Chapter 4 - Efficient Architectures

"*Any sufficiently advanced technology is indistinguishable from magic.*"
— Arthur C. Clarke, "Hazards of Prophecy: The Failure of Imagination" (1962)

"*Any technology that is distinguishable from magic is insufficiently advanced.*"
— Barry Gehm, quoted by Stan Schmidt in ANALOG magazine (1991)

So far, we have discussed generic techniques which are agnostic to the model architecture. These techniques can be applied in NLP, vision, speech or other domains. However, owing to their incremental nature, they offer limited gains. Sometimes, it can be rewarding to go back to the drawing board and experiment with another architecture that better suits the task. As an analogy, when renovating a house to improve the lighting, it is possible to repaint the walls with bright colors or upgrade to stronger lamps. However, the lighting gains would be substantial if we make structural changes to add a couple of windows and a balcony. Similarly, to gain orders of magnitude in terms of footprint or quality, we should consider employing suitable efficient architectures.

The progress of deep learning is characterized by the phases of architectural breakthroughs to improve on previous results and to drive down the cost of achieving those results. The evolution of multilayer perceptrons was one of the biggest architectural breakthroughs in the field of neural networks. It introduced the idea of stacking layers to learn complex relationships. Convolutional Neural Nets (CNNs) were another important breakthrough that enabled learning spatial features in the input. Recurrent Neural Nets (RNNs) facilitated learning from the sequences and temporal data. These breakthroughs contributed to bigger and bigger models. Although they improved the quality of the solutions, the bigger models posed deployment challenges. What good is a model that cannot be deployed in practical applications!

Efficient Architectures aim to improve model deployability by proposing novel ways to reduce model footprint and improve inference efficiency while preserving the problem solving capabilities of their giant counterparts. In the first chapter, we briefly introduced architectures like depthwise separable convolution, attention mechanism and the hashing trick. In this chapter, we will deepdive into their architectures and use them to transform large and complex models into smaller and efficient models capable of running on mobile and edge devices. We have also set up a couple of programming projects for a hands-on model optimization experience using these efficient layers and architectures. Let's start our journey with learning about embeddings in the next section.

Embeddings for Smaller and Faster Models

We humans can intuitively grasp similarities between different objects. For instance, when we see an image of a dog or a cat, it is likely that we would find them both to be cute. However, a picture of a snake or a grizzly bear might trigger caution or fear.

In a way, we subconsciously *group* these animals in our head. We don't necessarily know everything about a dog and cat, but we know that they are both cute, have been domesticated for a while and are safe. These two animals are more similar to each other than to a random animal like a chimp. Similarly, we know that we should maintain our distance from a snake, and definitely from a grizzly bear, if we ever accidentally cross paths. We build an associative memory when about them over our lifetime. This associative memory helps us visualize the similarities or differences between a pair of animals (or for that matter, people you see, books you read, food you enjoy and so on), without the need of knowing all the encyclopedic data about them.

When working with deep learning models and inputs such as text, which are not in numerical format, having an algorithmic way to meaningfully represent these inputs using a small number of numerical features, will help us solve tasks related to these inputs. Ideally this representation is such that similar inputs have similar representations. We will call this representation an *Embedding*.

An embedding is a vector of features that represent aspects of an input numerically. It must fulfill the following goals:

a) To compress the information content of *high-dimensional* concepts such as text, image, audio, video, etc. to a *low-dimensional* representation such as a fixed length vector of floating point numbers, thus performing dimensionality reduction[1].
b) The low-dimensional representation should allow us to compute the *distance* between any two inputs, which is a measure of their similarity.
c) Similar inputs should have a small distance, and dissimilar inputs should have a larger distance between each other.

Embeddings form a crucial part of modern deep-learning models, and we are excited to explain how they work. In the following section we will explain them through a toy example, but feel free to jump ahead if you are familiar with the motivation behind them.

---

[1] Dimensionality reduction is the process of transforming high-dimensional data into low-dimension, while retaining the properties from the high-dimensional representation. It is useful because it is often computationally infeasible to work with data that has a large number of features. However, not all features might be equally important, thus selecting the most informative features is crucial for making the training step efficient.

In the case of visual, textual, and other multimodal data, we often construct the features by hand (at least in the pre deep learning era). Techniques like Principal Components Analysis, Low-Rank Matrix Factorization, etc. are popular tools for dimensionality reduction. We will explain these techniques in further detail in chapter 6.

A Petting Zoo for Kids

Let's go back to our example of cute and dangerous animals, and represent each animal using two features, say *cute* and *dangerous*. We can assign values between 0.0 and 1.0 to these two features for different animals. The higher the value, the more that particular feature represents the given animal.

In Table 4-1 we manually assigned values for the cute and dangerous features for six animals[2], and we are calling the tuple of these two features an *embedding*, where the two features are its *dimensions*. We will shortly explain how we can use these embeddings.

Animal	Embedding (cute, dangerous)
dog	(0.85, 0.05)
cat	(0.95, 0.05)
snake	(0.01, 0.9)
bear	(0.5, 0.95)
raccoon	(0.5, 0.5)
mouse	(0.01, 0.2)

Table 4-1: A table consisting of embeddings of the various animals, using two features (cute and dangerous), each of which can take a value between 0.0 and 1.0. We manually picked these values for illustration.

Going through table 4-1, cat and dog have high values for the 'cute' feature, and low values for the 'dangerous' feature. On the other hand, a snake is dangerous and not cute for most people. Similarly, a bear might be extremely dangerous, even though stuffed teddy bears have conditioned us into thinking that they might be safe and cute.

A raccoon can seem to be cute (remember Rocket the raccoon from Guardians of the Galaxy?), but it is not the safest animal to be in close proximity, though still safer than a snake or a bear. A domestic mouse on the other hand is not cute but nor very dangerous, but you might want to stay away from it too.

Now that we have a two-dimensional embedding for each animal, where each feature represents one dimension, we can represent the animals on a 2-D plot. The feature cute can be

[2] These feature values are hand-picked based on what we thought was reasonable. The purpose of this toy-example is to illustrate how embeddings work, and we encourage you to try and construct your own example to understand it better.

represented on the x-axis, and the feature dangerous can be represented on the y-axis. Refer to Figure 4-1 for the plot.

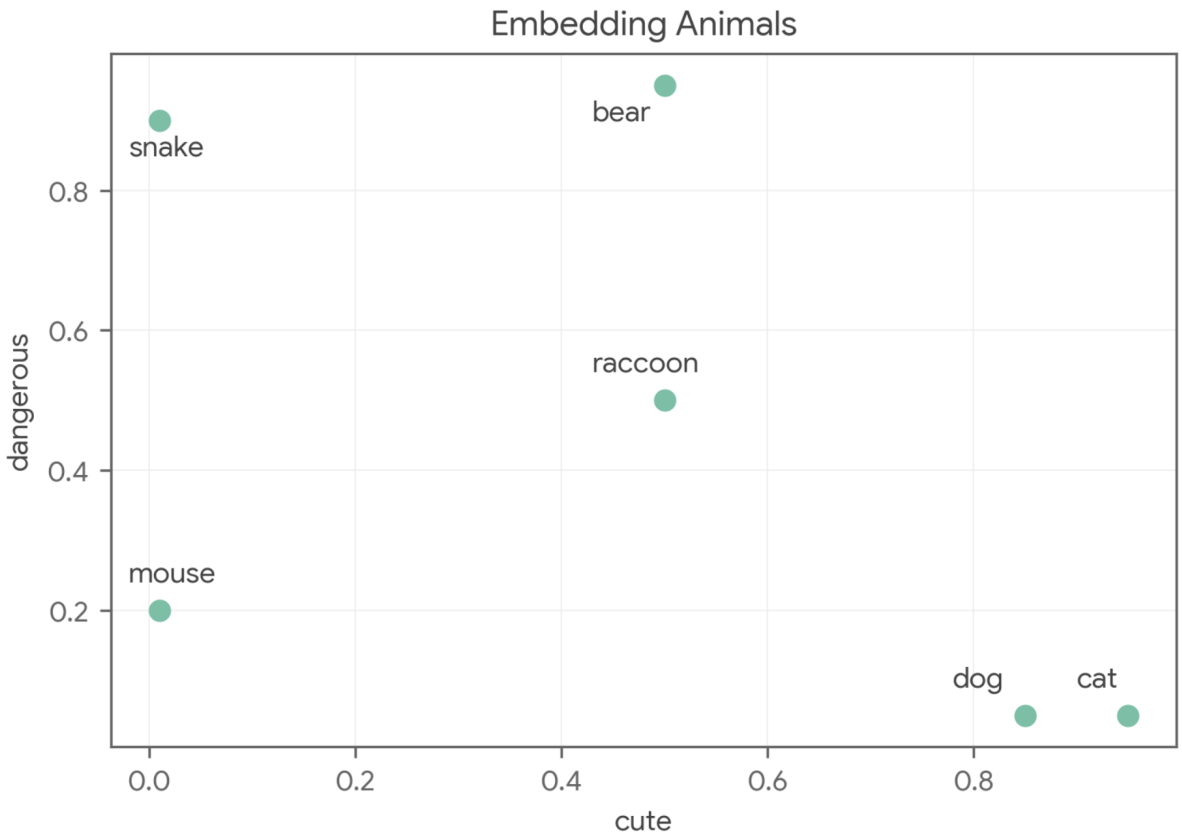


Figure 4-1: A plot of animals being *embedded* in a two-dimensional space, using 'cute' and 'dangerous' as the features, based on Table 4-1. The feature 'cute' occupies the x-axis, and the feature 'dangerous' occupies the y-axis. The animals on the bottom-right are cute and safe to play with. The dangerous animals occupy the top-left area of the plot.

Note how we have compressed the high-dimensional information about animals into just two dimensions, and established a relationship between them purely using numbers, where their relative closeness in the euclidean space on the plot denotes their similarity. We can verify this ourselves, by noting that the dog and cat are represented close to each other. A snake is closer to a bear cub, than to a dog because the former two are dangerous. We can also add other features to further improve our embedding, but for now the two dimensions help us visualize them clearly.

So far, we have created the embeddings and we have visualized them too. Let's start using them!

Consider the scenario of training a model to predict whether kids can safely enjoy interacting with an animal in a petting zoo. This is a binary classification problem in which our model classifies an input into one of the two classes: ‘Suitable’ and ‘Not Suitable’. Since in this scenario we have only a few examples it is easy to manually assign them a label identifying which class a given animal belongs to.

Puppies and cats would make instant favorites in the petting zoo because they are both cute and safe to handle. A snake, bear cub, or a raccoon would not be appropriate because they are dangerous animals. Similarly, the mouse will not make the cut because while it is not dangerous, not many people will find it cute.

Refer to Table 4-2 for our assigned labels.

Animal	Embedding (cute, dangerous)	Label: (suitable for the petting zoo)?
dog	(0.85, 0.05)	Suitable
cat	(0.95, 0.05)	Suitable
snake	(0.01, 0.9)	Not Suitable
bear	(0.5, 0.9)	Not Suitable
raccoon	(0.5, 0.5)	Not Suitable
mouse	(0.01, 0.01)	Not Suitable

Table 4-2: A table consisting of the labels assigned to each of the animals being considered for the petting zoo.

If we revisit the plot in Figure 4-1 with the newly assigned labels in the third column of Table 4-2, we can see a pattern. It is possible to *linearly separate*[3] the data points belonging to the two classes using a line.

The line separating the two classes is called a *decision boundary*, and this is only *one* possible decision boundary. Refer to Figure 4-3 where we draw one such decision boundary. If we had more than two features we would need to draw a hyper-plane to separate the points in more than two dimensions.

[3] Linear Separability - https://en.wikipedia.org/wiki/Linear_separability

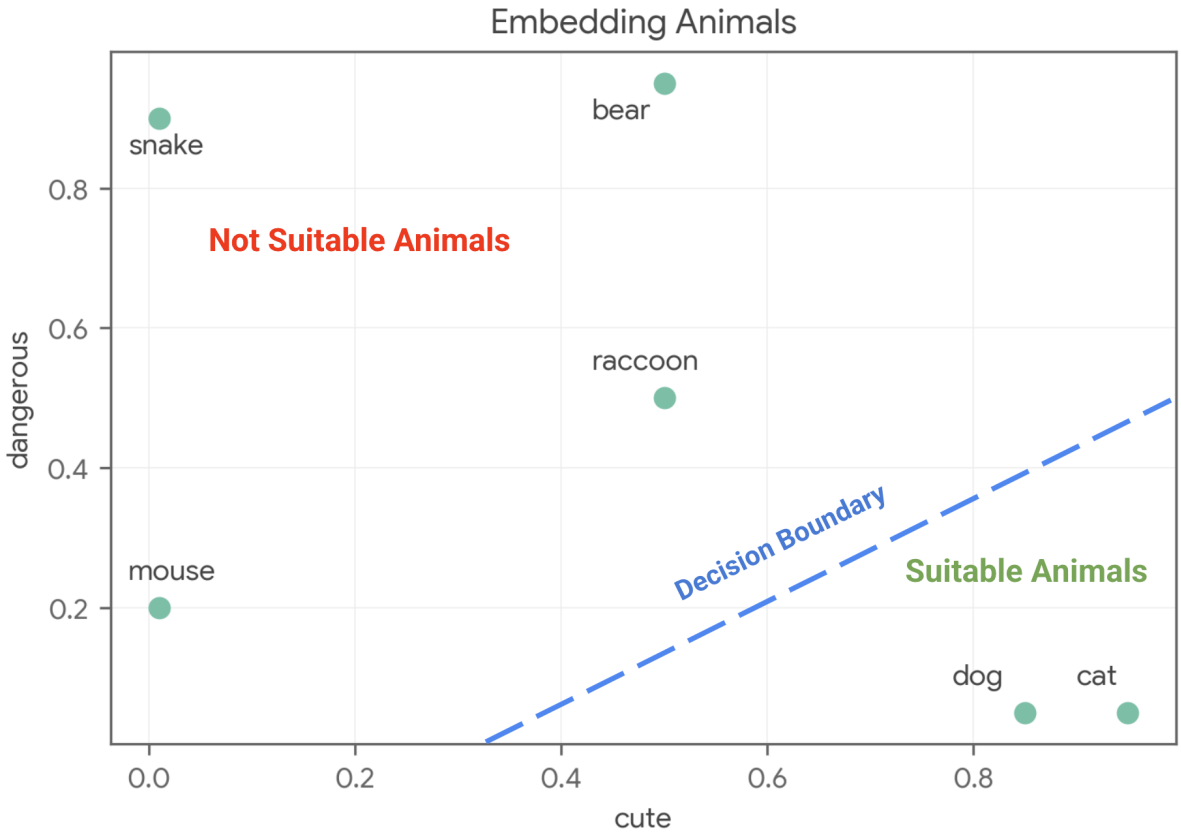


Figure 4-3: Extending figure 4-2, we draw a decision boundary to separate the two classes of animals (suitable and not suitable for the petting zoo).

In this case, we could manually separate the two classes (Suitable / Not Suitable), since there were very few examples. What if you have multiple classes / a large number of examples / more than two features? In those cases, we could use classical machine learning algorithms like the Support Vector Machine[4] (SVM) to learn classifiers that would do this for us. We could rely on deep learning models as well which can learn complex and non-linear decision boundaries.

We can train a deep learning model using the animals' embedding as the input. From the perspective of training the model, it is agnostic to what the embedding is for (a piece of text, audio, image, video, or some abstract concept).

Here is a quick recipe to train embedding-based models:

1. **Embedding Table Generation**: Generate the embeddings for the inputs using machine learning algorithms of your choice.
2. **Embedding Lookup**: Look up the embeddings for the inputs in the embedding table.

[4] Support Vector Machine - https://en.wikipedia.org/wiki/Support-vector_machine

3. **Train the model:** Train the model for the task at hand[5] with the embeddings as input.

Refer to Figure 4-4 that describes the three steps visually.

