

Deeper Introduction to PINNs

Starting with Physics informed Machine Learning

- What are we modelling

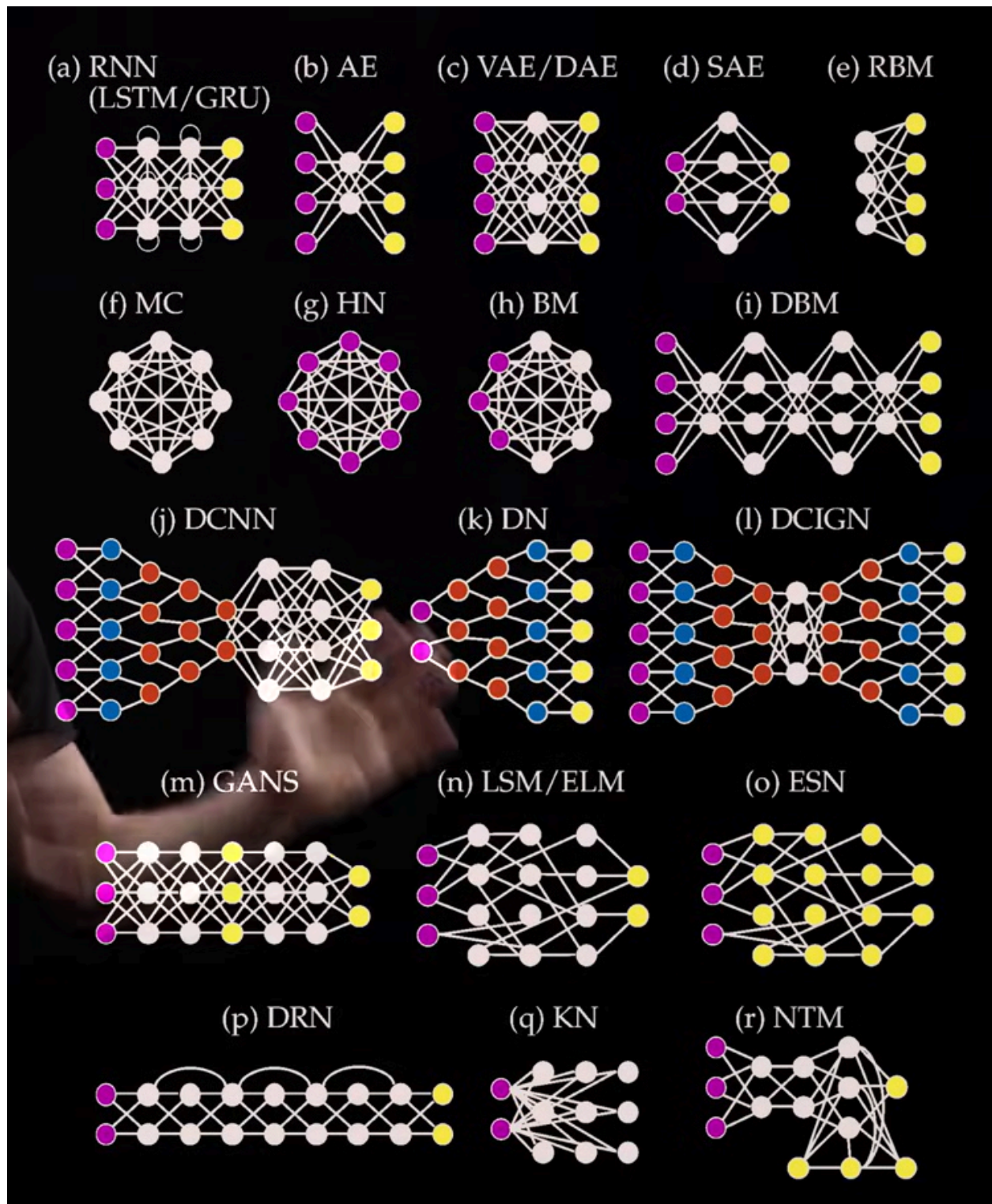
- What data will inform the model - labels, source

