# APS360: Applied Fundamentals of Deep Learning

Week 2: Artificial Neural Networks - Part III

# Content

- Hyperparameters
- Optimizers
- Learning Rate
- Normalization
- Regularization
- Pytorch Implementation
- Evaluation and Debugging

# Hyperparameters

# Hyperparameters

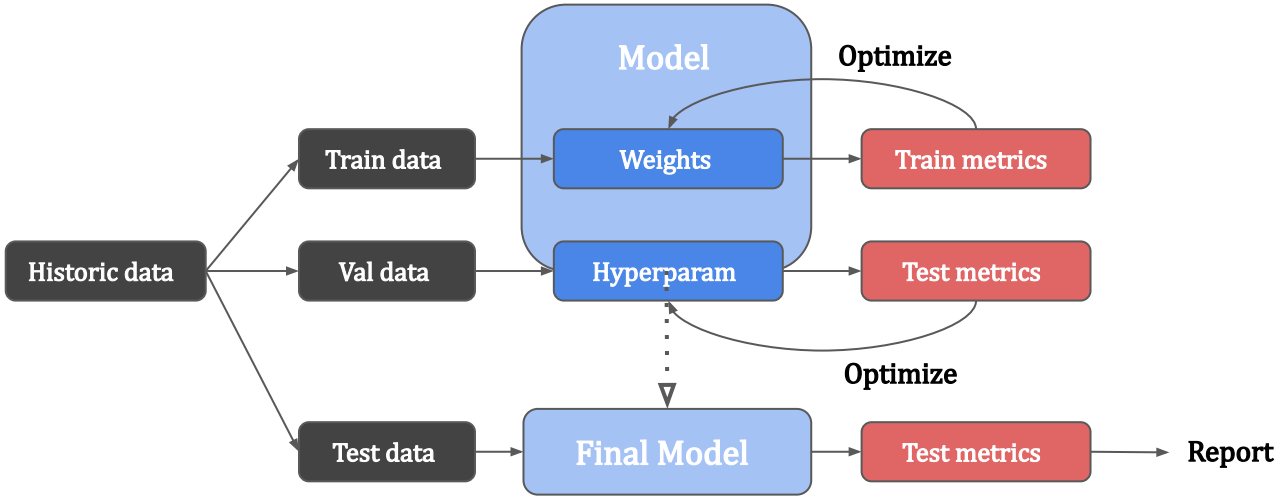
Neural network settings that are not optimized during learning:

- Batch size,
- Number of layers,
- Layer size,
- Type of activation function,
- Learning rate
- ...

Weights are updated through gradient descent (Inner loop of optimization).

How do we tune hyperparameters? (Outer loop of optimization)

# Hyperparameters

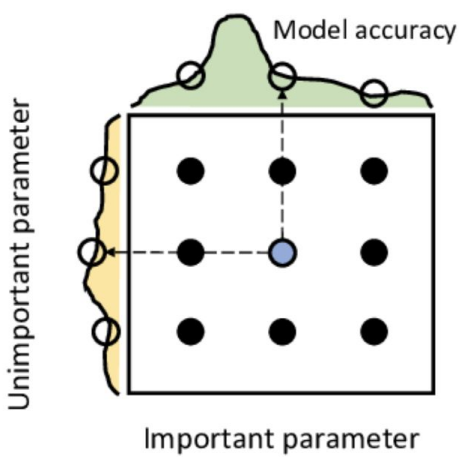


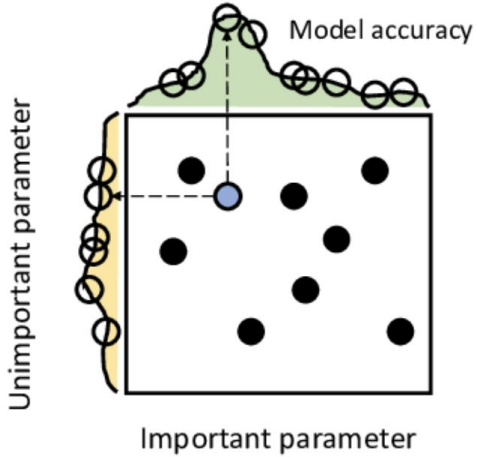

Stratified Sampling
without Replacement

# Hyperparameters

How do we tune hyperparameters?

**Grid Search**




**Random Search**

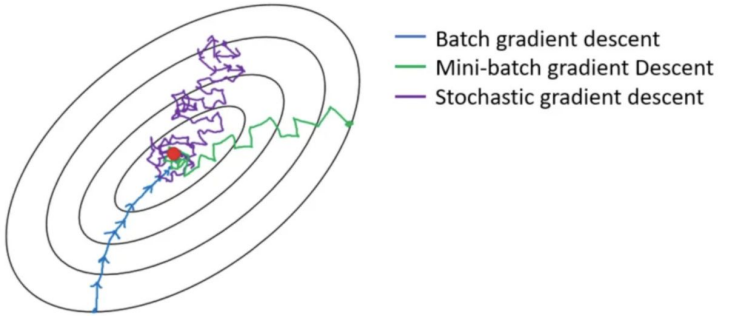
# Optimizers

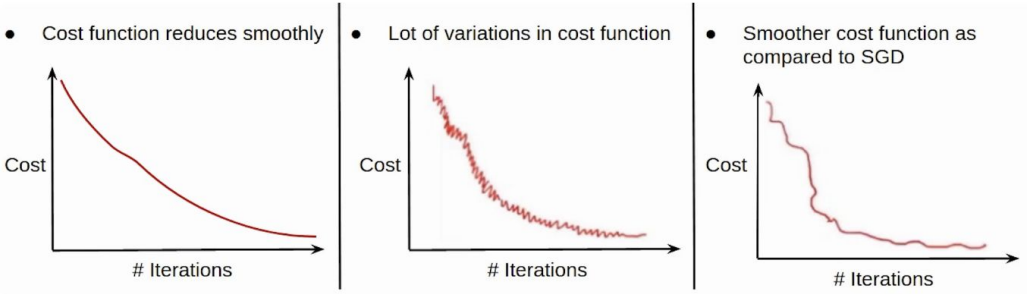
# Stochastic Gradient Descent (SGD)

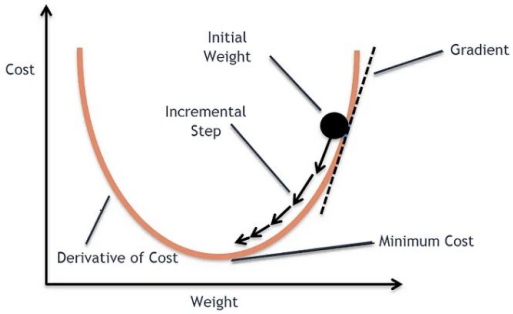
For each iteration evaluate a training sample from the dataset **taken at random** .

