CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 2**

This discussion will first talk about the bias-variance tradeoff and go through an example to illustrate how regularization can affect the bias and variance. We will then go over a review of notation in vector/matrix calculus that we will need to understand backpropagation, and then finally review several optimization algorithms covered in lecture.

1 Bias-Variance Tradeoff

1.1 Intuitive Understanding of Bias-Variance Tradeoff

First, recall the definitions of bias and variance from last discussion,

Definition 1 (Bias of an Estimator)**.** *The bias of an estimator is a measure of how much does the expected value of the estimator differ from the true target. Suppose we have a randomly sampled training set $\mathcal{D}$, and we select an estimator denoted $\theta = \hat{\theta}(\mathcal{D})$. Then, for a particular test input x, the bias of our estimator's prediction on x is given as* $\text{Bias}(f_\theta(x)) = \mathbb{E}_{y\sim p(y|x),\mathcal{D}}[f_\theta(x) - y]$*. The variance of an estimate is a measure of how much the estimate differs from the expected value of the estimate, and is given by* $\text{Var}(f_\theta(x)) = \mathbb{E}_{\mathcal{D}}[(f_\theta(x) - \mathbb{E}_{\mathcal{D}}[f_\theta(x)])^2]$.

In supervised learning, our goal is to learn a function that does well in terms of the true risk. However, we generally do not know the true distribution and only have access to a dataset of samples from the distribution, and instead learn by minimizing the *empirical risk* (often with an additional regularization term) instead.

Even though we cannot directly optimize the risk, we can still attempt to better understand sources of error in our estimation. In particular, when our loss is the squared error, we can derive the **bias-variance** decomposition of the MSE. Specifically, we will find that the MSE estimator is exactly to the variance of the estimator plus the square of its bias (and an irreducible error).

Intuitively, the bias and variance can be summarized by the following graphic:

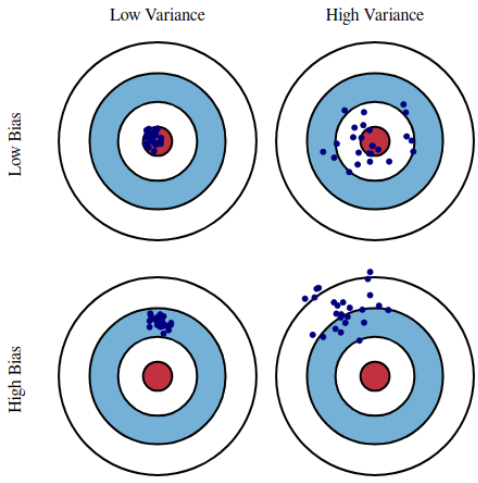


Figure 1: A visual explanation of the bias and variance. Figure from [**?**].

Here, notice that when there is a high variance, the estimates are more spread out, but when there is a high bias, we see a general deviation away from our target. The best estimates are those with low variance and low bias since they mostly hit the target.

1.2 Bias-Variance Tradeoff Mechanics

Problem 1: Deriving Bias-Variance Tradeoff

Suppose we have a randomly sampled training set $\mathcal{D}$ (drawn independently from our test data), and we select an estimator denoted $\theta = \hat{\theta}(\mathcal{D})$ (for example, via empirical risk minimization).
Show that we can decompose our expected mean squared error for a particular test input x into a bias, a variance and an irreducible error term as below:

$$\mathbb{E}_{Y\sim p(y|x),\mathcal{D}}[(Y - f_{\hat{\theta}(\mathcal{D})}(x))^2] = \text{Var}(f_{\hat{\theta}(\mathcal{D})}(x)) + \text{Bias}(f_{\hat{\theta}(\mathcal{D})}(x))^2 + \sigma^2$$

You may find it helpful to recall the formulaic definitions of Variance and Bias, reproduced for you below:

$$\text{Var}(f_{\hat{\theta}(\mathcal{D})})(x) = \mathbb{E}_{\mathcal{D}}\left[(f_{\hat{\theta}(\mathcal{D})}(x) - \mathbb{E}[f_{\hat{\theta}(\mathcal{D})}(x)])^2\right]$$
$$\text{Bias}(f_{\hat{\theta}(\mathcal{D})}(x)) = \mathbb{E}_{Y\sim p(Y|x),\mathcal{D}}[f_{\hat{\theta}(\mathcal{D})}(x) - Y]$$

We have now decomposed our test risk into a bias, variance and irreducible error term. As there is nothing we can do about the irreducible error, this tells us that we need to choose the learning algorithm and/or hyperparameters $\hat{\theta}(\cdot)$ in order to simultaneously achieve low bias and low variance. The next two questions will show how the choice of estimator $\hat{\theta}$ can influence bias and variance. In particular, we will see that ℓ_2 regularization in linear regression can provide a *tradeoff* between bias and variance.

