Deep Learning Frontiers: Hyper-Optimization and Learnable Parameters

Elouan Gardès [*1] **Arthur Juigner** [*1] **Thibault Mazette** [*1]

Abstract

In Machine-Learning, we make a clear distinction between parameters and Hyper-Parameters (HPs). In the context of Neural Networks (NNs), typical HPs include the learning rate, batch sizes, but also architectural choices and more. However, Deep-Learning and back-propagation are not restrictive about the definition of a learnable parameter and can in theory optimize some HPs too, a process referred to as **Hyper-Optimization** (HO). *Chandra et al. 2022* used Automatic Differentiation (AD) and made HO practical in usual frameworks for the Learning-Rate (LR). Inspired by their work, we dive deeper into the realm of HO. We first make a study of HO, motivated by the potential new trade-offs that appear with this technique regarding batch-sizes. We also introduce a new way to train deep NNs based on setting different LRs for every layer and show that it outperforms baselines. Finally, we propose a new GELU (Hendrycks & Gimpel, 2023) activation with learnable parameters and show that it outperforms GELU on all tested tasks.

1. Introduction

Chandra et al. 2022 proposed a simple and practical way to set learnable LRs, and our goal is to build upon those foundations to extend and study HO. First, we study the effect of batch size on convergence and on training dynamics.

Then, we propose two modifications to their algorithm to allow further improvement in training speed, generalization and stability. Indeed, one of the downsides of Chandra et al. 2022's method is its tendency to become unstable if the initial LR is not precisely set as will be shown. This removes some interest to the technique which claimed to eliminate the need for precise LR study and forces the user to set low LRs which take time to adjust. Furthermore, we observe that the learning rate sometimes takes negative values, which is nonsensical and unstable. We propose a new link function that solves these issues.

We also propose to use and tune different LRs for different layers in a NN. This helps with a common training problem, which is that gradients tend to have different magnitudes in different layers, leading to slow learning for some. Tuning multiple LRs by hand is hard, however it is a non-issue if LRs are learned automatically through HO.

We also propose learning other HPs such as skip connections' weights and activation functions aiming for performance and interpretability. Intuitively, a learnt skip connection lets the model "choose" during training which layers to give more importance to. We test this hypothesis and study the evolution of the learnt weights.

Activation functions are at the core of NNs' ability to learn (Nwankpa et al., 2018). Unlike the batchsize or depth or width, activation functions can be updated online as long as we allow their parameters to enter the computational graph in the forward pass, just like weights. Using learnable activation functions is not a new idea: Chen & Chang 1996 proposed a parametrization of $tanh$ and σ for instance. These activations are not widely used anymore however, which limits the benefits of their learnable versions. He et al. 2015b introduced a partially parametric ReLU (Agarap, 2019) learnt in the same pass as the other parameters and improved significantly over ReLU on ImageNet (Deng et al., 2009). While ReLU remains used, its smoother counterparts like GeLU are now prefered and show better performance. More recently, Goyal et al. 2020 and Fang et al. 2023 have proposed learnable polynomial activation functions with the idea that polynoms can theoretically learn the optimal activation for any problem, granted a high-enough order. These methods however show limitations in that they require significant regularization which limits their use and practicality.

We introduce AdaGELU, a learnable activation function based on GELU. We show that it performs better than GELU on various vision tasks. We also test a fully parametrized Leaky-ReLU (Maas et al., 2013) and use it with AdaGELU to derive hypotheses on what constitutes a good candidate for a learnable activation.

[*]Equal contribution [1]Department of Computer Science, ETH Zürich, Zürich, Switzerland .

Student Project of the Deep Learning Course at ETH Zürich, Switzerland, 2023.

