Carnegie Mellon University
Language Technologies Institute

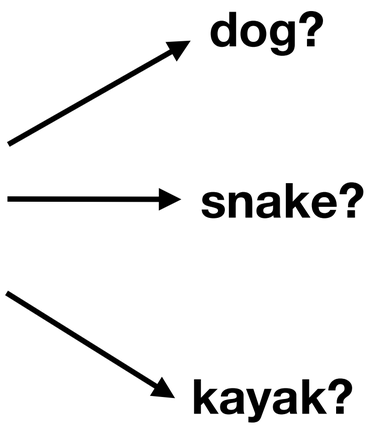
Lecture 3: ML + Neural Networks

Crash course from the lens of computational efficiency

Yonatan Bisk & Emma Strubell

Classification + Supervised Learning

1. **Dataset:** Collection of labeled samples $D : \{x_i, y_i\}$
2. **Training:** Learn classifier on training set
3. **Testing:** Evaluate classifier on held-out test set



Dataset

Nearest Neighbor Classification

- Find the closest sample(s) in my data — return the label

Training: O(1)

Inference: O(n)

Space: O(n)

Efficiency considerations:

- How expensive is your distance metric?
- Choice by threshold (“close enough”) or max?
- How memory intensive is your data?
- How redundant is your data? (K exemplars where K << N ?)

Linear Classification

- Instead of representing each label with its examples, we can *parameterize* the model with a vector of learned parameters for each label:

$$\psi(\boldsymbol{x}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) = \sum_{j=1} \theta_j \times f_j(\boldsymbol{x}, y),$$

  where $\boldsymbol{\theta}$ is a vector of weights, and $\boldsymbol{f}$ is a **feature function**.

- Now have a matrix of parameters mapping inputs to outputs (label embeddings).

- Need to learn good parameters by minimizing a **loss function** on a labeled dataset. A general strategy for minimization is **gradient descent**:

stepsize — loss fn — labels

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{i=1}^{N} \ell(\boldsymbol{\theta}^{(t)}; \boldsymbol{x}^{(i)}, y^{(i)})$$

parameters at time t, t+1 — examples — inputs

The perceptron classifier

- A simple learning rule:
  - Run the current classifier on an instance in the training data, obtaining $\hat{y} = \textsf{argmax}_y \, \psi(\boldsymbol{x}^{(i)}, y)$
  - If the prediction is incorrect:
    - Increase the weights for the features of the true label $y^{(i)}$
    - Decrease the weights for the features of the predicted label $\hat{y}$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$$

  - Repeat until all training instances are correctly classified (or you run out of time)
- If the dataset is **linearly separable** — if there is some θ that correctly labels all the training instances — then this method is guaranteed to find it.

Perceptron for Online On-Device Learning

- Simple update rule (no gradients, learning rates, …) just arithmetic
- Can be used with a more expressive encoder / feature space

Place feature extractor in hardware (e.g. audio FFT, sensor readings, …)

Tune classifier based on daily interactions

Loss Functions are not created equal

$$Err = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Mean Squared Error

$$Err = -1 * log\frac{e^{x_{n,y_n}}}{\sum_{c=1}^{C} e^{x_{n,c}}}$$

Cross Entropy Loss

$$Err = -1\sum_{c=1}^{C} log\frac{e^{x_{n,c}}}{\sum_{i=1}^{C} e^{x_{n,i}}}y_{n,c}$$

Non 1-hot Cross Entropy Loss

➡ What about contrastive learning?

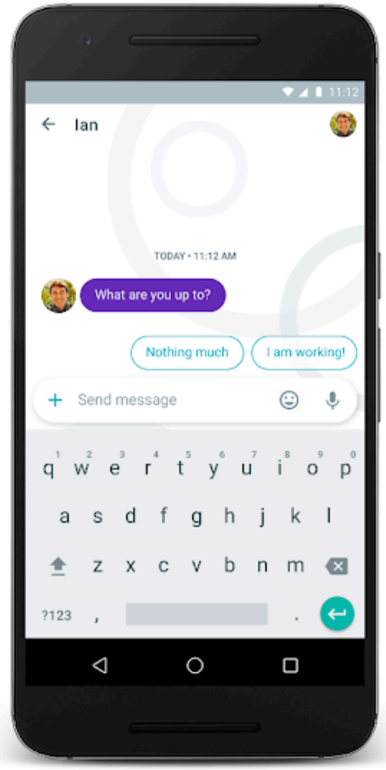
Softmax inefficiency

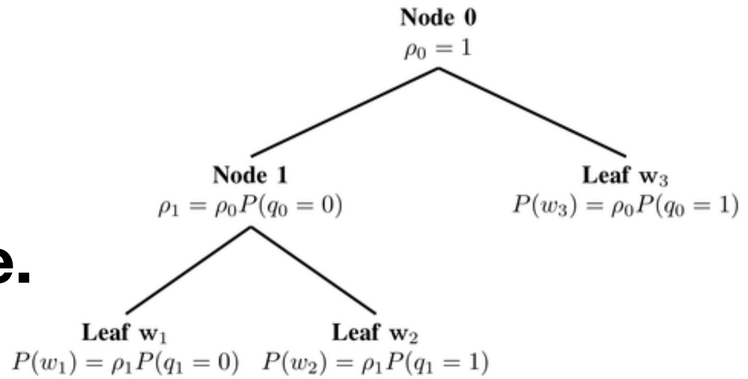
Problem:
Efficiency bottleneck is softmax (too many labels).

