Deep Learning

Deep Learning

Lecture: Convolutional Neural Networks – Fine Tuning and Ensembles

Ted Scully

Fine Tuning a CNN

- The second method of transfer learning is called <u>fine-tuning</u>. You will have noticed in the previous method when using networks as feature extractors we left the <u>weights of the original network unchanged</u>.

- Fine tuning utilizes a pre-trained CNN as a starting point but subsequently attempts to **<u>"fine tune" some of the weights in the convolutional layers and/or fully connected layers</u>** to perform well on the new classification task.

- This technique can often provide even better results than feature extraction.

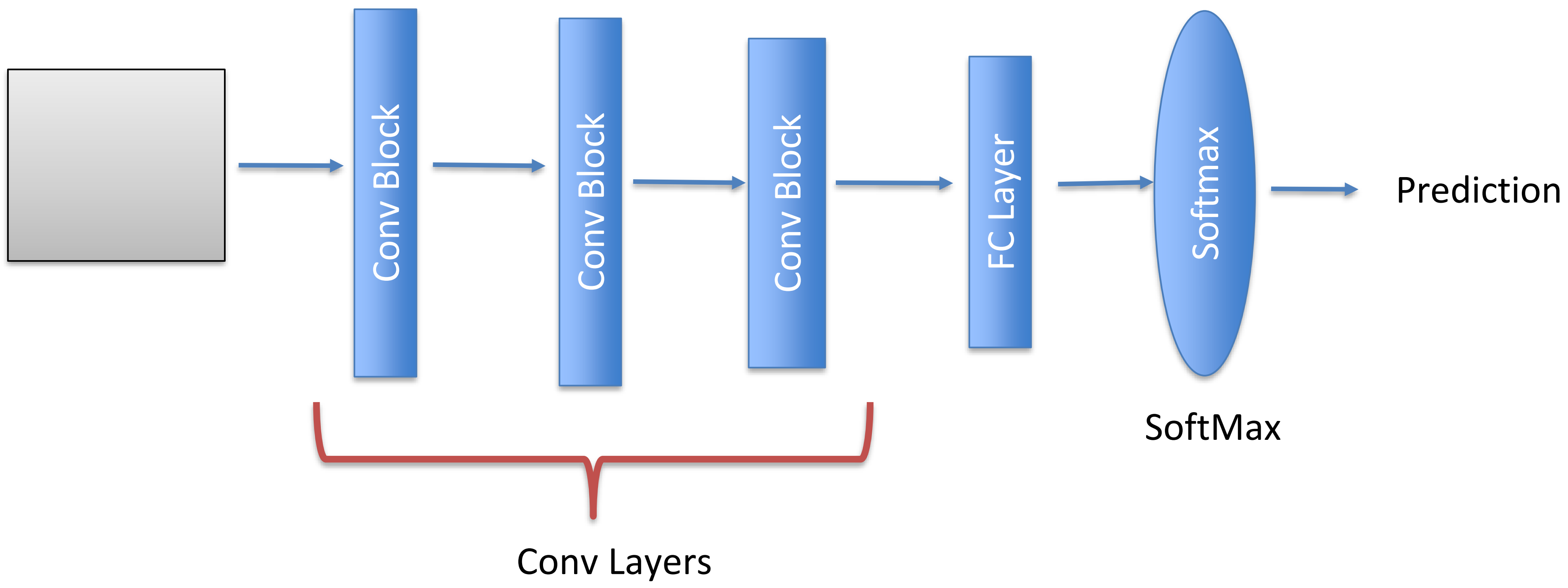
Fine Tuning for CNNs

▶ The following are the typical steps used to achieve fine tuning with a CNN

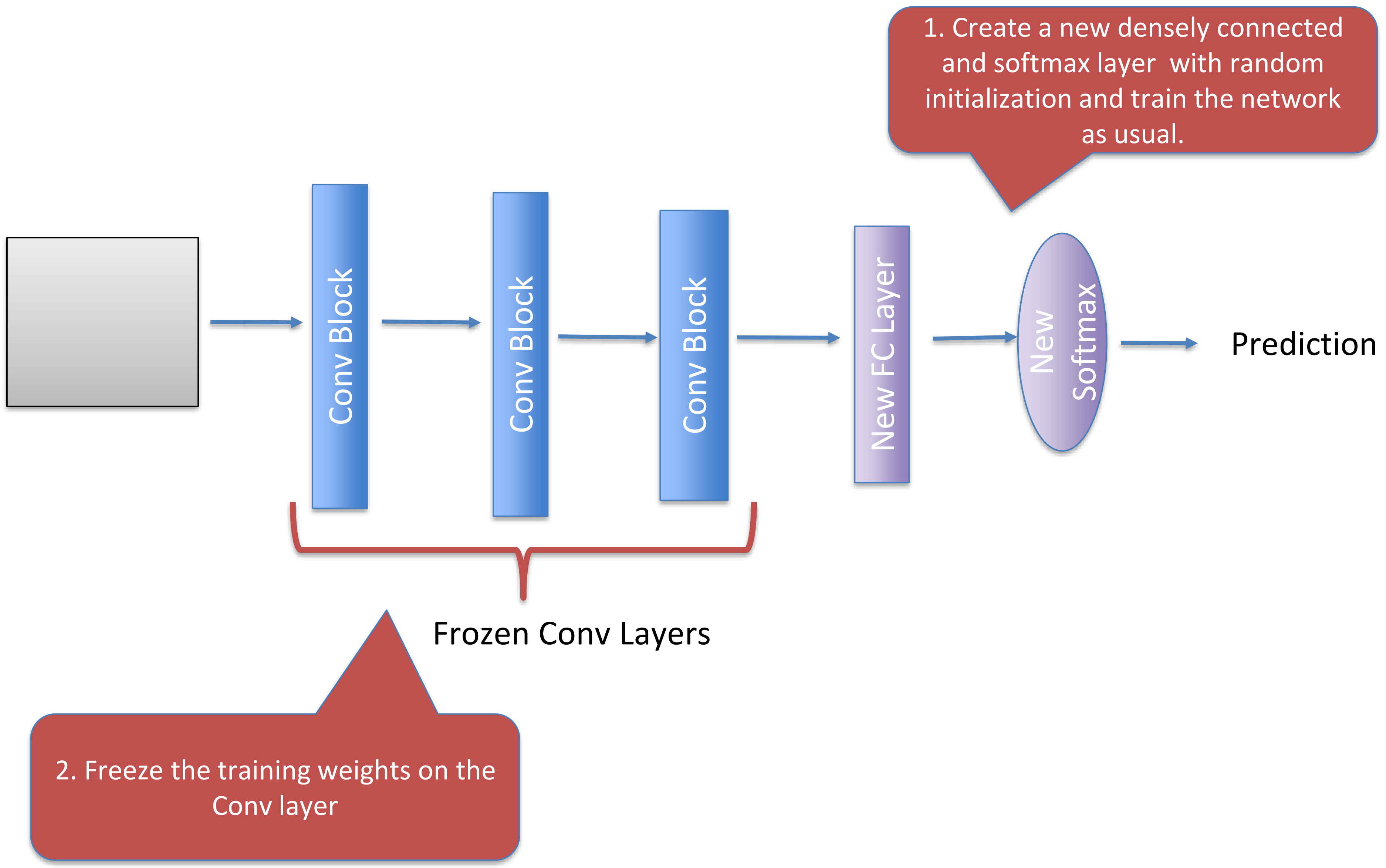
- **Add a new fully connected network** to the top of a pretrained network (in place of the existing FC layer). Weights randomly initialized.
- Next you will **freeze the trainable parameters** on all layers except the FC layers.
- **Train the weights** on the new FC layer.
- <u>**Unfreeze**</u> the trainable weights on some of the <u>**convolutional layers**</u> in the base network.
- Train the network again using a very small training rate.

