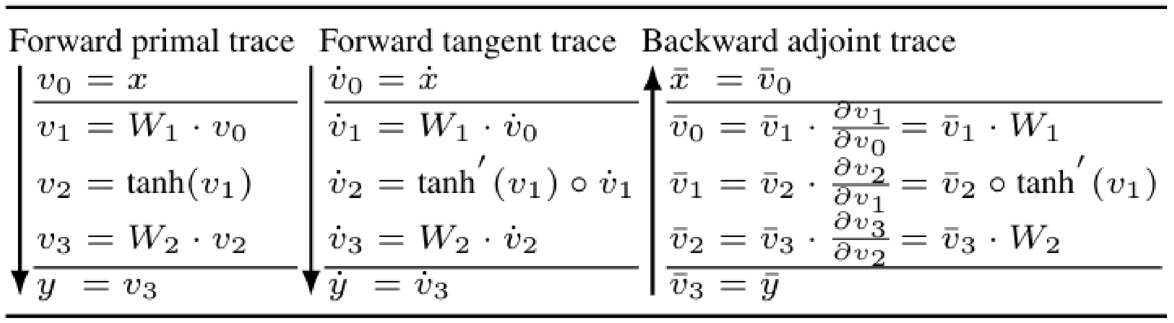
Forward Automatic Differentiation

In **forward-mode automatic differentiation**, we propagate derivatives alongside function evaluations (see figure below).

Table 1: An example of forward and reverse-mode AD in a two-layers tanh MLP. Here v_0 denotes the input variable, v_k the primals, $\dot{v}_k$ the tangents, $\bar{v}_k$ the adjoints, and W_1, W_2 the weight matrices. Biases are omitted for brevity.



Forward primal trace	Forward tangent trace	Backward adjoint trace
$v_0 = x$	$\dot{v}_0 = \dot{x}$	$\bar{x} = \bar{v}_0$
$v_1 = W_1 \cdot v_0$	$\dot{v}_1 = W_1 \cdot \dot{v}_0$	$\bar{v}_0 = \bar{v}_1 \cdot \frac{\partial v_1}{\partial v_0} = \bar{v}_1 \cdot W_1$
$v_2 = \tanh(v_1)$	$\dot{v}_2 = \tanh^{'}(v_1) \circ \dot{v}_1$	$\bar{v}_1 = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = \bar{v}_2 \circ \tanh^{'}(v_1)$
$v_3 = W_2 \cdot v_2$	$\dot{v}_3 = W_2 \cdot \dot{v}_2$	$\bar{v}_2 = \bar{v}_3 \cdot \frac{\partial v_3}{\partial v_2} = \bar{v}_3 \cdot W_2$
$y = v_3$	$\dot{y} = \dot{v}_3$	$\bar{v}_3 = \bar{y}$

That is, if we have a function $f(x)$, we want both

$$f(x) \quad \text{and} \quad f'(x).$$

The algorithm this differentiation uses is not **numerical differentiation** nor **symbolic differentiation.** This approch rely on the dual space. Therefore, in order to illustrate this algorithms, we need to define what dual number and dual vectors are.

Dual Space and Dual Vector

If V is a vector space (say over $\mathbb{R}$), then the **dual space** V^* is defined as:

$$V^* = \{w^* : V \to \mathbb{R} \mid w^* \text{ is linear}\}.$$

That is:

Every element w^* of V^* is a **linear map** that takes a vector $v \in V$ and produces a **scalar**.

Example:

The dot product in the space $\mathbb{R}^n$ is a linear functional that map a vector in $\mathbb{R}^n$ to $\mathbb{R}$ (scalar field)

Dual Numbers

A **dual number** is defined as:

$$x + \epsilon x'$$

where:

- x is the real (value) part,
- x' is the “infinitesimal” part, representing the derivative,
- and ϵ is a special symbol such that:

$$\epsilon^2 = 0, \quad \epsilon \neq 0.$$

That last rule is what makes it different from complex numbers.

