Machine Learning Dictionary

Max Sharnoff, Fall 2018

1 Loss Functions

Loss functions are defined as $C(\hat{y}, y)$ where $\hat{y}$ is a list (or vector) of the outputs of the network (sometimes referred to as predictions or probabilities), and y is the target or "ground truth" outputs.

1.1 Absolute Loss / L1 Loss

Absolute loss is defined as:

$$C(\hat{y}, y) = \sum\nolimits_i |\hat{y}_i - y_i|,$$

where $y \in \{0, 1\}^n$ and $\hat{y} \in [0, 1]^n$.

Absolute Loss is strong in its robustness to outliers, but lacks the fine tuning that comes with other, smoother, curves closer to the desired outputs. The derivative of Absolute Loss is:

$$\frac{\partial C}{\partial \hat{y}_i} = \frac{|\hat{y}_i - y_i|}{\hat{y}_i - y_i}$$

1.2 Mean Squared Error (MSE) / L2 Loss

MSE is defined as:

$$C(\hat{y}, y) = \frac{1}{2}\sum\nolimits_i (\hat{y}_i - y_i)^2,$$

where $y \in \{0, 1\}^n$ and $\hat{y} \in [0, 1]^n$.

MSE particularly struggles outlier predictions, which it will allow to affect the model more than is necessary. The derivative of MSE is:

$$\frac{\partial C}{\partial \hat{y}_i} = \hat{y}_i - y_i$$

1.3 Huber Loss

Huber Loss is defined as:

$$C_i = \begin{cases} \frac{1}{2}(\hat{y}_i - y_i)^2 & |\hat{y}_i - y_i| \le \delta \\ \delta|\hat{y}_i - y_i| - \frac{1}{2}\delta^2 & otherwise \end{cases},$$

where C_i is the loss for an individual class (summed to obtain the total loss), $y \in \{0, 1\}^n$, and $\hat{y} \in [0, 1]^n$.

Huber loss works to provide a middle-ground between L1 and L2 loss, as it is defined to be a transition from L2 to L1 at δ. Depending on the value of δ, it provides the fine gradient of L2 Loss while keeping the robustness to wildly

innacurate preditions of L1 Loss. The derivative of Huber Loss is similar to L1 and L2 Loss. It is:

$$\frac{\partial C}{\partial \hat{y}_i} = \begin{cases} \hat{y}_i - y_i & |\hat{y}_i - y_i| \leq \delta \\ \delta \frac{|\hat{y}_i - y_i|}{\hat{y}_i - y_i} & otherwise \end{cases}$$

1.4 Cross Entropy Loss / Negative Log Likelihood

Cross entropy loss is defined as:

$$C(\hat{y}, y) = -\sum_i y_i \, ln(\hat{y}_i),$$

where $y \in 0, 1^n$ and $\hat{y} \in [0, 1]^2$.
Cross Entropy Loss only measures the difference in the correct class, not any other classes. The derivative of Cross Entropy Loss is:

$$\frac{\partial C}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

1.5 Kullback-Leibler Divergence / Relative Entropy

K-L Divergence is defined as:

$$C(\hat{y}, y) = -\sum_i (y_i \, ln(\hat{y}_i) - y_i \, ln(y_i)),$$

where $y \in 0, 1^n$ and $\hat{y} \in [0, 1]^2$.
KL Divergence is very similar to Cross Entropy Loss – the added constant of $y_i \, ln(y_i)$ keeps the derivative the same. Its derivative is:

$$\frac{\partial C}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

1.6 Not yet documented:

- Hinge Loss — the main use case of Hinge Loss is for SVMs, so it has not been documented yet.

2 Random Number Generators

These are only mentioned because it is useful to have a common language for describing initializers.

2.1 Uniform

This is a uniformly random distribution. Future notation will refer to it as $G_u(r)$, where r is the range of the distribution.

2.2 Normal

This is a normal distribution, centered at 0. Initializer equations will refer to it as $G_n(\mu, \sigma)$, where μ is the center (average) of the distribution and σ is the standard deviation.

2.3 Truncated Normal

This is the Normal distribution, where values outside two standard deviations are re-drawn. Initializer equations will refer to it as $G_t(\mu, \sigma)$, with the same parameters as G_n.

