HW1.1_Introduction

January 19, 2025

[ ]:
```
# %%bash
# If you are on Google Colab, this sets up everything needed.
!(stat -t /usr/local/lib/*/dist-packages/google/colab > /dev/null 2>&1) && exit
! wget -O requirements.txt https://cs7150.baulab.info/2022-Fall/setup/
 ↪hw1_requirements.txt
! pip install -r requirements.txt
# If you are not on Google Colab, you can run these pip requirements on your␣
 ↪own command-line.
```

1 Homework 1. Foundations, and How to Code in Pytorch

1.1 Learning Objective

The goal of this homework is to get you familar with some of the foundational mathematical and programming tools used in deep learning, so we will review a little bit of calculus and linear algebra and introduce you to the pytorch API.

1.2 Introduction

You may already be familiar with pytorch, and if so, great - dive in to answer the questions below.

If you are new to pytorch, your first step is to get the necessary background. **Read and work through the notebooks in the David's tips on how to read pytorch series**, which will give you an overview of GPU usage, Autograd, optimizer classes, torch Modules, and data loading. (Don't hand in those notebooks; no points for working through those other notebooks, though they are highly recommended and will be helpful for learning to answer the questions in the current notebook.) Hand in this current notebook completed.

1.3 Readings

The following readings are not necessary to do this notebook, but you may find them interesting if you want to develop your sense for the origins of some of the important ideas in deep networks.

The practice of modeling a neural network as a computational object was pioneered in this classic paper by Warren McCulloch and Walter Pitts, and we will follow along with some of their constructions envisioning neural networks as graphs that can implement logic:

Warren S. McCulloch and Walter Pitts, A Logical Calculus of the Ideas Immanent in Nervous Activity, 1943.

Pytorch is pretty new, but its modular architecture has a long history, with roots in the torch project, which is itself based on the ideas in this paper:

Léon Bottou and Patrick Gallinari, A Framework for the Cooperation of Learning Algorithms, 1990.

We will talk about the cross-entropy loss. The use of cross-entropy loss for neural network training has its roots root in the late-1980s realization that often the units of measurement that should be used by neural networks are units of probability, not only because of its elegance, but also its excellent performance.

Sarah Solla, Esther Levin and Michael Fleisher, Accelerated Learning in Layered Neural Networks, 1988.

We will play with a state-of-the-art diffusion model that was released recently. The whole pipeline is complex, and it will take the whole semester to learn how its parts work, but a user's view of the model is descibed well in a blog post here:

Suraj Patil, Pedro Cuenca, Nathan Lambert and Patrick von Platen, Stable Diffusion with Diffusers, 2022.

More papers about the diffusion model are listed in that exercise.

1.4 Academic Integrity, Citations, and Collaborations

In all your homework, you must explicitly cite any sources (people or any materials) you consult.

In our class homework assignments, you should think about the problems yourself first before consulting outside help. And when you do seek out help, we strongly advise you to find a fellow classmate to talk with and work together rather than copying an answer from the internet. You will all learn much more by thinking collaboratively and explaining ideas to one another.

But if you are alone and stuck and you find some really useful insight on Stack Overflow or Github or a blog or some chat channel thread on Discord, it is not cheating to use and learn from that insight if you cite your sources. Learning from the internet is acceptable as long as you **do not misrepresent somebody else's work as your own**. Include citations in your writeup text or in comments in your code.

In this first exercise you will Google for a nice solution to a real problem and **make a proper citation of your source for the solution**. In this specific problem you will only get points if you look it up and cite the source. In general, avoid Googling and peeking at answers for homework problems, but in this problem we ask you to do it explicitly, and you should continue this practice of citing all your sources and collaborators in the future. Linked citations are a very cool, polite, honest and useful practice in real life. In classwork, citations are *required*.

1.5 Exercise 1.1: calculus review and autograd derivatives

The pytorch autograd framework is usually used to compute first derivatives, but it is perfectly capable of calculating higher order partial derivatives. In this exercise we will try it out.

In the code below, a function $p_0(x) =$ `polynomial(x)` $= x^4 - 2x^3 + 3x^2 - 4x + 5$ is given, and it is applied to a batch of 100 values of x in the range $[-2, 3]$ given as `x = torch.linspace(-2.0,`

`3.0, 100)`. The batch of 100 values of $p_0(x)$ is stored as `y`, and the results are plotted.

Question 1.1.1 Fill in the formula below. Understand why we suggest using gradients of `y.sum()`.

If $y_i = p_0(x_i)$ and $s = \sum_i y_i$, then the gradient $\nabla_x s$ is a vector that has components $\partial s / \partial x_i$, given by:

$$\frac{\partial s}{\partial x_i} = \boxed{4x_i^3 - 6x_i^2 + 6x_i - 4}$$

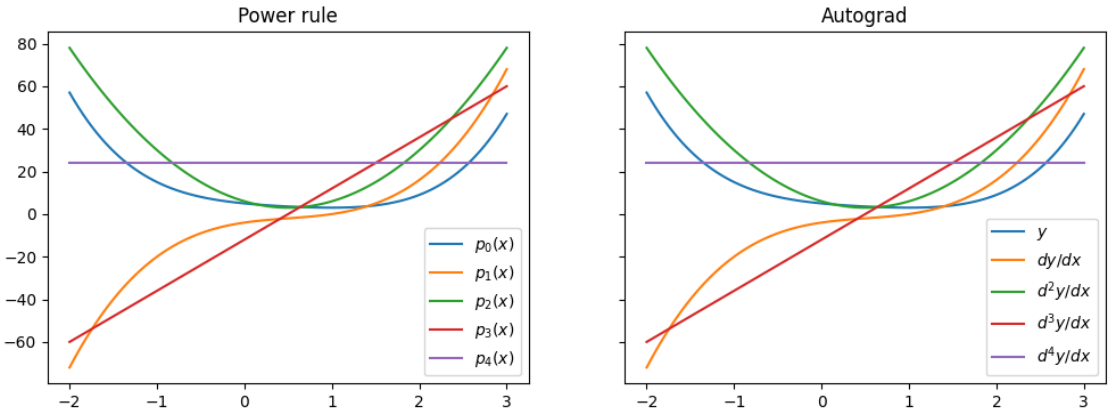
Question 1.1.2 Fill in the code below. To create your ground truth solutions, apply calculus by hand (just use the Power Rule) to compute the derivative of the polynomial, and put the formula in as the definition of `p1(x)`. Plot the reults. Do the same for `p2`, `p3` and `p4` for successive derivatives. To plot results correctly, the results should be a batch of the same size as the input; you might need to use `torch.ones_like(x)` or `x**0` or something similar to make this work in some cases.

Then make pytorch autograd do the work. Enable gradient computations on `x` by adding the `requires_grad` flag when it is created, and then modify the line that defines `dy_dx` to uncomment the call to `torch.autograd.grad` and make it work. Plot the results and make sure it looks the same as `p1`. Add more calls to `torch.autograd.grad` to compute the 2nd, 3rd, and 4th derivatives, and plot the results, and make sure they look right. The 2nd derivative should come from applying `grad` to the 1st derivative, and so on. Take a look at the advice about higher-order gradients from the pytorch documentation for `torch.autograd.grad` if you get stuck.

[4]:
```
import torch


def polynomial(x):

    return x**4 - 2 * x**3 + 3 * x**2 - 4 * x + 5


# MODIFY THE CODE BELOW TO DEFINE p1, p2, p3, p4 and d2y_x, d3y_x, and d4y_x


def p1(x):

    return 4 * x**3 - 6 * x**2 + 6 * x - 4  # The derivative of polynomial(x)


def p2(x):

    return 12 * x**2 - 12 * x + 6  # The derivative of p1(x)


def p3(x):

    return 24 * x - 12  # The derivative of p2(x)
```

```
def p4(x):

    return 24 * torch.ones_like(x)  # The derivative of p3(x)


x = torch.linspace(-2.0, 3.0, 100, requires_grad=True)  # Adding requires_grad
y = polynomial(x)


[dy_dx] = torch.autograd.grad(
    y.sum(), [x], create_graph=True
)  # torch.autograd.grad(y.sum(), [x])


[d2y_dx] = torch.autograd.grad(dy_dx.clone().sum(), [x], create_graph=True)
# The 2st derivative of y.sum() w.r.t. to x


[d3y_dx] = torch.autograd.grad(d2y_dx.clone().sum(), [x], create_graph=True)
# The 3rd derivative of y.sum() w.r.t. to x


[d4y_dx] = torch.autograd.grad(
    d3y_dx.sum(), [x], create_graph=True
)  # The 4th of y.sum() w.r.t. to x


# DO NOT CHANGE THE PLOTTING CODE OR TESTS BELOW


import matplotlib.pyplot as plt


fig, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 4), sharey=True)


with torch.no_grad():

    ax1.set_title("Power rule")

    for i in range(0, 5):

        ax1.plot(x, [polynomial, p1, p2, p3, p4][i](x), label=f"$p_{i}(x)$")

    ax1.legend()
```

```
    ax2.set_title("Autograd")

    ax2.plot(x, y, label="$y$")

    ax2.plot(x, dy_dx, label="$dy/dx$")

    for i in [2, 3, 4]:
        ax2.plot(x, [d2y_dx, d3y_dx, d4y_dx][i - 2], label=f"$d^{i}y/dx$")

    ax2.legend()


assert all((p1(x) - dy_dx).abs() < 1e-5)


assert all((p2(x) - d2y_dx).abs() < 1e-5)


assert all((p3(x) - d3y_dx).abs() < 1e-5)


assert all((p4(x) - d4y_dx).abs() < 1e-5)
```



1.6 Exercise 1.2: McCullough-Pitts Neural Networks

The modern conception of artificial neural networks is essentially the same as the model originally devised by McCullough and Pitts in their seminal 1943 paper. In that work, they observed that a biological neuron could be seen as an object that adds up its inputs, possibly weighting some inputs differently from others, and then firing an output only once some threshold is reached. This can be modeled as a weighted sum followed by a nonlinearity:

McCullough and Pitts reasoned about such neurons individually or in very small networks, and

they asked: what is the computational power of such networks? Can they reproduce any logical computation? We will follow along with their exploration by constructing networks for the various logical operations that can be created with two binary inputs.

