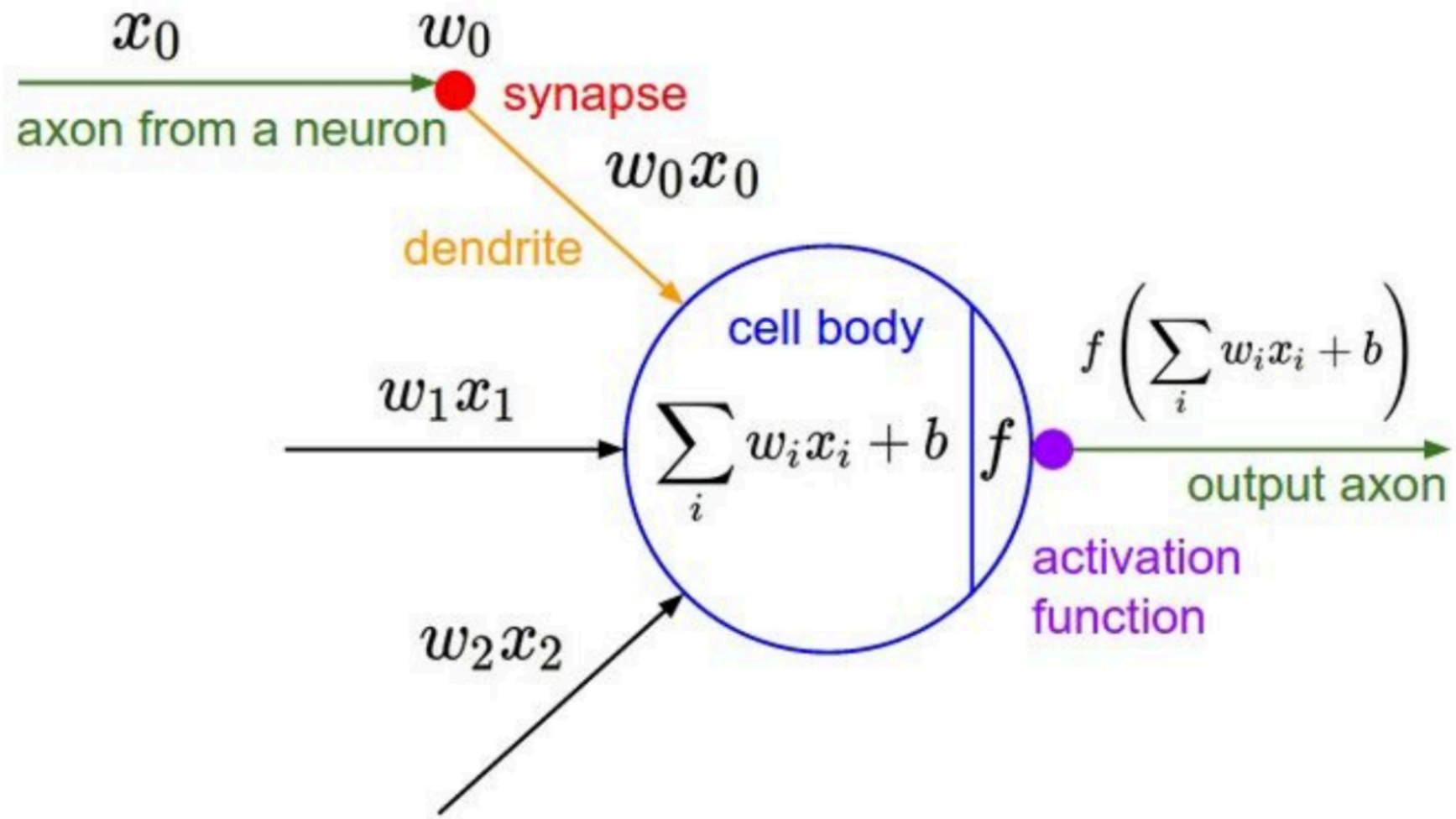
Training Neural Networks

RRC Summer School

Abhinav Gupta

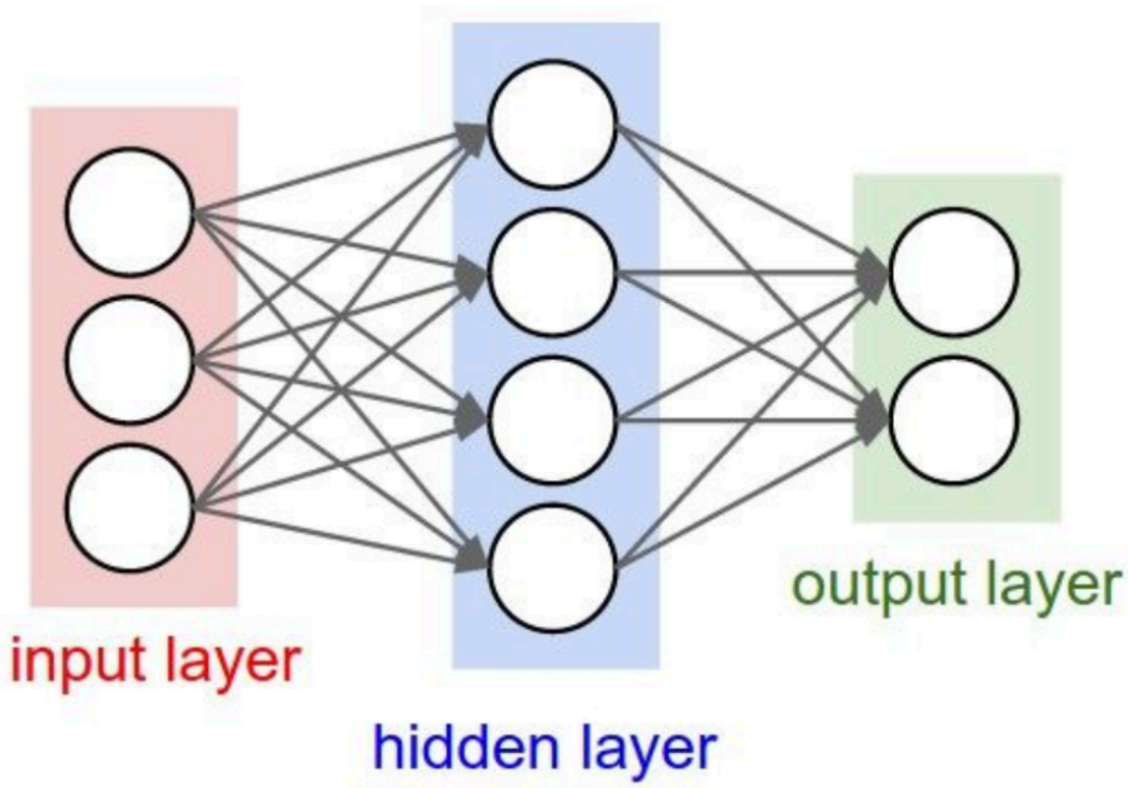
Adapted from Stanford's CS231 course, all images belong to them.

Just a quick recap…



A neural network is a computational graph, with many linear layers and non linearities in between.

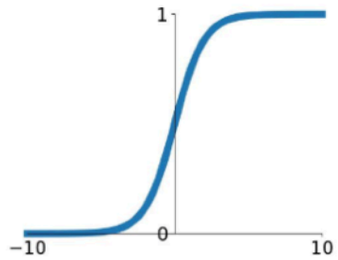
We wanna learn the values of these weights, and we do so through optimisation.



Activation Functions

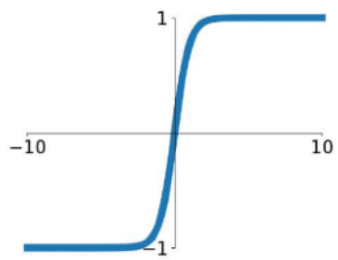
Sigmoid

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



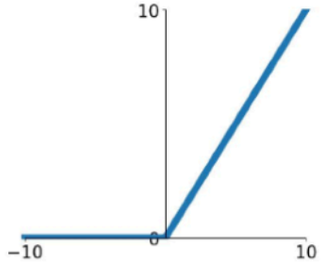
tanh

$$\tanh(x)$$



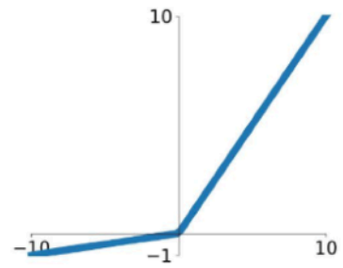
ReLU

$$\max(0, x)$$



Leaky ReLU

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

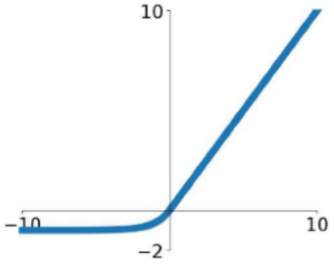


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU / SELU
  - To squeeze out some marginal gains
- Don’t use sigmoid or tanh

The Basic Pipeline

- We sample out a batch of the data.
- We forward propagate it.
- We compute the loss at the end, by comparing prediction and ground truth.
- We back-propagate it and calculate the gradients on the way back.
- We update the parameters/weights using these gradients, by taking steps in the direction of the negative gradient.

