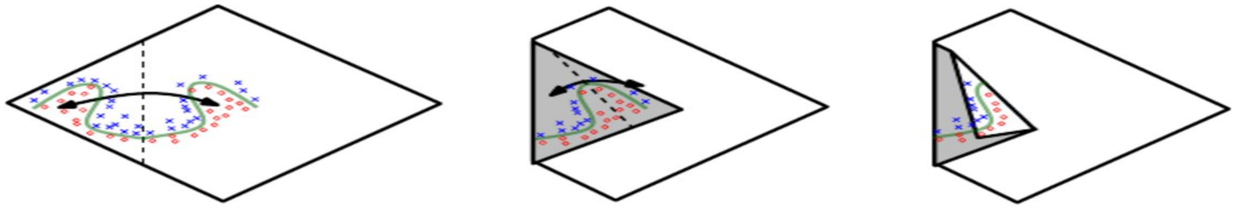


Figure 6.5: An intuitive, geometric explanation of the exponential advantage of deeper rectifier networks formally by Montufar *et al.* (2014). *(Left)*An absolute value rectification unit has the same output for every pair of mirror points in its input. The mirror axis of symmetry is given by the hyperplane defined by the weights and bias of the unit. A function computed on top of that unit (the green decision surface) will be a mirror image of a simpler pattern across that axis of symmetry. *(Center)*The function can be obtained by folding the space around the axis of symmetry. *(Right)*Another repeating pattern can be folded on top of the first (by another downstream unit) to obtain another symmetry (which is now repeated four times, with two hidden layers). Figure reproduced with permission from Montufar *et al.* (2014).

Theoretical results for deep architectures

More precisely, the main theorem in Montufar et al. (2014) states that the number of linear regions carved out by a deep rectifier network with d inputs, depth l, and n units per hidden layer, is

$O\left(\left({}^{n}C_{d}\right)^{(d(l-1))} * n^{d}\right)$

i.e., exponential in the depth l. In the case of maxout networks with k filters per unit, the number of linear regions is

