0.1 Linear classification and kNN

Nearest neighbor: Train, O(1): learn the training dataset; Predict, O(n): for each test image, find closest train image, predict label of nearest image; Hyperparameters: K neighbors and distance metric (L1, L2)

0.1.1 Linear classification

$y = WX + b$, defined by loss function to select W • accuracy not a suitable loss function because derivative is 0 everywhere
SVM loss ("hinge loss"): Hyperparameters: Δ and whether to square the hinge loss. Hinge loss is not unique to scaling: In a zero loss case, W and λW result in same loss. Note if we regularize, using Δ as a hyperparameter is redundant and can be set to 1.

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + \Delta & \text{else} \end{cases} = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \qquad L = \frac{1}{n} L_i$$

Softmax loss ("cross-entropy loss"): At initialization, all the s_i's will be approximately equal, so loss will be $\approx -log(1/C) = log(C)$ • Numerical stability is a concern when taking exponents. "Normalization trick" multiplies the top and bottom by $\exp - \max_j f_j$ • scaling the weights changes steepness of sigmoid function, therefore the loss is not invariant to scaling, however labeling under perfect separation may be invariant to scaling.

$$P(Y = k \mid X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \qquad L_i = -\log P(Y = k \mid X = x_i) = -\log \frac{e^{s_k}}{\sum_j e^{s_j}}$$

0.2 Optimization

```
# Stochastic Gradient Descent (SGD)
data_batch = sample_training_data(data, 256) # sample 256 examples
weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
weights += -step_size * weights_grad # perform parameter updates
# Problem: move in one direction more than another, local min/max, noisy; solve by building up "velocity" as running mean
# SGD+Momentum
vx = rho * vx + weights_grad # initialize vx to 0
weights += -step_size * vx
# Nesterov momentum: more advanced, uses gradient of where we're headed
vx = rho * vx - alpha * evaluate_gradient(loss_fun, data_batch, weights + rho * vx)
weights += -step_size * vx
# AdaGrad: add element-wise scaling of gradient; progress damped along steep directions, accelerated along flat directions
grad_squared += weights_grad * weights_grad
weights += -step_size * weights_grad / (np.sqrt(grad_squared) + 1e-7)
# RMSProp: addresses problem that step size decays to zero, solution: make it leaky!
grad_squared = decay_rate * grad_squared + (1 - decay_rate) * weights_grad * weights_grad
weights += -step_size * weights_grad / (np.sqrt(grad_squared) + 1e-7)
# Adam: can we combine momentum with element-wise scaling? (basically Momentum + RMSProp)?
first_moment = beta1 * first_moment + (1 - beta1) * weights_grad # calculate momentum, first_moment initialized to 0
second_moment = beta2 * second_moment + (1 - beta2) * weights_grad * weights_grad # RMSProp, second_moment initialized to 0
first_unbias = first_moment / (1 - beta ** t) # bias correction, t is the number of the iterations
second_unbias = second_moment / (1 - beta2**t) # bias correction needed b/c moments initialized to 0
weights += -step_size * weights_grad / (np.sqrt(grad_squared) + 1e-7) # RMSProp
```

Learning rate (aka step-size) is a hyperparameter that can be learned. It can also change over time: • Step decay: reduce learning rate at a few fixed points over the training • Cosine decay: $\alpha_t = 1/2 * \alpha_0 * (1 + \cos t\pi/T)$ • Linear decay: $\alpha_t = \alpha_0(1 - t/T)$

0.3 Backpropagation

If we can compute $\partial L/\partial W_i$ then we can learn the weights W_i through gradient descent
Matrix multiplication: $Y = WX$ for $W \in \mathbb{R}^{D \times M}, X \in \mathbb{R}^{N \times D}, Y \in \mathbb{R}^{N \times M}$

- $\partial L/\partial X = (\partial L/\partial y)W^T$ for $\partial L/\partial Y \in \mathbb{R}^{N \times M}, \partial L/\partial X \in \mathbb{R}^{N \times D}$
- $\partial L/\partial W = X^T(\partial L/\partial Y)$ for $\partial L/\partial W \in \mathbb{R}^{D \times M}$

