
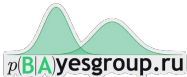






Nondifferentiable models

Ivan Rubachev

Yandex Research, HSE

Plan

- Nondifferentiability in DL
- How we deal with it?
 - Ignore it 
 - Re-parametrization 
 - RL  
 - Try to make it differentiable  
- The end

Nondifferentiability

What's the deal?



31



While digging through the topic of neural networks and how to efficiently train them, I came across the method of using very simple activation functions, such as the **rectified linear unit** (ReLU), instead of the classic smooth **sigmoids**. The ReLU-function is not differentiable at the origin, so according to my understanding the backpropagation algorithm (BPA) is not suitable for training a neural network with ReLUs, since the chain rule of multivariable calculus refers to smooth functions only. However, none of the papers about using ReLUs that I read address this issue. ReLUs seem to be very effective and seem to be used virtually everywhere while not causing any unexpected behavior. Can somebody explain to me why ReLUs can be trained at all via the backpropagation algorithm?

machine-learning

neural-network

deep-learning

backpropagation

Share Improve this question Follow

edited Aug 21, 2020 at 21:46



Stefan Zobel

2,957 ● 7 ● 27 ● 36

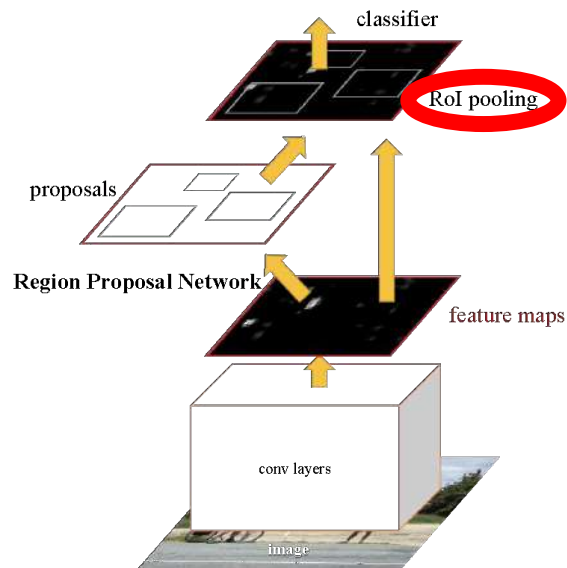
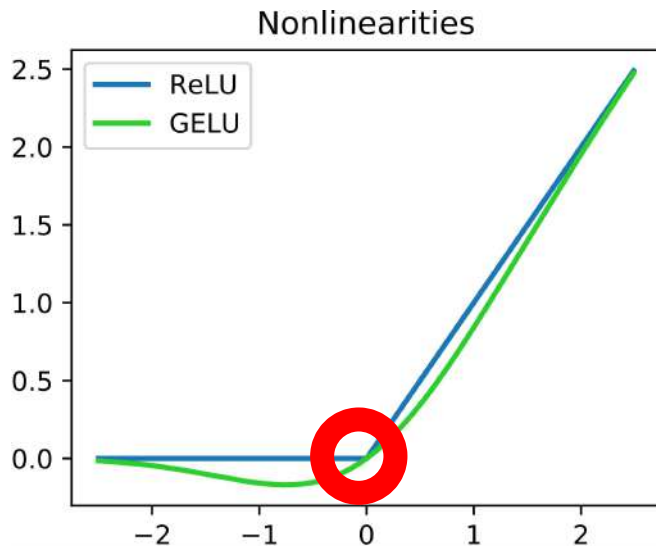
asked May 14, 2015 at 11:59



Yugw O'yu

311 ● 3 ● 4

What's the deal?



