Deep Forward Networks

Dr. Josif Grabocka

ISMLL, University of Hildesheim

Deep Learning

Outline

What is a Deep Forward Network (DFN)?

- **Feedforward networks**, **feedforward neural networks** or **multilayer perceptrons**

- Given a function $\mathbf{y} = \mathbf{f}^*(\mathbf{x})$ that maps input $\mathbf{x}$ to category $\mathbf{y}$
- A DFN defines a parametric mapping $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \theta)$ with parameters θ
- Aim is to learn θ such as $\mathbf{f}(\mathbf{x}, \theta)$ *best* approximates $\mathbf{f}^*(\mathbf{x})$!

Why *Feedforward*?

- Given a Feedforward Network $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \theta)$
    - Input $\mathbf{x}$, then pass through a chain of steps before outputting $\mathbf{y}$
- No feedback exists between the chains of steps
    - Feedback connections yield the **Recurrent Neural Network**
- Example $f^1(x)$, $f^2(x)$ and $f^3(x)$ can be chained as:
    - $f(x) = f^3(f^2(f^1(x)))$
        - f^1 is the first layer, or the **input** layer
        - f^2 is the second layer, or a **hidden** layer
        - f^3 is the last layer, or the **output** layer
- Number of hidden layers define the **depth** of the network
- Dimensionality of the hidden layers defines the **width** of the network

Why *Neural*?

- Loosely inspired by neuroscience, hence Artificial Neural Network
- Each hidden layer node resembles a neuron
- Input to a neuron are the synaptic connections from the previous attached neuron
- Output of a neuron is an aggregation of the input vector
- Signal propagates forward in a chain of "Neuron"-to-"Neuron" transmissions
- However, modern Deep Learning research is steered mainly by mathematical and engineering principles!

Why *Network*?

- A feed-forward network is an **acyclic directed graph**, but
  - Graph nodes are structured in layers
  - Directed links between nodes are **parameters/weights**
  - Each node is a computational functions
  - No inter-layer and intra-layer connections (but possible)
  - Input to the first layer is given (the features $\mathbf{x}$)
  - Output is the computation of the last layer (the target $\hat{\mathbf{y}}$)

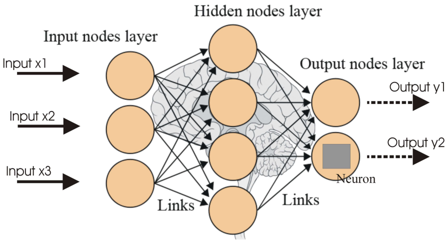


Figure 1 : FNN, Source `www.analyticsvidhya.com`

Nonlinear Mapping

- We can easily solve linear regression, but not every problem is linear.
- Can the function $f(x) = (x+1)^2$ be approximated through a linear function?

Nonlinear Mapping

- We can easily solve linear regression, but not every problem is linear.
- Can the function $f(x) = (x+1)^2$ be approximated through a linear function?
- Yes, but only if we **map** the feature x into a new space:

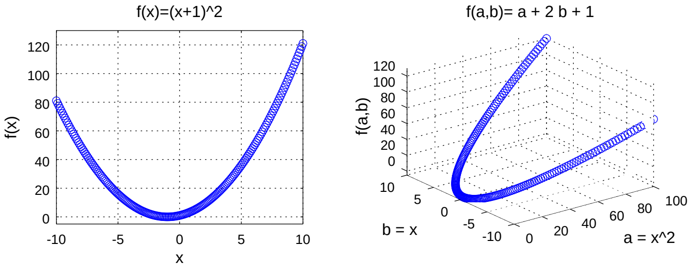


Figure 2 : Mapping feature x into a new dimensionality $x \rightarrow \phi(x) = (a, b)$

Nonlinear Mapping (II)

- Which mapping $\phi(x)$ is the best?

