Alice in Wonderland ML Notes

1 Mathemacial Preliminaries

1.1 Linear Algebra

- A **tensor** X is an n-dimensional array of elements of the same type. $X \sim (s_1, s_2, \cdot, s_n)$ denotes the shape of the tensor.

1.1.1 Vector Operations

- A property of the dot product is that the maximum value of the dot product of two normalized vectors occurs when both vectors are the same.
  - When $\mathbf{x}$, which represents the input, and $\mathbf{w}$, which represents adaptable parameters, resonate, the dot product is maximized.
  - This is called template matching.

1.1.2 Matrix Operations

- Given two matrices $\mathbf{X}$ and $\mathbf{Y}$, matrix multiplication is defined element wise as: $\mathbf{Z}_{ij} = \mathbf{X}_i \cdot \mathbf{Y}_j$ i.e. the element (i, j) of the product is the dot product of the i-th row of $\mathbf{X}$ and the j-th column of $\mathbf{Y}$.
- The Hadamard method of multiplying matrices is element wise multiplication where each element of the resulting matrix $\mathbf{Z}$ is given by $\mathbf{Z}_{ij} = \mathbf{X}_{ij} \cdot \mathbf{Y}_{ij}$.
- The Hadamard multiplication method is used primarily to mask matrices i.e. setting some elements to zero or scaling operations.
- The Hadamard multiplication method does not preserve linearity and cannot be used in operations where linearity is required. Additionally, it cannot be used in compositions of functions such as $f(g(x))$ because it operates element-wise rather than on the entire structure of the matrices.
- There are many operations that can be done element wise or with whole matrices. PyTorch has built in modules for both types of operations.

1.1.3 Higher-order Tensor Operations

- When in higher dimensions, most of the operations we are interested in are either batched variants matrix operations, or specific combinations of matrix operations and reduction operations.
- Example: with two tensors $\mathbf{X} \sim (n, a, b)$ and $\mathbf{Y} \sim (n, b, c)$, the batched matrix multiplication is defined as $\mathbf{Z} \sim (n, a, c)$ where $\mathbf{Z}_i = \mathbf{X}_i \cdot \mathbf{Y}_i$.

1.2 Gradients & Jacobians

- Gradients play a pivotal role in optimization algorithms by providing semi-automatic mechanisms deriving from gradient descent.

1.2.1 Gradients and Directional Derivatives

- The gradient of a function is defined as:

$$\nabla f(\mathbf{x}) = \partial f(\mathbf{x}) = \begin{bmatrix} \partial_{x_1} f(\mathbf{x}) \\ \vdots \\ \partial_{x_d} f(\mathbf{x}) \end{bmatrix}$$

- The directional derivative is the dot product of the gradient and the direction vector:

$$\nabla f(x) \cdot \mathbf{v}$$

1.2.2 Jacobians

- Let there be a function $f(x)$ that maps a vector input $\mathbf{x} \sim (d)$ to a vector output $\mathbf{y} \sim (c)$. To calculate the gradient for each output, we must create the **Jacobian** of f.

$$\partial f(\mathbf{x}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_d} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_c}{\partial x_1} & \frac{\partial y_c}{\partial x_2} & \dots & \frac{\partial y_c}{\partial x_d} \end{bmatrix}$$

- Each column of the Jacobian corresponds to the gradient of $f(x)$ that maximizes a specific value within the output vector $\mathbf{y}$.
- Each row of the Jacobian describes how the rate of change for the outputs changes with respect to a specific input.
- When c is equal to 1, i.e. when there is only a single output parameter, the matrix simplifies to a single row vector which is the gradient of the function $f(x)$.
- When $c = 1 = d$, the Jacobian becomes the standard derivative of the function.
- Jacobians inherit the properties of derivatives, including the fact that the Jacobian of a compositions of functions is now the matrix multiplication of the individual Jacobians.
- For a point x_0, the best linear approximation to $f(x)$ is $f(\mathbf{x}_0) + \partial f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0)$. This is called Taylor's theorem.
- A code example:

```
$ Generic mathematical function
f = lambda x: x**2 - 1.5*x

# Derivative
df = lambda x: 2*x - 1.5

x = 0.5
f_linearized = lambda h: f(x) + df(x)*(h-x)

#Comparing approximation to actual function
print(f(x + 0.01)) # [Out] = -0.5049
print(f_linearized(x + 0.01)) # [Out] = -0.5050
```

1.3 Numerical Optimization and Gradient Descent

- Consider the problem of trying to find the minimum of a function $f(x)$. Assuming the function has a single output **single-objective optimization**, we try to find a global minimum within an uncontrained domaine.
- It it possible to express the solution in closed-form (where there is a function to find the optimal $\mathbf{x}$), but in general we must resort to iterative procedures.
- Let's start with a random guess $\mathbf{x_0}$ and for every iteration, we decompose the new position as the sum of the old position + the magnitude of the step times the direction of the step:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \cdot \mathbf{p}_t$$

  where η_t is the length of the step and $\mathbf{p}_t$ is the normalized direction vector.
- We call η_t the **learning rate** and a direction $\mathbf{p}_t$ such that $f(\mathbf{x}_t) \leq f(\mathbf{x}_{t-1})$ the **descent direction**.

