Lecture 6

Multiclass Classification

Input(feature vector): $x \in \mathbb{R}^d$

Output(label): $y \in [C] = \{1, 2, \cdots, C\}$

Goal: learning a mapping $f : \mathbb{R}^d \to [C]$

Examples:

- recognizing digits ($C = 10$) or letters ($C = 26$ or 52)
- predicting weather: sunny, cloudy, rainy, etc
- predicting image category: ImageNet dataset ($C \approx 20K$)

Linear models: Binary to multiclass

a linear model for binary tasks (switching from $\{-1, +1\}$ to $\{1, 2\}$)

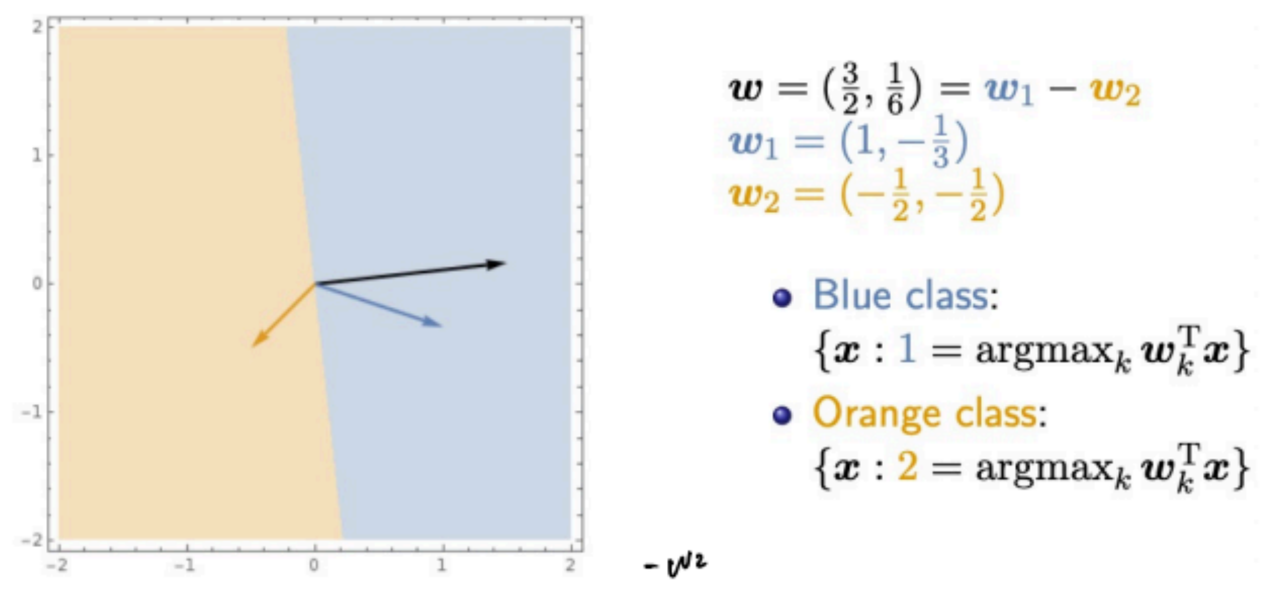
$$f(x) = \begin{cases} 1 & if\; w^T x \geq 0 \\ 2 & if\; w^T x < 0 \end{cases}$$

Can be written as

$$\begin{aligned} f(x) &= \begin{cases} 1 & if\; w_1^T x \geq w_2^T x \\ 2 & if\; w_1^T x < w_2^T x \end{cases} \\ &= \arg\max_{k \in \{1,2\}} w_k^T x \end{aligned}$$

For any w_1, w_2 s.t. $w = w_1 - w_2$.

Think of $w_k^T x$ as a score for class k .

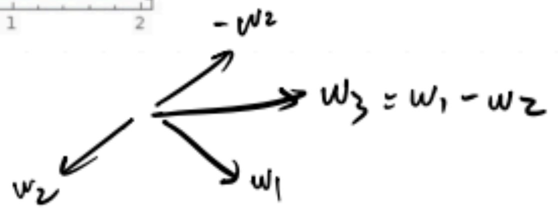


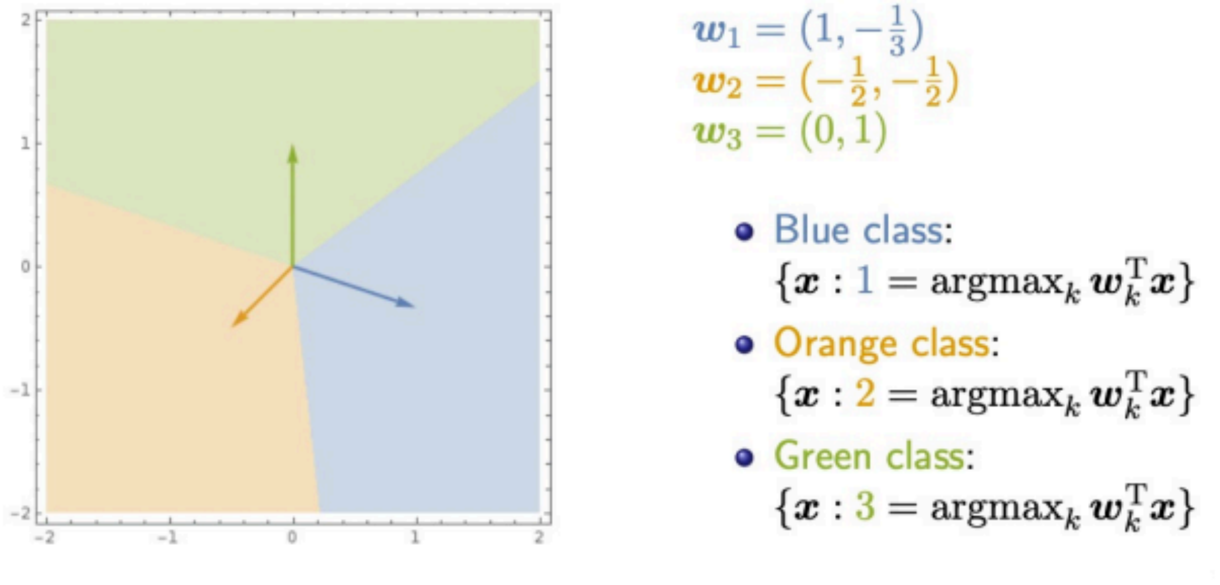
$$\boldsymbol{w} = \left(\tfrac{3}{2}, \tfrac{1}{6}\right) = \boldsymbol{w}_1 - \boldsymbol{w}_2$$

$$\boldsymbol{w}_1 = \left(1, -\tfrac{1}{3}\right)$$

$$\boldsymbol{w}_2 = \left(-\tfrac{1}{2}, -\tfrac{1}{2}\right)$$

- Blue class: $\{\boldsymbol{x} : 1 = \text{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$
- Orange class: $\{\boldsymbol{x} : 2 = \text{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$





Function loss: Linear models for multi class classification

$$\mathcal{F} = \left\{ f(x) = \arg\max_{k\in[C]} w_k^T x |\; w_1, \cdots, w_C \in \mathbb{R}^d \right\}$$
$$= \left\{ f(x) = \arg\max_{k\in[C]} (Wx)_k |\; W \in \mathbb{R}^{C\times d} \right\}$$

Lets try to generalize the loss functions. Focus on the logistic loss today.

