INFO 4000
Informatics III

Data Science Specialization - Advanced

Week2 API services, Data Augmentation & Transfer Learning

Course Instructor

Jagannath Rao

raoj@uga.edu

Class Exercise 2

NFL data

- For avid football fans and aspiring analysts, a wealth of statistical information is available online to dissect every play, player, and team performance.
- Whether you're building complex predictive models or simply want to settle a debate with friends, these websites offer a treasure trove of NFL data.
- **Pro-Football-Reference:** A favorite among sports researchers and analysts, this site is a veritable encyclopedia of NFL history.
- What if we can predict whether a team will have a winning season based on their regular season statistics.
- **Goal:** To predict if an NFL team will have a winning season.
- We'll use Pro-Football-Reference, as its website structure uses clean HTML <table> elements, which is perfect for the "pandas.read_html() "function. We will scrape the main standings page from the recently completed 2023 season, which contains both team records and key statistics.

Exercise 2 – Data Integration with read_html()
Build a model that can predict a winning team in NFL

1. https://www.pro-football-reference.com/years/2023/ - this website has some tables which you can checkout.

2. There are two tables of the conferences that we will use - The first table is AFC, the second is NFC

3. Tasks to do:

   – Use the given url to scrape the tables data

   – Combine and clean the two conference tables and convert stat columns to numeric types for modeling ('PF', 'PA', 'PD', 'SoS') and these are described as "points for, points against, points differential, strength of schedule"

   – Remove 'W-L%' column. The main reason is to prevent a problem called data leakage. In machine learning, data leakage happens when information from your target variable (the thing you're trying to predict) accidentally "leaks" into your feature variables (the data you're using to make the prediction).

   – Create a label column suitable for classification (1s and 0s) – think of a logical way of doing this as it is not readily given.

   – Create an SQL database called 'NFL' and store the cleaned data in a table called 'stats'. Use the pandas 'to_sql' function.

   – Stats columns will be your features, so select them and drop the rest.

   – Use a classification model to predict win or lose in a season

Exercise 2 – contd …

4. Predict on new data of a hypothetical team ‘Atlanta Vipers’ using the model. Add this data and the prediction as a new row in the database table you have created.

   - 'PF': [465],
   - 'PA': [380],
   - 'PD': [85],
   - 'SoS': [1.5]

Postman APP for APIs

Testing APIs with Postman App

Install app from - https://www.postman.com/downloads/

With documentation in hand, there are several ways you can begin to use an API as a client.

1. **HTTP Clients:** generic program that lets you quickly build HTTP requests to test with. You specify the URL, headers, and body, and the program sends it to the server properly formatted – e.g. Postman, web apps

2. **Writing Code:** To really harness the power of an API, you will eventually need custom software. If you aren't sure which language to choose, a great way to narrow down the selection can be to find an API you want to implement and see if the company provides a client **library**. (e.g., Financial data from providers).

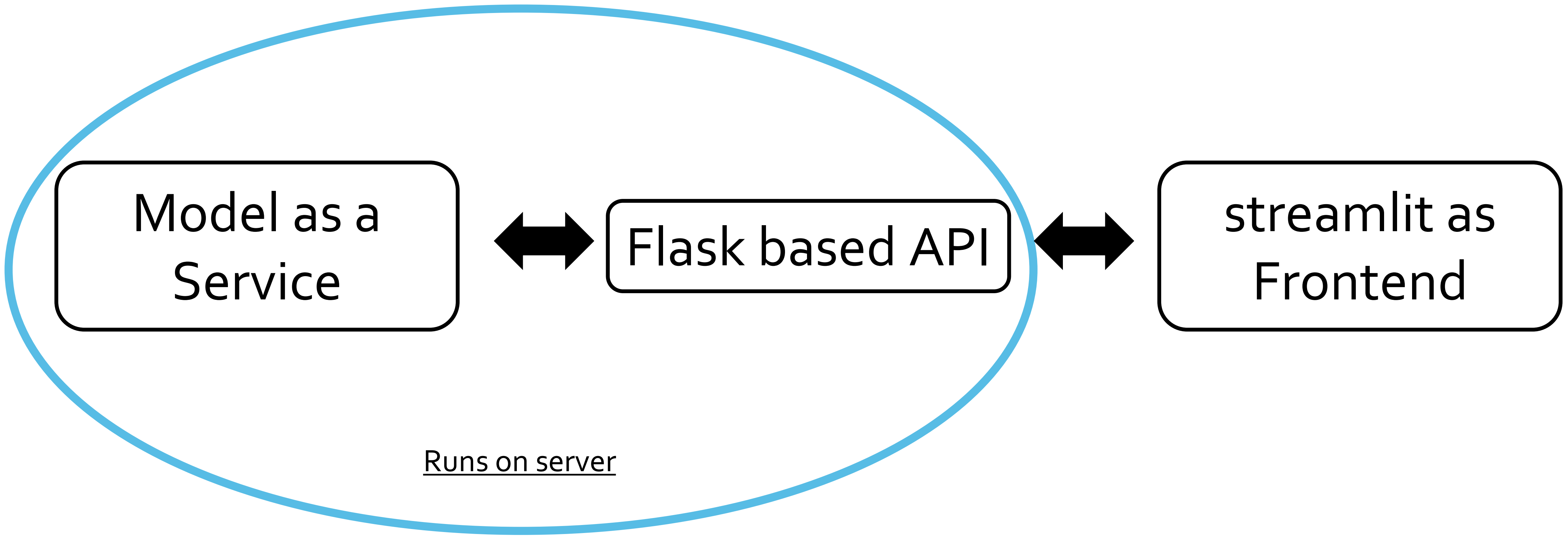
Let’s check out some available APIs

Alpha Vantage – Financial data site with free tier API key

ISS – Free without API key

OpenWeather Geocoding API -
https://openweathermap.org/api/geocoding-api#direct_zip

Deploy model as a service via an API



Let us set this up for one of our sklearn models and test it in Postman

Data Augmentation

Data Augmentation – Structured data

Feature Engineering:

1. **Creating New Features:** Combining existing features to create new ones.
2. **Transforming Features**: Applying mathematical transformations to features (e.g., log transformation, square root transformation).
3. **Encoding Categorical Features:** Converting categorical variables into numerical representations (e.g., one-hot encoding, label encoding).

Handling Missing Values:

1. **Imputation:** Replacing missing values with estimated values (e.g., mean, median, mode imputation).
2. **Deletion:** Removing rows or columns with missing values (if appropriate).

