

Optimization

Reading

- Deep Learning: chapters 4.3, 6.5, 8
 - <https://www.deeplearningbook.org/contents/optimization.html>
- Optimization overview, with a deep learning bias

Optimization Overview

- Optimization is a very large research field (typically taught/studied in engineering departments)
- Many tasks can be formulated as an optimization problem
 - Allocating different people to different jobs to maximize productivity
 - Choosing the best control action for your autonomous car
 - Finding the best parameters for your neural network
- Standard form

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & g(x) \leq C \end{array}$$

- Optimization is either minimization or maximization

Optimization in ML

- The optimization problem in ML is indirect
 - Want to perform well according to metric P (e.g., classification accuracy) but optimize some loss L (e.g., least squares)
 - Want to maximize performance on true data distribution but can only maximize performance on sampled data

Empirical Risk Minimization

- Expected value of loss function is called **risk** in ML

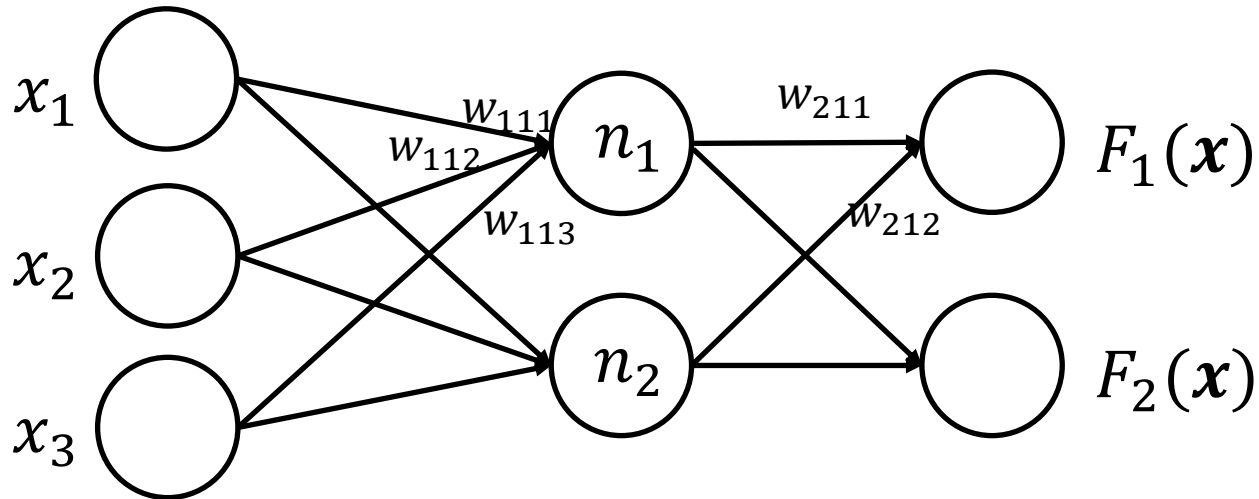
$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{X}, Y) \sim \mathbb{P}_{population}} L(f(\mathbf{X}; \boldsymbol{\theta}), Y)$$

- **Empirical risk** is the average of the loss function over dataset

$$\mathbb{E}_{(\mathbf{X}, Y) \sim \mathbb{P}_{data}} L(f(\mathbf{X}; \boldsymbol{\theta}), Y) = \frac{1}{N} \sum_{i=1}^N L(f(\mathbf{x}_i; \boldsymbol{\theta}), y_i)$$

- ML is all about empirical risk minimization
- 2 challenges
 - Formulating the right minimization problem
 - Pick the architecture, loss, regularization, etc.
 - Solving the minimization problem
 - Find global optimum, scale well with more data, complex models

Setup



- A 3-input, 2-output network
 - The inputs are $\mathbf{x} = [x_1 \ x_2 \ x_3]$
 - The parameters are $\boldsymbol{\theta} = [w_{111}, w_{112}, w_{113}, w_{121}, w_{122}, w_{123}, w_{211}, w_{212}, w_{221}, w_{222}]$
 - No offsets in this example

Some losses are better than others

- In classification, one is tempted to choose weights that minimize a 0-1 loss (1 for incorrect classification, 0 for correct)
 - However, picking the weights that minimize 0-1 loss is a hard computational task
- Other losses often more efficient
 - E.g., NLL is a smooth function of the data, which makes it easier to minimize
- Cannot compute solution in closed form for any loss, e.g.,

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N (y_i - f(\mathbf{x}_i; \theta))^2$$

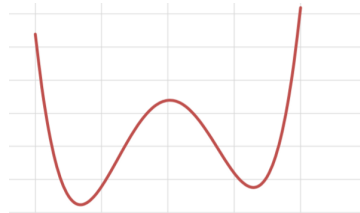
- Also, NN makes the loss functions non-convex
- Why would convexity be a nice property?

