Machine Learning Introduction

Inverse Problems:

$\hat{x} = f_\theta(y)$ (inference function)

x (unknown) (to be inferred) → THE WORLD → y (data) (measure-able) → INVERSION / INFERENCE → $\hat{x}$ (estimate)

θ (parameters)

*All sensor problems are inverse problems

$$\hat{x} = \boxed{B}\ \sigma \left\{ \boxed{A}\ \boxed{y} + \boxed{b} \right\}$$

(function applied) col vector, offset vector

Supervised Machine Learning:

Training data → x_k, y_k (data) (x_k, y_k) pairs → ML Training done via minimizing the loss → θ^*

Single Layer Dense NN:

y → Ay → + (b) → $\sigma(\cdot)$ → B → $\hat{x}$ ⇔

$$\hat{x} = B\sigma(Ay+b)$$

$\begin{cases} A \in \mathbb{R}^{N_1 \times N_y} \\ b \in \mathbb{R}^{N_1} \\ \sigma : \mathbb{R}^{N_1} \to \mathbb{R}^{N_1} \\ B \in \mathbb{R}^{N_x \times N_1} \end{cases}$

- matrix of multiplicative weights
- column vector of additive offsets
- point-wise activation function
- matrix of multiplicative weights

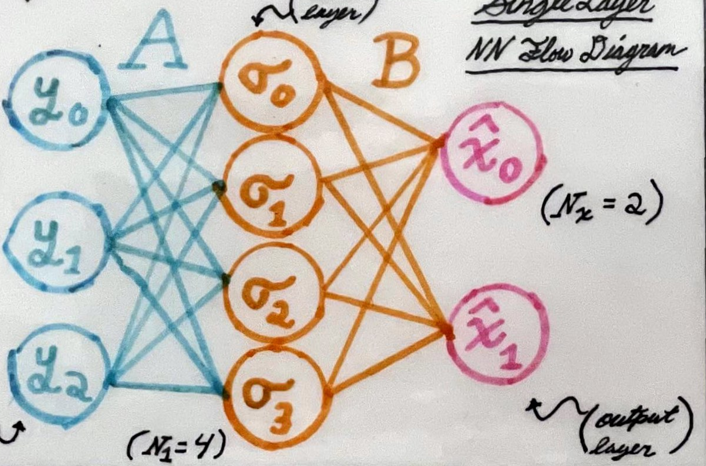
Abstraction:

$$\hat{x} = f_\theta(y) = B\sigma(Ay+b)$$

y → Ay → + (b) → $\sigma(\cdot)$ → B → $\hat{x}$

$\theta = (A, B, b)$ ("the set of all NN params.")

Single Layer NN Flow Diagram



Gradient matrix of Softmax Function:

*Dense matrix — mostly non-zero values

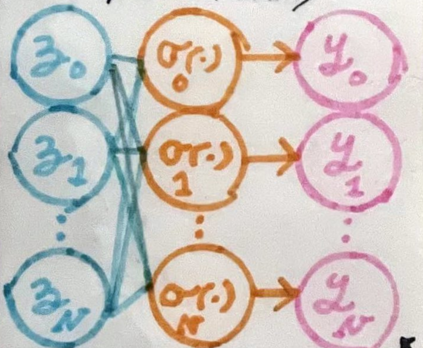
$$[\nabla\sigma(z)]_{i,j} = \frac{1}{\sum_k e^{z_k}}\left(e^{z_i}\delta_{i,j} - \frac{e^{z_i}e^{z_j}}{\sum_k e^{z_k}}\right)$$

*Slow 2 compute

$e^{z_i}\delta_{i,j} = \begin{bmatrix} e^{z_0} & & 0 \\ & \ddots & \\ 0 & & e^{z_N} \end{bmatrix}$ $\quad e^{z_i}e^{z_j} = \begin{bmatrix} e^{z_0} \\ \vdots \\ e^{z_N} \end{bmatrix}\begin{bmatrix} e^{z_0} & \cdots & e^{z_N} \end{bmatrix}$

$$\sigma_i(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$ ← (SOFTMAX ACT. FN)