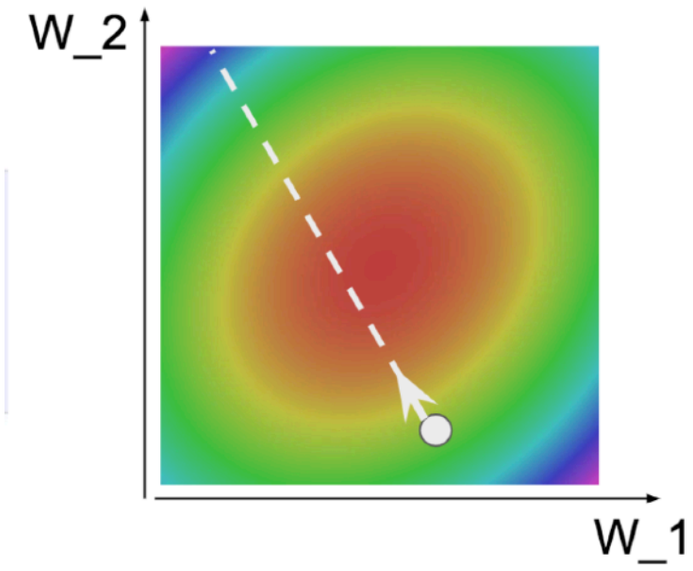
Optimisation Algorithms

- The core strategy in training Neural Networks in solving an optimisation problem! We have some sort of a loss function, that tells us how good our weights are. We need to optimise over the weights and get to the most minimum loss possible.

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent
A simple optimisation algorithm

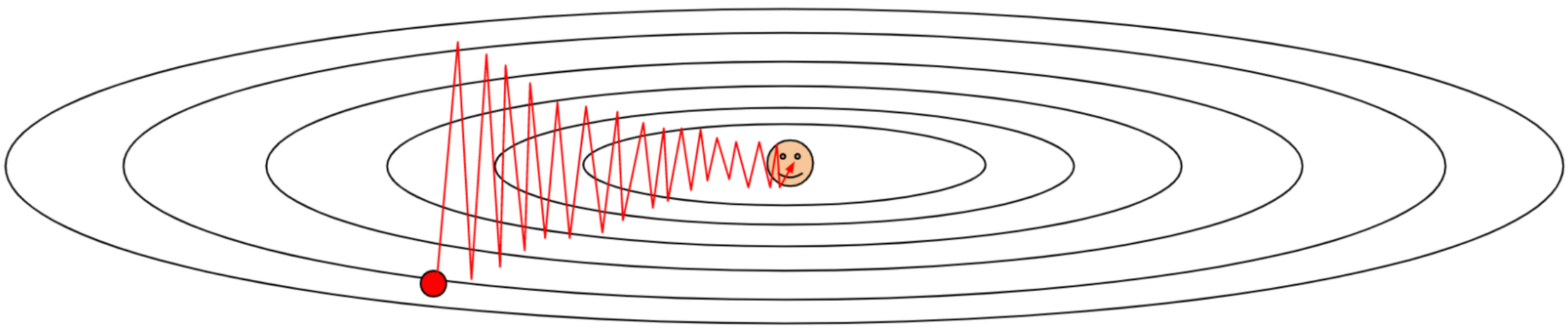


Problems with SGD

- What if the loss is sensitive in one direction and not in the other? That means, it changes quickly in w1 but slowly in w2.

- Very slow progress along horizontal dimension (the slow one). When we calculate the gradients and update, we kinda zig zag back and forth. Nasty progress along the fast changing one.

- This gets worse when we have more dimensions - which means more directions along which it can move.

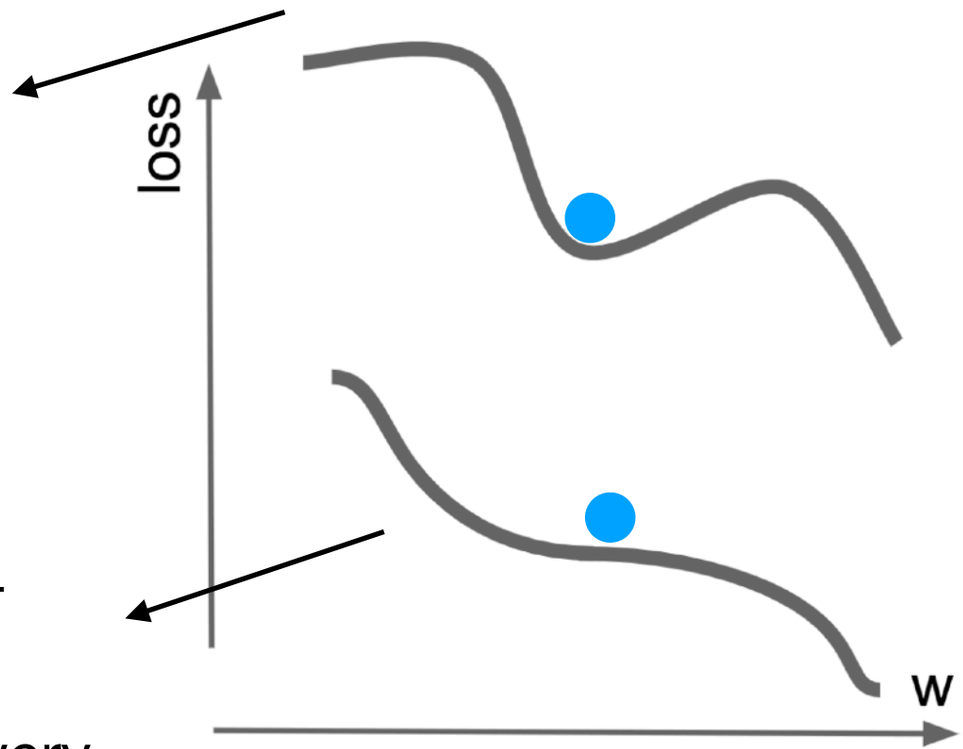


- If there a hundreds of different directions to move in, and the ratio of the largest and smallest one is quite large, then SGD wouldn’t perform well.

- Another problem is that of local minima or saddle point.

In this situation, SGD will get stuck!
Because at the local minima, the gradient is zero as it’s locally flat.
In SGD, we compute the gradient and we move in the negative direction. But since we are at a local minima, we’ll get stuck at this point.

Another Problem: In one direction we go up,
And in another we go down - SADDLE POINT
Then also it may get stuck.
The gradient is exactly zero, but the slope is very small. So if the gradient is very small,
we make REALLY SLOW progress



One dimensional weight

‘Stochastic’

- Loss function is computed by finding the loss for many different examples. But there could be too many, so we estimate the loss and gradient using a mini batch of training examples.

- So if there’s noise in the gradient estimates, then vanilla SGD might just wander around a bit before actually going towards the minima.

- What if we use just normal GD, that is, full-batch? Doesn’t really solve these problems, so we need a fancier optimisation problem.

SGD + Momentum

- We maintain a ‘velocity’ over time. We add our gradient estimates to the velocity. And we step in the direction of the velocity rather than the gradient.

- And a new ‘friction’ hyper-parameter which is used to decay the velocity at every time step.

