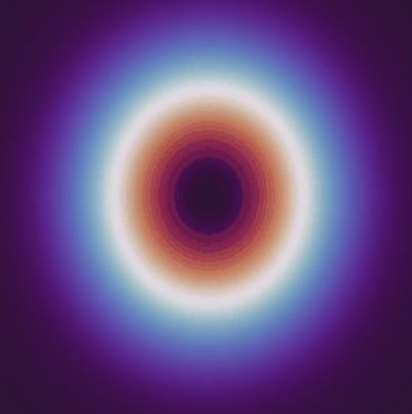
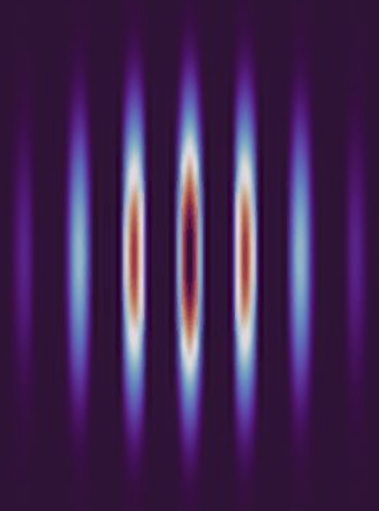
Predicting the Time Evolution of Wave Functions with U-Nets





Tyler Steffen
ECE-5995: Applied Machine Learning

Problem & Significance

$$\Psi(t+\delta t) = \Psi(t) + \delta t \frac{d}{dt}\Psi(t)$$

Goal: δt: small timestep
$\Psi(\vec{r}_n, t)$: n-dimensional wave function

$$\frac{\partial}{\partial t}\Psi(\vec{r},t) = \frac{1}{\hbar}V(\vec{r},t)\Psi(\vec{r},t) + \frac{\hbar}{2m}\Delta\Psi(\vec{r},t)$$

- Approximate the next step of the wave function with a neural network in the hopes of achieving a good estimate of physics at a much lower computational cost

Classically computing wave function evolution, even with high end modern hardware is very expensive and time consuming. Neural networks serve the purpose of function estimators. The function being estimated in this case is the time dependent Schrödinger equation, a linear partial differential equation.

```python
def update(self,
           dt: float,
           delta: float,
           device: torch.device) -> Field:
    """
    Update the field for an n-dimensional space.

    :param: self: The input field as an n-dimensional torch tensor.
    :param: device: The torch device on which to perform the calculations.
    :param: delta: The spacing between points in the field.
    :param: dt: Time step for the update.
    :return: The updated field which contains San n-dimensional torch tensor.
    """

    # Initialize potential
    V = torch.zeros_like(self.tensor)

    # Calculate laplacian
    laplaceField: torch.Tensor = Laplacian(field=self.tensor,
                                           delta=delta)

    # Iterate according to the time dependent Schrödinger equation
    self.tensor.add_(-1j * dt * (-0.5 * laplaceField + V))

    # Apply boundary conditions for n-dimensions
    for dim in range(self.tensor.dim()):
        # Set the first and last index along each dimension to 0
        self.tensor.index_fill_(dim,
                                torch.tensor([0, self.tensor.size(dim) - 1],
                                             device=device),
                                0)
    return self
```

```python
def forward(self, x):
    x1 = self.inc(x)
    x2 = self.down1(x1)
    x3 = self.down2(x2)
    x4 = self.down3(x3)
    x5 = self.down4(x4)
    x = self.up1(x5, x4)
    x = self.up2(x, x3)
    x = self.up3(x, x2)
    x = self.up4(x, x1)
    logits = self.outc(x)
    return logits
```

