Chapter 3 - Learning Techniques

"The more that you read, the more things you will know.
The more that you learn, the more places you'll go."
— Dr. Seuss

Model quality is an important benchmark to evaluate the performance of a deep learning model. A language translation application that uses a low quality model would struggle with consumer adoption because it wouldn't serve its intended purpose of helping them communicate effectively with others who speak different languages. An application that employs a high quality model with a reasonable translation accuracy would garner better consumer support. In this chapter, our focus will be on the techniques that enable us to achieve our quality goals. High quality models have an additional benefit in footprint constrained environments like mobile and edge devices where they provide the flexibility to trade off *some* quality for smaller footprints.

In the first chapter, we briefly introduced learning techniques such as regularization, dropout, data augmentation, and distillation to improve quality. These techniques can boost metrics like accuracy, precision, recall, etc. which often are our primary quality concerns. We have chosen two of them, namely data augmentation and distillation, to discuss in this chapter. This is because, firstly, regularization and dropout are fairly straight-forward to enable in any modern deep learning framework. Secondly, data augmentation and distillation can bring significant efficiency gains during the training phase, which is the focus of this chapter.

We start this chapter with an introduction to sample efficiency and label efficiency, the two criteria that we have picked to benchmark learning techniques. It is followed by a short discussion on exchanging model quality and model footprint. An in-depth discussion of data augmentation and distillation follows right after. Following the lead from the previous chapters, the theory is complemented with programming projects to assist readers to implement these techniques from scratch. Our journey of learning techniques also continues in the later chapters.

Learning Techniques and Efficiency

Data Augmentation and Distillation are widely different learning techniques. While data augmentation is concerned with samples and labels, distillation transfers knowledge from a large model or ensemble of models to smaller models. The obvious question at this point is: why are we talking about them in the same breadth as efficiency? To answer this question, let's break down the two prominent ways to benchmark the model in the training phase namely sample efficiency and label efficiency.

Sample Efficiency

Sample Efficiency is concerned with the total number of training samples including repeats seen by the model to reach the desired performance threshold (in terms of accuracy, precision, recall or other performance metrics). We designate a new model training setup to be more *sample efficient*, if it achieves similar or better performance with fewer data samples when compared to the baseline.

Think of it as teaching a child to recognize common household objects such as a toy, a cup or a saucer. The number of times you have to point and call out the kind of the objects such that the child identifies them correctly with desired accuracy is the total number of samples. If you have a training process that enables the child to reach the same accuracy by seeing a smaller number of samples, that process would be *sample efficient*.

Similarly, a sample efficient model training process requires fewer samples to achieve the same performance, which makes it cheaper to train. Refer to Figure 3-1 for an example of such a model, and note how it achieves accuracy similar to the baseline, but does so in fewer epochs. We could ideally save an epoch's worth of training time by terminating the training early, if we adopt this hypothetical sample efficient model training.

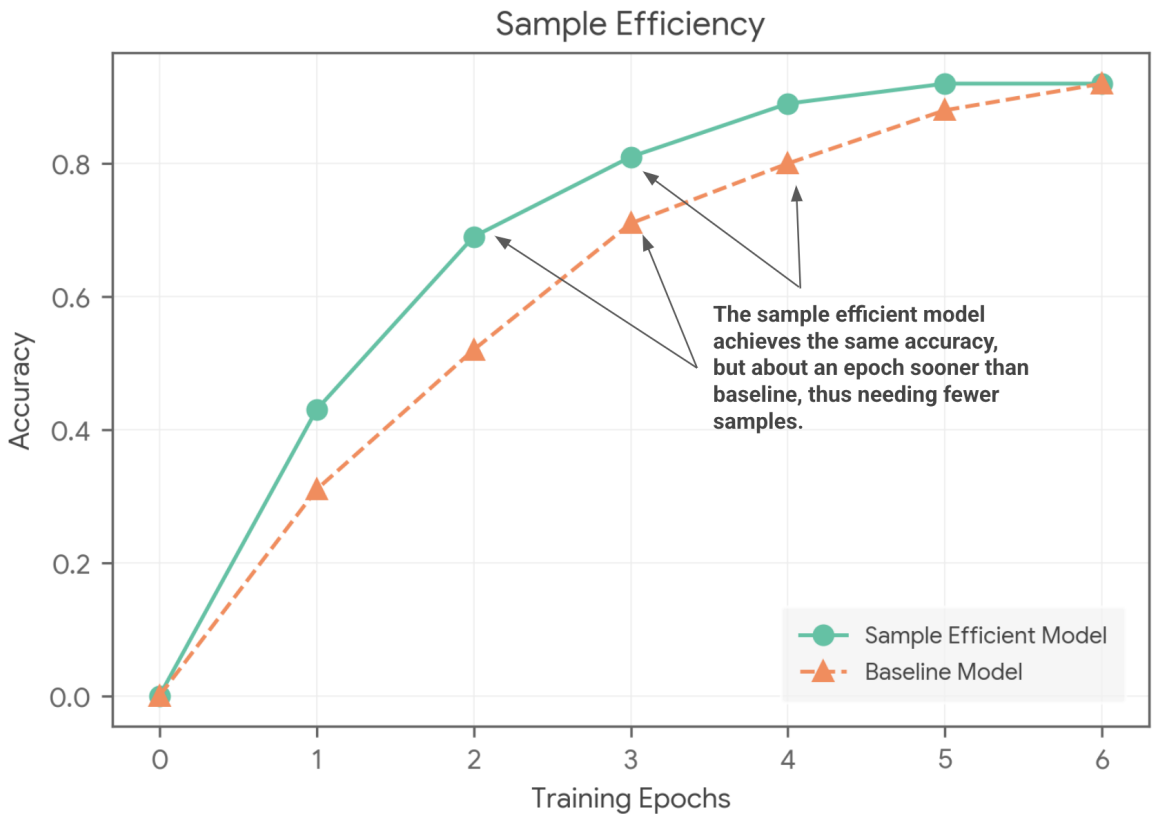


Figure 3-1: The above plot demonstrates sample efficiency between two model training setups. The sample efficient model achieves about the same accuracy, but reaches that point in fewer epochs, hence needing fewer samples.

Distillation is a learning technique which has been shown to reduce the number of samples that a model needs to see to converge to the desired accuracy. We will cover it in detail later on in this chapter. But first, let's get ourselves familiar with label efficiency.

Label Efficiency

The number of labeled examples required for a model to reach the desired performance benchmark is another important metric to evaluate the effective utilization of the training data. Labeling data is often an expensive process both in terms of time consumption and fiscal expenditure because it involves human labelers looking at each example and assigning them a label that they believe describes it best. The assigned labels are subjective to the perception of their labelers. For example, a human labeler might perceive the digit in figure 3-2 as a *1* and another one might see it as a *7*.

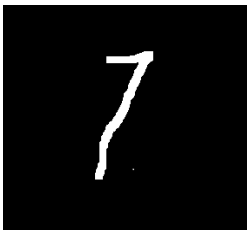


Figure 3-2: An example of handwritten digit that can potentially confuse the human labelers to choose a 1 or a 7 as the target label. Obtaining labels in many cases requires significant human involvement, and for that reason can be expensive and slow.

Many organizations and AI labs can only do limited data labeling because it is expensive. Moreover, the labelers need to be trained for the task such that they follow the guidelines correctly while labeling, which further adds to the costs. In many cases, to reduce the chances of mislabeling due to human error, data is labeled by multiple human labelers and the label that wins the consensus is assigned to the example. Given all the costs involved, it is imperative to utilize all the training data available to us, as efficiently as possible.

Extending the teaching-a-child analogy, consider the number of distinct examples of objects (labels) you must show a child before they can learn to identify them with high accuracy. All cups have the same basic shape. One possible way to teach a child is to look at the same cup from different angles and rotations, in varying degrees of light. The same process can be repeated for other objects. If the child learns to recognize these objects accurately with fewer numbers of distinct objects being shown, we have made this process more *label efficient*.

Similarly, if you could make the model training process *label efficient*, you would incur a lower cost to meet a performance benchmark. Refer to Figure 3-3 for an example of a label efficient model's training curve.

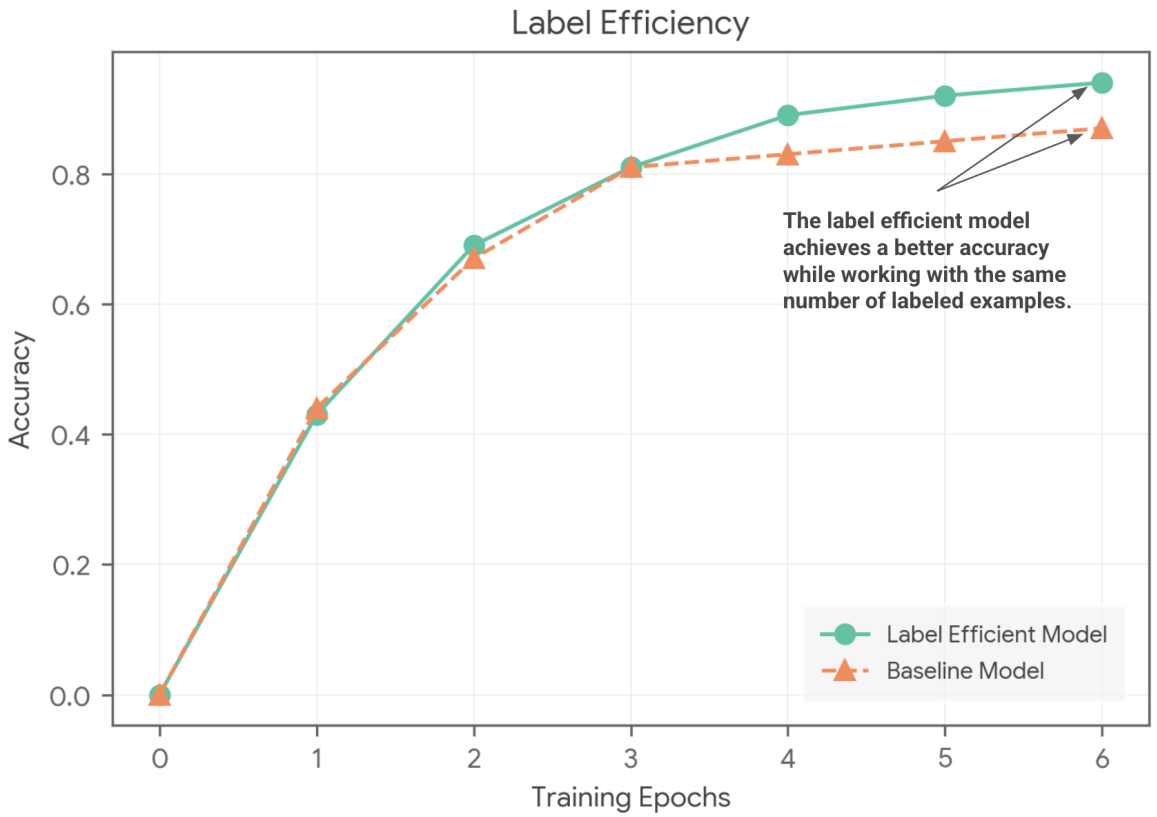


Figure 3-3: The above plot demonstrates label efficiency between two model training setups. The label efficient model achieves a higher accuracy with the same number of labeled training examples.

Data Augmentation is a set of techniques which leverage the original training data to generate more training examples without having to label them. We'll familiarize ourselves with these techniques later in this chapter.

Table 3-1 presents a concise summary of both, sample and label efficiency.

Sample Efficiency	Label Efficiency
• Optimize the number of training samples seen by the model to reach the desired performance threshold. • A sample efficient model needs fewer training epochs to match the baseline performance, **hence saving on training costs**.	• Optimize the number of distinct labeled samples needed to reach the desired performance threshold. • A label efficient model needs fewer labels to match the baseline performance, **hence saving on labeling costs**.

Table 3-1: A quick summary of sample and label efficiencies.

Both sample and label efficiency techniques help us reduce training costs. Assuming we do have a sample efficient and/or label efficient training setup, can we exchange some of this to achieve a model with a better footprint? The next subsection elaborates it further.

Using learning techniques to build smaller and faster efficient models

Overall, as summarized in table 3-1, improving sample efficiency enables faster model training, and label efficiency is useful to reduce the required number of labeled examples. This is great, but what if we are okay with our existing labeling and training costs, but would rather prefer to achieve a smaller and faster model?

Turns out, using learning techniques to improve sample and label efficiency, often helps to make resource efficient models feasible. By feasible, we mean that the model meets the bar for quality metrics such as precision, recall, accuracy or others'. Let's understand it with an example.

Assume that we are working on a model for a home-automation device. Figure 3-4 shows the high level workflow of such a device. The model continuously classifies audio signals into one of the *four* classes, *three* of which are the keywords that the device will accept: *hello*, *weather* and *time*. The fourth class (*none*) indicates the absence of an acceptable keyword in the input signal.

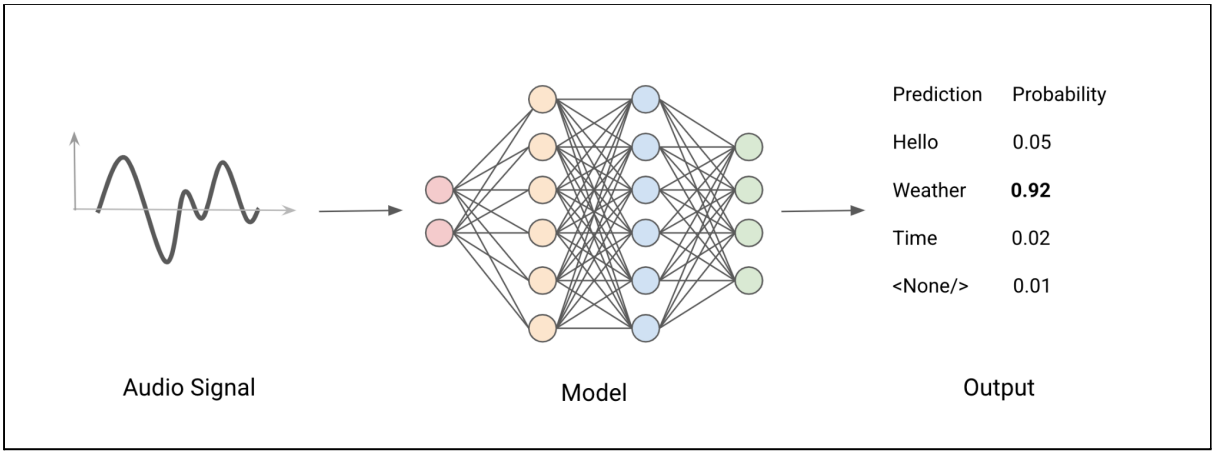


Figure 3-4: Workflow of a home-automation device which detects three spoken words: hello weather and time. The output is none when none of the three acceptable words are detected.

Now, let's say that the performance threshold for a given model to be considered feasible for deployment is a classification accuracy of 80%. Any additional improvement is not required, and we would prefer to choose the smallest model with best performance that meets this threshold.

Assume that we are given a 300 KB model that is trained with 10,000 labels for over 100,000 training steps and achieves 80% accuracy. Further assume that with the learning techniques that improve label and sample efficiency, the model now needs only 5,000 labels and 40,000

steps to reach that accuracy (reducing 5,000 labels and 10,000 steps when compared to the baseline).

Now, there can be a few different options available to us, based on what we want:

1. **We only care about reaching the accuracy goal of 80%**: In this case, it is perfectly fine to take the lower labeling and training costs and call it a day!

2. **We want the highest possible accuracy with the original training costs**: We can let the model train with the new learning techniques. In many cases, this will improve performance. Let's say that the 300 KB model stacked with our learning techniques achieves an 87% accuracy. Again, this is a straightforward scenario.

3. **We want to reduce model footprint, while keeping accuracy and training costs same**: We know that learning techniques give a boost to the model performance. They are also likely to boost the performance of smaller models (fewer parameters / layers, etc.). Concretely, we want to find the smallest model, which when trained with the learning techniques, meets the quality threshold (80% accuracy) with the original training budget.

   In our example, suppose that a trimmed model is 150KB in size and achieves an 80% accuracy with the same number of training steps and labels. Thus, delivering a *2x model compression*. Again, this is a hypothetical scenario which illustrates how learning techniques are leveraged to reduce the model footprint.

Table 3-2 shows a comparison of vanilla models (without the learning techniques) with the models that employ learning techniques in terms of model size and accuracy. The acceptable combinations of sizes and accuracies are marked in bold.

	100 KB	150 KB	200 KB	250 KB	300 KB
Without Learning Techniques	64%	71%	76%	79%	**80%**
With Learning Techniques	72%	**80%**	**84%**	**86%**	**87%**

Table 3-2: Accuracies of differently sized models with and without learning techniques. The acceptable combinations of sizes and accuracies are marked in bold.

