CHECK-IN

INTRODUCTION TO MACHINE LEARNING (PART 4 OF 5)

MONDAY, JANUARY 22 AT 2:00PM

https://cglink.me/2gj/c19387711116154454

1. **Open** the **My PrincetonU** app.
2. Select a Hub
3. Click on QR Code scanner.
4. Scan this QR Code and you are checked-in!

Download on the App Store

GET IT ON Google Play

Day 5 Hackathon

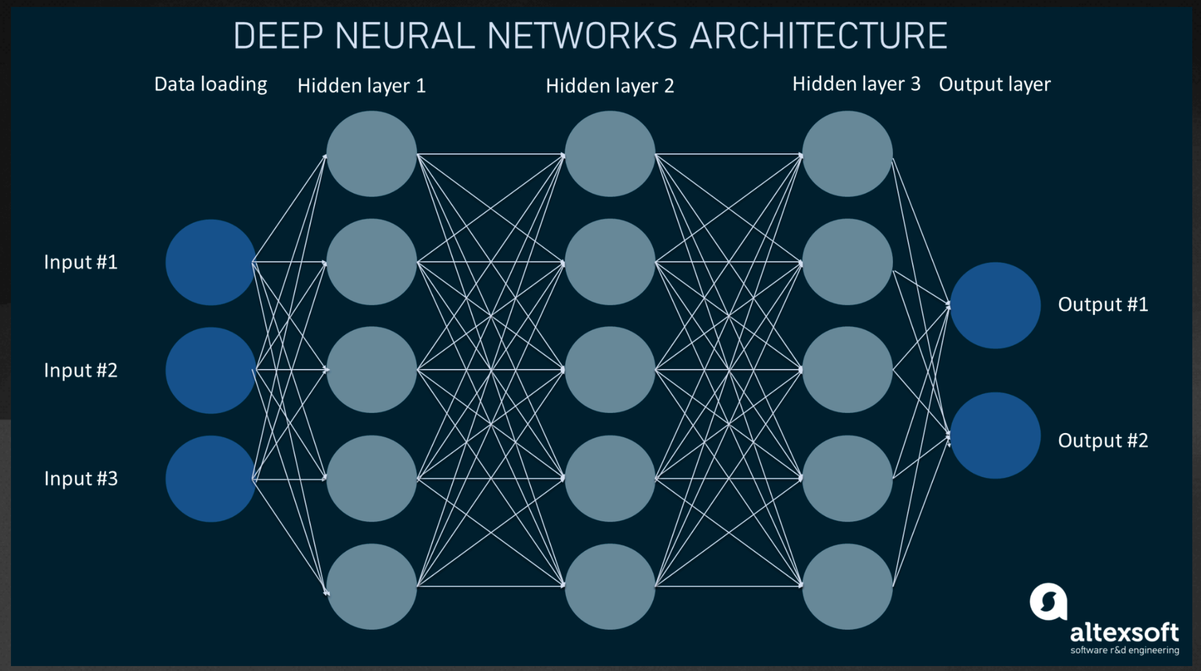
Project Descriptions

- **Computer Vision**: Learn more about CNNs, classify dogs versus cats using a simple CNN, and use transfer learning with an advanced CNN (ResNet-50) to classify dogs versus cats.
- **Diffusion Models**: Learn about diffusion models (e.g., DALL-E 2) then build one and train a generative model for images.
- **Large Language Models**: This session introduces the basics of language modeling using the transformer architecture. Participants will learn how to download and fine-tune an LLM using the Hugging Face library.

Deep Learning Architectures

Introduction to Machine Learning, Day 4

Gage DeZoort | 1/22/24



- Forward pass:

$$z_i^{(\ell+1)} = \sum_{i=1}^{n_\ell} W_{ij}^{(\ell+1)} \sigma(z_j^{(\ell)}) + b_i^{(\ell+1)}$$

- Backward pass:

$$W_{ij}^{(\ell+1)} = W_{ij}^{(\ell)} - \gamma \frac{\partial L}{\partial W_{ij}} \Big|_{W_{ij}^{(\ell)}}$$

$$b_i^{(\ell+1)} = b_i^{(\ell)} - \gamma \frac{\partial L}{\partial b_i} \Big|_{b_i^{(\ell)}}$$