A probabilistic view of multinomial logistic regression:

Observe: for binary logistic regression, with $w = w_1 - w_2$

$$\Pr(y=1|\; x; w) = \sigma(w^T x) = \frac{1}{1+e^{-w^T x}} = \frac{e^{w_1^T x}}{e^{w_1^T x} + e^{w_2^T x}} \propto e^{w_1^T x}$$
$$\Pr(y=2|\; x; w) \propto e^{w_2^T x}$$

So for multi class:

$$\Pr(y=k|\; x; W) = \frac{e^{w_k^T x}}{\sum_{k\in[C]} e^{w_k^T x}} \propto e^{w_k^T x}$$

This is called the softmax function.

Coverts scores $w_k^T x \rightarrow Pr(y=k|\; x; w) = \frac{e^{w_k^T x}}{\sum_{k\in[C]} e^{w_k^T x}}$.

MLE of Softmax

Use the MLE, Maximize probability of seeing labels $y_1, \cdots, y_n$ given $x_1, \cdots, x_n$:

$$P(W) = \prod_{i=1}^{n} Pr(y_i|\ x_i; W) = \prod_{i=1}^{n} \frac{e^{w_k^T x}}{\sum_{k\in[C]} e^{w_k^T x}}$$

By using $-\ln$, this is equivalent to minimizing:

$$F(W) = \sum_{i=1}^{n} \ln \frac{\sum_{k\in[C]} e^{w_k^T x}}{e^{w_k^T x}} = \sum_{i=1}^{n} \ln\Big(1 + \sum_{k\neq y_i} e^{(w_k - w_{y_i})^T x_i}\Big)$$

This is the multiclass logistic loss. It is an upper-bound on the 0-1 misclassification loss

$$\mathbb{I}[f(x) \neq y] \leqslant \log_2(1 + \sum_{k\neq y} e^{(w_k - w_y)^T x}) \quad (if\ x \geqslant 0, \log_2(1 + e^x \geqslant 0))$$

When $C = 2$, multiclass logistic loss is the same as binary logistic loss.

$F(W) = \sum_{i=1}^{n} \ln\Big(1 + \sum_{k\neq y_i} e^{(w_k - w_{y_i})^T x_i}\Big)$, Consider any $I \in [n]$,

For $y_i = 1$, $\ln(1 + e^{-(w_2 - w_1)^T x_i})$

For $y_i = 2$, $\ln(1 + e^{-(w_1 - w_2)^T x_i})$

For $w = w_1 - w_2$, and transferring labels from $\{1, 2\} \rightarrow \{1, -1\}$.

$$F(w) = \sum_{i=1}^{n} \ln(1 + e^{-y_i w^T x_i})$$

Optimization

Apply SGD, what is the gradient of

$$F(W) = \sum_{i=1}^{n} \ln\Big(1 + \sum_{k\neq y_i} e^{(w_k - w_{y_i})^T x_i}\Big)$$

It's a $C \times d$ matrix. Let's focus on the $k - th$ row.

If $k \neq y_i$

$$\nabla_{w_k^T} F(W) = \frac{e^{(w_k - w_{y_i})^T x_i}}{1 + \sum_{k\neq y_i} e^{(w_k - w_{y_i}) x_i}} x_i^T = \frac{e^{w_k^T x_i}}{e^{w_{y_i}^k x_i} + \sum_{k\neq y_i} e^{w_k^T x_i}} x_i^T = \Pr(y = k|x_i; W) x_i^T$$

Think about w_k^T is a row vector, thus the derivative itself $\nabla_{w_k^T} F(W)$ should also be a row vector.
$\frac{e^{(w_k - w_{y_i})^T x_i}}{1 + \sum_{k\neq y_i} e^{(w_k - w_{y_i}) x_i}}$ is a scalar, so x_i^T should be a row vector as well.

Else if $k = y_i$

$$\nabla_{w_k^T} F(W) = \frac{-(\sum_{k\neq y_i} e^{(w_k - w_{y_i})^T x_i})}{1+\sum_{k\neq y_i} e^{(w_k-w_{y_i})x_i}} x_i^T = \frac{-(\sum_{k\neq y_i} e^{w_k^T x_i})}{e^{w_{y_i}^k x_i} + \sum_{k\neq y_i} e^{w_k^T x_i}} x_i^T = \big(\Pr(y=k|x_i;W)-1\big) x_i^T$$

Algorithm:

Initialize $W = 0$ (Or randomly). Repeat:

1. Pick $i \in [C]$ uniformly at random
2. Update the parameters

$$W \leftarrow W - \eta \begin{pmatrix} \Pr(y=1|x_i;W) \\ \vdots \\ \Pr(y=y_i|x_i;W)-1 \\ \vdots \\ \Pr(y=C|x_i;W) \end{pmatrix} x_i^T$$

Think about why it makes sense intuitively.

Probabilities -> Prediction

Having learned W , we can either

make a deterministic prediction $\arg\max_{k\in[C]} w_k^T x$

make a randomized prediction according to $\Pr(y=k|x;W) \propto e^{w_k^T x}$

Beyond linear models

Suppose we have any model f (not necessary linear) which gives some score $f_k(x)$ for the datapoint x having the k -th label.

For linear model, $f_k(x) = w_k^T x$

How can we convert this score to probabilities? Use softmax function!

$$\tilde{f}_k(x) = \Pr(y=k|x;f) = \frac{e^{f_k(x)}}{\sum_{k'\in[C]} e^{f'_k(x)}} \propto e^{f_k(x)}$$

Once we have probability estimates, what is suitable loss function to train the model?

Use the log loss. Also known as the cross-entropy loss.

Log Loss / Cross-entropy loss: Binary case

$$\begin{aligned} LogLoss &= \mathbf{1}(y=1)\ln(\frac{1}{\tilde{f}(x)}) + \mathbf{1}(y=-1)\ln(\frac{1}{1-\tilde{f}(x)}) \\ &= -\mathbf{1}(y=1)\ln(\tilde{f}(x)) - \mathbf{1}(y=-1)\ln(1-\tilde{f}(x)) \end{aligned}$$

Why? If $y=1$, want to max $\ln$, this is equivalent to min $-\ln(\tilde{f}(x))$

When the model is linear, this reduces to the logistic regression loss we defined before!

Linear model: $\sigma(w^T x) = \frac{1}{1+e^{-w^T x}}$, $1 - \sigma(w^T x) = \sigma(-w^T x)$

$$\begin{aligned} LogLoss &= -\mathbf{1}(y=1)\ln((1+e^{-w^T x})^{-1}) - \mathbf{1}(y=-1)\ln((1+e^{w^T x})^{-1}) \\ &= \ln(1+e^{-yw^T x}) \quad (logistic\ regression\ loss) \end{aligned}$$

Log Loss / Cross-entropy loss: Multiclass case

This generalizes easily to the multiclass case. For datapoint (x, y) , if $\tilde{f}_k(x)$ is the predicted probability of label k ,

$$\begin{aligned} LogLoss &= \sum_{k=1}^{C} \mathbf{1}(y=k)\ln(\frac{1}{\tilde{f}_k(x)}) \\ &= -\sum_{k=1}^{C} \mathbf{1}(y=k)\ln(\tilde{f}_k(x)) \end{aligned}$$

When the model is linear, this reduces to the logistic regression loss we defined earlier!