Gradient Descent Idea

- Section 4.3 in the book
 - Gradient is the word for derivate in higher dimensions
- Some functions are easier to minimize than others
 - E.g., convex functions are minimized when derivative is 0

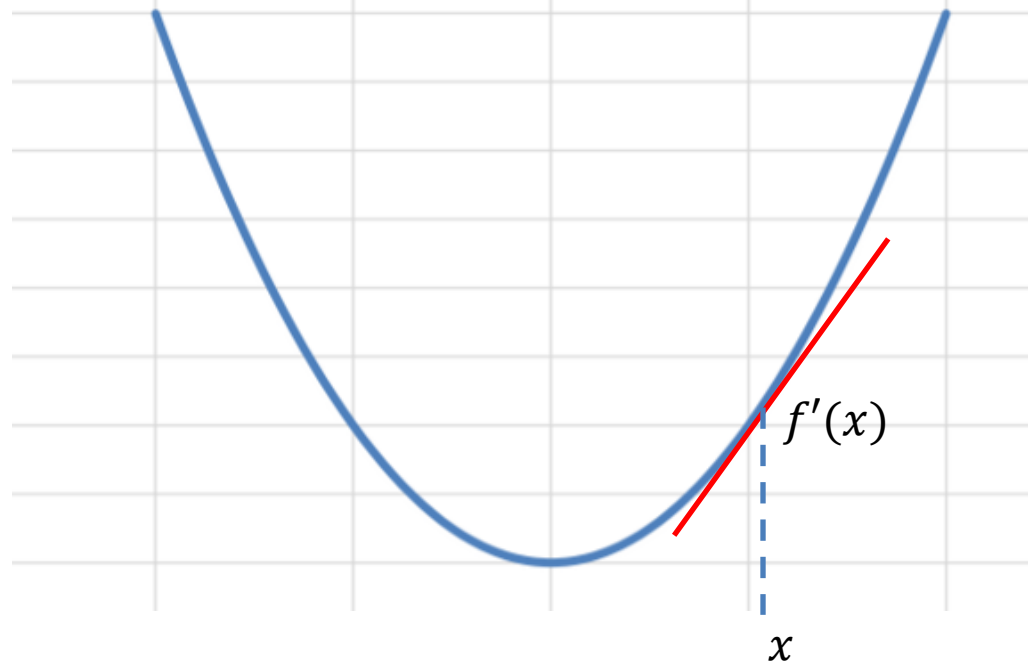


- Hard to find root of derivative in most cases
- Also, most functions are not convex (including neural nets)



