Deep Neutral Networks

Homework 0

Yuanteng Chen

1. Gradient Descent Doesn't go nuts with ill-conditions.

Show that for $t>0$, $\|w_t\|_2 \leq \|w_{t-1}\|_2 + \eta a\|y\|_2$:

Sollution:

$$W_t = W_{t-1} - \eta (F^T(FW_{t-1} - y))$$

acording to the assumption: learning rate η is small enough that gradient descent cannot possibly diverse and the Hint $(E-\eta F^T F)$.

first I make an assumption that singular value of $(E-\eta F^T F)$ is less than a specific number M

then I need to convert the original to a formula containing $(E-\eta F^T F)$ so that singular value of $(E-\eta F^T F) < M$ can be used.

$$\|w_t\|_2 = \|W_{t-1} - \eta (F^T(FW_{t-1} - y))\|_2$$

$= \| W_{t-1} - \eta F^T F W_{t-1} + \eta F^T y \|_2$

$= \| (E - \eta F^T F) W_{t-1} + \eta F^T y \|$

(We know that $\|A+B\|_2 \leq \|A\|_2 + \|B\|_2$)

$\leq \| (E - \eta F^T F) W_{t-1} \|_2 + \| \eta F^T y \|_2$

as the target is $\|W_{t-1}\|_2 + \eta \alpha \|y\|_2$

the latter one is obvious; $\| \eta F^T y \|_2 \leq \eta \alpha \|y\|_2$

But if we want to prove $\| (E - \eta F^T F) W_{t-1} \|_2 \leq \|W_{t-1}\|_2$,

we must prove that specific number M is 1

that is singular value of $(E - \eta F^T F)$ is less that 1

(but I don't know how to prove it)

2. Regularization from the Augmentation Perspective

Show that the ordinary least squares problem

$\underset{w}{\operatorname{argmin}} \| \vec{y} - \hat{X} \vec{w} \|_2^2$ has the same solution as

$$\vec{w} = (X^T X + \Sigma^{-1})^{-1} X^T y$$

Solution: in Tikhonov regularization.

$$\underset{w}{\operatorname{argmin}} \| \vec{y} - \hat{X} \vec{w} \|^2 + w^T \Sigma^{-1} w$$

the MAP (Maximum A Posteriori) of w is:

$$w = (X^T X + \Sigma^{-1})^{-1} X^T y$$

in the ordinary least squares problem

$$\underset{w}{\operatorname{argmin}} \|\vec{y} - \hat{X} w\|_2^2$$

OLS is a commonly used method for fitting linear models and estimating model parameters:

$$\hat{y}_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \cdots + \beta_p X_{pi}$$

$$\hat{\beta} = (X^T X)^{-1} X^T y \quad (\hat{\beta} \text{ is parameter estimate})$$

when $\hat{X} = \begin{bmatrix} X \\ \Gamma \end{bmatrix} \in R^{(n+d)\times d}$ and $\hat{y} = \begin{bmatrix} y \\ 0_d \end{bmatrix} \in R^{n+d}$

$$\begin{aligned}
\hat{\beta} &= (X^T X)^{-1} X^T y \\
&= \left([X^T, \Gamma^T] \cdot \begin{bmatrix} X \\ \Gamma \end{bmatrix} \right)^{-1} [X^T, \Gamma^T] \begin{bmatrix} y \\ 0_d \end{bmatrix} \\
&= (X^T X + \Gamma^T \Gamma)^{-1} (X^T \cdot y + \Gamma^T \cdot 0_d) \\
&= (X^T X + \Gamma^T \Gamma)^{-1} X^T y + (X^T X + \Gamma^T \Gamma) \cdot \Gamma^T \cdot 0_d \\
&= (X^T X + \Sigma^{-1})^{-1} X^T y + \underbrace{(X^T X + \Sigma^{-1}) \cdot \Gamma^T \cdot 0_d}_{0} \\
&= (X^T X + \Sigma^{-1})^{-1} X^T y
\end{aligned}$$

3. Vector Calculus Review

$\vec{x}, \vec{c} \in R^n$, $A \in R^{n\times n}$

(a) show $\frac{\partial}{\partial x}(x^T c) = c^T$

Solution:

$$\frac{\partial}{\partial x}(x^T c) = \frac{\partial}{\partial x}\left(\sum_i x_i \cdot c_i\right)$$

$$= \left[\frac{\partial(\sum_i x_i \cdot c_i)}{\partial x_1}, \frac{\partial(\sum_i x_i \cdot c_i)}{\partial x_2}, \cdots, \frac{\partial(\sum_i x_i c_i)}{\partial x_n}\right]$$

$$= [c_1, c_2, \cdots c_n] = c^T$$

(b). show $\frac{\partial}{\partial x}\|x\|_2^2 = 2x^T$

Solution: $\|x\|_2^2 = x_1^2 + \cdots + x_n^2$

$$\frac{\partial}{\partial x}\|x\|_2^2 = \left[\frac{\partial\|x\|_2^2}{\partial x_1}, \cdots \frac{\partial\|x\|_2^2}{\partial x_n}\right]$$

$$= [2x_1, \cdots 2x_n]$$

$$= 2x^T$$

(c) show $\frac{\partial}{\partial x}(A\vec{x}) = A$

Solution: $\frac{\partial}{\partial x}(A\vec{x})$

$$= \frac{\partial}{\partial x}[A_1 \cdot \vec{x}, A_2 \cdot \vec{x}, \cdots A_n\vec{x}]^T$$

$$\because \frac{\partial([A_1 \cdot \vec{x}, A_2 \cdot \vec{x}, \cdots, A_n\vec{x}]^T)}{\partial x_i}$$

$$= [A_{1i}, A_{2i}, \cdots, A_{ni}]^T = A^i$$

$\therefore \frac{\partial}{\partial x}(A\vec{x}) = [A^1, A^2, \cdots A^i] = A$

(d) show $\frac{\partial}{\partial x}(x^T A x) = x^T(A+A^T)$

Solution:

$$x^T \cdot A \cdot x = [x_1, \cdots x_n] \begin{bmatrix} A_{11}, \cdots A_{1n} \\ \vdots \quad\quad \vdots \\ A_{n1}, \cdots A_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$= \left[\sum_{i=1}^{n} x_i \cdot A_{i1}, \cdots \sum_{i=1}^{n} x_i \cdot A_{in}\right] \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$= \sum_{j=1}^{n} x_j \cdot \left(\sum_{i=1}^{n} x_i \cdot A_{ij}\right)$$

$$= \sum_{j=1}^{n} \sum_{i=1}^{n} A_{ij} \cdot x_i \cdot x_j$$

$$\frac{\partial\left(\sum_{j=1}^{n}\sum_{i=1}^{n} A_{ij} \cdot x_i \cdot x_j\right)}{\partial x_h} \Leftarrow h \text{ column of } \frac{\partial}{\partial x}(x^T A x)$$

it's obvious that there are only three subsets not equal to zero:

$\sum_{j \neq h} A_{hj} \cdot x_h \cdot x_j$, $\sum_{i \neq h} A_{ih} x_i \cdot x_h$ and $A_{hh} \cdot x_h \cdot x_h$

$$\therefore = \frac{\partial}{\partial x_h}\left(\sum_{j \neq h} A_{hj} x_h \cdot x_j + \sum_{i \neq h} A_{ih} x_i \cdot x_h + A_{hh} \cdot x_h^2\right)$$

$$= \sum_{j \neq h} A_{hj} \cdot x_j + \sum_{i \neq h} A_{ih} \cdot x_i + 2A_{hh} \cdot x_h$$

$$= \sum_{j=1}^{n} A_{hj} \cdot x_j + \sum_{i=1}^{n} A_{ih} \cdot x_i$$

$$= x \cdot A_h + x^T \cdot A^h$$

$$= x^T \cdot A_h^T + x^T \cdot A^h = x^T(A_h^T + A^h)$$

$\therefore \frac{\partial}{\partial X}(X^T \cdot A \cdot X)$

$= [X^T(A_1^T + A^1), \cdots, X^T(A_n^T + A^n)]$

$= X^T[(A_1^T + A^1), \cdots, (A_n^T + A^n)]$

$= X^T(A^T + A)$

(e). Under what condition is the previous derivative equal to $2X^TA$

Solution: in (d) we have proved

$\frac{\partial}{\partial X}(X^T \cdot A \cdot X) = X^T(A + A^T)$

when $A^T = A$ (A is symmetric)

$\frac{\partial}{\partial X}(X^T \cdot A \cdot X) = 2X^TA$

4. ReLu ELbow Update under SGD.

(i) The location of the 'elbow':

Solution: the location of the 'elbow' is where $wx + b = 0 \Leftrightarrow x = -\frac{b}{w}$

(ii) The derivative of the loss w.r.t $\phi(x)$, namely $\frac{dL}{d\phi}$

Solution: $L(x, y, \phi) = \frac{1}{2}||\phi(x) - y||_2^2$

$\therefore \frac{\partial L}{d\phi} = \frac{\partial(\frac{1}{2}||\phi(x) - y||_2^2)}{d\phi}$

$= \frac{1}{2}(2\phi(x) - 2y)$

$= \phi(x) - y$

(iii) The partial derivative of the loss w.r.t. w, namely $\frac{\partial l}{\partial w}$

Solution: $\because$ According to the chain rule

$$\frac{\partial l}{\partial w} = \frac{\partial l}{\partial \phi} \cdot \frac{\partial \phi}{\partial w}$$

we have proved $\frac{\partial l}{\partial \phi} = \phi(x) - y$

$$\frac{\partial \phi}{\partial w} = \begin{cases} x, & wx+b>0 \\ 0, & \text{else} \end{cases}$$

$$\therefore \frac{\partial l}{\partial w} = \begin{cases} x(\phi(x)-y) & wx+b>0 \\ 0 & \text{else} \end{cases}$$

(iv) The partial derivative of the loss w.r.t. b. namely $\frac{\partial l}{\partial b}$

Solution. $\frac{\partial l}{\partial b} = \frac{\partial l}{\partial \phi} \cdot \frac{\partial \phi}{\partial b}$

$$\frac{\partial \phi}{\partial b} = \begin{cases} 1 & wx+b>0 \\ 0 & \text{else} \end{cases}$$

$$\therefore \frac{\partial l}{\partial b} = \begin{cases} \phi(x)-y & wx+b>0 \\ 0 & \text{else} \end{cases}$$

(b).

Describe what happens to the slope and elbow of $\phi(x)$ when we perform gradient descent in the following cases:

(i) $\phi(x) = 0$

Solution: after performing gradient descent:

$b' = b - \Delta b = b - \eta \frac{\partial L}{\partial b}$ (η is learning rate)

$w' = w - \Delta w = w - \eta \frac{\partial L}{\partial w}$

when $\phi(x) = 0$, $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial b} = 0$

so both slop and elbow have no changes

(ii) $w>0$, $x>0$, and $\phi(x)>0$.

$\phi(x) - y = 1$

$$\begin{cases} \frac{\partial L}{\partial w} = x \\ \frac{\partial L}{\partial b} = 1 \end{cases} \Rightarrow \begin{cases} w' = w - \eta x < w \\ b' = b - \eta \end{cases}$$

$\because$ $w' < w$ $\therefore$ the slope becomes slower.

since I'm not sure if $b>0$ or $b<0$

the changes of elbow can't be determined.

(iii) $w>0$, $x<0$. and $\phi(x)>0$

$$\begin{cases} w' = w - \eta x > w \Rightarrow \text{the slope becomes steeper} \\ b' = b - \eta > b. \end{cases}$$

$\because$ $wx + b > 0$ and $x<0$ $\therefore$ $b>0$.

$\therefore$ $e' = -\frac{b'}{w'} < \frac{b}{w} < 0$ $\therefore$ elbow moves left

(iv) $w<0$, $x>0$. and $\phi(x)>0$

$$\begin{cases} w' = w - \eta x < w < 0 & \therefore |w'| > |w| \\ b' = b - \eta < b & (b>0) \end{cases}$$

$|w'| > |w| \Rightarrow$ slope becomes steeper.

$0 < e' = -\frac{b'}{w'} < -\frac{b}{w} \Rightarrow$ elbow moves left

(c) Derive the location e_i of the elbow of the i'th elementwise ReLU activation.

