Training Neural Nets

COMPSCI 371D — Machine Learning

Outline

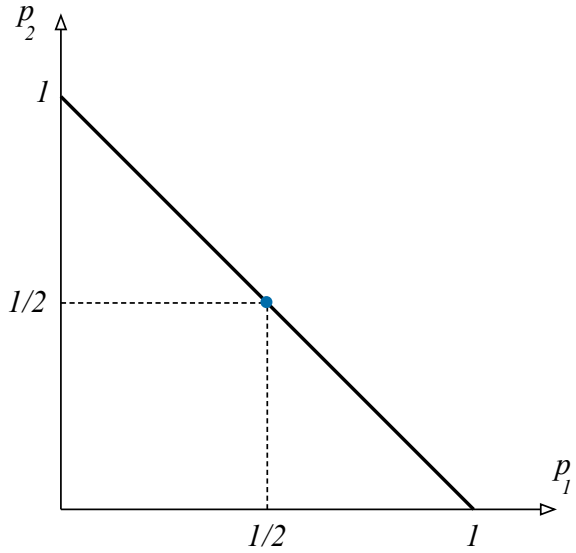
The Softmax Simplex

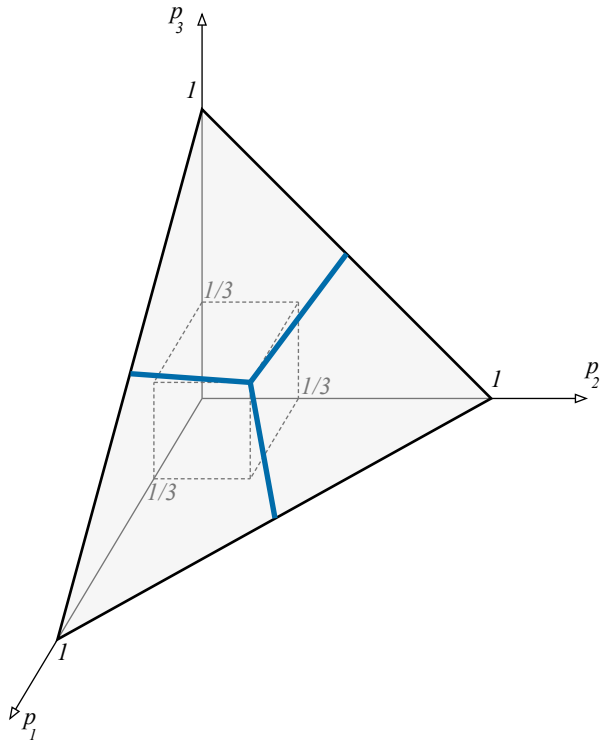
- Neural-net classifier: $\hat{y} = h(\mathbf{x}) \ : \ X \subseteq \mathbb{R}^d \to Y$
- The last layer of a neural net used for classification is a soft-max layer

  $\mathbf{p} = \sigma(\mathbf{z}) = \frac{\exp(\mathbf{z})}{\mathbf{1}^T \exp(\mathbf{z})}$
- The net is $\mathbf{p} = f(\mathbf{x}, \mathbf{w}) \ : \ X \to P$
- The classifier is $\hat{y} = h(\mathbf{x}) = \arg\max \mathbf{p} = \arg\max f(\mathbf{x}, \mathbf{w})$
- P is the set of all nonnegative real-valued vectors $\mathbf{p} \in \mathbb{R}^e$ whose entries add up to 1 (with $e = |Y|$):

$$P \stackrel{\text{def}}{=} \{\mathbf{p} \in \mathbb{R}^e \ : \ \mathbf{p} \geq \mathbf{0} \ \text{ and } \ \sum_{c=1}^{e} p_c = 1\} \ .$$

$$P \stackrel{\text{def}}{=} \{\mathbf{p} \in \mathbb{R}^e \ : \ \mathbf{p} \geq \mathbf{0} \ \text{ and } \ \textstyle\sum_{i=1}^{e} p_i = 1\}$$





