DEEP LEARNING FOR VISUAL RECOGNITION

Lecture 6 – Training Convolutional Neural Networks (part 2)

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

Today's agenda

- You will learn more about training ConvNets.
- Topics
  - Stochastic Gradient Descent (SGD)
  - SGD extensions: Momentum, AdaGrad, RMSProp, and Adam
  - Learning rate decay and cycling
  - Regularization: Early stopping, weight decay, dropout, data augmentation
  - Hyperparameter search
  - Transfer learning

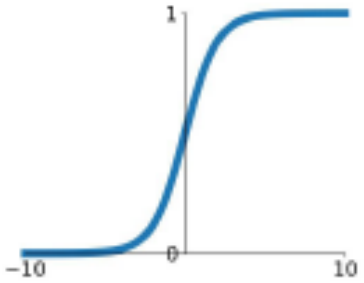
Training pipeline

- Set up network architecture = computational graph
- Train using mini-batch Stochastic Gradient Descent (SGD)
- Loop
  - **Sample** a batch of images
  - **Forward** prop it through the network (computational graph), and get loss
  - **Backprop** to calculate the gradient
  - **Update** parameters using the gradient

Where we are now

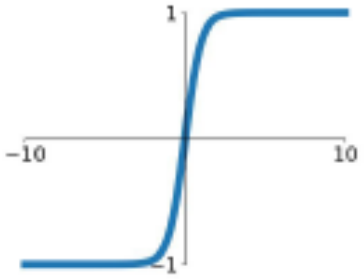
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



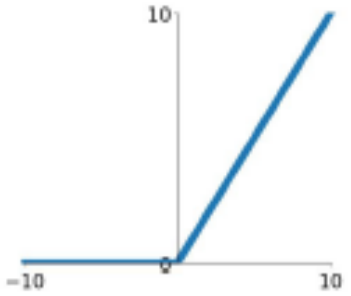
tanh

$$\tanh(x)$$



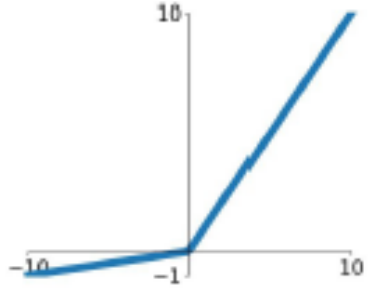
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

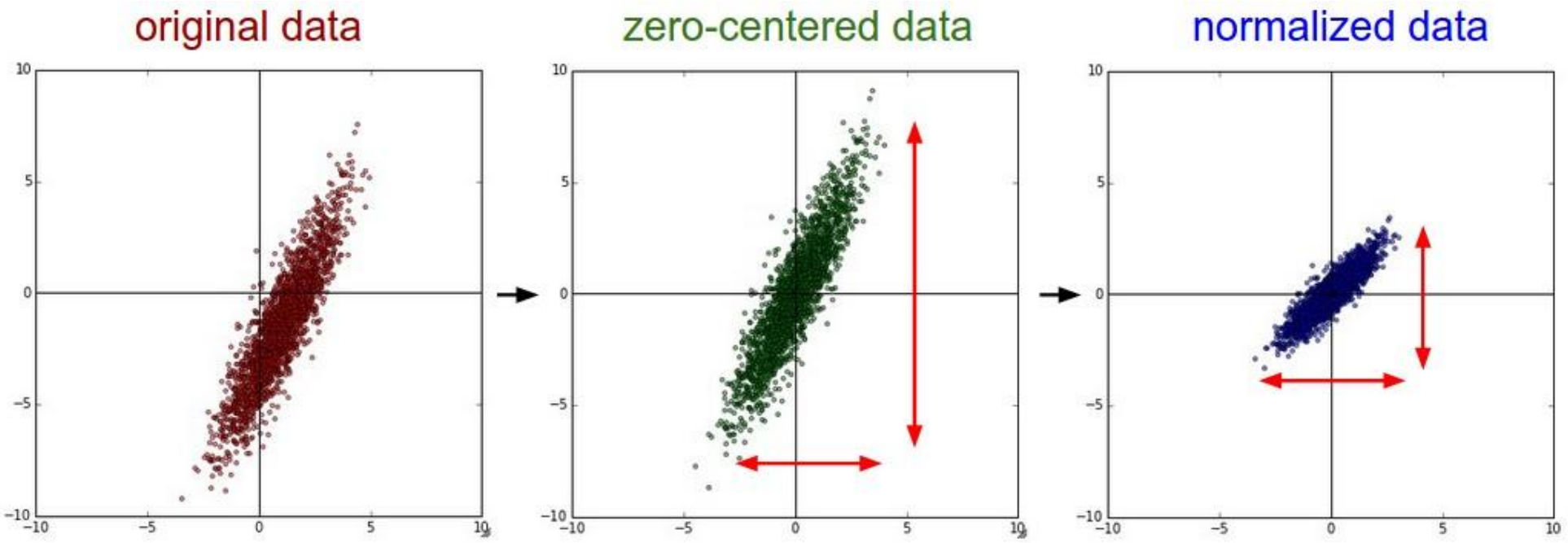


Activation functions

- Use **ReLU** but be careful with your learning rates.
- Try out **Leaky ReLU** (or more exotic alternatives not covered here, like ELU)
- Try out **tanh** but don't expect much.
- Don't use **sigmoid**, except maybe at the output layer, if you want values that reflect probabilities.

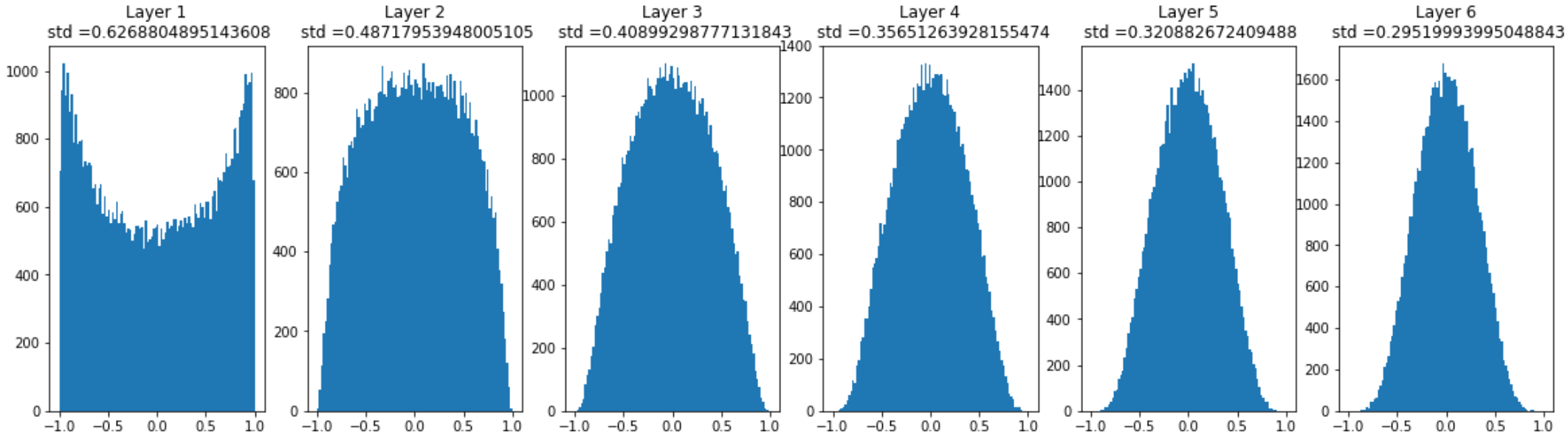
Where we are now

- Always zero-center your data (makes optimization faster)
- Optionally normalize to control variance
- When using a pretrained network, apply the same preprocessing as was used during training



Where we are now

- Improper **weight initialization** can cause vanishing gradients (slow learning or no learning at all) or exploding gradients.
- Use the correct weight initialization strategy (depends on activation function)

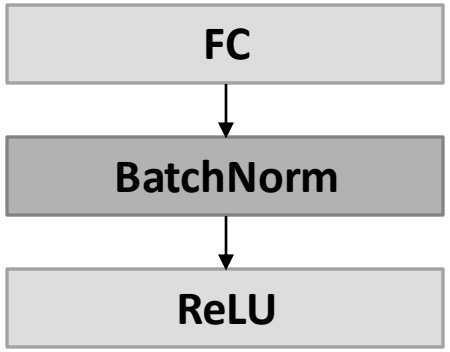


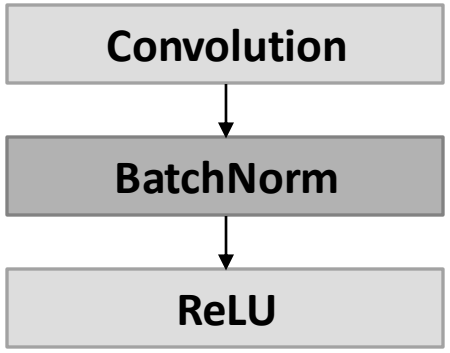
Where we are now

- Always use **batch normalization**
- Basic idea

$$z = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}(x)}} = \frac{x - \mu}{\sigma}$$

- Insert BatchNorm layers before non-linearities
- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!





Stochastic Gradient Descent (SGD) and momentum

Recall: Gradient descent

- Gradient descent is a way to **minimize a loss function** $J(w)$ of a model $h_w(x)$, parameterized by a set of parameters w.
- It works by updating the parameters in the opposite direction of the gradient $\nabla J(w)$ of the loss function w.r.t. to the parameters w:

$$w = w - \alpha \nabla J(w)$$

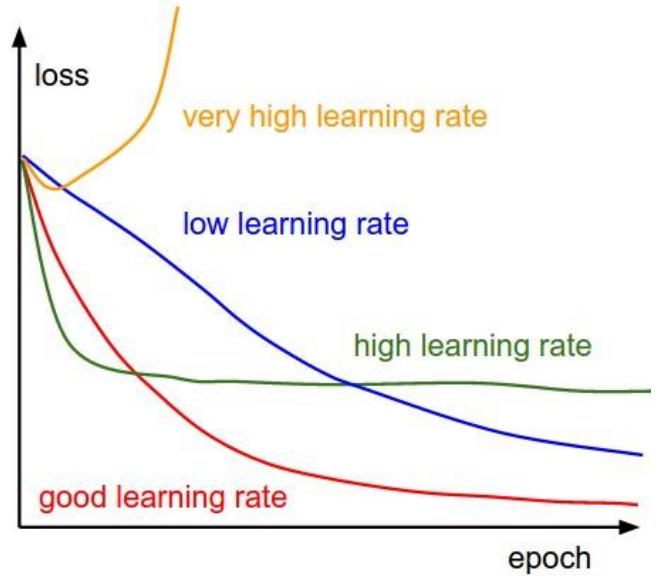
- The gradient is sometimes written $\nabla_w J(w)$.
- The learning rate α determines the size of the steps we take to reach a (local) minimum.
- In other words, we follow the direction of the slope of the surface created by the loss function downhill until we reach a valley.

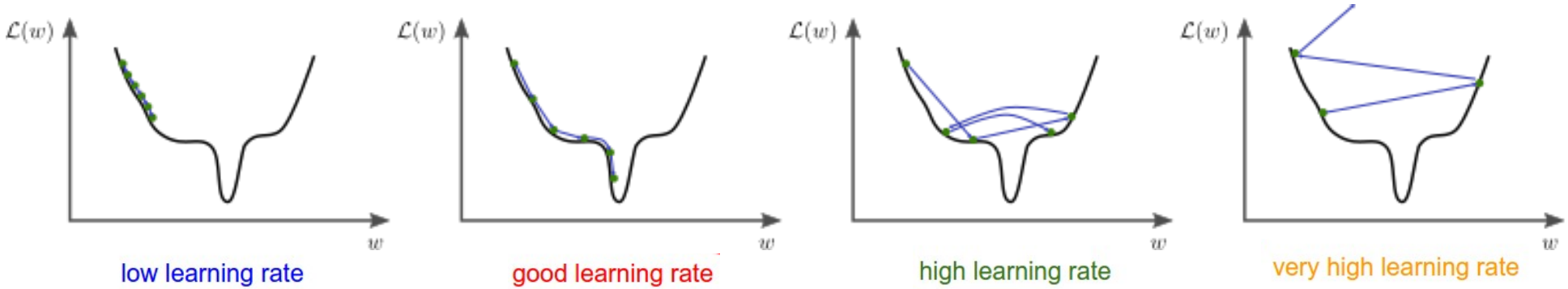
Gradient descent variants

- There are basically two variants of gradient descent.
- **Batch gradient descent:** Vanilla gradient descent, aka batch gradient descent, computes the gradient of the loss function w.r.t. to the parameters for the **entire training dataset**.
- **Stochastic gradient descent (SGD):** Mini-batch gradient descent, aka stochastic gradient descent, estimates the gradient using a small, random set of *n* samples, called a **mini-batch**.
- Notes on SGD:
  - *n* is commonly referred to as the **batch size**
  - Increasing *n* reduces the variance of the parameter updates, which can lead to more stable convergence.
  - On the other hand, decreasing *n* creates random fluctuations the help SGD avoid local minima.
  - Deep learning libraries can make use of highly optimized matrix optimizations (on the GPU) that make computing the gradient w.r.t. a mini-batch very efficient. It is a common practise to use batch sizes of powers of 2 to offer better run-time with GPUs.
  - Some define SGD as $n = 1$ and mini-batch gradient as $n > 1$. We will not distinguish between the two.

Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.





Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.
- **SGD has trouble navigating ravines**, i.e., areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.

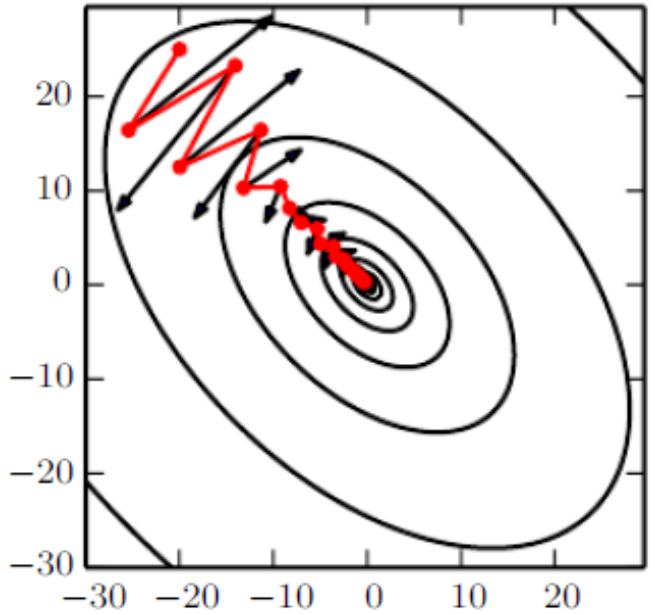
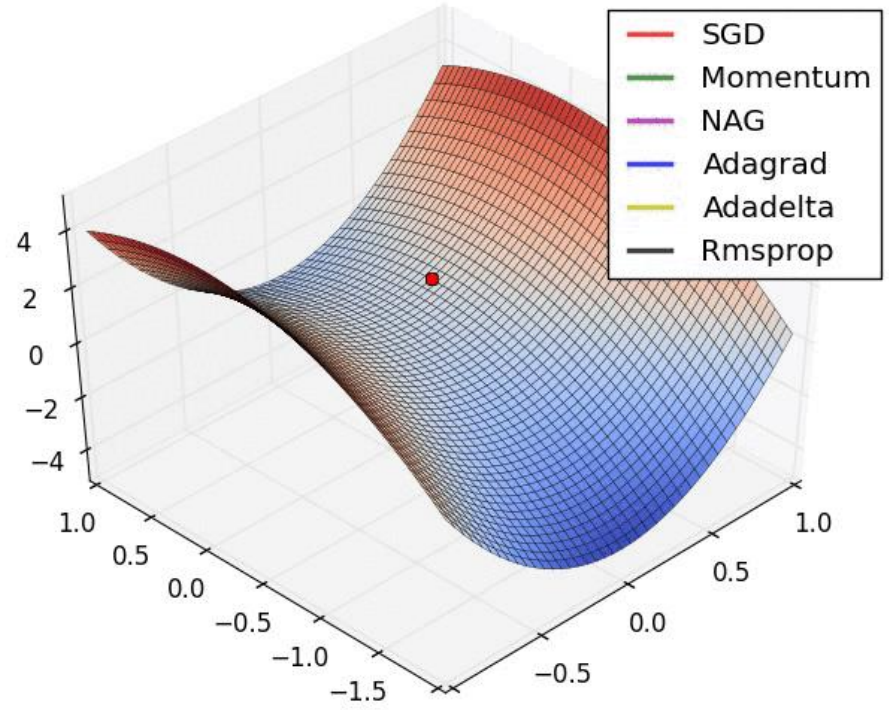


Figure 8.5 (book)

Challenges with standard SGD

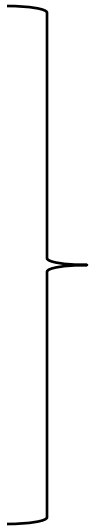
- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.
- **SGD has trouble navigating ravines**, i.e., areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is **avoiding getting trapped** in their numerous suboptimal local minima and/or saddle points (i.e., points where one dimension slopes up and another one slopes down, and there is zero gradient).



Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

Challenges with standard SGD

- Choosing a proper learning rate can be difficult.
- **Learning rate schedulers** try to adjust the learning rate during training but have to be defined in advance and are unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameters, ignoring the possibility that some features are more important than others.
- **SGD has trouble navigating ravines**, i.e., areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is **avoiding getting trapped** in their numerous suboptimal local minima and/or saddle points (i.e., points where one dimension slopes up and another one slopes down, and there is zero gradient).



These problems become more likely in higher dimensions.

Momentum

- SGD has trouble navigating ravines, i.e., areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as shown in Image 1 below:

Image 1: SGD without momentum

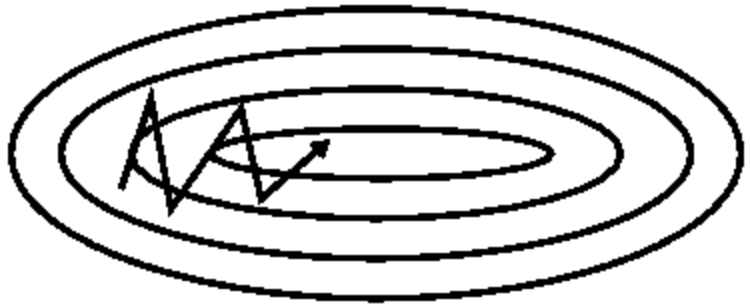


Image 2: SGD with momentum

- Momentum is a method that **helps accelerate SGD in the relevant direction and dampens oscillations** as can be seen in Image 2.

http://ruder.io/optimizing-gradient-descent/

Momentum – update equation

- Momentum dampens oscillations by adding a fraction of the update vector of the past time step to the current update vector:

$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w)$$
$$w = w - \alpha v_t$$

The momentum term γ is usually set to 0.9 or a similar value.