Solution:

assume W_i is the weight of the i'th and b_i is the bais of the i'th

then elbow $= -\frac{b_i}{W_i}$

6. Homework Process and Study Group

(a) stack overflow, CSDN

(b) none

(c) $\begin{cases} \text{writing : 5 hours} \\ \text{code : 4 hours} \end{cases}$

$4+5=9$ hours in total

In [1]:

```
!pip install ipympl torchviz
!pip install torch==1.13 --extra-index-url https://download.pytorch.org/whl/cpu
# restart your runtime after this step
```

```
Collecting ipympl
  Downloading ipympl-0.9.3-py2.py3-none-any.whl (511 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 511.6/511.6 kB 5.3 MB/s eta 0:00:00a 0:00:01
Collecting torchviz
  Downloading torchviz-0.0.2.tar.gz (4.9 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: ipython<9 in /usr/local/lib/python3.10/dist-packages (from ipympl) (7.34.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from ipympl) (1.23.5)
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages (from ipympl) (0.2.0)
Requirement already satisfied: pillow in /usr/local/lib/python3.10/dist-packages (from ipympl) (9.4.0)
Requirement already satisfied: traitlets<6 in /usr/local/lib/python3.10/dist-packages (from ipympl) (5.7.1)
Requirement already satisfied: ipywidgets<9,>=7.6.0 in /usr/local/lib/python3.10/dist-packages (from ipympl) (7.7.1)
Requirement already satisfied: matplotlib<4,>=3.4.0 in /usr/local/lib/python3.10/dist-packages (from ipympl) (3.7.1)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from torchviz) (2.0.1+cu118)
Requirement already satisfied: graphviz in /usr/local/lib/python3.10/dist-packages (from torchviz) (0.20.1)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.10/dist-packages (from ipython<9->ipympl) (67.7.2)
Collecting jedi>=0.16 (from ipython<9->ipympl)
  Downloading jedi-0.19.0-py2.py3-none-any.whl (1.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 10.2 MB/s eta 0:00:00
```

In [2]:

```
import math
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from torch.autograd import Variable
import tqdm

import IPython
from ipywidgets import interactive, widgets, Layout
from IPython.display import display, HTML
```

In [3]:
```
print(torch.__version__, torch.cuda.is_available())
# Homework 0 does not require a GPU
```

```
1.13.0+cpu False
```

In [4]:
```
# enable matplotlib widgets;

# on Google Colab
from google.colab import output
output.enable_custom_widget_manager()

%matplotlib widget
```

In [5]:
```
# Constants
cap_value = 1e-6          # Farads
R_init = 500              # Ohms
cutoff_mag = 1. / math.sqrt(2)
cutoff_dB = 20 * math.log10(cutoff_mag)
dataset_size = 1000
max_training_steps = 100000
```

In [6]:
```
print(cutoff_dB)
```

```
-3.0102999566398125
```

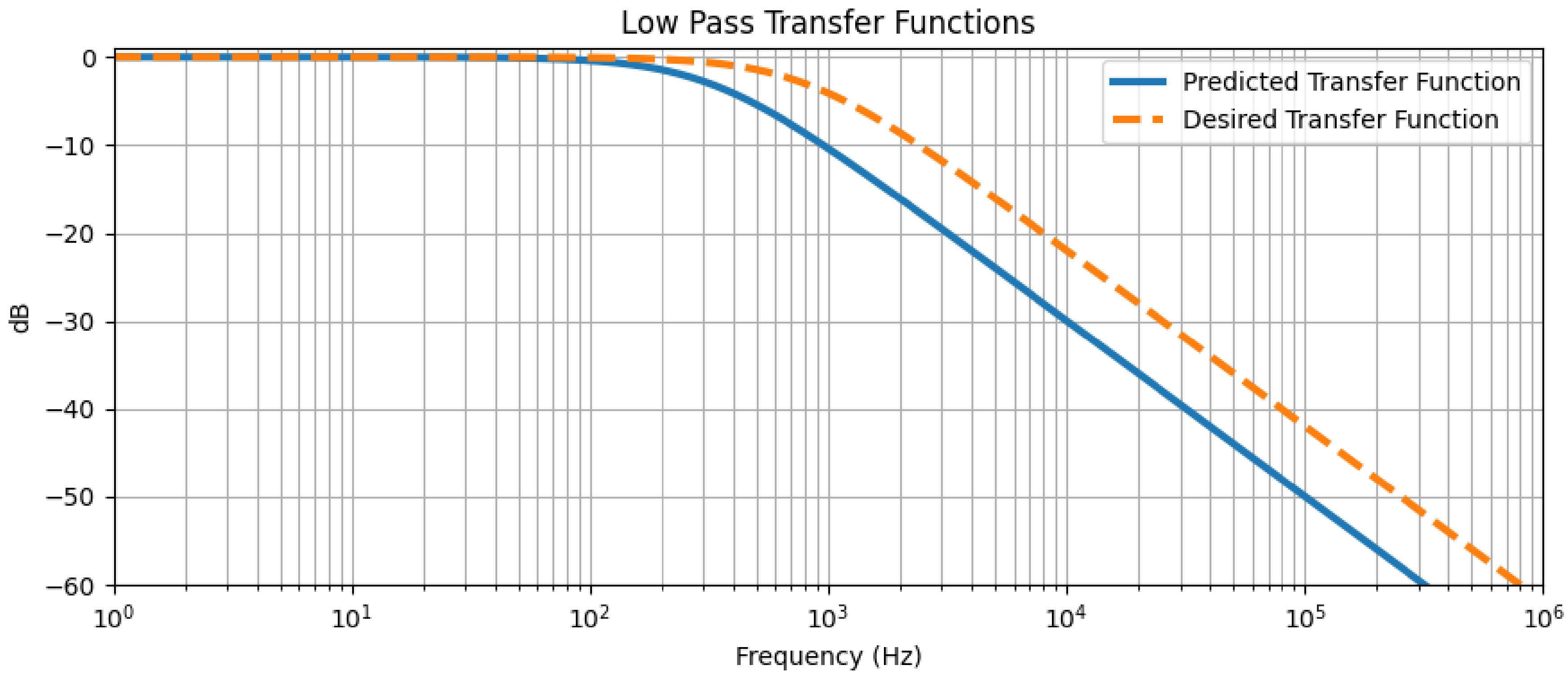
(a) Designing a Low Pass Filter by Matching Transfer Functions

In [7]:
```
# Transfer function: evaluates magnitude of given frequencies for a resistor value in the low pass circuit
def evaluate_lp_circuit(freqs, R_low):
    return 1. / torch.sqrt(1 + (R_low * cap_value * freqs) ** 2)
```

In [50]:
```
# Plot transfer function for a given low pass circuit
fig = plt.figure(figsize=(9, 4))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
mags = 20 * torch.log10(evaluate_lp_circuit(ws, R_init))
R_low_des = 1 / (2 * math.pi * 800 * cap_value)
mags_des = 20 * torch.log10(evaluate_lp_circuit(ws, R_low_des))
tf, = plt.semilogx(ws / (2 * math.pi), mags, linewidth=3)
tf_des, = plt.semilogx(ws / (2 * math.pi), mags_des, linestyle="--", linewidth=3)
plt.xlim([1, 1e6])
plt.ylim([-60, 1])
plt.title("Low Pass Transfer Functions")
plt.xlabel("Frequency (Hz)")
plt.ylabel("dB")
plt.grid(which="both")
leg = plt.legend(["Predicted Transfer Function", "Desired Transfer Function"])
plt.tight_layout()

# Main update function for interactive plot
def update_tfs(R=R_init):
    mags = 20 * torch.log10(evaluate_lp_circuit(ws, R))
    tf.set_data(ws / (2 * math.pi), mags)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_tfs,
                 R=widgets.IntSlider(value=R_init, min=1, max=1000, step=1, description="R", layout=Layout(width='100%')))
ip
```

```
interactive(children=(IntSlider(value=500, description='R', layout=Layout(width='100%'), max=1000, min=1), Out…
```

Figure



according to observation, the predicted and desired transfer functions match when R is about 200

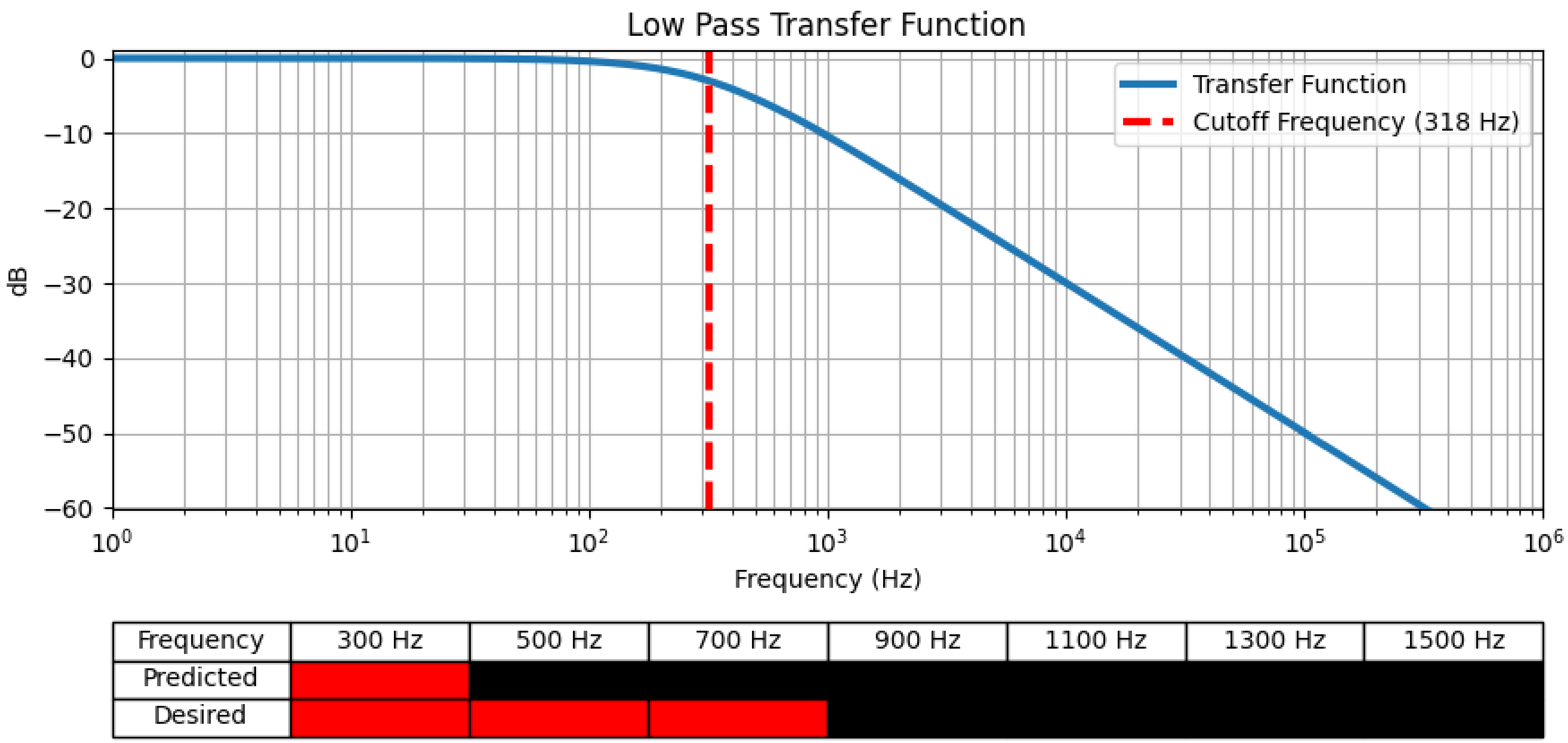
(b) Designing a Low pass Filter from Binary Data

In [51]:
```
# Plot transfer function for a given low pass circuit
fig = plt.figure(figsize=(9, 5))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
mags = 20 * torch.log10(evaluate_lp_circuit(ws, R_init))
cutoff = ws[np.argmax(mags < cutoff_dB)]
tf, = plt.semilogx(ws / (2 * math.pi), mags, linewidth=3)
cut = plt.axvline(cutoff / (2 * math.pi), c="red", linestyle="--", linewidth=3)
plt.xlim([1, 1e6])
plt.ylim([-60, 1])
plt.title("Low Pass Transfer Function")
plt.xlabel("Frequency (Hz)")
plt.ylabel("dB")
plt.grid(which="both")
leg = plt.legend(["Transfer Function", f"Cutoff Frequency ({1 / (2 * math.pi * R_init * cap_value):.0f} Hz)"])

# Plot table of LED on/off values (predicted and desired)
ws_test = 2 * math.pi * np.linspace(300, 1500, num=7)
table_txt = np.zeros((3, len(ws_test) + 1), dtype="U15")
table_txt[0, :] = ["Frequency"] + [f"{w / (2 * math.pi):.0f} Hz" for w in ws_test]
table_txt[1:, 0] = ["Predicted", "Desired"]
table_colors = np.zeros_like(table_txt, dtype=(np.int32, (3,)))
table_colors[-1, 1:4] = (1, 0, 0)
table_colors[1, 1] = (1, 0, 0)
table_colors[:, :1] = (1, 1, 1)
table_colors[:1, :] = (1, 1, 1)
tab = plt.table(table_txt, table_colors, bbox=[0.0, -0.5, 1.0, 0.25], cellLoc="center")
plt.tight_layout()

# Main update function for interactive plot
def update_lights(R=R_init):
    mags = 20 * torch.log10(evaluate_lp_circuit(ws, R))
    cutoff = ws[np.argmax(mags < cutoff_dB)]
    tf.set_data(ws / (2 * math.pi), mags)
    cut.set_xdata(cutoff / (2 * math.pi))
    for i, w in enumerate(ws_test):
        if w < cutoff:
            tab[(1, i+1)].set_facecolor((1, 0, 0))
        else:
            tab[(1, i+1)].set_facecolor((0, 0, 0))
    leg.get_texts()[1].set_text(f"Cutoff Frequency ({1 / (2 * math.pi * R * cap_value):.0f} Hz)")
    fig.canvas.draw_idle()
```

```python
# Include sliders for relevant quantities
ip = interactive(update_lights,
                R=widgets.IntSlider(value=R_init, min=1, max=1000, step=1, description="R", layout=Layout(width='100%')))
ip
```

```
interactive(children=(IntSlider(value=500, description='R', layout=Layout(width='100%'), max=1000, min=1), Out…
```

