Machine Learning

Stochastic Gradient Descent

Tarapong Sreenuch

8 February 2024

克明峻德，格物致知

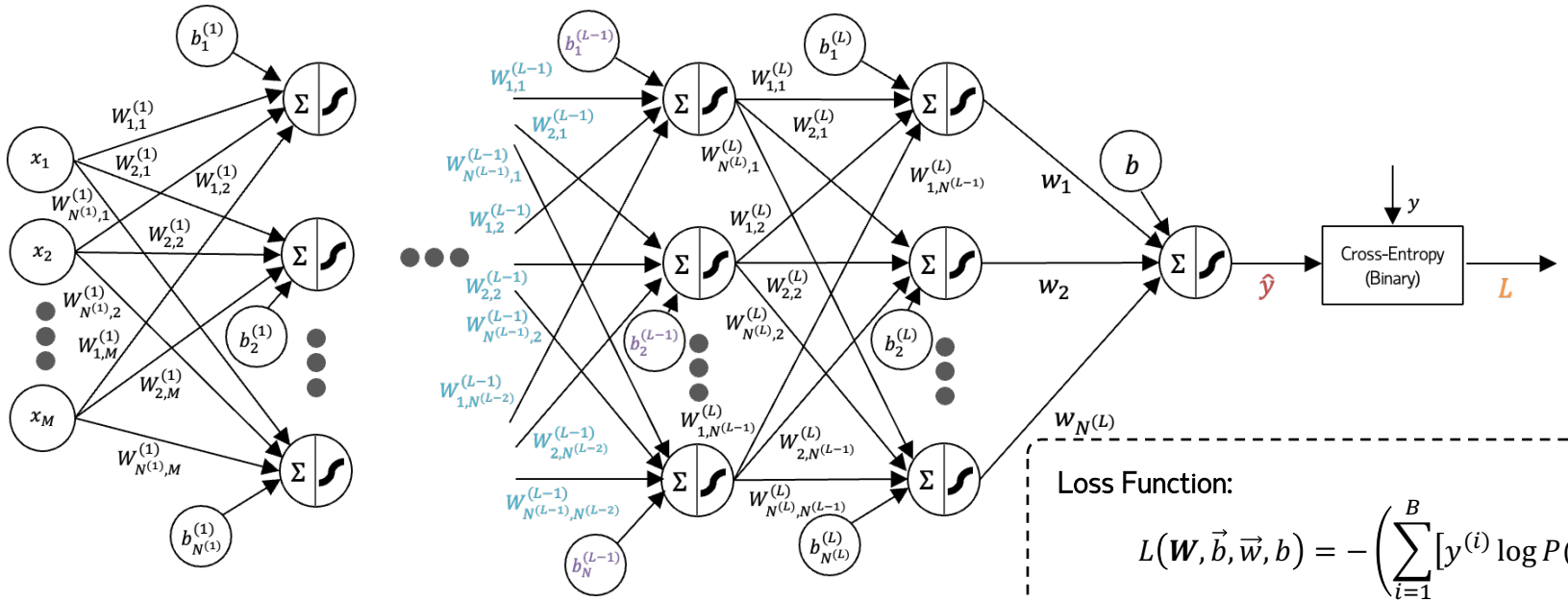
Waiting for the full tally will take hours.

Open 100 random ballots at each station—the sample count.

Q: But each sample is noisy; can we trust it?

A: Average all stations' samples; the noise cancels out—just like mini-batch SGD.

Deep Network



Forward Path:

$$\Phi^{(1)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(1)} \times \vec{x} + \vec{b}^{(1)}\right)$$

$$\Phi^{(L-1)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L-1)} \times \Phi^{(L-2)}(\vec{x}) + \vec{b}^{(L-1)}\right)$$

$$\Phi^{(L)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L)} \times \Phi^{(L-2)}(\vec{x}) + \vec{b}^{(L)}\right)$$

$$\hat{y} = \sigma\left(\vec{w}^{T} \times \Phi^{(L)}(\vec{x}) + b\right)$$

Loss Function:

$$L\left(\boldsymbol{W}, \vec{b}, \vec{w}, b\right) = -\left(\sum_{i=1}^{B}\left[y^{(i)} \log P\left(1 \middle| \vec{x}^{(i)}, \vec{w}, b\right) + \left(1 - y^{(i)}\right) \log\left(1 - P\left(1 \middle| \vec{x}^{(i)}, \vec{w}, b\right)\right)\right]\right)$$

Forward Pass: Each layer computes its output from the previous layer's numbers.

Backward Pass (Back-Prop): We now treat that loss as a function of every weight. Using the chain rule we calculate—layer by layer, in reverse order—the partial derivative of the loss with respect to each weight.

As models and datasets scale, every gradient-descent step must repeat heavy forward and backward passes, consuming massive CPU/GPU time and memory bandwidth—so training a deep network becomes a time-intensive process.

Q: Can we make it faster, but still the same result?

Why Sample? Estimating the Population with a Few Measurements

Population (N People)

Q: What are the heights of Singaporean males and females How can we determine this?

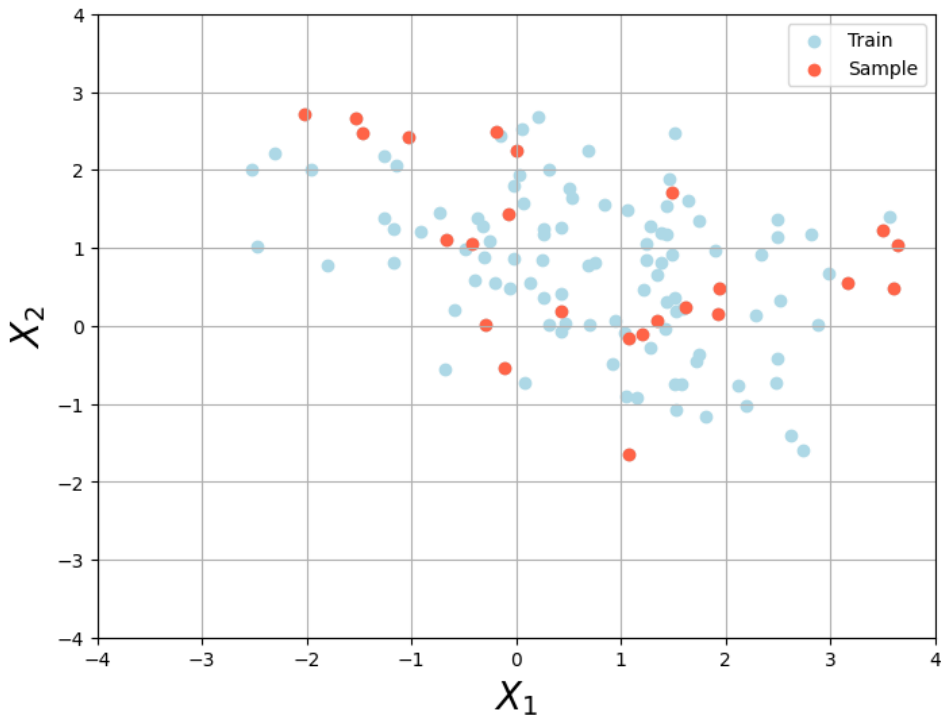
A: … Samples …

If our samples are representative, our estimates will be close to the the true values.

"Near enough is good enough."

Intuitively, estimates from samples approximate those from the full population.

Chain of Thoughts: SGD Intuition



Full-batch Gradient = Exact but Slow

Running one gradient step on the entire training set is precise, yet time-consuming when the data are large..

Single Sample? Too Narrow

Training on just one small subset is quick, but the model would never see most of the data, so the solution may be biased.

Mini-Batches = Fast Slices of the Whole

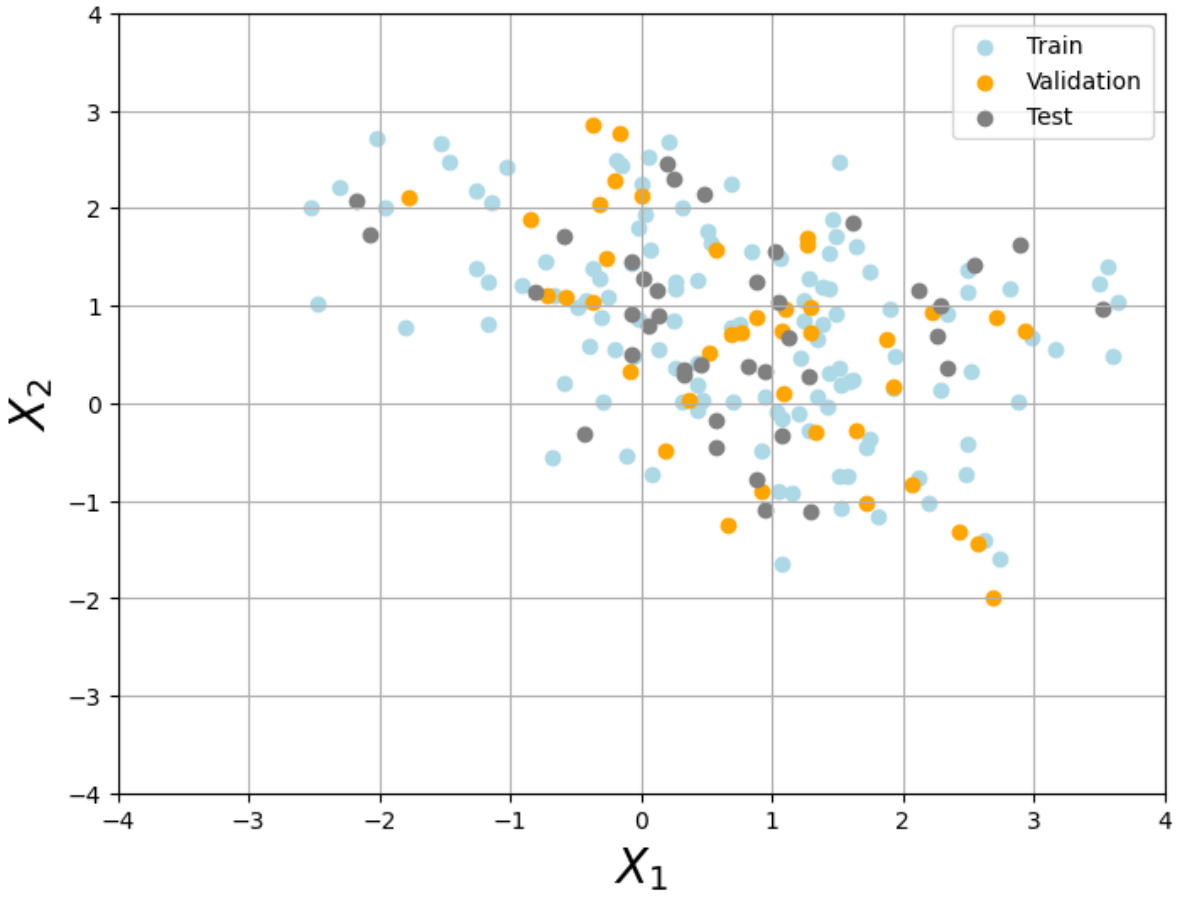
By drawing a fresh small batch each iteration, we keep each update cheap while ensuring that, over many steps, every training point is used.

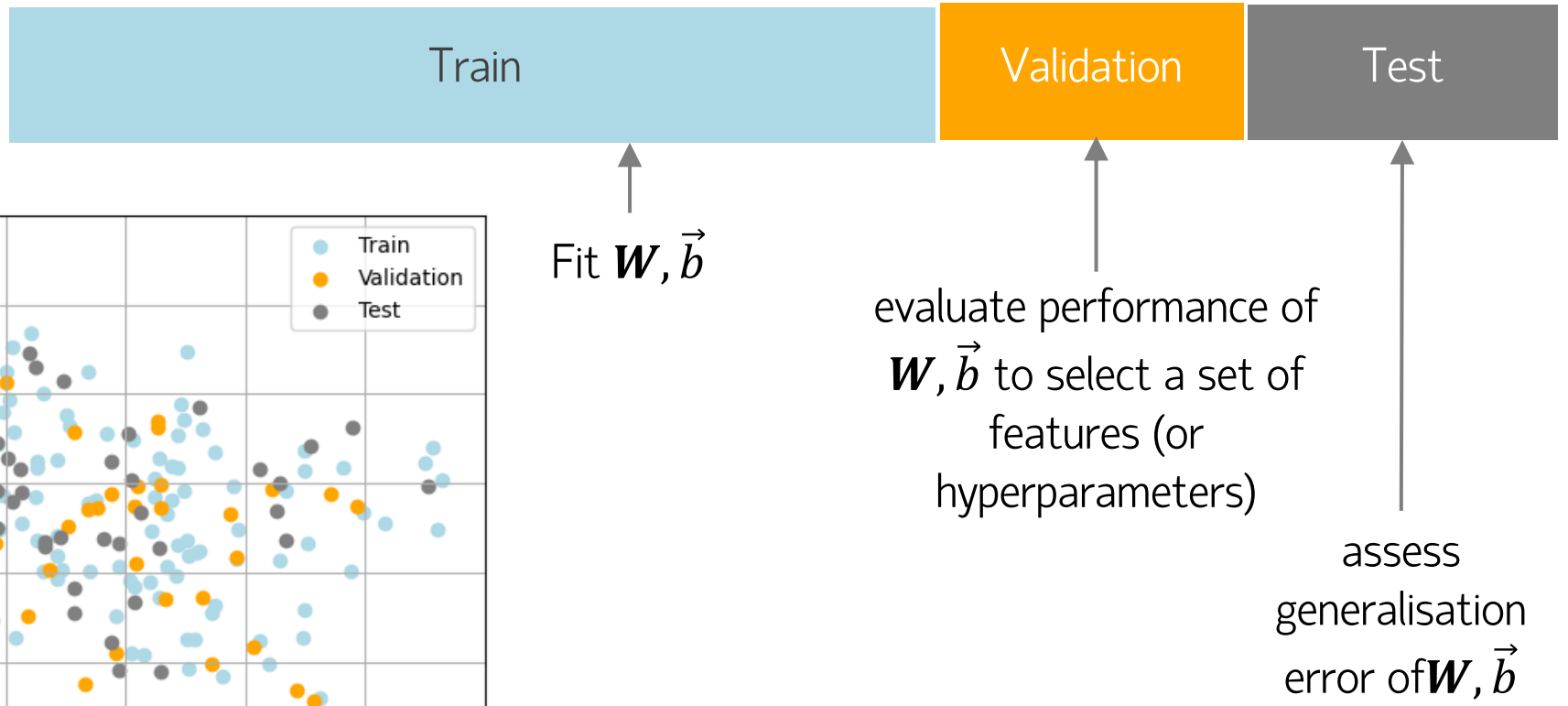
Same Destination, Less Time

As the mini-batches cycle through the dataset, their accumulated updates approximate the full-data gradient, giving nearly the same final model in far cheaper computationally.

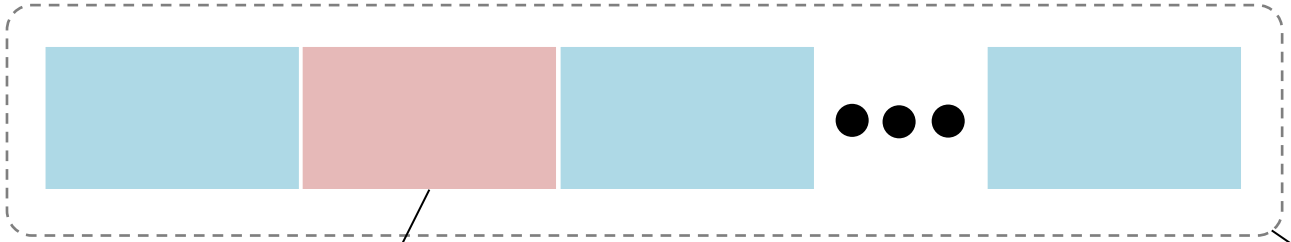
"Mini-batch SGD reaches the full-batch solution faster by trading one huge computation for many small, inexpensive ones."

