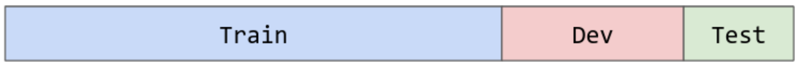
Improving Deep Neural networks: Hyperparameter tuning, Regularization and Optimization

Train, Dev and Test Set

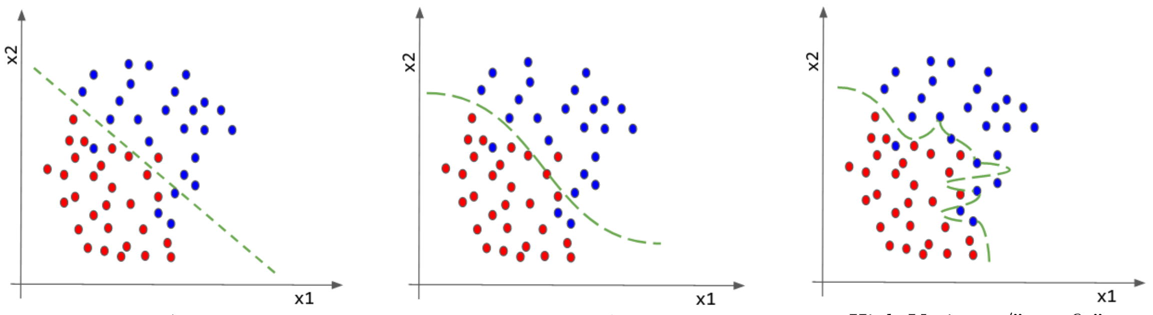
Making good choices on data split can impact the quality of model learning. Deep learning paradigms have seen a lot of success, yet the knowledge learnt in one domain cannot be easily transferred to another. Hence, we need to go through few cycles of development to get it right. The big question is "How could we optimize the process"?.



Data Splitting

We model our data on the `train` set and do iterations of evaluation on the `dev` set. Once the modeling is done, we can check the overall performance on test set. In traditional ML, the split was usually 70:30 (`train-dev`) or 60:20:20 (`train-dev-test`). In big-data regime, we can allocate much smaller portions to `dev/test` e.g. (98:2 or 98:1:1). It should be just big enough to evaluate the model reliably.
Another aspect to consider is mismatched distributions i.e. `train` and `dev` set are composed of different data pieces/sources. The rule of thumb to follow is that `dev` and `test` should come from the same distribution to evaluate coherently. It is also okay to not have a `test` set. We do not always have the luxury of having a separate `test` data. In its absence, the `dev` set is implied to cover both the roles.

Bias and Variance



High Bias/"underfit" Just Right High Variance/"overfit"

In high dimensional data, we cannot plot and decide the quality of fit. We rely on information gleaned from the error trends for insights.

- The bias is an error from erroneous assumptions in learning the data. High bias causes algorithm to miss relevant relations between features and target output → "underfitting"
- The variance error comes from sensitivity to small fluctuations in training set. High variance can cause algorithms to model random noise and outliers in `train`, rather than the intended features → "overfitting"

Under-fit models have high bias. It consistently gets answers wrong. Variance reflects how much model is dependent on training data. In the example shown, the model made a strong assumption that the data was linear in the first case. This did not cover most samples,hence under-fit. In the last case, the model did it best to account for every single sample as best possible, leading to high variance/over-fit. It essentially memorized the data and noise as well. Over-fit doesn't generalize well to test data. Humans are assumed to achieve $\approx 0\%$ error or the Bayes Error $\rightarrow$ The lowest error a classification model can achieve. With a higher Bayes error, the analysis could be different. For bias, always check with `train` set error and for variance, check the `dev` set error.

Regularization

If we suspect overfitting, regularization is the first step.

- L1 (or Lasso):
  $\underset{w,b}{min}\, J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{L}(\hat{y}^i, y^i) + \frac{\lambda}{2m}\sum_{i=1}^{n_x}|w_i|$
- L2 (or Ridge):
  $\underset{w,b}{min}\, J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{L}(\hat{y}^i, y^i) + \frac{\lambda}{2m}\sum_{i=1}^{n_x}w_i^2$

λ is the regularization parameter $\rightarrow$ hyperparameter. Using L2 regularization, we can tune model by penalizing large model weights. L1 regularization also penalizes the weights but has the effect of making the model "sparse". It is good for model compression and feature selection.

