Alice in Wonderland ML Notes

Arjun Dulam

Github Repository Link

Contents

2 Mathemacial Preliminaries

2.1 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.

2.1.1 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.

2.1.2 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.

2.1.3 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 $\mathbf{Z}_i = \mathbf{X}_i \cdot \mathbf{Y}_i$.

2.2 Gradients & Jacobians

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

2.2.1 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:

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

2.2.2 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
```

2.3 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.

2.3.1 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.

2.3.2 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.

3 Datasets and Losses

3.1 What is a Dataset?

- A supervised dataset S_n of size n is a set of n pairs

$$S_n = \{(x_i, y_i)\}_{i=1}^n$$

  where each (x_i, y_i) is an example of an input-output relationship we want to model. We further assume that each example is an identically and independently distributed draw from some unknown (and unknowable) probability distribution $p(x, y)$.

- A sample being **identicially distributed** means that we are trying to track something that is sufficiently stable in terms of change over time. Take the task of identifying car models from photos. Since car models change over time, we will not be able to to have an identical distribution of car models in a dataset with large discrepancies on the time when the data was collected
- A sample being **independently distributed** means that there is no inherent bias in our training data. This condition would not be satisfied if we exclusively collected data from outside a Tesla dealership.

3.1.1 Variants of Supervised Learning

- In datasets with not enough targets y_i, we can use **unsupervised learning**. Typical applications of unsupervised learning are **clustering algorithms**, where points between clusters are similar and points within clusters are dissimilar. An example of this would be grouping together simlar news articles in terms of topics. Another example is a **retrieval** system, where we retrieve the most similar elements to the user's query.
- Unsupervised learning in itself is not ideal for image classification, because the slightest modifiction to an image can lead to millions of pixels being changed. A model that's already optimized for image classification, a **pre-trained** model, can be used to extract features from the images.
- The states of this model can be interpreted as vectors in a higher-dimensional space. These vectors can be mapped and used to train a classifier.

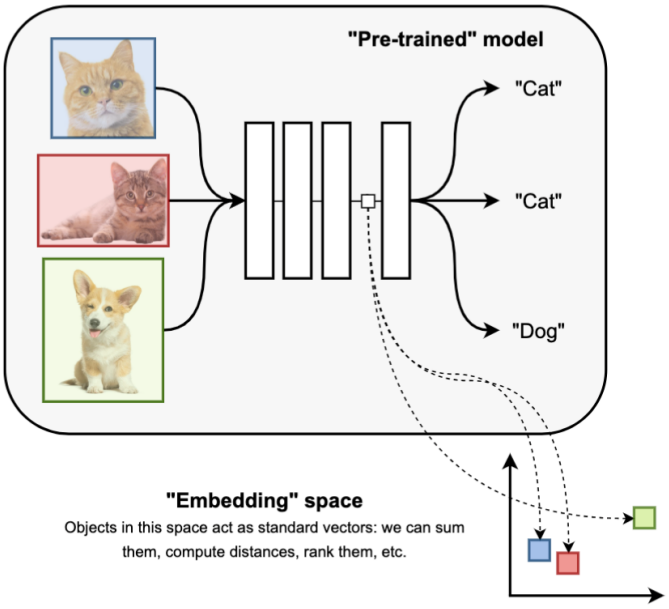


Figure 1: High level overview of using pre-trained model

- **Self-supervised learning** is a variant of un-supervised learning where the model is trained to find some supervised objective from a unsupervised dataset. An example of would be this: A model is given a large piece of text. The model removes a specific part from each sentence in the text and tries to guess the removed part. Comparing its guess to the actual removed part is the supervised objective, the model continuously learns from comparing its guess to the removed part.
- There are three ways of using trained models:
  - **Zero-Shot Learning**: An trained model is given a task it has not training on. It is given no extra data on the task, and must rely on its previous training.

  - **Few-Shot Prompting**: A trained model is given a task it has not training on. It is given a few examples of the task and uses its existing knowledge to make a new inference.
  - **Fine-Tuning**: A trained model is furthet trained on a specific task. This allows it to adapt to the specific needs of the task while using its prior existing knowledge.

- When fine tuning, the model can have all of its parameters changes, or change/add a few parametets. The latter is called **parameter efficient fine-tuning**.
- **Semi-supervised learning** is a variant of supervised learning where the model is trained on a dataset with a small number of labeled examples and a large number of unlabeled examples.

3.2 Loss Functions

- Given a desried targey y and the predicted value $\hat{y} = f(x)$ from a model f, a **loss function** $l(y, \hat{y}) \in \mathbb{R}$ is a scalar, differentiable function whose value correlates with the performance of the model. The performance of the model is measured by the minimization of the loss function i.e. $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ implies that $\hat{y}_1$ is a better prediction than $\hat{y}_2$ wrt the target y.
- The loss function's scalar and differentiable properties allow it to be minimized with a gradient descent algorithm.
- Given a dataset $S_n = \{(x_i, y_i)\}$, and a loss function $l(.,.)$, the optimization task at hand is to minimum average loss on the dataset by any possible differentiable model f:

$$f^* = \arg_f \min \underbrace{\frac{1}{n} \sum_{i=1}^{n}}_{\text{average}} \underbrace{l(y_i, f(x_i))}_{\text{loss value}}$$

- Essentially, we're trying to find the best model that minimizes the loss function. We do this by getting the average of the losses for every single prediction a model makes on a certain dataset. We compare this loss with the average loss of other models on the same dataset, the model with the lowest average loss is best at making predictions for inputs in the dataset.
- This is called **empirircal risk minimization** (risk is generic synonym for loss).
- Models can be parameterized by a set of tensors w (called parameters of the model), and minimization is done by searching for the optimal value of these parameters via numerical optimization, denoted by $f(x, w)$.
- $f(x, w)$ represents the prediction when giving an input x into a model with parameters w.
- Hence, the optimization task can be rewritten as:

$$w^* = \arg_w \min \frac{1}{n} \sum_{i=1}^{n} l(y_1, f(x_1, w))$$

- In this context, we determine the optimal parameters for a specific model that minimizes the average loss on the dataset.

On the differentiability of the loss function

- Consider a model f that outputs a $y \in \{-1, +1\}$ where the true target can only take on two values : -1 and +1.
- We can equate the two possible correct outputs with the sign of f, denoted sign(f(x)).

- One possible loss function is the **0/1 loss**:

$$l(y, \hat{y}) = \begin{cases} 0 \text{ if } \text{sign}(\hat{y}) = y \\ 1 \text{ otherwise} \end{cases}$$

  This is not differentiable, so the gradient descent algorithm would not work to minimize it.

- Another one is **margin** $y\hat{y}$, which will be positive if the prediction is correct and negative otherwise. This is preferable as is is continuously differentiable.

- The **hinge loss** function $l(x, y) = max(0, 1 - y\hat{y})$ is a continuous and differentiable loss function used to train support-vector models.

3.2.1 Expected Risk and Overfitting

- The loss function can be completely minimized if we only respond to inputs already within a dataset, however the objective is to minimize the loss function for all possible inputs.

- The **expected risk** given a probability distribution $p(x, y)$ and a loss function l is defined as:

$$\text{ER}[f] = \mathbb{E}_{p(x,y)}[l(y, f(x))]$$

- This expression shows the expected risk of a model f, denoted $\text{ER}[f]$, for all possible input-output pairs (x, y) on a probability distribution.

- The equation in unfeasiable to calculate, so the **empirical** risk is an estimate of the expected risk with a given dataset.

- The difference in loss between the expected and empirircal risk is called **generalization gap**.

- A overly specific model based on memorization will have a large generalization gap, as it will overfit to the training data, but does not respond well to new data.

- Generalization can be tested by using a separate **test dataset** that the model has not seen before.

3.2.2 Selecting Valid Loss Functions

- Assuming our examples come from a distribution $p(x, y)$, we can decompose it as $p(x, y) = p(x) \cdot p(y|x)$.

- The function $f(x)$ is used to predict $p(y|x)$, that is, the chance that the model will give the correct output y given the input x.

- Approximating $p(y|x)$ with a function $f(x)$ is viable if we assume that the probability mass is mostly centered around a single point y i.e. there are not multiple points $y_1, y_2, \cdots y_n$ that are likely to be be the output.

- However, if we move away from the previous definition of $f(x)$ and instead think of $f(x)$ as a parameterization of the chances of the different outputs $y_1, y_2, \cdots, y_n$ given x, we can represent $f(x)$ as:

$$f(x) = [p(y_1|x), p(y_2|x), p(y_3|x)]$$

- Similarly, we also define $\mathbf{y} \sim \text{Binary}(n)$ where $\mathbf{y}$ is a one-hot encoded vector that contains a 1 at the correct output's place.

- Thus, we can write:

$$p(\mathbf{y}|f(x) = \prod_{i=1}^{3} f_i(x)^{y_1}$$

- This chain of logic can be shown via an example: Assume there is a model, given an input x, outputs a $y \in \{1, 2, 3\}$. The model's chances of giving these outputs are, respectively, $f(x) = [0.2, 0.5, 0.3]$. Assume that the correct output is $y = 2$. Thus, the correct representation of $\mathbf{y} \sim 3$ is $[0, 1, 0]$. Following the expression above, $p(\mathbf{y}|f(x))$ can be calculated as:

