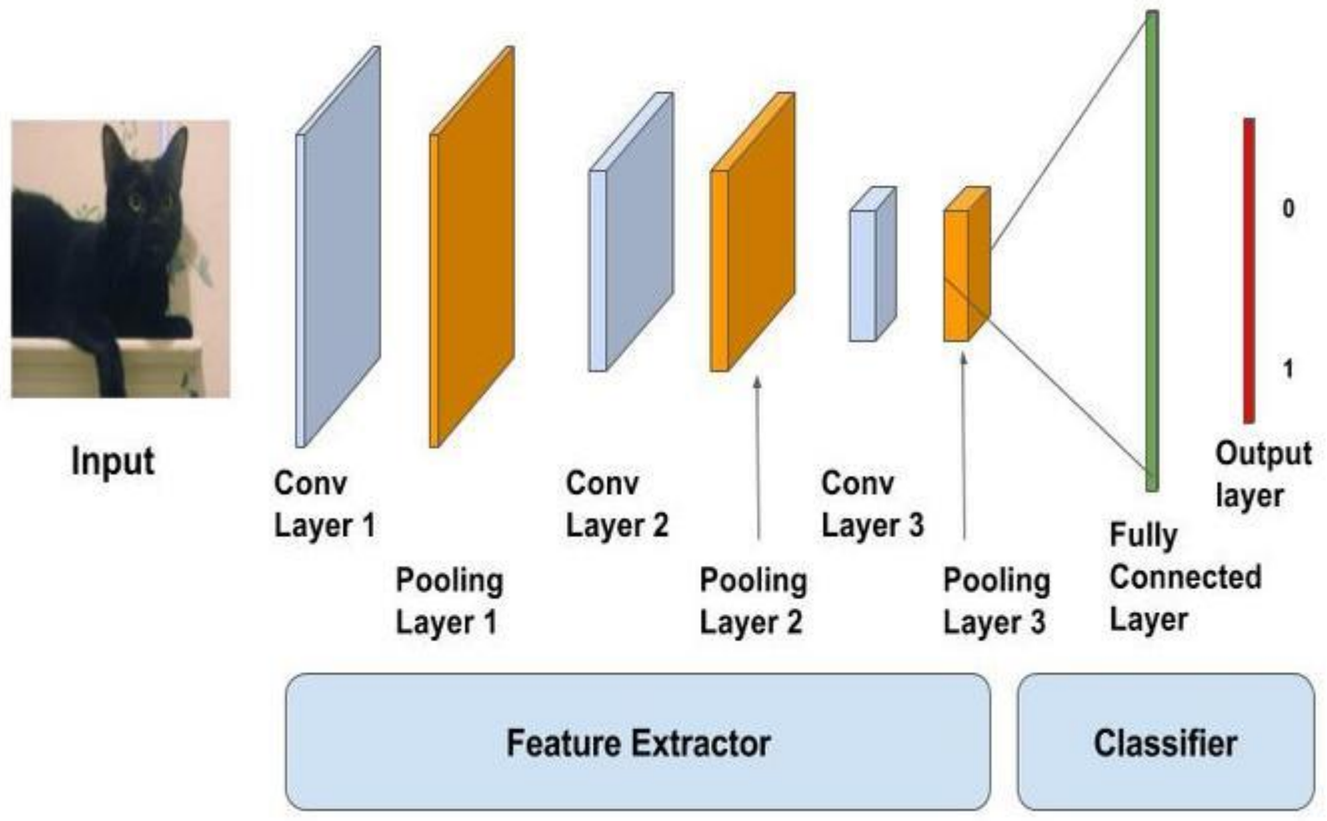
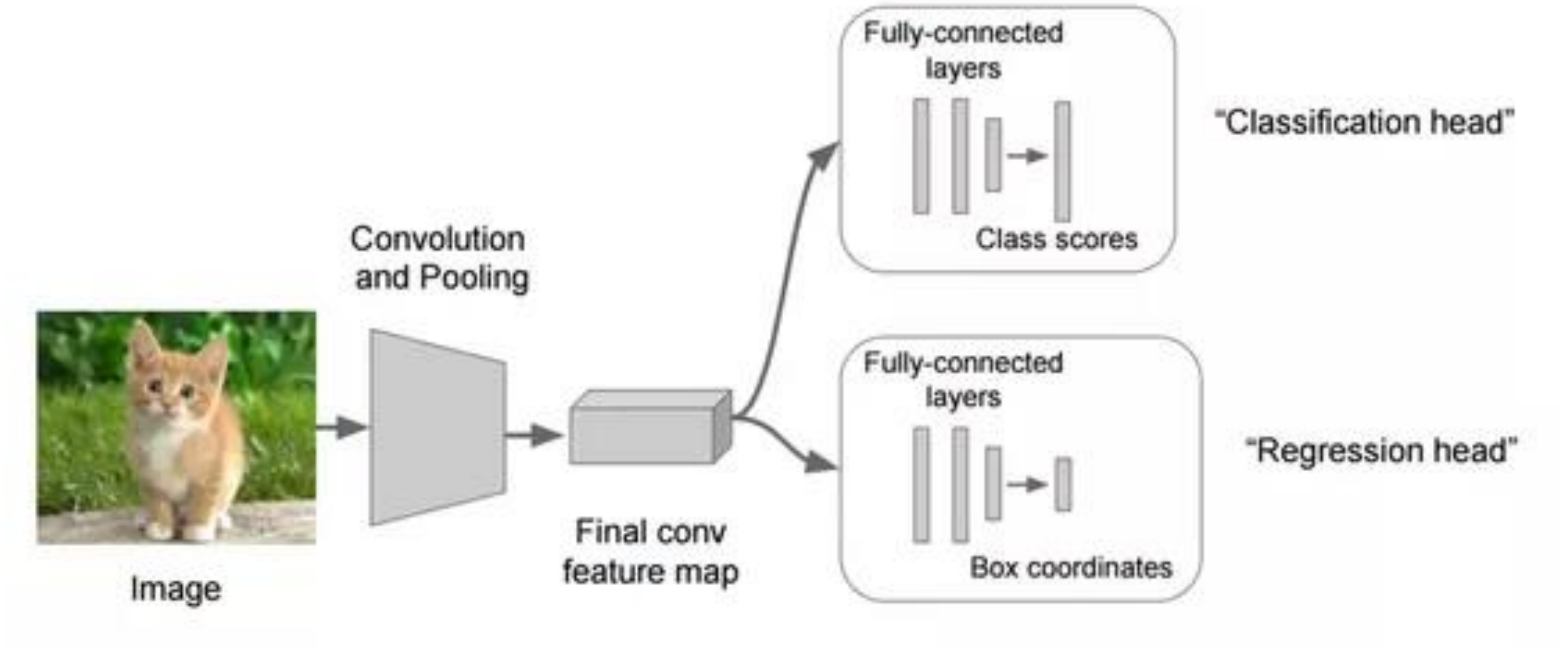
Loss functions and optimizers

Biplab Banerjee

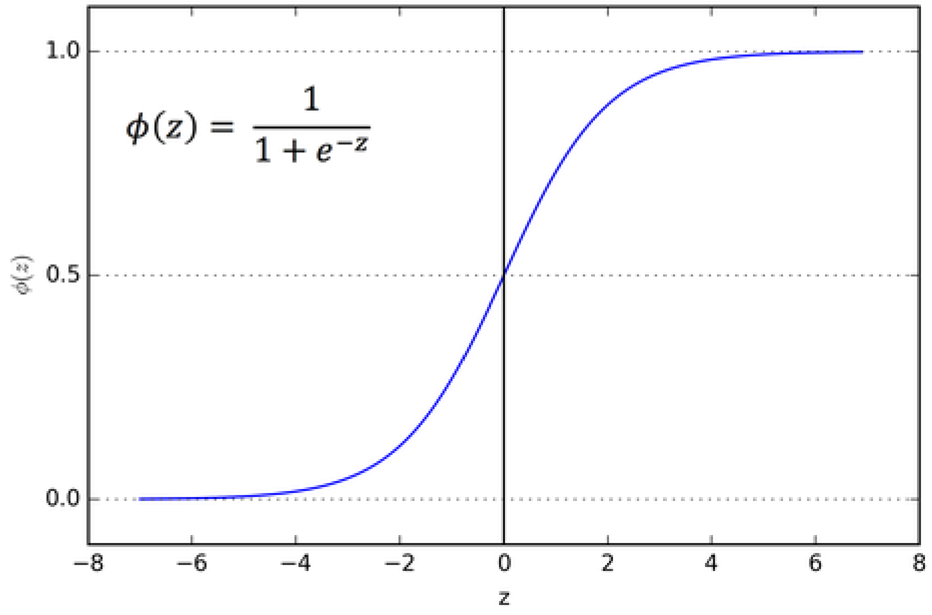
Vanilla CNN for classification

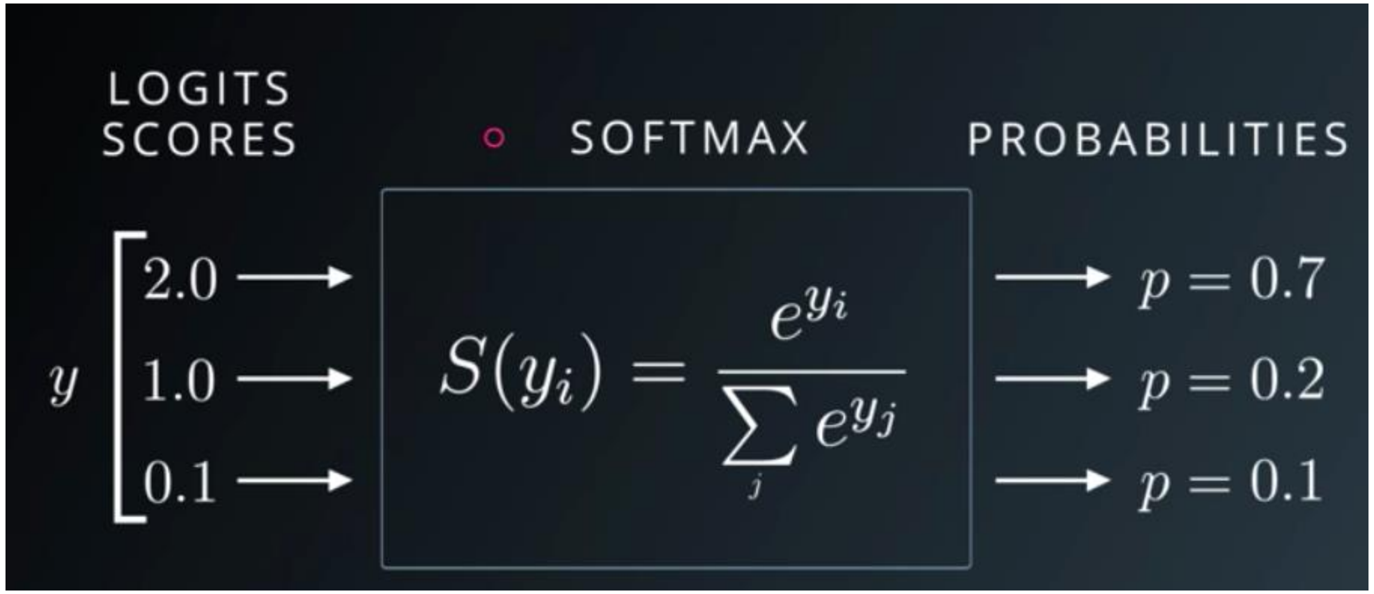


Vanilla CNN both for classification and regression – multi-task learning



Activation functions for classification





- ✓ Sigmoid and softmax – convert logits into probabilities
- ✓ It is a monotonic function but the gradient is not!

