

# **10-414/714 – Deep Learning Systems: Algorithms and Implementation**

## **Fully connected networks, optimization**

Fall 2022

J. Zico Kolter (this time) and Tianqi Chen  
Carnegie Mellon University

# Outline

Fully connected networks

Optimization

Initialization

# Outline

Fully connected networks

Optimization

Initialization

# Fully connected networks

Now that we have covered the basics of automatic differentiation, we can return to “standard” forms of deep networks

A *L-layer, fully connected network*, a.k.a. multi-layer perceptron (MLP), now with an explicit bias term, is defined by the iteration

$$\begin{aligned} z_{i+1} &= \sigma_i(W_i^T z_i + b_i), \quad i = 1, \dots, L \\ h_\theta(x) &\equiv z_{L+1} \\ z_1 &\equiv x \end{aligned}$$

with parameters  $\theta = \{W_{1:L}, b_{1:L}\}$ , and where  $\sigma_i(x)$  is the nonlinear activation, usually with  $\sigma_k(x) = x$

# Matrix form and broadcasting subtleties

Let's consider the matrix form of the iteration

$$Z_{i+1} = \sigma_i(Z_i W_i + \mathbf{1} b_i^T)$$

Notice a subtle point: to write things correctly in matrix form, the update for  $Z_{i+1} \in \mathbb{R}^{m \times n}$  requires that we form the matrix  $\mathbf{1} b_i^T \in \mathbb{R}^{m \times n}$


In practice, you don't form these matrices, you perform operation via *broadcasting*

- E.g. for a  $n \times 1$  vector (or higher-order tensor), broadcasting treats it as an  $n \times p$  matrix repeating the same column  $p$  times
- We could write iteration (informally) just as  $Z_{i+1} = \sigma_i(Z_i W_i + b_i^T)$
- Broadcasting does not copy any data (described more in later lecture)

# Key questions for fully connected networks

In order to actually train a fully-connected network (or any deep network), we need to address a certain number of questions:

- How do we choose the width and depth of the network?
- How do we actually optimize the objective? (“SGD” is the easy answer, but not the algorithm most commonly used in practice)
- How do we initialize the weights of the network?
- How do we ensure the network can continue to be trained easily over multiple optimization iterations?



All related questions that affect each other

There are (still) no definite answers to these questions, and for deep learning they wind up being problem-specific, but we will cover some basic principles

# Outline

Fully connected networks

Optimization

Initialization

# Gradient descent

Let's reconsider the generic gradient descent updates we described previously, now for a general function  $f$ , and writing iterate number  $t$  explicitly

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} f(\theta_t)$$

where  $\alpha > 0$  is step size (learning rate),  $\nabla_{\theta} f(\theta_t)$  is gradient evaluated at the parameters  $\theta_t$

