Adaptive Gradient Methods

Rather than using the same learning rate for all params, use a different learning rate for each computed "adaptively" based on past gradients.

Simplest: Momentum

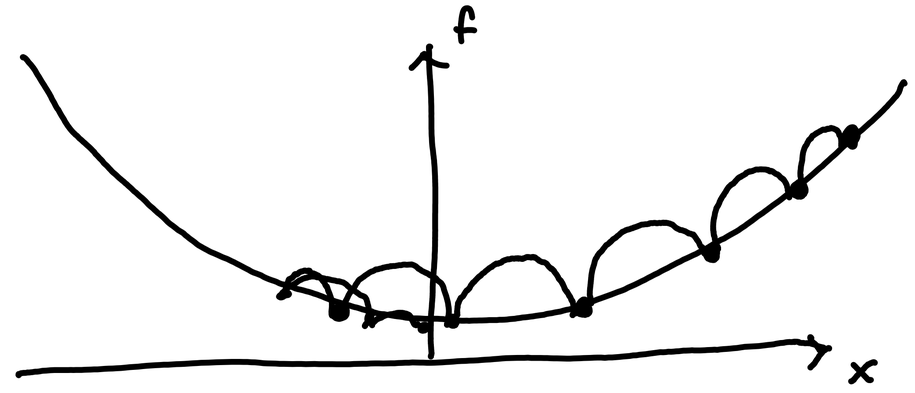
$$v_t \in \mathbb{R}^d \leftarrow \beta v_{t-1} + g_t \qquad x_t \leftarrow x_{t-1} - \eta v_t$$

($v_t \in \mathbb{R}^d$; β: new hparam; g_t: gradient vector at current timestep $\nabla_{x_{t-1}} f(x) \in \mathbb{R}^d$; η: learning rate)

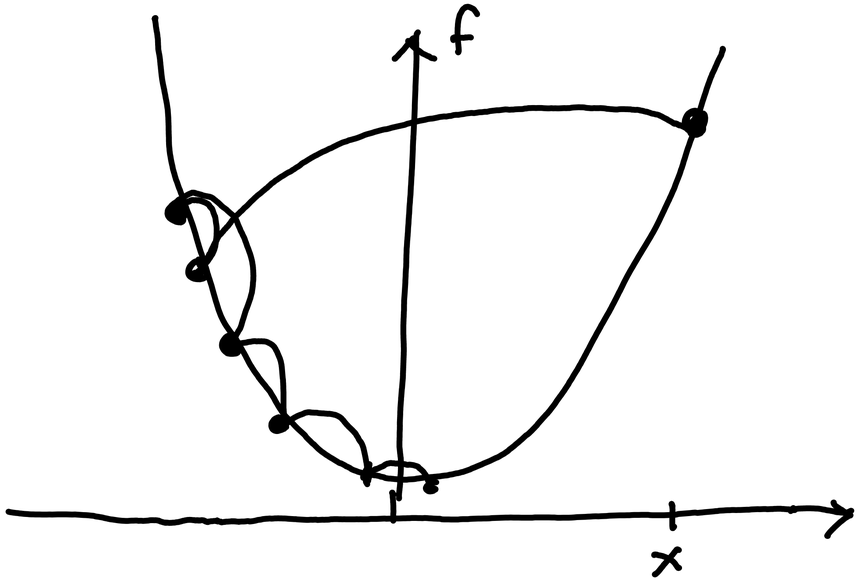
v_t will change move slowly than g_t. If g_t is consistent, v_t will grow in that direction. If g_t is inconsistent, it might "cancel out" changes.

$v_t \leftarrow \beta v_{t-1} + g_t$

(assume β is close to 1, 0.9 is typical)



gradient points in the same direction → take bigger steps



gradient direction is inconsistent → take smaller steps

Adam:

$$v_t \leftarrow \beta_1 v_{t-1} + (1-\beta_1) g_t \quad \text{("first moment" estimate)}$$

$$s_t \leftarrow \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \quad \text{("second moment" estimate)}$$

$$g_t' = \frac{\eta v_t}{\sqrt{s_t} + \varepsilon} \rightarrow \text{small constant}$$

$$x_t \leftarrow x_{t-1} - g_t$$

$$v_1 = \beta_1 v_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1 \quad (v_0 = 0)$$

$$v_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2$$

$$\beta_1 (1-\beta_1) + (1-\beta_1) = 1 - \beta_1^2 \quad \text{(total gradient rescaling)}$$

$$v_3 = \beta_1 (\beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2) + (1-\beta_1) g_3$$

$$\rightarrow 1 - \beta_1^3$$

Total gradient scale is $1 - \beta_1^t$

Typically, β_1 and β_2 are close to 1.