- Architecture - Any NN, Model

- Crafting loss fn

- Employ optimization algo - some epoch like gradient descent, ADAM -
hyperparameter tuning, using language multipliers, constrained least square

Zoo of NNs



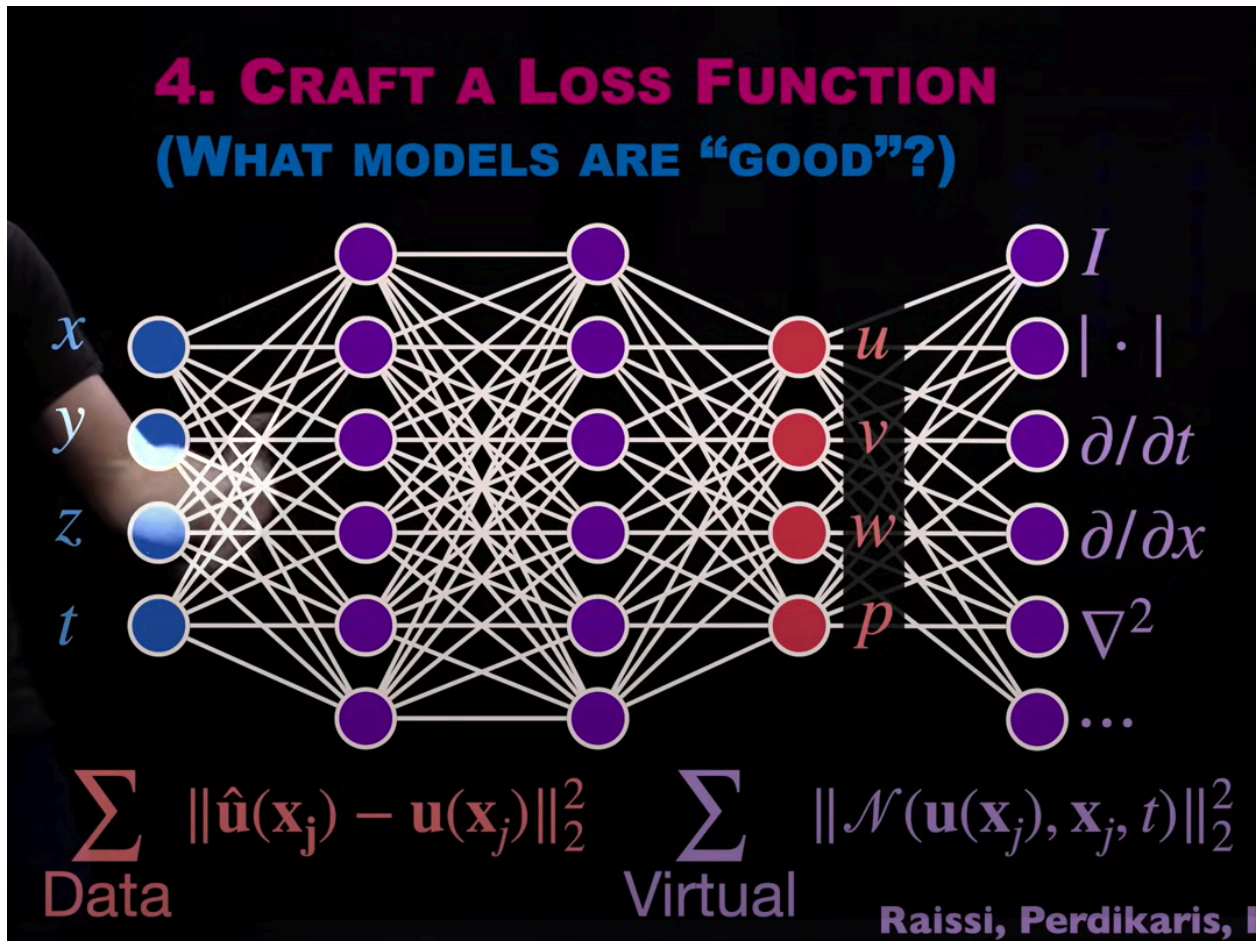
If your system has some invariance or symmetry, people just use it to make a bigger dataset