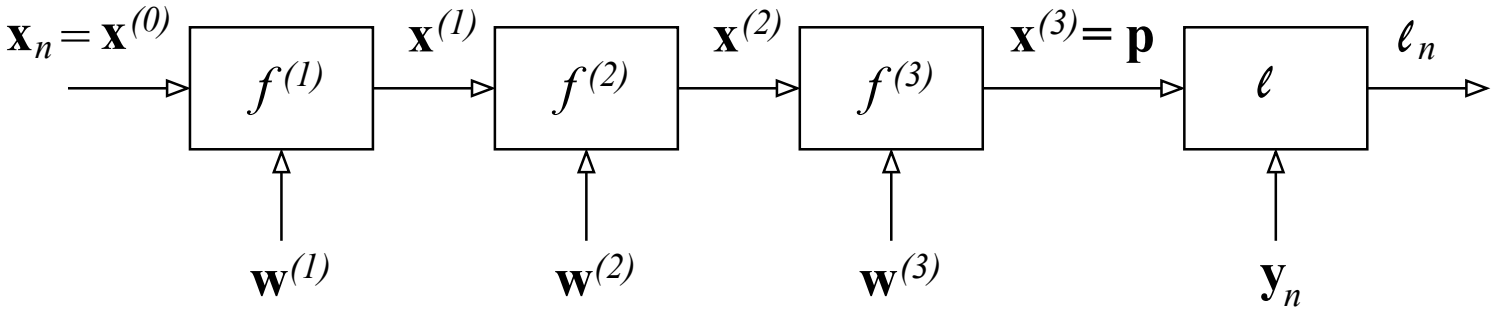
- Decision regions are polyhedral:
  $P_c = \{p_c \geq p_j \ \text{ for } \ j \neq c\} \quad \text{for} \quad c = 1, \ldots, e$
- A network transforms images into points in P

Loss and Risk (Déjà Vu)

- Ideal loss would be 0-1 loss on network output $\hat{y}$
- 0-1 loss is constant where it is differentiable!
- Not useful for computing a gradient
- Use cross-entropy loss on the softmax output $\mathbf{p}$ as a proxy loss

  $\ell(y, \mathbf{p}) = -\log p_y$
- Risk, as usual:
  $L_T(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ell_n(\mathbf{w}) \quad \text{where} \quad \ell_n(\mathbf{w}) = \ell(y_n, f(\mathbf{x}_n, \mathbf{w}))$
- We need $\nabla L_T(\mathbf{w})$ and therefore $\nabla \ell_n(\mathbf{w})$

Back-Propagation



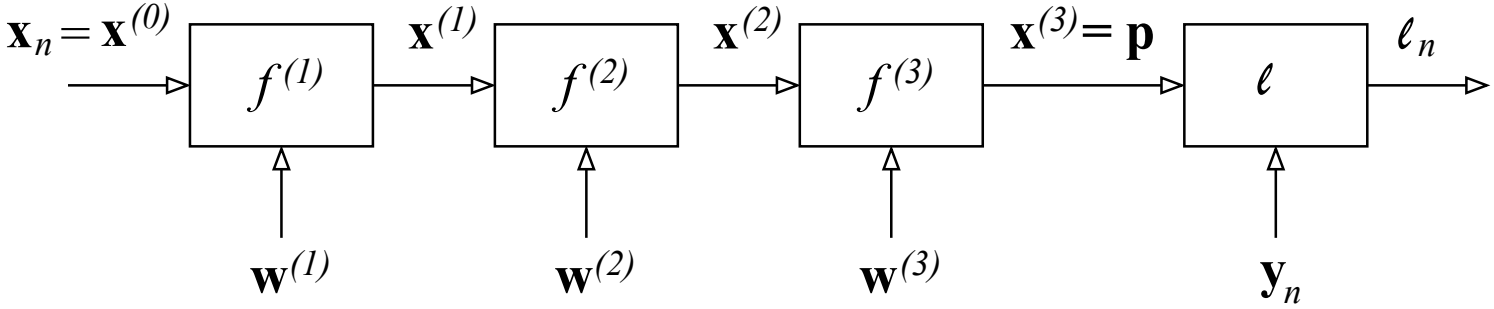
- We need $\nabla L_T(\mathbf{w})$ and therefore $\nabla \ell_n(\mathbf{w}) = \frac{\partial \ell_n}{\partial \mathbf{w}}$
- Computations from $\mathbf{x}$ to ℓ_n form a chain
- Apply the chain rule
- Every derivative of ℓ_n w.r.t. layers before k goes through $\mathbf{x}^{(k)}$