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

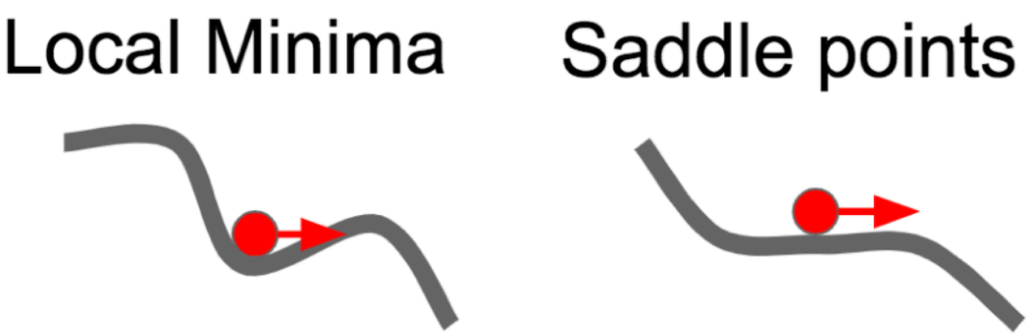
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

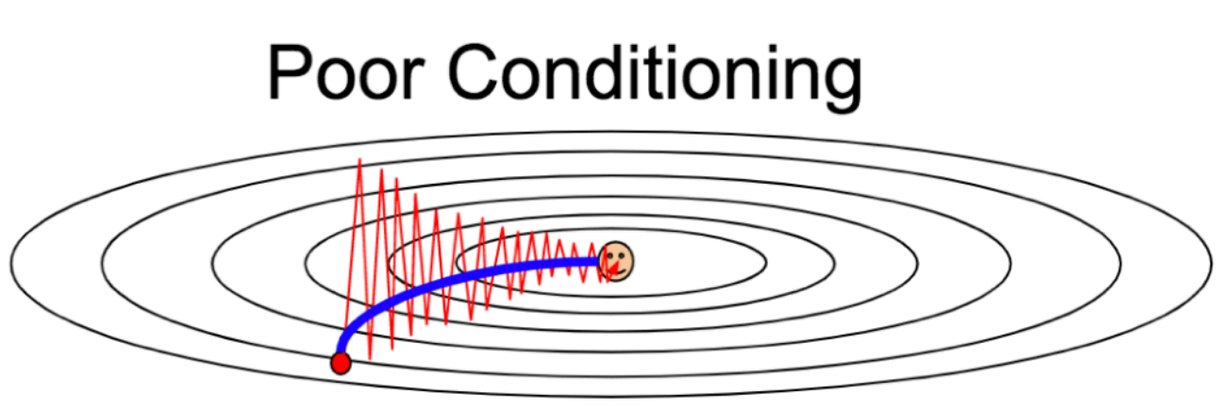
This solves all the problems

At local minima, the point may not have any gradient, but i**t'll still have a 'velocity'.**

So we can get over this local minima and still continue to let the boll roll.



At saddle points, even though the gradient is really less, we have built up a decent amount of velocity as we rolled downhill.



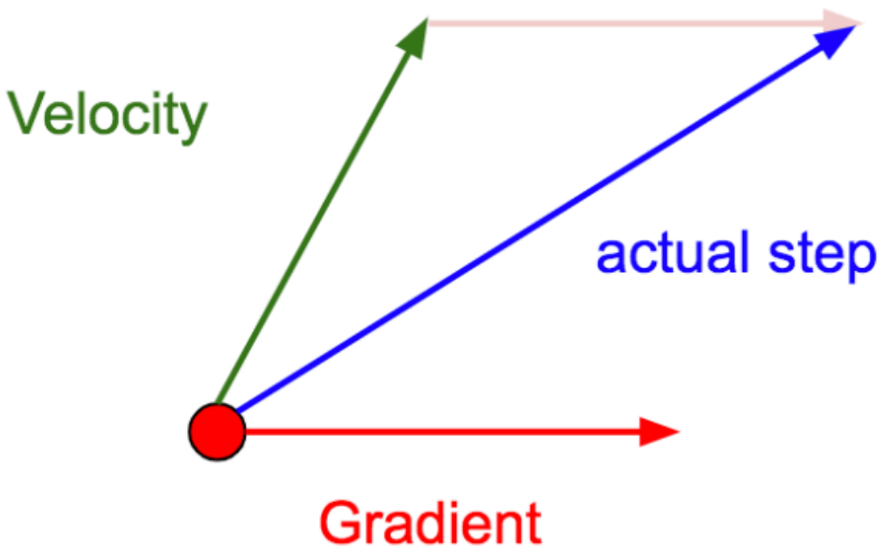
For the 'zig-zag' problem, the zig zags would cancel each other out while using momentum, reducing the amount by which we step in the sensitive direction, and the velocity will keep building up in the horizontal direction and accelerate across the less sensitive direction.

Momentum Updates

- We compute the weighted average of the gradient estimate and the velocity vector and we step across this ‘actual step’.

- So we’re taking a mix of the gradient and the velocity to overcome the noise and all the difficulties discussed.

- That’s why, SGD + Momentum



AdaGrad: Another optimisation strategy

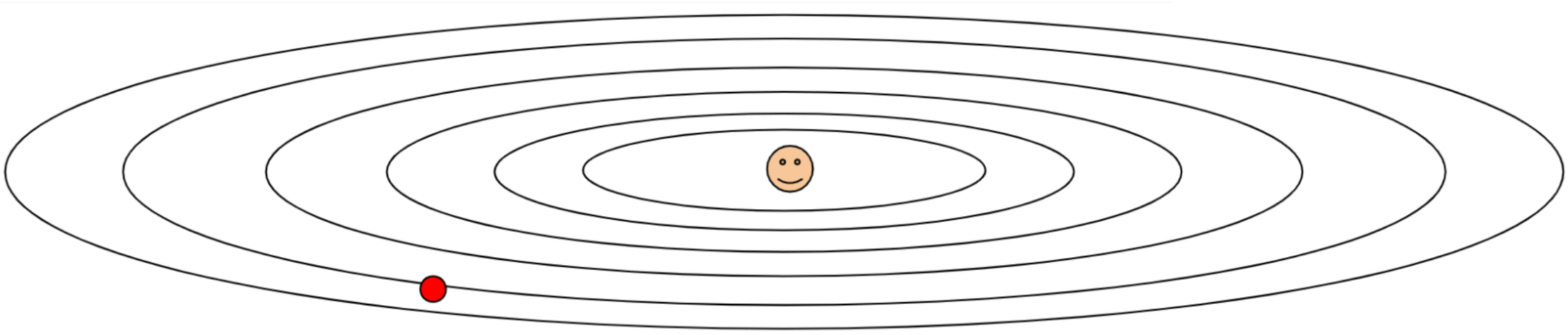
- AdaGrad keeps a running sum estimate of the squared gradients. So instead of velocity, we now have a ‘grad_squared’ term.

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

(So that we don’t divide by zero…)

How it helps!

- The idea is that, if there's a high gradient along one direction and low gradient along the other, then if add the sum of squares of the small gradient, we divide by a smaller number and hence, we'll accelerate movement along the slower dimension.
- And along the other dimension where the gradients tend to be very large, we divide by a larger number, thereby slowing down the progress along that wiggling dimension.



What happens over the course of training?

- In AdaGrad, progress along ‘steep’ directions is damped and progress along ‘flat’ directions is accelerated.

- But over time, the step size gets smaller and smaller. It’s good in the convex case because as you approach the minima, you want to slow down and converge.

- But in non convex case, if you come across a saddle point, you get stuck and can’t make any progress.

RMSProp: Leaky AdaGrad

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

We still keep the estimates of the squared gradients. But instead of just letting it accumulate over training, we let it decay over time - just like the momentum update we had seen earlier.
So it’s like a momentum update over squared gradients!
And we don’t slow down where don’t want to, as the estimates are leaky.

Adam

- The core idea behind momentum was to move in the direction of the velocity to not get stuck in local minima.

- The core idea behind AdaGrad and RMSProp was to compute a square of the gradients to slow down movement along the sensitive direction and prevent the zig zags.

- The core idea behind Adam: Why not combine these two?!

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

First moment: the velocity term
Second moment: the square gradients term

But there's a slight problem here…what happens at the first time-step?

- The second moment is initialised to zero. And the decay rate is very very low, like 0.9 or something (beta2). So after one update, second moment is still very close to zero.

- When we make the update step, we divide by a very small number and hence, make a huge step in the beginning, which is because we initialised it to zero.

- Of course, since the first moment is also very small, it may cancel out the second moment, but sometimes, it does result in taking very large steps in the beginning.

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

We add a ‘bias correction’ step

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

We create an unbiased estimate of the first and second moments using the current tilmestep ’t’. We make our steps using these unbiased estimates

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3 or 5e-4
is a great starting point for many models!