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

$$z_i^{(\ell+1)} = \sum_{i=1}^{n_\ell} W_{ij}^{(\ell+1)} \sigma(z_j^{(\ell)}) + b_i^{(\ell+1)}$$

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

$$W_{ij}^{(\ell+1)} = W_{ij}^{(\ell)} - \gamma \frac{\partial L}{\partial W_{ij}}\bigg|_{W_{ij}^{(\ell)}} \qquad b_i^{(\ell+1)} = b_i^{(\ell)} - \gamma \frac{\partial L}{\partial b_i}\bigg|_{b_i^{(\ell)}}$$

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

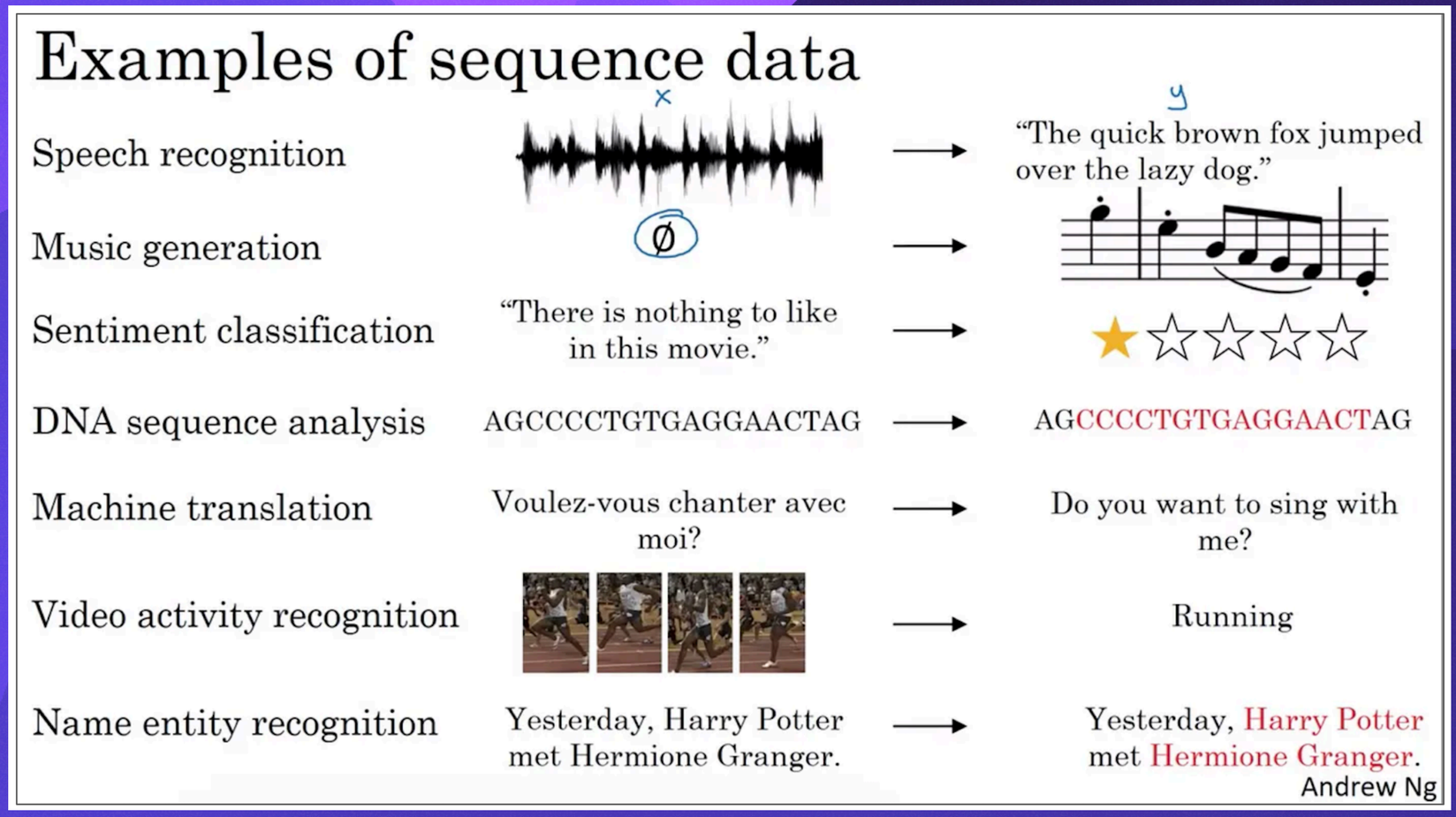
Beyond Simple DNNs

Survey of Deep Learning Architectures

- Deep NN (DNN) $\leftrightarrow$ Feed-Forward NN (FFNN) $\leftrightarrow$ Fully-Connected NN (FCNN)
- Many other architectures exist:
  - Recurrent NNs (RNNs): process sequential data