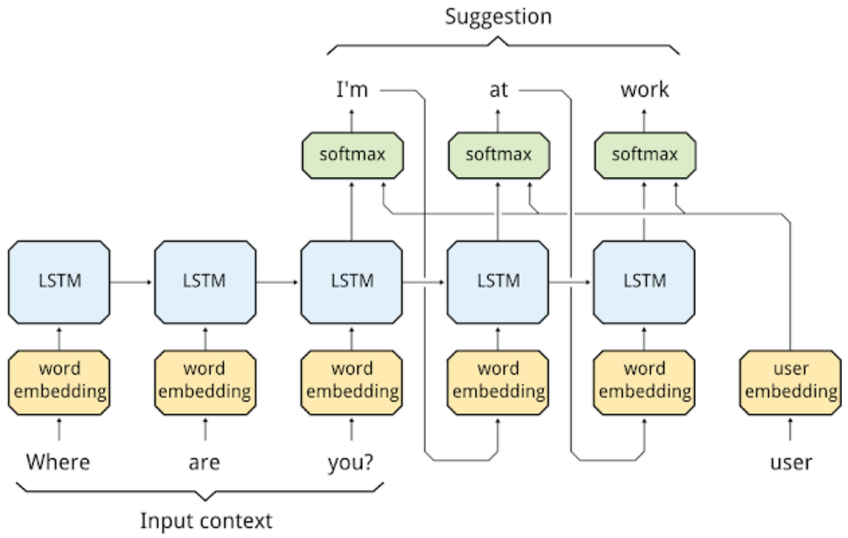
Solution:
Organize label embeddings in (binary) tree.

- Hierarchical softmax:
  Morin & Bengio (2005), Mnih & Hinton (2008)
- Differentiated softmax (D-Softmax; Chen et al. 2015)
- Adaptive softmax (Grave et al. 2017)
- Adaptive input representations (Baevski & Auli 2019)

Real use case:
Reducing Smart Reply latency from seconds to < 200ms on phone.







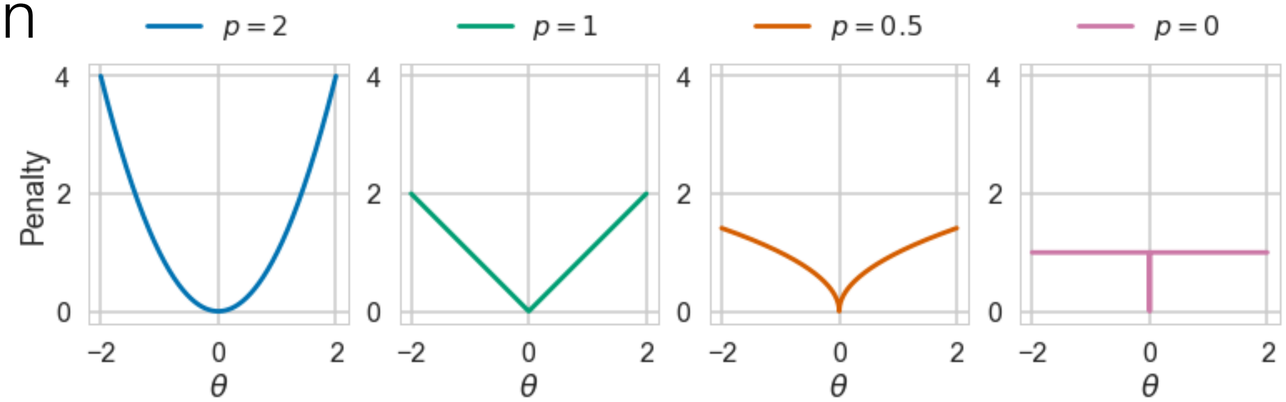
Figure/example: https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html

Efficiency considerations: Regularization

- Learning often made more robust by **regularization**: penalizing large weights. E.g. L_2 regularization:

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ell_{\text{LogReg}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) + \lambda ||\boldsymbol{\theta}||_2^2 \qquad ||\boldsymbol{\theta}||_2^2 = \sum_j \theta_j^2$$

- L_1 regularization: $\sum_j |\theta_j|$ shrinks parameters towards 0; encourages sparsity.
- L_0 regularization: $|\theta|$ explicitly penalizes # parameters — but not differentiable.
- Louizos et al. 2018: Reparameterization trick allows learning sparse neural networks from scratch w/ L_0



L_p norm penalties for a parameter θ

Feed Forward Neural Networks

- We can add more layers of model parameters (capacity) separated by nonlinear activation functions (expressivity) to learn better features from the data.

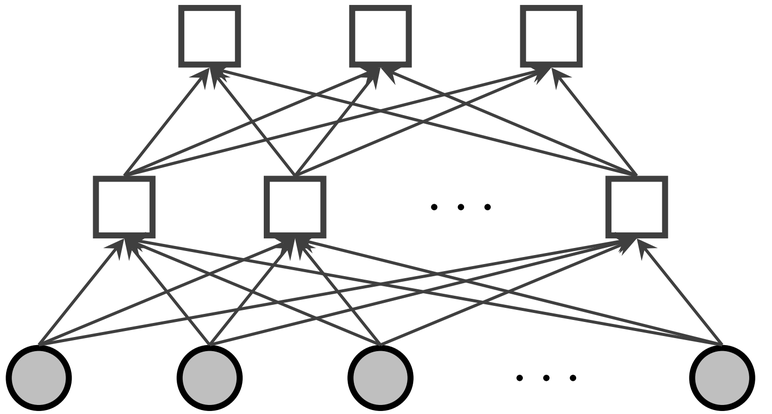
nonlinear activation fn, e.g. sigmoid σ

$$\boldsymbol{z} = \sigma(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x})$$

matrix-vector product. dims: [k, V] * [V, 1] = [k, 1]

$$p(\boldsymbol{y} \mid \boldsymbol{z}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z \to y)} \boldsymbol{z} + \boldsymbol{b})$$