The softplus function is a smooth approximation to the ReLU activation function, and is sometimes used in the neural networks in place of ReLU.

$$\mathrm{softplus}(x) = \log(1 + e^{x})$$

It is actually closely related to the sigmoid function. As $x \to -\infty$, the two functions become identical.

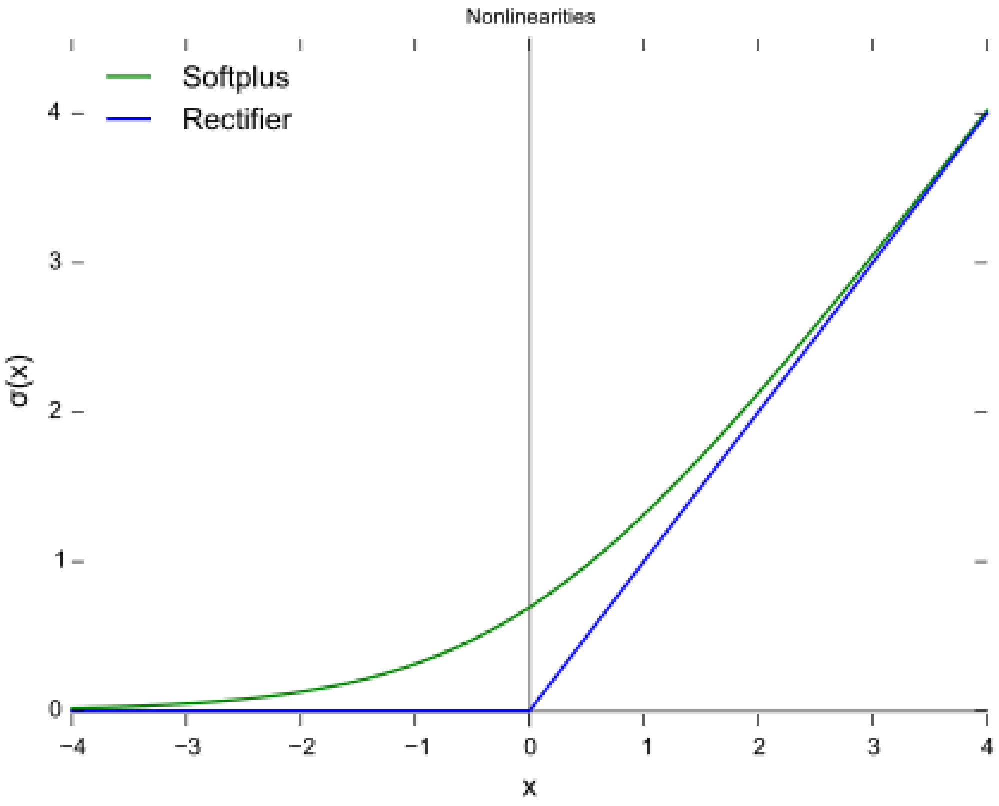
$$\mathrm{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

The softplus function also has a relatively unknown sibling, called softminus.

$$\mathrm{softminus}(x) = x - \mathrm{softplus}(x)$$

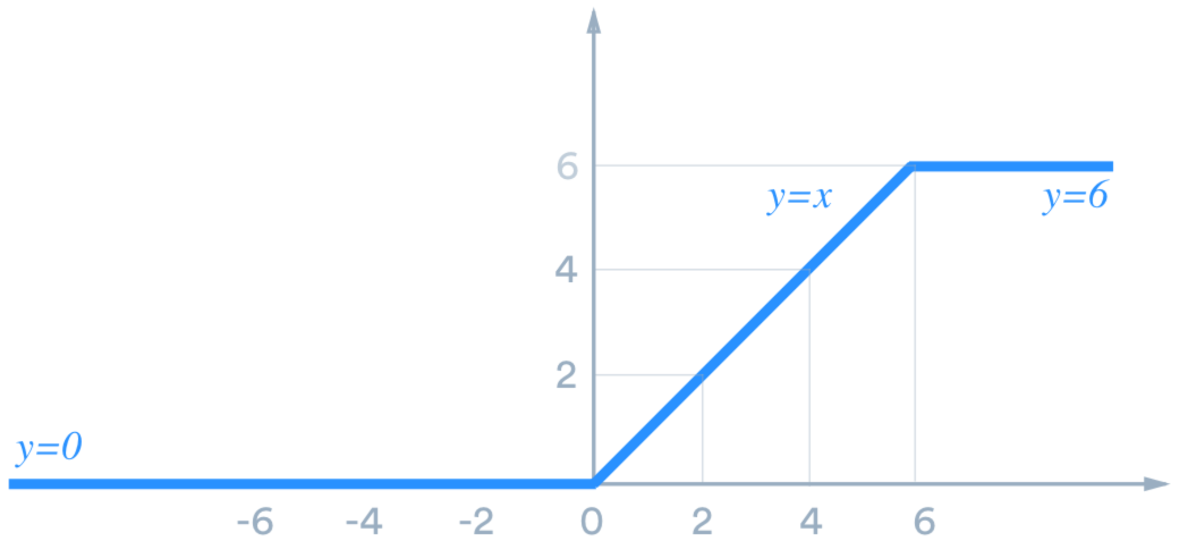
As $x \to +\infty$, it becomes identical to $\mathrm{sigmoid}(x) - 1$. In the following plots, you can clearly see the similarities between softplus & softminus and sigmoid.

Softplus and ReLU



Activation function for regression

- Depends upon the continuous target value we want to estimate
- Based on the range, it could be sigmoid, tanh, or relu
- ReLU has the problem of "dying out"
- Some variants: Leaky ReLU, Parametric ReLU, Concatenated ReLU, ReLU6 etc.



Loss functions

1. Regression Loss Functions
    1. Mean Squared Error Loss
    2. Mean Squared Logarithmic Error Loss
    3. Mean Absolute Error Loss
2. Binary Classification Loss Functions
    1. Binary Cross-Entropy
    2. Hinge Loss
    3. Squared Hinge Loss
3. Multi-Class Classification Loss Functions
    1. Multi-Class Cross-Entropy Loss
    2. Sparse Multiclass Cross-Entropy Loss
    3. Kullback Leibler Divergence Loss

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} (log(y_i + 1) - log(\hat{y}_i + 1))^2$$

•**Relative Differences:** MSLE focuses on the relative differences between the true and predicted values. The log transformation means that the loss is more sensitive to the ratio between the
•predicted and actual values rather than their absolute differences.
•**Reduced Sensitivity to Outliers:** The logarithm compresses the scale of large values, which helps in reducing the impact of outliers.
•**Scale Invariance:** MSLE is often used when the target variable spans several orders of magnitude or when percentage errors are more relevant than absolute errors.

True value	Predicted value	MSE loss	MSLE loss
30	20	100	0.02861
30000	20000	100 000 000	0.03100
	Comment	big difference	small difference

True value	Predicted value	MSE loss	MSLE loss
20	10 (underestimate by 10)	100	0.07886
20	30 (overestimate by 10)	100	0.02861
	Comment	no difference	big difference

Huber loss – regression problem with outliers

Quadratic

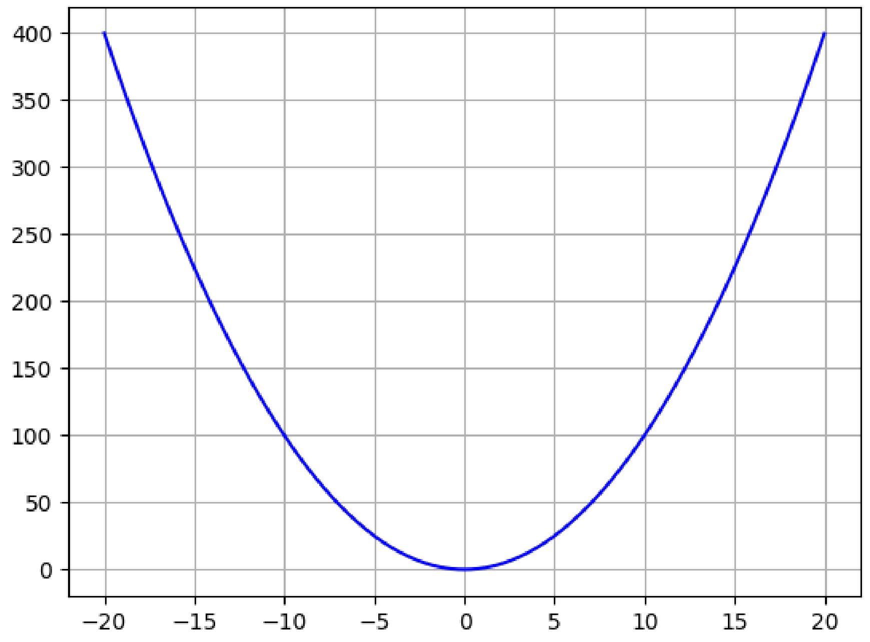
$$L_\delta = \begin{cases} \frac{1}{2}\left(y - f(x)\right)^2, & if\ |y - f(x)| \le \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & otherwise \end{cases}$$

