CS/DS 541: Class 15

Jacob Whitehill

Optimization

Optimization

- Like in the *Deep Learning* book, we define **optimization** as the algorithmic tools to help neural network training to reach a *lower loss value* during training.

Momentum

- SGD can suffer due to:
  - Noisy gradient estimates cause the weights to move in the wrong direction.
  - Slow convergence due to ill-conditioned loss function.
- Momentum is a commonly used technique to lessen these problems.

Momentum

- In SGD, instead of updating the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

we update them as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}), \quad \alpha \in [0, 1)$$

Momentum

- In SGD, instead of updating the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

  we update them as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}), \quad \alpha \in [0, 1)$$

Previous steps

- The actual weight update is a combination of a “moving average” of previous steps plus the current gradient estimate. *α* expresses how much we trust the average.

Momentum

- In SGD, instead of updating the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

  we update them as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

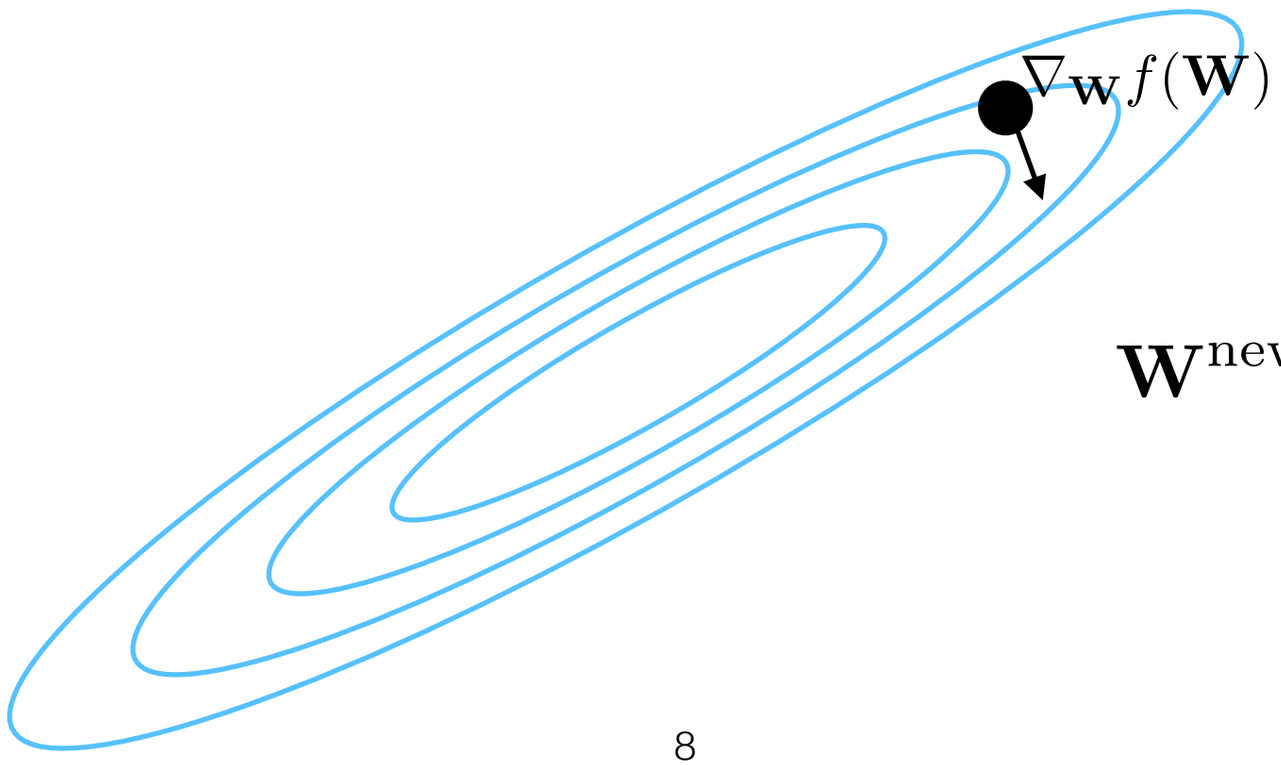
$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}), \quad \alpha \in [0, 1)$$

Current gradient estimate

- The actual weight update is a combination of a “moving average” of previous steps plus the current gradient estimate. *α* expresses how much we trust the average.

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

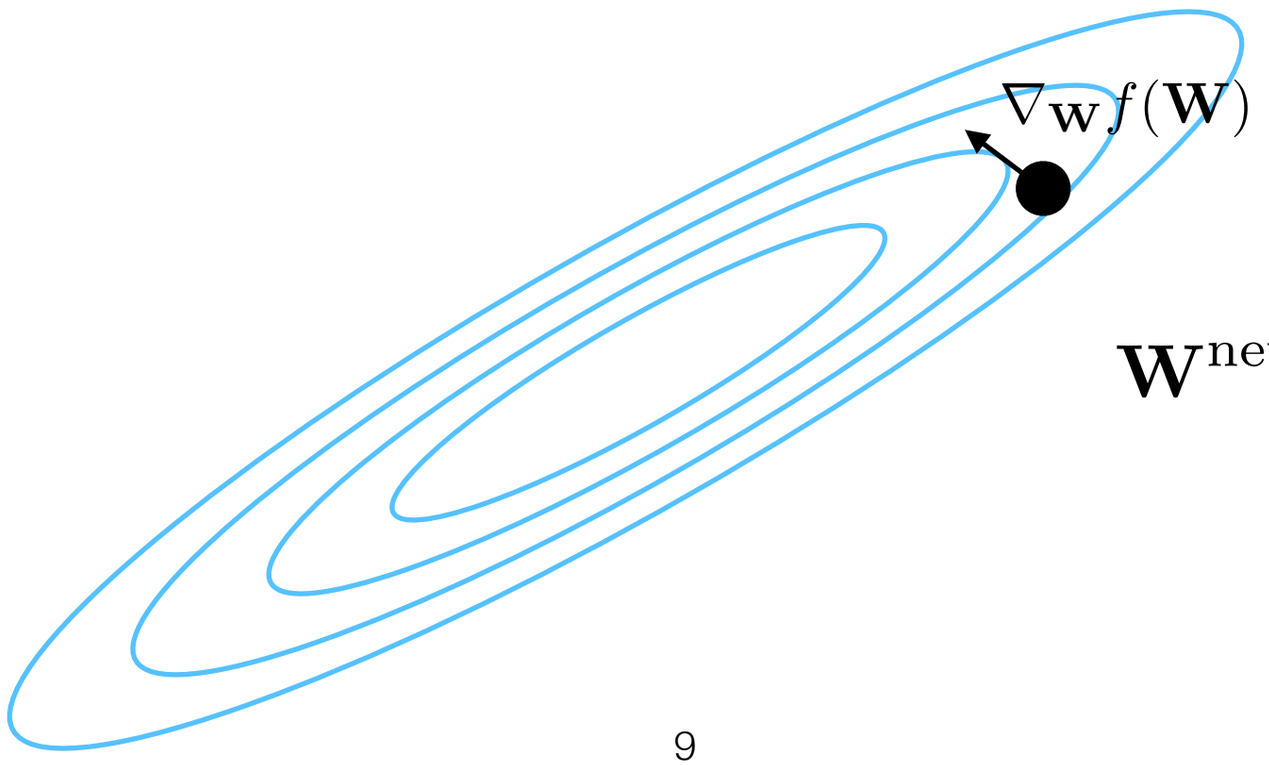


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

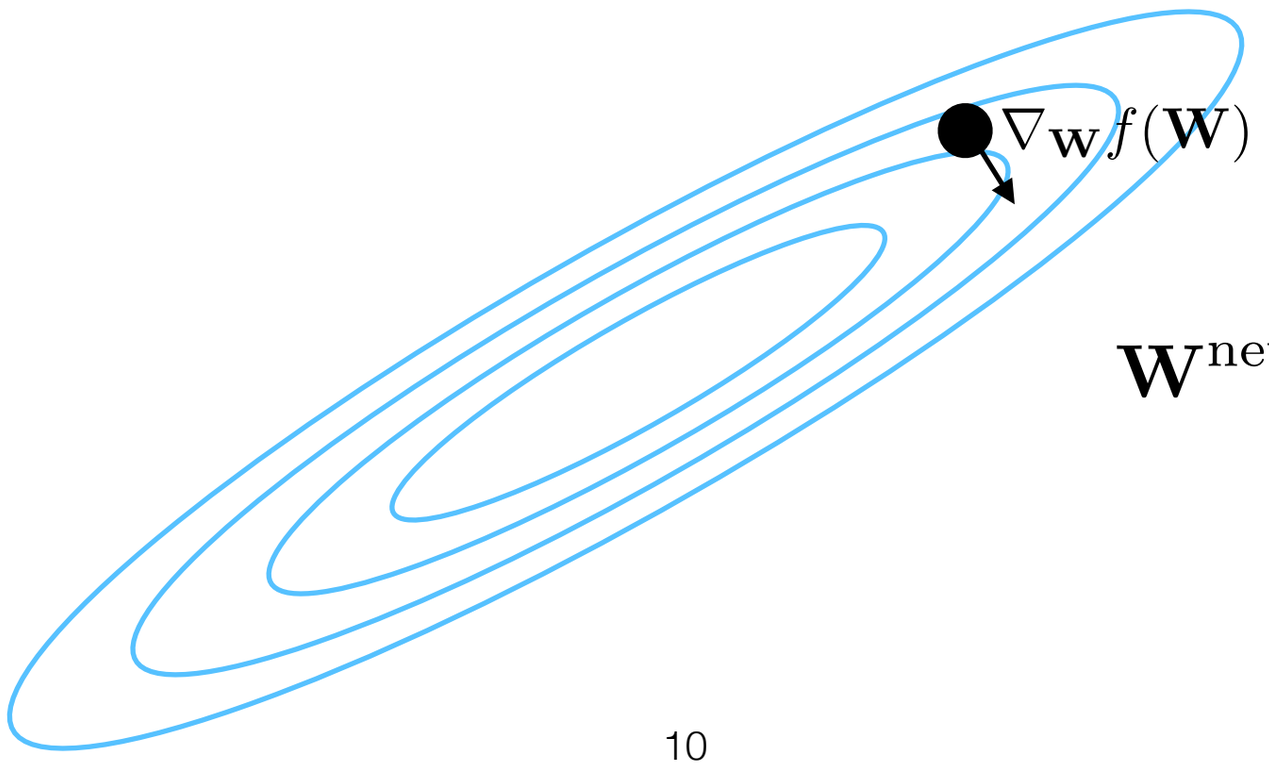


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

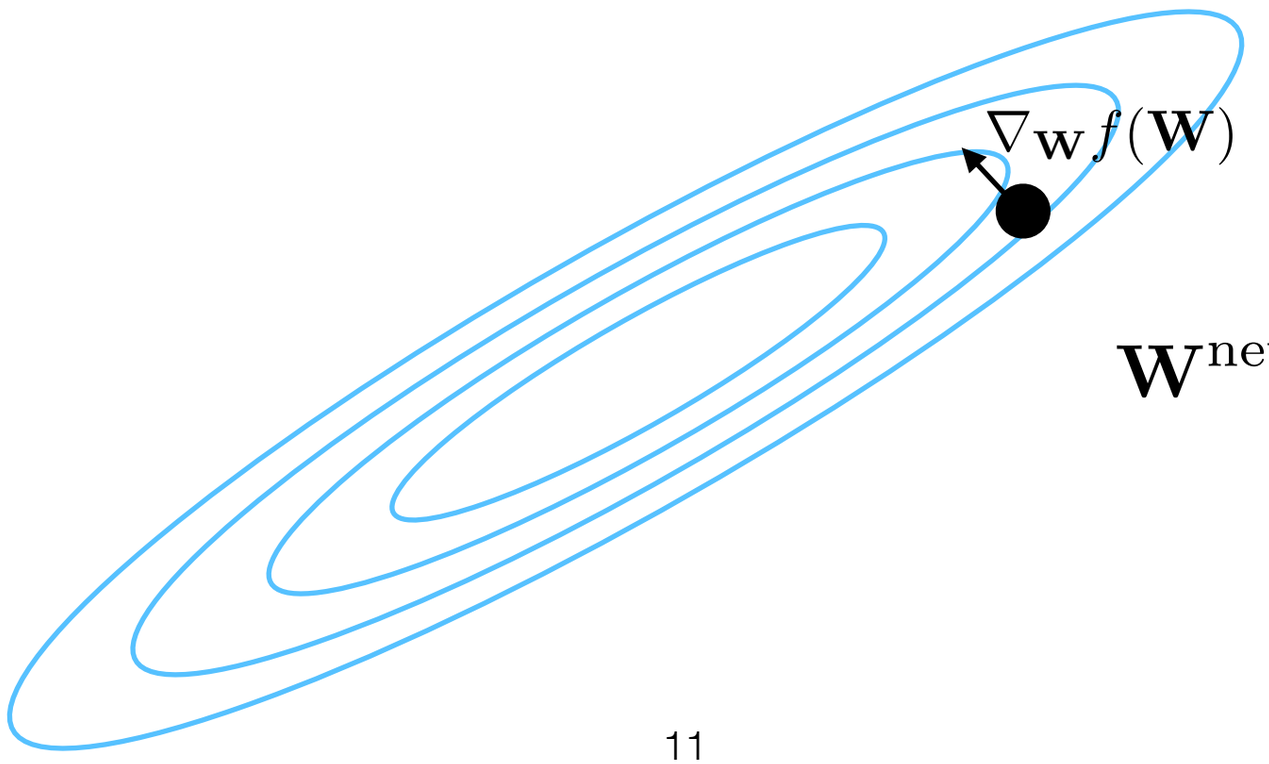


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

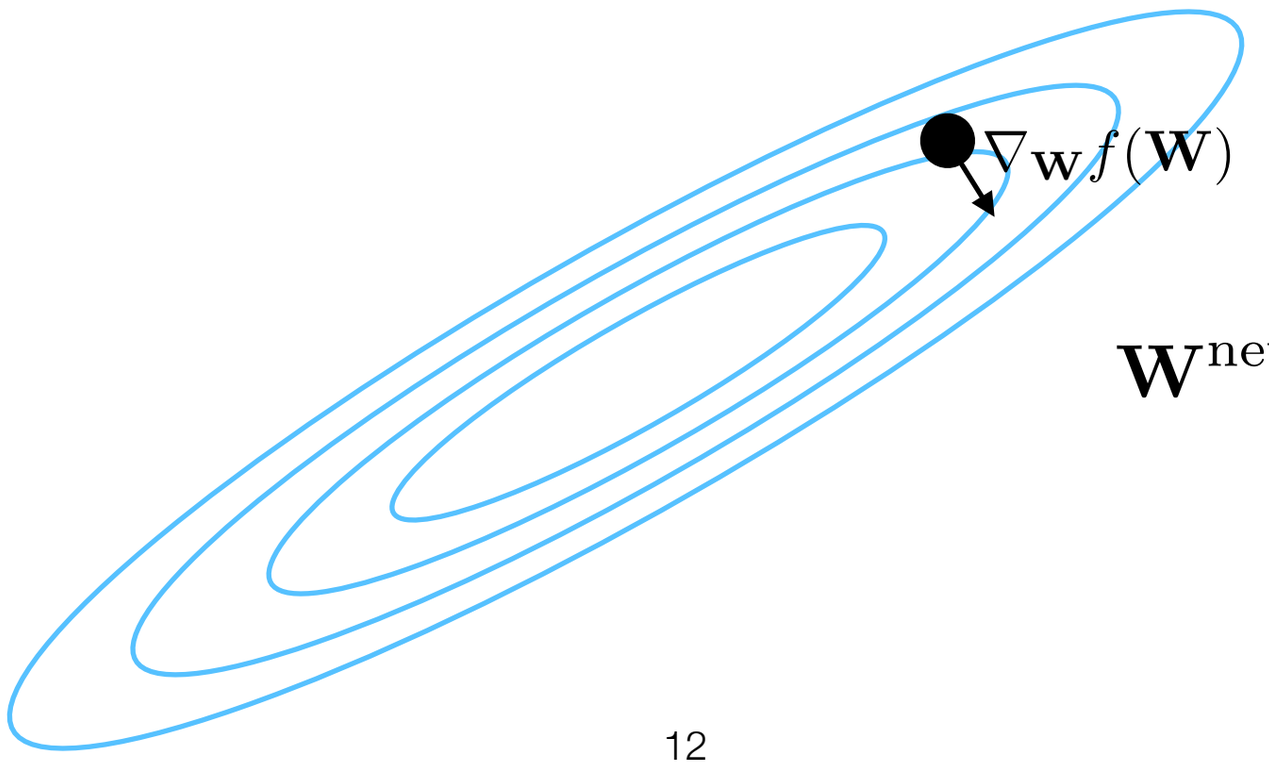


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

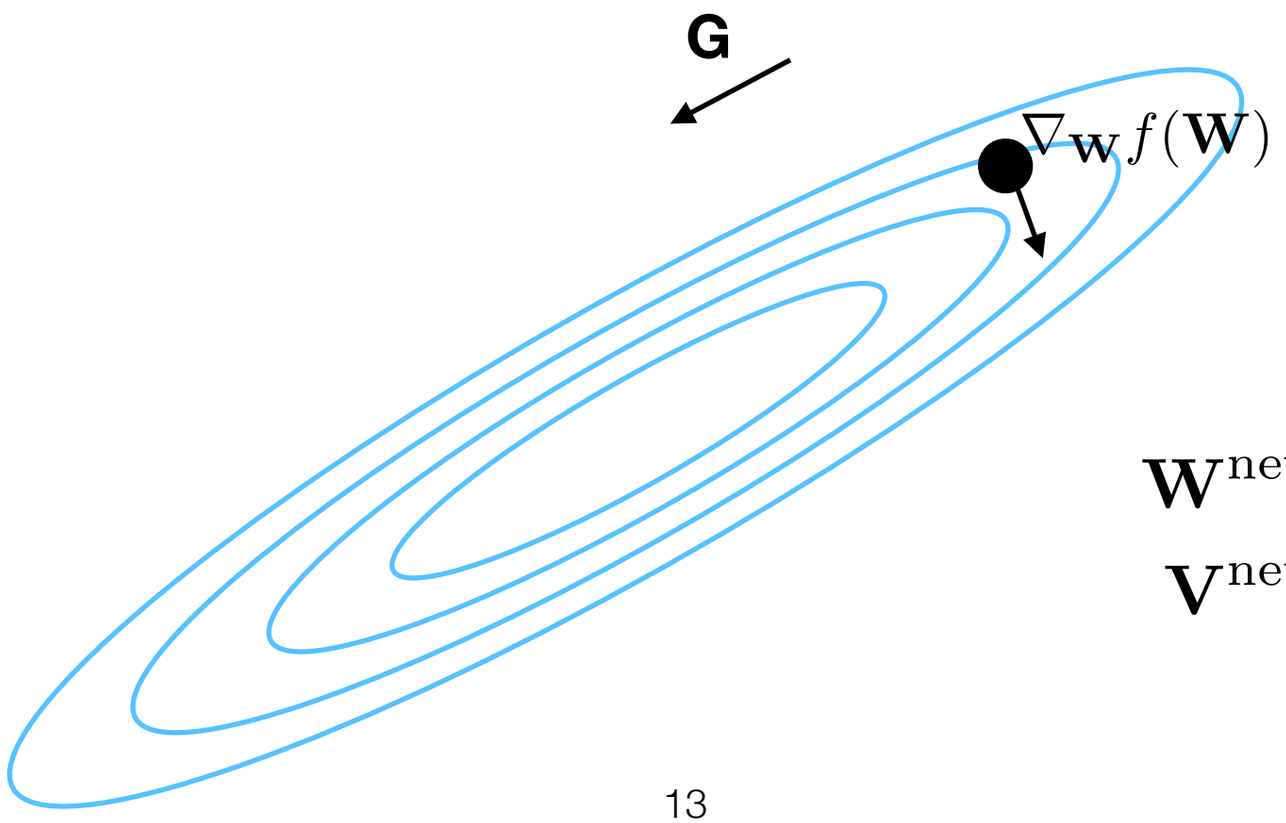


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



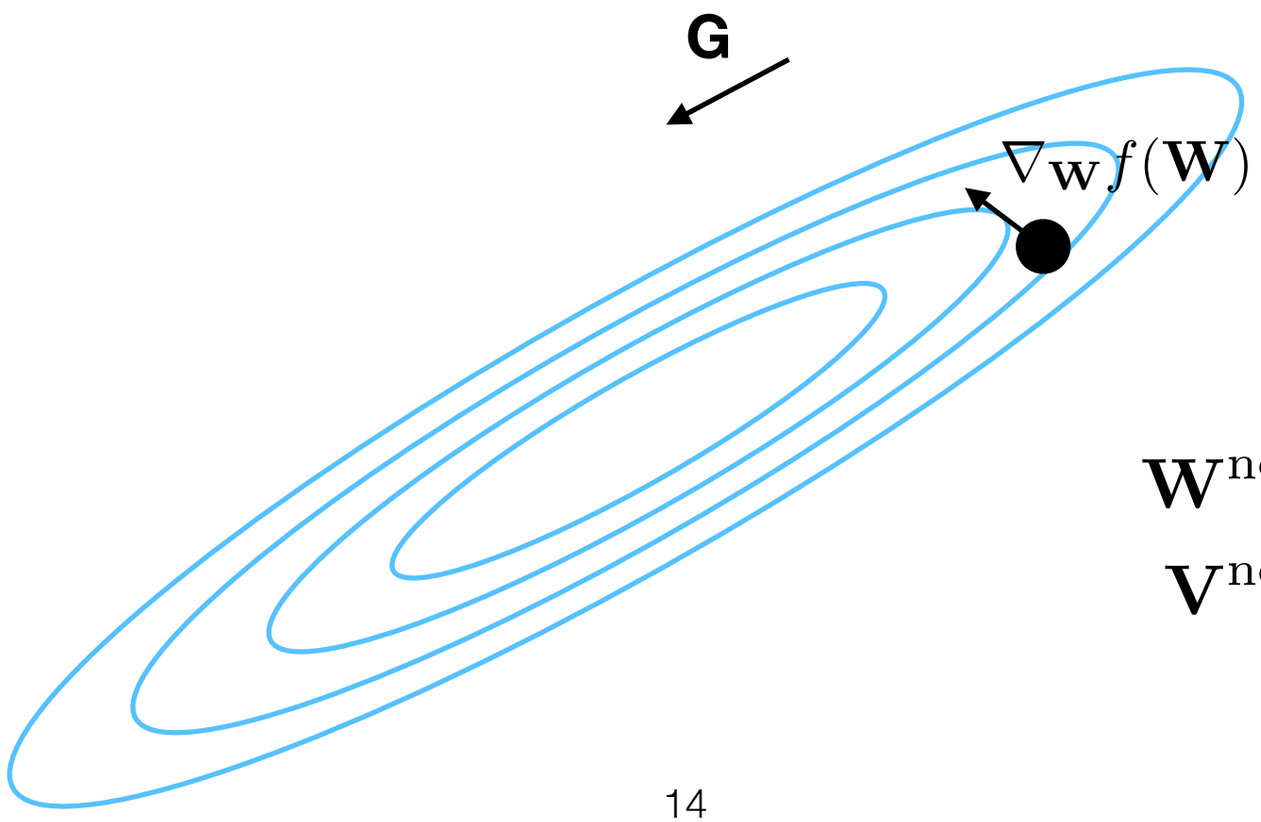
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha\mathbf{V} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



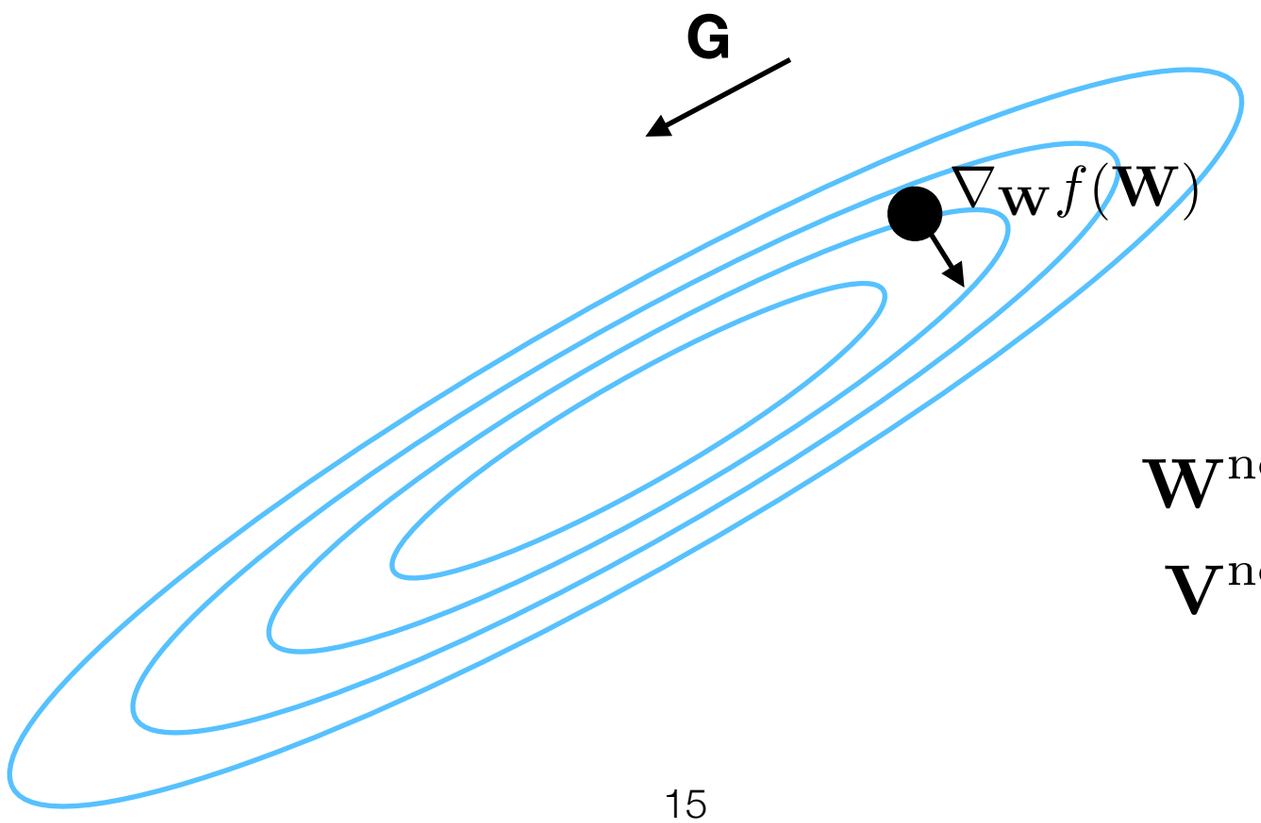
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha\mathbf{V} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



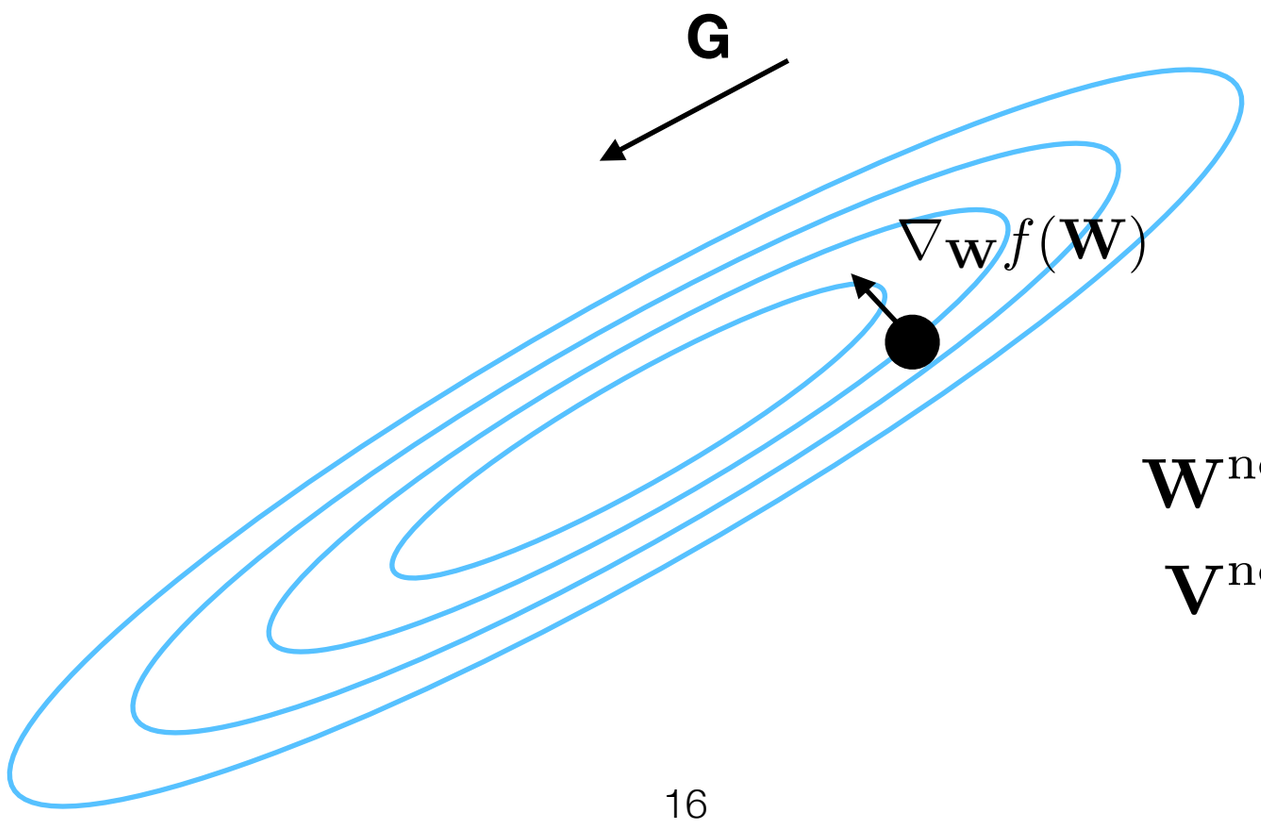
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha\mathbf{V} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



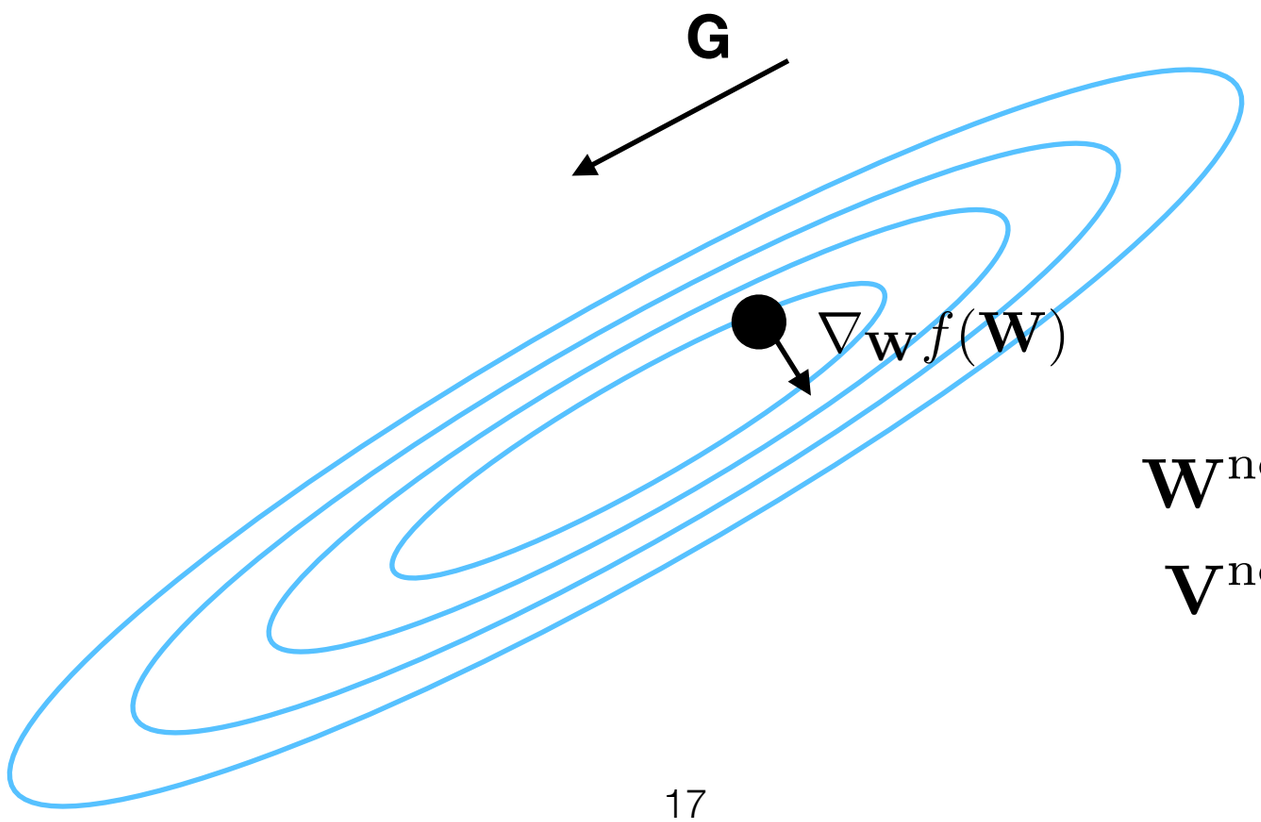
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- We can interpret α as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$\mathbf{v}^{(0)} = 0$$

Momentum

- We can interpret *α* as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$\mathbf{V}^{(0)} = 0$$

$$\mathbf{V}^{(1)} = \alpha \mathbf{V}^{(0)} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- We can interpret *α* as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$\begin{aligned}
\mathbf{V}^{(0)} &= 0 \\
\mathbf{V}^{(1)} &= \alpha \mathbf{V}^{(0)} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}) \\
&= \alpha \mathbf{V}^{(0)} + \epsilon \mathbf{G}
\end{aligned}$$

Momentum

- We can interpret *α* as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$
\begin{aligned}
\mathbf{V}^{(0)} &= 0 \\
\mathbf{V}^{(1)} &= \alpha \mathbf{V}^{(0)} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}) \\
&= \alpha \mathbf{V}^{(0)} + \epsilon \mathbf{G} \\
\mathbf{V}^{(2)} &= \alpha(\alpha \mathbf{V}^{(0)} + \epsilon \mathbf{G}) + \epsilon \mathbf{G}
\end{aligned}
$$

Momentum

- We can interpret *α* as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$
\begin{aligned}
\mathbf{V}^{(0)} &= 0 \\
\mathbf{V}^{(1)} &= \alpha\mathbf{V}^{(0)} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W}) \\
&= \alpha\mathbf{V}^{(0)} + \epsilon\mathbf{G} \\
\mathbf{V}^{(2)} &= \alpha(\alpha\mathbf{V}^{(0)} + \epsilon\mathbf{G}) + \epsilon\mathbf{G} \\
\mathbf{V}^{(3)} &= \alpha(\alpha(\mathbf{V}^{(0)} + \epsilon\mathbf{G}) + \epsilon\mathbf{G}) + \epsilon\mathbf{G} \\
&\vdots \\
\mathbf{V}^{(n)} &= \alpha^n\mathbf{V}^{(0)} + \alpha^{n-1}\epsilon\mathbf{G} + \ldots + \alpha^1\epsilon\mathbf{G} + \alpha^0\epsilon\mathbf{G}
\end{aligned}
$$

Momentum

- We can interpret *α* as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$
\begin{aligned}
\mathbf{V}^{(0)} &= 0 \\
\mathbf{V}^{(1)} &= \alpha\mathbf{V}^{(0)} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W}) \\
&= \alpha\mathbf{V}^{(0)} + \epsilon\mathbf{G} \\
\mathbf{V}^{(2)} &= \alpha(\alpha\mathbf{V}^{(0)} + \epsilon\mathbf{G}) + \epsilon\mathbf{G} \\
\mathbf{V}^{(3)} &= \alpha(\alpha(\mathbf{V}^{(0)} + \epsilon\mathbf{G}) + \epsilon\mathbf{G}) + \epsilon\mathbf{G} \\
&\vdots \\
\mathbf{V}^{(n)} &= \alpha^n\mathbf{V}^{(0)} + \alpha^{n-1}\epsilon\mathbf{G} + \ldots + \alpha^1\epsilon\mathbf{G} + \alpha^0\epsilon\mathbf{G} \\
&= \sum_{i=0}^{n-1} \alpha^i \epsilon\mathbf{G} \\
&= \epsilon\mathbf{G} \sum_{i=0}^{n-1} \alpha^i
\end{aligned}
$$

Momentum

- We can interpret *α* as determining the maximum acceleration factor due to momentum. Suppose that *all* gradient estimates point in the same direction **G**. Then:

$$
\begin{aligned}
\mathbf{V}^{(0)} &= 0 \\
\mathbf{V}^{(1)} &= \alpha\mathbf{V}^{(0)} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W}) \\
&= \alpha\mathbf{V}^{(0)} + \epsilon\mathbf{G} \\
\mathbf{V}^{(2)} &= \alpha(\alpha\mathbf{V}^{(0)} + \epsilon\mathbf{G}) + \epsilon\mathbf{G} \\
\mathbf{V}^{(3)} &= \alpha(\alpha(\mathbf{V}^{(0)} + \epsilon\mathbf{G}) + \epsilon\mathbf{G}) + \epsilon\mathbf{G} \\
&\vdots \\
\mathbf{V}^{(n)} &= \alpha^n\mathbf{V}^{(0)} + \alpha^{n-1}\epsilon\mathbf{G} + \ldots + \alpha^1\epsilon\mathbf{G} + \alpha^0\epsilon\mathbf{G} \\
&= \sum_{i=0}^{n-1} \alpha^i \epsilon\mathbf{G} \\
&= \epsilon\mathbf{G} \sum_{i=0}^{n-1} \alpha^i \\
\implies \lim_{n\to\infty} \mathbf{V}^{(n)} &= \frac{\epsilon\mathbf{G}}{1-\alpha}
\end{aligned}
$$

Momentum

- Hence, if $\alpha = 0.99$, then the maximum acceleration is 1/(1 - 0.99) = 100.

$$
\begin{aligned}
\mathbf{V}^{(0)} &= 0 \\
\mathbf{V}^{(1)} &= \alpha \mathbf{V}^{(0)} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}) \\
&= \alpha \mathbf{V}^{(0)} + \epsilon \mathbf{G} \\
\mathbf{V}^{(2)} &= \alpha(\alpha \mathbf{V}^{(0)} + \epsilon \mathbf{G}) + \epsilon \mathbf{G} \\
\mathbf{V}^{(3)} &= \alpha(\alpha(\mathbf{V}^{(0)} + \epsilon \mathbf{G}) + \epsilon \mathbf{G}) + \epsilon \mathbf{G} \\
&\vdots \\
\mathbf{V}^{(n)} &= \alpha^n \mathbf{V}^{(0)} + \alpha^{n-1} \epsilon \mathbf{G} + \ldots + \alpha^1 \epsilon \mathbf{G} + \alpha^0 \epsilon \mathbf{G} \\
&= \sum_{i=0}^{n-1} \alpha^i \epsilon \mathbf{G} \\
&= \epsilon \mathbf{G} \sum_{i=0}^{n-1} \alpha^i \\
\implies \lim_{n \to \infty} \mathbf{V}^{(n)} &= \frac{\epsilon \mathbf{G}}{1 - \alpha}
\end{aligned}
$$

Second order optimization

- Recall Newton's method:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \mathbf{H}^{-1} \nabla_{\mathbf{w}} f\left(\mathbf{w}^{(k)}\right)$$

- It requires computing the *inverse* of the Hessian matrix of the loss function w.r.t. the weights **w**.

- This takes time $O(m^3)$ and is thus impractical in most NNs.

Approximations to second-order methods

- However, several NN optimization methods try to approximate it to reach a local minimum more quickly.

- Several are based on the **Hessian outer-product approximation** (see Bishop 1995, *Neural Networks for Pattern Recognition*, for an explanation):

$$\mathbf{H}(\mathbf{w}) \approx \mathbb{E}\left[\left(\nabla_{\mathbf{w}} f(\mathbf{w})\right)\left(\nabla_{\mathbf{w}} f(\mathbf{w})\right)^{\top}\right]$$

$$= \mathbb{E}\begin{bmatrix} (\nabla_{w_1} f)(\nabla_{w_1} f) & \dots & (\nabla_{w_1} f)(\nabla_{w_m} f) \\ \vdots & \ddots & \dots \\ (\nabla_{w_m} f)(\nabla_{w_1} f) & \dots & (\nabla_{w_m} f)(\nabla_{w_m} f) \end{bmatrix}$$

Approximations to second-order methods

- However, several NN optimization methods try to approximate it to reach a local minimum more quickly.

- Several are based on the **Hessian outer-product approximation** (see Bishop 1995, *Neural Networks for Pattern Recognition*, for an explanation):

$$\mathbf{H}(\mathbf{w}) \approx \mathbb{E}\left[\left(\nabla_{\mathbf{w}} f(\mathbf{w})\right)\left(\nabla_{\mathbf{w}} f(\mathbf{w})\right)^{\top}\right]$$

- I.e., the Hessian can be approximated over many examples by the outer product of the gradient with itself.

Approximations to second-order methods

- We can make a further approximation by computing just the diagonal terms, i.e.:

$$\mathbf{H}(\mathbf{w}) \approx \mathbb{E}\begin{bmatrix} (\nabla_{w_1} f)^2 & \dots & 0 \\ 0 & (\nabla_{w_j} f)^2 & 0 \\ 0 & 0 & (\nabla_{w_m} f)^2 \end{bmatrix}$$
$$= \mathbb{E}[\mathrm{diag}[(\nabla_{\mathbf{w}} f) \odot (\nabla_{\mathbf{w}} f)]]$$

Approximations to second-order methods

- We can make a further approximation by computing just the diagonal terms, i.e.:

$$\mathbf{H}(\mathbf{w}) \approx \mathbb{E}\begin{bmatrix} (\nabla_{w_1} f)^2 & \dots & 0 \\ 0 & (\nabla_{w_j} f)^2 & 0 \\ 0 & 0 & (\nabla_{w_m} f)^2 \end{bmatrix}$$
$$= \mathbb{E}[\mathrm{diag}[(\nabla_{\mathbf{w}} f) \odot (\nabla_{\mathbf{w}} f)]]$$

- Computing the inverse approximate Hessian is then easy — just invert each element of the diagonal.

RMSProp, Adagrad, and Adam

- Over the past 10 years, three similar approximately second-order optimization methods have emerged for DL:
  - RMSProp (Hinton et al. 2012)
  - Adagrad (**Ada**ptive **grad**ient; Duchi et al. 2011)
  - Adam (**Ada**ptive **m**omentum; Kingma & Ba 2014)

- They all are based on the approximation above.

RMSProp

- RMSProp normalizes the gradient by a *sum* **r** involving the diagonal outer-product Hessian approximation:

$$w_j^{\text{new}} = w_j - \frac{\epsilon}{\delta + \sqrt{r_j}} \nabla_{w_j} f$$

$$\mathbf{r}^{\text{new}} = \mathbf{r} + (\nabla_{\mathbf{w}} f) \odot (\nabla_{\mathbf{w}} f)$$

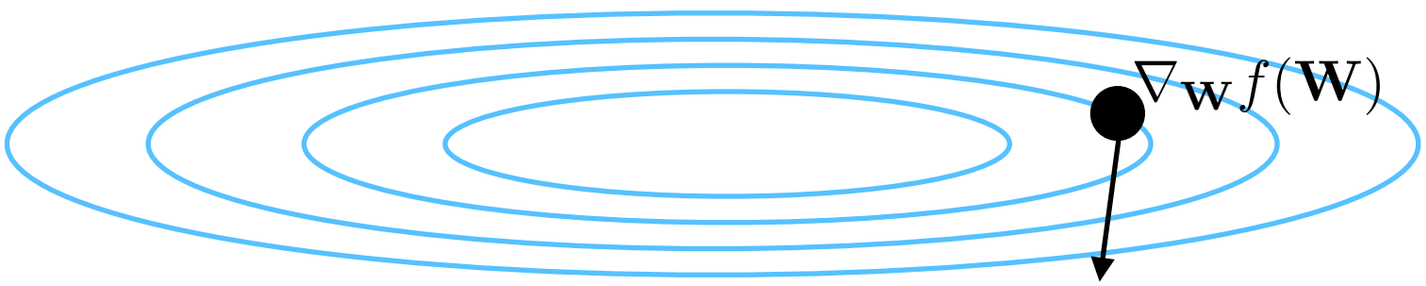
RMSProp

- RMSProp normalizes the gradient by a *sum* **r** involving the diagonal outer-product Hessian approximation:

$$w_j^{\text{new}} = w_j - \frac{\epsilon}{\delta + \sqrt{r_j}} \nabla_{w_j} f$$

$$\mathbf{r}^{\text{new}} = \mathbf{r} + (\nabla_{\mathbf{w}} f) \odot (\nabla_{\mathbf{w}} f)$$

- Another interpretation: it moves the weights less along directions with a history of large *squared* gradient.



Adagrad

- Adagrad is similar, but the sum **r** is replaced by a moving average; this can be useful for “forgetting” past information (e.g., from a previous valley).

$$w_j^{\text{new}} = w_j - \frac{\epsilon}{\delta + \sqrt{r_j}} \nabla_{w_j} f$$

$$\mathbf{r}^{\text{new}} = \rho \mathbf{r} + (1 - \rho)(\nabla_{\mathbf{w}} f) \odot (\nabla_{\mathbf{w}} f)$$

Adam

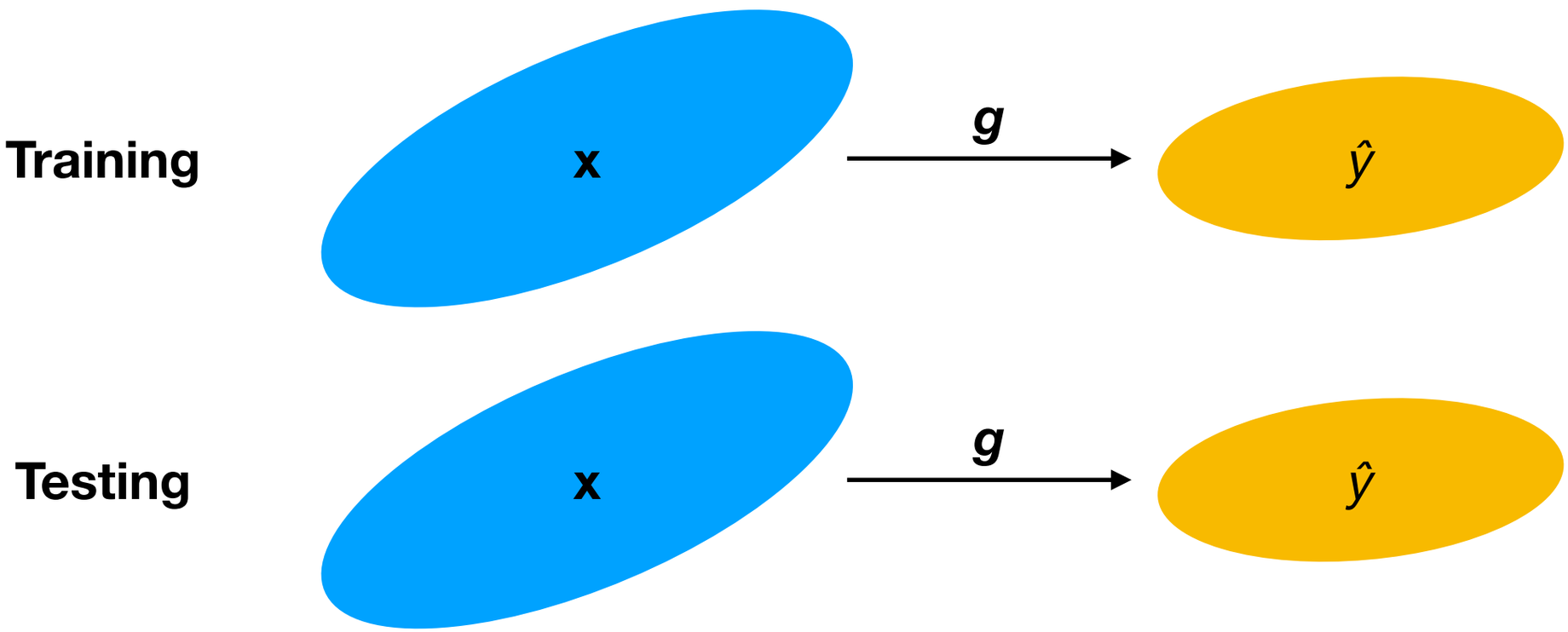
- Adam is similar to both of these but also includes a bias correction term to account for the fact that **r** was initialized at 0. (See Algorithm 8.7 in textbook for more details.)

Batch normalization

Covariate shift

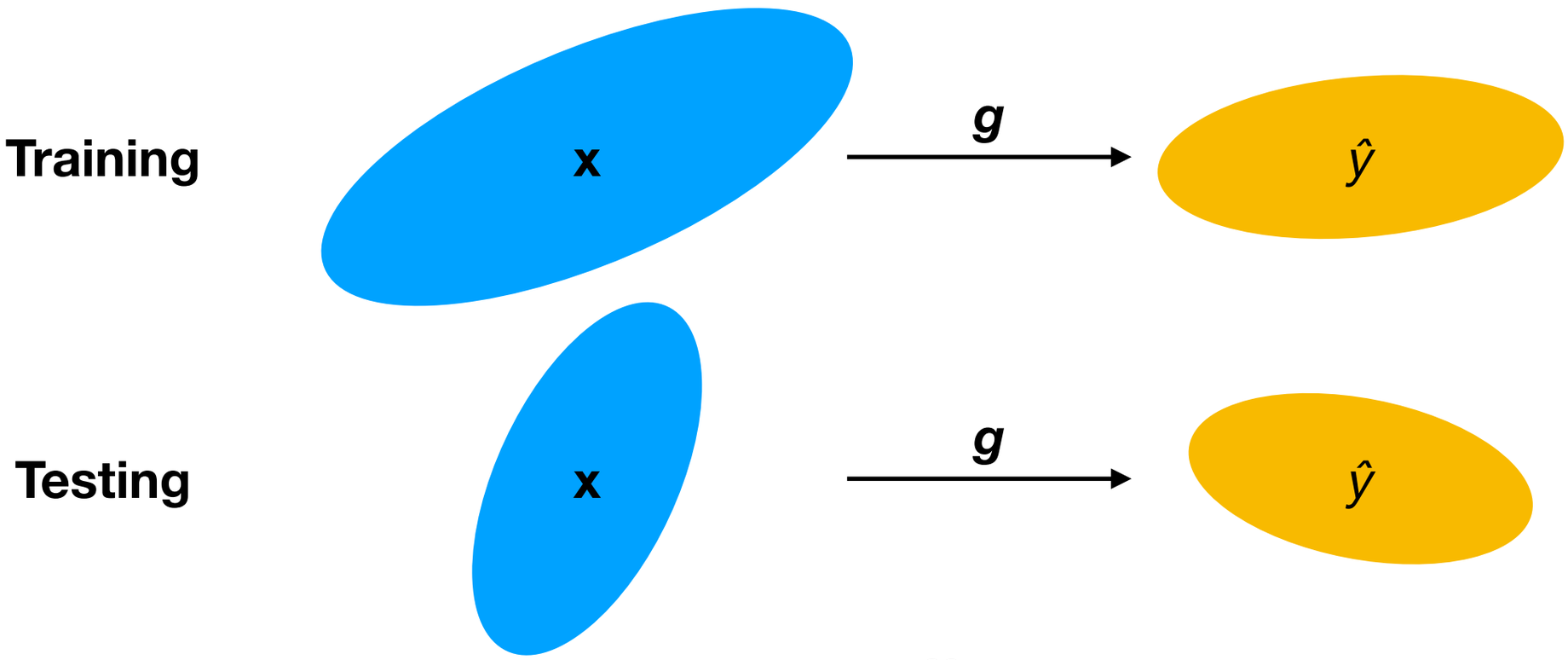
- In classical statistics, the features **x** that a machine uses to make a prediction are sometimes called **covariates**.

- Most of ML assumes that the probability distribution P(**x**) is the same during training and testing.



Covariate shift

- If $P_{tr}(\mathbf{x}) \neq P_{te}(\mathbf{x})$, then we have **covariate shift**.
- The accuracy of the machine will typically decrease as a result.



Covariate shift

- For example, to estimate a person's age from their face, the faces with facial hair might be difficult to classify.

- If facial hair occurs rarely in training, then the ML algorithm can mostly ignore them so as to perform better on un-bearded faces.

- But if the test faces include lots of facial hair, then the classifier may perform very poorly.

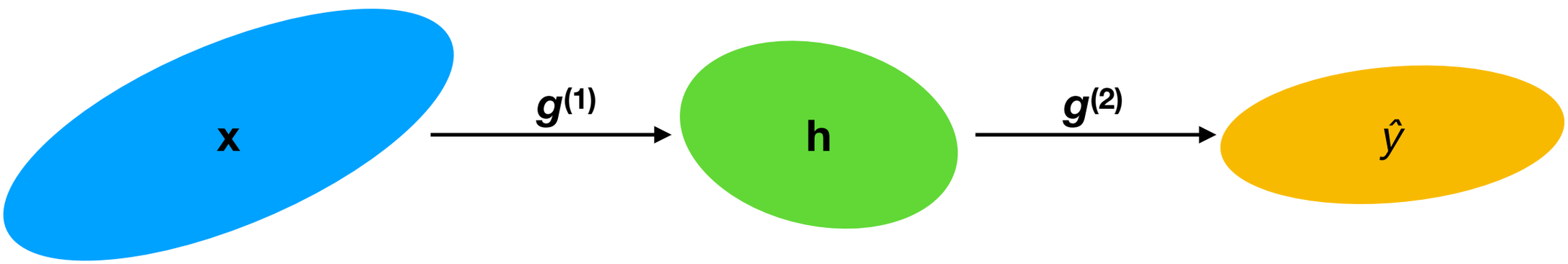
Batch normalization

- In 2015, Ioffe & Szegedy identified a problem when training deep NNs that they called internal covariate shift.

- Instead of occurring between training and testing data, it happens during training between different layers of the network.

Internal covariate shift

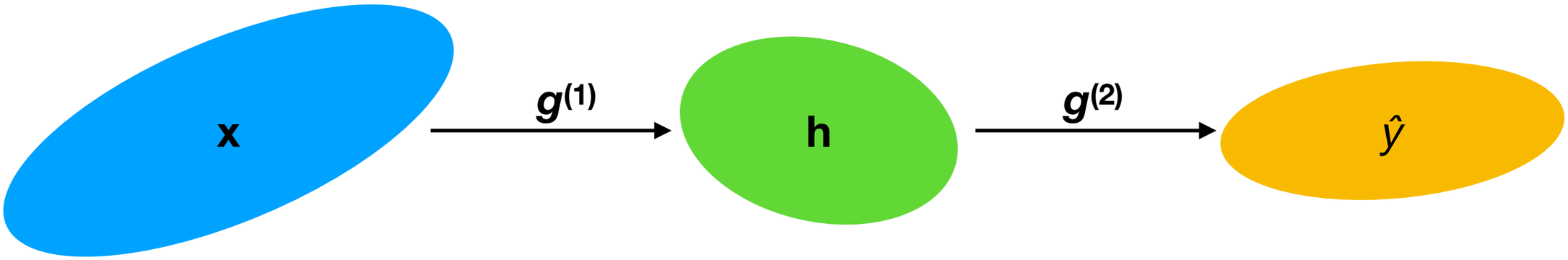
$$\hat{y} = g^{(2)}(g^{(1)}(\mathbf{x}))$$



- Each input **x** is processed by $g^{(1)}$ to compute **h**.
- Each **h** is processed by $g^{(2)}$ to compute $\hat{y}$.
- From the input data distribution P(**x**), we obtain a hidden state distribution P(**h**).
- From the hidden state distribution P(**h**), we obtain an output distribution P($\hat{y}$).

Internal covariate shift

$$\hat{y} = g^{(2)}(g^{(1)}(\mathbf{x}))$$

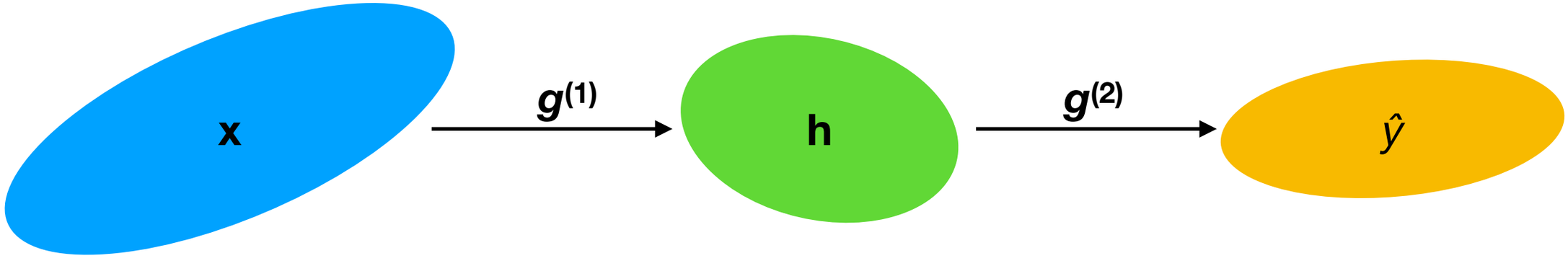


- Each gradient $\frac{\partial f}{\partial g^{(1)}}$ and $\frac{\partial f}{\partial g^{(2)}}$ tells us how to adjust the corresponding parameters to reduce the loss function *f*.

- In particular, $\frac{\partial f}{\partial g^{(2)}}$ tells us how to adjust $g^{(2)}$ so that — *conditioned on receiving* P(**h**) — it produces more accurate $\hat{y}$.

Internal covariate shift

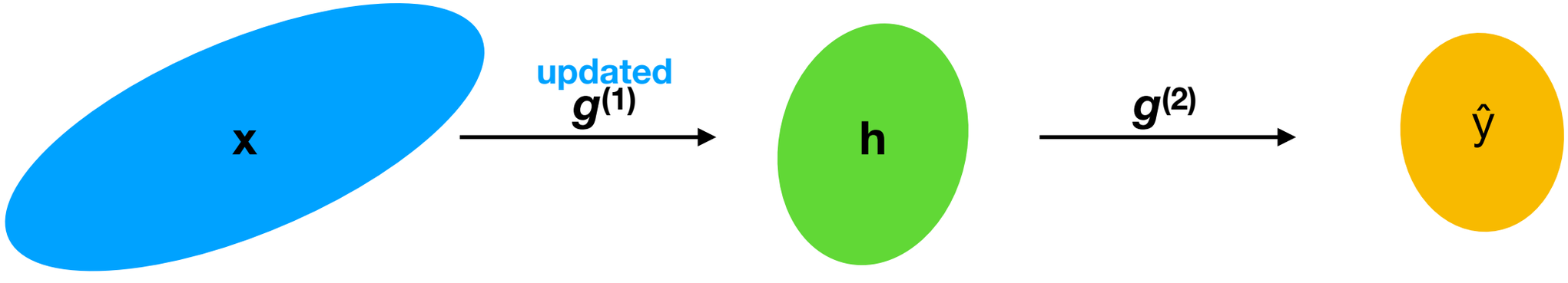
$$\hat{y} = g^{(2)}(g^{(1)}(\mathbf{x}))$$



- But back-propagation will update both $g^{(1)}$ and $g^{(2)}$.

Internal covariate shift

$$\hat{y} = g^{(2)}(g^{(1)}(\mathbf{x}))$$

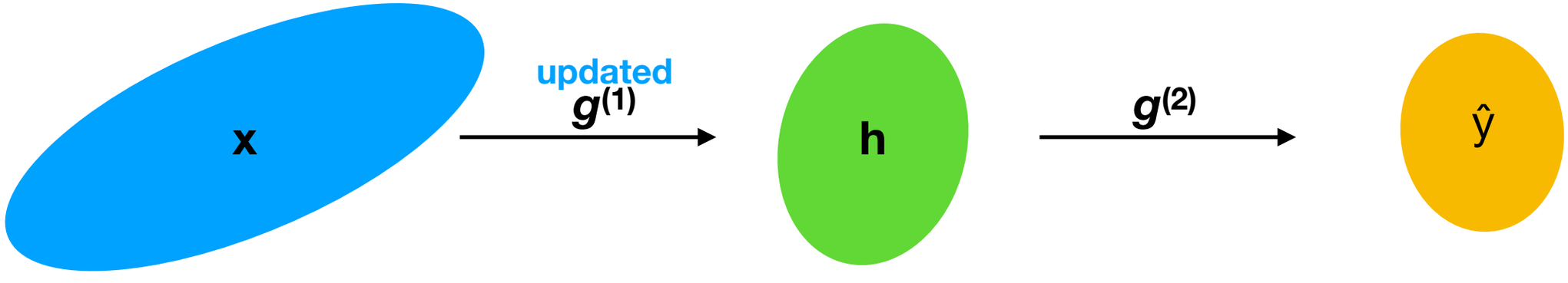


- But back-propagation will update both $g^{(1)}$ and $g^{(2)}$.

- This means that P(**h**), P($\hat{y}$) will be updated based on $\frac{\partial f}{\partial g^{(1)}}$.

