Image Classification (Lecture 1)

- Goal: predict a label or distribution over labels for a given image (probabilty)
- Often width (pixel) x height (pixel) x 3 (red, green, blue) arrays,
- flattend to one large array where each value is an integer from 0 (black) to 255 (white)

Some challenges for computer vision algorithm (must be invariant to cross product of these while retaining sensitivity to inter-class variations):

- Viewpoint variation (orientation of image)
- Scale variation (variation of size in image and real life)
- Deformation (objects are not rigid bodies)
- Occlusion (object can be hidden)
- Illumination conditions (effects are drastic on pixel level)
- Background clutter (objects may blend into environment)
- Intra-class variation (many different types)

Nearest Neighbor Classifier ($k = 1$)

Takes test image (array) and compares it to every image (array) in training set and takes image label of closest one. *It simply remembers all training data*

- With k = 1 we create "small islands of liekly incorrect predictions, always use higher k

How compare to arrays A and B?

L1 distance/norm: Sum of over all absolute pixel differences $d_1(A, B) = \sum_i |A_i - B_i|$ If $A = B$ than L1 will be zero.

```
class NearestNeighbor():
  def __init__(self):
    pass
  def train(self, X, y):
    """ X is N x D where N is number of examples and D flattend pixel array.
    Y is 1-dimension of size N.
    Training is just remembering all data"""
    self.X_train = X
    self.y_train = y

  def predict(self, X):
    """ X is N x D where each row is an example
    we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type
    # matches the input type
    Ypred = np.zeros(num_test, dtype = self.y_train.dtype)
    # loop over all test rows
    for i in range(num_test):
      #L1
      distances = np.sum(np.abs(self.X_train - X[i,:]), axis = 1)
      # get the index with smallest distance
      min_index = np.argmin(distances)
      # predict the label of the nearest example
      Ypred[i] = self.y_train[min_index]
    return Ypred
nn = NearestNeighbor()
nn.train(X_train, y_train)
y_test_predict = nn.predict(X_test)
print(f'accuracy: {np.mean(y_test_predict == y_test)}')
```

L2 distance/norm: euclidean distance between two vectors. Each pixelwise difference is squared, summed up and finally the square root is taken:

$$d_2(A, B) = \sqrt{\sum_i (A_i - B_i)^2}$$

```
# new distance L2: we could leave out np.sqrt and get the same final output (monotonic function)
distances = np.sqrt(np.sum(np.square(self.X_train - X[i,:]), axis = 1))
```

Difference L1 , L2 - L2 unforgiving: prefers many medium disagreements to one big one.

k-Nearest Neighbor Classifier

Find top k closets images and take "majority vote" of labels. Higher k-values have a smoothing effect and make classifier more resistant to outliers. - Training is very fast - Predicting is slow since it compares x_test to each x_training - In practice we want efficient predictions! - Space inefficient

```
class KNearestNeighbor():
  """ a kNN classifier with L2 distance """

  def __init__(self):
    pass

  def train(self, X, y):
    """ same as before """

  def predit(self, X, k = 1):
    """ returns output of methods: compute_distances and predict_labels"""

    return self.predict_labels(dists, k = k)

  def compute_distacnes(self, X):
    """ L2 Distance: for each image in test set X  calculate the L2 distance to each images in X_train. Return a matrix dists where each row is a test image and each column the L2 of corresponding train image"""

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
```

```
    dists = np.zeros((num_test, num_train))

    for i in range(num_test):
      dists[i, :] = np.sqrt(np.sum(np.square(
                              self.X_train - X[i,:]),
                              axis = 1))
    return dists

  def predict_labels(self, dists, k =1):
    """Returns: array with predicted labels """

    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)

    for i in range(num_test):

      closest_y = []
      # get the k indices with smallest distances
      min_indices = np.argsort(dists[i,:])[:k]
      closest_y = np.bincount(self.y_train[min_indices])
      # predict the label of the nearest example
      y_pred[i] = np.argmax(closest_y)
    return y_pred
```

- We could speed up with broadcasting rules (see lecture notes)
- gray regions in visualization of results are caused by ties in majority vote

k-fold Cross Validation (what k to chose?)

