Analysis of SVD Deep Neural Network parametrization

“Da kto takoi etot et al?!” team

Patakin Nikolay,
Yermekova Assel,
Shushkova Varvara,
Ildar Gabdrakhmanov,
Pudyakov Yaroslav

Theoretical Part

Problem

Time consuming operations!

Neural Networks →

1) Matrix inversion
2) Matrix Determinant
3) Spectral normalizations

Theoretical Part

Problem

Time consuming operations

Neural Networks →

1) Matrix inversion
2) Matrix Determinant
3) Spectral normalizations

Solution →

SVD decomposition

$$W = U\Sigma V^T$$

Σ - diagonal
U, V - orthogonal

What is the problem? →

Gradient Descent Update of Weight Matrix

$$\Sigma' = \Sigma - \eta\nabla_\Sigma$$

$$U' = U - \eta\nabla_U$$

$$V' = V - \eta\nabla_V$$

Theoretical Part

Problem

Time consuming operations

Neural Networks → 1) Matrix inversion
2) Matrix Determinant
3) Spectral normalizations

$\xrightarrow{\text{Solution}}$ SVD decomposition

$$W = U\Sigma V^T$$

Σ - diagonal
U, V - orthogonal

Theoretical Part

Problem

Time consuming operations

SVD decomposition

Neural Networks $\longrightarrow$

1) Matrix inversion
2) Matrix Determinant
3) Spectral normalizations

Solution $\longrightarrow$

$$W = U\Sigma V^T$$

Σ - diagonal

U, V - orthogonal

What is the problem? $\longrightarrow$ Gradient Descent Update of Weight Matrix

$$\Sigma' = \Sigma - \eta \nabla_{\Sigma}$$

$$U' = U - \eta \nabla_{U}$$

$$V' = V - \eta \nabla_{V}$$

Loss of orthogonality of U, V matrices

Theoretical Part

To preserve U, V orthogonality:

$$U \in \mathbb{R}^{d\times d}$$

Householder matrices

$$U = \prod_{i=1}^{d} H_i \qquad H_i = I - 2\frac{v_i v_i^T}{||v_i||_2^2} \qquad v_i \in \mathbb{R}^d.$$

All products of Householder matrices are orthogonal → Any dxd orthogonal matrix can be decomposed as a product of *d* Householder matrices → Allows to perform gradient descent over orthogonal matrices

Why Householder matrices are good?

- U **remains orthogonal** under gradient descent update at *i* step
- It allows to perform gradient descent to **preserve the SVD** of W during gradient descent updates

What is the FastH algorithm?

Need to calculate:

$$UX = H_1 \cdots (H_{d-1}(H_d \cdot X))$$
$$X \in \mathbb{R}^{d \times m}$$

- Sequential

$$O(d^2 m)$$

$O(d)$ sequential vector-vector operations

What is the FastH algorithm?

Need to calculate:

$$UX = H_1 \cdots (H_{d-1}(H_d \cdot X))$$
$$X \in \mathbb{R}^{d \times m}$$

- Sequential

$$O(d^2m)$$

$O(d)$ sequential vector-vector operations

- FastH

$$\boxed{O(d^2m)}$$

$O(\frac{d}{m} + m)$ sequential matrix operations

FastH. Forward Pass

Input: $X \in \mathbb{R}^{d \times m}, \ d > m > 1$

$H_1, \ldots, H_d \ , \ H_i \in \mathbb{R}^{d \times d}$ - Householder matrices

Want to compute: $A = H_1 \cdots H_d X$

$$H_i = I - \frac{2 v_i v_i^T}{\|v_i\|_2^2} \ , \qquad P_i = \underbrace{H_{(i-1)\cdot m+1} \cdots H_{i \cdot m}}_{m \text{ matrices}} \quad i = 1, \ldots, \frac{d}{m}$$

Then: $A = \underbrace{H_1 \cdots H_d}_{d \text{ multiplications}} X = \underbrace{P_1 \cdots P_{\frac{d}{m}}}_{\frac{d}{m} \text{ multiplications}} X$