Ignoring it

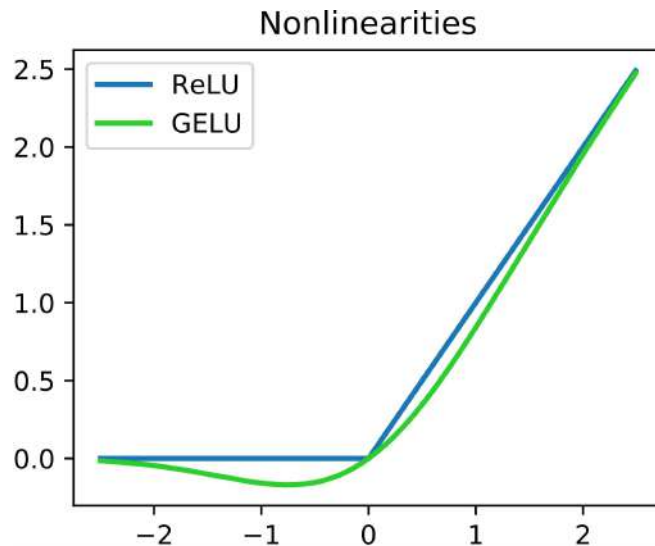
Backpropable



ReLU

- Problem only at 0
- It's rare

$$\nabla_x f(x) = \frac{f(x) - f(x + \varepsilon)}{\varepsilon}$$



Straight through estimators

<https://arxiv.org/abs/1308.3432>

Proposed by Hinton in his lecture 15b

Stochastic nondifferentiable neurons

$$h = f(x, \boxed{a})$$

Example:

$$f = \mathbf{1}_{a > \sigma(h)}, \text{ where } a \sim \text{Uniform}[0, 1]$$

Pretend It's an Identity function



Fast R-CNN

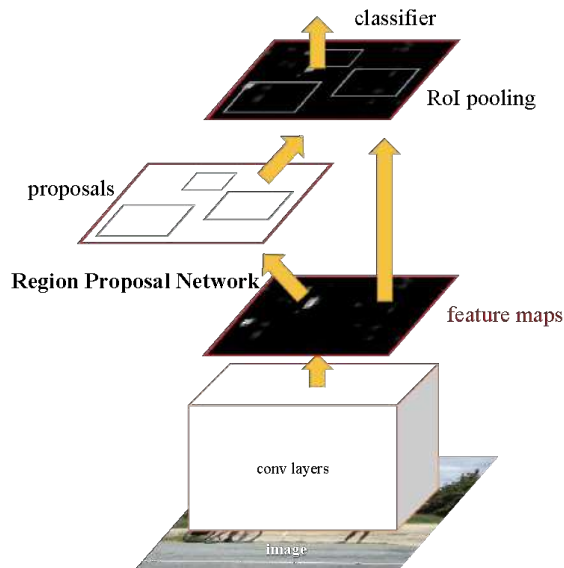
<https://arxiv.org/abs/1504.08083>

- RoI-Pooling is nondifferentiable w.r.t. coordinates
- We ignore and pass gradients through the argmax

Let $x_i \in \mathbb{R}$ be the i -th activation input into the RoI pooling layer and let y_{rj} be the layer's j -th output from the r -th RoI. The RoI pooling layer computes $y_{rj} = x_{i^*(r,j)}$, in which $i^*(r,j) = \operatorname{argmax}_{i' \in \mathcal{R}(r,j)} x_{i'}$. $\mathcal{R}(r,j)$ is the index set of inputs in the sub-window over which the output unit y_{rj} max pools. A single x_i may be assigned to several different outputs y_{rj} .

The RoI pooling layer's backwards function computes partial derivative of the loss function with respect to each input variable x_i by following the argmax switches:

$$\frac{\partial L}{\partial x_i} = \sum_r \sum_j [i = i^*(r,j)] \frac{\partial L}{\partial y_{rj}}. \quad (4)$$



Recent example (Git-Re-Basin)

<https://arxiv.org/abs/2209.04836>

- Searching for the best permutation of weights

$$\arg \min_{\pi} \|\text{vec}(\Theta_A) - \text{vec}(\pi(\Theta_B))\|^2 = \arg \max_{\pi} \text{vec}(\Theta_A) \cdot \text{vec}(\pi(\Theta_B)).$$

