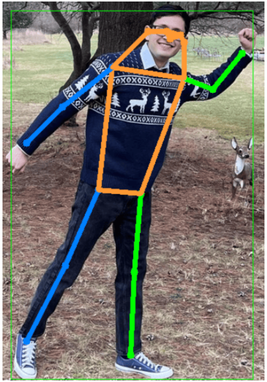
Deep Learning 4 Dummies 1

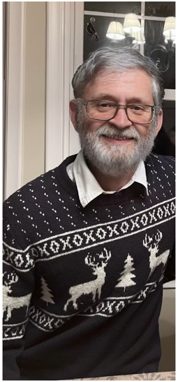
(by a dummy for other dummies)



Javier Salazar Cavazos

University of Michigan

November 1, 2023

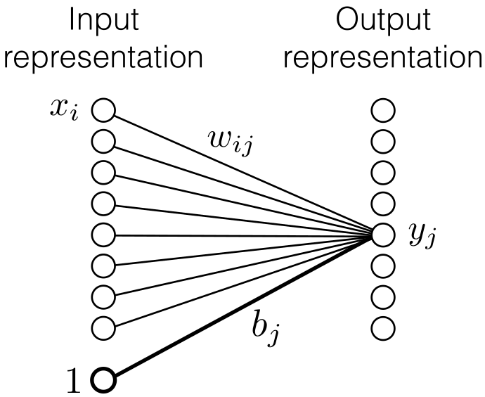


Preface

- I have some DL experience but pretty far from an expert!
- Some of the material is “stolen” from the great work of others below
- Some great resources used here are Wikipedia, [1], [2], [3]
  [1] being an exceptional resource
- Will go over: basics, optimization, & issues related to DL
- Recommended prereqs: EECS551 & EECS559

Basics: linear regression

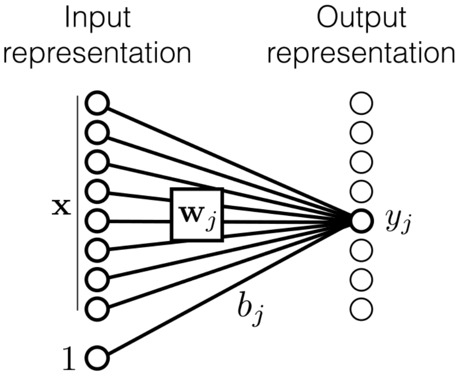
Linear layer



$$y_j = \sum_i \overset{\textbf{weights}}{w_{ij}} x_i + \underset{\textbf{bias}}{b_j}$$

Basics: linear regression 2

Linear layer



Full layer

weights + bias

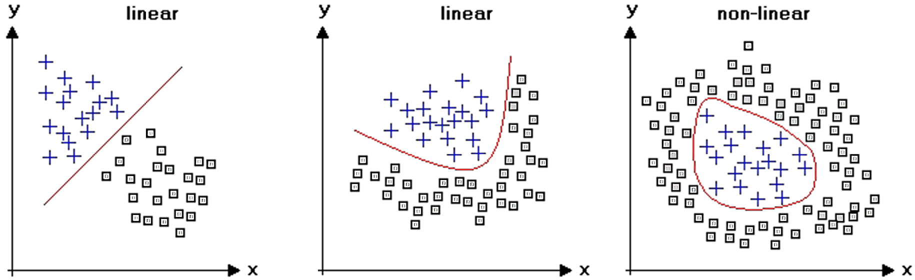
$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

Can again simplify notation by appending a 1 to $\mathbf{x}$

Question: Would adding more layers help to increase accuracy?

Basics: linear model disadvantages

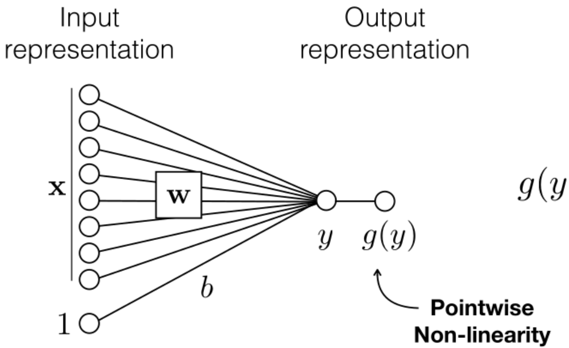
Classification here instead of regression but WLOG...