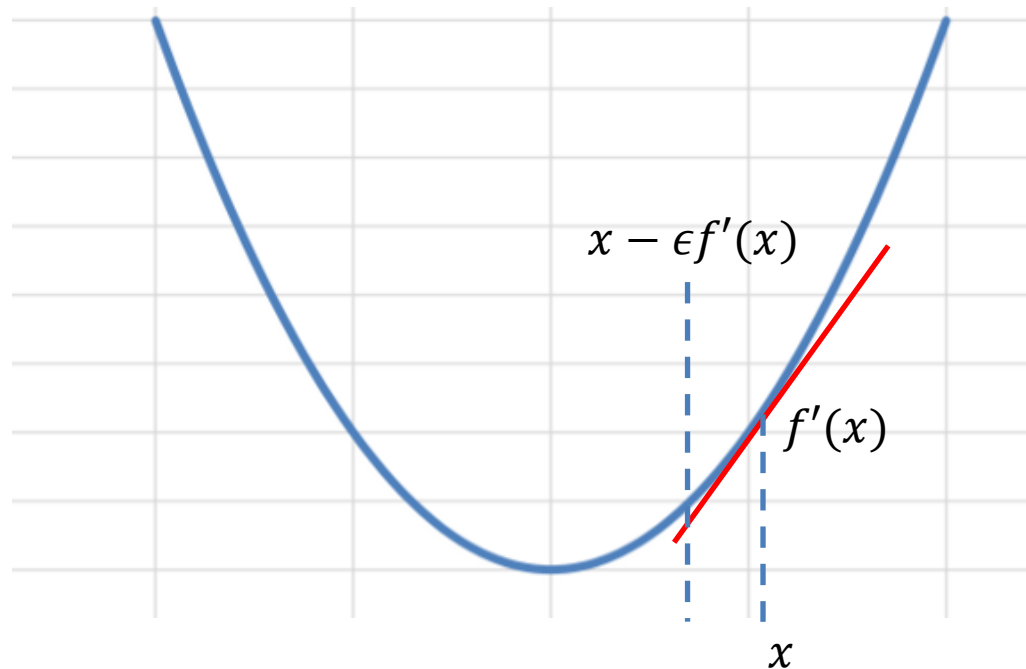
Gradient Descent, cont'd

- If you can't find the root of the derivative, you can try to iteratively minimize the function
 - Start from some x , compute $f'(x)$ and make a step in the opposite direction
 - We know that $f(x - \epsilon f'(x)) < f(x)$ for small ϵ
 - ϵ is learning rate



Gradient Descent, cont'd

- If you can't find the root of the derivative, you can try to iteratively minimize the function
 - Start from some x , compute $f'(x)$ and make a step in the opposite direction
 - We know that $f(x - \epsilon f'(x)) < f(x)$ for small ϵ
 - ϵ is learning rate



Aside: Vector Calculus

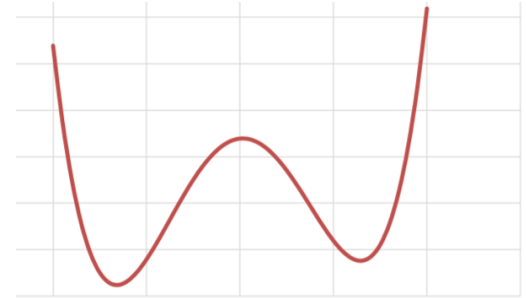
- Suppose we are given a function $f: \mathbb{R}^n \rightarrow \mathbb{R}$
- Suppose we want to know the derivative of f
- When $n = 1$, it is just the partial derivative $f' = \frac{\partial f}{\partial x}$
- When $n > 1$, the derivative is a vector of all partial derivatives:

$$\nabla_x f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

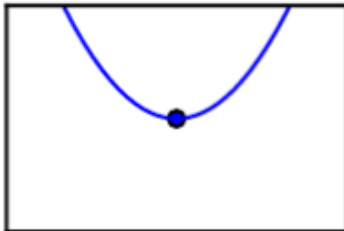
- This is called the “gradient” of f
- The gradient is the multi-dimensional extension of the derivative

Gradient Descent, cont'd

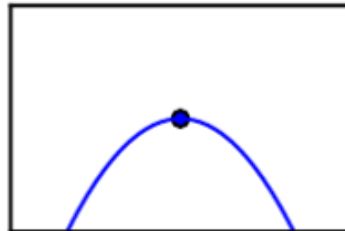
- What about non-convex functions?
 - Can easily get stuck in a local min
- What about saddle points?
 - Derivative can be very small
 - Major concern in high-dimensional spaces



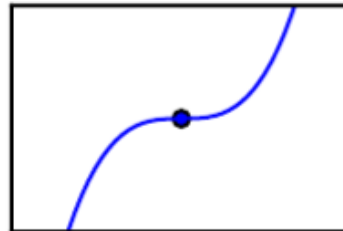
Minimum



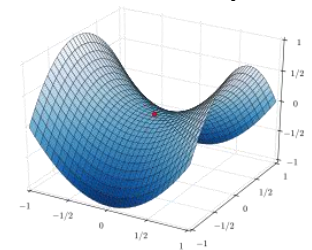
Maximum



Saddle point



Source: wikipedia



- Despite all these limitations, neural network training usually finds a good local minimum
 - Beware: larger networks can easily minimize the loss and overfit (more on this next)

