Session 11 - Class Activation Maps, LRs and Optimizers

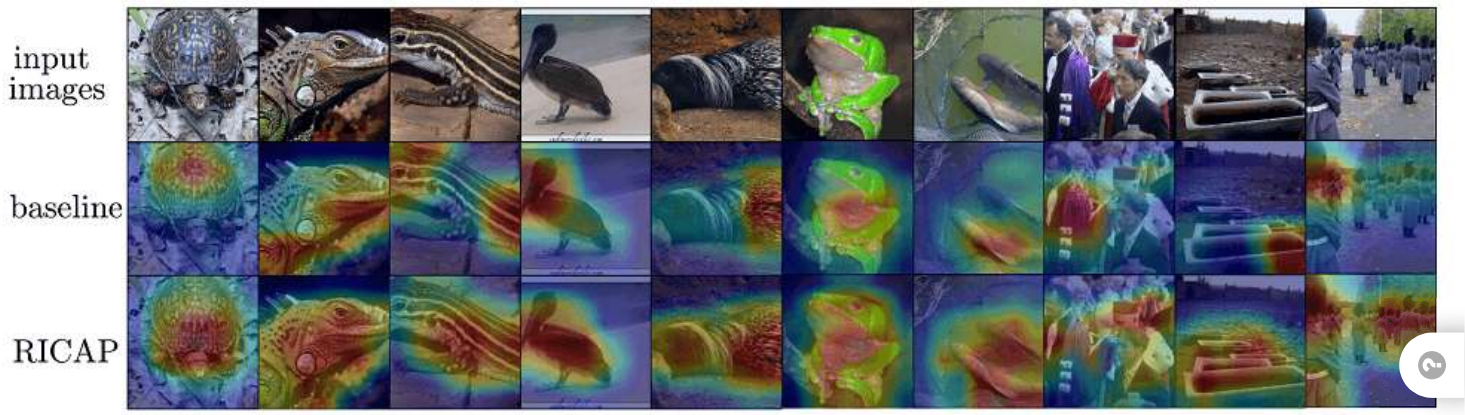
- Due Saturday by 9am
- Points 0
- Available after Apr 6 at 12am

Session 11 - CAMs, LRs, and Optimizers

- Class activation maps
- GradCAM
- Learning Rates
- Weight updates
- Constant vs Adaptive Learning Rates
- SGD
  - Gradient Perturbation
  - Momentum & Nesterov Momentum
- RMSProp
- Adam
- Best Optimizer
- LRs
  - One Cycle Policy
  - Reduce LR on Plateau
- What kind of minima do we want?
- Assignment

Class Activation Maps

Look at the image below. This was in the RICAP paper where they claimed that their augmentation strategy is better as it allows the network to "see" better. How did they figure out what was network looking at?



Above we see something called GradCAM's output, one of the Class Activation Mapping algorithms.

Class activation maps (CAMs) are visualizations that highlight which regions of an input image contribute the most to the classification decision made by a deep neural network. They are helpful in understanding how the model is making predictions, identifying model weaknesses and biases, and debugging and improving the model's performance. CAMs can also be used for generating more human-interpretable explanations for the predictions made by a model.

Let's check out the **library (https://jacobgil.github.io/pytorch-gradcam-book/introduction.html)** that you'd be using to see why they are useful.

Along with the above, they can also be used for further diagnostics:

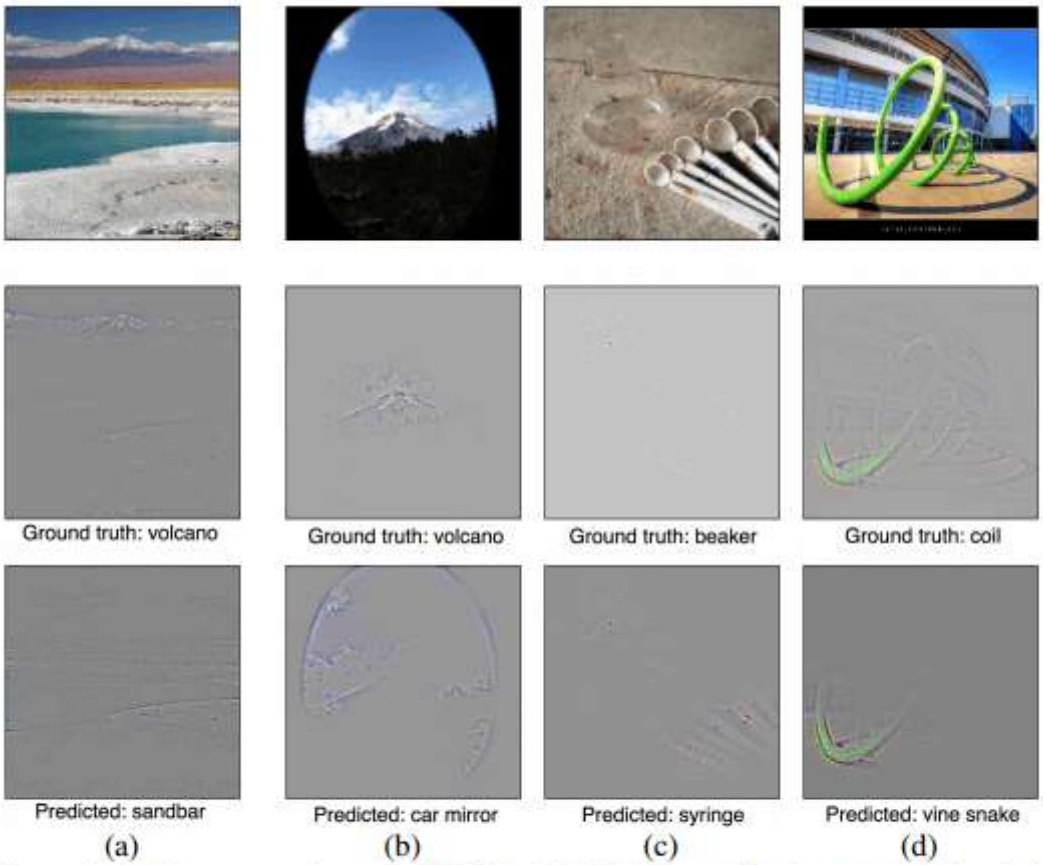


Figure 4: In these cases the model (VGG-16) failed to predict the correct class in its top 1 (a and d) and top 5 (b and c) predictions. Humans would find it hard to explain some of these predictions without looking at the visualization for the predicted class. But with Grad-CAM, these mistakes seem justifiable.

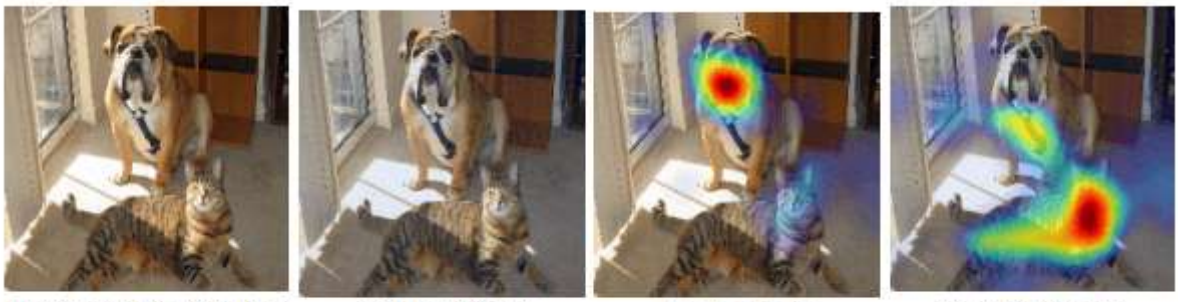


Figure 5: (a-b) Original image and the generated adversarial image for category "airliner". (c-d) Grad-CAM visualizations for the original categories "tiger cat" and "boxer (dog)" along with their confidence. Inspite of the network being completely fooled into thinking that the image belongs to "airliner" category with high confidence (>0.9999), Grad-CAM can localize the original categories accurately.

How do we create adversarial images?

1. Choose an input image and its label: Start with a clean image, labeled as a specific class, which you want the network to misclassify.
2. Define the loss function: The loss function should measure the difference between the predicted class and the target class, usually the opposite of the true class.
3. Calculate the gradient of the loss with respect to the input image: This step is done using backpropagation and it gives information on how the loss changes as the input image is changed.
4. Perturb the input image: Add a small, but targeted perturbation to the input image, proportional to the gradient calculated in the previous step. This is done to maximize the loss and hence, change the predicted class.
5. Repeat steps 3 and 4 until the desired misclassification is achieved: You may need to repeat these steps multiple times to generate a strong enough perturbation that causes the network to misclassify the image.
6. Clip the pixel values of the perturbed image if needed: Clipping ensures that the perturbed image remains visually similar to the original image and falls within a specified range of pixel values.