We can re-express this in terms of the full weights,

$$\arg \max_{\pi=\{P_i\}} \langle \mathbf{W}_1^{(A)}, \mathbf{P}_1 \mathbf{W}_1^{(B)} \rangle_F + \langle \mathbf{W}_2^{(A)}, \mathbf{P}_2 \mathbf{W}_2^{(B)} \mathbf{P}_1^\top \rangle_F + \cdots + \langle \mathbf{W}_L^{(A)}, \mathbf{W}_L^{(B)} \mathbf{P}_{L-1}^\top \rangle_F,$$

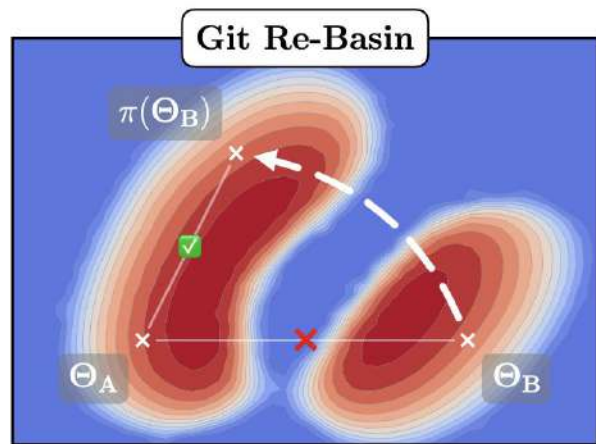


Figure 1: **Git Re-Basin merges models by teleporting solutions into a single basin.** Θ_B is permuted into $\pi(\Theta_B)$ so that it lies in the same basin as Θ_A .

Recent example (Git-Re-Basin)

<https://arxiv.org/abs/2209.04836>

- Searching for the best permutation of weights

Inspired by the success of straight-through estimators (STEs) in other discrete optimization problems (Bengio et al., 2013; Kusupati et al., 2021; Rastegari et al., 2016; Courbariaux & Bengio, 2016), we attempt here to “learn” the ideal permutation of weights $\pi(\Theta_B)$. Specifically, our goal is to optimize

$$\min_{\tilde{\Theta}_B} \mathcal{L} \left(\frac{1}{2} \left(\Theta_A + \text{proj}(\tilde{\Theta}_B) \right) \right), \quad \text{proj}(\Theta) \triangleq \arg \max_{\pi} \text{vec}(\Theta) \cdot \text{vec}(\pi(\Theta_B)), \quad (3)$$

where $\tilde{\Theta}_B$ denotes an approximation of $\pi(\Theta_B)$, allowing us to implicitly optimize π . However, eq. (3) involves inconvenient non-differentiable projection operations, $\text{proj}(\cdot)$, complicating the optimization. We overcome this via a “straight-through” estimator: we parameterize the problem in terms of a set of weights $\tilde{\Theta}_B \approx \pi(\Theta_B)$. In the forward pass, we project $\tilde{\Theta}_B$ to the closest realizable $\pi(\Theta_B)$. In the backwards pass, we then switch back to the unrestricted weights $\tilde{\Theta}_B$. In this way, we

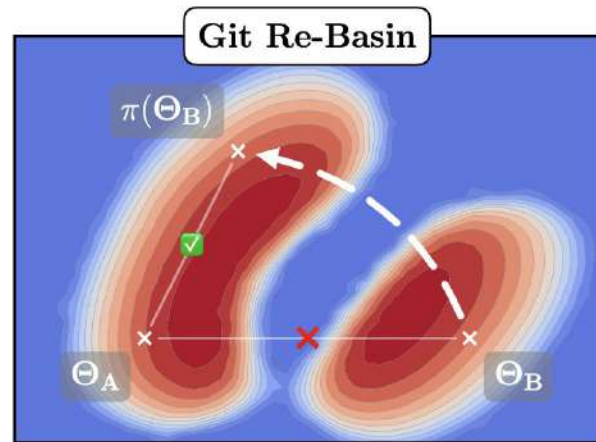


Figure 1: **Git Re-Basin merges models by teleporting solutions into a single basin.** Θ_B is permuted into $\pi(\Theta_B)$ so that it lies in the same basin as Θ_A .