Check out the lecture video at 43:05 - shows how well Adam works: https://www.youtube.com/watch?v=_JB0AO7QxSA

Recap: Optimisation Algorithms

- Stochastic Gradient Descent (The classic one)
- SGD + Momentum
- AdaGrad
- RMSProp
- Adam (The Best One)

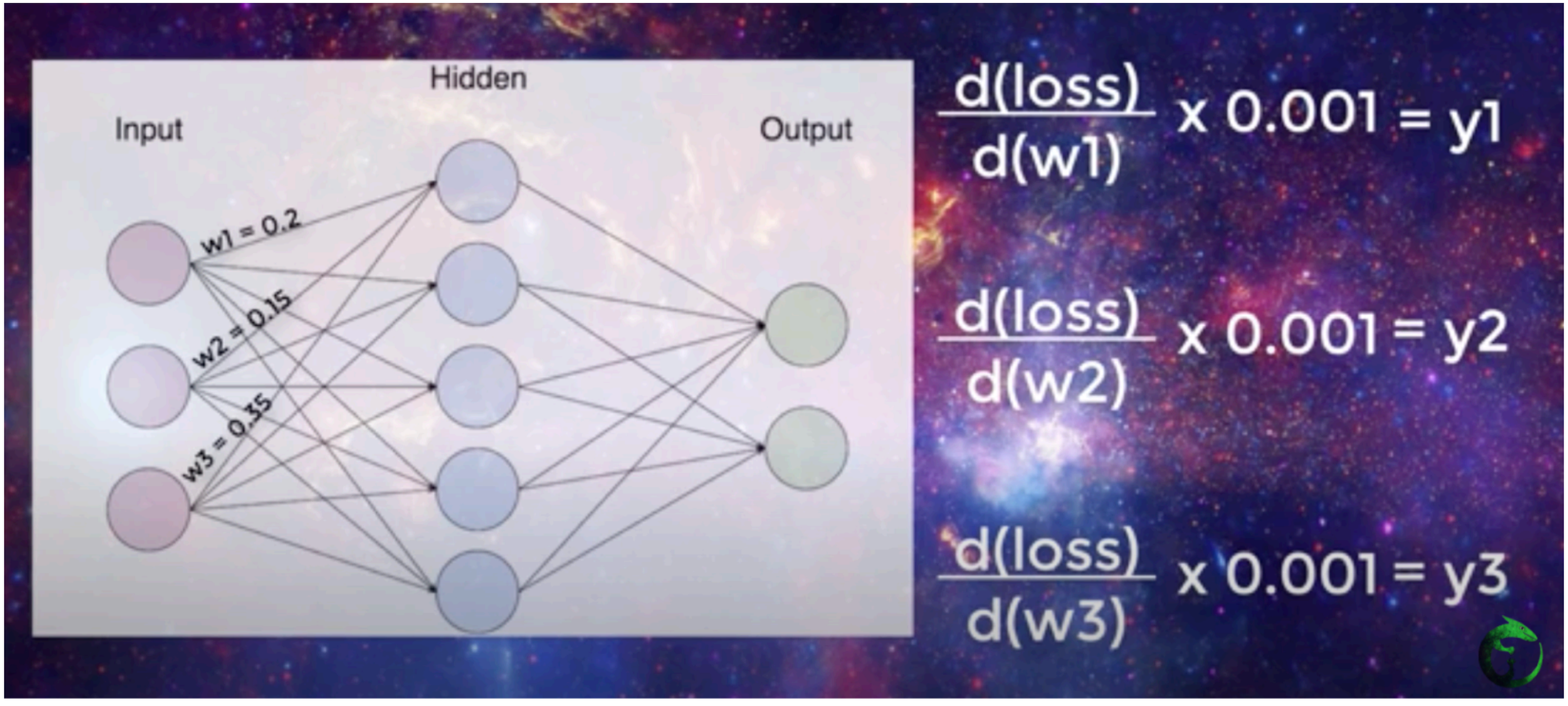
Learning Rate or Step Size

- A number by which we multiply our resulting gradient by!
- The objective during training is for SGD or Adam to minimise the loss between the actual output and predicted output.
- We start with an initialisation of the weights and keep updating them with every iteration to move towards the minimum loss.
- The size of the steps we take to reach this minimised loss is going to depend on the learning rate.

- After calculating loss for the given inputs, the gradient of that loss is calculated with respect to each of the weights in the model.

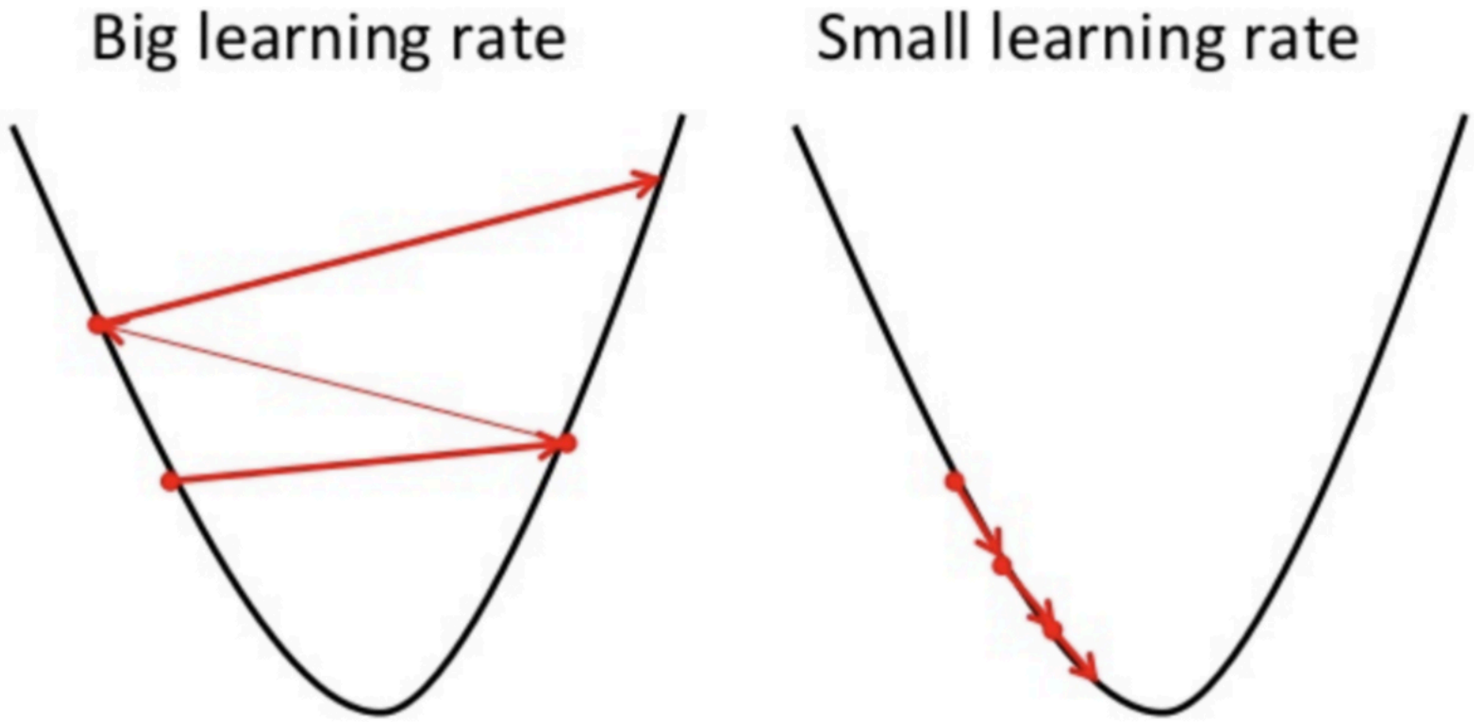
- These gradients will be multiplied by the learning rate (somewhere between 0.1 and 0.001).

- We then update the weights by subtracting these values from them (to move in the negative direction).



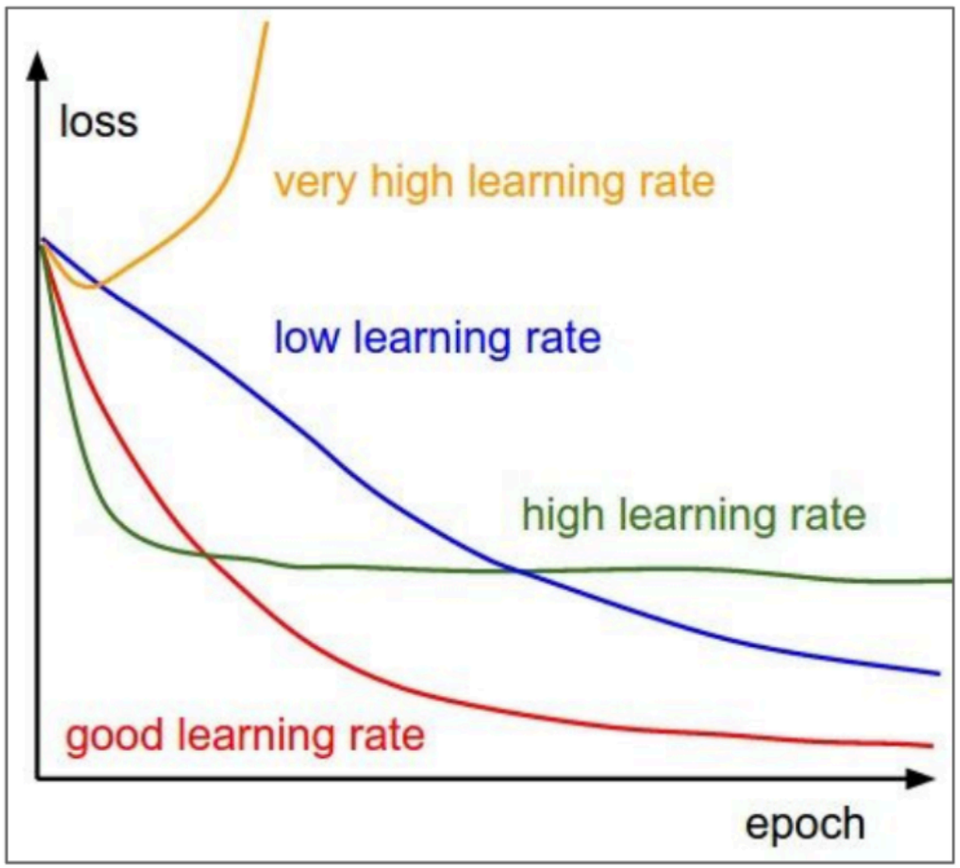
- If learning rate is too high? OVERSHOOTING, that is we take a step that's too large in the direction of the loss. So we shoot past the minimum and miss it.