2. Models and methods

2.1. Batch size

With better parallel compute, higher batch-sizes have allowed for faster computation in Deep Learning. Nevertheless, research by Keskar et al. 2017 has shown that using large batch sizes can lead to diminished generalization performance. Since using HO means that smaller batch-sizes are also linked to more frequent updates for HPs, we hypothesize that they could become even more beneficial.
We designed experiments to test this hypothesis drawing inspiration from the methodology proposed by Zhang et al. 2019. For a given model, optimizer, loss function, and learning rate, we systematically varied batch sizes, calculating the number of iterations and time required to achieve a target accuracy. Notably, we take into account the time spent evaluating accuracy on the validation set into the training time, considering its potential utility for early stopping or monitoring the training evolution in practical settings.

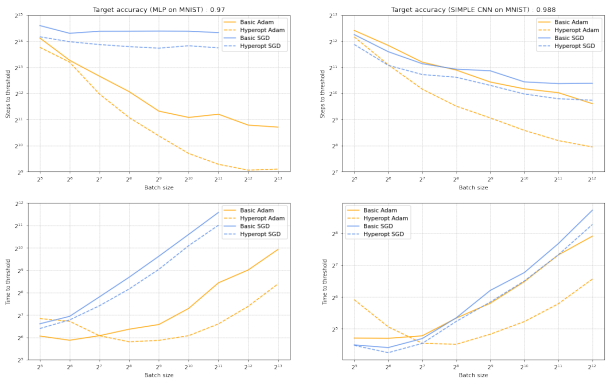

Figure 1. Empirical link between batch size and steps to result

Figure 1 shows the force of HO. However, our hypothesis seems wrong: the benefits on very low batch sizes were marginal or even negative. We believe that this phenomenon can be attributed to the substantial and typical variance introduced by small batch sizes, leading to training instability. Another idea is that in the case of the Adam optimizer (Kingma & Ba, 2017), a competition arises between HO, favoring previous directions, and Adam diminishing the weights of previous strong directions. We also see that when considering the computation time to reach a target accuracy, the optimal batch size in terms of time efficiency corresponds to a medium one for Adam, suggesting a trade-off. We thus reject our initial hypothesis but confirm the validity of using HO and explored most-effective batch-sizes.

2.2. Adding a link function to the adaptive learning rate

In *Chandra et al. 2022*'s work, the LR α is learnt through AD, which practically can lead to negative LRs or instability by blow-up, for which we need an extremely low hyper-LR in practice. We propose adding a link function, namely using $\exp(\alpha)$, where α is the learned parameter. This allows for more stability and flexibility for the choice of the hyper-LR, along with more expressiveness : in our experiments, the learned LR now evolves with more magnitude than it previously did. Furthermore, we typically think of learning rates on a log-scale rather than on a linear scale (intuitively considering 0.01, 0.001 and 0.0001 in a series when tuning), which makes adding an exponential link intuitive.

2.3. Multiple learning rates

Since HO enables learning the LR automatically, we propose having independent LRs for independant layers, and also having one LR per parameter of the model. We call these algorithms SGDLW-SGD and SGDPW-SGD (**L**ayer-**W**ise and **P**arameter-**W**ise) and we always couple them with the $\exp$ link function, for reasons mentioned above. These methods are motivated by the following observation: suppose you want to minimize the quadratic form $f(x) = x^T Hx$ with $H = diag(100, 1)$. One of the components will have large gradients, which with the same LR will lead to sub-optimal updates if not unstable ones. That would be fixed with independent and learnt LRs.

2.4. Adaptive skip connections

Skip connections *(Srivastava et al., 2015)* are a well-known addition to convolutional nets (CNNs, LeCun et al. 1995) and have proved their effectiveness many times, e.g in ResNets *(He et al., 2015a)*. We propose adding a learnable weight to the skip connection: instead of the output being $y = f(x) + x$, we make it $y = (1 - \sigma(w_s))f(x) + \sigma(w_s)x$, where w_s is a new parameter and σ the sigmoid function. We want for the model to have the ability during training to select for each skip whether to favor the skip-connection or the activation, as much for inference as for gradient flow. We also want to analyze $\sigma(w_s)$ and try to make the skip-connection even more interpretable. Notice that our approach differs from highway networks *(Srivastava et al., 2015)*: these networks add many parameters and expressiveness to the model, resulting in a brand new architecture. Our goal is to allow the model to learn whether to favor a skip connection or not in a much more restricted setting.