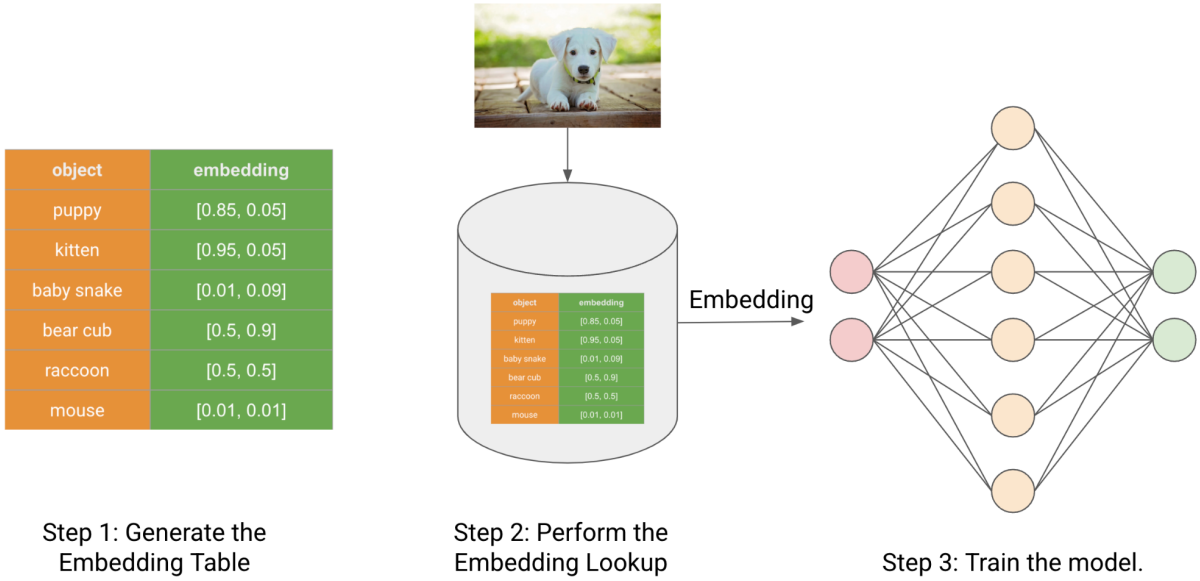


Figure 4-4: A high-level visualization of the embedding-based model training lifecycle. We start with generation of the embedding table, followed by looking up the embeddings for the inputs. Finally, we train a model that takes the embeddings as input.

In our petting zoo example:

- **Embedding Table Generation:** We have generated the embedding table.
- **Embedding Lookup:** For each input example, we will look up the embedding of the corresponding animal in the embedding table.
- **Train the model:** As we saw earlier the points are linearly separable. We can train a model with a single fully connected layer followed by a softmax activation, since it is a binary classification task.

An important caveat is that the model quality naturally depends on the quality of the embedding table. In the petting zoo example, we manually created the embeddings, which is not only slow but also relies on our intuition about both the features and their values. Is there a way to automate the embedding table generation? Turns out there is!

In the next section, let's go over a real world example of embedding table generation by leveraging deep learning to do the grunge work for us!

[5] A quick note that in some cases, we would want our embeddings to be trainable. This would allow us to adapt our embeddings to be more robust to the task at hand. We will demonstrate how to do this in just a bit.

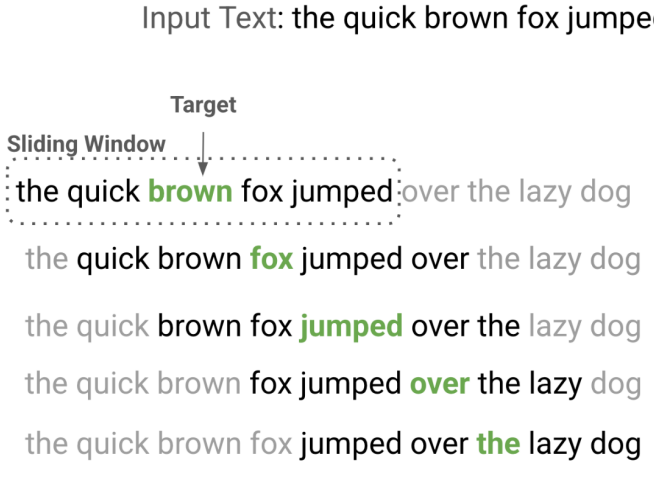
A Real World Example: Word2Vec

In the real world, we must automate the embedding table generation because of the high costs associated with manual embeddings. One example of an automated embedding generation technique is the word2vec family of algorithms[6] (apart from others like GloVe[7]) which can learn embeddings for word tokens for NLP tasks.

The embedding table generation process is done without having any ground-truth labels, which is an example of self-supervised learning using a large dataset like Wikipedia's pages in English. One of the tasks that we can train the model is to predict a hidden word in a sentence, given the words surrounding it (the context). This is known as the *Continuous Bag of Words* (CBOW) task, where the model learns to predict a masked word in a sentence given the surrounding words. An alternate task is to predict the surrounding words for a given word in a sentence. This is known as the *Skipgram* task.

In the CBOW task, taking the sentence "the quick brown fox jumps over the lazy dog", we can mask the word "jumps" and let the neural network predict the word it thinks fits in the sentence based on the surrounding words (context). Mathematically, we want to find a word w_i, which maximizes the conditional probability $P(w_i|w_{i-k}, w_{i-k+1}, , w_{i-1}, w_{i+1}, , w_{i+k})$, where the size of the sliding window of context is $2k+1$ (k words on each side of the masked word).

Let's take an example with $k=2$ such that our window size is $2k+1=5$. In this scenario, our training dataset with the above sentence would look something like in Figure 4-5. Similarly, Figure 4-6 demonstrates the Skipgram task.

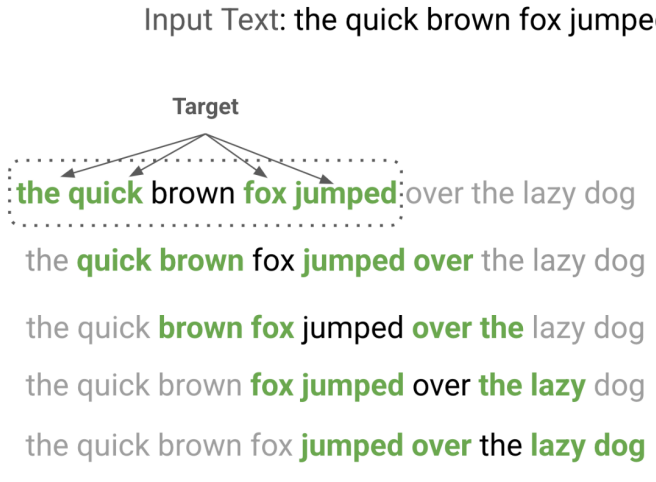


Model Input	Label
the, quick, fox, jumped	**brown**
quick, brown, jumped, over	**fox**
brown, fox, over, the	**jumped**
fox, jumped, the, lazy	**over**
jumped, over, lazy, dog	**the**

Figure 4-5: This figure depicts the sliding window of size 5, the hidden target word, model inputs, and the label for a given sample text in the CBOW task.

---

[6] Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).
[7] GloVe - https://nlp.stanford.edu/projects/glove



Model Input	Label
brown	**the, quick, fox, jumped**
fox	**quick, brown, jumped, over**
jumped	**brown, fox, over, the**
over	**fox, jumped, the, lazy**
the	**jumped, over, lazy, dog**

Figure 4-6: This figure depicts the sliding window of size 5, the hidden target word, model inputs, and the label for a given sample text in the Skipgram task.

Let's get to solving the CBOW task[8] step by step and train an embedding table in the process. We will start with creating a vocabulary of words in the first step. The second step assigns a unique index to the words. This process is called vectorization. An embedding table with a row for each word is initialized in the third step. Finally, in the fourth step, we train a model which trains the embedding table along with it. We use a single hidden layer network[9] with a softmax classification head for this task. The size of the softmax classification head is equal to the vocabulary size because the final output of the model is the word itself. Let's discuss each step in detail.

Step 1: Vocabulary Creation

In this step, we create a vocabulary of the top N words[10] (ordered by frequency) from the given training corpus. We would learn N embeddings of d dimensions each (where we can also view

[8] Solving Skipgram is going to be identical, and is left as an exercise to the reader! We always wanted to write this in our books, after having read this in many textbooks throughout our life. Hopefully, we have given enough information to make this actually straightforward.

[9] Implementation Detail: Using the cross entropy loss when $$N$$ is large can be computationally expensive due to the N-way softmax calculation. In the real world, as an efficient approximation, we use the Negative Sampling technique so that we only look at the output probability of the label class (which should be closer to 1.0), and the output probabilities of a few other random classes (which should be close to 0.0).

[10] We are dealing with word tokens as an example here, hence you would see the mention of words and their embeddings. In practice, we can be working with any kind of input feature. We could be tokenizing on individual characters (unigrams), or pairs of characters (bigrams), or trigrams, or a combination of them along with words, etc. The process of training the embeddings will remain identical across the different features.

d as the number of features that our model learns for each word). Note that one slot in the vocabulary is reserved for the Out of Vocabulary (OOV) token, which is used as a placeholder / catch-all for all the words that are not present in our vocabulary (also referred to as UNK, short for unknown), and one slot is reserved for the padding[11] token.

The choice of N is crucial because it controls the number of unique words for which we learn embeddings. A small value for N would result in loss of information because most of the words would get mapped to the OOV token. However, if N is too large, we would have to pay the cost of a very large embedding table.

Step 2: Dataset Preparation & Vectorization

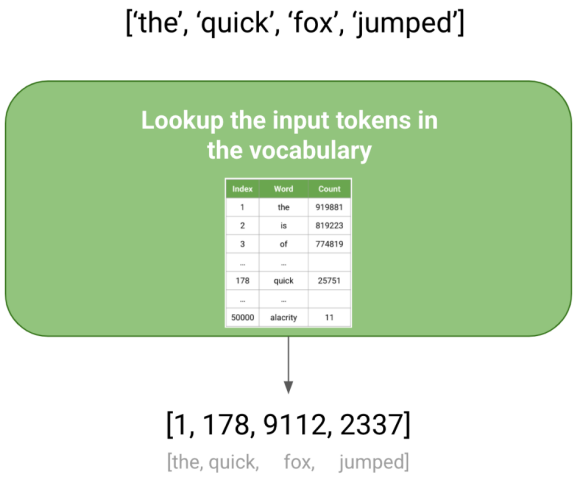
Once the window size ($2k + 1$) is chosen, the dataset is preprocessed (lowercase, strip punctuation, normalization etc.) to create pairs of input context (neighboring words), and the label (masked word to be predicted).

The word tokens are *vectorized* by replacing the actual words by their indices in our vocabulary. If a word doesn't exist in the vocabulary, we map it to the index of the OOV token. Similarly, we replace the label word with the index of that word in the vocabulary.

Refer to Figure 4-7 for visualizing step 1 & 2.

Index	Word	Count
1	the	919881
2	is	819223
3	of	774819
...	...	
178	quick	25751
...	...	
50000	alacrity	11

Step 1: Create the vocabulary.



Step 2: Vectorize the dataset by looking up indices in the vocabulary.

Figure 4-7: Creating the vocabulary and vectorizing the dataset.

[11] Padding tokens are added to input sequences to ensure that all input sequences have the same length.

Step 3: Embedding Table Initialization

Our embedding table E is a floating-point tensor of shape (N, d), where the i-th row is an embedding corresponding to the the i-th word in the vocabulary.

To start off, we initialize this table with small random values using a probability distribution, say the uniform distribution, with a nice range, say [-0.05, 0.05]. Refer to Figure 4-8.

Index	Embedding
1	[0.03, -0.01, 0.02]
2	[-0.01, 0.05, -0.04]
3	[0.0, -0.05, 0.01]
...	...
178	[0.04, -0.03, 0.01]
...	...
50000	[-0.01, -0.02, 0.05]

Step 3: Initialize embedding table with random values in a small range.

Figure 4-8: Initializing the embedding table at the beginning with a uniform probability distribution in a reasonable range ([0.05, 0.05] in this case).

Step 4: Embedding Table Training

Finally, we train the embedding table. Our dataset consists of pairs of the context and the label.

1. We start by looking up the embeddings for each of the words ($w_0, w_1, ..., w_{2k-1}$) in the context from the embedding table E. In this case, we want to make the table trainable, because that's the whole point of this task.
2. These embeddings are then averaged to obtain the averaged representation of all the context words, as follows:

$$e_{avg} = \frac{1}{2k} \sum_{i=0}^{2k-1} E[w_i]$$

3. The model takes e_{avg} as input to produce a probability vector of size N. The index with highest probability indicates the index of the hidden word.

Refer to Figure 4-9 for a visual depiction of the above training method.

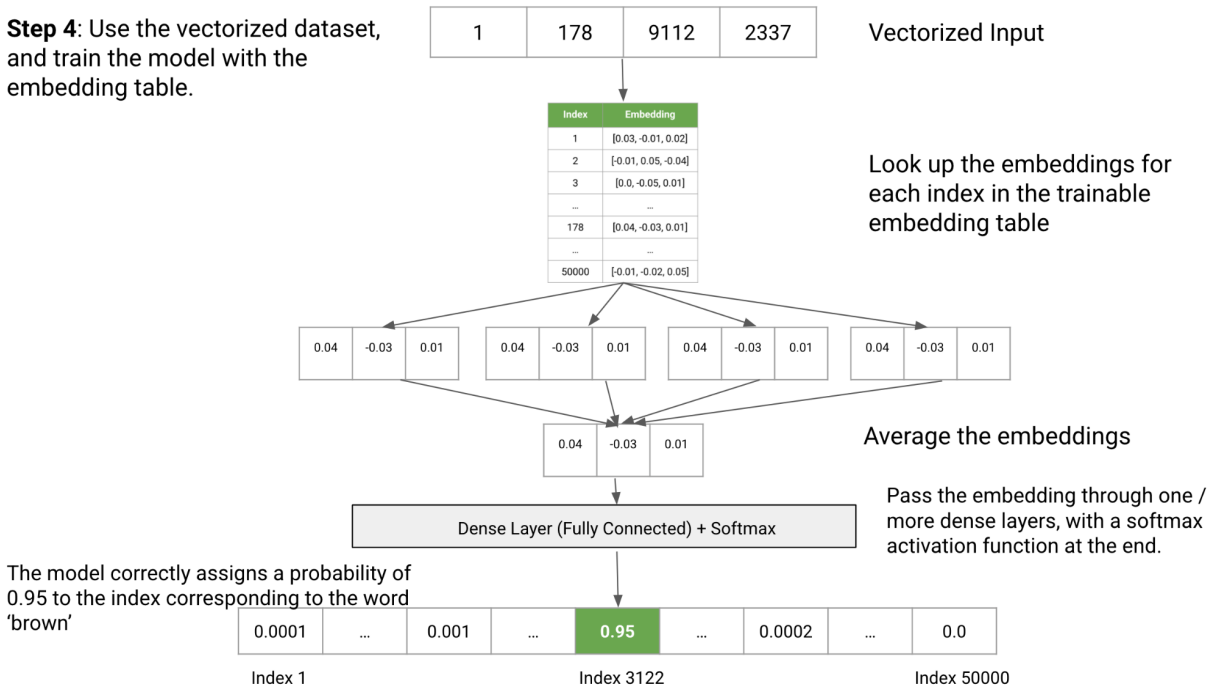


Figure 4-9: A flow depicting the final step of training the embedding table.

In the beginning, the embedding table has been initialized with random values, hence most of the predictions will be guesses. As we train the network, it will start to learn meaningful embeddings for the context words, and it will begin to pick up the co-occurrences between words in the context and the hidden word.

We have simplified the training process for the reader. If you are interested to learn more details, we encourage you to read them here[12].

You must have noticed that, in step 3, we don't use the positional information of the words. The words are all averaged to compute e_{avg}, and we would have got the same result for any other permutation of the words in the context. Hence the name *Bag of Words* for this family of model architectures. In practice, you need not be limited to this architecture for solving the CBOW (or Skipgram) task.