Large Dataset

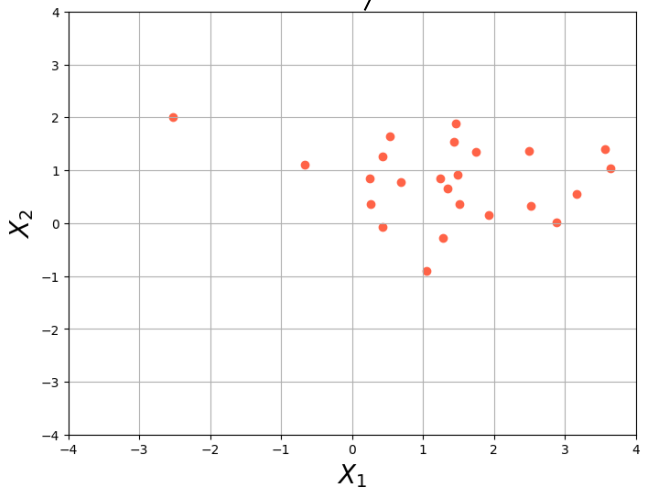




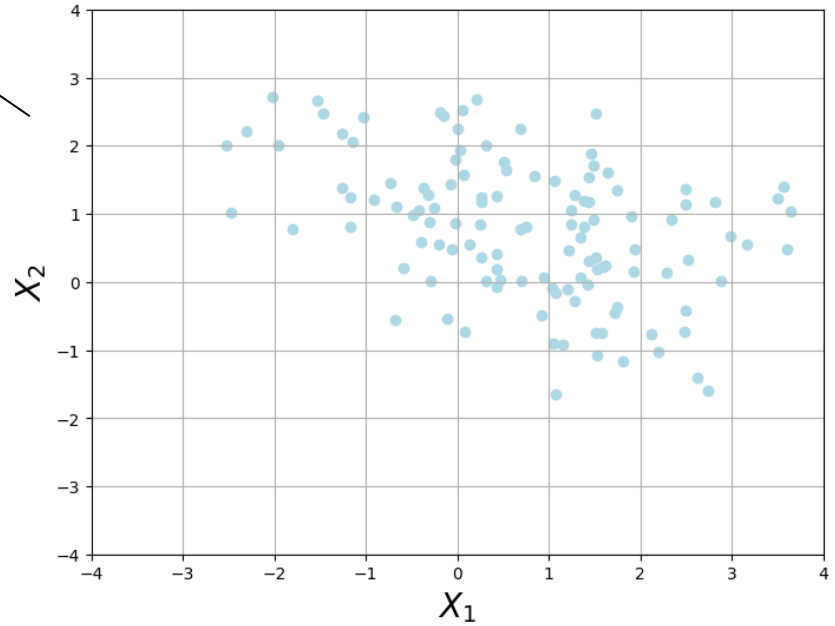
Stochastic Gradient Descent (1 of 7)



Mini-Batch

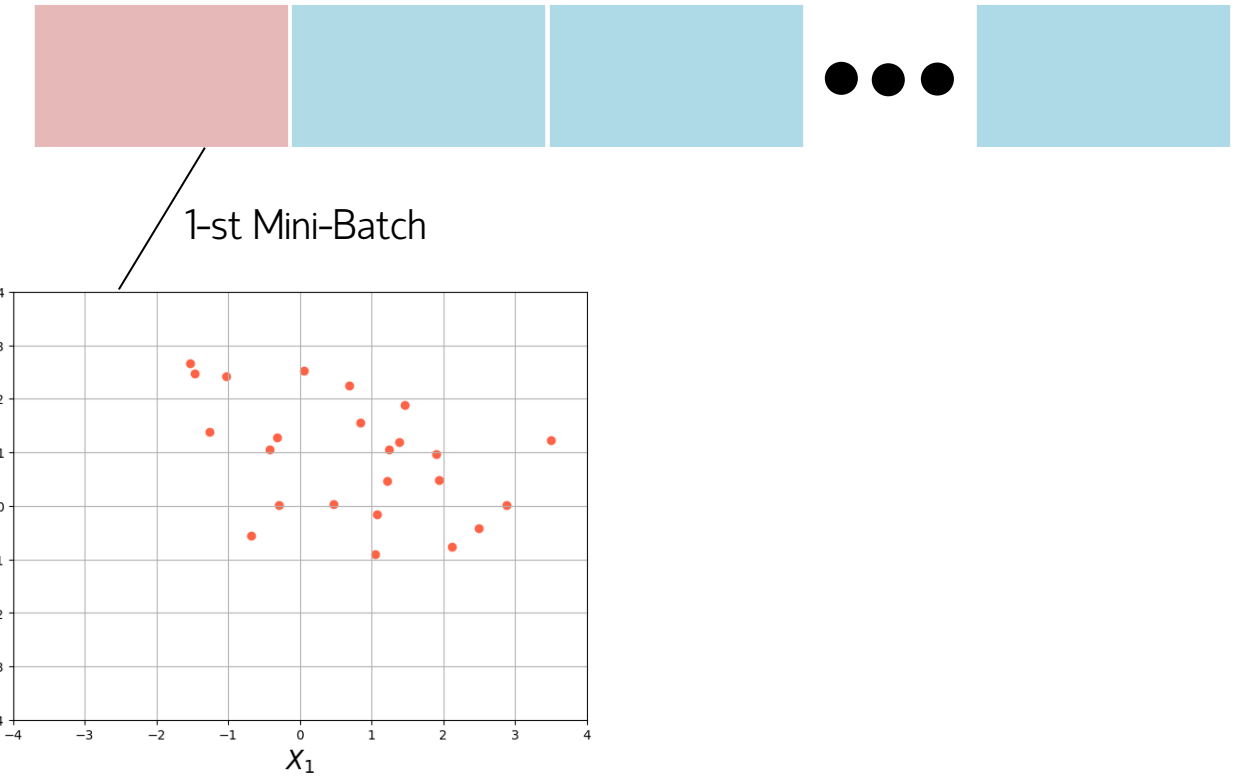


Training Data



In stochastic gradient descent (SGD), we shuffle the training set and break it into mini-batches of B data points; B is simply referred to as the batch size. $M = \lceil N/B \rceil$ is the number of batches.

Stochastic Gradient Descent (2 of 7)



The first mini-batch $\mathcal{B}^{(1)}$ is simply the first B training data points drawn after the data are shuffled.

For m = 1, ..., M ($= \lceil N/B \rceil$)

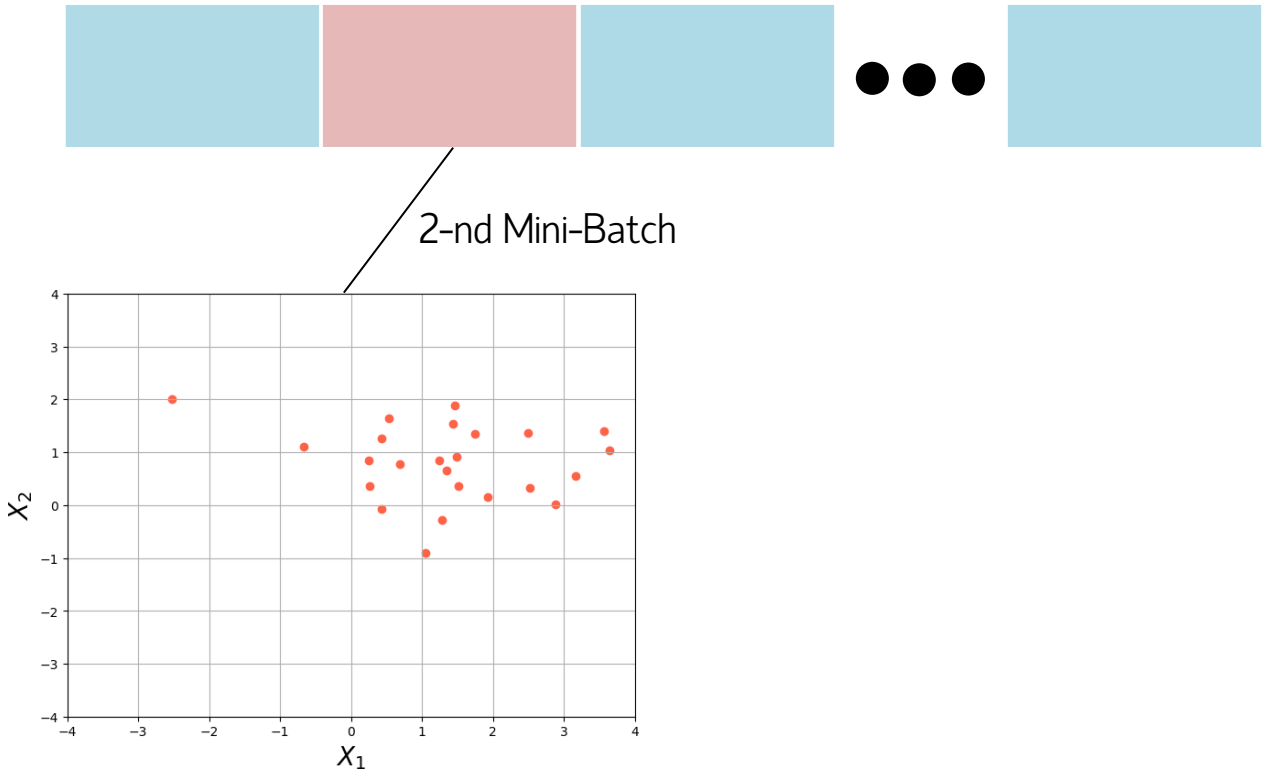
$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial W_{i,j}} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial b_i} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$t \leftarrow t + 1$$

The gradients are computed from the data points in the first mini-batch $\mathcal{B}^{(1)}$ $(m = 1)$.

Stochastic Gradient Descent (3 of 7)



The second mini-batch $\mathcal{B}^{(2)}$ contains a fresh, non-overlapping set of training points, completely separate from those in the first mini-batch $\mathcal{B}^{(1)}$.

For m = 1, ..., M ($= \lceil N/B \rceil$)

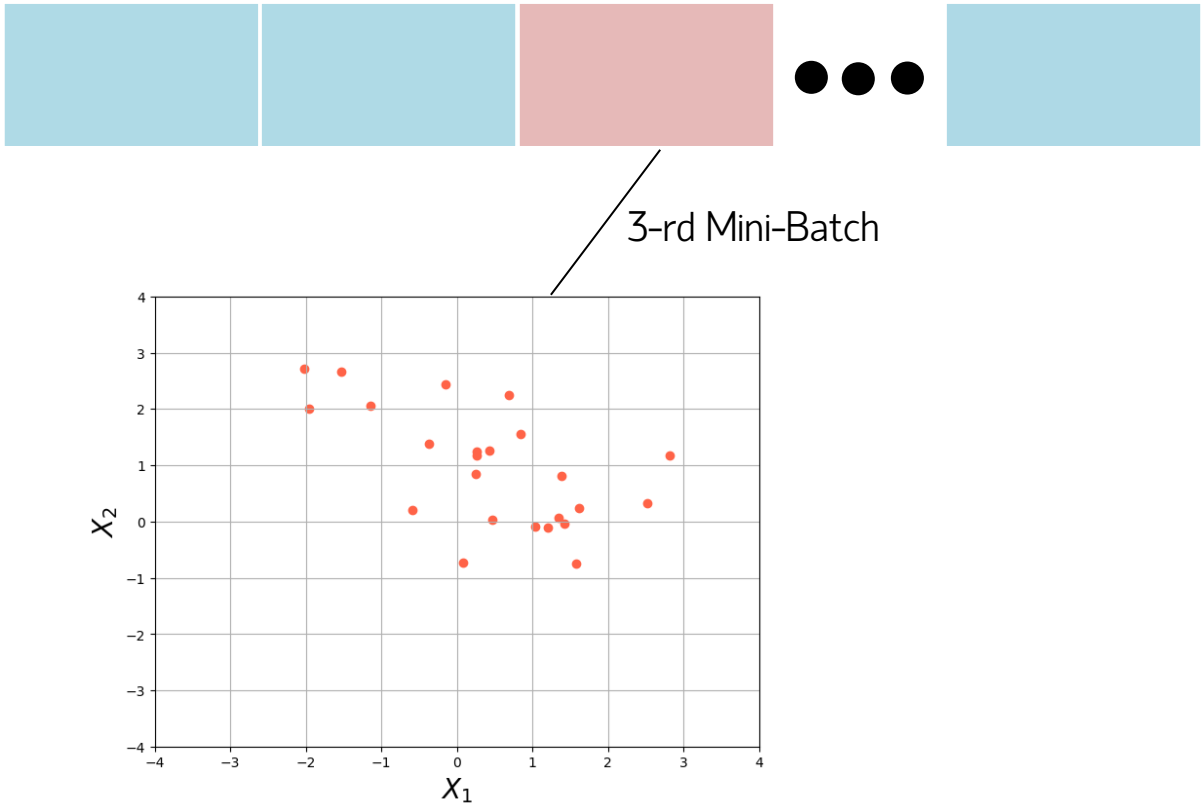
$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial W_{i,j}} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial b_i} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$t \leftarrow t + 1$$

The gradients are computed from the data points in the first mini-batch $\mathcal{B}^{(2)}$ $(m = 2)$.

Stochastic Gradient Descent (4 of 7)



The third mini-batch $\mathcal{B}^{(3)}$ contains a fresh, non-overlapping set of training points, completely separate from those in the first $\mathcal{B}^{(1)}$ and second $\mathcal{B}^{(2)}$ mini-batches.

For m = 1, ..., M ($= \lceil N/B \rceil$)

$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial W_{i,j}} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

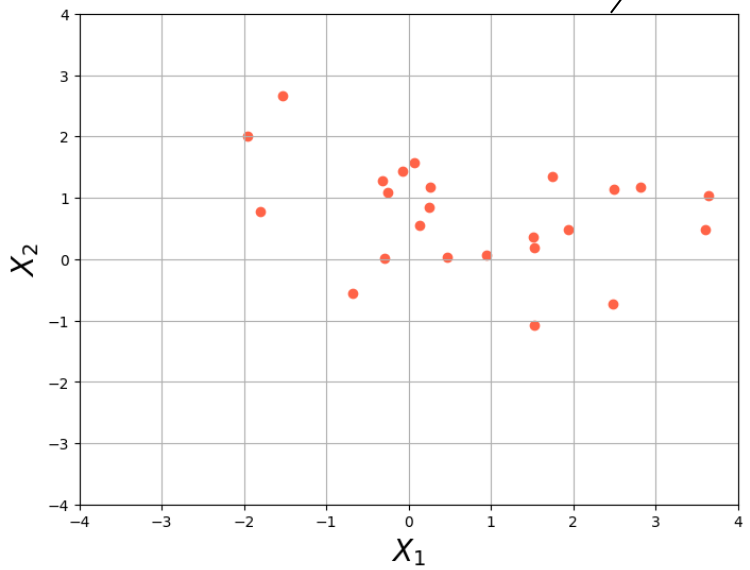
$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial b_i} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$t \leftarrow t + 1$$

The gradients are computed from the data points in the first mini-batch $\mathcal{B}^{(3)}$ $(m = 3)$.

Stochastic Gradient Descent (5 of 7)

M-th Mini-Batch



For m = 1, …, M (= $\lceil N/B \rceil$)

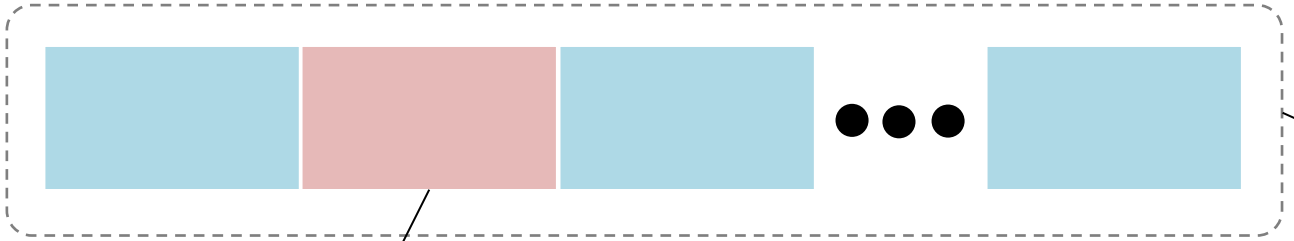
$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial W_{i,j}} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial b_i} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

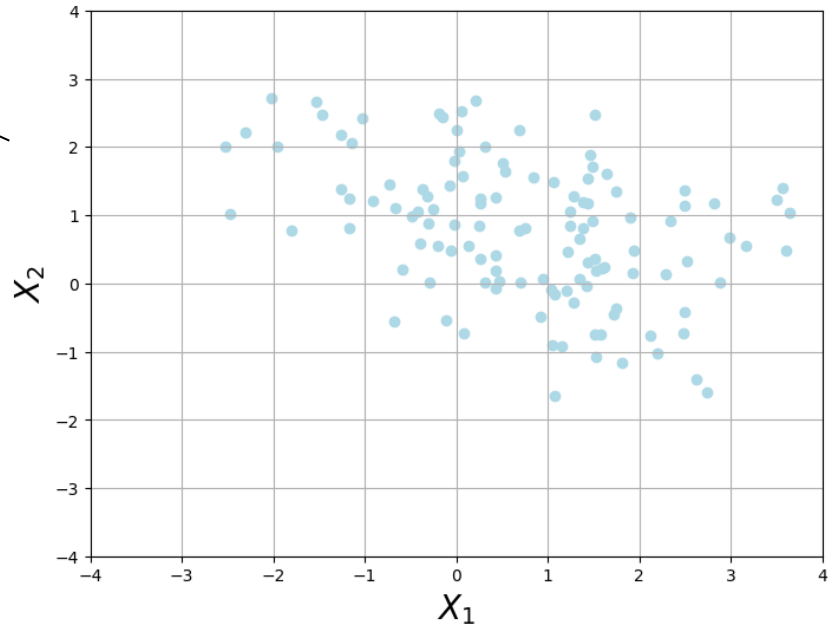
$$t \leftarrow t + 1$$

When we reach $m = M$, every mini-batch has been processed once, all training samples have been seen, and we count that pass through the data as one full epoch

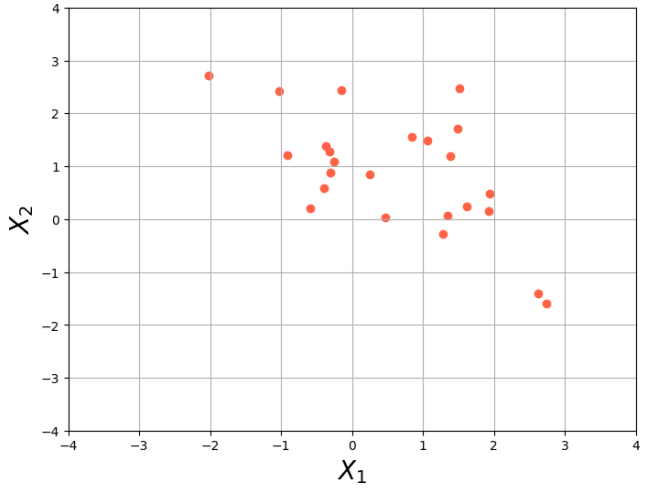
Stochastic Gradient Descent (6 of 7)



Training Data



Mini-Batch

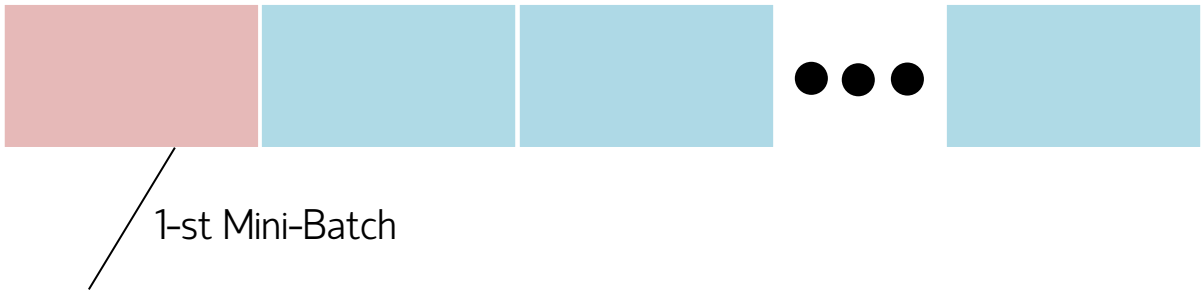


At the start of each new epoch we shuffle the training data again and slice it into mini-batches of size B. As a result, the data points in the second mini-batch of the new epoch differ from those in the second mini-batch of the previous epoch.

We re-shuffle at each epoch to ensure:

- Every example appears in a different mini-batch order each epoch.
- Gradient estimates remain unbiased, with no systematic pattern.
- The model sees a fresh mix of data on every update, improving convergence and generalisation.