In neural network, we define Frobenius Norm / Matrix norm / Euclidean norms as follows:

$$\underset{w,b}{min}\, J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{L}(\hat{y}^i, y^i) + \frac{\lambda}{2m}\sum_{i=1}^{L}\left\|w^{[l]}\right\|^2$$
$$\left\|w^{[l]}\right\|_{\mathbb{F}}^2 = \sum_{i=i}^{n^{[l]}}\sum_{j=i}^{n^{[l-1]}} w_{i,j}^{[l]2}$$

During gradient descent,

$$\begin{aligned}\delta w^{[l]} &= \left(\frac{\partial J}{\partial w^{[l]}} + \frac{\lambda}{m}w^{[l]}\right)\\ w^{[l]} &= w^{[l]} - \alpha\delta w^{[l]}\\ &= \left[1 - \frac{\alpha\lambda}{m}\right]w^{[l]} - \alpha\frac{\partial J}{\partial w^{[l]}}\end{aligned}$$

Because $\left[1 - \frac{\alpha\lambda}{m}\right] < 1$, the L2 norm is also called the weight decay regularization. It has an effect of shrinking the values in the matrix.

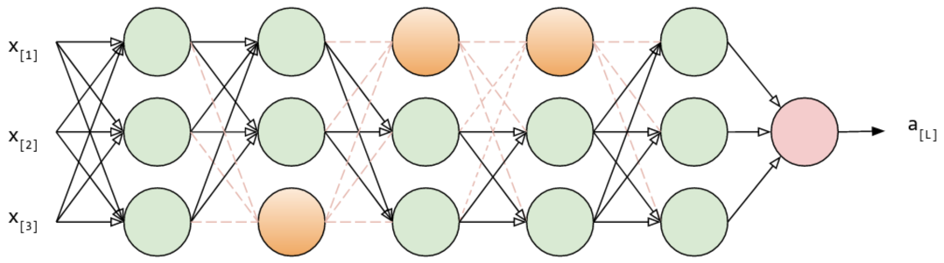
Regularization mitigating Over-fitting

Consider the cost function with matrix norm:

$$\underset{w,b}{min}\, J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{L}(\hat{y}^i, y^i) + \frac{\lambda}{2m}\sum_{i=1}^{L}\left\|w^{[l]}\right\|_F^2$$

The reason for over-fitting is the network computing a complex polynomial via several different pathways. Simpler networks generalizing on the data is preferable. If λ is large, the $w_{i,j}^{[l]}$ will be pushed down $\forall\, i,j,l$. Lot of neuron will insignificantly contribute leading to an effective network which is much smaller. Regularization has the effect of constraining weights such that input activation is constrained in linear region and not allowed to blow up (where corrective adjustments are also slow, if we consider sigmoid activations). Linear models cannot form very complex decision boundaries.

Dropout



Schematic of Dropout in a regular forward/back pass. Some nodes have been turned off at random for each layer.

In Dropout, we go through each layer and set some probability of its nodes in contributing to computation. With different samples, different set of nodes get knocked off or updated. Dropout randomly knocks out nodes so that only a smaller network is responsible for prediction. We do not rely on one feature or neuron. This has the effect of shrinking squared norm → implicit regularization. One downside of this method is that the cost function is no longer well defined, due to the random switching off.

Other Regularization techniques

Augmentation

Adding more training samples by geometrical transformations may improve the quality of training. It may or may not add more qualitative information e.g. flip, random zoom, center crop. We have to be careful about augmentation rules that may affect the task.

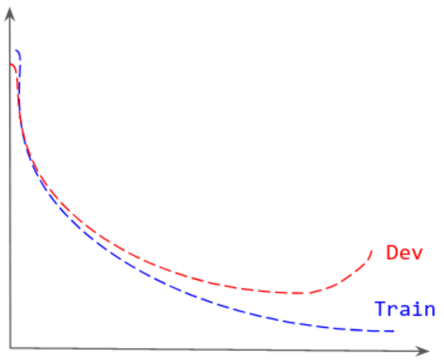
Early stopping

`dev` set error usually goes down and increases when network starts memorizing. Stopping at the lowest `dev` error level could prevent that. The only drawback of Early stopping is that it couples the cost function's optimization and over-fitting reduction → breaking orthogonalization. Best practices mandate using L2-norm since only λ needs ascertaining in a few tries.

