# Re-using Pre-trained models

In the module on Generative AI using an RNN, we linked (https://app.inferkit.com/demo) to a model that generates text that continues an initial "seed" sequence of words.

As we saw: the stories it produced were impressive.

This, and similar models, were trained on a simple task that we called "Predict the Next"

- given an initial sequence (e.g., of word)
- predict the next element of the sequence

We already know enough to be able to construct such a model ourselves:

- An initial network (e.g., LSTM, Transformer) that transforms a sequence into a fixed length representation
- Followed by a Classifier head (to generate a probability distribution over possible next elements)

We even demonstrated how to construct a training example for this task

For example, given a sequence of words

$\mathbf{s} =$ "I am taking a class in Machine Learning"

| $i$ | $\mathbf{x}^{(\mathbf{i})}$ | $\mathbf{y}^{(\mathbf{i})}$ |
|---|---|---|
| $1$ | $[\ \mathrm{I}\ ]$ | $\mathrm{am}$ |
| $2$ | $[\ \mathrm{I}, \mathrm{am}\ ]$ | $\mathrm{taking}$ |
| $3$ | $[\ \mathrm{I}, \mathrm{am}, \mathrm{taking}\ ]$ | $\mathrm{a}$ |

Simple enough.

But one reason the linked model was so powerful was because of its size and training

- 3 billion weights !
- Trained on 500 billion tokens
    - Used $ 42K of electricity

Even if this is not beyond our intellectual capacity, training such a model may be beyond our physical and financial capacities.

The good news is that researchers often publish their models and trained weights in a public repo.

We can re-use their models for free !

- In our DL Assignments, you were asked to save your models and trained weights in a file
- You can share your models too !

## HuggingFace (https://huggingface.co/models)

- Keras/PyTorch

## Keras_Hub (https://keras.io/keras_hub/getting_started/)

# Keras project

## Image

ImageNet pre-trained models (https://keras.io/applications/)

## NLP

Pre-trained word embeddings
(https://keras.io/examples/nlp/pretrained_word_embeddings/)

## Model zoo

Open source, pre-trained models (https://modelzoo.co)

- YOLO: object detection and classification (https://modelzoo.co/model/yolo-tensorflow)
- OpenPose: Human body, hand and facial keypoints (https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/.github/media/pose_face_hands.gif)

# Transfer Learning: how to learn from little data

Using pre-trained models is great if our task is identical to the training task and our test examples are from an *identical* distribution as the training examples.

But it is often the case that even if the task is identical, we want to use test examples from a *similar* distribution.

Consider how the mean of words change in the specialized domain of Quant Finance compared to General English

| Word | English meaning | Quant Finance meaning |
|---|---|---|
| call (noun) | cry; contact | type of option |
| bull | male bovine | optimist |

A "predict the next" word task trained on General English is likely to perform below expectations for examples from Quant Finance.

It would seem that a Quant Finance researcher would need to train on specialized examples.

The biggest constraint in training a model is obtaining a sufficient amount of training data.

The deeper (greater number of layers) your model

- The more weights/parameters need to be estimated
- Which increases the need for a larger quantity of training data

Obtaining large quantities of examples from specialized domains is challenging, labor-intensive and time consuming.

Enter *Transfer Learning*

- Import the architecture and weights of a Pre-Trained model from the Source Task
    - trained with a large quantity of general examples
- "Fine tune" the weights to adapt the model to solve the Target Task
    - by *continuing the training* on a *small* quantity of specialized examples

Recall our module on Interpreting the layers of a Neural Network (NN)

- layers close to the input seem to learn simple features
- layer $l$ creates new features that are combinations of features of layer $(l-1)$

Is it possible that we can "re-use" feature transformations ?

- Use the layers closest to input for a Neural Network trained on a "source" Task
- But apply these layers (and their transformations on input) to a new "target" Task ?

Yes !

