viz_nlgp

October 4, 2023

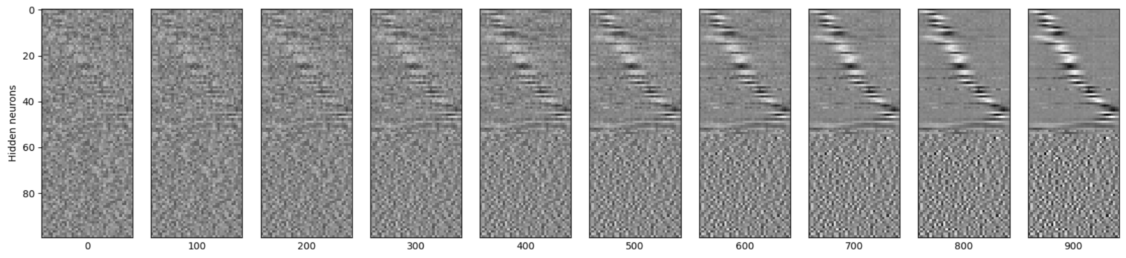
Same as viz, but for the NLGP dataset

```
[ ]: import os
     import numpy as np
     import pandas as pd
     import jax
     import jax.numpy as jnp
     import optax
     import datasets
     import models
     import samplers
     import matplotlib.pyplot as plt
     from conv_emergence import compute_entropy as entropy
     from experiments import make_key, simulate
     from viz_box import position_mean_var, ipr, entropy_sort, mean_sort, var_sort,␣
      ↪plot_receptive_fields
```

```
[ ]: config = dict(
         seed=0,
         num_dimensions=40,
         num_hiddens=100,
         gain=1.1,
         init_scale=1.0,
         activation='tanh',
         model_cls=models.SimpleNet,
         optimizer_fn=optax.sgd,
         learning_rate=1.0,
         batch_size=1,
         num_epochs=1000,
         dataset_cls=datasets.NonlinearGPDataset,
         xi1=1.1,
         xi2=0.1,
         class_proportion=0.5,
         sampler_cls=samplers.EpochSampler,
         init_fn=models.lecun_normal_init
     )
```

Let's see how the weights evolve over time.

```
[ ]: # path_key = make_key(**config)
path_key = "xi1=04.47_xi2=00.10_gain=100.
 ↪00_L=100_K=040_dim=1_batch_size=100_num_epochs=1000_loss=mse_lr=1.
 ↪000_activation=tanh_second_layer=0.0_init_scale=1.000"
evaluation_interval = 100
weights = np.stack([ np.load(f"results/weights/{path_key}/fc1_{i}.npy") for i␣
 ↪in range(100, 1001, 100) ])
fig, axs = plot_receptive_fields(weights, num_cols=20, figsize=(40, 20),␣
 ↪reordering_fn=entropy_sort, ind=-1, center_sort=True,␣
 ↪evaluation_interval=evaluation_interval)
# fig.savefig(f"results/weights/{path_key}/receptive_fields.png", dpi=300,␣
 ↪bbox_inches='tight')
```