  - Convolutional NNs (CNNs): process data on a grid
  - Graph Neural Networks (GNNs): process data on a graph / attention
  - Generative Models: produce new data
  - … and more!

Recurrent Neural Networks

Designed to Process Sequential Data



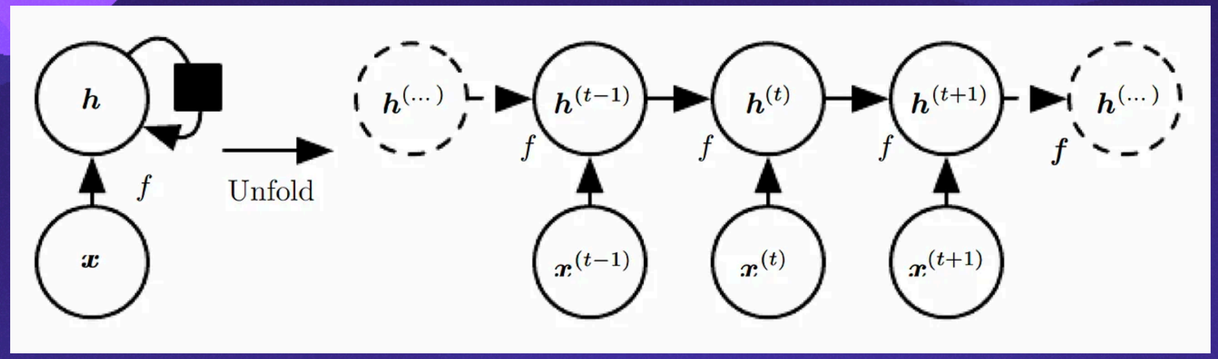
https://aiml.com/what-does-sequential-data-mean-which-models-are-best-suited-for-handling-sequential-data/

Recurrent Neural Networks

Basic Idea

- Time-indexed inputs: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}$
- Parameter sharing: apply the same set of learnable weights to all values of the time index
- Given a set of learnable parameters θ, the hidden units in many RNNs are calculated via. $\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta)$

Image from *Deep Learning* by Goodfellow et al.



