Machine Learning

A. G. Schwing & M. Telgarsky

University of Illinois at Urbana-Champaign, 2018

L8: Deep Neural Networks.

Lecture outline.

1. Review & motivation.
2. Basic neural networks.
3. Some modern usage.

Reading.

- I. Goodfellow et al.; Deep Learning; Chapters 6-9.

Review & Motivation.

Review.

Lecture 1. Basic ML; k-nn (k nearest neighbor).

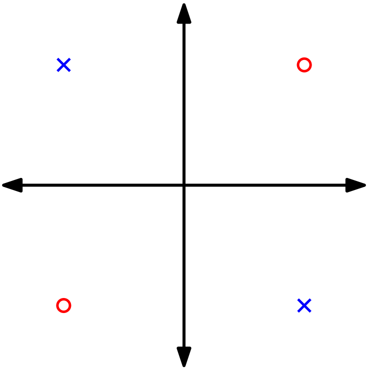
Lectures 2, 3, 6. Linear predictors
(least squares, logistic regression, SVM).

Lectures 4, 5. Convexity and optimization
(e.g., *how we can learn* linear predictors).

Lecture 7. Multiclass and *nonlinear* (kernel) SVM.

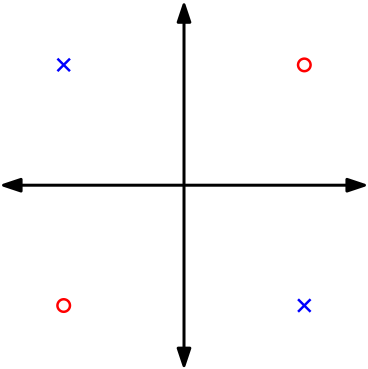
Lecture 8. **Deep neural networks.**

Limitations of linear predictors?



No linear separator classifies perfectly!

Limitations of linear predictors?



No linear separator classifies perfectly!

Magic fix:

use features $\phi(\boldsymbol{x}) := \boldsymbol{x}_1 \cdot \boldsymbol{x}_2$,

whereby $y = \operatorname{sgn}\left(\boldsymbol{w}^\top \phi(\boldsymbol{x})\right)$ with $\boldsymbol{w} = [1] \in \mathbb{R}^1$.

Kernel SVM review.

- Kernel SVM can be trained in the dual *kernels*:

$$\max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\left\|\sum_{i=1}^n \alpha_i y^{(i)}\phi(\mathbf{x}^{(i)})\right\|^2$$

$$= \max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y^{(i)}y^{(j)}(\phi(\mathbf{x}^{(i)}))^\top\phi(\mathbf{x}^{(j)})$$

$$= \max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y^{(i)}y^{(j)}\kappa(\mathbf{x}^{(i)},\mathbf{x}^{(j)}).$$

Why do this?

Kernel SVM review.

- Kernel SVM can be trained in the dual *kernels*:

$$\max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\left\|\sum_{i=1}^n \alpha_i y^{(i)} \phi(\mathbf{x}^{(i)})\right\|^2$$

$$= \max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y^{(i)} y^{(j)} (\phi(\mathbf{x}^{(i)}))^\top \phi(\mathbf{x}^{(j)})$$

$$= \max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y^{(i)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}).$$

Why do this? When writing out $\phi(\boldsymbol{x})$ is expensive.

Kernel SVM review.

- Kernel SVM can be trained in the dual *kernels*:

$$\max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^{n} \boldsymbol{\alpha}_i - \frac{1}{2}\sum_{i,j=1}^{n} \boldsymbol{\alpha}_i \boldsymbol{\alpha}_j y^{(i)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}).$$

Why do this? When writing out $\phi(\boldsymbol{x})$ is expensive.

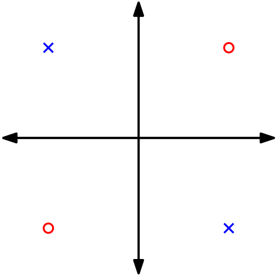
Kernel SVM review.

- Kernel SVM can be trained in the dual *kernels*:

$$\max_{\boldsymbol{\alpha} \in [0,C]^n} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}).$$

  Why do this? When writing out $\phi(\boldsymbol{x})$ is expensive.

