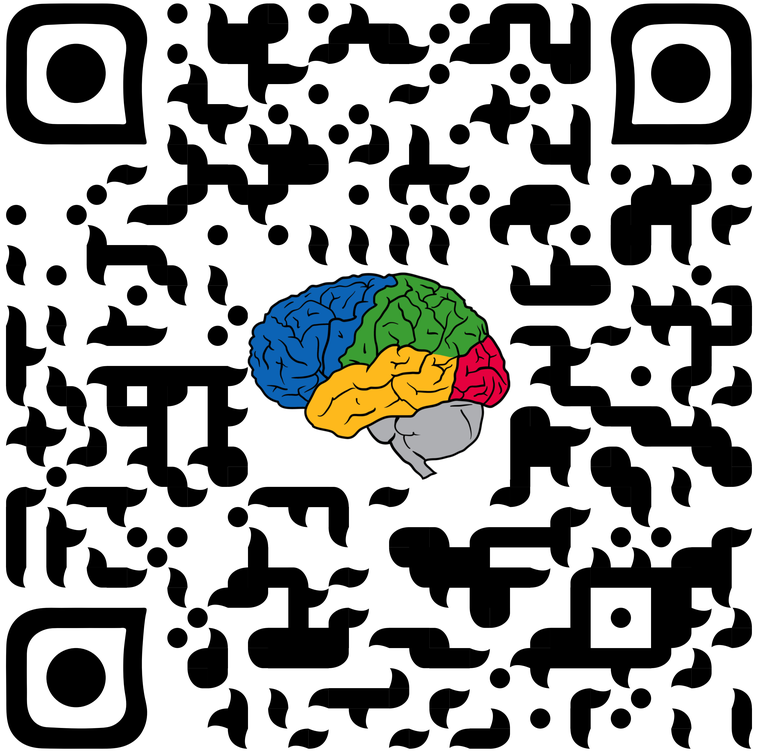
Neural Tangents:

Fast and Easy Infinite Neural Networks in Python



2nd Symposium on
Advances in Approximate Bayesian Inference

12/08/2019
Presentation by Roman Novak

github.com/google/neural-tangents

Collaborators

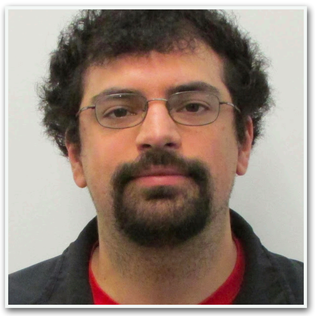
Lechao Xiao

Samuel S. Shoenholz

Jiri Hron

Jascha Sohl-Dickstein

Jaehoon Lee

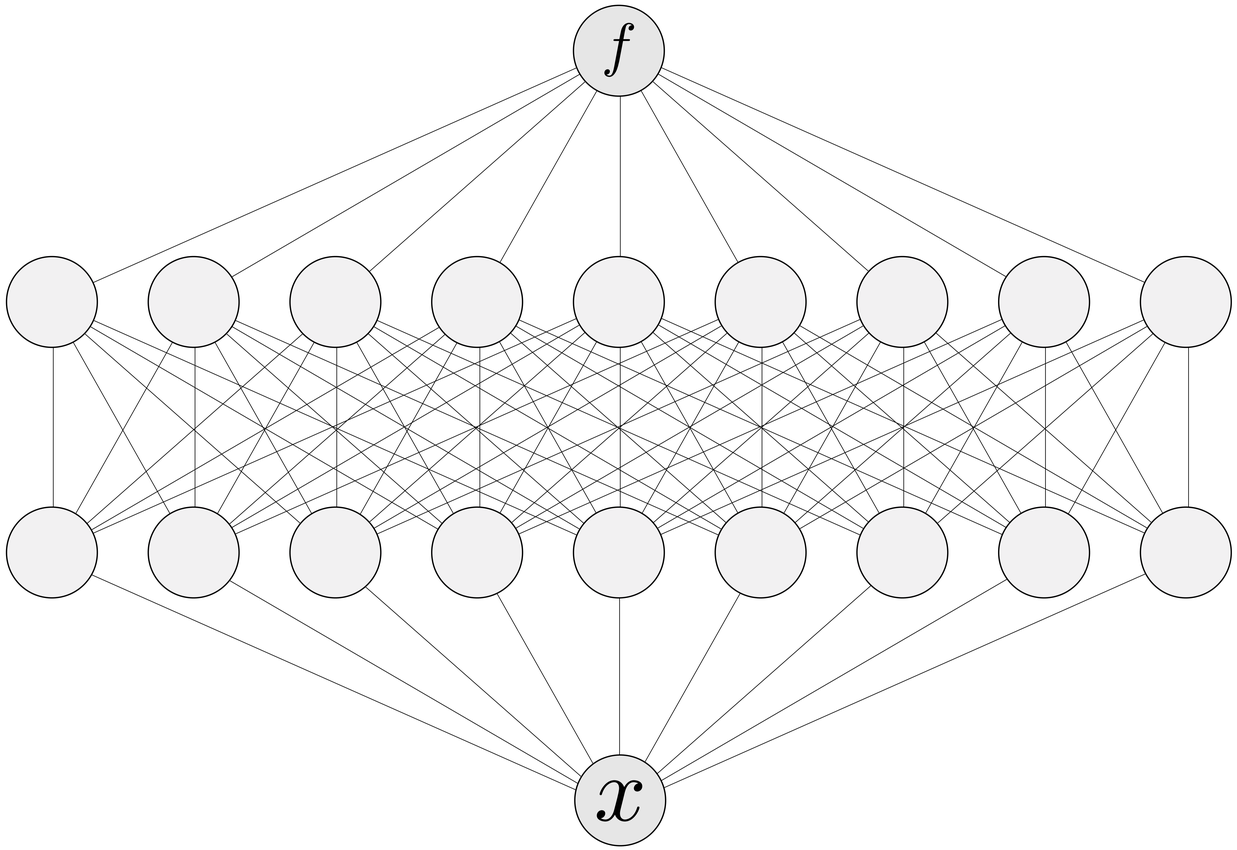


Alex Alemi

Roman Novak

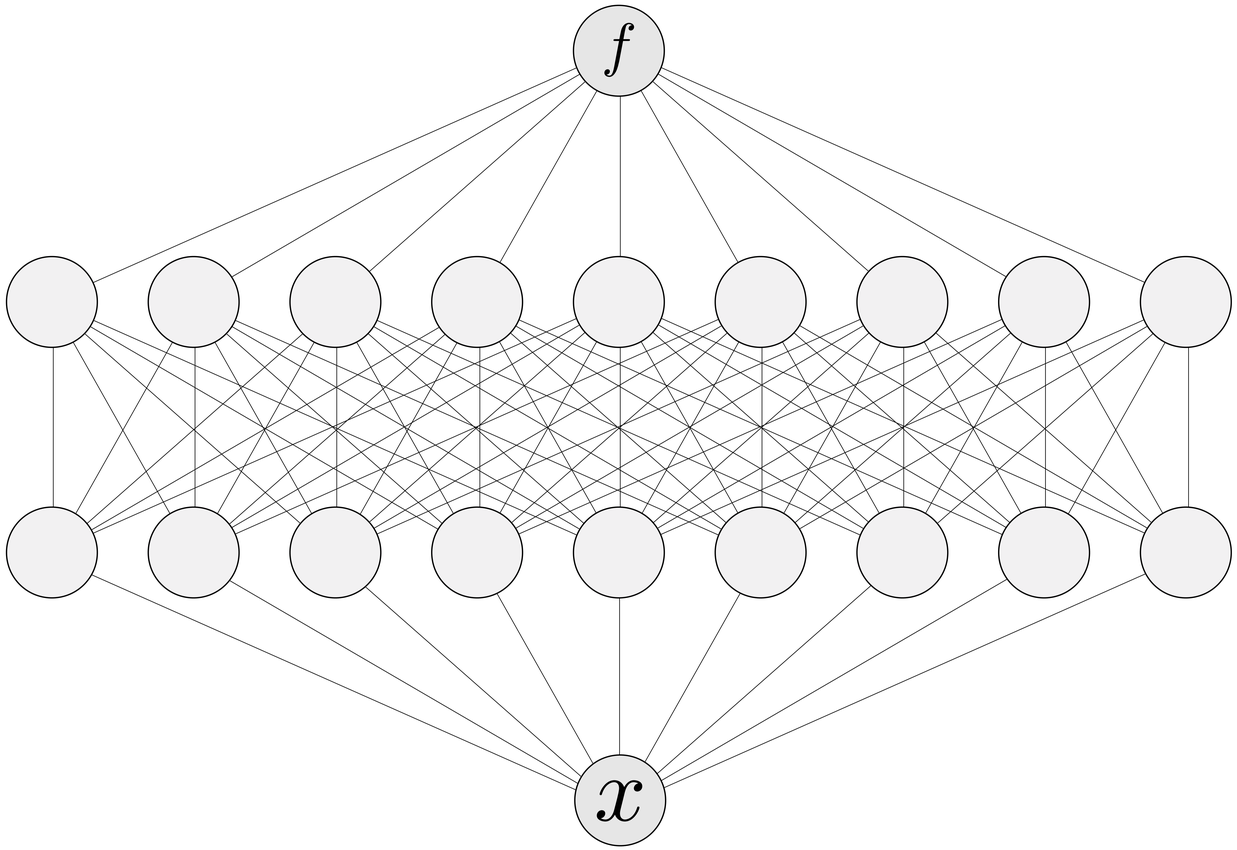
JAX

Distributions induced by NNs



$$\omega_i \sim \mathcal{N}\left(0, \frac{\sigma_\omega^2}{n}\right) \quad b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$$

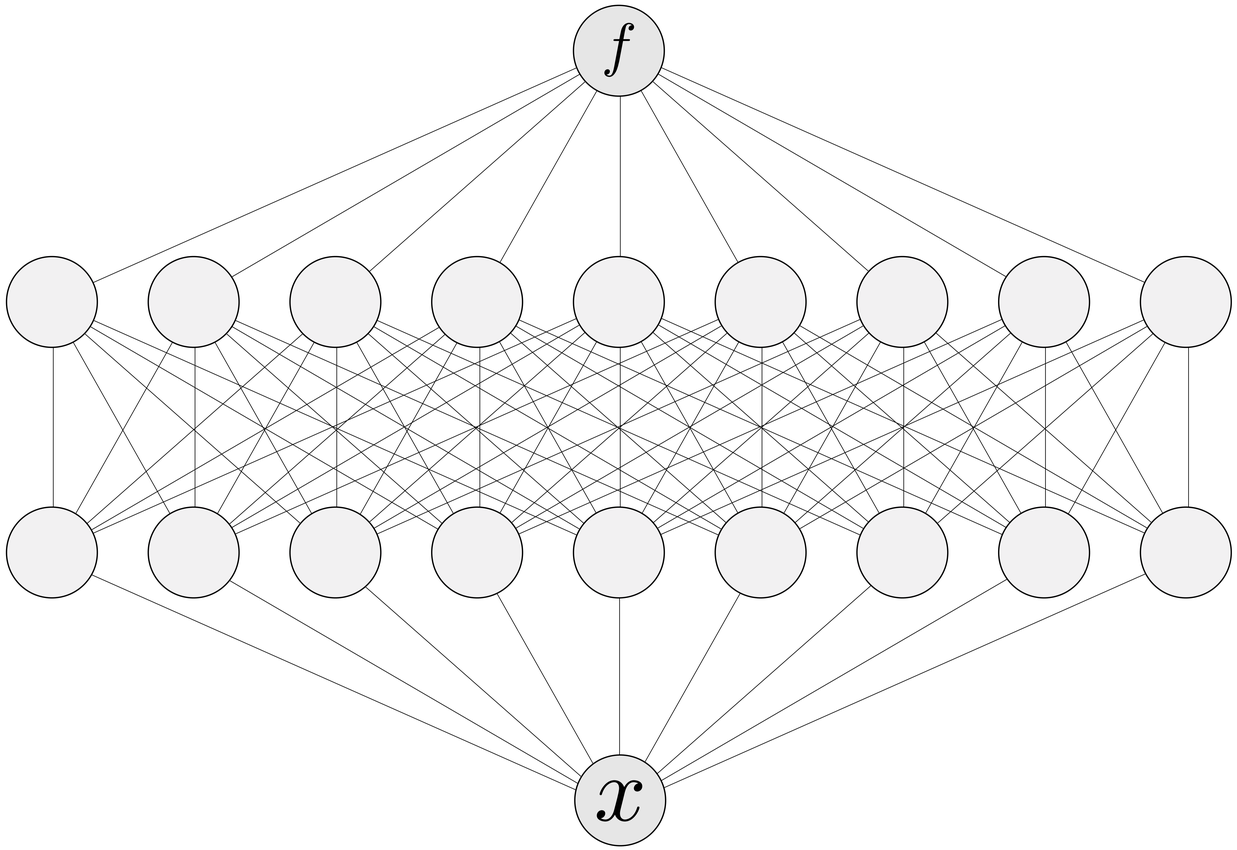
Distributions induced by NNs



$$[f(x^*)] \sim ?$$

$$\omega_i \sim \mathcal{N}\left(0, \frac{\sigma_\omega^2}{n}\right) \quad b_i \sim \mathcal{N}(0, \sigma_b^2)$$