By combining the softmax and the log-loss, we have a general loss $l(f(x), y)$ which we can use to train a multi-class classification model which assigns scores $f_k(x)$ to the k-th class. (These scores $f_k(x)$ are sometimes referred to as logits).

$$\begin{aligned} l(f(x), y) &= -\sum_{k=1}^{C} \mathbf{1}(y=k)\ln(\tilde{f}_k(x)) \\ &= \ln\left(\frac{\sum_{k\in C} e^{f_k(x)}}{e^{f_y(x)}}\right) \\ &= \ln\left(1 + \sum_{k\neq y} e^{f_k(x)-f_y(x)}\right) \end{aligned}$$

Multiclass logistic loss: Another view

Recall that we cab predict using $\arg\max_k f_k(x)$

$$\begin{aligned} l(f(x), y) &= \ln\Big(\sum_k \exp(f_k(x))\Big) - \ln\Big(\exp(f_y(x))\Big) \\ &= \ln\Big(\sum_k \exp(f_k(x))\Big) - f_y(x) \end{aligned}$$

$$\ln\Big(\sum_k \exp(f_k(x))\Big) \approx \max_k f_k(x)$$

Verify: $\max_{k\in[C]} f_k(x) \leqslant \ln(\sum_{k\in C}\exp(f_k(x))) \leqslant \max_k f_k(x) + \ln C$

$$\therefore\ l(f(x), y) \approx \max_k f_k(x) - f_y(x)$$

Note that $\max_k f_k(x) \geqslant f_y(x)$.

One-versus-all

Other techniques for multiclass classification: Cross-entropy is the most popular, but there are other black-box techniques to convert multiclass classification to binary classification: one-versus-all (one-versus-rest, one-against-all, etc.); one-versus-one (all-versus-all, etc.); Error-Correcting Output Codes (ECOC); tree-based reduction.

Idea: train C binary classifiers to learn "is class k or not" for each k .

Training: for each class $k \in [C]$,

- Relabel examples with class k as $+1$, and all others as -1
- Train a binary classifier k_k using this new dataset.

Prediction: for a new example x

- Ask each h_k
- Randomly pick among all $k's$ s.t. $h_k(x) = +1$

Issue: will make a mistake as long as one of h_k errs.

One-versus-one

Idea: train $\binom{C}{2}$ binary classifiers to learn "is class k or k'".

Training: for each pair (k, k')

- Relabel examples with class k as $+1$ and examples with class k' as -1
- Discard all other examples
- Train binary classifier $h_{(k,k')}$ using this new dataset

Prediction: for a new example x

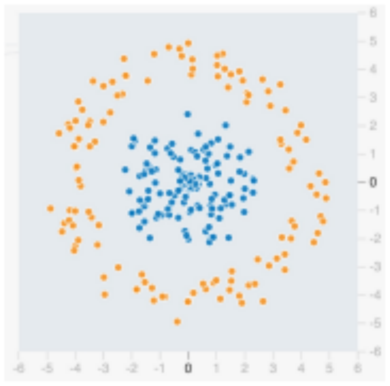
- Ask each classifier $h_{(k,k')}$ to vote for either class k or k'
- Predict the class with the most votes (break tie in some way)

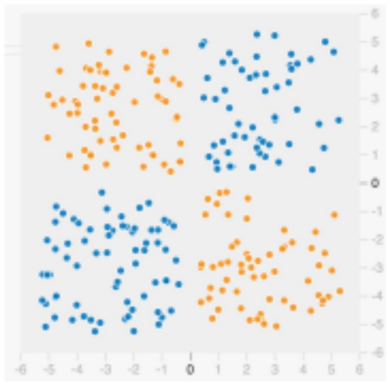
More robust than one-versus-all, but slower in prediction.

Neural Networks

Linear -> Fixed non-linear -> Learned non-linear map

Take polynomial kernel (add features x_1^2, x_2^2)



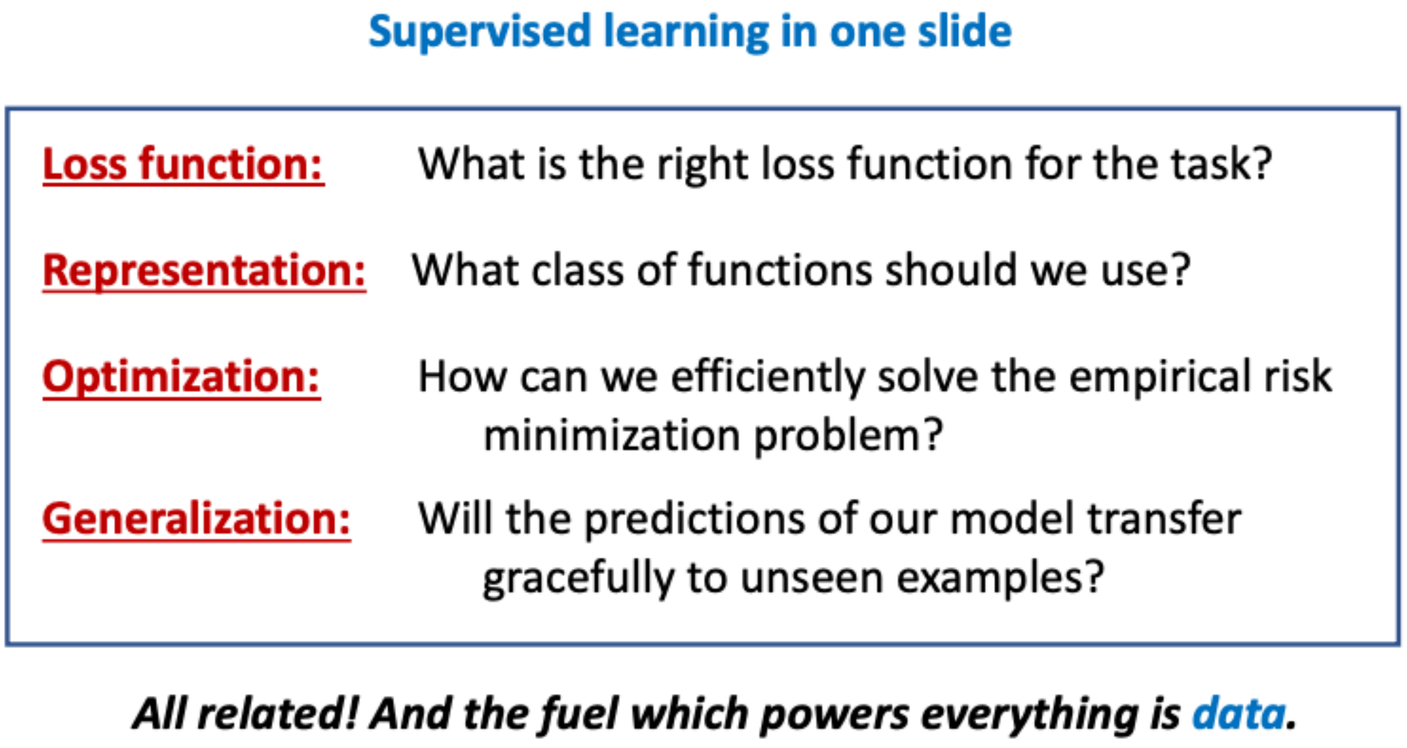


Linear models aren't always enough. As we discussed, we can use a nonlinear mapping and learn a linear model in the feature space:

$$\phi(x) : x \in \mathbb{R} \rightarrow z \in \mathbb{R}^M$$

But what kind of nonlinear mapping ϕ should be used?

