Transfer Learning with Neural Networks

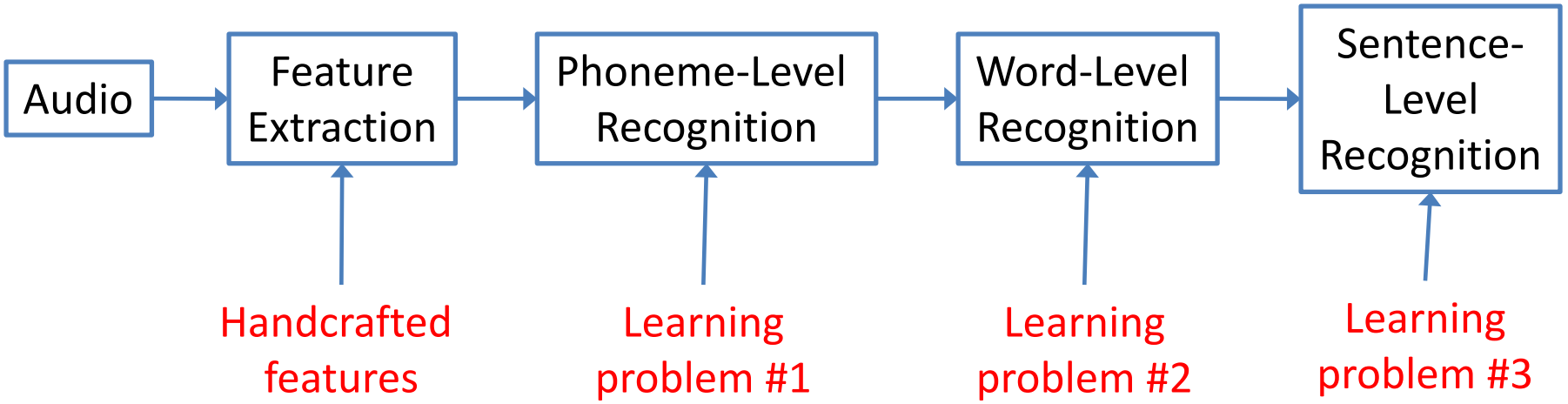
CSE 4311 – Neural Networks and Deep Learning
Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

What is “Deep” in Deep Learning

- To understand what is “deep” about deep learning, we should see how things were done back in the age of “shallow” learning.
- Example: speech recognition in the old days.

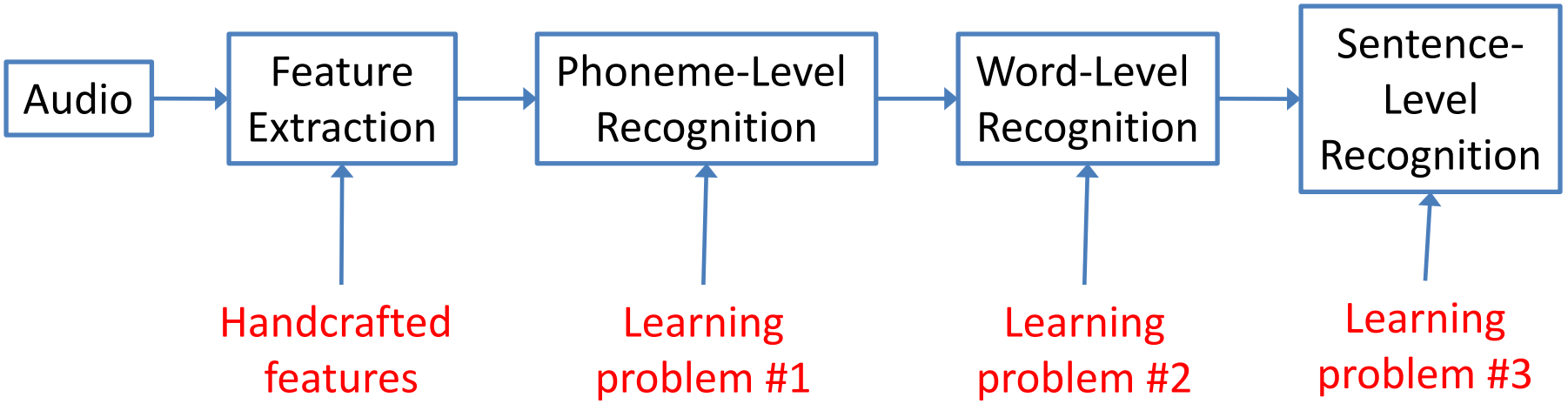
Shallow Learning

- To understand what is “deep” about deep learning, we should see how things were done back in the age of “shallow” learning.
- Example: speech recognition in the old days.



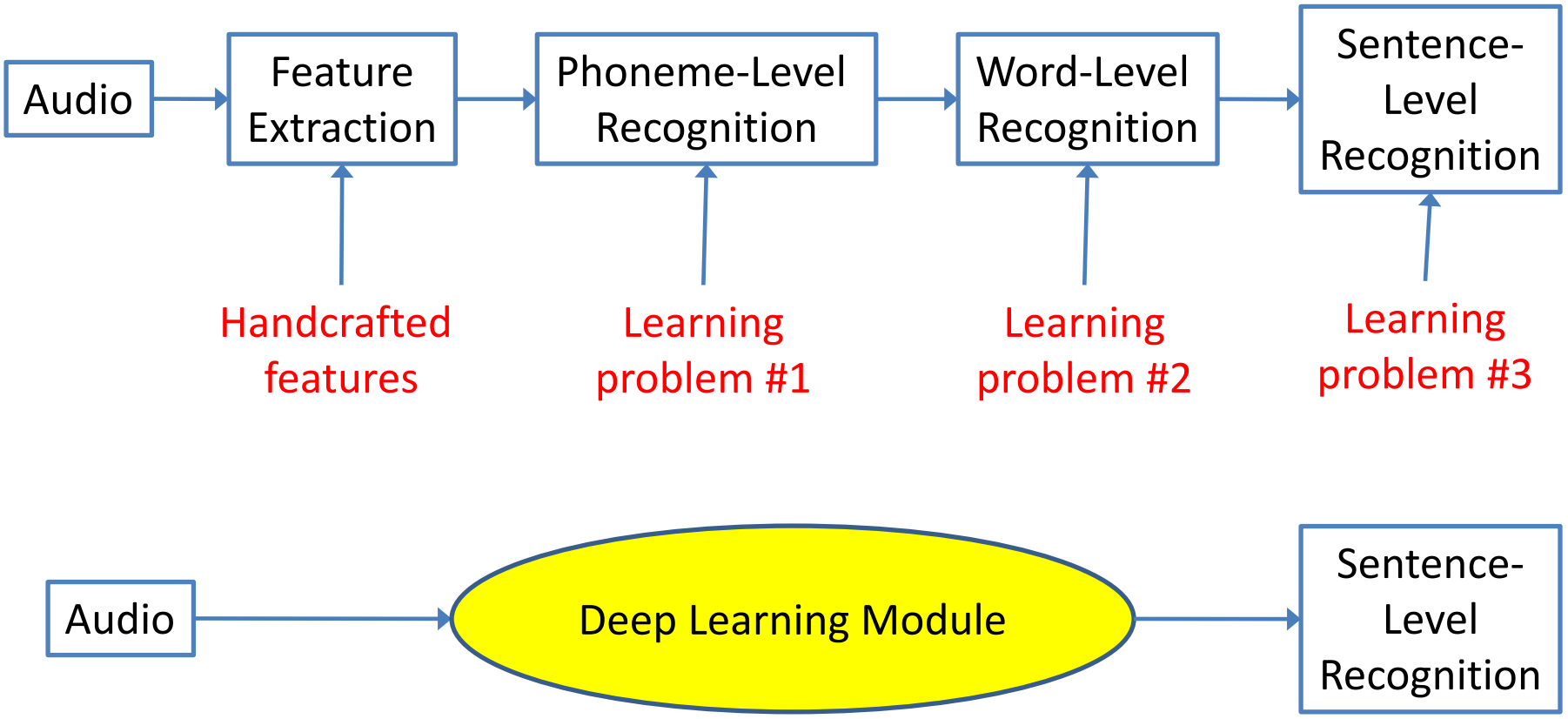
Shallow Learning

- To understand what is “deep” about deep learning, we should see how things were done back in the age of “shallow” learning.
- Example: speech recognition in the old days.



- The big problem was decomposed into smaller problems.
- A lot of manual design was needed for this decomposition.

Shallow vs. Deep Learning



- In deep learning, we try to build a single "deep" model for the entire end-to-end processing that maps input to output.
- Using a deep model eliminates the need to manually define and train separate subsystems for different parts of the process.