2.5. Learnable GELU

Following a usual parametrization of the GELU (e.g Lee 2023), we define:

$$AdaGELU(x, \alpha, \beta, \gamma) = x \cdot \hat{\Phi}(\alpha x, \beta, \gamma)$$
$$with\ \hat{\Phi}(\alpha x, \beta, \gamma) = \frac{1}{2}(1 + \tanh(\beta(\alpha x + \gamma(\alpha x)^3)))$$

where we further use the usual approximation for the Cumulative Distribution Function of the gaussian distribution, and

where the usual GELU is recovered for $\alpha = 1$, $\beta = \sqrt{\frac{2}{\pi}}$ and $\gamma = 0.044715$. In this parametrization, α controls the steepness of GELU, while β and γ are chosen to minimize the approximation error.

For training AdaGELU, we define an optimizer for its parameters and update them as for the model's. Pseudocode for the update rule can be found in B.1.
Unlike previous works, we do insist on setting a different optimizer for AdaGELU than for the model. That way, we are able to use different kinds of optimization strategies as well as different learning rates for both which further increases the generalizability of our method.
In a network where we would typically find different GELUs or ReLUs, we can instantiate arbitrarily many AdaGELUs which will each evolve with their own set of parameters and better fit to their own role in the model.
A parametrization of the Leaky-ReLU is also presented in B.3 and discussed in the results.

3. Results

3.1. Link function and separate learning rates

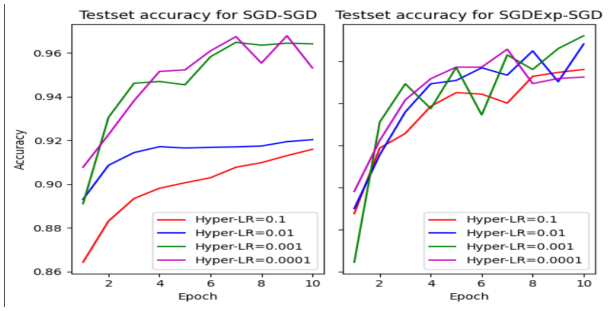

Figure 2. SGD-SGD's and SGDExp-SGD's: various hyper LRs

Figure 2 shows how the SGD-SGD algorithm from *Chandra et al. 2022* compares to our version with the exp link function (referred to as SGDExp-SGD) for different hyper-learning rates, when trained on the MNIST dataset with a small fully-connected neural network. We observe that our version is more stable and forgives "bad" initial choices of hyper-LRs. Moreover, the SGD-SGD version often made the LR slighty negative, altough it never made the model blow up in this small experiment.

In Figure 3, we then compared the training of relevant algorithms on CIFAR10, using a CNN of about 220k parameters: basic SGD, SGD-SGD, SGDExp-SGD, SGDLW-SGD, SGDPW-SGD (recall: SGDLW means one LR per layer, SGDPW means one per parameter). The best performance is obtained by having one LR per layer, and the second best by having one LR for everything, always using the exp link. We observed that one LR per layer always improves training, while one LR per parameter seems to give more erratic results over many runs. Adam should be tested next.

