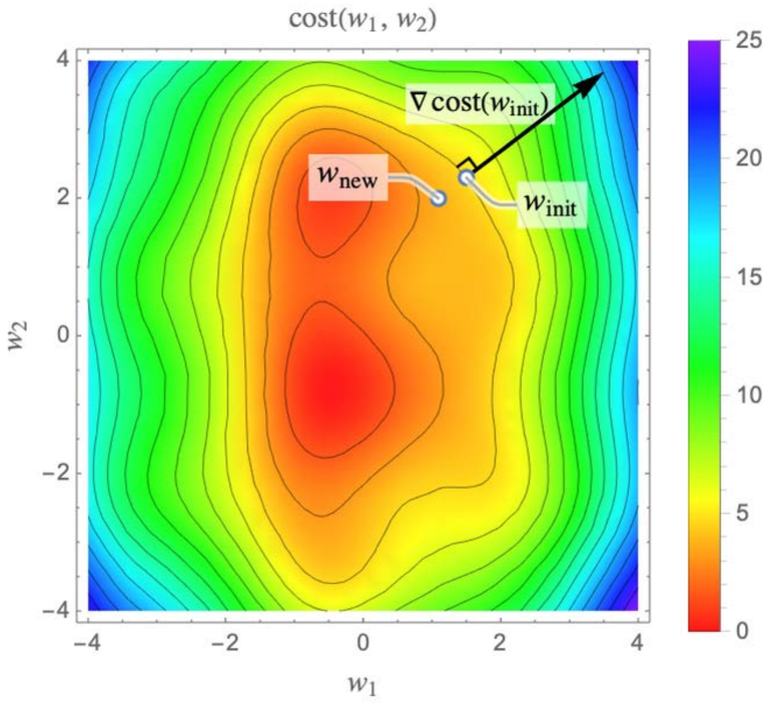
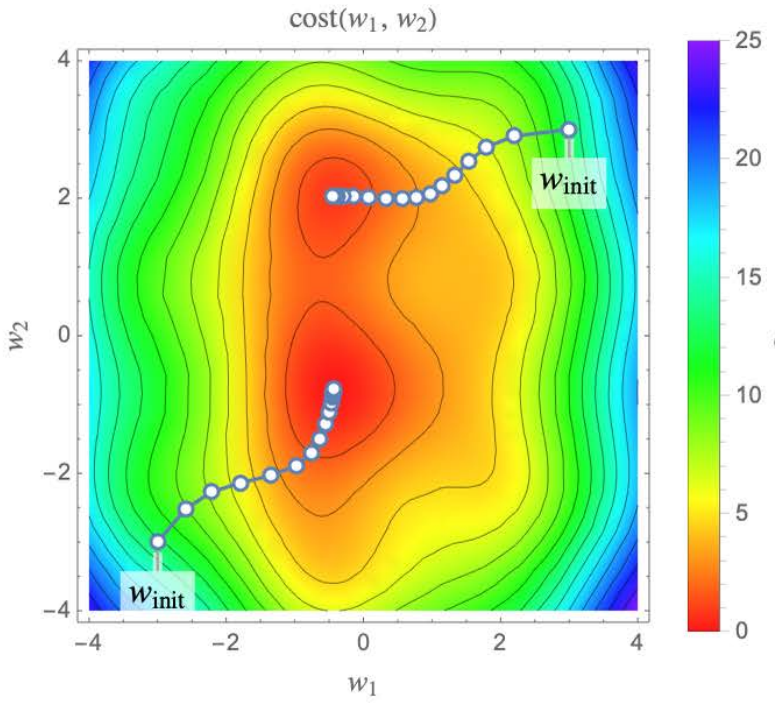
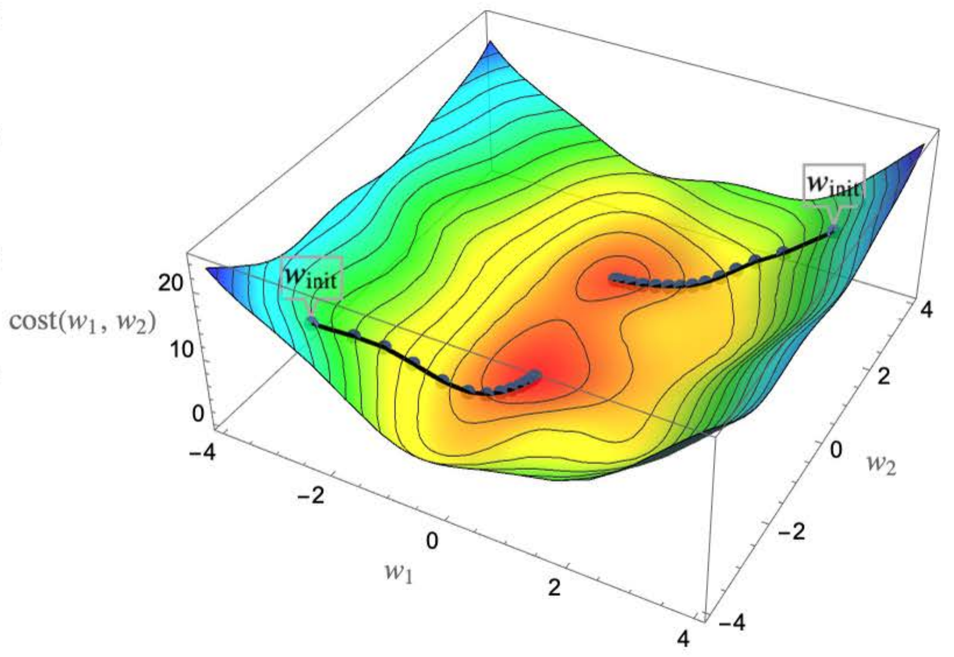
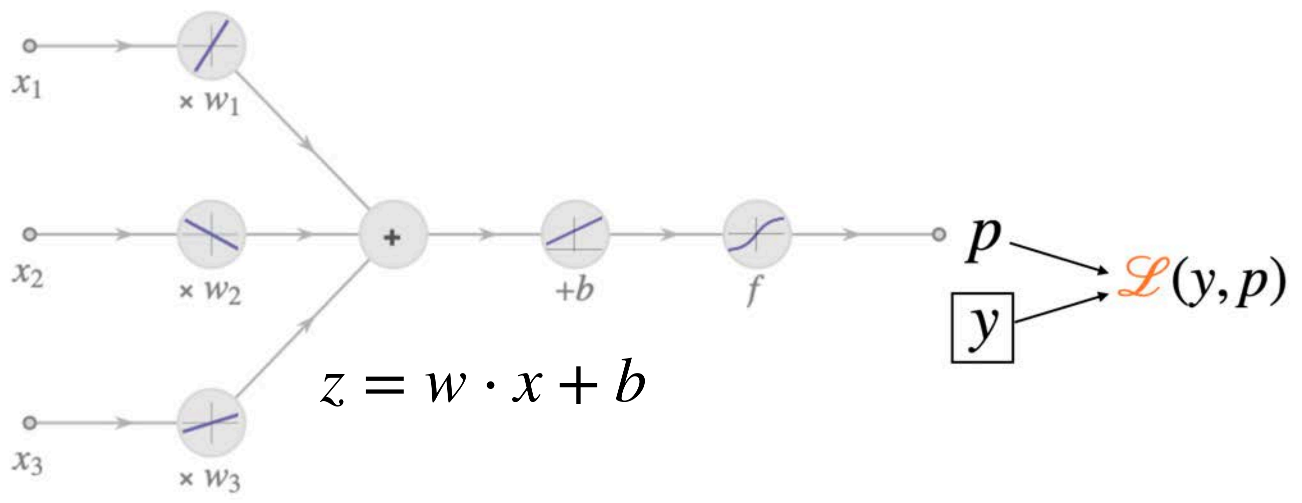
Gradient Descent: moving to minimize the loss









