Gradient search

We generalize the form of Gradient Descent

- we minimize a function F that depends on variables $\mathbf{v}$
$$\mathbf{v}^* = \operatorname*{argmin}_{\mathbf{v}} F(\mathbf{v})$$
- by iteratively updating $\mathbf{v}$
    - creating a sequence of $\mathbf{v}_{(0)}, \mathbf{v}_{(1}, \ldots$
- via the update equation
    - moving in the *negative* direction of the gradient of F wrt $\mathbf{v}$
    - moderated by learning rate α
$$\mathbf{v}_{(t)} = \mathbf{v}_{(t-1)} - \alpha * \frac{\partial F}{\partial \mathbf{v}}$$

Our familiar Gradient Descent

- identifies F with the Loss Function $\mathcal{L}$
    - which is a function of weights $\mathbf{W}$
- and searches for the optimal $\mathbf{W}^*$

We define a *maximization* version of the search

- we maximize a function F that depends on variables $\mathbf{v}$
$$\mathbf{v}^* = \underset{\mathbf{v}}{\operatorname{argmax}} F(\mathbf{v})$$
- by iteratively updating $\mathbf{v}$
  - creating a sequence of $\mathbf{v}_{(0)}, \mathbf{v}_{(1}, \ldots$
- via the update equation
  - moving in the *positive* direction of the gradient of F wrt $\mathbf{v}$
  - moderated by learning rate α
$$\mathbf{v}_{(t)} = \mathbf{v}_{(t-1)} + \alpha * \frac{\partial F}{\partial \mathbf{v}}$$

The maximization version is referred to as *Gradient Ascent* and is the topic of this module.

Gradient Ascent in Code

In code, one step of Gradient **Descent** looks like this

- from [Keras docs (https://colab.research.google.com/github/keras-team/keras-io/blob/master/guides/ipynb/customizing_what_happens_in_fit.ipynb#scrollTo=9z4](https://colab.research.google.com/github/keras-team/keras-io/blob/master/guides/ipynb/customizing_what_happens_in_fit.ipynb#scrollTo=9z4)
- inputs are a mini-batch of examples

```
with tf.GradientTape() as tape:
    y_pred = self(x, training=True)  # Forward pass
    # Compute the loss value
    # (the loss function is configured in `compile()`)
    loss = self.compiled_loss(y, y_pred, regularization_losses=self.losses)

# Compute gradients
trainable_vars = self.trainable_variables
gradients = tape.gradient(loss, trainable_vars)

# Update weights
self.optimizer.apply_gradients(zip(gradients, trainable_vars))
```

Key points

- Define a loss $\mathcal{L}$
    - the loss is dependent on the weights ("trainable variables") of the model
- Compute the loss within the scope of `tf.GradientTape()`
    - Enables TensorFlow to compute gradients of any variable accessed in the scope
        - Loss calculated via `self.compiled_loss` in this case
        - but any calculation that you would chose to define
- Obtain the gradients of the loss with respect to the trainable variables
- Updates the trainable variables
    - `self.optimizer.apply_gradients(zip(gradients, trainable_vars))` in this case
    - General case `weight += - learning_rate * gradient`
    - Subtract the gradient: we are descending (reducing loss)

Gradient Ascent is nearly identical

- Except that we update
    - a collection of variables
        - **not necesarilty** the weights
        - perhaps some other variable
    - in the *positive* direction of the gradients
- So as to *maximize* a function ("utility")
    - we will continue, in code, to use "loss" for the function/variable name

In code, it looks like this:

```
with tf.GradientTape() as tape:
    tape.watch(vars)
    loss = compute_loss(vars)

# Compute gradients.
gradients = tape.gradient(loss, vars)

vars += learning_rate * gradients
```

- `vars` is a list of variables
- loss is dependent on `vars`
- we compute the gradient of the loss with respect to `vars`
- we *add* the gradient wrt `vars`:
    - we are ascending (increasing loss: better to call it "utility")

