Vision Transformers Tutorial

https://github.com/eemlcommunity/PracticalSessions2021/tree/main/vision

Andreas Steiner

EEML summer school
July 8th 2021, Budapest (virtually)

Google Research

Outline

1. Flax models
   - ⇢ Practical part
2. Vision Transformer in Flax
   - ⇢ Practical part
3. Exploring pre-trained ViTs
   - ⇢ Practical part

Flax Models

And a little bit about JAX

Google Research

Researchers 😻 JAX

Sam Schoenholz

I would say that JAX has significantly (~5x-10x) improved my productivity and has enabled **research that would have been extremely difficult, if not impossible, with existing tools**. [...] Allowed me to write a molecular dynamics simulation in numpy [...] A first pass at the code took an afternoon to write.

Vikas Sindhwani

JAX is very compelling for trajectory optimization using Iterative LQR whose key bottleneck is linearization and solving LQR problems: we need fast gradients, Jacobians and Hessians. The **benchmark** is on small but realistic problem dimensions, where JAX turns out to be **competitive with a third party C/C++ QP solver**.

David Sussillo

It took me only a weekend to reimplement my LFADS sequential autoencoder algorithm. It's running on GPU, using vmap and jit'd. [...] The ease is amazing. [...] The **JAX model is easy for my neuroscience audience to understand**, to deploy, and to innovate upon.

Elliot Creager

[Differentially-private] SGD requires clipping the per-example parameter gradients, which is non-trivial to implement efficiently. [...] We are able to reproduce the MNIST results in the TensorFlow reference implementation at a **30X speedup** for the simple convolutional architecture used in the original DPSGD paper.

JAX is Numpy

```
import numpy as np
x = np.arange(9_000_000,
              dtype=np.float32)
x = x.reshape([3000, 3000])
x
```

```
array([[0.000000e+00, ...],
       ...,
       [..., 8.999999e+06]])
```

```
%%timeit
(x @ x)
```

```
1 loop, best of 5: 428 ms per loop
```

```
import jax.numpy as jnp
x = jnp.arange(9_000_000,
               dtype=jnp.float32)
x = x.reshape([3000, 3000])
x
```

```
DeviceArray([[0.000000e+00, ...],
             ...,
             [..., 8.999999e+06]])
```

```
%%timeit
(x @ x).block_until_ready()
```

```
1 loop, best of 5: 11.9 ms per loop
```

grad : Compute gradients

```
def loss_fn(weights, inputs, targets):
  outputs = model(weights, inputs)
  return ((targets - outputs)**2).mean()

def update_step(weights, inputs, targets):
  grads = jax.grad(loss_fn)(
      weights, inputs, targets)
  weights -= 0.1 * grads
  return weights

weights = init()
for inputs, targets in data:
  weights = update_step(
      weights, input, targets)
```

`grad()` is a *function transformation.*

Multiple signatures:

```
loss, grads = jax.value_and_grad(loss_fn)(
    weights, inputs, targets)
```

```
(loss, logits), grads = jax.value_and_grad(
    loss_fn, has_aux=True)(
    weights, inputs, targets)
```

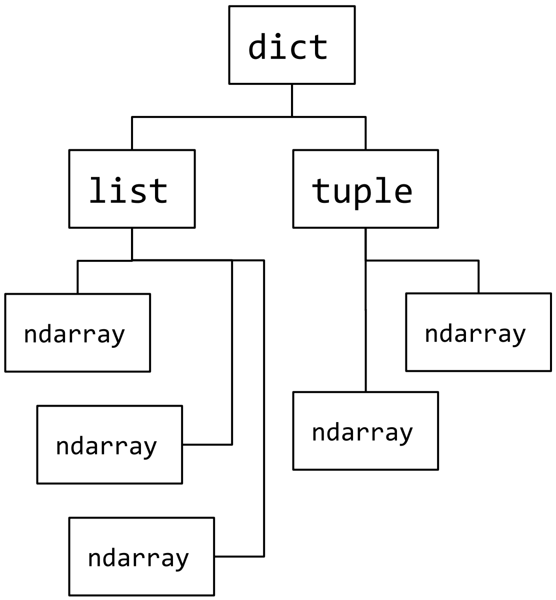
JAX-Arguments : Pytrees

https://jax.readthedocs.io/en/latest/pytrees.html

- Most JAX functions operate over pytrees of `jax.ndarray` (=leafs)
- Useful functions :

