



Walchand College of Engineering, Sangli

**Department
of**

**Computer Science and
Engineering**

TY CSE PE-2: Deep Learning 4CS335

Prof. Kiran P. Kamble

Module 2

Parameterized Learning and Optimization Methods

Outline

- Parameterized Learning:
 - Introduction to linear classification
 - Four components of parameterized learning
 - role of loss function.
- Optimization Methods:
 - Gradient descent
 - Stochastic gradient descent (SGD)
 - Extensions to SGD
 - Regularization

Parameterized Learning

- Machine learning model that can **learn patterns from our input data** during training time (requiring us to spend more time on the training process), but have the benefit of being defined by a *small number of parameters* that can easily be used to represent the model, regardless of training size. This type of machine learning is called **parameterized learning**,

Parameterized Learning

“A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a parametric model. No matter how much data you throw at the parametric model, it won’t change its mind about how many parameters it needs.” – Russell and Norvig (2009) [78]

Introduction to Linear Classification

- Four Components of Parameterized Learning
 - In the task of machine learning & DL, parameterization involves defining a problem in terms of four key components: **data**, a **scoring function**, a **loss function**, and **weights** and **biases**.

Data

- This component is our input data that we are going to learn from
- Includes both the **data points** (*i.e., raw pixel intensities from images, extracted features, etc.*) and their **associated class labels**.
- Typically, we denote our data in terms of a multi-dimensional **design matrix**

Data

Each row in the design matrix represents a data point while each column (which itself could be a multi-dimensional array) of the matrix corresponds to a different feature. For example, consider a dataset of 100 images in the RGB color space, each image sized 32×32 pixels. The design matrix for this dataset would be $X \subseteq R^{100 \times (32 \times 32 \times 3)}$ where X_i defines the i -th image in R . Using this notation, X_1 is the first image, X_2 the second image, and so on.

Along with the design matrix, we also define a vector y where y_i provides the class label for the i -th example in the dataset.

Scoring Function

The scoring function accepts our data as an input and maps the data to class labels. For instance, given our set of input images, the scoring function takes these data points, applies some function f (our scoring function), and then returns the predicted class labels, similar to the pseudocode below:

INPUT_IMAGES \Rightarrow F(INPUT_IMAGES) \Rightarrow OUTPUT_CLASS_LABELS

Loss Function

A loss function quantifies how well our *predicted class labels* agree with our *ground-truth labels*. The higher level of agreement between these two sets of labels, the *lower our loss* (and higher our classification accuracy, at least on the training set).

Our goal when training a machine learning model is to *minimize the loss function*, thereby increasing our classification accuracy.

Weights and Biases

The weight matrix, typically denoted as W and the bias vector b are called the **weights** or **parameters** of our classifier that we'll actually be optimizing. Based on the output of our scoring function and loss function, we'll be tweaking and fiddling with the values of the weights and biases to increase classification accuracy.

Next, let's look at how these components **can work together** to build a linear classifier, transforming the input data into actual predictions.

Linear Classification: From Images to Labels

To start, we need our *data*. Let's assume that our training dataset is denoted as x_i where each image has an associated class label y_i . We'll assume that $i = 1, \dots, N$ and $y_i = 1, \dots, K$, implying that we have N data points of dimensionality D , separated into K unique categories.

To make this idea more concrete, consider our “Animals” dataset

In this dataset, we have $N = 3,000$ total images. Each image is 32×32 pixels, represented in the RGB color space (i.e., three channels per image).

We can represent each image as $D = 32 \times 32 \times 3 = 3,072$ distinct values.

There are a total of $K = 3$ class labels one for the dog, cat, and panda classes, respectively.

Linear Classification: From Images to Labels

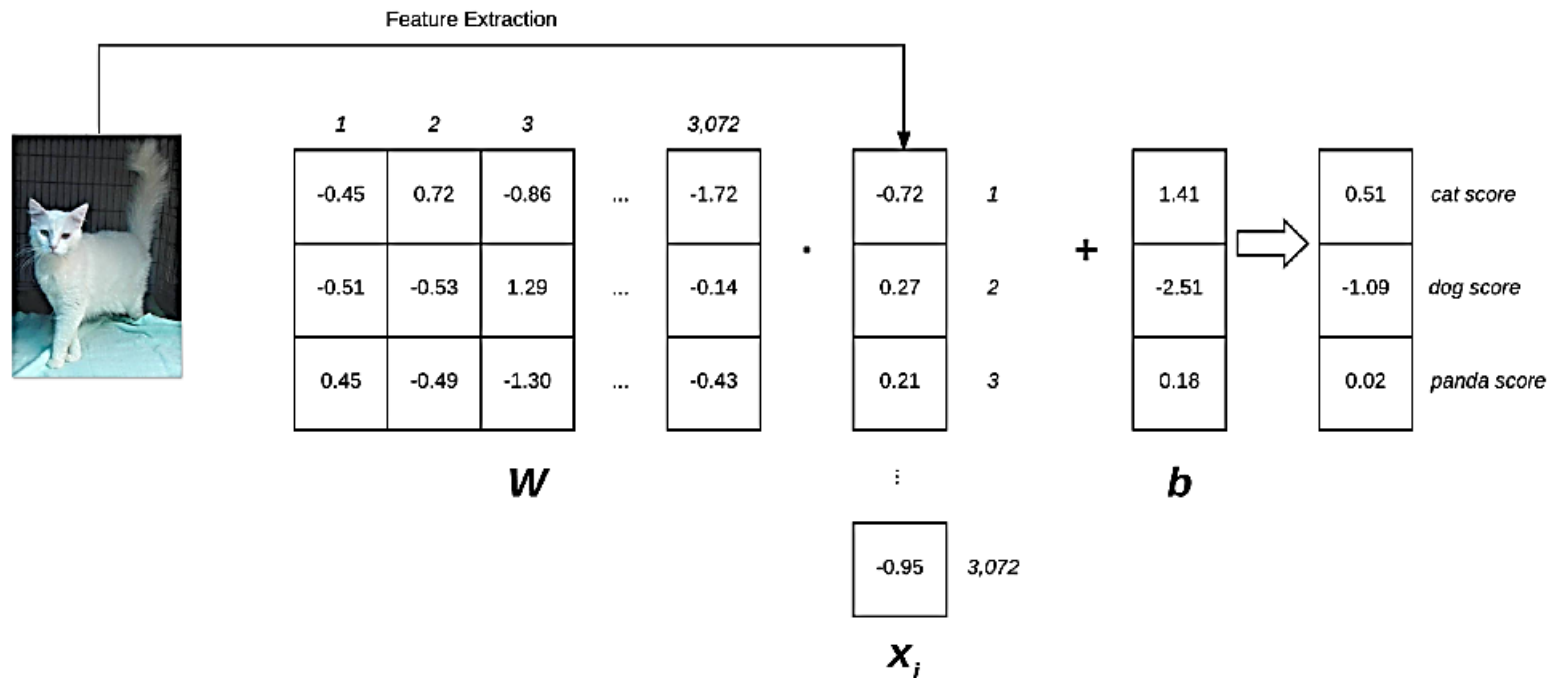
Given these variables, we must now define a scoring function f that maps the images to the class label scores. One method to accomplish this scoring is via a simple linear mapping:

$$f(x_i, W, b) = Wx_i + b$$

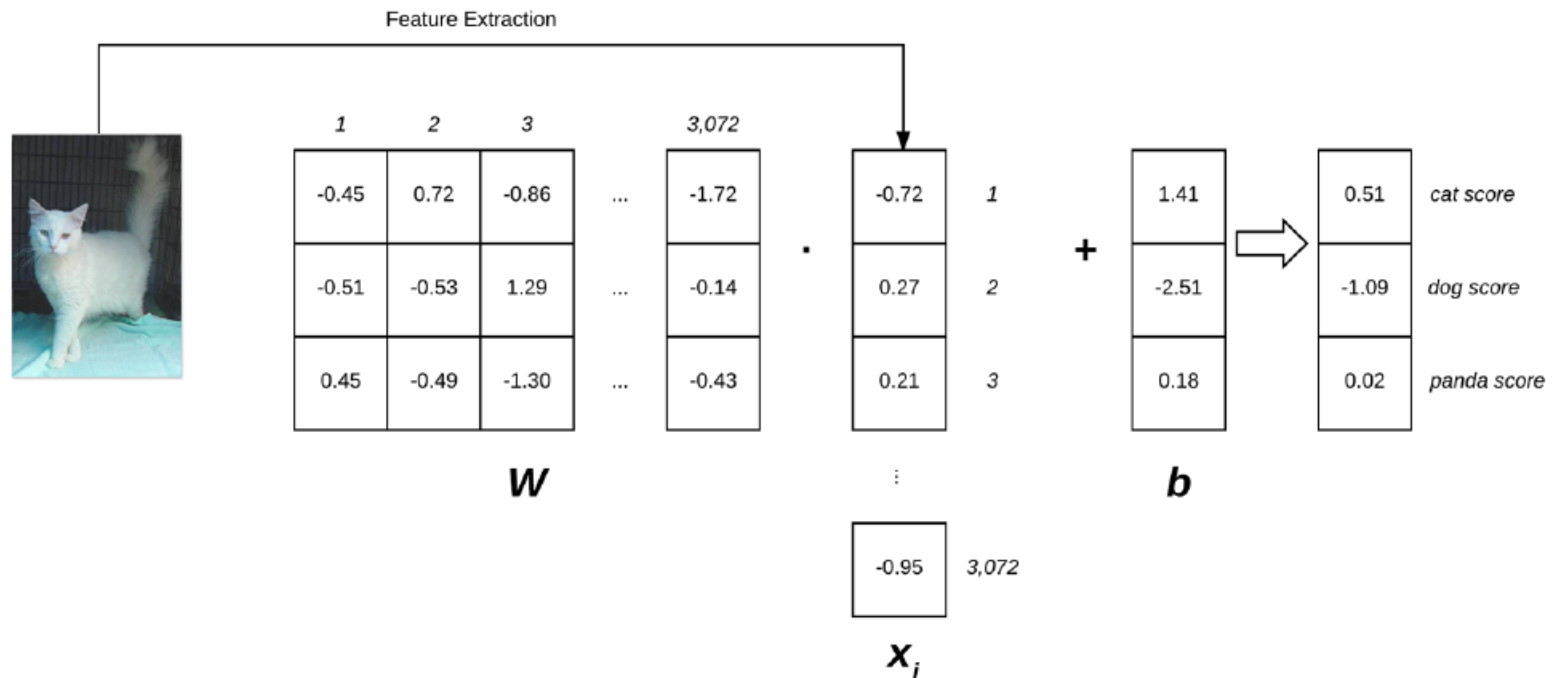
Let's assume that each x_i is represented as a single column vector with shape $[D \times 1]$ (in this example we would flatten the $32 \times 32 \times 3$ image into a list of 3,072 integers). Our weight matrix W would then have a shape of $[K \times D]$ (the number of class labels by the dimensionality of the input images). Finally b , the **bias vector** would be of size $[K \times 1]$. The bias vector allows us to shift and translate our scoring function in one direction or another without actually influencing our weight matrix W . The bias parameter is often critical for successful learning.

