```
    activation_fn=tf.nn.relu,
    normalizer_fn=tf.nn.l2_normalize,
    normalizer_params={"dim": 1},
    out_channels=64)
d3 = Dense(
    in_layers=[d2],
    activation_fn=tf.nn.relu,
    normalizer_fn=tf.nn.l2_normalize,
    normalizer_params={"dim": 1},
    out_channels=32)
d4 = Dense(
    in_layers=[d3],
    activation_fn=tf.nn.relu,
    normalizer_fn=tf.nn.l2_normalize,
    normalizer_params={"dim": 1},
    out_channels=16)
d4 = BatchNorm(in_layers=[d4])
d5 = Dense(in_layers=[d4], activation_fn=None, out_channels=9)
value = Dense(in_layers=[d4], activation_fn=None, out_channels=1)
value = Squeeze(squeeze_dims=1, in_layers=[value])
action_prob = SoftMax(in_layers=[d5])
```

### Is Feature Engineering Dead?

In this section, we feed the raw tic-tac-toe game board into TensorFlow for training the policy. However, it's important to note that for more complex games than tic-tac-toe, this may not yield satisfactory results. One of the lesser known facts about AlphaGo is that DeepMind performs sophisticated feature engineering to extract "interesting" patterns of Go pieces upon the board to make AlphaGo's learning easier. (This fact is tucked away into the supplemental information of DeepMind's paper.)

The fact remains that reinforcement learning (and deep learning methods broadly) often still need human-guided feature engineering to extract meaningful information before learning algorithms can learn effective policies and models. It's likely that as more computational power becomes available through hardware advances, this need for feature engineering will be reduced, but for the near term, plan on manually extracting information about your systems as needed for performance.

## The A3C Loss Function

We now have the object-oriented machinery set in place to define a loss for the A3C policy network. This loss function will itself be implemented as a `Layer` object (it's a convenient abstraction that all parts of the deep architecture are simply layers). The `A3CLoss` object implements a mathematical loss consisting of the sum of three terms:

a policy_loss, a value_loss, and an entropy term for exploration. See Example 8-21.

*Example 8-21. This Layer implements the loss function for A3C*

```
class A3CLoss(Layer):
  """This layer computes the loss function for A3C."""

  def __init__(self, value_weight, entropy_weight, **kwargs):
    super(A3CLoss, self).__init__(**kwargs)
    self.value_weight = value_weight
    self.entropy_weight = entropy_weight

  def create_tensor(self, **kwargs):
    reward, action, prob, value, advantage = [
        layer.out_tensor for layer in self.in_layers
    ]
    prob = prob + np.finfo(np.float32).eps
    log_prob = tf.log(prob)
    policy_loss = -tf.reduce_mean(
        advantage * tf.reduce_sum(action * log_prob, axis=1))
    value_loss = tf.reduce_mean(tf.square(reward - value))
    entropy = -tf.reduce_mean(tf.reduce_sum(prob * log_prob, axis=1))
    self.out_tensor = policy_loss + self.value_weight * value_loss
    - self.entropy_weight * entropy
    return self.out_tensor
```

There are a lot of pieces to this definition, so let's pull out bits of code and inspect. The A3CLoss layer takes in reward, action, prob, value, advantage layers as inputs. For mathematical stability, we convert probabilities to log probabilities (this is numerically much more stable). See Example 8-22.

*Example 8-22. This snippet from A3CLoss takes reward, action, prob, value, advantage as input layers and computes a log probability*

```
reward, action, prob, value, advantage = [
    layer.out_tensor for layer in self.in_layers
]
prob = prob + np.finfo(np.float32).eps
log_prob = tf.log(prob)
```

The policy loss computes the sum of all advantages observed, weighted by the log-probability of the action taken. (Recall that the advantage is the difference in reward resulting from taking the given action as opposed to the expected reward from the raw policy for that state). The intuition here is that the policy_loss provides a signal on which actions were fruitful and which were not (Example 8-23).

*Example 8-23. This snippet from A3CLoss defines the policy loss*

```
policy_loss = -tf.reduce_mean(
    advantage * tf.reduce_sum(action * log_prob, axis=1))
```

The value loss computes the difference between our estimate of $V$ (`reward`) and the actual value of $V$ observed (`value`). Note the use of the $L^2$ loss here (Example 8-24).

*Example 8-24. This snippet from A3CLoss defines the value loss*

```
value_loss = tf.reduce_mean(tf.square(reward - value))
```

The entropy term is an addition that encourages the policy to explore further by adding some noise. This term is effectively a form of regularization for A3C networks. The final loss computed by `A3CLoss` is a linear combination of these component losses. See Example 8-25.

*Example 8-25. This snippet from A3CLoss defines an entropy term added to the loss*

```
entropy = -tf.reduce_mean(tf.reduce_sum(prob * log_prob, axis=1))
```

## Defining Workers

Thus far, you've seen how the policy network is constructed, but you haven't yet seen how the asynchronous training procedure is implemented. Conceptually, asynchronous training consists of individual workers running gradient descent on locally simulated game rollouts and contributing learned knowledge back to a global set of weights periodically. Continuing our object-oriented design, let's introduce the `Worker` class.

Each `Worker` instance holds a copy of the model that's trained asynchronously on a separate thread (Example 8-26). Note that `a3c.build_graph()` is used to construct a local copy of the TensorFlow computation graph for the thread in question. Take special note of `local_vars` and `global_vars` here. We need to make sure to train only the variables associated with this worker's copy of the policy and not with the global copy of the variables (which is used to share information across worker threads). As a result `gradients` uses `tf.gradients` to take gradients of the loss with respect to only `local_vars`.