Problem 2: Deriving Bias and Variance of Linear Regression Models

Our dataset consists of $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$. We let the label vector $Y = \mathbf{X}\theta + \boldsymbol{\varepsilon}$ where θ is the true linear predictor and each noise variable $\boldsymbol{\varepsilon}_i$ is i.i.d. with mean 0 and variance 1. We use the ordinary least squares model. Calculate the bias and covariance of the $\hat{\theta}$ estimate and use that to compute the bias and variance of the prediction at particular test inputs x. Recall that the OLS solution is given by

$$\hat{\theta} = (X^\top X)^{-1} X^\top Y,$$

where $X \in \mathbb{R}^{n\times d}$ is our (nonrandom) data matrix, $Y \in \mathbb{R}^d$ is the (random) vector of training targets. For simplicity, assume that $X^\top X$ is diagonal (we could have applied an orthogonal transformation to make this the case), or for an even simpler problem that doesn't require linear algebra, assume $X \in \mathbb{R}^{n\times 1}$, making $X^\top X$ simply a scalar value.

Problem 3: Deriving Bias and Variance of Linear Regression Models (Challenge)

What happens to the bias and variance if we instead use an ℓ_2 regularized estimator

$$\tilde{\theta} = (X^\top X + \lambda I)^{-1} X^\top Y?$$

2 Vector and Matrix Calculus Review

In this section, we review vector and matrix calculus, and formalize the notation we will use. These notations will be required to understand backpropagation in the next lectures. Henceforth, we will denote scalars with lowercase letters (e.g., x), vectors with bolded lowercase letters (e.g., $\mathbf{x}$) and matrices with upper case letters (e.g., X). We will use similar conventions for functions depending on the shape of its output (e.g., $\mathbf{g}(\cdot)$ denotes a function with a vector valued output).

Gradients with respect to vectors We first define the gradient of a scalar function with respect to a vector input. Suppose we have a function $f : \mathbb{R}^d \to \mathbb{R}$, which maps a d-dimensional vector to a scalar. Then we define the gradient of f at a particular input x to be a column vector (the same shape as the input) consisting of partial derivatives at x:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\mathbf{x}) \end{bmatrix}.$$

We note that this choice of notation (laying out gradients to be the same shape as input) is not universally used; you will often find sources using the opposite convention (especially in mathematics) with gradients as row vectors (and Jacobians will be the transpose of what we describe next). However, we will use this convention for deep learning because it is intuitive (for example, in gradient descent, we often write $\theta \leftarrow \theta - \alpha \nabla L(\theta)$, which only makes sense when θ and $\nabla L(\theta)$ are the same shape) and because it is easily extended to gradients for matrices and higher dimensional arrays.

Problem 4: Gradient of squared ℓ_2 norm

Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$, and let $f(\mathbf{x}) = \|\mathbf{x}\|_2^2 = \mathbf{x}^\top \mathbf{x}$. Compute the gradient $\nabla f(\mathbf{x})$.

Jacobians We now consider the case where $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ has vector valued inputs and outputs. Let $f_i : \mathbb{R}^n \to \mathbb{R}$ be the function that outputs the ith component of $\mathbf{f}$. Then, we can view our Jacobian (which we shall denote as $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$) as stacking together the gradients of f_i for $i \in \{1, \ldots, m\}$. That is, the Jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ will be an $n \times m$ matrix with entries given by

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{ij} = \frac{\partial f_j}{\partial x_i}.$$

Problem 5: Jacobian of a linear map

Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{d \times n}$. Let $\mathbf{f}(\mathbf{x}) = A^\top \mathbf{x} \in \mathbb{R}^n$. Compute the Jacobian of $\mathbf{f}$ with respect to $\mathbf{x}$.