Deep Learning and Neural Networks

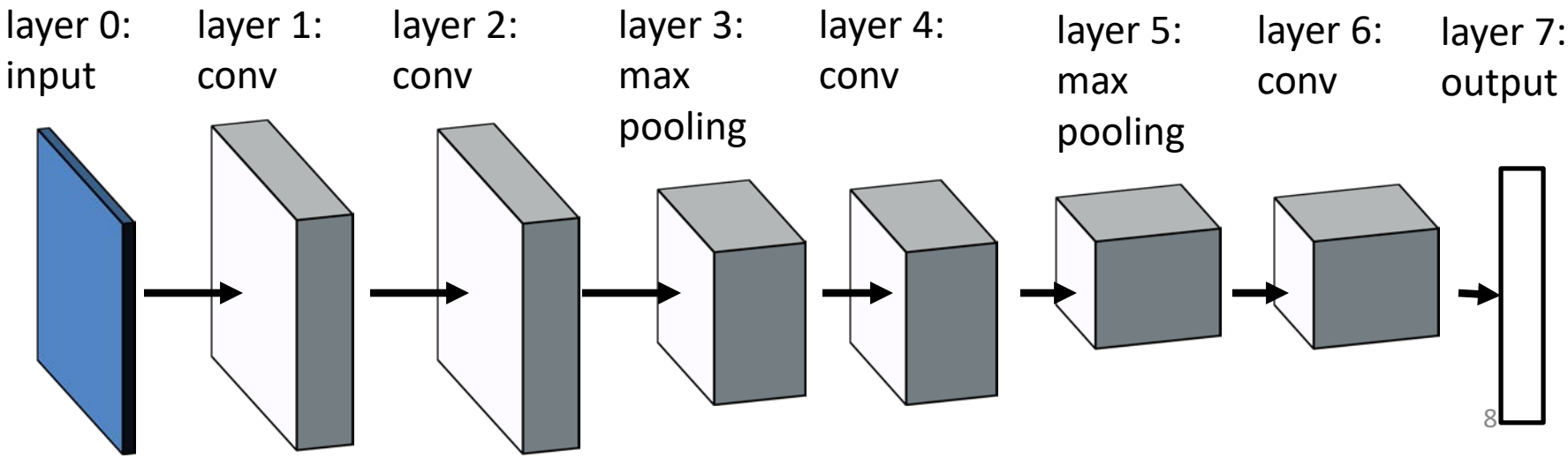
- In practice, deep learning models are neural networks.
- However, neural networks have been around since the 1960s.
  - The deep learning wave started around 2010.
- So, deep learning and neural networks are related, but not synonyms.
- Deep learning has two meanings, that in practice overlap a lot.
  - One meaning: a learning method that learns a “deep” model, as in our speech recognition example, that maps the input to the desired output.
  - Another meaning: a “deep” neural network, with “many” hidden layers.

Deep Learning and Neural Networks

- Naturally, terms like “deep model” and “many layers” are vague, there is no standard “threshold” that defines these terms.
  - The AlexNet model from 2012 had 7 hidden layers, and achieved 15.3% top-5 error on the ImageNet dataset.
  - The ResNet model from 2016 had 152 hidden layers and achieved 3.6% top-5 error on the ImageNet dataset.
- More layers do not automatically mean better performance.
  - There is always a risk that the model is too complicated and it overfits the training data.
  - Specific innovations were used in ResNet to obtain superior performance with that many layers.

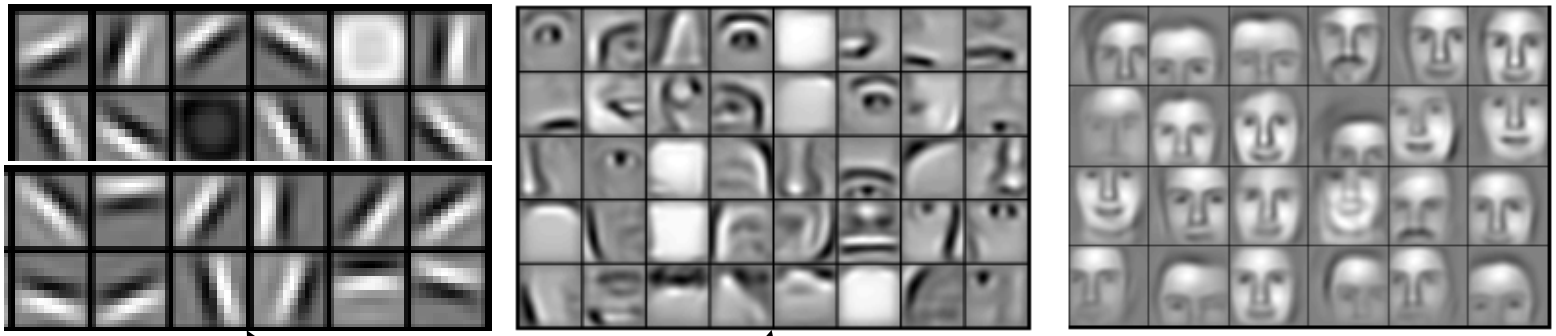
Example of a CNN Layout

- This is an example of a CNN with six hidden layers.
  - Four convolutional layers with stride 1.
  - Two max pooling layers.



CNN Visualization

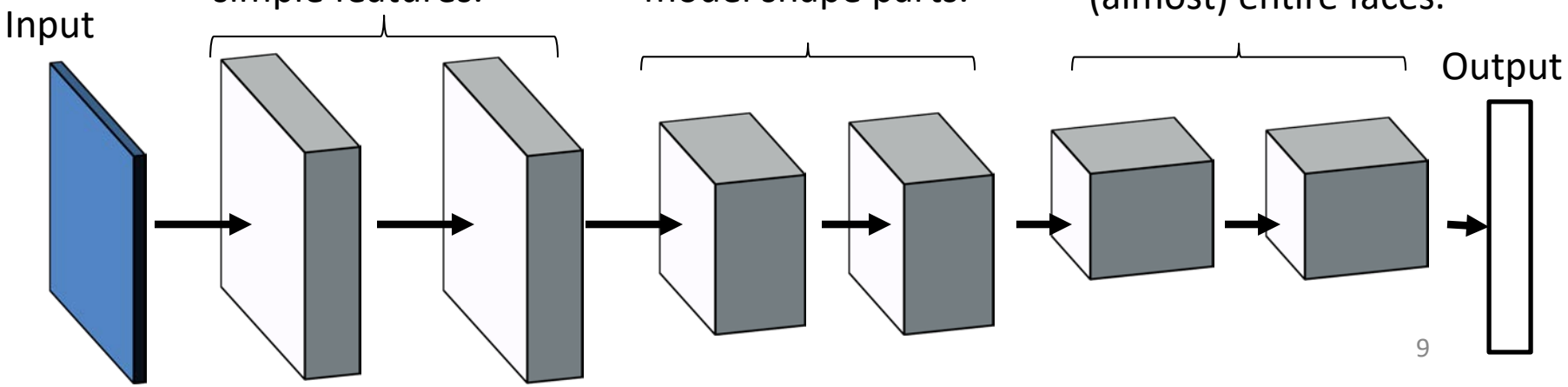
- Visualization of a deep neural network trained with face images.
  - Credit for feature visualizations: [Lee09] Honglak Lee, Roger Grosse, Rajesh Ranganath and Andrew Y. Ng., ICML 2009.



Early layers, extract simple features.

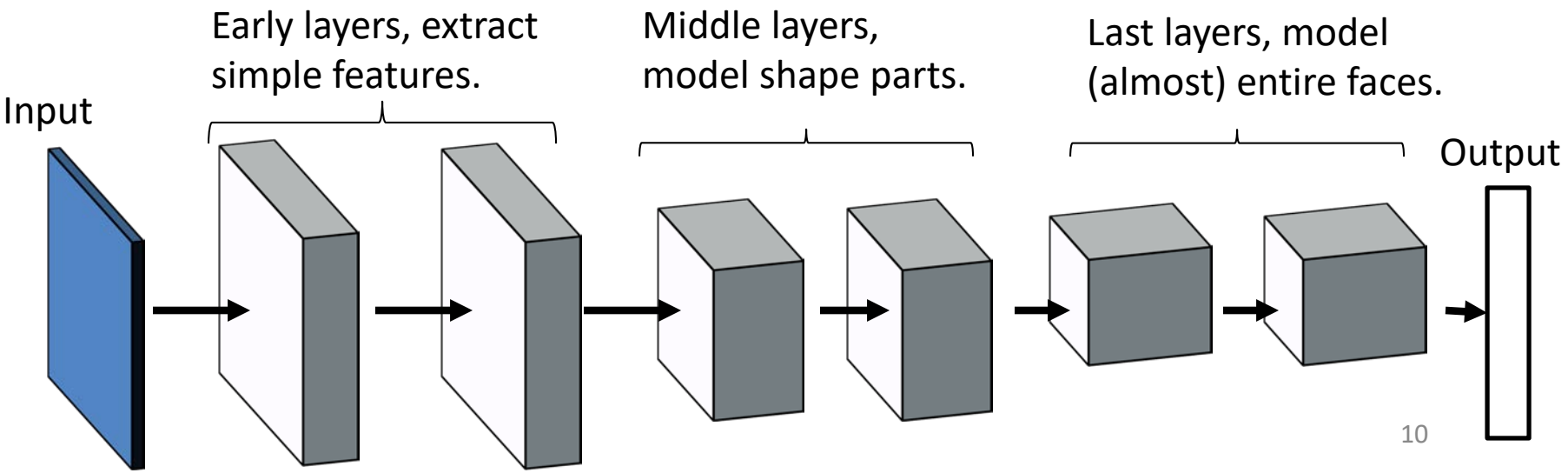
Middle layers, model shape parts.