Can we just learn the nonlinear mapping itself?



Supervised learning in one slide

Loss function: What is the right loss function for the task?

Representation: What class of functions should we use?

Optimization: How can we efficiently solve the empirical risk minimization problem?

Generalization: Will the predictions of our model transfer gracefully to unseen examples?

All related! And the fuel which powers everything is data.

Loss Function

For model which makes predictions $f(x)$ on labelled datapoint (x, y) , we can use the following losses.

Regression:

$$l(f(x), y) = (f(x) - y)^2$$

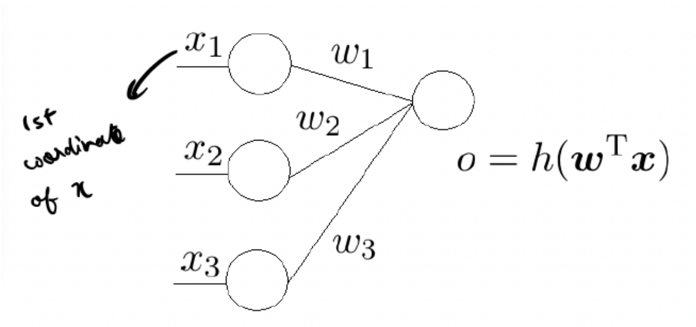
Classification:

$$l(f(x), y) = \ln\Big(\frac{\sum_{k\in[C]} e^{f_k(x)}}{e^{f_y(x)}}\Big) = \ln\Big(1 + \sum_{k\neq y} e^{f_k(x) - f_y(x)}\Big)$$

There're maybe other, more suitable options for the problem at hand, but these are the most popular for supervised problems.

Representation

Linear model as a one-layer neural network:



$$f(x) = w^T x = \sum_j x_j w_j$$

For a linear model, $h(a) = a$

To create non-linearity, can use some nonlinear (differentiable) function

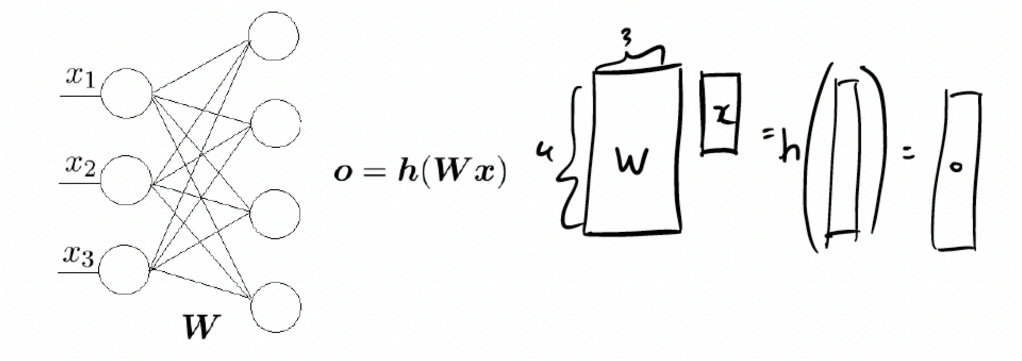
Rectified Linear Unit (ReLU): $h(a) = \max\{0, a\}$

Sigmoid function: $h(a) = \frac{1}{1+e^{-x}}$

Tanh: $h(a) = \frac{e^a - e^{-a}}{e^a + a^{-a}}$

...

Adding a layer:

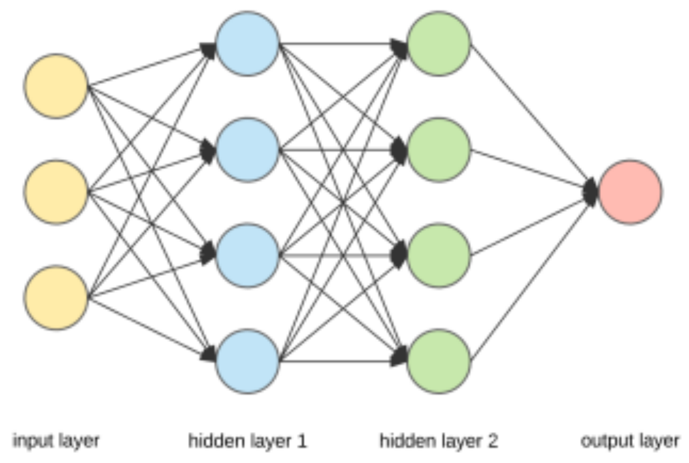

x_1
x_2
x_3
$\boldsymbol{W}$
$\boldsymbol{o} = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$
3
4
W
x
=h
=
o

$W \in \mathbb{R}^{4\times 3}, h : \mathbb{R}^4 \to \mathbb{R}^4$, so $h(a)_{apply\ function\ to\ each\ layer} = (h_1(a_1), h_2(a_2), h_3(a_3), h_4(a_4))$

Can think of this as a nonlinear mapping: $\phi(x) = h(Wx)$

Putting things together: a neural network

- Each node is called a neuron.
- h is called the activation function
  - can use $h(a) = 1$ for one neuron in each layer to incorporate bias term
  - Output neuron ca use $h(a) = a$
- Layers refers to hidden layers (plus 1 or 2 for input / output layers)
- Deep neural nets can have many layers and millions of parameters.
- This is a feedforward, fully connected neural net, there are many variants (convolutional nets, residual nets, recurrent nets, etc.)



An L-layer neural net can be written as

$$f(x) = h_L(W_L h_{L-1}(W_{L-1} \cdots h_1(W_1 x)))$$

- $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weights between layer $l-1$ and l
- $d_0 = d, d_1, \cdots, d_L$ are numbers of neurons at each layer
- $a_l \in R^{d_l}$ is input to layer l
- $o_l \in R^{d_l}$ is output of layer l

- $h_l : R^{d_l} \rightarrow R^{d_l}$ is activation functions at layer l

Now, for a given input x , we have recursive relations:

$$o_0 = x, a_l = W_l o_{l-1}, o_l = h_l(a_l), \quad (l = 1, \cdots, L)$$

Optimization

The optimization problem is to minimize

$$F(W_1, \cdots, W_L) = \frac{1}{n} \sum_{i=1}^{n} F_i(W_1, \cdots, W_L)$$

where

$$F_i(W_1, \cdots, W_L) = \begin{cases} ||f(x_i) - y_i||_2^2 & for\ regression \\ \ln\left(1 + \sum_{k \neq y} e^{f_k(x) - f_y(x)}\right) & for\ classification \end{cases}$$

Use SGD to solve this, we use backpropogation to compute the gradient efficiently. More on this soon.

Generalization

Overfitting is a concern for such a complex model, but there are ways to handle it.

