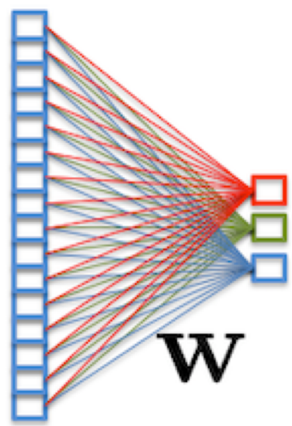
5IF - Deep Learning and Differentiable Programming

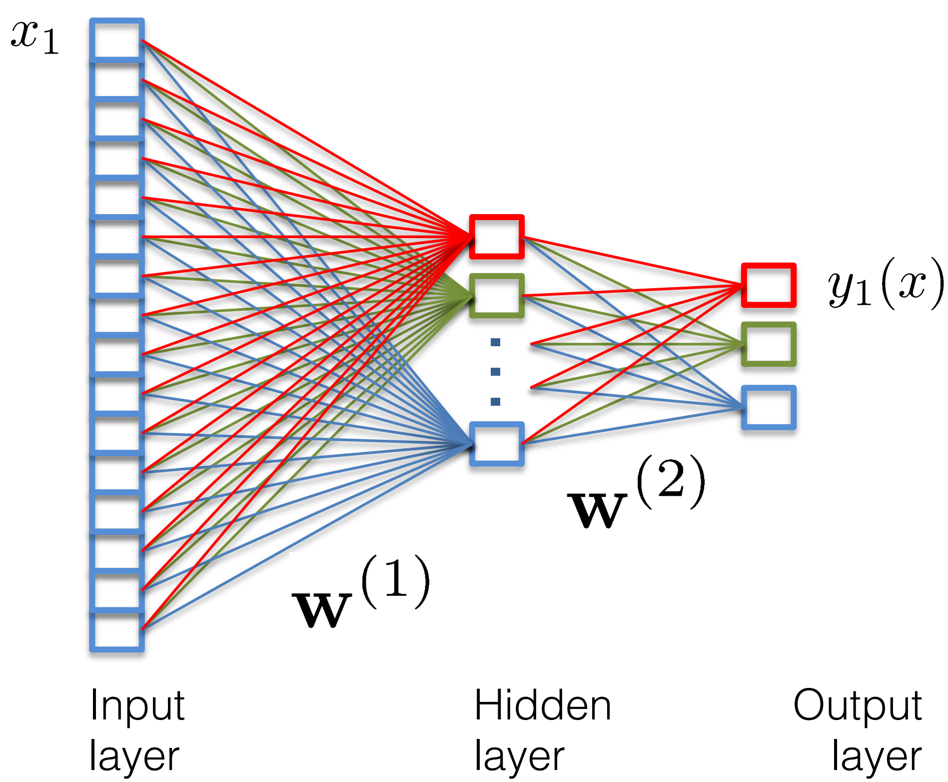
2.2 Gradient Back-propagation



INSA LYON Christian Wolf

Recall: Multi-layer Perceptron

$$y_k(x) = \sigma\left(\sum_j w_{kj}^{(2)} \zeta\left(\sum_i w_{ji}^{(1)} x_i\right)\right)$$



Gradient descent

One optimizer step:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}\left(h(x, \theta), y^*\right)$$

The gradient is a vector of partial derivatives:

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_0} \\ \\ \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \\ \vdots \\ \\ \frac{\partial \mathcal{L}}{\partial \theta_N} \end{bmatrix}$$

Solution with finite differences

Before the publication of the gradient backpropagation algorithm, gradients were calculated by finite differences:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \theta_0} &= \frac{\mathcal{L}(x,\theta)-\mathcal{L}(x,\theta+\Delta)}{\mathcal{L}(x,\theta)} + O(\Delta) \\ &\approx \frac{\mathcal{L}(x,\theta)-\mathcal{L}(x,\theta+\Delta)}{\mathcal{L}(x,\theta)}\end{aligned}$$

$\longrightarrow$ approximate solution, high complexity.

Calculate the exact gradients: linear nets

Let's consider a linear network

$$y_k = \sum_i w_{ki} x_i$$

and a sum of squared differences error function:

$$\mathcal{L}_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

where samples are indexed by n. The gradient w.r.t. sample n is:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = (y_{nj} - t_{nj})\, x_{ni}$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

Differentiate a multi-layer network (1)

We need to calculate the derivative of a function which is a composition of several other functions, e.g. linear functions and point-wise non-linearities.
A two layer network can be written in the following form (omitting parameters in the notation):

$$y = f_3(f_2(f_1(x)))$$

where

$$\begin{aligned} f_1(x) &= W^{(1)}x \\ f_2(x) &= \tanh(x) \\ f_3(x) &= W^{(2)}x \end{aligned}$$

It's all about chain rule of calculus...