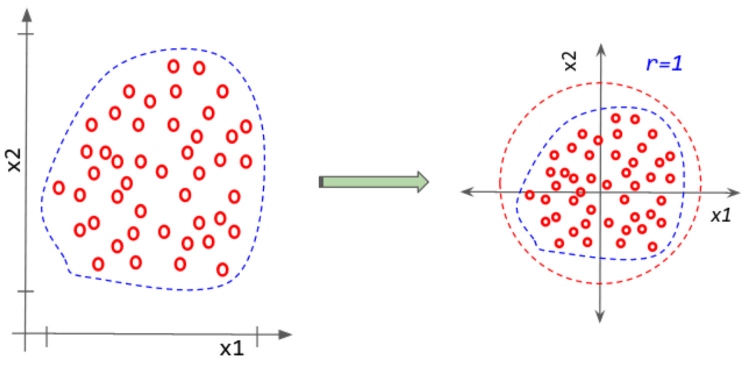
Normalization

Normalizing inputs has the effect of zero centering with unit variance. Both features are now centered about ($\mu = 0, \sigma^2 = 1$). Without normalization, the loss surface will be distorted or elongated because of different ranges for individual features. Gradient descent finds it hard to converge in such conditions, undergoing several phases or jitters or oscillations. With normalization, the loss surface is more uniform and learning is easier.

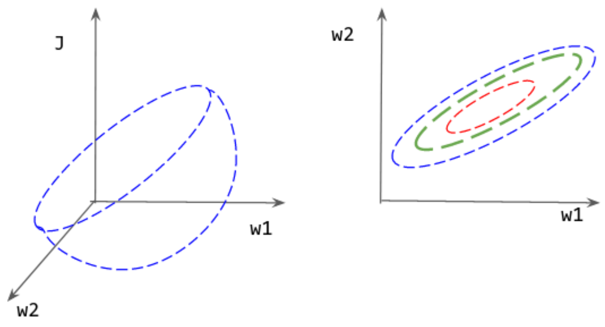
Srivastava (2014), "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR



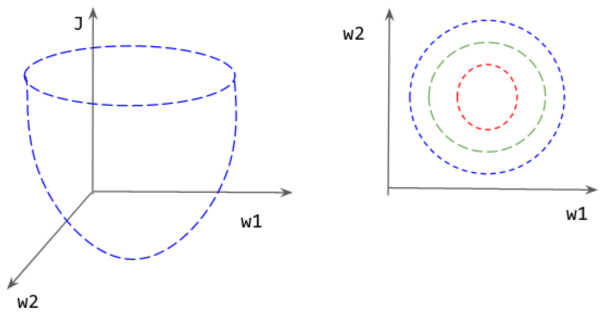
Early stopping schematic where the dev error again starts rising. The goal of early stopping is to stop training at the point of inflexion, so that model does not overfit to the train data.



Normalization process



Before normalization

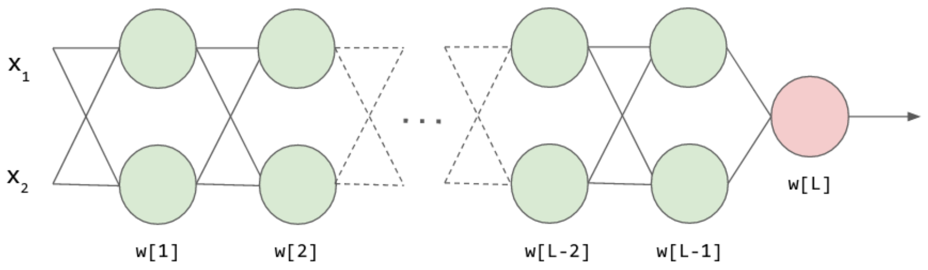


After Normalization

Vanishing and Exploding Gradients

If gradients become too large or small, that could be a problem in the learning process.
Say, we have $g(z) = z, b_i^{[l]} = 0 \, \forall \, i, l$ Then the output $\hat{y}$ is

$$\hat{y} = w^{[L]} \, w^{[L-1]} \, \ldots \, w^{[2]} \, w^{[1]} x$$



Schematic of a very long neural network. It has 2 nodes per layer and L layers for the purpose of illustrating vanishing or exploding gradients.

- Consider for $\delta > 0 \, \forall \, l$
$$w^{[l]} = \begin{bmatrix} 1+\delta & 0 \\ 0 & 1+\delta \end{bmatrix}$$
i.e., diagonal values slightly different from the identity matrix. The output $\hat{y}$, in such a case will be a 'exploding' value
$$\hat{y} = w^{[L]} \cdot \begin{bmatrix} 1+\delta & 0 \\ 0 & 1+\delta \end{bmatrix}^{(L-1)} \cdot x$$
- Consider for $\delta > 0 \, \forall \, l$
$$w^{[l]} = \begin{bmatrix} 1-\delta & 0 \\ 0 & 1-\delta \end{bmatrix}$$
i.e., diagonal values again slightly different from the identity matrix. The output $\hat{y}$, in this case will be 'vanishing'
$$\hat{y} = w^{[L]} \cdot \begin{bmatrix} 1-\delta & 0 \\ 0 & 1-\delta \end{bmatrix}^{(L-1)} \cdot x$$

Weight Initialization

Partial solution to exploding or vanishing gradients is by proper weight initialization. We know that $Z = (W_1X_1 + W_2X_2 \ldots W_NX_N + b)$. Larger the value of N, smaller the individual weights should be.
If v^2 be the variance of weights, and N being the number of fanning nodes.