$P_i X$ takes $O(d^2 m)$
Then compute A
takes $O(d^3)$

Still slow.... :c

Using decomposition of product ~ $O(dm^2)$

$H_1 \cdots H_m = I - 2WY^T$

helps us reduce complexity of computation to $O(d^2 m)$
and decrease number of matrix multiplication to $O(\frac{d}{m} + m)$

FastH. Backwards Propagation

Input: $A_1, \ldots, A_{\frac{d}{m}+1}, P_1, \ldots, P_{\frac{d}{m}+1}, \dfrac{\partial L}{\partial A_1}$ L - loss function

Want to compute: $\dfrac{\partial L}{\partial X}, \dfrac{\partial L}{\partial v_1}, \ldots, \dfrac{\partial L}{\partial v_d}$

The backward pass of FastH has two steps:

1. Compute $\dfrac{\partial L}{\partial A_2}, \dfrac{\partial L}{\partial A_3}, \ldots, \dfrac{\partial L}{\partial A_{\frac{d}{m}+1}}$ by $\dfrac{\partial L}{\partial A_{i+1}} = \left[\dfrac{\partial A_i}{\partial A_{i+1}}\right]^T \dfrac{\partial L}{\partial A_i} = P_i^T \dfrac{\partial L}{\partial A_i}$ $\longrightarrow$ gradient wrt. X since $X = A_{\frac{d}{m}+1}$

2. Compute $\dfrac{\partial L}{\partial v_j}$ for all j $\rightarrow$ can be split into d/m subproblems $\rightarrow$ can be solved in parallel $\rightarrow$ one subproblem for each $\dfrac{\partial L}{\partial A_i}$

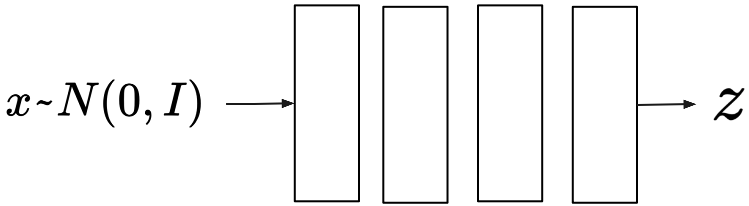
Applications

- Normalization flow models
  - Matrix determinant
  - Inverse matrix
- Spectral normalization

Normalization flow

What do we want?

simple distributions $\xrightarrow{\text{map}}$ complex distributions



How?

By a sequence of invertible and differentiable mappings

Change of variables

$$p_X(\mathbf{x}) = p_Z\left(f^{-1}(\mathbf{x})\right)\left|\det\left(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}}\right)\right|$$

For invertible matrix:

$$p_X(\mathbf{x}) = p_Z(\mathbf{z})\left|\det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right)\right|^{-1}$$

1. $f(x)$ has to be invertible
2. want fast calculation of determinant of Jacobian matrix

$$det\left(\frac{\partial f^{-1}(x)}{\partial x}\right)$$

Flow-based Generative Models

x - a high-dimensional random vector, collect a dataset D

generative process

$z \sim p_\theta(z)$

$x = g_\theta(z)$

f is composed of a sequence of transformations

$f = f_1 \circ f_2 \circ \ldots \circ f_K$

$$\mathbf{x} \overset{\mathbf{f}_1}{\longleftrightarrow} \mathbf{h}_1 \overset{\mathbf{f}_2}{\longleftrightarrow} \mathbf{h}_2 \cdots \overset{\mathbf{f}_K}{\longleftrightarrow} \mathbf{z}$$

normalizing flow

Normalizing flows are generative models which produce tractable distributions.

