Deep Learning: Lecture 1

Keith L. Downing

The Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
keithd@idi.ntnu.no

January 12, 2020

Outline

Gradient-Based Optimization (GBO)

- L = objective function.
- In GBO, use partial derivs of L (w.r.t. system params, W) to determine intelligent modifications of W so as to maximize or minimize L.

Simple Example

- Ψ = set of cases; each case is a pair (a,b)
- W = parameters of the system. W = (x,y)
- L(a,b) = ax + by = the objective function
- Goal = Maximize $\sum_{k \in \Psi} L(a_k, b_k)$
- Problem: You will never see complete contents of Ψ.
- Solution: Best alternative = Tune W to maximize L for the **samples** you do see. For each case (a_j, b_j), nudge x and y in directions that increase L(a,b) for that case. Size of nudge = learning rate = η.
- Mathematically, use gradients:
  - $\triangle x = \eta \frac{\partial L(a,b)}{\partial x}|_{(a_j,b_j)} = \eta a_j$
  - $\triangle y = \eta \frac{\partial L(a,b)}{\partial y}|_{(a_j,b_j)} = \eta b_j$

Gradient Descent Learning (GDL)

- L = loss or cost function.
- In GDL, use partial derivs of L (w.r.t. system params, W) to determine intelligent modifications of W so as to minimize L.

Simple Example

- Ψ = set of cases, c_i = $(f_{i,1}, f_{i,2}, ...f_{i,m} \; ; \; t_i)$ = features + target value.
- W = parameters (e.g. weights) of the system = a vector.
- $P(f_i) = f_i \bullet W$ = the prediction function
- Goal = Minimize L(Ψ) = **loss** function = $\sum_{k \in \Psi} |P(f_k) - t_k|$
- Problem: You will never see complete contents of Ψ, but you want to make good predictions $\forall c \in \Psi$
- Solution: Best alternative = Tune W to minimize L for the **samples** you do see. For each case $c_k = (f_k, t_k)$, nudge each w_i in direction that decreases $L(c_k)$. Size of nudge = learning rate = η.
- Mathematically, use gradients:
  - $\triangle w_i = -\eta \frac{\partial L(c)}{\partial w_i}|_{(f_k, t_k)} = \eta$?????? (see next slide)

Differentiating the Loss Function

Mean Squared Error (MSE) = a common loss function:

$$L(\Psi) = \frac{1}{\|\Psi\|} \sum_{k \in \Psi} (P(f_k) - t_k)^2$$

By the Chain Rule of calculus:

$$\frac{\partial L(\Psi)}{\partial w_i} = \frac{1}{\|\Psi\|} \sum_{k \in \Psi} 2(P(f_k) - t_k) \frac{\partial (P(f_k) - t_k)}{\partial w_i} = \frac{2}{\|\Psi\|} \sum_{k \in \Psi} (P(f_k) - t_k) f_{k,i}$$

since $\frac{\partial P(f_k)}{\partial w_i} = f_{k,i}$ and $\frac{\partial t_k}{\partial w_i} = 0$.

When we only have a **sample** S of all the cases in Ψ, we estimate the partial derivative (a.k.a. gradient) as:

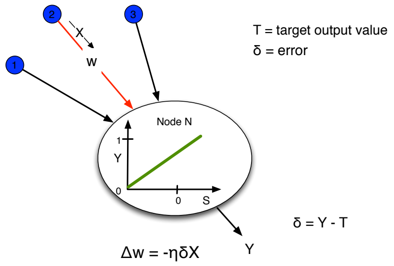
$$\frac{\partial L(S)}{\partial w_i} = \frac{2}{\|S\|} \sum_{k \in S} (P(f_k) - t_k) f_{k,i}$$

And we update each $w_i \in W$ to try to reduce this loss:

$$\triangle w_i = -\eta \frac{2}{\|S\|} \sum_{k \in S} (P(f_k) - t_k) f_{k,i}$$

The 2 in $\frac{2}{\|S\|}$ is usually ignored and implicitly incorporated into the chosen value of η

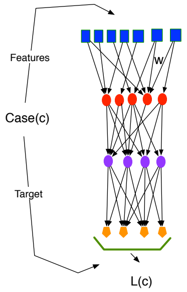
Complexifying Neural Networks



How do these networks get more complex?