$$z = w \cdot x + b$$

$$\frac{\partial \mathscr{L}}{\partial w} = \frac{\partial \mathscr{L}}{\partial p} \cdot \frac{\partial p}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Backpropagation

backward engine driving forward learning

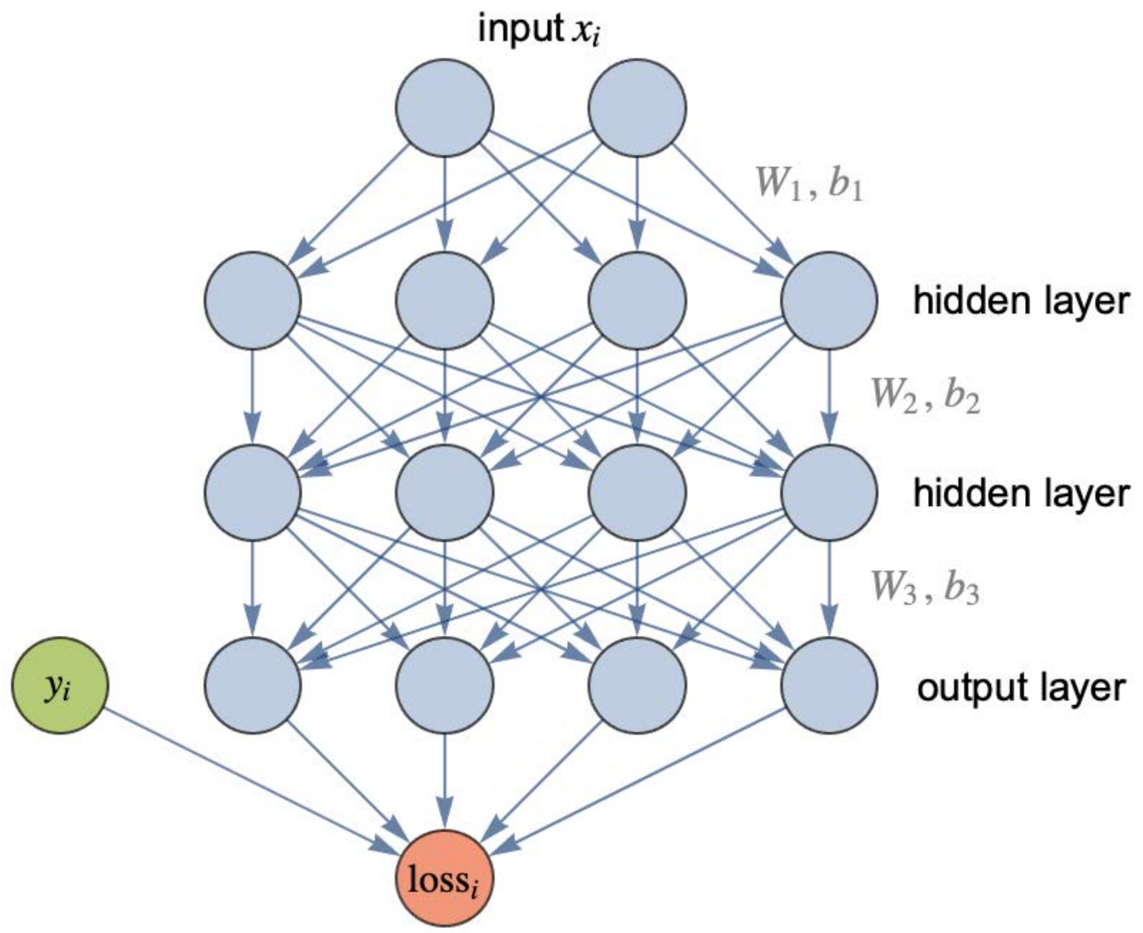
[Ilya Sutskever]: “How can it be that it’s so simple … that you can explain it to high school students without too much effort?”

[Hinton]: “I think that’s actually miraculous. This is also, to me, an indication that we are probably on the right track. [It can’t] be a coincidence that such simple concepts go so far.”

- ***Why machines learn*** *by A. Ananthaswamy*

Backpropagation

How to learn from mistakes (decision-making cycle)



Inference: make a decision (prediction outputs ‘p’ (launch product!)

→ Feedback: negative feedback (positive loss) signals something went wrong.

→ Assign blame! Trace the reporting chain backward, assign blame to each person (neuron) based on their weighted influence.

→ Weight update: To improve future decisions, give more credit to better contributor; less to who caused errors.

Repeat the cycle.

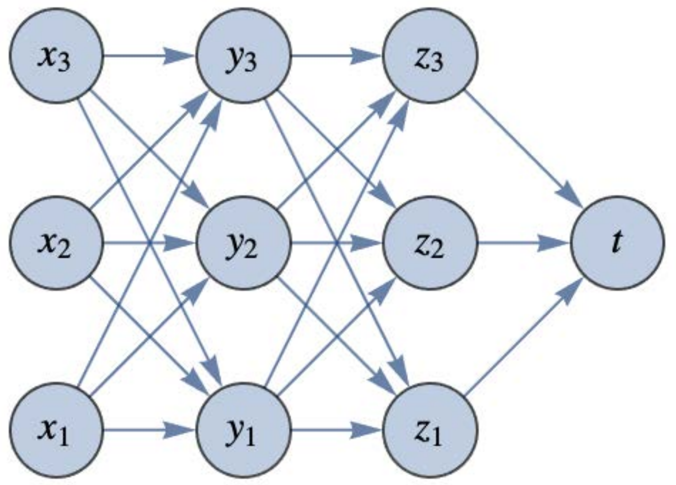
Backprop: Just a chain rule

let’s write this in a simpler way

$$\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} \xrightarrow{h} \begin{pmatrix}y_1\\y_2\\y_3\end{pmatrix} \xrightarrow{g} \begin{pmatrix}z_1\\z_2\\z_3\end{pmatrix} \xrightarrow{f} t$$

$$\text{gradient}(x_1, x_2, x_3) = \left\{\frac{\partial t}{\partial x_1}, \frac{\partial t}{\partial x_2}, \frac{\partial t}{\partial x_3}\right\}$$