- He Initialization: $v^2 = \frac{2}{N}$ [1]
- Glorot Initialization: $v^2 = \frac{1}{N}$ [2]
- Glorot-Bengio/ Xavier Initialization: $v^2 = \frac{1}{N_{avg}}$, where $N_{avg} = \frac{N_i + N_o}{2}$

[1]Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, ICCV 2015
[2]Understanding the difficulty of training deep feed forward neural networks, AISTATS 2010

Mini-batch gradient descent

Vectorization allows us to efficiently compute on the whole `train` set.

$$X = \left[x^{(1)}, x^{(2)} \dots x^{(m)}\right]_{(n_x,m)} \quad Y = \left[y^{(1)}, y^{(2)} \dots y^{(m)}\right]_{(1,m)}$$

If m is very large, there could be two scenarios:

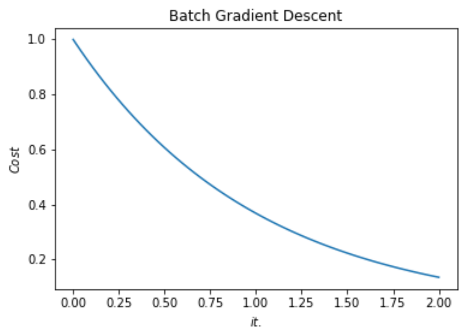
- GPU may not be able to fit the whole `train`.
- Gradient Descent becomes slow computing every iteration.

Mini-batch gradient descent helps mitigate this situation. We split the `train` and `valid` into mini-batches. We process one mini-batch at a time.

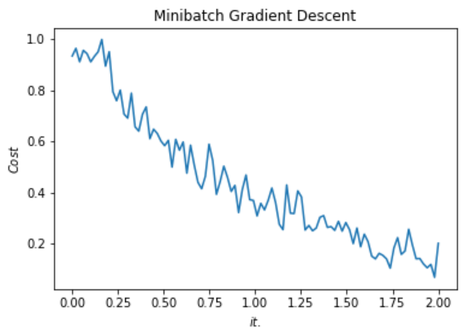
$$\begin{aligned} X &= \left[x^{(1)} \dots x^{(64)} | x^{(65)} \dots x^{(128)} | \dots x^{(m)}\right] \\ &= \left[x^{\{1\}} \mid x^{\{2\}} \mid \dots x^{\{m_b\}}\right] \\ Y &= \left[y^{(1)} \dots y^{(64)} | y^{(65)} \dots y^{(128)} | \dots y^{(m)}\right] \\ &= \left[y^{\{1\}} \mid y^{\{2\}} \mid \dots y^{\{m_b\}}\right] \end{aligned}$$

Training set of m-samples can be split into m_b mini-batches of l samples each. For any mini-batch `t` given as $\{x^{\{t\}}, y^{\{t\}}\}$, the dimension are (n_x, l) and $(1, l)$ respectively. Each mini-batch will compute its forward pass, backward pass and weight updates independently. Running through all mini-batch will constitute an `epoch`.

Epoch is a single pass through the entire training set. In batch gradient descent, each epoch has only one gradient descent step. In mini-batch gradient descent, each epoch consists of numerous gradient descent steps, equivalent to the number of mini-batches. In batch gradient descent, cost is expected to go down monotonically. In mini-batch gradient descent, the cost goes down but not as smoothly because we are processing a subset of the `train` set each time.



Batch G.D



Minibatch G.D

- If minibatch size is m, we get back Batch gradient descent.
- If minibatch size is 1, we get Stochastic Gradient Descent i.e. every sample is a minibatch.

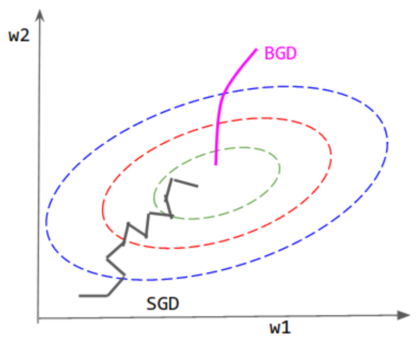
Batch gradient descent progressively moves towards global minimum. SGD is noisier but stays in the neighborhood of global minimum. In practice, we choose a `batch size` in between the 1 and `m` which covers good parts of both.

Batch Gradient Descent

- ✓ Converge always to minima
- ✗ Memory issue, More time per epoch.