We begin by creating a `torch.nn.Module` for the step-function nonlinearity. We call it `Sign` because it is based on the `sign` function, returning `+1` for positive numbers and `-1` for negative numbers. Like any torch `Module`, it will be a callable object. We define the calling behavior in our `forward` method. Notice that, as is typical in pytorch, our `Sign` module is designed to be able to operate on batches of data gathered in a `Tensor`.

[5]:
```
from torch import Tensor


class Sign(torch.nn.Module):
    """
    The Sign nonlinearity is a step function that returns +1 for all positive
    numbers and -1 for all negative numbers.  Zero stays as zero.
    """

    def forward(self, x):
        return x.sign()


demo = Sign()
x = Tensor([-3.14, 1e-6, 0.0, -0.0])
print("Sign of", x, "is", demo(x))
```

```
Sign of tensor([-3.1400e+00,  1.0000e-06,  0.0000e+00, -0.0000e+00]) is
tensor([-1.,  1.,  0.,  0.])
```

Next we implement a slightly more elaborate `torch.nn.Module` for a McCullough-Pitts neuron.

The `McCulloughPittsNeuron` torch Module object computes both the weighted sum (as a `torch.nn.Linear` operation called `summation`) and the `Sign` activation nonlinearity. The `forward` method does both steps.

The `McCulloughPittsNeuron` object can take any number of inputs; the number and names of the inputs and output are configured in the constructor, and the multiple named inputs are provided to the neuron as a dictionary contaiing `Tensors`. They are designed to be wired together in `torch.nn.Sequential` sequences.

Read the code below to see how to configure a small network of `McCulloughPittsNeurons`.

[6]:
```
import torch
from matplotlib import pyplot as plt
from baukit import PlotWidget, show


class McCulloughPittsNeuron(torch.nn.Module):
    """
```

```
    A McCullough-Pitts Neuron.  It computes a weighted sum of any number of inputs,
    then it thresholds the output through a nonlinear activation step function.
    It pulls named inputs from an input dictionary and puts output into the
    dictionary.  That allows networks to be created by sequencing neurons and
    connecting them by using dictionary names.

    Examples:

        net = McCulloughPittsNeuron(
                weight_a = 0.5,
                weight_b = -0.3,
                weight_c = 2.0,
                bias     = 1.0)
        print(net(dict(
                a=Tensor([1.0]),
                b=Tensor([-1.0]),
                c=Tensor([-1.0])))['out'])

    The above creates a single neuron with three inputs a, b, and c plus some bias.
    It is invoked by providing a dictionary of all the inputs as tensors.

        net = torch.nn.Sequential(
            McCulloughPittsNeuron(weight_a=-1.0, weight_b=1.0, output_name='d'),
            McCulloughPittsNeuron(weight_b=1.0, weight_d=1.0, bias=1.0),
        )
        print(net(dict(a=Tensor([1.0]), b=Tensor([-1.0])))['out'])

    The above creates and runs a network of two neurons in this configuration:
    ```
              a -----> +-----------+
                       | Neuron 0 | ---> d --+
              b ---+-> +-----------+   +--> +-----------+
                   | | Neuron 1 | ---> out
                   +------------------------------> +-----------+
    ```
    As the sequence is run, the dictionary grows; after the first neuron is run,
    the dictionary contains a, b, and d.  After the second neuron is run, the
    final dictionary contains a, b, d, and out.
    """

    def __init__(self, bias=0.0, activation=Sign, output_name="out", **kwargs):
        """
        Construct a neuron by specifying any number of input weights in the arguments:
```

```
            weight_a:    The weight for the 'a' input.
                         Each `weight_x` in the constructor adds an input named␣
↪'x'.
            bias:        The constant bias to add to the weighted sum.
            output_name: The output name, defaults to 'out'.
            activation:  The nonlinearity to use; defaults to the "Sign" step␣
↪function.
        """
        super().__init__()

        # We use the pytorch Linear module with a one-dimenaional output
        self.summation = torch.nn.Linear(len(kwargs), 1)
        self.activation = None if activation is None else activation()
        self.output_name = output_name
        self.input_names = []
        with torch.no_grad():
            self.summation.bias[...] = bias
            for k, v in kwargs.items():
                assert k.startswith("weight_"), f"Bad argument {k}"
                self.summation.weight[0, len(self.input_names)] = v
                self.input_names.append(k[7:])

    def forward(self, inputs):
        """
        The inputs should be a dictionary containing the expected input keys.
        The results are computed.  Then the return value will be a copy of the
        input dictionary, with the additional output tensor added.
        """
        state = inputs.copy()
        assert self.output_name not in state, f"Multiple {self.output_name}'s␣
↪conflict"
        x = torch.stack([inputs[v] for v in self.input_names], dim=1)
        x = self.summation(x)[:, 0]
        if self.activation is not None:
            x = self.activation(x)
        state[self.output_name] = x
        return state

    def extra_repr(self):
        return f"input_names={self.input_names}, output_name='{self.
↪output_name}'"


def visualize_logic(nets, arg1="a", arg2="b"):
    """
    Pass any number of McCullough-Pitts neurons or neural networks with two
    inputs named 'a' and 'b', and it will visualize all of their logic, using
```

```
    white squares to indicate +1, black squares to indicate -1, and orange
    squares to indicate intermediate values.
    """
    grid = torch.Tensor(
        [
            [
                [-1.0, 1.0],
                [-1.0, 1.0],
            ],
            [
                [1.0, 1.0],
                [-1.0, -1.0],
            ],
        ]
    )
    a, b = grid

    def make_viz(n, case=()):
        if isinstance(n, list):
            return [make_viz(net, case + (str(i + 1),)) for i, net in␣
↪enumerate(n)]

        def make_plot(fig):
            with torch.no_grad():
                out = n({arg1: a.view(-1), arg2: b.view(-1)})["out"].view(a.
↪shape)
            [ax] = fig.axes
            ax.imshow(out, cmap="hot", extent=[-2, 2, -2, 2], vmin=-1, vmax=1)
            ax.invert_yaxis()
            ax.xaxis.tick_top()
            ax.tick_params(length=0)
            ax.set_xticks([-1, 1], [f"{arg1}=-1", f"{arg1}=1"])
            ax.set_yticks([-1, 1], [f"{arg2}=-1", f"{arg2}=1"])

        return [
            PlotWidget(make_plot, figsize=(1.1, 1.1), dpi=100,␣
↪bbox_inches="tight"),
            show.style(margin="0 0 20px 45%", textAlign="right"),
            f'case {" ".join(case)}',
        ]

    show([show.WRAP, make_viz(nets)])
```

Below is an example of a two small networks using the `McCulloughPittsNeuron`: One single-neuron network, and one two-neuron network. The behavior of the networks on ± 1 input for `a` and `b` is visualized, with white for $+1$ output and black for -1; orange indicates an indecisive 0.

The networks correspond to the code below:

[7]:
```
visualize_logic(
    [
        # First network: just one neuron.
        McCulloughPittsNeuron(weight_a=-1.0, weight_b=0.5, bias=0.5),
        # Second network: two neurons hooked together.
        torch.nn.Sequential(
            McCulloughPittsNeuron(
                weight_a=-1.0, weight_b=1.0, bias=0.0, output_name="d"
            ),
            McCulloughPittsNeuron(weight_b=0.5, weight_d=0.5, bias=1.0),
        ),
    ]
)
```

```
<baukit.show.HtmlRepr at 0x7b98061f8d10>
```

Exercise 3.1 Use `McCulloughPittsNeurons` to implement and visualize all the following cases.

