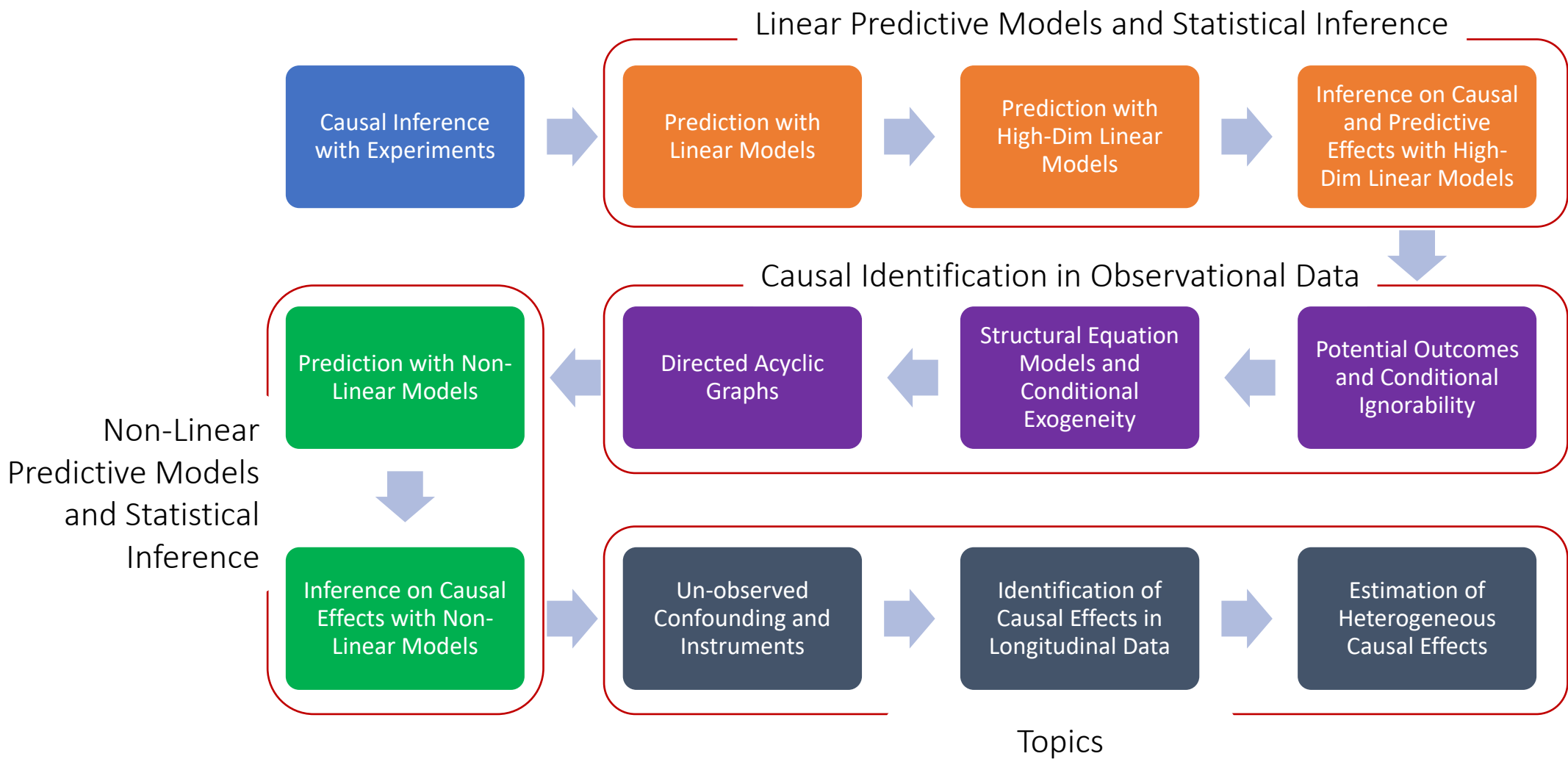
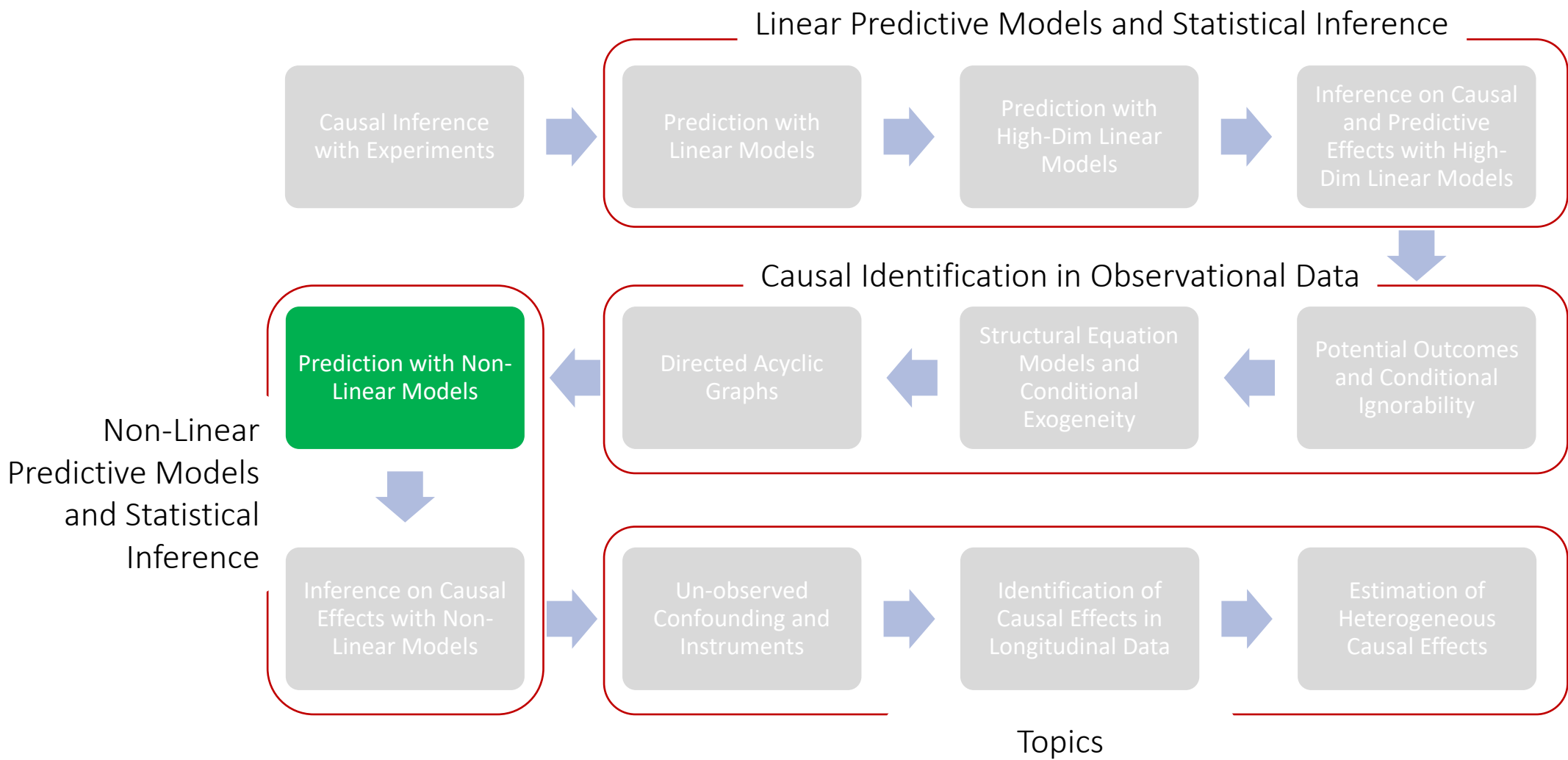