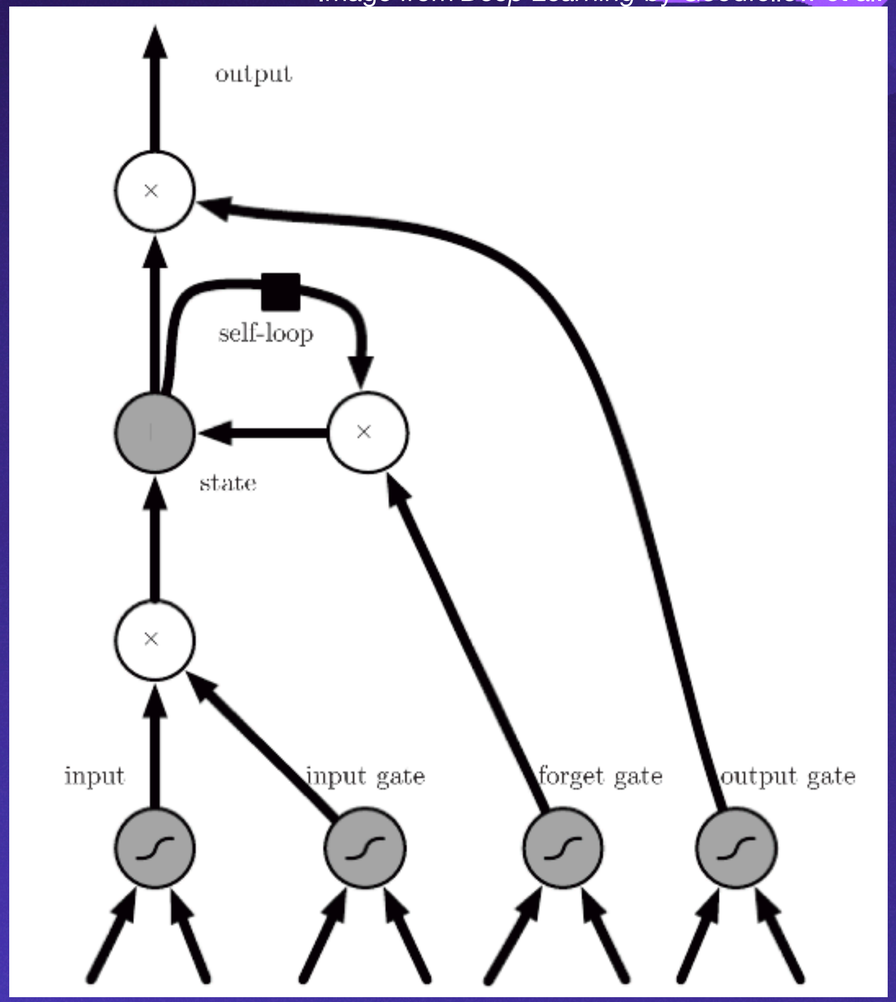
RNN with no outputs; information is processed sequentially, taking into account both $\mathbf{x}^{(t)}$ and $h^{(t-1)}$ but applying the *same function* $f(\cdot\,; \theta)$ *at each tilmestep*

Long Short-Term Memory (LSTM)

An Upgraded RNN Module

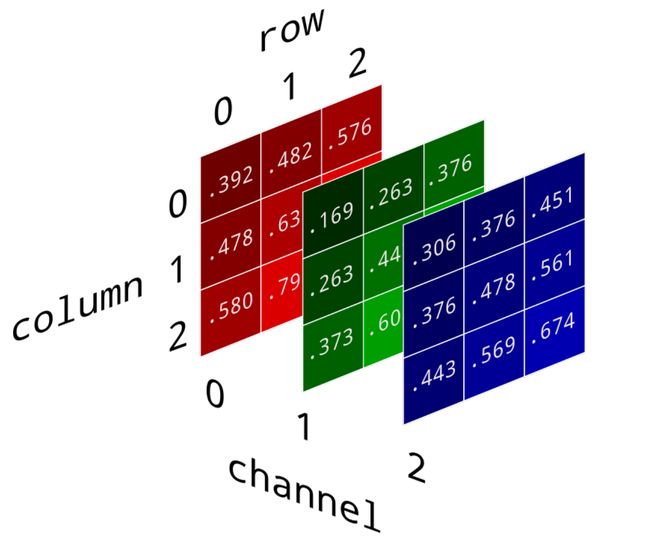
- RNNs are finicky to train; they often suffer from exploding/vanishing gradients
- This has motivated the development of more advanced RNNs like LSTMs
- The LSTM is a recurrent “cell” that is applied to all timesteps equally

Image from *Deep Learning* by Goodfellow et al.



Convolutional Neural Networks (CNNs)