Figure



the corresponding resistor value is 200Ω and cutoff frequency is about 796Hz

(c) Learning a Low Pass Filter from Desired Transfer Function Samples

In [10]:
```
# PyTorch model of the low pass circuit (for training)
class LowPassCircuit(nn.Module):
    def __init__(self, R=None):
        super().__init__()
        self.R = nn.Parameter(torch.tensor(R, dtype=float) if R is not None else torch.rand(1) * 1000)

    # Note: the forward function is called automatically when the __call__ function of this object is called
    def forward(self, freqs):
        return evaluate_lp_circuit(freqs, self.R)

# Generate training data in a uniform log scale of frequences, then evaluate using the true transfer function
def generate_lp_training_data(n):
    rand_ws = 2 * math.pi * torch.pow(10, torch.rand(n) * 6)
    labels = evaluate_lp_circuit(rand_ws, R_low_des)
    return rand_ws, labels

# Train a given low pass filter
def train_lp_circuit_tf(circuit, loss_fn, dataset_size, max_training_steps, lr):

    R_values = [float(circuit.R.data)]
    grad_values = [np.nan]
    train_data = generate_lp_training_data(dataset_size)
    print(f"Initial Resistor Value: R = {float(circuit.R.data):.0f}")
    iter_bar = tqdm.trange(max_training_steps, desc="Training Iter")
    for i in iter_bar:
        pred = circuit(train_data[0])
        loss = loss_fn(pred, train_data[1]).mean()
        grad = torch.autograd.grad(loss, circuit.R)
        with torch.no_grad():
            circuit.R -= lr * grad[0]

        R_values.append(float(circuit.R.data))
        grad_values.append(float(grad[0].data))
        iter_bar.set_postfix_str(f"Loss: {float(loss.data):.3f}, R={float(circuit.R.data):.0f}")
        if loss.data < 1e-6 or abs(grad[0].data) < 1e-6:
            break

    print(f"Final Resistor Value: R = {float(circuit.R.data):.0f}")
    return train_data, R_values, grad_values
```

In [11]:
```python
# Create a circuit, use mean squared error loss w/ learning rate of 200
circuit = LowPassCircuit(1000)
loss_fn = lambda x, y: (x - y) ** 2
lr = 200
train_data_low_tf, R_values_low_tf, grad_values_low_tf = train_lp_circuit_tf(circuit, loss_fn, dataset_size, max_training_steps, lr)
```

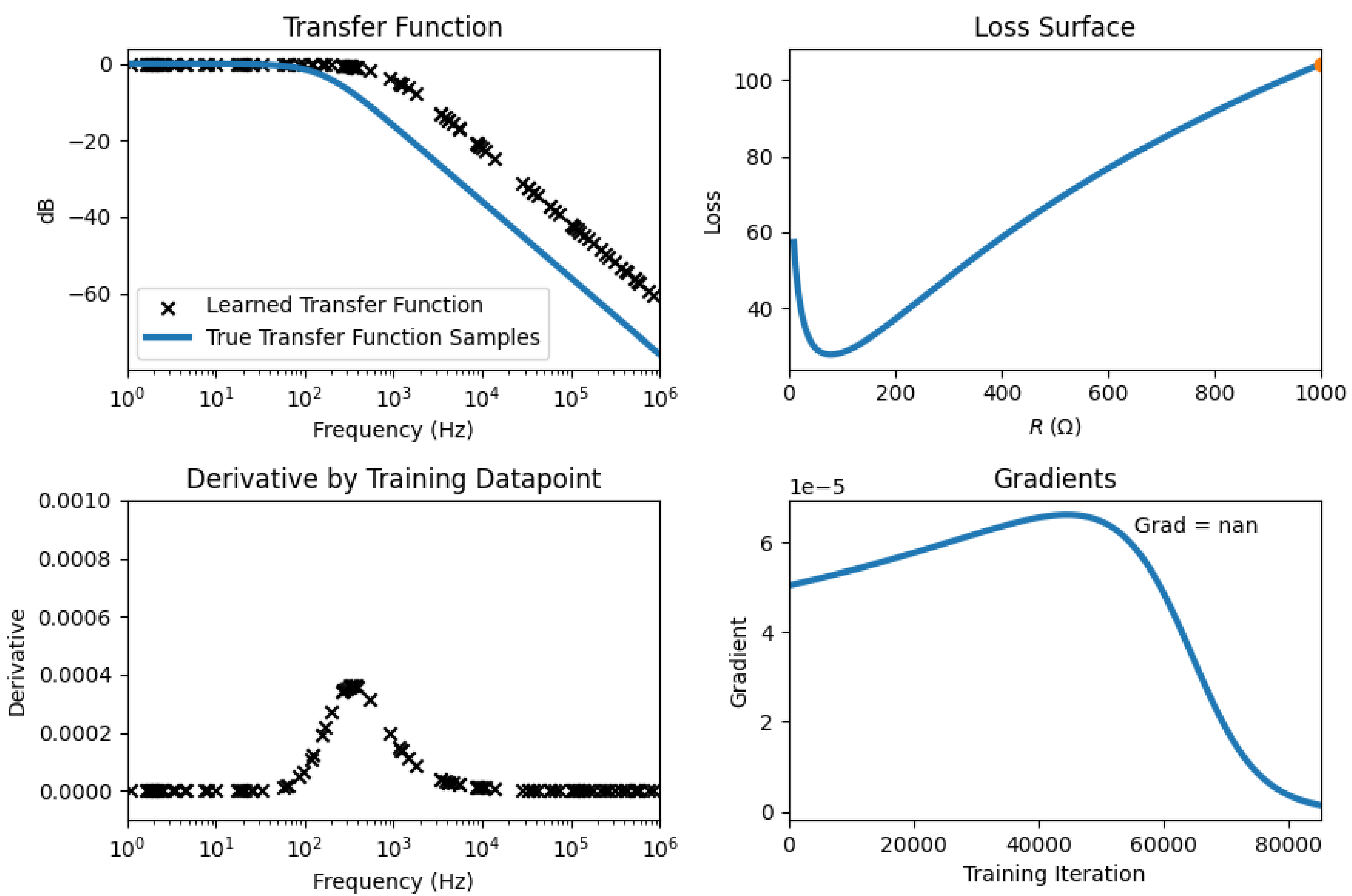
```
Initial Resistor Value: R = 1000

Training Iter:  85%|████████▌ | 85271/100000 [04:30<00:49, 298.52it/s, Loss: 0.000, R=200]
```

In [31]:
```
# Plot transfer function over training
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 100)
ax1.scatter(train_data_low_tf[0][::subsample] / (2 * math.pi), 20 * torch.log10(train_data_low_tf[1][::subsample]), c="k", marker="x")
learned_tf, = ax1.semilogx(ws / (2 * math.pi), 20 * torch.log10(evaluate_lp_circuit(ws, R_values_low_tf[0])), linewidth=3)
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("dB")
ax1.legend(["Learned Transfer Function", "True Transfer Function Samples"])

# Show loss surface over training
eval_pts = torch.arange(10, 1001, 1)
eval_vals = evaluate_lp_circuit(train_data_low_tf[0][:, None], eval_pts[None, :])
loss_surface_mse = loss_fn(eval_vals, train_data_low_tf[1][:, None].expand(eval_vals.shape))
ax2.plot(eval_pts, loss_surface_mse.sum(0), linewidth=3)
cur_loss, = ax2.plot(R_values_low_tf[0], loss_surface_mse[:, int(R_values_low_tf[0] - 10)].sum(0), marker="o")
cur_loss_label = ax2.annotate(f"R = {R_values_low_tf[0]:.0f}", (0, 0), xytext=(0.82, 0.9), textcoords='axes fraction')
ax2.set_title("Loss Surface")
ax2.set_xlim([0, 1000])
ax2.set_xlabel("$R \; (\Omega)$")
ax2.set_ylabel("Loss")

# Show loss contributions of each data point
cur_circuit = LowPassCircuit(R_values_low_tf[0])
data_losses = loss_fn(cur_circuit(train_data_low_tf[0][::subsample]), (train_data_low_tf[1][::subsample]).float())
data_grads = torch.zeros(len(data_losses))
for i, dl in enumerate(data_losses):
    data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
data_grads_scat = ax3.scatter(train_data_low_tf[0][::subsample] / (2 * math.pi), data_grads, marker="x", c="k")
ax3.set_xscale("log")
ax3.set_ylabel("Derivative")
ax3.set_xlim([1, 1e6])
ax3.set_ylim([-1e-4, 1e-3])
ax3.set_xlabel("Frequency (Hz)")
ax3.set_title("Derivative by Training Datapoint")

# Show total gradient at each training iteration
ax4.plot(np.arange(len(grad_values_low_tf)), grad_values_low_tf, linewidth=3)
cur_iter, = ax4.plot(0, grad_values_low_tf[0], marker="o")
```

```
cur_grad_label = ax4.annotate(f"Grad = {grad_values_low_tf[0]:.2e}", (0, 0), xytext=(0.65, 0.9), textcoords='axes fraction')
ax4.set_xlabel("Training Iteration")
ax4.set_ylabel("Gradient")
ax4.set_title("Gradients")
ax4.set_xlim([-1, len(grad_values_low_tf)])

plt.tight_layout()

# Main update function for interactive plots
def update_iter_tf(t=0):
    learned_tf.set_data(ws / (2 * math.pi), 20 * torch.log10(evaluate_lp_circuit(ws, R_values_low_tf[t])))
    cur_loss.set_data(R_values_low_tf[t], loss_surface_mse[:, int(R_values_low_tf[t] - 10)].sum(0))
    cur_loss_label.set_text(f"R = {R_values_low_tf[t]:.0f}")
    cur_iter.set_data(t, grad_values_low_tf[t])
    cur_grad_label.set_text(f"Grad = {grad_values_low_tf[t]:.2e}")
    cur_circuit = LowPassCircuit(R_values_low_tf[t])
    data_losses = loss_fn(cur_circuit(train_data_low_tf[0][::subsample]), (train_data_low_tf[1][::subsample]).float())
    data_grads = torch.zeros(len(data_losses))
    for i, dl in enumerate(data_losses):
        data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
    data_grads_scat.set_offsets(torch.stack((train_data_low_tf[0][::subsample] / (2 * math.pi), data_grads)).T)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_tf,
                 t=widgets.IntSlider(value=0, min=0, max=len(R_values_low_tf) - 1, step=1, description="Training Iteration", style={'
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=85…
```

Figure


Transfer Function
dB
0
−20
−40
−60
Learned Transfer Function
True Transfer Function Samples
Frequency (Hz)
Loss Surface
Loss
100
80
60
40
0
200
400
600
800
1000
R (Ω)
Derivative by Training Datapoint
Derivative
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
Frequency (Hz)
Gradients
1e−5
Grad = nan
Gradient
6
4
2
0
0
20000
40000
60000
80000
Training Iteration

(d) Learning a Low Pass Filter from Binary Data with Mean Squared Error Loss

In [32]:
```
# Train a given low pass filter from binary data
def train_lp_circuit_binary(circuit, loss_fn, dataset_size, max_training_steps, lr):

    R_values = [float(circuit.R.data)]
    grad_values = [np.nan]
    train_data = generate_lp_training_data(dataset_size)
    print(f"Initial Resistor Value: R = {float(circuit.R.data):.0f}")
    iter_bar = tqdm.trange(max_training_steps, desc="Training Iter")
    for i in iter_bar:
        pred = circuit(train_data[0])

        ### YOUR CODE HERE
        label_binary = (train_data[1] > cutoff_mag).float()

        loss = loss_fn(pred, label_binary).float().mean()
        ### END YOUR CODE
        grad = torch.autograd.grad(loss, circuit.R)
        with torch.no_grad():
            circuit.R -= lr * grad[0]

        R_values.append(float(circuit.R.data))
        grad_values.append(float(grad[0].data))
        iter_bar.set_postfix_str(f"Loss: {float(loss.data):.3f}, R={float(circuit.R.data):.0f}")
        if loss.data < 1e-6 or abs(grad[0].data) < 1e-6:
            break

    print(f"Final Resistor Value: R = {float(circuit.R.data):.0f}")
    return train_data, R_values, grad_values
```

In [33]:
```
# Create a circuit, use MSE loss with learning rate of 200
circuit = LowPassCircuit(500)
loss_fn = lambda x, y: (x - y) ** 2
lr = 200
train_data_low_bin, R_values_low_bin, grad_values_low_bin = train_lp_circuit_binary(circuit, loss_fn, dataset_size, max_training_steps
```