Linear Classification: From Images to Labels

Going back to the Animals dataset example, each x_i is represented by a list of 3,072 pixel values, so x_i , therefore, has the shape $[3,072 \times 1]$. The weight matrix W will have a shape of $[3 \times 3,072]$ and finally the bias vector b will be of size $[3 \times 1]$.

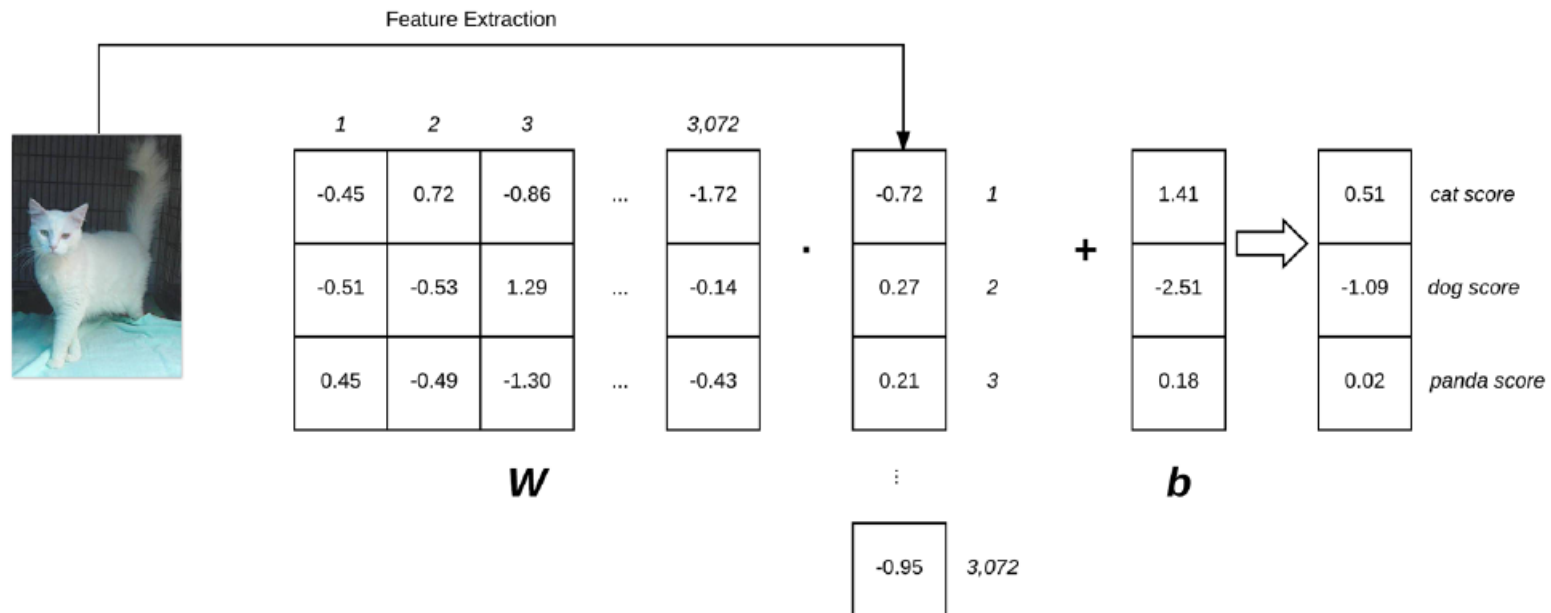


Linear Classification: From Images to Labels



Our weight matrix W contains three rows (one for each class label) and 3,072 columns (one for each of the pixels in the image). After taking the dot product between W and x_i , we add in the bias vector b – the result is our actual **scoring function**. Our scoring function yields three values on the *right*: the scores associated with the dog, cat, and panda labels, respectively.

Linear Classification: From Images to Labels



Looking at the above figure and equation, you can convince yourself that the input x_i and y_i are *fixed* and *not something we can modify*. Sure, we can obtain different x_i s by applying various transformations to the input image – but once we pass the image into the scoring function, **these values do not change**. In fact, the only parameters that we have any control over (in terms of parameterized learning) are our weight matrix W and our bias vector b . Therefore, our goal is to utilize both our scoring function and loss function to *optimize* (i.e., modify in a systematic way) the weight and bias vectors such that our classification accuracy *increases*.

Advantages of Parameterized Learning and Linear Classification

- There are two primary advantages to utilizing parameterized learning:
 1. **Once we are done training our model, we can discard the input data and keep only the weight matrix W and the bias vector b .** This *substantially* reduces the size of our model since we need to store two sets of vectors (versus the *entire* training set).
 2. **Classifying new test data is *fast*.** In order to perform a classification, all we need to do is take the dot product of W and x_i , follow by adding in the bias b (i.e., apply our scoring function). Doing it this way is *significantly faster* than needing to compare each testing point to *every* training example, as in the k-NN algorithm.

A Simple Linear Classifier With Python

- Show how we would initialize a weight matrix \mathbf{W} , bias vector \mathbf{b} , and then use these parameters to classify an image via a simple dot product.

A Simple Linear Classifier With Python

- Our goal here is to write a Python script that will correctly classify Figure as “dog”.



Our example input image that we are going to classify with a simple linear classifier

The Role of Loss Functions

- Parameterized learning allows us to take sets of **input data** and **class labels**, and **actually learn a function** that maps the input to the output predictions by defining a set of parameters and optimizing over them.
- But in order to actually “learn” the mapping from the input data to class labels via our scoring function, we need to discuss two important concepts:
 - Loss functions
 - Optimization methods

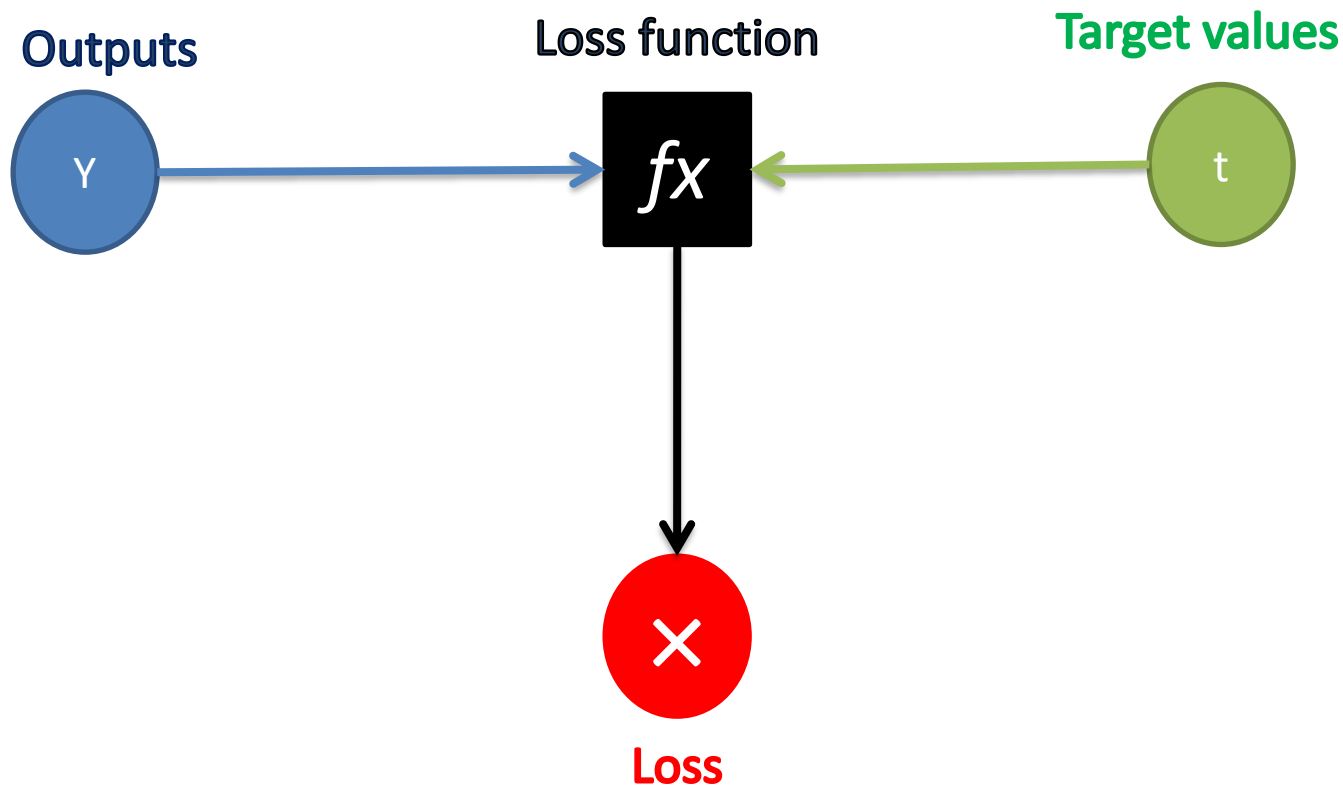
What Are Loss Functions?

- loss function quantifies how “good” or “bad” a given predictor is at classifying the input data points in a dataset.

To improve our classification accuracy, we need to tune the parameters of our weight matrix W or bias vector b . Exactly *how* we go about updating these parameters is an *optimization problem*, which we’ll be covering in the next chapter. For the time being, simply understand that a *loss function* can be used to quantify how well our *scoring function* is doing at classifying input data points.

