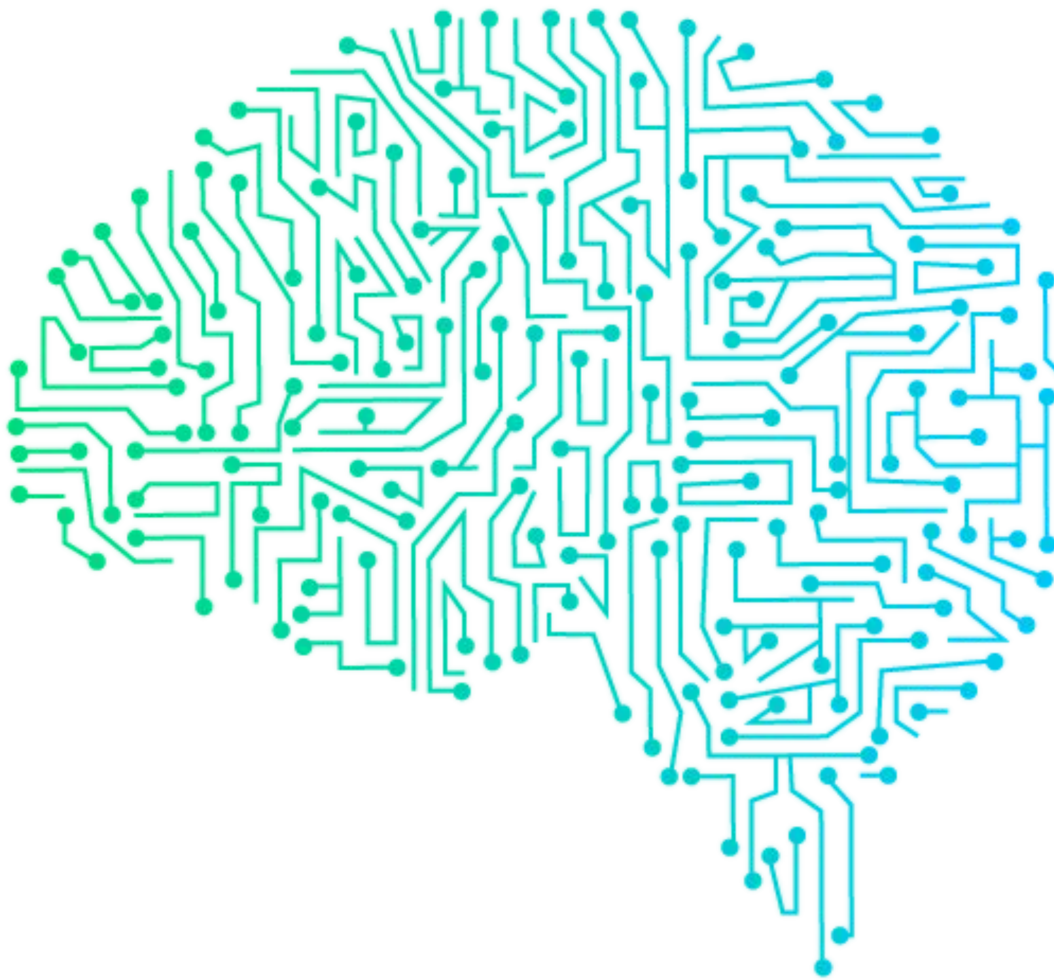
Transfer Learning

CS 175



What is transfer learning?

Transfer learning in practice

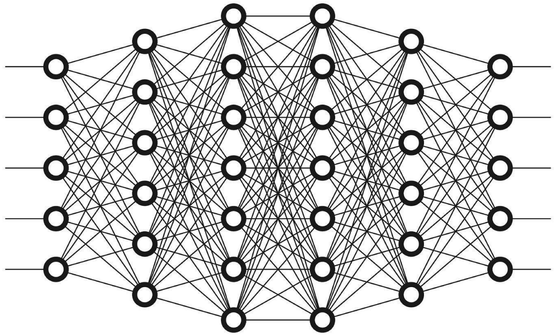
Bonus: *Guided Transfer Learning*,
a novel meta-learning methodology

What is transfer learning?

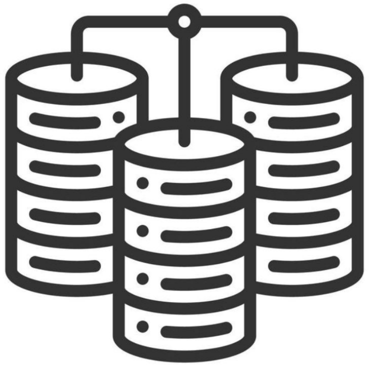
Transfer learning in practice

Bonus: *Guided Transfer Learning*,
a novel meta-learning methodology

Buzzwords you're sick of hearing



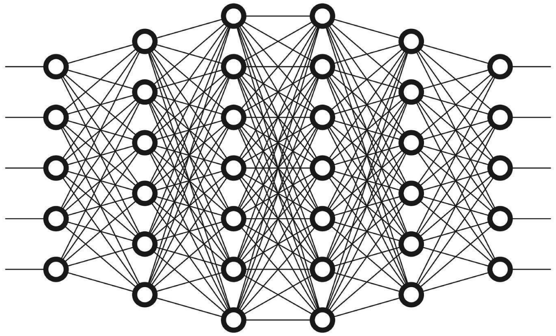
Deep Learning



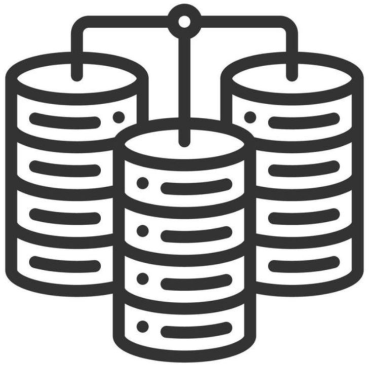
Big Data

What's the relationship?

Buzzwords you're sick of hearing



requires



Deep Learning

Big Data

(to not suck)

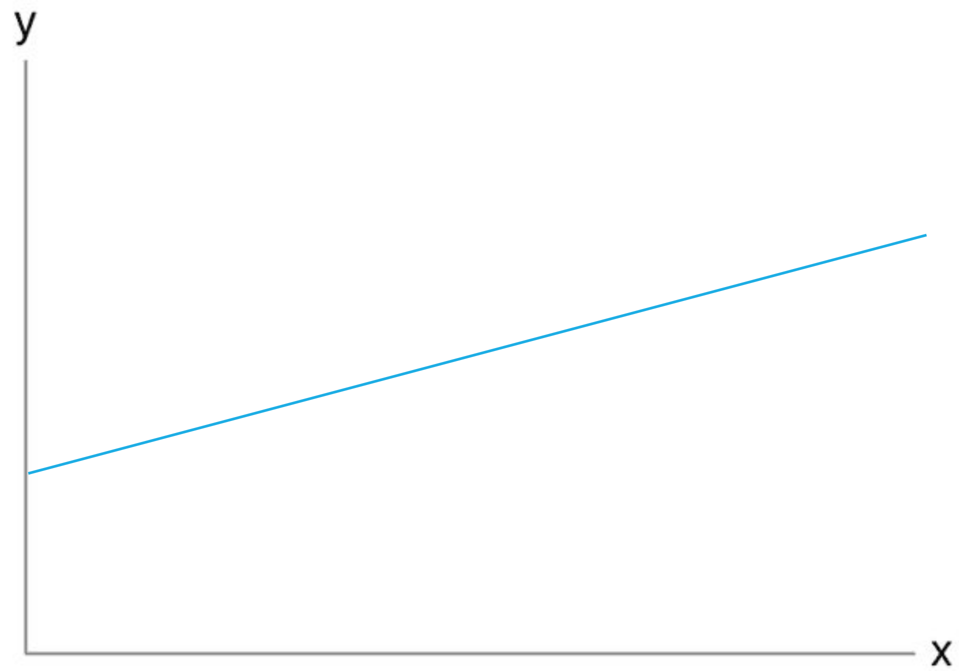
Why?

Deep neural networks are very *expressive*, and can thus be very powerful

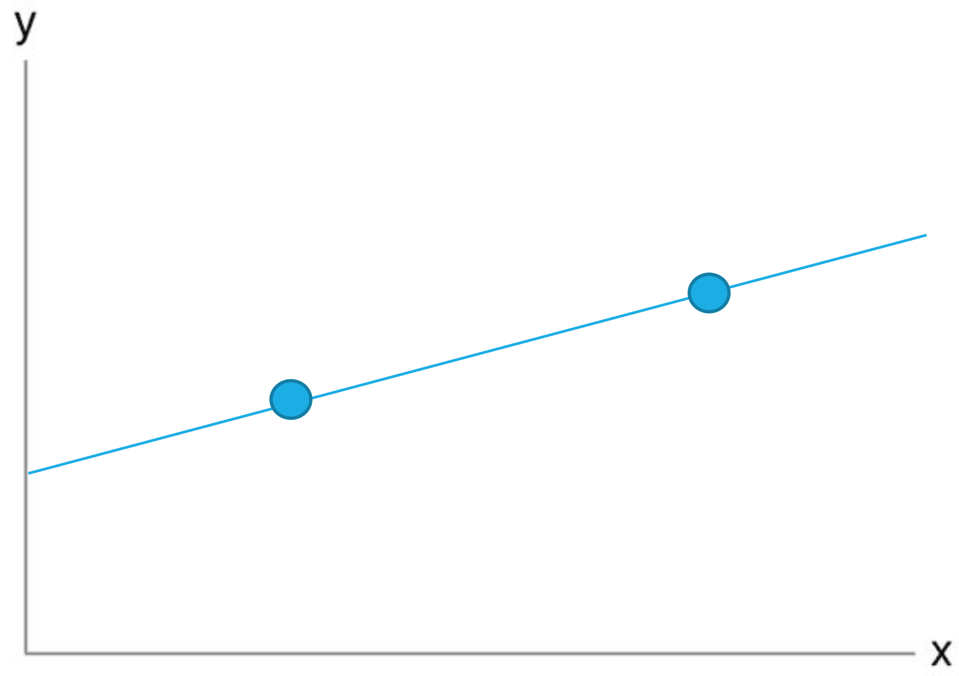
But with greater power…

Comes great risk of overfitting

This is the true, underlying relationship between inputs and outputs



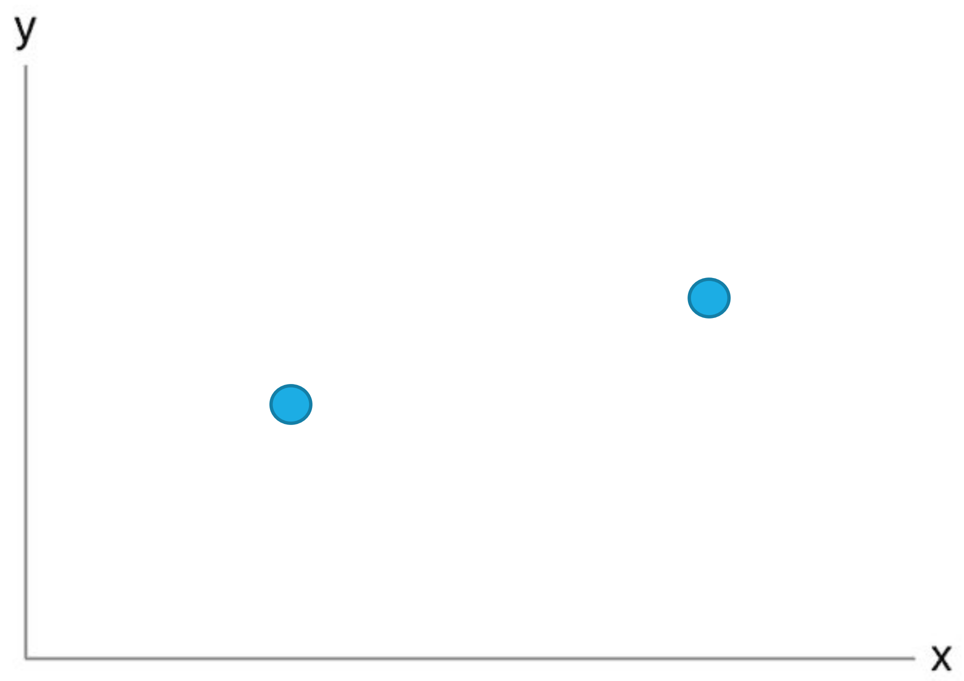
These are the the data points generated from this relationship that you give the model during training



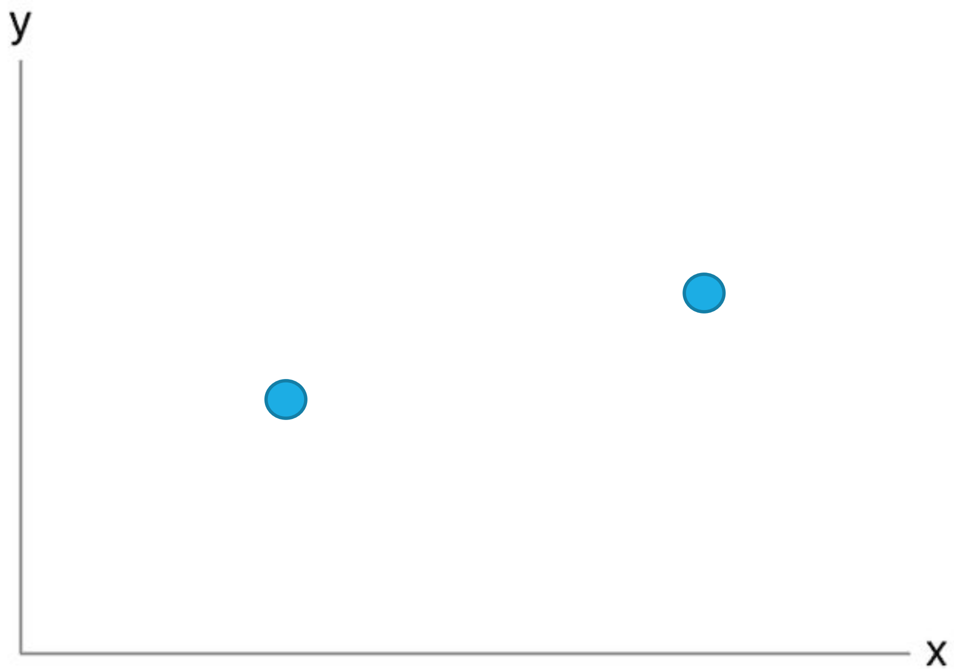
The model doesn't see the underlying relationship, only the observed training points.

