Neural Networks
(aka Deep Learning)

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

\[ x \rightarrow \phi(x) \quad h(x) = \phi(x)^T \nu + b \]

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 \( \sigma(x) \) is a non-linear transition function, which operates element-wise on each dimension.

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

\[ w^T \phi(x) + b = w^T(Ux) + b = w^TUx + b = \underbrace{v^T}_\text{linear} x + b \quad \text{where: } v = w^Tu \]
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) \[ U = \begin{bmatrix} u_1^T \\ u_2^T \\ u_3^T \end{bmatrix} \]

\[ h(z) = U^T \max(Uz + c) + b = \sum_i u_i^T \max(U_i z + c_i) + b \]  
 plungewise linear approximation \( h(z) \)

\[ u_i < 0 \text{ downwards slope} \]  
linear components

Target function

switches linear components on/off
Learning: We need to learn the parameters of $\phi$, and we can do simple gradient descent.

Let $\ell$ be any well-defined loss function, e.g. $\ell(h) = \frac{1}{2} \sum_{i=1}^{m} (h(x_i) - y_i)^2$ $h$ has parameters: $\widetilde{W}, U, \xi, b$

Simple gradient descent:

\[
\begin{align*}
\widetilde{W} &\leftarrow \widetilde{W} - \alpha \frac{\partial \ell}{\partial \widetilde{W}} \\
U &\leftarrow U - \alpha \frac{\partial \ell}{\partial U} \\
\xi &\leftarrow \xi - \alpha \frac{\partial \ell}{\partial \xi} \\
b &\leftarrow b - \alpha \frac{\partial \ell}{\partial b}
\end{align*}
\]

These gradients can be computed very efficiently with gradient backpropagation (aka Chain Rule).

Stochastic Gradient Descent (SGD):

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

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

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

$\nabla U \approx \frac{\partial \ell(h(x_i), y_i)}{\partial U} \Rightarrow$ 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 NL is that we can introduce multiple layers (and increase complexity):

$$\phi(x) = C \sigma(B \sigma(Ax + \theta) + \theta) + \epsilon$$

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 \( l = 1 : L \)

\[ a_l = U^T z_{l-1} + b_q \]

\[ z_l = \sigma_l(a_l) \]

End  

Return \( z_L \)

Amazingly, so is the backwards (gradient) pass:

**Backward pass: (Gradient Update)**

\[ \delta_L = \delta_{L+1} \circ \sigma'(a_L) \]

for \( l = L-1 : 1 \)

\[ U_b = U_b + \alpha \delta_l \sigma'_l(z_l^T) \]

\[ b_b = b_b + \alpha \delta_l \]

\[ \delta_{l-1} = \sigma'_{l-1}(a_{l-1}) \circ (U_b^T \delta_l) \]

End

\( \circ \) is the element-wise product

\[ \begin{pmatrix} a \\ b \end{pmatrix} \circ \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ bd \end{pmatrix} \]

\( \sigma' \) is the gradient of the transition function \( \sigma \)

Important optimization tricks:

- Use Momentum

- Reduce stepsize during optimization

- Use mini-batch (SGD with \( m \approx 64 \) inputs)

- Scale features to be within \([0,1]\)

- De-correlate features (PCA)

- For image data use Convolutional NN