Existing Work

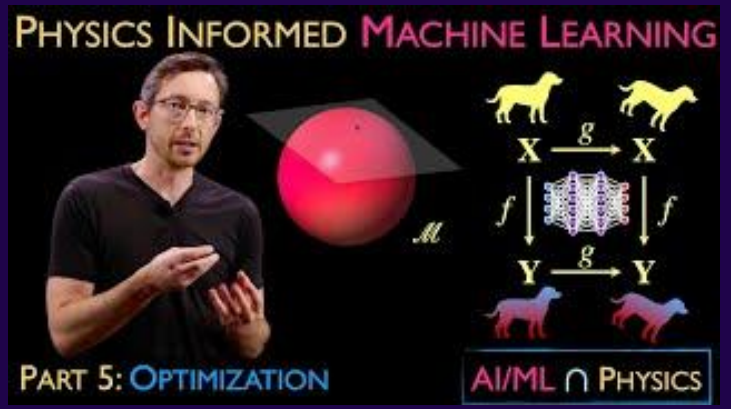
Physics informed machine learning is a massive and quickly growing field.

A large source of information on the subject has been from University of Washington professor Steve Brunton who is releasing an ongoing series on it.

He and his colleagues are also the source of several articles which use much more advanced techniques then I was able to implement, but were still very insightful.

Machine Learning for Fluid Mechanics
Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos

Data-driven discovery of partial differential equations
Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz



Approach

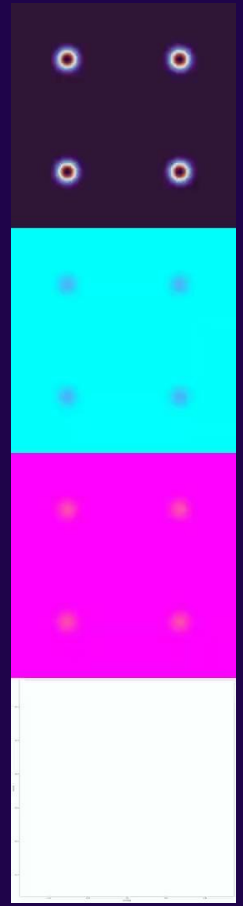
The process started with the development of the the simulation engine Psithon, which is still a work in progress. This is a PyTorch based package built to easily set up and parallelize quantum simulations on wide variety of hardware.

I have just recently made the github page public, but development is not complete.

For my first major training set I ran 10 simulations with 10-25 randomly initialized particles. Emphasizing the need to reduce computational intensity, these simulations took on average 3 hours each to complete consuming roughly 220W on an NVIDIA RTX 4090.

The simulations for this set were ran at a resolution of 1000 x 1000

30 million individual timesteps were computed to maintain computational stability. Only every one in 50,000 frames is rendered to the output field, yielding a total of 600 frames per simulation which are saved to a single hdf5 file.



$$\text{compute}\left[\Delta\Psi(\vec{r},t) = \sum_{i=1}^{n} \frac{\partial^2\Psi}{\partial r_i^2}\right] \forall\ \langle r_1, r_2, \ldots r_n\rangle \in \mathbb{Z}^n \mid 0 \leq r_i \leq \text{resolution}\ \forall\ i \in \mathbb{Z} \mid 0 \leq i \leq n$$

Approach continued

A custom dataset implementation was designed to facilitate training for this project:

```python
class QuantumFieldDataset(Dataset):
    def __init__(self, simulations: list[Simulation], fieldName: str):
        self.simulations = simulations
        self.fieldName = fieldName
        self.indexMap = self._createIndexMap()

    def _createIndexMap(self):
        indexMap = []
        for simIndex, sim in enumerate(self.simulations):
            for timestepIndex in range(sim.numTimeSteps(self.fieldName) - 1):
                indexMap.append((simIndex, timestepIndex))
        return indexMap

    def __len__(self):
        return len(self.indexMap)

    def __getitem__(self, idx):
        simIndex, timestepIndex = self.indexMap[idx]
        current_step = self.simulations[simIndex].fields[self.fieldName][timestepIndex]
        next_step = self.simulations[simIndex].fields[self.fieldName][timestepIndex + 1]

        # Processing steps remain unchanged
        current_step_real = current_step.real.unsqueeze(0)
        current_step_imag = current_step.imag.unsqueeze(0)
        next_step_real = next_step.real.unsqueeze(0)
        next_step_imag = next_step.imag.unsqueeze(0)

        current_step = torch.cat((current_step_real, current_step_imag), dim=0)
        next_step = torch.cat((next_step_real, next_step_imag), dim=0)

        return current_step.float(), next_step.float()
```