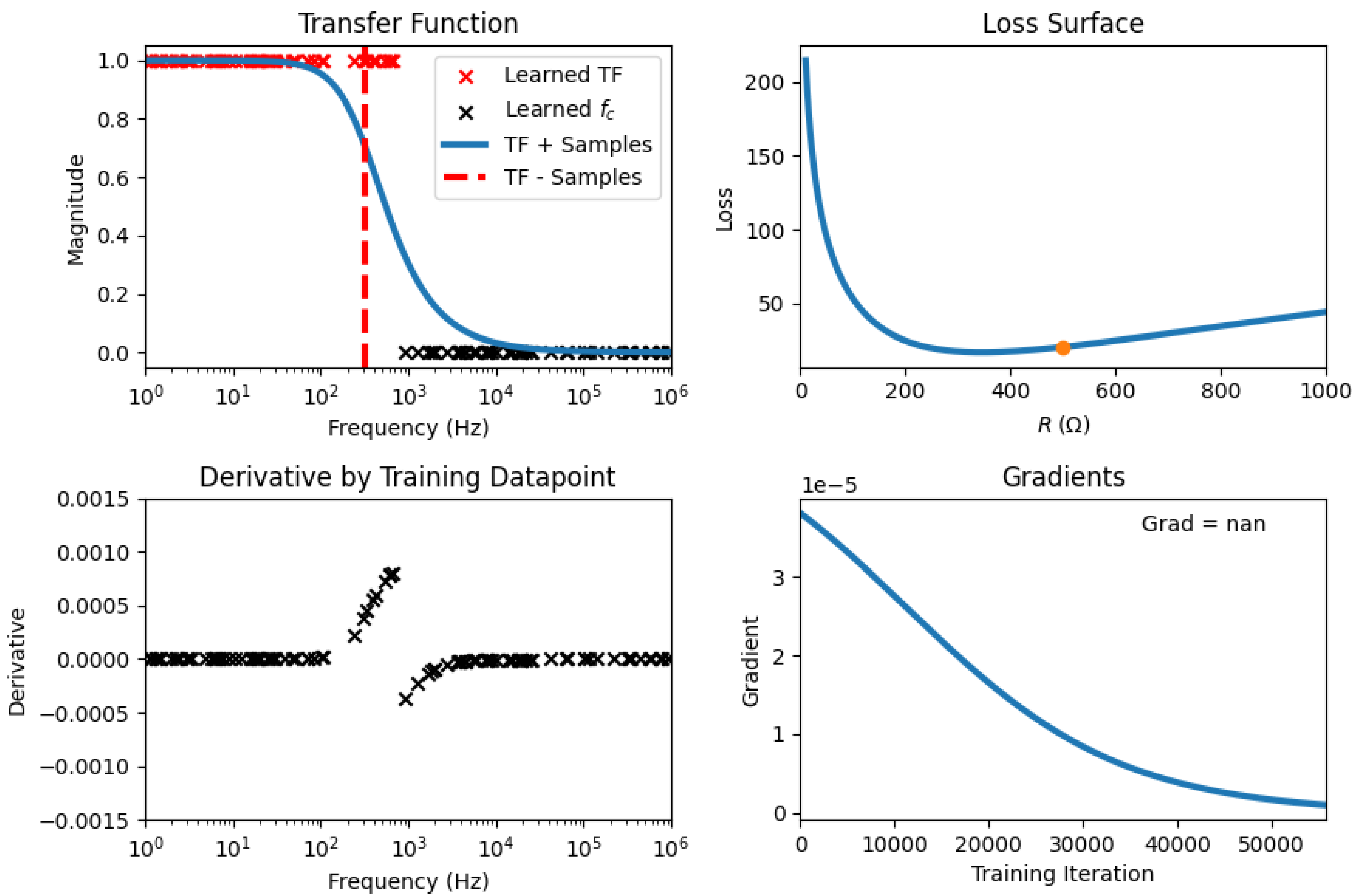
```
Initial Resistor Value: R = 500

Training Iter:  56%|█████        | 55906/100000 [02:44<02:10, 339.08it/s, Loss: 0.017, R=347]

Final Resistor Value: R = 347
```

In [34]:
```
# Plot transfer function over training
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 100)
train_data_mask = train_data_low_bin[1][::subsample] > cutoff_mag
ax1.scatter(train_data_low_bin[0][::subsample][train_data_mask] / (2 * math.pi), np.ones(train_data_mask.sum()), c="r", marker="x")
ax1.scatter(train_data_low_bin[0][::subsample][~train_data_mask] / (2 * math.pi), np.zeros((~train_data_mask).sum()), c="k", marker="
mags = evaluate_lp_circuit(ws, R_values_low_bin[0])
learned_tf, = ax1.semilogx(ws / (2 * math.pi), mags, linewidth=3)
cutoff = ws[np.argmax(mags < cutoff_mag)]
cut = ax1.axvline(cutoff / (2 * math.pi), c="red", linestyle="--", linewidth=3)
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("Magnitude")
ax1.legend(["Learned TF", "Learned $f_c$", "TF + Samples", "TF - Samples"])

# Show loss surface over training
eval_pts = torch.arange(10, 1001, 1)
eval_vals = evaluate_lp_circuit(train_data_low_bin[0][:, None], eval_pts[None, :])
loss_surface_mse = loss_fn(eval_vals, (train_data_low_bin[1][:, None].expand(eval_vals.shape) > cutoff_mag).float())
ax2.plot(eval_pts, loss_surface_mse.sum(0), linewidth=3)
cur_loss, = ax2.plot(R_values_low_bin[0], loss_surface_mse[:, int(R_values_low_bin[0] - 10)].sum(0), marker="o")
cur_loss_label = ax2.annotate(f"R = {R_values_low_bin[0]:.0f}", (0, 0), xytext=(0.82, 0.9), textcoords='axes fraction')
ax2.set_title("Loss Surface")
ax2.set_xlim([0, 1000])
ax2.set_xlabel("$R \; (\Omega)$")
ax2.set_ylabel("Loss")

# Show loss contributions of each data point
cur_circuit = LowPassCircuit(R_values_low_bin[0])
data_losses = loss_fn(cur_circuit(train_data_low_bin[0][::subsample]), (train_data_low_bin[1][::subsample] > cutoff_mag).float())
data_grads = torch.zeros(len(data_losses))
for i, dl in enumerate(data_losses):
    data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
data_grads_scat = ax3.scatter(train_data_low_bin[0][::subsample] / (2 * math.pi), data_grads, marker="x", c="k")
ax3.set_xscale("log")
ax3.set_ylabel("Derivative")
ax3.set_xlim([1, 1e6])
ax3.set_ylim([-1.5e-3, 1.5e-3])
ax3.set_xlabel("Frequency (Hz)")
```

```
ax3.set_title("Derivative by Training Datapoint")

# Show gradient at each training iteration
ax4.plot(np.arange(len(grad_values_low_bin)), grad_values_low_bin, linewidth=3)
cur_iter, = ax4.plot(0, grad_values_low_bin[0], marker="o")
cur_grad_label = ax4.annotate(f"Grad = {grad_values_low_bin[0]:.2e}", (0, 0), xytext=(0.65, 0.9), textcoords='axes fraction')
ax4.set_xlabel("Training Iteration")
ax4.set_ylabel("Gradient")
ax4.set_title("Gradients")
ax4.set_xlim([-1, len(grad_values_low_bin)])

plt.tight_layout()

# Main update function for interactive plots
def update_iter_low_bin(t=0):
    mags = evaluate_lp_circuit(ws, R_values_low_bin[t])
    learned_tf.set_data(ws / (2 * math.pi), mags)
    cutoff = ws[np.argmax(mags < cutoff_mag)]
    cut.set_xdata(cutoff / (2 * math.pi))
    cur_loss.set_data(R_values_low_bin[t], loss_surface_mse[:, int(R_values_low_bin[t] - 10)].sum(0))
    cur_loss_label.set_text(f"R = {R_values_low_bin[t]:.0f}")
    cur_iter.set_data(t, grad_values_low_bin[t])
    cur_grad_label.set_text(f"Grad = {grad_values_low_bin[t]:.2e}")
    cur_circuit = LowPassCircuit(R_values_low_bin[t])
    data_losses = loss_fn(cur_circuit(train_data_low_bin[0][::subsample]), (train_data_low_bin[1][::subsample] > cutoff_mag).float())
    data_grads = torch.zeros(len(data_losses))
    for i, dl in enumerate(data_losses):
        data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
    data_grads_scat.set_offsets(torch.stack((train_data_low_bin[0][::subsample] / (2 * math.pi), data_grads)).T)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_low_bin,
                 t=widgets.IntSlider(value=0, min=0, max=len(R_values_low_bin) - 1, step=1, description="Training Iteration", style={
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=55…
```

Figure


Transfer Function
Learned TF
Learned f_c
TF + Samples
TF - Samples
Magnitude
Frequency (Hz)
Loss Surface
Loss
R (Ω)
Derivative by Training Datapoint
Derivative
Frequency (Hz)
Gradients
1e−5
Grad = nan
Gradient
Training Iteration

(e) Learning a Low Pass Filter from Binary Data with a Different Loss

In [35]:
```
circuit = LowPassCircuit(500)
### YOUR CODE HERE

#in order to avoid the situation that the positive and negative samples
#in our training data are pulling the resistor value in opposite directions
#we should calculate loss in three different situations
#1: y=0 and x>cutoff_mag, then loss should be (x-cutoff_mag)
#2: y=1 and x<cutoff_mag, then loss should be (cutoff_mag-x)
#3: (y=0 and x<cutoff_mag) or (y=1 and x>cutoff_mag), loss should be 0

#include all three situations,
#loss_fn should be y * torch.where(x<cutoff_mag, cutoff_mag-x, 0)
#          + (1-y) * torch.where(x>cutoff_mag, x-cutoff_mag, 0)

loss_fn = lambda x, y: (y * torch.where(x<cutoff_mag, cutoff_mag-x, 0) + \
              (1-y) * torch.where(x>cutoff_mag, x-cutoff_mag, 0))
### END YOUR CODE
train_data_low_bin, R_values_low_bin, grad_values_low_bin = train_lp_circuit_binary(circuit, loss_fn, dataset_size, max_training_steps
```

```
Initial Resistor Value: R = 500

Training Iter:  53%|█████▎    | 52841/100000 [02:52<02:33, 306.26it/s, Loss: 0.000, R=205]

Final Resistor Value: R = 205
```

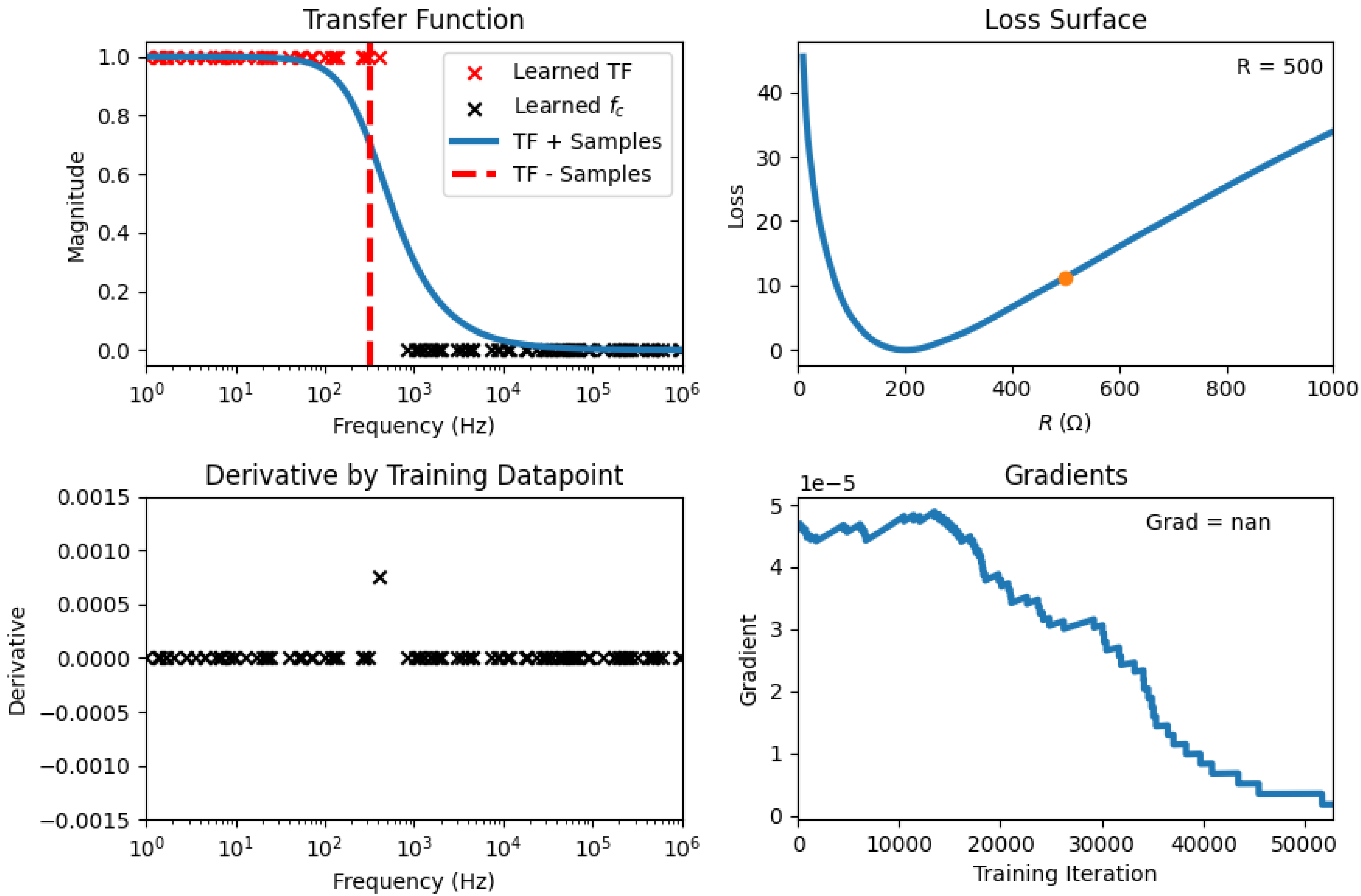
In [36]:

```
# Plot transfer function over training
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 100)
train_data_mask = train_data_low_bin[1][::subsample] > cutoff_mag
ax1.scatter(train_data_low_bin[0][::subsample][train_data_mask] / (2 * math.pi), np.ones(train_data_mask.sum()), c="r", marker="x")
ax1.scatter(train_data_low_bin[0][::subsample][~train_data_mask] / (2 * math.pi), np.zeros((~train_data_mask).sum()), c="k", marker="
mags = evaluate_lp_circuit(ws, R_values_low_bin[0])
learned_tf, = ax1.semilogx(ws / (2 * math.pi), mags, linewidth=3)
cutoff = ws[np.argmax(mags < cutoff_mag)]
cut = ax1.axvline(cutoff / (2 * math.pi), c="red", linestyle="--", linewidth=3)
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("Magnitude")
ax1.legend(["Learned TF", "Learned $f_c$", "TF + Samples", "TF - Samples"])

# Show loss surface over training
eval_pts = torch.arange(10, 1001, 1)
eval_vals = evaluate_lp_circuit(train_data_low_bin[0][:, None], eval_pts[None, :])
loss_surface_mse = loss_fn(eval_vals, (train_data_low_bin[1][:, None].expand(eval_vals.shape) > cutoff_mag).float())
ax2.plot(eval_pts, loss_surface_mse.sum(0), linewidth=3)
cur_loss, = ax2.plot(R_values_low_bin[0], loss_surface_mse[:, int(R_values_low_bin[0] - 10)].sum(0), marker="o")
cur_loss_label = ax2.annotate(f"R = {R_values_low_bin[0]:.0f}", (0, 0), xytext=(0.82, 0.9), textcoords='axes fraction')
ax2.set_title("Loss Surface")
ax2.set_xlim([0, 1000])
ax2.set_xlabel("$R \; (\Omega)$")
ax2.set_ylabel("Loss")

# Show loss contributions of each data point
cur_circuit = LowPassCircuit(R_values_low_bin[0])
data_losses = loss_fn(cur_circuit(train_data_low_bin[0][::subsample]), (train_data_low_bin[1][::subsample] > cutoff_mag).float())
data_grads = torch.zeros(len(data_losses))
for i, dl in enumerate(data_losses):
    data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
data_grads_scat = ax3.scatter(train_data_low_bin[0][::subsample] / (2 * math.pi), data_grads, marker="x", c="k")
ax3.set_xscale("log")
ax3.set_ylabel("Derivative")
ax3.set_xlim([1, 1e6])
ax3.set_ylim([-1.5e-3, 1.5e-3])
ax3.set_xlabel("Frequency (Hz)")
```

```
ax3.set_title("Derivative by Training Datapoint")

# Show gradient at each training iteration
ax4.plot(np.arange(len(grad_values_low_bin)), grad_values_low_bin, linewidth=3)
cur_iter, = ax4.plot(0, grad_values_low_bin[0], marker="o")
cur_grad_label = ax4.annotate(f"Grad = {grad_values_low_bin[0]:.2e}", (0, 0), xytext=(0.65, 0.9), textcoords='axes fraction')
ax4.set_xlabel("Training Iteration")
ax4.set_ylabel("Gradient")
ax4.set_title("Gradients")
ax4.set_xlim([-1, len(grad_values_low_bin)])

plt.tight_layout()

# Main update function for interactive plots
def update_iter_low_bin(t=0):
    mags = evaluate_lp_circuit(ws, R_values_low_bin[t])
    learned_tf.set_data(ws / (2 * math.pi), mags)
    cutoff = ws[np.argmax(mags < cutoff_mag)]
    cut.set_xdata(cutoff / (2 * math.pi))
    cur_loss.set_data(R_values_low_bin[t], loss_surface_mse[:, int(R_values_low_bin[t] - 10)].sum(0))
    cur_loss_label.set_text(f"R = {R_values_low_bin[t]:.0f}")
    cur_iter.set_data(t, grad_values_low_bin[t])
    cur_grad_label.set_text(f"Grad = {grad_values_low_bin[t]:.2e}")
    cur_circuit = LowPassCircuit(R_values_low_bin[t])
    data_losses = loss_fn(cur_circuit(train_data_low_bin[0][::subsample]), (train_data_low_bin[1][::subsample] > cutoff_mag).float())
    data_grads = torch.zeros(len(data_losses))
    for i, dl in enumerate(data_losses):
        data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
    data_grads_scat.set_offsets(torch.stack((train_data_low_bin[0][::subsample] / (2 * math.pi), data_grads)).T)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_low_bin,
                 t=widgets.IntSlider(value=0, min=0, max=len(R_values_low_bin) - 1, step=1, description="Training Iteration", style={
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=52…
```

