Hyper Parameters, Theory and Equations

```
- Thomas Ehling
```

Loss

1.

Equations :

$$L1 = \sum_{i=1}^{n} |\hat{y}^{(i)} - y^{(i)}|$$

$$L2 = \sum_{i=1}^{n} (\hat{y}^{(i)} - y^{(i)})^2$$

$$Hubert : \sum_{i=1}^{n} L_\delta = \begin{cases} \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2 & \text{for } |(y^{(i)} - \hat{y}^{(i)})| \le \delta, \\ \delta(|(y^{(i)} - \hat{y}^{(i)})| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

$$Log - cosh : L_i(\theta) = \sum_{i=1}^{n} log(cosh(\hat{y}^{(i)} - y^{(i)})$$

Analysis :

L1 is the loss corresponding to the MAE, Mean absolute Error where L2 is corresponding to MSE, Mean Square Error.

L1 Loss will be more robust to outliers, but the gradient is the same throughout, which means the gradient will be large even for small loss values. This isn't good for learning.

L2 Loss will have a greater variance and will be more sensitive to outliers, but the gradient loss is high for larger loss values and decreases as loss approaches 0, making it more precise at the end of training.

Hubert Loss is less sensitive to outliers in data than the squared error loss. It's also differentiable at 0. The choice of delta is critical because it determines what is to be considered as an outlier.
The problem with Huber loss is that we might need to train hyperparameter delta which is an iterative process.

Log-cosh is another function used in regression tasks that's smoother than L2. It has all the advantages of

Huber loss, and it's twice differentiable everywhere, unlike Huber loss.

2.

We want to maximize the Log Likelyhood :

$$L(\theta) = \prod_{i=1}^{m}\prod_{j=1}^{k}(P(y=j|X^{(i)}))^{y_j^{(i)}}$$

It is equivalent to minimizing the Negative Log Likelyhood :

$$l(\theta) = -logL(\theta) = -\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)} log(\hat{y}_j^{(i)})$$

And wich lead to the Cross-Entropy Loss :

$$L_i(\theta) = -\sum_{j=1}^{k} y_j^{(i)} log(\hat{y}_j^{(i)})$$

For a Random class assignment, the worst cross entropy is :

$$P(y=j|X^{(i)}) = 1/k -> -log(P(y=j|X^{(i)}) = k$$

3.

1. SoftMax loss consists in a SoftMax activation function on the last layer and cross-entropy loss:

$$L_{t,i}(\theta) = -log(\frac{e^{\hat{y}_t^{(i)}}}{\sum_j^k e^{\hat{y}_j^{(i)}}})$$

For the t^{th} output, with the i^{th} entry for k classes.
It is used to have an output as a probability for multiclass classification: each output is in [0,1] and the sum of the outputs equals 1.

4.

Kullback-Liebler Loss:

$$KL(p||q) = -\sum_{i=1}^{m} p(X_i)log(p(X_i)) - \sum_{i=1}^{m} P(X_i)log(q(X_i))$$

The KL Loss (also called relative entropy) is a measure of how one probability distribution is different from a second. In the simple case, a Kullback–Leibler divergence of 0 indicates that the two distributions in question are identical.

When $p(X_i) = y^{(i)}$, $q(X_i) = \hat{y}^{(i)}$, KL is equivalent to cross entropy.

5.

Hinge loss:

$$L_i(\theta) = \sum_{j=1}^{k} max(0, \hat{y}_j^{(i)} - \hat{y}_t^{(i)} + 1)$$

The Hinge Loss sum over inccorrect class labels. Inside the max function, it will be Positive if $\hat{y}_t^{(i)} < \hat{y}_j^{(i)} + 1$ (Incorrect), and Negative if $\hat{y}_t^{(i)} > \hat{y}_j^{(i)} + 1$ (coorect) and therefore 0 will be the max.

The squared Hinge Loss is :

$$L_i(\theta) = \frac{1}{2} \sum_{j=1}^{k} max(0, \hat{y}_j^{(i)} - \hat{y}_t^{(i)} + 1)^2$$

The Squzred Hinge loss is smoother.

The worst value we can expect is infinite, as the possible Hinge loss values are : $[0; \infty]$

6.

	X_1	X_2	X_3
$\hat{y}_1$	0.5	0.4	0.3
$\hat{y}_2$	1.3	0.8	-0.6
$\hat{y}_3$	1.4	-0.4	2.7
y	3	2	3

$L_1 = max(0, 1.3 - 1.4 + 1) + max(0, 0.5 - 1.4 + 1) = 0.9 + 0.1 = 1$

$L_2 = max(0, 0.4 - 0.8 + 1) + max(0, -0.4 - 0.8 + 1) = 0.6 + 0 = 0.6$

$L_3 = max(0, 0.3 - 2.7 + 1) + max(0, -0.6 - 2.7 + 1) = 0 + 0 = 0$

7.

