Machine Learning and Deep Learning. Theory behind them

Emin Mammadov

COVERED
TO DO
Scaling, standardization, normalizing: differences between all of them and when one is more appropriate than the other

Role of Bias

Losses

Each of hyperparameters

Statistics: EDA, Hypothesis formulation and testing. How does distribtion help us and what does it tell us?

ARTIFICIAL NEURAL NETWORKS
Activation functions
AFs are used to convert an input signal of a node in ANN to an output signal. That output signal is now an input to the next layer in the network. They determine the output of DL. model, accuracy, and computational efficiency of training a model - which can make or break a large scale NN. They also have a major effect on the ANN ability to converge and the convergence speed. AFs are mathematical equations that determine the output of a neural network ($\sum \omega_i \cdot x_i + b$). The function is attached to each neuron in the network and determines whether it should be activated (fired) or not, based on whether each neuron's input is relevant for the model's prediction. AFs can help to normalize the output of each neuron to a range between 1 and 0 or between 1 and -1.

In ANNs, numerical data points, called inputs, are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight fives the output of the neuron, which is transferred to the next layer. AFs are a mathematical "gate" in between the inputs feeding the current neurons and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, or more complicated; something that maps the input signals into outputs signals that are needed for the neural network to function.

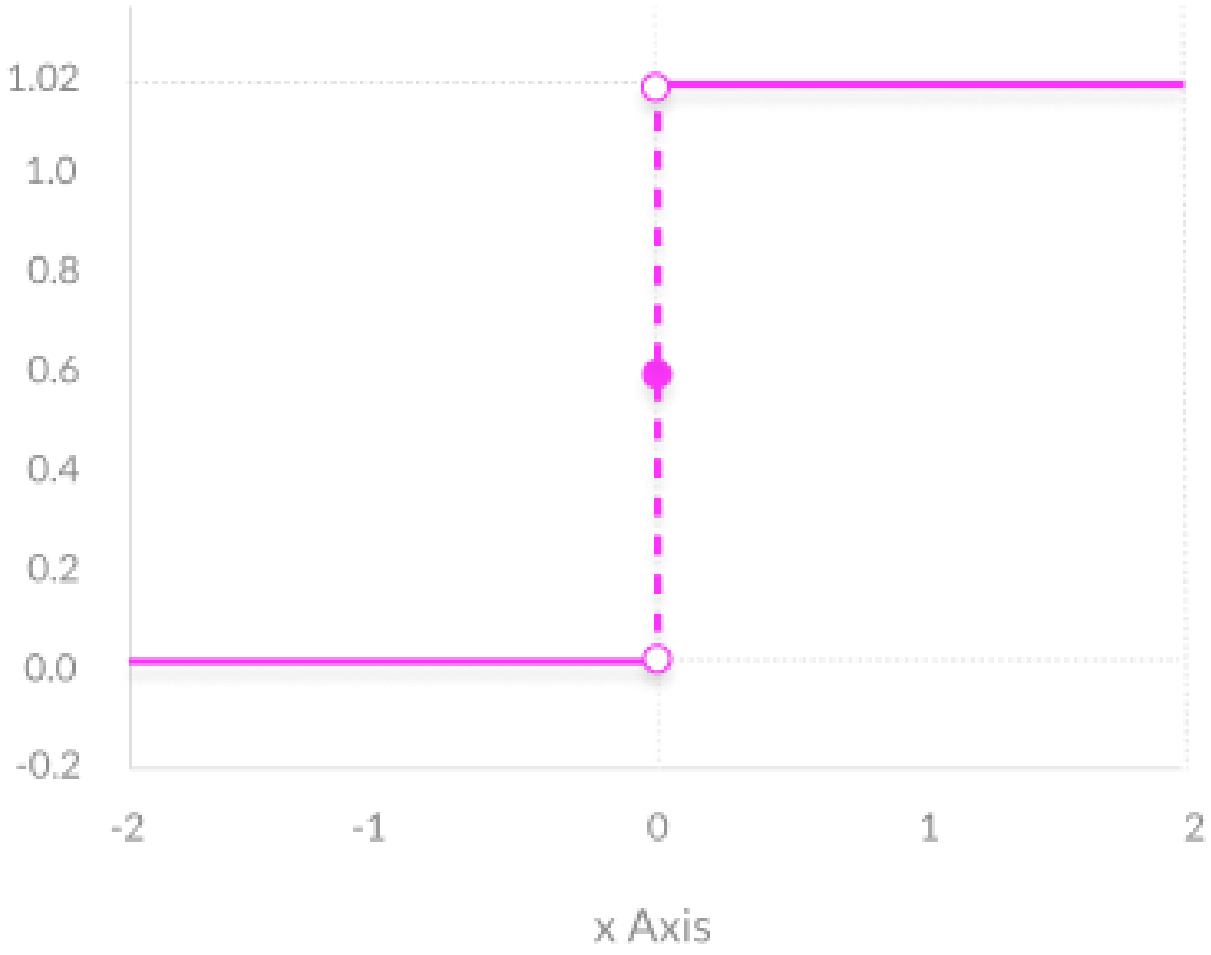
If we would not have activation functions, we would have a simple linear regression model and ANN would not learn complex functional mappings.