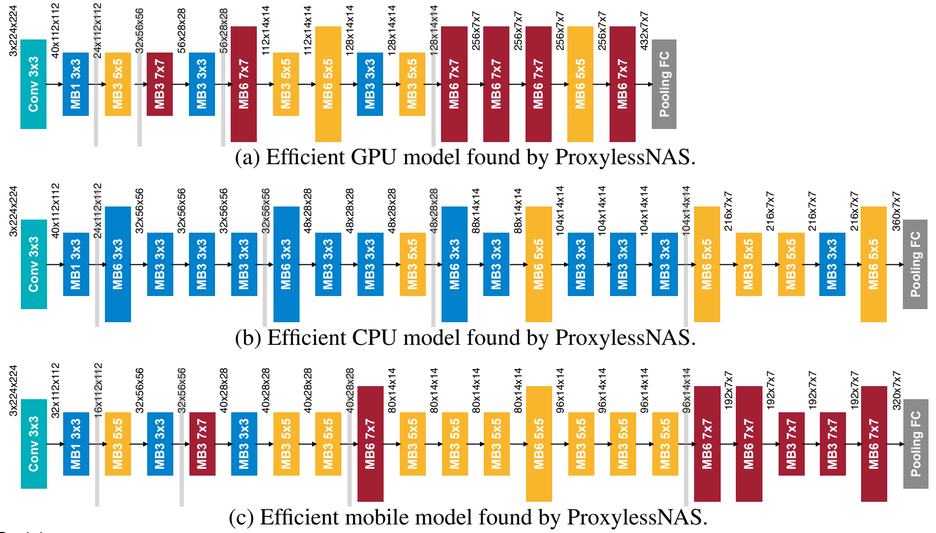
linear classifier (logistic regression)



- This makes p(*y* | ***x***) a complex, nonlinear function of ***x***.
- We can have multiple hidden layers $\boldsymbol{z}^{(1)}$, $\boldsymbol{z}^{(2)}$, … adding more expressiveness.

FFNNs: Efficiency considerations

- Given a parameter budget, should the network be wide or deep?
- Hardware constraints vs. end-task performance.
- Impact of input size compared to linear?

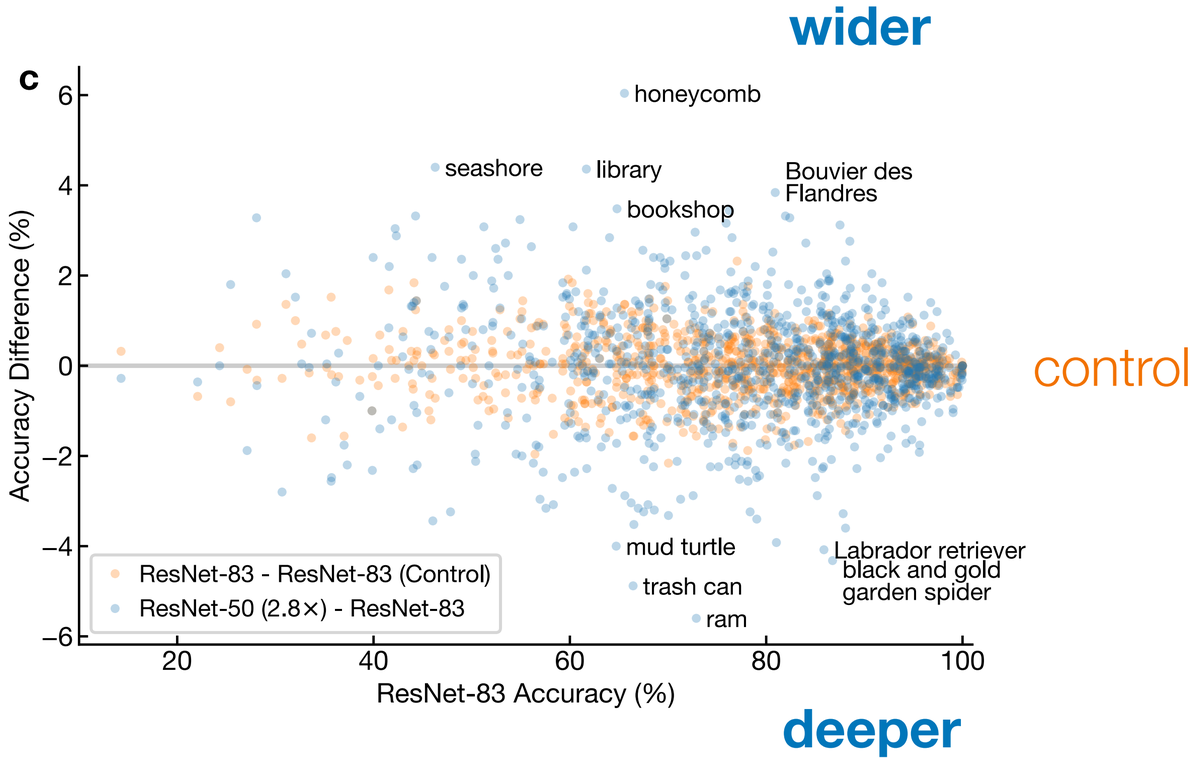


(a) Efficient GPU model found by ProxylessNAS.

(b) Efficient CPU model found by ProxylessNAS.

(c) Efficient mobile model found by ProxylessNAS.