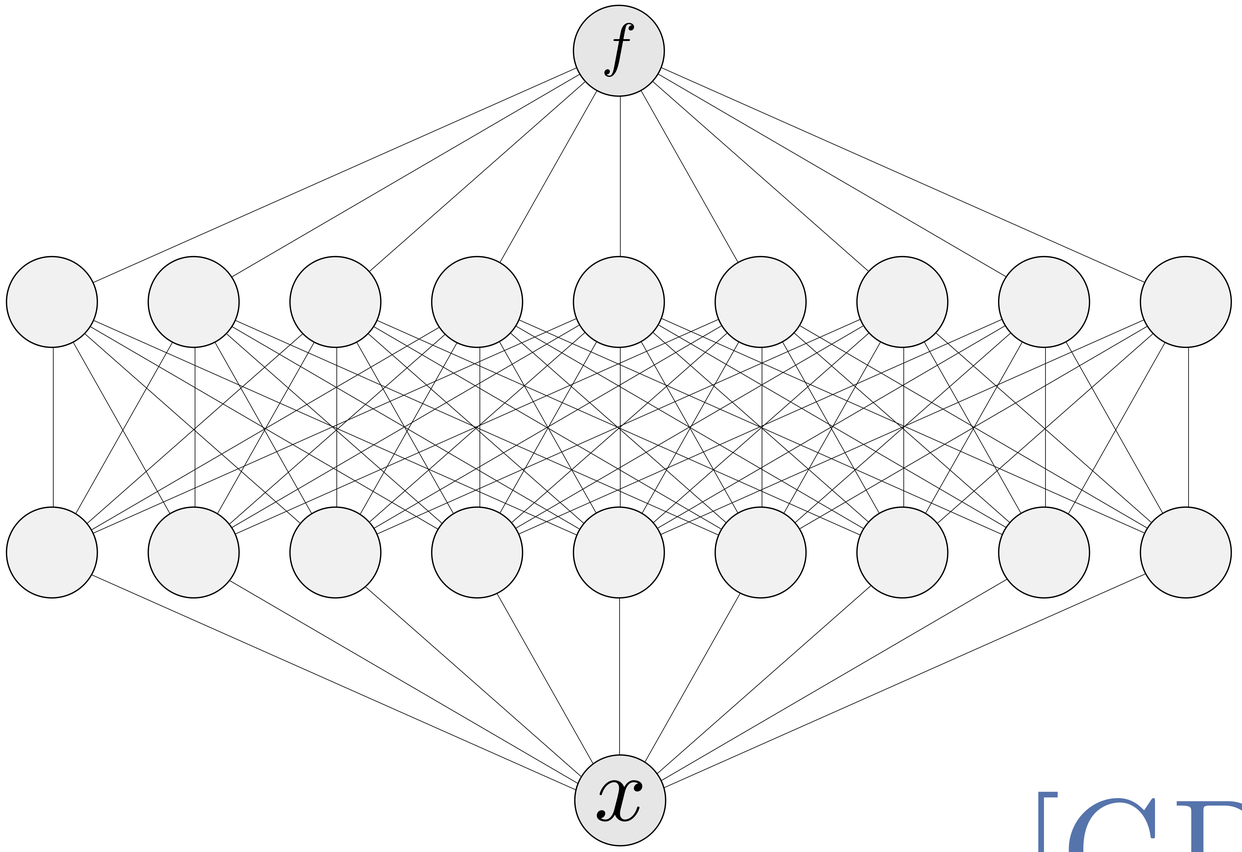
Distributions induced by NNs



$$\left[f\left(x^*\right) \middle| f\left(\mathcal{X}\right) = \mathcal{Y}\right] \sim ?$$

$$\omega_i \sim \mathcal{N}\left(0, \frac{\sigma_\omega^2}{n}\right) \quad b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$$

Distributions induced by NNs

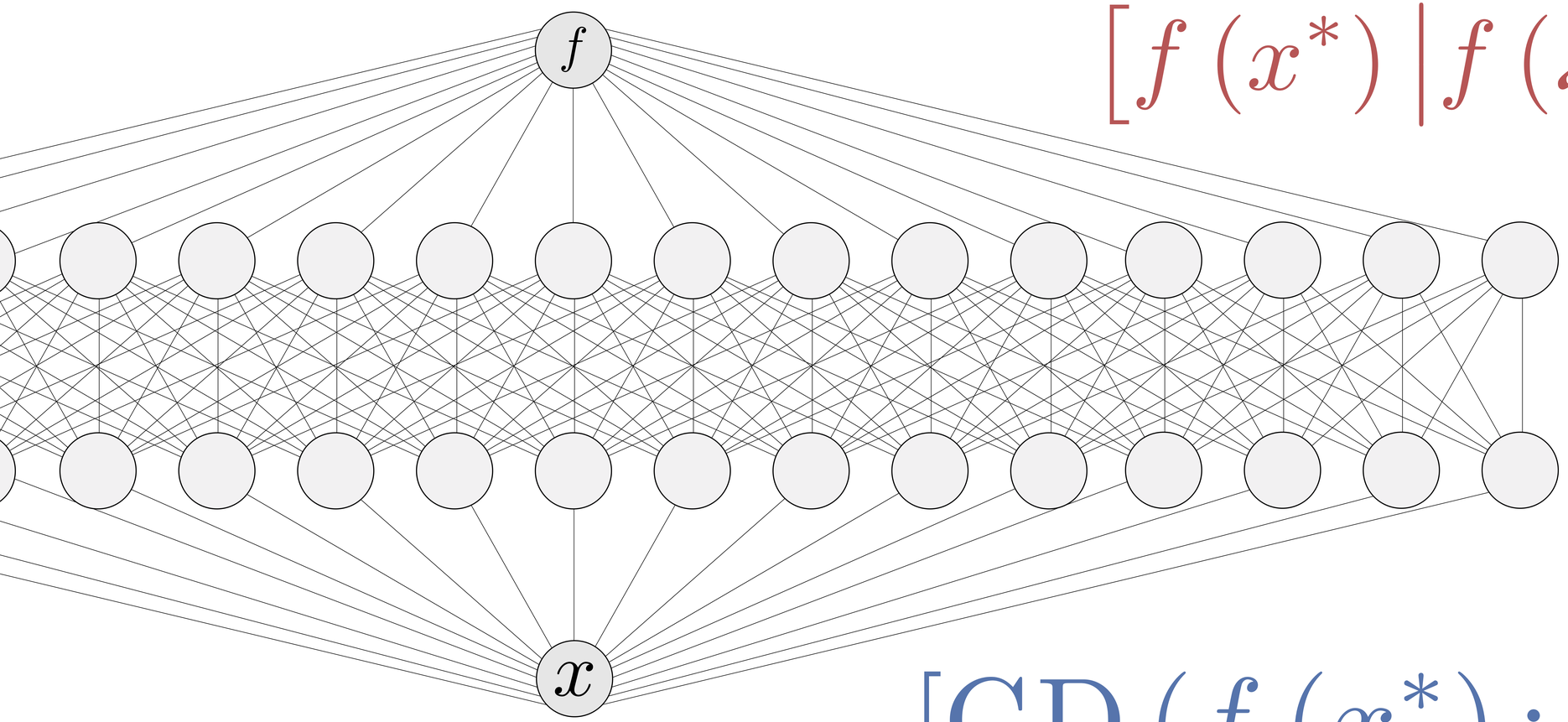


$$\left[f\left(x^*\right) \middle| f\left(\mathcal{X}\right) = \mathcal{Y}\right] \sim \ ?$$

$$\omega_i \sim \mathcal{N}\left(0, \frac{\sigma_\omega^2}{n}\right) \quad b_i \sim \mathcal{N}\left(0, \sigma_b^2\right)$$

$$\left[\mathrm{GD}\left(f\left(x^*\right); \mathcal{X}, \mathcal{Y}\right)\right] \sim \ ?$$

Distributions induced by wide NNs



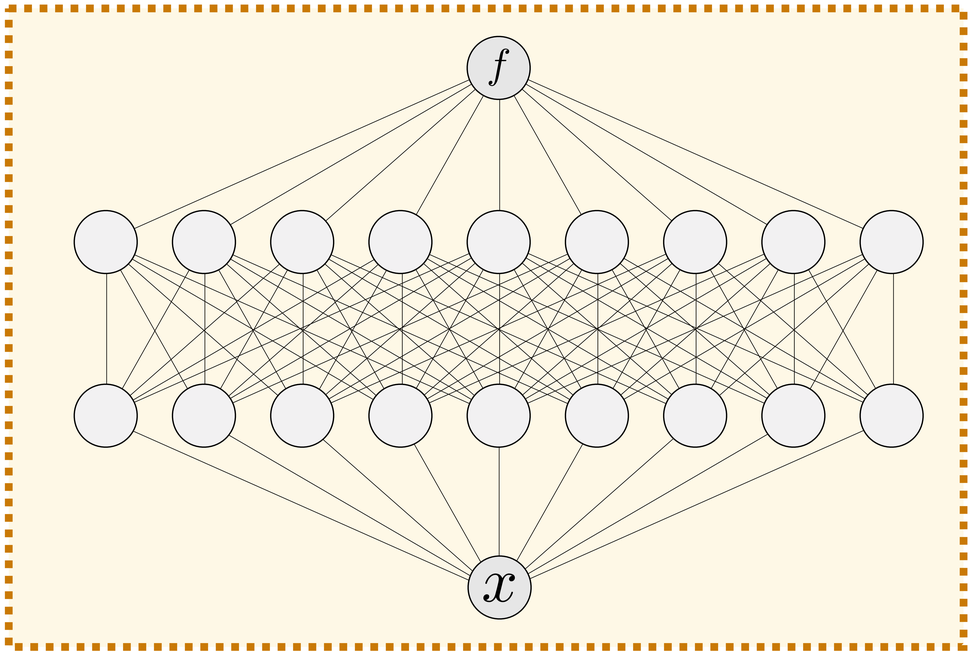
$$\left[f\left(x^*\right) \middle| f\left(\mathcal{X}\right) = \mathcal{Y}\right] \sim \, ?$$

$$n \to \infty$$

$$\left[\mathrm{GD}\left(f\left(x^*\right); \mathcal{X}, \mathcal{Y}\right)\right] \sim \, ?$$

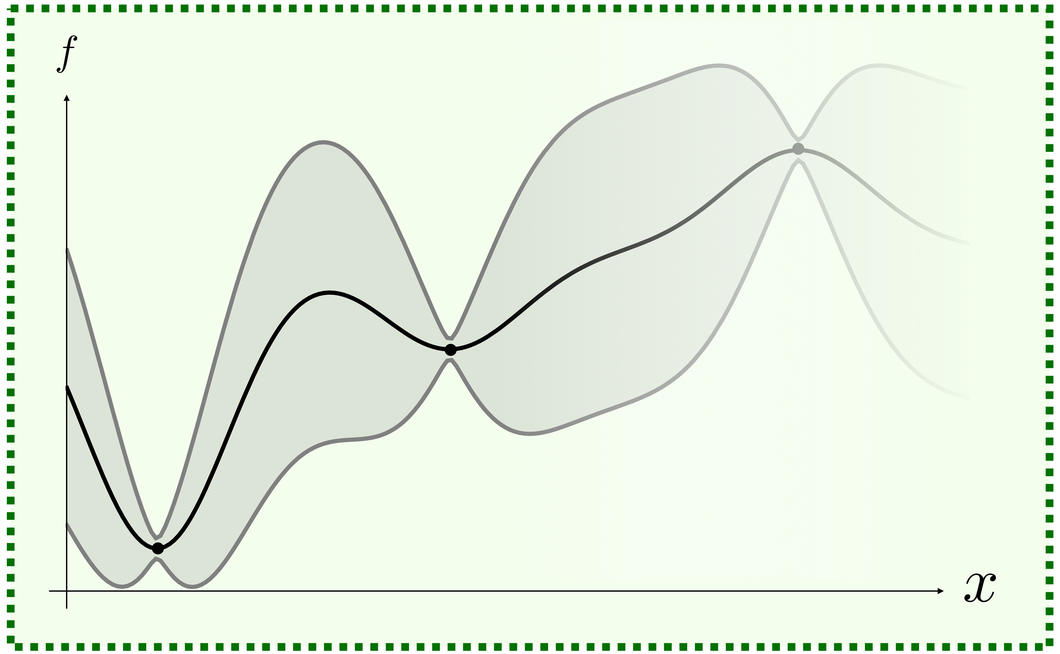
Distributions induced by infinite NNs

Neural network



$n \to \infty$

Gaussian process (GP)!