$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(k)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(k-1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}} \quad \text{(recursion!)}$$

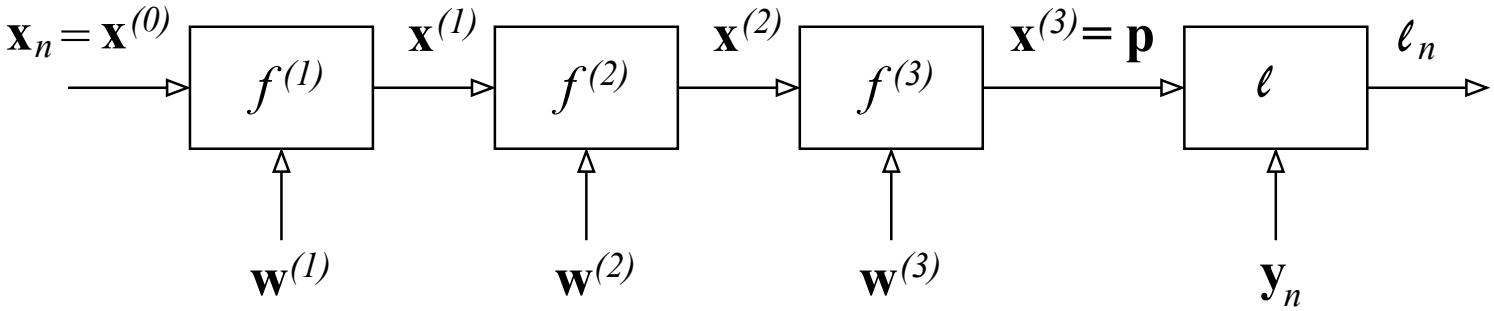
- Start: $\frac{\partial \ell_n}{\partial \mathbf{x}^{(K)}} = \frac{\partial \ell}{\partial \mathbf{p}}$

Local Jacobians



- Local computations at layer k: $\frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}}$ and $\frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}}$
- Partial derivatives of $f^{(k)}$ with respect to layer weights and input to the layer
- Local Jacobian matrices, can compute by knowing what the layer does
- The start of the process can be computed from knowing the loss function, $\frac{\partial \ell_n}{\partial \mathbf{x}^{(K)}} = \frac{\partial \ell}{\partial \mathbf{p}}$
- Another local Jacobian
- The rest is going recursively from output to input, one layer at a time, accumulating $\frac{\partial \ell_n}{\partial \mathbf{w}^{(k)}}$ into a vector $\frac{\partial \ell_n}{\partial \mathbf{w}}$

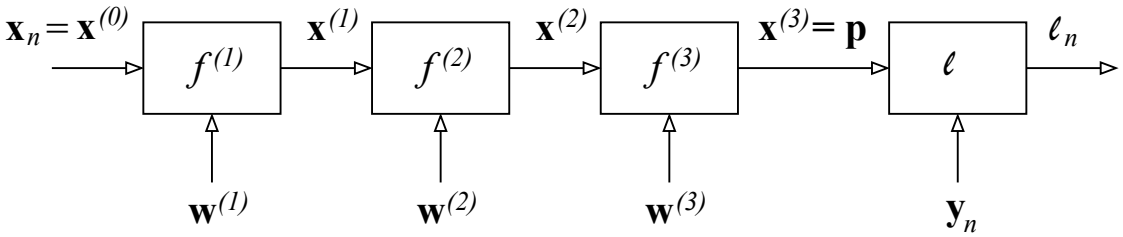
The Forward Pass



- All local Jacobians, $\frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}}$ and $\frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}}$, are computed numerically for the current values of weights $\mathbf{w}^{(k)}$ and layer inputs $\mathbf{x}^{(k-1)}$
- Therefore, we need to know $\mathbf{x}^{(k-1)}$ for training sample n and for all k
- This is achieved by a *forward pass* through the network:

  Run the network on input $\mathbf{x}_n$ and store $\mathbf{x}^{(0)} = \mathbf{x}_n$, $\mathbf{x}^{(1)}, \ldots$

Back-Propagation Spelled Out for $K = 3$



$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(k)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(k-1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}}$$

(after forward pass)

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(3)}} = \frac{\partial \ell}{\partial \mathbf{p}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(3)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(3)}} \frac{\partial \mathbf{x}^{(3)}}{\partial \mathbf{w}^{(3)}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(2)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(3)}} \frac{\partial \mathbf{x}^{(3)}}{\partial \mathbf{x}^{(2)}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(2)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(2)}} \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{w}^{(2)}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(2)}} \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(1)}}$$

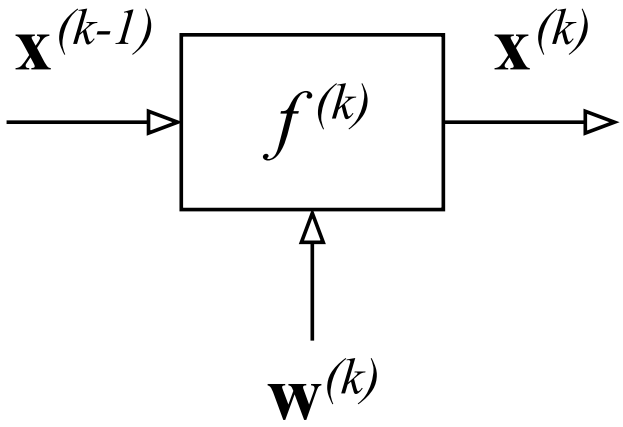
$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(1)}} \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{w}^{(1)}}$$

$$\left(\frac{\partial \ell_n}{\partial \mathbf{x}^{(0)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(1)}} \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{x}^{(0)}} \right)$$

$$\frac{\partial \ell_n}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial \ell_n}{\partial \mathbf{w}^{(1)}} \\ \\ \frac{\partial \ell_n}{\partial \mathbf{w}^{(2)}} \\ \\ \frac{\partial \ell_n}{\partial \mathbf{w}^{(3)}} \end{bmatrix}$$

(Jacobians in blue are local, those in red are what we want eventually)

Computing Local Jacobians



$$\frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}} \quad \text{and} \quad \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}}$$

- Easier to make a "layer" as simple as possible
- $\mathbf{z} = V\mathbf{x} + \mathbf{b}$ is one layer (Fully Connected (FC), affine part)
- $\mathbf{z} = \rho(\mathbf{x})$ (ReLU) is another layer
- Softmax, max-pooling, convolutional,...

Local Jacobians for a FC Layer

$$\mathbf{z} = V\mathbf{x} + \mathbf{b}$$

- $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = V$ (easy!)
- $\frac{\partial \mathbf{z}}{\partial \mathbf{w}}$: What is $\frac{\partial \mathbf{z}}{\partial V}$? Three subscripts: $\frac{\partial z_i}{\partial v_{jk}}$. A 3D tensor?
- For a general package, tensors are the way to go
- Conceptually, it may be easier to vectorize everything:

$$V = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \rightarrow$$

$$\mathbf{w} = [v_{11}, v_{12}, v_{13}, v_{21}, v_{22}, v_{23}, b_1, b_2]^T$$

- $\frac{\partial \mathbf{z}}{\partial \mathbf{w}}$ is a 2×8 matrix
- With e outputs and d inputs, an $e \times e(d+1)$ matrix

Jacobian$_{\mathbf{w}}$ for a FC Layer

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} w_7 \\ w_8 \end{bmatrix}$$