Fine Tuning for CNNs

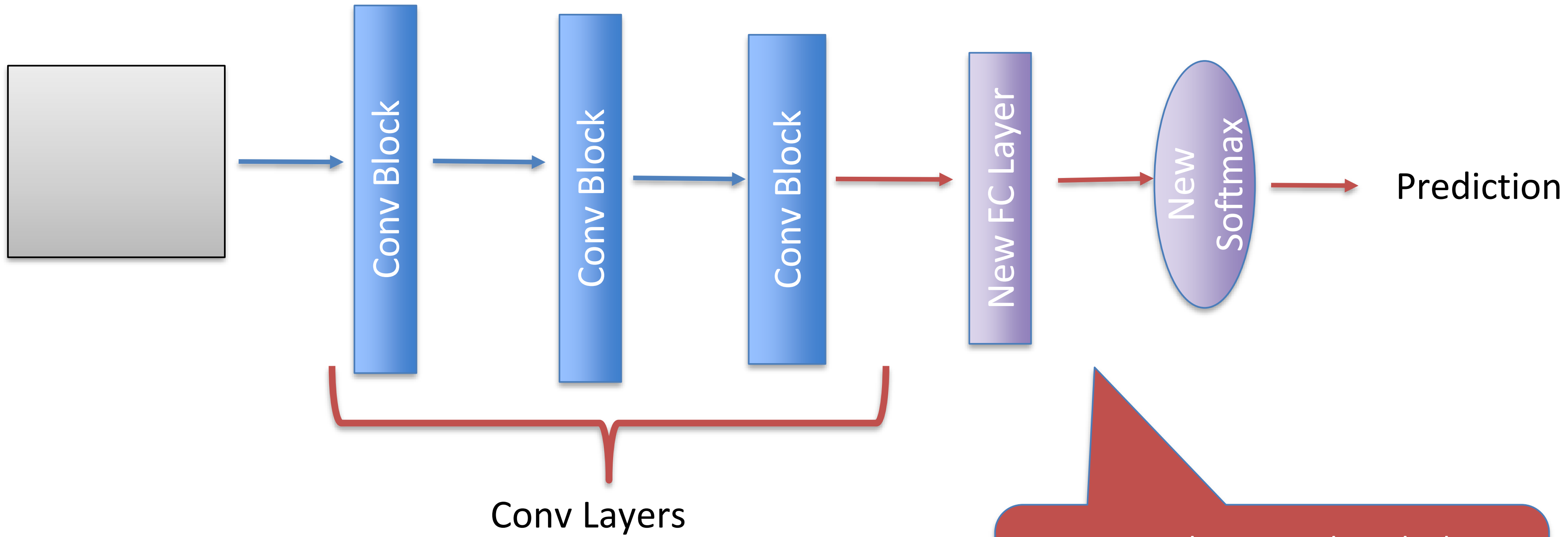
- For simplicity we will illustrate the process using this the following architecture.



Fine Tuning for CNNs – Phase A


1. Create a new densely connected and softmax layer with random initialization and train the network as usual.
Conv Block
Conv Block
Conv Block
New FC Layer
New Softmax
Prediction
Frozen Conv Layers
2. Freeze the training weights on the Conv layer