**Computing the gradient takes less time** , but… may not actually be faster…

SGD allows you to do more of a global search for an optimum, often results in a better set of weights for your model



# Mini-Batch Gradient Descent

Instead of working with one sample at a time… can apply **batching…**

1. Use our network to make the predictions for **n samples**
2. Compute the average loss for those **n samples**
3. Take a "step" to optimize the average loss of those **n samples**

**Batch size** : # Training examples used per optimization "step".

**Iteration:** One step: The parameters are updated once per iteration.

**Epoch** : # Times all the train data is used once to update the parameters.

- Suppose there are 1000 samples in train data, if we set batch size to 10 then 1 epoch will contain 100 iterations

# Ineffective Batch Size

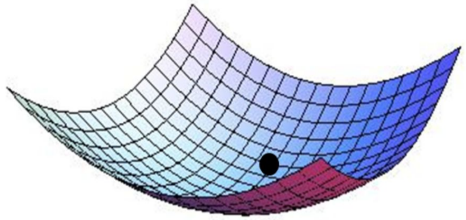
## Too small

- We optimize a (possibly very) different function loss at each iteration
- Noisy

## Too large

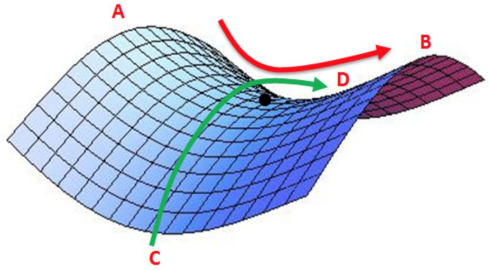
- Expensive
- Average loss might not change very much as batch size grows
- The true gradient is not always the best gradient for optimization. i.e. some amount of noise in your gradients can help training (converge faster): i.e. larger batch size is not always better.

# Gradient Descent: N-Dimensional



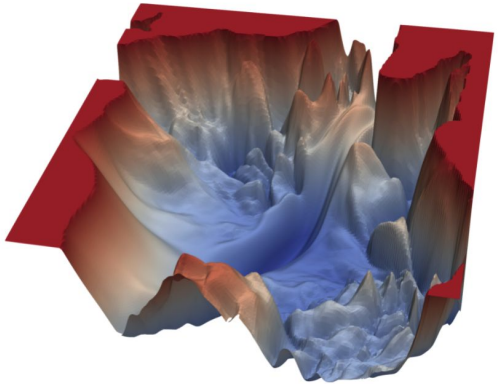
A deep neural network has **millions or billions** of parameters

Real gradient descent of a deep network is optimization in millions of dimensions!




Most points of zero gradients are saddle points.

Plateaus are a problem but can be addressed using specialized variants on gradient descent

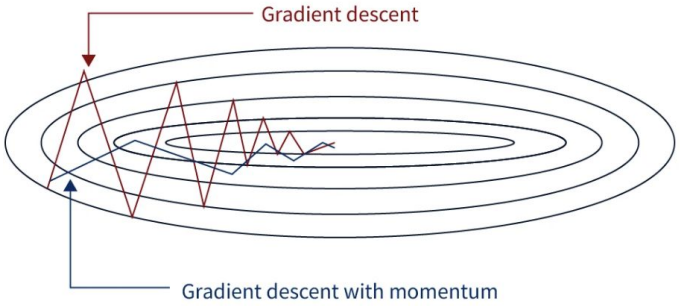


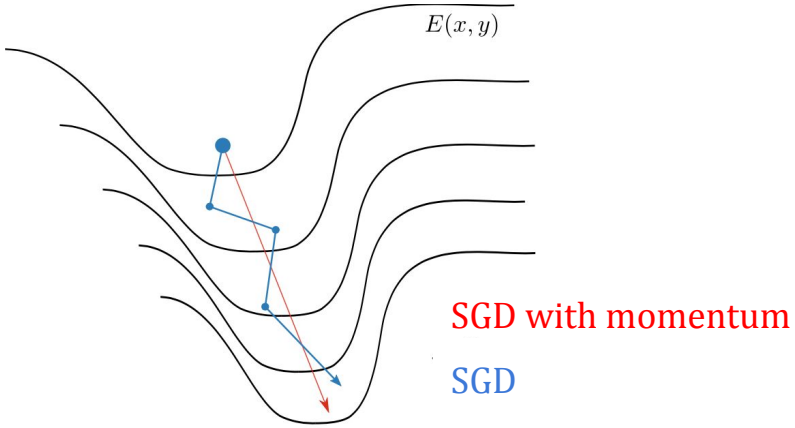
# SGD with Momentum

**Ravines** : areas where the surface curves much more steeply in one dimension than in another, common around local optima.

SGD has trouble navigating ravines → it oscillates across the slopes of the ravine

Momentum helps accelerate SGD in the relevant direction and dampens oscillations



# SGD with Momentum

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions

Analogy → we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way until it reaches its terminal velocity

$$\begin{cases} v_{ji}^{t} = \lambda v_{ji}^{t-1} - \gamma \dfrac{\partial E}{\partial w_{ji}} \\ w_{ji}^{t+1} = w_{ji}^{t} + v_{ji}^{t} \end{cases}$$

```
torch.optim.SGD(model.parameters(),lr=0.001,
momentum=0.01)
```