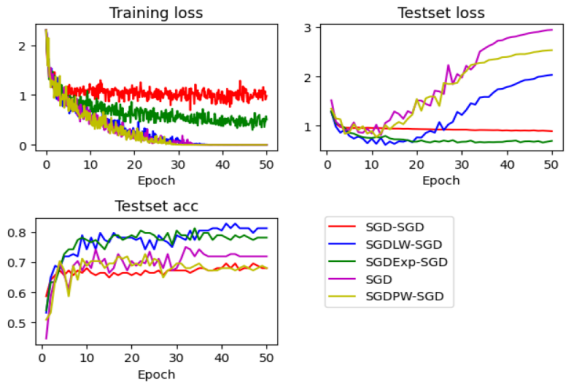

Figure 3. Comparison of five optimization methods. Starting LRs at exp(-2), hyper-LRs at 0.01 with link and 0.0001 without.

3.2. Adaptive skip connections

We created a CNN with multiple residual blocks each containing two convolutional layers, two activations σ, and a skip connection: $y = \sigma \circ conv1 \circ \sigma \circ conv2(x) + x$. We compared training with regular skip connections. Tests were mostly performed with 2-4 such blocks and on MNIST and CIFAR-10 ; performance did not noticeably change during training, and it does not seem promising for use in practice. However, looking at the evolution of the skip connections' weights is interesting, as seen in Figure 4 : we notice that in the first block, the skip connection's weight drops significantly. This corroborates the classical intuition that skip connections enable the gradients to flow deep into the model, since the first layer does not need to forward any gradient to previous layers, unlike for the next blocks where we still want the gradient to flow back to the first layers.

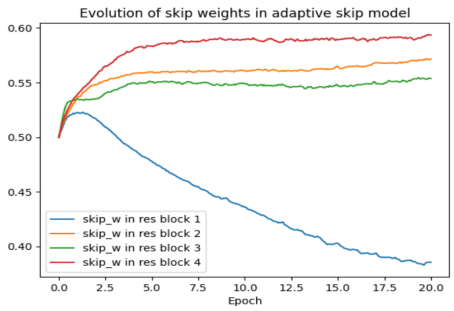

Figure 4. Skip connections' weights while training. Higher than 0.5 means the skip connection has more weight than the activation.

3.3. Learnable GELU

We tested AdaGELU on three sets of problems using two architectures. With a CNN, we tested AdaGELU on the

MNIST and CIFAR-10 datasets. With a small Resnet (He et al. 2015c, Nouman 2022), we tested AdaGELU on the bigger and harder Caltech-256 dataset (Griffin et al., 2022). The baseline was each time the same model using GELU or ReLU instead of AdaGELU in the same training conditions. AdaGELU systematically improved convergence speed as well as best metrics over GELU on all benchmarks as presented in fig 5. We also report the best training accuracies observed on Caltech-256 and CIFAR-10 in table 3.3.

Our results are particularly promising for bigger model and problems as a near 3 points gap just by changing an activation is really large. More tests are needed to confirm the general benefits but we emphasize that it was observed on every run, for every dataset and model tested.

From more extensive analysis (see also B.3), we believe that learnable activations like AdaGELU or AdaReLU have two effects. First, they increase training speed possibly by adjusting gradient flow. E.g AdaReLU adjusts the gradient directly by changing slopes, which is equivalent **at update time** to having a different LR. Second, unlike learnable LR, the learnt activation has an influence at inference. This influence is not always beneficial from our tests with AdaReLU, where if the activation changed a lot for gradient flow, it may not allow better generalization afterwards (see B.3 for further analysis). It seems that because AdaGELU's parameters have a more subtle effect on gradient flow (see B.2), it is able to learn useful patterns from the data and has shown generalization capabilities on all tests.