There are various ways of designing $\phi(x)$:

1. Hand-craft (manually engineer) $\phi(\mathbf{x})$
2. Use a very generic $\phi(\mathbf{x})$, RBF or polynomial expansion
3. Parametrize and learn the mapping $\mathbf{f}(\mathbf{x}; \theta, \mathbf{w}) := \phi(\mathbf{x}, \theta)^{\mathbf{T}}\mathbf{w}$

Deep Forward Networks follow the third approach, where:

- the hidden layers (weights θ) learn the mapping $\phi(x, \theta)^T$
- the output layer (weights w) learns the function $f(x; \theta, w)$

Outline

Introduction

Feedforward Computations

Output and Hidden Units

Back-propagation

An example - Learn XOR

- XOR is a function:

x_1	x_2	$y = f^*(x)$
0	0	0
0	1	1
1	0	1
1	1	0

- Can we learn a DFN $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \theta)$ such that f resembles f^*?
- Our dataset $\mathcal{X} = \{[0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T\}$
- Leading to the optimization:

$$\begin{aligned} \underset{\theta}{\operatorname{argmax}} \quad & J(\theta) \\ J(\theta) \quad = \quad & \frac{1}{4} \sum_{x \in \mathcal{X}} (f^*(x) - f(x, \theta))^2 \end{aligned}$$

An example - Learn XOR (2)

- We will learn a simple DFN with one hidden layer:

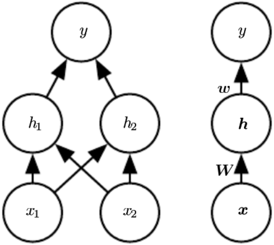


Figure 3 : Left: Detailed, Right: Compact, Source: Goodfellow et al., 2016

- Two functions are chained $h = f^1(x; W, c)$ and $y = f^2(y, w, b)$
  - For n-th instance: Hidden-layer $h_i^{(n)} = g\left(W_{:,i}^T x^{(n)} + c_i\right)$
  - For n-th instance: output layer: $\hat{y}_n = w^T h^{(n)} + b$
  - $W \in \mathbb{R}^{2\times 2}, c \in \mathbb{R}^{2\times 1}, w \in \mathbb{R}^{2\times 1}, b \in \mathbb{R}$

Rectified Linear Unit

The rectified linear unit (ReLU) is defined by the activation function $g(z) = \max\{0, z\}$, i.e.:

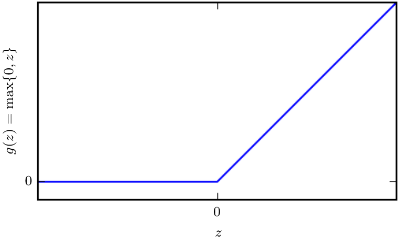


Figure 4 : The ReLU activation, Source: Goodfellow et al., 2016

Yielding the overall function:

$$\hat{y} = w^T \max\left\{0, W^T x + c\right\} + b$$

"Deus ex machina" solution?

Suppose I *magically* found out that:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \; c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \; w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \; b = 0$$

We would later on see an optimization technique called back-propagation to learn the network parameters.

XOR Solution - Hidden Layer Computations

$$
\begin{aligned}
h_1^{(1)} &= g\left(W_{:,1}^T x_1 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + 0\right) = g\left(0\right) = 0\\
h_2^{(1)} &= g\left(W_{:,2}^T x_1 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} - 1\right) = g\left(-1\right) = 0\\
h_1^{(2)} &= g\left(W_{:,1}^T x_2 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} + 0\right) = g\left(1\right) = 1\\
h_2^{(2)} &= g\left(W_{:,2}^T x_2 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} - 1\right) = g\left(0\right) = 0\\
h_1^{(3)} &= g\left(W_{:,1}^T x_3 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + 0\right) = g\left(1\right) = 1\\
h_2^{(3)} &= g\left(W_{:,2}^T x_3 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} - 1\right) = g\left(0\right) = 0\\
h_1^{(4)} &= g\left(W_{:,1}^T x_4 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + 0\right) = g\left(2\right) = 2\\
h_2^{(4)} &= g\left(W_{:,2}^T x_4 + c\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} - 1\right) = g\left(1\right) = 1
\end{aligned}
$$