Regularization is used to measure the complexity of the network, in order to find the simplest model possible.

Usually, we train without regularization and then add it in order to obtain better result, because a simpler model tends to less overfit.

Loss = Data Loss + $\lambda R(\theta)$

With $R(\theta)$ being the regularisation and λ is the regularization weight coefficient in order to control how important our regularization is. The higher λ is, the simpler our model will be.

L1 reg : $R(\theta) = \sum_{i,j} |\theta_{i,j}|$
L2 reg : $R(\theta) = \sum_{i,j} \theta_{i,j}^2$

L1 makes weights sparse (concentrate weights)
e.g. |0.5| + |0.5| => |1| + |0|

L2 make weights smaller while spreading :
e.g. $1^2 + 0 \Rightarrow 0.5^2 + 0.5^2$

8.

L1 and L2 loss terms affect gradients in the network by introducing weights decay.

For L2 during gradient descent, we will always subtract $\lambda\theta$.

For L1 during gradient descent, we will always subtract $\lambda sign(\theta)$. (Some may increase, that is the concentration in some coefficient)

9.

The Regularization in Keras :

$$\hat{y} = \sigma(Wx + b)$$

The kernel regularization is to regularize the W.

The bias regularization is to regularize the b, Useful if we know we will have low values.

The activity regularization is to regularize the output of the unit. Useful if the output of the output layer will be close to zero.

All these regularization are done by introducing a weight decay.

Optimization

1.

Direct computation of gradient is easy to compute but really slow. We use Back Propagation because it is easy to compute too, it use symbolic derivatives in python, and it is much faster.

A possible use of direct numerical computation is verification, gradient check on simplified network.

2.

GD is :

1. Start with initial guesd θ_0
2. Repeat :

$$\theta^{(i+1)} < -\theta^{(i)} - \eta\nabla L(\theta^{(i)})$$

3. Stop when :

$$(L(\theta^{(i+1)}) - L(\theta^{(i)})) < \tau$$

SGD is :

1. Randomly order examples
2. Repeat :

$$\theta^{(i+1)} < -\theta^{(i)} - \eta\nabla L(\theta^{(i)})$$

The Gradient descent will converge faster, however it will likekely converge to a local minimum.

3.

The tradeoff for batch size is :

- updating after processing every example : faster but less accurate
- updating after all examples : more accurate but slower

The ultimate solution is to select a mini-batch, that means updating after processing after each mini batch.

The 4 main problems for SGD are :

1. What should be the laerning rate ?
2. What happens if loss is more sensitive to own parameter ?
3. How to avoid getting stuck in a local minima ?
4. Mini-batch gradient estimates are noisy, how to fixe that ?

4.

Smooth out changes to the gradient using momentum :

$$V^{(i+1)} < -\rho V^{(i)} + \nabla(\theta^{(i)})$$

$$\theta^{(i+1)} < -\theta^{(i)} + \eta(V^{(i+1)})$$

For ordinary SGD : ρ=0

Advantages :

- **local minimum / saddle**: there is velocity vector even at such locations
- **noisy gradients**: smoother out by moving average
- **poor conditionning**: smoothed out by averaging with previous gradients.

5.

NAG :

$$V^{(i+1)} < -\rho V^{(i)} + \nabla(\theta^{(i)})$$

$$\theta^{(i+1)} < -\theta^{(i)} + V^{(i+1)} + \rho(V^{(i+1)} - V^{(i)})$$

The acceleration is $(V^{(i+1)} - V^{(i)})$ (velocity difference).

For SDG + momentum, we take the velocity and add the gradient. In NAG is the gradient is more accurate by predicting where the step will be, and using this onformation to compute the gradient.

6.

Learning rate decay solutions :

1. Step decay :
   - every k iterations : $\eta < -\eta/2$
2. Exponential decay :
   - $\eta = \eta.e^{-k/t}$
3. Fractionnal decay :
   - $\eta = \eta_0/(1 + k.t)$

7.

Newton Method:

- Goal : Compute the learning rate instead of specifying it.
- The learning rate for differents features may vary.
- Algorithm :
    - find x such that f(x)=0
    - start with guess Xo
    - find update ΔX so f(Xo + δX)=0
    - continue while f(Xi+ΔX)>ε

For our problem, we replace f(x) by J(θ) to minimize the loss.
The Hessian matrix here is equivalent to ΔX in the global Newton's method: it is the 2nd derivative of the loss.
We use the Hessian matrix as learning rate : the elements of the Hessian matrix will act as learning rates for their respective parameters. Higher values will cause low learning rates, because H^{-1} is used for parameters update.

8.

The condition number evaluates the complexity of the method:

$$Cn = \frac{SV_{max}}{SV_{min}}$$

It uses the singular values decomposition of the elements of H.