*Joint act. fn. can be interpreted as a probability density

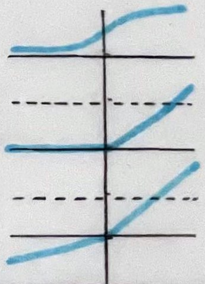


Point-Wise Activation Functions:

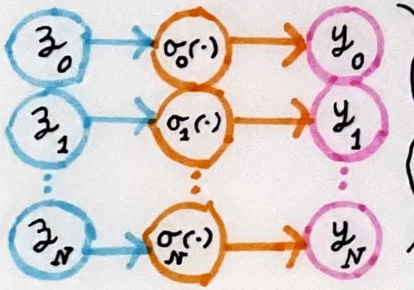
Logistic Sigmoid fn. ← $\sigma_i(z) = \frac{1}{1+e^{-z_i}}$

Rectified Linear Unit (RELU) ← $\sigma_i(z) = \begin{cases} 0, & z_i \le 0 \\ z_i, & z_i > 0 \end{cases}$

Leaky ReLU ← $\sigma_i(z) = \begin{cases} \alpha z_i, & z_i \le 0 \\ z_i, & z_i > 0 \end{cases}$



$\sigma : \mathbb{R}^N \to \mathbb{R}^N$ ← (point-wise act. fn)



z → $\boxed{\sigma(\cdot)}$ → y

("is a point-wise activation function")

Gradient Matrix of a Point-Wise Act. fn is:
- diagonal
- sparse (many zeros)
- fast 2 compute

$$\nabla\sigma(z) = \underbrace{\left[\frac{\partial \sigma_i(z)}{\partial z_j}\right]}_{(N \times N)} = \begin{bmatrix} d_0 & & 0 \\ & d_1 \ddots & \\ 0 & & d_N \end{bmatrix}$$

$$\left(\text{where } d_i = \frac{\partial \sigma(z_i)}{\partial z_i}\right)$$

Standard Encoding ← $\hat{x} \in \{0, \dots, M-1\}$ (each val repr. a class)

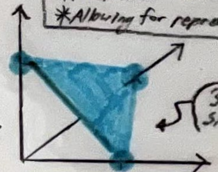
1-Hot Encoding (for classification) ← $\hat{x} \in \mathbb{R}^M$ s.t. $\hat{x}_i = \begin{cases} 1, & \text{class} == i \\ 0, & \text{class} != i \end{cases}$ ⇒

M-dimensional Simplex:

$$S^M = \left\{ x \in \mathbb{R}^M \,\middle|\, \forall i, x_i \ge 0 \ \& \ 1 = \sum_{i=0}^{M-1} x_i \right\}$$

$\hat{x} \in S^M \subset \mathbb{R}^M$ (like a probability density per class)

*Continuous function on a convex set
*Allowing for representation of prob. densities



(3D Simplex)

*Good for optimization

Tensors

(from "differential geometry", used by Einstein when formulating the theory of General Relativity)

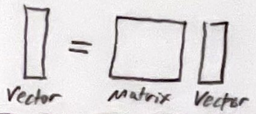
(Contravariant ⇔ Data) col vectors

(Covariant ⇔ Operator) row vectors

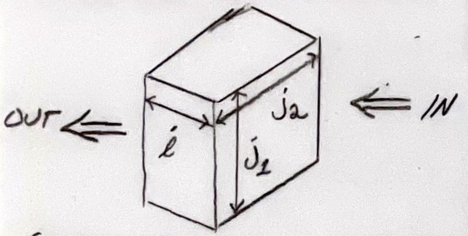
*Note: differentiating a 1D obj w.r.t. a 2D obj yeilds a ⇒ 3D object