- Selecting a descent direction for every iteration and being careful with choice of step size will allow us to converge to a local minimum.
- Given that $\mathbf{p}_t$ is the descent direction, it is known that $D_{\mathbf{p_t}} f(\mathbf{x_{t-1}}) \leq 0$.
- Given that the directional derivative is the dot product of the gradient and the direction vector, we can conclude:
$$D_{\mathbf{p_t}} f(\mathbf{x_{t-1}}) = \nabla f(x_{t-1}) \cdot \mathbf{p}_t = ||\nabla f(x_{t-1})|| \cdot ||\mathbf{p}_t|| \cdot \cos \alpha$$
where α is the angle between the gradient and the descent direction.
- The first term is a constant with respect to $\mathbf{p_t}$, and $||\mathbf{p_t}||$ can be assumed to be equal to 1 as it's a normalized direction vector. With this information, we can simplify the previous formula:
$$D_{\mathbf{p_t}} f(\mathbf{x_{t-1}}) = ||\nabla f(x_{t-1})|| \cdot \cos \alpha$$
- The properties of consine result in it being negative when $\frac{\pi}{2} < \alpha < \frac{3\pi}{2}$, therefore any $\mathbf{p_t}$ that forms an angle a satisfying the previous inequality will be a descent direction.
- The **steepest descent direction** is the direction where $\mathbf{p_t}$ forms an angle of π with $\nabla f(\mathbf{x_{t-1}})$ which is synonymous with $\mathbf{p_t} = -\nabla f(\mathbf{x_{t-1}})$.
- On an intutive level, this makes sense as the gradient points in the direction of greatest increase, so the negative of the gradient would point in the direction of greatest descrease.
- The previous formula can be rewritten as:
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})$$
- The step size doesn't matter all that much as long as the size is small enough for f to reduce with each iteration.

1.3.1 Convergence of Gradient Descent

- The formal definition for a local minimum of $f(x)$ is a point $\mathbf{x}^+$ such that the following is true for some $\epsilon > 0$:
$$f(\mathbf{x}^+) \leq f(\mathbf{x}) \; \forall \mathbf{x} : ||\mathbf{x} - \mathbf{x}^+|| < \epsilon$$
- In other words, the function $f(\mathbf{x})$ exists at a local minimum at a point $\mathbf{x}^+$ if for some positive value ϵ, $f(\mathbf{x}^+)$ is less than every point ϵ distance away from $\mathbf{x}^+$.
- By the definition of the local minimum, a function at some local minimum will only ever increase if it enters the neighborhood around the local minimum. Thus the gradient at a local minimum is zero and the gradient around the local minimum is pointing upwards.
- A **stationary point** of $f(\mathbf{x})$ is a point $\mathbf{x}^+$ such that $\nabla f(\mathbf{x}^+) = 0$.
- Stationary points exist at all minima, maxima, and saddle points i.e. where $\nabla f(\mathbf{x}) = 0$.
- Due to this, we can only guarantee that gradient descent will converge to a stationary point, not necessarily a local minimum.
- Ideally, we would want to attain the **global minimum** of a function, the one (or possibly one of many) point(s) in the domain where $f(\mathbf{x})$ attains its lowest possible value.
- For the sake of visualization, assume $f(\mathbf{x}) \in \mathbb{R}^3$. If the function assumes a parabolic shape, then every point in the domain will have a gradient pointing toward the global minimum.

- With the previous example, the topic of **convexity** comes up. A function $f(\mathbf{x})$ is convex if for any two points $\mathbf{x}_1$ and $\mathbf{x}_2$, and $\alpha \in [0, 1]$, we have:

$$f(\underbrace{\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2}_{\text{Interval from } \mathbf{x}_1 \text{ to } \mathbf{x}_2}) \leq \underbrace{\alpha f(\mathbf{x}_1) + (1-\alpha)f(\mathbf{x}_2)}_{\text{Line segment from } f(\mathbf{x_1}) \text{ to } f(\mathbf{x}_2)}$$

- In words, a function is convex if the line segment connecting two points $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$ is always greater than or equal to every single value on the function between $\mathbf{x}_1$ and $\mathbf{x}_2$.
- A convex function simplified our task greatly for the following reasons:
  - For a generic non-convex function, gradient descent will always converge onto a stationary point, not necessarily a local minimum.
  - For a convex function, the stationary point is the global minimum.
  - if the inequality earlier is satisfied in a strict way (**strict convexity**), then the global minimizer is guaranteed to be unique.
- Trying to find the global minimum is a non-convex problem with gradient descent is impossible because you must run the algorithm for an infinite amount of time to check the infinite amount of points from an infinite amount of initializations in the unconstrained domain.

1.3.2 Accelerating Gradient Descent

- A problem with the gradient approach is that it only points to the greatest descent direction in an extemely small neighborhood around the current point. This can lead to very noisy updates and slow convergence.
- To smooth out the erratic changes in descent direction, we can make the direction of the current step to affect the direction of the next step. Such a method is called **momentum**:

$$\mathbf{g}_t = -\underbrace{\eta_t \nabla f(\mathbf{x}_{t-1})}_{\text{gradient descent}} + \underbrace{\lambda \mathbf{g}_{t-1}}_{\text{momentum}}$$
$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$$

  where we initialize $\mathbf{g}_0 = 0$ and λ is a parameter that determines how much the previous term is dampened.
- Expanding two terms:

$$\begin{aligned}\mathbf{g}_t &= -\eta \nabla f(\mathbf{x}_{t-1}) + \lambda(-\eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda \mathbf{g}_{t-2}) \\ &= -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}\end{aligned}$$

- The momentum method has been shown to accelerate training by smoothing the optimization path. Also, modifying the step size depending on the gradient is another method. Usually, the step size and the gradient are inversely proportional.