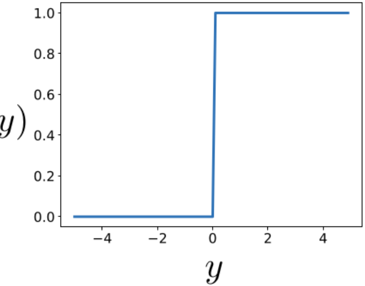
- Linearity means linear with respect to the coefficients!
- Linear models might not work for some problems/datasets!
- Hence, the model must be generalized!

Basics: activation functions

Activation functions (nonlinear) → extract nonlinear features of the data



$$g(y) = \begin{cases} 1, & \text{if} \quad y > 0 \\ 0, & \text{otherwise} \end{cases}$$



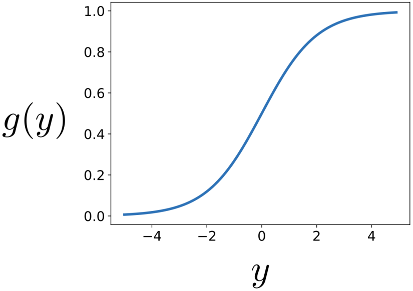
Question: Why is this not a good nonlinear function?

Basics: activation functions 2

- Interpreted as “firing rate of neuron”
- Bounded between $(0, 1)$
- Large inputs become saturated
- $\sigma(y)$ centered at 0.5 so $\tanh(y) = 2g(y) - 1$ more practical

Sigmoid

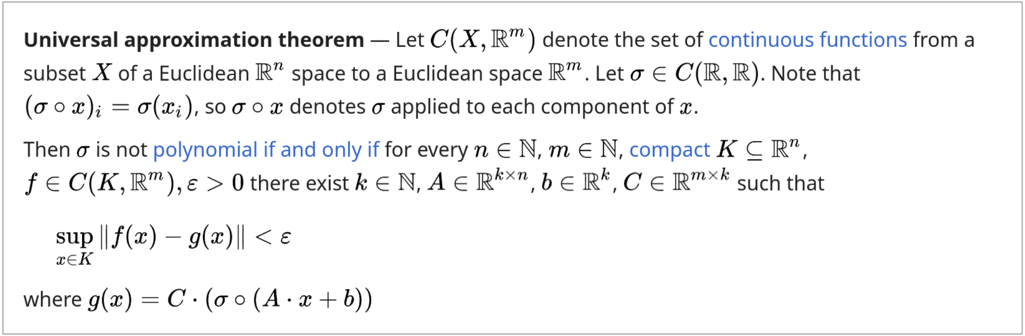
$$g(y) = \sigma(y) = \frac{1}{1 + e^{-y}}$$



Question: Would adding more layers with some $g(\cdot)$ help?

Basics: activation functions 3

But why does having activation functions help with accuracy?



Universal approximation theorem — Let $C(X, \mathbb{R}^m)$ denote the set of continuous functions from a subset X of a Euclidean $\mathbb{R}^n$ space to a Euclidean space $\mathbb{R}^m$. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Note that $(\sigma \circ x)_i = \sigma(x_i)$, so $\sigma \circ x$ denotes σ applied to each component of x.