- Nonlinear activation functions (applied to sum of weighted inputs)
- Multiple output neurons
- Multiple layers of neurons separating inputs from outputs.
- Various types of layers
- More elaborate objective (loss) functions.
- .... and much more ....

Amazingly Broad Applicability of Gradient Descent



- Despite increasingly complex NNs, the essence of gradient descent learning remains the same: for EVERY weight w_i in the network:

$$\triangle w_i = -\eta \frac{\partial L(c)}{\partial w_i}$$

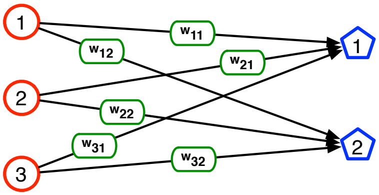
- Calculating these gradients just gets more complicated.
- ML packages such as Tensorflow, PyTorch and Theano automate this !!

Two Basic Types of Gradient Descent Learning

- Batch Gradient Descent (GD)
  - Run ALL available training cases (S) (a *batch*) through NN.
  - Calculate gradients **for each case** in S.
  - Update weights, but only after entire batch is run and all gradients are combined (e.g. averaged).
  - Repeat
  - * Each processing round (over an entire batch) = an epoch
- Stochastic Gradient Descent (SGD)
  - Run a **subset** C (often randomly selected) of S (a minibatch) through NN.
  - Calculate gradients for each $c \in C$
  - Update weights based on combined gradients.
  - Repeat
  - * Potentially unstable for small $\|C\|$.
  - ** For SGD, an epoch = $\frac{\|S\|}{\|C\|}$ minibatches.

For GD (SGD), each case in a batch (minibatch) experiences the same weights when run through the net.

Matrices for Neural Networks



$$\begin{array}{cccc} \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} & \times & \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} & = \\ \text{Output Values} & & \begin{array}{cc} 1 & 2 \\ \text{Incoming weights} & \text{Incoming weights} \end{array} & \end{array}$$

[1 x 3] x [3 x 2] => [1 x 2]

$$\begin{bmatrix} x_1 w_{11} + x_2 w_{21} + x_3 w_{31}, & x_1 w_{12} + x_2 w_{22} + x_3 w_{32} \end{bmatrix}$$

Sum weighted inputs to 1 Sum weighted inputs to 2

Standard Representations

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

* Standard notation: activation vectors are COLUMN vectors:

3 x 1 Column Vector

$$\overset{X^T}{\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}} \overset{W}{\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}} = X^T W$$

1 x 3 Row Vector

3 x 2 Matrix

1 x 2 Row Vector

$$X^T W = (W^T X)^T$$

Transposed Weight Matrices

Transpose all weight matrices (just once) and then use $W^T X$ to produce new **column** vectors. Then activation vectors need not be transposed back and forth between column and row vectors.

$$\underset{\text{2 x 3 Matrix}}{\overset{W^T}{\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}}} \underset{\text{3 x 1 Column Vector}}{\overset{X_0}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}} = \overset{\text{2 x 1 Column Vector}}{W^T X_0}$$

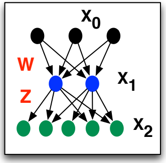
2 x 1 **Column** Vector $\quad X_1 = F_{act}(W^T X_0)$

F$_{act}$ and G$_{act}$ = activation funcs that are applied to each tensor element. They do not alter its shape.

Z = a 2 x 5 Weight Matrix

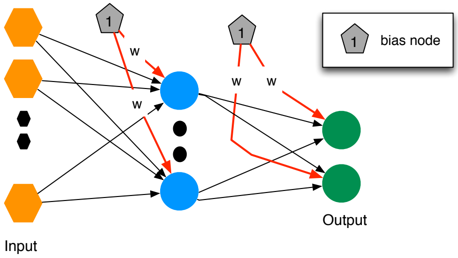
5 x 1 **Column** Vector $\quad X_2 = G_{act}(Z^T X_1)$

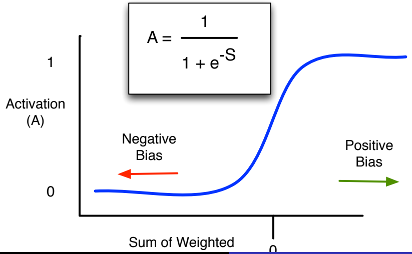
$$X_2 = G_{act}(Z^T F_{act}(W^T X_0))$$



Bias Nodes

A bias node has a constant output of 1 but has K independent output weights, one per node in the layer. These weights are modified by gradient descent just like all other weights. However, they are often initialized to zero.