For example, we can add l_2 regularization.

l_2 regularization: minimize

$$G(W_1, \cdots, W_L) = F(W_1, \cdots, W_L) + \lambda \sum_{all\ weights\ wDemo} w^2$$

Demo: http://playground.tensorflow.org/

Neural Networks: Diving Deeper

Representation: Very powerful function class

Universal approximation theorem (Cybenko, 89; Hornik, 91):

A feedforward neural net with a single hidden layer can approximate any continuous function.

It might need a huge number of neurons though, and depth helps!

Choosing the network architecture is important.

- for feedforward network, need to decide number of hidden layers, number of neurons at each layer, activation functions, etc.

Designing the architecture can be complicated, though various standard choices exist.

Optimization: Computing gradients efficiently using Backpropogation

To run SGD, need gradients of with respect to all the weights in all the layers. How do we get the gradient?

Here's a naive way to compute gradients. For some function $F(w)$ of a univariate parameter w ,

$$\frac{\mathrm{d}F(w)}{\mathrm{d}w} = \lim_{\epsilon \to 0} \frac{F(w+\epsilon) - F(w-\epsilon)}{2\epsilon}$$

If our network has m Weights, this scales as $O(m)$. (This is still useful for "gradient checking")

Backpropogation: A very efficient way to compute gradients of neural networks using an application of the chain rule (similar to dynamic programming)

Chain rule:

- For a composite function $f(g(w))$:

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial w}$$

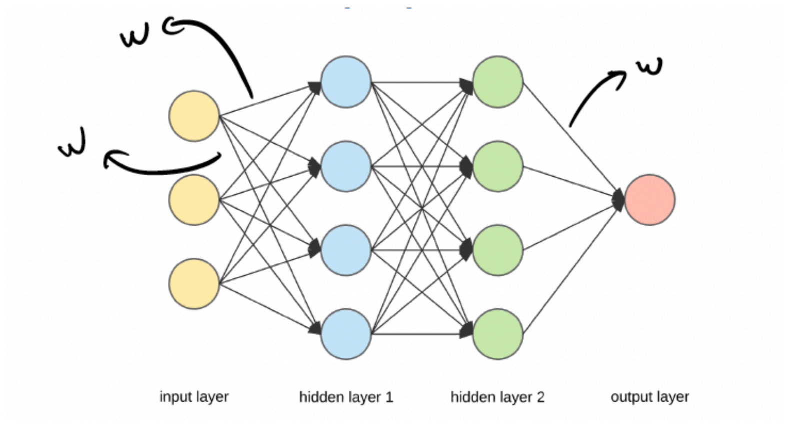
- For a composite function $f(g_1(w), \cdots, g_d(w))$

$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial w}$$

The simplest example $f(g_1(w), g_2(w)) = g_1(w)g_2(w)$

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g_1} \frac{\partial g_1}{\partial w} + \frac{\partial f}{\partial g_2} \frac{\partial g_2}{\partial w} = g_2(w)g_1'(w) + g_1(w)g_2'(w)$$

Backprop: Intuition



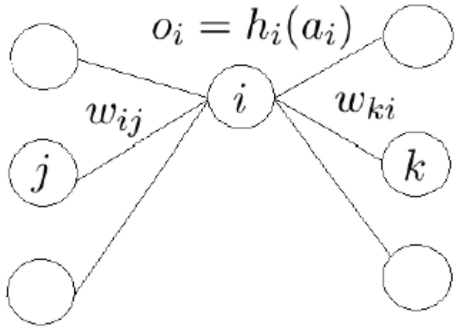
Naive: apply chain rule for each weight

Backprop: reuse computation by starting gradients want input to each layers (a_l)

Backprop: Derivation

Drop the subscript l for layer for simplicity. For this derivation, refer to the loss function as F_m (Instead of F_i) for convenience.

Find the derivative of F_m w.r.t. to $w_{i,j}$



a_i : input of neuron i

$a_k = w_{ki} o_i$

$$\frac{\partial F_m}{\partial w_{ij}} = \frac{\partial F_m}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial F_m}{\partial a_i}\frac{\partial (w_{ij} o_j)}{\partial w_{ij}} = \frac{\partial F_m}{\partial a_i} o_j$$
$$\frac{\partial F_m}{\partial a_i} = \frac{\partial F_m}{\partial o_i}\frac{\partial o_i}{\partial a_i} = \Big(\frac{\partial F_m}{\partial a_k}\frac{\partial a_k}{\partial o_i}\Big) h_i'(a_i) = \Big(\frac{\partial F_m}{\partial a_k} w_{ki}\Big) h_i'(a_i)$$

$\frac{\partial F_m}{\partial a_i}$ is the key quantity to store.

Adding the subscript for layer l:

$$\frac{\partial F_m}{\partial w_{l,ij}} = \frac{\partial F_m}{\partial a_{l,ij}} = \frac{\partial F_m}{\partial a_{l,i}} o_{l-1,j}$$
$$\frac{\partial F_m}{\partial a_{l,i}} = \Big(\sum_k \frac{\partial F_m}{\partial a_{l+1,k}} w_{l+1,k_i}\Big) h_{l,i}'(a_l, i)$$

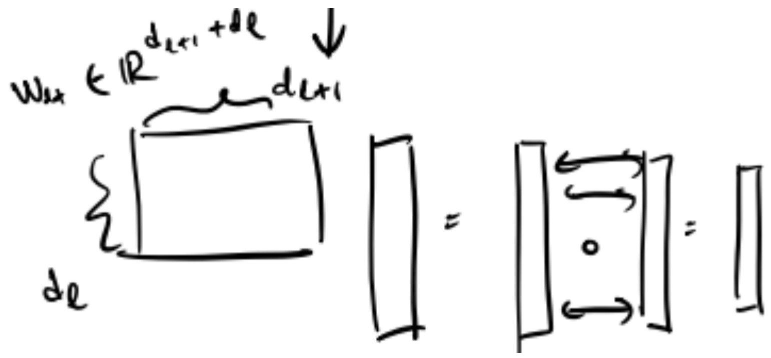
For the last layer L , for square loss

$$\frac{\partial F_m}{\partial a_{L,i}} = \frac{\partial (h_{L,i}(a_{L,i}) - y_{n,i})^2}{\partial a_{L,i}} = 2(h_{L,i}(a_{L,i}) - y_{n,i}) h_{L,i}'(a_{L,i})$$

Using matrix notation greatly simplifies presentation and implementation:

$$\frac{\partial F_m}{\partial W_l} = \frac{\partial F_m}{\partial a_l} o_{l-1}^T \in R^{d_l \times d_{l-1}}$$
$$\frac{\partial F_m}{\partial a_l} = \begin{cases} (W_{l+1}^T \frac{\partial F_m}{\partial a_{l+1}}) \circ h_l'(a_l) & if\, l < L \\ 2(h_L(a_L) - y_n) \circ h_L'(a_L) & if\, l = L \end{cases}$$

where $v_1 \circ v + (v_{11} v_{21}, \cdots, v_{1d} v_{2d}$ is the element-wise product (a.k.a. Hadamard product).