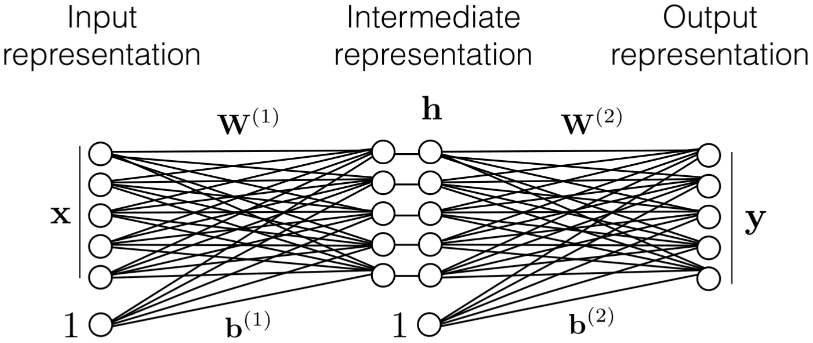
Then σ is not polynomial if and only if for every $n \in \mathbb{N}$, $m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n$, $f \in C(K, \mathbb{R}^m), \varepsilon > 0$ there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \| f(x) - g(x) \| < \varepsilon$$

where $g(x) = C \cdot (\sigma \circ (A \cdot x + b))$

We can approximate any compactly supported continuous function!

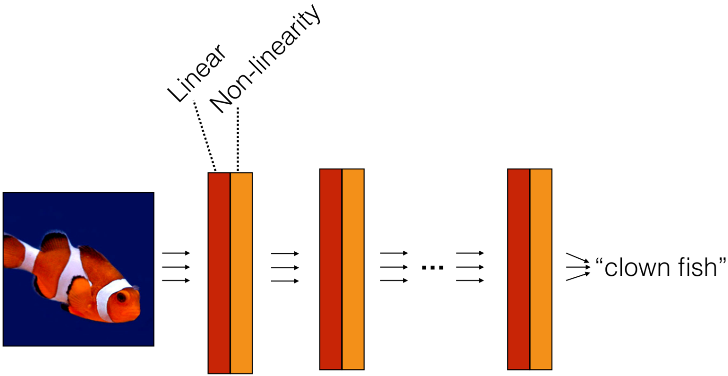
Basics: stacked layers



$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

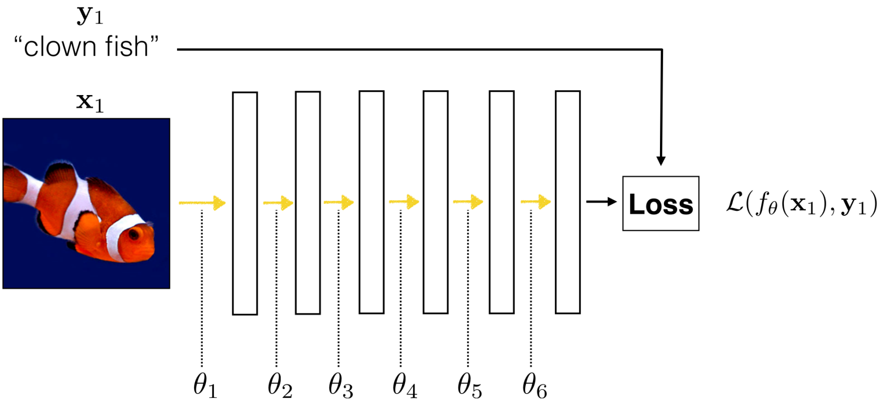
$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

Basics: entire model



$$f(\mathbf{x}) = f_L(\ldots f_2(f_1(\mathbf{x})))$$

Basics: entire model 2



$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Optimization: a simple example

Let's optimize for squared loss with single-variable networks:

$$L = \frac{1}{2}(y - f(x))^2 = \frac{1}{2}(y - \sigma(wx + b))^2$$

Let's write out the layers:

$z = wx + b$ affine/linear layer

$t = \sigma(z)$ nonlinear activation

$L = \frac{1}{2}(y - t)^2$ loss function

$$\nabla_w L = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial t}\frac{\partial t}{\partial z}\frac{\partial z}{\partial w} = \textcircled{1}\ \textcircled{2}\ \textcircled{3}$$

$$\nabla_b L = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial t}\frac{\partial t}{\partial z}\frac{\partial z}{\partial b} = \textcircled{1}\ \textcircled{2}\ \textcircled{4}$$

Partial derivatives:

$\textcircled{1} = t - y$

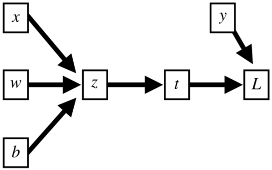
$\textcircled{2} = \sigma'(z)$

$\textcircled{3} = x$

$\textcircled{4} = 1$

Optimization: forward propagation

Compute values in each node in computational graph



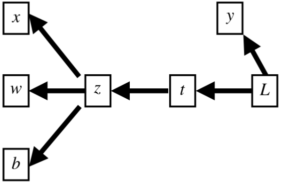
Forward pass:

```
for i = 0 … N:
    compute each node value $v_i$ using
    values from previous nodes
```

given ordered nodes $\nu_0, \ldots, \nu_N$ (meaning inputs precede outputs)

Optimization: backpropagation

Given calculated $\nu_0, \ldots, \nu_N$ compute derivatives



Backward pass:

```
$v_N = 1$
for i = N-1 .. 0:
    $\frac{\partial L}{\partial v_i} = \sum_{j \in \text{Outputs}(v_i)} \frac{\partial L}{\partial v_j} \frac{\partial v_j}{\partial v_i}$
```

starting from loss working your way to inputs

Optimization: putting it together

In practice, you have many (x, y) pairs due to Big Data™

$$J = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(x_i, y_i)$$

so you update the hyperparameters like this (gradient descent):

$$W_l^{k+1} = W_l^k - \eta \frac{\partial J}{\partial W_l}$$

by accounting for all of the data and averaging:

$$\frac{\partial J}{\partial W_l} = \frac{1}{N} \sum_{i=1}^{N} \left. \frac{\partial \mathcal{L}}{\partial W_l} \right|_{x_i, y_i}$$

which requires forward propagation and back propagation for each term!

Optimization: the gradient descent family

Gradient descent (all data):

$$\frac{\partial J}{\partial W} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial W}\bigg|_{x_i,y_i}$$

Stochastic GD (one data pair):

$$\frac{\partial J}{\partial W} \approx \frac{\partial \mathcal{L}}{\partial W}\bigg|_{x_i,y_i} \quad i \in \{1,\dots,N\}$$

Mini-Batch SGD (the compromise):

$$\frac{\partial J}{\partial W} \approx \frac{1}{|B|}\sum_{b\in B} \frac{\partial \mathcal{L}}{\partial W}\bigg|_{x_b,y_b} \quad B \subset \{1,\dots,N\}$$

Optimization: momentum

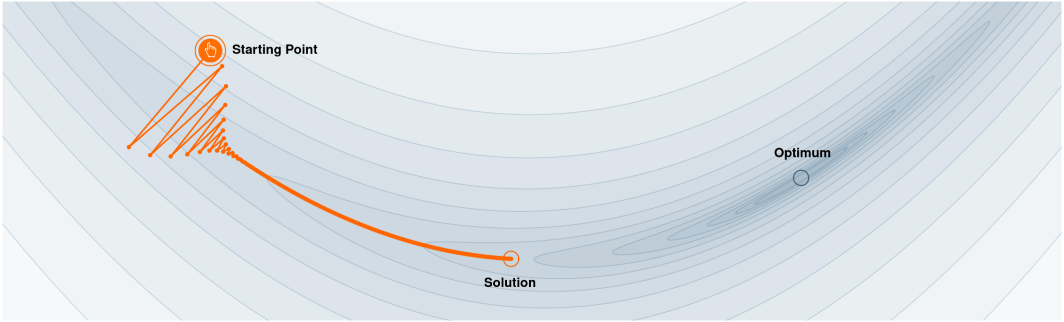
$$m^{(k+1)} = \beta m^{(k)} + \frac{\partial L}{\partial W}$$

$$W^{(k+1)} = W^{(k)} - \eta m^{(k+1)}$$

- It remembers gradients and “builds up speed”
- Reduces oscillations in high curvature regions (variance reduction)
- Becomes faster in low curvature regions (acceleration)
- Condition: $0 \leq \beta < 1$