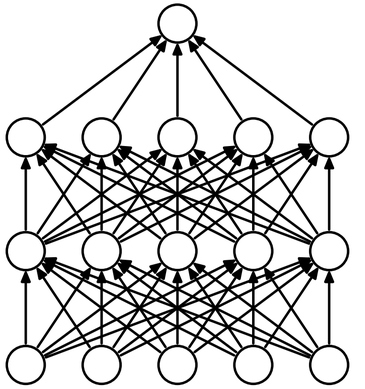
H. Cai, L. Zhu, S. Han.
ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. ICLR 2019.



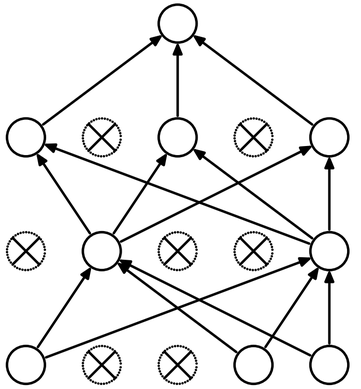
T. Nguyen, M. Raghu, S. Kornblith. Do Wide and Deep Networks Learn the Same Things? Uncovering How Neural Network Representations Vary with Width and Depth. ICLR 2021.

FFNNs: Efficiency considerations

- Typical regularization for neural networks: **Dropout** (Srivastava et al. 2014).

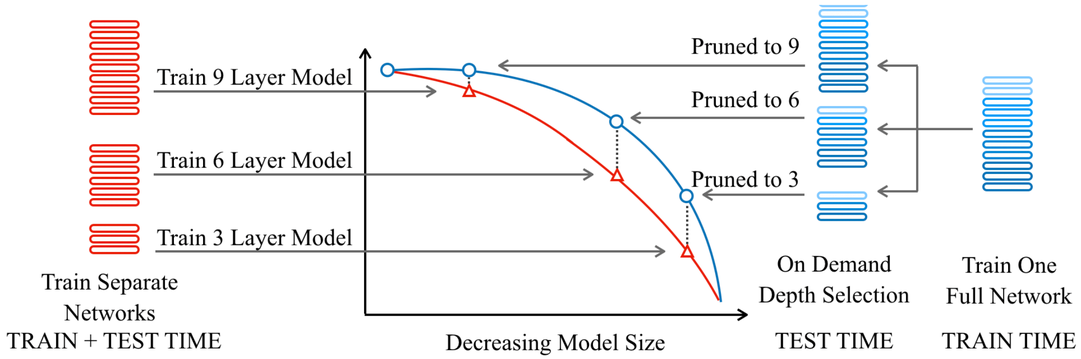


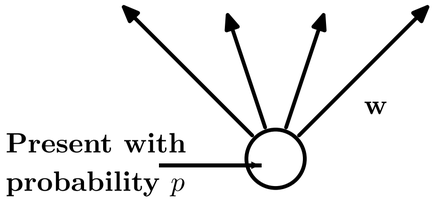
(a) Standard Neural Net



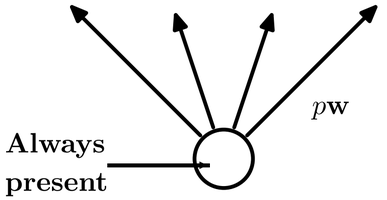
(b) After applying dropout.

- **LayerDrop** (Fan et al. 2019): Remove entire layers during training, use subsets of layers for inference at test time.





(a) At training time

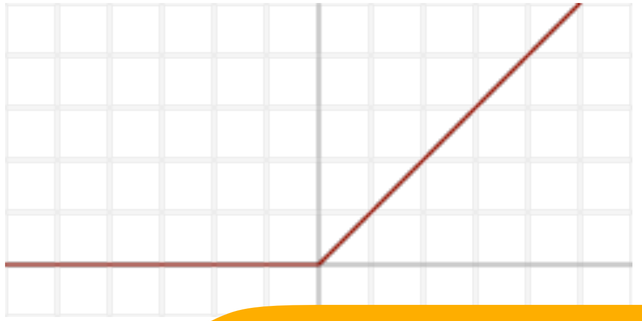


(b) At test time

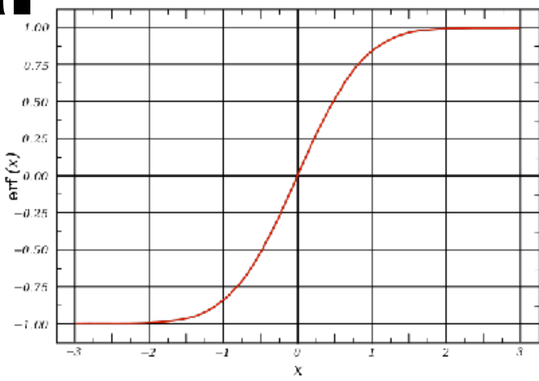
- **Nested Dropout** (Rippel et al. 2014): Impose an *ordering* on hidden units, drop units with higher probability as a function of their rank in the ordering — enables efficient nearest neighbor search using prefixes of embeddings.

Non-Linearities are not created equal

$$f(x) = max(0,x)$$



$$f(x) = \frac{e^x}{e^x + 1}$$



PAUSE!