Non-linear functions are those which have degree more than one and they have a curvature when we plot a non-linear function. We need ANN to learn and represent almost anything and any arbitrary complex function which maps inputs to outputs. ANNs are considered Universal Function Approximators; meaning that they can compute and learn any function at all. So, non-linear AFs allow the model to create complex mappings between the network's inputs and outputs and modeling complex data. Non-linear functions address the problems of linear activation function: they allow backpropagation because they have a derivation function which is related to the inputs; they allow stacking multiple layers of neurons.

Another important feature of AFs is that they should be differentiable, since we need to perform backpropagation optimization, while propagating backwards in the network to compute gradients of Error (loss) wrt. to weights and then accordingly optimize weights using gradient descent or any other optimization algorithm to reduce error.

Types of Activation Functions

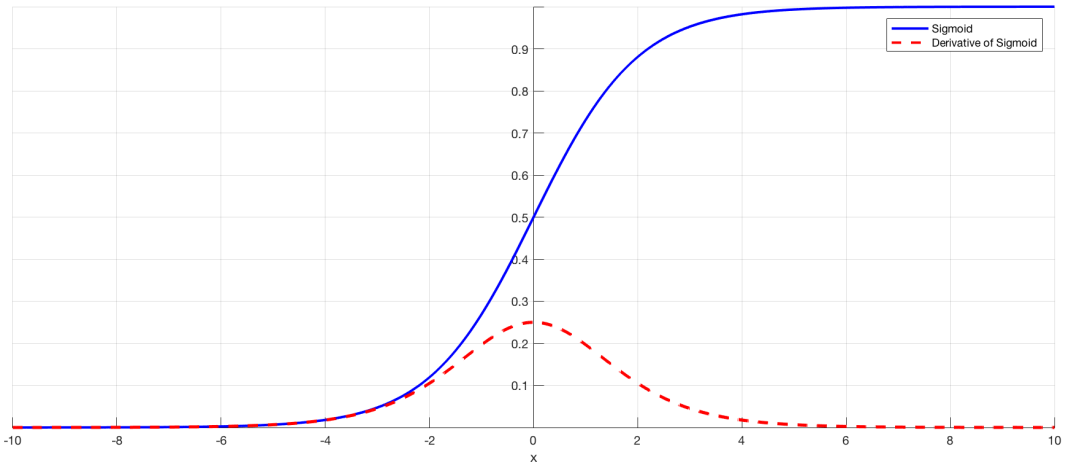
Binary Step function Threshold based. If input is above (or below (depending on the context)), the neuron is activated. It doesn't allow multi-value outputs; it cannot support classifying the inputs into one of several categories.



Linear $A = c \cdot x$. It takes the inputs, multiplied by the weights and creates the signal proportional to the input. It allows multiple outputs. The major problem is that it's not possible to use backpropagation, as the derivative of the function is constant and has not relation to the input, x. So, it's not possible to go back and understand which weights in the input neurons can provide a better prediction. Also, no matter how many layers are there, with linear AF, last layer is a linear function of the first layer, and NN becomes just one layer and it becomes a simple linear regression model. Also, it's a constant gradient and descent is going to be on constant gradient.

Sigmoid or logistic

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



Sigmoid takes a real-valued number and squashes it into range between 0 and 1; $\sigma(x) \in (0, 1)$. Large negative numbers become 0 and large positive numbers become 1. Two major drawbacks:

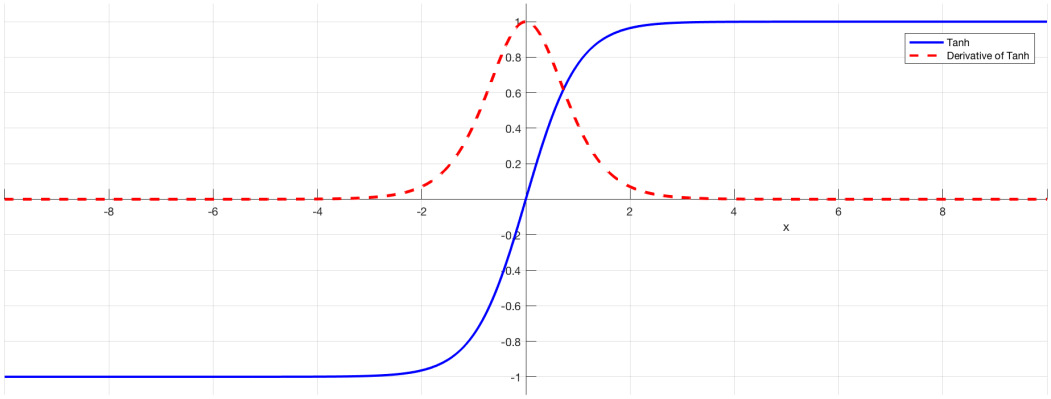
- Sigmoid saturate and kill gradients: If neuron activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero (see the red dotted line) (derivateve is $\sigma(x)' = \sigma(x) \cdot (1 - \sigma(x))$. During the backpropagation, this gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will kill the gradient and no signal will flow. Therefore, it is important to initialize the weights of sigmoid neurons to prevent saturations. On the other side, if initial weights are too large, then most neurons would become saturated and network will never learn. In other words, there is a vanishing gradient problem - for very high or very low values, there is almost no change to the prediction and that results in the network refusing to learn further, or being to slow to reach an accurate prediction.
- Sigmoid outputs are not zero-centered(?): It is undesirable since neurons in later layers of processing in ANN would be receiving data that is not zero-centered. This will have an effect during gradient descent, since if data coming into a neuron is always positive, then gradient on weights during backpropagation become either all positive, or all negative. This will introduce zig-zagging dynamics in the gradient updates for weights.