How many of the cases are able to be handled using a **single** neuron? 12

What are the names for the logical operations that require multiple neurons? XOR and XNOR

Put your code for implemeintting and visualizing each of the 4 neural networks (or single-neuron networks) below:

[8]:
```
# Modify this to implement and visualize the networks

visualize_logic(
    [
        # list your 14 networks here.
        McCulloughPittsNeuron(weight_a=1, weight_b=1, bias=1),
        McCulloughPittsNeuron(weight_a=1, weight_b=-1, bias=1),
        McCulloughPittsNeuron(weight_a=-1, weight_b=-1, bias=1),
        McCulloughPittsNeuron(weight_a=-1, weight_b=1, bias=1),
        McCulloughPittsNeuron(weight_a=1, weight_b=1, bias=-1),
        McCulloughPittsNeuron(weight_a=1, weight_b=-1, bias=-1),
        McCulloughPittsNeuron(weight_a=-1, weight_b=-1, bias=-1),
        McCulloughPittsNeuron(weight_a=-1, weight_b=1, bias=-1),
        McCulloughPittsNeuron(weight_a=1, weight_b=0, bias=0),
        McCulloughPittsNeuron(weight_a=0, weight_b=1, bias=0),
        # Case 6 + Case 8
        torch.nn.Sequential(
            McCulloughPittsNeuron(weight_a=-1, weight_b=1, bias=-1,␣
 ↪output_name="c"),
            McCulloughPittsNeuron(weight_a=1, weight_b=-1, bias=-1,␣
 ↪output_name="d"),
            McCulloughPittsNeuron(weight_c=1, weight_d=1, bias=0.5),
        ),
        # Case 5 + Case 7
```

```
        torch.nn.Sequential(
            McCulloughPittsNeuron(weight_a=1, weight_b=1, bias=-1,␣
 ↪output_name="c"),
            McCulloughPittsNeuron(weight_a=-1, weight_b=-1, bias=-1,␣
 ↪output_name="d"),
            McCulloughPittsNeuron(weight_c=1, weight_d=1, bias=0.5),
        ),
        McCulloughPittsNeuron(weight_a=-1, weight_b=0, bias=0),
        McCulloughPittsNeuron(weight_a=0, weight_b=-1, bias=0),
    ]
)
```

```
<baukit.show.HtmlRepr at 0x7b98003d7810>
```

1.7 Exercise 1.3: Softmax, KL, Cross-Entropy, and Squared Error

In the 1980's, researchers like Sarah Solla and John Hopfield discovered that networks are very effective when trained to model *probabilities* instead of just discrete binary logic. Even in the case where the output should make a choice between two alternatives, it is often best to have the network output its estimate of the *probability distribution* of the choice to be made, rather than just a 0 or a 1.

So in modern deep learning, we will often pursue the goal of matching some true vector of discrete probabilities $y \in \mathbb{R}^n$ by computing some model-predicted vector of probabilities $p \in \mathbb{R}^n$ that is derived from some raw neural network output $z \in \mathbb{R}^n$, and then measuring its deviation from some true distribution y.

This problem of generating a predicted probability distribution p to match some observed truth y is is so central and common in deep networks that you should make sure you are very familiar with the specific clever functions that everybody uses to do it, and why this approach works so well.

The modeling of p and the measurement of the distance to y is almost always done in the same way: **softmax** and **cross-entropy**.

Here is what a the softmax-cross-entropy computation looks like, when modeling a choice between two alternatives:

On the left we have some numbers z that are computed with the intention of modeling some choices in the real world. On the right we have a categorical probablity distribution y that is the true distribution of the choices actually observed in the world. (In our figure we have just drawn two choices, but a big model could estimate a distribution over hundreds or thousands of choices.) In the middle, we have two steps. First, p is the result of using the "softmax" function to convert the arbitrarily-scaled numbers z_i to nicely-scaled numbers p_i between 0 and 1 that could be interpreted as a categorical probablity distribution. Then to summarize the difference between the calculated p and the true y, some loss L is computed, where L is a single number that will be small if the vectors p and y are close. When working with categorical probabilities, L is almost always the cross-entropy loss function, but other choices could be used.

Below we introduce both the softmax and the cross-entropy (CE) loss function, and we also compare it to Kullback–Leibler (KL) divergence, as well as squared Euclidean vector distance, which is also known as the squared-error (SE) loss.

Question 1.2.1. Jacobians and the the softmax function.

The **softmax** function $p = \text{softmax}(z) : \mathbb{R}^n \to \mathbb{R}^n$ is defined as:

$$\text{softmax}(z)_i = p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

It is used to convert an vector of arbitrary score numbers z called *logits* into a vector p that is a valid categorical probability distribution. Fill in the following:

Write the simplest expression for the sum of $\text{softmax}(z)_i$ over all i:

$$\sum_i p_i = \sum_i \frac{e^{z_i}}{\sum_j e^{z_j}} = \boxed{1}$$

The input to softmax z are called *logits* because they can be through of as expressing probabilities on a logistic or log scale. Now suppose we have some new logits z^* which form a vector that is shifted from z by k in all dimensions, where $z_i^* = z_i + k$. How does such a shift affect the softmax? Work it out:

Assuming we have $p = \text{softmax}(z)$, write the simplest expression for $p^* = \text{softmax}(z^*) = \text{softmax}(z + k)$ in terms of only the original p_i and k:

$$p_i^* = \boxed{\frac{e^{z_i^*}}{\sum_j e^{z_j^*}} = \frac{e^{z_i+k}}{\sum_j e^{z_j+k}} = \frac{e^{z_i} \cdot e^k}{\sum_j e^{z_j} \cdot e^k} = \frac{e^{z_i}}{\sum_j e^{z_j}} = p_i}$$

This remarkable property means that the output of softmax does not depend so much on the specific values of z_i, but on the differences between the z_i.

It is useful to know the derivatives of softmax. Remember that the Jacobian of a vector function $f(z) : \mathbb{R}^n \to \mathbb{R}^m$ is the matrix

$$\mathbf{J}_f(z) = \begin{bmatrix} \frac{\partial f_1}{\partial z_1} & \frac{\partial f_1}{\partial z_2} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \frac{\partial f_2}{\partial z_1} & \frac{\partial f_2}{\partial z_2} & \cdots & \frac{\partial f_2}{\partial z_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_m}{\partial z_1} & \frac{\partial f_m}{\partial z_2} & \cdots & \frac{\partial f_m}{\partial z_n} \end{bmatrix}$$

Work out the Jacobian for softmax in the case where $n = 2$, writing the solutions for each partial derivative $\frac{\partial p_i}{\partial z_j}$. Write each partial derivative it in the simplest form in terms of p_i (and try to eliminate z_i). It might be helpful to combine terms by using the fact that $p_1 + p_2$ is a constant. Try to work it out yourself even though this problem is solved all over the web. Remember to cite your sources if you get help on the internet or with an AI.

$$\mathbf{J}_{\text{softmax}}(z) = \mathbf{J}_p(z) = \begin{bmatrix} \frac{\partial p_1}{\partial z_1} & \frac{\partial p_1}{\partial z_2} \\ \frac{\partial p_2}{\partial z_1} & \frac{\partial p_2}{\partial z_2} \end{bmatrix} = \begin{bmatrix} \boxed{p1 \cdot p2} & \boxed{-p1 \cdot p2} \\ \boxed{-p2 \cdot p1} & \boxed{p2 \cdot p1} \end{bmatrix}$$

References: * neuralthreads. (2021, December 6). Softmax function—It is frustrating that everyone talks about it but very few talk about its…. Medium. https://neuralthreads.medium.com/softmax-function-it-is-frustrating-that-everyone-talks-about-it-but-very-few-talk-about-its-54c90b9d0acd

Question 1.2.2. KL divergence, Cross-entropy, and mean squared error loss.

To measure and optimize the goal of matching p to some true real-world distribution y, we will need to define some number that summarizes the difference beween y and p. There are several natural possibilities to quantify the difference. Since both y and p are n-dimensional vectors, one natural choice is to look at the squared Euclidean distance between the two vectors; this is known as the **squared error** loss:

$$\mathrm{SE}(y,p) = ||y-p||^2 = \sum_i (y_i - p_i)^2$$

What is the value of SE if $y = p$?

$$\mathrm{SE}(y,y) = \boxed{0}$$

What is the partial derivative of $\mathrm{SE}(y,p)$ with respect to the ith component p_i? It should be possible to express the answer in terms of just y_i and p_i.

$$\frac{\partial\,\mathrm{SE}(y,p)}{\partial p_i} = \boxed{\frac{\partial}{\partial p_i}\sum (y_i - p_i)^2 = \frac{\partial}{\partial p_i}(y_i-p_i)^2 = -2(y_i-p_i) = 2(p_i-y_i)}$$

A different choice for comparing y and p is the famous **KL divergence** (Wikipedia article here) which is defined as

$$\mathrm{KL}(y;p) = \sum_i y_i \log \frac{y_i}{p_i}$$

What is the value of KL divergence if $y = p$?

$$\mathrm{KL}(y;y) = \boxed{0}$$

KL divergence can be written as the difference between entropy $\mathrm{H}(y) = -\sum_i y_i \log y_i$ and *cross-entropy* $\mathrm{CE}(y;p) = -\sum_i y_i \log p_i$ as follows:

$$\mathrm{KL}(y;p) = \sum_i y_i \log y_i - \sum_i y_i \log p_i = \mathrm{CE}(y;p) - \mathrm{H}(y)$$

$$\mathrm{CE}(y;p) = -\sum_i y_i \log p_i$$

Since $H(y)$ is a constant that does not depend on the model outputs p, the shape of the cross-entropy loss is the same as the KL loss, just shifted by a constant. In particular, when looking at derivatives of negative CE with respect to components of p, they are the same as derivatives of KL with respect to components of p. Let us compute some of those derivatives.