Recent example (Git-Re-Basin)

<https://arxiv.org/abs/2209.04836>

- Searching for the best permutation of weights

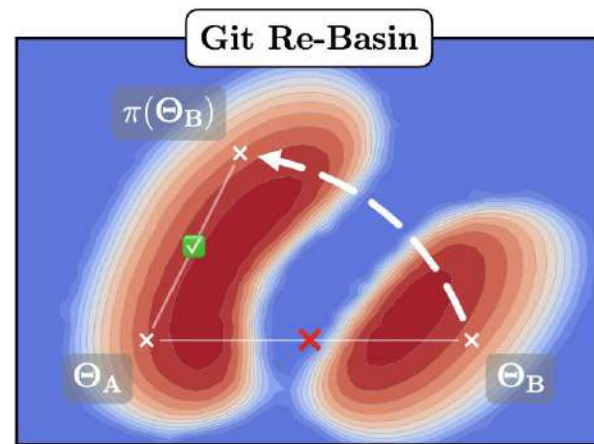
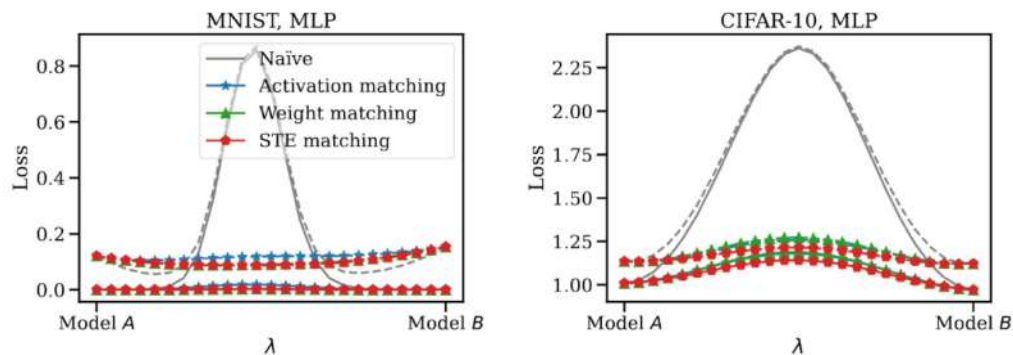
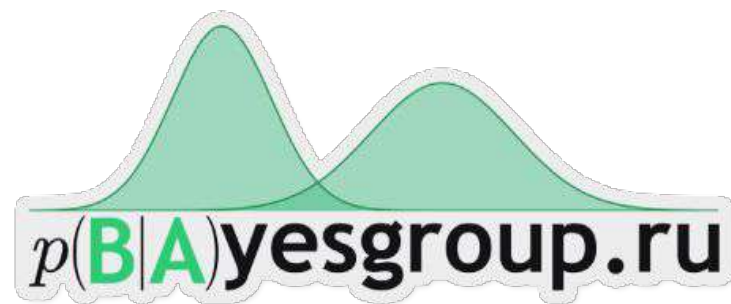


Figure 1: **Git Re-Basin merges models by teleporting solutions into a single basin.** Θ_B is permuted into $\pi(\Theta_B)$ so that it lies in the same basin as Θ_A .

Reparametrization



Обучение: дифференцируемый лосс

- Простой случай:
 - Лосс $L(w)$ – дифференцируемый
 - Распределение $p(w | \theta)$ – «хорошее»
- Репараметризация (если возможна) – самое лучшее решение!

- Разделение случайности и параметров

$$g(\theta, \varepsilon), \varepsilon \sim r(\varepsilon)$$

- Представим распределение $p(w | \theta)$ как

$$z = \mu_\theta(x) + \sigma_\theta(x)\varepsilon, \varepsilon \sim r(\varepsilon)$$

- Тогда градиент легко оценить:

$$\nabla_\theta = \nabla_\theta \int p(w|\theta) L(w) dw = \int r(\varepsilon) \nabla_\theta L(g(\theta, \varepsilon)) d\varepsilon$$

g – детерминированная функция
 ε – шум

- Дисперсия градиента сильно уменьшается

Какие распределения можно репараметризовать?