Last layers, model (almost) entire faces.



Significance of Multiple Layers

- This example illustrates how “deep” neural networks make “deep” learning possible.
  - In the past, it would be common for people to build and train separate systems for detecting features at multiple levels of complexity.
  - The layers of a deep neural network behave like these separate systems.
  - Deep neural networks simplify building a system, as we do not have to worry about building all these subsystems separately.

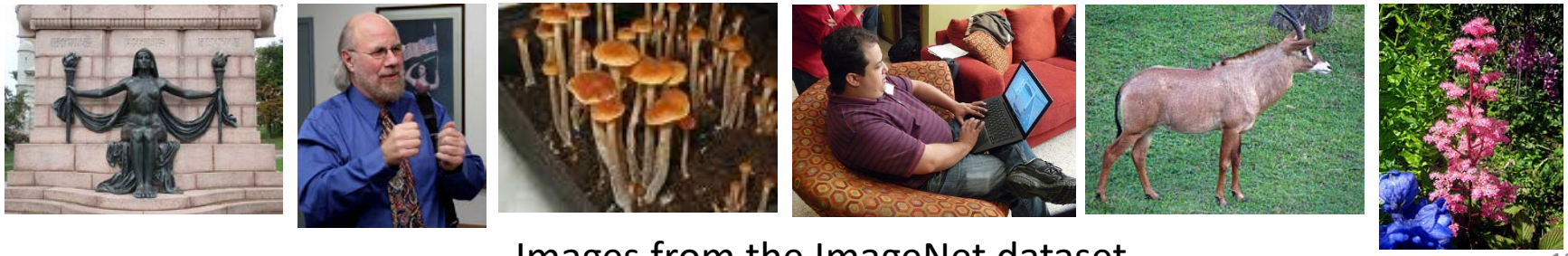


Transfer Learning

- Our next topic is to see how we can use deep neural networks to do what is called **transfer learning**.
- In transfer learning, we are typically working with two datasets:
  - A **source dataset**, that typically contains a large amount of data.
  - A **target dataset**, that is typically smaller, and contains classes that do not appear in the source dataset.
  - The question we try to address is: can we use the information in the source dataset to improve classification accuracy on the target dataset?
- If we do not try to do transfer learning, the default approach is to simply use the training examples available on the target dataset.
- So, the goal in transfer learning is to achieve better classification accuracy than this default approach.

Popular Source Dataset: ImageNet

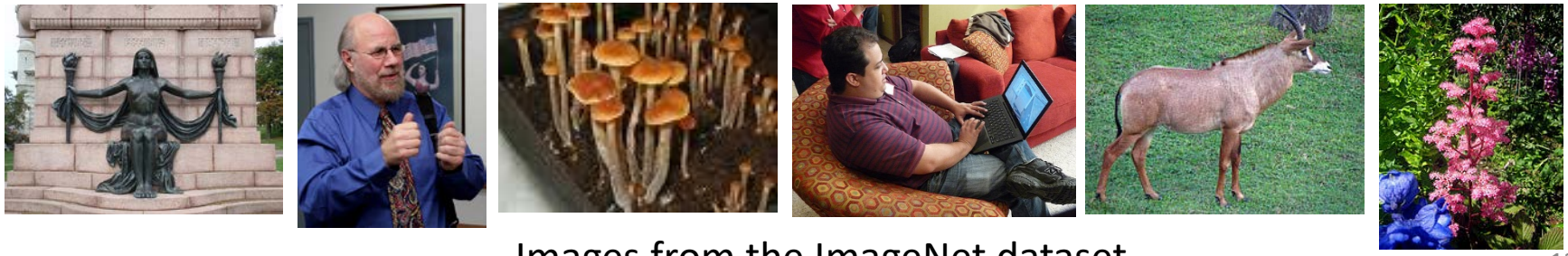
- ImageNet is a public dataset has over 14 million images, over 20,000 classes (such as “balloon” or “strawberry”).
  - http://www.image-net.org/
  - https://en.wikipedia.org/wiki/ImageNet
- Before deep learning: “top-5” error rate over 25%.
  - “Top-5” error rate is the percentage of test objects where the correct answer was NOT included in the top five answers.
- 2012: deep learning (CNN) model, 15.3% error rate.
  - Reference: A. Krizhevsky, I. Sutskever, and G. E. Hinton, NIPS 2012.
- As of 2021: the best top-5 error is close to 2%.



Images from the ImageNet dataset.

Transfer Learning

- Suppose that you have some training images of three animals that were recently discovered in a scientific expedition.
  - Let's say, 100 images for each animal.
  - You want a classifier that recognizes each of the three animals.
- Is the ImageNet data useful?
  - ImageNet does not contain images of those three animals, since they have just been discovered.



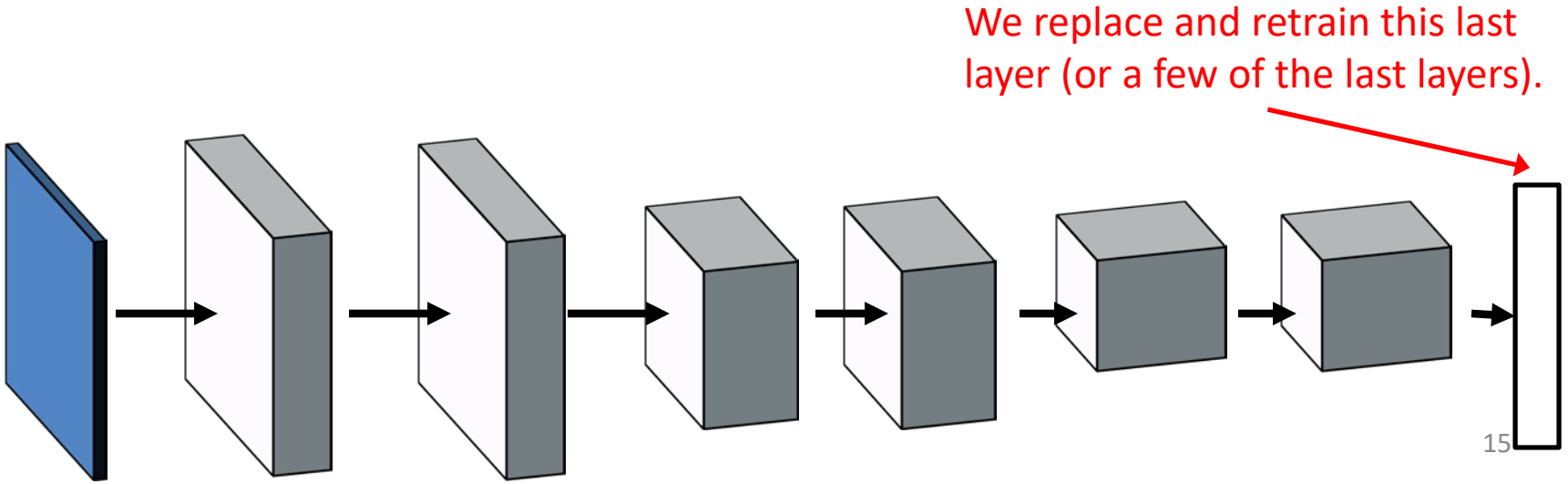
Images from the ImageNet dataset.

Transfer Learning

- Simple approach: train a classifier on your 300 images.
- 300 images is a rather small dataset, so you would use a relatively simple model.
  - Complicated models would probably overfit.
  - At the same time, a simple model would probably have limited accuracy.