Multivariate Chain Rule We first recall the basic chain rule when everything is scalar valued. Suppose we have an input x, compute $y = g(x)$, then compute $z = f(y)$. Then the chain rule says

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \frac{\partial z}{\partial y}.$$

Now let's consider the case where $\mathbf{y}$ is vector valued in $\mathbb{R}^n$ (x and z remain scalars). Summing over the

contributions of each entry of $\mathbf{y}$, we see $\frac{\partial z}{\partial x}$ is now a scalar given by

$$\sum_{i=1}^{n} \frac{\partial y_i}{\partial x} \frac{\partial z}{\partial y_i}.$$

Finally, let's consider the case when $\mathbf{x}$ is also a vector in $\mathbb{R}^m$. From our calculation with scalar x and vector $\mathbf{y}$, we know the jth entry of $\frac{\partial z}{\partial \mathbf{x}}$ is given by the partial derivative

$$\frac{\partial z}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x_j}.$$

Stacking together the entries $\frac{\partial z}{\partial x_j}$ into a vector, we see that the gradient of the output z with respect to x is given by the product of the Jacobian matrix of $\mathbf{y}$ with respect to $\mathbf{x}$ and the gradient of z with respect to $\mathbf{y}$:

$$\overbrace{\frac{\partial z}{\partial \mathbf{x}}}^{\mathbb{R}^m} = \overbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}^{\mathbb{R}^{m \times n}} \overbrace{\frac{\partial z}{\partial \mathbf{y}}}^{\mathbb{R}^n}.$$

Problem 6: Combining the two previous calculations with the chain rule

Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{d \times n}$. Let $\mathbf{g}(\mathbf{x}) = A^\top \mathbf{x} \in \mathbb{R}^n$, and let $f(\mathbf{y}) = \|\mathbf{y}\|_2^2$. Compute the gradient of $f(\mathbf{g}(\mathbf{x}))$ with respect to $\mathbf{x}$.

Gradients with respect to matrices and higher dimensional arrays Now suppose we have a function $f : \mathbb{R}^{d_1 \times d_2} \to \mathbb{R}$, which maps a d_1 by d_2 matrix to a scalar. We will again define the gradient at a particular input matrix X to be a matrix of the same shape as X, consisting of the partial derivatives with respect to each entry of the matrix.

$$\nabla_X f(X) = \begin{bmatrix} \frac{\partial f}{\partial X_{11}} & \cdots & \frac{\partial f}{\partial X_{1,d_2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial X_{d_1,1}} & \cdots & \frac{\partial f}{\partial X_{d_1,d_2}} \end{bmatrix}.$$

Similarly, we can generalize this convention of having the gradient match the shape of the input when our input were higher dimensional arrays (e.g. in the weights of a convolutional layer).

We can define a version of a Jacobian for vector-valued functions with matrix inputs that preserves the matrix dimensions (similarly for higher dimensional arrays as well). Suppose $\mathbf{f} : \mathbb{R}^{d_1,d_2} \to \mathbb{R}^n$, then we can define the Jacobian to be a rank-3 tensor (an array with 3 indices) in $\mathbb{R}^{d_1 \times d_2 \times n}$ with each entry given by

$$\left(\frac{\partial \mathbf{f}}{\partial X}\right)_{ijk} = \frac{\partial f_k}{\partial X_{ij}}.$$

We will now go through the chain rule calculation again, this time with a matrix input. Suppose $X \in \mathbb{R}^{d_1,d_2}$, $\mathbf{y} = \mathbf{g}(X) \in \mathbb{R}^n$ and $z = f(\mathbf{y}) \in \mathbb{R}$. Again, we have that the partial derivative with respect to each entry of the matrix X_{ij} is given by

$$\frac{\partial z}{\partial X_{ij}} = \sum_{k=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial X_{ij}}.$$

Similarly to the vector input case, we can again succinctly write out the full gradient with respect to the matrix X as

$$\overbrace{\frac{\partial z}{\partial X}}^{\mathbb{R}^{d_1 \times d_2}} = \overbrace{\frac{\partial y}{\partial X}}^{\mathbb{R}^{d_1 \times d_2 \times n}} \overbrace{\frac{\partial z}{\partial y}}^{\mathbb{R}^{n}}.$$

Note that the product of the rank-3 tensor (or 3-dimensional array) and vector can be a seen as a generalization of a matrix vector multiplication. Multiplying a matrix $X \in \mathbb{R}^{m \times n}$ by a vector $y \in \mathbb{R}^n$ results in a vector in $\mathbb{R}^m$ where each entry is the inner product of a row of X with y. The product of a rank-3 tensor $A \in \mathbb{R}^{d_1 \times d_2 \times n}$ with a vector $\mathbf{b} \in \mathbb{R}^n$ then forms a $d_1 \times d_2$ matrix, where each entry is the inner product of a "row" of A and $\mathbf{b}$.