Image 1: SGD without momentum

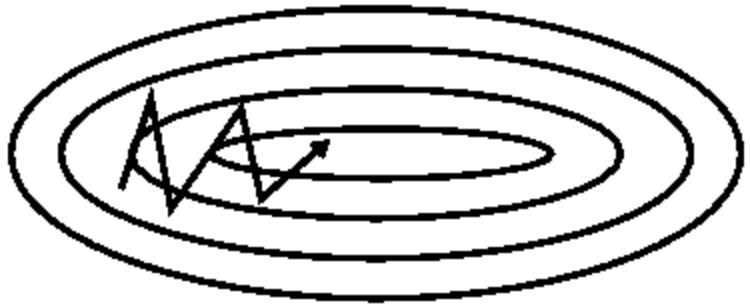


Image 2: SGD with momentum

- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

http://ruder.io/optimizing-gradient-descent/

Momentum – intuition

- You can think of momentum as **pushing a ball down a hill**. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Image 1: SGD without momentum

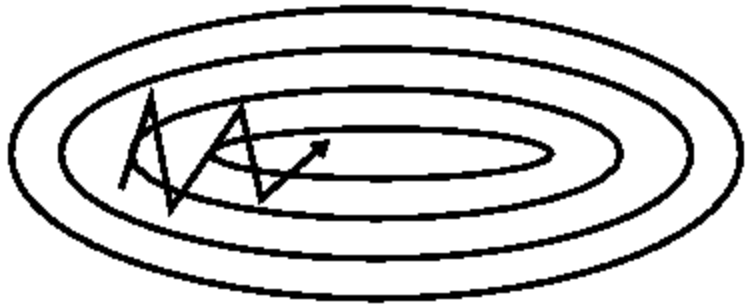
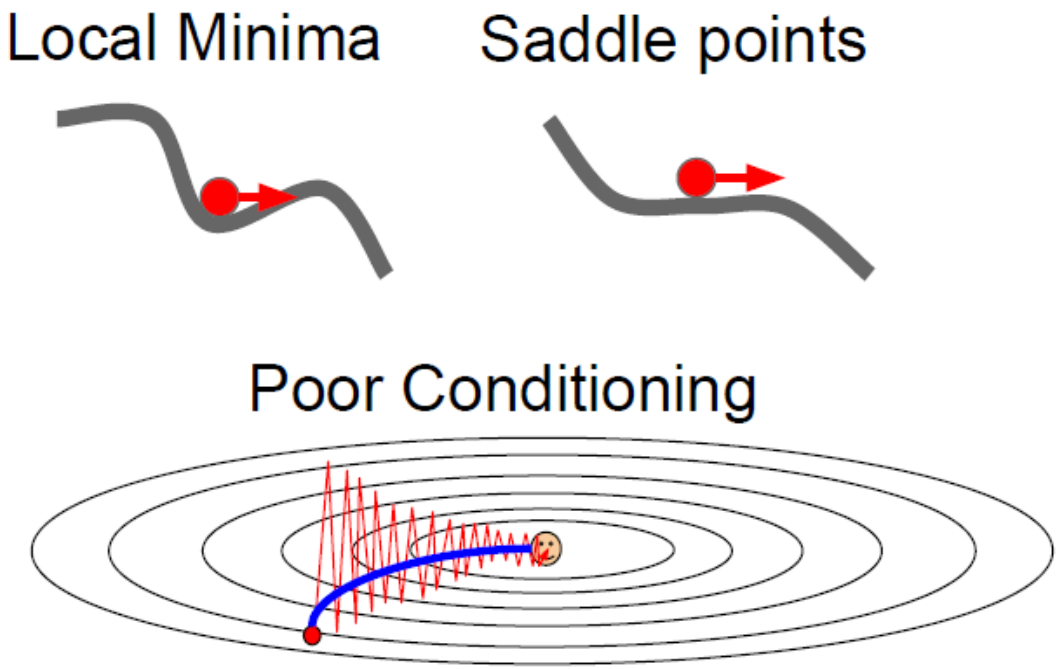


Image 2: SGD with momentum

http://ruder.io/optimizing-gradient-descent/

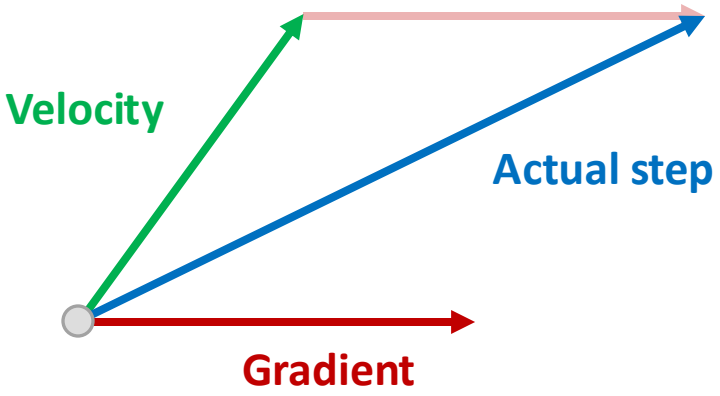
Momentum

- The velocity/momentum that we have built up also helps us escape local minima and saddle points (think of rolling ball).
- Also, the zig-zagging associated with SGD (formally explained by poor conditioning of the Hessian matrix) is dampened.
- The explanation of the latter is that the oscillations in the sensitive direction cancel each other out over time, because the momentum algorithm accumulates an exponentially decaying moving average of past gradients.



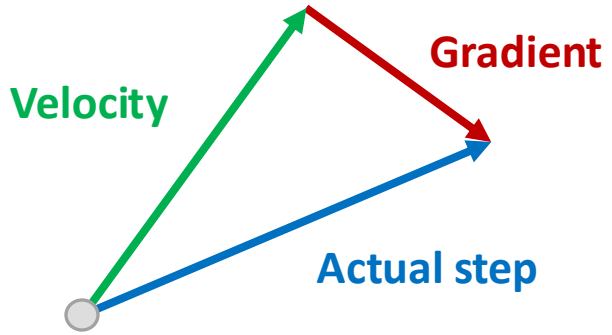
Nestorov momentum

Momentum update:



$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w)$$
$$w = w - \alpha v_t$$

Nestorov momentum:
(look ahead)

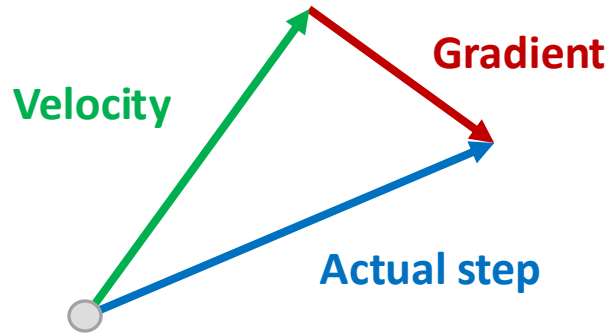


$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w + \gamma v_{t-1})$$
$$w = w - \alpha v_t$$

Nestorov momentum

Intuition: A ball that rolls down a hill, blindly following the slope, is unsatisfactory.
We'd like to have **a smarter ball**, a ball that has a notion of where it is going so that it **knows to slow down before the hill slopes up again**. Nestorov momentum does this by looking ahead in time. In practise, it usually works slightly better than conventional momentum.

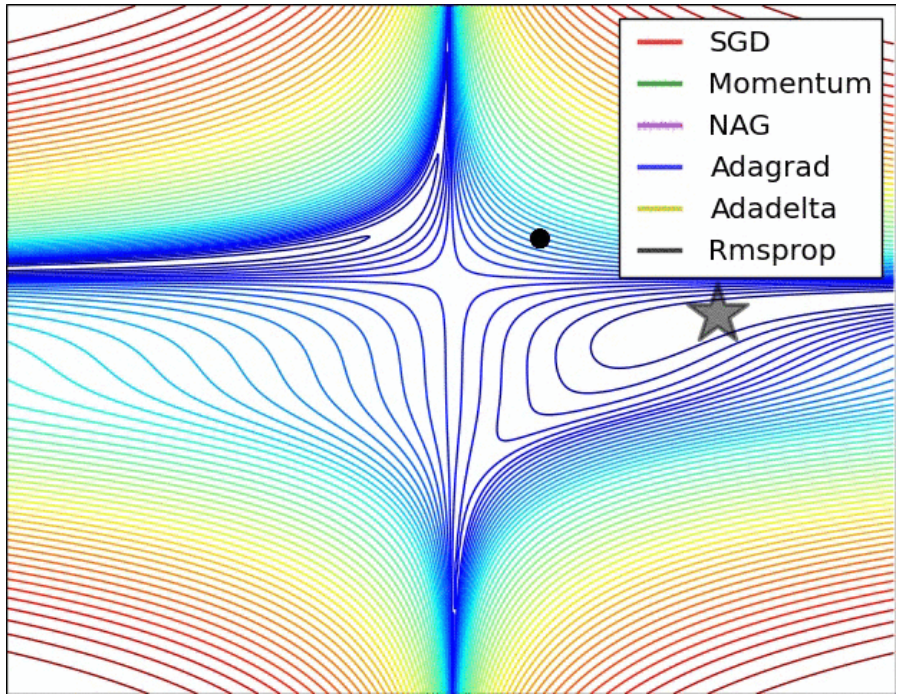
Nestorov momentum:
(look ahead)



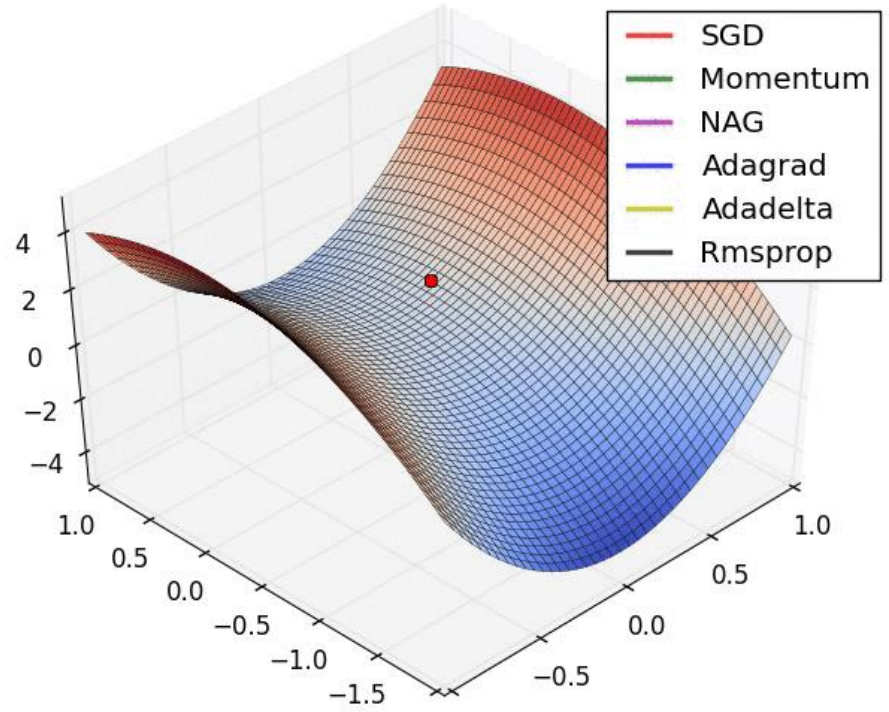
$$v_0 = 0$$
$$v_t = \gamma v_{t-1} + \nabla J(w + \gamma v_{t-1})$$
$$w = w - \alpha v_t$$

Nestorov momentum

Look for NAG: Nestorov Accelerated Gradient



Link to GIF animation:
http://ruder.io/content/images/2016/09/contours_evaluation_optimizers.gif



Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

In practise

- Always use momentum with SGD.
- Setting the momentum term (γ) between 0.9 and 0.99 usually works.
- Start with high learning rate and decay over time.

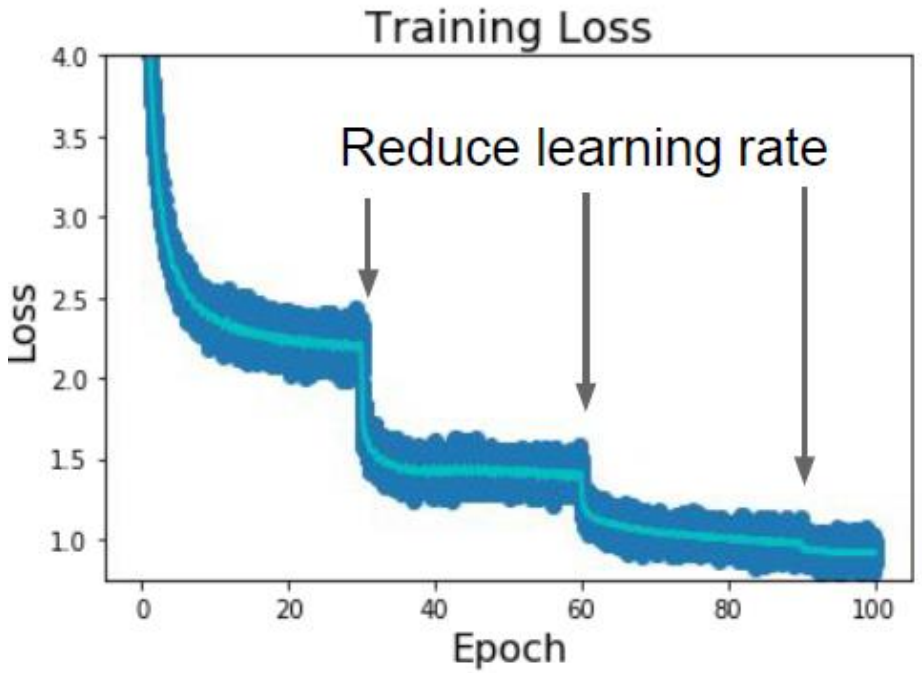
Learning rate decay and cycling

Learning rate decay

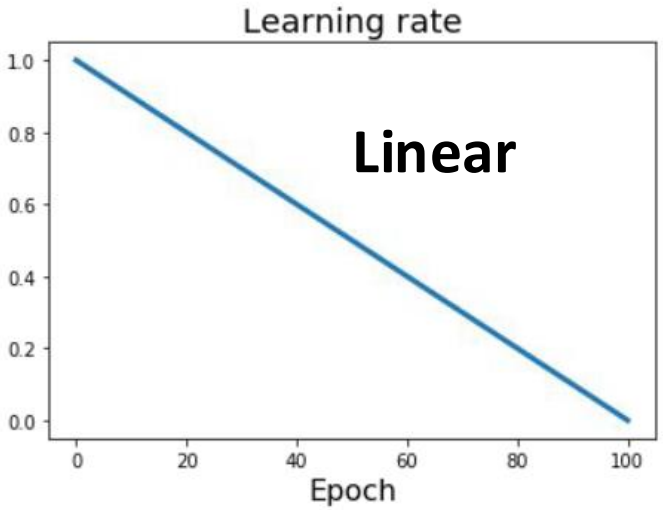
- In training deep networks, it is usually helpful to decay the learning rate over time.
- **Intuition**
  1. Start with high learning rate to **explore loss landscape**.
  2. Then slowly decay learning rate to **tune in on a local minimum**.
- Knowing when and how much to decay the learning rate **can be tricky**:
  - Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time.
  - But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can.

Step decay

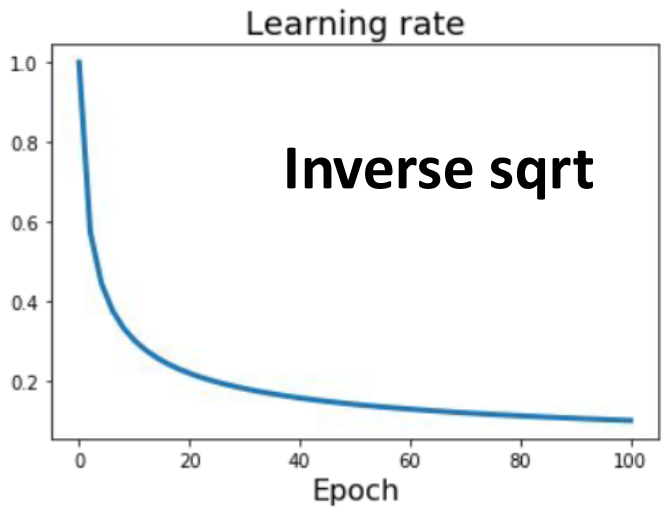
- Reduce the learning rate by some factor every few epochs.
- Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs.
- These numbers depend heavily on the type of problem and the model.
- One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate and reduce the learning rate by a constant (e.g., 0.5) whenever the validation error stops improving.



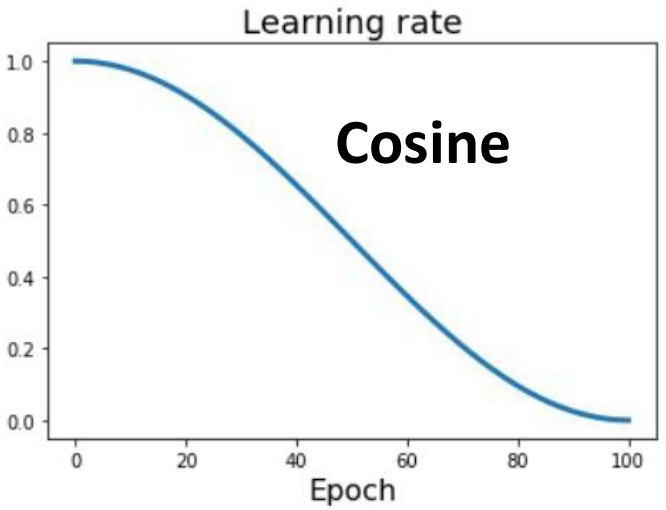
Other variants of learning rate decay



$\alpha = \alpha_0(1 - t/T)$



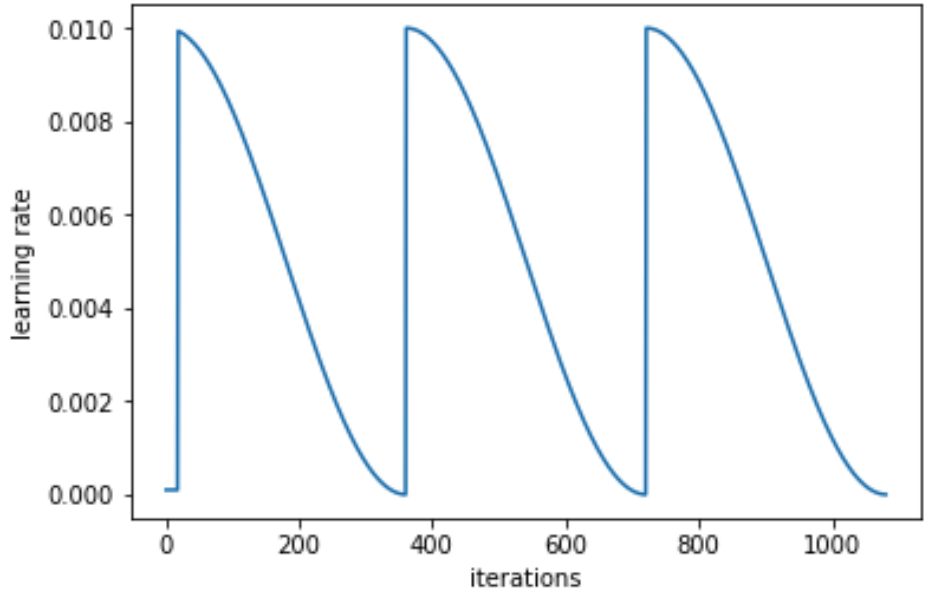
$\alpha = \alpha_0/\sqrt{t}$



$\alpha = \frac{1}{2}\alpha_0(1 + \cos(t\pi/T))$

Cyclical Learning Rate (CLR)

- The learning rate is the most important hyperparameter to tune for training deep neural networks.
- CLR is a method for setting the learning rate, which practically eliminates the need to experimentally find the best values and schedule for the global learning rates.
- Instead of monotonically decreasing the learning rate, **this method lets the learning rate cyclically vary between reasonable boundary values**.
- Training with cyclical learning rates instead of fixed values **achieves improved classification accuracy** without a need to tune and often in fewer iterations.

