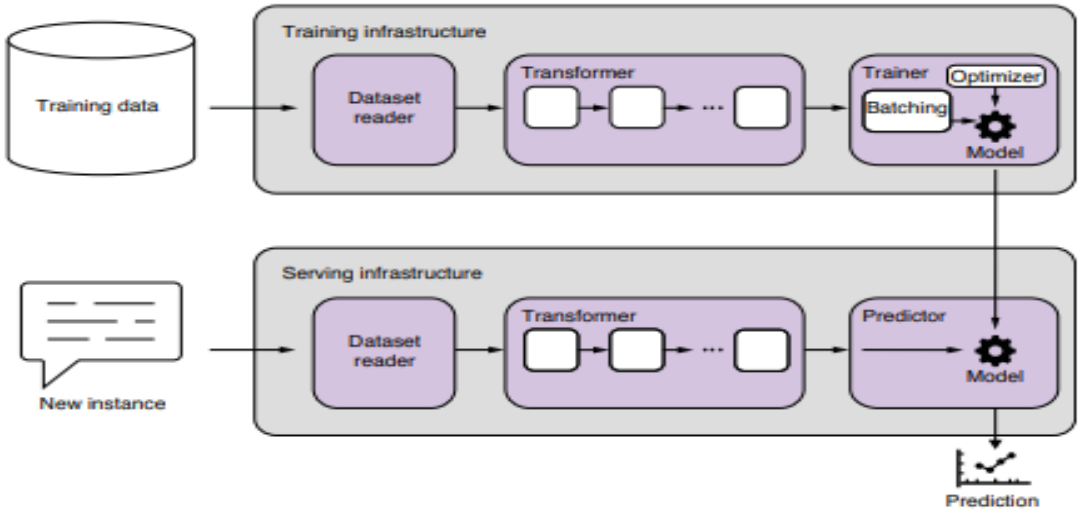
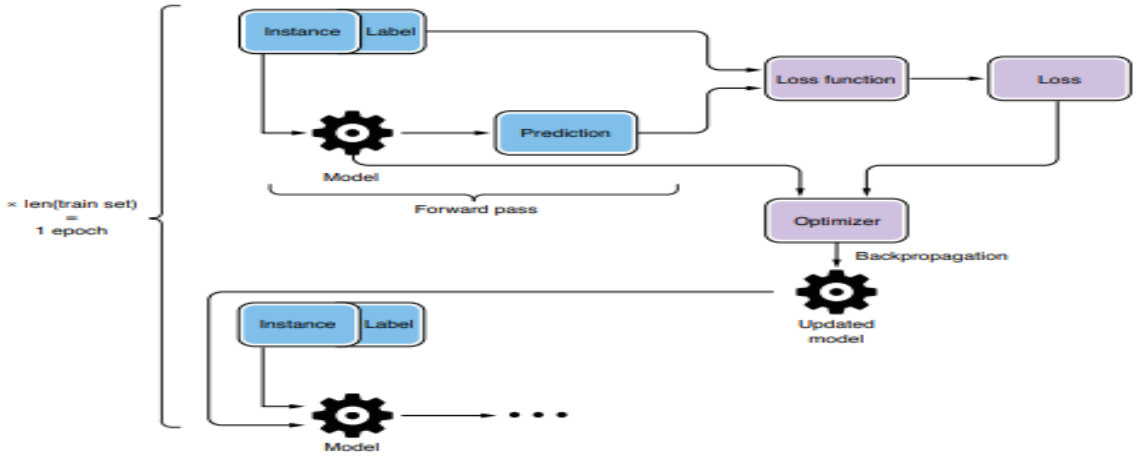
Building NLP applications

Typical structure of a modern NLP application



Train the NLP Model



- Deep neural network models such as RNNS, CNNS, and the Transformer, and modern NLP frameworks(Hugging Face)

➢ How do you train and make predictions efficiently?

➢ How do you avoid having your model over fit?

➢ How do you optimize hyper parameters?

➢ These factors could make a huge impact on the final performance and generalizability of your model.

NLP Models

- Logistic regression – 2 class binary sentimental analysis
- RNN- sequential labeling
- RNN with feedback – language models , Generation and detection
- Seq2 Seq- Encoder Decoder-Machine Tanslator,chatbot
- CNN – text classification- Document classification
- Seq2seq with attentions- machine Translation
- Seq2seq with self attentions- Transformers- MT,spell checking
- BERT- sentence classification- sentimental analysis
- NLI – sentence pair classification

How to build robust & accurate NLP application

Techniques used to build the model

- **Batching instances**- Padding, Sorting , Masking
- Handling **Unknown words** Tokenization for neural models - Character models, Subword models
- **Avoiding overfitting** -Regularization, Early stopping, Cross-validation

- **Dealing with imbalanced datasets** -Using appropriate evaluation metrics, Upsampling and downsampling, Weighting losses
- **Hyperparameter tuning**- epochs, parameters, Grid search vs. random search

1. **Batching**

- Batching is a machine learning technique where **instances are grouped together** to form batches and sent to the processor (CPU or, more often GPU).
- Batching is necessary when training large neural networks—it is critical for efficient and stable training.
- Batching is used for making model **more efficient computation**.
- Batching instances **methods**

➢ Padding,

➢ Sorting ,

➢ Masking

Batching- Padding

➢ Training large neural networks requires a number of linear algebra operations such as matrix addition and multiplication,

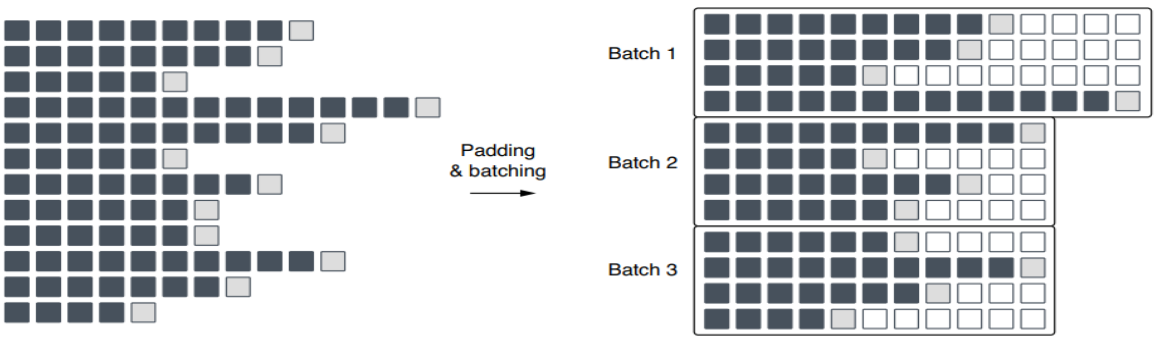
➢ it requires specialized hardware such as GPUs, processors designed to execute such operations in **a highly parallelized manner**

➢ Data as input is sent to the GPU as **tensors** (high-dimensional arrays) and do mathematical operations and the result is sent back as **another tensor**.

➢ In NLP, handling the sequences of text in different lengths.

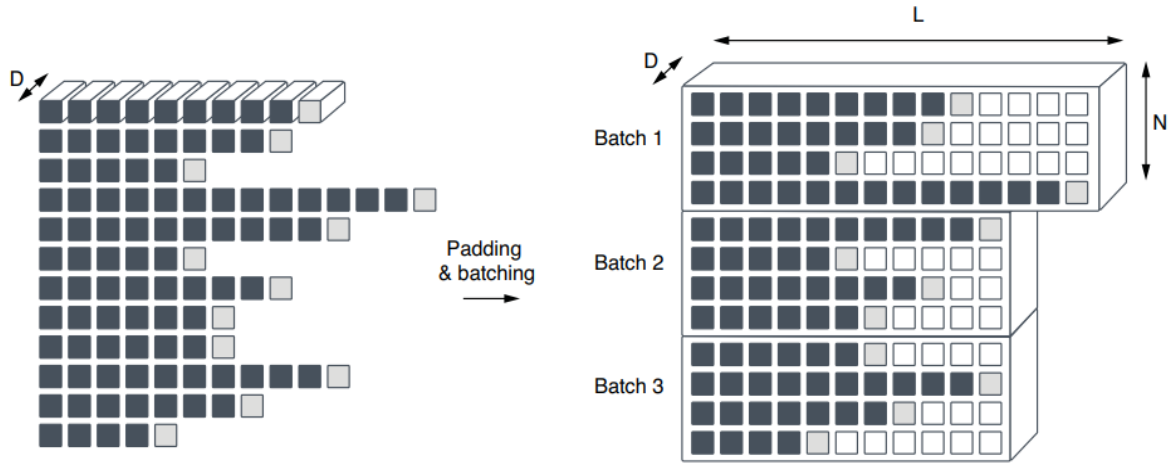
➢ Training the sequences in row should **be same length** and batches have to be **rectangular**, we need to do padding, (i.e., append special tokens,<PAD> ,

➢ The need to pad short sequences is to make as long as the longest sequence in the same batch. This is illustrated in figure

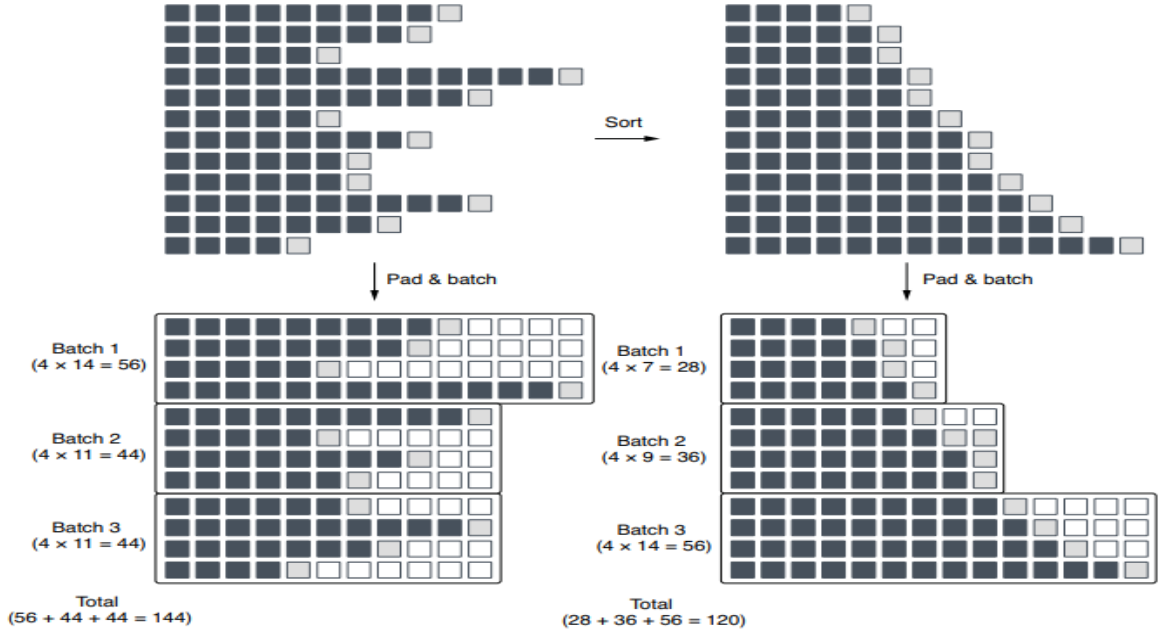


➢ Padding and batching. Black squares are tokens, gray ones are EOS tokens, and white ones are padding.

- Padding and batching of embedded sequences create rectangular, three-dimensional tensors.

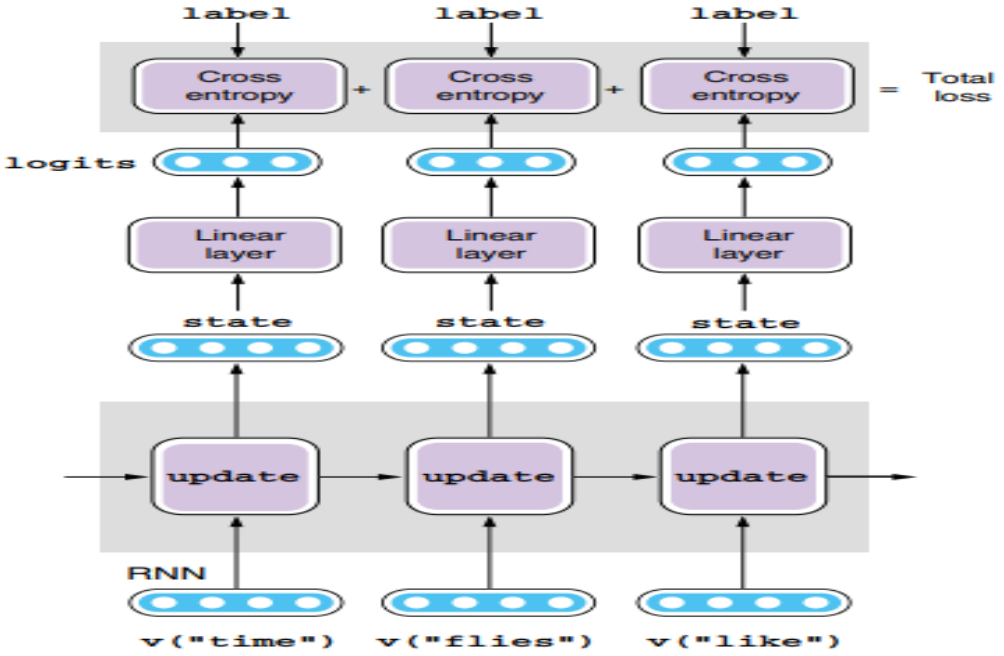


Sorting instances before batching (right) reduces the total number of tensors.