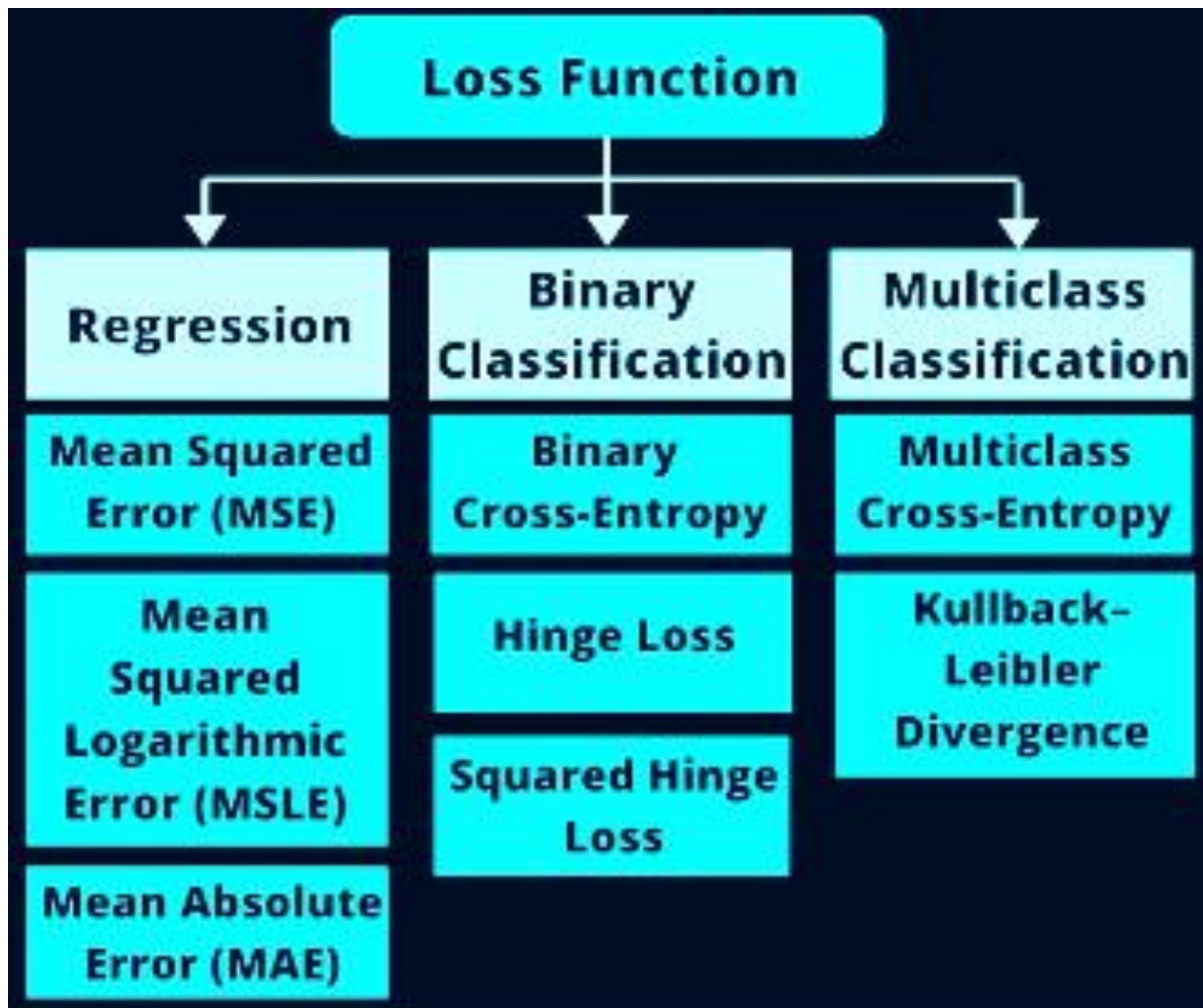
Loss Function

- Evaluating the performance of Model



Role of Loss Functions

- Input data and class labels
- Input \longrightarrow Output
- Quantifies how “**good**” or “**bad**”

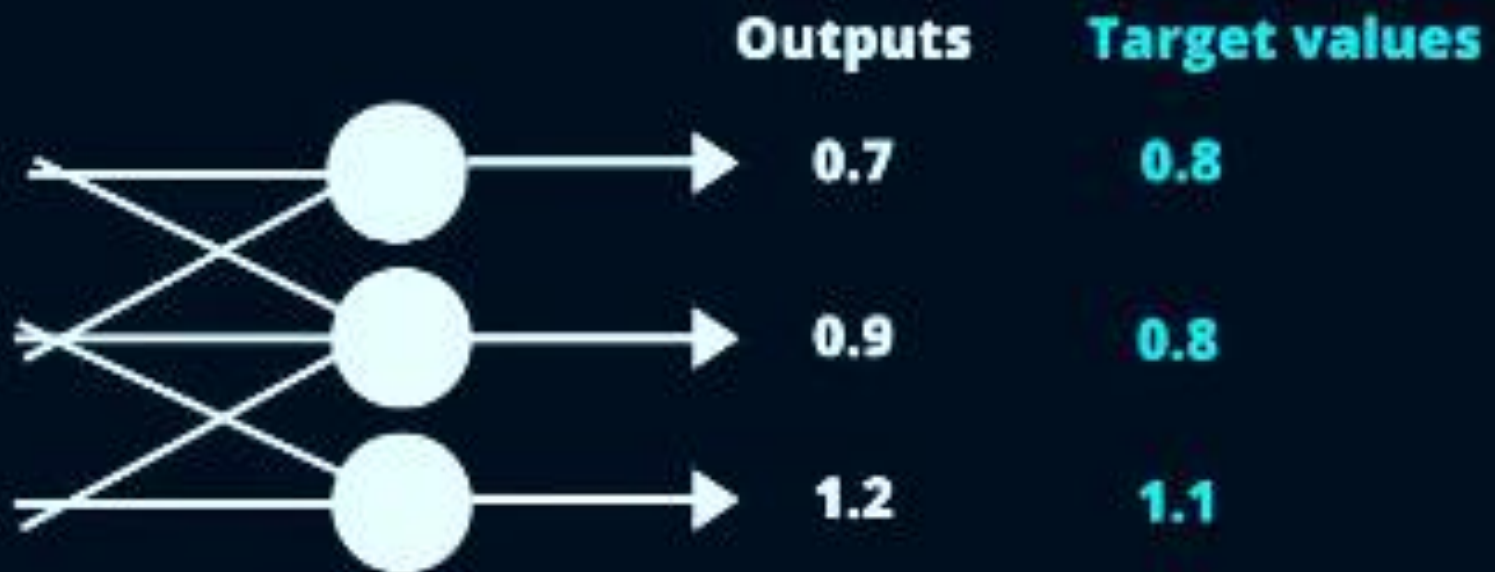


Mean Squared Error

$$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - t_i)^2$$

Output

Target value



In this case the **MSE** is calculated as follows:

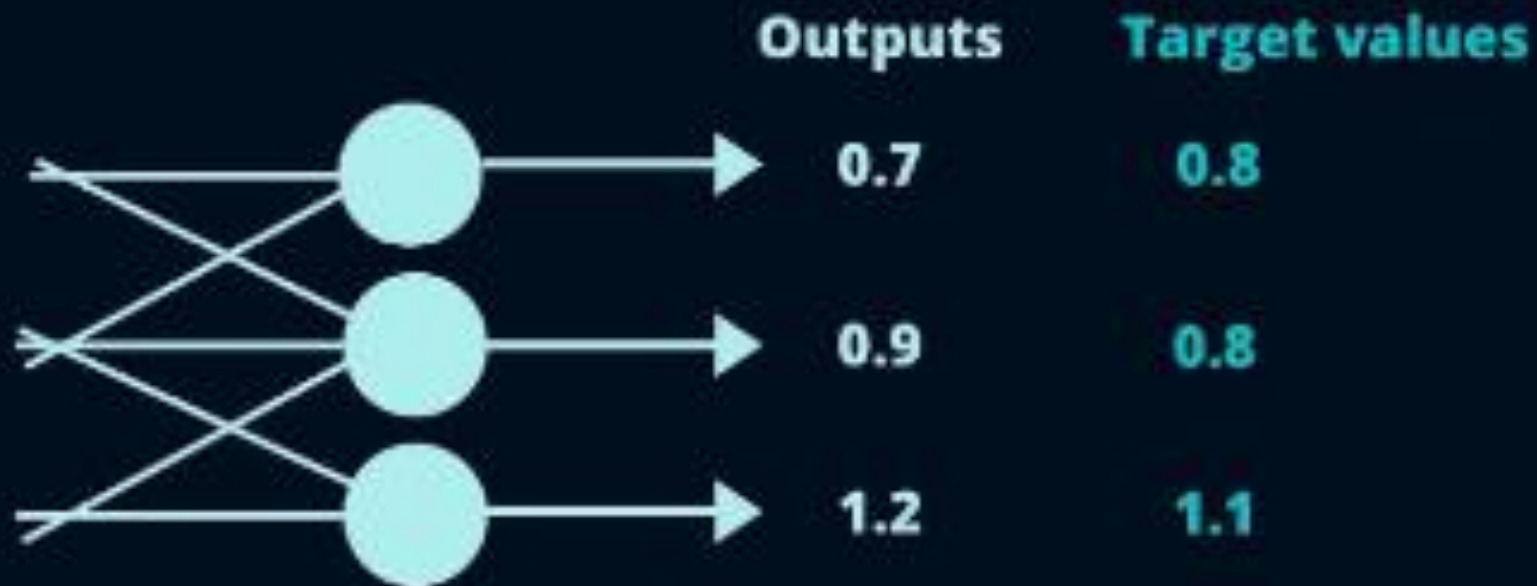
$$\begin{aligned}MSE &= \frac{1}{3} \left((y_1 - t_1)^2 + (y_2 - t_2)^2 + (y_3 - t_3)^2 \right) \\&= \frac{1}{3} \left((0.7 - 0.8)^2 + (0.9 - 0.8)^2 + (1.2 - 1.1)^2 \right) \\&= 0.01\end{aligned}$$