$O\left(k^{(l-1)+d}\right)$

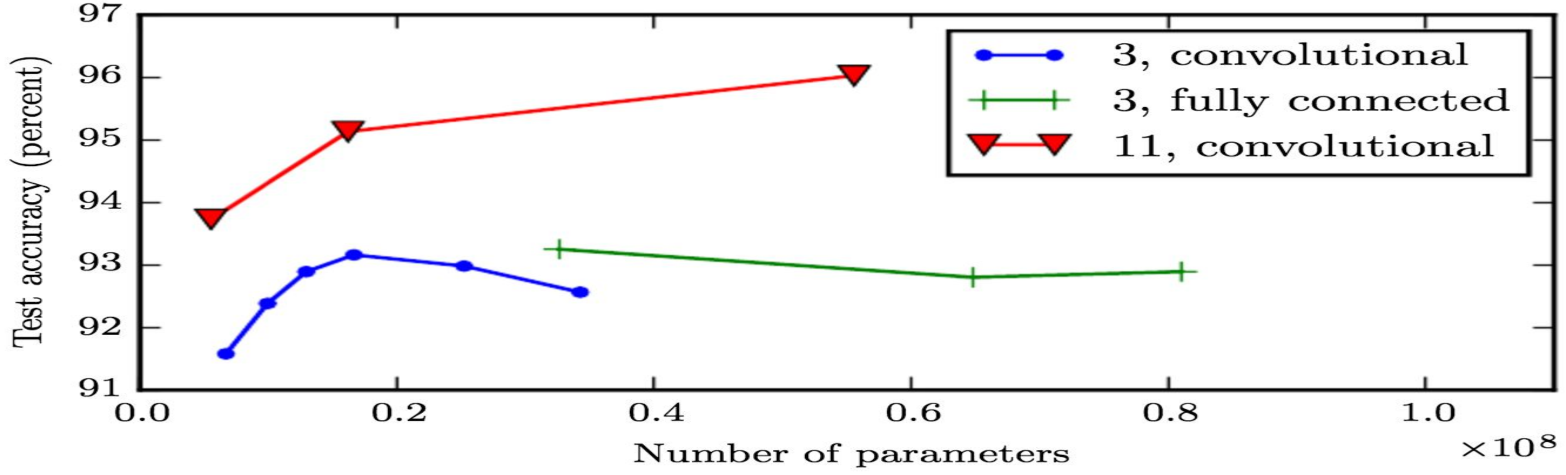


Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

Other architectural considerations

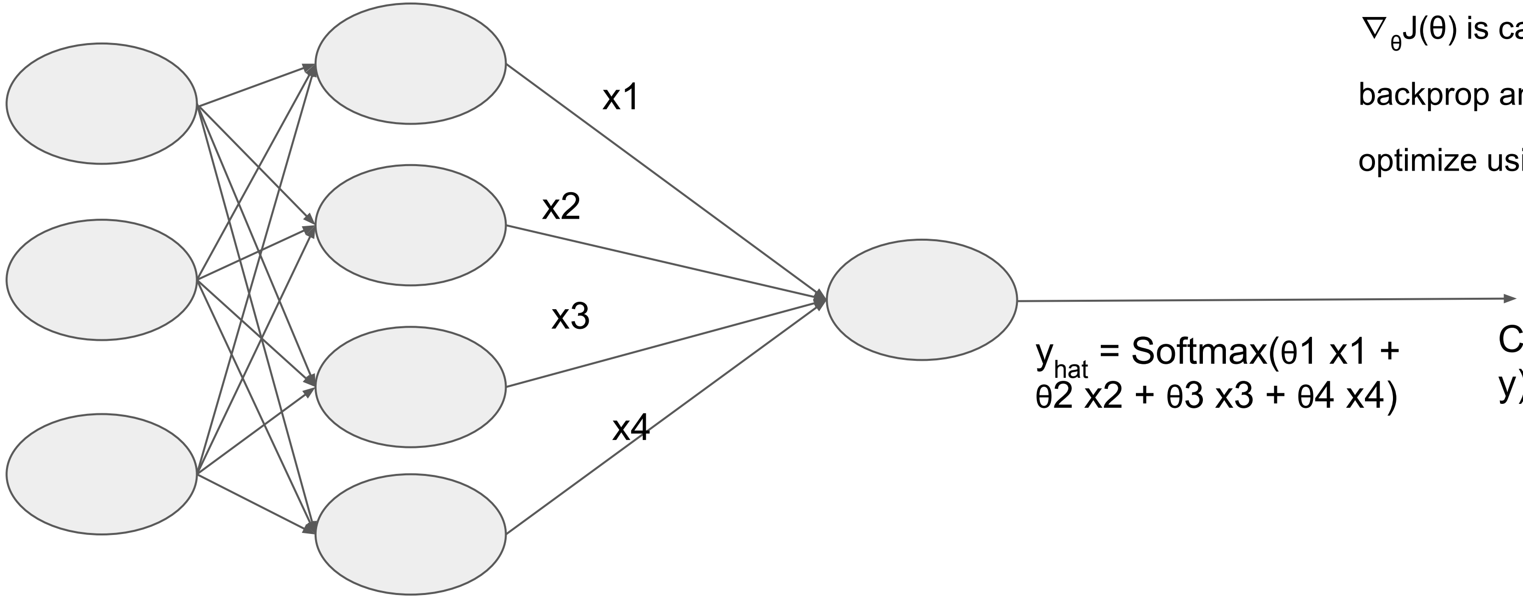
Specialized architectures: CNN, RNN

How to connect: Not all units connected to all units in the next layer

Units connected to layers beyond the second layer

Back Propagation - Forward pass / Backward Pass/ SGD

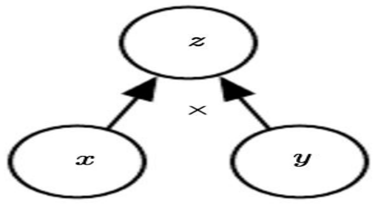
$\nabla_{\theta} J(\theta)$ is calculated by backprop and is used to optimize using SGD



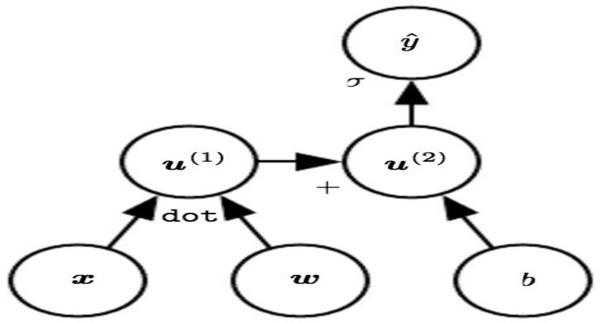
y_{hat} = Softmax($\theta 1$ x1 + $\theta 2$ x2 + $\theta 3$ x3 + $\theta 4$ x4)