Useful backprop code:

```
# AFFINE
x_reshape = np.reshape(x, (x.shape[0], -1)) # Preprocessing:
    reshape the image data into rows
out = np.dot(x_reshape, w) + b # Affine transformation
dx = np.dot(dout, w.T) # dL / dX = dL/dY * dY/dX = dout *
    dY/dX and dY/dx = w.T
dx = np.reshape(dx, x.shape)
x_2d = np.reshape(x, (x.shape[0], -1))
dw = np.dot(x_2d.T, dout) # dL / dW = dL/dY * dY/dW = dout *
    dY/dW and dY/dW = x.T
db = np.dot(dout.T, np.ones(dx.shape[0])) # dL / db = dL/dY
    * dY/db = dout * dY/db = 1 in affine function
# RELU
out, dx = np.maximum(0, x), (x >= 0)*1 * dout
# SOFTMAX LOSS
num_train, num_class = x.shape
x -= np.max(x, axis=1, keepdims=True) # Normalization trick
x_rightclass = x[np.arange(num_train), y]
L = np.log(np.sum(np.exp(x), axis=1)) - x_rightclass #
    implement softmax calculation on each row
loss = np.mean(L) # collapse to calculate loss
dx = np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True) #
    calculate gradient
dx[np.arange(num_train), y] -= 1
dx /= num_train
# SVM LOSS
num_train = x.shape[0]
```

```
x_rightclass = x[np.arange(num_train), y]
L = np.maximum(0, x - np.matrix(x_rightclass).T + 1)
    #implement svm loss
L[np.arange(num_train), y] = 0 # zero-out cells where j ==
    y_i before collapsing
loss = np.sum(L) / num_train
indicator = np.zeros(x.shape) # make a matrix of indicators
    if L>0
indicator[L > 0] = 1
coef_yi = np.sum(indicator, axis=1) #calculate the
    coefficient for when j=y_i
indicator[np.arange(num_train), y] = -coef_yi
dx = indicator / num_train # divide by N to get average
# BATCHNORM
sample_mean = np.mean(x, axis=0, keepdims=True)
sample_var = np.var(x, axis=0, keepdims=True)
running_mean = momentum * running_mean + (1 - momentum) *
    sample_mean
running_var = momentum * running_var + (1 - momentum) *
    sample_var
xnorm = (x - sample_mean) / np.sqrt(sample_var + eps)
out = gamma * xnorm + beta
dxhat = dout * gamma # START OF BACKWARD
dvar = (-1/2) * (dxhat * (x - sample_mean) * (sample_var +
    eps)**(-3/2)).sum(axis=0, keepdims=True)
dmean = (- dxhat / np.sqrt(sample_var + eps)).sum(axis=0,
    keepdims=True)
dx = dxhat * (1/np.sqrt(sample_var + eps)) +dvar * (2/n) *
    (x - sample_mean) +dmean * (1/n)
dx = (1/n) * (1/np.sqrt(sample_var + eps)) *(n * dxhat -
    dxhat.sum(axis=0) - xhat * (dxhat * xhat).sum(axis=0))
    # backward alternative
dgamma = (dout * xhat).sum(axis=0)
dbeta = dout.sum(axis=0)
# DROPOUT (inverted)
out = x * (np.random.rand(*x.shape) < p) / p
dx = dout * mask
# CONVOLUTIONAL FORWARD PASS
N, C, H, W = x.shape
F, CC, HH, WW = w.shape
N, C, Hp, Wp = x_padded.shape
H_out = 1 + (Hp - HH) // stride
W_out = 1 + (Wp - WW) // stride
# initialize out matrix
out = np.zeros([N, F, H_out, W_out])
# for each data point, each window of image, and each
    filter, convolute
for i in range(N):
  x_i = x_padded[i]