Adding a Column Vector for Biases

- B_w = a column vector of biases, one per node in the layer that produces X_1.
- B_z = a column vector of biases, one per node in the layer that produces X_2

$$X_1 = F_{act}(W^T X_0 + B_w)$$
$$X_2 = G_{act}(Z^T X_1 + B_z)$$

... or ....

$$X_2 = G_{act}(Z^T F_{act}(W^T X_0 + B_w) + B_z)$$

Using a MiniBatch of Inputs

- Instead of an m x 1 column vector (X_0) as input, use a minibatch (C) of n column vectors. C = m x n matrix.
- Every activation vector (X, originally a q x 1 column vector) now becomes a q x n matrix (χ), one column per minibatch item.
- The q x 1 bias vectors (B) become q x n matrices (β), where β = n copies of B. This is called **broadcasting**: expanding a matrix along one or more dimensions via copying.
- **The weight matrices and activation functions remain the same.**

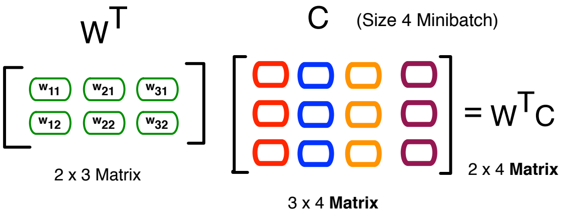
$$\chi_1 = F_{act}(W^T C + \beta_w)$$

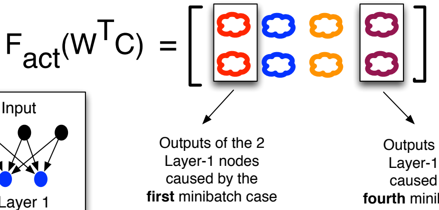
$$\chi_2 = G_{act}(Z^T \chi_1 + \beta_z)$$

... or ....

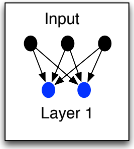
$$\chi_2 = G_{act}(Z^T F_{act}(W^T C + \beta_w) + \beta_z)$$

Forward Propagation of a Minibatch



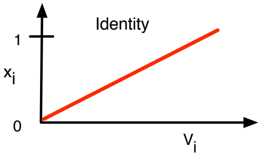


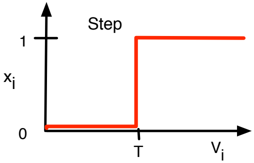
$F_{act}(W^T C)$ =
Outputs of the 2 Layer-1 nodes caused by the **first** minibatch case
Outputs of the 2 Layer-1 nodes caused by the **fourth** minibatch case

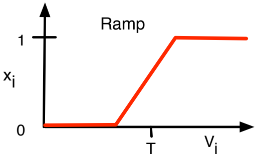


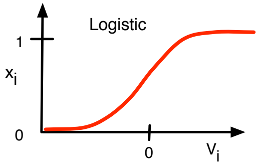
Most tensor calculations, including those deriving gradients, work seamlessly with the extra tensor dimension due to the minibatch