What is the partial derivative of $\mathrm{CE}(y;p)$ with respect to the ith component p_i? It should be possible to express the answer in terms of just y_i and p_i.

$$\frac{\partial\,\mathrm{CE}(y;p)}{\partial p_i} = \boxed{\frac{\partial}{\partial p_i}\left(-\sum y_i \log p_i\right) = y_i \cdot \frac{\partial\,(-\log p_i)}{\partial p_i} = y_i \cdot \left(-\frac{1}{p_i}\right) = -\frac{y_i}{p_i}}$$

Convince yourself that this is the same as $\frac{\partial\,\mathrm{KL}(y;p)}{\partial p_i}$.

Let's go further back from p_i and understand partial derivatives with respect to z_i. Remember how the chain rule works over vector functions: for example if we wish to compute $\partial L/\partial z_1$, we must consider multiple paths, both the path through p_1 and the path through p_2.

Question 1.2.3. Gradient of negative CE (or KL) loss on softmax, and gradient of SE loss on softmax.

Using the chain rule to combine answers for 2.3 and 2.4, compute the following partial derivative of cross-entropy with respect to the first component z_1. You should work to find simple expressions in terms of p_1 and y_1 instead of making a messy expression with the z_i. To simplify terms, you may find it useful to remember thet $y_1 + y_2 = 1$ and $p_1 + p_2 = 1$.

$$\frac{\partial\,\mathrm{CE}(y;p)}{\partial z_1} \boxed{\begin{aligned} &= \frac{\partial CE(y,P)}{\partial p_1} \cdot \frac{\partial p_1}{\partial z_1} + \frac{\partial CE(y,P)}{\partial p_2} \cdot \frac{\partial p_2}{\partial z_1} \\ &= \frac{-y_1}{p_1} \cdot p_1 p_2 + \frac{-y_2}{p_2} \cdot (-p_2 p_1) \\ &= -y_1 p_2 + p_1 y_2 \\ &= -y_1(1-p_1) + p_1 y_2 \\ &= p_1 y_1 + p_1 y_2 - y_1 \\ &= p_1(y_1 + y_2) - y_1 \\ &= p_1 - y_1 \end{aligned}}$$

Try to work it out on your own. If you consult the internet, please add a citation.

Next, compute the analogous partial derivative of SE with repect to z_1. Again, keep the expression as simple as you can, using only y_1 and p_1 if you can. Hint: it is a polynomial that can be written as the product of three terms.

$$\frac{\partial\,\mathrm{SE}(y,p)}{\partial z_1} \boxed{\begin{aligned} &= \frac{\partial SE(y,P)}{\partial p_1} \cdot \frac{\partial p_1}{\partial z_1} + \frac{\partial SE(y,P)}{\partial P_2} \cdot \frac{\partial P_2}{\partial z_1} \\ &= 2(p_1 - y_1) \cdot p_1(1-p_1) + 2[(1-p_1) - (1-y_1)] \cdot p_1(1-p_1) \\ &= 2p_1(1-p_1) \cdot \Big[p_1 - y_1 + (p_1 - y_1)\Big] \\ &= 4p_1(1-p_1) \cdot (p_1 - y_1) \end{aligned}}$$

Reference: * ChatGPT. (n.d.). Retrieved January 17, 2025, from https://chatgpt.com

Negative CE and SE applied to softmax have a lot of similarities, but they have some significant differences in their derivatives.

Let us visualize these derivatives.

Read and run the code below and interact with the widget. It shows how the KL, CE, and SE loss vary as a function of the logits. If you click on the CE checkbox, you can see how negative cross entropy is parallel to the KL loss curve.

Also notice how SE is very different from KL. In particular, notice how SE loss suffers from **vanishing graidents**: it saturates in regions where the predicted answer p is far from the true answer y. The flatness of the SE loss means that it does not really distinguish between the quality of bad answers, and it can be hard to use SE as a guide to improve a bad answer.

Also, notice that when the target probability y is imbalanced, e.g., $y = 0.1$, then SE is also noticably flatter than KL at the point of minimum loss, with a much flatter curvature. That means that SE is very accepting of not-very-good answers whereas KL does a better job at distinguishing very-good answers from slightly less-good answers.

Question 1.2.4. Plot and compare the partial derivatives of KL, and SE on softmax as well.

In the code below, the plot on the right is incomplete because it does not include the correct plot of partial dervatives for KL and SE losses. Copy your answers from 3.3 into the proper lines of the code below to visualize the derivatives as well.

Notice that CE has exactly the same shape as KL.

The plots you make explain why cross-entropy loss typically works much better than SE in practice. While both KL and SE are flat at the optimal point, unlike KL, SE flattens out again when the logits are far from the optimal point. We say that SE *saturates* and suffers from a *vanishing gradient* when the system is far from the optimum. Optimizations behave like a rolling stone: if you were to put a stone on the SE loss curve, it could easily get stuck in the high flat area of the curve. Whereas if you put a stone on the the KL loss curve, it would be on a steeper slope and roll quickly to the bottom.

[9]:
```
from baukit import PlotWidget, Range, Checkbox, show
import math

xmin, xmax = -6.0, 6.0
z = torch.stack(
    [
        torch.zeros(201),
        torch.linspace(xmin, xmax, 201),
    ]
)
p = torch.softmax(z, dim=0)


def compare_loss(fig, y1=0.5, dokl=True, dose=True, doce=True, dol1=True):
    [ax1] = fig.axes
    y0 = 1.0 - y1
    kl = y0 * (math.log(y0) - torch.log(p[0])) + y1 * (math.log(y1) - torch.
 ↪log(p[1]))
    ce = y0 * (-torch.log(p[0])) + y1 * (-torch.log(p[1]))
```

```
    se = ((p - torch.tensor([y0, y1])[:, None]) ** 2).sum(0)
    # sampled_se = (y0 * ((1-p[0])**2 + p[1]**2)) + (y1 * ((1-p[1])**2 +␣
 ↪p[0]**2))
    sampled_l1 = 2 * y0 * p[1] + 2 * y1 * p[0]
    ax1.clear()
    ax1.set_ylim(0, 3.0)
    ax1.set_xlim(xmin, xmax)
    ax1.set_ylabel("Loss")
    ax1.set_xlabel("Difference between logits $z_1 - z_0$")
    ax1.set_title(f"Loss curve on softmax when target $y_1={y1:.3f}$")

    if dokl:
        ax1.plot(z[1], kl, label="KL", color="b")
    if dose:
        ax1.plot(z[1], se, label="SE", color="r")
    if doce:
        ax1.plot(z[1], ce, label="CE", color="g", linestyle="dashed", alpha=0.6)
    if dol1:
        ax1.plot(
            z[1], sampled_l1, label="L1", color="orange", linestyle="dotted",␣
 ↪alpha=0.7
        )
    if dokl or dose or doce or dol1:
        ax1.legend()


def compare_grad(fig, y1=0.5, dokl=True, dose=True):
    [ax1] = fig.axes
    y0 = 1.0 - y1
    # fill me in so that d kl / d z1 is plotted.
    dkl_dz1 = p[1] - y1
    # fill me in so that d se / d z1 is plotted
    dse_dz1 = 2 * p[1] * (1 - p[1]) * (p[1] - y1)
    ax1.clear()
    ax1.set_ylim(-0.7, 0.7)
    ax1.set_xlim(xmin, xmax)
    ax1.set_xlabel("Difference between logits $z_1 - z_0$")
    ax1.set_title(f"Gradient of loss with repect to $z_1$ when $y_1={y1:.3f}$")

    if dokl:
        ax1.plot(
            z[1],
            dkl_dz1,
            color="b",
            label=r"$\frac{\partial \mathrm{KL}}{\partial z_1}$"
            + r"=$\frac{\partial \mathrm{CE}}{\partial z_1}$",
        )
```

```
    if dose:
        ax1.plot(
            z[1],
            dse_dz1,
            color="r",
            label=r"$\frac{\partial \mathrm{SE}}{\partial z_1}$",
        )
    ax1.axhline(0, color="gray", linewidth=0.5)
    if dokl or dose:
        ax1.legend(loc="upper left")


rw = Range(min=0.001, max=0.999, step=0.001, value=0.5)
bkl = Checkbox("KL", value=True)
bce = Checkbox("CE", value=True)
bse = Checkbox("SE", value=True)
bl1 = Checkbox("L1", value=False)
ploss = PlotWidget(
    compare_loss,
    y1=rw.prop("value"),
    dokl=bkl.prop("value"),
    dose=bse.prop("value"),
    doce=bce.prop("value"),
    dol1=bl1.prop("value"),
    bbox_inches="tight",
)
pgrad = PlotWidget(
    compare_grad,
    y1=rw.prop("value"),
    dokl=bkl.prop("value"),
    dose=bse.prop("value"),
    bbox_inches="tight",
)
show(
    [
        [
            show.raw_html("<div>target y<sub>1</sub> = </div>"),
            show.style(flex=12),
            rw,
            "Include:",
            bkl,
            bce,
            bl1,
            bse,
        ],
        [ploss, pgrad],
    ]
```

```
)
```

```
<baukit.show.HtmlRepr at 0x7b98000cf550>
```

1.8 Exercise 1.3: loading and using a pretrained SOTA model

Pytorch `Modules` make it easy to save, load, train, and use functions that are parameterized by lots of learned numbers. In this exercise you will load a big state-of-the art deep network model and use it. Do **not** worry that you do not understand how the specific model works or how it was trained. We will be covering the concepts in the course, and if you would like to spend the semester understanding a specific state-of-the-art system like this, you can choose it for your final project.