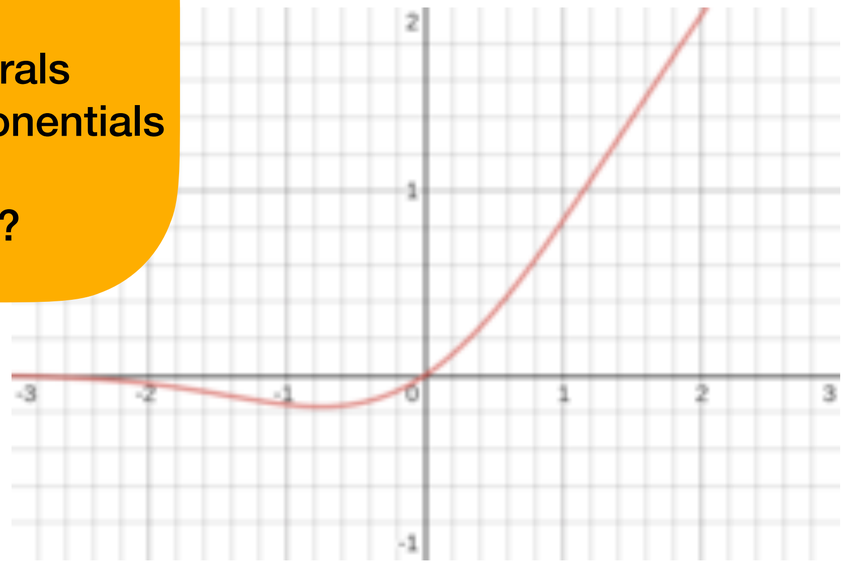
We aren't going to do integrals
We aren't even going to do exponentials

What are we going to do?

$$f(x) = \frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

$$\approx 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

$$\approx x\sigma(1.702x)$$



Tanh:

```
if ((ix|((lx|(-lx))>>31))>0x3ff00000) /* |x|>1 */
    return (x-x)/(x-x);
if(ix==0x3ff00000)
    return x/zero;
if(ix<0x3e300000&&(huge+x)>zero) return x; /* x<2**-28 */
SET_HIGH_WORD(x,ix);
if(ix<0x3fe00000) {   /* x < 0.5 */
    t = x+x;
    t = 0.5*__log1p(t+t*x/(one-x));
} else
    t = 0.5*__log1p((x+x)/(one-x));
if(hx>=0) return t; else return -t;
```

exp:

```
t  = x*x;
c  = x - t*(P1+t*(P2+t*(P3+t*(P4+t*P5))));
if(k==0)  return one-((x*c)/(c-2.0)-x);
else    y = one-((lo-(x*c)/(2.0-c))-hi);
if(k >= -1021) {
    u_int32_t hy;
    GET_HIGH_WORD(hy, y);
    SET_HIGH_WORD(y, hy+(k<<20));
    return y;
```

Log1p:

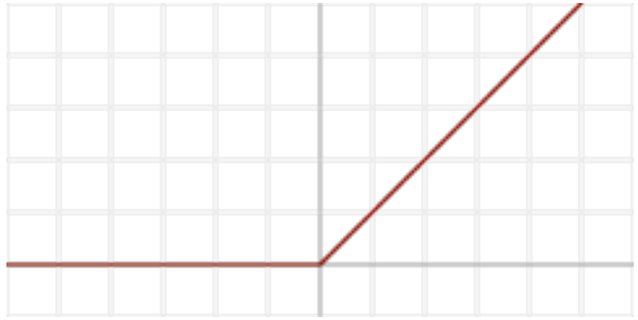
```
hfsq=0.5*f*f;
s = f/(2.0+f);
z = s*s;
R = z*(Lp1+z*(Lp2+z*(Lp3+z*(Lp4+z*(Lp5+z*(Lp6+z*Lp7))))));
if(k==0) return f-(hfsq-s*(hfsq+R)); else
   return k*ln2_hi-((hfsq-(s*(hfsq+R)+(k*ln2_lo+c)))-f);
```