Gradient Ascent: uses

Background

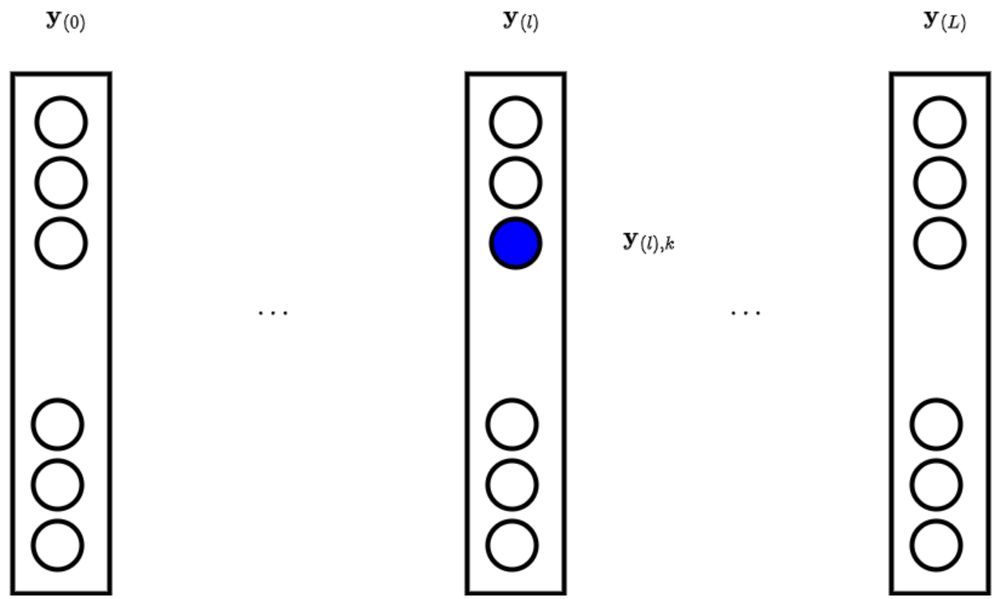
Note

This is addressed more specifically in the module on Intepretation (Interpretation_of_DL_Simple.ipynb#Probing)

There are many synthetic features created among the layers of a Neural Network.

How do we discern the purpose of a specific feature k at layer l: $\mathbf{y}_{(l),k}$?

Interpreting neuron for feature k in layer l



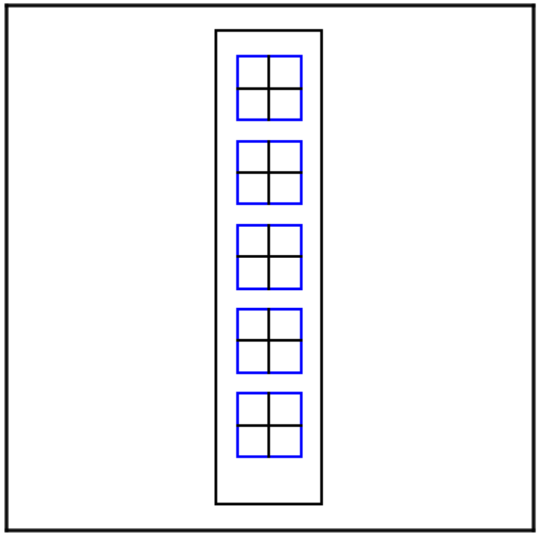
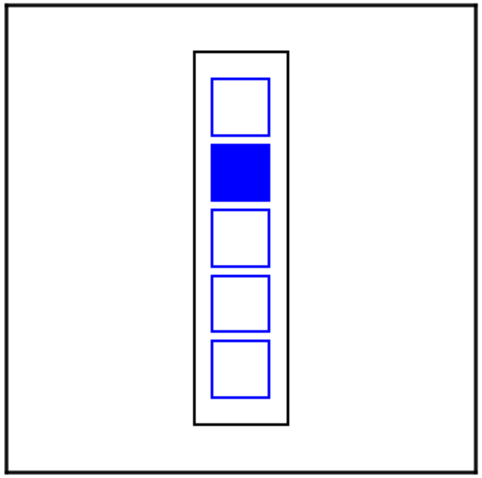
But when layer l has $N \geq 1$ non-feature dimensions

- the selected feature is really a *feature map*
- with dimensions matching the non-feature dimensions of the layer input
$$(d_1 \times d_2 \times \ldots d_N)$$

So there are $\prod_{i=1}^N d_i$ values (one per location) in the feature map

- rather than a single scalar value
- as in the case of layer outputs with only a feature dimension