3 Initializers

For all initializers, variable definitions are as follows:

w_i is the weight to be initialized,
n_{in} is the number of input values to the layer, and
n_{out} is the number of output values to the layer.

Additionally, random number generators (which have been previously, uniquely defined) will be referenced specifically. Those are:

$G_u(r)$ as the Uniform distribution,
$G_n(\mu, \sigma)$ as the Normal distribution, and
$G_t(\mu, \sigma)$ as the Truncated Normal distribution.

Some initializers also will have two separate versions: one for uniform distributions, and one for truncated normal distributions.

3.1 Constant

$$w_i = x,$$

where w_i is the weight to be initialized, and x is a fixed number.

3.2 Random

$$w_i = G(\dots),$$

where w_i is the weight to be initialized, and $G(\dots)$ is the random number generator.

3.3 Variance Scaling

$$w_i = G_t(\mu = 0, \sigma = \sqrt{\frac{f}{n}}),$$

where w_i is the weight to be initialized, f is a user-provided scaling factor, and n is determined by the control flow below:

```
if mode="in" {
    n=in
} else if mode="out" {
    n=out
} else if mode="average" {
    n=(in + out)/2
}
```

Here, 'in' is the number of input values to the neuron (or other layer component) to which the w_i belongs. 'out' is the number of values the layer provides as input to other layers. Variance scaling allows multiple modes, "in", "out", and "average", each of which is self-explanatory.

3.4 LeCun

$$w_i = G_t(\mu = 0, \sigma = \sqrt{\frac{1}{n_{in}}}), \text{ or}$$

$$w_i = G_u(\pm\sqrt{\frac{3}{n_{in}}})$$

3.5 He

$$w_i = G_t(\mu = 0, \sigma = \sqrt{\frac{2}{n_{in}}}), \text{ or}$$

$$w_i = G_u(\pm\sqrt{\frac{6}{n_{in}}})$$

The He initializer is $\sqrt{2}$ times the LeCun initializer.

3.6 Xavier / Glorot

$$w_i = G_t(\mu = 0, \sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}), \text{ or}$$

$$w_i = G_u(\pm\sqrt{\frac{6}{n_{in} + n_{out}}})$$

4 Activation Functions

Activation functions are additions to layers that (usually) do not have learned parameters themselves. The primary purpose of Activation Functions is to limit the range of outputs from a layer. Activation functions preserve the number of outputs from a layer, and can usually be applied to each index individually.

Variable definitions:

P_i is the ith value of the layer, before applying the activation function
a_i is the ith value of the activation function

4.1 Linear

Linear is the most basic of activation functions. It's defined as:

$$a_i = P_i$$

and its derivative is 1:

$$\frac{da_i}{dP_i} = 1$$

4.2 Rectified Linear Unit (ReLU)

$$a_i = \max(0, P_i) = \begin{cases} P_i & P_i > 0 \\ 0 & P_i \leq 0 \end{cases}$$

ReLU was designed with biological models in mind, and proves to be quite effective. Just like biological neurons, it cannot output negative values – instead it does not 'fire'. Its derivative is:

$$a_i = \begin{cases} 1 & P_i > 0 \\ 0 & P_i \leq 0 \end{cases}$$

Because the derivative is zero when $P_i \leq 0$, ReLUs speed up model calculation and backpropagation. They do come with some faults, however: If a neuron is initialized as negative or has learned to be negative, this neuron can become 'dead', where it no longer can have any impact on the model – and has effectively died.

4.3 Leaky ReLU (Leaky ReLU, LReLU)

$$a_i = \begin{cases} P_i & P_i > 0 \\ \alpha P_i & P_i \leq 0 \end{cases},$$

where α is a hyperparameter and $\alpha > 0$. α is usually very close to 0 to keep with the intention of vanilla ReLUs (try 0.01). The derivative is:

$$\frac{da_i}{dP_i} = \begin{cases} 1 & P_i > 0 \\ \alpha & P_i \leq 0 \end{cases}$$

Because there are no intervals on which the derivative is zero, this fixes the 'dead ReLU' problem, wherein some values become unusable because they are always zero.