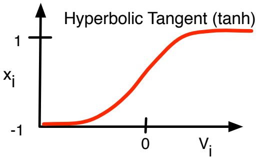
Classical Activation Functions









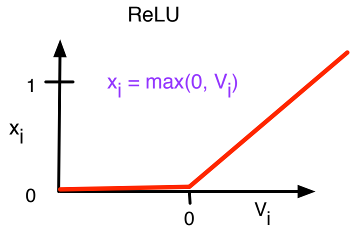


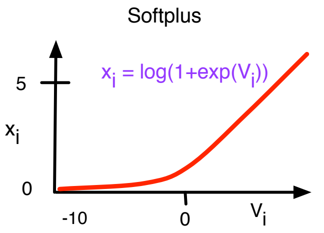
The Sigmoid Activation Function

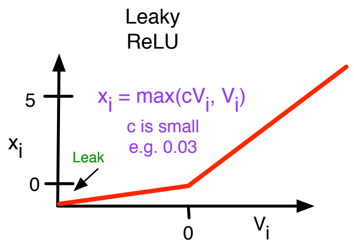
- Sigmoid (a.k.a. logistic) function **was** very popular for NNs, because a) common in biological systems, and b) implements a step/ramp but is continuous at the threshold, thus simplifying gradient calculations.
- But the sigmoid **saturates** for inputs of large (pos or neg) magnitude. These flat regions of the sigmoid curve have near-zero derivatives.
- Once a sigmoid saturates, the weights leading into that node have near-zero gradients $\rightarrow$ they change very little $\rightarrow$ learning halts.
- Hence, sigmoids are a poor choice for **hidden** nodes, particularly in deep networks.
- However, they are still very useful for **output** nodes, especially when combined with an objective function that reduces the risk of saturation (as discussed later).
- Also very useful in NNs that do not do backpropagation, and thus do not use gradients.
- The hyperbolic tangent (tanh) resembles the sigmoid, but since tanh(0) = 0 and it can also output negative values, it behaves a little more like an identity function (for small inputs). This makes it a bit easier to train.

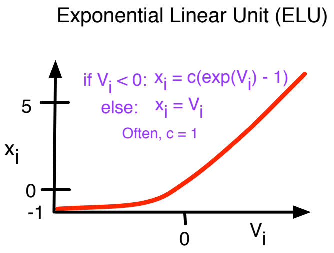
Contemporary Activation Functions

These have **greatly** improved the performance of NNs.

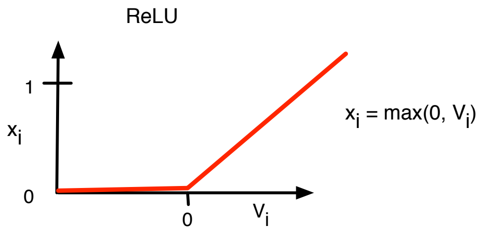








The Rectified Linear Unit (ReLU)



- Other activation functions yield error gradients that are strongly attenuated during backprop, such that only the last (closest to output) few layers of weights adapt properly.
- ReLUs permit sustained influence of gradients upon all weights throughout the network.
- By making many activation levels a hard zero, ReLUs sparsify activation patterns, which is more biologically realistic and better for learning.

Deep sparse rectifier neural networks, Glorot et. al., 2011

Comparing Gradients: Sigmoid -vs- ReLU

- v = V_i = sum of inputs to the activation function
- x = x_i = output of the activation function

f_T = Rectified Linear Unit (ReLU): $f_T(v)$ = max(0,v)

$$\frac{\partial f_T(v)}{\partial v} = 1 \quad \text{when } v > 0$$

$$\frac{\partial f_T(v)}{\partial v} = 0 \quad \text{otherwise}$$

The gradient is **significant** for all $v > 0$

f_T = Sigmoid: $f_T(v) = \frac{1}{1+e^{-v}}$

$$\frac{\partial f_T(v)}{\partial v} = x(1-x)$$

- This has max = 0.25 (when x = 0.5 and v = 0). When the sigmoid saturates, it is much lower: $\frac{\partial f_T(v)}{\partial v} = 0.048$ when x = 0.95 (or x = 0.05).
- These derivs get multiplied repeatedly (via Chain Rule) as backprop moves upstream, so products can get very small, very quickly.

Improvements to ReLU

ReLU Problems and Solutions

- Dying Neurons: When sum of inputs (V_i) is non-positive, the neuron outputs a zero AND its gradient is zero, so nothing changes. Inactive neurons often fail to come back to life.
- Solution: Leaky ReLU: Outputs may be small negative values, but these keep the neuron semi-active and yield non-zero gradients, which can eventually revive the neuron. Variations include Randomized Leaky ReLU (RReLU) and Parametric Leaky ReLU (PReLU).
- Instability: Gradients bounce around from positive to zero (when $V_i \approx 0$) due to discontinuity of ReLU and Leaky ReLU at 0, where deriv from right = 1 but deriv from left = 0.
- Solution: Exponential Linear Unit (ELU): This is negative for $V_i < 0$ (like the Leaky ReLU) but is continuous at $V_i = 0$, thus reducing oscillations of the gradient.