  # initialize indexes for out matrix
  hh_i = -1
  ww_i = -1
  # Bottom convolution at Hp - HH + 1
  for h_i in range(0, Hp - HH + 1, stride):
    hh_i += 1
    # Right-most convolution at Wp - WW + 1
    for w_i in range(0, Wp - WW + 1, stride):
      ww_i += 1
      x_i_hw = x_i[:,h_i:h_i+HH, w_i:w_i+WW]
      for f in range(F):
        # Convolution is element-wise product, summed, plus
            bias
        conv = np.sum(x_i_hw * w[f]) + b[f]
        out[i, f, hh_i, ww_i] = conv
    ww_i = -1
```

```
# CONVOLUTION BACKWARD PASS
N, F, H_out, W_out = dout.shape
N, C, Hp, Wp = x_padded.shape
db, dw, dx_padded = np.zeros(b.shape), np.zeros(w.shape),
    dx_padded = np.zeros(x_padded.shape)
for f in range(F): # get db, dw
  db[f] = np.sum(dout[:,f,:,:])
  for c in range(C):
    for h_i in range(HH):
      for w_i in range(WW):
        # slice x to get values hit by W_ii. Note
            H_out*stride = Hp-HH+1
        doutdW_ii = x_padded[:, c, h_i:h_i+Hp-HH+1:stride,
            w_i:w_i+Wp-WW+1:stride]
        dw[f,c,h_i,w_i] = np.sum(dout[:,f,:,:] * doutdW_ii)
for i in range(N): # get dx
  for f in range(F):
    # loop over each filter position
    for hout_i in range(H_out):
      for wout_i in range(W_out):
        h_i = hout_i*stride
        w_i = wout_i*stride
        dx_padded[i,:,h_i:h_i+HH,w_i:w_i+WW] +=
            dout[i,f,hout_i,wout_i] * w[f,:,:,:]
# remove padding from dx_padded
dx = dx_padded[:,:,pad:pad+H,pad:pad+W]
# MAX POOL FORWARD
N, C, H, W = x.shape
H_out = 1 + (H - pool_height) // stride
W_out = 1 + (W - pool_width) // stride
out = np.zeros([N, C, H_out, W_out]) # initialize
for h_i in range(H_out):
  for w_i in range(W_out):
    x_sub = x[:,:,h_i*stride:h_i*stride+pool_height,
        w_i*stride:w_i*stride+pool_width]
    out[:,:,h_i,w_i] = np.max(x_sub, axis=(2,3))
# MAX POOL BACKWARD
N, C, H_out, W_out = dout.shape
N, C, H, W = x.shape
HH, WW = pool_height, pool_width
dx = np.zeros(x.shape)
for i in range(N):
  for c in range(C):
    # loop over each pooling range
    for hout_i in range(H_out):
      for wout_i in range(W_out):
        h_i = hout_i*stride
        w_i = wout_i*stride
        # find max index in pooling range
        x_pool = x[i,c,h_i:h_i+HH,w_i:w_i+WW]
        x_pool = (x_pool ==
            x_pool.max(keepdims=True)).astype(int)
        # add to gradient after multiplying by proper dout
        dx[i,c,h_i:h_i+HH,w_i:w_i+WW] =
            dout[i,c,hout_i,wout_i] * x_pool
# SPATIAL BATCHNORM
N, C, H, W = x.shape
x_flat = x.transpose(0,2,3,1).reshape(N*H*W,C)
out_flat, cache = batchnorm_forward(x_flat, gamma, beta,
    bn_param)
out = out_flat.reshape(N, H, W, C).transpose(0,3,1,2)
N, C, H, W = dout.shape # START BACKWARD
dout_flat = dout.transpose(0,2,3,1).reshape(N*H*W, C)
dx_flat, dgamma, dbeta = batchnorm_backward(dout_flat, cache)
dx = dx_flat.reshape(N, H, W, C).transpose(0,3,1,2)
```