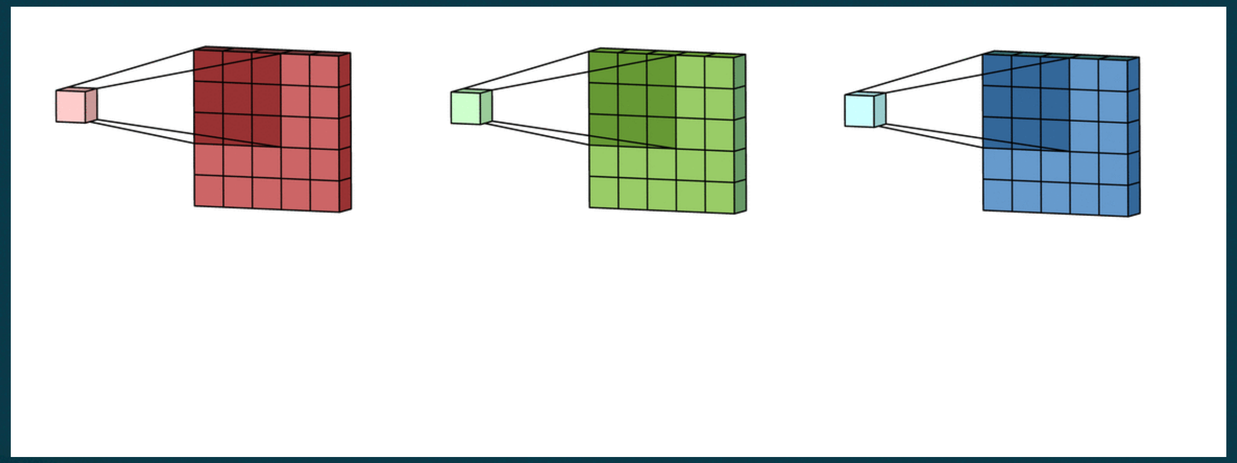
- Deep learning applied to images (data on a grid)
  - For square images, inputs are $I \in \mathbb{R}^{n_{\text{pixels}} \times n_{\text{pixels}} \times n_{\text{features}}}$

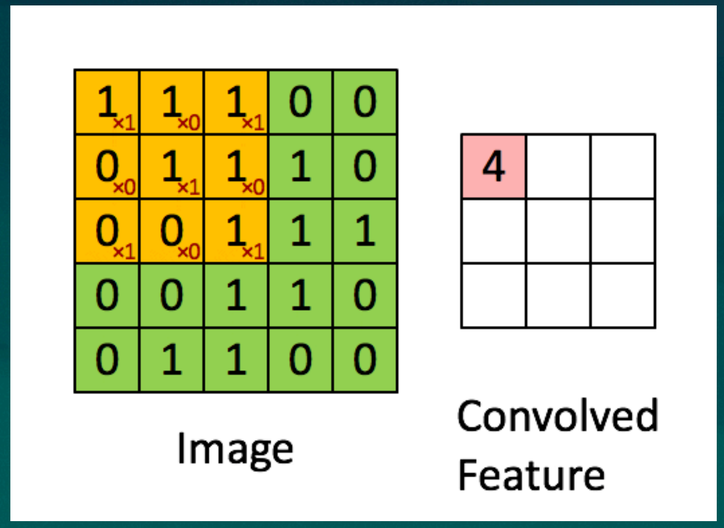


https://ai.plainenglish.io/complete-guide-image-classification-on-cifar-10-chapter-i-multiclass-logistic-regression-525daee0775b

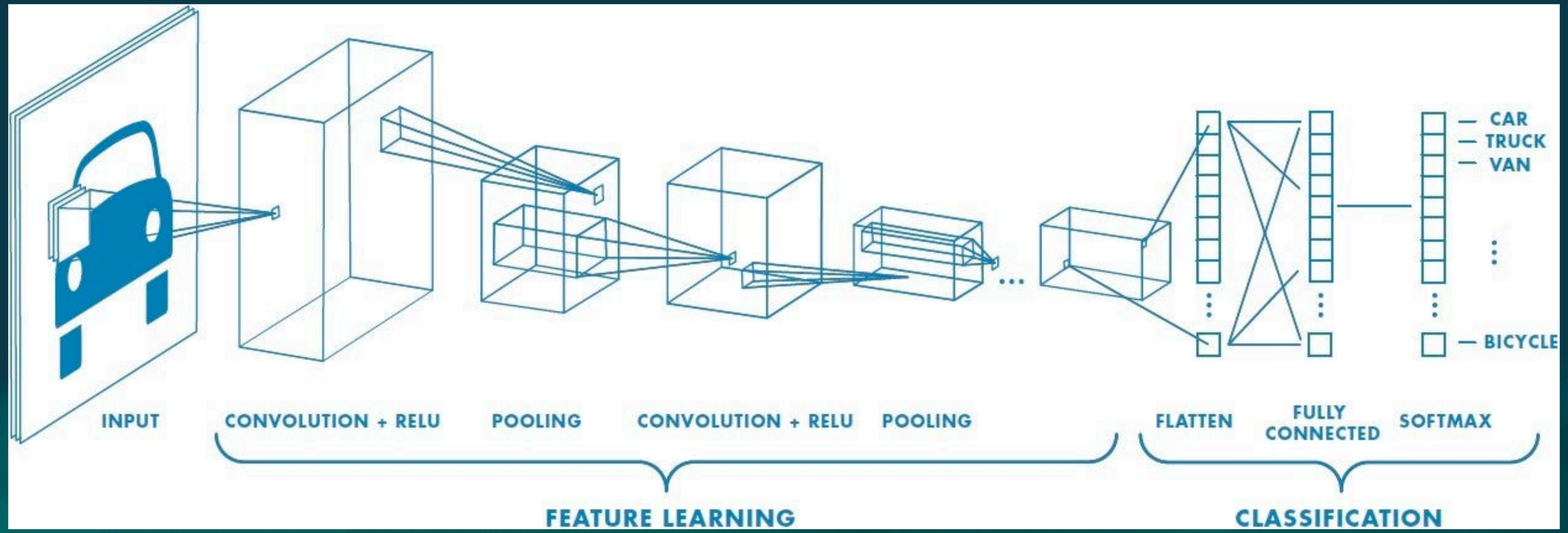
Convolutional Neural Networks (CNNs)