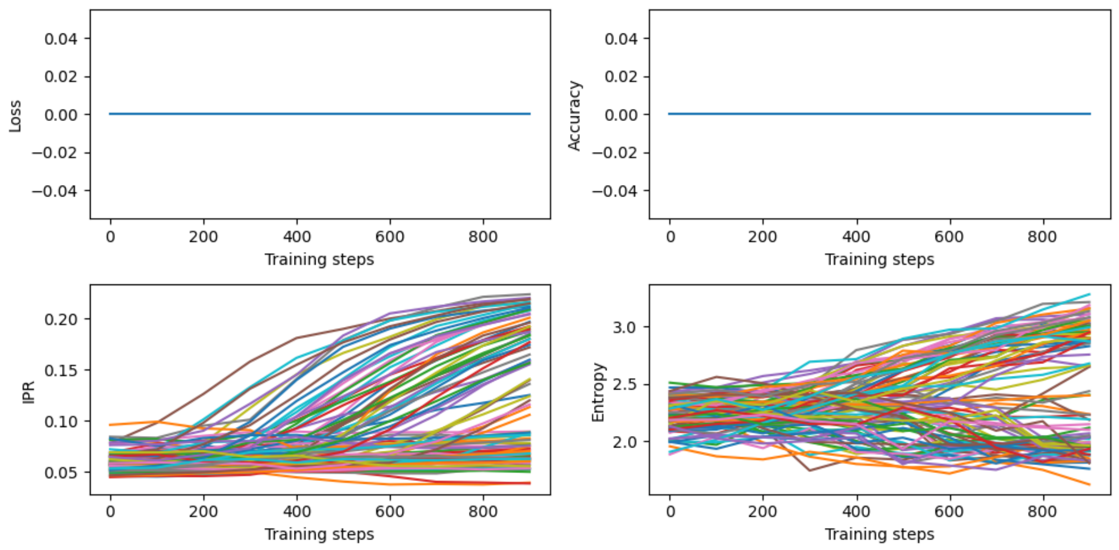


Now, let's see how some metrics change over time. We'll focus on loss, accuracy, IPR, and entropy for now.

```
[ ]: # metrics = pd.read_csv(f"results/weights/{path_key}/metrics.csv")
loss_ = np.zeros(10) #metrics['loss'].values
accuracy_ = np.zeros(10) #metrics['accuracy'].values
ipr_ = np.stack([ ipr(weight_) for weight_ in weights ])
entropy_ = np.stack([ entropy(weight_) for weight_ in weights ])
```

It seems like entropy is the more useful than IPR for discriminating between localized and "oscillatory" receptive fields.

```
[ ]: fig, axs = plt.subplots(2, 2, figsize=(10, 5))
for ax, metric, label in zip(axs.flatten(), [loss_, accuracy_, ipr_, entropy_],␣
 ↪['Loss', 'Accuracy', 'IPR', 'Entropy']):
    ax.plot(evaluation_interval * np.arange(len(metric)), metric)
    ax.set_xlabel('Training steps')
    ax.set_ylabel(label)
fig.tight_layout()
```

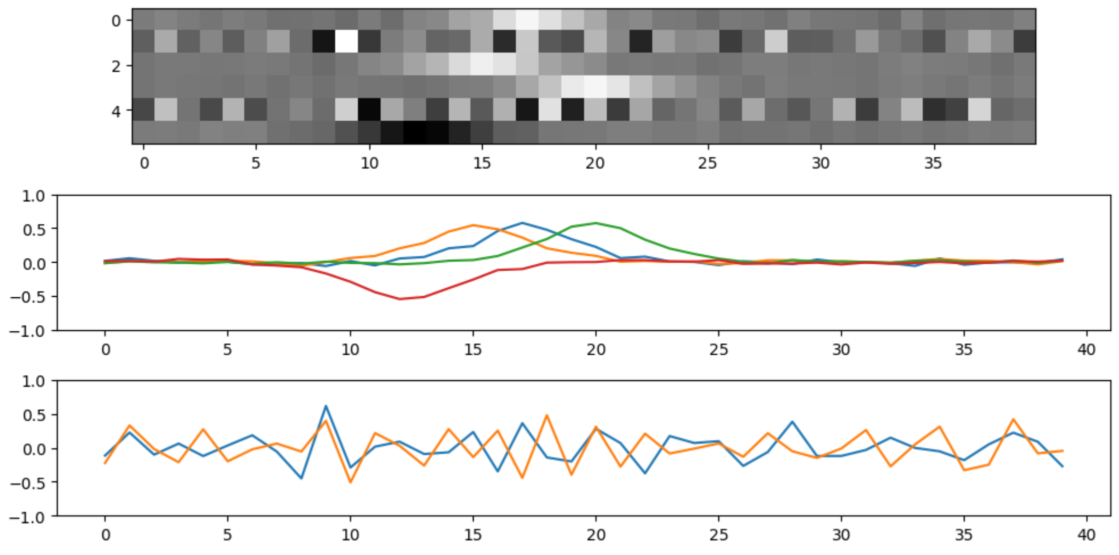


Now, let's see what the weights actually look like. That is, which kernel are the localized receptive field learning?

Looks like localized receptive fields have a baseline at 0.5 and drop to nearly -1.0. The oscillatory receptive fields hover around 0.

