Learning to Invert Heat Conduction with Scale-invariant Updates

This example can be run with PyTorch and TensorFlow. Adjust the import below to select the library.

In [1]:

```
from phi.torch.flow import *
# from phi.tf.flow import *
```

Generate batched 2D source `x` (non-observable) from 10 random rectangles

In [2]:

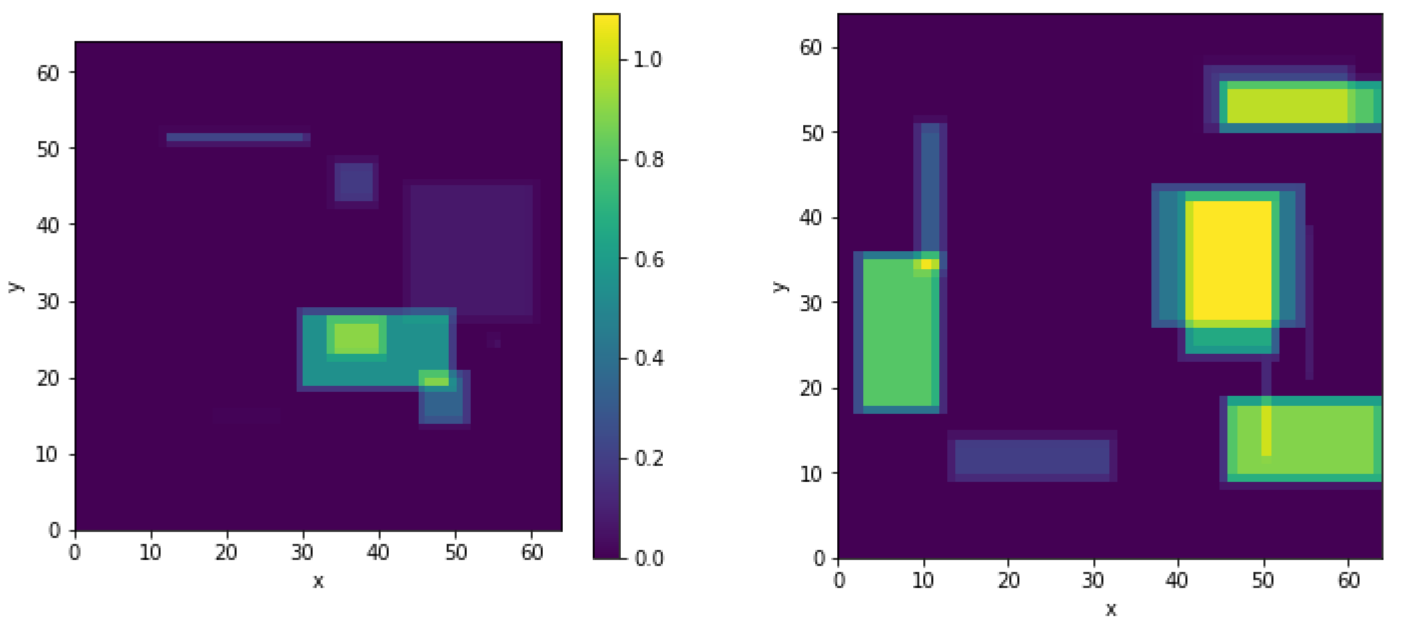
```
def generate_heat_example(*shape, bounds: Box = None):
    shape = math.merge_shapes(*shape)
    heat_t0 = CenteredGrid(0, extrapolation.PERIODIC, bounds, resolution=shape.spatial)
    bounds = heat_t0.bounds
    component_counts = math.to_int32(4 + 7 * math.random_uniform(shape.batch))
    positions = (math.random_uniform(shape.batch, batch(components=10), channel(vector=shape.spatial.names)) -
    for i in range(10):
        position = positions.components[i]
        half_size = math.random_uniform(shape.batch, channel(vector=shape.spatial.names)) * 10
        strength = math.random_uniform(shape.batch) * math.to_float(i < component_counts)
        position = math.clip(position, bounds.lower + half_size, bounds.upper - half_size)
        component_box = Cuboid(position, half_size)
        component_mask = SoftGeometryMask(component_box)
        component_mask = component_mask.at(heat_t0)
        heat_t0 += component_mask * strength
    return heat_t0
```

Look at two examples, generate data on-the-fly later during training.