Recall chain rule: given a function

$$y(x) = f(g(x))$$

or, in a slightly different notation, $y = f \circ g$,

$$y'(x) = f'(g(x))g'(x) \qquad \textit{(Lagrange's notation)}$$

or

$$(f \circ g)' = \big(f' \circ g\big) \cdot g'$$

or, if we name the intermediate variable $z = g(x)$

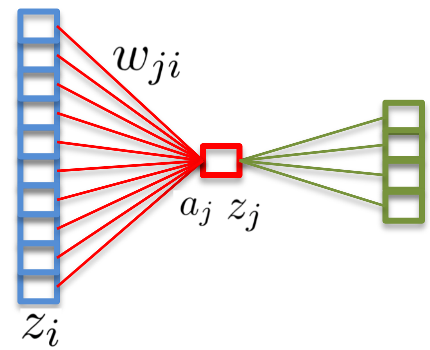
$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx} \qquad \textit{(Leibnitz's notation)}$$

All notations are equivalent.

Differentiate a multi-layer network (2)

Let's now consider a multi-layer network, in particular an arbitrary unit indexed by j and receiving inputs from units index by i, providing outputs z_i. It's activation a_j (before the non-linearity) and output z_j are:

$$a_j = \sum_i w_{ji} z_i, \quad z_j = h(a_j)$$



Its gradient is (using chain rule):

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

Differentiate a multi-layer network (3)

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

Since $a_j = \sum_i w_{ji} z_i$, we get

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

We write / define:

$$\delta_j \equiv \frac{\partial \mathcal{L}}{\partial a_j}$$

We obtain

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \delta_j z_i$$

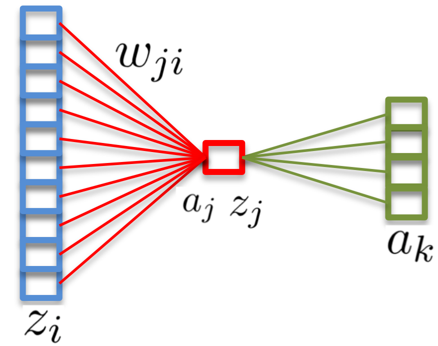
$\Rightarrow$ we need to calculate δ_j for each unit j.

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

Differentiate a multi-layer network (4)

If the output is linear (no activation) and we have a sum of squared differences loss, we get:

$$\delta_k = y_k - t_k$$



For the hidden units, we apply the chain rule:

$$\delta_j \equiv \frac{\partial \mathcal{L}}{\partial a_j} = \sum_k \frac{\partial \mathcal{L}}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

(a_k are units to which unit j **sends** information.)
Rewriting, and taking into account the activation function $h()$, we get:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

$\Rightarrow$ δ_j is calculated from the δ_k.
$\Rightarrow$ we traverse the network **backwards**!

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

The full backpropagation algorithm

1. **Forward pass**: stimulate the model with input x, calculate all a_j and z_j up to the output y.
2. Calculate the δ_j for the output units using the derivative of the loss function.
3. **Backward pass**: calculate all δ_j with

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

4. Calculate the gradients with

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \delta_j z_i$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

Remarks

- For batches, gradients are summed.
- The algorithm is general and can easily be adapted to other layers than fully-connected linear layers.
- The same algorithm can be used to calculate other gradients, e.g. deriving outputs with respect to inputs.
- Deep Learning frameworks calculate the gradients automatically given a definition of the foward pass (provided that the derivative of each sub function is available).

Backpropagation in PyTorch

Autograd

In PyTorch (and some other frameworks), `Autograd` performs automatic differentiation through a sequence of tensor instructions of an imperative language.

Let's consider a simple linear operation:

$$w = [\ 5\ 3\ ], \quad x = [\ 7\ 2\ ], \quad y = wx^T$$

The gradient of y w.r.t to x is given as

$$\nabla = \left[\frac{\partial y}{\partial x_i}\right] = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

The gradient of y w.r.t to w is given as

$$\nabla = \left[\frac{\partial y}{\partial w_i}\right] = \begin{bmatrix} 7 \\ 1 \end{bmatrix}$$

Autograd

In PyTorch, we will first create the tensors:

```
w = torch.tensor([5, 3], dtype=float, requires_grad=True)
x = torch.tensor([7, 1], dtype=float, requires_grad=True)
```

The `requires_grad` flag ensures that all calculations are tracked. We perform the linear operation:

```
y = torch.dot(w,x)
```

Since the tensor y has been calculated as result of operations on tracked tensors, it has a gradient function:

```
print (y)
```

```
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```

Autograd

We now run a backward pass on the variable y, which calculates gradients w.r.t. to all involved tensors:

```
y.backward()
```

The gradients are attached to each variable:

```
print (x.grad)
print (w.grad)
```

```
tensor([5., 3.], dtype=torch.float64)
tensor([7., 1.], dtype=torch.float64)
```

Autograd of a neural network

Recall our multi-layer network:

```
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = torch.nn.Linear(28*28, 300)
        self.fc2 = torch.nn.Linear(300, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Here, `torch.nn.Linear(A, B)` sets up a $A \times B$ weight matrix and a bias vector. All backward passes will calculate gradients with respect to these tensors:

```
model = MLP()
y = model(data)
loss = crossentropy(y, labels)
loss.backward()
```

Detaching tracking history

The tracking history uses memory in the tensor's space. If tracking is not used anymore for a tensor, it's tracking history can be detached:

```
print (y)
```

```
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```

```
z = y.detach()
print (z)
```

```
tensor(38., dtype=torch.float64)
```