Tanh

The hyperbolic tangent (tanh) function:

$tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$

Squashes real-valued numbers to the range between -1 and 1: $tanh(x) \in (-1, 1)$. The derivative is $tanh'(x) = 1 - tanh^2(x)$. The output is zero-centered (thus, makes it easier to model inputs that have stronly negative, neutral and strongly postivie values) but activation still saturates.



Softmax

$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = 1, 2, ..., K$

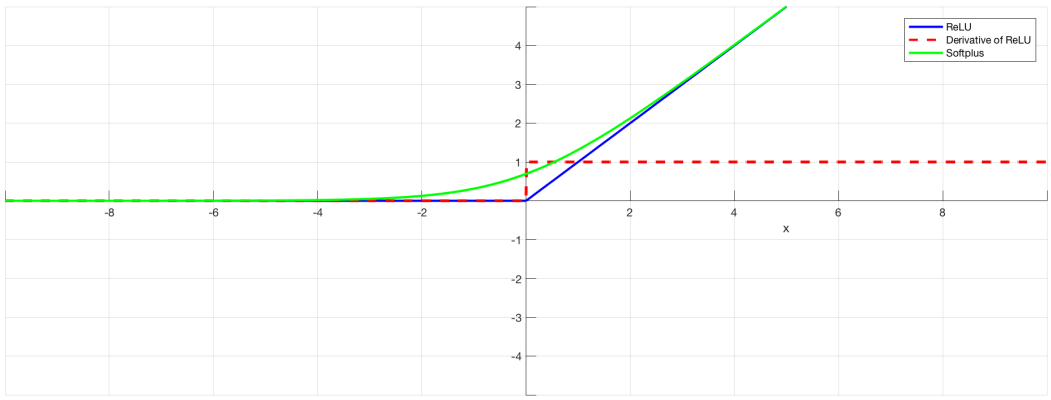
Softmax function, or normalized exponential function is a generalization of the logitistic function that squashed a K-dimensional vector z from arbitrary real values to a K-dim. vector $\sigma(z)$ of real values in the range [0,1] that add up to 1. In probability, an output can be used to represent a categorical distribution over K different possible outcomes. The input to the function is the result of K distinct linear functions, and the predicted probability for the jth class given a sample vector and a weighting factor w is

$P(y = j|x) = \frac{e^{x^T \cdot w_j}}{\sum_{k=1}^{K} e^{x^T \cdot w_k}}$

ReLU

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

where x is the input to a neuron. Activation is simply thresholded at zero. The range is between 0 and ∞.



More effective than sigmoid and tanh, because it reduces the computational cost. It is also non-linear and allows for backprop. Drawback: dying ReLU problem - when inputs appraoch zero or negative, the gradient of the function becomes zero, and network cannot perform backpropagation and cannot learn.

Leaky and Parametric ReLU

Leaky ReLU is one attempt to fix the dying ReLU problem. Instead of function being zero when x<0, a leaky ReLU will instead have a small negative slope. Then function is:

$$\begin{cases} \text{f(x)} = 0.01 \cdot x, (x < 0) \\ \text{f(x)} = \text{x, (x>=0)} \end{cases}$$

This form could not get consisten results and as the result, Parametric Relu were introduced. They would take the idea further by making the coefficient of leakage into a parameter that is learned along with the other neural network parameters:

$$\begin{cases} \text{f(x)} = \alpha \cdot x, (x < 0) \\ \text{f(x)} = \text{x, (x>=0)} \end{cases}$$

where α is a small constant (<1).

LOSS, COST AND OBJECTIVE FUNCTIONS

Loss Function is usually a function defined on a data point, prediction and label and that's something we measure for penalty. Examples: hinge loss, 0/1 loss

Cost Function is more general definition. It might be a sum of loss functions over the training set plus some model complexity penalty (regularization).

Objective function is the most general term for any function that you optimize during training. Example: a probability of generating training set in maximum likelihood approach is a well defined objective function but it's not a loss, nor cost function. Examples of objective functions are: MLE.

In general terms: a loss function is a part of a cost function which is a type of an objective function.

In regression problems, the usual loss function used is mean squared error. In binary-classification problem, it is cross-entropy (also known as logarithmis loss). In multi-class classification problem, it is the same. However, other loss functions used in regression problems are: mean square error/quadratic loss, mean absolute error, huber loss/smooth mean absolute error, log cosh loss, quantile loss. In classification tasks, loss functions are: log loss, focal loss, KL divergence/relative entropy, exponential loss, hinge loss.

Cross-Entropy

Cross-entropy loss, or log loss, measure the performance of a classification model whose output is a probability value between 0 and 1. It is commonly used to quantify the difference between two probability distributions. The true distribution (the one that the model is trying to match) is expressed in terms of one-hot distribution. For example, suppose for a specific training instance, the label is B. The one-hot distribution is: Pr(A) = 0; Pr(B)=1; Pr(C)=0. Now the model will predict the following probability distribution: Pr(A)=0.228; Pr(B)=0.619; Pr(C)=0.153. As cross-entropy is formulated as: $H(p,q) = -\sum p(x) \cdot logq(x)$, where $p(x)$ is the wanted probability and $q(x)$ is the actual probability, sum over three classes is 0.479.