- Back to our example, which needed "magic feature" $\phi(\boldsymbol{x}) = \boldsymbol{x}_1 \cdot \boldsymbol{x}_2$.



  We can use *polynomial kernel* κ.

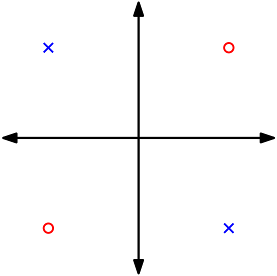
Kernel SVM review.

- Kernel SVM can be trained in the dual *kernels*:

$$\max_{\alpha \in [0,C]^n} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}).$$

  Why do this? When writing out $\phi(\boldsymbol{x})$ is expensive.

- Back to our example, which needed "magic feature" $\phi(\boldsymbol{x}) = \boldsymbol{x}_1 \cdot \boldsymbol{x}_2$.



  We can use *polynomial kernel* κ. *But this is still magic...*

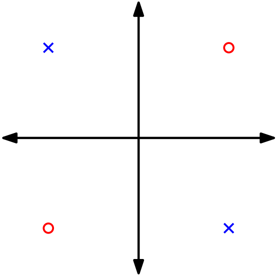
Kernel SVM review.

- Kernel SVM can be trained in the dual *kernels*:

$$\max_{\boldsymbol{\alpha}\in[0,C]^n} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i\alpha_j y^{(i)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}).$$

  Why do this? When writing out $\phi(\boldsymbol{x})$ is expensive.

- Back to our example, which needed "magic feature" $\phi(\boldsymbol{x}) = \boldsymbol{x}_1 \cdot \boldsymbol{x}_2$.



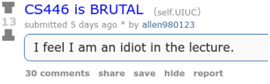
  We can use *polynomial kernel* κ. *But this is still magic...*

  Solution (today's lecture)**:** *Learn* ϕ.

End of review; another aside…

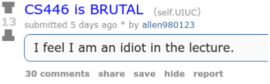
End of review; another aside…

- After lecture 4:



End of review; another aside. . .

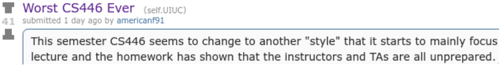
- After lecture 4:


CS446 is BRUTAL (self.UIUC)
13
submitted 5 days ago * by allen980123

I feel I am an idiot in the lecture.

30 comments share save hide report

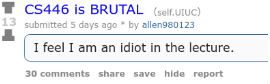
- After lecture 6:


Worst CS446 Ever (self.UIUC)
41
submitted 1 day ago by americanf91

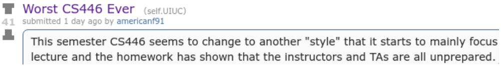
This semester CS446 seems to change to another "style" that it starts to mainly focus
lecture and the homework has shown that the instructors and TAs are all unprepared.

End of review; another aside…

- After lecture 4:



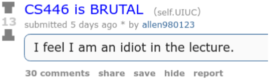
- After lecture 6:



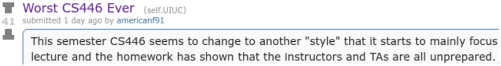
- After lecture 8: ???

End of review; another aside. . .

- After lecture 4:



- After lecture 6:



- After lecture 8: ???

Observations.

- Everything explicitly in threads.
- Implicit: communication insufficient (e.g., piazza).
- Implicit: homeworks not fun.
- What else?

Second aside: naming.

Artificial neural networks. (8 syllables.)

Neural networks. (4 syllables.)

Deep nets. (2 syllables.)

Basic neural networks.

Neural networks via *features*.

To make a linear predictor nonlinear, we rely upon feature mapping ϕ:

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{w}^\top \phi(\boldsymbol{x}).$$

We are at the mercy of the quality of ϕ.

Neural networks via *features*.

To make a linear predictor nonlinear, we rely upon feature mapping ϕ:

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{w}^\top \phi(\boldsymbol{x}).$$

We are at the mercy of the quality of ϕ.

Why not *learn* ϕ?

Neural networks via *features*.

