optimization for training deep models

- the difference between traditional optimization and that in deep nets is that in deep nets we are optimizing a cost function in hopes that our test set error reduces. the error over the true data generating distribution is called the risk. the error over the training set is called empirical risk. we minimize empirical risk in the hopes that true risk gets minimized.
- we don't want to minimize empirical risk at all costs. specifically, our real goal is risk minimization (for test set) rather than empirical risk minimization.
- we use a surrogate loss function → a differentiable function that might be different from the function we actually care about.
- batch gradient descent involves using the whole dataset for gradient computation. in stochastic gradient descent, we estimate gradient with smaller "minibatches". each iteration over the whole training set is called an epoch.
- challenges in neural network optimization:
  - ill-conditioning: this happens even in the case of convex functions, not just neural nets. the problem is that the Hessian is ill-conditioned which means that SGD will jump to points that will cost an increase in the cost function because the Hessian estimation is poorly conditioned.
  - model identifiability means given a model and a large enough dataset, can we get to the same parameter setting on different training runs. neural nets are highly non-model identifiable because we can simply scale the incoming weights of a unit by α if we also scale its outgoing units (in a ReLU network). this shows multiple points that have the same cost. this means there are tons of local minima with the same cost.
  - even though there are a ton of local minima, this is not actually believed to be a problem. empirically, run experiments to see this.
  - there are actually a lot more saddle points (where the Hessian has positive and negative eigenvalues) than local minima points. SGD is known to quite easily escape these saddle points though.
  - the thing that is a big problem is flat regions that are hard to escape because gradient is zero in these regions.
  - long term dependencies lead to vanishing and exploding gradient problems (like we talked about in the notes on the Nielsen book)
  - cliffs and exploding gradients are challenging because they make us jump too far into a place where the cost function might increase.
  - bad correspondance between gradient and the overall structure of a region can misguide us in paths that are non-optimal in getting to a peak/trough.
- learning rate is a size of step (what we multiply the gradient by before taking a step)
- Momentum methods:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

The velocity is set to an exponentially decaying average of the negative gradient.

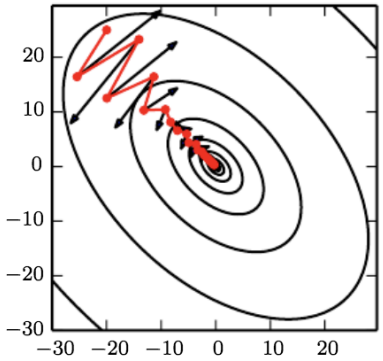

Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

- Nesterov momentum:
  Take the $\theta + \alpha\mathbf{v}$ step, compute the gradient there, then compute $\mathbf{v} = \alpha\mathbf{v} - \epsilon\nabla_x(\theta + \alpha\mathbf{v})$. Now $\theta = \theta + \mathbf{v}$ as usual.
- A difficulty with parameter initialization is that some points might be good for low training set error but bad for generalization → we want to avoid this but it is deceptive because of low training set error.
- We want to initialize neural net weights with a small random value. With ReLU, for example, we'd like to stay close to 0, maybe slightly positive to give it a chance to learn.
- For output units, we can initialize biases to indicate the marginal values of different label occurrences in the training set.
- We may set of bias of ReLU units to 0.1 to avoid saturation at initialization.
- supervised pretraining on a related task is also a good way to do initialization that may have not just optimization benefits, but more importantly, generalization benefits that take the parameters to good points in space.
- algorithms with adaptive learning rates:
  - delta-bar-delta: if partial derivative of loss with respect to a model param remains the same sign, then learning rate should increase, else decrease → is only effective on full batch optimization.
  - adagrad → scale each model param inversely proportional to the square root of all historical values. the params with largest partial derivative of loss have a rapid decrease and vice versa, thus making greater progress in the more gently sloped directions.
  - RMSProp → adagrad scales with the full history of gradient in mind. RMSProp uses an exponentially decaying average to discard history from extreme past so that it can converge rapidly after finding a convex bowl, even if past history has a large gradient (and hence gets scaled down a lot if we took full gradient history).
  - Adam (adaptive moments) → RMSProp + momentum. We apply momentum to the re-scaled gradients from RMSProp.
- Second order methods:

    - second order methods can make use of more curvature information but can be hard to scale because the Hessian blows up in size.
    - Newton's method is based on writing the taylor expansion upto second degree, solving analytically for the minimum and doing that iteratively (if it were a convex function and our quadratic approximation is perfect, we'd be done with just iteration)
    - Conjugate gradients and BFGS are two second order methods for neural net optimization.
    - BFGS works by iteratively finding a matrix approximation to the Hessian matrix needed for the newton's method of optimization.
    - conjugate gradient method: consider SGD with line search. Let the previous search direction be $\mathbf{d}_{t-1}$. At the minimum, where the line search terminates, the directional derivative is zero in direction $\mathbf{d}_{t-1}$, so $\nabla_\theta J(\theta) \odot \mathbf{d}_{t-1} = 0$. Since the gradient at this point defines the current search direction, $dt = \nabla_\theta J(\theta)$ will have no contribution in the direction $\mathbf{d}_{t-1}$. Thus it is orthogonal to $\mathbf{d}_{t-1}$. This gives rise to the zig-zag pattern of progress, where by descending to the minimum in the current gradient direction, we must reminimize the objective in the previous gradient direction.
      To overcome the above, we use the update rule, $\mathbf{d}_t = \nabla_\theta J(\theta) + \beta_{t-1}\mathbf{d}_{t-1}$. We won't go into how to come up with β_{t-1}.
- Batch normalization is an optimization trick where if we subtract by mean activation of a layer and divide by variance, we get better convergence. we do this normalization after each layer's output has been computed.
- why does batch norm do this? the gradient estimates change in cost function assuming all other params are fixed. but all params are updated simultaneously and aren't fixed. if you're a layer deep in the neural net, then if an image has a different contrast but similar deep layer representation for the layer before you, then you are protecting against having to learn from changes in contrast (magnitude of activations) and instead only learning from the magnitude of activation layers for the layer before you. when we backprop thru the normalization operation, the gradient will never propose an operation that acts simply to increase/decrease the standard deviation or mean!
- When you have a large dataset, it's important to optimize well, and not as important to regularize well, so batch normalization is more important for large datasets. But, batch norm, in addition to optimization, also has a regularizing effect, because it basically introduces noise and forces the network to be robust to it.
- polyak averaging and supervised pre-training are also useful but we won't go into them here → these are pretty general concepts.
- we should also aim to choose and come up with model families that are easier to optimize instead of relying on powerful optimization algorithms. most neural net advances have come from this strategy. eg. CNN, LSTM etc.
- Curriculum learning is a strategy where we learn the easy examples first and take the parameters to good places and then learn the harder examples. it has been shown to empirically work well.