- Hyperparameters = choiches needed to be made (distances, k, etc.)
- Try out different hyperparameters and use best
- We further take a small part of training set (e.g. 1000) as validation set which is used as a "**fake test set**" to tune the hyperparameter

```
X_val = X_train[:1000, :]
y_val = y_train[:1000]

X_train = X_train[1000:, :]
y_train = y_train[1000:]

validation_accuracies = []
for k in [1, 3, 5, 10, 20, 50, 100]:
  # use a particular value of k and evaluation on validation data
  nn = NearestNeighbor()
  nn.train(X_train, y_train)
  # here we assume a modified NearestNeighbor class that can
  # take a k as input
  y_val_predict = nn.predict(X_val, k = k)
  acc = np.mean(y_val_predict == y_val)
  print('accuracy: %f' % (acc,))
  # keep track of what works on the validation set
  validation_accuracies.append((k, acc))
```

When training data is smaller or computation is not heavy

- iterate over different validiation sets and average the performance across these
- 5-fold: split training into 5 parts, iterate 5 times while changing validation part
- In practice it is cleaner to no include the validation set in the final training of the model after determining the best hyperparameters

```
# k-fold cross validation
num_folds = 5
k_choices = [1, 3, 5, 7, 9, 10, 12, 15, 18, 20, 50, 100]
X_train_folds = []
y_train_folds = []
# Split up the training data into folds. After splitting,
# X_train_folds and y_train_folds should each be lists of
# length num_folds, where y_train_folds[i] is the label
# vector for the points in X_train_folds[i]
X_train_folds = np.split(X_train, [1000, 2000, 3000, 4000])
y_train_folds = np.split(y_train, [1000, 2000, 3000, 4000])
# A dictionary holding the accuracies for different values of
# k that we find when running cross-validation. After running
# cross-validation, k_to_accuracies[k] should be a list of
# length num_folds giving the different accuracy values that

# we found when using that value of k.
k_to_accuracies = {}
# We perform k-fold cross validation to find the best value of k.
# For each possible value of k, run the k-nearest-neighbor algorithm
# num_folds times, where in each case you use all but one of the folds
# as training data and the last fold as a validation set. Store the
# accuracies for all fold and all values of k in the k_to_accuracies
# dictionary.

for k in k_choices:

  k_to_accuracies[k] = []
  classifier = KNearestNeighbor()
  for i in range(num_folds):
    #we use ith fold as validation set
    X_cv_training = np.concatenate([x for k, x in
    enumerate(X_train_folds) if k!=i], axis=0)

    y_cv_training = np.concatenate([x for k, x in
      enumerate(y_train_folds) if k!=i], axis=0)

    classifier.train(X_cv_training, y_cv_training)
    dists = classifier.compute_distances_one_loop(X_train_folds[i])
    y_test_pred = classifier.predict_labels(dists, k=k)
    k_to_accuracies[k].append(np.mean(y_train_folds[i] == y_test_pred))
```

- in practice on may want to avoid cross-validation in favor of single validation split (computationally expensive)
- Size of single split of training data depends how many hyperparameters
- For example if the number of hyperparameters is large you may prefer to use bigger validation splits. If the number of examples in the validation set is small (perhaps only a few hundred or so), it is safer to use cross-validation. Typical number of folds you can see in practice would be 3-fold, 5-fold or 10-fold cross-validation.

Neural Networks (Lecture 2)

Linear Classificaiton of Images

1. score function: maps raw data to class scores
2. loss function: quantifies agreement between predicted and ground truth

$f(x_i, W, b) = Wx_i + b$

Parameters:

- $x_i = D * 1$ vector $D = 32 * 32 * 3$
- $W = K * D$ weight matrix where K is number of classes
- $b = K * 1$ (bias: influences output without interacting with data)

Note:

- Each single matrix multiplication Wx_i evaluates 10 separate classifiers in parallel (each row of W is a classifier)
  - provides 10 weighted sum of all image pixels: vector of class scores.
- After training we only need to save the parameters
- Prediction is just a single matrix multiplication and addition
- The weights can "like" or "dislike" certain colors at certain positions in the images
  - "Ship" classifier may have a lot of + weights across blue channels and negative weights in red/green channels: presence of these colors decrease score of ship
- Each image can be interpretated as point in D (3072) vector space
  - Each class score is a linear function over this space, so changing weights of a row in W will rotate linear function in different directions
  - Bias allows to move line up/down/right/left and lines are not forces to go through origin
- Other interpretation of W that each row is a prototpye of class (template matching)
- Often bias in incorporated into weight matrix and x is excdented by a constant 1.
- Class scores can be thought as unnormalized log-probabilities

```
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
```