Stochastic Gradient Descent (7 of 7)



For m = 1, ..., M ($= \lceil N/B \rceil$)

$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial W_{i,j}} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial b_i} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$t \leftarrow t + 1$$

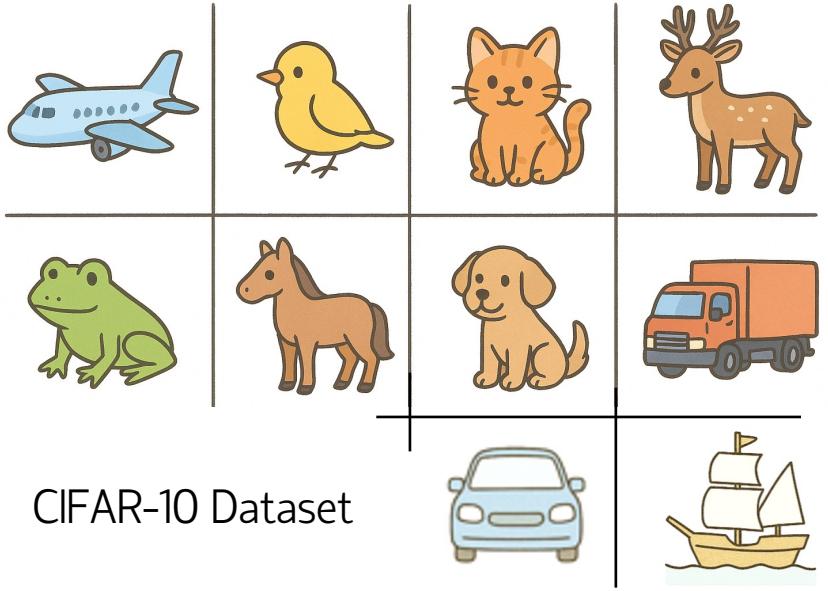
At the start of the new epoch, we draw the first mini-batch $\mathcal{B}^{(1)}$, run a gradient update on it, and then repeat the same process for each subsequent mini-batch until the whole training set has been covered again.

Pseudocode for Stochastic Gradient Descent

```
# Inputs
#   data      ← training set of N examples (x, y)
#   B         ← mini-batch size
#   n         ← learning-rate
#   epochs    ← number of full passes over the data
#   W, b      ← model weights and biases          (initialised)

FOR epoch = 1 TO epochs DO
    shuffle(data)                                # random order each epoch

    split data into consecutive mini-batches 𝓑¹, 𝓑², …, 𝓑ᴹ
        where M = ceil(N / B)

    FOR each mini-batch 𝓑ᵐ DO
        # ----- forward pass -----
        ŷ ← model(𝓑ᵐ.x ; W, b)                   # predictions for this batch
        L ← loss(ŷ, 𝓑ᵐ.y)                        # average loss over the batch

        # ----- backward pass -----
        g_W, g_b ← ∇_{W,b} L                     # gradients w.r.t. W and b

        # ----- parameter update -----
        W ← W - n(epoch, m) · g_W
        b ← b - n(epoch, m) · g_b
    END FOR
END FOR
```

Batches: Calculation Example

Component	Details
Input (X)	32 × 32 RGB Image → Tensor Shape 32 × 32 × 3 (3,072 float / uint8 Values)
Label (y)	One of 10 Classes: 0 Airplane · 1 Automobile · 2 Bird · 3 Cat · 4 Deer · 5 Dog · 6 Frog · 7 Horse · 8 Ship · 9 Truck
Training set	50,000 Image–Label Pairs
Test set	10,000 Image–Label Pairs



CIFAR-10 Dataset

Let's reserve 10 k images for validation, use the remaining 40 k for training, and set the batch size to 128.

Q: How many batches will we have?

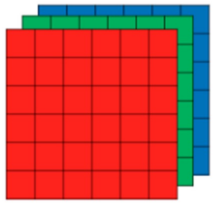
A: $M = \lceil N/B \rceil$. *40k/128 = 312.5 → 313.*

Q: How many data points are there in the 313-th mini-batch?

A: 40k – 312x128 = 64.

CIFAR-10 Pre-processing: Pixels → Flatten → Normalise

1



3-channel
Image Data

3x[32,32]

2

$$\begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 92 \\ 142 \end{bmatrix}$$

Flatten Data

3x32x32 = *3072*

3

$$\begin{bmatrix} 1.0 \\ 0.906 \\ 0.168 \\ \vdots \\ 0.361 \\ 0.557 \end{bmatrix}$$

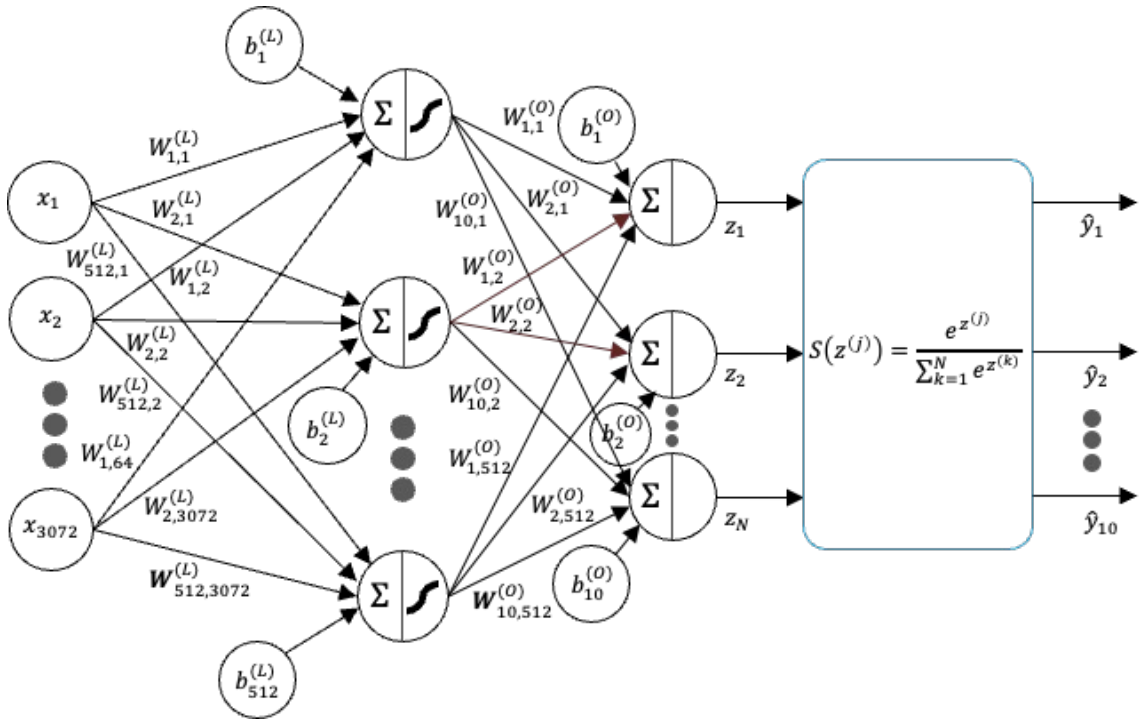
Normalised Data

3072

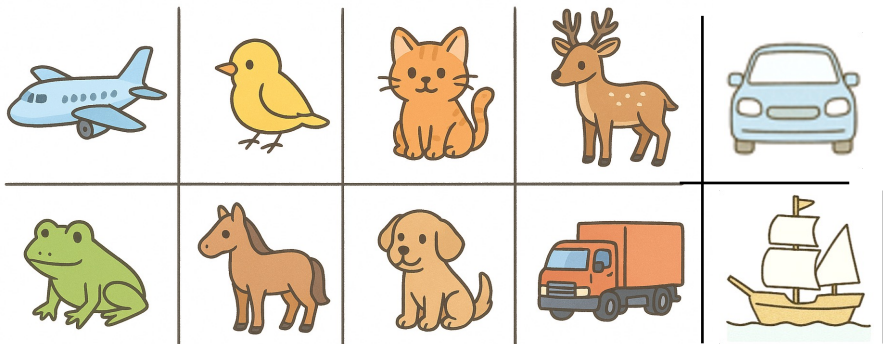
```
from tensorflow.keras import datasets, utils
import numpy as np

# Load CIFAR-10 (50 k train / 10 k test)
(x_train_full, y_train_full), (x_test, y_test) = datasets.cifar10.load_data()

# Flatten images → 32*32*3 = 3072 features
x_train_full = x_train_full.reshape(len(x_train_full), -1)
x_test = x_test .reshape(len(x_test), -1)

# Normalise pixels to [0,1]
x_train_full = x_train_full.astype("float32") / 255.0
x_test = x_test .astype("float32") / 255.0

# One-hot encode labels
num_classes = 10
y_train_full = utils.to_categorical(y_train_full, num_classes)
y_test = utils.to_categorical(y_test, num_classes)

# Train / validation split (40 k / 10 k)
x_train, x_val = x_train_full[:40_000], x_train_full[40_000:]
y_train, y_val = y_train_full[:40_000], y_train_full[40_000:]
```

TensorFlow

Tensorflow: Mini-Batch SGD

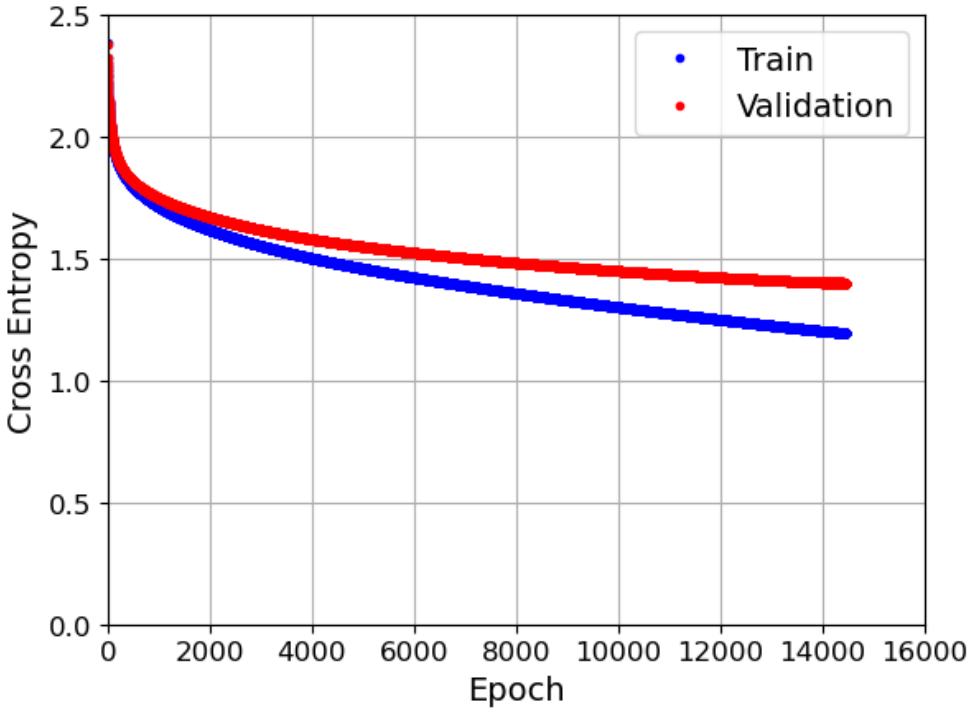


CIFAR-10 Dataset



```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import SGD

num_inputs = 3072         # input dimension
num_hidden_units = 512    # hidden units
num_outputs = 10          # output dimension

model = Sequential([
    Dense(num_hidden_units,
          activation=”relu",
          input_shape=(num_inputs,)),

    Dense(num_outputs,
          activation="softmax”)
])

model.compile(optimizer=SGD(learning_rate=1e-2),
              loss=”categorical_crossentropy",
              metrics=[“accuracy”])

batch_size = 128        # batch size
num_epochs = 100        # number of epochs

early_stop = EarlyStopping(monitor=‘val_loss’,
                           restore_best_weights=True, patience=5)

history = model.fit(X_train, y_train,
                    batch_size=batch_size,
                    epochs=num_epochs,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stop])
```

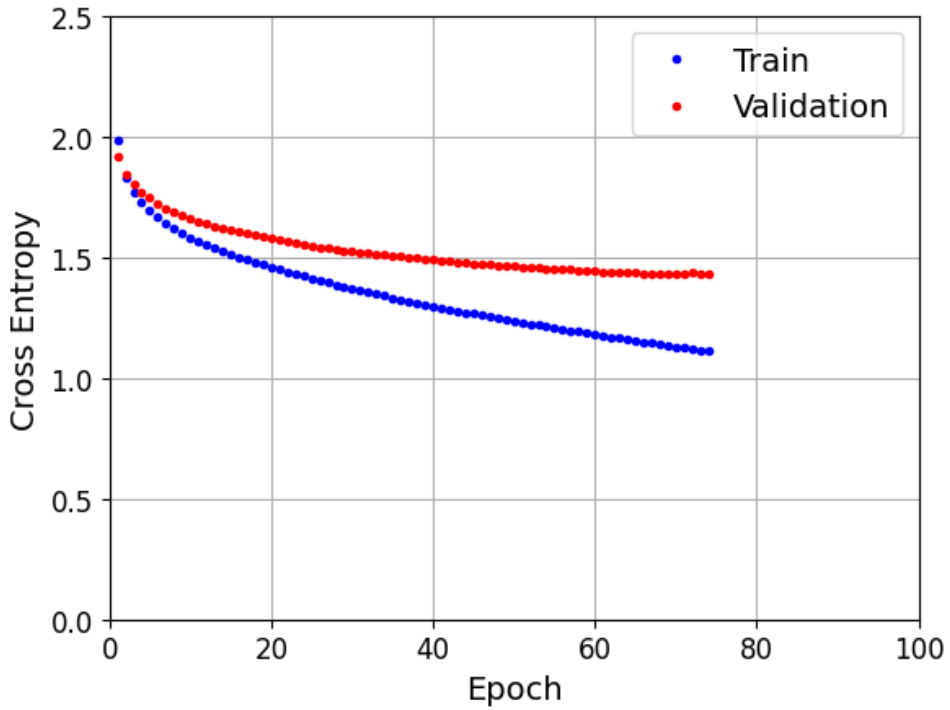
Mini-Batch SGD



- Batch Size: 40,000 (Full Training Data)
- Wall-Clock Time: 14,359 Seconds
- Validation Loss: 1.401
- Validation Accuracy: 0.513

The runs were on Colab with T4 GPU notebook setting.

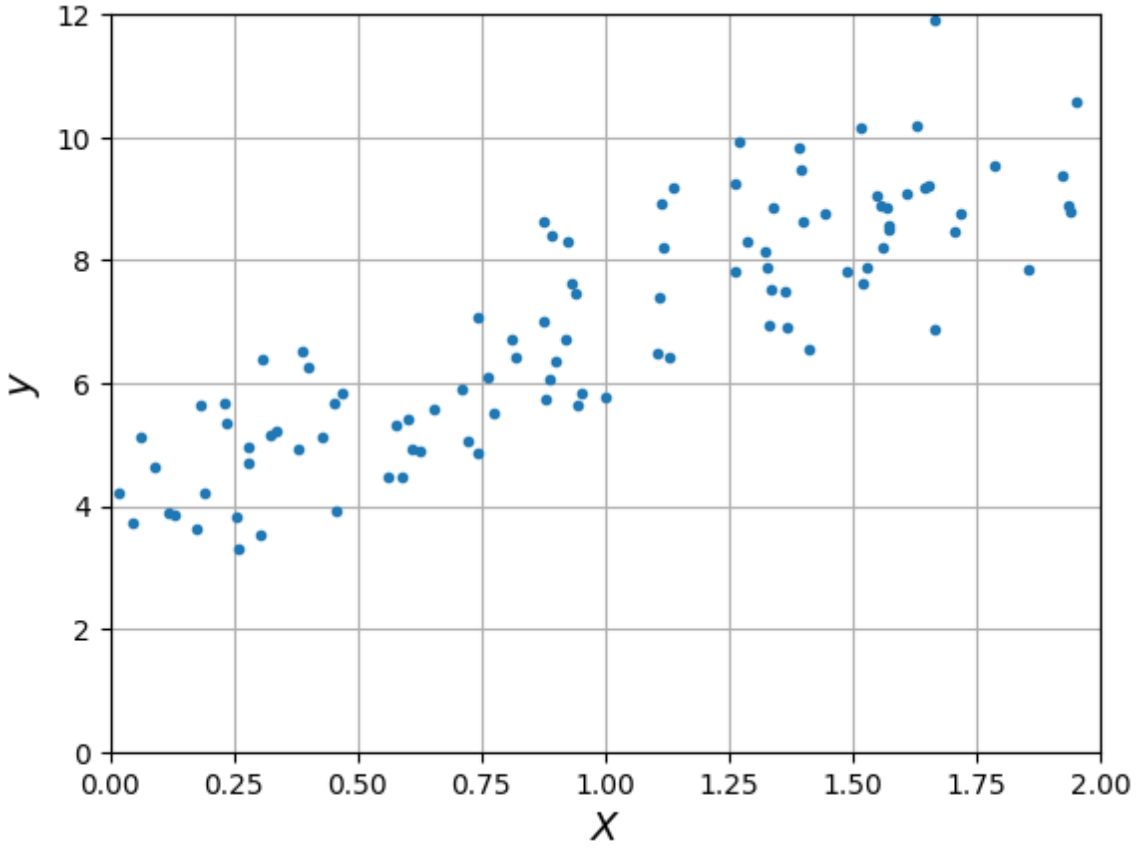
Google Colaboratory



- Batch Size: 128
- Wall-Clock Time 97 Seconds
- Validation Loss: 1.438
- Validation Accurcy: 0.504

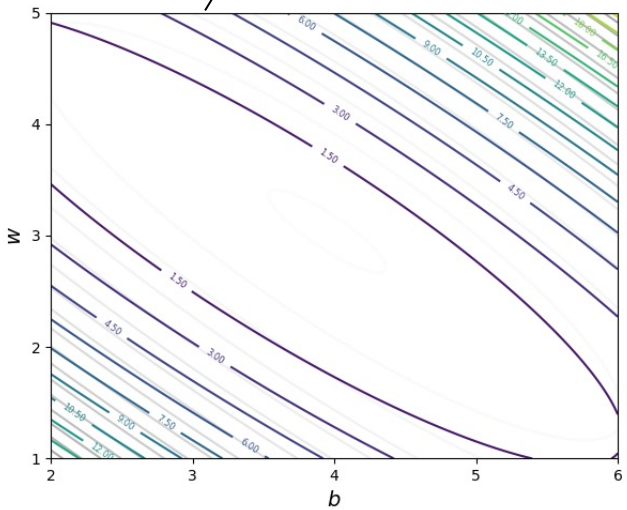
Comparable Accuracy in 1% of the Time

Example Dataset



Mini-Batch Loss and Gradient

1-st Mini-Batch



The loss surface is computed only from the 24 data points in $\mathcal{B}^{(1)}$ $(m = 1)$.