# MS&E 228: Modern Non-Linear Prediction

Vasilis Syrgkanis

MS&E, Stanford

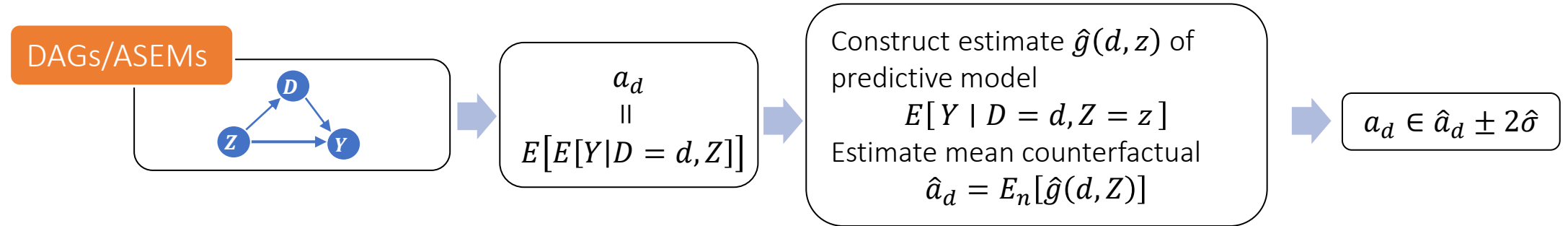




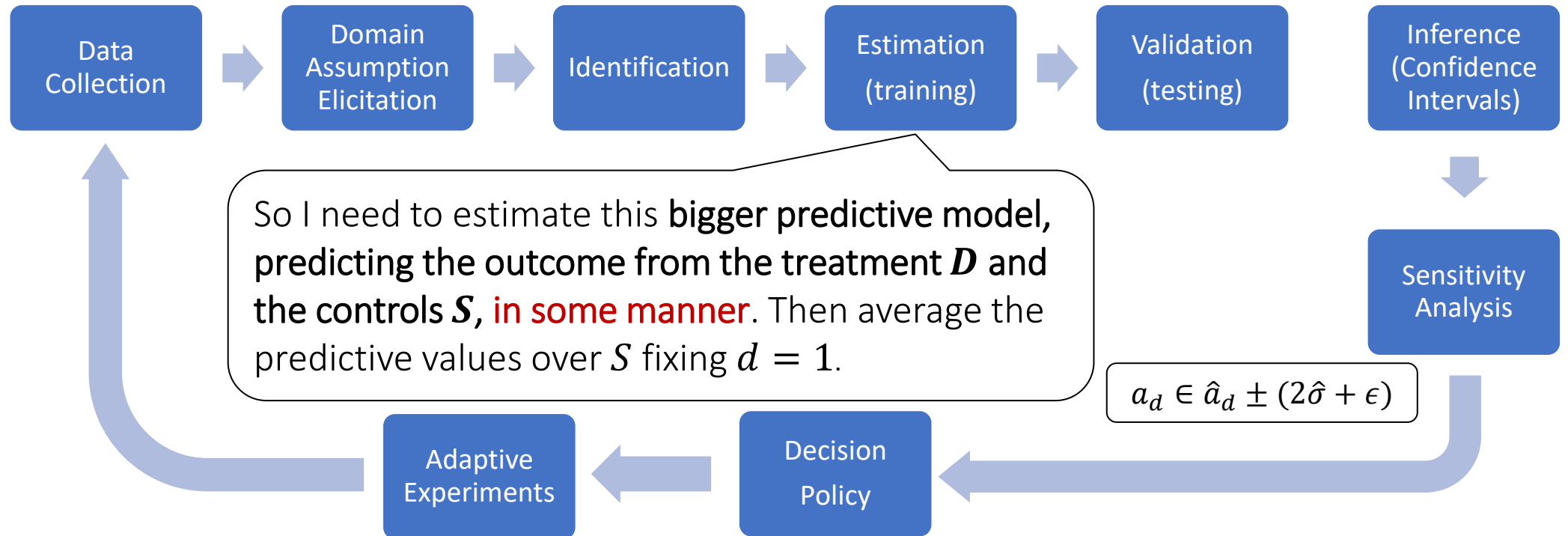
# Recap of Last Lecture

# Causal Inference Pipeline

Theory



Practice



# Problem Statement

- Given  $n$  samples  $(Z_1, Y_1), \dots, (Z_n, Y_n)$  drawn iid from a distribution  $D$
- Want an estimate  $\hat{g}$  that approximates the Best Prediction

$$g := \operatorname{argmin}_{\tilde{g}} E \left[ (Y - \tilde{g}(Z))^2 \right]$$

- Best Prediction rule is Conditional Expectation Function (CEF)

$$g(Z) = E[Y|Z]$$

- We want our estimate  $\tilde{g}$  to be close to  $g$  in RMSE

$$\|\hat{g} - g\| = \sqrt{E_Z(\hat{g}(S) - g(Z))^2} \rightarrow 0, \quad \text{as } n \rightarrow \infty$$

# The Curse of Dimensionality

- What if we make no real assumption on  $g(Z) := E[Y|Z]$
- Suppose we only assume  $g$  is a smooth function
- Formal form of smoothness:  $g$  is  $\beta$ -smooth if it has uniformly bounded and continuous  $\beta$ -high order derivatives
- Classic non-parametric statistics [Stone'82]: provably best you can do

$$\|g - \hat{g}\| \approx n^{-\frac{\beta}{2\beta+p}}$$

# Bypassing the Curse of Dimensionality

- Lasso scaled to  $p \gg n$  by adapting to notions of “effective dimension” (e.g.  $s/n$ , with  $s$  is number of relevant variables)
- We need methods with similar behavior for non-linear models
- Many modern machine learning techniques achieve exactly that
- Their error scales with appropriate notions of “effective dimension”
- Last time: Regression Trees, Random Forests, Gradient Boosted Forests



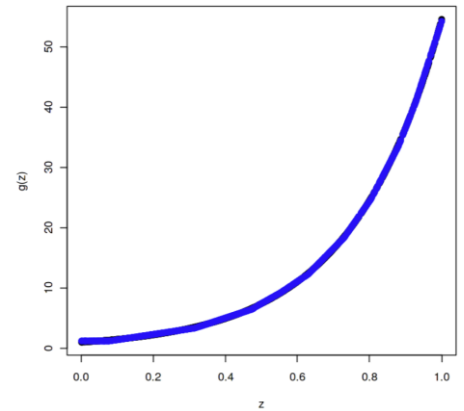
# Goals for Today

- Neural Networks
  - Some theoretical guarantees and justification (similar to lasso)
  - How to combine models (stacking)
  - How to automate the process (automl)
- 
- NNets for feature engineering