$1-\beta_1^t$ and $1-\beta_2^t$ will start near 0 and grow slowly. So v_t and s_t will "biased" towards zero.

To correct this bias: Divide out $1-\beta_1^t$ and $1-\beta_2^t$

$$v_t \leftarrow \beta_1 v_{t-1} + (1-\beta_1) g_t \qquad \hat{v}_t = \frac{v_t}{1-\beta_1^t}$$

$$s_t \leftarrow \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \qquad \hat{s}_t = \frac{s_t}{1-\beta_2^t}$$

$$g_t' = \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t} + \varepsilon}$$

$$x_t \leftarrow x_{t-1} - g_t'$$

Adam rescales gradients so that you use (approximately) the average gradient divided by the average magnitude.

Hopefully this helps us use the same learning rate for all parameters and all problems!

$\beta_1 = 0.9$

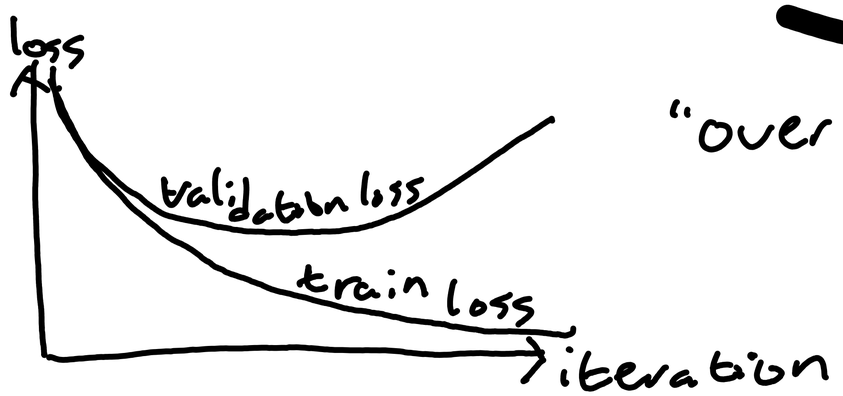
$\beta_2 = 0.999$

$\eta = 0.001$

$\epsilon = 10^{-8}$

Note: Have to store v_t and s_t somewhere...

Overfitting and regularization



"overfitting"

How can we overfit in a linear model?

Say $x_i \sim N(y, \sigma)$ $(y \in \{-1, 1\})$

We use use $\hat{y} = wx$

As dim(x) grows and size of training dataset shrinks, it becomes increasingly likely that for some feature x_i, we can achive zero loss → overfitting!

How can we prevent overreliance on a feature?

1. minimize $\|w\|^2$ by adding $\lambda \|w\|^2$ to our loss function. (λ hyperparam)

$$w - \eta \nabla_w (L + \lambda \|w\|^2) = \eta \nabla_w L + (1 - \eta 2\lambda) w$$

So we are performing "weight decay"

2. "Dropout" features at random:

During training,

$$x_i' = \begin{cases} 0 & \text{with probability } p \\ \frac{x_i}{1-p} & \text{with probability } (1-p) \end{cases}$$

$$\mathbb{E}[x_i'] = x_i$$

Multi-layer Perceptrons

"linear models" → an increase in a feature must always increase or decrease the model's output

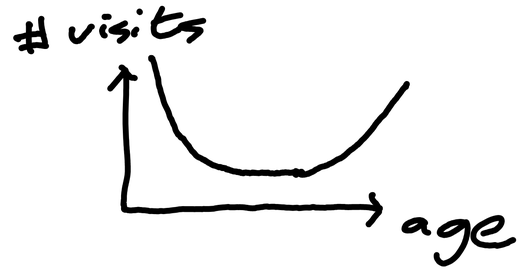
$$\hat{y} = wx + b$$

w pos: $x \uparrow$, $\uparrow \hat{y}$

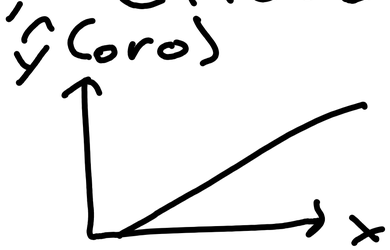
w neg: $x \uparrow$, $\downarrow \hat{y}$