https://elixir.bootlin.com/glibc/glibc-1.90/source/sysdeps/libm-ieee754/

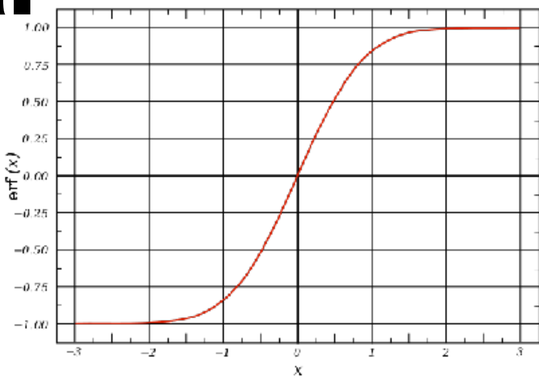
Non-Linearities are not created equal

$$f(x) = max(0,x)$$

7 lines of assembly code



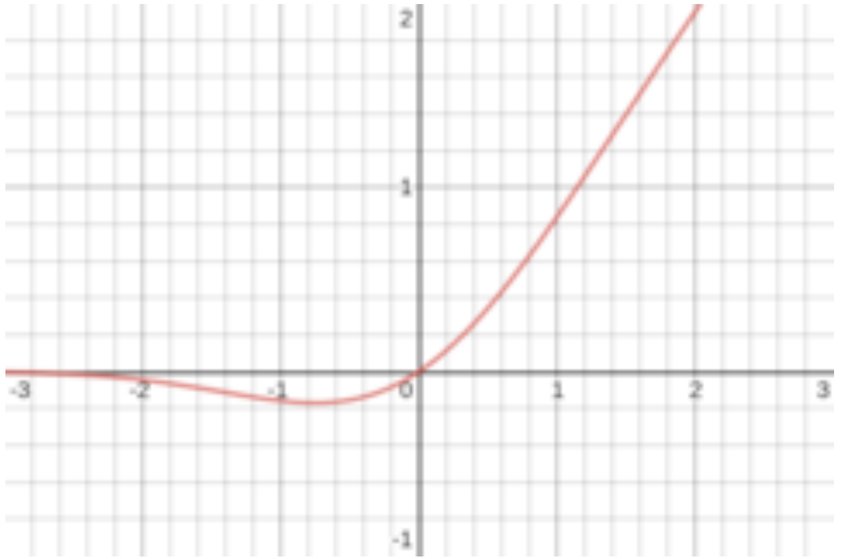
$$f(x) = \frac{e^x}{e^x + 1}$$



$$f(x) = \frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

$$\approx 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

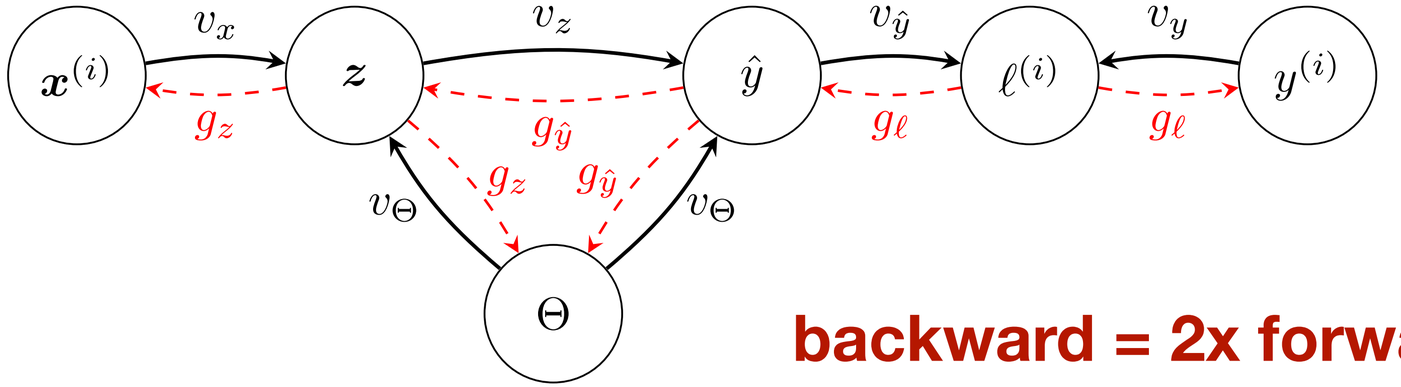
$$\approx x\sigma(1.702x)$$



```
OpenAI's gelu implementation, also used in Megatron:
   Gelu(x) = x * 0.5 * (1.0 + torch.tanh(0.79788456 * x * (1.0 + 0.044715 * x * x)))
```

Training Feed Forward Neural Networks

- **Backpropagation** as an algorithm: construct a directed acyclic computation graph with nodes for inputs, outputs, hidden layers, parameters.
  - **Forward pass**: values (v_x) go from parents to children.
  - **Backward pass**: gradients (g_z) go from children to parents — apply chain rule
  - As long as the gradient is implemented for a layer/operation, you can add it to the graph, and let **automatic differentiation** compute updates for every layer



backward = 2x forward

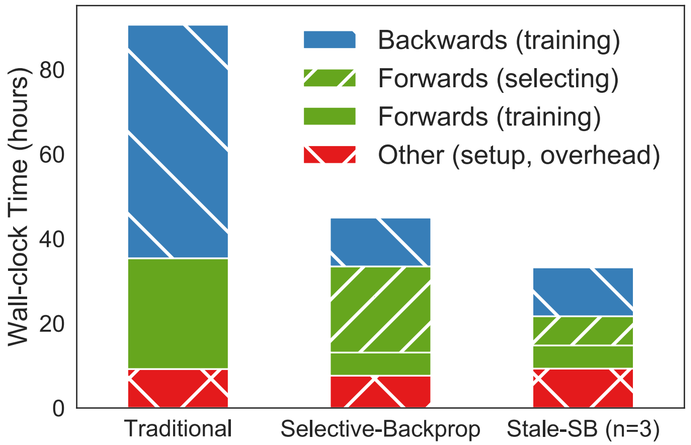


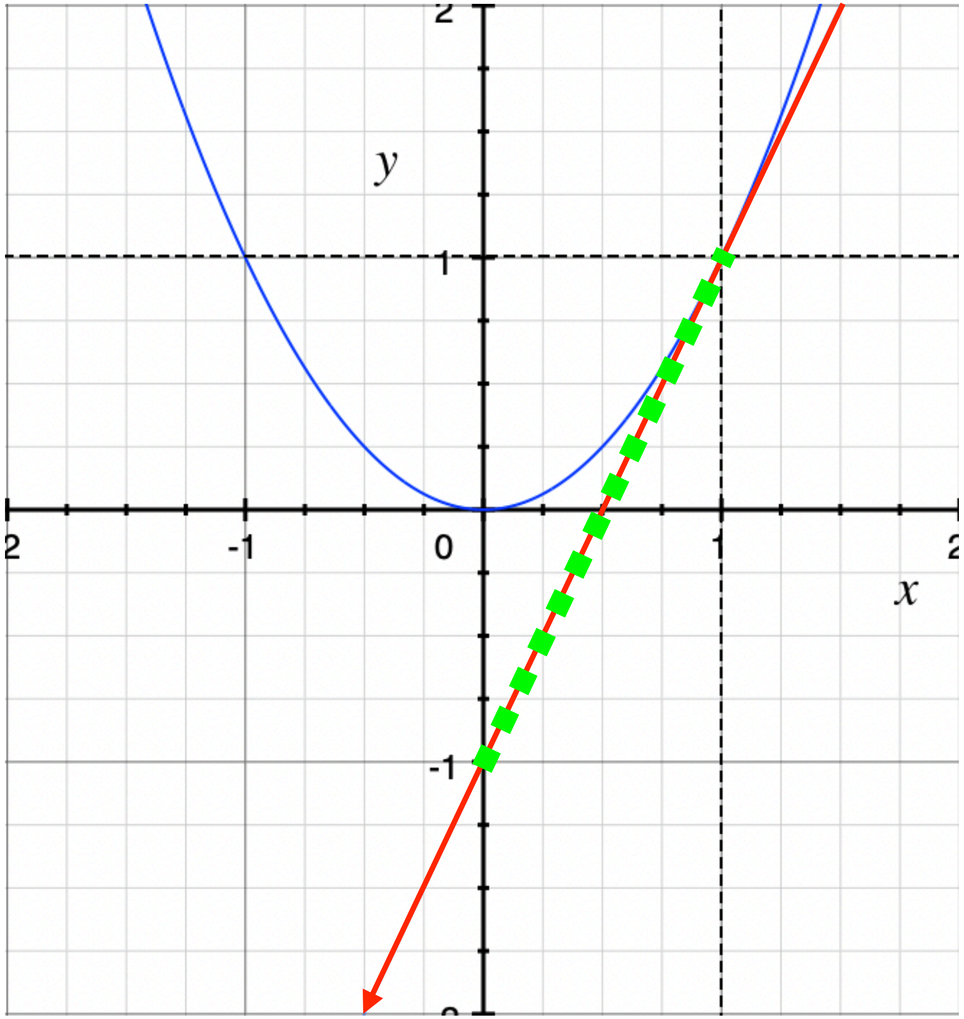
Figure from: Jiang, et al. Accelerating Deep Learning by Focusing on the Biggest Losers. 2019.