To see how this would work in the real world, we will use Hugging Face, which a platform being developed by an AI startup to host pretrained models.

First, sign up for a free huggingface.co account at https://huggingface.co/join if you don't already have one.

Second, log in. After you have signed up, you need to set up a login authentication key with your notebook by running the following notebook cell. This will allow your code to download some models through your account. When it prompts you to go to https://huggingface.co/settings/tokens it is enough to create a "Read" token for this homework.

[ ]:
```
from huggingface_hub import notebook_login

notebook_login()
```

Third, download the Stable Diffusion model pipeline by running the following cell. Stable Diffusion is very new, and these models were released on August 22, 2022. Read this blog entry about it: https://huggingface.co/blog/stable_diffusion

You do **not** need to understand the following citations. The code we are downloading is by Suraj Patil and others at Huggingface, and it implements text-conditioned diffusion modeling to a VAE latent space, as devised by Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser and Björn Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models" (Latent Diffusion, CVPR 2022, https://arxiv.org/abs/2112.10752) and trained by stability.ai. The method directly builds on work by Ho, et al (Denoising Diffusion, 2020), Radford, et al. CLIP 2021, Ho and Salimans (Classifier-free guidance, 2022), Ramesh, et al (Dall-E 2, 2022), Saharia, et al (Imagen, 2022), and Crowson (PLMS k-diffusion, 2022).

If the pipeline seems complex, be aware that the ideas did not all come from one person.

The following cell will take a few minutes to download the models. Running on a GPU machine is highly recommended.

[ ]:
```
import torch
from diffusers import StableDiffusionPipeline

device = "cuda" if torch.cuda.is_available() else "cpu"

# This function loads several neural networks to use together.
```

```
# The individual networks and preprocessors loaded are listed in this file:
# https://huggingface.co/CompVis/stable-diffusion-v1-4/blob/main/model_index.
 ↪json
pipe = StableDiffusionPipeline.from_pretrained(
    "CompVis/stable-diffusion-v1-4",
    revision="fp16",
    torch_dtype=torch.float16,
    use_auth_token=True,
).to(device)
```

Question 1.3.1.

The following cell lists the objects contained in the loaded pipeline; some of these objects are neural networks and some are not. Modify the cell below to figure out which of the four objects are neural networks (i.e., they extend `torch.nn.Module`) and then use `Module` methods to count how many submodules each one contains, how many Tensor parameters, and within those parameters, how many scalar parameters are within each network.

Write the code to figure it out and include it in the following cell.

Also, in answers into the following table. One row is already given.

Attribute	Type	Modules	Tensor params	Scalar params
`vae`	NeuralNetwork	220	248	83,653,863
`text_encoder`	NeuralNetwork	152	196	123,060,480
`unet`	NeuralNetwork	686	686	859,520,964
`safety_checker`	StableDiffusionSafetyChecker	299	396	303,981,588

[12]:
```
for name, obj in vars(pipe).items():
    print(f"pipe.{name}")
```

```
pipe._is_unet_config_sample_size_int
pipe._internal_dict
pipe.vae
pipe.text_encoder
pipe.tokenizer
pipe.unet
pipe.scheduler
pipe.safety_checker
pipe.feature_extractor
pipe.image_encoder
pipe.vae_scale_factor
pipe.image_processor
```

[13]:
```
# Iterate over the attributes of the pipeline
for attr in vars(pipe).items():
    attr_name = attr[0]
    attr = getattr(pipe, attr_name)
```

```
    if isinstance(attr, torch.nn.Module):
        print(f"{attr_name} is a neural network.")
        # Additional details
        print(f"  # of submodules: {len(list(attr.named_modules()))}")
        print(f"  # of parameters: {len(list(attr.named_parameters()))}")
        print(f"  # of scalar parameters: {sum(p.numel() for p in attr.
 ↪parameters())}")
```

```
vae is a neural network.
  # of submodules: 220
  # of parameters: 248
  # of scalar parameters: 83653863
text_encoder is a neural network.
  # of submodules: 152
  # of parameters: 196
  # of scalar parameters: 123060480
unet is a neural network.
  # of submodules: 686
  # of parameters: 686
  # of scalar parameters: 859520964
safety_checker is a neural network.
  # of submodules: 299
  # of parameters: 396
  # of scalar parameters: 303981588
```

Now run the code below. You will likely need a GPU-enabled machine to run it.

You **do not** need to understand every line, but if you are curious what is going on, this blog entry about the stable diffusion code is informative.

Try it with your own prompts. Can you come up with any interesting insights?

[22]:
```
from baukit import show, renormalize, pbar
from torch import autocast
import numpy

prompt = "Elon Musk shaking hands with Mark Zuckerberg"
seed = 1

# Stable Diffusion inference devised by Robin Rombach et al. (CVPR 2022, https:/
 ↪/arxiv.org/abs/2112.10752)
# Derived from the Huggingface Stable Diffusion pipeline by Suraj Patil and␣
 ↪others
# https://github.com/huggingface/diffusers/blob/main/src/diffusers/pipelines/
 ↪stable_diffusion/pipeline_stable_diffusion.py#L16-L171
with autocast(device), torch.no_grad():
    text_tokens = pipe.tokenizer(
        ["", prompt], padding="max_length", return_tensors="pt"
    )["input_ids"]
```

```
  text_vectors = pipe.text_encoder(text_tokens.to(device))[0]
  image_vectors = (
      torch.from_numpy(numpy.random.RandomState(seed).randn(1, 4, 64, 64))
      .float()
      .to(device)
  )

  # The scheduler uses a linear multistep (PLMS) method proposed by Katherine␣
↪Crowson
  # https://github.com/crowsonkb/k-diffusion
  scheduler = pipe.scheduler
  scheduler.set_timesteps(33)
  latent_scale = 0.18215
  guidance_strength = 5.0
  intermediates = []
  for i, t in enumerate(pbar(scheduler.timesteps)):
      if i % 6 == 0:
          intermediates.extend(
              renormalize.as_image(
                  pipe.vae.decode(image_vectors / latent_scale).sample
              )
          )
      # Pass two copies into the network, one to process with "" and the␣
↪other with prompt.
      image_vector_input = torch.cat([image_vectors] * 2)
      # pipe.unet is a neural network inputs image_vector_inputs and␣
↪text_vectors and outputs some updates
      update = pipe.unet(image_vector_input, t, text_vectors)["sample"]
      # Classifier-free guidance: see Jonathan Ho and Tim Salimans
      # (Neurips 2021 Workshop, https://arxiv.org/abs/2207.12598)
      strong_guidance = update[0] + guidance_strength * (update[1] -␣
↪update[0])
      image_vectors = scheduler.step(strong_guidance, t,␣
↪image_vectors)["prev_sample"]

  # pipe.vae is a neural network
  rgb_vectors = pipe.vae.decode(image_vectors / latent_scale).sample
  intermediates.extend(renormalize.as_image(rgb_vectors))
  show(show.WRAP, [[show.style(width=144), im] for im in intermediates])
```

```
<baukit.show.HtmlRepr at 0x7b9777cf2410>
```

Question 1.3.2.

Data flows through the pipeline in four tensors: `text_tokens`, `text_vectors`, `image_vectors`, and `rgb_vectors`.

Write some code below to check the `shape` and `dtype` for each of these tensors, and then determine the role of each of the tensor dimensions. Then fill in the following table.

