Gradient Descent

Gradient Descent w/ Squared Errors

- Log loss/ Cross entropy vs. Squared errors
  - CE: Summing the negative log of prob. the smaller the value the better!
  - SE: Sum of squares of the difference bt. true label & predicted label.

$$E = \frac{1}{2}\sum_{\mu}\sum_{j}\left[y_j^{\mu} - \hat{y}_j^{\mu}\right]^2$$

j ≡ output units

μ ≡ data points / samples

Remember that the output of a neural network, the prediction, depends on the weights

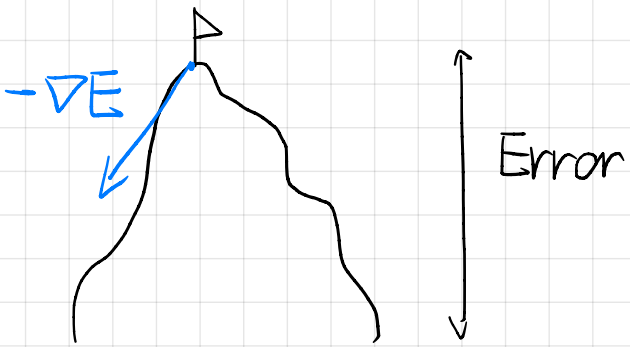
$$\hat{y}_j^{\mu} = f\left(\sum_i w_{ij} x_i^{\mu}\right)$$

and accordingly the error depends on the weights

$$E = \frac{1}{2}\sum_{\mu}\sum_{j}\left[y_j^{\mu} - f\left(\sum_i w_{ij} x_i^{\mu}\right)\right]^2$$

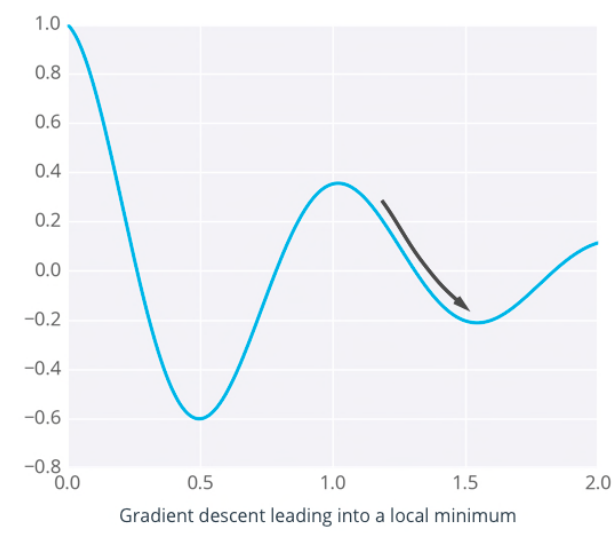
- Using a square to make + & − the same
- Penalize outliers!

Gradient Descent:



- Step in the direction of gradient is the greatest decrease
- A derivative generalized to function w/ more variables

Caveats



Gradient descent leading into a local minimum

- Fail to reach to global minimum due to wrongly initialized weights.
- Use Momentum as the optimizer.

Math

$$E = \frac{1}{2}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2$$

$$= \frac{1}{2}\sum_{\mu}(y^{\mu} - f(\sum_{i} w_i x_i^{\mu}))^2$$

Error as a function of weights

⇓

weights ≡ the knobs for adjusting the error.

DATA RECORDS

$$\begin{bmatrix} x_1^1 & x_2^1 & x_3^1 \\ x_1^2 & x_2^2 & x_3^2 \\ x_1^3 & x_2^3 & x_3^3 \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{bmatrix} \quad \begin{bmatrix} y^1 \\ y^2 \\ y^3 \\ \dots \\ \dots \end{bmatrix} \quad \mu = 1$$

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i \propto -\frac{\partial E}{\partial w_i} \longrightarrow \text{THE GRADIENT}$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

LEARNING RATE

Therefore, updating weights

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w}\frac{1}{2}(y - \hat{y}(w_i))^2$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w} \frac{1}{2}[y-\hat{y}(w_i)]^2$$

$$= [y-\hat{y}(w_i)] \cdot \frac{1}{2} \cdot 2 \cdot \frac{\partial}{\partial w_i}[y-\hat{y}(w_i)]$$

$$= [y-\hat{y}(w_i)] - \frac{\partial \hat{y}(w_i)}{\partial w_i}$$

Since $\hat{y}(w_i) = f(h) = f(\sum_i w_i x_i + b)$

$$\frac{\partial E}{\partial w_i} = [y-\hat{y}(w_i)] - f'(h) \frac{\partial}{\partial w_i}(\sum_i w_i x_i + b)$$

$$\frac{\partial}{\partial w_i}(\sum_i w_i x_i + b) = x_i$$

$$\frac{\partial E}{\partial w_i} = [y-\hat{y}(w_i)] f'(h) x_i$$

$$\Delta w_i = a(y-\hat{y}) f'(h) x_i$$

- DEFINE AN "ERROR TERM"

$$\delta = (y - \hat{y}) f'(h)$$

$$w_i = w_i + \eta \delta x_i$$

Weight Initialization

Trying to start w/ the optimal, initial weights values, makes things easier.