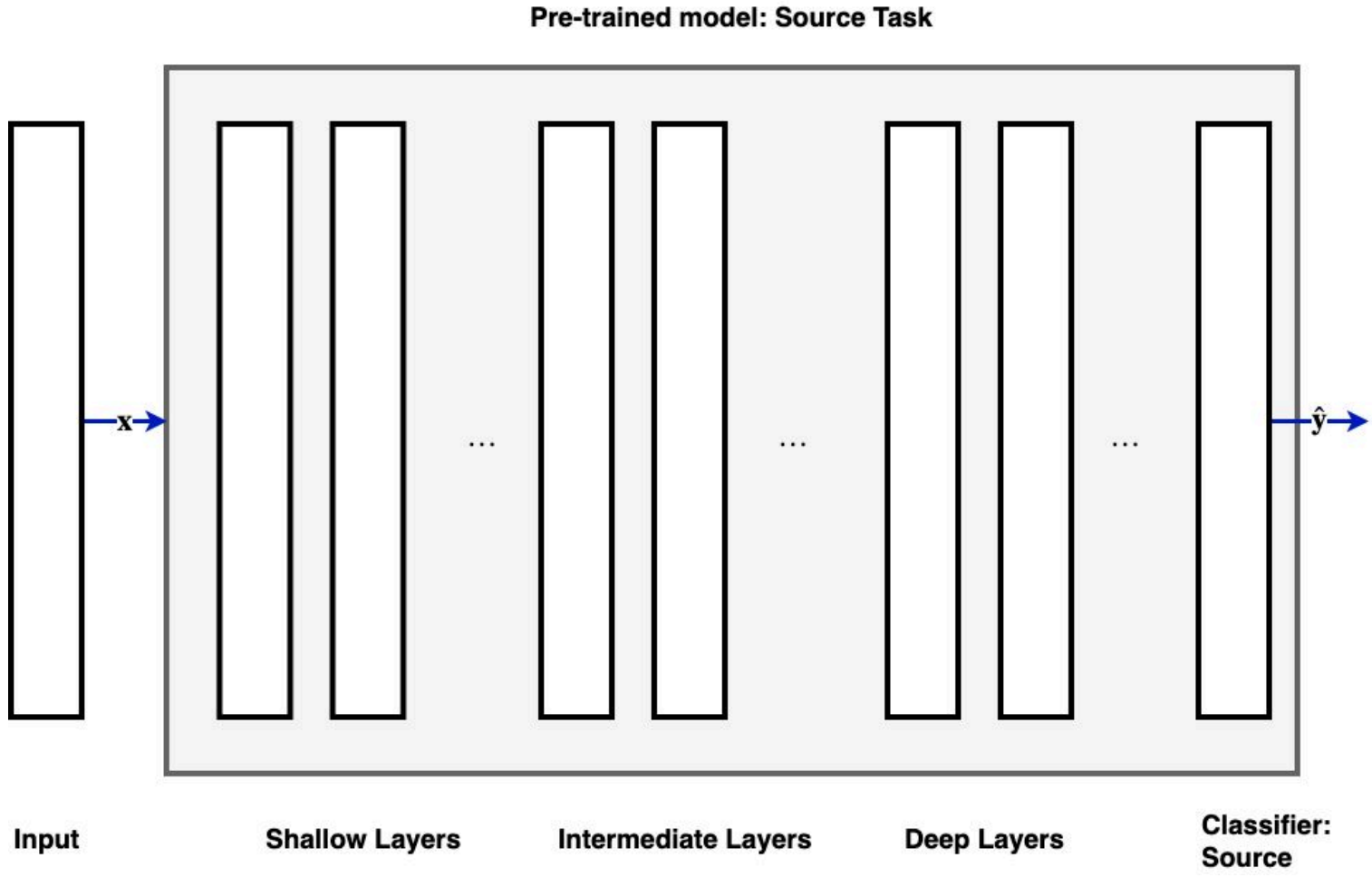
*Transfer Learning* creates a Neural Network for the new Target Task by

- Using some number of layers (closest to input) of a *pre-trained* model for some Source Task
- Appending new *untrained* layers for the Target Task
    - e.g., final "head": regression, classification
- Learning weights for the new *untrained* layers by training on examples from the Target Task
    - Possibly modifying weights on the layers imported from the Source Task

Hopefully a picture will clarify.