XOR Solution - Output Layer Computations

$$\begin{aligned}
\hat{y}^{(1)} &= w^T h^{(1)} + b = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0 = 0 \\
\hat{y}^{(2)} &= w^T h^{(2)} + b = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 = 1 \\
\hat{y}^{(3)} &= w^T h^{(3)} + b = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 = 1 \\
\hat{y}^{(4)} &= w^T h^{(4)} + b = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + 0 = 0
\end{aligned}$$

The computations of the final layer match exactly those of the XOR function.

Outline

Gradient-Based Learning - Maximum Likelihood

The loss/cost can be expressed in probabilistic terms as

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

We early saw that assuming normality $p_{\text{model}}(y \mid x) = \mathcal{N}(y; f(x,\theta), I)$

$$J(\theta) = \frac{1}{2}\mathbf{E}_{x,y\sim\hat{p}_{data}} ||y - f(x,\theta)||^2 + \text{const}$$

Solving for the optimal DFN parameters:

$$\theta^{\text{opt}} =: \underset{\theta}{\operatorname{argmax}}\, \mathbf{E}_{x,y\sim\hat{p}_{data}} ||y - f(x,\theta)||^2$$

Yields a function that outputs the mean: $f(x,\theta^{\text{opt}}) = \mathbf{E}_{x,y\sim\hat{p}_{data}(y|x)}[y]$

Output Units - Gaussian Output Distribution

- Affine transformation with no nonlinearity
  - Given features h, produces $\hat{y} = w^T h + b$
- Used to produce the mean of a conditional Gaussian distribution
  - $p(y \mid x) = \mathcal{N}(y; \hat{y}, I)$

Bernoulli Output Distributions

- Binary target variables follow a Bernoulli distribution
  $P(y=1)=p,\ P(y=0)=1-p$
- Train a DFN such that $\hat{y}=f(x,\theta)\in[0,1]$
- Naive Option: Clip a linear output layer:
  - $P(y=1\,|\,x)=\max\left\{0,\min\left\{1,w^T h+b\right\}\right\}$
- What is the problem with the clipped linear output layer?

Bernoulli Output Distributions (2)

- Use a smooth sigmoid output unit:

$$\begin{aligned} \hat{y} &= \sigma(z) = \frac{e^z}{e^z + 1} \\ z &= w^T h + b \end{aligned}$$

- The loss for a DFN $f(x, \theta)$ with a sigmoid output is:

$$J(\theta) = \sum_{n=1}^{N} -y_n \log(f(x_n, \theta)) - (1 - y_n) \log(1 - f(x_n, \theta))$$

- Also called as Cross-entropy Cost Function

Multinoulli Output Distribution

- For multi-category targets $\hat{y}_i = P(y = i|x),\ i \in \{1, \ldots, C\}$
- Let the unnormalized log probability be defined as

$$\begin{aligned} z_i &= w_i^T h + b \\ z_i &= log \tilde{P}(y = i|x) \end{aligned}$$

- Yielding the normalized probability estimation:

$$P(y = i|x) \approx \text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- Minimizing the log-likelihood loss:

$$\begin{aligned} J(\theta) &= \sum_{n=1}^{N} -1_{y_n=i} \log P(y = i|x) \\ J(\theta) &= -\sum_{n=1}^{N} 1_{y_n=i} \left(z_i - \log \sum_j e^{z_j} \right) \end{aligned}$$

Types of Hidden Units

- Question: Can we use a linear activation $h = W^T x + b$?

