Scaling MLPs: A Tale of Inductive Bias

Gregor Bachmann, Sotiris Anagnostidis, Thomas Hofmann

Published in 2023 in NeurIPS
13 citations as of this week

Presenter: Abdullah Mamun

Date: June 12, 2024

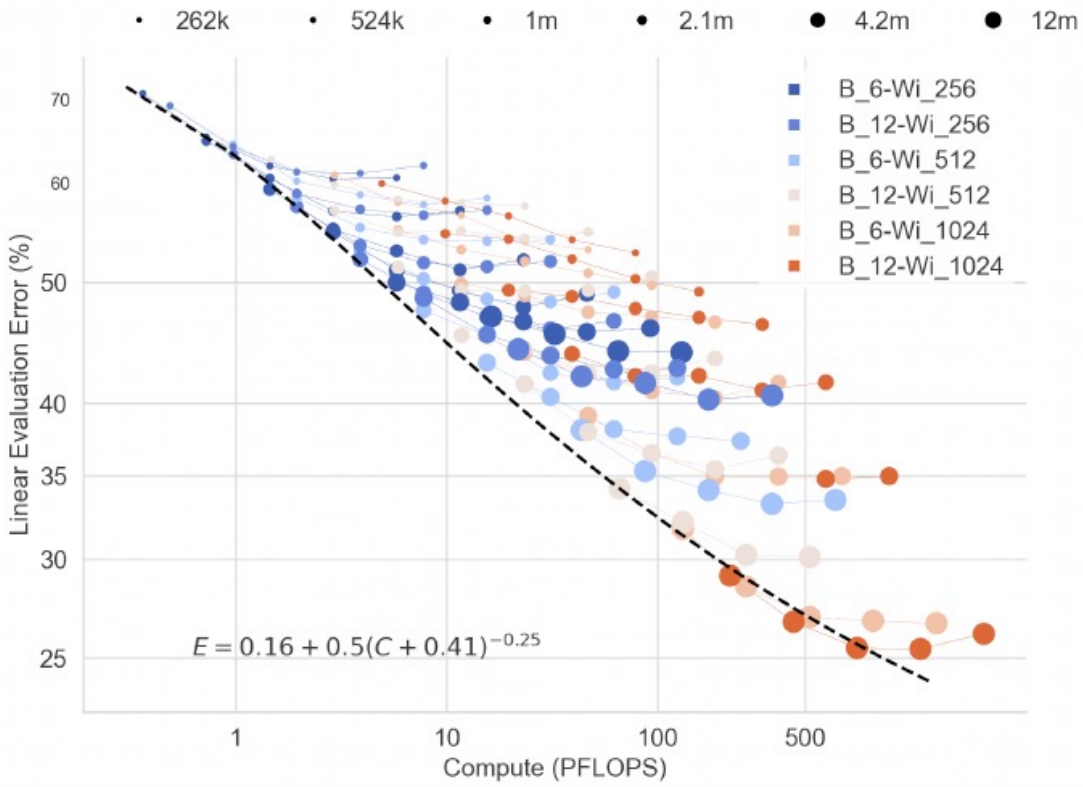
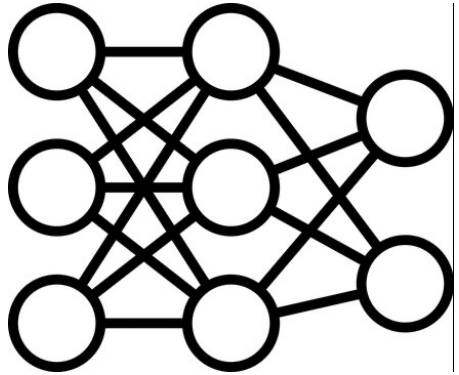


Figure 1: Test error on CIFAR100 as a function of PFLOPS.

Motivation



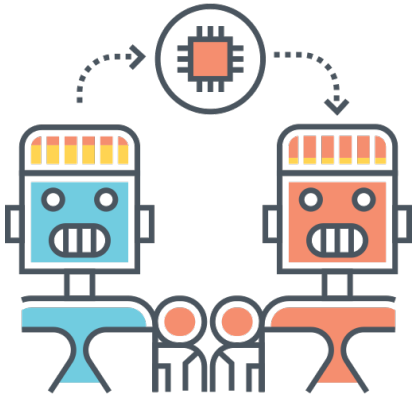
- MLPs are used frequently in formulating theory but their use in the modern settings are limited

- We want to explore if the **MLPs** can perform as well as **CNNs** or **Transformers** in vision tasks with proper scaling

Research question:
"Do MLPs reflect the empirical advances exhibited by practical models?"

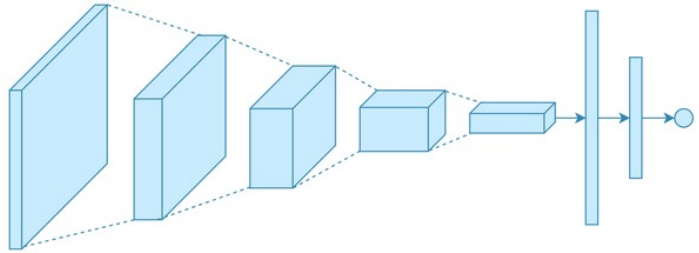
Motivated by the recent narrative:
"At large scales of compute, having less inductive bias is beneficial for performance"

(my intuition: what would you do if you had unlimited compute?)