$p(x y)$	$r(\epsilon)$	$g(\epsilon, y)$
$\mathcal{N}(x \mu, \sigma^2)$	$\mathcal{N}(\epsilon 0, 1)$	$x = \sigma\epsilon + \mu$
$\mathcal{G}(x 1, \beta)$	$\mathcal{G}(\epsilon 1, 1)$	$x = \beta\epsilon$
$\mathcal{E}(x \lambda)$	$\mathcal{U}(\epsilon 0, 1)$	$x = -\frac{\log \epsilon}{\lambda}$
$\mathcal{N}(x \mu, \Sigma)$	$\mathcal{N}(\epsilon 0, I)$	$x = A\epsilon + \mu, \text{ where } AA^T = \Sigma$

Slide credit: Dmitry Vetrov

Подробный обзор: [Mohamed et al., 2019; arXiv:1906.10652]

Нельзя репараметризовать дискретные распределения!

- Категориальное распределение
- **Надо для argmax !**
- Релаксация: Gumbel-Softmax

$$z \sim \operatorname{Discrete}(\alpha_1, \dots, \alpha_L)$$

$$z = (0, 1, 0, \dots, 0)$$

[Jang et al., 2017; Maddison et al., 2017]

$$(z_1, \dots, z_L) \sim \operatorname{RelaxedDiscrete}(\alpha_1, \dots, \alpha_L | T)$$

- $z_i = \frac{\exp((\log \alpha_i + G_i)/T)}{\sum_{j=1}^L \exp((\log \alpha_j + G_j)/T)}, G_k \sim \operatorname{Gumbel}$
- $G_k = -\log(-\log u_k), u_k \sim \operatorname{Uniform}[0, 1]$

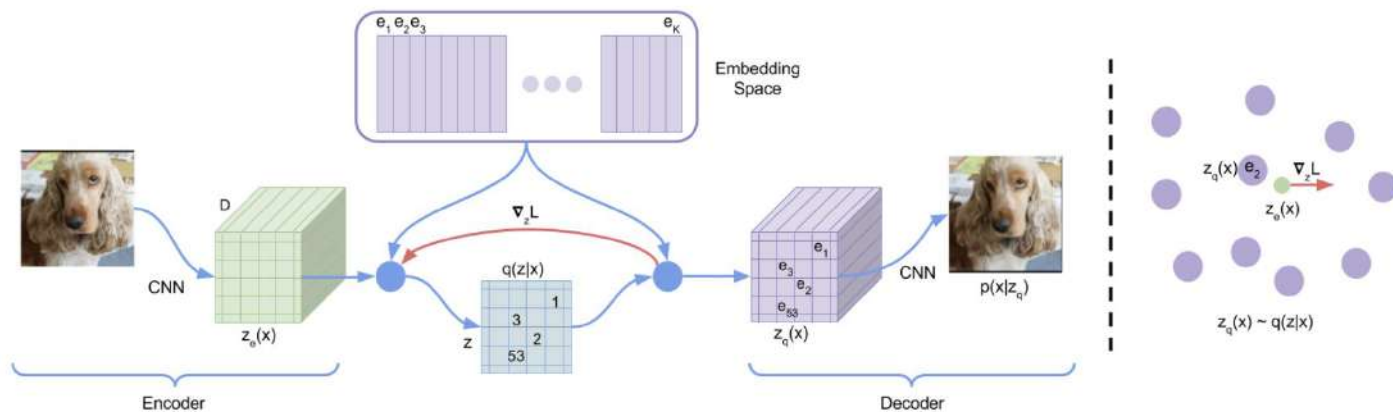
$$\operatorname{RelaxedDiscrete}(\alpha_1, \dots, \alpha_L | T) \xrightarrow{T \rightarrow 0} \operatorname{Discrete}(\alpha_1, \dots, \alpha_L)$$

[Hinton et al., 2015 course]
[Bengio et al. (2013)]

$$\nabla_{\alpha} \approx \nabla_z$$

VQVAE

(Straight-through estimators again)



RL



REINFORCE

In Reinforcement Learning (via the **log-derivative trick**)

$$E_h[(R - b) \cdot \frac{\partial \log p_\theta(h)}{\partial \theta}] = \frac{\partial E_h[R]}{\partial \theta}$$