Handling Outliers:

1. **Winsorization:** Capping extreme values at a certain percentile.
2. **Transformation:** Applying transformations to reduce the impact of outliers (e.g., log transformation).
3. **Scaling and Normalization:** Standardization: Scaling features to have zero mean and unit variance.
4. **Normalization:** Scaling features to a specific range (e.g., [0, 1]).

Data Augmentation: Unstructured data - Images

Geometric Transformations:

1. Flipping (horizontal/vertical)
2. Rotation
3. Cropping
4. Scaling
5. Translation
6. Shearing

Color Space Adjustments:

1. Brightness
2. Contrast
3. Saturation
4. Hue

Random Noise:

1. Gaussian Noise
2. Salt and Pepper Noise

Data Augmentation: Unstructured data – Text & Audio

Text Data Augmentation:

1. Synonym Replacement: Replacing words with their synonyms.
2. Back Translation: Translating a sentence to another language and then back to the original language.
3. Random Insertion: Inserting random words into a sentence.
4. Random Deletion: Deleting random words from a sentence.
5. Random Swap: Swapping the positions of two words in a sentence.

Audio Data Augmentation:

1. Time Stretching: Changing the speed of audio without affecting the pitch.
2. Pitch Shifting: Changing the pitch of audio without affecting the speed.
3. Adding Noise: Adding background noise to audio.
4. Time Shifting: Shifting audio samples in time.

Data Augmentation guidelines

Important Considerations:

Domain Knowledge: The choice of augmentation and transformation techniques should be informed by the specific domain and problem being addressed.

Experimentation: It's essential to experiment with different techniques and evaluate their impact on model performance.

Balance: Augmentation should be used to improve model generalization but avoid over-augmenting data which may lead to unrealistic training examples.

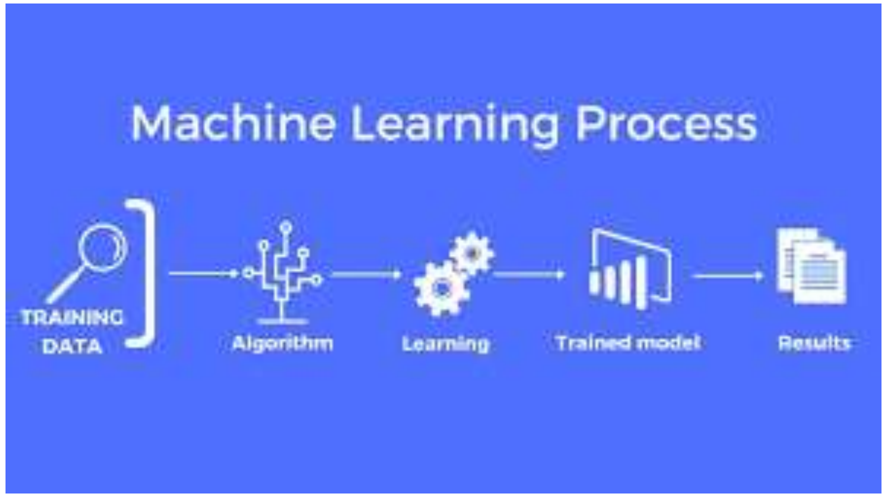
The key takeaway is that data augmentation provides a way to introduce variations and diversity to your training data ***without*** *increasing the size of your dataset on disk or the batch size. It helps improve the model's robustness and generalization capabilities by exposing it to a wider range of possible inputs during training.*

Data Augmentation Examples with code

Intro to Transfer Learning

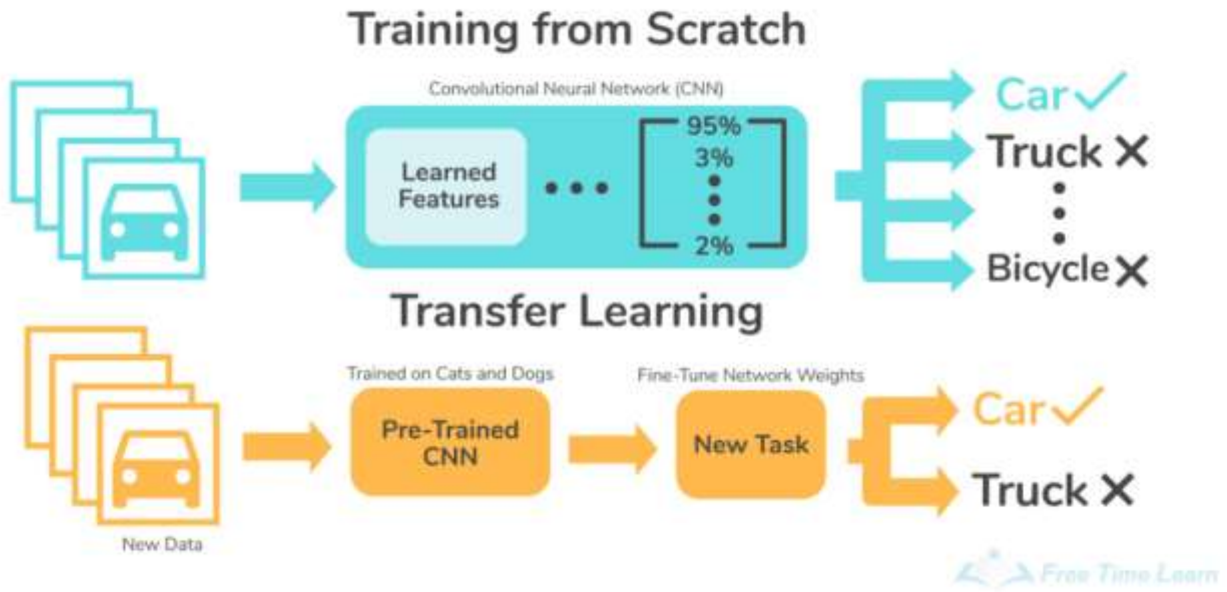
Recap – What does training a model mean?

- In the previous course, we have learned how to train Deep Learning models, for image classification, Natural Language processing etc.
- We also learned that to train such models we need huge amounts of training data.
- As an example, . the MNIST CNN model for digits needed about 70000 images to build a functional predictive model.
- These models can be saved and deployed for the world to use.
- Training these models require a large amount of computing resources and dollars.
- Which leads us to the topic of – can we use models that have already been created and build our applications by piggy backing on them?
- The answer is yes, and it is called – TRANSFER LEARNING.