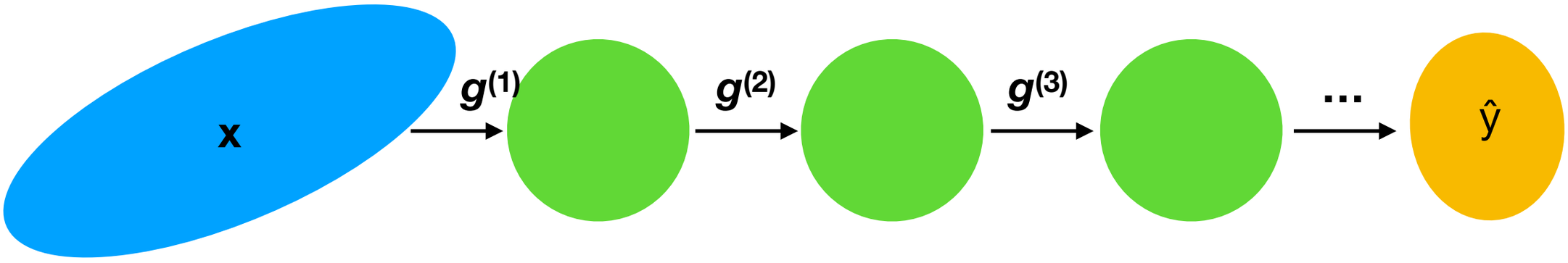
Internal covariate shift

$$\hat{y} = g^{(2)}(g^{(1)}(\mathbf{x}))$$



- But back-propagation will update both $g^{(1)}$ and $g^{(2)}$.
- This means that P(**h**), P($\hat{y}$) will be updated based on $\frac{\partial f}{\partial g^{(1)}}$.
- Is our update to $g^{(2)}$ (based on $\frac{\partial f}{\partial g^{(2)}}$) still valid given that P(**h**) has changed? Arguably it is not; we have **internal covariate shift**.

Batch normalization



- Batch normalization aims to reduce covariate shift by ensuring that the probability distribution at *every* hidden layer is always the same (0-mean, **I**-covariance Gaussian).

- If the distribution is always the same, then no covariate shift will ever occur.

Batch normalization

- How do we ensure that P(**h**) is always a standard normal Gaussian at every hidden layer **h**?

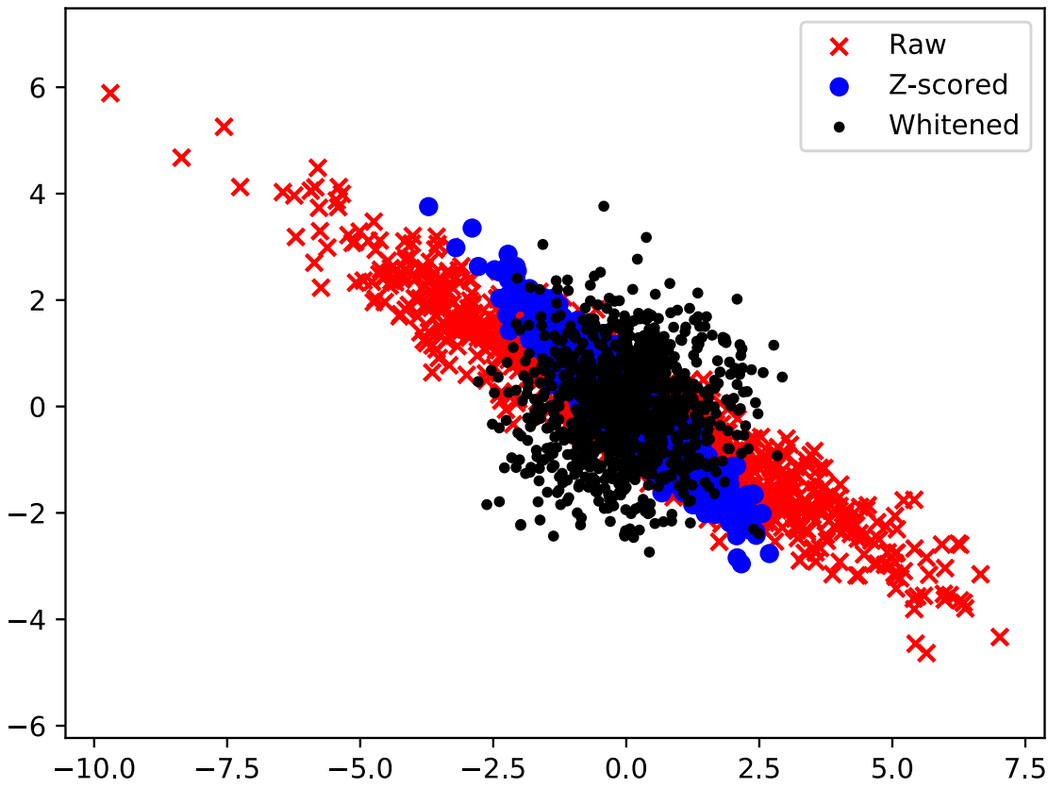
Batch normalization

- How do we ensure that P(**h**) is always a standard normal Gaussian at every hidden layer **h**? **Whiten** the **h** values, e.g., transform **h** by left-multiplying by $\Lambda^{-\frac{1}{2}^\top}\Phi^\top$.

- Unfortunately, a whitening transformation is expensive ($O(m^3)$ for m features) to perform at every SGD step.

z-score approximation

- We can approximate this by z-scoring the distribution P(**h**), i.e., ensuring that the E[**h**]=0 and the diagonal of E[**hh**^{T}] contains just 1s.



Covariances

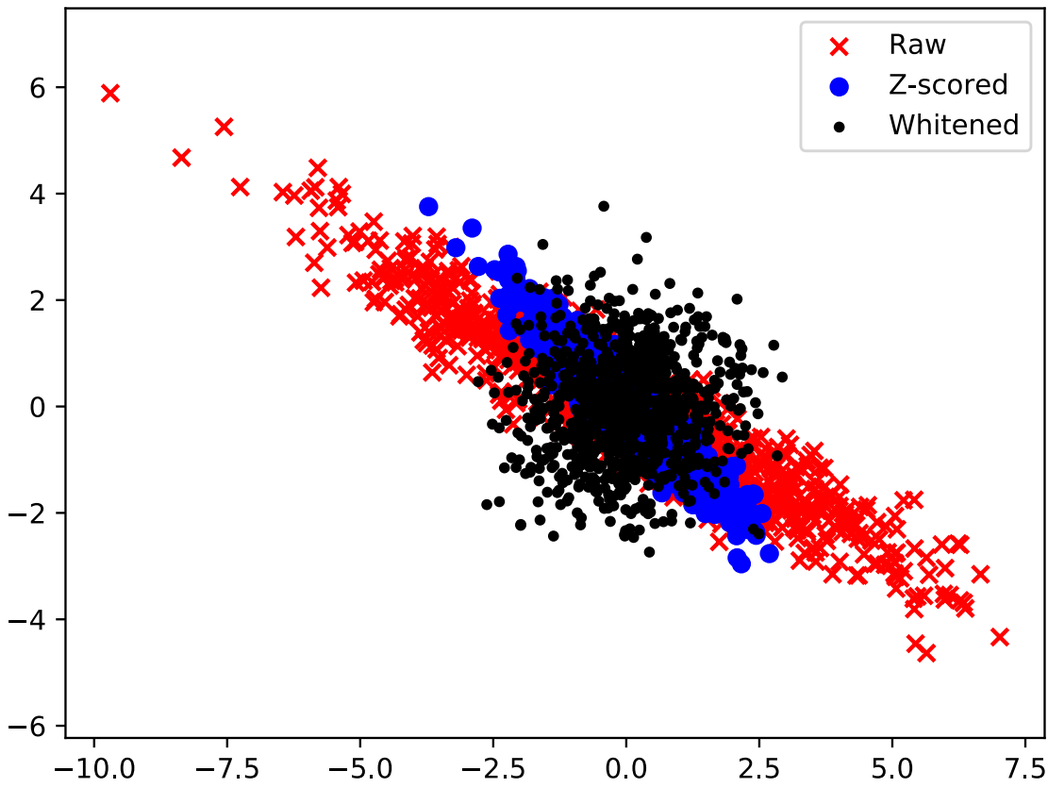
```
Raw:
[[ 6.81764404 -3.86103353]
 [-3.86103353  2.45930235]]

Whitened:
[[9.53471754e-01 1.31581988e-16]
 [1.31581988e-16 9.98898924e-01]]

Z-Scored:
[[ 1.001001   -0.94387604]
 [-0.94387604  1.001001  ]]
```

z-score approximation

- We can z-score a set $\{\mathbf{h}^{(i)}\}_{i=1}^{n}$ by computing:

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}^{(i)}$$

$$\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{h}_j^{(i)} - \mu_j)^2 \quad \forall j \in \{1,\ldots,m\}$$



Covariances

```
Raw:
[[ 6.81764404 -3.86103353]
 [-3.86103353  2.45930235]]

Whitened:
[[9.53471754e-01 1.31581988e-16]
 [1.31581988e-16 9.98898924e-01]]

Z-Scored:
[[ 1.001001   -0.94387604]
 [-0.94387604  1.001001  ]]
```

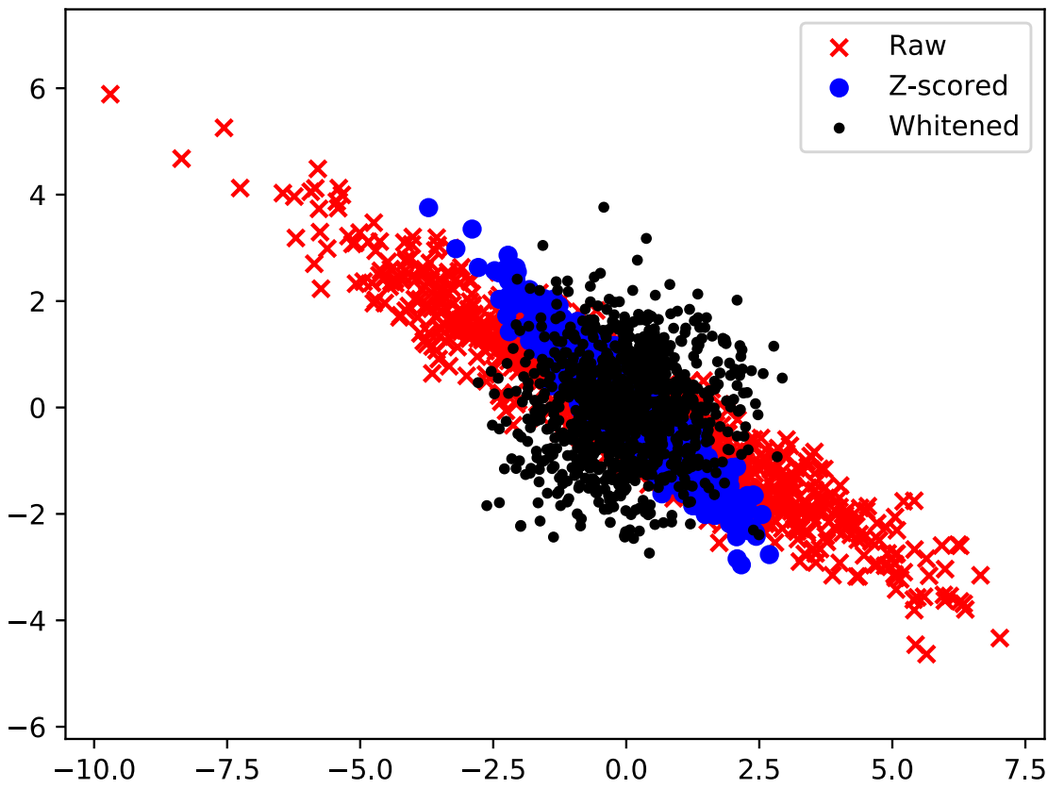
z-score approximation

- We can z-score a set $\{\mathbf{h}^{(i)}\}_{i=1}^{n}$ by computing:

$$\tilde{\mathbf{h}}_j^{(i)} = \frac{\mathbf{h}_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where ε is a numerical stability hyperparameter.