In our example, only the 300 KB vanilla model is acceptable for deployment (it meets 80% accuracy target). Whereas, among the models with the learning techniques, four models with the smallest being the 150 KB model are valid deployment candidates.

Learning techniques like data augmentation and distillation, might not be viewed as techniques that help you improve your model footprint. However, given that they lead to an improvement in quality metrics, we can use them to boost the performance of models that might have not been suitable earlier because of a lower accuracy, precision / recall, etc. Effectively, we are exchanging the improved quality for a better footprint. Table 3-2 illustrates this concept.

So far, we have introduced the learning techniques as ideas to improve quality metrics and exchange those improvements to reduce footprint metrics. This was necessary to build an intuition of the real world problems they aim to tackle. Now, let's dive into these learning techniques to understand what they are and how to employ them in deep learning workflows. We start with data augmentation in the next section.

Data Augmentation

Data Augmentation is a set of dataset manipulation techniques to improve sample and label efficiencies of deep learning models. Over the years, a wide range of techniques have been developed to facilitate input data generation and transformation.

These techniques help to overcome dataset shortcomings like: small size, skewed samples, or partial coverage. It is fair to ask: why don't we just get more data? Consider the following examples. MNIST dataset contains 70,000 handwriting samples sourced from the American Census Bureau employees and the American high school students. The creators went through a tedious sample collection and digitization process. It would cost substantial labor, time and money to collect more samples. In 2019, Kaggle[1] opened a competition to design a model to identify humpback whales from the pictures of their flukes[2]. The primary challenge with that dataset is the limited number of sample pictures for each whale. The dataset contains over 5000 individuals with more than 2000 having just a single sample, over 1200 individuals with two samples, and more than 500 individuals with three samples. As opposed to the previous examples, whale data collection is trickier. The data acquisition difficulties inspired researchers to invest in developing techniques that workaround this scarcity of data.

We have organized the data augmentation techniques in the following categories: label invariant transformations, label mixing transformations and synthetic sample generation. The label invariant techniques transform the input samples. The transformed samples are labeled identical to the original sample. A slightly tilted cat is still a cat! The label mixing transformations generate samples based on differently labeled inputs. The target label is a composite of the inputs that were combined. A combination of a dog with a hamster image (figure 3-5) is assigned a composite [dog, hamster] label!

---

[1] Kaggle is a platform to learn and compete on AI problems.
[2] A whale's tail fins are called flukes.

Figure 3-5: A mixed composite of a dog (30%) and a hamster (70%). The label assigned to this image is a composite of the two classes in the same proportion. Thus, the model would be expected to predict a 'dog' with a probability of 30% and a 'hamster' with a probability of 70%.

The sample generation techniques use models to generate samples for labels. Consider a training sample for English to Spanish translation: [English: "I am doing really well", Spanish: "Estoy muy bien"]. Let's say we have another model which translates Spanish to English. This model translates "Estoy muy bien" to "I am fine". This result can be used to train our original English to Spanish translation model.

Let's dig deeper into each of these categories using examples and code samples.

Label Invariant Transformations

Label invariant transformations transform samples such that the resulting samples preserve the original sample labels. They operate on a per sample basis which makes it straightforward to apply them on any dataset. A single transformation on every sample results in a dataset 2x the original size. Two transformations applied separately result in a dataset 3x the original size. Can we apply *N* transformations to create a dataset *N*x the size? What are the constraining factors? An image transformation recomputes the pixel values. The rotation of an RGB image of 100x100 requires at least 100x100x3 (3 channels) computations. Two transformations would require 2x100x100x3 computations. When the transformations are applied during the training process, it invariably increases the model training time.

A transformation also changes the dataset distribution. It should be chosen to address the dataset deficiencies with the expectation that the transformed distribution improves the model quality and performance. For instance, a dataset of cat images would likely have the cats positioned at various angles. It would make sense to have rotational transformation for such a dataset. A human face dataset would be less likely to have, for example, inverted faces. It wouldn't be much use spending time and resources to train a model that recognizes faces in any orientation if it is going to be used to scan people entering a building. I have yet to come across inverted people trying to gain access!

Popular deep learning frameworks provide quick ways to integrate these transformations during the training process. Tensorflow comes bundled with the ImageDataGenerator which can transform images during the training process, thus eliminating the need to generate samples beforehand. Next, we will discuss a few label invariant image and text transformation techniques.

Image Transformations

This discussion is organized into the following two categories: spatial transformation and value transformation. *Spatial transformation* transforms the positions of image pixels. For example, a vertical shift moves the image pixels along the y-axis by a specified amount. A 50 pixel point shift moves a pixel with initial coordinates (x, y) to the final coordinates $(x, y + 50)$. As a result, the image is vertically shifted by 50px as shown in the top middle image in figure 3-6. Such a shift has two side-effects. First, a part of the image "falls off" the top edge. That information will be lost. And the second, the lower part of the image doesn't have any pixel data because it has been shifted up. These pixels need to be "filled" up. The deep learning frameworks provide several fill-up or interpolation algorithms to address the holes.

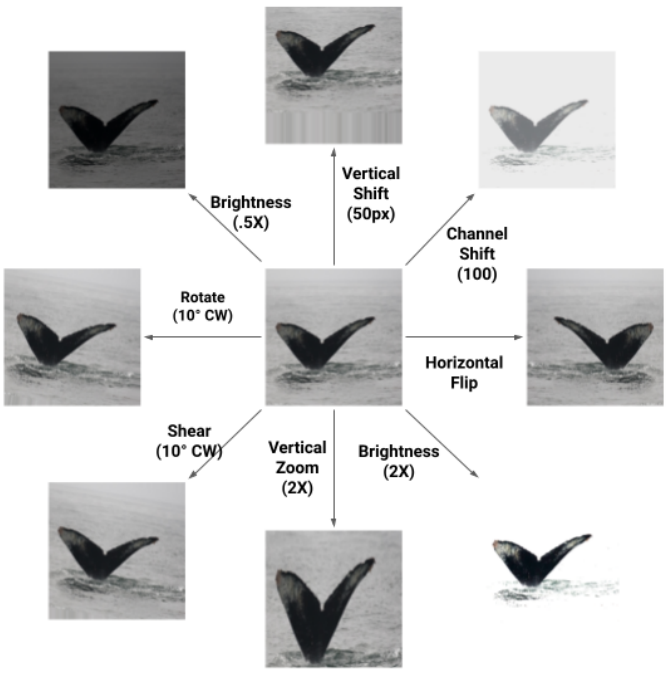


Figure 3-6: Image Transformations. The source image (center) is taken from Google Open Images Dataset V6. It is authored by Mike Baird and is licensed under CC BY 2.0. The image is resized to 224x224px prior to the transformations.

Value transformation operates on the pixel values. Let's take brightness transformation as an example. Figure 3-6 shows an image 2x bright (bottom-right corner) as compared to the original image (center). This transformation causes the whale fin to visually "stand out". This

transformation has a clipping side effect. An RGB image has a channel value range $[0, 255]$. Any channel values that exceed 255 after the 2x brightness transformation are clipped to 255 The channel values $c_v > 127$, after the 2x transformation, become $c_v^{'} > 255$. These values are clipped to 255.

We will discuss some examples of image transformations below. The code samples are provided to bridge the theory and practice gap. We have prepared a few helper functions: *load_image()*, *show_image()*, *transform()* and *transform_and_show()*, which will be used to transform the images. They facilitate various tasks such as loading an image from a url, applying various transformations to it and displaying the results.

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from keras.preprocessing.image import ImageDataGenerator
from urllib.request import urlopen

IMG_SIZE = 224

def load_image(url):
  with urlopen(url) as request:
      img_array = np.asarray(bytearray(request.read()), dtype=np.uint8)

  img = cv2.imdecode(img_array, cv2.IMREAD_COLOR)
  img = cv2.resize(img, (IMG_SIZE, IMG_SIZE), cv2.INTER_AREA)
  return cv2.cvtColor(img, cv2.COLOR_BGR2RGB).astype(int)

def show_image(image):
  # Display the image
  plt.axis('off')
  plt.imshow(image)

def transform(image, transform_opts):
  # An ImageDataGenerator instance to be used later to transform the input image.
  datagen = ImageDataGenerator()

  # Apply the transformations to the image.
  return datagen.apply_transform(image, transform_opts)

def transform_and_show(image_path, **transform_opts):
  # Load the image data
  # The data is formatted as (H, W, C)
  image = load_image(image_path)

  # Transformed Image
  transformed_image = transform(image, transform_opts)

  # Show the transformed image
  show_image(transformed_image.astype(int))
```

```
image_path = 'file:///whalefin.png'
```

Now, let's go through the various image transformations with code examples.

Rotation rotates the image pixels around the center. It is parameterized by θ. A positive θ value applies clockwise rotation whereas a negative θ rotates the image anticlockwise. The following code rotates an image 10° clockwise (leftmost image in the middle row in figure 3-6).

```
# Rotate 10 degrees in a clockwise direction
transform_and_show(image_path, theta=10)
```

Flip operation flips an image along the horizontal or the vertical axis. The rightmost image in the middle row in figure 3-6 is a horizontally flipped version of the central image.

```
# Horizontal Flip
transform_and_show(image_path, flip_horizontal=True)
```

Shift transformation slides an image in a horizontal or a vertical direction along their respective axis. A shift parameter, *s*, controls the slide amount in pixels. The middle image in the top row in figure 3-6 is a 50px upshifted image generated by below code.

```
# Horizontal Shift
transform_and_show(image_path, ty=50)
```

Zoom scales an area in an image. One of the techniques to identify whales uses the markings on their pectoral fins. A whale identification model is likely to perform better if the input image is transformed to focus on the markings. The bottom-central image in our example figure 3-6 is a 2x vertical zoom results in more pronounced fin markings.

```
# Vertical Zoom Transformation
transform_and_show(image_path, zx=.5) # A value of .5 implies 2X zoom
```

Shear transformation changes one coordinate while keeping the other fixed. In a sense, it is similar to a vertical or a horizontal shift. However, unlike the shifts where the moving coordinates are displaced perpendicular to the fixed axis, shear allows inclined movements. Shear has a θ parameter that specifies the displacement angle. The bottom-left image in figure 3-6 shows a 10° clockwise shear transformation.

```
# Clockwise Shear Transformation
transform_and_show(image_path, shear=10)
```

Brightness transformation is an example of value transformation. A 2X transformation doubles the pixel intensity values leading to a brighter image. A 2X brightness transformation (bottom-right corner in figure 3-6) of our whale image flattens the water texture causing the pectoral fin to stand out.

```
# 2X Brightness Transformation
transform_and_show(image_path, brightness=2)
```

Channel Intensity Shift shifts the RGB channel values uniformly across all channels. $p_c^{'} = p_c + s$ where c represents a channel and *s* is the shift amount. As opposed to the brightness transformation which scales the intensity values, channel shift adds or subtracts the shift amount. An add shift causes the minimum intensity value equal to or greater than *s*. A subtract shift clips the maximum intensity value equal to or lower than *s*. Top-right image in figure 3-6 shows a channel shift of 100 on our whale fin image.

```
# Shift towards high tones
transform_and_show(image_path, channel_shift_intensity=100)
```

In this section, we discussed a number of spatial and value transformation techniques for image data. We used an image of the whale to demonstrate the effects of transformations visually. The above list is not exhaustive, rather we have used it as a guide to help make better transformation choices. A few other commonly used techniques are contrast augmentation, color correction, hue augmentation, saturation, cutout, etc. Figure 3-7 shows a breakdown of the contributions of various transformations on the validation accuracy of a model trained on the CIFAR-10 dataset.

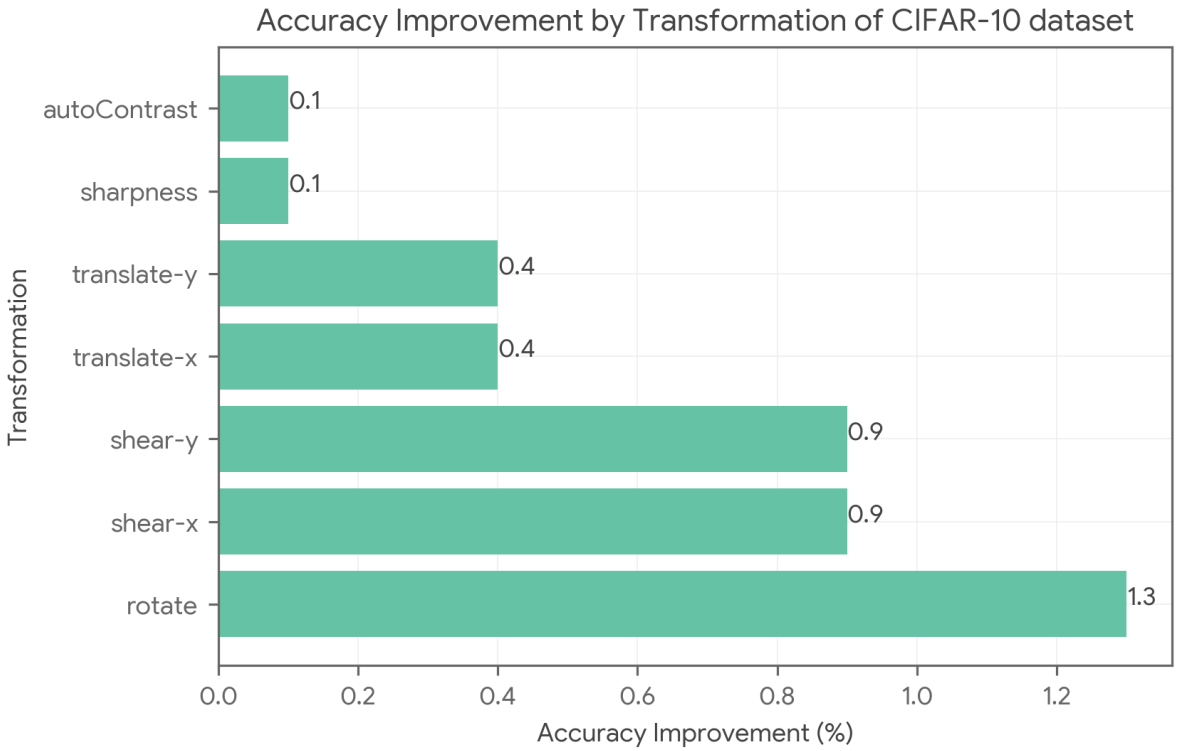


Figure 3-7: Validation Accuracy Improvements on the CIFAR-10 dataset for various transformations[3].

[3] Menghani, Gaurav. "Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better." *arXiv preprint arXiv:2106.08962* (2021).

It's time for a hands-on project to apply our recent learnings and measure their impact. We will use the *oxford_flowers102* dataset from tensorflow. It is a collection of 102 commonly occurring flowers in the UK (hence, the name). Instead of training a model from scratch, we will use a pre-trained ResNet50 model (trained on the imagenet dataset). Next, we will finetune (retrain) it with the flower dataset. Then, we will compare the performances with and without data augmentation to measure the benefits of the techniques we just learnt.

Project: Oxford Flowers Classification

The oxford_flowers102 dataset contains 1020 labeled examples each in the training and the validation sets. It is a small sample to train a good quality model. So, we use a pre-trained ResNet50 model and fine tune it. The code for this project is available as a Jupyter notebook **here**.

Tensorflow provides easy access to this dataset through the tensorflow-datasets package. Let's start by loading the training and validation splits of the dataset. The *make_dataset()* function takes the name of the dataset and loads the training and the validation splits as follows.

```
import tensorflow_datasets as tfds

def make_dataset(name):
  loadfn = lambda x: tfds.load(name, split=x)
  train_ds = loadfn('train')
  val_ds = loadfn('validation')

  return train_ds, val_ds

train_ds, val_ds = make_dataset('oxford_flowers102')
```

The dataset contains variable sized samples. Go ahead and resize them to 264x264 size. This is a required step because our model expects fixed-sized images.

```
import tensorflow as tf

# Target image size
IMG_SIZE = 264

def dsitem_to_tuple(item):
  return (item['image'], item['label'])

def resize_image(image, label):
  image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])
  image = tf.cast(image, tf.uint8)
  return image, label

train_ds = train_ds.map(dsitem_to_tuple).map(resize_image).cache()
val_ds = val_ds.map(dsitem_to_tuple).map(resize_image).cache()
```

```
print(train_ds.as_numpy_iterator().next()[0].shape)
print(val_ds.as_numpy_iterator().next()[0].shape)
```

```
(264, 264, 3)
(264, 264, 3)
```

Our dataset is ready. Let's work on the model. We use a pre-trained ResNet50 model with the top (softmax) layer replaced with a new softmax layer with 102 units (one unit for each class). Additionally, we add the recommended resnet preprocessing layer at the bottom (right after the input layer). We compile the model with a sparse cross entropy loss function (discussed in chapter 2) and the adam optimizer.