[12] The Illustrated Word2vec - https://jalammar.github.io/illustrated-word2vec/

The nifty [embedding projector tool] visualizes embeddings in three dimensions and enables to see which embeddings lie close to a given input. This can be useful to verify that the embedding table captures semantic relationships between the words as intended. For example, in Figure 4-10, we visually verify that the closest points to 'king' in 2-D are 'kingdom', 'crown', 'archbishop', and so on. These are all very relevant, and it passes the sanity check. You can further play with this tool to visualize the embeddings for different words.

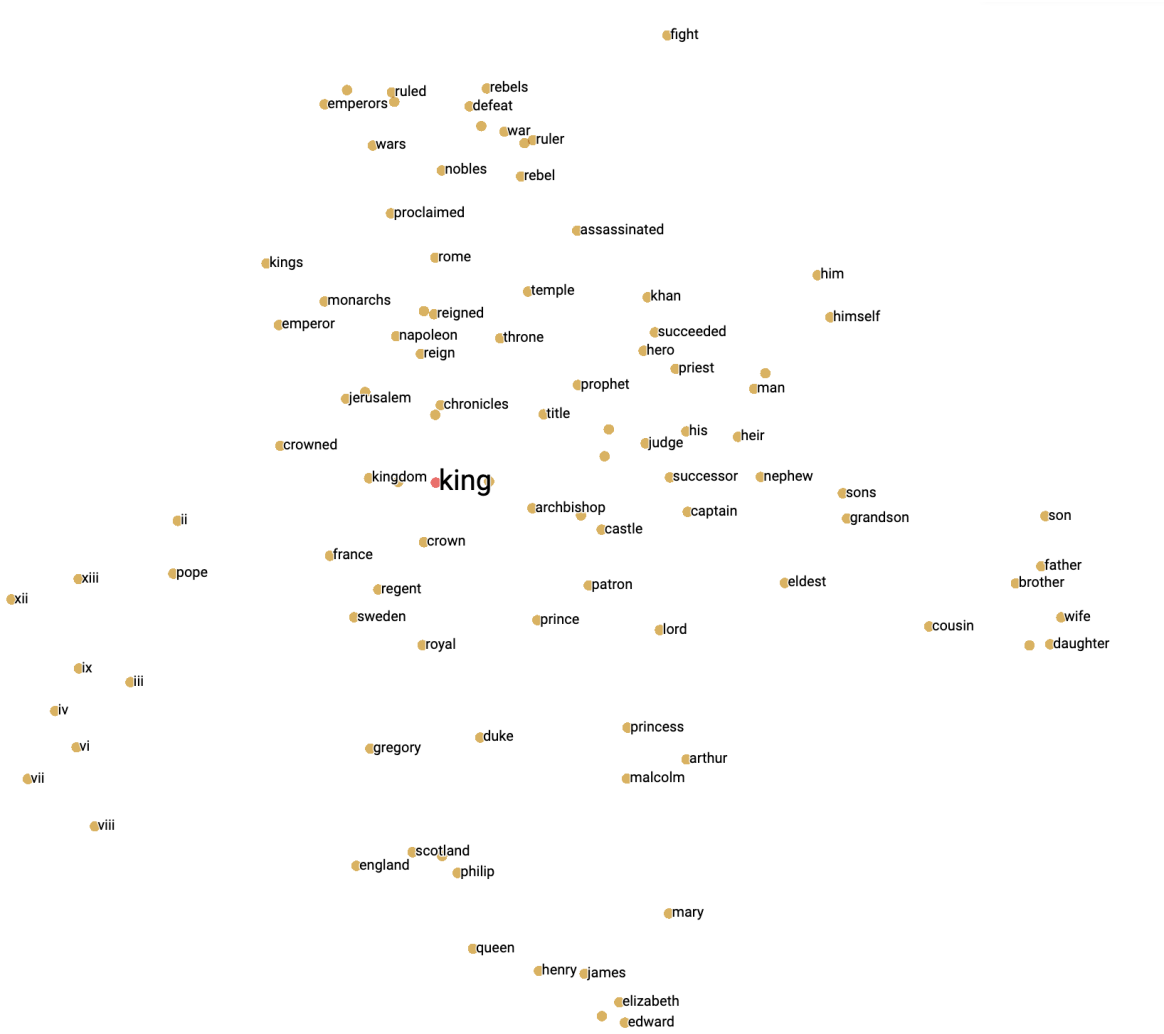


Figure 4-10: Using the embedding projector tool to visualize the word2vec embeddings in 3-D.

Now that we have trained the embeddings, in the next section, let's learn to use them to improve deep learning models.

Embedding Tables: Learn once, use many times!

The most interesting outcome of the above training is the word embeddings table, which can be reused in a new task *downstream*. The new task could be unrelated to the original training[13], similar to the toy example where we predicted if an animal was suitable for the petting zoo using a hand-crafted embedding table.

It is more efficient to learn the embeddings that generalize across many tasks once on a large training corpus than to do it on a per-task basis. This is similar in spirit to transfer learning, where we initialize the model with a checkpoint created earlier by training on a general task. Although transfer learning requires the model to have the exact same architecture as the original task, embeddings are agnostic to the model architecture of the downstream task.

In essence, the embedding tables provide us a portable memory bank of knowledge about our domain of interest. This knowledge can be freely used by downstream tasks, which gives them a boost in quality, and drastically reduces the training data size and time required.

The quality of the embeddings primarily depends on the below two factors:

> **Number of dimensions in each embedding (*d*)**: This is analogous to the features we manually computed in the toy example. Naturally, increasing *d* will increase the quality of the embeddings which might lead to better performance in downstream tasks, but it will also increase the size of the embedding table.
>
> **Size of the vocabulary (*N*):** It is expensive to learn an embedding for every possible token, so we limit our vocabulary to a smaller subset using heuristics like the top *N* most-frequent words in the corpus.

It is often straightforward to scale up or down the model quality by increasing or decreasing these two parameters respectively. The exact sweet-spot of embedding table size and model quality needs to be determined empirically, but *d* is often in hundreds for NLP problems, and N might range from thousands to millions.

Now that we are familiar with generating embeddings, how do we use them? If you have a generic task that has been solved before, it might be worth using its embeddings. However, if you are working on a specific task, you might want to train your embeddings similar to Word2Vec, GloVe, and other embedding methods.

How about we jump into a project now to demonstrate how embeddings can be used to achieve a high performing model while optimizing your training resources? Here we go!

---

[13] The embedding training is referred to as the *pretraining* step in the recent literature.

Using pre-trained embeddings to improve accuracy of a NLP task.

In this project we will work with the DBPedia dataset, which has snippets of text from Wikipedia. Our goal is to classify a given piece of text into one of the fourteen categories. The Jupyter notebook is available **here** for you to play with.

We have already downloaded the dataset in the dbpedia_csv directory, and the structure is as follows.

```
dbpedia_csv/
dbpedia_csv/train.csv
dbpedia_csv/readme.txt
dbpedia_csv/test.csv
dbpedia_csv/classes.txt
```

Let's explore the dataset! First, let's see what classes we have.

```
import os
import pprint

class_names = open(os.path.join('dbpedia_csv', 'classes.txt')).read().splitlines()
num_classes = len(class_names)

# The classes are as follows.
pprint.pprint(class_names)
```

There are fourteen classes as follows.

```
['Company',
 'EducationalInstitution',
 'Artist',
 'Athlete',
 'OfficeHolder',
 'MeanOfTransportation',
 'Building',
 'NaturalPlace',
 'Village',
 'Animal',
 'Plant',
 'Album',
 'Film',
 'WrittenWork']
```

The data is in CSV format with columns: *class-id*, *title* and *description.* The class id is 1-indexed, and the other two fields, title and description, are self-explanatory. Let's take a look at a few random entries from the *train.csv* file.

```
!shuf -n 10 dbpedia_csv/train.csv
```

```
9,"Surnets"," Surnets is a village in the municipality of Tervel in Dobrich Province in northeastern Bulgaria."
11,"Nephrodesmus"," Nephrodesmus is a genus of flowering plants in the legume family Fabaceae. It belongs to the subfamily Faboideae."
9,"Mahmudabad Kharameh"," Mahmudabad (Persian: محمودآباد also Romanized as Maḥmūdābād; also known as Maḥbūdābād-e Pā'īn Mahmood Abad Hoomeh Maḥmūdābād-e Ḩūmeh and Maḥmūdābād-e Pā'īn) is a village in Korbal Rural District in the Central District of Kharameh County Fars Province Iran. At the 2006 census its existence was noted but its population was not reported."
11,"Dichrocephala"," Dichrocephala is a genus of flowering plants in the daisy family Asteraceae."
8,"Carrick River"," The Carrick River is a river of Fiordland close to the southwesternmost point of New Zealand's South Island. Its course is predominantly southward and passes through numerous small lakes (most notably Lake Victor before reaching the sea at the islet Cove arm of Cunaris Sound."
11,"Oxalis stricta"," Oxalis stricta called the common yellow woodsorrel (or simply yellow woodsorrel) common yellow oxalis upright yellow-sorrel lemon clover or more ambiguously and informally sourgrass or pickle plant is an herbaceous plant native to North America parts of Eurasia and has a rare introduction in Britain. It tends to grow in woodlands meadows and in disturbed areas as both a perennial and annual."
6,"Europa Jupiter System Mission - Laplace"," The Europa Jupiter System Mission - Laplace (EJSM/Laplace) was a proposed joint NASA/ESA unmanned space mission slated to launch around 2020 for the in-depth exploration of Jupiter's moons with a focus on Europa Ganymede and Jupiter's magnetosphere."
```

Let's find the number of train and test examples.

```
!wc -l dbpedia_csv/train.csv
!wc -l dbpedia_csv/test.csv
```

```
560000 dbpedia_csv/train.csv
70000 dbpedia_csv/test.csv
```

It all looks good! Now, it's time to put our theory into practice. Even though we are going to use pre-trained embeddings, we will roughly follow steps similar to Word2Vec training. However, there would be some differences. For instance, here, we don't need to train embeddings from scratch. Let's review those four steps, and see how they apply in our case here.

Step 1: Vocabulary Creation

In this step, we will use a TextVectorization layer from Tensorflow to create a vocabulary of the most relevant words. It finds the top *N* words in a dataset, sorts them in the order of their frequencies, and assigns them an index. This process of mapping free form inputs to integer sequences is known as vectorization, as introduced in the Word2Vec subsection.

The TextVectorization layer takes the vocabulary size as an input. For this task, we choose the vocabulary size to be 5000 words. Why did we choose 5000, and not 500,000 words? We can, but as mentioned earlier this would mean we have a bigger vocabulary and a bigger embedding table. Additionally at some point, increasing N would give miniscule improvements in accuracy. Hence, this is a trade-off.

We also ensure that the tokenized input results in an integer sequence with exactly 250 tokens. This might mean padding short texts with padding tokens and truncating the longer ones to 250 tokens.

```
import tensorflow as tf

# Size of our vocabulary.
vocab_size = 5000

# This controls the max number of tokens the layer with tokenize, by truncating
# the rest of the sequence.
max_seq_len = 100

vectorization_layer = tf.keras.layers.TextVectorization(
    max_tokens=vocab_size,
    output_sequence_length=max_seq_len)
```

Once we have initialized the layer, we can invoke the *adapt()* method with the dataset to use as a source for building the vocabulary.

```
# This step allows the vectorization layer to build the vocabulary.
train_text_ds = tf.data.Dataset.from_tensor_slices(x_train).batch(512)
vectorization_layer.adapt(train_text_ds)
```

Let's checkout the top *ten* words in the vocabulary using the *vectorization_layer.get_vocabulary()* method as follows

```
vocabulary = vectorization_layer.get_vocabulary()
vocabulary[:10]
```

```
['', '[UNK]', 'the', 'in', 'of', 'is', 'a', 'and', 'was', 'by']
```

Notice that the first two elements are an empty string (a reserved token for padding) and a 'UNK' token (a token reserved for words which are not found in the vocabulary). Following these, we see the usual stop words like 'the', 'in', 'of', and so on.

Step 2: Dataset Preparation and Vectorization

The tensorflow *vectorization_layer* is a regular layer which can be invoked with a model as well as independently. The output of this layer is a vector with indices of tokens in the vocabulary.

Our string of interest is 'efficient deep learning', but we will add a random token at the end that we are confident will not be in the vocabulary.

```
edl_sequence_output = vectorization_layer(
    [['efficient deep learning x123!']]).numpy()[0, :4]
edl_sequence_output
```

```
array([   1, 1379, 1585,    1])
```

The code snippet above returns indices into the vocabulary we created in the previous step. Let's look up the indices in the vocabulary to make sure that it works well.

```
' '.join(np.take(vocabulary, edl_sequence_output))
```

```
'[UNK] deep learning [UNK]'
```

Notice that the random token (x123!) and the word ‘efficient’ are both transformed to the unknown token ([UNK]) because they aren’t available in our vocabulary. The random token is missing because it wasn’t present in the vectorization data while the word 'efficient' is missing because it wasn’t among the top 5K most frequent words in the vectorization data. It means that the vectorization process prioritizes the frequent words over the rare ones to reduce overfitting.

We can now vectorize the train and test datasets.

```
x_train_vectorized = vectorization_layer(x_train)
x_test_vectorized = vectorization_layer(x_test)
```

Step 3: Initialization of the Embedding Matrix

In this step, we will create an embedding matrix using the pre-trained Word2Vec embeddings from TFHub[14]. These embeddings use a 250 dimensional vector to represent a token in the vocabulary. The below code snippet transforms the vocab tokens to their corresponding embeddings.

```
import tensorflow_hub as tfhub

word2vec_hub_layer = tfhub.KerasLayer(
    'https://tfhub.dev/google/Wiki-words-250/2')
word2vec_embeddings = word2vec_hub_layer(vocabulary)
```

The shape of the `word2vec_embeddings` tensor should be (vocab_size, embedding_dim), i.e., (5000, 250). The following code snippet will verify that.

```
embedding_dim = 250

# The shape of the word2vec_embeddings would be (vocabulary_size, 250),
# since we are embedding each row in the vocabulary and the size of the
# word2vec embeddings is 250 dimensions.
print('Vocabulary Size:', len(vocabulary))
print('word2vec_embeddings.shape:', word2vec_embeddings.shape)

# Verify that this is true.
tf.assert_equal(word2vec_embeddings.shape, (len(vocabulary), embedding_dim))
```

Indeed, that is the case. It all looks good!

---

[14] TFHub (https://tfhub.dev/) is a collection of pre-trained checkpoints of models and layers that you can directly use in your model. There are a large number of popular models across image, text, audio, and video domains that are ready-to-deploy. For instance, you should not spend resources and time training your own ResNet model. Instead, you can directly get the model architecture and weights from TFHub, and fine-tune it to your problem.

```
Vocabulary Size: 5000
word2vec_embeddings.shape: (5000, 250)
```

Let's create a function *get_pretrained_embedding_layer()* to create an embedding layer initialized with the *word2vec_embeddings* tensor created previously. The job of the embedding layer, given a list of token indices, is to look up their embeddings in the *word2vec_embeddings* tensor. Note that our vectorization layer ensures that every input string is transformed into a sequence of integer ids. The maximum sequence length is 100. Therefore, for every input string, the embedding layer would receive 100 integer ids, and it would return a tensor of shape (100, 250) containing 100 embeddings of size 250 dimensions each.

We use Tensorflow's Embedding layer for this purpose. It allows us to configure the vocabulary size, embedding dimension size, the initializing tensor for the embeddings and several other parameters. It crucially also supports fine-tuning the table to the task by setting the layer as trainable. However, in our case, we have initialized it to the word2vec embeddings which might be good enough for generalization. Hence, we are keeping it untrainable.

```
def get_pretrained_embedding_layer(
    trainable=False,
    embedding_dim_size=embedding_dim,
    embedding_tensor=word2vec_embeddings):
  return tf.keras.layers.Embedding(
      vocab_size,
      embedding_dim_size,
      embeddings_initializer=tf.keras.initializers.Constant(
          word2vec_embeddings),
      trainable=trainable,
  )
```

Let's also create an alternate layer with a randomly initialized embedding table. We keep this layer trainable to learn the embedding table as we train the model. Towards the end, we will compare the qualities of the models trained with these two styles of embedding layers.

```
def get_untrained_embedding_layer(trainable=True, embedding_dim_size=250):
  return tf.keras.layers.Embedding(
      vocab_size,
      embedding_dim_size,
      trainable=trainable,
  )
```

Step 4: Training the models.
Now the fun part! We will create a simple model that uses these pre-trained embeddings. As a first candidate, let's try the humble Bag-of-Words (BOW) model which we saw when discussing the Word2Vec training.