Arithmetic rules of dual numbers

Because $\epsilon^2 = 0$, the addition and multiplication can be performed as,

Operation	Result
Addition	$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$

Operation	Result
Multiplication	$(a+b\epsilon)(c+d\epsilon) = ac + (ad+bc)\epsilon \quad (\text{since } \epsilon^2 = 0)$

Forward AD for Uni-Variant Function

Forward-mode AD using **dual numbers** is based on the idea:

$$\text{Dual number: } x + \epsilon x', \quad \text{where } \epsilon^2 = 0$$

If you apply a function f to this dual number:

$$f(x + \epsilon x') = f(x) + \epsilon f'(x)x'$$

Then the **value** of $f(x)$ is the real part (called the *primal*),
and the **derivative** $f'(x)x'$ is stored in the ϵ-part (called the *tangent*).

Example:

Let's say $f(x) = x^2 + 3x$.

Compute $f(x+\epsilon)$:

$$f(x+\epsilon) = (x+\epsilon)^2 + 3(x+\epsilon) = x^2 + 2x\epsilon + 3x + 3\epsilon = (x^2+3x) + (2x+3)\epsilon$$

So:

- The real part = $f(x)$
- The coefficient of ϵ = $f'(x)$.

So, every variable in the computation becomes a dual number $x + \epsilon$, and all operations are overloaded so that the ϵ-parts propagate derivatives automatically.

In [2]:
```
# Here we will implement this approach in PyTorch.
import torch

# Assuming  that we have an input point x = 3, and we want to find the derivative of f at x using the computational graph.
x_val = torch.tensor(3.0)

# Create a dual number (value + infinitesimal) via forward-mode AD
with torch.autograd.forward_ad.dual_level():
    x = torch.autograd.forward_ad.make_dual(x_val, torch.tensor(1.0))  # 1.0 -> dx/dx = 1, so we define x as, x + ε..
    f = x**2 + 3 * x
    # Extract the primal (real) and tangent (ε-part) components
    f_value , f_derivative  = torch.autograd.forward_ad.unpack_dual(f)

print(f"f(x) = {f_value.item():.2f}")
print(f"f'(x) = {f_derivative.item():.2f}")
```

```
f(x) = 18.00
f'(x) = 9.00
```

As you see, the result is not symbolic as the derivative was not found symbolically neither numerically. This algorithm is found to be efficient for two reasons, it does not use numerical approximation, and avoid complexity of finding the symbolic differentiation

---

Forward AD for multi-variant Funtion

Forward AD for multi-variant functions is also implemented via dual numbers, the difference is here what it computes for a given direction v is exactly the **Jacobian–vector product (JVP)**.

Formally, for a given $f(\mathbf{x})$ and direction v,

$$\text{Forward AD gives:} \quad (f(\mathbf{x}), \mathbf{J_f}(\mathbf{x}) \cdot \mathbf{v})$$

Similarly, this arithmetic computes **both** the function output and its directional derivative at once.

Dual numbers represent the JVP computation

When we evaluate f on a **dual number vector**

$$x + \epsilon v$$

(where v is a direction vector and $\epsilon^2 = 0$),
we get:

$$f(\mathbf{x} + \epsilon\mathbf{v}) = \mathbf{f}(\mathbf{x}) + \epsilon(\mathbf{J_f}(\mathbf{x}) \cdot \mathbf{v})$$

So:

- The **real part** = $f(\mathbf{x})$
- The ϵ **part** = $J_f(\mathbf{x}) \cdot \mathbf{v}$

Example:

Let $f(x, y) = (xy, x + y)$.

Before we apply the dual arithmetic to implement the forward AD for this function, we will find the Jacobian matrix symbolically.

For this function, the Jacobian matrix is

$$J_f(x, y) = \begin{bmatrix} \frac{\partial}{\partial x}(xy) & \frac{\partial}{\partial y}(xy) \\ \frac{\partial}{\partial x}(x + y) & \frac{\partial}{\partial y}(x + y) \end{bmatrix} \begin{bmatrix} y & x \\ 1 & 1 \end{bmatrix}.$$

Now, for this Jacobian matrix we will implment JVP assuming that the direction is (No dual arithmetic been used),

$$v = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

So,

$$J_f(x, y)v = \begin{bmatrix} y & x \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ 1 \end{bmatrix}$$

As as result, using this directional derivative we have found total partial derivative with respect to y, which the second column of $J_f(x, y)$. Also, if we compute JVP, with

$$v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Then,

$$J_f(x, y)v = \begin{bmatrix} y \\ 1 \end{bmatrix}$$

Which is the total partial derivative with respect to x.

Now, let's compute the JVP using the dual arithmetic in direction $v = (1, 2)$.