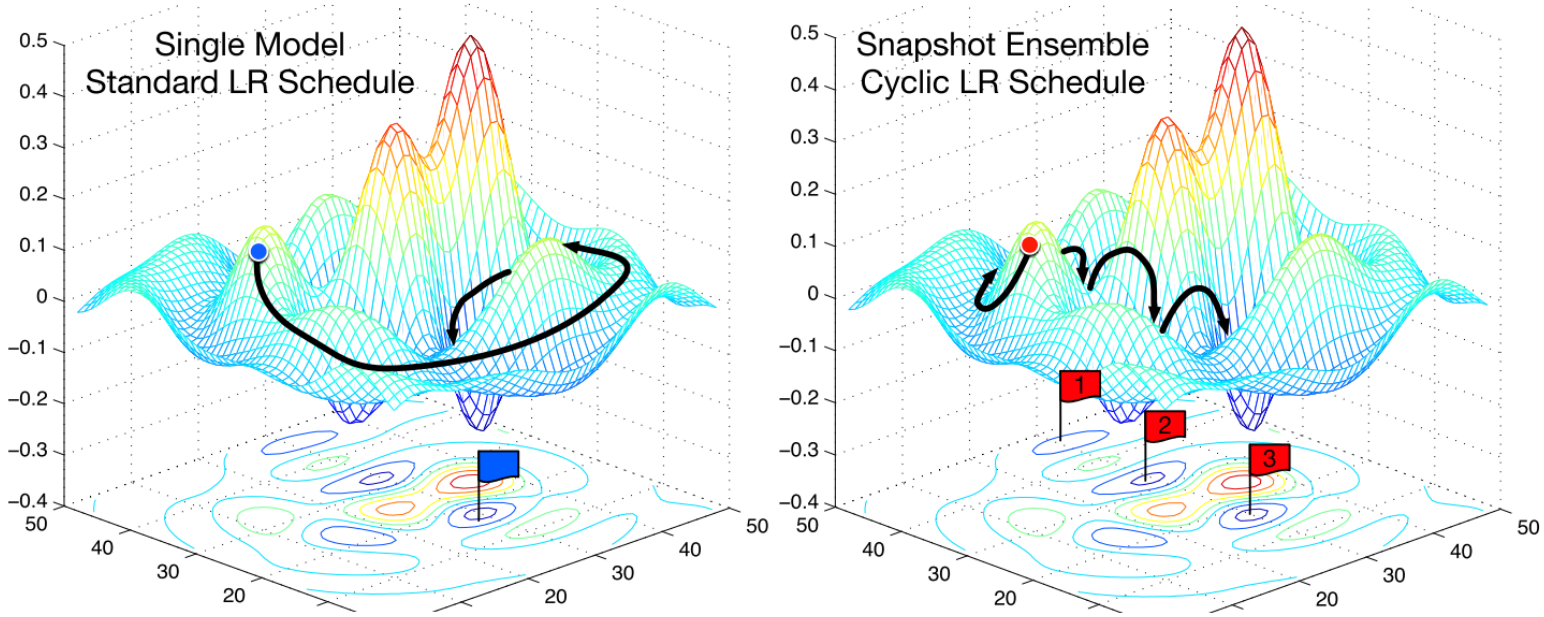


Smith (2015): Cyclical Learning Rates for Training Neural Networks

https://medium.com/@lipeng2/cyclical-learning-rates-for-training-neural-networks-4de755927d46

Cyclical Learning Rate (CLR)

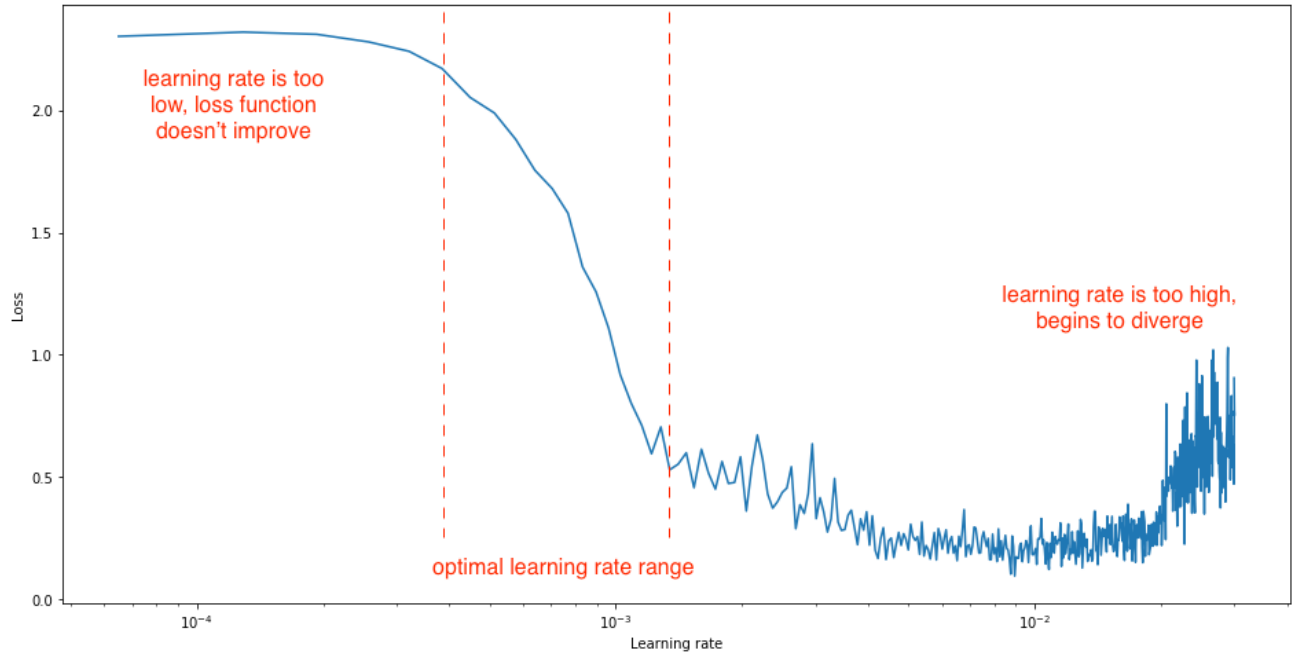
- The rationale is that increasing the learning rate **will force the model to jump to a different part of the weight space if the current area is "spikey".**
- In other words, it will force to find another local minimum if the current minimum is not robust, and make the model generalize better to unseen data. Below is an illustration of CLR schedule with three resets.



https://medium.com/@lipeng2/cyclical-learning-rates-for-training-neural-networks-4de755927d46

Cyclical Learning Rate (CLR)

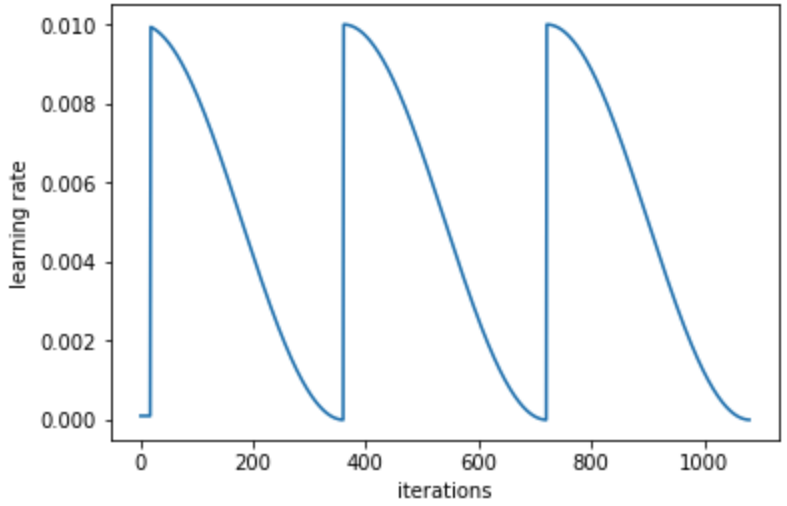
- **Step 1: Learning rate range test**
- Start training and increase learning rate linearly after each batch and calculate the loss.
- Then display loss as a function of learning rate.
- By manual inspection, select a range of suitable learning rates (look for strongest downward slope).



See Jeremy Howard's guide: https://youtu.be/Egp4Zajhzog?t=1674

Cyclical Learning Rate (CLR)

- **Step 2: Actual training**
- Train our model where the learning rate cyclically varies between the bounds that were found in step 1.
- You can combine with learning rate decay.



CLR in practise

- Fast.ai is the name of an online course *and* a deep learning framework built on top of PyTorch.
- Jeremy Howard from fast.ai uses CLR a lot.
- Jeremy's approach (see next slide):
  1. Download a pretrained convolutional encoder and add your own decoder (classifier)
  2. Train only the decoder for a few cycles (with cyclic learning rate). This is called transfer learning, and the purpose is just to get the weights of the decoder "approximately right".
  3. "Unfreeze" the encoder and run "learning rate range test" (see step 1 on previous slide)
  4. Plot loss vs. learning rate and pick suitable learning rate range
  5. Fine-tune your model for a couple of cycles (train all layers, including convolutional base).
- See demo here (Jupyter notebook).
- See brief explanation here (Youtube video – watch about 3 minutes).

CLR in practise

- Jeremy's approach:
  1. Download a pretrained convolutional encoder and add your own decoder (classifier)
  2. Train only the decoder for a few cycles (with cyclic learning rate). This is *transfer learning*.
  3. "Unfreeze" the encoder and run learning rate finder (see step 1 on previous slide)
  4. Plot loss vs. learning rate and pick suitable learning rate range
  5. Train your model for another few cycles (train all layers, including convolutional base). This is called *fine-tuning*.

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

Then run `fit_one_cycle` 4 times and see how we go. And we have a 2% error rate. So that's pretty good. Sometimes it's easy for me to recognize a black bear from a grizzly bear, but sometimes it's a bit tricky. This one seems to be doing pretty well.

```
learn.fit_one_cycle(4)
```

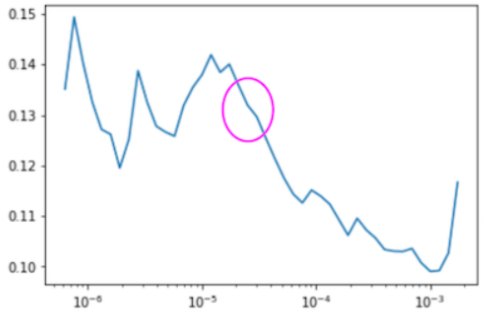
```
Total time: 00:54
epoch  train_loss  valid_loss  error_rate
1      0.710584    0.087024    0.021277    (00:14)
2      0.414239    0.045413    0.014184    (00:13)
3      0.306174    0.035602    0.014184    (00:13)
4      0.239355    0.035230    0.021277    (00:13)
```

```
learn.unfreeze()
```

Then we run the learning rate finder and plot it (it tells you exactly what to type). And we take a look.

```
learn.lr_find()
```

```
learn.recorder.plot()
```

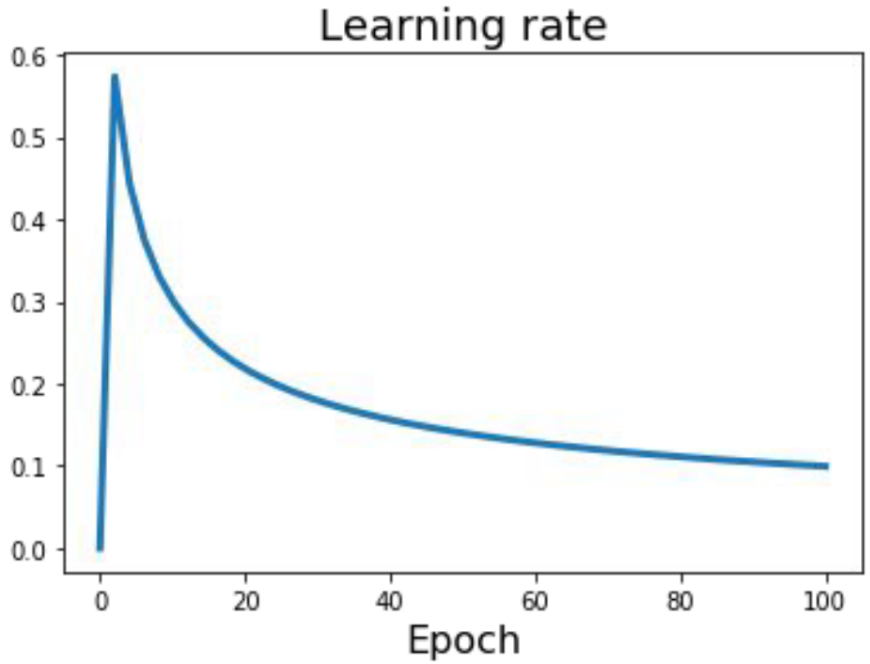



```
learn.fit_one_cycle(2, max_lr=slice(3e-5,3e-4))
```

```
Total time: 00:28
epoch  train_loss  valid_loss  error_rate
1      0.107059    0.056375    0.028369    (00:14)
2      0.070725    0.041957    0.014184    (00:13)
```

Linear warmup

- In some cases, deep neural networks can suffer from a sort of "early over-fitting".
- This can happen if your data is skewed or includes a cluster of related, strongly-featured observations.
- What happens then is that your model's initial training can skew badly toward those features – or worse, toward incidental features that aren't truly related to the topic at all.
- Warm-up is a way to reduce the primacy effect of the early training examples.
- Without it, you may need to run a few extra epochs to get the convergence desired, as the model un-trains those early superstitions.



Goyal et al (2017): Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Per-parameter adaptive learning rate methods

Motivation

- The approaches we've discussed so far **manipulate the learning rate globally and equally for all parameters**.
- Tuning the learning rates is an expensive process, so much work has gone into devising methods that can **adaptively tune the learning rates and even do so per parameter**.
- Many of these methods may still require other hyperparameter settings, but the argument is that they are well-behaved for a broader range of hyperparameter values than the raw learning rate.
- In this section we highlight some common adaptive methods you may encounter in practice:
  - Adagrad
  - RMSProp
  - Adam

Adagrad

- Adagrad adapts the learning rate to the parameters, performing **smaller updates** (i.e., low learning rates) for parameters associated with frequently occurring features, and **larger updates** (i.e., high learning rates) for parameters associated with infrequent features.
- Adagrad has been shown to improve the robustness of SGD.
- Basic idea:
  - Add element-wise scaling of the gradient based on the historical sum of squares in each dimension
  - "Per-parameter learning rates" or "adaptive learning rates"

$$G_{0,i} = 0$$
$$g_{t,i} = \nabla J(w_{t,i})$$
$$G_{t,i} = G_{t-1,i} + g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$$

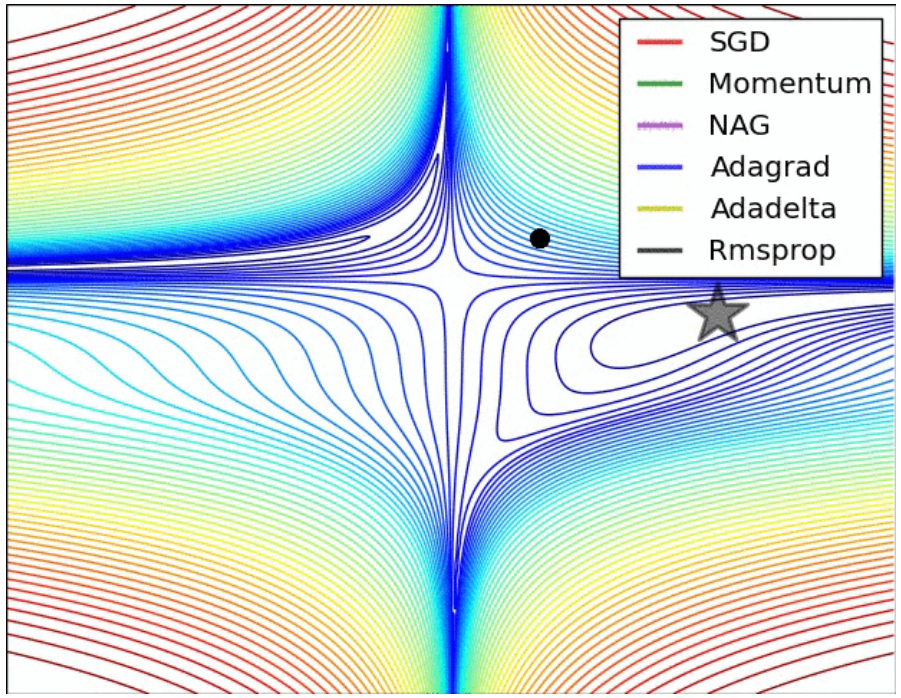
$g_{t,i}$ is the partial derivative of *i*'th parameter at time *t.*

$G_{t,i}$ is the sum of the squared partial derivative of *i*'th up to time *t.*

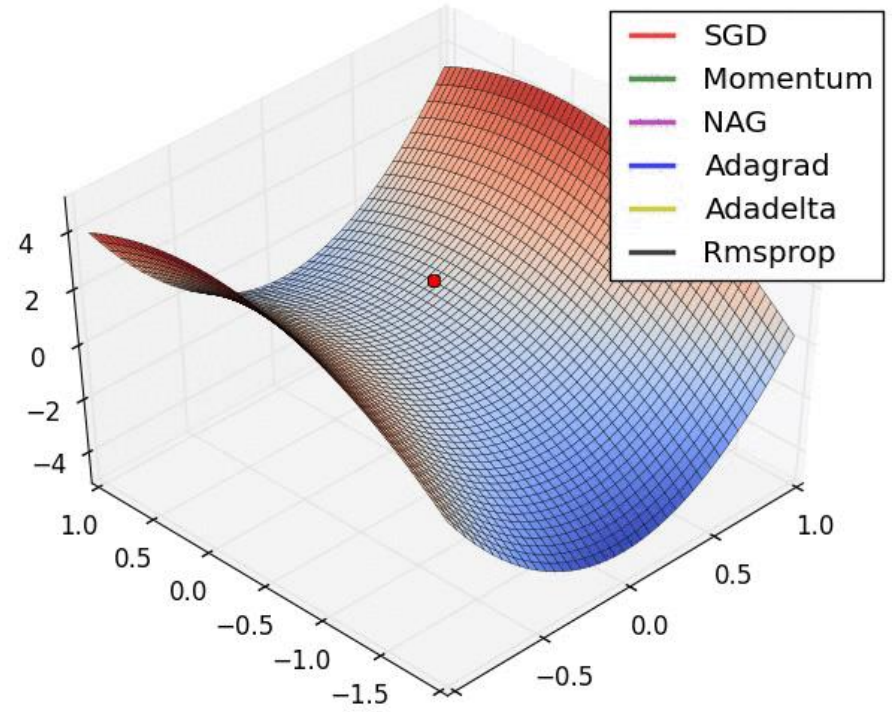
ε is a smoothing term to avoid division by 0 (usually 10^{-8} to 10^{-7}).

Adagrad

Progress along “steep” directions is damped, progress along “flat” directions is accelerated



Link to GIF animation:
http://ruder.io/content/images/2016/09/contours_evaluation_optimizers.gif



Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

RMSProp

- Adagrad's main weakness is its accumulation of the squared gradients in the denominator.
- Since every added term is positive, **the accumulated sum keeps growing during training**.
- This in turn **causes the learning rate to shrink and eventually become infinitesimally small**, at which point the algorithm is no longer able to acquire additional knowledge.
- Both RMSProp and Adadelta (not covered here) aim to resolve this flaw.