You can reduce the amount of padding by putting instances of similar size in the same batch. If shorter instances are batched only with other equally shorter ones, they don't need to get padded with many padding tokens. Similarly, if longer instances are batched only with other longer ones, they don't need a lot of padding either, because they are already long. One idea is to sort instances by their length and batch accordingly. Figure above compares two situations—one in which the instances are batched in their original order, and the other where instances are first sorted before batching. The numbers below each batch indicate how many tokens are required to represent the batch, including the padding tokens. Notice that the number of total tokens is reduced from 144 to 120 by sorting. Because the number of tokens in the original sentences doesn't change, this is purely because sorting reduced the number of padding tokens. Smaller batches require less memory to store and less computation to process, so sorting instances before batching improves the efficiency of training

Masking is an operation where you ignore some part of the network that corresponds to padding. This becomes relevant especially when you are dealing with a sequential-labeling or a language-generation model. To recap, sequential labeling is a task where the system assigns a label per token in the input sequence.



As shown in figure above, sequential-labeling models are trained by minimizing the per token loss aggregated across all tokens in a given sentence. We do this because we'd like to minimize the number of "errors" the network makes per token. This is fine as long as we are dealing with "real" tokens ("time," "flies," and "like" in the figure), although it becomes an issue when the input batch includes padded tokens. Because they exist just to pad the batch, they should be ignored when computing the total loss.

Tokenization for neural models

we covered the basic linguistic units (words, characters, and n-grams) and how to compute their embeddings. In this section, we will go deeper and focus on how to analyze texts and obtain these units—a process called tokenization. Neural network models pose a set of unique challenges on how to deal with tokens, and we'll cover some of the modern models to deal with these challenges.

Unknown words

A vocabulary is a set of tokens that an NLP model deals with. Many neural NLP models operate within a fixed, finite set of tokens. For example, when we built a sentiment analyzer in the pipeline first tokenized the training dataset and constructed a Vocabulary object that consists of all unique tokens that appeared more than, say, three times. The model then uses an embedding layer to convert tokens into word embeddings, which are some abstract representations of the input tokens.

But the number of all words in the world is not finite. We constantly make up new words that didn't exist before (I don't think people talked about "NLP" a hundred years ago). What if the model receives a word that it has never seen during training? Because the word is not part of the vocabulary, the model cannot even convert it to an index, let alone look up its embeddings. Such words are called out-of-vocabulary (OOV) words, and they are one of the biggest problems when building NLP applications.

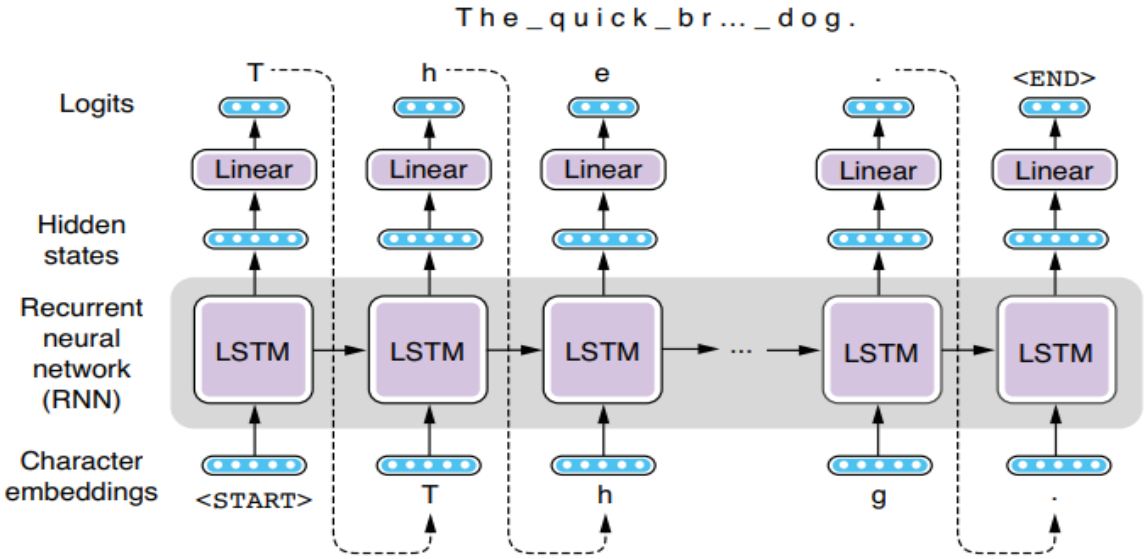
By far the most common (but not the best) way to deal with this problem is to represent all the OOV tokens as a special token, which is conventionally named UNK (for "unknown"). The idea is that every time the model sees a token that is not part of the vocabulary, it pretends it saw a special token UNK instead and proceeds as usual. This means that the vocabulary and the embedding table both have a designated "slot" for UNK so that the model can deal with words that it has never seen. The embeddings (and any other parameters) for UNK are trained in the same manner as other regular tokens. Do you see any problems with this approach? Treating all OOV tokens with a single UNK token means that they are collapsed into a single embedding vector. It doesn't matter if the word is "NLP" or "doggy"—as long as it's something unseen, it always gets treated as a UNK token and assigned the same vector, which becomes a generic, catch all representation of various words. Because of this, the model cannot tell the differences among OOV words, no matter what the identity of the words is.

This may be fine if you are building, for example, a sentiment analyzer. OOV words are, by definition, very rare and might not affect the prediction of most of the input sentences. However, this becomes a huge problem if you are building a machine translation system or a conversational engine. It wouldn't be a usable MT system or a chatbot if it produces "I don't know" every time it sees new words! In general, the OOV problem is more serious for language-generation systems (including machine translation and conversational AI) compared to NLP systems for prediction (sentiment analysis, POS tagging, and so on). How can we do better? OOV tokens are such a big problem in NLP that there has been a lot of research work on how to deal with them.character-based and subword-based models, two techniques commonly used for building robust neural NLP models.

Character models

The simplest yet effective solution for dealing with the OOV problem is to treat char□acters as tokens. Specifically, we break the input text into individual characters, even including punctuation and whitespace, and treat them as if they are regular tokens. The rest of the application is unchanged—"word" embeddings are assigned to characters, which are further processed by the model. If the model

produces text, it does so character-by-character. Instead of generating text word-by-word, the RNN produces text one character at a time, as illustrated in figure below. Thanks to this strategy, the model was able to produce words that look like English but actually aren't.



Character-based models are versatile and put few assumptions on the structure of the language. For languages with small sets of alphabets (like English), it effectively eradicates unknown words, because almost any words, no matter how rare they are, can be broken down into characters. Tokenizing into characters is also an effective strategy for languages with large alphabets (like Chinese), although you need to watch out for "unknown character" problems. However, this strategy is not without drawbacks. The biggest issue is its inefficiency. To encode a sentence, the network (be it an RNN or the Transformer) needs to go over all the characters in it. For example, a character-based model needs to process "t," "h," "e," and "_" (whitespace) to process a single word "the," whereas a word-based model can finish this in a single step. This inefficiency takes its biggest toll on the Transformers, where the attention computation increases quadratically when the input sequence gets longer

Subword models

are a recent invention that addresses this problem for neural networks. In subword models, the input text is segmented into a unit called subwords, which simply means something smaller than words. There is no formal linguistic definition as to what subwords actually are, but they roughly correspond to part of words that appear frequently. For example, one way to segment "dishwasher" is "dish + wash + er," although some other segmentation is possible.

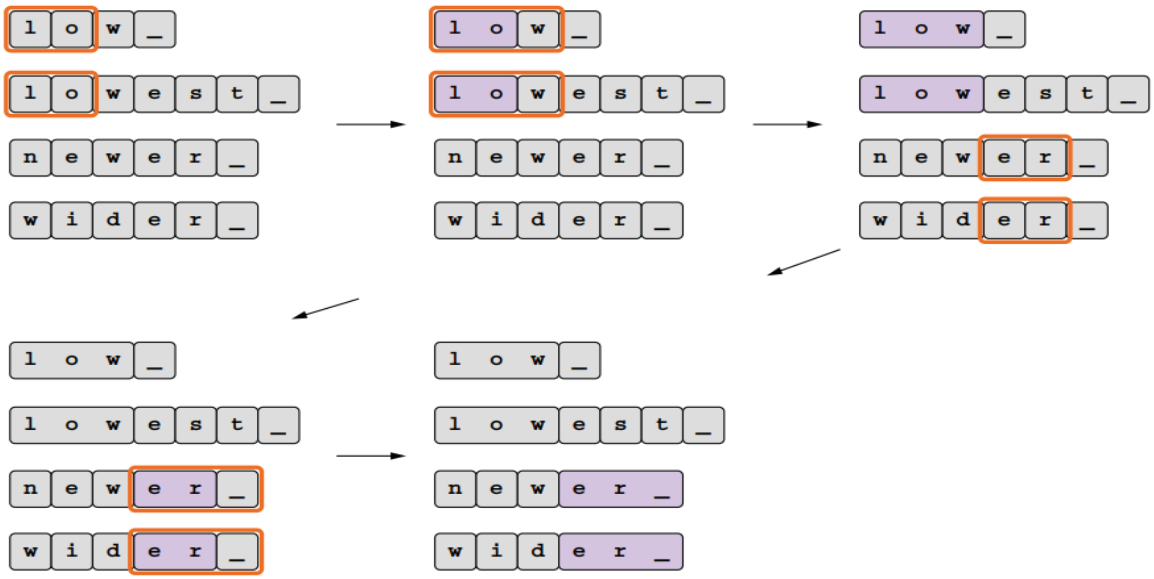
Tokenization - unknown words

- Tokenization that is both efficient and robust to unknown words?

- **Subword models** are a recent invention that addresses this problem for neural networks.
- In subword models, the input text is segmented into a unit called subwords, which simply means something smaller than words.
- There is **no formal linguistic definition** as to what subwords actually are, but they roughly correspond to part of words that appear frequently.
- For example, one way to segment "dishwasher" is "dish + wash + er," although some other segmentation is possible.
- Some varieties of algorithms (such as WordPiece and SentencePiece) tokenize input into subwords, but by far the most widely used is **byte-pair encoding (BPE**).

Byte-pair encoding (BPE)

- Byte-pair encoding (BPE) is a **compression algorithm**, used as a tokenization for neural models, particularly in machine translation.
- **The BPE is** to **keep frequent words** (such as "the" and "you"),**n-grams , unsegmented words (such as "-able" and "anti-"),** while **breaking up rarer words** (such as "dishwasher") into subwords ("dish + wash + er").
- Keeping frequent words and n-grams together helps the model process those tokens efficiently, whereas breaking up rare words ensures there are no UNK tokens, because everything can be ultimately broken up into individual characters.
- By flexibly choosing where to **tokenize based on the frequency**, BPE achieves the best of two worlds—being efficient while addressing the unknown word problem.
- Figure BPE learns subword units by iteratively merging consecutive units that cooccur frequently.
- Let's see how BPE determines where to tokenize with real examples. BPE is a purely statistical algorithm (it doesn't use any language-dependent information) and operates by merging the most frequently occurring pair of consecutive tokens, one at a time. First, BPE tokenizes all the input texts into individual characters. For example, if your input is four words low, lowest, newer, and wider, it will tokenize them into l o w _, l o w e s t _, n e w e r _, w i d e r _. Here, "_" is a special symbol that indicates the end of each word. Then, the algorithm identifies any two consecutive elements that appear most often. In this example, the pair l o appears most often (two times), so these two characters are merged, yielding lo w _, lo w e s t _, n e w e r _, w i d e r _. Then, lo w will be merged into low, e r into er, er _ into er_, at which time you have low _, low e s t _, n e w er_, w i d er_. This process is illustrated in figure below



Notice that, after four merge operations, lowest is segmented into low e s t where frequent substrings such as low are merged together whereas infrequent ones such as est are broken apart. To segment a new input (e.g., lower), the same sequence of merge operations is applied in order, yielding low e r _. If you start from 52 unique letters (26 upper- and lowercase letters), you will end up with 52 + N unique tokens in your vocabulary, where N is the number of merge operations executed. In this way, you have complete control over the size of the vocabulary.

Avoiding overfitting

- Overfitting is one of the most common and important issues when building any machine learning applications.
- An ML model is said to overfit when it fits the given data so well that it loses its generalization ability to unseen data.
- the model may capture the training data very well and show good performance on it, but it may not be able to capture its inherent patterns well and shows poor performance on data that the model has never seen before.
- To avoid overfitting a number of algorithms and techniques

➢ Regularization- L2 regularization (weight decay)

➢ Dropouts

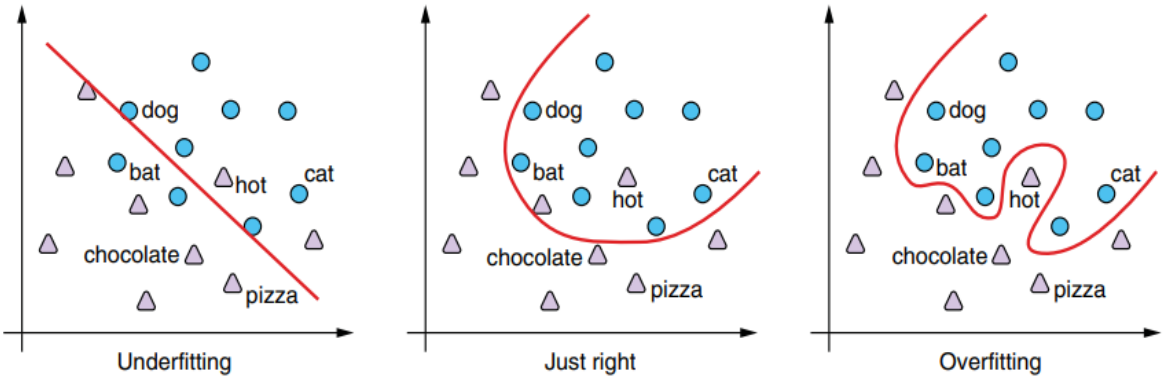
➢ and early stopping.