When doing Transfer learning.

- We already have the pre-trained. best wts.
- Initialize w/ best wts.

When doing a model not pre-trained:

- Initialize w/ 0
- Initialize w/ large wts
- Initialize w/ random values

CASE 1. [All 0s or 1s, or constant weights]

In the case of constant wts.

- Hard to identify error sources during backprop.
- Backprop is not designed to consistency

CASE 2: Randomness / Unique #s

- Want model to learn from their mistakes

Uniform Distribution

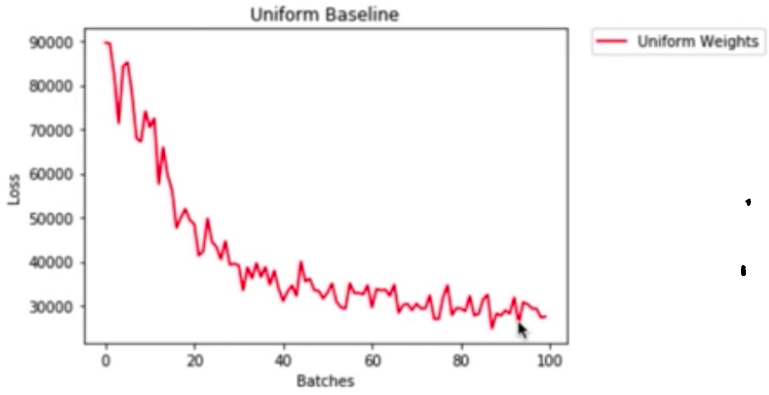
A uniform distribution has the equal probability of picking any number from a set of numbers. We'll be picking from a continuous distribution, so the chance of picking the same number is low. We'll use NumPy's `np.random.uniform` function to pick random numbers from a uniform distribution.

```
np.random_uniform(low=0.0, high=1.0, size=None)
```

Outputs random values from a uniform distribution.

The generated values follow a uniform distribution in the range [low, high). The lower bound minval is included in the range, while the upper bound maxval is excluded.

- **low:** The lower bound on the range of random values to generate. Defaults to 0.
- **high:** The upper bound on the range of random values to generate. Defaults to 1.
- **size:** An int or tuple of ints that specify the shape of the output array.

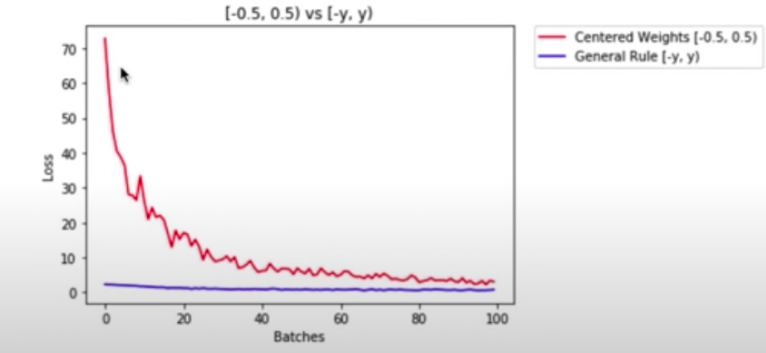



```
After 2 Epochs:
Validation Accuracy
   33.200% -- Uniform Weights
```

- Accuracy: 10% → 33.2%
- Values (0, 1) is not optimal

General Rule

- The more inputs a certain node sees, the less the wts should be, since wts act as multipliers.
- Set them close to 0, w/out being to small
- Range $[y, -y]$, $y = \frac{1}{\sqrt{n}}$




```
After 2 Epochs:
Validation Accuracy
   74.758% -- Centered Weights [-0.5, 0.5)
   85.817% -- General Rule [-y, y)
```

Normal Distribution

- Higher chance of picking values close to 0!

Plot Twist:

- Every PyTorch Layer has already wt. initialization implemented.
- Uniform distribution to be exact