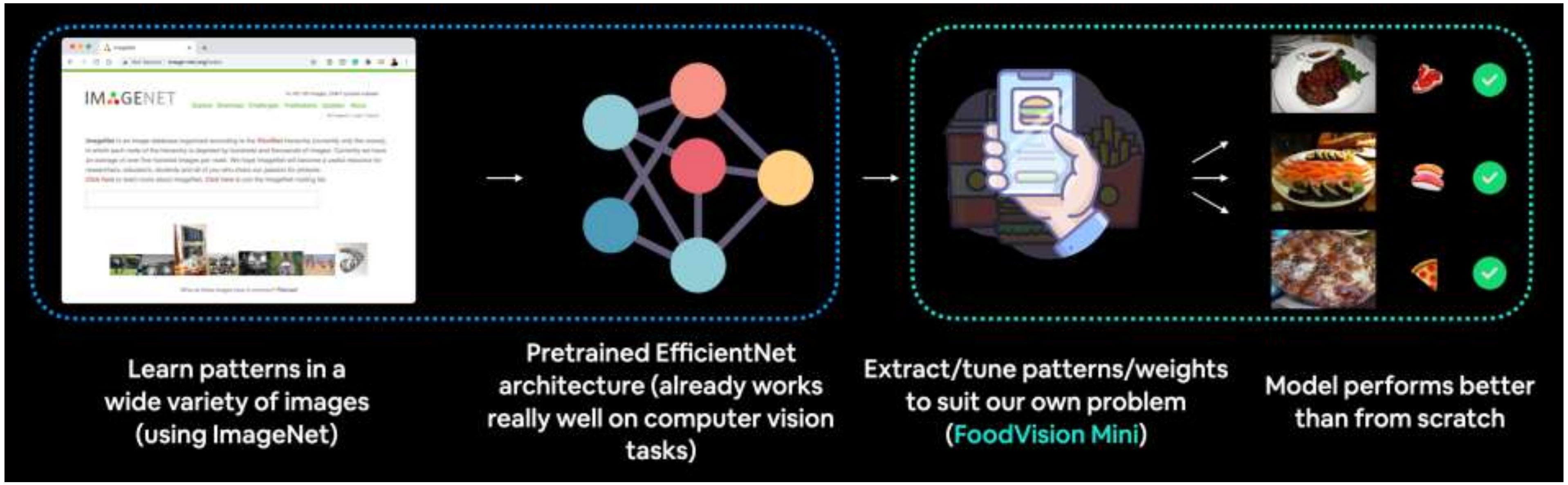
What is transfer learning?

- Instead of building models from scratch we ask the question, **are there well-performing models that already exist for our problem?**
- Companies like Google, Facebook, Hugging Face, etc. have invested in this and have built models for various image and NLP related applications and have open sourced it.
- These models are robust, have been tested in detail and perform extremely well.
- The technique to use these "pretrained" models to our specifications is called "Transfer Learning".
- **Transfer learning** allows us to take the patterns (also called weights) that another model has learned from a different use case and use it solve our own problem.
- Or we could take the patterns from a language model (a model that's been through large amounts of text to learn a representation of language) and use them as the basis of a model to classify different text samples.



Why use transfer learning?

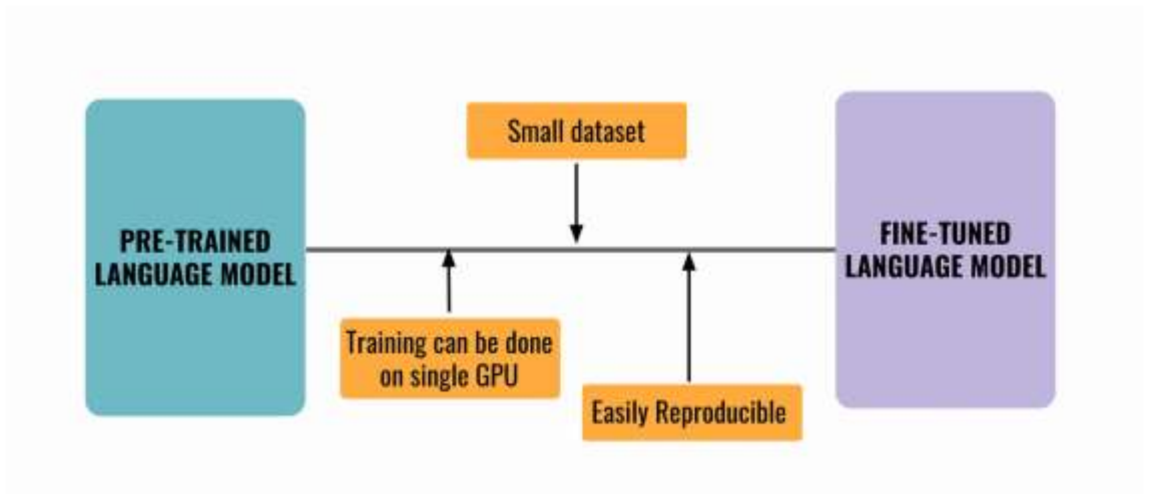
- The premise remains: find a well-performing existing model and apply it to your own problem.
- We can leverage an existing model (a neural network architecture) proven to work on problems like our own.
- We can leverage a working model which has **already learned** patterns on similar data to our own.
- This often results in achieving **great results with less custom data – available small dataset**.


IMAGENET
Learn patterns in a wide variety of images (using ImageNet)
Pretrained EfficientNet architecture (already works really well on computer vision tasks)
Extract/tune patterns/weights to suit our own problem (FoodVision Mini)
Model performs better than from scratch

Benefits of pretrained models

- A study into the effects of whether training from scratch or using transfer learning was better from a practitioner's point of view found transfer learning to be far more beneficial in terms of cost and time.
- Suddenly lots more people can do world-class work with less resources and less data. - that includes you and me.
- It should be common practice at the start of every deep learning problem you take on to ask, "Does a pretrained model exist for my problem?
- As a reminder, [PyTorch documentation]() and [PyTorch developer forums](), are very helpful places for all things PyTorch.
- Huggingface is another popular platform for open sourced pretrained models.