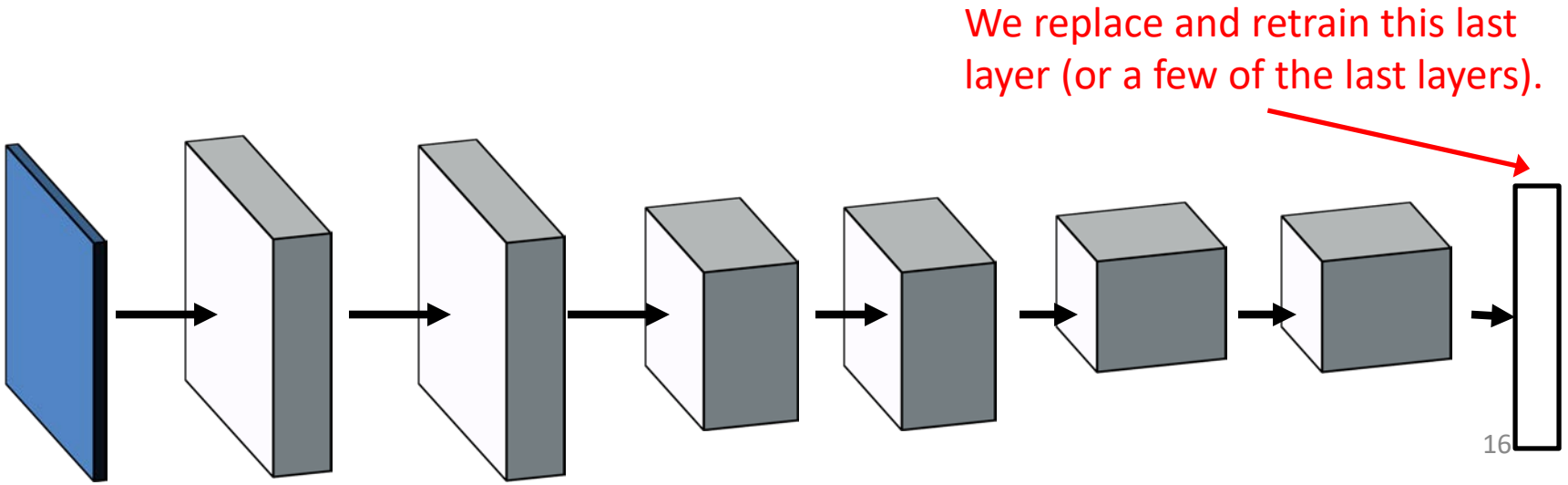
Transfer Learning

- Transfer learning approach:
  - 1st step: train a CNN on ImageNet.
    - Actually, these days you can just download a pre-trained model, so you do not have to do the training yourself.
  - 2nd step: throw away the output layer of the CNN.
    - The last layer recognizes ImageNet classes, which you don't care about.



Transfer Learning

- Transfer learning approach:
  - 1st step: train a CNN on ImageNet (or download a pre-trained model).
  - 2nd step: throw away the output layer of the CNN.
  - 3rd step: create a new output layer with three units, and connect it to the last hidden layer of the pre-trained CNN.
  - 4th step: train **just the weights of the new output layer** with your new dataset.



Transfer Learning

- Why is it called “transfer learning”?
  - Because, in order to learn a classifier for our three animals, we are transferring information learned from training examples (ImageNet) that did not include those three animals.
  - So, to recognize some classes, we use (partly) training examples from other classes.
- What information are we transferring?
  - All the layers and weights of the ImageNet-trained model, except for the output layer.

Transfer Learning

- Why would transfer learning be useful?
- The last hidden layer of the pre-trained model is used to recognize over 20,000 classes.
- Presumably, that last hidden layer computes features that are useful for recognizing a very wide variety of visual objects and shapes.
- Those features have been optimized using millions of training examples.
- Transfer learning assumes that those features would be useful for our three animals as well.

Transfer Learning

- Overall, we have a trade-off:
- 1st choice: Learn a model from scratch, using our three hundred images.
  - The model will be optimized exclusively for the three animals we want to recognize.
  - However, it will be a simple model, trained on a small training set.
- 2nd choice: Transfer learning.
  - Most of the model is optimized using examples that do not contain our three animals.
  - However, the features extracted by that model have been heavily optimized, and may be more useful than what we can learn from just 300 examples.

Example Code: tl_mnist.py

- File tl_mnist.py shows an example of how we can do transfer learning in Keras.
- We use the MNIST dataset as our original source of data.
- We have a “big” dataset, that contains all the training examples for classes “0”, “2”, “4”, “6”, “8”.
  - Total: 29492 training examples.
  - About 6,000 training examples per class.
- We have a “small” dataset for classes “1”, “3”, “5”, “7”, “9”.
  - Total: 100 training examples.
  - About 20 training examples per class.
- Goal: get the best classification accuracy we can get for classes “1”, “3”, “5”, “7”, “9”.

Example Code: tl_mnist.py

- We evaluate two options:
- Option 1 (no transfer learning): train a model using only the training examples of the “small” dataset.
- Option 2 (transfer learning): train a model on the large dataset, refine on the small dataset.
- These are the results:
  - Option 1, 15 trials, gave classification accuracies between 84.0% and 87.3%.
  - Option 2, 15 trials, gave classification accuracies between 91.0% and 92.6%.
- These results illustrate how information from a large dataset can be used to improve performance in a small dataset, whose classes do not appear in the large dataset.

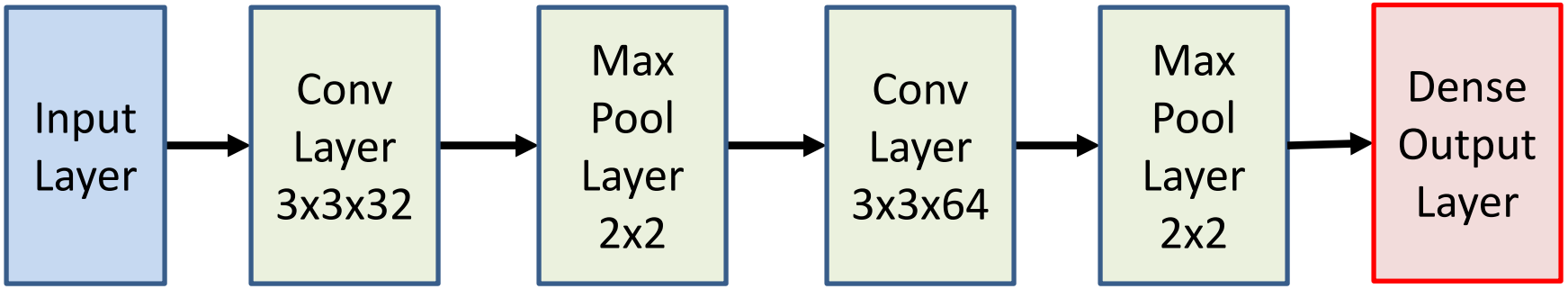
Example Code: tl_mnist.py

- We will take a look at some important parts of the code.
- The model is a relatively simple convolutional network.
  - Four hidden layers: two convolutional, two max pooling.

```
model = keras.Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_big_classes, activation="softmax"),
    ])
```

Example Code: tl_mnist.py

- This is a diagram illustrating the model.
- You should get used to reading such diagrams and converting them to Keras code.



Option 1: No Transfer Learning

```
model.compile(loss=keras.losses.SparseCategoricalCrossentropy(),
        optimizer="adam", metrics=["accuracy"])

model.fit(small_train_inputs[0:train_size],
        small_train_labels[0:train_size],
        epochs=100, batch_size=4)
test_loss, test_acc = model.evaluate(small_test_inputs, small_test_labels,
verbose=2)
print('\nTest accuracy: %.2f%%' % (test_acc * 100))
```

- Here we just train the model on the small dataset (100 training examples in total).
- 100 epochs, batch size = 4.
- 15 trials, gave classification accuracies between 84.0% and 87.3%.

Option 2: Transfer Learning

- To do transfer learning, we first train the model on the big dataset.
  - We train the model for 15 epochs, with a batch size of 128.
- Notice the call to **model.save()**, which saves the model to the specified file.
  - Saving models is useful when training takes a long time. Then, instead of training every time we run our code, we can just load a model we trained earlier.