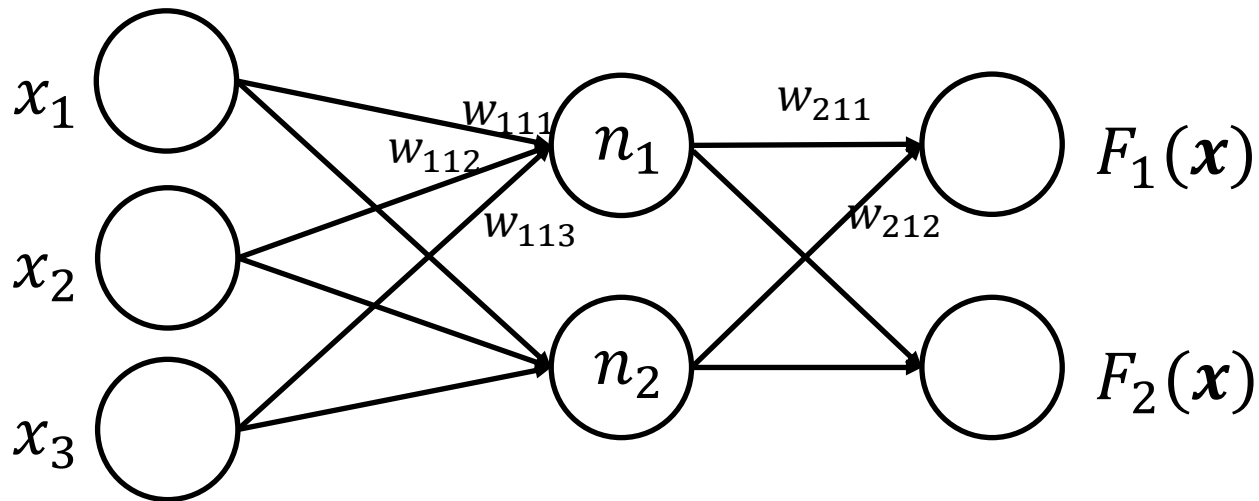
Back Propagation

- An algorithm for computing gradients quickly
 - This is what makes deep learning so efficient
 - No need to worry about it too much – implemented in deep learning libraries
 - But good to understand it when choosing an architecture/loss combination
- Computing NN derivatives involves multiple repeated expressions
 - Backprop is an efficient way of reusing previously computed values

Computing NN derivatives is a massive chain rule

- Most derivatives have interesting properties
 - $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
 - $\tanh'(x) = 1 - \tanh^2(x)$
 - $ReLU'(x) = Step(x) := \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$
- Most derivatives can be expressed in terms of the original function
 - Also appear multiple times

Example



- Suppose we have a two-neuron neural network with 3 inputs and 2 outputs
 - ReLU activation in hidden layer and linear last layer
- Suppose loss is least squares (assume $y_i \in \{0,1\}$)

$$\frac{1}{N} \sum_{i=1}^N (y_i - F_1(\mathbf{x}_i))^2 + ((1 - y_i) - F_2(\mathbf{x}_i))^2$$

Example, cont'd

- To compute the gradient, need to compute partial derivative w.r.t. each weight
- Start with w_{111}
- The partial derivative of the first term in the sum is

$$\begin{aligned}\frac{\partial (y_i - F_1(\mathbf{x}_i))^2}{\partial w_{111}} &= -2(y_i - F_1(\mathbf{x}_i)) \frac{\partial F_1(\mathbf{x}_i)}{\partial w_{111}} \\ \frac{\partial F_1(\mathbf{x}_i)}{\partial w_{111}} &= \frac{\partial (w_{211}n_1(\mathbf{x}_i) + w_{212}n_2(\mathbf{x}_i))}{\partial w_{111}} = w_{211} \frac{\partial n_1(\mathbf{x}_i)}{\partial w_{111}} \\ \frac{\partial n_1(\mathbf{x}_i)}{\partial w_{111}} &= \frac{\partial \text{ReLU}(w_{111}x_{i1} + w_{112}x_{i2} + w_{113}x_{i3})}{\partial w_{111}} \\ &= x_{i1} \text{Step}(w_{111}x_{i1} + w_{112}x_{i2} + w_{113}x_{i3})\end{aligned}$$

Example, cont'd

- Thus, the partial derivative of the 1st term w.r.t. w_{111} is
$$-2(y_i - F_1(\mathbf{x}_i))w_{211}x_{i1}\text{Step}(w_{111}x_{i1} + w_{112}x_{i2} + w_{113}x_{i3})$$
- The partial derivative of the 1st term w.r.t. w_{112} is
$$-2(y_i - F_1(\mathbf{x}_i))w_{211}x_{i2}\text{Step}(w_{111}x_{i1} + w_{112}x_{i2} + w_{113}x_{i3})$$
- ...
- Thus, the partial derivative of the 2nd term w.r.t. w_{111} is
$$-2((1 - y_i) - F_2(\mathbf{x}_i))w_{221}x_{i1}\text{Step}(w_{111}x_{i1} + w_{112}x_{i2} + w_{113}x_{i3})$$
- ...
- Need to do this for all weights and for all datapoints
 - Many repeated terms, especially for big NNs

