Automatic differentiation and neural networks

Outline

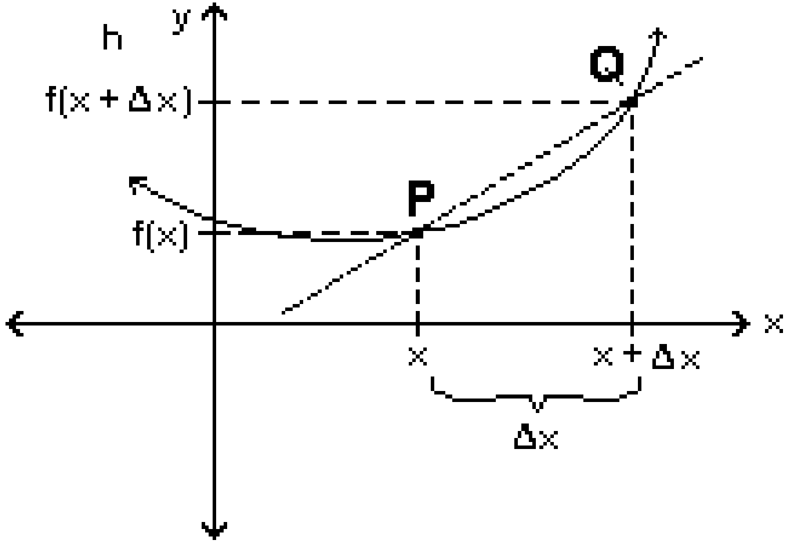
Lecture 1 : Automatic differentiation and neural networks

Lecture 2 : Physics informed neural networks

Lecture 3: Distillation of neural networks with symbolic regression

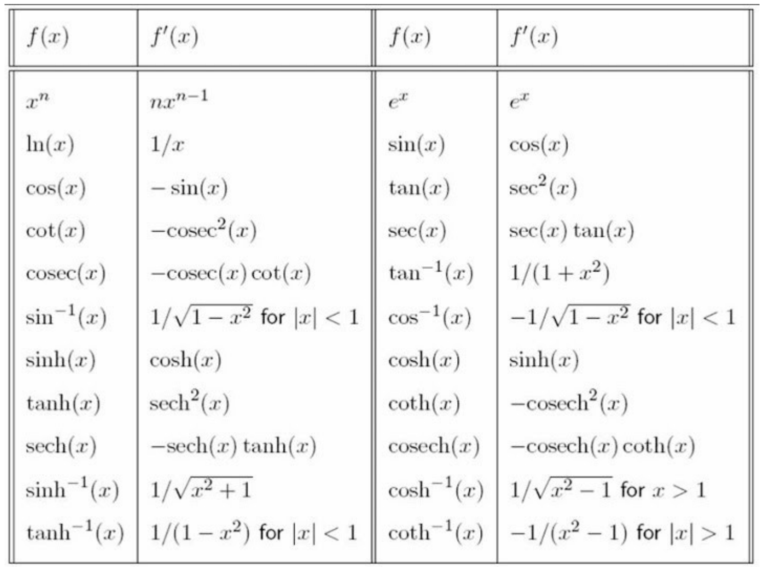
The derivative

$$\lim_{h\to 0}\frac{f\left(x+h\right)-f\left(x\right)}{h}$$



$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \ldots.$$

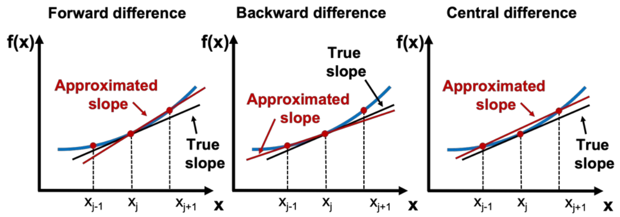
Analytical x numerical differentiation



$f(x)$	$f'(x)$	$f(x)$	$f'(x)$
x^n	nx^{n-1}	e^x	e^x
$\ln(x)$	$1/x$	$\sin(x)$	$\cos(x)$
$\cos(x)$	$-\sin(x)$	$\tan(x)$	$\sec^2(x)$
$\cot(x)$	$-\mathrm{cosec}^2(x)$	$\sec(x)$	$\sec(x)\tan(x)$
$\mathrm{cosec}(x)$	$-\mathrm{cosec}(x)\cot(x)$	$\tan^{-1}(x)$	$1/(1+x^2)$
$\sin^{-1}(x)$	$1/\sqrt{1-x^2}$ for $\|x\|<1$	$\cos^{-1}(x)$	$-1/\sqrt{1-x^2}$ for $\|x\|<1$
$\sinh(x)$	$\cosh(x)$	$\cosh(x)$	$\sinh(x)$
$\tanh(x)$	$\mathrm{sech}^2(x)$	$\coth(x)$	$-\mathrm{cosech}^2(x)$
$\mathrm{sech}(x)$	$-\mathrm{sech}(x)\tanh(x)$	$\mathrm{cosech}(x)$	$-\mathrm{cosech}(x)\coth(x)$
$\sinh^{-1}(x)$	$1/\sqrt{x^2+1}$	$\cosh^{-1}(x)$	$1/\sqrt{x^2-1}$ for $x>1$
$\tanh^{-1}(x)$	$1/(1-x^2)$ for $\|x\|<1$	$\coth^{-1}(x)$	$-1/(x^2-1)$ for $\|x\|>1$

$$F'(x) = f'(g(x)) \cdot g'(x)$$

$$\frac{f(x+\Delta x) - f(x)}{\Delta x}$$



$O(\Delta x^2)$ centered difference approximations:

$f'(x): \quad \{f(x+\Delta x) - f(x-\Delta x)\}/(2\Delta x)$

$f''(x): \quad \{f(x+\Delta x) - 2f(x) + f(x-\Delta x)\}/\Delta x^2$

$O(\Delta x^2)$ forward difference approximations:

$f'(x): \quad \{-3f(x) + 4f(x+\Delta x) - f(x+2\Delta x)\}/(2\Delta x)$

$f''(x): \quad \{2f(x) - 5f(x+\Delta x) + 4f(x+2\Delta x) - f(x+3\Delta x)\}/\Delta x^3$

$O(\Delta x^2)$ backward difference approximations:

$f'(x): \quad \{3f(x) - 4f(x-\Delta x) + f(x-2\Delta x)\}/(2\Delta x)$

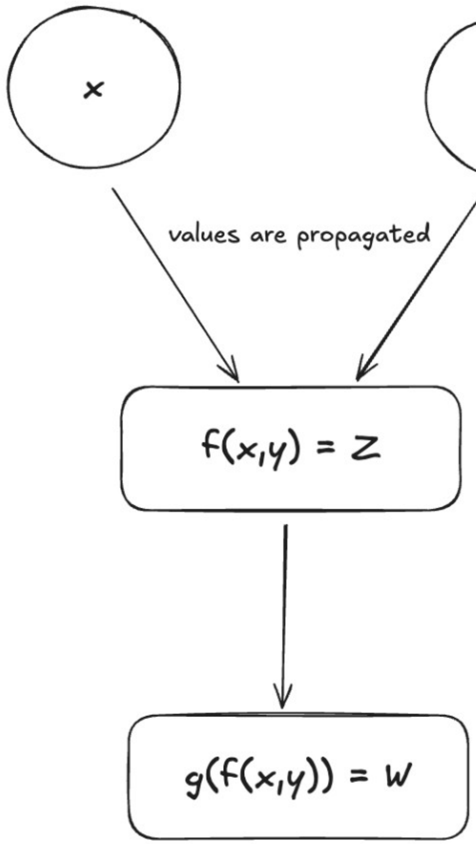
$f''(x): \quad \{2f(x) - 5f(x-\Delta x) + 4f(x-2\Delta x) - f(x-3\Delta x)\}/\Delta x^3$

$O(\Delta x^4)$ centered difference approximations:

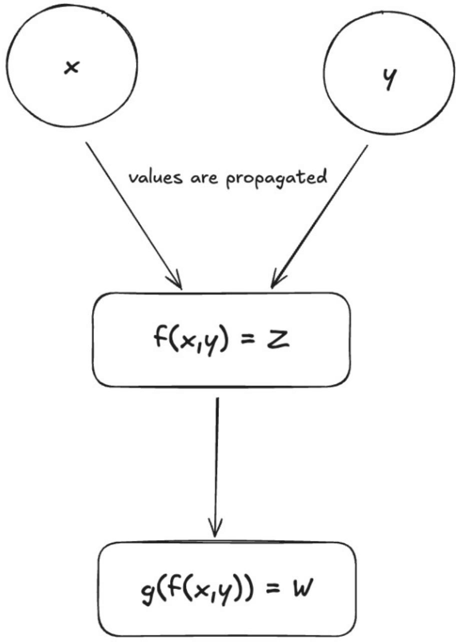
$f'(x): \quad \{-f(x+2\Delta x) + 8f(x+\Delta x) - 8f(x-\Delta x) + f(x-2\Delta x)\}/(12\Delta x)$

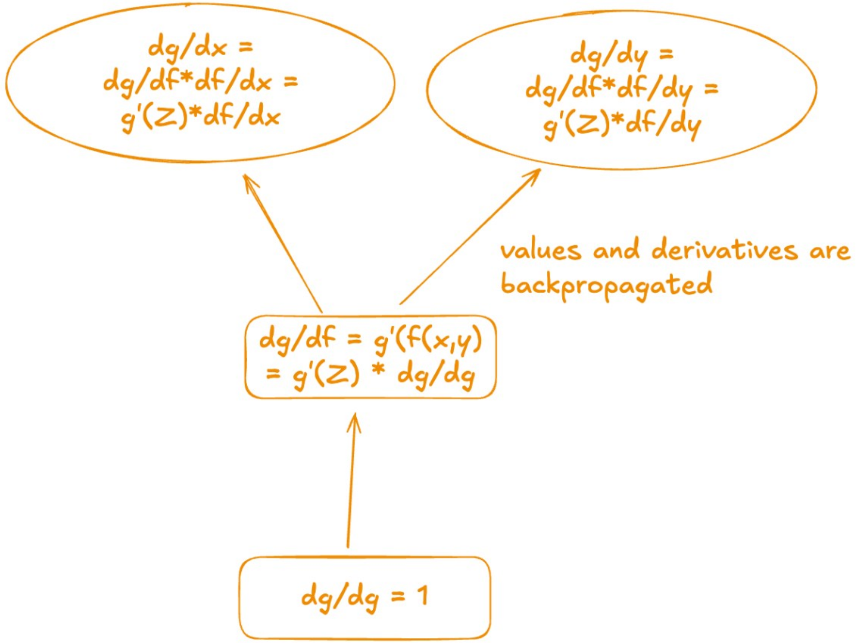
$f''(x): \quad \{-f(x+2\Delta x) + 16f(x+\Delta x) - 30f(t) + 16f(x-\Delta x) - f(x-2\Delta x)\}/(12\Delta x^2)$