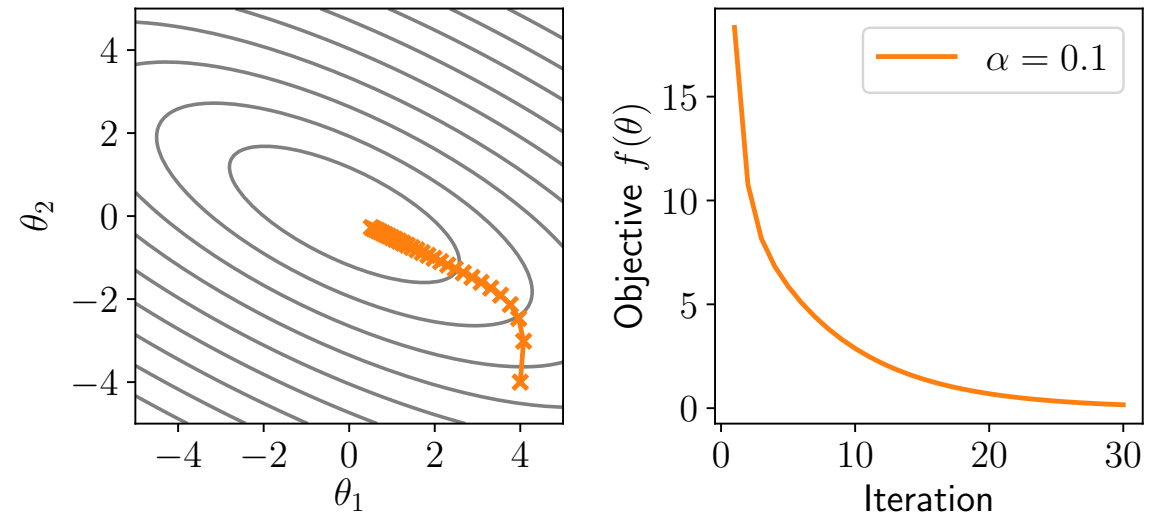
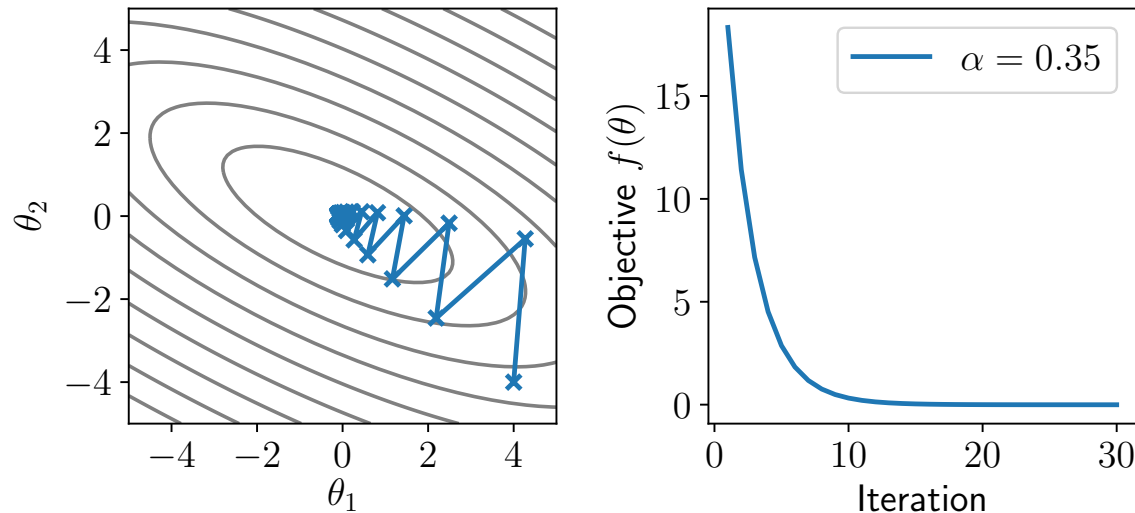
Takes the “steepest descent direction” locally (defined in terms of  $\ell_2$  norm, as we will discuss shortly), but may oscillate over larger time scales



# Illustration of gradient descent

For  $\theta \in \mathbb{R}^2$ , consider quadratic function  $f(\theta) = \frac{1}{2} \theta^T P \theta + q^T \theta$ , for  $P$  positive definite (all positive eigenvalues)

Illustration of gradient descent with different step sizes:



# Newton's Method

One way to integrate more “global” structure into optimization methods is Newton's method, which scales gradient according to inverse of the Hessian (matrix of second derivatives)

$$\theta_{t+1} = \theta_t - \alpha \left( \nabla_{\theta}^2 f(\theta_t) \right)^{-1} \nabla_{\theta} f(\theta_t)$$

where  $\nabla_{\theta}^2 f(\theta_t)$  is the *Hessian*,  $n \times n$  matrix of all second derivatives

Equivalent to approximating the function as quadratic using second-order Taylor expansion, then solving for optimal solution

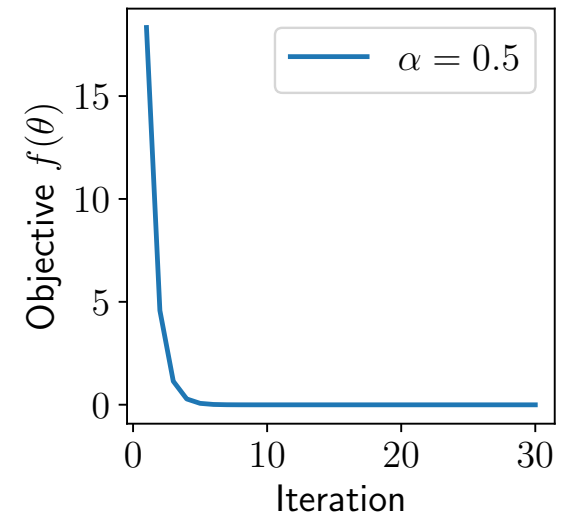
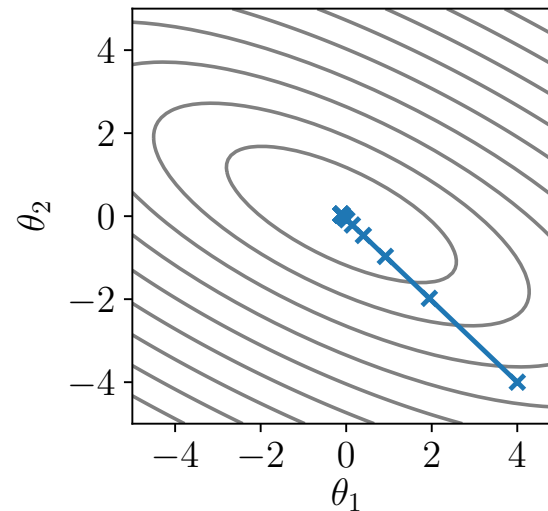
Full step given by  $\alpha = 1$ , otherwise called a *damped* Newton method