Hinge loss

Loss function used for training classifiers and mathematically described as: $l(y) = max(0, 1 - t \cdot y)$, where t is an intended output(+/-1) and y is a classifier score. Hinge loss is used for "maximum-margin" classification, most notably for SVMs. If we extend SVM to multi-class classification (one vs. all), then hinge loss is modified according to Crammer and Singer: $l(y) = max(0, 1 + max(w_t \cdot x - w_y \cdot x))$

KL divergence/relative entropy loss

NEURAL NETWORK ARCHITECTURE/TRAINING PARAMETERS

Number of input neurons

Number of input neurons must be equal to number of features/attributes/columns in the data.

Sample, Batch, Epoch

In NNs a sample is a single row of data. Contains the inputs that are fed into the algorithm and an output that is used to compare to the prediction and calculate the error. A training dataset contains many rows of data (many samples).

Batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters and as the result, dictates how often to update the parameters. Can be thought of as a for-loop iterating over one or more samples and making predictions. Training dataset is divided into one or more batches. If we use the whole training dataset to create one batch, the learning algorithm is called batch gradient descent. When we use one sample, the learning algorithm is called SGD. If batch-size is more than 1 but less than the size of training set => mini-batch gradient descent. Larger batch size -> faster training but it might degrade accuracy.

Epoch is a a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches. For example, as above, an epoch that has one batch is called the batch gradient descent learning algorithm. Can be thought of as a for-loop where each loop proceeds over the training dataset. Within this for-loop is another nested for-loop that iterates over each batch of samples

Examples: If we have 200 samples of data and we choose a batch size of 5 and 1000 epochs. That means that the whole dataset will be divided into 40 batches, each with 5 samples. The model weights will be updated after each of 5 samples. 1 epoch involves 40 batches or 40 updates to the model. With 1000 epochs the model will be exposed to or pass through the whole dataset 1000 times. That is the total of 40000 batches during the entire training process.

In the backpropagation, we update the weights after every batch. We use forward to get the output and backward to get the gradient and we update the weights using the average gradient. That gradient is the derivative of the loss wrt. the weights. We loop over the consecutive batches and update the weights. This more frequent weight updating is what allows mini-batch gradient descent to tend to converge more quickly.

Learning rate controls the rate at which the model learns. It controls the amount the weight are updated during training. Decaying the learning rate and increasing the batch size during training are equivalent.

Unit (neurons) often refer to the activation function in a layer by which the inputs are transformed via a nonlinear activation function. In the input layer, each input must be linearly independent from each other

Hidden layers receive the weighted inputs, transform them with the set of mostly non-linear functions and then passes these values as output to the next layer

Weight Initialization:

Initializing weights to 0 will make the model equivalent to a linear model. Doing so will make the derivative wrt. loss function the same for every ω, as the result all the weights will have the same values in the subsequent iterations. It will make the model no better than the linear model. In other words, if we set up all the weights to 0, every neuron in the network computes the same output and also compute the same gradients during backprop and undergo the exact same parameter updates. There is not source of asymmetry between neurons if their weights are initialized to be the same.

As the result, we want weights to be close to 0 but not zero. It is common to initialize the weights of the neurons to small numbers and refer to doing so as symmetry breaking. The primary idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse part of the full network. The weights are usually sampled according to the zero mean, unit standard deviation gaussian. The problem though is that distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. We can normalize the variance of each neuron output to 1 by scaling its weight vector by the square root of the number of inputs. This will ensure that all neurons in the network will have approximately the same output distribution and empirically improve the rate of convergence. In the paper "Understanding the difficulty of training deep feedforward neural network" by Glorot, authors recommend the initialization of the form $Var(\omega) = \frac{2}{n_{in}+n_{out}}$, where n_{in}, n_{out} are the number of units in the previous layer and the next layer. However, He in "Delving deep into rectifiers" derives this initialization specifically for ReLU neurons, reaching the conclusion that the variance of neurons should be $2/n$.

Another way to address the uncalibrated variances problem is to set all weight matrices to zero but every neuron is randomly connected (with a weight sampled from a small gaussian as above) to a fixed number of neurons below it. This approach is known as sparse initialization.

In practice, as recommended by He in "Delving Deep into Rectifiers ... ", the approach is to use ReLU units and use $\omega = np.random.randn(n) * sqrt(2/n)$

For biases, it is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights. For ReLU, 0.01 for all biases is a common number as it ensures that ReLU units fire in the beginning and therefore obtain and propagate the gradient.

Batchnorm saves a lot of headaches when it comes to properly initializing neural networks. What batchnorm does is it forces the activations throughout networks to take on a unit gaussian distribution during the trainings. Batchnorm can be interpreted as doing preprocessing at every layer of the network but integrated into the network itself in a differentiable manner.

Batch Normalization increases the stability of a neutral network by normalizing the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. However, after this shift/scale of activation outputs by some randomly initialized parameters, the weights in the next layer are no longer optimal. SGD undoes this normalization if it's a way for it to minimize the loss function.

