COMPILATION OF METHODS TO IMPROVE CLASSIFICATION PERFORMANCE

1) Augmentation tricks worth trying : Cropping images, padding images, resizing + Light augmentations library (albumentations), Normalization

i) Cropping+Resizing(with and w/o padding)**:**The images can be randomly or specifically(as per the use case) cropped to focus on the parts which will help it improve performance on the positive class. This cropped image is then resized to the image size required as input for the model. One thing that can also be tried is after cropping images one could pad the image with zeros and then resize the image or reflection padding can be used for the same. This in turn will lead to information retainment when the image goes through the pooling and convolutional layers.

When to use ? It is useful when the images have smaller objects that would help in classification. Random crops are useful when we expect the model to learn all the spatial features in bits and pieces.

When not to use ? On plotting heatmaps , one could figure how does the model localise whether its predicting the right objects in the image, if your model is doing the needful, this might not turn out to be useful

ii) Normalization: And some channels might tend to be really bright, some might tend to be really not bright at all, some might vary a lot, and some might not very much at all. It really helps train a deep learning model if each one of those red, green and blue channels has a mean of zero and a standard deviation of one.

iii) Light augmentations using albumentations: Apart from general affine transformations like scaling, rotation, shifting we can do certain light augmentations like changing the brightness, sharpness, contrast, introducing gaussian noise tend lead to better performance when the images vary pixel intensity-wise. Many instances or examples for the same can be found on Kaggle where this might not improve performance massively but tends to perform reasonably well.

2) One cycle learning:

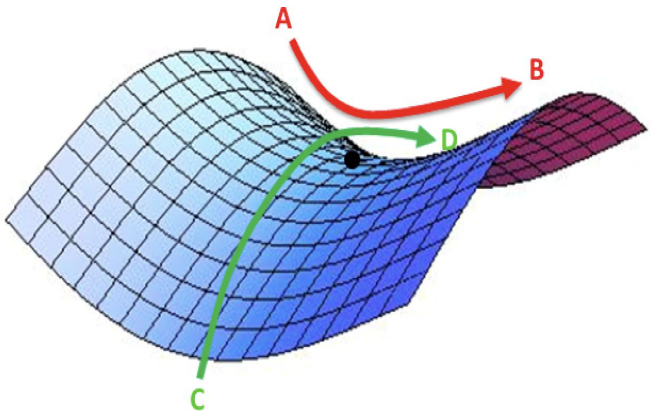
In the paper "A disciplined approach to neural network hyper-parameters: Part 1 — learning rate, batch size, momentum, and weight decay" , Leslie Smith describes approach to set hyper-parameters (namely learning rate, momentum and weight decay) and batch size. In particular, he suggests 1 Cycle policy to apply learning rates.

Author recommends to do one cycle of learning rate of 2 steps of equal length. We choose maximum learning rate using range test. We use lower learning rate as 1/5th or 1/10th of maximum learning rate. We go from lower learning rate to higher learning rate in step 1 and back to lower learning rate in step 2. We pick this cycle length slightly less than total number of epochs to be trained. And for the last remaining iterations, we annihilate learning rate way below lower learning rate value(1/10 th or 1/100 th).

The motivation behind this is that, during the middle of learning when learning rate is higher, the learning rate works as regularisation method and keep network from overfitting. This helps the network to avoid steep areas of loss and land better flatter minima.

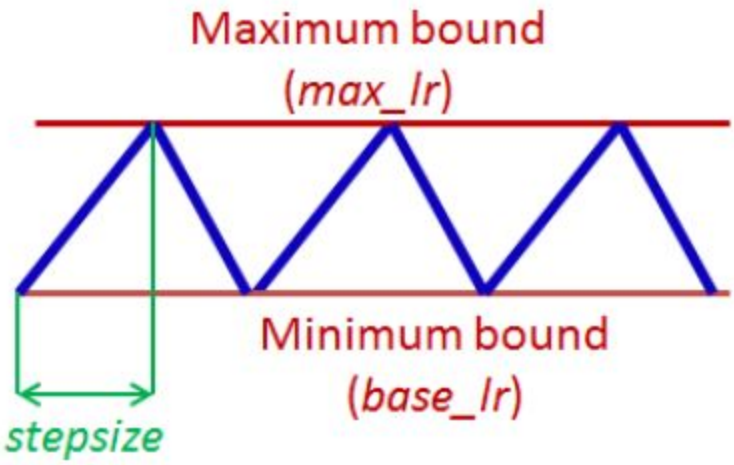
Try finding the right values for momentum, weight decay with trial and error.