  `jax.tree_util.tree_[un]flatten()`

  `jax.tree_[multi]map()`
- Flax extends pytrees to `@flax.struct.dataclass`



Neural nets in JAX

- Unlike stateful framework (e.g. PyTorch)
  Separate computation & variables
- Functional solution : provide `init()` & matching `apply()`
  for *every module*
- OMG lots of code 😱

Flax to the rescue

- Opinionated `nn.Module` abstraction
- "init" & "apply" possible in a single function

Simple PyTorch NN

```
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):
 def __init__(self):
   super(Net, self).__init__()
   self.conv = nn.Conv2d(1, 16, 3, 3)
   self.fc = nn.Linear(1296, 10)
 def forward(self, x):
   x = self.conv(x)
   x = F.relu(x)
   x = torch.flatten(x, 1)
   x = self.fc(x)
   return F.log_softmax(x, dim=1)
```

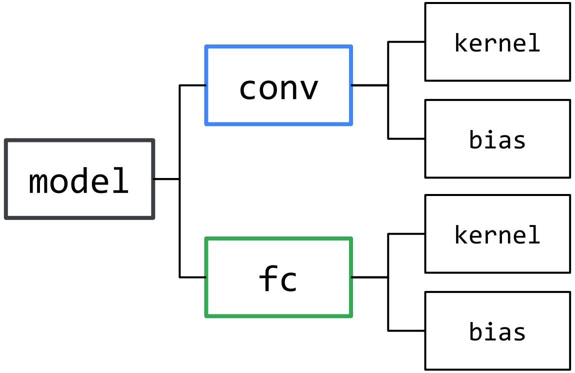
```
model = Net()
optimizer = optim.Adam(model.parameters(), lr=0.1)
optimizer.zero_grad()
outputs = model(inputs)
loss = F.nll_loss(outputs, labels)
loss.backward()
optimizer.step()
```

- Model: setup/forward phase with shape-dependent values
- Dataflow is not clear for the uninitiated (model↔loss↔optimizer)

Flax "Linen" modules

```python
import flax.linen as nn
class Net(nn.Module):
  units: int
  def setup(self):
    self.conv = nn.Conv(
        self.units, [3, 3], [3, 3])
    self.fc = nn.Dense(10)
  def __call__(self, x, *, train):
    x = self.conv(x)
    x = nn.relu(x)
    x = x.reshape([len(x), -1])
    x = self.fc(x)
    return nn.log_softmax(x)
```

- *Immutable* dataclasses.
- Predefined modules like `nn.Conv()`
- Auto shape inference.
- Automatically creates pytree(s) that reflects module hierarchy:



Linen modules - compact 😎

Avoids replicating logic in `__call__()` and `setup()`.

```
class Net(nn.Module):
  units: int
  @nn.compact
  def __call__(self, x):
    x = nn.Conv(self.units, [3, 3], [3, 3])(x)
    x = nn.relu(x)
    x = x.reshape([len(x), -1])
    x = nn.Dense(10)(x)
    return nn.log_softmax(x)
```

Using Linen modules

```
model = Model(num_classes=num_classes)
variables = model.init(key, inputs)
```

1. Instantiate module
2. Create initial variables

```
outputs = model.apply(variables, inputs)
```

No mutable state:

$$f(v_{in}, x) \rightarrow y$$

Practical Part

https://github.com/eemlcommunity/PracticalSessions2021/tree/main/vision

(Load data)