It allows multiple datasets to be loaded but still ensures that the next step comes from the same simulation, such that the training data is actually coherent

```python
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)
```

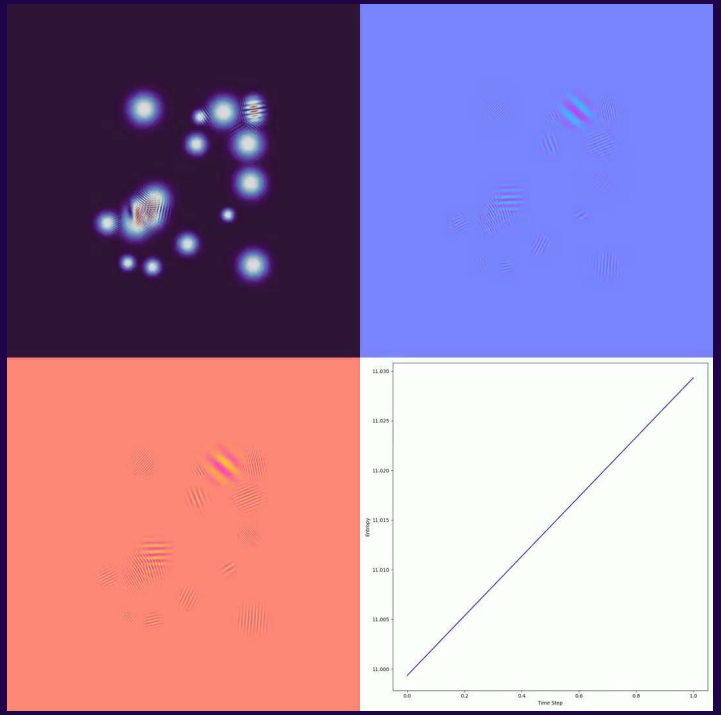
The architecture is a somewhat large, but very structurally standard U-Net

I used the standard torch implementation of MSELoss which will just take the error between the network approximation and actual next frame as the cost function.

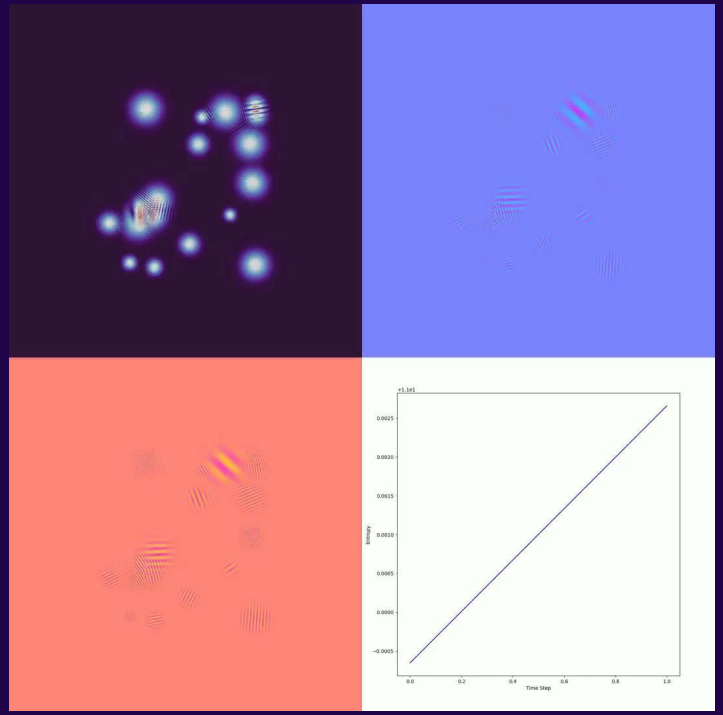
```python
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(unet_model.parameters(), lr=0.001)


def train_model(model, dataloader, optimizer, criterion, epochs=1):
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

        # Record the start time of the epoch
        epoch_start_time = time.time()

        for i, (current_step, next_step) in enumerate(dataloader):
            current_step = current_step.to(device)
            next_step = next_step.to(device)

            optimizer.zero_grad()
            output = model(current_step)
            loss = criterion(output, next_step)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        # Calculate and print the epoch duration
        epoch_duration = time.time() - epoch_start_time
        print(f'Epoch {epoch + 1} completed in {epoch_duration:.2f} seconds')

        # Saving a checkpoint with more than just the model state_dict
        checkpoint = {
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            # Add more components as needed
        }
        torch.save(checkpoint, "./unet_checkpoint.pth")
        print(f'Epoch {epoch+1}, Loss: {running_loss / len(dataloader)}')


train_model(unet_model, dataloader, optimizer, criterion, epochs=20)
```