Convolutional layer: $\mathbf{y}_{(l)}$: selecting a feature map to probe

Layer w/non-feature dimensions: $\mathbf{y}_{(l)}$	Layer w/non-feature dimensions, one element selected: $\mathbf{y}_{(l),j}$
	

In such a case

- we reduce each feature map (with non-feature dimensions)
- to a scalar
- using a Pooling operation to eliminate the non-feature dimensions
    - for example: Global Max Pooling

Convolutional layer: $\mathbf{y}_{(l)}$: selecting a feature map to probe
Global Pooling

Layer w/non-feature dimensions: $\mathbf{y}_{(l)}$	Layer w/non-feature dimensions, pooled, one element selected: $\mathbf{y}_{(l),j}$

For the remainder of this section

- we assume $\mathbf{y}_{(l),k}$ is a single scalar
- obtained by removing the non-feature dimensions, as above

Use Case: Find the maximally activating input

One way to discern the purpose of feature $\mathbf{y}_{(l),k}$

- is to find the input $\mathbf{y}_{(0)}$
- that maximizes the *activation* (value) of $\mathbf{y}_{(l),k}$

We use Gradient Ascent where

- $F = \mathbf{y}_{(l),k}$
- $\mathbf{v} = \mathbf{y}_{(0)}$

to discover the input vector $\mathbf{y}^*_{(0)}$ that *maximally activates* $\mathbf{y}_{(l),k}$

$$\mathbf{y}^*_{(0)} = \underset{\mathbf{y}_{(0)}}{\operatorname{argmax}} \left(\mathbf{y}_{(l),k} \big|_{\mathbf{y}_{(0)} = \mathbf{y}_{(0)}} \right)$$

Note

- $\mathbf{y}^*_{(0)}$ is **not necessarily** an input example $\mathbf{x} \in \mathbf{X}$, the training dataset
- it is an vector with the *shape* of an input

Visualizing what convnets learn, via Gradient Ascent

Let's make this concrete.

Suppose we have a sequential network with multiple CNN layers.

Consider layer l which is implemented by a CNN with $n_{(l)}$ features.

- There are $n_{(l)}$ feature maps as output of layer l
- Each is determined by a filter/kernel associated with each of these features

Let us focus on feature k.

Can we determine what in *the input image* ($\mathbf{y}_{(0)}$) is being detected by the filter for this feature ?

There are many non-feature dimensions (row and column for image)

- we remove the feature map's non-feature dimensions
- by Average Pooling
    - replacing the collection of values across multiple locations
    - by a single value: the mean value over the multiple locations
- As illustrated by the diagrams above.

This single value will be our proxy for the entire feature map.

What are we trying to maximize ?

- The single value that is the proxy for the feature map

What are the "parameters" that the optimizer can alter to achieve the maximization

- a pixel grid that would appear as input to layer 0

As described above

$$\mathbf{y}^*_{(0)} = \underset{\mathbf{y}_{(0)}}{\operatorname{argmax}} \left(\mathbf{y}_{(l),k} \big|_{\mathbf{y}_{(0)} = \mathbf{y}_{(0)}} \right)$$

$\mathbf{y}_{(l),k} \big|_{\mathbf{y}_{(0)} = \mathbf{y}_{(0)}}$ denotes

- the feature map (from which we remove non-feature dimensions via Average Pooling, resulting in a scalar)
- that is output by feature k of layer l when presented with input value $\mathbf{y}_{(0)}$ at Layer 0
- we search over all possible values of $\mathbf{y}_{(0)}$
    - to find the *maximally activating* $\mathbf{y}_{(0)}$ for feature k of layer l

Here is the code

Visualizing what convnets learn (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/visualizing_what_convnets_learn.ipynb#)

A blog post from a previous version (https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html) of the code shows the patterns of multiple feature maps at multiple layers.

Interesting sub-case: maximizing a logit in the Classifier Head

When the Neural Network solves a Classification task

- the head layer L is a Classifier Layer
- with number of features equal to the number of possible classes
- output feature k is the *probability* (or pre-probability "logit")
    - on input being in class k

So Gradient Ascent when

- $F = \mathbf{y}_{(L),k}$
- $\mathbf{v} = \mathbf{y}_{(0)}$

finds the input $\mathbf{y}^*_{(0)}$

- that results in the *most confident* prediction that it is in class k

In [2]:

```
print("Done")
```

```
Done
```