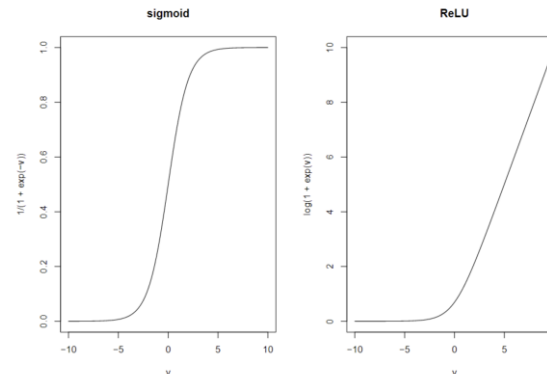
# Modern Non-Linear Predictive Models: Neural Networks

# (Shallow) Neural Networks

- We approximate the CEF with data-driven engineered features
$$g(z) := \beta' \phi(z; a)$$
- Typical choice of  $\phi$  is:
$$\phi(z; a) = \sigma(a'z)$$
- With  $\sigma$  some non-linear function



**Figure 2.12:** Approximation of  $g(Z) = \exp(4Z)$  by a Neural Network



# (Shallow) Neural Network Objective

- Parameters chosen by minimizing penalized empirical square loss

$$\min_{\alpha, \beta} E_n \left[ (Y - \beta' \phi(Z; \alpha))^2 \right] + \lambda \text{pen}(\alpha, \beta)$$

- Penalty is either  $\ell_1$  norm (sparsity inducing) or  $\ell_2$  norm (inducing small weights);  $\lambda$  is referred as *weight decay* in the case of  $\ell_2$
- Loss is typically minimized via *Stochastic Gradient Descent* (SGD)

$$(\alpha, \beta) \leftarrow (\alpha, \beta) - \eta \partial_{\alpha, \beta} \text{Loss}(B; \alpha, \beta)$$

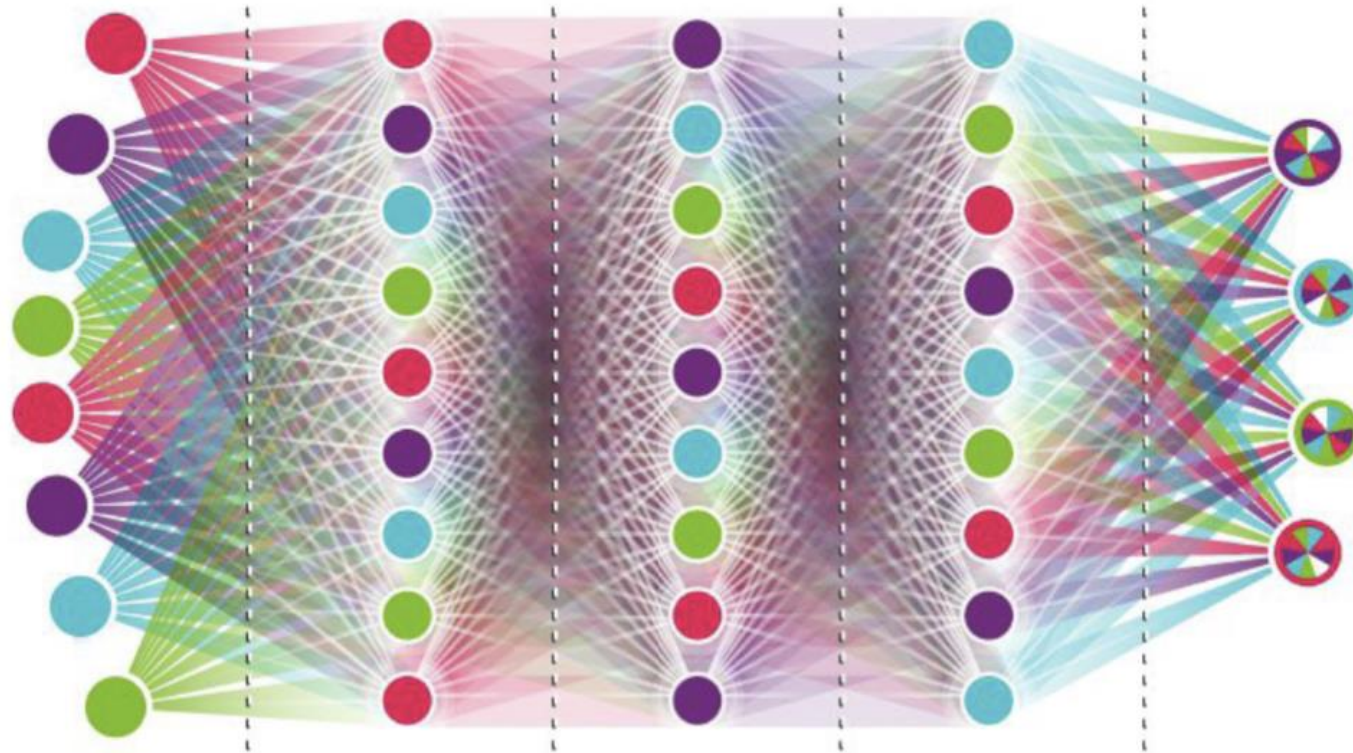
- $\text{Loss}(B; \alpha, \beta)$  is empirical loss calculated on a sub-sample  $B$  (*batch*)

$$\frac{1}{|B|} \sum_{i \in B} (Y_i - \beta' \phi(Z_i; \alpha))^2 + \lambda \text{pen}(\alpha, \beta)$$

- Every pass over all the data is referred to as an *epoch*

## DEEP NEURAL NETWORK

Input layer → Hidden layer 1 → Hidden layer 2 → Hidden layer 3 → Output layer



# Forms of Regularization

- *Stochasticity* and iterative nature of SGD is by itself a regularization method (implicit regularization)
- *Penalties* are explicit form of regularization
- *Drop-out*. At each training step, shutdown some of the neurons. Implicitly regularizes by having multiple neurons learn important concepts, acting as substitutes
- *Early stopping*. Measure out of sample performance after a few iterations of SGD and stop if it stops improving

Let's check it out!

# Some Theory

**Structured Sparsity and Smoothness.** Assume  $g$  is a composition

$$g = f_M \circ \cdots \circ f_0$$

Where  $i$ -th function  $f_i: \mathbb{R}^{p_i} \rightarrow \mathbb{R}^{p_{i+1}}$  has its  $p_{i+1}$  components  $\beta_i$ -smooth (*smoothness*) and depends only  $t_i \ll p_i$  input variables (*sparsity*)

*Effective dimension* is  $s := \max_i n^{-\frac{\beta_i}{2\beta_i + p_i}}$

**Theorem[Schmidt-Hieber'20].** If depth  $\sim \log(n)$  and width  $\geq s \log(n)$ , and several other regularity conditions, then error of an appropriately trained neural network is at most  $\approx \sqrt{\frac{s}{n}} \text{polylog}(n)$