# Illustration of Newton's method

Newton's method (with  $\alpha = 1$ ) will optimize quadratic functions in one step

Not of that much practical relevance to deep learning for two reasons

1. We can't efficiently solve for Newton step, even using automatic differentiation (though there are tricks to approximately solve it)
2. For non-convex optimization, it's very unclear that we even *want* to use the Newton direction



# Momentum

Can we find “middle grounds” that are as easy to compute as gradient descent, but which take into account more “global” structure like Newton’s method

One common strategy is to use *momentum* update, that takes into account a moving average of *multiple* previous gradients

$$\begin{aligned}u_{t+1} &= \beta u_t + (1 - \beta) \nabla_{\theta} f(\theta_t) \\ \theta_{t+1} &= \theta_t - \alpha u_{t+1}\end{aligned}$$

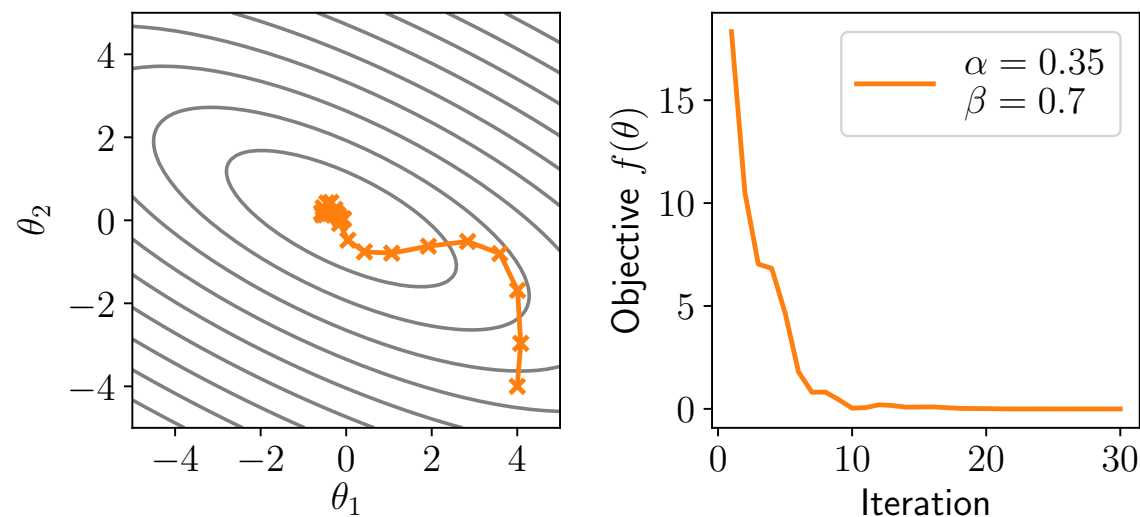
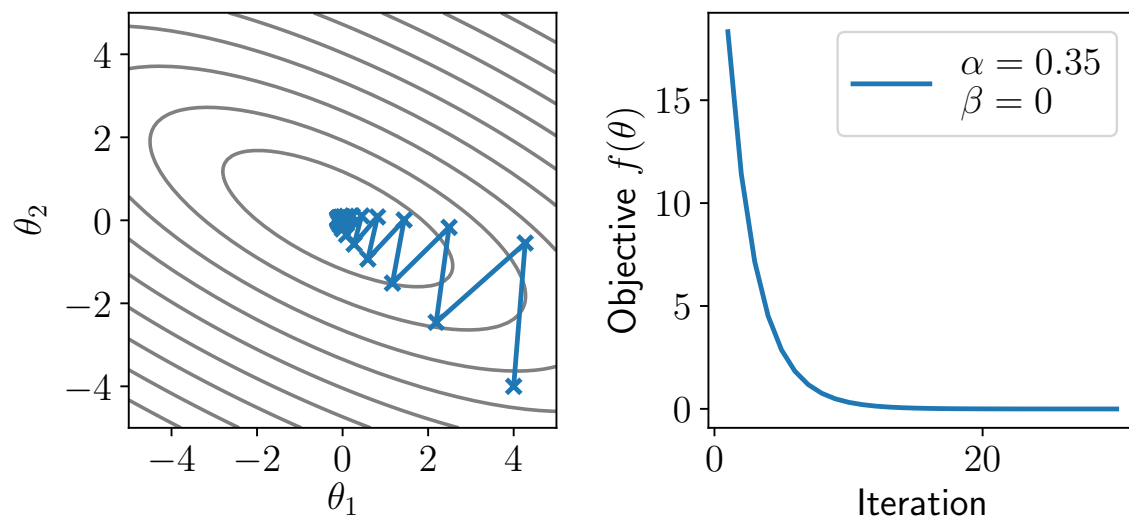
where  $\alpha$  is step size as before, and  $\beta$  is momentum averaging parameter

