All of the main ML libraries provide several choices of optimization algorithm for training models. Why? What’s the difference between them, and how to choose?

TensorFlow/Keras:

```python
model.compile(
    optimizer=keras.optimizers.RMSprop(),  # Optimizer
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)
```

**Available optimizers**

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl
# Algorithms For Training ML Models

**PyTorch:**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adadelta</td>
<td>Implements Adadelta algorithm.</td>
</tr>
<tr>
<td>Adagrad</td>
<td>Implements Adagrad algorithm.</td>
</tr>
<tr>
<td>Adam</td>
<td>Implements Adam algorithm.</td>
</tr>
<tr>
<td>AdamW</td>
<td>Implements AdamW algorithm.</td>
</tr>
<tr>
<td>SparseAdam</td>
<td>Implements lazy version of Adam algorithm suitable for sparse tensors.</td>
</tr>
<tr>
<td>Adamax</td>
<td>Implements Adamax algorithm (a variant of Adam based on infinity norm).</td>
</tr>
<tr>
<td>ASGD</td>
<td>Implements Averaged Stochastic Gradient Descent.</td>
</tr>
<tr>
<td>LBFGS</td>
<td>Implements L-BFGS algorithm, heavily inspired by minFunc.</td>
</tr>
<tr>
<td>RMSprop</td>
<td>Implements RMSprop algorithm.</td>
</tr>
<tr>
<td>Rprop</td>
<td>Implements the resilient backpropagation algorithm.</td>
</tr>
<tr>
<td>SGD</td>
<td>Implements stochastic gradient descent (optionally with momentum).</td>
</tr>
</tbody>
</table>
Algorithms For Training ML Models

sklearn:

```
from sklearn.neural_network import MLPClassifier

class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=100, activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000):

    solver : {'lbfgs', 'sgd', 'adam'}, default='adam'
    The solver for weight optimization.
    
    - 'lbfgs' is an optimizer in the family of quasi-Newton methods.
    - 'sgd' refers to stochastic gradient descent.
    - 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba
```
The main algorithms popular are:

- SGD - Stochastic Gradient Descent
- Adam - Adaptive Moment Estimation
- RMSProp
- Adagrad - Adaptive gradient
- L-BFGS - BFGS = Broyden-Fletcher-Goldfarb-Shannon algo, L=limited memory. Only suitable for smaller learning tasks

Let’s look more closely at these, plus some other algos for context.
Basic structure of an optimisation algorithm to minimise function $f(x)$ is:

(i) initialise $x$
(ii) calculate step i.e. change to $x$
(iii) $x = x - \text{step}$
(iv) repeat (ii)-(iii) until result is good enough or run out of time

In python:

```python
x = x0
for k in range(num_iters):
    step = calcStep(fn,x)
    x = x - step
```

How to calculate step?
Derivatives

* When $x$ is scalar the derivative of a function $f(\cdot)$ at point $x$ is the slope of the line that just touches the function at $x$ e.g.

* When $x = [x_1, x_2]$ then the derivative is the slope of the plane that just touches the function at $x$ e.g.

* And similarly when $x$ has $> 2$ elements, but can’t draw it.
Derivatives

* Equation of a line is $y = mx + c$. $m$ is slope, $c$ is intercept (when $x = 0$ then $y = c$).

![Graphs showing different lines with varying slopes and intercepts](image)
Derivatives

- **Derivative** of a function $f(\cdot)$ at point $x'$ is the *slope* of the line that just touches the function at $x'$.

- Equation of a line is $mx + c$, slope $m$ and intercept $c$.
- Line touches $f(\cdot)$ at point $x'$, so $mx' + c = f(x')$ i.e. $c = f(x') - mx'$
- Slope $m$ of line is derivative, i.e. $m = \frac{df}{dx}(x)$ (this is just notation\(^1\), the important point is that we can calculate $\frac{df}{dx}(x)$ and so $m$ using standard tools).
- Putting this together, equation of line is

$$\frac{df}{dx}(x)(x - x') + f(x')$$

and so

$$f(x) \approx f(x') + \frac{df}{dx}(x')(x - x')$$

\(^1\)Sometimes $f'(x)$ is used instead of $\frac{df}{dx}(x)$.
Derivatives:

Example:

1. \( f(x) = x^2 \). Then \( \frac{df}{dx}(x) = 2x \)
2. At point \( x' = 1 \),

\[
f(x) \approx f(1) + \frac{df}{dx}(1)(x - 1) = 1 + 2(x - 1)
\]
Derivatives:

Example:

* \( f(x) = x^2 \). Then \( \frac{df}{dx}(x) = 2x \)

* At point \( x' = -0.5 \),

\[
f(x) \approx f(-0.5) + \frac{df}{dx}(-0.5)(x - (-0.5)) = 0.5^2 + 2(-0.5)(x - (-0.5)) \]
\[
= 0.2 - (x + 0.5)
\]
Derivatives

\[ f(x) \approx f(x') + \frac{df}{dx}(x')(x - x') \]

Suppose we choose \( x = x' - \alpha \frac{df}{dx}(x') \), then

\[
f(x) \approx f(x') + \frac{df}{dx}(x')(x' - \alpha \frac{df}{dx}(x') - x')
\]

\[
= f(x') - \alpha \left( \frac{df}{dx}(x_0) \right)^2
\]

The square of a number is always \( \geq 0 \), so \( \left( \frac{df}{dx}(x') \right)^2 \geq 0 \). And so

\[ f(x) \approx f(x') \]

i.e. moving from point \( x' \) to \( x = x' - \alpha \frac{df}{dx}(x') \) tends to decrease function \( f(\cdot) \).