The backpropagation algorithm:

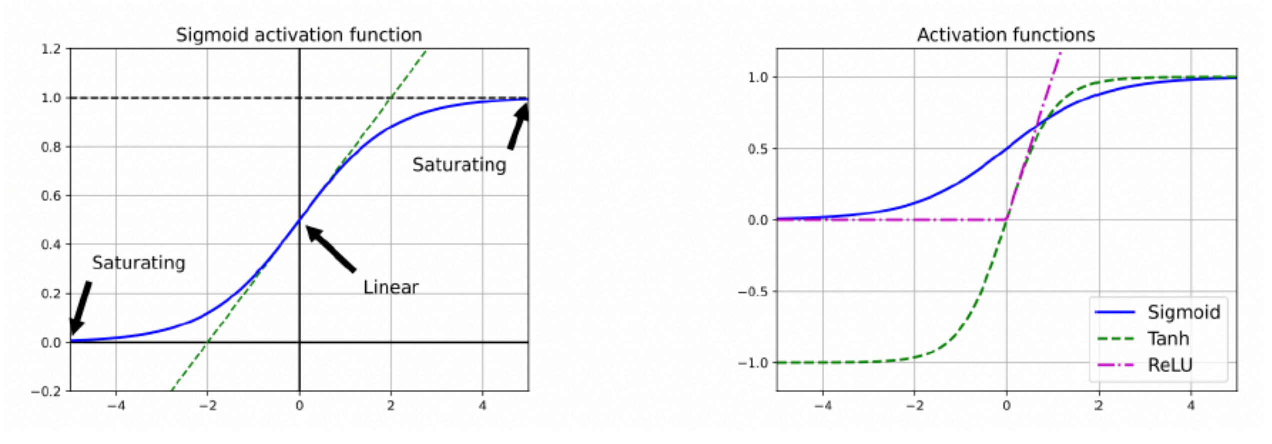
1. randomly pick one data point $I \in [n]$
2. Forward propagation: for each $l = L, \cdots, 1$
    - Compute $a_l = W_l o_{l-1}$ and $o_l = h_l(a_l) \quad (o_0 = x_i)$
3. Backward propagation: for each $l = L, \cdots, 1$
    - compute

$$\frac{\partial F_i}{\partial a_l} = \begin{cases} (W_{l+1}^T \frac{\partial F_m}{\partial a_{l+1}}) \circ h_l'(a_l) & if\, l < L \\ 2(h_L(a_L) - y_n) \circ h_L'(a_L) & else \end{cases}$$

    - Update weights

$$W_l \leftarrow W_l - \eta \frac{\partial F_i}{\partial W_l} = W_l - \eta \frac{\partial F_i}{\partial a_l} o_{l-1}^T$$

Non-saturating activation funcitons



Gradients depend on $h_l'(d_l)$. If activation function saturates $\longrightarrow$ gradient is too small

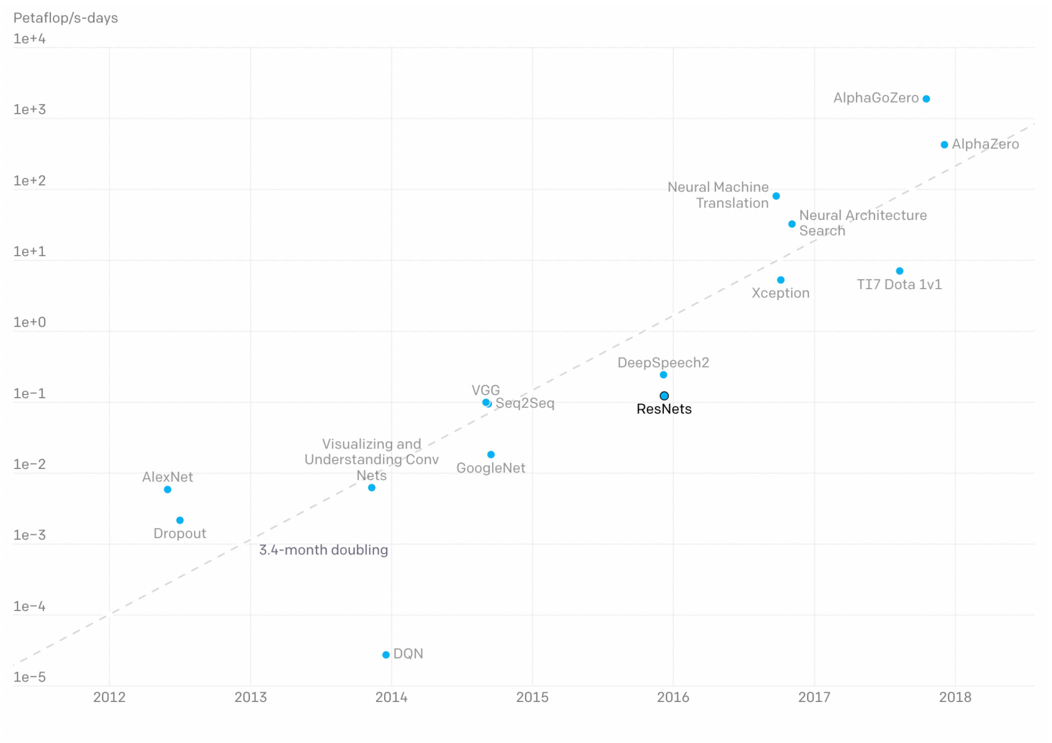
$ReLU(a) = \max(a, 0)$

Modern networks

They are huge, and training can take time.