Figure


Transfer Function
Learned TF
Learned f_c
TF + Samples
TF - Samples
Magnitude
Frequency (Hz)
Loss Surface
R = 500
Loss
R (Ω)
Derivative by Training Datapoint
Derivative
Frequency (Hz)
Gradients
1e−5
Grad = nan
Gradient
Training Iteration

(f) Learning a High Pass Filter from Binary Data

In [37]:
```
# Transfer function: evaluates magnitude of given frequencies for a resistor value in the high pass circuit
def evaluate_hp_circuit(freqs, R_high):
    ### YOUR CODE HERE
    RCF = R_high * cap_value * freqs
    return torch.sqrt(RCF ** 2) / torch.sqrt(1 + RCF ** 2)
    ### END YOUR CODE

# PyTorch model of the high pass circuit (for training)
class HighPassCircuit(nn.Module):
    def __init__(self, R=None):
        super().__init__()
        self.R = nn.Parameter(torch.tensor(R, dtype=float) if R is not None else torch.rand(1) * 1000)

    def forward(self, freqs):
        return evaluate_hp_circuit(freqs, self.R)

# Generate training data in a uniform log scale of frequences, then evaluate using the true transfer function
R_high_des = 1 / (2 * math.pi * 5000 * cap_value)
def generate_hp_training_data(n):
    rand_ws = 2 * math.pi * torch.pow(10, torch.rand(n) * 6)
    labels = evaluate_hp_circuit(rand_ws, R_high_des)
    return rand_ws, labels

# Train a given low pass filter from binary data
def train_hp_circuit_binary(circuit, loss_fn, dataset_size, max_training_steps, lr):

    R_values = [float(circuit.R.data)]
    grad_values = [np.nan]
    train_data = generate_hp_training_data(dataset_size)
    print(f"Initial Resistor Value: R = {float(circuit.R.data):.0f}")
    iter_bar = tqdm.trange(max_training_steps, desc="Training Iter")
    for i in iter_bar:
        pred = circuit(train_data[0])
        loss = loss_fn(pred, (train_data[1] > cutoff_mag).float()).mean()
        ### YOUR CODE HERE
        grad = torch.autograd.grad(loss, circuit.R)
        ### END YOUR CODE
        with torch.no_grad():
            ### YOUR CODE HERE
            circuit.R -= lr * grad[0]
            ### END YOUR CODE
```

```
        R_values.append(float(circuit.R.data))
        grad_values.append(float(grad[0].data))
        iter_bar.set_postfix_str(f"Loss: {float(loss.data):.3f}, R={float(circuit.R.data):.0f}")
        if loss.data < 1e-6 or abs(grad[0].data) < 1e-6:
            break

    print(f"Final Resistor Value: R = {float(circuit.R.data):.0f}")
    return train_data, R_values, grad_values
```

In [38]:

```
# Create a circuit, use loss_fn with learning rate of 1000
circuit = HighPassCircuit(500)
### YOUR CODE HERE
loss_fn = lambda x, y: (y * torch.where(x<cutoff_mag, cutoff_mag-x, 0) + \
              (1-y) * torch.where(x>cutoff_mag, x-cutoff_mag, 0))
### END YOUR CODE
lr = 1000
train_data_high_bin, R_values_high_bin, grad_values_high_bin = train_hp_circuit_binary(circuit, loss_fn, dataset_size, max_training_st
```

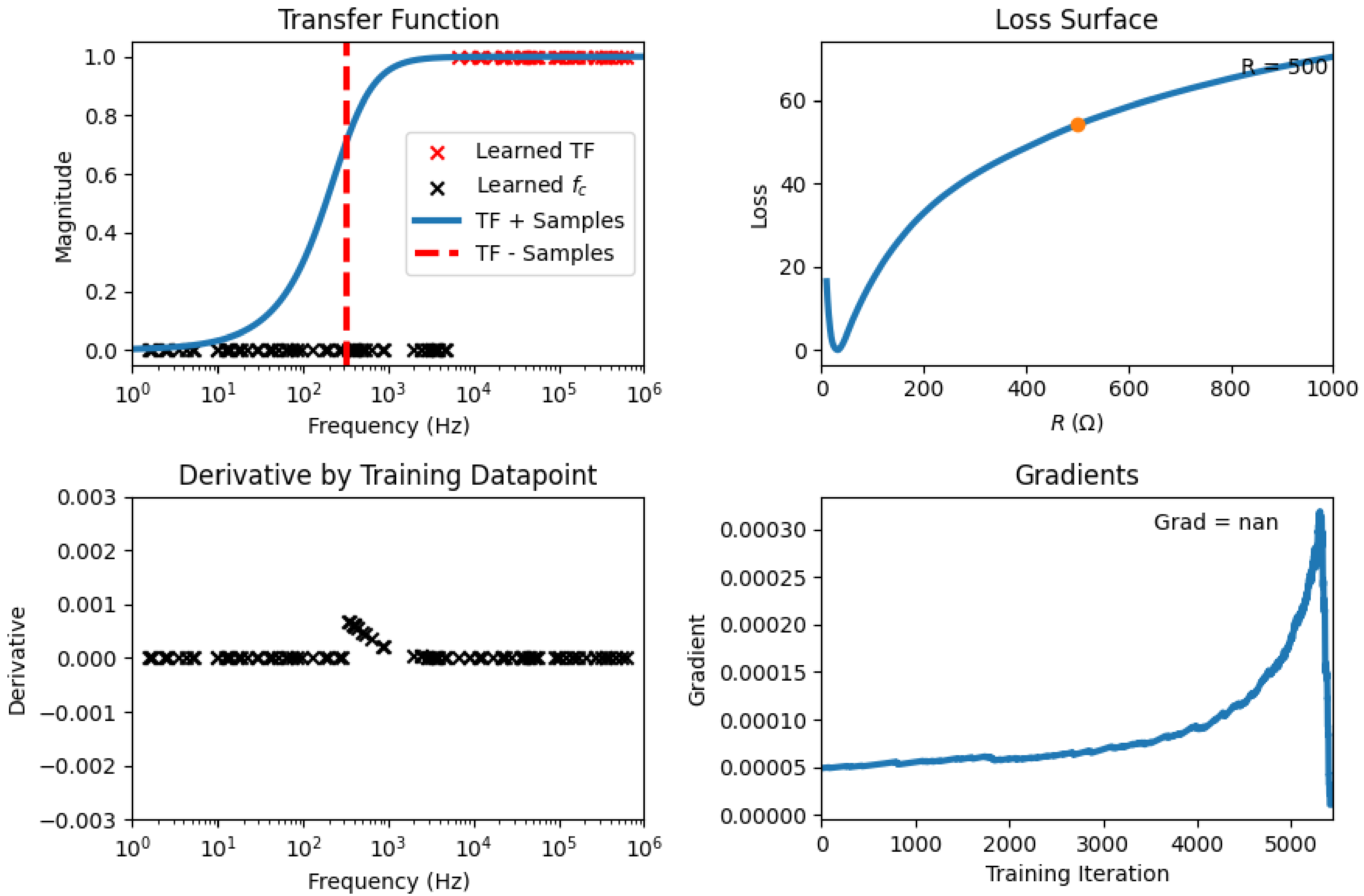
```
Initial Resistor Value: R = 500

Training Iter:   5%|█          | 5449/100000 [00:21<06:07, 257.47it/s, Loss: 0.000, R=32]

Final Resistor Value: R = 32
```

In [39]:
```
# Plot transfer function over training
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 100)
train_data_mask = train_data_high_bin[1][::subsample] > cutoff_mag
ax1.scatter(train_data_high_bin[0][::subsample][train_data_mask] / (2 * math.pi), np.ones(train_data_mask.sum()), c="r", marker="x")
ax1.scatter(train_data_high_bin[0][::subsample][~train_data_mask] / (2 * math.pi), np.zeros((~train_data_mask).sum()), c="k", marker=
mags = evaluate_hp_circuit(ws, R_values_high_bin[0])
learned_tf, = ax1.semilogx(ws / (2 * math.pi), mags, linewidth=3)
cutoff = ws[np.argmax(mags > cutoff_mag)]
cut = ax1.axvline(cutoff / (2 * math.pi), c="red", linestyle="--", linewidth=3)
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("Magnitude")
ax1.legend(["Learned TF", "Learned $f_c$", "TF + Samples", "TF - Samples"])

# Show loss surface over training
eval_pts = torch.arange(10, 1001, 1)
eval_vals = evaluate_hp_circuit(train_data_high_bin[0][:, None], eval_pts[None, :])
loss_surface_mse = loss_fn(eval_vals, (train_data_high_bin[1][:, None].expand(eval_vals.shape) > cutoff_mag).float())
ax2.plot(eval_pts, loss_surface_mse.sum(0), linewidth=3)
cur_loss, = ax2.plot(R_values_high_bin[0], loss_surface_mse[:, int(R_values_high_bin[0] - 10)].sum(0), marker="o")
cur_loss_label = ax2.annotate(f"R = {R_values_high_bin[0]:.0f}", (0, 0), xytext=(0.82, 0.9), textcoords='axes fraction')
ax2.set_title("Loss Surface")
ax2.set_xlim([0, 1000])
ax2.set_xlabel("$R \; (\Omega)$")
ax2.set_ylabel("Loss")

# Show loss contributions of each data point
cur_circuit = HighPassCircuit(R_values_high_bin[0])
data_losses = loss_fn(cur_circuit(train_data_high_bin[0][::subsample]), (train_data_high_bin[1][::subsample] > cutoff_mag).float())
data_grads = torch.zeros(len(data_losses))
for i, dl in enumerate(data_losses):
    data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
data_grads_scat = ax3.scatter(train_data_high_bin[0][::subsample] / (2 * math.pi), data_grads, marker="x", c="k")
ax3.set_xscale("log")
ax3.set_ylabel("Derivative")
ax3.set_xlim([1, 1e6])
ax3.set_ylim([-3e-3, 3e-3])
ax3.set_xlabel("Frequency (Hz)")
```

```
ax3.set_title("Derivative by Training Datapoint")

# Show gradient at each training iteration
ax4.plot(np.arange(len(grad_values_high_bin)), grad_values_high_bin, linewidth=3)
cur_iter, = ax4.plot(0, grad_values_high_bin[0], marker="o")
cur_grad_label = ax4.annotate(f"Grad = {grad_values_high_bin[0]:.2e}", (0, 0), xytext=(0.65, 0.9), textcoords='axes fraction')
ax4.set_xlabel("Training Iteration")
ax4.set_ylabel("Gradient")
ax4.set_title("Gradients")
ax4.set_xlim([-1, len(grad_values_high_bin)])

plt.tight_layout()

# Main update function for interactive plots
def update_iter_high_bin(t=0):
    mags = evaluate_hp_circuit(ws, R_values_high_bin[t])
    learned_tf.set_data(ws / (2 * math.pi), mags)
    cutoff = ws[np.argmax(mags > cutoff_mag)]
    cut.set_xdata(cutoff / (2 * math.pi))
    cur_loss.set_data(R_values_high_bin[t], loss_surface_mse[:, int(R_values_high_bin[t] - 10)].sum(0))
    cur_loss_label.set_text(f"R = {R_values_high_bin[t]:.0f}")
    cur_iter.set_data(t, grad_values_high_bin[t])
    cur_grad_label.set_text(f"Grad = {grad_values_high_bin[t]:.2e}")
    cur_circuit = HighPassCircuit(R_values_high_bin[t])
    data_losses = loss_fn(cur_circuit(train_data_high_bin[0][::subsample]), (train_data_high_bin[1][::subsample] > cutoff_mag).float()
    data_grads = torch.zeros(len(data_losses))
    for i, dl in enumerate(data_losses):
        data_grads[i] = torch.autograd.grad(dl, cur_circuit.R, retain_graph=True)[0]
    data_grads_scat.set_offsets(torch.stack((train_data_high_bin[0][::subsample] / (2 * math.pi), data_grads)).T)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_high_bin,
                 t=widgets.IntSlider(value=0, min=0, max=len(R_values_high_bin) - 1, step=1, description="Training Iteration", style=
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=54…
```