If the model is a deep neural network, it's very flexible, so it can learn a relationship like this…

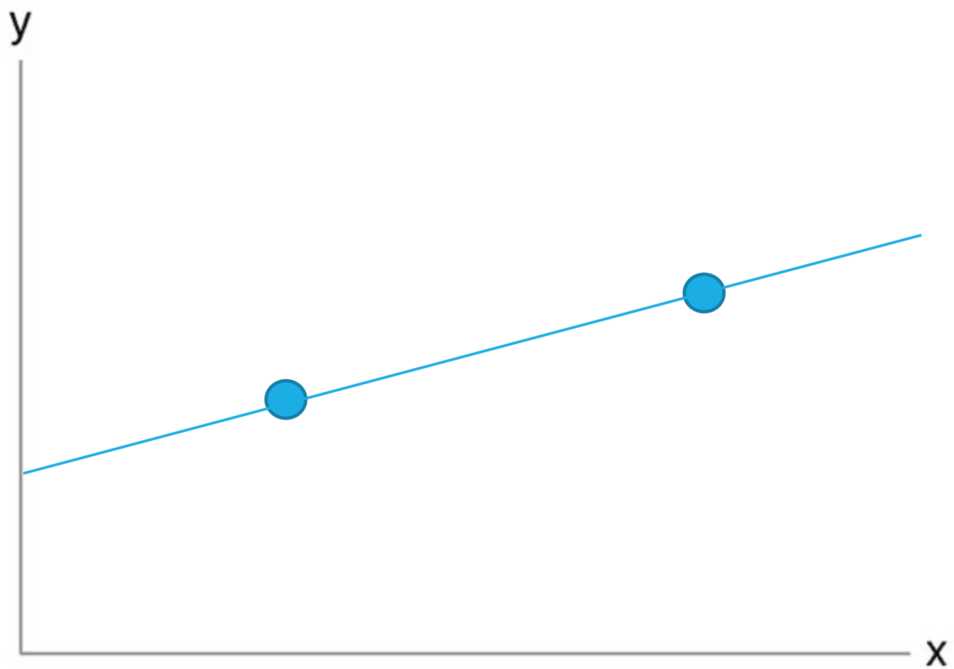
Or this…



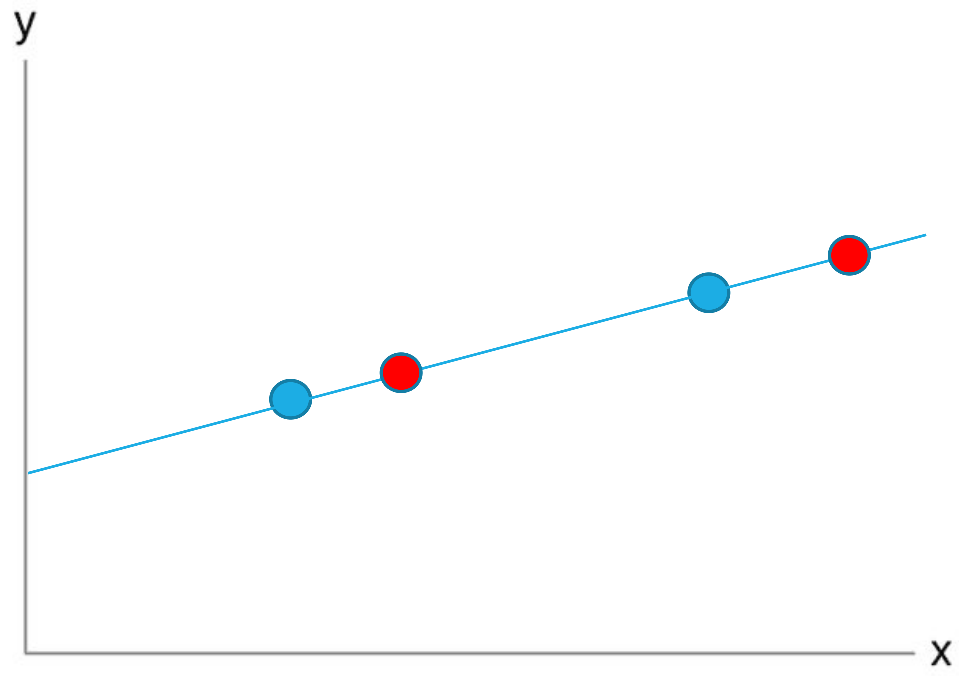
Or even this.
This learned relationship technically fits the training data…



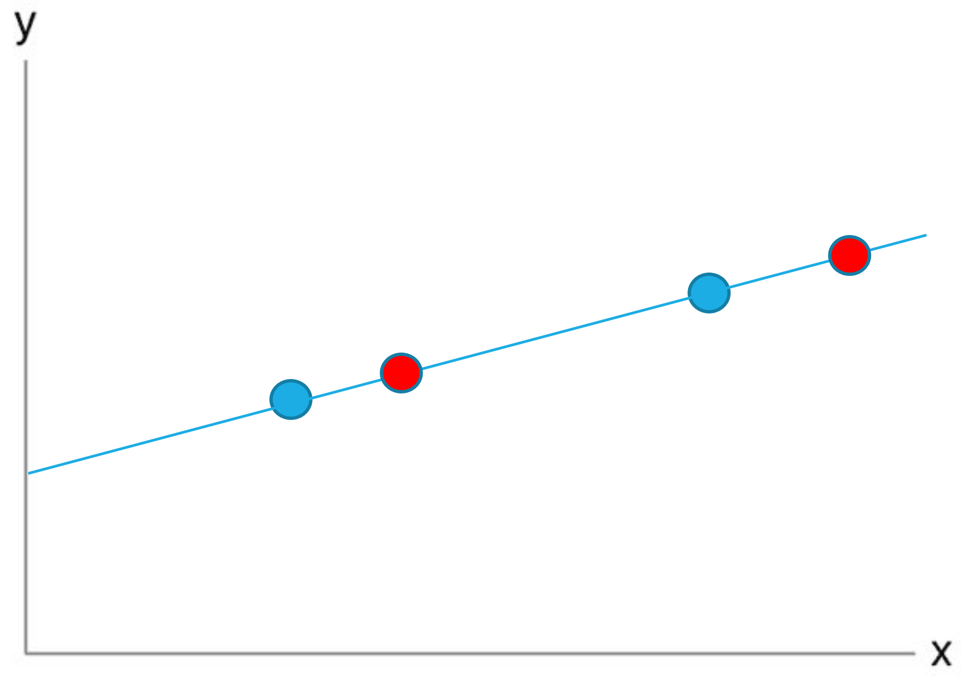
But is very far from the true underlying relationship.



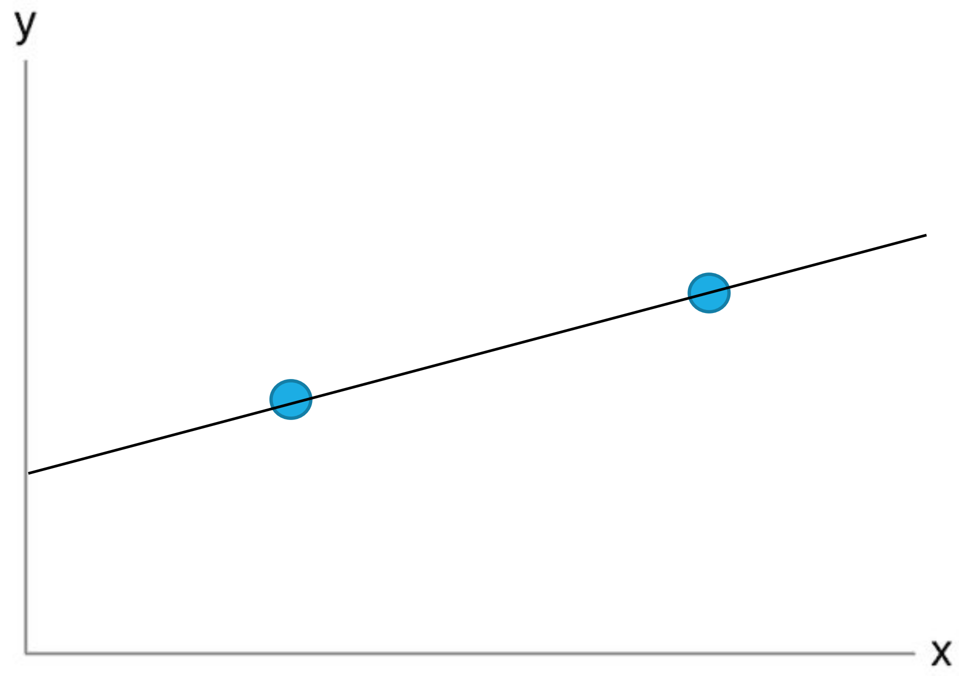
But is very far from the true underlying relationship.
And would not generalize to newly observed points
generated from that relationship.



There's just not enough data to constrain the possible relationships the model can learn.



There's just not enough data to constrain the possible relationships the model can learn.



The biggest problem regarding big data is that it is a big requirement to prevent overfitting…

but is also a big challenge to obtain for many applications

`Does it have to be this way?`

The biggest problem about big data is that it is a big requirement to prevent overfitting…

but is also a big challenge to obtain for many applications

Does it have to be this way?

A human child can be shown one cat and one dog and be able to distinguish between all future cats and dogs they see…

This is called **one-shot/few-shot learning**, how can we get AI closer to this level?

Transfer Learning: Stop learning from scratch!

Transfer Learning: Stop learning from scratch!

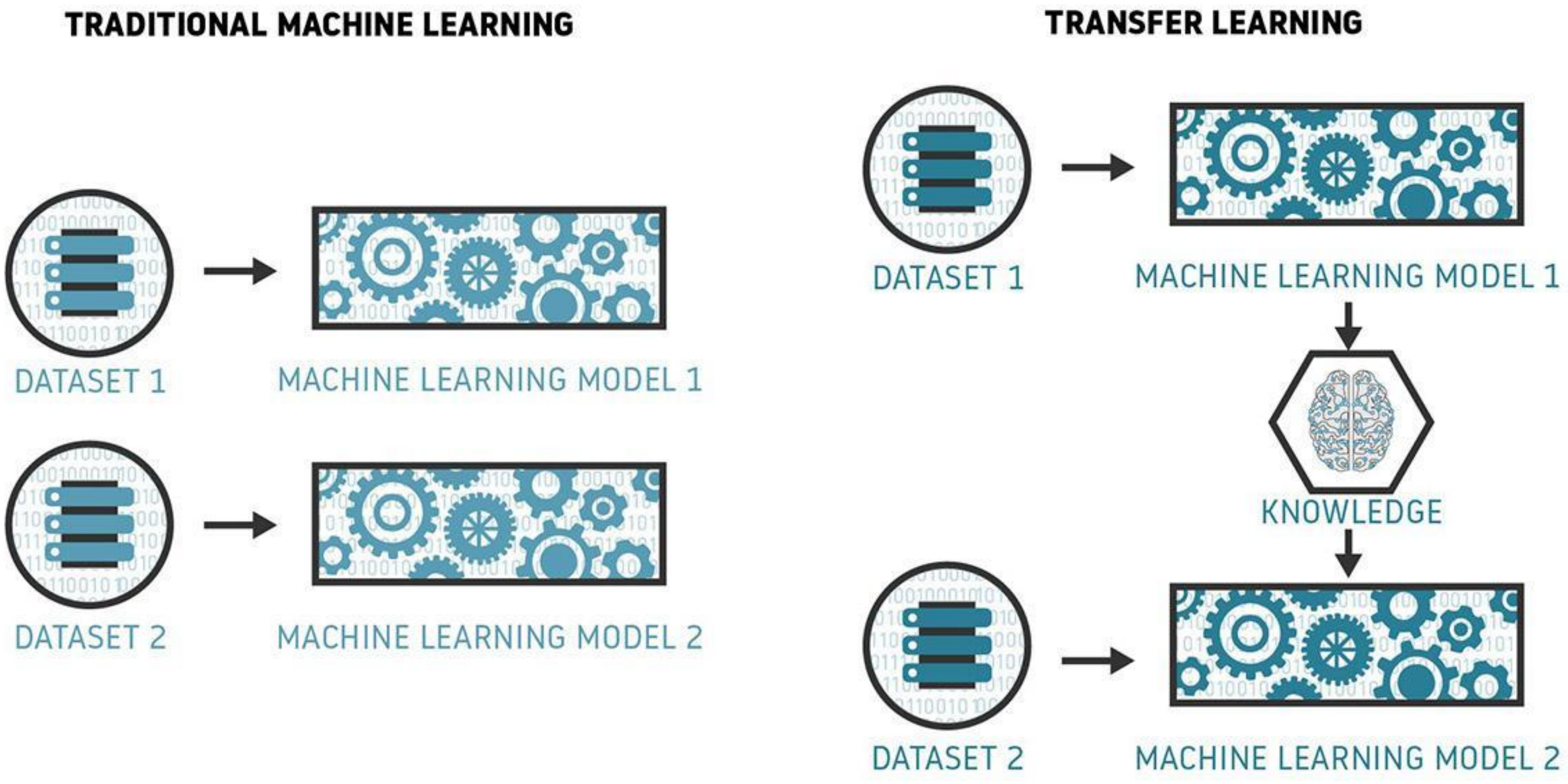
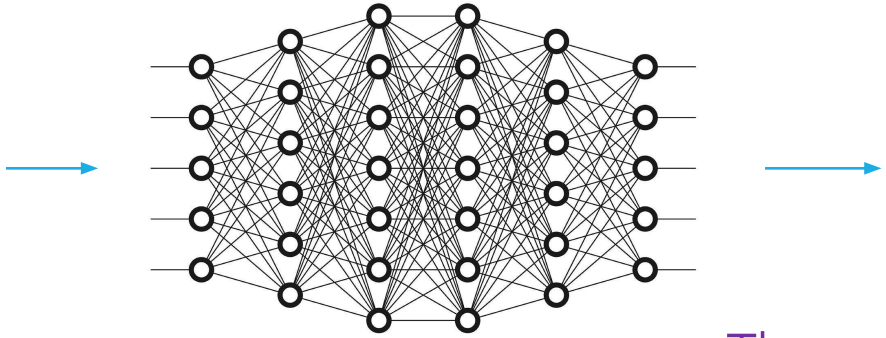


Image from Diego Lopez Yse

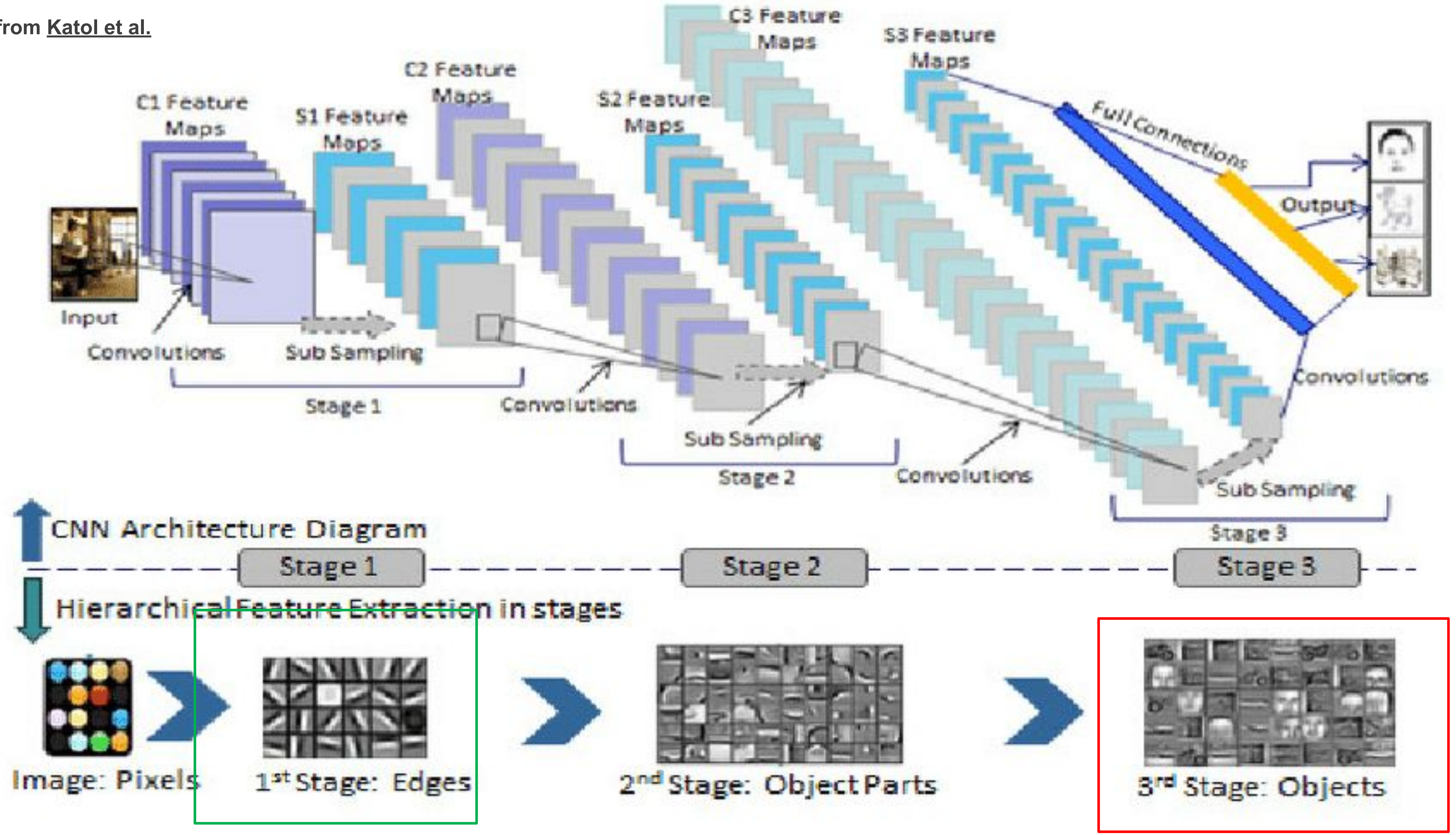
Task A: Train neural network to classify cat vs dog images



The weights of the trained neural network encode a lot of knowledge specific for differentiating between cats and dogs.

But they also encode for a lot of *general knowledge* about image data in general, like how to detect the edges of objects.

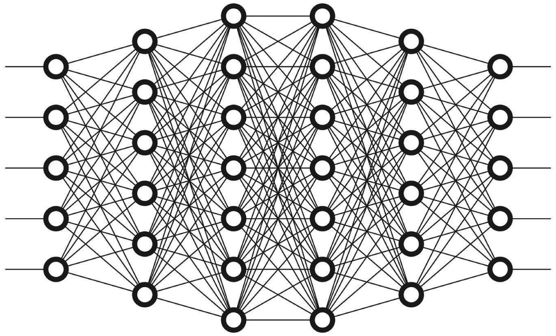
Image from Katol et al.