The bigger the condition number is the more difficult the problem is.

Poor conditioning will cause a big difference in the elements of the Hessian matrix and therefore a bigger condition number.

9.

$$B^{(i)} = diag(\sum_{j=1}^{i} \nabla J(\theta^{(i)}) \nabla J(\theta^{(j)})^T)^{\frac{1}{2}}$$

B is a diagonal matrix which is easy to inverse. It only computes an approximation of the speed of how the loss changes in respect to each parameter. The elements outside the diagonal are not computed.

10.

In AdaGrad, because we normalize by elementwise sum of square gradients, the step size well become smaller as iterations progress. To control this, we use in RMSProp a decay factor (e.g. 0.9) when adding new gradients to the gradient sum.

11.

Addam combine RMSProp with momentum.

Because the moments are initialized to zero, when dividing by two second moment we will get a large step.

=> use a bias corrected term dividing the momentum by a number depending on iteration number (so that initial moments are longer).

12.

Instead of a fixed step size, find the “best” step size by searching along the line in the same direction.

- Given direction :

$$\mu = \nabla f(X)$$

- The Best step size is :

$$\eta^* = argmin\eta f(X + \eta\mu)$$

- Gradient descent :

$$\theta^{(i+1)} < -\theta^{(i)} - \eta^{*(i)}\nabla f(\theta^{(i)})$$

2 methods :

- Bracketing : Given a bracket [a,b,c], X = (b+c)/2, and update the brackets to be smaller and smaller until it is small enough
- Successive line search, start with θ_0 and direction set $\{\mu^{(i)}\}$, and iterate.

13.

The goal of quasi Newton methods is to skip the computation of the Hessian matrix by replacing it by an approximation that is updated during each iteration.

Algorithm:

- Compute the quasi-Newton direction:

$$\Delta\theta = -(H^{(i-1)})^{-1}\nabla J(\theta^{(i-1)})$$

- Determine step size: η*

- Update parameters:

$$\theta^{(i)} = \theta^{(i-1)} + \mu * \Delta\theta$$

- Compute the updated Hessian approximate H(k)

The cost is reduced from O(n^3) to O(n^2) for the computation of the Hessian matrix, compared to the usual Newton method.
Adam uses an imitation of the curvature and can fail if a particular problem involves bias in that imitation. BFGS is more stable but might take more time, where Adam can be quicker but can fail.

Regularization

1.

The weight decay is related to adding a regularization term to the loss function because it will nullify the weights that are not reinforced during the learning. The nullified weights won’t affect the loss function.

2.

To stop early we need to train the model on the training data set to find some information (optimal number of iterations or loss) to stop the training early.

We have seen 2 strategies to reuse the validation set:

- train the model on training set, then find with the validation set the optimal number of iterations i*. Train the model with new initialized parameters, with both the training and validation set but only with i* iterations.
- train the model on training set, then find with the validation set the optimal loss ε. Then train the model with the previous parameters, with both the training and validation sets, until the loss is better than ε.

3.

Data augmentation is generating new data from the original dataset. (by adding noise, interpolation, rotation,…)
This new data is more diversified, it’ll bring new features to the model, and help preventing overfitting.

4.

Dropout is performed by implementing a probability p, so each node in the model has a probability $(1-p)$ to be dropped out. On future batches, the same process is applied to the original model, the deletion of nodes does not add up though batches. Dropout can also be applied to inputs of nodes: then the inputs are randomly set to 0.

Pros :
- Reduces overfitting
- Improves nodes independency

- Distributes features on multiple nodes
Cons :
- Longer training time, needs to test for different Dropout values, too high will mean information loss and will decrease the performance of the model, when too low will not change anything.

A standard dropout rate is between 20 and 50%

5.

To approximate the outputs of the networks we can sum all the expected values multiplied by their respective probability:

$$\hat{y} = \int P(D)f(X,D)dD$$

6.

For each batch, on some of the fully connected layers, the mean μ and standard deviation σ of each input is computed, and each of the value is modified:

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Then the result is modified by scale j and shift j which are trainable variables:

$$\tilde{x} = yj * \hat{x} + \beta j$$

Each batch has a random mean and standard deviation, using them will therefore introduce randomness with each batch.

7.

Scale and shift are used to control the effects of the batch normalization: its effects are nullified for $yj = \sigma$ and $\beta j = \mu$. They can be learned by backpropagation like other trainable variables. A good initialization is $yj = 1$ and $\beta j = 0$.

8.

Using multiple models to create a new one (using the mean or majority vote) will prevent overfitting because models are independent from each other : each one might be overfitting but using many of them reduces the odds that the final model overfits.
To have multiple models we can change data, change the parameters (initial or hyperparameters), or save snapshots of the model during training and consider the snapshots as independent.