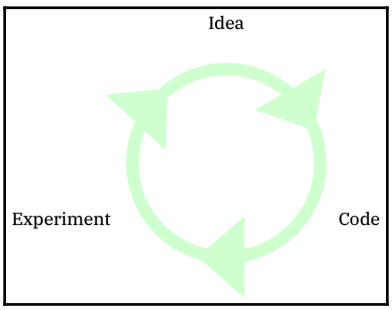
Improving your RNN model

When working on a problem using RNN (or any other network), your process looks like this:



First, you come up with an **idea for the model**, its hyperparameters, the number of layers, how deep the network should be, and so on. Then the model is **implemented and trained** in order to produce some results. Finally, these results are **assessed** and the necessary modifications are made. It is rarely the case that you'll receive meaningful results from the first run. This cycle may occur multiple times until you are satisfied with the outcome.

Considering this approach, one important question comes to mind: *How can we change the model so the next cycle produces better results?*

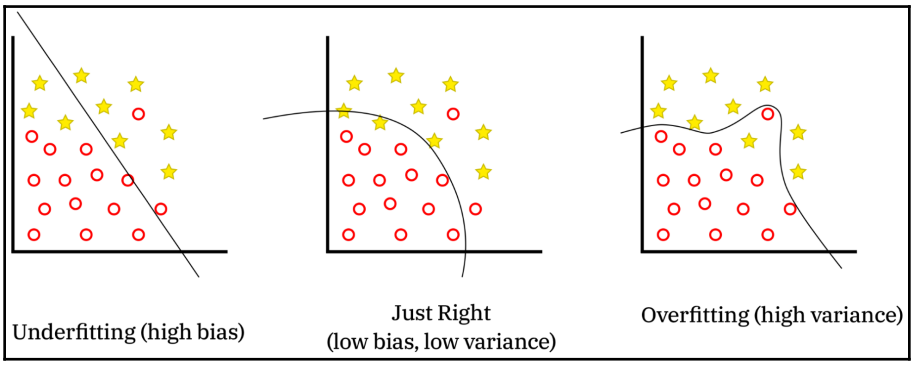
This question is tightly connected to your understanding of the network's results. Let's discuss that now.

As you already know, in the beginning of each model training, you need to prepare lots of quality data. This step should happen before the **Idea** part of the aforementioned cycle. Then, during the **Idea** stage, you should come up with the actual neural network and its characteristics. After that comes the **Code** stage, where you use your data to supply the model and perform the actual training. There is something important to keep in mind—*once your data is collected, you need to split it into 3 parts: training (80%), validation (10%) and testing (10%)*.

The **Code** stage only uses the training part of your data. Then, the **Experiment** stage uses the validation part to evaluate the model. Based on the results of these two operations, we will make the necessary changes.

You should use the testing data **only** after you have gone through all the necessary cycles and have identified that your model is performing well. The testing data will help you understand the rate of accuracy you are receiving on unseen data.

At the end of each cycle, you need to determine how good your model is. Based on the results, you will see that your model is always either **underfitting** (**high bias**) or **overfitting** (**high variance**) the data (by varying degrees). You should aim for both the bias and variance to be low, so there is almost no underfitting or overfitting. The next diagram may help you understand this concept better:



Examining the preceding diagram, we can state the following definitions:

- **Underfitting (high bias)**: This occurs when the network is not influenced enough by the training data, and generalizes the prediction
- **Just Right (low bias, low variance)**: This occurs when the network makes quality predictions, both during training and in the general case during testing
- **Overfitting (high variance)**: This occurs when the network is influenced by the training data too much, and makes false decisions on new entries.

The preceding diagram may be helpful to understand the concepts of high bias and high variance, but it is difficult to apply this to real examples. The problem is that we normally deal with data of more than two dimensions. That is why we will be using the loss (error) function values produced by the model to make the same evaluation for higher dimensional data.