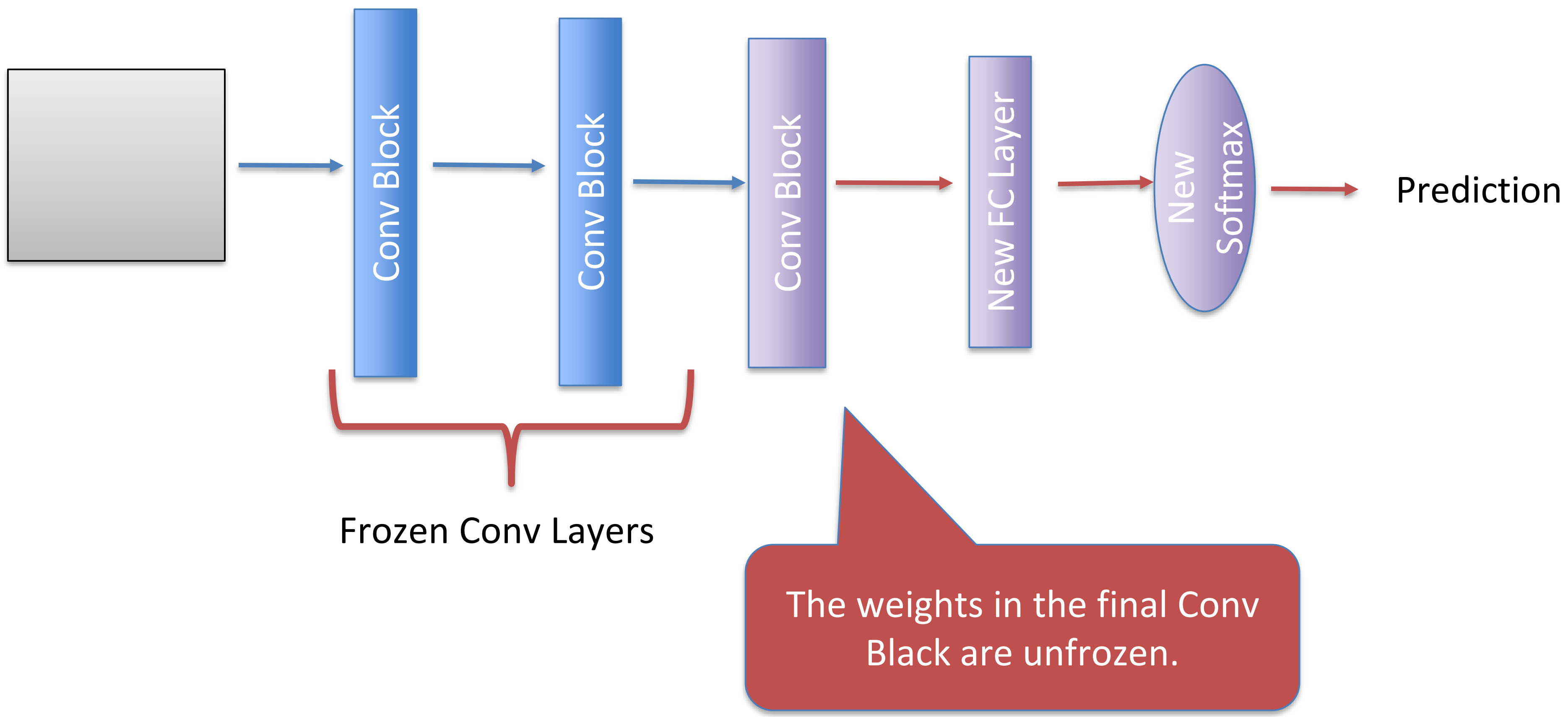
Fine Tuning for CNNs – Phase A



3. Train the network. Only the weights in the new FC and SoftMax layer get updated.

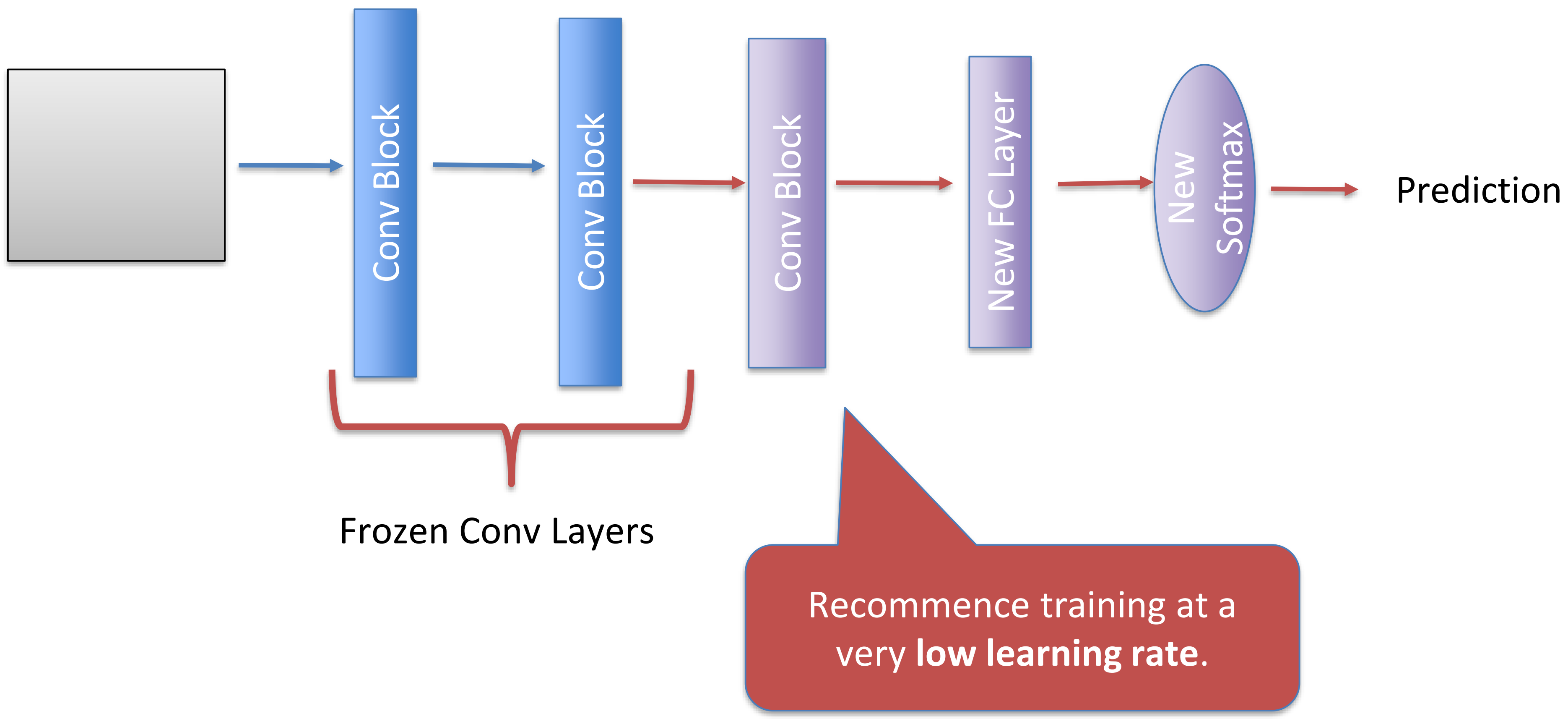
Fine Tuning for CNNs – Phase B

- One we have finished training we next unfreeze the weights on a selection of the earlier Conv layers. In this case we unfreeze the final Conv Block (note while we have only unfrozen the one layer here we could unfreeze multiple if needed).



Tuning for CNNs – Phase B

- We recommence training at a very low learning rate. This is important. Otherwise the loss value is high and the magnitude of change to the weights in the unfrozen convolutional layers can be very disruptive.



Fine Tuning for CNNs

- Over the next few slides we will look at an implementation of fine tuning with the Dogs and Cats dataset. There are two clear phases to the code.
    - **Phase A**: Replace the FC part VGG16 network and train new FC layers.
    - **Phase B**: In the second we use the resulting model, unfreeze some of the later convolutional layers and continue to train further.

