Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a 2015 paper,[7] Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change (which they call the *Internal Covariate Shift* problem).

The technique consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer.

In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs over the current mini-batch (hence the name "Batch Normalization"). The whole operation is summarized in Equation 11-3.

Equation 11-3. Batch Normalization algorithm

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$
3. $\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \widehat{\mathbf{x}}^{(i)} + \beta$

- μ_B is the empirical mean, evaluated over the whole mini-batch B.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.

7 "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," S. Ioffe and C. Szegedy (2015).

- $\widehat{\mathbf{x}}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

At test time, there is no mini-batch to compute the empirical mean and standard deviation, so instead you simply use the whole training set's mean and standard deviation. These are typically efficiently computed during training using a moving average. So, in total, four parameters are learned for each batch-normalized layer: γ (scale), β (offset), μ (mean), and σ (standard deviation).

The authors demonstrated that this technique considerably improved all the deep neural networks they experimented with. The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the logistic activation function. The networks were also much less sensitive to the weight initialization. They were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that "Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. […] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters." Finally, like a gift that keeps on giving, Batch Normalization also acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in the chapter).

Batch Normalization does, however, add some complexity to the model (although it removes the need for normalizing the input data since the first hidden layer will take care of that, provided it is batch-normalized). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. So if you need predictions to be lightning-fast, you may want to check how well plain ELU + He initialization perform before playing with Batch Normalization.

You may find that training is rather slow at first while Gradient Descent is searching for the optimal scales and offsets for each layer, but it accelerates once it has found reasonably good values.

Implementing Batch Normalization with TensorFlow

TensorFlow provides a `batch_normalization()` function that simply centers and normalizes the inputs, but you must compute the mean and standard deviation yourself (based on the mini-batch data during training or on the full dataset during testing, as just discussed) and pass them as parameters to this function, and you must also handle the creation of the scaling and offset parameters (and pass them to this function). It is doable, but not the most convenient approach. Instead, you should use the `batch_norm()` function, which handles all this for you. You can either call it directly or tell the `fully_connected()` function to use it, such as in the following code:

```
import tensorflow as tf
from tensorflow.contrib.layers import batch_norm

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

is_training = tf.placeholder(tf.bool, shape=(), name='is_training')
bn_params = {
    'is_training': is_training,
    'decay': 0.99,
    'updates_collections': None
}

hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          normalizer_fn=batch_norm, normalizer_params=bn_params)
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2",
                          normalizer_fn=batch_norm, normalizer_params=bn_params)
logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="outputs",
                         normalizer_fn=batch_norm, normalizer_params=bn_params)
```

Let's walk through this code. The first lines are fairly self-explanatory, until we define the `is_training` placeholder, which will either be `True` or `False`. This will be used to tell the `batch_norm()` function whether it should use the current mini-batch's mean and standard deviation (during training) or the running averages that it keeps track of (during testing).

Next we define `bn_params`, which is a dictionary that defines the parameters that will be passed to the `batch_norm()` function, including `is_training` of course. The algorithm uses *exponential decay* to compute the running averages, which is why it requires the `decay` parameters. Given a new value v, the running average $\hat{v}$ is updated through the equation $\hat{v} \leftarrow \hat{v} \times \text{decay} + v \times (1 - \text{decay})$. A good decay value is typically close to 1—for example, 0.9, 0.99, or 0.999 (you want more 9s for larger datasets and

smaller mini-batches). Finally, `updates_collections` should be set to `None` if you want the `batch_norm()` function to update the running averages right before it performs batch normalization during training (i.e., when `is_training=True`). If you don't set this parameter, by default TensorFlow will just add the operations that update the running averages to a collection of operations that you must run yourself.

Lastly, we create the layers by calling the `fully_connected()` function, just like we did in Chapter 10, but this time we tell it to use the `batch_norm()` function (with the parameters `nb_params`) to normalize the inputs right before calling the activation function.

Note that by default `batch_norm()` only centers, normalizes, and shifts the inputs; it does not scale them (i.e., γ is fixed to 1). This makes sense for layers with no activation function or with the ReLU activation function, since the next layer's weights can take care of scaling, but for any other activation function, you should add `"scale": True` to `bn_params`.

You may have noticed that defining the preceding three layers was fairly repetitive since several parameters were identical. To avoid repeating the same parameters over and over again, you can create an *argument scope* using the `arg_scope()` function: the first parameter is a list of functions, and the other parameters will be passed to these functions automatically. The last three lines of the preceding code can be modified like so:

```
[...]

with tf.contrib.framework.arg_scope(
        [fully_connected],
        normalizer_fn=batch_norm,
        normalizer_params=bn_params):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

It may not look much better than before in this small example, but if you have 10 layers and want to set the activation function, the initializers, the normalizers, the regularizers, and so on, it will make your code much more readable.

The rest of the construction phase is the same as in Chapter 10: define the cost function, create an optimizer, tell it to minimize the cost function, define the evaluation operations, create a `Saver`, and so on.

The execution phase is also pretty much the same, with one exception. Whenever you run an operation that depends on the `batch_norm` layer, you need to set the `is_training` placeholder to `True` or `False`:

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op,
                     feed_dict={is_training: True, X: X_batch, y: y_batch})
        accuracy_score = accuracy.eval(
            feed_dict={is_training: False, X: X_test_scaled, y: y_test}))
        print(accuracy_score)
```

That's all! In this tiny example with just two layers, it's unlikely that Batch Normalization will have a very positive impact, but for deeper networks it can make a tremendous difference.

Gradient Clipping

A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for recurrent neural networks; see Chapter 14). This is called *Gradient Clipping*.[8] In general people now prefer Batch Normalization, but it's still useful to know about Gradient Clipping and how to implement it.

In TensorFlow, the optimizer's `minimize()` function takes care of both computing the gradients and applying them, so you must instead call the optimizer's `compute_gradients()` method first, then create an operation to clip the gradients using the `clip_by_value()` function, and finally create an operation to apply the clipped gradients using the optimizer's `apply_gradients()` method:

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

You would then run this `training_op` at every training step, as usual. It will compute the gradients, clip them between –1.0 and 1.0, and apply them. The threshold is a hyperparameter you can tune.

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task

8 "On the difficulty of training recurrent neural networks," R. Pascanu et al. (2013).