Neural Networks
(aka Deep Learning)

Kernels:

Remember the kernel trick? We made linear classifiers non-linear by mapping the data *implicitly* into a *fixed* (very) high dimensional feature space.

$x \rightarrow \phi(x) \qquad h(x) = \phi(x)^T v + b$

Neural Networks:

Neural Nets were originally invented by Frank Rosenblatt in 1963 (at Cornell) and take a similar approach.
However, in contrast to kernels, here the mapping is *explicit* and *learned*.

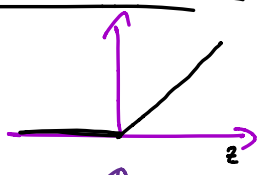
Multi-layer Perceptron (MLP): $\phi(x) = \sigma(Ux)$
Here $\sigma(z)$ is a non-linear transition function, which operates element-wise on each dimension.

Without the transition function, the classifier would just be linear:

$w^T\phi(x) + b = w^T(Ux) + b = w^TUx + b = \underbrace{\hat{w}^Tx + b}_{\text{linear!}}$ where: $\hat{w} = w^TU$

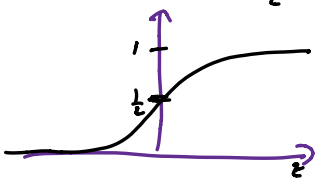
Example transition functions are:

$\sigma(z) = \max(z, 0)$



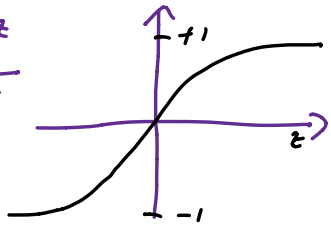
Rectified Linear Unit (ReLU)

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



Sigmoidal Unit

$\sigma(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$



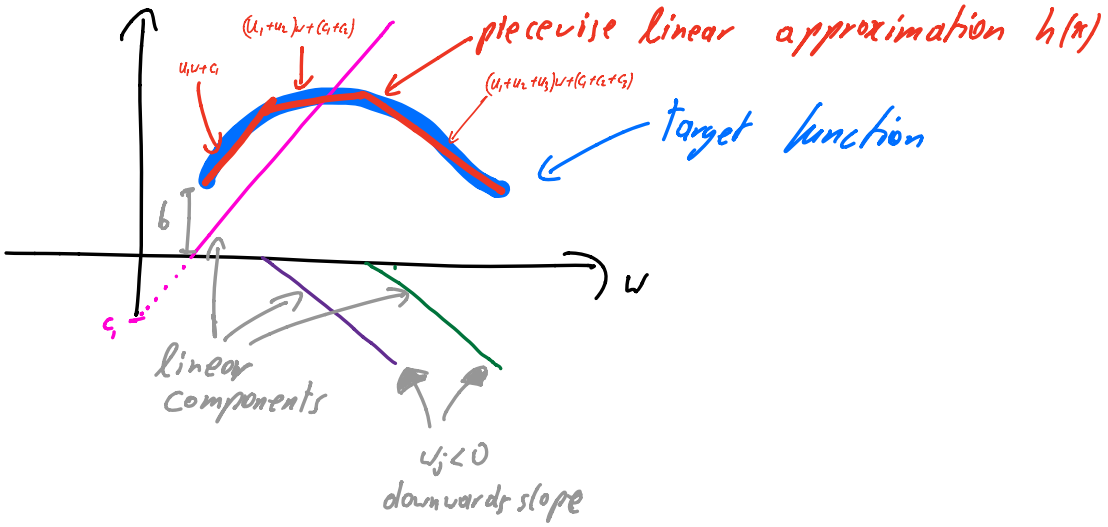
Hyperbolic Tangent

Example (ReLU): (Regression) $U = \begin{bmatrix} — u_1^T — \\ — u_2^T — \\ — u_j^T — \end{bmatrix}$

switches linear components on/off

$h(x) = w^T \max(Ux + \vec{c}) + b = \sum_j w_j \max(u_j^T w + c_j, 0) + b$

linear function; bias



Learning: We need to learn the parameters of ϕ, and we can do simple gradient descent.

