motivation - gradient descent!

The Gradient:

Let $f: \mathbb{R}^n \to \mathbb{R}$, then the gradient of f is the vector of partial derivatives of f:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}, \quad x \in \mathbb{R}^n$$

Example: let $v, \overset{\text{const}}{v^{(0)}} \in \mathbb{R}^n$, $mse(v, v^{(0)}) = \sum_{i=1}^{n} \frac{(v_i - v_i^{(0)})^2}{n}$

$$\frac{\partial\, mse(v, v^0)}{\partial x_i} = \dots = \frac{2(v_i - v_i^{(0)})}{n} \equiv g_i$$

$$\Delta\, mse = \begin{pmatrix} g_1 \\ \vdots \\ g_n \end{pmatrix}$$

<multi-variable functions>

the Jacobian

<vector-valued function>

generalization of the derivative and the gradient to vector-valued functions.

assume $f: \mathbb{R}^n \to \mathbb{R}^m$, $x \in \mathbb{R}^n$, $J_f(x) \in \mathbb{R}^{m\times n}$ describe the derivative of every coordinate in $f(x)$ w.r.t every coordinate of x:

$$J_f(x)_{ij} = \left(\frac{\partial f}{\partial x}\right)_{i,j} = \frac{\partial f_i}{\partial x_j}$$

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \nabla f_1(x) \\ \vdots \\ \nabla f_m(x) \end{bmatrix}$$

for example:

$$f(x) = \text{Relu}(x) = \begin{cases} 0 & x_i \le 0 \\ x_i & \text{else} \end{cases} = \begin{matrix} H(x_1)\cdot x_1 \\ \vdots \\ H(x_n)\cdot x_n \end{matrix}$$

where H is the heaviside step-function.

then $$J_{\text{Relu}}(x) = \begin{bmatrix} H(x_1) & & 0 \\ & \ddots & \\ 0 & & H(x_n) \end{bmatrix}$$

$$f(x) = W\cdot x, \quad W \in \mathbb{R}^{m\times n} \quad (f: \mathbb{R}^n \to \mathbb{R}^m)$$

$$f_i(x) = \sum_{k=1}^{n} W_{ik} x_k \to \nabla f_i(x) = W_{i1} \dots W_{in}$$

$$\frac{\partial f}{\partial x} = W$$

$$\frac{\partial f}{\partial W} \in \mathbb{R}^{m\times m\times n}, \quad \frac{\partial f}{\partial W}_{0,:,:} = \begin{bmatrix} x_1 & \dots & x_n \\ 0 & 0 \dots & 0 \\ & \ddots & \\ & & 0 \end{bmatrix}$$

Back Propagation

chain rule:

Let $f: \mathbb{R} \to \mathbb{R}$, $g: \mathbb{R} \to \mathbb{R}$,

$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f(\cdot)}{\partial g} \cdot \frac{\partial g(\cdot)}{\partial x}$$

<u>multi-var:</u>

Let $f: \mathbb{R}^n \to \mathbb{R}$,

$$\frac{\partial f(u_1 \dots u_n)}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial u_i} \cdot \frac{\partial u_i}{\partial x}$$

<u>vector:</u>

Let $g: \mathbb{R}^n \to \mathbb{R}^k$, $f: \mathbb{R}^k \to \mathbb{R}^m$

$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$, order does matters!

Backprop -

computational graph:

we can represent composition of functions as Directed Acyclic Graph.

$$\begin{tikzcd} \boxed{w} \arrow[r] & A \arrow[r] & B \arrow[r] & C \arrow[r] & \boxed{L} \\ \boxed{x} \arrow[ur] & \boxed{b} \arrow[ur] & & & \end{tikzcd}$$

$\sigma(wx+b) \rightarrow$

each Node represent operation and it's result.

Loops does not create cycles!
Loops are unrolled!

reminder - motivation: calculate $\frac{\partial loss}{\partial var}$

we want: $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial C} \cdot \frac{\partial C}{\partial B} \cdot \frac{\partial B}{\partial b}$

if multiple paths from b to L?
sum over the paths
as we saw in the multivar chain rule!

general formula:

Let n_i, n_j 2 nodes in the computational graph.

$$\frac{\partial n_i}{\partial n_j} = \sum_{P \in PATHS(n_j \to n_i)} \prod_{t=1}^{T_P} \frac{\partial p_t}{\partial p_{t-1}}$$

number of paths can be exponential in the depth of the graph.

Here come the Back-Propagation - simple and efficient way to do this in $O(n)$.
number of Nodes

Gradients as sums over Paths

1 Autoencoders

An autoencoder is a neural network architecture for unsupervised learning, where we wish to learn a lower dimensional representation of our data. There are many reasons why we would want to do dimensionality reduction, and we will go deeper into this concept in the manifold learning lectures.