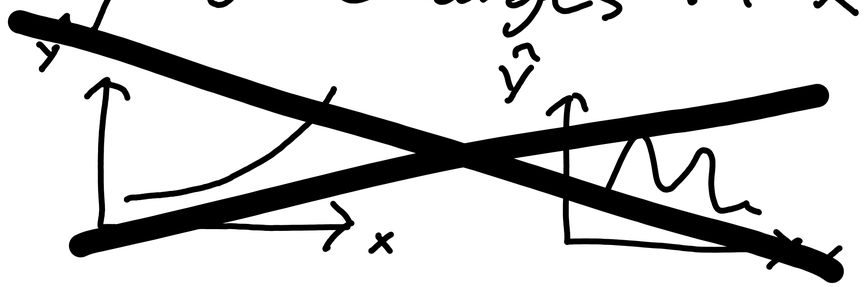
ex/ sqft ↑, value ↑

ex/ age ↑, # of doctor visits



also, effect on $\hat{y}$ of changes in x is always the same.





How can we get more complex functions?

Stack maltiple transformations.
(compose maltiple functions)

Let's try

$$H = XW_1 + b_1$$

hidden features — input

$$O = HW_2 + b_2$$

output

$$O = (XW_1 + b_1)W_2 + b_2$$

$$= XW_1W_2 + b_1W_2 + b_2$$

W — b

$$= XW + b$$

Need nonlinearity! Apply nonlinear functions.

$H = \sigma(XW_1 + b_1) \qquad O = HW_2 + b_2$

↳ nonlinear function!

One "layer" is a linear transformation and a nonlinearity.

σ is usually "elementwise"

$h_i = \sigma(x_i)$

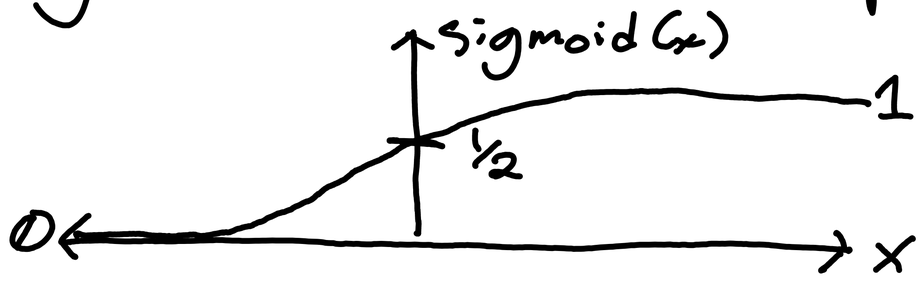
"Neural network"

"Multi-layer perceptron"

"Dense/feed-forward/fully connected"

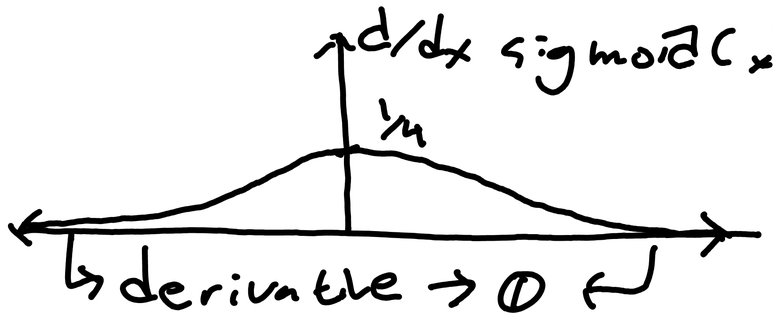
What to use for $\sigma(x)$?