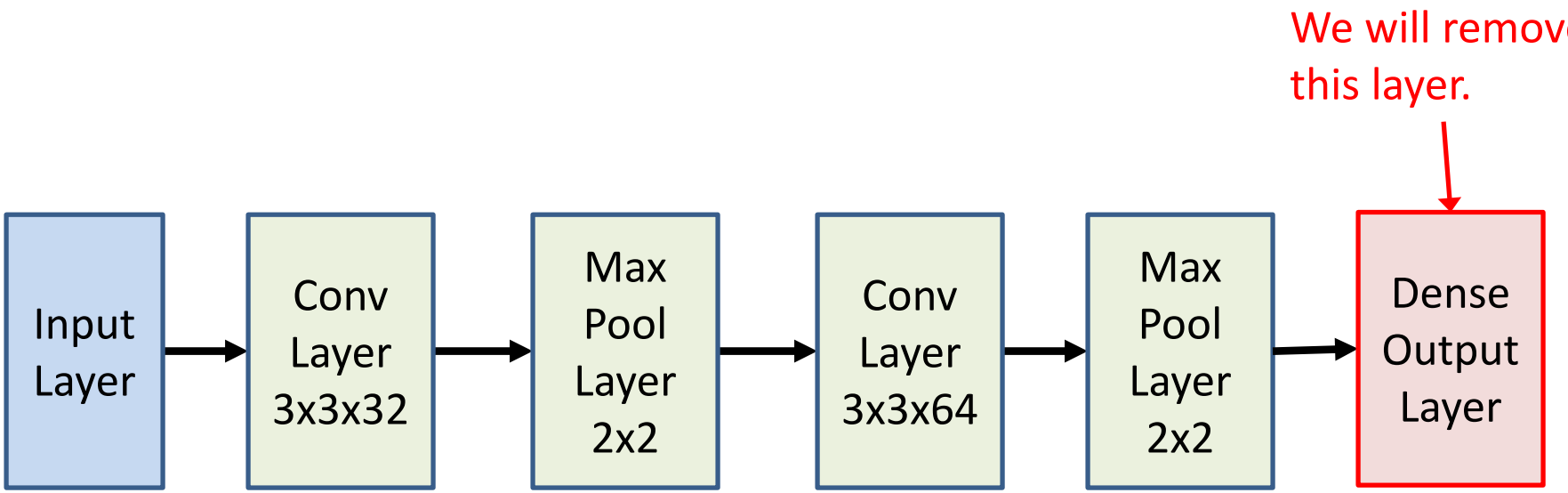
```
model.compile(loss=keras.losses.SparseCategoricalCrossentropy(),
optimizer="adam", metrics=["accuracy"])

model.fit(big_train_inputs, big_train_labels, epochs=15, batch_size=128)
model.save('mnist_5_15.h5')
```

Loading a Pre-Trained Model

- Using **load_model()** we load the model we had trained.

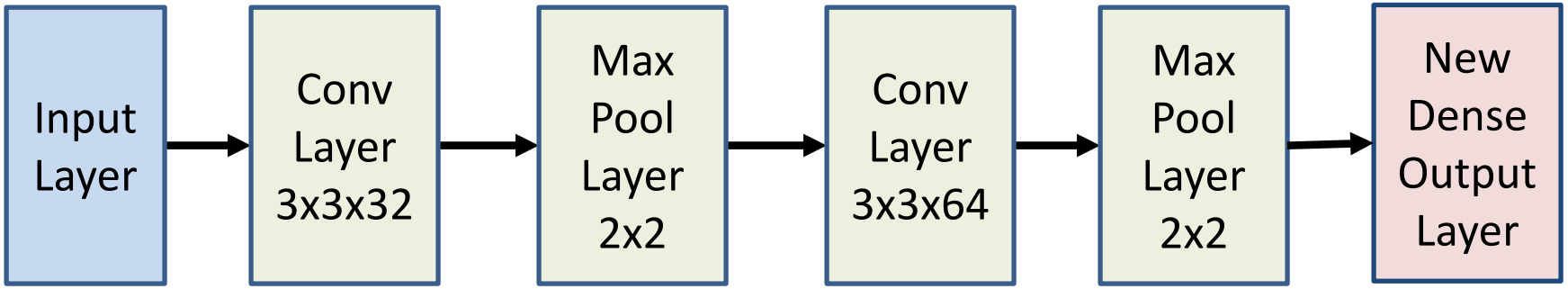
```
model = keras.models.load_model('mnist_5_15.h5')
```



Creating the New Model

```
num_layers = len(model.layers)
small_num_classes = len(small_classes)
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                    model.layers[0:num_layers-1]+
                    [layers.Dense(small_num_classes, activation="softmax")])
```

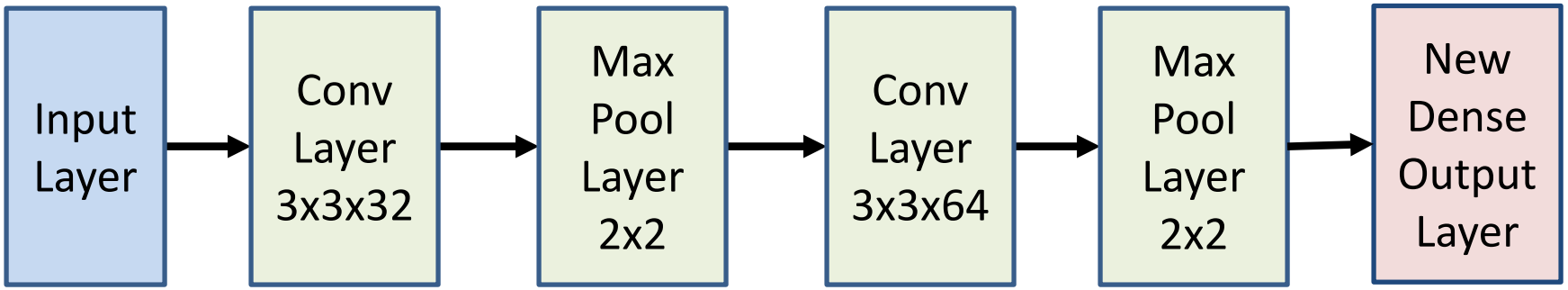
- The code above creates the new model.
- We call **keras.Sequential()**, and we specify the list of layers.



Creating the New Model

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                                 model.layers[0:num_layers-1]+
                                 [layers.Dense(small_num_classes, activation="softmax")])
```

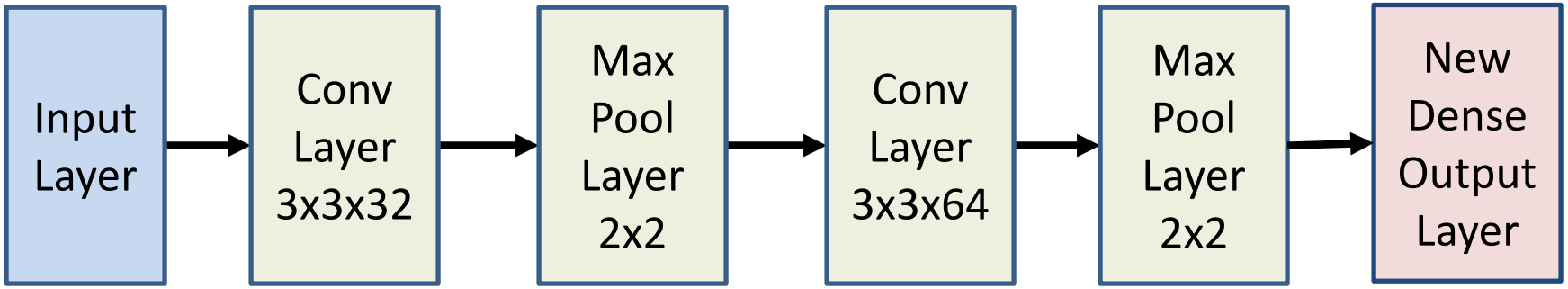
- Let's see what we have included in the list of layers that we pass to **Sequential()**.
- The first layer in the new model is the input layer.



Creating the New Model

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                    model.layers[0:num_layers-1]+
                    [layers.Dense(small_num_classes, activation="softmax")])
```

- After the input layer, we insert all layers of the pre-trained model, **except for the last layer**.
  - The layers we copy from the pre-trained model are shown in green.
- Here is a tricky Keras feature: doesn't the model include an input layer? Why did we include the input layer explicitly?



