EE147: Neural Networks and Deep Learning

Professor Kao

Thilan Tran

Winter 2021

Contents

EE147: Neural Networks and Deep Learning

- deep learning has many modern applications:
  - Google search
  - Youtube video recommendations
  - Yelp restuarant best foods
  - Instagram feeds
  - very smart image recognition:
    - what makes a car a car?
    - cannot classify purely based on physical attributes such as size or number of wheels
    - image may be obscured or obfuscated
  - fraud detection
  - cancer treatment
  - self driving car:
    - many concerns eg. traffic cones, school buses, pedestrian signals, police cars
    - much expressive capacity is required
  - AlphaGo, Deepmind's AI that plays Go:
    - there are more Go board configurations than atoms in the unvierse
    - cannot do something as simple as a tree search
    - although AlphaGo was trained off of *"big data"* of Go experts:
      - its successor AlphaGo Zero uses deep reinforcement learning, *without* using any human data
      - algorithm rather than data based
  - brain-machine interfaces

History

- the concept of neural networks have been around for a long time, since McCulloch and Pitts in 1943:
  - but has only become relevant as of recently
  - this early model was inspired by the nervous system activity (and did not have the capacity to learn):
    1. all or none: a brain neuron either fires or not, ie. 1 or 0
    2. synpases can sum together to trigger a neuron
- in 1958, Rosenblatt propsed the first **NN (neural network)** that could learn:
  - called the **perceptron**, it had a learning rule to train it to perform clas-

sifications
  - had m neurons or inputs, , m weights, a bias b, and a value v
    * where $v = w_1 x_1 + ... + w_m x_m + b$
  - this early perceptron had a **hard-limiter** function φ st. the output $y = \varphi(v)$ and $\varphi(x)$ outputs 1 if $x > 0$ and otherwise 0
    * inspired by observation (1)
  - the perceptron could act as a linear classifier with one layer
    * but failed for nonlinear classifications, such as the XOR problem, with only one layer
  - thus, there was a lot of pessimism towards researching multilayer neural networks around this time

- researchers would continue to use biological inspiration for developing neural networks:
  - in 1962, Hubel and Wiesel published research on the cat V1 visual neural system
  - Fukushima's neocognitron from 1982 used the insights from these visual system in a new neural network architecture
- in 1986, Rumelhart used **backpropagation** to finally train multilayer neural networks:
  - a new way to train multilayer perceptrons by essentially using the chain rule to pass partial derivatives
- in 1989, LeCun and researchers at Bell Labs used neural networks to recognize handwritten zipcodes from the MNIST dataset
- in 1998, LeCun introduced LeNet, the modern **CNN (convolutional neural network)**, similarly inspired by visual cortex experiments:
  - took inspiration from spatial independence and simple linear composition of neurons in the V1 system
  - but still just a loose inspiration, eg. neurons in brains have probabilistic rather than static weights
- why didn't CNNs and backpropagation develop widespread use then?
  - backpropagation was still only good for shallow neural networks
    * as networks are deeper, the propagated derivative becomes more inaccurate
  - in addition, neural networks are data hungry
- modern era of deep learning:
  - the famous large ImageNet dataset with over 1000 classes of images held a yearly competition

    - within a decade, deep learning teams improved drastically in the ImageNet competition, from a 25% error rate to less than 5%
- driven by the massive amount of data we have access to and computation power to process it
    - GPU hardware have accelerated the training of NNs
- trend of more and more layers used in neural networks

Basics of Machine Learning

- **machine learning** uses statistical tools to estimate ie. *learn* functions, some of which may be fairly complex:
  - **classification** produces a discrete output representing the category given an input vector $x \in \mathbb{R}^n$:
    - ie. which of k categories x belongs to
    - eg. classifying whether an image is a cat or dog = class focuses on this type of function
  - **regression** produces an analog output predicting the value given an input
    - eg. predicting housing prices from square footage, controlling position and velocity of a cursor through brain signals
  - **synthesis and sampling** generate new examples that resemble a training data
    - eg. used in generative adversarial networks (GANs)
  - **data imputation** fills in missing values of a vector
    - eg. Netflix predicting if you will like a show or movie
  - **denoising** takes a corrupt data sample and outputs a cleaner sample
    - eg. used in variational autoencoders
  - other types
- in **supervised learning**, input vectors x and their target vectors y are known:
  - the goal is to learn function $y = f(x)$ that predicts y given x
  - eg. takes in a dataset D of n tuples of data
- in **unsupervised learning**, goal is to discover structure in input vectors, absent of knowledge of target vectors
  - eg. finding similar input vectors in clustering, distributions of the inputs, visualization, etc.
- in **reinforcement learning**, goal is to find suitable actions in certain scenarios to maximize a given reward R
  - discovers policies through trial and error
- in this class, we will focus on supervised learning:
  - using the CIFAR-10 dataset for an image classification problem:
    - 10 possible image categories
    - 32 by 32 pixel images, represented as 32 by 32 by 3 data values (RGB colors)
    - ie. input vector $x \in \mathbb{R}^{3072}$
  - want to find a function $f(x)$ that outputs one of the 10 categories

Supervised Learning Example

- for a problem of renting a home in Westwood, we want to know if we were getting a good deal:
  - given the square footage of a house, output how much monthly rent we should expect to reasonably pay based on the training data we have
- first, we should determine how we model data:
  1. determine inputs and outputs
     - input x is the square footage, and the output y is the rent
  2. what model should we use?
     - try a linear model $y = ax + b$
       - a, b are the **parameters** that must be found in this chosen model
     - a different model could have been chosen eg. a nonlinear, higher order polynomial
       - many more parameters to tune with
  3. how do we assess how good our model is?
     - we need a **loss function** that *scores* how good the model is
     - for a prediction $\hat{y}_i = f(x_i)$ and actual sample output y_i, we can use a least squares loss function:

$$\textit{loss} = \textit{cost} = \sum_i (y_i - \hat{y}_i)^2$$

       - note that using least squares rather than absolute value puts higher weight on outlilers
- transforming with vectors:
  - writing the model using vectors where $\theta = \begin{bmatrix} a \\ b \end{bmatrix}$ and $\hat{x} = \begin{bmatrix} x \\ 1 \end{bmatrix}$:

$$\begin{aligned} \hat{y} &= ax + b \\ &= \theta^T \hat{x} \end{aligned}$$

  - writing the cost function using vectors where k is a normalization constant:

$$\begin{aligned} L(\theta) &= k \sum_i (y_i - \hat{y}_i)^2 \\ &= k \sum_i (y_i - \theta^T \hat{x}_i)^2 \end{aligned}$$

- we want to make loss $L(\theta)$ as *small* as possible, since θ represents the parameters we can control:
  - in this case, $L(\theta)$ will look like a parabola since it is squared
    - can solve for its minimum using optimization

1. calculate $\frac{dL}{d\theta}$
   - tells us the slope of the line with respect to θ
2. solve for θ such that $\frac{\partial L}{\partial \theta} = 0$

- however, θ is a vector, so how do we take derivatives with respect to it?
  - these derivatives are typically called **gradients** eg. $\frac{\partial y}{\partial x}$ or $\nabla_x y$
  - can be done with respect to vectors or matrices

- rewriting the cost function:

$$
\begin{aligned}
L &= \frac{1}{2}\sum_{i=1}^{N}(y_i - \theta^T \hat{x}_i)^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}(y_i - \theta^T \hat{x}_i)^T(y_i - \theta^T \hat{x}_i) \\
&= \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{x}_i^T \theta)^T(y_i - \hat{x}_i^T \theta) \\
&= \frac{1}{2}(\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} \hat{x}_1^T \\ \vdots \\ \hat{x}_N^T \end{bmatrix} \theta)^T(\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} \hat{x}_1^T \\ \vdots \\ \hat{x}_N^T \end{bmatrix} \theta) \\
&= \frac{1}{2}(Y - X\theta)^T(Y - X\theta) \\
&= \frac{1}{2}(Y^T - \theta^T X^T)(Y - X\theta) \\
&= \frac{1}{2}[Y^T Y - Y^T X\theta - \theta^T X^T Y + \theta^T X^T X\theta] \\
&= \frac{1}{2}[Y^T Y - 2Y^T X\theta + \theta^T X^T X\theta]
\end{aligned}
$$

  - where $Y \in \mathbb{R}^{N\times 1}$ and $X \in \mathbb{R}^{N\times 2}$
    - note that $\theta^T \hat{x}_i = \hat{x}_i^T \theta$ and $Y^T X\theta = \theta^T X^T Y$ since they are all scalars and inner product is commutative
  - we used **vectorization** to move from summation to a sum expressed as an equivalent inner product of vectors

- now we can take derivatives to optimize the cost function:

$$
\begin{aligned}
\frac{\partial L}{\partial \theta} &= \frac{1}{2}[0 - 2X^T Y + [X^T X + X^T X]\ \theta] \\
&= -X^T Y + X^T X\theta\ [=]\ 0 \\
X^T Y &= X^T X\theta \\
\theta &= (X^T X)^{-1} X^T Y \\
&\triangleq X^{\dagger} Y
\end{aligned}
$$

  - recall that $\frac{\partial z^T \theta}{\partial \theta} = z$ and $\frac{\partial \theta^T A\theta}{\partial \theta} = (A + A^T)\theta$

    - $Y^T X$ can be considered as a vector z
  - this solution $\theta = X^\dagger Y$ is called the **least-squares solution**
    - gives us the best parameters θ to minimize the least-squares cost
- alternatively, using the chain rule to optimize the cost function:

$$\begin{aligned} f(z) &= z^2 \\ \frac{\partial f}{\partial z} &= 2z \\ g(\theta) &= y_i - \theta^T x_i \\ \frac{\partial g}{\partial \theta} &= -x_i \end{aligned}$$

$$\begin{aligned} \frac{\partial f}{\partial \theta} &= \frac{1}{2}\sum_{i=1}^{N} \frac{\partial}{\partial \theta} f(z(\theta)) \\ &= \frac{1}{2}\sum_{i=1}^{N} -x_i \cdot 2(y_i - \theta^T x_i) \\ &= \sum_{i=1}^{N} -x_i(y_i - \theta^T x_i) \\ &= -\sum_{i=1}^{N} x_i(y_i - \theta^T x_i) \\ &= -X^T(Y - X\theta) \end{aligned}$$

  - critically, whenever we see the pattern of a vector-scalar multiply within a summation, a vectorization can be performed
  - ie. the product within the summation is equal to:

$$[x_1 \dots x_N] \begin{bmatrix} y_1 - \theta^T x_1 \\ \vdots \\ y_N - \theta^T x_N \end{bmatrix} = X^T(Y - X\theta)$$

    - since $X = \begin{bmatrix} x_1^T \\ \vdots \\ x_N^T \end{bmatrix}$ is usually represented as a matrix of data rows
    - to vectorize $\theta^T x_i$, we can equivalently multiply data rows of X with the column vector θ
- does our current least-squares formula allow for learning nonlinear polynomial fits of the form:

$$y = b + a_1 x_1 + a_2 x^2 + \dots + a_n x^n$$

  - yes, we just have to redefine the input vectors:

$$\hat{x} = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^n \end{bmatrix}, \quad \theta = \begin{bmatrix} b \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

  - a higher degree polynomial will *always* fit the training data *no worse* than a lower degree polynomial
    * encapsulates lower degree polynomials and can implement them by just setting the necessary coefficients to 0
  - but do we always want the highest possible degree polynomial?
    * eg. for housing price example, the linear model performs best for new inputs
    * the fundamental problem is that more complex models may not *generalize* as well if the data came from a different model
      · ie. relying on fitting the training data instead of generalizing

Maximum Likelihood Optimization

- note that there are alternative types of optimization rather than minimizing a mean-square error for the loss function:
  - in **maximum likelihood optimization**, we want to instead *maximize* the probability of having *observed* the data
  - others eg. MAP estimation, KL divergence
    * important to arrive at an appropriate model and cost function, and then *optimize* it
  - in these examples, we could differentiate and set the derivative equal to zero
    * in more complex models, there are more general ways to learn model parameters
- ex. Given a weighted coin and a sequence of flips, want to find the coins weight θ:
  - consider the example training data of `HTHHTTHT`
  1. if $\theta = 1$, the probability of observing the data is 0
  2. if $\theta = 0.75$, the probability of observing the data is $0.75^4 0.25^4 = 0.00124$
  3. if $\theta = 0.5$, the probability of observing the data is $0.5^4 0.5^4 = 0.0039$
  - thus we would want to choose model (3) since it maximizes the likelihood of seeing the data

- ex. Given a set of N paired data $\{x_i, y_i\}$ where the coordinate $x_i \in \mathbb{R}^2$ has a class y_i belongs to one of three classes, want to be able to estimate the class of a new coordinate:
  - say that each of the classes follows a normal / Gaussian distribution ie. $x_i|y_i = j \sim N(\mu_j, \Sigma_j)$
  - the μ, Σ are the parameters θ we can choose to match the data as close as possible
  - need to make some important assumptions:
    * all classes are equally probable a priori ie. $p(y_i) = \frac{1}{3} = k$
    * each data point is *independent* given the parameters:

$$p(\{x_i, y_i\}, \{x_j, y_j\}|\theta) = p(x_i, y_i|\theta)p(x_j, y_j|\theta)$$

      · very useful assumption, prevents dealing with a long expansion of the probability chain rule
      · not completely true in reality eg. pictures data may have both cars and dogs, time series can cause dependencies
- want to maximize the likelihood of having seen the dataset:

$$\begin{aligned} L &= p(\{x_1, y_1\}, \dots, \{x_N, y_N\}|\theta) \\ &= \prod_{i=1}^{N} p(x_i, y_i|\theta) \\ log(L) &= \sum_{i=1}^{N} log(p(x_i, y_i|\theta)) \\ &= \sum_{i=1}^{N} log(p(y_i|\theta)p(x_i|y_i, \theta)) \\ &= \sum_{i=1}^{N} log(p(y_i|\theta)) + log(p(x_i|y_i, \theta)) \\ &= Nlog\frac{1}{3} + \sum_{i=1}^{N} log(p(x_i|y_i, \theta)) \end{aligned}$$

  - using a log-likelihood technique to convert a product into a sum
  - results in a function of θ, so we can optimize θ to make the log function as large as possible
- optimizing:
  - after solving $\frac{\partial logL}{\partial \mu_i} = 0$ we find that the optimal μ_i is the empirical mean of the samples
  - similarly, solving x gives that the optimal Σ_i is the sample covariance

- with a new coordinate, how do we find the class x_{new} belongs to:

$$\begin{aligned} argmax_j \ p(j|x_{new}) &= argmax_j \ \frac{p(j, x_{new})}{p(x_{new})} \\ &= argmax_j \ \frac{p(j)p(x_{new}|j)}{p(x_{new})} \\ &= argmax_j \ p(j)p(x_{new}|j) \\ &= argmax_j \ p(x_{new}|j) \end{aligned}$$

  - ie. calculating the probability of it belonging to each of the distributions
  - simplifications occur since $p(x_{new})$ does not depend on j and $p(j)$ is uniform in this example

Generalizing the Model

- dangers of overfitting / underfitting:
  - **training data** is data that is used to learn the parameters of the model
  - **validation data** is the data used to optimize the hyperparameters of the model:
    - **hyperparameters** are the design choices of the model, eg. the order of the fitted polynomial
    - avoids the potential of overfitting to nuances in the testing dataset
  - **testing data** is data that is excluded in training and used to score the model
    - a "pristine" dataset used to score the final model with set parameters and hyperparameters
  - all datasets should follow the same distributions
  - a model with very low training error but high testing error is called **overfit**:
    - beyond a certain point, model begins to overfit the data
  - addressing overfitting:
    - more data helps ameliorate the issue of overfitting
      - may be appropriate to use more complex models when given much more data
    - regularization is another useful technique
- picking a best model:
  1. assess its generalization ie. validation error
  2. pick a setting of the parameters that results in minimal value
  - there are some scenarios where the database size is so limited that it is better to utilize model selection techniques

        - ie. penalizes the model for being overly complex
- evaluating generalization error:
    - in a common scenario, we are given a training and testing dataset
    - to train a model while validating hyperparameters, one common approach is k-fold **cross validation**:
        - split training data into k equal sets called **folds**, each with $\frac{N}{k}$ examples
        - $k-1$ folds are training datasets, while the remaining fold is a validation dataset
        - for each hyperparameter eg. polynomial order we are trying to validate
            - run k validation tests using each of the folds as a validation set, take the average as an overall validation error
        - note that class balance should be maintained across folds eg. using a stratified k-fold
    - after using cross validation to finalize hyperparameters, we can train a single model based on the entire training data

Supervised Classification

- supervised learning techniques are useful for classification, as well as some neural networks developments
  - a new renaissance in supervised learning due to the boom in computer vision applications
  - in classification, want to classify an input image x as one of several possible categories y
    - images are input into a computer vision algorithm as a $width \times height \times 3$ array representing RGB values from $[0, 255]$
- problems that arise in image classification:
  - viewpoint variation
    - cannot simply compare pixel values at certain locations
  - illumination that adjusts the RGB values
  - deformation
    - eg. different visible *features* of a cat depending on the image
  - occlusion
  - background clutter
  - intraclass variation
- different possible approaches to classifying an image:
  - have experts come in and manually craft important features of image classes
    - impossible to scale
  - instead, use a data driven approach:
    1. expose a machine learning algorithm to a lot of *data*
    2. *learns* a function mapping the image to class
    - these deep neural nets learn parameters that represent features useful for classifying the image well
      - developing their *own* features for image classes that may be difficult to interpret
    - then, test how the neural net performs on predicting the class of new images

k-Nearest Neighbors

- given a training set of input vectors $x_1, \ldots, x_m$ and their corresponding classes $y_1, \ldots, y_m$, want to estimate the class of a new data point x_{new}:
  - we previously found a way to classify through a probabilistic model where we had to learn parameters
    - is there a simpler way to classify without very much ML machin-

ery?