➢ Cross Validation

➢ Call Backs

These are popular in any ML applications (not just NLP) and worth getting under your belt.

Regularization



Regularization in machine learning refers to techniques that encourage the simplicity and the generalization of the model. You can think of it as one form of penalty you impose on your ML model to ensure that it is as generic as possible. What does it mean?

Say you are building an "animal classifier" by training word embeddings from a corpus and by drawing a line between animals and other stuff in this embedding space (i.e., you represent each word as a multidimensional vector and classify whether the word describes an animal based on the coordinates of the vector). Let's simplify this problem a lot and assume that each word is a two-dimensional vector, and you end up with the plot shown in figure above.

You can now visualize how a machine learning model makes a classification decision by drawing lines where the decision flips between different classes (animals and non-animals), which is called the classification boundary. How would you draw a classification boundary so that animals (blue circles) are separated from everything else (triangles)? One simple way to separate animals is to draw one

straight line, as in the first plot in figure 10.8. This simple classifier makes several mistakes (in classifying words like “hot” and “bat”), but it correctly classifies the majority of data points. This sounds like a good start.

What if you are told that the decision boundary doesn’t have to be a straight line? You may want to draw something like the one shown in the middle in figure above. This one looks better—it makes fewer mistakes than the first one, although it is still not perfect. It appears tractable for a machine learning model because the shape is simple. But there’s nothing that stops you here. If you want to make as few errors as possible, you can also draw something wiggly like the one shown in the third plot. That decision boundary doesn’t even make any classification errors, which means that we achieved 100% classification accuracy! Not so fast—remember that up until here, we’ve been thinking only about the training time, but the main purpose of machine learning models is to achieve good classification performance at the test time (i.e., they need to classify unobserved, new instances as correctly as possible). Now let’s think about how the three decision boundaries described earlier fare at test time. If we assume the test instances are distributed similarly to the training instances we saw in figure above, the new “animal” points are most likely to fall in the upper-right region of the plot. The first two decision boundaries will achieve decent accuracy by classifying the majority of new instances correctly. But how about the third one? Training instances such as “hot” shown in the plot are most likely exceptions rather than the norm, so the curved sections of the decision boundary that tried to accommodate as many training instances as possible may do more harm than good at the test time by inadvertently misclassifying test instances. This is exactly what overfitting looks like—the model fits the training data so well that it sacrifices its generalization ability, which is what’s happening here.

Then, the question is, how can we avoid having your model look like the third decision boundary? After all, it is doing a very good job correctly classifying the training data. If you looked only at the training accuracy and/or the loss, there would be nothing to stop you from choosing it. One way to avoid overfitting is to use a separate, held-out dataset (called a validation set) to validate the performance of your model. But can we do this even without using a separate dataset? The third decision boundary just doesn’t look right—it’s overly complex. With all other things being equal, we should prefer simpler models, because in general, simpler models generalize better. This is also in line with Occam’s razor, which states that a simpler solution is preferable to a more complex one. How can we balance between the training fit and the simplicity of the model? This is where regularization comes into play. Think of regularization as additional constraints imposed on the model so that simpler and/or more general models are preferred. The model is optimized so that it achieves the best training fit while being as generic as possible. Numerous regularization techniques have been proposed in machine learning because overfitting is such an important topic. We are going to introduce only a few of the most important ones—L2 regularization (weight decay), dropout, and early stopping

L2 REGULARIZATION

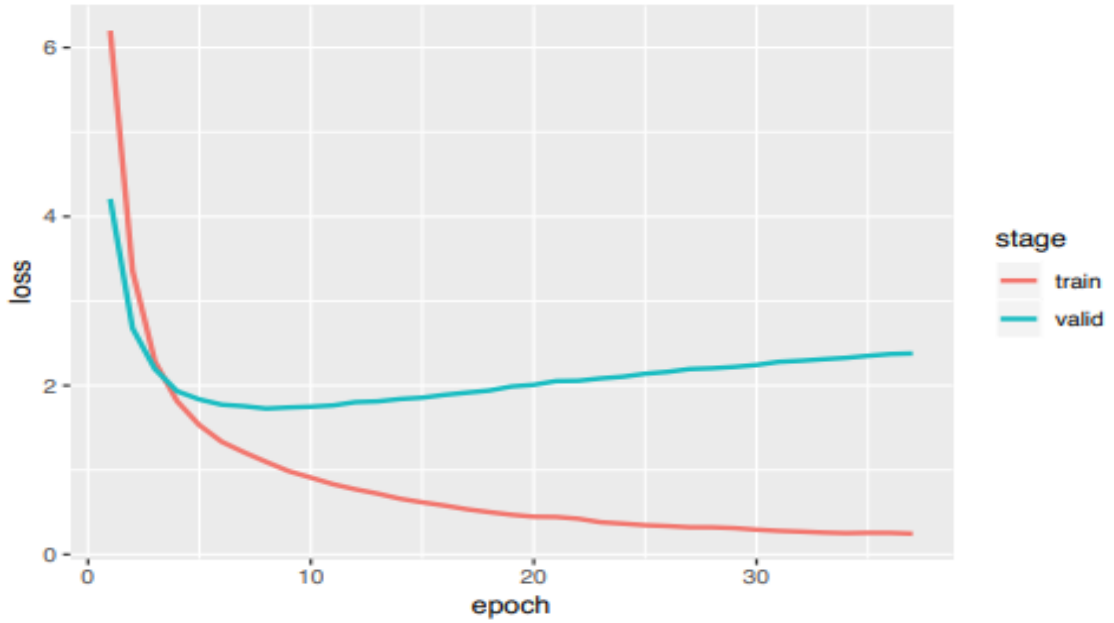
L2 regularization, also called weight decay, is one of the most common regularization methods not just for NLP or deep learning but for a wide range of ML models. We are not going into its mathematical details, but in short, L2 regularization adds a penalty for the complexity of a model measured by how large its parameters are. To represent a complex classification boundary, an ML model needs to adjust a large number of parameters (the “magic constants”) to extreme values, measured by the L2 loss, which captures how far away they are from zero. Such models incur a larger L2 penalty, which is why L2 encourages simpler models.

DROPOUT

- Dropout is another popular regularization technique commonly used with neural networks.
- Dropout works by randomly “dropping” neurons during training, where a “neuron” is basically a dimension of an intermediate layer .
- “dropping” means to mask it with zeros.

Early stopping

- Early stopping is a technique where you stop training your model when the model performance stops improving, usually measured by the validation set loss.
- Eg EnglishSpanish machine translation model the validation loss curve flattens out around the eighth epoch and starts to creep up after that, which is a sign of overfitting. Early stopping would detect this, stop the training, and use the result from the best epoch when the loss is lowest.
- early stopping has a “patience” parameter, which is the number of nonimproving epochs for early stopping to kick in. When patience is 10 epochs, for example, the training pipeline will wait 10 epochs after the loss stops improving to stop the training.

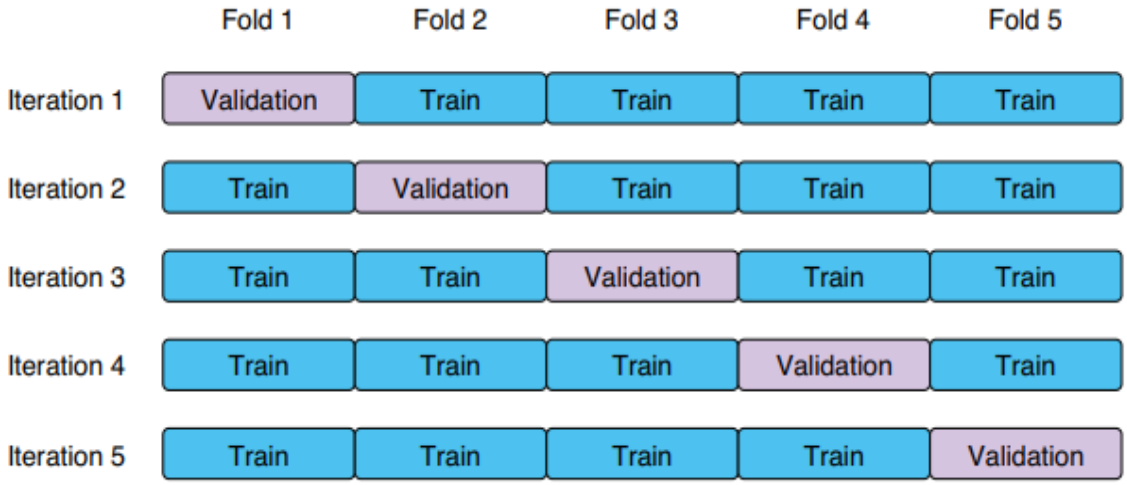


The validation loss curve flattens out around the eighth epoch and creeps back up.

Cross-validation

Cross-validation is not exactly a regularization method, but it is one of the techniques commonly used in machine learning. A common situation in building and validating a machine learning model is this—you have only a couple of hundred instances available for training. As we’ve seen so far in this book, you can’t train a reliable ML model just on the training set—you need a separate set for validation, and preferably another separate set for testing. How much you use for validation/testing depends on the task and the data size, but in general, it is advised that you set aside 5–20% of your training instances for validation and testing. This means that if your training data is small, your model is validated and

tested on just a few dozen instances, which can make the estimated metrics unstable. Also, how you choose these instances has a large impact on the evaluation metrics, which is not ideal. The basic idea of cross-validation is to iterate this phase (splitting the dataset into training and validation portions) multiple times with different splits to improve the stability of the result. Specifically, in a typical setting called k-fold cross validation, you first split the dataset into k different portions of equal size called folds. You use one of the folds for validation while training the model on the rest (k – 1 folds), and repeat this process k times, using a different fold for validation every time. See figure below for an illustration



k-fold cross validation, the dataset is split into k equally sized folds and one is used for validation.

How to deal with Imbalanced Datasets

Dealing with imbalanced datasets in NLP models is crucial to ensure that the model performs well across all classes, not just the majority class. Here are some strategies to handle imbalanced datasets:

- Evaluation Metrics (precision, recall, F1score, ROC curve…)
- sampling Techniques (upsampling,downsampling..)
- Algorithmic Techniques (class weights….)
- Data Augmentation (Back translation, Paraprasing)
- Ensemble Methods (Bagging, Boosting…)

By applying these techniques, you can mitigate the impact of class imbalance and build more robust and fair NLP models.

- To encounter the class imbalance problem in building NLP and ML models
- Eg: The goal of a classification task is to assign one of the classes (e.g., spam or nonspam). In document classification, some topics (such as politics or sports) are usually more popular than other topics.
- when some classes have way more instances than others is called imbalanced .
- techniques used to balance the dataset.
- Using appropriate evaluation metrics- F1-measure instead of accuracy
- Upsampling and downsampling
- Weighting losses

1. Using appropriate evaluation metrics

Before you even begin tweaking your dataset or your model, make sure you are validating your model with an appropriate metric. In one extreme case, if 90% of your instances belong to class A and the other 10% belong to class B, even a stupid classifier that assigns class A to everything can achieve 90% accuracy. This is called a majority class baseline.

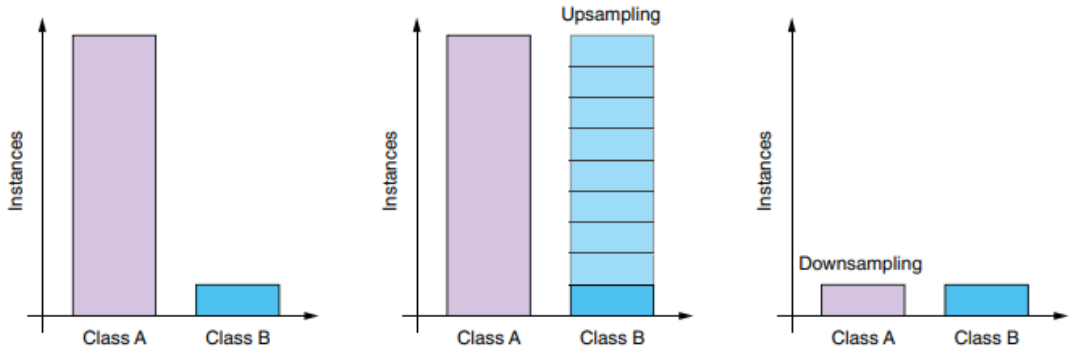
A slightly more clever (but still stupid) classifier that randomly assigns label A 90% of the time and label B 10% of the time without even looking at the instance will achieve 0.9 * 0.9 + 0.1 * 0.1 = 82% accuracy. This is called a random baseline, and the more imbalanced your dataset is, the higher the accuracy of these baseline models will become. But this kind of random baseline is rarely a good model for minority classes. Imagine what would happen to class B if you used the random baseline. Because it will assign class A 90% of the time no matter what, 90% of the instances belonging to class B will be assigned class A.

In other words, the accuracy of this random baseline for class B is only 10%. If this was a spam filter, it would let 90% of spam emails go through, no matter what the content is, just because 90% of emails you receive are not spam! This would make a terrible spam filter. If your dataset is imbalanced and you care about the classification performance on the minority class, you should consider using metrics that are more appropriate for such settings. For example, if your

task is a “needle in a haystack” type of setting, where the goal is to find a very small number of instances among others, you may want to use the F1-measure instead of accuracy. Because the F1-measure is calculated per class, it does not underrepresent minority classes. If you’d like to measure the model’s overall performance including majority classes, you can compute the macro-averaged F-measure, which is simply an arithmetic average of F-measures computed per class.

2. Upsampling and downsampling

Now let’s look at concrete techniques that can mitigate the class imbalance problem. First of all, if you can collect more labeled training data, you should seriously consider doing that first. Unlike academic and ML competition settings where the dataset is fixed while you tweak your model, in a real-world setting you are free to do whatever is necessary to improve your model (of course, as long as it’s lawful and practical). Often, the best thing you can do to improve a model’s generalization is expose it to more data. If your dataset is imbalanced and the model is making biased predictions, you can either upsample or downsample your data so that classes have roughly equal representations. In upsampling (see the second figure in figure below), you artificially increase the size of the minority class by copying the instances multiple times. Take the scenario we discussed earlier for example—if you duplicate the instances of class B and add eight extra copies of each instance to the dataset, they have an equal number of instances.