In this setup, the model takes a sequence of word token ids generated by the vectorization layer as input and transforms the individual token ids to their representative vectors in the embedding table. Next, the model averages all the embeddings in the input sequence to reduce each input

to a single vector. The result is passed through a few dense layers and a softmax activation to generate an output tensor of size num_classes.

This is similar to the Word2Vec example except the fact that there is no masking involved. Here, our task is to classify the input by estimating its probability of belonging to each of the 14 classes.

The below function, *get_bow_model()*, constructs our model with the *embedding_layer* provided as an argument.

```
def get_bow_model(embedding_layer):
  int_sequences_input = tf.keras.Input(shape=(None,), dtype='int64')
  embedded_sequences = embedding_layer(int_sequences_input)

  x = tf.reduce_mean(embedded_sequences, axis=1)
  x = tf.keras.layers.Dense(512, activation='relu')(x)
  x = tf.keras.layers.Dense(128, activation='relu')(x)
  x = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

  output = x
  model = tf.keras.Model(int_sequences_input, output, name='bow')
  model.summary()
  return model

# Train a model with pre-trained Word2Vec embeddings.
bow_model_w2v = get_bow_model(get_pretrained_embedding_layer(trainable=True))
bow_model_w2v.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=["accuracy"]
)
bow_model_w2v.summary()
```

```
Model: "bow"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_12 (InputLayer)       [(None, None)]            0

 embedding_10 (Embedding)    (None, None, 250)         1250000

 tf.math.reduce_mean_8 (TFOp  (None, 250)              0
 Lambda)

 dense_28 (Dense)            (None, 512)               128512

 dense_29 (Dense)            (None, 128)               65664

 dense_30 (Dense)            (None, 14)                1806

=================================================================
Total params: 1,445,982
Trainable params: 1,445,982
Non-trainable params: 0
_________________________________________________________________
```

Let's train this model!

```
bow_model_w2v_history = bow_model_w2v.fit(
    x_train_vectorized, y_train, batch_size=64, epochs=10,
    validation_data=(x_test_vectorized, y_test))
```

```
Epoch 1/10
313/313 [==============================] - 5s 13ms/step - loss: 0.8423 - accuracy: 0.7685 -
val_loss: 0.2341 - val_accuracy: 0.9361
Epoch 2/10
313/313 [==============================] - 3s 11ms/step - loss: 0.1382 - accuracy: 0.9641 -
val_loss: 0.1766 - val_accuracy: 0.9492
Epoch 3/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0662 - accuracy: 0.9823 -
val_loss: 0.1757 - val_accuracy: 0.9516
Epoch 4/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0343 - accuracy: 0.9911 -
val_loss: 0.2061 - val_accuracy: 0.9449
Epoch 5/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0211 - accuracy: 0.9944 -
val_loss: 0.1833 - val_accuracy: 0.9561
Epoch 6/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0120 - accuracy: 0.9969 -
val_loss: 0.1901 - val_accuracy: 0.9578
Epoch 7/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0104 - accuracy: 0.9974 -
val_loss: 0.2101 - val_accuracy: 0.9546
Epoch 8/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0083 - accuracy: 0.9976 -
val_loss: 0.2321 - val_accuracy: 0.9523
Epoch 9/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0083 - accuracy: 0.9975 -
val_loss: 0.2259 - val_accuracy: 0.9567
Epoch 10/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0060 - accuracy: 0.9983 -
val_loss: 0.2719 - val_accuracy: 0.9509
```

We achieved a first epoch accuracy of 93.61%, and a top accuracy of 95.78%. This is quite impressive!

How about we train a model without the pre-trained word2vec embeddings and see if it does better or worse? This should be easy to do with our helper methods.

```
# Train a model without pre-trained embeddings.
bow_model_no_w2v = get_bow_model(get_untrained_embedding_layer())
bow_model_no_w2v.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=["accuracy"]
)
bow_model_no_w2v.summary()
```

```
Model: "bow"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_13 (InputLayer)       [(None, None)]            0

 embedding_11 (Embedding)    (None, None, 250)         1250000

 tf.math.reduce_mean_9 (TFOp  (None, 250)              0
 Lambda)

 dense_31 (Dense)            (None, 512)               128512

 dense_32 (Dense)            (None, 128)               65664

 dense_33 (Dense)            (None, 14)                1806

=================================================================
Total params: 1,445,982
Trainable params: 1,445,982
Non-trainable params: 0
_________________________________________________________________
```

Let's train this model too, and observe its progress.

```
bow_model_no_w2v_history = bow_model_no_w2v.fit(
    x_train_vectorized, y_train, batch_size=64, epochs=10,
    validation_data=(x_test_vectorized, y_test))
```

```
Epoch 1/10
313/313 [==============================] - 4s 11ms/step - loss: 1.1667 - accuracy: 0.6370 -
val_loss: 0.2971 - val_accuracy: 0.9190
Epoch 2/10
313/313 [==============================] - 3s 11ms/step - loss: 0.1758 - accuracy: 0.9517 -
val_loss: 0.2038 - val_accuracy: 0.9437
Epoch 3/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0799 - accuracy: 0.9791 -
val_loss: 0.1877 - val_accuracy: 0.9494
Epoch 4/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0413 - accuracy: 0.9892 -
val_loss: 0.2038 - val_accuracy: 0.9479
Epoch 5/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0244 - accuracy: 0.9943 -
val_loss: 0.2270 - val_accuracy: 0.9454
Epoch 6/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0171 - accuracy: 0.9953 -
val_loss: 0.2302 - val_accuracy: 0.9494
Epoch 7/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0118 - accuracy: 0.9972 -
val_loss: 0.2482 - val_accuracy: 0.9479
Epoch 8/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0106 - accuracy: 0.9973 -
val_loss: 0.2758 - val_accuracy: 0.9448
Epoch 9/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0091 - accuracy: 0.9977 -
val_loss: 0.3000 - val_accuracy: 0.9441
Epoch 10/10
313/313 [==============================] - 3s 11ms/step - loss: 0.0071 - accuracy: 0.9979 -
val_loss: 0.2972 - val_accuracy: 0.9479
```

As we see in figure 4-11, the first epoch accuracy for this model is 91.90% which is lower than the first epoch accuracy of 93.60% for the BOW model with pre-trained embeddings. The top accuracy for this model is 94.94% which is also marginally lower than the top accuracy 95.78% of pre-trained embeddings model. The figure shows that the model with pre-trained embeddings performed better in all but one epoch.

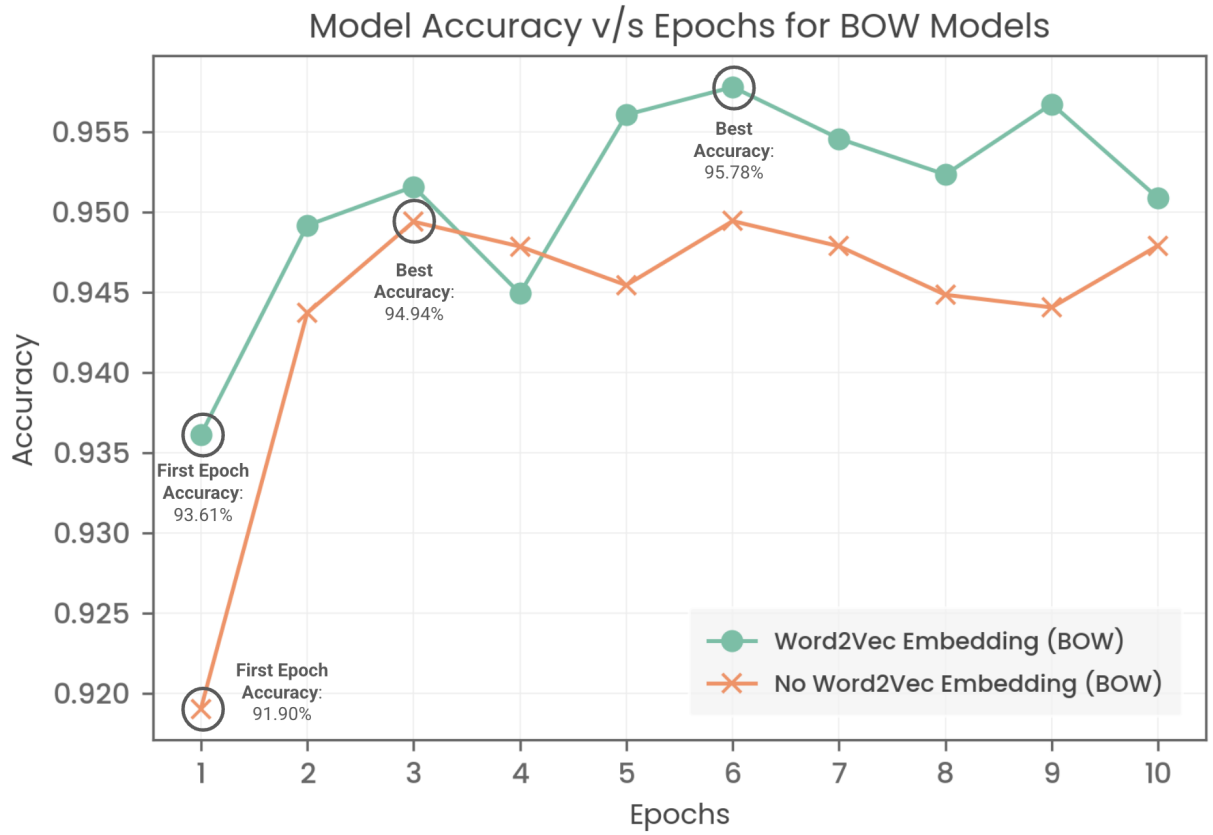


Figure 4-11: Accuracy of the BOW models as they are trained over 10 epochs. The only difference between the two curves is the embedding table initialization. It is apparent that when initializing with pre-trained embeddings, the BOW model reaches a higher accuracy.

Let's create a new model to capture more information from the sequence embeddings instead of just averaging them as we did in the BOW model.

While the BOW models are cheap to train and deploy, they ignore the sequential relationships between the word tokens. Such models treat the sequences with the same words in a different order in an identical manner. But, in reality, they might have a completely different meaning. For instance, ‘it was not bad – very good in fact’, and ‘it was not good – very bad in fact’, indicate opposite sentiments.

We solve this problem for image datasets by using convolutional layers, where we convolve the filters over sliding windows of the input in two dimensions. In this case, we can convolve 1D convolutional filters over sliding windows of the input in one dimension, since the input is a one dimensional sequence of integers.

One possible implementation is as presented below. We use pooling layers to reduce the dimensionality via max-pooling, and to average out the entire sequence dimension via global average pooling.

```
def get_cnn_model(embedding_layer):
  int_sequences_input = tf.keras.Input(shape=(None,), dtype="int64")
  embedded_sequences = embedding_layer(int_sequences_input)

  # Run a convolutional layer on top of the sequences, and use pooling to
  # reduce the sequence length.
  x = layers.Conv1D(128, 5, activation="relu")(embedded_sequences)
  x = layers.MaxPooling1D(3)(x)
  x = layers.Conv1D(128, 5, activation="relu")(x)
  x = layers.MaxPooling1D(3)(x)
  x = layers.Conv1D(128, 5, activation="relu")(x)

  # Global average pooling averages out the entire sequence.
  x = layers.GlobalMaxPooling1D()(x)
  x = layers.Dense(128, activation="relu")(x)
  x = layers.Dropout(0.5)(x)
  preds = layers.Dense(14, activation="softmax")(x)
  model = tf.keras.Model(int_sequences_input, preds)

  model.summary()
  return model
```

We can train this model with and without the pre-trained embeddings similar to the BOW model.

The training with the pre-trained embeddings is as follows:

```
# Training with pre-trained Word2Vec embeddings.
cnn_model_w2v = get_cnn_model(get_pretrained_embedding_layer(trainable=True))
cnn_model_w2v.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=["accuracy"]
)

cnn_model_w2v_history = cnn_model_w2v.fit(
    x_train_vectorized, y_train, batch_size=128, epochs=10,
    validation_data=(x_test_vectorized, y_test))
```

The model training without pre-trained embeddings follows right after:

```
# Training without pre-trained embeddings.
cnn_model_no_w2v = get_cnn_model(get_untrained_embedding_layer())
cnn_model_no_w2v.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=["accuracy"]
)

cnn_model_no_w2v_history = cnn_model_no_w2v.fit(
    x_train_vectorized, y_train, batch_size=128, epochs=10,
    validation_data=(x_test_vectorized, y_test))
```

In both the cases, the model architecture is identical as shown below:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, None)]            0

 embedding_2 (Embedding)     (None, None, 250)         1250000

 conv1d (Conv1D)             (None, None, 128)         160128

 max_pooling1d (MaxPooling1D  (None, None, 128)        0
 )

 conv1d_1 (Conv1D)           (None, None, 128)         82048

 max_pooling1d_1 (MaxPooling  (None, None, 128)        0
 1D)

 conv1d_2 (Conv1D)           (None, None, 128)         82048

 global_max_pooling1d (Globa  (None, 128)              0
 lMaxPooling1D)

 dense_6 (Dense)             (None, 128)               16512

 dropout (Dropout)           (None, 128)               0

 dense_7 (Dense)             (None, 14)                1806

=================================================================
Total params: 1,592,542
Trainable params: 1,592,542
Non-trainable params: 0
```

We will spare you the training logs here, but you are welcome to inspect them in the notebook directly. Figure 4-12 shows that the CNN models perform better than BOW as they benefit from the word order in the sequences. Moreover, pre-trained embeddings achieve a superior accuracy of 96.42% than the randomly initialized trainable embeddings.

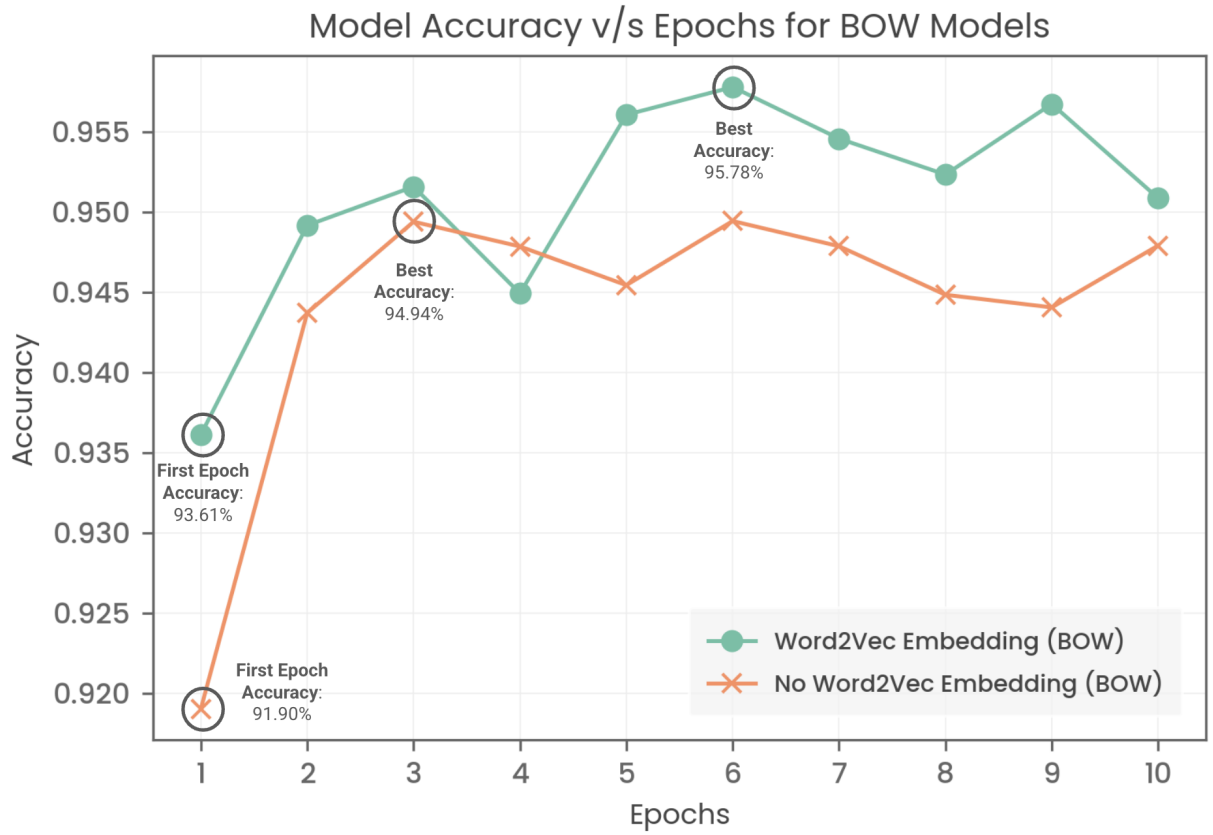


Figure 4-12: Accuracy of the CNN models (with and without pre-trained word2vec embeddings) as they are trained over 10 epochs. The gap between the two approaches is even wider in the case of CNN models, with the pre-trained embedding model achieving a top accuracy of 96.42% v/s a top accuracy of 95.51% when not using pre-trained embeddings.

Step 5: Exporting an end-to-end model.
Ideally, we would prefer the model to accept string inputs (instead of a sequence of integers) and have it perform the vectorization as well. It is possible to achieve by inserting the vectorization layer before the embedding layer.