Def.: Tensor

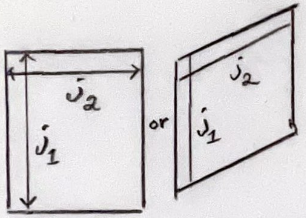
– The generalization of a
- matrix operator, for dim > 2
- vector data to > 1 dim



MATRICES ARE OPERATORS

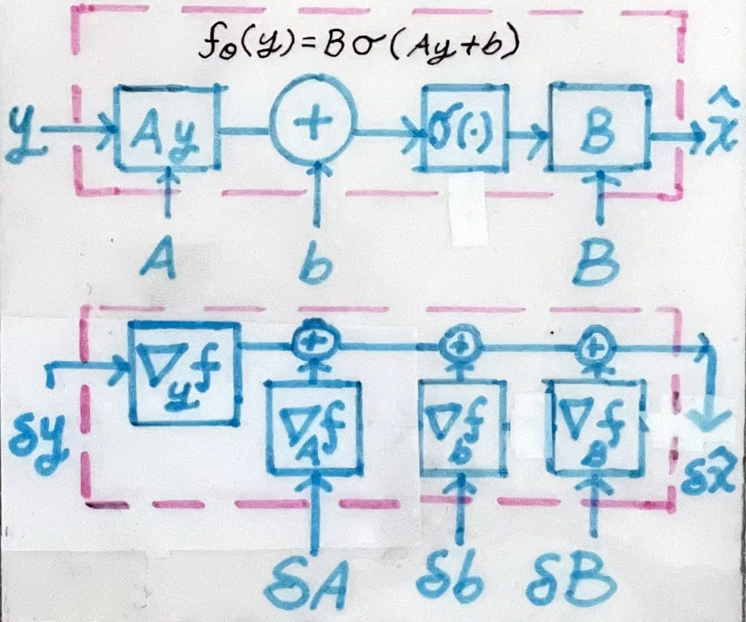


Example of a tensor generalizing a matrix operator for > 2 dims



Example of a tensor generalizing a vector to a higher dimensional space / representation

$f_\theta(y) = B\sigma(Ay+b)$



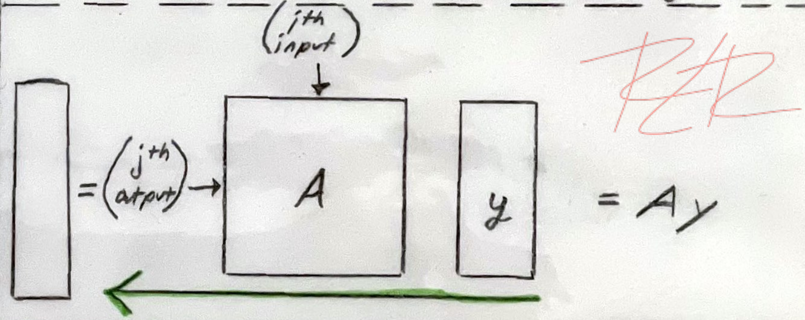
Parameters: $\theta = (A, b, B)$

Gradients
- $\nabla_\theta f_\theta(y)$
- (at the input) $\nabla_y f_\theta(y)$ or $\nabla_y f$

Partial Derivative of Ay_j w.r.t. y_i

$$\frac{\partial [Ay]_j}{\partial y_i} \xrightarrow{\text{(only depends on)}} \frac{\partial (A_{ji} y_i)}{\partial y_i} = A_j^{\ i}$$

(i.e., $[\nabla_y (Ay)]_j^{\ i} = A_j^{\ i}$)



