Random Fourier Features

MATH7339

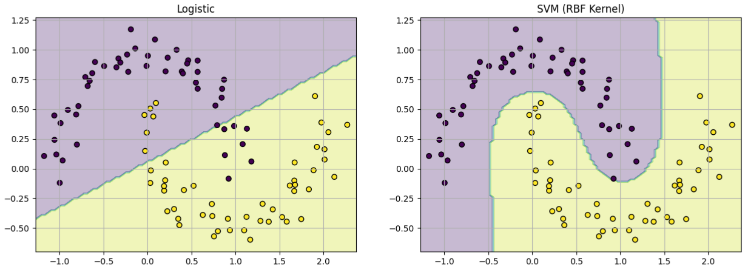
Harris Bubalo

February 25, 2025

Kernel Machines

- **Kernel machines**, the most popular of which being the *Support Vector Machine* (SVM), are a powerful option for a variety of learning tasks.
- This is particularly true when linear models would otherwise fail.

SVM Example


Logistic
SVM (RBF Kernel)
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

The Kernel Trick

The **kernel trick** allows us to apply a linear model in a higher-dimensional space without explicitly computing the transformation.

- Define a **feature map** $\phi : \mathbb{R}^d \to \mathcal{H}$, mapping inputs into a high-dimensional (possibly infinite) space.
- Compute inner products in $\mathcal{H}$ using a **kernel function** $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$$

- **Avoid explicit computation of** $\phi(x)$ by directly working with $k(x, x')$.

Computational Limitations

Consider the Gram matrix:

$$K = \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_n) \\ \dots & \dots & \dots \\ k(x_n, x_1) & \dots & k(x_n, x_n) \end{bmatrix}$$

Computational Cost:

- Computing K requires $O(n^2)$ kernel evaluations.
- Memory complexity is $O(n^2)$.
- Inverting K (e.g., in Gaussian Processes) requires $O(n^3)$ operations.
- For large values of n, we run into scaling issues.

What Can We Do?

- What if we could approximate the kernel $k(x, x')$ using some explicit feature map?

What Can We Do?

- What if we could approximate the kernel $k(x, x')$ using some explicit feature map?
- What if we could do this in a way that is both performant and scalable?

What Can We Do?

- What if we could approximate the kernel $k(x, x')$ using some explicit feature map?
- What if we could do this in a way that is both performant and scalable?
- **Solution:** Random Fourier Features

Some Context

- Random Fourier Features (RFFs) were first introduced by Benjamin Recht and Ali Rahimi in their seminal work, *Random Features for Large-Scale Kernel Machines* (2007).
- This work won them the Test of Time Award at NeurIPS in 2017

Prerequisite: Fourier Transform

- The Fourier transform converts a function from the time domain to frequency domain.

Definition (Fourier Transform)

For a function $f(x)$, the continuous Fourier transform $F(\xi)$ is the complex-valued function:

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-i\xi x} dx$$

Prerequisite: Shift-Invariant Kernels

- A kernel $k(x, y)$ is shift-invariant (also called stationary) if it depends only on the difference of its arguments, rather than their absolute positions:

$$k(x, y) = k(x - y) = k(\delta)$$

where $\delta = x - y$

Examples

$$\text{Gaussian (RBF):} \quad k(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

Bochner's Theorem

- Bochner's theorem characterizes shift-invariant positive definite kernels.

Theorem (Bochner's)

A continuous, shift-invariant kernel $k(x, y) = k(x - y)$ on $\mathbb{R}^d$ is positive definite if and only if it is the Fourier transform of a non-negative measure.

- With this, such kernels can be written as...

$$k(x - y) = \int p(\omega) e^{i\omega^T (x-y)} d\omega$$

- ...where $p(\omega)$ is a probability measure/distribution.
- This theorem comes from classic harmonic analysis.