3) Cyclic Learning Rate (3 strategies):



CLR is to cycle the learning rate between lower bound and upper bound during complete run. Conventionally , the learning rate is decreased as the learning starts converging with time. Intuitively, it is helpful to oscillate the learning rate towards higher learning rate. As the higher learning rate may help to get out of saddle points. If saddle point is elaborate plateau, the lower learning rates might not be able get gradient out of saddle point.

Cycle is the number of iterations where we go from lower bound learning rate to higher bound and back to lower bound. Cycle may not have boundary on epoch, but in practice it usually does. Step Size is half of cycle. So Stepsize is the number of iterations where we want learning rate to go from one bound to the other. There are three different strategies including triangular, triangular2 and exponential which is a trial and error thing



i) Choosing the right learning rate:
The range test run for few epochs to find out good learning rate, where we train from some low learning rate and increase the learning rate after each mini-batch till the loss value starts to explode.
The idea is to start with small learning rate (like 1e-4, 1e-3) and increase the learning rate after each mini-batch till loss starts exploding. Once loss starts exploding stop the range test run. Plot the learning rate vs loss plot. Choose the learning rate one order lower than the learning rate where loss is minimum(if loss is low at 0.1, good value to start is 0.01). This is the value where loss is still decreasing.

Reference:
https://medium.com/@psk.light/one-cycle-policy-cyclic-learning-rate-and-learning-rate-range-test-f90c1d4d58da
https://github.com/titu1994/keras-one-cycle

Code reference:
https://github.com/keras-team/keras-contrib/blob/master/keras_contrib/callbacks/cyclical_learning_rate.py

4) **Dealing with data imbalance:**

i) Sampling the dataset:

There are two main methods that you can use to even-up the classes:
You can add copies of instances from the under-represented class called over-sampling (or more formally sampling with replacement), or
You can delete instances from the over-represented class, called under-sampling.

These approaches are often very easy to implement and fast to run. They are an excellent starting point. In fact, I would advise you to always try both approaches on all of your imbalanced datasets, just to see if it gives you a boost in your preferred accuracy measures. (Use SMOTE)

ii) Class weight penalisation: *(Already on Wiki)*
iii) Use of focal loss instead of cross-entropy:
https://leimao.github.io/blog/Focal-Loss-Explained/

5) **Dealing with misclassified samples:** Pick out wrongly classified samples from the training set , figure out a pattern if there's any. Otherwise fine tune the model with those samples using High LR or transfer learn the original model.

6) **Use of custom loss function:** Combine the already available loss function with some values to penalise in a certain way as to prioritize any of the classes. (*Code already shared on Wiki)*

7) **Discriminative Learning Rates:** Using low learning rates for the initial part of the network so that it generalizes better and does not lose its features over a period of time and use high learning rate to finetune the later part of network to specialize better and converge faster. Basically, slicing the networks and learning rates respectively.

8) **Progressive Resizing:**
It is the technique to sequentially resize all the images while training the CNNs on smaller to bigger image sizes.. A great way to use this technique is to train a model with smaller image size say 64x64, then use the weights of this model to train another model on images of size 128x128 and so on. Each larger-scale model incorporates the previous smaller-scale model layers and weights in its architecture. The smaller sized images are generalise better and later the larger images are bound to learn specific features.
https://towardsdatascience.com/boost-your-cnn-image-classifier-performance-with-progressive-resizing-in-keras-a7d96da06e20

9) **Cosine Annealing**
In SGDR, the learning rate is reset at the start of each epoch to the original chosen value which decreases over the epoch as in *Cosine Annealing*. The main benefit of this is that since the learning rate is reset at the start of each epoch, the learner is capable of jumping out of a local minima or a saddle point it may be stuck in.