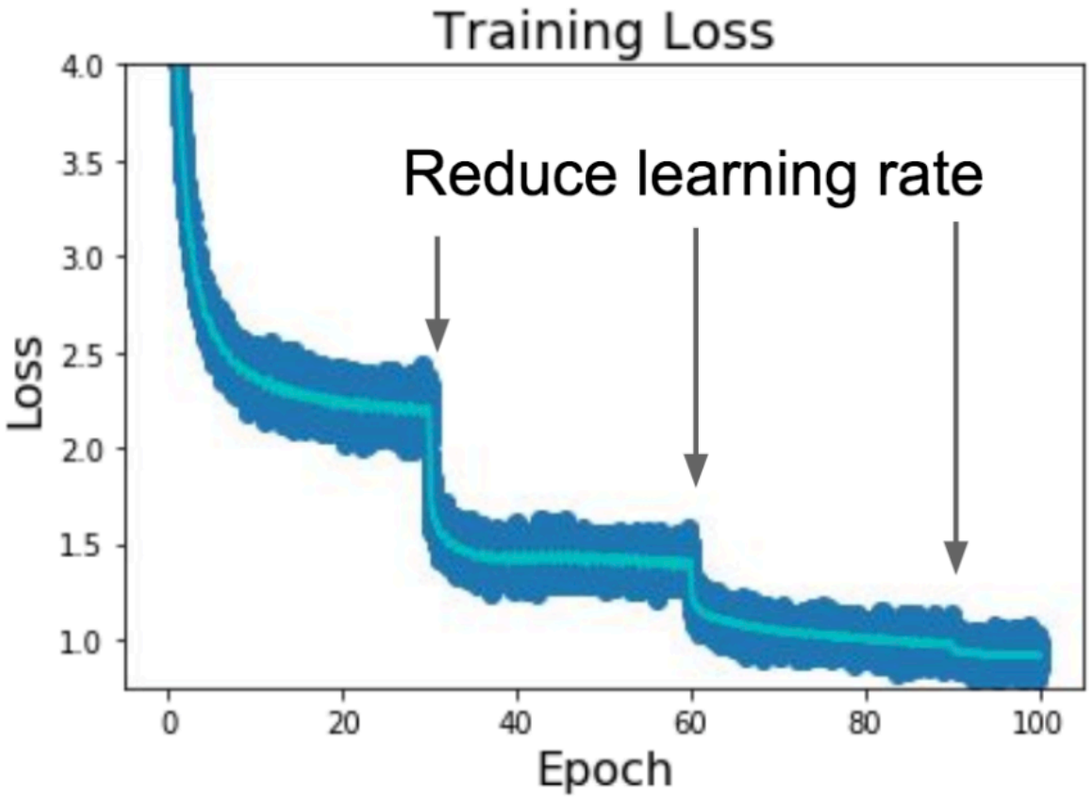
- If small learning rate, then we take very small baby steps.



Effect of learning rate on gradient descent

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



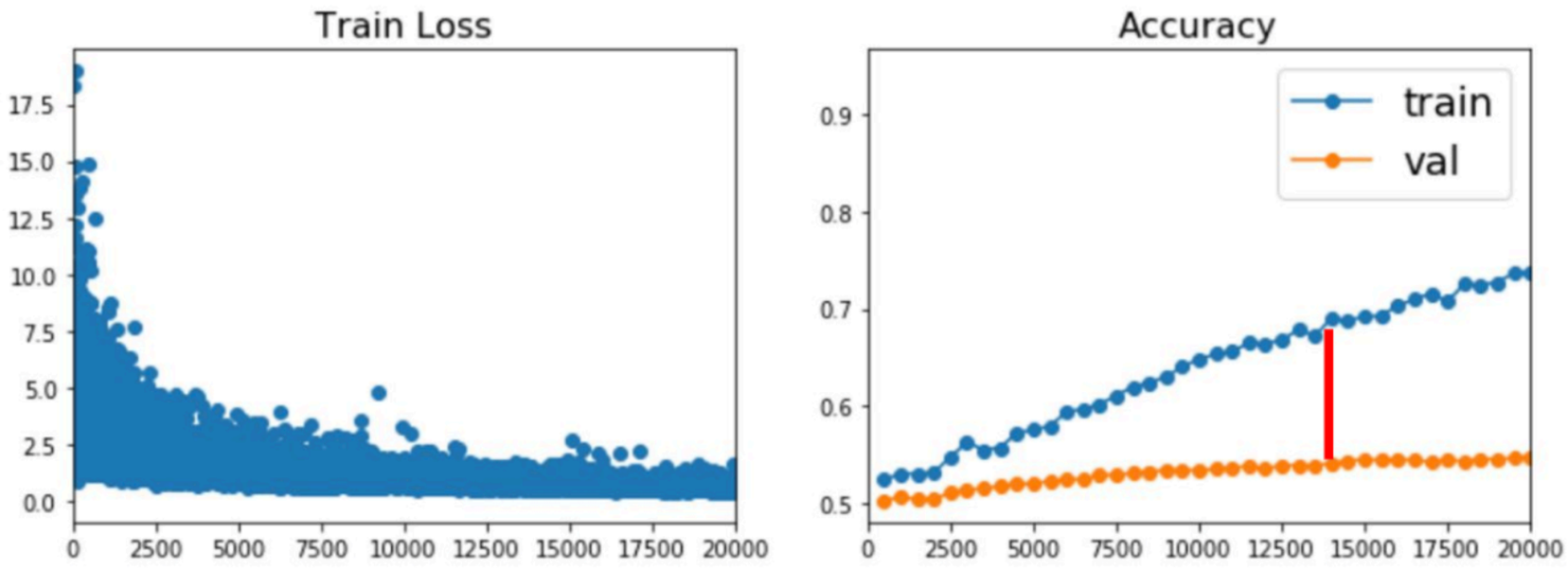


Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

‘Step Decay’ - we reduce the learning rate at particular steps, so if it would’ve gotten stuck somewhere, we can take smaller baby steps and maybe reach a better region.

Beyond Training Error

- So far, we've been trying to reduce the train error, the one between predicted output and actual output of the network.
- But what we care about is the performance on unseen data!

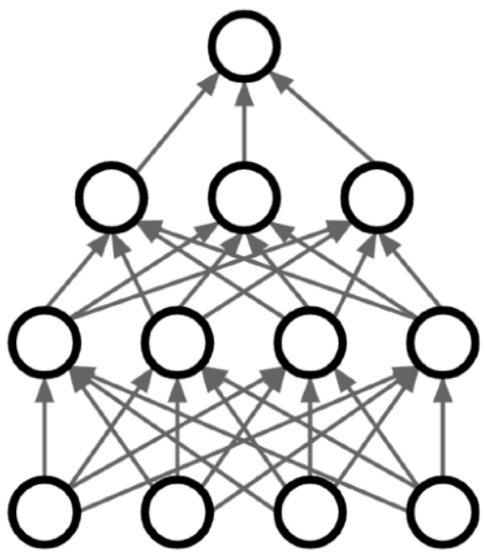


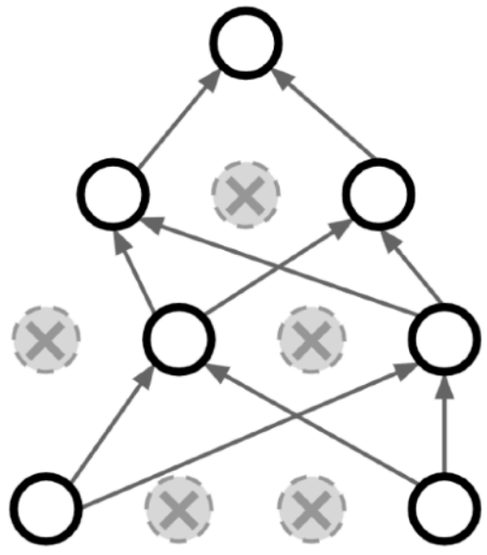
Regularisation: Dropout

- We add something to our model to prevent it from fitting our training model too well, so it could perform better on unseen data!