Softmax - For Classification Problems

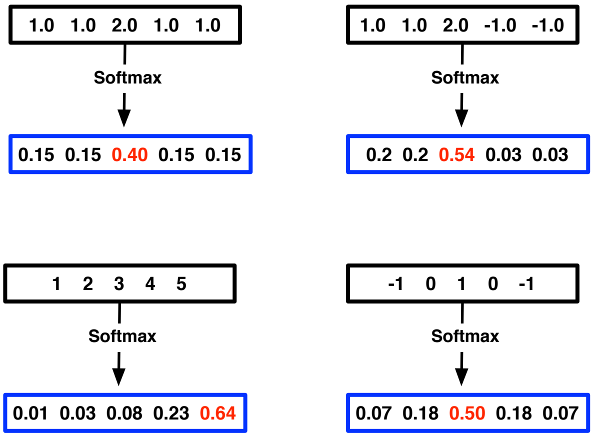
- Many classification problems benefit from an output layer that represents a probability distribution over the possible classes.
- Hence, all outputs must be non-negative, and they must sum to 0.
- No individual activation function can enforce this summation condition.
- Outputs must be combined and scaled (Boltzmann scaling).
- For an output layer of size N with a vector of output values, x_i:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{N} e^{x_k}}$$

- Note how this easily combines both positive and negative x_i, insuring that every softmax'd value is positive.
- Softmax embodies **competition** among the outputs, thus modelling lateral inhibition found in many brain regions.

Scaling Outputs with Softmax

Accentuating strength and weakness of output activations

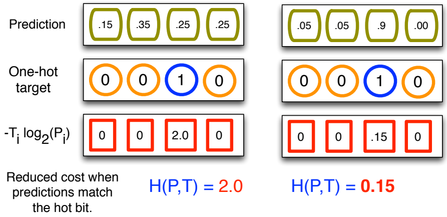


Cross Entropy as a Loss Function

- Useful when targets and outputs both represent probability distributions. So combine softmax'd outputs with cross-entropy loss function.
- When combined with a sigmoid, the log counteracts the exp to reduce saturation (and gradient decay).
- The standard usage: targets = 1-hot vectors and outputs are a normalized distribution over class probabilities.

$$\text{Cross Entropy: } H(P,T) = -\sum_{k=1}^{\|T\|} T_k log(P_k)$$

where P = network's output (prediction) and T = target vector.



Cross Entropy to Compare 2 Distributions

Prediction	.15	.35	.25	.25
Target	.8	.05	.05	.1
$-T_i \log_2(P_i)$	2.2	.08	0.1	0.2

Pay heavy cost when a high-probability target is not matched by the prediction

Pay much lower cost when a low probability target is not matched.

$H(P,T) = 2.58$

Prediction	.8	.1	.09	.01
Target	.8	.05	.05	.1
$-T_i \log_2(P_i)$	.26	.17	.17	.67

$H(P,T) = \mathbf{1.27}$

Cross Entropy for Logistic Regression

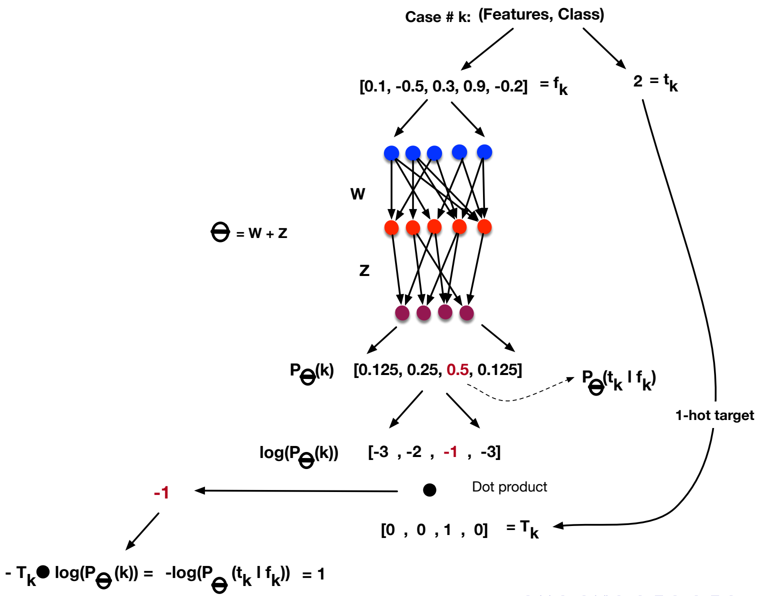
Logistic (or Logit) Regression: Single-value prediction where output = prob. of class membership. E.g. Insurance risk or not?