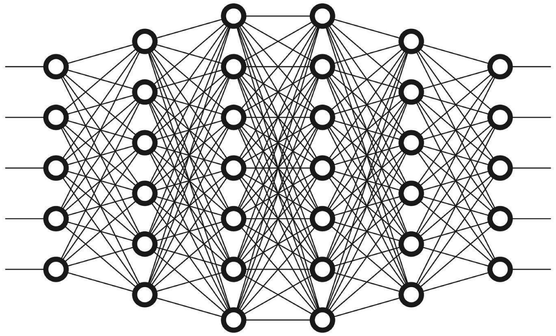
General knowledge about image data, transferrable to other image-related tasks

Knowledge specific to a particular task, must be fine-tuned for each new task.

Task A: Train neural network to classify cat vs dog images

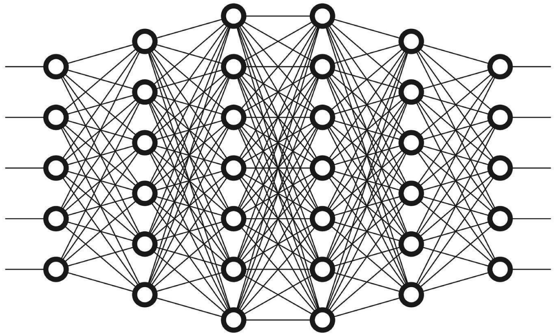


Cat or Dog?



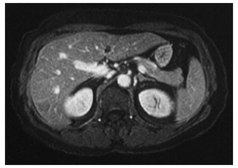
We can then take this model, with its already pre-trained weights…

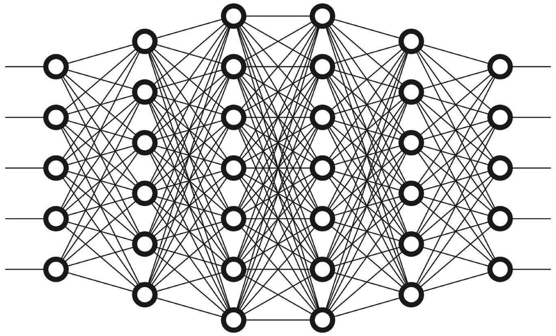
Task A: Train neural network to classify cat vs dog images



Cat or Dog?

And use it as a starting point for a new, *similar* training task within the *same domain* (i.e. another image classification task)

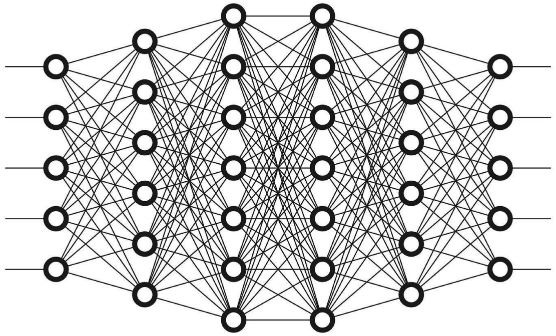




Cancer or Not Cancer?

Task B: Train neural network to classify cancer vs not cancer MRI scans

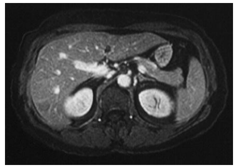
Task A: Train neural network to classify cat vs dog images

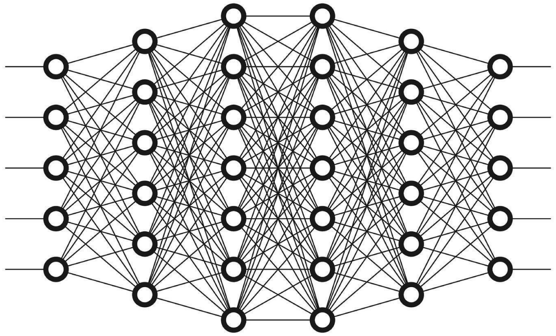


Cat or Dog?

For task B, the model will have to learn specifics about how to differentiate cancer from non-cancer.

But it *doesn't* have to re-learn general knowledge about image data (like how to detect the edges of an object/entity in an image)!

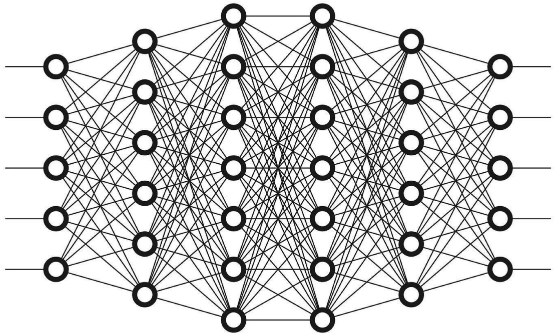




Cancer or Not Cancer?

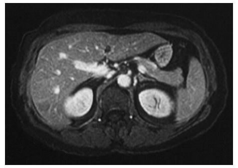
Task B: Train neural network to classify cancer vs not cancer MRI scans

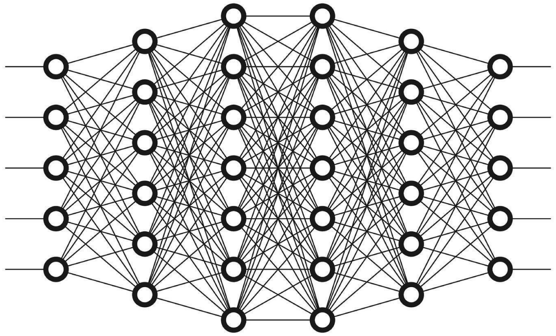
Task A: Train neural network to classify cat vs dog images



Cat or Dog?

That is, general knowledge TRANSFERS over to the new, related task

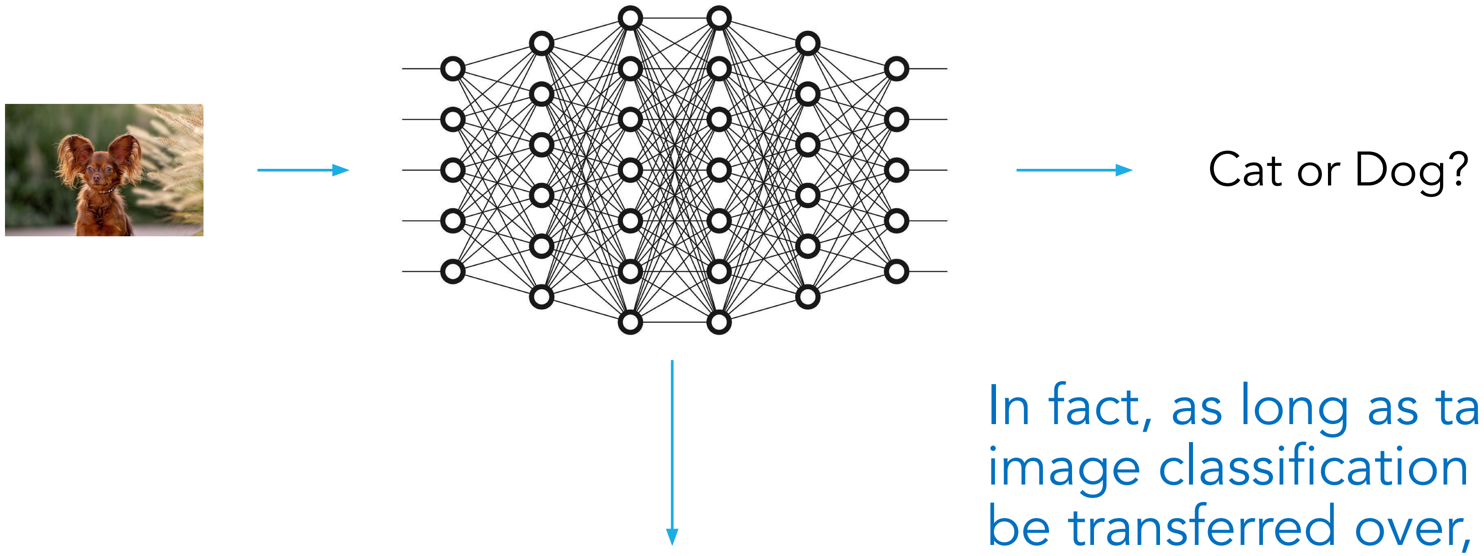




Cancer or Not Cancer?

Task B: Train neural network to classify cancer vs not cancer MRI scans

Task A: Train neural network to classify cat vs dog images

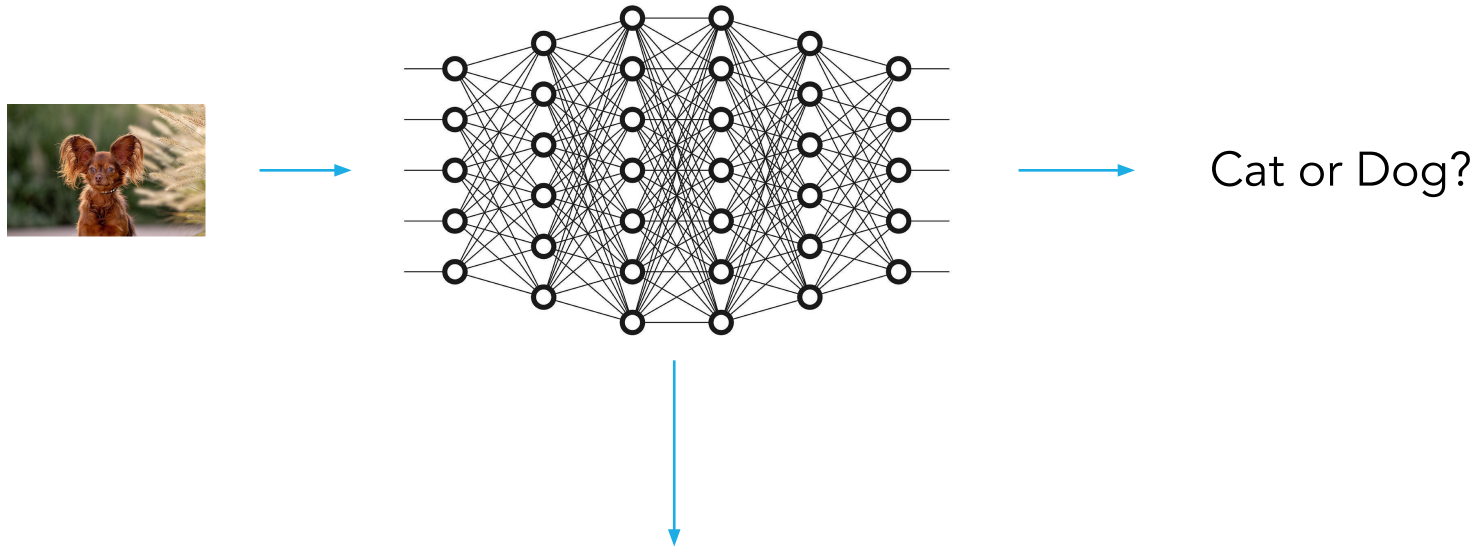


In fact, as long as task B is related to task A (e.g. is an image classification task), some general knowledge can be transferred over, and training will be more efficient!

Task B: really just any image classification task

Task B is also known as the *downstream task*

Task A training is known as pre-training



Task B training is known as the fine-tuning

So pre-training with task A helps improve efficiency for task B training

But what does "more efficient" mean? Less time/fewer epochs needed to train the model on task B?

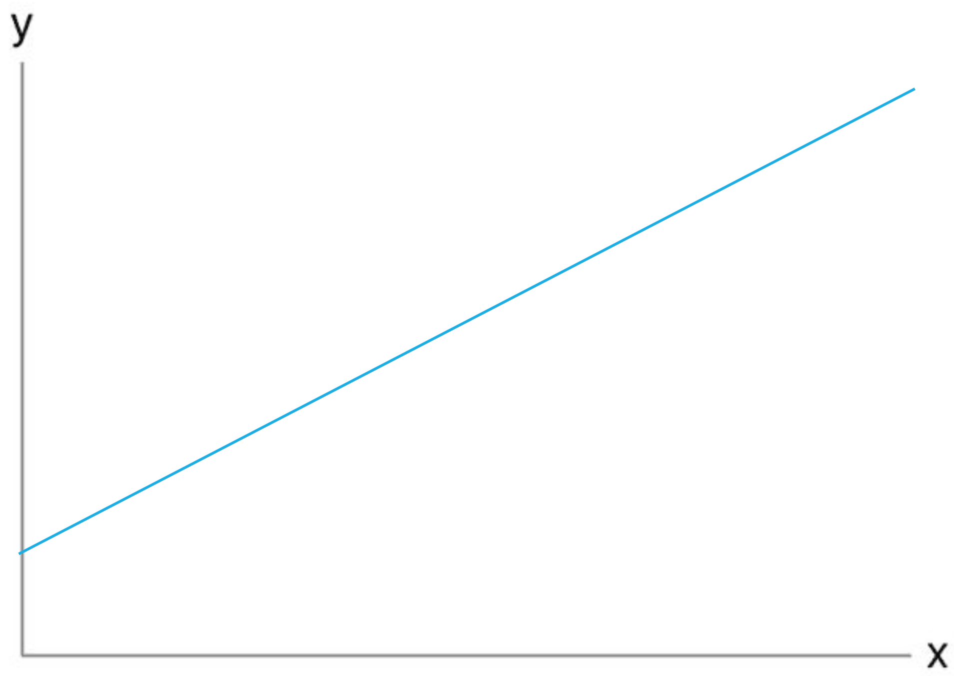
Yes, but also…

Training for task B will require fewer training samples!

Why?