$$\begin{aligned} p(\mathbf{y}|f(x)) &= f_1(x)^{y_1} \cdot f_2(x)^{y_2} \cdot f_3(x)^{y_3} \\ &= 0.2^0 \cdot 0.5^1 \cdot 0.3^0 \\ &= 0.5 \end{aligned}$$

  Thus, the probability of the model giving the correct output is 0.5.

- Our formula does not directly give the probability for the output, but instead gives a probability distribution over the possible outputs, with the correct one being one of them.

3.2.3 Maximum Likelihood

- Assume that the samples are independent and identically distributed (i.i.d) from a probability distribution $p(x, y)$. Recall that the probability distribution $p(x, y)$ describes the probability of observing the input x and y together. Hence, the probability assigned to the dataset itself by a specific choice f of function is given by the product of each sample in the dataset:

$$p(\mathcal{S}_n|f) = \prod_{i=1}^{\pi} p(y_i|f(x_i))$$

- This formula defines the probability that a given model f will provide outputs $y_1, y_2, \cdots, y_n$ that are the same as the outputs in the dataset $\mathcal{S}_n$ given the inputs $x_1, x_2, \cdots, x_n$.

- The quantity $p(\mathcal{S}_n|f)$ is called the **likelihood** of the set. This quantity is to be maximized up to a certain extent, where the model's outputs do not deviate greatly from the dataset's outputs but also where the model is also not overfitted with the dataset (and thus struggling with inputs outside of the dataset).

- Given a dataset $\mathcal{S}_n = \{\{x_i, y_i\}\}$ and a family of probability distributions $p(y|f(x)$ parameterized by $f(x)$, the maximum likelihood solution is given by:

$$f^* = \arg_f \max \prod_{i=1}^{\pi} p(y_i|f(x_i))$$

  Essentially, this formula provides the maximum likelihood across all models by providing the likelihood of the best function f^*.

- Switching to a minimization problem, we get:

$$\arg_f \max \left\{ \log \prod_{i=1}^{n} p(y_i|f(x_i)) \right\} = \arg_f \min \left\{ \sum_{i=1}^{n} -\log(p(y_i|f(x_i)) \right\}$$

  A few notes on this equation:

  - We use the log function because multiplying probabilities < 1 can result in a very small product, leading to underflow when storing into memory. Using log helps alleviate this issue.
  - Reminder: the log of a product is equivalent to the sum of the logs of items being multiplied:

$$\log(ab) = \log(a) + \log(b)$$

    Converting this to an addition problem results in reduced computing demand.
  - Adding the negative sign on the right-hand side of the equations turns this into a minimization problem which has been covered earlier.

3.3 Bayesian Learning

- When designing a probability function $p(y|f(X))$ instead of $f(x)$, we can handle situations where the model might give different possible outputs.
- This procedure however only looks an one singular function with a specific parameterization. What if we had multiple functions with different parameterization that all provide good outputs? It would be wasteful to rely on one singular function instead of relying on multiple and choosing the best model for an input based on its output.
- We can achieve this by defining a **prior probability distribution** $p(f)$ over all possible functions. Recall that f is a model with a certain set of parameters.
  - Recall that a **Bayesian prior** is a initial belief on what a parameter might be.
- Functions with smaller norms are preferred so setting up an inverse relationship with the parameters' norm would be useful in creating our priors:

$$p(f) \propto \frac{1}{||f||}$$

- Once a dataset is observed, the probability over f shifts depending on the prior and the likelihood, and the update is given by **Bayes' theorem**.

$$\underbrace{p(f|\mathcal{S}_n)}_{posterior} = \frac{p(\mathcal{S}_n|f)) \cdot \overbrace{p(f)}^{prior}}{p(\mathcal{S}_n)}$$

  The term $p(f|\mathcal{S})$ is called the **posterior distribution function**, while the term $p(\mathcal{S}_n)$ is called the **evidence** and normalizes the right-hand side of the equation.
- Given an input x, we can make a prediction by averaging all possible models based on their posterior's weight:

$$p(y|x) = \int_f p(y|f(x)) \cdot p(f|\mathcal{S}_n) \approx \frac{1}{k} \sum_{i=1}^{k} p(y|f_i(x)) \cdot p(f_i|\mathcal{S}_n)$$

  Here, we take the average of all models when we take the summation and divide by the amount of models on the right hand side.
- If we are solely interested in maximizing the posterior term, we can discard the evidence term as that remains relatively constant. As a result, we can choose only to focus on the terms in the numerator:

$$f^* = \arg\max p(\mathcal{S}_n|f)p(f) = \arg_f \max \left\{ \underbrace{\log p(\mathcal{S}_n|f)}_{likelihood} + \underbrace{\log p(f)}_{regularization} \right\}$$

  This is the **maximum a posteriori** (MAP) solution.
- If all functions have the same weight a priori, then the proble is reduced down to a maximum likelihood solution. The regularizer term pushes the solution toward the basin of attraction defined by the prior distribution.

4 Linear Models

4.1 Least-Squares Regression

4.1.1 Problem setup

Recall that a supervised learning problem can be defined by choosing the input type x, the output type y, the model f, and the loss function l.

- The input is a vector $\mathbf{x} \sim (c)$, corresponding to c number of features in the input.
- The output is a single real value $\in \mathbb{R}$.
  - If y can take any real value, this is a **regression** task/
  - If y can only take one out of m possible values, this is a **classification** task.
  - If y can only take one of two possible values, this is a **binary classification** task.
- We assume f is a linear model.
- We assume that:
  - $n \rightarrow$ size of the database
  - $c \rightarrow$ features of input
  - $m \rightarrow$ classes of outputs

4.1.2 Regression Losses: The Squared Loss and Variants

- Finding the loss for regression is very simple since the prediction error $e = \{\hat{y} - y\}$. However, since we do not concern ourselves with the sign of the loss, rather its absolute value, we can change the loss function to be:
$$l(\hat{y}, y) = (\hat{y} - y)^2$$
- The squared loss function provides many benefits, one of which is that it's rather easy to find the gradient of such a linear function of the model's input.
- Recalling the maximum likelihood principle, the squared loss can be obtained by assuming the outputs of the model follow a Gaussian distribution centered in $f(x)$ with a constance variance of σ^2.
$$p(y|f(\mathbf{x}) = \mathcal{N}(y|f(\mathbf{x})), \sigma^2)$$
Using properties of logs, we can rewrite the log-likelihood as:
$$\log(p(y|f(\mathbf{x}, \sigma^2)) = -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma^2}(y - f(\mathbf{x}))^2$$
- When we minimize for f, the first two terms in the RHS of the equation are constant, leaving the third to be the squared loss. Minimizing for σ^2 is an independent operation and will be touched on later.
- A disadvantage to using the squared loss is that mislabelled points in the source dataset can have an undue effect on the model. Higher errors will be be punished with quadratically growing strenght, which lets **outliers** to negatively impact the model.
- Some loss functions that diminish the influence of outliers are the absolute value loss:
$$l(\hat{y}, y) = |\hat{y} - y|$$
and the **Huber loss**, a combination of the squared loss and the absolute loss:
$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 \text{ if } |y - \hat{y}| \leq 1 \\ (|y - \hat{y}| - \frac{1}{2}) \text{ otherwise} \end{cases}$$
which is quadratic in the proximity of 0 error, and linear otherwise. The $-\frac{1}{2}$ is added for continuity, plot the equations in a 3D grapher to see why.

- Although the absolute value loss function might seem invalid due to its point of non-differentiability, a slight generalization of the derivative called the **subgradient**, can handle this point. On a practical level, gradient descent will never reach the point with perfect precision, so we can assume that derivatives of $|\epsilon|$ for any $\epsilon > 0$ is always defined.

4.1.3 The Least-Squares Model

- A linea model on an input $\mathbf{x}$ is defined as:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

  where $\mathbf{w} \sim (c)$ and $b \in \mathbb{R}$ (the bias) are trainable parameters.

- The intuition is that given an input $\mathbf{x}$ and parameters $\mathbf{w}$ both holding the same number of features and parameters, each feature of the input will be multiplicated by its respective parameter to produce a number $\in \mathbb{R}$. This number will then be added to the bias b.

- The bias term can be avoided when assuming that a 1 exists as the last feature of $\mathbf{x}$:

$$f\left(\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}\right) = \mathbf{w}^\top \begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \mathbf{w}_{1:c}^\top \mathbf{x} + w_{c+1}$$

- Comibining the linear model, the squared loss, and the empirical risk minimization problem, the **least-squares regression problem** is:

$$\mathbf{w}^*, b^* = \arg_{w,b} \min \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i - b)^2$$

- In code, the linear model can be described as:

```
def linear_model (w: Float[Tensor, "c"],
        b: Float
        x: Float[Tensor, "n␣c"])
        -> Float[Tensor, "n"]:
    return X @ w + B
```

- If we rewrite the least-squares in **vectorized** form, we can achieve optimal computing efficiency:
  The input in vectorized form:

$$\mathbf{X} = \begin{bmatrix}\mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top\end{bmatrix} \sim (n, c)$$

  The output in vectorized form:

$$y = \begin{bmatrix}y_1, \cdots, y_n\end{bmatrix}^\top$$

  The output model for a batch of values is:

$$f(\mathbf{X}) = \mathbf{X}\mathbf{w} + 1b$$

- A note: the ordering of the rows of the input and output do not matter the as the changes will be reflected in $f(\mathbf{X})$. This concept is called **permutation variance** and will be touched on later.

- The vectorized least-squares problem becomes:

$$LS(\mathbf{w}, b) = \frac{1}{n} ||\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}b||^2$$

4.1.4 Solving the Least-Squares Problem

- Applying differentiation to the least-squares equation, we get the gradient:

$$\nabla LS(\mathbf{w}) = \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$$

  The global minimum can then be found by setting the gradient to 0:

$$\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \Rightarrow \mathbf{X}^\top\mathbf{X}\mathbf{w} = \mathbf{X}^\top\mathbf{y}$$

  Solving for the optimal parameters $\mathbf{w}$ leads us to this:

$$\mathbf{w}_* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

- The matrix $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ is called the **pseudoinverse** of the non-square matrix $\mathbf{X}$. This inversion is not always possible: for example, if one features is a scalar multiple of another.

- When implementing the numerical solution to this problem, we must know when to stop. We can do this by evalutating the difference between two iterations for some numerical threshold $\epsilon > 0$.

$$||\mathbf{w}_{t+1} - \mathbf{w}_t||^2 < \epsilon$$

- An implementation of solving for the solution using gradient descent:

```
def least_squares_gd(X: Float[Tensor, "n␣c"],
                     y: Float[Tensor, "n"],
                     learning_rate=1e-3) \
                     -> Float[Tensor, "c"]:
    # Initializing the parameters
    w = torch.randn((X.shape[1], 1))