Mean Absolute Error

$$MAE = \frac{1}{m} \sum_{i=1}^m |y_i - t_i|$$

Output

Target value




In this case the **MAE** is calculated as follows:

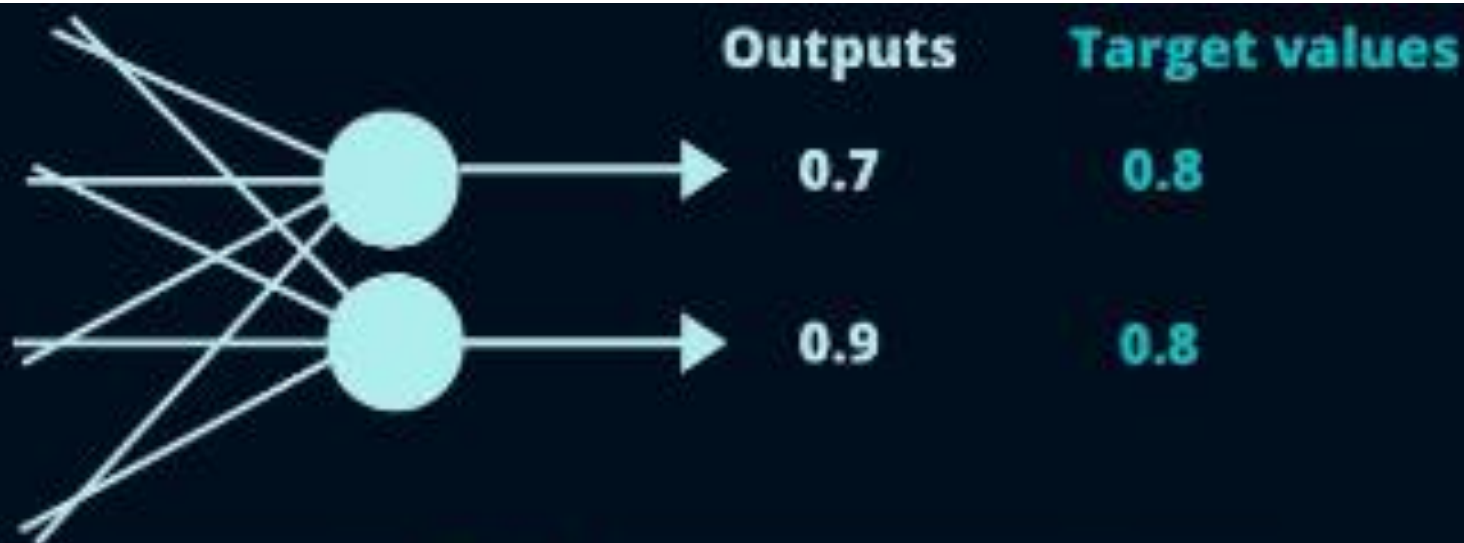
$$\begin{aligned} MAE &= \frac{1}{3} (|y_1 - t_1| + |y_2 - t_2| + |y_3 - t_3|) \\ &= \frac{1}{3} (|0.7 - 0.8| + |0.9 - 0.8| + |1.2 - 1.1|) \\ &= 0.1 \end{aligned}$$

Mean squared logarithmic Error

$$MSLE = \frac{1}{m} \sum_{i=1}^m (\log(y_i + 1) - \log(t_i + 1))^2$$

Output **Target value**

The diagram shows the MSLE formula with two white arrows pointing upwards from the labels 'Output' and 'Target value' to the terms y_i and t_i respectively in the formula.



In this case the **MSLE** is calculated as follows:

$$\begin{aligned}
 &= \frac{1}{2} \left((\log(y_1 + 1) - \log(t_1 + 1))^2 + (\log(y_2 + 1) - \log(t_2 + 1))^2 \right) \\
 &= \frac{1}{2} \left((\log(0.7 + 1) - \log(0.8 + 1))^2 + (\log(0.9 + 1) - \log(0.8 + 1))^2 \right) \\
 &= \frac{1}{2} \left((\log(1.7) - \log(1.8))^2 + (\log(1.9) - \log(1.8))^2 \right) \\
 &= 0.000058
 \end{aligned}$$

Multi-class SVM Loss

- Inspired by (Linear) Support Vector Machines (SVMs)
- Which uses a scoring function f to map our data points to numerical scores for each class labels.

Multi-class SVM Loss

$$f(x_i, W, b) = Wx_i + b$$

Now that we have our scoring function, we need to determine how “good” or “bad” this function is (given the weight matrix W and bias vector b) at making predictions. To make this determination, we need a *loss function*.

Recall that when creating a machine learning model we have a *design matrix* X , where each row in X contains a data point we wish to classify. In the context of image classification, each row in X is an image and we seek to correctly label this image. We can access the i -th image inside X via the syntax x_i .

Multi-class SVM Loss

Similarly, we also have a vector \mathbf{y} which contains our class labels for each \mathbf{X} . These \mathbf{y} values are our *ground-truth labels* and what we hope our scoring function will correctly predict. Just like we can access a given image as x_i , we can access the associated class label via y_i .

As a matter of simplicity, let's abbreviate our scoring function as s :

$$s = f(x_i, W)$$

Multi-class SVM Loss

Which implies that we can obtain the predicted score of the j -th class via the i -th data point:

$$s_j = f(x_i, W)_j$$

Using this syntax, we can put it all together, obtaining the *hinge loss function*:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

A given x_i is classified correctly when the loss $L_i = 0$ (I'll provide a numerical example in the following section). To derive the loss across our *entire training set*, we simply take the mean over each individual L_i :

$$L = \frac{1}{N} \sum_{i=1}^N L_i$$

Multi-class SVM Loss

Another related loss function you may encounter is the *squared hinge loss*:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

The squared term penalizes our loss more heavily by squaring the output, which leads to quadratic growth in loss in a prediction that is incorrect (versus a linear growth).

A Multi-class SVM Loss Example

- “Animals” dataset which aims to classify a given image as containing a **cat**, **dog**, or **panda**.

A Multi-class SVM Loss Example



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

A Multi-class SVM Loss Example



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Let's start by computing the loss L_i for the “dog” class:

```
1 >>> max(0, 1.33 - 4.26 + 1) + max(0, -1.01 - 4.26 + 1)  
2 0
```

A Multi-class SVM Loss Example



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Similarly, we can compute the hinge loss for Image #2, this one containing a cat:

```
3 >>> max(0, 3.76 - (-1.20) + 1) + max(0, -3.81 - (-1.20) + 1)  
4 5.96
```


A Multi-class SVM Loss Example



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Finally, let's compute the hinge loss for the panda example:

```
>>> max(0, -2.37 - (-2.27) + 1) + max(0, 1.03 - (-2.27) + 1)  
5.199999999999999
```

A Multi-class SVM Loss Example



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

We can then obtain the *total loss* over the three examples by taking the average:

```
7 >>> (0.0 + 5.96 + 5.2) / 3.0
8 3.72
```

A Multi-class SVM Loss Example

- Take note that our **loss was zero** for only one of the three input images, implying that **two of our predictions were incorrect**.

Therefore, given our three training examples our overall hinge loss is 3.72 for the parameters W and b .

Cross-entropy Loss and Softmax Classifiers

While hinge loss is quite popular, you're *much* more likely to run into cross-entropy loss and Softmax classifiers in the context of deep learning and convolutional neural networks.

Why is this? Simply put:

Softmax classifiers give you *probabilities* for each class label while hinge loss gives you the *margin*.

It's much easier for us as humans to interpret probabilities rather than margin scores.

Understanding Cross-entropy Loss

The Softmax classifier is a generalization of the binary form of Logistic Regression. Just like in hinge loss or squared hinge loss, our mapping function f is defined such that it takes an input set of data x_i and maps them to output class labels via dot product of the data x_i and weight matrix W

$$f(x_i, W) = Wx_i$$

However, unlike hinge loss, we can interpret these scores as *unnormalized log probabilities* for each class label, which amounts to swapping out the hinge loss function with cross-entropy loss:

$$L_i = -\log(e^{s_{y_i}} / \sum_j e^{s_j})$$

$$L_i = -\log P(Y = y_i | X = x_i)$$

The probability statement can be interpreted as:

$$P(Y = y_i | X = x_i) = e^{s_{y_i}} / \sum_j e^{s_j}$$

Understanding Cross-entropy Loss

$$s = f(x_i, W)$$

As a whole, this yields our final loss function for a *single* data point, just like above:

$$L_i = -\log(e^{s_{y_i}} / \sum_j e^{s_j})$$

Take note that your logarithm here is actually base e (natural logarithm) since we are taking the inverse of the exponentiation over e earlier. The actual exponentiation and normalization via the sum of exponents is our *Softmax function*. The negative log yields our actual *cross-entropy loss*.

Just as in hinge loss and squared hinge loss, computing the cross-entropy loss over an entire dataset is done by taking the average:

$$L = \frac{1}{N} \sum_{i=1}^N L_i$$

Cross-entropy Loss and Softmax Classifiers

A Worked Softmax Example

	Scoring Function
Dog	-3.44
Cat	1.16
Panda	3.91



Input Image

	Scoring Function	Unnormalized Probabilities
Dog	-3.44	0.03
Cat	1.16	3.19
Panda	3.91	49.90

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities
Dog	-3.44	0.0321	0.0006
Cat	1.16	3.1899	0.0601
Panda	3.91	49.8990	0.9393

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities	Negative Log Loss
Dog	-3.44	0.0321	0.0006	
Cat	1.16	3.1899	0.0601	
Panda	3.91	49.8990	0.9393	0.0626

Cross-entropy Loss and Softmax Classifiers

In this case, our Softmax classifier would correctly report the image as *panda* with 93.93% confidence. We can then repeat this process for all images in our training set, take the average, and obtain the overall cross-entropy loss for the training set. This process allows us to quantify how good or bad a set of parameters are performing on our training set.