Stochastic Gradient Descent

- ✓ Rapid convergence
- ✗ Loss of vectorization
- ✗ Global minimum not guaranteed



Batch vs. Stochastic GD

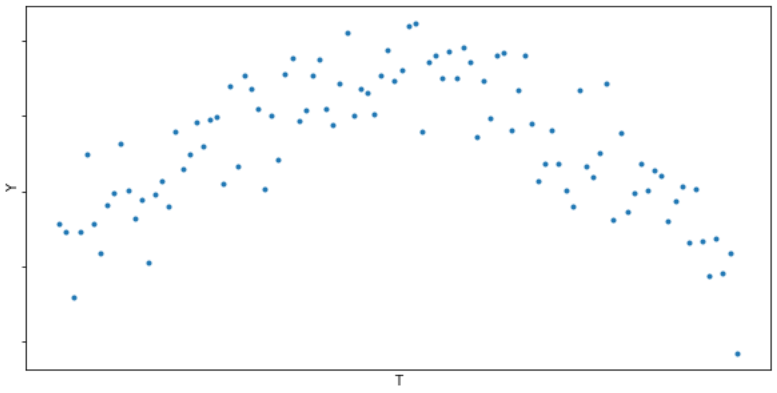
Additional

The applicability of batch or stochastic gradient descent really depends on the error manifold expected. Batch gradient descent computes the gradient using the whole dataset. This is great for convex/smooth error manifolds. In this case, we move somewhat directly towards an optimum solution, either local or global. Additionally, batch gradient descent, given an annealed learning rate, will eventually find the minimum located in it's basin of attraction.
Stochastic gradient descent (SGD) computes the gradient using single sample. Most applications of SGD actually use a minibatch of several samples. SGD works well for error manifolds that have lots of local maxima or minima. In this case, the somewhat noisier gradient calculated using the reduced number of samples tends to jerk the model out of local minima into a region that hopefully is more optimal. Single samples are really noisy, while minibatches tend to average some noise out. Thus, the amount of jerk is reduced when using minibatches. A good balance is struck when the minibatch size is small enough to avoid some of the poor local minima, but large enough that it doesn't keep missing the global minima. One benefit of SGD is that it is computationally a whole lot faster. Large datasets often can't be held in RAM, which makes vectorization much less efficient. Rather, each sample or batch of samples must be loaded, worked with, the results stored, and so on. Minibatch SGD, on the other hand, is usually intentionally made small enough to be computationally tractable. Usually, this computational advantage is leveraged by performing many more iterations of SGD, making many more steps than conventional batch gradient descent. This usually results in a model that is very close to that which would be found via batch gradient descent, or better.
One way to imagine how SGD works is by considering one point that represents the input distribution. The model is attempting to learn that input distribution. Surrounding the input distribution is a shaded area that represents the input distributions of all of the possible minibatches. It is usually a fair assumption that the minibatch input distributions are close in proximity to the true input distribution. Batch gradient descent, at all steps, takes the steepest route to reach the true input distribution. SGD, on the other hand, chooses a random point within the shaded area, and takes the steepest route towards this point. At each iteration, though, it chooses a new point. The average of all of these steps will approximate the true input distribution quite well.*

* https://stats.stackexchange.com/questions/49528/batch-gradient-descent-versus-stochastic-gradient-descent

Exponentially Weighted Average

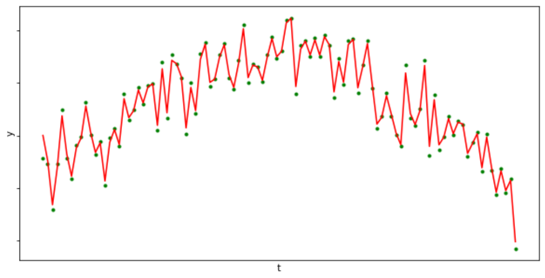
Consider the data distribution shown in figure.



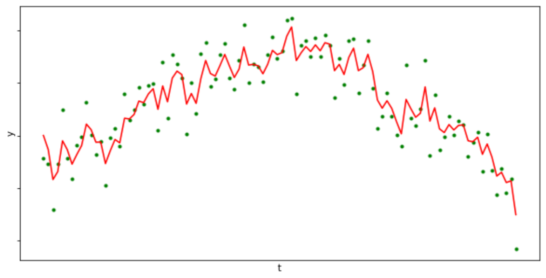
Data distribution of a variable T, say denoting the mean temperature over day of a particular ZIP code

If we use the equation shown, varying the averaging parameter β, we get the curve-fitted plots shown,

