

Image classification and Neural Networks

TRack Module II

Author: Ekaterina Golubeva

Supervisors: Prof. Dr. Thomas Ott, Dr. Stefan Glüge

Institution: ZHAW

Date: Spring semester 2023

Contents

[1. Milestones of Track module 2 2](#_Toc138173399)

[2. Technical Preparation 2](#_Toc138173400)

[Access to the HPC cluster 3](#_Toc138173401)

[Cyberduck 4](#_Toc138173402)

[Visual Studio Code 5](#_Toc138173403)

[Pytorch 6](#_Toc138173404)

[Tensorboard 6](#_Toc138173405)

[3. Neural Networks Data Preparation and Model tuning 7](#_Toc138173406)

[Data augmentation 7](#_Toc138173407)

[Effect of batch size on training dynamics 7](#_Toc138173408)

[Learning rate schedulers 8](#_Toc138173409)

[Checkpoints 14](#_Toc138173410)

[Model evaluation: 15](#_Toc138173411)

[4. Transfer Learning 15](#_Toc138173412)

[5. Visual Transformers 18](#_Toc138173413)

[6. Neural Architecture Search 23](#_Toc138173414)

# Milestones of Track module 2

* Get used to the environment of VSC for running and debugging code
* Get used to using GPU for training models
* Understand transfer learning and use a pretrained model
* Apply Transfer learning, Data augmentation, CNN learning to a simple Malarial cells classification problem
* Use Tensorboard to run experiments, store, visualize results and compare architectures
* Learn hyperparameters tuning for NNs : batch size, learning rate, lr scheduler
* Get introduces to Neural Architecture search (Automatic architecture search) and trying to build more complex architectures
* Try more complex architectures with Hyperparameters Optimization and Neural Architecture Search
* Master thesis topic. First draft : *Application of a pretrained ViT on blood cell dataset classification problem and comparison with simpler approaches*

# Technical Preparation

## 2.1 Access to the HPC cluster

Connect to VPN <https://servicedesk.zhaw.ch/tas/public/login/saml>

Access to GPU guidelines <https://github.zhaw.ch/glue/icls-gpu-node/blob/main/usage.md#get-an-account>

Run the script, first activate the environment :

source Trackmodule2/TM2-env/bin/activate

From the command line, run address of the cluster

ssh [golubeka@160.85.79.231](mailto:golubeka@160.85.79.231)

Command to check the state of the GPU’s usage : nvidia-smi

Choose the device : export CUDA\_VISIBLE\_DEVICES=1

Use the most powerful GPU (the 5th), otherwise use anyone which is not used at the moment.

Text

Description automatically generated with medium confidence

## 2.2 Cyberduck

Cyberduck is an open-source client for FTP and SFTP, WebDAV, and cloud storage available for macOS and Windows (as of version 4.0) licensed under the GPL. It supports FTP/TLS, using AUTH TLS as well as directory synchronization. The user interacts with the user interface (GUI), including file transfer by drag and drop and notifications via Growl. This tool allows me to interact with the HPC cluster, access, upload and download files.

A screenshot of a computer

Description automatically generated with medium confidence

Cyberduck : Open Connexion > Server [golubeka@160.85.79.231](mailto:golubeka@160.85.79.231)

Whenever I need to access a file that is on the HPC cluster, I copy its directory from Cyberduck.

## 2.3 Visual Studio Code

Python in Visual Studio Code tutorial <https://code.visualstudio.com/docs/languages/python>

Virtual environments cheat sheet <https://aaronlelevier.github.io/virtualenv-cheatsheet/>

If I am training a model and it takes a lot of time, I should open a permanent session, so I don’t risk losing everything once I’m closing the window. For this reason, we use Terminal Multiplexor Tmux. It allows us to open multiple windows (using keybord path ctrl + B + n).