# Adaptive Moment Estimation (Adam)

Adaptive learning rates → **each weight has its own rate**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left(\frac{\partial E}{\partial w_{ji}}\right)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial E}{\partial w_{ji}}\right)^2$$

$$w_{ji}^{t+1} = w_{ji}^{t} - \frac{\gamma}{\sqrt{v_t}+\epsilon} m_t$$

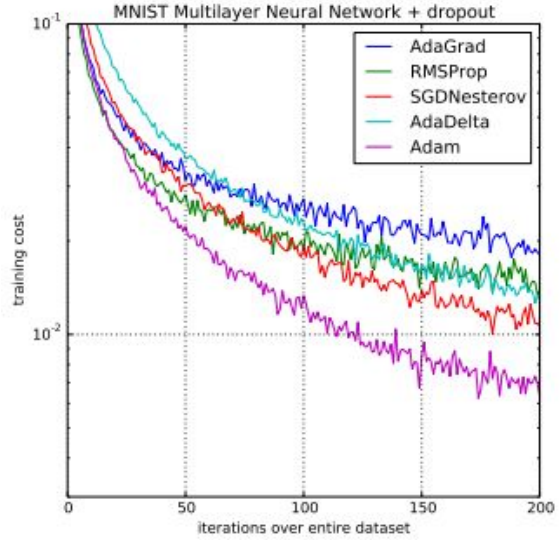
# Adaptive Moment Estimation (Adam)

Adaptive learning rates → **each weight has its own rate**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left(\frac{\partial E}{\partial w_{ji}}\right)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial E}{\partial w_{ji}}\right)^2$$

$$w_{ji}^{t+1} = w_{ji}^{t} - \frac{\gamma}{\sqrt{v_t}+\epsilon} m_t$$




Incorporates momentum and adaptive learning rate:

- rapid convergence
- requires minimal tuning
- commonly used optimizer

```
torch.optim.Adam(model.parameters(),lr=0.001)
```

# Learning Rate

# Learning Rate

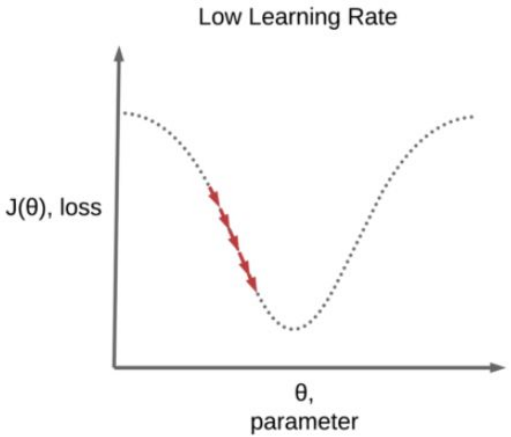
The learning rate determines the **size of the step** that an optimizer takes during each iteration.

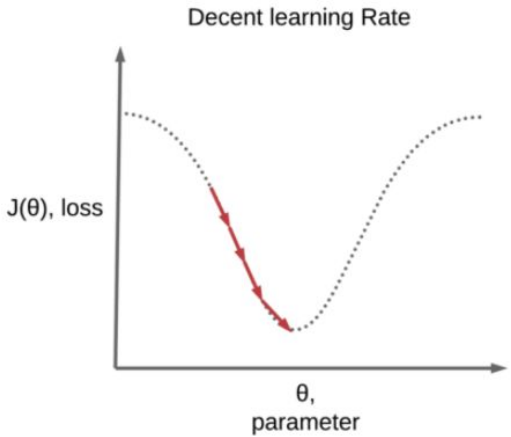
$$w_{ji}^{t+1} = w_{ji}^{t} - \gamma \frac{\partial E}{\partial w_{ji}}$$

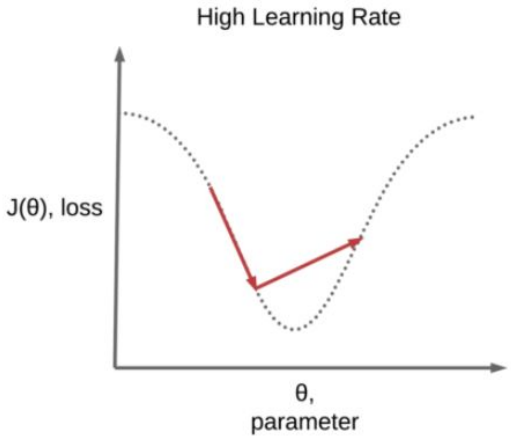
**Larger step size** = make a bigger change in the parameters (weights) in each iteration.

Q: What happens if the learning rate is too small? Too large?

# Learning Rate Size



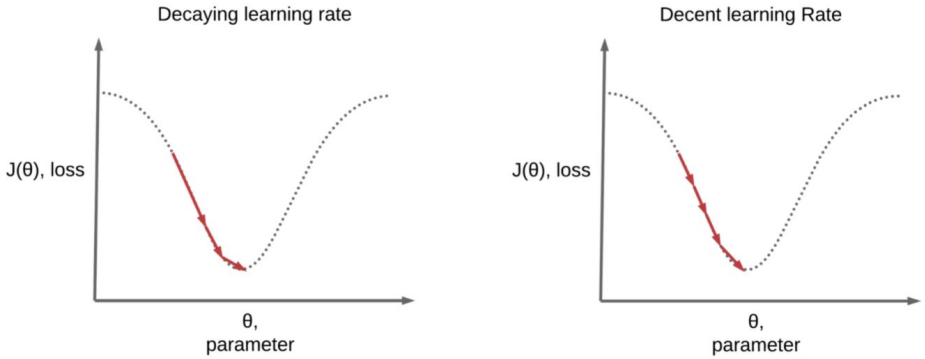

- Very small parameter change
- Longer training time

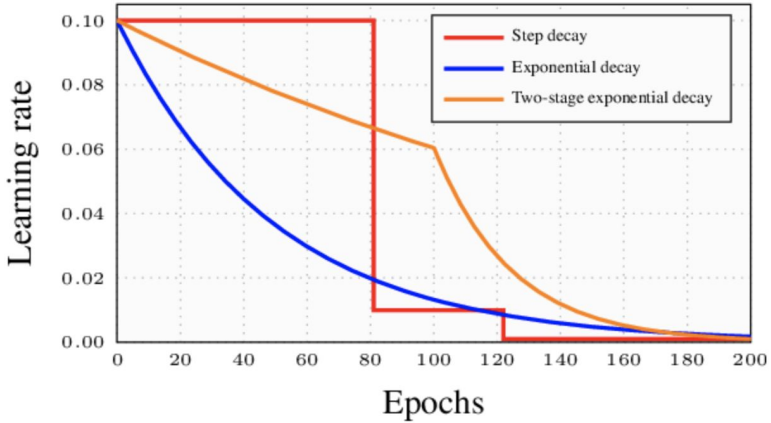
- Noisy
- detrimental to training

# Appropriate Learning Rate

Depends on:

- The learning problem
- The optimizer
- The batch size
  - Large batch → larger learning rates.
  - Small batch → smaller learning rate.
- The stage of training
  - Reduce as training progresses



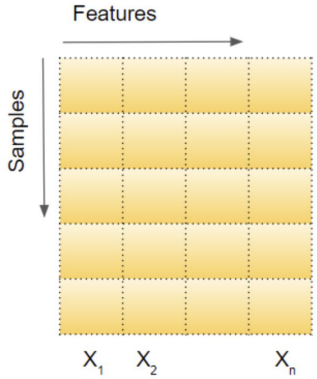



# Normalization

# Why Normalization?

We always normalize the inputs to prevent the model from paying attention to the features with larger range.

$$X_i = \frac{X_i - \mu_i}{\sigma_i}$$




year
$sq^2$
#rooms
finished

$$\begin{pmatrix} 2001 \\ 3000 \\ 5 \\ 1 \end{pmatrix} \Rightarrow X = \begin{pmatrix} 2001 \\ 3000 \\ 5 \\ 1 \end{pmatrix}$$

price

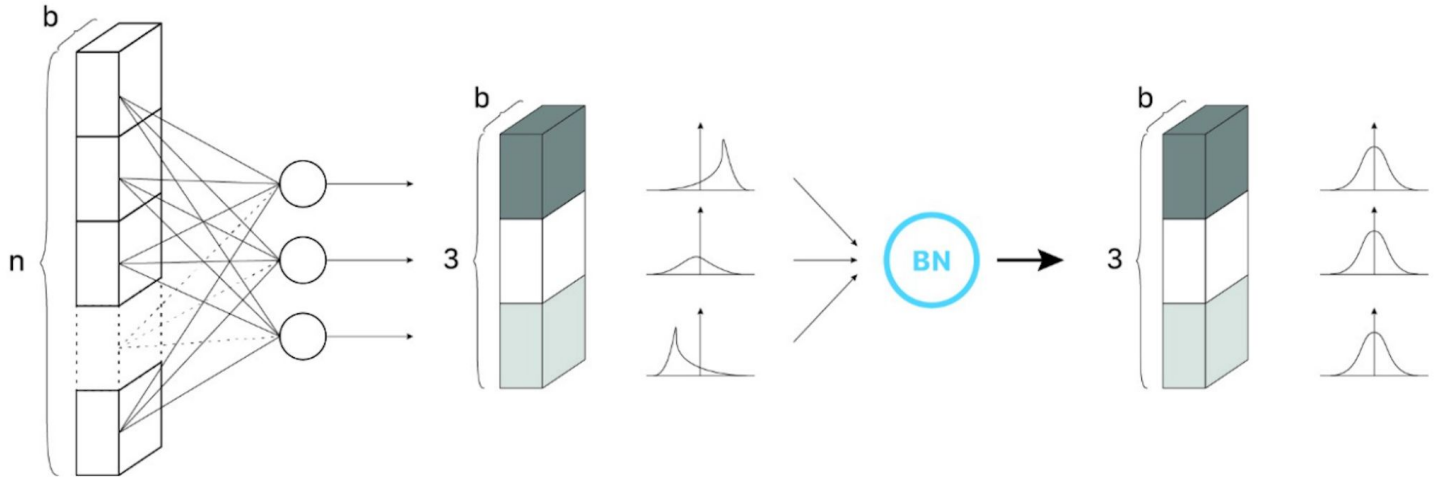
$$\begin{pmatrix} 1.5M \end{pmatrix} \Rightarrow y_t = \begin{pmatrix} 1.5M \end{pmatrix}$$

This only normalizes the data for the first layer.

How to normalize activations of each layer for the next layer?

# Why Normalization?

How to normalize activations of each layer for the next layer?



# Batch Normalization

Normalize activations batch-wise for each layer

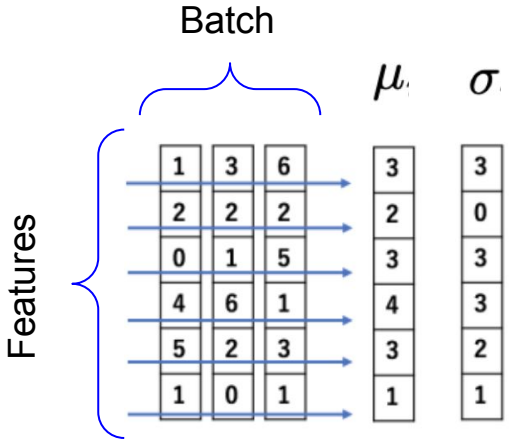
**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

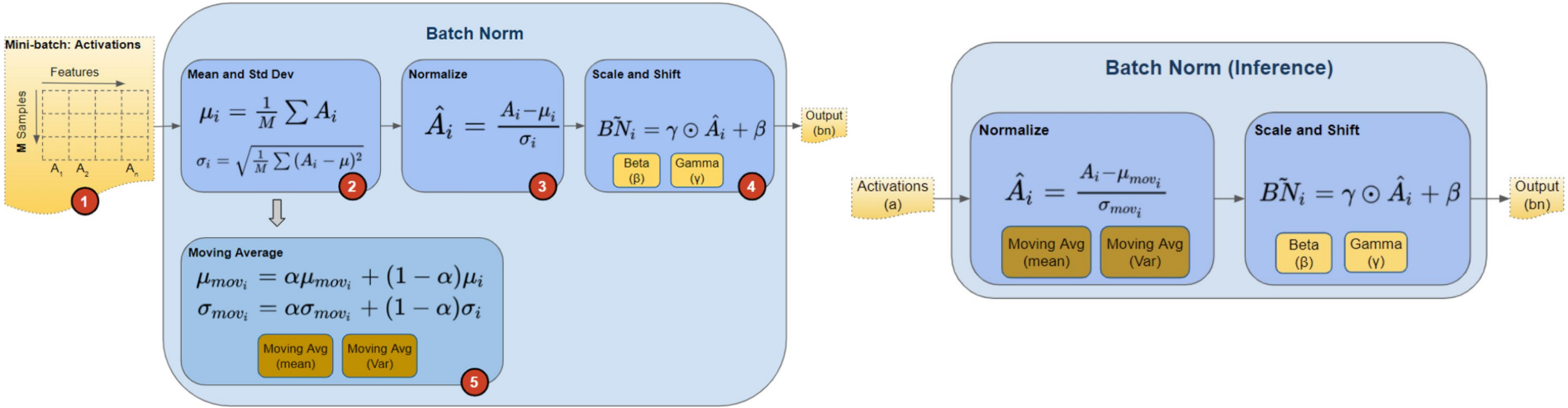
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