In [3]:

```
vis.plot(generate_heat_example(batch(view_examples=2), spatial(x=64, y=64)))
```

Out[3]:



```
<Figure size 432x288 with 0 Axes>
```

Define scale-invariant updates. Take probabilistic viewpoint to avoid numerical explosions (see `probability_signal` in `apply_damping()`)

In [4]:

```
def apply_damping(kernel, inv_kernel, amp, f_uncertainty, log_kernel):
    signal_prior = 0.5
    expected_amp = 1. * kernel.shape.get_size('x') * inv_kernel  # This can be measured
    signal_likelihood = math.exp(-0.5 * (abs(amp) / expected_amp) ** 2) * signal_prior  # this can be NaN
    signal_likelihood = math.where(math.isfinite(signal_likelihood), signal_likelihood, math.zeros_like(signal_
    noise_likelihood = math.exp(-0.5 * (abs(amp) / f_uncertainty) ** 2) * (1 - signal_prior)
    probability_signal = math.divide_no_nan(signal_likelihood, (signal_likelihood + noise_likelihood))
    action = math.where((0.5 >= probability_signal) | (probability_signal >= 0.68), 2 * (probability_signal - (
    prob_kernel = math.exp(log_kernel * action)
    return prob_kernel, probability_signal


def inv_diffuse(grid: Grid, amount: float, uncertainty: Grid):
    f_uncertainty: math.Tensor = math.sqrt(math.sum(uncertainty.values ** 2, dim='x,y'))  # all frequencies hav
    k_squared: math.Tensor = math.sum(math.fftfreq(grid.shape, grid.dx) ** 2, 'vector')
    fft_laplace: math.Tensor = -(2 * np.pi) ** 2 * k_squared
    # --- Compute sharpening kernel with damping ---
    log_kernel = fft_laplace * -amount
    log_kernel_clamped = math.minimum(log_kernel, math.to_float(math.floor(math.log(math.wrap(np.finfo(np.float
    raw_kernel = math.exp(log_kernel_clamped)  # inverse diffusion FFT kernel, all values >= 1
    inv_kernel = math.exp(-log_kernel)
    amp = math.fft(grid.values)
    kernel, sig_prob = apply_damping(raw_kernel, inv_kernel, amp, f_uncertainty, log_kernel)
    # --- Apply and compute uncertainty ---
    data = math.real(math.ifft(amp * math.to_complex(kernel)))
    uncertainty = math.sqrt(math.sum(((f_uncertainty * kernel) ** 2))) / grid.shape.get_size('x')  # 1/N normal
    uncertainty = grid * 0 + uncertainty
    return grid.with_values(data), uncertainty, abs(amp), raw_kernel, kernel, sig_prob
```

Set up neural network + Adam We will use 64 bit precision for physics but 32 for network. Batch size 128.

In [5]:

```
math.set_global_precision(64)
BATCH = batch(batch=128)

math.seed(0)
net = u_net(1, 1)
optimizer = adam(net, 0.001)
```

Define loss function. Gradients will always computed as d(loss_function)/dθ (network weights). For SIP

- Invert the physics
- Define proxy loss L2(prediction - correction) where correction is taken as constant (based on `x = field.stop_gradient(prediction)`)