Define a model

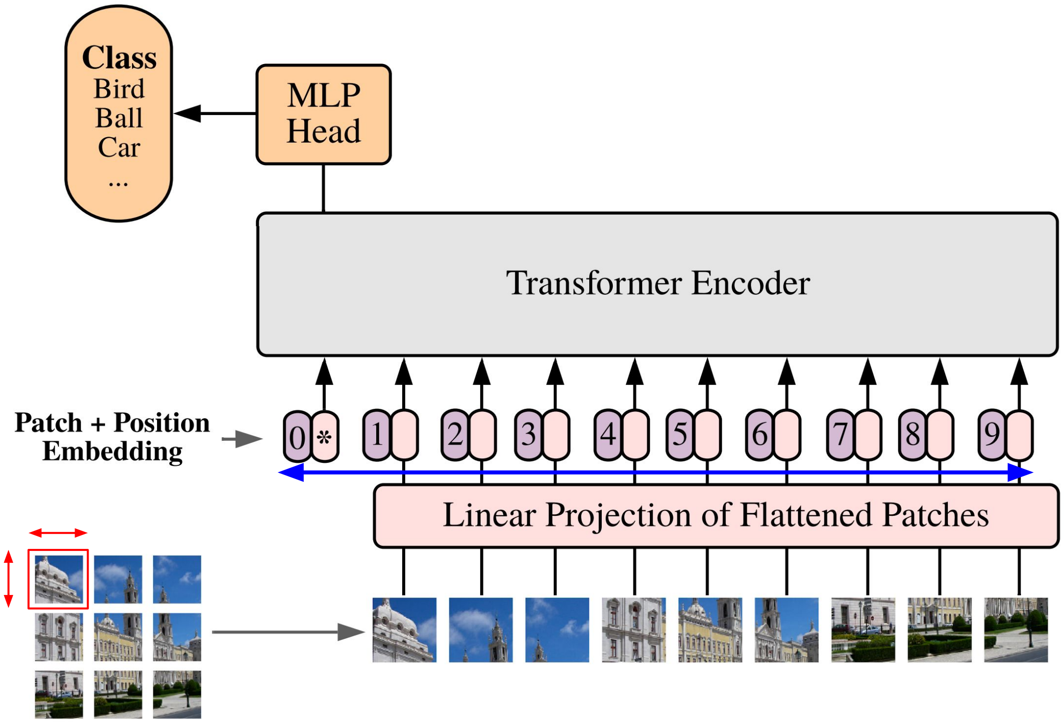
Create and use model weights

Train, evaluate + predict

Google Research

Vision Transformers in Flax

Vision Transformer (ViT)



Idea: Take a transformer and apply it directly to image patches

patch_size
e.g. B/32

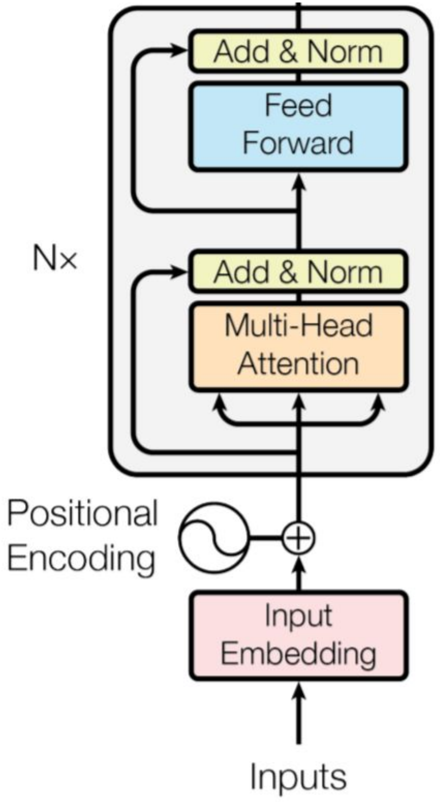
sequence_length
depends on image size

Cordonnier et al., *On the Relationship between Self-Attention and Convolutional Layers*, ICLR 2020

Dosovitskiy et al., *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, ICLR 2021

Transformer

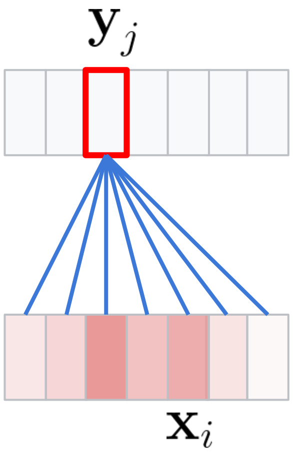
- Transformer “encoder”
  - A stack of alternating self-attention and MLP blocks
  - Residuals and LayerNorm
- Transformer “decoder” (not shown)
  - A slightly more involved architecture useful when the output space is different from the input space (e.g. translation)