- in the k-**nearest neighbors** algorithm:
  - find the k closest points or neighbors in the training set according to an appropriate metric
  - each of these neighbors then vote according to what class it is in, and x_{new} is assigned to be the class with the most votes
  - k and the distance metric are hyperparameters
    - can test hyperparameters to find the settings with lowest errors and lower variances
  - an example of **instance-based classification** where similar instances will have similar classification
- k-nearest neighbors more formally:
  1. choose an appropriate distance metric $d(x, y)$ ie. just Euclidian distance or $d(x, y) = ||x - y||_2$
  2. choose the number of nearest neighbors k
  3. calculate $d(x_{new}, x_i) \ \forall \ i = 1, \dots, m$
  4. classify x_{new} as the class that occurs most frequently among the corresponding classes of the k closest neighbors
- how do we train the classifier?
  - just have to cache the entire dataset so it can run the algorithm on testing data
  - *pros*:
    - fast in $O(1)$ and simple
      - no need to copy, can just use pointers or references to point to data
  - *cons*:
    - memory intensive since we have to store all of the training data
    - memory scales with the number of training examples
      - vs. the MLE algorithm, which needs to store a constant number of parameters regardless of training examples
- how do we test a new data point?
  - have to calculate the distances from every point in the training set, and sort them
  - *pros*:
    - simple
  - *cons*:
    - takes a long time, scales with the amount of data given in $O(N)$
  - want the *reversed* complexity times for training and testing:
    - testing should be $O(1)$, while it is more acceptable for training to be slower
    - eg. in computer vison, want instantaneous results
- why might this algorithm fail for image classification?
  - calculating distance between the inputs doesn't have a semantic mean-

ing correlating to image similarity:
    * eg. shifted vs. tinted vs. images overlaid with boxes would have similar distances from the original although the boxed images may look very different than the tinted or shifted versions
  - the *"curse of dimensionality"*:
    * the number of dimensions starts to scale exponentiality given larger, more complex input data
    * as feature space gets larger, the feature vectors become sparser ie. *farther* apart
      · the notion of similarity thus begins to break down in higher dimensions
    * ie. distances in different dimensions may start to mean different things other than similarities

Softmax Classifier

- a better approach may be to develop a score for an image coming from each class and then select the class with the largest score:
  - based on **linear classification**, which consists of two major components:
    1. a *linear* score function that maps the raw data to class scores
    2. a loss function that measures how good the scoring functions is at predicting the labels
  - linear classifiers are a building block for neural networks
    * each layer of a neural network is a linear classifier that is then passed through nonlinearity
- consider a matrix $W \in \mathbb{r}^{c \times N}$ where $W = \begin{bmatrix} w_1^T \\ \vdots \\ w_c^T \end{bmatrix}$:
  - with c number of classes
  - $y = Wx + b$ is a vector of scores where its ith element corresponds to the score of x being in class i:

$$y = \begin{bmatrix} W_1^T x + b_1 \\ \vdots \\ W_c^T x + b_c \end{bmatrix}$$

    * b is a vector of bias terms
      · $b, y \in \mathbb{R}^{10}$ in CIFAR
    * x has dimensions $x \in \mathbb{R}^{3072}$ in CIFAR
  - the output will be the index of the highest score in y
    * ie. $a_i(x)$ is the ith entry of y
  - note that due to the dot product, when w_i^T is *similar* to x, the score will be higher

    - thus each w_i, when recomposed as in image, acts as a template for the *average* image in that class
- so what is a linear classifier actually doing?

$$\begin{aligned} w_i^T x &= ||w_i||\ \ ||x|| cos\theta \\ &= ||x|| cos\theta \end{aligned}$$

  - assuming $||w_i|| = 1$
  - in 2D, any point x that lies on the same line *perpendicular* to w_i has the *same score*:
    - since $||x|| cos\theta = ||x|| \frac{||adjacent||}{||hypotenuse||} = ||adjacent||$
    - ie. taking x and calculating its *projection* onto every weight vector
    - the *intersection* of these perpendicular lines indicates a **linear decision boundary** between different weights
  - eg. in binary classification in 2 dimensions, the boundary is the single line on the $\mathbb{R}^2$ plane that divides up the points into two classes
  - eg. in higher dimensions and multi-class classification, the boundaries become defined by *multiple* hyperplanes:
    - either side of each hyper plane can be interpreted as whether one of two possible classes is more likely
  - ie. linear classifiers break up space into regions bounded by hyperplanes
- where might linear classifiers fail?
  - when data is not linearly separable
    - eg. the XOR problem
  - *however*, can sometimes use tricks using change of bases:
    - eg. *radial* data can be expressed in polar to become linearly separable
    - this is a foreshadowing of what neural networks do
      - ie. find features that make the data linearly separable themselves
- collecting the received scores into a loss function:
  - we can use the **softmax** function to transform the scores into a probability

$$softmax_i(x) = \frac{e^{a_i(x)}}{\sum_{j=1}^{c} e^{a_j(x)}}$$

    - ie. normalizes the scores to probabilities while handling negative or very large scores
      - thus all the softmax probabilities will add up to 100%
    - note that the choice of $softmax$ for loss gives a smoother curve that is much easier to optimize compared to $argmax$ or $distance$
  - $softmax_i(x)$ can be interpreted as the probability x belongs to class i:

$$Pr(y_j = i | x_j, \theta) = softmax_i(x_j)$$

    - where $\theta = \{w_j, b_j\} \ \forall \ j \in \{1, ..., c\}$
  - optimizing softmax loss function ie. the **cross-entropy loss** with respect to θ:

$$p(x_1, ..., x_m, y_1, ..., y_m|\theta) = \prod_{i=1}^{m} p(x_i, y_i|\theta) = \prod_{i=1}^{m} p(x_i|\theta)p(y_i|x_i, \theta)$$

$$\begin{aligned} argmax_\theta \prod_{i=1}^{m} p(x_i|\theta)p(y_i|x_i, \theta) &= argmax_\theta \prod_{i=1}^{m} p(y_i|x_i, \theta) \\ &= argmax_\theta \sum_{i=1}^{m} log(softmax_{y_i}(x_i)) \\ &= argmax_\theta \sum_{i=1}^{m} log[\frac{e^{a_{y_i}(x_i)}}{\sum_j e^{a_j(x_i)}}] \\ &= argmax_\theta \sum_{i=1}^{m} [a_{y_i}(x_i) - log(\sum_{j=1}^{c} e^{a_j(x_i)})] \\ &= argmin_\theta \sum_{i=1}^{m} [log(\sum_{j=1}^{c} e^{a_j(x_i)}) - a_{y_i}(x_i)] \end{aligned}$$

  - note that $p(x_i|\theta)$ is *independent* of θ ie. not dependent of chosen parameters, so it can be taken out of the $argmax_\theta$
  - in addition, $argmax_\theta f(\theta) = argmin_\theta \ -f(\theta)$
- intuition behind the name of the softmax classifier:
  - the output of the softmax can be interpreted as the probability of a class and is typically considered with log ie. the **log likelihood**

$$\begin{aligned} log(Pr(y = i|x)) &= log(softmax_i(x)) \\ &= a_i(x) - log(\sum_{j=1}^{c} e^{a_j(x)}) \end{aligned}$$

  - the latter term can be approximated by $max_j a_j(x)$ since the biggest a_j dominates
  - if $a_i(x)$ produces the largest score, then the log likelihood is approximately 0
  - if $a_j(x)$ produces the largest score for $j \neq i$, then $a_i(x) - a_j(x)$ is negative, and the log likelihood is negative
  - ie. in cross-entropy, want to minimize the *negative* log likelihood of the correct class
- a potential problem when implementing a softmax classifier is overflow:
  - if $a_i(x) \gg 0$, then $e^{a_i(x)}$ may overflow

  - thus it is standard practice to normalize the softmax function as follows:

$$\tilde{a}_i(x) = a_i(x) + logk = \frac{ke^{a_i(x)}}{k\sum_j e^{a_j(x)}} = \frac{e^{a_i(x)+logk}}{\sum_j e^{a_j(x)+logk}}$$

    * we usually set $logk = -max_i\ a_i(x)$, which makes the maximal argument of the exponent 0

- softmax **temperature** is a scaling constant T used for tuning the softmax classifier:

$$softmax_i(x) = \frac{e^{\frac{a_i(x)}{T}}}{\sum_{j=1}^{c} e^{\frac{a_j(x)}{T}}}$$

  - affects the distribution of softmax probabilities
    * change in T is analagous to a change in base
  - as $T \to \infty$, $softmax_i(x) \to \frac{1}{c}$ ie. approaches a uniform distribution
  - as $T \to 0$, the max scores gets close to 1, and all others go to 0
  - temperature can be used to perform **knowledge distillation** that reduces the number of parameters used ina classifier:
    * uses a teacher model that outputs a softmax probability distribution to a fresh student model
    * here, temperature adjustment is important to improve efficiency of the distillation

Support Vector Machine

- another common decision boundary classifier is the **support vector machine (SVM)**:
  - the SVM finds a boundary that maximizes the margin or *gap* between the boundary and the data points
  - if a point is *further* away from the decision boundary, there ought to be greater *confidence* in classifying that point
- informally, to calculate the loss of a chosen boundary:
  - points very close to the boundary should incur small losses, even if they are correctly classified:
    * while different classifiers would not penalize these points at all
    * encourages the model to find a boundary with a large margin
  - misclassified data points that are incorrect should have a very large loss though they may be close to the boundary
  - points past a certain margin of the boundary should incur no loss
- the **hinge loss** function:
  - standardly defined for a binary output $y \in \{-1, 1\}$

  - when $y_i = 1$, want $w^T x_i + b$ to be large and positive, while when $y_i = -1$, want $w^T x_i + b$ to be large and negative

$$hinge_{y_i}(x_i) = max(0, 1 - y_i(w^T x_i + b))$$

    * when $y_i = 1$ and $w^T x_i + b \gg 1$, the hinge loss is 0
      · zero error if signs match and there is a large margin
    * when $y_i = 1$ and $w^T x_i + b = 0.3$, the hinge loss is 0.7
      · nonzero error if signs match, but there is a small margin
    * when $y_i = 1$ and $w^T x_i + b = -1$, the hinge loss is 2
      · larger error when signs do not match
    * here, 1 acts as the margin value
      · note that we can set it as 1 without a loss of generality, since a change in the margin could be compensated by changing the weights
- hinge loss extension to multiple classes:

$$hinge_{y_i}(x_i) = \sum_{j \neq y_i} max(0, 1 + a_j(x_i) - a_{y_i}(x_i))$$

  - given c classes where 1 is correct and $c - 1$ are incorrect, and $a_j(x_i) = w_j^T x_i + b_j$
  - when the correct class achieves the highest score:

$$a_{y_i}(x_i) \geq a_j(x_i), \;\; 0 \leq hinge_{y_i}(x_i) \leq c - 1$$

  - when the correct class is much higher than the other scores:

$$a_{y_i}(x_i) \gg a_j(x_i) + 1, \;\; hinge_{y_i}(x_i) = 0$$

  - when the correct class achieves an equal score:

$$a_{y_i}(x_i) = a_j(x_i), \;\; hinge_{y_i}(x_i) = c - 1$$

  - when an incorrect class achieves the highest score:

$$a_{y_i}(x_i) < a_j(x_i), \;\; a_j(x_i) - a_{y_i}(x_i) \geq 0$$

    * has the potential to be large
  - in general, the model is encouraged to make correct margins larger and incorrect margins smaller
- softmax vs. SVM intuition:
  - the softmax is a maximum likelihood loss function:
    * change the parameters to optimize having seen the data
    * cross-entropy is the most common loss function typically used

  - the hinge loss is a human-constructed heuristic:
    * bound the loss at zero, and calculate a margin from a difference of scores
    * the margin in the SVM may help more with noisy data and outliers
    * thus SVM could be more useful empirically in some scenarios
- optimizing the SVM cost function:

$$argmin_\theta \ \frac{1}{m}\sum_{i=1}^{m} hinge_{y_i}(x_i)$$

$$argmin_\theta \ \frac{1}{m}\sum_{i=1}^{m}\sum_{j\neq y_i} max(0, 1 + a_j(x_i) - a_{y_i}(x_i))$$

  - where $a_j(x_i) = W_j^T x_i + b$, $\theta = \{W, b\}$, $W = \begin{bmatrix} W_1^T \\ \vdots \\ W_c^T \end{bmatrix}$

Gradient Descent

- from calculus, the derivative of a function tells us its slope at a point:

$$f(x + \epsilon) \approx f(x) + zf'(x)$$

  - when the derivative is 0, we are at a stationary or critical point of the function:
    * may be a local or gobal maximimum or minimum, or a saddle point
    * however, when f contains nonlinear or non-differentiable functions, cannot simply set the derivative equal to 0
    * instead want to iteratively approach a critical point via **gradient descent**
  - in this class, need to optimize f with respect to vectors and matrices
- terminology:
  - a **global minimum** x_g achieves the absolute lowest values of f, ie. $f(x) \geq f(x_g) \ \forall \ x$
  - a **local minimum** x_l is a critical point that is lower than its neighboring points, however, $f(x_l) > f(x_g)$
    * analagous definitions for maximums
  - a **saddle point** is a critical point that is not a not local maximum or minimum
    * local neighbors are larger and smaller on either side
- the gradient is a vector that tells us how small changes in Δx affects $f(x)$ through:

$$f(x + \Delta x) \approx f(x) + \Delta x^T \nabla_x f(x)$$

  - to find how $f(x)$ changes in some direction of a unit vector u:

$$u^T \nabla_x f(x)$$

- to minimize $f(x)$, want to find the direction in which $f(x)$ decreases the fastest:

$$\begin{aligned} min_{u,||u||=1} u^T \nabla_x f(x) &= min_{u,||u||=1} ||u||\ ||\nabla_x f(x)||\ cos(\theta) \\ &= min_u\ ||\nabla_x f(x)||\ cos(\theta) \end{aligned}$$

  - this quantity is minimzed for u pointing in the opposite direction of the gradient such that $cos(\pi) = -1$
  - thus to update x as to minimize $f(x)$, we repeatedly calculate:

$$x := x - \epsilon \nabla_x f(x)$$

    * ϵ is known as the **learning rate**, and may change over iterations
- how to pick the right step size ie. learning rate:
  - when the step size is too large, the linear approximation of the gradient will fail and the descent may keep overshooting its target
  - if the step size is smaller, the linear approximation of the gradient should hold
    * but with too small a step size, computation time greatly increases
  - with more and more dimensions, should we step in every direction at the same rate?
    * in a **first-order method**, we take steps in every direction at equal rates
    * in a **second-order method**, we compute the curvature of the surface in every direction to calculate how much to step in each direction
      · but this is prohibitively expensive to calculate
    * first-order methods with heuristics may be a better alternative
  - can empirically test learning rate by examining the cost function at each iteration:
    * want the loss to quickly and smoothly decrease
    * abnormalities or a plateau may indicate too high a step size
    * a slowly decreasing loss may indicate too small a step size
- why not instead use a numerical gradient as follows:

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

  - with millions of parameters represented by h, we would have to calculate the loss at many settings of h
    * extremely slow

Hinge Loss Gradient

- want to find the gradient for the hinge loss:

$$\begin{aligned} L(\theta) &= \frac{1}{m}\sum_{i=1}^{m}\sum_{j\neq y_i} max(0, 1 + w_j^T x_i - w_{y_i}^T x_i) \\ L_i &= \sum_{j\neq y_i} max(0, 1 + w_j^T x_i - w_{y_i}^T x_i) \\ &= \sum_{j\neq y_i} max(0, z_j) \end{aligned}$$

$$\begin{aligned} \nabla_{W_j} L_i &= \begin{cases} 0 & z_j \leq 0 \\ x_i & z_j > 0 \end{cases} = \mathbb{1}(z_j > 0)x_i \\ \nabla_{W_{y_i}} L_i &= -\sum_{j\neq y_i} \mathbb{1}(z_j > 0)x_i \end{aligned}$$

  - where $W = \begin{bmatrix} W_1^T \\ \vdots \\ W_c^T \end{bmatrix}$
  - notes:
    - the biases are dropped for simplicity
    - the gradient can be applied on the inside of the averaging term since the gradient is a linear operator
      - ie. the gradient is taken over every training example and then averaged
    - manually setting the derivative at 0 to 0 using the concept of sub-gradients, though it is technically undefined
    - the indicator function $\mathbb{1}$ returns 0 if its argument ≤ 0 and 1 otherwise
- in the earlier illustrated gradient descent, we know the function f exaclty and can calculate the gradient at that point exactly:
  - in optimization, we instead differentiate the cost function with respect to the parameters
    - thus this gradient is a *function* of the training data
  - ie. each data point provides a noisy estimate of the gradient at that point
  - however, it's expensive to calculate the gradient with *every* example in the set
- instead, there are alternatives approaches to calculating the gradient:
  - in a batch algorithm, use all m examples in the training set to calculate the gardient
  - in a minibatch algorithm, approximate the gradient by calculating it with k examples where $m > k > 1$