4.4 Parametric ReLU (PReLU)

$$a_i = \begin{cases} P_i & P_i > 0 \\ \alpha P_i & P_i \leq 0 \end{cases},$$

where α is a learned parameter, and $\alpha_0 \neq 0$. This is essentially the same as a Leaky ReLU, but with run-time customization. Because of this, its derivative

is the same:

$$\frac{da_i}{dP_i} = \begin{cases} 1 & P_i > 0 \\ \alpha & P_i \leq 0 \end{cases}$$

Additionally, the derivative of the function with respect to α is:

$$\frac{da_i}{d\alpha} = \begin{cases} 0 & P_i > 0 \\ P_i & P_i \leq 0 \end{cases}$$

4.5 Exponential Linear Unit (ELU)

$$a_i = \begin{cases} P_i & P_i > 0 \\ e^{P_i} - 1 & P_i \leq 0 \end{cases}$$

ELU is continuous approximation of ReLU that also allows negative outputs – based on research that shows that models learn faster when the average activation is closer to zero.

Some versions will add α before $e^{P_i} - 1$, but 1 is the only value for which the function is continuous at the origin. This addition of α could – in theory – be used to allow α to be a learned parameter, in a similar fashion to PReLUs. The derivative of ELU is:

$$\frac{da_i}{dP_i} = \begin{cases} 1 & P_i > 0 \\ e^{P_i} & P_i \leq 0 \end{cases}$$

For more information on ELU, see the original paper.

4.6 Softplus

$$a_i = ln(1 + e^{P_i})$$

Softplus is a smooth approximation of ReLU, in a similar fashion to ELU, but it appproaches 0 towards negative infinity. Towards positive infinity, it approaches the line $y = x$, and it does so very quickly. $(f(4) = 4.02)$ The derivative of Softplus is the sigmoid function:

$$\frac{da_i}{dP_i} = \frac{1}{1 + e^{P_i}}$$

4.7 Tanh

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

Tanh is makes a smooth curve that approaches -1 towards negative infinity and 1 towards positive infinity. It approaches its limit much more quickly than either Logistic or Softsign. The derivative of tanh is:

$$\frac{da_i}{dP_i} = 1 - \tanh^2(P_i)$$

4.8 Logistic / Sigmiod

$$a_i = \frac{1}{1 + e^{-P_i}} = 0.5 + 0.5 \tanh(0.5 P_i)$$

The logistic function is similar to tanh in the shape of its curve, but its range is only between 0 and 1 and it takes longer to approach its limits. (It is actually a scaled version of tanh) It also has a particular problem because of the magnitude of its derivative: Because the maximum of its derivative is only 0.25, the model's gradients diminish as they pass through more layers – this makes it more difficult to train Deep Neural Networks.

The derivative of the logistic function is:

$$\frac{da_i}{dP_i} = a_i(1 - a_i)$$

4.9 Hard Tanh / Hard Sigmoid

These are both linear, piecewise approximations of their smooth counterparts, which can be modeled as versions of the same general equation:

$$a_i = \max(a, \min(\frac{b-a}{2\alpha} P_i + \frac{b+a}{2}, b)) = \begin{cases} a & P_i < -\alpha \\ \frac{b-a}{2\alpha} P_i + \frac{b+a}{2} & -\alpha \leq P_i \leq \alpha \\ b & P_i > \alpha \end{cases},$$

where a is the minimum value, b is the maximum value, and α is the distance from zero the values start to occur at. These approximations do still work if $a \geq b$. For Hard Tanh, these values would be:

$$a = -1, b = 1, \alpha = 1$$

For Hard Sigmoid, these values are:

$$a = 0, b = 1, \alpha = 2$$

The derivative of the general case is:

$$a_i = \begin{cases} 0 & P_i < -\alpha \\ \frac{b-a}{2\alpha} & -\alpha \leq P_i \leq \alpha \\ 1 & P_i > \alpha \end{cases}$$

Note: For some reason, the majority of sources seem to prefer using the name 'Logistic Function' to describe the smooth curve, but switch to 'Hard Sigmoid' to describe the linear approximation.