- For any Keras model we can specify all weights in the network as **trainable** using **modelName.trainable = True**

- We can access all layers in a model using (returns a list all layer objects)

  vggModel.layers

- Similiarly we can enable or disable the trainable weights for a specific layer by using

  layer.trainable = True

- We can access the name of an individual layer by using

  layer.name

The initial code is exactly the same as we have seen before.

It reads in the images and converts them into NumPy arrays that are then normalized.

```
# This dictionary will contain all labels with an associated int value
labelDictionary ={}

# The procImages function will read all image data from a folder
# convert it to a Numpy array and add to a list
# It will also add the label for the image, the folder name
trainImages, trainLabels = procImages(trainDataDir,
              labelDictionary, width, height)
valImages, valLabels = procImages(validationDataDir,
              labelDictionary, width, height)

# Normalize data
trainImages = trainImages.astype("float") / 255.0
valImages = valImages.astype("float") / 255.0

# Map string label values to integer values.
trainLabels = (pd.Series(trainLabels).map(labelDictionary)).values
valLabels = (pd.Series(valLabels).map(labelDictionary)).values
```

Associated Colab Notebook

Notice we load the VGG network and immediately set the trainable weights to **False**. Therefore, any training performed will not alter these weights.

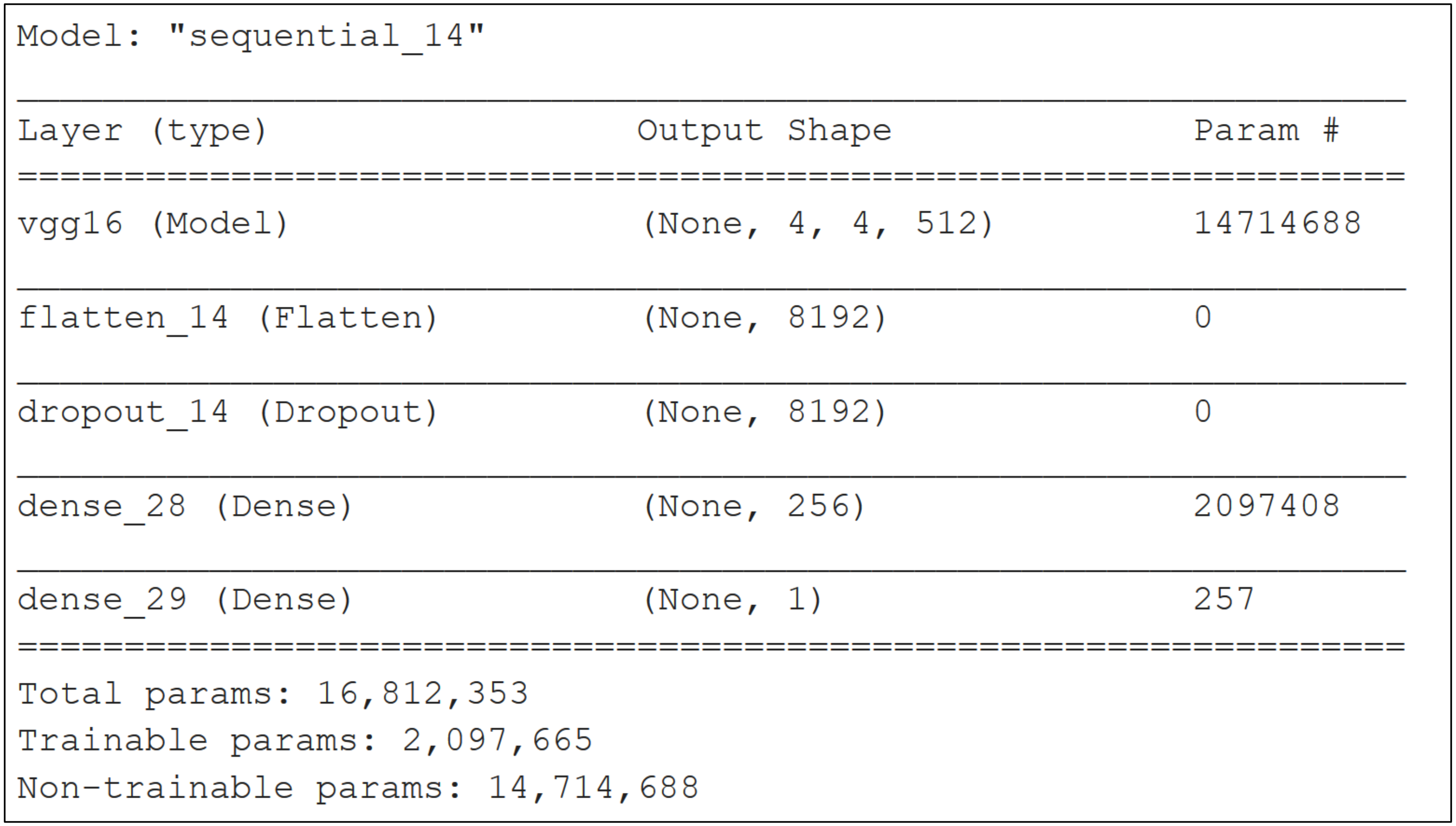
Next we create a new Keras Sequential model. A really useful feature in Keras is to that we can add an existing Keras model to a sequential model. Notice that we can add the vggModel directly to our new model.

We flatten the VGG output. Next we add a new Dense FC layer with 256 neurons with dropout.

We also add a Sigmoid activation function as this is a binary classification problem.

Associated Colab Notebook

```
vggModel = tf.keras.applications.VGG16(weights='imagenet',
            include_top=False, input_shape=(150, 150, 3))

vggModel.trainable = False
model = tf.keras.models.Sequential()
model.add(vggModel)
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
print (model.summary())
```



```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten_14 (Flatten)         (None, 8192)              0
_________________________________________________________________
dropout_14 (Dropout)         (None, 8192)              0
_________________________________________________________________
dense_28 (Dense)             (None, 256)               2097408
_________________________________________________________________
dense_29 (Dense)             (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 2,097,665
Non-trainable params: 14,714,688
```

The code here is again very similar to what we have seen before.

We compile and build our model in the usual way.