- Very similar approach to DNNs (non-sequential inputs, feed-forward approach), except now we use *convolutional* layers
  - Convolution: filter is *convolved* (weighted sum with learnable weights) with the input image
  - Again, parameter sharing!





Convolutional Neural Networks (CNNs)

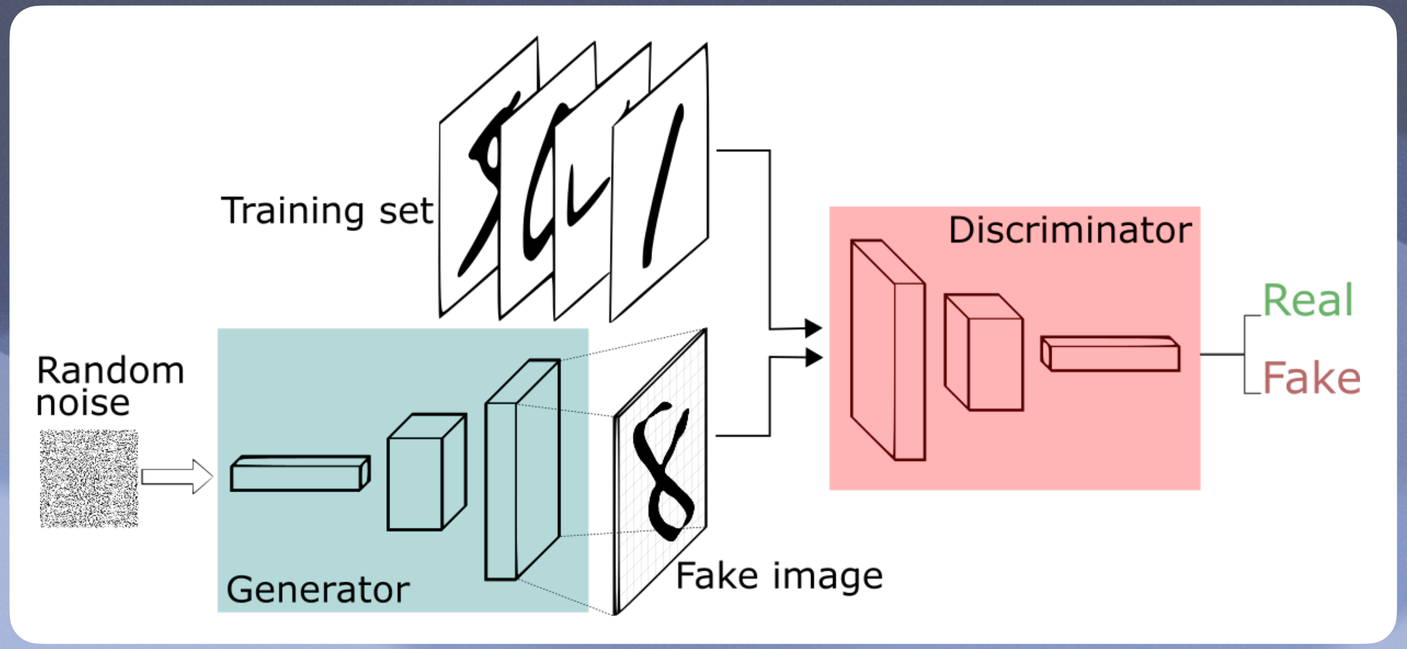


https://miro.medium.com/v2/resize:fit:1358/1*vkQ0hXDaQv57sALXAJquxA.jpeg

Generative Adversarial Networks (GANs)