- Note: often written in alternative forms  $u_{t+1} = \beta u_t + \nabla_{\theta} f(\theta_t)$  (or  $u_{t+1} = \beta u_t + \alpha \nabla_{\theta} f(\theta_t)$ ) but I prefer above to keep  $u$  the same “scale” as gradient

# Illustration of momentum

Momentum “smooths” out the descent steps, but can also introduce other forms of oscillation and non-descent behavior

Frequently useful in training deep networks in practice

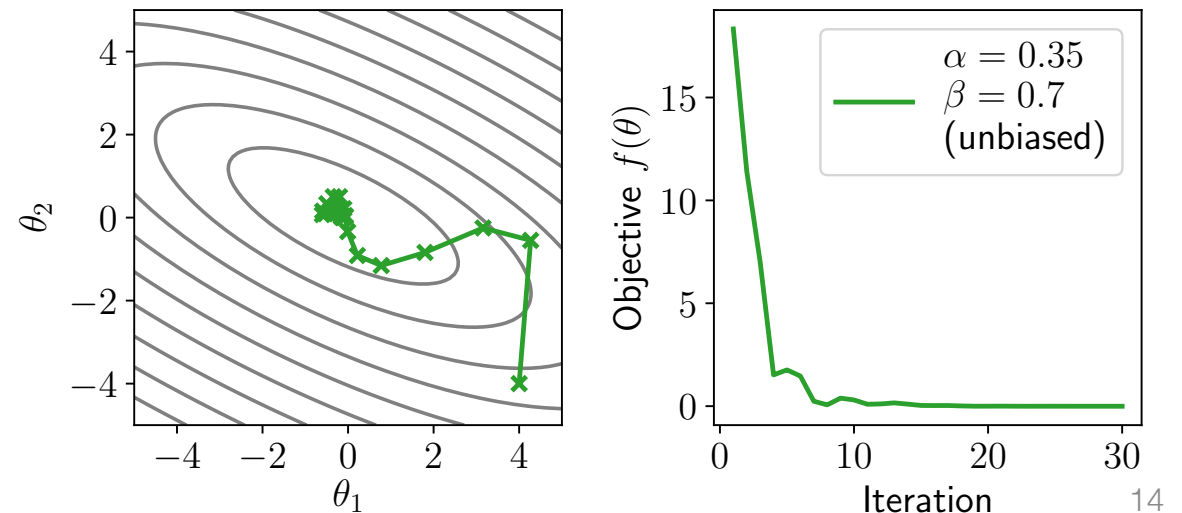
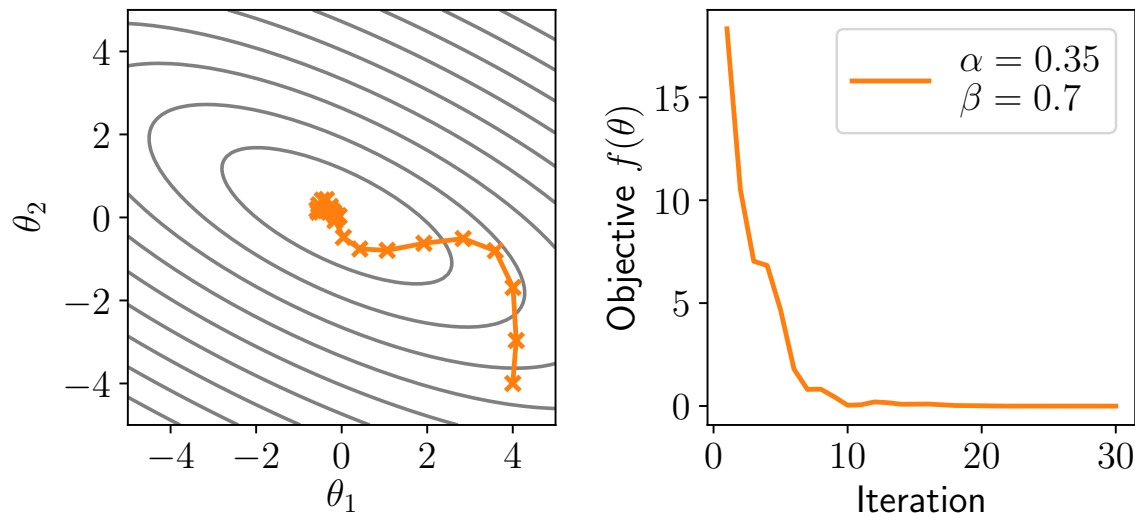