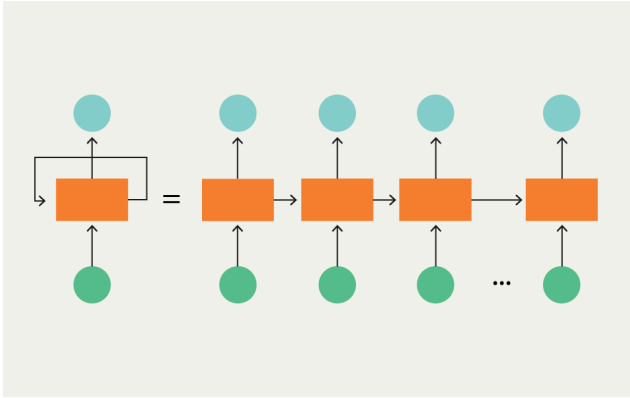


Study designed considering the hypothesis:
"Lack of inductive bias can be compensated by scaling compute."

Background – Inductive Bias

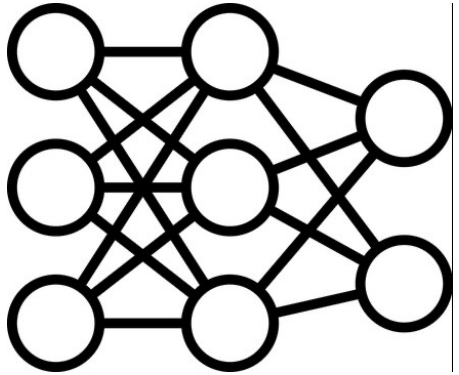


"The **inductive bias** (also known as **learning bias**) of a learning algorithm is the set of assumptions that the learner uses to predict outputs of given inputs that it has not encountered." (Wikipedia)



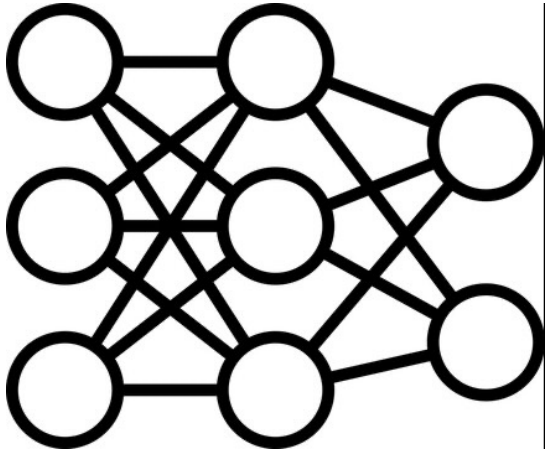
For example:
- The design/architecture of a specific CNN/RNN introduces a bias to the model for specific type of data processing/outcomes

- Goal is to use the right type of inductive bias to make learning easy for the specific task.



Background – Inductive bias of MLPs

- MLPs are not free of inductive biases. They have a hierarchical feature structure.
- But they do not have any vision-specific inductive bias.
- E.g., they do not have properties to use locality properties. A flattened vector of image fed to an MLP is nothing but a collection of numbers, and invariant to any fixed permutation of the pixels.



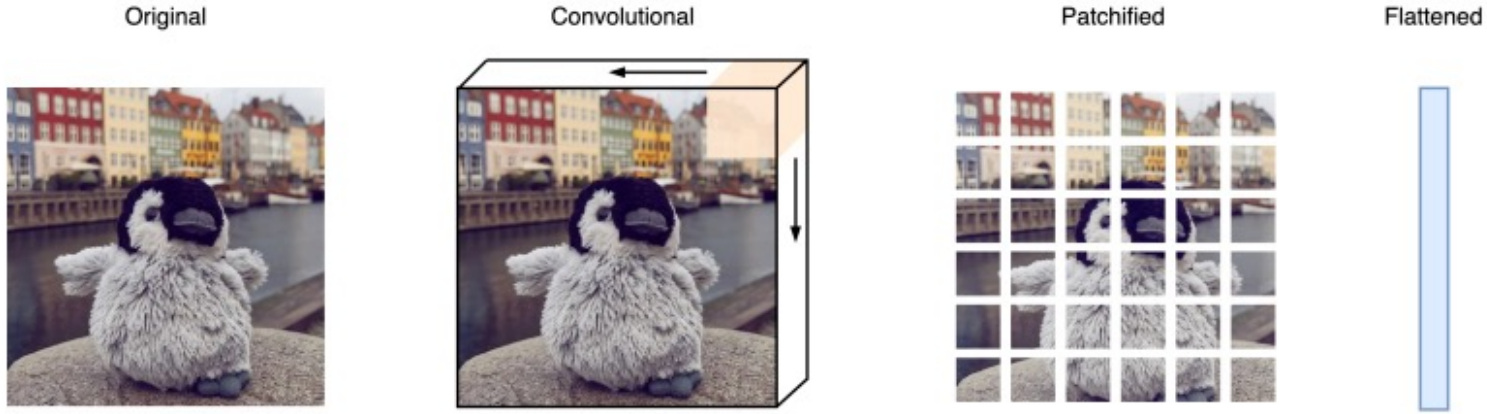


Figure 2: Different architectures process images differently. Convolutions directly operate on the image, ViTs and MLP-Mixers work with patches while the MLP takes the flattened image as input.