0.4 Neural Networks

Activation functions

- **Sigmoid:** $1/(1+e^{-x})$. Two major drawbacks, i) kills gradients: gradients near either tail are almost zero, so it cannot pass through upstream gradients ii) not zero-centered: can produce zig-sagging dynamics in gradient updates since it will lead to all pos. or all neg. gradient updates
- **Tanh:** $tanh(x)$. Zero-centered, but still kills gradients
- **ReLU:** $max(0, x)$. Most popular for its simple compute. ReLU units can "die" during training. Large gradient can cause weights to update so that the neuron will not activate again.
- **Leaky ReLU:** $max(\gamma x, x), \gamma$ small. Attempts to fix "dying ReLU". **Maxout** further generalizes leaky ReLU: $\max(w_1^T x+b_1, w_2^T x+b_2)$

- **ELU:** $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

Initialization: Problem with initializing weights to 0: if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. Solution is to initialize to random small inputs • Xavier initialization for tanh divides by $\sqrt{D_{in}}$ to preserve distribution of initialized values (for conv layers, $D_{in} = K \times K \times C$) • Kaiming initialization for relu multiplies Xavier initialization by $\sqrt{2}$ • Initializing to 0 is not a concern for SVM

0.4.1 Regularization

L1, L2, elastic net • **Max norm constraints:** clamp the weight vector of each neuron to satisfy $\|W\|_2 < c$ • **Dropout:** keep each neuron active with probability p. During testing, no dropout is applied, but all weights are scaled by p. In practice "inverted dropout" is implemented to improve test-time prediction (i.e., divide by p in ReLU step during training)

0.5 Convolutions

Feature representation: color histogram • histogram of oriented gradients • bag of words
Convolutional neural nets (CNN) learn these features so they don't need to be defined ahead of time. Further, CNNs take advantage of data structure.

- Convolution filters are of dimension C2xC1xHFxWF: C2 out-channels, C1 in-channels (same as num channels in input layer), HF filter height, WF filter width. They also include a C2-dimensional bias vector.
- Output activation layer size, assuming square input, HxH, square filter, FxF, pad, P, and stride, S: $(H + 2P - F)/S + 1$
- Successive convolution adds $F - 1$ to the receptive field size; with L layers, the receptive field size is $1 + L * (K - 1)$
- Number of parameters: $F^2 * C1 * C2 + C2$

0.5.1 Pooling

Makes representations smaller and more manageable • Operates on each activation map independently • No learnable parameters • introduces spatial invariance • Filter size, FxF, stride, S, produces output $H_2 = (H_1 - F)/S + 1$, $W_2 = (W_1 - F)/S + 1$

0.5.2 Batch normalization

Usually inserted after fully connected or convolutional layers • allows higher learning rates and faster convergence • more robust to initialization • as with all normalization techniques, normalization parameters are determined from the training data only and applied to train/test/val • Fully connected nets, $x \in \mathbb{R}^{N \times D} \implies (\mu, \sigma, \gamma, \beta) \in \mathbb{R}^{1 \times D}$ • Convolutional nets $x \in \mathbb{R}^{N \times C \times H \times W} \implies (\mu, \sigma, \gamma, \beta) \in \mathbb{R}^{1 \times C \times 1 \times 1}$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \ (\gamma, \beta \text{ learned}), \quad \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}, \quad \mu_j = \frac{1}{n} \sum_i x_{i,j}, \quad \sigma_j^2 = \frac{1}{n} \sum_i (x_{i,j} - \mu_j)^2$$

Test time: Mean and variance estimates depend on each minibatch, so we need to use the running average of each (calculated during training) to use when testing the model.
Layer normalization: Fully connected nets, $x \in \mathbb{R}^{N \times D} \implies (\mu, \sigma) \in \mathbb{R}^{N \times 1}, (\gamma, \beta) \in \mathbb{R}^{1 \times D}$ • Convolutional nets $x \in \mathbb{R}^{N \times C \times H \times W} \implies (\mu, \sigma) \in \mathbb{R}^{N \times 1 \times 1 \times 1}, (\gamma, \beta) \in \mathbb{R}^{1 \times C \times 1 \times 1}$ • note this means that normalization can't be reversed by learned parameters, γ, β
Instance normalization: Convolutional nets $x \in \mathbb{R}^{N \times C \times H \times W} \implies (\mu, \sigma) \in \mathbb{R}^{N \times C \times 1 \times 1}, (\gamma, \beta) \in \mathbb{R}^{1 \times C \times 1 \times 1}$
Group normalization: Convolutional nets $x \in \mathbb{R}^{N \times C \times H \times W}$ and $C/g = G$ groups $\implies (\mu, \sigma) \in \mathbb{R}^{N \times G \times 1 \times 1}, (\gamma, \beta) \in \mathbb{R}^{1 \times C \times 1 \times 1}$