<u>Adagrad:</u>

$$G_{0,i} = 0$$
$$g_{t,i} = \nabla J(w_{t,i})$$
$$G_{t,i} = G_{t-1,i} + g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$$

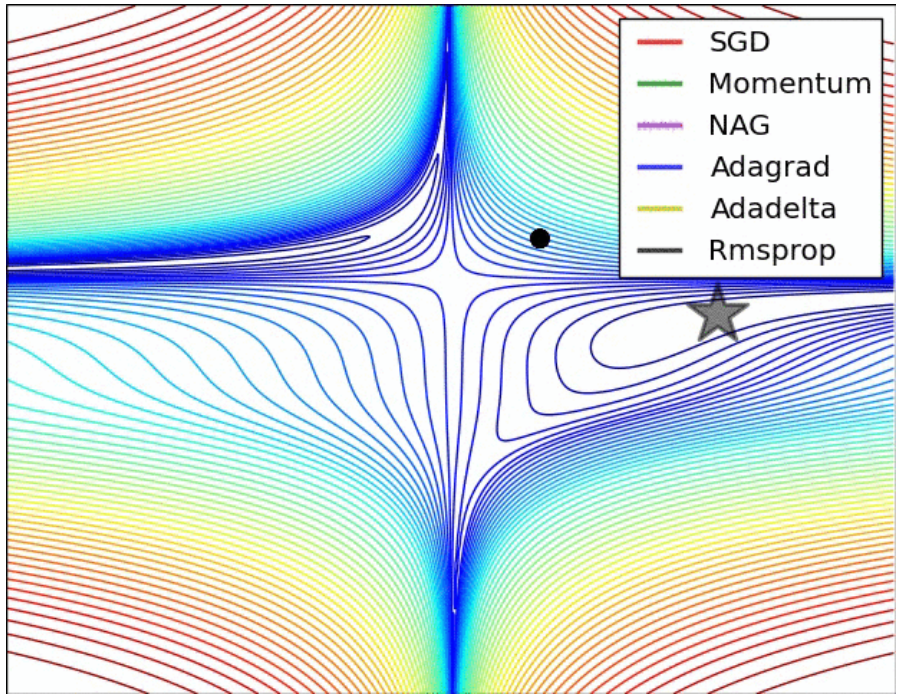
<u>RMSprop:</u>

$$G_{0,i} = 0$$
$$g_{t,i} = \nabla J(w_{t,i})$$
$$G_{t,i} = \gamma G_{t-1,i} + (1-\gamma) g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$$

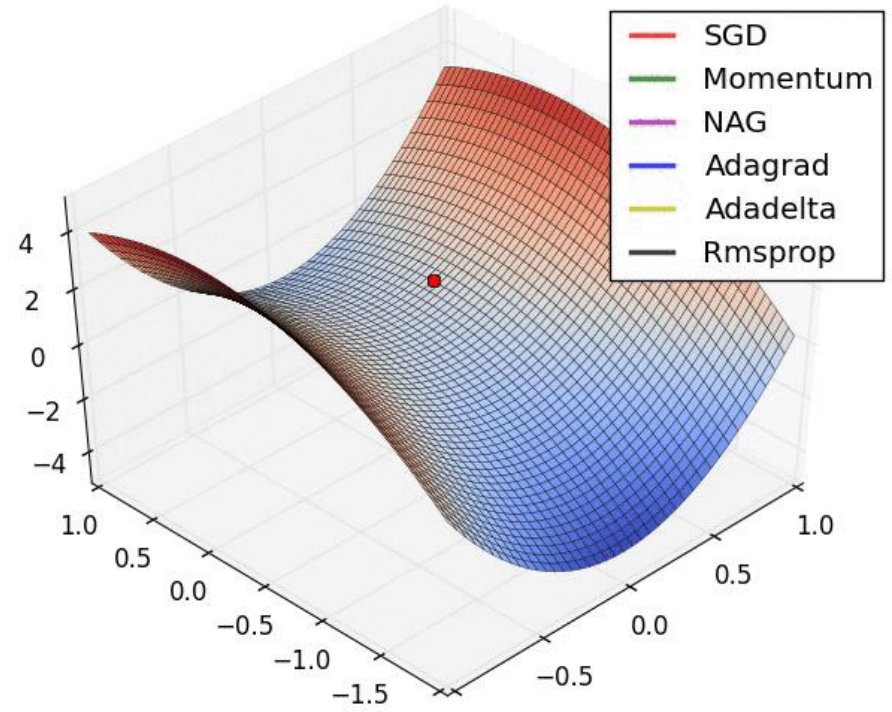
γ is a decay rate set to around 0.9

RMSProp

Slightly faster convergence compared to Adagrad



Link to GIF animation:
http://ruder.io/content/images/2016/09/contours_evaluation_optimizers.gif



Link to GIF animation:
http://ruder.io/content/images/2016/09/saddle_point_evaluation_optimizers.gif

Adam

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
- **Sort of like RMSProp with momentum:** In addition to storing an exponentially decaying average of past squared gradients, like RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.
- **Intuition:** Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Almost Adam:

$$m_{0,i}, v_{0,i} = 0$$
$$m_{t,i} = \beta_1 m_{t-1,i} + (1-\beta_1) g_{t,i}$$
$$v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2) g_{t,i}^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{v_{t,i}+\varepsilon}} m_{t,i}$$

Adam: Add bias correction (link)

$$\hat{m}_{t,i} = \frac{m_{t,i}}{1-\beta_1^t} \text{ and } \hat{v}_{t,i} = \frac{v_{t,i}}{1-\beta_2^t}$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{\hat{v}_{t,i}+\varepsilon}} \hat{m}_{t,i}$$

Kinga & Ba (2014): Adam: A Method for Stochastic Optimization

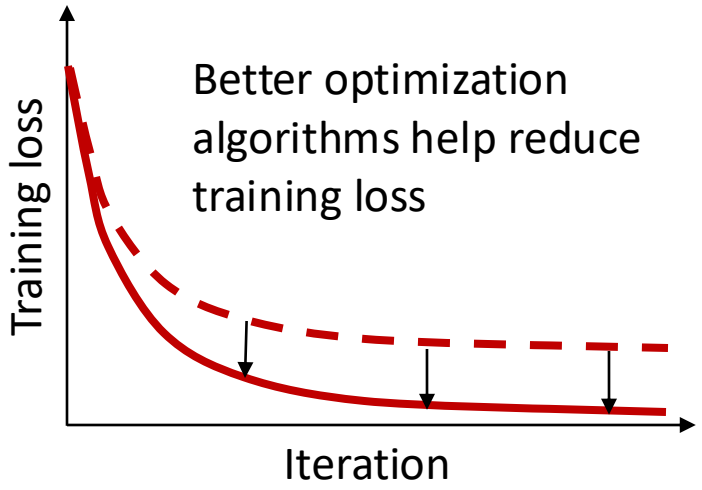
In practise

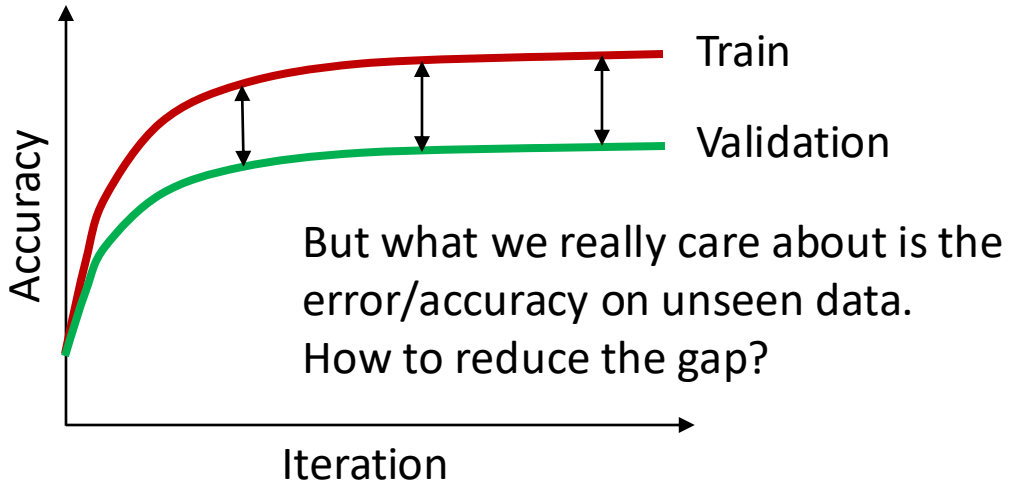
- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD + Momentum** can outperform Adam but may require more tuning of learning rate and learning rate schedule

Regularization

What is regularization?

- A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs (**unseen data**).
- Many strategies used in machine learning are **explicitly designed to reduce the test/validation error**, possibly at the expense of increased training error. These strategies are known collectively as **regularization**.
- The overall purpose is to **reduce overfitting**.



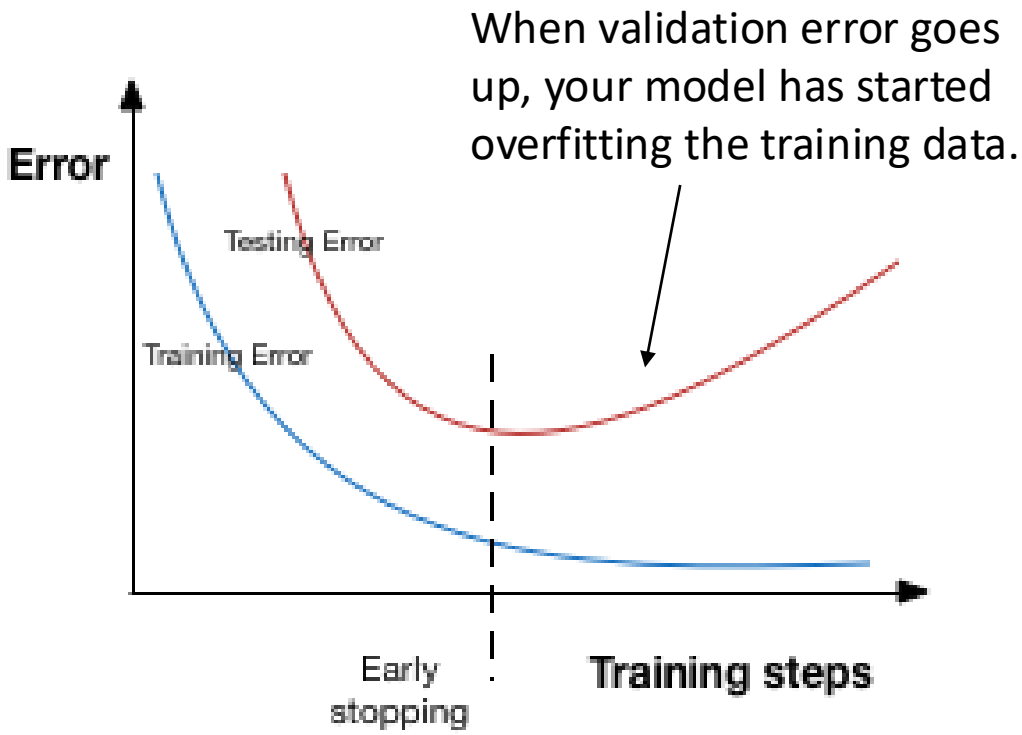


Overview

- Early stopping
- Model ensembles
- Dropout
- Data augmentation
- Noise robustness (Cutout, Mixup and Label smoothing)
- Multi-task learning
- Regularization terms

Early stopping

- Always do this!
- In early stopping, we keep one part of the dataset as the validation set.
- **When performance on the validation set is getting worse, we immediately stop the training on the model.**
- Alternatively, train for a long time, but always keep track of the **model snapshot** that worked best on the validation set.
- Early stopping can be automated (e.g., using callback in Keras), but in practise its often easier to just inspect the loss curves manually.



Model ensembles

- **Basic idea:** Train multiple (possibly independent) models, and at test time **average their predictions**.
- In practice, this gives a few percent extra performance (see for instance AlexNet paper).
- There are a few approaches to forming an ensemble:
  - Same model, different initializations.
  - Top models discovered using, say, cross-validation.
  - Different checkpoints of a single model (see for instance "snapshot ensemble" paper referenced below).
  - Running average of parameters during training.
- Snapshot ensembles: Train 1, get M for free

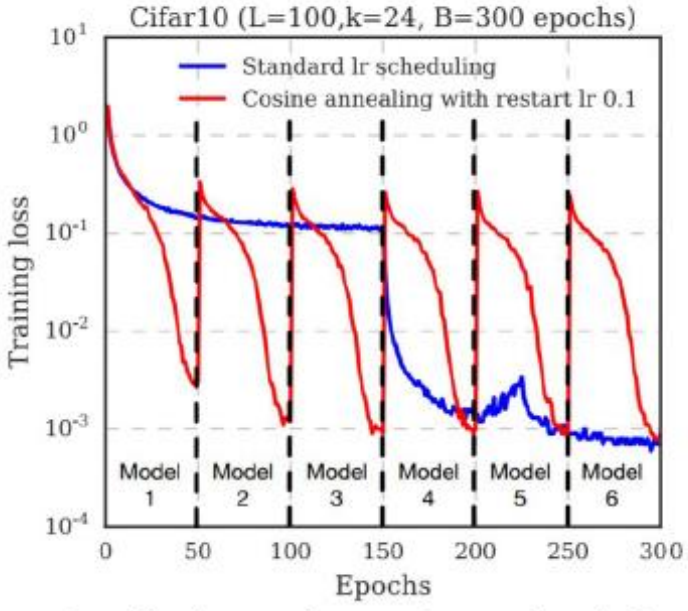
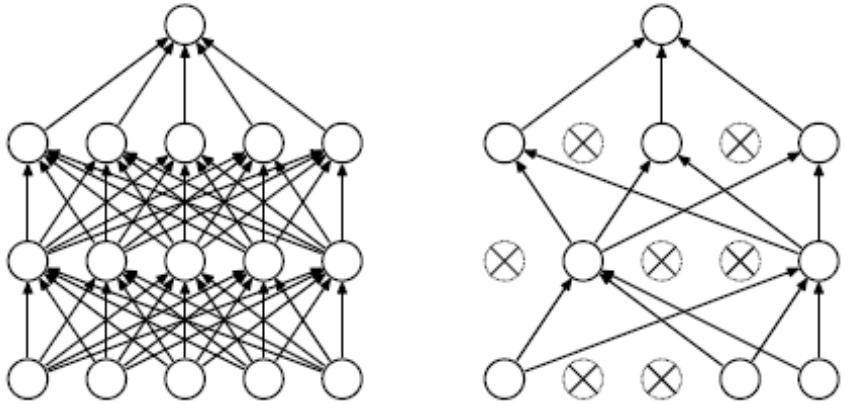


Figure 2 (Snapshot ensemble paper): Training loss of 100-layer DenseNet on CIFAR10 using standard learning rate (blue) and M = 6 cosine annealing cycles (red). The intermediate models, denoted by the dotted lines, form an ensemble at the end of training

Dropout

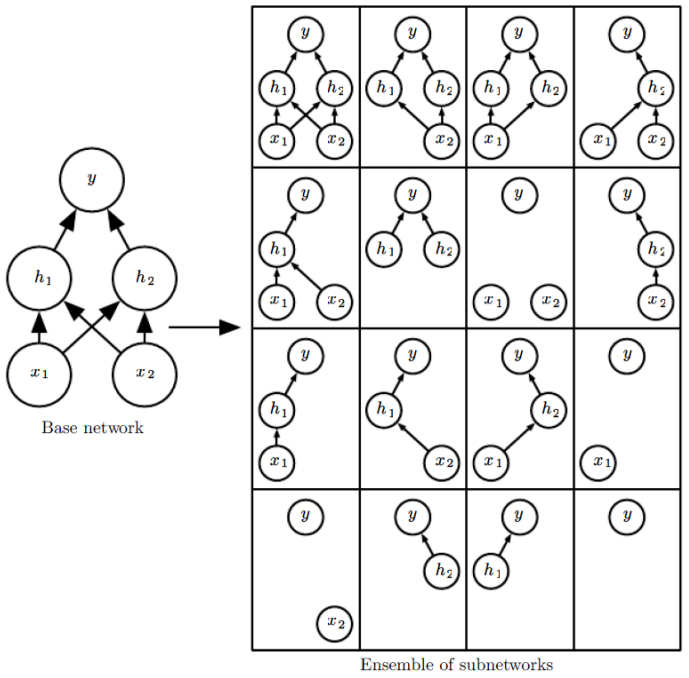
- Dropout provides a computationally inexpensive but powerful method of regularization.
- Dropout can be thought of as turning your neural network into a **giant ensemble of smaller models** during training and combining all these models into one model at test time.
- **Basic idea:** In each forward pass, randomly set some neurons to zero.
- Probability p of dropping is a hyperparameter ($p = 0.5$ is common)



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Figure 7.6 (book)



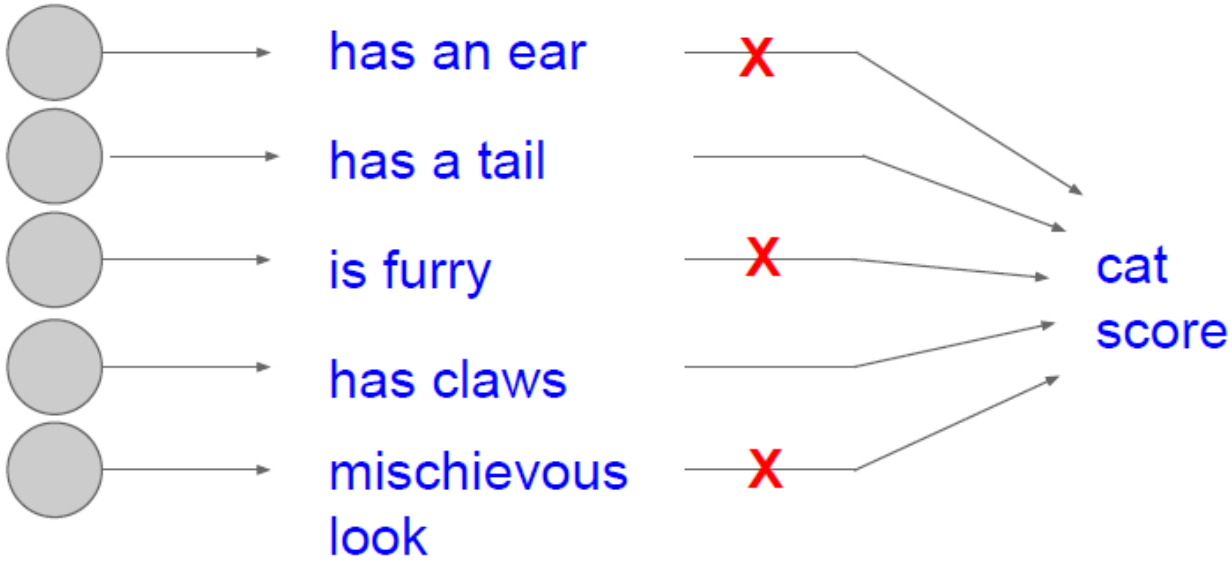
Srivastava et al. (2014): Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Dropout

- How can this possibly be a good idea?
- Another interpretation: Dropout prevents neurons from co-adapting too much. In other words, it forces the network to learn a redundant representation, which eventually makes the model better at generalizing (by reducing overfitting).

With dropout the network cannot rely on any single neuron to measure the "cat-ness".

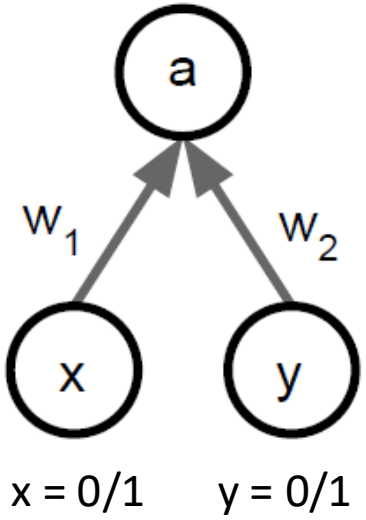
Instead, it is forced to distribute its idea of cat-ness across many different neurons.



https://towardsdatascience.com/simplified-math-behind-dropout-in-deep-learning-6d50f3f47275

Dropout at test time

- Dropout makes our output random!
- Want to “average out” the randomness at test-time.
- At test time all neurons are active always => We must scale the activations so that for each neuron: output at test time = expected output at training time
- **Solution:** At test time, perform **correction** by multiplying output by dropout probability (p).



During training ($p = \frac{1}{2}$):

$$\mathrm{E}[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + w_2 y) + \frac{1}{4}(0x + 0y) = \frac{1}{2}(w_1 x + w_2 y)$$

At test time (without correction):

$$\mathrm{E}[a] = w_1 x + w_2 y$$

At test time (with correction):

$$\mathrm{E}[a] = p(w_1 x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

Same expectation during training and at test time

Data augmentation

- The best way to make a machine learning model generalize better is to **train it on more data**.
- In practice, the amount of data we have is limited.
- One way to get around this problem is to **create fake data** and add it to the training set.
- For most visual recognition tasks, it is reasonably straightforward to create new fake data.
- **Examples:**
  - Keras ImageDataGenerator class (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator)
  - Albumentations (https://github.com/albumentations-team/albumentations)

Data augmentation – types of variation

Viewpoint variation

Scale variation

Deformation

Occlusion

Illumination conditions

Background clutter

Intra-class variation

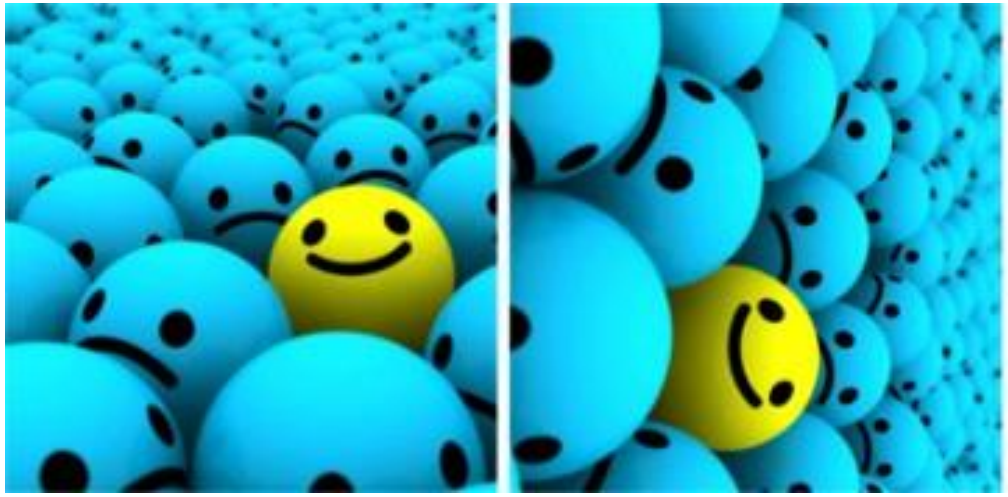
Data augmentation - examples

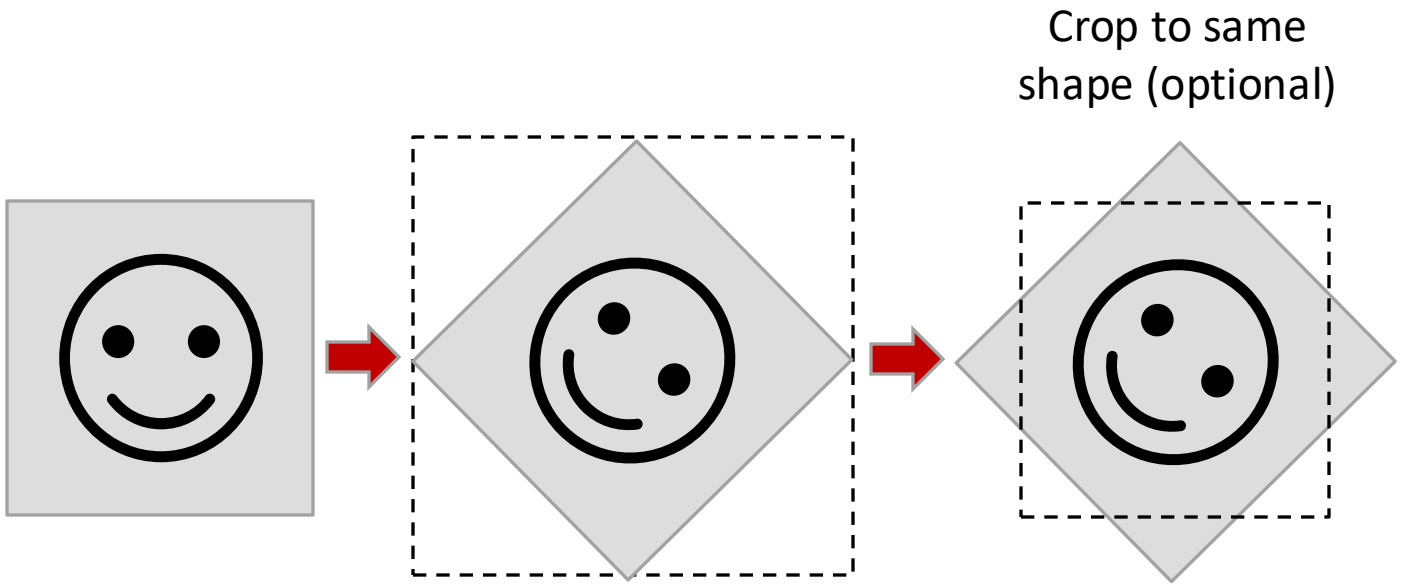
- **Flip -** flip images horizontally and vertically.
- Be careful only to flip images, if the original class properties are preserved (so not MNIST).