# “Unbiasing” momentum terms

The momentum term  $u_t$  (if initialized to zero, as is common), will be smaller in initial iterations than in later ones

To “unbias” the update to have equal expected magnitude across all iterations, we can use the update

$$\theta_{t+1} = \theta_t - \alpha u_t / (1 - \beta^{t+1})$$

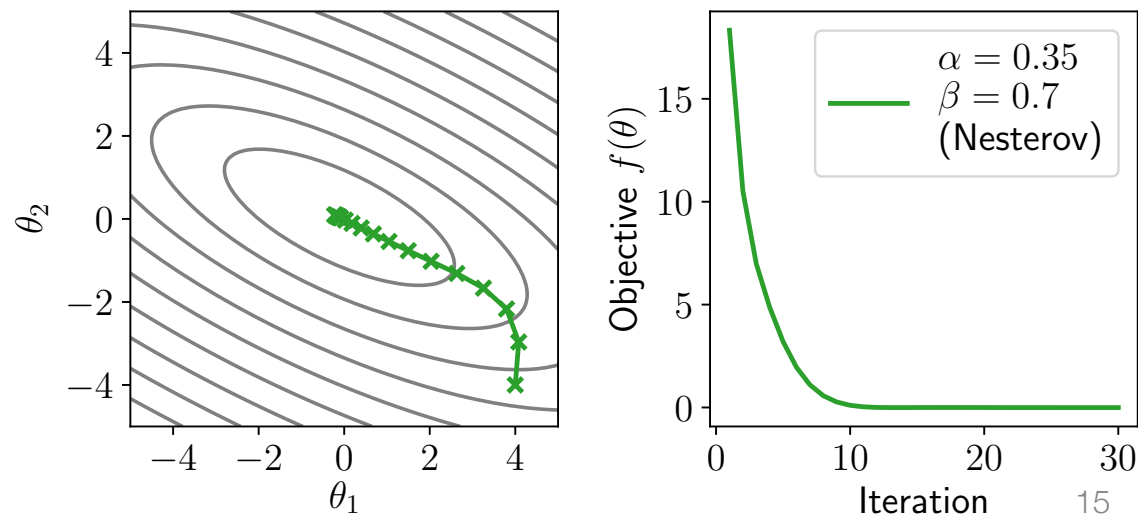
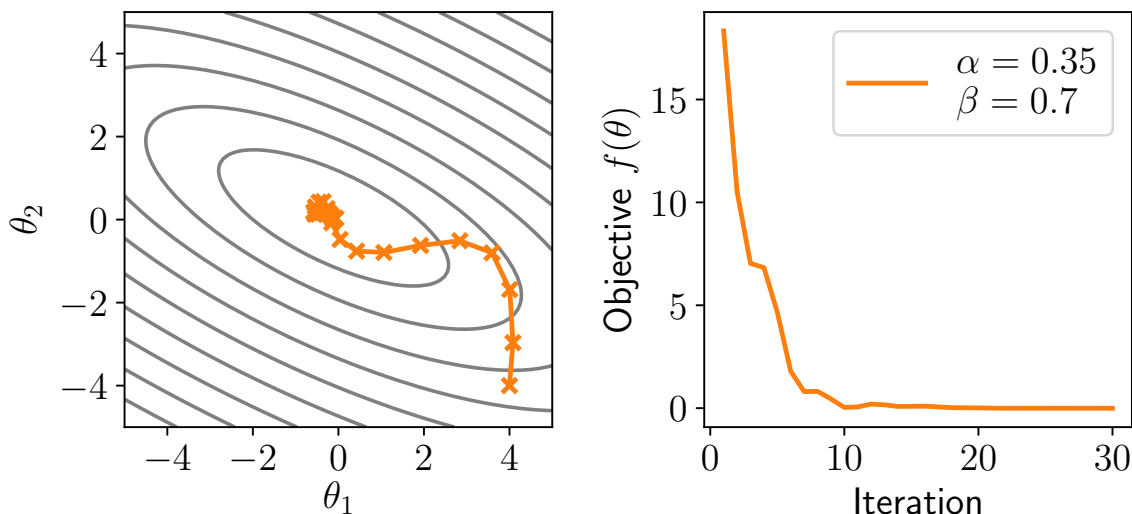


# Nesterov Momentum

One (admittedly, of many) useful tricks in the notion of Nesterov momentum (or Nesterov acceleration), which computes momentum update at “next” point

$$\begin{aligned} u_{t+1} &= \beta u_t + (1 - \beta) \nabla_{\theta} f(\theta_t) \\ \theta_{t+1} &= \theta_t - \alpha u_t \end{aligned} \quad \Rightarrow \quad \begin{aligned} u_{t+1} &= \beta u_t + (1 - \beta) \nabla_{\theta} f(\theta_t - \alpha u_t) \\ \theta_{t+1} &= \theta_t - \alpha u_t \end{aligned}$$

A “good” thing for convex optimization, and (sometimes) helps for deep networks



# Adam

The *scale* of the gradients can vary widely for different parameters, especially e.g. across different layers of a deep network, different layer types, etc

So-called *adaptive gradient* methods attempt to estimate this scale over iterations and then re-scale the gradient update accordingly

Most widely used adaptive gradient method for deep learning is Adam algorithm, which combines momentum and *adaptive scale estimation*

$$u_{t+1} = \beta_1 u_t + (1 - \beta_1) \nabla_{\theta} f(\theta_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) (\nabla_{\theta} f(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \alpha u_{t+1} / (v_{t+1}^{1/2} + \epsilon)$$

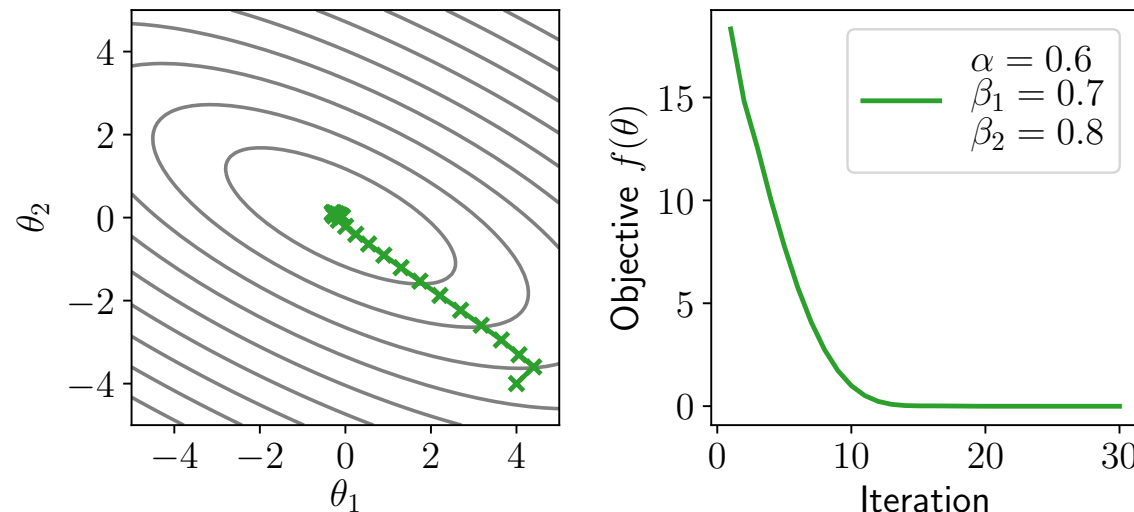
(Common to use unbiased momentum estimated for both terms)



# Notes on / illustration of Adam

Whether Adam is “good” optimizer is endlessly debated within deep learning, but it often seems to work quite well in practice (maybe?)

There are alternative universes where endless other variants became the “standard” (no unbiasing? average of absolute magnitude rather than squared? Nesterov-like acceleration?) but Adam is well-tuned and hard to uniformly beat



# Stochastic variants

All the previous examples considered *batch* update to the parameters, but the single most important optimization choice is to use **stochastic variants**

Recall our machine learning optimization problem

$$\underset{\theta}{\text{minimize}} \quad \frac{1}{m} \sum_{i=1}^m \ell(h_{\theta}(x^{(i)}), y^{(i)})$$

which is the minimization of an empirical *expectation* over losses

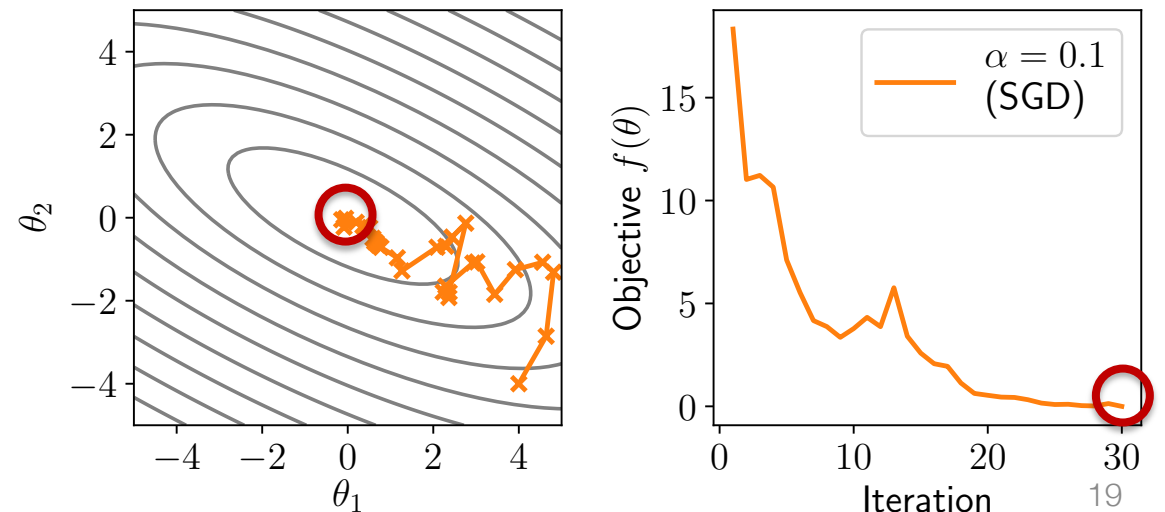
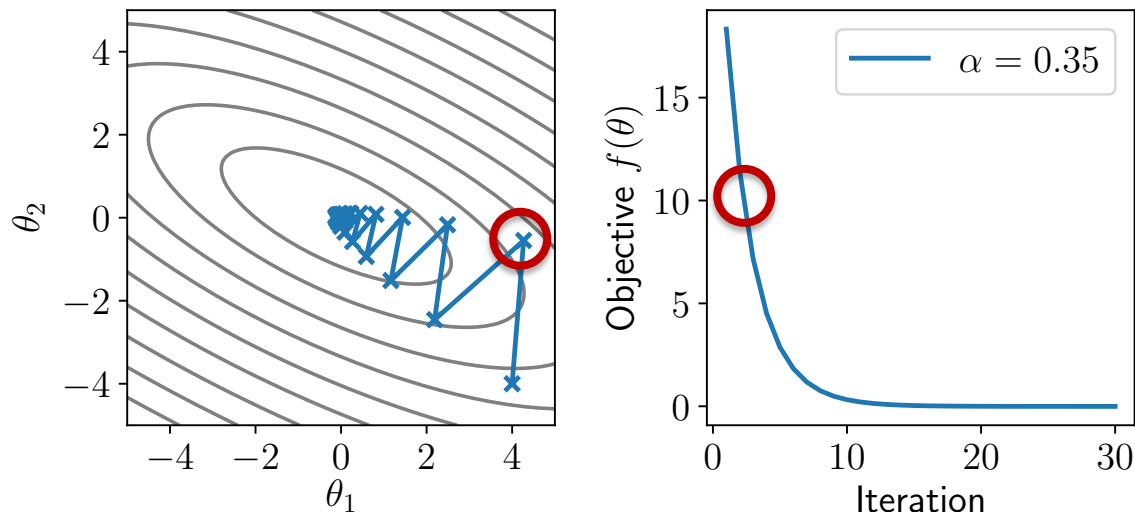
We can get a **noisy (but unbiased) estimate of gradient by computing gradient of the loss over just a subset of examples (called a minibatch)**

# Stochastic Gradient Descent

This leads us again to the SGD algorithm, repeating for batches  $B \subset \{1, \dots, m\}$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{B} \sum_{i \in B} \nabla_{\theta} \ell(h(x^{(i)}), y^{(i)})$$

Instead of taking a few expensive, noise-free, steps, we take *many* cheap, noisy steps, which ends having much strong performance per compute



# The most important takeaways

*All* the optimization methods you have seen thus far presented are *only* actually used in their stochastic form

The amount of valid intuition about these optimization methods you will get from looking at simple (convex, quadratic) optimization problems is limited

You need to constantly experiment to gain an understanding / intuition of how these methods actually affect deep networks of different types

# Outline

Fully connected networks

Optimization

Initialization

# Initialization of weights

Recall that we optimize parameters iteratively by stochastic gradient descent, e.g.

$$W_i := W_i - \alpha \nabla_{W_i} \ell(h_\theta(X), y)$$

But how do we choose the *initial* values of  $W_i$ ,  $b_i$ ? (maybe just initialize to zero?)