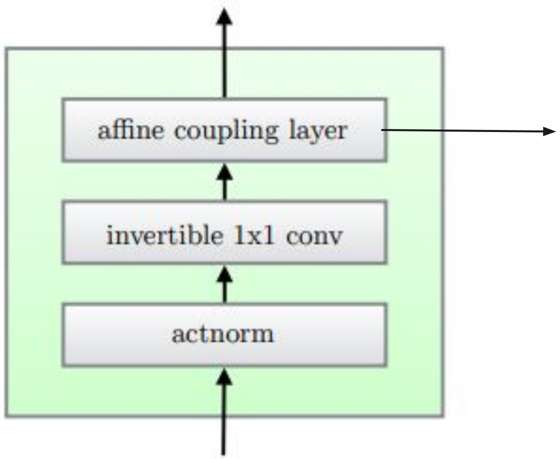
The probability density function of the model given a datapoint:

$$\boxed{\begin{aligned}\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \log|\det(d\mathbf{z}/d\mathbf{x})| \\ &= \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \sum_{i=1}^{K} \log|\det(d\mathbf{h}_i/d\mathbf{h}_{i-1})|\end{aligned}}$$

The log-determinant is the change in log-density when going from h_{i-1} to h_i under transformation f_i.

Glow: Generative Flow

Generative flow where each step



$$(x_a, x_b) \mapsto (y_a, y_b)$$

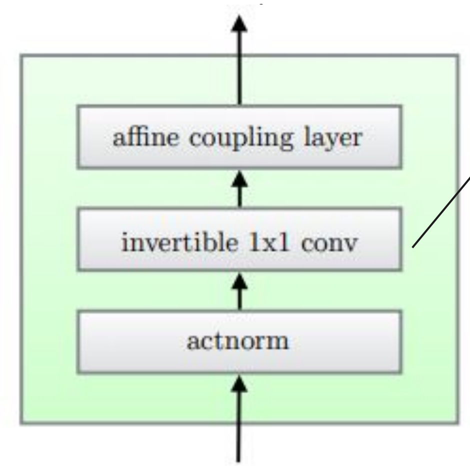
$$y_a = x_a$$

$$y_b = s(x_a) \odot x_b + t(x_a)$$

$$det\left(\frac{dy}{dx}\right) = det\begin{pmatrix} I & 0 \\ \frac{dy_a}{dx_a} & \frac{dy_b}{dx_b} \end{pmatrix} = det\left(\frac{dy_b}{dx_b}\right)$$

Glow: Generative Flow

Generative flow
where each step



$$y_{ij} = W x_{ij}$$
$$x_{ij} \in \mathbb{R}^d, W \in \mathbb{R}^{d \times d}$$

Log-determinant

$$det(\frac{dy}{dx}) = h \cdot w \cdot log|det(W)|$$

$$W = PL(U + \text{diag}(s))$$

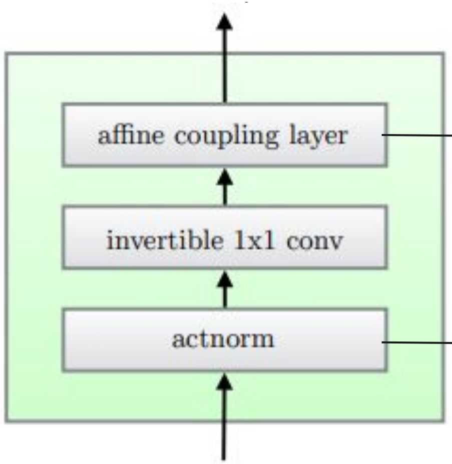
or

$$\boxed{W = U \Sigma V^T}$$

Can use FastH(SVD) here

Glow: Generative Flow

Generative flow
where each step



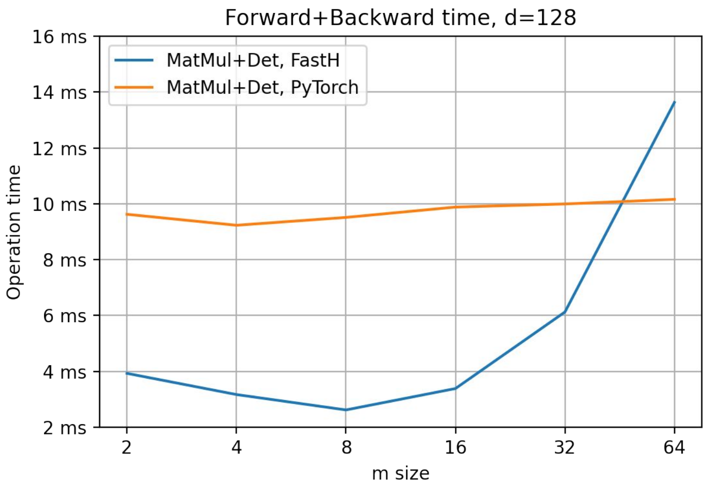
$$(x_a, x_b) \mapsto (y_a, y_b)$$
$$y_a = x_a$$
$$y_b = s(x_a) \odot x_b + t(x_a)$$