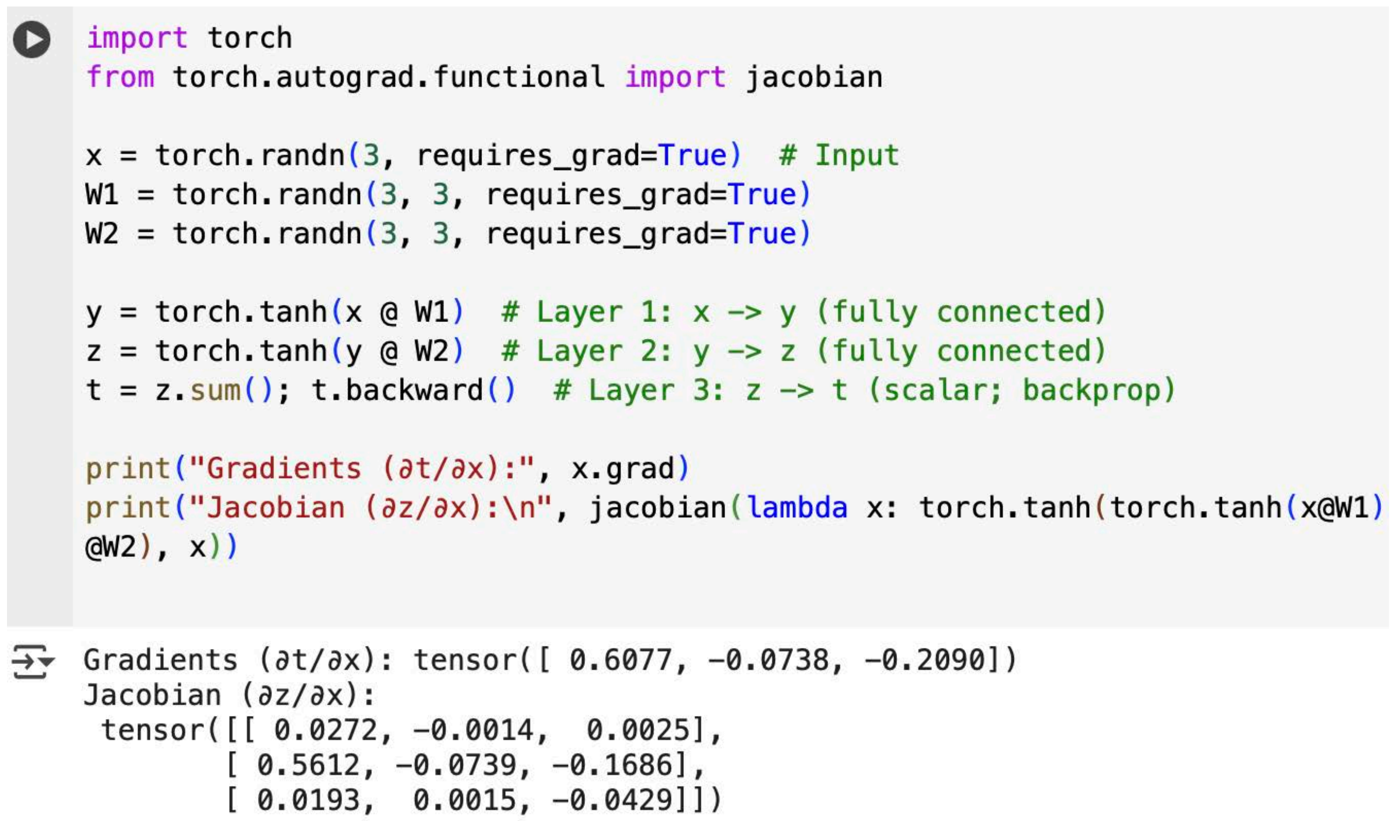
$$\text{gradient}(x_1, x_2, x_3) = \frac{\partial t}{\partial(x_1, x_2, x_3)}$$



$$\frac{\partial t}{\partial(z_1, z_2, z_3)} = \begin{pmatrix}\frac{\partial t}{\partial z_1}\\ \frac{\partial t}{\partial z_2}\\ \frac{\partial t}{\partial z_3}\end{pmatrix}$$

$$\frac{\partial(z_1, z_2, z_3)}{\partial(y_1, y_2, y_3)} = \begin{pmatrix}\frac{\partial z_1}{\partial y_1} & \frac{\partial z_1}{\partial y_2} & \frac{\partial z_1}{\partial y_3}\\ \frac{\partial z_2}{\partial y_1} & \frac{\partial z_2}{\partial y_2} & \frac{\partial z_2}{\partial y_3}\\ \frac{\partial z_3}{\partial y_1} & \frac{\partial z_3}{\partial y_2} & \frac{\partial z_3}{\partial y_3}\end{pmatrix}$$

Jacobian

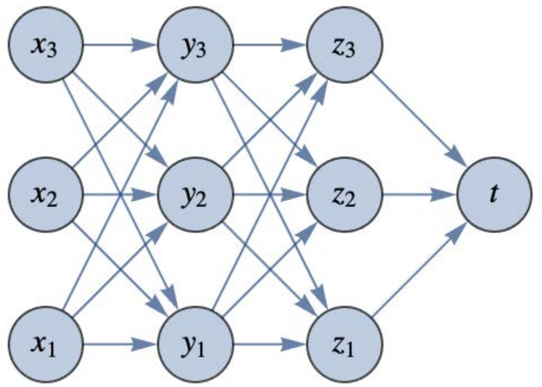


```
import torch
from torch.autograd.functional import jacobian

x = torch.randn(3, requires_grad=True)  # Input
W1 = torch.randn(3, 3, requires_grad=True)
W2 = torch.randn(3, 3, requires_grad=True)

y = torch.tanh(x @ W1)  # Layer 1: x -> y (fully connected)
z = torch.tanh(y @ W2)  # Layer 2: y -> z (fully connected)
t = z.sum(); t.backward()  # Layer 3: z -> t (scalar; backprop)

print("Gradients (∂t/∂x):", x.grad)
print("Jacobian (∂z/∂x):\n", jacobian(lambda x: torch.tanh(torch.tanh(x@W1)
@W2), x))
```

```
Gradients (∂t/∂x): tensor([ 0.6077, -0.0738, -0.2090])
Jacobian (∂z/∂x):
 tensor([[ 0.0272, -0.0014,  0.0025],
        [ 0.5612, -0.0739, -0.1686],
        [ 0.0193,  0.0015, -0.0429]])
```

Backpropagation

Why is it a better approach for DNNs?