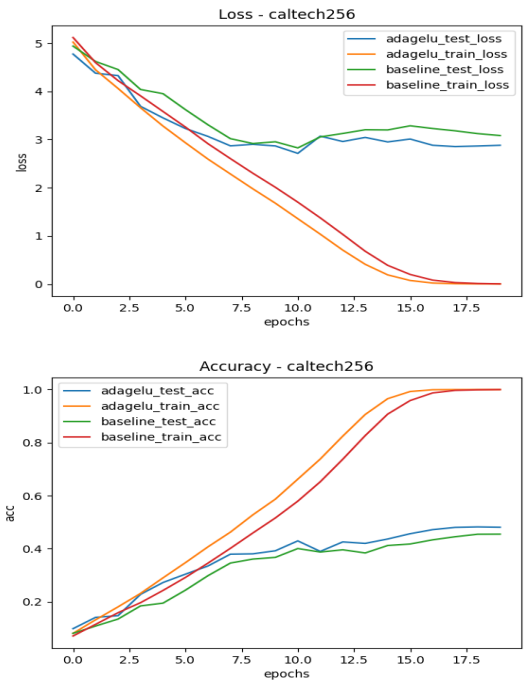

Figure 5. AdaGELU on Caltech-256 with a RNN

Dataset+base model	Baseline	Ours - AdaGELU
CIFAR-10+CNN	72.3	**73.3**
Caltech-256+RNN	45.5	**48.2**

Table 1. AdaGELU best test accuracies

4. Discussion

In light of computational resource constraints, our experiments were conducted on restricted model configurations. It is necessary to conduct more experiments on scaled-up models and datasets to confirm the benefits of the proposed methods.

We also want to mention proposals which were not followed because of these constraints: Regarding hyperparameter transfer, the works from Yang et al. 2022 only work well for width but not depth according to Bordelon et al. 2023. Along their solution, we could use our technique of one LR per layer to also solve the depth transfer problem.
Regarding model ensemble with hyperparameter learning, we would need further resource to assess its relevance.
Regarding the use of the learnt LR to assess the local loss landscape, our early analysis showed very little potential as the learnt LR does not converge to known optimal LRs of simple problems.

5. Summary

We showed that the classical interpretation of skip connections as a "highway for gradients" is in line with the adaptive skip connection experiments, although these did not seem to help much with training.

We also proposed two new tricks (exp link function for the LR and per-layer LRs) to successfully enhance the adaptive learning rate strategy from *Chandra et al. 2022*. In practice, we saw great improvement and no drawbacks from these techniques, and they intuitively help counteract the problems of unstability with respect to hyper-LRs, and uneven gradient norms throughout layers.

We finally introduced new learnable activations and in particular we proposed AdaGELU. We showed that it improved training dynamics and generalization on all tested tasks.

References

Agarap, A. F. Deep learning using rectified linear units (relu), 2019.

Bordelon, B., Noci, L., Li, M. B., Hanin, B., and Pehlevan, C. Depthwise hyperparameter transfer in residual networks: Dynamics and scaling limit, 2023.

Chandra, K., Xie, A., Ragan-Kelley, J., and Meijer, E. Gradient descent: The ultimate optimizer, 2022.

Chen, C.-T. and Chang, W.-D. A feedforward neural network with function shape autotuning. *Neural networks*, 9 (4):627–641, 1996.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

Fang, H., Lee, J.-U., Moosavi, N. S., and Gurevych, I. Transformers with learnable activation functions, 2023.

Goyal, M., Goyal, R., and Lall, B. Learning activation functions: A new paradigm for understanding neural networks, 2020.

Griffin, G., Holub, A., and Perona, P. Caltech 256, Apr 2022.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015a.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015b.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015c.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus), 2023.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

LeCun, Y., Bengio, Y., et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

Lee, M. Gelu activation function in deep learning: A comprehensive mathematical analysis and performance, 2023.

Maas, A. L., Hannun, A. Y., Ng, A. Y., et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, pp. 3. Atlanta, GA, 2013.

Nouman. Writing ResNet from Scratch in PyTorch, howpublished = https://blog.paperspace.com/writing-resnet-from-scratch-in-pytorch/, note = Accessed: 2024-01-01, 2022.

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. Activation functions: Comparison of trends in practice and research for deep learning, 2018.