        - typically used in deep learning
    - in a stochastic algorithm, approxmiate the gradient by calculating it over a single example
    - the smaller the batch size, the more steps we can take in the same amount of calculation time:
        - though more noise may be introduced into the gradient estimation
        - however, more noise may be beneficial in acting as regularization
            - ie. generalizing the model better to avoid overfitting
- to find the gradient on the softmax loss, take the chain rule with the gradient of the softmax function itself

Neural Networks

- the inspiration from **neural networks** is from neural science:
  - neurons are the main signaling units of the nervous system, with 3 main parts:
    1. the dendrites are a tree like structure that receive input signals
       - each dendrite may have a synaptic weight associated with it
    2. the axon hillock is an *integration center* for summing propagated input signals
       - an *all-or-nothing* rather than analog spike for triggering some action potential
    3. axons are long tubular structures for carrying output signals
       - connects to other downstream neurons
  - in the brain, spiking responses are probabilistic:
    - exact spikes are different even for the same trials
    - instead, we can track of the *rate* at which spikes occur per second
    - this rate is what neural networks attempt to encode
- the neural network neuron vs. real neuron:
  - receives various inputs $x_1, \dots, x_N$ that act as dendrites, each with a unique fixed weight $w_1, \dots, w_N$
    - each input may be an output of prior neurons
  -
  - has a summation computation that sums up the "dendritic-processed" signals $w_i x_i$ and a bias:

$$f(\sum_i w_i x_i + b)$$

    - performs a sum and passes it through a nonlinearity f
    - like the spike threshold, there is an aspect of nonlinearity for the neuron to fire
- some differences between artificial and real neurons:
  - synaptic transmission are probabilistic, nonlinear, and *dynamic*
  - dendritic integration is probabilistic and may be nonlinear
  - there are many different cell and neuron types
  - in general, though neural networks are inspired by biology, they approxmiate biological computation at a fairly crude level
- nomenclature:
  - the first layer is an **input layer** typically represented with the variable x
  - the last layer is the **output layer** typically represented with the variable z

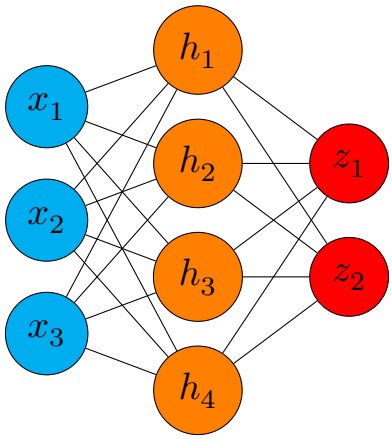

Figure 1: 2-Layer Neural Network

    - typically the output of the entire neural network are a processed version of z eg. sent through softmax as $softmax(z)$
  - intermediate layers are known as the **hidden layers** represented by the variable h:
    - generally, these layers are extracting some kind of features from the input data
    - they are able to learn which features make the data linearly separable so that the processing softmax classifier can classify them
    - importantly, these features do not have to be handcrafted
  - when we specify a network has N layers, this does not include the input layer

- in the 2-layer network from Figure 1:
  - layers are the input, hidden, and output layer
  - $x \in \mathbb{R}^3$ inputs are processed into a four dimensional intermediate representation $h \in \mathbb{R}^4$
    - this representation is then transformed into a two dimensional output $z \in \mathbb{R}^2$
  - can express the outputs of each layer:

$$\begin{bmatrix} h_1 \\ \vdots \\ h_4 \end{bmatrix} = f(\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ \vdots & \vdots & \vdots \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b_1)$$

$$h = f(W_1 x + b_1)$$

$$z = W_2 h + b_2$$

    - applying f to a vector applies the function elementwise
  - $W_1 \in \mathbb{R}^{4\times3}$, $W_2 \in \mathbb{R}^{2\times4}$, $b_1 \in \mathbb{R}^4$, and $b_2 \in \mathbb{R}^2$
  - network has 6 neurons not counting the input, 20 weights, and 6 biases
    - for a total of 26 learnable parameters

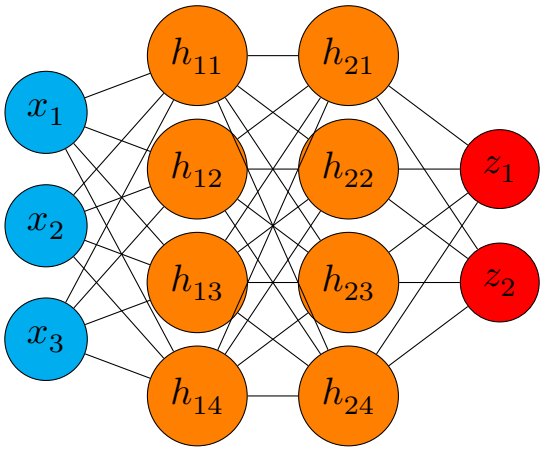

Figure 2: 3-Layer Neural Network

  - convolutional neural networks typically have on the order of hundreds of millions of parameters
    * with more hidden layers, can potentially extract even more features and make the data more linearly separable

- in the 3-layer network from Figure 2:
  - we now have two hidden layers h_1, h_2 and three sets of weights w_1, w_2, w_3
  - this network is a **fully connected network (FC network)**
    * alternatively a **multi-layer perceptron (MLP)** or a **feed-forward network**
  - outputs at layers:

$$\begin{aligned} h_1 &= f(W_1 x + b_1) \\ h_2 &= f(W_2 x + b_2) \\ z &= W_3 h_2 + b_3 \end{aligned}$$

  - network has 10 neurons, 36 weight, and 10 biases

Defining the 3-layer network in Python:

```
f = lambda x: x * (x > 0)
h1 = f(np.dot(W1, x) + b1)
h2 = f(np.dot(W2, h1) + b2)
z  = np.dot(W3, h2) + b3
```

Nonlinear Activation Functions

- what if we set f to just be the identity function ie. another linear operation:
  - then, each layer can be composed linearly of the previous one eg. $h_2 = W_2(W_1 x + b_1) + b_2$

    - but in this, case we simply have yet another linear mapping $h_2 = \tilde{W}x + \tilde{b}$ where $\tilde{W} = W_2W_1$ and $\tilde{b} = W_2b_1 + b_2$
  - ie. any composition of linear functions can be reduced to a single linear function:

$$\tilde{W} = W_N ... W_2W_1$$
$$\tilde{b} = b_N + W_Nb_{N-1} + ... + W_N ... W_2b_1$$

  - this severely limits the computations we can perform, so we should not set f to be linear
    - eg. cannot solve the XOR problem
  - however, this may be useful in some contexts:
    - eg. when $dim(h) \ll dim(x)$, this corresponds to finding a low-rank representation of the inputs
    - ie. performing **dimensionality reduction** to compress the features of the input into fewer ones
- instead we want to introduce nonlinearity to increase the network capacity:
  - introduce the nonlinearity f at the output of each artifical neuron
  - f is also called the **activation function**
  - f is not typically applied on the output layer z:
    - z can be interpreted as scores that a softmax or SVM classifier will use to classify the input data
    - eg. the final hidden layer output h_{N-1} acts as the input vector into $softmax(z')$ where $z' = z = W_Nh_{N-1} + b_N$
  - "one recurring theme through neural network design is that the gradient of the cost function ($\frac{\partial L}{\partial W}$) to be large and predictable enough to serve as a good guide for the learning algorithm" - Goodfellow et al.
    - important consideration when choosing nonlinears ie. activation functions

- **sigmoid** activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

  - *pros*:
    - around $x = 0$, the unit behaves linearly
    - is differentiable everywhere
  - *cons*:
    - at extremes, the unit *saturates* and thus has zero gradient:
      - leads to slower or no learning in these areas

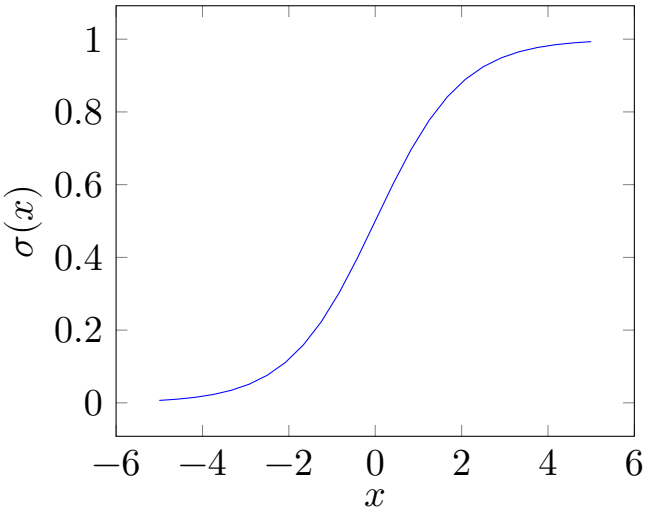

Figure 3: Sigmoid Function

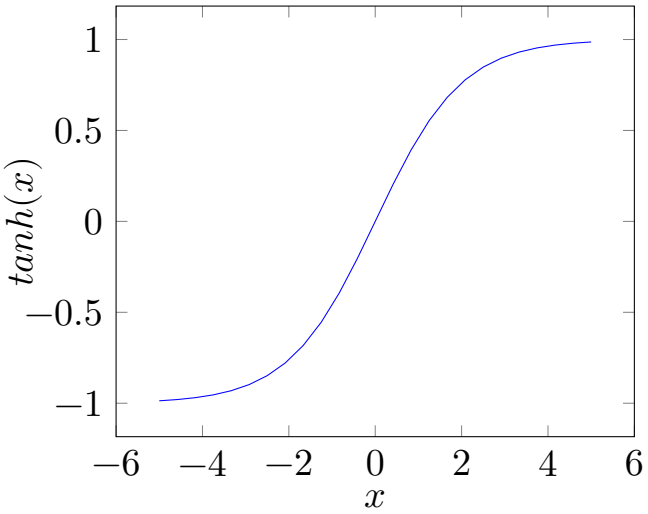

Figure 4: Hyperbolic Tangent Function

  - even if you scale ϵ to be larger, the gradient is still near zero or is not as informative for meaningful learning
* the sigmoid is centered around 0.5 and not zero-centered, causing the **zig-zagging problem** during gradient descent:
  - occurs because the sigmoid is always non-negative
  - the elements of the gradient are all ≥ 0, so $\frac{\partial L}{\partial W}$ will have either *all* entries positive or negative causing the descent to go back and forth
  - in practice, zig-zagging not very much of a problem

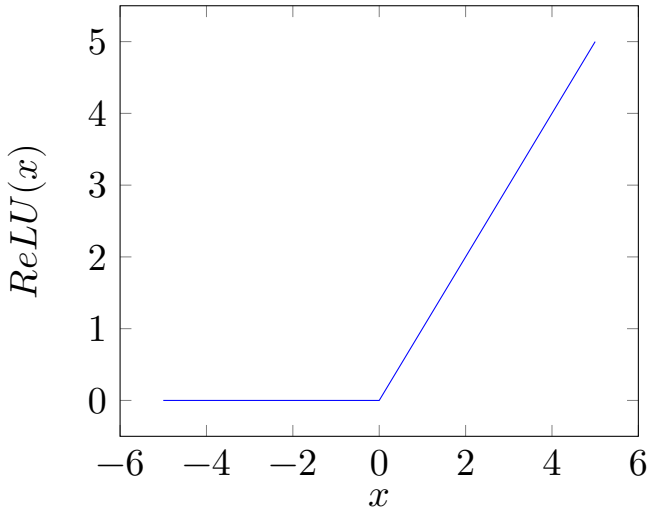

Figure 5: ReLU Function

- **hyperbolic tangent** activation function:

$$tanh(x) = 2\sigma(x) - 1$$

$$\frac{dtanh(x)}{dx} = 1 - tanh^2(x)$$

  - essentially a zero-centered sigmoid
  - *pros*:
    - around $x = 0$, the unit behaves linearly
    - is differentiable everywhere
    - is zero-centered
  - *cons*:
    - at extremes, also saturates

- **rectified linear unit (ReLU)** activation function:

$$ReLU(x) = max(0, x)$$

$$\frac{dReLU(x)}{dx} = \mathbb{1}(x)$$

  - though this function is nonlinear, it is essentially a piecewise linear function, so is this enough to give modeling capacity?
    - yes, is *still* a nonlinearity
  - ReLU is not differentiable at $x = 0$, but we can set its subgradient there to be either the left or right gradient 0 or 1
  - *pros*:
    - in practice, learning with ReLU converges faster than sigmoid and $tanh$

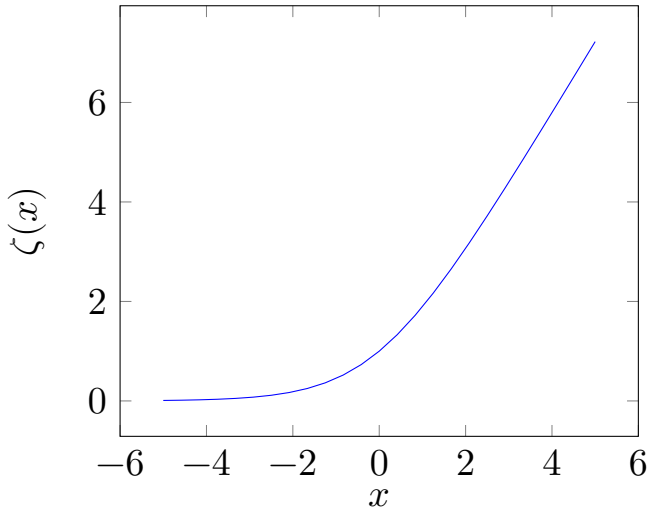

Figure 6: Softplus Function

    - derivative is always 0 or 1
    - there is no saturation if $x > 0$
  - *cons*:
    - not zero-centered and zigzags
    - not differentiable at x = 0
      - in practice, the subgradient estimation here is reasonable given digital computation is already subject to numerical error
    - learning does not happen for examples with zero activation
      - can be fixed by using a leaky ReLU or a maxout unit

- **softplus** activation function:

$$\zeta(x) = log(1 + e^x)$$

$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

  - intuitively, softplus resembles ReLU and is differentiable everywhere
  - however, empirically, performs worse than ReLU

- **leaky ReLU** activation function:

$$f(x) = max(\alpha x, x)$$

  - *pros*:
    - leaky ReLU avoids stopping of learning when $x < 0$
  - *cons*:
    - additional parameter α
  - α can be treated as a selected hyperparameter, or even another optimizable parameter in PReLU
  - leaky RELU allows for negative valeus:

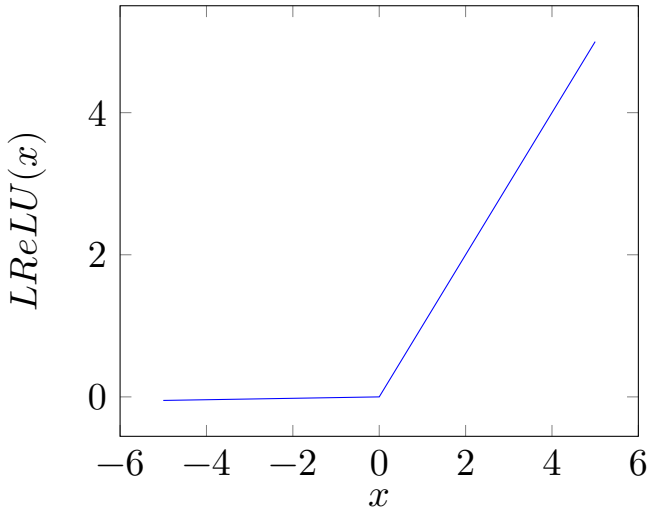

Figure 7: Leaky ReLU Function

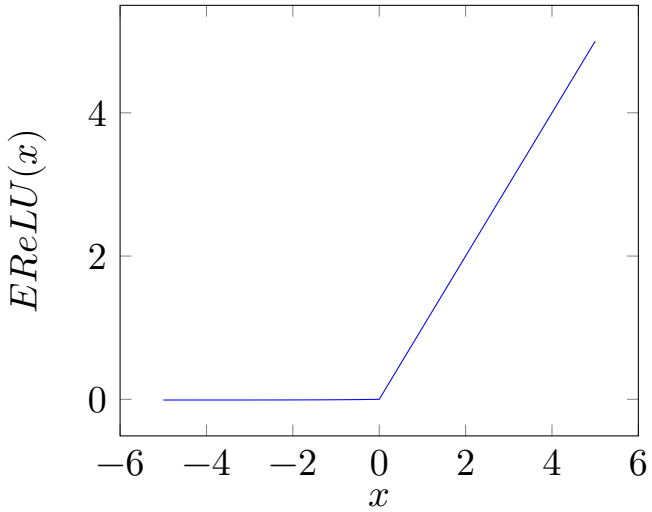

Figure 8: Exponential ReLU Function

    - the activation functions encode the probability neuron will fire
        - what does a negative rate intuitively mean?
    - want best efficiency, so depart from biological analogy
    - zero doesn't allow for learning, while a negative value still does