This plot (as well the full set of receptive fields above) shows us that all the localized receptive fields have roughly the same bandwidth of about five units.

```
[ ]: # Let's pick out a few weights to visualize
localized_inds = np.array([0, 2, 3, 5])
oscillatory_inds = np.array([1, 4])
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 5))
ax1.imshow(weights[-1][:6], cmap='gray')
ax2.plot(weights[-1][localized_inds].T)
ax2.set_ylim(-1, 1)
ax3.plot(weights[-1][oscillatory_inds].T)
ax3.set_ylim(-1, 1)
fig.tight_layout()
```



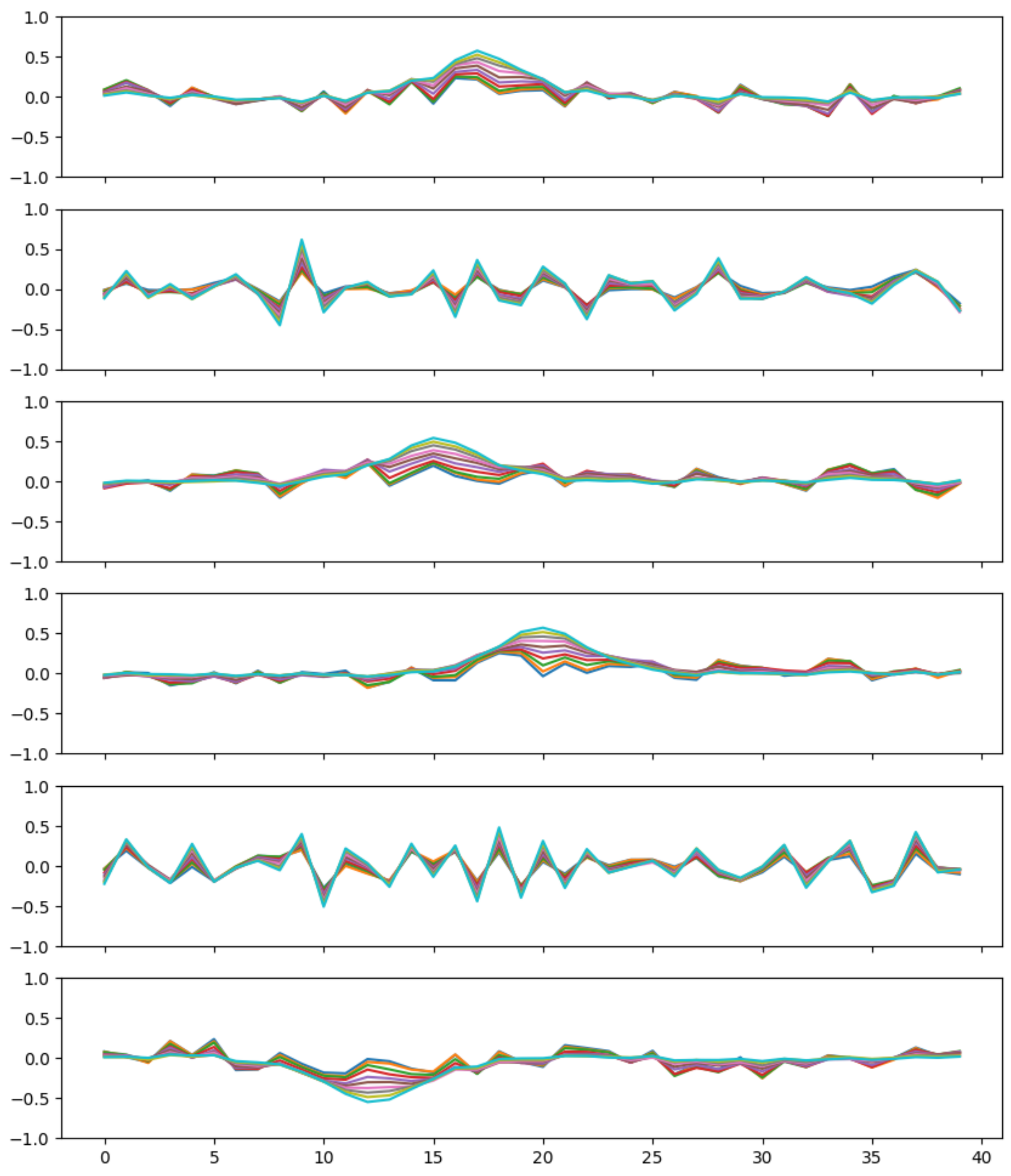
Now, let's see how the localized and oscillatory receptive fields evolve over time. First, we'll visualize each receptive fields at various points in time. Then, we'll see how the group evolves over time.