Pytorch. LeakyReLU - Pytorch 2.1 documentation, howpublished = https://pytorch.org/docs/stable/generated/torch.nn.leakyrelu.html, note = Accessed: 2023-11-30.

Srivastava, R. K., Greff, K., and Schmidhuber, J. Highway networks, 2015.

Yang, G., Hu, E. J., Babuschkin, I., Sidor, S., Liu, X., Farhi, D., Ryder, N., Pachocki, J., Chen, W., and Gao, J. Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer, 2022.

Zhang, G., Li, L., Nado, Z., Martens, J., Sachdeva, S., Dahl, G. E., Shallue, C. J., and Grosse, R. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model, 2019.

A. Batch sizes experiments hyperparameters

Table 2. Hyperparameters for batch sizes experiments

DATA SET	MODEL	OPTIMIZER	LR	TARGET ACCURACY	ITERATIONS
MNIST	MLP	SGD	0.1	0.97	10
MNIST	MLP	SGD HYPEROPT	0.1	0.97	10
MNIST	MLP	ADAM	0.001	0.97	10
MNIST	MLP	ADAM HYPEROPT	0.001	0.97	10
MNIST	SIMPLE CNN	SGD	0.1	0.988	10
MNIST	SIMPLE CNN	SGD HYPEROPT	0.1	0.988	10
MNIST	SIMPLE CNN	ADAM	0.0001	0.988	10
MNIST	SIMPLE CNN	ADAM HYPEROPT	0.0001	0.988	10

B. Learnable activations

More details and complementary work with approaches which did not perform as hoped.

B.1. Code, models and hyperparameters used

Below is the code for AdaGELU in pytorch, python.

```
The following code was used for AdaGELU:

class adaGeLU(nn.Module):

    def __init__(self):

        super(adaGeLU, self).__init__()

        self.parameters = {'alpha': nn.Parameter(torch.tensor(1., requires_grad=True)),
                           'beta': nn.Parameter(torch.tensor(np.sqrt(2/np.pi),
    requires_grad=True)),
                           'gamma': nn.Parameter(torch.tensor(0.044715, requires_grad=True
    ))}
        self.all_params_with_gradients = [self.parameters['alpha'], self.parameters['beta'
    ], self.parameters['gamma']]

    def forward(self, input):
        output = (1/2) * input * (1 + F.tanh(self.parameters['beta'] *
                                             (self.parameters['alpha']*input +
                                              self.parameters['gamma']*(self.parameters['
    alpha']*input)**3)))
        return output
```

Listing 1. AdaGELU code

Here is the pseudo-code for the update rule of AdaGELU:

Algorithm 1 AdaGELU update rule

$\alpha \leftarrow 1$
$\beta \leftarrow \sqrt{\frac{2}{\pi}}$
$\gamma \leftarrow 0.044715$
adaGELU(x) $\leftarrow x \cdot \hat{\Phi}(\alpha x, \beta, \gamma)$
adaOpt $\leftarrow$ Optimizer(adaGELU.parameters, LR)
while training **do**
    pred $\leftarrow$ model(data)
    loss $\leftarrow$ Loss(pred, true)
    modelOpt.zeroGrad()
    adaOpt.zeroGrad()
    loss.backward()
    modelOpt.step()
    adaOpt.step()
end while

where we assume that AdaGELU is used as an activation in model and where modelOpt is an optimizer set on the model's parameters.