Time to Approximate

Let's expand more on the previous form.

$$\begin{aligned} k(x,y) &= k(x-y) \\ &= \int p(\omega) e^{i\omega^T(x-y)} d\omega \\ &= E_\omega[\exp(i\omega^T(x-y))] \end{aligned}$$

Note that this means that $\exp(i\omega^T(x-y))$ is an unbiased estimator of our kernel $k(x,y)$ when ω is drawn from p. We can then use a **Monte Carlo approximation** to approximate the above expectation with lowered variance:

$$\begin{aligned} k(x,y) &= E_\omega[\exp(i\omega^T(x-y))] \\ &\approx \frac{1}{M}\sum_{j=1}^{M} \exp(i\omega_j^T(x-y)) \\ &= f(x)^T f(y)^* \end{aligned}$$

Let's Get Real

Now, we note that both our kernel $k(x, y)$ and our probability distribution $p(\omega)$ are real-valued, so we can do the following using Euler's formula:

$$\begin{aligned}\exp(i\omega^T(x-y)) &= \cos(\omega^T(x-y)) - i\sin(\omega^T(x-y)) \\ &= \cos(\omega^T(x-y))\end{aligned}$$

We can then define a function $z_\omega(x)$ where...

$$\omega \sim p(\omega)$$

$$b \sim \text{Uniform}(0, 2\pi)$$

$$z_\omega(x) = \sqrt{2}\cos(\omega^T x + b)$$

Some Trigonometry

Why are we doing this?

Using the fact that $2\cos(a)\cos(b) = \cos(a+b) + \cos(a-b)$, we can show the following:

$$\begin{aligned} E_\omega[z_\omega(x)z_\omega(y)] &= E_\omega[\sqrt{2}\cos(\omega^T x + b)\sqrt{2}\cos(\omega^T y + b)] \\ &= E_\omega[\cos(\omega^T(x+y) + 2b)] + E_\omega[\cos(\omega^T(x-y))] \\ &= E_\omega[\cos(\omega^T(x-y))] \end{aligned}$$

Putting it Together

We are now ready to define the RFF map we were aiming for all along!
Let $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^M$ be defined as

$$\mathbf{z}(x) = \begin{bmatrix} \frac{1}{\sqrt{M}} z_{\omega_1}(x) \\ \vdots \\ \frac{1}{\sqrt{M}} z_{\omega_M}(x) \end{bmatrix}$$

With this, we have...

$$\begin{aligned} \mathbf{z}(x)^T \mathbf{z}(y) &= \frac{1}{M} \sum_{j=1}^{M} z_{\omega_j}(x) z_{\omega_j}(y) \\ &= \frac{1}{M} \sum_{j=1}^{M} 2\cos(\omega_j^T x + b_j)\cos(\omega_j^T y + b_j) \\ &\approx E_\omega[\cos(\omega^T (x - y))] \\ &= k(x, y) \end{aligned}$$

The Result

There we have it! We have found a randomized map $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^M$ where

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}} \approx \mathbf{z}(x)^T \mathbf{z}(x')$$

For values of $M \ll n$, working with RFFs has complexity $O(nM)$, a nice improvement on what we had initially with the kernel trick!

The RFF Algorithm

Working with RFFs looks like the following:

1. Decide which shift-invariant kernel you would like to use for your purposes. (e.g. Gaussian)
2. Generate M by d random samples of $\omega \sim p(\omega)$ and M random samples of $b \sim \text{Uniform}(0, 2\pi)$.
3. Compute the mapped dataset $Z = \sqrt{\frac{2}{M}} \cos(XW^T + B)$.

$$X = \underbrace{\begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix}}_{\text{n x d}} \qquad W = \underbrace{\begin{bmatrix} w_1^T \\ \vdots \\ w_M^T \end{bmatrix}}_{\text{m x d}} \qquad B = \underbrace{\begin{bmatrix} b_1 & b_2 & \dots & b_M \\ \vdots & \vdots & \vdots & \vdots \\ b_1 & b_2 & \dots & b_M \end{bmatrix}}_{\text{n x M}}$$

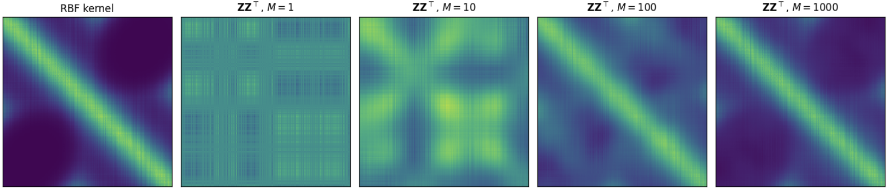
Where are we Sampling From?