Let's say we are evaluating the Spanish-to-English translator neural network from `Chapter 4`, *Creating a Spanish-to-English Translator*. We can assume that the lowest possible error on that task can be produced by a human, and it is 1.5%. Now we will evaluate the results based on all the error combinations that our network can give:

- Training data error: ~2%; Validation data error: ~14%: **high variance**
- Training data error: ~14%; Validation data error: ~15%: **high bias**
- Training data error: ~14%; Validation data error: ~30%: **high variance, high bias**
- Training data error:~2%; Validation data error: ~2.4%: **low variance, low bias**

The desired output is having low variance and low bias. Of course, it takes a lot of time and effort to get this kind of improvement, but in the end, it is worth doing.

You have now got familiar with how to read your model results and evaluate the model's performance. Now, let's see what can be done to **lower both the variance and the bias of the model**.

How can we lower the variance? (fixing overfitting)

A very useful approach is to collect and transform more data. This will generalize the model and make it perform well on both the training and validation sets.

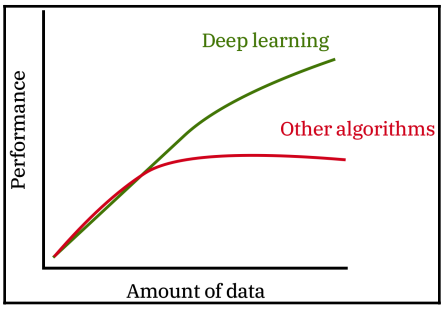
How can we lower the bias? (fixing underfitting)

This can be done by increasing the network depth—that is, changing the numbers of layers and of hidden units, and tuning the hyperparameters.

Next, we will cover both of these approaches and see how to use them effectively to improve our neural network's performance.

Improving performance with data

A large amount of quality data is critical for the success of any deep learning model. A good comparison can be made to other algorithms, where an increased volume of data does not necessarily improve performance:



But this doesn't mean that gathering more data is always the right approach. For example, if our model suffers from underfitting, more data won't increase the performance. On the other hand, solving the overfitting problem can be done using exactly that approach.

Improving the model performance with data comes in three steps: **selecting data**, **processing data**, and **transforming data**. It is important to note that all three steps should be done according to your specific problem. For some tasks, such as recognizing digits inside an image, a nicely formatted dataset can be found easily. For more concrete tasks (e.g. analyzing images from plants), you may need to experiment and come up with non-trivial decisions.

Selecting data

This is a pretty straightforward technique. You either collect more data or invent more training examples.

Finding more data can be done using an online collection of datasets (`https://skymind.ai/wiki/open-datasetshttps://skymind.ai/wiki/open-datasets`). Other methods are to scrape web pages, or use the advanced options of Google Search (`https://www.google.com/advanced_search`).

On the other hand, inventing or augmenting data is a challenging and complex problem, especially if we are trying to generate text or images. For example, a new approach (`https://www.quora.com/What-data-augmentation-techniques-are-available-for-deep-learning-on-text`) for augmenting text was created recently. It is done by translating an English sentence to another language and then back to English. This way we are getting two slightly different but meaningful sentences, which increases and diversifies our dataset substantially. Another interesting technique for augmenting data, specifically for RNN language models, can be found in the paper on *Data Noising as Smoothing in Neural Network Language Models* (`https://arxiv.org/abs/1703.02573`).

Processing data

After you have selected the required data, the time comes for processing. This can be done with these three steps:

- **Formatting**: This involves converting the data into the most suitable format for your application. Imagine, for example, that your data is the text from thousands of PDF files. You should extract the text and covert the data into CSV format.
- **Cleaning**: Often, it is the case that your data may be incomplete. For example, if you have scraped book metadata from the internet, some entries may have missing data (such as ISBN, date of writing, and so on). Your job is to decide whether to fix or discard the metadata for the whole book.
- **Sampling**: Using a small part of the dataset can reduce computational time and speed up your training cycles while you are determining the model accuracy.

The order of the preceding steps is not determined, and you may revisit them multiple times.