Training costs roughly 530W between the CPU and GPU for $\geqslant 70$ seconds per dataset per epoch

Results/Discussion



Classical Method
t ~ 3 hours @ 220W

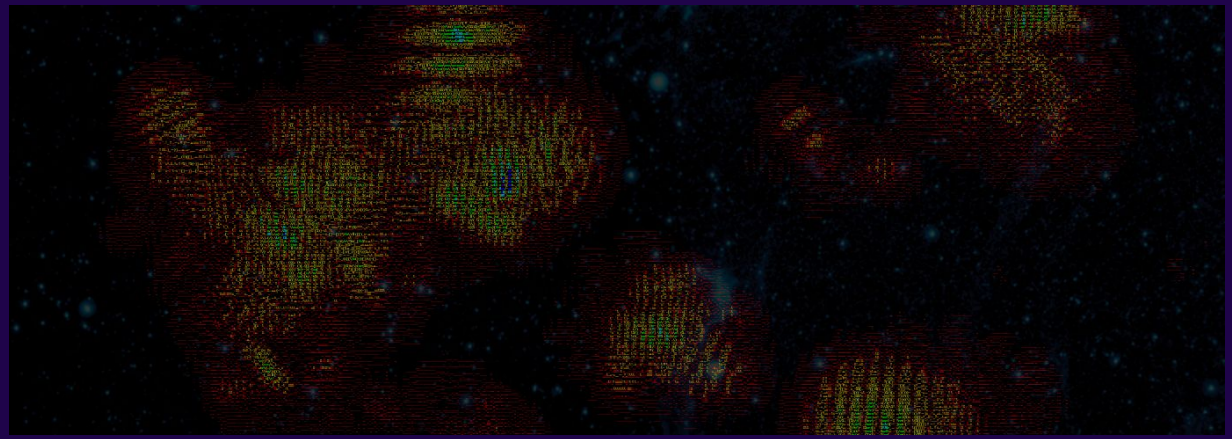


U-Net Simulation
t < 25 seconds @ 315W

Results/Discussion continued

The entropy of the network simulation behaves similarly to what is expected. This is an especially nice result considering it was not incorporated in the loss function of the current model

The U-Net model tends to approach the end state of the system more rapidly than the actual simulation. Wave packets do not stay as coherent for as long as they should. These are undesirable results, but this is with no loss function contributions other than MSE.



Conclusion/Future

I am very satisfied with the results of this initial experiment, especially considering just how generalized the U-Net implementation being used here is.

I have a couple primary paths of investigation:

- Improved loss functions including:
  - entropy, spectral domains, energy conservation, symmetry
- Enforcing conservation and symmetry through architectural choices
  - With clever choices of custom network layers, I am confident there are ways to reinforce desired behaviors such as hermiticity at a very low cost.

Blu.Psithon

https://github.com/Naitry/Blu