Loss Function /Cost Function

Measures the **unhappiness** with the output. Loss is high with poor classifying the training data

Cross-Entropy Loss of Softmax Classifier

Softmax Classifier is generalization of binary logistic Regression classifier to multiple classes:

- gives normalized class probabilites
- is applied to every class score
- softmax function takes class scores z_j and squashes it to a vector of values between zero and one which sum to one (= probabilites). This can be interpreted as confidence in each class

$\frac{e^{z_j}}{\sum_k e^{z_k}}$

Cross-entropy loss $L_i = -log(\frac{e^z_{y_i}}{\sum_j e^{z_j}}) = -z_{y_i} + log \sum_j e^{z_j} + \lambda \sum_k \sum_l W^2_{k,l}$

with z_j being the j-th element of vector of class scores z and z_{y_i} score of true class. So we just take $-log()$ of the softmax ouput for **true class logit** and ignore the other logits/softmax ouputs. The *higher* the softmax output of true class, the *lower* the loss.

Full loss of dataset is the **mean** of L_i over N training examples: $L = \frac{1}{N} \sum_i L_i$

Numberic stability: we subract our class scorse by the max score in vector, therefore largest value is zero

```
def softmax_loss_vectorized(W, X, y):
  """
  Softmax loss function, vectorized version.
  """
  # Initialize the loss
  loss = 0.0
  # Compute the softmax loss
  num_train = X.shape[0]
  f = X.dot(W)
  # max of every sample
  f -= np.max(f, axis=1, keepdims=True)
  sum_f = np.sum(np.exp(f), axis=1, keepdims=True)
  p = np.exp(f)/sum_f
  loss = np.sum(-np.log(p[np.arange(num_train), y]))
  return loss
```

- lowest loss is zero.

Hinge Loss Multiclass Support Vector Machine loss (Max Margin loss)

Used in SVM. Multiclass Support Vector Machine loss. Takes class scores and summes max of all wrong class scores - class score of correct to determine the loss (end up with N losses):

$L_i = \sum_{j=y_i} max(0, W_j^T * x_i - W_{y_i}^T * x_i + \Delta)$

Delta is a hyperparameter which represents a margin that the correct class needs to be higher otherwise we collect loss.

Without regularization weight matrix can be multiplied by any number and the same loss would be uptained. Therefore, we add a regularization term (L2 norm) to the loss function (elementwise quadratic penalty over all parameters):

$R(W) = \sum_k \sum_l W_{k,l}^2$

Giving the full loss of $L = \frac{1}{N} \sum_i L_i + \lambda R(W)$

L2-norm can improve generalization because no single parameter should have large influence. Intuitively, this is because the weights in w2 are smaller and more diffuse. Since the L2 penalty prefers smaller and more diffuse weight vectors, the final classifier is encouraged to take into account all input dimensions to small amounts rather than a few input dimensions and very strongly (e.g. [1,0,0,0] vs [0.25, 0.25, 0.25, 0.25]). With the regularization term we can never have a loss of zero, only if we set all W to zero.

- Setting Δ is more or less meaningless because weights can shrink or stretch diffferences in margin.
- Therefore only λ reflects the real tradeoff between data loss and regularization loss
- In practice, however, if the data is noisy or not perfectly separable, a too-large Δ can force the model to memorize the data (pushing scores very far apart), leading to overfitting.

Comparison of both Loss

- They are comparable, with the main difference that Softmax loss is never happy

Moreover, in the limit where the weights go towards tiny numbers due to very strong regularization strength λ, the output probabilities would be near uniform. Hence, the probabilities computed by the Softmax Classifier are better thought of as confidences where, similar to the SVM, the ordering of the scores is interpretable, but the absolute numbers (or their differences) technically are not. n other words, the Softmax Classifier is never fully happy with the scores it produces: the correct class could always have a higher probability and the incorrect classes always a lower probability and the loss would always get better. However, the SVM is happy once the margins are satisfied and it does not micromanage the exact scores beyond this constraint. This can intuitively be thought of as a feature: For example, a car classifier which is likely spending most of its "effort" on the difficult problem of separating cars from trucks should not be influenced by the frog examples, which it already assigns very low scores to, and which likely cluster around a completely different side of the data cloud.