What is during the training of a neural network, the weights will become large and will cause the output of the unit to be large and will cause an instability. Batchnorm is useful in this case scenario. It will normalize output from activation function; then it multiples by an arbitrary parameter (z*g) and then adds another arbitrary parameter to resulting product (z*g)+b. This calculation sets a new std and mean of data. All those parameters are trainable and become optimized during the training. This process ensures that weights don't become extremely high or low, as normalization is included now. Everything occurs on per batch basis.

machinelearningmastery Batchnorm is proposed as a technique to help coordinate the update of multiple layers in the model. It is the best to use it before the AF that may result in non-gaussian distribution (ReLU) or after AFs if it is for s-shaped functions like the tanh and logistic function. BN makes the network more stable and as the result, it is possible to increase the learning rate to a higher value. BN can also make the initial weight initialization less important. At the same time, it is not recommended to use with dropouts; the reason behind it is due to statistics becoming noisy as the random nodes are dropped out during dropout.

So how does it actually work? *https://github.com/hiromis/notes/blob/master/Lesson6.md*

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

The algorithm is going to take a mini batch. Batch norm is the layer, so it will take in some activations that will be denoted as x_1, x_2, x_n. So the whole algorithm consists of 4 distinct steps: finding the mean with those activations; finding the variance of those activations; normalizing it; taking the resulting values and add a vector of biases (that are β here). And after that we will multiply the j by γ. Those two terms are learnable parameters. β is called a normal bias layer and γ is a multiplicative bias layer. The γ has a direct gradient to increase the scale and to change the gradient mean β is used. By doing that, it will be easy to shift the outputs up and down, in and out.

OPTIMIZATION

The primary objective is to minimize the loss function. As the goal of the objective function is a loss function that depends on the training data, the optimizer will reduce the training error. However, for statistical inference (and for our purposes), we need to reduce the generalization error. The major challenges are local minima, saddle points, and vanishing gradients.

Local minima

For the objective function $f(x)$, if the value of $f(x)$ at x is smaller than the values of that function at any other x, then $f(x)$ is a local minima. If that $f(x)$ corresponds to the minimum of the whole function, then it's a global minimum

Saddle points

A saddle point is any location where all gradietns of a function vanish but which is neither a global nor local minima. If we look at it from via a Hessian matrix, then solution of a function where the function gradient is zero:

- When the evalues of Hessian @ zero-gradient position are all positive, we will have a local minimum
- When evalues of Hessian @ zero-gradient position are all negative, we will have a local maximum
- When evalues of Hessian @ zero-gradient position are negative and positive, then we got a saddle point

Just a note; in convex functions, evalues of Hessian are never negative.

Vanishing gradients

Gradient will become 0. Prior to ReLu AF, training deep learning models were tricky

Convexity

Convexity plays an important role in optimization. If algorithms perform poorly with convex function, then there is no point in moving forward.

A set X in a vector space is convex if for any $a, b \in X$ the line segment connecting a, b is also in X. Mathematically, $\lambda \cdot a + (1 - \lambda) \cdot a \in X$ for all $a, b \in X$.

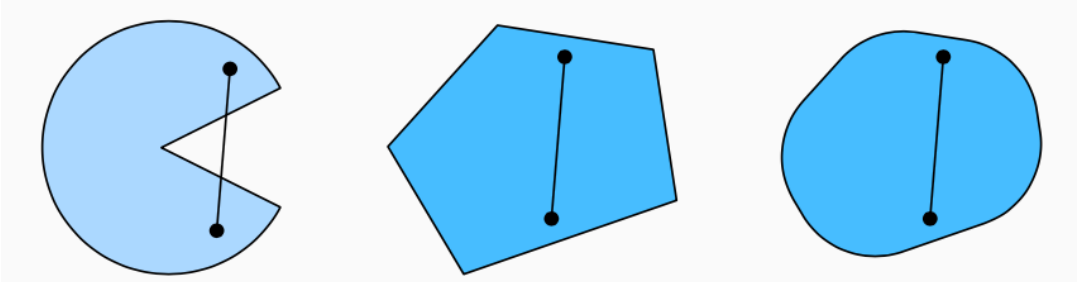


Fig. 10.2.1 Three shapes, the left one is nonconvex, the others are convex

If X, Y are convex, then $X \wedge Y$ are convex as well. If $a, b \in X \wedge Y$, the line segment connecting a, b also need to be contained in $X \wedge Y$.

The problems in deep learning are defined on convex domains. For convex functions local minima is a global minima.

Given a convex set X a function defined on it $f : X- > \mathbf{R}$ is convex if for all $x, x' \in X$ and for all $\lambda \in [0, 1]$, we have:

$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x')$

Jensen's Inequality It amounts to a generalization of the definition of convexity

Convex functions have several desirable properties.

No local minima

Convex functions do not have a local minima. That means that the optimization algorithm doens't get stuck.However, that doesn't mean that there is no other global/local minimums.

NOTE: Finish this part

Gradient Descent
There are three variants of gradient descent, which differ in amount of data needed to compute the gradient of the objective function

Batch Gradient Descent
Computes gradient descent of the cost function wrt/ to the parameters θ for the *entire* training set:

$$\theta := \theta - \eta \cdot \nabla J(\theta)$$

We perform gradient for the whole dataset using just *one* update and as the result, the process is quite slow. It also doesn't allow us to update the parameters online. So, if $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^i - y^i)$, we will load the whole dataset with m samples and process them before making a first step. As one cycle through the entire training dataset is known as epoch, the batch gradient descent updates the weights at the end of each training epoch. That does have some positives: decreased update frequency results in a more stable error gradient and may result in a more stable convergence on some problems; fewer updates means that it's more computationally efficient that SGD. Downsides are: loading the whole dataset into memory (possible RAM limitation); might result in a premature convergence to a less optimal set of parameters; too slow for larger datasets.