Image Classification using Pre-trained Models in PyTorch

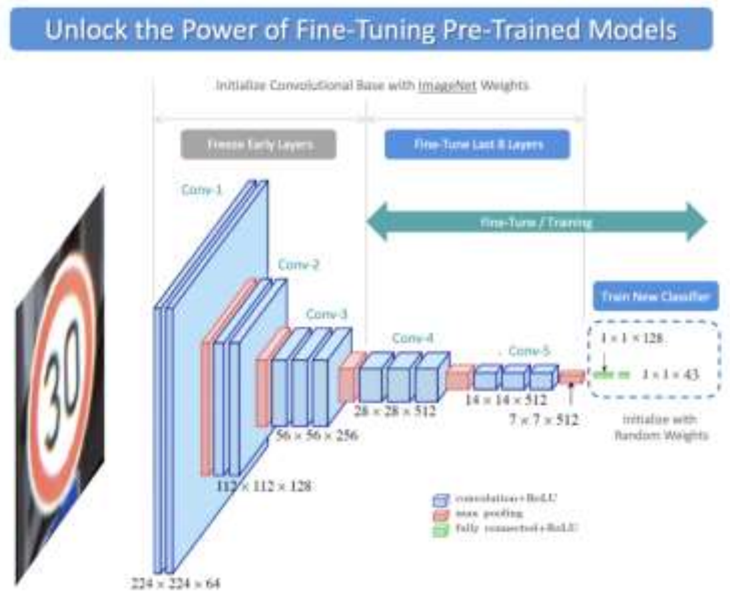


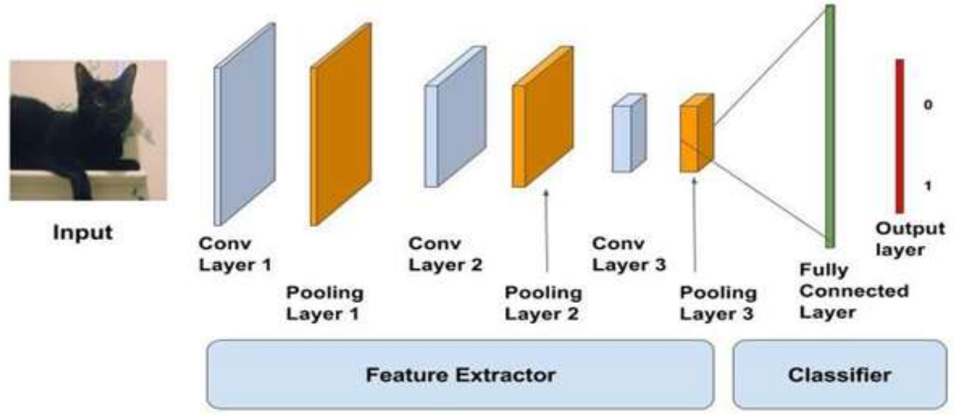
PRE-TRAINED LANGUAGE MODEL
Small dataset
Training can be done on single GPU
Easily Reproducible
FINE-TUNED LANGUAGE MODEL

2 ways to use pretrained models in CNN applications

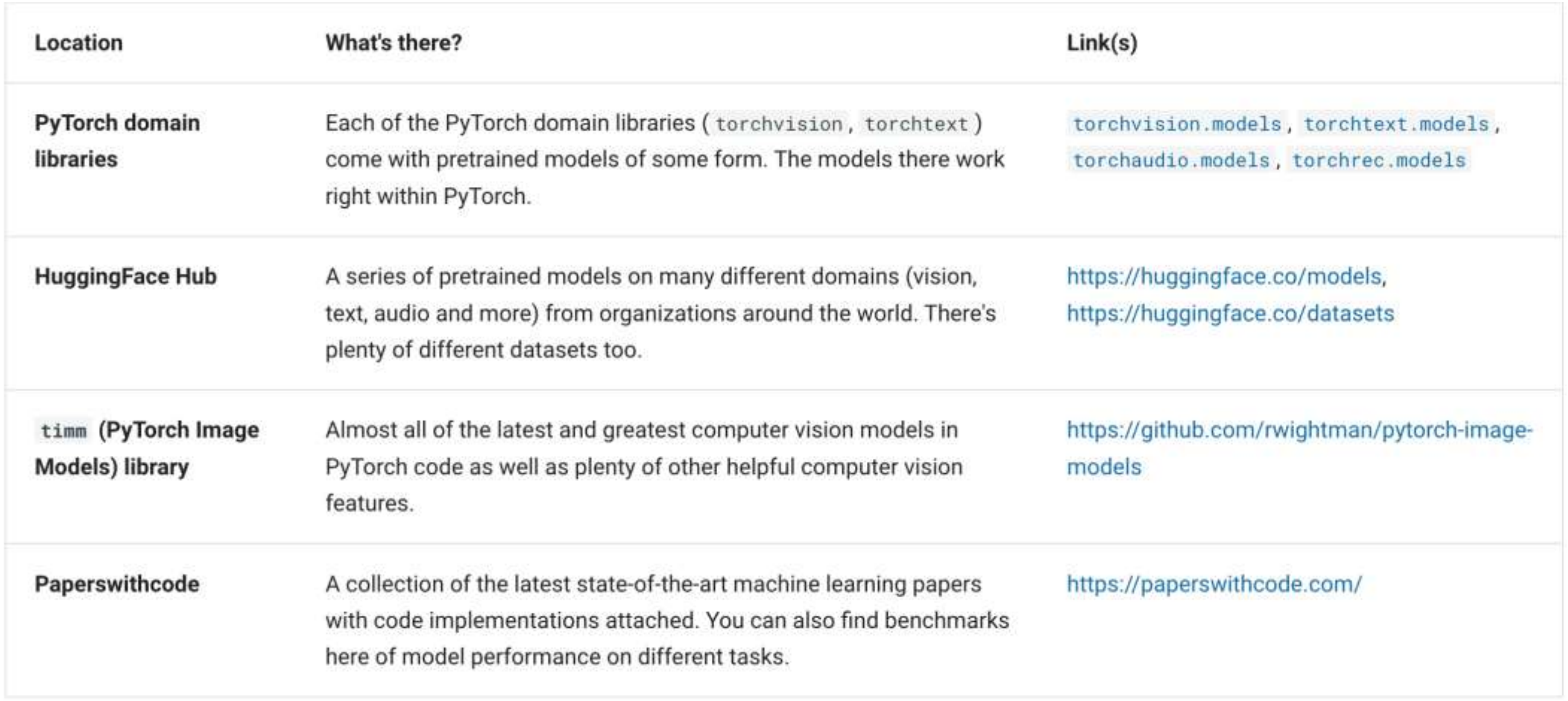
- **Finetuning the convnet**:
  - Instead of random initialization, we initialize the network with a pretrained network.
  - Rest of the training looks as usual.

- **ConvNet as fixed feature extractor**:
  - Here, we will freeze the weights for all the Convolution layers.
  - The last / last few fully connected layers are replaced with a new one with random weights and only this layer is trained.





