# Lecture 6: Stochastic Gradient Descent (SGD)

Dr. Huiping Cao

## Outline

- Stochastic gradient descent (SGD) algorithm
- Discussions
- Multiclass classification

## Motivation

- The Adaline method minimizes a cost function by taking a step in the opposite direction of a cost gradient. The cost gradient is calculated from the **whole training set**. This method is called **batch gradient descent**.
- For large dataset, running batch gradient descent can be computationally costly. A popular alternative is **stochastic gradient descent**.

# Major idea – weight update

- Batch GD

$$\Delta \mathbf{w} = \eta \sum_{i} (y^{(i)} - \phi(z^{(i)}))\, \mathbf{x}^{(i)}$$

- SGD

$$\Delta \mathbf{w} = \eta (y^{(i)} - \phi(z^{(i)}))\mathbf{x}^{(i)}$$

## SGD: convergence and optimality

- SGD can be considered as an approximation of gradient descent, it typically **convergences much faster**.
- The error surface is **noisier** than in batch gradient descent since each gradient is calculated based on a single training example.
- SGD does not reach the global minimum, but an area very close to it.

## SGD: data shuffle

- Present the training data in a random order
  - Iteration 1: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}$
  - Iteration 2: $\mathbf{x}^{(2)}, \mathbf{x}^{(4)}, \mathbf{x}^{(1)}, \dots \mathbf{x}^{(n)}, \dots, \mathbf{x}^{(3)}$
  - Etc.
- To obtain satisfying results via SGD.

## Implementation

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []

    for _ in range(self.n_iter):
        net_input = self.net_input(X)
        output = net_input
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)
    return self
```

```
for i in range(self.n_iter):
    X, y = self._shuffle(X, y)
    cost = []
    for xi, target in zip(X, y):
        cost.append(self._update_weights(xi, target))
    avg_cost = sum(cost) / len(y)
    self.cost_.append(avg_cost)
```

# Shuffle

```
for i in range(self.n_iter):
    X, y = self._shuffle(X, y)
    cost = []
    for xi, target in zip(X, y):
        cost.append(self._update_weights(xi, target))
    avg_cost = sum(cost) / len(y)
    self.cost_.append(avg_cost)
```

```
def _shuffle(self, X, y):
        r = self.rgen.permutation(len(y))
        return X[r], y[r]
```

**numpy.random.permutation**(*x*): Randomly permute a sequence, or return a permuted range.

**Example**:

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])

>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12])
```

If r = array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])

**X[r]:** a matrix formed by $\mathbf{x}^{(1)}, \mathbf{x}^{(7)}, \mathbf{x}^{(4)}, \ldots, \mathbf{x}^{(6)}$

## Weight update

```
for i in range(self.n_iter):
    X, y = self._shuffle(X, y)
    cost = []
    for xi, target in zip(X, y):
        cost.append(self._update_weights(xi, target))
    avg_cost = sum(cost) / len(y)
    self.cost_.append(avg_cost)
```