Optimization Methods

- *“Nearly all of deep Learning is powered by one very important algorithm: Stochastic Gradient Descent (SGD)”*
- **Optimization** algorithms are the engines that power neural networks and enable them to learn patterns from data.
- Find set of W and b

Optimization Methods

- High accuracy classifier is dependent on finding a set of weights \mathbf{W} and \mathbf{b} such that our data points are correctly classified
- But how do we go about **finding** and **obtaining** a weight matrix \mathbf{W} and bias vector \mathbf{b} that obtains high classification accuracy?
- Do we **randomly** initialize them, evaluate, and repeat over and over again, hoping that at some point we land on a set of parameters that **obtains reasonable classification**?

Optimization Methods

- We could – but given that modern deep learning networks have parameters that number in the **tens of millions**, it may take us a **long time** to blindly stumble upon a reasonable set of parameters.
- Instead of relying on pure randomness, we need to define an **optimization algorithm** that allows us to literally improve **\mathbf{W}** and **\mathbf{b}** .

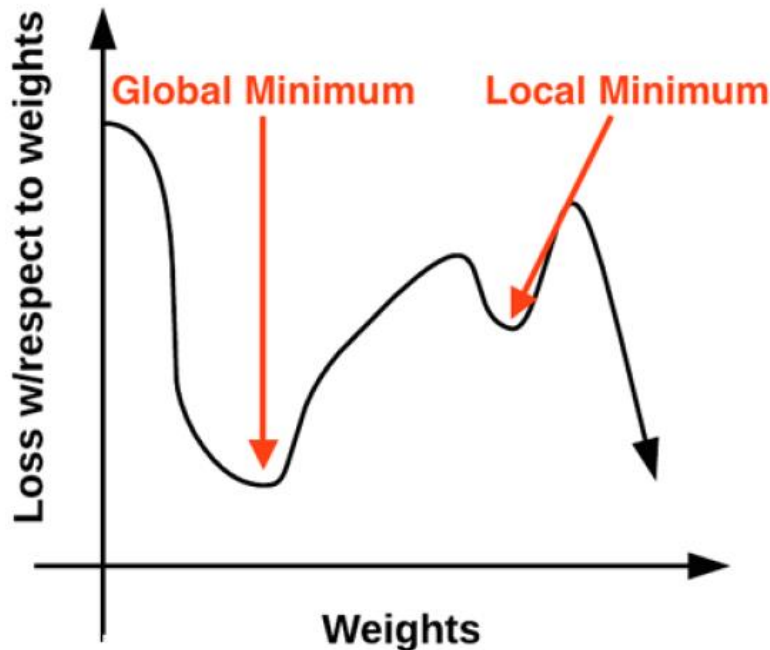
Optimization Methods

- Most common algorithm used to train neural networks and deep learning models – **gradient descent**.
- Gradient descent has many variants (which we'll also touch on), but, in each case, the idea is the same: **iteratively evaluate your parameters, compute your loss, then take a small step in the direction that will minimize your loss.**

OM: Gradient Descent

- The gradient descent algorithm has two primary flavors:
 1. The standard “**vanilla**” implementation.
 2. The optimized “**stochastic**” version that is more commonly used.

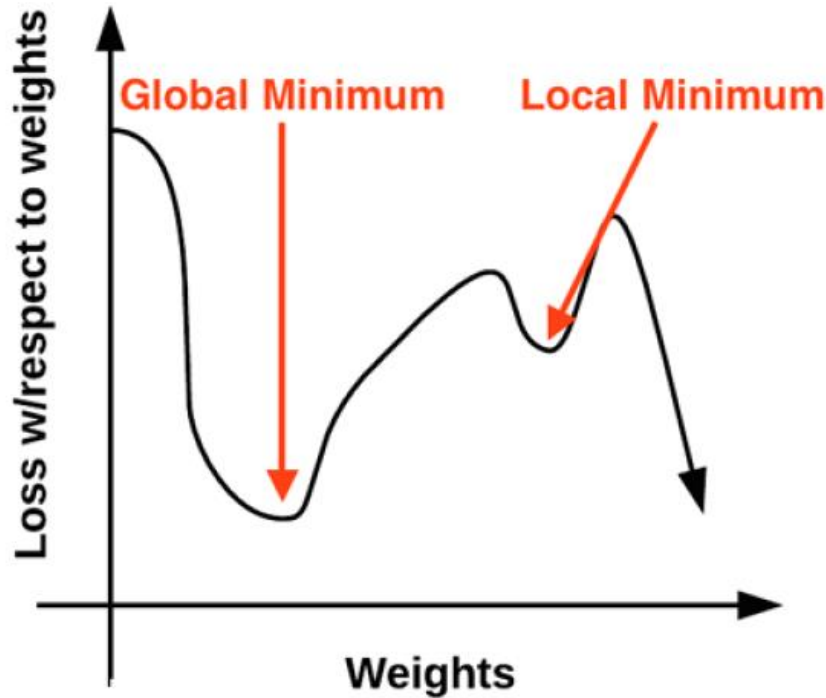
OM: Gradient Descent



The gradient descent method is an iterative optimization algorithm that operates over a loss landscape (also called an optimization surface).

The canonical gradient descent example is to **visualize our weights along the x-axis** and then the **loss for a given set of weights along the y-axis**.

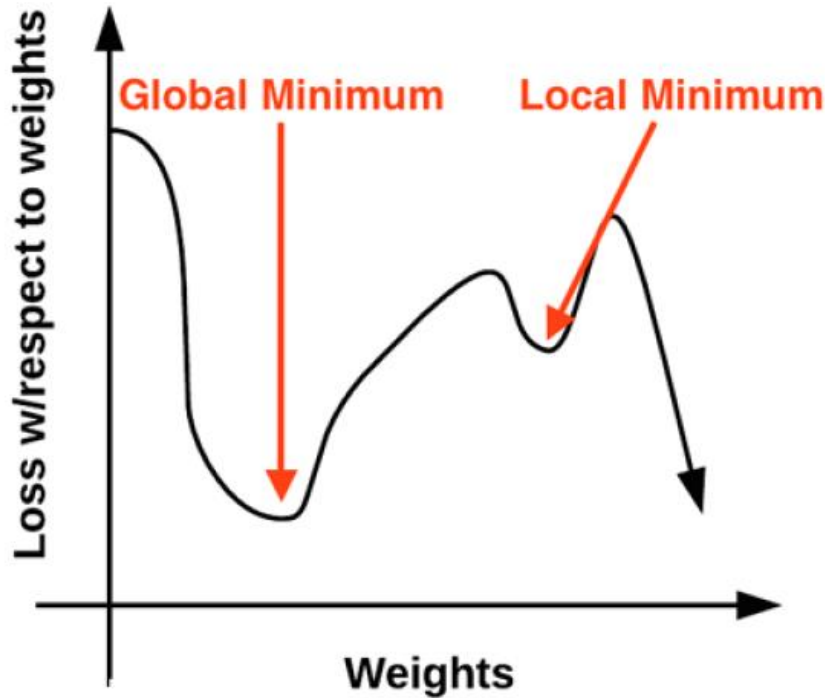
OM: Gradient Descent



Left: The “naive loss” visualized as a 2D plot.

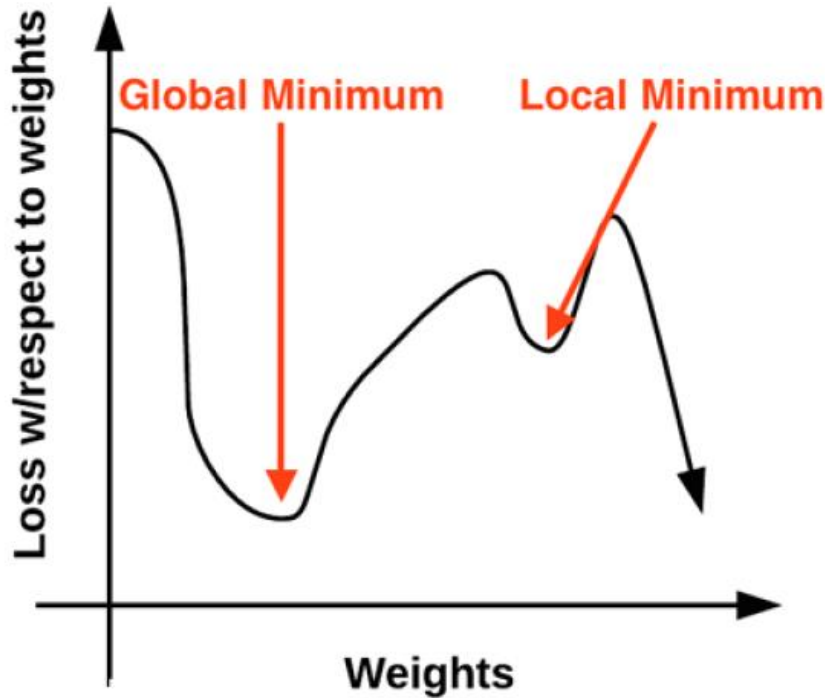
Right: A more realistic loss landscape can be visualized as a bowl that exists in multiple dimensions. Our goal is to apply gradient descent to navigate to the **bottom of this bowl (where there is low loss)**.

OM: Gradient Descent



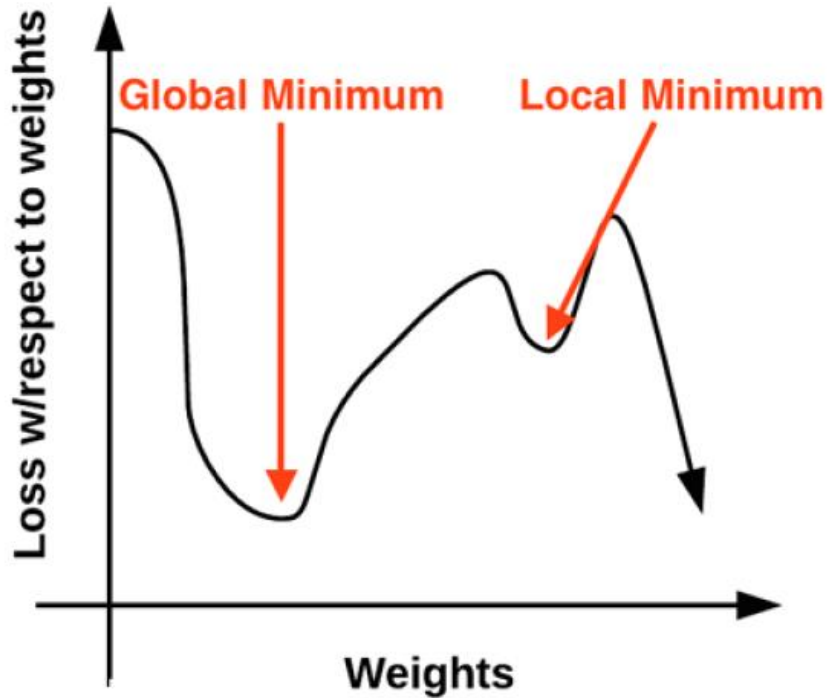
As we can see, our loss landscape has many **peaks** and **valleys** based on which values our parameters take on.