Figure


Transfer Function
Magnitude
Frequency (Hz)
Learned TF
Learned f_c
TF + Samples
TF - Samples
Loss Surface
Loss
R = 500
R (Ω)
Derivative by Training Datapoint
Derivative
Frequency (Hz)
Gradients
Gradient
Grad = nan
Training Iteration

(g) Learning a Band Pass Filter from Binary Data

In [40]:
```
# Transfer function: evaluates magnitude of given frequencies for resistor values in the band pass circuit
def evaluate_bp_circuit(freqs, R_low, R_high):
    ### YOUR CODE HERE
    # bp_circuit is a concatenation of hp and lp
    return evaluate_hp_circuit(freqs, R_high) * evaluate_lp_circuit(freqs, R_low)
    ### END YOUR CODE

# PyTorch model of the band pass circuit (for training)
class BandPassCircuit(nn.Module):
    def __init__(self, R_low=None, R_high=None):
        super().__init__()
        self.R_low = nn.Parameter(torch.tensor(R_low, dtype=float) if R_low is not None else torch.rand(1) * 1000)
        self.R_high = nn.Parameter(torch.tensor(R_high, dtype=float) if R_high is not None else torch.rand(1) * 1000)

    def forward(self, freqs):
        return evaluate_bp_circuit(freqs, self.R_low, self.R_high)

# Generate training data in a uniform log scale of frequences, then evaluate using true transfer function
R_low_des = 1 / (2 * math.pi * 4000 * cap_value)
R_high_des = 1 / (2 * math.pi * 1000 * cap_value)
def generate_bp_training_data(n):
    rand_ws = 2 * math.pi * torch.pow(10, torch.rand(n) * 6)
    labels = evaluate_bp_circuit(rand_ws, R_low_des, R_high_des)
    return rand_ws, labels

# Train a given low pass filter from binary data
def train_bp_circuit_binary(circuit, loss_fn, dataset_size, max_training_steps, lr):

    R_values = [[float(circuit.R_low.data), float(circuit.R_high.data)]]
    grad_values = [[np.nan, np.nan]]
    train_data = generate_bp_training_data(dataset_size)
    print(f"Initial Resistor Values: R_low = {float(circuit.R_low.data):.0f}, R_high = {float(circuit.R_high.data):.0f}")
    iter_bar = tqdm.trange(max_training_steps, desc="Training Iter")
    for i in iter_bar:
        pred = circuit(train_data[0])
        loss = loss_fn(pred, (train_data[1] > cutoff_mag).float()).mean()
        ### YOUR CODE HERE
        grad = torch.autograd.grad(loss, [circuit.R_low,circuit.R_high] )
        ### END YOUR CODE
        with torch.no_grad():
            ### YOUR CODE HERE
```

```
            circuit.R_low -= lr * grad[0]
            circuit.R_high -= lr * grad[1]
            ### END YOUR CODE

        R_values.append([float(circuit.R_low.data), float(circuit.R_high.data)])
        grad_values.append([float(grad[0].data), float(grad[1].data)])
        iter_bar.set_postfix_str(f"Loss: {float(loss.data):.3f}, R_low={float(circuit.R_low.data):.0f}, R_high={float(circuit.R_high.d
        if loss.data < 1e-6 or (abs(grad[0].data) < 1e-6 and abs(grad[1].data) < 1e-6):
            break

    print(f"Final Resistor Values: R_low = {float(circuit.R_low.data):.0f}, R_high = {float(circuit.R_high.data):.0f}")
    return train_data, R_values, grad_values
```

In [41]:

```
# Create a circuit, use loss_fn with learning rate of 1000
circuit = BandPassCircuit(500, 500)
lr = 1000
train_data_band_bin, R_values_band_bin, grad_values_band_bin = train_bp_circuit_binary(circuit, loss_fn, dataset_size, max_training_st
```

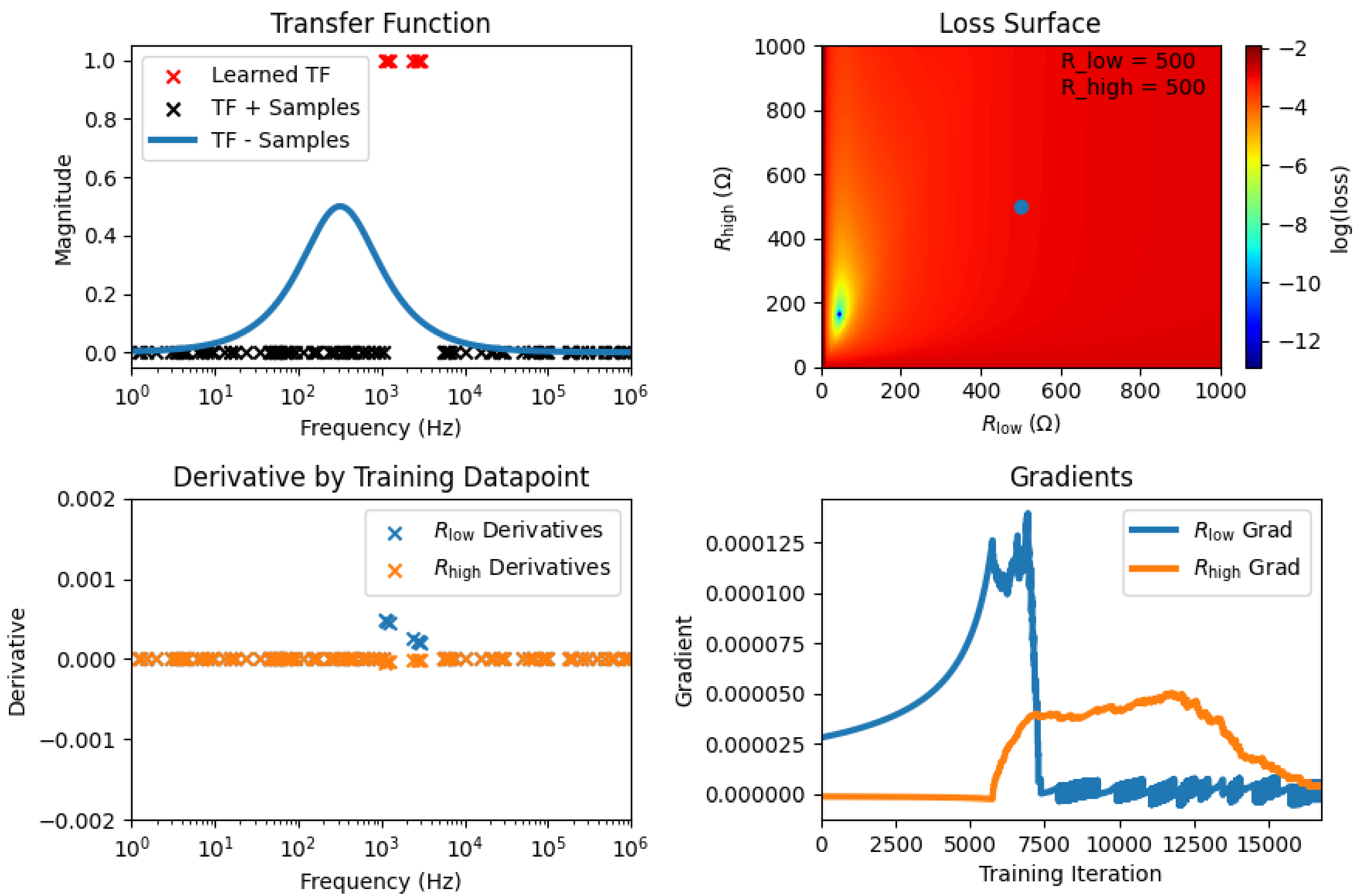
```
Initial Resistor Values: R_low = 500, R_high = 500

Training Iter:  17%|██        | 16731/100000 [01:05<05:28, 253.52it/s, Loss: 0.000, R_low=40, R_high=160]

Final Resistor Values: R_low = 40, R_high = 160
```

In [42]:
```
# Plot transfer function over training
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 100)
train_data_mask = train_data_band_bin[1][::subsample] > cutoff_mag
ax1.scatter(train_data_band_bin[0][::subsample][train_data_mask] / (2 * math.pi), np.ones(train_data_mask.sum()), c="r", marker="x")
ax1.scatter(train_data_band_bin[0][::subsample][~train_data_mask] / (2 * math.pi), np.zeros((~train_data_mask).sum()), c="k", marker=
learned_tf, = ax1.semilogx(ws / (2 * math.pi), evaluate_bp_circuit(ws, *R_values_band_bin[0]), linewidth=3)
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("Magnitude")
ax1.legend(["Learned TF", "TF + Samples", "TF - Samples"])

# Show loss surfaces for BCE and MSE Loss
eval_pts = torch.stack(torch.meshgrid((torch.arange(0, 1000, 10), torch.arange(0, 1000, 10)), indexing="ij"))
eval_vals = evaluate_bp_circuit(train_data_band_bin[0][:, None, None], eval_pts[0][None, ...], eval_pts[1][None, ...])
loss_surface = loss_fn(eval_vals, (train_data_band_bin[1][..., None, None].expand(eval_vals.shape) > cutoff_mag).float())
loss_surf = ax2.imshow(torch.log(loss_surface.mean(0)).T, cmap=plt.cm.jet, extent=(0, 1000, 0, 1000), aspect="auto", origin="lower")
cur_loss, = ax2.plot(*R_values_band_bin[0], marker="o")
cur_loss_label = ax2.annotate(f"R_low = {R_values_band_bin[0][0]:.0f}\nR_high = {R_values_band_bin[0][1]:.0f}", (0, 0), xytext=(0.6, 0
ax2.set_title("Loss Surface")
ax2.set_xlabel("$R_\mathrm{low} \; (\Omega)$")
ax2.set_ylabel("$R_\mathrm{high} \; (\Omega)$")
fig.colorbar(loss_surf, ax=ax2, label="log(loss)")

# Show loss contributions of each data point
cur_circuit = BandPassCircuit(*R_values_band_bin[0])
data_losses = loss_fn(cur_circuit(train_data_band_bin[0][::subsample]), (train_data_band_bin[1][::subsample] > cutoff_mag).float())
data_grads = torch.zeros((len(data_losses), 2))
for i, dl in enumerate(data_losses):
    data_grads[i] = torch.tensor(torch.autograd.grad(dl, (cur_circuit.R_low, cur_circuit.R_high), retain_graph=True))
data_grads_scat1 = ax3.scatter(train_data_band_bin[0][::subsample] / (2 * math.pi), data_grads[:, 0], marker="x")
data_grads_scat2 = ax3.scatter(train_data_band_bin[0][::subsample] / (2 * math.pi), data_grads[:, 1], marker="x")
ax3.set_xscale("log")
ax3.set_ylabel("Derivative")
ax3.set_xlim([1, 1e6])
ax3.set_ylim([-2e-3, 2e-3])
ax3.set_xlabel("Frequency (Hz)")
ax3.set_title("Derivative by Training Datapoint")
ax3.legend(["$R_\mathrm{low}$ Derivatives", "$R_\mathrm{high}$ Derivatives"])
```

```
# Show gradient at each training iteration
ax4.plot(np.arange(len(grad_values_band_bin)), grad_values_band_bin, linewidth=3)
cur_grad0, = ax4.plot(0, grad_values_band_bin[0][0], marker="o")
cur_grad1, = ax4.plot(0, grad_values_band_bin[0][1], marker="o")
ax4.set_xlabel("Training Iteration")
ax4.set_ylabel("Gradient")
ax4.set_title("Gradients")
ax4.set_xlim([-1, len(grad_values_band_bin)])
ax4.legend(["$R_\mathrm{low}$ Grad", "$R_\mathrm{high}$ Grad"])

plt.tight_layout()

# Main update function for interactive plots
def update_iter_band_bin(t=0):
    mags = evaluate_bp_circuit(ws, *R_values_band_bin[t])
    learned_tf.set_data(ws / (2 * math.pi), mags)
    cur_loss.set_data(*R_values_band_bin[t])
    cur_loss_label.set_text(f"R_low = {R_values_band_bin[t][0]:.0f}\nR_high = {R_values_band_bin[t][1]:.0f}")
    cur_grad0.set_data(t, grad_values_band_bin[t][0])
    cur_grad1.set_data(t, grad_values_band_bin[t][1])
    cur_circuit = BandPassCircuit(*R_values_band_bin[t])
    data_losses = loss_fn(cur_circuit(train_data_band_bin[0][::subsample]), (train_data_band_bin[1][::subsample] > cutoff_mag).float()
    data_grads = torch.zeros((len(data_losses), 2))
    for i, dl in enumerate(data_losses):
        data_grads[i] = torch.tensor(torch.autograd.grad(dl, (cur_circuit.R_low, cur_circuit.R_high), retain_graph=True))
    data_grads_scat1.set_offsets(torch.stack((train_data_band_bin[0][::subsample] / (2 * math.pi), data_grads[:, 0])).T)
    data_grads_scat2.set_offsets(torch.stack((train_data_band_bin[0][::subsample] / (2 * math.pi), data_grads[:, 1])).T)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_band_bin,
                 t=widgets.IntSlider(value=0, min=0, max=len(R_values_band_bin) - 1, step=1, description="Training Iteration", style=
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=16…
```