Types of Hidden Units

- Question: Can we use a linear activation $h = W^T x + b$?
- Remember the most used hidden layer is ReLU:

$$h = g(W^T x + b) = \max(0, W^T x + b)$$

- Alternatively, the sigmoid function:

$$h = \sigma(z)$$

- or, the hyperbolic tangent:

$$h = \tanh(z) = 2\sigma(2z) - 1$$

Architecture of Hidden Layers

A DFN with L hidden layers:

$$\begin{aligned} h^{(1)} &= g^{(1)}({W^{(1)}}^T x + b^{(1)}) \\ h^{(2)} &= g^{(2)}({W^{(2)}}^T h^{(1)} + b^{(2)}) \\ &\dots \\ h^{(L)} &= g^{(L)}({W^{(L)}}^T h^{(L-1)} + b^{(L)}) \end{aligned}$$

Different layers can have different activation functions.

Outline

Computational Graphs

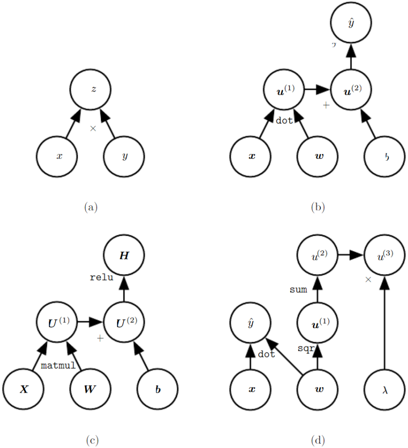


Figure 5 : a) multiplication, b) logistic regression prediction, c) ReLU, d) linear regression prediction and regularization, Source: Goodfellow et al., 2016

Chain-rule of Calculus

- Suppose $y = g(x)$ and $z = f(g(x)) = f(y)$, then

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

- In the vector case, suppose $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, and $y = g(x), g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ together with $z = f(y), f : \mathbb{R}^n \rightarrow \mathbb{R}$:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

- Compactly written using the Jacobian matrix $\frac{\partial y}{\partial x} \in \mathbb{R}^{n \times m}$ as

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

Backpropagation in Computational Graphs

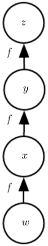


Figure 6 : x=f(w), y=f(x), z=f(y), Source: Goodfellow et al., 2016

Subexpression $f(w)$ is repeated:

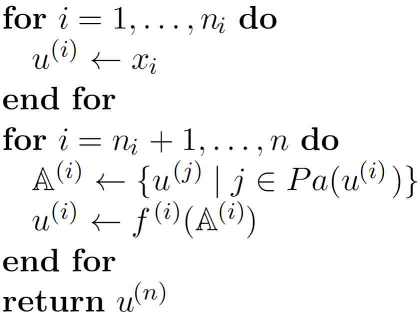
$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}\\ &= f'(y)f'(x)f'(w)\\ &= f'(f(f(w)))f'(f(w))f'(w)\end{aligned}$$

Backpropagation in Computational Graphs (2)

- Assume we want to compute a scalar $u^{(n)}$, e.g. loss of an instance
- Need to compute gradient w.r.t. n_i **input** nodes $u^{(1)}, \ldots, u^{(n_i)}$, i.e.
- Need to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}, i \in \{1, \ldots, n_i\}$
- We assume the nodes are ordered such that the computations are sequential, i.e. starting from $u^{(n_i+1)}$ until $u^{(n)}$
- Let $\mathbb{A}^{(i)}$ be the set of parent/predecessor nodes to $u^{(i)}$:
    - $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$

Backpropagation in Computational Graphs (3)

The feed-forward steps are:



```
for $i = 1, \ldots, n_i$ do
    $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ do
    $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
    $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$
```