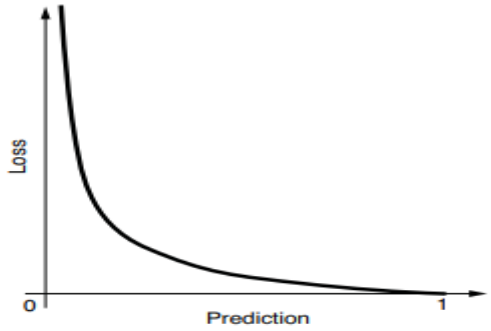


This can mitigate the biased prediction issue. More sophisticated data augmentation algorithms such as SMOTE5 are available, although they are not widely used in NLP, due to the inherent difficulty in generating linguistic examples artificially. If your model is biased not because the minority class is too small but because the majority class is too large, you can instead choose to downsample (the third figure in figure above). In downsampling, you artificially decrease the size of the majority class by choosing a subset of the instances belonging to that class. For example, if you sample one out of nine instances from class A, you’ll end up with the equal number of instances in classes A and B. You can downsample in multiple ways—the easiest is to randomly choose the subset. If you would like to make sure that the downsampled dataset still preserves the diversity in the original data, you can try stratified sampling, where you sample some number of instances per group defined by some attributes. For example, if you have too many nonspam emails and want to downsample, you

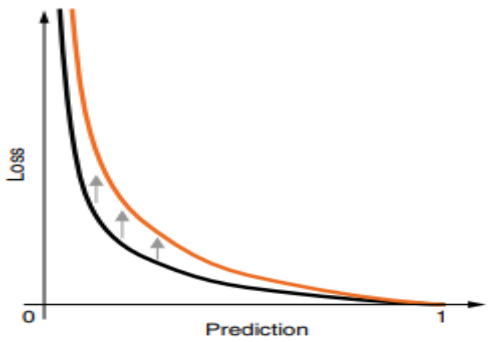
can group them by the sender's domain first, then sample a fixed number of emails per domain. This will ensure that your sampled dataset will contain a diverse set of domains.

3. Weighting losses

Another approach for mitigating the class imbalance problem is to use weighting when computing the loss, instead of making modification to your training data. Remember that the loss function is used to measure how "off" the model's prediction for an instance is compared against the ground truth. When you measure how bad the model's prediction is, you can tweak the loss so that it penalizes more when the ground truth belongs to the minority class. Let's take a look at a concrete example. The binary cross-entropy loss, a common loss function used for training a binary classifier, looks like the curve shown in figure below, when the correct label is 1.



The x-axis is the predicted probability of the target class, and the y-axis is the amount of loss the prediction will incur. When the prediction is perfectly correct (probability = 1), there's no penalty, whereas as the prediction gets worse (probability < 1), the loss goes up. If you care more about the model's performance on the minority class, you can tweak this loss. Specifically, you can change the shape of this loss (by simply multiplying it by a constant number) just for that class so that the model incurs a larger loss when it makes mistakes on the minority class. One such tweaked loss curve is shown in the figure below as the top curve. This weighting has the same effect as upsampling the minority class, although modifying the loss is computationally cheaper because you don't need to actually increase the amount of training data.



Hyperparameters

Hyperparameters are parameters about the model and the training algorithm. This term is used in contrast with parameters, which are numbers that are used by the model to make predictions from the input. Correctly tuning hyperparameters is critical for many machine learning models to work properly and achieve their highest potential, and ML practitioners spend a lot of time tuning hyperparameters. Knowing how to tune hyperparameters effectively has a huge impact on your productivity in building NLP and ML systems.

Examples of hyperparameters

Hyperparameters are "meta"-level parameters—unlike model parameters, they are used not to make predictions but for controlling the structure of the model and how the model is trained. For example, if you are working on word embeddings or an RNN, how many hidden units (dimensions) to use for representing words is one important hyperparameter. The number of RNN layers to use is another hyperparameter.

In addition to these two hyperparameters (the number of hidden units and layers), the Transformer which has a number of other parameters, such as the number of attention heads and the dimension of the feedforward network. Even the type of architecture you use, such as RNN versus Transformer, can be thought of as one hyperparameter.

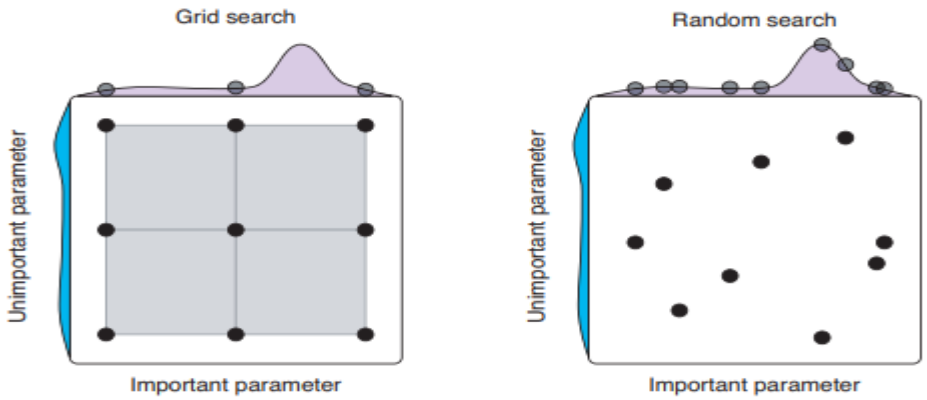
Besides, the optimization algorithm you use may have hyperparameters, too. For example, the learning rate one of the most important hyperparameters in many ML settings, determines how much to tweak the model parameters per optimization step. The number of epochs (iterations through the training dataset) is also an important hyperparameter, too.

Hyperparameters can have a huge impact on the performance of machine learning models. Many ML practitioners tune hyperparameters by hand. This means that you start from a set of hyperparameters that look reasonable and measure the model's performance on a validation set. Then you change one or more of the hyperparameters slightly and measure the performance again. You repeat this process several times until you hit the "plateau," where any change of hyperparameters provides only a marginal improvement. One issue with this manual tuning approach is that it is slow and arbitrary.

Grid search vs. random search

We understand that manual optimization of hyperparameters is inefficient, but how should we go about optimizing them, then? We have two more-organized ways of tuning hyperparameters—grid search and random search.

In grid search, you simply try every possible combination of the hyperparameter values you want to optimize. For example, let's assume your model has just two hyperparameters—the number of RNN layers and the embedding dimension. You first define reasonable ranges for these two hyperparameters, for example, [1, 2, 3] for the number of layers and [128, 256, 512] for the dimensionality. Then grid search measures the model's validation performance for every combination—(1, 128), (1, 256), (1, 512), (2, 128), . . . , (3, 512)—and simply picks the best-performing combination. If you plot these combinations on a 2-D plot, it looks like a grid (see the illustration in figure below), which is why this is called grid search. Grid search is a simple and intuitive way to optimize the hyperparameters. However, if you have many hyperparameters and/or their ranges are large, this method gets out of hand. The number of possible combinations is exponential, which makes it impossible to explore all of them in a reasonable amount of time.



A better alternative to grid search is random search. In random search, instead of trying every possible combination of hyperparameter values, you randomly sample the val□ues and measure the model's performance on a specified number of combinations (which are called trials). For example, in the previous example, random search may choose (2, 87), (1, 339), (2, 101), (3, 254), and so on until it hits the specified number of trials. See the illustration in figure above.

Hyperparameter tuning with Optuna plug in

- How should you go about implementing it in practice?

- You can always write your own for-loop (or “for-loops,” in the case of grid search), although it would quickly get tiring if you need to write this type of boilerplate code for every model and task you work on.
- Hyperparameter optimization is such a universal topic that many ML researchers and engineers have been working on better algorithms and software libraries.
- For example, AllenNLP has its own library called Allentune (https://github.com/allenai/allentune) that you can easily integrate with your AllenNLP training pipeline.
- Another hyperparameter tuning library called Optuna (https://optuna.org/) and show how to use it with AllenNLP to optimize your hyperparameters.
- Optuna implements state-of-the-art algorithms that search for optimal hyperparameters efficiently and provides integration with a wide range of machine learning frameworks, including TensorFlow, PyTorch, and AllenNLP.
- First, we assume that you have installed AllenNLP (1.0.0+) and the Optuna plugin for AllenNLP. These can be installed by running the following:

pip install allennlp

pip install allennlp_optuna

Summary

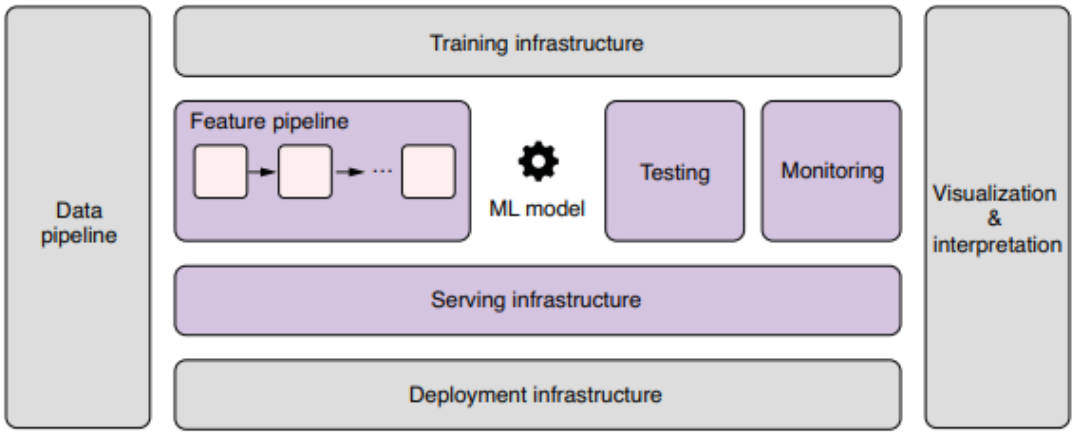
- Instances are sorted, padded, and batched together for more efficient computation.
- Subword tokenization algorithms such as BPE split words into units smaller than words to mitigate the out-of-vocabulary problem in neural network models.
- Regularization (such as L2 and dropout) is a technique to encourage model simplicity and generalizability in machine learning.
- data upsampling, downsampling, or loss weights for addressing the data imbalance issue.
- Hyperparameters are parameters about the model or the training algorithm. They can be optimized using manual, grid, or random search.

Deploying and Serving NLP Applications

Architecting and Deploying NLP models

Introduction

- Model deployment is the process of integrating your model into an existing production environment. The model will receive input and predict an output for decision-making for a specific use case.
- *"Only when a model is fully integrated with the business systems, we can extract real value from its predictions". — Christopher Samiullah*
- For example, a model can be deployed in an e-commerce site and it can predict if a review about a specific product is **positive** or **negative**.
- As shown in figure below, only a tiny fraction of a typical real-world ML system is the ML code, but the "ML code" part is supported by numerous components that provide various functionalities, including data collection, feature extraction, and serving.



Ex: nuclear power plant

In operating a nuclear power plant, only a tiny fraction concerns nuclear reaction.

- Everything else is a vast and complex infrastructure that supports safe and efficient generation and transportation of materials and electricity—
- how to use the generated heat to turn the turbine to make electricity?
- how to cool and circulate water safely?

- how to transmit the electricity efficiently? and so on.

All those supporting infrastructures have little to do with nuclear physics

1. Architecting your NLP application

- There are different ways you can deploy your NLP model into production, like using Flask, Django, streamlit,FastAPI or other frameworks.
- Lets discuss some best practices specific to designing and building NLP/ML applications.

1.1 Before machine learning

- Check whether you need ML for your product before you start working on your NLP application.
- Doing this offers many benefits—
- first, by solving the task manually, you get a feel of what's important when it comes to solving the problem and whether it's something that a machine can really solve automatically.
- Second, by putting yourself in the machine's shoes, you gain a lot of insights into the task (what the data looks like, how the input and output data are distributed, and how they are related), which become valuable when it comes to actually designing an ML system to solve it.

1.2 Choosing the right architecture

- An NLP application can take many forms. many NLP components can be structured as a one-off task that takes some static data as its input and produces transformed data as its output.
- For example, if you have a static database of some documents and you'd like to classify them by their topics, your NLP classifier can be a simple one-off Python script that runs this classification task. If you'd like to extract common entities (e.g., company names) from the same database, you can write a Python script that runs a named entity recognition (NER) model to do it.
- NLP components can also be designed so that they serve real-time prediction.
- This is necessary when, for example, an audience needs real-time subtitles for a speech.
- Another example is when the system wants to show ads based on the user's real-time behavior.

- For these cases, the NLP service needs to receive a stream of input data (such as audio or user events) and produce another stream of data (such as transcribed text or ad-click probabilities).
- Real-time streaming frameworks such as Apache Flink (https://flink.apache.org/) are often used for processing such stream data.
- Also, if your application is based on a server-client architecture, as with typical mobile and web apps, and you want to show some real-time prediction to the users, you can choose to run ML/NLP models on the client side, such as the web browser or the smartphones. Client-side ML frameworks such as TensorFlow.js (https://www.tensorflow.org/js), Core ML (https://developer.apple.com/documentation/coreml), and ML Kit (https://developers.google.com/ml-kit) can be used for such purposes.

1.3 Project structure

- A typical NLP project may need to manage datasets to train a model from, intermediate files generated by preprocessing data, model files produced as a result of training, source code for training and inference, and log files that store additional information about the training and inference.
- Because typical NLP applications have many components and directories in common, it'd be useful if you simply follow best practices as your default choice when starting a new project.