In batch-gradient descent, we update parameters in the direction of the gradients wrt. to the size of learning rate. Batch gradient will converge to the local minimum for non-convex surfaces, and to the global minimum for convex functions.

Stochastic Gradient Descent
SGD performs a parameter update for *each* training example

$$\theta := \theta - \eta J(\theta; x^{(i)}; y^{(i)})$$

So, if $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^i - y^i)$, the algorithm can be rewritten as and repeated for every training sample m:

$$\theta := \theta - \eta \cdot (\hat{(y)} - y) \cdot x_j$$

Since it performs the update for each training example, it is often called an online learning algorithm. SGD has advantages: might result in faster learning; frequent updates will give a fast understanding on how the model performs. The downsides are there too: frequent updates will result in the noise signal (will have a high variance); as there is a high variance, it will make it hard for the algorithm to settle on an error minimum for the model.

In SGD, the model is updated m times, since epoch means that we see every training instance once

Mini-batch Gradient Descent

Mini-batch is the most optimal approach. The training set is divided into n batches and updates are performed every n training examples:

$$\theta := \theta - \eta \cdot \nabla J(\theta; x^{i;i+n}; y^{i:i+n})$$

This algorithm can be rewritten in the following manner. Let's say batch size is 10 and we have 1000 samples. We will repeat the following algorithm for 1000 times:

$$\theta := \theta - \frac{1}{10} \sum_{k=i}^{i+9} (\hat{y}^k - y^k) \cdot x^k$$

MB has good and bad sides. Good: more computationally efficient; updates more frequently. Bad: another hyper-parameter (batch size). Also there are challenges: the same learning rate is applied to all parameter updates. If the matrix is sparse and features have different frequencies, that will lead to same update, when what we want is a larger update for rarely occuring features.

In mini-batch we update parameters after each batch. So, if we have m samples and batch size is of size b, parameters are updated about m/b.

All of the approaches need to be checked for proper working: if the loss function doesn't reduce, or exhibits the chaotic behaviour, then gradient descent doesn't work.

Momentum

Momentum is a method that accelerates SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta)$$

$$\theta_t = \theta_{t-1} - v_t$$

where v is the velocity term, the γ is the decaying hyperparameter that tells how quickly accumulated previous gradients will decay. If $\gamma \gg \eta$, the accumulated previous gradients will control the update rule Momentum term increases for dimensions whose gradient points in the same directions and reduces updates for dimensions whose gradients change directions.

Exponentially weighted averages: Deals with sequence of numbers. If we have a noisy sequence S, we want some kind of moving average that would denoise the data and make it closer to the original function. Explonentially weighted averages define a new sequence V with the following equation:

$$V_t = \beta \cdot V_{t-1} + (1 - \beta) \cdot S_t \qquad for \beta \in [0, 1]$$

where V is a new sequence and S is an original noisy sequence.

So the fact is that exponentially moving average depends on the previous terms, so if we expand the previous sequence to 3 terms, we get:

$$\begin{aligned} V_t &= \beta V_{t-1} + (1 - \beta) S_t \\ V_{t-1} &= \beta V_{t-2} + (1 - \beta) S_{t-1} \\ V_{t-2} &= \beta V_{t-3} + (1 - \beta) S_{t-2} \end{aligned}$$

So, if we plug-in all the terms, we will see that β terms for older sequences gets smallers and as the result, doesn't add much weight to calculation of a new sequence of V_t.

Essentially, SGD with momentum is a moving average of gradients and can be expressed as:

$$V_t = \beta \cdot V_{t-1} + (1 - \beta) \cdot \nabla J(theta; x; y)$$

$$\theta := \theta - \eta \cdot V_t$$

Nesterov Accelerated gradient
The idea is that the current parameter is at some position, we want to "look-ahead" and try to approximate what the parameters will be like, if we take the next step. Mathematically,

$$v_t = \gamma \cdot v_{t-1} + \eta \nabla J(\theta - \gamma v_{t-1})$$
$$\theta := \theta - v_t$$

The idea is that when we compute $\theta - \gamma \cdot v_{t-1}$, we get an approximation of the next position of the parameters and a rough idea of where our parameters are going to be.

The major difference between momentum and Nesterov is that with momentum we first compute the current gradient and then make a move; with NAG we make a move in the direction of previously accumulated gradient, measure the gradient and then correct it.

With NAG we can speed up SGD but it would be great to adapt learning parameters to each individual feature to perform larger or smaller updates

Adagrad
Adagrad adapts a learning parameter to the parameters, where larger updates are performed for infrequent and smaller updates for frequent parameters. This makes Adagrad suitable for sparse data.

Unlike in SGD, where we performed parameter updates for all parameters at once, in Adagrad we will use a different learning parameter for every parameter θ, at every time step t. in the following mathematical expressions, $g_{t,i}$ is the gradient of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

SGD updates every parameter θ at each time step becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

In its update rule, Adagrad modifies the general learning rate at each time step for every parameter θ, based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

In this equation, G_t is a diagonal matrix where each diagonal element i, i is the sum of squares of the gradients wrt/ θ up to time step t, while ϵ is a smoothing term that avoids division by zero.

One of the benefits of Adagrad is that it eleminates the need to manually tune the learning rate. Most implementations use the default learning rate of 0.01 and do not change it. The drawback is that since every added term is positive, accumulated sum keeps growing in the denominator during training. This causes lr to vanish. Adadelta addresses this problem.