# Batch Normalization: Inference Time

Keep a moving average during training and use it at inference time



# Batch Normalization

Pros:

- Higher learning rate → speeding up the training
- Regularizes the model
- Less sensitivity to initialization

Cons:

- Depends on batch size → No effect with small batches
- Cannot work with SGD

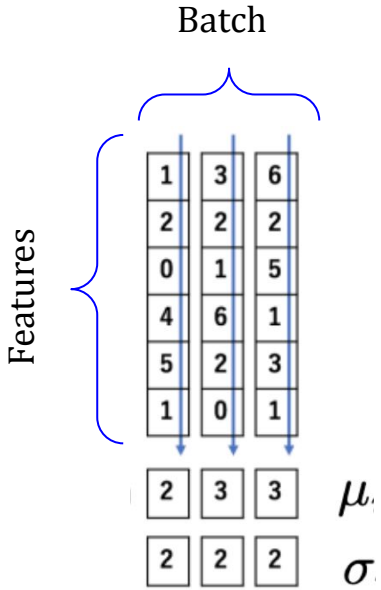
```
linear =
torch.nn.Linear(64,128)
bn = torch.nn.BatchNorm1D(128)
...
Output = bn(linear(input))
```

# Layer Normalization

Normalization is applied on the neuron for a single instance across all features

- Simpler to implement, no moving averages or parameters
- Not dependent on batch size

```
linear =
torch.nn.Linear(64,128)
ln = torch.nn.LayerNorm(128)
...
Output = ln(linear(input))
```



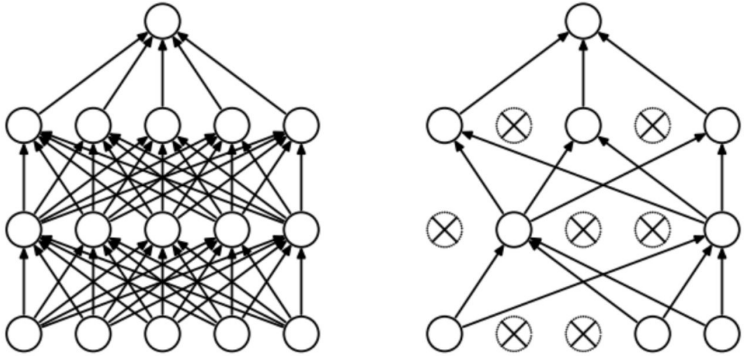
# Regularization

# Dropout

Forces a neural network to learn more robust features

- **During training** → Drop activations (set to 0) with probability p
- **During inference** → multiply weights by (1-p) to keep the same distribution

```
linear =
torch.nn.Linear(64,128)
drop = torch.nn.Dropout(p=0.3)
...
Output = drop(linear(input))
```



(a) Standard Neural Net (b) After applying dropout.

# Weight Decay (L2)

Prevents the weights from growing too much → Lowering variance

$$E(W;y,t) = E(W;y,t) + \frac{\alpha}{2} \parallel W \parallel_2^2 \longrightarrow \frac{\partial E}{\partial W} = \frac{\partial E}{\partial W} + \alpha W$$

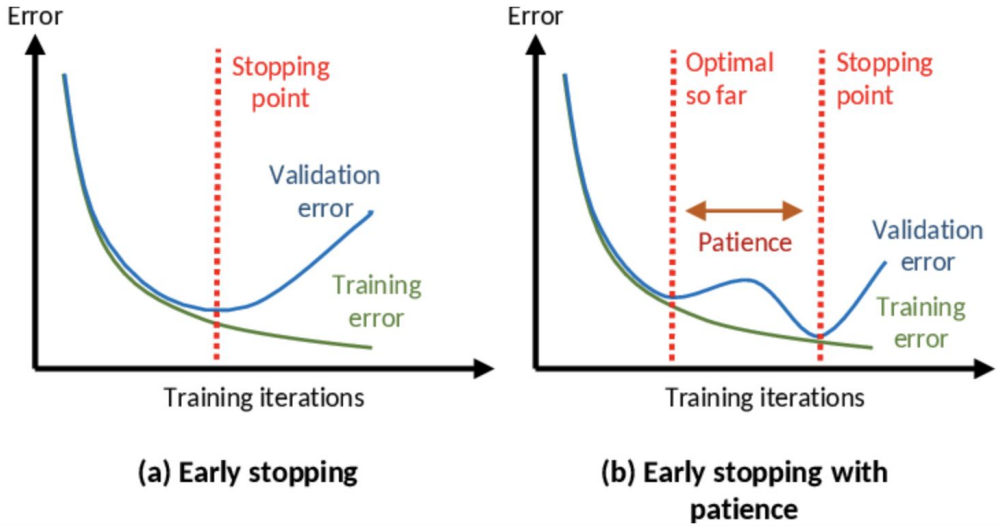
$$W_{t+1} = W_t - \gamma \left( \alpha W_t + \frac{\partial E}{\partial W} \right)$$

Weight reduction is multiplicative and proportion to the scale of W

```
torch.optim.Adam(model.parameters(),lr=0.001,
weight_decay=1e-8)
```

# Early Stopping with Patience

- In each training iteration observe the validation loss
- As soon as validation loss starts to increase, start a counter
- If the validation loss decreases, reset the counter
- Otherwise, wait for a fixed iterations (patience) and then stop the training