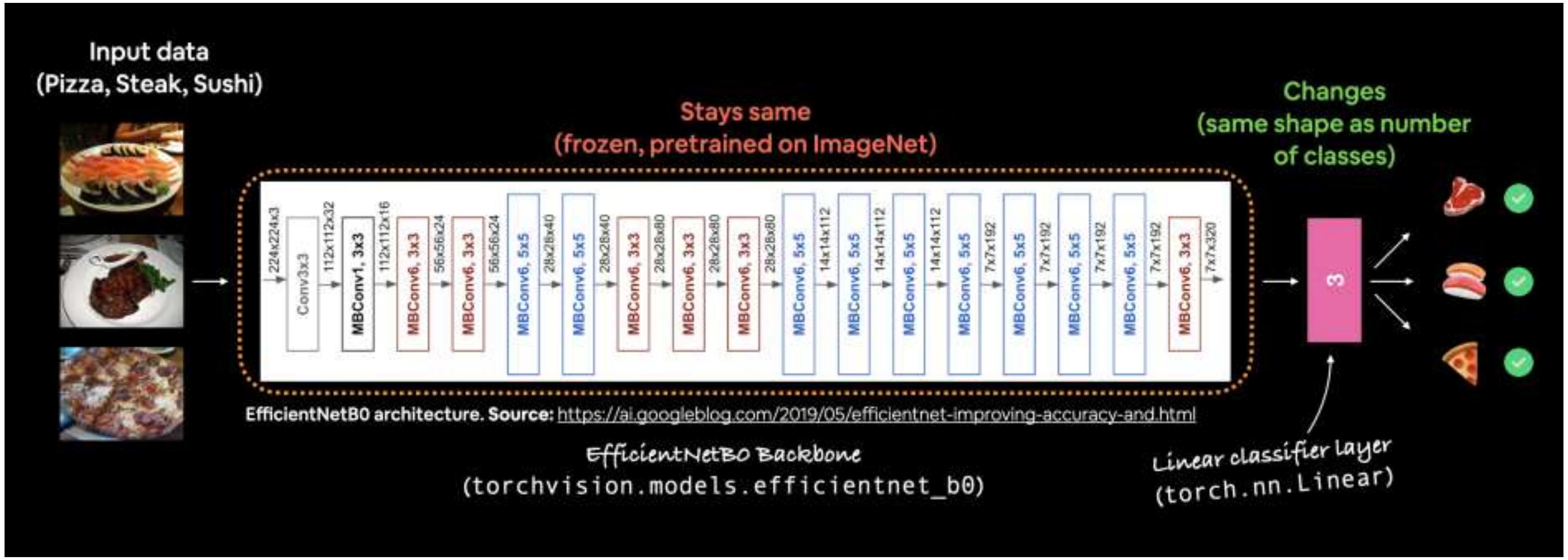
Where to find pretrained models?



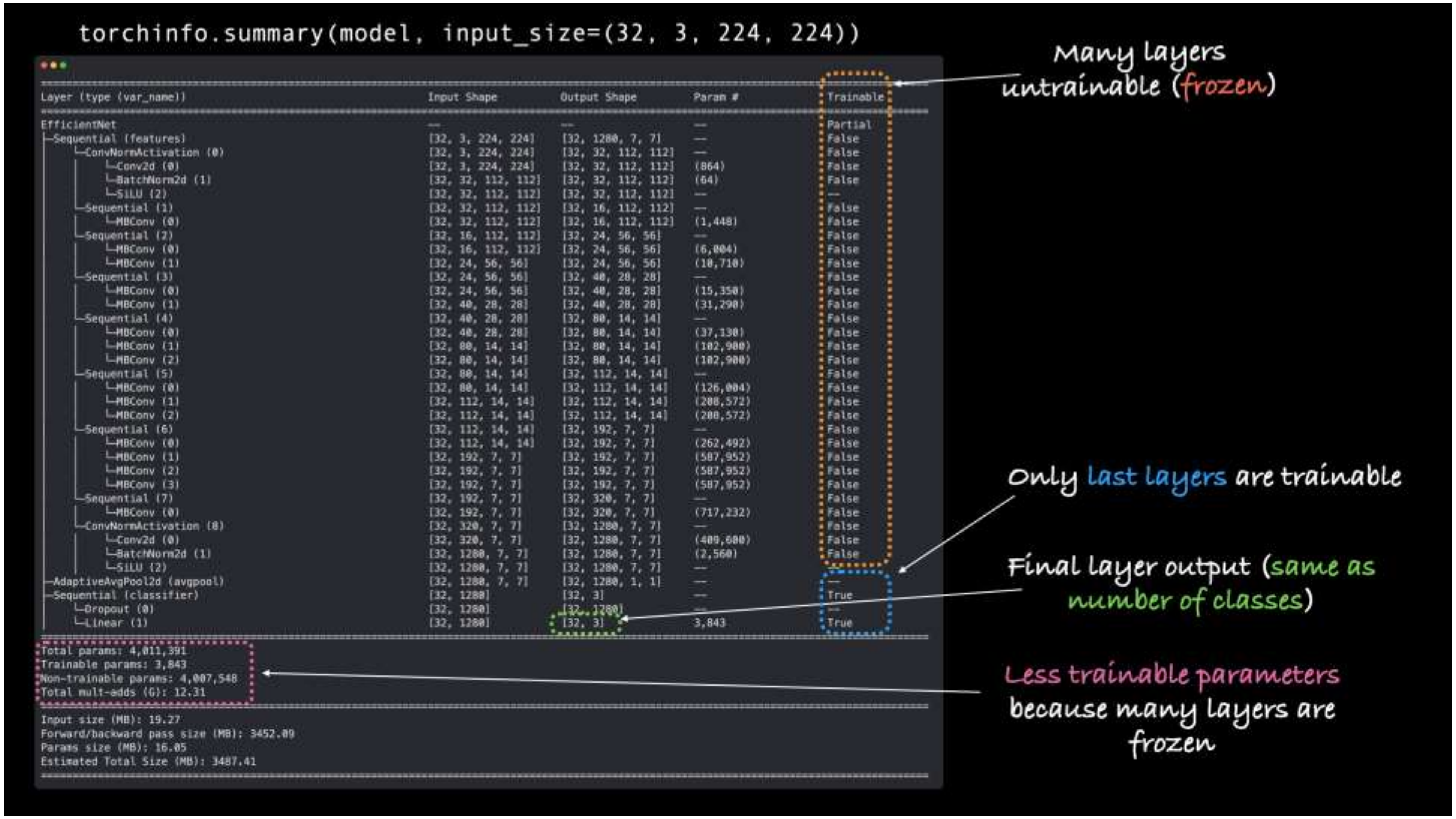
Location	What's there?	Link(s)
PyTorch domain libraries	Each of the PyTorch domain libraries (`torchvision`, `torchtext`) come with pretrained models of some form. The models there work right within PyTorch.	`torchvision.models`, `torchtext.models`, `torchaudio.models`, `torchrec.models`
HuggingFace Hub	A series of pretrained models on many different domains (vision, text, audio and more) from organizations around the world. There's plenty of different datasets too.	https://huggingface.co/models, https://huggingface.co/datasets
`timm` **(PyTorch Image Models) library**	Almost all of the latest and greatest computer vision models in PyTorch code as well as plenty of other helpful computer vision features.	https://github.com/rwightman/pytorch-image-models
Paperswithcode	A collection of the latest state-of-the-art machine learning papers with code implementations attached. You can also find benchmarks here of model performance on different tasks.	https://paperswithcode.com/