Adadelta

Adadelta is an extension of Adagrad that battles the vanishing learning rate. Instead of accumulating *all* past squared gradients, Adadelta limits the accumulation to some predefined window of size w. Instead of storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ depends only on the previous *average* and the current gradient:

$$E[g^2]_t = \gamma \cdot E[g^2]_{t-1} + (1-\gamma) \cdot g_t^2$$

Re-writing SGD in slightly different notations:

$$\delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \delta\theta_t$$

Rewriting Adagrad:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

Replacing the diagonal matrix G_t with the decaying average over the past squared gradients:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

This formula can be rewritten with RMS, as the it is a denominator

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} \cdot g_t$$

Now, in the original paper authors promote another formula to match the units in the update. This new formula is another exponentially decaying average of *squared parameter updates*

$$E\Delta_t^2 = \gamma E[\Delta^2]_{t-1} + (1-\gamma) \cdot \Delta\theta_t^2$$

As the result, RMS is rewritten as:

$$RMS[\Delta\theta_t] = \sqrt{E[\Delta g^2]_t + \epsilon}$$

As that RMS is unknown, we approximate it with the RMS of parameter updates until the previous time step. We replace the learning rate in the previous update rule parameter with RMS yields the final update rule, that eliminates the need in η:

$$\theta_t = -\frac{RMS[\Delta]_{t-1}}{RMS[g]_t} \cdot g_t$$

Adam

Adaptive moment estimation (Adam) is another method, developed at UToronto, that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients (similar to Adadelta), Adam also keeps an exponentially decaying average of past gradients m_t:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Where m_t, v_t are estimates of the first moment (mean) and the second moment (uncentered variance) of the gradients respectively. Both m_t, v_t are initialized at vectors of 0, and in the original paper it was observed that they are biased towards 0, especially during the initial time steps and with small decay rates (β_1/β_2 are 1). Authors counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m_t} = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v_t} = \frac{v_t}{1-\beta_2^t}$$

Which results in the update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon}\hat{m_t}$$

INFORMATION THEORY

Entropy

Entropy is a measure of unpredictability (uncertainty) of information content. In the original paper, it is stated that entropy tells us the theoretical minimum average encoding size for events that follow a particular probability distribution. In ML, entropy of a distribution is the expected amount of information in an event drawn from that distribution. Mathematically, it's expressed as:

$$H = -\sum_i p(x_i) \cdot log_2(p(x_i))$$

This formula is known as Shannon Entropy. In the formula, i is a class; p_i is a fraction of examples in a given class. Entropy is used in machine learning in the following cases; central idea of cross-entropy; in policy gradient optimization in reinforcement learning.

The information part is: $I(E) = -log(P)$, where E is a stochastic event (tossing a coin). If we are dealing with information that is expressed in bits (0,1) then log has a base of 2. The expected value of variable X is given by: $E[X] = \sum_{i=1}^{n} x_i p_i$, where x_i is the possible value of x and p is the probability of that value occurring. In that case the main formula written above can be derived as: $H(X) = E[I(X)] = E[-log(P(X))] = -\sum_{i=1}^{n} P(x_i) log(P(X_i))$

As information theory is measured in bits, entropy will output the uncertainty in bits too. Entropy is zero, when there is 100% chance that an event will occur. If a coin is biased (there is a higher percentage that we will get heads), then entopy is the lowest. Entropy is the highest, if there is a high level of uncertainty; 50% chance that we will get a head or tail (unbiased coin) (highest uncertainty).

There are some properties related to entropy:

- Uniform distributions have the maximum uncertainty
- Uncertainty is additive for independent events: $H(X,Y) = H(X) + H(Y)$
- Adding an outcome with zero probability has no effect
- The measure of uncertainty is continuous in all its arguments. There are no gaps in the function. The smallest change in the input will cause the smallest change in the output. Log functions are continuous at every point for which they are defined. As the result, sums and products are continuous on a subset. From this it follows, that the entropy function is continuous in its probability arguments.

The only family of functions that satisfies those properties is: $H(p_1, ..., p_n) = -\lambda \sum_{i=1}^{n} p_i log(p_i)$, where $\lambda > 0$. This is known as Uniqueness theorem and if we set $\lambda = 1$, and use base 2 for log, we get Shannon entropy

The concept of entropy is closely related to Information Gain; IG measures how much information a feature gives us about the class. It is based on decrease in entropy (in decision trees, it happens after a dataset is split on an attribute): $IG(T, X) = Entropy(T) - Entropy(T, X)$

Cross-Entropy/Log Loss

$$H(p, q) = -\sum_{x} p(x) log(q(x))$$

The above-mentioned formula can be rewritten for binary classification as:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

where y is the target (true class) and $p(y)$ is the predicted probability (calculated class) of the points being a label we need (green dots)

Cross entropy between two probability distributions p and q over the same underlying set of events measure the average number of bits needed to identify an event drawn from the set, if a coding scheme is used that is optimized for an "unnatural" probability distribution q, rather than true distribution p.

In simple terms, imagine we have two classes: red and green dots (squares, anything). In binary classification, the task is simple: given x_i, predict the label. In this example, we can make green labels a positive class and red ones belong to the negative class. If we fit a model, it will predict the probability of being green to each one of our points. The whole point of the loss function is to evaluate how good are the predicted probabilities. If the probability associated with the true class is 1, then loss is zero.