```
from tensorflow.keras import applications as apps
from tensorflow.keras import layers, optimizers, metrics

DROPOUT_RATE = 0.2
LEARNING_RATE = 0.0002
NUM_CLASSES = 102

def create_model():
  # Initialize the core model
  core_args = dict(input_shape=(IMG_SIZE, IMG_SIZE, 3), include_top=False)
  core = apps.resnet50.ResNet50(**core_args)
  core.trainable = False

  # Create the full model with input, preprocessing, core and softmax layers.
  model = tf.keras.Sequential([
    layers.Input([IMG_SIZE, IMG_SIZE, 3], dtype = tf.uint8),
    layers.Lambda(lambda x: tf.cast(x, tf.float32)),
    layers.Lambda(lambda x: apps.resnet.preprocess_input(x)),
    core,
    layers.Flatten(),
    layers.Dropout(DROPOUT_RATE),
    layers.Dense(NUM_CLASSES, activation='softmax')
  ])

  adam = optimizers.Adam(learning_rate=LEARNING_RATE)
  model.compile(
    optimizer=adam,
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
  return model

model = create_model()
model.summary()
```

```
Downloading data from
https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim
_ordering_tf_kernels_notop.h5
94773248/94765736 [==============================] - 1s 0us/step
94781440/94765736 [==============================] - 1s 0us/step
Model: "sequential"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 264, 264, 3)       0
_________________________________________________________________
lambda_1 (Lambda)            (None, 264, 264, 3)       0
_________________________________________________________________
resnet50 (Functional)        (None, 9, 9, 2048)        23587712
_________________________________________________________________
flatten (Flatten)            (None, 165888)            0
_________________________________________________________________
dropout (Dropout)            (None, 165888)            0
_________________________________________________________________
dense (Dense)                (None, 102)               16920678
=================================================================
Total params: 40,508,390
Trainable params: 16,920,678
Non-trainable params: 23,587,712
_________________________________________________________________
```

Great! We have initialized the dataset and created a model. Let's continue and create a training function. Once we have the training function, we can kick-off the training process. The *train()* is simple. It takes the model, training set and validation set as parameters. It also has two hyperparameters: batch_size and epochs. We use a small batch size because our dataset has just 1020 samples. A large batch size, say 256, will result in a small number (5) of gradient updates per epoch. Finally, it calls the *fit()* method on the model object to start training.

```
from tensorflow.keras.callbacks import ModelCheckpoint

def train(model, tds, vds, batch_size=24, epochs=100):
  tds = tds.shuffle(1000, reshuffle_each_iteration=True)
  batch_tds = tds.batch(batch_size).prefetch(tf.data.AUTOTUNE)
  batch_vds = vds.batch(256).prefetch(tf.data.AUTOTUNE)

  # Save the best weights during training
  tmpl = 'weights-epoch-{epoch:d}-val_accuracy-{val_accuracy:.4f}.h5'
  checkpoints = ModelCheckpoint(tmpl, save_best_only=True, monitor="val_accuracy")

  history = model.fit(
    batch_tds,
    validation_data=batch_vds,
    epochs=epochs,
    callbacks=[checkpoints]
  )

  return history
```

Let's run a baseline training session for 100 epochs without data augmentation.

```
train(model, train_ds, val_ds, batch_size=24, epochs=100)
```

```
Epoch 1/100
43/43 [==============================] - 77s 968ms/step - loss: 4.5983 - accuracy: 0.3833 -
val_loss: 1.8394 - val_accuracy: 0.6833

Epoch 2/100
43/43 [==============================] - 21s 493ms/step - loss: 0.1100 - accuracy: 0.9784 -
val_loss: 2.3430 - val_accuracy: 0.6745
                          xxxxxxxxx Skip to 98th epoch xxxxxxxxx
Epoch 98/100
43/43 [==============================] - 21s 497ms/step - loss: 2.1797e-07 - accuracy:
1.0000 - val_loss: 1.9971 - val_accuracy: 0.7000
Epoch 99/100
43/43 [==============================] - 21s 496ms/step - loss: 1.9424e-07 - accuracy: 1.0000
- val_loss: 1.9958 - val_accuracy: 0.7010
Epoch 100/100
43/43 [==============================] - 21s 497ms/step - loss: 1.6654e-07 - accuracy:
1.0000 - val_loss: 1.9950 - val_accuracy: 0.7010
```

Figure 3-8 shows a plot of validation accuracies of the baseline training run. It indicates that the accuracy stabilized after the first couple of epochs. The top accuracy achieved is 70.10% highlighted with a dot on the curve. Can we do better with the help of image transformation techniques?

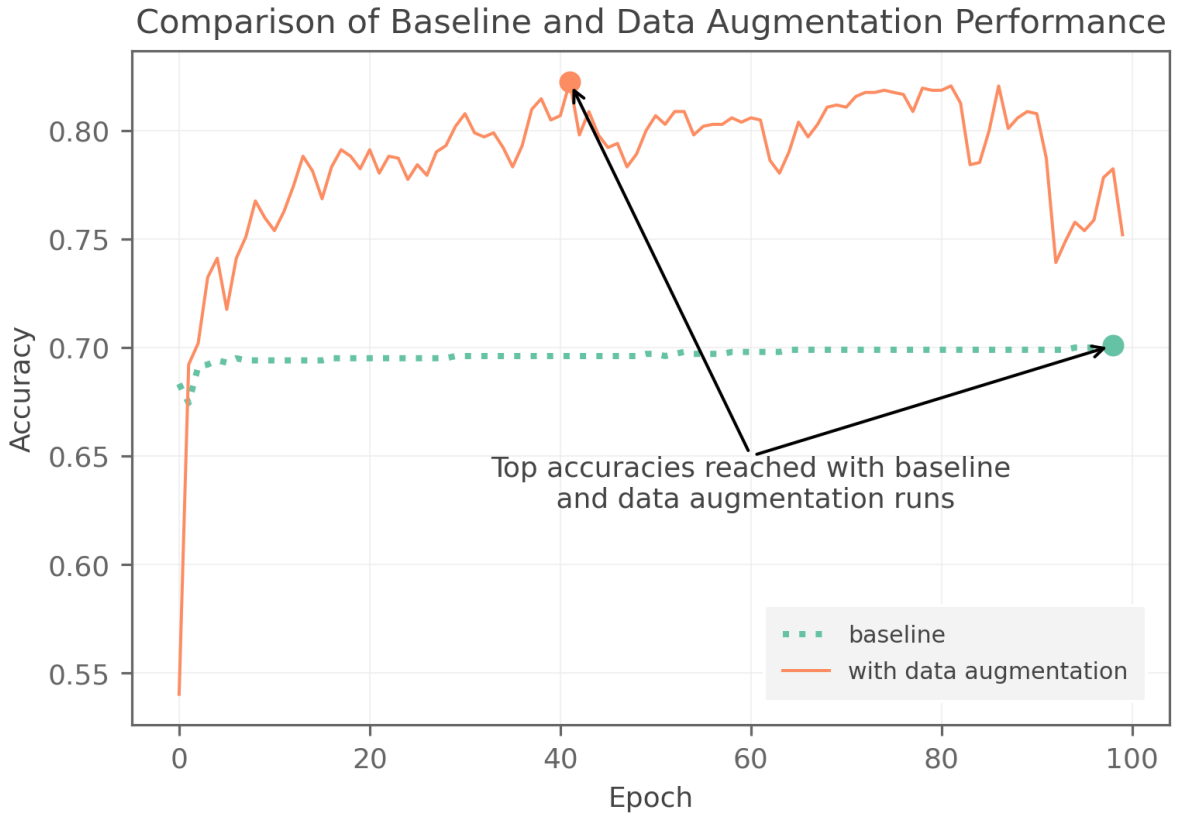


Figure 3-8: Baseline and data augmentation validation accuracies. The two graphs have a clear gap that demonstrates the gains achieved with data augmentation (horizontal flip and rotation transformations).

Let's apply a couple of image transformations to the dataset like a horizontal flip and a random rotation of .1 (value x implies a rotation of $2\pi x$ radians). How did we choose them? Well, rotation is one of the most common transformations. Imagine you are taking a picture of a flower using your phone. The orientation of a camera is unlikely to be the same between two successive pictures. Even though it could be a slight change, it is still a change. The random rotation transformation attempts to simulate that outcome. The random nature of the transformation implies that a value in range [-.1, .1] is chosen randomly and applied to the sample image. The horizontal flip transformation leverages the symmetric nature of flowers to augment the dataset.

We create a layer composite of random flip and rotation. Then, we map each image through this layer to apply the transform. The mapping is done using a handy *map()* method on the tf.data.Dataset object.

```
augs = tf.keras.Sequential([
  layers.RandomFlip("horizontal"),
  layers.RandomRotation(0.1),
])

def augfn(image, label):
  image = tf.expand_dims(image, 0) # Introduce a batch dimension
  image = augs(image, training=True) # Apply augmentation
  image = tf.squeeze(image, axis=0) # Squeeze the batch
  return image, label

train_aug_ds = train_ds.map(augfn)
```

That's it! Simple, isn't it? Let's run the training function again with the augmented dataset.

```
train(model, train_aug_ds, val_ds, batch_size=24, epochs=100)
```

```
Epoch 1/100
43/43 [==============================] - 29s 502ms/step - loss: 14.9726 - accuracy: 0.3549
- val_loss: 12.6783 - val_accuracy: 0.5402
Epoch 2/100
43/43 [==============================] - 29s 500ms/step - loss: 2.5899 - accuracy: 0.8353 -
val_loss: 6.2555 - val_accuracy: 0.6922
                          xxxxxxxxx Skip to 40th epoch xxxxxxxxx
Epoch 40/100
43/43 [==============================] - 29s 499ms/step - loss: 0.0220 - accuracy: 0.9961 -
val_loss: 3.9962 - val_accuracy: 0.8049
Epoch 41/100
43/43 [==============================] - 29s 500ms/step - loss: 0.0035 - accuracy: 0.9980 -
val_loss: 3.9918 - val_accuracy: 0.8069
Epoch 42/100
43/43 [==============================] - 29s 499ms/step - loss: 0.0038 - accuracy: 0.9990 -
val_loss: 3.7707 - val_accuracy: 0.8225
                          xxxxxxxxx Skip to 99th epoch xxxxxxxxx
Epoch 99/100
```

```
43/43 [==============================] - 29s 496ms/step - loss: 0.0317 - accuracy: 0.9971 -
val_loss: 4.6451 - val_accuracy: 0.7824
Epoch 100/100
43/43 [==============================] - 29s 496ms/step - loss: 0.0463 - accuracy: 0.9961 -
val_loss: 5.7786 - val_accuracy: 0.7520
```

With the augmented data, the model achieved a top validation accuracy of 82.25% (epoch 43). The accuracy v/s epoch plot with the augmented data is shown in figure 3-6. The augmented run shows a clear improvement in validation accuracy throughout the training run. You must be wondering, how to revert to epoch 43 training state to get the best performance? Take a closer look at the *train()* function. The *fit()* method on the model object has a callbacks argument. We set a *ModelCheckpoint* object as a callback which saves the best weights for later use!

That's it for the label invariant image transformations! In this section, we presented various image transformation techniques that can be used to augment an image dataset. Using code samples, we explained their effects visually on the original image. Using the oxford flowers classification project, we showed that the augmented run performed much better with a *12% improvement* in the validation accuracy over the baseline run. Figure 3-6 shows that the model with data augmentation achieves the baseline accuracy in fewer epochs, thus it demonstrates *improved* sample efficiency. It is also possible to show that data augmentation *improves* label efficiency of the training process by using fewer labeled samples and achieving the baseline accuracy. Now, let's move to the textual domain and learn some techniques to improve the performance of NLP models. An interesting hands-on project awaits towards the end of the text transformation section as well.

Text Transformations

Like the image transformations, there are simple text transformation techniques that would make a robust model. These techniques involve insertion, deletion, swapping or replacement of words in a sentence. We will discuss some of them in detail in this section. The code is available **here** as a Jupyter notebook for you to experiment.

The following code snippet sets up the modules, functions and variables that will be used later on. It initializes the Natural Language Toolkit (NLTK) and creates a text sequence from a sentence.

```
from random import choice, randint
from keras.preprocessing.text import text_to_word_sequence

# NLTK Import
try:
  from nltk.corpus import wordnet
  # Placeholder search to ensure wordnet data is available.
  wordnet.synsets('hello')
except LookupError as e:
  import nltk
  nltk.download('wordnet')
```

```
"""
  It returns a list of synonyms of the input word.
  The output list may contain the original word.
"""
def synonyms(word):
  results = set()
  for syn in wordnet.synsets(word):
    for lemma in syn.lemmas():
      results.add(lemma.name())

  return list(results)

"""
  It handles the cases when the synonyms for a word are unavailable.
  It returns the original word in such cases.
"""
def synonym_or_self(word):
  return choice(synonyms(word) or [word])

original = 'We enjoyed our short vacation in Mexico'
words = text_to_word_sequence(original) # Tokenize the sentence.
```

Now, let's go through the different text transformations with code examples.

Synonym Replacement is a technique to replace words with their synonyms. It is a simple idea to augment the dataset without compromising the text sentiment. Consider the following two sentences:

Original: We enjoyed our short vacation in Mexico.
*Transformed: We enjoyed our short **holiday** in Mexico.*

The meaning of the sentence is intact after replacing "vacation" with the word "holiday".

```
candidates = ['vacation'] # These are the words that can be replaced with their
synonyms

def syn_transformation(words, candidates):
  transformed_words = []
  for word in words:
    if word in candidates:
      transformed_words.append(synonym_or_self(word))
    else:
      transformed_words.append(word)

  return transformed_words

syn_transformation(words, candidates)
```

```
>> ['we', 'enjoyed', 'our', 'short', 'holiday', 'in', 'mexico']
```

Random Insertion technique inserts a word at a random position in the sentence. The inserted word, typically, is a synonym of one of the words in the sentence. Let's take an example. The python code follows thereafter.

Original: We enjoyed our short vacation in Mexico.
*Transformed: We enjoyed our short **holiday** vacation in Mexico.*

In this example, we perform a random synonym insertion for the word "vacation". The synonym "holiday" got inserted right before the word "vacation". The result is grammatically incorrect. However, the sentiments of the original and the transformed sentences are consistent. This can be used for sentiment analysis. This transformation has two main implications. First, it augments our dataset with additional examples. And second, it teaches the model additional words and the sentiments associated with them.

```
candidates = ['vacation'] # These are the words whose synonyms will be inserted.

"""
  It inserts a synonym for every candidate word at a random position in the word
sequence.
"""
def ins_transformation(words, candidates):
  for candidate in candidates:
    pos = randint(0, len(words) - 1) # Random insertion position for the candidate
    syn_word = synonym_or_self(candidate) # Get a random synonym
    words.insert(pos, syn_word)

  return words

ins_transformation(words.copy(), candidates)
```

```
>> ['we', 'enjoyed', 'our', 'short', 'holiday', 'vacation', 'in', 'mexico']
```

Random Deletion transforms the sentence by deleting a word at random. Here is an example with the python code.

Original: We enjoyed our short vacation in Mexico.
Transformed: We enjoyed ~~our~~ short vacation in Mexico.

The word "our" is deleted from the original sentence. Although the transformed sentence is missing an article, it conveys the original meaning and the sentiment.

```
"""
  It deletes a random word in the input sequence.
"""
def del_transformation(words):
  pos = randint(0, len(words) - 1) # Random deletion position
```

```
  words.pop(pos)

  return words

del_transformation(words.copy())
```

```
>> ['we', 'enjoyed', 'short', 'vacation', 'in', 'mexico']
```

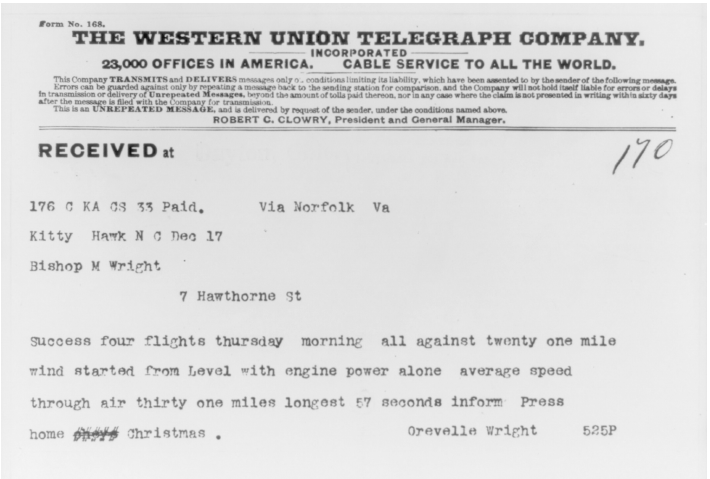
Random Swap swaps two random words in a sentence. Here is an example of a random swap transformation.

Original: We enjoyed our short vacation in Mexico.
Transformed: We enjoyed our short Mexico in vacation.