In [6]:

```
# @math.jit_compile
def loss_function(x_gt: CenteredGrid, sip: bool):
    y_target = diffuse.fourier(x_gt, 8., 1)
    with math.precision(32):
        prediction = field.native_call(net, field.to_float(y_target)).vector[0]
        prediction += field.mean(x_gt) - field.mean(prediction)
    prediction = field.to_float(prediction)  # is this call necessary?
    x = field.stop_gradient(prediction)
    if sip:
        y = diffuse.fourier(x, 8., 1)
        dx, _, amp, raw_kernel, kernel, sig_prob = inv_diffuse(y_target - y, 8., uncertainty=abs(y_target - y)
        correction = x + dx
        loss = field.l2_loss(prediction - correction)
        y_l2 = field.l2_loss(y - y_target)
    else:
        y = diffuse.fourier(prediction, 8., 1)
        y_l2 = loss = field.l2_loss(y - y_target)
    return loss, x, y, field.l2_loss(x_gt - x), y_l2
```

Training loop, generate data on-the-fly. (Switch between SIP and GD with the use_SIP checkbox or make two loops)

In [10]:

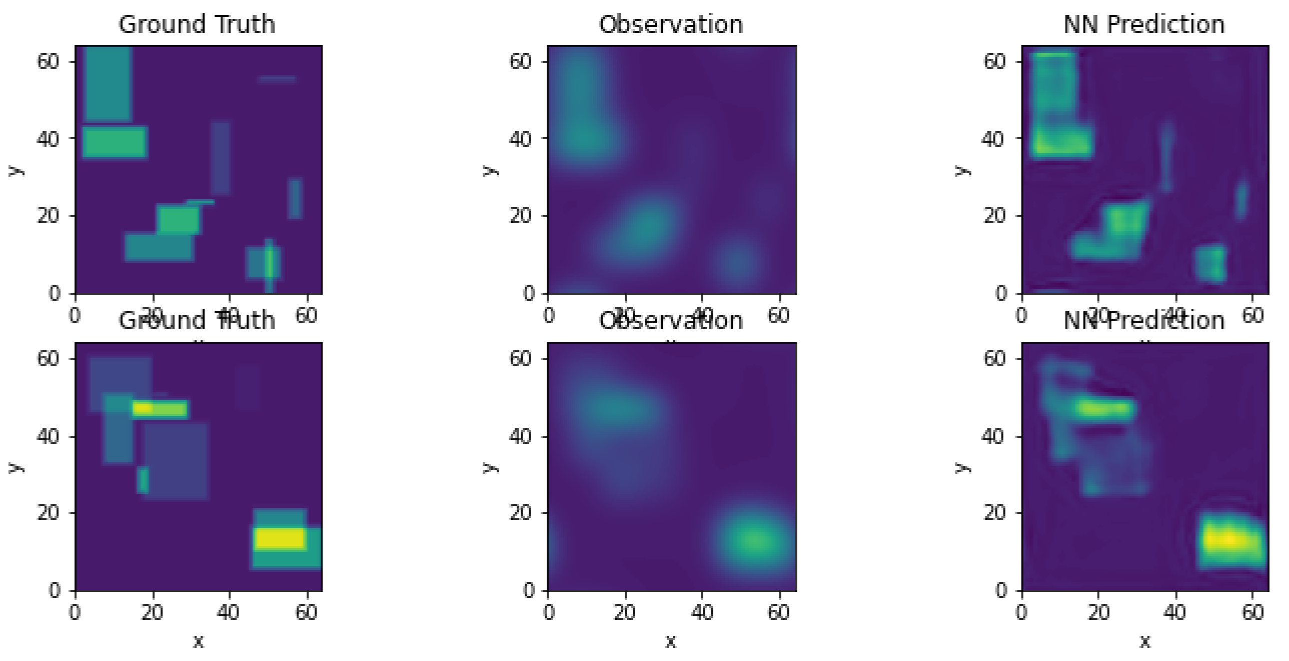
```
for training_step in range(500):
    data = generate_heat_example(spatial(x=64, y=64), BATCH)
    loss_value, x, y, x_l2, y_l2 = update_weights(net, optimizer, loss_function, data, sip=True)
    if training_step % 10 == 0: print(x_l2)
```

```
(batchᵇ=128) 10.493 ± 6.369 (2e+00...3e+01)
(batchᵇ=128) 10.187 ± 6.252 (2e+00...3e+01)
(batchᵇ=128) 11.047 ± 6.128 (2e+00...3e+01)
(batchᵇ=128) 10.072 ± 6.347 (1e+00...4e+01)
(batchᵇ=128) 8.682 ± 5.031 (1e+00...2e+01)
(batchᵇ=128) 8.241 ± 4.486 (1e+00...2e+01)
(batchᵇ=128) 8.466 ± 4.927 (1e+00...2e+01)
(batchᵇ=128) 9.183 ± 6.186 (1e+00...5e+01)
(batchᵇ=128) 8.269 ± 5.729 (1e+00...4e+01)