To make a linear predictor nonlinear, we rely upon feature mapping ϕ:

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{w}^\top \phi(\boldsymbol{x}).$$

We are at the mercy of the quality of ϕ.

Why not *learn* ϕ? e.g.,

$$\arg\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y^{(i)} \boldsymbol{w}^\top \mathbf{x}^{(i)}\right) \quad \text{becomes} \quad \arg\min_{\boldsymbol{w}, \phi \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y^{(i)} \boldsymbol{w}^\top \phi(\mathbf{x}^{(i)})\right)$$

where $\mathcal{F}$ is some class of functions **(why not every function?)**.

Neural networks as iterated linear prediction (part 1).

Natural choice: build feature maps out of linear predictors!

Neural networks as iterated linear prediction (part 1).

Natural choice: build feature maps out of linear predictors!

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{v}^\top \phi(\boldsymbol{x}) \qquad \text{where } \phi(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \mathbf{b}$$

with $\boldsymbol{w} \in \mathbb{R}^d$, $\boldsymbol{v} \in \mathbb{R}^m$, $\boldsymbol{A} \in \mathbb{R}^{m \times d}$, $\mathbf{b} \in \mathbb{R}^m$.

Neural networks as iterated linear prediction (part 1).

Natural choice: build feature maps out of linear predictors!

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{v}^\top \phi(\boldsymbol{x}) \qquad \text{where } \phi(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \mathbf{b}$$

with $\boldsymbol{w} \in \mathbb{R}^d$, $\boldsymbol{v} \in \mathbb{R}^m$, $\boldsymbol{A} \in \mathbb{R}^{m \times d}$, $\mathbf{b} \in \mathbb{R}^m$.

There is something wrong with this!

Neural networks as iterated linear prediction (part 1).

Natural choice: build feature maps out of linear predictors!

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{v}^\top \phi(\boldsymbol{x}) \quad \text{where } \phi(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \mathbf{b}$$

with $\boldsymbol{w} \in \mathbb{R}^d, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{A} \in \mathbb{R}^{m \times d}, \mathbf{b} \in \mathbb{R}^m$.

There is something wrong with this!
Gained nothing! $\boldsymbol{v}^\top(\boldsymbol{A}\boldsymbol{x} + \mathbf{b}) = (\boldsymbol{A}^\top \boldsymbol{v})^\top \boldsymbol{x} + \boldsymbol{v}^\top \mathbf{b}$.

Neural networks as iterated linear prediction (part 1).

Natural choice: build feature maps out of linear predictors!

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \qquad \text{becomes} \qquad \boldsymbol{x} \mapsto \boldsymbol{v}^\top \phi(\boldsymbol{x}) \qquad \text{where } \phi(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \mathbf{b}$$

with $\boldsymbol{w} \in \mathbb{R}^d, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{A} \in \mathbb{R}^{m \times d}, \mathbf{b} \in \mathbb{R}^m$.

There is something wrong with this!
Gained nothing! $\boldsymbol{v}^\top(\boldsymbol{A}\boldsymbol{x} + \mathbf{b}) = (\boldsymbol{A}^\top \boldsymbol{v})^\top \boldsymbol{x} + \boldsymbol{v}^\top \mathbf{b}$.

Fix: introduce **nonlinearity/transfer/activation** $\sigma : \mathbb{R}^m \to \mathbb{R}^m$:

$$\phi(\boldsymbol{x}) := \sigma\left(\boldsymbol{A}\boldsymbol{x} + \mathbf{b}\right).$$

Neural networks as iterated linear prediction (part 2).

We will predict with

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \phi(\boldsymbol{x}) \qquad \text{where } \phi(\boldsymbol{x}) = \sigma\left(\boldsymbol{A}\boldsymbol{x} + \mathbf{b}\right).$$

We will train with

$$\underset{\boldsymbol{w}\in\mathbb{R}^m, \boldsymbol{A}\in\mathbb{R}^{m\times d}, \mathbf{b}\in\mathbb{R}^m}{\arg\min} \frac{1}{n}\sum_{i=1}^{n} \ell\left(y^{(i)}\boldsymbol{w}^\top \sigma\left(\boldsymbol{A}\mathbf{x}^{(i)} + \mathbf{b}\right)\right).$$

(**Question:** which training procedure? Why does it work?)