- **exponential linear unit** activation function:

$$f(x) = max(\alpha(e^x - 1), x)$$

    - *pros*:
        - again avoids stopping of learning when $x < 0$
    - *cons*:
        - requires more expensive computation of the exponential
- **maxout unit** activation function:

$$maxout(x) = max(W_1^T x + b_1, W_2^T x + b_2)$$

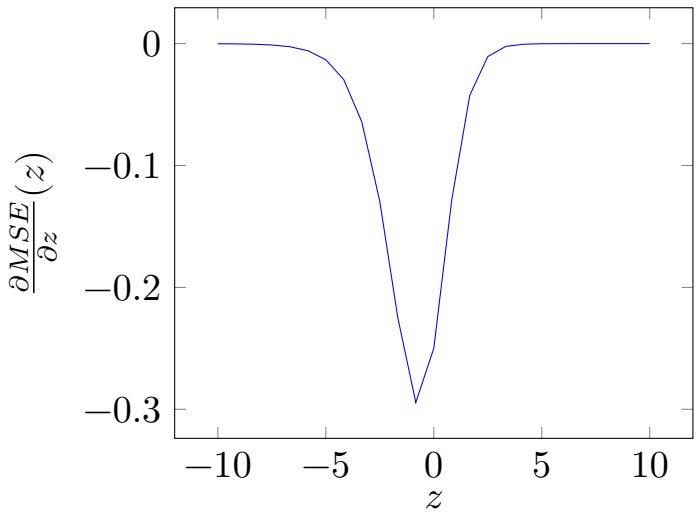

Figure 9: MSE Deriviative wrt. z_i with $y_i = 1$

  - is a generalization of ReLU and PReLU
  - *cons*:
    - doubles the number of parameters
- choosing an activation function in practice:
  - ReLU is very popular
  - sigmoid is almost never used since $tanh$ is preferred
  - leaky ReLU, PReLU, ELU, and maxout may be worth trying out for different applications

Output Activations

- there are different ways to process the output scores z to arrive at a cost function:
  - a linear output unit $\hat{y} = z$:
    - these units typically specify the conditional mean of a Gaussian distribution ie. $p(y|z) = \mathcal{N}(z, I)$
    - in this case, the MLE is equivalent to MSE, rather than CE
  - a sigmoid output $\hat{y} = \sigma(z)$
    - typically used in binary classification to approximate a Bernoulli distribution
  - a softmax output activation unit where $\hat{y}_i = softmax_i(z)$:
    - softmax is the generalization of the sigmoid to multiple classes
    - this the most common output activation
- ex. Consider a binary classification that outputs a single score z with the sigmoid chosen as the output unit:
  - thus $\hat{y}_i = \sigma(z_i)$ for a training example i

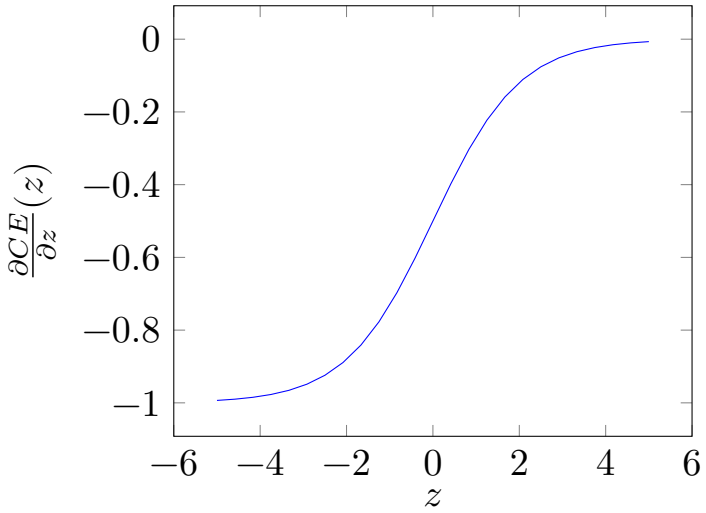

Figure 10: CE Deriviative wrt. z_i with $y_i = 1$

  - the sigmoid function returns a score between 0 and 1 for each input x_i representing the probability x_i is in class 1
- two considerations for the cost function are mean-square error or cross-entropy (ie. MLE) specifically for binary classification:

$$MSE = \frac{1}{2}\sum_{i=1}^{n}(y_i - \sigma(z_i))^2$$

$$CE = -\sum_{i=1}^{n}[y_i log(\sigma(z_i)) + (1 - y_i)log(1 - \sigma(z_i))]$$

- a large positive z_i should indicate the data belongs to class 1, and thus $\hat{y}_i = 1$
  - while a large negative z_i should indicate the data belongs to class 0, and thus $\hat{y}_i = 0$
- consider using MSE for binary classification with some $x_i, y_i = 1$ and $z_i = -50$:

$$\begin{aligned}\frac{\partial MSE}{\partial z_i} &= \frac{\partial MSE}{\partial \sigma(z_i)} \cdot \frac{\partial \sigma(z_i)}{\partial z_i} \\ &= -2(y_i - \sigma(z_i))(\sigma(z_i)(1 - \sigma(z_i))) \\ &\approx 0\end{aligned}$$

  - since $\frac{\partial \sigma(z_i)}{\partial z_i} \approx 0$ for this z_i from the graph of sigmoid, we can intuitively guess that $\frac{\partial MSE}{\partial z_i}$ will be close to 0
  - in fact, we can see from the Figure 9 that when z is very negative ie. the classification is strongly incorrect, the derivative ie. the gradient saturates to 0
    - thus no learning occurs in this region!

    - however, this is where we would like *much* learning to occur ie. want the derivative to be significant to enough to move towards the right classification
        - on the other hand, it is acceptable that the gradient saturates to 0 when z is very positive since in this case the classification is correct
    - note that the term y_i appears in the derivative, which flips the graph when $y_i = 0$ so that the same issue appears for the other class
  - now consider using CE for the same binary classification with x_i, y_i, z_i:

$$\begin{aligned}\frac{\partial CE}{\partial z_i} &= \sigma(z_i) - 1 \\ &\approx -1\end{aligned}$$

    - in this case, as seen in Figure 10, when z is very negative, learning will still occur
        - in fact the derivative is largest ie. we have the steepest change in this region
    - learning only begins to *stall* ie. take smaller gradient steps once z approaches the right answer
    - note that this calculation is specific to $c = 2$ and $y_i = 1$
        - when $y_i = 0$, the graph would similarly flip so that learning again correctly stalls when z approaches the desired answer
  - thus we should *always* use cross-entropy loss when using softmax AKA sigmoid as the output unit
    - note that MSE may still be appropriate for other problems eg. regressions

Backpropagation

- now that the architecture of a neural network has been defined, including activation and output functions, how do we learn its parameters?
  - by using versions of gradient descent
  - however, these networks have several layers:
    - the parameters in the earliest layers are far removed from the loss function
    - need to use a technique called backpropagation to calculate the gradient of the loss function with respect to parameters
- **backpropagation** is an application of the chain rule for derivatives:
  - in a neural network, the weights in the earlier layers are connected to the loss function through a composition of functions:
    - ie. $h_N = f(h_{N-1}) = f(f(h_{N-2})) = ...$

    - thus computing the gradient should involve repeated applications of the chain rule
  - in **foward propagation**, we calculate the values of the hidden and output units of a neural network given an input:
    - take input x, and propagate it through each hidden unit sequentially until we get output y
    - also gives the cost function $J(\theta)$
    - the foward propagated signals are the *activations*
  - in backpropagation AKA backprop:
    - information is passed *backwards* from the cost function and outputs to inputs
    - the backpropagated signals are the *gradients*
    - enables the calculation of gradients at every stage going back to the input layer
    - eg. want to calculate $\frac{\partial L}{\partial W_1}$ from $\frac{\partial L}{\partial W_2}$
    - note that the backprop operations requires knowing the foward propagated activation values
      - ie. perform a forward pass into a backward pass
    - can consider the const function as a **computation graph** ie. a directed acyclic graph where each node in the graph denotes a mathematical operation
  - compared to analytical gradients:
    - evaluating analytical gradients may be computationally expensive
    - backprop is generalizable ie. modular and often inexpensive
      - eg. libraries such as Tensorflow and PyTorch can take gradients of arbitrary functions using a generalized, mechanized backprop algorithm
    - want to share repeated computations whenever possible
  - note that backprop is *not* the learning algorithm, it is only the method of computing gradients
    - the learning algorithm used with backprop is stochastic gradient descent
  - in addition, backprop is not specific to NNs, but is a general way to compute function derivatives

- from Figure 11, we can calculate backprop derivatives (in red) as follows,

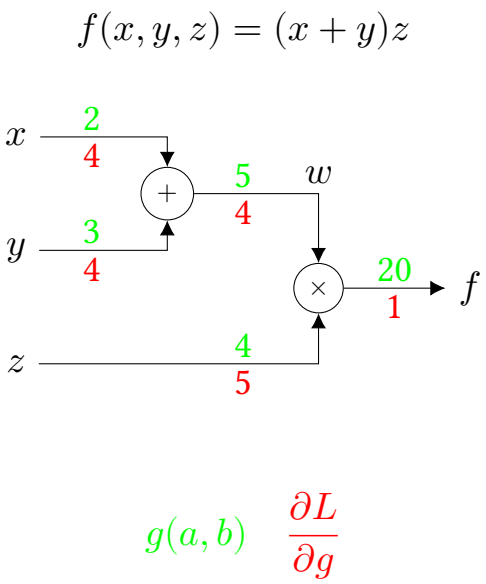

Figure 11: Simple Backprop Example

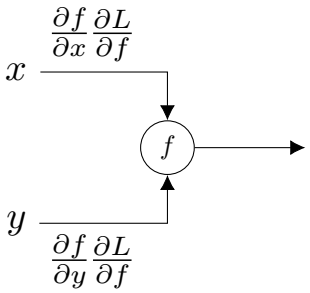

Figure 12: Backward Pass Step

where we let $L = f$ and $\frac{\partial L}{\partial f} = 1$:

$$
\begin{aligned}
f &= w \cdot z \\
\frac{\partial f}{\partial z} &= w, \quad \frac{\partial f}{\partial w} = z \\
w &= x + y \\
\frac{\partial w}{\partial x} &= 1, \quad \frac{\partial w}{\partial y} = 1 \\
\frac{\partial L}{\partial z} &= \frac{\partial f}{\partial z}\frac{\partial L}{\partial f} = w \cdot 1 = 5 \\
\frac{\partial L}{\partial w} &= \frac{\partial f}{\partial w}\frac{\partial L}{\partial f} = z \cdot 1 = 4 \\
\frac{\partial L}{\partial x} &= \frac{\partial w}{\partial x}\frac{\partial L}{\partial w} = 1 \cdot 4 = 4, \quad \frac{\partial L}{\partial y} = \frac{\partial L}{\partial x} = 4
\end{aligned}
$$

  - in the forward pass, we simply apply a function to the node inputs to calculate the output
  - in the backwards pass, we take the **upstream derivative** and apply a **local gradient** to calculate to backpropagated derivative
    * in Figure 12, the upstream derivative is $\frac{\partial L}{\partial f}$ and the local gradients are $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$
- the intuition of backprop:
  - break up the calculation into *small* and simple steps
  - each node is the graph represents a straightforward gradient calculation through multiplying an input with an application of the chain rule
    * composing the gradients together returns the overall gradient
  - with backprop, as long as we can break the computation into components where we know the local gradients, we can find the gradient of anything
  - note that in the foward pass, calculations are cached so that performing backprop is not as expensive as doing an analytic computation
- additionally, we can also interpret backpropagation as acting as different types of gradient *gates* depending on the function f:
  - an add gate distributes the gradient:

$$
\begin{aligned}
f &= x + y \\
\frac{\partial f}{\partial x} &= 1, \quad \frac{\partial f}{\partial y} = 1 \\
\frac{\partial L}{\partial x} &= \frac{\partial L}{\partial f}, \quad \frac{\partial L}{\partial y} = \frac{\partial L}{\partial f}
\end{aligned}
$$

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

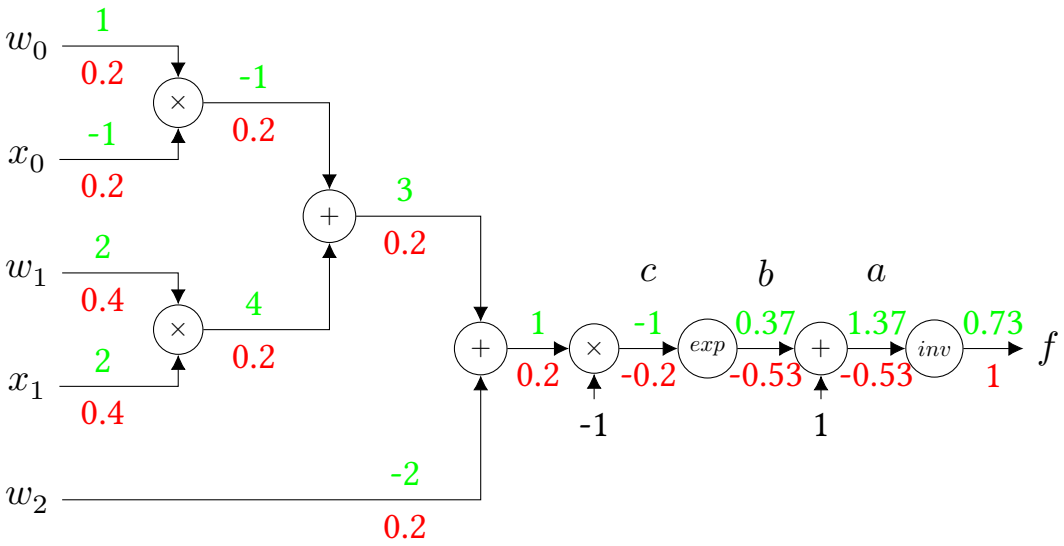

Figure 13: More Involved Scalar Example

– a *mult* gate switches the gradient:

$$f = x \times y$$
$$\frac{\partial f}{\partial x} = y, \quad \frac{\partial f}{\partial y} = x$$
$$\frac{\partial L}{\partial x} = y\frac{\partial L}{\partial f}, \quad \frac{\partial L}{\partial y} = x\frac{\partial L}{\partial f}$$

– a *max* gate routes the gradient:

$$f = max(x,y)$$
$$\frac{\partial f}{\partial x} = \mathbb{1}(x > y), \quad \frac{\partial f}{\partial y} = \mathbb{1}(y > x)$$
$$\frac{\partial L}{\partial x} = \mathbb{1}(x > y)\frac{\partial L}{\partial f}, \quad \frac{\partial L}{\partial y} = \mathbb{1}(y > x)\frac{\partial L}{\partial f}$$

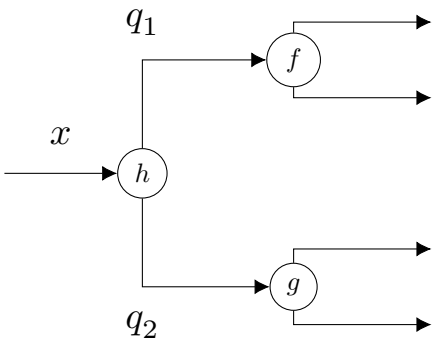

Figure 14: Converging Gradient Paths

- performing backprop on Figure 13:

$$\begin{aligned} f &= \frac{1}{a} \\ \frac{\partial f}{\partial a} &= -\frac{1}{a^2} \\ \frac{\partial L}{\partial a} &= \frac{\partial f}{\partial a}\frac{\partial L}{\partial f} = (-\frac{1}{1.37^2}) \cdot 1 = -0.53 \end{aligned}$$

$$\begin{aligned} b &= e^c \\ \frac{\partial b}{\partial c} &= e^c \\ \frac{\partial L}{\partial c} &= \frac{\partial b}{\partial c}\frac{\partial L}{\partial b} = e^{-1} \cdot -0.53 = -0.2 \end{aligned}$$

  - the rest of the calculation involves patterns with gradient gates we have already derived

- when we have converging gradient paths as in Figure 14, the calculation differs
  - by the law of total derivatives:

$$\frac{\partial L}{\partial x} = \sum_{i=1}^{n} \frac{\partial L}{\partial q_i} \cdot \frac{\partial q_i}{\partial x}$$

  - thus we have:

$$\begin{aligned} \frac{\partial L}{\partial x} &= \frac{\partial q_1}{\partial x}\frac{\partial L}{\partial q_1} + \frac{\partial q_2}{\partial x}\frac{\partial L}{\partial q_2} \\ &= \frac{\partial L}{\partial q_1} + \frac{\partial L}{\partial q_2} \end{aligned}$$