Delta Function:

$$\delta_j^i = \begin{cases} 1, & i=j \\ 0, & i \neq j \end{cases}$$

⇓

$G_j^k = G_i^k \delta_j^i$ (sum over the i's) ⇒

(Gradient w.r.t. Matrix) $[\nabla_A (Ay)]^i_{j_1, j_2} = \delta^i_{j_1} y^{j_2}$

tensor notation $[\nabla y(Ay)]_j^{\ i} = A_j^{\ i}$

(Gradient w.r.t. Vector) $\nabla y (Ay) = A$

Contravariant vectors: $x = gy$

*Column vectors representing data / describe the position of something

* y^j for $0 \leq j < N$ & $x \in \mathbb{R}$

Covariant vector:

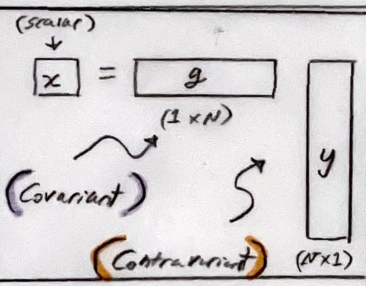
* Row vector (gradient vec) that operates on data

* g_j for $0 \leq j < N$

Einstein Notation:

$$x = g_j y^j = \sum_{j=0}^{N-1} g_j y^j$$

- leave out Summation
- Always sum over any 2 idxs that occur twice



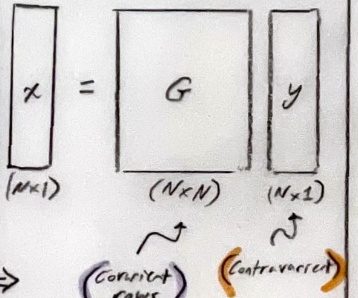
$x = Gy$

***1D Contravariant Vector**
- y^j for $0 \leq j \leq N_y$
- x^j for $0 \leq j \leq N_x$

***2D Tensor (i.e., matrix)**
- G_j^i for $0 \leq i \leq N_x$ & $0 \leq j \leq N_y$

⇔

Vector-Matrix Products:



$$x^i = G_j^i y^j = \sum_{j=0}^{N_y - 1} G_j^i y^j$$

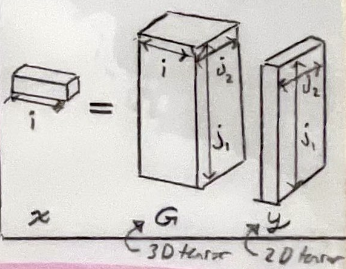
Tensor Products: (sum over)

$$x^{i_1, i_2} = G^{i_1, i_2}_{\ \ j_1, j_2}\, y^{j_1, j_2}$$

2D Contravariant Vectors:
- y^{j_1, j_2} for $0 \leq j_1, j_2 < N_y$
- x^{i_1, i_2} for $0 \leq i_1, i_2 < N_x$

4D Tensor:
- $G^{i_1, i_2}_{\ \ j_1, j_2}$ for $0 \leq i_1, i_2 < N_x$ & for $0 \leq j_1, j_2 < N_y$

Example: $x^i = G^i_{j_1, j_2} y^{j_1, j_2}$



G is a tensor with 2D covariant input & 2D contravariant output

$G \in \mathbb{R}^{N_o \times N_i}$ ← General Linear Tra

Gradient Descent Optimization (GD)

GD Algorithm:

$d = -\nabla \mathcal{L}(\theta)$ ← $1 \times N_x$ (row vector)

repeat until converged {
  $d \leftarrow -\nabla \mathcal{L}(\theta)$
  $\theta \leftarrow \theta + \alpha d^t$
}

if α is too { small → SLOW; large → UNSTABLE

Steepest Descent: (EXPENSIVE)