# A Reminder: Fancy isn't always better

---

Predictive performance for  
predicting wages

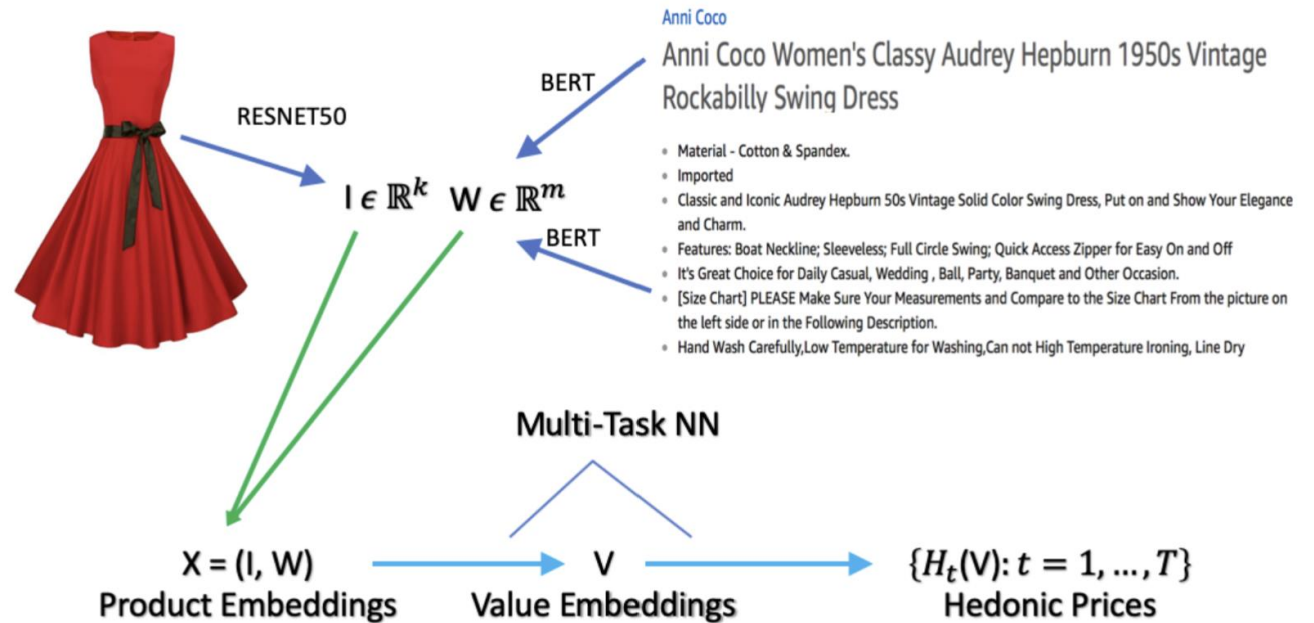
	MSE	S.E.	$R^2$
Least Squares (basic)	0.229	0.016	0.282
Least Squares (flexible)	0.243	0.016	0.238
Lasso	0.234	0.015	0.267
Post-Lasso	0.233	0.015	0.271
Lasso (flexible)	0.235	0.015	0.265
Post-Lasso (flexible)	0.236	0.016	0.261
Cross-Validated Lasso	0.229	0.015	0.282
Cross-Validated Ridge	0.234	0.015	0.267
Cross-Validated Elastic Net	0.230	0.015	0.280
Cross-Validated Lasso (flexible)	0.232	0.015	0.275
Cross-Validated Ridge (flexible)	0.233	0.015	0.271
Cross-Validated Elastic Net (flexible)	0.231	0.015	0.276
Random Forest	0.233	0.015	0.270
Boosted Trees	0.230	0.015	0.279
Pruned Tree	0.248	0.016	0.224
Neural Net	0.276	0.012	0.148



# But many times it is crucial

Predicting prices from  
product characteristics at  
Amazon

Bajari et al. 2021, *Hedonic  
prices and quality adjusted  
price indices powered by AI.*



# Which method should I use?

Stacking and Ensembling

# Use all and combine: Stacking

- If you have many models  $\hat{g}_1, \dots, \hat{g}_K$  we can combine based on out-of-sample performance

$$\begin{aligned} \text{Best -- Loss} &= \max_k E \left[ (Y - \hat{g}_k(Z))^2 \right] \\ &= \max_{w \geq 0: \sum_k w_k = 1} \sum_k w_k E \left[ (Y - \hat{g}_k(Z))^2 \right] \\ &\geq \max_{w \geq 0: \sum_k w_k = 1} E \left[ \left( Y - \sum_k w_k \hat{g}_k(Z) \right)^2 \right] = \text{Loss of Best Ensemble} \end{aligned}$$

# Stacking

- Train an OLS on the out-of-sample data predicting  $Y$  with features  $g_1(Z), \dots, g_K(Z)$  to learn weights  $w$ ; return ensemble prediction

$$\hat{g}(Z) := \sum_{k=1}^K w_k \hat{g}_k(Z)$$

- If models are too many, we can train Lasso on out-of-sample to learn weights, to avoid overfitting!

Let's check it out!

How do I choose all these  
hyperparameters?

# Use Auto-ML frameworks!

- Automatic and clever search over the hyperparameter space
- Very few lines of code
- Typically much better performance than handpicking yourself
- Unless a lot of domain knowledge of what types of functions are better approximators
- Many user-friendly tools: [H2O-AutoML](#), [Auto-Gluon](#), [Azure-AutoML](#), [FLAML](#), [Auto-Sklearn](#), [HyperOpt-Sklearn](#)

# Feature Engineering with Pre-Trained Neural Networks

PCA, Large Language Models, Large Vision Models

# Auto-Encoders



# Reducing Dimensionality via Latent Embeddings

- Suppose we have a high dimensional set of variables  $W \in \mathbb{R}^p$
- One way to address the curse of dimensionality: find an equally good low dimensional representation of  $W$
- Find small set of features  $X \in \mathbb{R}^K$  that “capture all information in  $W$ ”

# Reconstruction Error Objective

- We should be able to predict (reconstruct)  $W$  very accurately from  $X$
- For every original feature  $W_j$  we can predict it well from features  $X$

$$\min_{a_j} E_n \left[ \left( W_j - a_j' X \right)^2 \right] \ll \epsilon$$

- Overall, the following *reconstruction error* should be small

$$\min_{a_1, \dots, a_p} \sum_{j=1}^p E_n \left[ \left( W_j - a_j' X \right)^2 \right] = \min_A E_n [\|W - A'X\|_2^2]$$