OM: Gradient Descent



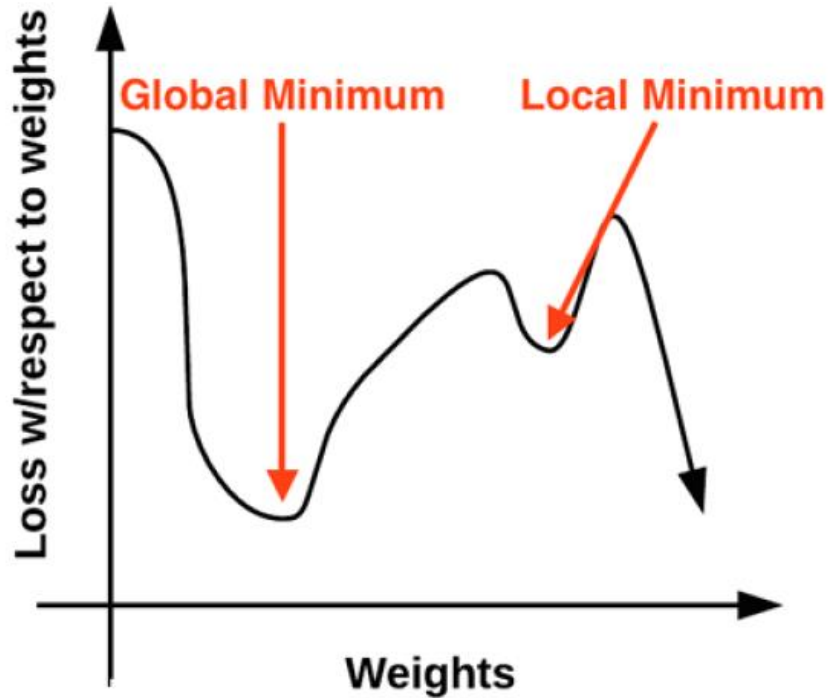
Each peak is a **local maximum** that represents *very high regions of loss* – the local maximum

OM: Gradient Descent



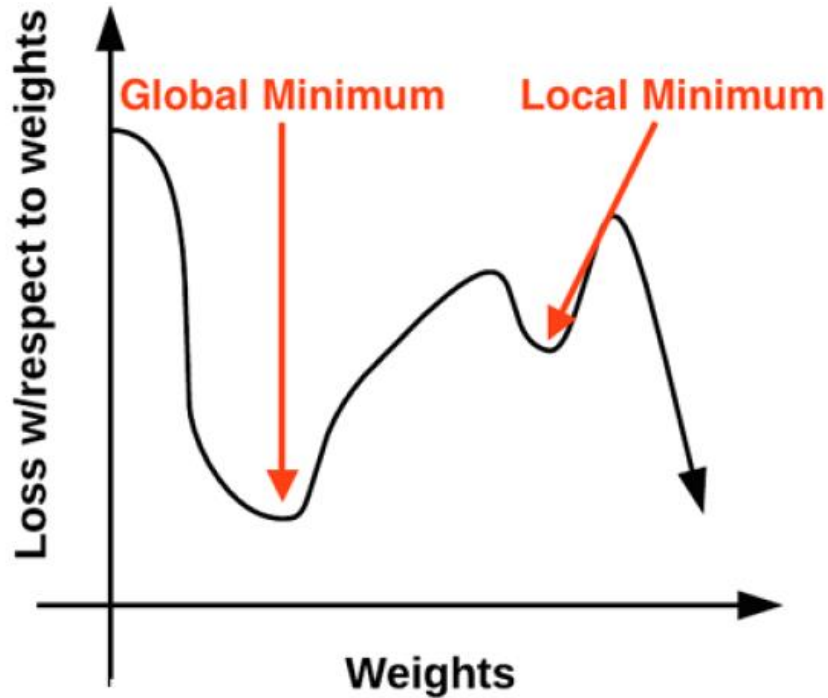
Largest loss across the **entire loss landscape** is the **global maximum**

OM: Gradient Descent



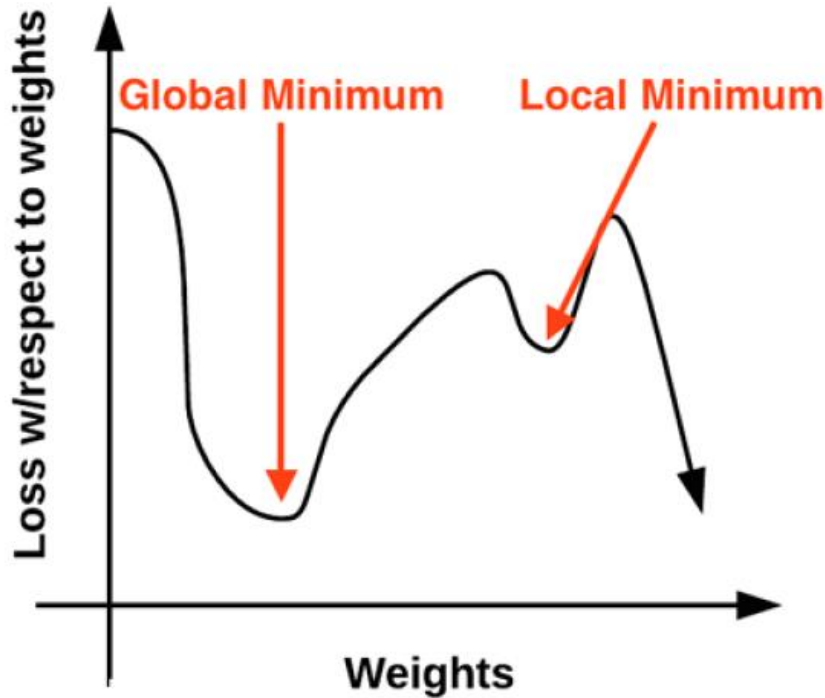
Local minimum which represents *many small regions of loss*.

OM: Gradient Descent



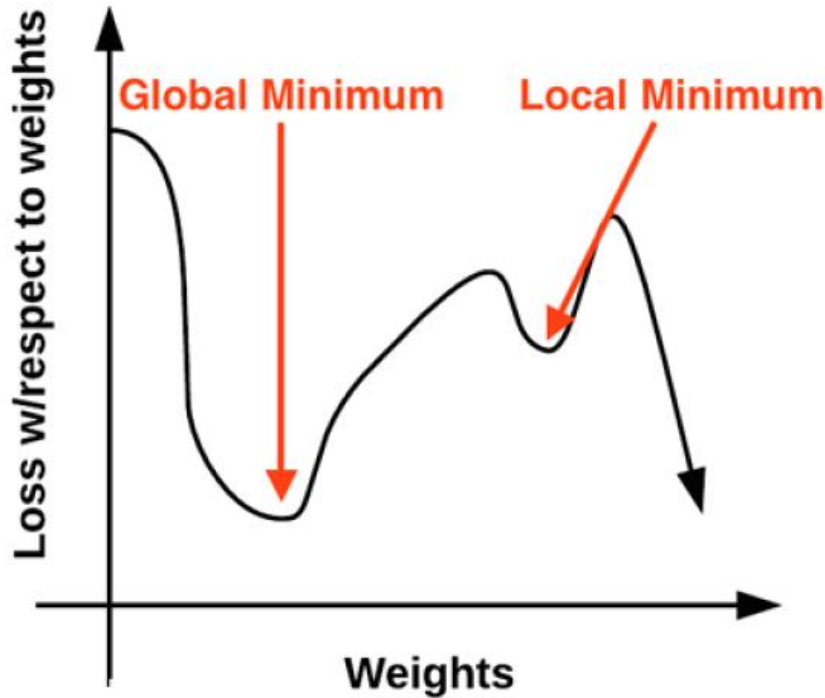
The local minimum with the smallest loss across the loss landscape is our **global minimum**

OM: Gradient Descent



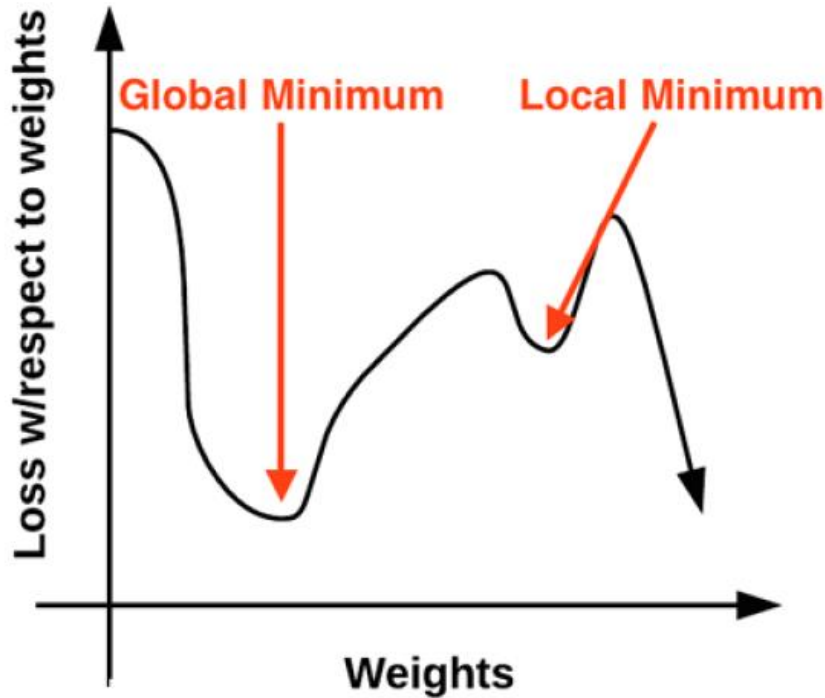
In an ideal world, we would like to find this **global minimum**, ensuring our parameters take on the most optimal possible values.

