⑦

Training Neural Network (II)

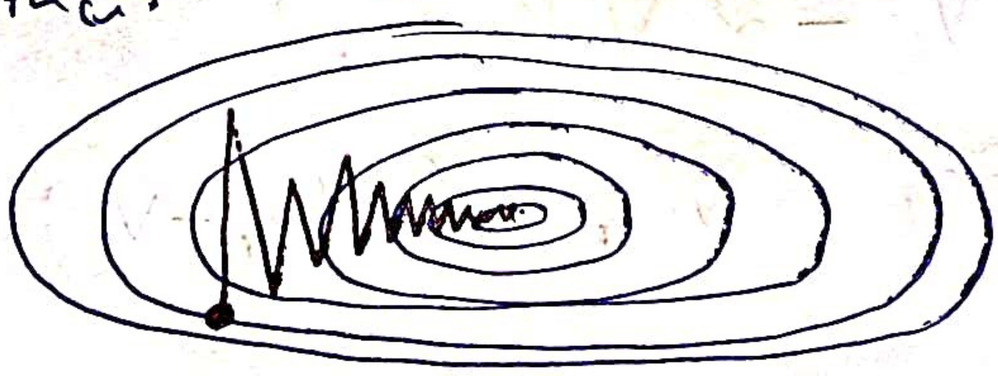
→ Fancier optimization
→ Regularization
→ Transfer Learning

* Optimization

→ Core strategy in training neural network is an optimization Problem.

(#) Problems with SGD

① → If loss changes quickly in one direction and slowly in another.



{ Loss function has high Condition number }

⇒ Very slow progress along shallow dimension, jitter along steep direction.

⇒ This problem is much more common in higher dimention.

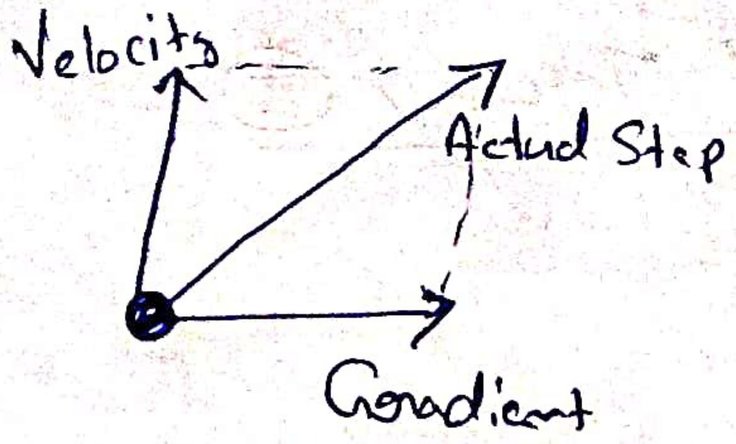
② → If the loss function has a local minima or saddle point.

① SGD + Momentum

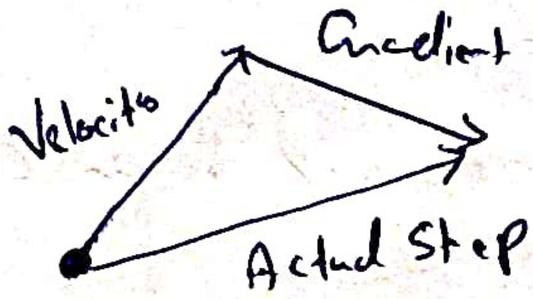
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

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

⇒ This simple strategy solves
- → Local minima
- → Saddle points
- → & poor conditioning problems.



② Nesterov Momentum



$$V_{t+1} = \rho V_t - \alpha \nabla f(x_t + \rho V_t)$$

$$x_{t+1} = x_t + V_{t+1}$$

⇒ Change of variable $\tilde{x}_t = x_t + \rho V_t$ and rearrange:

$$V_{t+1} = \rho V_t - \alpha \nabla f(\tilde{x}_t)$$

$$\tilde{x}_{t+1} = \tilde{x}_t + V_{t+1} + \rho(V_{t+1} - V_t)$$

⇒ Adding momentum has an effect of overshooting the minimum.