Inductive Bias - Using physics early helping in the downstream

The Digital Twin

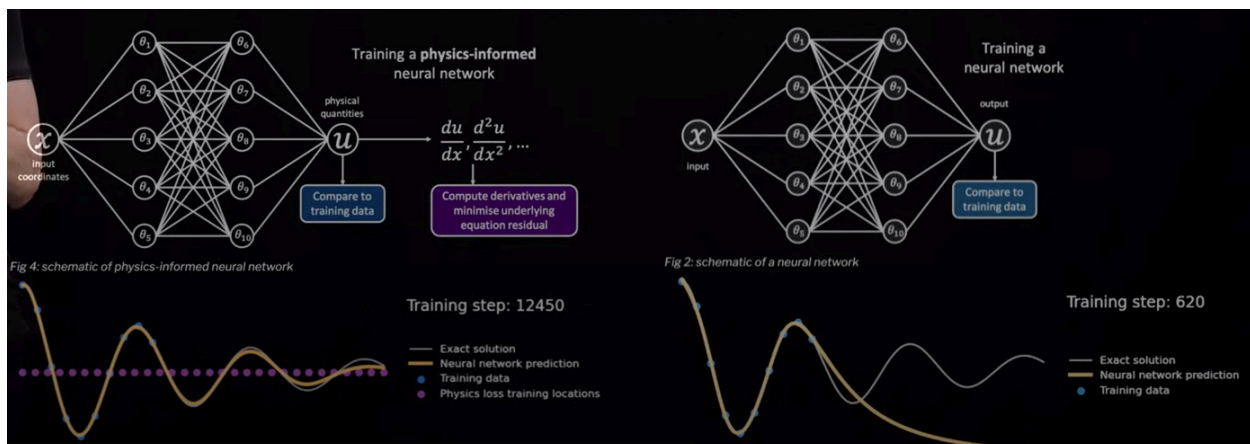
PINNs

PINNs Loss Function



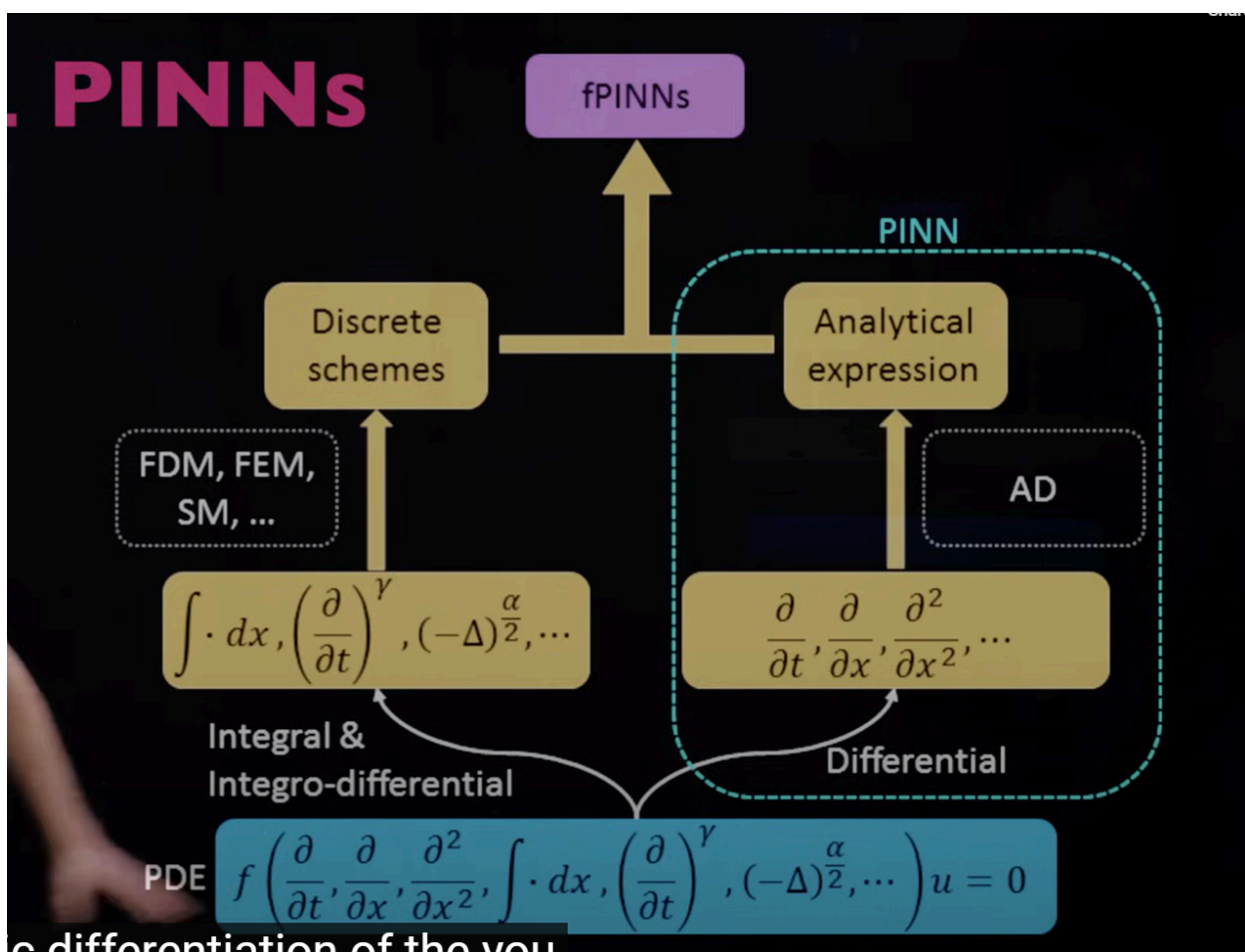
These loss functions are usually something which is zero, eg Divergence of vector field being zero