Enter your answers into this table

Tensor	Dtype	Shape	numel	batch size	feature size, if any	spatial size, if any
`text_tokens`	int64	2x77	154	2	None	77 (seq_len)
`text_vectors`	float32	2x77x768	118272	2	768 (emb_dim)	77 (seq_len)
`image_vectors`	float32	1x4x64x64	16384	1	4 (RGB+Alpha)	64 (y) x 64 (x)
`rgb_vectors`	float16	1x3x512x512	786,432	1	3 (RGB)	512 (y) x 512 (x)

[29]:
```python
def analyze_tensors(tensors_dict):
    results = []
    for name, tensor in tensors_dict.items():
        # Get basic attributes
        shape = list(tensor.shape)
        dtype = tensor.dtype
        numel = tensor.numel()

        # Extract specific dimensions
        batch_size = shape[0]

        # Determine feature/spatial sizes
        feature_size = None
        spatial_size = None

        if name == "text_tokens":
            spatial_size = shape[1]  # sequence length
        elif name == "text_vectors":
            feature_size = shape[2]  # embedding dim
            spatial_size = shape[1]  # sequence length
        elif name == "image_vectors":
            feature_size = shape[1]  # latent channels
            spatial_size = f"{shape[2]}x{shape[3]}"  # height x width
        elif name == "rgb_vectors":
            feature_size = shape[1]  # RGB channels
            spatial_size = f"{shape[2]}x{shape[3]}"  # height x width

        results.append(
            {
                "name": name,
                "dtype": dtype,
                "shape": shape,
                "numel": numel,
                "batch_size": batch_size,
                "feature_size": feature_size,
```

```
                "spatial_size": spatial_size,
            }
        )
    return results
```

[30]:
```
tensors = {
    "text_tokens": text_tokens,
    "text_vectors": text_vectors,
    "image_vectors": image_vectors,
    "rgb_vectors": rgb_vectors,
}

results = analyze_tensors(tensors)
```

[31]:
```
# Print markdown table
print("| Tensor | Dtype | Shape | numel | batch size | feature size | spatial␣
 ↪size |")
print(
   ␣
 ↪"|--------|--------|--------|--------|------------|--------------|---------------|"
)
for r in results:
    print(
        f"| {r['name']} | {r['dtype']} | {r['shape']} | {r['numel']} |␣
 ↪{r['batch_size']} | {r['feature_size']} | {r['spatial_size']} |"
    )
```

```
| Tensor | Dtype | Shape | numel | batch size | feature size | spatial size |
|--------|--------|--------|--------|------------|--------------|---------------
|
| text_tokens | torch.int64 | [2, 77] | 154 | 2 | None | 77 |
| text_vectors | torch.float32 | [2, 77, 768] | 118272 | 2 | 768 | 77 |
| image_vectors | torch.float32 | [1, 4, 64, 64] | 16384 | 1 | 4 | 64x64 |
| rgb_vectors | torch.float16 | [1, 3, 512, 512] | 786432 | 1 | 3 | 512x512 |
```

[32]:
```
# Use this cell to write test code to check the size, type, and meaning of each␣
 ↪tensor dimension
pipe.tokenizer.decode(text_tokens[1, 3])
```

[32]:
```
'shaking'
```

1.9 Exercise 1.4: use a dataloader and run a NSFW filter

The Stable Diffusion pipeline comes with a NSFW filter neural network called `safety_checker`.

In this exercise, you will test out this network by passing 1000 images to it.

Like most pytorch neural networks, this network is configured to run on *batches* of data. You will pass the images to the network in batches of 10, using a DataLoader with `batch_size=10`.

The code below downloads a small classroom dataset called `coco_humans` of *individual* images.

[16]:
```
import os
from torchvision.datasets.utils import download_and_extract_archive

if not os.path.isdir("coco_humans"):
    download_and_extract_archive(
        "https://cs7150.baulab.info/2023-Fall/data/coco_humans.zip",␣
 ↪"coco_humans"
    )
```

```
Downloading https://cs7150.baulab.info/2023-Fall/data/coco_humans.zip to
coco_humans/coco_humans.zip
```

```
100%|      | 39.1M/39.1M [00:00<00:00, 50.5MB/s]
```

```
Extracting coco_humans/coco_humans.zip to coco_humans
```

Here is some information about the `safety_checker` neural network.

In pytorch, a neural network is a `torch.nn.Module`, and every `torch.nn.Module` is a *callable* object that can be called just like a function.

To see how to call `pipe.safetey_checker` you can consult the original Stable Diffusion code that calls it: https://github.com/huggingface/diffusers/blob/main/src/diffusers/pipelines/stable_diffusion/pipeline_st

```
image, has_nsfw_concept = self.safety_checker(images=image,
clip_input=safety_cheker_input.pixel_values)
```

Specifically, the `safety_checker` network is expecting two inputs when it is called.

The second `clip_input` argument is quite typical and conventional for a vision network. It is a read-only 4-dimensional pytorch tensor that should contain the number data for a batch of normalized RGB images that the network will examine for possible offensive images.

To pass a batch of 10 images as `clip_input`, you will need to normalize the data correctly - that is, you will need to paticular minimum and maximum numbers to represent the range from black pixels to white pixels, and you will want to use the same range that the network expects. Since this is a "CLIP" network, it expects CLIP standard normalization as defined in the `clip_transform` in the code below (source is cited in the code). Check the `torchvision.datasets.ImageFolder` documentation about how to use an image transform like this when loading data.

The first `safety_checker` argument, `images`, is pretty unusual for a neural network, and we can almost ignore it. It is a mutable list of tensors for the image data, which the network will use to alter the original images to black out any suspected NSFW regions. (Most neural networks don't do mutations that alter their input data.) Since we're not interested in blacking out anything for our test, we would prefer to ignore this argument, but it has to be supplied, so in the code below we create a `numpy_list` which you can provide as `images=numpy_list`, and then which you can then ignore.

Question 1.4.1

Complete the code below to find any images in the `coco_humans` data set that are flagged by `safety_checker`. * Pass numpy_list and a single 10x3x224x224 pytorch tensor, correctly normal-

ized, each time you call `pipe.safety_checker`. * You might need to make sure the tensor has a device data type that matches the network. * If any of the has_nsfw_concept flags come back `True`, then print the specific image number and display the image.

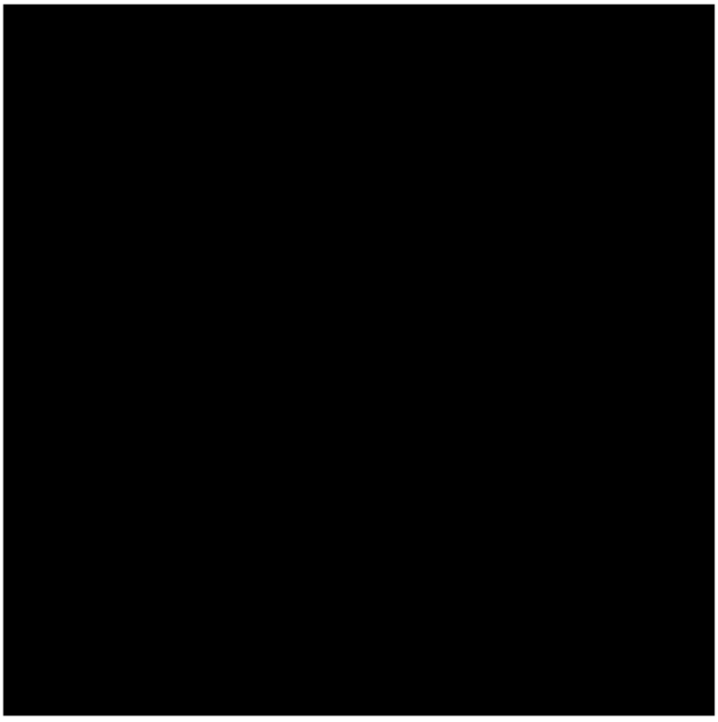
Some hints and sanity checks: It is not a classification dataset, so all the data items have the same class number according to `ImageFolder`. It should say that 1000 images are available in the data set; and if you examine image number 213 in the data set, you should see a slalom skiier. A small handful of images will be flagged, including image number 162. To display a flagged image, you could conver it back to a PIL image, or you could just create a second ImageFolder dataset, one for getting PIL image objects, and one for getting tensors.

[63]:
```
from PIL import Image


def display_image(numpy_image):
    pil_image = Image.fromarray((numpy_image.transpose(1, 2, 0) * 255).
 ↪astype("uint8"))
    plt.imshow(pil_image)
    plt.axis("off")
    plt.show()
```

[65]:
```
from torchvision.transforms import Compose, Normalize, ToTensor
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader

image_dataset = ImageFolder("coco_humans")
print(len(image_dataset), "images available")

# CLIP net standard normalization puts numerical image data in a range
# with zero mean and unit variance based on empirical data. Source:
# https://github.com/openai/CLIP/blob/c5478aac7b9/clip/clip.py#L85
clip_transform = Compose(
    [
        ToTensor(),
        Normalize(
            [0.48145466, 0.4578275, 0.40821073], [0.26862954, 0.26130258, 0.
 ↪27577711]
        ),
    ]
)


# Fix this, but leave the batch_size as 10.
image_dataset = ImageFolder("coco_humans", transform=clip_transform)

for batch_idx, (image_batch, class_numbers) in enumerate(
    DataLoader(image_dataset, batch_size=10)
):
```

```
    numpy_list = [im.numpy() for im in image_batch]
    # use pipe.safety_checker to flag any images
    # and display the image and the index number of each flagged image

    image_batch = image_batch.to(device)

    # Call the safety_checker
    images, has_nsfw_concept = pipe.safety_checker(
        images=numpy_list, clip_input=image_batch
    )

    # Check for flagged images
    for idx, flagged in enumerate(has_nsfw_concept):
        if flagged:
            print(f"Image {batch_idx * 10 + idx} flagged as NSFW.")
            display_image(numpy_list[idx])
```

