CS/DS 541: Class 8

Jacob Whitehill

Why “deep” learning?

Why “deep” learning?

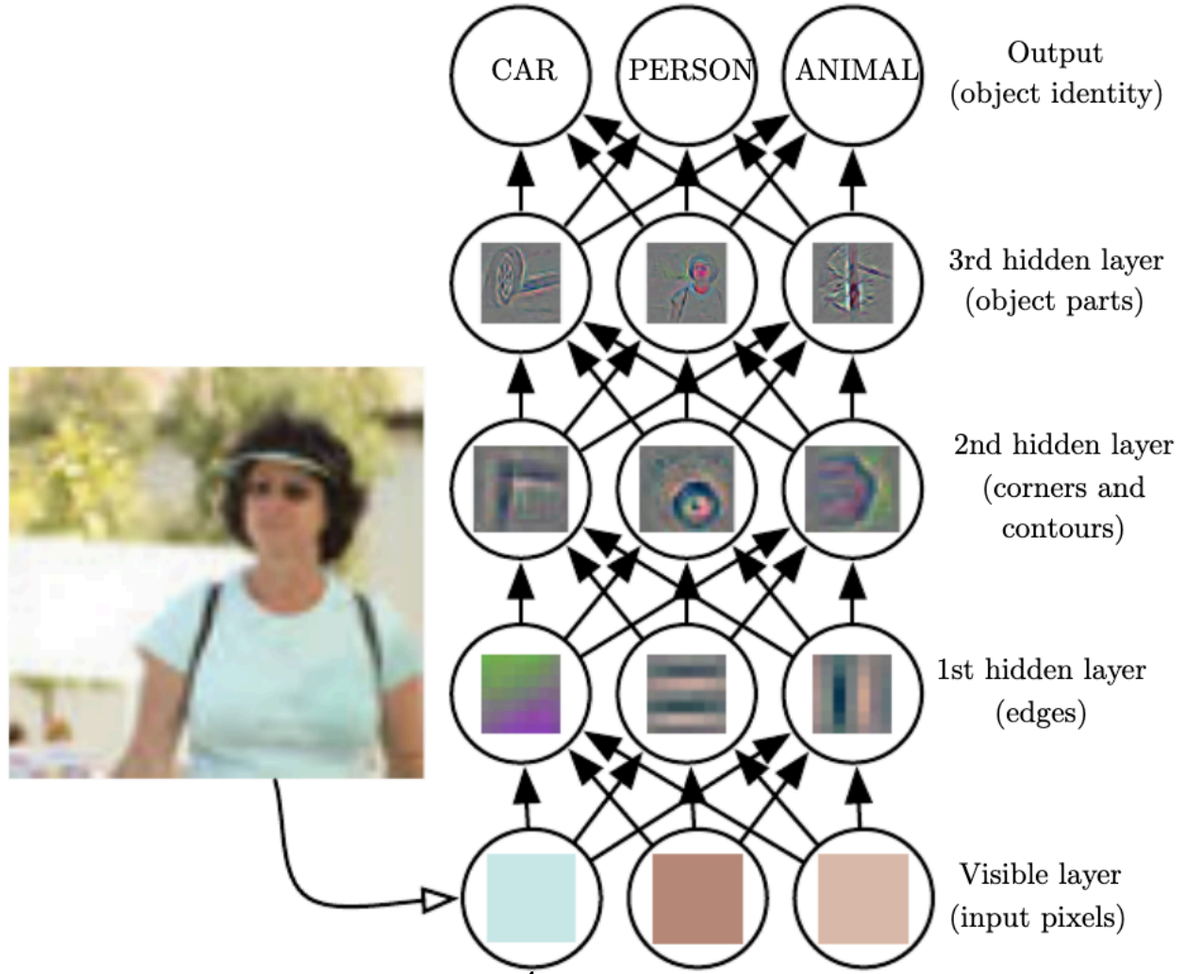
- One of the reasons for the resurgence of NNs around 2010 is that increasing the depth is very powerful.

- There is some theory, and an abundance of empirical results, that deeper networks are more powerful (can represent more complicated functions) and more accurate than shallow networks.

- Why?

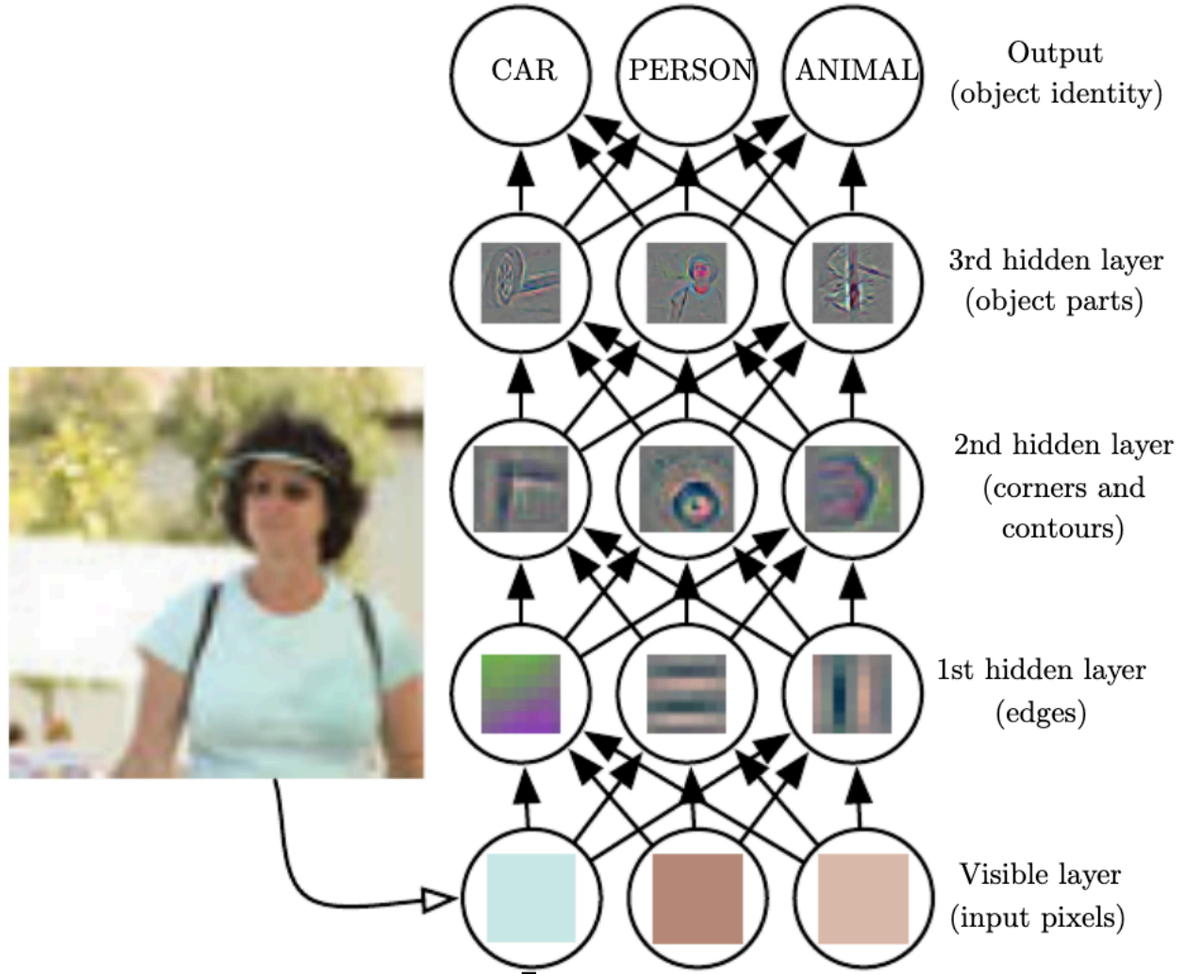
Why “deep” learning?

- One explanation for the success of deep NNs is that each layer can represent **increasingly abstract representations** of the input (from Zeiler & Fergus 2014):



Why “deep” learning?

- The hidden units can represent the content of the input in a compact way.



Demo

- As a simple illustration, let's consider two feed-forward NNs with the **same number of weights (60)**:
  - 3 layers with dimensions: (2, 20, 1)
  - 6 layers with dimensions: (2, 4, 4, 4, 4, 1)
- If we initialize their weights randomly, and use a non-linear activation function (`abs` in this case), then what kinds of functions can they represent?

Difficulty in deep learning

- Training very deep neural networks has traditionally been very difficult for two main reasons:
  - **Vanishing gradients**: gradients of weights tends to 0 as depth increases. => no learning
  - **Exploding gradient**: gradients of weights tends to infinity as depth increases. => requires small learning rate.

Difficulty in deep learning

- Since ~2008, DL researchers have:
  - Harnessed faster hardware to achieve good accuracy more quickly.
  - Found better architectures and training algorithms that prevent the gradients from behaving “badly”.

Practical suggestions

Tackle the problem based on your past experience

- Think of the machine learning problem you have solved that is *most similar* to the one you are tackling now, e.g.:
  - Same task (object detection, semantic segmentation, speaker diarization, etc).
  - Same dataset (MNIST, ImageNet, Faces in the Wild, etc.)
- How did you solve it?
  - NN design
  - Training procedures
  - Hyperparameters
- Start with the approach you used before, make sensible adjustments, and start training.

Start small

- Debug your code on a smaller version of the problem:
  - Subset of data (e.g., just 10K images instead of 50K)
  - Subset of classes (e.g., just 2 MNIST classes instead of 10)

- Advantages:
  - Less time for initialization.
  - Less time for training.
  - Fewer variables to examine during forward/back-propagation

Start simple

- Until you gain confidence & experience, train a simple model first:
    - They’re often faster to train and easier to debug than more powerful models.

Start simple

- Until you gain confidence & experience, train a simple model first:
    - They’re often faster to train and easier to debug than more powerful models.
- Make sure your model’s accuracy is above chance:
    - Take the prior class probabilities into account! (If the classes are 90/10, then the baseline rate for guessing the majority class is 90%.)
    - Make sure your model is not always predicting the dominant class.

But not too simple

- Sometimes a complex neural network is necessary to solve a problem with high accuracy.

- For instance, a 3-layer NN may not have enough representational power to analyze complex images.

- You might need to use a standard architecture (e.g., ResNet) — but use one of the simpler versions (e.g., 50 instead of 100 layers).

Start small & simple

- Try to find a model (and hyper-parameters) whose training loss decreases *smoothly*.

- Afterwards, increase the size of the training set and model complexity.

Regularization

- If there is a large divergence between training accuracy and testing accuracy (i.e., overfitting), then try regularizing the model:
  - Increasing L_1, L_2 regularization strength.
  - Adding/increasing dropout (for NNs).
  - Reducing number of training epochs (for NNs).
  - Synthesizing more training examples with label-preserving transformations (geometric & noise-based).

Hyper-parameter optimization

- Try a variety of hyper-parameters:
- Find a few values manually that seem to work; use these as a guide to pick a reasonable range (e.g., for learning rate, 1e-5 to 1e0, spaced logarithmically).

- Be disciplined about optimizing parameters on a validation set, not the test set!

Hyper-parameter optimization

- Try a variety of hyper-parameters:
- When you have intuition, then it's sometimes worthwhile to watch the loss evolve over time.
- When you do not have intuition, then automate the process and get on with your life — do not succumb to the temptation to watch learning curves like a movie.

Normalization

- It can be helpful to put every feature onto the same scale.

- In particular, the scale can interact with the L_2 regularization strength.

Normalization: example

- Suppose you are predicting tomorrow's temperature based on (1) today's temperature and (2) wind speed.

- Suppose we measure temperature in Kelvin and wind speed in km/h.

- Suppose the optimal weights w_1, w_2 for these two features, for L_2-regularized linear regression, are 1 and 2, i.e.:

  - $\hat{y} = w_1 t + w_2 s$ (t = today's temp, s = today's wind speed)
    $\hat{y} = 1*t + 2*s$

Normalization: example

- Now, suppose we change the units for wind speed from km/h to m/s.
  - E.g., 18 km/h = 5 m/s **Numerical values reduced by 3.6x**

- If we don't adjust our model weights w_1, w_2, then our predictions will be wrong:

  - $\hat{y} = 1^*t + 2^*s$
    $\hat{y}(4, 18) = 4 + 36 = 40$ **km/h**
    $\hat{y}(4, 5) = 4 + 10 = 14$ **m/s**

Normalization: example

- Because the numerical values of the wind speed were reduced by factor of 3.6, the corresponding weight w_2 must compensate by increasing by 3.6x, i.e.:
  - $\hat{y} = w_1 t + \tilde{w}_2 s$ (t = today's temp, s = today's wind speed)
    $\hat{y} = 1{*}t + 3.6{*}2{*}s$

- Without regularization, the training procedure (e.g., minimize f_{MSE}) will account for the change-of-scale seamlessly, i.e.:

$$\arg\min_{\tilde{w}_2} f_{\mathrm{MSE}}^{\mathrm{m/s}}(\cdot) \quad \textbf{= 3.6 *} \quad \arg\min_{w_2} f_{\mathrm{MSE}}^{\mathrm{km/h}}(\cdot)$$

Normalization: example

- But with L_2 regularization, the issue is more complicated:

$$\arg\min_{\tilde{w}_2} \left[f_{\text{MSE}}^{\text{km/h}}(\cdot) + \frac{1}{2} w_2^2 \right]$$

- The regularization term "discourages" w_2 from growing too big:
  - When we rescale from km/h to m/s, the L_2 term prevents the weight w_2 from compensating exactly.

Exercise

- Will the regularization effect of L_2 normalization increase or decrease when we change from km/h to m/s?

$$\arg\min_{\tilde{w}_2} \left[f_{\text{MSE}}^{\text{km/h}}(\cdot) + \frac{1}{2} w_2^2 \right]$$

Data normalization

Data normalization

- In many ML settings (not just DL), it is important or useful to normalize the input data to improve accuracy or ease of training.

- Common normalization methods include:
  - Mapping *each* feature into a fixed range (e.g., [0,1], [-1,1]).
  - Z-scoring *each* feature (so that the mean and stddev are 0 and 1, respectively).
  - Whitening *all* features jointly (to have 0 mean and *I*-covariance).

Data normalization

- There are (at least) two ways to achieve this.
- Strategy 1: Learn transformation on training data:
  - Compute the normalization parameters on the training set; save them.
  - Apply the normalization to the training data *and* each of the testing data.

Data normalization

- There are (at least) two ways to achieve this.

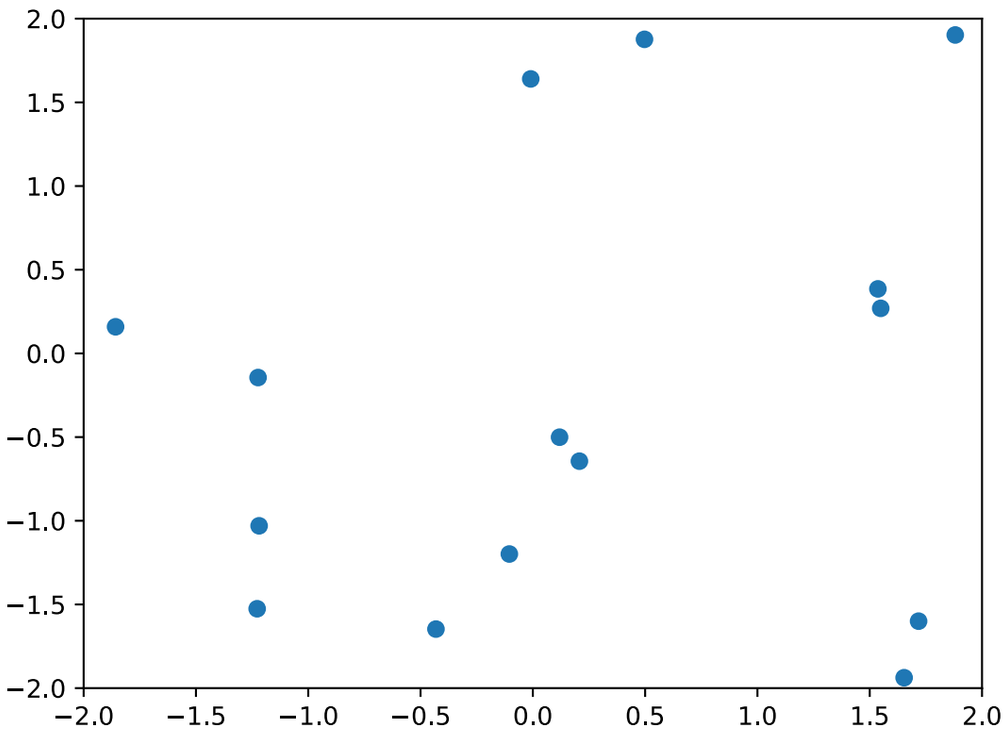
- Strategy 2: Learn transformations for training and testing data *separately*:

  - Compute the normalization parameters on the *training* set, and apply it to each of the *training* data.

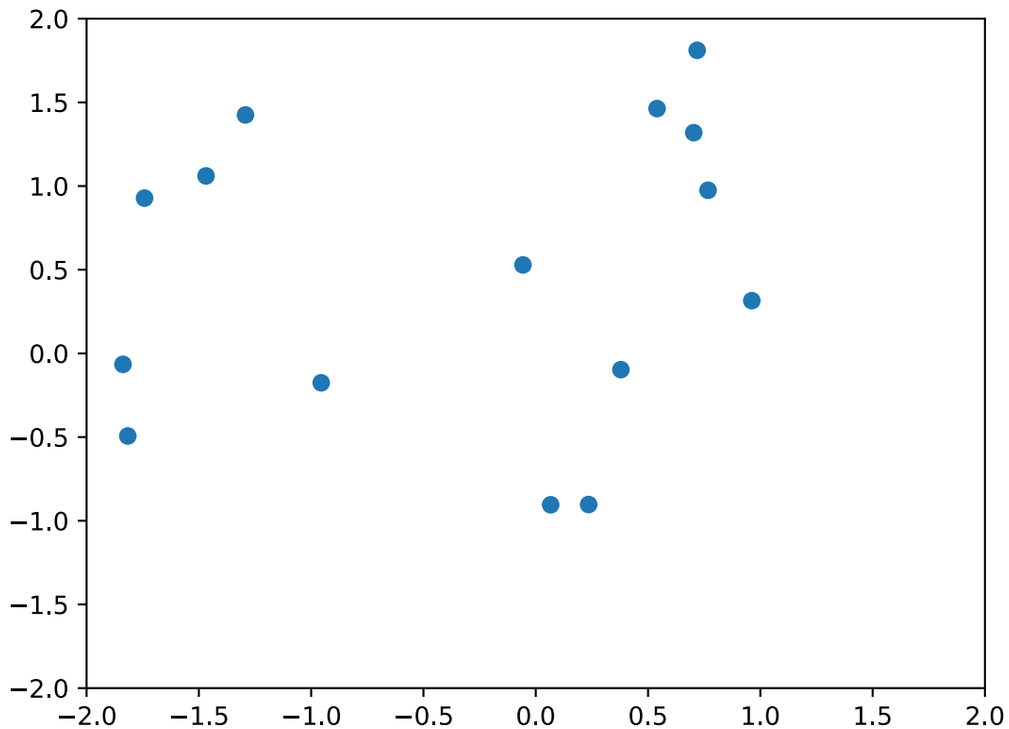
  - Compute the normalization parameters on the *testing* set, and apply it to each of the *testing* data.

Strategy 1

Unnormalized training data

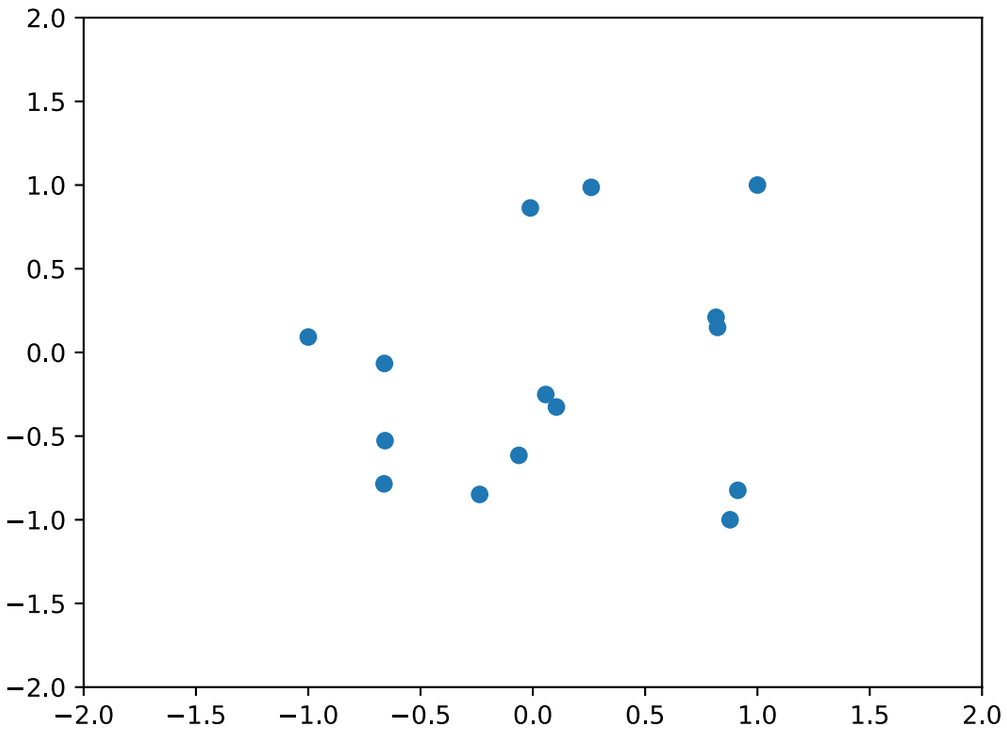


Unnormalized testing data

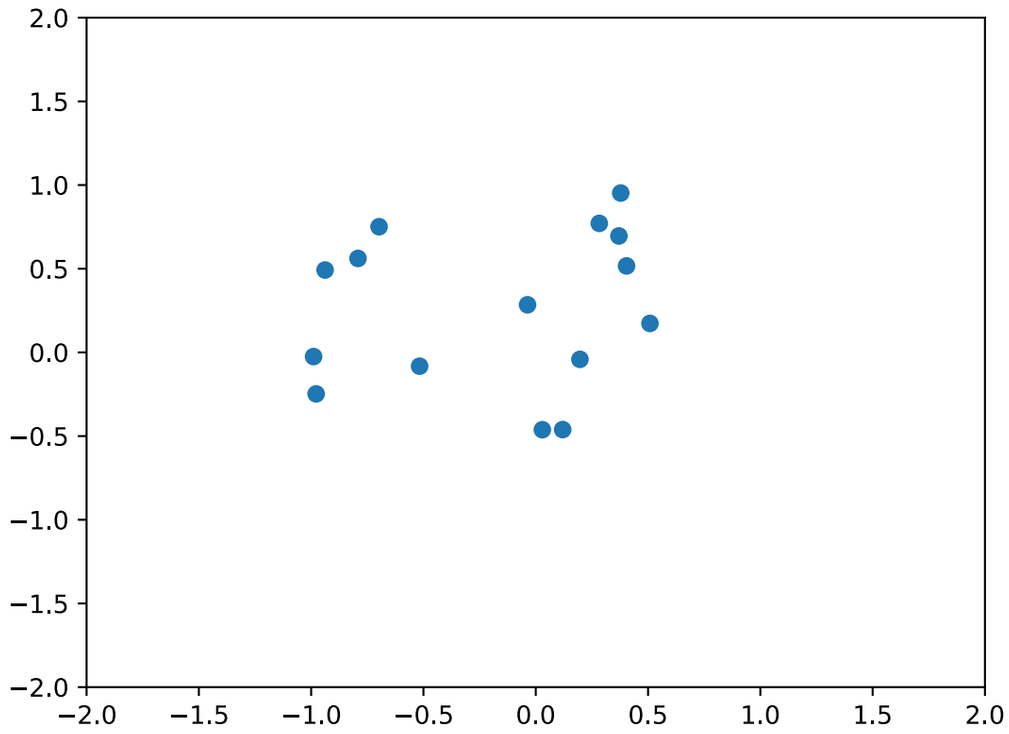


Strategy 1

Normalized training data

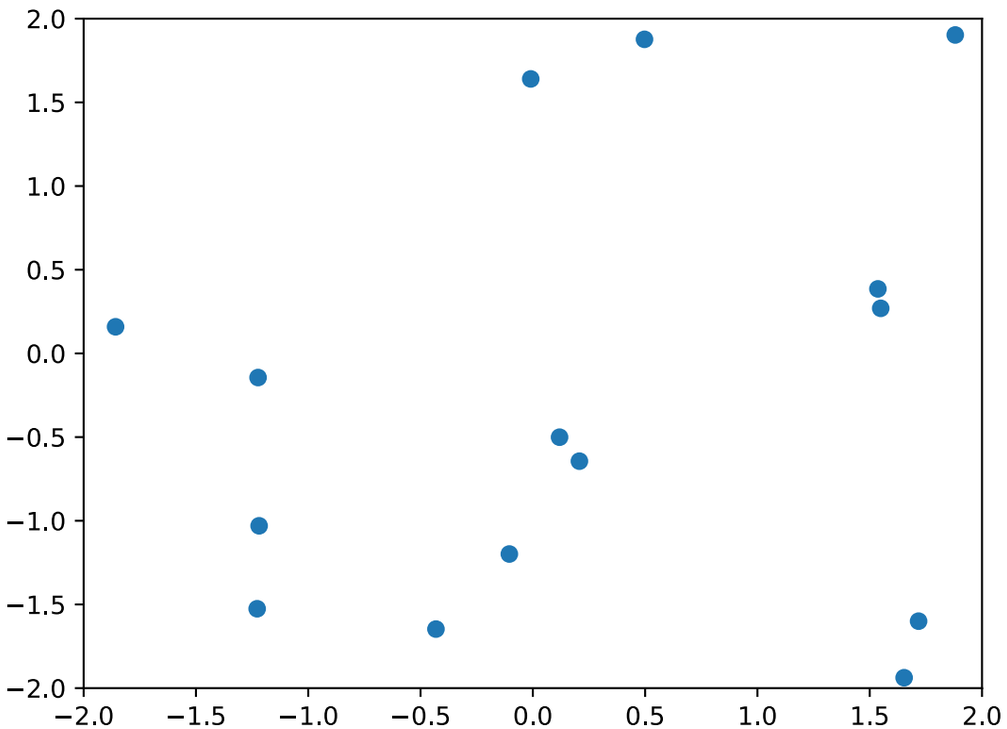


Normalized testing data

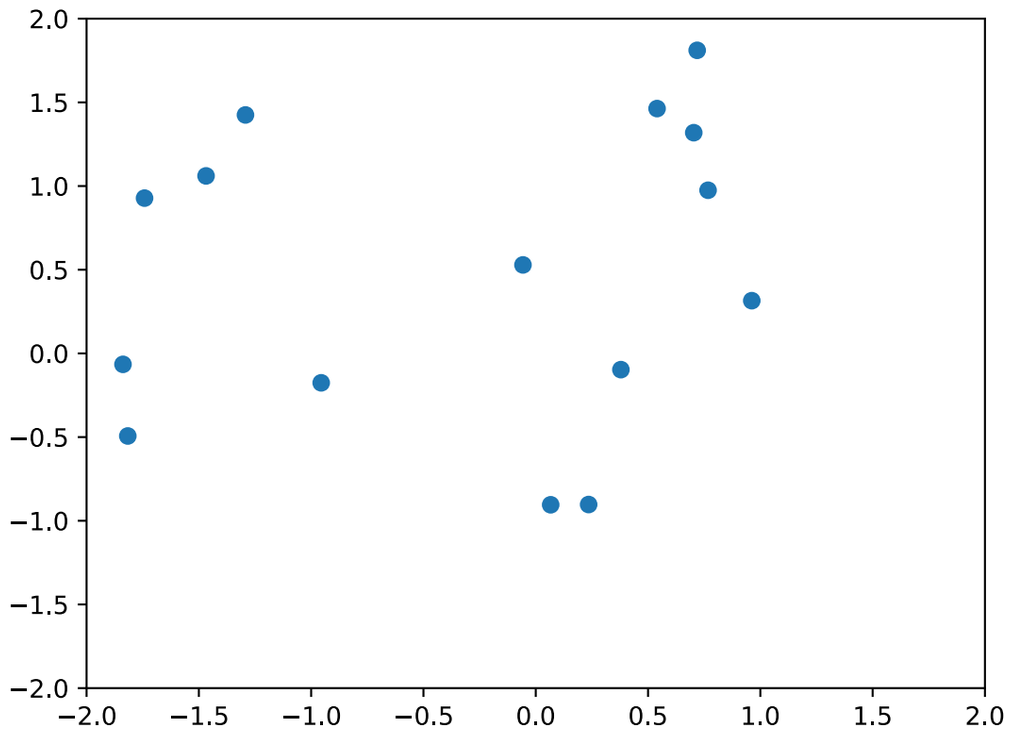


Strategy 2

Unnormalized training data

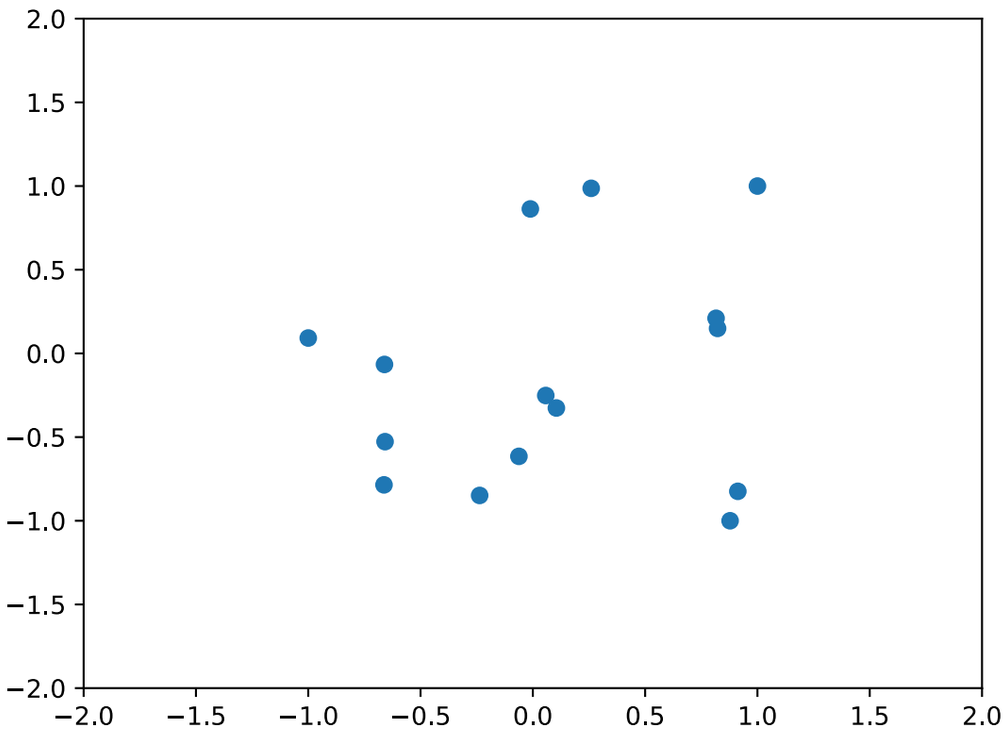


Unnormalized testing data

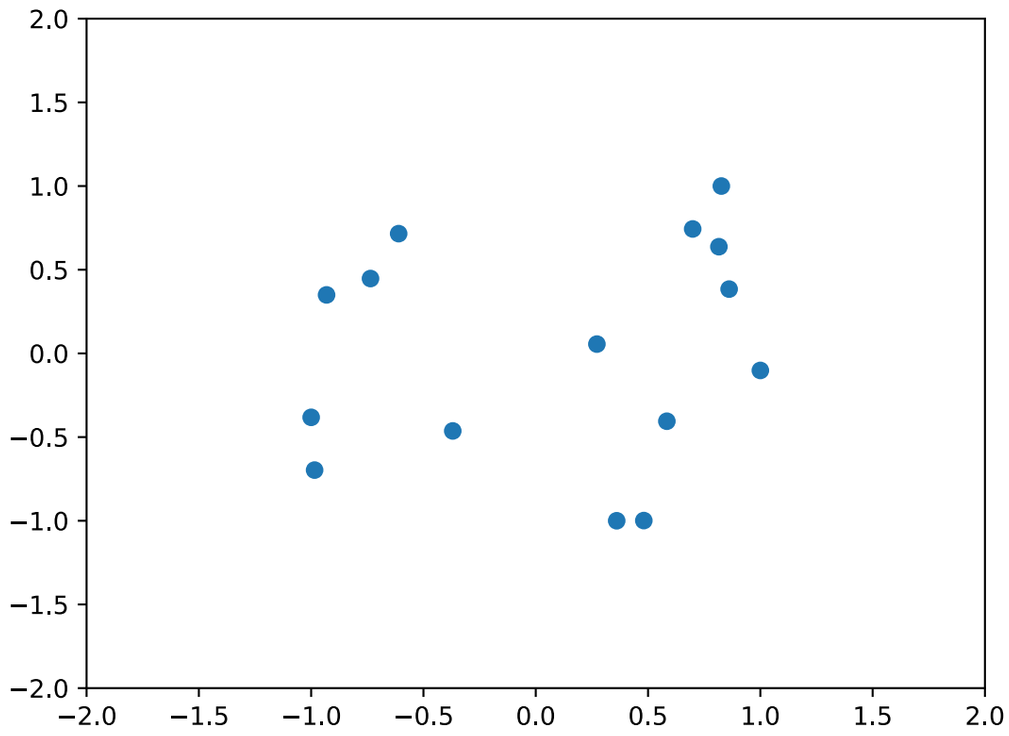


Strategy 2

Normalized training data



Normalized testing data



Strategy 1 vs. Strategy 2

- Strategy 2 results in normalized testing data that exactly fit the range [-1, +1].

Strategy 1 vs. Strategy 2

- Strategy 2 results in normalized testing data that exactly fit the range [-1, +1].

- However, normalizing in this way requires knowledge of the test distribution.

- This means it is necessary to collect a sufficiently large sample of testing data *before* running the classifier.

Data augmentation

Data augmentation

- The more training data you have, the less is the risk of overfitting.
- Unfortunately, training data are often hard to find.
- Can we synthesize new training examples automatically?

Data augmentation

- **Data augmentation** is the creation of new examples based on existing ones.

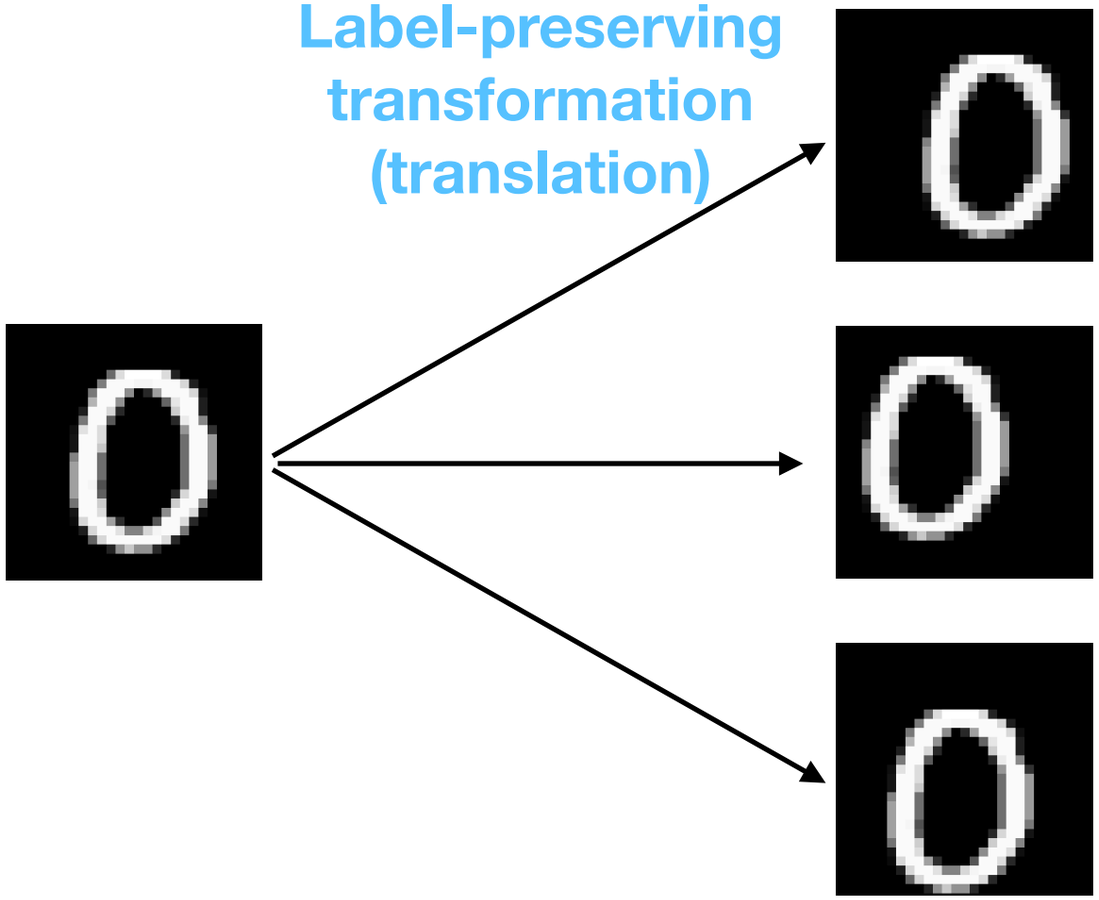
- If we can alter an existing training example without affecting its associated label, then we can generate many new training examples and train on them.

Data augmentation

- Several commonly used methods of data augmentation:
  - Adding noise to existing examples (e.g., Gaussian, Laplacian).
  - Geometric transformations (e.g., flip left/right, rotate, translate).

Example: translation

- From an existing MNIST image, translate all the pixels by some random amount (dx, dy).



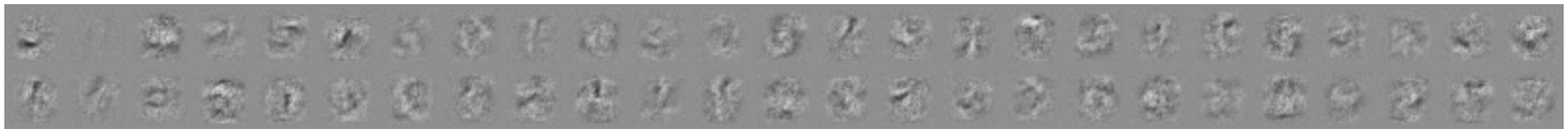
Example: translation

- Data augmentation via translation encourages the NN to learn **translation-invariant** features — they are useful for classification no matter where in the image they occur.

Example: translation

- Here are the weights $\mathbf{W}^{(1)}$ (transformed to 100x28x28) of a MNIST classification network **without** data augmentation:

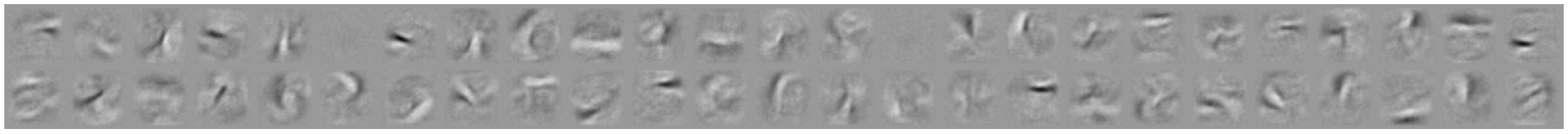
Acc=98.07



Example: translation

- Here are the weights $\mathbf{W}^{(1)}$ (transformed to 100x28x28) of a MNIST classification network **with** data augmentation:

Acc=98.44



- Compared to the previously shown weights, these show visually more well-defined contours.

(Unsupervised)
pre-training

Feature representations

- One of the reasons why NNs are so powerful is that they can learn **feature representations** of the raw input data.

- Classifying/regressing the target variable $\hat{y}$ is often easier once the raw data have been transformed into a different feature space.

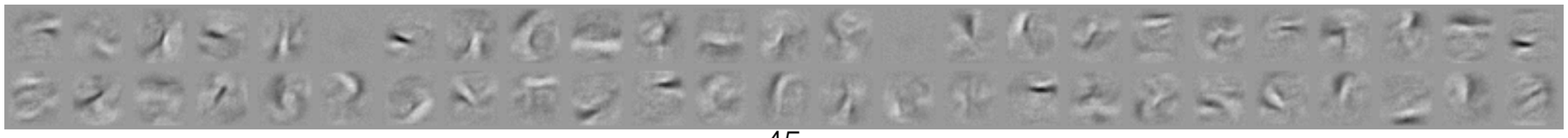
  - We saw this with XOR.

Feature representations

- One of the reasons why NNs are so powerful is that they can learn **feature representations** of the raw input data.

- Classifying/regressing the target variable $\hat{y}$ is often easier once the raw data have been transformed into a different feature space.

  - With MNIST, the NN seemed to recognize brush-strokes:



Unlabeled data

- In some application domains (e.g., object recognition/ detection in images), collecting *labeled* data is hard, but collecting *unlabeled* data is easy.

- How might abundant unlabeled data help us to train better ML models?

Learning good features

- We can harness unsupervised learning algorithms to learn good feature representations from unlabeled data.

  - **Unsupervised**: examples without labels.

Learning good features

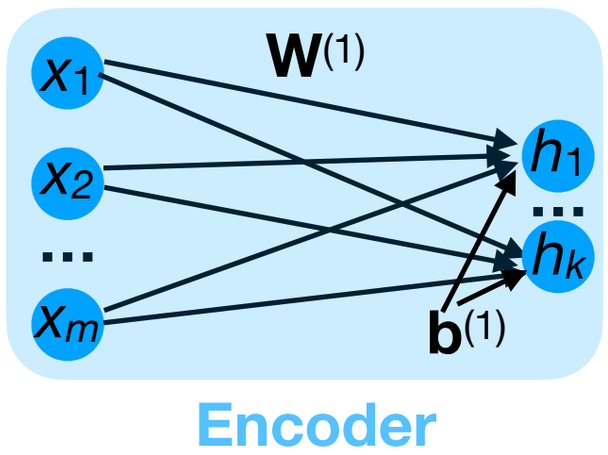
- We can harness unsupervised learning algorithms to learn good feature representations from unlabeled data.
    - **Unsupervised**: examples without labels.

- Key intuition: a good representation captures the essence of the raw input data.
    - We can “compress” the data into a smaller representation.
    - We can “uncompress” it to *reconstruct* the original data.

Auto-encoders

- Let **x** be the raw input data.
- Let **h** be the hidden representation that captures the "essence" of **x**. We say **h** has been **encoded** from **x**.
- We can compute **h** using a neural network (just 1 layer in this example, but could be deeper).

$$\mathbf{h} = \sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

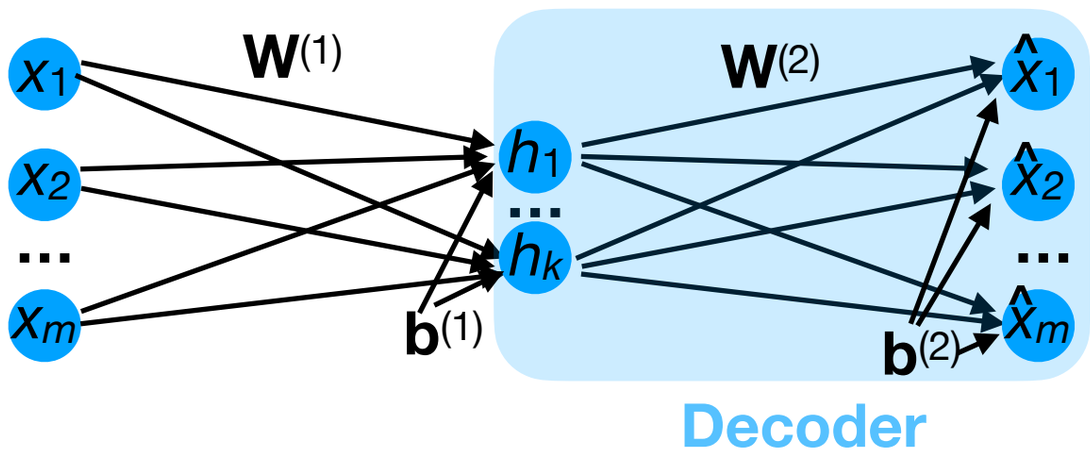


Auto-encoders

- If **h** contains the essential features of **x**, then we can use **h** to reconstruct (approximate) the original data **x**.

- Let $\hat{\mathbf{x}}$ denote our reconstruction of **x**. We say $\hat{\mathbf{x}}$ has been **decoded** from **h**.

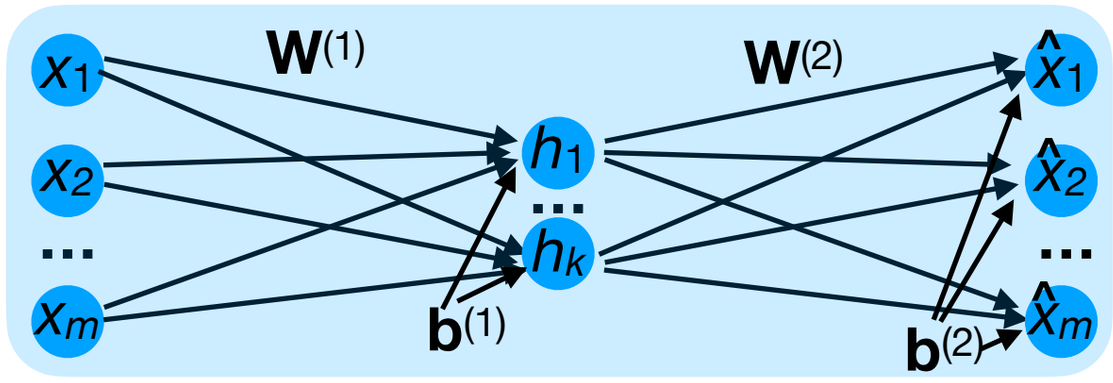
$$\hat{\mathbf{x}} = \sigma^{(2)}\left(\mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}\right)$$



Auto-encoders

- Putting the two components (encoder+decoder) together, we arrive at an **auto-encoder**.

$$\hat{\mathbf{x}} = \sigma^{(2)}\left(\mathbf{W}^{(2)}\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)$$

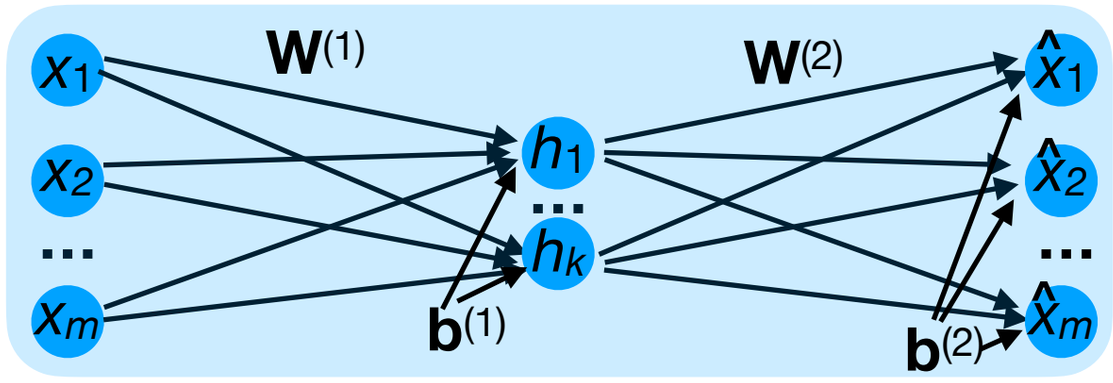


Auto-encoders: training loss function

- With auto-encoders, we optimize $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$, $\mathbf{b}^{(1)}$, $\mathbf{b}^{(2)}$ to make our reconstructions as accurate as possible, i.e., minimize:

$$f_{\text{MSE}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)})^2$$

$$= \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)})^\top(\hat{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)})$$



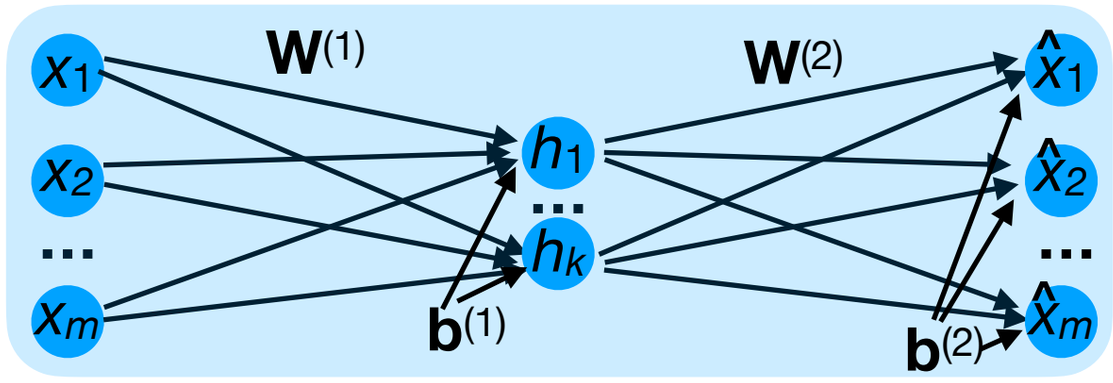
Auto-encoder

Auto-encoders: training loss function

- Notice that this loss function does not require any training labels — there is no mention of any *y*!

$$f_{\text{MSE}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)})^2$$

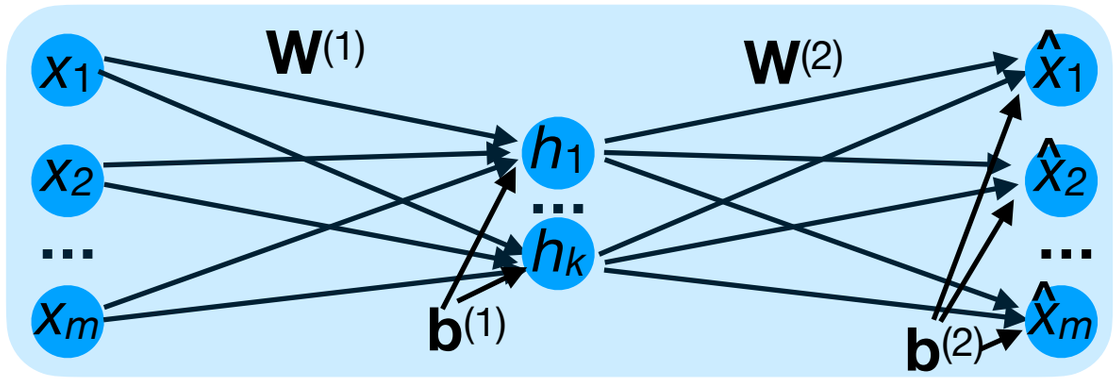
$$= \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)})^\top(\hat{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)})$$



Auto-encoder

Bottleneck

- If we let $k=m$, then what is an easy (but useless) way to set the weights to give a perfect reconstruction (0 MSE)?

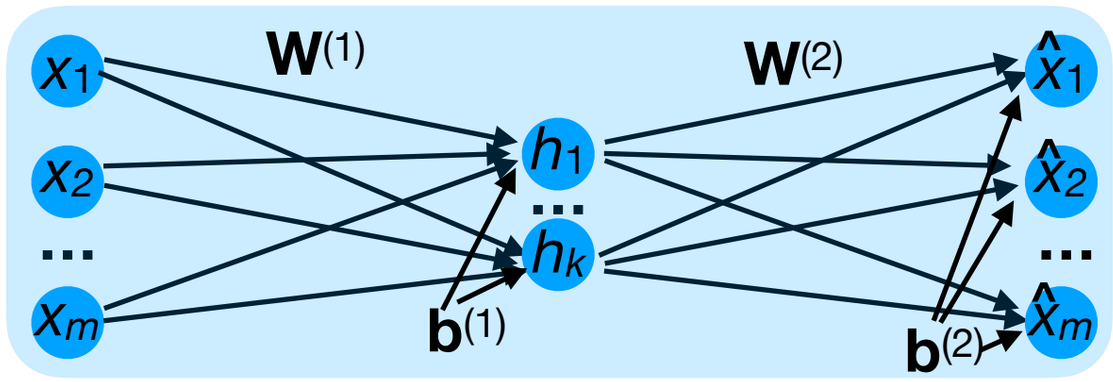


Auto-encoder

Bottleneck

- If we let k=m, then what is an easy (but useless) way to set the weights to give a perfect reconstruction (0 MSE)?

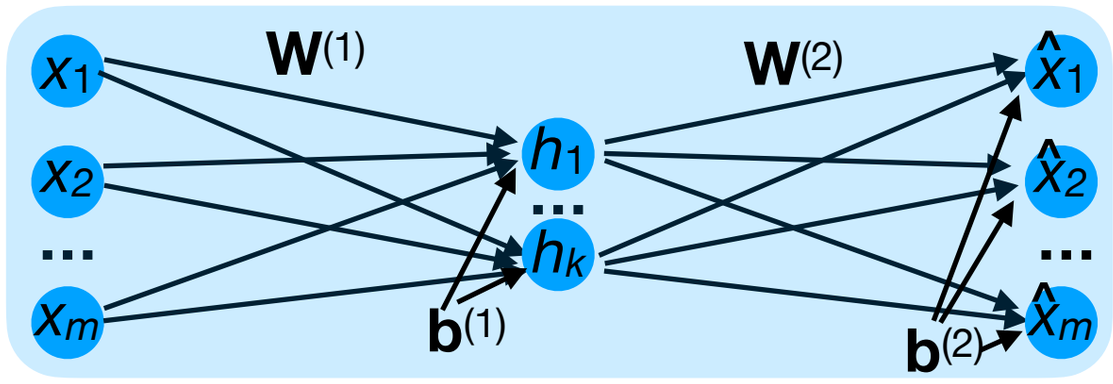
- If $k = m$, then we can just set $\mathbf{W}^{(1)} = \mathbf{W}^{(2)} = \mathbf{I}$ (identity matrix). This gives 0 MSE but does not learn any interesting representation!



Bottleneck

- If we let $k=m$, then what is an easy (but useless) way to set the weights to give a perfect reconstruction (0 MSE)?

- For this reason, we usually set $k < m^*$; the hidden layer is then called a **bottleneck**.



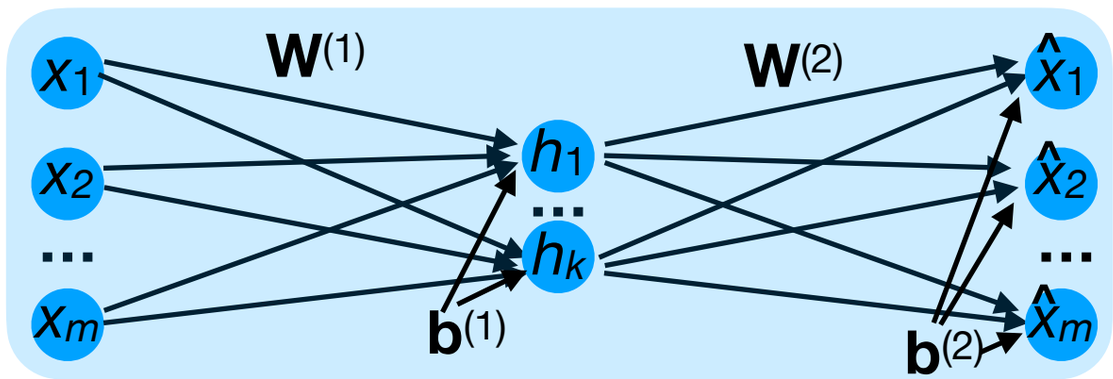
Auto-encoder

*An important exception are de-noising auto-encoders.

Autoencoders vs PCA

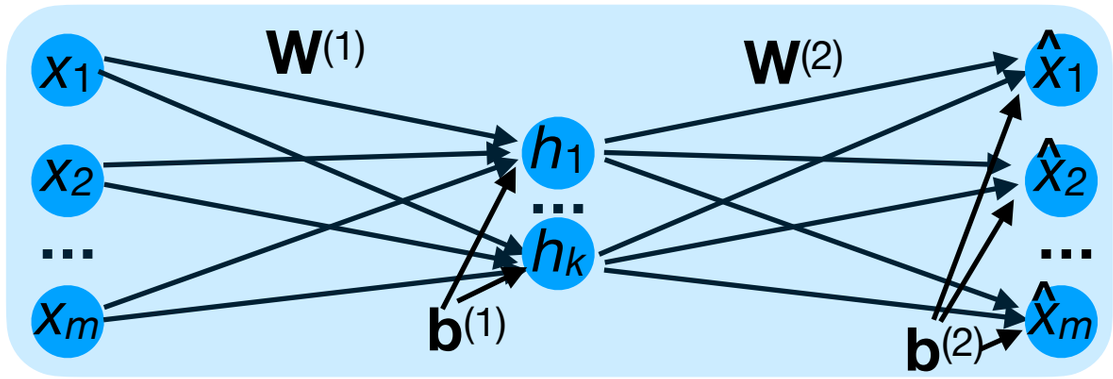
- When the activation functions are linear, and we constrain $\mathbf{W}^{(2)} = \mathbf{W}^{(1)\top}$, then the auto-encoder is almost identical to principal component analysis (with *k* PCs):
  - The span of the vectors in $\mathbf{W}^{(1)}$ is the same as the span of the *k* PCs.
- However, the vectors in $\mathbf{W}^{(1)}$ need not be orthogonal.

$$\hat{\mathbf{x}} = \sigma^{(2)}\left(\mathbf{W}^{(2)}\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)$$



Auto-encoders for unsupervised pre-training

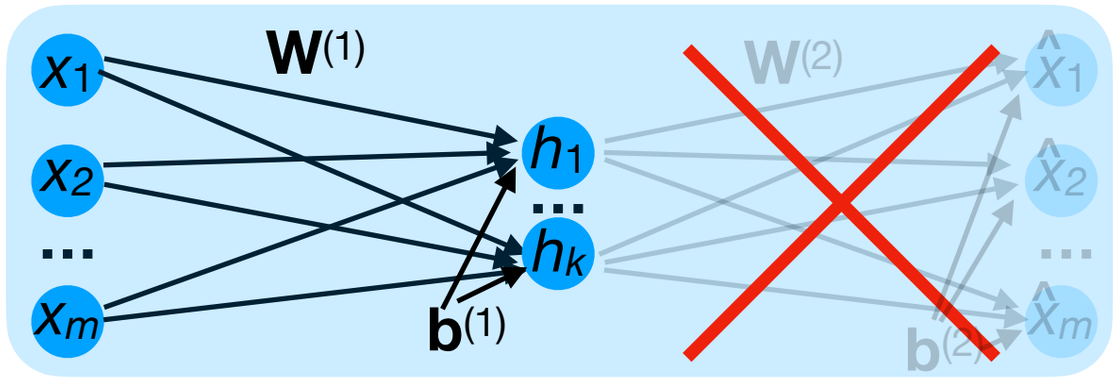
- After training the auto-encoder NN, $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ have hopefully learned to encode $\mathbf{x}$ into a representation that is useful for a variety of classification/regression problems.



Auto-encoder

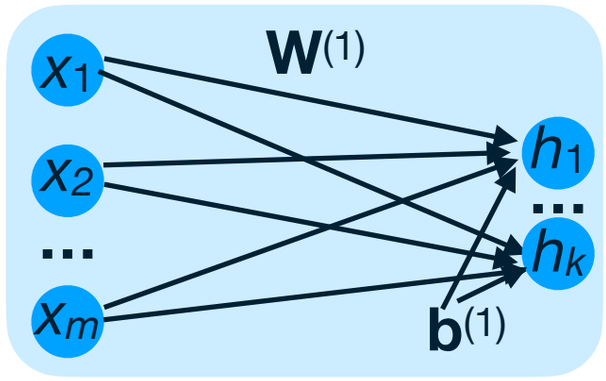
Auto-encoders for unsupervised pre-training

- After training the auto-encoder NN, $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ have hopefully learned to encode $\mathbf{x}$ into a representation that is useful for a variety of classification/regression problems.
- We can now just “chop off” the decoder layer(s)…



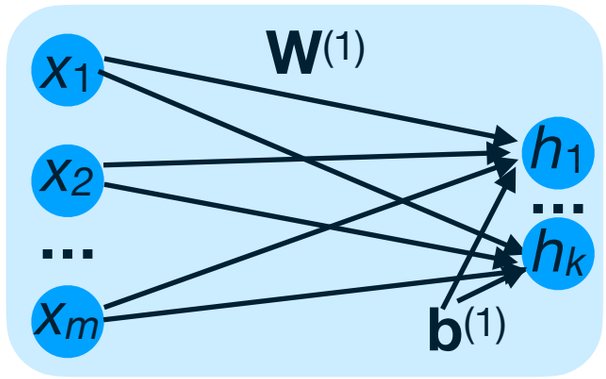
Auto-encoders for unsupervised pre-training

- After training the auto-encoder NN, $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ have hopefully learned to encode $\mathbf{x}$ into a representation that is useful for a variety of classification/regression problems.
- …and keep just the encoder layer(s).



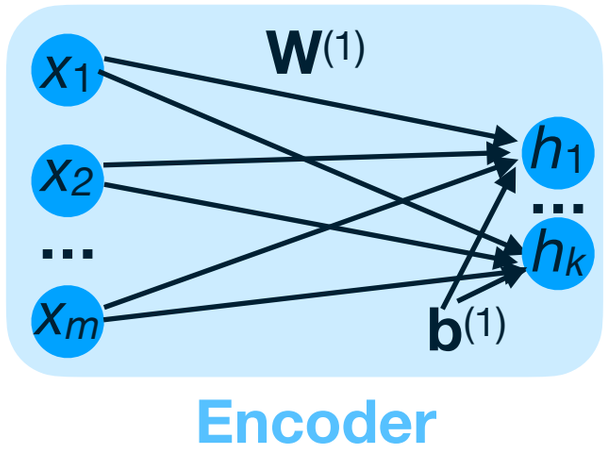
Auto-encoders for unsupervised pre-training

- We now have a trained encoder network that can "compress" every input example **x** into its "essence" **h**.



Auto-encoders for unsupervised pre-training

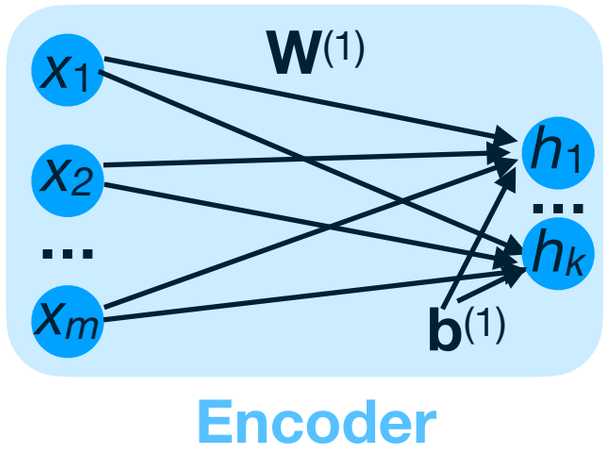
- Now, suppose we also have a (typically smaller) set of *labeled* examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$.



Auto-encoders for unsupervised pre-training

- Now, suppose we also have a (typically smaller) set of *labeled* examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$.

- We can use the encoder network to convert each **x** to **h** to obtain $\{(\mathbf{h}^{(i)}, y^{(i)})\}_{i=1}^{n}$.

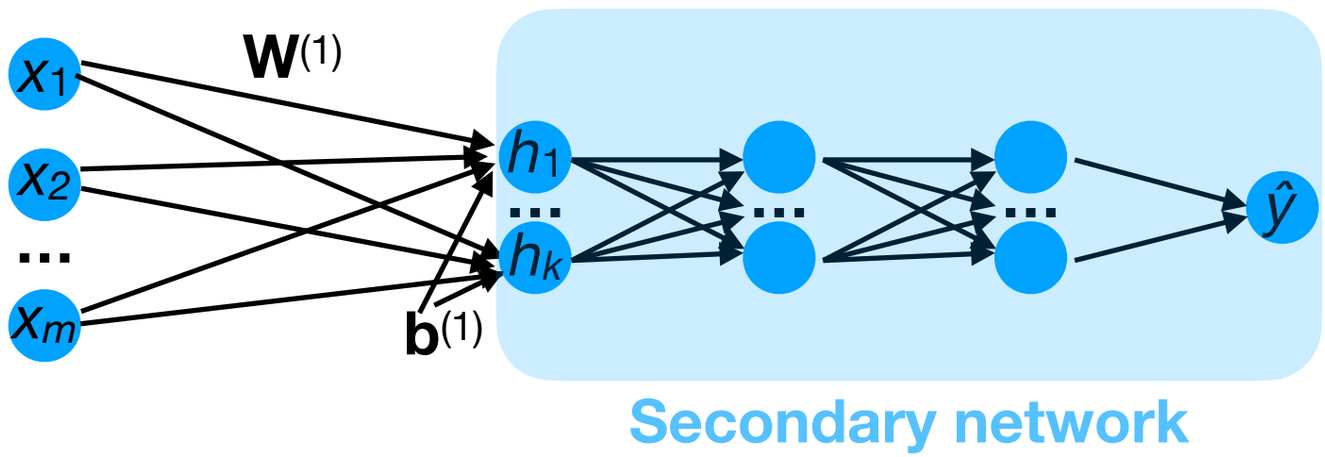


Auto-encoders for unsupervised pre-training

- Now, suppose we also have a (typically smaller) set of *labeled* examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$.

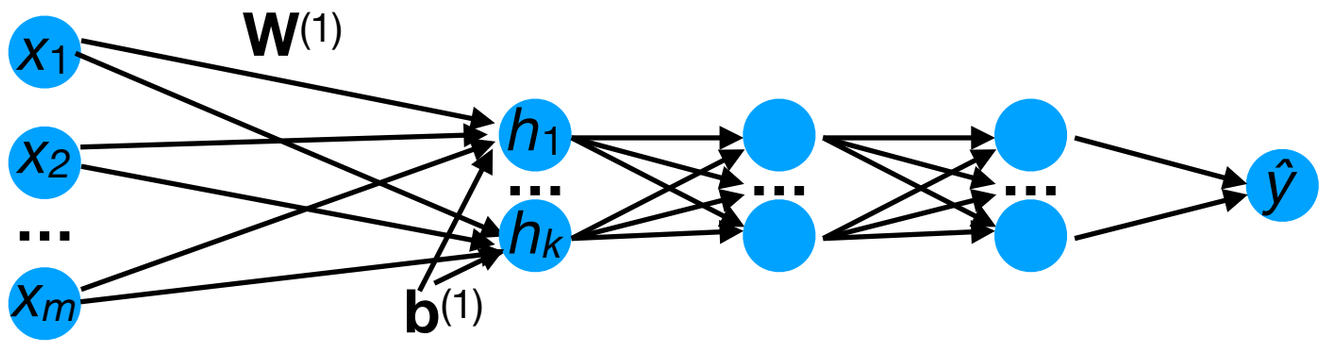
- We can use the encoder network to convert each **x** to **h** to obtain $\{(\mathbf{h}^{(i)}, y^{(i)})\}_{i=1}^{n}$.

- We then train a *secondary* NN (or any other ML model) to predict *y* from **h**.



Auto-encoders for unsupervised pre-training

- After training, the two networks (encoder + secondary) can be seen as a *single NN* that analyzes each input **x** to make a prediction $\hat{y}$.

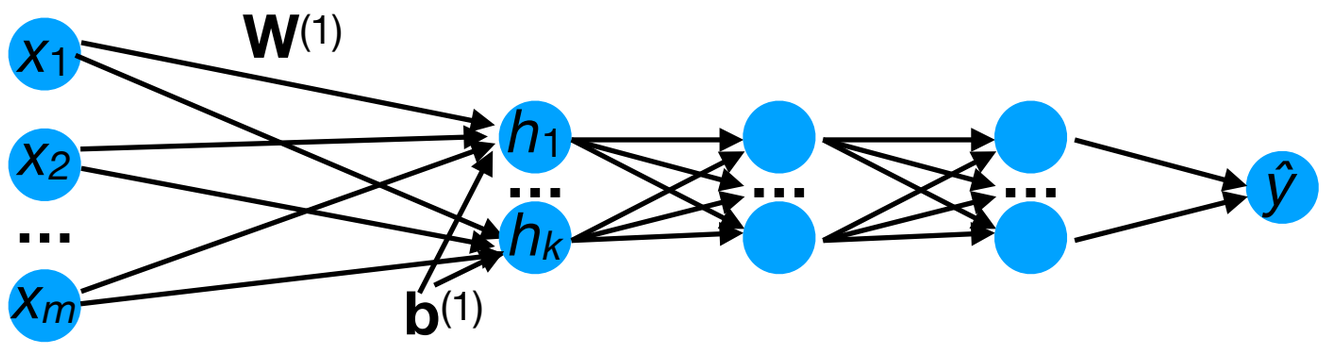


Auto-encoders for unsupervised pre-training

- Why does this help?

- The first layers of the overall network were trained on a large amount of data.

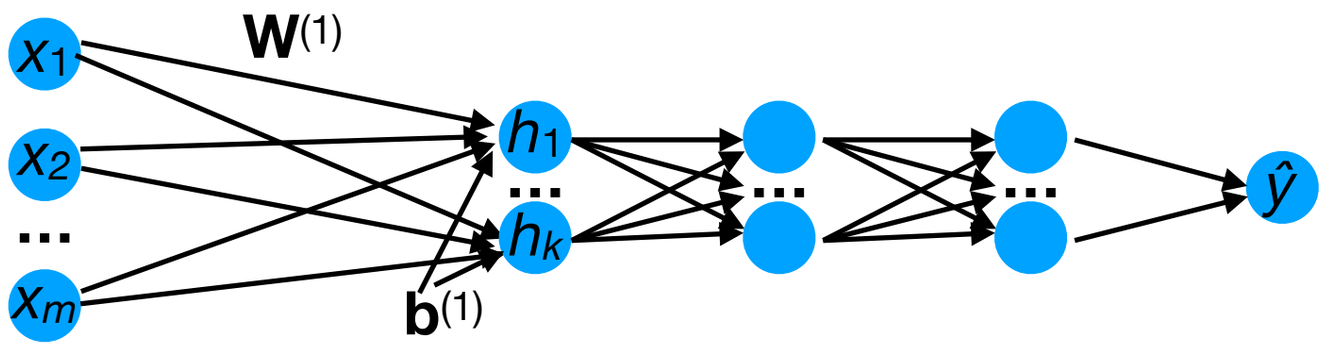
- Compressing **x** into **h** makes the secondary predictions (hopefully) easier.



Auto-encoders for unsupervised pre-training

- In addition to training the secondary NN, we can — optionally — adjust the parameters of the encoder network.

- Since the encoder was trained on a much larger (unlabeled) dataset, we don't want to "mess up" its weights too much based on just a small labeled dataset.

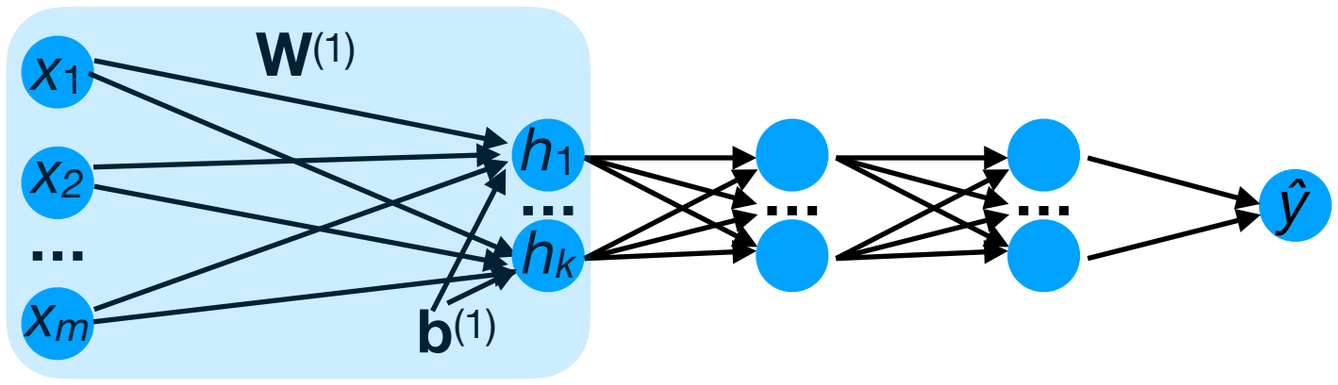


Auto-encoders for unsupervised pre-training

- In addition to training the secondary NN, we can — optionally — adjust the parameters of the encoder network.

- Since the encoder was trained on a much larger (unlabeled) dataset, we don't want to "mess up" its weights too much based on just a small labeled dataset.

- Hence, we often use a small learning rate ==> **fine-tuning**.



(Supervised)
pre-training

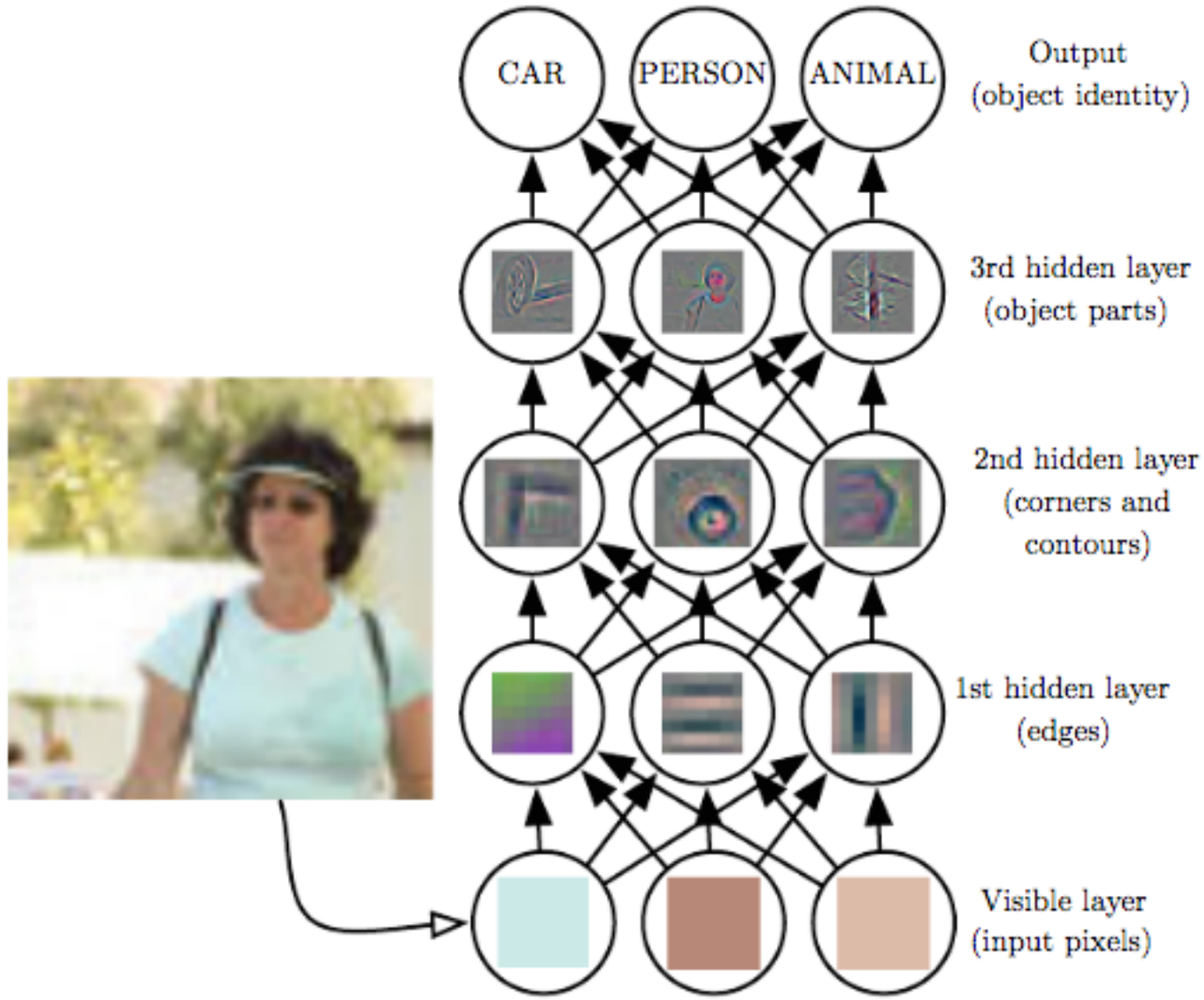
Supervised pre-training

- An alternative strategy to finding good feature representations is to borrow a NN from a related task.

- For instance, there now exist high-accuracy networks for recognizing 1000+ object categories from images (next slide).

- We can “borrow” the feature representation from one ML model and apply it to another application domain…

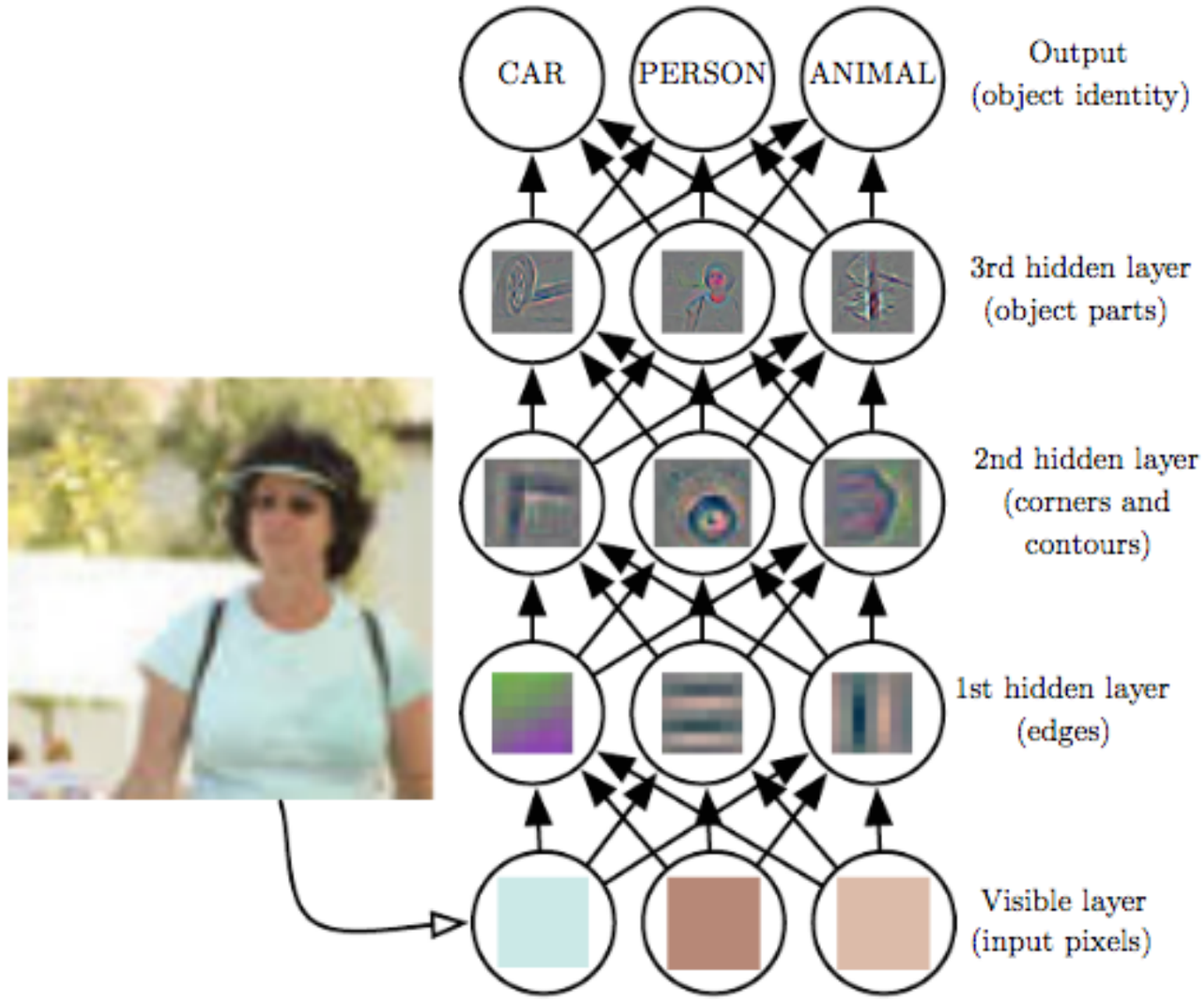
Learning representations



- The first feature representation looks vaguely like the representation learned by my MNIST network.

http://www.deeplearningbook.org/contents/intro.html

Learning representations



- Each layer of the network finds successively more abstract feature representations.
  - This was not “hard-coded” — it just turned out that these representations were useful for predicting the target labels.

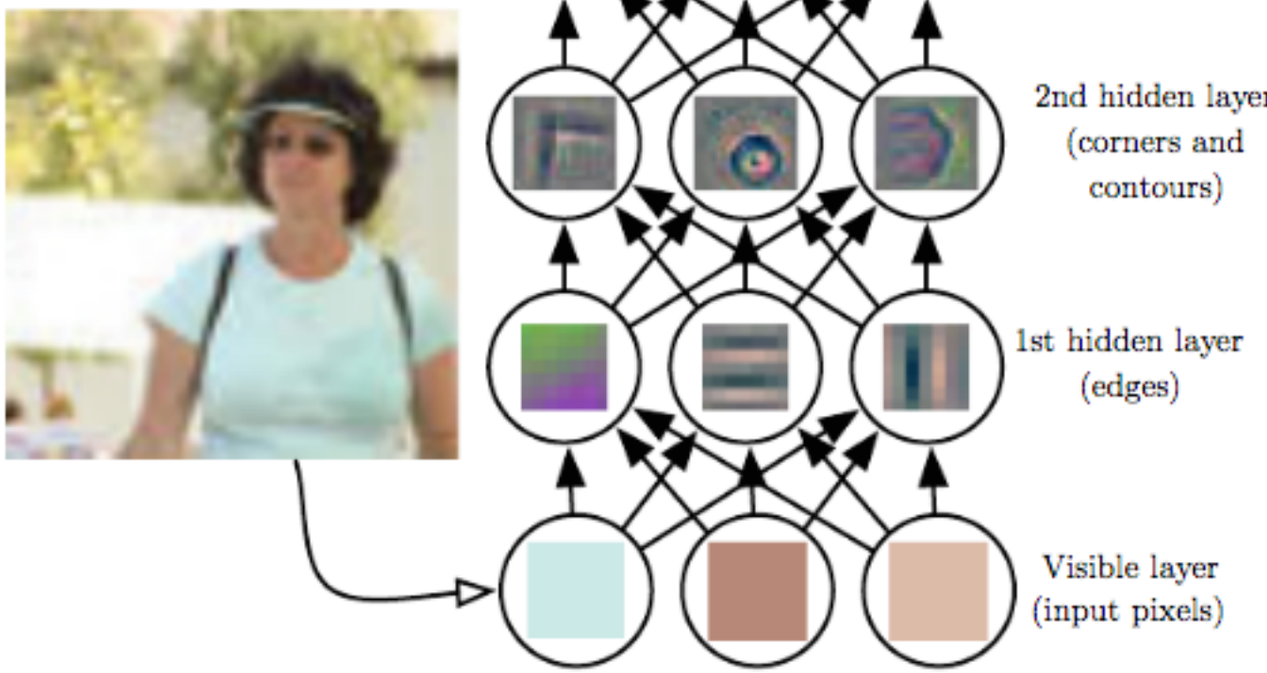
http://www.deeplearningbook.org/contents/intro.html

Supervised pre-training

- Might one (or more) of the feature representations from this NN do well on a different but related problem, e.g., smile detection or age estimation?
- Strategy:
  1. Pre-train a NN on a large dataset (e.g., ImageNet) for a general-purpose image recognition task.
  2. “Chop off” the final layer(s).
  3. Add a secondary network in place of the deleted layers, and train it for the new prediction task.
  4. Optional: fine-tune the rest of the NN.

Supervised pre-training

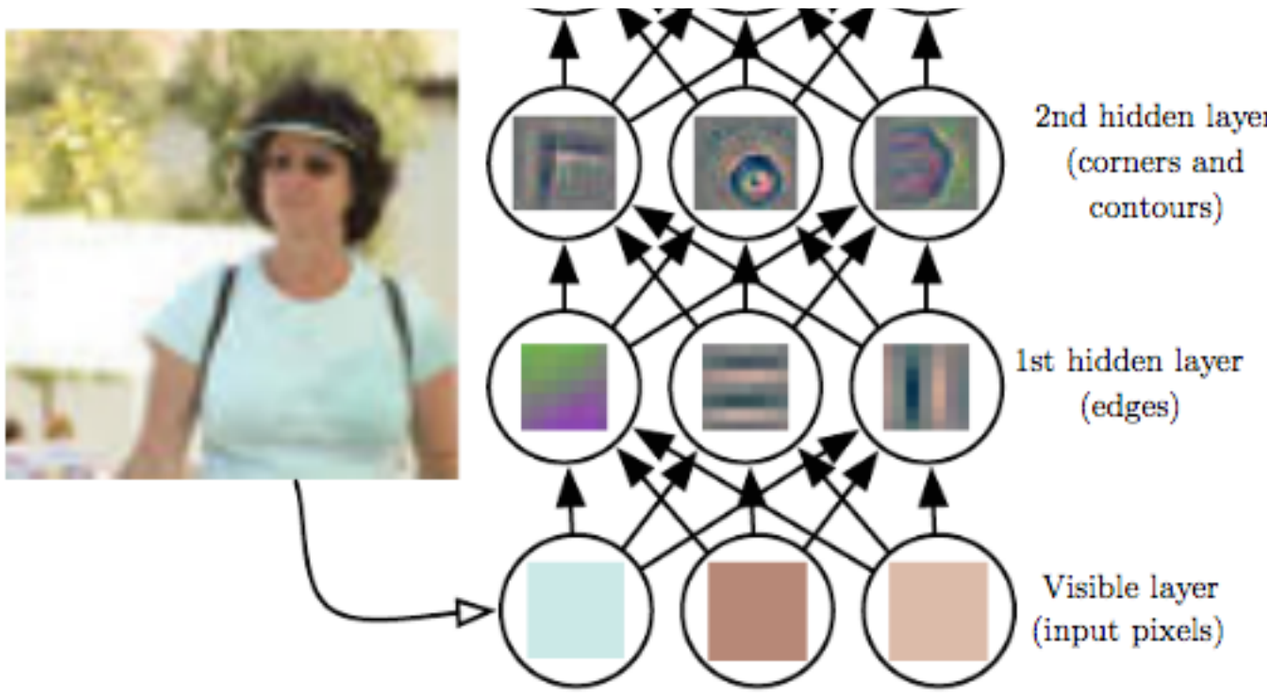
Replace with secondary network for new application domain.



http://www.deeplearningbook.org/contents/intro.html

Supervised pre-training

Chop off.



http://www.deeplearningbook.org/contents/intro.html

Supervised pre-training

- This strategy is known as **supervised pre-training** and can be highly effective for application domains for which only a small number of labeled data are available.

Multi-task learning (MTL)

Multi-task learning (MTL)

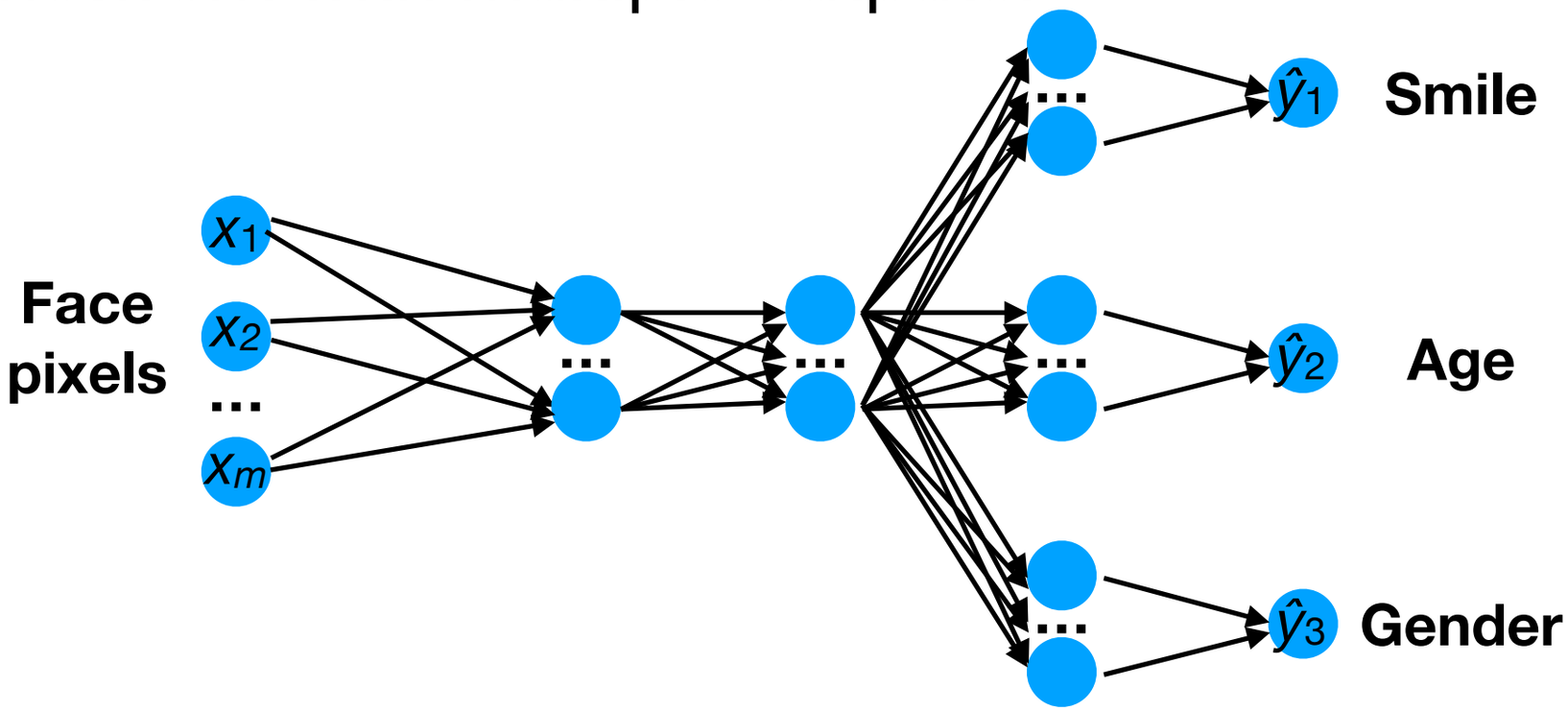
- A NN can generalize to unseen data when it computes a hidden representation that explains the data well.

- We can sometimes encourage the NN to learn a general hidden representation by training it to solve multiple related tasks.

- E.g., for automated face analysis:
  - Smile detection
  - Age estimation
  - Gender detection

Multi-task learning (MTL)

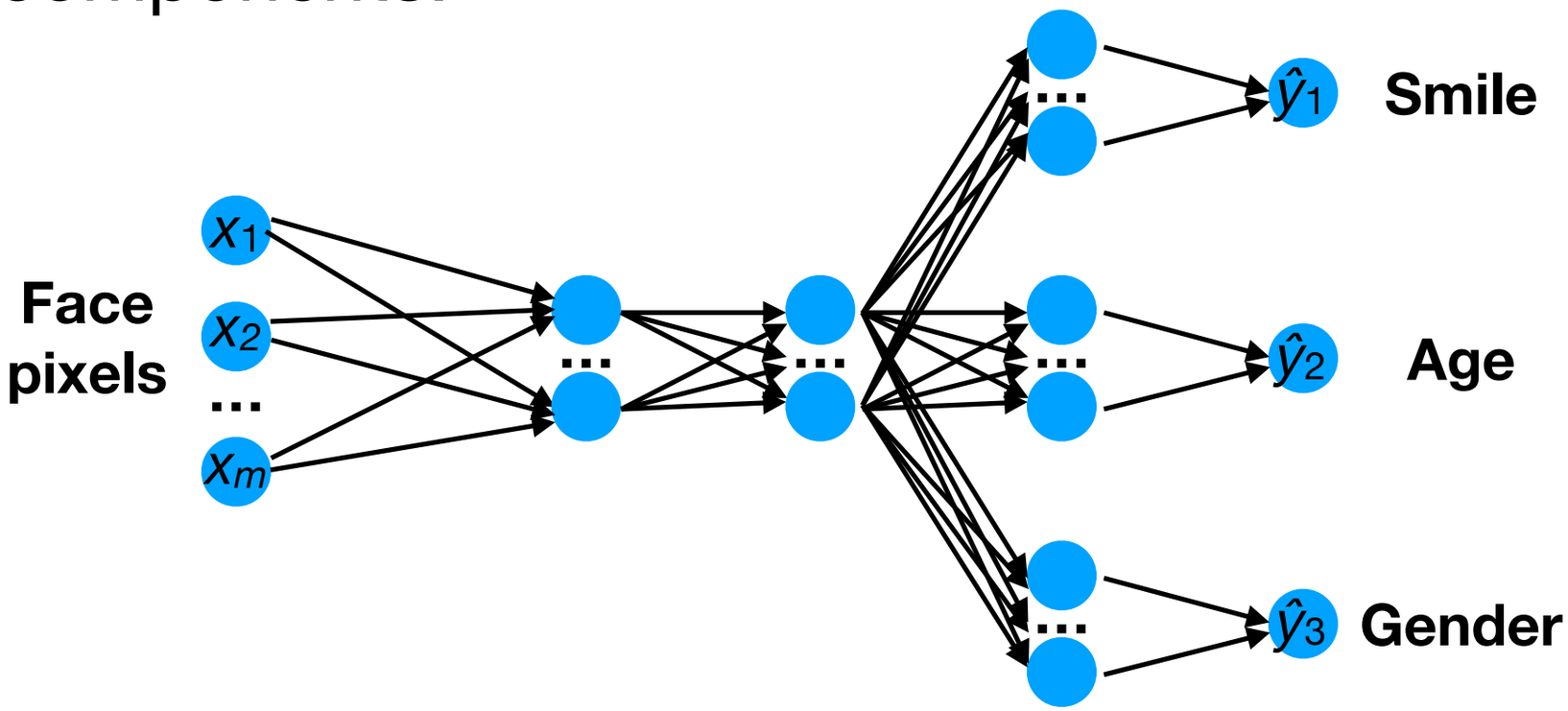
- If we have labeled data for all these tasks, we can train a network with multiple outputs:



- Note that the labels for the auxiliary tasks can be useful even if we only care about one task.

Multi-task learning (MTL)

- With MTL, our loss function consists of multiple components:



$$f_{\text{MTL}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots) = f_{\text{smile}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots) + f_{\text{age}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots) + f_{\text{gender}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots)$$

Fairness in ML

Fairness in ML

- Suppose we are training a classifier to perceive whether a person is smiling based on their face image.


vs.


- Suppose the joint probability distribution of our training labels is:

	Male	Female
Smile	0.47	0.02
Non-smile	0.45	0.06

Almost all data are male faces

Fairness in ML

- Suppose we are training a classifier to perceive whether a person is smiling based on their face image.


vs.


- Suppose the joint probability distribution of our training labels is:

	Male	Female
Smile	0.47	0.02
Non-smile	0.45	0.06

Almost all female faces are non-smiling

Fairness in ML

- To minimize prediction error, the training algorithm can harness the fact that *most women do not smile* ***in this dataset***.


vs.


	Male	Female
Smile	0.47	0.02
Non-smile	0.45	0.06

Fairness in ML

- At test time, that machine might implicitly infer that the face is female, and then use that to “downgrade” the estimate of the smile probability.


vs.


	Male	Female
Smile	0.47	0.02
Non-smile	0.45	0.06

Fairness in ML

- In contrast, a male face has a roughly equal chance of being classified as smile/ non-smile.


vs.


	Male	Female
Smile	0.47	0.02
Non-smile	0.45	0.06

Fairness in ML

- Consider the following definition of ML fairness:
    - For all subgroups (i.e., smiling males, non-smiling males, smiling females, non-smiling females), the prediction accuracy should be equal.

Fairness in ML

- Consider the following definition of ML fairness:
    - For all subgroups (i.e., smiling males, non-smiling males, smiling females, non-smiling females), the prediction accuracy should be equal.
- Then the classifier described above is unfair because female faces would likely be perceived less accurately compared to male faces.

Fairness in ML

- Mitigating strategies:
    - Collect more training data for specific populations.
    - Train the classifier with structural constraints that prevent the exploitation of correlated but non-causal features:
        - In this dataset, gender is correlated with smile, but does not cause smile!