    * typically h is the identity function when gradient paths converge, so $\frac{\partial q_1}{\partial x} = 1$ and $\frac{\partial q_2}{\partial x} = 1$

$$y = softplus(x) = log(1 + e^x)$$

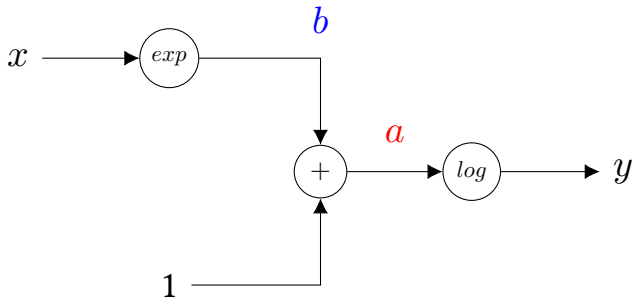

Figure 15: Backprop for Softplus

- from Figure 15, we can compute the derivative of the softplus function in two ways:
  - we can analytically find:

$$\frac{dy}{dx} = \frac{e^x}{1 + e^x}$$

  - or alternatively, we can use backprop:

$$\begin{aligned}
\frac{dy}{da} &= \frac{d}{da} log(a) = \frac{1}{a} \\
\frac{dy}{db} &= \frac{da}{db}\frac{dy}{da} = \frac{dy}{da} \\
\frac{dy}{dx} &= \frac{db}{dx}\frac{dy}{db} = e^x \frac{1}{a}
\end{aligned}$$

- to perform multivariate backpropagation, we need to use the multivariate chain rule:
  - to take the derivative of a vector with respect to a vector, construct the Jacobian J that tells us how $\Delta y \approx J\Delta x$ and $J = \nabla_x y^T$
  - eg. we will often need to take the derivative of Wx with respect to x where $W \in \mathbb{R}^{h \times n}$, $x \in \mathbb{R}^n$, and $f \in \mathbb{R}^h$:

$$\begin{aligned}
\nabla_x Wx &= \nabla_x \begin{bmatrix} w_{11}x_1 + ... + w_{1n}x_n \\ \vdots \\ w_{h1}x_1 + ... + w_{hn}x_n \end{bmatrix} \\
&= \begin{bmatrix} w_{11} \dots w_{h1} \\ \vdots \ddots \vdots \\ w_{1n} \dots w_{hn} \end{bmatrix} \\
&= W^T
\end{aligned}$$

$$h = ReLU(Wh_i + b)$$

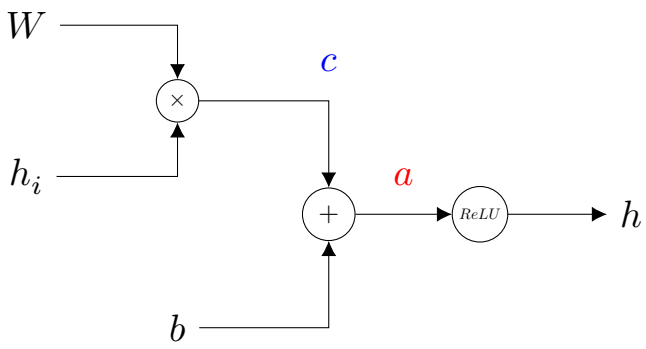

Figure 16: Backprop for a NN Layer

  * expect $\nabla_x Wx \in \mathbb{R}^{n \times h}$

- consider Figure 16, where we perform backprop on a nerual network layer that is using $ReLU$ as an activation function:
  - note that $h \in \mathbb{R}^h, h_i \in \mathbb{R}^m, W \in \mathbb{r}^{h \times m}, b \in \mathbb{R}^h$
    * h_i represents the calculation of a previous layer
  - doing backprop from h given some $\frac{\partial L}{\partial h}$ that depends on the chosen loss function:

$$\begin{aligned}
\frac{\partial L}{\partial a} &= \mathbb{1}(a > 0) \bigodot \frac{\partial L}{\partial h} \\
\frac{\partial L}{\partial c} &= \frac{\partial L}{\partial b} = \frac{\partial L}{\partial a} \\
\frac{\partial L}{\partial h_i} &= \frac{\partial c}{\partial h_i}\frac{\partial L}{\partial c} = W^T \frac{\partial L}{\partial c} \\
\frac{\partial L}{\partial W} &= \frac{\partial c}{\partial W}\frac{\partial L}{\partial c} = \frac{\partial L}{\partial c} h_i^T
\end{aligned}$$

    * notes on calculation:
      · $ReLU(x) = max(0, x)$ is just a gradient gate
      · $\frac{\partial c}{\partial h_i} = \frac{\partial}{\partial h_i} Wh_i = W^T$
      · the Hadamard product $C = A \bigodot B$ performs elementwise multiplication such that $C_{ij} = A_{ij} B_{ij}$
    * sometimes derivatives will be expressed in Python as transposes to facilitate broadcasting or faster execution
      · eg. $(\frac{\partial L}{\partial h_i})^T = \frac{\partial L}{\partial c}^T W$
  - intuition behind calculating $\frac{\partial c}{\partial W}$:
    * this is a derivative of a vector with respect to a matrix, which is a tensor derivative

- however, we can use intuitively consider matrix dimensions to find the answer without a rigorous derivation as seen in the appendix

* consider the following previous vector-matrix derivatives we have calculated:

$$\nabla_h h^T y = y$$
$$\nabla_h W h = W$$

- thus $\frac{\partial}{\partial W} W h_i$ should *look like* h_i^T

* note that $\frac{\partial L}{\partial W} \in \mathbb{R}^{h \times m}$ and $\frac{\partial L}{\partial c} \in \mathbb{R}^{h \times 1}$, and we can construct the desired matrix of dimension $(h \times m)$ by multiplying a column vector $(h \times 1)$ with a row vector of dimension $(1 \times m)$
* thus by shuffling dimensions:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial c} h_i^T$$

- with dimensions $(h \times 1)(1 \times m) = (h \times m)$

Optimizing Neural Networks

- several techniques can be used to aid training of neural networks
  - eg. weight initialization, batch normalization, regularizations, dataset augmentation

Weight Initialization

- how do we determine what values our weights should start at?
  - generally want to use some sort of Gaussian distribution for each layer, but what should the variances of the distribution be?
    - eg. small vs. large, using some heuristic, etc.
  - note that handling initializations only affects the early iterations, and doesn't deal with other issues with training that may arise in later iterations
- using *small* random weight initializations:
  - motivated by the fact that we don't want large initial values that may bias training
  - empirically, this initialization causes all activations to *decay* to zero
    - activations are the outputs of each layer ie. later layers output approximately zero
  - when we consider backprop with a small weight initialization:
    - backpropagating over later layers gives a gradient ≈ 0
    - must propagate through a multiplication gate that multiplies by layers where $h_n \approx 0$
  - other activations such as $tanh$ have these same undesired properties as ReLU with small weight initializations
  - may be appropriate in much smaller neural networks
- using *large* random weight initializations:
  - empirically, this initialization coupled with ReLU causes the units to explode in magnitude
    - gradients become much too large, leading to zig-zagging
  - for $tanh$, leads to highly saturated activation regions with small gradients and thus little to no learning
- in **Xavier initialization**:
  - argues that the variance of all units across all layers ought to be the same:
    - similarly for backpropagated gradients
    - motivated by the assumption that the input itself has equal variance across all dimensions

  - concretely, for any unit in the ith layer h_i
    * $\mathbb{E}(h_i) = \mathbb{E}(h_j)$, $var(h_i) = var(h_j)$, and $var(\nabla_{h_i} J = \nabla_{h_j} J)$
- then we can argue the following, where n_{in} is the number of units in the previous $i-1$th layer:

$$\begin{aligned} h_i &= \sum_{j=1}^{n_{in}} w_{ij} h_{i-1,j} \\ var(wh) &= \mathbb{E}^2(w) var(h) + \mathbb{E}^2(h) var(w) + var(w) var(h) \\ &= var(w) var(h) \end{aligned}$$

$$var(h_i) = var(h_{i-1}) \cdot \sum_{j=1}^{n_{in}} var(w_{ij})$$

$$\begin{aligned} \sum_{j=1}^{n_{in}} var(w_{ij}) &= \frac{var(h_i)}{var(h_{i-1})} = 1 \\ n_{in} \cdot var(w_{ij}) &= 1 \\ var(w_{ij}) &= \frac{1}{n_{in}} \end{aligned}$$

  - assumptions:
    * all units are linear
    * $\mathbb{E}(w) = \mathbb{E}(h) = 0$
    * w_{ij} and h_{i-1} are independent, allowing their variances to be separated
    * we want all weights to become identically distributed
  - in addition, the assumption of zero expected value only works for $tanh$ since ReLU does not have an expected value of 0
  - the same argument can be made for the backpropagated gradients to argue that $var(w_{ij}) = \frac{1}{n_{out}}$
    * where n_{out} is the number of units in the next layer
  - thus, we have the following:

$$\begin{aligned} n_{avg} &= \frac{n_{in} + n_{out}}{2} \\ var(w_{ij}) &= \frac{1}{n_{avg}} \end{aligned}$$

    * ie. initialize each weight in layer i to be drawn from $\mathcal{N}(0, \frac{2}{n_{in}+n_{out}})$
- however, as previously mentioned, this Xavier initialization typically leads to dying ReLU units:
  - in the He initialization uses an additional normalizer factor of 2 when using Xavier initialization with ReLU

  - motivated by the intuitiion that half of linear activations should be killed by ReLU, which should decrease the variance by half
  - ie. set the variance of each unit to $var(w_{ij}) = \frac{2}{n_{avg}}$

Normalization

- normalization is motivated by a problem called **internal covariate shift**:
  - an obstacle to standard training is that the distribution of inputs to each layer changes as learning occurs in *previous* layers
    - thus the unit activations can be very variable
  - in addition, when we do gradient descent, we calculate how to update each parameter *assuming* the other layers do not change:
    - but these layers may change drastically
    - ie. $\frac{\partial L}{\partial W_i}$ tells how L changes if W_i *alone* is "wiggled"
  - leads to several issues:
    - learning rates may be forced to be smaller than if the distributions were not so variable
    - networks are more sensitive to initializations
    - difficulties in networks that saturate, where learning will no longer occur
  - one technique may be to only change a single W_i at a time, but this will drastically increase the training time by a factor of the number of layers
- the idea of **batch-normalization** is to normalize the output of each layer to have unit statistics:
  - learning then becomes simpler because parameters in the lower layers do not change the statistics of the input to a given layer
  - ie. for a layer $h_i = relu(x_i)$, want $\mathbb{E}(x_i) = 0$ and $var(x_i) = 1$
  - the batch-norm component can be placed in different positions:
    - in the original paper, the batch-norm was performed after the affine unit but before the ReLU such that $h_i = f(batchnorm(W_i h_{i-1} + b_i))$
      - note that putting this batch-norm before ReLU will actually initially kill off half the activations
    - more recently, it has become common practice to perform batch-norm after the ReLU to improve performance
  - another issue with batch-norm is that it forces features to be learned a certain way ie. with unit mean and variance:
    - batch-norm gives an alternative way to change the mean and variance as extra parameters
    - allows network to undo batch-norm for specific features
  - thus batch-norm with parameters *encourages* the network to start with

$$y_j = batchnorm(x_j, \varepsilon)$$

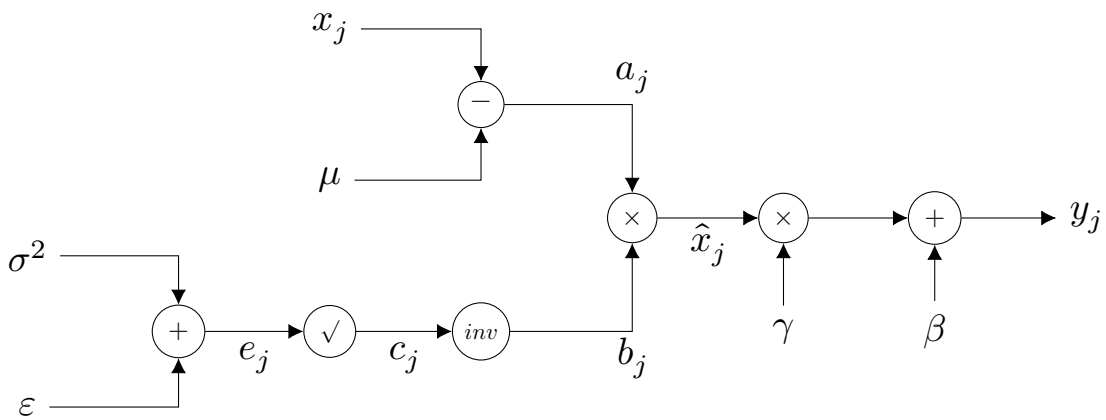

Figure 17: Backprop for Batch-Norm

unit statistics, but gives a way to *undo* the normalization in case the network finds a more optimal way to learn
    * empirically, allows higher learning rates to be used and reduces strong dependence on initialization
- in practice, to implement batch-norm, we can do the following:

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$
$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$
$$\sigma_i^2 = \frac{1}{m}(x_i^{(j)} - \mu_i)^2$$
$$y_i = \gamma_i \hat{x}_i + \beta_i$$

  - scaling is done on a *per unit* rather than a per layer basis for computational efficiency
    * to normalize the entire layer, we would have to use covariance matrix multiplication ie. $\Sigma^{\frac{-1}{2}}(x - \mu)$
      · requires expensive matrix inversion and multiplication to backprop through
  - scaling and shifting parameters γ_i, β_i are applied *per unit* in order to allow the network to rescale the activations and thus undo the normalization
  - ε is a small hyperparameter to avoid division by zero

- calculating the backprop of batch-norm from the computational graph in Figure 17, for a single unit i (the i subscript is dropped from the calculation for

simplicity):

$$\begin{aligned}
\frac{\partial L}{\partial \beta} &= \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \\
\frac{\partial L}{\partial \gamma} &= \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \hat{x}_j \\
\frac{\partial L}{\partial \hat{x}_j} &= \frac{\partial L}{\partial y_j} \gamma_i \\
\frac{\partial L}{\partial a_j} &= \frac{1}{\sqrt{\sigma^2 + \varepsilon}} \frac{\partial L}{\partial \hat{x}_j} \\
\frac{\partial L}{\partial \mu} &= -\frac{1}{\sqrt{\sigma^2 + \varepsilon}} \sum_{j=1}^{m} \frac{\partial L}{\partial \hat{x}_j} \\
\frac{\partial L}{\partial b_j} &= (x_j - \mu) \frac{\partial L}{\partial \hat{x}_j} \\
\frac{\partial L}{\partial c_j} &= -\frac{1}{\sigma^2 + \varepsilon} (x_j - \mu) \frac{\partial L}{\partial \hat{x}_j} \\
\frac{\partial L}{\partial e_j} &= -\frac{1}{2} \frac{1}{(\sigma^2 + \varepsilon)^{\frac{3}{2}}} (x_j - \mu) \frac{\partial L}{\partial \hat{x}_j} \\
\frac{\partial L}{\partial \sigma^2} &= \sum_{j=1}^{m} \frac{\partial L}{\partial e_j}
\end{aligned}$$

  - note that the law of total derivatives is used to sum up all the gradients over all m training examples
  - now, σ^2 and μ are both dependent on x, so we can use the law of total derivatives again:

$$\begin{aligned}
\frac{\partial L}{\partial x_j} &= \frac{\partial L}{\partial a_j} + \frac{\partial \sigma^2}{\partial x_j} \frac{\partial L}{\partial \sigma^2} + \frac{\partial \mu}{\partial x_j} \frac{\partial L}{\partial \mu} \\
&= \frac{1}{\sqrt{\sigma^2 + \varepsilon}} \frac{\partial L}{\partial \hat{x}_j} + \frac{2(x_j - \mu)}{m} \frac{\partial L}{\partial \sigma^2} + \frac{1}{m} \frac{\partial L}{\partial \mu}
\end{aligned}$$

Regularization

- **regularization** is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error:
  - ie. used to improve the *generalizability* of the model and reduce overfitting

  - it may be appropriate to choose a large model while regularizing it, without worrying about the danger of overfitting that comes with increased complexity
- there are many different forms of regularization:
  1. adding a soft constraint on parameter values in the objective function:
     - to account for prior knowledge eg. parameters have a bias
     - to prefer simpler model classes that promote generalization
     - to make an undetermined problem determined
  2. dataset augmentation
  3. ensemble methods ie. essentially combining the output of several models
  4. some training algorithms themselves can be seen as a type of regularization
- one common regularization approach is using parameter norm penalties:
  - ie. modifying the cost function with a **parameter norm penalty**, typically denoted $\Omega(\theta)$
    - not specific to neural networks eg. commonly used in linear regression
  - results in a new cost function of the form:

$$L(\theta, X, y) + \alpha\Omega(\theta)$$

    - where $\alpha \geq 0$
  - α is a hyperparameter that weights the contribution of the norm penalty:
    - with $\alpha = 0$, no regularization is being done
    - as $\alpha \to \infty$, the cost function becomes irrelevant and the model will set parameters to minimize $\Omega(\theta)$
    - choice of α can strongly affect generalization performance
- a common parameter norm regularization is to penalize the size of the weights:
  - promotes models with parameters that are closer to 0 and thus "simpler"
  - this is known as L^2 regularization AKA **ridge regression** or Tikhonov regression
  - the regularization is expressed as:

$$\Omega(\theta) = \frac{1}{2}||w||_2^2 = \frac{1}{2}w^T w$$

    - for matrices, we would use the Frobenius norm
  - calculating the gradient:

$$\tilde{L}(w, X, y) = J(w, X, y) + \frac{\alpha}{2}w^T w$$
$$\nabla_w \tilde{J}(w, X, Y) = \alpha w + \nabla_w J(w, X, y)$$