References

1. Bayesian FCNs (fully connected) → GP convergence:
   - **Neal, 1994** (single hidden layer)
   - Hazan & Jaakkola, 2015 (two layers)
   - Lee & Bahri et al, 2018; Matthews et al, 2018 (multi-layer)

2. Bayesian CNNs:
   - Borovykh, 2018 (single input sample)
   - Garriga-Alonso et al, 2019; Novak & Xiao et al, 2019 (multiple inputs)

3. Compositional kernels and wide NNs:
   - Cho & Saul, 2009 (rectified polynomials)
   - Daniely et al, 2016 (approximation guarantees, convergence)

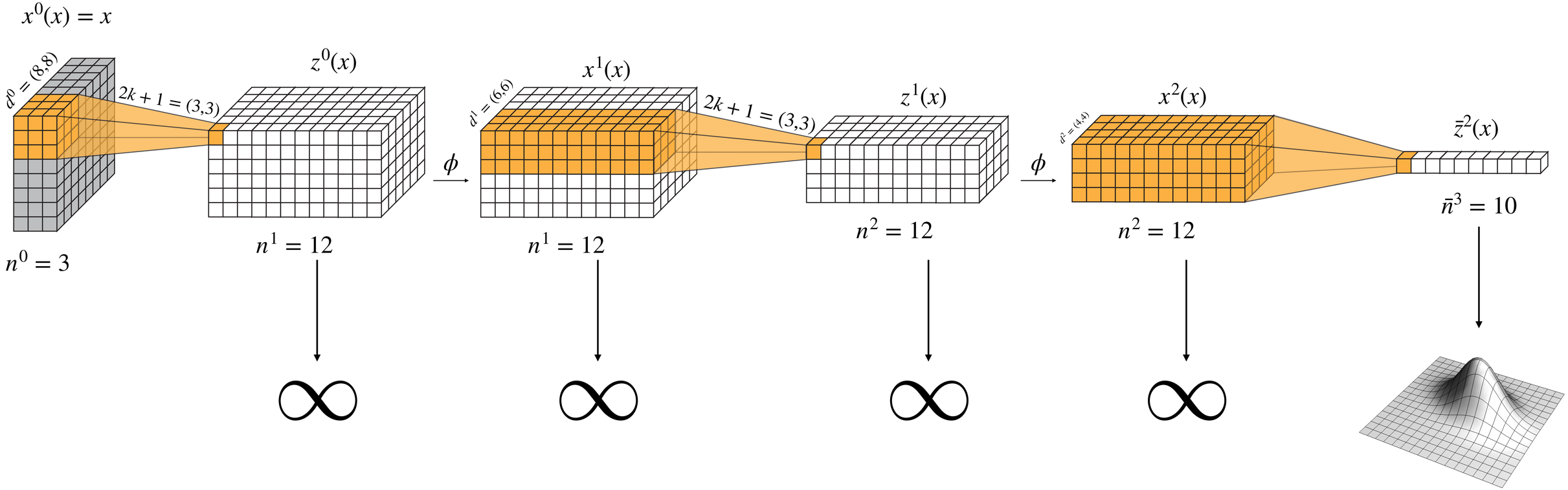
4. Mean field analysis:
   - Poole et al, 2016, Schoelholz et al, 2017 (FCNs)
   - Xiao et al, 2018 (CNNs)

5. GD NN → GP convergence:
   - **Jacot et al, 2018**
   - Lee & Xiao et al, 2019
   - Chizat et al, 2019
   - Du et al, 2018a;b
   - Arora et al, 2019

6. Generic architectures:
   - Yang, 2019

...(many more references in the paper)

Infinite networks are GPs*

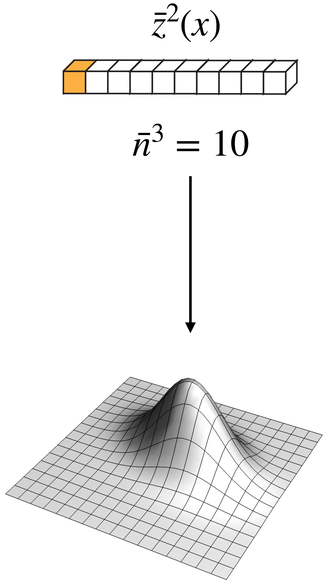


- Infinite networks are Gaussian Processes (GPs) with architecture-specific kernel (covariance) functions.*

Infinite Bayesian networks are GPs*

- **NNGP** (Neural Network Gaussian Process) kernel function, computable in closed form for many architectures:

$$\mathcal{K}(x, x') = \lim_{n \to \infty} f(\theta; x) f(\theta; x')^T$$



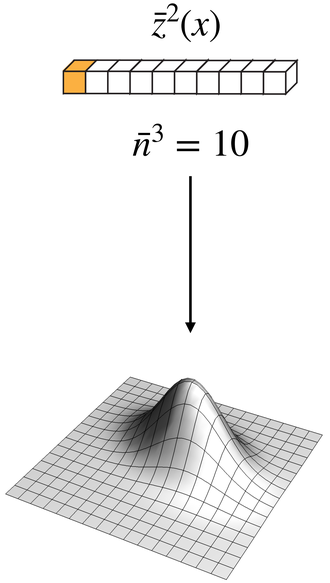
- Test set predictions, a multivariate normal:

$$f^* \sim \mathcal{N}\left(\mathcal{K}(x^*, \mathcal{X})\mathcal{K}^{-1}\mathcal{Y}, \mathcal{K}(x^*, x^*) - \mathcal{K}(x^*, \mathcal{X})\mathcal{K}^{-1}\mathcal{K}(\mathcal{X}, x^*)\right)$$

Infinite GD-trained networks are GPs*

- **NTK** (Neural Tangent Kernel), computable in closed form for many architectures:

$$\Theta\left(x, x'\right) = \lim_{n \to \infty} \frac{\partial f\left(\theta; x\right)}{\partial \theta} \left(\frac{\partial f\left(\theta; x'\right)}{\partial \theta} \right)^T$$

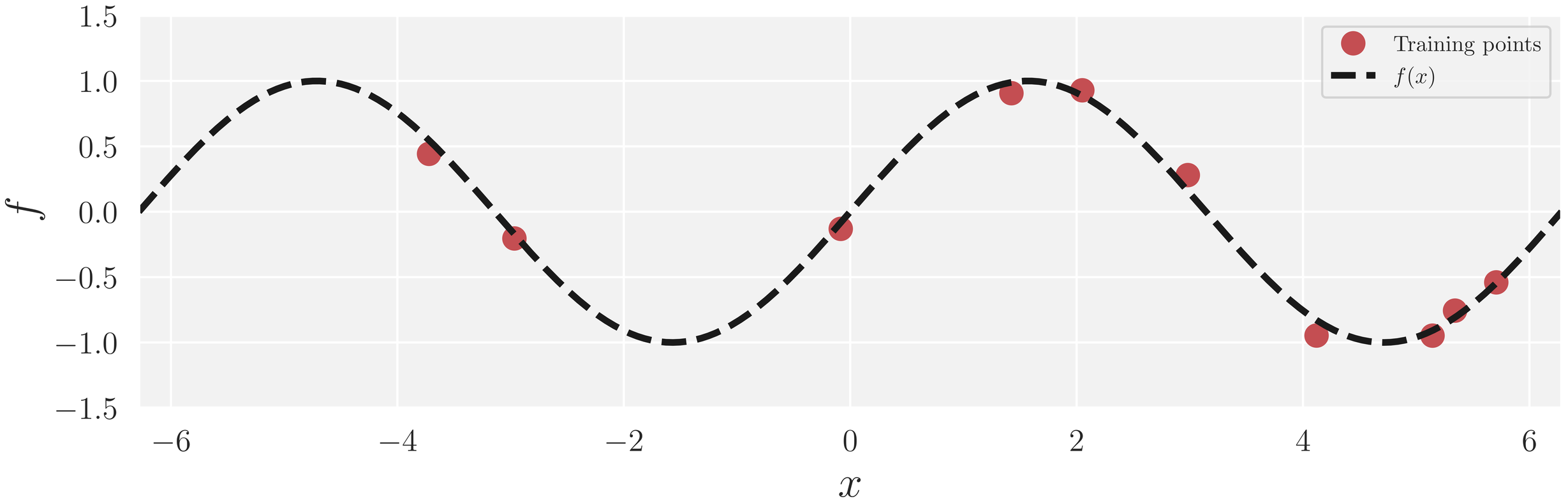


- Test set predictions, a multivariate normal:

$$f^* \sim \mathcal{N}\left(\Theta\left(x^*, \mathcal{X}\right)\Theta^{-1}\mathcal{Y}, \mathcal{K}\left(x^*, x^*\right) + \Theta(x^*, \mathcal{X})\Theta^{-1}\mathcal{K}\Theta^{-1}\Theta\left(\mathcal{X}, x^*\right) - \left(\Theta(x^*, \mathcal{X})\Theta^{-1}\mathcal{K}\left(\mathcal{X}, x^*\right) + h.c.\right)\right)$$

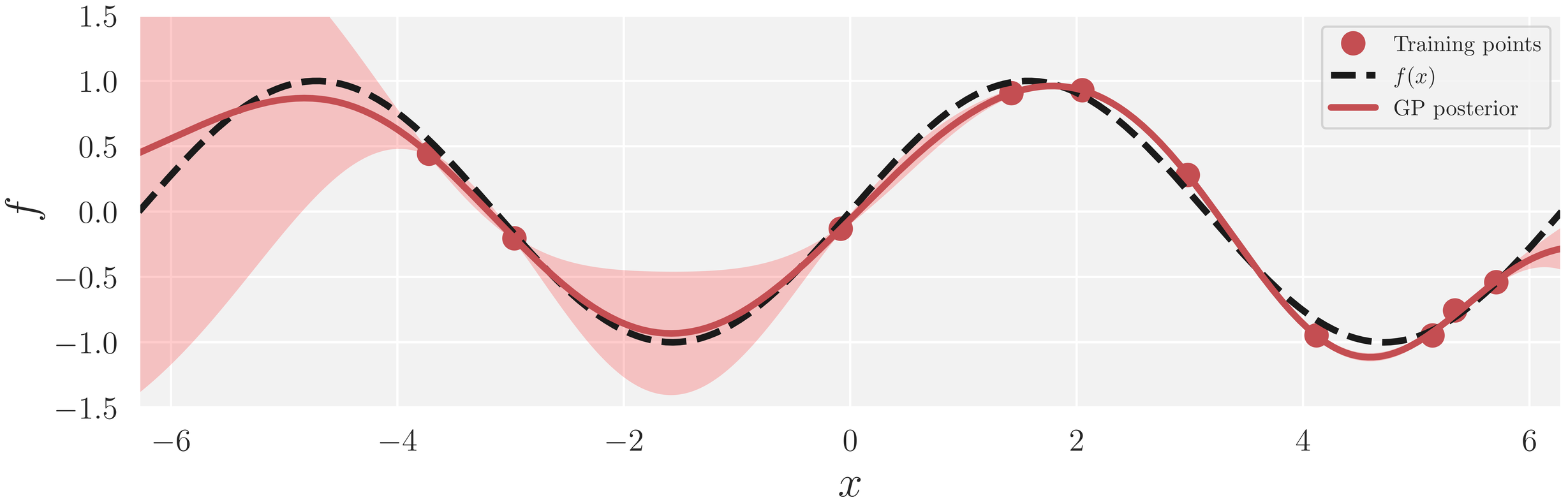
GP example: training set

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) = \sin(x) \qquad y = f(x) + \mathcal{N}\left(0, \sigma^2\right)$$