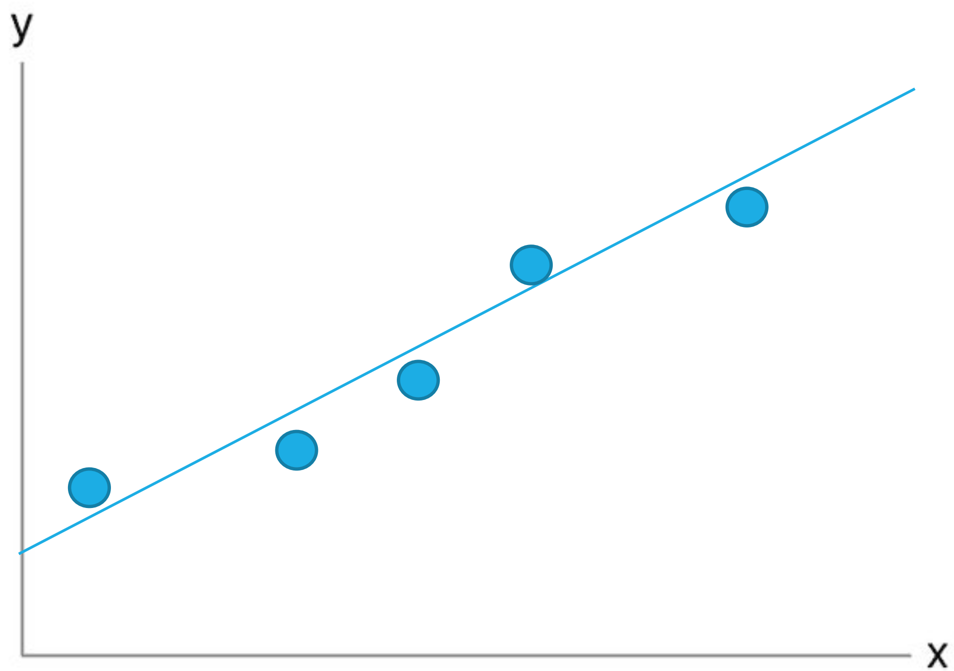
Task A

Let's say the true underlying relationship is this



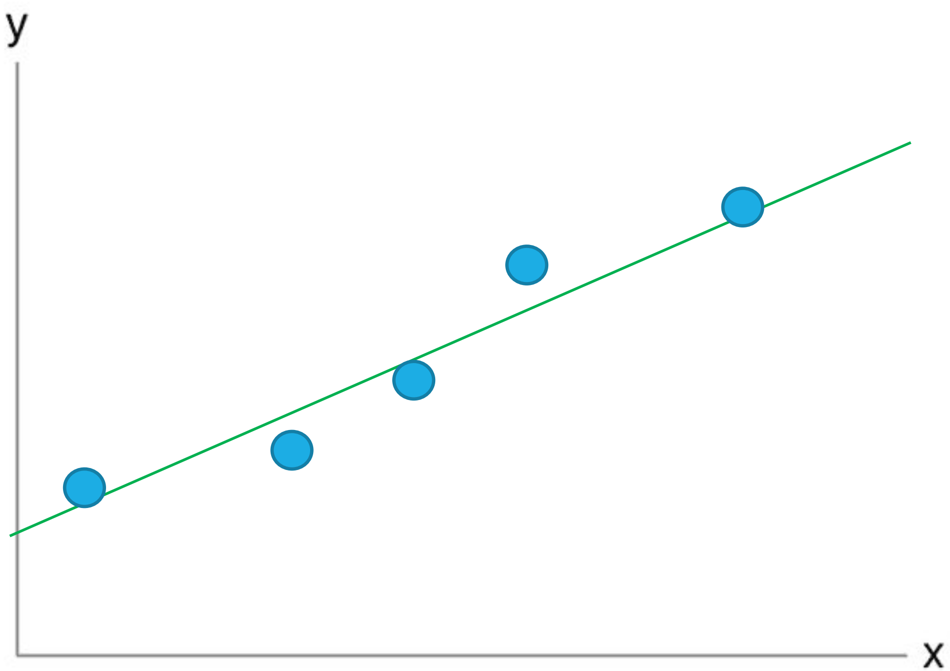
Task A

And we have these training points

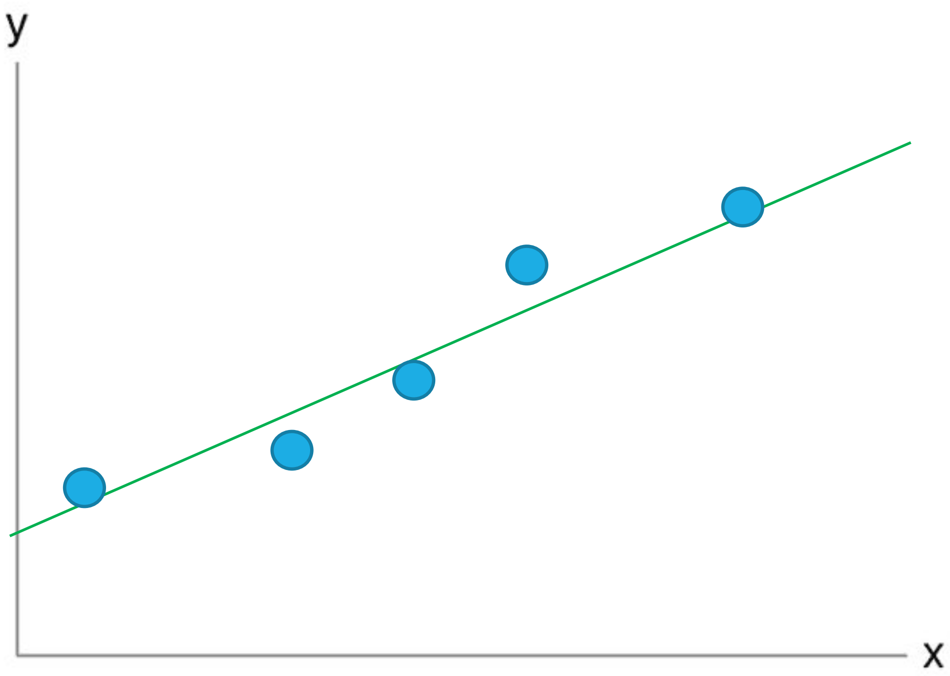


Task A

After training, the following relationship is learned (in green)

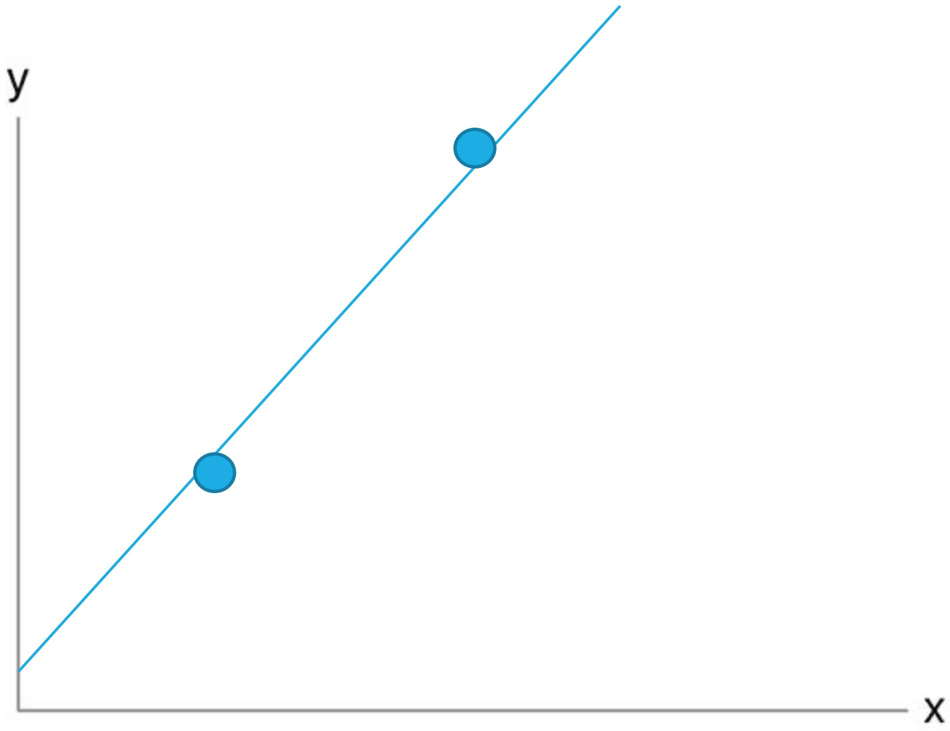


Task A



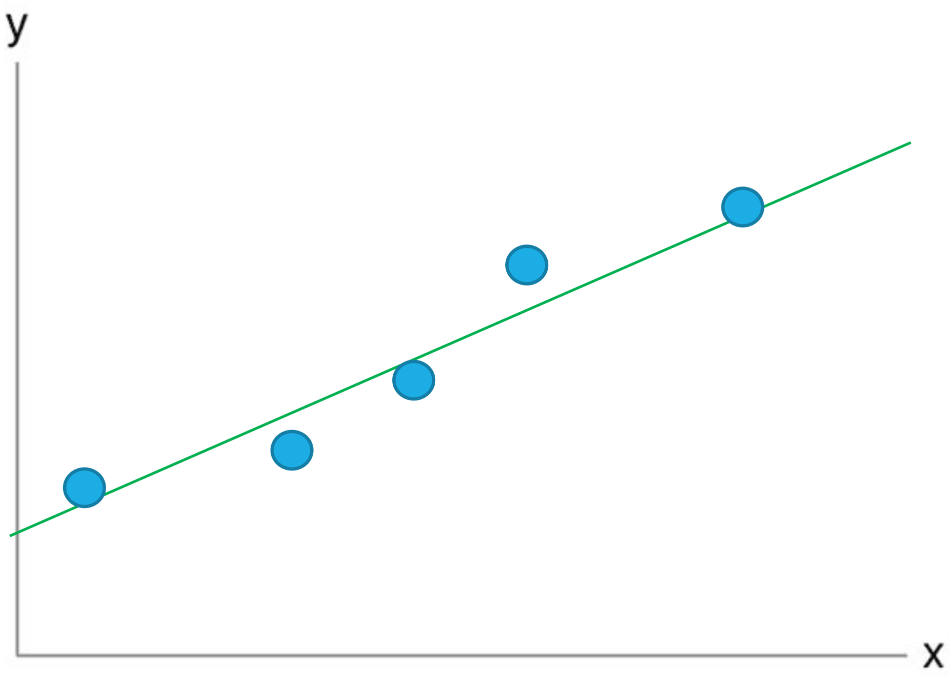
Task B

Let's say this is the true underlying relationship (in blue) with the following observed training points.



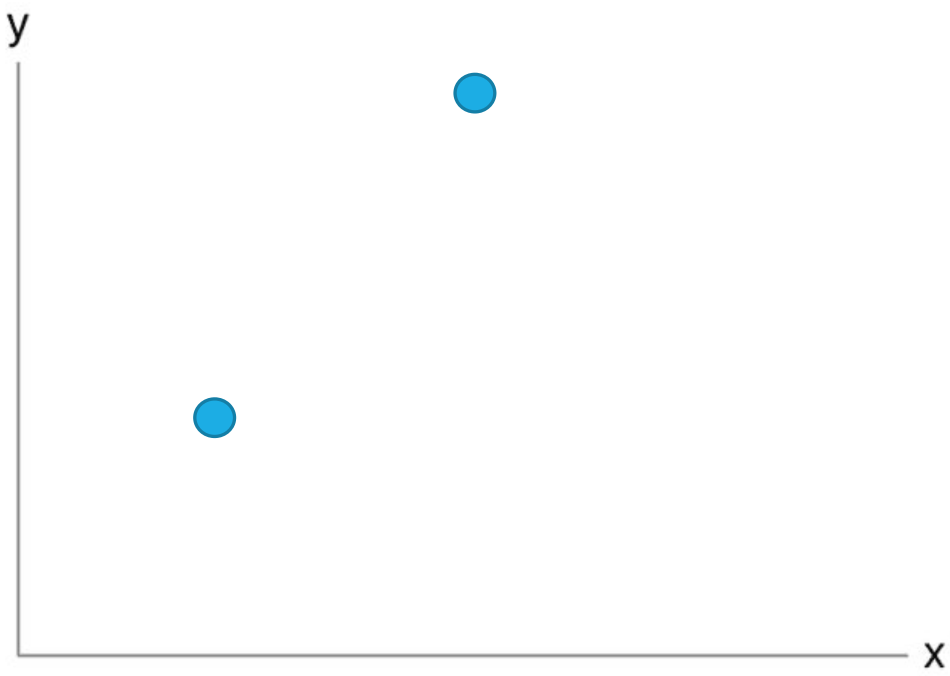
Note that because task B is *related* to task A, the underlying relationships will likely be similar (e.g. in this case, both linear)

Task A

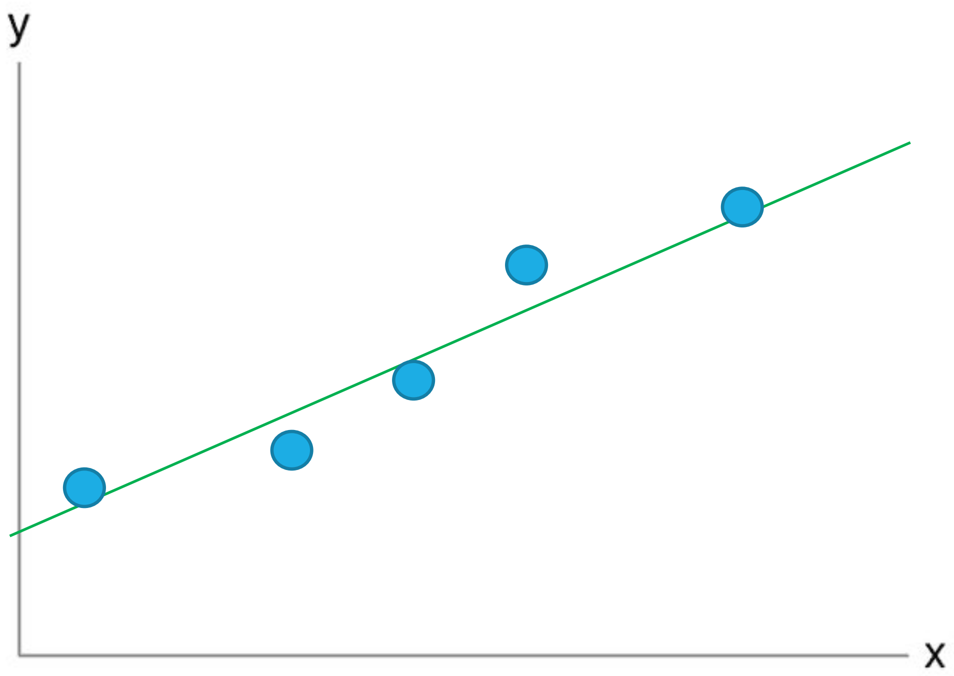


Task B

If we train the model from scratch (starting from a randomized relationship) on these two training points, we can get a huge range of possible learned relationships (**overfitting**)