"panda"
57.7% confidence

"gibbon"
99.3% confidence

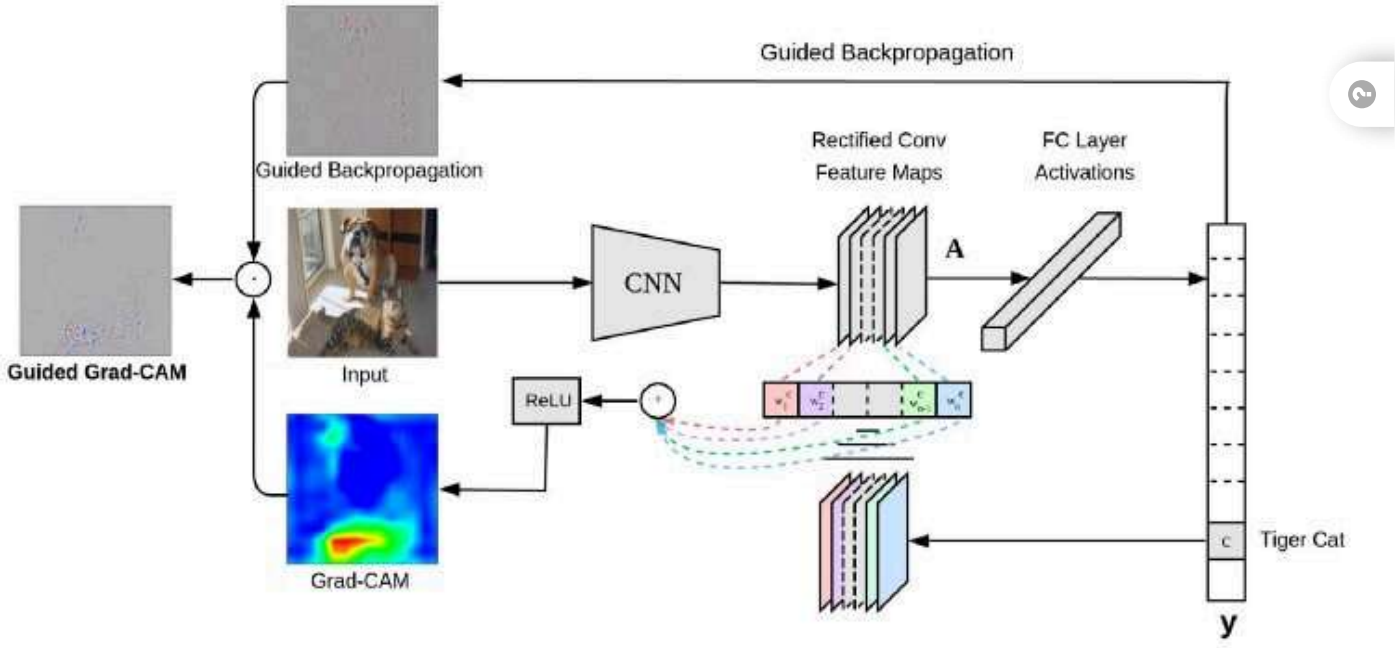
In this section we provide qualitative examples showing the explanations from the two models trained for distinguishing doctors from nurses- model1 which was trained on images (with an inherent bias) from a popular search engine, and model2 which was trained on a more balanced set of images from the same search engine.

As shown in Fig. A4, Grad-CAM visualizations of the model predictions show that the model had learned to look at the person's face / hairstyle to distinguish nurses from doctors, thus learning a gender stereotype.

Using the insights gained from the Grad-CAM visualizations, we balanced the dataset and retrained the model. The new model, model2 not only generalizes well to a balanced test set, it also looks at the right regions.

Figure A4: Grad-CAM explanations for model1 and model2. In (a-c) we can see that even though both models made the right decision, the biased model (model1) was looking at the face of the person to decide if the person was a nurse (b), whereas the unbiased model, was looking at the short sleeves to make the decision (c). For example image (d) and example (g) the biased model made the wrong prediction (misclassifying a doctor as a nurse) by looking at the face and the hairstyle (e, h), where as the unbiased model made the right prediction looking at the white coat, and the stethoscope (f, i).

GRADCAM



Grad-CAM algorithm for visualizing the important regions of an input image that contribute to the predictions of a VGG model can be implemented in the following steps:

1. Load the pre-trained VGG model: The first step is to load a pre-trained VGG model, which has already been trained on a large dataset.
2. Load the input image: Next, you would load the input image that you want to analyze.
3. Inferring: Run the model's forward pass to obtain the final class scores and prediction.

4. Output of the last Convolutional layer: Extract the feature maps of the last Convolutional layer before the final fully connected layers.
5. Computing Gradients: Compute the gradients of the final class scores with respect to the feature maps.
6. Pooling the Gradients: Pool the gradients spatially, typically by taking the average along the height and width dimensions.
7. Weighing the outputs: Multiply the feature maps with the pooled gradients to get the weighted feature maps.
8. Averaging Feature Maps along Channels: Sum the weighted feature maps along the channel dimension to get the final activation map.
9. Normalizing the Heatmap: Normalize the activation map to obtain the final Grad-CAM heatmap, which highlights the regions of the input that were important for the prediction.

This heatmap can be overlaid on the input image to visualize the regions that were important for the model's prediction.

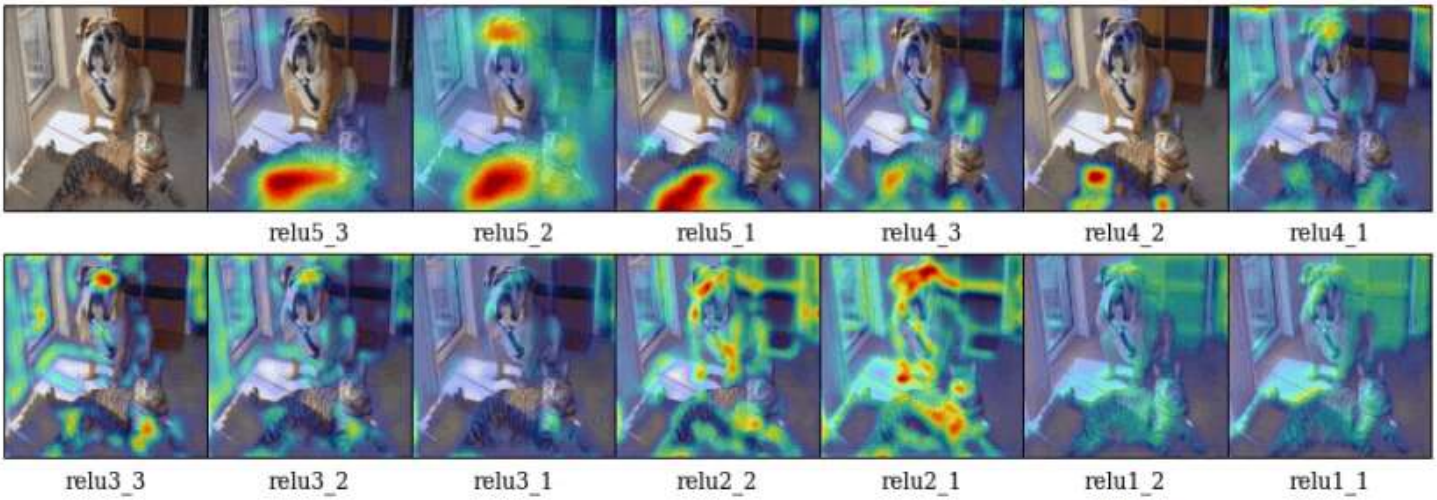


Figure A6: Grad-CAM at different convolutional layers for the 'tiger cat' class. This figure analyzes how localizations change qualitatively as we perform Grad-CAM with respect to different feature maps in a CNN (VGG16 [45]). We find that the best looking visualizations are often obtained after the deepest convolutional layer in the network, and localizations get progressively worse at shallower layers. This is consistent with our intuition described in Section 3 of main paper.

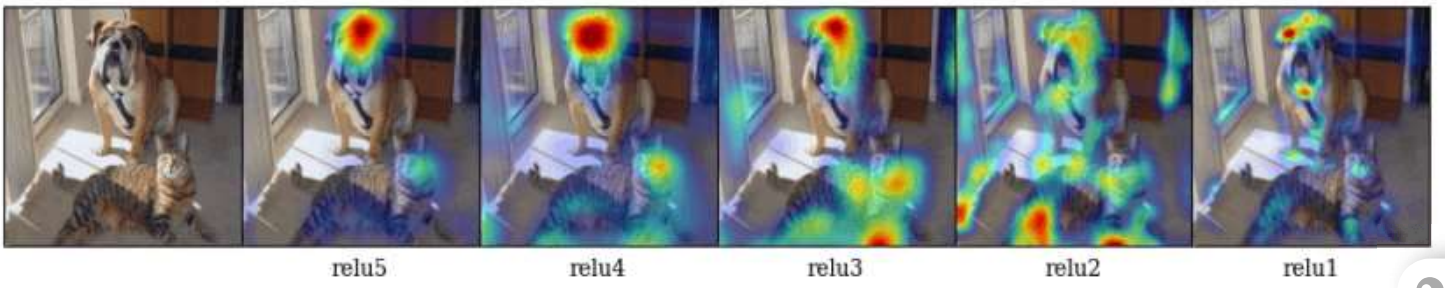


Figure A7: Grad-CAM localizations for "tiger cat" category for different rectified convolutional layer feature maps for AlexNet.

Learning Rates

The learning rate is a hyperparameter that controls how much we are adjusting the weights of our network with respect to the loss gradient.

The lower the value, the slower we travel along the downward slope.

If we move slowly, there is a good chance we will reach our minima, but it would take a long time to converge. If we move fast, we might reduce loss faster, but the minima would elude us.

This is how we calculate loss:

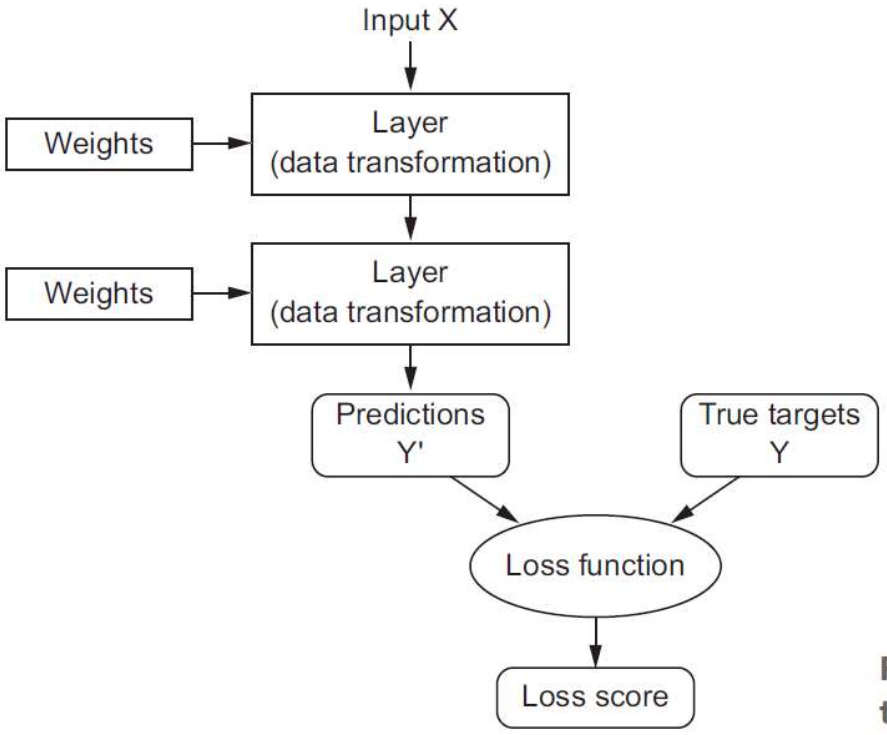
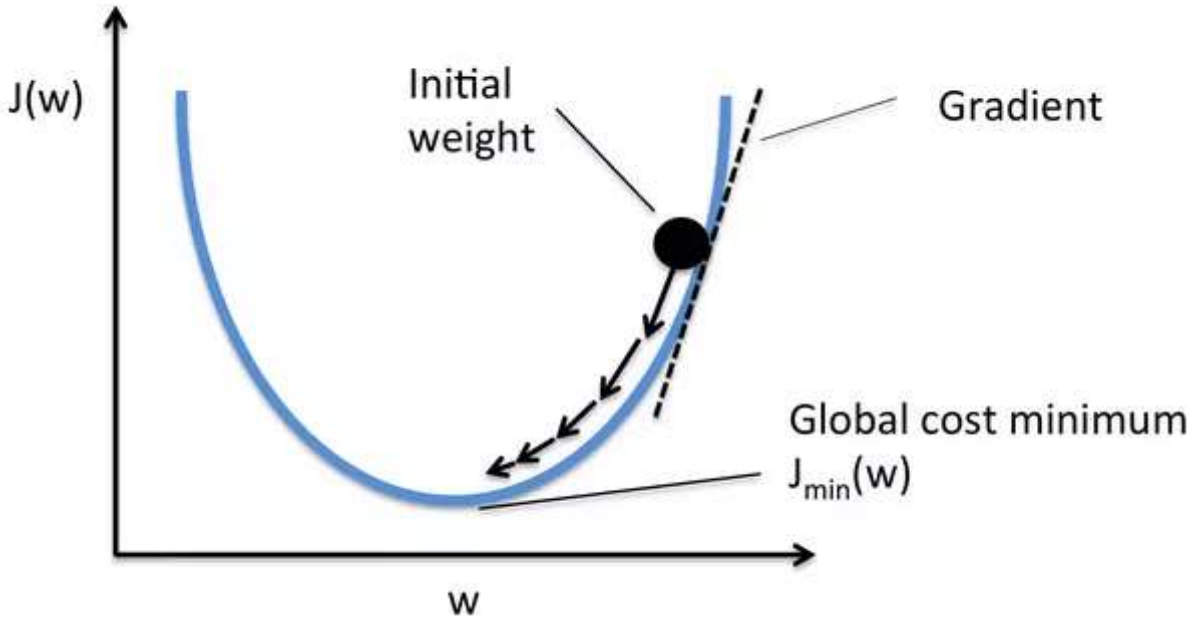


Figure 1.8 A loss function measures the quality of the network's output.

This is how the gradients are related to our loss function



Weight Updates

$$\omega = \omega - \alpha \left(\frac{\delta L}{\delta \omega} \right)$$

Mathematically gradient is a partial derivate of the loss function w.r.t the weights. We use a negative gradient. Alpha is the learning rate.

Let's spend some time on this and understand what that negative number does!

Constant vs Adaptive Learning Rates

Constant Learning Rates

Most widely used Optimization Algorithm, the Stochastic Gradient Descent falls under this category.

$$W^{(k+1)} = W^{(k)} - \eta \cdot (\Delta J(W))$$

Here η is called a "learning rate" which is a hyperparameter and has to be tuned. Small η is a snail and large is a missile. We need to find the correct one.

We can improve this situation through momentum (covered later).

Adaptive Learning Rates

Adaptive gradient descent algorithms such as Adagrad, Adadelta, RMSProp, and Adam, provide an alternative to classical SGD.

Gradient Descent: calculates the gradient for the whole dataset and updates in a direction opposite to the gradients until we find local minima. S**tochastic Gradient Descent** performs a parameter update for each batch instead of the whole dataset. This is much faster and can be further improved through momentum and learning rate finder.

Adagrad is preferable for a sparse data set as it makes big updates for infrequent parameters and small updates for frequent parameters. It uses a different learning rate for each parameter at a time step based on the past gradients that were computed for that parameter. Thus we do not need to manually tune the learning rate.

Adam stands for Adaptive Moment Estimation. It also calculates a different learning rate. Adam works well in practice, is faster, and outperforms other techniques*.

Source (https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c)

SGD or Stochastic Gradient Descent

SGD refers to an algorithm that operates on a batch size equal to 1, while **Mini-Batch Gradient Descent** is adopted when the batch size is greater than 1.

We will refer to MBGD as SGD.

Let us assume our loss function for a single sample is:

$$L(\overline{x_i}, \overline{y_i}; \overline{\theta_i})$$

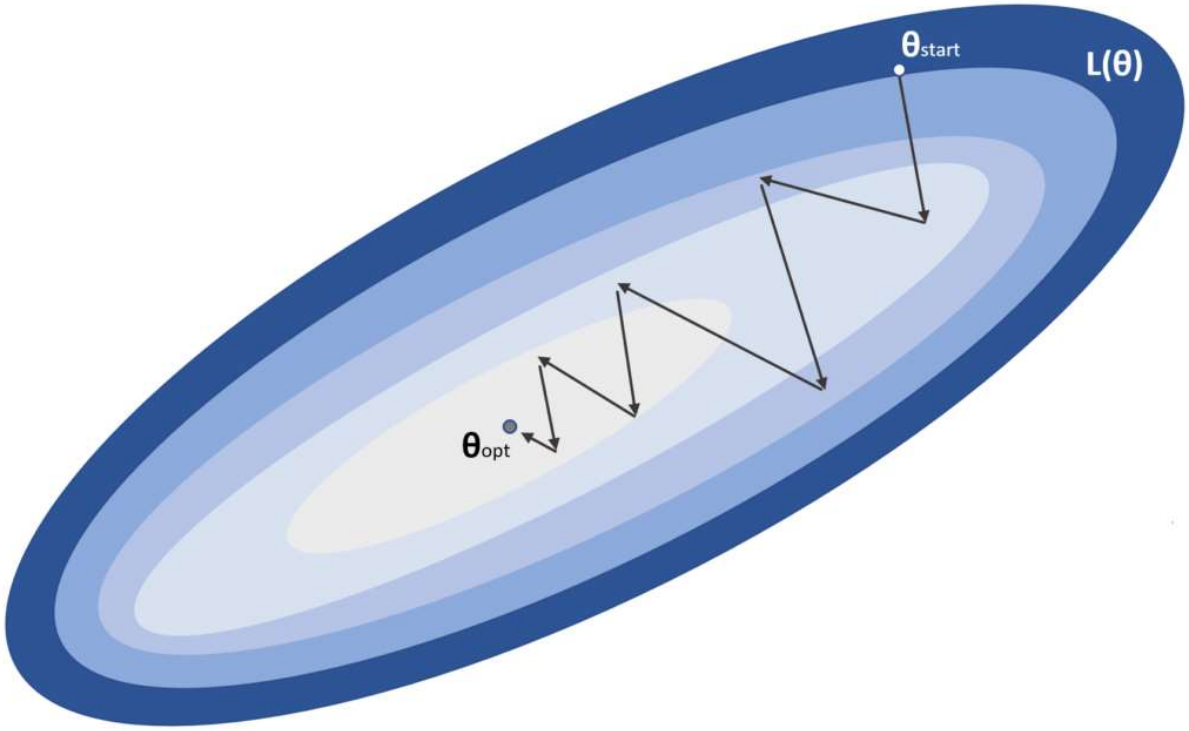
where x is the input sample, y is the label, and θ is the weight. We can define the partial derivative cost function for a batch size equal to N as:

$$L(\overline{\theta_i}) = \frac{1}{N} \sum_{i=1}^{N} L\left(\overline{x_i}, \overline{y_i}; \overline{\theta_i}\right)$$

The vanilla SGD algorithm is based on a θ update rule that must move the weights in the opposite direction of the gradient of L.

$$\overline{\theta}^{(t+1)} = \overline{\theta} - \alpha \nabla_{\theta} L \overline{\theta}$$

This process is represented in the following figure:

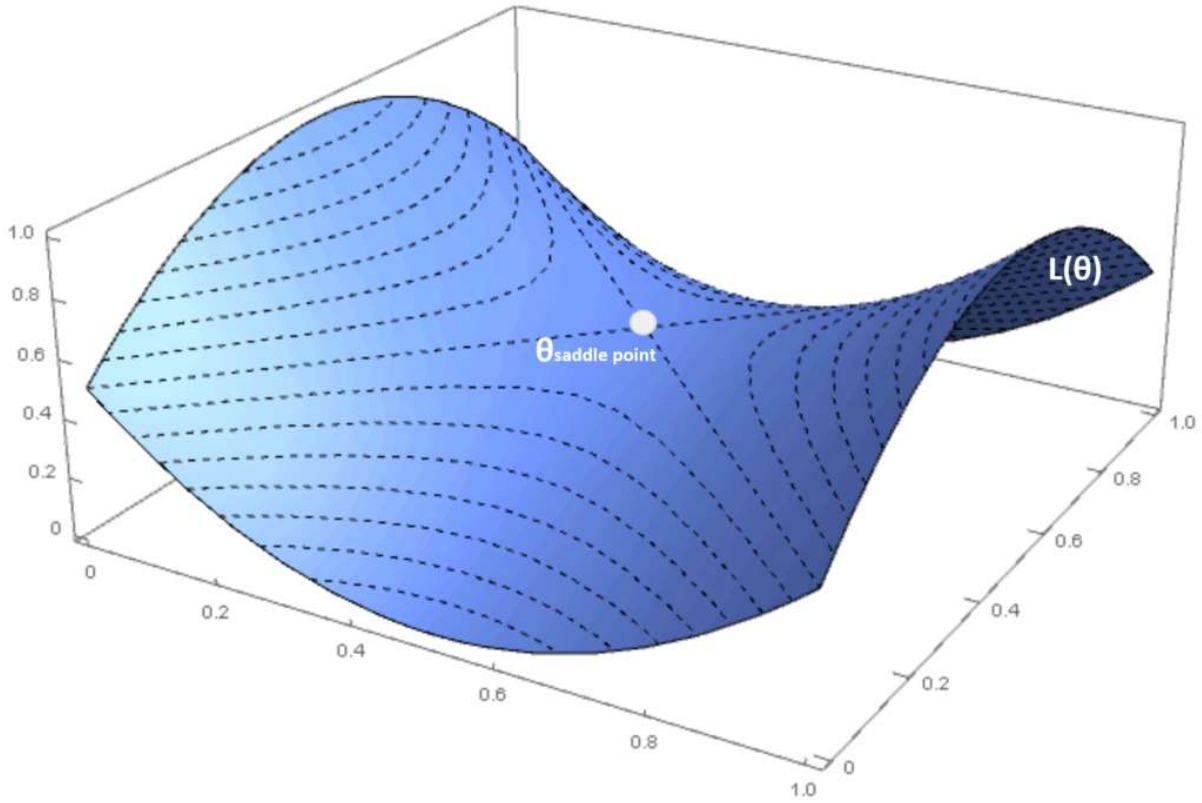


α is the learning rate, while θstart is the initial point and θopt is the global minimum we're looking for.