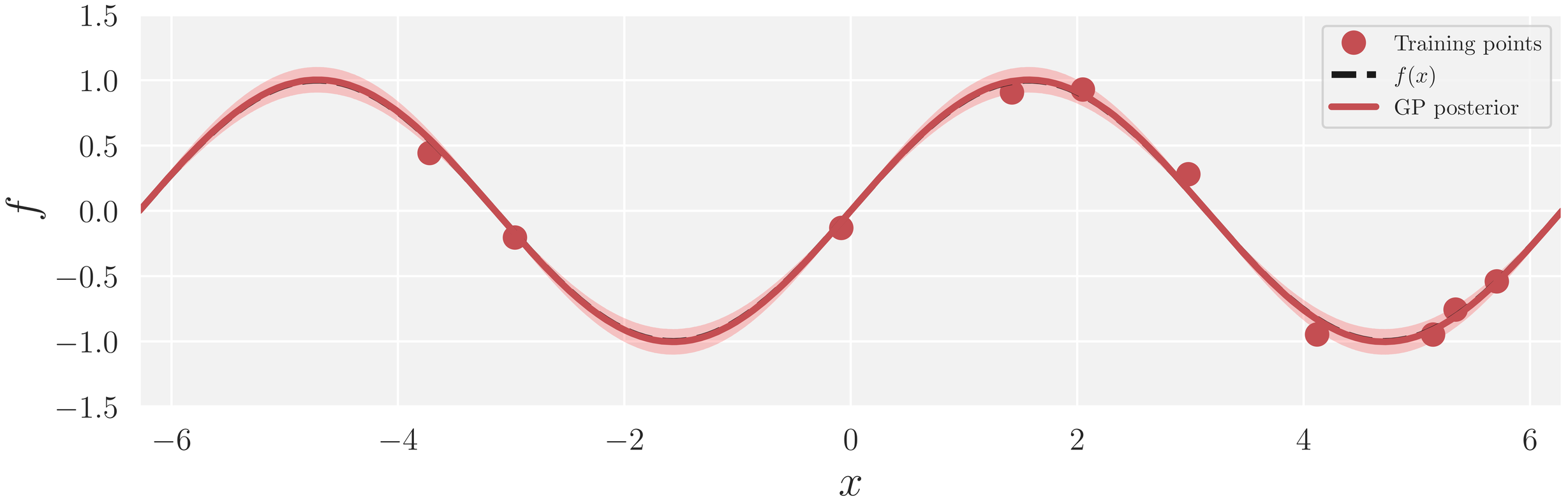
GP example: Gaussian kernel

$$K(x_1, x_2) \sim e^{-(x_1 - x_2)^2}$$



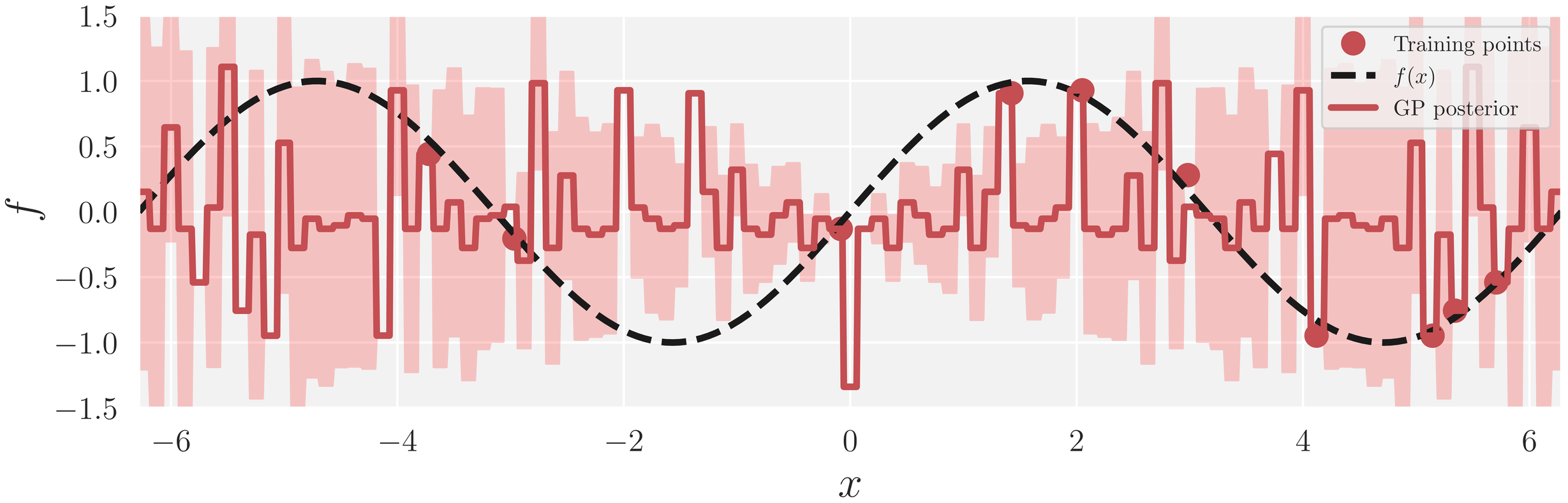
GP example: sine-product kernel

$$K(x_1, x_2) \sim \sin(x_1)\sin(x_2)$$



GP example: GCD kernel

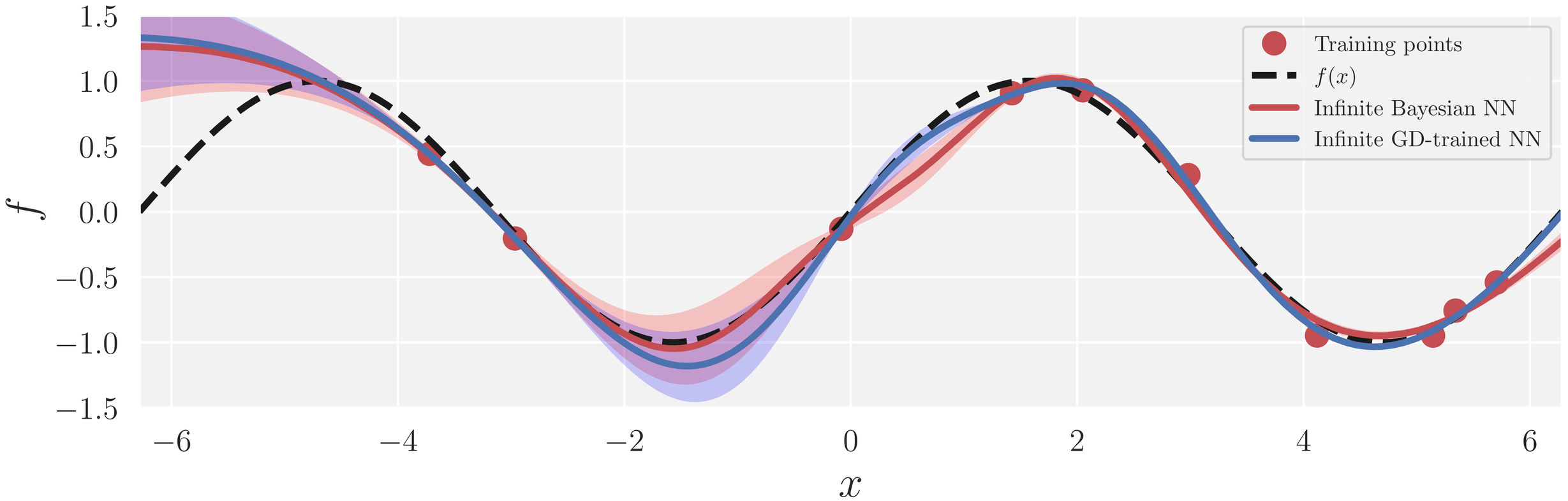
$$K(x_1, x_2) \sim \mathrm{GCD}(x_1, x_2)$$



Bayesian/GD-trained deep NN kernels

$$K(x_1, x_2) \sim \mathcal{K}(x_1, x_2) \quad K(x_1, x_2) \sim \Theta(x_1, x_2)^*$$

(NNGP) **(NTK)**



1. Insight into wide networks

1. Closed-form weights and outputs training dynamics
2. Predicting trainability and generalization
3. Learning rate, weight and bias variance selection
4. Correct initialization based on signal propagation

...

2. Standalone ML models

1. SOTA on CIFAR10 among non-trainable kernels

   (Novak & Xiao et al, 2019; Arora et al, 2019a; Li et al, 2019)

2. Strong performance on small datasets

   (Arora et al, 2019b)

3. Strong performance against fully-connected (FCNs) and locally-connected (LCNs; CNNs w/o weight sharing) networks

   (Lee & Bahri et al, 2018; Novak & Xiao et al, 2019)

4. Natural choice as baselines for Bayesian NNs

5. Principled uncertainty estimates

How to use infinite networks?

- Implement everything from scratch.
- Derive 6D kernel expressions by hand for each new architecture.
- Know how to implement the computation in a hardware-friendly and numerically-stable way.

How to use infinite networks?

- ... or use Neural Tangents!

```
pip install neural-tangents
```

How to use Neural Tangents?

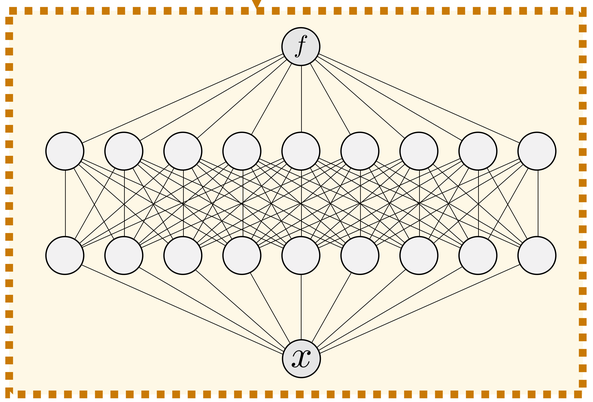
1. Define your (infinite) network:

```
from neural_tangents import stax
init_fn, apply_fn, kernel_fn = stax.serial(stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
                                           stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
                                           stax.Dense(1, W_std=1.5, b_std=0.05))
```

How to use Neural Tangents?

1. Define your (infinite) network:

```
from neural_tangents import stax
init_fn, apply_fn, kernel_fn = stax.serial(stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
                                           stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
                                           stax.Dense(1, W_std=1.5, b_std=0.05))
```

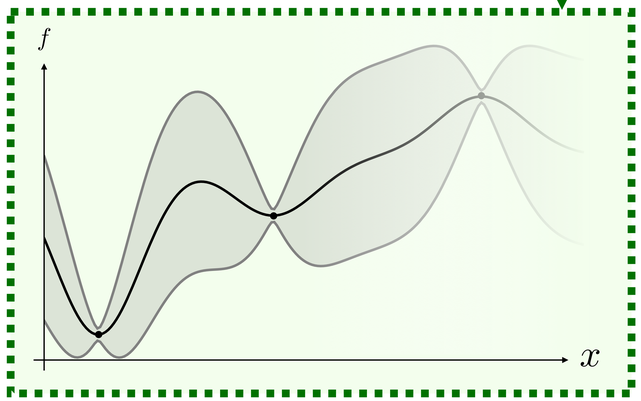


Init and forward-pass functions of the finite model

How to use Neural Tangents?