0.5.3 Architectures

VGGNet: Smaller and more convolution layers, leading to same receptive field, more activation layers, fewer learned parameters.
GoogLeNet: Apply parallel filter op.s (1x1, 3x3, 5x5) and pooling op., concatenate together channel-wise. Reduce depth with 1x1x64 convolutional layer first. Many fewer params, avoids expensive FC layers
ResNet: Deeper models should work better, but are harder to optimize. In ResNet, update $x \to F(x) \to y = ReLU(F(x)) \to H(y) \to z = ReLU(H(y))$, to $x \to F(x) \to y = ReLU(F(x)) \to H(y) \to z = ReLU(H(y) + x)$

0.5.4 Transfer learning

Freeze most layers of a trained CNN, reinitialize and train last layer, or last several layers • These last several layers can be used as inputs to other models, like what is used in object detection, and other more complex deep learning problems

0.5.5 Visualizing model

- Saliency maps: compute gradient of unnormalized class score wrt an image, take absolute value and max over RGB channels
- Gradient ascent: generate a synthetic image that maximally activates a neuron; i) initialize image to zeros, ii) forward image to get current scores, iii) backprop to get gradient w.r.t. image pixels, iv) update image. Add regularizer to image to make it more smooth. Can use same approach to visualize intermediate layers.

```
def compute_saliency_maps(X, y, model):
# Forward pass: compute scores and loss
scores = model(X)
scores_for_right_label = scores.gather(1, y.view(-1,
    1)).squeeze()

# loss is the sum of the probabilities for the correct label
loss = torch.sum(scores_for_right_label)

# backpropagate and get image gradient
loss.backward()
X_grad = X.grad
abs_X_grad = X_grad.abs()
saliency, max_indices = abs_X_grad.max(dim=1)
# FOOLING IMAGES
make_fooling_image(X, target_y, model):
while True:
  scores = model(X_fooling)
  prediction_label = scores.argmax()

  if prediction_label == target_y:
    return X_fooling

  scores[:, target_y].backward()
  g = X_fooling.grad
  g_norm = torch.norm(g,2)
  dX = learning_rate * g / g_norm
  with torch.no_grad():
    X_fooling += dX
  X_fooling.grad.zero_()
# CLASS VISUALIZATION
# get loss and backpropagate
scores = model(img)
loss = scores[:,target_y]
loss.backward()

# get gradient and ascend gradient
g = img.grad - 2 * l2_reg * img
dX = learning_rate * g
with torch.no_grad():
  img += dX
img.grad.zero_()
```

0.6 Object detection and segmentation

- Learnable upsampling: for image segmentation we want to downsize (for efficiency) and later upsize (since output is same HxW dims as input). Can upsize with transposed convolution, which takes a smaller, pooled input, and casts it onto a larger vector, with overlap defined by stride.
- Region proposals: for object detection, use model to find image regions that may contain objects, and run image classification CNNs on those. Problem: very slow to run independent image classifications over each region of interest. Update: pass image through convnet before cropping so you only do it once! Need to do ROI Align to do this, by sampling at regular points in each subregion using bilinear interpolation, then using a linear combination of features at neighboring grid cells. Further update: region proposals come from the CNN

0.7 Recurrent neural networks

- Backpropagation through chunks of the sequence instead of the whole sequence
- Many types: One-to-many (sentence generation), many-to-one (sentiment classification), many-to-many where $N_x = N_y$ (name entity recognition), many-to-many where $N_x \neq N_y$ (machine translation)
- Loss function, $\mathcal{L}$ is based on the loss at each time step. Gradient at time step, T, is $\partial\mathcal{L}^{(T)}/\partial W = \sum_{t=1}^{T} \partial\mathcal{L}^{(T)}/\partial W|_{(t)}$
- $\partial h_t/\partial h_{t-1} = tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$
- Attention allows for many-to-many scenarios with different encoding lengths. Allows us to change attention weights from each output $f_W(h_{t-1}, x_{t-1})$. Context vector becomes a linear combination of attention weights, instead of simply a final output, h_T. • permutation invariant, doesn't care about ordering of features.
- **Self attention:** composes queries, keys, and values (all $C \times H \times W$) to generate attention weights ($(H \times W) \times (H \times W)$) and output features