In the entropy, if we the know the true distribution of the random variable $q(y)$, we can compute its' entropy (using the formula in the previous subsection). If we don't know that probability distribution, we cannot calculate entropy. So we will need to estimate the true distribution with some other distribution, $p(y)$. Let's assume that our points follow this distribution, while they actually come from the true distribution $p(y)$. If we compute the entropy using the first formula, then we are actually computing the cross-entropy between two distributions. If those two distribution, match perfectly then cross-entropy is 0. The smaller the cross-entropy error is, the better that model learned how to fit data.

Cross-entropy is used to measure classification error or measures the performance of a classification model whose output value is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. A perfect model would have a log loss of 0. When the actual observation is 1, and predicted label is 0, then the loss will be high.

If we look at the first formula, we can adapt it to the actual problem and understand the concept of two distributions being different. Let's say we have 3 classes and their true distribution is the following:

- Pr(Class A) = 0
- Pr(Class B) = 1

- Pr(Class C) = 0

After training the algorithm, we get the following predictions:

- Pr(Class A) = 0.228
- Pr(Class B) = 0.619
- Pr(Class C) = 0.153

How close is the predicted distribution to the true one? Using the first formula, we get:

$$H = -(0 * log_2(0.228) + 1 * log_2(0.619) + 0 * log_2(0.153)) = 0.692$$

This is how far (or how wrong) the prediction is from the true distribution.

We can minimize this loss function using one of the optimizers (SGD, Adam).

Kullback-Leibler Divergence

$$D_{KL}(p(X)||q(X)) = \sum_{x \in X} p(x) log(\frac{p(x)}{q(x)}) = H(p,q) - H(p)$$

What we are looking here is the expectation of the log difference between the probability data in the original distribution with the approximate distribution. Measuring the likelihood that samples represented by empirical distribution p is generated by fixed distribution q.

The relationship between KL and Cross entropy is

$$H(p,q) = H(p) + D_{KL}(p||q)$$

where $H(p,q)$ is the cross entropy and $H(p)$ is the entropy. If $H(p)$ is constant, then minimizing the KL is essentially the same as minimizing cross-entropy. It can be constant if we are given a dataset $P(D)$ which represents a problem to be solved, and the learning purpose is to make the model estimated distribution $P(model)$ as close to $P(D)$ as possible. In most cases, the underlying distribution is given and it doesn't change, and $H(p)$ is constant. So the problem ends up being minimization of cross-entropy.

KL can also be viewed as the extra amount of information needed to send a message containing symbols drawn from true distribution P, when we use an optimized model distribution Q. In other words, it can be viewed as the distance between optimized distribution $q(x)$ and the true distribution $p(x)$.

KL is not symmetric: $D_{KL}(p(x)|q(x) \neq D_{KL}(q(x)|p(x)$

Joint Entropy

$$H(X,Y) = -\sum_{x,y} p(x,y) log(p(x,y))$$

Joint entropy is the entropy of a joint probability distribution. Joint entropy is no different from regular entropy; we just have to compute entropy over all possible pairs of two random variables

Mutual Information

$$I(X;Y) = \sum_{x,y} p(x,y) \cdot \frac{log(p(x,y)}{p(x)p(y)}$$

It's a measure of the mutual dependence between two variables. It quantifies the amount of information obtained about one random variable through observing the other random variable. Mutual Information is also known as information gain. Mutual information determines how similar the joint distribution of pair (x, y) is to the product of the marginal distribution of x and y. MI captures dependency between random variables and it's more generalized than correlation, that captures only the linear relation. This is very useful in feature selection, when non-linear dependencies can be explored. Also, in Bayesian Network, MI is used to learn relationship structure between random variables and define the strength of those relationship.

DENSITY ESTIMATION

Parametric methods assume that the data is normally distributed. Non-parametric methods don't and density estimation techniques are mostly non-parametric methods.

Density estimation is the learning of $p(x)$ given examples of x.

The Kernel density estimation is a non-parametric way to estimate the probability density function of a random variable. In comparison to parametric estimators where the estimator has a fixed functional form and the parameters of this function are the only information we need, non-parametric assumes no fixed structure and depends fully on the given data.

KDE is useful when it comes down to historgrams; when we construct them, we need to consider the width of bins and the end points of the bins. As a result, the major problem with histograms is that they depend on width of bins and the end points of the bins. Those problems are solvable with KDE. KDEs construct kernel at each data point. If we use a smooth kernel function for the building block, we will have a smooth density estimate. Kernel estimators smooth out the contribution of each observed data point over a local neghborhood of that data point. The contribution of the data point x_i to the estimate at some point x^* depends on how apart x_i and x^* are. The extent of this contribution depends upon the shape of the kernel function adopted and the width of that kernel function. If we denote the kernel function as K and its' bandwidth as h, then estimated density at any point x is:

$$\hat{f(x)} = \frac{1}{n} \sum_{i=1}^{n} K(\frac{x - x_i}{h})$$

To yield meaningful estimates, a kernel function must satisfy three properties:

- Symmetric about 0
- $\int K(u)du = 1$
- $\int u^2 K(u)du > 0$

There are several kernels that can be used (even though Gaussian is a popular one)

- Uniform: $\frac{1}{2} I(|u| \leq 1)$
- Triangle: $(1 - |u|)I(|u| \leq 1$
- Gaussian: $\frac{1}{\sqrt{2\pi}} \exp{(-\frac{1}{2}u^2)}$