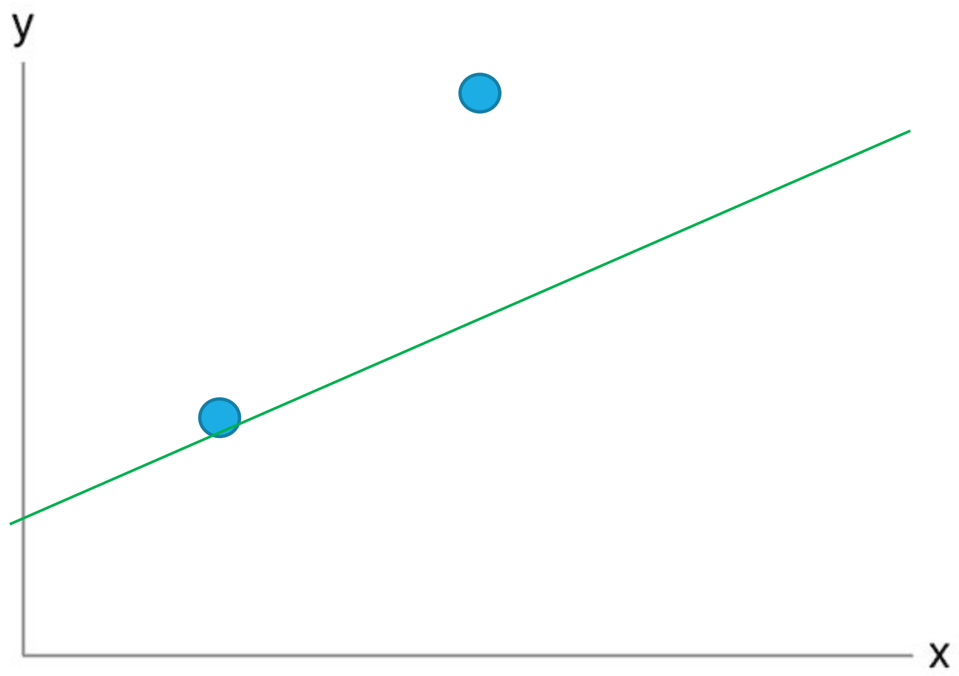


Task A

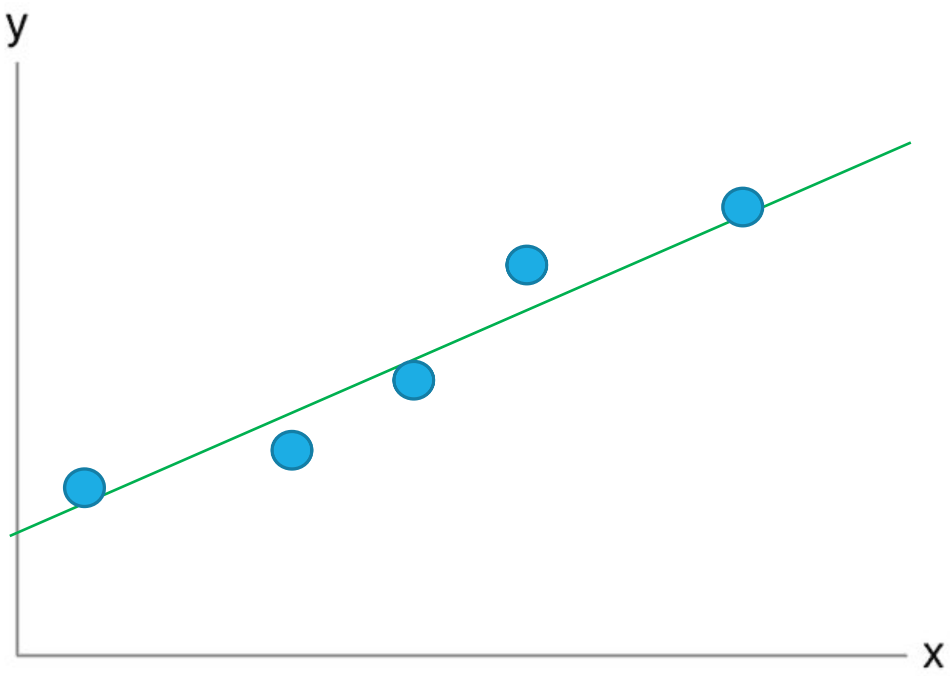


Task B

But, if we *start* with the trained model from Task A, and train from there…

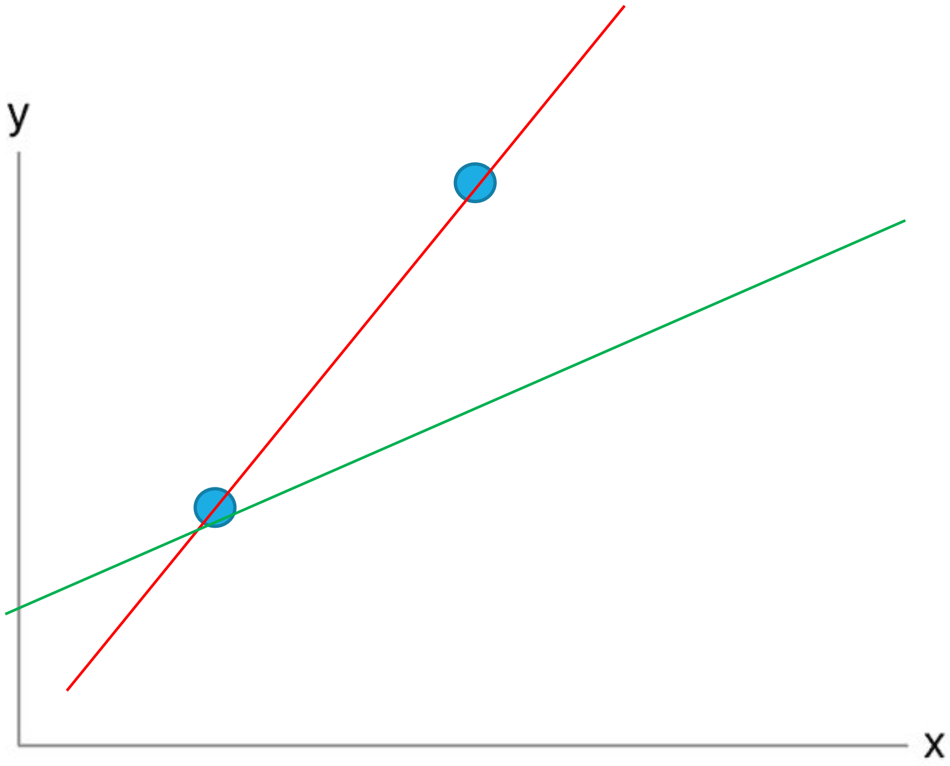


Task A

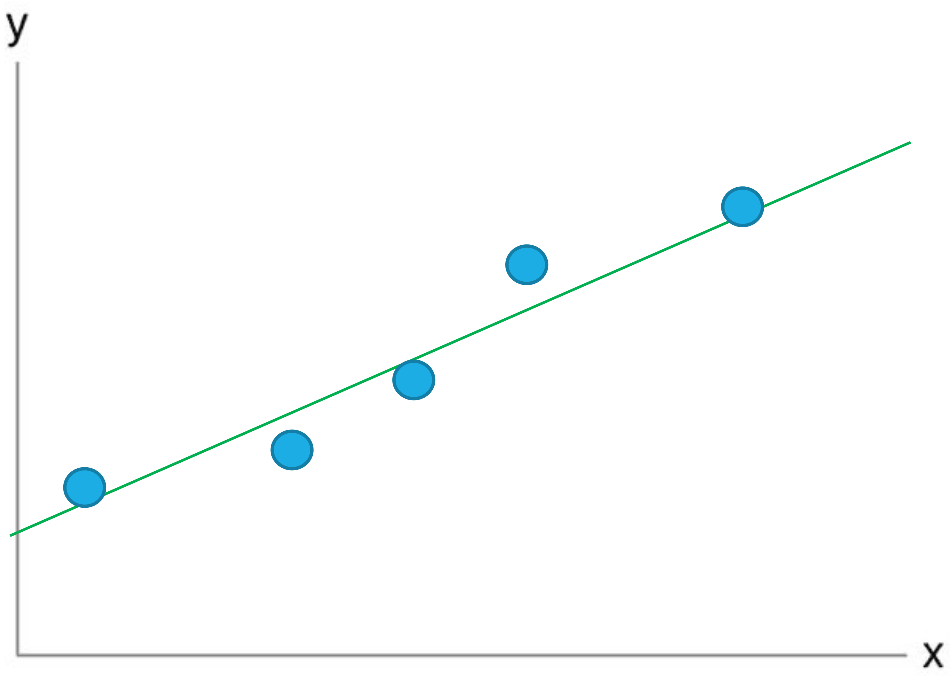


Task B

It's easier for the model to learn this relationship…

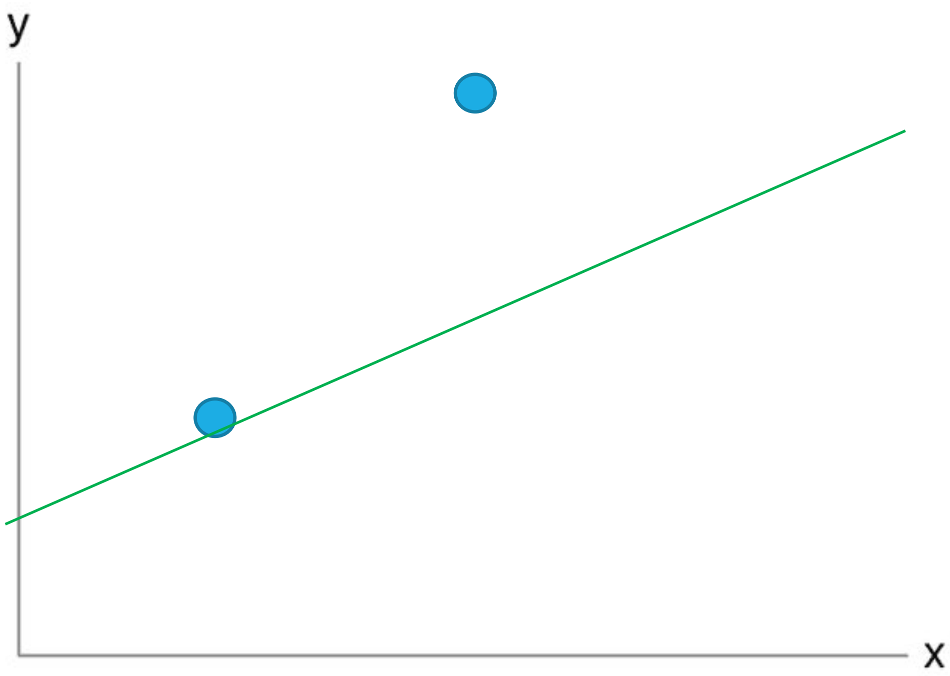


Task A

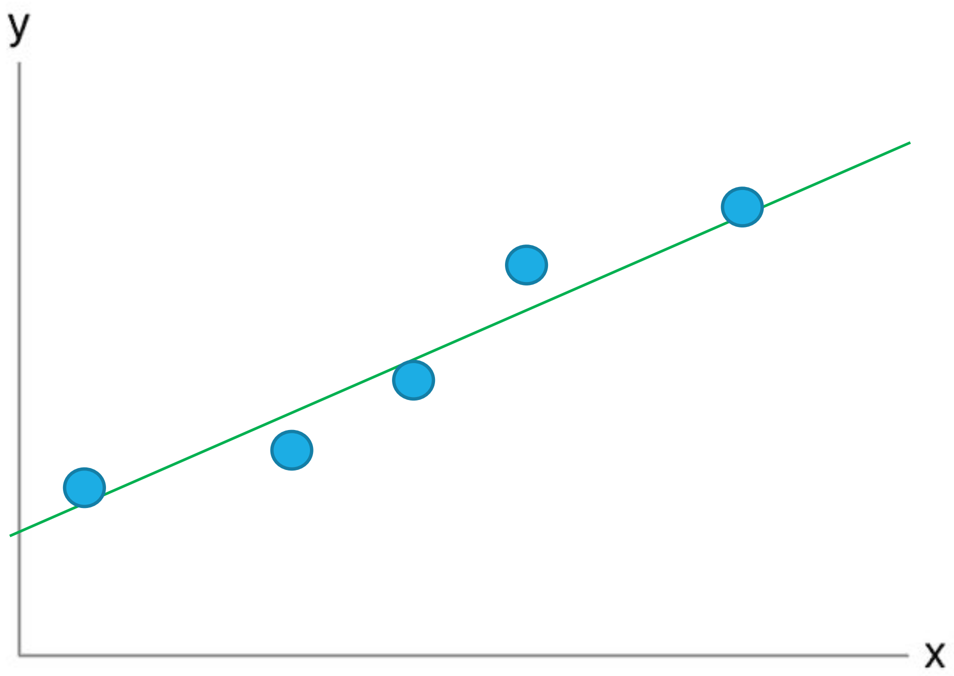


Task B

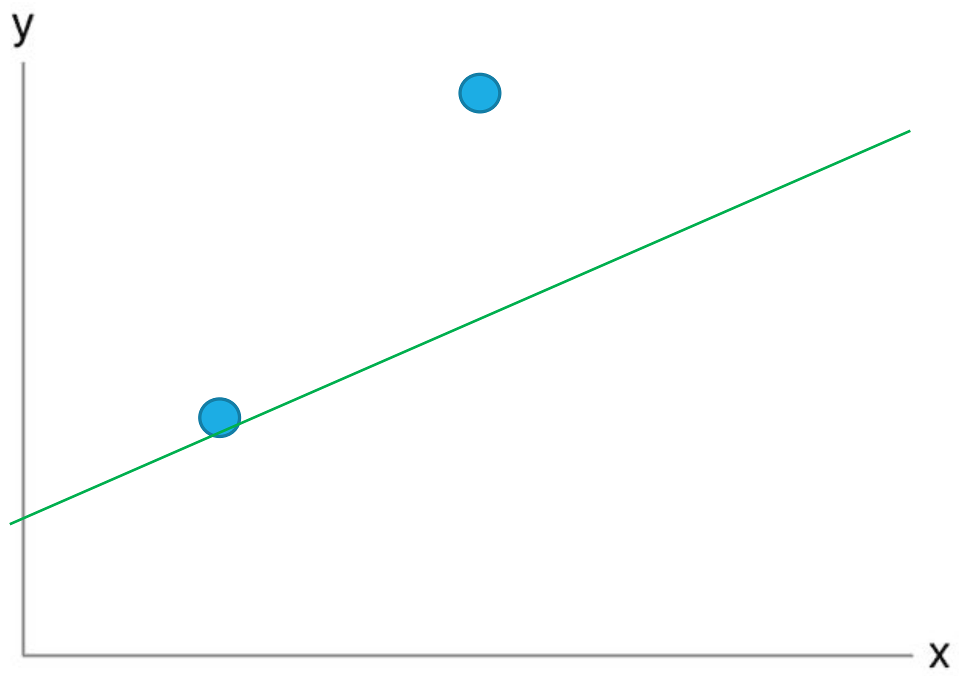
Than this one.



Task A

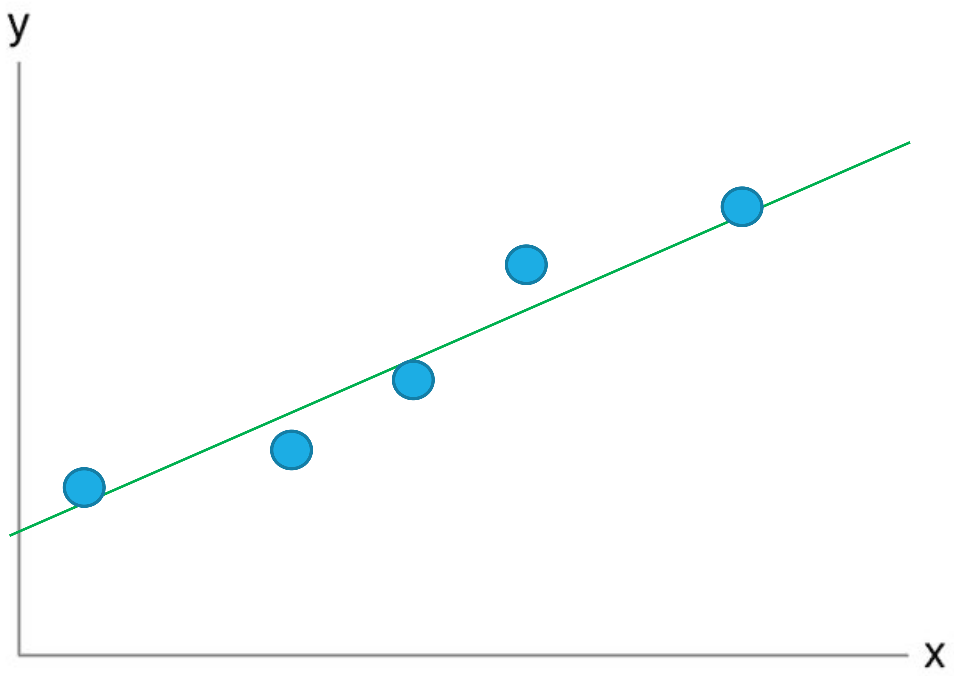


Task B

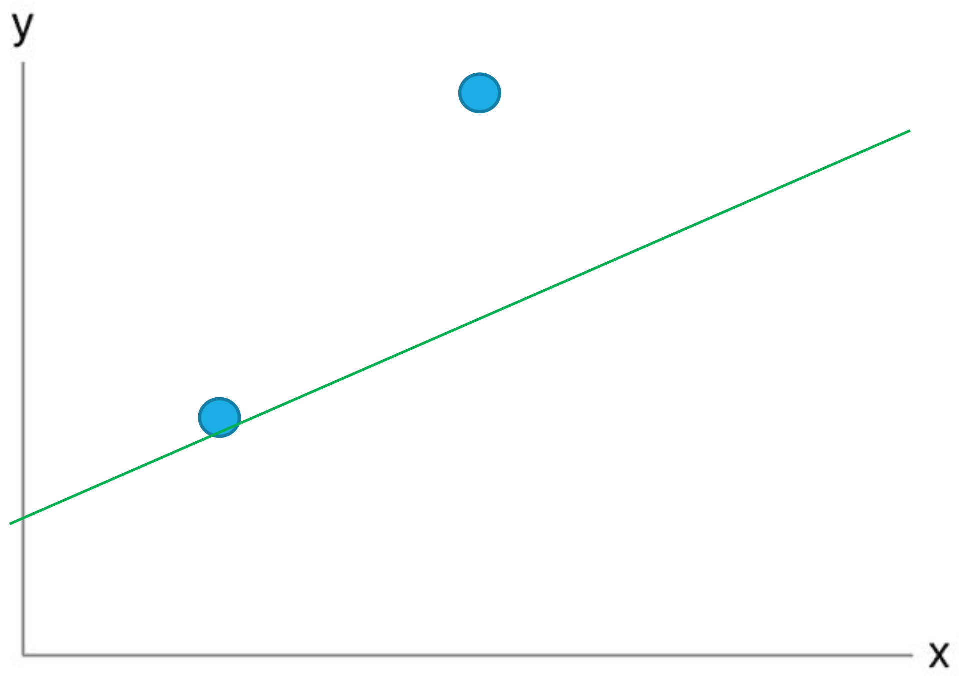


Pre-training with Task A thus imposes a regularizing effect on Task B training
In other words, pre-training **limits** the types of relationships the model can easily learn in Task B