Figure


Transfer Function
Learned TF
TF + Samples
TF - Samples
Magnitude
Frequency (Hz)
Loss Surface
R_low = 500
R_high = 500
R_{high} (Ω)
R_{low} (Ω)
log(loss)
Derivative by Training Datapoint
R_{low} Derivatives
R_{high} Derivatives
Derivative
Frequency (Hz)
Gradients
R_{low} Grad
R_{high} Grad
Gradient
Training Iteration

(h) Learning a Band Pass Filter Bode Plot from Transfer Function Samples

In [43]:
```
def evaluate_bp_bode(freqs, low_cutoff, high_cutoff):
    return -20 * nn.ReLU()(torch.log10(freqs / low_cutoff)) + -20 * nn.ReLU()(torch.log10(high_cutoff / freqs))

# PyTorch model of the band pass bode plot
class BandPassBodePlot(nn.Module):
    def __init__(self, low_cutoff=None, high_cutoff=None):
        super().__init__()
        self.low_cutoff = nn.Parameter(torch.rand(1) * 5000 if low_cutoff is None else torch.tensor(float(low_cutoff)))
        self.high_cutoff = nn.Parameter(torch.rand(1) * 5000 if high_cutoff is None else torch.tensor(float(high_cutoff)))

    def forward(self, freqs):
        return evaluate_bp_bode(freqs, self.low_cutoff, self.high_cutoff)

# Train a given band pass bode plot
def train_bp_bode(bode, loss_fn, dataset_size, max_training_steps, lr):

    cutoff_values = [[float(bode.low_cutoff.data), float(bode.high_cutoff.data)]]
    grad_values = [[np.nan, np.nan]]
    train_data = generate_bp_training_data(dataset_size)
    print(f"Initial Cutoff Values: f_c,l = {float(bode.low_cutoff.data / (2 * math.pi)):.0f} Hz, f_c,h = {float(bode.high_cutoff.data
    iter_bar = tqdm.trange(max_training_steps, desc="Training Iter")
    for i in iter_bar:

        pred = bode(train_data[0])
        loss = loss_fn(pred, 20 * torch.log10(train_data[1])).mean()
        grad = torch.autograd.grad(loss, (bode.low_cutoff, bode.high_cutoff))
        with torch.no_grad():
            bode.low_cutoff -= lr * grad[0]
            bode.high_cutoff -= lr * grad[1]

        cutoff_values.append([float(bode.low_cutoff.data), float(bode.high_cutoff.data)])
        grad_values.append([float(grad[0].data), float(grad[1].data)])
        iter_bar.set_postfix_str(f"Loss: {float(loss.data):.3f}, f_c,l = {float(bode.low_cutoff.data / (2 * math.pi)):.0f} Hz, f_c,h =
        if loss.data < 1e-6 or (abs(grad[0].data) < 1e-6 and abs(grad[1].data) < 1e-6):
            break

    print(f"Final Cutoff Values: f_c,l = {float(bode.low_cutoff.data / (2 * math.pi)):.0f} Hz, f_c,h = {float(bode.high_cutoff.data /
    return train_data, cutoff_values, grad_values
```

In [44]:
```
bode = BandPassBodePlot()
loss_fn = lambda x, y: (x - y) ** 2    # MSE loss
lr = 1000
train_data_band_bode, cutoffs_band_bode, grad_values_band_bode = train_bp_bode(bode, loss_fn, dataset_size, max_training_steps, lr)
```

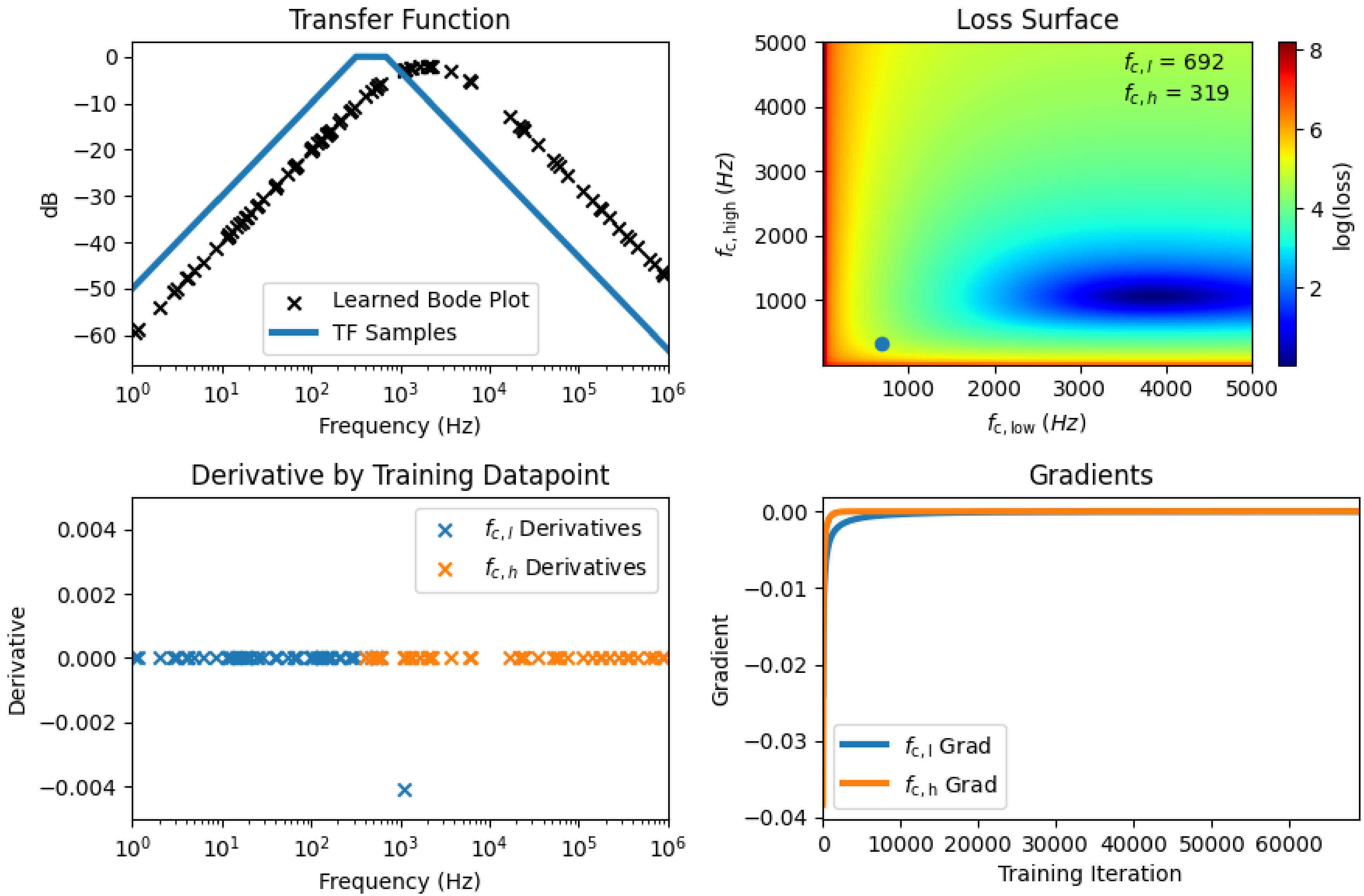
```
Initial Cutoff Values: f_c,l = 692 Hz, f_c,h = 319 Hz

Training Iter:  69%|███████    | 68909/100000 [03:49<01:43, 300.76it/s, Loss: 1.053, f_c,l = 3827 Hz, f_c,h = 1035 Hz]

Final Cutoff Values: f_c,l = 3827 Hz, f_c,h = 1035 Hz
```

In [45]:
```
# Plot transfer function over training
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 100)
train_data_mask = train_data_band_bode[1][::subsample] > cutoff_mag
ax1.scatter(train_data_band_bode[0][::subsample]/ (2 * math.pi), 20 * torch.log10(train_data_band_bode[1][::subsample]), c="k", marker
learned_tf, = ax1.semilogx(ws / (2 * math.pi), evaluate_bp_bode(ws, *cutoffs_band_bode[0]), linewidth=3)
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("dB")
ax1.legend(["Learned Bode Plot", "TF Samples"])

# Show loss surfaces for BCE and MSE Loss
eval_pts = torch.stack(torch.meshgrid((torch.arange(1, 5001, 50), torch.arange(1, 5001, 50)), indexing="ij"))
eval_vals = evaluate_bp_bode(train_data_band_bode[0][:, None, None], 2 * math.pi * eval_pts[0][None, ...], 2 * math.pi * eval_pts[1
loss_surface = loss_fn(eval_vals, 20 * torch.log10(train_data_band_bode[1])[..., None, None].expand(eval_vals.shape))
loss_surf = ax2.imshow(torch.log(loss_surface.mean(0)).T, cmap=plt.cm.jet, extent=(1, 5000, 1, 5000), aspect="auto", origin="lower")
cur_loss, = ax2.plot(cutoffs_band_bode[0][0] / (2 * math.pi), cutoffs_band_bode[0][1] / (2 * math.pi), marker="o")
cur_loss_label = ax2.annotate(f"$f_{{c,l}}$ = {cutoffs_band_bode[0][0]:.0f}\n$f_{{c,h}}$ = {cutoffs_band_bode[0][1]:.0f}", (0, 0), xyt
ax2.set_title("Loss Surface")
ax2.set_xlabel("$f_\mathrm{c,low} \; (Hz)$")
ax2.set_ylabel("$f_\mathrm{c,high} \; (Hz)$")
fig.colorbar(loss_surf, ax=ax2, label="log(loss)")

# Show loss contributions of each data point
cur_bode = BandPassBodePlot(*cutoffs_band_bode[0])
data_losses = loss_fn(cur_bode(train_data_band_bode[0][::subsample]), 20 * torch.log10(train_data_band_bode[1][::subsample]))
data_grads = torch.zeros((len(data_losses), 2))
for i, dl in enumerate(data_losses):
    data_grads[i] = torch.tensor(torch.autograd.grad(dl, (cur_bode.low_cutoff, cur_bode.high_cutoff), retain_graph=True))
data_grads_scat1 = ax3.scatter(train_data_band_bode[0][::subsample] / (2 * math.pi), data_grads[:, 0], marker="x")
data_grads_scat2 = ax3.scatter(train_data_band_bode[0][::subsample] / (2 * math.pi), data_grads[:, 1], marker="x")
ax3.set_xscale("log")
ax3.set_ylabel("Derivative")
ax3.set_xlim([1, 1e6])
ax3.set_ylim([-5e-3, 5e-3])
ax3.set_xlabel("Frequency (Hz)")
ax3.set_title("Derivative by Training Datapoint")
ax3.legend(["$f_{c,l}$ Derivatives", "$f_{c,h}$ Derivatives"])
```

```
# Show gradient at each training iteration
ax4.plot(np.arange(len(grad_values_band_bode)), grad_values_band_bode, linewidth=3)
cur_grad0, = ax4.plot(0, grad_values_band_bode[0][0], marker="o")
cur_grad1, = ax4.plot(0, grad_values_band_bode[0][1], marker="o")
ax4.set_xlabel("Training Iteration")
ax4.set_ylabel("Gradient")
ax4.set_title("Gradients")
ax4.set_xlim([-1, len(grad_values_band_bode)])
ax4.legend(["$f_\mathrm{c,l}$ Grad", "$f_\mathrm{c,h}$ Grad"])

plt.tight_layout()

# Main update function for interactive plots
def update_iter_band_bode(t=0):
    learned_tf.set_data(ws / (2 * math.pi), evaluate_bp_bode(ws, *cutoffs_band_bode[t]))
    cur_loss.set_data(cutoffs_band_bode[t][0] / (2 * math.pi), cutoffs_band_bode[t][1] / (2 * math.pi))
    cur_loss_label.set_text(f"$f_{{c,l}}$ = {cutoffs_band_bode[t][0] / (2 * math.pi):.0f}\n$f_{{c,h}}$ = {cutoffs_band_bode[t][1] / (2
    cur_grad0.set_data(t, grad_values_band_bode[t][0])
    cur_grad1.set_data(t, grad_values_band_bode[t][1])
    cur_bode = BandPassBodePlot(*cutoffs_band_bode[t])
    data_losses = loss_fn(cur_bode(train_data_band_bode[0][::subsample]), 20 * torch.log10(train_data_band_bode[1][::subsample]))
    data_grads = torch.zeros((len(data_losses), 2))
    for i, dl in enumerate(data_losses):
        data_grads[i] = torch.tensor(torch.autograd.grad(dl, (cur_bode.low_cutoff, cur_bode.high_cutoff), retain_graph=True))
    data_grads_scat1.set_offsets(torch.stack((train_data_band_bode[0][::subsample] / (2 * math.pi), data_grads[:, 0])).T)
    data_grads_scat2.set_offsets(torch.stack((train_data_band_bode[0][::subsample] / (2 * math.pi), data_grads[:, 1])).T)
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_band_bode,
                 t=widgets.IntSlider(value=0, min=0, max=len(cutoffs_band_bode) - 1, step=1, description="Training Iteration", style=
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=68…
```