In this example I just run the training for 15 epochs for both phase A and B.

```
model.compile(loss='binary_crossentropy’,

optimizer=tf.keras.optimizers.RMSprop(lr=0.001),
          metrics=['accuracy’])

H =model.fit(trainImages, trainLabels,
        epochs=NUM_EPOCHS, batch_size=32,
                validation_data=(valImages, valLabels))
```

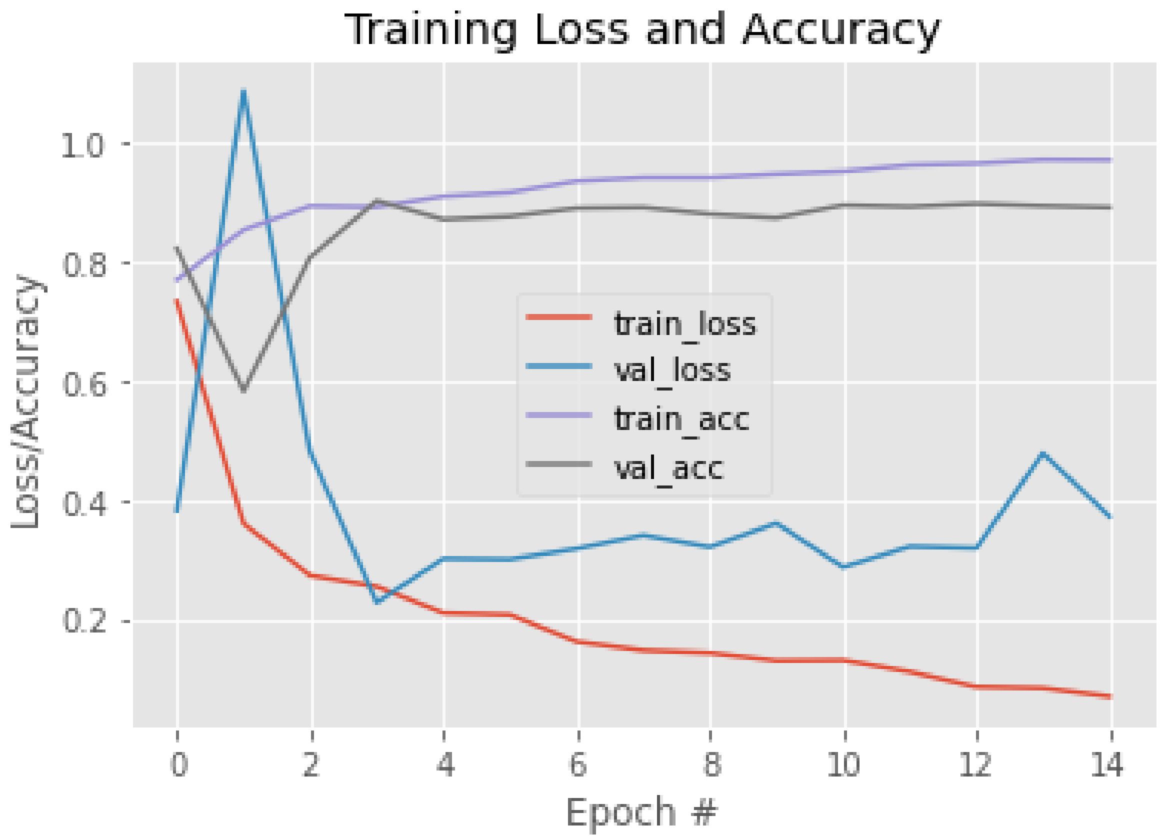
Associated Colab Notebook

Phase A Results

After phase A, when we just train the new FC layer we get an impressive accuracy value of **0.89**.

Once phase A is completed we then move on to the phase where we unfreeze some of the convolution layers in the network and continue to train at a very low training rate.

On the next slide we look at the architecture of the VGG model.



```
Layer (type)              Output Shape              Param #
=================================================================
input_5 (InputLayer)        (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)       (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)       (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)  (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)       (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)       (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)  (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)       (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)       (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)       (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)  (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)       (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)       (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)       (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)  (None, 9, 9, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)       (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)       (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)       (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)  (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
```

We are going to unfreeze all layers from block4_conv1 on.

We will then train the full network at a low learning rate.

The code at the top of this slide is very important. It allows to enable and disable the trainable weight on each layer.

We set a Boolean variable trainableFlag initially to False.

We can obtain a list of all layers using .layers. The for loop iterates through every layer in the model starting at block1_conv1.

It sets the trainable weights in each layer to the value of the flat variable. Therefore, all layers weights are set to False until we reach block4_con1. After this all subsequent layers weights are set to True.

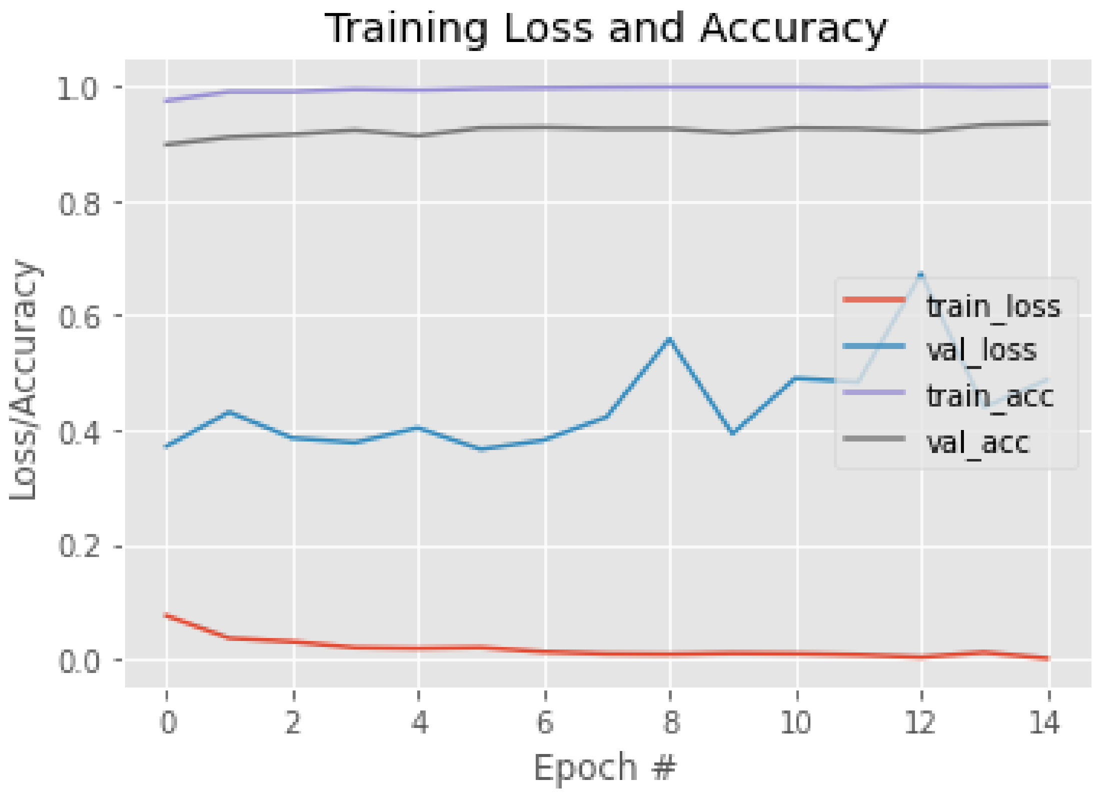
```
vggModel.trainable = True
trainableFlag = False

for layer in vggModel.layers:

    if layer.name == 'block4_conv1':
        trainableFlag = True
    layer.trainable = trainableFlag

model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(lr=1e-5),
              metrics=['accuracy’])

H =model.fit(trainImages, trainLabels,
              epochs=NUM_EPOCHS, batch_size=32,
              validation_data=(valImages, valLabels))
```