1. Define your (infinite) network:

```
from neural_tangents import stax
init_fn, apply_fn, kernel_fn = stax.serial(stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
                                           stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
                                           stax.Dense(1, W_std=1.5, b_std=0.05))
```



Kernel functions of the infinite models $\mathcal{K}(\cdot,\cdot)$, $\Theta(\cdot,\cdot)$

(NNGP) (NTK)

How to use Neural Tangents?

- Note: designed as a drop-in import replacement of stax, JAX neural network library.

jax.experimental.stax:

```
from jax.experimental import stax

init_fn, apply_fn = stax.serial(
    stax.Dense(512), stax.Relu,
    stax.Dense(512), stax.Relu,
    stax.Dense(1)
)
```

neural_tangents.stax:

```
from neural_tangents import stax

init_fn, apply_fn, kernel_fn = stax.serial(
    stax.Dense(512), stax.Relu(),
    stax.Dense(512), stax.Relu(),
    stax.Dense(1)
)
```

How to use Neural Tangents?

A. Perform Bayesian inference with an infinite NN

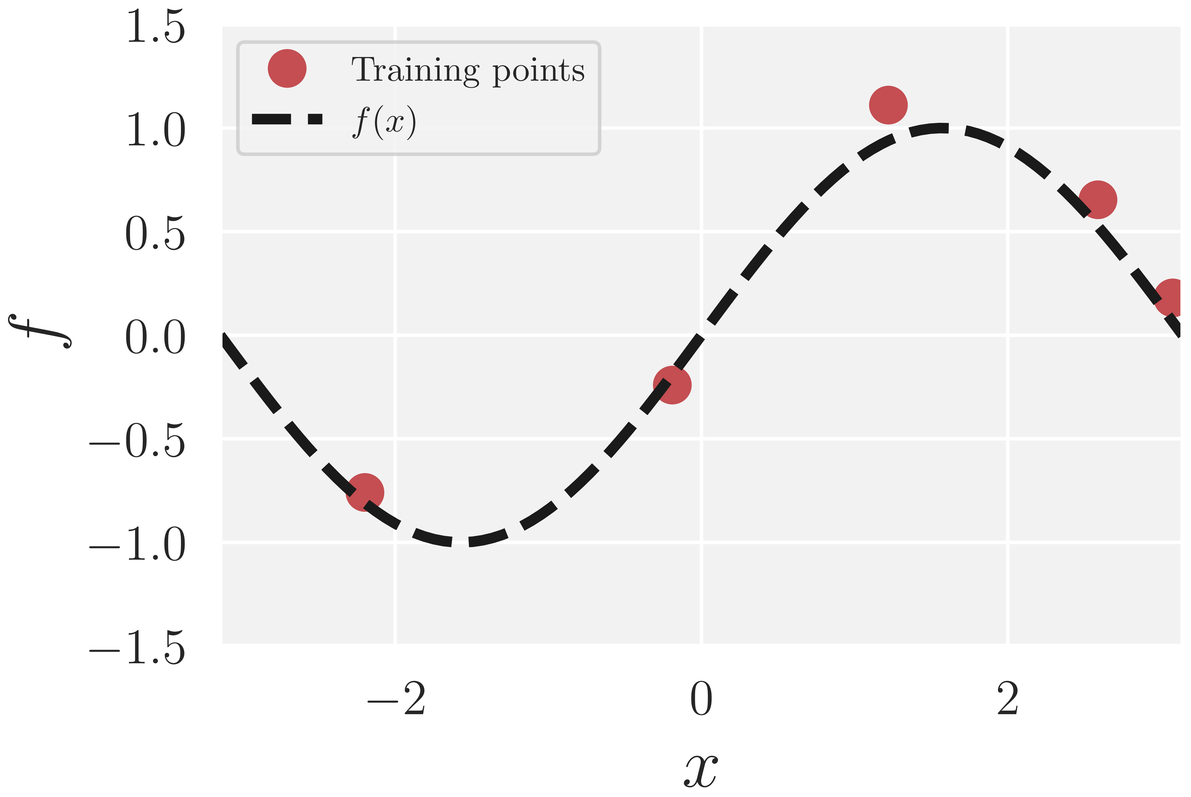
```
from neural_tangents import predict

f_mean, f_cov = predict.gp_inference(kernel_fn, x_train, f_train, x_test,
                                     get='nngp', diag_reg=1e-4, compute_cov=True)
```

$$f_t^* \sim \mathcal{N}\left(\mathcal{K}\left(x^*, \mathcal{X}\right)\mathcal{K}^{-1}\mathcal{Y}, \quad \ldots\right)$$

How to use Neural Tangents?

A. Perform Bayesian inference with an infinite NN

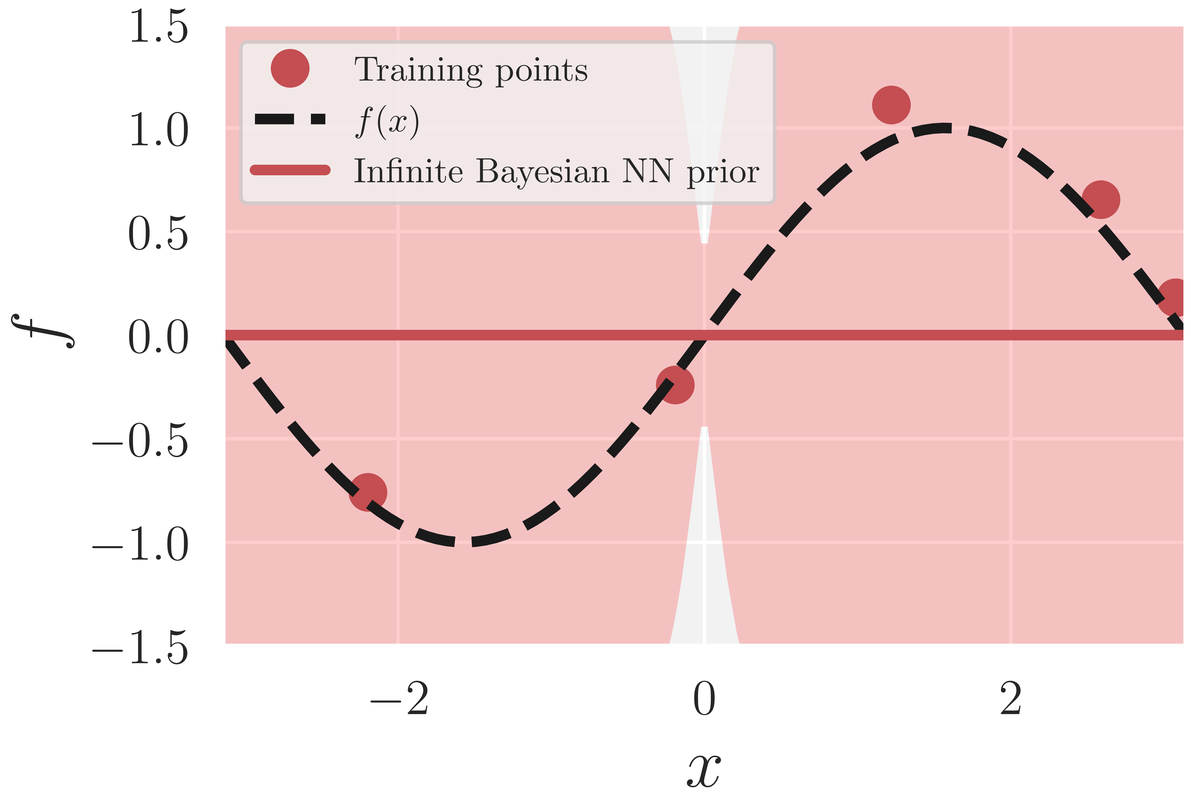
Training data:



How to use Neural Tangents?

A. Perform Bayesian inference with an infinite NN

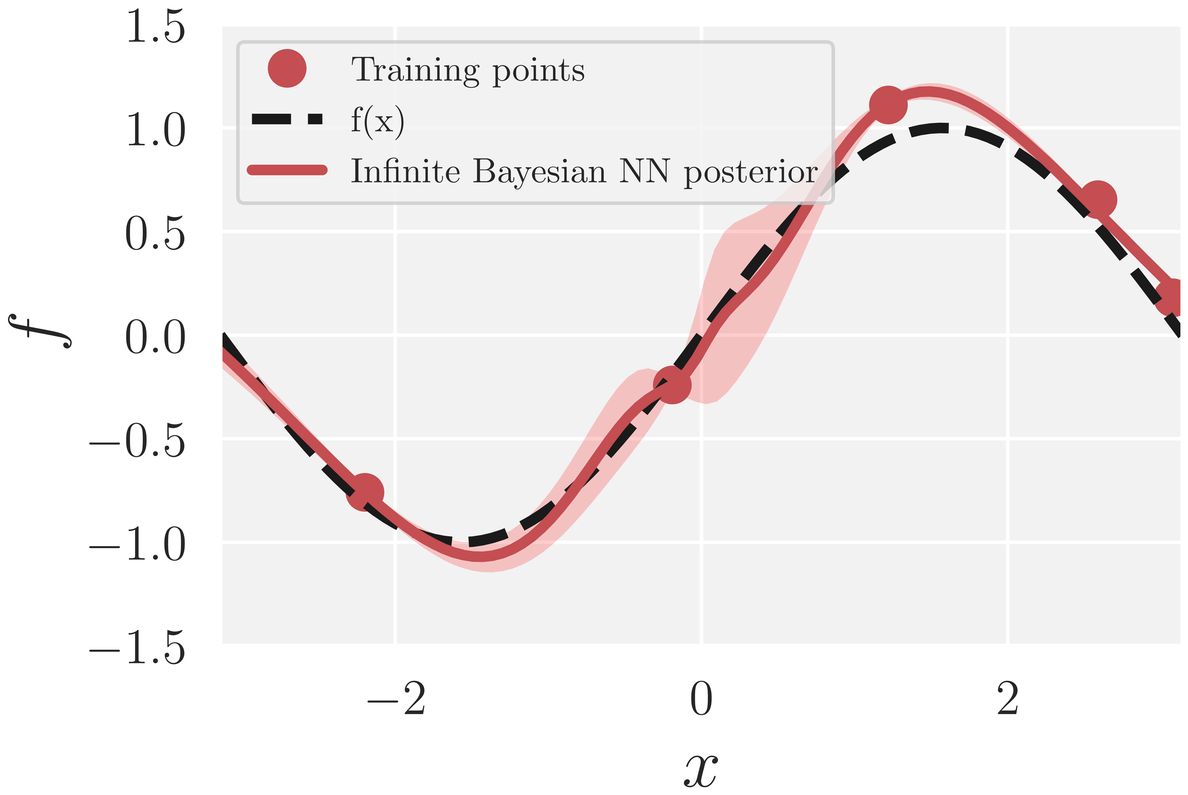
Prior:



How to use Neural Tangents?

A. Perform Bayesian inference with an infinite NN

Posterior:



How to use Neural Tangents?