Tmux session cheat sheet <https://tmuxcheatsheet.com> <https://www.ocf.berkeley.edu/~ckuehl/tmux/>

Conda cheat sheet <https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf>

Access GPU via VSC : open new window, choose HPC address. To access my host session via VSC, I am required to enter the password.

A screenshot of a computer

Description automatically generated with medium confidence

## 2.4 Pytorch vs Tensorflow

PyTorch is a fully featured framework for building deep learning models, which is a type of machine learning that's commonly used in applications like image recognition and language processing.

TensorFlow is an end-to-end open-source deep learning framework developed by Google and released in 2015. It is known for documentation and training support, scalable production and deployment options, multiple abstraction levels, and support for different platforms, such as Android.

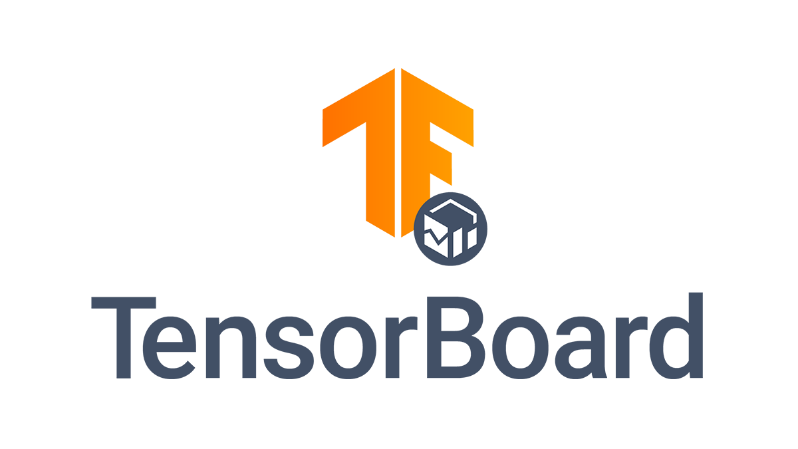
TensorFlow is a symbolic math library used for neural networks and is best suited for dataflow programming across a range of tasks. It offers multiple abstraction levels for building and training models.

| **Comparison** | **PyTorch** | **TensorFlow** |
| --- | --- | --- |
| API Level | Low | High and Low | |
| Architecture | Complex, less readable | Not easy to use | |
| Datasets | Large datasets, high performance | Large datasets, high performance | |
| Debugging | Good debugging capabilities | Difficult to conduct debugging | |
| Does It Have Trained Models? | Yes | Yes | |
| Popularity | Third most popular | Second most popular | |
| Speed | Fast, high-performance | Fast, high-performance | |
| Written In | Lua | C++, CUDA, Python | |

## 2.5 Tensorboard

A screenshot of a graph

Description automatically generated with medium confidence

[](http://localhost:6010/?darkMode=true&smoothing=0.99#timeseries&_smoothingWeight=0&runSelectionState=eyJFeHBlcmltZW50X21hbGFyaWFsX1Jlc25ldCI6ZmFsc2UsIkV4cGVyaW1lbnRfbWFsYXJpYWxfUmVzbmV0X211bHRpc3RlcExSIjpmYWxzZX0%3D)

TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics like loss and accuracy, visualizing the model graph, and much more.

I use Tensorboard **to run experiments, store and visualize results.**

After installing, I write

tensorboard --logdir=runs

in the terminal and open Tensorboard in the browser.