Layers of the New Model

```
model = keras.models.load_model('mnist_5_15.h5')
model.summary()
```

Output:

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
conv2d_18 (Conv2D)           (None, 26, 26, 32)        320
max_pooling2d_18 (MaxPooling2D)  (None, 13, 13, 32)        0
conv2d_19 (Conv2D)           (None, 11, 11, 64)        18496
max_pooling2d_19 (MaxPooling2D)  (None, 5, 5, 64)          0
flatten_9 (Flatten)          (None, 1600)              0
dropout_9 (Dropout)          (None, 1600)              0
dense_9 (Dense)              (None, 5)                 8005
=================================================================
```

- Notice that the summary of the pre-trained model does NOT include the input layer.

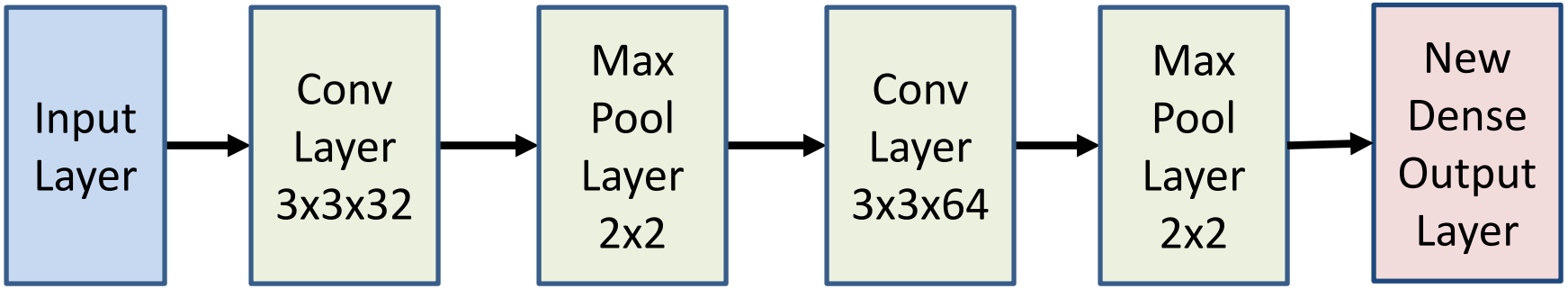
Detour: Input Layers in Keras

- When you call **keras.Sequential()** to define a model, starting with a **keras.Input** layer is **optional**.
- The model will automatically figure out the shape of the input when it is given inputs for the first time (for example, when training starts).
- Only (minor) inconvenience: you cannot call **model.summary()** until the model has figured out the input shape.
- The **summary()** method prints out (and thus needs to know) the output shape and number of weights for each layer.
  - That cannot be known if the input shape is not known.
  - So, to call **summary()**, either you specify the input shape in advance, or you wait until the model figures it out on its own.

Layers of the New Model

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                                 model.layers[0:num_layers-1]+
                                 [layers.Dense(small_num_classes, activation="softmax")])
```

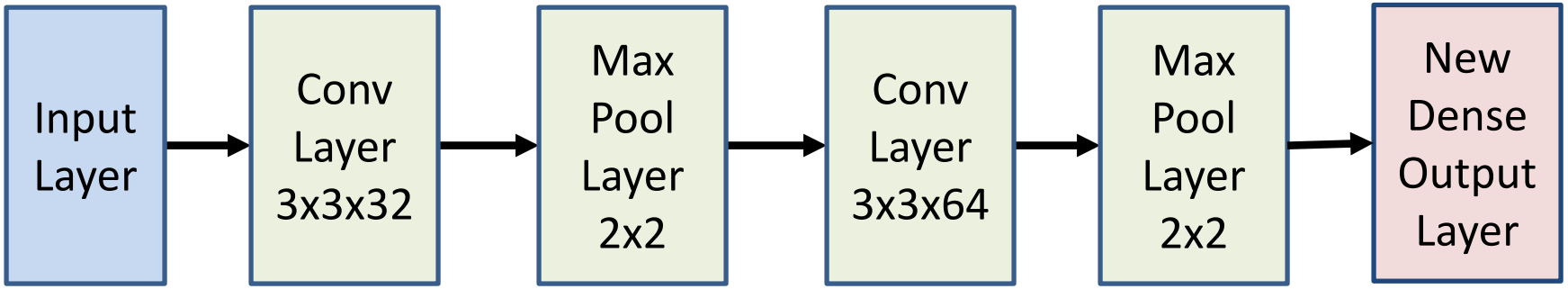
- After we insert all layers of the pretrained model except for the output layer, we create a new output layer.
- Question: what are the weights of these layers?



Layers of the New Model

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                    model.layers[0:num_layers-1]+
                    [layers.Dense(small_num_classes, activation="softmax")])
```

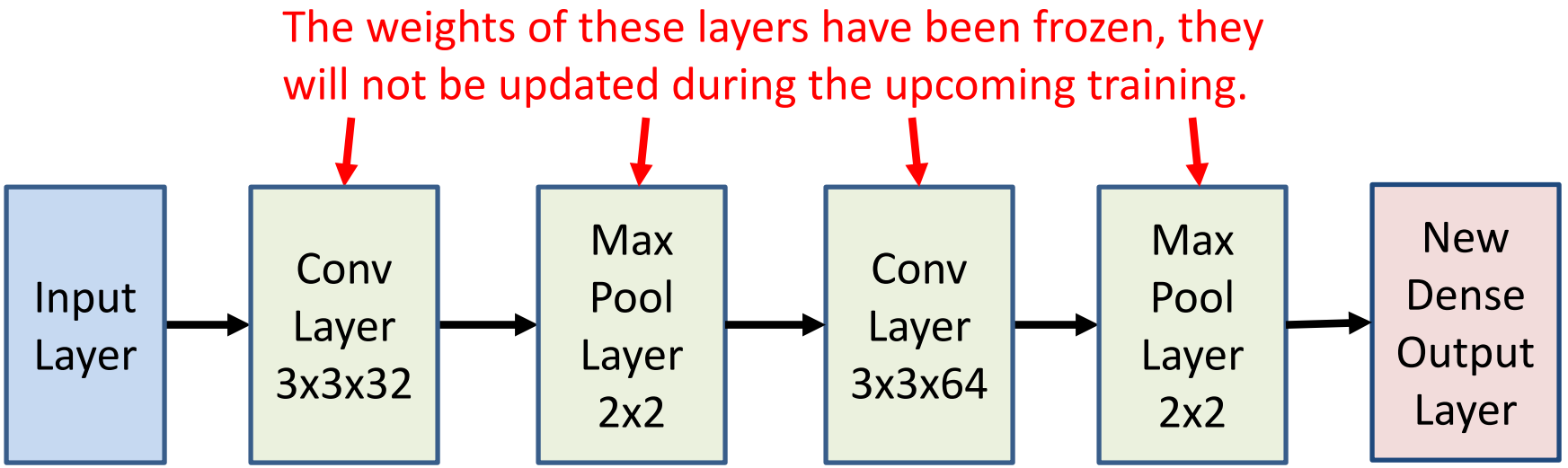
- Question: what are the weights of these layers?
- The hidden layers come from the pre-trained model, so their weights have been trained.
- The weights of the new output layer are randomly initialized.



Freezing Pre-Trained Layers

```
for i in range(0, num_layers-1):
    refined_model.layers[i].trainable = False
```

- This piece of code **freezes the weights** of the hidden layers.
- These weights are the information we **transfer** from the big dataset. We do not want to lose this information.



Trainable Parameters in **summary()**

refined_model.summary()

Output:

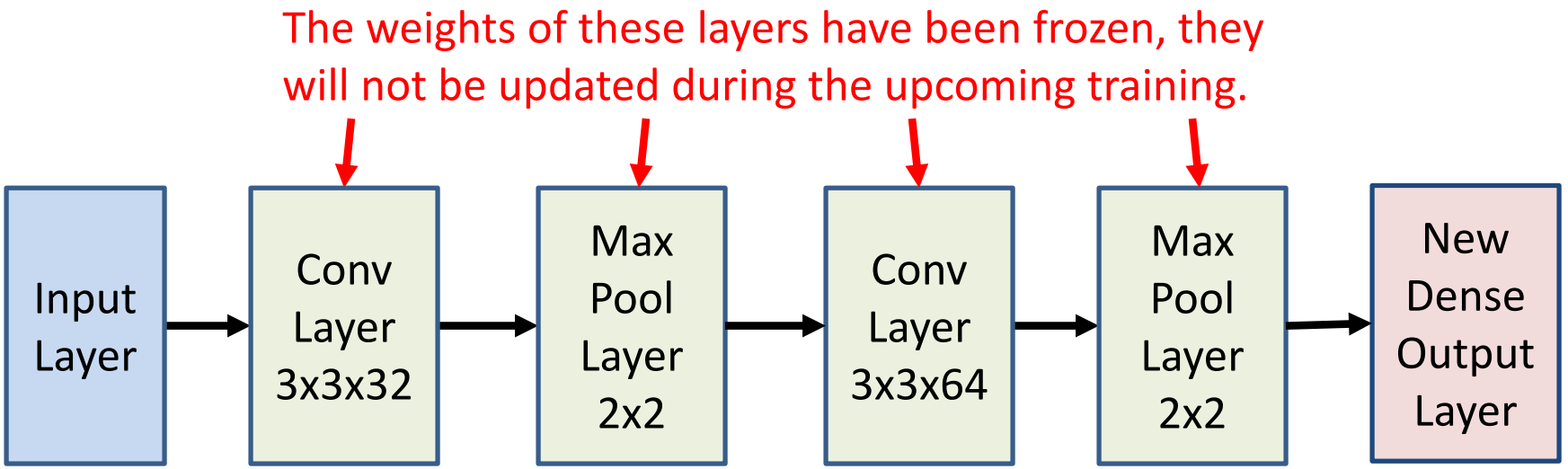
```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
conv2d_18 (Conv2D)           (None, 26, 26, 32)        320
max_pooling2d_18 (MaxPooling2D)  (None, 13, 13, 32)        0
conv2d_19 (Conv2D)           (None, 11, 11, 64)        18496
max_pooling2d_19 (MaxPooling2D)  (None, 5, 5, 64)          0
flatten_9 (Flatten)          (None, 1600)              0
dropout_9 (Dropout)          (None, 1600)              0
dense_9 (Dense)              (None, 5)                 8005
=================================================================
Total params: 26,821
Trainable params: 8,005
Non-trainable params: 18,816
```

- The last lines of the summary tell us how many parameters are trainable and non-trainable.
  - As expected, only the weights of the last layer are trainable.

Training on Small Dataset