The calculation graph



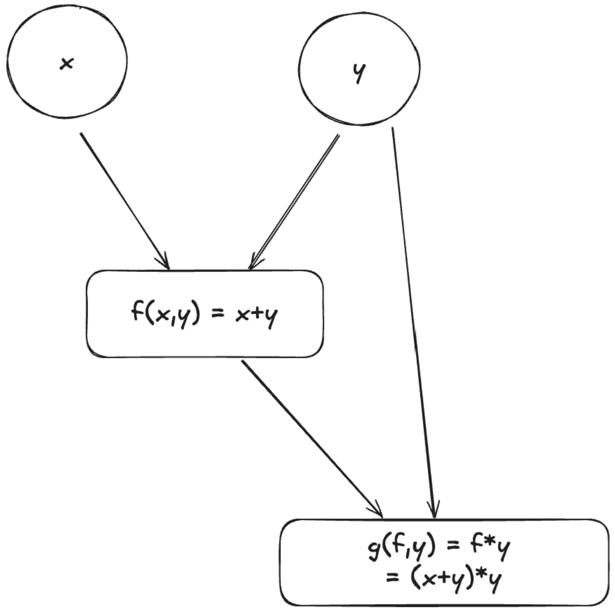
Automatic differentiation



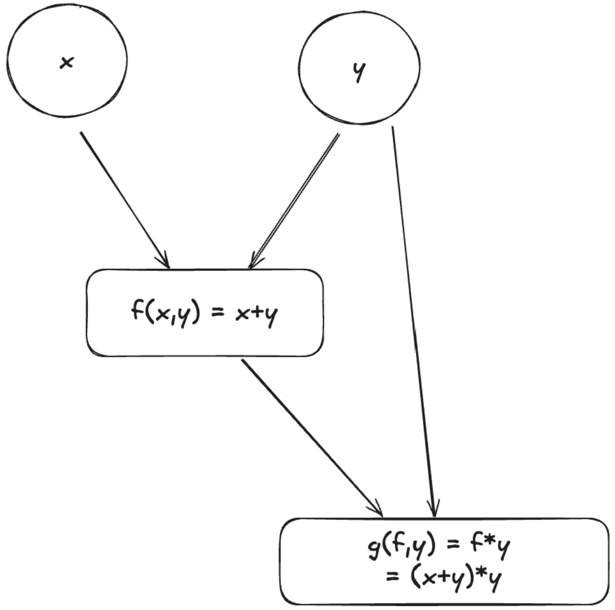

dg/dx = dg/df*df/dx = g'(Z)*df/dx
dg/dy = dg/df*df/dy = g'(Z)*df/dy
values and derivatives are backpropagated
dg/df = g'(f(x,y) = g'(Z) * dg/dg
dg/dg = 1

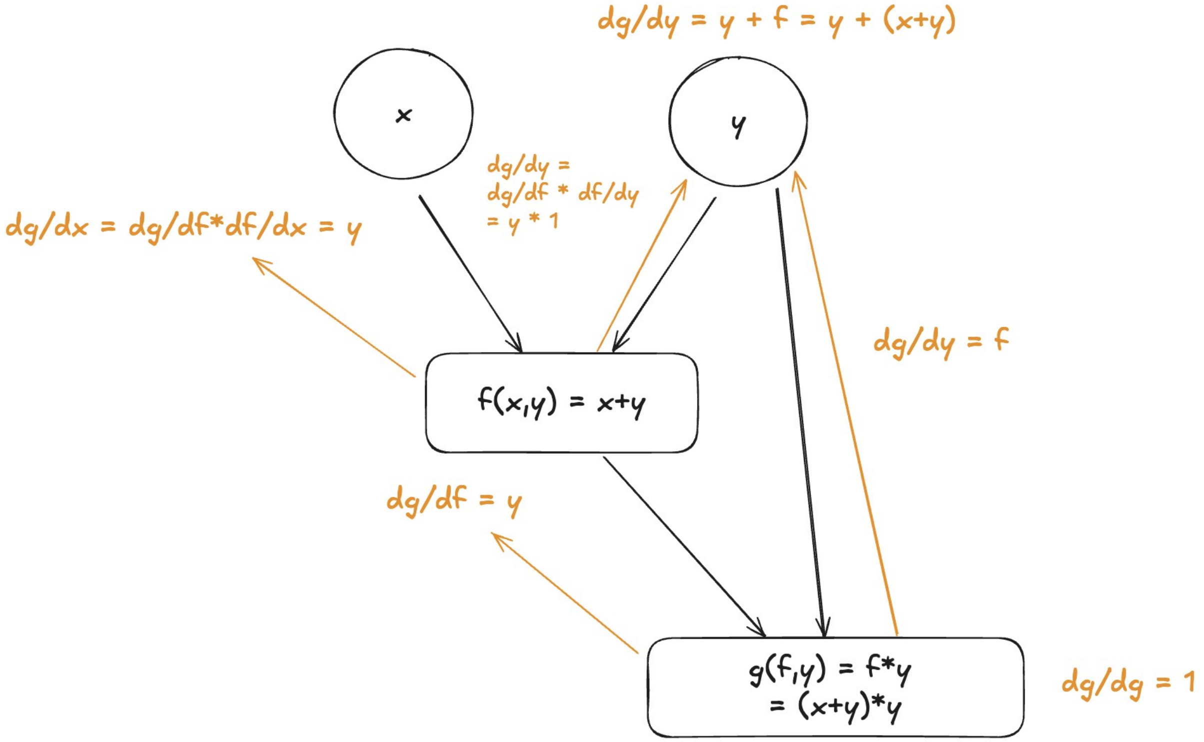
Therefore, every node needs to store the forward and backprop values, in addition to the analytical definition of its derivative."