Iteration	w1	w2	Historic Loss	Learning Rate
1	0.5	4.0	1.0	0.1
2	0.6	3.9	0.8	0.1
3	0.7	3.8	0.6	0.1
4	0.8	3.7	0.4	0.1
5	0.9	3.6	0.2	0.1

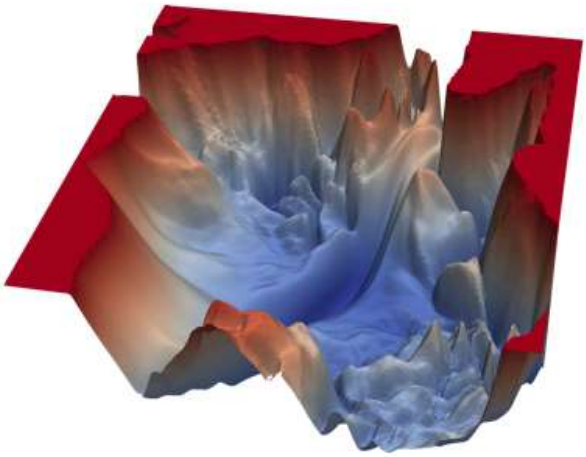
In a standard optimization problem, without particular requirements, the algorithm converges in a limited number of iterations.

Unfortunately, the reality is a little bit different, in particular in deep models, where the number of parameters is in the order of ten or one hundred million. When the system is relatively shallow, it's easier to find local minima where the training process can stop, while in deeper models, the probability of a local minimum becomes smaller and, instead, saddle points become more and more likely. Saddle point is a point at which a function of two variables has partial derivatives equal to zero but at which the function has neither a maximum nor a minimum value.

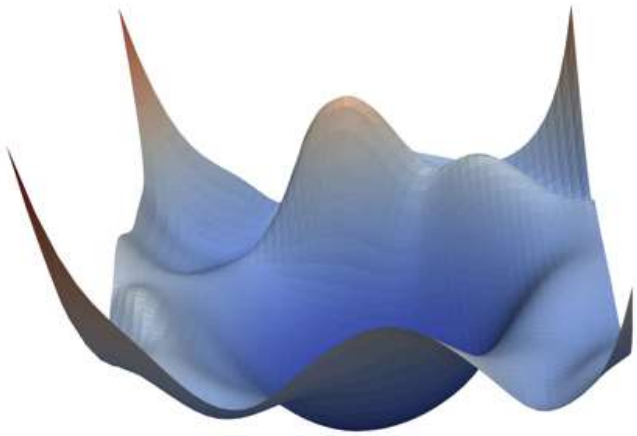


How often do you think they occur?

No residual connections

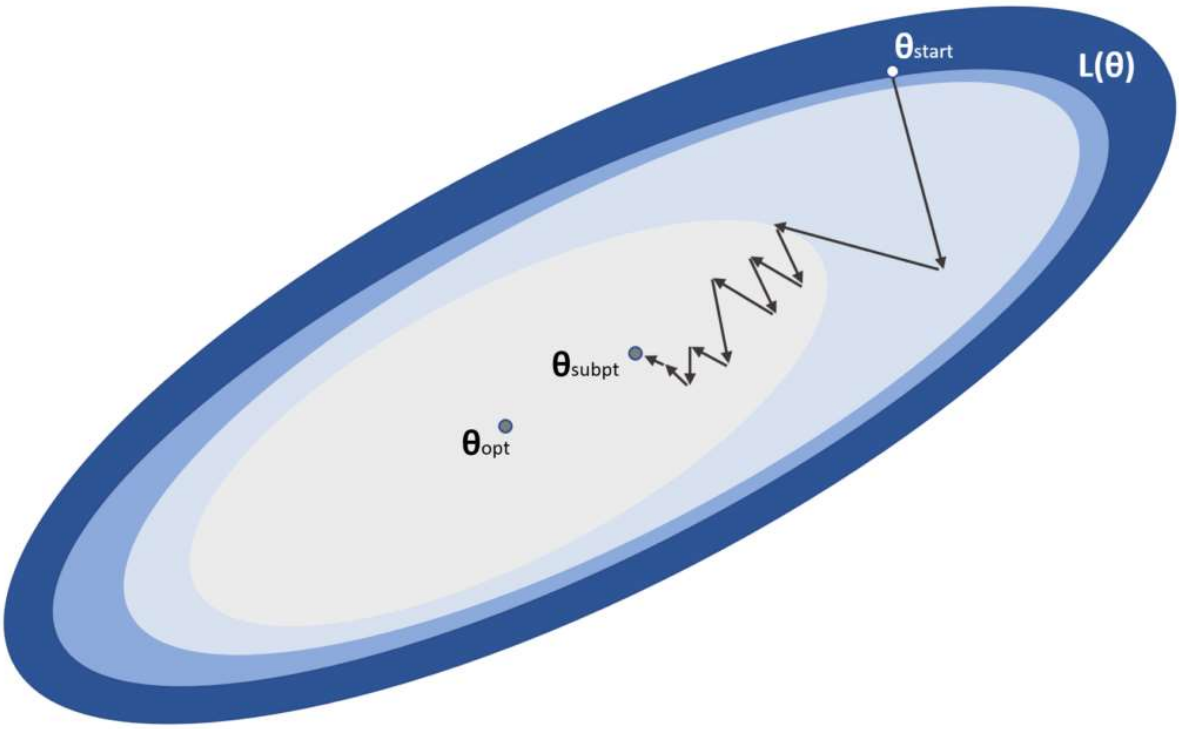


With residual connections



Same general network architecture

Then there is a problematic condition called **plateaus**, where L(θ) is almost flat in a very wide region.



How do we fix these problems?

Gradient Perturbation

A very simple approach to the problem of plateaus is adding a small noisy term (Gaussian noise) to the gradient:

$$\overline{\theta}^{(t+1)} = \overline{\theta}^{t} - \alpha\left(\nabla_{\theta} L\left(\overline{\theta}\right) + n\left(t\right)\right); n\left(t\right) \sim N\left(t; 0; \sigma^{2}\right)$$

The variance should be carefully chosen (for example, it could decay exponentially during the epochs). However, this method can be a very simple and effective solution to allow movement even in regions where the gradient is close to zero.

```python
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=1.):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean, self.std)

# then
transform = transforms.Compose([
    transforms.ToTensor(),
    ......,
    AddGaussianNoise(0., 1.), #good practise
])
```

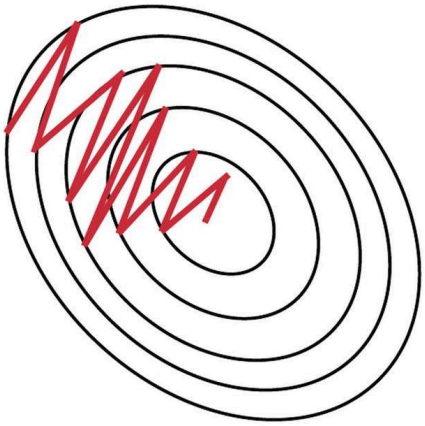
I would say we are indirectly doing this already. We have discussed doing this in two ways, what are those?

Momentum & Nesterov Momentum

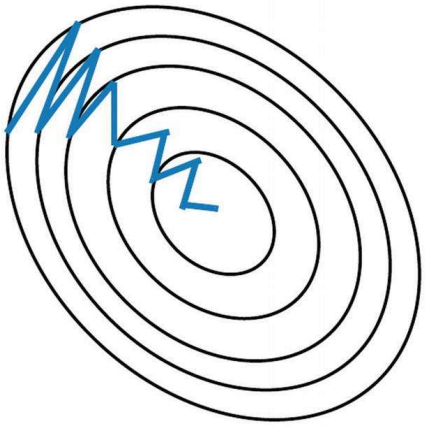
It is never a good idea to interfere with the network, and in the previous approach, we are doing exactly that.

A more robust solution is provided by introducing an **exponentially weighted moving (https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average)** average for the gradients.

The idea is very intuitive: instead of considering only the current gradient, we can attach part of its history to the correction factor, so as to avoid an abrupt change when the surface becomes flat.



Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum

The Momentum algorithm is, therefore:

$$\begin{cases} v^{(t+1)} = \mu v^{(t)} - \alpha \nabla_\theta L(\bar{\theta}) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

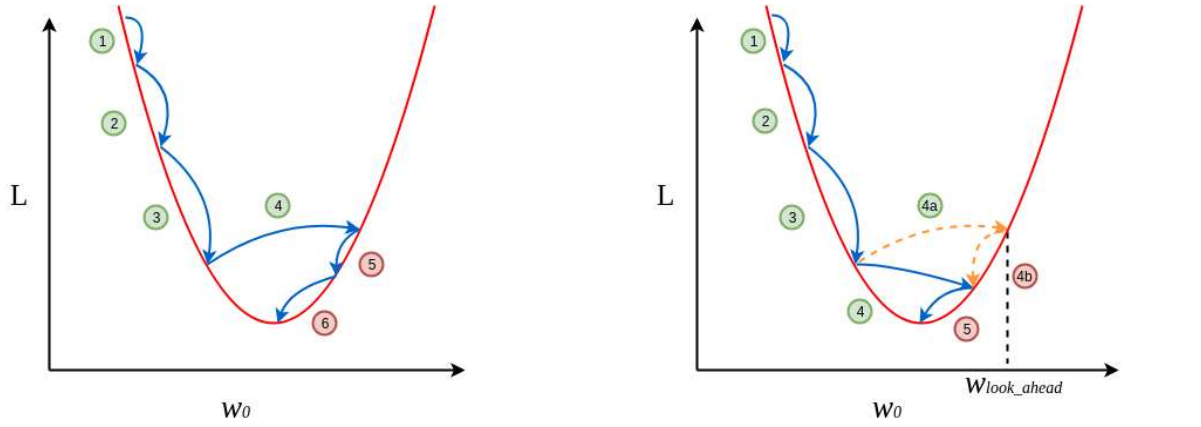
The first equation computes the correction factor considering the weight μ.