Figure 7 : Feedforward in Computational Graphs, Source: Goodfellow et al., 2016

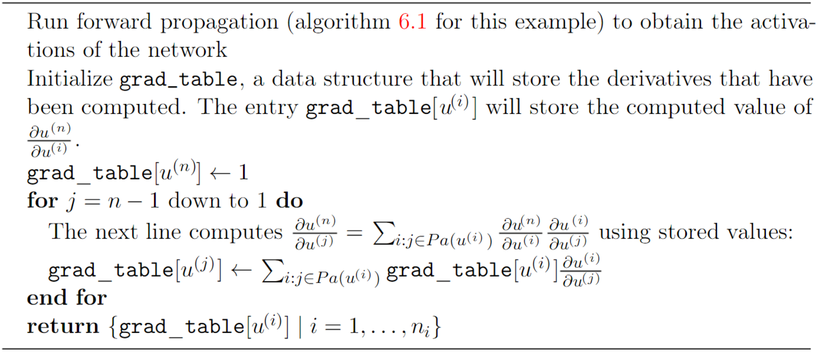
Backpropagation in Computational Graphs (4)

- The back-propagation is based on the chain-rule:

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

- However, to avoid repeated computations, each node *i* computes and stores $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ in a table.
- In that way, gradients of final node with respect to successor nodes are not re-computed

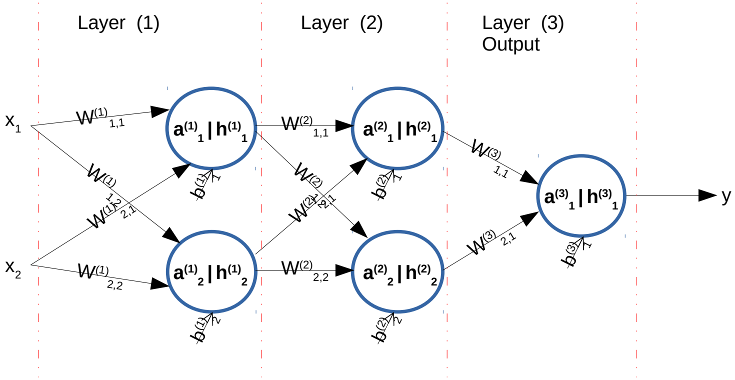
Back-propagation in Computational Graphs (5)



```
Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network
Initialize grad_table, a data structure that will store the derivatives that have been computed. The entry grad_table[$u^{(i)}$] will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.
grad_table[$u^{(n)}$] $\leftarrow 1$
for $j = n - 1$ down to 1 do
    The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:
    grad_table[$u^{(j)}$] $\leftarrow \sum_{i:j \in Pa(u^{(i)})}$ grad_table[$u^{(i)}$]$\frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return {grad_table[$u^{(i)}$] $\mid i = 1, \dots, n_i$}
```

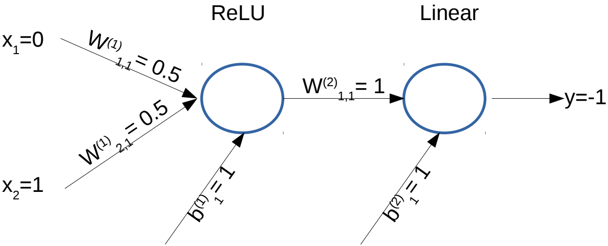
Figure 8 : Back-propagation in Computational Graphs, Source: Goodfellow et al., 2016