[Vaswani et al., *Attention Is All You Need*, NeurIPS 2017]

Self-attention

- Each of the tokens (=vectors) attends to all tokens
  - Extra tricks: learned key, query, and value projections, inverse-sqrt scaling in the softmax, and multi-headed attention (omit for simplicity)
- It’s a set operation (permutation-invariant)
  - ...and hence need “position embeddings” to “remember” the spatial structure
- It’s a global operation
  - Aggregates information from all tokens



$$\boldsymbol{\alpha}_j = softmax\left(\frac{K\mathbf{x}_1 \cdot Q\mathbf{x}_j}{\sqrt{d_K}}, \ldots, \frac{K\mathbf{x}_n \cdot Q\mathbf{x}_j}{\sqrt{d_K}}\right)$$

$$\mathbf{y}_j = \sum_{i=1}^{n} \alpha_{ji} V\mathbf{x}_i$$

Simplified! Multi-headed attention not shown

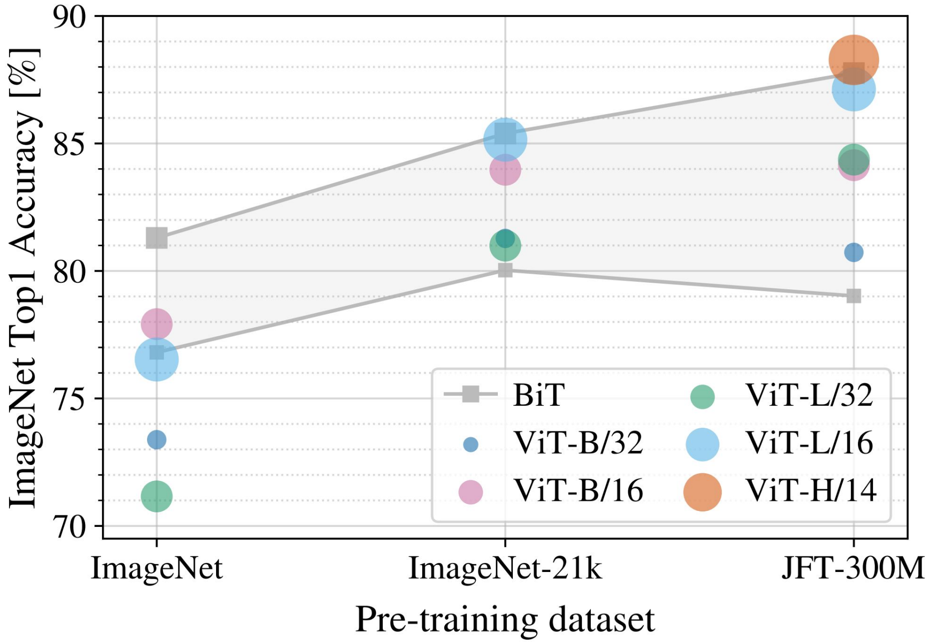
Scaling with Data

<u>Key</u>
ViT = Vision Transformer
BiT = Big Transfer (~ResNet)

ViT overfits on ImageNet, but shines on larger datasets.

* with heavy regularization ViT has been shown to also work on ImageNet (Touvron et al.)

** training ViT on ImageNet with the sharpness-aware minimizer (SAM) also works very well (Chen et al.)



Dosovitskiy et al., *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, ICLR 2021
Xiangning Chen et al., *When Vision Transformers Outperform ResNets without Pretraining or Strong Data Augmentations*, arXiv 2021
Touvron et al., *Training data-efficient image transformers & distillation through attention*, arXiv 2020