If μ is small, the previous gradients are soon discarded. If, instead, μ → 1, their effect continues for a longer time.

A common value in many applications is between 0.75 and 0.99, however, it's important to consider μ as a hyperparameter to adjust in every application.

The second term performs the parameter update. In the following figure, there's a vectorial representation of a Momentum step:

A slightly different variation is provided by the **Nesterov Momentum**.



(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent

$$\Rightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)} \qquad \Rightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$$

SOURCE (https://towardsdatascience.com/learning-parameters-part-2-a190bef2d12)

The difference with the base algorithm is that we first apply the correction with the current factor v(t) to determine

the gradient and then compute v(t+1) and correct the parameters:

$$\begin{cases} \bar{\theta}_N^{(t+1)} = \bar{\theta}^{(t)} + \mu v^{(t)} \\ v^{(t+1)} = \mu v^{(t)} - \alpha \nabla_\theta L\left(\bar{\theta}_N^{(t+1)}\right) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

Mathematically awesome, but in deep learning contexts, Nesterov doesn't seem to produce awesome results.

RMSProp

This algorithm, proposed by G. Hinton, is based on the idea to adapt the correction factor for each parameter, so as to increase the effect on slowly-changing parameters and reduce it when their change magnitude is very large. This approach can dramatically improve the performance of a deep network, but it's a little bit more expensive than Momentum because we need to compute a speed term for each parameter

RMSProp is a gradient-based optimization algorithm that adjusts the learning rate for each weight based on an exponential moving average of the historical squared gradients, and a scaling factor based on the square root of the moving average.

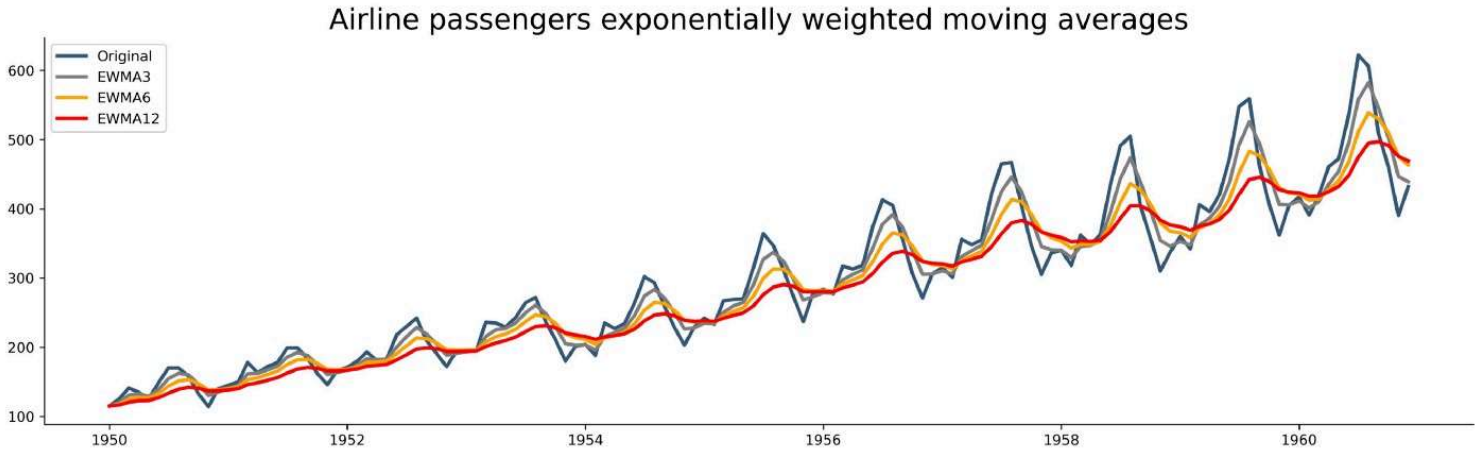
In simpler terms, RMSprop adjusts the learning rate dynamically for each parameter in the model, instead of using a fixed learning rate for all parameters. It does this by dividing the learning rate by an exponentially decaying average of the squared gradients for each

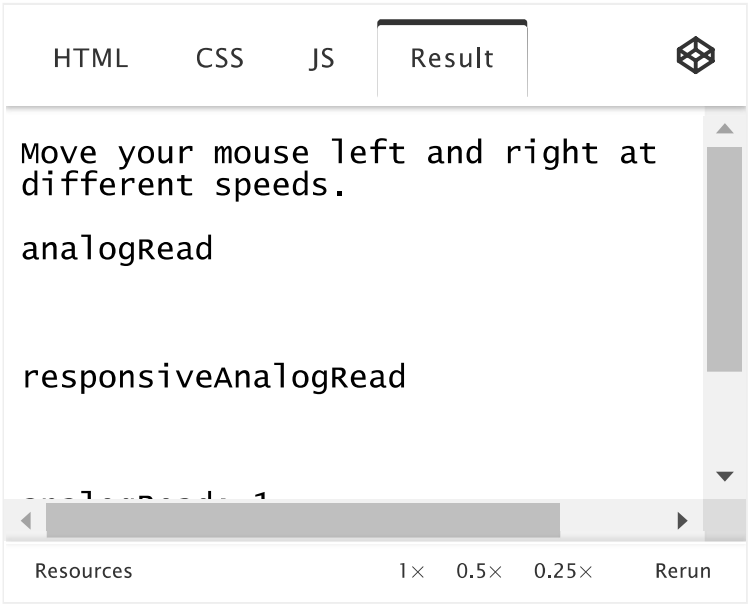
parameter. This helps to avoid oscillation or divergence in the optimization process and allows the model to converge faster and more accurately.

The main idea behind RMSprop is to shrink the learning rate for parameters that have a high historical gradient magnitude while keeping the learning rate high for parameters that have a low gradient magnitude. This allows the optimization process to make larger updates for parameters that have a low gradient magnitude and smaller updates for parameters with a high gradient magnitude, leading to faster convergence.

$$v^{(t+1)}(\theta_i) = \mu v^{(t)}(\theta_i) + (1-\mu)\left(\nabla_\theta L(\bar{\theta})\right)^2$$

This term computes the exponentially weighted moving average of the gradient squared (element-wise).





$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \frac{\alpha}{\sqrt{v^{(t+1)}(\bar{\theta}) + \delta}} \nabla_\theta L(\bar{\theta})$$

α is the learning rate and **δ** is a small constant (~ 1e-6 ÷ 1e-5) introduced to avoid a division by zero when the speed is null. As it's possible to see, each parameter is updated with a rule that is very similar to the *vanilla* Stochastic Gradient Descent, but the actual learning rate is adjusted per single parameter using the reciprocal of the square root of the relative speed. It's easy to understand that large gradients determine large speeds and, adaptively, the corresponding update is smaller and vice-versa. **RMSProp** is a very powerful and flexible algorithm and it is widely used in Deep Reinforcement Learning, CNN, and RNN-based projects.

Iteration	w1	w2	Historic Loss	Learning Rate w1	Learning Rate w2
1	0.5	4.0	1.0	0.1	0.1
2	0.6	3.5	0.8	0.08	0.12
3	0.7	3.0	0.6	0.05	0.18

Iteration	w1	w2	Historic Loss	Learning Rate w1	Learning Rate w2
4	0.8	2.5	0.4	0.03	0.24
5	0.9	2.0	0.2	0.01	0.32

Adam

Adam is an adaptive algorithm that could be considered an extension of **RMSProp**. Instead of considering the only exponentially weighted moving average of the gradient square, it computes also the same value for the gradient itself:

$$\begin{cases} g^{(t+1)}(\theta_i) = \mu_1 g^{(t)}(\theta_i) + (1-\mu_1)\nabla_\theta L(\bar{\theta}) \\ v^{(t+1)}(\theta_i) = \mu_2 v^{(t)}(\theta_i) + (1-\mu_2)\left(\nabla_\theta L(\bar{\theta})\right)^2 \end{cases}$$

μ1 and μ2 are forgetting factors like in the other algorithms. The authors suggest values greater than 0.9. As both terms are moving estimations of the first and the second moment, they can be biased (see **this article** (https://en.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation) for further information). Adam provided a bias correction for both terms:

$$\begin{cases} \hat{g}(\theta_i) = \dfrac{g^{(t+1)}(\theta_i)}{1-\mu_1{}^{(t)}} \\ \hat{v}(\theta_i) = \dfrac{v^{(t+1)}(\theta_i)}{1-\mu_2{}^{(t)}} \end{cases}$$

The parameter update rule becomes:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \frac{\alpha g^{(t+1)}(\bar{\theta})}{\sqrt{v^{(t+1)}(\bar{\theta}) + \delta}}$$

Iteration	w1	w2	Historic Loss	Learning Rate w1	Learning Rate w2
1	0.5	4.0	1.0	0.1	0.1
2	0.6	3.7	0.8	0.08	0.12
3	0.7	3.4	0.6	0.05	0.15
4	0.8	3.1	0.4	0.03	0.18
5	0.9	2.8	0.2	0.01	0.21

ADAM and RMSprop are both gradient-based optimization algorithms that dynamically adapt the learning rate for each weight during the optimization process. However, they differ in how they estimate the mean and variance of the gradients, and in how they use this information to adjust the learning rates.