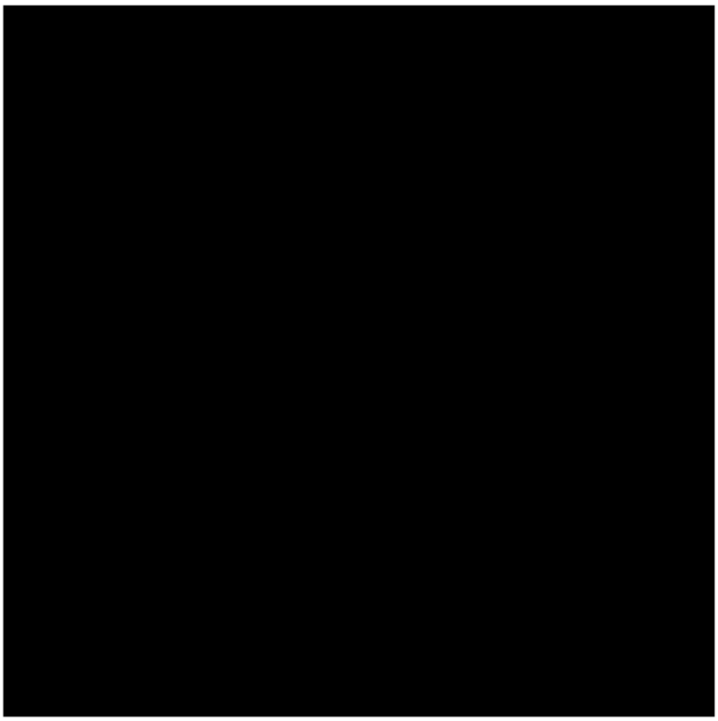
```
1000 images available

Potential NSFW content was detected in one or more images. A black image will be
returned instead. Try again with a different prompt and/or seed.

Image 65 flagged as NSFW.
```



```
Potential NSFW content was detected in one or more images. A black image will be
```

returned instead. Try again with a different prompt and/or seed.

Image 71 flagged as NSFW.



Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 162 flagged as NSFW.

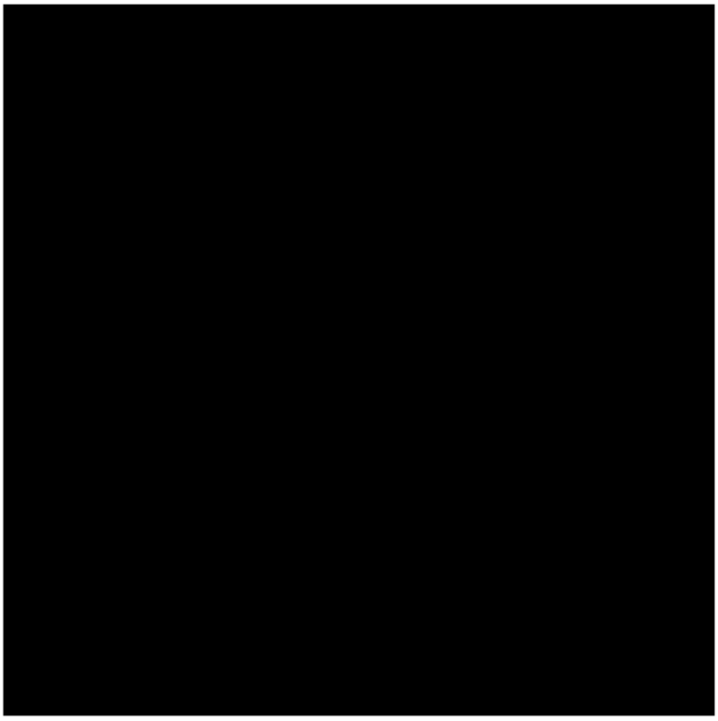
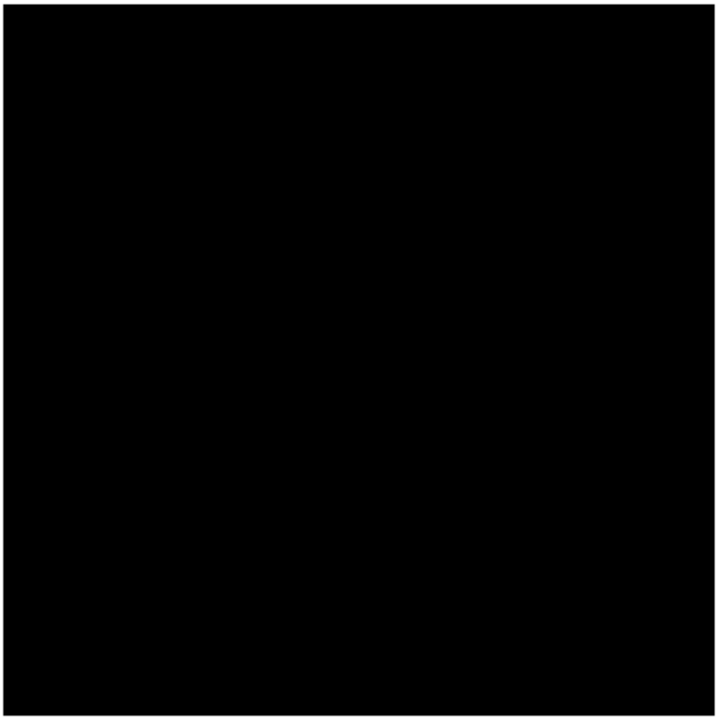
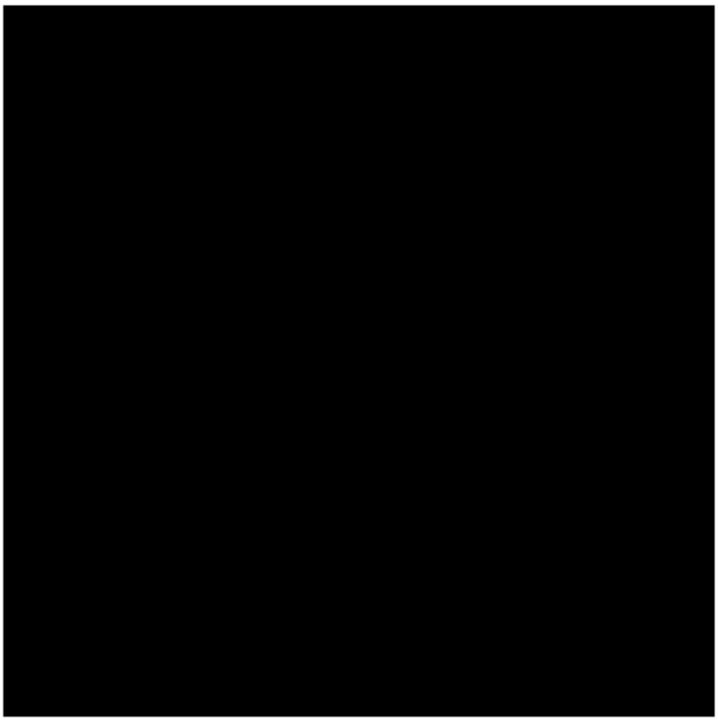


Image 166 flagged as NSFW.



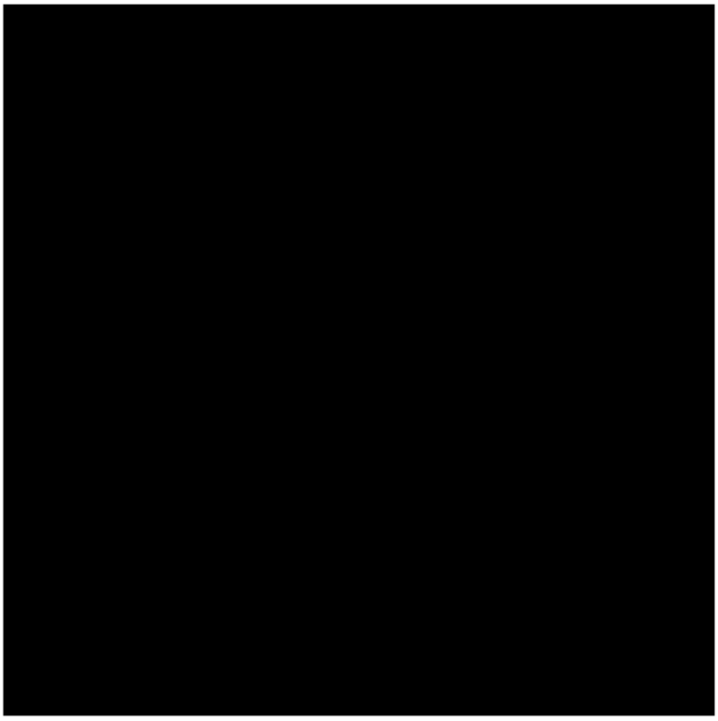
Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 392 flagged as NSFW.