Background – MLP and MLP mixers

- MLP mixers work won patches
- MLPs work on pixels

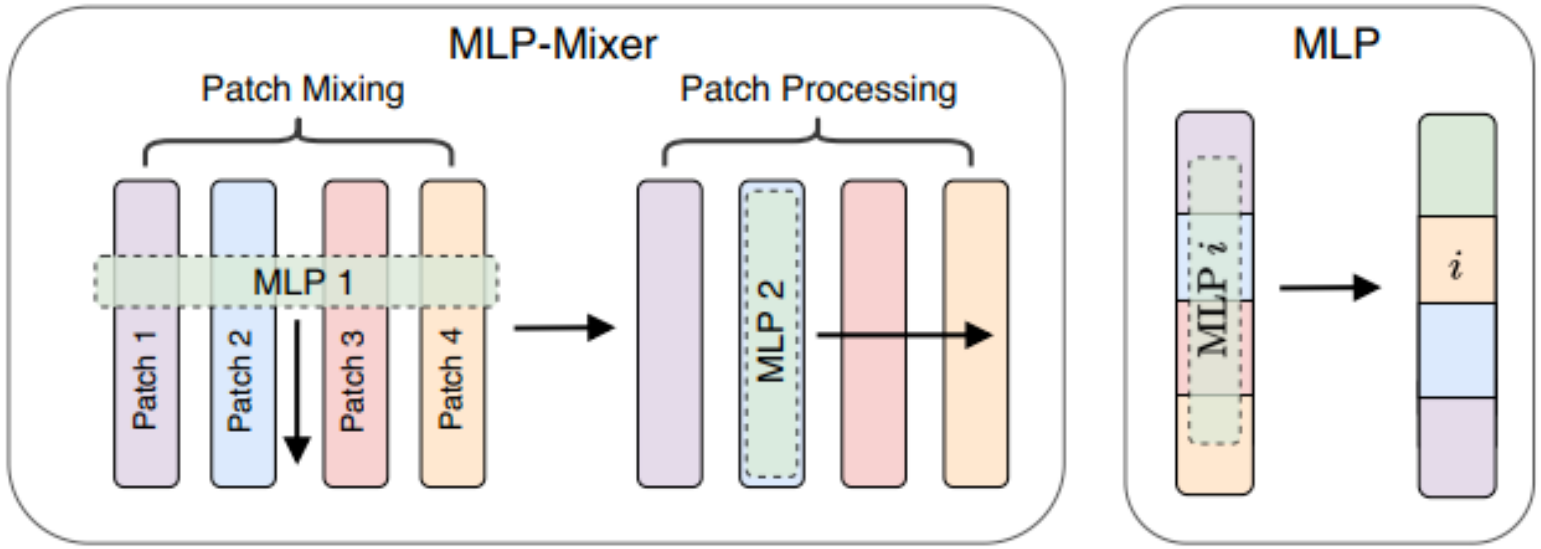
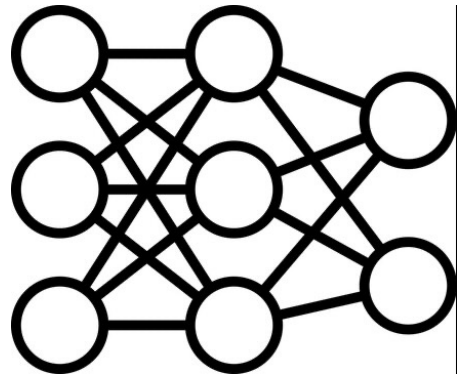


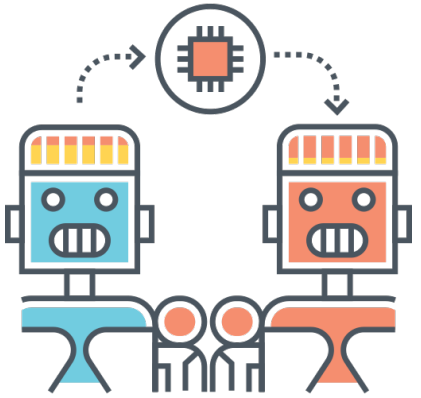
Figure 3: A simplified depiction of the differences between an MLP-Mixer and an MLP.

Contributions

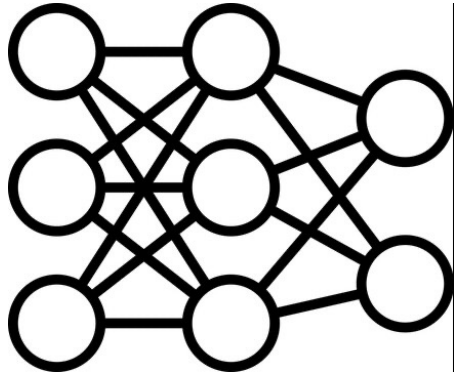
- MLPs are trained in modern settings for the first time.
- MLPs with proper care are comparable to the modern architectures.
- Inductive bias is not crucial at large scales.