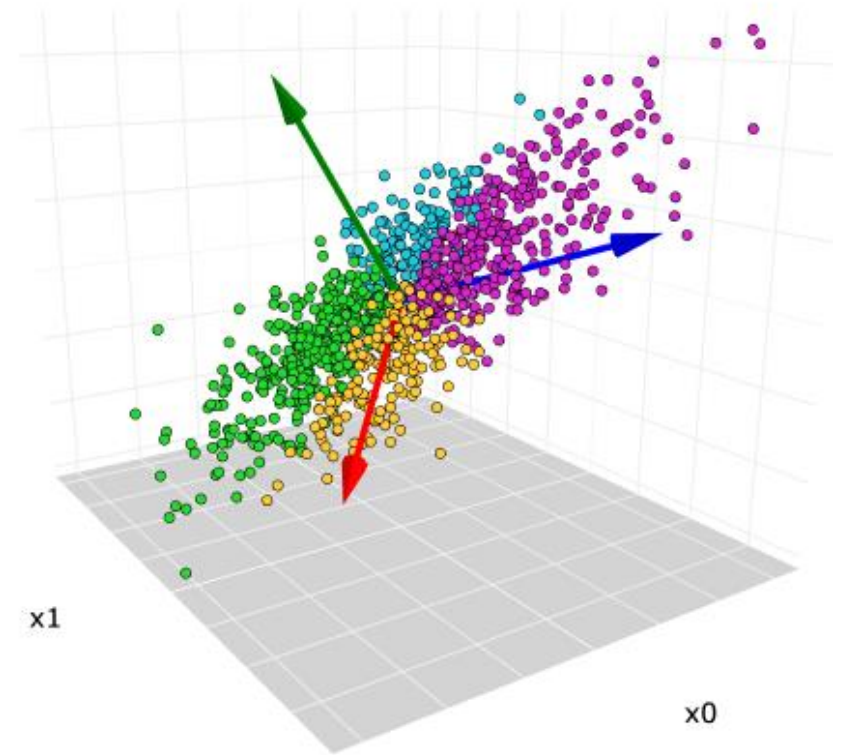
# Principal Components

- Popular approach: find  $X_1, \dots, X_K$  that are un-correlated
- Each feature captures an independent (orthogonal) dimension of variation of the original variables  $W$
- Find  $K$  orthogonal projections of the original variables

$$X_k = c_k' W$$

$$c_k' c_j = 0 \text{ and } c_k' c_k = 1 \text{ and } E[X_k X_j] = 0 \text{ (} k \neq j \text{)}$$

- The best such set of projections  $c_1, \dots, c_k$  such that the “reconstruction error” is minimized is the principal components! (top eigenvalues of cov. matrix  $E_n[WW']$ )



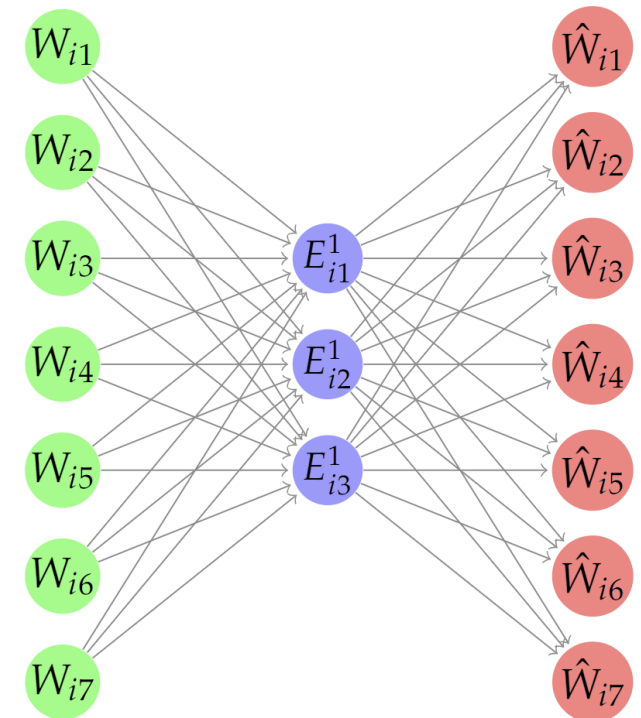
# Principal Components as Encoders-Decoders

- Principal components encode the original data with a linear transform

$$W \rightarrow C'_K W := E \rightarrow A'E =: \hat{W}$$

- “Encoding” process takes a high-dimensional set of features and “encodes” them or “embeds” them in a low dimensional space
- “Decoding” process takes an “encoding” or “embedding” in this low dimensional space and reconstructs the original set of features in the high-dimensional space

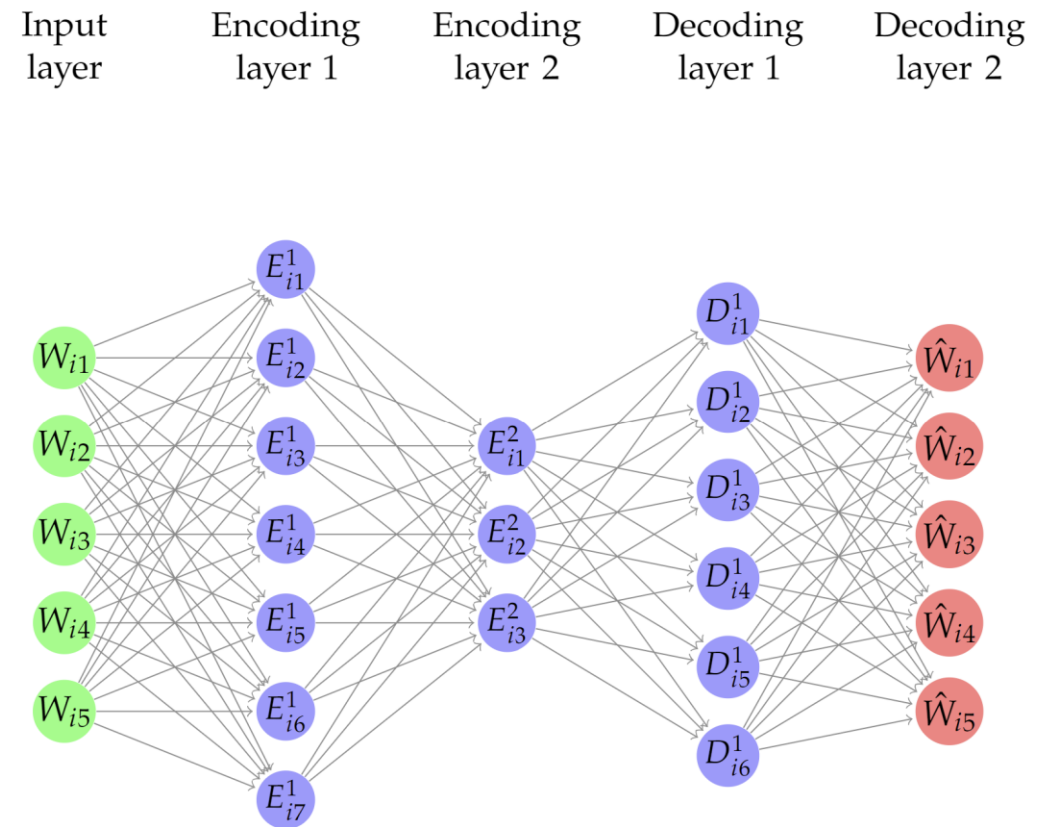
Input layer      Encoding layer      Decoding layer



# Deep Encoders-Decoders (Auto-Encoders)

- Why only linear encoding and decoding functions
- Neural network-based encoders-decoders

$$W \xrightarrow{g_1} E_1 \xrightarrow{g_2} \dots \xrightarrow{g_k} E_k \xrightarrow{f_1} D_1 \xrightarrow{f_2} \dots \xrightarrow{f_m} D_m =: \hat{W}$$