To initialize Tensorboard writer :

from torch.utils.tensorboard import SummaryWriter

# Writer will output to ./runs/ directory by default

writer = SummaryWriter(act\_result\_path)

To save scalars during training :

writer.add\_scalar('training loss', epoch\_loss, epoch)

writer.add\_scalar('train Accuracy', epoch\_acc, epoch)

To see images :

img\_grid = torchvision.utils.make\_grid(image)

writer.add\_image('ResNet\_model',img\_grid)

writer.close()

Sometime Tensorboard doesn’t work if the model runs on the gpu but the images are on the cpu or vice-versa : <https://stackoverflow.com/questions/59013109/runtimeerror-input-type-torch-floattensor-and-weight-type-torch-cuda-floatte>

Sometimes the chosen port is taken, so I can change its number

tensorboard --logdir runs --host "0.0.0.0" --port 6008

tensorboard --logdir /home/golubeka/Trackmodule2/runs --port 6009

Tensorboard tutorials and guides

<https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html>

<https://pytorch.org/tutorials/intermediate/tensorboard_tutorial.html>

<https://youtu.be/VJW9wU-1n18>

<https://www.youtube.com/watch?v=pSexXMdruFM>

<https://towardsdatascience.com/a-complete-guide-to-using-tensorboard-with-pytorch-53cb2301e8c3>

<https://youtu.be/k7KfYXXrOj0>

# Neural Networks Data Preparation and Model tuning

## 3.1 Data augmentation

Data augmentation is a technique in machine learning used to reduce overfitting when training a machine learning model, by training models on several slightly-modified copies of existing data. It allows the model to be more robust especially when the dataset is too small.

In PyTorch, I define the image data generator that defines which augmentations I want to apply to the images.

datagen = ImageDataGenerator(

        rotation\_range=40,

        width\_shift\_range=0.2,

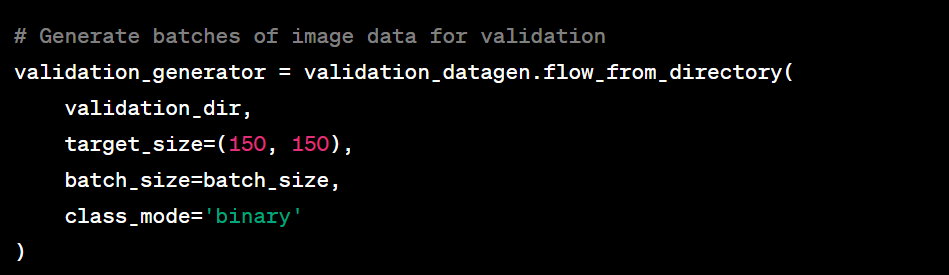
        height\_shift\_range=0.2,

        shear\_range=0.2,

        zoom\_range=0.2,

        horizontal\_flip=True,

        fill\_mode='nearest')



## 3.2 Effect of batch size on training dynamics

**Batch Size** is among the important hyperparameters in Machine Learning. It is the hyperparameter that defines the number of samples to work through before updating the internal model parameters. It can be one of the crucial steps to making sure your models hit peak performance.

Studies have shown that increasing batch size doesn’t lower performance as long as the learning rate is adjusted.

Larger Batch → Weak Generalization. But this can be fixed.

Large Batches require less computational power because we execute fewer updates.

This is why in most cases, you will see models trained with different batch sizes. It’s very hard to know off the bat what the perfect batch size for your needs is. However, there are some trends that you can use to save time.

If costs are important, large batch size might be the right thing.

