Dropconnect is a variant of dropout where individual connections are dropped randomly rather than whole neurons. In general dropout performs better.

Max-Norm Regularization

Another regularization technique that is quite popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \le r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

We typically implement this constraint by computing $\|\mathbf{w}\|_2$ after each training step and clipping $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w}\frac{r}{\| \mathbf{w} \|_2}$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients problems (if you are not using Batch Normalization).

TensorFlow does not provide an off-the-shelf max-norm regularizer, but it is not too hard to implement. The following code creates a node `clip_weights` that will clip the `weights` variable along the second axis so that each row vector has a maximum norm of 1.0:

```
threshold = 1.0
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

You would then apply this operation after each training step, like so:

```
with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            clip_weights.eval()
```

You may wonder how to get access to the `weights` variable of each layer. For this you can simply use a variable scope like this:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")

with tf.variable_scope("hidden1", reuse=True):
    weights1 = tf.get_variable("weights")
```

Alternatively, you can use the root variable scope:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
```

```
[...]

with tf.variable_scope("", default_name="", reuse=True):  # root scope
    weights1 = tf.get_variable("hidden1/weights")
    weights2 = tf.get_variable("hidden2/weights")
```

If you don't know what the name of a variable is, you can either use TensorBoard to find out or simply use the `global_variables()` function and print out all the variable names:

```
for variable in tf.global_variables():
    print(variable.name)
```

Although the preceding solution should work fine, it is a bit messy. A cleaner solution is to create a `max_norm_regularizer()` function and use it just like the earlier `l1_regularizer()` function:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None  # there is no regularization loss term
    return max_norm
```

This function returns a parametrized `max_norm()` function that you can use like any other regularizer:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          weights_regularizer=max_norm_reg)
```

Note that max-norm regularization does not require adding a regularization loss term to your overall loss function, so the `max_norm()` function returns `None`. But you still need to be able to run the `clip_weights` operation after each training step, so you need to be able to get a handle on it. This is why the `max_norm()` function adds the `clip_weights` node to a collection of max-norm clipping operations. You need to fetch these clipping operations and run them after each training step:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            sess.run(clip_all_weights)
```

Much cleaner code, isn't it?

Data Augmentation

One last regularization technique, data augmentation, consists of generating new training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting, making this a regularization technique. The trick is to generate realistic training instances; ideally, a human should not be able to tell which instances were generated and which ones were not. Moreover, simply adding white noise will not help; the modifications you apply should be learnable (white noise is not).

For example, if your model is meant to classify pictures of mushrooms, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see Figure 11-10). This forces the model to be more tolerant to the position, orientation, and size of the mushrooms in the picture. If you want the model to be more tolerant to lighting conditions, you can similarly generate many images with various contrasts. Assuming the mushrooms are symmetrical, you can also flip the pictures horizontally. By combining these transformations you can greatly increase the size of your training set.

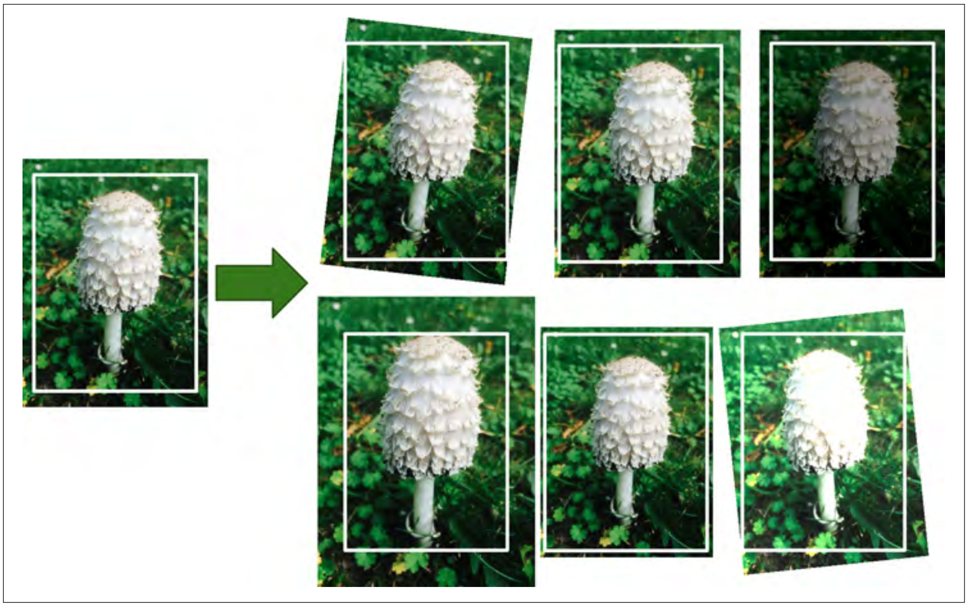

Figure 11-10. Generating new training instances from existing ones

It is often preferable to generate training instances on the fly during training rather than wasting storage space and network bandwidth. TensorFlow offers several image manipulation operations such as transposing (shifting), rotating, resizing, flipping, and cropping, as well as adjusting the brightness, contrast, saturation, and hue (see