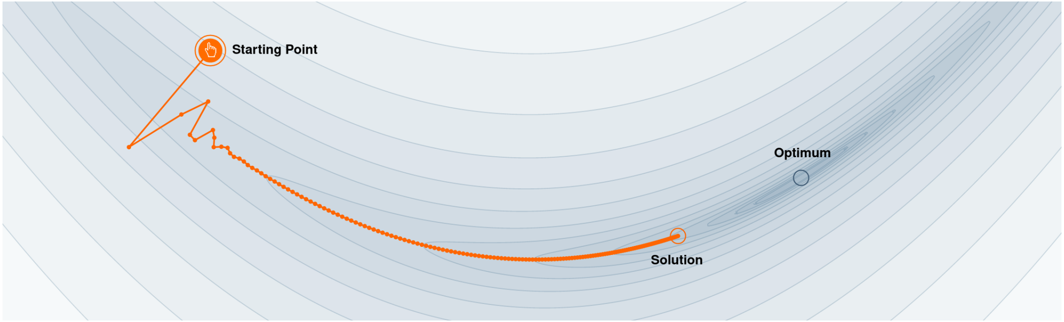
Optimization: momentum visualized 1

No momentum:



Optimization: momentum visualized 2

With momentum ($\beta = 0.5$):



Optimization: the real methods

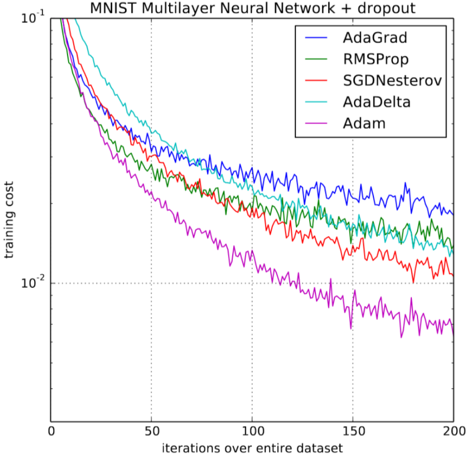
These methods are “basic”!
Use either SGDNesterov or Adam.

SGDNesterov:

- SGD + Nesterov momentum

Adam:

- Most common method used!
- Momentum + second moments②
- ② meaning variance of gradients
- ② used for adaptive learning rates



Issues: learning rate

How do you chose the step size η?

- Optimal initial rate η_0 is close to largest rate that does not cause divergence (by a factor of 1/2)
- Given normalized inputs, rule of thumb is $10^{-6} < \eta_0 < 1$
- Decrease step size by a factor of 10 once validation error stagnates or every fixed X epochs
- Or, use adaptive rate such as $\eta_k = \dfrac{\eta_0 \tau}{\text{max}(k, \tau)}$

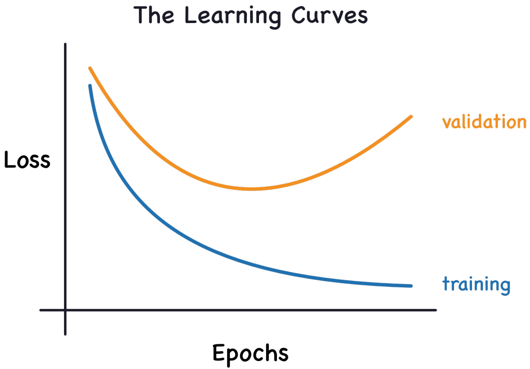
Issues: batch size

How to pick B? (heavily constrained by GPU memory)

- Usually done in CPU thread multiples $\{1, 2, 4, 8, 16, 32, \ldots\}$
- As $|B|$ increases, more FLOPS due to parallelism
- Too large is bad! Less updates per same computation time...
- There is no point to computing the gradient exactly anyway...
- Usually, $|B| = 32$ is a good default to start with

Issues: training iterations

How long to train?



Essentially the same as asking, how do you prevent overfitting?

Issues: overfitting

Many different techniques for this!

Stopping the network early based on validation accuracy is commonly used

Another option is to reduce the size of the network to reduce complexity

Two other common techniques: weight decay and dropout

Issues: overfitting - weight decay

Linear regression model: $f(x) = w_0 + w_1 x + w_2 x^2 + \dots$

$\|w\|_1$ promotes sparsity $\implies$ there are some $w_i = 0$

This is reducing the complexity of the model!