        * iL the gradient step, we set w to $(1 - \varepsilon\alpha)w - \varepsilon J(w, X, Y)$
            · the $(1 - \varepsilon\alpha)$ term is known as the weight decay
    - equivalent to maximum a-posteriori inference, where the prior on the parameters has a unit Gaussian distribution ie. $w \sim \mathcal{N}(0, \frac{1}{\alpha}I)$
    - when performing this regularization, the component of w aligned with the ith eigenvector of the Hessian is rescaled by a factor $\frac{\lambda_i}{\lambda_i+\alpha}$
    - in linear regression, the least squares solution becomes:

$$w = (X^T XL + \alpha I)^{-1} X^T y$$

        * scales the variance of each input feature
- other similar forms of L^2 regularization include:
    - instead of a soft constraint that the parameters be small, we may have prior knowledge that w is close to some value b:

$$\Omega(\theta) = ||w - b||_2$$

    - alternatively, we may have prior knowledge two parameters w_1, w_2 ought to be close to each other:

$$\Omega(\theta) = ||w_1 - w_2||_2$$

- in L^1 regularization, we instead define the parameter norm penalty as:

$$\Omega(\theta) = ||w||_1 = \sum_i |w_i|$$

    - also intuitively causes the weights to be small
    - however, the subgradient of $||w||_1$ is $sign(w)$, which makes the gradient the same regardless of the size of w, unlike in L^2 regularization:
        * thus typically results in sparse solutions where $w_i = 0$ for several i
        * in L^2 regularization, due to the squared factor, the gradient when L is small, eg. $\ll 1$, is less emphasized
    - can be useful for **feature selection** where features corresponding to zero weight may be discarded
    - equivalent to to maximum a-posteriori inference, where the prior on the parameters has an isotropic Laplace distribution ie. $w_i \sim Laplace(0, \frac{1}{\alpha})$
    - instead of having sparse parameters, it may be appropriate to have sparse *representations*:
        * ie. in hidden layers h_i, set $\Omega(h_i) = ||h_i||_1$
        * allows certain units h_i to be removed, instead of only weights

Dataset Augmentation

- neural networks achieved much popularity when they were applied for computer vision:
  - **dataset augmentation** is a technique used often in computer vision to improve performance:
    - eg. the same cat in an image is still a cat when the image is fliped, croped, rotated, has brightness adjusted, etc.
    - creates *multiple* data samples from a single training image
    - training on these adjusted images should make the model *more* generalizable and robust
      - can be interpreted as adding additional noise into the network
  - augmentation can be done specifically to target certain desired features we want the network to be robust towards
  - typically, there are heuristics used in order to keep the input sizes the same
    - eg. sampling smaller patches when resizing or scaling
- other types of dataset augmentation:
  - injecting noise into the network
    - can be done at various levels eg. perturbing pixels of the image or perturbing weights and biases
  - in **label smoothing**, instead of having a one-hot representation of the correct class, each class for a training example has an associated probability:
    - eg. one class may have probability 0.9 and the remaining classes a uniform probability distribution of 0.011
    - the correct label-smoothed probability eg. 0.9 can be a hyperparameter
    - helps decrease the error rate of networks

Smarter Learning and Ensembling

- in **multitask learning**, we can have the model trained to perform multiple tasks:
  - intiutively, multiple tasks may share common factors that explain variations in the data
    - ie. solve multiple tasks using similar features
  - pass the input through an encoder that generates shared features for multiple decoders that each perform a task
  - eg. in computer vision, could split into the following tasks:
    - labeling the class of every pixel
    - label all instances of a class
    - determining the depth of an object

  - useful when we know specific sub-tasks the network should work on
  - then we can calculate a *multi-task* loss by summing together the losses from each specific task
    - could weight each task in the overall multi-task loss using a hyper-parameter or even a learnable parameter
  - the entire model need not be shared across different tasks
- in **transfer learning**:
  - we can take neural networks trained in one context and use them in another with little additional training
    - if the tasks are similar enough, then the features at later layers of the network ought to be good features for the *new* task
  - useful when little training data is available, but tasks are similar
    - may need to only train a *single* new linear layer at the output of the *pre-trained* network
  - if more data is available, it may still be a good idea to use transfer learning and tune more of the layers
  - eg. use natural features on MRI images to detect specific pathologies for those medical images
- another way to get a boost in performance for very little cognitive work is to use **ensemble** methods:
  1. train multiple different models
     - models don't have to use the same hyperparameters or even belong to the same class
  2. average their results together at test time
  - almost always increases performance by substantial amounts eg. a few percentage improvment
  - intuitively, want to average out the natural errors that occur when training on a single individual network
    - if models are independent, they will usually not all make the *same* errors on the test set
  - with k independent models, the average model error will decrease by a factor $\frac{1}{k}$:

$$\begin{aligned}\mathbb{E}[(\frac{1}{2}\sum_{i=1}^{k}\varepsilon_i)^2] &= \frac{1}{k^2}\sum_{i=1}^{k}\mathbb{E}\varepsilon_i^2 \\ &= \frac{1}{k}\mathbb{E}\varepsilon_i^2\end{aligned}$$

    - assuming models are independent and $\mathbb{E}(\varepsilon_i, \varepsilon_j) = \mathbb{E}(\varepsilon_i)\mathbb{E}(\varepsilon_j)$
      - in addition assuming $\mathbb{E}\varepsilon_i = 0$ and the error statistics are the same across all models
  - even if the models are not independent, the calculated average model

error will still be less than or equal of the error of a single model:

$$\frac{1}{k}\mathbb{E}\varepsilon_i^2 + \frac{k-1}{k}\mathbb{E}(\varepsilon_i\varepsilon_j)$$

- one implementation of ensemble methods is via **bagging** AKA bootstrap aggregation:
  1. construct k datasets using the bootstrap:
     - set a dataset size N
     - draw with replacement from the orignal dataset to get N samples
     - repeat k times
  2. train k different mdoels using the k datasets
  - handles the problem of ensembling that if models are trained on exactly the same datasets, they are probably not completely independent:
    * by bagging with replacement, the datasets will be similar, but still noticeably different from one another
    * however, given different initializations and hyperparameters, models will still tend to produce partially independent errors, even if they are trained from the same dataset
  - very expensive computationally for neural networks, since the time to train models can be very large
- avoiding computational expense:
  1. take snapshots at the model at different local minima and average the results all together
  2. use a technique called dropout
- **dropout** is a computationally inexpesnive yet effective method for generalization:
  - can be viewed as approximating the bagging procedure for exponentially many models, while only optimizing a *single* set of parameters
  - to use the dropout regularizer:
    1. on a single given training iteration, sample a binary mask for all input and hidden units in the network:
       * mask elements will be 1 with probability p and 0 with probability $1-p$ ie. using a Bernoulli hyperparameter p
       * p represents the probability that a unit in the network will remain active for an iteration
       * each iteration, a new mask is generated
    2. apply ie. elementwise multiply the mask to all units
    3. during prediction, mutliply each hidden unit by the Bernoulli mask parameter p as a scaling factor
  - how does dropout emulate model ensembling?
    * considering a neural nework with a different configuration each iteration
      · there are 2^N configurations where N is the number of neurons

        - essentially want the randomly chosen active features to be robust in as many configurations ie. contexts as possible
            - ie. robust enough to work in almost a different network
        - each config should be good at predicting the output
        - dropout may cause units to encode redundant features for robustness
            - eg. recognizing a cat by fur, ears, and a tail
    - but we are changing our loss surface on every iteration, so shouldn't this warp the learning?
        - p is typically around 0.8 and 0.9, so only some units are dropped and the surface isn't radically changing
        - so dropout can be interpreted as generalizing the loss even further so the model performs better
    - training will take longer since generally the number of neurons will be increased
    - the dropout calculation must be backpropagated
        - just multiplciation, so have to cache the masks
- at test time, we need to multiply all units by the Bernoulli probability:
    - the contribution of a single unit to the output should be p times its output
        - ie. over many iterations, the contribution of $w_i h_i$ to the output $h_o ut$ is just $p w_i h_i$
    - thus in test time, there is no additional complexity unlike true ensembling
        - with m ensemble models, testing time evaluation would have scaled by $o(m)$
    - in practice, we can perform **inverted dropout** where the scaling by $\frac{1}{p}$ is already done during training:
        - going from 0 and 1s on the mask to 0 and $\frac{1}{p}$s
        - thus testing will look the same irrespective of if we used dropout or not

Optimizing Gradient Descent

- initialization, regularization, and data augmentation can be used to improve performance of neural networks
    - how do we improve the performance with respect to the chosen optimizer, stochastic gradient descent?
- in typical stochastic gradient performance we compute the gradient over the

examples as:

$$g = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(\theta)$$
$$\theta \leftarrow \theta - \varepsilon g$$

- can we address the zig-zagging issue with this plain gradient descent?
  - using **momentum**, we maintain a running mean of the gradients, which then updates the parameters

  1. initialize $v = 0$ and $\alpha \in [0, 1]$, typically 0.9 or 0.99
  2. until stopping criterion is met:

$$v \leftarrow \alpha v - \varepsilon g$$
$$\theta \leftarrow \theta + v$$

  - essentially augments the gradient with the running average of previous gradients
    - using momentum helps to *average* away the zig-zagging components
  - ie analagous to a gradient momentum
  - we can calculate the first several v values as follows:

$$v_0 = 0$$
$$v_1 = -\varepsilon g_1$$
$$v_2 = \alpha v_1 - \varepsilon g_2 = -\varepsilon(\alpha g_1 + g_2)$$
$$v_3 = \alpha v_2 - \varepsilon g_3 = -\varepsilon(\alpha^2 g_1 + \alpha g_2 + g_3)$$

- does momentum help with finding local optima?
  - momentum moves *ahead* of ordinary gradient descent:
    - thus gradient considering momentum may not be zero where gradient descent may have saturated to zero at some local gradient
    - due to the gradient carrying previous momentum, so it may converge at different local minima that could be better
  - momentum tends to find *shallow* local optima, ie. very flat at the bottom such that the momentum *slows* down:
    - this is desired, since a shallow local optima indicates the loss is not very sensitive to changes in the weights in this region of the loss surface
      - vs. deeper local optima are much more sensitive to weight changes
    - thus we can intuitively predict that shallower local optima should generalize better

- **Nesterov momentum** is an extension on momentum:
  - similar to momentum, except the gradient is calculated *after* taking a step along the direction of the momentum:

$$v \leftarrow \alpha v - \varepsilon \nabla_\theta L(\theta + \alpha v)$$
$$\theta \leftarrow \theta + v$$

    - ie. step in the overall momentum first and then upgrade the parameters based on gradient descent
  - tends to work better than normal momentum
  - but this requires an additional complex backprop step due to the changed loss function:
    - we can instead perform a change of variables from $\theta + \alpha v$ to $\tilde{\theta}$ to simplify the Nesterov momentum calculation
    - no need to compute the extra gradient
  - Nesterov momentum with change of variables:

$$v_{new} = \alpha v_{old} - \varepsilon \nabla_{\tilde{\theta}_{old}} L(\tilde{\theta}_{old})$$
$$\tilde{\theta}_{new} = \tilde{\theta}_{old} + v_{new} + \alpha(v_{new} - v_{old})$$
$$v_{new} \leftarrow v_{old}$$
$$\tilde{\theta}_{new} \leftarrow \tilde{\theta}_{old}$$

- adaptive gradient methods:
  - choosing ε judiciously can be important for learning
  - in the beginning, a larger learning rate is typically better since bigger updates in parameters accelerate learning
  - however, as time goes on, ε may need to be small to make appropriate updates to the parameters
  - sometimes, the we can perform **annealing** on the learning rate to apply a decay rule on it
    - common form to anneal the learning rate is to do so manually when the loss plateaus, or to anneal it after a set number of epochs of gradient descent
  - instead, we can update the learning base adaptively based on the *history* of gradients

- in **Adagrad** or adaptive gradient:
  - the learning rate is decreased through division by the historical gradient norms
    - uses a variable a denoting a running sum of squares of gradient norms

1. initialize $a = 0$ and set v to a small value to avoid division by zero eg. 1×10^{-7}
2. until stopping criterion is met:

$$a \leftarrow a + g \bigodot g$$
$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{a} + v} \bigodot g$$

- however, this history of gradients grows very quickly, monotonically:

$$a_0 = 0$$
$$a_1 = g_1^2$$
$$a_2 = g_1^2 + g_2^2$$
$$a_3 = g_1^2 + g_2^2 + g_3^2$$

- **RMSProp** augments Adagrad by making the gradient accumulator an exponentially weighted moving average:
  - allows the average to *not* monotonically increase

1. initialize $a = 0$, set v to a small value, and set β between 0 and 1, typically 0.99
2. until stopping criterion is met:

$$a \leftarrow \beta a + (1 - \beta) g \bigodot g$$
$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{a} + v} \bigodot g$$

- RMSProp can also be combined with momentum:

1. initialize $a = 0$, set v to a small value, and set α, β between 0 and 1
2. until stopping criterion is met:

$$a \leftarrow \beta a + (1 - \beta) g \bigodot g$$
$$v \leftarrow \alpha v - \frac{\varepsilon}{\sqrt{a} + v} \bigodot g$$
$$\theta \leftarrow \theta + v$$

  - a holds the accumulated gradient while v holds the averaged gradient
  - it is also to do RMSProp with Netserov momentum

- however, there is a more principled way to combine RMSProp with momentum using **Adam** or adaptive moments optimizer:
  - one of the most commonly used and robust, eg. to hyperparameters, optimizers

  - Adam is composed of a momentum-like step, followed by an Adagrad/RMSProp-like step
  - Adam without a bias correction:

1. initializations:
   - set $v = 0$ as the "first" moment and $a = 0$ as the "second" moment
   - set β_1, β_2 to be between 0 and 1, typically 0.9 and 0.999, respectively
   - set ν to be small
2. until stopping criterion is met:

$$
\begin{aligned}
v &\leftarrow \beta_1 v + (1 - \beta_1) g \\
a &\leftarrow \beta_2 a + (1 - \beta_2) g \bigodot g \\
\theta &\leftarrow \theta - \frac{\varepsilon}{\sqrt{a} + \nu} \bigodot v
\end{aligned}
$$

- Adam incorporates a bias correction on the moments:
  - intiution is to account for initialization
    - since the bias corrections amplify the second moments, extremely large steps are not taken at the start of the optimization
  - Adam with bias correction:

1. initializations:
   - set $v = 0$ as the "first" moment and $a = 0$ as the "second" moment
   - set β_1, β_2 to be between 0 and 1, typically 0.9 and 0.999, respectively
   - set ν to be small and $t = 0$
2. until stopping criterion is met:

$$
\begin{aligned}
t &\leftarrow t + 1 \\
v &\leftarrow \beta_1 v + (1 - \beta_1) g \\
a &\leftarrow \beta_2 a + (1 - \beta_2) g \bigodot g \\
\tilde{v} &= \frac{1}{1 - \beta_1^t} v \\
\tilde{a} &= \frac{1}{1 - \beta_2^t} a \\
\theta &\leftarrow \theta - \frac{\varepsilon}{\sqrt{\tilde{a}} + \nu} \bigodot \tilde{v}
\end{aligned}
$$

Appendix

Python Libraries

NumPy

- linera algebra:
  - `linalg.inv(m)` inverts matrix `m`
  - `ndarray.dot(b)` takes the dot product of two arrays
  - `ndarray.T` gives the transpose of the array
  - `vstack(tuple)` stacks the arrays in `tuple` in sequence vertically
    - useful for constructing transposed matrices
- building arrays and distributions:
  - `arange(start, stop, step)` returns evenly spaced values within an interval given a step size
  - `linspace(start, stop, num)` returns `num` evenly spaced numbers over an interval given a number of steps
  - `ones(shape)` returns a new array of shape filled with ones
  - `ones_like(a)` returns an array of ones with the same shape and type as given array `a`
  - `random.uniform(low, high, size)` draws `size` samples from a uniform distribution between `low` and `high`
  - `random.normal(loc, scale, size)` draws `size` samples from a normal distribution with mean `loc` and standard deviation `scale`
  - `flatnonzero(a)` returns indices that are non-zero in the flattened version of `a`
  - `random.choice(a, size, replace)` generates a random sample from `a`
  - `split(a, sections, axis)` divides an array into `sections` subarrays along a specified axis
- other methods:
  - `concatenate` joins a sequence of arrays along an existing axis
  - `count_nonzero(a)` counts the number of non-zero values in the array `a`

MatPlotLib

- `plt.figure` creates a new figure
- `Figure.gca` gets the current axes of a figure

- `Axes.plot(x, y, fmt)` plots a figure with points or line nodes given by x, y
  - `fmt` is a format string eg. `'ro'` for red circles, `'.'` for dots, `'x'` for crosses
- `Axes.set_xlabel(lbl)` and `Axes.set_ylabel(lbl)` sets the labels for the axes
- `Axes.legend()` places a legend on the axes