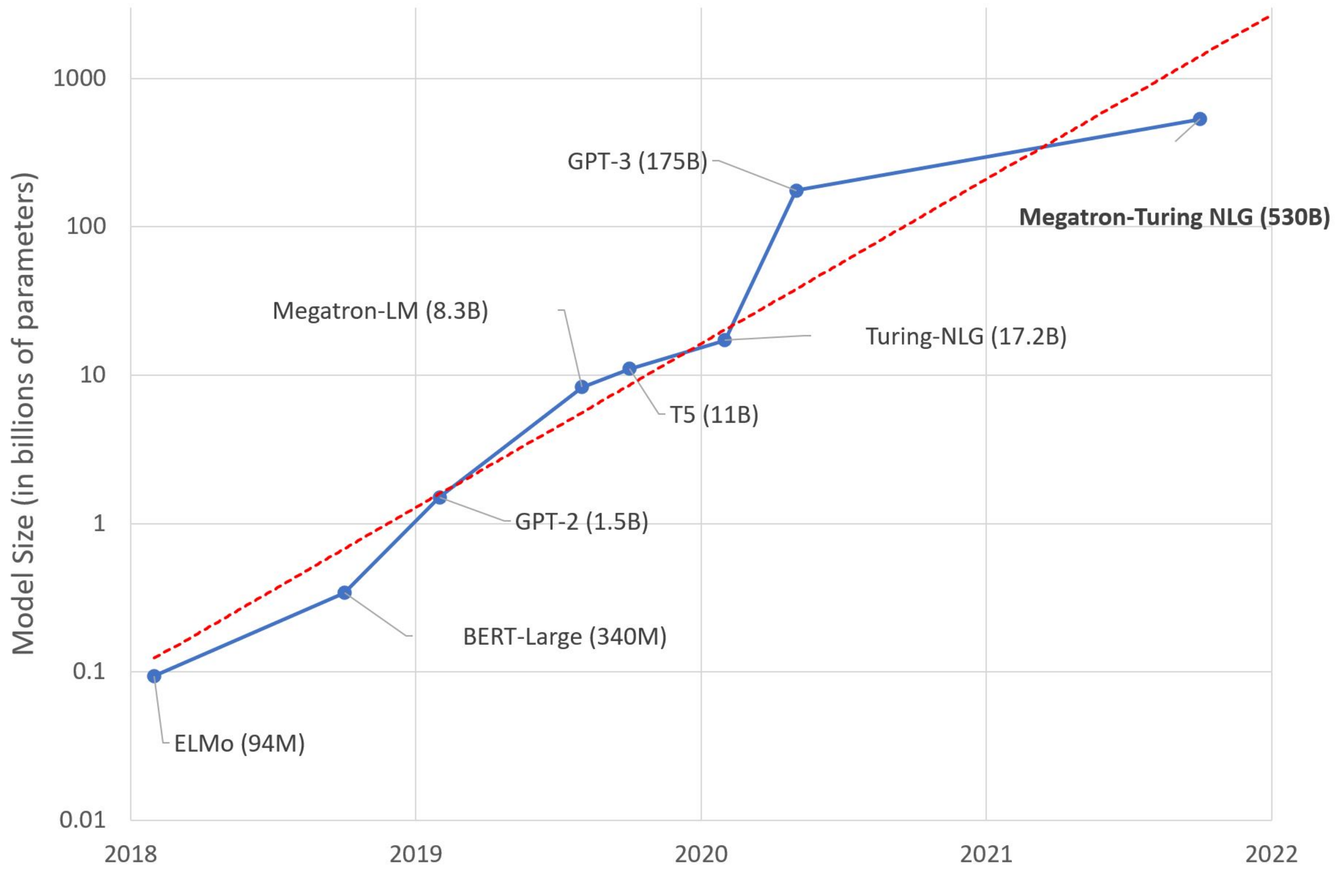
Since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.4-month doubling time (by comparison, Moore's Law had a 2-year doubling period). Since 2012, this metric has grown by more than 300,000x (a 2-year doubling period would yield only a 7x increase).

https://openai.com/blog/ai-and-compute/



The total amount of compute, in petaflop/s-days,[2] used to train selected results that are relatively well known, used a lot of compute for their time, and gave enough information to estimate the compute used.

https://huggingface.co/blog/large-language-models



Optimization: Variants on SGD

Mini-batch: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)

Consider $F(w) = \sum_{i=1}^{n} F_i(w)$, where $F_i(w)$ is the loss function for the i-th datapoint.

Recall that any $\nabla \tilde{F}(w)$ is a stochastic gradient of $F(w)$ if

$$\mathbb{E}[\nabla \tilde{F}(w)] = \nabla F(w)$$

Mini-batch SGD (also known as mini-batch GD): sample $S \subset 1, \cdots, n$ at random, and estimate the average gradient over these batch of $|S|$ samples:

$$\nabla \tilde{F}(w) = \frac{1}{|S|} \sum_{j \in S} \nabla F_j(w)$$

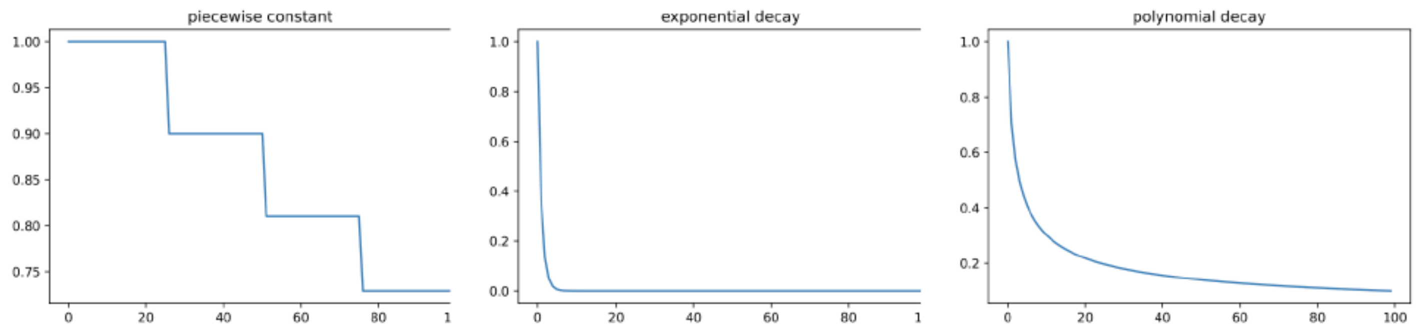
Common batch size: $32, 64, 128$, etc.

With $S = 1$, $Mini - batch\ GD \Rightarrow SGD$

Batch size s.t. batch fits in "GPU memory".

Adaptive learning rate tuning: choose a different learning rate for each parameter (and vary this across iterations),based on the magnitude of previous gradients for that parameter (used in Adagrad, RMSProp)

We often use a learning rate schedule.



Adaptive learning rate methods (Adagrad, RMSProp) scale the learning rate of each parameter based on some moving average of the magnitude of the gradients.

Momentum: add a "momentum" term to encourage model to continue along previous gradient direction.

"move faster along directions that were previously good, and to slow down along directions where the gradient has suddenly changed, just like a ball rolling downhill." [PML]