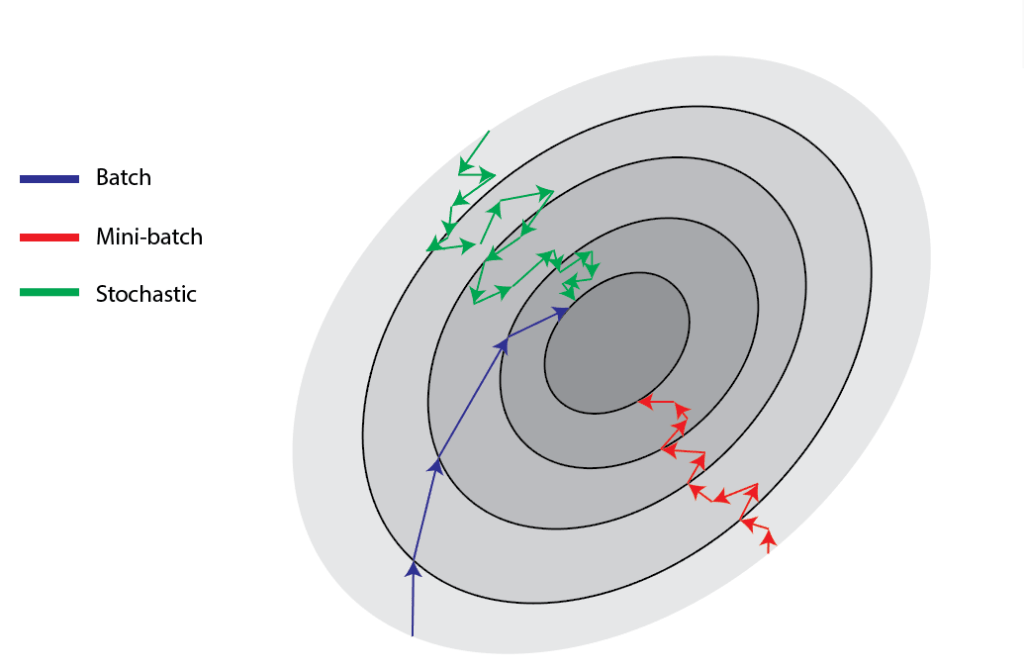
Small batch sizes might help when you care about generalization and need to throw something up quickly.

There are three types of gradient descent in respect to the batch size:

Batch gradient descent – uses all samples from the training set in one epoch.

Stochastic gradient descent – uses only one random sample from the training set in one epoch.

Mini-batch gradient descent – uses a predefined number of samples from the training set in one epoch.



The mini-batch gradient descent is the most common, empirically showing the best results. For instance, let’s consider the training size of 1000 samples and the batch size of 100. A neural network will take the first 100 samples in the first epoch and do forward and backward propagation. After that, it’ll take the subsequent 100 samples in the second epoch and repeat the process.

Overall, the network will be trained for the predefined number of epochs or until the desired condition is not met. The batch size affects some indicators such as overall training time, training time per epoch, quality of the model, and similar. Usually, we chose the batch size as a power of two, in the range between 16 and 512. But generally, the size of 32 is a rule of thumb and a good initial choice.

Sources :

<https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e#:~:text=Finding%3A%20large%20batch%20size%20means,all%20about%20the%20same%20size>

<https://medium.com/geekculture/how-does-batch-size-impact-your-model-learning-2dd34d9fb1fa>

Understand relationship between batch size and learning rate <https://www.baeldung.com/cs/learning-rate-batch-size>