Another example

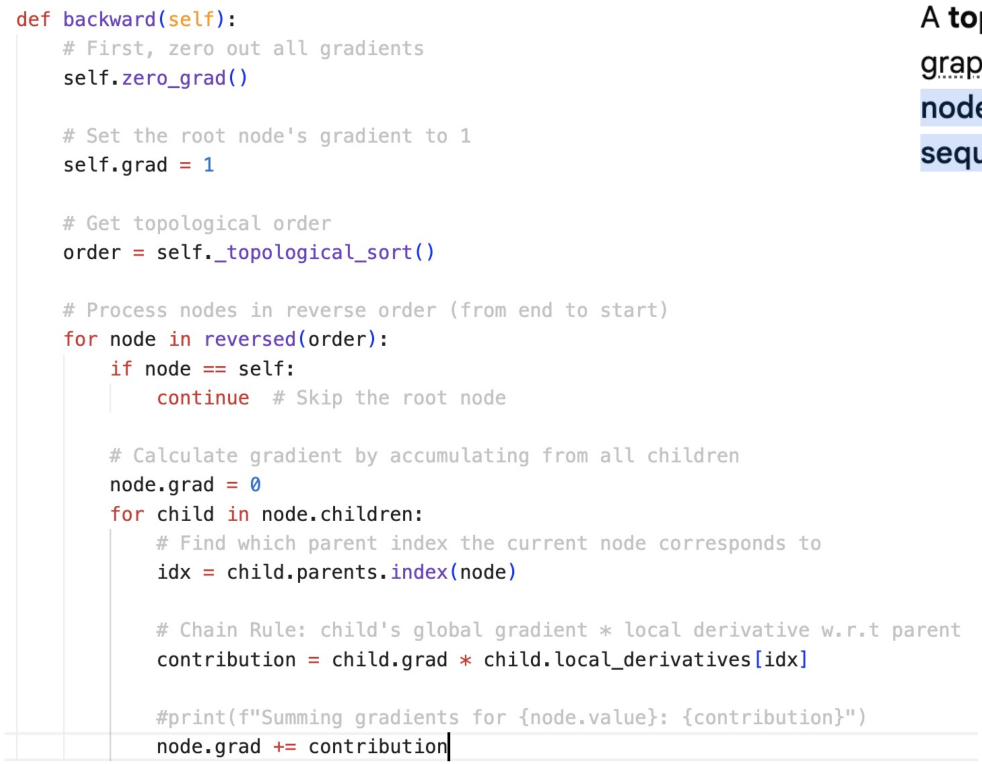


Another example

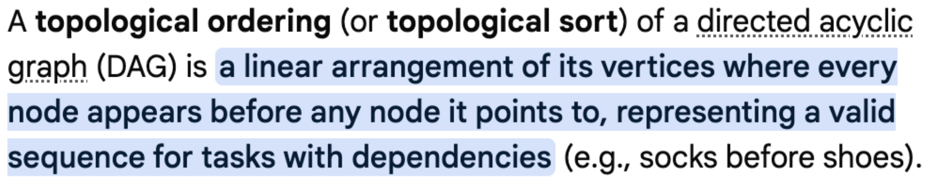




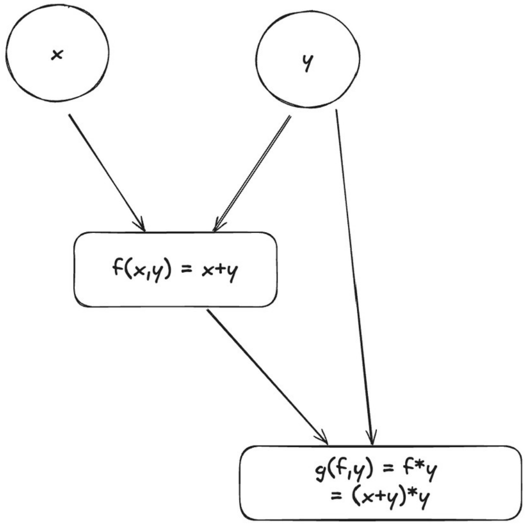
The need for topological ordering



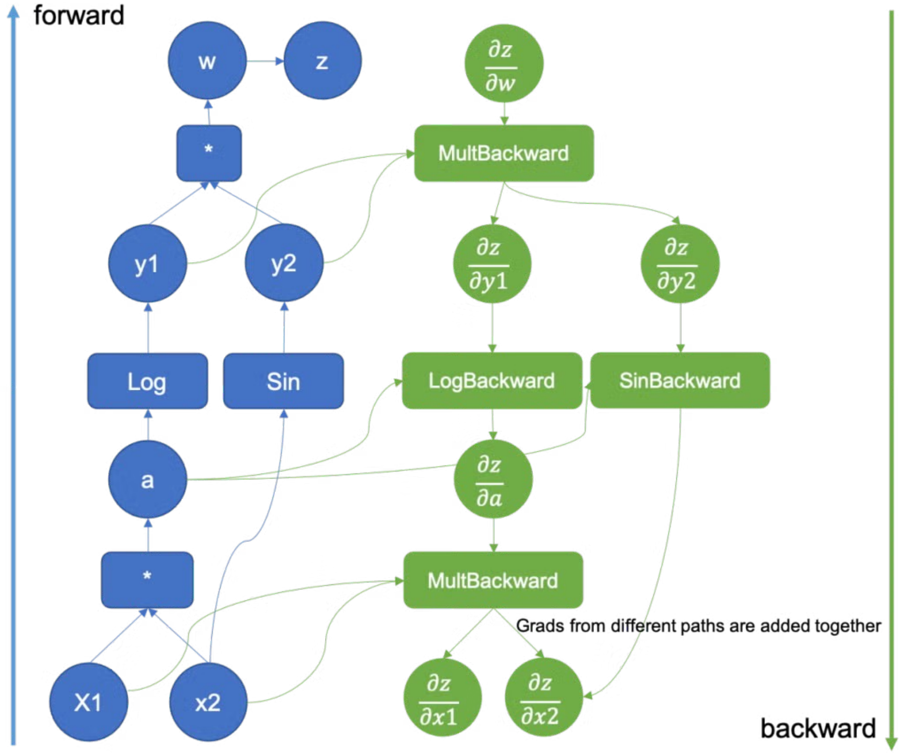
```python
def backward(self):
    # First, zero out all gradients
    self.zero_grad()

    # Set the root node's gradient to 1
    self.grad = 1

    # Get topological order
    order = self._topological_sort()

    # Process nodes in reverse order (from end to start)
    for node in reversed(order):
        if node == self:
            continue  # Skip the root node

        # Calculate gradient by accumulating from all children
        node.grad = 0
        for child in node.children:
            # Find which parent index the current node corresponds to
            idx = child.parents.index(node)

            # Chain Rule: child's global gradient * local derivative w.r.t parent
            contribution = child.grad * child.local_derivatives[idx]

            #print(f"Summing gradients for {node.value}: {contribution}")
            node.grad += contribution
```



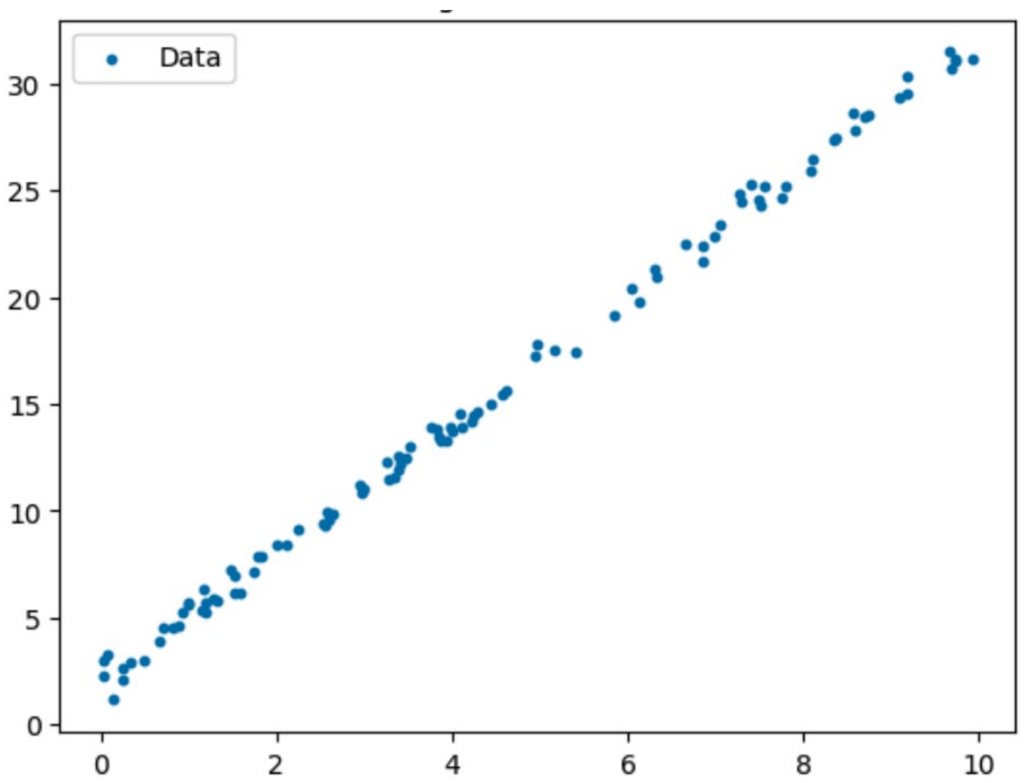
A **topological ordering** (or **topological sort**) of a directed acyclic graph (DAG) is a linear arrangement of its vertices where every node appears before any node it points to, representing a valid sequence for tasks with dependencies (e.g., socks before shoes).



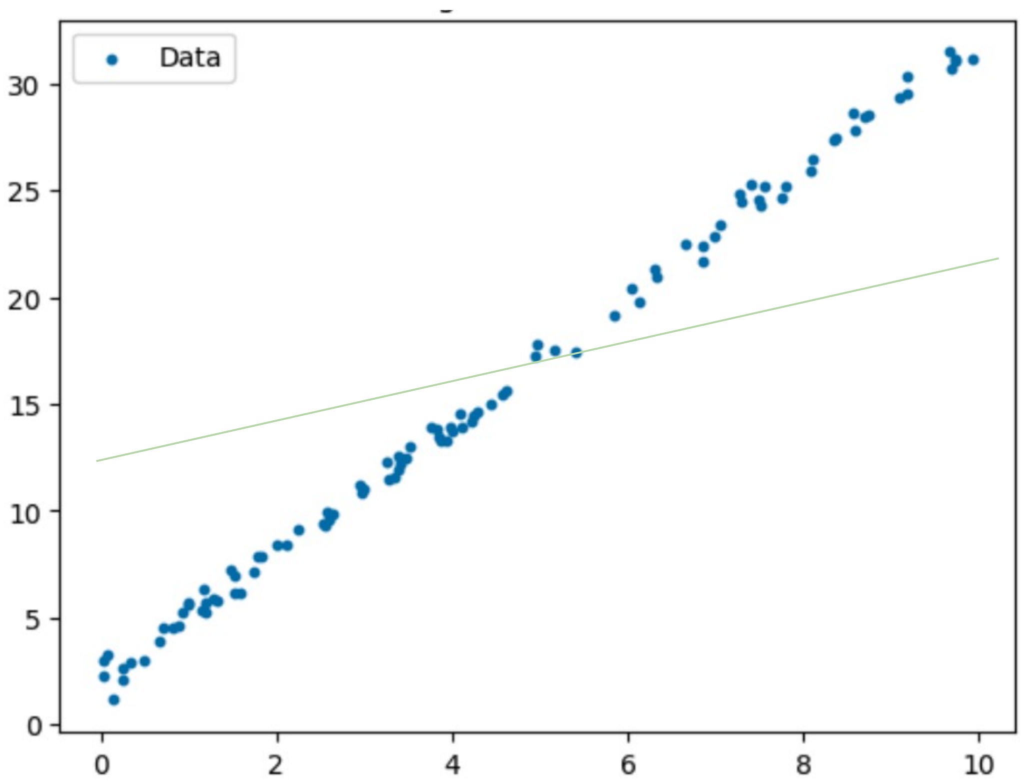
Our stack


forward
w
z
*
y1
y2
Log
Sin
a
*
X1
x2
$\frac{\partial z}{\partial w}$
MultBackward
$\frac{\partial z}{\partial y1}$
$\frac{\partial z}{\partial y2}$
LogBackward
SinBackward
$\frac{\partial z}{\partial a}$
MultBackward
Grads from different paths are added together
$\frac{\partial z}{\partial x1}$
$\frac{\partial z}{\partial x2}$
backward