```
"""
  It swaps two random words in the input sequence.
"""
def swap_transformation(words):
  random_pos = lambda: randint(0, len(words) - 1)
  pos1 = random_pos() # First random position
  pos2 = random_pos() # Second random position

  temp = words[pos1]
  words[pos1] = words[pos2]
  words[pos2] = temp

  return words

swap_transformation(words.copy())
```

```
>> ['we', 'enjoyed', 'our', 'short', 'mexico', 'in', 'vacation']
```

Figure 3-9 is a reproduction of the telegraph sent by Orville Wright in 1903 to inform the press of his successful flights. The author has taken grammatical liberties to compress the information to save dollars. In spite of the obvious errors, the message is loud and clear.



Form No. 158.

THE WESTERN UNION TELEGRAPH COMPANY.

INCORPORATED

23,000 OFFICES IN AMERICA. CABLE SERVICE TO ALL THE WORLD.

This Company TRANSMITS and DELIVERS messages only o . conditions limiting its liability, which have been assented to by the sender of the following message. Errors can be guarded against only by repeating a message back to the sending station for comparison, and the Company will not hold itself liable for errors or delays in transmission or delivery of Unrepeated Messages, beyond the amount of tolls paid thereon, nor in any case where the claim is not presented in writing within sixty days after the message is filed with the Company for transmission.

This is an UNREPEATED MESSAGE, and is delivered by request of the sender, under the conditions named above.

ROBERT C. CLOWRY, President and General Manager.

RECEIVED at 170

176 C KA CS 33 Paid. Via Norfolk Va

Kitty Hawk N C Dec 17

Bishop M Wright

7 Hawthorne St

Success four flights thursday morning all against twenty one mile wind started from Level with engine power alone average speed through air thirty one miles longest 57 seconds inform Press home ~~Christmas~~ Christmas .

Orevelle Wright 525P

Figure 3-9: This telegram was sent by Orville Wright in December 1903 from Kitty Hawk,

North Carolina, following the first successful airplane flight.

In a Spanish speaking country, an elementary level Spanish speaker's response, "estoy ir mercado", sufficiently conveys the information that the person is going to the market. A version of this example could be a native english speaker's response, "I go market", to an elementary level english speaker. Although the native english speaker can formulate a better sentence, a simplified version is more likely to convey their message to the elementary level english speaker.

A transformation mutates a text sample such that the agreement between the text and the original label is intact. In the context of sentiment analysis, the transformation must preserve the original sentiment of the text. For a language translation model, the label sequence and the mutated input must have the same meaning. It is fair to say that heavy mutations would break the association between inputs and the labels. Wei et al.[4] experimented with these transformation techniques and studied the impact of the extent of mutations. They showed that the performance gains start to drop after more than 20% of the text sample is mutated.

The text transformations we have discussed so far are word level transformations. They involve swap, insertion or deletion of a word or words from the text. Next, we are going to discuss techniques that involve sentence or paragraph level transformations.

Random Shuffle is useful for the NLP tasks that involve large text samples, such as text summarization, spam filtering, resume filtering. The basic idea is that shuffling sentences, paragraphs or sections in a text must preserve the original meaning. Paragraph or sentence shuffling of emails should preserve the labels for spam filtering. For example, a resume with shuffled sections preserves the candidate's profile.

Below is an example of a paragraph picked from the Telegram Style page on wikipedia. The first paragraph is the original version. The shuffled version follows it. Barring the *confused* usage of "This" in the shuffled sentence, both the original and the shuffled sentences convey identical information.

Original: "In some ways, "telegram style" was the precursor to the modern language abbreviations employed in "texting" or the use of short message standard (SMS) services such as Twitter. For telegrams, space was at a premium—economically speaking—and abbreviations were used as necessity. This motivation was revived for compressing information into the 160-character limit of a costly SMS before the advent of multi-message capabilities. Length constraints, and the initial handicap of having to enter each individual letter using multiple keypresses on a numeric pad, drove readoption of telegraphic style, and continued space limits and high per-message cost meant the practice persisted for some time after the introduction of built-in predictive text assistance despite it then needing more effort to write (and read)."

---

[4] Wei, Jason, and Kai Zou. "Eda: Easy data augmentation techniques for boosting performance on text classification tasks." *arXiv preprint arXiv:1901.11196* (2019).

Shuffled: " This motivation was revived for compressing information into the 160-character limit of a costly SMS before the advent of multi-message capabilities. For telegrams, space was at a premium—economically speaking—and abbreviations were used as necessity. In some ways, "telegram style" was the precursor to the modern language abbreviations employed in "texting" or the use of short message standard (SMS) services such as Twitter. Length constraints, and the initial handicap of having to enter each individual letter using multiple keypresses on a numeric pad, drove readoption of telegraphic style, and continued space limits and high per-message cost meant the practice persisted for some time after the introduction of built-in predictive text assistance despite it then needing more effort to write (and read)."

Here is a code example that implements random shuffling:

```
# NLTK Import
try:
  from nltk.tokenize import sent_tokenize
  # Placeholder search to ensure wordnet data is available.
  sent_tokenize('hello')
except LookupError as e:
  import nltk
  nltk.download('punkt')

# Input paragraph
paragraph = "" # **Content is removed for brevity**

# Sentence tokenization
sentences = list(sent_tokenize(paragraph))

# Sentence shuffle
shuffle(sentences)

# Paragraph recomposition
shuffled_paragraph = ' '.join(sentences)
```

Enough theory! Let's apply this knowledge to a sentiment classification problem. We will use a small training sample from the IMDB reviews[5] dataset with a larger validation set to measure the performance of text augmentations. Let's dive in!

Project: IMDB Reviews Sentiment Classification

The imdb reviews dataset contains samples of labeled movie reviews. A label value 1 indicates a positive review while a value 0 implies a negative review. We will train a model to classify a review as a positive or a negative review. We use a small training set to simulate the small dataset scenarios. The validation set contains 10000 samples. As in the previous project, we start with setting up the required libraries, and loading the training and validation sets. We leverage the nlpaug library to perform the augmentations. It provides a simple

[5] Maas, Andrew, et al. "Learning word vectors for sentiment analysis." *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 2011.

mechanism to chain multiple augmentations. It can be replaced with any other library per individual preference.

```
%%capture
# We will use nlpaug to augment the text samples.
!pip install nlpaug

import tensorflow as tf
import numpy as np
import tensorflow_datasets as tfds

from tensorflow.keras import layers, optimizers
from pathlib import Path
from time import time

%%capture
(train500_ds, train1000_ds, val_ds), ds_info = tfds.load(
  name='imdb_reviews',
  split=['train[:500]', 'train[:1000]', 'train[60%:]'],
  as_supervised=True,
  with_info=True,
  read_config=tfds.ReadConfig(try_autocache=False)
)
```

We are using two training sets, one with 500 samples and another with 1000 samples. Their specific goal will be explained as we progress. We initialize a few customary variables below to describe the learning rate, the length of the text, the size of the word vector (each word is translated to a vector) and the locations of initial weights and training checkpoints. A sample text is represented with a sequence of words. We have limited the sequence length to 500 words. If a sample is longer, it is truncated. A shorter sample is padded with *null* words. The *null* words have zero word vectors. We will further explain the process of transformation of a sentence to a word vector sequence later on.

```
LEARNING_RATE = 0.001
MAX_SEQ_LEN = 500 # The sentences are truncated to this word count.
WORD2VEC_LEN = 300 # The size of the word vector
CHKPT_DIR = Path('chkpt')
CHKPT_TMPL=                                                    Path(CHKPT_DIR,
'epoch-{epoch:d}-val_accuracy-{val_accuracy:.4f}/weights')
INITIAL_WEIGHTS = Path('initial_weights/weights')

Path(CHKPT_DIR).mkdir(parents=True, exist_ok=True)
```

Let's create our model. It is short and simple. It has two bidirectional LSTM (Long Short-Term Memory Layer) layers and two dense layers interleaved with dropouts. The LSTM layers help to learn the probabilities of words occurring in a sequence. The input shape of our model is *(MAX_SEQ_LEN, WORD2VEC_LEN)* which represents the number of

representative words for a sample text (500 words) and the size of the embedding vector to represent each word (an array of 300 float values) respectively.

```
def create_model():
  model = tf.keras.Sequential([
    layers.Bidirectional(
        layers.LSTM(64, return_sequences=True),
        input_shape=(MAX_SEQ_LEN, WORD2VEC_LEN)
    ),
    layers.Dropout(0.5),
    layers.Bidirectional(layers.LSTM(32, return_sequences=False)),
    layers.Dropout(0.5),
    layers.Dense(20, activation='relu'),
    layers.Flatten(),
    layers.Dense(1, activation='sigmoid'),
  ])
  adam = optimizers.Adam(learning_rate=LEARNING_RATE)

  model.compile(optimizer=adam, loss='binary_crossentropy', metrics=['accuracy'])

  return model

model = create_model()
model.summary()
model.save_weights(INITIAL_WEIGHTS)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bidirectional_2 (Bidirectio  (None, 500, 128)         186880
 nal)

 dropout_2 (Dropout)         (None, 500, 128)          0

 bidirectional_3 (Bidirectio  (None, 64)               41216
 nal)

 dropout_3 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 20)                1300

 flatten_1 (Flatten)         (None, 20)                0

 dense_3 (Dense)             (None, 1)                 21

=================================================================
Total params: 229,417
Trainable params: 229,417
Non-trainable params: 0
_________________________________________________________________
```

We will use the below training function. It is similar to the one used in the image augmentation project earlier.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

def train(model, tds, vds, epochs=100):
  tds = tds.prefetch(buffer_size=tf.data.AUTOTUNE)
  vds = vds.prefetch(buffer_size=tf.data.AUTOTUNE)

  cb_checkpoint = ModelCheckpoint(
    str(CHKPT_TMPL),
    monitor="val_accuracy",
    save_best_only=True,
    save_weights_only=True
  )
  cb_earlystopping = EarlyStopping(
    monitor='val_accuracy',
    patience= 15,
    restore_best_weights=True
  )

  callbacks = [cb_checkpoint, cb_earlystopping]
  history = model.fit(tds, validation_data=vds, epochs=epochs, callbacks=callbacks)

  return history
```

We already have our model, training function, and the training and validation datasets. All that is left is the translation of the sample text to their representative vectors. Below, we define a *vectorize_fn()* function that takes a batch of text samples as input and converts them into embedding[6] vectors. We use the english language model[7] from spacy that comes bundled with these vectors. Let's load it first.

```
%%capture
!python -m spacy download en_core_web_md

import en_core_web_md

nlp = en_core_web_md.load()
```

Use the language model to transform the sentences to sequences of word vectors.

```
def vectorize_fn(text, label):
  def pyfn(text):
    text = np.char.decode(text.numpy().astype(np.bytes_), 'UTF-8').tolist()
```

[6] We have used the words "embeddings" and "vectors" interchangeably in the text. Embeddings is a term for vectors embedded in the vocabulary space such that related words lie close to each other.
[7] A language model is trained on written texts like blogs, news articles and comments to learn the vocabulary, syntax, entities and word embeddings or vectors. Related words have closer word vectors.

```
    # Initialize a representation vector BATCH SIZE x MAX_SEQ_LEN x WORD2VEC_LEN
with zero values
    vector = np.zeros(shape=(len(text), MAX_SEQ_LEN, WORD2VEC_LEN))

    # Fill up zero vector with the actual word vectors from the language model
    for tidx, doc in enumerate(nlp.pipe(text)):
      for didx in range(min(MAX_SEQ_LEN, len(doc))):
        vector[tidx][didx] = doc[didx].vector

    return vector

  vector = tf.py_function(pyfn, inp=[text], Tout=tf.float32)
  vector.set_shape((None, MAX_SEQ_LEN, WORD2VEC_LEN))
  return vector, label
```

Notice that in the 2nd for loop above, we limit the representation to the first *MAX_SEQ_LEN* words in the sequence. We are ready to train. Let's start with the baseline run with 500 training samples.

```
tds = train500_ds.shuffle(500, reshuffle_each_iteration=True)
tds = tds.batch(16).map(vectorize_fn)
vds = val_ds.batch(64).map(vectorize_fn)

# Reset the model state before training
model.load_weights(INITIAL_WEIGHTS)
baseline500_hist = train(model, tds, vds, epochs=100)
```

```
Epoch 1/100
2021-11-09 14:44:20.431426: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN
version 8005
32/32 [==============================] - 366s 12s/step - loss: 0.6981 - accuracy: 0.4860 -
val_loss: 0.6880 - val_accuracy: 0.5446
Epoch 2/100
32/32 [==============================] - 23s 739ms/step - loss: 0.6822 - accuracy: 0.5740 -
val_loss: 0.6779 - val_accuracy: 0.5736
                          xxxxxxxxx Skip to 9th epoch xxxxxxxxx
Epoch 9/100
32/32 [==============================] - 23s 720ms/step - loss: 0.4114 - accuracy: 0.8200 -
val_loss: 0.7851 - val_accuracy: 0.6534
Epoch 10/100
32/32 [==============================] - 22s 706ms/step - loss: 0.3813 - accuracy: 0.8500 -
val_loss: 0.7741 - val_accuracy: 0.6359
                          xxxxxxxxx Skip to 23rd epoch xxxxxxxxx
Epoch 23/100
32/32 [==============================] - 23s 713ms/step - loss: 0.0047 - accuracy: 0.9980 -
val_loss: 2.0963 - val_accuracy: 0.6419
Epoch 24/100
32/32 [==============================] - 23s 705ms/step - loss: 0.0070 - accuracy: 0.9980 -
val_loss: 2.1043 - val_accuracy: 0.6424
```

The baseline run achieves the best validation accuracy of 65.34% in the 9th epoch. It was terminated after the 24th epoch as it did not make any progress for 15 epochs as configured through the early stopping callback in the training function.

Now, let's add some text augmentations to the mix and see if that helps. The *nlpaug* python library offers concise ways to apply sentence, word and character augmentations. We shuffle sentences, substitute, swap, crop and delete words, and apply character misspelling augmentations. We also limit the augmentation probability and maximum number of augmentations for each type. Moreover, we let the augmentation pipeline choose to apply the individual augmentations probabilistically (with a probability of 0.3 in this case).

```
%%capture
from nltk import download as nltk_download
from nlpaug.augmenter import sentence as nas
from nlpaug.augmenter import word as naw
from nlpaug.augmenter import char as nac
from nlpaug import flow as naf

[nltk_download(item) for item in ['punkt', 'wordnet']]

aug_args = dict(aug_p=0.3, aug_max=40)

chain = [
    nas.random.RandomSentAug(**aug_args),
    naw.RandomWordAug(action='delete', **aug_args),
    naw.RandomWordAug(action='swap', **aug_args),
    naw.RandomWordAug(action='crop', **aug_args),
    naw.RandomWordAug(action='substitute', **aug_args),
    nac.KeyboardAug()
]
flow = naf.Sometimes(chain, pipeline_p=0.3)
```

The *nlpaug_fn()* function just wraps up the augmentation calls in a *tf.py_function*, a tensorflow way to call python code.

```
def nlpaug_fn(aug):
  def pyfn(text):
    text = text.numpy().decode("utf-8")
    text = aug.augment(text)
    return text

  def aug_text_fn(text, label):
    aug_text = tf.py_function(pyfn, inp=[text], Tout=tf.string)
    aug_text.set_shape(text.shape)

    return aug_text, label

  return aug_text_fn
```

Now that the augmentation functions are ready, prepare the training and the validation sets, and kick-off a run.

```
tds = train500_ds.shuffle(500, reshuffle_each_iteration=True)
tds = tds.map(nlpaug_fn(flow)).batch(16).map(vectorize_fn)
vds = val_ds.batch(64).map(vectorize_fn)

# Reset the model state before training
model.load_weights(INITIAL_WEIGHTS)
nlpaug500_hist = train(model, tds, vds, epochs=100)
```

```
Epoch 1/100
32/32 [==============================] - 25s 778ms/step - loss: 0.6983 - accuracy: 0.4940 -
val_loss: 0.6925 - val_accuracy: 0.5201
Epoch 2/100
32/32 [==============================] - 24s 768ms/step - loss: 0.6927 - accuracy: 0.5180 -
val_loss: 0.6993 - val_accuracy: 0.5017
                          xxxxxxxxx Skip to 32nd epoch xxxxxxxxx
Epoch 33/100
32/32 [==============================] - 25s 756ms/step - loss: 0.2065 - accuracy: 0.9280 -
val_loss: 0.7391 - val_accuracy: 0.7719
Epoch 34/100
32/32 [==============================] - 25s 796ms/step - loss: 0.2284 - accuracy: 0.9240 -
val_loss: 0.7597 - val_accuracy: 0.7148
                          xxxxxxxxx Skip to 47th epoch xxxxxxxxx
Epoch 47/100
32/32 [==============================] - 26s 787ms/step - loss: 0.1165 - accuracy: 0.9660 -
val_loss: 0.8536 - val_accuracy: 0.7422
Epoch 48/100
32/32 [==============================] - 25s 788ms/step - loss: 0.1479 - accuracy: 0.9500 -
val_loss: 0.8756 - val_accuracy: 0.7330
```

The augmented run achieves the best validation accuracy of 77.19% in the 33rd epoch before termination at 48th epoch. It is an *11.85% improvement* over baseline accuracy. Now, let's double the training data size and evaluate its baseline performance. Recall that we have already prepared a 1000 sample training set.

```
tds = train1000_ds.shuffle(500, reshuffle_each_iteration=True)
tds = tds.batch(16).map(vectorize_fn)
vds = val_ds.batch(64).map(vectorize_fn)

model.load_weights(INITIAL_WEIGHTS)
baseline1000_hist = train(model, tds, vds, epochs=100)
```

```
Epoch 1/100
2021-11-09 15:38:34.694059: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN
version 8005
63/63 [==============================] - 380s 6s/step - loss: 0.6932 - accuracy: 0.5150 -
val_loss: 0.6847 - val_accuracy: 0.5435
Epoch 2/100
63/63 [==============================] - 40s 627ms/step - loss: 0.6638 - accuracy: 0.6170 -
val_loss: 0.6631 - val_accuracy: 0.6120
                          xxxxxxxxx Skip to 12th epoch xxxxxxxxx
Epoch 12/100
```

```
63/63 [==============================] - 40s 631ms/step - loss: 0.1917 - accuracy: 0.9350 -
val_loss: 0.7626 - val_accuracy: 0.7076
Epoch 13/100
63/63 [==============================] - 40s 632ms/step - loss: 0.1434 - accuracy: 0.9550 -
val_loss: 0.9277 - val_accuracy: 0.7477
                        xxxxxxxxx Skip to 27th epoch xxxxxxxxx
Epoch 27/100
63/63 [==============================] - 39s 625ms/step - loss: 0.0076 - accuracy: 0.9980 -
val_loss: 1.6582 - val_accuracy: 0.7257
Epoch 28/100
63/63 [==============================] - 39s 623ms/step - loss: 0.0680 - accuracy: 0.9800 -
val_loss: 1.5581 - val_accuracy: 0.6633
```

The baseline 1000 samples run achieves top accuracy of 74.77% in epoch 13. The augmented 500 samples run beats it by *2.42%*. Hence, the augmented run is *label efficient*. The baseline 500 run is slightly more sample efficient than the augmented run because it achieves a higher validation accuracy an epoch before the corresponding augmented run. However, the top accuracy of the augmented run is *11.85% more* than the baseline run. Figure 3-10 shows a comparison of validation accuracies of the three runs. The augmented run clearly performs better than both the baseline runs.