B. Train the infinite NN with gradient descent (GD):

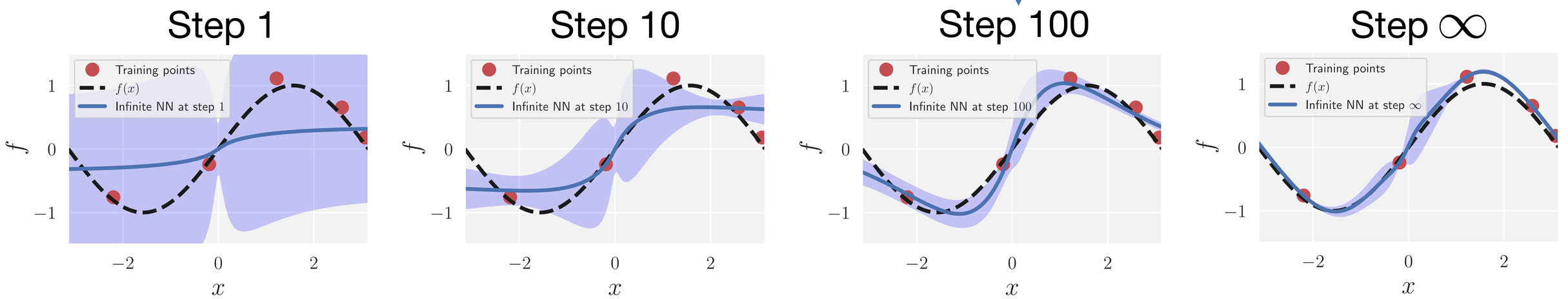
```
predict_fn = predict.gradient_descent_mse_gp(kernel_fn, x_train, f_train, x_test,
                                             get='ntk', diag_reg=1e-4, compute_cov=True)
f_mean_t, f_cov_t = predict_fn(t=100)
```

$$f_t^* \sim \mathcal{N}\left(\Theta\left(x^*, \mathcal{X}\right)\Theta^{-1}\left(I - e^{-\eta\Theta t}\right)\mathcal{Y}, \quad \ldots\right)$$

How to use Neural Tangents?

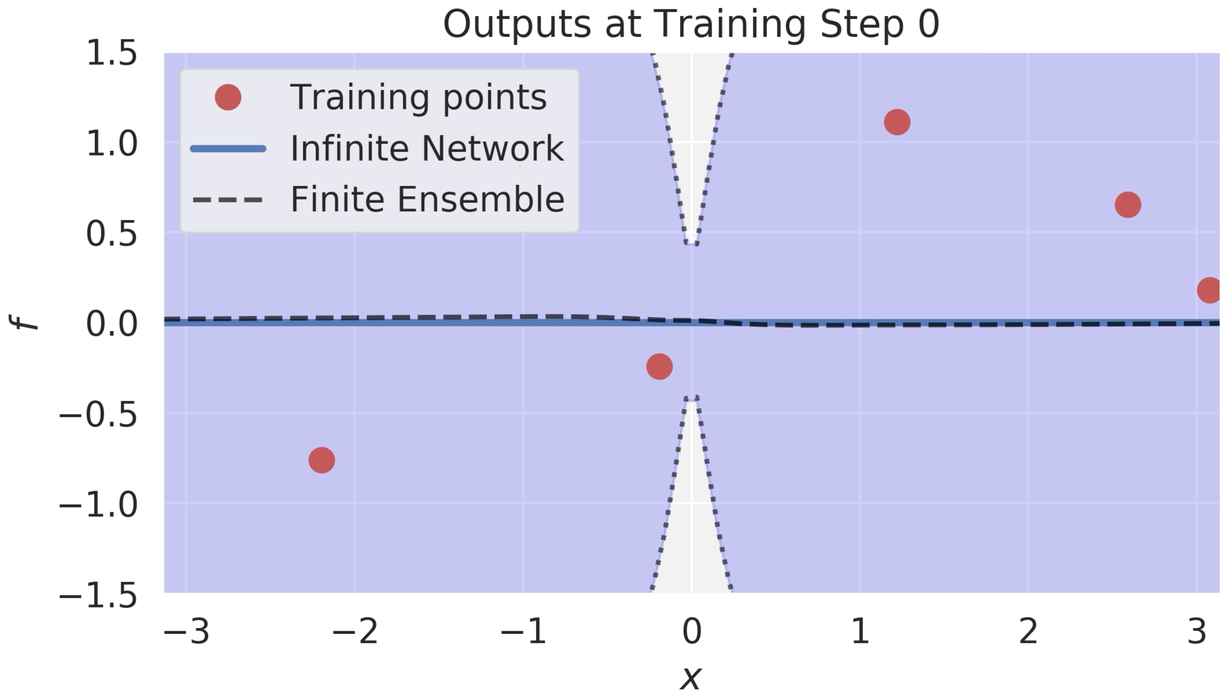
B. Train the infinite NN with gradient descent (GD):

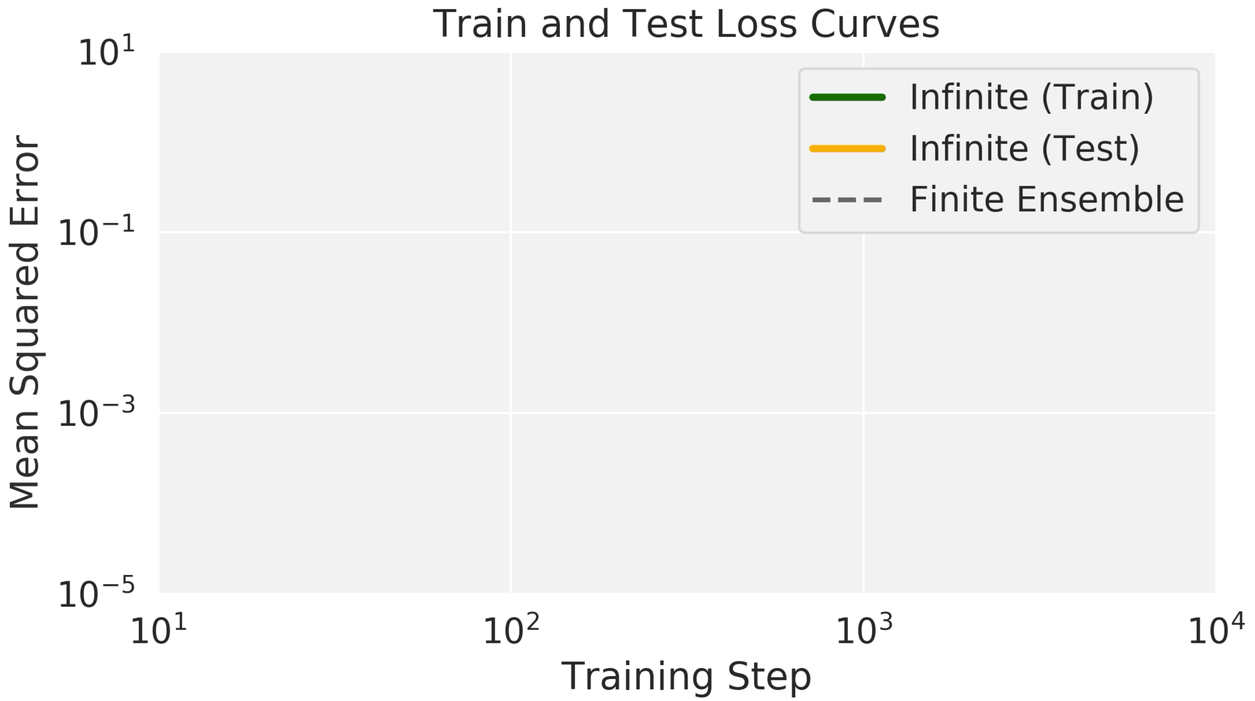
```
predict_fn = predict.gradient_descent_mse_gp(kernel_fn, x_train, f_train, x_test,
                                             get='ntk', diag_reg=1e-4, compute_cov=True)
f_mean_t, f_cov_t = predict_fn(t=100)
```



How to use Neural Tangents?

B. Train the infinite NN with gradient descent (GD):





Example 1: infinite FCN

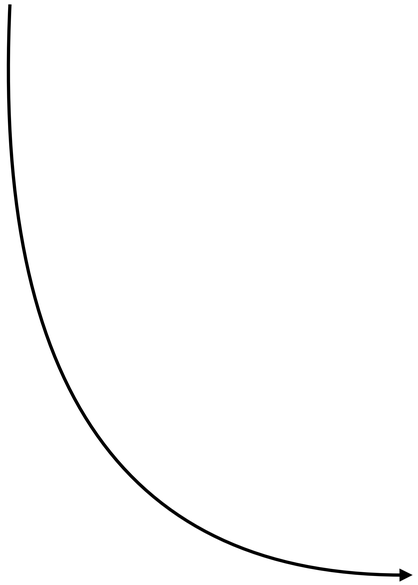
```
from neural_tangents import stax

def FullyConnectedNetwork(depth, W_std=1.0, b_std=0.0):
  layers = [stax.Flatten()]
  for _ in range(depth):
     layers += [stax.Dense(1, W_std, b_std), stax.Relu()]
  layers += [stax.Dense(1, W_std, b_std)]
  return stax.serial(*layers)
```

Example 2: infinite CNN

```
from neural_tangents import stax

def ConvolutionalNetwork(depth, W_std=1.0, b_std=0.0):
  layers = []
  for _ in range(depth):
    layers += [stax.Conv(1, (3, 3), W_std, b_std, padding='SAME'), stax.Relu()]
  layers += [stax.Flatten(), stax.Dense(1, W_std, b_std)]
  return stax.serial(*layers)
```

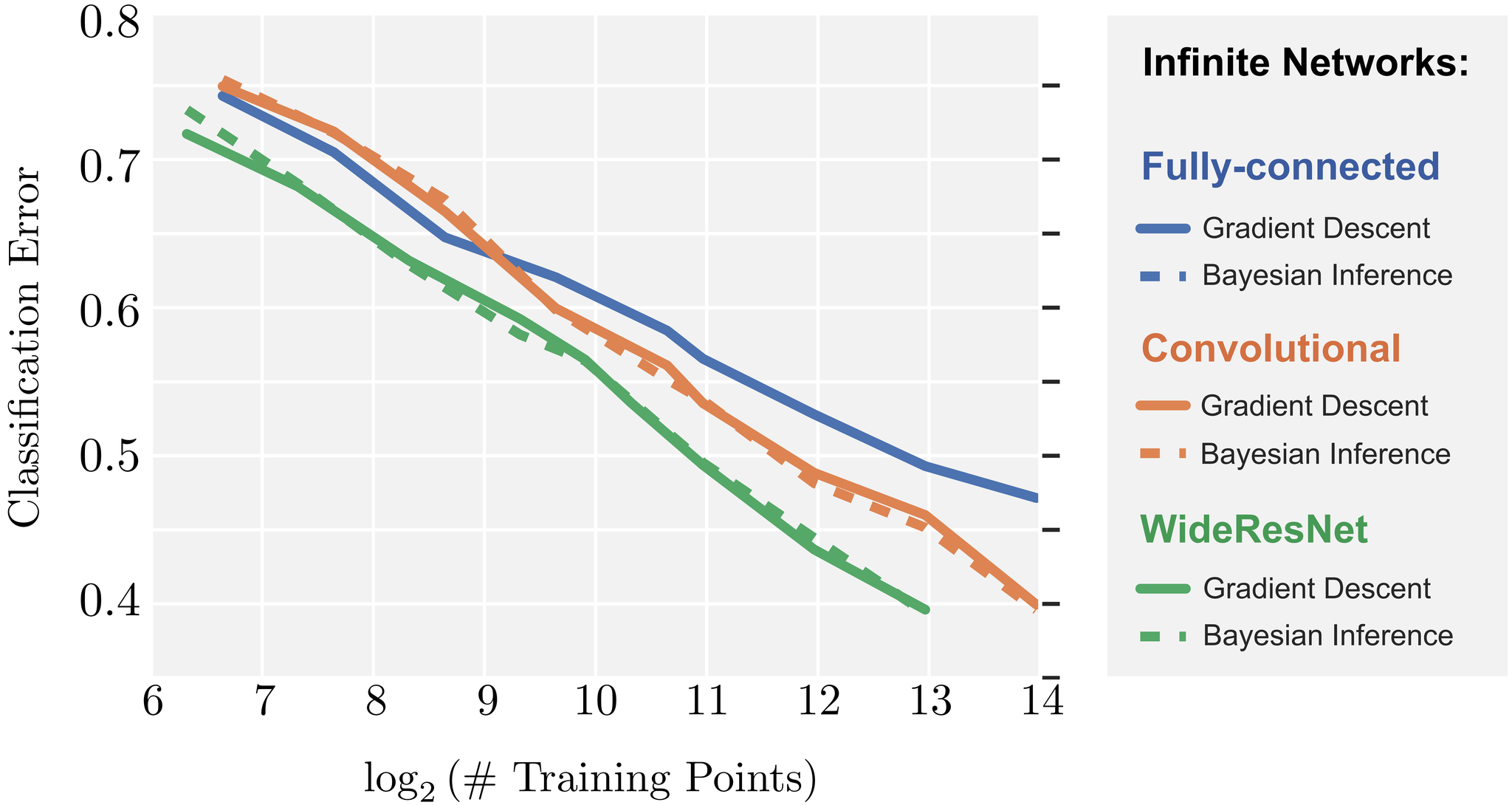
Example 3: infinitely WideResNet

No overhead for defining the infinite model(s), which are defined concurrently with the finite network.