Neural networks as iterated linear prediction (part 2).

We will predict with

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \phi(\boldsymbol{x}) \qquad \text{where } \phi(\boldsymbol{x}) = \sigma\left(\boldsymbol{A}\boldsymbol{x} + \mathbf{b}\right).$$

We will train with

$$\underset{\boldsymbol{w}\in\mathbb{R}^m, \boldsymbol{A}\in\mathbb{R}^{m\times d}, \mathbf{b}\in\mathbb{R}^m}{\arg\min} \frac{1}{n}\sum_{i=1}^{n} \ell\left(y^{(i)}\boldsymbol{w}^\top\sigma\left(\boldsymbol{A}\mathbf{x}^{(i)} + \mathbf{b}\right)\right).$$

(**Question:** which training procedure? Why does it work?)

Why stop there? We can also do

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \sigma_1\left(\boldsymbol{A}_1\phi(\boldsymbol{x}) + \mathbf{b}_1\right) \qquad \text{where } \phi(\boldsymbol{x}) = \sigma_2\left(\boldsymbol{A}_2\boldsymbol{x} + \mathbf{b}_2\right),$$

and iterate further.

Neural networks as iterated linear prediction (part 2).

We will predict with

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \phi(\boldsymbol{x}) \qquad \text{where } \phi(\boldsymbol{x}) = \sigma\left(\boldsymbol{A}\boldsymbol{x} + \mathbf{b}\right).$$

We will train with

$$\underset{\boldsymbol{w}\in\mathbb{R}^m, \boldsymbol{A}\in\mathbb{R}^{m\times d}, \mathbf{b}\in\mathbb{R}^m}{\arg\min} \frac{1}{n}\sum_{i=1}^{n} \ell\left(y^{(i)}\boldsymbol{w}^\top\sigma\left(\boldsymbol{A}\mathbf{x}^{(i)} + \mathbf{b}\right)\right).$$

(**Question:** which training procedure? Why does it work?)

Why stop there? We can also do

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \sigma_1\left(\boldsymbol{A}_1 \phi(\boldsymbol{x}) + \mathbf{b}_1\right) \qquad \text{where } \phi(\boldsymbol{x}) = \sigma_2\left(\boldsymbol{A}_2\boldsymbol{x} + \mathbf{b}_2\right),$$

and iterate further. *This is a neural network.*

Classical **formulation of neural networks as graphs.**

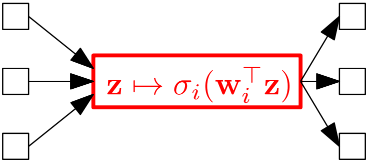
("Classical" because tensorflow "computation graphs" differ slightly.)

Classical formulation of neural networks as graphs.

("Classical" because tensorflow "computation graphs" differ slightly.)

Node j in this graph:

- Collects a vector $\boldsymbol{z}$ from its in-edges;
- Computes $\boldsymbol{z} \mapsto \sigma_j(\boldsymbol{w}_j^\top \boldsymbol{z} + b_j)$;
- Propagates this value along its out-edges.

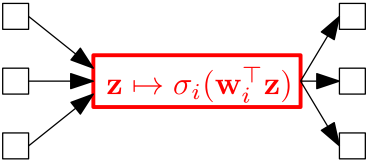


Classical formulation of neural networks as graphs.

("Classical" because tensorflow "computation graphs" differ slightly.)

Node j in this graph:

- Collects a vector $\boldsymbol{z}$ from its in-edges;
- Computes $\boldsymbol{z} \mapsto \sigma_j(\boldsymbol{w}_j^\top \boldsymbol{z} + b_j)$;
- Propagates this value along its out-edges.



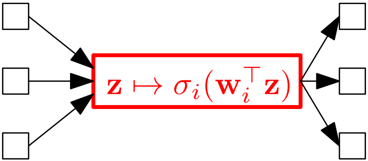
Computation of whole network can be written this way.

Classical formulation of neural networks as graphs.

("Classical" because tensorflow "computation graphs" differ slightly.)