Newton's Method — Ideal Optimization

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

$$f(x) = x^2$$
$$f'(x) = 2x$$
$$f''(x) = 2$$

$$\Rightarrow \quad 1 - \frac{2}{2} = 0$$



Newton's Method — Ideal Optimization

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

Direction to move

How much to move

$$f(\vec{x}) = \mathbb{R}^n \rightarrow \mathbb{R}$$
$$f'(\vec{x}) = \mathbb{R}^n \rightarrow \mathbb{R}^n$$
$$f''(\vec{x}) = \mathbb{R}^n \rightarrow \mathbb{R}^{n*n}$$

Can we handle a quadratic?

Is computing and storing a large matrix worth it for fewer updates?

What is Adam doing? 🤔

Stepsize

$$f''(\vec{x}) = \mathbb{R}^n \rightarrow \mathbb{R}^{n*n}$$

Hessian, Invert, Multiply

Quasi-Newton (L-BFGS)

Store k vectors to approximate inverse

Adam? Momentum? …

A scalar for quantifying gradient change

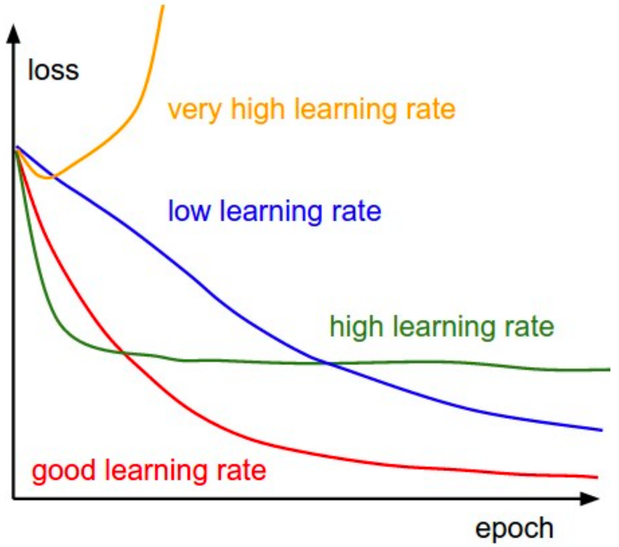
Vanilla SGD

YOLO!

Img: https://www.thesymbolism.com/dreams/symbols/driving-off-a-cliff/

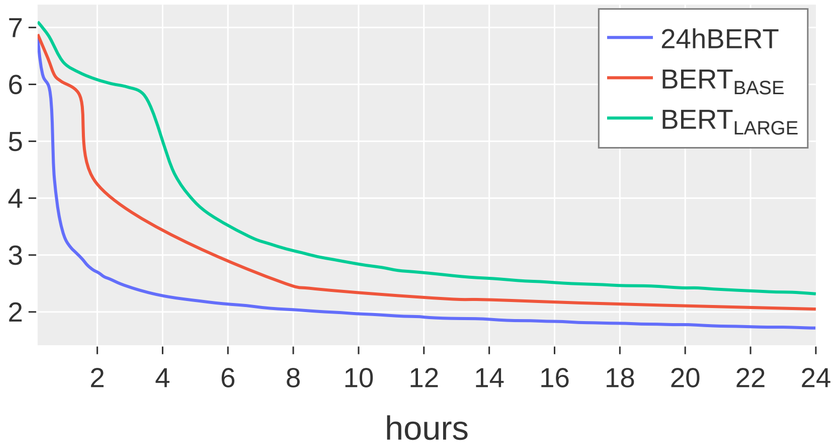
Hyperparameters & training speed

Figure: https://cs231n.github.io/neural-networks-3/



- Large learning rate + large batch size = fast convergence.
- But not necessarily good generalization.
- **24h BERT** (Izsak et al. 2021): Synchronize hyperparameters (batch size, learning rate schedule) with target training time.

	bsz	steps	samples	days
Google $\text{BERT}_{\text{BASE}}$	256	1000k	256M	5.85
Google $\text{BERT}_{\text{LARGE}}$	128†	2000k	256M	26.33
Our $\text{BERT}_{\text{LARGE}}$	128	2000k	256M	14.11
	256	1000k	256M	8.34
	4096	63k	256M	2.74
	8192	31k	256M	2.53
	16384	16k	256M	2.41