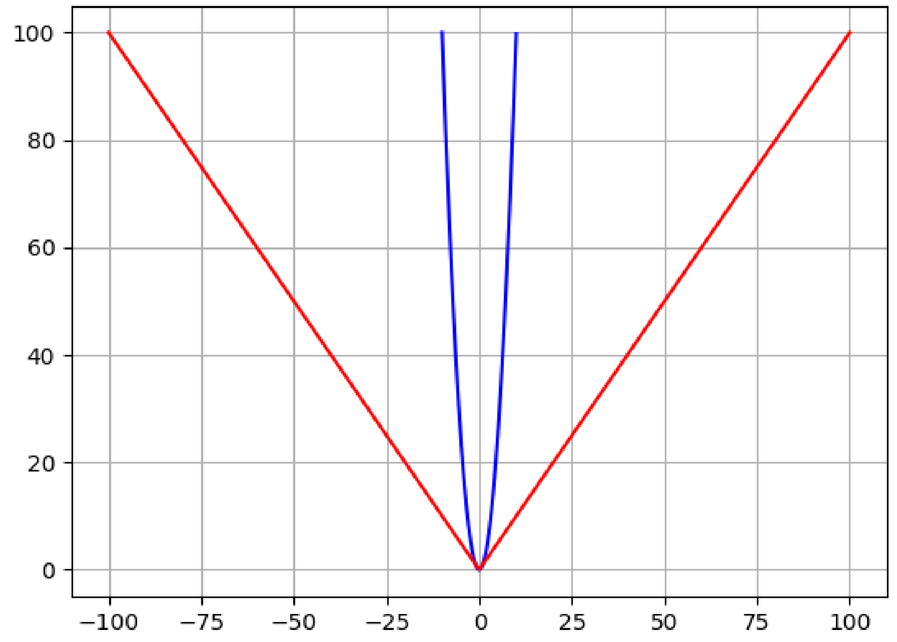
Linear

Combines the best of MSE and MAE

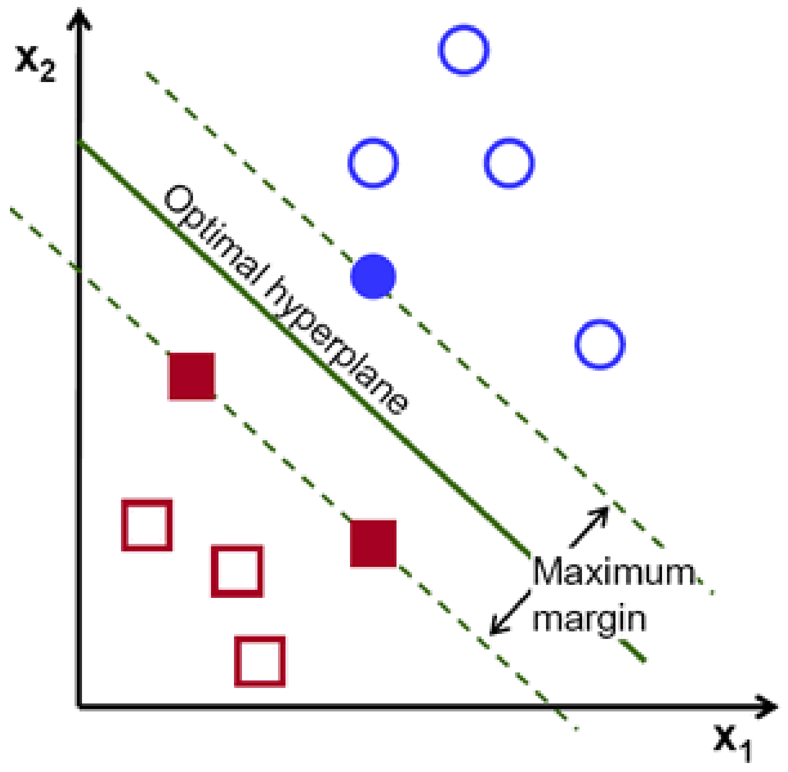
$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$





Hinge loss – margin based loss



$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max\left(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta\right)\right] + \lambda \sum_k \sum_l W_{k,l}^2$$

- ✓ Ensures better between-class separation
- ✓ \Delta controls the margin

Image retrieval

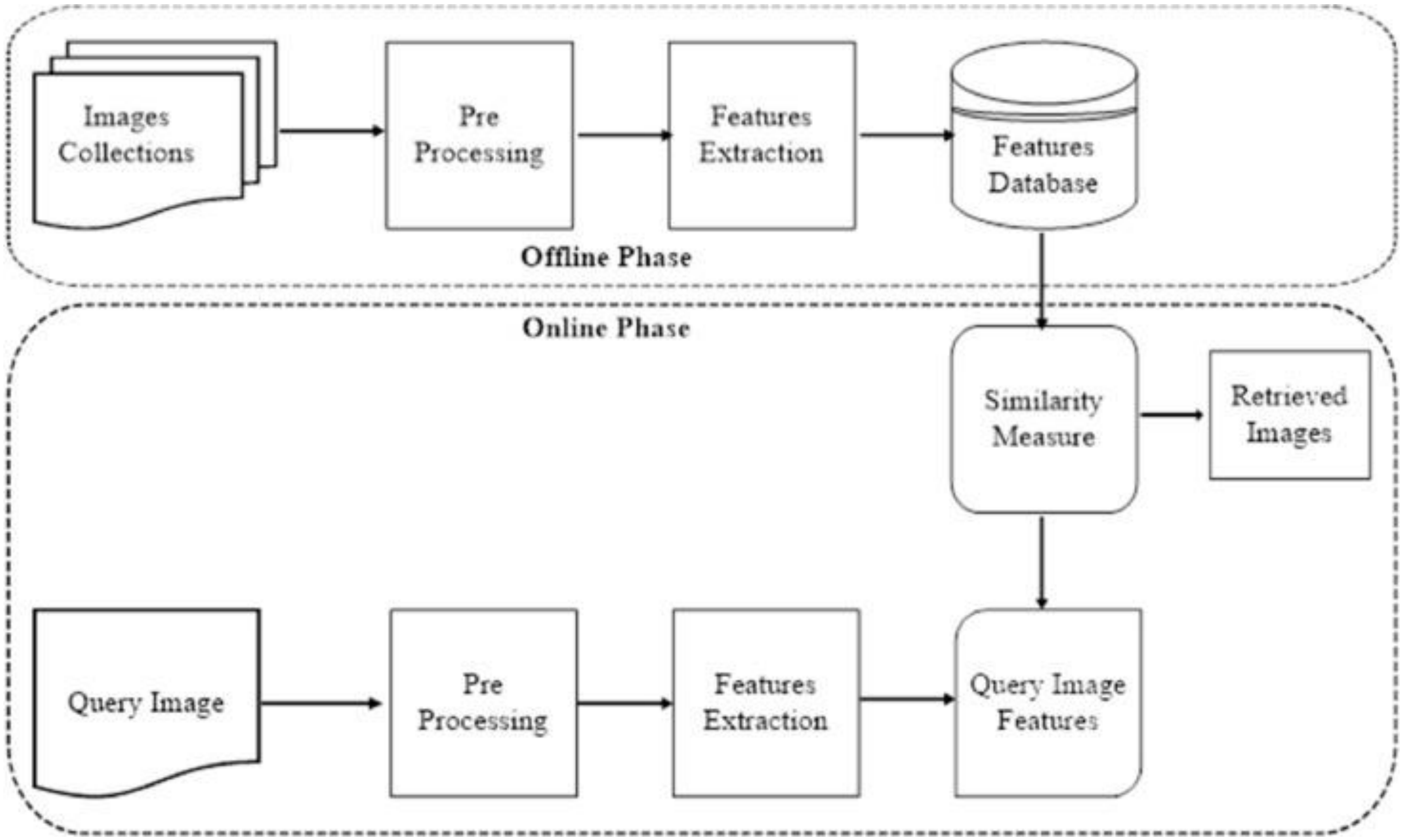
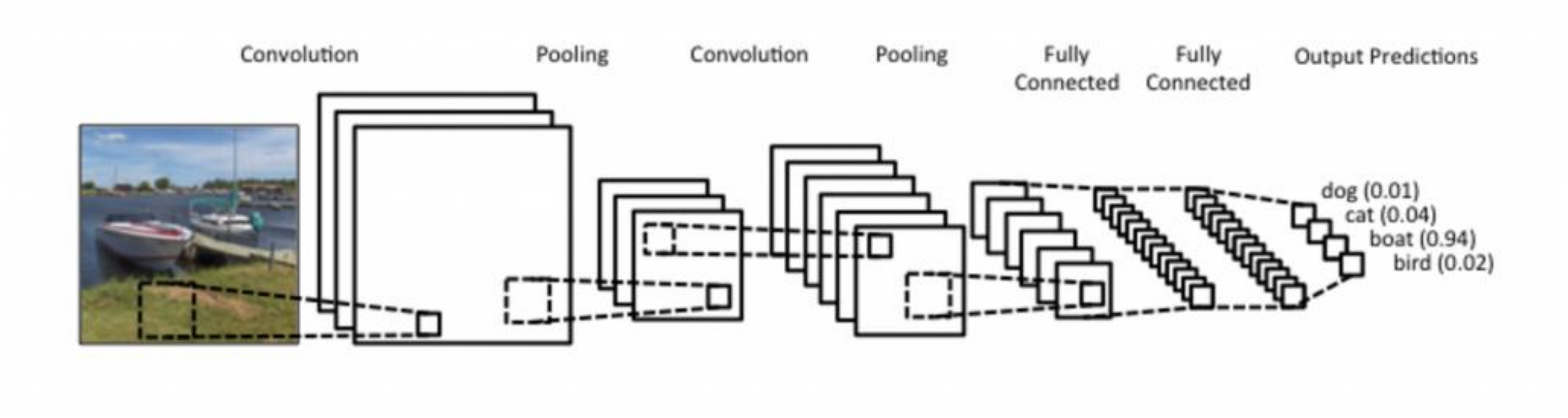


Image retrieval - what do we expect?