    # Fixed number of iterations
    for i in range(15000):
        # Note the sign: the derivative has a minus!
        w = w + learning_rate * X.T @ (y - X @ w)
    return w
```

- Once training has finished, σ^2 can also be optimized:

$$\sigma_*^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}_*^\top\mathbf{x}_i)^2$$

4.1.5 Some Computational Considerations

- A matrix $\mathbf{X}^\top X$ is well conditioned if the solution of the system remains stable and accurate regardless of small changes to the input. if we have an unstable system, the quality of the solution may degrade by a large amount and large numerical errors may arise.

- Computing the vector-product $\mathbf{X}\mathbf{w}$ can help avoid quadratic time complexity in the equation for the gradient.

- This can lead to linear time complexity in both c and n.

- Such optimization measures are necessary for **reverse-mode automatic differentiation** a.k.a **back-propagation**.

4.1.6 Regularizing the Least-Squares Solution

- When the matrix $\mathbf{X}$ is singular, we can modify the problem to achieve a solution which is "as close as possible" to the original one.
- Adding a small multiple $\lambda > 0$ of the identity matrix to the matrix being inverted:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

  Going back to the modified optimization problem:

$$LS - Reg(\mathbf{w}) = \frac{2}{n} ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2 + \frac{\lambda}{2} ||\mathbf{w}||^2$$

- This now becomes the regularized least-squares.

4.2 Linear Models for Classification

- Recall that m defines the number of classes. To find the classification, requiring the model to return a certain integer is not a good idea. Instead we can regress on a real value $\tilde{y_i} \in [1, m]$. During inference, given the output $\hat{y_i} = f(\mathbf{x_i})$, we can map back to the original domain using:

$$\text{Predicated class} = \text{round}(\hat{y_i})$$

- However, this presents some disadvantages. By defining certain classifications as being close to each other i.e. class 2 is "closer" to class 3 than class 4, we trick the model into thinking that class 2 and class 3 are more related than class 2 and class 4. In reality, class 2 may be dog, class 3 a dolphin, and class 4 a cat.
- A method of avoiding this problem is using a **one-hot encoded** version of y, which we denote by $\mathbf{y}^{oh} \sim \text{Binary}(m)$.

$$[\mathbf{y}^{oh}]_j \begin{cases} 1 \text{ if } y = j \\ 0 \text{ otherwise} \end{cases}$$

- This notation provides some advantages in that the Euclidean distance between two classes is either 0 (same class) or $\sqrt{2}$ (different classes).
- This approach is not vaiable since this is discrete and we need a continiuous representation to train our dataset through gradient descent.
- A more better and more elegant solution exists in the form of **logistic regression**.

4.2.1 The Probability Simplex and the Softmax Function

- The **probability simplex** Δ_n is the set of vectors $\mathbf{x} \sim \Delta(n)$ such that:

$$x_i \geq 0, \sum_i x_i = 1$$

- Essentially, the probability simplex is the set of vectors whose elements describe the probability of the output classes and who all sum up to 1. Given a value $\mathbf{x} \in \Delta_n$, we can project to the predicted class using:

$$\arg_i \max \mathbf{x}_i$$

- An example: let the model output the following:

$$[0.2, 0.3, 0.4]$$

  where class 0 is dog, class 1 is cat, and class 2 is bird. In this case, the maximum would be 0.4 and as such, the model would ouput bird as the output class.

- The previous method of linear regression assumed y would be a scalar value. In this new implementation, we take y to be an m-dimensional vector:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where $\mathbf{W} \sim (m, c)$ is m linear regressions models running in parallel and $b \sim (m)$.

- After computing $\mathbf{y}$, it is not guaranteed to be in the simplex. That is why we must use a **softmax** function to project the output inside the simplex:
For a generic vector $\mathbf{x} \sim (m)$:

$$[\,\text{softmax}(\mathbf{x})\,]_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

- Breaking down the formula, this is what it does: First, exponentiation converts the numerator into a positive value.

$$h_i = \exp(x_i)$$

Next, we compute a normalization factor as the sum of exponentiated elements of the vector:

$$Z = \sum_j h_j$$

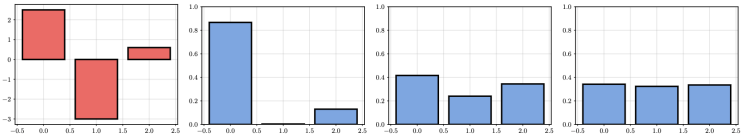
The output of the softmax is given by dividing h_i by Z, ensuring all elements of the final output vector after the softmax sum to 1:

$$y_i = \frac{h_i}{Z}$$

- We can also add another parameter $\tau > 0$ called the temperature:

$$\text{softmax}(\mathbf{x}; \tau) = \text{softmax}(\mathbf{x}/\tau)$$

- The softmax keeps the order between elements intact, but the temperature influences how far away they are from each other.



(a) *Inputs* (b) $\tau = 1$ (c) $\tau = 10$ (d) $\tau = 100$

Figure 2: Example of softmax function with temperature set to 1(b), 10(c), and 100(d)

- There are certain limiting cases:

$$\lim_{\tau \to \infty} \text{softmax}(\mathbf{x}; \tau) = 1/c$$

$$\lim_{\tau \to 0} \text{softmax}(\mathbf{x}; \tau) = \arg_i \max \mathbf{x}$$

4.2.2 The Logistic Regression Model

- Bringing together softmax and the parameterized output linear model, we get:

$$\hat{y} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

The prenomalized outputs $\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}$ are called the logits of the model.

- Covering the loss function, we can call upon previously discussed topics and derive this:

$$p(\mathbf{y}^{oh}|\hat{\mathbf{y}}) = \prod_i \hat{y}_i^{y_i^{oh}}$$

- The **cross-entropy** loss function between $\mathbf{y}^{0h}$ and $\hat{\mathbf{y}}$ is given by:

$$\text{CE}(\mathbf{y}^{oh}, \hat{\mathbf{y}}) = -\sum_i y_i^{oh} \log(\hat{y}_i)$$

  The cross-entropy loss function is widely used for classification tasks, especially in multi-class problems. It measures the dissimilarity between the true probability distribution (actual labels) and the predicted probability distribution (model's predictions). The goal of cross-entropy is to quantify how well the predicted probabilities match the actual labels.

- Given that only one value of $\mathbf{y}^{oh}$ will be non-zero corresponding to the true class $y = \arg_i \max\{y_i^{oh}\}$, we can simplify the loss as:

$$\text{CE}(y, \hat{\mathbf{y}}) = -\log(\hat{y}_y)$$

- It is evident that minimizing CE loss is equivalent to maximizing the output probability corresponding to the true class. Of course, the presence of the softmax function implies that an increase in the probabilithy of one ouput leads to a decrease in the probability of all other outputs. Putting everything together, we get the **logistic regression optimization problem**.

$$\text{LR}(\mathbf{W}, \mathbf{b}) = \frac{1}{n} \sum_{i=1}^{n} \text{CE}(\mathbf{y}_i^{oh}, \text{softmax}(\mathbf{W}\mathbf{x}_i + \mathbf{b}))$$

4.3 Additional Topics on Classification

4.3.1 Binary Classification

- With the special case of $m = 2$, we have the problem of **binary classification** also known as **concept learning**.

- Normally, the function would return an output made up of two elements, however this is unnecessary considering the softmax function ensures all probabilities sum up to 1. As such, we can rely on the function producing one output $\in$ [0,1] and derive the classification as:

$$\text{Predicted class} = \text{round}(f(\mathbf{x})) = \begin{cases} 0 \text{ if } f(\mathbf{x}) \leq 0.5 \\ 1 \text{ otherwise} \end{cases}$$

- To achieve the normalization, the first output of a two-valued softmax can be rewritten as:

$$\frac{\exp(x_1)}{1 + \exp(x_1)}$$

  Note: we assume x_0 to be 0, hence $e^0 = 1$ is in the denominator.

- Simplifying by dividing by $\exp(x_1)$, we get the **sigmoid** function $\sigma(s) : \mathbb{R} \to [0, 1]$ function:

$$\sigma(s) = \frac{1}{1 + \exp(-s)}$$

  The sigmoid maps any real number to a number between 0 and 1.

- The **binary logistic regression** model is obtained by combining a one-dimensional linear model with a sigmoid scaling of the output:

$$f(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

- The cross-entropy simplifies down to:

$$\mathrm{CE}(\hat{y}, y) = \underbrace{-y \log(\hat{y})}_{\text{class 1 loss}} \underbrace{-(1-y)\log(1-\hat{y})}_{\text{class 2 loss}}$$

- In the binary classification case, we can solve the problem with two equivalent approaches: (a) a two-valued model with standard softmax, or (b) a simplified one-valued output with a sigmoid output transformation.