- You may be wondering what probability distribution $p(\omega)$ represents.
- Note that it is not something we can choose arbitrarily; it is tied directly to the chosen kernel $k(x - y)$ via Bochner's Theorem.

Kernel Name	$k(\Delta)$	$p(\omega)$
Gaussian	$e^{-\frac{\|\Delta\|_2^2}{2}}$	$(2\pi)^{-\frac{D}{2}} e^{-\frac{\|\omega\|_2^2}{2}}$
Laplacian	$e^{-\|\Delta\|_1}$	$\prod_d \frac{1}{\pi(1+\omega_d^2)}$
Cauchy	$\prod_d \frac{2}{1+\Delta_d^2}$	$e^{-\|\Delta\|_1}$

RFFs In Action

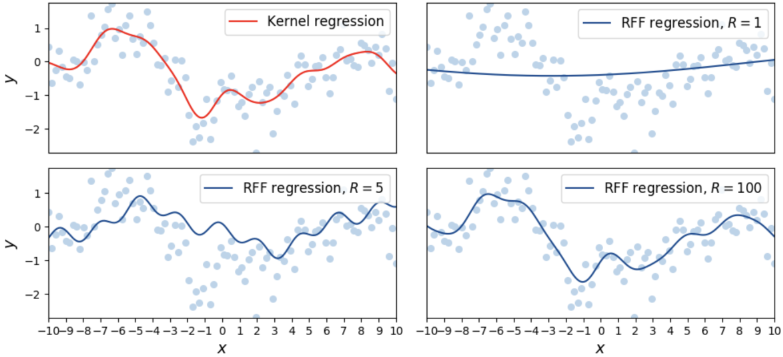
- First, let's verify that RFFs are indeed a good approximation of the true kernel in practice.
- Using a matplotlib visualization of each approximate matrix ZZ^T, we see that they get pretty close to the exact RBF kernel as we increase M.



How do RFFs perform?

- On a dataset with 100000 points and 20 features, how do RFFs compare to the true kernel?
- Generated data with make_classification in sklearn and performed a 75/25 train test split.
- RBF SVM took about 33 seconds to train, 13 to test, and achieved an accuracy score of 0.889.
- RFF method ($M = 800$) took 10 seconds to train, 0.06 seconds to test, and achieved an accuracy score of 0.878.
- So using RFFs resulted in a similar accuracy score with a much faster running time!

Kernel Ridge Regression



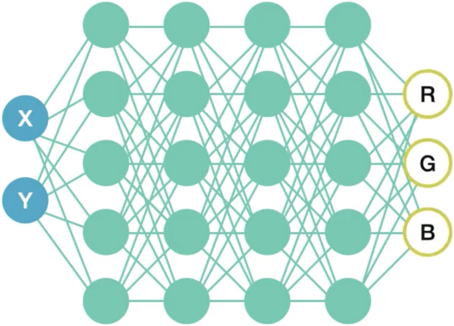
$$\hat{\beta} = \underbrace{(\mathbf{Z}_X^\top \mathbf{Z}_X + \lambda \mathbf{I}_R)}_{\mathbf{A}}^{-1} \mathbf{Z}_X^\top \mathbf{y}.$$

What Else Can RFFs Do?

- We have seen that RFFs can be used to obtain similar results to kernel methods in a fraction of the time.
- But is it also possible that RFF results can be *better* than results from unmapped data?
- As it turns out, it is!

Representing Images With an MLP

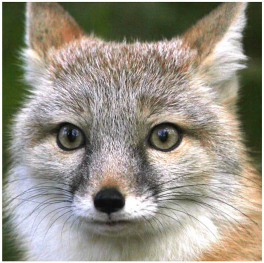
Say we'd like to use a Multi-Layered Perceptron (MLP) to represent an image, taking in as input pixel coordinates (x, y) and outputting color values (r, g, b).