Node j in this graph:

- Collects a vector $\boldsymbol{z}$ from its in-edges;
- Computes $\boldsymbol{z} \mapsto \sigma_j(\boldsymbol{w}_j^\top \boldsymbol{z} + b_j)$;
- Propagates this value along its out-edges.



Computation of whole network can be written this way.

Tensorflow computation graphs: everything needed to train is in the graph; e.g., parameters get nodes.

Neural networks as functions.

A linear predictor (**one layer network**) has the form

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x}.$$

A **two layer network** has the form

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \sigma_1\left(\boldsymbol{A}_1 \boldsymbol{x} + \mathbf{b}_1\right).$$

Iterating, a **multi-layer network** has the form

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \sigma_1\left(\boldsymbol{A}_1 \sigma_2\left(\cdots \boldsymbol{A}_{L-2}\sigma_{L-1}\left(\boldsymbol{A}_{L-1}\boldsymbol{x} + \mathbf{b}_{L-1}\right) + \mathbf{b}_{L-2}\cdots\right) + \mathbf{b}_1\right).$$

Neural networks as functions.

A linear predictor (**one layer network**) has the form

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x}.$$

A **two layer network** has the form

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \sigma_1\left(\boldsymbol{A}_1 \boldsymbol{x} + \mathbf{b}_1\right).$$

Iterating, a **multi-layer network** has the form

$$\boldsymbol{x} \mapsto \boldsymbol{w}^\top \sigma_1\left(\boldsymbol{A}_1 \sigma_2\left(\cdots \boldsymbol{A}_{L-2} \sigma_{L-1}\left(\boldsymbol{A}_{L-1} \boldsymbol{x} + \mathbf{b}_{L-1}\right) + \mathbf{b}_{L-2} \cdots\right) + \mathbf{b}_1\right).$$

ERM now takes the form

$$\arg\min_{\boldsymbol{w}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_{L-1}, \mathbf{b}_1, \ldots, \mathbf{b}_{L-1}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y^{(i)} \boldsymbol{w}^\top \sigma_1\left(\cdots \sigma_{L-1}\left(\boldsymbol{A}_{L-1} \mathbf{x}^{(i)} + \mathbf{b}_{L-1}\right) \cdots\right)\right).$$

Neural network (univariate) activations.

We mentioned that **nodes** compute

$$\boldsymbol{z} \mapsto \sigma\left(\boldsymbol{v}^\top \boldsymbol{z}\right),$$

where *activation/transfer/nonlinearity* $\sigma : \mathbb{R} \to \mathbb{R}$ is:

- ReLU (Rectified Linear Unit) $z \mapsto \max\{0, z\}$;
- Sigmoid $z \mapsto \frac{1}{1+\exp(-z)}$;
- $\ldots$.

By $\boldsymbol{z} \mapsto \sigma(\boldsymbol{A}\boldsymbol{z} + \mathbf{b})$,
we meant "apply a univariate σ coordinate-wise".

Neural network (univariate) activations.

We mentioned that **nodes** compute

$$\boldsymbol{z} \mapsto \sigma\left(\boldsymbol{v}^\top \boldsymbol{z}\right),$$

where *activation/transfer/nonlinearity* $\sigma : \mathbb{R} \to \mathbb{R}$ is:

- ReLU (Rectified Linear Unit) $z \mapsto \max\{0, z\}$;
- Sigmoid $z \mapsto \frac{1}{1+\exp(-z)}$;
- ....

By $\boldsymbol{z} \mapsto \sigma(\boldsymbol{A}\boldsymbol{z} + \mathbf{b})$,
we meant "apply a univariate σ coordinate-wise".

Soon we will see multivariate nonlinearities,
sometimes with output dimension $\neq$ input dimension!

Some modern usage.

Multiclass output.

Modern networks often end with **softmax** nonlinearity:

$$\boldsymbol{z} \mapsto \sum_{i=1}^{k} \frac{\exp(\boldsymbol{z}_i)\mathbf{e}_i}{\sum_{j=1}^{k} \exp(\boldsymbol{z}_j)}$$

(where $\mathbf{e}_i$ is i^{th} standard basis vector.)
Output is now a probability vector!

Multiclass output.