```
string_input = tf.keras.Input(shape=(1,), dtype='string')
x = vectorization_layer(string_input)
predictions = cnn_model_w2v(x)
end_to_end_model = tf.keras.Model(string_input, predictions)
```

We can now classify a new piece of text, and map it to a class name.

```
probabilities = end_to_end_model.predict(
    [['Usain Bolt is a very well known sprinter and Olympic medal winner.']]
)
class_names[np.argmax(probabilities[0])]
```

```
'Athlete'
```

It works! It's been quite a journey, let's pause and ponder over what we learnt.

Some more thoughts and optimizations related to Embeddings

So far we learnt about embeddings, how pre-trained embeddings are useful, and gave some recipes for training and using them to train classification models. They also help improve training time by achieving convergence early, so you need to train your models for fewer epochs. However, we should discuss a couple of follow-up topics around how to scale them to NLP applications and beyond.

My embedding table is huge! Help me!

While embedding tables help in dimensionality reduction by capturing relationships between different possible inputs, they often occupy a significant portion of the model size on disk and in memory. Although this comes with the cost of the table taking up significant disk space and memory, this issue can be a bottleneck if the model is going to be deployed on-device (smartphones, IoT devices, etc.), where transmitting the model to the device is limited by the user's bandwidth, and the memory available might be limited too.

Let's see what our options are:

1. **The embedding table is too large on-disk**: We can use a smaller vocabulary, and see if the resulting quality is within the acceptable parameters. For on-device models, TFLite offers post-training quantization as described in chapter 2. We could also incorporate compression techniques such as sparsity, k-means clustering, etc. which will be discussed in the later chapters.

2. **Even after compression, the vocabulary itself is large**: Large vocabularies have a tangible footprint by themselves, which excludes the actual embeddings. They are persisted with the model to help with input vectorization. For example, assuming a vocabulary size of 10,000 and an average token of length 6 unicode characters, with each character being 2 bytes, the vocabulary would take up (10,000 * 6 * 2 = 120,000) ~ 120 KB. For a small model, such a vocabulary would have substantial contributions to its total size. An option to consider here is the feature hashing or the hashing trick. It helps to reduce the vocabulary with little or no performance trade-off.

   The core idea of the hashing trick is as follows:
   1. Choose the desired vocabulary size N, and the number of dimensions d.
   2. For each token (or a feature computed on the token) f:
      a. Compute $idx = h(f) \% N$, where h is a hash function.
      b. This maps f to the index idx in the embedding table.

3. We use the embedding at index *idx*, when referring to *f* during training or inference. This step is completely independent of the vocabulary! We just need to make sure that we compute *h(f)* in an identical manner during training and inference.

Tensorflow provides the Hashing layer to conveniently apply the hashing trick. Figure 4-13 shows the hashing trick mechanism. On the left is the list of tokens. The tokens are hashed using the hash function in the center column. The output of the hash function % vocab_size is an index in [0, vocab_size - 1] and is used to refer to the elements in the embedding table.

In the figure, 'bar' and 'hello' map to the same slot in the embedding table and will learn the same embedding.

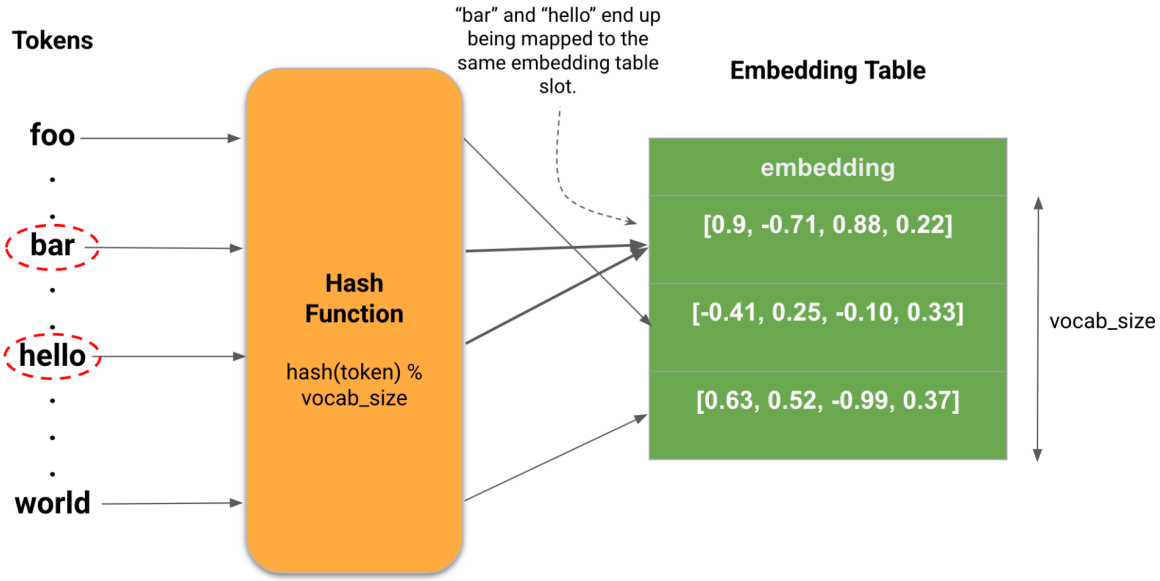


Figure 4-13: Hashing trick as an option for reducing the vocabulary size. In the figure, 'bar' and 'hello' map to the same embedding slot, but collisions are expected, since our embedding table's vocabulary size is smaller than our actual vocabulary size.

Note that since we are computing the index modulo N, there will be collisions if N is smaller than your previous vocabulary size ($N_{original}$). For example, if originally your embedding table had $N_{original}$ = 10,000 entries, and you decide that you can only deploy an embedding table of size N = 1,000, the expected number of tokens mapping to the same slot in the embedding table will be 10,000 / 1,000 = 10. The actual number of tokens mapping to each slot might be slightly higher or lower than this number.

Even though the multiple tokens mapping to each slot might be very different from each other, the model will learn one embedding per embedding table slot. Which implies that the tokens mapping to the same slot will share the same embedding. This might seem

confusing because we are learning the same representation for unrelated tokens, which is bound to cause accuracy degradation. However, in practice we have seen that the degradation is dependent on the task, and can often be limited by choosing a reasonable value of N.

Another idea for improving the performance of the hashing trick is to create multiple features per token, and repeat this process for each feature type. For example, instead of using words, we can use ngrams of different lengths (bi-grams, tri-grams, quad-grams, etc.), and maintain one embedding table per n-gram length.

3. **Even after all the above optimizations, the embedding tables are still prohibitively expensive:** While embedding tables offer significant benefits in terms of quality, they still take up 47-71% of the number of parameters of large NLP models[15]. In this situation, embedding-free approaches like pQRNN[16] are a viable alternative. pQRNN uses the projection operation which maps a given input token to a B-bit fingerprint using a hash function, and this B-bit fingerprint is transformed into a d-dimensional representation using a learned linear transformation (essentially a trainable dense layer of size d, without the activation function). Thus, the only part that needs to be saved on disk / in memory is a matrix of shape (B, d), instead of a vocabulary of size N, and an embedding table of shape (N, d). pQRNN demonstrated a model 140x smaller than an LSTM with pre-trained embeddings. An on-device friendly implementation of pQRNN is available in the Tensorflow repository here.

We learnt about NLP, but what about embeddings for other domains?

For NLP problems, we demonstrated a reproducible path to pre-train embeddings on a word, character, unicode etc. level tokenization and store them in an embedding table for easy lookup and use later. For many other domains, we can continue to follow similar recipes.

For example, if you were to embed all the actors on IMDb, you might consider a pre-training task of predicting the actor, given a fixed number of the actor's other cast members in each movie. As a result of this step, actors working together would get embedded closer to each other. For example, Robert Downey Jr. and Chris Evans would get embeddings with a small distance from each other because they have acted in multiple Marvel movies together, as compared to an actor such as Audrey Hepburn with whom they did not overlap.

For other domains like images, it is not practical to store all possible inputs, or even the top K frequent inputs. There isn't a vocabulary as such to begin with, since images with a single pixel of difference will look like a new image (but are semantically equivalent).

---

[15] Chung, H. W., Fevry, T., Tsai, H., Johnson, M., & Ruder, S. (2020). Rethinking embedding coupling in pre-trained language models. *arXiv preprint arXiv:2010.12821*.
[16] Kaliamoorthi, P., Siddhant, A., Li, E., & Johnson, M. (2021). Distilling Large Language Models into Tiny and Effective Students using pQRNN. *arXiv preprint arXiv:2101.08890*.

A common solution for visual domains is to use a model like ResNet pre-trained on a generic dataset like ImageNet, with its weights frozen and being used as a feature extractor. The image is passed to the model and one of the last few layers' output (typically the penultimate layer) is used as the embedding (referred to as the extracted features, but essentially it is the same thing). Here is a document describing the use of TFHub models for this purpose. You only need to train a small classification head[17] that works on the embeddings directly, similar to what we did in the petting zoo example.

Another alternative is to train a model to specifically learn representations by showing similar and dissimilar inputs, and forcing the model to generate embeddings that have a small distance between each other for similar inputs, and a large distance between embeddings of dissimilar inputs. This method is known as *contrastive learning*. In the SimCLR papers[18] authors use this approach by using data augmentation to create similar inputs without needing labels for the images. Once the model generating the representation is trained, it can then be frozen and a new classification head can be trained as usual.

We hope that this section on embeddings gave you a flavor of what we mean by efficient architectures, and how they help us outperform baseline methods. Another example in this domain is the attention mechanism, which forms the backbone of the state of the art NLP model architectures such as the Transformer, which is now showing great promise in computer vision applications as well!

Learn Long-Term Dependencies Using Attention

Imagine yourself in your favorite buffet restaurant. A variety of food items from savory to sweet are lined up before you on the buffet table. Your eyes light up as soon as you lay them on your favorite *chocolate cake*. The dessert is chosen! You notice your favorite *chicken curry* and decide to pick it up as well. This happens before you even queue up. Out of the several available choices, you paid attention to chocolate cake the most and the chicken curry the second most and so on. In the context of textual data, the words have a certain *affinity* to each other. They are also *attracted* to the subject matter. We term this *affinity* or *attraction* as attention. Words like *run*, *kick* or *throw* have higher affinity to sports than finance. On the other hand, words like *bank* or *institution* dominate the financial literature more than sports columns. The *attention mechanism* leverages this relationship between the words to aid the learning process.

Consider the problem of classifying a news article into one of the following categories: Sports, Business, Sci/Tech and World. Given an article titled "*Messi scored the winning goal in the final moments of the game*", we can instantly classify it as a sports article. We recognize that the words scored, winning, goal and game occur together in sports category with high probability.

[17] A head is a trainable sub-network that takes in the output of the base model (in this case the frozen model that generates the embedding), and returns a new output.

[18] Chen, T., Kornblith, S., Swersky, K., Norouzi, M., & Hinton, G. E. (2020). Big self-supervised models are strong semi-supervised learners. *Advances in neural information processing systems*, *33*, 22243-22255.

The attention mechanism works in a similar fashion. It learns the affinities between the words and their respective positions. Before we dive into the attention mechanism, let's understand its predecessor, Recurrent Neural Network or RNN which, unlike attention, doesn't have the flexibility to look at the entire text sequence.

A RNN contains a recurrent cell which operates on an input sequence $x = x_1, x_2, x_3, \ldots, x_n$ at time step t to output a hidden state h_t such that

$$h_t = f(h_{t-1}, x_t)$$

The output at the time step t_n is h_n which represents the entire sequence. This architecture has been used to solve various tasks where the input data can be represented as sequences. In the news article classification problem mentioned earlier, each news article can be represented as a sequence of words. Hence, an RNN with a *softmax classifier* stacked on top is a good choice to solve this problem.

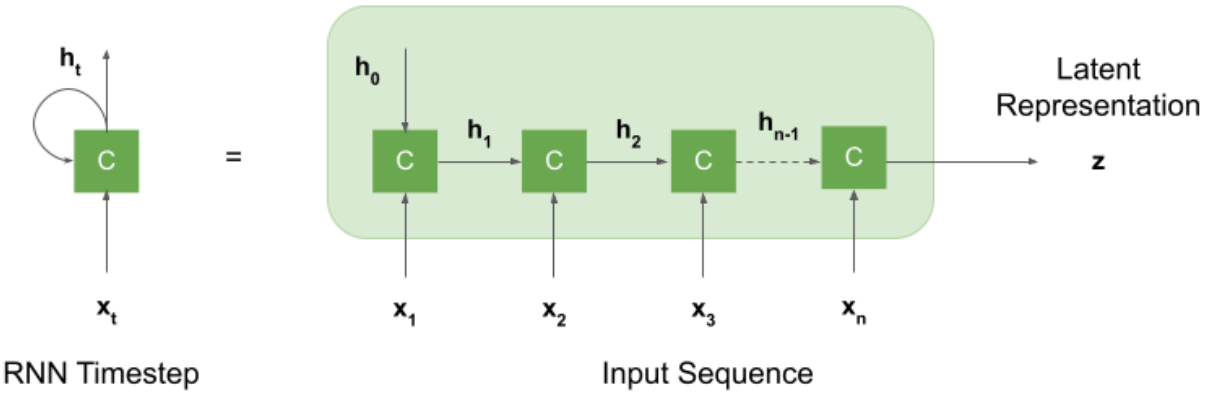


Figure 4-14: A pictorial representation of a Recurrent Neural Network. The image on the left shows a recurrent cell processing the input sequence element at time step *t*. The image on the right explains the processing of the entire input sequence across *n* time steps.

RNNs are also used for sequence to sequence applications like machine translation, where both the input and output are sequences. Consider the task of training a model to translate English language text to Spanish language. Input and outputs of such a model are English language token sequences and Spanish language token sequences respectively. This problem requires two RNN networks namely: an encoder network and a decoder network as shown in figure 4-15. The encoder RNN transforms the english sequence x to a latent representation z. The decoder component receives z and outputs spanish language sequence y.

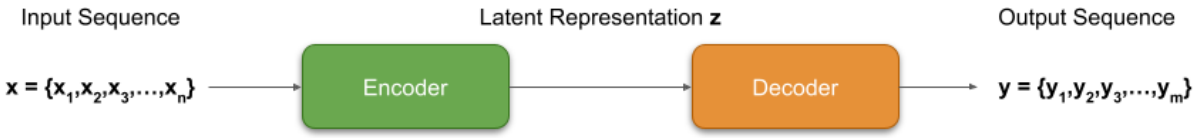


Figure 4-15: RNN Encoder-Decoder

This basic idea of RNN has been refined over the years to solve the inherent architectural problems like exploding gradients and vanishing gradients. This led to the rise of novel cell designs like Long and Short Term Memory[19] (LSTM) and Gated Recurrent Unit[20] (GRU) cells. However, RNNs are slow to train because of their sequential design such that the current timestamp execution depends on the results of previous timestep. Another drawback of a sequential architecture is the loss of context between the elements that are far apart in the sequence. In other words, a sequential architecture has inherent limitations with learning long term dependencies.

Attention, on the other hand, evaluates entire sequences at once. It computes a rectangular attention matrix of size $n \times m$ (*n* and *m* are the lengths of the two sequences) over the two sequences. The attention matrix contains the scores which are a measure of the relationship between their elements together with their respective positions. The score value indicates the amount of attention a word at position *p* in the first sequence pays to a word at position *q* in the second sequence. Figure 4-16 shows a sample attention matrix for an English-Spanish translation task. Matrix cells represent the attention scores between the respective english and spanish words for an example pair of sequences. A higher score indicates a greater affinity between the corresponding words and their positions. The cell with value 0.60 at (0, 0) indicates that the first position in the english sequence has a strong relationship with the first element in the spanish sequence. That makes sense because typically[21] the sentences in both the languages begin with a subject.

The attention matrix contains scores for each pair of elements from the two sequences. It takes into account, for instance, the relationship of the first element in the first sequence and the last element in the second sequence. Hence, it addresses the limitations of RNN with long term dependencies.

---

[19] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[20] Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).

[21] Note that in the Spanish language, pronouns can be omitted. "*Estoy muy bien*" and "*Muy bien*" are identical.

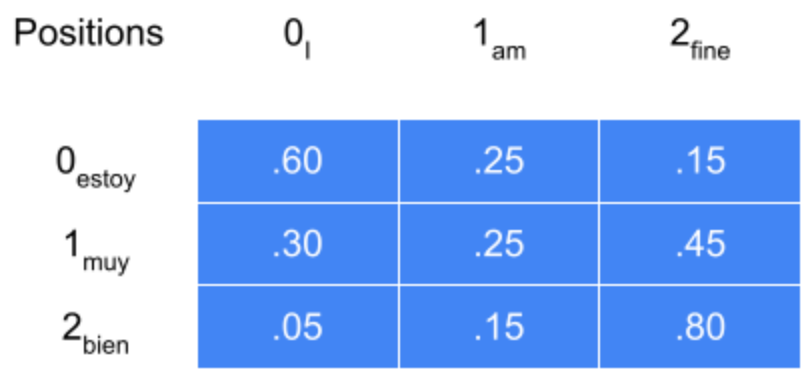


Figure 4-16: Sample attention matrix for a Spanish-English translation task. Matrix cells represent the attention scores between the positions for a pair of sequences.

In their seminal paper titled: Attention Is All You Need[22], Vaswani et. al. introduced a novel architecture called *Transformer* which achieved state of the art performance for common NLP tasks primarily using attention. A transformer replaces the recurrent units in the RNN Encoder-Decoder model with attention layers. It uses two flavors of attention: encoder-decoder attention and self-attention. Encoder-decoder attention computes attention between the encoder output sequence and the target sequence. Self-attention is a special type of attention which operates over a single sequence to compute the relationship between its own elements. It replaces the recurrent units in the encoder and the decoder blocks. Although there are other differences between a RNN Encoder-Decoder and a Transformer, the replacement of recurrent cells with attention layers is the most significant one.

The goal of attention is to produce a score matrix (figure 4-16) for a pair of sequences to supplement a target sequence with the knowledge of the relationship between the sequence elements. Over the years, several algorithms have been proposed to compute the score matrix such as the Luong[23] style and the Bahdanau[24] style attention. In this book, we have chosen to discuss the Luong algorithm because it is used in Tensorflow's attention layers. However, we encourage the interested readers to further explore the attention domain.

The attention (Luong) mechanism learns three weight matrices namely W_Q (query weight), W_K (key weight) and W_V (value weight) which are used to compute the query, key and value matrices for input sequences. Then, a softmax is applied to the scaled dot product of query and key matrices to obtain a score matrix (figure 4-16). Finally, the values are weighted based on the positional relationship encoded in the score matrix to obtain the final representation.

---

[22] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
[23] Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. "Effective approaches to attention-based neural machine translation." *arXiv preprint arXiv:1508.04025* (2015).
[24] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

Mathematically, we are given a pair of sequences X and Y with shapes *(n, d) and (m, d)* where *n* and *m* are the number of elements in each sequence and *d* represents the number of dimensions of each element. The weight matrices W_Q, W_K, and W_V are identically shaped as *(d, d_k)*. The query, key and the value matrices are computed as follows:

$$\begin{aligned} \text{Query} \quad & Q = XW_Q \\ \text{Key} \quad & K = YW_K \\ \text{Value} \quad & V = YW_V \end{aligned}$$

The resulting Q, K, and V matrices are shaped as *(n, d_k), (m, d_k),* and *(m, d_k)* respectively. Next, we compute the score matrix,

$$\text{Scores} \quad S = Softmax(QK^T/\sqrt{d_k}))$$

The score matrix has dimensions *(n, m).* The final representation, *Z* for the target sequence is computed as,

$$\text{Final Representation} \quad Z = SV$$

The mechanism to compute self-attention is identical. It uses the same sequence as both X and Y. In case of self-attention, the final representation Z contains information about the relationship between the sequence elements. In the next section, we will use self-attention to classify news articles.

News Classification Using RNN and Attention Models

Let's apply our learnings about RNN and attention to classify the news articles in AGNews[25] dataset. AGNews is a collection of news articles where each article belongs to one of the following four classes: Sci/Tech, World, Sports and Business. Our task here is to compare the performances of a RNN and an attention model on this dataset. First, we will train a RNN model using a GRU cell to establish a *baseline*. The training of the attention model will follow right after. Finally, we will compare their training efficiency and quality metrics. As always, the code is available as a Jupyter notebook **here** for you to experiment.

Let's get started with loading the dataset.