4.10 Softsign

$$a_i = \frac{P_i}{|P_i| + 1}$$

Softsign appears similar in shape to Tanh or Logistic (once scaled). At the origin, its slope is equal to Tanh (which is steeper than Logistic), but it levels

off very quickly and takes much longer to approach 1 than the others. Its derivative is:

$$\frac{da_i}{dP_i} = \frac{1}{(|P_i| + 1)^2}$$

4.11 Softmax

$$a_i = \sigma(P)_i = \frac{e^{P_i}}{\sum_k e^{P_k}}$$

While softmax does use all of the values in the layer to calculate individual values, it is still considered an activation function. Softmax 'squishes' all of the values of a layer to 1. Softmax is often (and only) used as the final layer of a neural network. The derivative of softmax is:

$$\frac{da_i}{dP_i} = a_i(1 - a_i)$$

Note: This is the same derivative as from the logistic function.

5 Regularization

The bias / variance tradeoff is a central problem in training supervised learning models. Bias is the tendency of a model to generalize (and make erroneous predictions because of it), and variance is the tendency of a model to overfit the training data. Regularization aims to ensure the model falls in the middle-ground between these two.

Some definitions:

$\boldsymbol{\theta}$ is the vector of all weights in the network. Even though the weights are also defined as w, regularization methods frequently refer to them as θ. θ has been left un-bolded for syntactic elegance.

$C(\theta)$ is the cost function in use

5.1 L1 (Lasso)

$$E_{\text{L1}} = C(\theta) + \underbrace{\lambda \sum_i |\theta_i|}_{\text{L1}}$$

L1 Regularization is an extra pentalty added onto each weight, where $\lambda > 0$ is the strength of this penalty (usually very close to zero). This prevents their values from growing unecessarily large (positively or negatively). The penalty itself is only the underbraced section 'L1', and its derivative is:

$$\frac{d\text{L1}}{d\theta_i} = \lambda \frac{\theta_i}{|\theta_i|}$$

5.2 L2 (Ridge)

$$E_{\text{L2}} = C(\theta) + \underbrace{\lambda \sum_i \theta_i^2}_{\text{L2}},$$

where $\lambda > 0$ is again the strength of the penalty. L2 Regularization is similar to L1, but the penalty is squared. The derivative of L2 Regularization is:

$$\frac{d\text{L2}}{d\theta_i} = 2\lambda\theta_i$$

5.3 Elastic-Net (L1/L2)

Elastic-Net serves as a combination of L1 and L2 Regularization, such that (ideally) the model gets the benefits of both.

$$E_{L1/L2} = C(\theta) + \lambda((1-\alpha)\sum_i \theta_i^2 + \alpha \sum_i |\theta_i|),$$

where $\lambda > 0$ is the total strength of the penalty – affecting both the L1 and L2 components, and α is the ratio between them. For example: If $\alpha = 0$, Elastic-Net will be identical to L2 Regularization; if $\alpha = 1$, it will instead be L1 Regularization. Its derivative is:

$$\frac{d\text{L1/L2}}{d\theta_i} = \lambda((1-\alpha)2\theta_i + \alpha\frac{\theta_i}{|\theta_i|})$$

5.4 Dropout

Dropout is a technique usually applied to fully-connected layers of neurons. During training iteration, it gives each neuron a probability of P of 'dropping out', where it is temporarily removed from the network. At test time, all weights are multiplied by P to ensure outputs the domain of inputs to neurons is roughly the same. Dropout is effective because it simulates the combination of simulating exponential numbers of smaller networks.

For more information, see the original paper.

5.5 Batch Normalization (batchnorm)

Batch Normalization is a the process of normalizing (from statistics) the activations of a layer over the course of a mini-batch. This is based on research that shows that normalizing the input data (with $\sigma = 1$) speeds up training. The fundamental formula of Batch Normalization appears complex, but it is built of smaller parts. The normalization process occurs in relation to the other values for the same dimension for the mini-batch, so the dimension specification is omitted for clarity.

We start with a mini-batch with size m:

$$\mathcal{B} = \{x_1, x_2, \ldots x_m\}$$

To apply batchnorm, we first compute the mean (μ) and variance (σ^2) of the batch:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=0}^{m} x_i$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=0}^{m}(x_i - \mu_{\mathcal{B}})^2$$

Then, normalize each x to produce $\hat{x}$:

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

ϵ is "a constant added for numerical stability." Finally, shift and scale by learnable parameters γ and β:

$$y_i = \gamma\hat{x}_i + \beta$$

The authors note that it is possible to un-learn this normalization if γ assumes the value of $\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}$ and β assumes the value of $\mu_{\mathcal{B}}$. These parameters are initialized at $\gamma = 1$ and $\beta = 0$ to start with the full normalization. After training, population statistics are used, with the normalization becoming more accurate as the number of samples increases.