Modern networks often end with **softmax** nonlinearity:

$$\boldsymbol{z} \mapsto \sum_{i=1}^{k} \frac{\exp(\boldsymbol{z}_i)\mathbf{e}_i}{\sum_{j=1}^{k} \exp(\boldsymbol{z}_j)}$$

(where $\mathbf{e}_i$ is i^{th} standard basis vector.)
Output is now a probability vector!

Alternate notation: output vector $\boldsymbol{v}_i \propto \exp(\boldsymbol{z}_i)$.

Cross-entropy loss.

Given *one hot* $\boldsymbol{y} \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$ and probability vector $\hat{\boldsymbol{y}} \in \mathbb{R}^k$,

$$\ell(y, \hat{y}) = -\sum_{i=1}^{k} \boldsymbol{y}_i \ln(\hat{\boldsymbol{y}}).$$

Combined with softmax $\hat{\boldsymbol{y}} \propto \exp(\boldsymbol{z})$:

$$-\sum_{i=1}^{k} \boldsymbol{y}_i \ln\left(\frac{\exp(\boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{z}_j)}\right) = -\sum_{i=1}^{k} \boldsymbol{y}_i \boldsymbol{z}_i + \ln\left(\sum_{i=1}^{k} \exp(\boldsymbol{z}_i)\right).$$

(For numerical stability, use $\ln \sum_i \exp v_i = c + \ln \sum_i \exp(v_i - c)$.)

Cross-entropy loss.

Given *one hot* $\boldsymbol{y} \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$ and probability vector $\hat{\boldsymbol{y}} \in \mathbb{R}^k$,

$$\ell(y, \hat{y}) = -\sum_{i=1}^{k} \boldsymbol{y}_i \ln(\hat{\boldsymbol{y}}).$$

Combined with softmax $\hat{\boldsymbol{y}} \propto \exp(\boldsymbol{z})$:

$$-\sum_{i=1}^{k} \boldsymbol{y}_i \ln\left(\frac{\exp(\boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{z}_j)}\right) = -\sum_{i=1}^{k} \boldsymbol{y}_i \boldsymbol{z}_i + \ln\left(\sum_{i=1}^{k} \exp(\boldsymbol{z}_i)\right).$$

(For numerical stability, use $\ln \sum_i \exp v_i = c + \ln \sum_i \exp(v_i - c)$.)

(In binary case $k = 2$: generalizes logistic loss.)

Cross-entropy loss.

Given *one hot* $\boldsymbol{y} \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$ and probability vector $\hat{\boldsymbol{y}} \in \mathbb{R}^k$,

$$\ell(y, \hat{y}) = -\sum_{i=1}^{k} \boldsymbol{y}_i \ln(\hat{\boldsymbol{y}}).$$

Combined with softmax $\hat{\boldsymbol{y}} \propto \exp(\boldsymbol{z})$:

$$-\sum_{i=1}^{k} \boldsymbol{y}_i \ln\left(\frac{\exp(\boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{z}_j)}\right) = -\sum_{i=1}^{k} \boldsymbol{y}_i \boldsymbol{z}_i + \ln\left(\sum_{i=1}^{k} \exp(\boldsymbol{z}_i)\right).$$

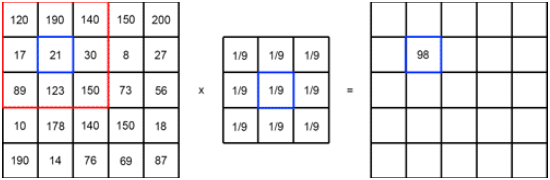
(For numerical stability, use $\ln \sum_i \exp v_i = c + \ln \sum_i \exp(v_i - c)$.)

(In binary case $k = 2$: generalizes logistic loss.)

Question: since last expression is convex in $\boldsymbol{z}$,
is the corresponding ERM problem convex?

Convolutions.

Convolve input with a filter:



Convolutions.

Convolve input with a filter:

120	190	140	150	200
17	21	30	8	27
89	123	150	73	56
10	178	140	150	18
190	14	76	69	87

x

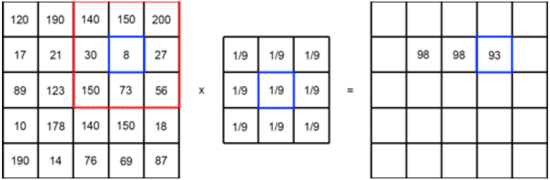
1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

=

	98	98		

Convolutions.