<https://youtu.be/Y-zswp6Yxf0>

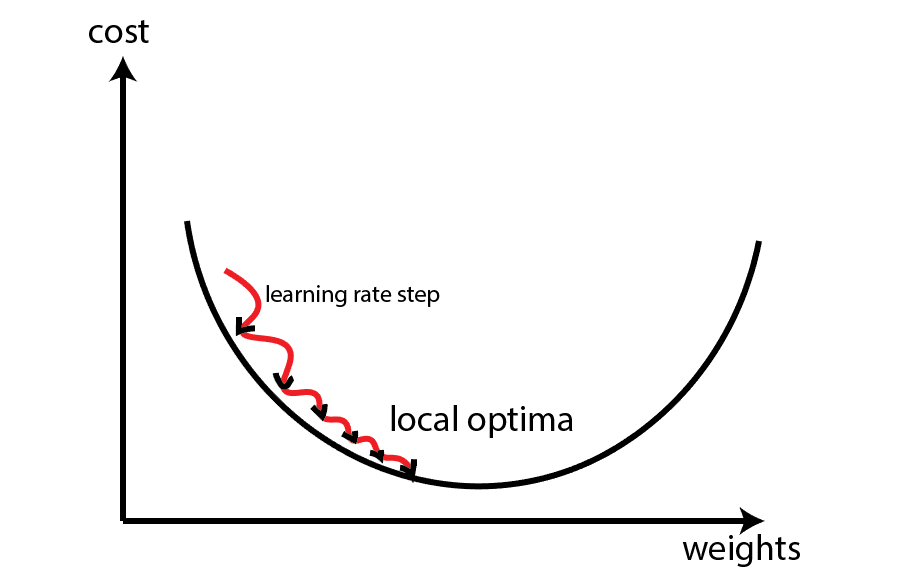
## 3.3 Learning rate schedulers

Learning rate

Learning rate is a term that we use in machine learning and statistics. Briefly, it refers to the rate at which an algorithm converges to a solution. Learning rate is one of the most important hyperparameters for training neural networks. Thus, it’s very important to set up its value as close to the optimal as possible.

Usually, when we need to train a neural network model, we need to use some optimization techniques based on a gradient descent algorithm. After calculating the gradient of the loss function with respect to weights, that gradient has a direction to the local optima. We use a learning rate hyperparameter to tune weights towards that direction and optimize the model.

The learning rate indicates the step size that gradient descent takes towards local optima:



Consequently, if the learning rate is too low, gradient descent will take more time to reach the optima. Conversely, if the learning rate is too big, the gradient descent might start to diverge, and it’ll never reach the optimal solution.

Also, the learning rate doesn’t have to have a fixed value. For example, we might define a rule that the learning rate will decrease as epochs for training increase. Besides that, some adaptive learning rate optimization methods modify the learning rate during the training. We can find more details about choosing the learning rate and gradient descent method in this article.

**Relation Between Learning Rate and Batch Size**

The question arises is there any relationship between learning rate and batch size. Do we need to change the learning rate if we increase or decrease batch size? First of all, if we use any adaptive gradient descent optimizer, such as Adam, Adagrad, or any other, there’s no need to change the learning rate after changing batch size.

Because of that, we’ll consider that we’re talking about the classic mini-batch gradient descent method.

**Learning rate scheduler**

Pytorch learning rate scheduler is used to find the optimal learning rate for various models by conisdering the model architecture and parameters.

|  |  |  |
| --- | --- | --- |
| **LR Scheduler** | **Description** | **Code** |
| ReduceLROnPlateau | Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a ‘patience’ number of epochs, the learning rate is reduced. | scheduler = optim.lr\_scheduler.ReduceLROnPlateau(optimizer, factor = 0.1, patience = 5, verbose = True) |
| StepLR | Decays the learning rate of each parameter group by gamma every step\_size epochs. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last\_epoch=-1, sets initial lr as lr. | scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size, gamma=0.1, last\_epoch=- 1, verbose=False) |
| MULTISTEPLR | Decays the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestones ( list of epoch indices). Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last\_epoch=-1, sets initial lr as lr. | torch.optim.lr\_scheduler.MultiStepLR(optimizer, milestones, gamma=0.1, last\_epoch=- 1, verbose=False) |
| CONSTANTLR | Decays the learning rate of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: total\_iters. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last\_epoch=-1, sets initial lr as lr. | torch.optim.lr\_scheduler.ConstantLR(optimizer, factor=0.3333333333333333, total\_iters=5, last\_epoch=- 1, verbose=False) |
| LINEARLR | Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: total\_iters. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last\_epoch=-1, sets initial lr as lr. | torch.optim.lr\_scheduler.LinearLR(optimizer, start\_factor=0.3333333333333333, end\_factor=1.0, total\_iters=5, last\_epoch=- 1, verbose=False) |
| EXPONENTIALLR | Decays the learning rate of each parameter group by gamma every epoch. When last\_epoch=-1, sets initial lr as lr. | torch.optim.lr\_scheduler.ExponentialLR(optimizer, gamma, last\_epoch=- 1, verbose=False) |

Chart, line chart

Description automatically generated

Figure ReduceOnPlateau

When calling ReduceLROnPlateau.step() a positional argument: 'metrics' is required. I have to specify the metrics I’m evaluating, for example : epoch\_loss.