10) **Use of denseNet for smaller datasets or use of ResNeXt, SE-ResNext, EfficientNet:**
Using DenseNet architecture for smaller datasets as it keeps the original pixels intact while going down the network. Generally a helpful backbone for UNet while solving segmentation problems.

11) Ensembling:

i) General (Max-voting or average of predictions) - Different architectures or same architecture with different hyperparameters like different loss functions and optimizers

ii) Cross validation ensemble: Ensembling models cross-validated on the data using k-fold cross-validation

iii) Snapshot ensembling: While training the model, you can save the best weights every time, and then ensemble these models at the end of it. The intuition behind this is every time the model is optimized in local minima it has different weights and different data patterns which are being picked up and might change over time.

12) Test time Augmentation(TTA) and 10-crop technique:

Test Time Augmentation involves taking a series of different versions of the original image and passing them through the model. The average output is then calculated from the different versions and given as the final output for the image. The augmentations are generally the same which are used while training the model

A similar technique called 10-crop testing was used in the past. The 10-crop technique involves cropping the original image along the four corners and once along the centre giving 5 images. Repeating the same for it's inverse, gives another 5 images, a total of 10 images. Test time augmentation is however faster then 10-crop technique. In our case 5-crops could do the job for us.

13) Combining Loss functions(Add or weighted combination)

A combination of loss functions can be used to optimize the model. Needs some effort to go into the detail of which loss functions could be combined for classification problems.

14) Mixed Precision training-PyTorch:

15) Gradient Accumulation-PyTorch:

From Hidden Trends in ML system Paper

* Never cascade models for improving the original model for some new type fof problem. It increases the dependency on the original model.

* Ensembles better than single models , different models learn different features , it helps while there are changes in data.

Optimizers and their comparison

1) Gradient Descent and its variants:

The traditional *Batch Gradient Descent* will calculate the gradient of the whole Data set but *will perform only* ***one update , hence it can be very slow and hard to control for datasets which are very very large and don't fit in the Memory***.
Due to **frequent updates while using Stochastic Gradient Descent**, parameters updates have **high variance** and causes the **Loss function to fluctuate to different intensities**. This is actually a good thing because it helps us **discover new and possibly better local minima** , whereas Standard Gradient Descent will only converge to the minimum of the basin as mentioned above.
An improvement to avoid all the problems and demerits of SGD and standard Gradient Descent would be to use **Mini Batch Gradient Descent** as it takes the best of both techniques and performs an update for every batch with n training examples in each batch.

Challenges faced while using Gradient Descent and its variants —

1. Choosing a proper learning rate can be difficult.

   Low LR- Baby steps, slow convergence

   High LR: Might diverge and never give optimal weights

2. Additionally, the **same learning rate applies to all parameter updates**. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
3. Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous **sub-optimal local minima**. Actually, Difficulty arises in fact not from local minima but from **saddle points**, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

2) Adaptive Optimizers

1. Adagrad:

It simply allows the learning Rate -**η** to **adapt** based on the parameters. So it makes big updates for infrequent parameters and small updates for frequent parameters. For this reason, it is **well-suited for dealing with sparse data.**
The main benefit of Adagrad is that we don't need to manually tune the

learning Rate. Most implementations use a default value of 0.01 and leave it at that.

Disadvantage

Its main weakness is that its **learning rate-η is always Decreasing and decaying.**Because we know that as the learning rate gets smaller and smaller the ability of the Model to learn quickly decreases and which gives very slow convergence and takes very long to train and learn i.e learning speed suffers and decreases. (Vanishing LR problem

2. AdaDelta

It is an extension of AdaGrad which tends to remove the *decaying learning Rate* problem of it. Instead of accumulating all previous squared gradients, *Adadelta* **limits the window of accumulated past gradients to some fixed size w.**

3. Adam

Adam stands for Adaptive Moment Estimation. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta ,Adam also keeps an exponentially decaying average of past gradients

Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quiet Fast and efficient and also it rectifies every problem that is faced in other optimization techniques such as vanishing Learning rate , slow convergence or High variance in the parameter updates which leads to fluctuating Loss function