For our tests on Caltech-256, we used the simple RNN from Nouman 2022 where we replaced ReLUs by GELUS for the baseline and by AdaGELUs for our method. We used Adam as an optimizer with a LR of 0.001, a batchsize of 256 was used and the following transforms were operated.

```
transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224, 224)),
    Lambda(lambda x: x.convert("RGB")),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

Listing 2. transforms

We used three AdaGELUs total (see Nouman 2022) and updated them with Adam and a LR of 0.01.

For CIFAR-10 and MNIST, we used the following CNN:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(16384, 128)
        self.fc2 = nn.Linear(128, 10)
        self.bn1 = nn.BatchNorm2d(3)
        self.bn2 = nn.BatchNorm2d(32)
        self.bn3 = nn.BatchNorm1d(16384)

    def forward(self, x):
        x = self.bn1(x)
        x = self.conv1(x)

        x = adaGelu1(x)

        x = self.bn2(x)
        x = self.conv2(x)
```

```
        x = adaGelu2(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)

        x = self.bn3(x)
        x = self.fc1(x)

        x = adaGelu3(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

Listing 3. CNN code

We also used a batchsize of 256 and Adam with LR 0.001 for the model, and 0.01 for the AdaGELUs.

B.2. AdaGELU visualized

We initialize three AdaGELUs (one per residual block, one last in the end layers) instead of all three GELUs of the baseline.

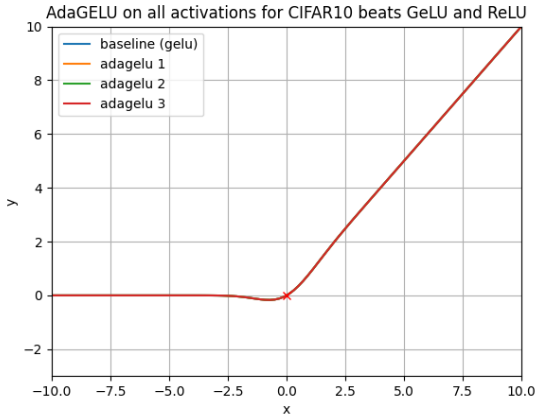

Figure 6. AdaGELUs on Caltech-256 at initialization

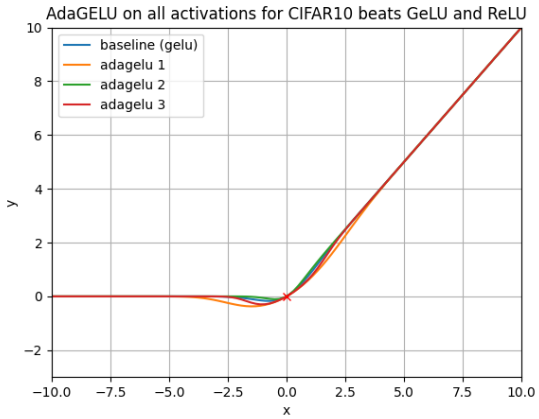

Figure 7. AdaGELUs on Caltech-256 during training

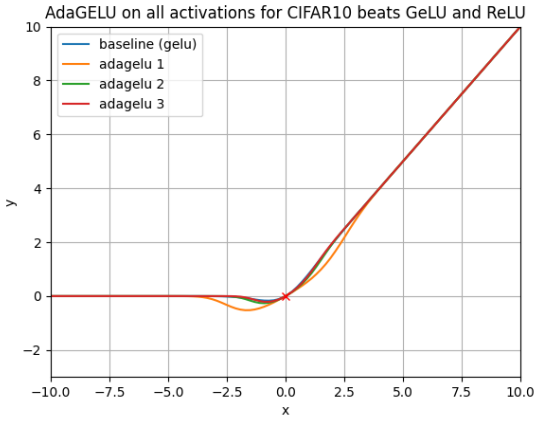

Figure 8. AdaGELUs on Caltech-256 after training

The gradient flow difference is contained but the slight adjustment at the center seem to make all the difference during inference. The first residual block is the most affected by the change. Further analysis are needed to fully comprehend why as well as to test it on other problems.

B.3. AdaReLU parametrized and gradient flow hypotheses

We believe that learnable activations should ideally not concern parameters having a large effect on gradient flow, such as the slopes of AdaReLU. To demonstrate this let us first show the AdaReLU learnt from a low LR (SGD at 0.001) on MNIST with a small CNN:

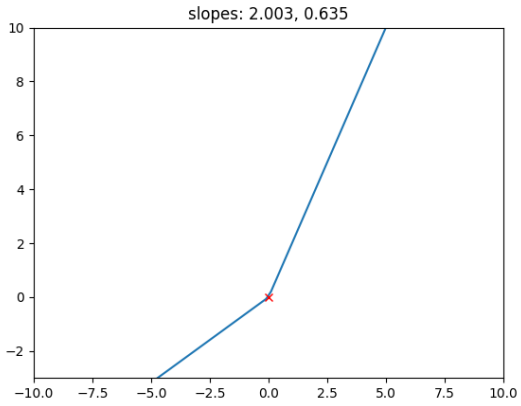

Figure 9. AdaReLU on MNIST with low LR

Let us now compare it with the same AdaReLU learnt with an SGD set at 0.06 (60 times higher):

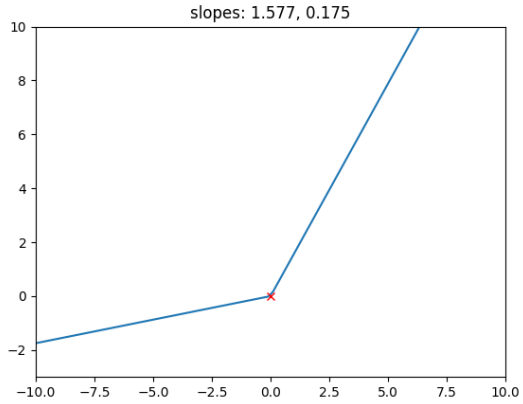