Setting up a pretrained model -1

- freeze some base layers of a pretrained model (typically the features section) and
- then adjust the output layers (also called head/classifier layers) to suit your needs.
- In the picture below, we freeze all the layers/parameters in the features section of this efficientnet_bo model. Freeze means the weights are not trainable OR as we know it "require_grad = False".
- When using a pretrained model, it's important that **your custom data going into the model is prepared in the same way as the original training data that went into the model**.



Setting up a pretrained model - 2



```
torchinfo.summary(model, input_size=(32, 3, 224, 224))
```

```
==========================================================================================================
Layer (type (var_name))                      Input Shape        Output Shape       Param #       Trainable
==========================================================================================================
EfficientNet                                 --                 --                 --            Partial
├─Sequential (features)                      [32, 3, 224, 224]  [32, 1280, 7, 7]   --            False
│    └─ConvNormActivation (0)                [32, 3, 224, 224]  [32, 32, 112, 112] --            False
│    │    └─Conv2d (0)                       [32, 3, 224, 224]  [32, 32, 112, 112] (864)         False
│    │    └─BatchNorm2d (1)                  [32, 32, 112, 112] [32, 32, 112, 112] (64)          False
│    │    └─SiLU (2)                         [32, 32, 112, 112] [32, 32, 112, 112] --            --
│    └─Sequential (1)                        [32, 32, 112, 112] [32, 16, 112, 112] --            False
│    │    └─MBConv (0)                       [32, 32, 112, 112] [32, 16, 112, 112] (1,448)       False
│    └─Sequential (2)                        [32, 16, 112, 112] [32, 24, 56, 56]   --            False
│    │    └─MBConv (0)                       [32, 16, 112, 112] [32, 24, 56, 56]   (6,004)       False
│    │    └─MBConv (1)                       [32, 24, 56, 56]   [32, 24, 56, 56]   (10,710)      False
│    └─Sequential (3)                        [32, 24, 56, 56]   [32, 40, 28, 28]   --            False
│    │    └─MBConv (0)                       [32, 24, 56, 56]   [32, 40, 28, 28]   (15,350)      False
│    │    └─MBConv (1)                       [32, 40, 28, 28]   [32, 40, 28, 28]   (31,290)      False
│    └─Sequential (4)                        [32, 40, 28, 28]   [32, 80, 14, 14]   --            False
│    │    └─MBConv (0)                       [32, 40, 28, 28]   [32, 80, 14, 14]   (37,130)      False
│    │    └─MBConv (1)                       [32, 80, 14, 14]   [32, 80, 14, 14]   (102,900)     False
│    │    └─MBConv (2)                       [32, 80, 14, 14]   [32, 80, 14, 14]   (102,900)     False
│    └─Sequential (5)                        [32, 80, 14, 14]   [32, 112, 14, 14]  --            False
│    │    └─MBConv (0)                       [32, 80, 14, 14]   [32, 112, 14, 14]  (126,004)     False
│    │    └─MBConv (1)                       [32, 112, 14, 14]  [32, 112, 14, 14]  (208,572)     False
│    │    └─MBConv (2)                       [32, 112, 14, 14]  [32, 112, 14, 14]  (208,572)     False
│    └─Sequential (6)                        [32, 112, 14, 14]  [32, 192, 7, 7]    --            False
│    │    └─MBConv (0)                       [32, 112, 14, 14]  [32, 192, 7, 7]    (262,492)     False
│    │    └─MBConv (1)                       [32, 192, 7, 7]    [32, 192, 7, 7]    (587,952)     False
│    │    └─MBConv (2)                       [32, 192, 7, 7]    [32, 192, 7, 7]    (587,952)     False
│    │    └─MBConv (3)                       [32, 192, 7, 7]    [32, 192, 7, 7]    (587,952)     False
│    └─Sequential (7)                        [32, 192, 7, 7]    [32, 320, 7, 7]    --            False
│    │    └─MBConv (0)                       [32, 192, 7, 7]    [32, 320, 7, 7]    (717,232)     False
│    └─ConvNormActivation (8)                [32, 320, 7, 7]    [32, 1280, 7, 7]   --            False
│    │    └─Conv2d (0)                       [32, 320, 7, 7]    [32, 1280, 7, 7]   (409,600)     False
│    │    └─BatchNorm2d (1)                  [32, 1280, 7, 7]   [32, 1280, 7, 7]   (2,560)       False
│    │    └─SiLU (2)                         [32, 1280, 7, 7]   [32, 1280, 7, 7]   --            --
├─AdaptiveAvgPool2d (avgpool)                [32, 1280, 7, 7]   [32, 1280, 1, 1]   --            --
├─Sequential (classifier)                    [32, 1280]         [32, 3]            --            True
│    └─Dropout (0)                           [32, 1280]         [32, 1280]         --            --
│    └─Linear (1)                            [32, 1280]         [32, 3]            3,843         True
==========================================================================================================
Total params: 4,011,391
Trainable params: 3,843
Non-trainable params: 4,007,548
Total mult-adds (G): 12.31
==========================================================================================================
Input size (MB): 19.27
Forward/backward pass size (MB): 3452.09
Params size (MB): 16.05
Estimated Total Size (MB): 3487.41
==========================================================================================================
```

Many layers untrainable (frozen)
Only last layers are trainable
Final layer output (same as number of classes)
Less trainable parameters because many layers are frozen

Setting up a pretrained model - 3

Torchvision pretrained models expect:

– Almost all pre-trained models expect input images normalized in the same way, i.e., mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224.