For m = 1, …, M $(= \lceil N/B \rceil)$

$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial W_{i,j}} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

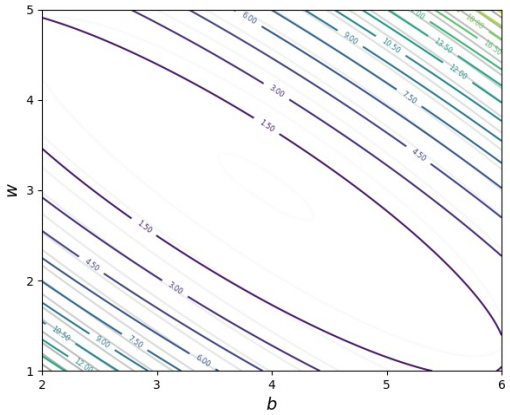
$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} \frac{\partial L}{\partial b_i} (W^{(t)}, \vec{b}^{(t)}, \vec{x}^{(m)(k)}) \right)$$

$$t \leftarrow t + 1$$

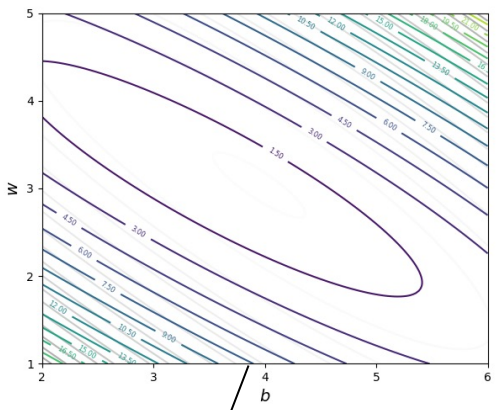
Each iteration uses just one mini-batch to compute both the loss surface and its gradient; the process repeats batch-by-batch until the epoch completes.

Loss Contours

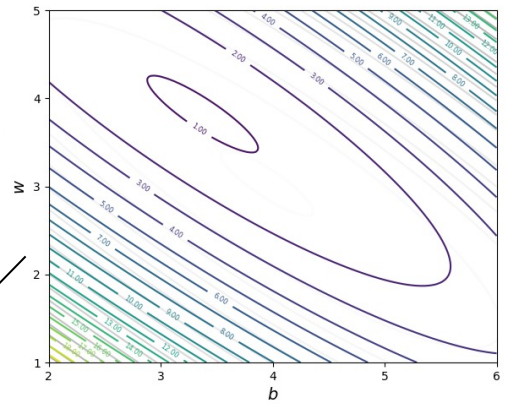
1st Mini-Batch Loss Surface

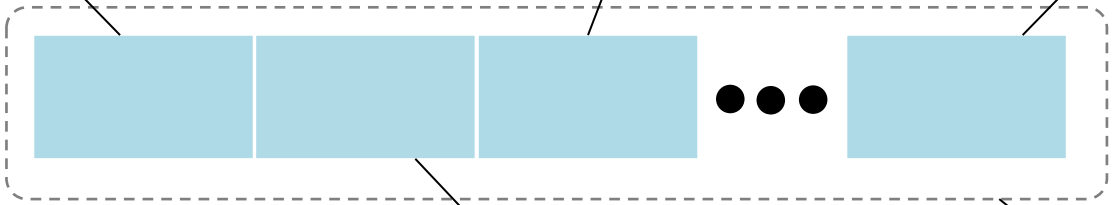


3rd Mini-Batch Loss Surface

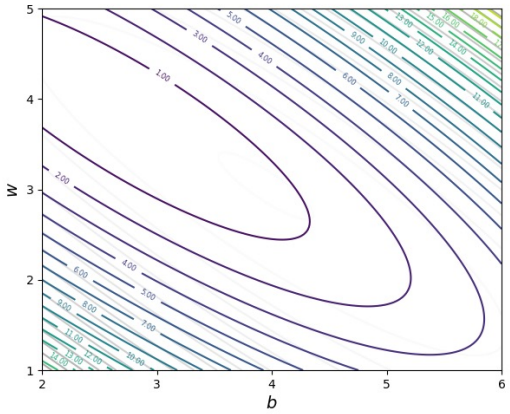


Mth Mini-Batch Loss Surface

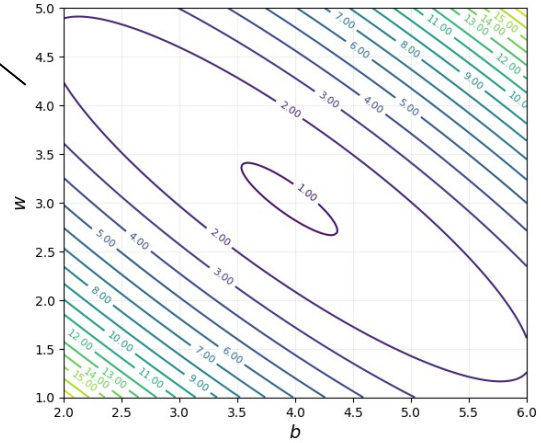




2nd Mini-Batch Loss Surface

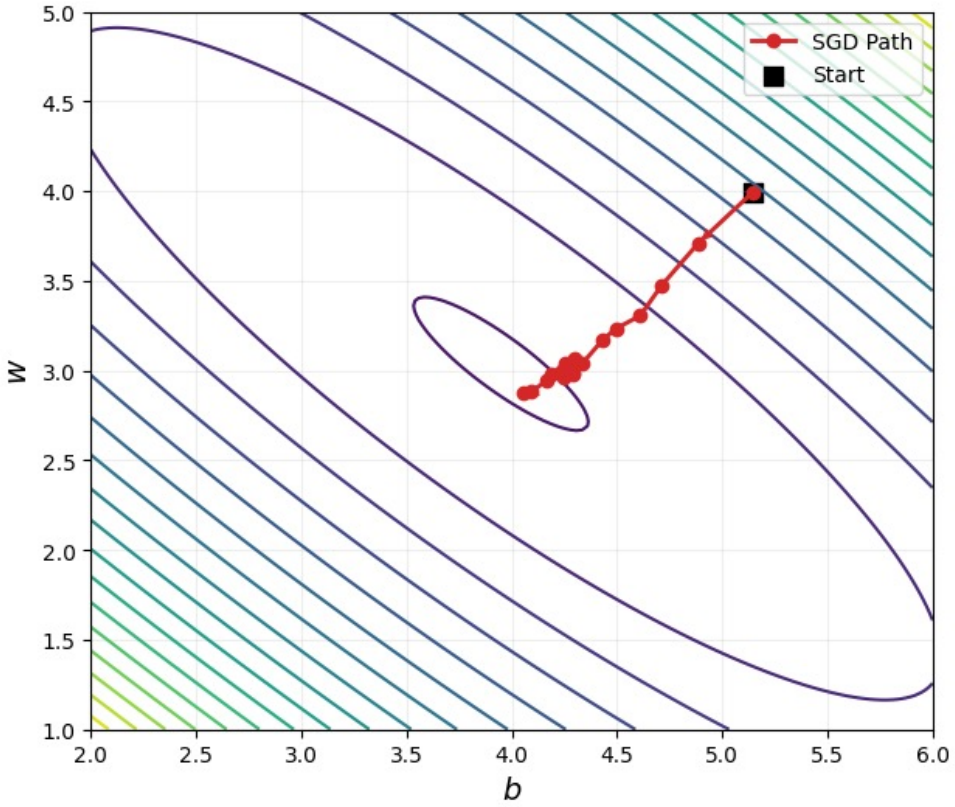


Full Training Loss

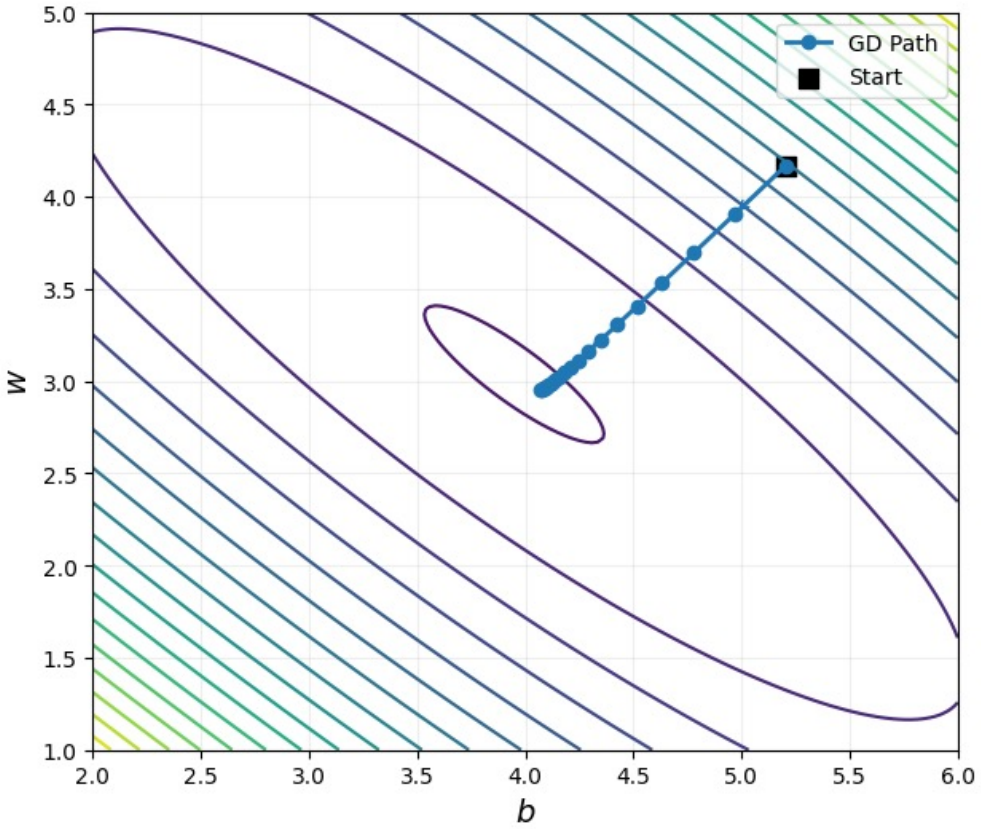


Each batch contains fraction of the data, so its contours wobble around the full-data ellipse, yet the overall shape still points toward the global minimum.

Weight-Update Trajectories: SGD vs Full-Batch Gradient Descent

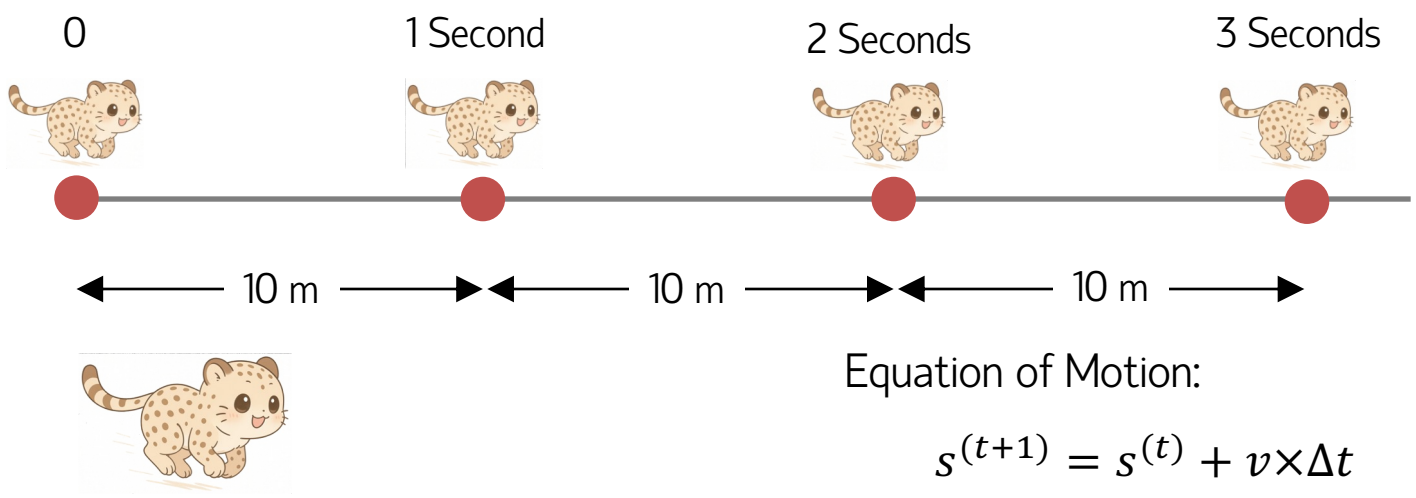


SGD follows the "shifting loss surface". By stepping through one mini-batch after another, SGD explores a series of approximate loss landscapes—cheap to compute, yet accurate enough to guide the weights toward the true minimum.



Full-batch GD follows a smooth, direct descent. Because every update uses the *exact* gradient of the whole training set, the path glides along a single, unchanging loss surface. There is no zig-zagging or back-tracking caused by mini-batch noise.

Gradient Descent



10 m/s of Constant Velocity (v)

Equation of Motion:

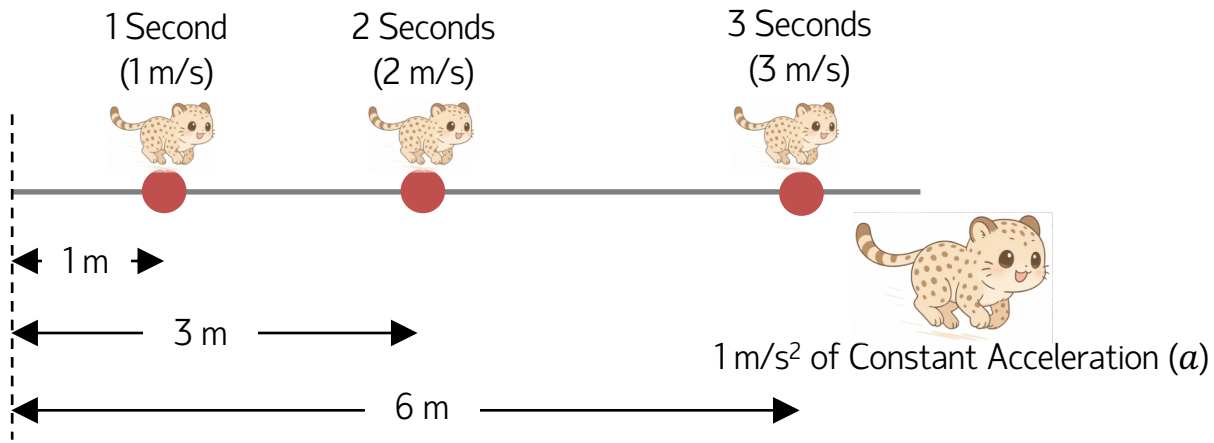
$$s^{(t+1)} = s^{(t)} + v \times \Delta t$$

Gradient ≈ Velocity

Gradient Descent:

$$w^{(t+1)} = w^{(t)} - \eta \times \frac{\partial L}{\partial w}(w^{(t)})$$

*In GD, the next weight value $w^{(t+1)}$ is computed **only from** the gradient $\frac{\partial L}{\partial w}$ **evaluated at the current weight** $w^{(t)}$.*



1 m/s^2 of Constant Acceleration (a)

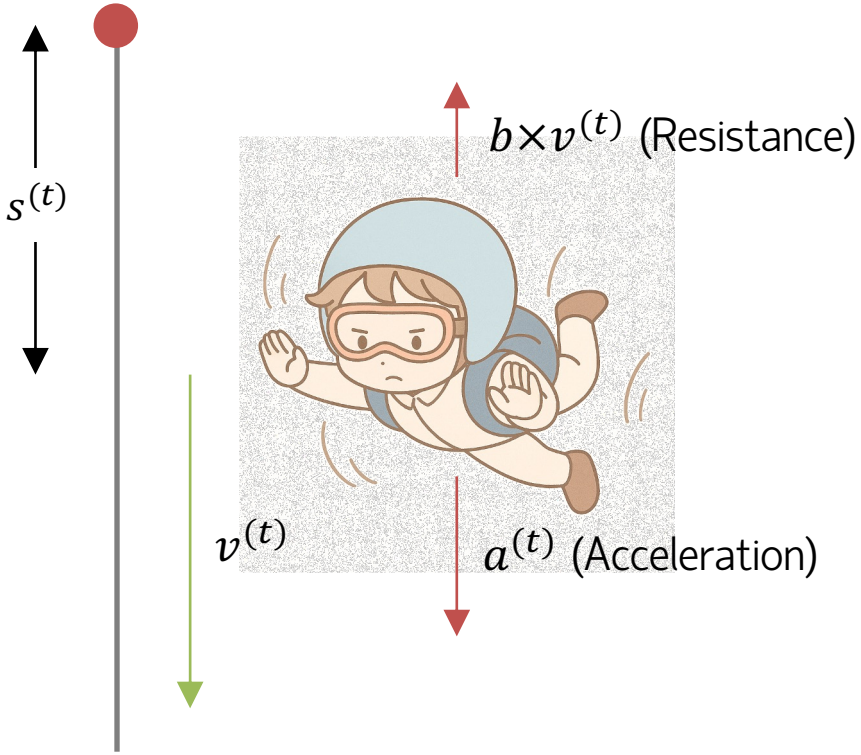
Forward-Euler: (Discrete-Time Approximation)

$$v^{(t+1)} = v^{(t)} + a^{(t)} \times \Delta t$$

$$s^{(t+1)} = s^{(t)} + v^{(t+1)} \times \Delta t$$

Euler Error: $\mathcal{O}(\Delta t) \to 0$ if $\Delta t \to 0$

Momentum Algorithm



Forward-Euler: (Discrete-Time Approximation)

$$v^{(t+1)} = v^{(t)} + \left(a^{(t)} - b \times v^{(t)}\right) \times \Delta t$$

$$= (1 - b \times \Delta t) \times v^{(t)} + a^{(t)} \times \Delta t$$

$$s^{(t+1)} = s^{(t)} + v^{(t+1)} \times \Delta t$$

Euler Error: $\mathcal{O}(\Delta t) \to 0$ if $\Delta t \to 0$

Gradient ≈ Acceleration

Momentum Algorithm:

$$v^{(t+1)} = \beta \times v^{(t)} - \eta \times \frac{\partial L}{\partial w}(w^{(t)})$$

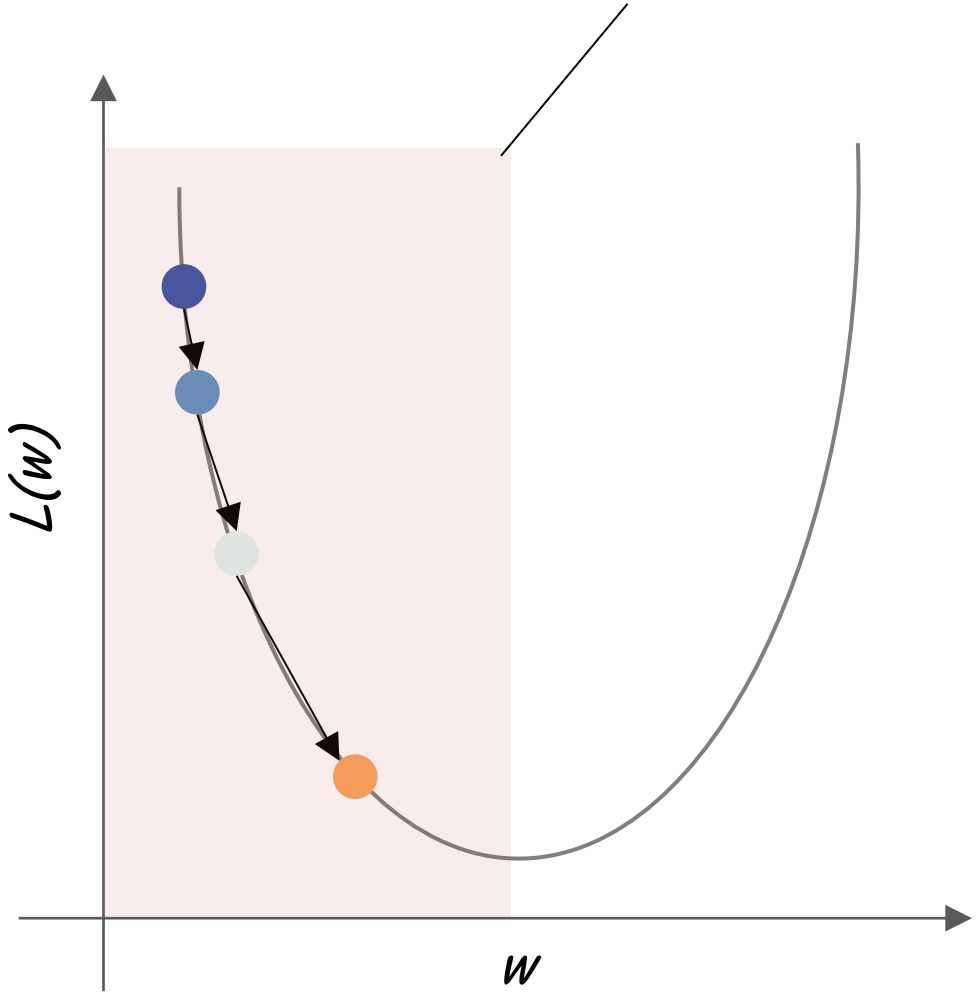
$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

In momentum, the next weight $w^{(t+1)}$ is updated from the current gradient $\frac{\partial L}{\partial w}(w^{(t)})$ plus a velocity term $v^{(t)}$ that aggregates gradients from previous iterations.