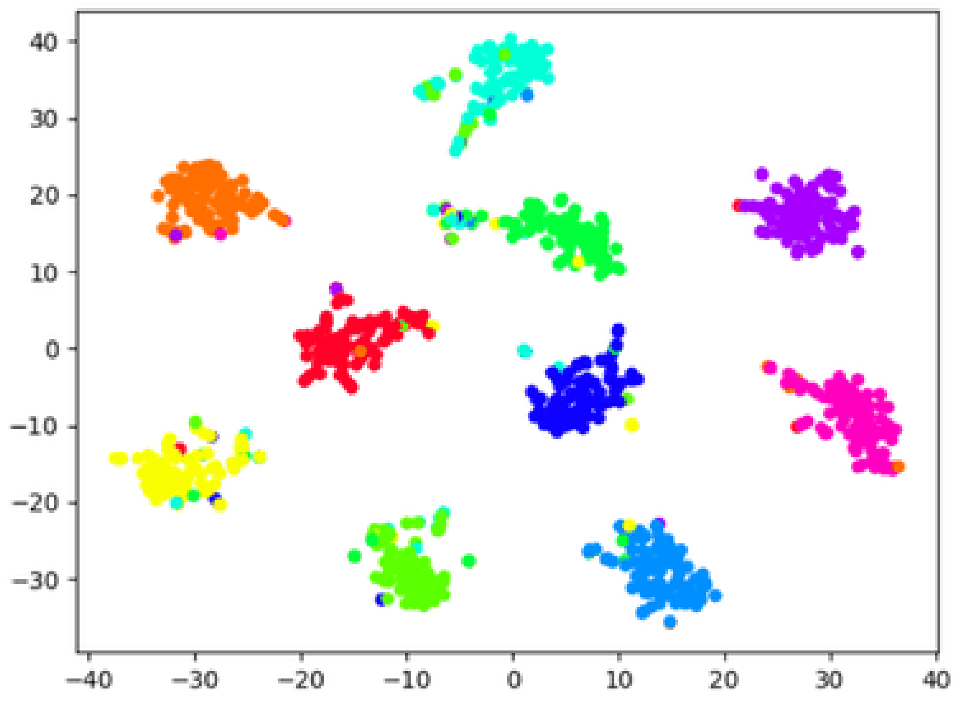
- Images of a given class should be compact in the feature space
- Images of different classes should be far apart
- For a given query image, there should be a ranking of all the possible retrieved images as per similarity

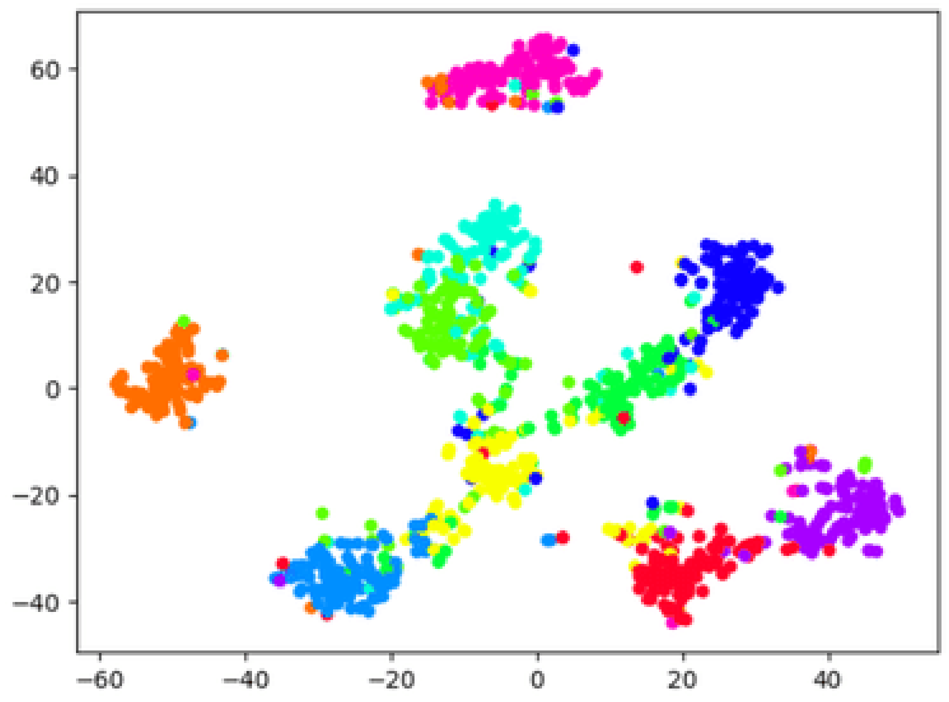
What is the feature vector corresponding to the image?



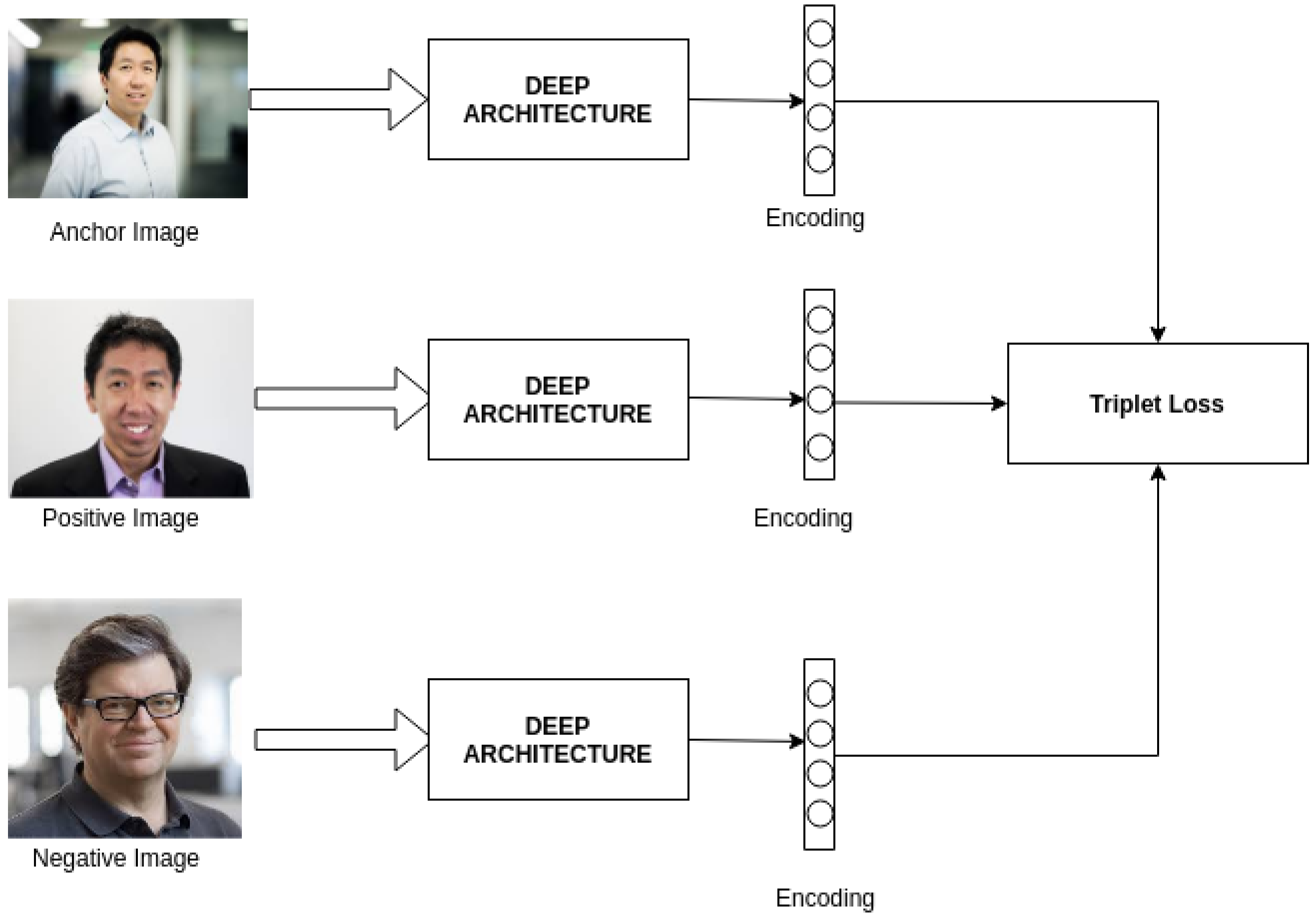
Similar images should have similar type features
Cosine similarity of such features will be high than features of two different category images

Which is a better feature space?