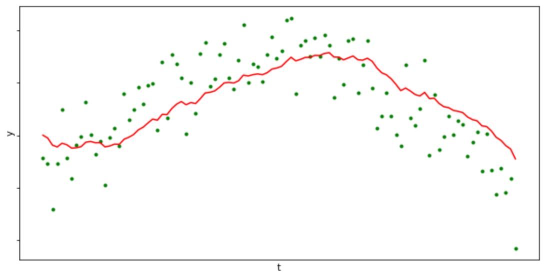
$$v_t = \beta\, v_{t-1} + (1 - \beta)\, y_t$$



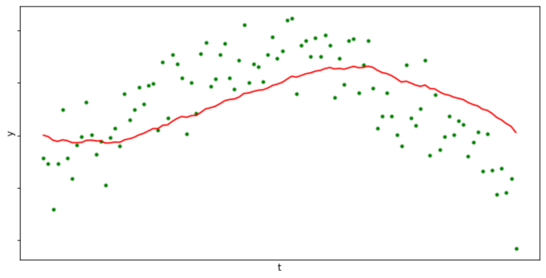
$\beta = 0.1$



$\beta = 0.5$



$\beta = 0.9$



$\beta = 0.95$

The averaging happens over approximately $\frac{1}{1-\beta}$ points i.e. if β is 0.5, the moving average is over 2 points. Higher the value of β, more weightage on previous datapoint sets in.

Note:

- All the coefficients roughly add to 1.

$$\begin{aligned} v_t &= 0.1\,y_t + 0.9\,v_{t-1} \\ &= 0.1\,y_t + 0.9\,[0.1\,y_{t-1} + 0.9\,v_{t-2}] \\ &\dots \\ &= (0.1)\,y_t + (0.1)\,(0.9)\,y_{t-1} + (0.1)\,(0.9)^2\,y_{t-2} + \cdots + (0.1)\,(0.9)^N\,y_1 \end{aligned}$$

- $(1-\beta)^{\frac{1}{\beta}} \approx \frac{1}{e}$. With bigger values of β we are averaging over more points, until they fall below significance level of $\frac{1}{e}$. If $\beta = 0.5$, only two terms from the equation are significant, and the rest are too small. If $\beta = 0.8$ *or* 0.9, up to 5 or 10 terms respectively are relevant in the expression.

Bias Correction

In the equation $v_t = \beta\,v_{t-1} + (1-\beta)\,y_t$, the first few terms will be "pulled down" for v_t, leading to a poor estimate. Changing $v_t \to v_t/(1-\beta^t)$ can mitigate this to a good extent. As $t \to N$, the denominator becomes negligible.

Gradient Descent with Momentum

Compute exponentially weighted average of gradients and use that to update the parameters. Gradient descent takes smoother, controlled jumps on loss surface. Commonly $\beta = 0.9$.

Algorithm: Gradient Descent with Momentum

```
$V_{\partial w} \leftarrow 0$
$V_{\partial b} \leftarrow 0$
while $t \leq m_b$ do
    At iteration $t$ ;
    $\partial w \leftarrow \frac{\partial J}{\partial w}$ ;
    $\partial b \leftarrow \frac{\partial J}{\partial b}$ ;
    $V_{dw} = \beta\,V_{dw} + (1-\beta)\,dw$ ;
    $V_{db} = \beta\,V_{db} + (1-\beta)\,db$ ;
    $w = w - \alpha\,V_{dw}$ ;
    $b = b - \alpha\,V_{db}$ ;
end
```

Bias correction can be omitted because the number of iterations being large can cancel the DC effect of beta.

RMSProp

Root Mean Square propagation by Hinton et al. . RMSprop lies in the realm of adaptive learning rate methods, which have been growing in popularity in recent years, but also getting some criticism. It's famous for not being published, yet being very well-known; most deep learning framework include the implementation of it out of the box.

https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf

Algorithm: RMSProp

```
$S_{\partial w} \leftarrow 0$
$S_{\partial b} \leftarrow 0$
while $t \leq m_b$ do
    $\partial w \leftarrow \frac{\partial J}{\partial w}$ ;
    $\partial b \leftarrow \frac{\partial J}{\partial b}$ ;
    $S_{\partial w} = \beta \, S_{\partial w} + (1 - \beta) \, \partial w^2$ ;
    $S_{\partial b} = \beta \, S_{\partial b} + (1 - \beta) \, \partial b^2$ ;
    $w = w - \alpha \, \frac{\partial w}{\sqrt{S_{\partial w} + \epsilon}}$ ;
    $b = b - \alpha \, \frac{\partial b}{\sqrt{S_{\partial b} + \epsilon}}$ ;
end
```

Derivatives in direction of larger differences will be damped down and prevent oscillatory convergence. Derivatives where differences are small will not be affected much.