- In dropout, we randomly set some neurons to zero for each forward pass.

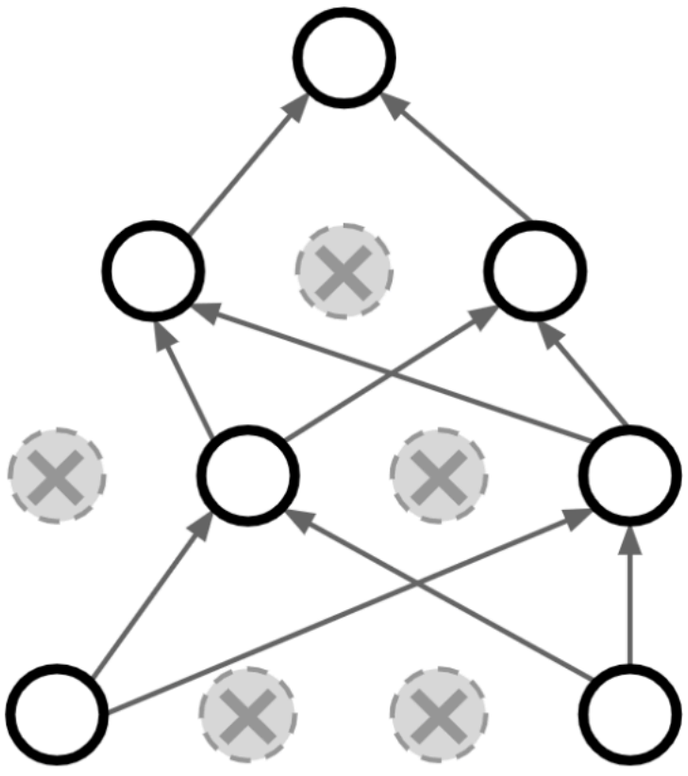
- Dropout is like training a large ensemble of models!



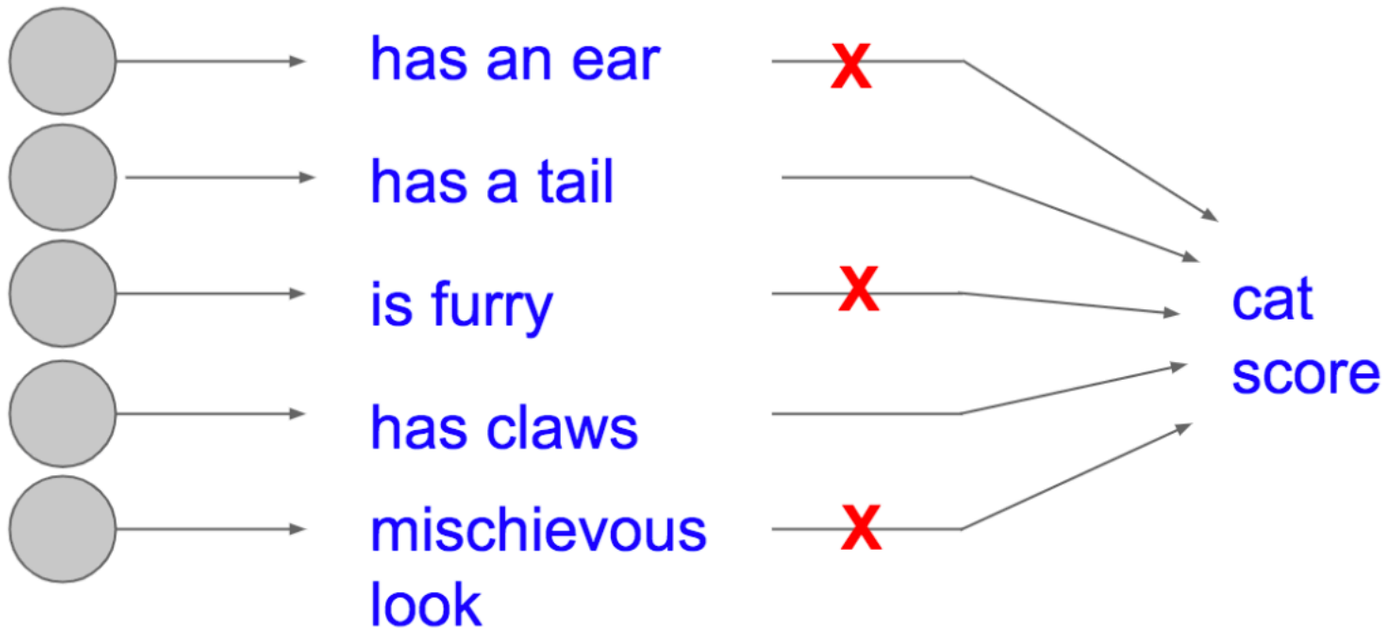


Regularization: Dropout

How can this possibly be a good idea?

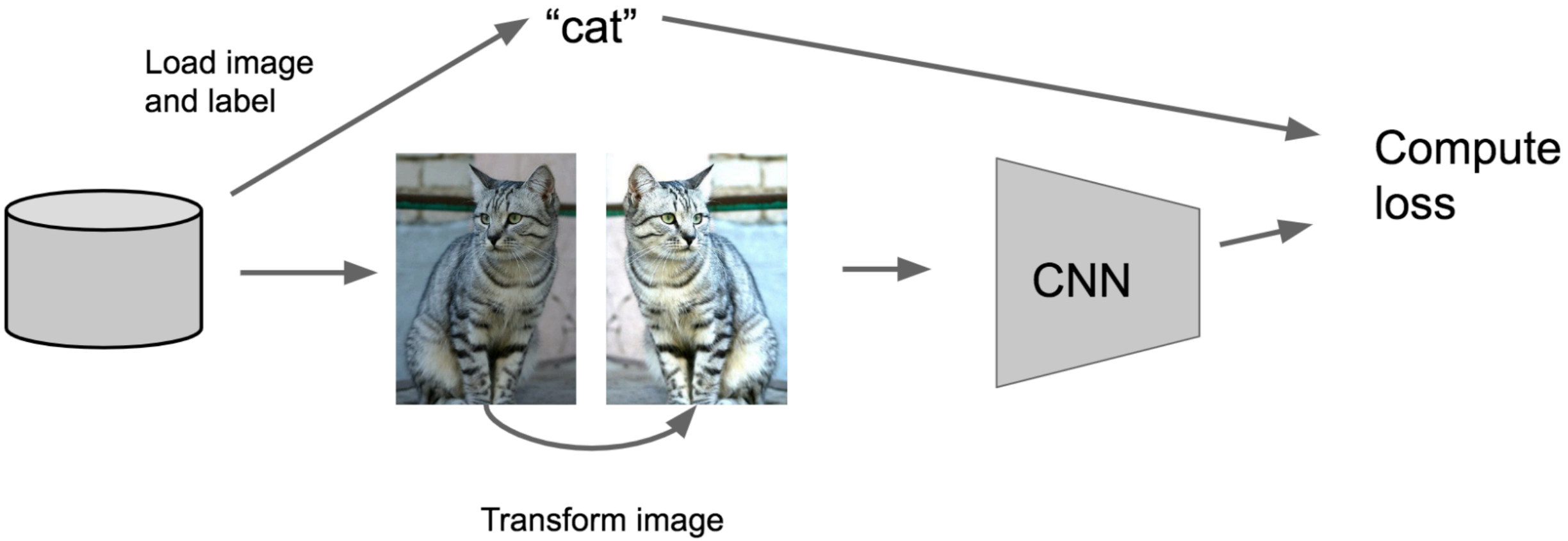


Forces the network to have a redundant representation;
Prevents co-adaptation of features