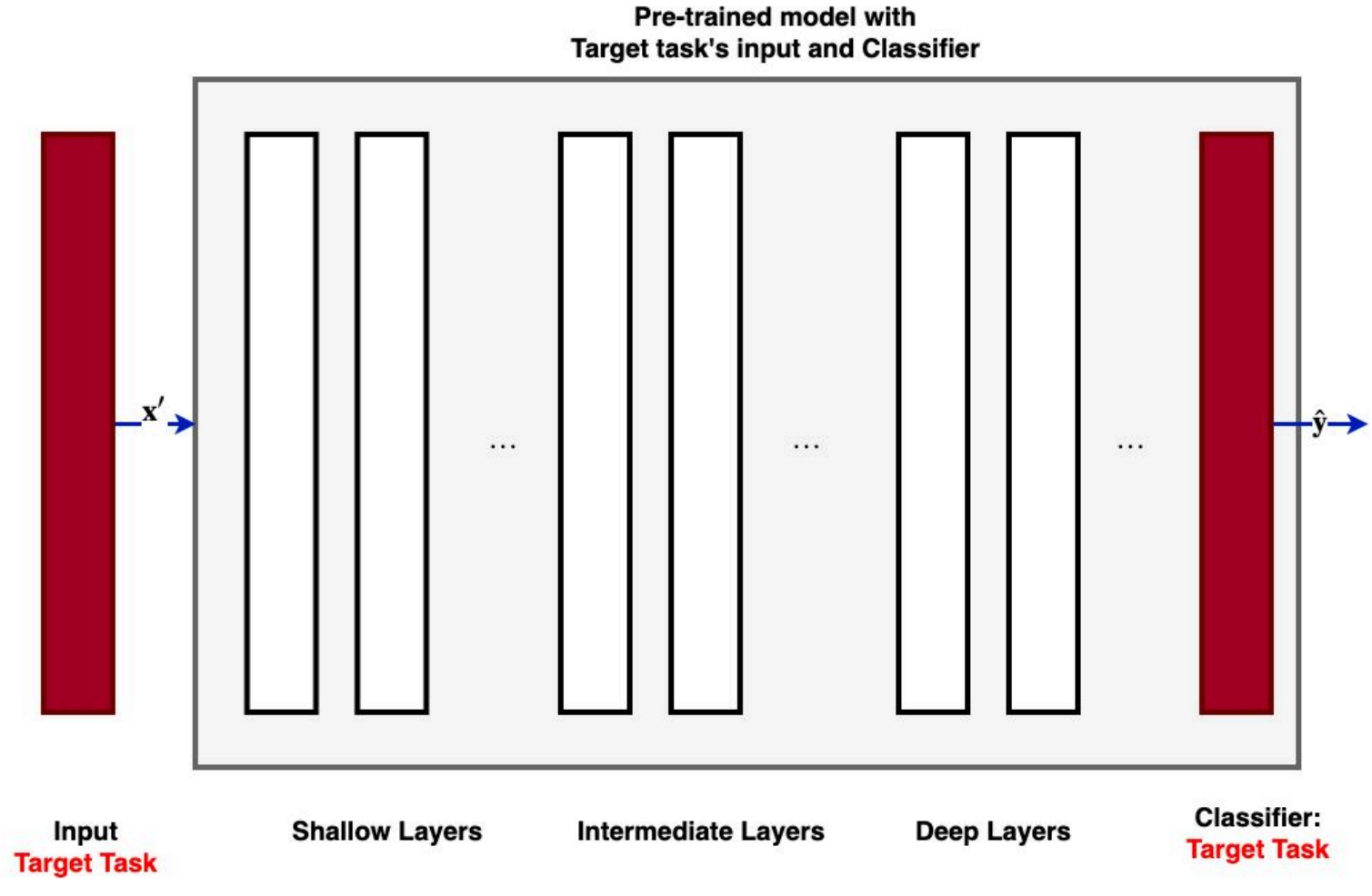
Here is the multi-layer network for the Source Task

- The Classifier "head" is specialized to the Source Task (e.g., the classes of the Source task)
- For example: recognizing images from 1000 possible classes


Pre-trained model: Source Task
x
...
...
...
ŷ
Input
Shallow Layers
Intermediate Layers
Deep Layers
Classifier: Source