Optimization

- Finding parameters W that minimizes loss function
- This is done using gradient descend in the weight/parameter space
- Mathematically guaranteed direction of deepest descent = gradient of the loss function
- gradient is generalization of slope for functions that take vector of numbers and is a vector of slopes (partial derivatives) for each dimension.
- Two calculations: numerical gradient (slow but easy), analytic gradients (fast but error prone)
- The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step.
- Step size = learning rate (hyperparameter)
- Linear complecity in number of parameters: for single step D evaluations

Gradient Check

Compute analytic gradient (faster) and compare to numerical gradien

Gradient Descent

is procedure of computing gradient of loss function and repeatedly evaluating gradient and updating parameter

Mini-batch gradient descent

Computing loss function only for batch (e.g. 256) to perform weight updates. This works sind data points are often correlated.

- Stochastic Gradient Descent (SGD): extreme case with only 1 example

- Size of batch is a hyperparameter, set with crossvalidation but often a number to the power of 2 because faster in vectorization

Neural Network

A single neuron can be trained as Binary SVM Classifier or Binary Softmax Classifier

Use **normalization** (don't touch training data):

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Activation Functions

Sigmoid Function drawbacks:

- Sigmoid saturate and kill gradients: If saturated (close to 0 or 1) network barely learns during backprogragtion
- Sigmoid ouput are not zero-centered: (Page 35)

Tanh (squashes values to -1:1):

- is zero-centered but same problem with saturation

Rectified Linear Unit (ReLUs):

$f(x) = max(0, x)$

- fast convergence in gradient descent
- easy implementation treshholding a matrix at zero
- but can be fragile during training and die (if learning rate is set too high)
- **Leaky ReLUs** helps against dying (has a small negative slope below x = zero)
- **Maxout neuron** combined best of ReLU and Leaky ReLU but doubles number of parameters

NN Architectures

- A NN with one hidden layer can represent any continous function but this is of no practicle relevance
- Non-convex function ### Layer-wise organization
- Input layer is not counted towards arichtecture: so not when taking about layers and not when taking about untis/neurons
- Input layer don't have weights or biases but output layer does
- For 2-layer NN(4 units + 2 output) with 3 inputs we have: $3 * 4 + 4 * 2 = 20$ weights and 1 bias parameter for each unit/neuron. Total = 26 learnable parameters

Feed-Foward Computation

The forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function. Matrix form of parameter (e.g. Fully connected 3 * 4 * 4 * 1, with first as input layer):

- Input layer 3 * 1
- First hidden layer matrix of 4 * 3, where each row are the 3 weights of a neuron. This allows for easy multiplication from left to right
- First hidden layer has a bias vector of 4 * 1
- Second hidden layer: 4 * 4 matrix
- Output layer: 1 * 4

```
# forward-pass of a 3-layer neural network:
# activation function (use sigmoid)
f = lambda x: 1.0/(1.0 + np.exp(-x))
# random input vector of three numbers (3x1)
x = np.random.randn(3, 1)
# calculate first hidden layer activations (4x1)
h1 = f(np.dot(W1, x) + b1)
# calculate second hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)
# output neuron (1x1)
out = np.dot(W3, h2) + b3
```

- output layer has no activation function
- x in the input layer could be a batch of data where each example is a column vector and would be evaluted in parallel
- In practice, it is always better to use these methods to control overfitting instead of the number of neurons.
  - Because smaller NN are harder to train with graident descent. Local minima are increased with larger NN and of more value.

  - On the other hand, if you train a large network you'll start to find many different solutions, but the variance in the final achieved loss will be much smaller. In other words, all solutions are about equally as good, and rely less on the luck of random initialization.

Optimization with Backpropagation

Computing gradients through recursive application of **chain rule**.