Slide by Alexey Dosovitskiy

Practical Part

https://github.com/eemlcommunity/PracticalSessions2021/tree/main/vision

Forming patches + position embeddings

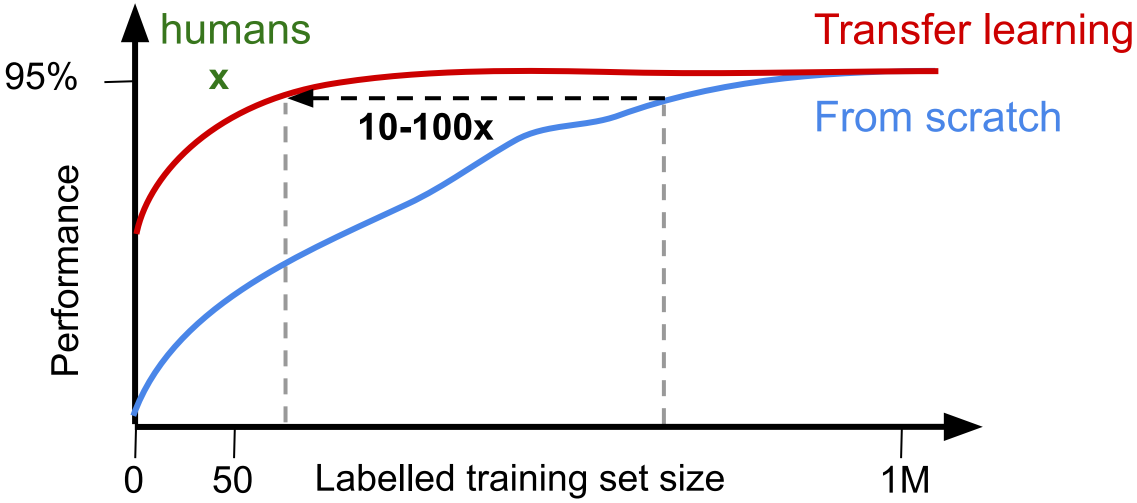
TransformerLayer + Transformer

Understanding ViT params

Refined training

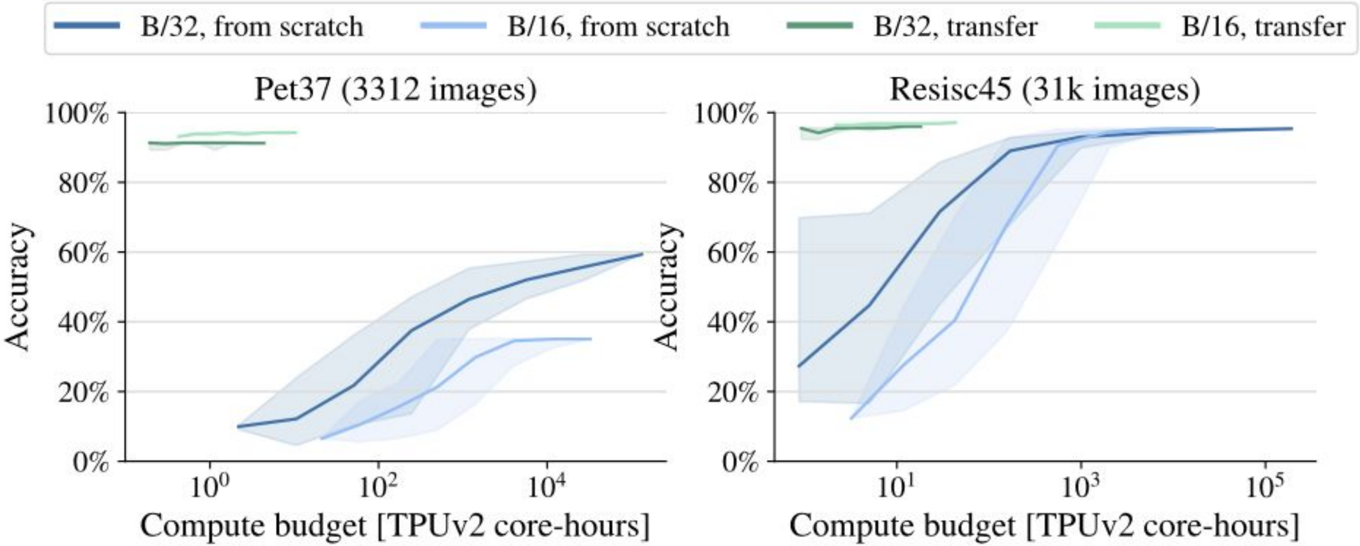
Exploring pre-trained ViTs

Why transfer?