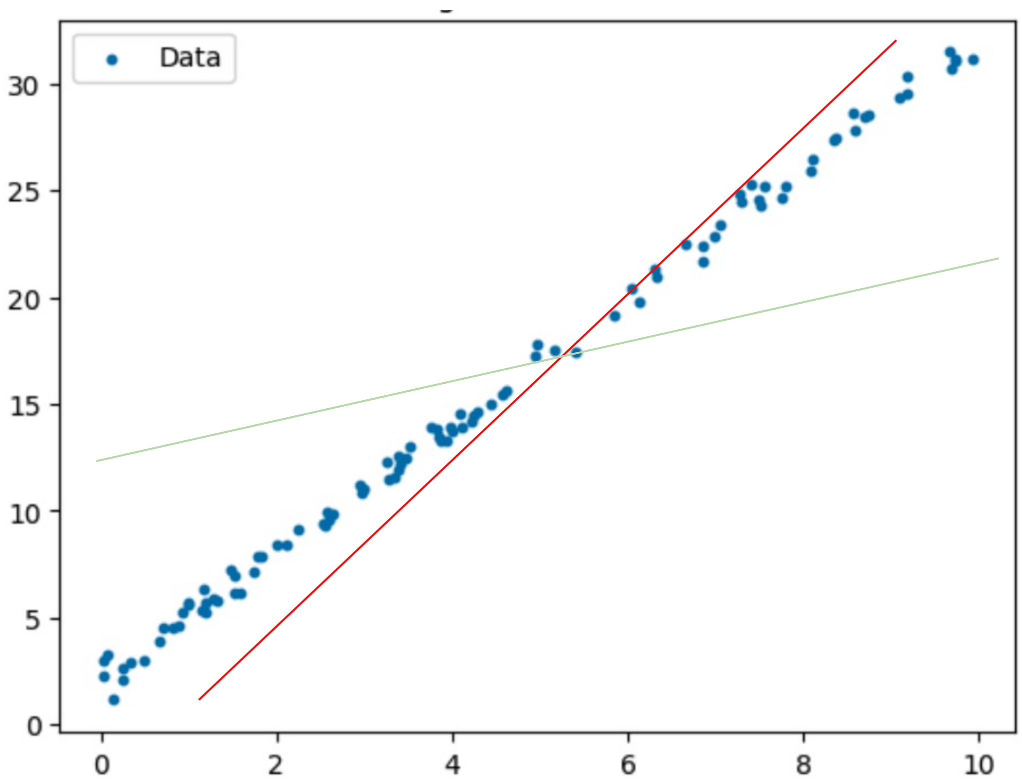
Approximating analytical functions



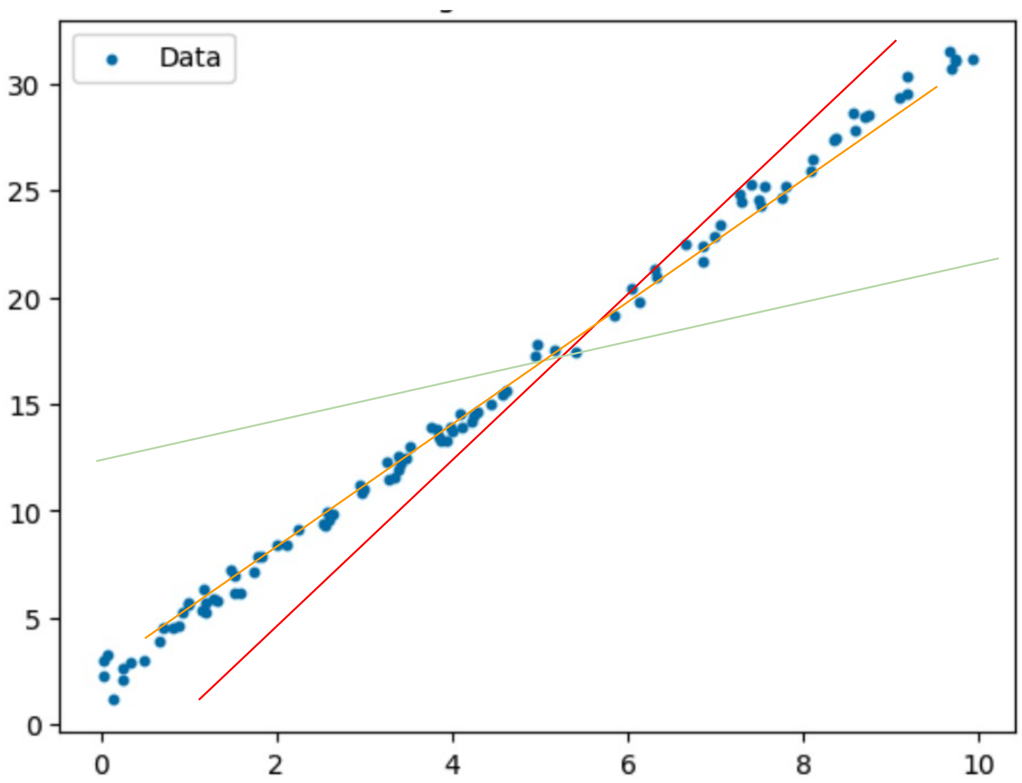
Approximating analytical functions



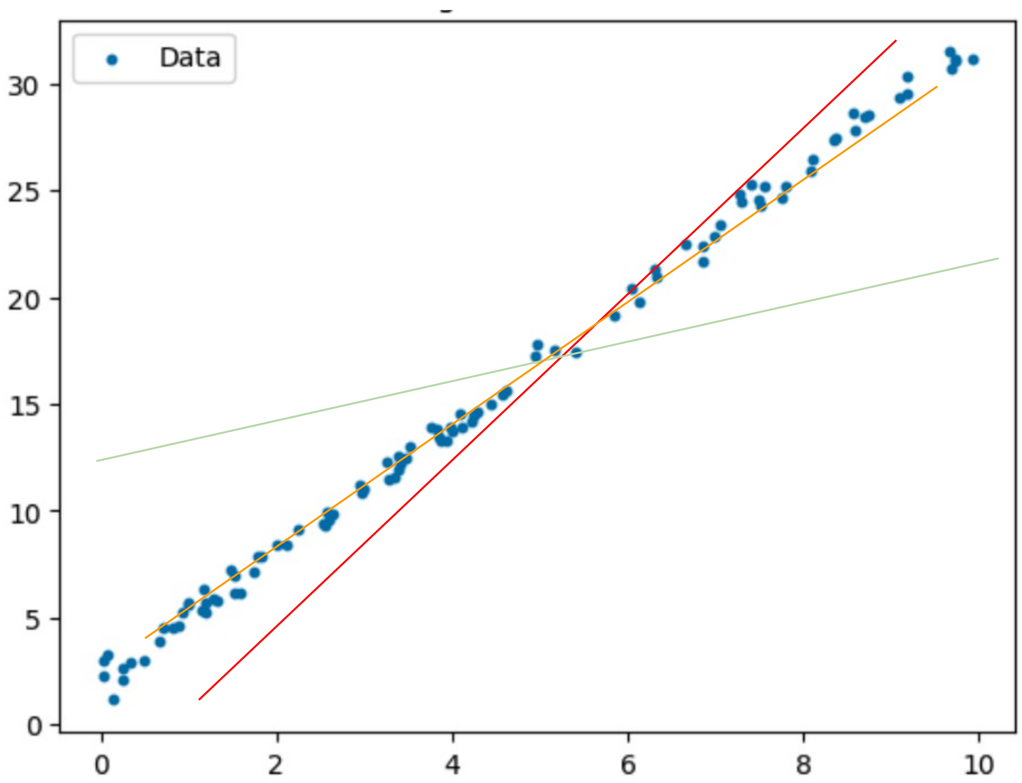
Approximating analytical functions



Approximating analytical functions



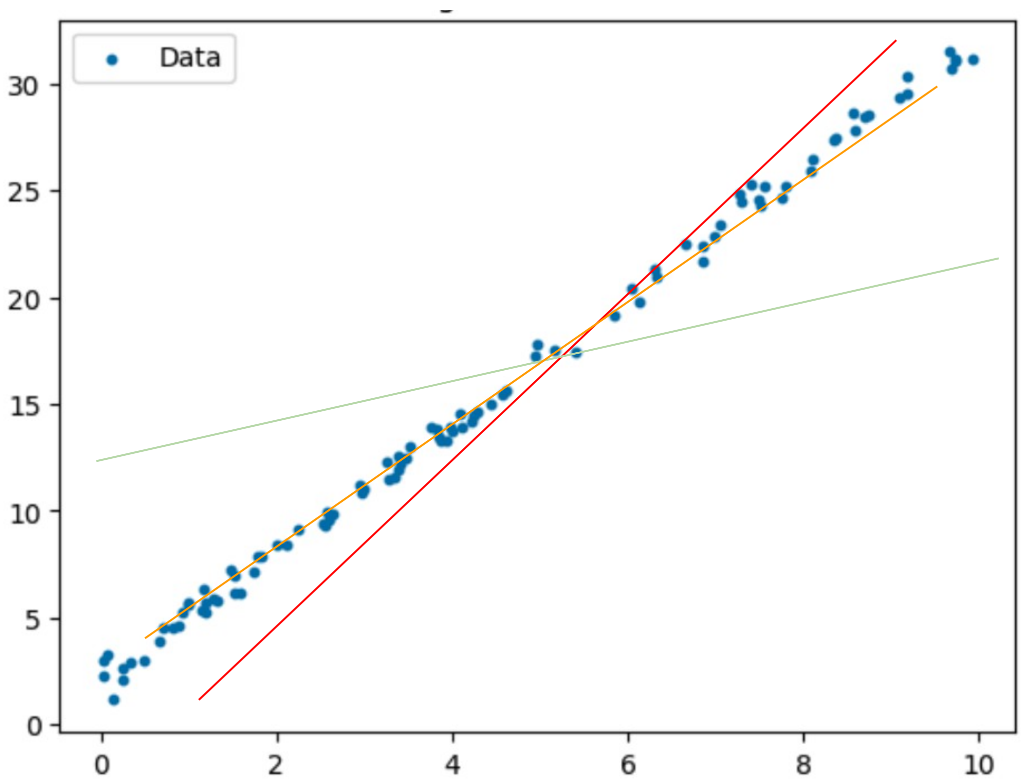
Approximating analytical functions



$$L(w,b) = ||wx + b - y||^2$$

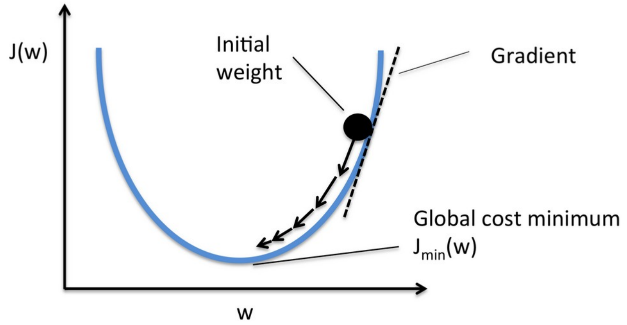
We must minimize the loss!

Approximating analytical functions



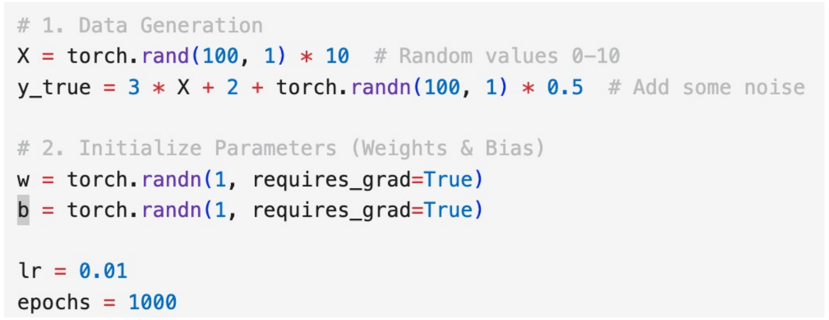
$$L(w, b) = ||wx + b - y||^2$$

With gradient descent:

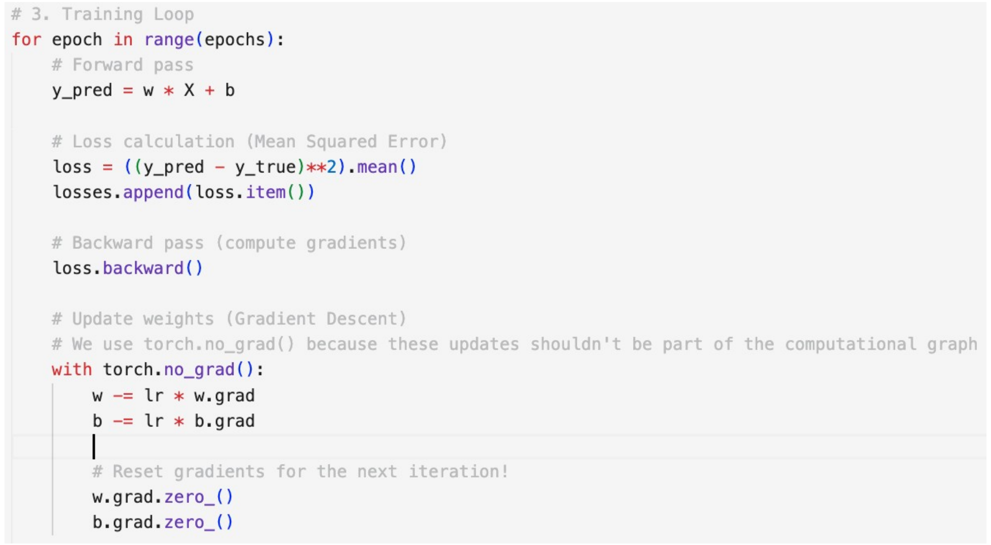


$$w_{t+1} = w_t - \alpha \frac{dJ}{dw}$$

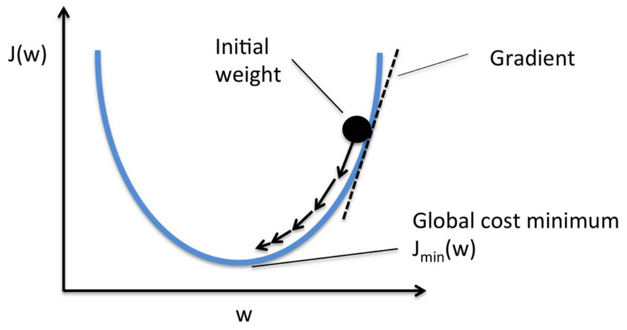
Approximating analytical functions



```python
# 1. Data Generation
X = torch.rand(100, 1) * 10  # Random values 0-10
y_true = 3 * X + 2 + torch.randn(100, 1) * 0.5  # Add some noise

# 2. Initialize Parameters (Weights & Bias)
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

lr = 0.01
epochs = 1000
```