Transforming data

Finally, you need to transform the data using techniques such as scaling, decomposition, and feature selection. First, it is good to plot/visualize your data using Matplotlib (a Python library) or TensorFlow's TensorBoard (`https://www.tensorflow.org/guide/summaries_and_tensorboard`).

Scaling is a technique that converts every entry into a number within a specific range (0-1) without mitigating its effectiveness. Normally, scaling is done within the bounds of your activation functions. If you are using sigmoid activation functions, rescale your data to values between 0 and 1. If you're using the hyperbolic tangent (tanh), rescale to values between -1 and 1. This applies to inputs (x) and outputs (y).

Decomposition is a technique of splitting some features into their components and using them instead. For example, the feature time may have minutes and hours, but we care only about the minutes.

Feature selection is one of the most important decisions you would make when building your model. A great tutorial to follow when deciding how to choose the most appropriate features is Jason Brownlee's *An Introduction to Feature Selection* (`https://machinelearningmastery.com/an-introduction-to-feature-selection/`).

Processing and transforming data can be accomplished using the vast selection of Python libraries, such as NumPy, among others. They turn out to be pretty handy when it comes to data manipulation.

After you have gone through all of the preceding steps (probably multiple times), you can move forward to building your neural network model.

Improving performance with tuning

After selecting, processing, and transforming your data, it's time for a second optimization technique—hyperparameter tuning. This approach is one of the most important components in building your model and you need to spend the time necessary to execute it well.

Every neural network model has parameters and hyperparameters. These are two distinct sets of values. Parameters are learned by the model during training, such as weights and biases. On the other hand, hyperparameters are predefined values that are selected after careful observation. In a standard recurrent neural network, the set of hyperparameters includes the number of hidden units, number of layers, RNN model type, sequence length, batch size, number of epochs (iterations), and the learning rate.

Your task is to identify the best of all possible combinations so that the network performs pretty well. This is a pretty challenging task and often takes a lot of time (hours, days, even months) and computational power.

Following Andrew Ng's tutorial on hyperparameter tuning (`https://www.coursera.org/lecture/deep-neural-network/hyperparameters-tuning-in-practice-pandas-vs-caviar-DHNcc`), we can separate this process into two different techniques: *Pandas* vs *Caviar*.

The *Pandas* approach follows the way pandas (that is, the animal) raise their children. We initialize our model with a specific set of parameters, and then improve these values after every training operation until we achieve delightful results. This approach is ideal if you lack computational power and multiple GPUs to train neural networks simultaneously.

The *Caviar* approach follows the way fish reproduce. We introduce multiple models at once (using different sets of parameters) and train them at the same time, while tracking the results. This technique will likely require access to more computational power.

Now the question becomes: *How can we decide what should be included in our set of hyperparameters?*

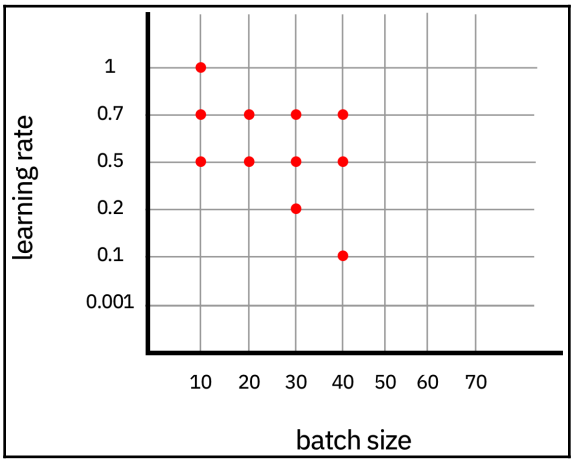
Summarizing a great article on hyperparameters optimization (`http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html#tree-structured-parzen-estimators-tpe`), we can define five ways for tuning:

- **Grid search**
- **Random search**
- **Hand-tuning**
- **Bayesian optimization**
- **Tree-structured Parzen Estimators** (**TPE**)

During the beginning phase of your deep learning journey, you will mostly be utilizing grid search, random search, and hand-tuning. The last two techniques are more complex in terms of understanding and implementation. We will cover both of them in the following section, but bear in mind that, for trivial tasks, you can go with normal hand-tuning.