From Computational Graphs to MLP - An example



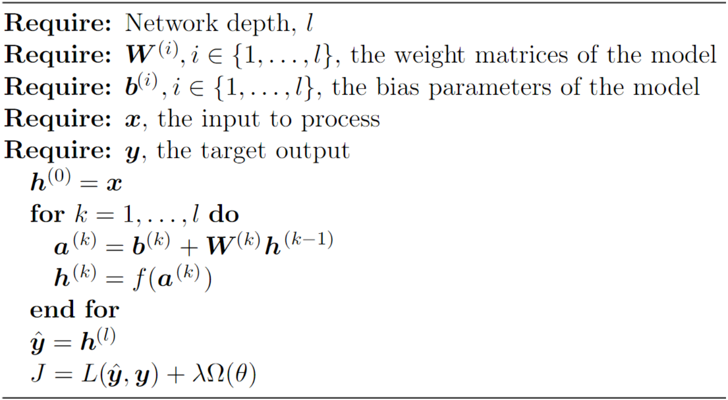
- How can we compute $\hat{y}$?
- How about $\frac{\partial \mathcal{L}(y,\hat{y})}{W_{q,r}^{(\ell)}}$ and $\frac{\partial \mathcal{L}(y,\hat{y})}{b_{q}^{(\ell)}}$, $\ell \in \{1,2,3\}, q \in \{1,2\}, r \in \{1,2\}$?
- Lets derive on the board ...

Motivating Back-propagation



- Apply one gradient descent update on $W_{2,1}^{(1)}$ with a learning rate 0.5.
- Lets see the reduction of loss on the board

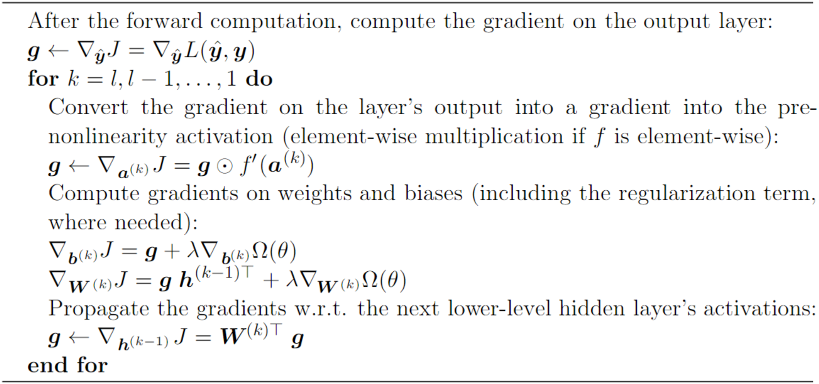
Forward Computations in MLP



```
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
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$
```

Figure 9 : Forward Computations for MLP, Source: Goodfellow et al., 2016

Back-propagation in MLP



```
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
```

Figure 10 : Back-propagation for MLP, Source: Goodfellow et al., 2016

Symbol-to-Symbol Derivatives

- A software engineering strategy for learning deep networks
- Add nodes in a computational graph to provide a symbolic description of the derivatives (Theano, Tensorflow)

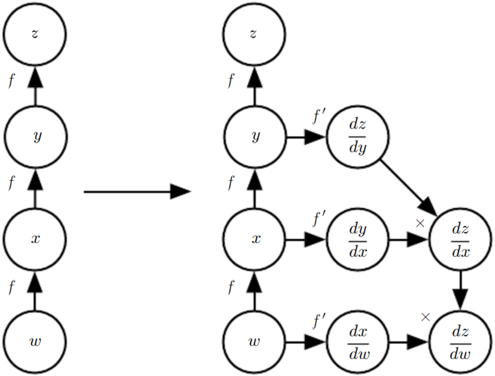


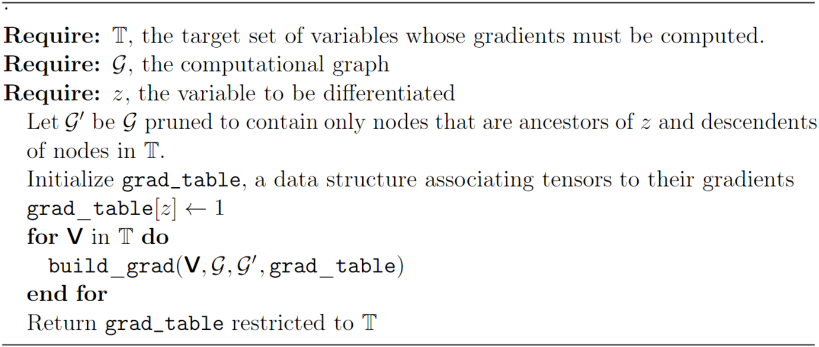
Figure 11 : Symbol-to-Symbol derivative, Source: Goodfellow et al., 2016