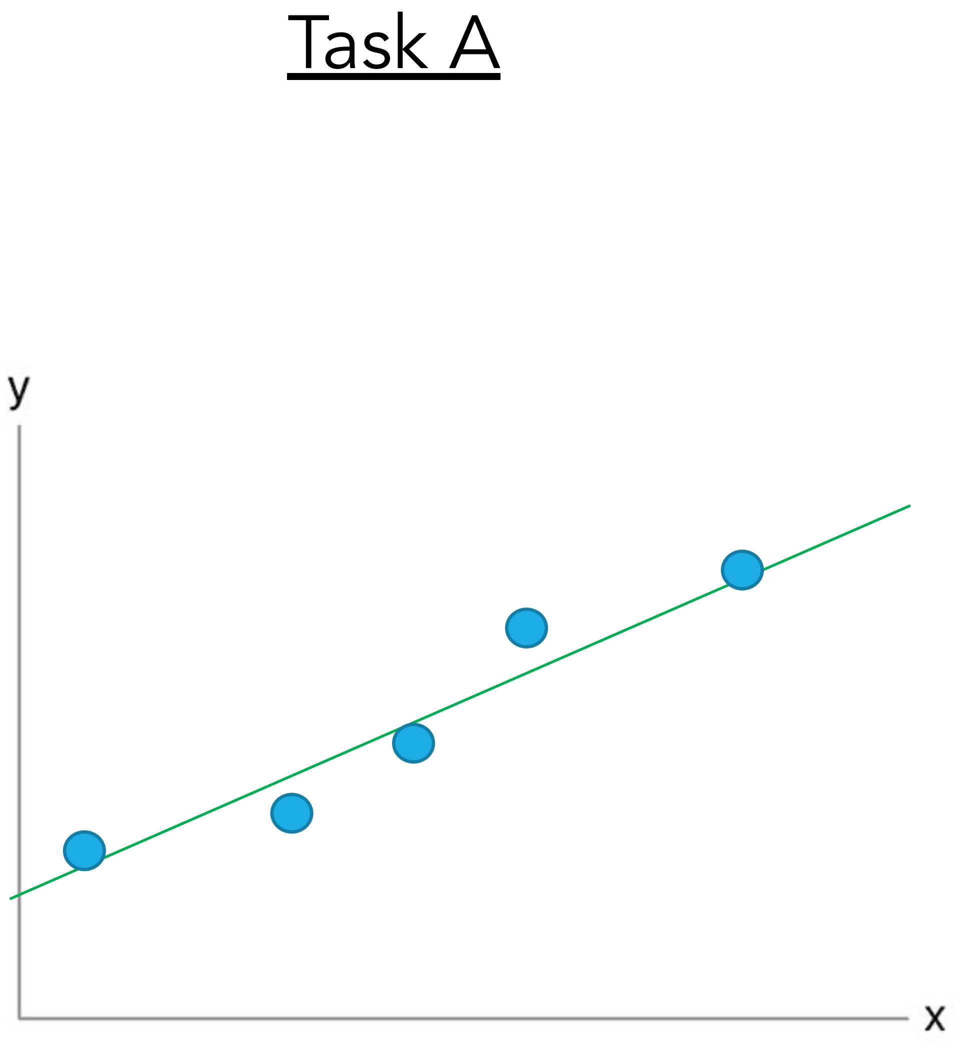
Task A

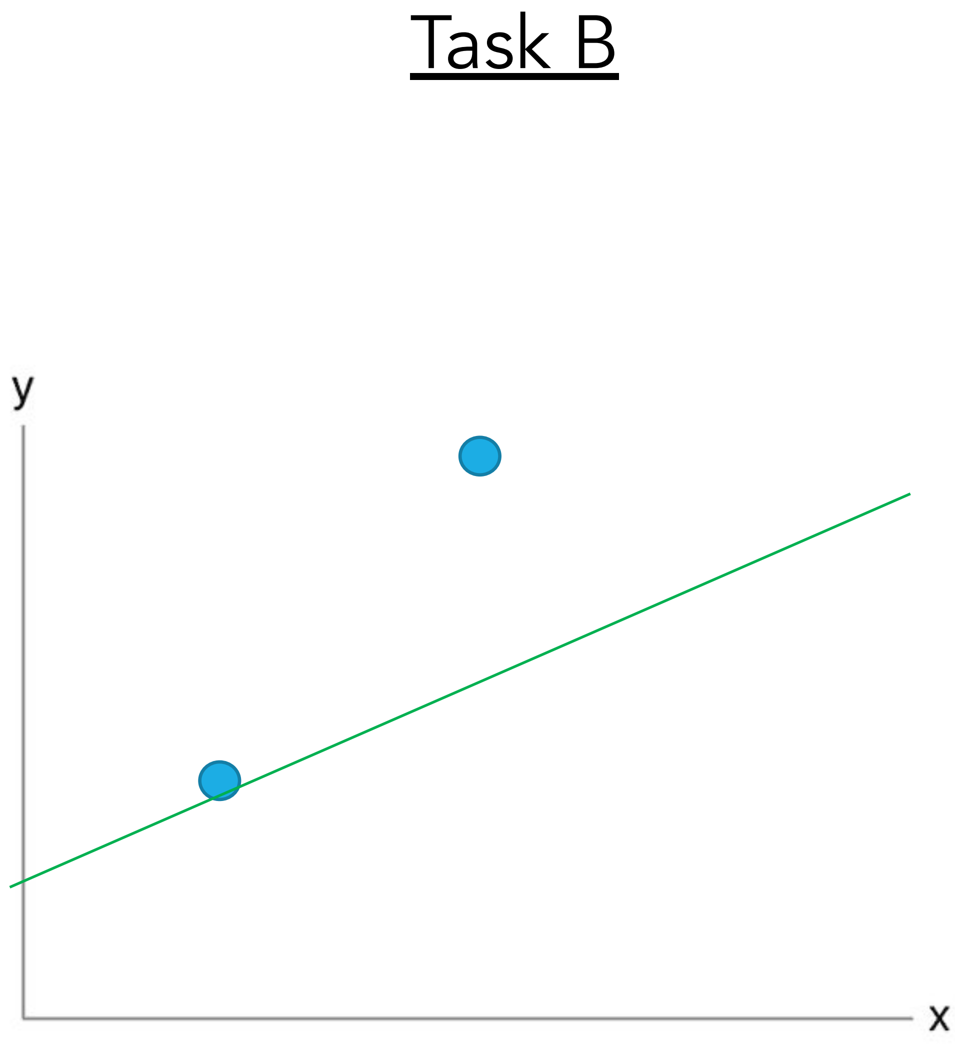


Task B



In this example, since we start Tasb B training from the learned linear relationship in Task A, the learned relationship for Task B is more likely to be linear.





General knowledge transferred over: “this domain of tasks usually has linear relationships”

Pre-training thus mitigates overfitting when the downstream task has low sample size

So, if we want to train a deep neural network on a specific task, but we don't have much training data

(e.g. classifying whether or not an MRI image contains a rare cancer)….

So, if we want to train a deep neural network on a specific task, but we don't have much training data

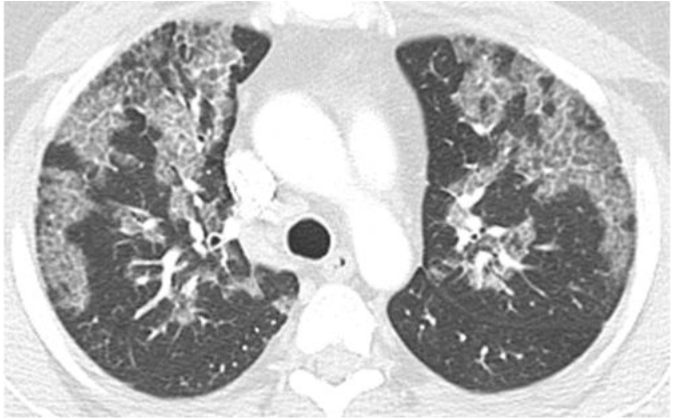
(e.g. classifying whether or not an MRI image contains a rare cancer)....

We pre-train first on a related task that *does* have a lot of training data

(e.g. identifying cat vs dog images, which is still an image classification task)

So, if we want to train a deep neural network on a specific task, but we don't have much training data

(e.g. classifying whether or not an MRI image contains a rare cancer)….



We pre-train first on a related task that *does* have a *lot* of training data

(e.g. identifying cat vs dog images, which is still an image classification task)

Then fine-tune on the target downstream task (might have low sample size).

(e.g. identifying cancer MRIs)

What is transfer learning?

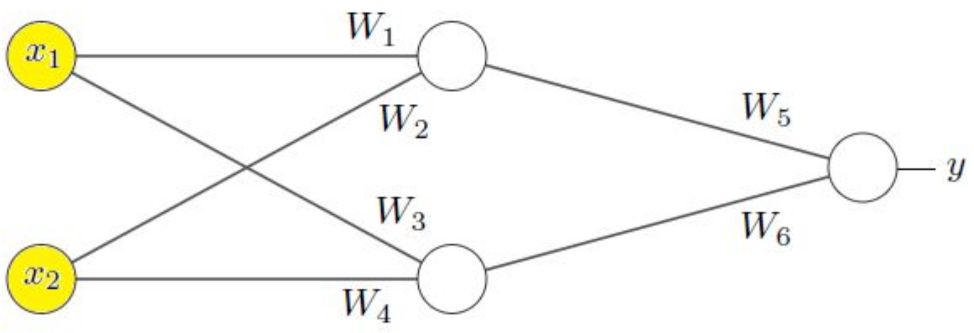
Transfer learning in practice

Bonus: *Guided Transfer Learning*, a novel meta-learning methodology

Saving models in PyTorch: the `state_dict`

```
class NadiaModel(nn.Module):
    def __init__(self):
        super(NadiaModel, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2, 1)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```




```
model = NadiaModel()
model.state_dict()
```

```
OrderedDict([('fc1.weight',
              tensor([[ 0.0734, -0.5261],
                      [-0.3899, -0.3112]])),
             ('fc1.bias', tensor([-0.1867, -0.2358])),
             ('fc2.weight', tensor([[0.4118, 0.6775]])),
             ('fc2.bias', tensor([0.6916]))])
```

The `state_dict` is a Python dictionary that contains the values of all the parameters in the model's current state.

Saving models in PyTorch: `torch.save`

Pre-training:

```
model = NadiaModel()

# ... code to train model ...

torch.save(model.state_dict(), 'pretrained.pt')
```

This saves the state_dict of the trained model as a *.pt* file.

Saving models in PyTorch: `torch.save`

Pre-training:

```
model = NadiaModel()

# ... code to train model ...

torch.save(model.state_dict(), 'pretrained.pt')
```

This saves the state_dict of the trained model as a *.pt* file.

Fine-tuning:

```
downstream_model = NadiaModel()
pretrained_weights = torch.load('pretrained.pt')
pretrained_weights
```

```
OrderedDict([('fc1.weight',
              tensor([[ 0.0734, -0.5261],
                      [-0.3899, -0.3112]])),
             ('fc1.bias', tensor([-0.1867, -0.2358])),
             ('fc2.weight', tensor([[0.4118, 0.6775]])),
             ('fc2.bias', tensor([0.6916]))])
```

Then you can read this .pt file into a Python dictionary in your downstream model training file…

Saving models in PyTorch: `torch.save`

Pre-training:

```
model = NadiaModel()

# ... code to train model ...

torch.save(model.state_dict(), 'pretrained.pt')
```

This saves the state_dict of the trained model as a *.pt* file.

Fine-tuning:

```
downstream_model = NadiaModel()
pretrained_weights = torch.load('pretrained.pt')
pretrained_weights
```

```
OrderedDict([('fc1.weight',
              tensor([[ 0.0734, -0.5261],
                      [-0.3899, -0.3112]])),
             ('fc1.bias', tensor([-0.1867, -0.2358])),
             ('fc2.weight', tensor([[0.4118, 0.6775]])),
             ('fc2.bias', tensor([0.6916]))])
```

Then you can read this .pt file into a Python dictionary in your downstream model training file…

```
downstream_model.load_state_dict(pretrained_weights)
```

```
<All keys matched successfully>
```

…and load those pre-trained weights into your downstream model for fine-tuning!

Saving models in PyTorch: `torch.save`

Pre-training:

```
model = NadiaModel()

# ... code to train model ...

torch.save(model.state_dict(), 'pretrained.pt')
```

Fine-tuning:

```
downstream_model = NadiaModel()
pretrained_weights = torch.load('pretrained.pt')
pretrained_weights
```

```
OrderedDict([('fc1.weight',
              tensor([[ 0.0734, -0.5261],
                      [-0.3899, -0.3112]])),
             ('fc1.bias', tensor([-0.1867, -0.2358])),
             ('fc2.weight', tensor([[0.4118, 0.6775]])),
             ('fc2.bias', tensor([0.6916]))])
```