$$\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$$



$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2}$$

$$= \text{sigmoid}(x)\,(1-\text{sigmoid}(x))$$



$$1-\text{sigmoid}(-x)$$

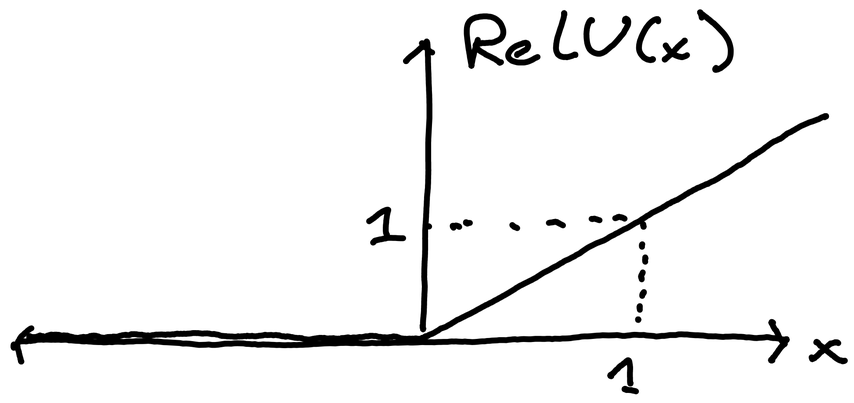
$$= 1 - \frac{1}{1+\exp(-x)}$$

$$= \frac{1+\exp(-x)}{1+\exp(-x)} - \frac{1}{1+\exp(-x)}$$

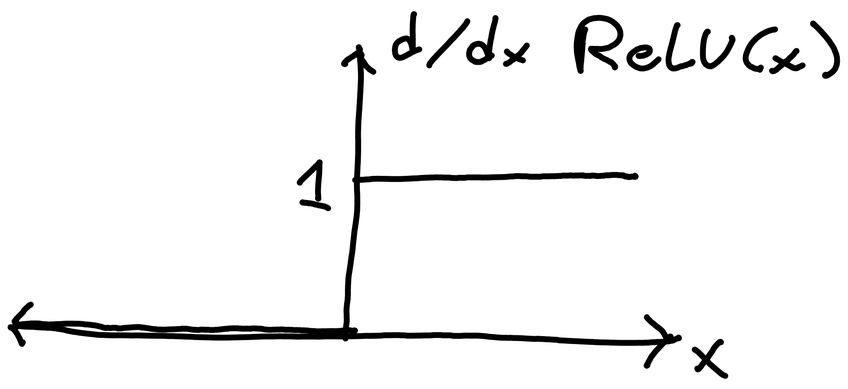
$$= \frac{\exp(-x)}{1+\exp(-x)}$$

ReLU - "Rectified Linear Unit"

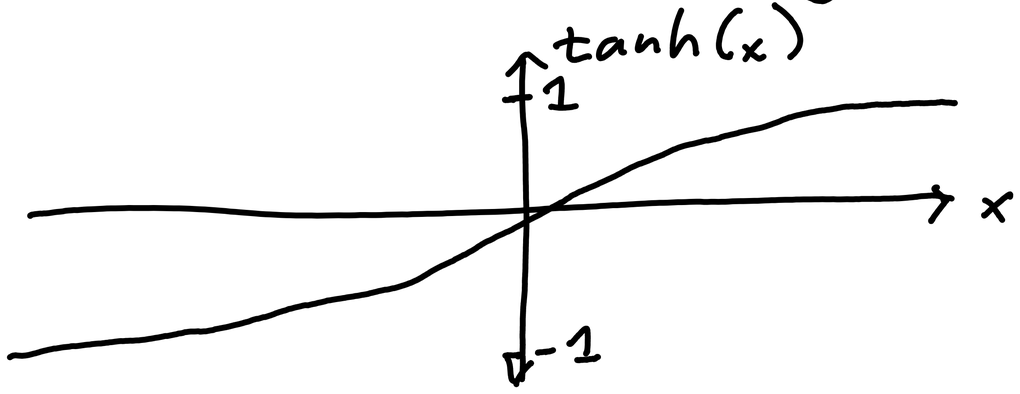
$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} = \max(0, x)$$



$$\frac{d}{dx} ReLU(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} = step(x)$$



$$\tanh(x) = 2\,\text{sigmoid}(2x) - 1$$



Universal Approximation Theorem:
A sufficiently "wide" MLP with one hidden layer can approximate any function arbitrarily well.

("width" is the dimensionality of the hidden features)

Is universal approximation useful?

1. It means we can overfit. → regularization
2. The model might be huge. → different architectures
3. It doesn't tell us how. → gradient descent (via backprop)