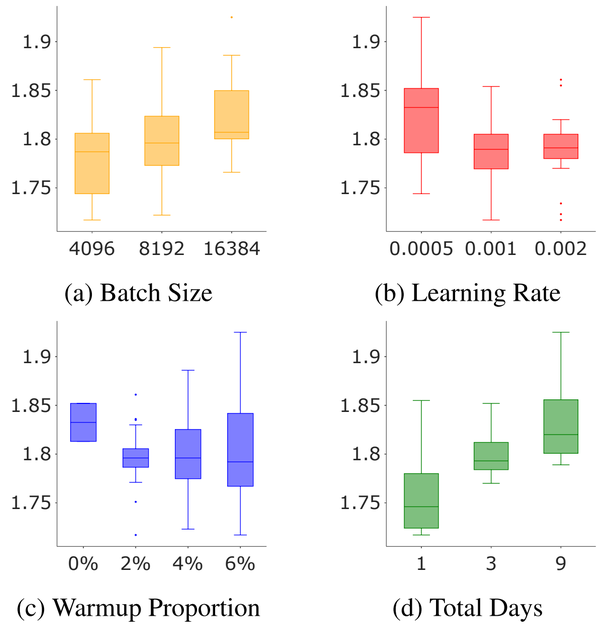
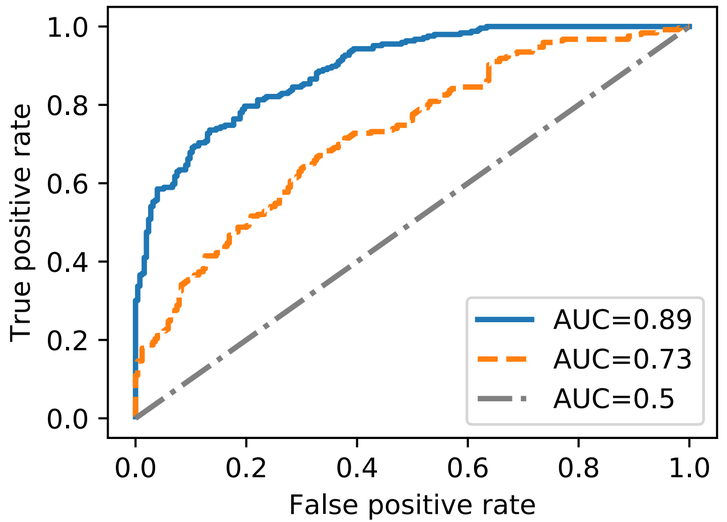


Figure 1: Distribution of the validation-set loss after 24 hours of training across different hyperparameters.

Evaluation

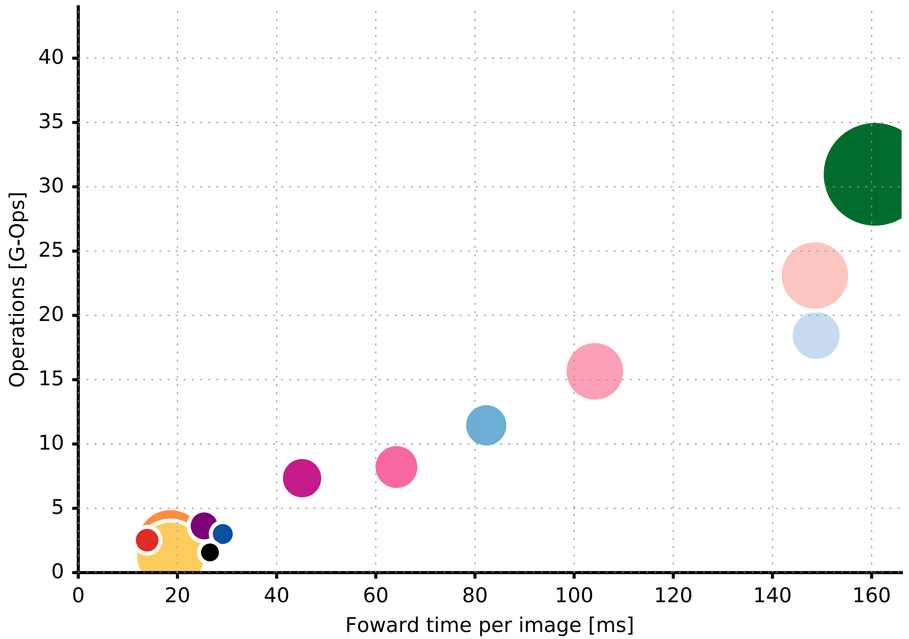
- **No free lunch**: when making models more efficient, you may maintain top-1 accuracy but *at what cost?*
  - **Precision vs. recall:**
    - For a medical diagnosis, prefer high recall. False positives screened out later.
    - Beyond reasonable doubt" standard of U.S. criminal law implies a preference for high precision.
- **Generalization & bias:** Is your model sacrificing accuracy on less frequent classes/phenomena in order to maintain accuracy on more common cases?



Evaluating efficiency

- Most common methods for evaluating efficiency:
  - Floating Point OPs (**FLOPs**): Number of additions and multiplications.
  - **Latency** / wall-clock time: Number of {milli,micro,nano}seconds.
  - **Parameter count**.
- What about hardware, memory locality, training vs. fine-tuning amortization the ML model lifecycle?

  ➡ More discussion later this semester.



A. Canziani, A. Paszke, E. Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. 2017.

Isn’t everything just math and GLIBC anyway?

- When did you create your control flow and compute graph?
- Are your graph operations compiled to a local hardware binary or intermediate representation?
- Are you bookkeeping for gradients or optimizers?
- Are you doing precise or fuzzy operations?