We'll see that the localized receptive fields' kernels begin to emerge immediately and grow in magnitude continuously and uniformly throughout training. The oscillatory receptive field barely changes at all during training. A closer look later (might not be plotted) reveals that oscillatory neurons seem to hover around -0.1 (just eyeballing, not quantitative at all).

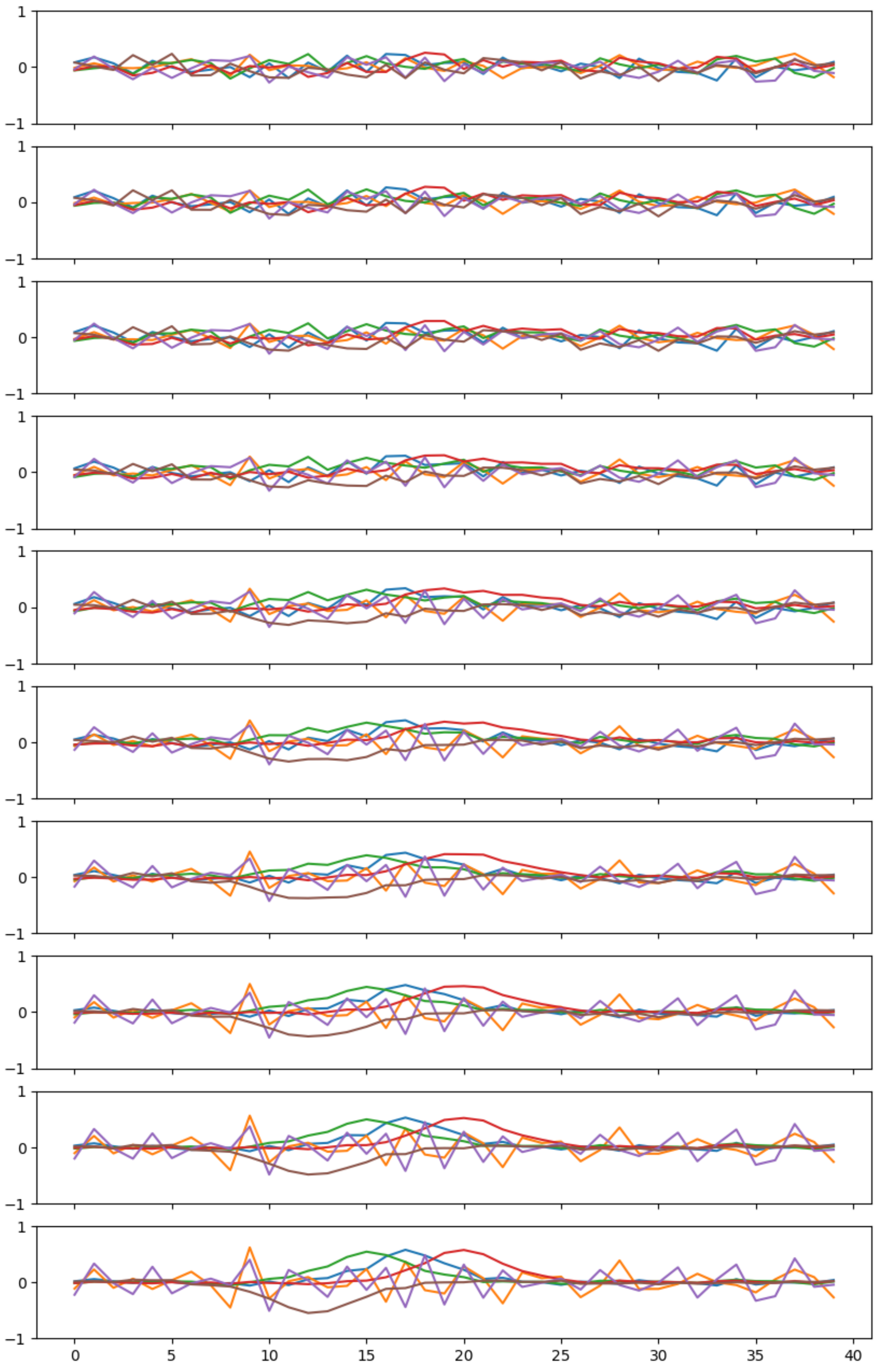
This begs the question: What happens if we prune the oscillatory receptive fields?