$$\frac{\partial t}{\partial(x_1, x_2, x_3)} = \frac{\partial(y_1, y_2, y_3)}{\partial(x_1, x_2, x_3)} \cdot \frac{\partial(z_1, z_2, z_3)}{\partial(y_1, y_2, y_3)} \cdot \frac{\partial t}{\partial(z_1, z_2, z_3)}$$

$$\frac{\partial t}{\partial(x_1, x_2, x_3)} = \boxed{\begin{pmatrix} 0.63 & -0.78 & 0.58 \\ -0.62 & -0.52 & -0.87 \\ 0.084 & -0.54 & -0.21 \end{pmatrix} \cdot \begin{pmatrix} 0.4 & -0.58 & 0.5 \\ -0.15 & -0.51 & 0.95 \\ 0.65 & 0.85 & 0.16 \end{pmatrix}} \cdot \begin{pmatrix} -0.41 \\ -0.58 \\ 0.16 \end{pmatrix}$$

Forward: (matrix) ✖ (matrix)

Backward: (matrix) ✖ (vector)

$$\frac{\partial t}{\partial(x_1, x_2, x_3)} = \begin{pmatrix} 0.63 & -0.78 & 0.58 \\ -0.62 & -0.52 & -0.87 \\ 0.084 & -0.54 & -0.21 \end{pmatrix} \cdot \boxed{\begin{pmatrix} 0.4 & -0.58 & 0.5 \\ -0.15 & -0.51 & 0.95 \\ 0.65 & 0.85 & 0.16 \end{pmatrix} \cdot \begin{pmatrix} -0.41 \\ -0.58 \\ 0.16 \end{pmatrix}}$$

$$= \begin{pmatrix} 0.63 & -0.78 & 0.58 \\ -0.62 & -0.52 & -0.87 \\ 0.084 & -0.54 & -0.21 \end{pmatrix} \cdot \begin{pmatrix} 0.25 \\ 0.51 \\ -0.74 \end{pmatrix}$$

$$= \begin{pmatrix} -0.67 \\ 0.22 \\ -0.1 \end{pmatrix}$$

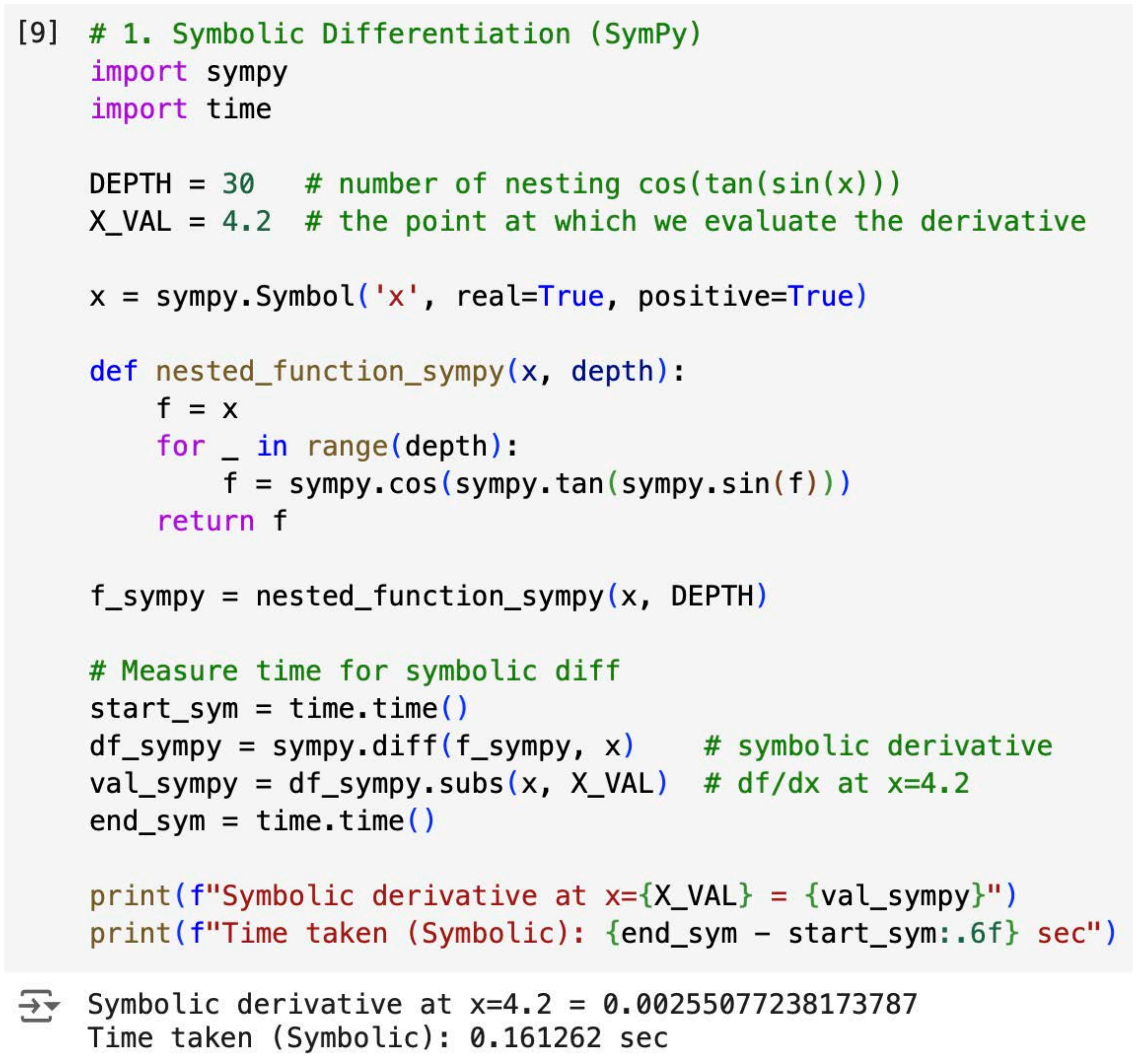
Going backward in DNN means multiplying a vector (Nx1) with its nearby matrix (say, NxN).

By always multiplying the vector first, we can avoid multiplying two matrices, which is computationally more expensive.

In neural nets, backpropagating the gradient is the optimal strategy, since the output of the loss function is a scalar (the loss value).