General setup: one-hot encoding

- To make writing losses easier, the training labels are often stored as one-hot encodings
- Suppose we have a label y_i
 - The one-hot encoding is $\mathbf{y}_i = [0 \ 0 \ \dots \ 1 \ 0 \ \dots \ 0]$
 - With a 1 in position y_i
- Thus, $\mathbf{y}_i$ has the same dimension as the NN output layer
- Can now write least squares as:

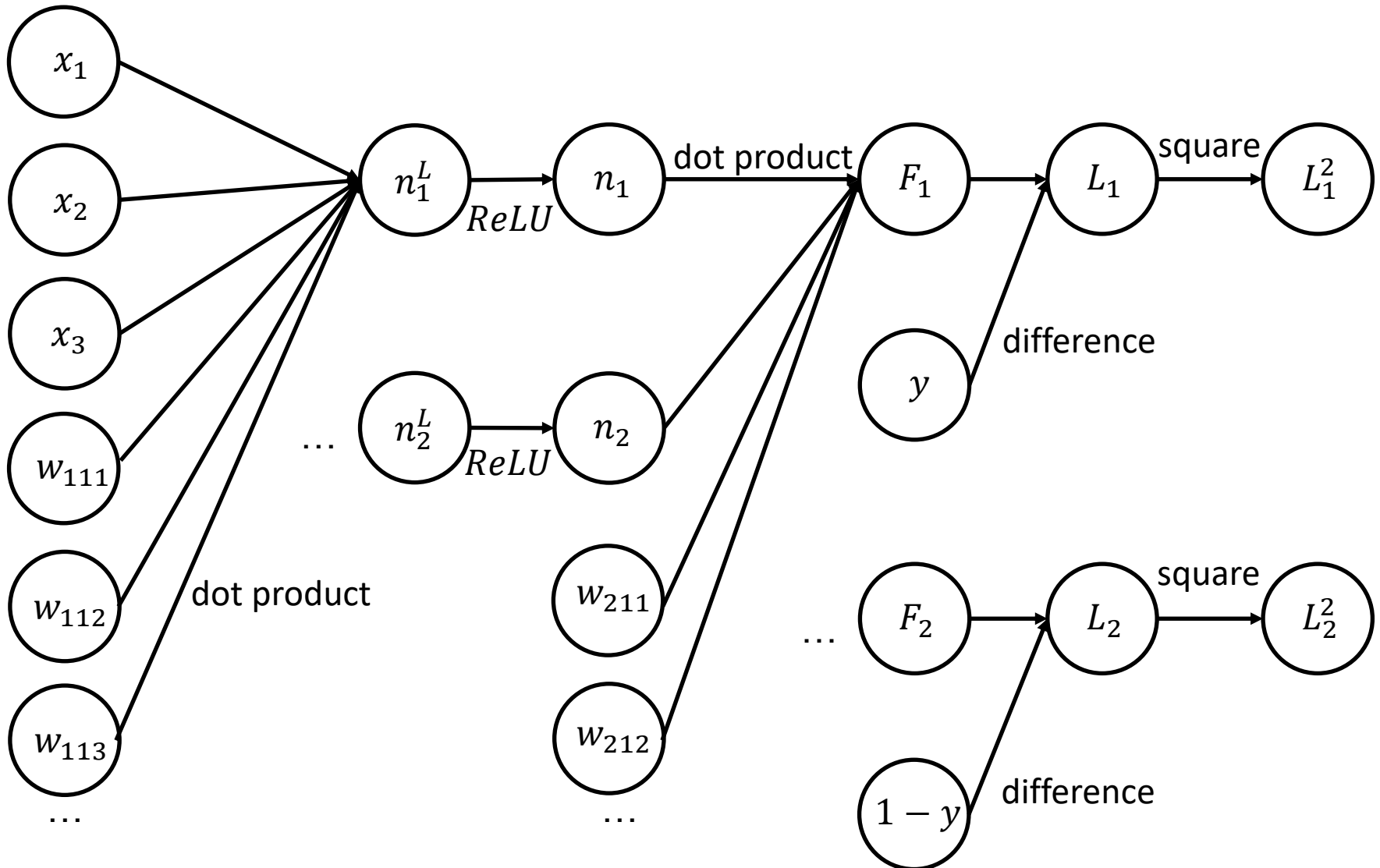
$$\sum_{i=1}^N ||\mathbf{F}(x_i) - \mathbf{y}_i||_2^2 = \sum_{i=1}^N (\mathbf{F}(x_i) - \mathbf{y}_i)^T (\mathbf{F}(x_i) - \mathbf{y}_i)$$