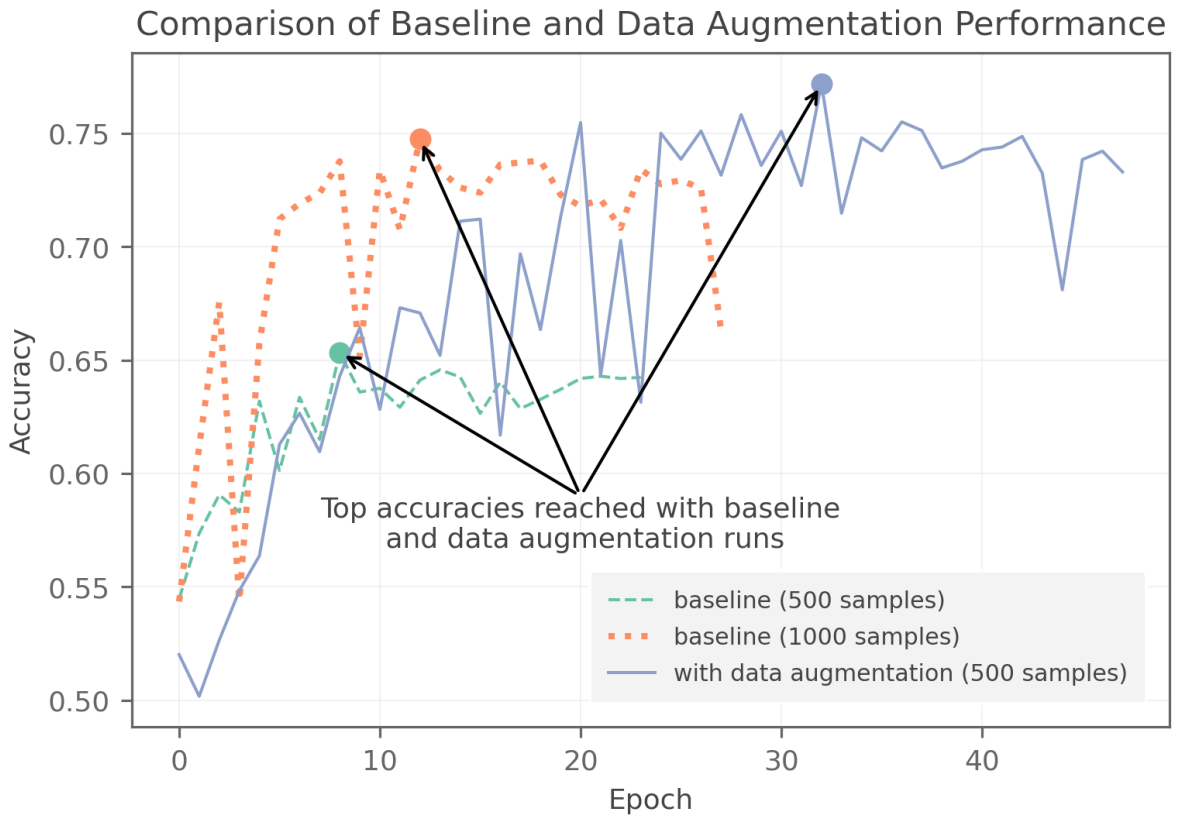


Figure 3-10: Validation accuracies of the three runs: baseline 500 samples, data augmentation (500 samples) and baseline 1000 samples. The graphs clearly show that the augmented run achieves the best accuracy among the three runs. The augmented run is label efficient because it achieves a higher (2.42%) accuracy with fewer samples. Both the augmented and the baseline 500 runs almost match the sample efficiencies (baseline 500 achieves a higher accuracy one epoch sooner than the augmented run).

That's all for label invariant transformations. The key benefit of these transformations is that they are intuitive and can be applied without changes to the model architecture. Their benefit is clear in the low data situations as demonstrated through the projects. In the next section we will discuss label invariant transformations which transform both the inputs and the labels.

Label Mixing Transformations

The transformations we have discussed so far operate on a per sample basis without altering the labels. In the rotation example (figure 3-6), the rotation transformation is applied to an image of the whale. The class labels before and after the rotation are identical. In contrast, the label mixing transformations operate over multiple samples together and recalculate the class labels. However, both are powerful techniques that can be applied individually or in conjunction to produce additional samples to complement arbitrary datasets.

Human beings can recognize overlapping images. Take a look at figure 3-11. The center images are the originals. The images on the left and the right are *mixed images* obtained after applying *label mixing* transformations. We, as human beings, can identify with ease that the mixed images contain glimpses of both turtle and tortoise images. Label mixing techniques extend this idea to the deep learning field which has traditionally relied on *hard labels*[8]. The labels produced by the mixing techniques are *soft labels* which encode the probabilities of an input belonging to the target classes.

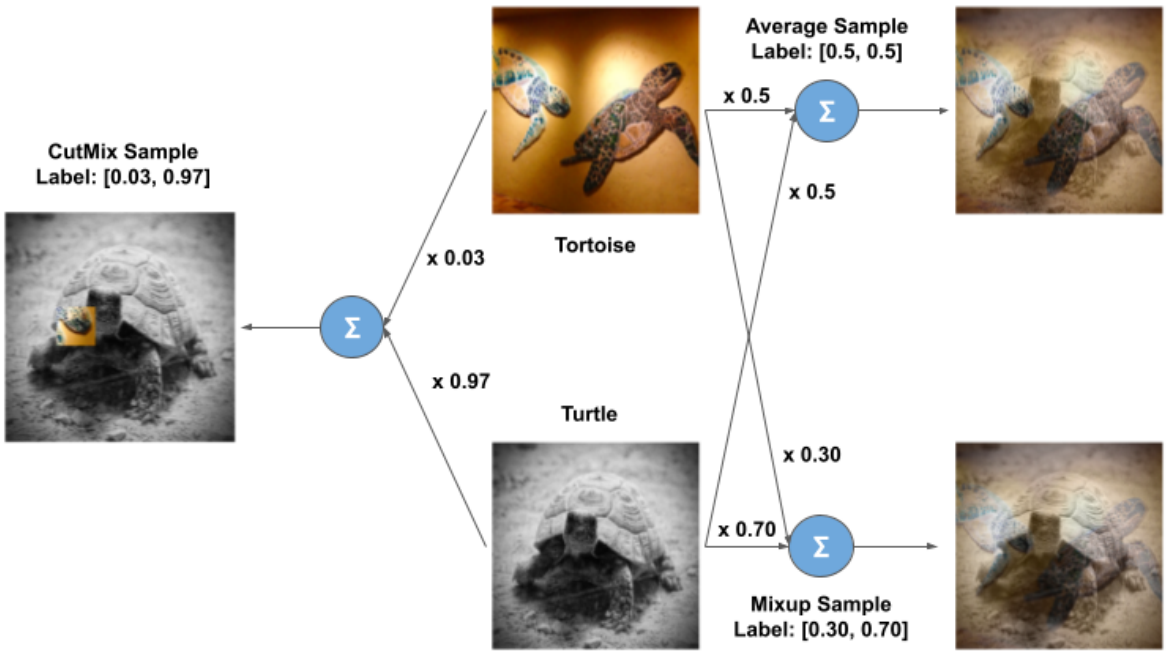


Figure 3-11: The image on the left is a *cut-mix* of turtle (3%) and tortoise (97%) images in the center column and the top-right image is their *average mix*. Bottom-right is a *mixup* of turtle

[8] When the samples in a dataset are assigned a unique label, those labels are called *hard labels*.

(30%) and tortoise (70%) images. The top-center is a turtle (resized) image from Open Images Dataset V6 and authored by Joy Holland. The bottom-center is a tortoise (resized) from Open Images Dataset V6 and authored by J. P. Both the images are licensed under CC BY 2.0.

For example, the top-right image is an *average mix* of turtle and tortoise images in the center column. The *average mixing* is a label mixing technique that averages the sample images to produce the mixed image. The original turtle and tortoise image labels are one-hot encoded as $[1, 0]$ and $[0, 1]$ respectively. The label (soft label) for the *average mixed* sample is $[0.5, 0.5]$ which indicates its probability of being a turtle is 50% and that of it being a tortoise is also 50%. We have used the average mixing as an example. In practice, there are various ways of mixing images together. Let's continue and discuss some of them in detail.

Mixup[9] is a variation of the average mixing technique which mixes the two samples with probability λ and $(1 - \lambda)$. The value of λ is sampled using a probability distribution. It is worth mentioning that the average mixing technique is a special case of mix-up with a fixed $\lambda = 0.5$. The equations shown below mix two samples (X_1,Y_1) and (X_2, Y_2) to create a mixed sample (X_{mixed}, Y_{mixed}):

$$\begin{aligned} \lambda &= BetaDistribution(\alpha, \alpha), \quad \alpha = \text{shape parameter} \\ X_{mixed} &= \lambda * X_1 + (1 - \lambda) * X_2 \\ Y_{mixed} &= \lambda * Y_1 + (1 - \lambda) * Y_2 \end{aligned}$$

The bottom-right image in figure 3-11 shows an example of mixup with $\lambda = 0.30$. The 30/70 mixup example contains "a *smaller amount of turtle*" than the average mix sample.

CutMix[10] replaces a section of pixels in an image sample with a section of pixels from another image sample. The source and the destination sections have the same positional offsets. The left image in figure 3-11 shows the *cut-mixed* turtle and tortoise samples. Similar to mixup, the λ parameter is sampled from a probability distribution and is used as the ratio of the area of the cut-mixed section (A_s^i) to the area of the sample image (A^i).

$$\begin{aligned} \lambda &= BetaDistribution(\alpha, \alpha), & \alpha &= \text{shape parameter} \\ \text{Cut-mixed Area} \quad A_s^i &= A^i * \lambda, & A^i &= \text{Area of the image} \end{aligned}$$

The positional offsets of the sections are sampled from a probability distribution as follows:

$$\begin{aligned} \text{Section Size} \quad & s_s = \lfloor \sqrt{A_s^i} \rfloor \\ \text{X-offset} \quad & x_s = RandomUniform(0, s - s_s), \quad s = \text{Image size} \\ \text{Y-offset} \quad & y_s = RandomUniform(0, s - s_s) \end{aligned}$$

---

[9] Zhang, Hongyi, et al. "mixup: Beyond empirical risk minimization." arXiv preprint arXiv:1710.09412 (2017).

[10] Yun, Sangdoo, et al. "Cutmix: Regularization strategy to train strong classifiers with localizable features." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019.

The cut-mixed sample is calculated as follows:

$$X_{ij}^{cut-mixed} = \begin{cases} X_{ij}^{2} & x_s \leq i < (x_s + s_s) \quad y_s \leq j < (y_s + s_s) \\ X_{ij}^{1} & \text{otherwise} \end{cases}$$

$$Y^{cut-mixed} = \lambda * Y^1 + (1 - \lambda) * Y^2$$

The image on the left in figure 3-11 uses $\lambda = 0.03$.

We have learnt a number of transformation techniques and the intuition behind them. The obvious question now is how do they compare in practice. Hendrycks[11] et. al. compared the performances of label invariant and mixing transformations for various models. Table 3-3 shows the classification error as percentages for each transformation technique. AugMix, which randomly applies a transformation technique, achieves the best results.

Dataset	Standard (Random Flip & Crop)	Mixup	CutMix	AugMix
CIFAR-10-C	29.0	23.5	30.3	12.5
CIFAR-100-C[12]	55.6	52.6	55.5	38.3

Table 3-3: Average classification error percentages with different mixing techniques.

These results drive the point that there is no single best transformation technique applicable in every scenario. Rather, we should understand them as perturbations to the datasets to induce small changes. These changes are reflective of real world behaviors. For instance, two separate camera captures of the same object are unlikely to produce identical images at the pixel level.

Leveraging these techniques gives us flexibility to mimic the real world behaviors from the comfort of our desk and save the costs of generating humongous datasets. However, the techniques we have discussed so far leverage existing input samples to create transformed samples. Label invariant transformations transform single samples while label mixing techniques mix two samples together. Can we somehow generate new samples? The next section introduces synthetic sample generation which leverages statistical or deep learning to generate new samples which in turn are fed to improve the target models.

---

[11] Hendrycks, Dan, et al. "Augmix: A simple data processing method to improve robustness and uncertainty." *arXiv preprint arXiv:1912.02781* (2019).

[12] Hendrycks, Dan, and Thomas Dietterich. "Benchmarking neural network robustness to common corruptions and perturbations." *arXiv preprint arXiv:1903.12261* (2019).

Synthetic Sample Generation

Synthetic Sample Generation extends the idea of leveraging the available dataset to generate more samples. While label invariant and mixing transformations use just a few inputs per augmentation, these techniques learn models from a large set of inputs to generate synthetic samples. They are expensive in comparison to the previous two data-augmentation techniques because learning from a large set of inputs to produce synthetic samples requires more computational resources. Nevertheless, for the data scarce scenarios, extra computational resources might still be cheaper than human labor costs to produce training samples. We have chosen *four* techniques for deeper discussion to cover both statistical and deep learning based synthetic data generation models.

Back Translation is a useful technique to generate synthetic data for Machine Translation tasks when training data is plentiful to translate language X to Y but scarce in the other direction from Y to X. Consider the task of translating English language sentences to German. Assume that the training data for *english-german* translation is scarce whereas *german-english* translation pairs are readily available. We train a *forward model* to translate in the intended direction from English to German language using the available data. Additionally, we also train a german-english translation model over german-english translation pairs. Let's call this a *backward model*. Now, we can use these two models in conjunction by feeding the predictions of the forward model as input to the reverse model and using its outputs and inputs as training pairs to boost the forward model.

In figure 3-12, an english sentence "*The whale is jumping **in** the water*" is translated to "*The whale jumps **into** the water*" using the forward and the reverse models. The final english output along with the intermediate german text "*Der Wal springt ins Wasser*" forms a synthetic sample to train the forward model.