- we get gradients for all parameters: weight matrix and bias vector
- to do

Keras

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
layers.Dense(500, activation="relu", input_shape=(784,)),
layers.Dense(50, activation="relu"),
layers.Dense(10, activation="softmax")
])
```

- Fully connect hidden layer with 500 units, each unit takes input from all 784 units in previous layer
- Fully connected hidden layer with 50 units, each unit takes input from all 500 units
- Each unit outpus one value everytime!
- Ouput layer: 10 units softmax

```
model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.summary()

# train model with:
history = model.fit(X_train, y_train_cat, epochs=5, batch_size=128)

# test on new data
test_loss, test_acc = model.evaluate(X_test, y_test_cat)
print(f"test_acc: {test_acc}")
```

- Batch: During each epoch (one full pass through the data), the model divides the data into mini-batches of 128 examples. (ca. 60000 /128 = 469 batches)
- For each batch: forward pass, compute loss, backward pass, update weights
  - After all 469 batches, one epoche is complete

Sequential adding of layers:

```
model = tf.keras.Sequential()
# From Input to first hidden layer
model.add(tf.keras.layers.Dense(100, activation= "relu",
                                input_shape=(2,)))
# From first hidden layer to output layer
model.add(tf.keras.layers.Dense(3, activation="softmax"))
```

Compile():

- Compiling the Keras model calls the backend Tensorflow and binds the optimizer, loss function, and other parameters required before the model can be run on any input data.

Training and Optimizing

Preprocessing a data matrix x with N x D.

Mean Subtraction (Zero-Centered)

Subtracting mean of all features. Centering data around origin

```
X -= np.mean(X, axis = 0)

# images we could do it for each channel seperatly
X -= np.mean(X, axis = (0,1,2))
```

Normalization

Features are same scale. Divide by standard deviation after zero-centered

PCA and Whitening

First centered. Next covariance matrix (tell us about correlation structure). Then we can compute the SD factorization of the data covariance matrix

Exercise

Simple Celsisus to fahrenheit fun part

```
input = tf.keras.layers.Input((1,))
l0 = tf.keras.layers.Dense(units=4)
l1 = tf.keras.layers.Dense(units=4)
l2 = tf.keras.layers.Dense(units=1)
model = tf.keras.Sequential([input, l0, l1, l2])
model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(0.1))
model.fit(celsius_q, fahrenheit_a, epochs=500, verbose=False)
print("Finished training the model")
print("Model predicts that 100 degrees Celsius is: {} degrees Fahrenheit".format(model.predict(np.array([100.0]))))
print("These are the l0 variables: {}".format(l0.get_weights()))
print("These are the l1 variables: {}".format(l1.get_weights()))
print("These are the l2 variables: {}".format(l2.get_weights()))
```

Linear Classifier

```
#Train a Linear Classifier

# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in range(200):

  # evaluate class scores, [N x K]
  scores = np.dot(X, W) + b

  # compute the class probabilities
  exp_scores = np.exp(scores)
  # [N x K]
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

  # compute the loss: average cross-entropy loss and regularization
  correct_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(correct_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W)
  loss = data_loss + reg_loss
  if i % 10 == 0:
    print("iteration %d: loss %f" % (i, loss))

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples

  # backpropate the gradient to the parameters (W,b)
  dW = np.dot(X.T, dscores)
  db = np.sum(dscores, axis=0, keepdims=True)

  dW += reg*W # regularization gradient

  # perform a parameter update
  W += -step_size * dW
  b += -step_size * db
```

NN

- broadcasting of bias vector b:np.dot gives 300x100 matrix and bias is 100x1. Broadcasting give us 300x100 by repeating the vector.
- Input values can be bound from 0 to 1: e.g. for greyscale divide by 255, or use normalization
- One hot encoding of labels

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in range(10000):

  # evaluate class scores, [N x K]
  hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
  scores = np.dot(hidden_layer, W2) + b2

  # compute the class probabilities
  exp_scores = np.exp(scores)
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

  # compute the loss: average cross-entropy loss and regularization
  correct_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(correct_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
  loss = data_loss + reg_loss
  if i % 1000 == 0:
    print("iteration %d: loss %f" % (i, loss))

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples

  # backpropagate the gradient to the parameters
  # first backprop into parameters W2 and b2
  dW2 = np.dot(hidden_layer.T, dscores)
  db2 = np.sum(dscores, axis=0, keepdims=True)
  # next backprop into hidden layer
  dhidden = np.dot(dscores, W2.T)
  # backprop the ReLU non-linearity
  dhidden[hidden_layer <= 0] = 0
  # finally into W,b
  dW = np.dot(X.T, dhidden)
  db = np.sum(dhidden, axis=0, keepdims=True)

  # add regularization gradient contribution
  dW2 += reg * W2
  dW += reg * W

  # perform a parameter update
  W += -step_size * dW
  b += -step_size * db
  W2 += -step_size * dW2
  b2 += -step_size * db2
```