Linear Algebra Review

Vectors

- $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ is a **column vector** with n elements
- $z = \begin{bmatrix} z_1 z_2 \dots z_n \end{bmatrix}$ is a **row vector** with n elements
- the **transpose** of a column vector is a row vector, and vice versa
  - eg. $x = \begin{bmatrix} x_1 x_2 \dots x_n \end{bmatrix}^T$
- the **dot product** of two column vectors is given by:

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

  - the dot product of two vectors is commutative
- the **norm** of a vector measures its length
- the **p-norm** of a vector is given by the following, where $p \geq 1$:

$$||x||_p = (\sum_{i=1}^{n} |x_i|^p)^{\frac{1}{p}}$$

- the **Euclidian norm** is the 2-norm, and can also be writen as:

$$||x|| = ||x||_2 = \sqrt{x^T x}$$

  - the 2-norm is often more convenient to work with
  - a **unit vector** is a vector with $||x||_2 = 1$

- the dot product can also be written as the following, where θ is the angle between the vectors:
$$x^T y = ||x||||y||cos\theta$$
- x and y are **orthogonal** if $x^T y = 0$:
  - if both vectors have nonzero norm, than they are at a 90 degree angle to each other
  - in $\mathbb{R}^n$ at most n vectors may be mutually orthogonal with nonzero norm
  - if the vectors are orthogonal and also have unit norm, they are **orthonormal**
- a **linear combination** of vectors is a summation of those vectors scaled by a constant:
$$\sum_i c_i v_i$$
  - the **span** of a set of vectors is the set of all points obtainable by linear combinations of the vectors

Matrices

- $A = \begin{bmatrix} a_{11} \dots a_{1n} \\ \vdots \quad \ddots \quad \vdots \\ a_{m1} \dots a_{mn} \end{bmatrix}$ is an $m \times n$ **matrix**
- the product operation of two matrices $C = AB$ is defined by:
$$C_{ij} = \sum_k A_{ik} B_{kj}$$
  - matrix multiplication is distributive and associative
  - however, it is *not* commutative
- matrix multiplication is usually used to write down a system of linear equations, where $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, x \in \mathbb{R}^n$:
$$Ax = b$$
$$A_1 x = b_1$$
$$A_2 x = b_2$$
$$\dots$$
$$A_m x = b_m$$
  - this system can be solved using **matrix inversion** where $A^{-1}A = I_n$

and I_n is the $n \times n$ **identity matrix**:

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ I_n x &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

  - however, $Ax = b$ may not always have a solution:
    * the **column space** is the span of the columns of A
    * to have a solution for all values of $b \in \mathbb{R}^m$, the column space of A must be all of $\mathbb{R}^m$
    * thus A should have at least m columns or $m > n$:
      · however, some of the columns may be redundant ie. **linearly dependent** as well
      · in addition, we need each equation to have at *most* one solution for each value of b, so A can also have at most m columns
    * therefore, the system will have a solution if it is square and all the columns are **linearly independent** ie. no vector in the columns is a linear combination of the other vectors
      · a square matrix with linearly dependent columns is **singular**
    * the **rank** of a matrix is the number of linearly independent columns it has

- the **determinant** of a square matrix $det(A)$ is a function that maps matrices to real scalars:
  - the determinant is equal to the product of all eigenvalues of a matrix
  - thus, since eigenvalues measure the scaling of eigenvectors, the absolute value of the determinant is a measure of how much the matrix expands or contracts space
  - if the determinant is 0, then space is contracted *completely* along at least one dimension, losing all its volume

- the **transpose** of a matrix satisfies:

$$A_{ij} = (A^T)_{ji}$$

  - a matrix is **symmetric** if $A = A^T$

- if the matrix is square ie. $m = n$ with rank n, then the **inverse** of a matrix satisfies the following, where I is the $n \times n$ **identity** matrix:

$$A^{-1}A = AA^{-1} = I$$

- the **trace** of a matrix is the sum of its diagonal elements:

$$tr(A) = \sum_{i=1}^{n} a_{ii}$$

  - the trace operator is invariant to transposition:

$$tr(A) = tr(A^T)$$

  - the trace operator is invariant to cyclic permutations of its input (even if the resulting product has different shapes):

$$tr(ABC) = tr(CAB) = tr(BCA)$$

  - the trace operator is linear:

$$tr(aX + bY) = atr(X) + btr(Y)$$

- the **Frobenius norm** of matrix $A \in \mathbb{R}^{m \times n}$ is:

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{tr(AA^T)}$$

- a **diagonal** matrix consists of only nonzero entries along the main diagonal:
  - ie. $D_{ij} = 0 \ \forall \ i \neq j$
  - eg. the identity matrix
  - useful properties of diagonal matrices:
    - multiplying by a diagonal matrix is computationally efficient
      - to find Dx, we just need to scale each element x_i by D_{ii}
    - to compute the inverse of a square diagonal matrix where each element on the diagonal is nonzero, just take the reciprocal $\frac{1}{D_{ii}}$ on the diagonal
    - nonsquare diagonal matrices do not have inverses, but can still be multiplied cheaply
- a **symmetric** matrix is one that is equal to its own transpose $A = A^T$
- given a symmetric matrix A:
  - A is called **positive definite** if $x^T Ax > 0 \ \forall \ x$
  - if $x^T Ax \geq 0$, A is **positive semidefinite**
  - similarly for **negative definite** and **negative semidefinite** matrices
- an **orthogonal matrix** is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

$$A^T A = AA^T = I$$

$$A^{-}1 = A^T$$

  - thus the inverse of these matrices are easily computed

Decomposition

- an **eigenvector** u_i and its corresponding **eigenvalue** λ_i of a square matrix $A \in \mathbb{R}^{n \times n}$ satisfy:

$$Au_i = \lambda_i u_i$$

  - the eigenvalues can be found by solving:

$$det(A - \lambda I) = 0$$

- collecting all of A's eigenvectors and eigenvalues into the following matrices gives the following **eigendecomposition** of A:

$$U = [u_1 u_2 ... u_n] \quad \Lambda = \begin{bmatrix} \lambda_1 & 0 & ... & 0 \\ 0 & \lambda_2 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \lambda_n \end{bmatrix}$$

$$A = U\Lambda U^{-1}$$

  - this decomposes square matrices in a *unique*, guaranteed way that show us information about their fundamental functional properties
  - tells us that these transformations *scale* space by eigenvalue λ_i in the direction of eigenvector v_i
  - in addition, makes the calculation of A^p easier, since $A^p = U\Lambda^p U^{-1}$

- specifically, if U's columns are an orthonormal set of the eigenvectors:

$$A = U\Lambda U^T$$

- the eigendecomposition can be derived as follows from the definition of an eigenvector:

$$\begin{aligned} Au_1 &= \lambda_2 u_1 \\ Au_2 &= \lambda_2 u_2 \\ A[u_1 u_2] &= [u_1 u_2] \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \\ AU &= U\Lambda \\ A &= U\Lambda U^{-1} \end{aligned}$$

- if A is **normal**, then its eigenvectors are **orthonormal**:

$$u_i^T u_j = 0 \ \ \forall \ \ i \neq j, \quad u_i^T u_i = 1$$

- the **singular value decomposition (SVD)** of a matrix $A \in \mathbb{R}^{m \times n}$ is:

$$A = U\Sigma V^T$$

  - where U is an $m \times m$ matrix with orthonormal columns and V is an $n \times n$ matrix with orthonormal columns
    - the columns of U are the **left singular vectors** of A and are the orthonormal eigenvectors of AA^T
    - the columns of V are the **right singular vectors** of A and are the orthonormal eigenvectors of A^TA
  - Σ is a diagonal $m \times n$ matrix with σ_i as its ith diagonal element
    - σ_i is called the ith **singular value** of A and can be calculated as:

$$\sigma_i = \lambda_i^{\frac{1}{2}}(A^TA) = \lambda_i^{\frac{1}{2}}(AA^T)$$

  - essentially factorizing a matrix into singular vectors and singular values by performing an eigendecomposition for A^TA
  - unlike an eigendecomposition, SVD is applicable to nonsquare matrices as well eg. can solve $Ax = b$ for nonsquare and perform **principal component analysis (PCA)**

Mathematical Tools

- useful properties of common functions:
  1. the **logistic sigmoid**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

     - commonly used to produce the probability parameter of a Bernoulli distribution
     - its range is $(0, 1)$, and saturates when its argument is very positive or negative
  2. the **softplus function**:

$$\zeta(x) = log(1 + e^x)$$

     - useful for producing the $\sigma = \frac{1}{\beta}$ parameter of a normal distribution
     - its range is $(0, \infty)$

Probability

- notation note:
  - $Pr(E)$ is the probability of the event E

  - $Pr(X = x)$ or equivalent shorthand $p(x)$ is the probability of random variable X taking on the value x

- manipulating probabilities revolves around two main rules:

  1. the **law of total probability** ie. sum rule:

$$p(x) = \sum_y p(x, y), \quad x, y \text{ discrete}$$

$$p(x) = \int_y p(x, y)dy, \quad x, y \text{ continuous}$$

  - more particularly, if $A_1, \dots, A_n$ forms a partition of the sample space S, then the probability of an event B is:

$$Pr(B) = \sum_{i=1}^{n} Pr(B \cap A_i)$$

  - alternatively, using the conditional probability definition:

$$Pr(B) = \sum_{i=1}^{n} Pr(B|A_i)Pr(A_i)$$

  2. the **probability chain rule** ie. product rule:

$$\begin{aligned} Pr(E_1, E_2) &= Pr(E_1)Pr(E_2|E_1) \\ &= Pr(E_2)Pr(E_1|E_2) \end{aligned}$$

  - used to break up a joint probability into a product probability
  - can be further decomposed as follows:

$$\begin{aligned} p(w, x, y, z) &= p(w, x)p(y, z|w, x) \\ &= p(x)p(w|x)p(y, z|w, x) \\ &= p(x)p(w|x)p(z|w, x)p(y|z, w, x) \end{aligned}$$

    * any event that has been in front of the conditioning bar must be bind the conditioning bar for all other probability expressions
    * ie. assuming a random variable *has* taken on a value, and evaluating the remaining events
  - can also represent conditional independencies in graphical models
  - generalized for a joint probability over many variables:

$$p(x_1, \dots, x_n) = p(x_1) \prod_{i=2}^{n} p(x_i|x_1, \dots, x_{i-1})$$

    - could be proved through induction

- chain rule examples:

$$\begin{aligned} p(b,c|d,e) &= \frac{q}{p(d)p(e|d)} \\ p(d)p(e|d)p(b,c|d,e) &= q \\ q &= p(b,c,d,e) \end{aligned}$$

$$\begin{aligned} p(d,e)q &= \frac{p(a,b,c,d,e)}{p(a|b,c,d,e)} \\ p(d,e)p(a|b,c,d,e)q &= p(a,b,c,d,e) \\ q &= p(b,c|d,e) \end{aligned}$$

- **Bayes' rule** gives the following relationship:

$$p(x|y) = \frac{p(y|x)p(x)}{\sum_x p(y|x)p(x)}$$

- an intuition on **Bayesian inference**, which appears frequently in machine learning:
    - let x represent model parameters we wish to infer denoted θ and y correspond to data we have observed D:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{\sum_x p(D|\theta)p(\theta)}$$

    - $p(\theta|D)$ is the **posterior distribution**, ie. the probability distribution of model parameters given the data
    - $p(D|\theta)$ is the **likelihood** of the data, ie. the probability of having seen the data given a chosen set of model parameters
    - $p(\theta)$ are **prior parameters**, ie. the probabilities of the model parameters *absent* of any data
        - we can consider that the prior is *updated* by the likelihood to arrie at the posterior distribution on the parameters
    - in Bayesian inference, we calculate $p(\theta|D)$, giving a distribution over the model parameters given the data we observed
        - concretely gives us all the parameters of our model
    - in **Frequentist inference** or **maximum-likelihood estimation**, we calculate $p(D|\theta)$, wanting to infering the θ that makes the data most likely to have been observed
        - ie. we choose the parameters that maximize the likelihood of the data

Derivatives

- in machine learning, we want to find the *best* model according to some performance metric:
  - this requires **optimization**, in which derivatives are crucial
  - in simple cases, we can find minima and maxima by simply setting the derivative equal to 0
  - however, in more complex cases, there is no closed-form solution, but the derivative is still useful in telling us how a change in the model parameters will affect the performance
- the definition of a **derivative** of a function $f : \mathbb{R} \to \mathbb{R}$ at a point $x \in \mathbb{R}$ is:

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

  - gives how much a small change in x affects f:

$$f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$$

  - given $y = f(x)$, we denote the derivative of y with respect to x as $\frac{dy}{dx}$, such that:

$$\Delta y \approx \frac{dy}{dx} \Delta x$$

- the **scalar chain rule** states that if $y = f(x)$ and $z = g(y)$:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

  - ie. a small change in x will cause a small change in y that will in turn cause a small change in z as follows:

$$\begin{aligned} \Delta y &\approx \frac{dy}{dx} \Delta x \\ \Delta z &\approx \frac{dz}{dy} \Delta y \\ &= \frac{dz}{dy}\frac{dy}{dx} \Delta x \end{aligned}$$

- the **gradient** generalizes the scalar derivative to multiple dimensions:
  - if $f : \mathbb{R}^n \to \mathbb{R}$ transforms a vector to a scalar and $y = f(x)$, the gradient with respect to a vector x is :

$$\nabla_x y = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$$

  - the gradient is a vector that is the same size as x
  - each dimension of $\nabla_x y$ tells us how small changes in x in that dimension affect y
  - ie. after changing the ith dimension of x by a small amount affects y as follows:

$$\Delta y \approx \frac{\partial y}{\partial x_i} \Delta x_i$$

    * equivalently, $\frac{\partial y}{\partial x_i} = (\nabla_x y)_i$
  - similarly, after changing multiple dimensions of x, y is changed as follows in a dot product:

$$\begin{aligned} \Delta y &= \sum_i \frac{\partial y}{\partial x_i} \Delta x_i \\ &= (\nabla_x y)^T \Delta x \end{aligned}$$

- ex. If $f(x) = \theta^T x$, find $\nabla_x f(x)$ where $\theta, x \in \mathbb{R}^n, y \in \mathbb{R}$:
  - by rules of the gradient, $\nabla_x y \in \mathbb{R}^n$
  1. expand the dot product in $f(x)$:

$$f(x) = \theta_1 x_1 + ... + \theta_n x_n$$

  2. write out the gradient:

$$\nabla_x y = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} = \theta$$

  - dimensions match up
- ex. If $f(x) = x^T A x$, find $\nabla_x f(x)$ where $A \in \mathbb{R}^{n \times n}, x \in \mathbb{R}^n, y \in \mathbb{R}$:
  - by rules of the gradient, $\nabla_x y \in \mathbb{R}^n$
  1. expand $f(x)$:

$$f(x) = \sum_i \sum_j a_{ij} x_i x_j$$

2. write out the gradient:

$$\begin{aligned}
\frac{\partial y}{\partial x_1} &= \frac{\partial(a_{11}x_1^2)}{\partial x_1} + a_{12}x_2 + ... + a_{1n}x_n + a_{21}x_2 + ... + a_{n1}x_n \\
&= 2a_{11}x_1 + \sum_{j=2}^{n} a_{1j}x_j + \sum_{i=2}^{n} a_{1i}x_i \\
&= \sum_{j=1}^{n} a_{1j}x_j + \sum_{i=1}^{n} a_{i1}x_i \\
&= (Ax)_1 + (A^Tx)_1 \\
\frac{\partial y}{\partial x_i} &= (Ax)_i + (A^Tx)_i \\
\frac{\partial y}{\partial x} &= \nabla_x f(x) = Ax + A^Tx
\end{aligned}$$

   - an intuition check is to consider the problem in a single dimension:
     - ie. when $n = 1$, $f(x) = xax = ax^2$ and $\frac{\partial f(x)}{\partial x} = 2ax$
     - when A is symmetric, the gradient is analogously just $2Ax$

- derivative of a scalar with respect to a matrix:
  - given a scalar y and a matrix $A \in \mathbb{R}^{m \times n}$, the derivative is given by:

$$\nabla_A y = \begin{bmatrix} \frac{\partial y}{\partial a_{11}} & \cdots & \frac{\partial y}{\partial a_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial a_{m1}} & \cdots & \frac{\partial y}{\partial a_{mn}} \end{bmatrix}$$

  - like the gradient, the i, jth element of $\nabla_A y$ tells us how small changes in a_{ij} affect y
  - this layout is called **denominator layout** notation, in which the dimensions of $\nabla_A y$ and A are the same
    - in **numerator layout**, the dimensions are transposed
- derivative of a vector with respect to a vector:
  - given $y \in \mathbb{R}^n$ as a function of $x \in \mathbb{R}^m$, the derivative of y with respect to x would be used as follows:

$$\Delta y_i = \nabla_x y_i \cdot \Delta x$$

  - thus, the derivative J should be an $n \times m$ matrix as follows:

$$J = \begin{bmatrix} (\nabla_x y_1)^T \\ \vdots \\ (\nabla_x y_n)^T \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