- Other losses can be written similarly

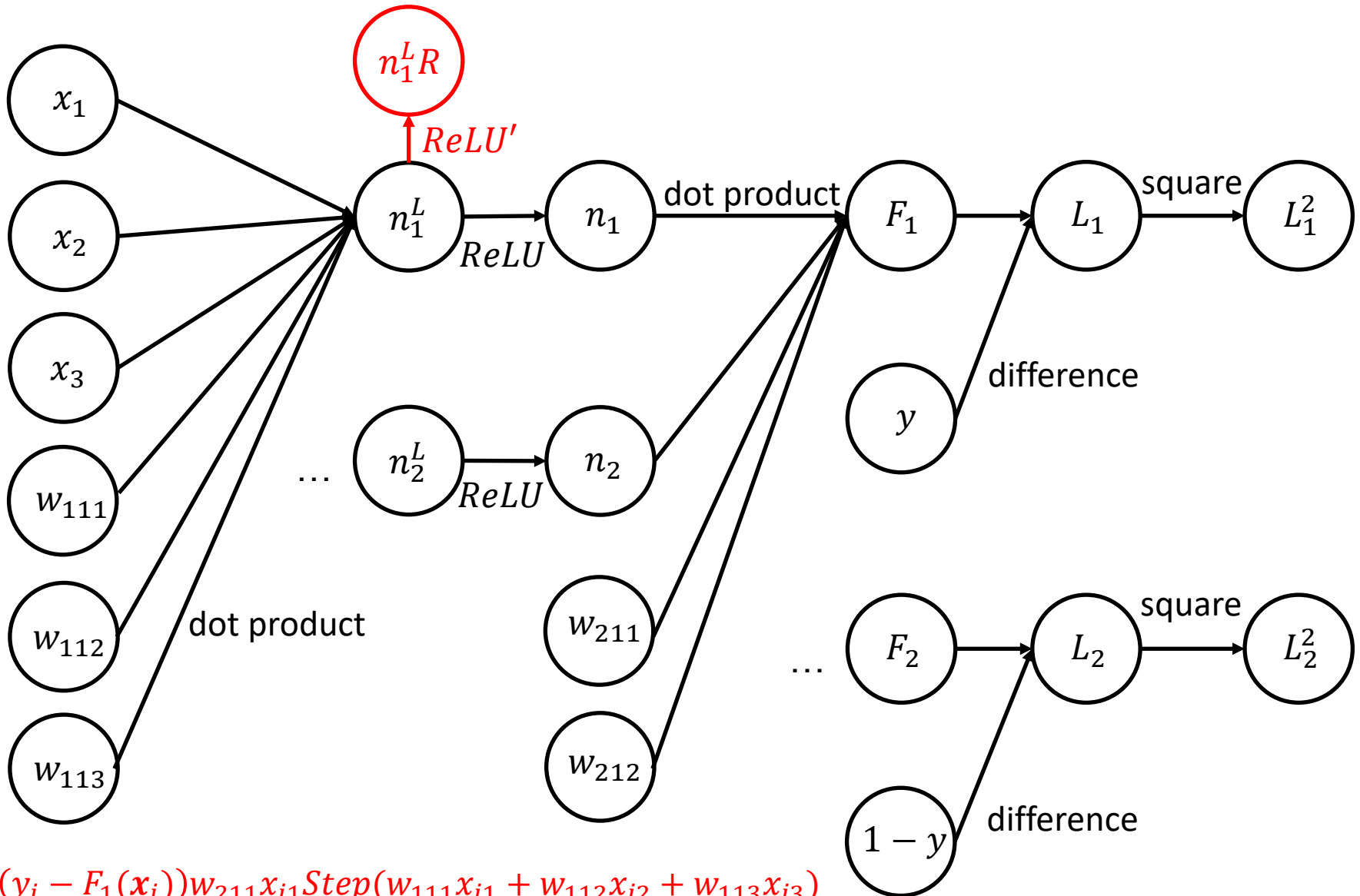
Computational Graphs

- Store all operations in a graph to be reused later
 - Nodes represent intermediate variables
 - Edges represent operations on variables
- Most derivatives appear multiple times
 - Graph representation can save a lot of time
 - Same idea as dynamic programming
- Gradient computation really involves two computations
 - Forward propagation: compute the actual value of the loss
 - Backward propagation: compute the gradient using the chain rule