We form the dual vector:

$$(x, y) = (x + \varepsilon v_1, y + \varepsilon v_2).$$

Compute f:

$$f(x, y) = ((x + \varepsilon v_1)(y + \varepsilon v_2),\ (x + \varepsilon v_1) + (y + \varepsilon v_2)).$$

Expand and collect ε terms:

$$f(x, y) = (xy + \varepsilon(v_2 x + v_1 y),\ x + y + \varepsilon(v_1 + v_2)).$$

Thus:

$$J_f(x, y)\, v = (v_2 x + v_1 y,\ v_1 + v_2)$$

at $v = (1, 2)$ we have,

$$J_f(x, y)\, v = (2x + y, 3)$$

That ε-part is the **JVP**.

But, the actual results of this forward mode derivative is not symbolic as I represent it here, the result is numeric vector directly, and for each (x, y) the derivative is been computed automatically along the computational graph.

To sum up, the vector v determine the direction of the derivative. That is, if I want partial derivative of the function I will choose this vector as,

Direction v	Meaning	JVP result	Interpretation
$v = (1, 0)$	perturb x, keep y fixed	$(y, 1)$	derivative w.r.t x
$v = (0, 1)$	perturb y, keep x fixed	$(x, 1)$	derivative w.r.t y
$v = (v_1, v_2)$	perturb both	$(v_1 y + v_2 x, v_1 + v_2)$	directional derivative along (v_1, v_2)

In [4]:
```
# Here we will implement this approach in PyTorch.
import torch

# Assuming  that we have an input point (2,5), and we want to find the derivative of f wrt x using the computational graph.
p1 = torch.tensor([2.0,5.0])
def f(x):
    return torch.stack([
        x[0] * x[1],   # f1 = x * y
        x[0] + x[1]    # f2 = x + y
    ])
# Create a dual number (value + infinitesimal) via forward-mode AD
with torch.autograd.forward_ad.dual_level():
    x = torch.autograd.forward_ad.make_dual(p1, torch.tensor([1.0,0.0]))  # p1 = (x+εv, y+εv), v is with x- direction
    evaluation = f(x)
    f_value , f_derivative  = torch.autograd.forward_ad.unpack_dual(evaluation)

print("f(x, y) =", f_value)
print("df/dx =", f_derivative)
```

```
f(x, y) = tensor([10.,  7.])
df/dx = tensor([5., 1.])
```

The results of the both examples we presented are obtained using the built in functionality in PyTorch. It maybe useful to see how we can rebuild this functionality from scratch using Algebraic concepts of vector spaces.

To construct the **dual number space**, we apply the following theorem to our set $\mathbb{S}$, which is an extension of the real numbers $\mathbb{R}$.
Such that, for every $x \in \mathbb{R}$, we define an element of $\mathbb{S}$ as $x + \epsilon x'$, where ϵ is an infinitesimal such that $\epsilon^2 = 0$.

Theorem:
A set V is a **vector space** if it is closed under vector addition and scalar multiplication.

Before proceeding to implement the `Dual` class in code, we first need to establish the **algebraic arithmetic** of dual numbers.
In the previous table, we have already defined the rules for **addition** and **multiplication**.
What remains is to define the rules for **subtraction** and **division**.

Since subtraction directly follows the same structure as addition, we will focus here on deriving the **division rule** for dual numbers.

Derivation of the Division Rule for Dual Numbers

We want to compute the quotient of two dual numbers:

$$\frac{a + \varepsilon a'}{b + \varepsilon b'}, \quad \text{where } \varepsilon^2 = 0.$$

We start by writing the reciprocal as:

$$\frac{1}{b + \varepsilon b'} = \frac{1}{b} \cdot \frac{1}{1 + \varepsilon \frac{b'}{b}}.$$

So, we can use the Taylor expansion for $\frac{1}{1+\varepsilon \frac{b'}{b}}$. The Taylor expansion is,

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + \dots$$

Let $x = \varepsilon \frac{b'}{b}$, then all higher-order terms vanish because $\varepsilon^2 = 0$. Thus, the first-order expansion is **exact**, not approximate:

$$\frac{1}{1+\varepsilon\frac{b'}{b}} = 1 - \varepsilon\frac{b'}{b}.$$

Now, ubstituting this into the reciprocal expression gives:

$$\frac{1}{b+\varepsilon b'} = \frac{1}{b}\left(1-\varepsilon\frac{b'}{b}\right) = \frac{1}{b} - \varepsilon\frac{b'}{b^2}.$$