Recall the manual backpropagation forward/backward passes (without bias):

$$Z_{i+1} = \sigma_i(Z_i W_i)$$

$$G_i = \left( G_{i+1} \circ \sigma'_i(Z_i W_i) \right) W_i^T$$

- If  $W_i = 0$ , then  $G_j = 0$  for  $j \leq i$ ,  $\implies \nabla_{W_i} \ell(h_\theta(X), y) = 0$
- I.e.,  $W_i = 0$  is a (very bad) local optimum of the objective

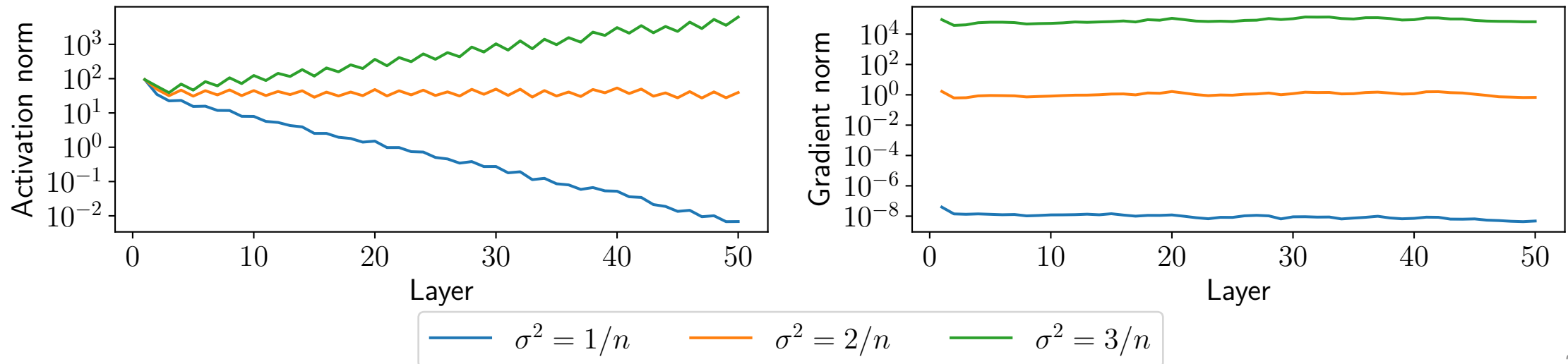
# Key idea #1: Choice of initialization matters

Let's just initialize weights “randomly”, e.g.,  $W_i \sim \mathcal{N}(0, \sigma^2 I)$

The choice of variance  $\sigma^2$  will affect two (related) quantities:

1. The norm of the forward activations  $Z_i$
2. The norm of the the gradients  $\nabla_{W_i} \ell(h_\theta(X), y)$

Illustration on MNIST  
with  $n = 100$  hidden  
units, depth 50,  
ReLU nonlinearities



## Key idea #2: Weights don't move “that much”

Might have the picture in your mind that the parameters of a network converge to some similar region of points regardless of their initialization

This is not true ... weights often stay much closer to their initialization than to the “final” point after optimization from different

End result: initialization matters ... we'll see some of the practical aspects next lecture



# What causes these effects?

Consider independent random variables  $x \sim \mathcal{N}(0,1)$ ,  $w \sim \mathcal{N}(0, \frac{1}{n})$ ; then

$$\mathbf{E}[x_i w_i] = \mathbf{E}[x_i] \mathbf{E}[w_i] = 0, \quad \mathbf{Var}[x_i w_i] = \mathbf{Var}[x_i] \mathbf{Var}[w_i] = 1/n$$

so  $\mathbf{E}[w^T x] = 0$ ,  $\mathbf{Var}[w^T x] = 1$  ( $w^T x \rightarrow \mathcal{N}(0,1)$  by **central limit theorem**)

Thus, informally speaking if we used a linear activation and  $z_i \sim \mathcal{N}(0, I)$ ,  $W_i \sim \mathcal{N}(0, \frac{1}{n} I)$  then  $z_{i+1} = W_i^T z_i \sim \mathcal{N}(0, I)$

If we use a ReLU nonlinearity, then “half” the components of  $z_i$  will be set to zero, so we need twice the variance on  $W_i$  to achieve the same final variance, hence  $W_i \sim \mathcal{N}(0, \frac{2}{n} I)$

# What causes these effects?

If  $\sigma^2 \neq 2/n$  for a ReLU network, then each iteration the variance of the the hidden units increases/decreases geometrically by some factor

$$\mathbf{Var}[Z_{i+1}] = \gamma \cdot \mathbf{Var}[Z_i] = \gamma^i$$

This happens for both the forward *and* backward passes

$$\mathbf{Var}[G_i] = \gamma \cdot \mathbf{Var}[G_{i+1}] = \gamma^{D-i}$$

Thus, we have that (naively), the variance of the gradients will be approximately *constant over layers*, but heavily affected by choice of  $\sigma^2$ , depth  $D$

$$\nabla_{W_i} \ell(h_\theta(X), y) \propto Z_i^T G_i \propto \gamma^D$$