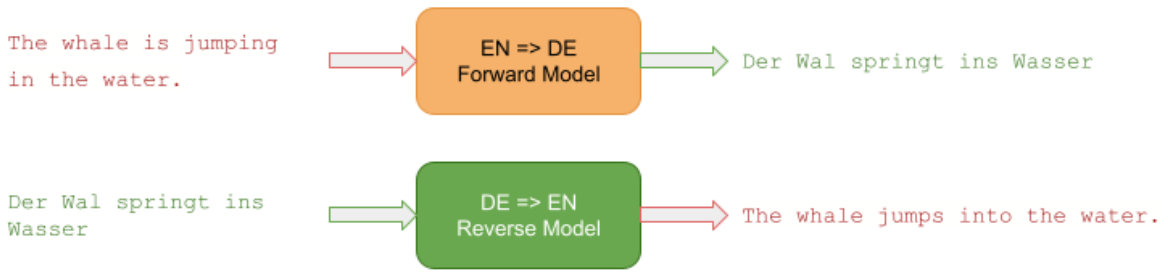


Figure 3-12: An example of back-translation. EN=>DE model in this case is *facebook/wmt19-en-de* and DE=>EN model is *facebook/wmt19-de-en*.

Table 3-4 shows the performance comparison of regular and augmented *english-german* translation models on WMT datasets as reported by Sennrich[13] et. al. The performances are measured in terms of BLEU scores. A higher score indicates better performance. Synthetic data augmented models clearly perform better for both the datasets.

[13] Sennrich, Rico, Barry Haddow, and Alexandra Birch. "Improving neural machine translation models with monolingual data." *arXiv preprint arXiv:1511.06709* (2015).

	BLEU Scores	
Datasets	newstest2014	newstest2015
Original	20.4	23.6
Original + Synthetic	23.8	26.5

Table 3-4: English-German translation performance scores on WMT datasets: newtests2014 and newstest2015. Augmented datasets produce models with higher BLEU scores.

Language Model Based Data Augmentation[14] *(LAMBADA)* generates text samples for classes using the GPT-2[15] language model which has the ability to predict the next word in a given text sequence. LAMBADA concatenates class labels and text samples from the training data to create sequences to finetune GPT-2. The concatenated sequences are just the labels followed by the sample texts with a *<SEP>* in between them. Figure 3-13 shows a fine tuned GPT-2 model generating a synthetic text with a *positive* sentiment for a sentiment classification task. The label at the beginning of the input sequence guides the model to generate text with positive sentiment.

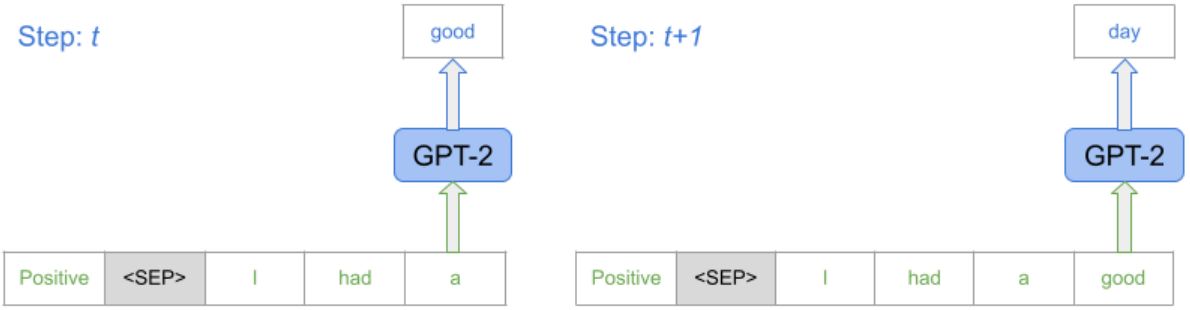


Figure 3-13: Synthetic sample generation using LAMBDA technique. GPT-2 model completes the partial sequence *"Positive<SEP>I had a"* with words *good* and *day.* The final sentence has a positive sentiment as expected.

Table 3-5 shows the performance improvements of various classification models that were trained with a mix of original and synthetic data generated by the LAMBADA technique. These models achieved significantly better performances for ATIS, TREC and WVA datasets when synthetic data was added to the training mix.

Dataset	Improvement(%) with various models		
	BERT	SVM	LSTM
ATIS (Flight Reservations)	58.5	58.7	16.2

[14] Anaby-Tavor, Ateret, et al. "Do not have enough data? Deep learning to the rescue!." *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. No. 05. 2020.
[15] Radford, Alec, et al. "Language models are unsupervised multitask learners." OpenAI blog 1.8 (2019): 9.

TREC (Open Domain Questions)	6.6	2.8	45.0
WVA (Telco Customer Support)	2.1	4.5	23.0

Table 3-5: Accuracy improvements with synthetic data on various classification tasks for various classification models.

The usage of synthetic data generation is not just limited to the textual domain. Let's direct this discussion to the visual domain and introduce techniques like *Synthetic Minority Oversampling Technique*[16] (SMOTE) and *Generative Adversarial Network*[17] (GAN) which can generate synthetic data for images. While SMOTE leverages statistical models for sample generation, GANs are purely deep learning based models.

SMOTE is a technique to oversample the minority classes in an imbalanced dataset. Let's say you are given a handwritten *digit recognition* dataset which contains handwritten samples for digits from *0 to 9*. Further assume that the class *0* is underrepresented such that it has far fewer samples than other classes in the dataset. How would we go about balancing this dataset to reduce the training bias? SMOTE is just the perfect technique for this problem. Given a sample that belongs to the underrepresented class, SMOTE randomly selects one of its k closest neighbors with the same label. It creates a sample that lies between the candidate and the selected neighbor. Figure 3-14 illustrates this idea using a sample from the MNIST dataset. The synthetic image on the right is created at the midpoint between the candidate and its selected neighbor on the left.

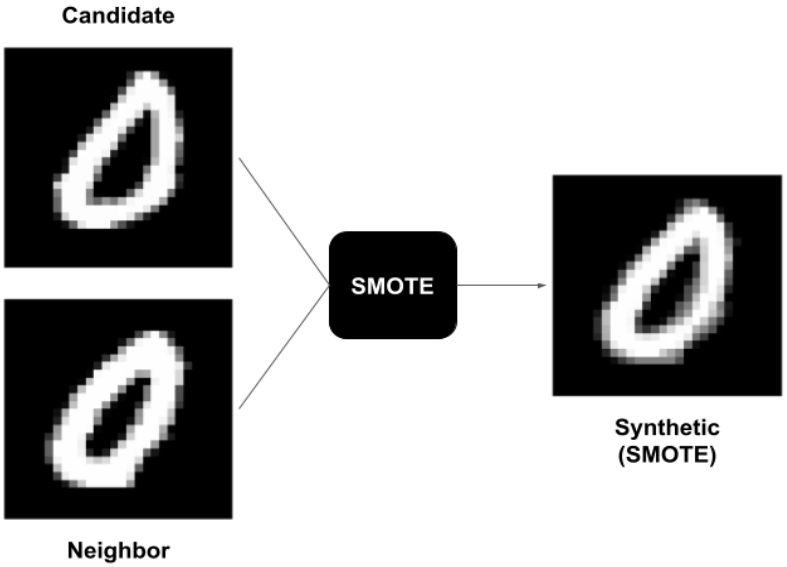


Figure 3-14: A synthetic image for the handwritten digit recognition dataset class *0* is generated using SMOTE. On the left is the candidate (and its neighbor) to be synthesized. The rightmost is the synthesized image for class *0*.

[16] Chawla, Nitesh V., et al. "SMOTE: synthetic minority over-sampling technique." *Journal of artificial intelligence research* 16 (2002): 321-357.
[17] Goodfellow, Ian, et al. "Generative adversarial nets." *Advances in neural information processing systems* 27 (2014).

It is worth noting that techniques like SMOTE are hard to generalize. SMOTE works on datasets like MNIST because the training samples are carefully curated to position the digits in the center of the images. Imagine a task of classifying cats and dogs where the animals are randomly positioned with varied orientations. Such datasets required a more sophisticated approach for sample generation. Let's learn about GAN, which is one such approach that leverages deep learning for this purpose.

A GAN is composed of two neural networks: *a generator network* and *a discriminator network* as shown in figure 3-15. The generator creates synthetic samples from *random inputs* (noise) and the discriminator's job is to classify its inputs as *real* or *fake*. During the training phase, the generator tunes its outputs to look *real* to the discriminator. On the other hand, the discriminator learns to detect its inputs (real and synthetic) as *real* or *fake*. The discriminator works like a fraud detection department in a bank which detects fraudulent transactions. In the bank analogy, a fraudster is a generator who comes up with novel schemes to fool the fraud detection department, a discriminator, in order to accept its transaction as real. Both the fraud detection department and the fraudster evolve over time to be increasingly sophisticated agents.

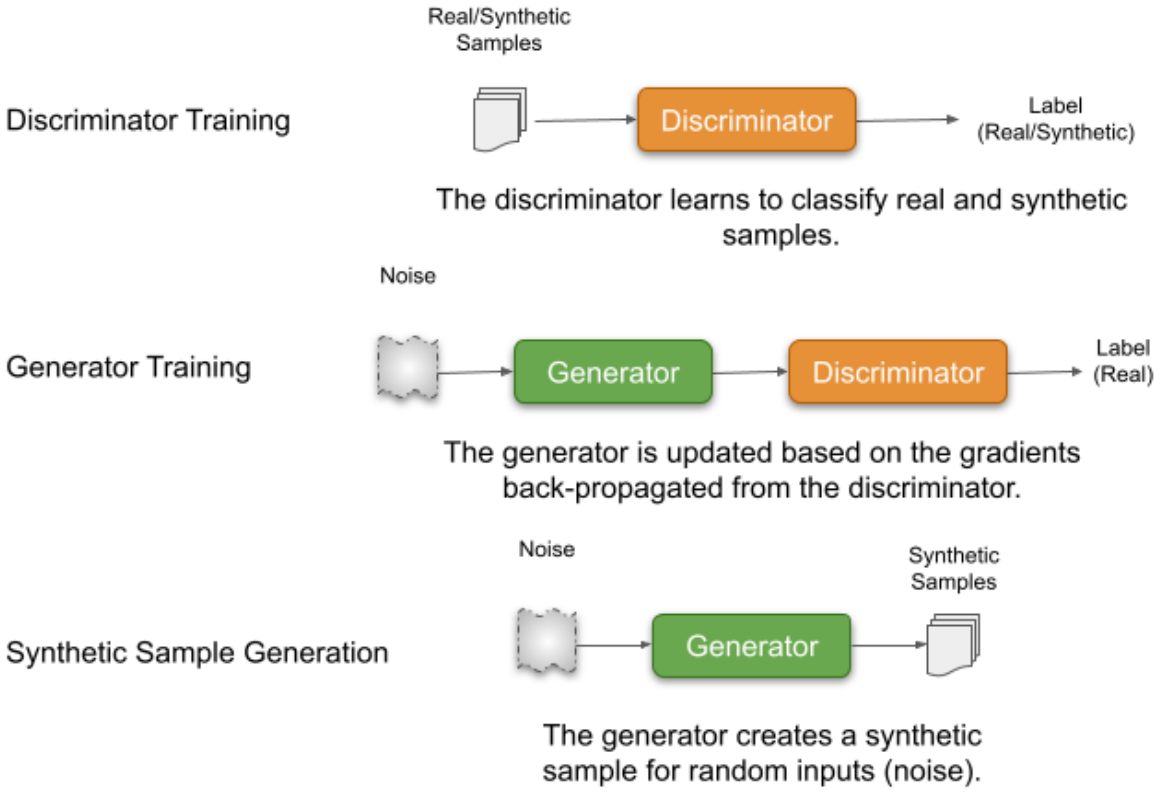


Figure 3-15: Architecture of a Generative Adversarial Network (GAN). It has three phases: discriminator training, generator training and the synthetic sample generation phase. The output of synthetic sample generation phase is used to augment the datasets with scarce samples.

We train the discriminator on both real and synthetic data. Real samples get a positive (real) label and synthetic samples get a negative (fake) label. The discriminator learns to identify samples as real or fake as shown in the top row in figure 3-13. The GAN (middle row) presents the synthetic samples from the generator to the discriminator as real samples. The gradients are back-propagated to the generator. The generator's performance and the quality of synthetic samples improves over time with the feedback (gradients) from the discriminator. Figure 3-16 shows synthetic samples for the MNIST dataset by a GAN that was trained on a subset (10,000 training samples) of the dataset for just 50 epochs.

Figure 3-16: Samples generated by a GAN trained on 10,000 MNIST images for 50 epochs.

In this section, we have discussed a number of techniques which either tweak a sample, mix together two or more samples or synthesize new samples based on statistical or deep learning models learned from the data. The augmented dataset facilitates the model to learn from variations (augmentations) of original samples across training epochs thereby improving the sample and label efficiencies.

The key takeaway for the reader from this section is that augmentation techniques can help to expand the datasets to reduce or eliminate the need for expensive data collection in data scarce situations. Recall the GAN technique where the core idea is to use deep learning models to synthesize data. What if, instead of having a discriminator teach a generator to synthesize images to augment a classification dataset, we can use it to teach a student classification model directly without going through the intermediate step of synthesizing samples? This technique, called Distillation, is explained in the next section.

Distillation

When training models, an important consideration is to avoid overfitting so that the model performs well on unseen data. If the model improves its performance on unseen data, it is said to have generalized better.

Ensembling is a technique often used to improve model generalization. It involves training multiple independent models whose predictions are aggregated using averaging, voting or other similar techniques. The intuition is that it will be less likely to overfit multiple models. It is similar to typical human behavior when making a big decision (a big purchase or an important life event). We discuss with friends and family to decide whether it is a good decision. We rely on their perspectives and life experiences to guide us through the process. Similarly, when ensembling we hope that each individual model would learn its own interpretation of how to solve the problem by using a diverse set of features, which would reduce the likelihood of overfitting and improve generalization. This is great since we can

improve the overall performance when compared to a single model, but now we have multiple models which also multiplies our deployment costs.

Hinton et al.[18], in their seminal work explored how smaller *student* networks can be taught to extract "dark knowledge" from single or ensembles of larger *teacher* models in a slightly different manner. Instead of having to generate synthetic-data, they use the teacher models to generate *soft-labels* for the training samples. The soft labels replace the binary values in the ground-truth labels with the probabilities of the sample image belonging to each class. A student network can then learn from both the soft and the hard ground-truth labels. This technique is called *Distillation*. Figure 3-17 shows the high level design of distillation technique. It shows a teacher network that learns from the training data as usual using hard labels. The trained teacher then makes soft predictions for the training images. A smaller student network leverages these predictions as soft labels which, along with the hard labels, are used to learn the *dark knowledge*.

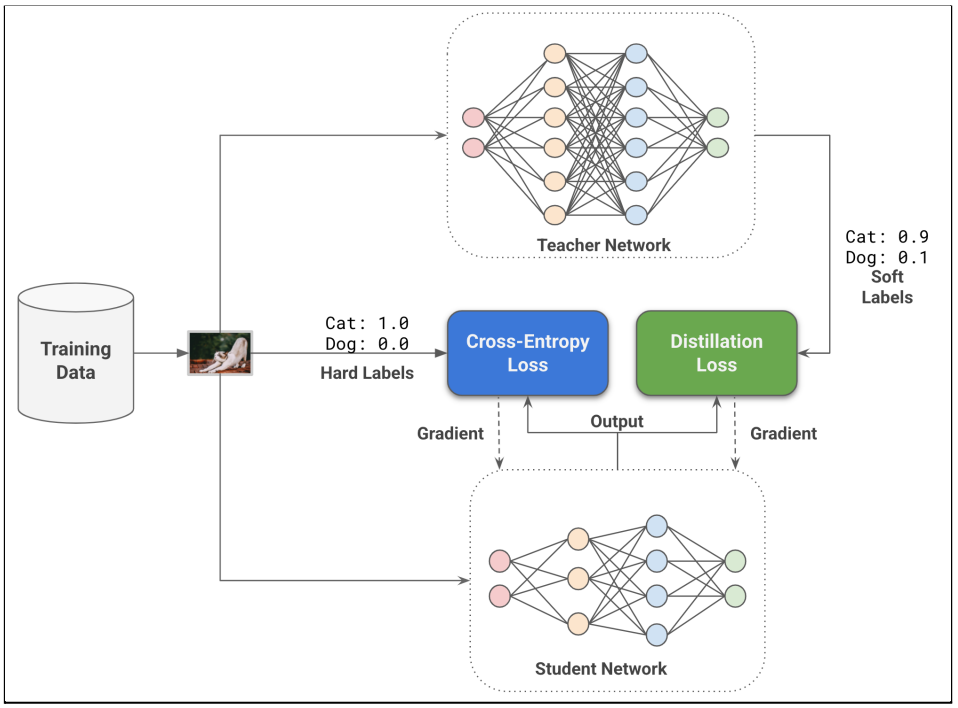


Figure 3-17: Distillation of a smaller student model from a larger pre-trained teacher model. Both the teacher and student models receive the same input. The teacher is used to generate `soft-labels' for the student, which gives the student more information than just hard binary labels. The student is trained using the regular cross-entropy loss with the hard labels, as well as using the distillation loss function which uses the soft labels from the teacher. In this setting, the teacher is frozen, and only the student receives the gradient updates.