Now, we multiply this by the numerator $a+\varepsilon a'$:

$$(a+\varepsilon a')\left(\frac{1}{b} - \varepsilon\frac{b'}{b^2}\right) = \frac{a}{b} + \varepsilon\left(\frac{a'}{b} - \frac{ab'}{b^2}\right).$$

Finally, we simplify the result by combining the real and infinitesimal parts to get:

$$\frac{a+\varepsilon a'}{b+\varepsilon b'} = \frac{a}{b} + \varepsilon\frac{a'b-ab'}{b^2}.$$

Connection to the quotient rule in calculus

The coefficient of ε,

$$\frac{a'b-ab'}{b^2},$$

is exactly the **quotient rule** from differentiation:

$$\left(\frac{a}{b}\right)' = \frac{a'b-ab'}{b^2}.$$

This shows that **dual number algebra inherently encodes the rules of differentiation**.
Hence, when we perform division on dual numbers, the tangent part automatically follows the derivative of a quotient.

In [18]:
```
import torch
```

```python
# We’ll use double precision for numerical accuracy
dtype = torch.float64

# 1. Define a Dual number class
class Dual:
    def __init__(self, val, tan=None):
        # val: torch tensor (the actual value)
        # tan: torch tensor (the derivative / tangent part)
        self.val = torch.as_tensor(val, dtype=dtype) # This is how we define attribute for object
        if tan is None:
            self.tan = torch.zeros_like(self.val) # only real part (val)
        else:
            self.tan = torch.as_tensor(tan, dtype=dtype)

    def __repr__(self):
        return f"{self.val} + ε {self.tan}" # Automatically represent our object in readable form


    # Define arithmetic operations using dual number algebra

    def __add__(self, other):
        other = other if isinstance(other, Dual) else Dual(other)
        return Dual(self.val + other.val, self.tan + other.tan)
    __radd__ = __add__ # Since addition is commutitive

    def __sub__(self, other):
        other = other if isinstance(other, Dual) else Dual(other)
        return Dual(self.val - other.val, self.tan - other.tan)
    def __rsub__(self, other):
        other = other if isinstance(other, Dual) else Dual(other)
        return other - self # The regualr subtraction now already defined.

    def __mul__(self, other):
        other = other if isinstance(other, Dual) else Dual(other)
        # (a + εa')(b + εb') = ab + ε(ab' + a'b)
        val = self.val * other.val
        tan = self.val * other.tan + self.tan * other.val
        return Dual(val, tan)
    __rmul__ = __mul__ # Since multiplication is commutitive

    def __truediv__(self, other):
        other = other if isinstance(other, Dual) else Dual(other)
        if torch.all(other.val == 0):
            raise ZeroDivisionError("Only nonzero denomerator supported.")
        val = self.val / other.val
        tan = (self.tan * other.val - self.val * other.tan) / (other.val ** 2)
        return Dual(val, tan)

    def __rtruediv__(self, other):
        other = other if isinstance(other, Dual) else Dual(other)
        return other / self

    def __pow__(self, p):
        # Only scalar powers
        if isinstance(p, (int, float)):
            val = self.val ** p
            tan = p * (self.val ** (p - 1)) * self.tan
            return Dual(val, tan)
        else:
            raise NotImplementedError("Only scalar powers supported.")

    # Accessors helps if I want to call single attribute of my object.
    def value(self):
        return self.val

    def tangent(self):
        return self.tan
```

In [24]:
```python
p1 = Dual(1,2)
p2 = Dual(2,3)
p3 = Dual(0,2)
```

In [29]:
```python
print(p1.val.item())
print(p1.tan)
print(p1)
print((p1 + p2))
print(p1*p2)
print(p1 - p2)
print(p3 / p1)
```

```
1.0
tensor(2., dtype=torch.float64)
1.0 + ε 2.0
3.0 + ε 5.0
2.0 + ε 7.0
-1.0 + ε -1.0
0.0 + ε 2.0
```

In [ ]:
```
print(p1 / p3)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[25], line 1
----> 1 print(p1/p3)

Cell In[18], line 46, in Dual.__truediv__(self, other)
     44 other = other if isinstance(other, Dual) else Dual(other)
     45 if torch.all(other.val == 0):
---> 46     raise ZeroDivisionError("Only nonzero denomerator supported.")
     47 val = self.val / other.val
     48 tan = (self.tan * other.val - self.val * other.tan) / (other.val ** 2)

ZeroDivisionError: Only nonzero denomerator supported.
```