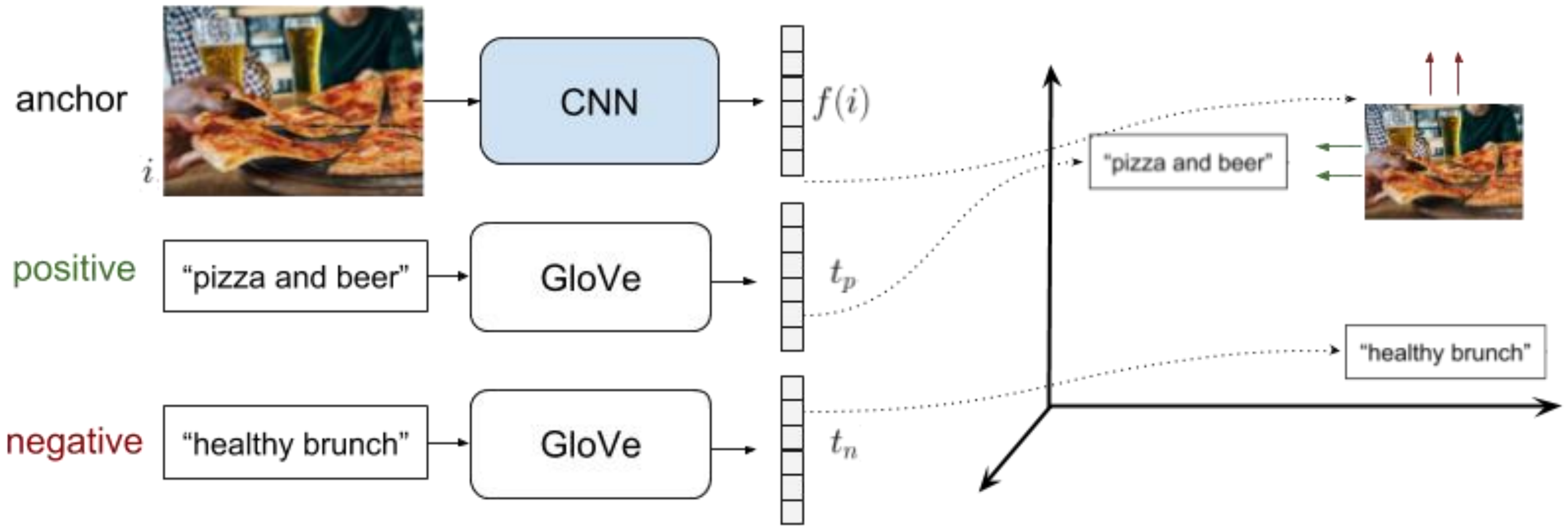




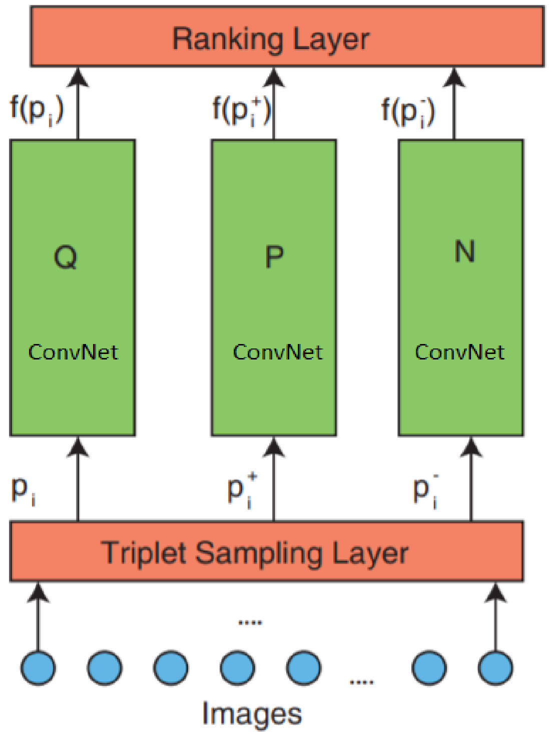
Metric loss - CBIR



Metric loss - TBIR



Triplet network



What will be ranking loss here?

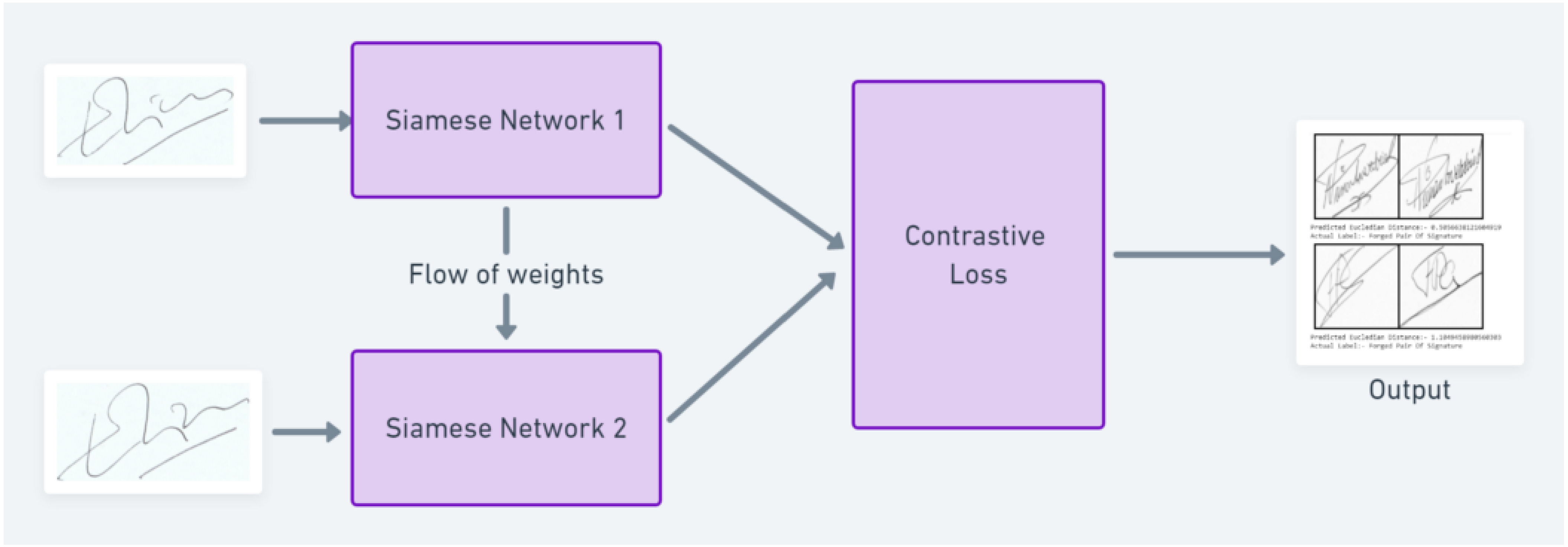
$$D(f(p_i), f(p_i^+)) < D(f(p_i), f(p_i^-)),$$
$$\forall p_i, p_i^+, p_i^- \text{ such that } r(p_i, p_i^+) > r(p_i, p_i^-)$$

$$l(p_i, p_i^+, p_i^-) = \max\{0, g + D(f(p_i), f(p_i^+)) - D(f(p_i), f(p_i^-))\}$$

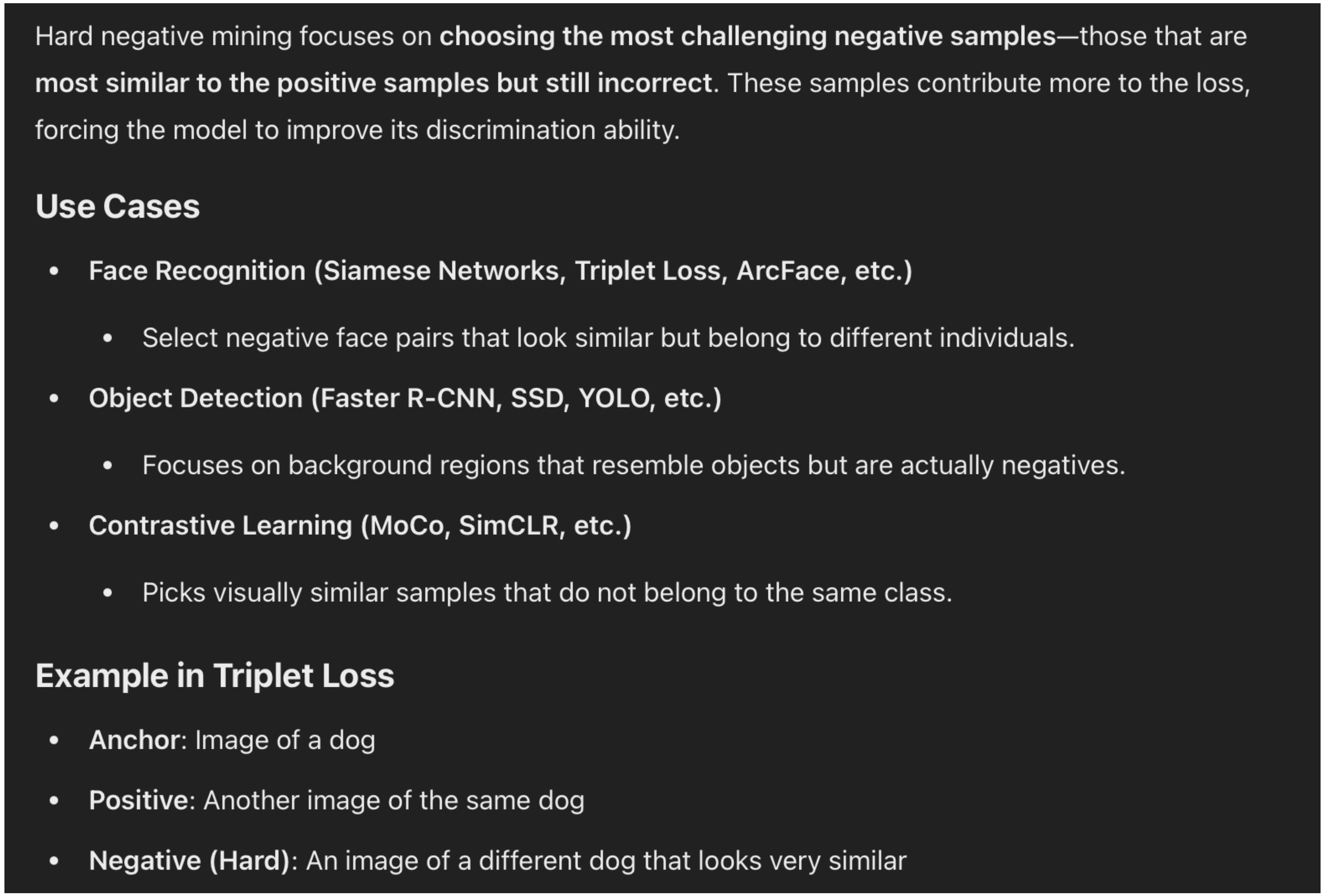
Siamese loss

Y=0 for similar and 1 for dissimilar pairs

$$(1-Y)\frac{1}{2}(D_W)^2 + (Y)\frac{1}{2}\{max(0, m - D_W)\}^2$$



Negative mining in metric learning



Hard negative mining focuses on **choosing the most challenging negative samples**—those that are **most similar to the positive samples but still incorrect**. These samples contribute more to the loss, forcing the model to improve its discrimination ability.

Use Cases

- **Face Recognition (Siamese Networks, Triplet Loss, ArcFace, etc.)**
  - Select negative face pairs that look similar but belong to different individuals.
- **Object Detection (Faster R-CNN, SSD, YOLO, etc.)**
  - Focuses on background regions that resemble objects but are actually negatives.
- **Contrastive Learning (MoCo, SimCLR, etc.)**
  - Picks visually similar samples that do not belong to the same class.