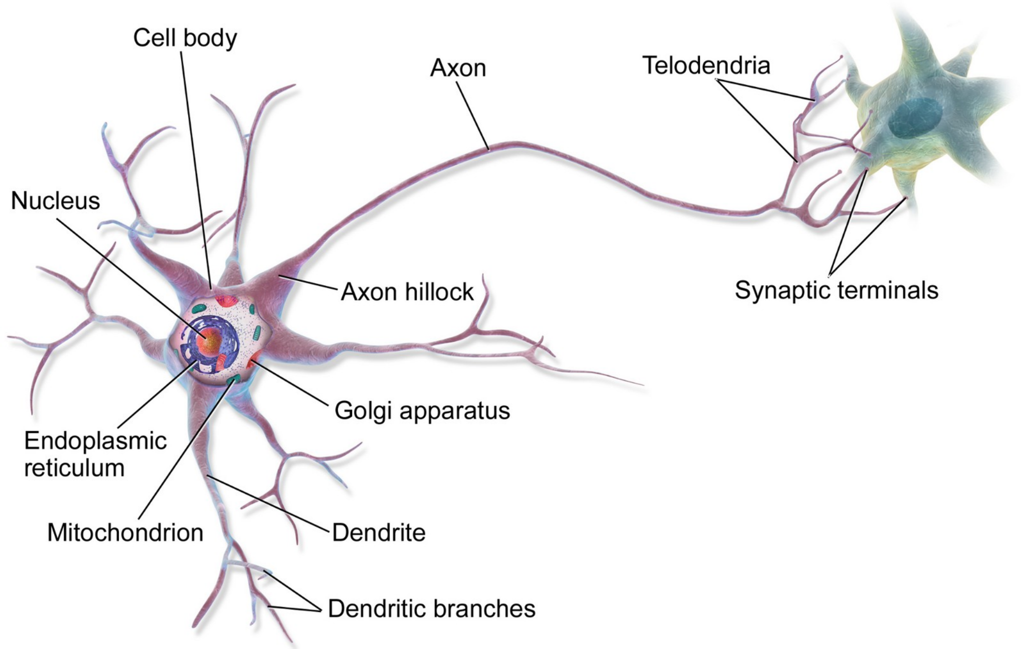


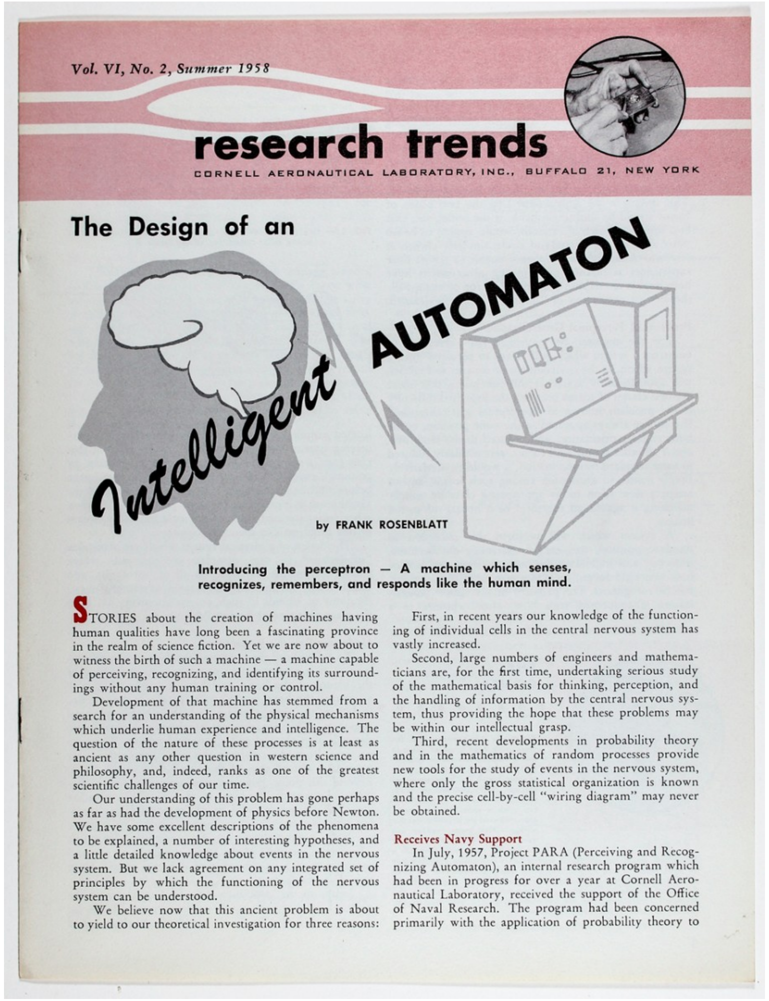
```python
# 3. Training Loop
for epoch in range(epochs):
    # Forward pass
    y_pred = w * X + b

    # Loss calculation (Mean Squared Error)
    loss = ((y_pred - y_true)**2).mean()
    losses.append(loss.item())

    # Backward pass (compute gradients)
    loss.backward()

    # Update weights (Gradient Descent)
    # We use torch.no_grad() because these updates shouldn't be part of the computational graph
    with torch.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

        # Reset gradients for the next iteration!
        w.grad.zero_()
        b.grad.zero_()
```



$$w_{t+1} = w_t - \alpha \frac{dJ}{dw}$$

The neuron




Vol. VI, No. 2, Summer 1958

research trends

CORNELL AERONAUTICAL LABORATORY, INC., BUFFALO 21, NEW YORK

The Design of an *Intelligent* AUTOMATON

by FRANK ROSENBLATT

Introducing the perceptron — A machine which senses, recognizes, remembers, and responds like the human mind.

STORIES about the creation of machines having human qualities have long been a fascinating province in the realm of science fiction. Yet we are now about to witness the birth of such a machine — a machine capable of perceiving, recognizing, and identifying its surroundings without any human training or control.

Development of that machine has stemmed from a search for an understanding of the physical mechanisms which underlie human experience and intelligence. The question of the nature of these processes is at least as ancient as any other question in western science and philosophy, and, indeed, ranks as one of the greatest scientific challenges of our time.

Our understanding of this problem has gone perhaps as far as had the development of physics before Newton. We have some excellent descriptions of the phenomena to be explained, a number of interesting hypotheses, and a little detailed knowledge about events in the nervous system. But we lack agreement on any integrated set of principles by which the functioning of the nervous system can be understood.

We believe now that this ancient problem is about to yield to our theoretical investigation for three reasons:

First, in recent years our knowledge of the functioning of individual cells in the central nervous system has vastly increased.

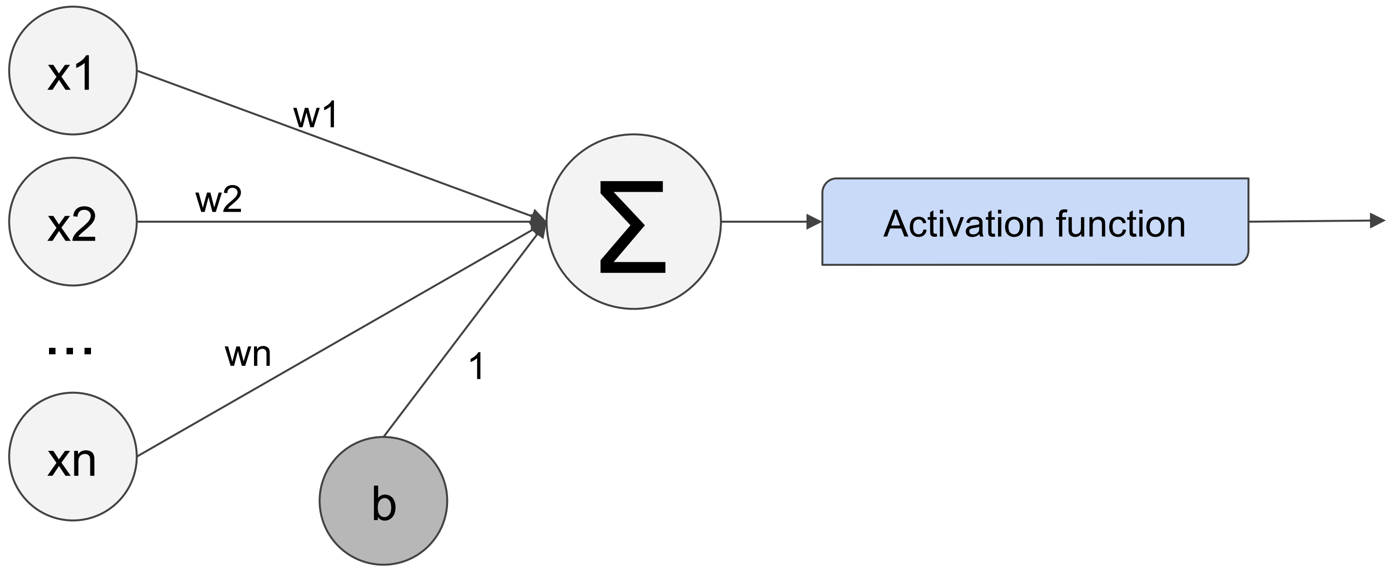
Second, large numbers of engineers and mathematicians are, for the first time, undertaking serious study of the mathematical basis for thinking, perception, and the handling of information by the central nervous system, thus providing the hope that these problems may be within our intellectual grasp.

Third, recent developments in probability theory and in the mathematics of random processes provide new tools for the study of events in the nervous system, where only the gross statistical organization is known and the precise cell-by-cell "wiring diagram" may never be obtained.