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation - examples

- **Rotation**
- Note that image dimensions may not be preserved after rotation.





https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation - examples

- **Scaling**
- The image can be scaled outward or inward.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation - examples

- **Crop -** randomly sample a section from the original image and resize this section to the original image size.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation - examples

- **Translation -** just involves moving the image along the X or Y direction (or both).
- Very useful as most objects can be located at almost anywhere in the image.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation - examples

- **Gaussian noise –** adding just the right amount of noise can enhance the learning capability.
- Overfitting usually happens when your neural network tries to learn **high frequency features** (patterns that occur a lot) that may not be useful. Gaussian noise, which has zero mean, essentially has data points in all frequencies, effectively distorting the high frequency features.

https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation – filling empty gaps

- **Filling in the empty gaps** - after we perform spatial transformations, we need to preserve our original image size. Since our image does not have any information about things outside its boundary, we need to make some assumptions.
- Deep learning frameworks have some standard ways with which you can decide on how to fill the unknown space. They are defined as follows:

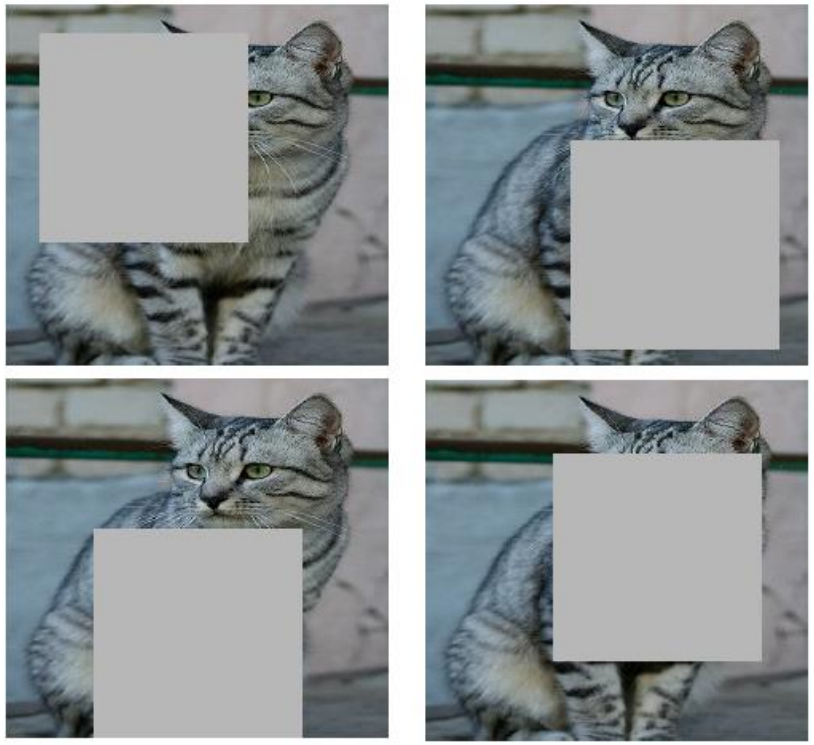
Constant value | Extent edge value | Reflect values | Symmetric | Wrap

Noise robustness

- A common pattern in regularization is:
  - **Training:** Add some kind of randomness
  - **Test:** Marginalize over the noise (means to average out the noise…)
- Examples: Dropout and data augmentation
- Other examples
  - Cutout
  - Mixup
  - Label smoothing

Noise robustness

- Cutout:
  - **Training:** Set random image region to zero (or constant value)
  - **Test:** Use full image
- Teaches network to become **invariant to occlusion.**

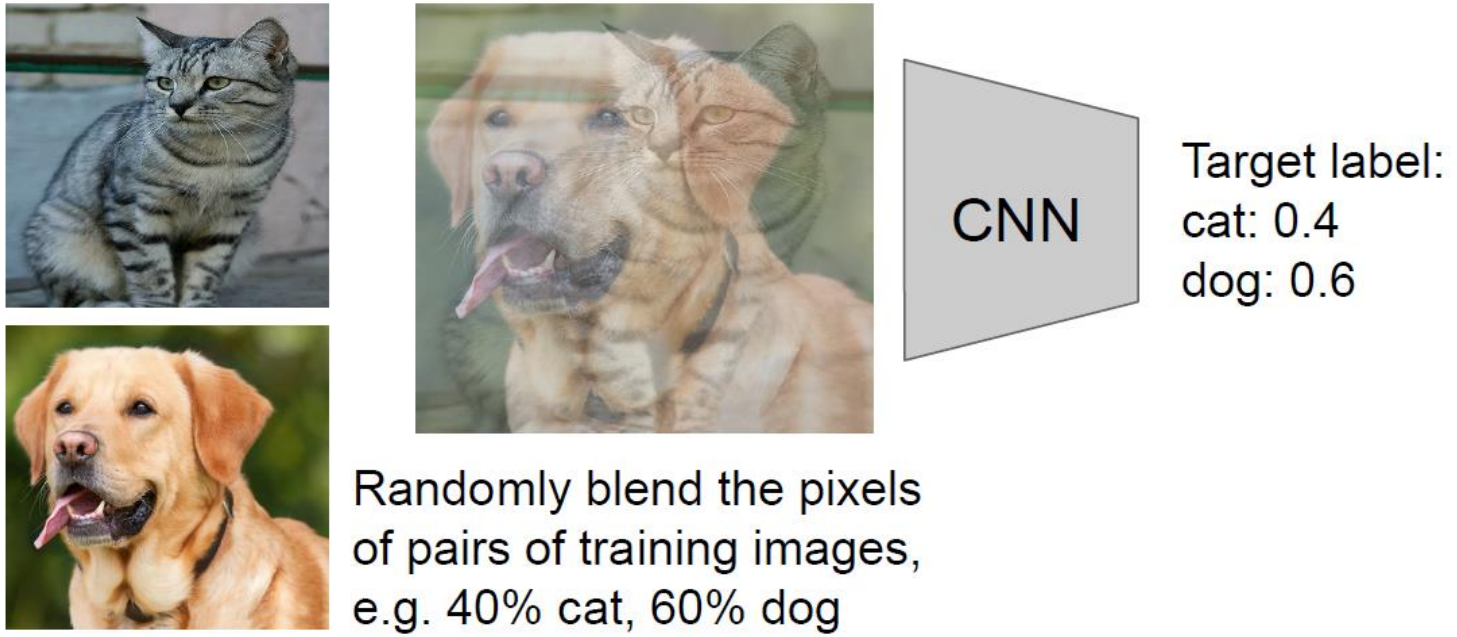


Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

https://arxiv.org/abs/1708.04552

Noise robustness

- Mixup:
  - **Training:** Train on random blends of images
  - **Test:** Use original image
- Mixup can be understood as a form of data augmentation that encourages the model to behave linearly in-between training examples.
- This linear behaviour reduces the number of undesirable oscillations when predicting outside the training examples (unseen data).



https://arxiv.org/abs/1710.09412

Noise robustness

- Label Smoothing:
  - **Training:** Train with slightly noisy labels
  - **Test:** Use original images
- The problem with cross-entropy loss is the hard targets. The model has to produce large *logit* value for the correct label. It encourages the differences between the largest logit and all others to become large, and this, combined with the bounded gradient reduces the ability of the model to adapt, resulting in a model **too confident about its predictions**.
- This is particularly problematic if the dataset has some number of mistakes in the *y* labels (which most data sets have). All of this, in turn, can **lead to overfitting**.
- The assumption underlying Label Smoothing is that for some small constant *e*, the training set label y is correct with probability 1 − *e*, and otherwise any of the other possible labels might be correct. Label smoothing regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $e/(k-1)$ and $1 - e$, respectively.

https://medium.com/@nainaakash012/when-does-label-smoothing-help-89654ec75326

Szegedy (2015): Rethinking the Inception Architecture for Computer Vision

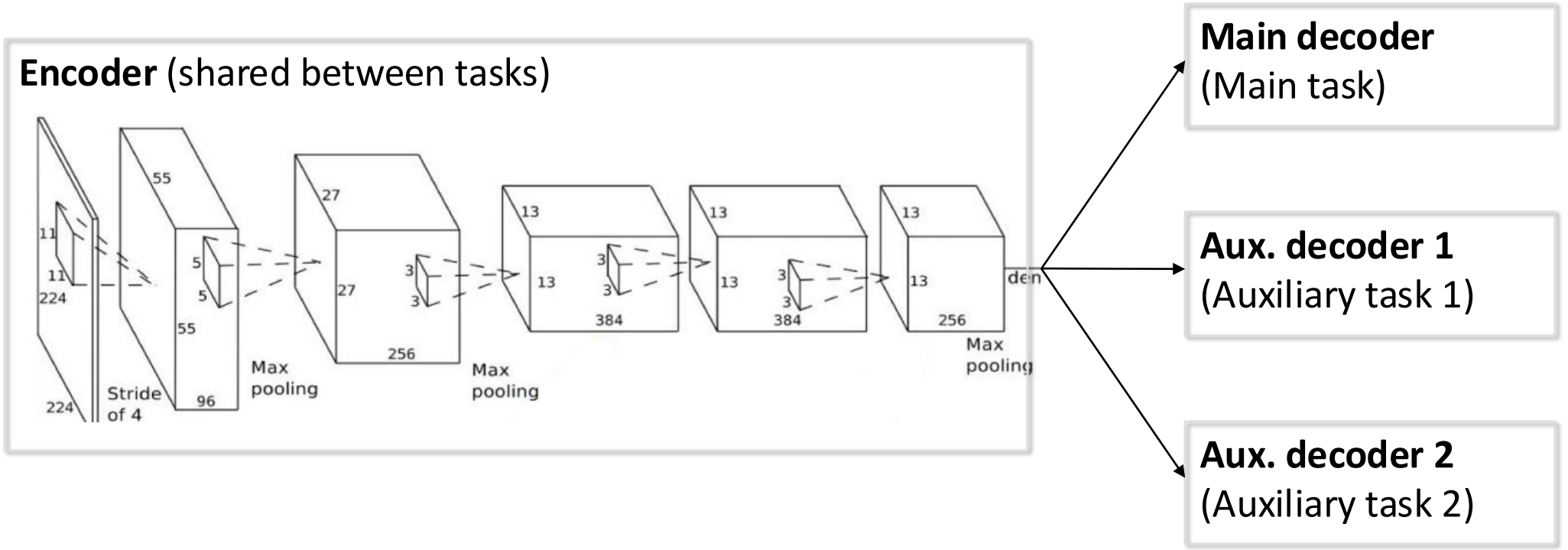
Multi-task learning

- We generally train a single model or an ensemble of models to perform our desired task by **focusing on optimizing for a particular metric**.
- By being laser-focused on our single task, we ignore information that might help us do even better on the metric we care about. Specifically, this information comes from the training signals of related tasks.
- By **sharing representations between related tasks**, we can enable our model to generalize better on our original task.
- This approach is called Multi-Task Learning (MTL).

http://ruder.io/multi-task/index.html#fn3

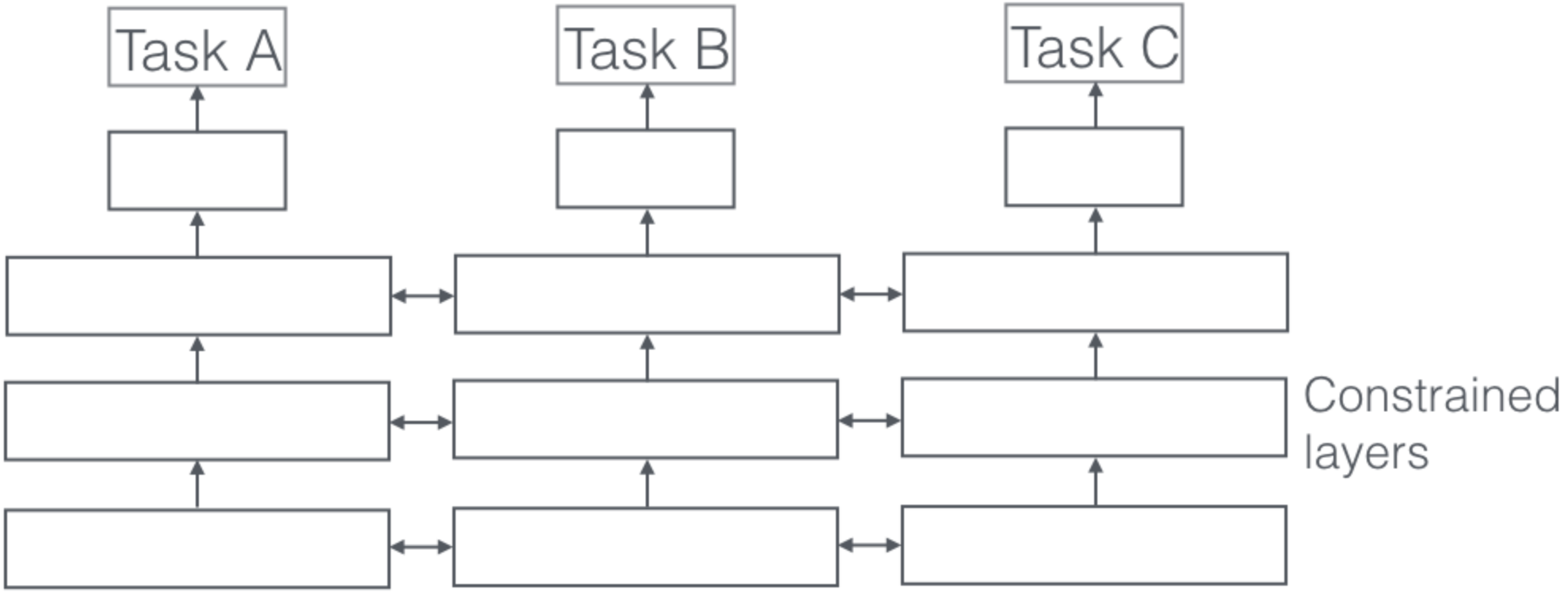
Multi-task learning

- **Hard parameter sharing -** train the same encoder to solve multiple tasks (with multiple decoders)
- At test time only use the decoder of your main task.
- Hard parameter sharing greatly reduces the risk of overfitting.



Multi-task learning

- **Soft parameter sharing -** In soft parameter sharing on the other hand, each task has its own model with its own parameters. The distance between the parameters of the model is then regularized in order to encourage the parameters to be similar.



Multi-task learning

- **Why does it work?**
- MTL effectively increases the sample size that we are using for training our model.
- As all tasks are at least somewhat noisy, when training a model on some task A, our aim is to learn a good representation for task A that ideally ignores the data-dependent noise and generalizes well.
- As different tasks have different noise patterns, a **model that learns two tasks simultaneously is able to learn a more general representation**.
- Learning just task A bears the risk of overfitting to task A, while learning A and B jointly enables the model to obtain a better representation F through **averaging the noise patterns**.