Cost = $J_{\theta}(y_{hat}, y)$

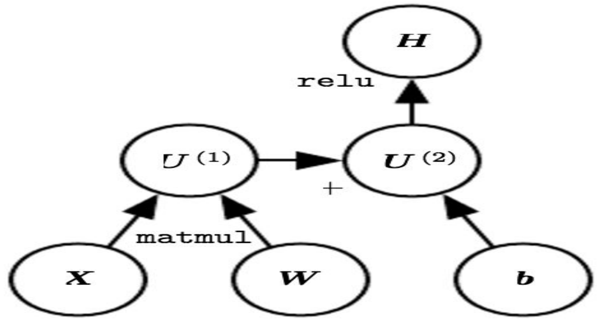
Computational graph



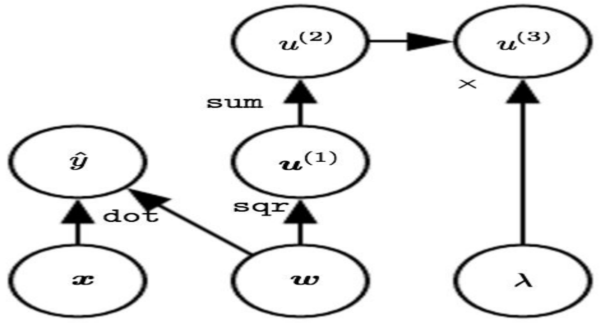
(a)



(b)



(c)



(d)

Chain Rule

Simple Chain Rule: y =g(x) and z=f(g(x)) =f(y)

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

Matrix Chain Rule: y =g(x) and z=f(g(x)) =f(y) x ∈ Rm , y ∈ Rn

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}.$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}}\, z,$$

Also, this can be easily extended to tensors. Tensors can just be treated like vectors.

Chain Rule: Forward pass

During the computation of the gradient, many subexpressions may be repeated multiple times. Computing it again and again could even lead to repeating computations exponential number of times.

```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \dots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \dots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$
```

Chain Rule: Backward Pass

```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ do
    Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
    Compute gradients on weights and biases (including the regularization term, where needed):
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
    Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$
end for
```