- The gradient of the binary logistic regression model with respect to $\mathbf{w}$ is:

$$\nabla \mathrm{CE}(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)\mathbf{x}$$

  which is similar to the linear least-squares regression:

$$\nabla \mathrm{LS}(\mathbf{w}) = \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$$

- We may also rewrite our model as:

$$\underbrace{\mathbf{w}^\top \mathbf{x} + b}_{\text{logits}} = \underbrace{\log(\frac{y}{1-y})}_{\sigma^{-1}(y)}$$

4.3.2 The logsumexp Trick

- Consider the i-th term of the cross entropy in terms of the logits $\mathbf{p}$:

$$-\log(\frac{\exp p_i}{\sum_j \exp p_j})$$

  Due to the unknown magnitude of the logits, the exponentiation of the logits can cause numerical computing errors.

- To solve, we can rewrite as:

$$-p_i + \log(\sum_j \exp p_j)$$

  The first term does not suffer from instabilities, and the second term can be taken care of by using a scalar sum/difference c which won't change the final result when added back in:

$$\mathrm{logsumexp}(\mathbf{p}) = \mathrm{logsumexp}(\mathbf{p} - c) + c$$

4.3.3 Calibration and Classification

- We must highlight a key difference in the conclusions we get from a model's output.
  The following sentence is justified:

  > "The predicted class of $f(\mathbf{x})$ is $\arg_i \max [f(\mathbf{x})]_i$."

  But this one is not:

  > "The probability of $\mathbf{x}$ being of class i is $[f(\mathbf{x})]_i$."

- When the confidence scores of the network match the probability of a given prediction being correct, the network's outputs are **calibrated**.

- Different predictions may lead to different costs, and a *cost matrix* assigning a cost C_{ij} for any input of class i predicted as class j. An example is shown below: This may very well be the cost matrix for a classification model used in medical diagnosis. A false negative may be much more harmful than a false positive.

	True class 0	True class 1
Predicted class 0	0	10
Predicted class 1	1	0

Figure 3: Example of cost matrix for a classification problem having assymetric costs of missclassification

- A cost matrix $C \sim (m, m)$ can be created for a multiclass problem and used to minize the expected cost assigned by our model:

$$\arg_i \min \sum_{j=1}^{m} C_{ij}[f(\mathbf{x})]_j$$

  Essentially, you want to predict something by minimizing the combination of its original probability with the associated costs of choosing it.

4.3.4 Estimating the Calibration Error

- To estimate the calibration of a model, we can bin the predictions like this:
  - Split the interval [0.1] into b equispaced bins such that each bin is of size $1/b$.
  - Take a validation dataset of size n and denote by $\mathcal{B}_i$, the elements whose confidence falls into bin i.
  - For each bin, we can calculate the average confidence p_i of the model and the average accuracy a_i.
- To have a single scalar metric of calibration, we can use the **expected calibration error**:

$$\text{ECE} = \sum_i \underbrace{\frac{|\mathcal{B}_i|}{n}}_{\text{bin } i} \overbrace{|a_i - p_i|}^{calibration}$$

  The calibration is intended to punish instances where accuracy and confidence are not aligned together (low confidence/high accuracy & vice versa)
- A low expected calibration error suggests that the prediction confidence is aligned with the accuracy of the prediction.
- If the model is found to be calibrated, the model will need to be modified. We can do this by rescaling the predictions via temperature scaling or optimizing with a different loss function.
- As opposed to direct calibration of a model, we can do **conformal prediction**. Take a threshold probability γ. We can take the set of classes predicted by the modle whose corresponding probability is higher than γ.

$$\mathcal{G}(\mathbf{x}) = \{i \mid [f(\mathbf{x})]_i > \gamma\}$$

  Now the set $\mathcal{G}(\mathbf{x})$ is the set of potential classes.
- Essentially, we're defining a bare minimum level of probability of choosing the correct class. The minimum is γ, and the acceptable error is $1 - \gamma = \alpha$.

5 Fully-Connected Models

5.1 The Limitations of Linear Models

- As a review, linear models work by multiplying each feature of the input with a weight, then adding a bias to get the complete result.

- Linear models do not work well when features are not independent of each other.
- For example, two clients of a bank who are identical in all aspects except for their income, with $\mathbf{x}'$ having double the income of of $\mathbf{x}$. If f is a linear model with no bias, we have:

$$f(\mathbf{x}') = \overbrace{f(\mathbf{x})}^{\text{Original}} + \underbrace{w_j x_j}_{\text{Change induced by } \mathbf{x}_j = 2x_j}$$

- A consequence is that the change in input reflects a similar sized change in the output. This will not work for relationships where features can affect other features or certain combinations of features have different outputs. Take for example this scenario where we're trying to score the users: *an income of 1500 is low, except if the age $<$ 30.*

5.2 Composition and Hidden Layers

- Imagining our model f is a composition of two trainable operations results in :

$$f(\mathbf{x}) = (f_2 \odot f_1)(\mathbf{x}) = f_2(f_1(\mathbf{x}))$$

Being more specific, we describe the relation as:

$$\mathbf{h} = f_1(\mathbf{x}) = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$
$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2$$

But doing this will result in the two projections collapsing into a single one:

$$y = (\underbrace{\mathbf{w_2}^\top \mathbf{W_1}}_{\mathbf{A}})\mathbf{x} + (\underbrace{\mathbf{w}_2^\top \mathbf{b}_1 + b_2}_{\mathbf{c}}) = \mathbf{A}\mathbf{x} + \mathbf{c}$$

- The idea behind **fully-connected** (FC) models, also known as **multi-layer perceptrons** (MLPs), is to insert a simple elementwise non-linearity $\phi : \mathbb{R} \to \mathbb{R}$ in-between projections to avoid the collapse:

$$\mathbf{h} = f_1(\mathbf{x}) = \phi(\mathbf{W_1}\mathbf{x} + \mathbf{b_1})$$

$$y = f_2(\mathbf{h}) = \mathbf{w_2}^\top \mathbf{h} + b_2$$

The function ϕ can be non-linearity such as a polynomial, a square-root, or the sigmoid function σ. A good choice for the function is something which is non-linear enough to prevent collapse, but linear enough to preserve the original identity in its derivative.

A good default choice is the **rectified linear unit** (ReLU)

- The **rectified linear unit** is defined elementwise as:

$$\text{ReLU}(s) = \max(0, s)$$

- With the addition of ϕ, we can chain as many transformations as we want:

$$y = \mathbf{w}_l^\top \phi(\mathbf{W}_{l-1}(\phi(\mathbf{W}_{l-2}\phi(\ldots) + \mathbf{b}_{l-2})) + \mathbf{b}_{l-1}) + b_l$$

On the Terminology in Differentiable Models

This will serve as an summary of terminology.

- Each f_i is called a **layer** of the model, with f_l being the **output layer**, $f_i, i = 1, \ldots, l-1$ the **hidden layers**.
- With some notation overloading, $\mathbf{x}$ is the input layer.
- With this terminology, we can restate the definition of the **fully-connected layer** in batched form:
- For a batch of n vectors, each of size c, represented as a matrix $\mathbf{X} \sim (n, c)$, a **fully-connected** (FC) layer is defined as:

$$\text{FC}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

- The parameters of the layer are the matrix $\mathbf{W} \sim (c', c)$ and the bias vector $\mathbf{b} \sim (c')$, for a total of $(c' + 1)c$ parameters (assuming ϕ does not have any parameters). Its hyper-parameters are the width c' and the non-linearity ϕ.

  Reminder: A hyperparameter is one set by the user to help define the behavior of the model.

  Note: The above definition is one from the book and may contain an error. The expression $\mathbf{XW}$ is not valid if $\mathbf{X}$ is of dimensions $n \times c$ and $\mathbf{W}$ is of dimensions $c' \times c$. The no. of columns in the first matrix does not match the no. of rows in the second. The product $\mathbf{XW}$ should be of dimensions $n \times c'$.

  Note: The addition of the product $\mathbf{XW}$ and the bias $\mathbf{b}$ might be seen as invalid due to dimension mismatches, however ML frameworks can extend the bias' dimensions to allow addition to occur. So the result of $\mathbf{XW} + \mathbf{b}$ would have the dimensions $n \times c'$.

- The outputs $f_i(\mathbf{X})$ are called the **activations** of the layer. We can also differentiate between the **pre-activation** and the **post-activation** (before and after the non-linearity has been applied).
- An implementation of a fully-connected layer using PyTorch:

```
import torch
import torch.nn as nn
from torch.nn.functional import relu

class FullyConnectedLayer(nn.Module):
    def __init__(self, c: int, cprime: int):
        super().__init__()
        # Initialize the parameters
        self.W = nn.Parameter(torch.randn(c, cprime))
        self.b = nn.Parameter(torch.randn(1, cprime))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return relu(x @ self.W + self.b)
```

- The non-linearity ϕ can be called the **activation function** and each output of f_i is called a **neuron**. The width (shape of the output) of each layer is a hyperparameter that can be set by the user. The width only affects the input going into the next layer.

5.2.1 Approximation Properties of MLPs

- Training MLPs is similar to training linear models where we minimize a loss function.
- Linear models has one global minimum that gradient descent would be guided toward. However, given the non-linearity of multiple layers, there can exist many local minima with no guranatee that gradient descent will the find the global minimum. Regardless, research has shown that non-convex models can still provide good performance.

- The question naturally arises, if linear models can only solve tasks which are linearly separable, what class of tasks can non-linear models be used for?

  Having a single hidden layer is enough to have **universal approximation** capabilities.

- **Universal approximatino of MLPs:** Given a continuous function $f : \mathbb{R}^d \to \mathbb{R}$, we can always find a model $f(\mathbf{x})$ that is a MLP with a single hidden layer such that:

$$|f(\mathbf{x}) - g(\mathbf{x})| \leq \epsilon, \forall \mathbf{x}$$

- In english, what this means is that given a function, we can find a model that can approximate that function with a certain degree of acceptable error ϵ.

5.3 Stochastic Optimization

- When n (the size of the dataset) grows very large, preserving the linear time complexity becomes unreasonable especially when n is in the order of 10^4 or more.

- A solution to this problem is to use subsets of data to computer descent direction. To this end, for iteration t of gradient descent we sample a subset $\mathcal{B}_t \subset \mathcal{S}_n$ of r points (with $r \ll n$). Shuffling the dataset, splitting into batches, then training with each subset of the data, and finally reshuffling the dataset before training again.

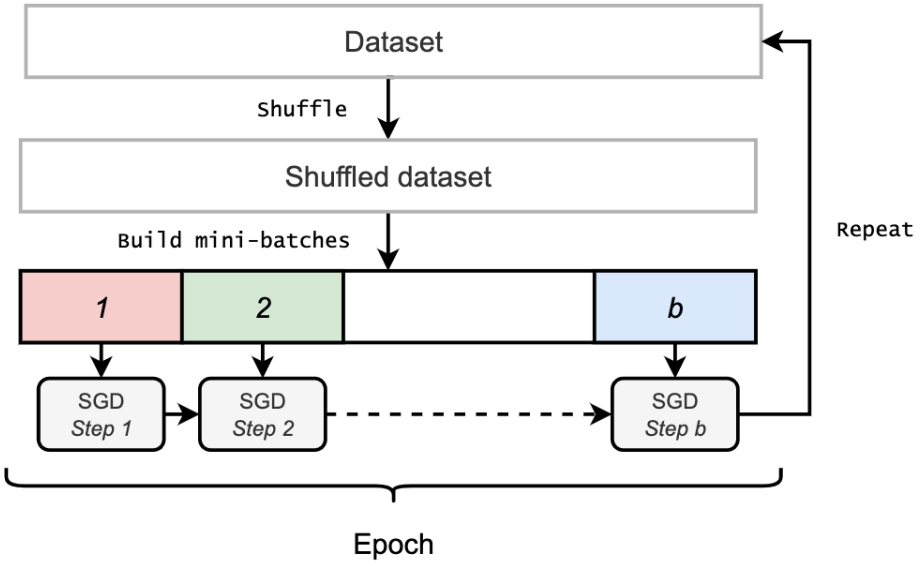


Figure 4: Building the mini-batch sequence

- Computing the approximated loss only considering the minibatch as:

$$\widetilde{L}_t = \underbrace{\frac{1}{r} \sum_{(x_i, y_i) \in \mathcal{B}_t}}_{\text{Mini-batch}} l(y_i, f(x_i)) \approx \underbrace{\frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{S}_n}}_{\text{Full dataset}} l(y_i, f(x_i))$$

- For the computation of the gradient and the loss for a batch, the complexity grows with r. Having lower values of r results in higher gradient variance, while higher r results in slower and mroe precise iterations.

- Gradient descent applied on mini-batches of data is called **stochastic gradient descent** (SGD).
- How can we create the mini-batches? Sampling random datapoints can be computationally expensive with large datasets, so another approach is needed. We can begin by shuffling the dataset, selecting batches of consecutive elements, training with them, then shuffling again. Assuming a database of size $n = br$, we will have b batches of data with r elements in them.
- One complete loop of the shuffle/batch/train cycle is called an **epoch** of training. The amount of epochs in the training stage is a hyperparameter that can be set by the user. It is given by $\frac{n}{b}$. Setting the batch size will set the number of epochs.

5.4 Activation Functions

- Regarding the selection of the non-linearity function, any element-wise non-linearity function is valid, but not all of them have good performance or numerical stability. For example, the non-linearity function
$$\phi(s) = s^p$$
does preserve linearity but with increasing quantities of s, the function becomes computationally expensive and numerically unstable.
- Looking through the frame of neurology, the weights $\mathbf{w}^\top$ in the dot product $\mathbf{w}^\top\mathbf{x}$ were simple models of synapses, the bias b was the threshold, and the neuron was activated when the sum of the inputs surpased the threshold.
$$s = \mathbf{w}^\top\mathbf{x} - b, \phi(s) = \mathbb{I}_{s\geq 0}$$
where $\mathbb{I}_b$ is an indicator function that is 1 if b is true, else 0.
- This indicator function is not differentiable, so we cannot run gradient descent on it. As an alternative, we can use the sigmoid function with a tunable slope a such that $\sigma_a(s) = \sigma(as)$. Therefore, we have:
$$\lim_{a\to\text{inf}} \sigma_a(s) = \mathcal{I}_{s\geq 0}$$
- Modern neural networks use ReLU:
$$\text{ReLU}(s) = \begin{cases} s \text{ if } s \geq 0 \\ 0 \text{ if } s < 0 \end{cases}$$
This function is not differentiable and can result in *dead neurons* because all negative inputs are set to 0. To fix this, we can use the **Leaky ReLU**:
$$\text{LeakyReLU}(s) = \begin{cases} s \text{ if } s \geq 0 \\ as \text{ otherwise} \end{cases}$$
for a very small a e.g. $a = 0.01$.
- We can train a different a for each neuron, since the function is differentiable w.r.t. a. Forcing the same a on all neurons is not ideal as it would limit their flexibility to respont to diffeerent features of the input. An activation function using the leaky ReLU is called a **parameteric ReLU** (PReLU).
- Fully-differentiable variants of ReLU are available, such as the **softplus**:
$$\text{softplus}(s) = \log(1 + \exp(s))$$
which does not pass through the origin and is always ¿ 0.
- The **exponential linear unit** (ELU):
$$\text{ELU}(s) = \begin{cases} s \text{ if } s \geq 0 \\ \exp(s) - 1 \text{ otherwise} \end{cases}$$

6 Automatic Differentiaion

6.1 Problem Setup

- Assume we have a set of **primitives**:

$$\mathbf{y} = f_i(\mathbf{x}, \mathbf{w}_i)$$

Each primitive represents an operation on an input vector $\mathbf{x} \sim (c_i)$, parameterized by vector $\mathbf{w}_i \sim (p_i)$, and giving as output another vector $y \sim (c_1')$.

- We assume for each primitive, we can calculation the partial derivatives w.r.t. to the input arguments. We can do this due to us selectively choosing differentiable models.

$$\textbf{Input Jacobian: } \partial_x[f(\mathbf{x}, \mathbf{w})] \sim (c', c)$$

$$\textbf{Weight Jacobian: } \partial_x[f(\mathbf{x}, \mathbf{w})] \sim (c', p)$$

6.1.1 Automatic Differentiation: Problem Statement

- Consider a sequence of l primitive calls, followed by final summation:

$$\begin{aligned}
\mathbf{h}_1 &= f_1(\mathbf{x}, \mathbf{w}_1) \\
\mathbf{h}_2 &= f_2(\mathbf{x}, \mathbf{w}_2) \\
&\vdots \\
\mathbf{h}_l &= f_l(\mathbf{x}, \mathbf{w}_l) \\
y &= \sum \mathbf{h}_l
\end{aligned}$$

This is the **evaluation trace** of the program. The first $l-1$ operations represent several layers of the differentiable model, operaiton l can be a per-input loss (e.g. cross-entropy). The final operation sums up the losses and creates a scalar value to represent the total loss. This whole program can be abbreviated as $F(\mathbf{x})$.

- **Automatic Differentiation**:The gradient of the loss function can be calculated by using the chain rule and applying it to the loss functions in the previous layers of the model:

$$\text{AD(F}(\mathbf{x})) = \{\partial_{\mathbf{w}_i} y\}_{i=1}^{L}$$

- There are two major classes of AD: **forward-mode** and **backward-mode**. Backward mode, also known as **back-propogation**, is much more efficient.