Assume that we are training a model to classify images in four classes namely *cat*, *dog*, *pigeon*, and *parrot*. Table 3-6 shows a sample of images and their labels (hard) from the training set. The label values correspond to cat, dog, pigeon and parrot classes in that order.

[18] Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network." arXiv preprint arXiv:1503.02531 (2015).

Note that these are one-hot labels. Exactly one class is assigned a value of 1 and the rest are assigned a value of 0. Hence, a vector of four floating point values needs to be stored with each image. Another alternate and space efficient representation simply stores the index of the correct class, instead of the full vector.

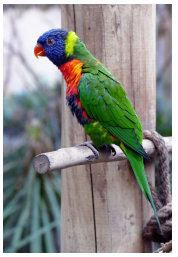
Labels	Images			
	 Source	 Source		
Hard[19]	[1, 0, 0, 0]	[0, 1, 0, 0]	[0, 0, 1, 0]	[0, 0, 0, 1]
Soft	[.80, .15, .03, .02]	[.15, .75, .05, .05]	[.03, .02, .85, .10]	[.05, .05, .20, .79]

Table 3-6: A depiction of differences between hard and soft labels for different classes in the training set. Hard labels are *one-hot* labels and the soft labels are the probabilities of an image belonging to the given classes. The cat and dog images are sourced from wikipedia Cat and Dog pages under CC BY-SA 3.0 license. They are authored by wikipedia users Joaquim Alves Gaspar and Losch respectively. The pigeon and parrot images are sourced under Pexels Free To Use license. They are authored by Pixabay and Anthony respectively.

If we generate predictions for these images using a teacher model, they will contain the probabilities of the image belonging to each of the four classes. These probabilities are soft labels as shown in table 3-6. We can make the following observations about them:

- Notice that each image has probabilities assigned to every class. Hence the *soft*-ness as compared to the hard labels which are *one* and *zero* for the correct and incorrect classes respectively.

- The predicted probability for the cat image actually being a dog is *.15*. The model predicts that the cat image is a pigeon or a parrot with probabilities .03 and .02 respectively. We can infer that the model *thinks* that a cat looks more similar to a dog than a pigeon or a parrot.

  It makes sense because cats and dogs have several common features such as four legs, a tail and similar face structures. We can see the same pattern for pigeons and parrots as well. Overall, dogs and cats' soft probabilities are closer to one another and relatively far from those of pigeons and parrots. Same is true for pigeons and parrots because they are both birds.

[19] Typically, hard labels take float values as well. We have used *integer* values to improve readability.

Distillation captures the relationship between classes which is not represented through the hard (ground-truth) labels in the original dataset. A student model can then learn it using a loss function that minimizes the cross-entropy for both soft and hard labels. The combined loss function is as follows:

$$L = \lambda_1 \cdot L_{\text{ground-truth}} + \lambda_2 \cdot L_{\text{distillation}}$$

In the above equation, $L_{\text{ground-truth}}$ denotes the original loss function (cross-entropy) that uses the hard labels, and $L_{\text{distillation}}$ denotes the distillation loss function which uses the soft labels. λ_1 and λ_2 are hyper-parameters that weigh the two loss functions appropriately.

When $\lambda_1 = 0$ and $\lambda_2 = 1$, the student model is trained with just the distillation loss. Similarly, when $\lambda_1 = 1$ and $\lambda_2 = 0$, it is equivalent to the training with just the hard labels. Usually, the teacher network is pre-trained and frozen during this process, and only the student network is updated.

The distillation and ground-truth loss computations are similar except that the distillation loss uses teacher's outputs as labels. Assume an input tensor $\mathbf{X}$, a hard label tensor $\mathbf{Y}$, a teacher's output tensor $\mathbf{Y^{(t)}}$, a student model's output tensor $\mathbf{Y}^{(s)}$, and the parameters of the student model θ, then the ground truth loss $L_{\text{ground-truth}}$ and the distillation loss $L_{\text{distillation}}$ are defined as follows:

$$L_{\text{ground-truth}} = \text{CrossEntropy}(\mathbf{Y}, \mathbf{Y^{(s)}}; \theta)$$
$$L_{\text{distillation}} = \text{CrossEntropy}(\mathbf{Y^{(t)}}, \mathbf{Y^{(s)}}; \theta)$$

The original paper suggests introducing a 'temperature' value to *soften* the probabilities produced by the teacher model. This might be helpful when the model assigns probabilities very close to 0.0 or 1.0, which makes it almost identical to the regular ground-truth labels.

In this case, we use the 'logits' of the teacher model, which is the input to the final softmax activation layer, and divide the student model's logits tensor by the temperature value (typically >= 1.0). This reduces the skew towards either very large or very low probabilities.

If the input tensor is $\mathbf{X}$, and the teacher model's logits are $\mathbf{Z^{(t)}}$, the teacher model's softened probabilities with temperature T are calculated as follows using the familiar softmax function:

$$\mathbf{Y}_i^{(t)} = \frac{\exp(\mathbf{Z_i^{(t)}}/T)}{\sum_{j=1}^{n} \exp(\mathbf{Z_j^{(t)}}/T)}$$

Note that as T increases, the relative differences between the various elements of $Y^{(t)}$ decreases. This happens because if all elements are divided by the same constant, the

softmax function would lead to a larger drop for the bigger values. Hence, as the temperature T increases, we see the distribution of $Y^{(t)}$ softens further.

Let us try to compute the probabilities using a given logits tensor. We can do this by implementing the Softmax function in NumPy.

```
import numpy as np

# A dummy logits tensor.
logits_tensor = np.array([1.0, 2.0, 3.0])

# Compute e(logits)
exp_tensor = np.exp(logits_tensor)

# Compute Softmax
probabilities_tensor = exp_tensor / np.sum(exp_tensor)

print(probabilities_tensor)
[0.09003057 0.24472847 0.66524096]
```

Notice that the probabilities increase exponentially with an increase in the corresponding logit value. Now, can you change the code such that it takes in the logits_tensor and a temperature value? Try it on a tensor with a skew in the logits value and different values of temperature.

Here is one possible solution:

```
def soften_probabilities(logits_tensor, temperature=1.0):
 # Compute e(logits)
 exp_tensor = np.exp(logits_tensor / temperature)
 # Compute Softmax
 probabilities_tensor = exp_tensor / np.sum(exp_tensor)

 return probabilities_tensor

> soften_probabilities(np.array([1.0, 5.0, 10.0]), temperature=1.0)
array([1.2257e-04, 6.6920e-03, 9.9319e-01])

> soften_probabilities(np.array([1.0, 5.0, 10.0]), temperature=2.0)
array([0.0102, 0.0751, 0.9148])

> soften_probabilities(np.array([1.0, 5.0, 10.0]), temperature=5.0)
array([0.1078, 0.2399, 0.6522])

> soften_probabilities(np.array([1.0, 5.0, 10.0]), temperature=10.0)
array([0.202 , 0.3013, 0.4967])
```

The class probabilities at various temperatures are shown in figure 3-18. As you can observe, when the temperature is 1.0, Class 2 has been assigned a probability of 0.9931. In this case, the probabilities assigned to Class 0 and Class 1 are close to 0. As we increase the temperatures, the distribution of probability becomes less skewed. The right temperature is a hyper-parameter that would need to be experimented with.

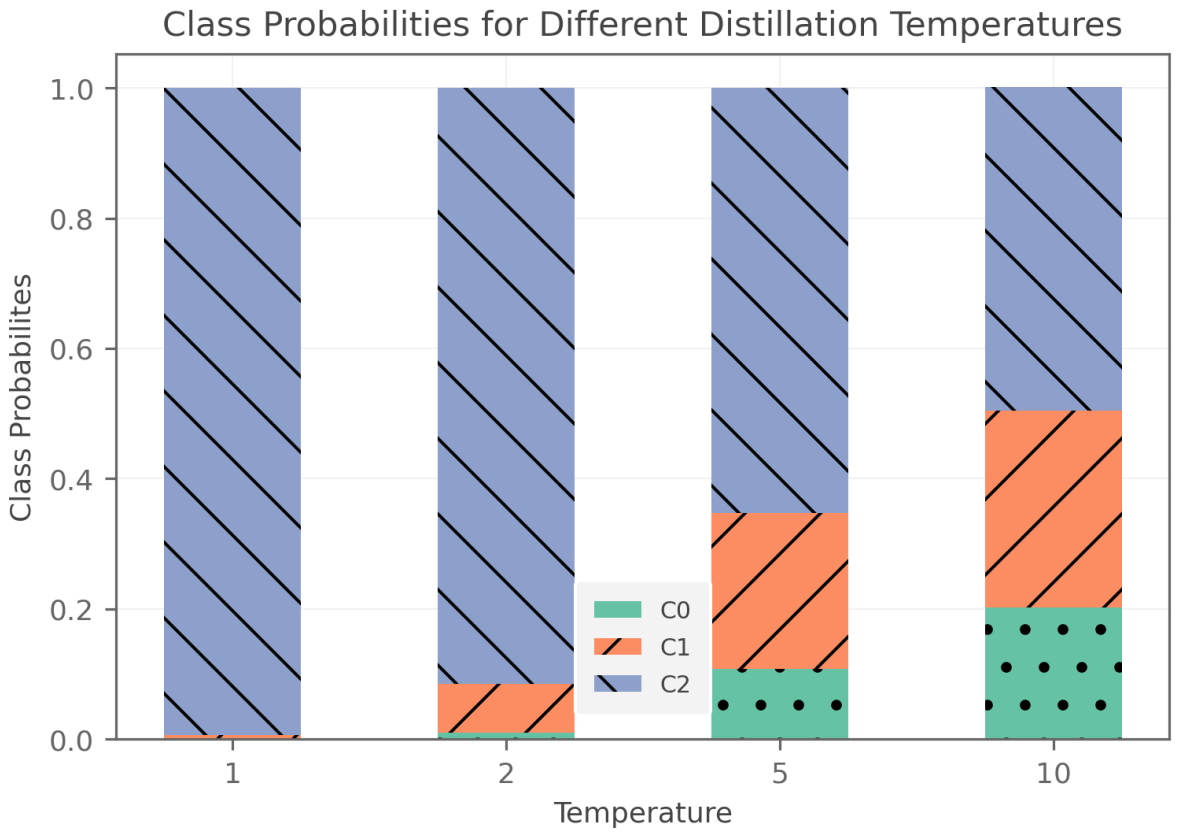


Figure 3-18: Output class probabilities at various distillation temperatures. They exhibit an increase in noise with temperature.

Let us now try to apply distillation in its entirety for a model trained to classify speech commands into one of 12 classes, using the speech commands dataset.The classes

Project: Distillation of a Speech Model.

The code for this project is available **here** as a Jupyter notebook.

Let's start off with doing the regular imports:

```
import numpy as np
import tensorflow_datasets as tfds
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.losses as losses
```

We will install the pydub dependency required by the tensorflow_datasets package for processing audio data, and load the speech_commands dataset from TFDS.

```
!pip install pydub
data_ds = tfds.load(
   name='speech_commands',
   read_config=tfds.ReadConfig(try_autocache=False)
)
```

Next, we introduce a function to preprocess the dataset. It ensures that all the audio-clips are of the same length, and applies the Short-Time Fourier Transform on the clips.

```
def get_processed_ds(ds, num_examples, max_len=16000, num_classes=12):
 """Get the processed dataset."""
 i = 0
 x = np.zeros([num_examples, max_len], dtype=np.float32)
 y = np.zeros([num_examples, 1], dtype=np.uint8)

 for i, example in enumerate(tfds.as_numpy(ds)):
   audio_clip = example['audio']
   label = example['label']

   # Trim and pad the audio clips to the maximum length.
   audio_clip = audio_clip[:max_len]
   audio_clip = np.pad(audio_clip, (0, max_len - audio_clip.shape[0]))
   audio_clip = np.reshape(audio_clip, (max_len)).astype(np.float32)

   x[i] = audio_clip
   y[i] = label

   # i = i + 1
   if i + 1 >= num_examples:
     break

 # STFT to extract the fourier transform on sliding windows of the input.
 # Apply STFT on the audio data, but keep only the magnitude.
 x = tf.abs(tf.signal.stft(x, frame_length=256, frame_step=128))

 # Convert the labels to a one-hot vector.
 y = keras.utils.to_categorical(y, num_classes)
 return x, y

x_train, y_train = get_processed_ds(data_ds['train'], 16000)
x_test, y_test = get_processed_ds(data_ds['test'], 4890)
```

Next we introduce some callbacks to save (and load) the best checkpoint seen so far. We use the standard ModelCheckpoint callback to be able to keep track of and save the best checkpoint, and the categorical accuracy as the metric to monitor. The callback will save the checkpoint with the maximum accuracy during evaluation. The load_best_checkpoint

callback simply restores the weights into the given Model object from the given checkpoint path. We are going to use these callbacks in future projects as well.

```
import os

# Now let us create a callback for saving the best checkpoint so far.
# It tries to find the checkpoint with the maximum categorical accuracy.
# We will provide this to the model.fit function.

CHECKPOINTS_DIR='checkpoints'

# Create a directory for the checkpoints.
!mkdir -p $CHECKPOINTS_DIR

def best_checkpoint_callback(model_name):
  checkpoint_dir_path = os.path.join(CHECKPOINTS_DIR, model_name)
  checkpoint_path = os.path.join(checkpoint_dir_path, model_name)
  return tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    save_weights_only=True,
    monitor='val_categorical_accuracy',
    mode='max',
    save_best_only=True)


def load_best_checkpoint(model, model_name):
  checkpoint_dir_path = os.path.join(CHECKPOINTS_DIR, model_name)
  checkpoint_path = os.path.join(checkpoint_dir_path, model_name)
  model.load_weights(checkpoint_path)
  return model
```

Now we define a simple convolutional neural network (CNN) based model where we can configure the number of parameters by providing a width_multiplier parameter. The higher the value, the larger the number of parameters in the model. Apart from that, the model is fairly straight-forward with Conv, Dense (fully-connected), and Dropout layers.

```
def base_model(
   num_classes=12,         # Number of classes in the model.
   width_multiplier=1.0,   # Factor for scaling the network size.
   params={},              # Additional hyper-params.
):
  # Use the width_multiplier for scaling the network, the larger the value,
  # the larger the network. Keep a bound on the width though.
  w = min(max(width_multiplier, 0.05), 3.0)

  inputs = keras.Input(shape=input_shape)
  x = inputs

  # Create a regularizer to be used.
  reg = keras.regularizers.l2(params.get('l2_reg_weight', 2e-4))
```

```
# Find the dropout rate for helping with further regularization.
dropout_rate = params.get('dropout_rate', 1.0)

# We will keep the first 'block' of layers scale invariant.
x = layers.Conv1D(
    32, (9), padding='same', activation='relu', kernel_regularizer=reg)(
        x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(
    32, (9), padding='same', activation='relu', kernel_regularizer=reg)(
        x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling1D(pool_size=(4))(x)
x = layers.Dropout(dropout_rate)(x)

# A short hand for the round method.
r = lambda v: round(v)

# We will start to scale layers from here on.
x = layers.Conv1D(
    # Note that the number of filters grows / shrinks with `w`.
    r(64 * w),
    (9),
    padding='same',
    activation='relu',
    kernel_regularizer=reg)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(
    r(64 * w),
    (9),
    padding='same',
    activation='relu',
    kernel_regularizer=reg)(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling1D(pool_size=(4))(x)
x = layers.Dropout(dropout_rate)(x)

x = layers.Conv1D(
    r(128 * w),
    (9),
    padding='same',
    activation='relu',
    kernel_regularizer=reg)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(
    r(128 * w),
    (9),
    padding='same',
    activation='relu',
```

```
      kernel_regularizer=reg)(x)
  x = layers.BatchNormalization()(x)
  x = layers.MaxPooling1D(pool_size=(4))(x)
  x = layers.Dropout(dropout_rate)(x)

  x = layers.Flatten()(x)
  x = layers.Dropout(dropout_rate)(x)
  x = layers.Dense(r(512 * w), activation='relu', kernel_regularizer=reg)(x)
  x = layers.Dropout(dropout_rate)(x)
  x = layers.Dense(num_classes, kernel_regularizer=reg)(x)
  logits = x
  probabilities = layers.Activation('softmax')(logits)
  outputs = probabilities

  return keras.Model(inputs=inputs, outputs=outputs)
```

We can now compile and train the model as usual, using the standard compile() and fit() APIs.

```
def get_compiled_model(
    width_multiplier,
    num_classes=12,
    params={}):
 """Helper method to create the compiled model."""
 learning_rate = params.get('learning_rate', 1e-3)

 model = base_model(
     num_classes, width_multiplier=width_multiplier, params=params)

 opt = keras.optimizers.Adam(learning_rate=learning_rate)
 model.compile(loss='categorical_crossentropy', optimizer=opt,
               metrics='categorical_accuracy')
 return model

def get_model_name(width_multiplier, distillation=False):
 """A helper method to create a unique identifier for a model."""
 return 'wm_{}{}'.format(
     str(width_multiplier).replace('.', ''),
     ('_dist' if distillation else '')
 )

def train_model(width_multiplier, params={}, batch_size=128, epochs=100):
 """Train a solo model."""
 model_name = get_model_name(width_multiplier, distillation=False)

 model = get_compiled_model(
     width_multiplier=width_multiplier,
     params=params)
 model.summary()
```

```
 model_history = model.fit(
     x_train,
     y_train,
     batch_size=batch_size,
     epochs=epochs,
     validation_data=(x_test, y_test),
     callbacks=[best_checkpoint_callback(model_name)],
     shuffle=True)

 return model, model_history.history
```