A picture containing line, diagram, plot, text

Description automatically generated

Chart, line chart

Description automatically generated

Figure Comparing different learning rate schedulers

Sources :

Learning rate scheduler <https://youtu.be/P31hB37g4Ak>

Pytorch Lr SCHEDULER : ADJUST LR FOR BETTER RESULTS <https://youtu.be/81NJgoR5RfY>

<https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html#torch.optim.lr_scheduler.ReduceLROnPlateau>

## 

## Checkpoints

A picture containing text, font, screenshot

Description automatically generated**Checkpoints** is a callback, it’s an object that can perform certain action such as stopping the training once we achieved the desired accuracy.

When we fit the model we specify a parameters callbacks = callbacks\_list that is defined earlier.

We’ll explore only 3 types of callbacks here:

1. Model Checkpoint - Saves a model at certain interval in a folder saved\_models. We can save the model after every epoch or whenever the accuracy improves ( monitor = val\_accuracy, save\_best\_only = True, mode= max) or the loss is reduced ( monitor= loss, save\_best\_only = True, mode= min)
2. Early stopping - This callback will stop the training when there is no improvement in the validation loss for three consecutive epochs (patience = number of epochs with no improvement).

A picture containing text, font, screenshot

Description automatically generated

1. CSV Logger – creates a csv file with all the logs (epoch number, accuracy, loss, validation accuracy and validation loss). It might be useful to plot the results after training and evaluate the model.

A picture containing text, font, white, screenshot

Description automatically generated

Source: <https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping>

## Model evaluation

We can call the model.fit\_generator history and we store all the information about the model with respect to the epoch number. In order to retrieve this information we will call history.history. It creates a dictionary of size 4 : acc, loss, val\_acc, val\_loss.

A picture containing text, font, screenshot

Description automatically generated

# Transfer Learning

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest. The three major Transfer Learning scenarios look as follows:

**ConvNet as fixed feature extractor**. Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer’s outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. In an AlexNet, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier. We call these features **CNN codes**. It is important for performance that these codes are ReLUd (i.e. thresholded at zero) if they were also thresholded during the training of the ConvNet on ImageNet (as is usually the case). Once you extract the 4096-D codes for all images, train a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset.

**Fine-tuning the ConvNet**. The second strategy is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it’s possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset. In case of ImageNet for example, which contains many dog breeds, a significant portion of the representational power of the ConvNet may be devoted to features that are specific to differentiating between dog breeds.

**Pretrained models**. Since modern ConvNets take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final ConvNet checkpoints for the benefit of others who can use the networks for fine-tuning. For example, the Caffe library has a [Model Zoo](https://github.com/BVLC/caffe/wiki/Model-Zoo) where people share their network weights.

Using pretrained ResNet on Malarial dataset

Text

Description automatically generated

When the classifier is defined sequentially, I should modify the fc (fully connected layer) to change the classification to binary (in stead of 1000). It should happen using

model\_ft.fc = nn.Linear(num\_ftrs,2)

**ResNet18:**

(avgpool): AdaptiveAvgPool2d(output\_size=(1, 1))

**(fc):** Linear(in\_features=512, out\_features=1000, bias=True)

Using pretrained VGG16 on Malarial dataset

**VGG16 : => change classifier(6) last layer**

**(classifier): Sequential(**

(0): Linear(in\_features=25088, out\_features=4096, bias=True)

(1): ReLU(inplace=True)

(2): Dropout(p=0.5, inplace=False)

(3): Linear(in\_features=4096, out\_features=4096, bias=True)

(4): ReLU(inplace=True)

(5): Dropout(p=0.5, inplace=False)

(6): Linear(in\_features=4096, out\_features=1000, bias=True)