Survey of Deep Learning Architectures

- “Generative” AI: use ML to create new images, sounds, etc.
- GANs: two agents (the *generator* and the *discriminator*) are given competing tasks:

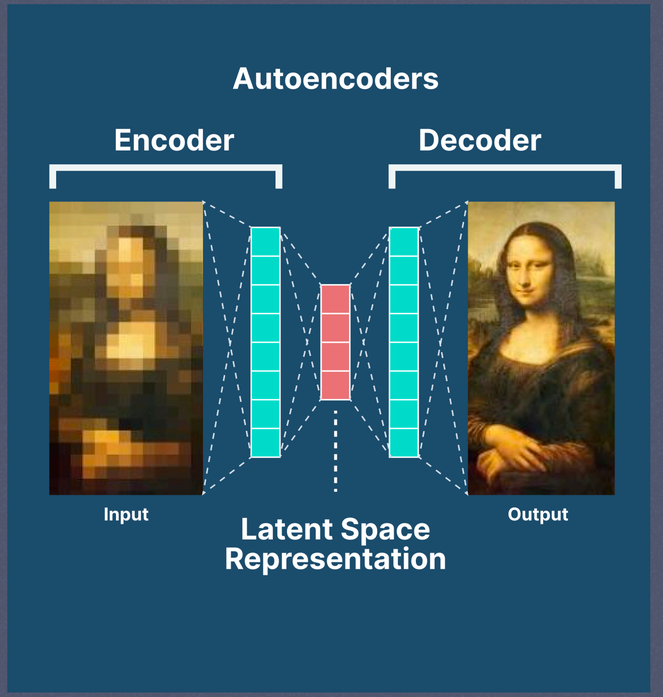


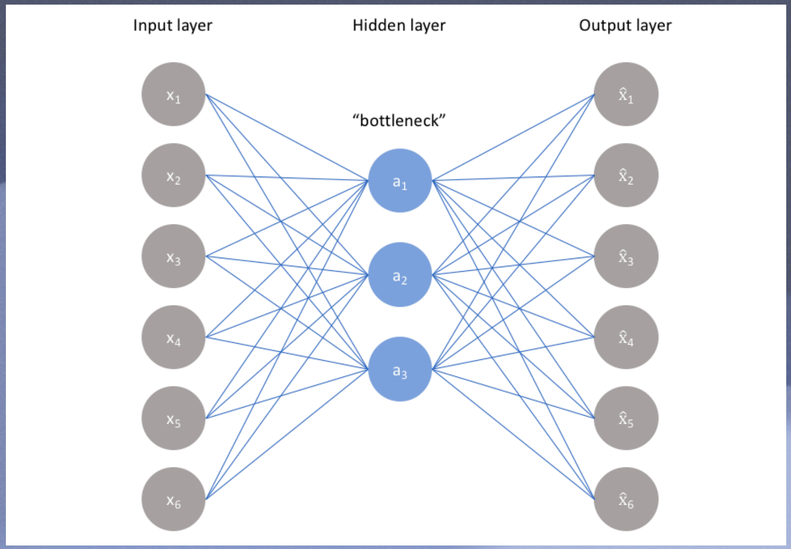
https://sthalles.github.io/intro-to-gans/

Autoencoders

Learning Efficient Codings

- Autoencoders are used to produce compressed data representations
  - **Encoder**: produces a lower-dimensional (compressed) “latent” representation of the input data
  - **Decoder**: given the compressed representation, reconstruct the original data
- Decoded representations typically less noisy,
- Uses: efficient encoding, image denoting, generative modeling, anomaly detection