ADAM

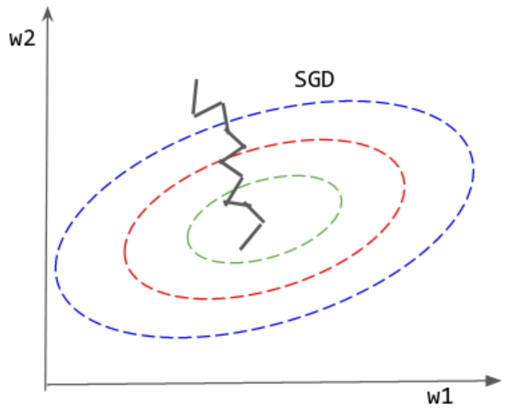
ADAM $\rightarrow$ *Adaptive Moment Estimation.* It uses both the Bias corrected Momentum and RMSProp component for gradient descent.

Algorithm: ADAM

```
$V_{\partial w}, S_{\partial w} \leftarrow 0$
$V_{\partial b}, S_{\partial b} \leftarrow 0$
while $t \leq m_b$ do
    $\partial w \leftarrow \frac{\partial J}{\partial w}$ ;
    $\partial b \leftarrow \frac{\partial J}{\partial b}$ ;
    $V_{\partial w} = \beta_1 \, V_{\partial w} + (1 - \beta_1) \, \partial w$ ;
    $V_{\partial b} = \beta_1 \, V_{\partial b} + (1 - \beta_1) \, \partial b$ ;
    $S_{\partial w} = \beta_2 \, S_{\partial w} + (1 - \beta_2) \, \partial w^2$ ;
    $S_{\partial b} = \beta_2 \, S_{\partial b} + (1 - \beta_2) \, \partial b^2$ ;
    for $V_{\partial w}, V_{\partial b}, S_{\partial w}, S_{\partial b}$ do
        $V_{corr, \partial w} = \frac{V_{\partial w}}{(1 - \beta_1 t)}$;
        $V_{corr, \partial b} = \frac{V_{\partial b}}{(1 - \beta_1 t)}$;
        $S_{corr, \partial w} = \frac{S_{\partial w}}{(1 - \beta_2 t)}$;
        $S_{corr, \partial b} = \frac{S_{\partial b}}{(1 - \beta_2 t)}$;
    end
    $W = W - \alpha \, \frac{dW \; V_{corr, \partial w}}{\sqrt{S_{corr, dw} + \epsilon}}$;
    $b = b - \alpha \, \frac{db \; V_{corr, \partial b}}{\sqrt{S_{corr, db} + \epsilon}}$
end
```

Learning Rate Decay

SGD will keep wandering about and overshooting the global minimum when learning rate is not reduced. With convergence, α needs to ramp down. There are several simple schemes for doing so with a decay factor $\in$ [0,1].



Gradient roaming in vicinity of global minimum.

Schemes ($\gamma \rightarrow$ Decay factor, N $\rightarrow$ epochs)

- $\alpha = \left[\frac{1}{1+\gamma N}\right] \alpha_0$
- $\alpha = \gamma^N \alpha_0$
- $\alpha = \left[\frac{k}{\sqrt{N}}\right] \alpha_0$

Notes

- When doing gradient checking, it is better to do 2-sided gradient computation (from first principles) than one sided since errors are one order smaller.

$$\texttt{2-sided} \longrightarrow f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$$
$$\texttt{1-sided} \longrightarrow f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta)}{\epsilon}$$

- In deep learning, with hundreds of parameters it is unlikely to come across local minimum i.e. spots which are 'troughs' universally for all parameters. Loss function does not get stuck in local minimums. Instead, we come across Saddle Points, which are regions where the gradients are near zero in several directions. The cost function struggles to get out of such places without clever interventions. We can use momentum based methods or using gradient restarts such as SGD-R.

Hyperparameter Tuning

In pre-deep learning era, it was common to perform a `grid search` since the number of variables to tune was quite less. In Deep learning, we choose a random set of variables (within prescribed limits) and improve outcomes by eliminating bad choices or tweaking the favorable ones. Some hyperparameters are more valuable than others. We have to adopt *Coarse to fine* approach to tune. For a hyperparameters, sampling uniformly over all values is not a great approach. We should restrict search over a reasonable bracket.