# Deep Auto-Encoders

- Encoding part is an arbitrary function  $e(W): \mathbb{R}^p \rightarrow \mathbb{R}^k$  with  $k \ll p$
- Decoding part is an arbitrary function  $d(X): \mathbb{R}^k \rightarrow \mathbb{R}^p$
- $X = e(W)$  is low dimensional “representation” or “embedding” of  $W$
- Goal is to minimize some notion of “reconstruction error”

$$E_n \left[ \text{loss} \left( W, d(e(W)) \right) \right]$$

# From PCA to ICA

- The PCA reasoning lead to latent embeddings that are un-correlated
- Why not *independent*?
- Independent component analysis (ICA): find latent embeddings  $X$  that can reconstruct  $W$  but are jointly independent (not just uncorrelated)
- *Linear ICA*: find such latent embeddings that are also linear  $X = C'W$
- Admits a clean solution with provable guarantees ([Common'94](#))
- *Non-Linear ICA*: new literature on “disentangled representations”
- No clean solution ([Locatello et al.'19](#)); requires auxiliary information on latent factors  $X$  (e.g. [Hyvarinen's work](#))

# Variational Auto-Encoders

- So far mapping from  $W$  to  $X$  was deterministic
- More realistic:  $W$  is a stochastic projection of a low dimensional vector  $X$  onto a high-dimensional space
- In this case, we cannot reverse engineer  $X$  from  $W$  deterministically
- But we can maybe find a “posterior distribution” of  $X$  given  $W$



# Variational Auto-Encoders

- *Bottomline*: introduce randomness in the encoding part
- Introduce noise vector  $Z$  exogenously drawn (e.g. multivariate normal)
- $X = e(W, Z)$  is a low dimensional “sampled representation” of  $W$ , attempting to approximate the posterior distribution of  $X$  given  $W$
- Goal is to minimize some regularized notion of “reconstruction error”

$$E_n \left[ \text{loss} \left( W, d(e(W, Z)) \right) \right]$$

# Variational Auto-Encoders

- Typically,  $Z$  is multivariate standard normal and  $e(W, Z)$  of the form
$$e(W, Z) = \mu(W) + \Sigma(W) \cdot Z$$
- $\mu(W)$ : deterministic encoding of the mean of the posterior  $X|W$
- $\Sigma(W)$ : deterministic encoding of the variance of the posterior  $X|W$
- The deterministic quantities  $\mu(W)$  and  $\Sigma(W)$  can be used as “embeddings” or engineered features on downstream tasks

# General Embeddings

# From Auto-Encoders to General Embeddings

- Auto-encoding (reconstruction objective) not the only objective to construct good embeddings
- *Generally*: consider many auxiliary tasks with different target outcomes  $A$  that resemble the task we want to solve
- *Goal*: find a common embedding  $X := e(W)$  that can be used to accomplish all tasks well, i.e.  $\min_f E_n[\text{loss}(A, f(X))] \ll \text{small}$ , for all target outcomes  $A$
- *Reasoning*: if the embedding carries sufficient information from  $W$  to be able to accomplish all these related tasks, then it should carry enough information to solve the task we care about

# From Auto-Encoders to General Embeddings

- For text data, find an embeddings that perform well in many language tasks (e.g. Q&A, fill the gaps, predict next word)
- For image data, find embeddings that perform well in many vision tasks (e.g. classification, object detection etc.)
- Typically, we have much larger data sets for these auxiliary tasks than for the task at hand (e.g. web crawling data)
- We are essentially using these auxiliary data for more informed dimensionality reduction

# Embeddings for Text Data

# Word2vec

- We have a corpus of documents, containing a dictionary of  $n$  words
- Trivially we can embed words in  $n$ -dimensional one-hot encoding

$$e_j = (0, \dots, 1, \dots, 0)$$

- Too high-dimensional and not carrying “similarity” information
- Alternative: we want to find lower representations

$$u_1, \dots, u_n$$

# Word2vec

- Construct representations by solving a “fill-the-gap” text problem
- For each “middle word”  $s$  calculate average representation of  $K$  neighboring words  $U_s = \frac{1}{K} \sum_{t \in N} u_t$
- Predict middle word equal to  $j$  given “context”  $U_s$ ; logistic regression

$$p(j; \theta, u) = \Pr(T_s = j | \{T_t\}_{t \in N}) \propto \exp(\theta_j' U_s)$$

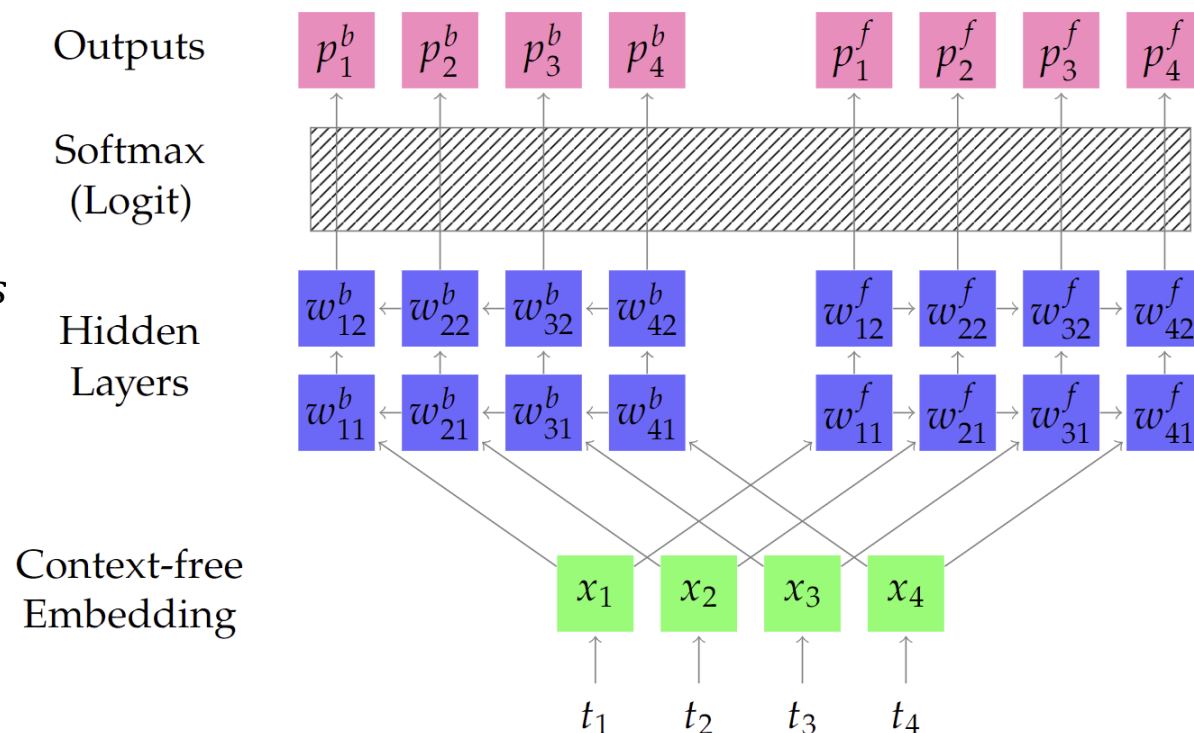
- Maximize over  $\{\theta_j\}_{j=1}^n$  and  $\{u_t\}_{t=1}^K$  the log-likelihood of observed data

$$\sum_s \log(p(T_s; \theta, u))$$



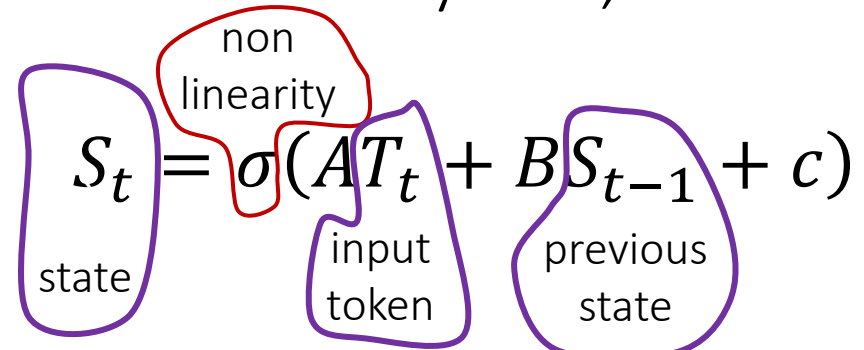
# Second Generation: ELMo

- Word2vec uses very rigid neighborhood model
- Distance from target word is not even incorporated in neighborhood embedding  $U_s$
- [ELMo](#) addresses these issues by using Recurrent Neural Networks