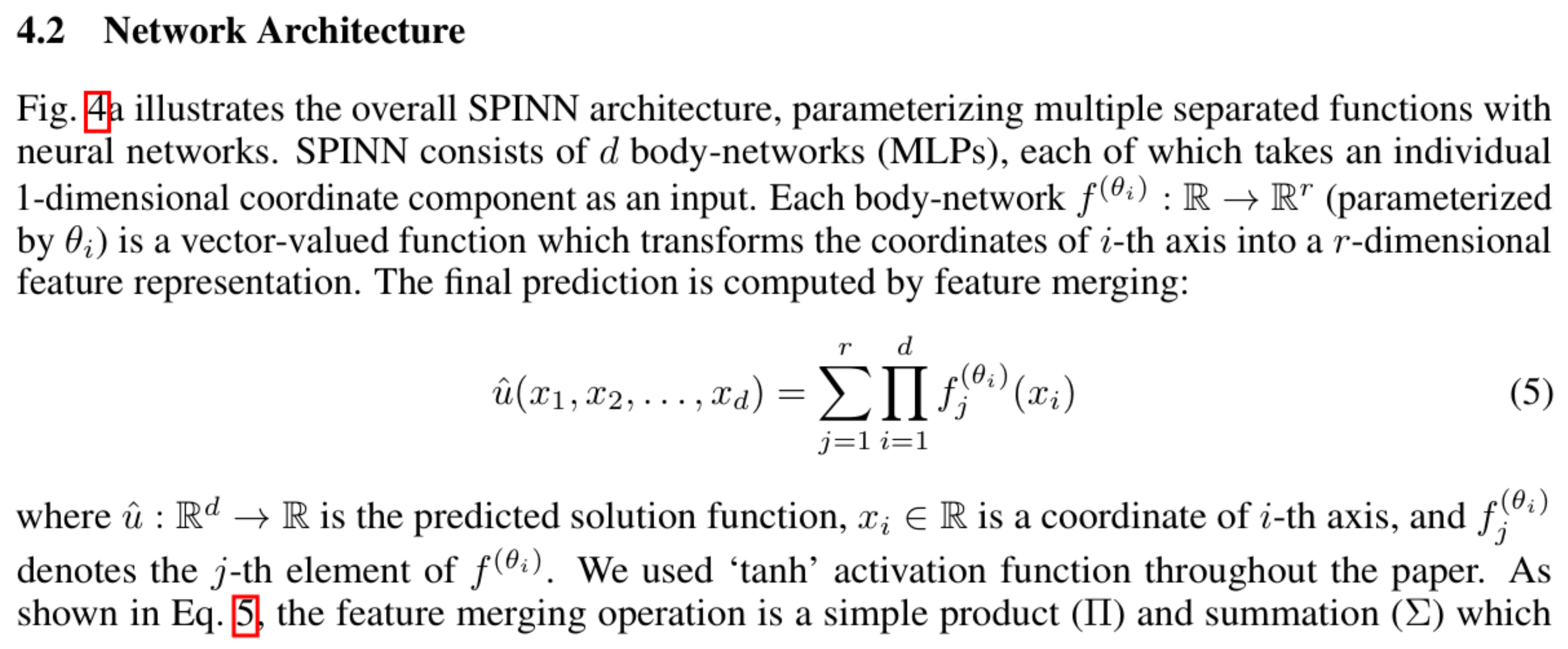
Implment our Uni-Variant Example Using This Class

In [ ]:
```
x0 = torch.tensor(3.0, dtype=dtype)

# Create dual number with tangent = 1, i.e derivative wrt x
x = Dual(x0, torch.tensor(1.0, dtype=dtype))
def f_univariate(x):
    return x**2 + 3*x
# Evaluate
f_dual = f_univariate(x) # Now it will use the arithmetic we defined in the class

print("f(x):", f_dual.val.item())
print("f'(x):", f_dual.tan.item())
```

```
f(x): 18.0
f'(x): 9.0
```

-- Examin Outer Product

First of all we will check how we merge features for single point, mentiond in the paper here.