We also note that this calculation of the gradient with respect to a matrix X is equivalent to first flattening X to a vector, computing the gradient with respect to the flattened X using the previous multivariate chain rule for vectors, and then reshaping the gradients back to match the original matrix shape of X.

Problem 7: Revisiting with a matrix derivative instead

In problem 5, we computed the gradient of $z = \left\|A^\top \mathbf{x}\right\|_2^2$ with respect to $\mathbf{x}$. We will now repeat this exercise, but instead compute the gradient with respect to A.
Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{d \times n}$. Let $\mathbf{g}(A) = A^\top \mathbf{x} \in \mathbb{R}^n$, and let $f(\mathbf{y}) = \|\mathbf{y}\|_2^2$. Compute the gradient of $f(\mathbf{g}(A))$ with respect to A.

Finally, as a matter of notation, note that the way we order derivatives in our chain rule (with the final output on the rightmost side) is again a result of our chosen convention for gradients and Jacobians. You may notice in other texts that the chain rule is written in the reversed order using a different convention for Jacobians.

3 Optimization methods

To perform empirical risk minimization, we need to choose an algorithm to compute the optimal parameters for the empirical risk. In deep learning, we almost always use methods based off stochastic gradient descent due its scalability (both in terms of dataset size and model size), and we'll go through and review several optimization methods as introduced in lecture.

3.1 Gradient Descent

For all our algorithms, we assume we can compute the gradients of our loss function for each data point $\nabla_\theta L(\boldsymbol{x}_i, y_i, \boldsymbol{\theta})$. The negative gradient of a function gives the steepest descent direction; that is, the direction we should move in order to decrease the loss most quickly if we moved an infinitesimally small amount.

Given this, the most natural method for minimizing our training loss is to iteratively compute the gradient for the entire dataset $\mathcal{D}$, and update our parameter some small amount in that direction. This leads to the (batch) **gradient descent** algorithm which computes iterates as

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \frac{\alpha}{|\mathcal{D}|} \sum_{\boldsymbol{x}_i, y_i \in \mathcal{D}} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}_i, y_i, \boldsymbol{\theta}^t),$$

where α is a fixed scalar known as the step size. However, this naive algorithm requires us to take the average gradient over the entire training dataset for each iteration, which can be too slow for larger datasets.

3.2 Stochastic Gradient Descent (SGD)

In order to speed up gradient descent, we can instead sample a random subset of the dataset for each iteration, leading to the (minibatch) **stochastic gradient descent** algorithm. The gradient estimate at each iteration is now noisy (with the amount of noise depending on the minibatch size), but is an unbiased estimator for the true gradient and can still provide useful directions to update our parameters. Compared to batch gradient descent, we are trading off variance in the gradients for much faster gradient computation that does not need to scale with the dataset size. While extra noise can mean it requires more iterations to converge, the faster individual iterate time can more than offset the cost.

The SGD algorithm proceeds exactly as the gradient descent algorithm, only replacing the full training set $\mathcal{D}$ with a new random minibatch B^t for each iteration

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \frac{\alpha}{|B^t|} \sum_{\boldsymbol{x}_i, y_i \in B^t} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}_i, y_i, \boldsymbol{\theta}^t) \qquad\qquad B^t \subseteq \mathcal{D}.$$

All algorithms below are all compatible with stochastic gradients, so for notational convenience, we will now denote the (possibly stochastic) gradient estimate at iterate t to simply be $\nabla L(\theta^t)$. The new SGD/GD update in this notation would simply be

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \alpha \nabla L(\boldsymbol{\theta}^t).$$

3.3 When does gradient descent work poorly?

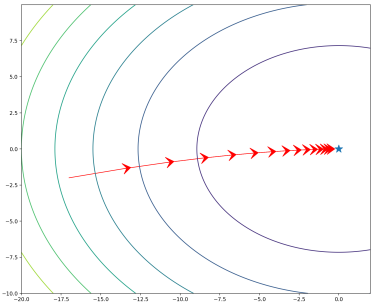
We'll first understand when gradient descent fails for a very simple convex problem, which will motivate different extensions. We consider a simple quadratic loss function $f(x_1, x_2) = a_1 x_1^2 + a_2 x_2^2$ and we'll look at the behavior of GD for different settings of the loss parameters a_1, a_2.