6.1.2 Numerical and Symbolic Differentiaion

- There are two ways to differentiate functions. We can do **numerical integration** which applies the fundamental theorem of calculus:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

However this requires two function calls and is computationally inefficient.

- Another approach is **symbolic differentiation** where we use a symbolic engine to pre-compute the full, symbolic equation of the derivative. One such output could be:

$$f(x) = a\sin(x) + bx\sin(x)$$

In a realistic implementation, we would compute the value $\sin(x)$ and store it so we would not have to compute it again.

6.2 Forward-Mode Automatic Differentiation

- **Forward Automatic Differentiation** is a process that when reaching its end, stores how much each layers ouputs contributes to the entire neural network's output.
- It does this by a series of steps:
  - For a new layer i, we calculate $\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}$. If we are on the first layer (i.e. no layers before current layer), we initialize to $\frac{\partial \mathbf{h}_1}{\partial \mathbf{w}_1}$.
  - For all layers before the current layer, we multiply $\frac{\partial \mathbf{h_i}}{\partial \mathbf{h}_{i-1}}$ to their gradients w.r.t to their weights.
  - For our current layer, we then initialize $\frac{\partial \mathbf{h}_i}{\partial \mathbf{w}_i}$.
  - Move onto next layer.

  A more formal mathematical explanation is as follows:

  - Consider the first instruction, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, in our program. Because nothing has been stored, we initialize the tangent matrix for w_1 as its weight Jacobian:

$$\widehat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$$

  - Move onto next instruction: $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$. We update the previous tangent matrix whiel simulataneously initializing the second one:

$$\overbrace{\widehat{\mathbf{W}}_1}^{\substack{\text{updated } \mathbf{w}_1 \\ \text{tangent} \\ \text{matrix}}} \leftarrow \overbrace{\left[\partial_{\mathbf{h}_1} \mathbf{h}_2\right]}^{\substack{f_2 \text{ input} \\ \text{Jacobian}}} \widehat{\mathbf{W}}_1$$

$$\widehat{\mathbf{W}}_2 = \frac{\partial \mathbf{h}_2}{\mathbf{w}_2}$$

  - Considering a more general case of the i-th layer given by $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$, we can initialize the tangent matrix for $\mathbf{w}_i$ while updating all previous matrices:

$$\widehat{\mathbf{W}}_j \leftarrow \overbrace{\left[\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i\right]}^{\substack{f_i \text{ input} \\ \text{Jacobian}}} \widehat{\mathbf{W}}_j \quad \forall j < i$$

$$\widehat{\mathbf{W}}_2 = \frac{\partial \mathbf{h}_2}{\mathbf{w}_2}$$

Table 1: Dimensions of key components in forward-mode automatic differentiation (FAD).

Quantity	**Symbol**	**Dimensions**
Input to layer i	$\mathbf{h}_{i-1}$	$d_{\text{in}} \times 1$
Weights of layer i	$\mathbf{W}_i$	$d_{\text{out}} \times d_{\text{in}}$
Bias of layer i	$\mathbf{b}_i$	$d_{\text{out}} \times 1$
Output of layer i	$\mathbf{h}_i$	$d_{\text{out}} \times 1$
Final network output	$\mathbf{h}_n$	$d_{\text{out}}^{\text{final}} \times 1$
Weight tangent (FAD)	$\widehat{\mathbf{W}}_i$	$d_{\text{out}}^{\text{final}} \times (d_{\text{out}} \cdot d_{\text{in}})$
Bias tangent (FAD)	$\widehat{\mathbf{b}}_i$	$d_{\text{out}}^{\text{final}} \times d_{\text{out}}$
Row vector of ones	$\mathbf{1}^\top$	$1 \times d_{\text{out}}^{\text{final}}$
Final gradient (weights)	$\nabla_{\mathbf{W}_i} y$	$1 \times (d_{\text{out}} \cdot d_{\text{in}})$
Final gradient (biases)	$\nabla_{\mathbf{b}_i} y$	$1 \times d_{\text{out}}$

- The space complexity of this algorithm will roughly be proportional to the space complexity of the program we are differentiating.
- However, the time complexity of the algorithm will be $\mathcal{O}(n^2 d^2 p_j)$ where n is the mini-batch dimension, d represents the input/output features, and p_j is the size of $\mathbf{w}_j$.

6.3 Reverse-Mode Automatic Differentiation

- Let's unroll the computation of a single gradient term corresponding to the i-th weight matrix:

$$\nabla_{\mathbf{w}_i} y = \mathbf{1}^\top \left[\partial_{\mathbf{h}_{i-1}} \mathbf{h}_l\right] \dots \left[\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}\right] \left[\partial_{\mathbf{w}_i} \mathbf{h}_i\right]$$

- What it's saying is the gradient of the output w.r.t to any set of weights $\mathbf{w}_i$ is $\mathbf{1}^\top$ times the product of the derivative of the ouput $\mathbf{h}_i$ w.r.t to the weights $\mathbf{w}_i$ and derivative of all subsequent outputs w.r.t to their previous output.
- We can combine the above formula to this:

$$\widetilde{\mathbf{h}}_i = \mathbf{1}^\top \prod_{j=i+1}^{l} \partial_{h_{j-1}} \mathbf{h}_j$$

- Because matrix multiplication associative, we can peform the multiplications in any order. We can note the following:
  - Vector-matrix multiplication is computaitonally better than matrix-matrix multiplication. Its output is also another vector.
  - The product of all input Jacobians from layer i to layer l can be computed recursively starting from the *last* term and iteratively multiplying by the input Jacobians in the reverse order.
- This is the means by which **reverse-mode automatic differentiation** works:
  - Differently from FAD, we execute the *entire* program to be differentiated, storing all immediate outputs.
  - We initialize a vector $\widetilde{\mathbf{h}}_i = \mathbf{1}^\top$, which corresponds to the leftmost term in the previous equation
  - Moving in reverse order, for an index $i \in [1, l]$, we first compute the gradient with respect to the i-th weight matrix as:

    $$\partial_{\mathbf{w}_i} y = \widetilde{\mathbf{h}} \left[\partial_{\mathbf{w}_i} \mathbf{h}_i\right]$$

    This is the value which will be stored for each layer.
  - Then we update our back-propagated input Jacobian as:

    $$\widetilde{\mathbf{h}} \leftarrow \widetilde{\mathbf{h}} \left[\partial_{h_{i-1}} h_i\right]$$

- In more plain english:
  - At each layer i, we recieve the upstream gradient:

    $$\frac{\partial y}{\partial h_i}$$

  - We compute the local Jacobian:

    $$\frac{\partial h_i}{\partial w_i}$$

  - We store the product of the previous two values in our current layer:

    $$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_i}$$

– We then update the value we pass on to the previous layer:

$$\widetilde{\mathbf{h}}_i \leftarrow \widetilde{\mathbf{h}}_i \left[\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right]$$
$$\widetilde{\mathbf{h}}_i \leftarrow \frac{\partial y}{\partial \mathbf{h}_i} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{h}_{i-1}}$$
$$\widetilde{\mathbf{h}}_i \leftarrow \frac{\partial y}{\partial \mathbf{h}_{i-1}}$$

The previous steps describe a program which is roughly symmetrical to the original program, which we call the **dual** or **reverse** program. The terms $\widetilde{\mathbf{h}}$ are called the **adjoints** and they store sequentially all the gradients of the output with respect to the variables $mbfh_1, mbfh_2, \dots mbfh_l$.

Table 2: Dimensions of key components in backward-mode automatic differentiation (Backpropagation).

Quantity	**Symbol**	**Dimensions**
Activation at layer i	$\mathbf{h}_i$	$d_{\text{out}} \times 1$
Weights of layer i	$\mathbf{W}_i$	$d_{\text{out}} \times d_{\text{in}}$
Bias of layer i	$\mathbf{b}_i$	$d_{\text{out}} \times 1$