Multi-task learning: examples

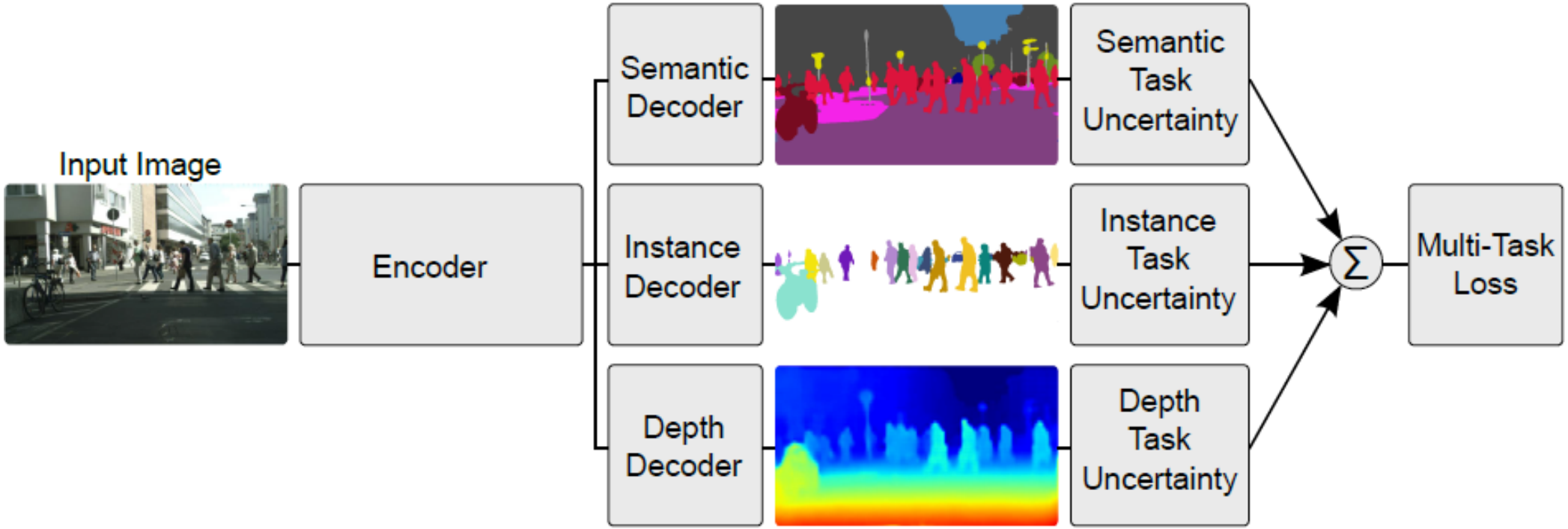
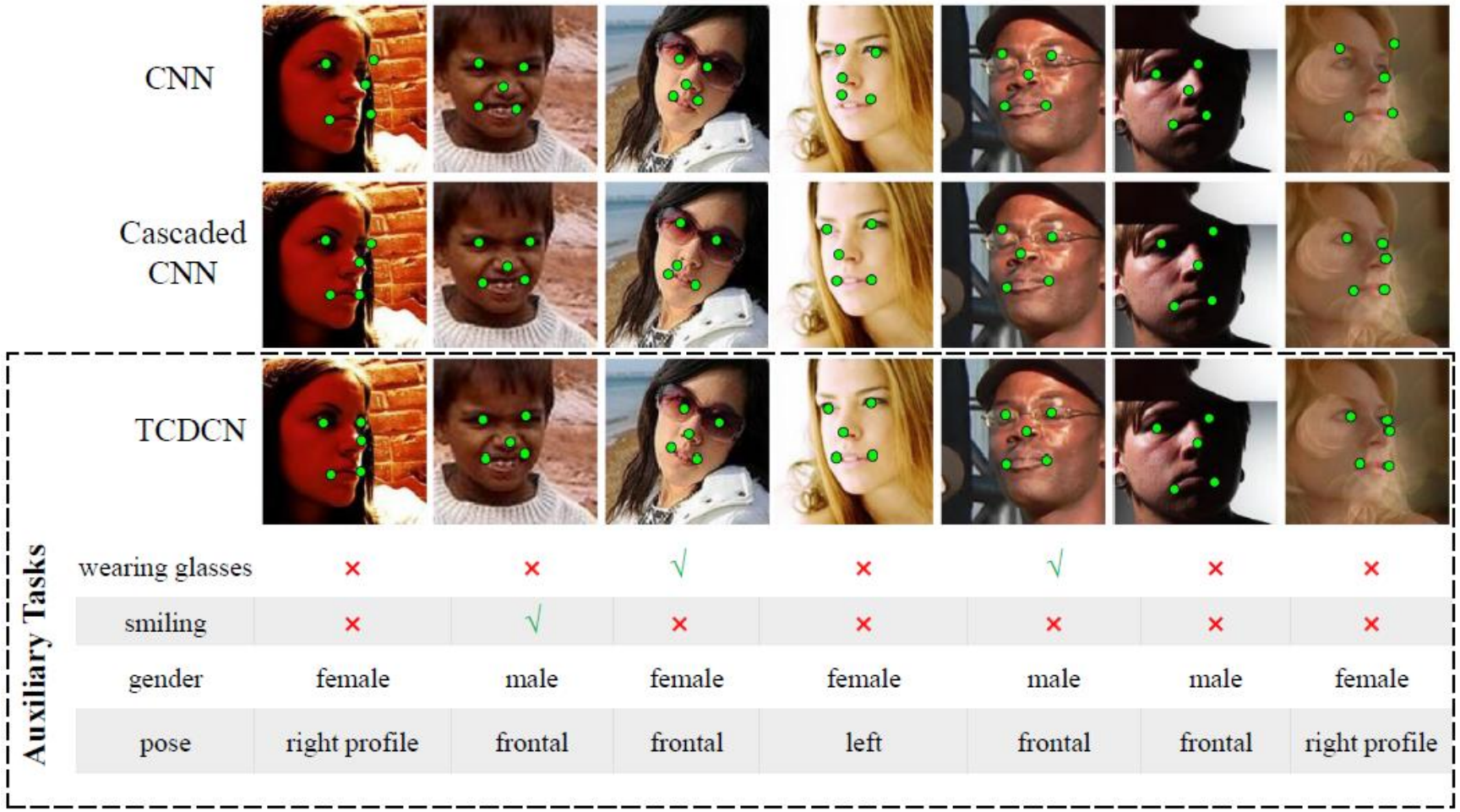


Figure 1: **Multi-task deep learning.** We derive a principled way of combining multiple regression and classification loss functions for multi-task learning. Our architecture takes a single monocular RGB image as input and produces a pixel-wise classification, an instance semantic segmentation and an estimate of per pixel depth. Multi-task learning can improve accuracy over separately trained models because cues from one task, such as depth, are used to regularize and improve the generalization of another domain, such as segmentation.

Kendall et al: Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics

Multi-task learning: examples



Zhang et al: Facial Landmark Detection by Deep Multi-task Learning

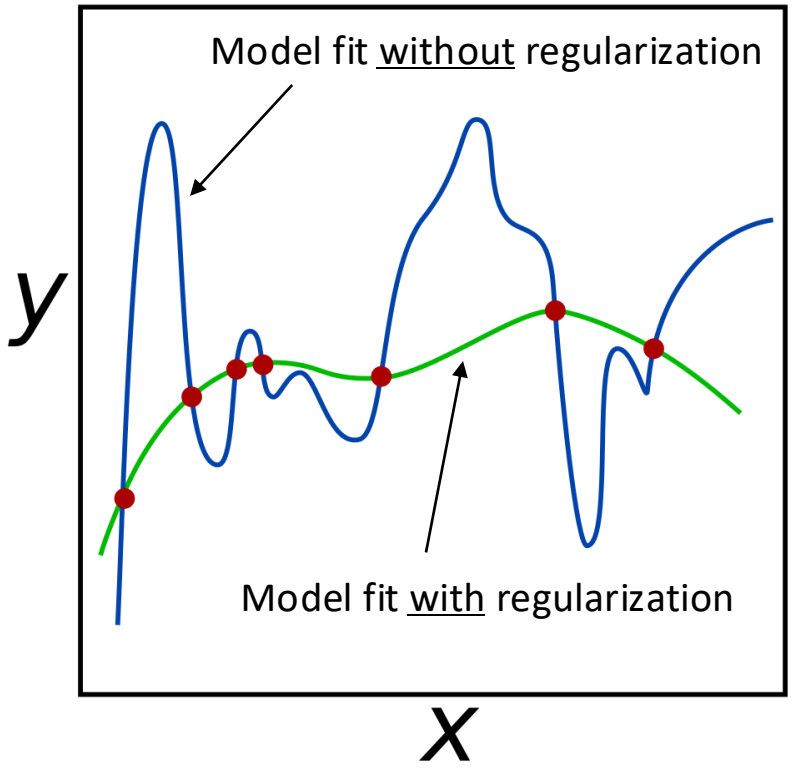
And of cause: Weight decay

- Extend the data loss $J(w)$ with an extra term

$$J_{reg}(w) = J(w) + \lambda R(w)$$

where R(w) is either the L1-norm or the L2-norm of w.

- The last term as called the *regularization term*, and λ is the regularization parameter.
- λ determines the trade-off between minimizing the data loss and minimizing the model parameters w.
- By keeping the weights small, the regularization term makes the model simpler to **avoid overfitting**.
- So, weight decay pushes against fitting the data *too* well, so **we don't fit noise in the data**.



Effect of weight decay

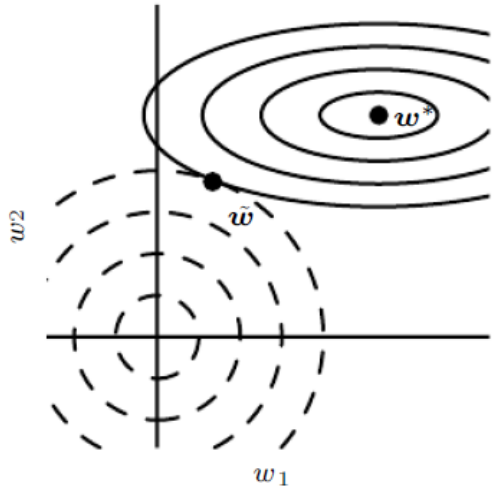
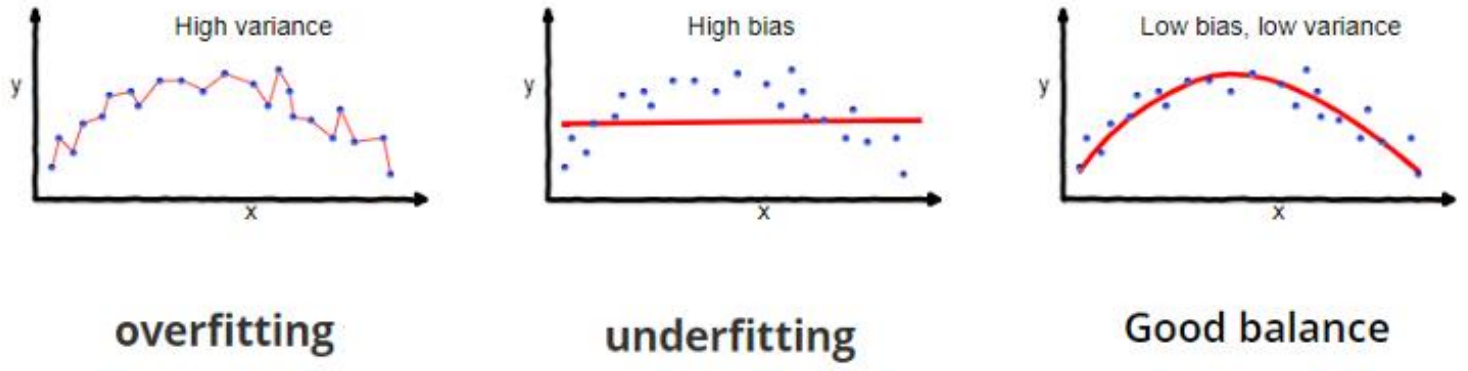
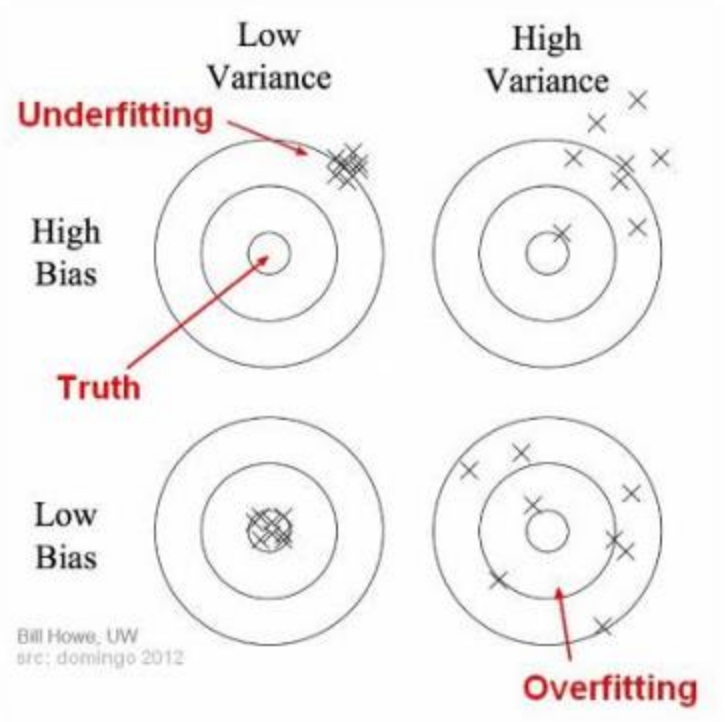


Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little.

Side-note: Bias-variance tradeoff

- Bias and variance are two types of prediction errors
- **Bias** is the difference between the average prediction of our model and the correct value which we are trying to predict. High bias oversimplifies the model (underfitting).
- **Variance** is the variability of model prediction at a given point; it tells us about the spread of our predictions. Model with high variance pays a lot of attention to training data and does not generalize to unseen data (overfitting).

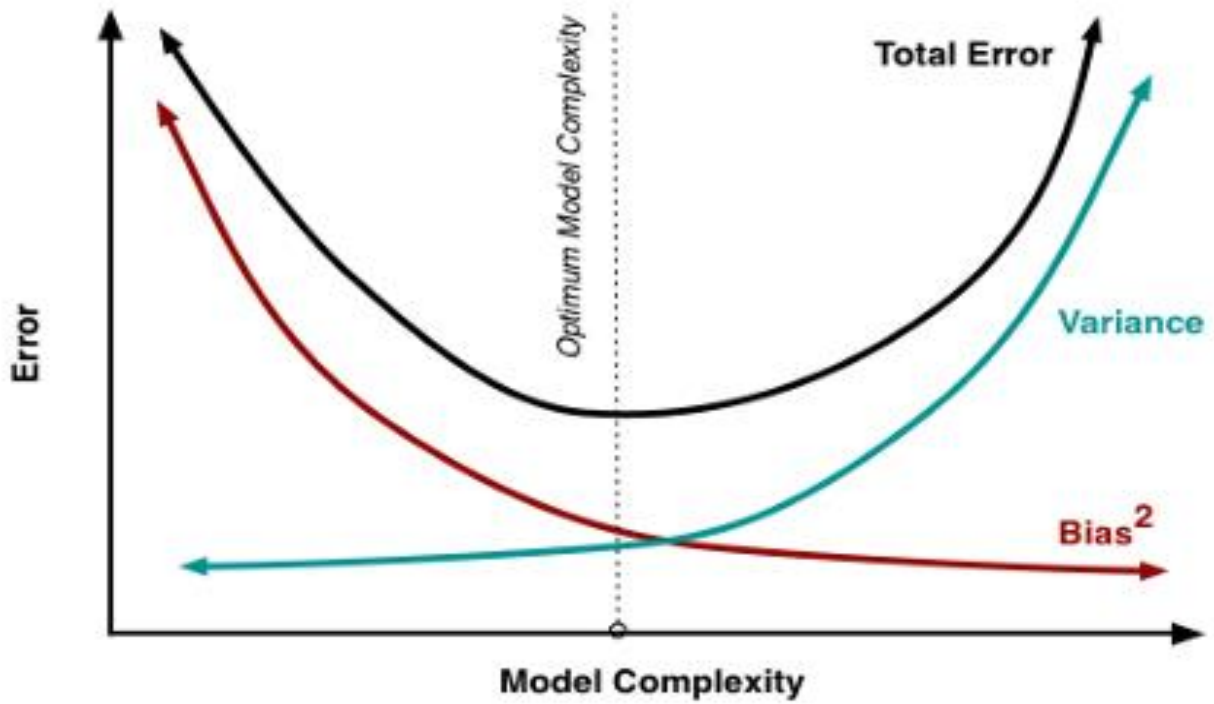




https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

Side-note: Bias-variance tradeoff

- If our model is too simple and has very few parameters, then it may have high bias and low variance. On the other hand, if our model has large number of parameters then it's going to have high variance and low bias. So, we need to find the right/good balance without overfitting and underfitting the data. This is essentially the bias-variance trade-off.



https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

In practise

- Always use early stopping and batch normalization.
- Regularization terms (weight decay) are often included by default.
- Consider dropout for large fully-connected layers.
- Data augmentation almost always a good idea.
- Try cutout, mixup, label smoothing especially for small classification datasets.
- Try multi-task learning if your dataset has training data to solve more than one task.

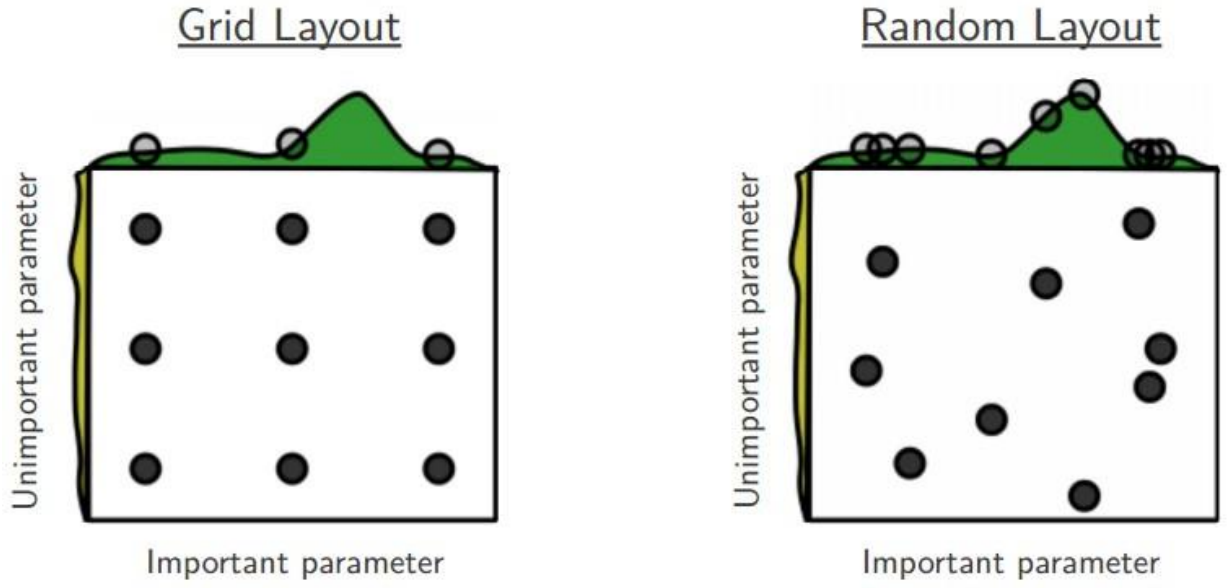
Hyperparameter tuning

Motivation

- As we've seen, training neural networks can involve **many hyperparameter settings**.
- The most common hyperparameters include:
  - the initial learning rate
  - learning rate decay schedule (such as the decay constant)
  - regularization strength (L2 penalty, dropout strength)
  - batch size
- There are many more relatively less sensitive hyperparameters, for example in per-parameter adaptive learning methods, the setting of momentum and its schedule, etc.
- **How do we find the optimal hyperparameters?**
- Really nice guide: https://towardsdatascience.com/guide-to-choosing-hyperparameters-for-your-neural-networks-38244e87dafe

General recommendations

- Performing hyperparameter search can take many hours/days/weeks.
- **Use a single validation set of respectable size.**
- **Search for hyperparameters on log scale:** A typical sampling of the learning rate would be 10^{-6}, 10^{-5}, 10^{1}. Some parameters (e.g., dropout) are instead usually searched in the original scale.
- **Prefer random search to grid search** (it's usually faster and easier to spot important parameters).



Bergstra & Bengio: Random Search for Hyper-Parameter Optimization

General recommendations

- **Careful with best values on border.** Sometimes it can happen that you're searching for a hyperparameter (e.g., learning rate) in a bad range. Once we receive the results, it is important to double check that the final learning rate is not at the edge of this interval, or otherwise you may be missing more optimal hyperparameter setting beyond the interval.
- **Stage your search from coarse to fine.** In practice, it can be helpful to first search in coarse ranges and then depending on where the best results are turning up, narrow the range.
- **Start with 1 epoch or less (coarse).** Also, it can be helpful to perform the initial coarse search while only training for 1 epoch or even less, because many hyperparameter settings can lead the model to not learn at all, or immediately explode with infinite cost.
- **… then add more epochs (fine).** The second stage could then perform a narrower search with 5 epochs, and the last stage could perform a detailed search in the final range for many more epochs (for example).

Bengio (2012): Practical recommendations for gradient-based training of deep architectures

Possible strategy

- Step 1: Check initial loss

  - Just a sanity check
  - Does it seem reasonable?

Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample

- Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches)
- This should always be possible if 1) your model has enough capacity, and 2) you remember to set regularization to zero.
- Fiddle with architecture, learning rate, batch size, weight initialization
- Loss not going down? Learning rate too low, bad initialization, batch size too low
- Loss explodes to Inf or NaN? Learning rate too high, bad initialization

Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down

- Use the architecture from the previous step, use all training data, turn on small weight decay
- Find a learning rate that makes the loss drop significantly within ~100 iterations
- Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Course grid, train for 1-5 epochs

- Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.
- Good weight decay to try: 1e-4, 1e-5, 0

Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Course grid, train for 1-5 epochs
- Step 5: Refine grid, train longer

- Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Possible strategy

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Course grid, train for 1-5 epochs
- Step 5: Refine grid, train longer
- **Step 6: Look at loss curves**

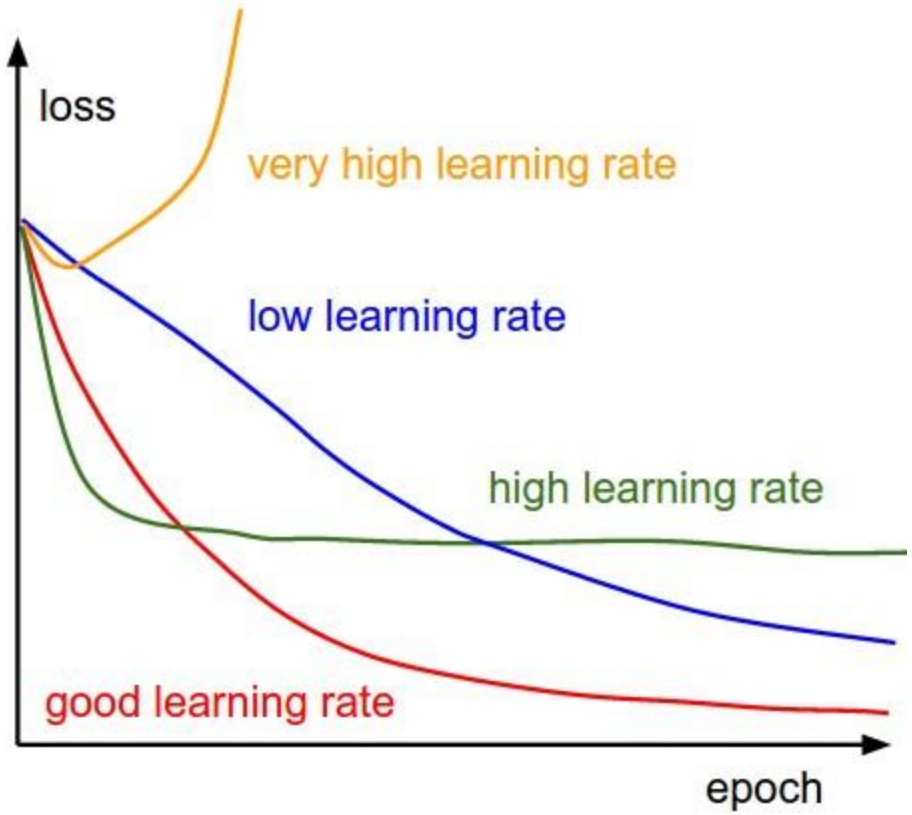
- This is the final and most important step (explained in more detail in the next section)
- If the loss curves don't look right, you'll need to go back an repeat one or more of the previous steps.