Only approximate though, because line is only an approximation to function \( f(\cdot) \) near point \( x' \).
Derivatives

* Equation of a plane is $y = m_1 x_1 + m_2 x_2 + c$
* Note: we now have two slopes $m_1$, $m_2$ and intercept $c$
* Notation: For plane just touching $f(\cdot)$ at point $x'$
  * $m_1 = \frac{\partial f}{\partial x_1} (x')$, $m_2 = \frac{\partial f}{\partial x_2} (x')$
  * $\frac{\partial f}{\partial x_1} (x')$ is the partial derivative of $f(\cdot)$ wrt $x_1$ at point $x'$
  * $\frac{\partial f}{\partial x_2} (x')$ is the partial derivative of $f(\cdot)$ wrt $x_2$ at point $x'$
  * $\nabla f(x') = [\frac{\partial f}{\partial x_1} (x'), \frac{\partial f}{\partial x_2} (x'), \ldots, \frac{\partial f}{\partial x_n} (x')]$, the vector of partial derivatives. And sometimes $\nabla x_1 f(x')$ is used for $\frac{\partial f}{\partial x_1} (x')$ etc.
* This plane is an approximation to function $f(\cdot)$ near point $x'$ i.e.

$$f(x) \approx f(x') + \frac{\partial f}{\partial x_1} (x')(x_1 - x'_1) + \frac{\partial f}{\partial x_2} (x')(x_2 - x'_2)$$

* If we choose $x_1 = x' - \alpha \frac{\partial f}{\partial x_1}$ and $x_2 = x' - \alpha \frac{\partial f}{\partial x_2}$ then moving from point $x'$ to $x$ tends to decrease function $f(\cdot)$ i.e. $f(x) \lessgtr f(x')$
In general, when vector $x$ as $n$ elements then

$$f(x) \approx f(x') + \frac{\partial f}{\partial x_1}(x')(x_1 - x'_1) + \frac{\partial f}{\partial x_2}(x')(x_2 - x'_2) + \cdots + \frac{\partial f}{\partial x_n}(x')(x_n - x'_n)$$

Choosing $x$ with $x_1 = x' - \alpha \frac{\partial f}{\partial x_1}$, $x_2 = x' - \alpha \frac{\partial f}{\partial x_2}$, $\ldots$, $x_n = x' - \alpha \frac{\partial f}{\partial x_n}$ then moving from point $x'$ to $x$ tends to decrease function $f(\cdot)$
Calculating Derivatives

* Derivative of $f(x) = x^2$:

```python
import sympy
x = sympy.symbols('x', real=True)
f=x**2
dfdx = sympy.diff(f,x)
print(f,dfdx)
```

Output is:

$x^2$ 2*x

* Derivative of $f(x) = 0.5(x_0^2 + 10x_1^2)$, vector $x = [x_0, x_1]$:

```python
import sympy
x0, x1 = sympy.symbols('x0, x1', real=True)
x=sympy.Array([x0,x1])
f=0.5*(x[0]**2+10*x[1]**2)
dfdx = sympy.diff(f,x)
print(f,dfdxc)
```

Output is:

$0.5*x0**2 + 5.0*x1**2$ [1.0*x0, 10.0*x1]

* Also tensorflow https://www.tensorflow.org/guide/autodiff, pytorch https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
Can use *finite difference* approximation to a derivative as a sanity check:

* Recall

\[
\begin{align*}
  f(x) & \approx f(x') + \frac{\partial f}{\partial x_1}(x')(x_1-x'_1) + \frac{\partial f}{\partial x_2}(x')(x_2-x'_2) + \cdots + \frac{\partial f}{\partial x_n}(x')(x_n-x'_n)
\end{align*}
\]

* Select \( x = x' \) and then add a small amount \( \delta \) to element 1 of \( x \) i.e.

\[
x = [x'_1 + \delta, x'_n, \ldots, x'_n]
\]

Then

\[
f(x) \approx f(x') + \frac{\partial f}{\partial x_1}(x')\delta
\]

i.e.

\[
\frac{\partial f}{\partial x_1}(x') \approx \frac{f(x) - f(x')}{\delta}
\]

* Perturbation \( \delta \) needs to be small e.g. 0.01 or less.
Example:

* \( f(x) = x^2, \quad \frac{df}{dx}(x) = 2x \)
* At point \( x' = 1 \) then \( \frac{df}{dx}(1) = 2.0 \)
* Finite difference:

\[
\frac{f(x' + \delta) - f(x')}{\delta} = \frac{f(1 + \delta) - f(1)}{\delta} = \frac{(1 + \delta)^2 - 1}{\delta}
\]

For \( \delta = 0.1 \rightarrow 2.1 \)

For \( \delta = 0.01 \rightarrow 2.01 \)

For \( \delta = 0.001 \rightarrow 2.0010 \)
Verifying Derivative Calculations

Derivative of \( f(x) = x^2 \):

```python
import sympy
x = sympy.symbols('x', real=True)
f = x**2
dfdx = sympy.diff(f, x)
f = sympy.lambdify(x, f)
dfdx = sympy.lambdify(x, dfdx)
x = np.array([1.0])
print(dfdx(*x))
delta = 0.01
print(((x + delta)**2 - x**2) / delta)
```

Output is:
2.0
2.01
x=x0
for k in range(num_iters):
    step = calcStep(fn,x)
    x = x − step

Now we know one way to choose the step, namely:

\[
\text{step} = \alpha \left[ \frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right]
\]

where \( \alpha \) is the \textit{step size} or \textit{learning rate}