Análisis Matemático para Inteligencia Artificial

Martín Errázquin (merrazquin@fi.uba.ar)

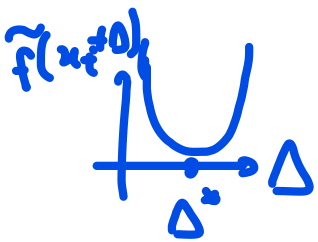
Especialización en Inteligencia Artificial

Métodos de segundo orden

Método de Newton

Recordemos el polinomio de Taylor de grado 2 de una función $f(x)$ alrededor de un punto x_t evaluada en un punto $\tilde{x} = x_t + \Delta$ con Δ pequeño:

$$f(\tilde{x}) \approx f(x_t) + f'(x_t)(\tilde{x} - x_t) + \frac{1}{2} f''(x_t)(\tilde{x} - x_t)^2$$



$$f(x_t + \Delta) \approx f(x_t) + f'(x_t)\Delta + \frac{1}{2} f''(x_t)\Delta^2$$

Si derivamos e igualamos a 0, obtenemos el mínimo en $\Delta^* = -\frac{f'(x_t)}{f''(x_t)}$. En versión multivariada, esto es $\Delta^* = H^{-1} \nabla_f(x_t)^T$.
El método de Newton es eso:

$$\theta_{t+1} = \theta_t - H^{-1} \nabla_J(\theta_t)^T$$

Pro: Tiene convergencia local cuadrática.
Con: Es *caro* estimar $H^{-1} \nabla_J(\theta_t)^T$ (Según Goodfellow 10^4 vs 10^2 para $\nabla_J(\theta_t)^T$.

BFGS

Como calcular $H^{-1}\nabla_J(\theta_t)^T$ en cada iteración es muy caro, se plantea aproximar H usando H_t iterable que sea simple. Se pide que H_t sea simétrica y definida positiva, y que además cumpla la *ecuación secante*:

$$H_t y_t = s_t$$

donde $y_t = \Delta\nabla_J(\theta) = \nabla_J(\theta_t) - \nabla_J(\theta_{t-1})$ y $s_t = \Delta\theta = \theta_t - \theta_{t-1}$.

La regla de update resulta:

$$H_{t+1} = V_t^T H_t V_t + \frac{s_t s_t^T}{y_t^T s_t}$$

donde $V_t = I - \frac{s_t y_t^T}{y_t^T s_t}$. Observar que el método, si bien es más eficiente, es sub-cuadrático ("Quasi-Newton") en iteraciones.

Nota: ¿Cómo inicializar H_0? No hay fórmula, suele usarse $H_0 = I$. Por ejemplo, Sklearn/SciPy lo hacen.

L-BFGS

Un problema inherente a BFGS (y cualquier método similar) es que es $\mathcal{O}(n^2)$ en memoria. L-BFGS plantea usar sólo información de las últimas m iteraciones para aproximar H, específicamente los pares (s_k, y_k) con $k = t-1, \ldots, t-m$. Luego se preestablece un $H_t^0 = \frac{s_{t-1}^T y_{t-1}}{y_{t-1}^T y_{t-1}} I$ y se aplican los m pasos de BFGS hasta llegar a H_t. Si bien la cuenta es engorrosa, es eficiente de computar $H_t \nabla_J(\theta_t)$:

```
q = grad_t
for i in k-1, ..., k-m:
  alpha[i] = s[i].T @ q / (y[i].T @ s[i])
  q -= alpha[i] * y[i]
res = H_k0 @ q
for i in k-m, ..., k-1:
  res += s[i] * (alpha[i] - y[i].T @ r)
```

Luego de obtener el resultado, se elimina del buffer el par (s, y) más viejo y se reemplaza por el último, siendo entonces $\mathcal{O}(mn)$.

¿Cuándo conviene usar qué?

En términos generales, los métodos de 1º orden son mucho más rápidos. Además, si bien existen variantes que lo solventan, en general los métodos de 2º orden requieren entrenar *en batch*.

Sin embargo, hay casos donde esto está bien. Por ejemplo, las librerías de árboles boosteados (GBDT) como LightGBM o XGBoost usan métodos de 2º orden *entre árboles*.

- Para datasets "chicos" y/o modelos tradicionales de pocos parámetros es más que aceptable usar métodos de 2º orden.
- Para datasets o modelos muy grandes resulta prohibitivo entrenar en batch, y suelen premiarse *updates rápidos*.
- Como siempre, el método correcto *depende* de la situación.