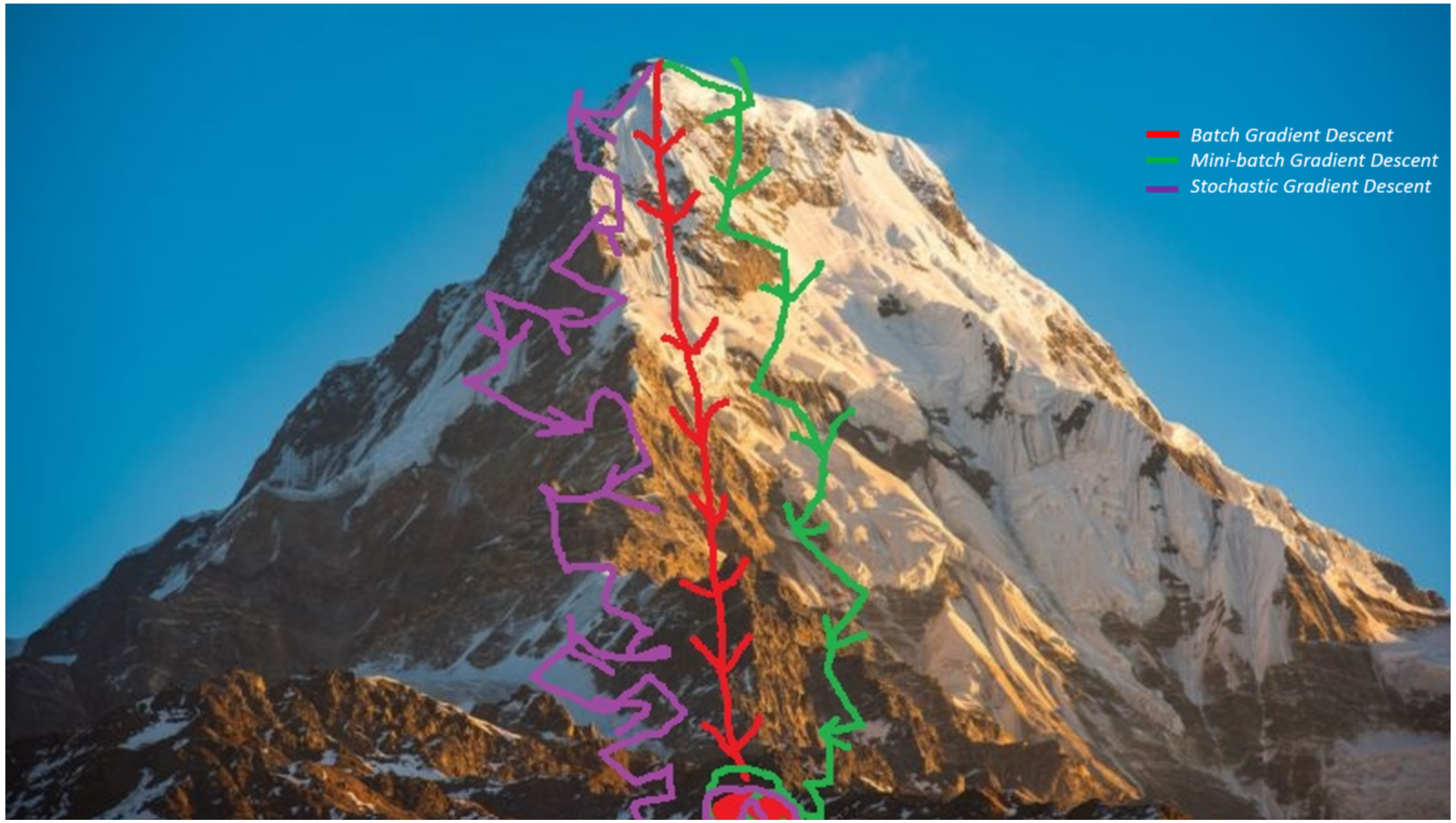
```
def _update_weights(self, xi, target):
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost
```

xi.dot(error) is a vector
$(\boldsymbol{x}_{\mathbf{1}}^{(i)}.error, \boldsymbol{x}_{\mathbf{2}}^{(i)}.error, \ldots, \boldsymbol{x}_{\boldsymbol{m}}^{(i)}.error)$

## Mini-batch gradient descent

- Mini-batch gradient descent: a compromise between batch gradient descent and SGD
- Apply batch gradient descent to smaller subsets of the training data.
- Advantage over batch GD: converge faster
- Advantage over SGD: computationally more efficient

# Gradient descent algorithms



Picture: courtesy of [https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3](https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3)

## Discussions: learning rate

- The learning rate is a **hyperparameter** that controls how much to change the model in response to the estimated error each time the model weights are updated.
- Controls the **rate or speed** at which a model learns
  - a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final set of weights
  - a smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train
- **Small positive value** in the range of (0, 1)

More reading: Understand the Impact of Learning Rate on Neural Network Performance

## SGD: Learning rate

- Important hyperparameter to configure
- Choosing the learning rate is **challenging**
  - A value too small may result in a long training process that could get stuck
  - A value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.
  - No analytically calculation can be used to decide this training rate.
- Tuning: trial and error
- Default value: 0.1 or 0.01

## SGD: Learning rate

- An alternative to using a fixed learning rate is to instead **vary the learning rate** over the training process.
- Learning rate schedule (**learning rate decay**): The way in which the learning rate changes over time (training epochs)
- **Adaptive learning rate**: the performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response.
    - Three popular approaches: AdaGrad, RMSProp, and Adam

More reading:
How to Configure the Learning Rate When Training Deep Learning Neural Networks
Adam — latest trends in deep learning optimization

## Multiclass classification

- One-versus-Rest (**OvR**), which is also called One-versus-All (**OvA**) techniques.
- **OvR**: fitting one classifier per class. A particular class is treated as the positive class and the samples from other classes are considered negative classes.
  - If we were to classify a new data sample, we would use *n* classifiers, where n is the number of class labels, and assign the class label with the highest confidence to the particular sample.
  - Interpretability advantage: Since each class is represented by one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier.
  - Most commonly used strategy and a fair default choice.
- Most commonly used strategy for multiclass classification.

## Multiclass classification

- **One-versus-one**: constructs one classifier per pair of classes.
    - The classifier for class labels i and j only needs to be trained using instances with these two class labels.
- At prediction time, the class which receives the most votes is selected.
    - Tie
- Fit n* (n-1)/2 classifiers, each learning problem only involves a small subset of the data.
- This method may be advantageous for algorithms such as kernel algorithms which don’t scale well with the number of samples.