There can be something known as lambda, which is the like managing both the data and virtual as they will constantly be fighting over each other



Difference between PINNs and Naive NN

fPINNs - Fractional PINNs when you have both PDE as well ODE, having powers in integrals or differentials and thus you divide into two categories



Start the lambda with zero and slowly increase to optimal

How to implement PINNs in Python

1. DXDE recc but pytorch can be used
2. If you have a ODE well and good, just use Loss ODE = 0, if there are residuals they are the loss, same way also take the boundary conditions as loss, and square them up
3. tf.GradientTape is used for automatic Derivate
4. tensorflow is used neural network building
5. Building a Neural Network

```
def create_model():
    model = {
        'dense1': tf.keras.layers.Dense(50, activation='tanh'),
        'dense2': tf.keras.layers.Dense(50, activation='tanh'),
        'dense3': tf.keras.layers.Dense(50, activation='tanh'),
        'output_layer': tf.keras.layers.Dense(1)
    }
    return model

# call_model() - This function defines the forward pass of the neural network.
#               It takes as input a dictionary model (created by create_model()) and an input tensor x.
def call_model(model, x):
    x = model['dense1'](x)
    x = model['dense2'](x)
    x = model['dense3'](x)
    x = model['output_layer'](x)
    return x

# model = create_model()
# print(model)
```

6. Building the PDE loss and Boundary condition loss functions

```

# Define the differential equation using tf.GradientTape
def pde(x, model):
    with tf.GradientTape(persistent=True) as tape:
        tape.watch(x)
        y_pred = call_model(model, x)
        y_x = tape.gradient(y_pred, x)
        y_xx = tape.gradient(y_x, x)
    del tape
    return y_xx + np.pi**2 * tf.sin(np.pi * x)

```

```

# Define the loss function
def loss(model, x, x_bc, y_bc):
    res = pde(x, model)
    # Compute the mean squared error of the boundary conditions
    loss_pde = tf.reduce_mean(tf.square(res))
    y_bc_pred = call_model(model, x_bc)
    # Compute the mean squared error of the boundary conditions
    loss_bc = tf.reduce_mean(tf.square(y_bc - y_bc_pred))
    return loss_pde + loss_bc

```

7.