β controls how much past gradients remain in the velocity term — larger β (commonly ≈0.9) keeps older gradients influential, smaller β lets them fade quickly.

$\beta \times v^{(t)} \approx$ Decay Term $(1 - b \times \Delta t) \times v^{(t)}$. It limits how large the velocity can grow.

Faster Descent on One-Sided Slopes



Momentum Algorithm:

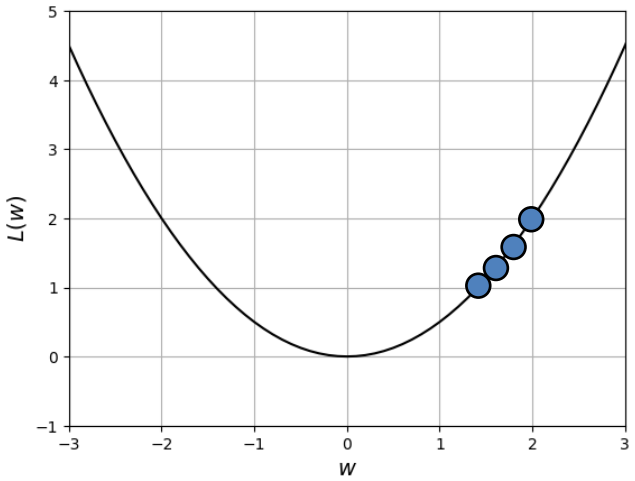
$$v^{(t+1)} = \beta \times v^{(t)} - \eta \times \frac{\partial L}{\partial w}(w^{(t)})$$

$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

When successive gradients keep the same sign (we are still on one side of the valley), the velocity $v^{(t)}$ accumulates:

- Each new gradient adds to the stored velocity.
- The update step therefore grows with every iteration.
- Result: the weights travel farther each step, so we reach the minimum faster.

SGD with Momentum

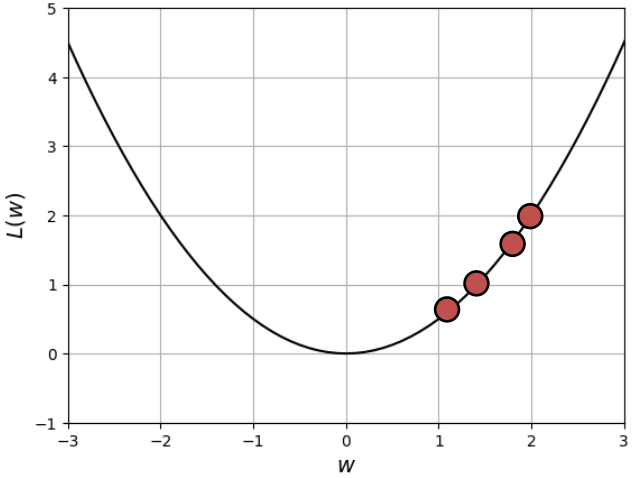


Gradient Descent Algorithm:

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \times \frac{\partial L}{\partial w}\left(w^{(t)}\right)$$

Iteration	SGD Weight $\left(w^{(t)}\right)$	SGD Step $\left(\eta \frac{\partial L}{\partial w}\left(w^{(t)}\right)\right)$
0	2.000	–
1	1.800	0.200
2	1.620	0.180
3	1.458	0.162

Setup: $\eta = 0.1,\ \beta = 0.9,\ w^{(0)} = 2.0$



Momentum Algorithm:

$$v^{(t+1)} \leftarrow \beta \times v^{(t)} - \eta \times \frac{\partial L}{\partial w}\left(w^{(t)}\right)$$

$$w^{(t+1)} \leftarrow w^{(t)} + v^{(t+1)}$$

Iteration	SGDM Weight $\left(w^{(t)}\right)$	SGDM Gradient $\left(\eta \frac{\partial L}{\partial w}\left(w^{(t)}\right)\right)$	SGDM Step $\left(v^{(t)}\right)$
0	2.000	–	–
1	1.800	0.200	0.200
2	1.440	0.180	0.360
3	0.972	0.144	0.468

- When successive gradients point the same way, $v^{(t)}$ compounds: steps grow (0.200 → 0.468) and the optimiser "builds speed."
- As the slope flattens or flips sign, the new gradient opposes the stored $v^{(t)}$. Velocity shrinks, so steps shorten *without* manual LR tweaks.

Velocity Accumulates… Until What Slows It Down?

Momentum Algorithm:

$$v^{(t+1)} = \beta \times v^{(t)} - \eta \times \frac{\partial L}{\partial w}(w^{(t)})$$

$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

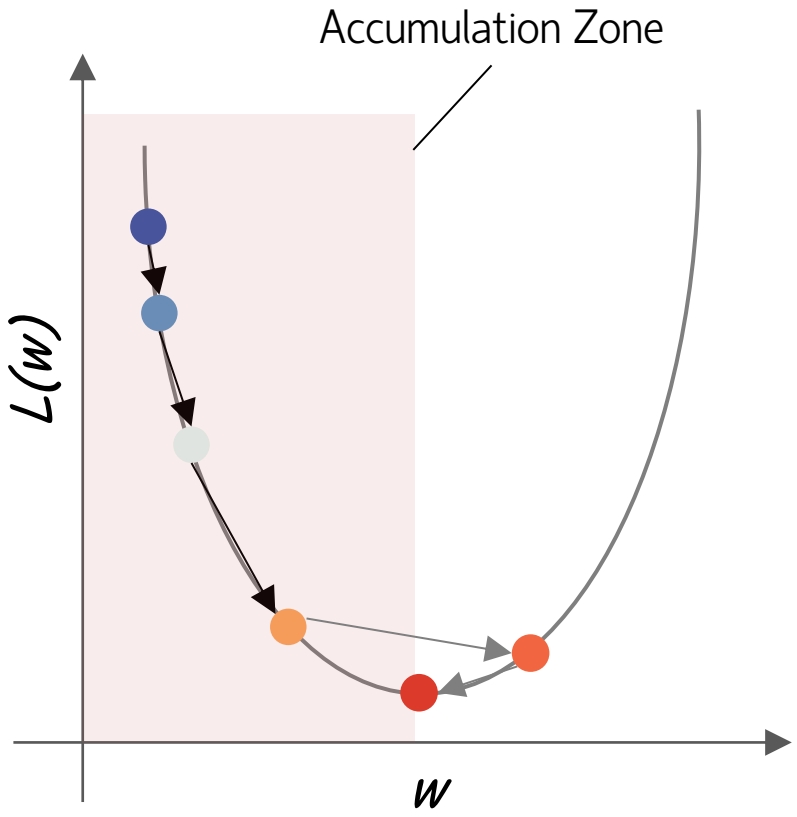
Q: Which value of v will w converge?

A: $v \approx 0$.

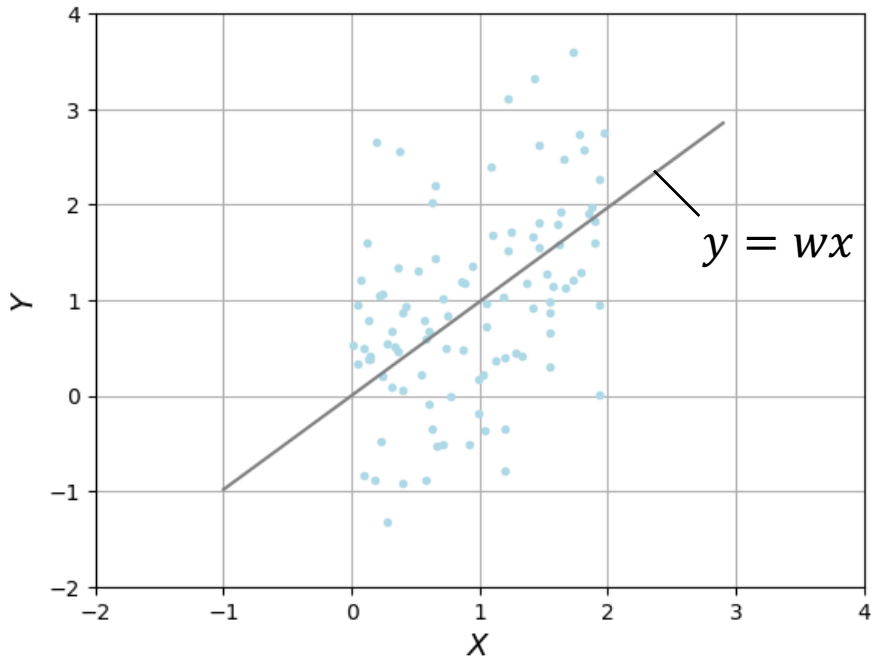
Gradient Sign Constant → Velocity Increases

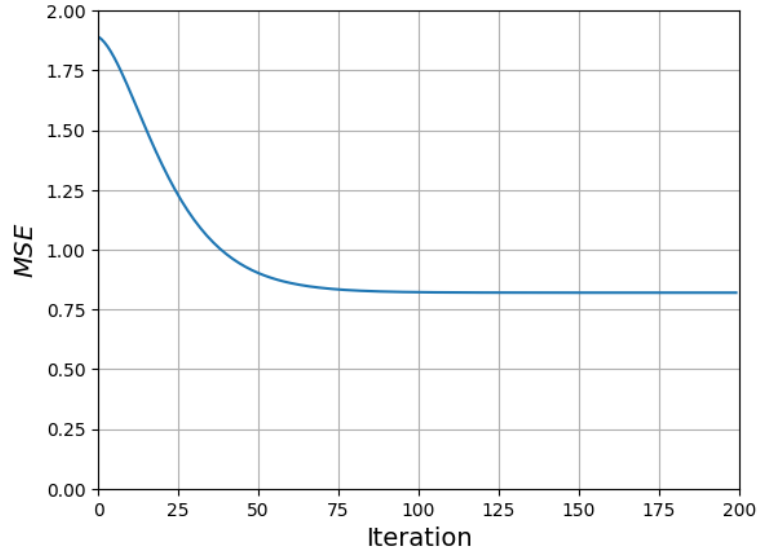
Q: Won't the weights cross the valley floor and flip the gradient and bring it back toward zero?

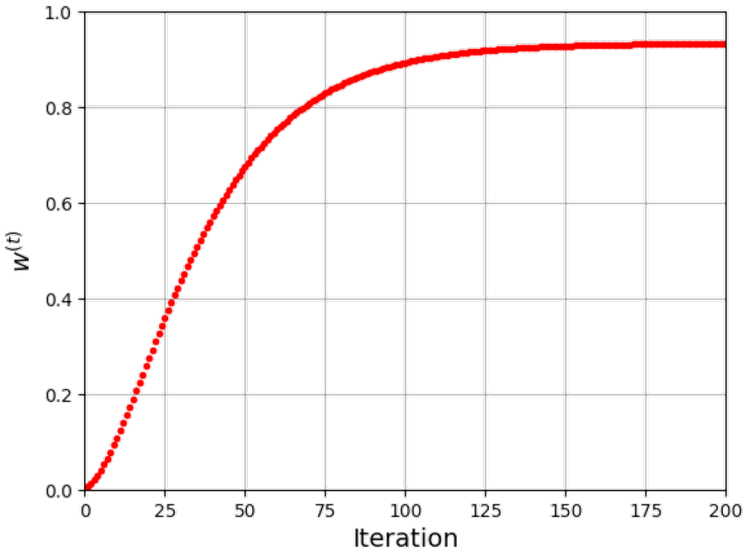
A: Not necessary. If $\beta = 1$, then 'yes'.



Toy Linear-Regression Run







$w = 0.98, b = 0$

$MSE = 0.82$

The loss falls sharply during the first 20 iterations because momentum allows larger steps while consecutive gradients continue to point in the same direction.

Velocity Builds, Then Fades

Momentum Algorithm:

$$v^{(t+1)} = \beta v^{(t)} - \eta \frac{\partial L}{\partial w}(w^{(t)})$$

$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

$$v^{(t+1)} = \beta v^{(t)} - \eta \frac{\partial L}{\partial w}(w^{(t)})$$

$$v^{(t+2)} = \beta^2 v^{(t)} - \beta\eta \frac{\partial L}{\partial w}(w^{(t)}) - \eta \frac{\partial L}{\partial w}(w^{(t+1)})$$

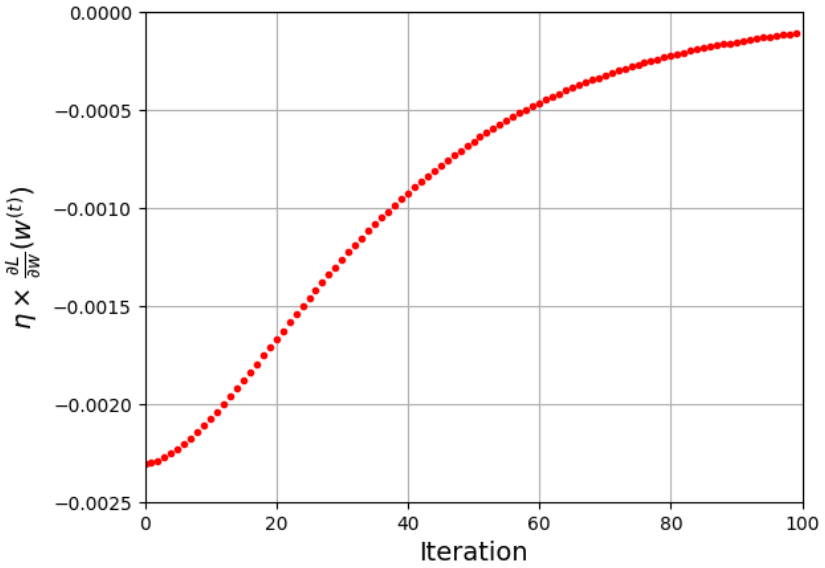
$$v^{(t+3)} = \beta^3 v^{(t)} - \beta^2\eta \frac{\partial L}{\partial w}(w^{(t)}) - \beta\eta \frac{\partial L}{\partial w}(w^{(t+1)}) - \eta \frac{\partial L}{\partial w}(w^{(t+2)})$$

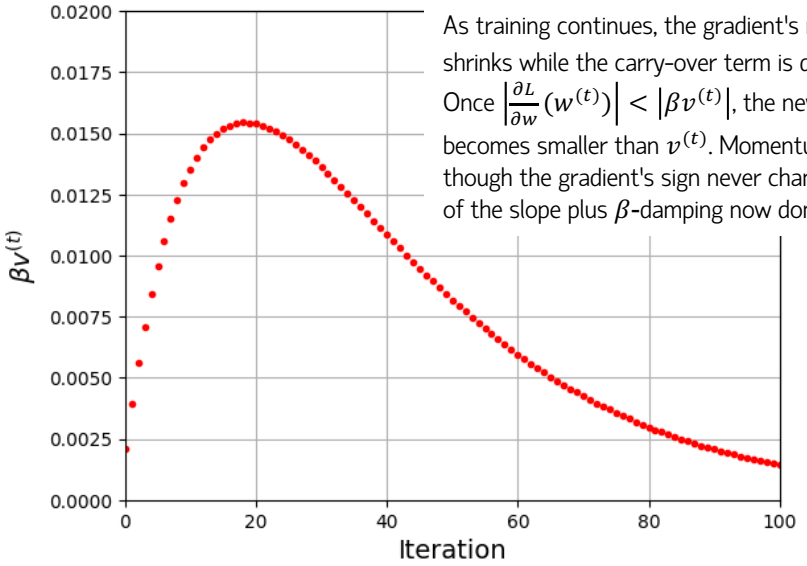
Each past gradient is multiplied by β once per step, so after k iterations its contribution is

$$\beta^k \eta \frac{\partial L}{\partial w}(w^{(t-k)})$$

That is, every time you advance one iteration the weight of *all* previous gradients is reduced by a factor β ,

In other words, a gradient that is 10 steps old still contributes about $\beta^{10} = 0.9^{10} \approx 0.35$ of its original weight; after roughly 20 steps it has shrunk to $\beta^{20} = 0.9^{20} \approx 0.12$, and by 30 steps it is down near 4%. So practically you can think of momentum with $\beta = 0.9$ as "remembering" on the order of 10–20 past updates, with older gradients fading exponentially fast.