Could also be used to estimate gradients in stochastic neurons

(Bernoulli “actions” with probability given by parameter in case of a threshold)

$$f = \mathbf{1}_{a > \sigma(h)}, \text{ where } a \sim \text{Uniform}[0, 1]$$

$$p_\theta(h) \frac{\nabla p_\theta(h)}{p_\theta(h)} = p_\theta(h) \nabla \log p_\theta(h)$$

Making it differentiable



Differentiable everything



Swift for TensorFlow (Archived)

Swift for TensorFlow was an experiment in the next-generation platform for machine learning, incorporating the latest research across machine learning, compilers, differentiable programming, systems design, and beyond. It was archived in February 2021. Some significant achievements from this project include:

- Added [language-integrated differentiable programming](#) into the Swift language. This work continues in the official Swift compiler.
- Developed a mutable-value-semantics-oriented [deep learning API](#).
- Fostered the development of [a model garden](#) with more than [30 models from a variety of deep learning disciplines](#).
- Enabled novel research that [combines deep learning with probabilistic graphical models](#) for 3D motion tracking and beyond.
- Powered a(n almost) pure-Swift prototype of a [GPU+CPU runtime supporting parallel map](#).
- Spun off multiple open source side efforts which continue to be under active development:
 - [PythonKit](#): Python interoperability with Swift.
 - [swift-jupyter](#): Enables use of Swift within Jupyter notebooks.
 - [swift-benchmark](#): Provides a robust benchmarking suite for Swift code.

Differentiable everything

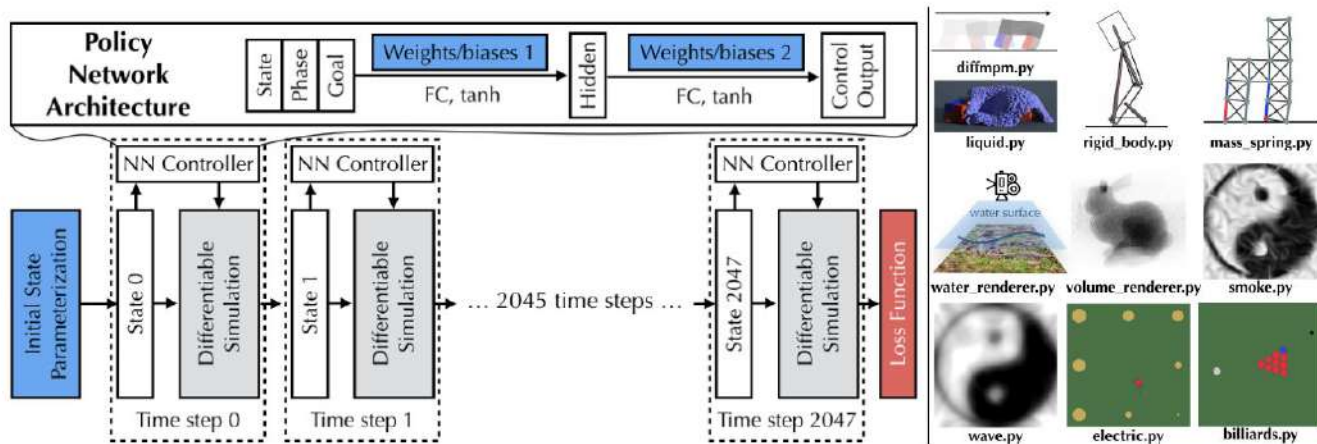


Figure 1: **Left:** Our language allows us to seamlessly integrate a neural network (NN) controller and a physical simulation module, and update the weights of the controller or the initial state parameterization (blue). Our simulations typically have 512 ~ 2048 time steps, and each time step has up to one thousand parallel operations. **Right:** 10 differentiable simulators built with DiffTaichi.

Beware

(insert high variance estimation error: didn't find the tweet with a nice plot)

Conclusion

- Ignore
 - ReLU, max pooling,
- Stochastic
 - Reparametrization
 - RL
- Making differentiable
- Better avoid it