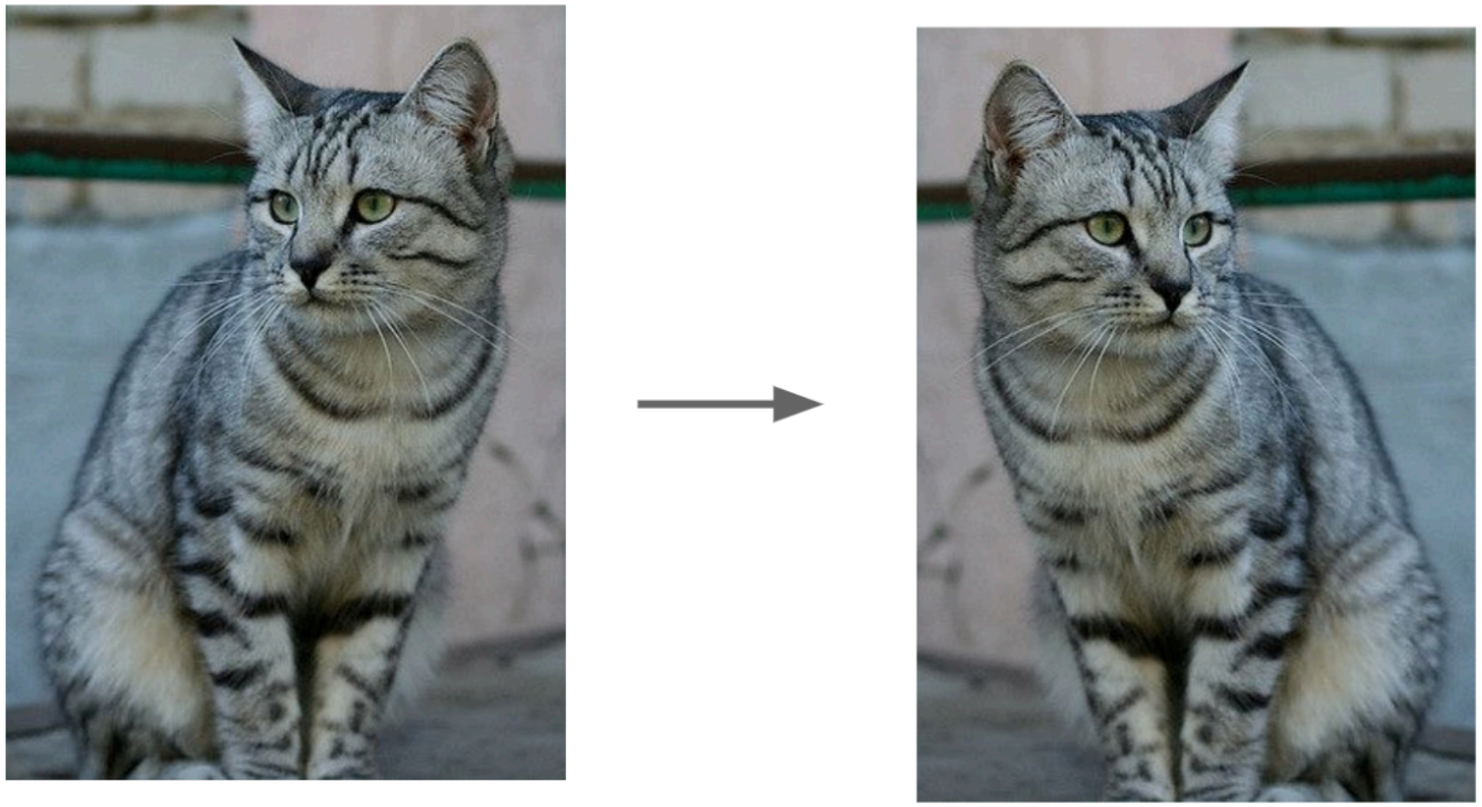


We want to prevent over fitting and co adaptation of all the features. So even if the cat doesn't have a tail or isn't furry, the model should be able to detect the cat-ness. This helps increase performance for unseen data.

Regularisation: Data Augmentation



Horizontal Flip



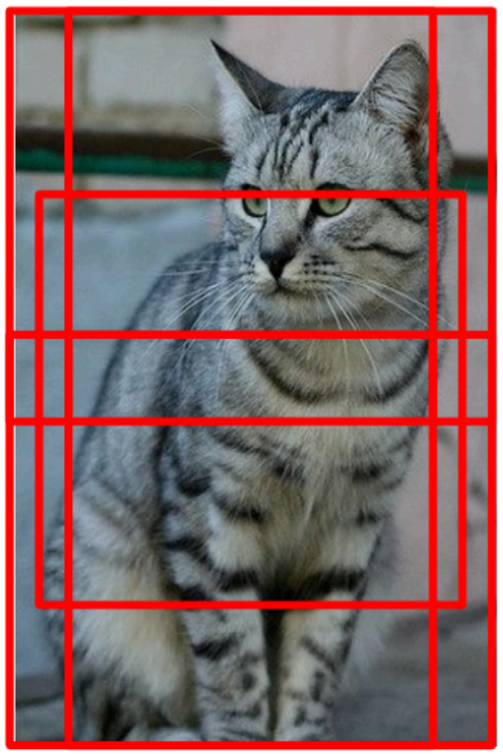
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



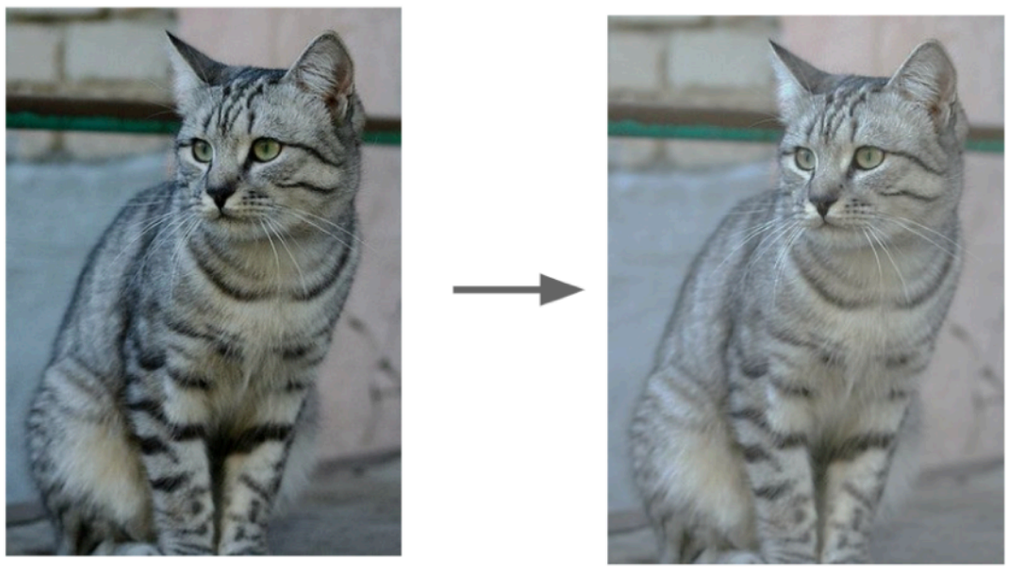
Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Color Jitter

Simple: Randomize
contrast and brightness



Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

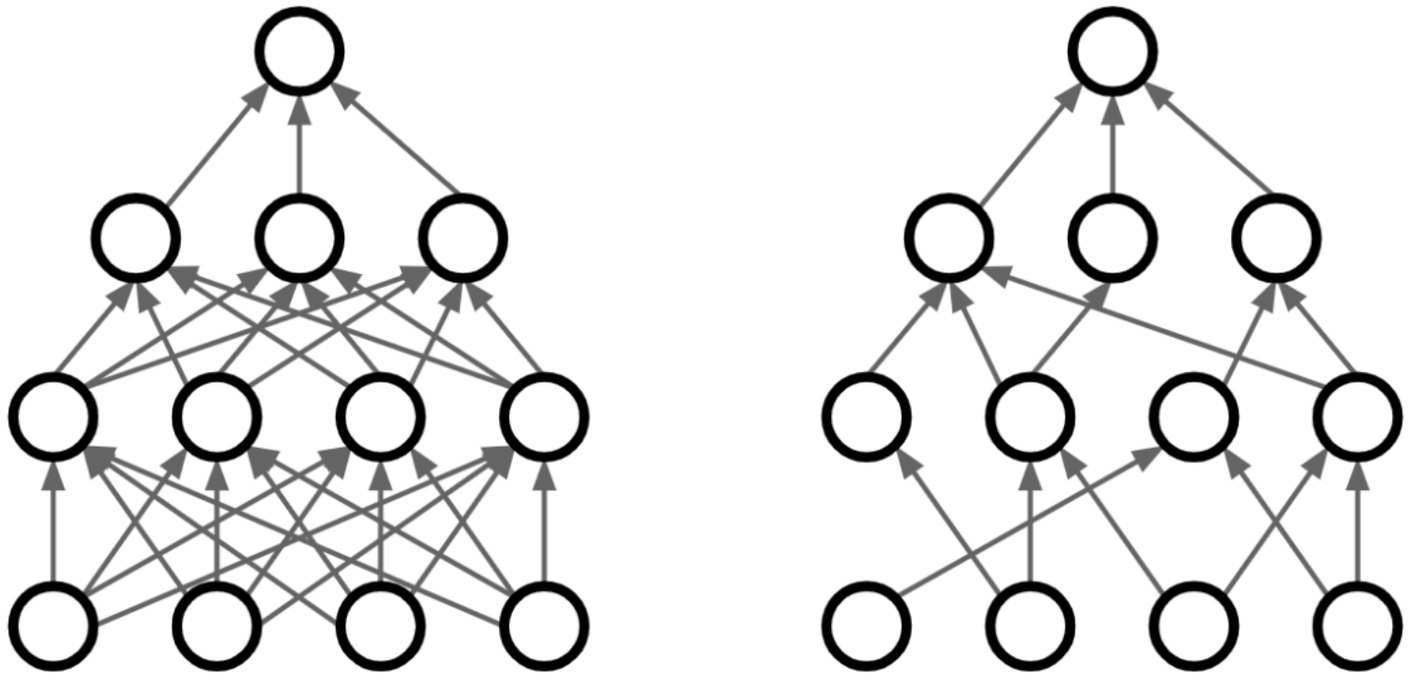
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