Figure


Transfer Function
dB
Frequency (Hz)
Learned Bode Plot
TF Samples
Loss Surface
$f_{c,l}$ = 692
$f_{c,h}$ = 319
$f_{c,\mathrm{high}}$ (Hz)
$f_{c,\mathrm{low}}$ (Hz)
log(loss)
Derivative by Training Datapoint
Derivative
Frequency (Hz)
$f_{c,l}$ Derivatives
$f_{c,h}$ Derivatives
Gradients
Gradient
Training Iteration
$f_{c,\mathrm{l}}$ Grad
$f_{c,\mathrm{h}}$ Grad

(i) Learn a Color Organ Circuit

In [46]:
```
# PyTorch model of the color organ circuit
class ColorOrganCircuit(nn.Module):
    def __init__(self, R_low=None, R_high=None, R_band_low=None, R_band_high=None):
        super().__init__()
        self.low = LowPassCircuit(R_low)
        self.high = HighPassCircuit(R_high)
        self.band = BandPassCircuit(R_band_low, R_band_high)

    def forward(self, freqs):
        return torch.stack((self.low(freqs), self.band(freqs), self.high(freqs)))


# Generate training data in a uniform log scale of frequences, then evaluate using the true transfer function
R_low_des = 1 / (2 * math.pi * 800 * cap_value)
R_band_low_des = 1 / (2 * math.pi * 4000 * cap_value)
R_band_high_des = 1 / (2 * math.pi * 1000 * cap_value)
R_high_des = 1 / (2 * math.pi * 5000 * cap_value)
def generate_co_training_data(n):
    rand_ws = 2 * math.pi * torch.pow(10, torch.rand(n) * 6)
    labels = torch.stack((evaluate_lp_circuit(rand_ws, R_low_des), evaluate_bp_circuit(rand_ws, R_band_low_des, R_band_high_des), eval
    return rand_ws, labels

# Train a given color organ circuit
def train_co_circuit(circuit, loss_fn, dataset_size, max_training_steps, lr):

    R_values = [[float(circuit.low.R.data), float(circuit.band.R_low.data), float(circuit.band.R_high.data), float(circuit.high.R.data
    grad_values = [[np.nan, np.nan, np.nan, np.nan]]
    train_data = generate_co_training_data(dataset_size)
    print(f"Initial Resistor Values: LP: {float(circuit.low.R.data):.0f} Ohms, BP (Low): {float(circuit.band.R_low.data):.0f} Ohms, BP

    iter_bar = tqdm.trange(max_training_steps, desc="Training Iter")
    for i in iter_bar:
        pred = circuit(train_data[0])
        loss = loss_fn(pred, (train_data[1] > cutoff_mag).float()).mean()
        grad = torch.autograd.grad(loss, (circuit.low.R, circuit.band.R_low, circuit.band.R_high, circuit.high.R))
        with torch.no_grad():
            circuit.low.R -= lr * grad[0]
            circuit.band.R_low -= lr * grad[1]
            circuit.band.R_high -= lr * grad[2]
            circuit.high.R -= lr * grad[3]
```

```
        R_values.append([float(circuit.low.R.data), float(circuit.band.R_low.data), float(circuit.band.R_high.data), float(circuit.hig
        grad_values.append([float(grad[0].data), float(grad[1].data), float(grad[2].data), float(grad[3].data)])
        iter_bar.set_postfix_str(f"Loss: {float(loss.data):.3f}, Rs = {float(circuit.low.R.data):.0f}, {float(circuit.band.R_low.data)
        if loss.data < 1e-6 or (abs(grad[0].data) < 1e-6 and abs(grad[1].data) < 1e-6):
            break

    print(f"Final Resistor Values: LP: {float(circuit.low.R.data):.0f} Ohms, BP (Low): {float(circuit.band.R_low.data):.0f} Ohms, BP (
    print(f"Final Cutoff Frequencies: LP: {1 / (2 * math.pi * cap_value * float(circuit.low.R.data)):.0f} Hz, BP (Low): {1 / (2 * mat
    return train_data, R_values, grad_values
```

In [47]:

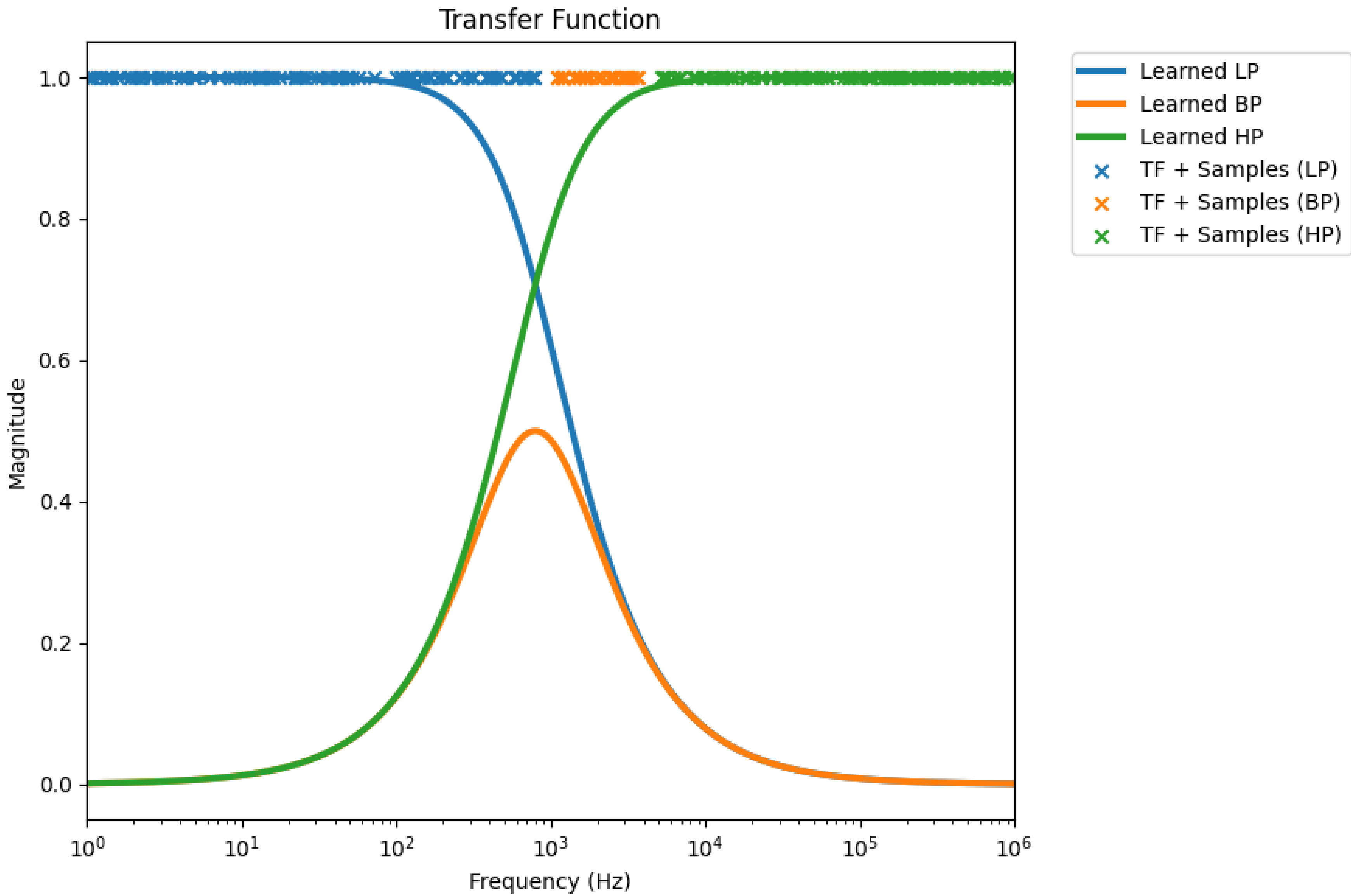
```
co = ColorOrganCircuit(200, 200, 200, 200)
loss_fn = lambda x, y: (x - (0.3 + 0.7 * y)) ** 2   # weighted MSE loss
lr = 500
train_data_co, R_values_co, grad_values_co = train_co_circuit(co, loss_fn, dataset_size, max_training_steps, lr)
```

```
Initial Resistor Values: LP: 200 Ohms, BP (Low): 200 Ohms, BP (High): 200 Ohms, HP: 200 Ohms

Training Iter:   8%|█          | 8383/100000 [00:39<07:07, 214.47it/s, Loss: 0.047, Rs = 210, 39, 159, 34]

Final Resistor Values: LP: 210 Ohms, BP (Low): 39 Ohms, BP (High): 159 Ohms, HP: 34 Ohms
Final Cutoff Frequencies: LP: 759 Hz, BP (Low): 4120 Hz, BP (High): 999 Hz, HP: 4656 Hz
```

In [48]:
```
# Plot transfer function over training
fig, ax1 = plt.subplots(1, 1, figsize=(9, 6))
ws = 2 * math.pi * 10 ** torch.linspace(0, 6, 1000)
subsample = int(dataset_size / 250)
train_data_mask = train_data_co[1][:, ::subsample] > cutoff_mag
learned_tf1, = ax1.semilogx(ws / (2 * math.pi), evaluate_lp_circuit(ws, R_values_co[0][0]), linewidth=3)
learned_tf2, = ax1.semilogx(ws / (2 * math.pi), evaluate_bp_circuit(ws, *R_values_co[0][1:3]), linewidth=3)
learned_tf3, = ax1.semilogx(ws / (2 * math.pi), evaluate_hp_circuit(ws, R_values_co[0][-1]), linewidth=3)
ax1.scatter(train_data_co[0][::subsample][train_data_mask[0]] / (2 * math.pi), np.ones(train_data_mask[0].sum()), c=learned_tf1.get_co
ax1.scatter(train_data_co[0][::subsample][train_data_mask[1]] / (2 * math.pi), np.ones(train_data_mask[1].sum()), c=learned_tf2.get_co
ax1.scatter(train_data_co[0][::subsample][train_data_mask[2]] / (2 * math.pi), np.ones(train_data_mask[2].sum()), c=learned_tf3.get_co
# ax1.scatter(train_data_co[0][::subsample][(~train_data_mask).all(0)] / (2 * math.pi), np.zeros((~(train_data_mask.any(0))).sum(), c
ax1.set_xlim([1, 1e6])
ax1.set_title("Transfer Function")
ax1.set_xlabel("Frequency (Hz)")
ax1.set_ylabel("Magnitude")
ax1.legend(["Learned LP", "Learned BP", "Learned HP",
            "TF + Samples (LP)", "TF + Samples (BP)", "TF + Samples (HP)",
            "TF - Samples"], bbox_to_anchor=(1.05, 1), loc='upper left', ncol=1)

plt.tight_layout()

# Main update function for interactive plots
def update_iter_co(t=0):
    learned_tf1.set_data(ws / (2 * math.pi), evaluate_lp_circuit(ws, R_values_co[t][0]))
    learned_tf2.set_data(ws / (2 * math.pi), evaluate_bp_circuit(ws, *R_values_co[t][1:3]))
    learned_tf3.set_data(ws / (2 * math.pi), evaluate_hp_circuit(ws, R_values_co[t][-1]))
    fig.canvas.draw_idle()

# Include sliders for relevant quantities
ip = interactive(update_iter_co,
                 t=widgets.IntSlider(value=0, min=0, max=len(R_values_co) - 1, step=1, description="Training Iteration", style={'desc
ip
```

```
interactive(children=(IntSlider(value=0, description='Training Iteration', layout=Layout(width='100%'), max=83…
```

Figure


Transfer Function
Learned LP
Learned BP
Learned HP
TF + Samples (LP)
TF + Samples (BP)
TF + Samples (HP)
Magnitude
1.0
0.8
0.6
0.4
0.2
0.0
10^0
10^1
10^2
10^3
10^4
10^5
10^6
Frequency (Hz)

Visualizing the computation graph for the Color Organ

In [49]:
```
from torchviz import make_dot
make_dot(co(generate_co_training_data(dataset_size)[0]), params=dict(co.named_parameters()))
```

Out[49]:
```
<graphviz.graphs.Digraph at 0x7fa80cdb9990>
```