```
import tensorflow as tf
import tensorflow_datasets as tfds

from tensorflow.keras import layers, optimizers, metrics

train_ds, = tfds.load('ag_news_subset', split=['train[:50%]'], as_supervised=True)
```

[25] Zhang, Xiang, Junbo Zhao, and Yann LeCun. "Character-level convolutional networks for text classification." *Advances in neural information processing systems* 28 (2015): 649-657.

We vectorize the input samples using a TextVectorization layer. This layer collects the most frequent words in the text to create a vocabulary of predefined size. We have chosen a vocabulary size of 5000 words. Keep in mind that a large vocabulary results in a bigger model which has higher latencies and a bigger footprint. On the other hand, a small vocabulary affects the model quality. The vectorization layer maps each word in the input sequence to an integer value. New words (the words which are not in the vocabulary) are also assigned a value 0 (or other fixed value). We truncate or pad sequences to ensure equal length sequences. Longer sequences are truncated and the smaller sequences are padded with 0s.

```
VOCAB_SIZE = 5000
SEQ_LEN = 64

encoder = tf.keras.layers.TextVectorization(max_tokens=VOCAB_SIZE,
output_sequence_length=SEQ_LEN)
encoder.adapt(train_ds.map(lambda text, label: text))

def encode_fn(text, label):
    return (encoder(text), label)
```

We train a single layer RNN baseline model for 5 epochs. This model contains an input layer, an embedding layer, an RNN layer that uses a GRU cell followed by a classification head.

```
def create_rnn_model():
    model = tf.keras.Sequential([
        tf.keras.Input(shape=(SEQ_LEN), dtype=tf.float32),
        layers.Embedding(len(encoder.get_vocabulary()), 62),
        layers.RNN(layers.GRUCell(SEQ_LEN)),
        layers.Dense(4, activation='softmax')
    ])

    adam = optimizers.Adam(learning_rate=.001)
    loss = 'sparse_categorical_crossentropy'
    metrics = 'accuracy'

    model.compile(optimizer=adam, loss=loss, metrics=metrics)
    return model

rnn_model = create_rnn_model()
rnn_model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_7 (Embedding)     (None, 64, 62)            310000

 rnn_5 (RNN)                 (None, 64)                24576
```

```
 dense_7 (Dense)             (None, 4)                 260

=================================================================
Total params: 334,836
Trainable params: 334,836
Non-trainable params: 0
_________________________________________________________________
```

```
# Train the model
rnn_model.fit(train_ds.batch(256).map(encode_fn), epochs=5)
```

```
Epoch 1/5
235/235 [==============================] - 31s 124ms/step - loss: 1.3863 - accuracy: 0.2525
Epoch 2/5
235/235 [==============================] - 29s 123ms/step - loss: 1.3329 - accuracy: 0.3023
Epoch 3/5
235/235 [==============================] - 28s 117ms/step - loss: 0.8278 - accuracy: 0.5762
Epoch 4/5
235/235 [==============================] - 29s 124ms/step - loss: 0.4669 - accuracy: 0.8379
Epoch 5/5
235/235 [==============================] - 30s 126ms/step - loss: 0.3242 - accuracy: 0.8948
```

Let's compare the baseline model to a similar size attention model. The attention model simply replaces the recurrent layer in the baseline model with an attention layer which scores the relationship between the sequence elements.

```
def create_model():
    text = tf.keras.Input(shape=(SEQ_LEN), dtype=tf.float32)
    embedding = layers.Embedding(len(encoder.get_vocabulary()), 64)(text)

    self_attention = layers.Attention()([embedding, embedding])
    flattened = layers.Flatten()(self_attention)
    output = layers.Dense(4, activation='softmax')(flattened)

    model = tf.keras.Model(inputs=[text], outputs=[output])

    adam = optimizers.Adam(learning_rate=.001)
    loss = 'sparse_categorical_crossentropy'
    metrics = 'accuracy'

    model.compile(optimizer=adam, loss=loss, metrics=metrics)
    return model

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

```
Model: "model_1"
```

```
__________________________________________________________________________________________________
___
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
===
 input_6 (InputLayer)           [(None, 64)]         0           []

 embedding_5 (Embedding)        (None, 64, 64)       320000      ['input_6[0][0]']

 attention_1 (Attention)        (None, 64, 64)       0           ['embedding_5[0][0]',
                                                                  'embedding_5[0][0]']

 flatten_1 (Flatten)            (None, 4096)         0           ['attention_1[0][0]']

 dense_5 (Dense)                (None, 4)            16388       ['flatten_1[0][0]']

==================================================================================================
===
Total params: 336,388
Trainable params: 336,388
Non-trainable params: 0
__________________________________________________________________________________________________
___
```

Let's train it for 5 epochs as well to observe the behavior of the attention layer.

```
model.fit(train_ds.batch(256).map(encode_fn), epochs=5)
```

```
Epoch 1/5
235/235 [==============================] - 12s 46ms/step - loss: 0.6550 - accuracy: 0.7823
Epoch 2/5
235/235 [==============================] - 11s 47ms/step - loss: 0.3117 - accuracy: 0.8975
Epoch 3/5
235/235 [==============================] - 11s 49ms/step - loss: 0.2700 - accuracy: 0.9090
Epoch 4/5
235/235 [==============================] - 11s 48ms/step - loss: 0.2478 - accuracy: 0.9151
Epoch 5/5
235/235 [==============================] - 11s 46ms/step - loss: 0.2333 - accuracy: 0.9196
```

Recall the buffet example at the beginning of this section in which customers' order of choice of food items is based on *how quickly those items catch their eye*. Similarly, the attention layer scores the affinity of every element in the first sequence with every element in the second sequence. In our attention model, both the sequences are the same. So, the attention layer scores the affinity of every element with all the elements in the sequence including itself. This is called self-attention. Below, we define a function *get_attention_scores()* to extract the self-attention scores learned by the attention layer for an input text. We also define a *decode()* function to map the vectorized text to the words in the input text.

```
def decode(items, vocab):
    text_items = []
```

```
    for item in items:
        text_item = [vocab[token] for token in item]
        text_items.append(text_item)

    return text_items

def get_attention_scores(text):
    # Extract embedding and attention layers from the model
    embedding_layer, attention_layer = model.layers[1], model.layers[2]
    encoding = encoder([text])
    embedding = embedding_layer(encoding)

    # We use the same embedding to perform self-attention
    _, scores = attention_layer([embedding, embedding], return_attention_scores=True)

    return encoding, scores

text = "Rupert Murdoch #39;s News Corporation today took steps to head off a potential takeover by rival mogul John Malone with a  quot;poison pill quot; plan to dilute the shareholdings of would-be predators."
encoding, scores = get_attention_scores(text)
text_tokens = decode(encoding, encoder.get_vocabulary())
print('Text tokens: ', text_tokens)
print('Tokens: ', encoding)
print('Self-Attention scores: ', scores)
```

```
Text tokens:  [['rupert', 'murdoch', '39s', 'news', 'corporation', 'today', 'took', 'steps',
'to', 'head', 'off', 'a', 'potential', 'takeover', 'by', 'rival', '[UNK]', 'john', '[UNK]',
'with', 'a', '[UNK]', '[UNK]', 'quot', 'plan', 'to', '[UNK]', 'the', '[UNK]', 'of', '[UNK]',
'[UNK]', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '']]
Tokens:  tf.Tensor(
[[3626 3688   11  129 1404   80  309 2393    4  303  106    3 1069  694
    18  409    1  218    1   12    3    1    1   83  248    4    1    2
     1    5    1    1    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0]], shape=(1, 64), dtype=int64)
Self-Attention scores:  tf.Tensor(
[[[0.02609748 0.02851956 0.0181938  ... 0.01508285 0.01508285 0.01508285]
  [0.0276615  0.04025185 0.01893727 ... 0.01453829 0.01453829 0.01453829]
  [0.01833969 0.01968127 0.02290314 ... 0.01522904 0.01522904 0.01522904]
  ...
  [0.01553787 0.01544147 0.01556366 ... 0.01571876 0.01571876 0.01571876]
  [0.01553787 0.01544147 0.01556366 ... 0.01571876 0.01571876 0.01571876]
   [0.01553787 0.01544147 0.01556366 ... 0.01571876 0.01571876 0.01571876]]], shape=(1, 64,
64), dtype=float32)
```

The scores are a 64x64 dimensional matrix which indicates the affinity between the sequence positions. Let's pick a word, say 'corporation', and take a look at its scores. The word 'corporation' lies at index 4 in the vectorized representation.

```
# Attention scores for the word 'corporation'
scores[0, 4]
```

```
<tf.Tensor: shape=(64,), dtype=float32, numpy=
array([0.01683364, 0.02314224, 0.01643265, 0.01896303, 0.0370173 ,
       0.01553664, 0.01117901, 0.01904086, 0.01551224, 0.01027294,
       0.01754347, 0.01482745, 0.01072181, 0.02153939, 0.01743654,
       0.01658158, 0.01486312, 0.01386998, 0.01486312, 0.01363553,
       0.01482745, 0.01486312, 0.01486312, 0.01445088, 0.01655908,
       0.01551224, 0.01486312, 0.01399966, 0.01486312, 0.0166438 ,
       0.01486312, 0.01486312, 0.01496924, 0.01496924, 0.01496924,
       0.01496924, 0.01496924, 0.01496924, 0.01496924, 0.01496924,
       0.01496924, 0.01496924, 0.01496924, 0.01496924, 0.01496924,
       0.01496924, 0.01496924, 0.01496924, 0.01496924, 0.01496924,
       0.01496924, 0.01496924, 0.01496924, 0.01496924, 0.01496924,
       0.01496924, 0.01496924, 0.01496924, 0.01496924, 0.01496924,
       0.01496924, 0.01496924, 0.01496924, 0.01496924], dtype=float32)>
```

It primarily attends to three words (figure 4-18) in the text: *murdoch* (index 1)*, corporation* (index 4) and *takeover* (index 13)*.* Other than itself, the other two words are quite far apart in the sequence. Moreover, the attention layer attends to both past and future positions.