Gradient descent can work well when f is "nice" in some sense (which we'll refer to as being well-conditioned), which will be the case if a_1 and a_2 are similar in magnitude. We see this in Figure 2a, where the direction of gradient always aligns closely with the direction of the optimum, and GD can work with a fairly large learning rate to make quick steady progress to the optimum.

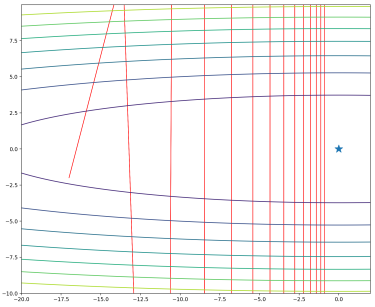
In Figures 2b and 2c, we consider an "ill-conditioned" problem where $a_2 \gg a_1$. In Figure 2b, with the same learning rate as before, the gradients tend to have a much larger vertical component than horizontal, and

the iterates now oscillate greatly along the vertical axis and diverge. If we use a much smaller learning rate in Figure 2c, we are able to stabilize the parameter updates in the vertical axis, but we end up making much slower progress to the optimum along the horizontal axis.

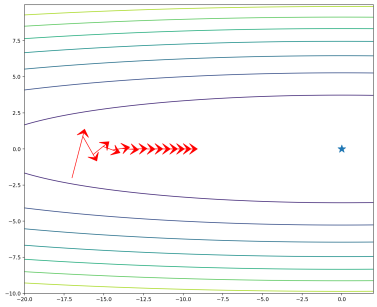
Thus, when our iterates are forced to always move in the direction of the gradient, we can have situations where either the iterates are very unstable, or our learning rate is so small that we make very slow progress. To address this pathology, we shall now examine different methods which do not update the parameters exactly in the direction of the gradient, but instead modifies the direction based on past gradients seen.



(a) With a large learning rate, GD on this well-conditioned problem makes fast steady progress towards the optimum.



(b) With a large learning rate, GD on this ill-conditioned problem oscillates and diverges in the vertical direction.



(c) With a small learning rate, GD on this ill-conditioned problem remains stable, but makes very slow progress towards the optimum.

Figure 2: Different behaviors of gradient descent on different problems and with different learning rates.

3.4 SGD with momentum

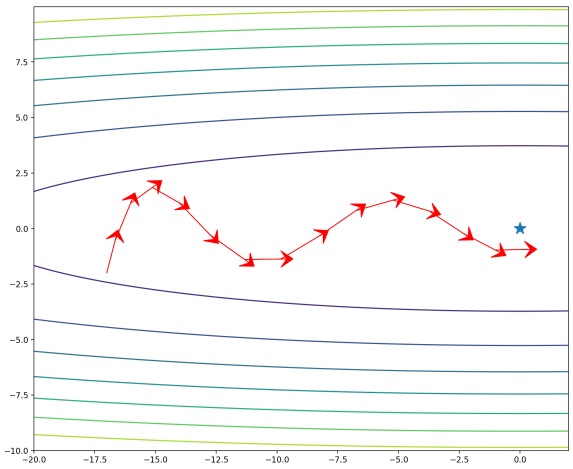


Figure 3: Effect of using momentum on the ill-conditioned quadratic problem. SGD with momentum quickly realizes that all the gradients consistently point towards the right, so it "gains momentum" and moves faster along that direction over time. On the other hand, the alternating vertical gradients tend to cancel out over the past iterations, damping the amount of vertical oscillation. This way, the iterates make quick progress toward the optimum along the horizontal direction, while not diverging along the vertical axis.