Grid search

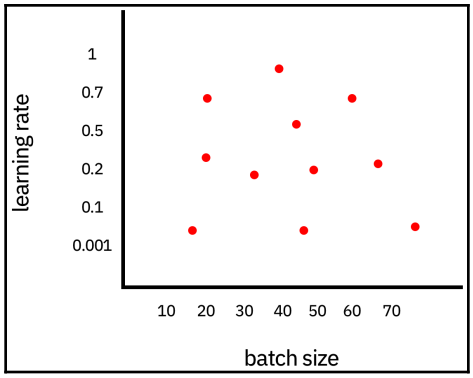
This is the most straightforward way of finding the right hyperparameters. It follows the approach in this graph:



Here, we generate all the possible combinations of values for the hyperparameters and perform separate training cycles. This works for small neural networks, but is impractical for more complex tasks. That is why we should use the better approach listed in the following section.

Random search

This technique is similar to grid search. You can follow the graph here:



Instead of taking all the possible combinations, we sample a smaller set of random values and use these values to train the model. If we see that a particular group of closely positioned dots tends to perform better, we can examine this region more closely and focus on it.

Hand-tuning

Bigger networks normally require more time for training. This is why the aforementioned approaches are not ideal for such situations. In these cases, we often use the hand-tuning technique. The idea is to initially try one set of values, and then evaluate the performance. Then, our intuition, as well as our learning experience, may lead to ideas on a specific sequence of changes. We perform those tweaks and learn new things about the model. After several iterations, we have a good understanding of what needs to change for future improvement.

Bayesian optimization

This approach is a way of learning the hyperparameters without the need to manually determine different values. It uses a Gaussian process that utilizes a set of previously evaluated parameters, and the resultant accuracy, to make an assumption about unobserved parameters. An acquisition function uses this information to suggest the next set of parameters. For more information, I suggest watching Professor Hinton's lecture on *Bayesian optimization of Hyper Parameters* (`https://www.youtube.com/watch?v=cWQDeB9WqvU`).

Tree-structured Parzen Estimators (TPE)

The idea behind this approach is that, at each iteration, TPE collects new observations, and at the end of the iteration, the algorithm decides which set of parameters it should try next. For more information, I suggest taking a look at this amazing article on *Hyperparameters optimization for Neural Networks* (`http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html#tree-structured-parzen-estimators-tpe`).

Optimizing the TensorFlow library

This section focuses mostly on practical advice that can be directly implemented in your code. The TensorFlow team has provided a large set of tools that can be utilized to improve your performance. These techniques are constantly being updated to achieve better results. I strongly recommend watching TensorFlow's video on training performance from the 2018 TensorFlow conference (`https://www.youtube.com/watch?v=SxOsJPaxHME`). This video is accompanied by nicely aggregated documentation, which is also a must-read (`https://www.tensorflow.org/performance/`).

Now, let's dive into more details around what you can do to achieve faster and more reliable training.

Let's first start with an illustration from TensorFlow that presents the general steps of training a neural network. You can divide this process into three phases: **data processing**, **performing training**, and **optimizing gradients**:

1. **Data processing (step 1)**: This phase includes fetching the data (locally or from a network) and transforming it to fit our needs. These transformations might include augmentation, batching, and so on. Normally, these operations are done on the **CPU**.
2. **Perform training (steps 2a, 2b and 2c)**: This phase includes computing the forward pass during training, which requires a specific neural network model—LSTM, GPU, or a basic RNN in our case. These operations utilize powerful **GPUs** and **TPUs**.
3. **Optimize gradients (step 3)**: This phase includes the process of minimizing the loss function with the aim of optimizing the weights. The operation is again performed on **GRUs** and **TPUs**.