OM: Gradient Descent



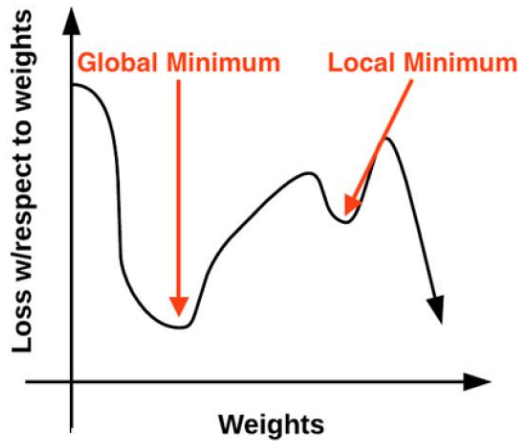
So that raises the question: “If we want to reach a **global minimum**, why not just **directly jump to it**? It’s clearly visible on the plot?”

OM: Gradient Descent



So that raises the question: “If we want to reach a **global minimum**, why not just **directly jump to it**? It’s clearly visible on the plot?”

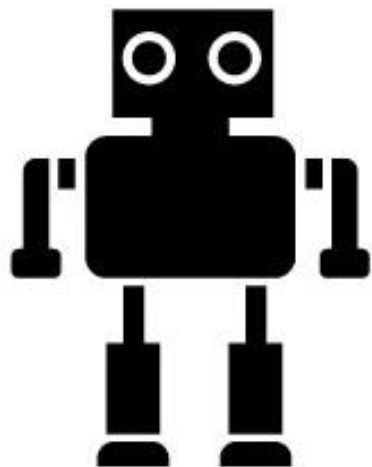
OM: Gradient Descent



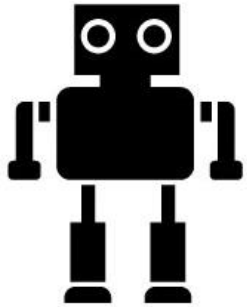
Therein lies the problem – the loss landscape is **invisible** to us. We don't actually know what it looks like.

If we're an optimization algorithm, we would be blindly placed somewhere on the plot, having no idea what the landscape in front of us looks like, and we would have to navigate our way to a loss minimum without accidentally climbing to the **top of a local maximum**.

The “Gradient” in Gradient Descent



The “Gradient” in Gradient Descent



What is Chiti to do? The answer is to apply gradient descent. All Chiti needs to do is follow the **slope of the gradient \mathbf{W}** . We can compute the **gradient \mathbf{W} across all dimensions** using the following equation:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

The “Gradient” in Gradient Descent



1. It's an approximation to the gradient.
2. It's painfully slow.

In practice, we use the analytic gradient instead. The method is exact and fast, but **extremely challenging to implement due to partial derivatives and multi-variable calculus.**

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Vanilla gradient descent

- Vanilla gradient descent only performs a weight update once for every epoch – in this example, **we trained our model for 100 epochs, so only 100 updates took place.**
- Depending on the initialization of the weight matrix and the size of the learning rate, it's possible that we may not be able to learn a model that **can separate the points (even though they are linearly separable).**







Follow the slope

- In multiple dimensions, the gradient is the **vector of (partial derivatives)** along each dimension

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (first dim):

[0.34 + 0.0001,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25322

gradient dW:

[-2.5,
?,
?,

$$(1.25322 - 1.25347)/0.0001 = -2.5$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

?,
?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (second dim):

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25353

gradient dW:

[-2.5,
?,
?,
?,
?,
?,
?,
?,
?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (second dim):

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,

$$(1.25353 - 1.25347)/0.0001 = 0.6$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,...]

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

gradient dW:

$[-2.5,$
 $0.6,$
 $0,$
 $?,$
 0

$$(1.25347 - 1.25347)/0.0001 = 0$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

! , . .]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?

Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

?,...]

This is silly. The loss is just a function of W :

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Use calculus to compute an
analytic gradient



[This image](#) is in the public domain



[This image](#) is in the public domain

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

$dW = \dots$
(some function
data and W)

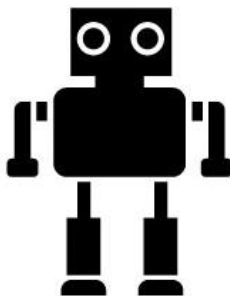


gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,...]

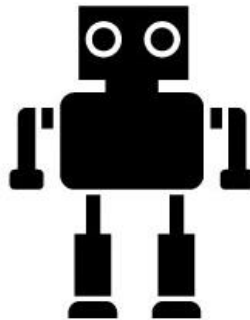
Treat It Like a Convex Problem (Even if It's Not)

- Using the bowl in as a visualization of the loss landscape also allows us to draw an important conclusion in modern day neural networks – we are treating the loss landscape as a **convex problem**, even if it's not.



Treat It Like a Convex Problem (Even if It's Not)

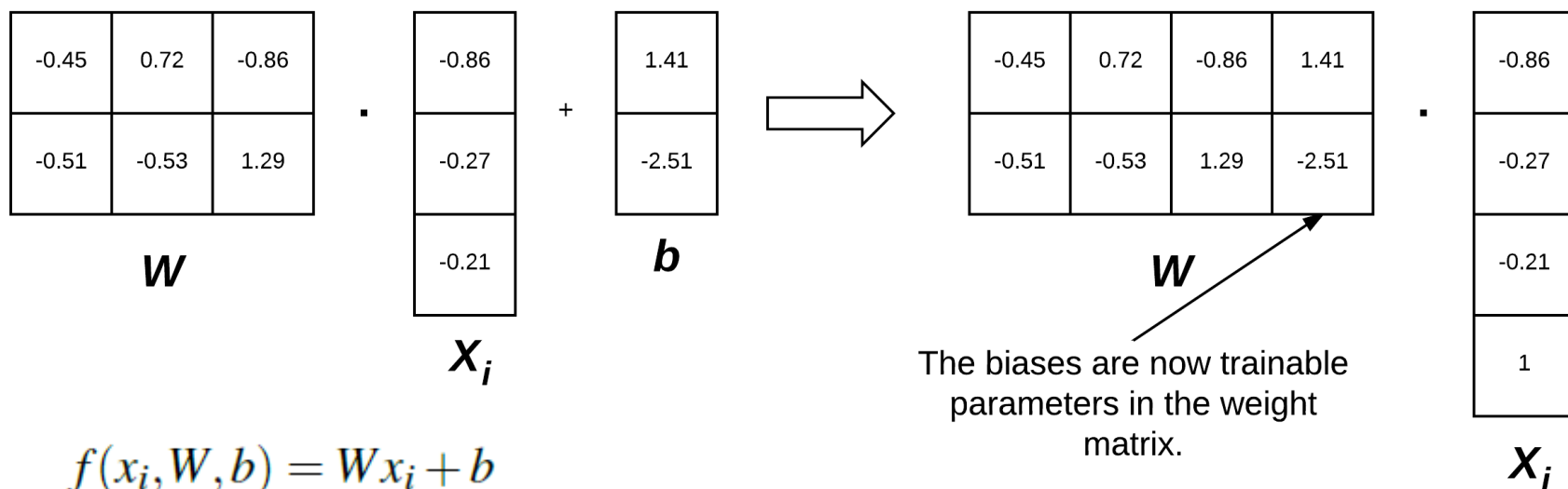
- Given the non-convex nature of our datasets, why do we apply gradient descent? The answer is simple: because it does a good enough job.



To quote Goodfellow et al

“[An] optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the [loss] function quickly enough to be useful.”

The Bias Trick



Again, we are allowed to omit the b term here as it is *embedded* into our weight matrix.

In the context of our previous examples in the “Animals” dataset, we’ve worked with $32 \times 32 \times 3$ images with a total of 3,072 pixels. Each x_i is represented by a vector of $[3072 \times 1]$. Adding in a dimension with a constant value of one now expands the vector to be $[3073 \times 1]$. Similarly, combining both the bias and weight matrix also expands our weight matrix W to be $[3 \times 3073]$ rather than $[3 \times 3072]$. In this way, we can treat the bias as a *learnable parameter within the weight matrix* that we don’t have to explicitly keep track of in a separate variable.

Pseudocode for Gradient Descent

- Vanilla gradient descent algorithm

```
1  while True:
2      Wgradient = evaluate_gradient(loss, data, W)
3      W += -alpha * Wgradient
```

This pseudocode is what *all* variations of gradient descent are built off of. We start off on **Line 1** by looping until some condition is met, typically either:

1. A specified number of epochs has passed (meaning our learning algorithm has “seen” each of the training data points N times).
2. Our loss has become *sufficiently low* or training accuracy *satisfactory high*.
3. Loss has not improved in M subsequent epochs.

Line 2 then calls a function named `evaluate_gradient`. This function requires three parameters:

1. `loss`: A function used to compute the loss over our current parameters W and input `data`.
2. `data`: Our training data where each training sample is represented by an image (or feature vector).
3. `W`: Our actual weight matrix that we are optimizing over. Our goal is to apply gradient descent to find a W that yields minimal loss.

Pseudocode for Gradient Descent

- Vanilla gradient descent algorithm

```
1  while True:
2      Wgradient = evaluate_gradient(loss, data, W)
3      W += -alpha * Wgradient
```

The `evaluate_gradient` function returns a vector that is K -dimensional, where K is the number of dimensions in our image/feature vector. The `Wgradient` variable is the actual gradient, where we have a gradient entry for each dimension.

We then apply *gradient descent* on **Line 3**. We multiply our `Wgradient` by `alpha` (α), which is our *learning rate*. **The learning rate controls the size of our step.**

DEMO

Stochastic Gradient Descent (SGD)

- “vanilla” implementation of gradient descent can be **prohibitively slow to run on large datasets – in fact, it can even be considered computationally wasteful.**
- Modification to the standard gradient descent algorithm that computes the gradient and updates the **weight matrix W on small batches of training data, rather than the entire training set.**

Stochastic Gradient Descent (SGD)

- While this modification leads to “**more noisy**” **updates**, it also allows us to take more steps along the gradient (**one step per each batch versus one step per epoch**)
- Ultimately leading to **faster convergence and no negative effects to loss and classification accuracy**.