One way to help alleviate this problem is to accumulate gradient information across previous iterates in order to damp the oscillations and focus on the directions where we have consistently been moving in. In the example in the left of Figure 2b, we notice that each gradient consistently moves the iterate towards the right (towards the optimum), while alternating iterations have gradients point in different directions along

the vertical axis. If we average the past gradients using momentum (Figure 4), we see that the vertical components of the gradient will tend to cancel out and stabilize, allowing the horizontal movement towards the right to dominate and lead us to the optimum more quickly. In the stochastic gradient setting, momentum can also reduce the impact of noise, again by smoothing out the oscillations int he gradient direction incurred by the random data sampling.

Concretely, the momentum method (the particular variant we use is also called the heavy-ball method) keeps a moving average of our gradients, weighting more recent gradients more heavily, and updates our iterate in the direction of the weighted average. The iterates proceed as

$$\begin{aligned} \mathbf{v}^t &= m\mathbf{v}^{t-1} + \nabla L(\boldsymbol{\theta}^t) \\ \boldsymbol{\theta}^{t+1} &= \boldsymbol{\theta}^t - \alpha \mathbf{v}^t. \end{aligned}$$

Here $\mathbf{v}^t$ is our accumulated gradient vector, m controls how much we remember the past gradients, and α is our step size as before.

3.5 RMSProp and Adagrad

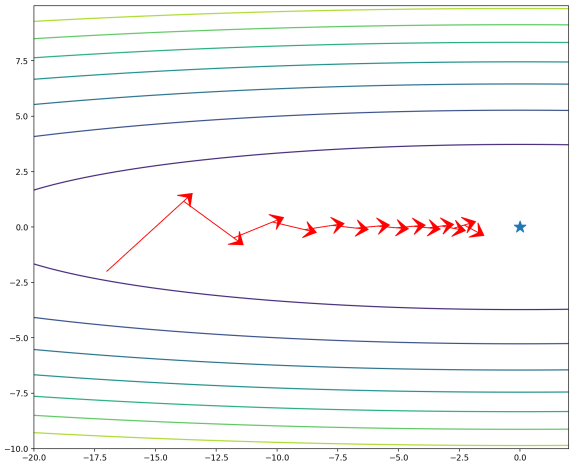


Figure 4: Effect of using adaptive learning rates on the ill-conditioned quadratic problem. In this case, we use RMSProp, which is able to rescale the different parameter updates to make fast progress to the optimum instead of oscillating as regular gradient descent did.

Adaptive learning rate methods, which include RMSprop and Adagrad, are an alternative approach towards selecting a better direction based on rescaling the different components of the gradient to form a new direction. One issue we notice with the ill-conditioned loss in the previous figures is that the loss is much more sensitive along the vertical axis than the horizontal one, and that the extreme sensitivity along the vertical axis was what forced us to use a small learning rate for gradient descent. Intuitively, if we could decouple learning rates for each coordinate and rescale our updates to place more importance on the horizontal direction, we would move much faster towards the optimum.

Concretely, both RMSProp and Adagrad do this individual rescaling by estimating a vector $\mathbf{s}^t$ that tracks the "size" of the past gradients in each dimension. They both update the kth coordinate of the parameters according to

$$\theta_k^{t+1} = \theta_k^t - \frac{\alpha}{\sqrt{s_k^t + \epsilon}} \nabla_{\theta_k} L(\theta^t),$$

where ϵ is a small constant for numeric stability (to avoid dividing by zero).

RMSProp and Adagrad differ in how they update s^t:

$$s_k^t = \beta s_k^{t-1} + (1-\beta)(\boldsymbol{\nabla}_{\theta_k} L(\theta^t)^2 \qquad \text{RMSProp}$$
$$s_k^t = s_k^{t-1} + \beta(\boldsymbol{\nabla}_{\theta_k} L(\theta^t))^2 \qquad \text{Adagrad}$$

RMSProp keeps a *running average* of per dimension gradient magnitudes, while Adagrad keeps a *sum*. Thus, for Adagrad, the vector $\mathbf{s}^t$ monotonically increases over time, causing the effective learning for each coordinate to decrease monotonically.

3.6 Adam and other algorithms

Adam is a popular optimizer in deep learning that essentially combines the adaptive learning rates of RMSProp with momentum.

For additional information about these optimization methods (as well as numerous others), here's a blog post with a list of popular optimization methods for deep-learning: http://ruder.io/optimizing-gradient-descent/