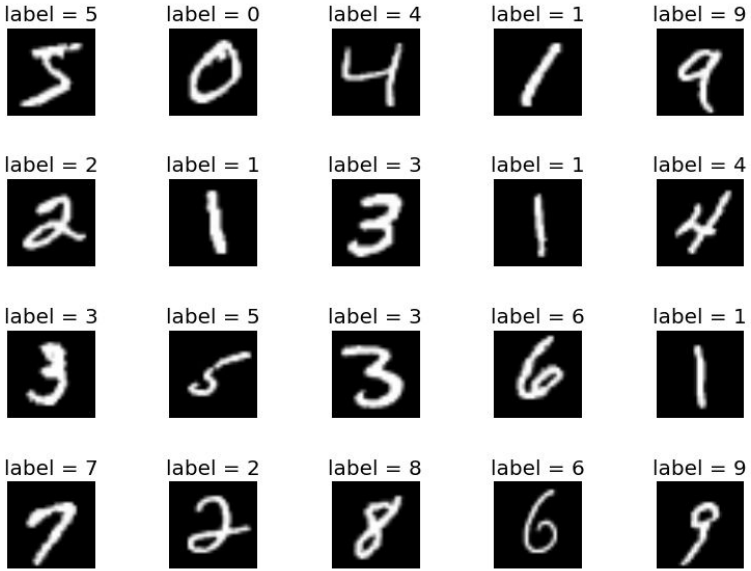


# Intermission
# (5 to 10 min break)

# PyTorch Implementation

# MNIST Dataset

- Input: 28x28 pixel image
- Output: Whether the digit is small (0, 1, 2)
  - output=1 means that the digit is small
  - output=0 means that the digit is not small
- Q: Is this a supervised or unsupervised lear problem?
- Q: Is this a regression or classification probl

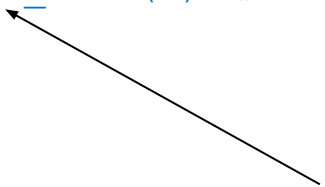


# PyTorch: ANN Setup

Import all the necessary modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt  # for
plotting
import torch.optim as optim

torch.manual_seed(1)  # set the random seed
```



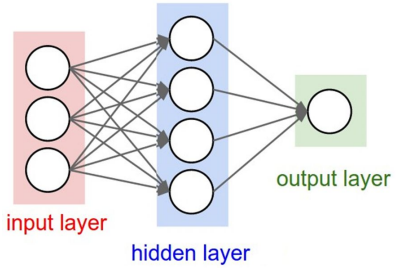
Important for **reproducing** results

# PyTorch: ANN Architecture

```
# define a 2-layer neural network
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 30)
        self.drop = nn.Dropout(p=0.3)
        self.layer2 = nn.Linear(30, 1)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.drop(self.layer1(flattened))
        activation1 = F.relu(activation1)
        activation2 =
self.drop(self.layer2(activation1))
        return activation2

model = NN()
```




class nn.Linear defines a fully-connected layer

forward() method defines how to make a prediction

where is the final sigmoid activation?

```
activation2 =
F.sigmoid(activation2)
```

# Loss Function and Final Activation

- PyTorch implementation can be confusing
- You would expect to see the sigmoid activation applied to the output layer. Indeed this would be the case if we used:

```
criterion = nn.BCELoss()
```

- Due to numerical stability, we will use:

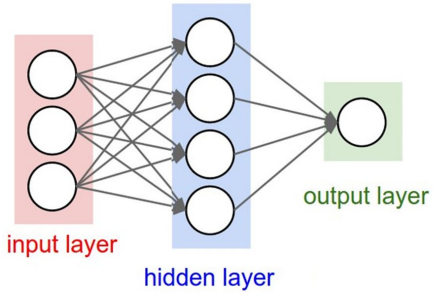
```
criterion = nn.BCEWithLogitsLoss()
```

- Applies sigmoid activation internally!

# Forward and Backward Pass

- Forward pass: **makes a prediction**
  - e.g. `model(input)`, which calls network.forward method
  - Information flows forwards from input to output layer

- Backward pass: **computes gradients** for making changes to weights
  - e.g. `loss.backward()`
  - Information flows backwards from output to input layer



# Pytorch: Forward and Backward Pass

Training code for binary classification problem

```
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005, momentum=0.9)

for (image, label) in mnist_train:
    # ground truth: is the digit less than 3?
    actual = torch.tensor(label < 3).reshape([1,1]).type(torch.FloatTensor)

    out = model(img_to_tensor(image))   # make prediction
    loss = criterion(out, actual)       # calculate loss
    loss.backward()                     # obtain gradients
    optimizer.step()                    # updates parameters
    optimizer.zero_grad()               # a clean up step - important!
```

# PyTorch: Training and Validation Error

Assessing **model performance** by tracking error rate and accuracy

```
# computing the error and accuracy on the training set
error = 0
for (image, label) in mnist_train:
    prob = torch.sigmoid(model(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3):
        error += 1
print("Training Error Rate:", error/len( mnist_train))
print("Training Accuracy:", 1 - error/len( mnist_train))
```

We set a threshold at prob = 0.5

Replace mnist_train with mnist_val to obtain error and accuracy on test (validation) set

# Multi-Class Classification

- What about when we have more than 2 choices (classes) to pick from?
- Identify each image to its corresponding digit 0-9

## One-hot encoding

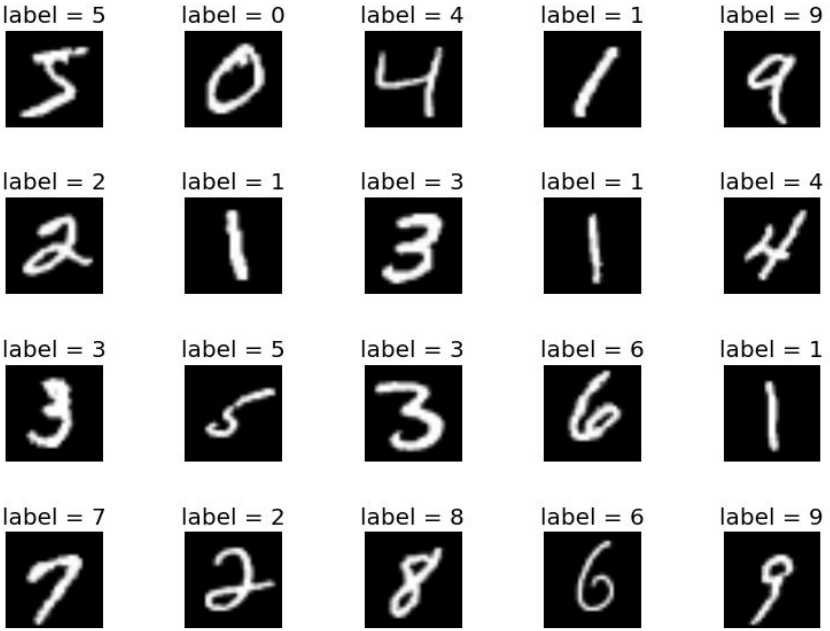
$0 \rightarrow [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

$1 \rightarrow [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$

$2 \rightarrow [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$

$\vdots$

$9 \rightarrow [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]$



# Multi-Class Classification

Requires minor changes to our PyTorch implementation:

1. The final output layer has **as many neurons as classes** .
2. Apply the **softmax activation function** on the final layer to obtain class probabilities
3. Use the **multiclass cross-entropy** loss function

# PyTorch: Multi-Class ANN Architecture