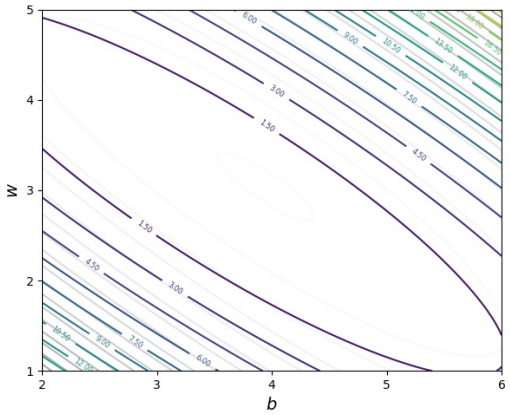




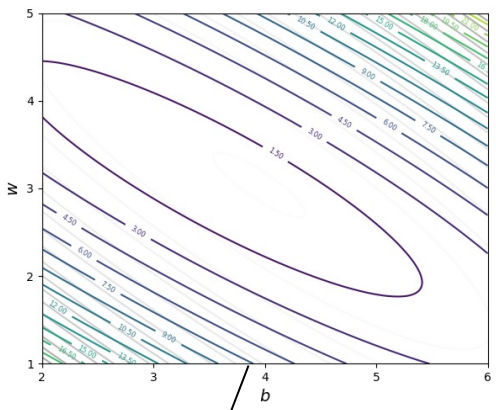
As training continues, the gradient's magnitude $\left|\frac{\partial L}{\partial w}(w^{(t)})\right|$ shrinks while the carry-over term is damped by β each step. Once $\left|\frac{\partial L}{\partial w}(w^{(t)})\right| < |\beta v^{(t)}|$, the new velocity $v^{(t+1)}$ becomes smaller than $v^{(t)}$. Momentum then decays—even though the gradient's sign never changes—because flattening of the slope plus β-damping now dominates.

Recall: SGD Loss Contours

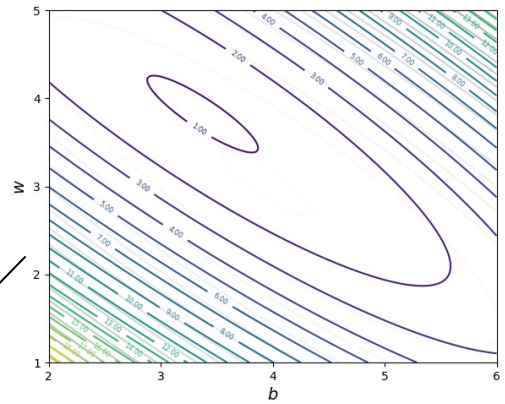
1st Mini-Batch Loss Surface

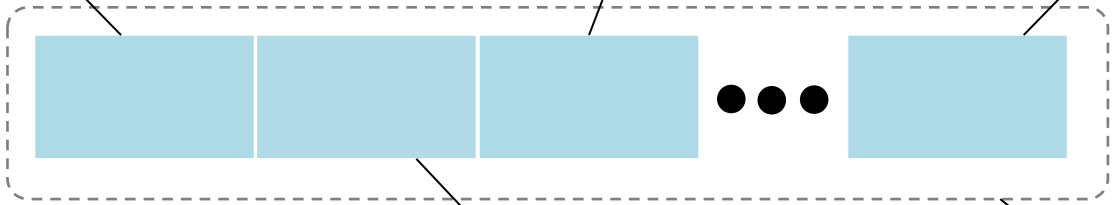


3rd Mini-Batch Loss Surface

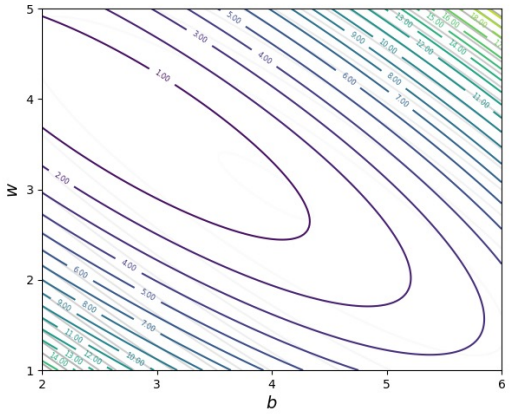


Mth Mini-Batch Loss Surface

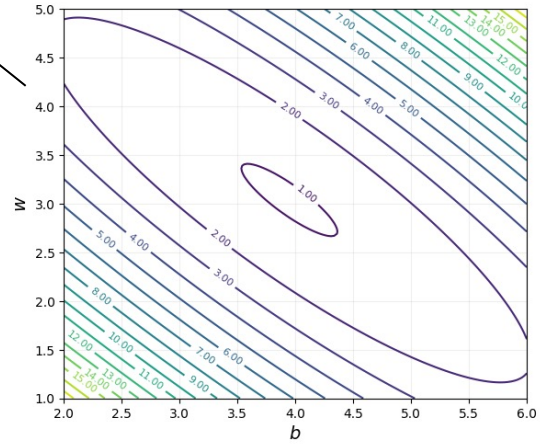




2nd Mini-Batch Loss Surface



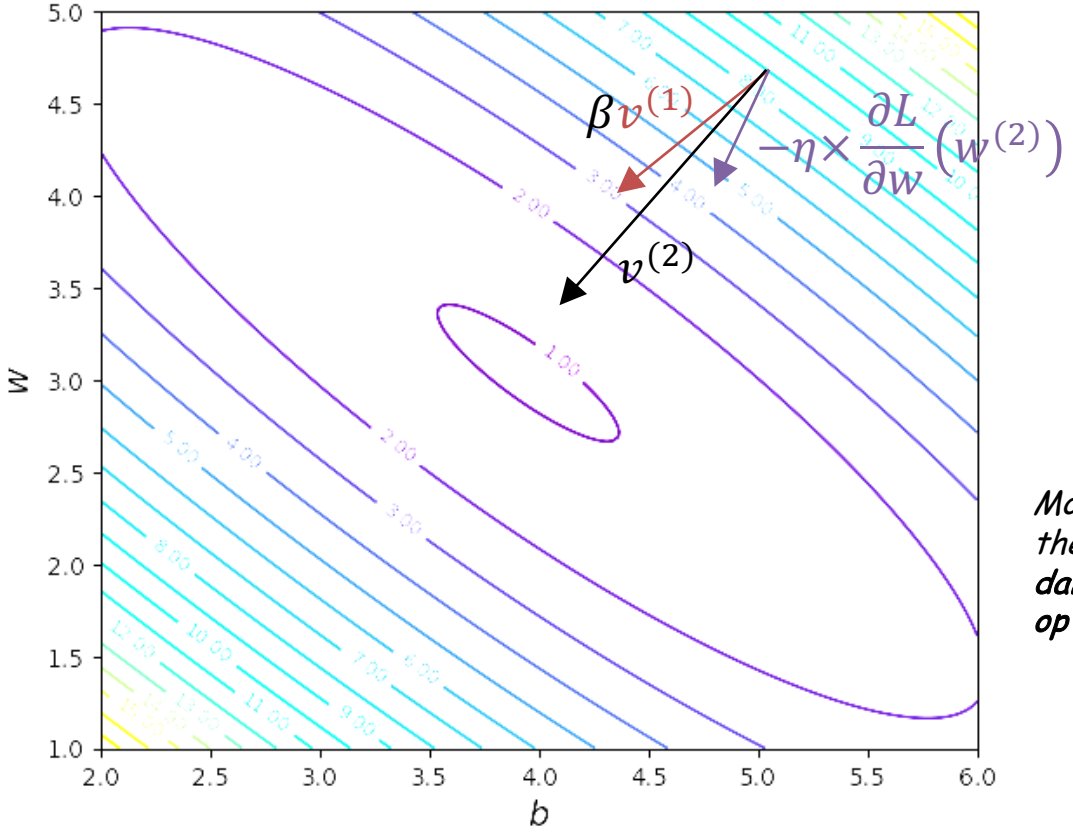
Full Training Loss



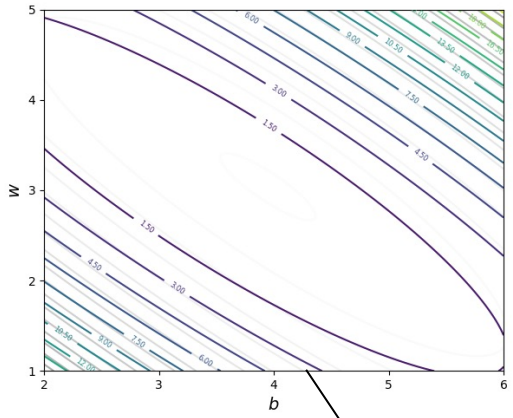
Each batch contains fraction of the data, so its contours wobble around the full-data ellipse, yet the overall shape still points toward the global minimum.

Momentum Smooths Mini-Batch Loss Contours

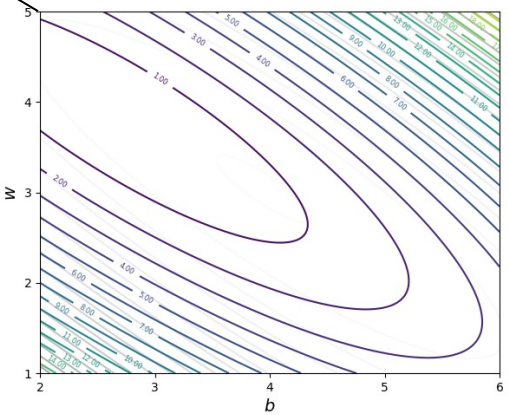
Full Training Loss



1st Mini-Batch Loss Surface

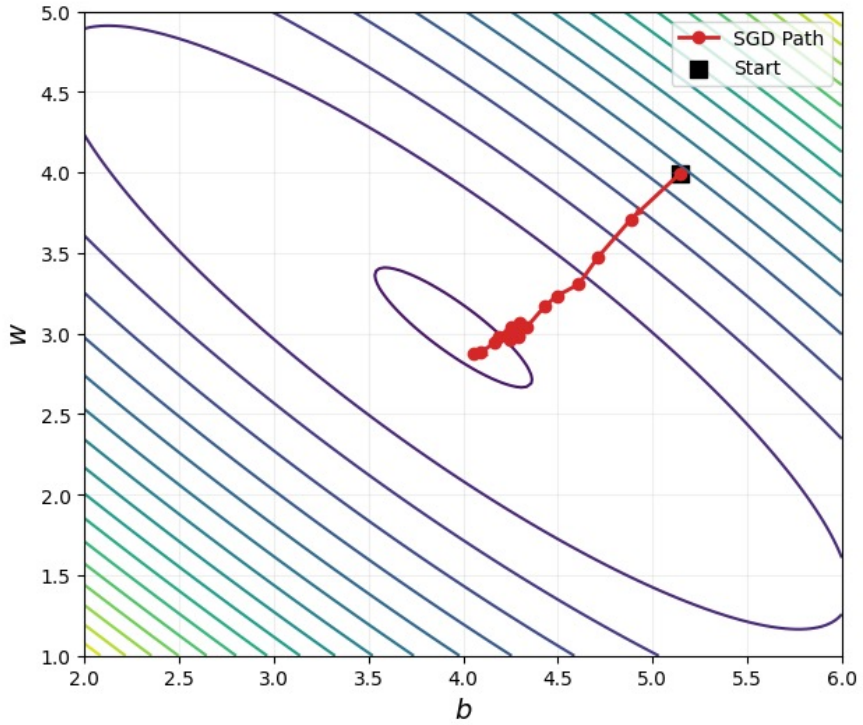


2nd Mini-Batch Loss Surface

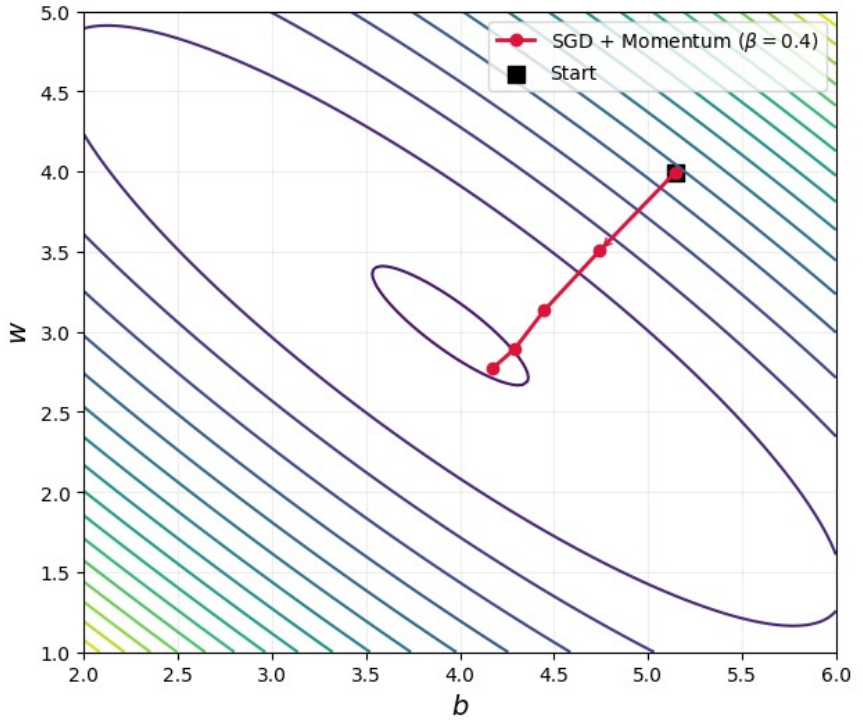


Momentum treats gradients like a rolling average: the $\beta v^{(t)}$ term ***amplifies correlated gradients and damps directional conflicts, yielding a smoother optimization trajectory.***

Batch-Noise Cancellation via Momentum



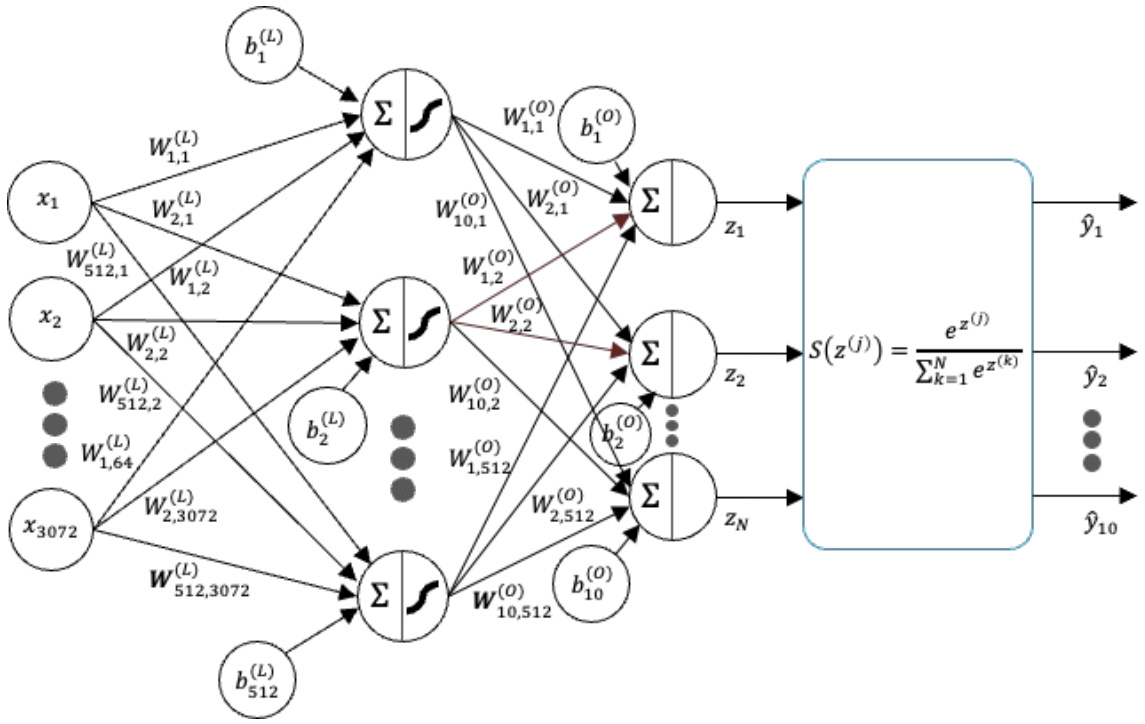
+ Momentum $\left(\beta v^{(t)}\right)$



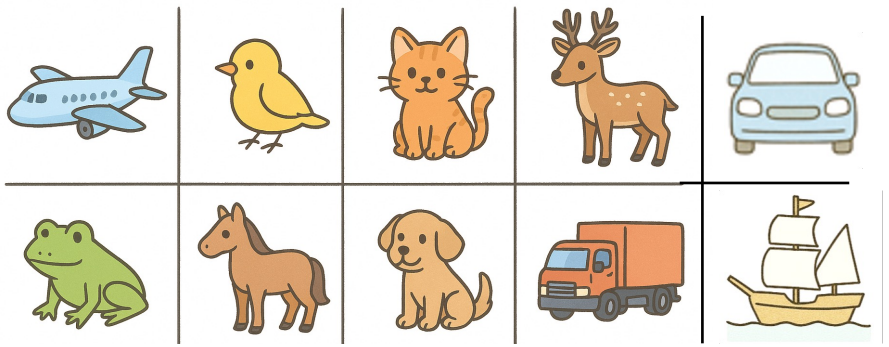
Each batch's gradient points a slightly different way, so the weight vector keeps oscillating across the valley walls.

Accumulated velocity keeps the update pointing in a consistent direction → longer, straighter trajectories toward the minimum.