Batchnorm has several problems in application, most notably the impracticality of computing statistics on all of a layer's inputs throughout an entire mini-batch before providing a single output. This can be solved by using running averages instead, but that solution runs into problems when the batch size is small – it usually takes many iterations before the running averages are stable and accurate.

For more information, see the original paper.

5.5.1 Batch Renomalization (Batch Renorm)

Batch Renormalization proposes to fix the problems that Batch Normalization encounters with small batch sizes. Here's the paper.

5.6 Data Agumentation

Overfitting data can also be caused by not having enough training data. One method for solving this is data agumentation – changing existing data to generate more. This can come in many forms. There are many ways to do this for images, but they may still apply to other problem domains:

- Adding noise
  - Gaussian noise can effectively 'shade' images.
  - "Salt and Pepper" noise is the addition of random white or black pixels throughout the image.
- Image transformations (Note: Some of these do not apply to some image domains)

  - Rotation — either by 90 degrees, where the image is effectively unchanged, or continuously for finer grades.
  - Flipping (horizontally, vertically)
  - Zooming in or out (scaling)
  - Cropping — actually crop, then resize to fill
  - Translations (up/down, left/right)

- Wrapping methods for when the image doesn't fill all of the inputs:
  - Using a constant border
  - Wrapping (tiling) the image
  - Extending the edges
  - Reflecting the image

6 Optimizers

Some definitions:
$\eta > 0$ is the learning rate, close to 0
θ_t is the weight we're adjusting, at the current iteration
$f(\theta)$ is the objective function to minimize, and
$\nabla f(\theta)$ is the gradient of the objective function with respect to the weight we're adjusting

6.1 Gradient Descent

For each time-step (iteration),

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t)$$

6.2 Momentum

Momentum adds a velocity v, which is modified by the current gradient. The operation at each iteration t happens in the order it is written here:

$$v_{t+1} = \mu v_t - \eta \nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

where $\mu \in [0, 1]$ is the momentum coefficient, a hyperparameter. The original paper (Polyak, 1964) can be found here.

6.3 Nesterov Momentum / Nesterov Accelerated Gradient (NAG)

Nesterov Momentum adds an extra step between velocity calculation and weight updating, using a set of intermediary values, y.

$$\theta_t = y_t - \eta \nabla f(y_t)$$

$$y_{t+1} = \theta_t + \mu(\theta t - \theta_{t-1})$$

where $\mu \in [0, 1]$ is the momentum coefficient, a hyperparameter. The original paper (Nesterov, 1983) can be found here. Other variations on Nesterov Momentum have since been made.

6.3.1 Sutskever Nesterov Momentum

Sutskever et al. redefine Nesterov Momentum in terms of θ and velocity to allow it to more easily fit the previous model:

$$v_{t+1} = \mu v_t - \eta \nabla f(\theta_t + \mu v_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

The original paper (Sutskever et al., 2013) can be found here, with an excellent explanation here.

6.3.2 Bengio Nesterov Momentum

Bengio et al. provide another transformation of the Nesterov Momentum equations, but it will not be covered here. Their paper can be found here, and an explanation here.

6.4 Adagrad

For the sake of syntactic brevity, we define:

$$g_t = \nabla f(\theta_t)$$

Then, let G_t equal the sum of the squares of the gradients at each time step:

$$G_t = \sum_{i=0}^{t} \nabla f(\theta_i)$$

Adagrad rewrites the weight update step from $\theta_{t+1} = \theta_t - \eta g_t$ to:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t,$$

where ϵ is an arbitrarily small, positive value (typically around 10^{-8}). An excellent explanation can be found here.