Associated Colab Notebook

After phase B, where we unfreeze block 4 and 5 convolutional layers we obtain an accuracy of **0.93** on the dataset.

It's common to use **small learning rates** for CNN weights (phase B) that are being fine-tuned. This is because we expect the weights of the pretrained CNN to already be good, so we don't want to distort them too quickly and by too much.

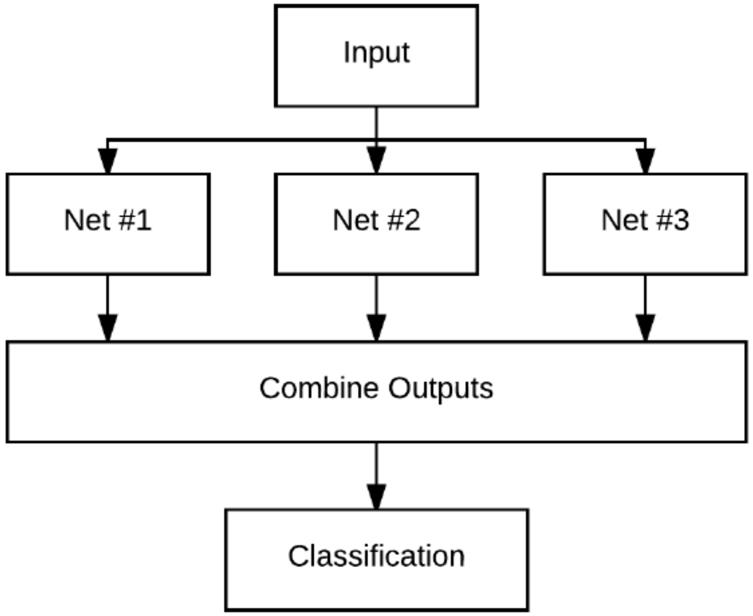
You will have noticed that after Phase A and B there is evidence of overfitting. **Data augmentation** can be used to help address this and achieve even higher levels of performance. In fact Data Augmentation used on this problem can help obtain 0.97-0.98 accuracy.



Ensembles

- We have already seen the concept of ensemble learning, where we generate multiple ML models (base models) and form a **meta-learner** that will aggregate the predictions from each of the **base models**.

- By aggregating or combining the predictions from multiple machine learning models together, we can often outperform (i.e., achieve higher accuracy) using just a single model.



Ensembles

- The approach to implementing an Ensemble is typically influenced by the following three factors.

- **Variability in underling training data**. That is you train multiple models on different subsets of the data (such as bagging). Subspace sampling as with a random forest.

- **Variability in the base learners**. You may build a model combined from different types of ML models (random forest combined with densely connected network, with gradient boosting). That is, vary the selection of models used in the ensemble.

- **Method of Aggregation**: The most basic method we use for combining the results of base model is by averaging. An alternative is to weigh the importance of different models in the aggregation process. An alternative method is to develop another model to learn the optimal weights to apply to each of the base learners (stacking).

Ensembles in Neural Networks

- We can also apply ensemble techniques effectively with Neural Networks.
- In fact, nearly all **state-of-the-art publications** that compete in the **ImageNet** challenges report their best findings over ensembles of Neural Networks (ResNet, Inception, AlexNet). AlexNet for example explored combinations of 2, 5 and 7 different models.
- Most of the winners of **Kaggle competitions** use some form an ensemble technique.
- It should be noted that while neural network ensembles are very useful for obtaining good results on benchmarks and when participating in competitions they come with **added training time and inference cost**.
- For example, rather than having to contend with the inference time for just a single model now you have to contend with multiple models. As such they are rarely used in real-time ML models.

Ensembles in Neural Networks

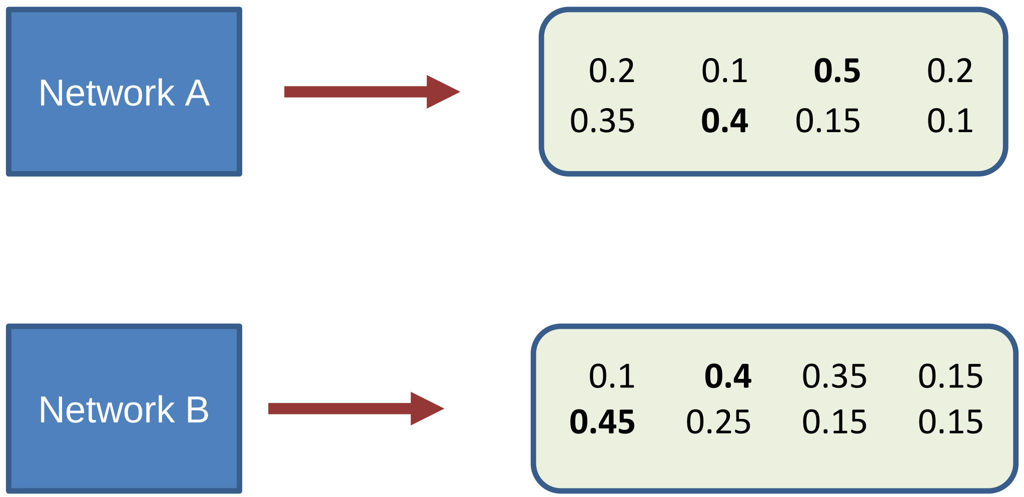
- It is important to understand that when we build a neural network ensemble for a multi-class classification problem that we don’t just perform **majority voting**.