```
class MNISTClassifier(nn.Module):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 50)
        self.bn = nn.BatchNorm1D(50)
        self.layer2 = nn.Linear(50, 20)
        self.ln = nn.LayerNorm(20)
        self.layer3 = nn.Linear(20,  10)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = bn(F.relu(self.layer1(flattened)))
        activation2 = ln(F.relu(self.layer2(activation1)))
        output = self.layer3(activation2)
        return output

model = MNISTClassifier()
```

one output neuron for each of the 10 digits

where is the softmax activation?

```
activation2 = F.softmax(activation2,
dim=-1)
```

# Loss Function and Softmax Activation

- You would expect to see the softmax activation applied to the output layer. Indeed this would be the case if we used:

```
criterion = nn.NLLLoss()
```

- Due to numerical stability, we will use:

```
criterion = nn.CrossEntropyLoss()
```

- Applies softmax activation internally!

# PyTorch: Output Probabilities

To obtain output probabilities we have to apply the softmax:

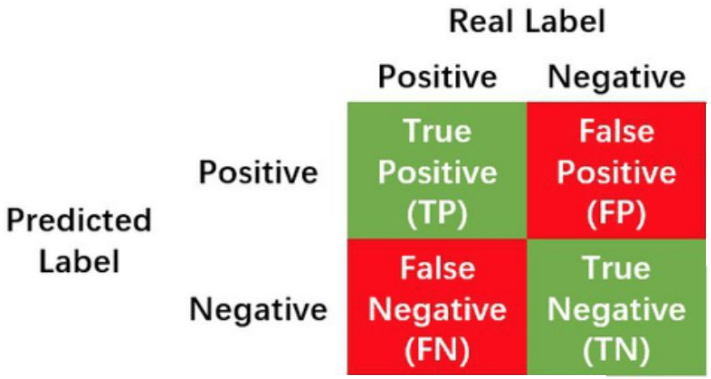
```
prob = F.softmax(output, dim=1)
print(prob)
print(sum(prob[0]))