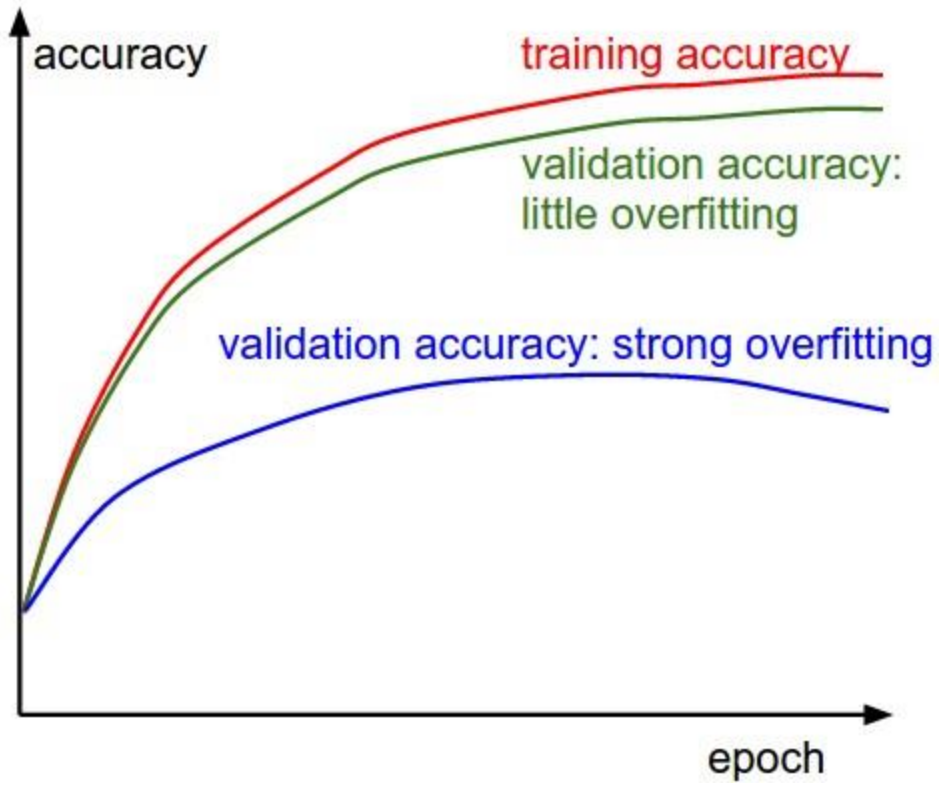
Babysitting the learning process

What to monitor during training?

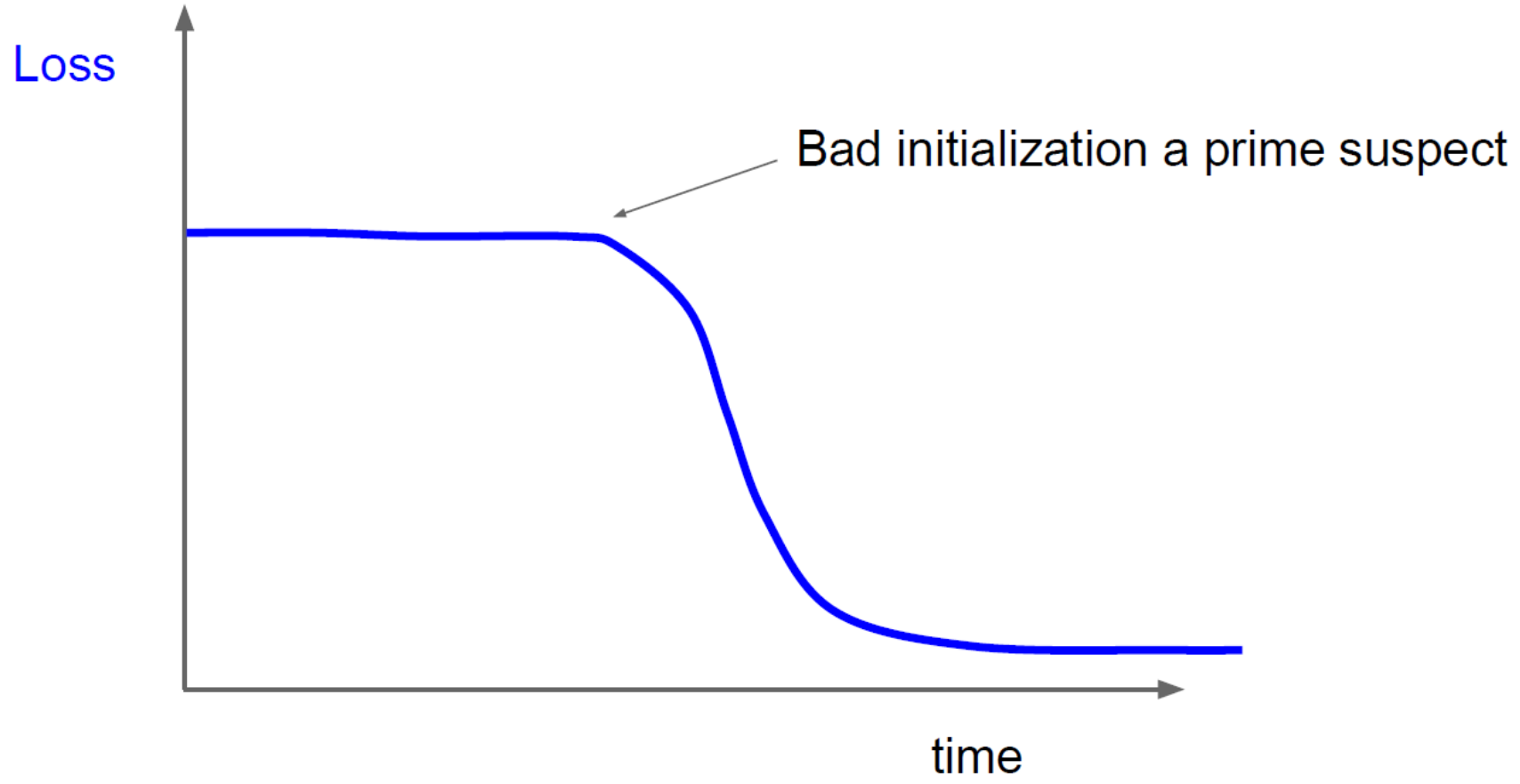
- There are multiple useful quantities you should monitor during training of a neural network.
- The most important ones are the **loss curves** (including validation error or accuracy).
- These curves are the window into the training process and should be utilized to get intuitions about different hyperparameter settings and how they should be changed for more efficient learning.
- Other useful things to monitor include
  - Activation / Gradient distributions per layer
  - First-layer visualizations

Quick summary

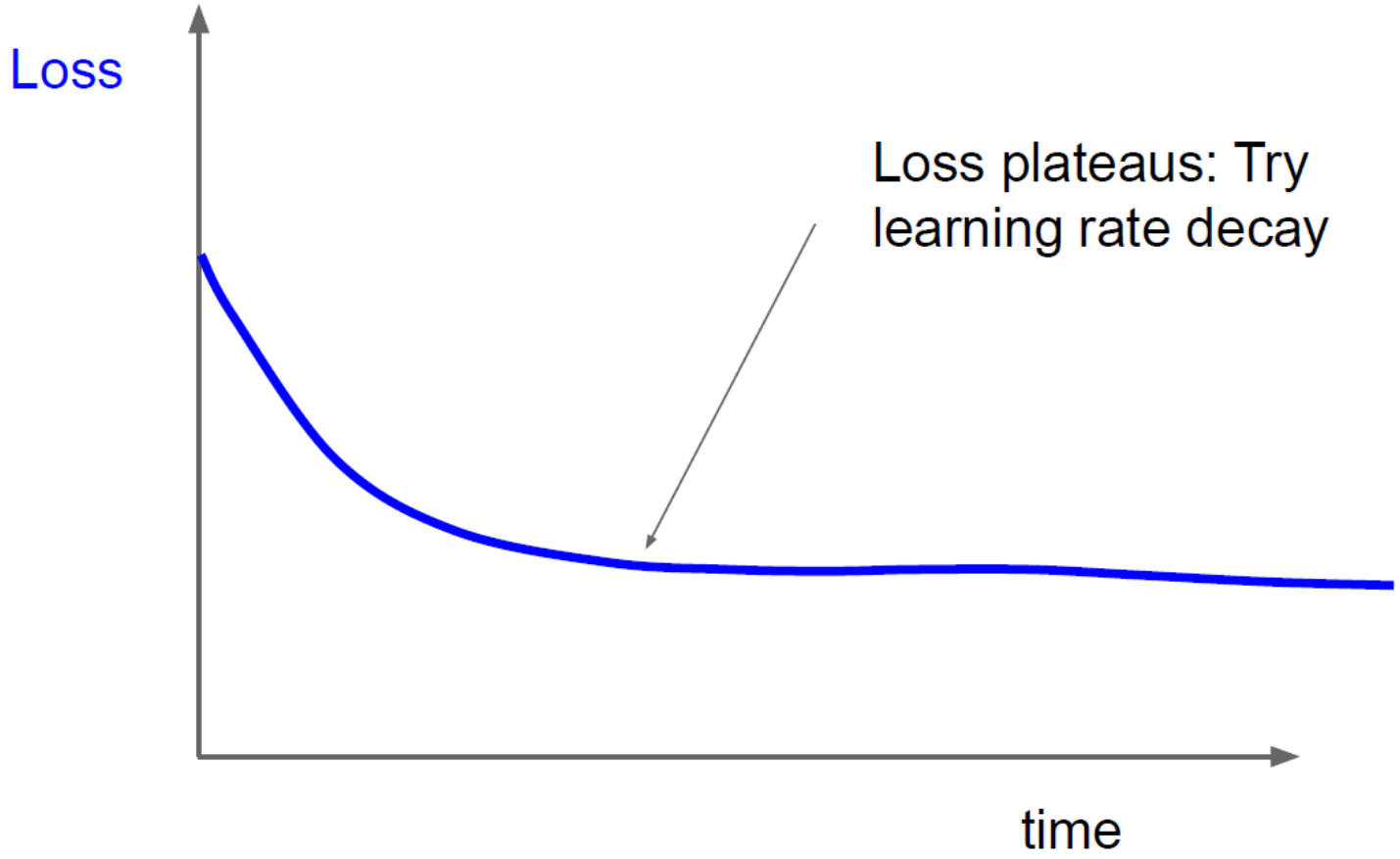




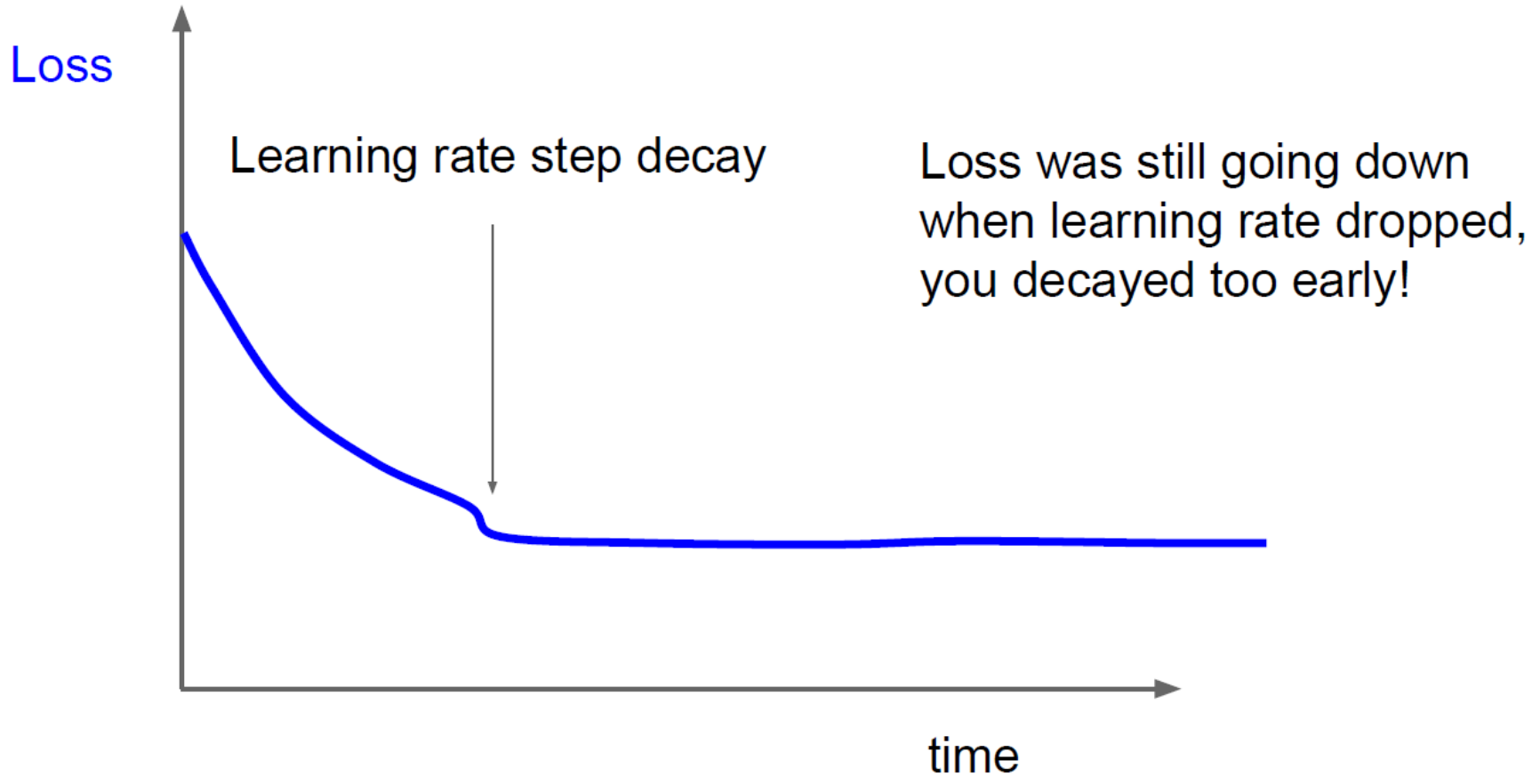
Loss curves



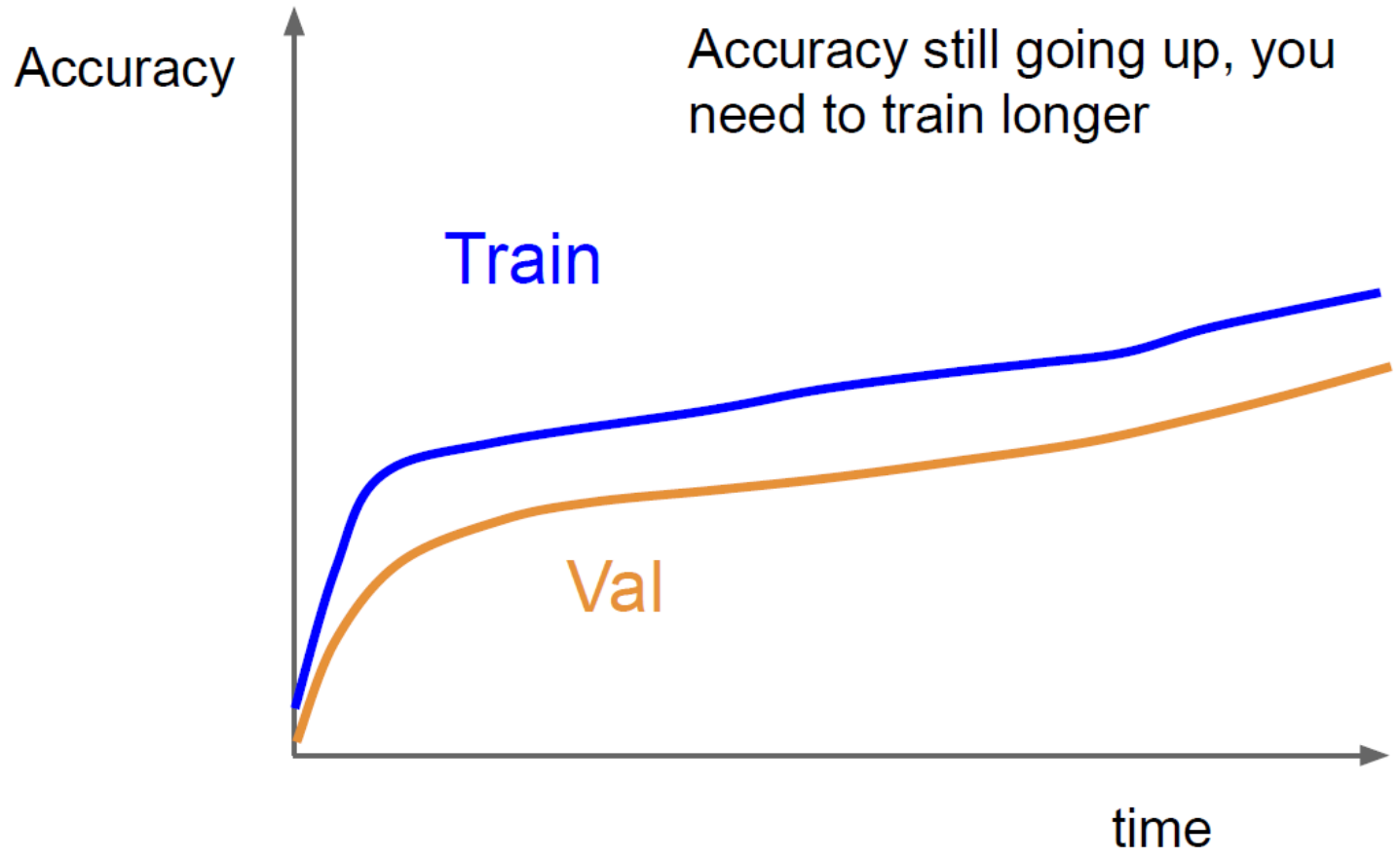
Loss curves



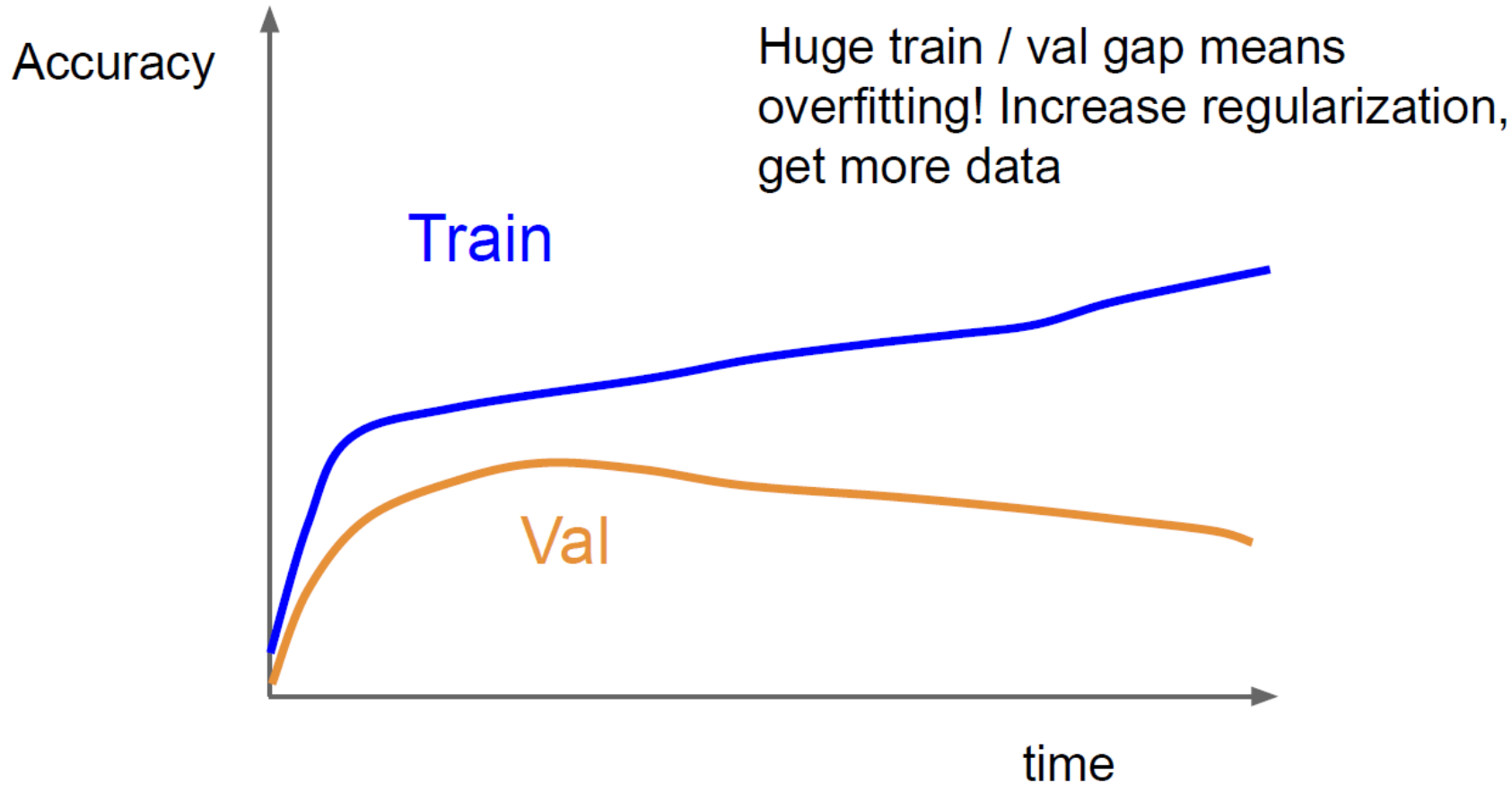
Loss curves


Loss
Learning rate step decay
Loss was still going down
when learning rate dropped,
you decayed too early!
time