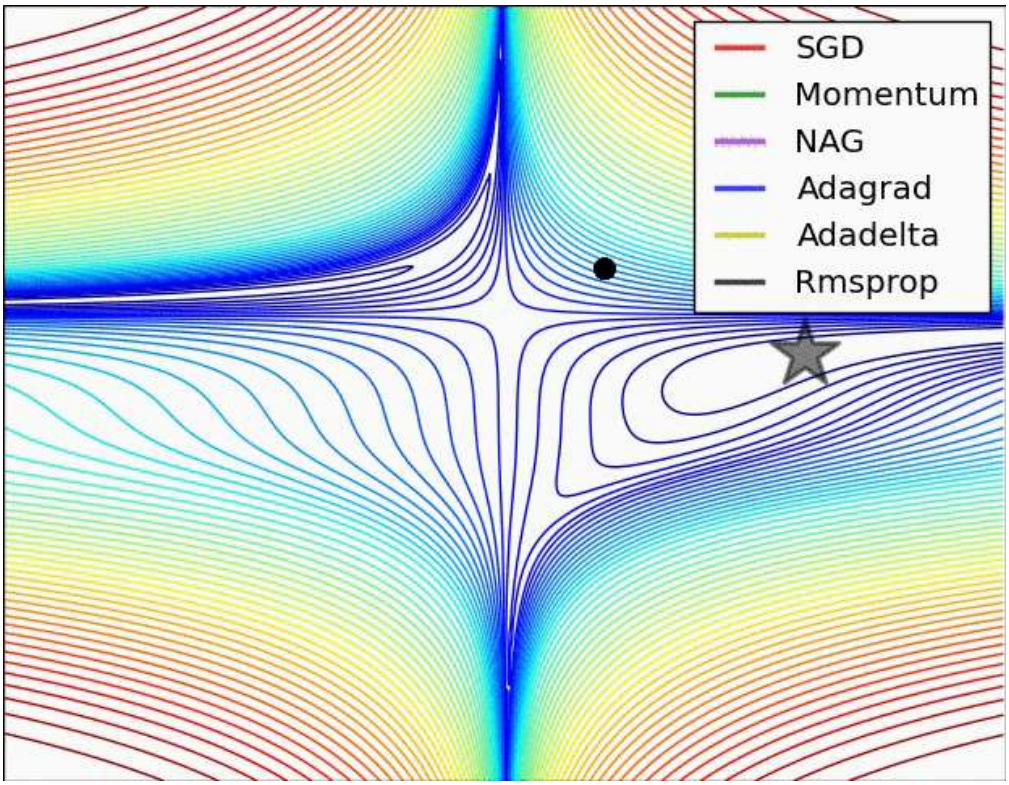
RMSprop estimates the mean and variance of the gradients by computing an exponential moving average of the squared gradients. The learning rate for each weight is then computed using a decay term that controls the decay rate of the historical moving average, and a scaling factor that is based on the square root of the moving average.

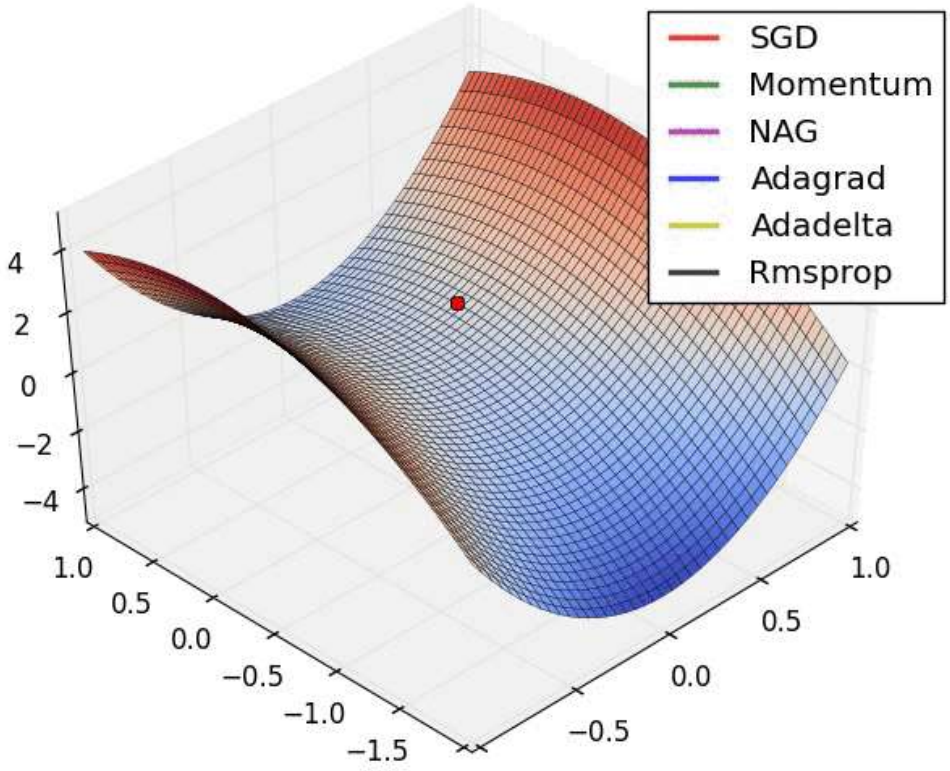
In contrast, ADAM uses a different technique to estimate the mean and variance of the gradients. It computes an exponential moving average of the gradients and another exponential moving average of the squared gradients. The learning rate for each weight

is then computed using these two moving averages, as well as two decay terms that control the decay rate of the moving averages.

Then we have AdaGrad and AdaDelta. Full **Source (https://www.bonaccorso.eu/2017/10/03/a-brief-and-comprehensive-guide-to-stochastic-gradient-descent-algorithms/)**