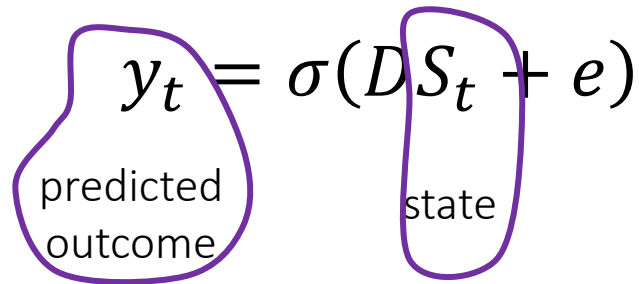


# Recurrent Neural Network

- A non-linear version of an auto-regressive model (Box-Jenkins)
- Nnet parses input “tokens” one-by-one; at each step  $t$

$$S_t = \sigma(AT_t + BS_{t-1} + c)$$


The diagram shows the equation  $S_t = \sigma(AT_t + BS_{t-1} + c)$  with several handwritten annotations in purple and red. A purple oval around  $S_t$  is labeled "state". A red oval around the  $\sigma$  function is labeled "non linearity". A purple oval around  $T_t$  is labeled "input token". A purple oval around  $S_{t-1}$  is labeled "previous state".

$$y_t = \sigma(DS_t + e)$$


The diagram shows the equation  $y_t = \sigma(DS_t + e)$  with handwritten annotations in purple. A purple oval around  $y_t$  is labeled "predicted outcome". A purple oval around  $S_t$  is labeled "state".

# Recurrent Neural Network

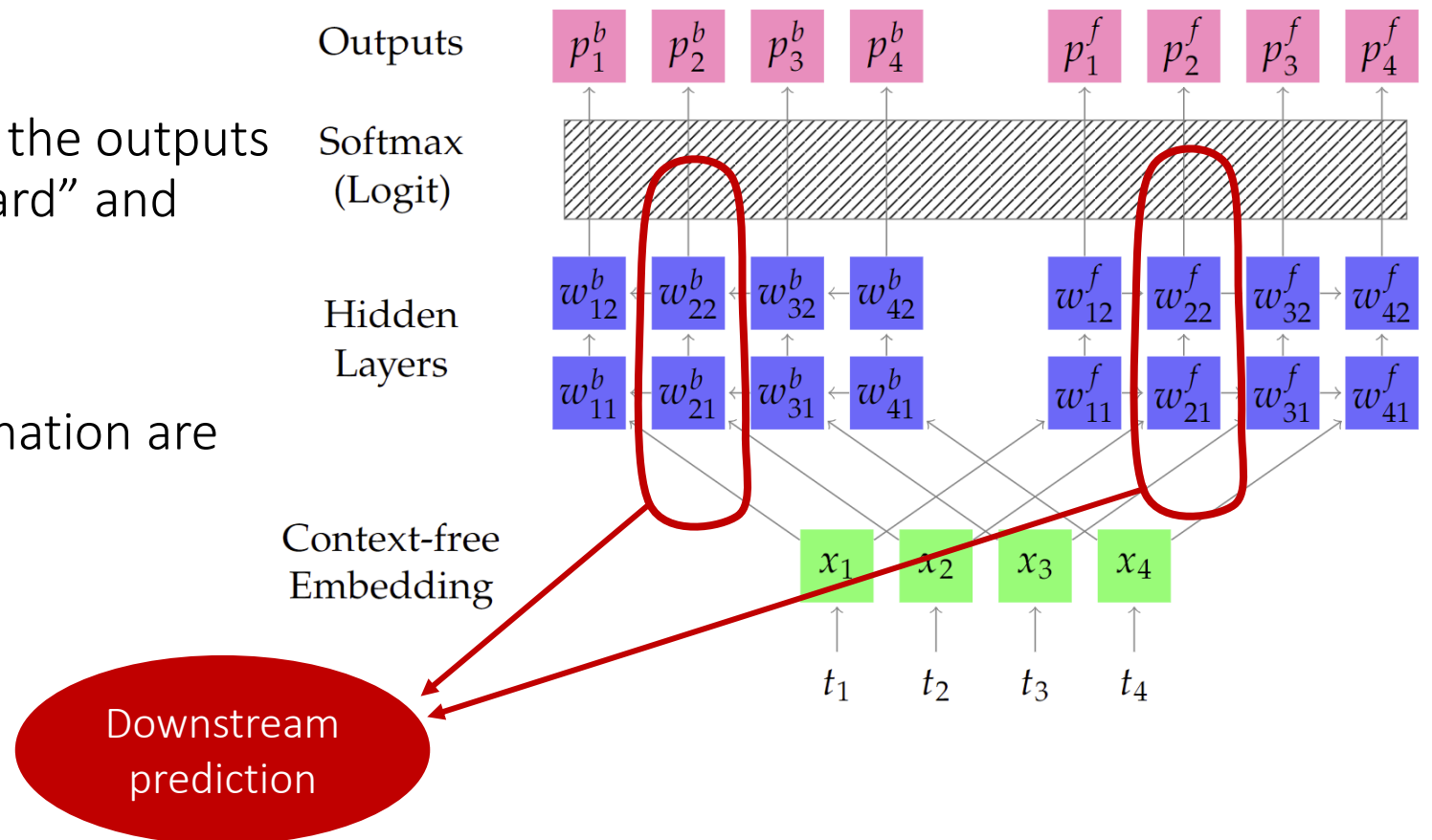
- A non-linear version of an auto-regressive model (Box-Jenkins)
- Nnet parses input “tokens” one-by-one; at each step  $t$

$$S_t = \sigma(AT_t + BS_{t-1} + c)$$
$$y_t = \sigma(DS_t + e)$$

- When used for next word prediction problem it uses the actual “sequence” information; distance is encoded in the state
- Can capture long-range dependencies (albeit not-directly and unstable training)
- For this ELMo uses special RNN called a Long Short-Term Memory (LSTM) that enables more long-range “effects”

# Embeddings from ELMo

- For each token in position  $k$
- Consider a linear combination of the outputs of the neurons in the  $k$ -th “forward” and “backward” prediction column
- The weights on this linear combination are trained for the downstream task



# Second Generation: BERT

- ELMo is very sequential and one-directional
- The context of a word in the prediction is either “previous words” or “subsequent words”
- Better to use context from both “directions” aka “bi-directional”
- Long-range dependencies in language are hard to capture (despite LSTM)
- We might want to create higher level “neighborhoods” that go beyond the local neighborhoods implicitly used in LSTM/RNN