Implementing General Back-propagation

- Each variable $\mathbf{V}$ is associated with three subroutines:
    - `get_operation(`$\mathbf{V}$`)`: Get the operation that produced $\mathbf{V}$
    - `get_consumers(`$\mathbf{V}$, $\mathcal{G}$`)`: Get the children of $\mathbf{V}$ in graph $\mathcal{G}$
    - `get_inputs(`$\mathbf{V}$, $\mathcal{G}$`)`: Get the parents of $\mathbf{V}$ in graph $\mathcal{G}$

- Every operation `op` has a `bprop` operation:
    - $\texttt{op.bprob(inputs, } \mathbf{X}, \ \mathbf{G}) = \sum_i (\nabla_{\mathbf{X}} \texttt{op.f(inputs)}_i) \mathbf{G_i}$
    - where $\mathbf{G}$ is the gradient of the loss w.r.t. the output of the operation
    - where `inputs` are an abstraction for operation parameters
    - where $\mathbf{X}$ is the specific input for which we would like to compute the gradient of the loss w.r.t. it
    - where `op.f` is the function that this operation performs

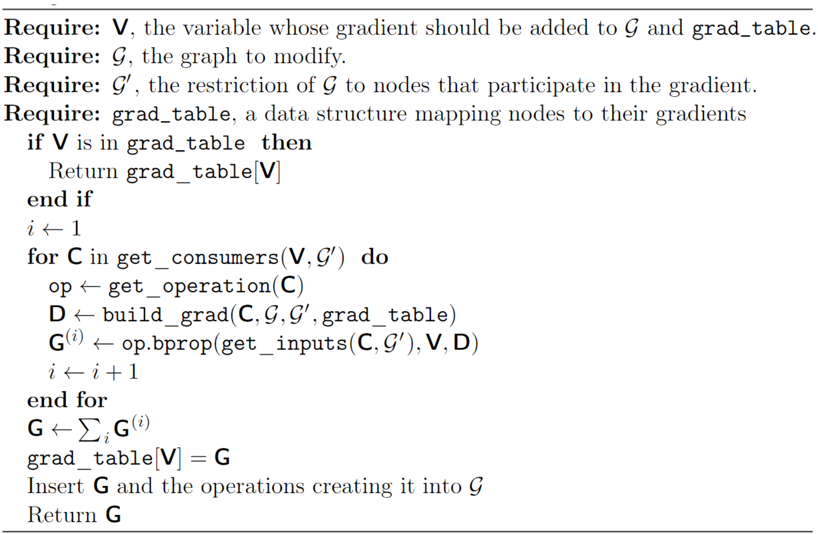
General Back-propagation - Start Method (I)



```
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 $\mathbb{T}$
```

Figure 12 : Interface to General Back-prop, Source: Goodfellow et al., 2016

General Back-propagation - Recursion (II)



```
Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and grad_table.
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 $\mathbf{V}$ is in grad_table then
    Return grad_table[$\mathbf{V}$]
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in get_consumers($\mathbf{V}, \mathcal{G}'$) do
    op $\leftarrow$ get_operation($\mathbf{C}$)
    $\mathbf{D} \leftarrow$ build_grad($\mathbf{C}, \mathcal{G}, \mathcal{G}'$, grad_table)
    $\mathbf{G}^{(i)} \leftarrow$ op.bprop(get_inputs($\mathbf{C}, \mathcal{G}'$), $\mathbf{V}, \mathbf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  grad_table[$\mathbf{V}$] = $\mathbf{G}$
  Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathbf{G}$
```

Figure 13 : Recursive General Back-prop, Source: Goodfellow et al., 2016