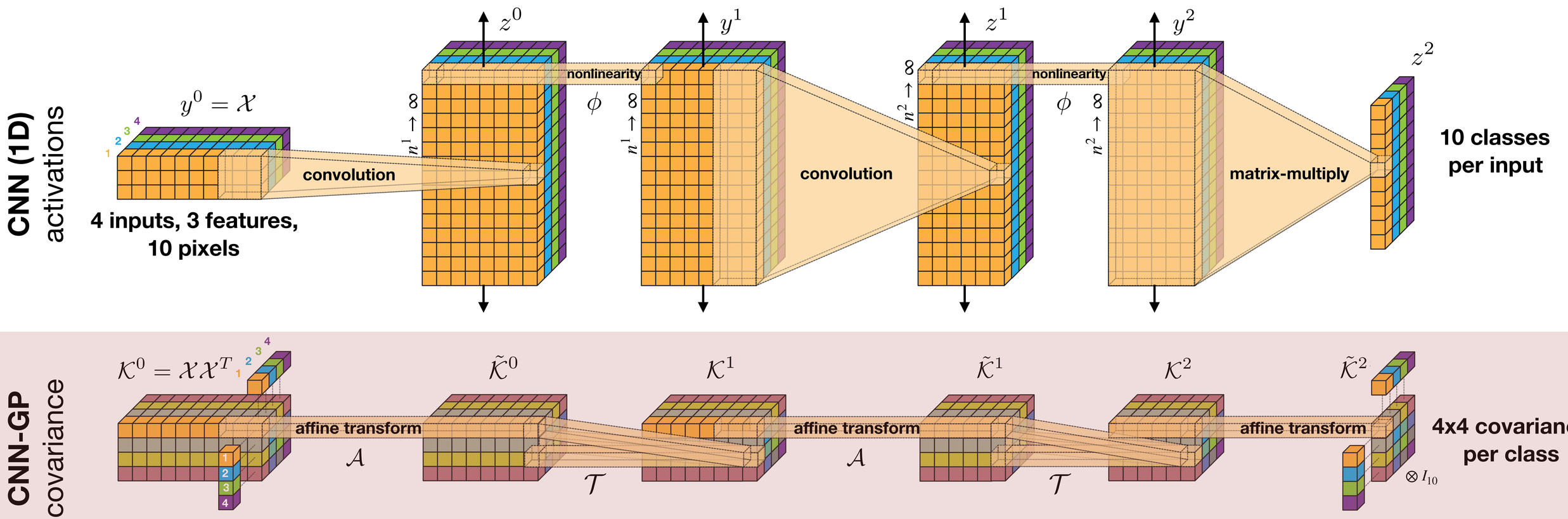


```
from neural_tangents import stax

def WideResNetBlock(channels, strides=(1, 1), channel_mismatch=False):
  Main = stax.serial(stax.Relu(), stax.Conv(channels, (3, 3), strides, padding='SAME'),
                     stax.Relu(), stax.Conv(channels, (3, 3), padding='SAME'))
  Shortcut = (stax.Identity() if not channel_mismatch else
              stax.Conv(channels, (3, 3), strides, padding='SAME'))
  return stax.serial(stax.FanOut(2), stax.parallel(Main, Shortcut), stax.FanInSum())

def WideResNetGroup(n, channels, strides=(1, 1)):
  blocks = [WideResNetBlock(channels, strides, channel_mismatch=True)]
  for _ in range(n - 1):
    blocks += [WideResNetBlock(channels, (1, 1))]
  return stax.serial(*blocks)

def WideResNet(block_size, k, num_classes):
  return stax.serial(stax.Conv(16, (3, 3), padding='SAME'),
                     WideResNetGroup(block_size, int(16 * k)),
                     WideResNetGroup(block_size, int(32 * k), (2, 2)),
                     WideResNetGroup(block_size, int(64 * k), (2, 2)),
                     stax.GlobalAvgPool(), stax.Dense(num_classes))

init_fn, apply_fn, kernel_fn = WideResNet(block_size=4, k=1, num_classes=10)
```

Model comparison



How does Neural Tangents work?



Infinite Bayesian NN covariance

What is ready in Neural Tangents?

- `serial`, `parallel`
- `FanOut`, `FanInSum`
- `Dense`, `Conv`
- `Relu`, `LeakyRelu`, `Abs`, `ABRelu` , `Erf`, `Identity`
- `Flatten`, `AvgPool`, `GlobalAvgPool`, `GlobalSelfAttention`
- `LayerNorm`

What is not in Neural Tangents?

- `Sigmoid`, `Tanh`, `Swish`, `Softmax`, `LogSoftMax`, `Softplus`, `MaxPool`

…

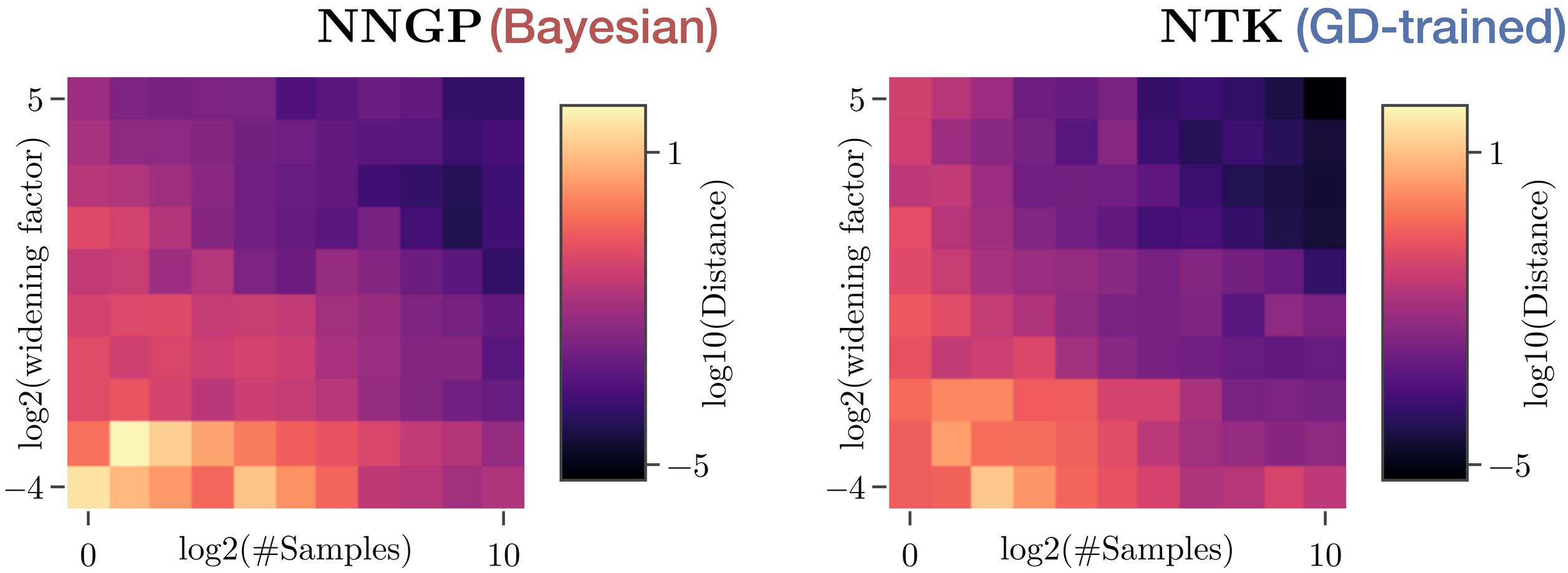
Many layers admit an infinite width limit,
but **have no known closed-form covariance expression**.

Infinite networks by sampling finite ones

Solution: use `nt.monte_carlo_kernel_fn` to sample finite-width networks to estimate the covariances of the infinite models.

```
from jax import random
from jax.experimental import stax
import neural_tangents as nt


init_fn, apply_fn = stax.serial(stax.Dense(64), stax.BatchNorm(), stax.Sigmoid, stax.Dense(1))
kernel_fn = nt.monte_carlo_kernel_fn(init_fn, apply_fn, key=random.PRNGKey(1), n_samples=128)
kernel = kernel_fn(x_train, x_train)
```

Infinite networks by sampling finite ones



WideResNet converges to its infinite limit as it becomes wider (↑) or more samples (→) are drawn.

```
$ ls neural_tangents/
```

$ ls neural_tangents/

stax.py

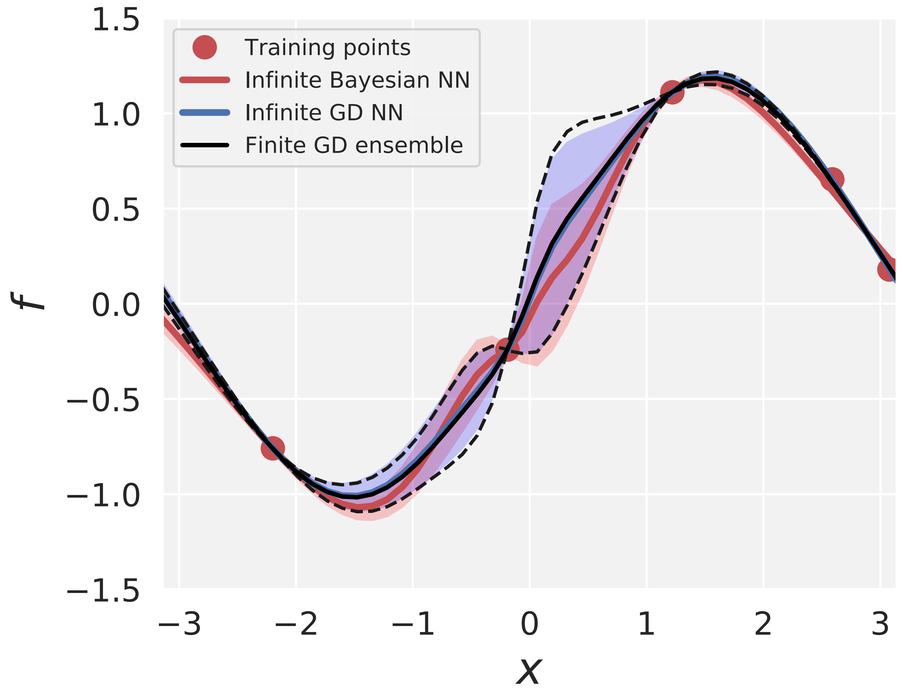
```
from neural_tangents import stax

def WideResNetBlock(channels, strides=(1, 1), channel_mismatch=False):
  Main = stax.serial(stax.Relu(), stax.Conv(channels, (3, 3), strides, padding='SAME'),
                     stax.Relu(), stax.Conv(channels, (3, 3), padding='SAME'))
  Shortcut = (stax.Identity() if not channel_mismatch else
              stax.Conv(channels, (3, 3), strides, padding='SAME'))
  return stax.serial(stax.FanOut(2), stax.parallel(Main, Shortcut), stax.FanInSum())

def WideResNetGroup(n, channels, strides=(1, 1)):
  blocks = [WideResNetBlock(channels, strides, channel_mismatch=True)]
  for _ in range(n - 1):
    blocks += [WideResNetBlock(channels, (1, 1))]
  return stax.serial(*blocks)

def WideResNet(block_size, k, num_classes):
  return stax.serial(stax.Conv(16, (3, 3), padding='SAME'),
                     WideResNetGroup(block_size, int(16 * k)),
                     WideResNetGroup(block_size, int(32 * k), (2, 2)),
                     WideResNetGroup(block_size, int(64 * k), (2, 2)),
                     stax.GlobalAvgPool(), stax.Dense(num_classes))

init_fn, apply_fn, kernel_fn = WideResNet(block_size=4, k=1, num_classes=10)
```

