Hopefully by now, we know about *Gradient Descent*

- Solving for weights/parameters
- That minimize a loss function
- By updating weights/parameters in the *negative* direction of the gradients with respect to the parameters/weights

In code, it 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)
- one step of Gradient Descent (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")

# Uses of Gradient Ascent

We will show some interesting things you can do using Gradient Ascent.

We need to identify

- the property being maximized
- the variables that will be adjusted by the maximizer

Our examples identify

- the property as defined by variable $\mathcal{L}$ (which we are maximizing, not minimizing)
- the variables being adjusted are the **inputs** to the NN
    - `vars` = $\mathbf{y}_{(0)}$

That is

- we are solving for the inputs that maximize a property.

## Property to maximize: value of a single "logit" of the Classifier head

Suppose our Neural Network $\mathbb{C}$ terminates in a Classifier head, over classes $\{c_1, \ldots c_k\}$.

The Classifier Head is a `Dense` layer with $k$ units ("logits"), one per class.

For example, MNIST digit classification

- input is a $(28 \times 28)$ grid of pixel values
- there are 10 "logits"
    - corresponding to each of the digits $\{0, 1, \ldots, 9\}$

Define the property to be maximized

- The value of logit corresponding to $c_j$

Gradient Ascent will find the input value to $\mathbb{C}$ that will be classified with highest probability as being from class $c_j$.

This is the "paradigmatic" input of class $c_j$.

Using MNIST digit classification as our example.

Suppose we choose the property: "Maximize the logit for digit 8"

- we want to solve for a $(28 \times 28)$ pixel grid
- that is classified as an "8" with highest probability

Note

- the pixel grid solution *does not necessarily* look like a digit !

## Property to maximize: summary of values of one feature map

We can generalize the MNIST example

- suppose we have a multi-layer Sequential NN
- we want to *interpret* the purpose of layer $l$
    - in our first case, $l = L$ (the Classifier layer)
    - here: $l$ can be an intermediate layer

Recall that a *feature map* is

- a Tensor (with shape equal to the spatial dimensions)
- corresponding to the value of a single feature at some layer $l$
    - over each spatial location

Since this feature is not a singleton, imagine we reduced it to a single value

- e.g., maximum value

Define the property to be maximized as

- a single value that summarizes *all* the values of the single feature map at layer $l$
    - for example: the maximum value over the Tensor

We use Gradient Ascent

- to solve for a $(28 \times 28)$ pixel grid
- the *maximally activates* the feature map at layer $l$

Again, the solution pixel grid may not look like a digit

- but it may reveal a pattern that "triggers" this layer
    - for example: "strong vertical line"
        - indicative of digits 1, 4, 9

This is one way in which we attempt to discern what role each layer serves.

### Visualizing what convnets learn, via Gradient Ascent

Let's illustrate Gradient Ascent to visualize what one feature map within a Convolutional Layer of an Image Classifier is "looking for"

[Visualizing what convnets learn (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/visualizing_what_convnets_learn.ipynb#)](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] of the code shows the patterns of multiple feature maps at multiple layers.

In [2]:

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

Done