Let ℓ be any well-defined loss function,

e.g. $\ell(h) = \frac{1}{2}\sum_{i=1}^{n} (h(x_i) - y_i)^2$ h has parameters: $\vec{w}, U, \vec{c}, b$

simple gradient descent:

Repeat

$$\vec{w} \leftarrow \vec{w} - \alpha \frac{\partial \ell}{\partial \vec{w}}$$
$$U \leftarrow U - \alpha \frac{\partial \ell}{\partial U}$$
$$\vec{c} \leftarrow \vec{c} - \alpha \frac{\partial \ell}{\partial \vec{c}}$$
$$b \leftarrow b - \alpha \frac{\partial \ell}{\partial b}$$

These gradients can be computed very efficiently with gradient back propergation (aka Chain Rule)

Stochastic Gradient Descent (SGD):

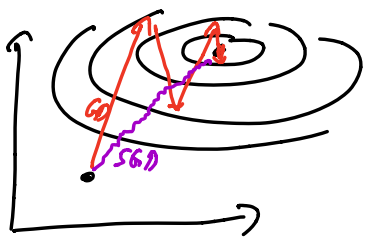
Loss function: $\sum_{i=1}^{n} \ell(h(x_i), y_i)$ e.g. $\frac{1}{2}\sum_{i=1}^{n} (h(x_i) - y_i)^2$

Gradient: $\nabla U = \sum_{i=1}^{n} \frac{\partial \ell(h(x_i), y_i)}{\partial U}$

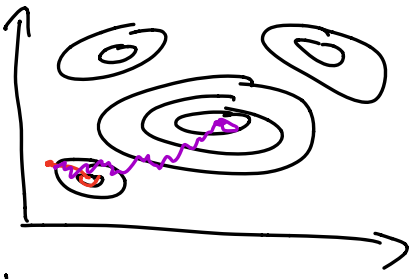
SGD approximates gradient with only 1 or $m \ll n$ samples.

$\nabla U \approx \frac{\partial \ell(h(x_i), y_i)}{\partial U}$ ← single x_i

Perform one tiny update for each sample.



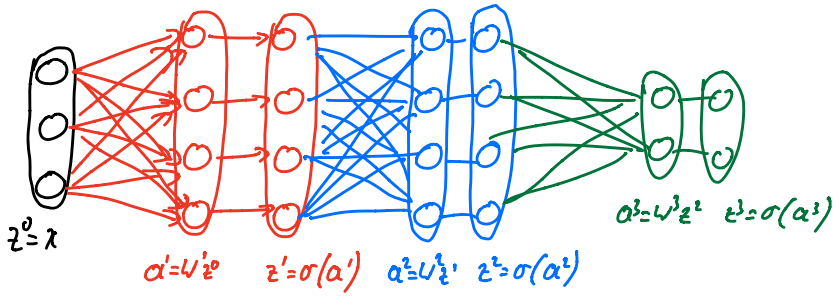
SGD is very noisy. However this turns out to be a crucial "feature" to avoid local minima and saddle points. More precise optimization methods (especially Newton or approximated Newton methods) land in local minima. SGD is "too imprecise" to hit small narrow (bad) local minima.



Deep Networks: One nice property of NN is that we can introduce multiple layers (and increase complexity):

$$\phi(x) = C\,\sigma(B\,\sigma(Ax + \vec{a}) + \vec{b}) + \vec{c}$$

Now it gets more complicated. The first layer learns linear functions. The second layer combines these to multiple non-linear functions. The third layer combines multiple of these non-linear functions to even more complex non-linear functions.



The forward propagation is really simple to implement:

Forward pass: (Prediction)

$z_0 = x$

For $l = 1 : L$

$a_l = W^l z_{l-1} + b_l$

$z_l = \sigma_l(a_l)$

End

Return z_L

b_l: bias term in layer l

Comments: - Typically the last transition function, σ^L, can differ from the others

Amazingly, so is the backwards (gradient) pass:

Backward pass: (Gradient update)

$\vec{\delta}_L = \frac{\partial L}{\partial z_L} \odot \sigma_L'(a_L)$

for $l = L : -1 : 1$

$W_l = W_l - \alpha \vec{\delta}_l z_{l-1}^T$

$b_l = b_l - \alpha \vec{\delta}_l$

$\vec{\delta}_{l-1} = \sigma_{l-1}'(a_{l-1}) \odot (W_l^T \delta_l)$

End

$\odot$ is the element-wise product

$\begin{pmatrix} a \\ b \end{pmatrix} \odot \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \cdot c \\ b \cdot d \end{pmatrix}$

σ_l' is the gradient of the transition function σ_l

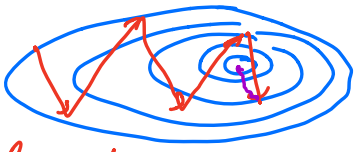
Important optimization tricks:

- Use Momentum

$G \leftarrow \beta G + \frac{\partial L}{\partial W}$

$W \leftarrow W - \alpha G$

- Reduce stepsize during optimization
- Use mini-batch (SGD) with $m \simeq 64$ inputs)
- scale features to be within $[0, 1]$
- de-correlate features (PCA)
- For image data use Convolutional NN



large step-size
small step-size