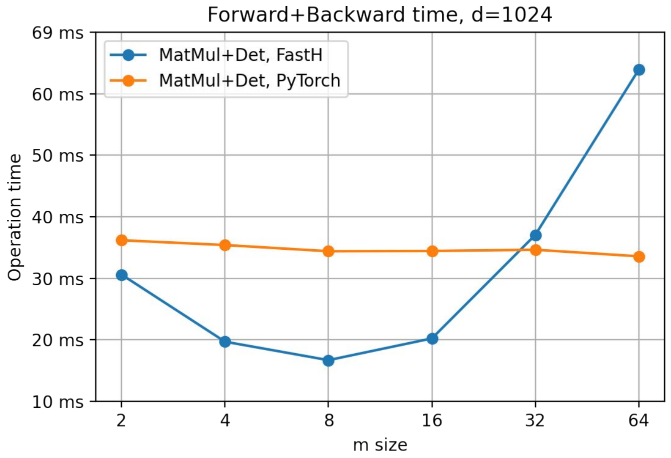
$$det\left(\frac{dy}{dx}\right) = det\begin{pmatrix} I & 0 \\ \frac{dy_a}{dx_a} & \frac{dy_b}{dx_b} \end{pmatrix} = det\left(\frac{dy_b}{dx_b}\right)$$

$$\forall i, j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$$

$$det\left(\frac{dy}{dx}\right) = h \cdot w \cdot \mathrm{sum}(\log|\mathbf{s}|)$$

How we can use FastH(SVD) here





How we can use FastH(SVD) here

Efficient FastH operation: $UX, U \in \mathbb{R}^{d\times d}, X \in \mathbb{R}^{d\times m}$

But! By algorithm formulation $\boldsymbol{m} \ll \boldsymbol{d}$

For conv 1x1 $X \in \mathbb{R}^{d\times BHW}$

B - batch size
H - height
W - width

$\Rightarrow$ $\boldsymbol{m} \gg \boldsymbol{d}$

Can not be applied straightforwardly!

How we can use FastH(SVD) here

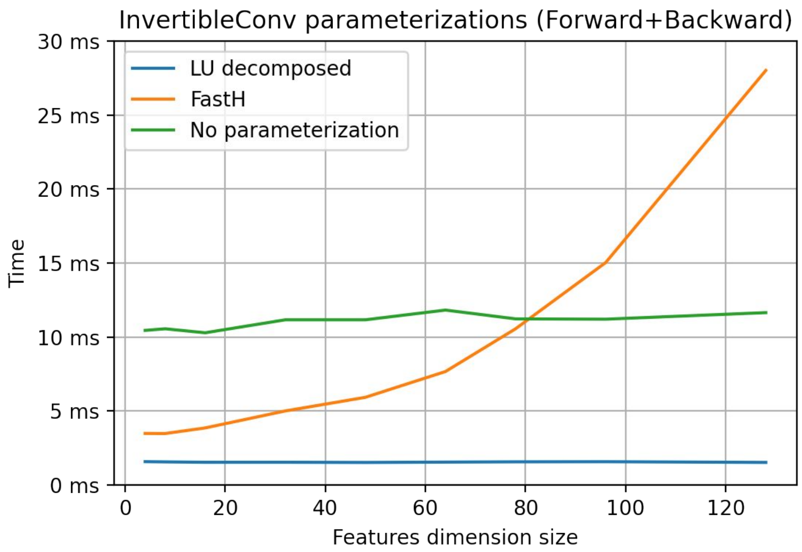
Alternative: 1) extract weight matrix:

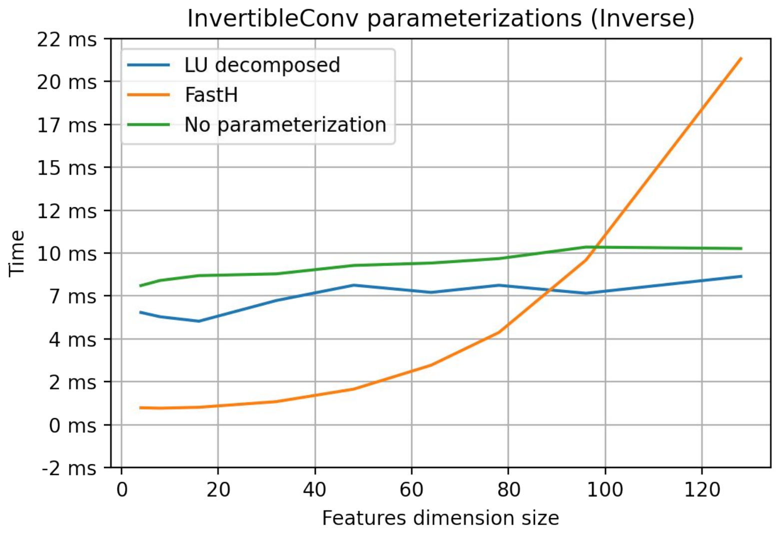
$$W = \begin{bmatrix} U \begin{pmatrix} I_{d/2} \\ 0 \end{pmatrix} \\ U \begin{pmatrix} 0 \\ I_{d/2} \end{pmatrix} \end{bmatrix}$$

using two FastH forwards

2) Apply 2D 1x1 convolution using W matrix

How we can use FastH(SVD) here





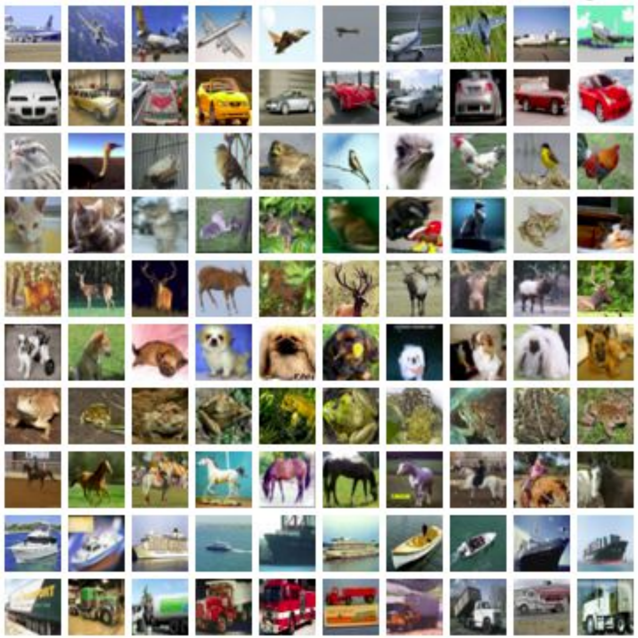
GLOW performance

	Pytorch	FastH
Forward	160 ms	**112 ms**
Inverse	475 ms	**113 ms**
Forward+Backward	513 ms	**245 ms**

~7.5 min (out of 15) per epoch reduction!
(CIFAR10)