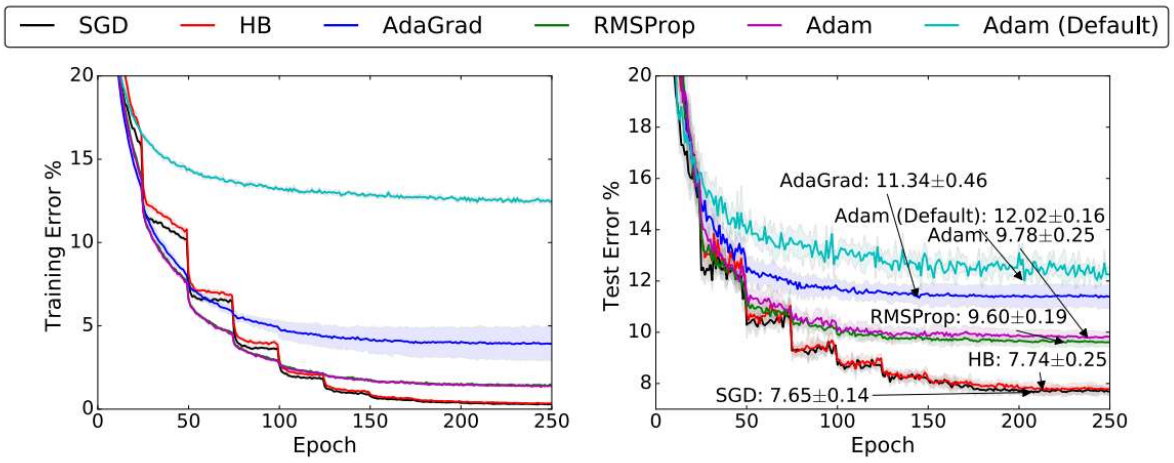
Best Optimizer



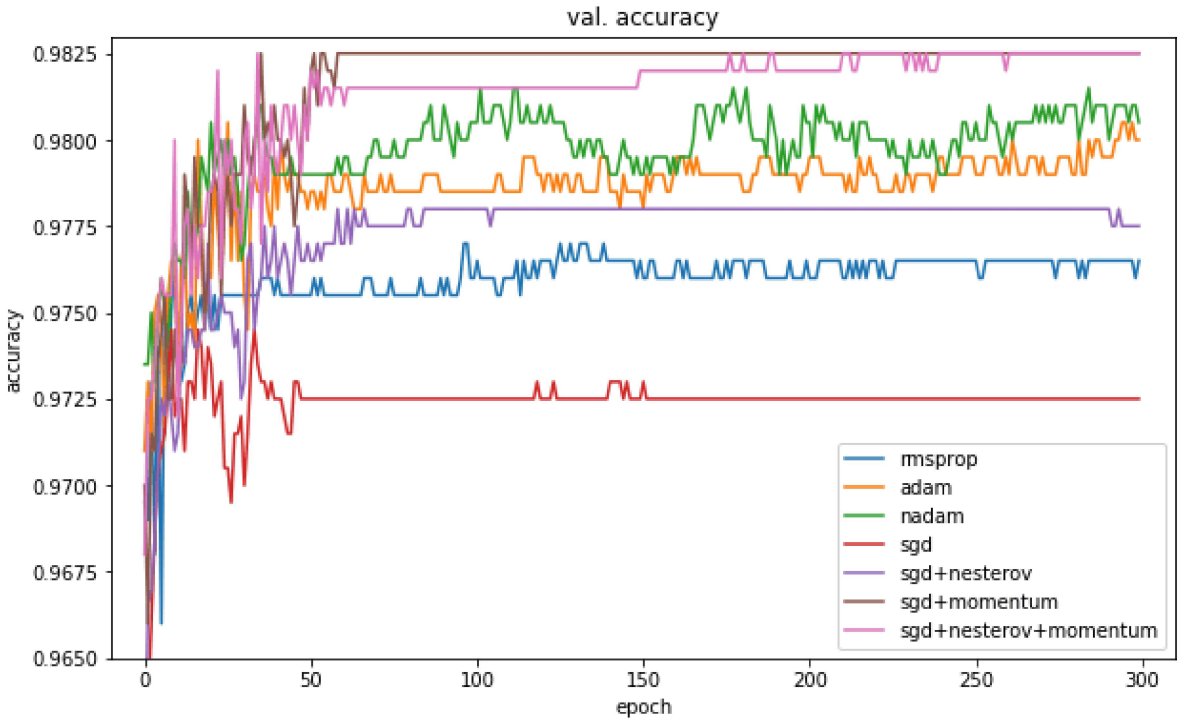


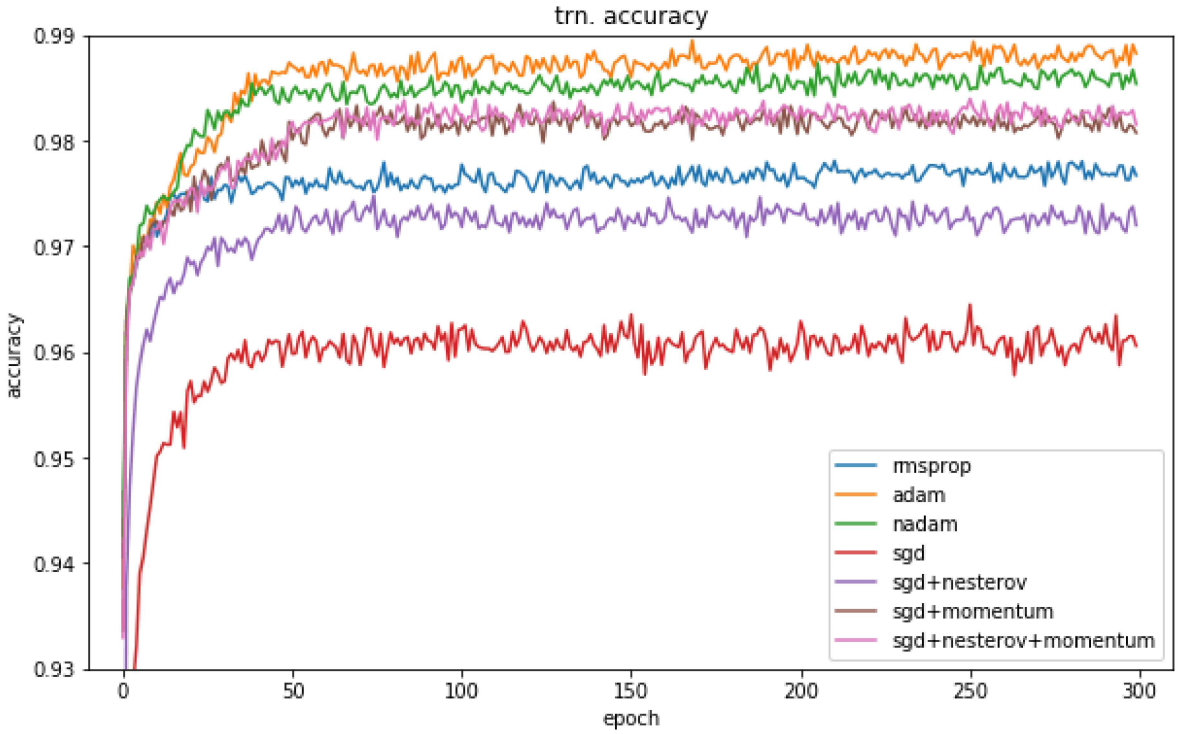
Please note that the above image does not compare Adam.

Here are some of the [execution (https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/)](https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/) results:



(a) CIFAR-10 (Train) **(b)** CIFAR-10 (Test)


val. accuracy
accuracy
epoch
0.9825
0.9800
0.9775
0.9750
0.9725
0.9700
0.9675
0.9650
0
50
100
150
200
250
300
rmsprop
adam
nadam
sgd
sgd+nesterov
sgd+momentum
sgd+nesterov+momentum


trn. accuracy
accuracy
epoch
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0
50
100
150
200
250
300
rmsprop
adam
nadam
sgd
sgd+nesterov
sgd+momentum
sgd+nesterov+momentum

Adam and others are better than SGD/SGD+.

Although adaptive optimizers have better training performance, it does not imply higher accuracy (better generalization)

Some common observations:

1. Adam and others generally have the lowest training error/loss, but not validation error/loss
2. It is common to use SGDs for SOTA performance.
    1. ResNet (2015) - SGD
    2. DenseNet (2016) - SGD
    3. ResNeXt (2016) - SGD
    4. SE-Net (2017) - SGD
    5. NasNet (2018) - SGD
    6. BERT (2018) - Adam
    7. EfficientNet (2019) - RMSprop
    8. GPT-3 (2020) - AdamW
    9. RegNet (2020) - AdamW
    10. ViT (2021) - AdamW
3. Adam and others are **preferred** (https://arxiv.org/pdf/1705.08292.pdf) for GANs and Q-learning with function approximations
4. SGD needs lesser memory since it only needs the first momentum
5. It has a much better regularization property compared to Adam (this can be fixed, see e.g **Fixing Weight Decay Regularization in Adam** (https://openreview.net/forum?id=rk6qdGgCZ))
6. If your input data is sparse you are likely to achieve the best results using one of the adaptive learning-date methods.

Completely different approach

Don't decay the learning rate, increase the batch size!

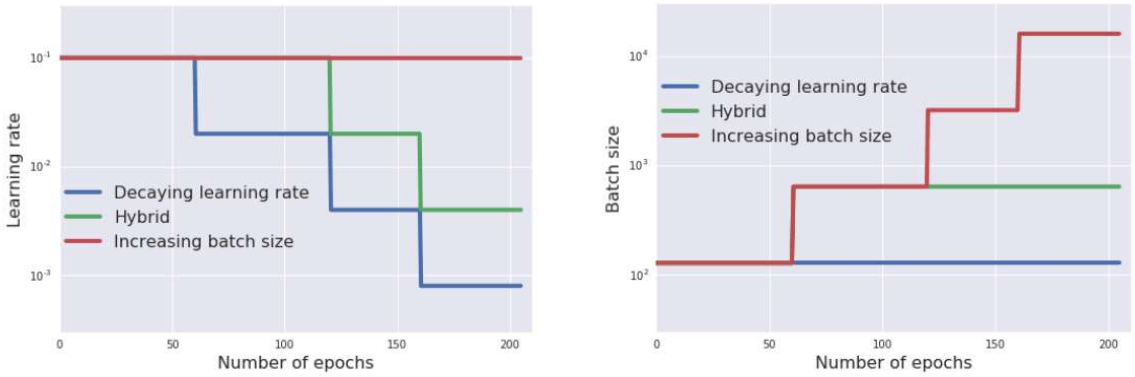
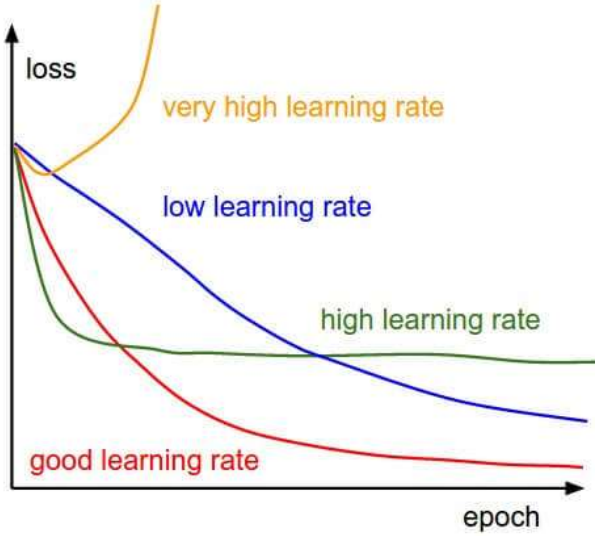


Figure 1: Schedules for the learning rate (a) and batch size (b), as a function of training epochs.

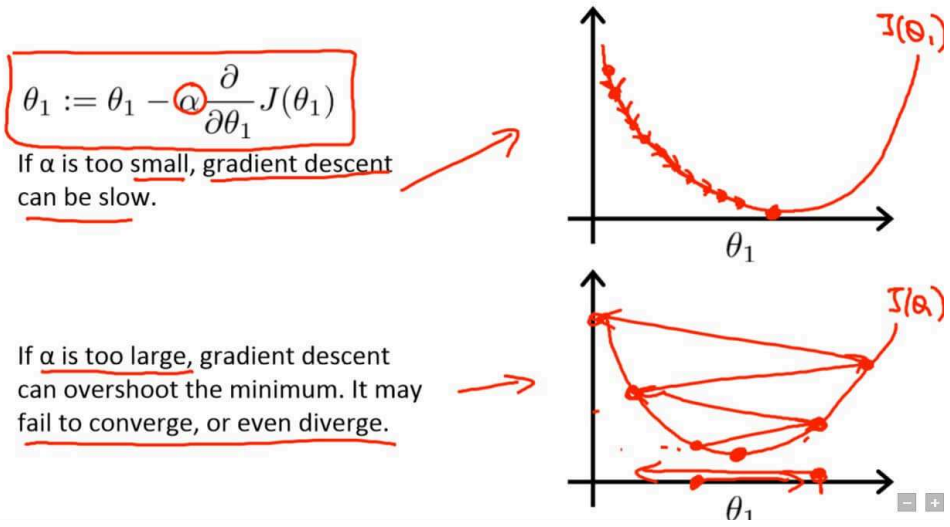
Batch size and learning rate are hyperparameters that affect the training of deep neural networks, but they are independent of each other and their relationship depends on the specific problem and architecture being used. A larger batch size generally requires a smaller learning rate, while a smaller batch size requires a larger learning rate. The optimal values for these hyperparameters can be found through experimentation and trial-and-error.

A larger batch size can provide more information per iteration and therefore a more stable gradients estimate, allowing the model to converge using a smaller learning rate. A smaller learning rate would help prevent overshooting of the optima, as a larger batch size provides a more robust update step, so the learning rate can afford to be smaller. Conversely, a smaller batch size provides a less stable gradients estimate, and therefore requires a larger learning rate to prevent oscillation or divergence during training.

Learning Rates



For all the above methods, we still need to find the learning rate.



Source: Andrew Ng

Starting with a large learning rate at the beginning of training is important because the random weights are far from the optimal values. With a large learning rate, the weight updates will be large, allowing the model to quickly move toward the optimal weights. However, as the weights approach the optimal values, the model's improvement in accuracy slows down. At this point, a smaller learning rate is used to make smaller and

more fine-grained weight updates, allowing the model to converge to the optimal weights more accurately.

Using a large learning rate in the beginning of training also helps to escape from any poor local minima, while using a smaller learning rate later in training helps to converge to a global minimum with high accuracy.

The schedule for decreasing the learning rate during training is called a learning rate schedule, and there are several commonly used schedules such as step decay, time decay, and cyclical learning rate schedules. The exact schedule to be used depends on the specific problem and the model being used.

We can naively try different values or try a smarter way.

Leslie N. Smith describes a powerful technique to select a range of learning rates for a neural network in section 3.3 of the 2015 paper "**Cyclical Learning Rates for Training Neural Networks (https://arxiv.org/abs/1506.01186)** ".