4. RMSProp

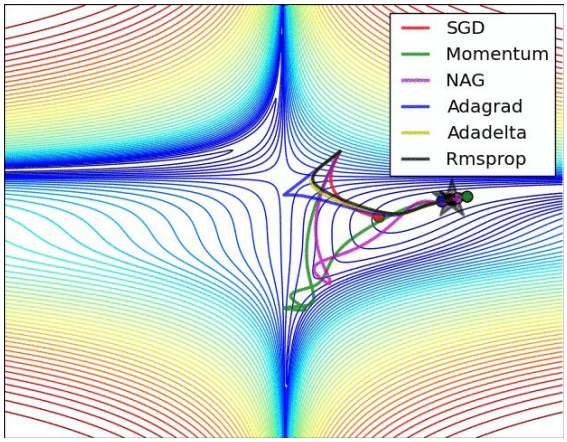
- RMSProp tries to resolve Adagrad's radically diminishing learning rates by **using a moving average of the squared gradient**. It utilizes the magnitude of the recent gradient descents to normalize the gradient.
- In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
- RMSProp divides the learning rate by the average of the exponential decay of squared gradients

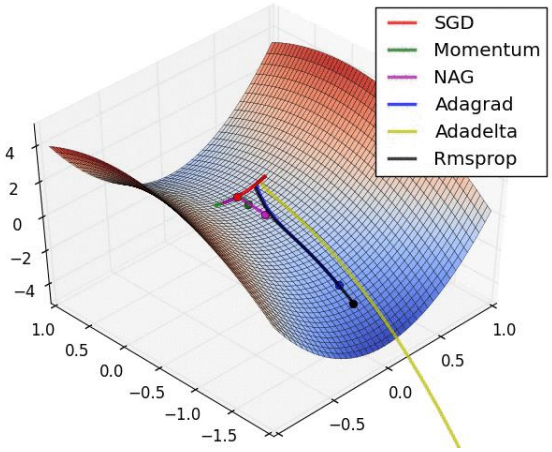
Visualization of algorithms

The following two figures provide some intuitions towards the optimization behaviour of the presented optimization algorithms.

In Figure 4a, we see the path they took on the contours of a loss surface (the Beale function). All started at the same point and took different paths to reach the minimum. **Note that Adagrad, Adadelta, and RMSprop headed off immediately in the right direction and converged similarly fast, while Momentum and NAG were led off-track, evoking the image of a ball rolling down the hill. NAG, however, was able to correct its course sooner due to its increased responsiveness by looking ahead and headed to the minimum.**

Figure 4b shows the **behaviour of the algorithms at a saddle point**, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope, which pose a difficulty for SGD as we mentioned before. **Notice here that SGD, Momentum, and NAG find it difficulty to break symmetry, although the latter two eventually manage to escape the saddle point, while Adagrad,RMSprop, and Adadelta quickly head down the negative slope, with Adadelta leading the charge**.





(a) SGD optimization on loss surface contours (b) SGD optimization on saddle point

As we can see, the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios

Which optimizer should we use?

The question was to choose the best optimizer for our Neural Network Model in order to converge fast and to learn properly and tune the internal parameters so as to minimize the Loss function .

***Adam* works well in practice and outperforms other Adaptive techniques.**

If your input data is sparse then methods such as SGD,NAG and momentum are inferior and perform poorly. For sparse data sets one should use one of the *adaptive learning-rate* methods. An additional benefit is that we won't need to adjust the learning rate but likely achieve the best results with the default value.An additional benefit is that you will not need to tune the learning rate but will likely achieve the best results with the default value.

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numerator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances.

Empirically, its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.

Interestingly, many recent papers use vanilla SGD without momentum and a simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. Consequently, if you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.