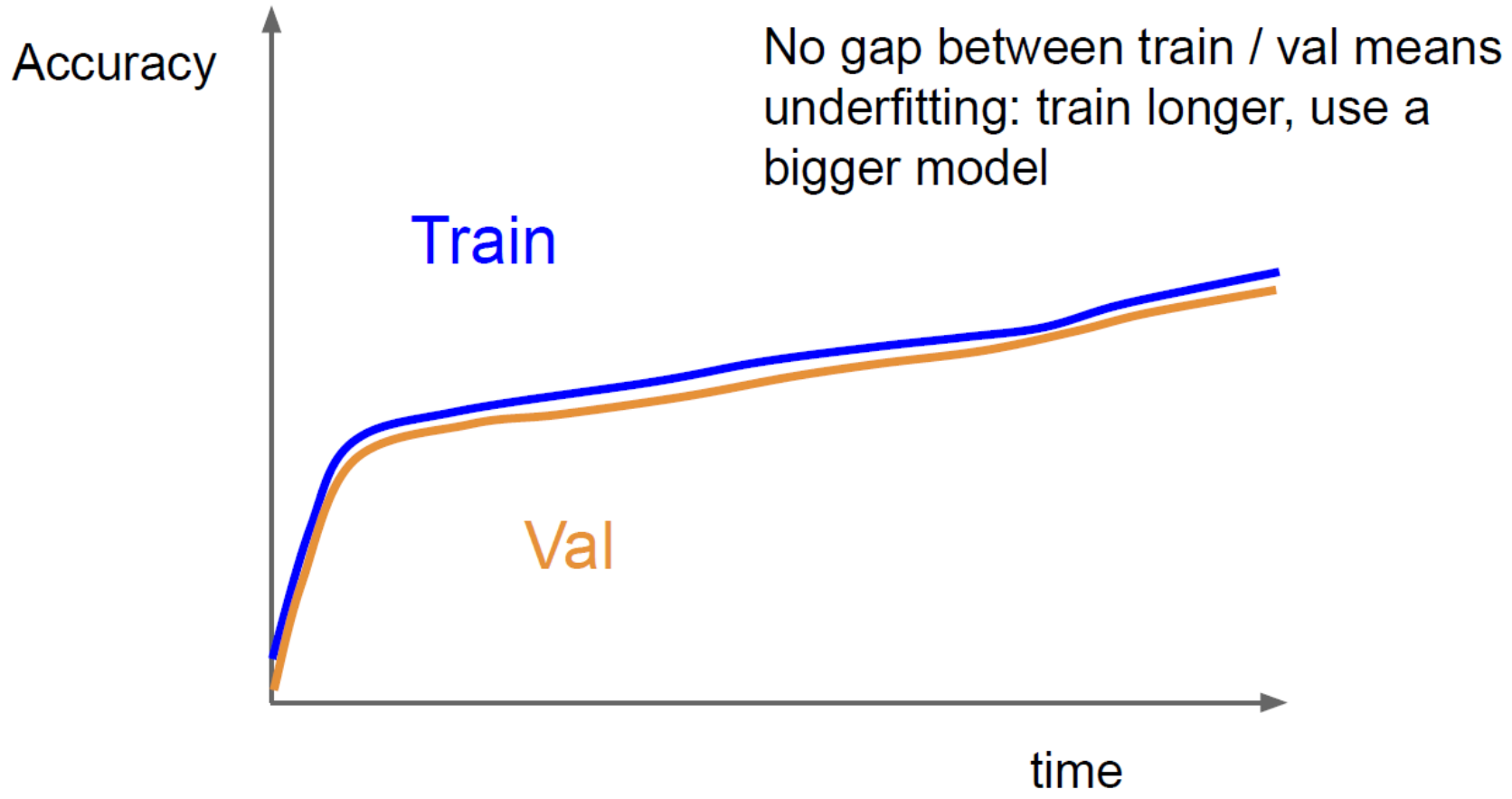
Train vs validation curves



Train vs validation curves

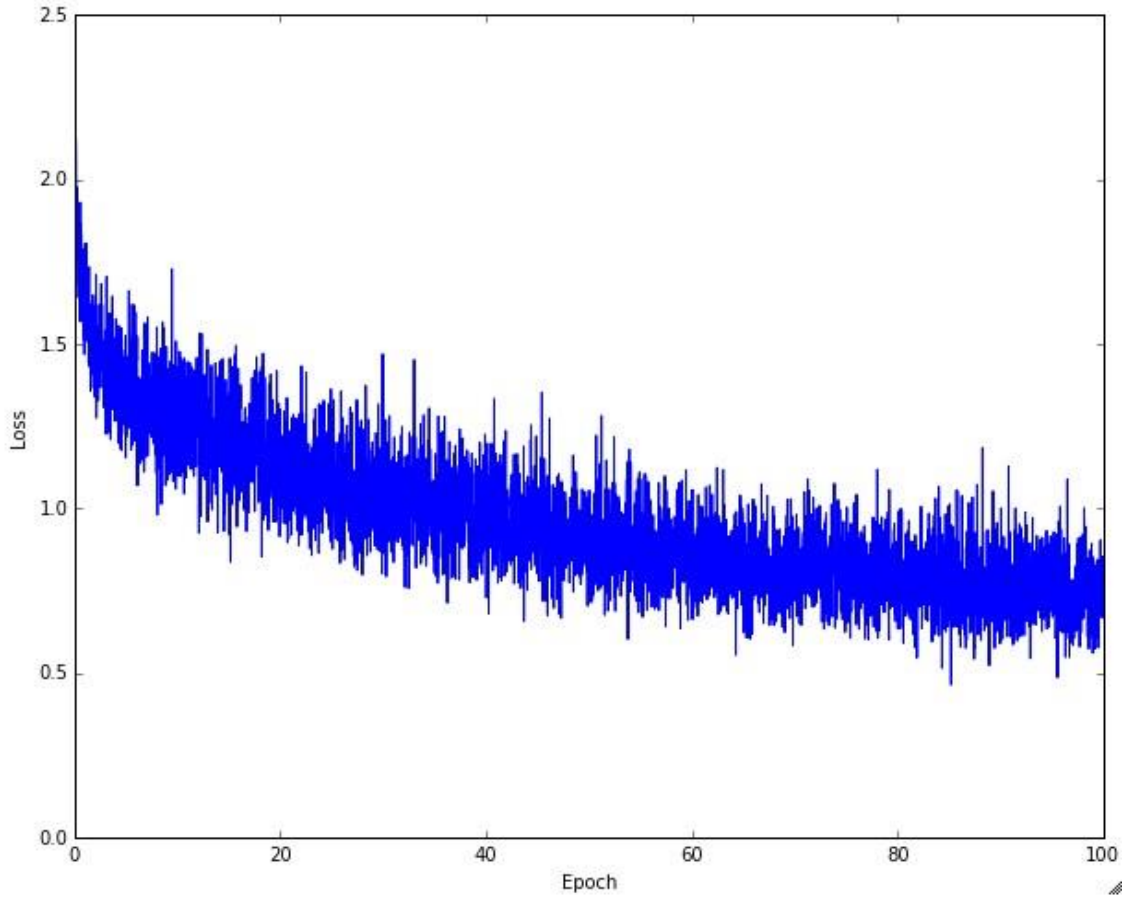


Train vs validation curves



Noisy curves

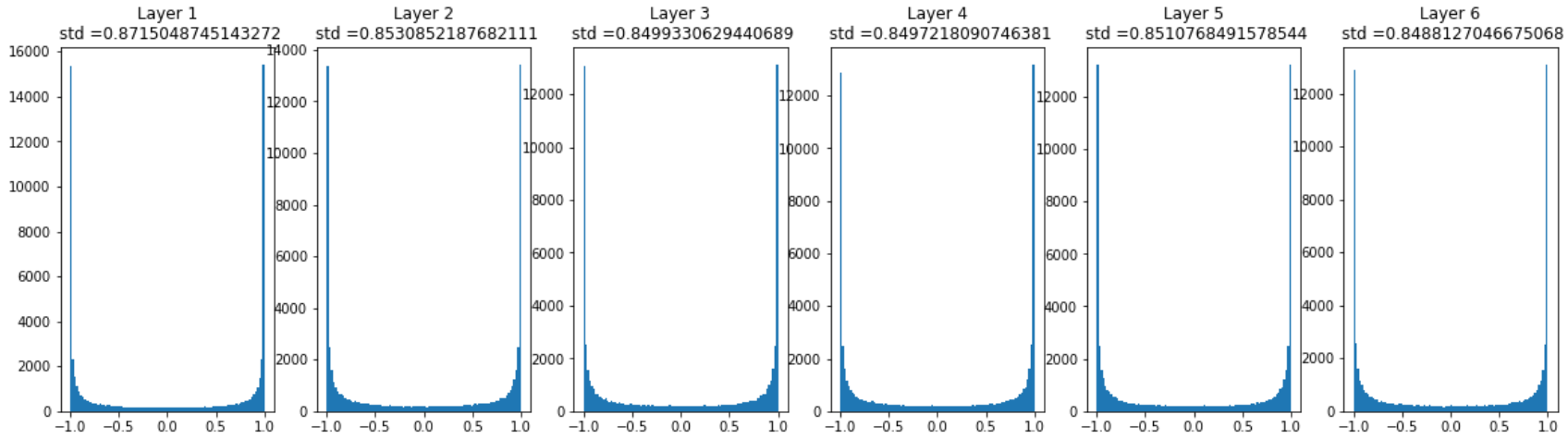
- In practise the curves are always somewhat noisy.
- Look at the overall trend
- This curve looks okay
- If it’s too noisy, try increasing the batch size
- Why? Because the loss is an average over a mini-batch, and if the batch size is small, the average becomes noisy.



Activation / Gradient distributions per layer

- An incorrect initialization can slow down or even completely stall the learning process due to **vanishing gradients**.
- This issue can be diagnosed by plotting activation/gradient histograms for all layers of the network.
- Intuitively, it is not a good sign to see any strange distributions - e.g., with tanh neurons we would like to see a distribution of neuron activations between the full range of [-1,1], instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

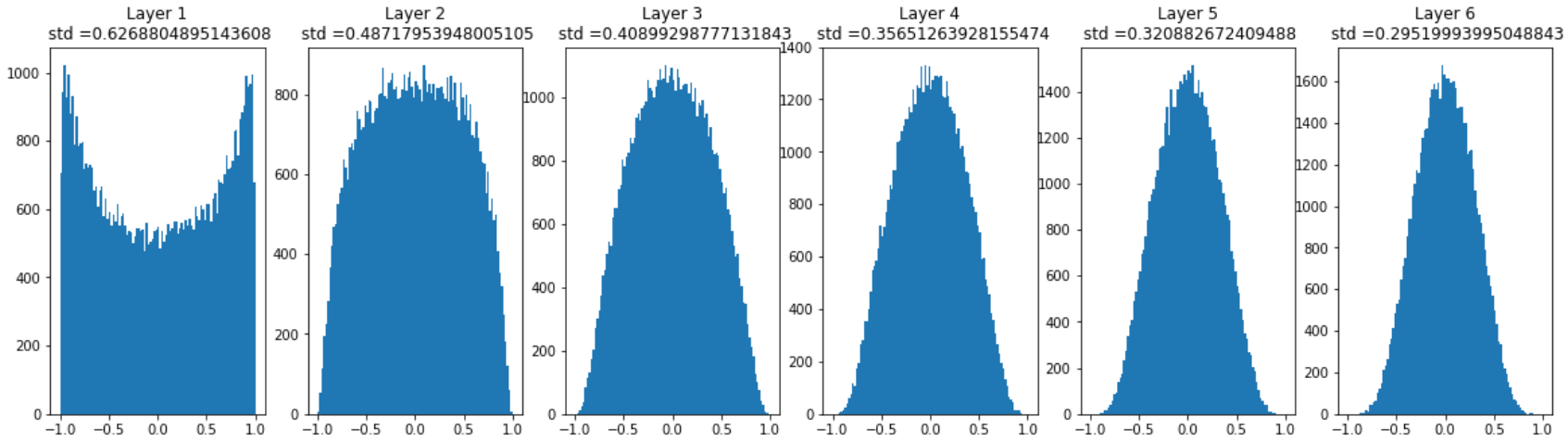
Not ideal



Activation / Gradient distributions per layer

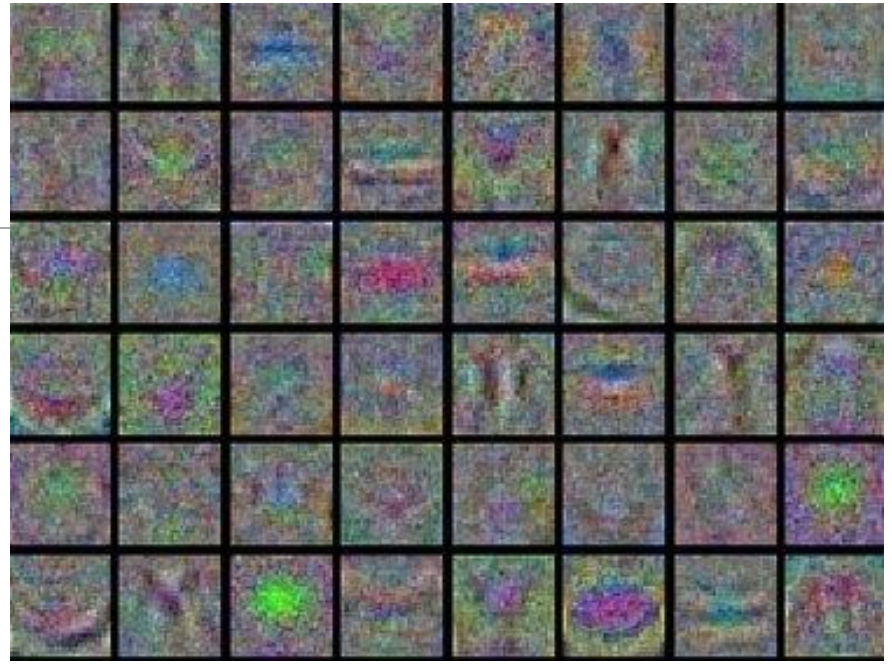
- An incorrect initialization can slow down or even completely stall the learning process due to **vanishing gradients**.
- This issue can be diagnosed by plotting activation/gradient histograms for all layers of the network.
- Intuitively, it is not a good sign to see any strange distributions - e.g., with tanh neurons we would like to see a distribution of neuron activations between the full range of [-1,1], instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

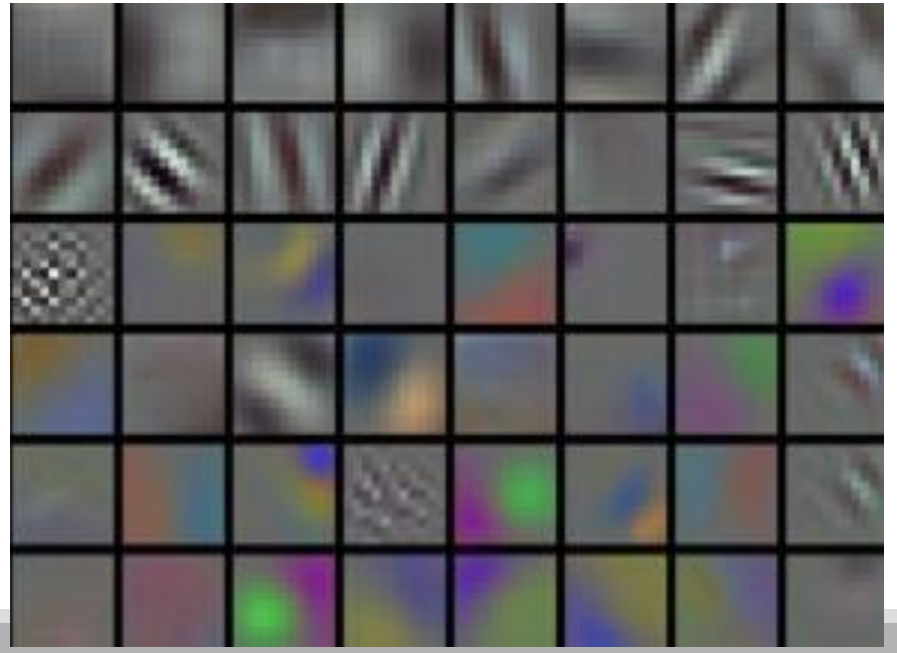
Much better



First-layer visualizations

- Lastly, when one is working with image pixels it can be helpful and satisfying to plot the first-layer features visually:
- **Top**: Noisy features could indicate
  - un-converged network
  - improperly set learning rate
  - very low weight regularization penalty
- **Bottom:** Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.





Summary

- **Activation functions:** Use ReLU
- **Data preprocessing:** Subtract mean
- **Weight initialization:** Use Xavier/Kaiming
- **Batch normalization:** Use always
- **Optimizer:** Use SGD+Momentum or Adam
- **Regularization:** Early stopping and data augmentation almost always a good idea
- **Hyperparameter search:** Course-to-fine search and monitor loss curves
- **Transfer learning (next lecture):** Almost always a good idea

Further reading + online videos/demos

- Optimization:
  - https://medium.com/analytics-vidhya/optimization-algorithms-for-deep-learning-1f1a2bd4c46b
  - http://ruder.io/optimizing-gradient-descent/
  - https://lilianweng.github.io/lil-log/2019/03/14/are-deep-neural-networks-dramatically-overfitted.html
  - https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2
- Stanford CS231:
  - http://cs231n.github.io/neural-networks-2/
  - http://cs231n.github.io/neural-networks-3/
  - http://cs231n.github.io/transfer-learning/