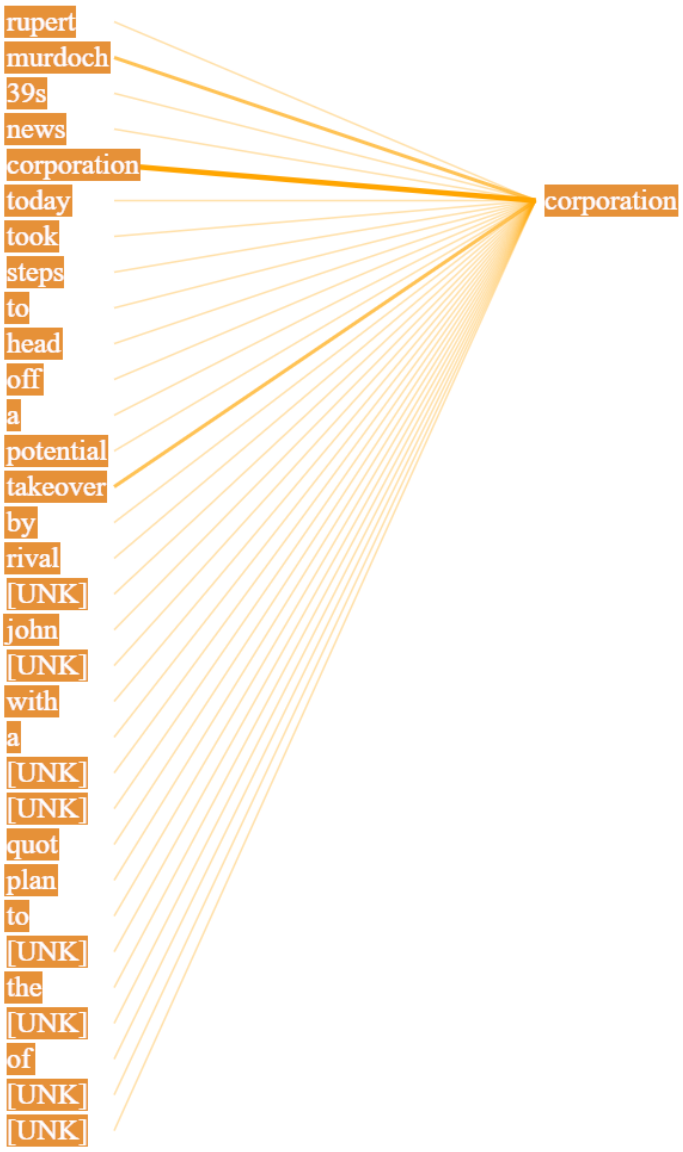


Figure 4-18: A visual representation of the attention the word *corporation* pays to each sequence element. The left column is a tokenized representation of our example text. The thickness of the connecting lines between left and right columns represent the strength of attention. The word *corporation* pays most attention to the words *takeover* and *murdoc* in the sequence. Note that these words are far apart. Moreover, the word *corporation* takes into account both the past and future words.

Table 4-3 shows a comparison of the quality metrics and the latencies of the two models. The attention model trains 3X faster than its counterpart. It also achieves marginally higher accuracy in the same training epochs. The inference with attention model is significantly faster. Because of these advantages over RNN in dealing with the sequence data, the attention mechanism is

widely adopted in NLP, speech and image processing, and other fields of deep learning. It is used as a backbone in many state of the art text and visual models like BERT, GPT-2 and ViT.

Model	Accuracy (%)	Training Latency (seconds/epoch)	Inference Latency[26] (microseconds)
RNN	89	29	19,726
Attention	92	11	513

Table 4-3: A comparison of RNN and attention models on various metrics. The attention model has higher accuracy, faster training and inference latencies.

On the flipside, attention has higher memory and computational requirements. The score matrix requires $\mathcal{O}(n \times m)$ memory for computation. The size of the score matrix grows with the increase in the lengths of the two sequences. In comparison, RNN memory requirements are independent of their lengths. Similar to the memory requirements, an attention layer's computation requirements are also quadratic as compared to the linear computation complexity of RNNs. However, attention is still faster in wall clock time because it processes entire sequences together.

The quadratic complexity of attention is addressed through several works[27] where the authors propose transformer variants with efficient self-attention mechanisms. These ideas tackle the quadratic complexity at various levels. The simplest idea is to chunk the input sequence of length *n* into blocks of length *b* where *b* <<< *n*. The resulting score matrices have computation and memory complexities of the order $\mathcal{O}(b^2)$. Further, these blocks can be strided, randomly chosen or even learnt during the training process. The Fixed/Factorized/Random and Learnable Pattern groups in figure 4-19 show the examples of efficient transformers based on these optimizations. Some efficient transformers connect these blocks via a recurrence cell. They fall under the Recurrence group.

The efficient transformers under the Memory/Downsampling group use additional parameters to act as a memory. This memory is used to sample most relevant parts of the input sequence, thus compressing it in the process. S*parse attention* attempts to reduce the complexity by picking a subset of parameters for attention computation. This subset can also be learned during the training process. The transformers that use sparse attention are grouped under the Sparse group. After input sequence and the attention parameters, the next component to attack is the softmax computation. The Low Rank methods project the keys and the values to a smaller dimension *k* to reduce the computation and memory complexity of the score matrix to the order of $\mathcal{O}(n \times k)$. Further, there are algorithms which compute softmax using clever approximations.

[26] The inference latency is measured with a TFLite converted model on an ARM Android device.
[27] Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*.

The transformers which leverage the low rank methods and kernel approximations are grouped under Low Rank/Kernel group.

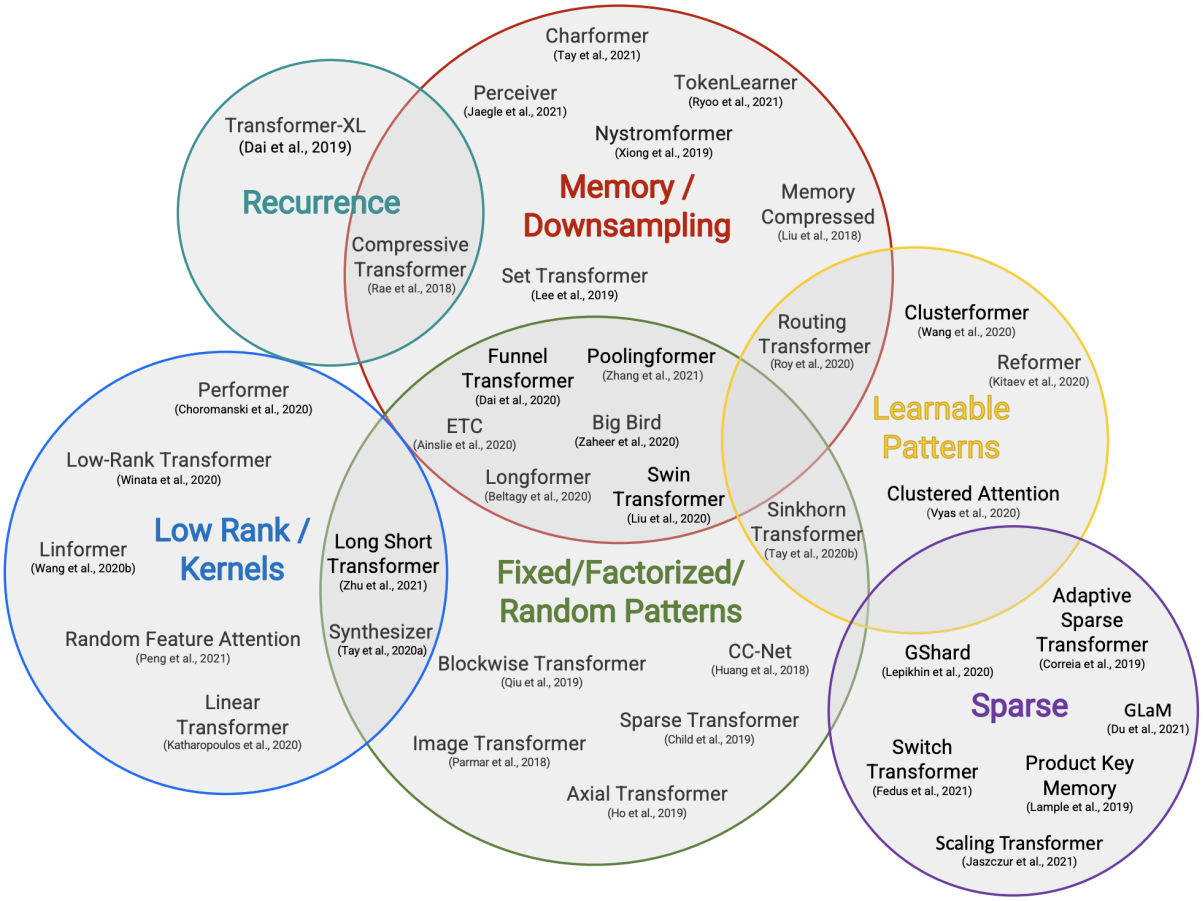


Figure 4-19: Taxonomy of efficient transformers, depicting the various approaches to reduce the quadratic complexity of the self-attention layer. (Source: Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: A survey. arXiv preprint arXiv:2009.06732.)

While attention provided a breakthrough for efficiently learning from sequential data, depthwise separable convolution extended the reach of convolution models to mobile and other devices with limited compute and memory resources. This layer aims to reduce the footprint of convolutional layers with minimal quality compromise. The next section describes it in detail.

Efficient On-Device Convolutions

Depthwise Separable Convolution[28] (DSC) is a two-step convolution designed to reduce the computational complexity of regular convolution. Its output shape is identical to a convolution layer which makes it an attractive choice to replace convolution blocks in the models deployed

[28] Chollet, François. "Xception: Deep learning with depthwise separable convolutions." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

on mobile and edge devices. Let's say you want to design a mobile application to highlight pets in a picture. A DSC model is a perfect choice for such an application because it has a smaller footprint than a regular convolution network. Howard et. al. demonstrated that their proposed MobileNets[29] (DSC model family for mobile vision applications) perform at par with the regular convolution as shown in table 4-4. They are significantly smaller and require fewer computations in comparison to their regular cousins while making a small compromise in accuracy.

Mobilenet	ImageNet Accuracy @ 1	Mult-Adds Ops (Millions)	Parameters (Millions)
Regular Convolution	71.7%	4866	29.3
Depthwise Separable	70.6%	569	4.2

Table 4-4: A comparison of quality (accuracy) and footprint metrics (number of multiplication-addition ops, and number of parameters) between two flavors of MobileNets on the ImageNet classification task

Consider an input with dimensions *(h, w, m)* where *h, w* represent the spatial dimensions (height and width) and *m* is the number of input channels. Figure 4-20 demonstrates a regular convolution operation over this input using *n* kernels of dimensions (d_k, d_k, m) where d_k is the spatial dimension of each kernel. The regular convolution operation with a single stride produces an output with dimensions *(h, w, n)* where *n* is the number of output channels. This operation requires $h \times w \times n \times d_k \times d_k \times m$ operations.

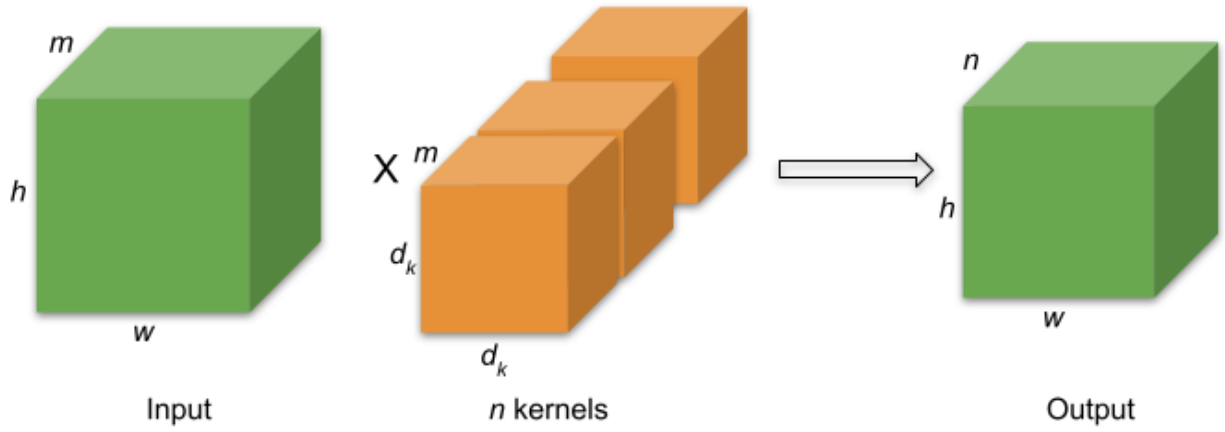


Figure 4-20: Depiction of input, output and kernel shapes for a regular convolution with single stride.

[29] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).

On the other hand, a DSC block (figure 4-21) performs two step convolution. In the first step, the input is convolved with *m* (d_k, d_k, *1)* shaped kernels. The *i-th* channel of the input is convolved with the *i-th* kernel. It involves *h x w x m x* d_k *x* d_k operations and produces *a (h, w, m) shaped* output.

The second step performs a pointwise convolution using *n (1, 1, m)* dimensional kernels. It requires *h x w x m x n* operations. Hence, the total number of operations are *h x w x m x* (d_k *x* d_k *+ n).*

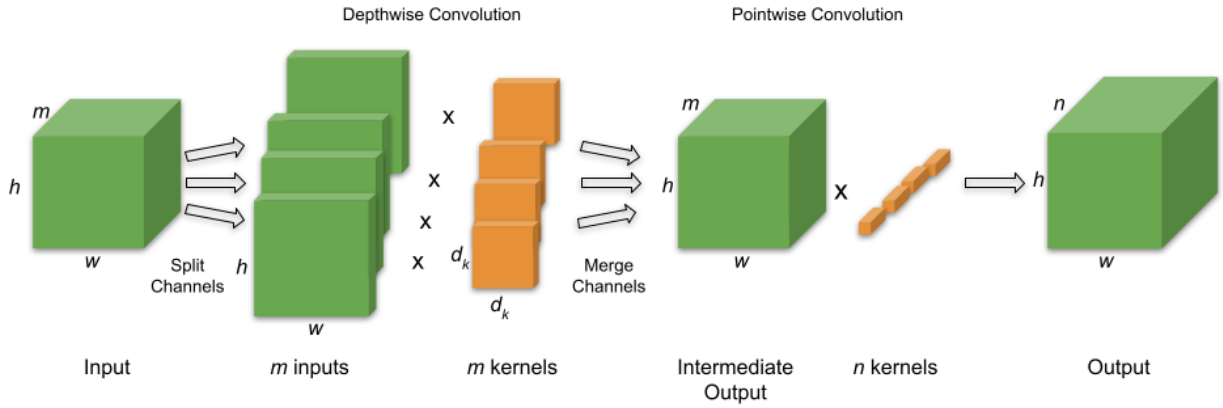


Figure 4-21: Depiction of input, output and kernel shapes for a depthwise separable convolution.

Let's work out the computations to convolve a *(64, 64, 3)* dimensional input with a *(3, 3)* kernel to generate an output with dimensions *(64, 64, 32).*

$$
\begin{aligned}
h, w &= 64 \\
m &= 3 \\
d_k &= 3 \\
n &= 32 \\
\text{Standard Convolution Computations} &= 64 * 64 * 32 * 3 * 3 * 3 \\
&= 3,538,944 \\
\text{DSC Computations} &= 64 * 64 * 3 * (3 * 3 + 32) \\
&= 503,808
\end{aligned}
$$

In this example, the number of regular convolution operations are 7X more than the number of DSC operations. How do DSC models perform on the quality metrics? To answer this question, we have prepared a programming project for you in the next section. We take up a novel task to train a model that predicts a segmentation mask over an object in the input sample. This model will be used within a mobile application. Mobile devices are resource constrained. Let's see if we can reduce the model footprint without a significant quality compromise.