* compute best α via line search

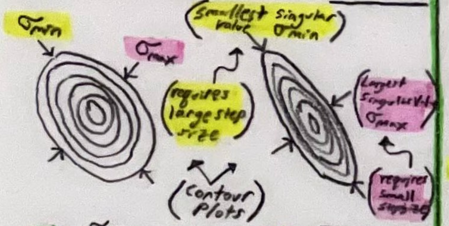
repeat until converged {
  $d \leftarrow -\nabla L(\theta)$
  $\alpha^* \leftarrow \arg\min_{\alpha} \{L(\theta + \alpha d^t)\}$
  $\theta \leftarrow \theta + \alpha^* d^t$
}

Coordinate Descent:

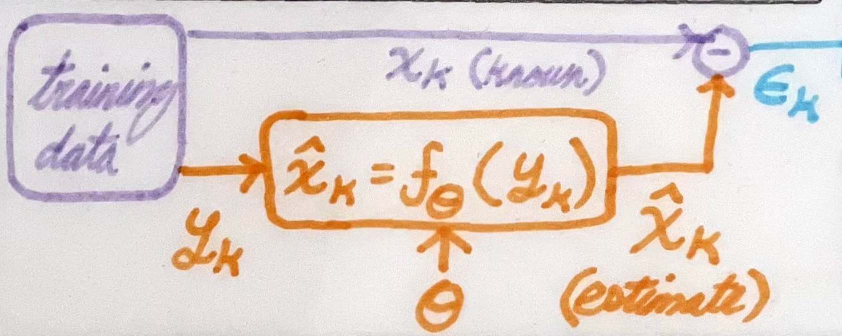
* Update 1 param. at a time
* Fast but requires many updates

Slow Convergence of Gradient Descent

* Sensitive to condition number (CN) of the problem, no perfect step size choice
* Solution → Newton's Method, correct for local 2nd derivative ... "sphere the ellipse" (too computational / difficult)
* Alternative methods such as momentum



$CN = \frac{\sigma_{max}}{\sigma_{min}} = 2$ $CN = \frac{\sigma_{max}}{\sigma_{min}} = 10$



* Compute gradient via chain rule
* Compute Adjoint gradient
* Back Propagate

train via minimizing loss:

$$\theta^* = \arg\min_{\theta} \{\mathcal{L}(\theta)\}$$

$d = -\nabla \mathcal{L}(\theta)$ $(1 \times N_x)$ (gradient is a row vector)

⇒ Gradient Descent Convergence
do while {
  $d \leftarrow -\nabla L(\theta)$
  $\theta \leftarrow \theta + \alpha d^t$ (transpose)
} converges

Loss Gradient

* Note that the "Adjoint matrix" / conjugate transpose / Hermitian transpose is just the transpose for real matrices → A^H or $A^* = A^T$

* The Norm of a vector is the square root of the sum of the square of its elements (euclidean distance)

$$\nabla_\theta \mathcal{L}_{MSE}(\theta) = \nabla_\theta \left\{ \frac{1}{K} \sum_{k=0}^{K-1} \| x_k - f_\theta(y_k) \|^2 \right\} = \frac{1}{K} \sum_{k=0}^{K-1} \nabla_\theta \{ \| x_k - f_\theta(y_k) \|^2 \}$$

$$= \frac{2}{K} \sum_{k=0}^{K-1} (x_k - f_\theta(y_k))^t \nabla_\theta (x_k - f_\theta(y_k)) = \frac{-2}{K} \sum_{k=0}^{K-1} (x_k - f_\theta(y_k))^t \nabla_\theta f_\theta(y_k)$$

$$\therefore \quad -\nabla_\theta L_{MSE}(\theta) = \underbrace{\boxed{\frac{2}{K} \sum_{k=0}^{K-1}}}_{\text{(Summation over training data)}} \underbrace{\boxed{(x_k - f_\theta(y_k))^t}}_{\text{(Prediction Error)}} \underbrace{\boxed{\nabla_\theta f_\theta(y_k)}}_{\text{(Gradient of Function)}}$$

(Gradient of inference function is enabled by AUTOMATIC DIFFERENTIATION) (i.e., Back-propagation for NN)