```
finetuning_model.load_state_dict(pretrained_weights)
```

```
<All keys matched successfully>
```

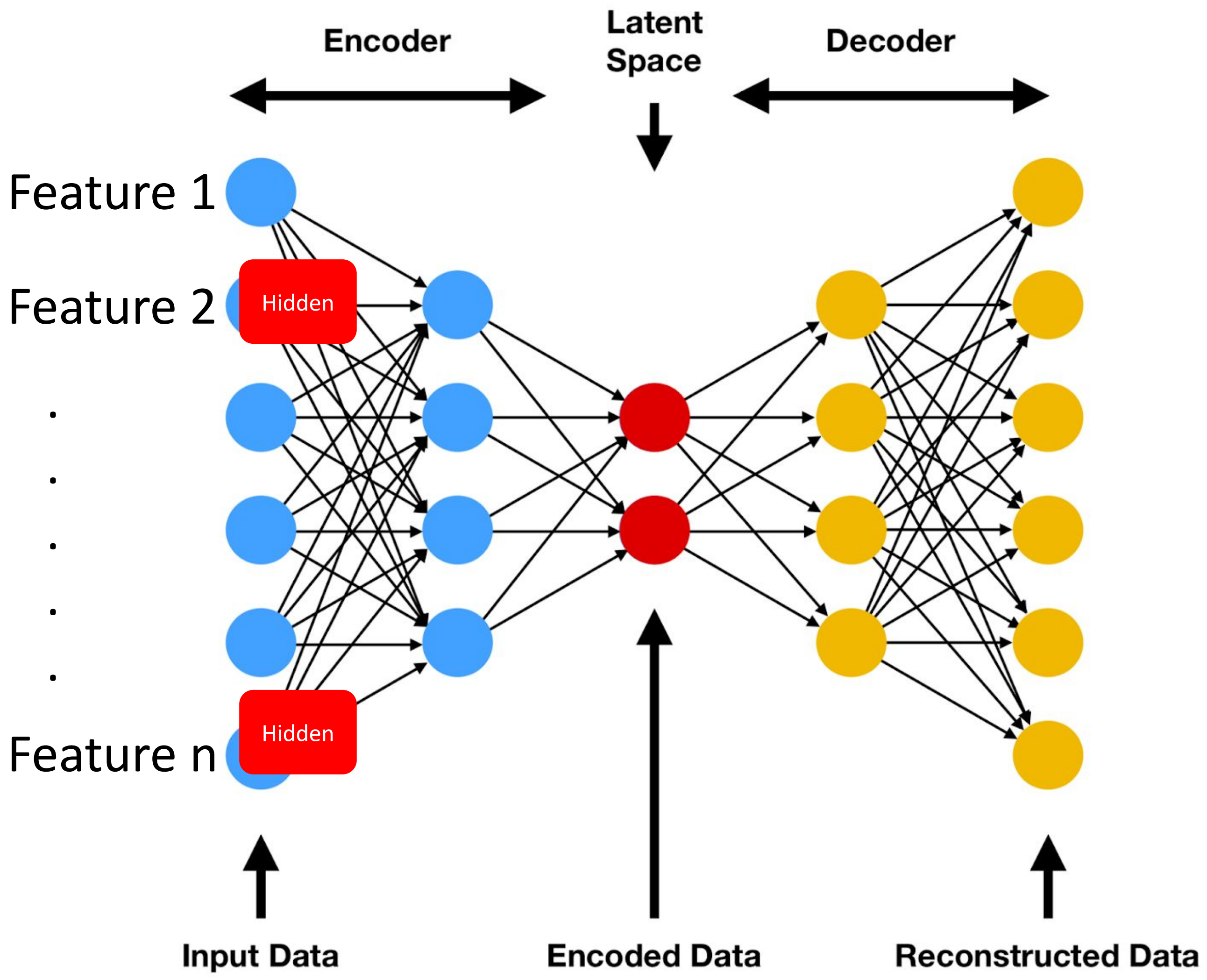
Note that the model architecture for the upstream and downstream models must be the same so that the weights are transferrable!

What if there is no large, *labeled* pre-training dataset available for the domain of tasks I want to work with?

That's okay. We can pre-train with large amounts of *unlabeled* data (self-supervised learning),
and fine-tune on *labeled* data (supervised learning)

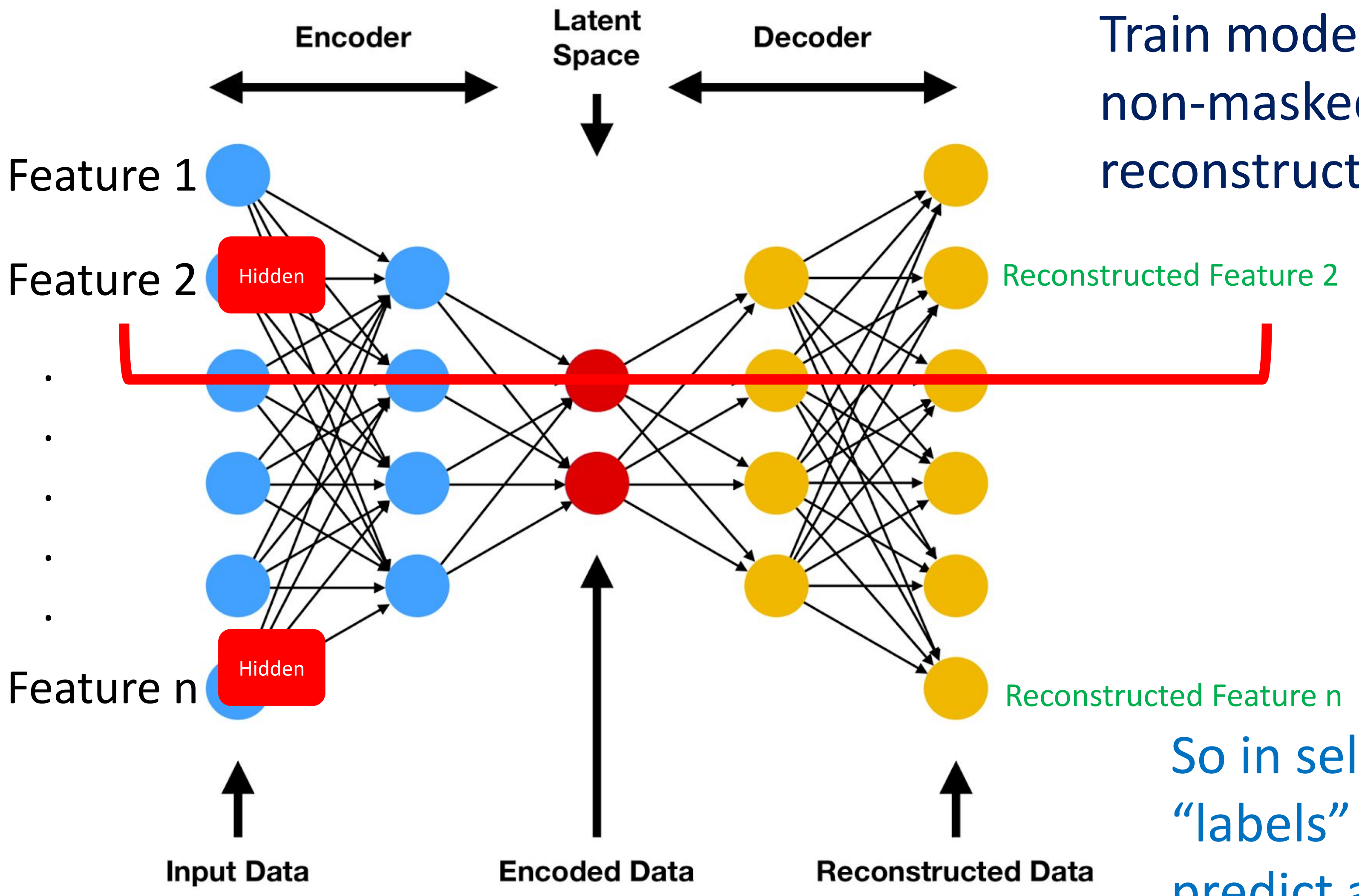
How does self-supervised learning work?

Self-supervised Pretraining



Randomly mask (hide) values of some features in the input data

Self-supervised Pretraining

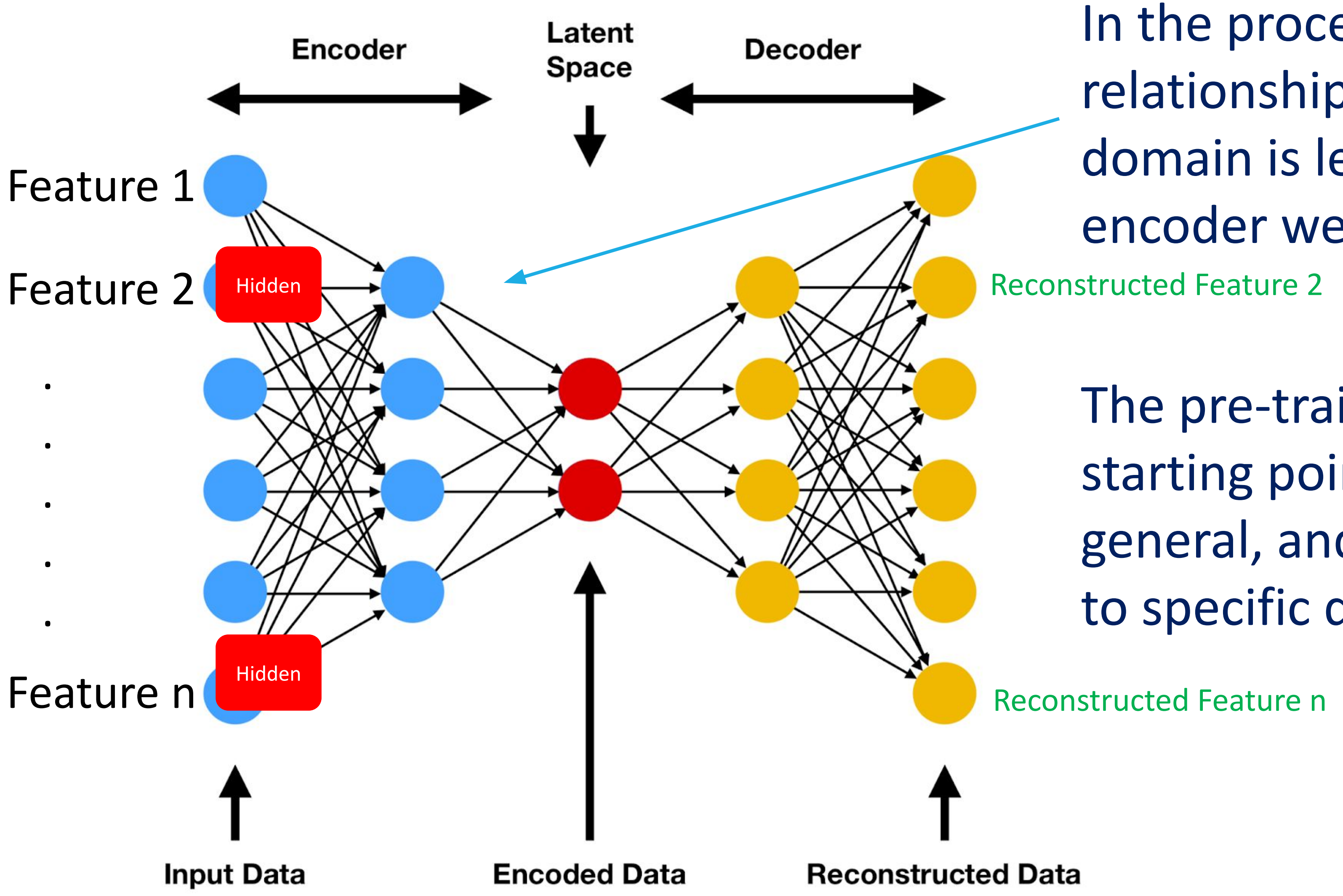


Train model to use the values of other, non-masked feature values to reconstruct the masked values

The goal is to minimize the difference between the reconstructed and original (hidden) expression values

So in self-supervised learning, the "labels" the model is trying to predict are just the original input feature values!

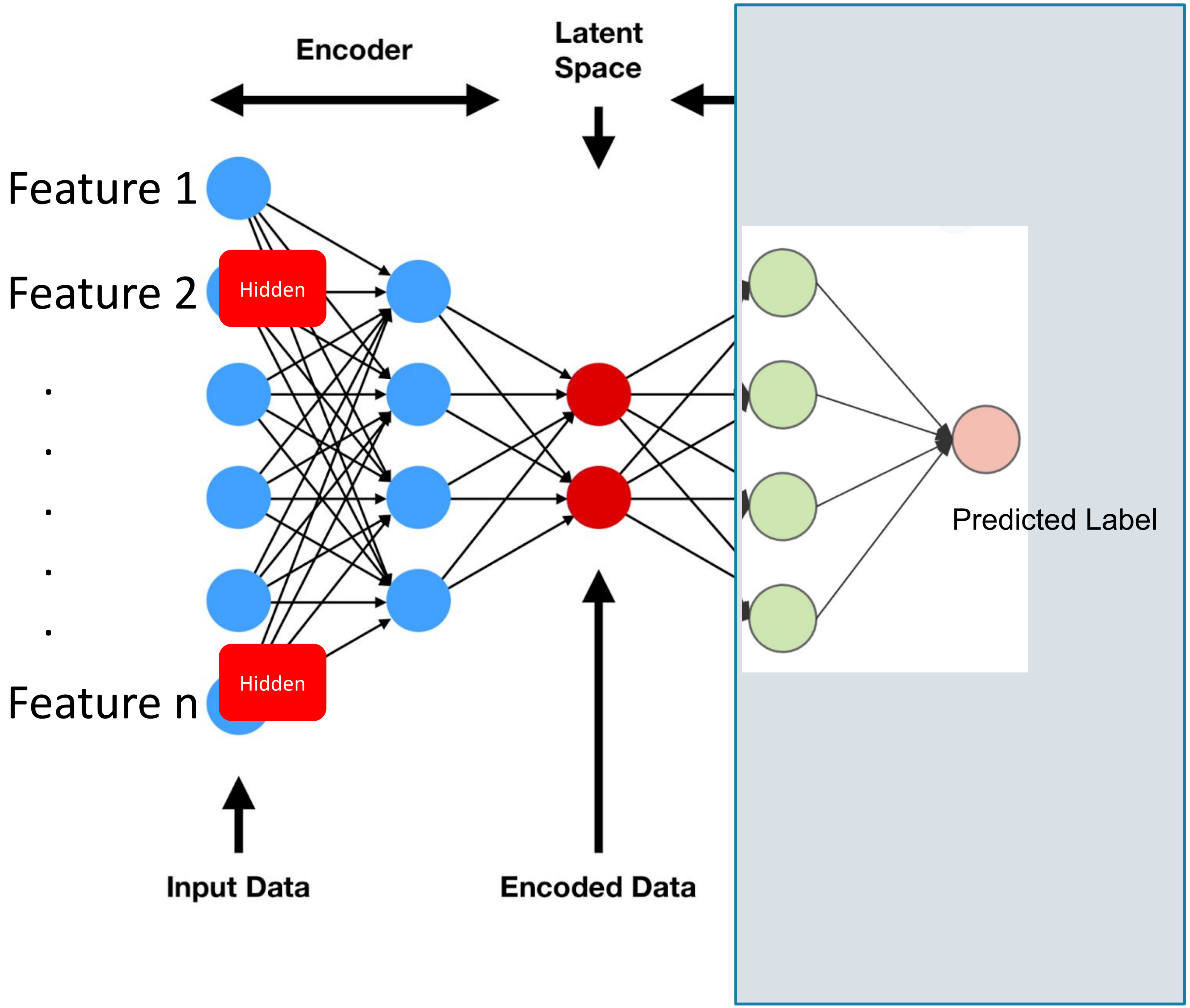
Self-supervised Pretraining



In the process, general knowledge about relationships between features in this domain is learned and stored in the encoder weights

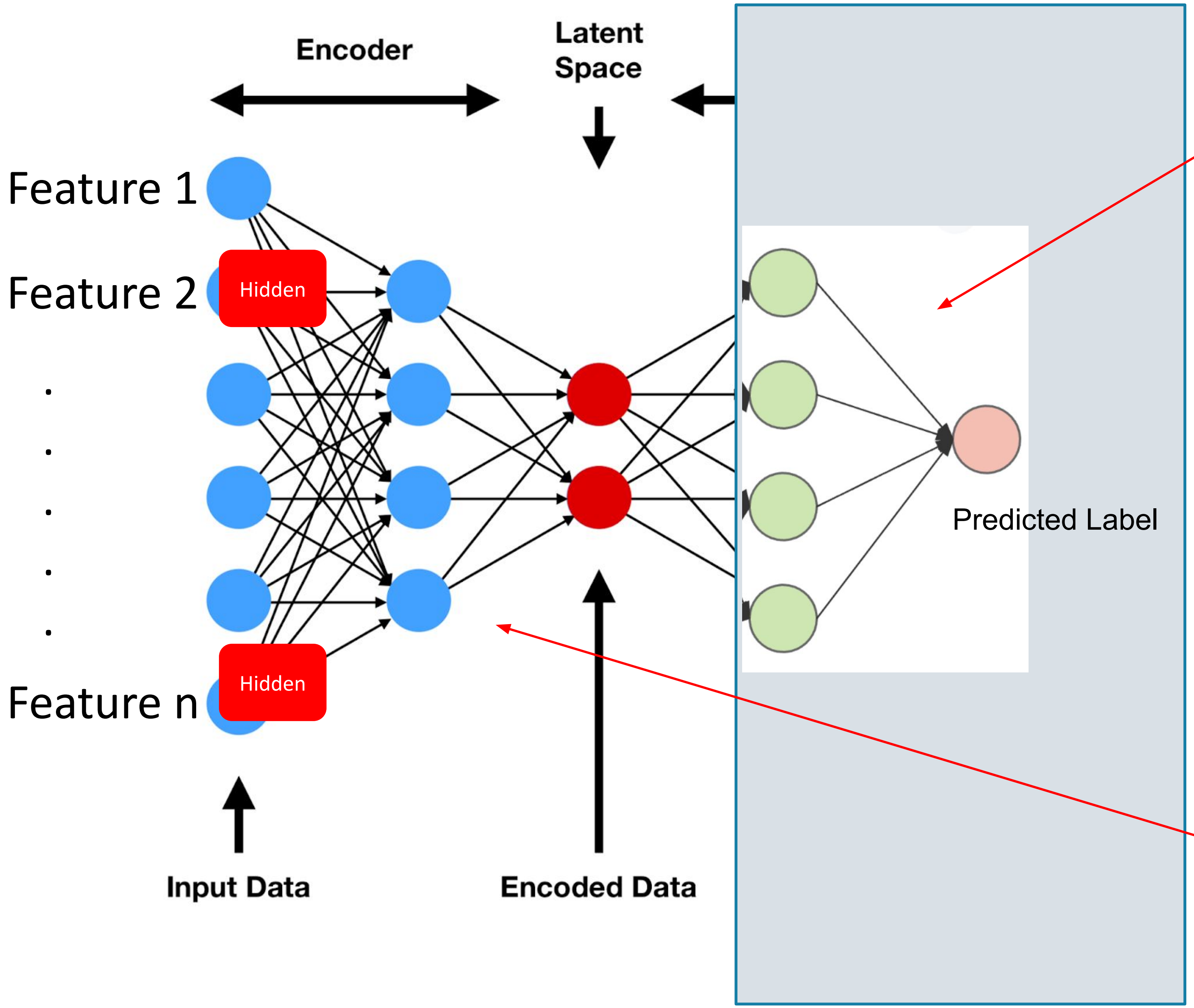
The pre-trained weights are a good starting point for tasks of this domain in general, and they can then be fine-tuned to specific downstream tasks

Supervised fine-tuning



When we fine-tune the model to a *supervised* task, we replace the decoder/reconstructor portion of the architecture with a prediction layer

Supervised fine-tuning

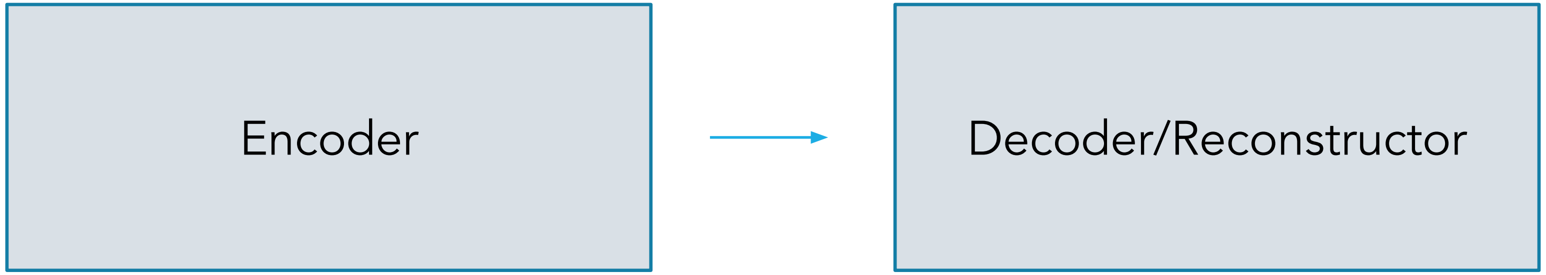


The newly added prediction layer has no pre-trained weights, which is fine because it is specific to the particular downstream task

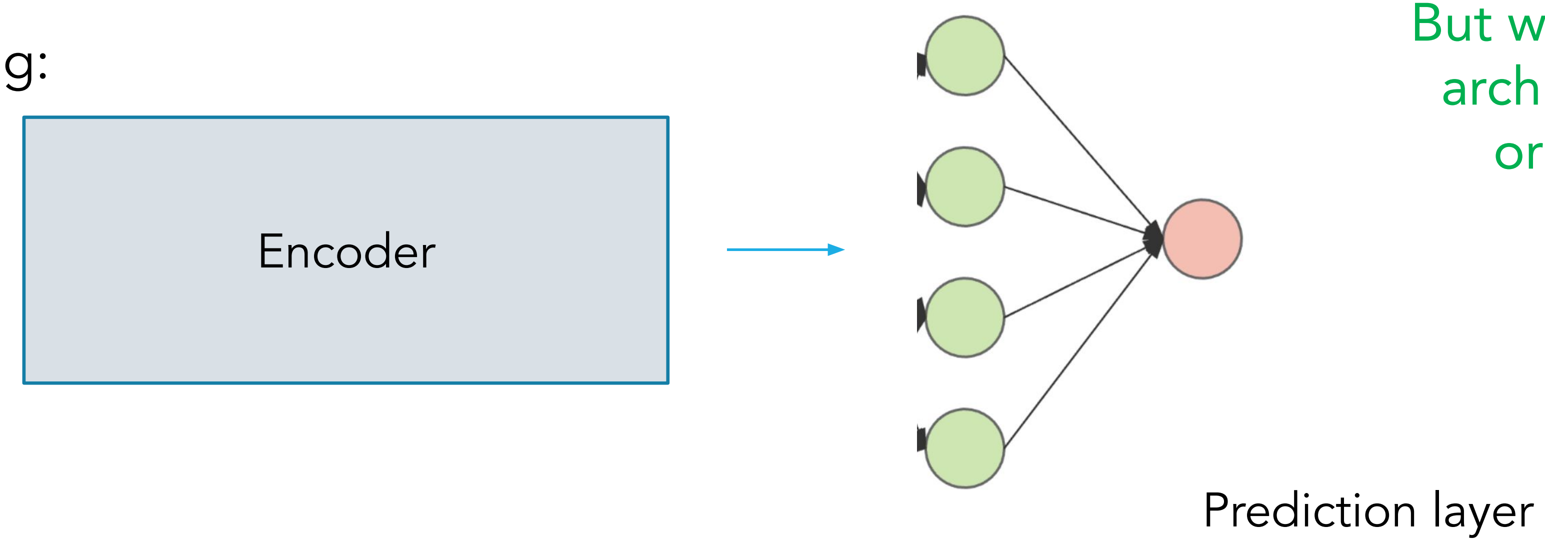
General knowledge is stored in the pre-trained weights of the encoder portion.

Wait, there's now a mismatch between the architecture used for pre-training and the architecture used for fine-tuning…

Pre-training:



Fine-tuning:



But we said we needed the architectures to match in order to transfer the state_dict!

Keeping separate state_dicts for the encoder and decoder