③ AdaGrad

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

④ 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 / sqrt(grad_squared + 1e-7)
```

⑤ Adam (almost)

```
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 / sqrt(second_moment + 1e-7)
```

⇒ Sort of like RMSProp with momentum.

⑥ Adam (full form)

first_moment = 0

second_moment = 0

for t in range (num_iterations):

dx = compute_gradient(x)

first_moment = beta1 * first_moment + (1-beta1) * dx

second_moment = beta2 * second_moment + (1-beta2) * dx^2

$$first\_unbias = \frac{first\_moment}{1 - beta1 ** t}$$

$$second\_unbias = \frac{second\_moment}{1 - beta2 ** t}$$

$$x \mathrel{-}= learning\_rate * \frac{first\_unbias}{\sqrt{second\_unbias} + 1e-7}$$

⇒ Bias correction for the fact that first & second moment estimates start at zero.

⇒ Adam with beta1 = 0.9

beta2 = 0.999

learning_rate = 1e-3 or 1e-5

is a great starting point for many models!

★ Learning rate decay over time!

⊛ **Step decay**

⇒ Decay learning rate by half every few epochs.

⊛ **Exponential decay**

$$\alpha = \alpha_0 e^{-kt}$$

⊛ **1/t decay**

$$\alpha = \frac{\alpha_0}{1+kt}$$

⇒ More critical with SGD + Momentum, Less common with Adam.

★ Second-Order Optimization

(1) Use gradient & Hessian to form quadratic approximation

(2) Step to the minima of the approximation.

⇒ Second-order Taylor expansion:

$$J(\theta) \approx J(\theta_0) + (\theta-\theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta-\theta_0)^T H (\theta-\theta_0)$$

⇒ Solving for the critical point we obtain the Newton parameter update:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)$$

⇒ Hessian has $O(N^2)$ elements, inverting takes $O(N^3)$

↳ For deep learning this is not a good idea

∽ N = (Tens or Hundreds of) Millions.

* Quasi-Newton methods (BGFS most popular)
  ↳ This comes with approximation of hessian.

⇒ L-BFGS (Limited memory BGFS)

- → Does not form/store the full inverse Hessian.
- → Usually works very well in full batch, deterministic mode
- → Does not transfer very well to mini-batch setting.

⇒ In practice:

- → Adam is a good default choice in most cases
- → If you can afford to do full batch updates then try out L-BFGS.

* Model Ensembles

1. Train multiple independent models
2. At test time average their results

   Enjoy 2% extra performance

⇒ Instead of training independent models, use multiple snapshots of a single model during training.

⇒ Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time. (Polyak averaging)

★ Regularization : Dropout

⇒ On each forward pass, randomly set some neurons to zero. Probability of dropping is a hyperparameter (0.5 is common)

⇒ Forces the network to have a redundant representation
↳ Prevents co-adaptation of features.

⇒ Dropout makes our output random!

$$y = f_W(x, z)$$

z → Random mask

→ Want to "average out" the randomness at test time

$$y = f(x) = E_z[f(x,z)] = \int p(z)\, f(x,z)\, dz$$

But this integral seems hard...

⇒ 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

⇒ More common **inverted dropout**

↓

{On the train time divide by p}
, to make test time fast

★ Regularization: A Common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness
(Sometime approximate)

$$y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z)\, dz$$

★ Data Augmentation

⇒ Random crops & scales

Training: Sample random crops / scales

Testing: Average a fixed set of crops

⇒ Color Jitter, translation, rotation
, stretching, shearing, lens distortion etc...

⇒ Other regularization:

- Drop Connect
- Fractional Max Pooling
- Stochastic Depth

* Transfer Learning

"You need a lot of data if you want to train / use CNNs"

	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 large number of layers

⇒ Takeaway for your projects & beyond:

Have Some dataset of interest but it has < ~1M images?

① Find a very large dataset that has Similar data, train a big ConvNet there

② Transfer learn to your dataset.

⇒ Deep learning frameworks provide a "Model Zoo" of pretrained models so you don't need to train your own.