- Don't be afraid to spell things out:

$$z_1 = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_7$$

$$z_2 = w_4 x_1 + w_5 x_2 + w_6 x_3 + w_8$$

$$\frac{\partial \mathbf{z}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial z_1}{\partial w_1} & \frac{\partial z_1}{\partial w_2} & \frac{\partial z_1}{\partial w_3} & \frac{\partial z_1}{\partial w_4} & \frac{\partial z_1}{\partial w_5} & \frac{\partial z_1}{\partial w_6} & \frac{\partial z_1}{\partial w_7} & \frac{\partial z_1}{\partial w_8} \\ \frac{\partial z_2}{\partial w_1} & \frac{\partial z_2}{\partial w_2} & \frac{\partial z_2}{\partial w_3} & \frac{\partial z_2}{\partial w_4} & \frac{\partial z_2}{\partial w_5} & \frac{\partial z_2}{\partial w_6} & \frac{\partial z_2}{\partial w_7} & \frac{\partial z_2}{\partial w_8} \end{bmatrix}$$

$$\frac{\partial \mathbf{z}}{\partial \mathbf{w}} = \begin{bmatrix} x_1 & x_2 & x_3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & x_1 & x_2 & x_3 & 0 & 1 \end{bmatrix}$$

- Obvious pattern: Repeat $\mathbf{x}^T$, staggered, e times
- Then append the $e \times e$ identity at the end

Training

- Compute $\nabla \ell_n(\mathbf{w}) = \nabla \ell(y_n, h(\mathbf{x}_n; \mathbf{w}))$
- Loop over T to compute $\nabla L_T(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \nabla \ell_n(\mathbf{w})$
- $\hat{\mathbf{w}} = \arg\min L_T(\mathbf{w})$
- $L_T(\mathbf{w})$ is (very) non-convex, so we look for local minima
- $\mathbf{w} \in \mathbb{R}^m$ with m very large: No Hessians
- *Gradient descent*
- Even so, every step calls back-propagation N times
- Back-propagation computes m derivatives $\nabla \ell_n(\mathbf{w})$
- Computational complexity is $\Omega(mN)$ ***per step***
- Even gradient descent is way too expensive!