Let us train a model with the width_multiplier param set to 1.0. We would like to use it as our teacher model, to use with a smaller student model.

```
width_multiplier = 1.0
params = {
   'learning_rate': 1e-3,
   'dropout_rate': 0.5,
}
model_wm_10, _ = train_model(width_multiplier, params, epochs=50)
```

The model's summary is as follows:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 124, 129)]        0
_________________________________________________________________
conv1d (Conv1D)              (None, 124, 32)           37184
_________________________________________________________________
batch_normalization (BatchNo (None, 124, 32)           128
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 124, 32)           9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 124, 32)           128
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 31, 32)            0
_________________________________________________________________
dropout (Dropout)            (None, 31, 32)            0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 31, 64)            18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 31, 64)            256
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 31, 64)            36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 31, 64)            256
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 7, 64)             0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 64)             0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 7, 128)            73856
```

```
_________________________________________________________________
batch_normalization_4 (Batch (None, 7, 128)            512
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 7, 128)            147584
_________________________________________________________________
batch_normalization_5 (Batch (None, 7, 128)            512
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 1, 128)            0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1, 128)            0
_________________________________________________________________
flatten (Flatten)            (None, 128)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 512)               66048
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 12)                6156
_________________________________________________________________
activation (Activation)      (None, 12)                0
=================================================================
Total params: 397,292
Trainable params: 396,396
Non-trainable params: 896
_________________________________________________________________
Epoch 1/50
125/125  [==============================]  -  36s  272ms/step  -  loss:  1.9889  -
categorical_accuracy: 0.5926 - val_loss: 2.7277 - val_categorical_accuracy: 0.0969
Epoch 2/50
125/125  [==============================]  -  34s  271ms/step  -  loss:  1.5461  -
categorical_accuracy: 0.6311 - val_loss: 2.5237 - val_categorical_accuracy: 0.1429
                        xxxxxxxxx Skip to 45th epoch xxxxxxxxx
Epoch 45/50
125/125  [==============================]  -  33s  268ms/step  -  loss:  0.3998  -
categorical_accuracy: 0.9247 - val_loss: 0.6210 - val_categorical_accuracy: 0.8618
Epoch 46/50
125/125  [==============================]  -  33s  268ms/step  -  loss:  0.4074  -
categorical_accuracy: 0.9209 - val_loss: 0.5831 - val_categorical_accuracy: 0.8806
Epoch 47/50
125/125  [==============================]  -  34s  269ms/step  -  loss:  0.3975  -
categorical_accuracy: 0.9241 - val_loss: 0.6566 - val_categorical_accuracy: 0.8456
Epoch 48/50
125/125  [==============================]  -  33s  268ms/step  -  loss:  0.3954  -
categorical_accuracy: 0.9247 - val_loss: 0.6242 - val_categorical_accuracy: 0.8663
Epoch 49/50
125/125  [==============================]  -  33s  268ms/step  -  loss:  0.3857  -
categorical_accuracy: 0.9276 - val_loss: 0.5835 - val_categorical_accuracy: 0.8693
Epoch 50/50
125/125  [==============================]  -  34s  268ms/step  -  loss:  0.3952  -
categorical_accuracy: 0.9279 - val_loss: 0.5686 - val_categorical_accuracy: 0.8796
```

To use this model as a teacher, we need to extract it's logits. We can achieve this by creating a new Model object which uses the model we just trained, except that it returns the output of the second-last layer, which generates the logits. We will also evaluate this model as usual, because the logit value is proportional to the corresponding probability, as we saw earlier.

```
# Create a new Model object from the previously trained model with just the
# logits layer. We will copy it's weights from the best checkpoint so far.
# This model will be used as the teacher, to extract the logits.
model_wm_10_without_softmax = tf.keras.Model(
   inputs=model_wm_10.inputs,
   outputs=model_wm_10.layers[-2].output)
model_wm_10_without_softmax.compile(metrics='categorical_accuracy')

# Load the weights and evaluate the model.
# Even though the previous Model object's output was probabilities, and this new
# model's output are the logits, it is essentially the same because the logits
# will retain the same order as the probabilities. Thus, for the accuracy metric
# they are identical.
model_wm_10_without_softmax = load_best_checkpoint(
   model_wm_10_without_softmax,
   get_model_name(1.0))
model_wm_10_without_softmax.evaluate(x_test, y_test)
```

```
153/153   [==============================]   -   3s   17ms/step   -   loss:   0.1374   -
categorical_accuracy: 0.8861
[0.13739605247974396, 0.8860940933227539]
```

Let's try to train a much smaller student model (solo for now), with width_multiplier=0.1. This model uses 52,491 parameters as compared to the 397,292 parameters used by the larger model with width_parameter=1.0 (these are reported in the output of the model.summary() method). We skip the training logs below for brevity, and directly report the best evaluation accuracy.

```
# Now let's train a solo model, with width_multiplier=0.1.
width_multiplier = 0.1
params = {
   'learning_rate': 1e-3,
   'dropout_rate': 0.2,
}
model_wm_01, _ = train_model(width_multiplier, params, epochs=50)

# Let us find the best checkpoint and compute its accuracy.
model_wm_01 = load_best_checkpoint(model_wm_01, get_model_name(0.1))
model_wm_01.evaluate(x_test, y_test)
```

```
153/153   [==============================]   -   2s   12ms/step   -   loss:   0.8865   -
categorical_accuracy: 0.7247
[0.8864668011665344, 0.7247443795204163]
```

As we see, the categorical accuracy is 72.47% for the smaller model, as compared to 88.61% for the larger model. We will try to improve on this accuracy.

Let us first establish our teacher model to be the larger model with the softmax layer trimmed.

```
teacher_model = model_wm_10_without_softmax
```

We will now create the distillation labels by stacking both the teacher model's outputs and the ground-truth labels in the same tensor together[20]. Since both of them are of the same dimensions ([batch_size, num_classes]). This allows us to pass a single label tensor to the model containing both the ground-truth labels, as well as the teacher's outputs (logits).

```
# Create the distillation labels by stacking the teacher model's outputs
# and the ground-truth one-hot labels together.
def get_distillation_labels(teacher_model_instance, x_ds, y_ds):
 """Add the teacher logits to the label tensor."""
 # Generate the teacher model's logits.
 teacher_logits = teacher_model_instance.predict(x_ds)
 # Create the stacked tensor.
 y_dist_ds = tf.stack([y_ds, teacher_logits], axis=1)
 return y_dist_ds

y_dist_train = get_distillation_labels(teacher_model, x_train, y_train)
y_dist_test = get_distillation_labels(teacher_model, x_test, y_test)
```

The code below is the crux of the distillation process. The *distillation_loss_fn()* takes in the combined label tensor, and splits it to extract the ground-truth and the logits (which are converted to soft-labels). The individual ground-truth and distillation losses are computed using the cross-entropy loss function as usual, and aggregated together by a *distillation_weight* parameter (defaults to 1.0).

Similarly, the categorical_accuracy method computes the accuracy of the model using the ground-truth labels. The distillation_loss_fn is used as the loss function when compiling the model, and the categorical_accuracy method is used as the metric. This is pretty much all there is to it.

```
def get_compiled_distillation_model(
   width_multiplier=1.0,
   num_classes=12,
   params={}):
 learning_rate = params.get('learning_rate', 1e-3)
 temperature = params.get('temperature', 1.0)
 distillation_weight = params.get('distillation_weight', 1.0)

 model = base_model(
     num_classes, width_multiplier=width_multiplier, params=params)
```

---

[20] Note that it is possible to avoid the need for creating the dataset, by using the teacher to generate the labels during the training of the student. An example is here. This avoids the need to create a new dataset and also get the exact teacher's output for an augmented input, instead of assuming that the output of the teacher remains the same.

However, this approach also requires keeping the model in memory throughout the entire training, which might be expensive when using very large models.

```
 def distillation_loss_fn(y_true_combined, y_pred):
   """Custom distillation loss function."""
   # We will split the y tensor to extract the ground-truth and the teacher's
   # soft-labels.
   y_true_split = tf.split(y_true_combined, 2, axis=1)

   # Create the ground-truth loss function as usual.
   ground_truth = tf.squeeze(y_true_split[0], axis=1)
   gt_loss = losses.CategoricalCrossentropy()(ground_truth, y_pred)

   # Create the distillation loss using the soft-labels.
   logits = tf.squeeze(y_true_split[1], axis=1)
   soft_labels = tf.nn.softmax(logits / temperature)
   dist_loss = losses.CategoricalCrossentropy()(soft_labels, y_pred)

   # Combine the two into a single loss function.
   loss = gt_loss + distillation_weight * dist_loss * (temperature ** 2.0)
   return loss

  def categorical_accuracy(labels, model_pred):
   # Extract the ground-truth labels.
   ground_truth = tf.split(labels, 2, axis=1)[0]
   ground_truth = tf.squeeze(ground_truth, axis=1)

   # Compute the accuracy as usual.
   return tf.keras.metrics.categorical_accuracy(ground_truth, model_pred)

 opt = keras.optimizers.Adam(learning_rate=learning_rate)

 # Compile the model with the custom loss function and metric.
 model.compile(
     loss=distillation_loss_fn,
     metrics=[categorical_accuracy],
     optimizer=opt)
 return model

def train_dist_model(width_multiplier, params={}, batch_size=128, epochs=100):
 model_name = get_model_name(width_multiplier, distillation=True)

 model = get_compiled_distillation_model(
     width_multiplier=width_multiplier,
     params=params)
 model.summary()

 # The change here would be that we are using the distillation labels.
 model_history = model.fit(
     x_train,
     y_dist_train,
     batch_size=batch_size,
     epochs=epochs,
```

```
      validation_data=(x_test, y_dist_test),
      callbacks=[best_checkpoint_callback(model_name)],
      shuffle=True)

  return model, model_history.history
```

Now, we can train the smaller model in a distillation setting. We see that it achieves an accuracy of 81%! This is *an improvement of 7.53%*, which is quite a significant jump in performance.

```
# Train a model with width_multiplier=0.1 in a distillation setting.
width_multiplier = 0.1
params = {
   'learning_rate': 1e-3,
   'dropout_rate': 0.2,
   'distillation_weight': 1.0,
}
model_wm_01_dist, _ = train_dist_model(width_multiplier, params, epochs=50)

model_wm_01_dist = load_best_checkpoint(
   model_wm_01_dist,
   get_model_name(0.1, distillation=True))
model_wm_01_dist.evaluate(x_test, y_dist_test)
```

```
153/153 [==============================] - 2s 12ms/step - loss: 1.2935 -
categorical_accuracy: 0.8100
[1.293522834777832, 0.8100204467773438]
```

As you see, we achieved an impressive improvement over the baseline model. It is also straight-forward to remark that *distillation is label efficient* because we achieved a better accuracy (figure 3-17) with the same number of labels. Distillation can also help with faster convergence, thus it helps with improving sample efficiency too. This is true in our case too, since the distilled model achieves the same accuracy in fewer epochs as shown in figure 3-19 in comparison to the baseline.

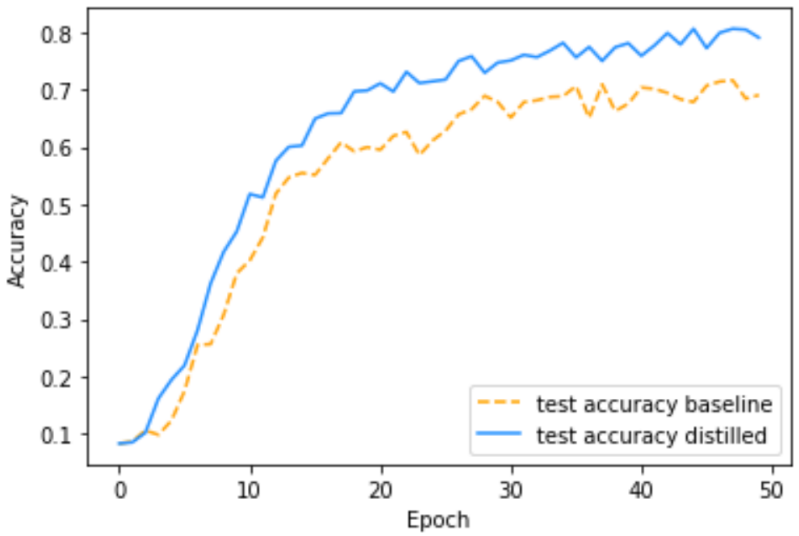


Figure 3-19: Plot of model accuracy over epochs, for the baseline and distilled models. Note how the solid curve (distillation) reaches the same accuracy in fewer epochs when compared to the dashed curve (baseline without distillation), hence it is sample efficient. It also outperforms the baseline by achieving a better accuracy with the same number of labeled examples, hence it is label efficient too.

We hope that this project gave you an insight into how to use distillation for your tasks. The next section gives a quick insight into some of the research into distillation related methods.

Distillation in Scientific Literature

A number of researchers have demonstrated the effectiveness of the distillation technique on single models as well as on ensemble of models. Hinton et al. were able to closely match the accuracy of an ensemble of *ten* models for a speech recognition task with a single distilled model. Urban et al.[21] demonstrated that distillation significantly improves the performance of *shallow* student networks on tasks like CIFAR-10. Sanh et al.[22] used the distillation loss for compressing a BERT[23] model. Their model retains 97% of the performance while being 40% smaller and 60% faster on CPU.

Distillation is a flexible technique that can adapt to different circumstances with minimal changes in the training infrastructure. It allows, for example, for the teacher's predictions to be collected offline if resource constraints prohibit the execution of both the student and the teacher models in tandem. These predictions can then be treated as another data source. When sufficient labeled data is available, there is ample evidence that distillation improves the student model's predictions. Furthermore, distillation can also improve the accuracy of a student using the pseudo-labels generated by the teacher for unlabeled data. Strategies for

---

[21] Urban, Gregor, et al. "Do deep convolutional nets really need to be deep and convolutional?." *arXiv preprint arXiv:1603.05691* (2016).
[22] Sanh, Victor, et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." *arXiv preprint arXiv:1910.01108* (2019).
[23] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).

intermediate-layer distillation have also proved to be effective for complex networks. In such scenarios, a new loss term that minimizes the difference between the outputs of the two networks at a semantically identical intermediate point(s) needs to be added to the loss function.

This concludes our final topic of this chapter. We hope to have provided the reader with insights into the potential ways to improve model efficiencies. Let's summarize our discussion in the next section.

Summary

In this chapter, we introduced two important benchmarks to measure training efficiency and a set of learning techniques to improve on those benchmarks. These benchmarks called sample and label efficiencies can be used as lenses to observe the models during their training phases and evaluate the impact of various learning techniques on the training process.

Data Augmentation techniques introduce the idea that the training samples retain their objective descriptions (labels) after small perturbations. Affine transformations such as rotation or flip augment the training sets with transformed inputs and their original labels which has an effect similar to training a model with a larger training set. We extend the idea of sample perturbations to the labels through label mixing techniques which generate new samples by mixing two or more original samples together. The labels resulting from sample mixing combine the objective descriptions of its constituents. Further, we enter the domain of statistical and deep learning models which synthesize training samples by learning the intrinsic properties of a training set. These models can be viewed as an extension of mixing techniques as statistical or deep learning techniques are equivalent to mixing a large number of samples together to generate new ones.

While data augmentation is associated with generating training samples, distillation concerns itself with transferring the *lessons* (learnings) from a larger model or an ensemble of models to a smaller model. In a way, it is an extension of model based data augmentation techniques like GANs such that instead of training a generator capable of synthesizing training samples, it takes a *shortcut* to directly teach a smaller model capable of performing better than the original model or ensemble of models, thus eliminating the need of synthetic sample generation altogether. Distillation has been empirically shown to improve both the accuracy as well as the speed of convergence of the target models across many domains by enabling the training of smaller models which might otherwise not have an acceptable quality for deployment.

Both of these learning techniques help to improve training efficiencies and contribute to model quality and smaller footprints. This chapter has various exercises and projects to substantiate their benefits. While in the beginning, it might have been unclear how learning techniques fit in the domain of training efficiency, at this point in the chapter we hope that the readers are convinced how these techniques help to achieve similar or better

performance with fewer samples and labels. We strongly recommend our readers to try and experiment with the provided exercises and projects on their own. In the next chapter, we will introduce efficient layers and architectures that you can directly use in the models to achieve improvements in latency, footprint, and performance.