Sources :

Introduction to Transfer learning https://github.com/madsendennis/notebooks/tree/master/pytorch

Transfer learning tutorials

https://pytorch.org/tutorials/beginner/transfer\_learning\_tutorial.html

https://cs231n.github.io/transfer-learning/

tutorial https://youtu.be/t6oHGXt04ik

Alexnet pretrained model <https://medium.com/analytics-vidhya/pytorch-directly-use-pre-trained-alexnet-for-image-classification-and-visualization-of-the-dea0de3eade9>

PyTorch model zoo containing pretrained models

<https://pytorch.org/serve/model_zoo.html>

# 5. Visual Transformers

Diagram

Description automatically generated

Diagram

Description automatically generated

The transformer is primarily used in the field of natural language processing. Recently, it has been adopted and shows promise in the computer vision (CV) field. Medical image analysis (MIA), as a critical branch of CV, also greatly benefits from this state-of-the-art technique. The core component of the transformer is the attention mechanism. The applications can be categorized in a sequence of different tasks, including classification, segmentation, captioning, registration, detection, reconstruction, denoising, localization, and synthesis.

Recently, transformer-based architectures have attracted a lot of attention and have been widely used in different tasks. They leverage the self-attention mechanism to encode long-range dependency. However, when applied to medical image analysis tasks, transformer-based architectures still face many challenges, such as **large-scale data requirements during training** and **high computational complexity**. Consequently, there is a strong desire for innovative transformer-based methodologies that can effectively solve the tasks in the field of medical image analysis.

**Vision Transformers: A New Computer Vision Paradigm**

Transformers have great success with NLP and are now applied to images. CNN uses pixel arrays, whereas Visual Transformer(ViT) divides the image into visual tokens.

ViT splits an image into fixed-size patches, linearly embed each of them, add positional embedding as an input to Transformer Encoder. When ViT is trained on sufficient data it outperforms the state-of-the-art CNN by about four times fewer computational resources.

Self-attention layer in ViT allows it to integrate information globally across the entire image. ViT learns to encode the relative location of the patches to reconstruct the image structure from the training data.

Diagram

Description automatically generated

Graphical user interface, diagram

Description automatically generated

In this model, the CNN is used to extract the low-level features in an image, and ViT is used for relating high-level concepts.

The low-level features from CNN are fed to the ViT. Visual Transformer pays attention to only important regions, encodes the semantic concepts into few visual tokens by relating the spatially distant concepts using self-attention. These visual tokens can be used for image classification or projected back to the feature map for semantic segmentation.

Sources :

<https://arxiv.org/abs/2208.06643>

<https://www.frontiersin.org/research-topics/33732/transformer-in-the-field-of-medical-image-analysis>

<https://medium.com/swlh/visual-transformers-a-new-computer-vision-paradigm-aa78c2a2ccf2>

# Neural Architecture Search

**A picture containing text, screenshot, font, circle

Description automatically generated**

Neural Architecture Search (NAS) automates the process of architecture design of neural networks.  NAS approaches optimize the topology of the networks, incl. how to connect nodes and which operators to choose. User-defined optimization metrics can thereby include accuracy, model size or inference time to arrive at an optimal architecture for specific applications. Due to the extremely large search space, traditional evolution or reinforcement learning-based AutoML algorithms tend to be computationally expensive. Hence recent research on the topic has focused on exploring more efficient ways for NAS. In particular, recently developed gradient-based and multi-fidelity methods have provided a promising path and boosted research in these directions.

There are mainly three crucial components for a neural architecture search task, namely,

• Model search space that defines a set of models to explore.

• A proper strategy as the method to explore this model space.

• A model evaluator that reports the performance of every model in the space.