Here are few recommendations for structuring your NLP projects

- Data management
- Virtual environment
- Experiment management
- Source code

<u>Data management</u>

- Make a directory called data and put all the data in it.
- It may also be helpful to subdivide this into raw, interim, and result directories.

- The raw directory contains the unprocessed dataset files you obtained externally (such as the Stanford Sentiment Treebank we've been using throughout this book) or built internally. It is very critical that you do not modify any files in this raw directory by hand. If you need to make changes, write a script that runs some processing against the raw files and then writes the result to the interim directory, which serves as a place for intermediate results. Or make a patch file that manages the "diff" you made to the raw file, and version-control the patch files instead.
- The final results such as predictions and metrics should be stored in the result directory.

Virtual environment—It is strongly recommended that you work in a virtual environment so that your dependencies are separated and reproducible.

- You can use tools like Conda (https://docs.conda.io/en/latest/) (my recommendation) and venv (https://docs.python.org/3/library/venv.html) to set up a separate environment for your project and use pip to install individual packages. Conda can export the environment configuration into an environment.yml file, which you can use to recover the exact Conda environment. You can also keep track of pip packages for the project in a requirements.txt file.
- Even better, you can use Docker containers to manage and package the entire ML environment. This greatly reduces dependency-related issues and simplifies deployment and serving.

Experiment management

- Training and inference pipelines for an NLP application usually consist of several steps, such as preprocessing and joining the data, converting them into features, training and running the model, and converting the results back to a human-readable format.
- These steps can easily get out of hand if you try to remember to manage them manually. A good practice is to keep track of the steps for the pipeline in a shell script file so that the experiments are reproducible with a single command, or use dependency management software such as GNU Make, Luigi (https://github.com/spotify/luigi), and Apache Airflow (https://airflow.apache.org/).

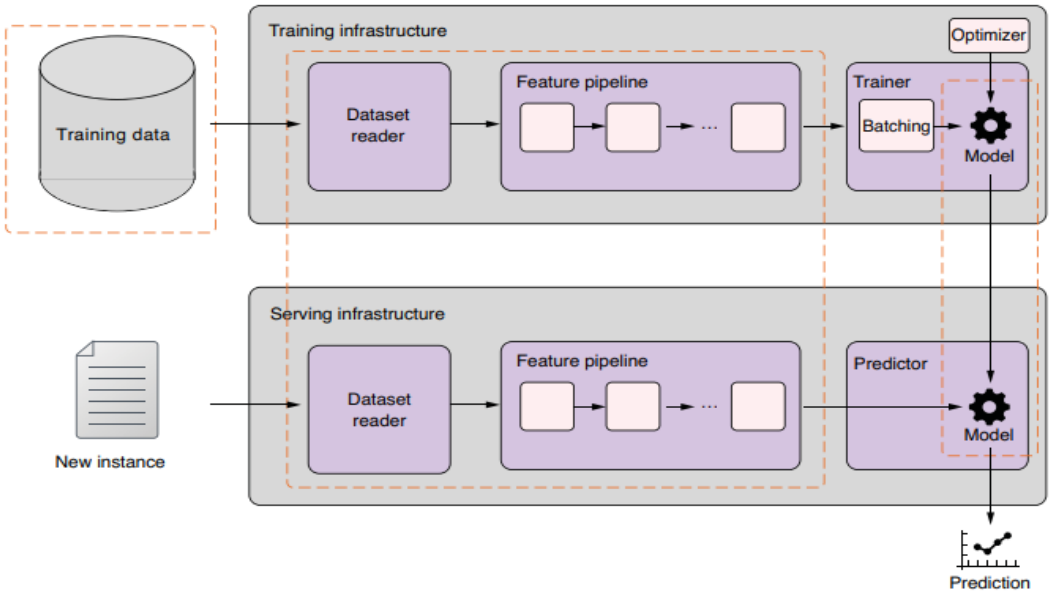
Source code

- Python source code is usually put in a directory of the same name as the project, which is further subdivided into directories such as data (for data-processing code), model (for model code), and scripts (for putting scripts for training and other one-off tasks)

1.4 Version control

- Tools like Git help you keep track of the changes and manage different versions of the source code.
- In addition to version-controlling your source code, it is also important to version control your data and models. This means that you should version-control your training data, source code, and models separately.

  This is one of the major differences between regular software projects and ML applications. Machine learning is about improving computer algorithms through data. By definition, the behavior of any ML system depends on data it is fed. This could lead to a situation where the behavior of the system is different even if you use the same code



- Tools like Git Large File Storage (https://git-lfs.github.com/) and DVC (https:// dvc.org) can version-control your data and models in a seamless way.

When keeping track of experiments, be sure to record the following information for each experiment:

- Versions of the model code, feature pipeline, and the training data used Hyperparameters used to train the model
- Evaluation metrics for the training and the validation data

2. Deploying your NLP model

- Deployment stage is where your NLP application is put on a server and becomes available for use.
- We'll discuss practical considerations when deploying NLP/ML applications.

2.1 Testing

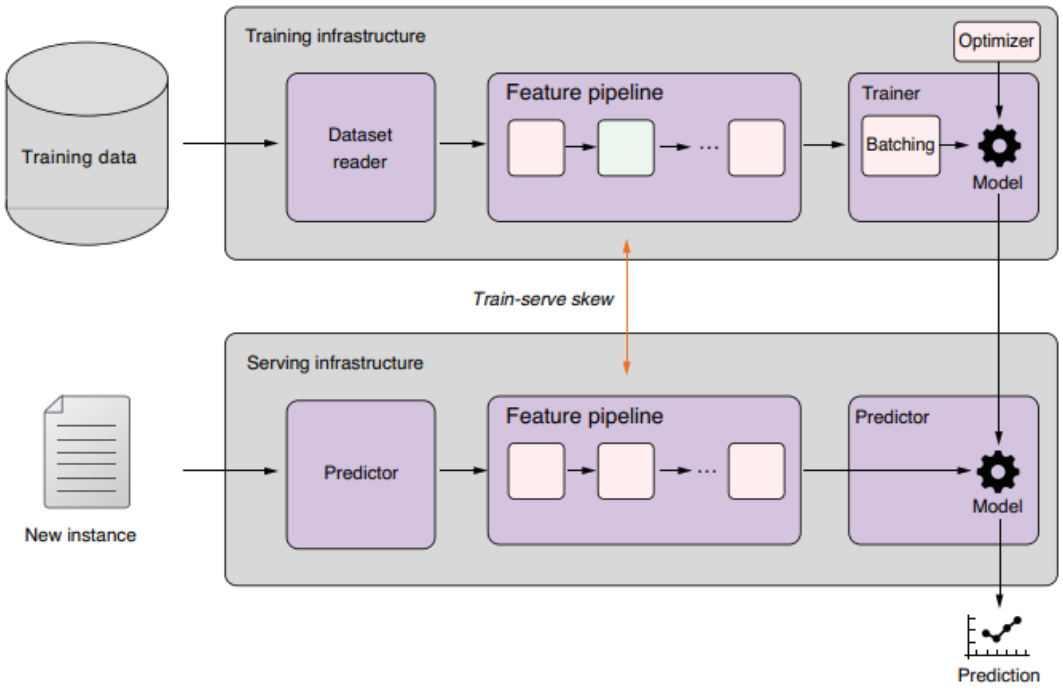
- Technique for testing NLP/ML models is sanity checks against the model output. You can start with a small and simple model and just a few toy instances with obvious labels.

If you are testing a sentiment analysis model, for example, this goes as follows:

- Create a small and simple model for debugging, such as a toy encoder that simply averages the input word embeddings with a softmax layer on top.
- Prepare a few toy instances, such as "The best movie ever!" (positive) and "This is an awful movie!" (negative).
- Feed these instances to the model, and train it until convergence. Because we are using a very small dataset without a validation set, the model will heavily overfit to the instances, and that's totally fine. Check whether the training loss goes down as expected.
- Feed the same instances to the trained model, and check whether the predicted labels match the expected ones.
- Try the steps above with more toy instances and a larger model

2.2 Train-serve skew

- One common source of errors in ML applications is called train-serve skew, a situation where there's a discrepancy between how instances are processed at the training and the inference times.

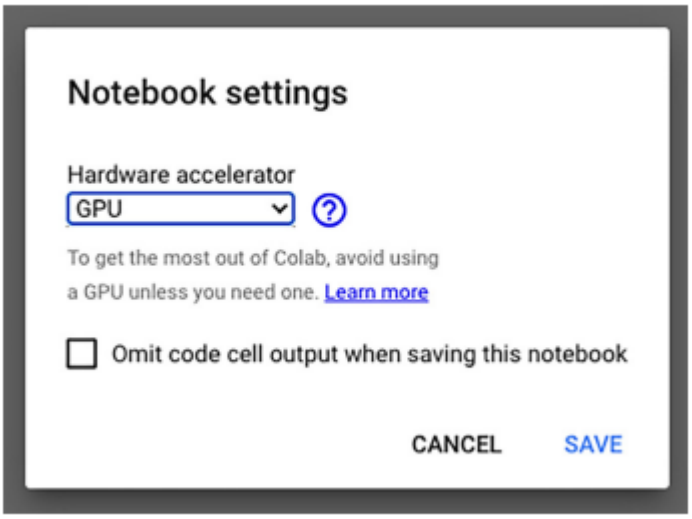


2.3 Monitoring

- In addition to the usual server metrics (e.g., CPU and memory usage), you should also monitor metrics related to the input and the output of the model. Specifically, you can monitor some higher-level statistics such as the distribution of input values and output labels.
- As mentioned earlier, logic errors, which are a type of error that causes the model to produce wrong results without crashing it, are the most common and hardest to find in ML systems.
- Monitoring those high-level statistics makes it easier to find them. Libraries and platforms like PyTorch Serve and Amazon SageMaker support monitoring by default.

2.4 Using GPUs

- Training large modern ML models almost always requires hardware accelerators such as GPUs.
- If you don't own GPUs or have never used cloud-based GPU solutions before, the easiest way to "try" GPUs for free is to use Google Colab. Go to its URL (https://colab.research.google.com/), create a new notebook, go to the Runtime menu, and choose "Change runtime type." This will bring up the dialog box shown in figure below.


Notebook settings
Hardware accelerator
GPU
To get the most out of Colab, avoid using
a GPU unless you need one. Learn more
Omit code cell output when saving this notebook
CANCEL
SAVE

Deploying and Serving NLP Applications

Case study: Serving and deloying NLP applications- Interpreting and visualizing model predictions

- Deploy an NLP model built with Hugging Face. Specifically, we'll take a pretrained language generation model (DistilGPT2) or your choice, serve it with TorchServe, and deploy it to a cloud server using Amazon SageMaker.

1. Serving models with TorchServe

- Deploying an NLP application is more than just writing an API for your ML model.
- we need to take care of a number of production-related considerations, including how to deal with high traffic by parallelizing model inference with multiple workers, how to store and manage different versions of multiple ML models, how to consistently handle pre- and postprocessing of the data, and how to monitor the health of the server as well as various metrics about the data**.**
- You can use TorchServe (https://github.com/pytorch/serve), an easy-to-use framework for serving PyTorch models jointly developed by Facebook and Amazon.
- TorchServe is shipped with many functionalities that can address the issues mentioned earlier.

2. Deploying models with SageMaker

- Amazon SageMaker is a managed platform for training and deploying machine learning models.
- It enables you to spin up a GPU server, run a Jupyter Notebook inside it, build and train ML models there, and directly deploy them in a hosted environment.
- Our next step is to deploy the machine learning model as a cloud SageMaker endpoint so that production systems can make requests to it.

The concrete steps for deploying an ML model with SageMaker consist of the following:

- 1 Upload your model to S3.
- 2 Register and upload your inference code to Amazon Elastic Container Registry (ECR). 3 Create a SageMaker model and an endpoint.
- 4 Make requests to the endpoint.

3. Interpreting and visualizing model predictions

- we use the Language Interpretability Tool (LIT) (https://pair-code.github.io/lit/) for visualizing and interpreting the predictions and behavior of NLP models.
- LIT is an open source toolkit developed by Google and offers a browser-based interface for interpreting and visualizing ML predictions.
- It is framework agnostic, meaning that it works with any Python-based ML frameworks of choice, including AllenNLP and Hugging Face Transformers.
- LIT offers a wide range of features, including the following:
- Saliency map—Visualizing in color which part of the input played an important role to reach the current prediction
- Aggregate statistics—Showing aggregate statistics such as dataset metrics and confusion matrices
- Counterfactuals—Observing how model predictions change for generated new examples

Summary

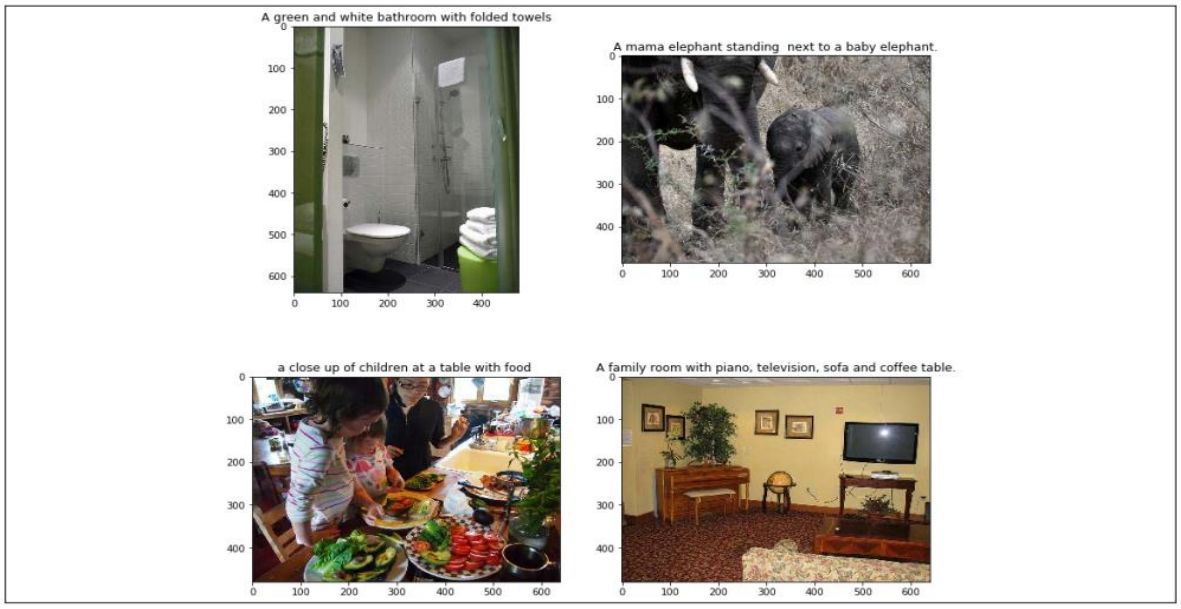
- Machine learning code is usually a small portion in real-world NLP/ML systems, supported by a complex infrastructure for data collection, feature extraction, and model serving and monitoring.
- NLP modules can be developed as a one-off script, a batch prediction service, or a real-time prediction service.
- It is important to version-control your model and data, in addition to the source code. Beware of train-serve skew that causes discrepancies between the training and the testing times.
- You can easily serve PyTorch models with TorchServe and deploy them to Amazon SageMaker.
- Explainable AI is a new field for explaining and interpreting ML models and their predictions. You can use LIT (Language Interpretability Tool) to visualize and interpret model predictions.