– The images must be loaded into a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]. These were calculated from the data. Specifically, the ImageNet dataset by taking the means and standard deviations across a subset of images.

– So, we must transform our data to fit into this. Fortunately, we have an automatic way of creating those transforms using PyTorch. However, the tradeoff of using automatically created transforms is a lack of customization.

Example:

Let us inspect a pretrained model

Which pretrained model to use?

- It depends on your problem/the device you're working with.
- Generally, the higher number in the model's name (e.g., efficientnet_b0() -> efficientnet_b1() -> efficientnet_b7()) means *better performance* but a *larger* model.
- But **some better performing models are too big for some devices**.
- For example, say you'd like to run your model on a mobile-device, you'll have to take into account the limited compute resources on the device, thus you'd be looking for a smaller model.
- But if you've got unlimited compute power, you'd likely take the biggest, most compute hungry model you can.
- Understanding this **performance vs. speed vs. size tradeoff** will come with time and practice.
- Even though we're using efficientnet_bX, it's important not to get too attached to any one architecture, as they are always changing as new research gets released. Best to experiment, experiment, experiment and see what works for your problem.

Building a model for a custom dataset

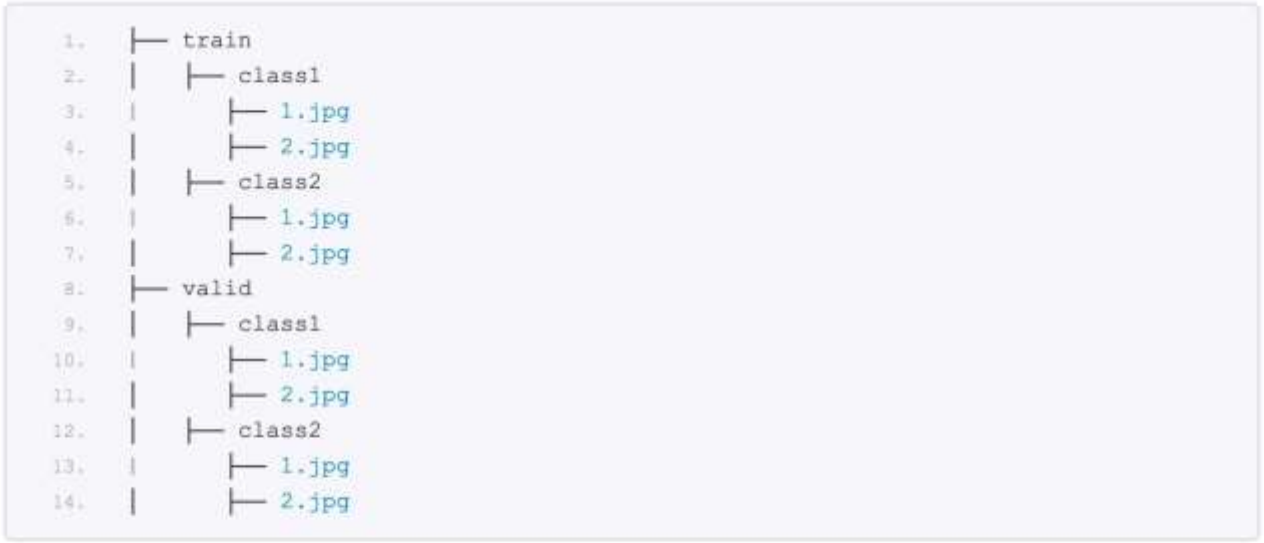
- A **custom dataset** is a collection of data relating to a specific problem you're working on. Usually, we have a small number of samples, but enough to tune a pretrained model.
- In essence, a **custom dataset** can be comprised of almost anything. For example, if we were building a food image classification app then, our custom dataset might be images of food.
- Or if we were trying to build a model to classify whether or not a text-based review on a website was positive or negative, our custom dataset might be examples of existing customer reviews and their ratings.
- Or if we were trying to build a sound classification app, our custom dataset might be sound samples alongside their sample labels.
- Or if we were trying to build a recommendation system for customers purchasing things on our website, our custom dataset might be examples of products other people have bought.

Example:

Let us put all these ideas to work with “ConvNet as a Feature Extractor”

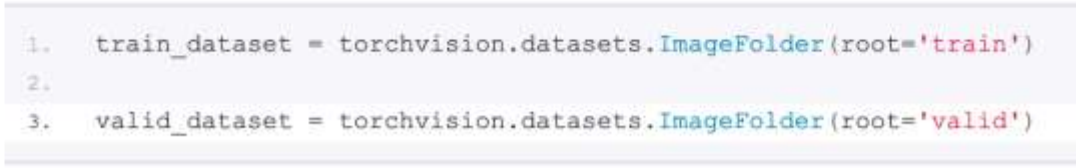
Refresh - PyTorch “ImageFolder” Class

- The **ImageFolder** class is a part of the torchvision library’s datasets module.
- We can easily access it using the following syntax: ***torchvision.datasets.ImageFolder***
- This class helps us to easily create PyTorch training and validation datasets without writing custom classes.
- The image datasets folder should be of the following structure



```
├── train
|   ├── class1
|   |   ├── 1.jpg
|   |   ├── 2.jpg
|   ├── class2
|   |   ├── 1.jpg
|   |   ├── 2.jpg
├── valid
|   ├── class1
|   |   ├── 1.jpg
|   |   ├── 2.jpg
|   ├── class2
|   |   ├── 1.jpg
|   |   ├── 2.jpg
```

- Syntax to create the training and validation datasets:



```
train_dataset = torchvision.datasets.ImageFolder(root='train')

valid_dataset = torchvision.datasets.ImageFolder(root='valid')
```

Example:

Let us put all these ideas to work with “ConvNet with FineTuning”

Fine Tuning a pretrained model

- 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.
- 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.
- In Fine Tuning, the target model copies all model designs with their parameters from the source pretrained model except the output layer, and fine-tunes these parameters based on the target dataset.
- Because the parameters are of a trained model, we do not want the weights to be moving around too far from the original and hence we tend to use low learning rates.

When and how to fine tune?

How do you decide what type of transfer learning you should perform on a new dataset?

- ***New dataset is small and similar to original dataset***. Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.

- ***New dataset is large and similar to the original dataset.*** Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.

- ***New dataset is small but very different from the original dataset.*** Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier form the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.