- Instead we **leverage the probabilities** that are outputted from the **Softmax layer**.

- We **average all of the output probabilities** from the Softmax layer and make the classification decision based on these averaged probabilities.

Ensembles in Neural Networks

- Let’s assume we have a **4 class classification** problem and we have an ensemble with just **2 networks**.
- We push **2 test instances** through each network and collect the results.
- Each **row of the output matrix** corresponds to the probabilities for each test instance.
- Each **column in the matrix** corresponds to the class (column 1 is class 1, column 2 is class 2 etc)

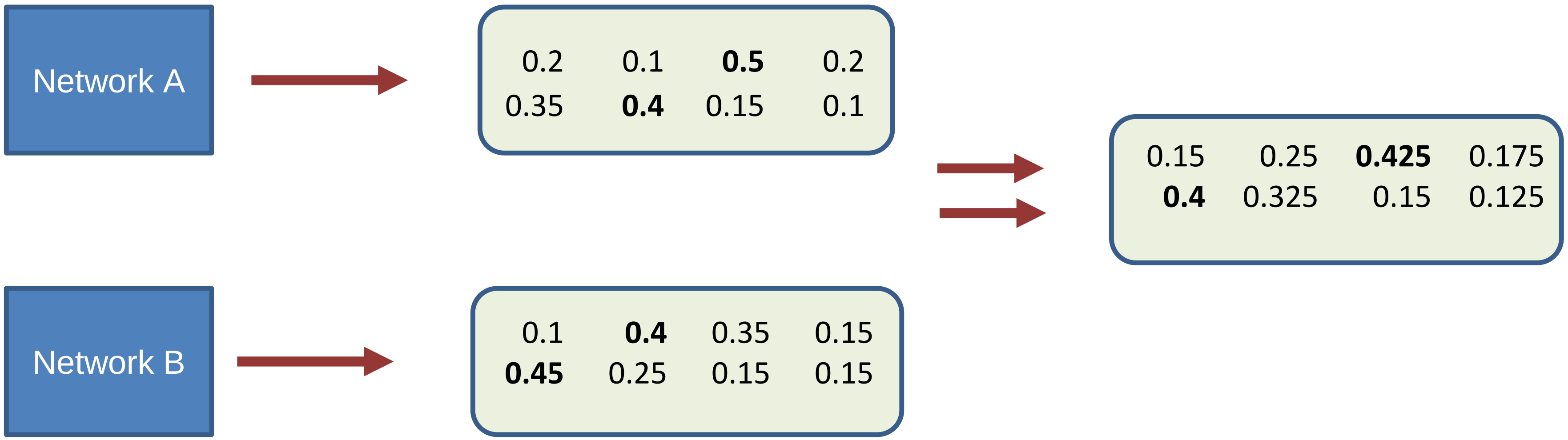


0.2	0.1	**0.5**	0.2
0.35	**0.4**	0.15	0.1

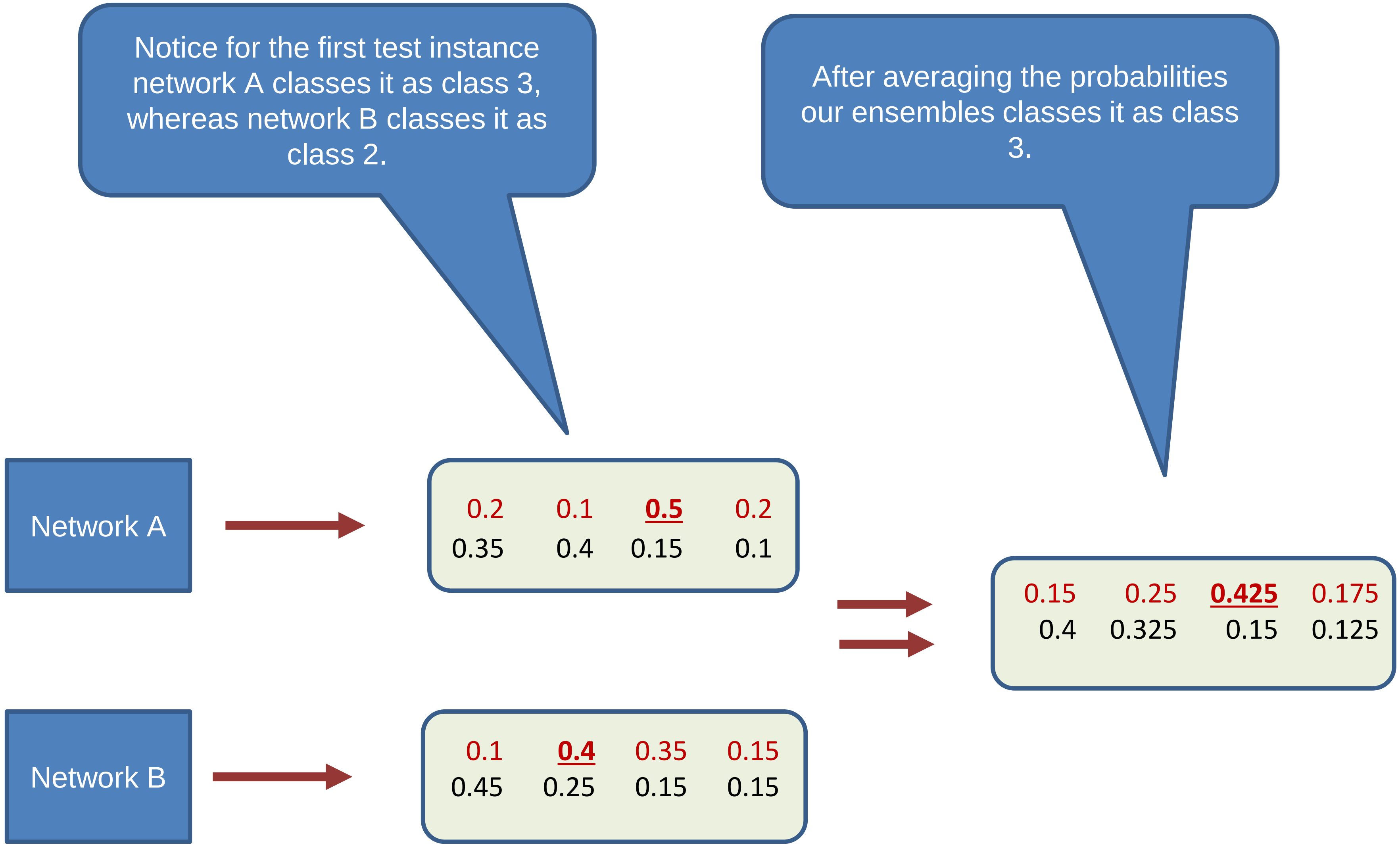
0.1	**0.4**	0.35	0.15
0.45	0.25	0.15	0.15

Ensembles in Neural Networks

- We then average the probabilities across the two networks and then identify the class with the highest probability for each test instance.