Initialize w_0 and (velocity) $v = 0$

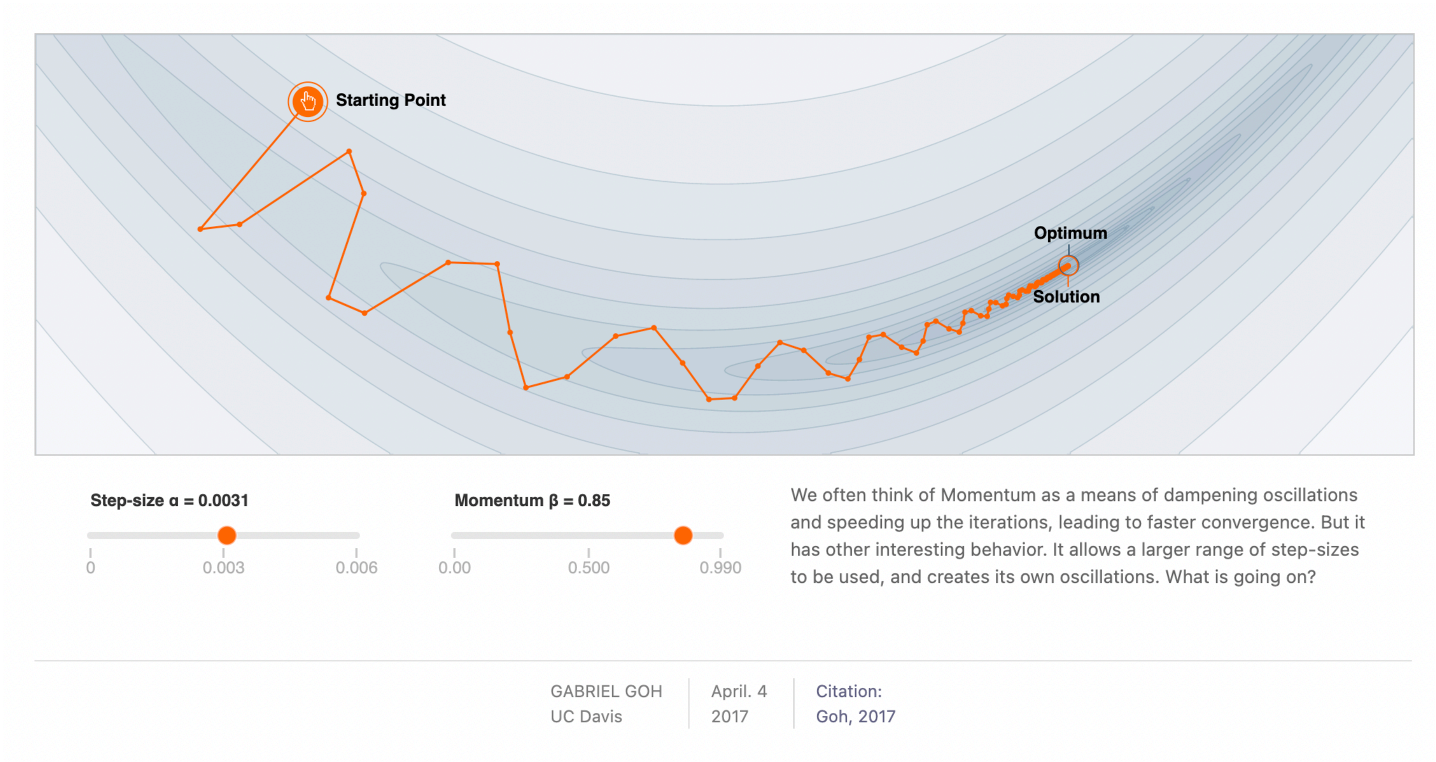
For $t = 1, 2, \ldots$

- estimate a stochastic gradient g_t
- update $v_t \leftarrow \alpha v_{t-1} + g_t$ for some discount factor ! " $\alpha \in (0, 1)$
- update weight $w_t \leftarrow w_{t-1} - \eta v_t$

Updates for first few rounds:

- $w_1 = w_0 - \eta g_1$
- $w_2 = w_1 - \alpha\eta g_1 - \eta g_2$ ($v = \alpha g_1 + g_2$)
- $w_3 = w_2 - \alpha^2\eta g_1 - \alpha\eta g_2 - \eta g_3$ ($v = \alpha^2 g_1 + \alpha g_2 + g_3$)
- $\ldots$

Why momentum really works? https://distill.pub/2017/momentum/



Step-size α = 0.0031

Momentum β = 0.85

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

Many other variants and tricks such as **batch normalization**: normalize the inputs of each layer over the mini-batch (to zero-mean and unit-variance; like we did in HW1)

Generalization: Preventing Overfitting

Overfitting can be a major concern since neural nets are very powerful.

Methods to overcome overfitting:

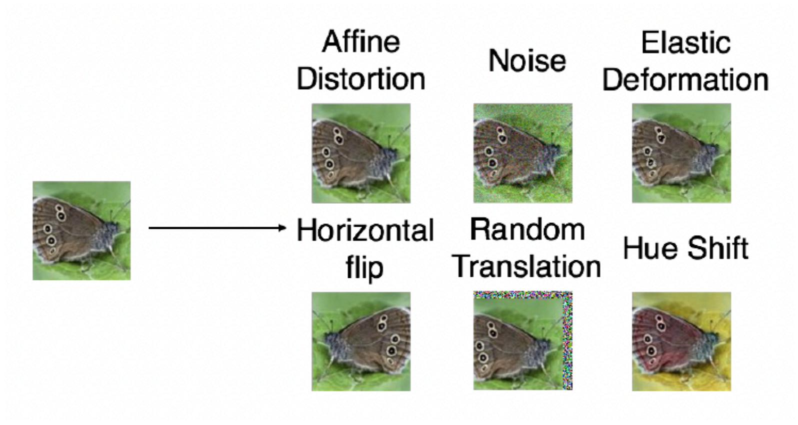
- data augmentation
- regularization
- dropout
- early stopping

Data augmentation:

The best way to prevent overfitting? Get more samples.

What if you cannot get access to more samples?

Exploit prior knowledge to add more training data:

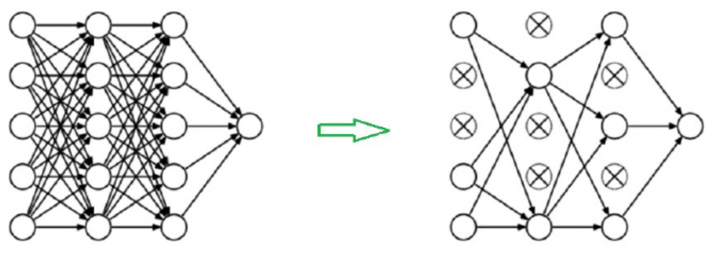


Regularization & Dropout: We can use regularization techniques such as l_2 regularization.

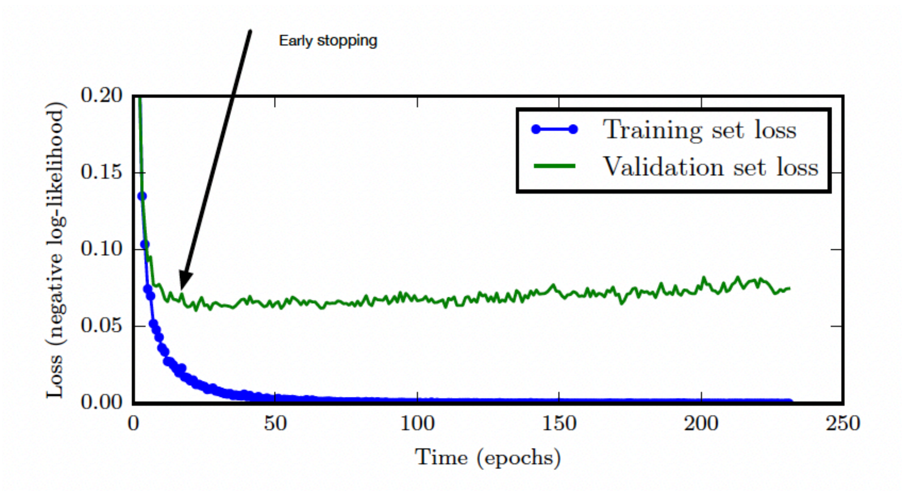
l_2 regularization: minimize

$$G(W_1, \cdots, W_L) = F(W_1, \cdots, W_L) + \lambda \sum w^2$$

A very popular technique is Dropout. Here, we independently delete each neuron with a fixed probability (say 0.1), during each iteration of Backprop (only for training, not for testing)



Early stopping: Stop training when the performance on validations set stops improving



Neural Networks Summary

There are big mysteries about how and why deep learning works

- Why are certain architectures better for certain problems? How should we design architectures?
- Why do gradient-based methods work on these highly-nonconvex problems?
- Why can deep networks generalize well despite having the capacity to so easily overfit?
- What implicit regularization effects do gradient-based methods provide?

Deep neural networks

- are hugely popular, achieving best performance on many problem.
- do need a lot of data to work well.
- can take a lot of time to train (need GPUs for massive parallel computing).
- take some work to select architecture and hyperparameters.
- are still not well understood in theory.