Figure 10. AdaReLU on MNIST with high LR

Both activations started with slopes of 1 and 0.01.
When looking at training and testing statistics, we further see that while AdaReLU does improve significantly over traning metrics (both accuracy and loss), generalization is worse than the baseline. We analyze these curves and statistic with the hypotheses that learnable activations with parameters having a large effect not only on the function, but also directly on gradient flow, are to avoid. Because they are trained through backpropagation, they may learn to correct for poorly-set LRs by either increasing or decreasing gradient flow in the model to reduce the training loss quicker. However this behaviour is not desirable as the actually-learnt activation has then little reason to better generalize.

This hypotheses ought to me confirmed on further testing and more rigourous analysis of the gradient flow during training, for which we lacked time.

Here is a **parametrizaion for AdaReLU**: we expand upon the work by He et al. 2015b and consider the following :

$$AdaReLU(x, \alpha, \beta) = \begin{cases} \alpha x & \text{if } x \geq 0, \\ \beta x & \text{if } x < 0. \end{cases}$$

We then initialize AdaReLU with $\alpha = 1$ and $\beta = 0.01$ following the default from Pytorch.

B.4. Other learnable activations

Motivated by the idea to build very general and learnable activations, we also tried two other approaches.
First, we rediscovered learnable polynoms but we encountered large difficulties training models in different settings despite high regularization. As soon as the polynom's order becomes high-enough to enable benefits, very problem-specific efforts need to be made to prevent numerical difficulties.

We also introduced Fourier-series as potential universal-learner in that they can theoretically fit a very wide range of functions while being very stable as an n-order Fourier-series is bounded by 2n+1 when the coefficients are bounded by 1. We tested the following Fourier-Series Activation (FSA):

$$FSA_n(x) = a_0 + \sum_{k=1}^{n} \left(a_k \cos(\frac{2\pi k}{T} x) + b_{k-1} \sin(\frac{2\pi k}{T} x) \right),$$

where $(a_k)_{k\in[0:n]}$, $(b_k)_{k\in[1:n]}$ and some period T would all be learnt during training. We tested this activation with an initialization on usual activations like ReLU (meaning we fit the series to ReLU before training from scratch a NN) and with random initialization but we were never able to achieve benefits in training or testing statistics even though stability was indeed observed.