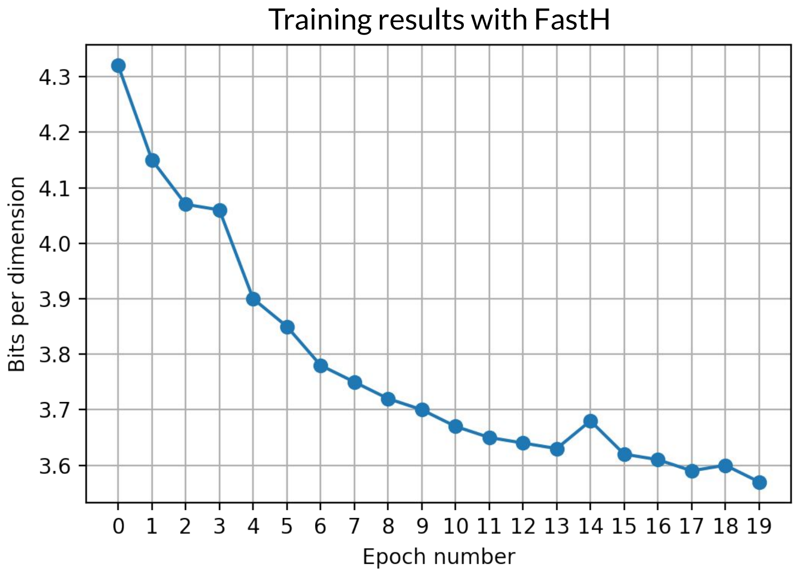
CIFAR-10

Real examples



After 20 epochs

CIFAR-10



$$BPD = \frac{NLL}{3 \ln 2 \cdot H \cdot W}$$

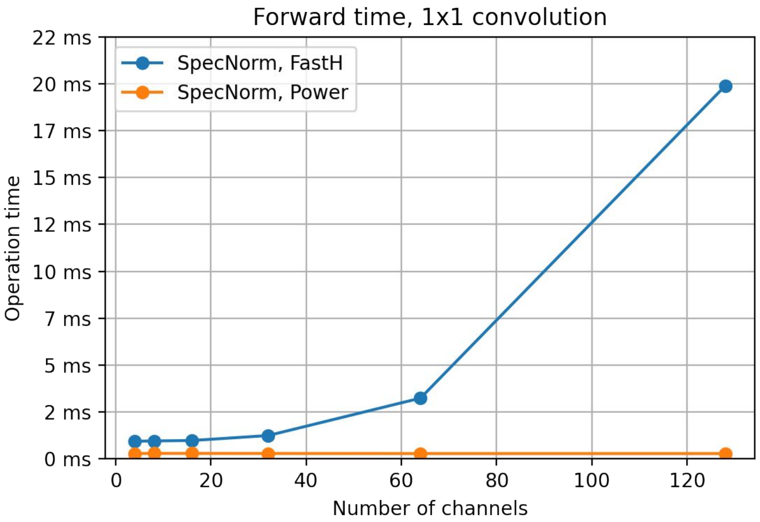
➔ After 20 epochs: **3.57 bpd**

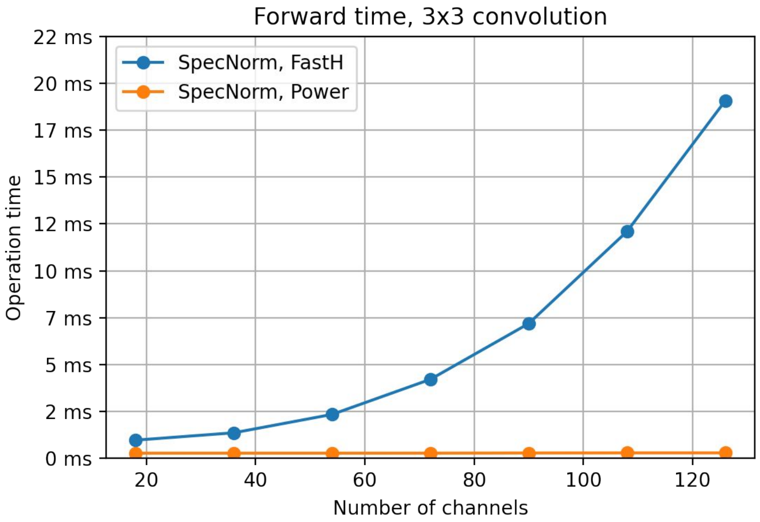
➔ For original GLOW should be ~3.48 after 80 epochs

It works!

Spectral normalization

$$W \in \mathbb{R}^{C_{out} \times C_{in} \times K \times K} \qquad \sigma(W) \to 1$$





Conclusion

- ➔ Practical speed-up can be achieved for GLOW model

BUT:

- ➔ A lot of limitations on parameters choice
- ➔ There is more powerful models (e.g. Neural Splines Flow)
- ➔ The only normalization flow model can be accelerated?

Spectral norms:

- ➔ Slower than power iteration