These graphs also suggest that looking at the largest absolute value of the weights for each neuron will be useful for discriminating localized and oscillatory receptive fields. I confirm this in the final plot, where we see the largest and mean absolute values grow linearly during training for localized neurons while they are constant for oscillatory neurons.

[ ]:
```
fig, axs = plt.subplots(6, 1, figsize=(10, 12), sharex=True, sharey=True)
for i, ax in enumerate(axs.flatten()):
    ax.plot(weights[:,i,:].T)
    ax.set_ylim(-1, 1)
```



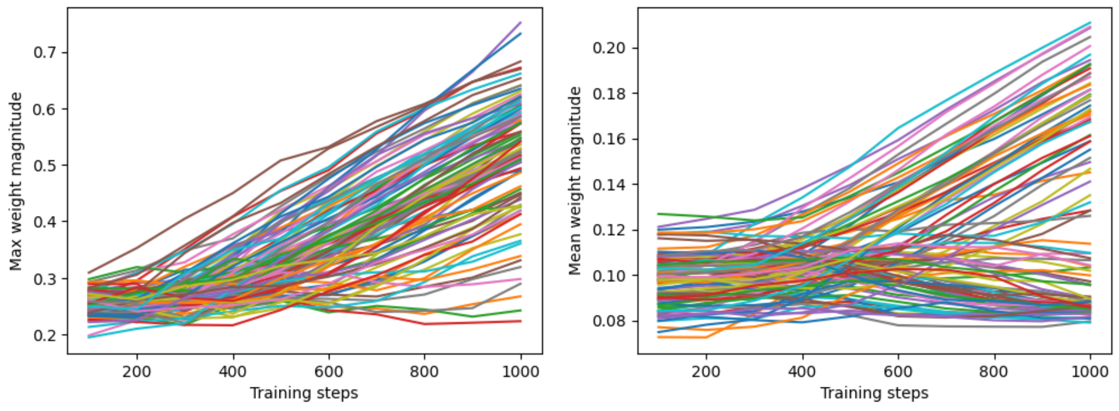
```
[ ]: fig, axs = plt.subplots(10, 1, figsize=(10, 16), sharex=True, sharey=True)
     for i, ax in enumerate(axs.flatten()):
         ax.plot(weights[i,:6,:].T)
         ax.set_ylim(-1, 1)
```



1
0
−1
0
5
10
15
20
25
30
35
40

[ ]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot( np.arange(1,11) * evaluation_interval, np.max(np.abs(weights),␣
 ↪axis=2) )
ax1.set_xlabel('Training steps')
ax1.set_ylabel('Max weight magnitude')
ax2.plot( np.arange(1,11) * evaluation_interval, np.mean(np.abs(weights),␣
 ↪axis=2) )
ax2.set_xlabel('Training steps')
ax2.set_ylabel('Mean weight magnitude')
```

[ ]:
```
Text(0, 0.5, 'Mean weight magnitude')
```



0.1 IGNORE EVERYTHING BELOW THIS POINT, NOT UPDATED FOR NLGP DATASET

Pruning oscillatory neurons Let's see how much behavior changes when we prune the oscillatory neurons. We'll sort them using the mean weight magnitude, since that's where we seem to get the cleanest separation.

It seems like the oscillatory neurons collectively act as a bias term. Inspecting their activations, we see that they (seemingly) always output ~1.0 in the hidden layer. The localized neurons tend to be strongly negative, but are more positive for the positive class (which has longer pulses).

So, the localized neurons must learn to offset the positive bias in prediction from the oscillatory neurons. Recall that the oscillatory neurons do not change throughout training. So why do the oscillatory neurons exist at all then? Why aren't their weights adjusted in addition to those of the localized neurons? Why do the localized neurons have to learn to correct for them?

[ ]:
```
localized_mask = np.mean(np.abs(weights[-1]), axis=-1) > 0.25
weights_ = weights[-1]
weights_localized = weights_[localized_mask]
```