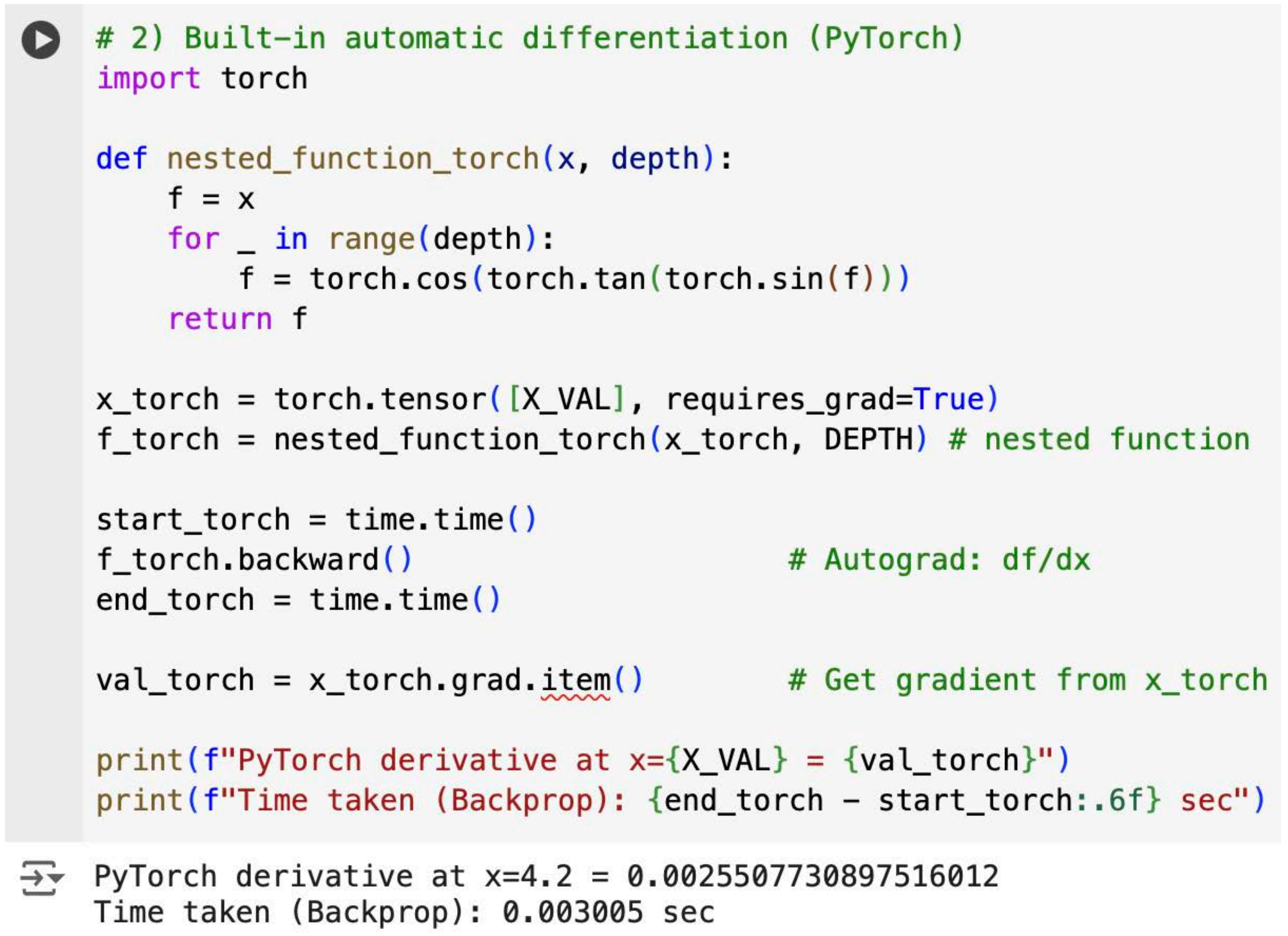
Efficiency of Backpropagation (and PyTorch)

Calculate df/dx for f(x) = cos(tan(sin(cos(tan(sin(cos(tan(sin……. cos(tan(sin(x)))) 30 nests



```
[9] # 1. Symbolic Differentiation (SymPy)
    import sympy
    import time

    DEPTH = 30   # number of nesting cos(tan(sin(x)))
    X_VAL = 4.2  # the point at which we evaluate the derivative

    x = sympy.Symbol('x', real=True, positive=True)

    def nested_function_sympy(x, depth):
        f = x
        for _ in range(depth):
            f = sympy.cos(sympy.tan(sympy.sin(f)))
        return f

    f_sympy = nested_function_sympy(x, DEPTH)

    # Measure time for symbolic diff
    start_sym = time.time()
    df_sympy = sympy.diff(f_sympy, x)    # symbolic derivative
    val_sympy = df_sympy.subs(x, X_VAL)  # df/dx at x=4.2
    end_sym = time.time()

    print(f"Symbolic derivative at x={X_VAL} = {val_sympy}")
    print(f"Time taken (Symbolic): {end_sym - start_sym:.6f} sec")
```

```
Symbolic derivative at x=4.2 = 0.00255077238173787
Time taken (Symbolic): 0.161262 sec
```



```
# 2) Built-in automatic differentiation (PyTorch)
import torch

def nested_function_torch(x, depth):
    f = x
    for _ in range(depth):
        f = torch.cos(torch.tan(torch.sin(f)))
    return f

x_torch = torch.tensor([X_VAL], requires_grad=True)
f_torch = nested_function_torch(x_torch, DEPTH) # nested function

start_torch = time.time()
f_torch.backward()                        # Autograd: df/dx
end_torch = time.time()

val_torch = x_torch.grad.item()           # Get gradient from x_torch

print(f"PyTorch derivative at x={X_VAL} = {val_torch}")
print(f"Time taken (Backprop): {end_torch - start_torch:.6f} sec")
```

```
PyTorch derivative at x=4.2 = 0.0025507730897516012
Time taken (Backprop): 0.003005 sec
```

Symbolic derivative: exact solution, but too slow as the network complexity increases.

Common issues in BP

Vanishing gradients

- Sigmoid and tanh activation functions can cause gradients to decrease exponentially in deeper layers.
- Weights in earlier layers update very slowly - poor learning.
- Solution: Use activation with non-saturating gradients (ReLU, leaky ReLU…)

Exploding gradients

- Gradients grow exponentially; unstable training process

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(l+1)}} \cdot \frac{\partial a^{(l+1)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

for sigmoid activation $a = \frac{1}{1+e^{-z}}$, what is $\frac{\partial a}{\partial z}$?

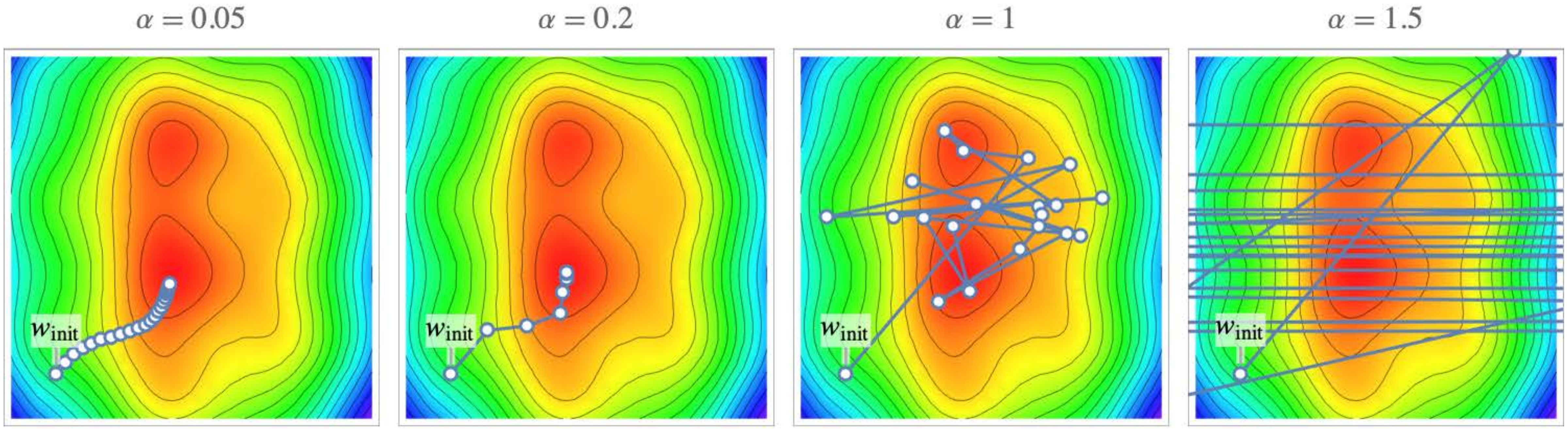
when does vanishing gradients occur?