Ensembles in Neural Networks



Notice for the first test instance network A classes it as class 3, whereas network B classes it as class 2.

After averaging the probabilities our ensembles classes it as class 3.

Network A:

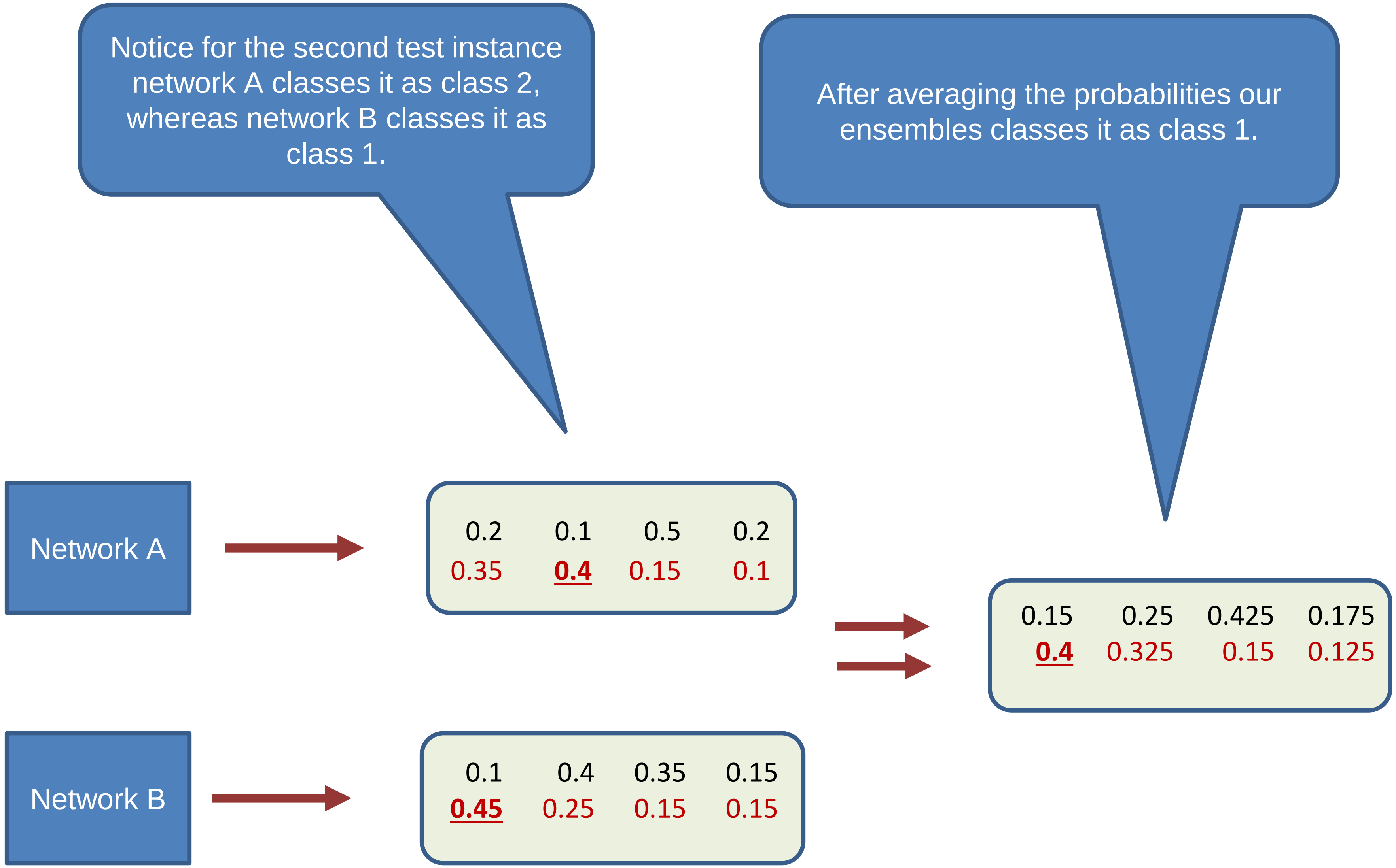
0.2	0.1	**0.5**	0.2
0.35	0.4	0.15	0.1

Network B:

0.1	**0.4**	0.35	0.15
0.45	0.25	0.15	0.15

0.15	0.25	**0.425**	0.175
0.4	0.325	0.15	0.125

Ensembles in Neural Networks


Notice for the second test instance network A classes it as class 2, whereas network B classes it as class 1.
After averaging the probabilities our ensembles classes it as class 1.
Network A
0.2 0.1 0.5 0.2
0.35 0.4 0.15 0.1
Network B
0.1 0.4 0.35 0.15
0.45 0.25 0.15 0.15
0.15 0.25 0.425 0.175
0.4 0.325 0.15 0.125

Build NN Ensembles

- Typically the number **of base models in a Neural Network ensemble is smaller** than you would typically encounter, which is primarily due to the computational cost of building these models.

- There are many different ensemble variants in deep learning:

- A very basic approach to building an ensemble with neural networks is to train a network with a **fixed structure** and configuration **multiple times**. Each of the models are initialized with different initial weight and are trained on the same data. Final predictions are arrived at through averaging. However, the amount of variation is somewhat limited with the approach.

- A simple variant of the above would be to train networks that have a different structure with different random initializations.

- Another variant is to **combine the outputs of multiple pre-trained networks.**

- Another is called referred to as **snapshot ensemble**s where multiple base models are extracted during each training process.