$ ls neural_tangents/

stax.py

```python
from neural_tangents import stax

def WideResNetBlock(channels, strides=(1, 1), channel_mismatch=False):
  Main = stax.serial(stax.Relu(), stax.Conv(channels, (3, 3), strides, padding='SAME'),
                     stax.Relu(), stax.Conv(channels, (3, 3), padding='SAME'))
  Shortcut = (stax.Identity() if not channel_mismatch else
              stax.Conv(channels, (3, 3), strides, padding='SAME'))
  return stax.serial(stax.FanOut(2), stax.parallel(Main, Shortcut), stax.FanInSum())

def WideResNetGroup(n, channels, strides=(1, 1)):
  blocks = [WideResNetBlock(channels, strides, channel_mismatch=True)]
  for _ in range(n - 1):
    blocks += [WideResNetBlock(channels, (1, 1))]
  return stax.serial(*blocks)

def WideResNet(block_size, k, num_classes):
  return stax.serial(stax.Conv(16, (3, 3), padding='SAME'),
                     WideResNetGroup(block_size, int(16 * k)),
                     WideResNetGroup(block_size, int(32 * k), (2, 2)),
                     WideResNetGroup(block_size, int(64 * k), (2, 2)),
                     stax.GlobalAvgPool(), stax.Dense(num_classes))

init_fn, apply_fn, kernel_fn = WideResNet(block_size=4, k=1, num_classes=10)
```

predict.py

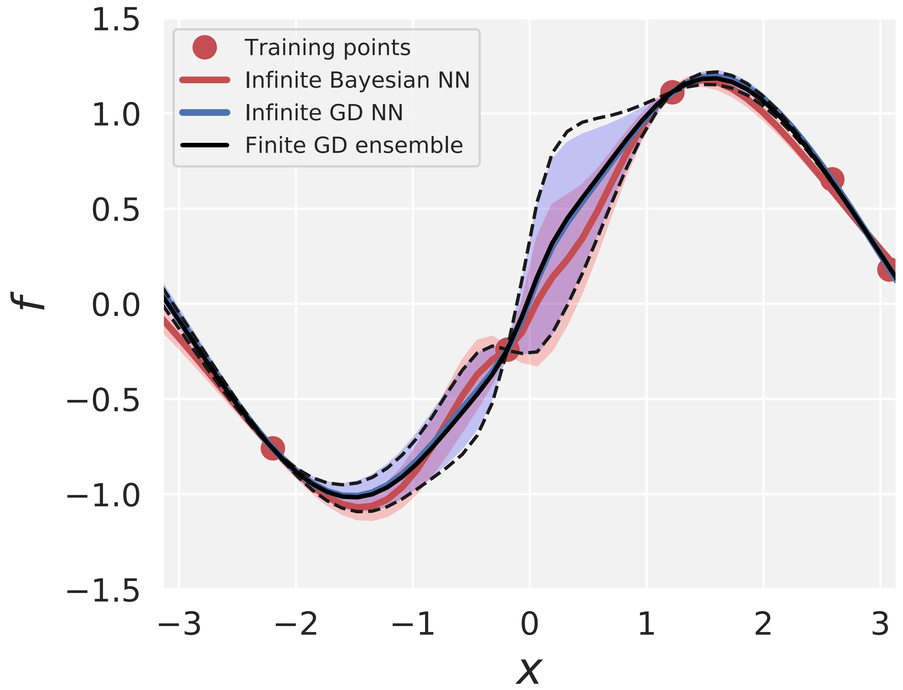


$ ls neural_tangents/

stax.py

```python
from neural_tangents import stax

def WideResNetBlock(channels, strides=(1, 1), channel_mismatch=False):
  Main = stax.serial(stax.Relu(), stax.Conv(channels, (3, 3), strides, padding='SAME'),
                     stax.Relu(), stax.Conv(channels, (3, 3), padding='SAME'))
  Shortcut = (stax.Identity() if not channel_mismatch else
              stax.Conv(channels, (3, 3), strides, padding='SAME'))
  return stax.serial(stax.FanOut(2), stax.parallel(Main, Shortcut), stax.FanInSum())

def WideResNetGroup(n, channels, strides=(1, 1)):
  blocks = [WideResNetBlock(channels, strides, channel_mismatch=True)]
  for _ in range(n - 1):
    blocks += [WideResNetBlock(channels, (1, 1))]
  return stax.serial(*blocks)

def WideResNet(block_size, k, num_classes):
  return stax.serial(stax.Conv(16, (3, 3), padding='SAME'),
                     WideResNetGroup(block_size, int(16 * k)),
                     WideResNetGroup(block_size, int(32 * k), (2, 2)),
                     WideResNetGroup(block_size, int(64 * k), (2, 2)),
                     stax.GlobalAvgPool(), stax.Dense(num_classes))

init_fn, apply_fn, kernel_fn = WideResNet(block_size=4, k=1, num_classes=10)
```

predict.py

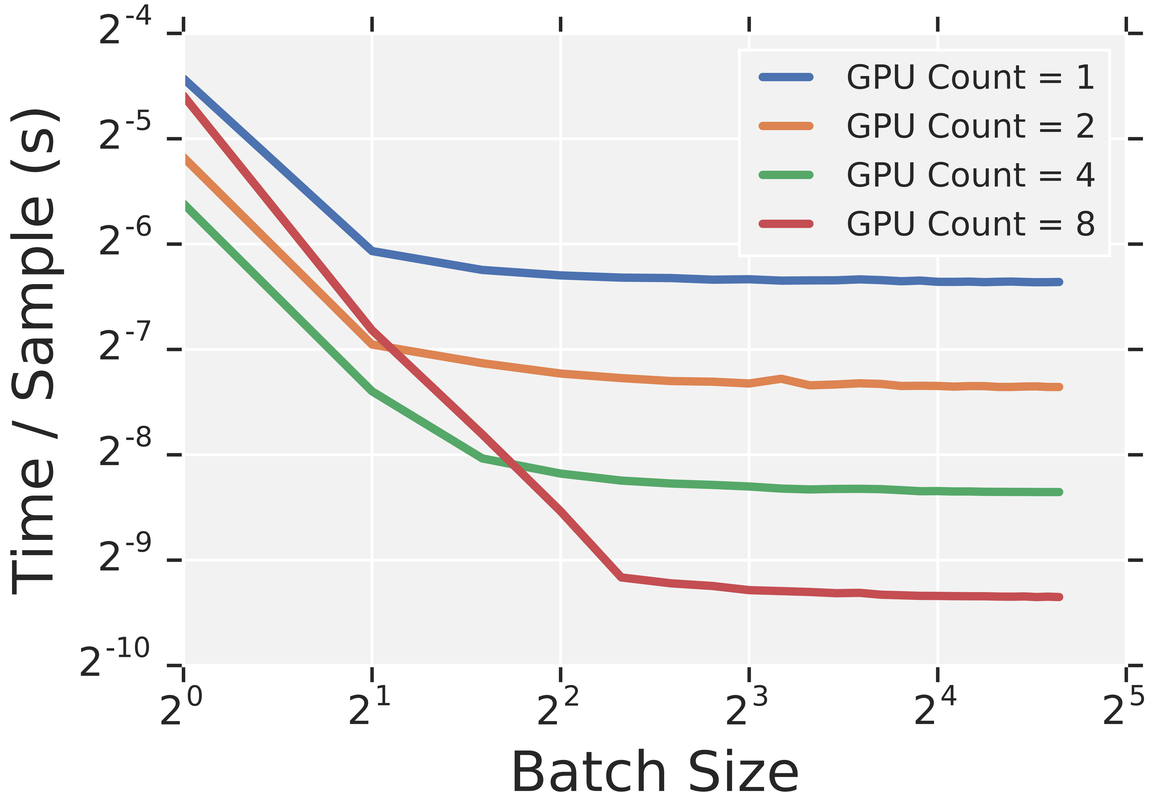


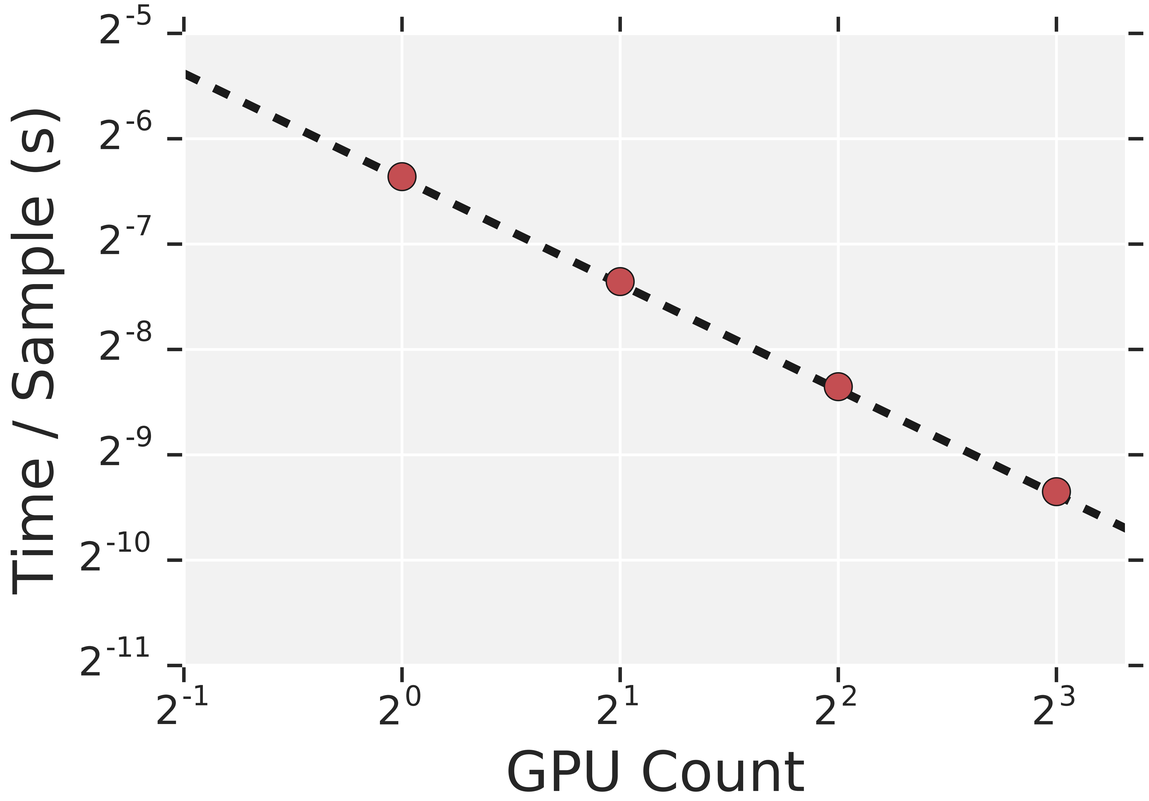
utils/

- `nt.batch`
- `nt.monte_carlo_kernel_fn`
- `nt.linearize`
- `nt.taylor_expand`
- `nt.empirical_kernel_fn`

Infinite networks in batches, in parallel

```
batched_kernel_fn = nt.batch(kernel_fn, batch_size)
batched_kernel_fn(x, x) == kernel_fn(x, x)  # True!
```





Expectation management

Architecture → Dataset size ↓	**Fully-connected**	**CNNs**	**CNNs w/ pooling**
O(100)			
O(10,000)	CIFAR10: **O(0.1) GPU-hours**	CIFAR10: **O(1) GPU-hours**	CIFAR10: **O(1000) GPU-hours**
O(1,000,000)			

github.com/google/neural-tangents