Project: Snapchat-Like Filters for Pets

Popular social media applications like Instagram or Snapchat have filters which can be applied over photos. For example, a mustache filter adds mustache to the faces in a photo. Have you ever wondered how they do that? The first step to develop such a feature is to predict masks over the faces. This is followed by patching the filter over the masked area. In this project, we will train a model to produce a mask over a pet in an image. This model will be deployed with a pet filter application for mobile devices which would let you replace one pet with another. We will show you the first step to predict the pet masks and leave the second part of applying filters to the reader. Since this model will be deployed on mobile devices, footprint metrics along with quality are the key concerns. Given an image of a pet, our model will predict a segmentation mask over the pet. An example input image and segmentation mask is shown in figure 4-22. We are going to use the Oxford-IIIT Pet dataset[30] for training purposes. It contains images and segmentation masks for 37 pet categories. We will train two models using regular convolutions and depthwise separable convolutions respectively. It will be followed by an analysis of their performance on key metrics. The code is below, but the entire Jupyter notebook is **here** for you to experiment with.

We begin with loading the necessary modules. The Oxford-IIIT dataset is available through the tensorflow_datasets package. We apply the standard preprocessing routines to resize and normalize the images.

```
import tensorflow as tf
import tensorflow_datasets as tfds

from tensorflow.keras import layers, optimizers, metrics, losses, utils, applications
as apps, callbacks as cbs
from tensorflow.keras.utils import plot_model
from matplotlib import pyplot as plt

# Load Oxford-IIIT Pet Dataset
train_ds, test_ds = tfds.load(
    'oxford_iiit_pet:3.*.*',
    split=['train', 'test'],
    read_config=tfds.ReadConfig(try_autocache=False)
)

IMG_SIZE = 128

def preprocess(item):
    image, mask = item['image'], item['segmentation_mask']

    # Resize image and mask to IMG_SIZE
    image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])
```

[30] Parkhi, Omkar M., et al. "Cats and dogs." *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012.

```
    mask = tf.image.resize(mask, [IMG_SIZE, IMG_SIZE])

    # Normalize the image and realign the mask.
    image = tf.cast(image, tf.float32) / 255.0
    mask -= 1

    return image, mask

train_prep_ds = train_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE)
test_prep_ds = test_ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE)
```

The input to the models are 128x128x3 dimensional pet images. The models predict a 128x128x3 dimensional output. The first two output dimensions correspond to the pixel location. The last dimension is a one hot representation of mask value. A mask value can be 0, 1 or 2. Mask value 0 indicates that the pixel belongs to the foreground (the pet), the value 1 implies that it is a background (surroundings) pixel and 2 represents the boundary pixel. Both models have identical architecture with the exception of convolutional layers.

We define a *create_model()* function capable of creating a regular convolution model as well as a depthwise separable convolution model. The model contains a set of downsampling layers followed by a set of upsampling layers. The downsampling layers are based on MobileNets. They create an intermediate representation of the input. The upsampling layers reshape the intermediate representation to the required output dimensions while preserving the spatial features. They are required to achieve our task specific goal to classify each input pixel into one of the three output classes. We also define a *get_conv_builder()* function that chooses between a regular convolution block and a depthwise convolution block.

```
LEARNING_RATE = 0.001
N_CLASSES = 3

layer_id = -1
def get_layer_id():
    global layer_id
    layer_id += 1

    return str(layer_id)

def get_regular_conv_block(filters, strides):
    return tf.keras.Sequential(
        [
            layers.Conv2D(filters, 3, padding='same', strides=strides),
            layers.BatchNormalization(),
            layers.ReLU(),
        ],
        name='regular_conv_block_' + get_layer_id()
    )
```

```
def get_dsc_block(filters, strides):
    return tf.keras.Sequential(
        [
            layers.DepthwiseConv2D(3, padding='same', strides=strides),
            layers.BatchNormalization(),
            layers.ReLU(),
            layers.Conv2D(filters, 1, padding='same', strides=1),
            layers.BatchNormalization(),
            layers.ReLU(),
        ],
        name='dsc_block_' + get_layer_id()
    )

def get_conv_upsampling_block(filters):
    """
        It creates a block to upsample i.e. increase the spatial dimensions
        of the inputs. Upsampling is required to ensure that the model
        outputs three probabilities corresponding to the three classes for
        each input pixel as required by the task.
    """
    initializer = tf.random_normal_initializer(0., 0.02)
         kwargs  =  dict(strides=2,  padding='same',  kernel_initializer=initializer,
use_bias=False)

    return tf.keras.Sequential(
        [
            layers.Conv2DTranspose(filters, 3, **kwargs),
            layers.BatchNormalization(),
            layers.ReLU()
        ],
        name='conv_upsampling_block_' + get_layer_id()
    )

def get_conv_builder(block_type='conv'):
    if (block_type == 'conv'):
        return get_regular_conv_block
    elif (block_type == 'dsc'):
        return get_dsc_block
    else:
        raise ValueError('Invalid block type: ', block_type)

def create_model(conv_type='conv'):
    get_conv_block = get_conv_builder(conv_type)

    model = tf.keras.Sequential([
        tf.keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),
```

```
        get_regular_conv_block(32, strides=2),
        get_conv_block(64, strides=1),
        get_conv_block(128, strides=2),
        get_conv_block(128, strides=1),
        get_conv_block(256, strides=2),
        get_conv_block(256, strides=1),
        get_conv_block(512, strides=2),
        get_conv_block(512, strides=1),
        get_conv_block(512, strides=1),
        get_conv_block(512, strides=1),
        get_conv_block(512, strides=1),
        get_conv_block(512, strides=1),
        get_conv_block(1024, strides=2),
        get_conv_upsampling_block(512),
        get_conv_upsampling_block(256),
        get_conv_upsampling_block(128),
        get_conv_upsampling_block(64),
        get_conv_upsampling_block(3),
    ])

    optimizer = optimizers.Adam(learning_rate=LEARNING_RATE)
    loss = losses.SparseCategoricalCrossentropy(from_logits=True)
    model.compile(optimizer=optimizer, loss=loss, metrics='accuracy')

    return model
```

Let's define a *train()* function to simplify the training process as follows.

```
EPOCHS = 50
BATCH_SIZE = 16

def train(model, tds, vds, epochs, callbacks=[]):
    tds = tds.prefetch(buffer_size=tf.data.AUTOTUNE)
    vds = vds.prefetch(buffer_size=tf.data.AUTOTUNE) if vds else None
    history = model.fit(tds, validation_data=vds, epochs=epochs, callbacks=callbacks)

    return history
```

With all the foundational work behind us, let's create and train the regular convolution model for 50 epochs.

```
conv_model = create_model()
conv_model.summary()
```

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 regular_conv_block_180 (Sequential)    (None, 64, 64, 32)     1024

 regular_conv_block_181 (Sequential)    (None, 64, 64, 64)     18752

 regular_conv_block_182 (Sequential)    (None, 32, 32, 128)    74368

 regular_conv_block_183 (Sequential)    (None, 32, 32, 128)    148096

 regular_conv_block_184 (Sequential)    (None, 16, 16, 256)    296192

 regular_conv_block_185 (Sequential)    (None, 16, 16, 256)    591104

 regular_conv_block_186 (Sequential)    (None, 8, 8, 512)      1182208

 regular_conv_block_187 (Sequential)    (None, 8, 8, 512)      2361856

 regular_conv_block_188 (Sequential)    (None, 8, 8, 512)      2361856

 regular_conv_block_189 (Sequential)    (None, 8, 8, 512)      2361856

 regular_conv_block_190 (Sequential)    (None, 8, 8, 512)      2361856

 regular_conv_block_191 (Sequential)    (None, 8, 8, 512)      2361856

 regular_conv_block_192 (Sequential)    (None, 4, 4, 1024)     4723712

 conv_upsampling_block_193 (Sequential)  (None, 8, 8, 512)     4720640

 conv_upsampling_block_194 (Sequential)  (None, 16, 16, 256)   1180672

 conv_upsampling_block_195 (Sequential)  (None, 32, 32, 128)   295424

 conv_upsampling_block_196 (Sequential)  (None, 64, 64, 64)    73984

 conv_upsampling_block_197 (Sequential)  (None, 128, 128, 3)   1740

=================================================================
Total params: 25,117,196
Trainable params: 25,105,350
Non-trainable params: 11,846
_________________________________________________________________
```

```
tds = train_prep_ds.cache().shuffle(1000, reshuffle_each_iteration=True)
tds = tds.batch(BATCH_SIZE)
vds = test_prep_ds.batch(256).cache()

train(conv_model, tds, vds, epochs=EPOCHS)
```

```
Epoch 1/50
230/230 [==============================] - 69s 241ms/step - loss: 0.8638 - accuracy: 0.5524 -
val_loss: 1.8870 - val_accuracy: 0.4178
xxxxxxxxx Skip to 42nd epoch xxxxxxxxx
```

```
Epoch 42/50
230/230 [==============================] - 45s 196ms/step - loss: 0.0936 - accuracy: 0.9549 -
val_loss: 0.4974 - val_accuracy: 0.8586
xxxxxxxxx Skip to 50th epoch xxxxxxxxx
Epoch 50/50
230/230 [==============================] - 45s 196ms/step - loss: 0.0808 - accuracy: 0.9596 -
val_loss: 0.5647 - val_accuracy: 0.8554
```

Let's create and train the DSC model as well.

```
dsc_model = create_model(conv_type='dsc')
dsc_model.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 regular_conv_block_198 (Sequential)    (None, 64, 64, 32)    1024

 dsc_block_199 (Sequential)             (None, 64, 64, 64)    2816

 dsc_block_200 (Sequential)             (None, 32, 32, 128)   9728

 dsc_block_201 (Sequential)             (None, 32, 32, 128)   18816

 dsc_block_202 (Sequential)             (None, 16, 16, 256)   35840

 dsc_block_203 (Sequential)             (None, 16, 16, 256)   70400

 dsc_block_204 (Sequential)             (None, 8, 8, 512)     137216

 dsc_block_205 (Sequential)             (None, 8, 8, 512)     271872

 dsc_block_206 (Sequential)             (None, 8, 8, 512)     271872

 dsc_block_207 (Sequential)             (None, 8, 8, 512)     271872

 dsc_block_208 (Sequential)             (None, 8, 8, 512)     271872

 dsc_block_209 (Sequential)             (None, 8, 8, 512)     271872

 dsc_block_210 (Sequential)             (None, 4, 4, 1024)    536576

 conv_upsampling_block_211 (Sequential)  (None, 8, 8, 512)    4720640

 conv_upsampling_block_212 (Sequential)  (None, 16, 16, 256)  1180672

 conv_upsampling_block_213 (Sequential)  (None, 32, 32, 128)  295424

 conv_upsampling_block_214 (Sequential)  (None, 64, 64, 64)   73984

 conv_upsampling_block_215 (Sequential)  (None, 128, 128, 3)  1740

=================================================================
Total params: 8,444,236
Trainable params: 8,424,518
Non-trainable params: 19,718
_________________________________________________________________
```

```
tds = train_prep_ds.cache().shuffle(1000, reshuffle_each_iteration=True)
tds = tds.batch(BATCH_SIZE)
vds = test_prep_ds.batch(256).cache()

train(dsc_model, tds, vds, epochs=EPOCHS)
```

```
Epoch 1/50
230/230 [==============================] - 44s 162ms/step - loss: 0.8378 - accuracy: 0.6490 -
val_loss: 0.9283 - val_accuracy: 0.6302
xxxxxxxxx Skip to 50th epoch xxxxxxxxx
Epoch 50/50
230/230 [==============================] - 32s 140ms/step - loss: 0.1600 - accuracy: 0.9079 -
val_loss: 0.5852 - val_accuracy: 0.8061
```

Both the models locate the object in the image with high accuracy and predict similar masks as shown in figure 4-22. However, the DSC model is 3X smaller with 8.4 million parameters as compared to 25 million parameters of the regular model. Table 4-5 shows that the on-disk size of the depthwise separable model is roughly 3X smaller and it achieves 80.61% accuracy. The regular model achieves better quality with 85.86% accuracy but it requires 3X disk space to do that.

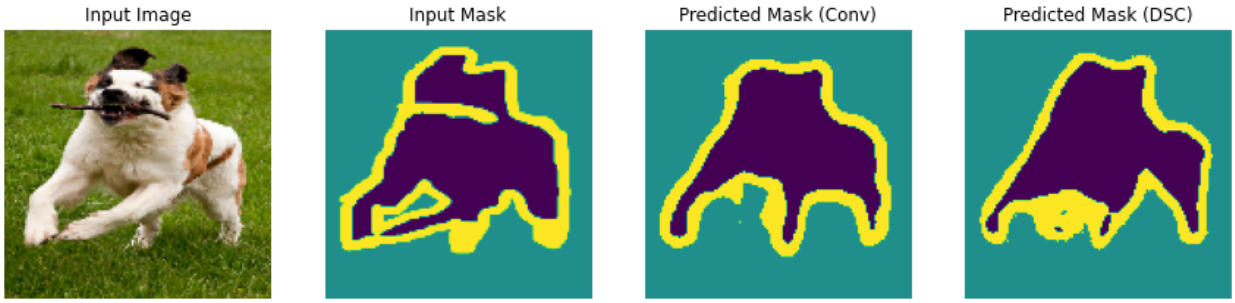


Figure 4-22: A comparison of segmentation masks predicted using the regular convolution model with the depthwise separable convolution model. The convolution block in the former is swapped with depthwise separable convolution block in the later. Otherwise, both the model's architectures are identical.

Model Type	Total Parameters	Accuracy	On Disk Size[31] (MB)	Training Latency (secs/epoch)	Inference[32] Latency (ms)
Regular	25,117,196	85.86%	288	45	1,620
Depthwise Separable	8,444,236	80.61%	97	32	496

Table 4-5: A comparison of latencies, quality and footprint metrics between regular and depthwise separable convolutions.

We started out with a goal to create a mobile friendly model to predict segmentation masks for pet images. We trained two models using two different convolution approaches. The regular convolution produced a better quality model. However, it requires roughly 3X more space than

[31] The disk size is calculated for a tensorflow model. Conversion to TFLite reduces their sizes to a ⅓.
[32] The inference latency is calculated with a TFLite converted model on an ARM Android device.

the depthwise separable model. The DSC model can fit in less than 100MB of memory. Despite this reduction, it still may not be suitable for a range of mobile and edge devices. Do you recall a technique that can reduce it further? Yes, Quantization! We will leave it for you as an exercise. Tell us how well it works!

Summary

This chapter was focussed on two different sets of architectures. The first set of architectures which includes embeddings and attention leverage learning the data distribution better to improve model quality. For example, embeddings learn the relationships between vocabulary elements, and attention learns the hidden affinity between the sequences and their elements to boost quality. The second set of architectures which include Depth Separable Convolutions are focussed on mobile and embedded environments where memory and footprint are an important concern. These architectures help to reduce the footprint with a minimal quality compromise as demonstrated through the project in the efficient on-device convolution section.

Efficient architectures provide a better footprint-quality tradeoff than their respective baseline model architectures. For instance, embeddings help by significantly reducing the dimensionality of the input. If the size of the vocabulary is 1 Million words, the naive input representation of a word token would be a one-hot vector of size 1 Million. This is both impractical to use because of the large input size, as well as would be hard to train because of the sparse input vector. While we saw in chapter 3 that we can legitimately exchange model quality and footprint by reducing / increasing the number of trainable model parameters, it will not work in this case because we have deeper scalability challenges with the baseline model. We need to redesign the model architecture itself. The embedding table approach removes this bottleneck, at which point we can scale the embedding model's quality and footprint metrics as discussed.

We can combine other ideas from compression techniques and learning techniques on top of efficient architectures. As an example, we can train an attention + embeddings based model using data augmentation to achieve higher performance and subsequently apply compression or distillation to further reduce its footprint.

With this chapter, we hope to have set the stage for your exploration of efficient layers and architectures for your deep learning projects. They can often be combined with other approaches like quantization, distillation, data augmentation, that we already learned. In the next chapter we will explore some more advanced model compression techniques like pruning.