Multimodal Deep learning

- The dictionary definition of "modality" states that it is "a particular mode in which something exists or is experienced or expressed.
- " Sensory modalities, like touch, taste, smell, vision, and sound, allow humans to experience the world around them.
- Speech recognition models using lip-reading information were developed in the field of Audio-Visual Speech Recognition (AVSR).
- There are several exciting applications of multi-modal deep learning models in medical devices and diagnosis, learning technology, and other Artificial Intelligence (AI) areas

Vision and language tasks

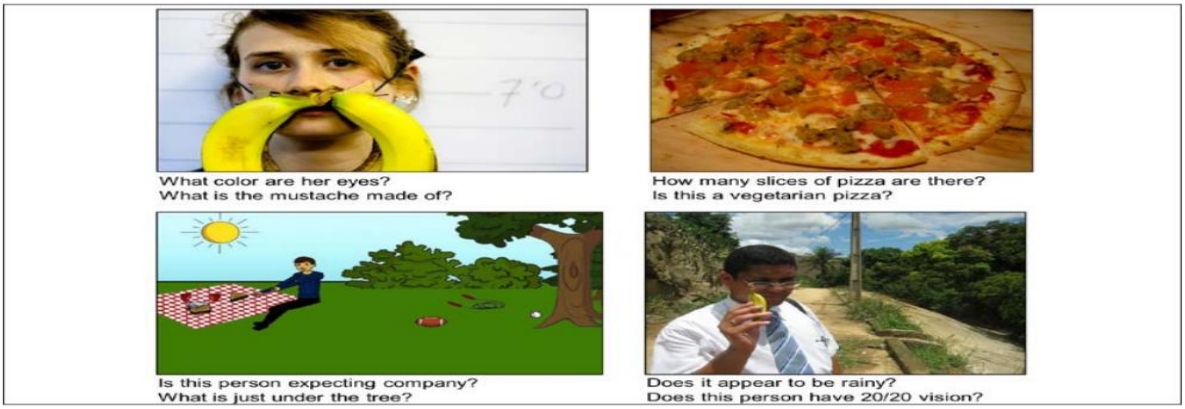
- A combination of Computer Vision (CV) and Natural Language Processing (NLP) allows us to build smart AI systems that can see and talk.
- CV and NLP together produce interesting tasks for model development. Taking an image and generating a caption for it is a well-known task. A practical application of this task is generating alt-text tags for images on web pages.
- Visually impaired readers use screen readers, which can read these tags while reading the page, improving the accessibility of web pages.
- Other topics in this area include video captioning and storytelling – composing a story from a sequence of images.
- The following image shows some examples of images and captions. Our primary focus in this chapter is on image captioning



Visual Question Answering (VQA) is the challenging task of answering questions about objects in the image. The following image shows some examples from the VQA dataset.

Compared to image captioning, where prominent objects are reflected in the caption, VQA is a more complex task. Answering the question may also require some reasoning.

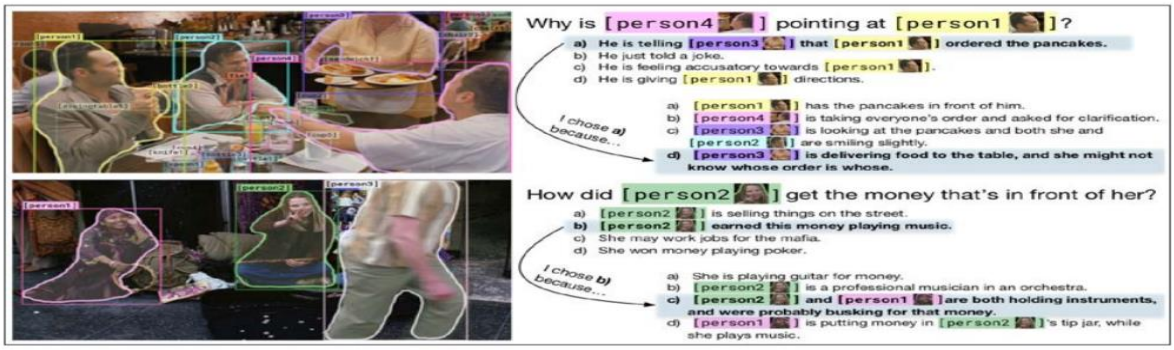
Consider the bottom-right panel in the following image. Answering the question, "Does this person have 20/20 vision?" requires reasoning. Datasets for VQA are available at visualqa.org



Reasoning leads to another challenging but fascinating task – **Visual Commonsense Reasoning (VCR).**

When we look at an image, we can guess emotions, actions, and frame a hypothesis of what is happening. Such a task is quite easy for people and may even happen without conscious effort. The aim of the VCR task is to build models that can perform such a task. These models should

also be able to explain or choose an appropriate reason for the logical inference that's been made. The following image shows an example from the VCR dataset.

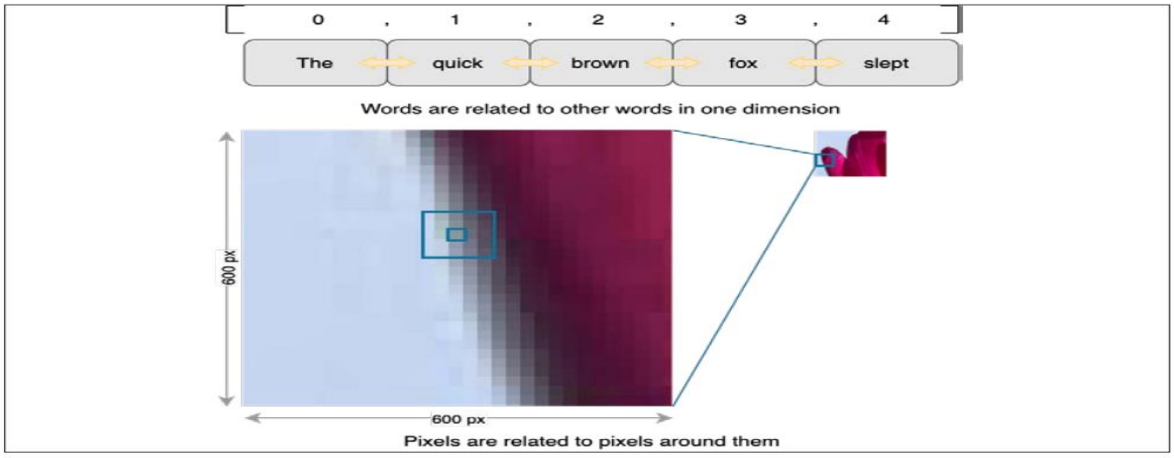


- A critical concept in this area is **visual grounding**.
- Grounding enables tying concepts in language to the real world. Simply put, it matches words to objects in a picture. By combining vision and language, we can ground concepts from languages to parts of an image.
- There can be more abstract concepts that can be grounded. For example, a short elephant and a short person have different measurements. Grounding provides us with a way to see what models are learning and helps us guide them in the right direction

Image captioning

- Image captioning is all about describing the contents of an image in a sentence. Captions can help in content-based image retrieval and visual search.
- In text summarization, the input is a sequence of the long-form article, and the output is a short sequence summarizing the content.
- In image captioning, the output is similar in format to summarization. However, it may not be obvious how to structure an image that consists of pixels as a sequence of embeddings to be fed into the Encoder.
- A naïve solution for representing images as a sequence could be expressing them as a list of pixels.
- So, an image of size 28x28 pixels becomes a sequence of 784 tokens.

- When the tokens represent text, an Embedding layer learns the representation of each token. If this Embedding layer had a dimension of 64, then each token would be represented by a 64-dimensional vector.
- This embedding vector was learned during training.
- Extending our analogy of using a pixel as a token, a straightforward solution is to use the value of the Red/Green/Blue channels of the pixel in an image to generate a three-dimensional embedding.
- However, training these three dimensions does not sound like a logical approach. More importantly, pixels are laid out in a 2D representation, while the text is laid out in a 1D representation.
- This concept is illustrated in the following image. Words are related to words next to each other. When pixels are laid out in a sequence, the data locality of these pixels is broken since the content of a pixel is related to the pixels all around it, not just to the left and right of it.
- This idea is shown by the following super zoomed in image of a tulip:



Data locality and translation invariance are two critical properties of images. Translation invariance is the idea that an object can appear in various spots in an image. In a fully connected model, the model would try to learn the position of the object, which would prevent the model from generalizing. The specialized architecture of Convolutional Neural Networks (CNNs) can be used to exploit these properties and extract signals from the image. At a high level, we use CNNs, specifically the ResNet50 architecture, to convert the image into a tensor that can be fed to a seq2seq architecture.

Our model will combine the best of CNNs and RNNs to handle the image and text parts under the seq2seq model. The following diagram shows our architecture at a very high level:

The main steps of building our model are summarized here:

1. Downloading the data: Given the large size of the dataset, this is a timeconsuming activity.
2. Pre-processing captions: Since the captions are in JSON format, they are flattened into a CSV for easier processing.
3. Feature extraction: We pass the image files through ResNet50 to extract features and save them to speed up training.
4. Transformer training: A full Transformer model with positional encoding, multi-head attention, an Encoder, and a Decoder is trained on the processed data.
5. Inference: Use the trained model to caption some images!
6. Evaluating performance: Bilingual Evaluation Understudy (BLEU) scores are used to compare the trained models with ground truth data

MS-COCO dataset for image captioning

Microsoft published the Common Objects in Context or COCO dataset in 2014. All the versions of the dataset can be found at cocodataset.org.

The COCO dataset is a big dataset that's used for object detection, segmentation, and captioning, among other annotations.

Features

1. **Images**:
    1. Over 330,000 images.
    2. Images cover 91 object types that are commonly found in daily life.
    3. Each image has annotations that describe the objects within it.
2. **Annotations**:

1. **Object Detection**: Bounding boxes around objects.
2. **Segmentation**: Object instance masks.
3. **Keypoints**: Locations of keypoints for humans (e.g., joints in a human body).
4. **Captions**: Each image comes with five different captions that describe the scene. These captions are natural language sentences that provide a description of the content and context of the image.

Structure of the Dataset

- **Training Set**: Contains the majority of the images and annotations used to train models.
- **Validation Set**: Used for model validation during training.
- **Test Set**: Contains images without annotations, used for evaluating model performance on unseen data.

Download and Usage

- **Website**: MS-COCO Official Site
- **Data Download**: The dataset can be downloaded from the official site, and it is provided in different formats:
    - Images only.
    - Annotations only.
    - Combined packages.
- **APIs and Tools**: MS-COCO provides an API for Python that can be used to access and manipulate the dataset easily.

Image processing with CNNs and ResNet50

In the world of deep learning, specific architectures have been developed to handle specific modalities. CNNs have been incredibly successful in processing images and are the standard architecture for CV tasks.

CNNs

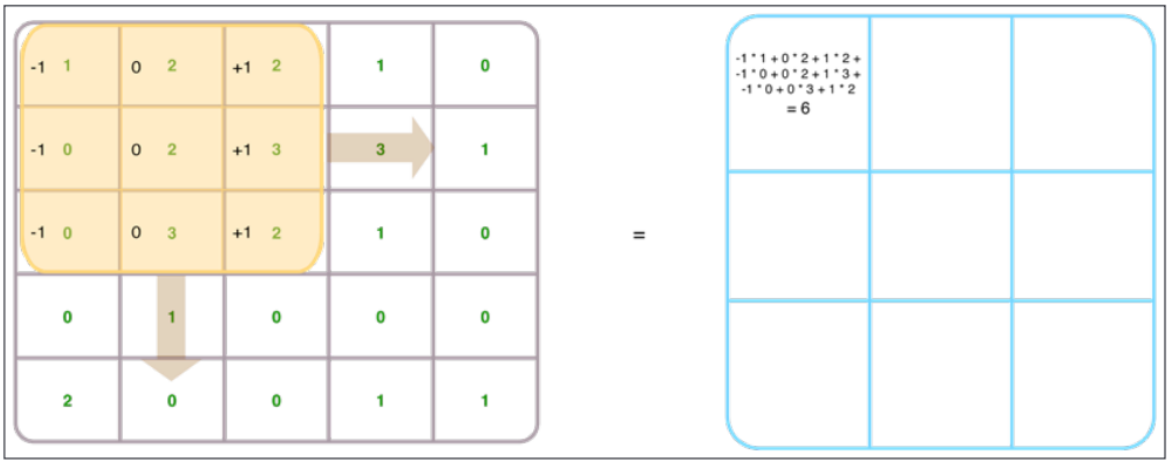
CNNs are an architecture designed to learn from the following key properties, which are relevant to image recognition:

• Data locality: The pixels in an image are highly correlated to the pixels around them. •Translation invariance: An object of interest, for example, a bird, may appear at different places in an image. The model should be able to identify the object, irrespective of the object's position in the image.