```
refined_model.compile(loss=keras.losses.SparseCategoricalCrossentropy(),
        optimizer="adam", metrics=["accuracy"])
refined_model.fit(small_inputs, small_labels, epochs=100, batch_size=4)
```

- We train our model on the small dataset.
  - This training only updates the weights of the output layer.



Transfer Learning Results

- As a reminder, these are the results.
- Option 1, no transfer learning:
  - 15 trials, gave classification accuracies between 84.0% and 87.3%.
- Option 2, transfer learning:
  - 15 trials, gave classification accuracies between 91.0% and 92.6%.

Using a Pre-Trained Model: VGG-16

- VGG-16 is a model pre-trained on ImageNet, that is available on Keras.
- Introduced by this paper:

Karen Simonyan, Andrew Zisserman: “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015.

https://arxiv.org/pdf/1409.1556.pdf

- The VGG-16 model is a good example of a model that is:
  - Deep, having 16 learnable layers (i.e., convolutional or fully connected).
  - Making simple design choices that are easy to describe:
    - Convolutional filters have 3 rows and 3 columns, with stride 1.
    - Max pool layers operate on a 2x2 neighborhood.

VGG-16 Layers

Input Image
Convolutional Layer, 3x3, 64 filters
Convolutional Layer, 3x3, 64 filters
Max Pooling Layer, 2x2 neighborhood
Convolutional Layer, 3x3, 128 filters
Convolutional Layer, 3x3, 128 filters
Max Pooling Layer, 2x2 neighborhood
Convolutional Layer, 3x3, 256 filters
Convolutional Layer, 3x3, 256 filters
Convolutional Layer, 3x3, 256 filters
Max Pooling Layer, 2x2 neighborhood
Convolutional Layer, 3x3, 512 filters
Convolutional Layer, 3x3, 512 filters
Convolutional Layer, 3x3, 512 filters
Max Pooling Layer, 2x2 neighborhood

Convolutional Layer, 3x3, 512 filters
Convolutional Layer, 3x3, 512 filters
Convolutional Layer, 3x3, 512 filters
Max Pooling Layer, 2x2 neighborhood
Fully Connected, 4096 Units
Fully Connected, 4096 Units
Fully Connected, 1000 Units
Output Layer: SoftMax

Loading a Pre-Trained VGG-16 Model

```
vgg16 = keras.applications.vgg16.VGG16(
   weights="imagenet",
   include_top=False,
   input_shape=input_shape)
```

- Here we load the vgg16 model, which is predefined in Keras.
- The “weights” option specifies that we want weights that were learned using the ImageNet dataset.
- Note the “include_top” option.
  - If set to false, the returned model does NOT include the last four layers of VGG-16.
  - This way, we do not have to manually remove these last layers later.

VGG-16 on MNIST: mnist_vgg.py

- File mnist_vgg.py shows an application of the pre-trained VGG-16 model on the MNIST dataset.
- As before, we have a “small” training set of 100 examples from classes “1”, “3”, “5”, “7”, “9”.
  - We remove the last four layers of the pre-trained VGG-16.
  - We freeze the weights of the remaining layers.
  - We add one or a few new layers at the end (the code evaluates a few different options).
  - We train the weights of those layers using the 100 training examples from our “small” training set.

Some Models, and Results

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                    vgg16.layers +
                    [layers.Flatten(),
                    layers.Dense(small_num_classes, activation="softmax")])
```

- Accuracy rate: 83.5% - 84.6%

Some Models, and Results

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                                 vgg16.layers +
                                 [layers.Flatten(),
                                  layers.Dropout(0.5),
                                  layers.Dense(small_num_classes, activation="softmax")])
```

- Just adding a dropout rate of 50% on the layer right before the output layer.
- Accuracy rate: 79.4% - 81.9%
  - In this case, adding the dropout option led to lower classification accuracy.

Some Models, and Results

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                                 vgg16.layers +
                                 [layers.Flatten(),
                                  layers.Dense(512, activation="tanh"),
                                  layers.Dropout(0.5),
                                  layers.Dense(small_num_classes, activation="softmax")])
```

- Adding a fully connected hidden layer with 512 units.
  - This allows more “learning” to be done on the target dataset.
  - Of course, this also increases the risk of overfitting.
- Accuracy rate: 85.9% - 86.1%
  - In our case, adding the hidden layer did improve performance.

Some Models, and Results

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                                 vgg16.layers +
                                 [layers.Flatten(),
                                  layers.Dropout(0.5),
                                  layers.Dense(512, activation="tanh"),
                                  layers.Dropout(0.5),
                                  layers.Dense(small_num_classes, activation="softmax")])
```

- Here we use dropout in the two last hidden layers, as opposed to just the last hidden layer.
- Accuracy rate: 81.1% - 85.9%.
  - The extra dropout led to somewhat lower accuracy at the lower end.

Some Models, and Results

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                    vgg16.layers +
                    [layers.Flatten(),
                     layers.Dense(512, activation="tanh"),
                     layers.Dense(small_num_classes, activation="softmax")])
```

- Removing all dropout options from the previous model.
- Accuracy rate: 85.9% - 86.1%.
  - Similar to the accuracy we got when using dropout only for the last hidden layer.

Some Notes on mnist_vgg.py

```
temp = small_train_inputs
(num, _, _, _) = temp.shape
temp = np.repeat(temp, 3, axis=3)
small_train_inputs = np.zeros((num,32,32,3))
small_train_inputs[:,2:30,2:30,:] = temp
```

- When we use the VGG-16 layers, we need each input image to have:
  - Three channels.
    - To achieve that, the code above repeats the single channel of MNIST images three times.
  - At least 32 rows and 32 columns.
    - To achieve that, the code "pads" each image by adding some zero rows and columns at the top, bottom, left and right of the image.

VGG-16 on MNIST: mnist_vgg.py

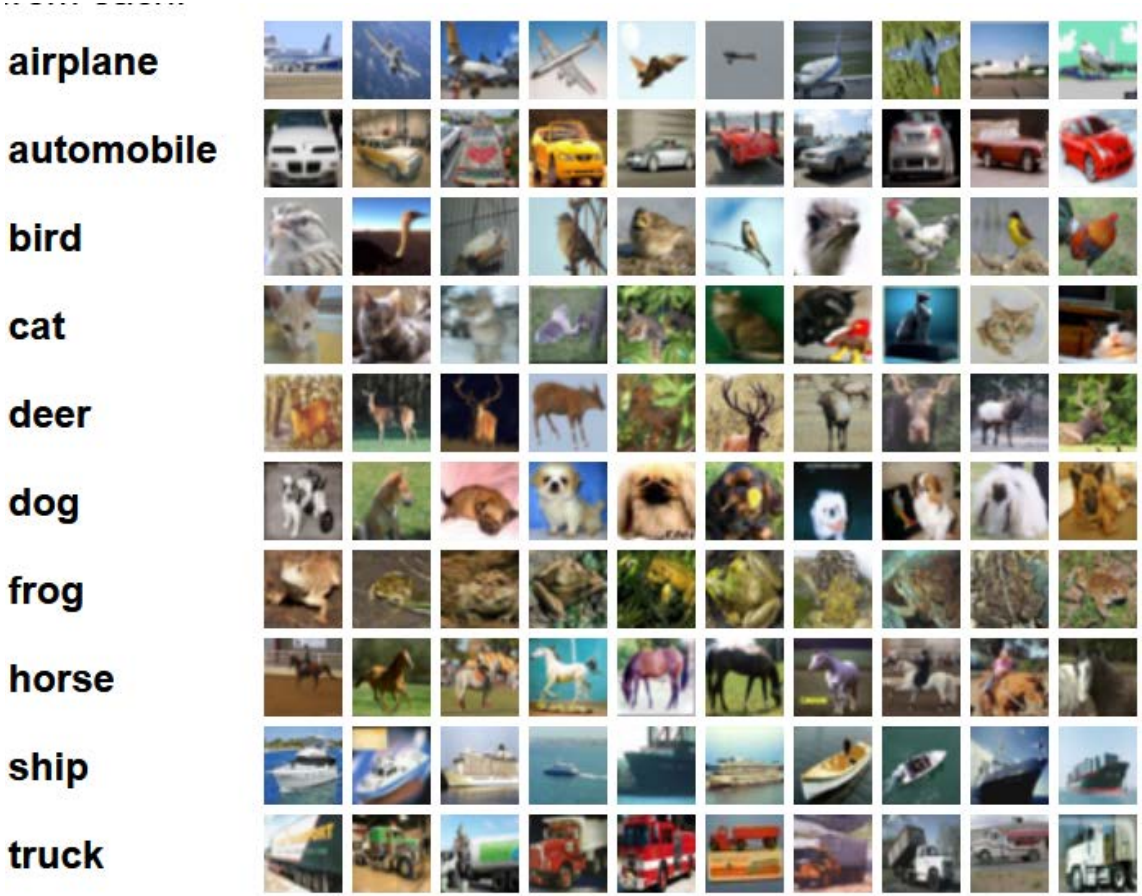
- Interestingly, this is a case where transfer learning does not work well.
- The best accuracy rate obtained using the options implemented in the code was 86.1%.
- It is interesting to compare this accuracy with the results that we saw earlier:
  - A simple CNN model with no transfer learning gave accuracy rates between 84% and 87%.
  - Pre-training a model on 29492 examples from classes “0”, “2”, “4”, “6”, “8” and re-training the output layer on our “small” training set gave accuracies between 91.0% and 92.6%.
- What might be the reasons that transfer learning using VGG-16 did not work here?