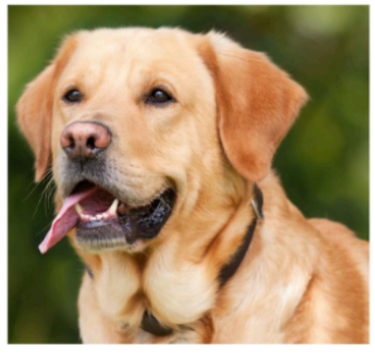
Regularization: Mixup

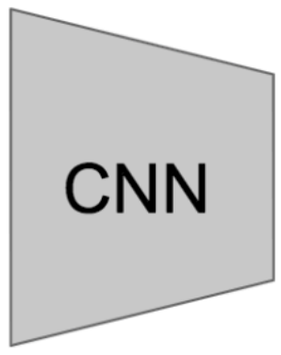
Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup





Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

Transfer Learning

- You need a lot of data if you wanna train CNN's. If you have a really small dataset, then data augmentation helps to increase the amount of data you have.

- But we can also accomplish this without having to do data augmentation, by Transfer Learning.

- As the name suggests, you 'transfer' the weights the network learns and learn some more with the small dataset. That is, you fine tune the network for the new dataset, keeping the weights of the previous one.

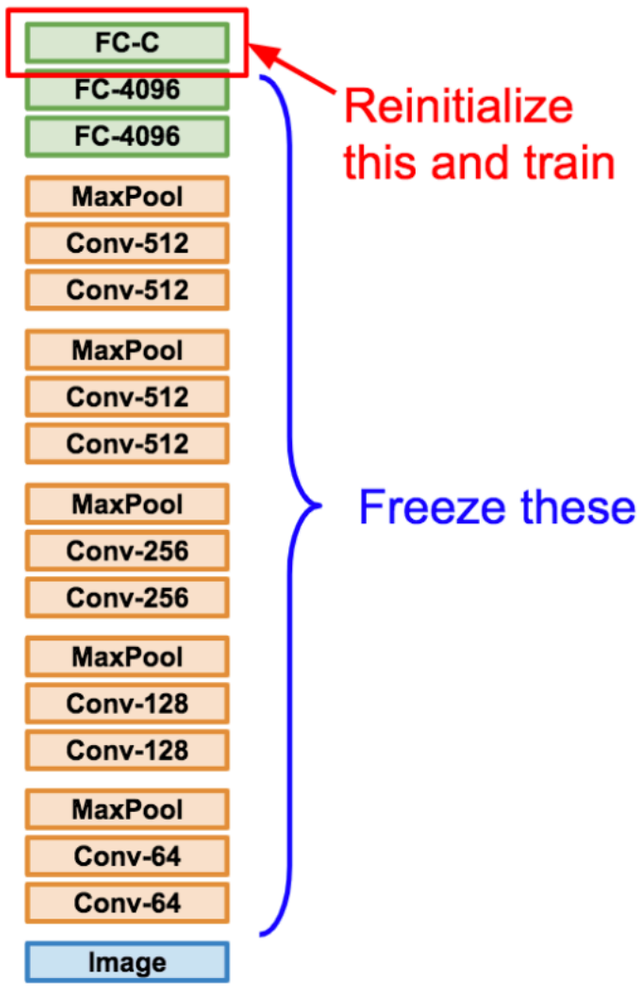
- Very common nowadays, even Fast RCNN uses a CNN that is pertained on ImageNet.

Transfer Learning with CNNs

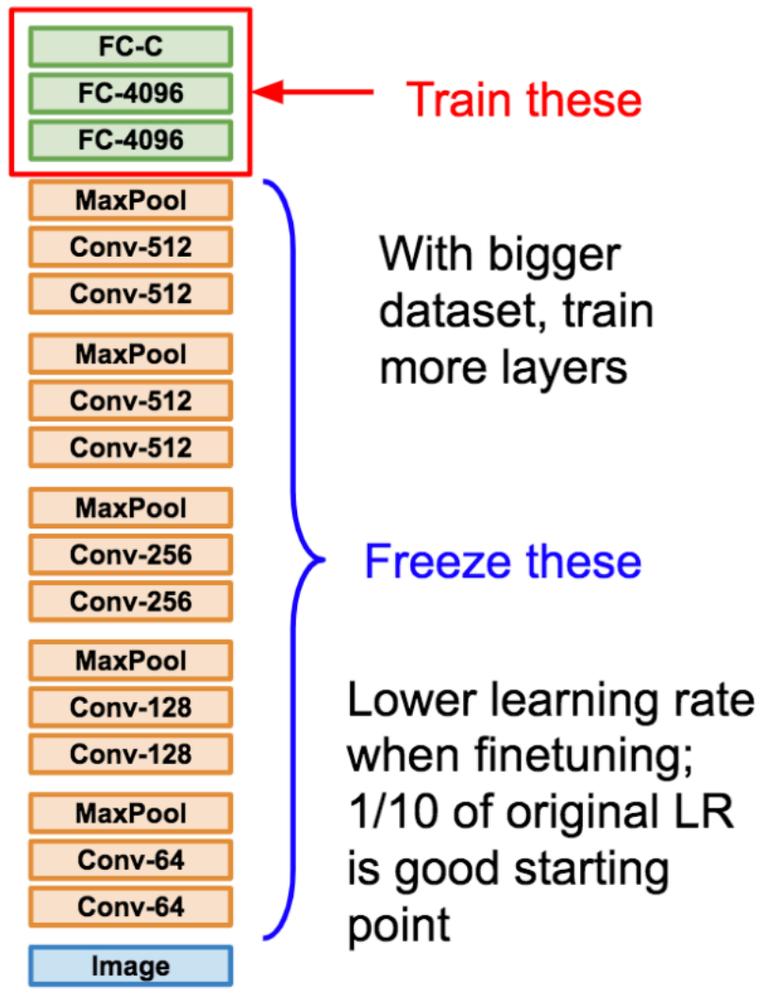
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

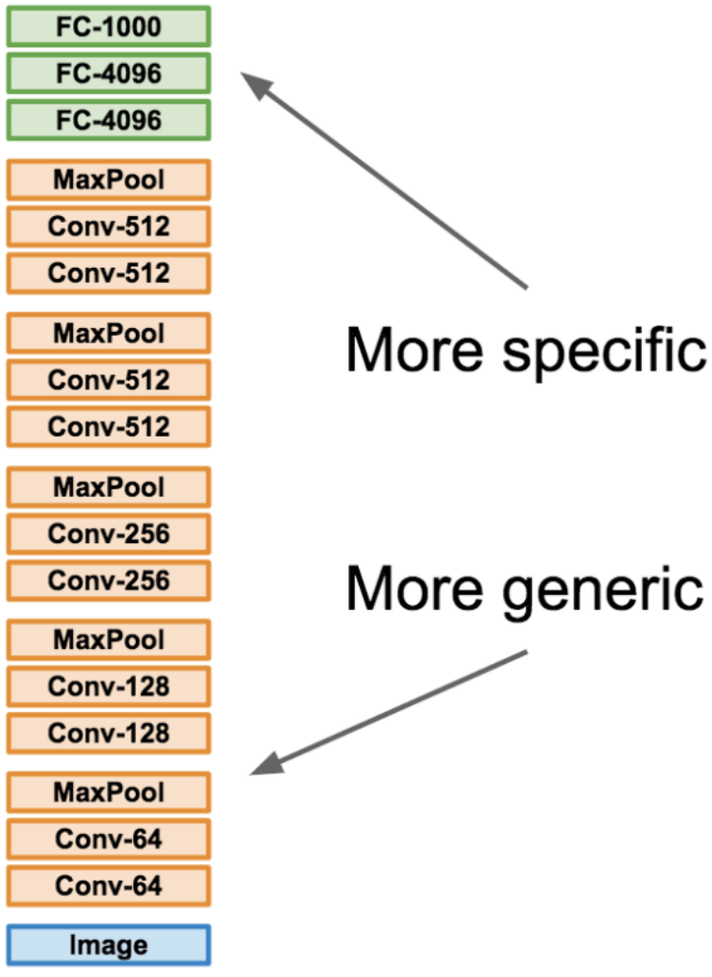
1. Train on Imagenet

2. Small Dataset (C classes)



3. Bigger dataset





	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers