Deep Learning

Deep Learning

Neural networks became popular in the 1980s.
Lots of successes, hype, and great conferences: NeurIPS, Snowbird.

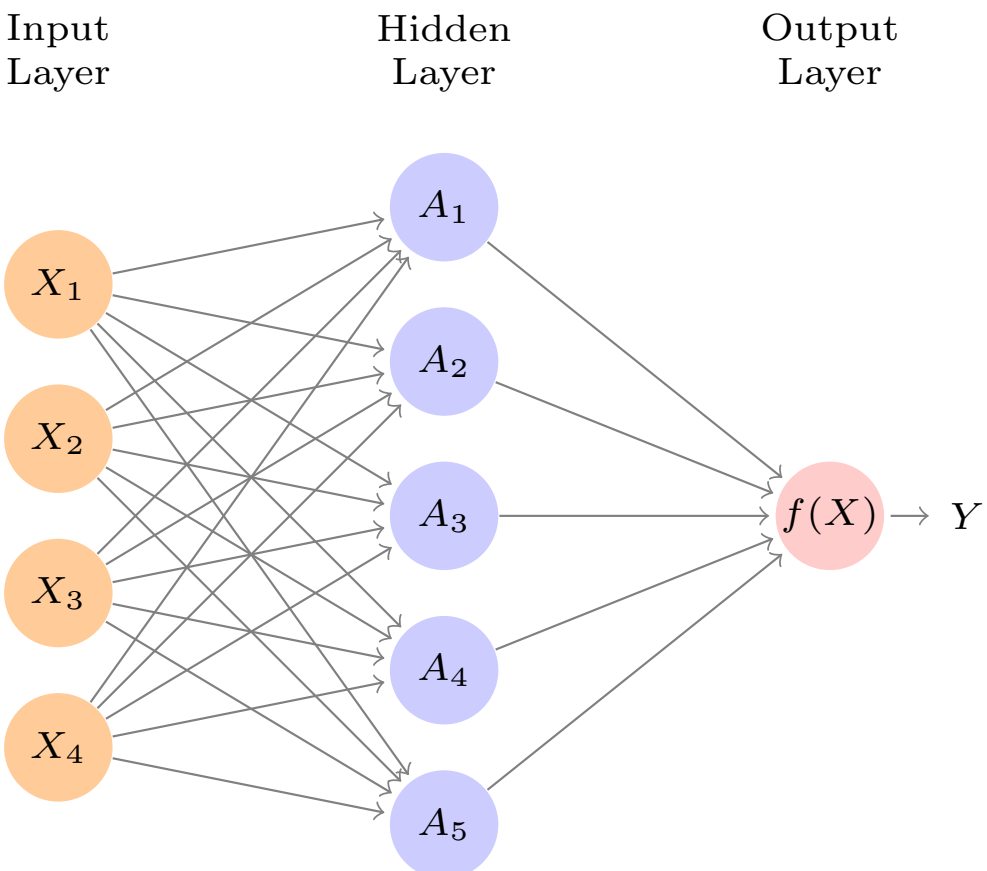
Then along came SVMs, Random Forests and Boosting in the 1990s, and Neural Networks took a back seat.

Re-emerged around 2010 as *Deep Learning.*
By 2020s very dominant and successful.

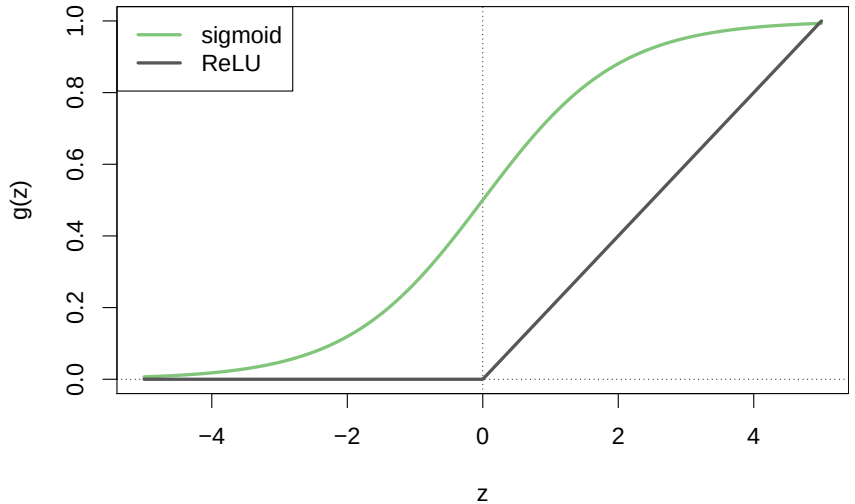
Part of success due to vast improvements in computing power, larger training sets, and software: Tensorflow and PyTorch.

Single Layer Neural Network

$$
\begin{aligned}
f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\
&= \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j).
\end{aligned}
$$

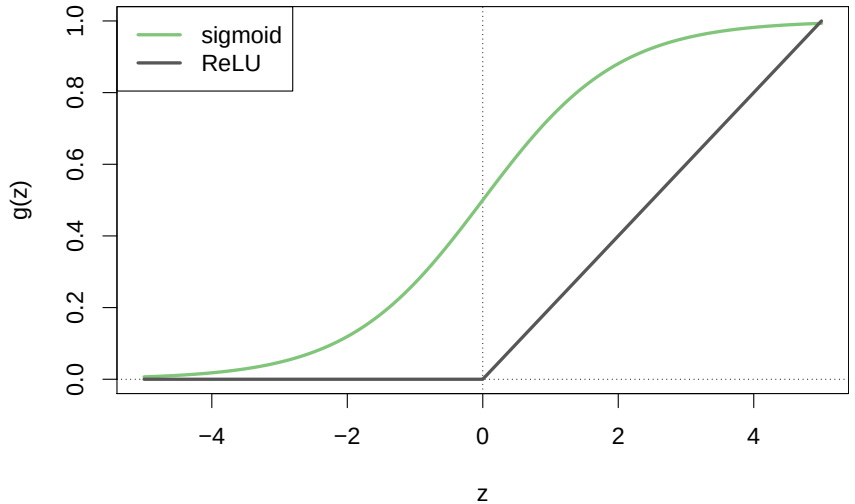


Details



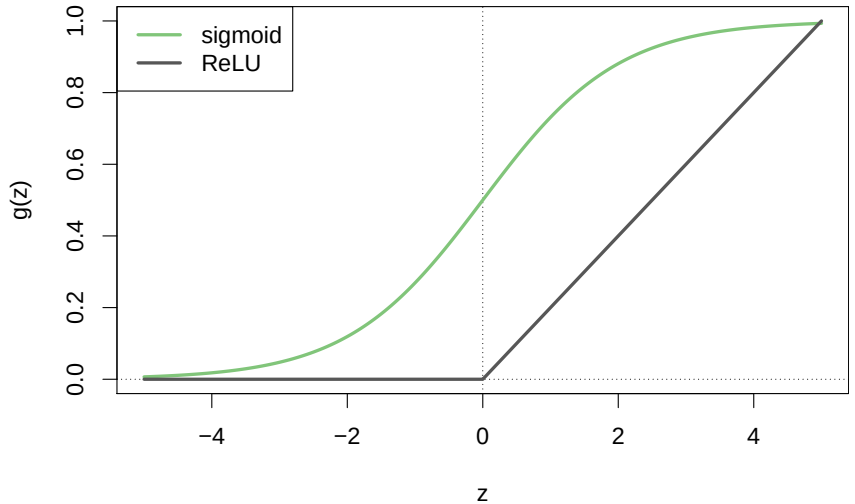
- $A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ are called the *activations* in the *hidden layer*.

Details



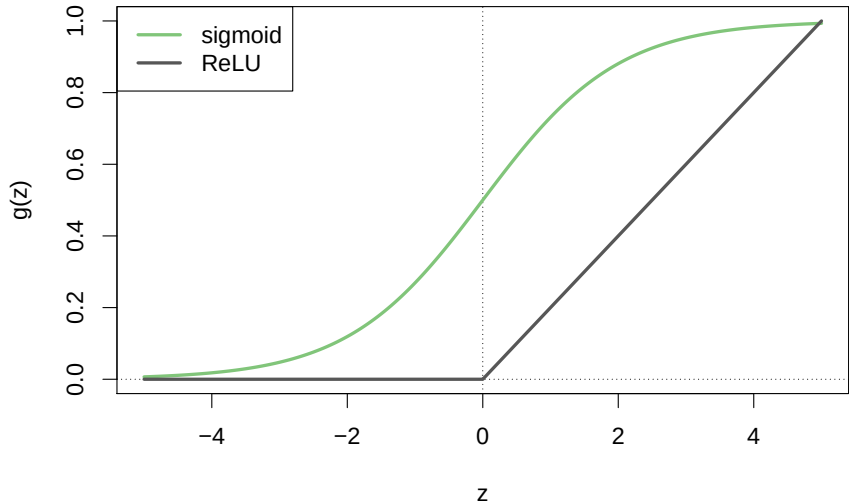
- $A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ are called the *activations* in the *hidden layer*.
- $g(z)$ is called the *activation function*. Popular are the *sigmoid* and *rectified linear*, shown in figure.

Details



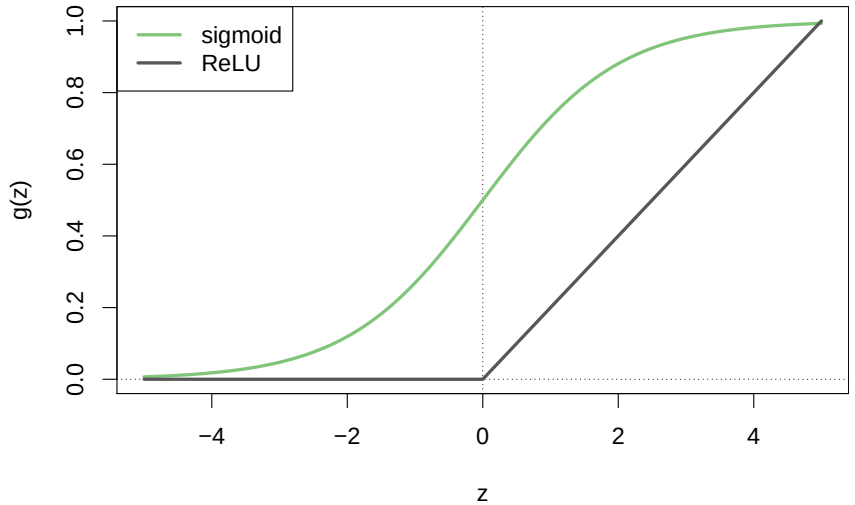
- $A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ are called the *activations* in the *hidden layer*.
- $g(z)$ is called the *activation function*. Popular are the *sigmoid* and *rectified linear*, shown in figure.
- Activation functions in hidden layers are typically nonlinear, otherwise the model collapses to a linear model.

Details



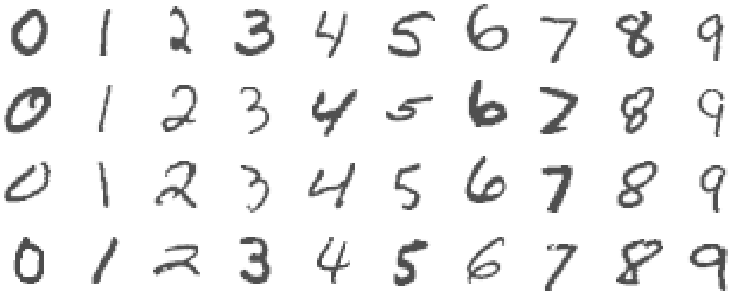
- $A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ are called the *activations* in the *hidden layer*.
- $g(z)$ is called the *activation function*. Popular are the *sigmoid* and *rectified linear*, shown in figure.
- Activation functions in hidden layers are typically nonlinear, otherwise the model collapses to a linear model.
- So the activations are like derived features — nonlinear transformations of linear combinations of the features.

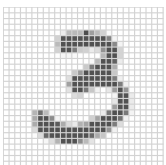
Details

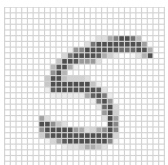


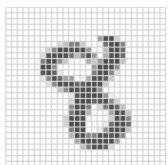
- $A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ are called the *activations* in the *hidden layer*.
- $g(z)$ is called the *activation function*. Popular are the *sigmoid* and *rectified linear*, shown in figure.
- Activation functions in hidden layers are typically nonlinear, otherwise the model collapses to a linear model.
- So the activations are like derived features — nonlinear transformations of linear combinations of the features.
- The model is fit by minimizing $\sum_{i=1}^{n} (y_i - f(x_i))^2$ (e.g. for regression).

Example: MNIST Digits



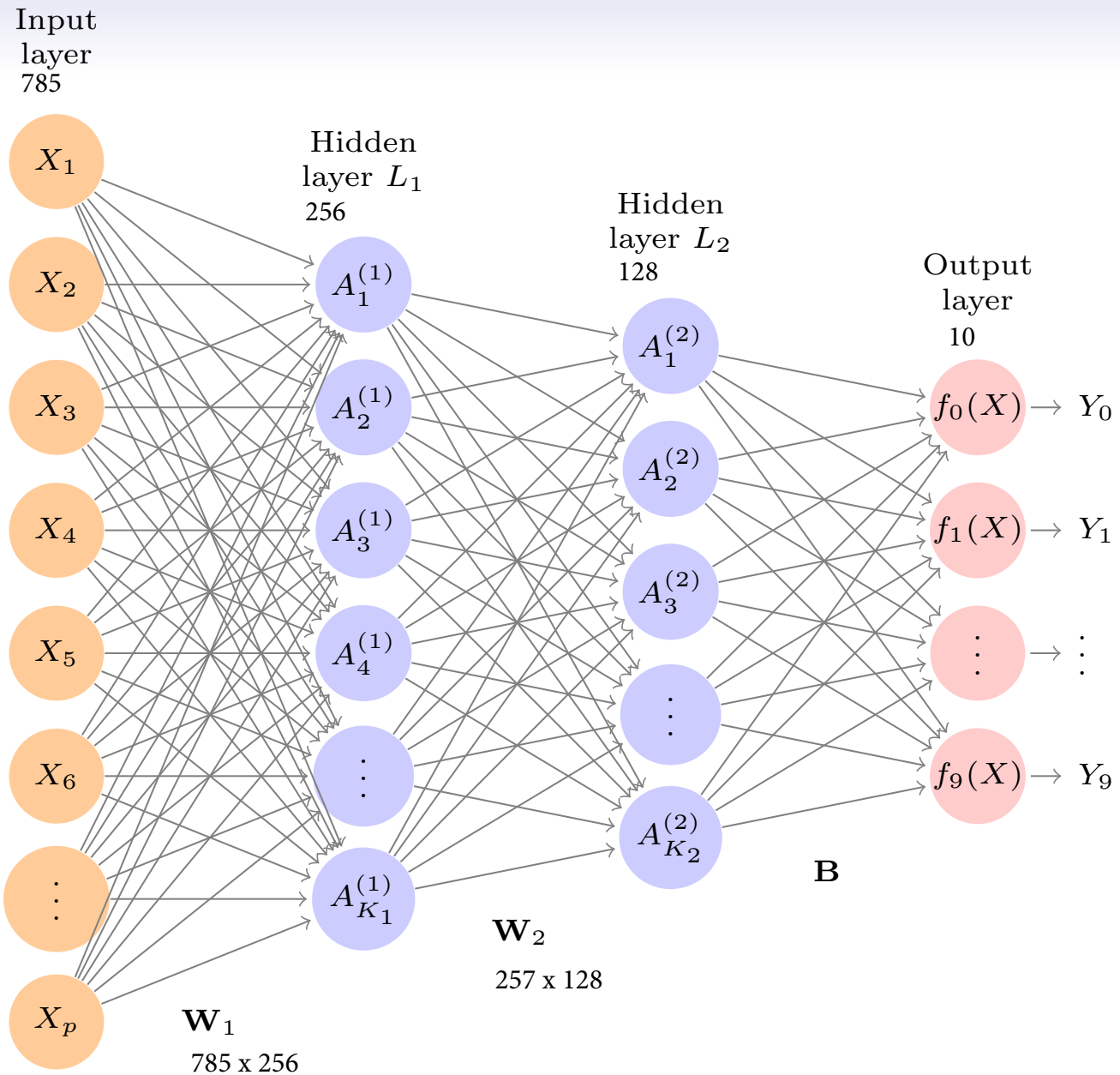






Handwritten digits
28×28 grayscale images $60K$ train, $10K$ test images
Features are the 784 pixel grayscale values $\in (0, 255)$
Labels are the digit class 0–9

- Goal: build a classifier to predict the image class.
- We build a two-layer network with 256 units at first layer, 128 units at second layer, and 10 units at output layer.
- Along with intercepts (called *biases*) there are 235,146 parameters (referred to as *weights*)


Input
layer
785
X_1
X_2
X_3
X_4
X_5
X_6
X_p
Hidden
layer L_1
256
$A_1^{(1)}$
$A_2^{(1)}$
$A_3^{(1)}$
$A_4^{(1)}$
$A_{K_1}^{(1)}$
Hidden
layer L_2
128
$A_1^{(2)}$
$A_2^{(2)}$
$A_3^{(2)}$
$A_{K_2}^{(2)}$
Output
layer
10
$f_0(X)$ → Y_0
$f_1(X)$ → Y_1
$f_9(X)$ → Y_9
$\mathbf{W}_1$
785 x 256
$\mathbf{W}_2$
257 x 128
$\mathbf{B}$

Details of Output Layer

- Let $Z_m = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}$, $m = 0, 1, \ldots, 9$ be 10 linear combinations of activations at second layer.
- Output activation function encodes the *softmax* (i.e multinomial logit) function

$$f_m(X) = \Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{\ell=0}^{9} e^{Z_\ell}}.$$

- We fit the model by minimizing the negative multinomial log-likelihood (or *cross-entropy*):

$$-\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log(f_m(x_i)).$$

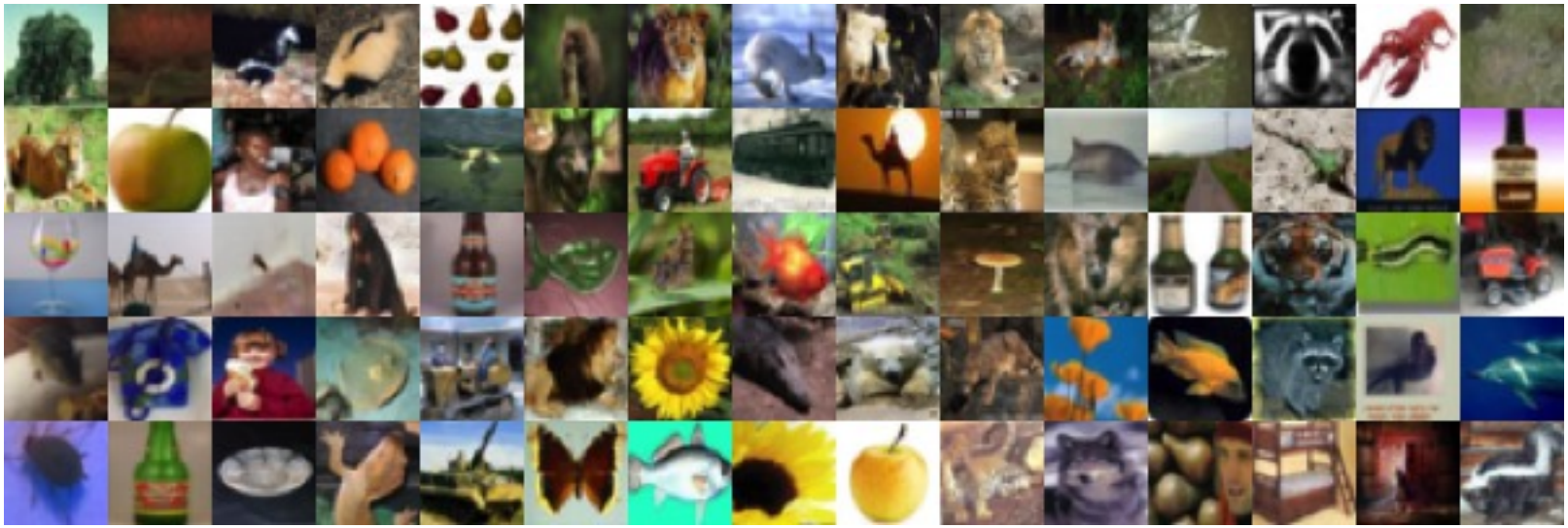
- y_{im} is 1 if true class for observation i is m, else 0 — i.e. *one-hot encoded* (i.e. class dummy variables)

Results

Method	Test Error
Neural Network + Ridge Regularization	2.3%
Neural Network + Dropout Regularization	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

- Early success for neural networks in the 1990s.
- With so many parameters, regularization is essential.
- Some details of regularization and fitting will come later.
- Very overworked problem — best reported rates are $< 0.5\%$!
- Human error rate is reported to be around 0.2%, or 20 of the 10K test images.

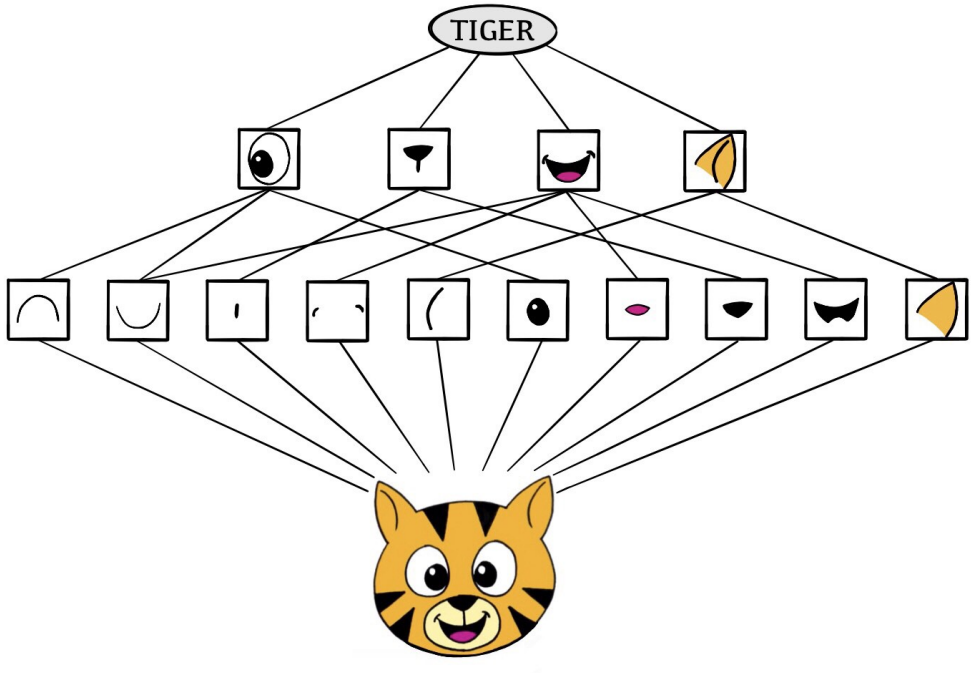
Convolutional Neural Network — CNN



- Major success story for classifying images.
- Shown are samples from `CIFAR100` database. 32×32 color natural images, with 100 classes.
- $50K$ training images, $10K$ test images.

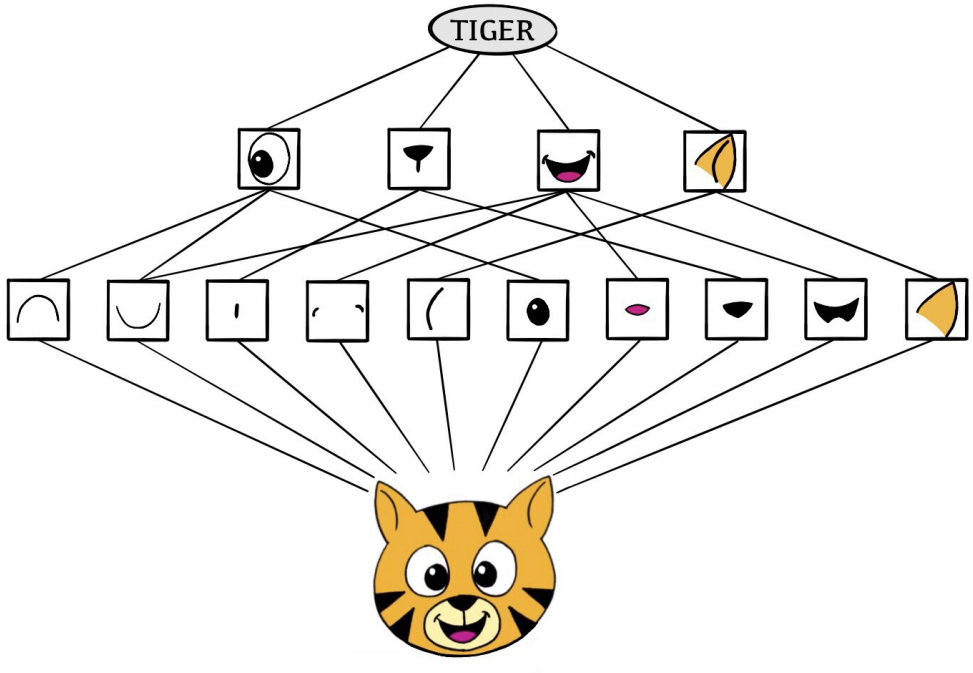
Each image is a three-dimensional array or *feature map:* $32 \times 32 \times 3$ array of 8-bit numbers. The last dimension represents the three color channels for red, green and blue.

How CNNs Work



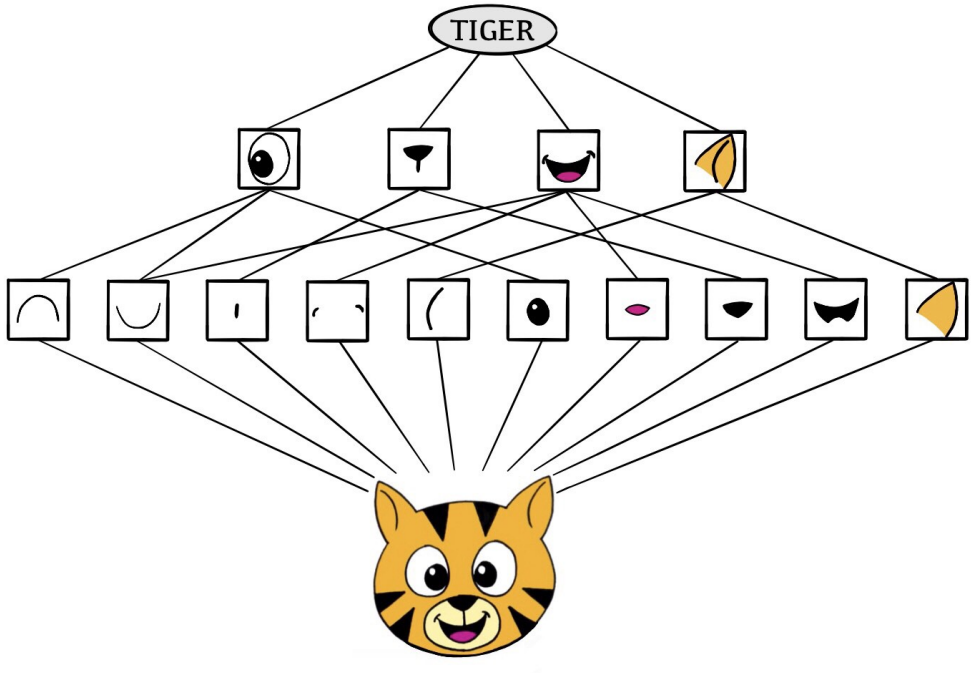
- The CNN builds up an image in a hierarchical fashion.

How CNNs Work



- The CNN builds up an image in a hierarchical fashion.
- Edges and shapes are recognized and pieced together to form more complex shapes, eventually assembling the target image.

How CNNs Work



- The CNN builds up an image in a hierarchical fashion.
- Edges and shapes are recognized and pieced together to form more complex shapes, eventually assembling the target image.
- This hierarchical construction is achieved using *convolution* and *pooling* layers.

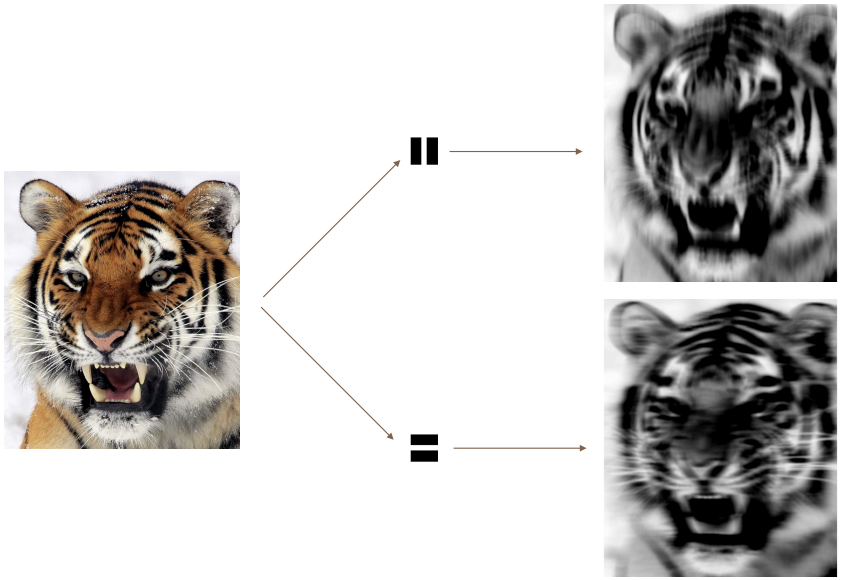
Convolution Filter

$$\text{Input Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \quad \text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

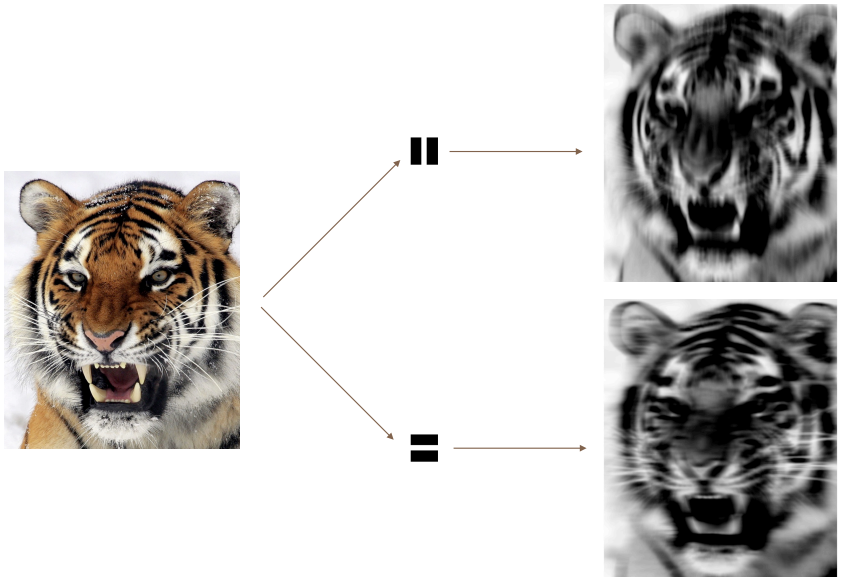
- The filter is itself an image, and represents a small shape, edge etc.
- We slide it around the input image, scoring for matches.
- The scoring is done via *dot-products*, illustrated above.
- If the subimage of the input image is similar to the filter, the score is high, otherwise low.
- The filters are *learned* during training.

Convolution Example



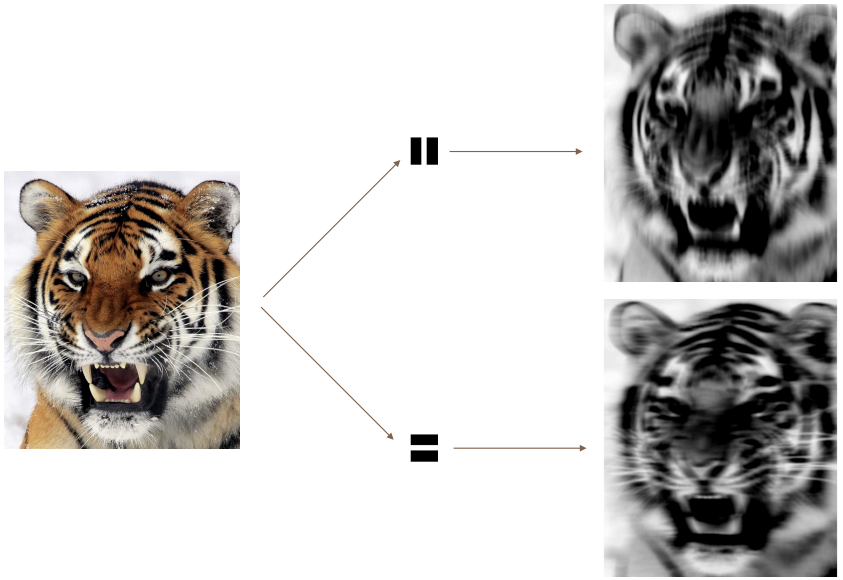
- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.

Convolution Example



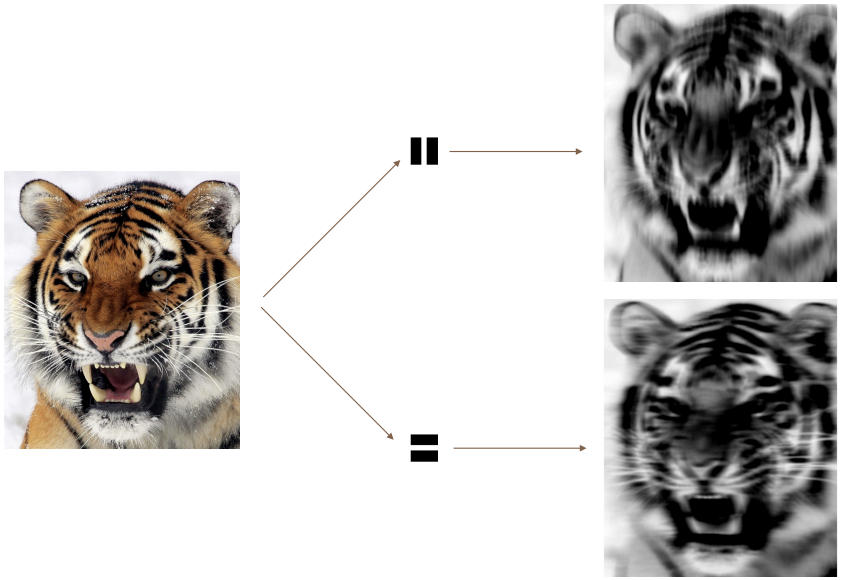
- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.
- The two filters shown here highlight vertical and horizontal stripes.

Convolution Example



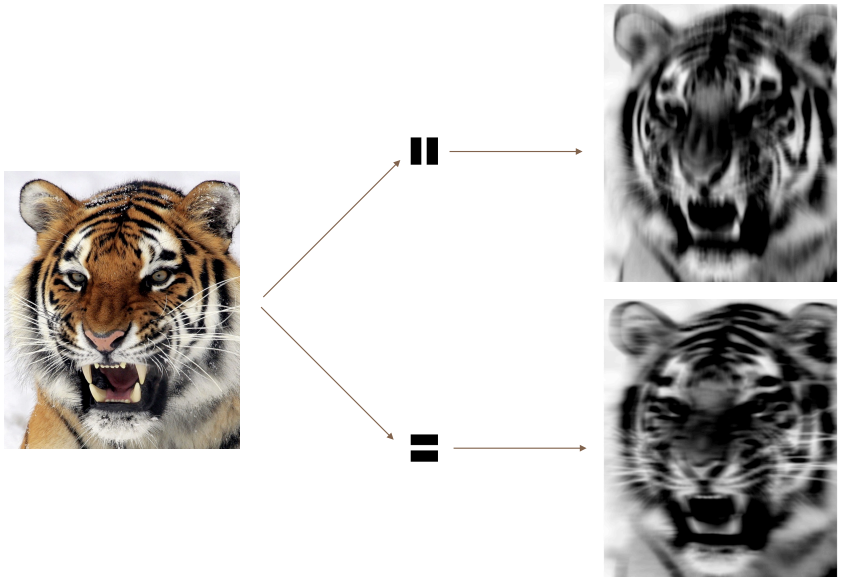
- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.
- The two filters shown here highlight vertical and horizontal stripes.
- The result of the convolution is a new feature map.

Convolution Example



- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.
- The two filters shown here highlight vertical and horizontal stripes.
- The result of the convolution is a new feature map.
- Since images have three colors channels, the filter does as well: one filter per channel, and dot-products are summed.

Convolution Example



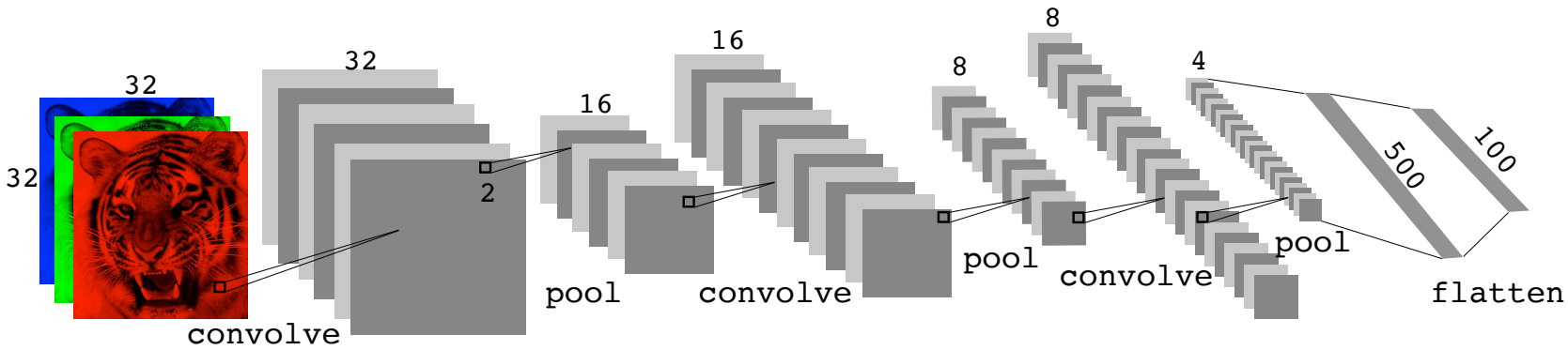
- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.
- The two filters shown here highlight vertical and horizontal stripes.
- The result of the convolution is a new feature map.
- Since images have three colors channels, the filter does as well: one filter per channel, and dot-products are summed.
- The weights in the filters are *learned* by the network.

Pooling

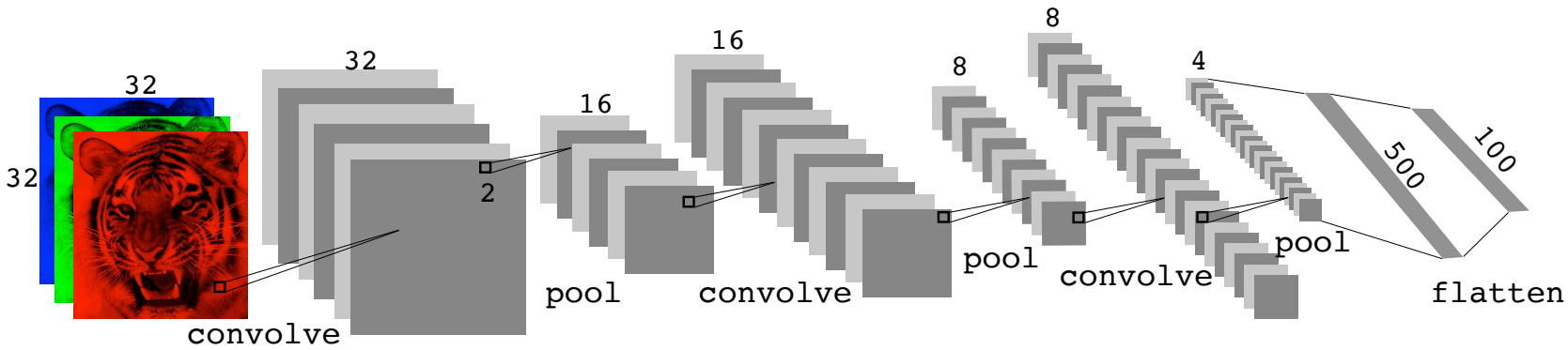
$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

- Each non-overlapping 2×2 block is replaced by its maximum.
- This sharpens the feature identification.
- Allows for locational invariance.
- Reduces the dimension by a factor of 4 — i.e. factor of 2 in each dimension.

Architecture of a CNN

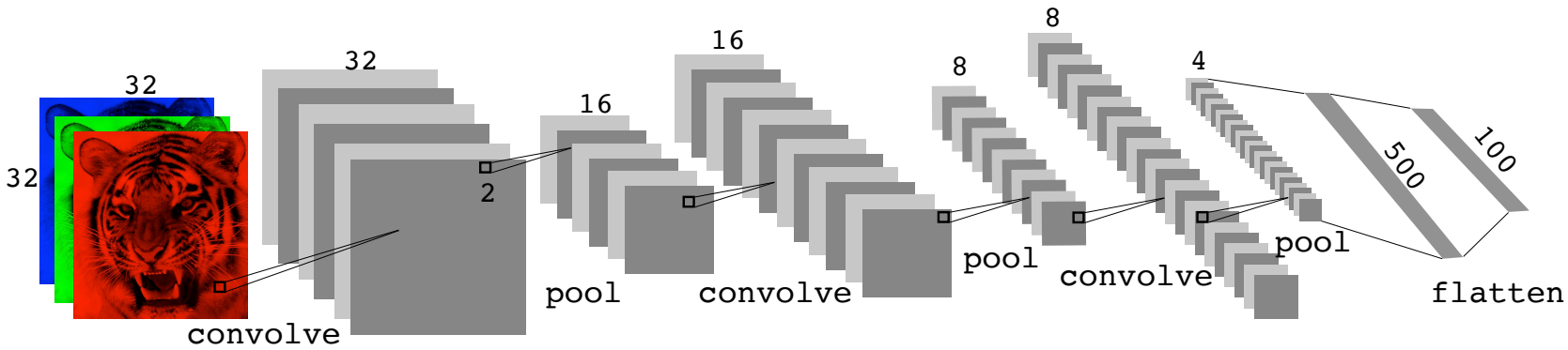


Architecture of a CNN



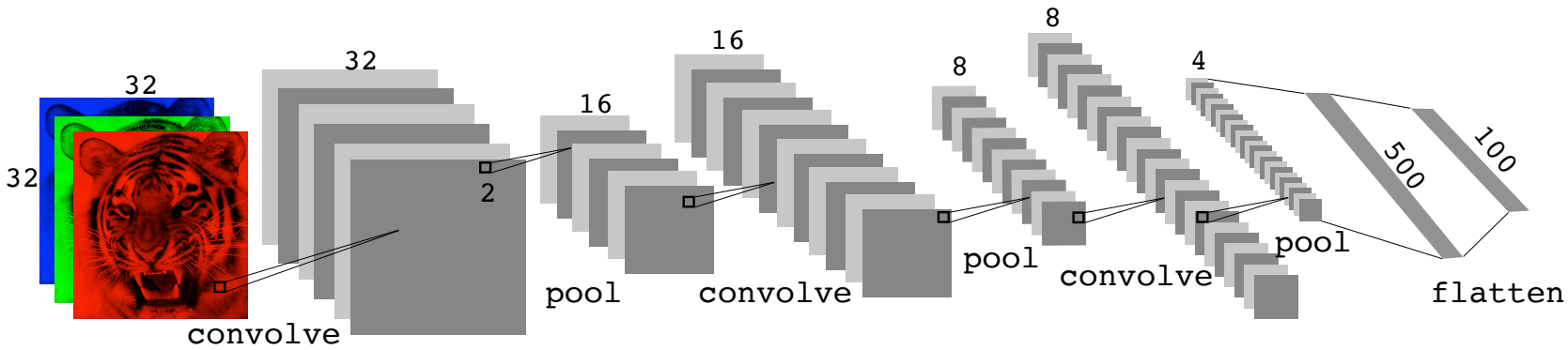
- Many convolve + pool layers.

Architecture of a CNN



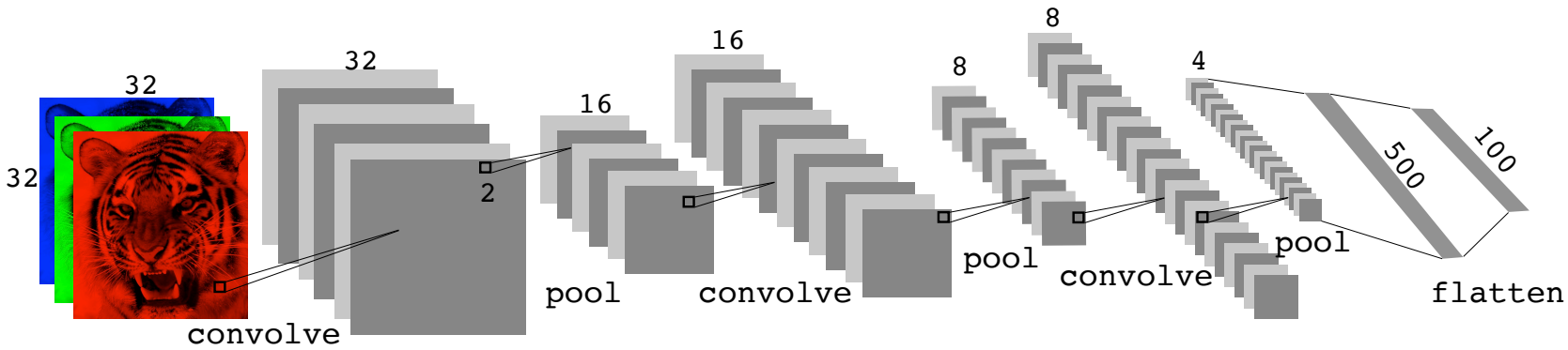
- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.

Architecture of a CNN



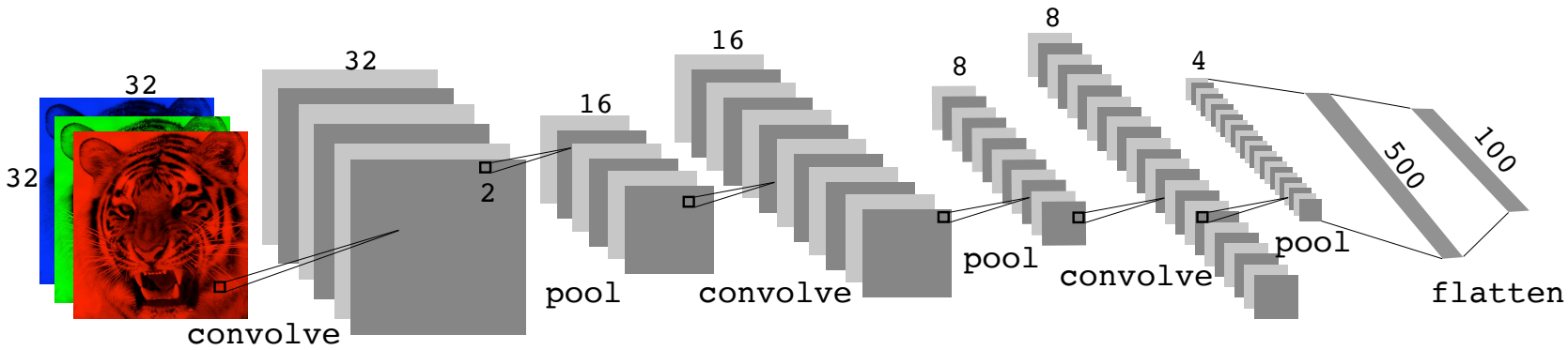
- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.
- Each filter creates a new channel in convolution layer.

Architecture of a CNN



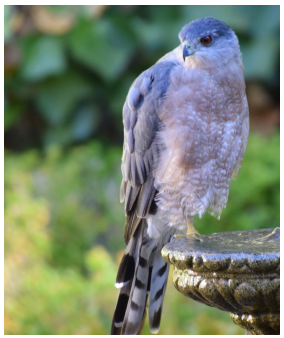
- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.
- Each filter creates a new channel in convolution layer.
- As pooling reduces size, the number of filters/channels is typically increased.

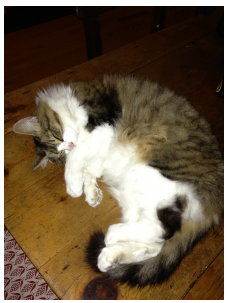
Architecture of a CNN



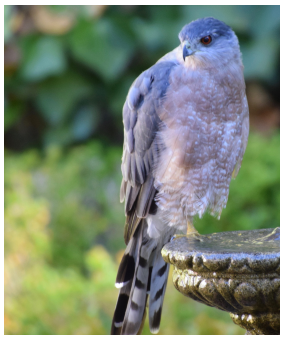
- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.
- Each filter creates a new channel in convolution layer.
- As pooling reduces size, the number of filters/channels is typically increased.
- Number of layers can be very large. E.g. `resnet50` trained on `imagenet` 1000-class image data base has 50 layers!

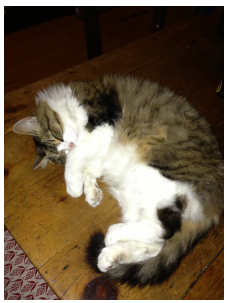
Using Pretrained Networks to Classify Images


Using Pretrained Networks to Classify Images


flamingo		Cooper's hawk		Cooper's hawk	
flamingo	0.83	kite (raptor)	0.60	fountain	0.35
spoonbill	0.17	great grey owl	0.09	nail	0.12
white stork	0.00	robin	0.06	hook	0.07
Lhasa Apso		**cat**		**Cape weaver**	
Tibetan terrier	0.56	Old English sheepdog	0.82	jacamar	0.28
Lhasa	0.32	Shih-Tzu	0.04	macaw	0.12
cocker spaniel	0.03	Persian cat	0.04	robin	0.12

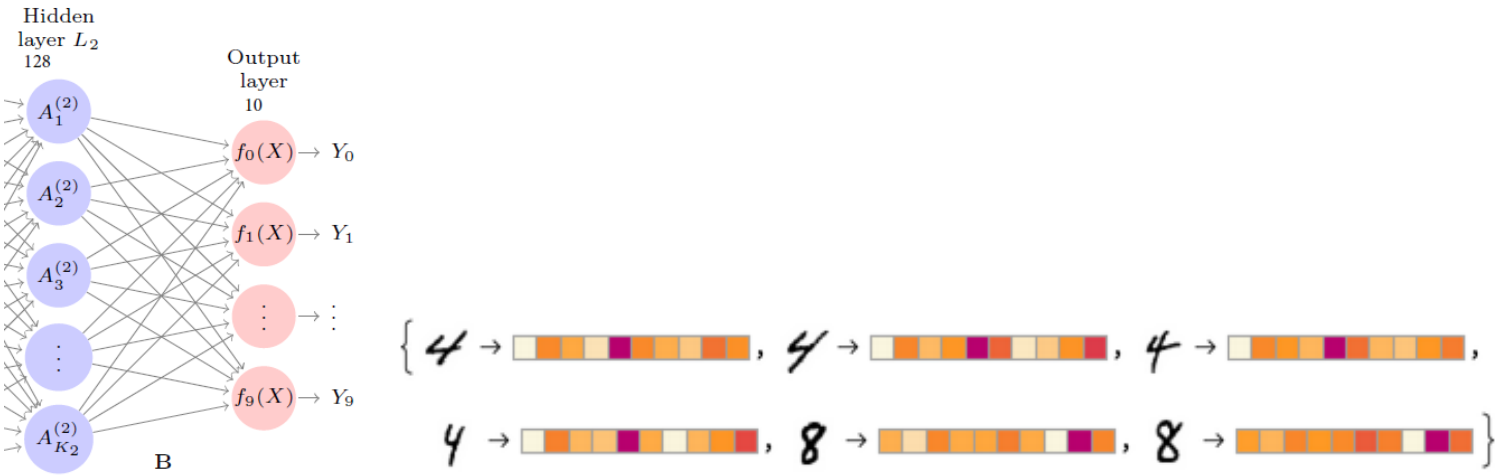
Here we use the 50-layer `resnet50` network trained on the 1000-class `imagenet` corpus to classify some photographs.

RNN and IMDB Reviews

- The document feature is a sequence of words $\{\mathcal{W}_\ell\}_1^L$. We typically truncate/pad the documents to the same number L of words (we use $L = 500$).
- Each word $\mathcal{W}_\ell$ is represented as a *one-hot encoded* binary vector X_ℓ (dummy variable) of length $10K$, with all zeros and a single one in the position for that word in the dictionary.
- This results in an extremely sparse feature representation, and would not work well.
- Instead we use a lower-dimensional pretrained *word embedding* matrix $\mathbf{E}$ ($m \times 10K$, next slide).
- This reduces the binary feature vector of length $10K$ to a real feature vector of dimension $m \ll 10K$ (e.g. m in the low hundreds.)

Embeddings for numbers

Last step in a neural network predicting numbers is a multinomial logit, the scores that are inputs to the logit can be used as embeddings:

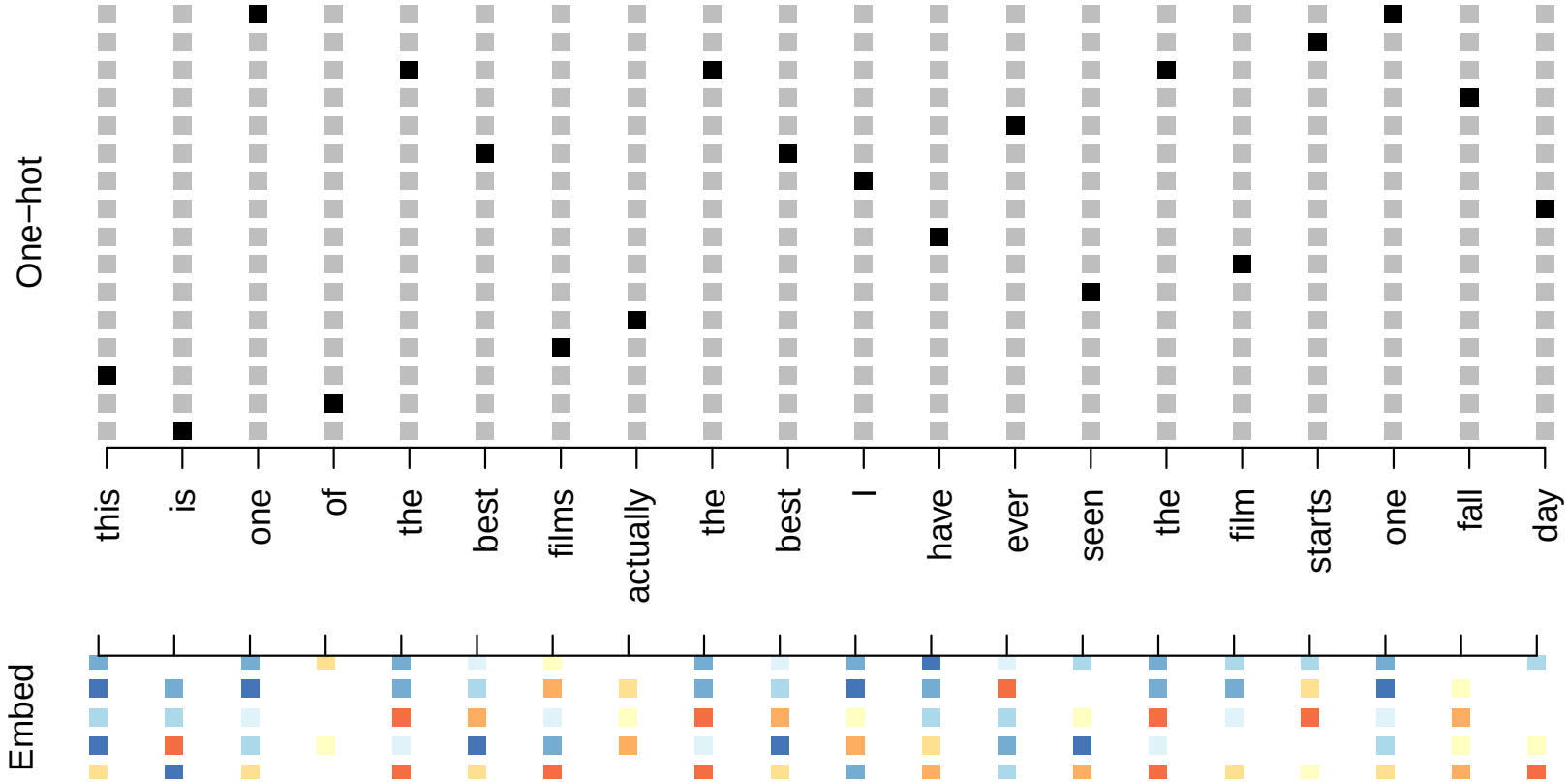


These embeddings are particular to a specific problem such as identiying numbers in a picture.

Embeddings for words

- Can we find a prediction problem for words,
  (similar to the numbers example above)?
- Words with similar meaning are used in similar contexts.
- Predict missing word based on surrounding words using a neural net.
  - Example: "the ___ cat" -> black, white, cute, etc., frequent.
- Use the hidden layers just before the prediction
  as embeddings that characterize the word.
  - word2vec, GloVe, BERT, GPT, ...

Word Embedding



this is one of the best films actually the best I have ever seen the film starts one fall day $\cdots$.

Embeddings are pretrained on very large corpora of documents, using methods similar to principal components. `word2vec` and `GloVe` are popular.

Transformers: attention + embedding

- Designed to handel sequential data, such as text.
- Pay attention (weight) more to certain parts of the input sequence. Process all elements in parallel.
- Popular transformer-based models:
  - BERT (Bidirectional Encoder Representations from Transformers),
  - GPT (Generative Pretrained Transformer), and
  - T5 (Text-To-Text Transfer Transformer

- Masked (encoder) bidirectional language models
  - Create contextualized representations of words in a sequence,
  - Trained by predicting masked tokens.
    - Attend to all words in the sequence (bidirectional).
- Generative (decoder) model
  - Typically used for text generation.
  - Trained guessing the next word.
    - Only attend to prior tokens.

- Simpler problem, guess the next letter.
  - Guess based on letter probability distribution, bigram, trigram etc.

0	on gxeeetowmt tsifhy ah aufnsoc ior oia itlt bnc tu ih uls
1	ri io os ot timumumoi gymyestit ate bshe abol viowr wotybeat mecho
2	wore hi usinallistin hia ale warou pothe of premetra bect upo pr
3	qual musin was witherins wil por vie surgedygua was suchinguary outheydays theresist
4	stud made yello adenced through theirs from cent intous wherefo proteined screa
5	special average vocab consumer market prepara injury trade consa usually speci utility

- Problem 40,000 words $\Rightarrow$ 60 trillion possible trigrams.
  - To few documents to learn probabilities $\Rightarrow$ need model.

GPT

- Neural network model based on transformers.
- Predict next word based on input sequence of previous words.
- LLM model 96 layers and 175 billion parameters.
- Fine tuned with feedback from people.

GPT prompting

- Uses entire conversation.
- Prompt engineering
  - Zero-shot prompting.
  - Few-shot prompting.

GPT Classification of individual speeches

Limit on input token string length: classify individual speeches (not all speeches at once).

```
In [14]: completion = openai.ChatCompletion.create(
    ...:   model="gpt-3.5-turbo",
    ...:   messages=[
    ...:     {"role": "system", "content": "You will be provided with a set speeches by a US senator,
and your task is to classify whether the senator is a Democrat or Republican."},
    ...:     {"role": "user", "content": """Mr. President, I am pleased that the Senate passed H.R.
4110, the Veterans Programs Enhancement Act of 1998. This measure strives to improve the services and
```

```
{
  "role": "assistant",
  "content": "Based on the content of the speeches, it is difficult to determine the political
affiliation of the senator. However, based on their support for the Veterans Programs Enhancement
Act and emphasis on conservation and protection of endangered species, it is likely that the
senator is a Democrat."
}
```

```
{
  "role": "assistant",
  "content": "Based on the speech provided, it is highly likely that the senator is a Democrat. Some indicators include the
senator's emphasis on improving services and benefits for veterans, support for extending eligibility for VA-guaranteed home
loans, and opposition to legislation that could weaken the Endangered Species Act. Additionally, the senator expresses concern
for Hawaii's unique ecosystem and the need for conservation efforts."
}
```

GPT Classification of individual speeches

- Questions for scientific implementation:
  - Answers a different question than "does language predict ideology"?
  - What is out-of-sample: given that the training data is the entire web?
  - Replicability

Using RoBertA embeddings for classification

1000 speeches (of 54000) takes 30 minutes.
To train (fine tune) model takes even longer.

```
##print some example rows
print(df.head(3))
```

```
      name state                                             speech  party
0  abraham    mi    . , during debate on final passage of the Om...      0
1  abraham    mi    . , I rise to register serious concern over ...      0
2  abraham    mi   . , I rise on the occasion of the Senate's pa...      0
```

```
[11] df.rename(columns={"party": "label"}, inplace=True)
```

```
[12] dfsmall=df.head(1000)
```

Classification using Huggingface models

```
df_clf_output = lt.classify_rows(dfsmall, on="speech", model="distilroberta-base")
df_clf_output
```

LLMs vs bag-of-words for speech prediction

- LLM outperform bag-of-words
- Take much longer are intransparent/harder to employ.
- Limit on input token string length:
  - classify individual speeches 1,000 speeches (of 54,000) took 30 mins.

Table 3. Performance on detecting support for President Trump in Tweets

Model	MCC	F1	Accuracy (%)	Sweep time	Hardware
Bag-of-words classifiers					
Logistic regression	0.51	0.67	77	1.53s	CPU
Random Forest	0.49	0.62	76	2 min. 2 s	CPU
SVM	0.49	0.68	76	37.8 s	CPU
Language models					
RoBERTa	0.60	0.75	81	46 min	GPU
BERTweet	0.63	0.77	83	44 min	GPU
PoliBERTweet	**0.68**	**0.80**	**85**	42 min	GPU

Bold numbers represent the best performing model on a given metric. Language models offer better classification for higher compute demands. Bag-of-words classifiers were trained with an exhaustive grid search and a Bayesian sweep across 10 models was used to train the language models.

Record Linkage

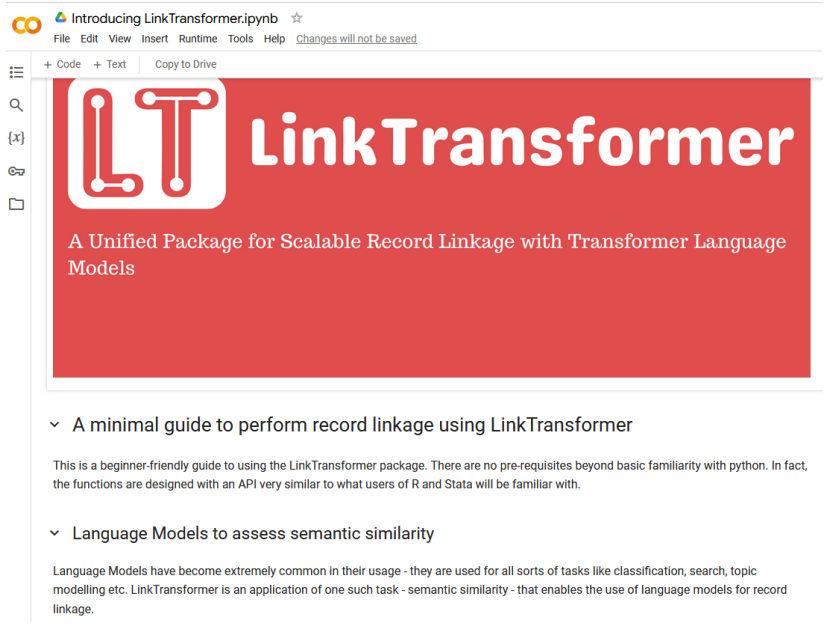
- Matching datasets: individuals, firms, product categories.
- Traditional approach Levenshtein edit distance
  - number of character insertions, deletions, and substitutions to convert one string into another.
- ML approach: use embeddings that capture meaning.
  - Encode text using transformer.
  - Match to closest neighbor in embedding space.
  - Distance uses semantic knowledge of pre-trained language model
  - ABC Corporation, ABC Co., and ABCC are semantically close.

Link Medical Subject Headings with International Classification of Diseases

- ICD codes from patient registers (45,461) with
- MeSH codes from science publications (27,815)
- Problem disease and MesH headings use different words for the same disease.
- I used the
  - LinkTransformer Python package (Arora and Dell, 2024).
  - Google Colab: a cloud service optimized for deep learning.
  - Models from the HuggingFace library.

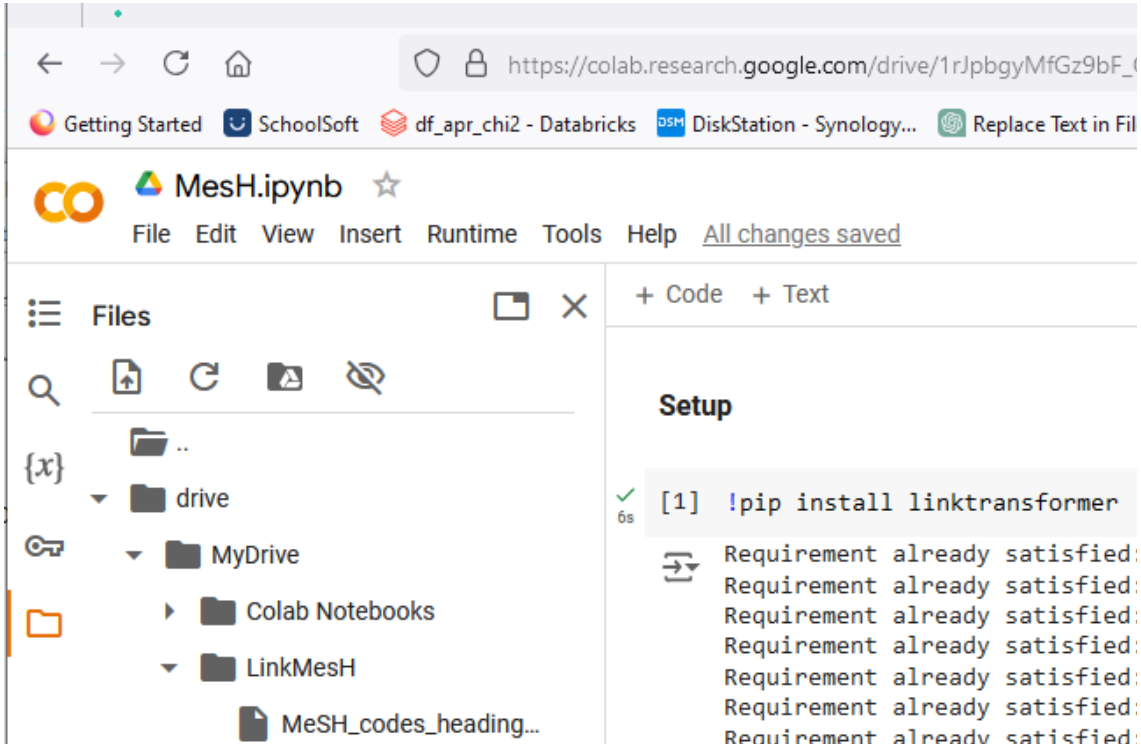
LinkTransformer (Arora and Dell, 2024)

- Tutorial Notebooks: https://linktransformer.github.io/



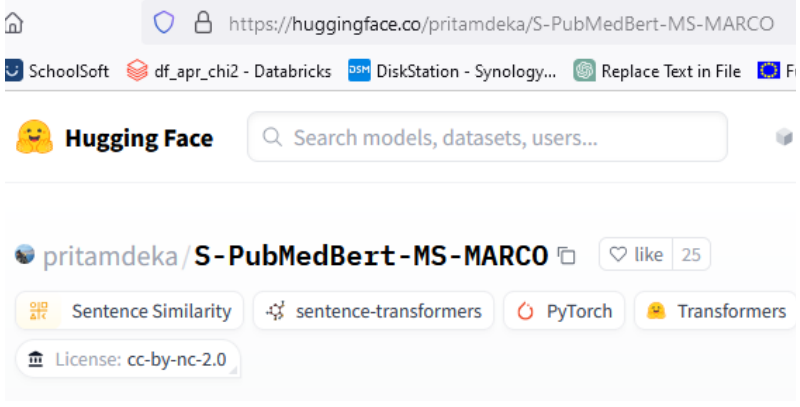
Cloud services optimized for deep learning

- Google Colab, or "Colaboratory",
  - Allows you to write and execute Python in your browser,
  - Access to GPUs free of charge
  - Access to your files through Google Docs.



HuggingFace

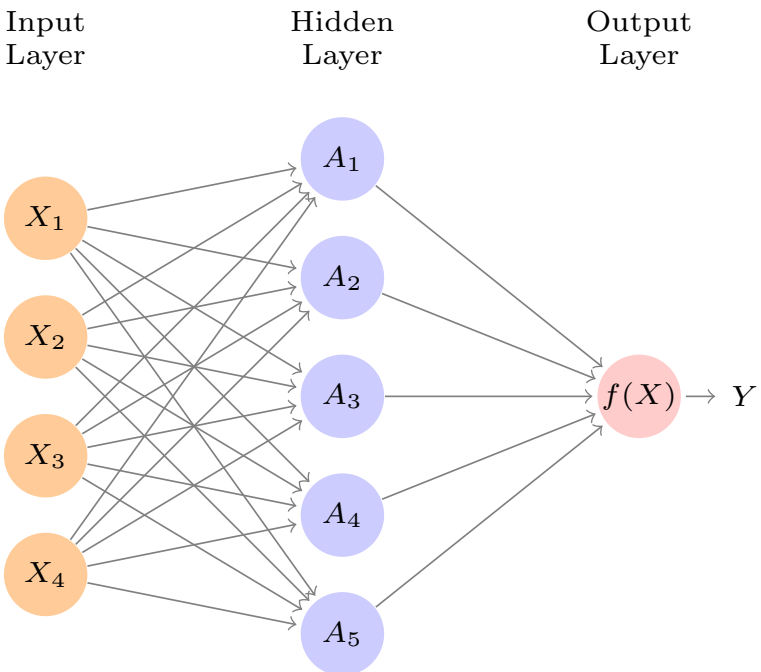
- Library of transformers models, data sets and demo apps.
- S-PubMedBert-MS-MARCO
  - Sentence-transformers model
    - Maps sentences & paragraphs to a 768 dimensional vector space.
  - BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext pretrained using
    - abstracts from PubMed and
    - full-text articles from PubMedCentral.
  - fine-tuned over the MS-MARCO dataset using sentence-transformers.



Result

heading	mesh_heading	score
Abdominal actinomycosis	Actinomycetales	.9519267
Abdominal and pelvic pain	Abdominal Muscles	.9248496
Abdominal aortic aneurysm, ruptured	Aneurysm, Ruptured	.9588628
Abdominal aortic aneurysm, without rupture	Aorta, Abdominal	.9388993
Abdominal aortic ectasia	Aorta, Abdominal	.9419204
Abdominal distension (gaseous)	Abdominal Cavity	.919881
Abdominal migraine	Abdominal Muscles	.9208636

Fitting Neural Networks

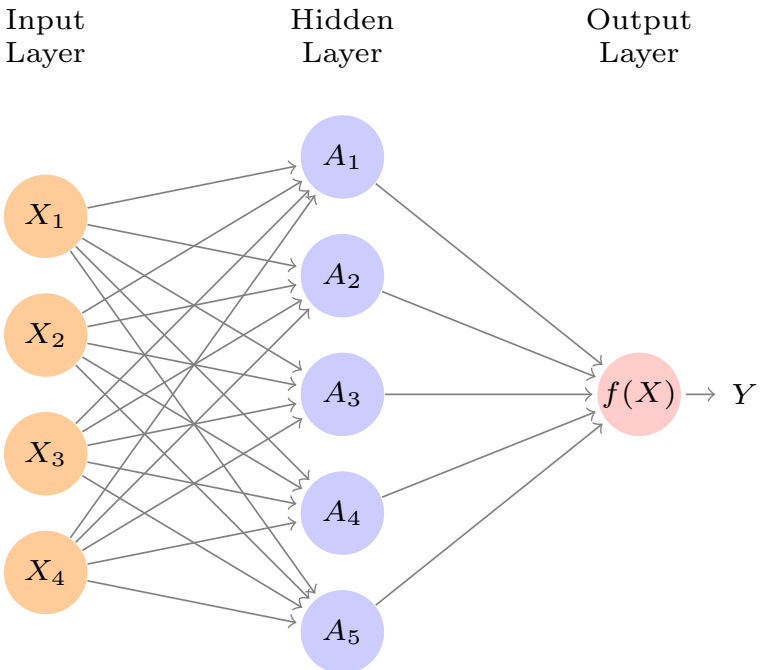


$$\underset{\{w_k\}_1^K,\,\beta}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big).$$

Fitting Neural Networks



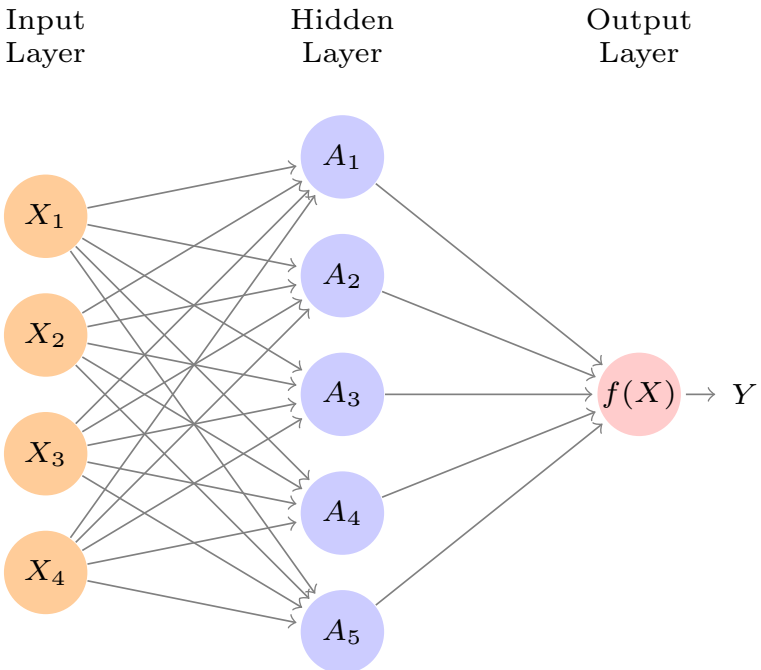
$$\underset{\{w_k\}_1^K,\,\beta}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big).$$

This problem is difficult because the objective is *non-convex*.

Fitting Neural Networks



$$\underset{\{w_k\}_1^K,\,\beta}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where

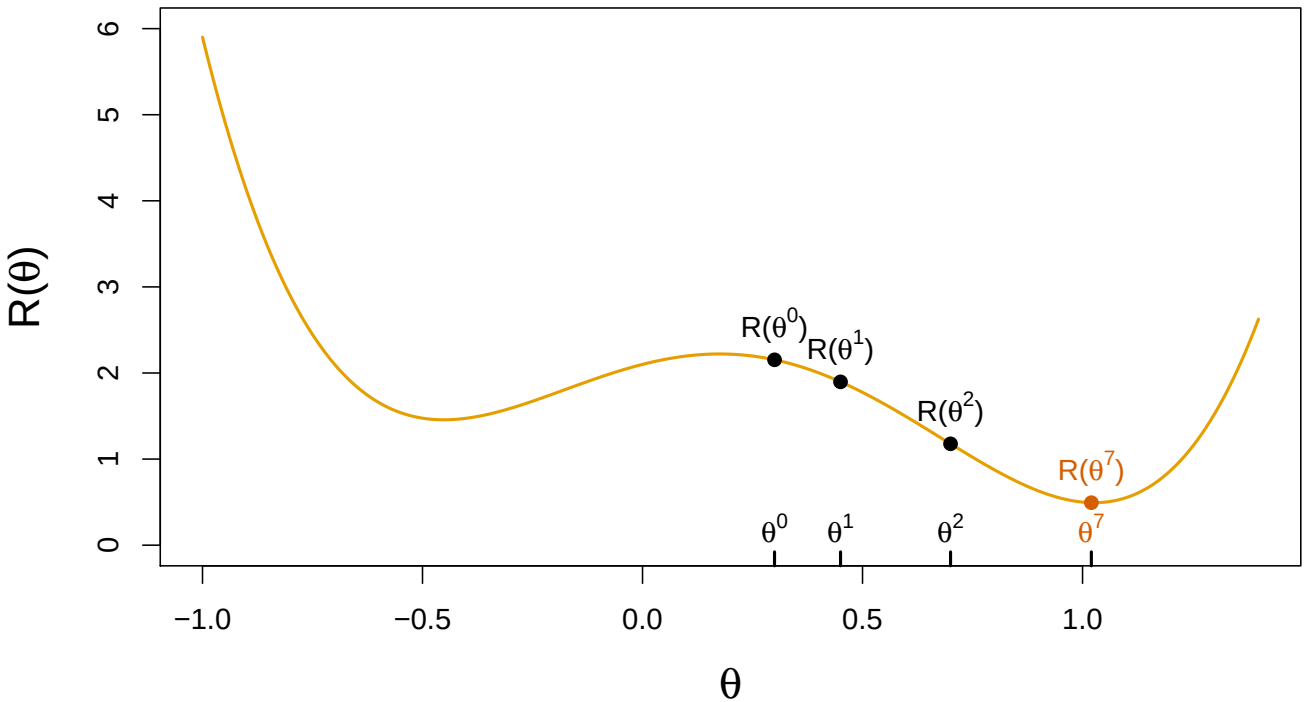
$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big).$$

This problem is difficult because the objective is *non-convex*.

Despite this, effective algorithms have evolved that can optimize complex neural network problems efficiently.

Non Convex Functions and Gradient Descent

Let $R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$ with $\theta = (\{w_k\}_1^K,\ \beta)$.



1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.
2. Iterate until the objective $R(\theta)$ fails to decrease:
   (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ *reduces* the objective; i.e. $R(\theta^{t+1}) < R(\theta^t)$.
   (b) Set $t \leftarrow t + 1$.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.
- An *epoch* is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e. $60K/128 \approx 469$ for `MNIST`.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.
- An *epoch* is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e. $60K/128 \approx 469$ for `MNIST`.
- *Regularization*. Ridge and lasso regularization can be used to shrink the weights at each layer. Two other popular forms of regularization are *dropout* and *augmentation*, discussed next.

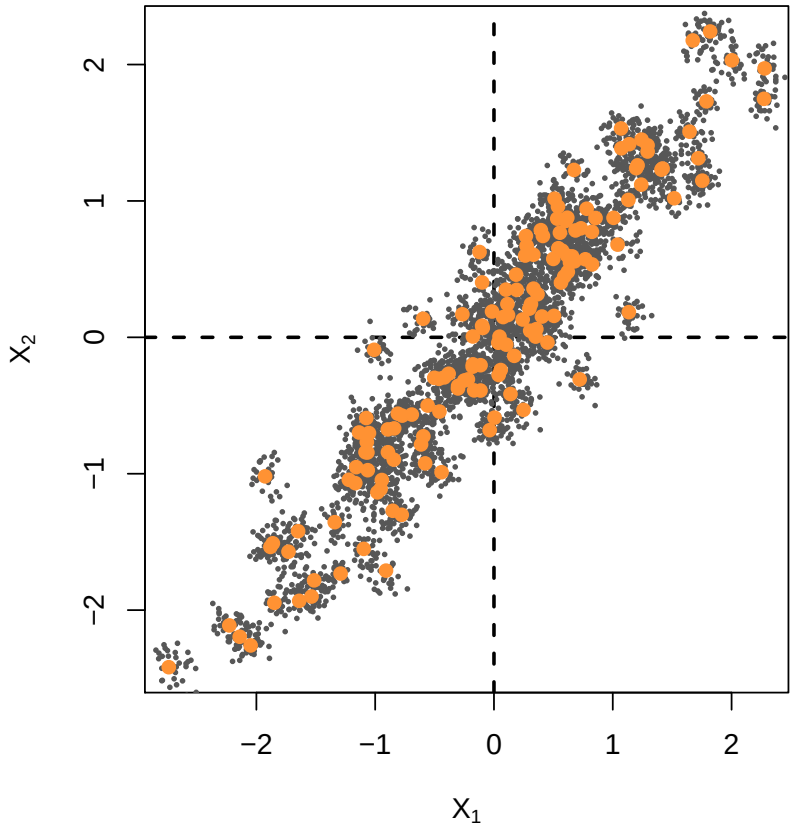
Dropout Learning

- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.

Dropout Learning

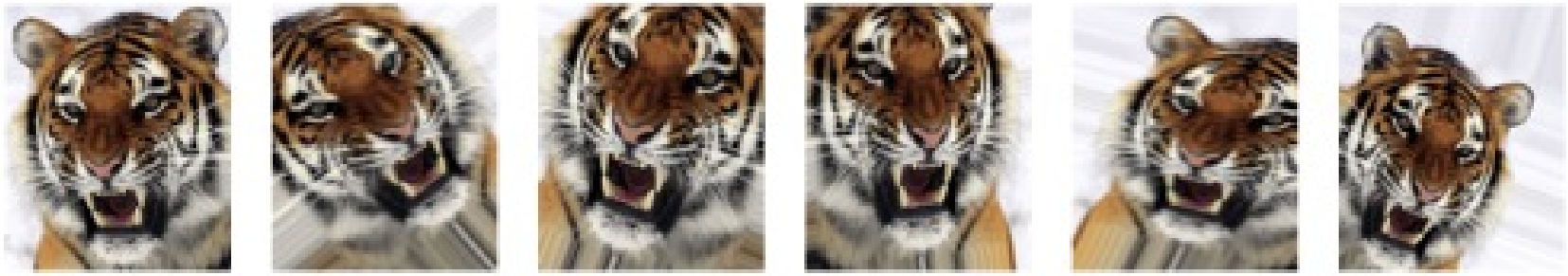
- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.
- In simple scenarios like linear regression, a version of this process can be shown to be equivalent to ridge regularization.
- As in ridge, the other units *stand in* for those temporarily removed, and their weights are drawn closer together.
- Similar to randomly omitting variables when growing trees in random forests (Chapter 8).

Ridge and Data Augmentation



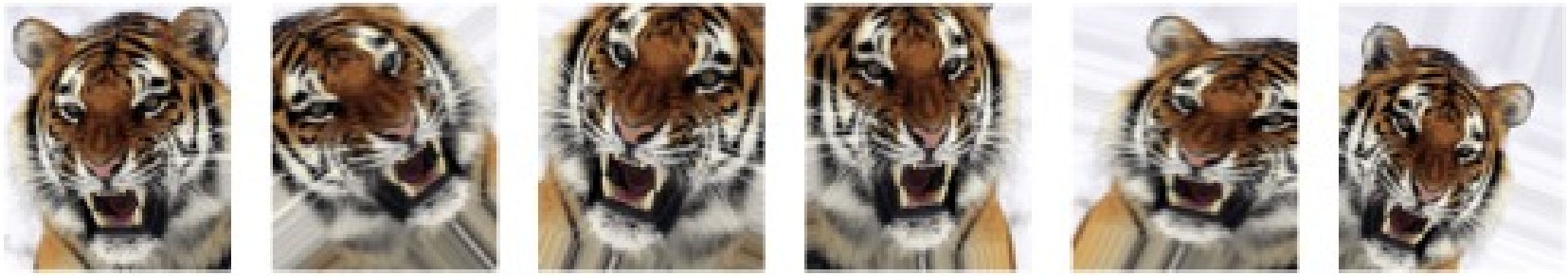
- Make many copies of each (x_i, y_i) and add a small amount of Gaussian noise to the x_i — a little cloud around each observation — but *leave the copies of y_i alone!*
- This makes the fit robust to small perturbations in x_i, and is equivalent to ridge regularization in an OLS setting.

Data Augmentation on the Fly



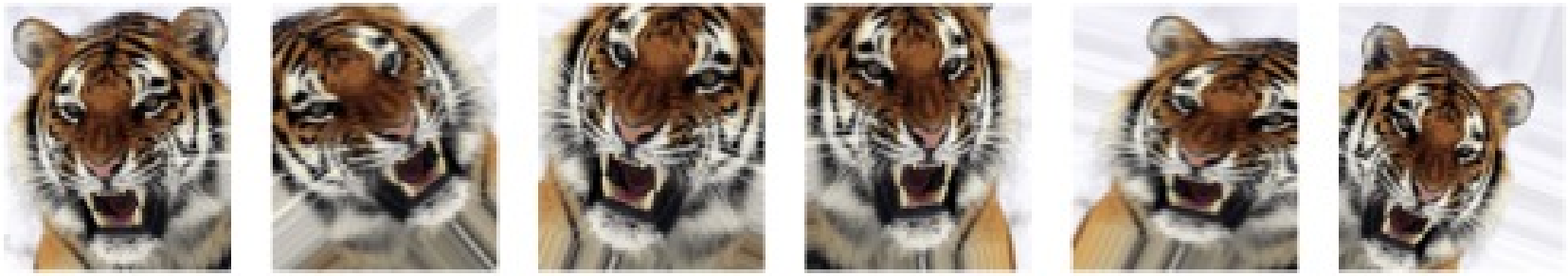
- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.

Data Augmentation on the Fly



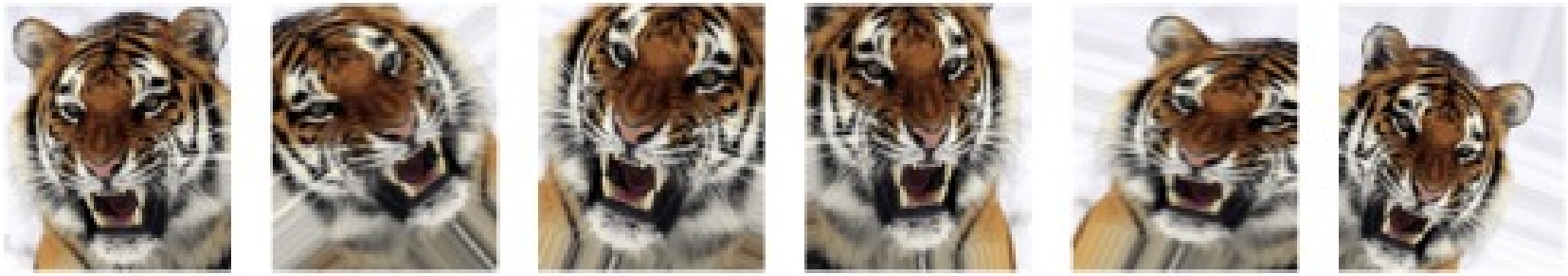
- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural transformations are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.

Data Augmentation on the Fly



- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural transformations are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.
- The label is left unchanged — in each case still `tiger`.

Data Augmentation on the Fly



- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural transformations are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.
- The label is left unchanged — in each case still `tiger`.
- Improves performance of CNN and is similar to ridge.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running stochastic gradient descent till zero training error often gives good out-of-sample error.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running stochastic gradient descent till zero training error often gives good out-of-sample error.
- Increasing the number of units or layers and again training till zero error sometimes gives *even better* out-of-sample error.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running stochastic gradient descent till zero training error often gives good out-of-sample error.
- Increasing the number of units or layers and again training till zero error sometimes gives *even better* out-of-sample error.

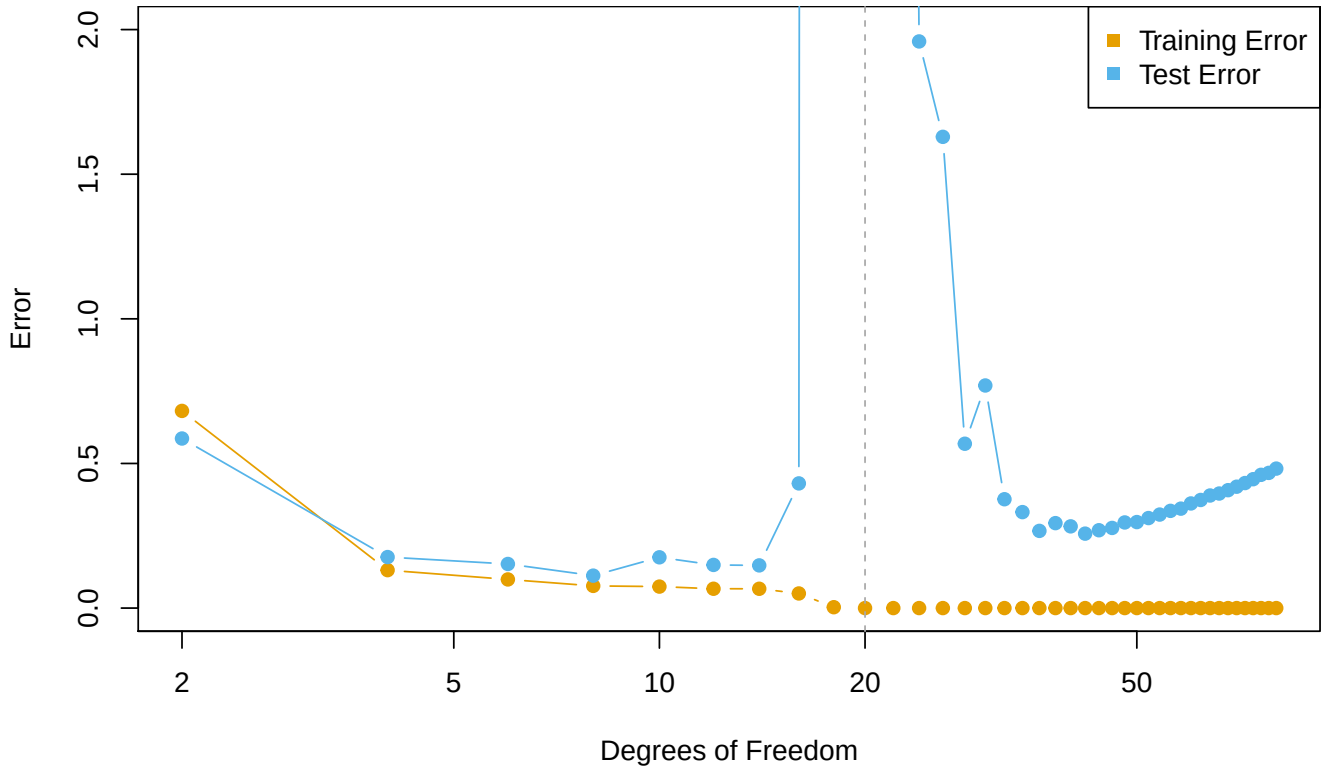
What happened to overfitting and the usual bias-variance trade-off?

Belkin, Hsu, Ma and Mandal (arXiv 2018) *Reconciling Modern Machine Learning and the Bias-Variance Trade-off.*

Simulation

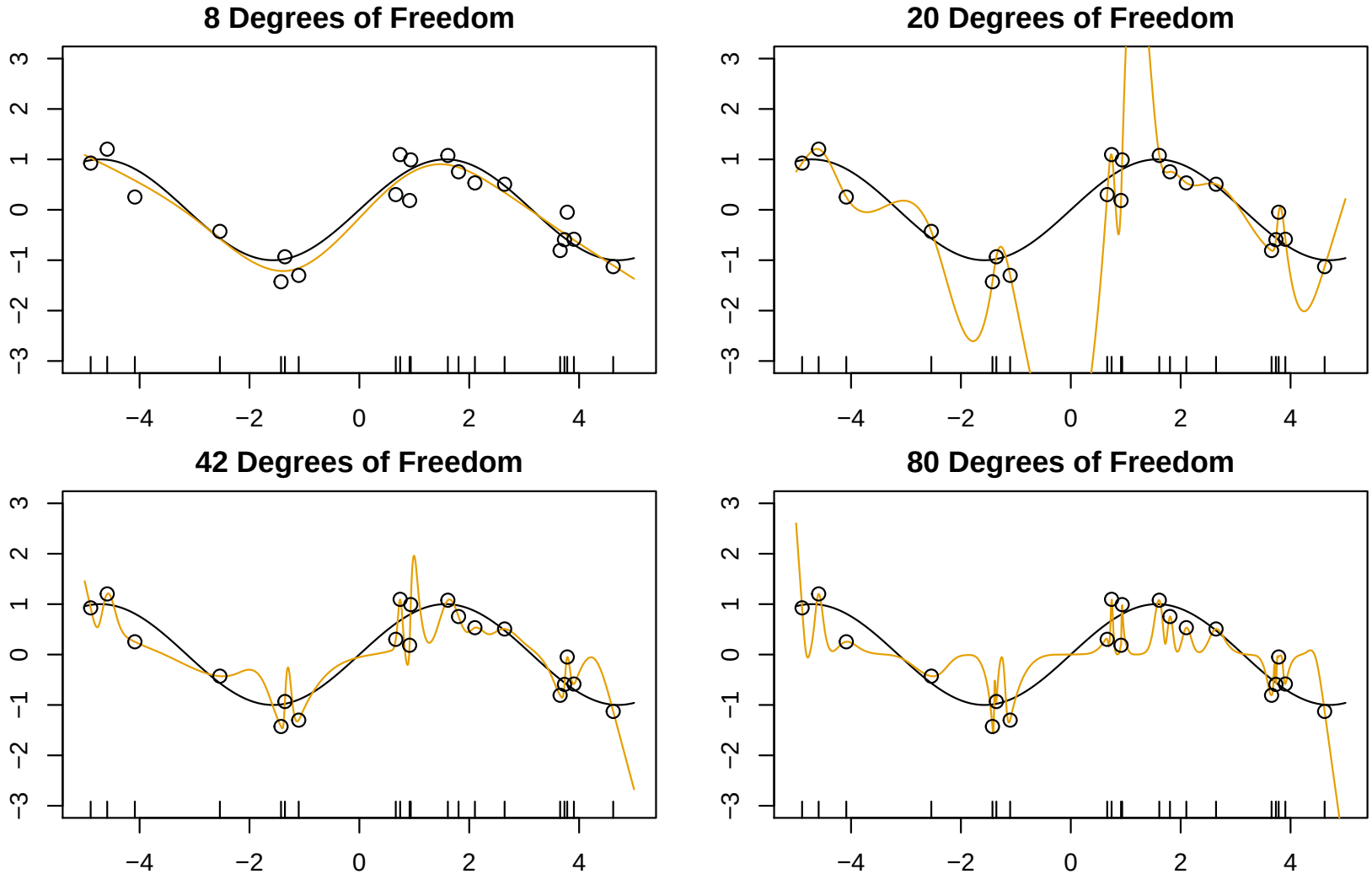
- $y = \sin(x) + \varepsilon$ with $x \sim U[-5, 5]$ and ε Gaussian with S.D. $= 0.3$.
- Training set $n = 20$, test set very large (10K).
- We fit a natural spline to the data (Section 7.4) with d degrees of freedom — i.e. a linear regression onto d basis functions: $\hat{y}_i = \hat{\beta}_1 N_1(x_i) + \hat{\beta}_2 N_2(x_i) + \cdots + \hat{\beta}_d N_d(x_i)$.
- When $d = 20$ we fit the training data exactly, and get all residuals equal to zero.
- When $d > 20$, we still fit the data exactly, but the solution is not unique. Among the zero-residual solutions, we pick the one with *minimum norm* — i.e. the zero-residual solution with smallest $\sum_{j=1}^{d} \hat{\beta}_j^2$.

The Double-Descent Error Curve



- When $d \leq 20$, model is OLS, and we see usual bias-variance trade-off
- When $d > 20$, we revert to minimum-norm. As d increases above 20, $\sum_{j=1}^{d} \hat{\beta}_j^2$ *decreases* since it is easier to achieve zero error, and hence less wiggly solutions.

Less Wiggly Solutions



To achieve a zero-residual solution with $d = 20$ is a real stretch!
Easier for larger d.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.
- Stochastic gradient *flow* — i.e. the entire path of SGD solutions — is somewhat similar to ridge path.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.
- Stochastic gradient *flow* — i.e. the entire path of SGD solutions — is somewhat similar to ridge path.
- By analogy, deep and wide neural networks fit by SGD down to zero training error often give good solutions that generalize well.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.
- Stochastic gradient *flow* — i.e. the entire path of SGD solutions — is somewhat similar to ridge path.
- By analogy, deep and wide neural networks fit by SGD down to zero training error often give good solutions that generalize well.
- In particular cases with *high signal-to-noise ratio* — e.g. image recognition — are less prone to overfitting; the zero-error solution is mostly signal!

Software

- Wonderful software available for neural networks and deep learning. `Tensorflow` from Google and `PyTorch` from Facebook. Both are `Python` packages.
- In the Chapter 10 lab we demonstrate `tensorflow` and `keras` packages in `R`, which interface to Python. See textbook and online resources for `Rmarkdown` and `Jupyter` notebooks for these and all labs for the second edition of ISLR book.
- The `torch` package in R is available as well, and implements the `PyTorch` dialect. The Chapter 10 lab will be available in this dialect as well; watch the resources page at `www.statlearning.com`.

Problem set

This problem set is to link names data sets using the LinkTransformer Python package. The data sets and a template Jupyter Notebook will be supplied. The problem set is to create a virtual Python environment where this Notebook is executed and then to modify the Notebook.

1. Decide whether you want to work in VS Code, Google Colab or Jupyter Lab.
2. Set up a virtual conda environment based on Python 3.11.
3. Upload the data
4. Run the Jupyter Notebook Link_Records_with_Linktransformer.ipynb
5. Go to HuggingFace and find an alternative language model that is trained to detect similarity in company names. Use this model to match company names in the command

df_lm_matched = lt.merge(df2, df1, merge_type='1:m', on="CompanyName", model="...", left_on=None, right_on=None).