Weight Decay:

$$R(W_l) = \|W_l\|_{1,1} \text{ or } R(W_l) = \|W_l\|_F^2$$

$$\min_{\theta} \mathcal{L}(f_\theta(x), y) + \sum_l R(W_l)$$

Issues: overfitting - dropout

Dropout:
Randomly drop nodes in the hidden/input layers

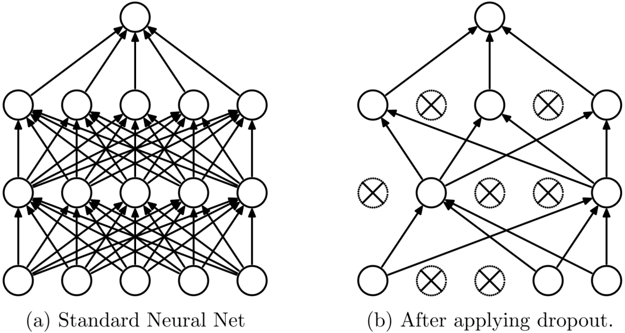
Let $\boldsymbol{y}$ be output of last layer

Do $\tilde{\boldsymbol{y}} = \boldsymbol{y} \odot \boldsymbol{r}$ and continue

$\boldsymbol{r} \sim$ Bernoulli(p)

input only: $p \in (0.8, 1.0)$
hidden only: $p \in (0.5, 0.8)$

careful!
$\tilde{\alpha_0} \in (10\alpha_0, 100\alpha_0)$
$\beta \in (0.95, 0.999)$


(a) Standard Neural Net

(b) After applying dropout.

Issues: activation functions

Which $g(\cdot)$?

Sigmoid/tanh common early on

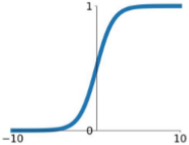
ReLU achieved 6x faster convergence than $\sigma(x)$

Default choice in many networks!

If not, usually a variation of ReLU like GeLU (e.g. GPT)

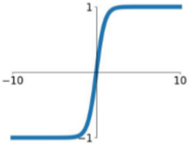
Sigmoid

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



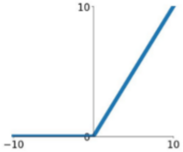
tanh

$\tanh(x)$



ReLU

$\max(0, x)$

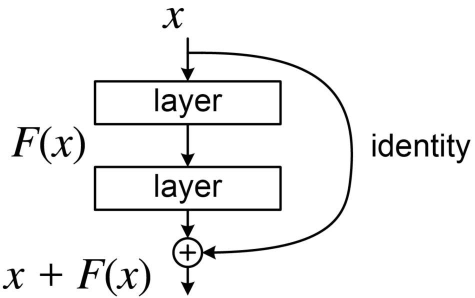


Issues: vanishing/exploding gradients 1

NNs used to be fairly shallow (deep networks $\rightarrow \|\nabla_\theta \mathcal{L}_\theta(x, y)\| \approx 0$)
Two different ideas help with this: skip connections & normalization

Skip connections:

- Because of identity mapping, gradients can propagate faster
- Only have to learn small residuals in each block
- Will "smoothen" the landscape

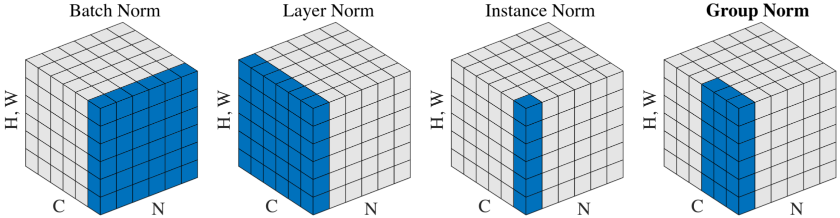


Issues: vanishing/exploding gradients 2

Normalization:

Let $x_i = f_{\theta_l}(\text{image}_i)$ (feature map of ith image)

$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i)$, $(\sigma_i, \mu_i$ depend on method)



Question: BN does not work well with small batches, why?