• Scale invariance: An object of interest may have a smaller or large size, depending on the zoom. Ideally, the model should be able to identify objects of interest in an image, irrespective of their size. Convolution and pooling layers are key components that aid CNNs in extracting features from images.

Convolutions

A convolution is a mathematical operation that is performed on patches taken from an image with a filter. A filter is a matrix, usually square and with 3x3, 5x5, and 7x7 as common dimensions. The following image shows an example of a 3x3 convolution matrix applied to a 5x5 image. The image patches are taken from left to right and then top to bottom. The number of pixels this patch shifts by every step is called the stride length. A stride length of 1 in a horizontal and vertical direction reduces a 5x5 image to a 3x3 image, as shown here



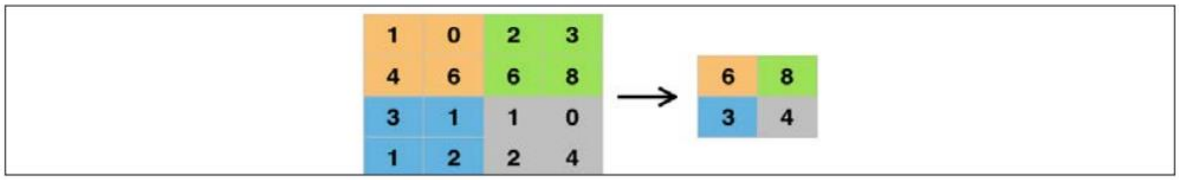
The specific filter that was applied here is an edge detection filter. Prior to CNNs, CV relied heavily on handcrafted filters. Sobel filters are an example of a special filter for the purpose of edge detection.

CNNs can learn many such filters by treating the filter matrices as learnable parameters. CNNs often pass an image through hundreds or thousands of such filters, referred to as channels, and stack them together. You can think of each filter as detecting some features, like vertical lines, horizontal lines, arcs, circles, trapezoids, and so on. However, the magic happens when multiple such layers are put together. Stacking multiple layers leads to learning hierarchical representations. An easy way to understand this concept is by imagining that earlier layers are learning simple shapes like lines and arcs, middle layers are learning shapes like circles and hexagons, and the top layers are learning complex objects like stop signs and steering wheels. The convolution operation is the key innovation that exploits data locality and extracts features that enable translation invariance. A consequence of this layering is the amount of data flowing through the model increasing. Pooling is an operation that helps reduce the dimensions of the data flowing through and further highlights these features.

Pooling

Once the values from the convolution operation have been computed, a pooling operation can be applied to patches to further concentrate the signal in the image. The most common form of pooling is called Max pooling and is demonstrated in the following diagram. It is as simple as taking the maximum value in a patch.

The following diagram shows max pooling on non-overlapping 2x2 patches:



Another way to pool is by averaging the values. While pooling reduces the complexity and computation load, it also helps modestly with scale invariance. However, there is a chance that such a model overfits and does not generalize well. Dropout is a technique that helps with regularization and enables such models to generalize better.

Image feature extraction with ResNet50

ResNet50 models are trained on the ImageNet dataset. This dataset contains millions of images in over 20,000 categories. The large-scale visual recognition challenge, ILSVRC, focuses on the top 1,000 categories for models to compete on recognizing images. Consequently, the top layers of the ResNet50 that perform classification have a dimension of 1,000. The idea behind using a pre-trained ResNet50 model is that it is already able to parse out objects that may be useful in image captioning.

The tensorflow.keras.applications package provides pre-trained models like ResNet50. At the time of writing, all the pre-trained models provided are related to CV. Loading up the pre-trained model is quite easy. All the code for this section is in the feature-extraction.py file in this chapter's folder on GitHub. The main reason for using a separate file is that it gives us the ability to run feature extraction as a script.

Given that we will be processing over 100,000 images, this process may take a while. CNNs benefit greatly from a GPU in computation. Let's get into the code now. First, we must set up the paths for the CSV file we created from the JSON annotations

```
prefix = './data/'
save_prefix = prefix + "features/"  # for storing prefixes
annot = prefix + 'data.csv'
# load the pre-processed file
inputs = pd.read_csv(annot, header=None, names=["caption", "image"])
```

ResNet50 expects each image to be 224x224 pixels with three channels. The input images from the COCO set have different sizes. Hence, we must convert the input files into the standard that ResNet was trained on:

```
# We are going to use the last residual block of
# the ResNet50 architecture
# which has dimension 7x7x2048 and store into individual file
def load_image(image_path, size=(224, 224)):
    # pre-processes images for ResNet50 in batches
    image = tf.io.read_file(image_path)
    image = tf.io.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, size)
    image = preprocess_input(image)  # from keras.applications.ResNet50
    return image, image_path
```

The highlighted code shows a special pre-processing function provided by the ResNet50 package. The pixels in the input image are loaded into an array via the decode_jpeg() function. Each pixel has a value between 0 and 255 for each color channel. The preprocess_input() function normalizes the pixel values so that their mean is 0. Since each input image has five captions, we should only process the unique images in the dataset:

```
uniq_images = sorted(inputs['image'].unique())
print("Unique images: ", len(uniq_images))  # 118,287 images
```

Next, we must convert the dataset into a tf.dat.Dataset, which makes it easier to batch and process the input files using the convenience function defined previously:

```
image_dataset = tf.data.Dataset.from_tensor_slices(uniq_images)
image_dataset = image_dataset.map(
    load_image, num_parallel_calls=tf.data.experimental.AUTOTUNE).
batch(16)
```

For efficiently processing and generating features, we must process 16 image files at a time. The next step is loading a pre-trained ResNet50 model:

```
rs50 = tf.keras.applications.ResNet50(
    include_top=False,
    weights="imagenet",
    input_shape=(224, 224, 3)
)

new_input = rs50.input
hidden_layer = rs50.layers[-1].output

features_extract = tf.keras.Model(new_input, hidden_layer)
features_extract.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Connected to
==================================================================
input_1 (InputLayer)         [(None, 224, 224, 3) 0
_________________________________________________________________
<CONV BLOCK 1>
_________________________________________________________________
<CONV BLOCK 2>
_________________________________________________________________
<CONV BLOCK 3>
_________________________________________________________________
<CONV BLOCK 4>
_________________________________________________________________
<CONV BLOCK 5>
==================================================================
Total params: 23,587,712
Trainable params: 23,534,592
Non-trainable params: 53,120
_________________________________________________________________
```

The preceding output has been abbreviated for brevity. The model contains over 23 million trainable parameters. We don't need the top classification layer as we are using the model for feature extraction. We defined a new model with the input and output layer. Here, we took the output from the last layer. We could take output from different parts of ResNet by changing the definition of the hidden_layer variable. In fact, this variable can be a list of layers, in which case the output of the features_extract model will be the output from each of the layers in the list.

Next, a directory must be set up to store the extracted features:

```
save_prefix = prefix + "features/"
try:
    # Create this directory
    os.mkdir(save_prefix)
except FileExistsError:
    pass # Directory already exists
```

The feature extraction model can work on batches of images and predict the output. The output is 2,048 patches of 7x7 pixels for each image. If a batch of 16 images is supplied, then the output from the model will be a tensor of dimensions [16, 7, 7, 2048]. We store the features of each image file as a separate file while flattening the dimensions to [49, 2048]. Each image has

now been converted into a sequence of 49 pixels, with an embedding size of 2,048. The following code performs this action:

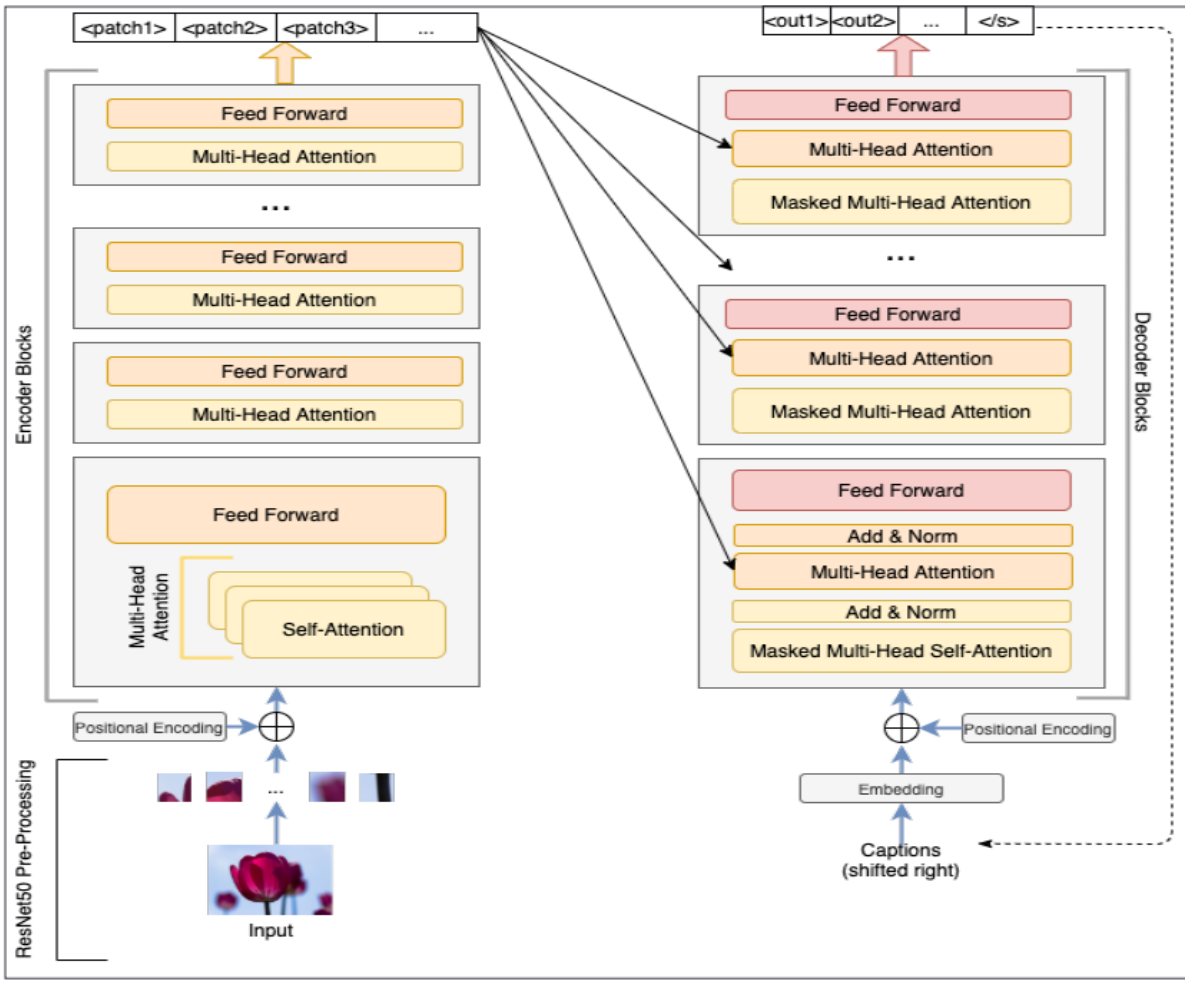
```
for img, path in tqdm(image_dataset):
    batch_features = features_extract(img)
    batch_features = tf.reshape(batch_features,
                                (batch_features.shape[0], -1,
                                 batch_features.shape[3]))

    for feat, p in zip(batch_features, path):
        filepath = p.numpy().decode("utf-8")
        filepath = save_prefix + filepath.split('/')[-1][:-3] + "npy"
        np.save(filepath, feat.numpy())

print("Images saved as npy files")
```

we used a slightly different way of accomplishing the same technique to show how you can accomplish the same objective in different ways. With that, pre-processing and feature extraction is complete. The next step is defining the Transformer model. Then, we will be ready to train the model.

The Transformer model

The Transformer model was inspired by the seq2seq model and has an Encoder and a Decoder part. Since the Transformer model does not rely on RNNs, input sequences need to be annotated with positional encodings, which allow the model to learn about the relationships between inputs.

This innovation of the Transformer model has made very large-sized models such as BERT and GPT-3 possible. Here we will modify the Encoder part of the Transformer to create a visual Encoder, which takes image data as input instead of text sequences. There are some other small modifications to be made to accommodate images as input to the Encoder. The Transformer model we are going to build is shown in the following diagram. The main difference here is how the input sequence is encoded. In the case of text, we will tokenize the text using a Subword Encoder and pass it through an Embedding layer, which is trainable.



As training proceeds, the embeddings of the tokens are also learned. In the case of image captioning, we will pre-process the images into a sequence of 49 pixels, each with an "embedding" size of 2,048. This actually simplifies padding the inputs. All the images are pre-processed so that they're the same length. Consequently, padding and masking the inputs is not required:

The following pieces of code need to be implemented to build the Transformer model: • Positional encoding of the inputs, along with input and output masks. Our inputs are of a fixed length, but the output and captions are of a variable length. • Scaled dot-product attention and multi-head attention to enable the Encoders and Decoders to focus on specific aspects of the data

- An Encoder that consists of multiple repeating blocks.
- A Decoder that uses the outputs from the Encoder through its repeating blocks.

The code for the Transformer has been taken from the TensorFlow tutorial titled Transformer model for language understanding. We will be using this code as the base and adapting it for the image captioning use case. One of the beautiful things about the Transformer architecture is that if we can cast a problem as a sequence-to□sequence problem, then we can apply the Transformer model.