- We fill the gap between theory and practice, providing the first results for MLPs trained in modern settings.
- We show that MLPs mostly behave comparably to their modern counterparts, making them a good proxy for theory. We observe however that the roles of regularization and implicit bias of SGD significantly differ and theory hence needs to adapt.
- We provide further evidence that inductive bias is not crucial at large scales, showing that even "bad" architectures like MLPs can achieve strong downstream performance. We however identify a shift in compute-optimality, showing that optimal MLPs invest their compute significantly more into dataset size compared to model size.

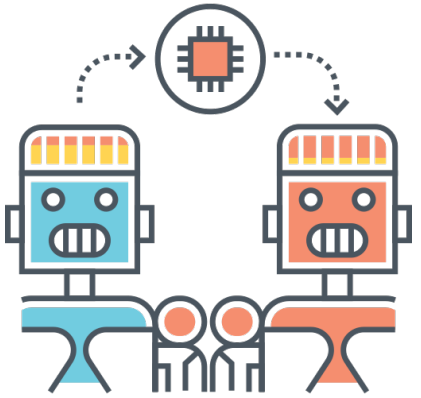


Observations (spoiler alert)



Key observations:

- With pretraining, MLPs can achieve accuracies close to the modern architectures on select datasets
- Size of dataset is more important than the size of the network if you are constrained by compute
- Augmentation may be more important/effective than pretraining



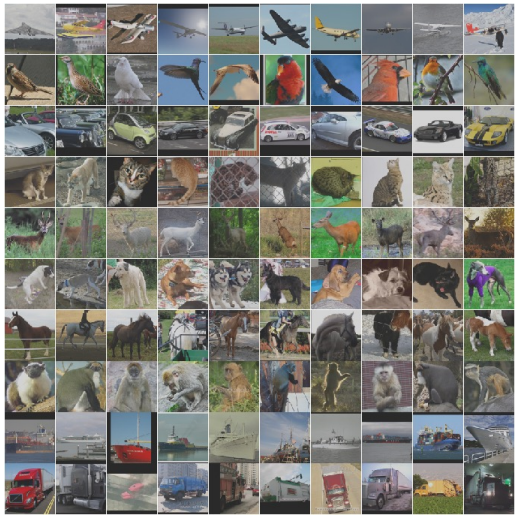
I also have some of my own opinion on this paper.

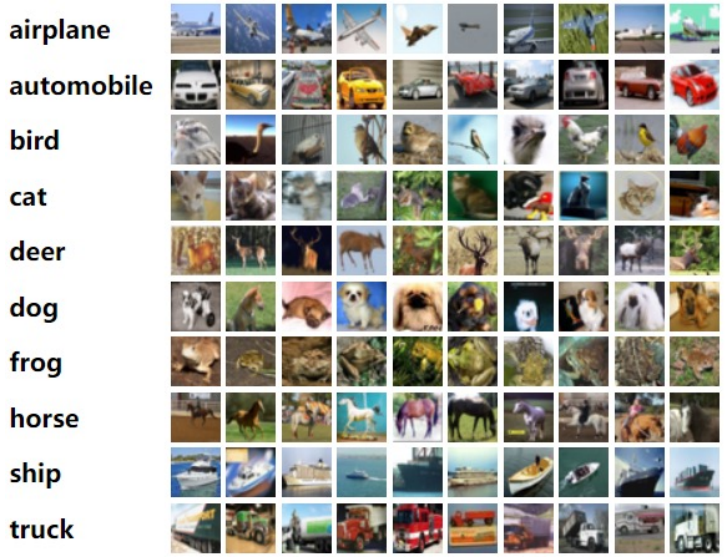
Datasets

ImageNet 1K: ImageNet with 1,000 classes and ~1.2 million images
ImageNet 21k: ImageNet with 21,841 classes and ~14 million images
Tiny ImageNet: 200 classes, 100 thousand images
ImageNet ReaL:

CIFAR10: 10 classes, 60 thousand images
CIFAR100: 100 classes, 60 thousand images

STL10: 10 classes: 5000 training images,





Models

Standard MLP (S-MLP)

Standard MLP. As a first starting point, we investigate simple MLPs with ReLU activations and isotropic design, i.e. except for the first, every layer has the same width $m \in \mathbb{N}$. In order to avoid training instabilities we further enhance the standard MLP with layer normalizations (Ba et al., 2016) placed after the activations. We thus compose several blocks of the form

$$\text{Block}(\boldsymbol{z}) = \sigma\left(\boldsymbol{W}\,\text{LN}(\boldsymbol{z})\right)$$

Bottleneck MLP (B-MLP)
(has skip connections)

Inverted Bottleneck MLP. Inspired by Lin et al. (2015); Tolstikhin et al. (2021) we add a bottleneck structure to an MLP block as well as skip connections as follows:

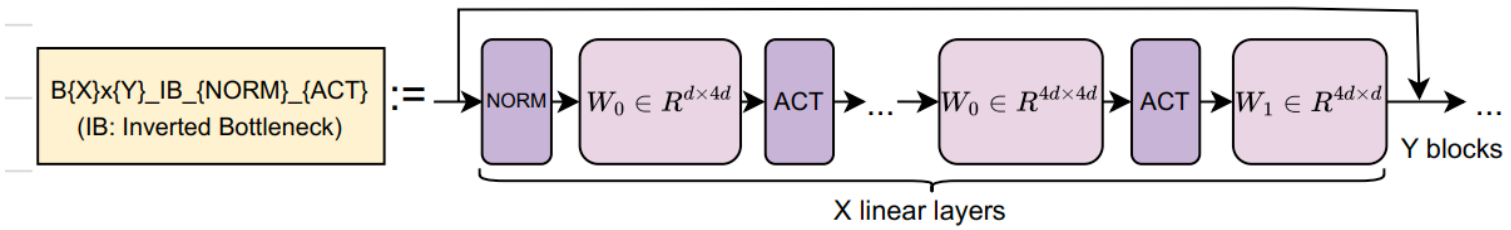
$$\text{Block}(\boldsymbol{z}) = \boldsymbol{z} + \boldsymbol{W}^c\sigma\left(\boldsymbol{W}^e\,\text{LN}\left(\boldsymbol{z}\right)\right)$$

where $\boldsymbol{W}^e \in \mathbb{R}^{km\times m}$ expands the dimension to km for $k \in \mathbb{N}$ and $\boldsymbol{W}^{(c)} \in \mathbb{R}^{m\times km}$ collapses it back to width m. For most experiments we set $k = 4$. While the additions of skip connections and bottleneck layers to the architecture arguably add some amount of inductive bias, we believe that in comparison to modern architectures such enhancements remain negligible. We will denote this variant by *B-MLP*.

LION optimizer is used in the models

The default S-MLP has depth 6 and width 1024

while the default B-MLP has depth 6, width 1024 and an expansion factor of 4.



Models

```python
class StandardMLP(nn.Module):
    def __init__(self, dim_in, dim_out, widths):
        super(StandardMLP, self).__init__()
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.widths = widths
        self.linear_in = nn.Linear(self.dim_in, self.widths[0])
        self.linear_out = nn.Linear(self.widths[-1], self.dim_out)
        self.layers = []
        self.layer_norms = []
        for i in range(len(self.widths) - 1):
            self.layers.append(nn.Linear(self.widths[i], self.widths[i + 1]))
            self.layer_norms.append(nn.LayerNorm(widths[i + 1]))

        self.layers = nn.ModuleList(self.layers)
        self.layernorms = nn.ModuleList(self.layer_norms)

    def forward(self, x):
        z = self.linear_in(x)
        for layer, norm in zip(self.layers, self.layer_norms):
            z = norm(z)
            z = nn.GELU()(z)
            z = layer(z)

        out = self.linear_out(z)

        return out
```

D Inverted Bottleneck MLP Code

We provide PyTorch-style pseudo-code for the inverted bottleneck MLP to highlight its simplicity.

```python
from torch import nn

class Block(nn.Module):
    def __init__(self, dim, expansion_factor=4, dropout=0.):
        super().__init__()
        self.fn = nn.Sequential(
            nn.Linear(dim, int(expansion_factor * dim)),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(int(expansion_factor * dim), dim),
            nn.Dropout(dropout)
        )
        self.ln = nn.LayerNorm(dim)

    def forward(self, x):
        return x + self.fn(self.ln(x))


def MLP(image_size, channels, dim, depth, num_classes,
    expansion_factor=4, dropout=0.):
    return nn.Sequential(
        nn.Flatten(start_dim=1, end_dim=-1),
        nn.Linear(image_size * image_size * channels, dim),
        *[Block(dim, expansion_factor, dropout) for _ in range(depth)
    ],
        nn.Linear(dim, num_classes)
    )
```

First a Linear layer
Then unwraps a number of blocks
Each block has two linear layers and a layer normalization.

Background – LION vs Adam

Symbolic Discovery of Optimization Algorithms

Xiangning Chen[1 2 § *] **Chen Liang**[1 §] **Da Huang**[1] **Esteban Real**[1]

Kaiyuan Wang[1] **Hieu Pham**[1] **Xuanyi Dong**[1] **Thang Luong**[1]

Cho-Jui Hsieh[2] **Yifeng Lu**[1] **Quoc V. Le**[1]

[§]Equal & Core Contribution

[1]Google [2]UCLA

Abstract

We present a method to formulate algorithm discovery as program search, and apply it to discover optimization algorithms for deep neural network training. We leverage efficient search techniques to explore an infinite and sparse program space. To bridge the large generalization gap between proxy and target tasks, we also introduce program selection and simplification strategies. Our method discovers a simple and effective optimization algorithm, **Lion** (*EvoLved Sign Momentum*).

Table 1: Accuracy of BASIC-L [72] on ImageNet and several robustness benchmarks. We apply Lion to both vision tower pre-training and vision-language contrastive training stages. The previous SOTA results on *zero-shot* and *fine-tuning* ImageNet accuracy are 86.3% and 91.0% [100].