Mini-batch SGD

- Vanilla gradient descent algorithm: **run very slowly on large datasets**
- The reason for this slowness is because each iteration of gradient descent requires us to compute a prediction for each training point in our training data before we are allowed to update our weight matrix. **For image datasets such as ImageNet where we have over 1.2 million training images, this computation can take a long time.**

Mini-batch SGD

- Instead, what we should do is batch our updates. We can update the pseudocode to transform vanilla gradient descent to become SGD by adding an extra function call:

```
1  while True:
2      batch = next_training_batch(data, 256)

3      Wgradient = evaluate_gradient(loss, batch, W)
4      W += -alpha * Wgradient
```

Mini-batch SGD

- Instead of computing our gradient over the entire data set, we instead sample our data, yielding a batch. We evaluate the gradient on the batch, and update our weight matrix **W**.

```
1  while True:
2      batch = next_training_batch(data, 256)

3      Wgradient = evaluate_gradient(loss, batch, W)
4      W += -alpha * Wgradient
```

Mini-batch SGD

- We often use mini-batches that are > 1 . Typical batch sizes include **32, 64, 128, and 256**.
- In general, the mini-batch size is not a hyper parameter you should worry too much about. If you're using a **GPU to train your neural network, you determine how many training examples will fit into your GPU and then use the nearest power of two as the batch size such that the batch will fit on the GPU.**

Investigating the actual loss values at the end of the 100th epoch, you'll notice that loss obtained by SGD is *nearly two orders of magnitude lower* than vanilla gradient descent (0.006 vs 0.447, respectively). This difference is due to the multiple weight updates per epoch, giving our model more chances to learn from the updates made to the weight matrix. This effect is even more pronounced on large datasets, such as ImageNet where we have millions of training examples and small, incremental updates in our parameters can lead to a low loss (but not necessarily optimal) solution.

Extensions to SGD

- Momentum:
 - Method used to accelerate SGD
 - Enabling it to learn faster by focusing on dimensions whose gradient point in the same direction.
- Nesterov acceleration: An extension to standard momentum.

Momentum

- **Increase** the strength of updates for dimensions whose gradients point in same direction.
- **Decrease** the strength of updates for dimensions whose gradients switch directions

Momentum

Our previous weight update rule simply included the scaling of the gradient by our learning rate:

$$W = W - \alpha \nabla_W f(W)$$

We now introduce the momentum term V , scaled by γ :

$$V = \gamma V_{t-1} + \alpha \nabla_W f(W)$$

$$W = W - V_t$$

The momentum term γ is commonly set to 0.9; although another common practice is to set γ to 0.5 until learning stabilizes and then increase it to 0.9 – it is extremely rare to see momentum < 0.5.

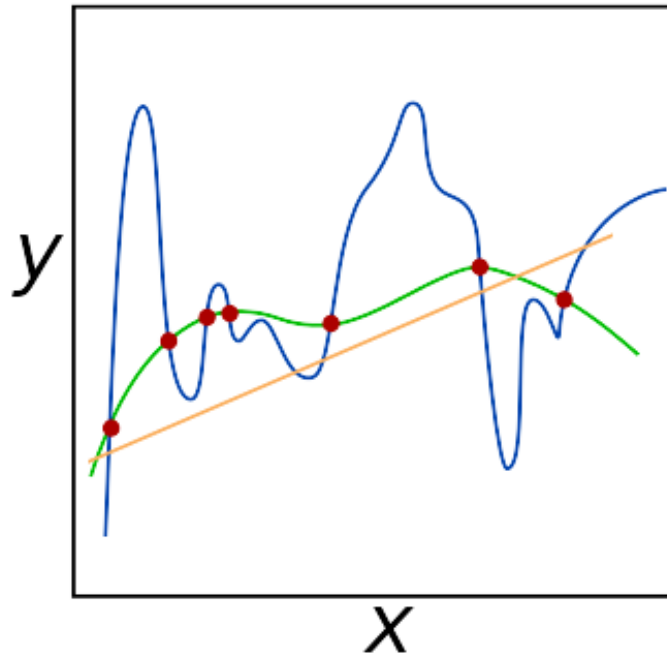
Regularization

- “Many strategies used in machine learning are explicitly designed to **reduce the test error**, possibly at the **expense of increased training error**. These strategies are collectively known as regularization”

What Is Regularization and Why Do We Need It?

Regularization helps us control our model capacity, ensuring that our models are better at making (correct) classifications on data points that they were *not* trained on, which we call *the ability to generalize*. If we don't apply regularization, our classifiers can easily become too complex and *overfit* to our training data, in which case we lose the ability to generalize to our testing data (and data points *outside* the testing set as well, such as new images in the wild).

However, too much regularization can be a bad thing. We can run the risk of *underfitting*, in which case our model performs poorly on the training data and is not able to model the relationship between the input data and output class labels (because we limited model capacity too much). For example, consider the following plot of points, along with various functions that fit to these points



Type of regularization techniques

- **L1 regularization**
- **L2 regularization**
 - Commonly called “weight decay”
- **Elastic Net:** that are used by updating the loss function itself, adding an additional parameter to constrain the capacity of the model.

Type of regularization techniques

Types of Regularization Techniques

In general, you'll see three common types of regularization that are applied directly to the loss function. The first, we reviewed earlier, L2 regularization (aka “weight decay”):

$$R(W) = \sum_i \sum_j W_{i,j}^2$$

We also have L1 regularization which takes the absolute value rather than the square:

$$R(W) = \sum_i \sum_j |W_{i,j}|$$

Elastic Net regularization seeks to combine both L1 and L2 regularization:

$$R(W) = \sum_i \sum_j \beta W_{i,j}^2 + |W_{i,j}|$$

Types of regularization that can be explicitly added

- We also have types of regularization that can be explicitly added to the network architecture – **dropout** is the quintessential example of such regularization.

Implicit forms of regularization

- Implicit forms of regularization that are applied during the training process
- Examples of implicit regularization include **data augmentation and early stopping.**

Updating Our Loss and Weight Update To Include Regularization

Let's start with our cross-entropy loss function

$$L_i = -\log(e^{s_{y_i}} / \sum_j e^{s_j})$$

The loss over the entire training set can be written as:

$$L = \frac{1}{N} \sum_{i=1}^N L_i$$

Updating Our Loss and Weight Update To Include Regularization

Let's start with our cross-entropy loss function

$$L_i = -\log(e^{s_{y_i}} / \sum_j e^{s_j})$$

The loss over the entire training set can be written as:

$$L = \frac{1}{N} \sum_{i=1}^N L_i$$

Now, let's say that we have obtained a weight matrix \mathbf{W} such that *every data point* in our training set is classified correctly, which means that our loss $L = 0$ for all L_i .

Awesome, we're getting 100% accuracy – but let me ask you a question about this weight matrix – *is it unique?* **Or, in other words, are there *better* choices of \mathbf{W} that will improve our model's ability to generalize and reduce overfitting?**

If there is such a \mathbf{W} , how do we know? And how can we incorporate this type of penalty into our loss function? The answer is to define a **regularization penalty**, a function that operates on our weight matrix. The regularization penalty is commonly written as a function, $R(\mathbf{W})$.

Updating Our Loss and Weight Update To Include Regularization

Now, let's say that we have obtained a weight matrix W such that *every data point* in our training set is classified correctly, which means that our loss $L = 0$ for all L_i .

Awesome, we're getting 100% accuracy – but let me ask you a question about this weight matrix – *is it unique?* **Or, in other words, are there *better* choices of W that will improve our model's ability to generalize and reduce overfitting?**

If there is such a W , how do we know? And how can we incorporate this type of penalty into our loss function? The answer is to define a **regularization penalty**, a function that operates on our weight matrix. The regularization penalty is commonly written as a function, $R(W)$.

$$R(W) = \sum_i \sum_j W_{i,j}^2$$

Updating Our Loss and Weight Update To Include Regularization

$$R(W) = \sum_i \sum_j W_{i,j}^2$$

What is the function doing exactly? In terms of Python code, it's simply taking the sum of squares over an array:

```
1 penalty = 0
2
3 for i in np.arange(0, W.shape[0]):
4     for j in np.arange(0, W.shape[1]):
5         penalty += (W[i][j] ** 2)
```

Updating Our Loss and Weight Update To Include Regularization

What is the function doing exactly? In terms of Python code, it's simply taking the sum of squares over an array:

```
1 penalty = 0
2
3 for i in np.arange(0, W.shape[0]):
4     for j in np.arange(0, W.shape[1]):
5         penalty += (W[i][j] ** 2)
```

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W)$$

Updating Our Loss and Weight Update To Include Regularization

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W)$$

The first term we have seen before – it is the average loss over all samples in our training set.

The second term is new – *this is our regularization penalty*. The λ variable is a hyperparameter that controls the *amount* or *strength* of the regularization we are applying. In practice, both the learning rate α and the regularization term λ are the hyperparameters that you'll spend the most time tuning.

Updating Our Loss and Weight Update To Include Regularization

Expanding cross-entropy loss to include L2 regularization yields the following equation

$$L = \frac{1}{N} \sum_{i=1}^N [-\log(e^{s_{y_i}} / \sum_j e^{s_j})] + \lambda \sum_i \sum_j W_{i,j}^2$$

We can also expand Multi-class SVM loss as well:

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} [\max(0, s_j - s_{y_i} + 1)] + \lambda \sum_i \sum_j W_{i,j}^2$$

Updating Our Loss and Weight Update To Include Regularization

Now, let's take a look at our standard weight update rule:

$$W = W - \alpha \nabla_W f(W)$$

This method updates our weights based on the gradient multiplied by a learning rate α . Taking into account regularization, the weight update rule becomes:

$$W = W - \alpha \nabla_W f(W) - \lambda R(W)$$

Here we are adding a negative linear term to our gradients (i.e., gradient descent), penalizing large weights, with the end goal of making it easier for our model to generalize.

DEMO

Thank You