```
class Encoder(nn.Module):
    ## code for Encoder architecture

class Decoder(nn.Module):
    ## code for Decoder architecture

class Autoencoder(nn.Module):
    ## full encoder-decoder architecture
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        latent = self.encoder(x)
        x_recon = self.decoder(latent)
        return x_recon
```

We have separate classes for the Encoder and Decoder parts of the model.

Then connect the two together in our full model (but they each still have their own, individual state_dicts).

```
model = Autoencoder()

# ... code to train model ...

torch.save(model.encoder.state_dict(), 'pretrained_encoder.pt')
```

We only save the state_dict for `model.encoder` after pre-training!

Keeping separate state_dicts for the encoder and decoder

Fine-tuning:

```
class DownstreamModel(nn.Module):
  def __init__(self):
      super(DownstreamModel, self).__init__()
      self.encoder = Encoder()

      ## Prediction layers
      self.fc1 = nn.Linear(...)
      self.relu = nn.ReLU()
      self.fc2 = nn.Linear(...)

  def forward(self, x):
      ## code for forward pass
```

Downstream model uses the same encoder architecture, but uses some fully connected layers for prediction instead of the decoder

```
downstream_model = DownstreamModel()
pretrained_encoder = torch.load('pretrained_encoder.pt')
downstream_model.encoder.load_state_dict(pretrained_encoder)
```

We only load the pre-trained encoder state_dict into `downstream_model.encoder`

Now where do I go find a giant dataset for pre-training?

If the target downstream tasks are in this domain:	*Consider pre-training with these datasets:*
`Image Data`	• **CIFAR-100**: 100 classes of labeled images with 600 samples each (`torchvision.datasets.CIFAR100`) • **ImageNet:** 1000 classes, 1mil samples (`torchvision.datasets.ImageNet`)
`Text Data`	**(Too many)** • **Project Gutenberg** • **WordNet** • **Yelp Reviews** • **UCI Spambase** • **Sentiment140** • **IMDb Movie Reviews**
`RNA-seq Data`	• **recount3:** 750,000 uniformly processed mouse and human RNA-seq samples

Surely someone has already gone through the trouble to pre-train some pretty good models… can I just use theirs?

If the target downstream tasks are in this domain:	*Consider using the following, state-of-the-art (SOTA) pre-trained models*
`Image Data`	• **VGG19** • **Inceptionv3 (GoogLeNet)** • **ResNet50** • **EfficientNet**
`Text Data`	• **BERT** variations • **GPT** series • **ELMo** variations
`RNA-seq Data`	• **scBERT**

What is transfer learning?

Transfer learning in practice

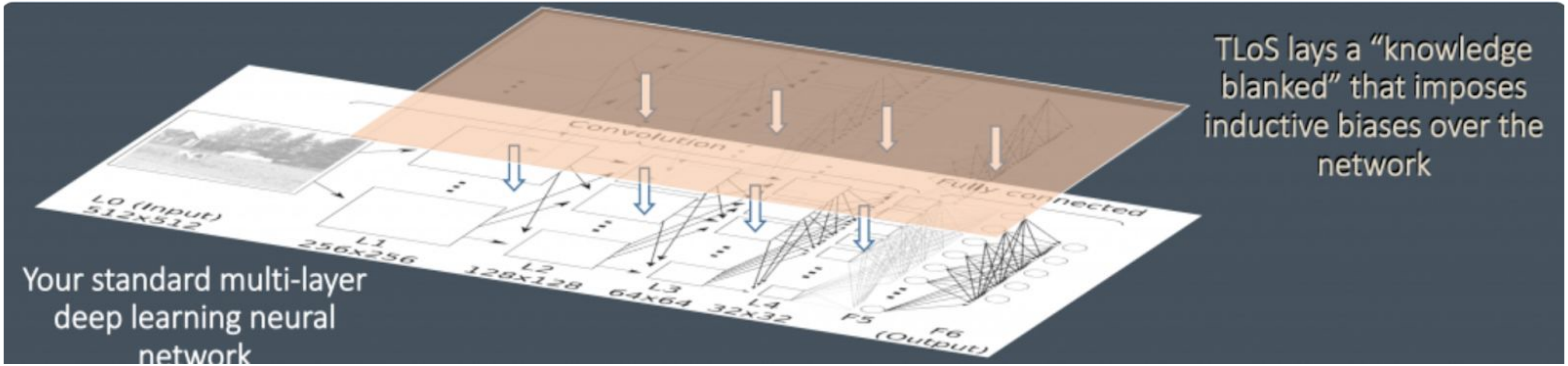
Bonus: *Guided Transfer Learning,*
a novel meta-learning methodology

One Level Further – Guided Transfer Learning: Learning How to Learn

- Developed by Robots Go Mental

- **Main Idea:** We don't want the AI to just learn general prior knowledge during pre-training, we also want it learn **how** to learn new knowledge from the domain *more efficiently*

- During pre-training, learn inductive biases (which affect future learning) in addition to parameter weights

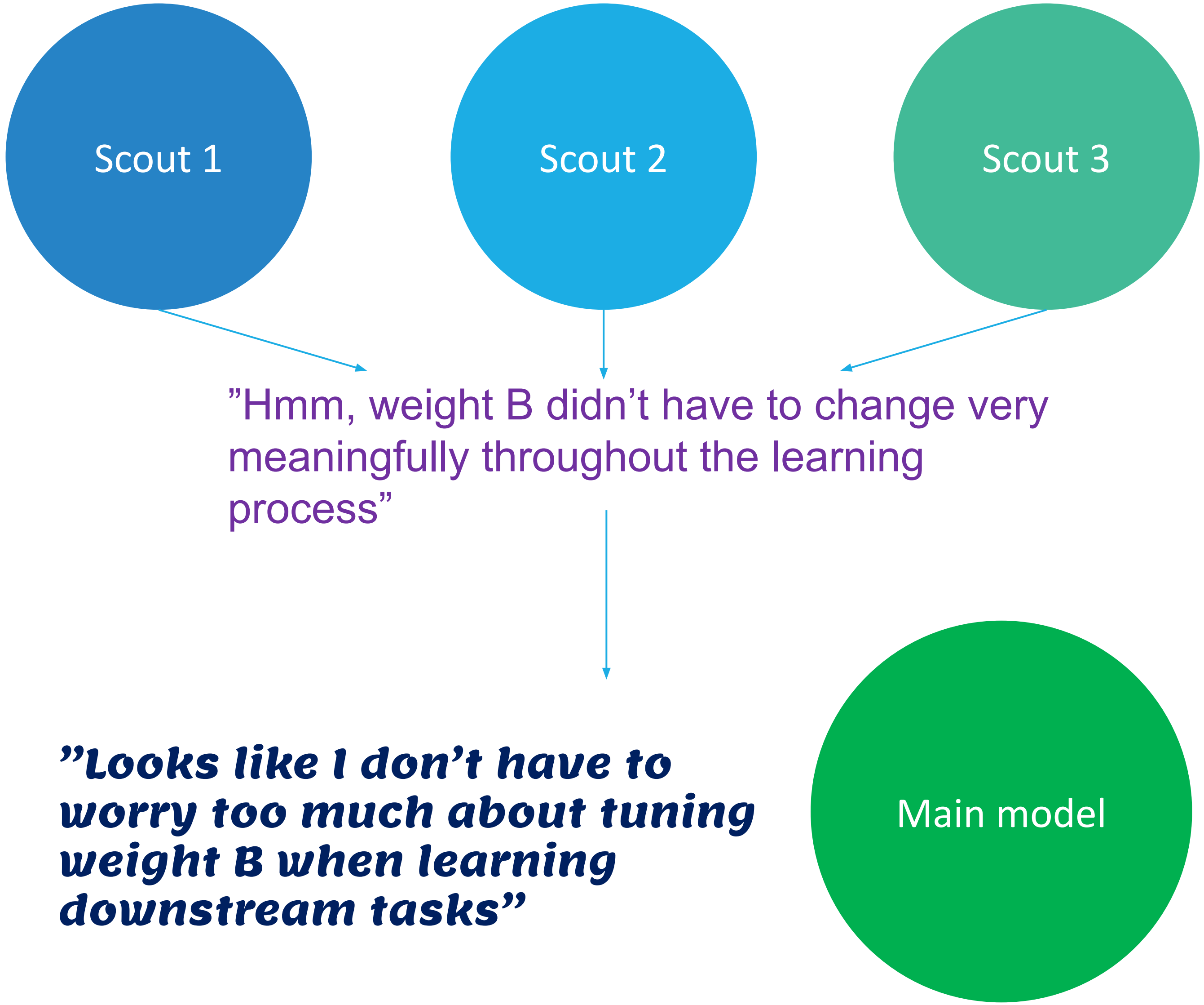


Guided Transfer Learning: Scouting

Send “scout” models to learn easier subproblems with the pre-training dataset

Scouts reflect on the learning process

Inductive biases imposed on main model based on scout learning process



How are these “inductive biases” represented?

After scouting, every weight will have a corresponding guide value

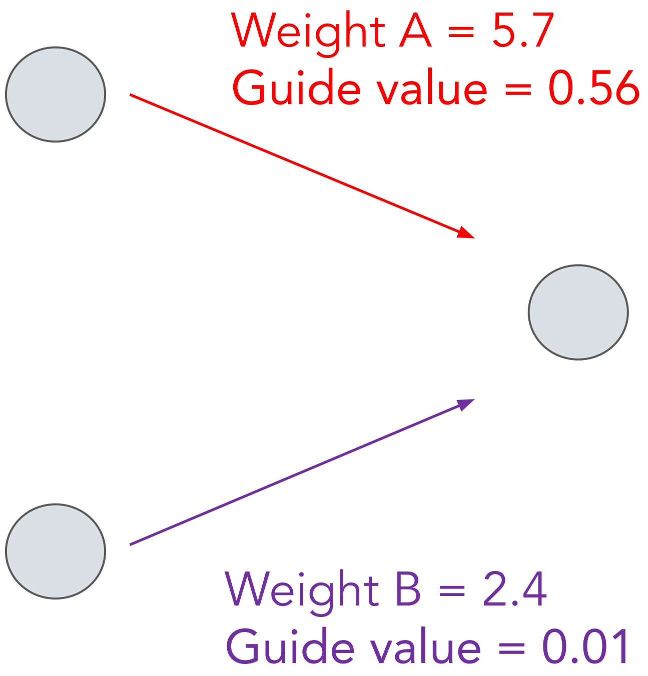
update for weight = normal gradient descent update for weight × guide value for weight

The larger the guide value, the more “flexible” that weight is (the more it is allowed to be updated

How are these “inductive biases” represented?

After scouting, every weight will have a corresponding guide value

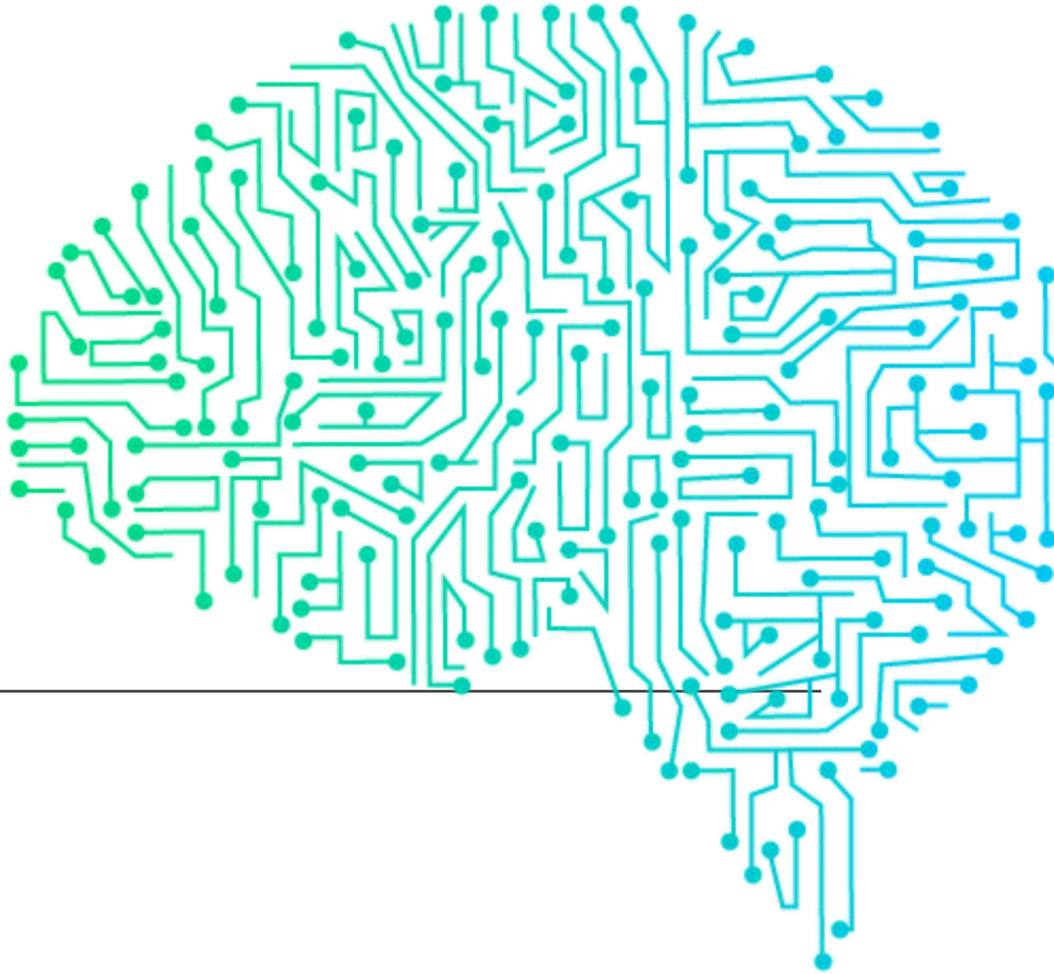
update for weight = normal gradient descent update for weight × guide value for weight



In this example, weight A is more flexible than weight B.

Weight B is pretty much frozen (barely allowed to update at all)

Transfer Learning



CS 175