Error Vector $(1 \times N_x)$ $= \epsilon_k^t = (x_k - f_\theta(y_k))^t$

$(\because x_k = f_\theta(y_k) + \text{error})$

Param. Vector $(P \times 1)$ $= [-\nabla_\theta L_{MSE}]^t =$ (dimensionality of param. vector)

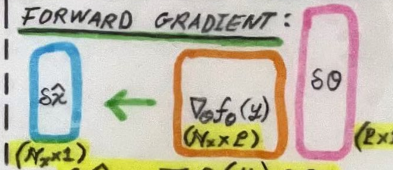
j (param. dim), i (output dim)

Inference Function Gradient

$$\frac{\partial [f_\theta(y_k)]_i}{\partial \theta_j} \ (N_x \times P) = \nabla_\theta f_\theta(y_k)$$

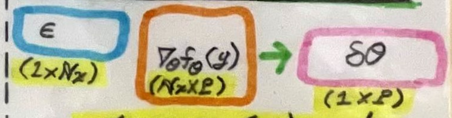
(gradient of function)

FORWARD GRADIENT:



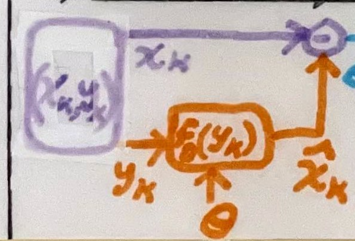
$\delta\hat{x} \leftarrow \nabla_\theta f_\theta(y)\, \delta\theta$

BACKWARD (ADJOINT) GRADIENT:



$\epsilon^t \nabla_\theta f_\theta(y) \rightarrow d$

⇒ Update Direction for Supervised Training:



$$d = \frac{2}{K} \sum_{k=0}^{K-1} \epsilon_k^t \, \nabla_\theta f_\theta(\cdot)$$

(directional grad. vector)

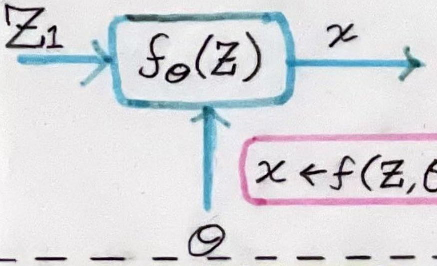
$(1 \times P) = (1 \times N_x)(N_x \times P)$

Backprop. 4 CNNs

Adj. Grad. 4 Convolution | Adjoint Gradient | Gradient of Convolution (output w.r.t. input)

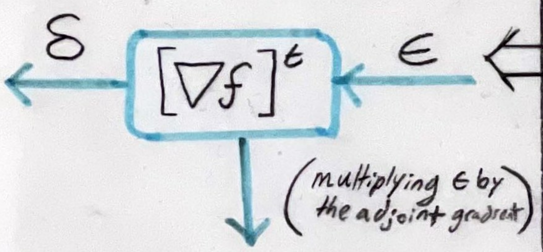
* 2 functions required per Node

1) forward propagation:



$x \leftarrow f(z,\theta)$

2) adjoint gradient



$g = (g_\omega, g_b)$

$[\delta, g_\omega, g_b] \leftarrow G(\epsilon, z, \theta)$

Adj. Grad. 4 Convolution

* Gradient of output w.r.t. weights

$x_i = z_i * w_i = \sum_j z_{i-j} w_j \Leftrightarrow x = Aw$

$\frac{\partial x_i}{\partial w_j} = A_{i,j} = z_{i-j}$

$[A^t]_{i,j} = A_{j,i} = z_{j-i}$

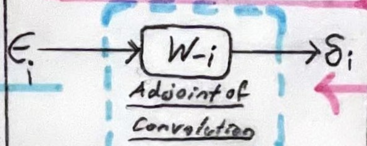
$\delta_i = \sum_j z_{j-i}\, \epsilon_j$