Optimizer	Zero-shot						Fine-tune
	ImageNet	V2	A	R	Sketch	ObjectNet	ImageNet
Adafactor	85.7	80.6	85.6	95.7	76.1	82.3	90.9
Lion	**88.3**	**81.2**	**86.4**	**96.8**	**77.2**	**82.9**	**91.1**

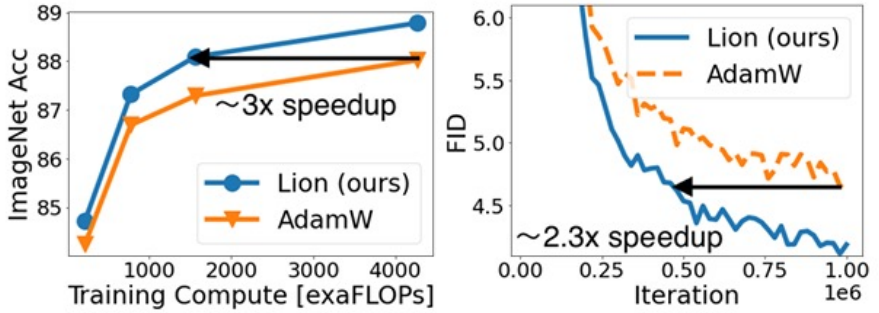


Figure 1: **Left**: ImageNet fine-tuning accuracy vs. pre-training cost of ViT models on JFT-300M. **Right**: FID of the diffusion model on 256^2 image generation. We use DDPM for 1K steps w/o guidance to decode image. As a reference, the FID of ADM is 10.94 [24].

Background – LION vs Adam

Program 1: Discovered optimizer Lion. $\beta_1 = 0.9$ and $\beta_2 = 0.99$ by default are derived from Program 4. It only tracks momentum and uses the sign operation to compute the update. The two gray lines compute the standard decoupled weight decay, where λ is the strength.

```
def train(weight, gradient, momentum, lr):
  update = interp(gradient, momentum, β1)
  update = sign(update)
  momentum = interp(gradient, momentum, β2)
  weight_decay = weight * λ
  update = update + weight_decay
  update = update * lr
  return update, momentum
```

Program 2: An example training loop, where the optimization algorithm that we are searching for is encoded within the `train` function. The main inputs are the weight (`w`), gradient (`g`) and learning rate schedule (`lr`). The main output is the `update` to the weight. `v1` and `v2` are two additional variables for collecting historical information.

```
w = weight_initialize()
v1 = zero_initialize()
v2 = zero_initialize()
for i in range(num_train_steps):
  lr = learning_rate_schedule(i)
  g = compute_gradient(w, get_batch(i))
  update, v1, v2 = train(w, g, v1, v2, lr)
  w = w - update
```

Program 3: Initial program (AdamW). The bias correction and ϵ are omitted for simplicity.

```
def train(w, g, m, v, lr):
  g2 = square(g)
  m = interp(g, m, 0.9)
  v = interp(g2, v, 0.999)
  sqrt_v = sqrt(v)
  update = m / sqrt_v
  wd = w * 0.01
  update = update + wd
  lr = lr * 0.001
  update = update * lr
  return update, m, v
```

Program 4: Discovered program after search, selection and removing redundancies in the raw Program 8 (Appendix). Some variables are renamed for clarity.

```
def train(w, g, m, v, lr):
  g = clip(g, lr)
  g = arcsin(g)
  m = interp(g, v, 0.899)
  m2 = m * m
  v = interp(g, m, 1.109)
  abs_m = sqrt(m2)
  update = m / abs_m
  wd = w * 0.4602
  update = update + wd
  lr = lr * 0.0002
  m = cosh(update)
  update = update * lr
  return update, m, v
```

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Algorithm 2 Adam and AdamW

given $\alpha_t = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow SelectBatch(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w_t \boldsymbol{x}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ here, β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ here, β_2 is taken to the power of t
  $\eta_t \leftarrow SetScheduleMultiplier(t)$ ▷ can be fixed, decay, be used for warm restarts
  $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha_t \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + w_t \boldsymbol{x}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{x}_t$

Experimental results – training from scratch

S-MLP = Standard MLP
B-MLP = Inverted Bottleneck MLP

E = Epochs
DA = Data augmentation

	CIFAR10	CIFAR100	TINYIMAGENET	IMAGENET
S-MLP (@100 E)	54.2	28.8	8.5	9.2
S-MLP + DA (@ 1000 E)	68.9	43.3	25.2	24.3
S-MLP + DA (@ 5000 E)	72.3	44.5	27.3	26.8
B-MLP (@ 100 E)	58.1	30.5	8.9	8.7
B-MLP + DA (@1000 E)	70.1	48.3	27.2	28.7
B-MLP + DA (@5000 E)	75.4	50.4	31.2	31.7
RESNET18[2] + DA	93.2	75.6	68.9	69.7

Table 1: Test accuracies (in %) without any pre-training. The S-MLP has depth 6 and width 1024 while the B-MLP has depth 6, width 1024 and an expansion factor of 4.