4.2 Network Architecture

Fig. 4a illustrates the overall SPINN architecture, parameterizing multiple separated functions with neural networks. SPINN consists of d body-networks (MLPs), each of which takes an individual 1-dimensional coordinate component as an input. Each body-network $f^{(\theta_i)} : \mathbb{R} \to \mathbb{R}^r$ (parameterized by θ_i) is a vector-valued function which transforms the coordinates of i-th axis into a r-dimensional feature representation. The final prediction is computed by feature merging:

$$\hat{u}(x_1, x_2, \ldots, x_d) = \sum_{j=1}^{r} \prod_{i=1}^{d} f_j^{(\theta_i)}(x_i) \tag{5}$$

where $\hat{u} : \mathbb{R}^d \to \mathbb{R}$ is the predicted solution function, $x_i \in \mathbb{R}$ is a coordinate of i-th axis, and $f_j^{(\theta_i)}$ denotes the j-th element of $f^{(\theta_i)}$. We used 'tanh' activation function throughout the paper. As shown in Eq. 5, the feature merging operation is a simple product (Π) and summation (Σ) which

- Output of the architecture at single point $(x, t) \in \mathbb{R}^2$

Take:

- $(d = 2)$ (two coordinates: (x, t))

- ($r = 3$) (each body-network outputs a vector of length 3).

Body-network outputs

$$f^{(\theta_1)}(x) = \left(f_1^{(\theta_1)}(x), f_2^{(\theta_1)}(x), f_3^{(\theta_1)}(x)\right)$$

$$f^{(\theta_2)}(t) = \left(f_1^{(\theta_2)}(t), f_2^{(\theta_2)}(t), f_3^{(\theta_2)}(t)\right)$$

At inputs ($x = 0.5, t = 0.7$):

$$f^{(\theta_1)}(x) = (2, 1, 4), \quad f^{(\theta_2)}(t) = (3, 5, 6)$$

Feature merging (simple product across i)

For each feature index ($j = 1, 2, 3$):

$$\prod_{i=1}^{2} f_j^{(\theta_i)}(x, t)$$

- For ($j = 1: \ f_1^{(\theta_1)}(x) \cdot f_1^{(\theta_2)}(t) = 2 \cdot 3 = 6$)
- For ($j = 2: \ 1 \cdot 5 = 5$)
- For ($j = 3: \ 4 \cdot 6 = 24$)

So after the product step we have this vector of length r `rank`:

$$(6, 5, 24)$$

Finally we take the Summation over j

$$\hat{u}(x, t) = 6 + 5 + 24 = 35$$

So, for single point in $\mathbb{R}^2$ we have got corresponding single scalar function output through our SPINN architecture network. Now, what if we have more than single point to pass through the network?. Here where the outer product concept has been used.

- Output of the architecture at multiple points (3 batches) (x_1, t_1), (x_2, t_2), $(x_3, t_3) \in \mathbb{R}^2$

In practice, the input coordinates are given in a batch during training and inference. Assume that N input coordinates (training points) are sampled from each axis. Note that the sampling resolutions for each axis need not be the same. The input coordinates $X \in \mathbb{R}^{N\times d}$ is now a matrix. The batchified form of feature representation $F \in \mathbb{R}^{N\times r\times d}$ and Eq. 5 now becomes

$$\hat{U}(X_{:,1}, X_{:,2}, \ldots, X_{:,d}) = \sum_{j=1}^{r} \bigotimes_{i=1}^{d} F_{:,j,i}, \tag{6}$$

where $\hat{U} \in \mathbb{R}^{N\times N\times\ldots\times N}$ is the discretized solution tensor, $\bigotimes$ denotes outer product, $F_{:,:,i} \in \mathbb{R}^{N\times r}$ is an i-th frontal slice matrix of tensor F, and $F_{:,j,i} \in \mathbb{R}^{N}$ is the j-th column of the matrix $F_{:,:,i}$. Fig. 4b shows an illustrative procedure of Eq. 6. Due to its structural input points and outer products between feature vectors, SPINN's solution approximation can be viewed as a low-rank tensor decomposition where the feature size r is the rank of the reconstructed tensor. Among many decomposition methods, SPINN corresponds to CP-decomposition [16], which approximates a tensor by finite summation

---

Example

Consider,

- Dimension: $d = 2$, $((x, t))$
- Batch size: $(N = 3)$ (three sampled points along each axis)
- Feature size: $(r = 2)$