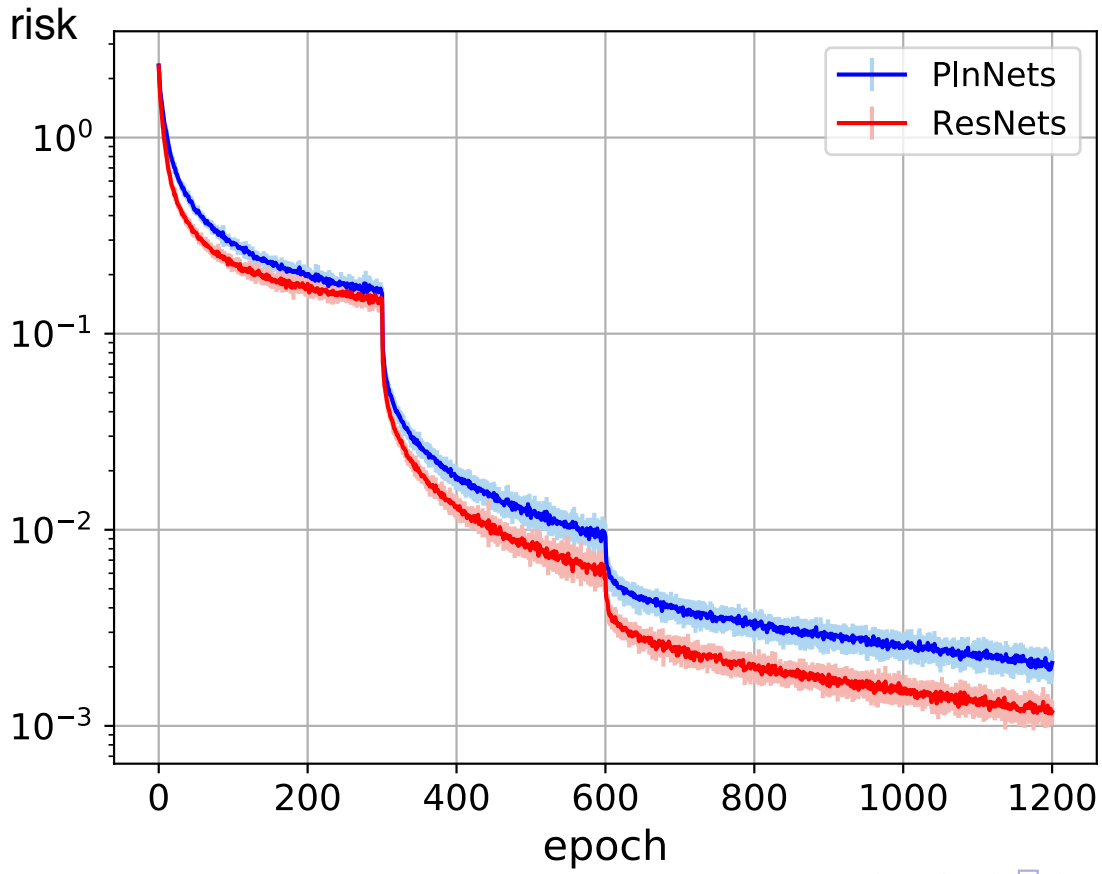
No Line Search

- Line search is out of the question
- Fix some step multiplier α, called the *learning rate*

  $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla L_T(\mathbf{w}_t)$
- How to pick α? Cross-validation is too expensive
- Tradeoffs:
  - α too small: Slow progress
  - α too big: Jump over minima
- Frequent practice:
  - Start with α relatively large, and monitor $L_T(\mathbf{w})$
  - When $L_T(\mathbf{w})$ levels off, decrease α
- Alternative: Fixed decay schedule for α
- Another (recent) option: Change α adaptively (Adam, 2015, later improvements)

Manual Adjustment of α

- Start with α relatively large, and monitor $L_T(\mathbf{w}_t)$
- When $L_T(\mathbf{w}_t)$ levels off, decrease α
- Typical plots of $L_T(\mathbf{w}_t)$ versus iteration index t:



Batch Gradient Descent (Review)

- We have seen GD and SGD under function optimization
- We review these as they are crucial for neural networks
- $\nabla L_T(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \nabla \ell_n(\mathbf{w})$
- Taking a macro-step $-\alpha \nabla L_T(\mathbf{w}_t)$ is the same as taking the N micro-steps $-\frac{\alpha}{N} \nabla \ell_1(\mathbf{w}_t), \ldots, -\frac{\alpha}{N} \nabla \ell_N(\mathbf{w}_t)$
- First compute all the N steps at $\mathbf{w}_t$, then take all the steps
- Thus, standard gradient descent is a *batch* method: Compute the gradient at $\mathbf{w}_t$ using the entire batch of data, then move
- Even with no line search, N micro-steps are expensive
- Can we spend the same amount of effort more effectively?

Stochastic Gradient Descent (Review)

- Taking a macro-step $-\alpha \nabla L_T(\mathbf{w}_t)$ is the same as taking the N micro-steps $-\frac{\alpha}{N}\nabla \ell_1(\mathbf{w}_t), \ldots, -\frac{\alpha}{N}\nabla \ell_N(\mathbf{w}_t)$
- First compute all the N steps at $\mathbf{w}_t$, then take all the steps
- Can we spend the same amount of effort more effectively?
- *Key observation:* $-\nabla \ell_n(\mathbf{w})$ is a poor estimate of $-\nabla L_T(\mathbf{w})$, *but an estimate all the same*: Micro-steps are correct on average!
- After each micro-step, we are on average in a better place
- How about *computing a new micro-gradient after every micro-step*?
- Now each micro-step gradient is evaluated at a point that is on average better (lower risk) than in the batch method

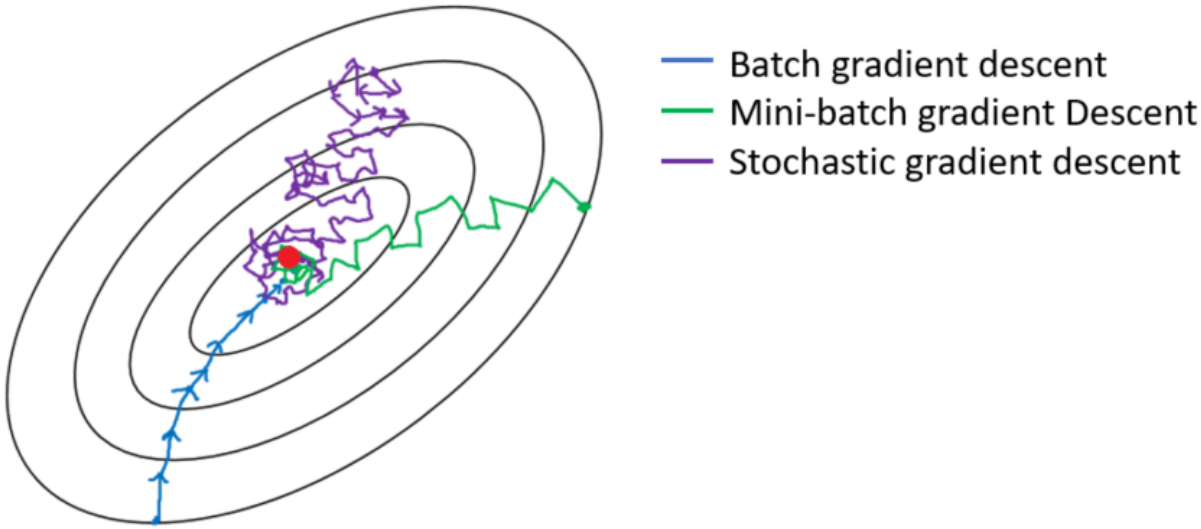
Batch vs Stochastic GD (Review)