Experimental results – fine tuned models

S-MLP = Standard MLP
B-MLP = Inverted Bottleneck MLP

TTA = Test time augmentation

All models on this table are Bottleneck MLPs.

Depth = 6 or 12
Width = 1024
Expansion factor = 4 (default)

	CIFAR10	CIFAR100	STL10	TINY-IN	IN	REAL
B-6/Wi-1024	69.9±0.1	43.0±0.4	51.5±0.1	47.1±0.1	15.2±0.2	20.3±0.2
B-6/Wi-1024 + DA	91.5±0.02	76.4±0.2	85.0±0.2	62.7±0.1	38.7±0.1	47.0±0.15
B-12/Wi-1024 + DA	94.2±0.05	80.0±0.05	89.9±0.1	69.9±0.4	43.3±0.06	48.6±0.2
B-12/Wi-1024 + DA + TTA	95.5±0.05	82.6±0.2	92.2±0.05	73.1±0.5	51.5±0.1	57.9±0.1

Table 2: Fine-tuning Top-1 accuracies (in %) when pretrained on ImageNet21k. Accuracies are averaged over 3 runs. For readability, we abbreviate ImageNet as IN.

Role of augmentations. Data augmentation is very pronounced for MLPs, providing indirect inductive bias to the model. Remarkably, a model pre-trained on 12 million examples without data augmentation shows inferior performance on CIFAR10 compared to a network trained from scratch with augmentations turned on. This emphasizes that augmentations go beyond merely leading to a bigger dataset but provide the model with useful invariances. We investigate the learnt weights in

Discussion – Test time augmentation and Multiclass labels

With TTA, performance improves.

With Multiclass labels, performance improves.

It suggests that MLPs may be weak at localizing the object of interest (weak inductive bias). Hence, scaling is essential.

Test-Time Augmentations. For ImageNet1k we further notice that objects tend to not be centered, in contrast to datasets like CIFAR10. We suspect that this might lead to the comparatively weaker performance. To test this, we leverage test-time augmentations (TTA). As introduced by Krizhevsky et al. (2012), for each test image, we produce a fixed number of 100 random crops and use the averaged logits for prediction. We observe significant improvements across all datasets, especially for ImageNet we obtain an increase of roughly 8%. This indeed indicates that MLPs struggle to localize the object of interest, especially for the more complicated ImageNet1k task. Using a large number of crops alleviates this problem to some degree. This also explains why the gains on tasks like CIFAR10 are smaller as the objects there usually are perfectly centered.

ReaL accuary. As observed in (Beyer et al., 2020), the ImageNet labels do not capture that a single image might contain multiple objects of distinct classes. ImageNet accuracy can thus be misleading in the sense that model classes such as convolutional networks might have implicitly adapted to the particular labeling strategy due to the repeated benchmarking on the same validation set. MLPs most likely lack such an implicit adaptation as this work is to our knowledge the first to evaluate them on ImageNet1k. To address this, Beyer et al. (2020) introduced a novel set of validation labels that better capture the multi-label nature, where a prediction is deemed correct if it matches one of the categories present in the image. We observe further very significant improvements of $\approx 7\%$ when employing ImageNet ReaL.

Overall, these results underline that a bad inductive bias as exhibited by an MLP can indeed be overcome if subjected to enough scale. For theory, the results are double-edged; while MLPs prove to be a good proxy to understand transfer learning, data augmentation proves to be a crucial component. Also test-time augmentations significantly boost performance. Both these components on the other hand remain rather understudied in theoretical works.

Which direction to scale? Data or Model?

Empirical observations suggest that for MLPs, optimal strategy is to invest more compute into dataset size.

Parameters or examples. Given a fixed level of compute C, what is the optimal way to allocate it to parameter count P and number of examples N? To be more comparable to previous work, we assume a fixed training time $T = 50$. To answer this question, we follow the approach outlined in Hoffmann et al. (2022) and plot the optimal compute models identified in Fig. 1 both against model size P and number of examples N and visualize the results in Fig. 7. We empirically observe that the optimal parameter count $P^*(C)$ and dataset size $N^*(C)$ as a function of compute C exhibit power-law behaviour of the approximate form

$$P^*(C) \propto C^{0.35} \qquad N^*(C) \propto C^{0.65}$$

While for transformers, the number of examples (or tokens) N and parameters P are scaled equally (Hoffmann et al., 2022) (i.e. $\alpha_P \approx \alpha_N \approx 0.5$), in contrast we observe that the optimal strategy for MLPs invests significantly more compute into dataset size N. This is further evidence for the weaker inductive bias present in MLPs, which needs more examples in order to be compensated for.

Experimental results – Optimal model and data size

Examples can be increased comparatively faster than the model size given the number of compute.