So the discretized solution is:

$$\hat{U} \in \mathbb{R}^{3\times 3}$$

Inputs (3 batches, that is three points in $\mathbb{R}^2$)

Sampled coordinates:

$$X \in \mathbb{R}^{N,d}$$

$$X_{:,:} = \begin{bmatrix} 0.1 & 0.2 \\ 0.5 & 0.6 \\ 0.9 & 1.0 \end{bmatrix}$$

$$X_{:,x} = (0.1, 0.5, 0.9), \quad X_{:,t} = (0.2, 0.6, 1.0)$$

Body-network outputs

The feature matix $F \in \mathbb{R}^{N,r,d}$ where $j = 1, 2, \ldots, r$, and $i = 1, 2, \ldots, d$ in our case $d = 2$ and $r = 2$ and $N = 3$ For spatial axis (x):

$$F_{:,:,x} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

For time axis (t):

$$F_{:,:,t} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \\ 4 & 5 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

Perform Outer product (per feature j)

Feature ($j = 1$):

$$F_{:,1,x} = (1, 3, 5), \quad F_{:,1,t} = (2, 0, 4)$$

Outer product:

$$F_{:,1,x} \otimes F_{:,1,t} = \begin{bmatrix} 1\cdot 2 & 1\cdot 0 & 1\cdot 4 \\ 3\cdot 2 & 3\cdot 0 & 3\cdot 4 \\ 5\cdot 2 & 5\cdot 0 & 5\cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 4 \\ 6 & 0 & 12 \\ 10 & 0 & 20 \end{bmatrix}$$

Feature ($j = 2$):

$$F_{:,2,x} = (2,4,6), \quad F_{:,2,t} = (1,3,5)$$

Outer product:

$$F_{:,2,x} \otimes F_{:,2,t} = \begin{bmatrix} 2\cdot 1 & 2\cdot 3 & 2\cdot 5 \\ 4\cdot 1 & 4\cdot 3 & 4\cdot 5 \\ 6\cdot 1 & 6\cdot 3 & 6\cdot 5 \end{bmatrix} = \begin{bmatrix} 2 & 6 & 10 \\ 4 & 12 & 20 \\ 6 & 18 & 30 \end{bmatrix}$$

Perform the Summation over j

$$\hat{U} = \begin{bmatrix} 2 & 0 & 4 \\ 6 & 0 & 12 \\ 10 & 0 & 20 \end{bmatrix} + \begin{bmatrix} 2 & 6 & 10 \\ 4 & 12 & 20 \\ 6 & 18 & 30 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 14 \\ 10 & 12 & 32 \\ 16 & 18 & 50 \end{bmatrix}$$

Final Result The predicted discretized solution is:

$$\hat{U}(x,t) = \begin{bmatrix} 4 & 6 & 14 \\ 10 & 12 & 32 \\ 16 & 18 & 50 \end{bmatrix} \in \mathbb{R}^{3\times 3}$$

In [1]:
```python
import numpy as np
# apply the outer product using the library NumPy
netX = np.array([[1,2],[3,4],[5,6]])
netT = np.array([[2,1],[0,3],[4,5]])
N,r=3,2
output = np.zeros((N,N))
for j in range(2):
   Fx=netX[:,j]
   Ft=netT[:,j]
   product = np.outer(Fx,Ft)
```

```
    output += product
print(output)
```

```
[[ 4.  6. 14.]
 [10. 12. 32.]
 [16. 18. 50.]]
```

In [3]:
```
import torch
# PyTorch implementaion
# apply the outer product using the method einsum for CP decomposition (this is more efficient due to avoiding massive looping
netX = torch.tensor([[1,2],[3,4],[5,6]])
netT = torch.tensor([[2,1],[0,3],[4,5]])
N,r=3,2
output = torch.einsum('nj,mj->nm',netX,netT)
print(output)
```

```
tensor([[ 4,  6, 14],
        [10, 12, 32],
        [16, 18, 50]])
```

Refrences (Outer Product):

- The Paper of Separable Physics Informe Neural Networks
- Here you find details about tensor decompositions, CP decomposition
- Tensor Decompositions and Applications

Refrences (Forward AD):

- The Paper of Separable Physics Informe Neural Networks
- A Hitchhiker's Guide to Automatic Differentiation
- PyTorch document Forward-mode Automatic Differentiation
- MIT University Lecture Notes: Forward and Reverse-Mode Automatic Differentiation