Tensorflow: SGD with Momentum

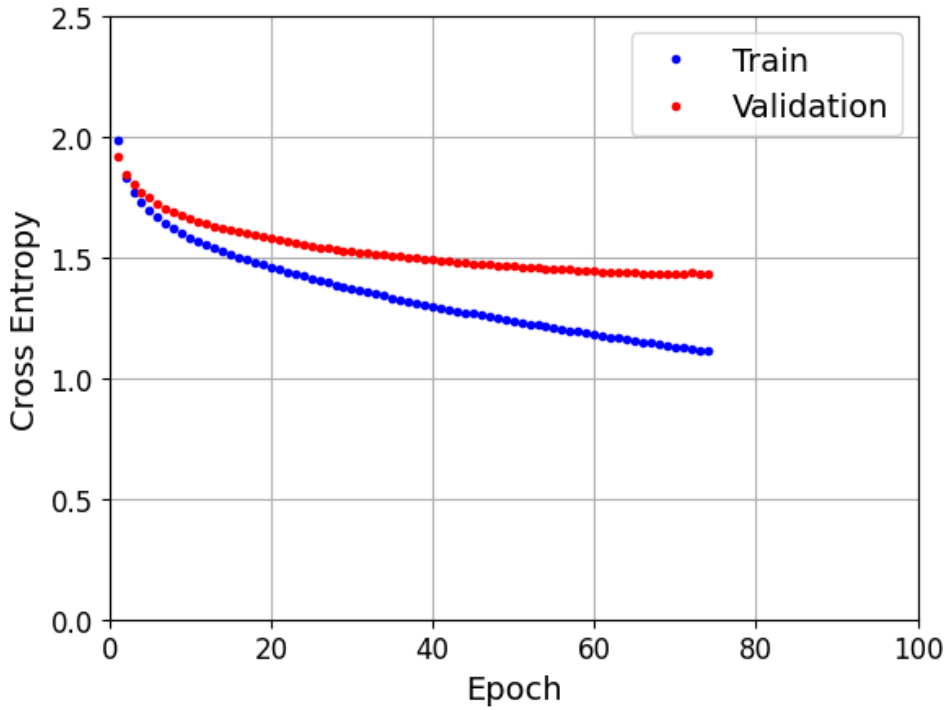


CIFAR-10 Dataset



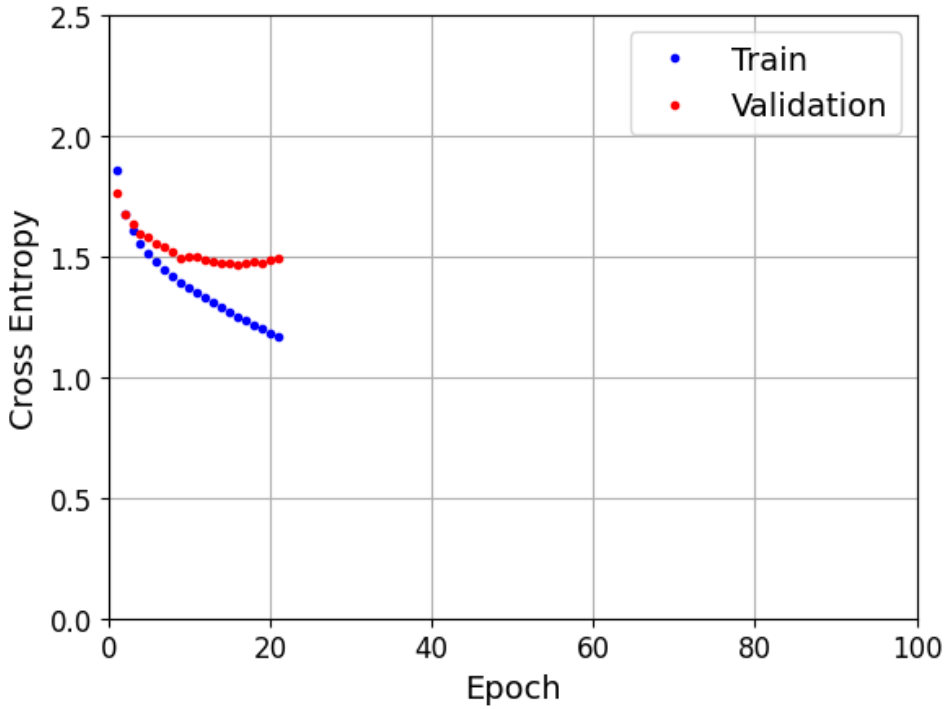
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import SGD

num_inputs = 3072         # input dimension
num_hidden_units = 512    # hidden units
num_outputs = 10          # output dimension

model = Sequential([
    Dense(num_hidden_units,
          activation=”relu",
          input_shape=(num_inputs,)),

    Dense(num_outputs,
          activation="softmax”)
])

model.compile(optimizer=SGD(learning_rate=1e-2, momentum=0.9),
              loss=”categorical_crossentropy",
              metrics=[“accuracy”])

batch_size = 128        # batch size
num_epochs = 100        # number of epochs

early_stop = EarlyStopping(monitor=‘val_loss’,
                           restore_best_weights=True, patience=5)

history = model.fit(X_train, y_train,
                    batch_size=batch_size,
                    epochs=num_epochs,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stop])
```

Effect of Momentum on Training Speed and Loss



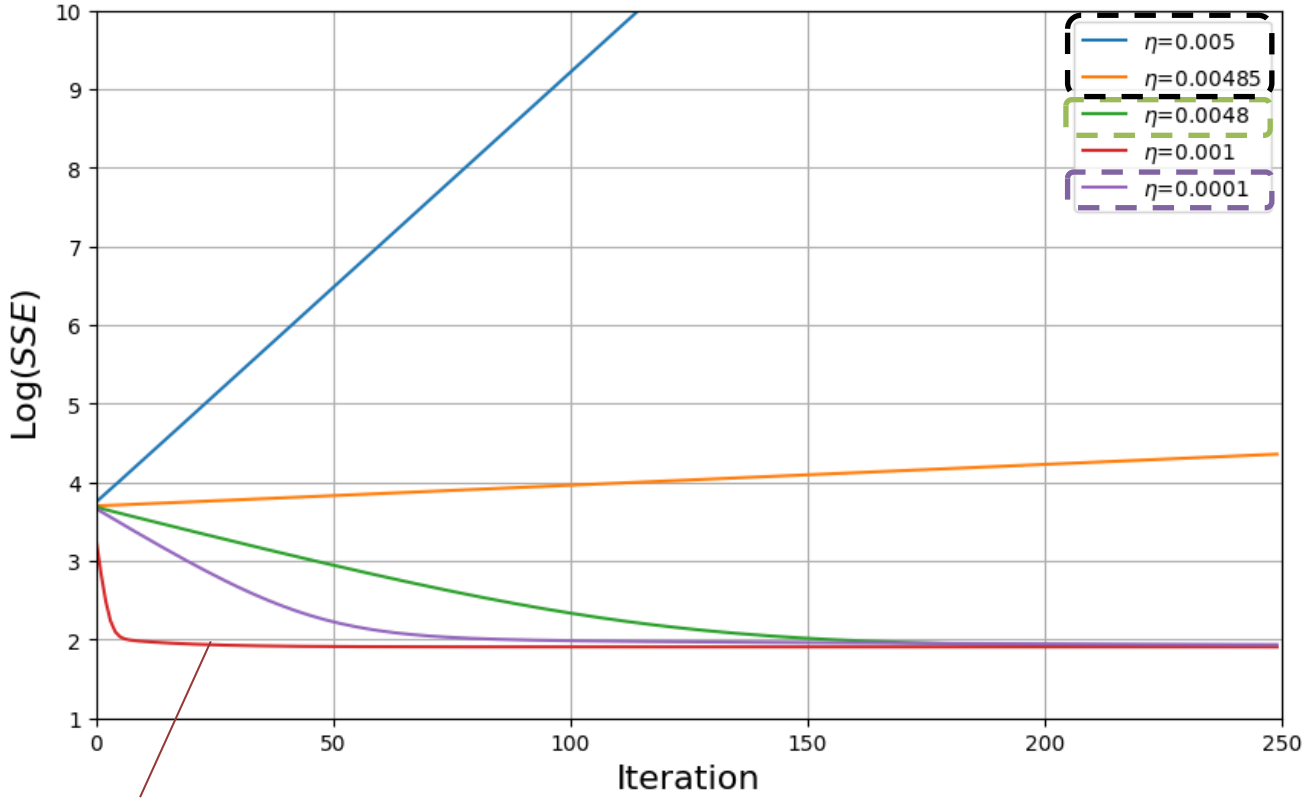
- Batch Size: 128
- Wall-Clock Time 97 Seconds
- Validation Loss: 1.438
- Validation Accurcy: 0.504

Momentum
$(\beta = 0.9)$



- Batch Size: 128
- Wall-Clock Time 31.4 Seconds
- Validation Loss: 1.494
- Validation Accurcy: 0.496

Learning Rate



...just right...

- Low learning rate leads to slow convergence, which will require iterations.
- High learning rate can also lead to slow convergence, which is caused by oscillations.
- Very high learning rate will cause gradient descent not to converge.

The learning rate η is yet another hyperparameter we must tune.

Q: Can we make η adapt automatically? — Ideally, it should start large and decay to smaller values as training progresses.

AdaGrad – Accumulate, Divide, Adapt, Gradually

In gradient descent, we choose a single, constant learning-rate η and apply that same step size to every weight update throughout training.

Gradient Descent:

$$w_i^{(t+1)} = w_i^{(t)} - \eta \times \frac{\partial L}{\partial w_i}(\vec{w}^{(t)})$$

Because $\left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2 \geq 0$, the AdaGrad accumulator s_i is monotonically increasing.

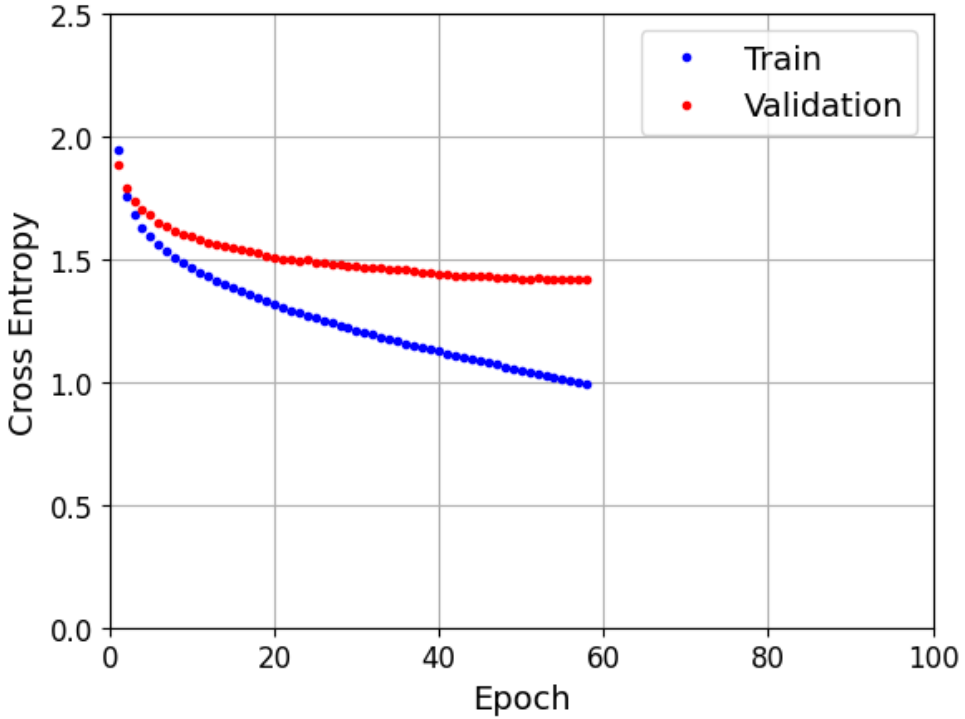
AdaGrad Algorithm:

$$s_i^{(t+1)} = s_i^{(t)} + \left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)} + \varepsilon}} \times \frac{\partial L}{\partial w_i}(\vec{w}^{(t)})$$

AdaGrad assigns its own learning-rate to every weight and automatically decays each one as its accumulator of past gradients grows during training.

Tensorflow: AdaGrad

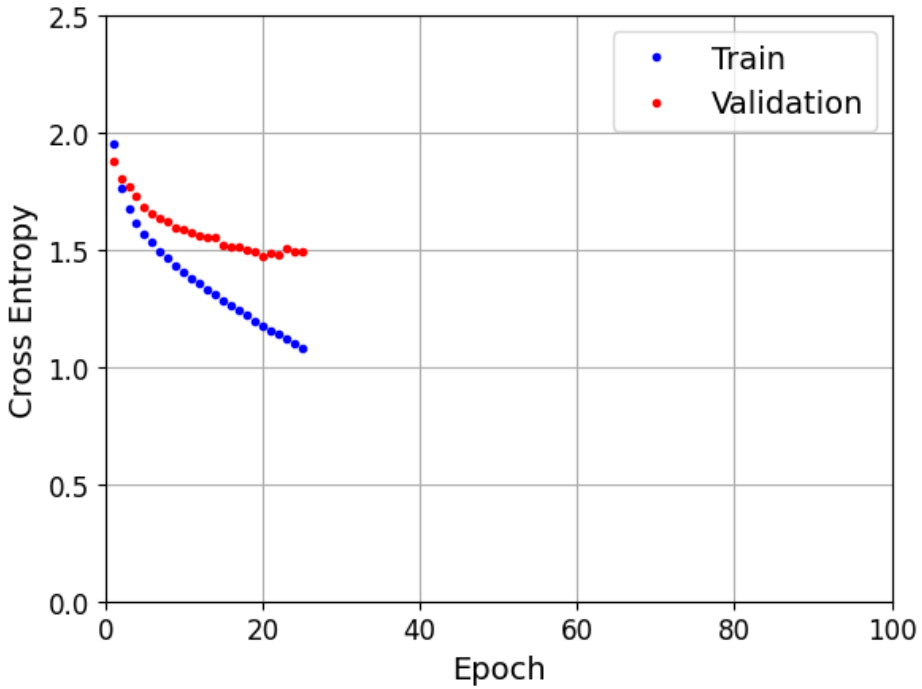


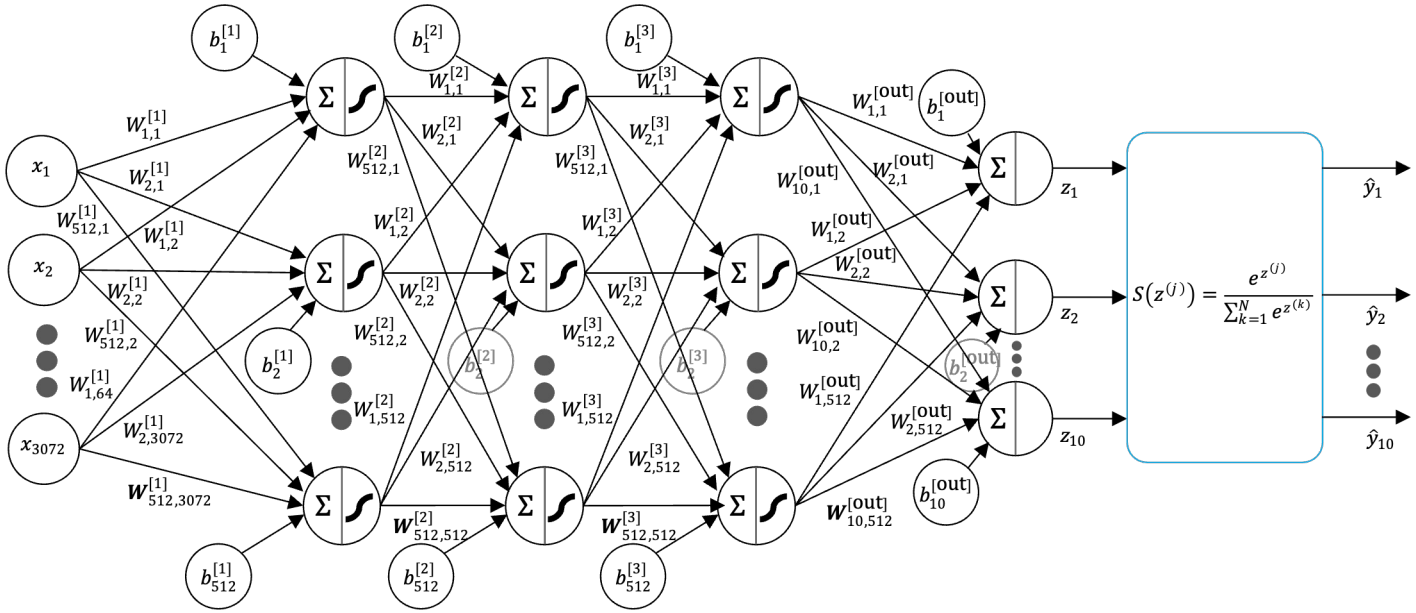
- Batch Size: 128
- Epoch: 53 (Restored Best Weights)
- Wall-Clock Time 84 Seconds (on Google Colab T4 GPU Setting)
- Validation Loss: 1.42
- Validation Accurcy: 0.51

Recall: SGD converged at 74 epochs and achieved validation loss of 1.438.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adagrad

num_inputs = 3072         # input dimension
num_hidden_units = 512    # hidden units
num_outputs = 10          # output dimension

model = Sequential([
    Dense(num_hidden_units,
          activation=”relu",
          input_shape=(num_inputs,)),

    Dense(num_outputs,
          activation="softmax”)
])

model.compile(optimizer=Adagrad(learning_rate=1e-2),
              loss=”categorical_crossentropy",
              metrics=[“accuracy”])

batch_size = 128        # batch size
num_epochs = 100        # number of epochs

early_stop = EarlyStopping(monitor=‘val_loss’,
                           restore_best_weights=True, patience=5)

history = model.fit(X_train, y_train,
                    batch_size=batch_size,
                    epochs=num_epochs,
                    validation_data=(X_val, y_val),
                   callbacks=[early_stop])
```

AdaGrad on 3-Hidden-Layer Neural Network





- Hidden Layers: 3
- Learning Rate: 10^{-2}
- Epoch: 21 (Best Val Loss)
- Wall-Clock Time 47 Seconds (on Google Colab T4 GPU Setting)
- Validation Loss: 1.49
- Validation Accurcy: 0.48

AdaGrad's accumulator grows monotonically, so per-weight learning-rates shrink continuously. In deep networks the earliest layers can end up with $\eta_{\text{eff}} \to 0$, freezing progress and flattening the loss curve.

The shallow net (1 hidden layer) achieved validation loss of 1.420.

Why AdaGrad Stalls

Source	What Reported	Why AdaGrad Stalls
Wilson et al., 2017 – "The Marginal Value of Adaptive Gradient Methods"	On CIFAR-10/100 and ImageNet, AdaGrad (and Adam/RMSProp) makes the largest initial drop in training loss, yet its validation accuracy plateaus early; momentum-SGD eventually surpasses it.	The cumulative accumulator $s_i^{(t)}$ grows without bound, so each coordinate's step size $\frac{\eta}{\sqrt{s_i^{(t)}+\varepsilon}}$ shrinks to the point that later updates are numerically negligible.
Reddi et al., 2018 – "Adaptive Methods for Non-Convex Optimization"	Prove that AdaGrad's unbounded $s_i^{(t)}$ can drive per-coordinate learning-rates below machine precision, blocking convergence on some convex and non-convex problems unless $s_i^{(t)}$ is reset or modified.	Because $\left(\frac{\partial L}{\partial w_i}\right)^2 \geq 0$, $s_i^{(t)}$ only increases; eventually $\frac{\eta}{\sqrt{s_i^{(t)}+\varepsilon}} \to 0$, so weights stop moving.
Zambrano & Bowen, 2023 – CNN cancer-cell study	On a histopathology CNN, AdaGrad lowers loss fastest for ≈ 10 epochs, then flattens; SGD and Adam continue improving and finish with higher validation accuracy.	Same mechanism: the growing denominator dominates after the early burst, throttling later updates.
Goodfellow, Bengio & Courville, Deep Learning (2016, §8.5)	Textbook note: "AdaGrad can stop learning too early on deep nets; later variants (RMSProp, Adam) replace the running sum with an exponential average so step sizes don't vanish."	RMSProp/Adam "forget" old gradients, keeping $s_i^{(t)}$ bounded and learning-rates alive; AdaGrad does not.