High Frequency Signals in Low Dimensional Domains

- The MLP will struggle to capture high-frequency details in the image, such as sharp edges and fine textures.
- This is because it is operating in a low-dimensional domain, i.e. 2D coordinates.
- The phenomenon where NNs learn low-frequency components of a function faster than high-frequency components is called **spectral bias**.

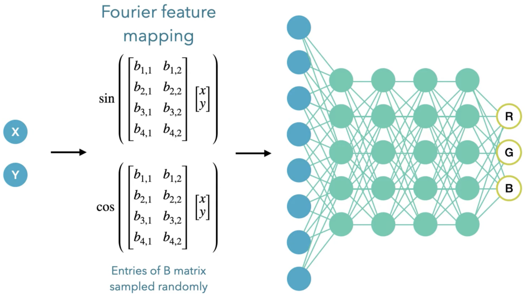
MLP output



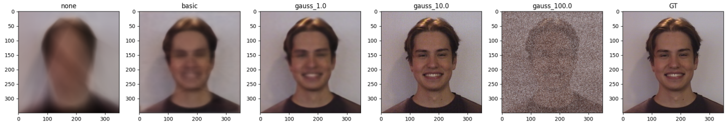
Supervision image

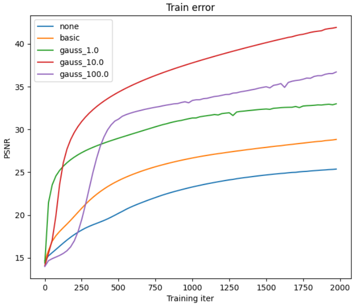
Representing Higher Frequency Functions Using RFFs

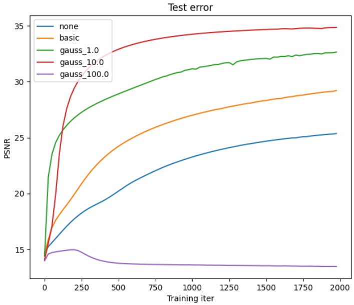
- How do we fix this? **By using RFFs, of course!**
- Let's change our MLP architecture. Instead of simply inputting the low-dimensional coordinates directly into the MLP, we can first apply a Fourier feature map.
- This allows the MLP to operate in a transformed space where high-frequency information is already present.



MLP Results With Fourier Features







Why Does This Work?

- Consider kernel regression, where a function $f(x)$ is approximated as:

$$\hat{f}(x) = \sum_{i=1}^{n} w_i k(x, x_i)$$

- It can be shown that training a neural network with gradient descent becomes the same as performing kernel regression as the width of each layer approaches infinity.
- More specifically, it converges over the course of training to the kernel regression solution obtained when using a special kernel called the *neural tangent kernel*.
- Fourier feature mapping lets us change the width of the NTK, which in turn changes how well the MLP is able to learn different frequency components.

What is in Store for RFFs?

- We have shown that Random Fourier features can be a powerful and highly scalable tool.
  - Effectively approximates kernels and achieves great accuracy
  - Reduces the computational complexity of kernel methods from $O(n^2)$ to $O(nM)$ for $M \ll n$
  - Enriches NNs by enabling them to learn high-frequency information
- However, there is still so much more to learn and discover regarding Fourier features.
  - Learned Fourier Features
  - Extending to different kernel types (asymmetric, non-stationary, etc.)

References

- Rahimi, A., & Recht, B. (2007). Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., ... & Barron, J. T. (2020). Fourier features let networks learn high frequency functions in low dimensional domains. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Jang, W., Lee, S., Kim, K., & Moon, I.-C. (2021). Learnable Fourier features for multi-dimensional spatial positional encoding. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Gundersen, G. (2019, December 23). Random Fourier Features. *Blog post*. Retrieved from `https://gregorygundersen.com/blog/2019/12/23/random-fourier-features`.
- Fabien, M. (n.d.). Large Scale Kernel Methods. *Blog post*. Retrieved from `https://maelfabien.github.io/machinelearning/largescale/#`.