Upstream gradient	$\tilde{\mathbf{h}}_i = \frac{\partial y}{\partial \mathbf{h}_i}$	$1 \times d_{\text{out}}$
Local Jacobian (w.r.t. weights)	$\frac{\partial \mathbf{h}_i}{\partial \mathbf{W}_i}$	$d_{\text{out}} \times (d_{\text{out}} \cdot d_{\text{in}})$
Local Jacobian (w.r.t. input)	$\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}$	$d_{\text{out}} \times d_{\text{in}}$
Gradient w.r.t. weights	$\nabla_{\mathbf{W}_i} y$	$1 \times (d_{\text{out}} \cdot d_{\text{in}})$
Gradient w.r.t. biases	$\nabla_{\mathbf{b}_i} y$	$1 \times d_{\text{out}}$
Backpropagated value	$\tilde{\mathbf{h}}_{i-1} = \frac{\partial y}{\partial \mathbf{h}_{i-1}}$	$1 \times d_{\text{in}}$

– In the terminology of neural networks, we sometimes say the original (**primal**) program is a **forward** poss (not forward mode auto-diff) and the reverse program is a **backward pass**.

– Differently from FAD, RAD must store the intermediate outputs to "unroll" the computational graph.

– Computationally, RAD is much more efficient than FAD because matrix-vector multiplication has lower time complexity than matrix-matrix multiplication. However, RAD does need much more memory to store all the intermediate ouputs.

– Specific techniques such as **gradient checkpointing** can be used to improve on this tradeoff by increasing number of computations in exchange for reduced memory demand. It does this by storing only some intermediate outputs at certain checkpoints, and reclaculating the outputs in the backward pass.

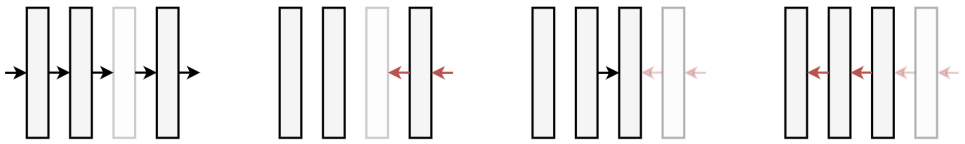


Figure F.6.1: *An example of **gradient checkpointing**. (a) We execute a forward pass, but we only store the outputs of the first, second, and fourth blocks (**checkpoints**). (b) The backward pass (red arrows) stops at the third block, whose activations are not available. (c) We run a second forward pass starting from the closest checkpoint to materialize again the activations. (d) We complete the forward pass. Compared to a standard backward pass, this requires 1.25x more computations. In general, the less checkpoints are stored, the higher the computational cost of the backward pass.*

6.4 Practial Considerations

6.4.1 Vector-Jacobian Products

- Looking at step (4) in the RAD algorithm, we can note that the only operaiton we need is a product between row vector v and Jacobian of f. We can restore dimensional consistency by addding a transpose to the vector:
- Gven a function $\mathbf{y} = f(\mathbf{x})$ with $\mathbf{x} \sim (c)$ and $\mathbf{y} \sim (c')$, the **Vector-Jacobian product** is another function defined as:

$$\text{vjp}_f(\mathbf{v}) = \mathbf{v}^\top \partial f(\mathbf{x})$$

  where $\mathbf{v} \sim (c')$. If f has multiple parameters $f(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, we can define n individual VJPs as $\text{vjp}_{f,\mathbf{x}_1}(\mathbf{v}), \ldots, \text{vjp}_{f,\mathbf{x}_n}(\mathbf{v})$.
- In our case, we can define two types of VJPs, corresponding to the input and weight argument respectively:

$$\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$$
$$\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$$

- Now we can rewrite the two operations in step (4) of the RAD algorithm as two VJP calls of the primitive function with the adjoint values, corresponding to the adjoint times the weight VJP, and the adjoint times the input VJP:

$$\partial_{\mathbf{w}} y = \text{vjp}_{f,\mathbf{w}}(\widetilde{\mathbf{h}})$$
$$\widetilde{\mathbf{h}} \leftarrow \text{vjp}_{f,\mathbf{h}}(\widetilde{\mathbf{h}})$$

- To recover the Jacobians' computation, we can repeatedly call the VJPs with the basis vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$, to generate them one row at a time e.g., for the input Jacobian we have:

$$\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) = \begin{bmatrix} \text{vjp}_{f,x}(\mathbf{e}_1) \\ \text{vjp}_{f,x}(\mathbf{e}_2) \\ \vdots \\ \text{vjp}_{f,x}(\mathbf{e}_n) \end{bmatrix}$$

- To put into other words, we don't care about how *each input* affects *each output*. We only care about how the *total output* affects the loss and how the *total input* affects the total output.

- Let's look at the VJPs of a fully-connected layer, which is composed of linear projections and elementwise non-linearities. Consider a linear projection with no bias:

$$f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$$

- The input Jacobian is simply $\mathbf{W}$, but the weight Jacobian is a rank-3 tensor. By comparison, the input VJP has no special structure:

$$\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \mathbf{W}^\top = [\mathbf{W}\mathbf{v}]^\top$$

- The weight VJP, instead, turns out to be a simple outer product, which avoids rank-3 tensors completely:

$$\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}\mathbf{x}^\top$$

6.4.2 Implemeneting a R-AD System

- Take a function $f : \mathbb{R}^c \to \mathbb{R}^{c'}$. Then, $\frac{\partial f}{\partial \mathbf{x}} \in \mathbb{R}^{c' \times c}$. A VJP of vector $\mathbf{v} \in \mathbb{R}^{c'}$ returns:

$$\mathbf{v}^\top \cdot \frac{\partial f}{\partial \mathbf{x}} \in \mathbb{R}^c$$

6.4.3 Choosing an Activation Function

- When training deep neural networks, you compute gradients by applying the chain rule layer by layer:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{h}_L} \cdot \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_1}{\partial \mathbf{x}}$$

  If each layer has an activation function ϕ, then in each layer you multiply by its derivative ϕ'.

- With models with many layers, this can give rise to two pathalogical behaviors:
  - If $\phi'(.) < 1$ everywhere, there is a risk of the gradient being shrank to 0 exponentially fast through the layers. This is the **vanishing gradient** problem.
  - However if $\phi(.) < 1$ everwhere, there is a risk of the gradient converging to infinity be represented properly. This is the **exploding gradient** problem.

- We need a function which is linear enough to avoid gradient issues but non-linear enough to separate the linear layers. The ReLU ends up being a good candidate since:

$$\partial_s \,\text{ReLU}(s) \begin{cases} 0 & s < 0 \\ 1 & s > 0 \end{cases}$$

  The gradient is either zeroed-out, inducing sparsity in the computation or multiplied by 1, avoiding scaling issues.

6.4.4 Subdifferentiability and Correctness of AD

- The ReLU is non-differentiable in 0, making the whole network not smooth. What happens now?

- In practical applications, the probability of ending up at exactly $s = 0$ is null, while the gradient is defined in $\text{ReLU}(\epsilon)$ for any $|\epsilon| > 0$.

- But mathematically, we can make use of subgradients. Given a convex function $f(x)$, a subgradient in x is a point z such that, for all y:

$$f(y) \geq f(x) + z(y - x)$$

- A subgradient is the slope of a line tangent to $f(x)$ such that the entire f is lower bounded by it. If f is differentiable in x, then only one such line exists, which is the derivative of f in x. In a non-smooth point, multiple subgradients exist and they form a set called the subdifferential of f in x:

$$\partial_x f(x) = \{z | z \text{ is a subgradient of } f(x)\}$$

- With this definition in hand, we can complete our analysis of the gradient of the ReLU function:

$$\partial_s \operatorname{ReLU}(s) = \begin{cases} 0 & s < 0 \\ 1 & s > 0 \\ [0, 1] & s = 0 \end{cases}$$

- Any value in [0,1] is a valid subgradient in 0, with most implementations favoring $\operatorname{ReLU}'(0) = 0$. Selecting subgradients at every step of an iterative descent procedure is called subgradient descent.