RMSProp

Because $\left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2 \geq 0$, the AdaGrad accumulator s_i is monotonically increasing. Consequently, the effective learning rate $\frac{\eta}{\sqrt{s_i^{(t+1)}+\varepsilon}}$ decays over time.

AdaGrad Algorithm:

$$s_i^{(t+1)} = s_i^{(t)} + \left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)}+\varepsilon}} \times \frac{\partial L}{\partial w_i}(\vec{w}^{(t)})$$

β	Approx. Memory	Typical Use
0.5	≈ 2 samples	Rapid-Reacting Control Loops
0.9	≈ 10	RMSProp Default (Variance Over ~10 Mini-Batches)
0.99	≈ 100	Adam's 2nd-Moment Decay for Large-Batch Training

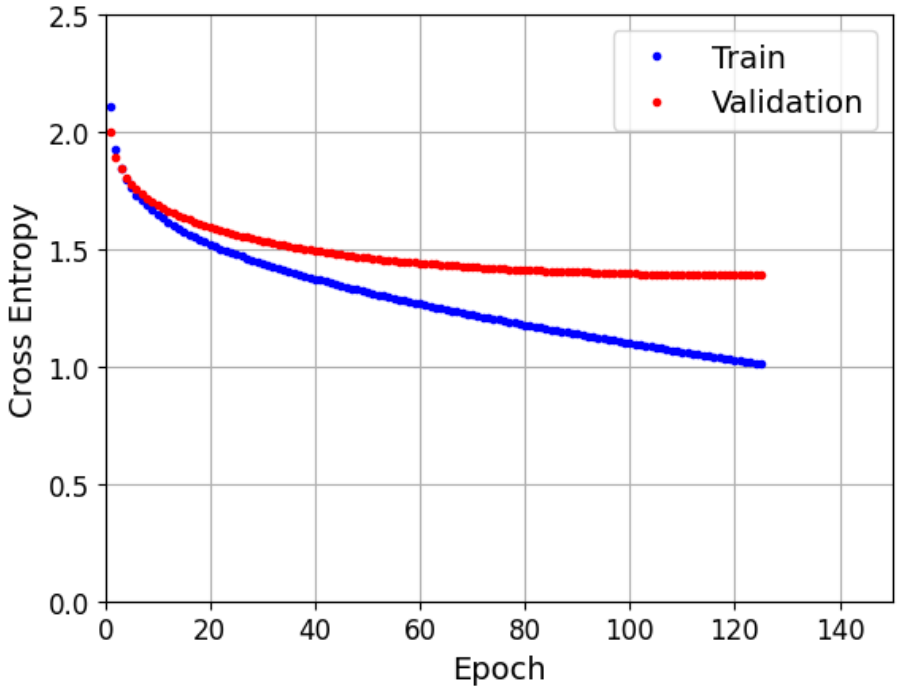
The RMSProp fixes this by accumulating $\left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2$ only from the most recent iterations. The role of β, commonly ≈0.9, is to carry memory forwards, while $(1-\beta)$ is to inject new information and renormalize the series.

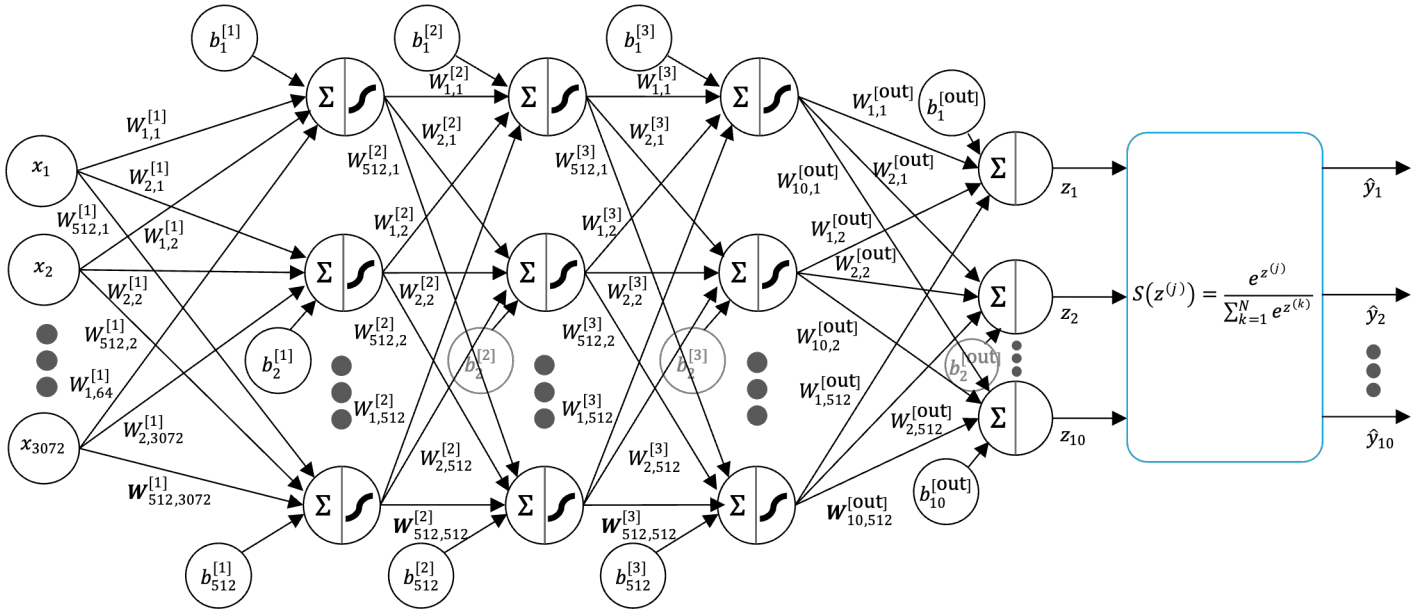
RMSProp Algorithm:

$$s_i^{(t+1)} = \beta \times s_i^{(t)} + (1-\beta) \times \left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)}+\varepsilon}} \times \frac{\partial L}{\partial w_i}(\vec{w}^{(t)})$$

RMSProp on 3-Hidden-Layer Neural Network (Revisit)






```
model.compile(optimizer=RMSprop(learning_rate=1e-5),
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

TensorFlow

- Hidden Layers: 3
- Learning Rate: 10^{-5}
- Epoch: 120 (Best Val Loss)
- Training Time: 203 Seconds (on Google Colab T4 GPU Setting)
- Validation Loss: 1.39
- Validation Accurcy: 0.52

AdaGrad ($\eta = 10^{-5}$): 2.02 Validation Loss at 500-th Epoch (Max Iter.)

With AdaGrad, and a very small base rate (10^{-5}), the sum of past-squared gradients keeps increasing, so the effective step $\eta / \sqrt{s_i^{(t)} + \varepsilon}$ steadily shrinks throughout training. After many steps, the deeper layers receive steps so tiny that their weights barely change.

Adam – Adaptive Moment Estimation

RMSProp Algorithm:

$$s_i^{(t+1)} = \beta \times s_i^{(t)} + (1-\beta) \times \left(\frac{\partial L}{\partial w_i} (\vec{w}^{(t)}) \right)^2$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)} + \varepsilon}} \times \frac{\partial L}{\partial w_i} (\vec{w}^{(t)})$$

+

Momentum Algorithm:

$$v^{(t+1)} = \beta \times v^{(t)} - \eta \times \frac{\partial L}{\partial w_i} (\vec{w}^{(t)})$$

$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

=

Adam Algorithm:

$$v^{(t+1)} = \beta_1 \times v^{(t)} - (1-\beta_1) \times \frac{\partial L}{\partial w_i} (\vec{w}^{(t)})$$

$$s_i^{(t+1)} = \beta_2 \times s_i^{(t)} + (1-\beta_2) \times \left(\frac{\partial L}{\partial w_i} (\vec{w}^{(t)}) \right)^2$$

$$\hat{v}_i^{(t+1)} = \frac{v_i^{(t+1)}}{1-\beta_1^t}$$

$$\hat{s}_i^{(t+1)} = \frac{s_i^{(t+1)}}{1-\beta_2^t}$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)} + \varepsilon}} \times \hat{v}_i^{(t+1)}$$

Per-Weight Learning-Rate (β_2 Branch)

Adam keeps an exponential moving average of each weight's squared gradients $\left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2$. $\beta_2 \approx 0.999$ carries the past variance forward (long memory). $(1-\beta_2)$ injects the newest squared-gradient and keeps the weights of all terms normalised to 1.

$$s_i^{(t+1)} = \beta_2 \times s_i^{(t)} + (1-\beta_2) \times \left(\frac{\partial L}{\partial w_i}(\vec{w}^{(t)})\right)^2$$

$$\hat{s}_i^{(t+1)} = \frac{s_i^{(t+1)}}{1-\beta_2^t}$$

Why divide by $1-\beta_2^t$? Scales the variance estimate *up* in the first few steps (because the EMA started at 0); as t grows, the factor $\rightarrow 1$, so the scale smoothly settles to its true value.

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)} + \varepsilon}} \times \hat{v}_i^{(t+1)}$$

Adam assigns its own learning-rate to every weight and automatically decays each one as its accumulator of past gradients grows during training.

Momentum (β_1 branch)

The velocity EMA ($\beta_1 \approx 0.9$) term averages ~10 recent gradients, smoothing mini-batch noise and giving the weights extra push along consistent directions.

$$v^{(t+1)} = \beta_1 \times v^{(t)} - (1-\beta_1) \times \frac{\partial L}{\partial w_i}(\vec{w}^{(t)})$$

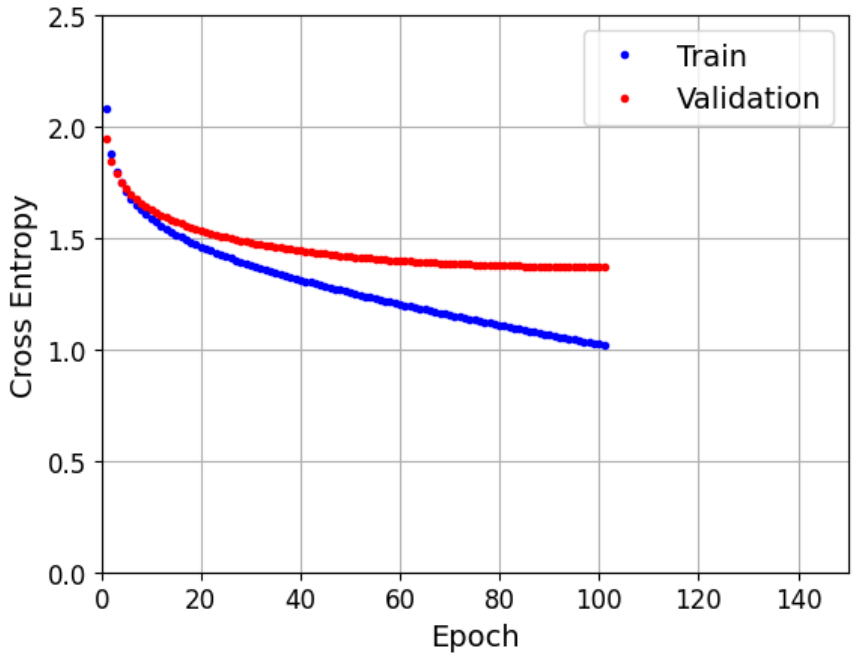
$$\hat{v}_i^{(t+1)} = \frac{v_i^{(t+1)}}{1-\beta_1^t}$$

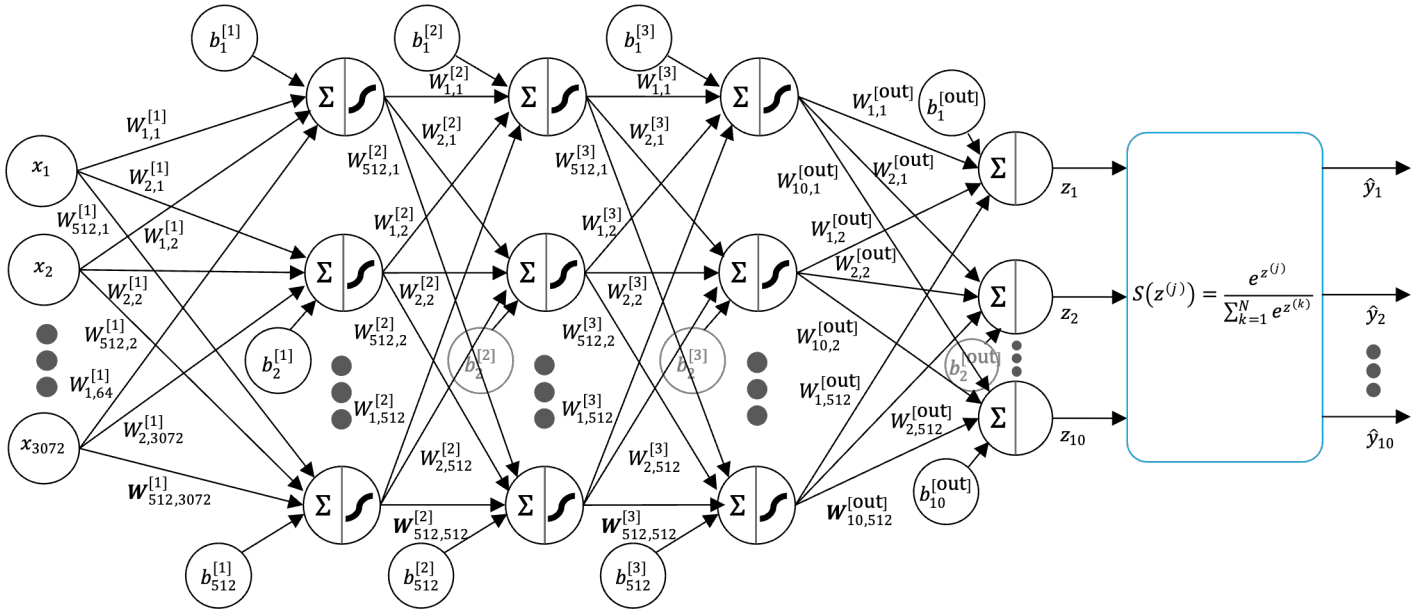
The bias-correction term scales the momentum *up* in the first few steps (the EMA started at 0); as t grows the factor $\to 1$, so the scale settles to its true mean.

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{s_i^{(t+1)} + \varepsilon}} \times \hat{v}_i^{(t+1)}$$

The numerator $\hat{v}_i^{(t+1)}$ provides direction & acceleration; the denominator $\sqrt{s_i^{(t+1)} + \varepsilon}$ (from the β_2 branch) sets each weight's effective learning-rate, keeping steps in range.

Adam on 3-Hidden-Layer Neural Network





- Hidden Layers: 3
- Learning Rate: 10^{-5}
- Epoch: 96 (Best Val Loss)
- Training Time: 166 Seconds (on Google Colab T4 GPU Setting)
- Validation Loss: 1.37
- Validation Accurcy: 0.53

RMSProp converged at 120 epochs and achieved validation loss of 1.37.

```
model.compile(optimizer=Adam(learning_rate=1e-5),
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

TensorFlow

Adam's accumulated-gradient term $v^{(t)}$ lets it accelerate along a stable descent direction. RMSProp lacks this feature, so Adam achieved the same validation loss in fewer epochs.

Optimisers: Speed and Stability

Optimiser	Small / Shallow Datasets	Large / Deep Models
SGD	Starts slowly and needs a well-chosen learning-rate schedule to avoid stalling. Once tuned, it learns clean, often flatter solutions that generalise well.	Scales to big data with modest memory, but still lags others in early epochs; step-decay or cosine LR is almost mandatory. Stable when tuned, but high LR can explode training.
SGD + Momentum	Momentum (≈0.9) accelerates progress and damps zig-zags, so convergence is noticeably faster than vanilla SGD. Usually reaches the best test accuracy on small sets after a few LR drops.	The workhorse of vision: with cosine or step decay it approaches adaptive optimisers' speed but keeps SGD's strong generalisation. Overshoot is rare thanks to momentum's damping.
Adagrad	Per-feature step-size boosts rare features and gives lightning-fast first epochs, especially on sparse UCI data. Its ever-shrinking LR then causes an early plateau in accuracy.	Same rapid early drop on deep nets, but learning stalls when LR decays to near-zero; final accuracy often trails others. Safe and rarely diverges, yet hard to "un-freeze" later.
RMSProp	Maintains a running RMS of past gradients, so it adapts quickly without the aggressive decay that hobbles Adagrad. Stable on tricky RNN toy problems with minimal tuning.	Matches Adam's early loss reduction on CNNs and small transformers, and usually trains without oscillation. May need LR decay + ε tuning to squeeze out the very last percent.
Adam	Combines RMSProp + momentum, giving the fastest loss drop and tolerance to higher LR; perfect for quick prototyping. Can overfit tiny datasets unless you add weight-decay or early-stop.	Default choice for transformers: reaches target accuracy in the fewest steps and scales to huge batches. Extremely stable across wide LR ranges, though it may land in slightly sharper minima unless weight-decay (AdamW) is used.

Summary

- Mini-batch SGD visits the dataset in small, shuffled blocks and averages their gradients. This saves repeated passes over all samples yet still steers the weights to the same region of the loss surface that full-batch training would reach.
- Batch–to-batch randomness limits over-fitting. Because every mini-batch sees only part of the data, successive gradients differ slightly; those differences push the optimiser away from narrow, highly-curved minima and prefer broader basins that generalise better.
- Momentum (accumulated gradients) speeds convergence. A running average of recent gradients reinforces directions that agree and dampens directions that conflict, converting zig-zag steps into a smoother, longer step size toward the minimum and reducing the total number of epochs required.
- RMSProp-style scaling controls step size per weight. Dividing each update by the square-root of its own recent squared gradients lets infrequently changing weights move farther while highly variable weights move cautiously; AdaGrad's unbounded sum keeps growing and can shrink steps almost to zero.
- Adam combines the scaling with momentum. The variance term sets a safe, per-weight learning-rate; the momentum term adds directional acceleration. Together they allow Adam to reach the same validation loss in roughly two-thirds of the epochs that RMSProp, which lacks momentum, requires.