Issues: limited data

Imbalanced/limited data?

Transfer learning:

Earlier features tend to be more generic than latter half

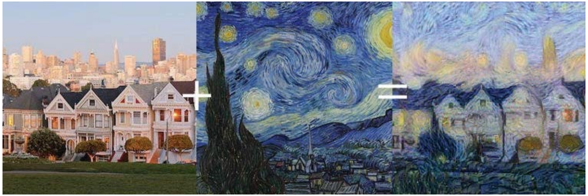
Traditional augmentation:

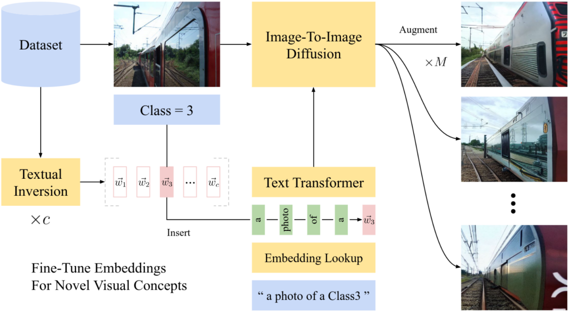
Perform operations such as cropping, rotation, noise, etc...

Modern augmentation:

Neural style transfer

GANs/Diffusion models





Dataset

Class = 3

Image-To-Image Diffusion

Augment

$\times M$

Textual Inversion

$\times c$

$\vec{w}_1$ $\vec{w}_2$ $\vec{w}_3$ $\cdots$ $\vec{w}_c$

Insert

Text Transformer

a photo of a $\vec{w}_3$

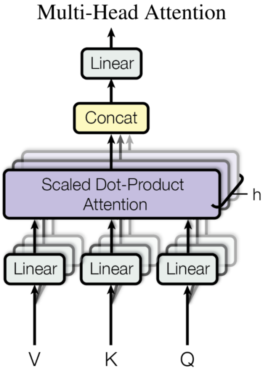
Embedding Lookup

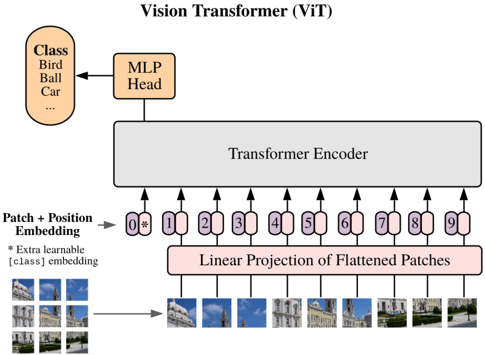
" a photo of a Class3 "

Fine-Tune Embeddings For Novel Visual Concepts

Future work: DL4D2

More interesting presentation next semester!




Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0
*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

History & intuition of various network architectures...

References

[1] A. Torralba, P. Isola, and W. Freeman, *Foundations of Computer Vision*. MIT Press, 2024.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[3] R. Szeliski, *Computer vision: algorithms and applications*. Springer Nature, 2022.

[4] Y. Bengio, *Practical recommendations for gradient-based training of deep architectures*, 2012. arXiv: `1206.5533 [cs.LG]`.

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

References

[6] Y. Wu and K. He, *Group normalization*, 2018. arXiv: `1803.08494 [cs.CV]`.

[7] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: `1512.03385 [cs.CV]`.

[8] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: `1412.6980 [cs.LG]`.

[9] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[10] G. Goh, "Why momentum really works," *Distill*, vol. 2, no. 4, e6, 2017.

[11] B. Trabucco, K. Doherty, M. Gurinas, and R. Salakhutdinov, *Effective data augmentation with diffusion models*, 2023. arXiv: `2302.07944 [cs.CV]`.

References

[12] L. A. Gatys, A. S. Ecker, and M. Bethge, *A neural algorithm of artistic style*, 2015. arXiv: `1508.06576 [cs.CV]`.