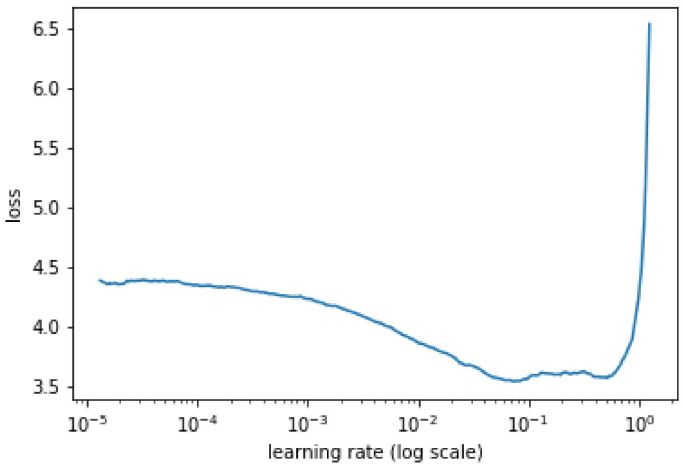
The trick is to train a network starting from a low learning rate and increase the learning rate exponentially for every batch.

Implementing the one-cycle policy involves the following steps:

1. Define the maximum learning rate: The maximum learning rate is the highest learning rate that will be used during training. It should be set to a value that is high enough to allow for significant weight updates, but not so high that the model becomes unstable. The process of finding the maximum learning rate usually involves the following steps:
    1. Choose a range of learning rates: Choose a range of learning rates that covers the range of possible learning rates, starting from a very small value (e.g., 0.01) and increasing to a large

value (e.g., 10).

2. Train the model with each learning rate: For each learning rate in the range, train the model for one or a few epochs and track the training loss after each batch.
3. Plot the learning rate vs training loss: Plot the learning rate on the x-axis and the training loss on the y-axis. This will show how the training loss changes with different learning rates.



4. Find the maximum learning rate: The maximum learning rate is the learning rate that gives the largest decrease in the training loss, or the steepest slope in the learning rate vs training loss plot.

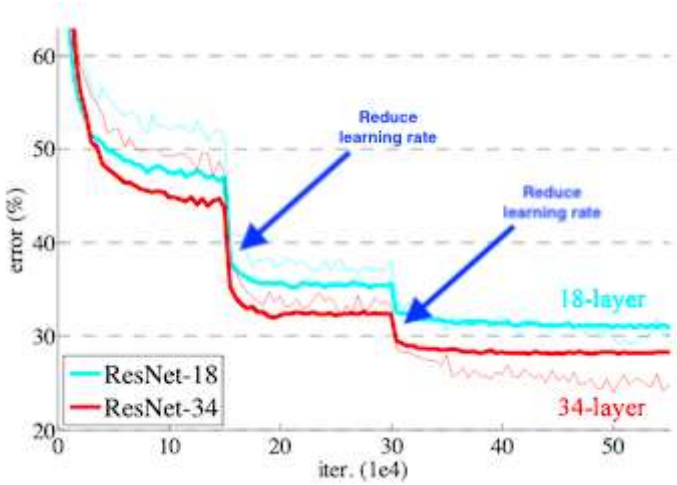
   First, with low learning rates, the loss improves slowly, then training accelerates until the learning rate becomes too large and loss goes up: the training process diverges.

   We need to select a point on the graph with the fastest decrease in the loss. In this example, the loss function decreases fast when the learning rate is between 0.001 and 0.01.

2. Define the learning rate schedule: The learning rate schedule is a function that maps the number of training iterations to a learning rate value. The schedule should start at a low learning rate, gradually increase to the maximum learning rate, and then gradually decrease back to the low learning rate
3. Implement the learning rate schedule: In the training loop, use the learning rate schedule to determine the learning rate for each iteration. The learning rate should be updated at each iteration, based on the current iteration number and the learning rate schedule
4. Train the model: Use the learning rate schedule to train the model, updating the weights at each iteration based on the current learning rate
5. Monitor the model performance: Monitor the model performance during training, including accuracy, loss, and any other relevant metrics. If the performance is not

satisfactory, adjust the maximum learning rate, the learning rate schedule, or the model architecture as necessary.

Reduce LR on Plateau




```
CLASS torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,
      patience=10, verbose=False, threshold=0.0001, threshold_mode='rel', cooldown=0,
      min_lr=0, eps=1e-08)
```

The "patience" and "threshold" parameters in the ReduceLROnPlateau algorithm control how the learning rate is adjusted based on the validation loss.

"Patience" refers to the number of epochs to wait before decreasing the learning rate if the validation loss has not improved. For example, if the patience value is set to 5, the learning rate will only be decreased if the validation loss has not improved for 5 epochs.

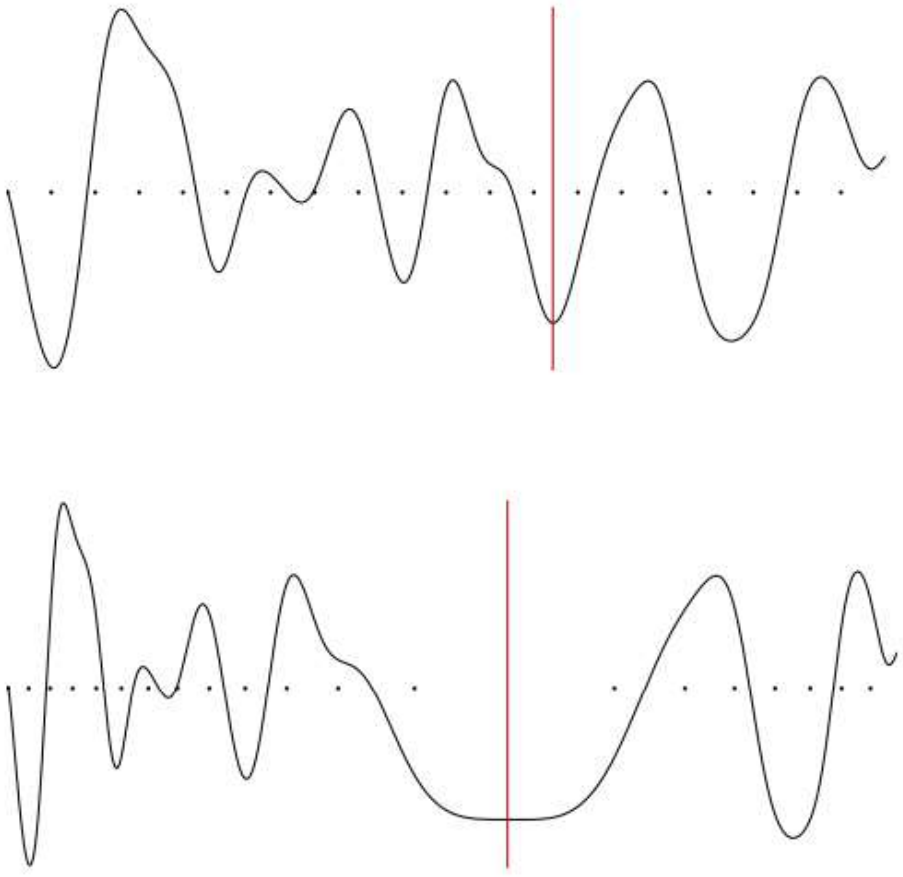
"Threshold" refers to the magnitude by which the validation loss should improve in order to consider it an improvement. For example, if the threshold value is set to 0.01, the validation loss should decrease by at least 0.01 to be considered an improvement.

These are not yet easy to pick by the way. For example, you really need to understand the difference between **rel** vs **abs** threshold_mode.

What kind of Minima do we want?

The rationale is that increasing the learning rate will force the model to jump to a different part of the
weight space if the current area is “spikey”.

Below is a picture of three same minima with different opening widths (or robustness). Which minima would you prefer?



Assignment

Assignment:

1. Check this Repo out: **https://github.com/kuangliu/pytorch-cifar (https://github.com/kuangliu/pytorch-cifar)**
2. (Optional) You are going to follow the same structure for your Code (as a reference). So Create:
    1. models folder - this is where you'll add all of your future models. Copy resnet.py into this folder, this file should only have ResNet 18/34 models. **Delete Bottleneck Class**
    2. main.py - from Google Colab, now onwards, this is the file that you'll import (along with the model). Your main file shall be able to take these params or you should be able to pull functions from it and then perform operations, like (including but not limited to):
        1. training and test loops

    2. data split between test and train
    3. epochs
    4. batch size
    5. which optimizer to run
    6. do we run a scheduler?
3. utils.py file (or a folder later on when it expands) - this is where you will add all of your utilities like:
    1. image transforms,
    2. gradcam,
    3. misclassification code,
    4. tensorboard related stuff
    5. advanced training policies, etc
    6. etc
3. Your assignment is to build the above training structure. Train ResNet18 on Cifar10 for 20 Epochs. The assignment must:
    1. pull your Github code to Google Colab (don't copy-paste code)
    2. prove that you are following the above structure
    3. that the code in your Google Colab notebook is NOTHING.. barely anything. There should not be any function or class that you can define in your Google Colab Notebook. Everything must be imported from all of your other files
    4. your colab file must:
        1. train resnet18 for 20 epochs on the CIFAR10 dataset
        2. show loss curves for test and train datasets
        3. show a gallery of 10 misclassified images
        4. show **gradcam (https://github.com/jacobgil/pytorch-grad-cam)** output on 10 misclassified images. **Remember if you are applying GradCAM on a channel that is less than 5px, then please don't bother to submit the assignment.** 🤬🤬🤬🤬🤬
    5. Once done, upload the code to GitHub, and share the code. This readme must link to the main repo so we can read your file structure.
    6. Train for 20 epochs
    7. Get 10 misclassified images
    8. Get 10 GradCam outputs on any **misclassified images (remember that you MUST use the library we discussed in class)**
    9. Apply these transforms while training:
        1. RandomCrop(32, padding=4)
        2. CutOut(16x16)

4. Assignment Submission Questions:

    1. Share the COMPLETE code of your model.py or the link for it
    2. Share the COMPLETE code of your utils.py or the link for it
    3. Share the COMPLETE code of your main.py or the link for it

4. Copy-paste the training log (cannot be ugly)
5. Copy-paste the 10/20 Misclassified Images Gallery
6. Copy-paste the 10/20 GradCam outputs Gallery
7. Share the link to your MAIN repo
8. Share the link to your README of Assignment (cannot be in the MAIN Repo, but Assignment 11 repo)

Videos

Studio Video

GM Video


ERA V2 Session 11 GM