Example, forward propagation



Example, backward propagation (start)



Implementation

- Many optimizations to make gradient computation fast
 - Linear operations performed on GPUs (gamers know why)
 - Variables stored as tensors (high-dimensional matrices)
- Several popular deep learning libraries
 - Mostly in python
 - Tensorflow – a bit clunky, but fairly flexible
 - Pytorch – a bit less flexible, but very easy to use
- You don't need to worry about most of the low-level details in this lecture when implementing NNs
 - However, you need to have a good working knowledge of the low-levels if you want your code to work

Minibatch Algorithms

- A major reason for the success of deep learning
- Computing the gradient over all examples each time is too expensive
- What if use just a few examples per gradient computation?
- Randomly sample a few examples each time
 - Sample called a minibatch
 - Compute gradient on minibatch
 - Algorithm called **stochastic gradient descent (SGD)**

SGD Properties

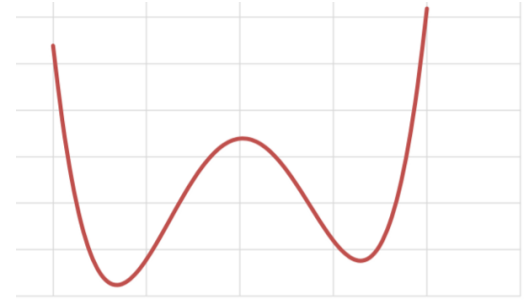
- Standard error of the gradient is $\frac{\sigma}{\sqrt{n}}$
 - σ is the true standard deviation for one example
 - n is the number of examples in the minibatch
 - Standard error decreases slowly: $O(1/\sqrt{n})$
 - Larger minibatches don't bring significant benefits
- Using minibatches also useful when data has low natural variance (why?)
 - Not usually true, but many examples may be similar
- Entire minibatch can be processed in parallel on GPU
 - The bottleneck is fitting all data in memory
- Overall, computation speedup offsets noise due to using a minibatch

Epochs

- Ideally, each minibatch is selected randomly every time
 - Nearby examples may often be correlated
 - Impractical for big datasets
- Instead, we shuffle the dataset before training and then process minibatches in order
 - Each pass of the full dataset is called an **epoch**
 - Other hyper-parameters may also change in between epochs
 - E.g., learning rate, regularization, etc.

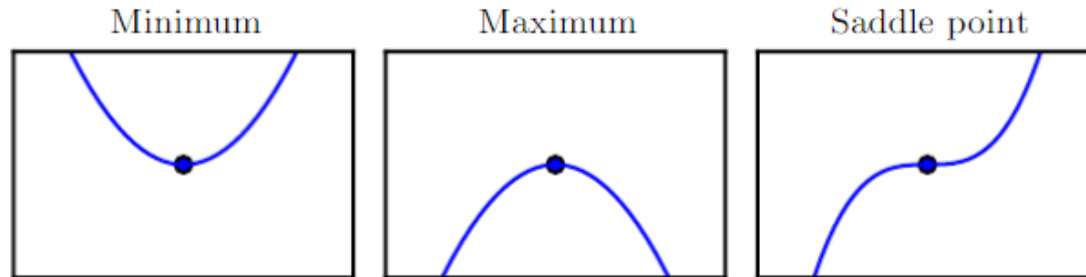
Optimization Challenges

- Local Minima
- Gradient is ~ 0 , so no progress can be made
- Local minima are very common
 - One of the most impressive achievements of NNs is that they are able to generalize well despite using suboptimal weights
 - A possible explanation for this phenomenon is that all local minima have similar values
 - A lot of research in this area