1.1 Linear Dimensionality Reduction

A classic method for dimensionality reduction is PCA, which tries to learn encoding and decoding functions that take the input from a n-dimensional space to a d-dimensional one and back ($d < n$), such that the reconstruction error is minimized. PCA constrains the functions to be linear and optimizes the following loss function:

$$\underset{U\in\mathbb{R}^{n\times d}, V\in\mathbb{R}^{d\times n}}{argmin} \Big(\sum_{x\in S} ||UVx - x||^2\Big)$$

PCA has an analytical solution which consists of diagonalizing the centered covariance matrix of our data (this is actually what we did when calculating the A matrix of the ICA model in the first exercise):

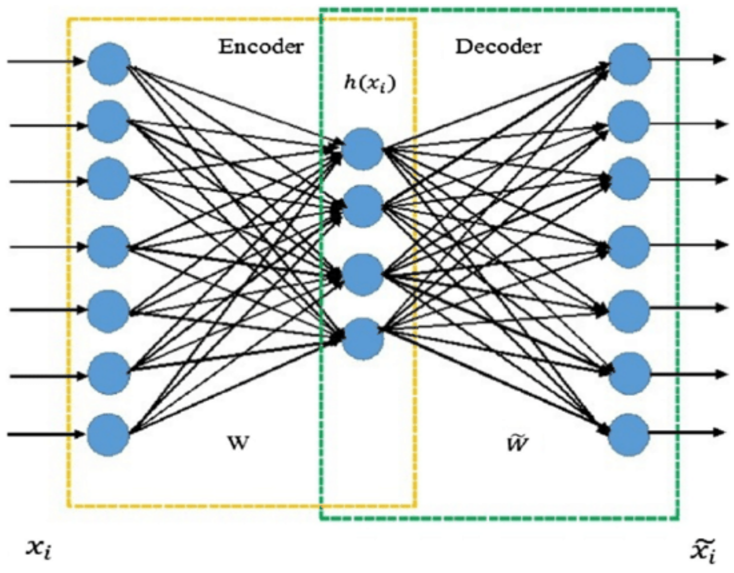
$$\Sigma = \frac{1}{n}\sum_{x\in S}(x-\mu)(x-\mu)^T = P\Lambda P^T$$

We then take the first d orthogonal vectors of P, corresponding to the top d eigenvalues in Λ. The solution is then $V = P_d$ and $U = P_d^T$.

PCA works OK, but we might get better results if we don't constrain the encoding and decoding functions to being linear. What if we replaced these functions with neural networks?

1.2 Autoencoders

An autoencoder swaps U and V for two neural networks which are connected via their "embedding layer", which is the code created by the **encoder** network (which the **decoder** network decodes). We can view PCA as being a special case of an autoencoder, only with a single fully connected layer for every network and no activation function:



But now, we can replace the single linear fully connected layer with several layers with non linear activation functions and get a much more expressive encoder and decoder. If we call the encoder e and the decoder d, we get the following loss function:

$$\underset{e,d}{argmin}\Big(\sum_x ||d(e(x)) - x||^2\Big)$$

This loss function can now be optimized with back-propagation as usual. Once the two networks have converged, we reduce the dimensionality of the data using our trained encoder.

1.3 Regularized Autoencoders

Autoencoders as described above perform better than PCA, but they can be improved greatly by adding different forms of regularization (either explicitly or implicitly). This can either be done using methods we will see in the second lecture (dropout, weight decay...), or done in ways which are specific to autoencoders.

1.3.1 Contractive Autoencoders (CAE)

Contractive autoencoders assume that the embedding layer should be smooth with respect to the inputs to the network. This is reasonable - we don't want the representation to change too much when the inputs to the network change a little. We can write this mathematically as wanting the gradient of the encoding with respect to the input to be small. This leads to the following loss function, which incorporates our regularization term:

$$\underset{e,d}{argmin}\Big(\sum_x ||d(e(x)) - x||^2 + \lambda||\frac{\partial e}{\partial x}||_F^2\Big)$$

1.3.2 Denoising Autoencoders (DAE)

Denoising autoencoders are used both to get a good representation of the input and as a denoising model. The assumption here is that we would like to learn a lower dimensional representation which is able to ignore corruptions in the input and represent the meaningful parts. This means that an input with different corruptions will be decoded as the same original input.

To train a denoising autoencoder, we need to define the type of corruptions it should handle (missing pixel values / additive noise / convolutional blurring and so on), and create a function that stochastically corrupts an input, $C(x) = \hat{x}$. Given this corruption function, we would like the autoencoder to be able to recover the true x, even after the random corruption. This leads to the following loss function:

$$\underset{e,d}{argmin}\Big(\sum_x \mathbb{E}_{\hat{x}\sim C(x)}\big[||d(e(\hat{x})) - x||^2\big]\Big)$$