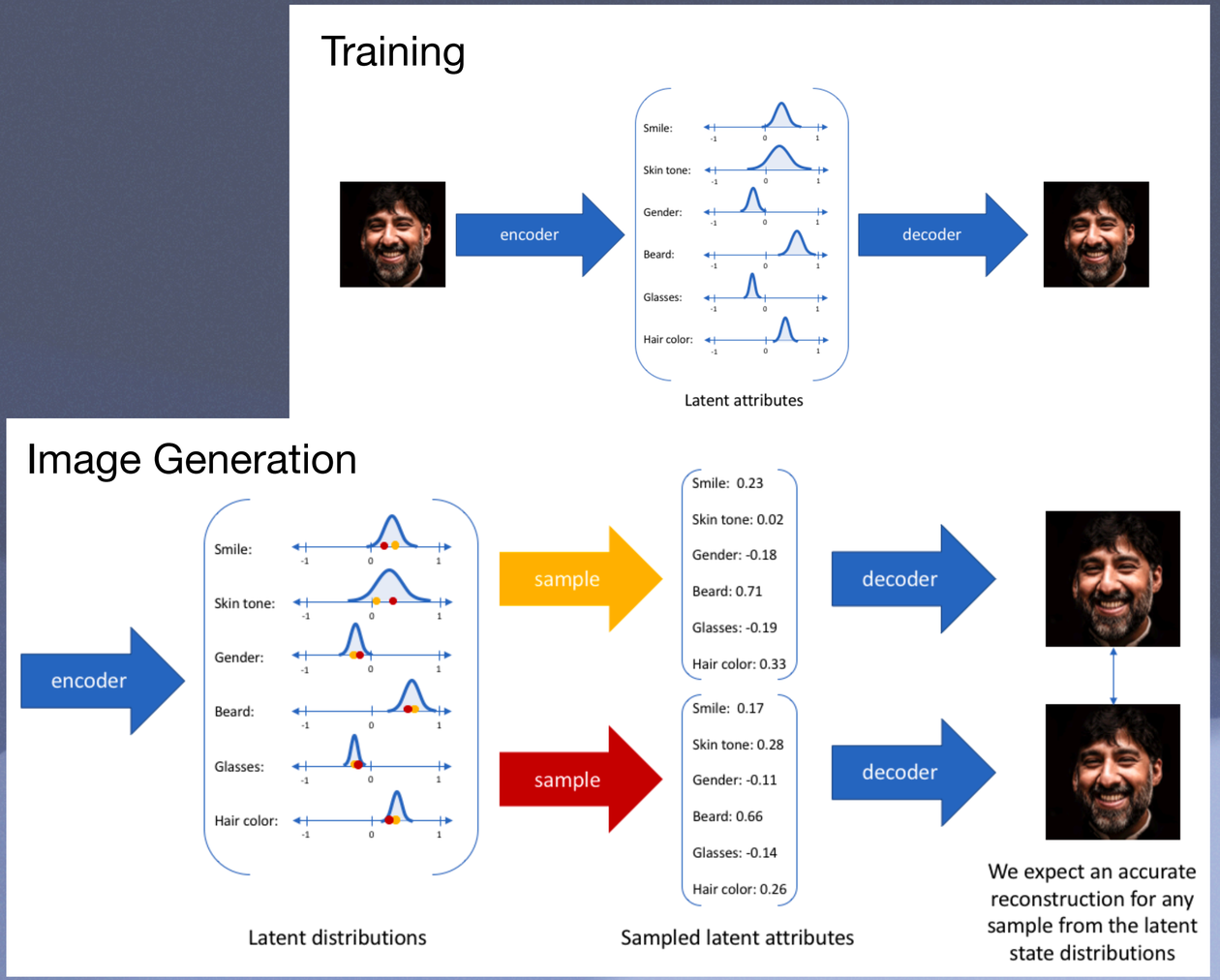


https://www.v7labs.com/blog/autoencoders-guide#:~:text=An%20autoencoder%20is%20an%20unsupervised,even%20generation%20of%20image%20data.

Variational Autoencoder (VAEs)

Generative Modeling via Autoencoders

- Generate realistic images from random noise
  - **Encoder**: predict means and standard deviations of a *probability distribution* over the latent features
  - **Decoder:** given a random sample from the latent distributions, produce the corresponding output



https://www.jeremyjordan.me/variational-autoencoders/


SCAN ME