- ***New dataset is large and very different from the original dataset.*** Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch. However, in practice it is very often still beneficial to initialize with weights from a pretrained model. In this case, we would have enough data and confidence to fine-tune through the entire network.

Transformers

Intuitive explanation of understanding textual data

- Think of these sentences:
  - The money in my bank is not enough
  - The bank of the river is narrow.
  - I cannot bank on others to achieve my goal.
- The word bank appears in all three but in different contexts and with different meaning. So, if a model must understand a piece of text well, it needs to understand the context.
- To derive context, there needs to be a mechanism which can somehow connect the words
  - bank & money
  - river & bank
  - bank & on others
- If we have such a mechanism, we achieve a couple of things:
  - The NN models can be trained to learn and use context
  - A side effect is that we can now look at a complete sentence, derive context and process it at the same time: parallel processing instead of sequential.

What is a Transformer?

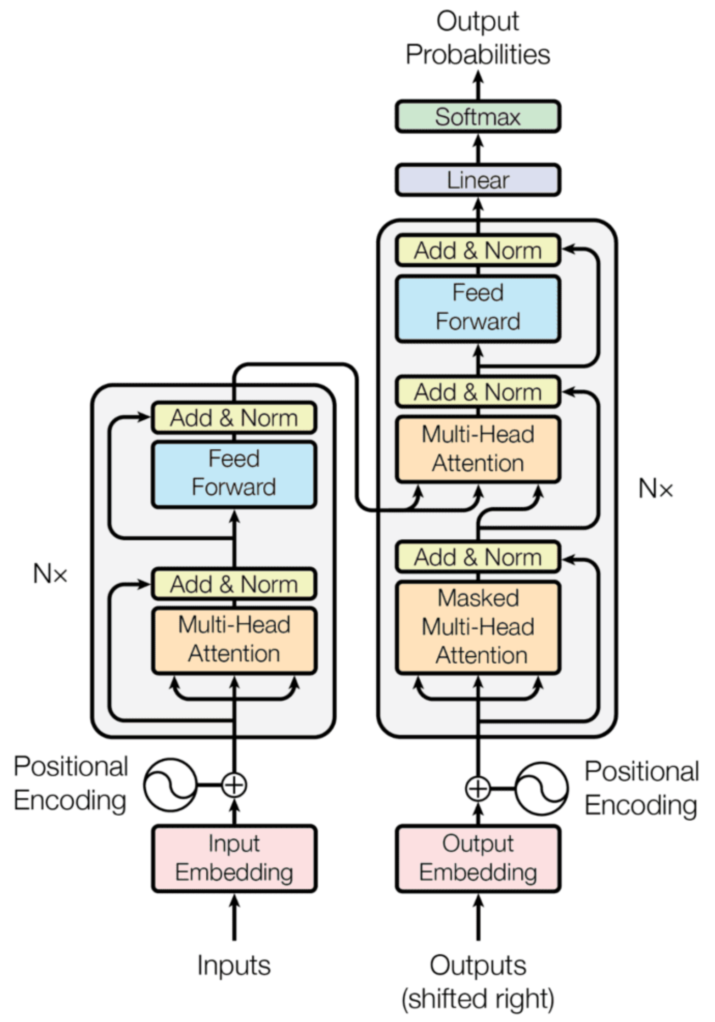
- A transformer model is a neural network that learns context and therefore meaning, by tracking relationships between the words, in a sentence.
- First described in a 2017 paper from Google called “Attention is all you need”, transformers are among the newest and one of the most powerful classes of models invented to date.
- Any application using sequential text, image or video data is a candidate for transformer models.
- Transformers are in many cases replacing convolutional and recurrent neural networks (CNNs and RNNs), the most popular types of deep learning models just a few years ago.
- RNNs / LSTMs work in a sequential manner (words coming in time steps) and hence do not lend themselves to parallel processing.
- Transformers on the other hand, process the words in parallel (as they develop an understanding for the context of the text) and hence can-do parallel processing. Makes them faster to train and they function in near real time.

“Attention Mechanism” – Mechanism for deriving context

- While understanding the meaning of the word "bank" in a sentence, your brain refers to other parts of the sentence or paragraph to determine whether "bank" means a financial institution or the side of a river.
- This act of referring and determining the importance of previous words is analogous to the attention mechanism.
- In the attention mechanism, every word in a sentence can focus on every other word, including itself.
- This focus is quantified by weights (often called attention scores). A higher weight means the word is paying more "attention" to another specific word.
- For example, for the sentence "He put money in the bank," when trying to understand the context of the word "bank," the model might assign higher attention scores to "money" and "put" to infer that it's about a financial institution and not the river's side.
- Using these attention scores, the model computes a weighted combination of all input words, generating a new representation for each word. This new representation carries the context.
- For the word "bank" in our example, its new representation will carry more context from "money" and "put" due to the higher attention weights.

Transformer architecture

- The Transformer architecture has two main parts: the encoder and the decoder. .
- The encoder reads the entire input text and understands its meaning by paying attention to all the words and their relationships.
- The decoder then takes this understanding and generates new text, word by word, by focusing on both the input text and what it has generated so far.
- There are applications where we need only the encoder and similarly, only the decoder.
  - We don't need the decoder when we are only interested in understanding or processing the input text, such as in tasks like text classification, sentiment analysis, or extracting information. We use only the encoder part.
  - We don't need the encoder when we are generating text from scratch based on some initial input, such as in text generation tasks where the model continues a given prompt. This is when we use only the decoder part of the transformer, known as a "GPT-like" model.



“Multi-Head Attention” Mechanism

- In Transformer models, the attention mechanism is not performed just once. It's done multiple times in parallel, each with different learned parameters, allowing the model to focus on different aspects of the input.
- These parallel operations are called "heads" in multi-head attention.
- For instance, one "head" might focus on syntactic aspects (the structure of the sentence) while another might focus on semantic aspects (the meaning of the words).
- Combining these multiple perspectives allows the model to generate richer context.
- Attention allows deep learning models to dynamically focus on different parts of the input data, making them particularly powerful for tasks where context from various parts of the input is crucial, like in NLP.
- The Transformer architecture utilizes this mechanism to great effect, leading to state-of-the-art performance in many tasks.
- Transformer needs a way to consider the position of words in a sequence. Positional encoding is a method to give the model information about the position of words in a sequence.
- Once the positional information is added, the modified embeddings are passed through the attention mechanism.
- Now, when the model calculates attention scores, it has access to both word meaning and position, allowing it to generate context-aware representations of words based on their content and order in the sequence.

End of Lesson