(batchᵇ=128) 7.814 ± 4.365 (8e-01...2e+01)
(batchᵇ=128) 8.390 ± 5.162 (1e+00...2e+01)
(batchᵇ=128) 7.937 ± 4.595 (2e+00...2e+01)
(batchᵇ=128) 7.315 ± 4.118 (1e+00...2e+01)
(batchᵇ=128) 7.617 ± 4.776 (1e+00...3e+01)
(batchᵇ=128) 7.730 ± 4.983 (7e-01...2e+01)
(batchᵇ=128) 7.126 ± 3.905 (5e-01...2e+01)
(batchᵇ=128) 7.256 ± 6.348 (8e-01...6e+01)
(batchᵇ=128) 7.313 ± 3.860 (8e-01...2e+01)
(batchᵇ=128) 8.344 ± 5.242 (5e-01...3e+01)
(batchᵇ=128) 6.463 ± 4.300 (6e-01...2e+01)
(batchᵇ=128) 7.400 ± 4.349 (9e-01...3e+01)
(batchᵇ=128) 7.453 ± 4.910 (5e-01...3e+01)
(batchᵇ=128) 7.298 ± 5.256 (1e+00...4e+01)
(batchᵇ=128) 7.223 ± 4.199 (7e-01...2e+01)
(batchᵇ=128) 6.610 ± 4.331 (3e-01...2e+01)
(batchᵇ=128) 6.666 ± 3.827 (3e-01...2e+01)
(batchᵇ=128) 6.979 ± 4.320 (1e+00...2e+01)
(batchᵇ=128) 7.171 ± 4.482 (3e-01...2e+01)
(batchᵇ=128) 6.919 ± 3.900 (1e+00...2e+01)
(batchᵇ=128) 6.410 ± 4.150 (6e-01...2e+01)
(batchᵇ=128) 6.273 ± 3.477 (5e-01...2e+01)
(batchᵇ=128) 7.019 ± 4.924 (9e-01...3e+01)
(batchᵇ=128) 7.876 ± 4.400 (2e+00...2e+01)
(batchᵇ=128) 6.431 ± 4.248 (5e-01...2e+01)
(batchᵇ=128) 6.557 ± 3.771 (8e-01...2e+01)
(batchᵇ=128) 5.735 ± 3.723 (5e-01...2e+01)
(batchᵇ=128) 7.207 ± 4.388 (6e-01...2e+01)
(batchᵇ=128) 6.715 ± 3.744 (1e+00...2e+01)
(batchᵇ=128) 6.823 ± 4.558 (8e-01...2e+01)
(batchᵇ=128) 5.899 ± 3.558 (5e-01...2e+01)
```

In [11]:

```
test_data = generate_heat_example(batch(test_examples=2), spatial(x=64, y=64))
_, test_pred, test_y, *_ = loss_function(test_data, True)
vis.plot({"Ground Truth": test_data, "Observation": test_y, "NN Prediction": test_pred}, show_color_bar=False,
```

Out[11]:



```
<Figure size 432x288 with 0 Axes>
```