- The output node typically uses a sigmoid activation (a.k.a. logistic function), whose output value (in range (0,1)) represents the probability of class membership, p_k, for the kth case.
- Each target value, t_k, is binary.
- Cost function, called Log Loss applied to minibatch C:

$$L(C) = \frac{-1}{\|C\|} \sum_{k \in C} t_k log(p_k) + (1 - t_k) log(1 - p_k)$$

- Depending upon t_k, only one of the 2 terms is non-zero for each case, k:
  - $t_k = 1$: p_k should be **high** to reduce cost.
  - $t_k = 0$: p_k should be **low**, so $(1 - p_k)$ is high, to reduce cost.
- Same principle as cross entropy, but now the single output node represents **two** classes: YES or NO. When the target = NO, the second term of the sum kicks in, asking, ”How close is the prediction to a NO?”

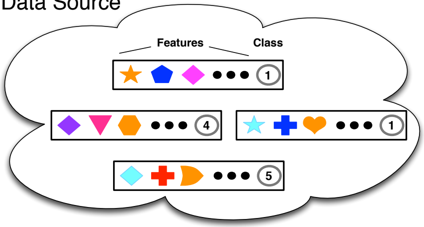
Cross Entropy Loss Overview



Case # k: (Features, Class)
[0.1, -0.5, 0.3, 0.9, -0.2] = f_k
2 = t_k
W
Θ = W + Z
Z
P_Θ(k) [0.125, 0.25, 0.5, 0.125]
P_Θ(t_k | f_k)
1-hot target
log(P_Θ(k)) [-3 , -2 , -1 , -3]
-1
●
Dot product
[0 , 0 , 1 , 0] = T_k
- T_k ● log(P_Θ(k)) = -log(P_Θ(t_k | f_k)) = 1

Maximum Likelihood

Data Source



The Modeling Objective

- Given a data source, S, build a model that best captures the statistical relationships (between features and classes) of S.
- View S as one big state, then the model (θ) should maximize the probability of that big state, i.e., the probability of ALL cases in the state being true:

$$\pi(S) = \prod_{k \in S} p(f_k, t_k) = \prod_{k \in S} p(t_k | f_k) p(f_k)$$

- where f_k = features and t_k = class (target) for case k.

Conditional Maximum Likelihood

- The model (θ) will estimate $p(t_k|f_k)$ with $p_\theta(t_k|f_k)$. E.g. θ are the weights and biases of a neural network that takes f_k as input and produces an output vector of probabilities (P_k), one for each class.
- θ and $p(f_k)$ are independent, as the latter are determined solely by S.
- We want to choose θ to maximize:

$$\pi_\theta(S) = \prod_{k \in S} p_\theta(t_k|f_k)p(f_k)$$

- This is equivalent to maximizing its log:

$$log(\pi_\theta(S)) = \sum_{k \in S} log(p_\theta(t_k|f_k)) + log(p(f_k)) = \sum_{k \in S} log(p_\theta(t_k|f_k)) + \sum_{k \in S} log(p(f_k))$$

- The latter (red) sum is independent of θ, so maximizing $\pi_\theta(S)$ is equivalent to maximizing the former (blue) sum.
- The Conditional Maximum Likelihood Estimator is defined as:

$$\theta_{ML} = argmax_\theta \sum_{k \in S} log(p_\theta(t_k|f_k))$$

- I.e., the θ that maximizes the log output values of nodes corresponding to the **correct** class for each case.

Cond Max Likelihood and Cross Entropy

- Assume each case $k \in S$ – or, more likely, a sample ($\tilde{S}$) of S – is run through the model (with parameters θ) to produce P_k. Take the log of each output, yielding $log(P_k)$, which is then compared to a 1-hot target vector (T_k) that encodes the correct class. This comparison of outputs to targets is done via a vector dot product, which selects 1 value.
- Thus, for each case, $T_k \bullet log(P_k) = log(p_\theta(t_k|f_k))$, so:

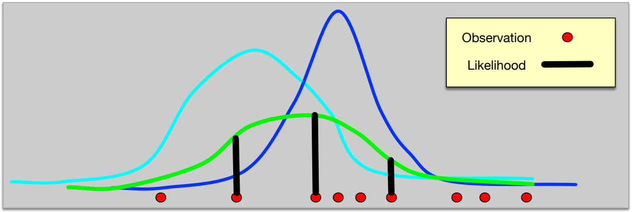
$$\sum_{k \in \tilde{S}} T_k \bullet log(P_k) = \sum_{k \in \tilde{S}} log(p_\theta(t_k|f_k))$$