Receives Navy Support

In July, 1957, Project PARA (Perceiving and Recognizing Automaton), an internal research program which had been in progress for over a year at Cornell Aeronautical Laboratory, received the support of the Office of Naval Research. The program had been concerned primarily with the application of probability theory to

The perceptron



Activation functions

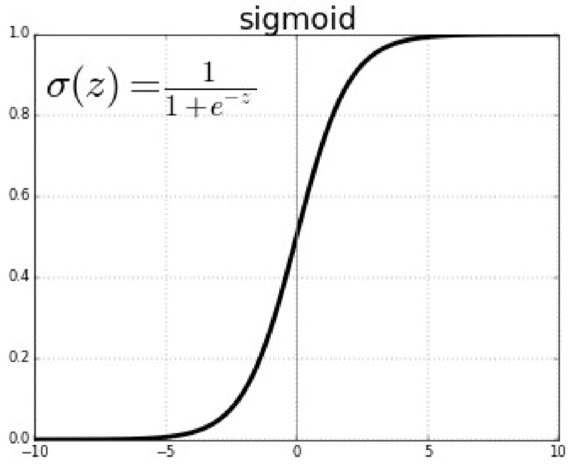
$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

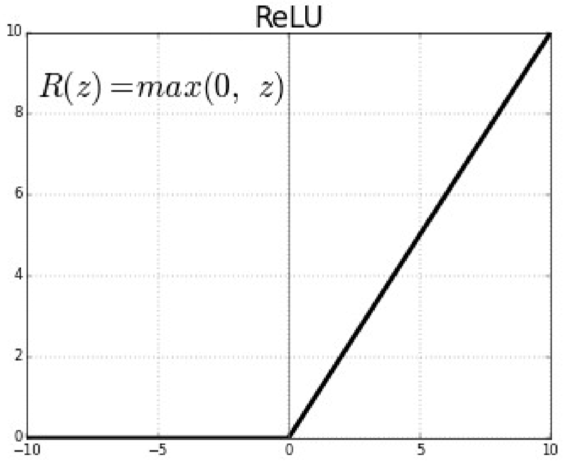
$$\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

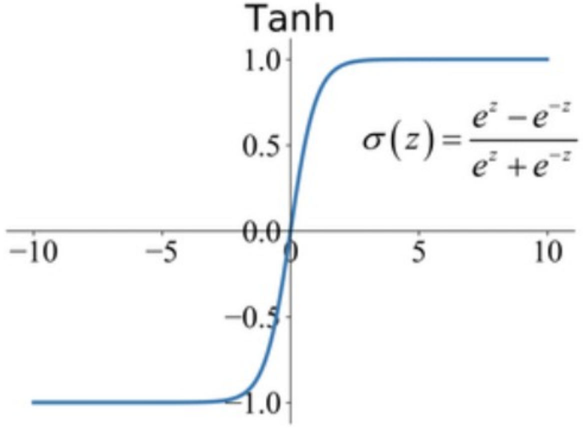
Activation functions

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

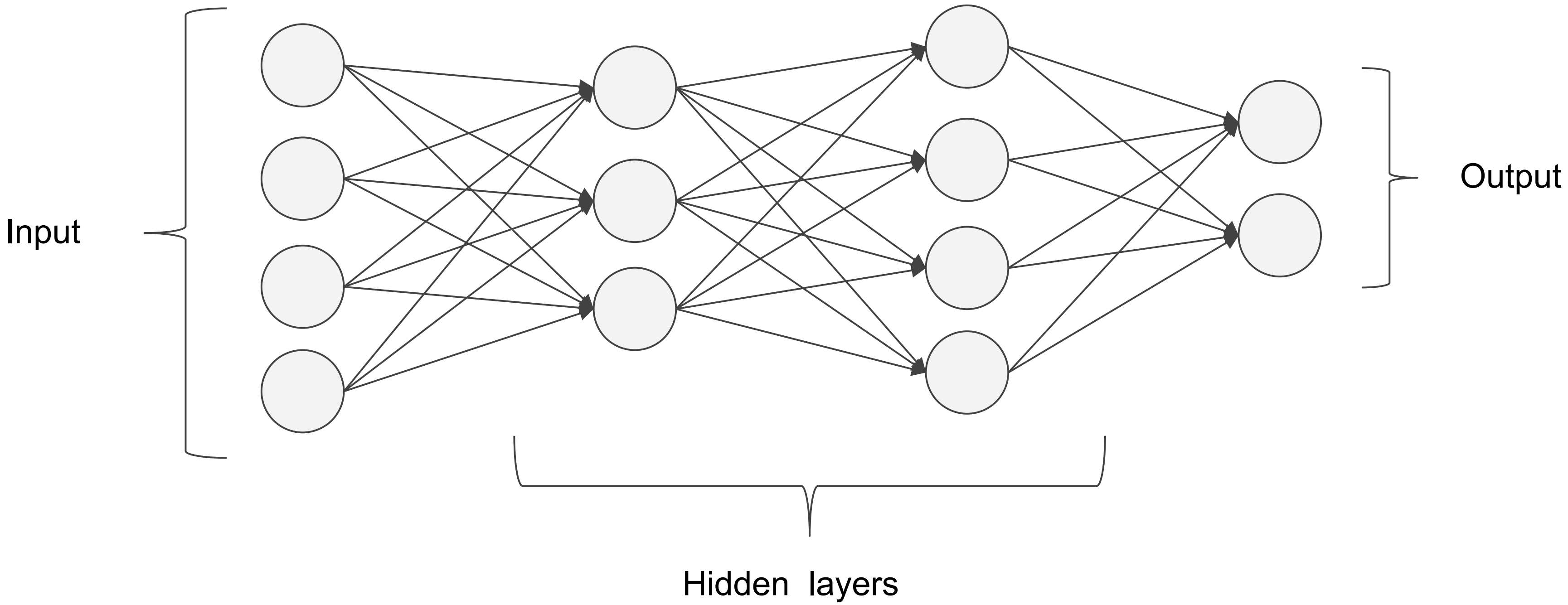
$$\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$



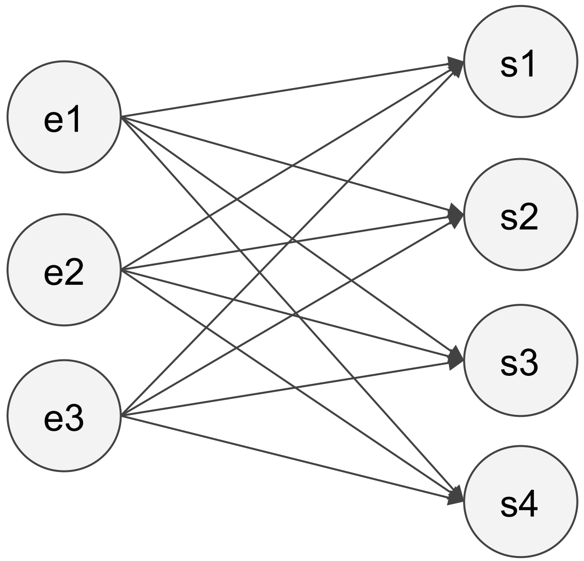




Neural networks

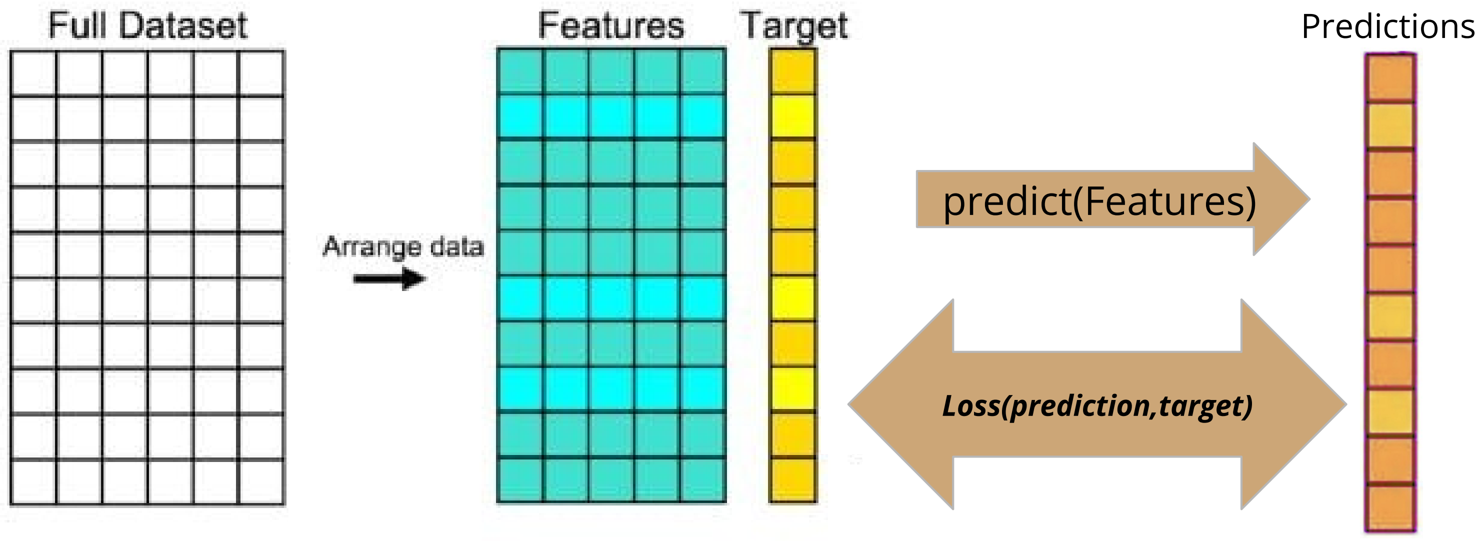


The layer

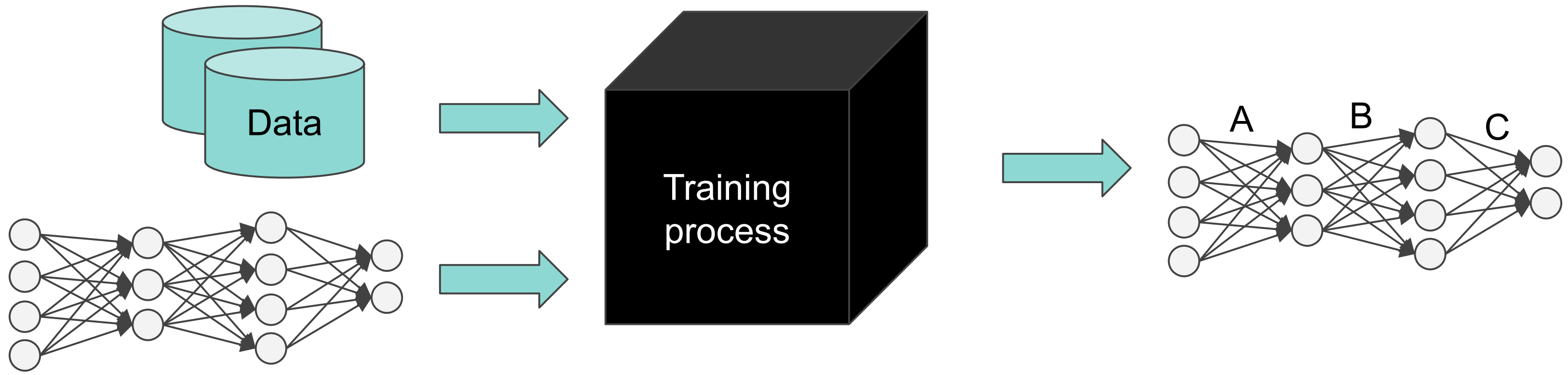


$$\Rightarrow \begin{bmatrix} e1 & e2 & e3 \end{bmatrix} \underbrace{\begin{pmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \end{pmatrix}}_{A} = \begin{pmatrix} s1 \\ s2 \\ s3 \\ s4 \end{pmatrix}^{T}$$