And here

- We change the inputs from $\mathbf{x}$ (Source Task) to $\mathbf{x}'$ (Target Task)
- Replacing the Classifier for the Source Task
- With a Classifier specialized to the Target Task (e.g., the classes of the Target task)
- For example: recognizing images from 2 new classes distinct from the Source task's


Pre-trained model with
Target task's input and Classifier
x′
ŷ
Input
Target Task
Shallow Layers
Intermediate Layers
Deep Layers
Classifier:
Target Task

- We **import** all the weights from the Source Task layers
- We **retrain** the new Target Task Classifier Head using the training data for the Target Task

Transfer Learning

- Leverages all the effort in training the Source Task
- To benefit training the Target Task

This technique is most valuable when the model for the Source task has been trained

- With a **large** quantity of data
- For a **long** time

and when The Target task

- Has a **small** quantity of training data
- Limited computational resources

Why might this work ?

- The weights of the Source Task
- Have been laboriously trained
- To recognize "concepts"
- That transfer to other Tasks

Some examples may help

**Image Classification**

- Source Task: Label images to one of a 1000 possible classes (ImageNet)
- Target Task: Label images with classes **distinct** from the Source classes

Target task may have a very limited training data

**Self driving cars**

- Source Task: Frames from a video game driving simulator
- Target Task: Driving a real car

Real training data (actual driving) hard to obtain

- Simulation is easy

# Training the Target task

When we import the Source Task's layers

- We retain (**freeze**) their weights.
- While we train/learn weights for the newly appended suffix of Target Task layers (e.g., the new head)

Why is this a good idea ?

The newly appended layers for the Target Task

- Have **uninitialized** weights
- Which means that the gradients for the Loss function will likely be large at first

If we **did not** freeze the imported weights

- The imported layers would likely "forget" the concepts that were learned

Once the suffix has been sufficiently trained so that it's weights are no longer random

- We *may* unfreeze weights of the imported layers
- That are *closest to the new head*
- Using a *much lower* learning rate

In other words: we try to "fine-tune" the prefix.

Fine tuning may allow the imported layers to adapt to the Target task.

The key is to remember that

- The imported layers have been trained on **many** examples
- So their weights are less likely to need to adapt
- Than the weights of the suffix, which have been trained on **few** examples

Sometimes the learning rate is varied by layer

- Imported layers have low learning rates
- Suffix layers have higher learning rates
- Degenerate case: learning rate for imported layers is zero

There is no guarantee that the features learned by the Source Task will be useful for the Target task.

The chances may improve if the domains of the Source and Target tasks are more similar.

# Transfer learning in Keras

From a coding perspective, Transfer Learning looks something like this:

target_model = Sequential() # Import the prefix of source_model # - Import the architecture # - and the weights # Freeze the imported weights for layer in source_model.layer[:num_prefix_layers]: target_model.add( layer, trainable=False ) # Add the Suffix of the target task to the model target_model.add( ... )

The Keras Sequential model is similar to an array of layers

- We can index into the array to obtain slices (the prefix)
- We can append to the array to add the new suffix

We will show some real code shortly.

# How to choose the prefix of the Source task

It is not necessary (nor advisable) to keep **all** the layers of the Source Task's model

A smaller prefix might be better.

But where should we truncate the Source task's model ?