Symbol to symbol approach: Theano and TF

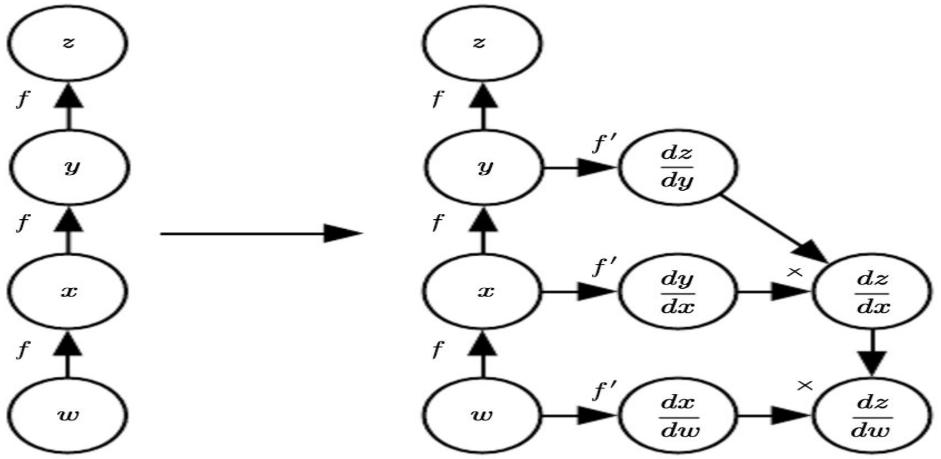


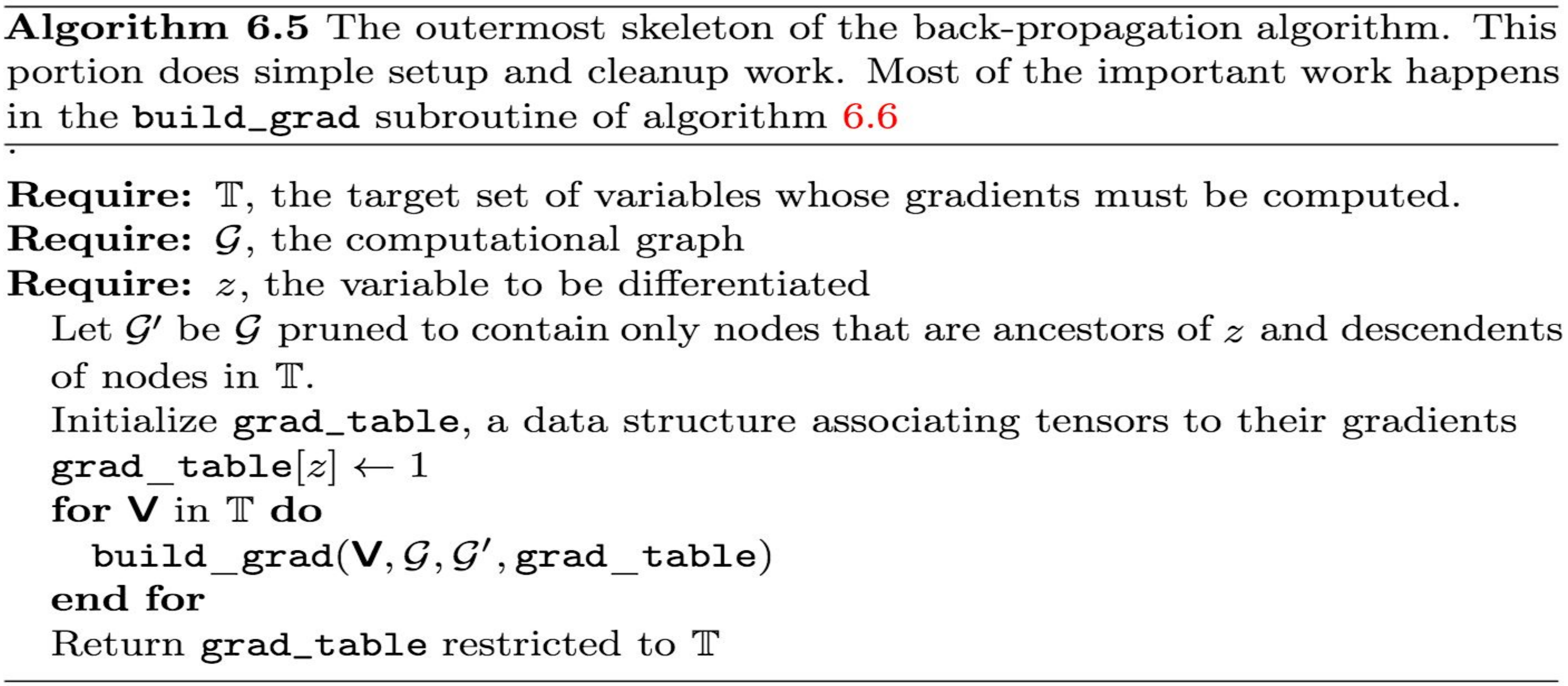
Figure 6.10: An example of the symbol-to-symbol approach to computing derivatives. In this approach, the back-propagation algorithm does not need to ever access any actual specific numeric values. Instead, it adds nodes to a computational graph describing how to compute these derivatives. A generic graph evaluation engine can later compute the derivatives for any specific numeric values. *(Left)*In this example, we begin with a graph representing $z = f(f(f(w)))$. *(Right)*We run the back-propagation algorithm, instructing it to construct the graph for the expression corresponding to $\frac{dz}{dw}$. In this example, we do not explain how the back-propagation algorithm works. The purpose is only to illustrate what the desired result is: a computational graph with a symbolic description of the derivative.

Symbol to symbol approach: Caffe and Torch

They do not expose the full derivative graph but only give the numerical derivative wrt to the weights

Because TF and Theano just add nodes and make a new computation graph, it can be used to compute higher derivatives

General Back Propagation



Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the **build_grad** 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 variable 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 grad_table, a data structure associating tensors to their gradients
  grad_table[$z$] $\leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ do
    build_grad($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, grad_table)
  end for
  Return grad_table restricted to $\mathbb{T}$
```