Machine learning



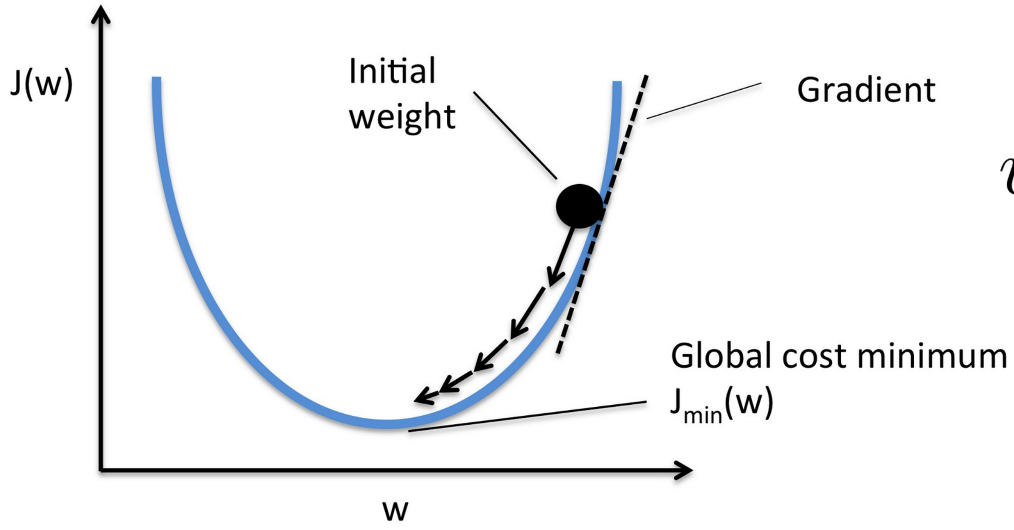
Machine learning



We need A,B,C (weights) that minimize some ***loss function***

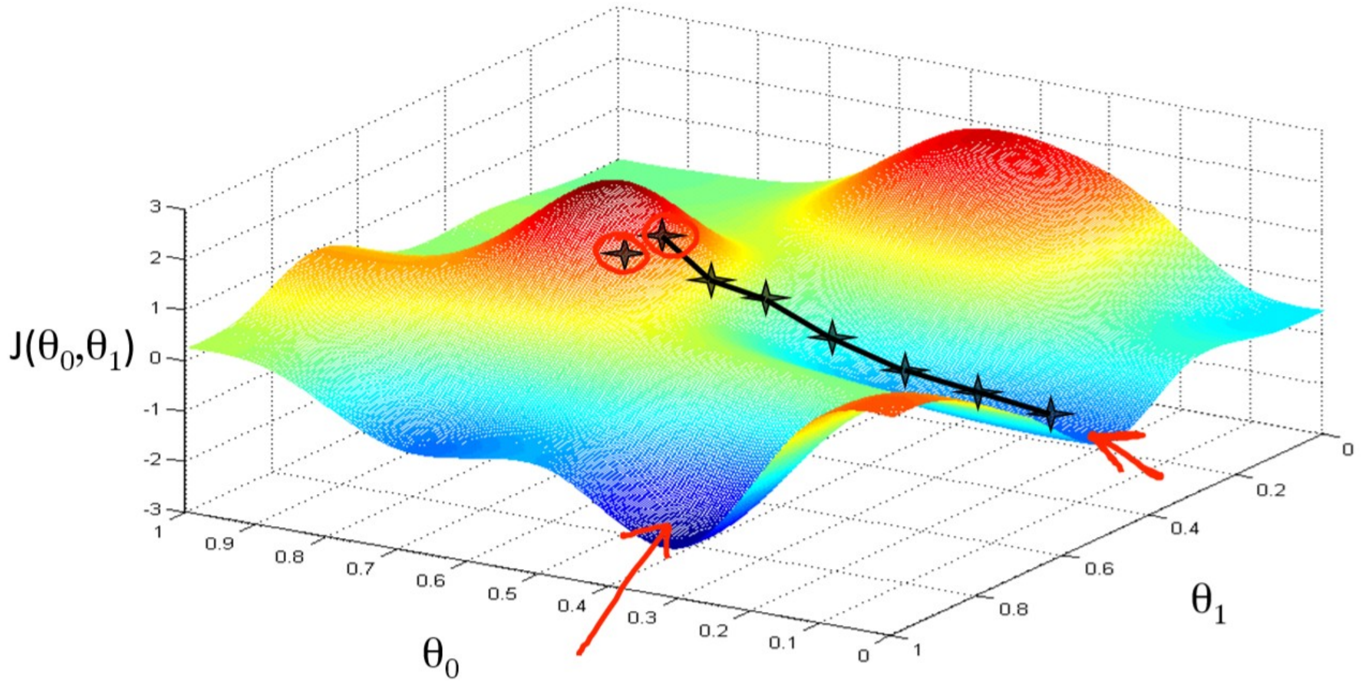
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad CrossEntropy = -\sum_{i=1}^{n}(y_i \ln \hat{y}_i)$$

Gradient descent

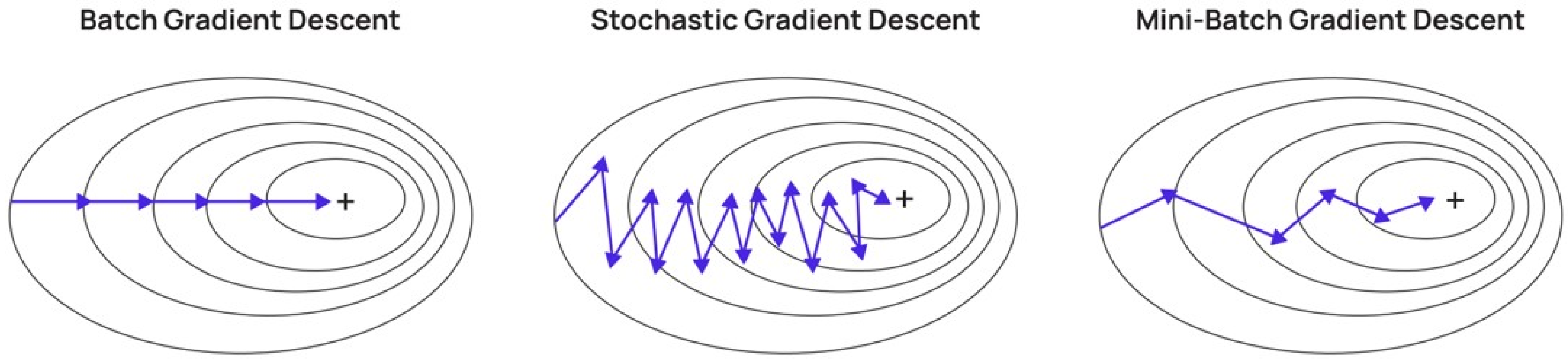


$$w_{t+1} = w_t - \alpha \frac{dJ}{dw}$$

Gradient descent



Gradient descent



Full training loop

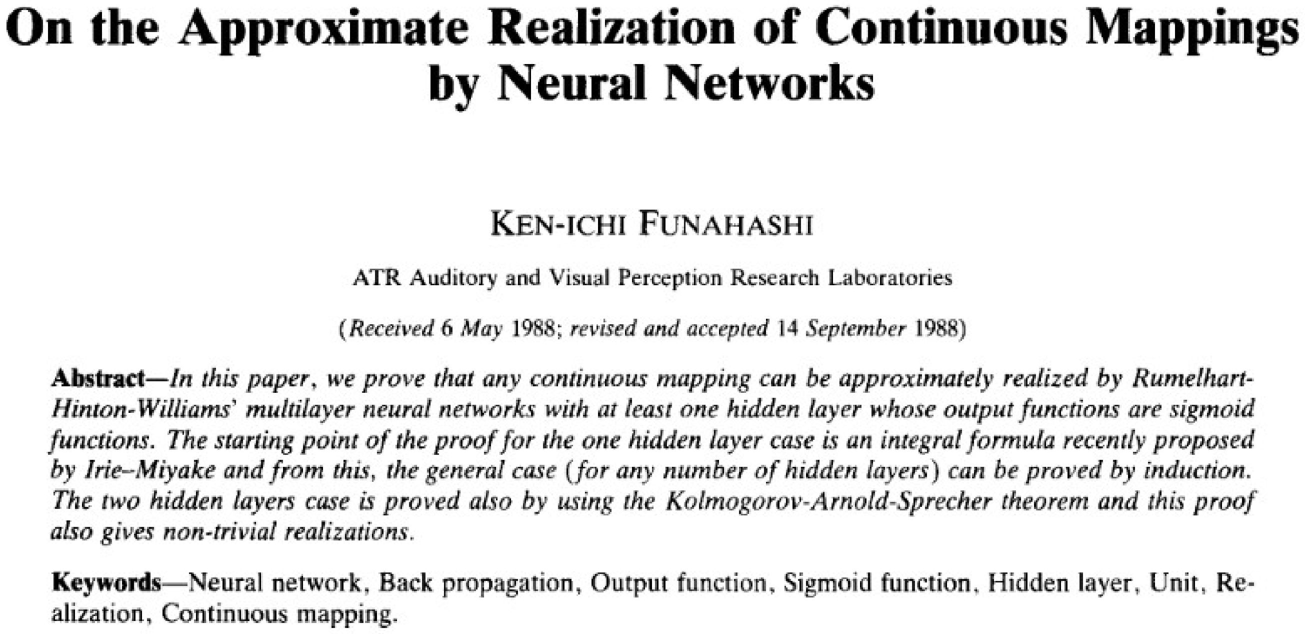
Weights are randomly initialized.

The dataset is divided into batches, for example, with a size of 32. Whenever the entire dataset is processed, an **epoch** is completed.