References:

https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

http://cs231n.github.io/neural-networks-3/#baby

https://arxiv.org/abs/1910.05446

Use Adam or RMSprop in general

Comparison of common loss functions for classification

The hinge loss penalizes predictions not only when they are incorrect, but even when they are correct but not confident. It penalizes gravely wrong predictions significantly, correct but not confident predictions a little less, and only confident, correct predictions are not penalized at all. Let's formalize this by writing out the hinge loss in the case of binary classification:

$$\sum_i \max(0, 1 - y_i * h_\theta(x_i))$$

The main difference between the hinge loss and the cross entropy loss is that the former arises from trying to maximize the margin between our decision boundary and data points - thus attempting to ensure that each point is correctly and confidently classified*, while the latter comes from a maximum likelihood estimate of our model's parameters. The softmax function, whose scores are used by the cross entropy loss, allows us to interpret our model's scores as relative probabilities against each other.

1. Regression Loss Functions
    1. Mean Squared Error Loss
    2. Mean Squared Logarithmic Error Loss
    3. Mean Absolute Error Loss
2. Binary Classification Loss Functions
    1. Binary Cross-Entropy
    2. Hinge Loss
    3. Squared Hinge Loss
3. Multi-Class Classification Loss Functions
    1. Multi-Class Cross-Entropy Loss
    2. Sparse Multiclass Cross-Entropy Loss -- Too many labels
    3. Kullback Leibler Divergence Loss

Final words:

The MSE loss comes from a probabilistic interpretation of the regression problem, and the cross-entropy loss comes from a probabilistic interpretation of binary classification. The MSE loss is therefore better suited to regression problems, and the cross-entropy loss provides us with faster learning when our predictions differ significantly from our labels, as is generally the case during the first several iterations of model training.

Other loss functions can be found in Kaggle kernels

Comparison of activations

1) **Sigmoid:**

   Advantage of this function is that it produces a value in the range of (0,1) when encountered with (- infinite, + infinite) as in the linear function. So the activation value does not vanish

   Problem:

   If we look carefully at the graph towards the ends of the function, y values react very little to the changes in x. The derivative values in these regions are very small and converge to 0. This is called **the vanishing gradient** and the learning is minimal. if 0, no learning. When slow learning occurs, the optimization algorithm that minimizes error can be attached to local minimum values and cannot get maximum performance from the artificial neural network model.

2) **Hyperbolic Tangent:**

   The advantage over the sigmoid function is that its derivative is more steep, which means it can get more value. This means that it will be more efficient because it has a wider range for faster learning and grading.

   Problem:

   The problem of gradients at the ends of the function continues

3) Softmax:

Used for multiclass classification in general. Works the best with crossentropy loss .

How to form an ensemble ? (Source: cs231n)

In practice, one reliable approach to improving the performance of Neural Networks by a few percent is to train multiple independent models, and at test time average their predictions. As the number of models in the ensemble increases, the performance typically monotonically improves (though with diminishing returns). Moreover, the improvements are more dramatic with higher model variety in the ensemble. There are a few approaches to forming an ensemble:

- **Same model, different initializations**. Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization. The danger with this approach is that the variety is only due to initialization.
- **Top models discovered during cross-validation**. Use cross-validation to determine the best hyperparameters, then pick the top few (e.g. 10) models to form the ensemble. This improves the variety of the ensemble but has the danger of including suboptimal models. In practice, this can be easier to perform since it doesn't require additional retraining of models after cross-validation
- **Different checkpoints of a single model**. If training is very expensive, some people have had limited success in taking different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble. Clearly, this suffers from some lack of variety, but can still work reasonably well in practice. The advantage of this approach is that is very cheap.
- **Running average of parameters during training**. Related to the last point, a cheap way of almost always getting an extra percent or two of performance is to maintain a second copy of the network's weights in memory that maintains an exponentially decaying sum of previous weights during training. This way you're averaging the state of the network over last several iterations. You will find that this "smoothed" version of the weights over last few steps almost always achieves better validation error. The rough intuition to have in mind is that the objective is bowl-shaped and your network is jumping around the mode, so the average has a higher chance of being somewhere nearer the mode.

One disadvantage of model ensembles is that they take longer to evaluate on test example. An interested reader may find the recent work from Geoff Hinton on "Dark Knowledge" inspiring, where the idea is to "distill" a good ensemble back to a single model by incorporating the ensemble log likelihoods into a modified objective.