6.5 RMSprop

RMSprop is actually an unpublished optimization algorithm, presented by Geoff Hinton. Its derivation starts with the update step from Adagrad:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

We then define $E[x^2]_t$ as the decaying average of x:

$$E[x^2]_t = \rho E[x^2]_{t-1} + (1 - \rho) x_t^2,$$

where $\rho \in [0, 1]$ (somtimes γ) is the decay factor, usually set at 0.9. Substituting in $E[g^2]_t$ instead of G_t, we get:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

The denominator is equal to $\text{RMS}[g]_t$ (the Root Mean Square function), so we can write:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\text{RMS}[g]_t} g_t,$$

giving us the definition of RMSprop. Hinton recommends $\rho = 0.9$ and $\eta = 0.001$.

6.6 Adadelta

Adadelta is an extension of Adagrad that seeks to reduce its "aggressive, monotonically decreasing learning rate."[1] Starting from RMSprop, Adadelta substitutes η for $\text{RMS}[\Delta\theta]_{t-1}$:

$$\theta_{t+1} = \theta_t - \frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t$$

where $\Delta\theta_t$ is defined as:

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Simple rephrasing to just show $\Delta\theta$ yeilds the formal definition of Adadelta:

$$\Delta\theta_t = -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

It will not be covered here, but the original paper explains (albeit briefly) why $\text{RMS}[\Delta\theta]_{t-1}$ is a suitable substitution for η.

6.7 Adam (Adaptive Moment Estimation)

Adam is a slightly more complex algorithm, in terms of its number of hyperparameters. For each time-step (iteration) t, weights are updated such that:

$$m_t = \alpha m_{t-1} + (1 - \alpha) g_t$$
$$v_t = \beta v_{t-1} + (1 - \beta) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \alpha^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta^t}$$
$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where $g_t = \nabla f(\theta_t)$, $m_0 = v_0 = 0$ and $\alpha, \beta \in [0, 1)$ Note: $g_t^2 = (g_t)^2$.
Suggested default values are: $\eta = 0.001$, $\alpha = 0.9$, $\beta = 0.999$, and $\epsilon = 10^{-8}$

[1] http://ruder.io/optimizing-gradient-descent/index.html#adadelta

Note: In their paper, the authors instead refer to variables as $\alpha \to \beta_1$, $\beta \to \beta_2$, and $\eta \to \alpha$

6.7.1 AdaMax

In the same paper, the authors discuss AdaMax, "a variant of Adam based on the infinity norm." The derivation is long, but the end result is a slightly different algorithm:

$$m_t = \alpha\, m_{t-1} + (1-\alpha) g_t$$
$$u_t = \max(\beta\, u_{t-1}, |g_t|)$$
$$\theta_{t+1} = \theta - (\frac{\eta}{1-\alpha^t})\frac{m_t}{u_t}$$

where $\frac{\eta}{1-\alpha^t}$ is the bias-corrected learning rate. The authors suggest different learning rate for this algorithm: $\eta = 0.002$, instead of $\eta = 0.001$.

6.8 Nadam (Nesterov-accelerated Adaptive Moment Estimation)

The fundamental idea behind Nadam is applying Nesterov Momentum to Adam. The authors remark that Nesterov Momentum is an improvement upon basic momentum, and where Adam is built upon momentum, Nadam is built upon Nesterov Momentum.

The derivation is long, so the just the algorithm is written. For each time-step t:

$$g_t = \nabla f(\theta_t)$$
$$\hat{g} = \frac{g_t}{1 - \Pi_{i=0}^{t} \mu_i}$$
$$m_t = \mu\, m_{t-1} + (1-\mu) g_t$$
$$\hat{m}_t = \frac{m_t}{1 - \Pi_{i=0}^{t+1} \mu_i}$$
$$n_t = v\, n_{t-1} + (1-v) g_t^2$$
$$\hat{n}_t = \frac{n_t}{1 - v^t}$$
$$\overline{m}_t = (1-\mu_t)\hat{g}_t + \mu_{t+1}\hat{m}_t$$
$$\theta_{t+1} = \theta_t - \eta \frac{\overline{m}_t}{\sqrt{\hat{n}_t} + \epsilon}$$

where $v \in (0, 1)$ is a large hyperparameter typically set around 0.9, and μ has been allowed to change over time. For more information, read the paper.

6.9 Not yet documented

- AMSgrad — It has yet to be seen whether or not AMSgrad is a reasonable competitor to the optimizer it was trying to beat: Adam.

7 To add

- Layers
- Appendix (containing extended derivations)