Positional encoding and masks

Transformer models don't use RNNs. This allows them to compute all the outputs in one step, leading to significant improvements in speed and also the ability to learn dependencies across long inputs. However, it comes at the cost of the model not knowing anything about the relationship between neighboring words or tokens. A positional encoding vector, with values for the odd and even positions of the tokens to help the model learn relationships between the positions of inputs, helps compensate for the lack of information about the ordering of tokens.

Scaled dot-product and multi-head attention

The purpose of the attention function is to match a query to a set of key-value pairs. The output is a sum of the values, weighted by the correspondence between the query and the key. multi-head attention learns multiple ways to compute the scaled dot-product attention and combines it. Scaled dot-product attention is computed by multiplying the query vector by the key vector. This product is scaled by the square root

of the dimensions of the query and key. Note that this formulation assumes that the key and query vectors have the same dimensions. Practically, the dimensions of the query, key, and value vectors are all set to the size of the embedding.

VisualEncoder

The diagram shown in the The Transformer model section shows the Encoder's structure. The Encoder processes the inputs with positional encodings and masks, and then passes them through stacks of multi-head attention and feed-forward blocks. The implementation deviates from the TensorFlow tutorial as the input in the tutorial is text. In our case, we are passing 49x2,048 vectors that were generated by passing images through ResNet50. The main difference is in how the inputs are handled. VisualEncoder is built as a layer to allow composition into the eventual Transform model

Decoder

The Decoder is also composed of blocks, just like the Encoder. Each block of the Decoder, however, contains three sub-blocks, as shown in the diagram in the The Transformer model section. There is a masked multi-head attention sub-block, followed by a multi-head attention block, and finally a feed-forward sub-block. The feed-forward sub-block is identical to the Encoder sub-block. We must define a Decoder layer that can be stacked to construct the Decoder.

Transformer

The Transformer is composed of the Encoder, the Decoder, and the final Dense layer for generating output token distributions across the subword vocabulary.

Keras in TensorFlow will provide a higher-level API for defining a Transformer model without you having to write the code out. If this was too much to absorb, then focus on the masks and VisualEncoder as they are the only deviations from the standard Transformer architecture.

Training the Transformer model with VisualEncoder

Training the Transformer model can take hours as we want to train for around 20 epochs. It is best to put the training code into a file so that it can be run from the command line. Note that the model will be able to show some results even after 4 epochs of training. The training code is in the caption-training.py file. At a high level,

the following steps need to be performed before starting training. First, the CSV file with captions and image names is loaded in, and the corresponding paths for the files with extracted image features are appended. The Subword Encoder is also loaded in. A tf.data.Dataset is created with the encoded captions and image features for easy batching and feeding them into the model for training. A loss function, an optimizer with a learning rate schedule, is created for use in training. A custom training loop is used to train the Transformer model. Let's go over these steps in detail.

- Loading training data
- Instantiating the Transformer model

  We will instantiate a small model in terms of the number of layers, attention heads, embedding dimensions, and feed-forward units:

```
# Small Model
num_layers = 4
d_model = 128
dff = d_model * 4
num_heads = 8
```

  For comparison, the BERT base model contains the following parameters:

```
# BERT Base Model
# num_layers = 12
# d_model = 768
# dff = d_model * 4
# num_heads = 12
```

- Custom learning rate schedule This rate schedule is identical to the one proposed in the Attention Is All You Need paper.
- Loss and metrics The loss function is based on categorical cross-entropy. It is a common loss function that we have used in previous chapters. In addition to the loss, an accuracy metric is also defined to track how the model is doing on the training set
- Checkpoints and masks We need to specify a checkpoint directory for TensorFlow to save progress. We will use a CheckpointManager here, which automatically manages the checkpoints and stores a limited number of them. A

checkpoint can be quite large. Five checkpoints for the small model would take up approximately 243 MB of space. Larger models would take up more space.

- Next, a method that will create masks for the input images and captions must be defined:

  Inputs are always a constant length, so the input sequence is set as ones. Only the captions, which are used by the Decoder, are masked. There are two types of masks for the Decoder. The first mask is the padding mask. Since the captions are set to the maximum length to handle 99% of the captions, which works out at about 22 tokens, any captions that are smaller than this number of tokens have padding appended to the end of them. The padding mask helps separate caption tokens from padding tokens. The second mask is the look-ahead mask. It prevents the Decoder from seeing tokens from the future or tokens it has not generated yet. Now, we are ready to train the model.

- Custom training Similar to the summarization model, teacher forcing will be used for training. Consequently, a custom training function will be used. First, we must define a function that will train on one batch of data.

Generating captions

Since the Transformer model is complex, we should consider all these building blocks an important part of your toolkit when you try and solve an NLP problem. For captioning some images. we will use a Jupyter notebook for inference so that we can quickly try out different images.

First we need to load the Subword Encoder, set up masking, instantiate a ResNet50 model to extract features from test images, and generate captions a token at a time until the end of the sequence or a maximum sequence length is reached.

Let's go over these steps one at a time. Once we've done the appropriate imports and optionally initialized the GPU, we can load the Subword Encoder that was saved when we pre-processed the data:

We must now instantiate the Transformer model. This is an important step to ensure the parameters are the same as the checkpoint ones

Restoring the model from the checkpoint requires the optimizer, even though we are not training the model. So, we will reuse the custom scheduler from the training code. As this code was provided previously, it has been omitted here. For the checkpoint, I used a model that was trained for 40 epochs, but without positional encoding in the Encoder

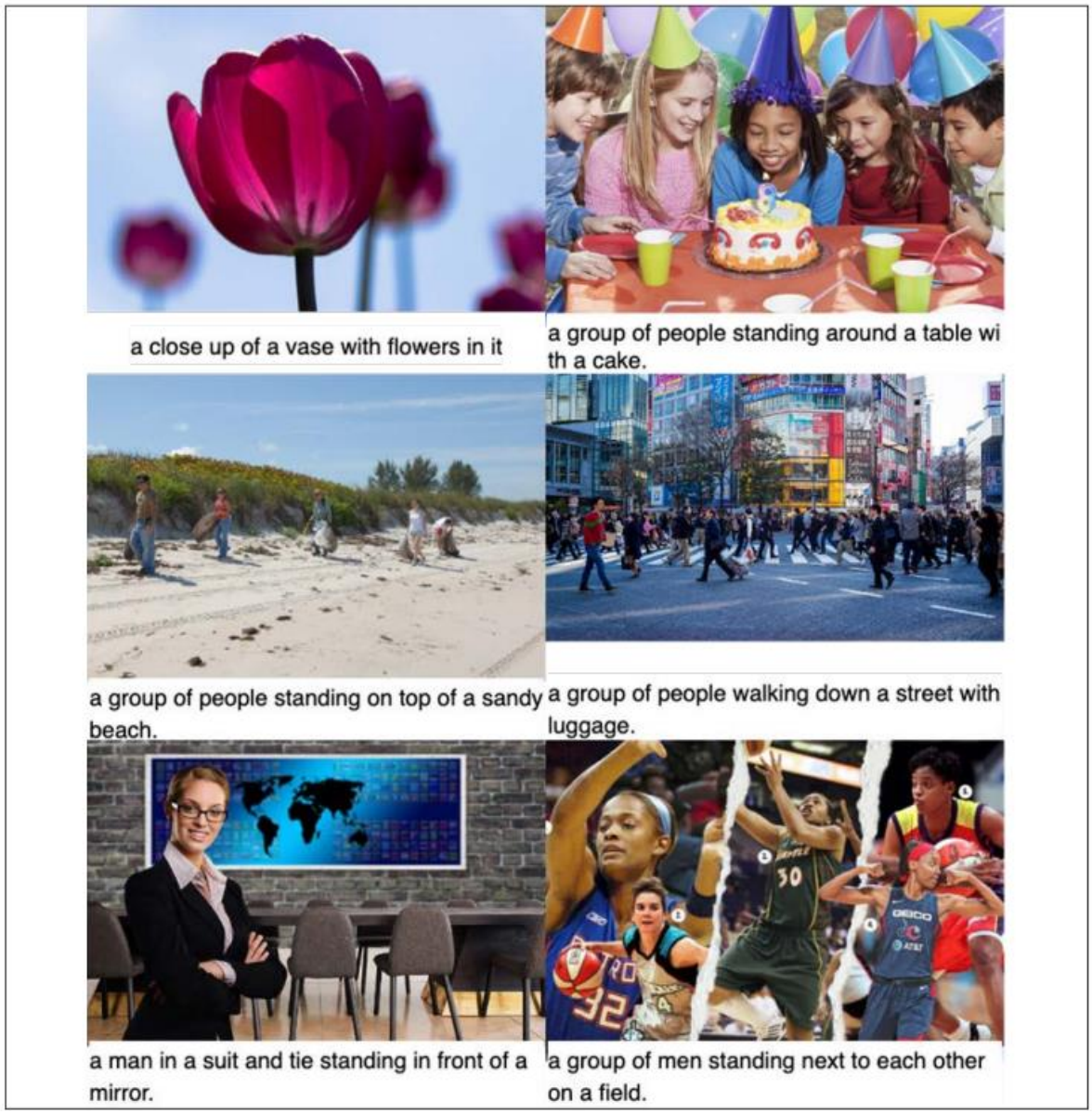
Finally, we must set up the masking function for the generated captions. Note that the look ahead masks don't really help during inference as future tokens have not been generated yet

The main code for inference is in an evaluate() function. This method takes in the image features generated by ResNet50 as input and seeds the output caption sequence with the start token. Then, it runs in a loop to generate a token at a time while updating the masks, until an end of sequence token is encountered or the maximum length of the caption is reached:

The following is the example image and its caption:

Generated caption - A man is riding a surfboard on a wave

The following image shows some more examples of images and their captions. The notebook contains several good, as well as some atrocious, examples of the generated labels:



A quick note on metrics for evaluating the quality of captions. We saw ROUGE metrics in the previous chapters. ROUGE-L is still applicable in the case of image captioning. You can use a mental model of the caption as a summary of an image, as opposed to the summary of a paragraph in text summarization. There can be more than one way to express the summary, and ROUGE-L tries to capture the intent. There are two other commonly reported metrics:

• BLEU: This stands for Bilingual Evaluation Understudy and is the most popular metric in machine translation. We can cast the image captioning problem as a machine translation problem as well. It relies on n-grams for computing the overlap of the predicted text with a number of reference texts and combines the results into one score.

• CIDEr: This stands for Consensus-Based Image Description Evaluation and was proposed in a paper by the same name in 2015. It tries to deal with the difficulty of automatic evaluation when multiple captions could be reasonable by combining TF-IDF and n-grams. The metric tries to compare the captions generated by the model against multiple captions by human annotators and tries to score them based on consensus.

Improving performance and state-of-the art models

Let's first talk through some simple experiments you can try to improve performance before talking about the latest models. Recall our discussion on positional encodings for inputs in the Encoder. Adding or removing positional encodings helps or hinders performance. In the previous chapter, we implemented the beam search algorithm for generating summaries. You can adapt the beam search code and see an improvement in the results with beam search. Another avenue of exploration is the ResNet50. We used a pre-trained network and did not fine-tune it further. It is possible to build an architecture where ResNet is part of the architecture and not a pre-processing step. Image files are loaded in, and features are extracted from ResNet50 as part of the VisualEncoder. ResNet50 layers can be trained from the get-go, or only in the last few iterations. This idea is implemented in the resnet finetuning.py file for you to try. Another line of thinking is using a different object detection model than ResNet50 or using the output from a different layer. You can try a more complex version of ResNet like ResNet152, or a different object detection model like Detectron from Facebook or other models. It should be quite easy to use a different model in our code as it is quite modular.

The second-biggest gain in image captioning comes from pre-training. Recall that BERT and GPT are pre-trained on specific pre-training objectives. Models differ based on whether the Encoder is pre-trained or both the Encoder and Decoder are pre-trained. A common pre-training objective is a version of the BERT MLM task. Recall that BERT inputs are structured as [CLS] I1 I2 … In [SEP] J1 J2 … Jk [SEP], where some of the tokens from the input sequence are masked. This is adapted for image captioning, where the image features and caption tokens in the input are concatenated. Caption tokens are masked similar to how they are in the BERT model, and the pre-training objective is for the model to predict the masked token.

After pre-training, the output of the CLS token can be used for classification or fed to the Decoder to generate the caption. Care must be exercised to not pre-train on the same dataset, like that for evaluation. An example of the setup could be using the Visual Genome and Flickr30k datasets for pre-training and COCO for fine-tuning.

Summary

In the world of deep learning, specific architectures have been developed to handle specific modalities. Convolutional Neural Networks (CNNs) have been incredibly effective in processing images and is the standard architecture for CV tasks. However, the world of research is moving toward the world of multi-modal networks, which can take multiple types of inputs, like sounds, images, text, and so on and perform cognition like humans. After reviewing multi-modal networks, we dived into vision and language tasks as a specific focus. There are a number of problems in this particular area, including image captioning, visual question answering, VCR, and text-to-image, among others. Building on our learnings from previous chapters on seq2seq architectures, custom TensorFlow layers and models, custom learning schedules, and custom training loops, we implemented a Transformer model from scratch. Transformers are state of the art at the time of writing. We took a quick look at the basic concepts of CNNs to help with the image side of things. We were able to build a model that may not be able to generate a thousand words for a picture but is definitely able to generate a human-readable caption. Its performance still needs improvement, and we discussed a number of possibilities so that we can try to do so, including the latest techniques.

It is apparent that deep models perform very well when they contain a lot of data. The BERT and GPT models have shown the value of pre-training on massive amounts of data. It is still very hard to get good quality labeled data for use in pre-training or fine-tuning. In the world of NLP, we have a lot of text data, but not enough labeled data.