Convolve input with a filter:



Convolutions.

Convolve input with a filter:

$$
\begin{array}{|c|c|c|c|c|}
\hline
120 & 190 & 140 & 150 & 200 \\ \hline
17 & 21 & 30 & 8 & 27 \\ \hline
89 & \boxed{123} & 150 & 73 & 56 \\ \hline
10 & 178 & 140 & 150 & 18 \\ \hline
190 & 14 & 76 & 69 & 87 \\ \hline
\end{array}
\;\times\;
\begin{array}{|c|c|c|}
\hline
1/9 & 1/9 & 1/9 \\ \hline
1/9 & \boxed{1/9} & 1/9 \\ \hline
1/9 & 1/9 & 1/9 \\ \hline
\end{array}
\;=\;
\begin{array}{|c|c|c|c|c|}
\hline
 & & & & \\ \hline
 & 98 & 98 & 93 & \\ \hline
 & \boxed{84} & & & \\ \hline
 & & & & \\ \hline
 & & & & \\ \hline
\end{array}
$$

Convolutions.

Convolve input with a filter:

120	190	140	150	200
17	21	30	8	27
89	123	150	73	56
10	178	140	150	18
190	14	76	69	87

x

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

=

	98	98	93	
	84	97		

Convolutions.

Convolve input with a filter:

$$\begin{bmatrix} 120 & 190 & 140 & 150 & 200 \\ 17 & 21 & 30 & 8 & 27 \\ 89 & 123 & 150 & \boxed{73} & 56 \\ 10 & 178 & 140 & 150 & 18 \\ 190 & 14 & 76 & 69 & 87 \end{bmatrix} \times \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & \boxed{1/9} & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \begin{bmatrix} \; & \; & \; & \; & \; \\ & 98 & 98 & 93 & \\ & 84 & 97 & \boxed{72} & \\ & & & & \\ & & & & \end{bmatrix}$$

Convolutions.

Convolve input with a filter:

120	190	140	150	200
17	21	30	8	27
89	123	150	73	56
10	178	140	150	18
190	14	76	69	87

x

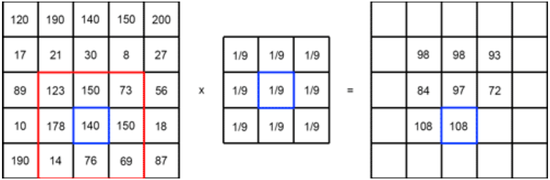
1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

=

	98	98	93	
	84	97	72	
	108			

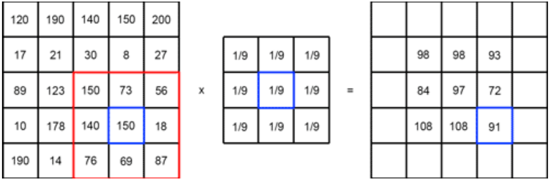
Convolutions.

Convolve input with a filter:



Convolutions.

Convolve input with a filter:



Convolutions.

Convolve input with a filter:

120	190	140	150	200
17	21	30	8	27
89	123	150	73	56
10	178	140	150	18
190	14	76	69	87

x

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

=

	98	98	93	
	84	97	72	
	108	108	91	

Written as matrix-vector product $\boldsymbol{Ax}$:

$$\begin{bmatrix} \text{offset 0} & & & \\ & \text{offset 1} & & \\ & & \text{offset 2} & \\ & & & \text{offset 3} \end{bmatrix} \begin{bmatrix} \uparrow \\ \boldsymbol{x} \\ \downarrow \end{bmatrix}.$$

Convolutions.