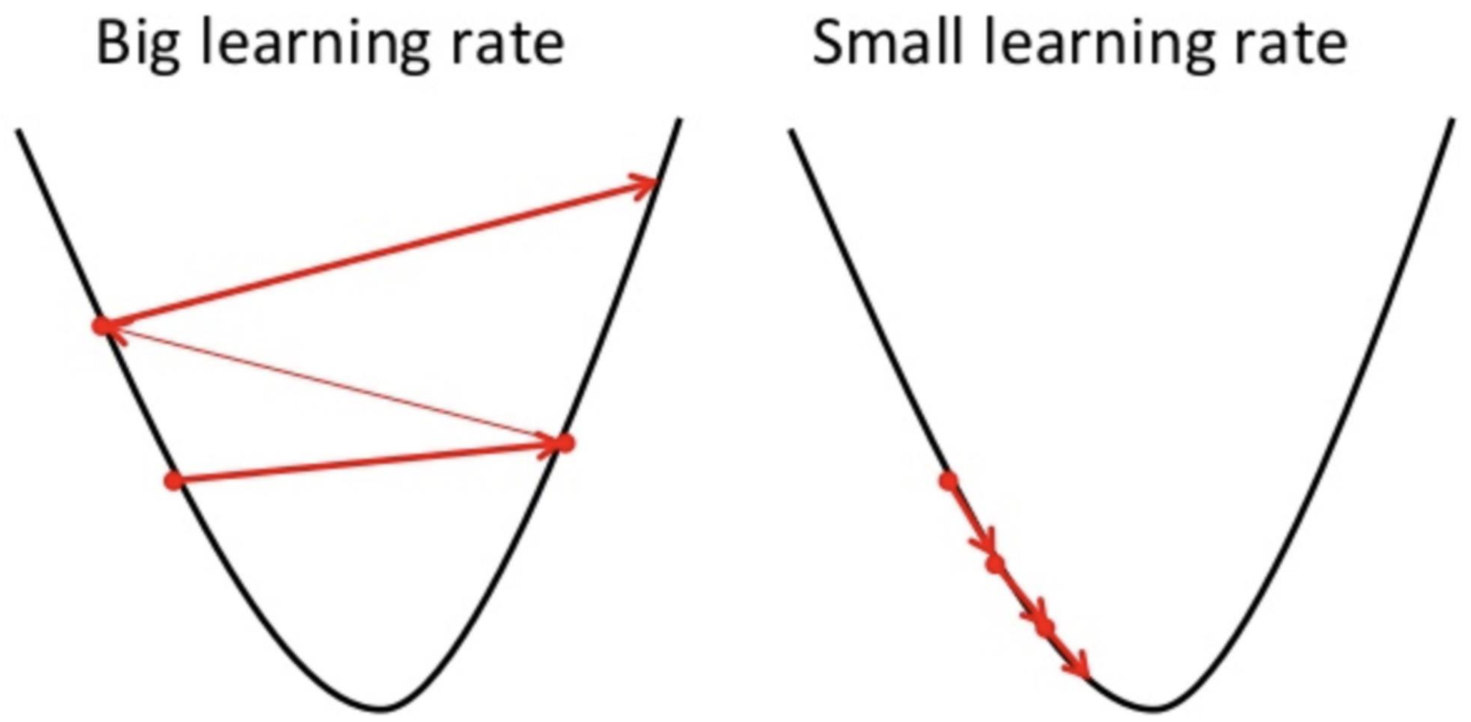
Example in Triplet Loss

- **Anchor**: Image of a dog
- **Positive**: Another image of the same dog
- **Negative (Hard)**: An image of a different dog that looks very similar

Some advanced optimizers

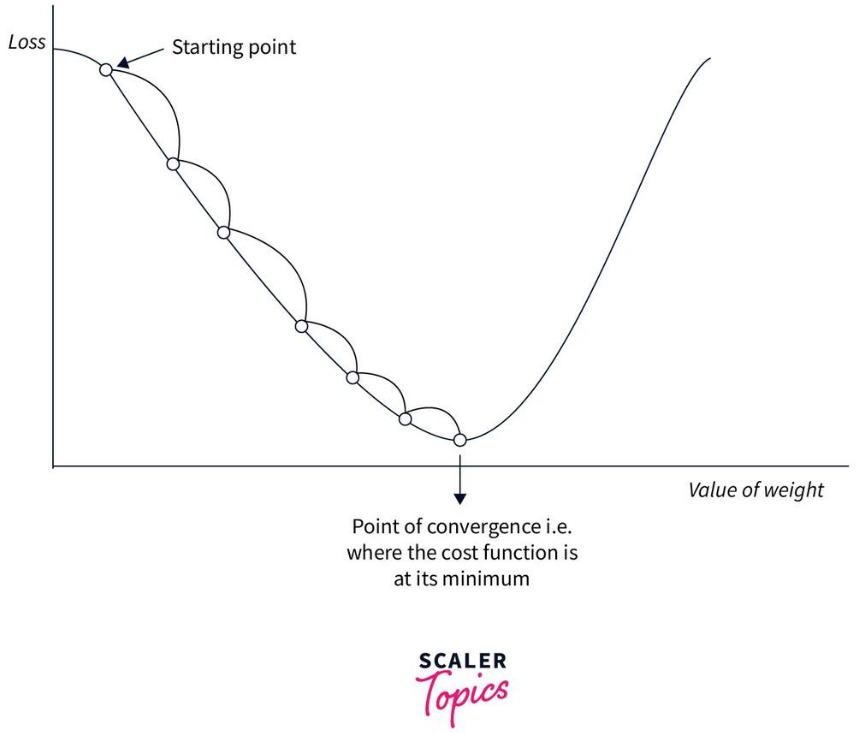
- Adam
- Adagrad
- Adadelta
- RMSProp
- SGD with momentum
- And many more…

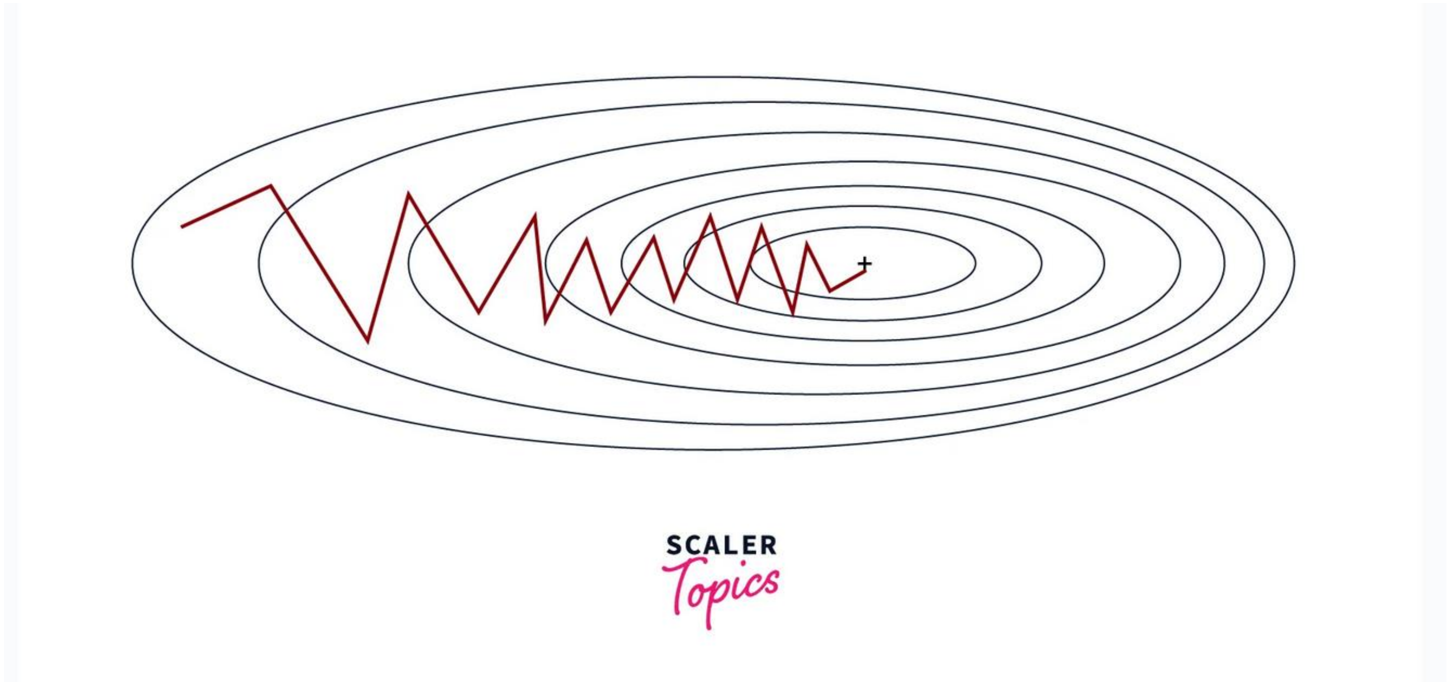
Problem in selecting the learning rate



Batch vs stochastic GD

The zigzag pattern causes delay in convergence

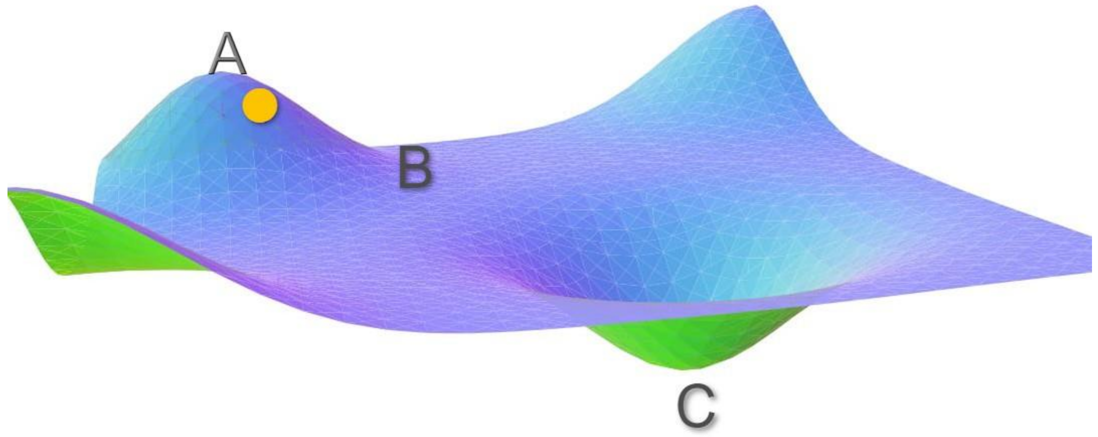

Loss
Starting point
Value of weight
Point of convergence i.e. where the cost function is at its minimum
SCALER Topics



$$\theta = \theta - \alpha \cdot \nabla_\theta L(\theta)$$

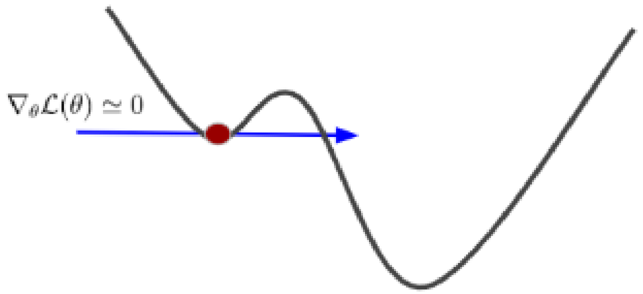
$$\theta = \theta - \alpha \cdot \nabla_\theta L(\theta; x^{(i)}; y^{(i)})$$