- In practice, we perform this within each mini-batch.



Covariances

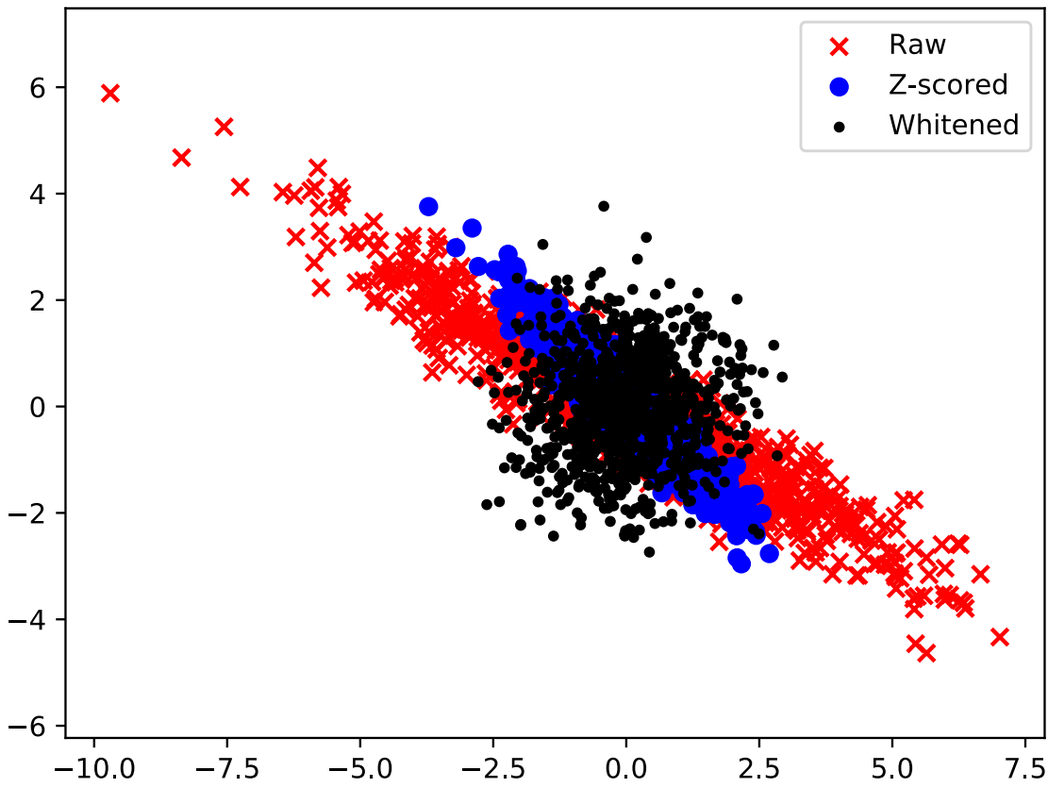
```
Raw:
[[ 6.81764404 -3.86103353]
 [-3.86103353  2.45930235]]

Whitened:
[[9.53471754e-01 1.31581988e-16]
 [1.31581988e-16 9.98898924e-01]]

Z-Scored:
[[ 1.001001   -0.94387604]
 [-0.94387604  1.001001  ]]
```

z-score approximation

- Note that it's easy to "undo" the z-scoring; just compute:

$$\mathbf{h}_j^{(i)} = \tilde{\mathbf{h}}_j^{(i)} \sqrt{\sigma_j^2 + \epsilon} \; + \mu_j$$

- Why this is useful will become apparent shortly…



Covariances

```
Raw:
[[ 6.81764404 -3.86103353]
 [-3.86103353  2.45930235]]

Whitened:
[[9.53471754e-01 1.31581988e-16]
 [1.31581988e-16 9.98898924e-01]]

Z-Scored:
[[ 1.001001   -0.94387604]
 [-0.94387604  1.001001  ]]
```

Batch normalization

- z-scoring can approximately whiten the distribution P(**h**) and is computationally efficient ($O(m)$).

- However, it may also overly restrict the representational capacity of the NN.

- We want to "selectively" z-score P(**h**) when it makes sense — as determined automatically during optimization.

- Batch normalization can achieve both.

Batch normalization

- Batch normalization is a layer that can be inserted into a NN when we want to z-score the *previous* layer's outputs.

- Given **h**, the BatchNorm layer produces **n**.

$$\tilde{\mathbf{h}}_j^{(i)} = \frac{\mathbf{h}_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$\mathbf{n}_j^{(i)} = \gamma_j \tilde{\mathbf{h}}_j^{(i)} + \beta_j$$

Batch normalization

- Batch normalization is a layer that can be inserted into a NN when we want to z-score the *previous* layer's outputs.

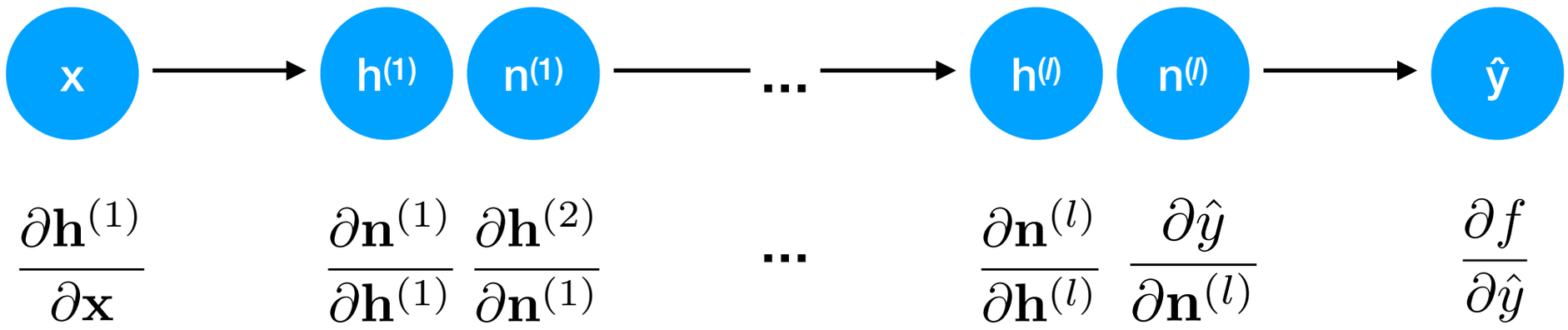
- Given **h**, the BatchNorm layer produces **n**.

$$\tilde{\mathbf{h}}_j^{(i)} = \frac{\mathbf{h}_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$\mathbf{n}_j^{(i)} = \gamma_j \tilde{\mathbf{h}}_j^{(i)} + \beta_j$$

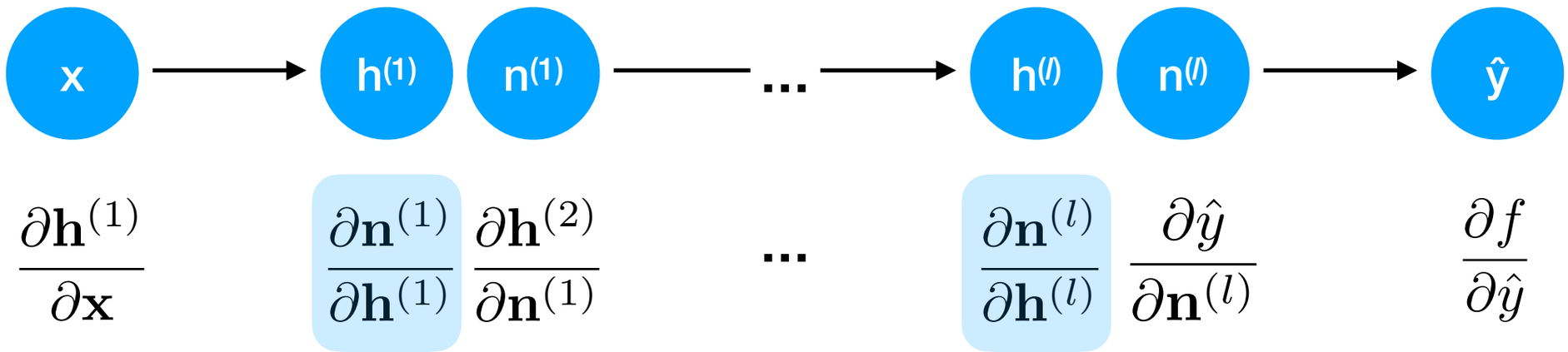
- The layer contains learned weights (*m*-vectors) *β* and *γ*:
  - If each $\gamma_j = 1, \beta_j = 0$, then the layer z-scores its inputs.
  - If $\gamma_j = \sqrt{\sigma_j^2 + \epsilon}, \beta_j = \mu_j$, then the layer recovers an identity transformation.

Batch normalization



- Crucially, subtracting the mean and dividing by the standard deviation occur as part of forward-propagation.

Batch normalization



- Crucially, subtracting the mean and dividing by the standard deviation occur as part of forward-propagation.

- This means that they are also taken into account seamlessly during back-propagation via the BatchNorm's Jacobian matrices.

Success of batch normalization

- Since the original 2015 paper, some researchers have questioned *why* batch normalization is so effective.

- There is some evidence (Santurkar et al. 2018) that batch normalization's benefit has more to do with smoothing the loss function.

Recent articles on NN optimization

- Step size: Nar and Sastry 2018

- Batch size: Keskar et al. 2017