Convolve input with a filter:

$$\begin{bmatrix} 120 & 190 & 140 & 150 & 200 \\ 17 & 21 & 30 & 8 & 27 \\ 89 & 123 & 150 & 73 & 56 \\ 10 & 178 & 140 & 150 & 18 \\ 190 & 14 & 76 & 69 & 87 \end{bmatrix} \times \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \begin{bmatrix} & & & & \\ & 98 & 98 & 93 & \\ & 84 & 97 & 72 & \\ & 108 & 108 & 91 & \\ & & & & \end{bmatrix}$$

Written as matrix-vector product $\boldsymbol{Ax}$:

$$\begin{bmatrix} \text{offset 0} & & & \\ & \text{offset 1} & & \\ & & \text{offset 2} & \\ & & & \text{offset 3} \end{bmatrix} \begin{bmatrix} \uparrow \\ \boldsymbol{x} \\ \downarrow \end{bmatrix}.$$

Why? Major space savings (#params = filter size).
Magical effectiveness on real-world data.

Pooling.

Again slide a window over the input;
now take average or maximum.

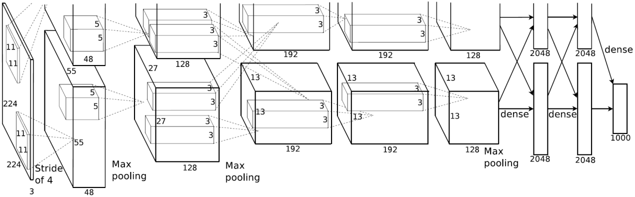
Pooling.

Again slide a window over the input;
now take average or maximum.

Why? Major space savings (output dim < input dim).
Effectiveness on real-world data.

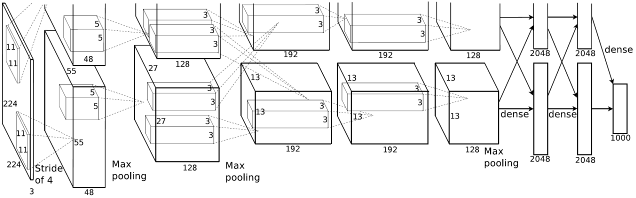
Alexnet.

Alexnet.



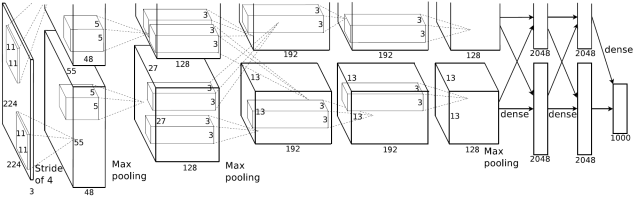
11
11
224
11
11
224
Stride of 4
3
5
5
48
55
5
5
55
48
Max pooling
3
3
3
128
27
3
3
27
3
3
128
Max pooling
3
3
192
13
13
3
3
192
3
3
192
13
13
3
3
192
128
13
13
128
Max pooling
dense
2048
2048
2048
2048
dense
dense
1000

Alexnet.



. . . *What?* Modern version:

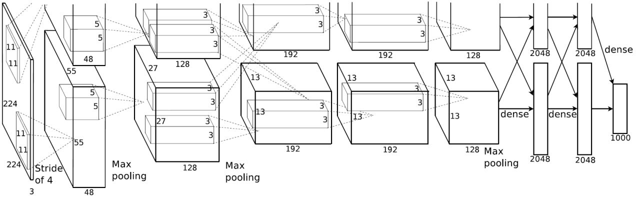
Alexnet.



. . . *What?* Modern version:

2dconv, relu, 2dmaxpool;
2dconv, relu, 2dmaxpool;
dense, relu;
dense, relu;
dense, softmax.

Alexnet.



... *What?* Modern version:

2dconv, relu, 2dmaxpool;
2dconv, relu, 2dmaxpool;
dense, relu;
dense, relu;
dense, softmax.

Differences with original: no "two "tubes" (there for GPUs), no normalization, filter size and stride tweaks...

Regularization.

- "Weight decay": $+\frac{\lambda}{2}\|\boldsymbol{v}\|^2$ in objective ($\boldsymbol{v}$ = all params).
- Dropout: randomly nullify node outputs in training.
- Batch normalization: "standardize" node output distribution.
- Use SGD! (Implicit regularization.)

Advanced topics.

- Recurrent links; loops.
- Conditional execution.
- "Differentiable" programming.

Summary.

Deep networks
as learning features; iterated linear predictors; graphs.

Cross-entropy loss, convolution layers, max-pooling.