  - like the gradient, we can see how small changes in x affect y as follows:

$$\Delta y \approx J \Delta x$$

    * J is called the **Jacobian** matrix
  - since in the denominator layout, the denominator vector changes along rows (instead of along columns, as in the Jacobian):

$$\begin{aligned} J &= (\nabla_x y)^T \\ &= (\frac{\partial y}{\partial x})^T \\ \nabla_x y &= J^T \end{aligned}$$

- the **Hessian** matrix of a function $f(x)$ is a square matrix of second-order partial derivatives of f as follows:

$$H = \begin{bmatrix} \frac{\partial f}{\partial x_1^2} & \frac{\partial f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_n} \\ \frac{\partial f}{\partial x_2 \partial x_1} & \frac{\partial f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_m \partial x_1} & \frac{\partial f}{\partial x_m \partial x_2} & \cdots & \frac{\partial f}{\partial x_m^2} \end{bmatrix}$$

  - the Hessian is denoted as $\nabla_x(\nabla_x f(x))$ or equivalently $\nabla_x^2 f(x)$

Chain Rule

- the **chain rule** for vector valued functions:
  - in the denominator layout, the chain rule runs from right to left
  - if $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $z \in \mathbb{R}^p$ and $y = f(x)$ for $f : \mathbb{R}^m \to \mathbb{R}^n$ and $z = g(y)$ for $g : \mathbb{R}^n \to \mathbb{R}^p$, then:

$$\begin{aligned} \nabla_x z &= \nabla_x y \nabla_y z \\ \frac{\partial z}{\partial x} &= \frac{\partial y}{\partial x} \frac{\partial z}{\partial y} \end{aligned}$$

    - $\nabla_x z$ should have dimensionality $\mathbb{R}^{m \times p}$
    - since $\nabla_x y \in \mathbb{R}^{m \times n}$ and $\nabla_y z \in \mathbb{R}^{n \times p}$, the operations are dimensionally correct
- composing the chain rule:
  - intuitively, a small change Δx affects Δz through the Jacobian $(\nabla_x z)^T$:

$$\Delta z \approx (\nabla_x z)^T \Delta x$$

  - then, through composition:

$$\Delta y \approx (\nabla_x y)^T \Delta x$$
$$\Delta z \approx (\nabla_y z)^T \Delta y$$
$$\Delta z \approx (\nabla_y z)^T (\nabla_x y)^T \Delta x$$

  - thus reduces to the right to left chain rule:

$$(\nabla_x z)^T = (\nabla_y z)^T (\nabla_x y)^T$$
$$\nabla_x z = \nabla_x y \nabla_y z$$

Tensors

- we may need to take a derivative that is more than 2-dimensional:
  - eg. the derivative of a vector with respect to a matrix would be a 3-dimensional **tensor**
    - a tensor is an array with more than two axes
  - if $y = Wx \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, and $W \in \mathbb{R}^{m \times n}$ then $\nabla_W z$ is a 3-dimensional tensor with shape $\mathbb{R}^{m \times n \times n}$:
    - breaking down the vector y into scalar-matrix derivatives that we do know how to compute
    - ie. each $m \times n$ slice is the matrix derivative $\nabla_W y_i$
  - in general, can usually use intuition to calculate tensor derivatives rather than a rigorous derivation
    - actually calculating and storing tensor derivatives can be very expensive for memory and computation
- ex. Consider the squared loss function:

$$\begin{aligned} L &= \frac{1}{2} \sum_{i=1}^{N} ||y_i - W x_i||^2 \\ &= \frac{1}{2} \sum_{i=1}^{N} (y_i - W x_i)^T (y_i - W x_i) \\ &= \frac{1}{2} \sum_{i=1}^{N} z_i^T z_i \end{aligned}$$

$$\frac{\partial L}{\partial W} = \frac{\partial z}{\partial W} \frac{\partial L}{\partial z}$$

$$\begin{aligned}
\frac{\partial z_k}{\partial w} &= \frac{\partial}{\partial W}[y_k - \sum_{j=1} W_{kj}x_j] \\
\frac{\partial z_k}{\partial W_{ip}} &= -\frac{\partial}{\partial W_{ip}}[\sum_j W_{kj}x_j] \\
&= -\frac{\partial}{\partial W_{ip}}[W_{k1}x_1 + ... + W_{kn}x_n] \\
&= \begin{cases} 0 & i \neq k, \\ -x_p & i = k \end{cases}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \varepsilon}{\partial W} &= \frac{\partial z}{\partial W}\frac{\partial \varepsilon}{\partial z} \\
&= \sum_{k=1}^{m} \frac{\partial z_k}{\partial W}\frac{\partial \varepsilon}{\partial z_k} \\
&= \frac{\partial \varepsilon}{\partial z_1}\begin{bmatrix} -x^T \\ 0 \\ \vdots \\ 0 \end{bmatrix} + ... + \frac{\partial \varepsilon}{\partial z_m}\begin{bmatrix} 0 \\ \vdots \\ 0 \\ -x^T \end{bmatrix} \\
&= \begin{bmatrix} -\frac{\partial \varepsilon}{\partial z_1}x^T \\ \vdots \\ -\frac{\partial \varepsilon}{\partial z_m}x_T \end{bmatrix} \\
&= -\frac{\partial \varepsilon}{\partial z}x^T
\end{aligned}$$

- calculation notes:
  - we drop the summation and calculate the gradient of the loss over a single example where $L_i = \varepsilon$
  - $z \in \mathbb{R}^m$, $W \in \mathbb{R}^{m \times n}$, $\frac{\partial z}{\partial W} \in \mathbb{R}^{m \times n \times m}$, and $\frac{\partial z_k}{\partial W} \in \mathbb{R}^{m \times n}$
  - the dimensionality of the gradient $\frac{\partial L}{\partial W}$ should give $(m \times n \times m)(m \times 1) = (m \times n)$
  - $\frac{\partial z_k}{\partial W}$ is a matrix where all the entries are 0, except the kth row which is equal to $-x^T$
- in lecture, we used the trace operator to circumvent the tensor derivative calculation

Discussion Problems

Linear Algebra Review

- ex. Show the following properties for matrices:
  1. if $b^T Ab > 0 \ \forall \ b \in \mathbb{R}^n$, then all eigenvalues of A are positive:

$$\begin{aligned} Av_i &= \lambda_i v_i \\ v_i^T Av_i &= \lambda_i v_i^T v_i \\ v_i^T Av_i &= \lambda_i ||v_i||_2^2 > 0 \\ &\therefore \ ||v_i||_2^2 > 0, \lambda_i > 0 \end{aligned}$$

     - this is a positive definite matrix
  2. if $A \in \mathbb{R}^{n \times n}$ is an orthogonal matrix, then all eigenvalues of A have norm 1:

$$\begin{aligned} Av_i &= \lambda_i v_i \\ A^T Av_i &= \lambda_i A^T v_i \\ v_i &= \lambda_i A^T v_i \\ ||v_i||_2^2 &= |\lambda_i|^2 ||A^T v_i||_2^2 \\ &= |\lambda_i|^2 (A^T v_i)^T (A^T v_i) \\ &= |\lambda_i|^2 v_i^T AA^T v_i \\ &= |\lambda_i|^2 ||v_i||_2^2 \\ &\therefore \ |\lambda_i| = 1 \end{aligned}$$

  3. If $A \in \mathbb{R}^{m \times n}$ is a matrix with rank r, then $\sigma_i(A) = \lambda_i^{\frac{1}{2}}(AA^T)$:

$$\Sigma\Sigma^T = diag(\sigma_1^2, \dots, \sigma_n^2)$$

$$\begin{aligned} A &= U\Sigma V^T \\ AA^T &= (U\Sigma V^T)(U\Sigma V^T)^T \\ &= U\Sigma V^T V\Sigma^T U^T \\ &= U\Sigma\Sigma^T U^T \\ &= U diag(\sigma_1^2, \dots, \sigma_n^2) U^T \\ &\therefore \ \sigma_i(A) = \lambda_i^{\frac{1}{2}}(AA^T) \end{aligned}$$

     - producing an eigendecomposition of AA^T

Vector and Matrix Derivatives

- trace has useful properties for computing derivatives:
  - the **trace** of a matrix is the sum of the diagonal entries

  - $Tr(AB) = Tr(BA)$
  - $Tr(A) = Tr(A^T)$
  - $Tr(A^T B) = \sum_i \sum_j a_{ij} b_{ij}$

- ex. $\nabla_A tr(AB)$ where $A = \begin{bmatrix} a_1^T \\ \vdots \\ a_n^T \end{bmatrix}$ and $B = [b_1 \dots b_n]$:

$$\begin{aligned} Tr(AB) &= a_1^T b_1 + \cdots + a_n^T b_n \\ &= \sum_{i=1}^{n} a_i^T b_i \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{ji} \\ [\nabla_A tr(AB)]_{ij} &= b_{ji} \end{aligned}$$

$$\nabla_A tr(AB) = B^T$$

- ex. $\nabla_A (x^T A x)$:

$$Tr(x^T A x) = x^T A x$$

$$\begin{aligned} \nabla_A (x^T A x) &= nabla_A (Tr(x^T A x)) \\ &= \nabla_A (Tr(A x x^T)) \\ &= (x x^T)^T \\ &= x x^T \end{aligned}$$

- ex. $\nabla_z (x - z)^T \Sigma^{-1} (x - z)$ where $y = f(z) = x - z$, $r = g(y) = y^T \Sigma^{-1} y$, and Σ^{-1} is symmetric:

$$\begin{aligned} \nabla_z y &= \nabla_z (x - z) \\ &= -I \end{aligned}$$

$$\begin{aligned} \nabla_y r &= \nabla_y (y^T \Sigma^{-1} y) \\ &= \Sigma^{-1} y + (\Sigma^{-1})^T y \\ &= 2\Sigma^{-1} y \end{aligned}$$

$$\begin{aligned} \nabla_z r &= \nabla_z y \nabla_y r \\ &= -2\Sigma^{-1} y \\ &= -2\Sigma^{-1} (x - z) \end{aligned}$$

- a technique used to address the overfitting found in the normal least squares approach is called **regularization**:
  - this produces the regularized least squared problem with the following cost function where λ is a tunable regularization parameter:

$$L = \frac{1}{2}\sum_{i=1}^{N}(y_i - \theta^T \hat{x}_i)^2 + \frac{\lambda}{2}||\theta||_2^2$$

    - want a least squares solution with a smaller ie. simpler θ norm
  - optimizing L:

$$\begin{aligned} L &= \frac{1}{2}(Y - X\theta)^T(Y - X\theta) + \frac{\lambda}{2}||\theta||_2^2 \\ L(\theta) &= \frac{1}{2}[Y^TY - Y^TX\theta - \theta^TX^TY + \theta^TX^TX\theta] + \frac{\lambda}{2}\theta^T\theta \\ &= \frac{1}{2}Y^TY - Y^TX\theta + \frac{1}{2}[\theta^T(X^TX + \lambda I)\theta] \\ \nabla_\theta L(\theta) &= -\nabla_\theta[Y^TX\theta] + \frac{1}{2}\nabla_\theta[\theta^T(X^TX + \lambda I)\theta] \\ &= -X^TY + \frac{1}{2}2(X^TX + \lambda I)\theta \ \ [=] \ \ 0 \\ \theta &= (X^TX + \lambda I)^{-1}X^TY \end{aligned}$$

    - note that $X^TX + \lambda I$ is symmetric

Supervised Classification and Gradients

- examine how k-NN classifiers can be more robust to noise:
  - given two labels 0 and 1, a test point x, and its k nearest neighbors z_i where p_i is the probability the label of z_i is not equal to x
  - let $p_1 = 0.1$ and $p_i = 0.2 \ \ \forall \ \ i > 1$
  - the probability that the 1-NN classifier makes a mistake classifying is 0.1
  - the probability that the 3-NN classifier makes a mistake occurs when at least 2 of the 3 nearest neighbors have a different label than x:
    - $Pr(\textit{all different}) = 0.1 \times 0.2^2$
    - $Pr(\textit{first different, one other different}) = 0.1 \times 0.2 \times 0.8$
    - $Pr(\textit{second and third different}) = 0.9 \times 0.2^2$
    - altogether, the sum is of probabilities is 0.072
  - thus the 3-NN classifier is more robust, since it checks more neighbors
- taking the derivative of the softmax function:

$$f(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

- calculate $\nabla_{z_i} f(z_j)$ for $i = j$:

$$\begin{aligned}
\nabla_{z_i} f(z_j) &= \frac{\partial}{\partial z_i}[\frac{e^{z_i}}{e^{z_1} + ... + e^{z_c}}] \\
&= \frac{e^{z_i} \sum_k e^{z_k} - e^{z_i} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} \\
&= \frac{e^{z_i}}{\sum_k e^{z_k}} - \frac{(e^{z_i})^2}{(\sum_k e^{z_k})^2} \\
&= f(z_i) - (f(z_i))^2 \\
&= f(z_i)(1 - f(z_i))
\end{aligned}$$

  - uses the quotient rule for derivatives
- calculate $\nabla_{z_i} f(z_j)$ for $i \neq j$:

$$\begin{aligned}
\nabla_{z_i} f(z_j) &= \frac{\partial}{\partial z_i}[\frac{e^{z_j}}{e^{z_1} + ... + e^{z_c}}] \\
&= \frac{0 \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} \\
&= \frac{-e^{z_j}}{\sum_k e^{z_k}} \frac{e^{z_i}}{\sum_k e^{z_k}} \\
&= -f(z_j) f(z_i)
\end{aligned}$$

Backpropagation

- performing backprop on the following regularized linear classification model:

$$\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
L &= \frac{1}{2}(y - t)^2 \\
R &= \frac{1}{2} w^2 \\
L_{reg} &= L + \lambda R
\end{aligned}$$

- want to be able to update w, b using $\frac{\partial L_{reg}}{\partial w}$ and $\frac{\partial L_{reg}}{\partial b}$

- from the chain rule:

$$\begin{aligned}\frac{\partial L_{reg}}{\partial w} &= [\sigma(wx+b)-t](\frac{\partial}{\partial w}[\sigma(wx+b)-t]) + \lambda w \\ &= [\sigma(wx+b)-t]\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) + \lambda w \\ &= [\sigma(wx+b)-t]\sigma'(wx+b)x + \lambda w\end{aligned}$$

$$\frac{\partial L_{reg}}{\partial b} = [\sigma(wx+b)-t]\sigma'(wx+b)$$

  - using the full chain rule is not modular and involves redundant calculations

- using backprop on the computation graph in Figure 18:

$$\begin{aligned}\frac{\partial L_{reg}}{\partial L} &= \frac{\partial L_{reg}}{\partial R} = 1 \\ \frac{\partial L_{reg}}{\partial g} &= g \\ \frac{\partial L_{reg}}{\partial q_2} &= \lambda q_2 \\ \frac{\partial L_{reg}}{\partial g} &= g \\ \frac{\partial L_{reg}}{\partial y} &= \sigma'(z)g \\ \frac{\partial L_{reg}}{\partial b} &= \sigma'(z)g \\ &= \sigma'(wx+b)[\sigma(wx+b)-t] \\ \frac{\partial L_{reg}}{\partial m} &= \sigma'(z)g \\ \frac{\partial L_{reg}}{\partial q_1} &= x\sigma'(z)g \\ \frac{\partial L_{reg}}{\partial} &= \frac{\partial L_{reg}}{\partial q_1} + \frac{\partial L_{reg}}{\partial q_2} \\ &= x\sigma'(z)g + \lambda q_2 \\ &= x\sigma'(wx+b)[\sigma(wx+b)-t] + \lambda w\end{aligned}$$

  - note that h is the identity function, so its gradients drop out in the law of total derivatives

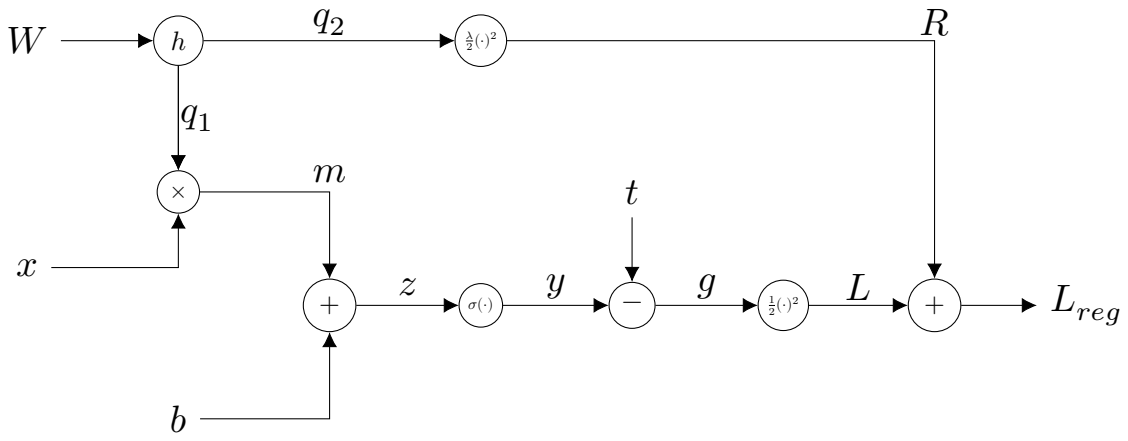

Figure 18: Backprop for a Regularized Linear Classification