Learning rate (lr)

$$w_{new} = w_{old} - \alpha \cdot \frac{\partial \mathscr{L}}{\partial w}$$

hyperparameter controlling the step size (toward neg gradient of L)

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
```

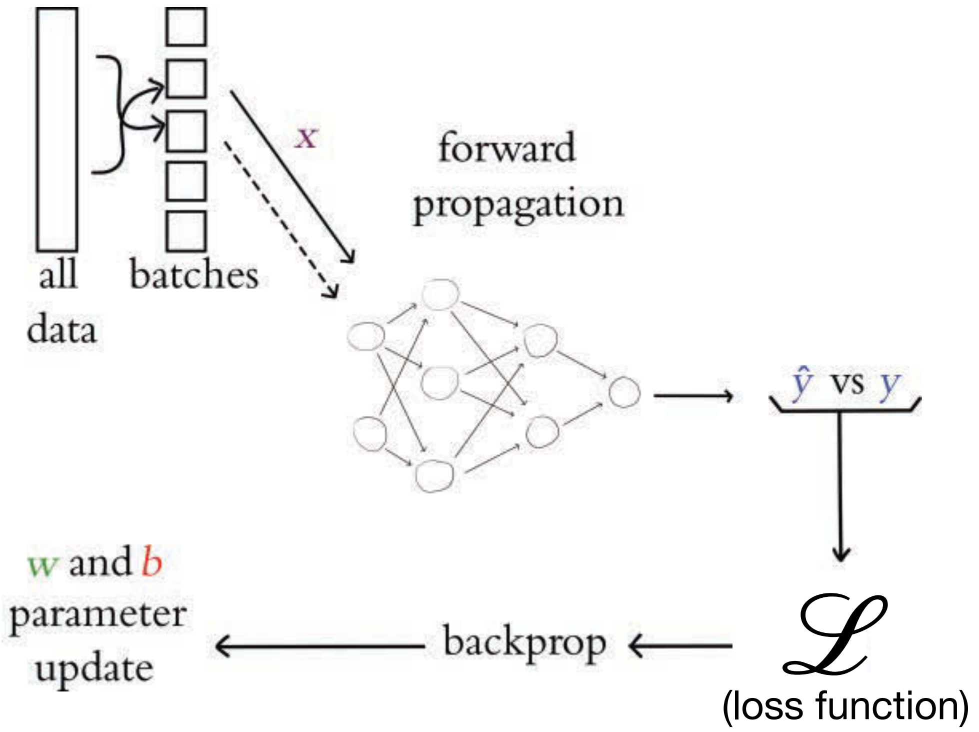


'lr' too small (slow) optimal large: overshoots minimum

Stochastic Gradient Descent (SGD)

Why simple gradient descent doesn't work in real world?

- Computing gradients over entire dataset is prohibitive for millions of examples
- Solution (SGD): Approximate gradients using small, random subsets (mini-batches).
- Faster convergence, can escape local minima due to noisy updates.
- Enables training on large datasets



```
import tensorflow as tf

(x_train, y_train), _ = tf.keras.datasets.mnist.load_data() #60000 images
x_train = x_train / 255.0