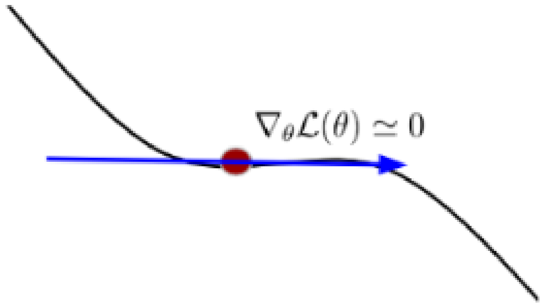
Loss landscape



Local optima, Global Optima, Saddle point

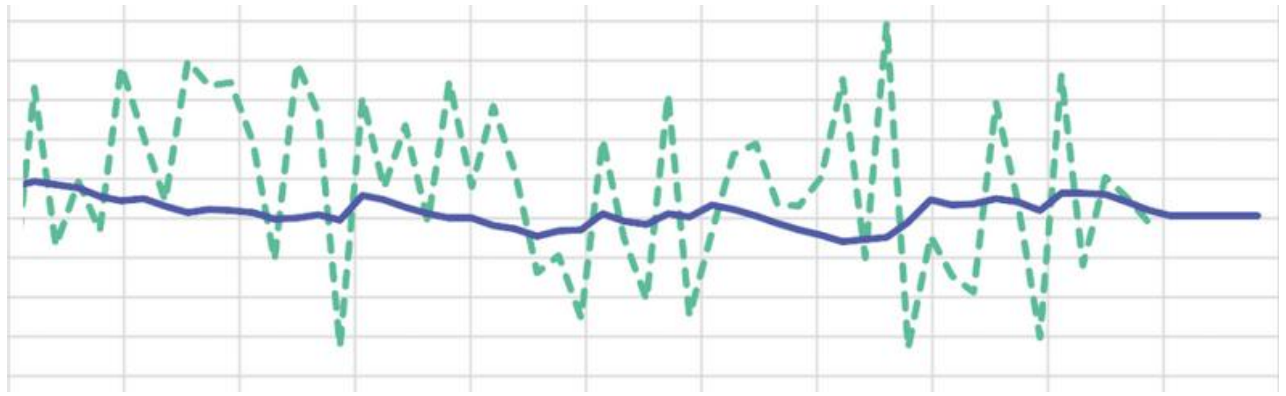
The local optimal and saddle points cause the optimizer escape the global optima





Idea of momentum, Moving average

In classical physics, momentum increases the speed of an object and reduces its likelihood to change directions. Similarly, in optimization, momentum helps the gradient descent algorithm to push over flat areas of the loss surface more quickly and dampen oscillations in the steep directions.

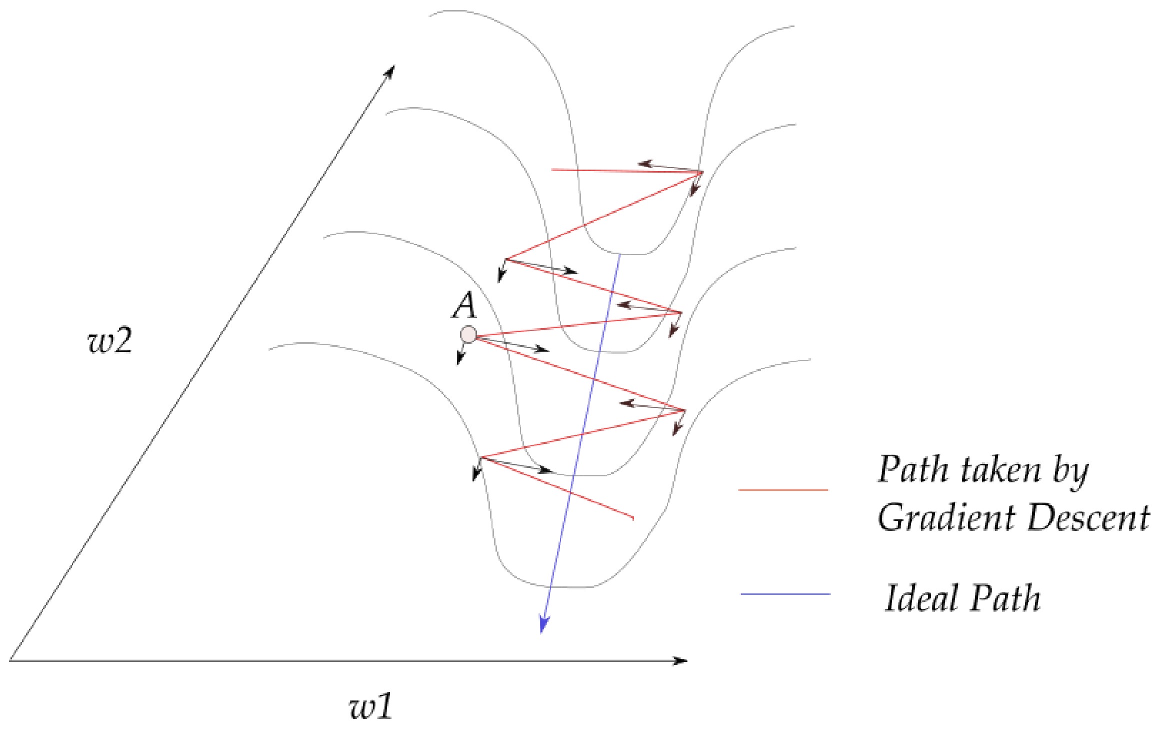


$$EWMA_t = \alpha \cdot x_t + (1 - \alpha) \cdot EWMA_{t-1}$$

Where:

- x_t is the input value at time t,
- $EWMA_{t-1}$ is the previous EWMA value,
- α is the smoothing factor, and it lies between 0 and 1.

Momentum



1. **Velocity Update**:

$$v_t = \gamma v_{t-1} + \eta \nabla f(\theta_{t-1})$$

2. **Parameter Update**:

$$\theta_t = \theta_{t-1} - v_t$$

Where:

- θ represents the parameters to be optimized.
- v is the velocity vector, which accumulates the weighted average of past gradients.
- η is the learning rate.
- γ is the momentum coefficient, typically set between 0.9 and 0.99.
- $\nabla f(\theta_{t-1})$ is the gradient of the function to be minimized, evaluated at the parameter values from the previous step.

Key Features

- **Faster Convergence:** By using the weighted average of the gradients, momentum can lead to faster convergence, especially in the context of high-dimensional and non-convex optimization problems.
- **Reduction in Oscillation:** Momentum dampens oscillations in directions that consistently have high gradients (steep directions) by averaging out updates across iterations. This leads to smoother convergence paths.
- **Escaping Saddle Points**: The accumulation of gradients in the velocity vector helps in potentially escaping shallow saddle points more efficiently than standard gradient descent.

Momentum

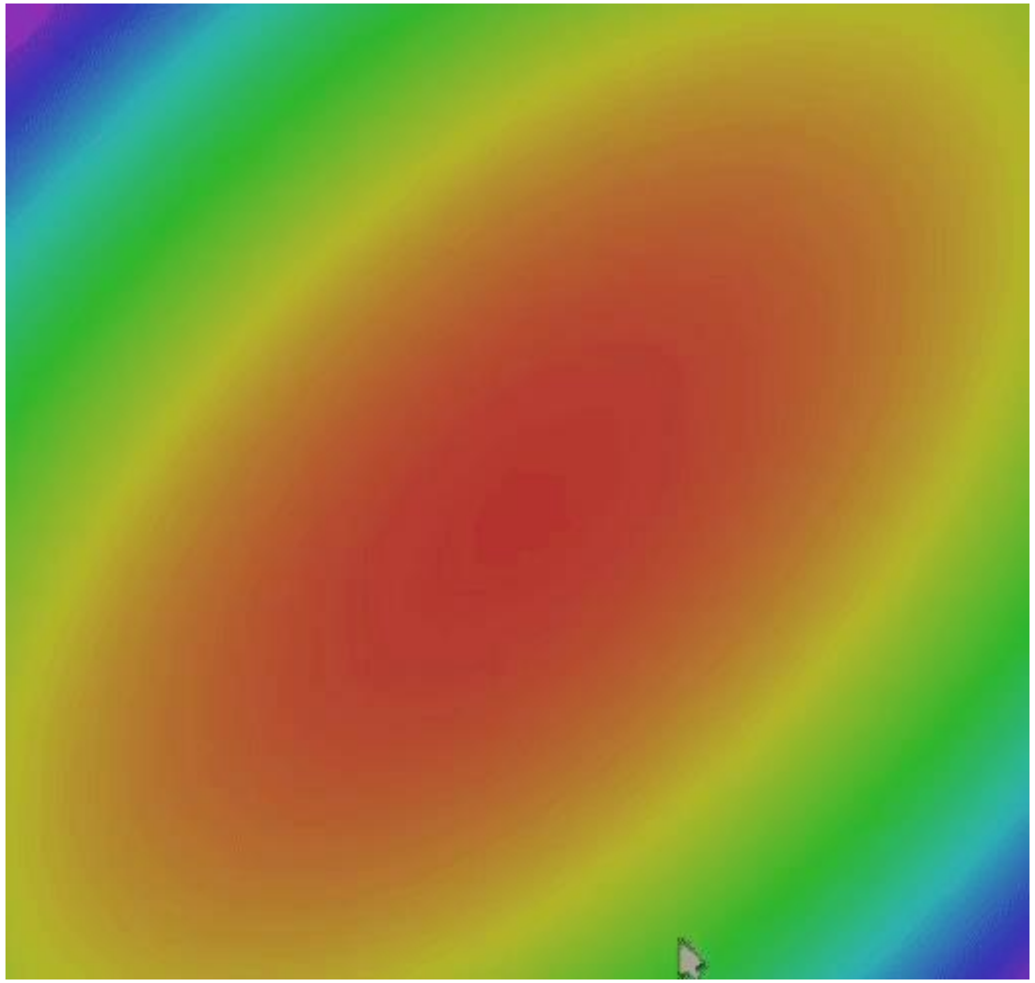
Example: 1. Let us consider the initial gradient is 0 and \nu=0.9