Bottleneck MLPs

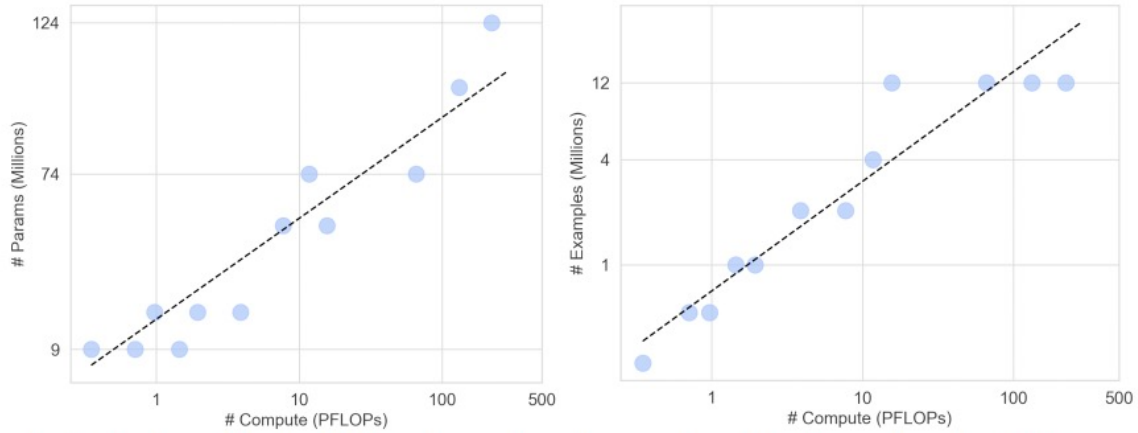


Figure 7: Optimal model size (left) and number of examples (right) for a given level of compute for linear evaluation on CIFAR100, on a log-log scale.

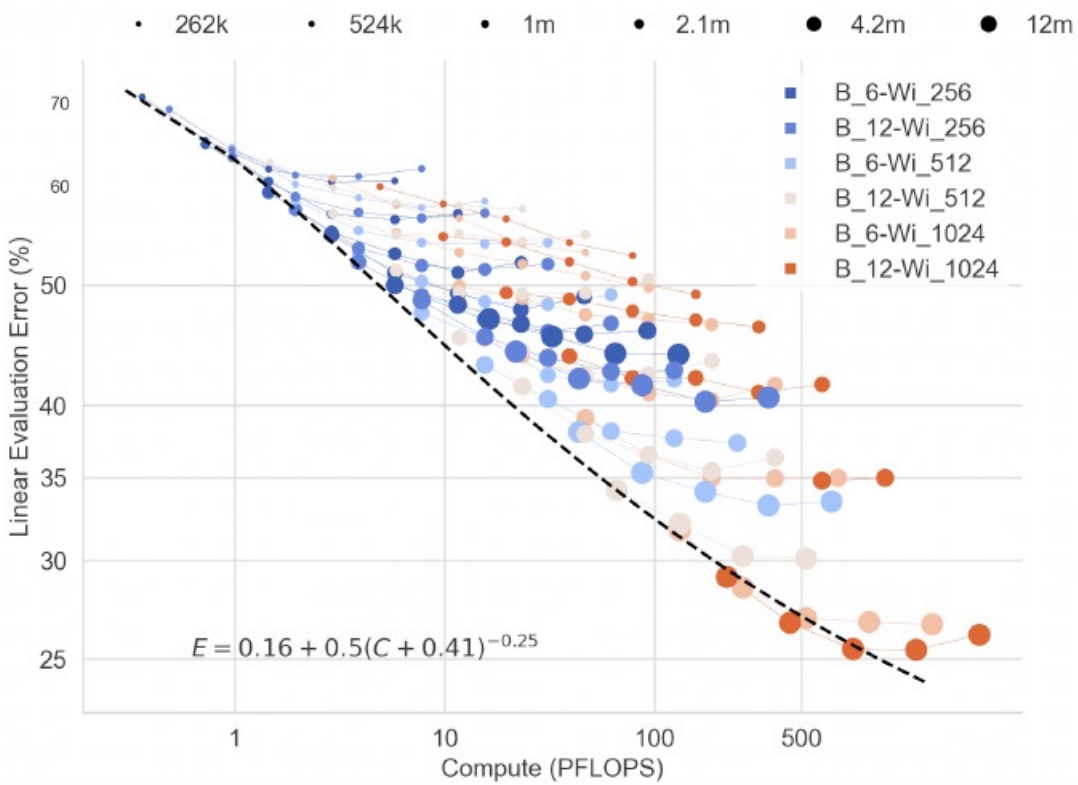


Figure 1: Test error on CIFAR100 as a function of PFLOPS.

B_<depth>_Wi_<width>

My opinion regarding this paper

Strengths:
- Informative background discussion
- Comparison of optimal dataset size vs optimal model size for fixed compute
- Nicely explained the effect of pretraining and role of augmentation
- Using test time augmentation to explain the model's limitations

Weaknesses:
- Poor design of the figures.
- Unclear abbreviations on figures and tables. In my opinion, figures and tables need to be self-sufficient in terms of describing uncommon abbreviations/jargons.

What if I was a reviewer?
- I would recommend borderline and change to acceptance upon addressing the weaknesses.

What were the real reviews?

<u>4 reviewers</u>
1 strong accept (rating 8)
2 weak accepts (rating 6)
1 borderline reject (rating 4)

https://abdullah-mamun.com
a.mamun@asu.edu