```

# Evaluation and Debugging

# Debugging Neural Networks

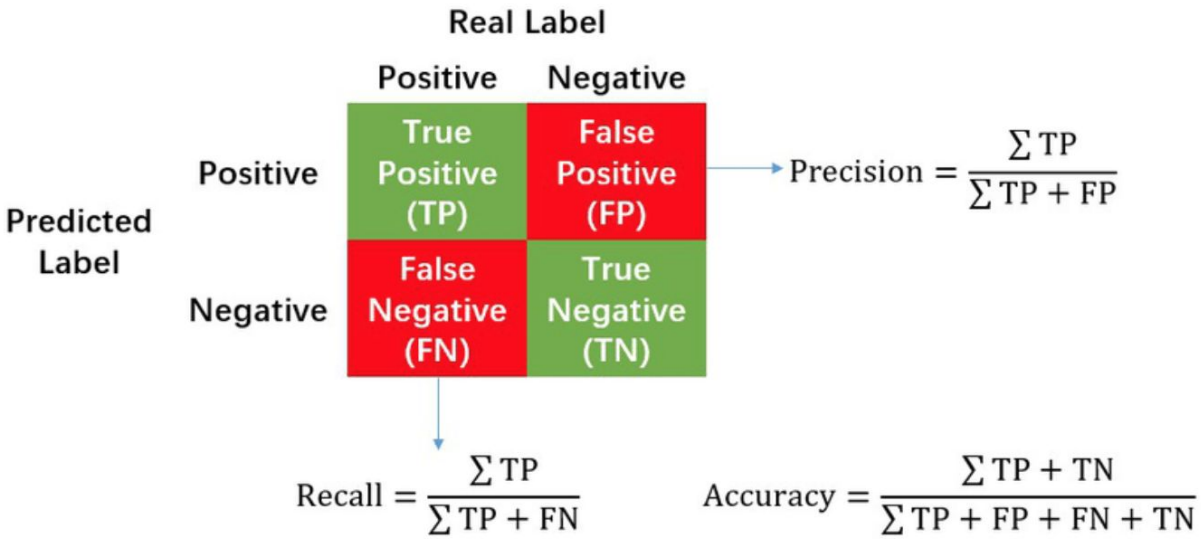
- Make sure your model can overfit
  - Make sure you can get loss to decrease w.r.t training data
- Make sure that your network is training: i.e. loss is going down.
  - Sanity check!
- Ensures that you are using the right variable names, and **rule out other programming bugs** that are difficult to discern from architecture issues.
- Confusion Matrix
  - True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN)
- 2D Projections of Data
  - PCA, t-SNE

# Confusion Matrix



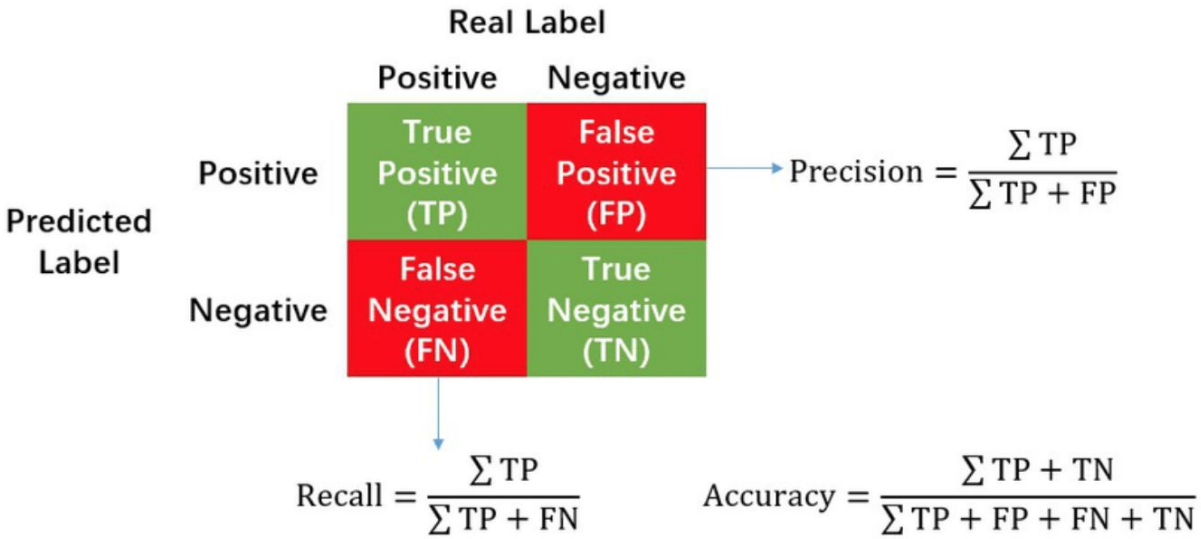

$$\text{Accuracy} = \frac{\sum \text{TP} + \text{TN}}{\sum \text{TP} + \text{FP} + \text{FN} + \text{TN}}$$

# Confusion Matrix

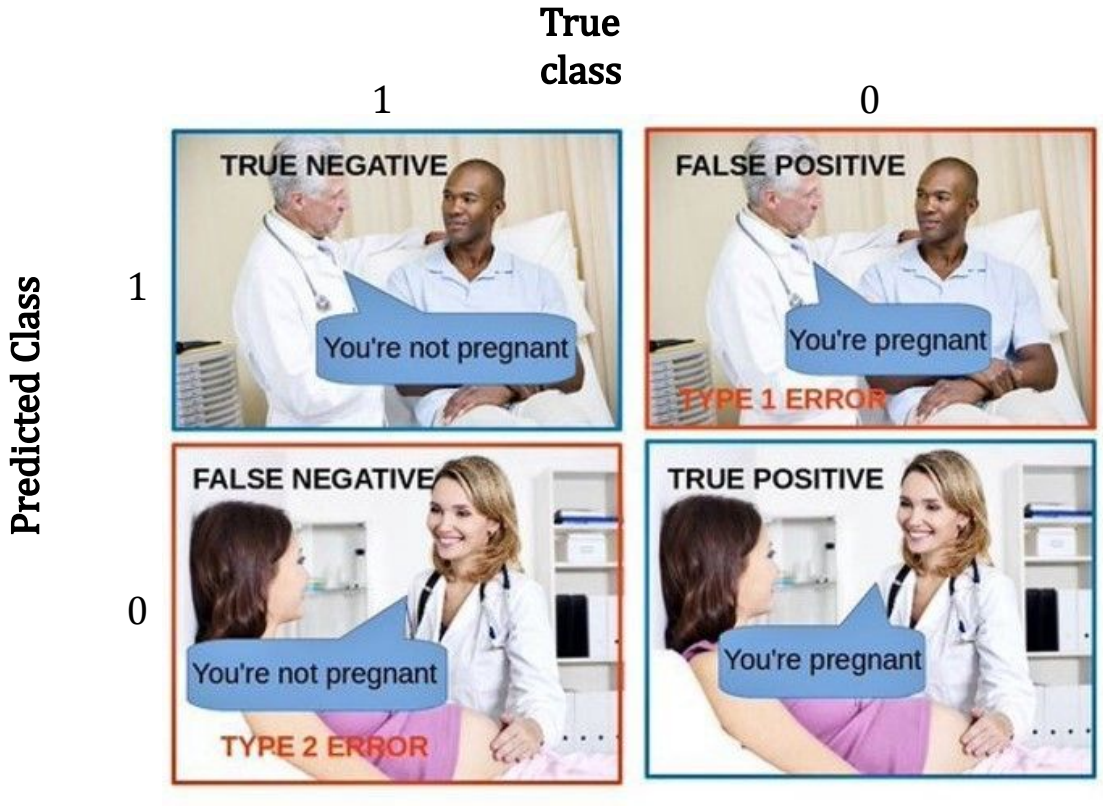


# Confusion Matrix

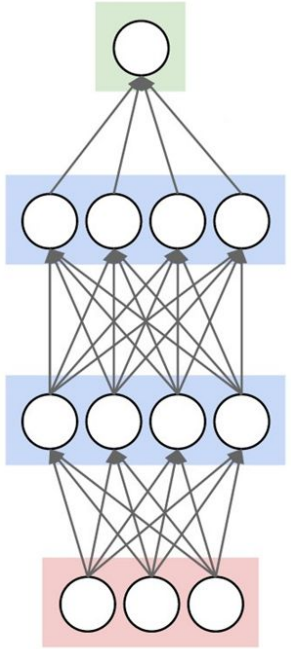
$$F1\ Score = 2 \times \frac{recall \times precision}{recall + precision}$$



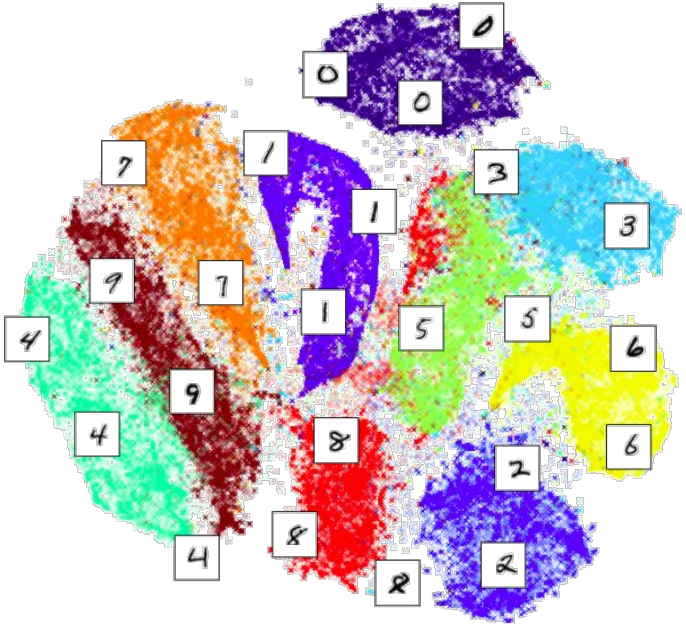
# Confusion Matrix Example



# MNIST 2D Visualization



t-SNE for 2D projection and visualization of data structure



https://projector.tensorflow.org/

# Questions?