VGG-16 on MNIST: mnist_vgg.py

- What might be the reasons that transfer learning using VGG-16 did not work here?
- A caveat: we cannot eliminate the possibility that transfer learning might have worked better if we had used some different hyperparameters or design choices.
- Nonetheless, one thing to note is that the **source domain** had different characteristics than the **target domain**.
  - **Source domain** is the (typically larger) dataset used for pre-training the model.
  - **Target domain** is the (typically smaller) dataset where we refine and evaluate the model.
- Source domain: ImageNet, a dataset of RGB photographs.
- Target domain: MNIST, a dataset of grayscale handwritten digits.

Another Example: CIFAR10 Dataset

- CIFAR10 dataset: 60,000 color images, size 32x32, 10 classes.
- It is publicly available from: https://www.cs.toronto.edu/~kriz/cifar.html
- Here some example images, from the source website:



VGG-16 on CIFAR10: cifar_vgg.py

- We create a “small” training set by choosing 100 training examples per class.
- This gives a total of 1000 training examples.
- Again, we compare using transfer learning to not using transfer learning.
  - Option 1 (no transfer learning): train a model using only the training examples of the “small” dataset.
  - Option 2 (transfer learning): train a model on the large dataset, refine on the small dataset.
- Here, transfer learning gave indeed better results.
  - The best accuracy rates without transfer learning were 40.0%-41.4%.
  - The best accuracy rates using transfer learning were 48.0%-49.5%

cifar_vgg.py: Looking Into the Code

```
(training_set, test_set) = keras.datasets.cifar10.load_data()
(training_inputs, training_labels) = training_set
(test_inputs, test_labels) = test_set
training_inputs = training_inputs.astype("float32") / 255
test_inputs = test_inputs.astype("float32") / 255
np.savez("cifar10", training_inputs=training_inputs,
        training_labels=training_labels,
        test_inputs=test_inputs, test_labels=test_labels)
```

- We download the CIFAR10 data by calling **keras.datasets.cifar10.load_data()**
  - This can take a bit of time (downloading 180MB).
- We normalize the input values to be between 0 and 1.
- To save time, we save the data locally using the numpy **savez()** function.

cifar_vgg.py: Looking Into the Code

```
cifar_data = np.load("cifar10.npz")
training_inputs = cifar_data["training_inputs"]
training_labels = cifar_data["training_labels"]
test_inputs = cifar_data["test_inputs"]
test_labels = cifar_data["test_labels"]
```

- This code shows how to load our local copy of CIFAR10.

cifar_vgg.py: Looking Into the Code

```
model = keras.Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="tanh"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="tanh"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="tanh"),
        layers.Conv2D(64, kernel_size=(3, 3), activation="tanh"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),    ])
```

- This is the model we use for the no-transfer-learning option.
  - Accuracy: 40%-41.4%

cifar_vgg.py: Looking Into the Code

- For the no-transfer-learning option, you can also try using ReLU and sigmoid as activation functions for the hidden layers.
  - Accuracy using tanh for all hidden layers: 40%-41.4%
  - Accuracy using ReLU for all hidden layers: 36.4%-38.8%
  - Accuracy using sigmoid for all hidden layers: 37.3%-38.5%

cifar_vgg.py: Looking Into the Code

```
vgg16 = keras.applications.vgg16.VGG16(
   weights="imagenet",
   include_top=False,
   input_shape=input_shape)
```

- Here we load the vgg16 model, which is predefined in Keras.
- Note the “include_top” option.
  - We set it to false, to NOT include the last four layers of VGG-16.
  - This way, we do not have to manually remove the output layer later.

cifar_vgg.py: Looking Into the Code

```
refined_model = keras.Sequential([keras.Input(shape=input_shape)]+
                                 vgg16.layers +
                                 [layers.Flatten(),
                                  layers.Dropout(0.5),
                                  layers.Dense(512, activation="tanh"),
                                  layers.Dropout(0.5),
                                  layers.Dense(num_classes, activation="softmax")])
```

- Here we create our model.
- It includes the VGG-16 model (minus the layers we removed).
- We add two dense layers at the end.
  - One hidden layer with 512 units.
  - One output layer.

cifar_vgg.py: Looking Into the Code

```
num_base_layers = len(vgg16.layers)
for i in range(0, num_base_layers):
    refined_model.layers[i].trainable = False
```

- Here we freeze the weights of the VGG-16 layers that we are using, so that they do not get modified when training on our “small” CIFAR10 subset.

cifar_vgg.py: Looking Into the Code

```
hist_tr100_den2_512 = refined_model.fit(small_training_inputs,
                                        small_training_labels,
                                        epochs=epochs,
                                        batch_size=batch_size,
                                        validation_data=(test_inputs, test_labels))
```

- We use some new tricks in our call to **fit()**, that does the training.
- We specify using the test data as “validation data”.
- This way, after every epoch, we see both training accuracy and test accuracy.
- This allows us to see if training starts overfitting.
  - Then training accuracy goes up, test accuracy goes down.

cifar_vgg.py: Looking Into the Code

```
hist_tr100_den2_512 = refined_model.fit(small_training_inputs,
                                        small_training_labels,
                                        epochs=epochs,
                                        batch_size=batch_size,
                                        validation_data=(test_inputs, test_labels))
```

- We also save the return value of **fit()** into a variable.
- This is a history variable, that contains accuracy and loss information for every epoch. This info is accessible as:
  - hist_tr100_den2_512.history["val_accuracy"] for validation accuracy.
  - hist_tr100_den2_512.history["val_loss"] for validation loss.
  - hist_tr100_den2_512.history["accuracy"] for training accuracy.
  - hist_tr100_den2_512.history["loss"] for training loss.

Options We Did Not Explore

- There are many other hyperparameters and design choices that could make a difference, for better or for worse.
- Optimization method:
  - We used “adam”, did not try “RMSprop” or other options.
  - Learning rate: we used the default value. Various references suggest that using a smaller learning rate in transfer learning may help.
- “Unfreezing” some of the pre-trained weights.
  - The “Deep Learning with Python” textbook, by François Chollet, describes an additional stage (after we have trained the new layers), where we unfreeze and re-train some of the last layers of the pre-trained model.
- Data augmentation. We artificially create a larger training set by adding variations of training examples.
  - Variations can include rotating, flipping, translation, scaling, adding noise…
  - Data augmentation can be used with or without transfer learning.

Transfer Learning: Recap

- The goal in transfer learning is to exploit the information contained in large datasets, to improve classification accuracy in small datasets.
  - The problem is that the classes in the small datasets do not appear in the large datasets.
- With deep neural networks, it is easy to write code that does transfer learning.
- As usual, we must make many design choices, including:
  - How many layers of the pre-trained model to discard.
  - How many layers, and what type, to add.
- Transfer learning may not work, if the target data has some fundamentally different characteristics than the source data.
  - Example: when the source is ImageNet and the target is MNIST.