```
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: grad_table, a data structure mapping nodes to their gradients
  if $\mathsf{V}$ is in grad_table then
    Return grad_table[$\mathsf{V}$]
  end if
  $i \leftarrow 1$
  for $\mathsf{C}$ in get_consumers($\mathsf{V}, \mathcal{G}'$) do
    op $\leftarrow$ get_operation($\mathsf{C}$)
    $\mathsf{D} \leftarrow$ build_grad($\mathsf{C}, \mathcal{G}, \mathcal{G}'$, grad_table)
    $\mathsf{G}^{(i)} \leftarrow$ op.bprop(get_inputs($\mathsf{C}, \mathcal{G}'$), $\mathsf{V}, \mathsf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathsf{G} \leftarrow \sum_i \mathsf{G}^{(i)}$
  grad_table[$\mathsf{V}$] $= \mathsf{G}$
  Insert $\mathsf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathsf{G}$
```

Example

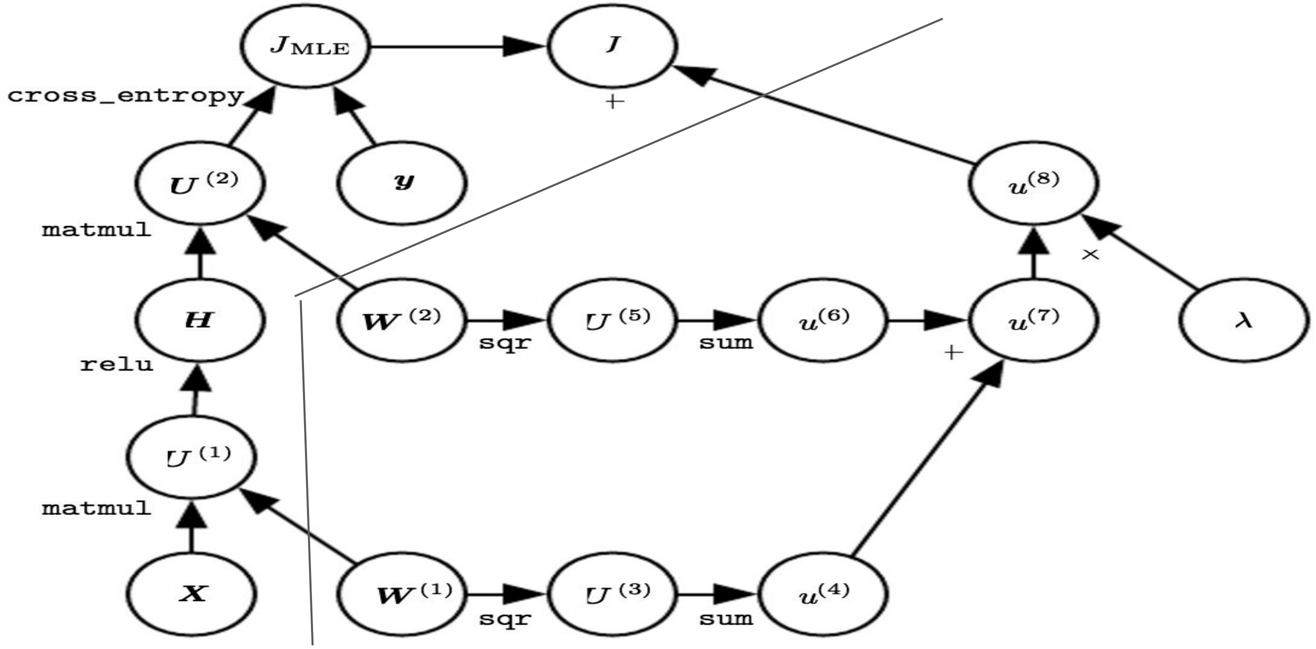


Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

Complications

Too much memory

Undefined gradients

Different data types: int, float 64 32 bit

More than one tensor as output. For eg, if we want the max value and index, we might need 2 pass.

Differentiation outside deep learning community

Reverse mode accumulation is a broader class of techniques of which backpropagation is a member.

The subexpression of the chain rule can be computed in a different order. Determining the order that will lead to the minimum computation is NP-Hard

Some optimizations come from observation. For eg, the differentiation of the softmax function can be simply expressed as y_{hat} - y. But chain rule will not catch that. Theano does some improvements like these.

When the forward computation graph has a single output and each partial derivative can be computed in constant amount of computation, the backward pass can be done in the same order of computations as the forward pass. And it can be optimized

Differentiation outside deep learning community

. The relationship between forward mode and backward mode is analogous to the relationship between left-multiplying versus right-multiplying a sequence of matrices, such as

. Forward accumulation is used when the number of outputs > than the number of inputs.

Higher order methods

Higher order derivatives are hard to find in deep learning because of the computational overhead. For a function f: Rn -> R , the Hessian is of shape n,n

Instead of explicitly computing the Hessian, the typical deep learning approach is to use Krylov methods. Krylov methods are a set of iterative techniques for performing various operations like approximately inverting a matrix or finding approximations to its eigenvectors or eigenvalues, without using any operation other than matrix-vector products.

$$Hv = \nabla_{x[}(\nabla_x f(x))v]$$