If the prefix is too small

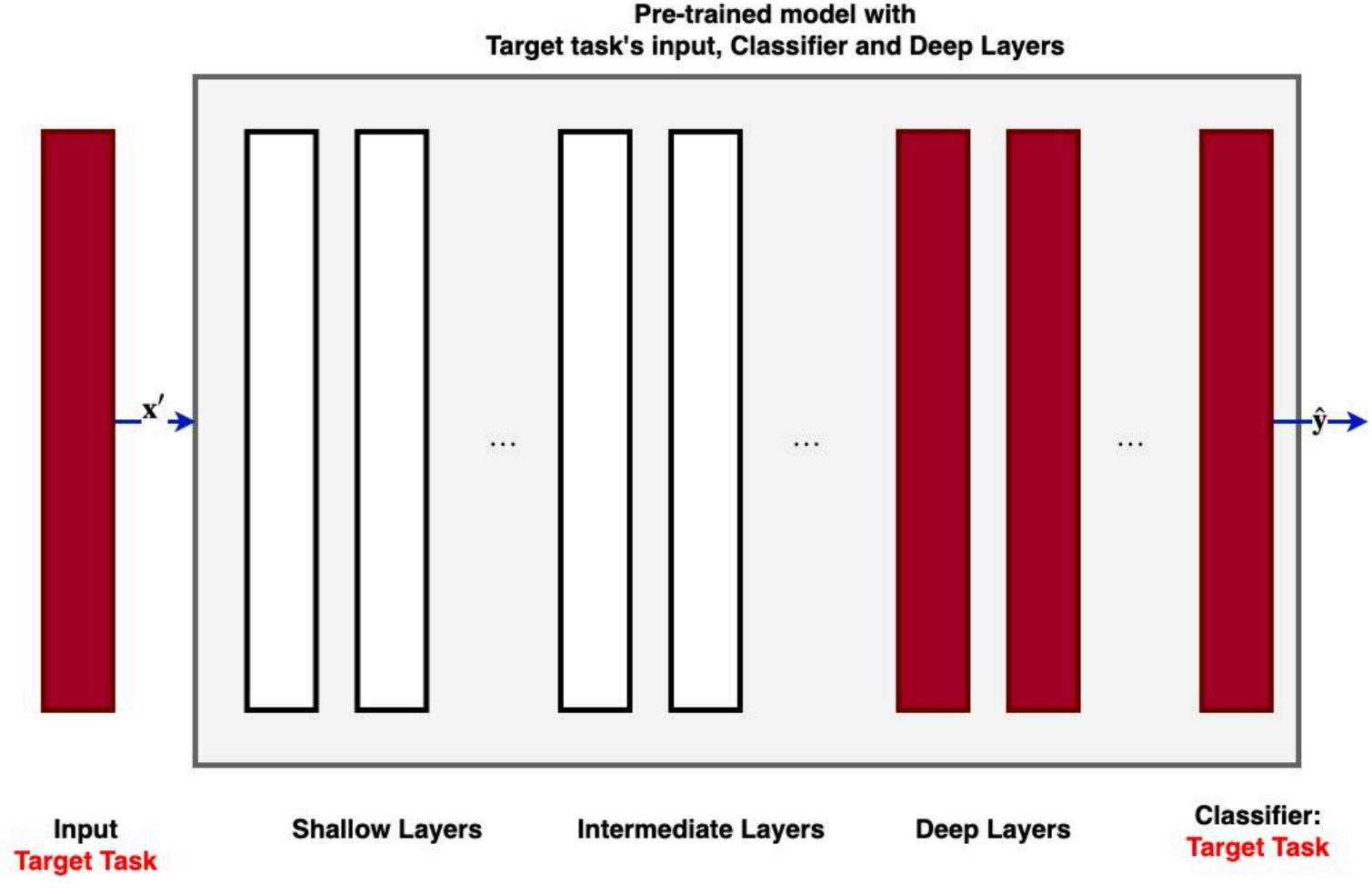
- That is: we retain only the shallow layers of the Source Task's model
- We may not benefit from the fullness of the concepts learned by the Source Task
    - Similar to underfitting

On the other hand, if the prefix is too large

- That is: we retain very deep layers of the Source Task's model
- We may have concepts that are so *specialized* for the Source Task
- That they fail to benefit the Target Task

The best advice: experiment !

We will do exactly that in the example notebook for this module.


Pre-trained model with
Target task's input, Classifier and Deep Layers
x′
...
...
...
ŷ
Input
Target Task
Shallow Layers
Intermediate Layers
Deep Layers
Classifier:
Target Task

# Conclusion

Transfer learning is a method to make you highly productive

- Leverage an existing model that may have been very expensive to train
    - Revolutionized Image Processing and Natural Language Processing
- "Cut off the head" and retrain *new head* on smaller number of examples

But there is still an element of art in knowing how much of the head to cut off

- Deeper layers may have over-specialized; best to cut them off
- Shallower layers may only recognize generic features; best to keep more of them

The Keras Guide to Transfer Learning & Fine-Tuning (https://keras.io/guides/transfer_learning/) is a good reference for further information.

In [4]:
```
print("Done")
```

```
Done
```