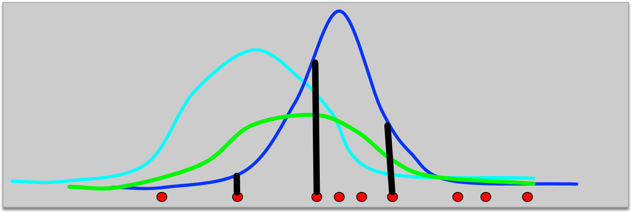
- The same θ_{ML} maximizes both, and minimizes the negation:

$$-\sum_{k \in \tilde{S}} T_k \bullet log(P_k) = \text{cross-entropy(T,P)}$$

- Thus, the conditional maximum likelihood estimate (θ_{ML}) minimizes the cross entropy, a common objective function for classification problems. It also minimizes the mean squared error (MSE) for regression problems (details to follow).
- Cross entropy and MSE are very popular loss functions for DL.

Max Likelihood of Observations





Goal: Find the parameters (μ, σ) of a distribution (i.e. Gaussian / Normal) that maximizes the likelihood of all data/observations.

Calculating Maximum Likelihood Estimate (MLE)

The Normal (Gaussian) Distribution

$$p(x_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$$

Maximum Likelihood of a Gaussian

Given data $(x_i, ..., x_n)$, MLE = values of μ and σ that maximize this:

$$\prod_{i=1}^{n} p(x_i) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$$

It's simpler, and equivalent, to find μ and σ that maximize its log:

$$log(\prod_{i=1}^{n} p(x_i)) = \sum_{i=1}^{n} log(p(x_i)) = \sum_{i=1}^{n} -log(\sigma) - \frac{1}{2} log(2\pi) - \frac{(x_i-\mu)^2}{2\sigma^2}$$

$$= -nlog(\sigma) - \frac{n}{2} log(2\pi) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i-\mu)^2 = LogL$$

Calculating MLE (2)

Finding the MLE: $\hat{\mu}$ and $\hat{\sigma}$

Setting $\frac{\partial LogL}{\partial \mu} = 0$ and solving for μ, yields:

$\hat{\mu} = \tilde{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i =$ The sample mean

Similarly

Setting $\frac{\partial LogL}{\partial \sigma} = 0$ and solving for σ, yields:

$\hat{\sigma} = \tilde{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} =$ The sample standard deviation

So the MLE for mean and standard deviation are just the mean and standard deviation of our original data !!

Substituting $(\tilde{\mu}, \tilde{\sigma})$ for (μ, σ) in LogL

$$-nlog(\tilde{\sigma}) - \frac{n}{2}log(2\pi) - \frac{1}{2\tilde{\sigma}^2}\sum_{i=1}^{n}(x_i - \tilde{\mu})^2 = -nlog(\tilde{\sigma}) - \frac{n}{2}log(2\pi) - \frac{n\tilde{\sigma}^2}{2\tilde{\sigma}^2}$$

$= -nlog(\tilde{\sigma}) - \frac{n}{2}(log(2\pi) + 1)$

Calculating MLE (3)

To maximize LogL:

LogL $= -n log(\tilde{\sigma}) - \frac{n}{2}(log(2\pi) + 1)$

you need to minimize $\tilde{\sigma}$, where:

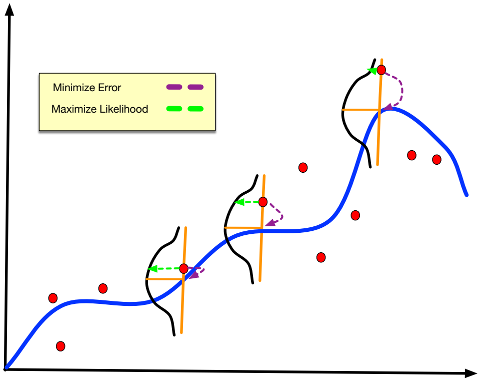
$\tilde{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \tilde{\mu})^2}$

That's the same as minimizing the mean-squared error:

MSE = $\frac{1}{n}\sum_{i=1}^{n}(x_i - \tilde{\mu})^2$

$\Rightarrow$ Maximizing the Likelihood of Data = Minimizing that data's Mean Squared Error

MLE and MSE for Regression



- f = the regression function (blue)
- By maximizing the likelihood of the observations (by choosing a good function), we minimize MSE ∝ distance from observation (x_i, y_i) (red point) to $(x_i, f(x_i))$.
- For regression problems, MLE = the function that minimizes the MSE.