(Auto correlating ϵ_j w/ time reverse of y_j)

Adjoint Gradient

$[A^t]_{i,j} = A_{j,i} = W_{j-i}$

$\delta_i = \sum_j W_{j-i}\, \epsilon_j = W_{-i} * \epsilon_i$

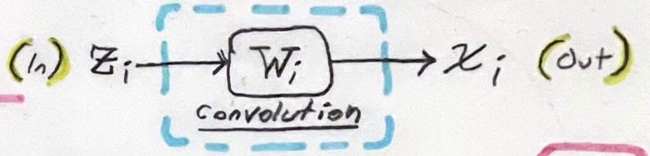


* Adjoint uses the time-reversed impulse response

(TRANSPOSE OF A)

Gradient of Convolution (output w.r.t. input)

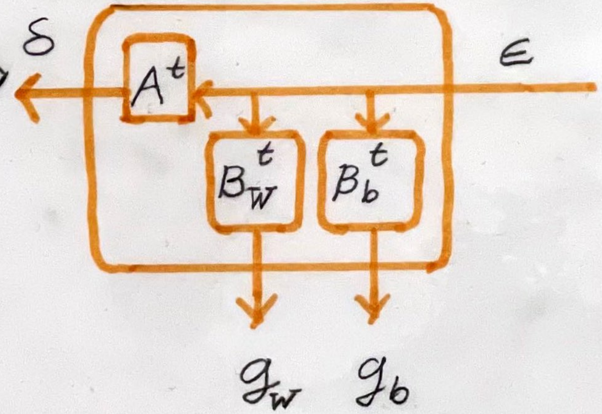
* fwd prop., then find gradient of the output w.r.t. the input } A



$x_i = w_i * z_i = \sum_j w_{i-j} z_j \Leftrightarrow x = Az$

$\frac{\partial x_i}{\partial z_j} = A_{i,j} = W_{i-j}$ (TOEPLITZ MATRIX)

$f(y) = \sigma\left(W_{(j_1,j_2),i}^{\;j_3} * z_i^{(j_1,j_2),i} + b^{j_3}\right)$



$A = \nabla_z f_\theta(z) \leftarrow$ Gradient w.r.t. input, z

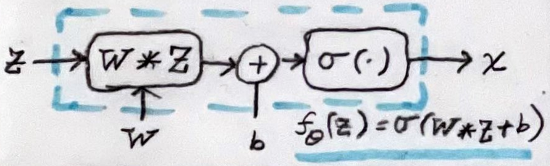
$B_W = \nabla_W f_\theta(z) \leftarrow$ Gradient w.r.t. filter weights, W

$B_b = \nabla_b f_\theta(z) \leftarrow$ Gradient w.r.t. offsets, b

* Convolutions are commutative i.e., $w_i * z_i \Leftrightarrow z_i * w_i$

* REMINDER → SW Implementation of the convolution is really a correlation operation

Single Layer CNN Example:



$\nabla_\theta f_\theta(z) = \left[\nabla_\omega f_{(\omega,b)}(z), \nabla_b f_{(\omega,b)}(z)\right]$

Adj. Grad. w.r.t. $\theta = (\omega, b)$, & then get $\nabla_z f_\theta(z)$ (gradient w.r.t. input)

(Adjoint Gradient) (Adjoint Function Gradient) (Error Vector) Backpropagating the error, ϵ

$$\underbrace{\delta}_{(N_y \times 1)} = \underbrace{A_{j,i} = \frac{\partial [f_\theta(y_x)]_j}{\partial z_i}}_{(N_y \times N_x)} \underbrace{\epsilon}_{(N_x \times 1)}$$

(Huge Matrix)

(Input gradients)

FAST ADJOINT GRADIENT APPROACH

⇒ Directly compute the outputs w/o constructing the gradient matrix prior & requiring both memory & computational resources ("fast" ∵ A is never computed!)