For each batch:

- **Forward pass:** Each input is processed by the network, producing outputs that may be far from the expected values. All intermediate values for the batch are stored.
- The **batch error** is measured using the desired loss function.
- **Backpropagation** is used to calculate how much each edge (weight) contributes to the error.
- A **gradient descent step** is performed: all weights are updated using the calculated gradient.

The approximation theorem



On the Approximate Realization of Continuous Mappings by Neural Networks

KEN-ICHI FUNAHASHI

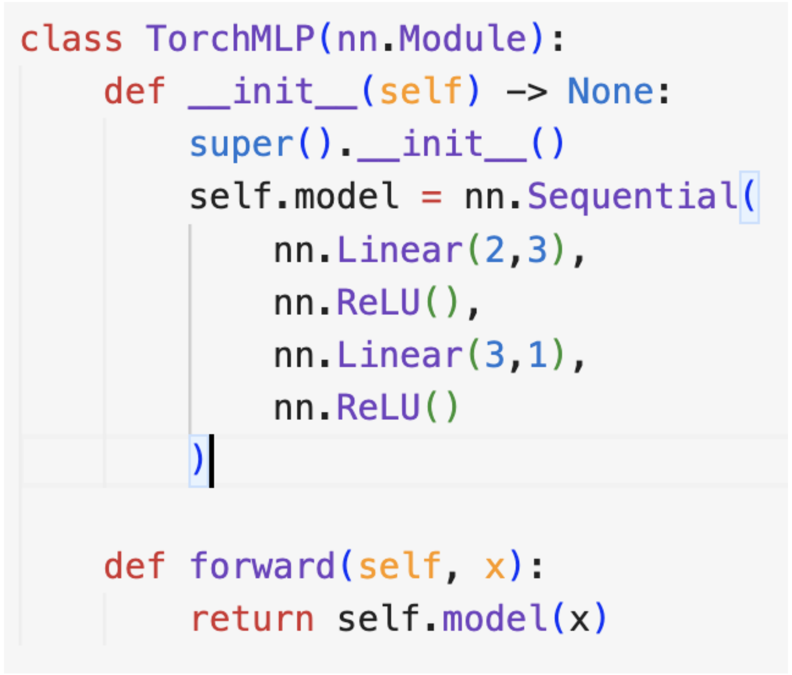
ATR Auditory and Visual Perception Research Laboratories

(*Received 6 May* 1988; *revised and accepted* 14 *September* 1988)

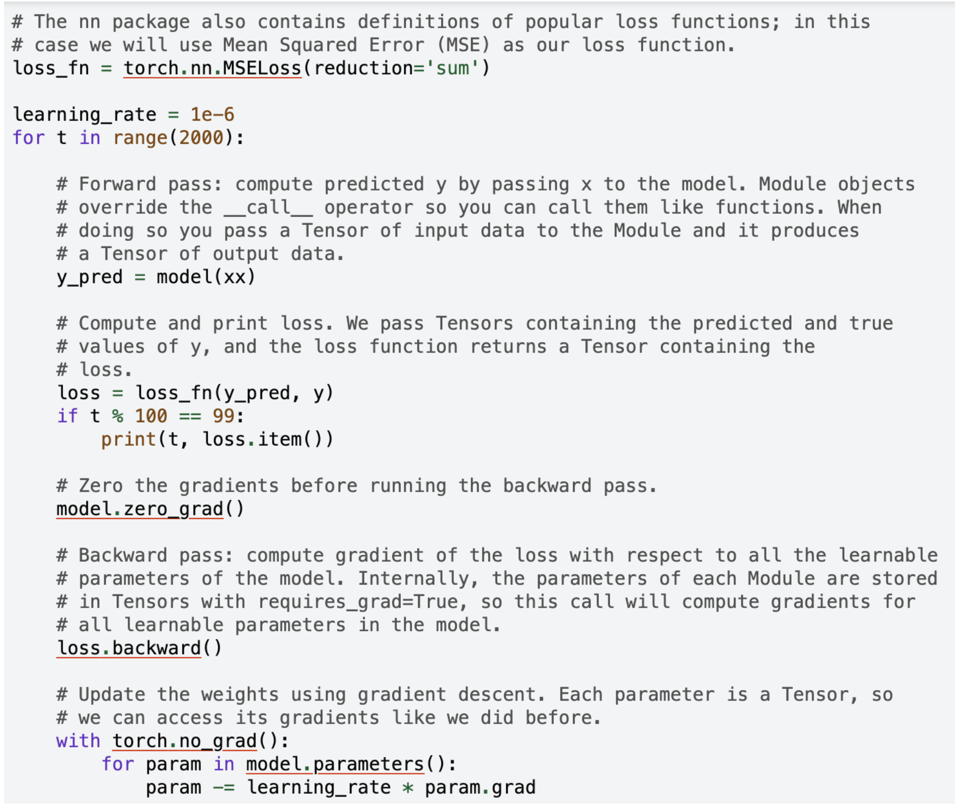
Abstract—*In this paper, we prove that any continuous mapping can be approximately realized by Rumelhart-Hinton-Williams' multilayer neural networks with at least one hidden layer whose output functions are sigmoid functions. The starting point of the proof for the one hidden layer case is an integral formula recently proposed by Irie–Miyake and from this, the general case (for any number of hidden layers) can be proved by induction. The two hidden layers case is proved also by using the Kolmogorov-Arnold-Sprecher theorem and this proof also gives non-trivial realizations.*

Keywords—Neural network, Back propagation, Output function, Sigmoid function, Hidden layer, Unit, Realization, Continuous mapping.

Approximating analytical functions

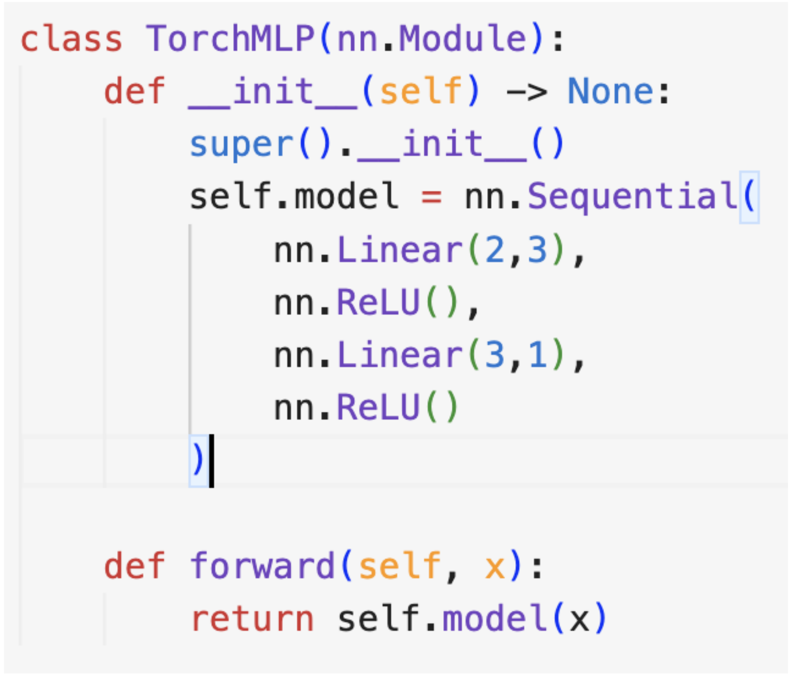


```python
class TorchMLP(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2,3),
            nn.ReLU(),
            nn.Linear(3,1),
            nn.ReLU()
        )

    def forward(self, x):
        return self.model(x)
```

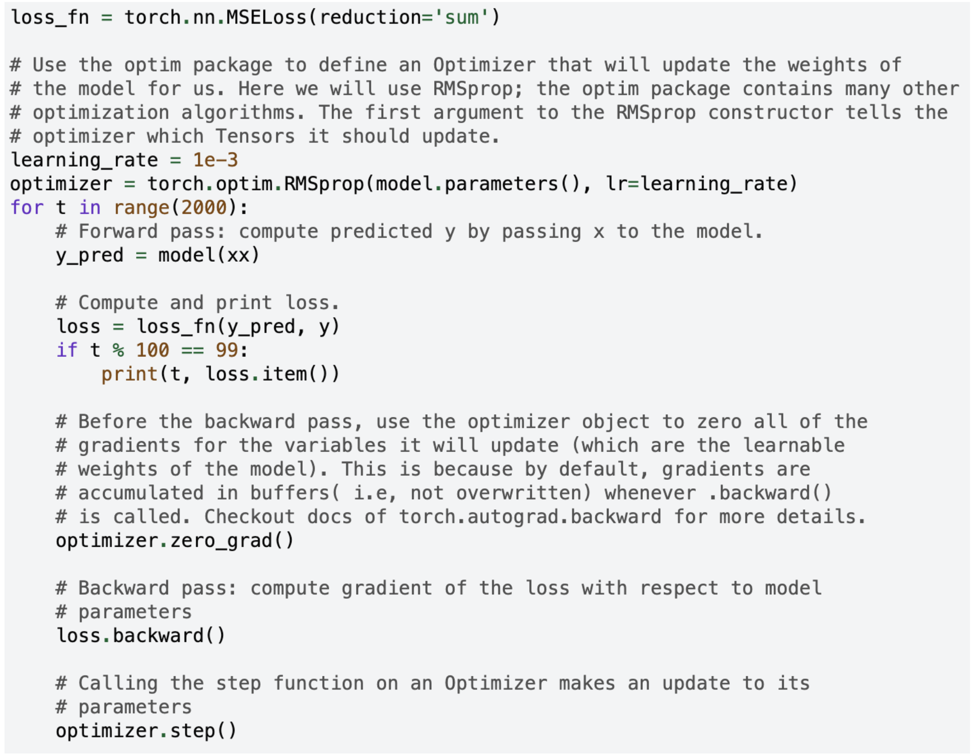


```python
# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):

    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(xx)

    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the
    # loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Tensors with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

Approximating analytical functions



```python
class TorchMLP(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2,3),
            nn.ReLU(),
            nn.Linear(3,1),
            nn.ReLU()
        )

    def forward(self, x):
        return self.model(x)
```



```python
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use RMSprop; the optim package contains many other
# optimization algorithms. The first argument to the RMSprop constructor tells the
# optimizer which Tensors it should update.
learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
for t in range(2000):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

Exercises

1. Produce a neural network to fit the function:

$$f(x,y) = 20 + (x^2 - 10\cos(2\pi x)) + (y^2 - 10\cos(2\pi y))$$

Domain: Usually evaluated in the square $x, y \in [-5.12, 5.12]$.

How does this work outside the training interval?

2. Produce a neural network to fit XOR (how many layers did you need?)
3. Produce a neural network to predict housing prices.

House Price Regression Dataset