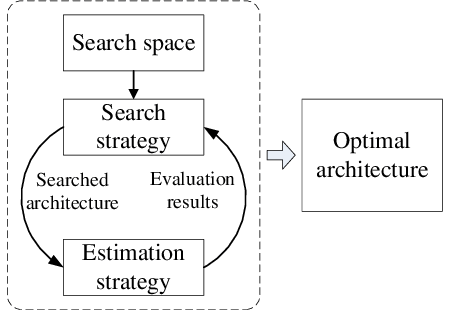
I can create the Search space / Model Space by changing the layerchoice and Valuechoice parameters. When defining the search space I should keep in mind to have at least 1 hidden layer, for more complex tasks 3< #hidden layers < 12 approximatively.

A screenshot of a computer program

Description automatically generated with medium confidence

A screen shot of a computer code

Description automatically generated with low confidence



**Define Model Space**

Model space is defined by users to express a set of models that users want to explore, which contains potentially good-performing models. In this framework, a model space is defined with two parts: a base model and possible mutations on the base model.

**Define Model Mutations**

A base model is only one concrete model not a model space. NNI provides API and Primitives for users to express how the base model can be mutated. That is, to build a model space which includes many models.

**Constructing Model Space**

pip install nni

from nni.retiarii.nn.pytorch import LayerChoice

Graphical user interface, application

Description automatically generated

**Explore the Defined Model Space**

There are basically two exploration approaches:

1. search by evaluating each sampled model independently, which is the search approach in multi-trial NAS
2. one-shot weight-sharing based search, which is used in one-shot NAS.

First, users need to pick a proper exploration strategy to explore the defined model space. Second, users need to pick or customize a model evaluator to evaluate the performance of each explored model.

Exploration Strategies :

Graphical user interface, email

Description automatically generated

**Pick or customize a model evaluator**

In the exploration process, the exploration strategy repeatedly generates new models. A model evaluator is for training and validating each generated model to obtain the model’s performance. The performance is sent to the exploration strategy for the strategy to generate better models.

**My first NAS experiment :** python nni\_hello\_hpo/main.py in terminal

The `nnictl hello` command is used as a quick way to verify that the NNI framework is correctly installed and accessible from the command line. It helps ensure that you can proceed with running more complex experiments and using other features of NNI for neural network architecture search and hyperparameter tuning.

A screenshot of a computer

Description automatically generated with medium confidence

**A diagram of a structure

Description automatically generated with low confidence**

**A close-up of a search strategy

Description automatically generated with low confidence**

**A picture containing text, diagram, screenshot, line

Description automatically generatedA diagram of cell types

Description automatically generated with medium confidence**

**A picture containing text, font, logo, design

Description automatically generatedA picture containing text, font, screenshot, design

Description automatically generatedA diagram of cell types

Description automatically generated with low confidence**

Advantages of NAS :

- Automation: NAS automates the process of neural network design, saving time and effort for researchers and practitioners.

- Performance Improvement: NAS can discover architectures that outperform manually designed ones, achieving state-of-the-art results.

- Domain-Specific Architectures: NAS can generate architectures tailored to specific domains, optimizing performance for specific tasks.

Limitations of NAS :

- Computational Cost: NAS often requires substantial computational resources, such as GPUs or even specialized hardware, due to the extensive search space and training of numerous candidate architectures.

- Search Space Constraints: NAS is constrained by the predefined search space, which limits the variety of architectures it can explore.

- Lack of Interpretability: Generated architectures may lack interpretability, making it challenging to understand the underlying mechanisms.

Sources:

Introduction to Neural Architecture Search from codelabsacademy <https://youtu.be/td820ts6gUU>

<https://www.automl.org/nas-overview/>

Neural Network Intelligence documentation <https://nni.readthedocs.io/en/stable/>

More research on NAS : <https://deci.ai/neural-architecture-search/>

AutoPyTorch tutorial : <https://github.com/automl/Auto-PyTorch> (only for tabular data)

List of APIs for constructing search space

<https://nni.readthedocs.io/en/stable/nas/construct_space.html>