- $\mathbf{s}_n(\mathbf{w}) = -\frac{\alpha}{N}\nabla \ell_n(\mathbf{w})$
- Batch:
  - Compute $\mathbf{s}_1(\mathbf{w}_t), \ldots, \mathbf{s}_N(\mathbf{w}_t)$
  - Move by $\mathbf{s}_1(\mathbf{w}_t)$, then $\mathbf{s}_2(\mathbf{w}_t)$, ... then $\mathbf{s}_N(\mathbf{w}_t)$ (or equivalently move once by $\mathbf{s}_1(\mathbf{w}_t) + \ldots + \mathbf{s}_N(\mathbf{w}_t)$)
- Stochastic (SGD):
  - Compute $\mathbf{s}_1(\mathbf{w}_t)$, then move by $\mathbf{s}_1(\mathbf{w}_t)$ from $\mathbf{w}_t$ to $\mathbf{w}_t^{(1)}$
  - Compute $\mathbf{s}_2(\mathbf{w}_t^{(1)})$, then move by $\mathbf{s}_2(\mathbf{w}_t^{(1)})$ from $\mathbf{w}_t^{(1)}$ to $\mathbf{w}_t^{(2)}$

    $\vdots$
  - Compute $\mathbf{s}_N(\mathbf{w}_t^{(N-1)})$, then move by $\mathbf{s}_N(\mathbf{w}_t^{(N-1)})$ from $\mathbf{w}_t^{(N-1)}$ to $\mathbf{w}_t^{(N)} = \mathbf{w}_{t+1}$
- In SGD, each micro-step is taken from a better (lower risk) place *on average* than in batch descent

Why "Stochastic?" (Review)

- Progress occurs only *on average*
- Many micro-steps are bad, but they are good on average
- Progress is a random walk



https://towardsdatascience.com/

Reducing Variance: Mini-Batches (Review)

- Each data sample is a poor estimate of T: High-variance micro-steps
- Each micro-step take full advantage of the estimate, by moving right away: Lower-bias micro-steps than batch steps
- High variance *may* hurt more than low bias helps
- Can we lower variance at the expense of slightly increased bias?
- Average B samples at a time: Take *mini-steps*
- With bigger B,
  - Higher bias
  - Lower variance
- The B samples are a *mini-batch*

Mini-Batches (Review)

- Scramble T at random (T has N samples)
- Divide T into J mini-batches T_j of size B, so $N \approx JB$
- $\mathbf{w}^{(0)} = \mathbf{w}$
- For $j = 1, \ldots, J$:
  - Batch gradient:
    $\mathbf{g}_j = \nabla L_{T_j}(\mathbf{w}^{(j-1)}) = \frac{1}{B} \sum_{n=(j-1)B+1}^{jB} \nabla \ell_n(\mathbf{w}^{(j-1)})$
  - Move: $\mathbf{w}^{(j)} = \mathbf{w}^{(j-1)} - \alpha \mathbf{g}_j$
- This `for` loop amounts to *one* macro-step
- Each execution of the entire loop uses the training data once
- Each execution of the entire loop is an *epoch*
- Repeat over several epochs until a stopping criterion is met