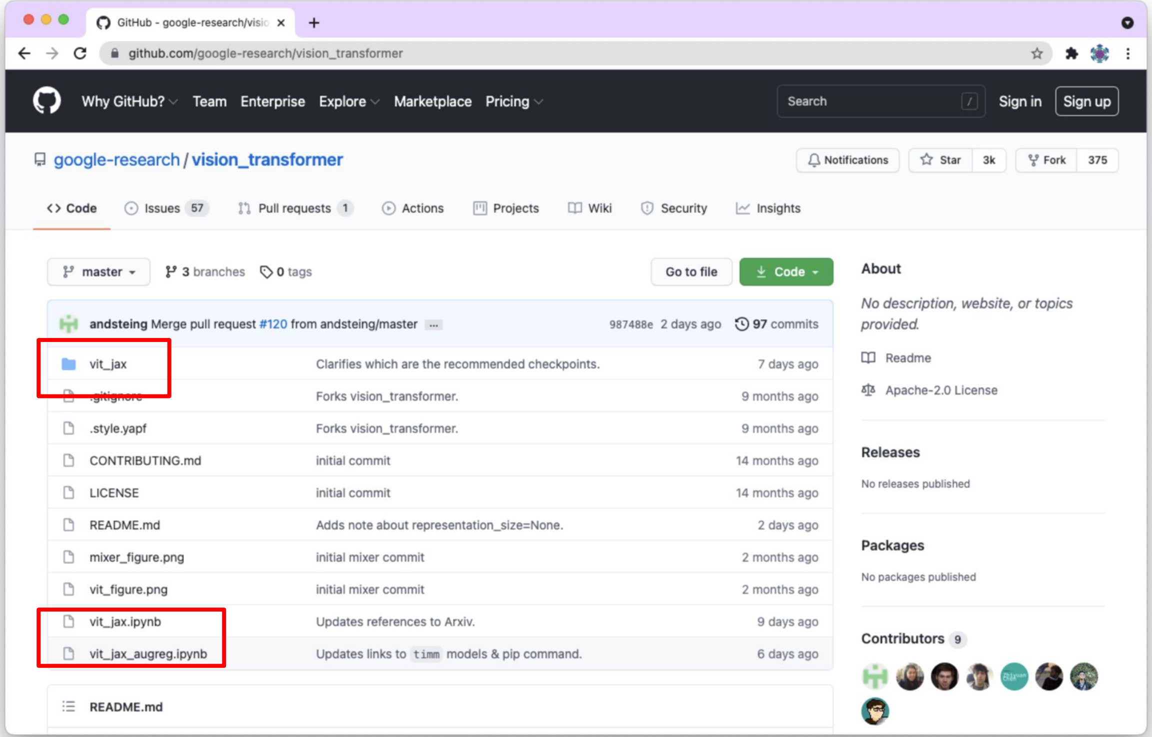
Slide by Neil Houlsby

Why transfer?



Steiner et al., *How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers*, ArXiV 2021

vision_transformer Github repo



>50k Pre-trained checkpoints

```python
# Upstream AugReg parameters (section 3.3):
(
df.groupby(['ds', 'name', 'wd', 'do', 'sd', 'aug']).filename
  .count().unstack().unstack().unstack()
  .dropna(1, 'all').astype(int)
  .iloc[:7]  # Just show beginning of a long table.
)
```

		aug	light0		light1		medium1		medium2		none		strong1		strong2	
		sd	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1
		do	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1
ds	name	wd														
i1k	B/16	0.03	68	68	68	68	68	68	68	68	68	68	68	68	68	68
		0.10	68	68	68	68	68	68	68	68	68	68	68	68	68	68
	B/32	0.03	68	68	68	68	68	68	68	68	68	68	68	68	68	68
		0.10	68	68	68	68	68	68	68	68	68	68	68	68	68	68
	L/16	0.03	68	68	68	68	68	68	68	68	68	68	68	68	68	68
		0.10	68	68	68	68	68	68	68	68	68	68	68	68	68	68
	R+Ti/16	0.03	68	68	68	68	68	68	68	68	68	68	68	68	68	68

Practical Part

https://github.com/eemlcommunity/PracticalSessions2021/tree/main/vision

Use repository code in Colab

Explore checkpoints

Load checkpoints + inference

Fine-tune checkpoints