Batch Normalization

Activations are inputs to the next layer. We normalize these values so that similar to normalizing features, we can make learning process go faster. We normalize $Z^{[l]}$ instead of $A^{[l]}$. Given intermediate values for layer-l, with $\{X^m, Y^m\}$ for the mini-batch,the samples generate $Z^{(1)}, Z^{(1)}, Z^{(1)} \dots Z^{(1)}$ in linear part. We have the following

$$\begin{aligned}\mu &= \frac{1}{m}\sum_m Z^{(i)} = \bar{Z} \\ \sigma^2 &= \frac{1}{m}\sum_m (Z^{(i)} - \mu)^2 \\ &= \frac{1}{m}\sum_m (Z^{(i)} - \bar{Z})^2\end{aligned}$$

We define $Z^{(i)}_{norm}$ as,

$$Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

With $\{\gamma, \beta\}$ as parameters, we could define BatchNorm output $\tilde{Z}^{(i)}$ as,

$$\tilde{Z}^{(i)} = \gamma Z^{(i)}_{norm} + \beta.$$

If $\gamma = \sqrt{\sigma^2 + \epsilon}, \beta = \mu$, we can recover back $Z^{(i)}$ (as $Z^{(i)} \approx \tilde{Z}^{(i)}$). We usually use $\tilde{Z}^{(i)}$ in place of $Z^{(i)}$ because it is easier to center the values to our desired effect. $\{\gamma, \beta\}$ allows $\tilde{Z}$ to have flexible values. It is especially useful when working with Sigmoid or Arctan activations, which have narrow linear regions. When using batch normalization, $b^{[l]}$ is not required because the normalization nulls its effect. It is more effective to retain $\beta^{[l]}$

Mechanism for BatchNorm

By normalizing the features, we have attempted speeding up the learning previously. BatchNorm produces similar effect, but with hidden layers. A second reasoning emerges from `Covariate Shift`. If the weights are more immune to covariate shifts, i.e., change in input features, network will perform better in predictions. This is because BatchNorm-processed networks have their $\tilde{Z}^{(l)}$ centered around a well-defined center dictated by $\{\gamma, \beta\}$, instead of being left to its own devices. This centering is independent of the mean and variance which computed $Z^{(l)}_{norm}$. BatchNorm is robust to sudden changes in input features, which may change Z values significantly.

Batch Normalization also has a small regularization effect. Each mini-batch output is scaled by mean and variance computed $\underline{\text{only}}$ for the mini-batch. This adds some noise in the values of $Z^{[l]}$ across mini-batches (similar to Dropout which creates random fluctuations across mini-batches by knocking out some nodes). During test-time, we need to come up with an estimate of the mean and standard deviation for the intermediate values. This can be done by exponentially-weighted averages of μ, σ across mini-batches.

Softmax Regression

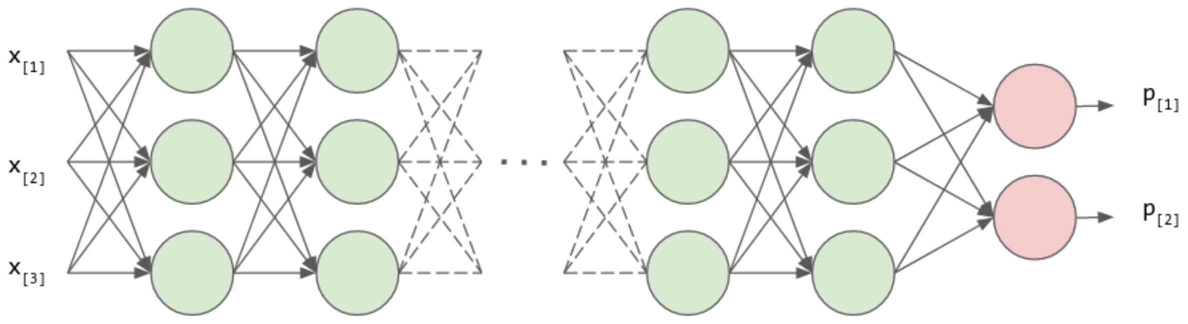
Softmax regression is generalization of Logistic regression over multiple classes. For DNN, the output layer will have the same number of nodes as the number of classes. Their output corresponds to the probability of the classes in question, given the input features. These probabilities should sum to 1. Softmax activation is used in the L^{th} layer of the multi-class classification. The function is defined as,

$$\begin{aligned} A^{[L]} &= \sigma(z_i) \\ &= \frac{e^{(z_i)}}{\sum\limits_{i=1}^{C} e^{(z_i)}} \; \forall\; i \; \in C, \sigma : \mathbb{R}^C \to \mathbb{R}^C \end{aligned}$$

Softmax is used in place of `HardMax` which outputs Boolean values for class prediciton. Good for doing one-hot coding.

Loss function

$\mathbb{L}(\hat{y}, y) = -\sum\limits_i y_i \, log(\hat{y}_i)$ During gradient descent, $dz^{[l]} = \hat{y} - y$



Softmax final layer