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='sparse_categorical_crossentropy')
model.fit(x_train, y_train, batch_size=128, epochs=2)
```

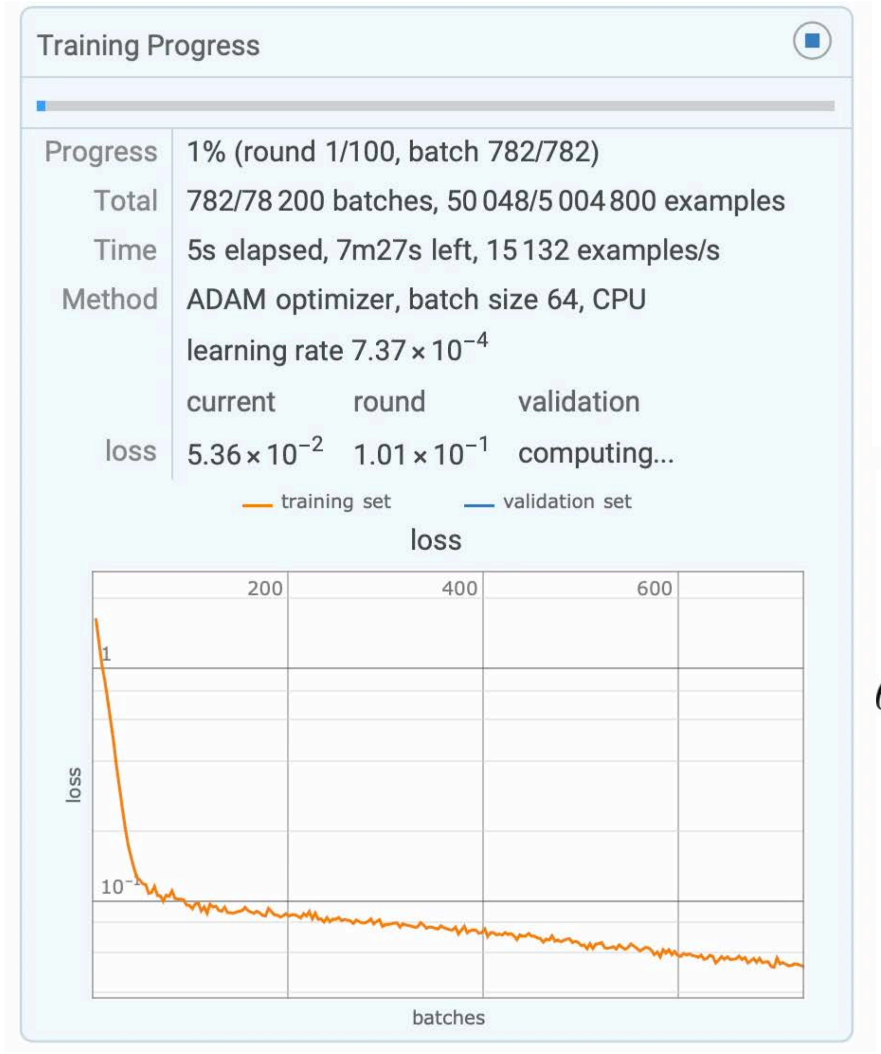
Stochastic Gradient Descent: learn with noise

SGD: ‘Sweet spot’ noise can make DNNs learn better

- Averaging noise measurements (audio, image, radar…) improves S/N (reduce random fluctuations)
- Trade-off: More averaging improves S/N but increases compute cost and slows performance.
- Full gradient descent (complete averaging): Computes gradients using the full dataset. Accurate, clean gradients, but slow and requires massive memory.
- (pure) Stochastic Gradient Descent (no averaging): Use a single data point to calculate the gradient at each step. Very noisy and inefficient for GPU parallelism
- (sweet spot!) **Mini-batch gradient descent**: Uses a small subset (32, 64, 128…) of data to compute gradients, balancing speed and precision. Controlled noise prevents overfitting, helps escape local minima, and mini-batches are perfect for parallel GPU processing.
- Why faster? Mini-batches enable immediate parameter updates: The model dynamically refines its trajectory, like an explorer constantly updating their map as they travel. This **real-time adaptation** leverages new information to improve the search direction on the fly, rather than waiting to process all possible paths. As the model nears the minimum, gradients from smaller batches align more closely with the true gradient, making updates more precise as we move along.