Optimization Challenges, cont'd

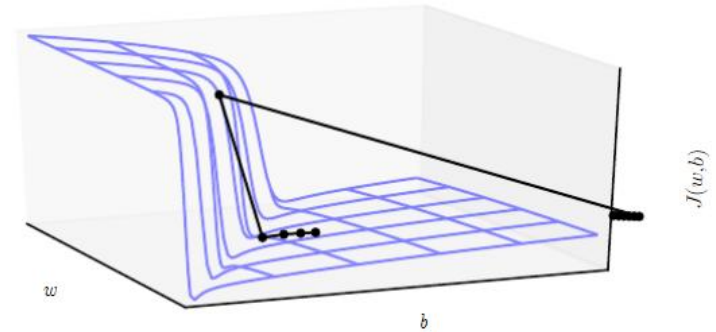
- Plateaus, saddles



- More problematic than local minima
 - Gradient is also ~ 0 , but loss is not low
 - Very common in high-dimensional spaces and in NN optimization
 - However, gradient descent is usually able to escape

Optimization Challenges, cont'd

- Exploding gradients
- Gradients can get very large when reaching a cliff in loss function
 - Can destabilize training (parameters jumping around)
 - Can also cause numerical issues
- Disaster can be avoided by using gradient clipping
 - If gradient norm above some threshold, reduce learning rate



SGD Summary

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \dots$

Require: Initial parameter θ

$k \leftarrow 1$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Apply update: $\theta \leftarrow \theta - \epsilon_k \hat{\mathbf{g}}$

$k \leftarrow k + 1$

end while

- Learning rate is usually gradually decreased to some final value
 - Linear, exponential rates of decay both work
 - Typically, you can keep it constant for small enough tasks
 - What are the trade-offs between small/large learning rate?

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ , momentum parameter α

Require: Initial parameter θ , initial velocity v

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

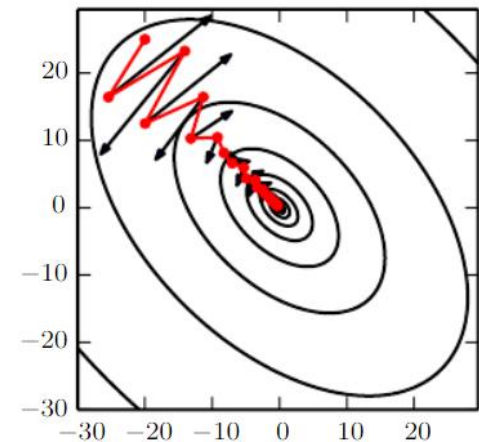
 Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

 Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$.

 Apply update: $\theta \leftarrow \theta + \mathbf{v}$.

end while

- Descent direction is smoothed out over time in order to filter out noise due to minibatch variance
 - Essentially a low-pass filter (in signal processing terms)
 - Allows you to increase the learning rate somewhat



Parameter Initialization

- Parameter initialization may have a serious impact on training
- Unlikely to be a major issue but it could slow down training
 - If you try hard, you could also find unstable initializations
- Initialization strategies are heuristics
 - Not fully clear why they work and when
 - Make sure to avoid the same weights across units (why?)
 - Gradients will be the same; weights will always remain the same

- Standard choices are initial uniform distributions

$$U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right), U\left(-\frac{6}{\sqrt{n+m}}, \frac{6}{\sqrt{n+m}}\right)$$

- here, n is number of neurons in layer, m is number of inputs
- biases are initialized similarly

Batch Normalization

- Another important factor for the success of deep learning
- It is common practice to normalize all training data to be 0-mean and bounded between $[-0.5, 0.5]$

$$\frac{X - \mu}{\sigma}$$

- Can do the same for inputs to all hidden layers also
- Gradient descent can be brittle for deep networks
 - Updates all layers simultaneously, using a local linear approximation
 - However, the output of the NN is a non-linear (composite) function of the weights
 - Complex non-linear relationships may make it hard to choose the right learning rate

Batch Norm, cont'd

- For a given minibatch, let $\mathbf{H}_l$ be the output of layer l
- We can normalize it as follows

$$\mathbf{H}'_l = \frac{\mathbf{H}_l - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

- where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the (element-wise) mean and variance of $\mathbf{H}_l$ over the minibatch
- Crucially, we backpropagate through this operation in order to stabilize the gradients across layers
- At test time, we can use a running average of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ accumulated during training

Batch Norm Summary

- In practice, we introduce learned parameters γ and β such that the output of the batch norm layer is
$$\gamma H' + \beta$$
- Seems a bit counter-intuitive since we are adding back a mean and a variance
 - The hope is that gradient descent finds suitable parameters that make training more stable
- Batch normalization most useful for deep convolutional NNs