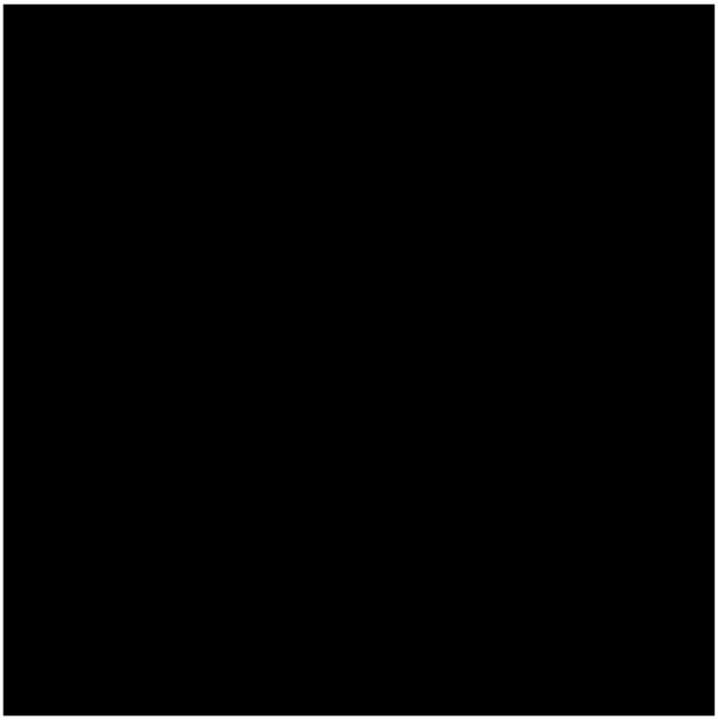
Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 432 flagged as NSFW.



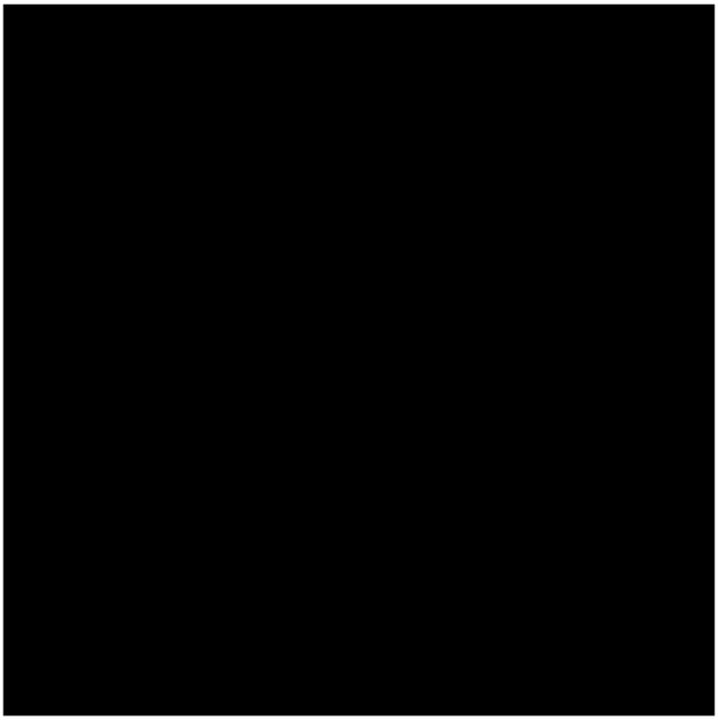
Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 440 flagged as NSFW.



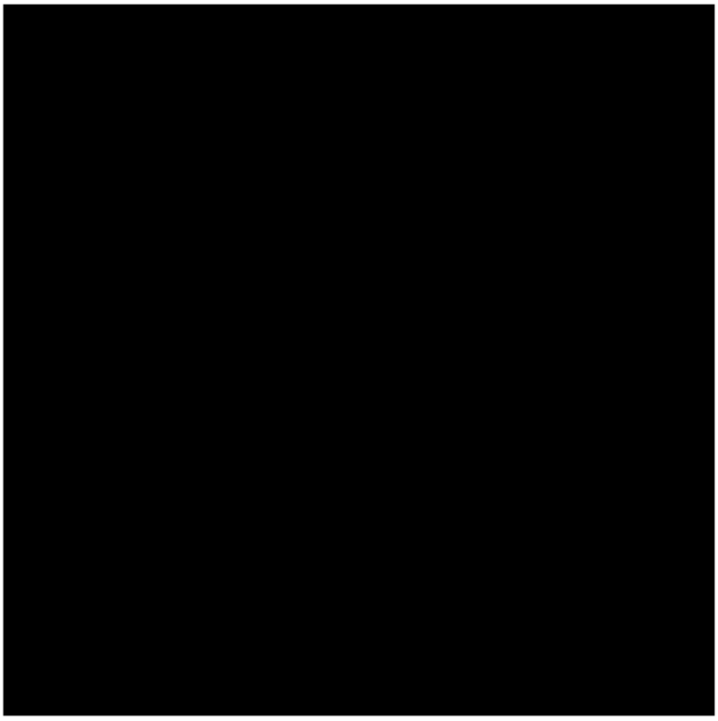
Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 572 flagged as NSFW.



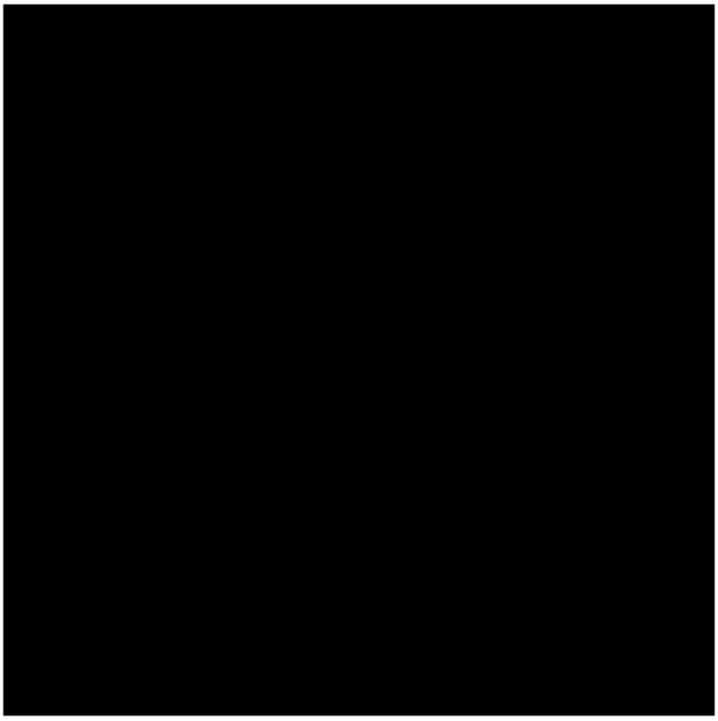
Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 626 flagged as NSFW.



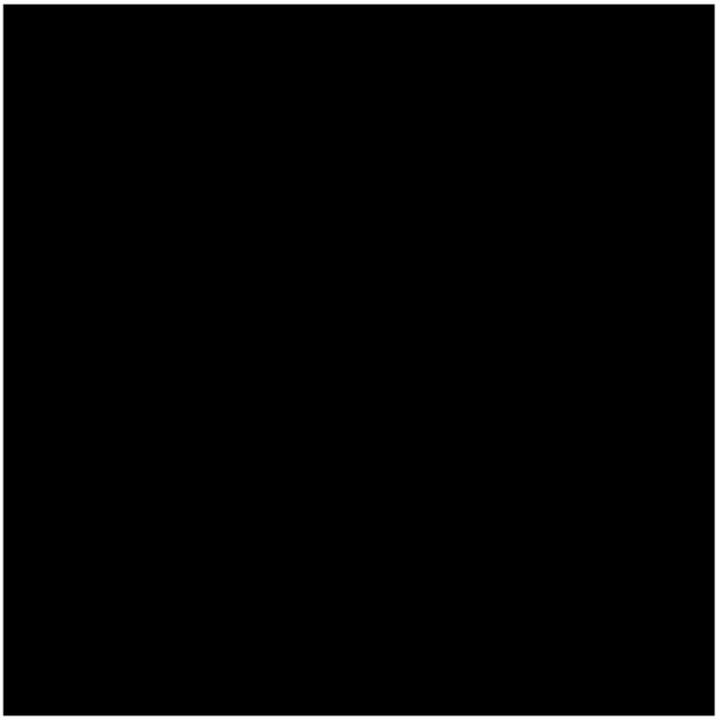
Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 928 flagged as NSFW.



Potential NSFW content was detected in one or more images. A black image will be returned instead. Try again with a different prompt and/or seed.

Image 964 flagged as NSFW.



Problem 1.4.2

Fill in the following answers:

In the `coco_humans` dataset, the Stable Diffusion safety checker found $\boxed{11}$ unsafe images and when looking at them actually $\boxed{0}$ were offensive. When automatic machine-leaned filters are used to omit data, they are typically used with the intention of reducing the chance of propagating offensive or illegal content. What other potential benefits or drawbacks do you see to using a neural network to filter content?

1.9.1 Answer

Benefits: * Reduces the budgets and exploitation on human annotators. * More senstive to imperceptible details.

Drawbacks: * Restricted to Precision/Recall Tradeoff

1.10 Backpropagation

For part 2 of the homework, proceed to the next notebook.