Pure SGD (1), mini-batch GD, Full GD

Sweet Spot: Mini-Batch GD – Converges Faster, Optimal Noise, Better GPU Usage

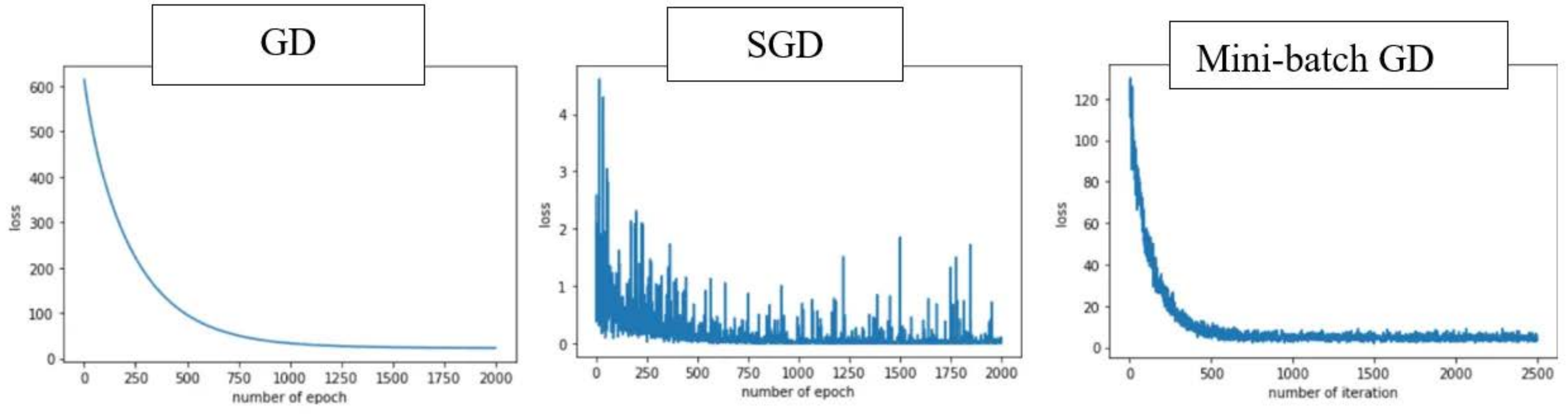


Training Progress

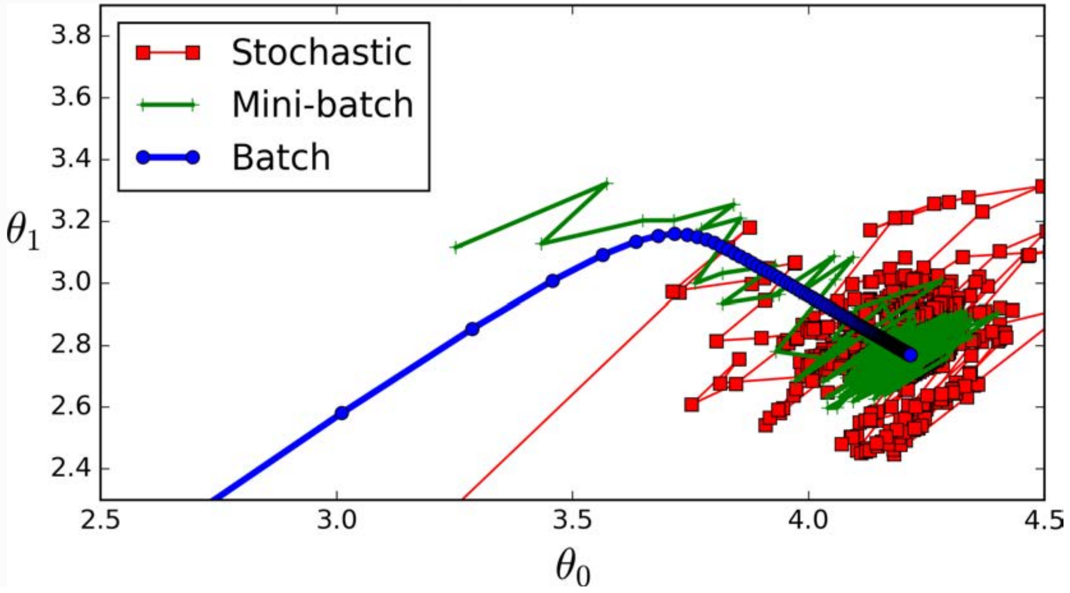
Progress	1% (round 1/100, batch 782/782)
Total	782/78 200 batches, 50 048/5 004 800 examples
Time	5s elapsed, 7m27s left, 15 132 examples/s
Method	ADAM optimizer, batch size 64, CPU
	learning rate 7.37×10^{-4}

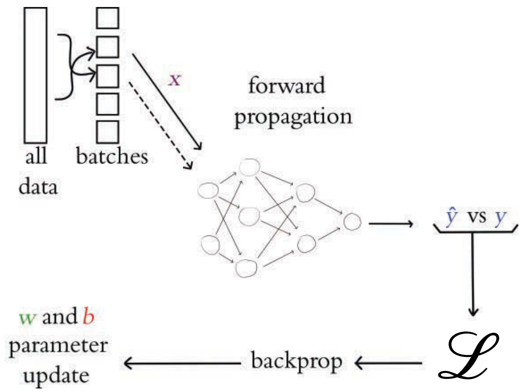
	current	round	validation
loss	5.36×10^{-2}	1.01×10^{-1}	computing...

animation



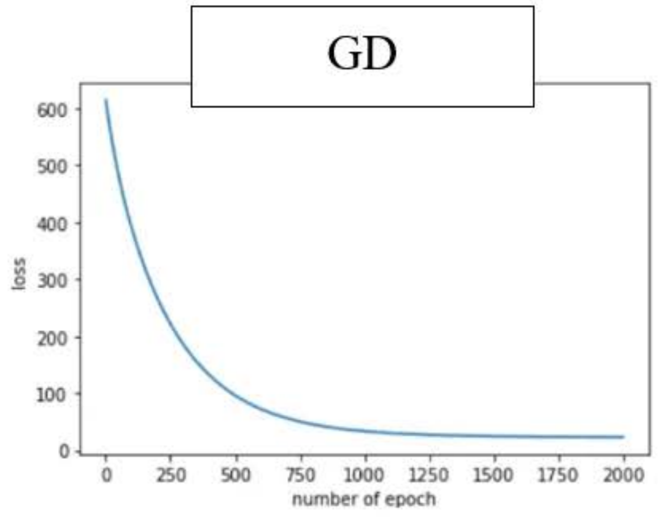
source: Aurelian Geron

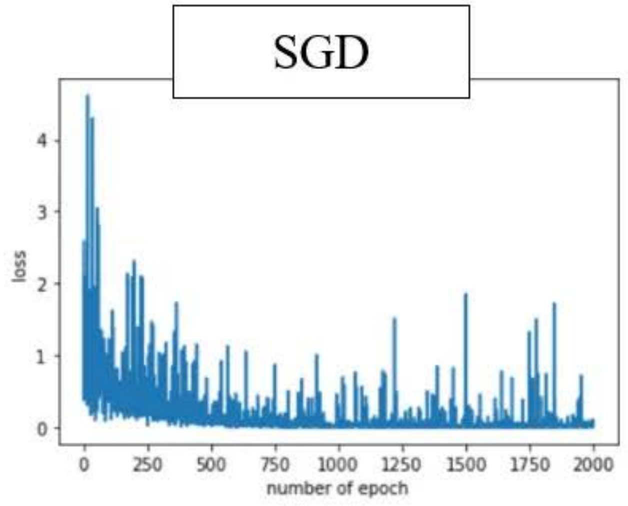


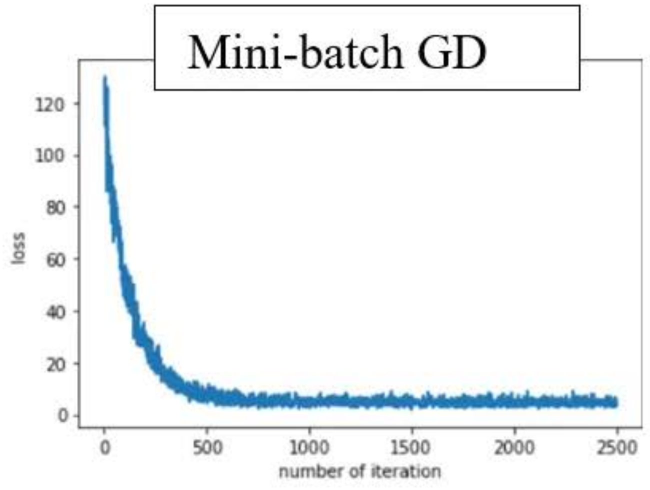


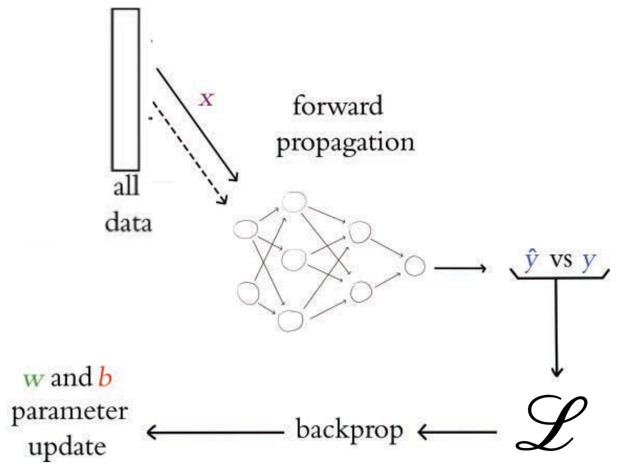
Pure SGD (1 batch) vs Mini SGD vs Full GD

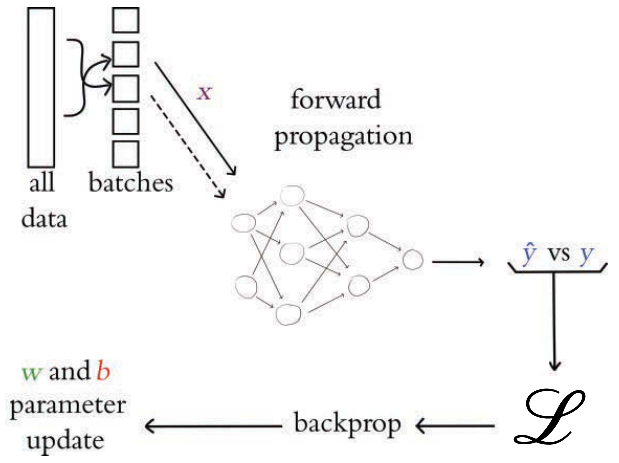
Sweet Spot: Mini-Batch GD – Converges Faster, Optimal Noise, Better GPU Usage

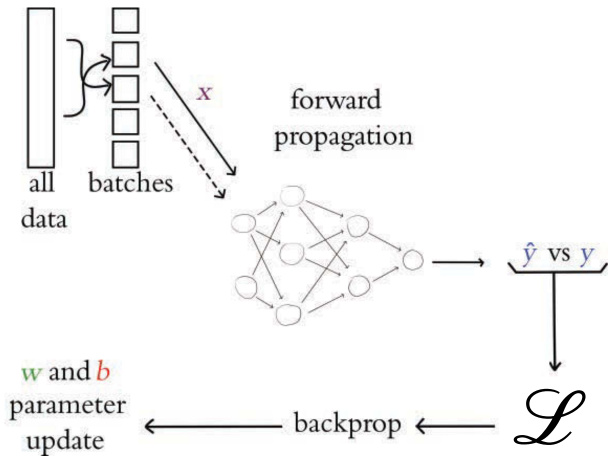

GD
loss
number of epoch


SGD
loss
number of epoch


Mini-batch GD
loss
number of iteration


all data
x
forward propagation
ŷ vs y
ℒ
backprop
w and b parameter update


all data
batches
x
forward propagation
ŷ vs y
ℒ
backprop
w and b parameter update


all data
batches
x
forward propagation
ŷ vs y
ℒ
backprop
w and b parameter update