# Second Generation: BERT

- Better to use context from both “directions” aka “bi-directional”
- We might want to create higher level “neighborhoods” that go beyond the local neighborhoods implicitly used in LSTM/RNN
- We need the network to also learn “neighborhood structures”
- These “neighborhood structures” are known as “attention regions”
- When we calculate the “context” for predicting a word we want to take into account the average context of all the words in the attention region related to that word

# Second Generation: BERT

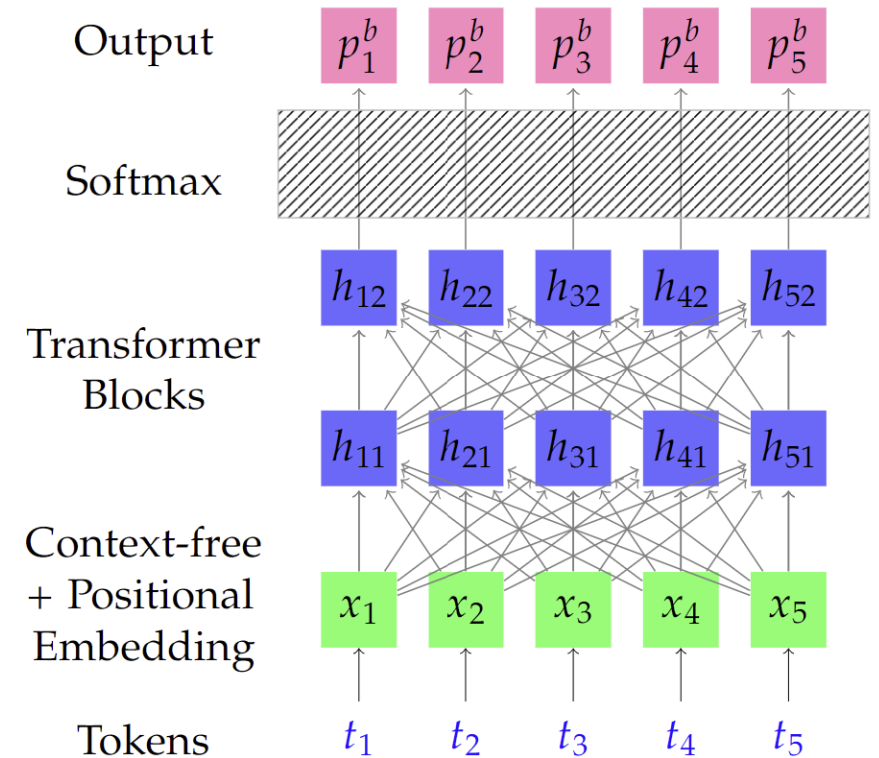
- Better to use context from both “directions” aka “bi-directional”
- We might want to create higher level “neighborhoods” that go beyond the local neighborhoods implicitly used in LSTM/RNN
- Constructing such “trainable” attention regions associated with a word is what a “Transformer” architecture does
- So [BERT](#) uses **Bi-directional Encoder Representations** built with Transformers

# Second Generation: BERT

- The parameters are trained using two language tasks
- Mask: a random set of words in the document is masked and the goal is to predict them
- Pair: pairs of sentences are given and the goal is to predict if one sentence follows the other

Embeddings from BERT can be done in two ways:

- Outputs of the last few Transformer layers as features
- Fine tune whole network to target task; append a predictor at the end of the Transformer layers



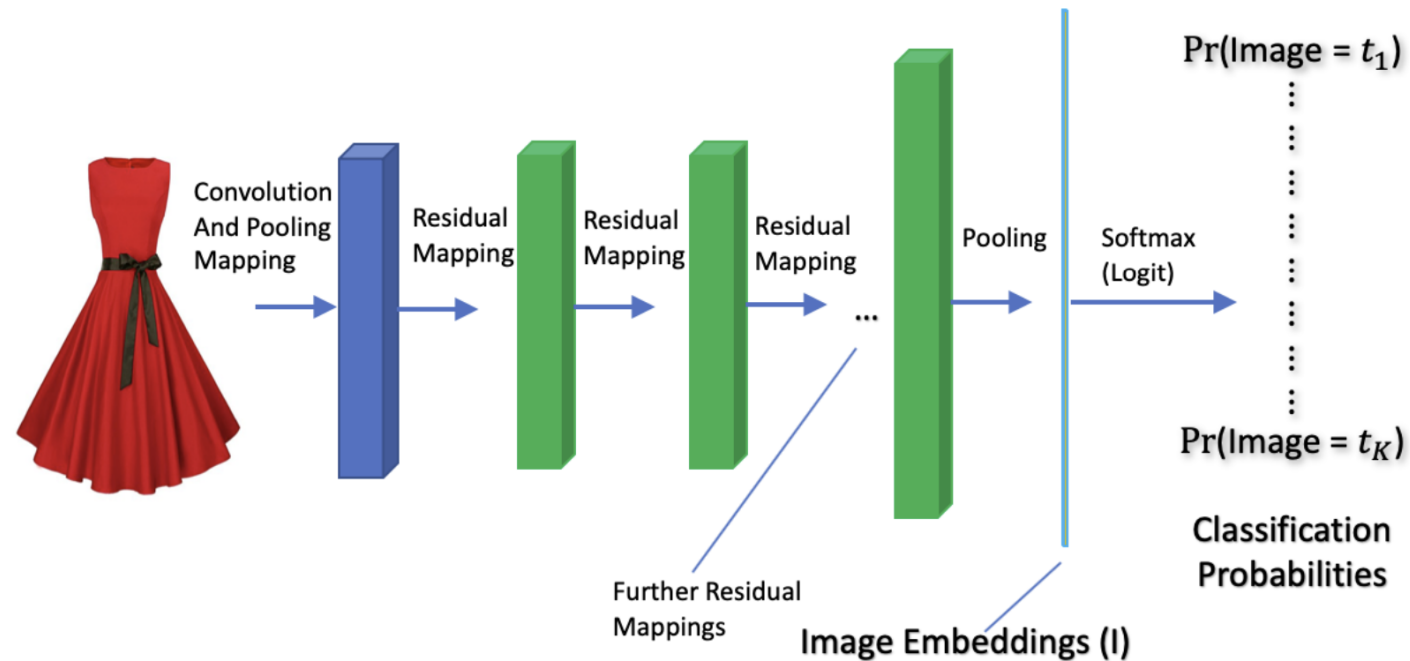


## Third Generation: GPT (Generative Pre-Training) Family

- Ideas from the auto-regressive approach of ELMo (predict next words)
- With attention based encoding ideas in BERT, i.e. Transformers

# Embeddings for Image Data

# Classification Tasks and Convolutional Nets



ResNet: each block allows the input signal to also flow in-tact without a non-linearity

# Embeddings for Price Prediction

Predicting prices from product characteristics at Amazon

Bajari et al. 2021, *Hedonic prices and quality adjusted price indices powered by AI.*

Nnet on top of embeddings  $\approx 90\%$

Random Forest on embeddings  $\approx 80\%$

Linear model of embeddings  $\approx 70\%$

Linear model on simple features  $\approx 40\%$

