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

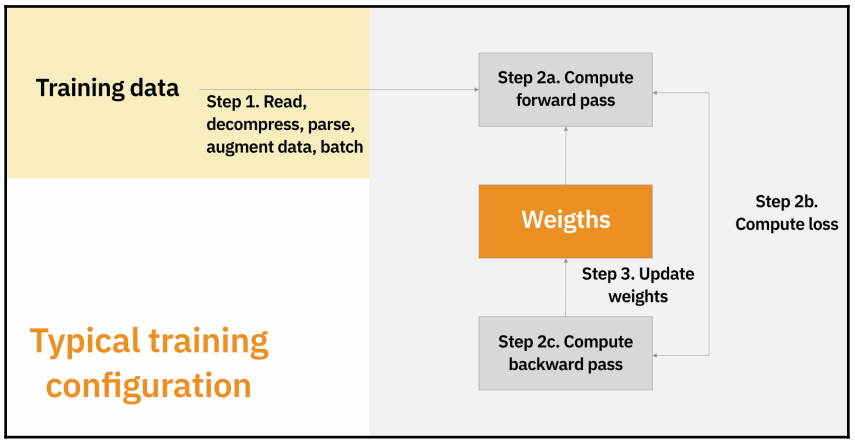
This section focuses mostly on practical advice that can be directly implemented in your code. The TensorFlow team has provided a large set of tools that can be utilized to improve your performance. These techniques are constantly being updated to achieve better results. I strongly recommend watching 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**.

This graph illustrates the above steps:



Next, let's explain how to improve each of these steps.

Data processing

You need to examine if loading and transforming the data is the bottleneck of your performance. You can do this with several approaches, some of which involve estimating the time it takes to perform these tasks, as well as tracking the CPU usage.

After you have determined that these operations are slowing down the performance of your model, it's time to apply some useful techniques to speed things up.

As we said, these operations (loading and transforming data) should be performed on the CPU, rather than the GPU, so that you free up the latter for training. To ensure this, wrap your code as follows:

```
with tf.device('/cpu:0'):
    # call a function that fetches and transforms the data
    final_data = fetch_and_process_data()
```

Then, you need to focus on both the process of loading (fetching) and transforming the data.

Improving data loading

The TensorFlow team has been working hard to make this as easy as possible by providing the `tf.data` API (`https://www.tensorflow.org/performance/performance_guide`), which works incredibly well. To learn more about it and understand how to use it efficiently, I recommend watching TensorFlow's talk on `tf.data` (`https://www.youtube.com/watch?v=uIcqeP7MFH0`). This API should always be used, instead of the standard `feed_dict` approach you have seen so far.

Improving data transformation

Transformations can come in different forms, for example, cropping images, splitting text, and rendering and batching files. TensorFlow offers solutions for these techniques. For example, if you are cropping images before training, it is good to use `tf.image.decode_and_crop_jpeg`, which decodes only the part of the image required. Another optimization can be made in the batching process. The TensorFlow library offers two methods:

```
batch_normalization = tf.layers.batch_normalization(input_layer,
fused=True, data_format='NCHW')
```

The second method is as follows:

```
batch_normalizaton = tf.contrib.layers.batch_norm(input_layer, fused=True,
data_format='NCHW')
```

Let's clarify these lines:

- Batch normalization is performed to a neural network model to speed up the process of training. Refer to this amazing article, *Batch Normalization in Neural Networks*, for more details: `https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c`.
- The `fused` parameter indicates whether or not the method should combine the multiple operations, required for batch normalization, into a single kernel.
- The `data_format` parameter refers to the structure of the Tensor passed to a given Operation (such as summation, division, training, and so on). A good explanation can be found under *Data formats* in the TensorFlow performance guide (`https://www.tensorflow.org/performance/`).

Performing the training

Now, let's move on to the phase of performing the training. Here, we are using one of TensorFlow's built-in functions for initializing recurrent neural network cells and calculating their weights using the preprocessed data.

Depending on your situation, different techniques for optimizing your training may be more appropriate:

- For small and experimental models, you can use `tf.nn.rnn_cell.BasicLSTMCell`. Unfortunately, this is highly inefficient and takes up more memory than the following optimized versions. That is why using it is **not** recommended, unless you are just experimenting.
- An optimized version of the previous code is `tf.contrib.rnn.LSTMBlockFusedCell`. It should be used when you don't have access to GPUs or TPUs and want run a more efficient cell.
- The best set of cells to use is under `tf.contrib.cudnn_rnn.*` (`CuddnnCompatibleGPUCell` for GPU cells and more). They are highly optimized to run on GPUs and perform significantly better than the preceding ones.

Finally, you should always perform the training using `tf.nn.dynamic_rnn` (see the TensorFlow documentation: `https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn`) and pass the specific cell. This method optimizes the training of the recurrent neural networks by occasionally swapping memory between GPUs and CPUs to enable training of large sequences.

Optimizing gradients

The last optimization technique will actually improve the performance of our backpropagation algorithm. Recall from the previous chapters that your goal during training is to minimize the loss function by adjusting the weights and biases of the model. Adjusting (optimizing) these weights and biases can be accomplished with different built-in TensorFlow optimizers, such as `tf.train.AdamOptimizer` and `tf.train.GradientDescentOptimizer`.