$$\nu_1 = -G_1$$

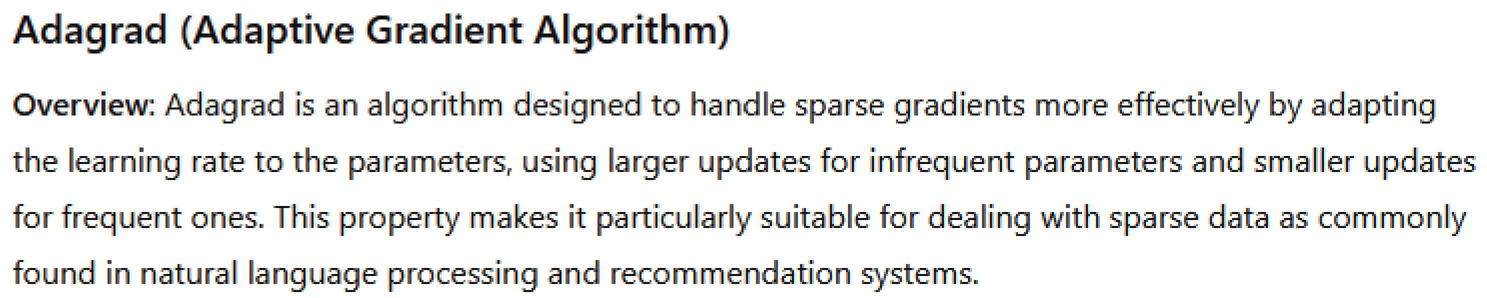
$$\nu_2 = -0.9 * G_1 - G_2$$

$$\nu_3 = -0.9 * (0.9 * G_1 - G_2) - G3 = -0.81 * (G_1) - (0.9) * G_2 - G_3$$

Essentially, there is exponential decay in the weights of the previous gradient terms!

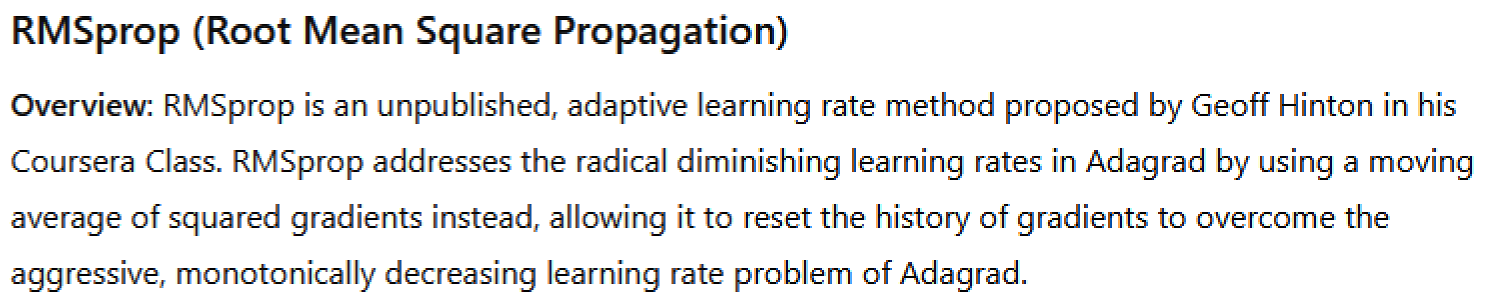


ADAGRAD / RMSPROP – scaling the learning rate



Adagrad (Adaptive Gradient Algorithm)

Overview: Adagrad is an algorithm designed to handle sparse gradients more effectively by adapting the learning rate to the parameters, using larger updates for infrequent parameters and smaller updates for frequent ones. This property makes it particularly suitable for dealing with sparse data as commonly found in natural language processing and recommendation systems.



RMSprop (Root Mean Square Propagation)

Overview: RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in his Coursera Class. RMSprop addresses the radical diminishing learning rates in Adagrad by using a moving average of squared gradients instead, allowing it to reset the history of gradients to overcome the aggressive, monotonically decreasing learning rate problem of Adagrad.

ADAGRAD

- **Gradient Accumulation:** First, Adagrad modifies the general learning rate at each time step, t, for every parameter θ_i based on the past gradients that have been computed for that parameter:

$$G_{t,ii} = G_{t-1,ii} + (\nabla_{t,\theta_i} f(\theta))^2$$

  Here, $G_{t,ii}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θ_i up to time step t.

- **Parameter Update:** The parameters are then updated using the following formula:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \nabla_{t,\theta_i} f(\theta)$$

  where η is a global learning rate, ϵ is a smoothing term that avoids division by zero (commonly set to $1e-8$), and $\nabla_{t,\theta_i} f(\theta)$ is the gradient of the objective function with respect to parameter θ_i at time t.

RMSPROP

- **Gradient Squared Exponential Decay:** Unlike Adagrad, RMSprop maintains a moving average of the squared gradients, and the update is modified as follows:

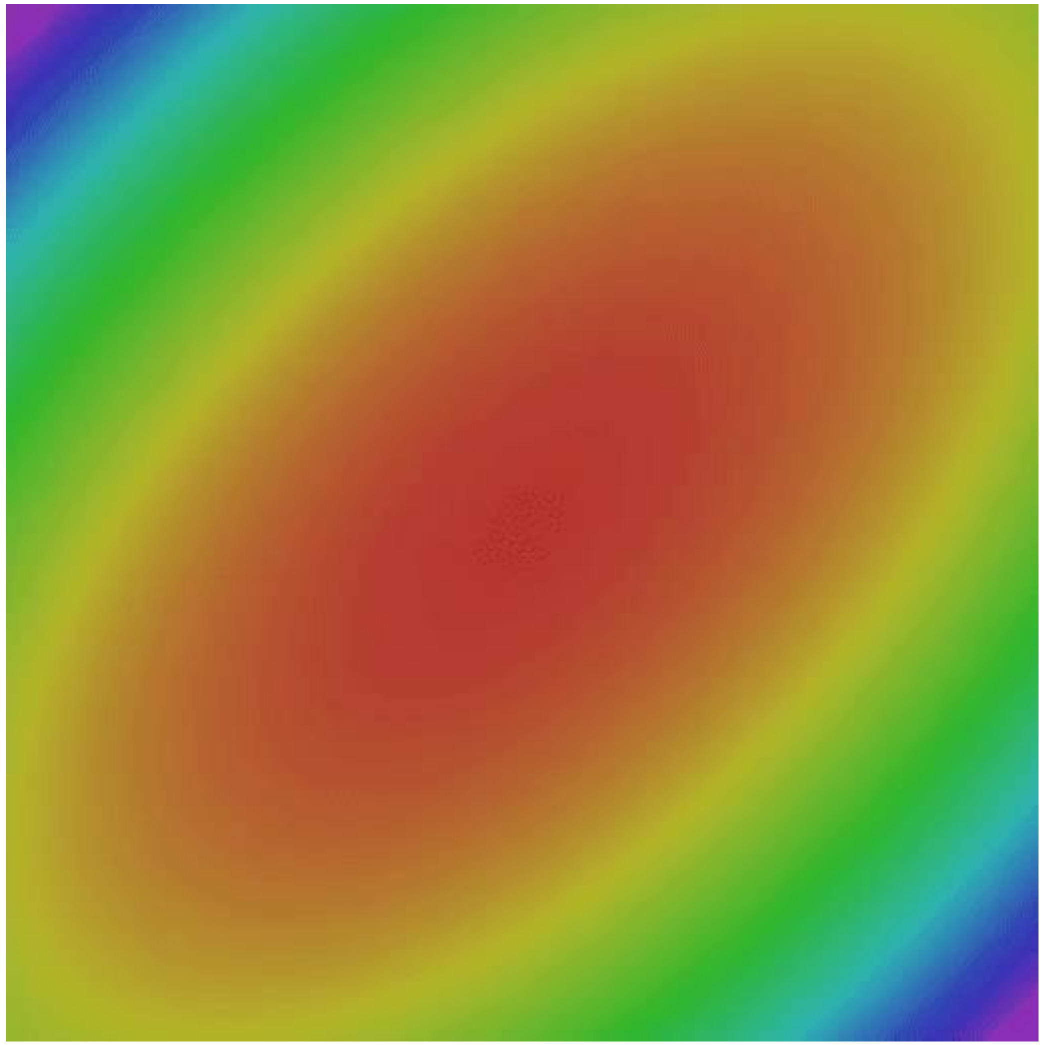
$$v_t = \gamma v_{t-1} + (1-\gamma)(\nabla_{t,\theta} f(\theta))^2$$

  where γ is a decay rate, typically around 0.9.

- **Parameter Update:** Parameters are updated using this moving average:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla_{t,\theta} f(\theta)$$

  Here, η is the learning rate, v_t is the moving average of the squared gradients, and ϵ is a small constant.



Adam

1. **First Moment Update (Mean of the gradients)**

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta J(\theta)$$

Motivation: This equation is an exponentially decaying average of past gradients. m_t approximates the first moment (the mean) of the gradients. The coefficient β_1 (typically set to 0.9) controls the decay rate and is akin to momentum in accelerating the gradients in the directions of steepest descent. This helps in smoothing out the updates and provides a directionally aware update that aids in faster convergence and stability in optimization.

2. **Second Moment Update (Uncentered variance of the gradients)**

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta))^2$$

Motivation: This equation keeps track of the squared gradients and computes an exponentially decaying average of past squared gradients. v_t approximates the second moment (the uncentered variance) of the gradients. The β_2 parameter (commonly set to 0.999) helps in controlling the decay rate, which is crucial for adaptive learning rate adjustments. It adjusts the learning rate for each parameter based on the historical squared gradients, preventing too large updates that could lead to divergence, particularly in the case of noisy gradients.

3. **Bias-Correction for First and Second Moments**

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Motivation: Initially, m_t and v_t are initialized as vectors of zeros, leading to biased estimates at the beginning of training (especially when β_1 and β_2 are close to 1). These bias-corrected terms, $\hat{m}_t$ and $\hat{v}_t$, counteract these biases by adjusting the estimates based on how many iterations have been performed. This correction ensures that the estimates of the first and second moments are more accurate, leading to more effective updates, especially during the early stages of training.

4. **Parameter Update Rule**

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Motivation: This is the actual parameter update rule where η represents the learning rate, and ϵ is a small constant added to improve numerical stability (preventing division by zero). This rule adjusts the parameters in a direction that minimizes the loss, where the size of the step is inversely proportional to the square root of the average of the squares of the gradients. This adaptive step size is beneficial for handling different scales in the data and parameter space effectively, allowing for larger updates for parameters associated with infrequent features and smaller updates for parameters associated with frequent features, which is particularly useful for sparse data.